RAO'S INSTITUTE
OF
COMPUTER SCIENCE

R

PROGRAMMING

Prepared by
Shaik Tippu Sulthan
Assistant Professor
Department of Computer Science

ELECTIVE: MCA1750 PROGRAMMING USING 'R

UNIT-I:

Introduction, what is R, Basic Features of R, Design of the R System, Limitation of R, R Nuts & Bolts How to run R, R Sessions and Functions, Basic Math, Getting data in and out of R, Reader Packages, Variables, Data Types, Vectors, Conclusion, Advanced Data Structures, Data Frames, Lists, Matrices, Arrays, Classes.

UNIT-II:

R Programming Structures, Control Statements, Loops, - Looping Over Non-vector Sets,- If-Else, Arithmetic and Boolean Operators and values, Default Values for Argument, Return Values, Deciding Whether to explicitly call return- Returning Complex Objects, Functions are Objective, No Pointers in R, Recursion, A Quick sort Implementation-Extended Extended Example: A Binary Search Tree.

UNIT-III:

Doing Math and Simulation in R, Math Function, Extended Example Calculating Probability- Cumulative Sums and Products-Minima and Maxima- Calculus, Functions Fir Statistical Distribution, Sorting, Linear Algebra Operation on Vectors and Matrices, Extended Example: Vector cross Product- Extended Example: Finding Stationary Distribution of Markov Chains, Set Operation, Input /output, Accessing the Keyboard and Monitor, Reading and writer Files,

UNIT-IV:

Graphics, Creating Graphs, The Workhorse of R Base Graphics, the plot() Function – Customizing Graphs, Saving Graphs to Files.

TEXT BOOKS:

1. The Art of R Programming, A K Verma, Cengage Learning.
2. R for Everyone, Lander, Pearson
3. The Art of R Programming, Norman Matloff, No starch Press.

REFERENCE BOOKS:

1. R Cookbook, Paul Teetor, Oreilly.
2. R in Action, Rob Kabacoff, Manning
3. R Programming for Data Science, Roger D. Peng Lean Publishing.

STATISTICS WITH R PROGRAMMING

UNIT-1

1.1 Introduction

1.2 How to run R

1.3 R Sessions and Functions

1.4 Basic Math

1.5 Variables

1.6 Data Types

1.7 Vectors

1.8 Data Frames

1.9 Lists

1.10 Matrices

1.11 Arrays

1.1 INTRODUCTION

1.1. A. What Is R?

- R is a scripting language for statistical data manipulation and analysis.
- It was inspired by, and is mostly compatible with, the statistical language S developed by AT&T.
- The name S, obviously standing for statistics,was an allusion to another programming language developed at AT&T with a one-letter name, C.
- S later was sold to a small firm, which added a GUI interface and named the result S-Plus.
- R has become more popular than S/S-Plus, both because it's free and because more people are contributingto it.
- R is sometimes called "GNU S."

1.1. B. Why Use R for Your Statistical Work?

R is a scripting language is inexpensive and beautiful.
R - a scripting language which is

- a public-domain implementation of the widely-regarded S statistical language.
- comparable, and often superior, in power to commercial products in most senses
- available for Windows, Macs, Linux
- in addition to enabling statistical operations, it's a general programming language, so that you can automate your analyses and create new functions
- object-oriented and functional programming structure your data sets are saved between sessions, so you don't have to reload each time
- open-software nature means it's easy to get help from the user community, and lots of new functions get contributed by users, many of which are prominent statisticians

1.1. C. The advantages of R Language are:

- Clearer, more compact code.
- Potentially much faster execution speed.
- Less debugging (since you write less code).
- Easier transition to parallel programming.

1.2 How to Run R

R has two modes, interactive and batch. The former is the typical one used.

1.2. A. Running R in Interactive Mode

- You start R by typing "R" on the command line in Linux or on a Mac, or in a Windows Run window. You'llget a greeting, and then the R prompt, >.
- We can then execute R commands and execute own R code like R files with .r extension.
- The command> source("z.r")which would execute the contents of that file.

1.2. B. Running R in Batch Mode

- Sometimes it's preferable to automate the process of running R. For example, we may wish to run an R scriptthat generates a graph output file, and not have to bother with

manually running R. Here's how it could bedone. Consider the file z.r, which produces a histogram and saves it to a PDF file:

```
pdf("xh.pdf") # set graphical output file
hist(rnorm(100)) # generate 100 N(0,1) variates and plot their histogram
dev.off() # close the file
```

To run it automatically by simply typing

```
R CMD BATCH --vanilla <z.r
```

The –vanilla option tells R not to load any startup file information, and not to save any.

1.3 R Sessions and Functions

1.3.A. Sessions

Session in R is a workspace between start and end point in R console.

Session starts when R console is initiated. It end when we quit from the R console.

Example

Start R from our shell command line, and get the greeting message and the > prompt:
R : Copyright 2005, The R Foundation for Statistical Computing
Version 2.1.1 (2005-06-20), ISBN 3-900051-07-0
...
Type 'q()' to quit R.
>q()
Save workspace image? [y/n/c]: y
(Session Saved)

1.3. B. 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.

Programming Example

In the following example, define a function oddcount() while in R's interactive mode. Then call the function on a couple of test cases. The function is supposed tocount the number of odd numbers in its argument vector.

```
# comment: counts the number of odd integers in x
>oddcount<- function(x) {
+ k <- 0
+ for (n in x) {
+ if (n %% 2 == 1) k <- k+1
+ }
+ return(k)
+ }

>oddcount(c(1,3,5))
[1] 3

>oddcount(c(1,2,3,7,9))
[1] 4
```

Here is what happened: We first told R that we would define a function oddcount() of one argument x. Theleft brace demarcates the start of the body of the function. We wrote one R statement per line. Since wewere still in the body of the function, R reminded us of that by using + as its prompt1 instead of the usual

>. After we finally entered a right brace to end the function body, R resumed the > prompt.

Note that arguments in R functions are read-only, in that a copy of the argument is made to a local variable,and changes to the latter don't affect the original variable. Thus changes to the original variable are typicallymade by reassigning the return value of the function.

Types of Functions

Functions are two type, they are

1. Built-in function
2. User defined functions

1. 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
```

2. 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 in 1: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 in 1: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 in 1:5) {
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
```

1.4 Basic Math in R

1.4.1. R 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.

Arithmetic Operators in R

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulo Division
%/%	Integer Division

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
```

Examples in Arithmetic operators

```
# 1 plus 1
1+1
```

```
[1] 2
```

```
# 4 minus 3
4-3
```

```
[1] 1
```

```
# 14 divided by 10
14/10
```

```
[1] 1.4
```

```
# 10 multiplied by 5
10*5
```

```
[1] 50
```

```
# 3 squared
3^2
```

```
[1] 9
```

```
# 5 mod 2
5%%2
```

```
[1] 1
```

```
# 4 divided by 2 (integer division)
4%/%2
```

```
[1] 2
```

```
# log to the base e of 2
log(2)
```

```
[1] 0.6931472
```

```
# antilog of 2
exp(2)
```

```
[1] 7.389056
```

```
# log to base 2 of 3
```

```
log(3,2)
```
```
[1] 1.584963
```
```
# log to base 10 of 2
log10(2)
```
```
[1] 0.30103
```
```
# square root of 2
sqrt(2)
```
```
[1] 1.414214
```
```
# !5
factorial(4)
```
```
[1] 24
```
```
# largestinterger smaller than 2
floor(2)
```
```
[1] 2
```
```
# smallest integer greater than 6
ceiling(6)
```
```
[1] 6
```
```
# round 3.14159 to three digits
round(3.14159, digits=3)
```
```
[1] 3.142
```
```
# create 10 random digits between zero and 1 (from a uniform distribution)
runif(10)
```
```
 [1] 0.07613962 0.66543266 0.48379168 0.40593920 0.67715428 0.49170373
 [7] 0.62351598 0.19275859 0.48018351 0.34890640
```
```
# cosine of 3
cos(3)
```
```
[1] -0.9899925
```
```
# sine of 3
sin(3)
```
```
[1] 0.14112
```
```
# tangent of 3
tan(3)
```
```
[1] -0.1425465
```
```
# absolute value of -3
abs(-3)
```
```
[1] 3
```

1.4.2 R 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
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not 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

>y==16
[1] TRUE

>x !=5
[1] FALSE
```

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]11 5 17 7

>x-1# Scalar 1 is recycled to 1,1,1,1
[1]1 0 7 2

>x+c(1,2,3)
[1]3 3 11 4
Warning message:
In x +c(1,2,3):
longerobject length isnot a multiple of shorter object length
```

1.4.3 R 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  TRUE  TRUE FALSE

>x&y
[1] FALSE FALSE  FALSE  TRUE

>x&&y
[1] FALSE

>x|y
[1]  TRUE  TRUE FALSE  TRUE

> x||y
[1] TRUE
```

1.5 Variables

- A variable provides us with named storage that our programs can manipulate.
- A variable in R can store an atomic vector, group of atomic vectors or a combination of many Robjects.
- A valid variable name consists of alphabets, numbers, the dot(.) or underline characters.
- The variable name starts with a letter or the dot
- The variable name should not followed by a number.

Examples

Variable Name	Validity	Reason
var_name2.	Valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.
2var_name	Invalid	Starts with a number
.var_name , var.name	Valid	Can start with a dot(.) but the dot(.)should not be followed by a number.
.2var_name	Invalid	The starting dot is followed by a number making it invalid.
_var_name	Invalid	Starts with _ which is not valid

Variable Assignment

The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using **print()** or **cat()**function. The **cat()** function combines multiple items into a continuous print output.

```
# Assignment using equal operator.
var.1=c(0,1,2,3)
```

```
# Assignment using leftward operator.
var.2<-c("learn","R")

# Assignment using rightward operator.
c(TRUE,1)->var.3

print(var.1)
cat("var.1 is ",var.1,"\n")
cat("var.2 is ",var.2,"\n")
cat("var.3 is ",var.3,"\n")
```

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

```
[1] 0 1 2 3
var.1 is 0 1 2 3
var.2 is learn R
var.3 is 1 1
```

Note − The vector c(TRUE,1) has a mix of logical and numeric class. So logical class is coerced to numeric class making TRUE as 1.

Data Type of a Variable

In R, a variable itself is not declared of any data type, rather it gets the data type of the R - object assigned to it. So R is called a dynamically typed language, which means that we can change a variable's data type of the same variable again and again when using it in a program.

```
> x<-"Hello"
>class(x)
[1] "character"

> y<-34.5
>class(y)
[1] "numeric"
```

Finding Variables – ls() function

To know all the variables currently available in the workspace we use the **ls()** function. Also the ls() function can use patterns to match the variable names.

```
print(ls())
```

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

```
[1] "x"  "y"
```

Deleting Variables – rm() function

Variables can be deleted by using the **rm()** function. Below we delete the variable "x". On printing the value of the variable error is thrown.

```
> rm(x)
> x
```

```
Error: object 'x' not found
```

All the variables can be deleted by using the **rm()** and **ls()** function together.

```
rm(list =ls())
print(ls())
```

R Assignment Operators on Variables

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.

The <<− operator 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
```

1.6 Data Types

There are numerous data types in `R` that store various kinds of data. The four main types of data most likely to be used are `numeric`, `character` (string), `Date`/`POSIXct` (time-based) and `logical` (`TRUE`/`FALSE`).

The type of data contained in a variable is checked with the `class` function.

```
>class(x)

[1] "numeric"
```

1.6.1. Numeric Data

The most commonly used numeric data is `numeric`. This is similar to a `float` or `double` in other languages. It handles integers and decimals, both positive and negative, and, of course,

zero. A numeric value stored in a variable is automatically assumed to be `numeric`. Testing whether a variable is `numeric` is done with the function `is.numeric`.

```
>is.numeric(x)

[1] TRUE
```

Another important, if less frequently used, type is `integer`. As the name implies this is for whole numbers only, no decimals. To set an integer to a variable it is necessary to append the value with an `L`. As with checking for a `numeric`, the `is.integer` function is used.

```
>i<- 5L
>i

[1] 5

>is.integer(i)

[1] TRUE
```

Do note that, even though `i` is an `integer`, it will also pass a `numeric` check.

```
>is.numeric(i)

[1] TRUE
```

`R` nicely promotes `integer`s to `numeric` when needed. This is obvious when multiplying an `integer` by a `numeric`, but importantly it works when dividing an `integer` by another `integer`, resulting in a decimal number.

```
>class(4L)

[1] "integer"

>class(2.8)

[1] "numeric"
```

1.6.2. Character Data

The character data type is a string type data which is very common in statistical analysis and must be handled with care. `R` has two primary ways of handling character data: `character` and `factor`.

Example:

```
> x <- "data"
>x
[1] "data"

> y <- factor("data")
> y
[1] data
Levels: data
```

Notice that `x` contains the word “data” encapsulated in quotes, while `y` has the word “data” without quotes and a second line of information about the `levels` of `y`.

`Characters` are case sensitive, so “Data” is different from “data” or “DATA.”

To find the length of a `character` (or `numeric`) use the `nchar` function.

```
>nchar(x)
[1] 4

>nchar("hello")
[1] 5

>nchar(3)
[1] 1

>nchar(452)
[1] 3
```

This will not work for `factor` data.

Example:

```
>nchar(y)
Error: 'nchar()' requires a character vector
```

1.6.3. Dates

The "Date" data type dealing with dates and times. The most useful are `Date` and `POSIXct`.

`Date` stores just a date while `POSIXct` stores a date and time.

Both objects are actually represented as the number of days (`Date`) or seconds (`POSIXct`) since January 1, 1970.

Example:

```
> date1 <- as.Date("2012-06-28")
> date1
[1] "2012-06-28"

>class(date1)
[1] "Date"

>as.numeric(date1)
[1] 15519

> date2 <- as.POSIXct("2012-06-28 17:42")
> date2
[1] "2012-06-28 17:42:00 EDT"


>class(date2)
[1] "POSIXct" "POSIXt"

>as.numeric(date2)
[1] 1340919720
```

Using functions such as `as.numeric` or `as.Date` does not merely change the formatting of an object but actually changes the underlying type.

Example:

```
>class(date1)
[1] "Date"

>class(as.numeric(date1))
[1] "numeric"
```

1.6.4. Logical

`logical`s are a way of representing data that can be either `TRUE` or `FALSE`. Numerically, `TRUE` is the same as 1 and `FALSE` is the same as 0. So `TRUE` * 5 equals 5 while `FALSE` * 5 equals 0.

```
> TRUE * 5
[1] 5

> FALSE * 5
[1] 0
```

Similar to other types, `logical`s have their own test, using the `is.logical` function.

```
> k <- TRUE
>class(k)
[1] "logical"


>is.logical(k)
[1] TRUE
```

`R` provides `T` and `F` as shortcuts for `TRUE` and `FALSE`, respectively, but it is best practice not to use them, as they are simply variables storing the values `TRUE` and `FALSE` and can be overwritten, which can cause a great deal of frustration as seen in the following example.

```
>TRUE
[1] TRUE

> T
[1] TRUE

>class(T)
[1] "logical"

> T <- 7
> T
[1] 7

>class(T)
[1] "numeric"
```

`Logical` can result from comparing two numbers, or characters.

Example:

```
> # does 2 equal 3?
> 2 == 3

[1] FALSE

> # does 2 not equal three?
>2 != 3

[1] TRUE

> # is two less than three?
> 2 < 3

[1] TRUE

> # is two less than or equal to three?
> 2 <= 3

[1] TRUE
```

1.7 Vectors

- A `vector` is a collection of elements, all of the same type. For instance, `c(1, 3, 2, 1, 5)` is a `vector` consisting of the numbers 1, 3, 2, 1, 5, in that order.
- Similarly, `c("R", "Excel", "SAS", "Excel")` is a `vector` of the `character` elements "R," "Excel," "SAS" and "Excel."
- A `vector` cannot be of mixed type.
- `Vectors` play a crucial, and helpful, role in `R`.
- `Vectors` do not have a dimension, meaning there is no such thing as a column `vector` or row `vector`.
- These `vectors` are not like the mathematical `vector` where there is a difference between row and column orientation.
- The most common way to create a `vector` is with `c`. The "c" stands for combine because multiple elements are being combined into a `vector`.

Example:

```
> x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

1.7.1. Vector Operations

Now that we have a `vector` of the first ten numbers, we might want to multiply each element by 3. In `R` this is a simple operation using just the multiplication operator (`*`).

Example:

```
>x * 3
[1]  3  6  9 12 15 18 21 24 27 30
```

No loops are necessary. Addition, subtraction and division are just as easy. This also works for any number of operations.

Example:

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

> x - 3
[1] -2 -1  0  1  2  3  4  5  6  7

> x/4
[1] 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25 2.50

> x^2
[1]   1   4   9  16  25  36  49  64  81 100

>sqrt(x)
[1] 1.000 1.414 1.732 2.000 2.236 2.449 2.646 2.828 3.000 3.162
```

Earlier we created a `vector` of the first ten numbers using the `c` function, which creates a `vector`. A shortcut is the `:` operator, which generates a sequence of consecutive numbers, in either direction.

Example:

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

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

> -2:3
[1] -2 -1  0  1  2  3

> 5:-7
[1]  5  4  3  2  1  0 -1 -2 -3 -4 -5 -6 -7
```

Vector operations can be extended even further. Let's say we have two vectors of equal length. Each of the corresponding elements can be operated on together.

Example:

```
> # create two vectors of equal length
> x <- 1:10
> y <- -5:4
> # add them
> x + y

 [1] -4 -2  0  2  4  6  8 10 12 14

> # subtract them
> x - y
 [1] 6 6 6 6 6 6 6 6 6 6

> # multiply them
> x * y
 [1] -5 -8 -9 -8 -5 0 7 16 27 40

> # check the length of each
>length(x)
[1] 10

>length(y)
[1] 10

> # the length of them added together should be the same
>length(x + y)
[1] 10
```

- Things get a little more complicated when operating on two `vector`s of unequal length. The shorter `vector` gets recycled, that is, its elements are repeated, in order, until they have been matched up with every element of the longer `vector`.
- If the longer one is not a multiple of the shorter one, a warning is given.

Example:

```
> x + c(1, 2)
 [1]  2  4  4  6  6  8  8 10 10 12

> x + c(1, 2, 3)
Warning: longer object length is not a multiple of shorter object
```

```
length
 [1] 2  4  6  5  7  9  8 10 12 11
```

Comparisons also work on `vector`s. Here the result is a vector of the same length containing `TRUE` or `FALSE` for each element.

Example:

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

> x > y
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

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

To test whether all the resulting elements are `TRUE`, use the `all` function. Similarly, the `any` function checks whether any element is `TRUE`.

```
> x <- 10:1
> y <- -4:5
>any(x < y)
[1] TRUE

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

The `nchar` function also acts on each element of a `vector`.

Example:

```
> q <- c("Hockey", "Football", "Baseball", "Curling", "Rugby",
+        "Lacrosse", "Basketball", "Tennis", "Cricket", "Soccer")
>nchar(q)
 [1]  6  8  8  7  5  8 10  6  7  6

>nchar(y)
 [1] 2 2 2 2 1 1 1 1 1 1
```

Accessing individual elements of a `vector` is done using square brackets (`[ ]`). The first element of `x` is retrieved by typing `x[1]`, the first two elements by `x[1:2]` and nonconsecutive elements by `x[c(1, 4)]`.

```
>x[1]
[1] 10

> x[1:2]
[1] 10 9

> x[c(1, 4)]
[1] 10  7
```

1.7.2. Factor Vectors

Factors are used to create a vector to the integer data type.

Let's create a simple vector of text data that has a few repeats. We will start with the vector q we created earlier and add some elements to it.

Example:

```
> x <- c("Hockey", "Cricket", "Volleyball", "Tennis")
> x
[1] "Hockey"     "Cricket"    "Volleyball" "Tennis"
```

Converting this to a factor is easy with as.factor.

```
> y<-as.factor(x)
> y
[1] Hockey     Cricket    Volleyball Tennis
Levels: Cricket Hockey Tennis Volleyball
```

1.7.3 Missing Data

Missing data plays a critical role in both statistics and computing, and R has two types of missing data, NA and NULL. While they are similar, they behave differently and that difference needs attention.

NA

Often we will have data that has missing values for any number of reasons. Statistical programs use varying techniques to represent missing data such as a dash, a period or even the number 99. R uses NA. NA will often be seen as just another element of a vector. is.na tests each element of a vector for missing.

Example:

```
> z <- c(1, 2, NA, 8, 3, NA, 3)
> z

[1] 1 2 NA 8 3 NA 3

>is.na(z)

[1] FALSE FALSE TRUE FALSE FALSE TRUE FALSE
```

NA is entered simply by typing the letters "N" and "A" as if they were normal text. This works for any kind of vector.

NULL

NULL is the absence of anything. It is not exactly missingness, it is nothingness. Functions can sometimes return NULL and their arguments can be NULL. An important difference between NA and NULL is that NULL is atomical and cannot exist within a vector. If used inside a vector it simply disappears.

```
> z <- c(1, NULL, 3)
> z

[1] 1 3
```

ven though it was entered into the vector z, it did not get stored in z. In fact, z is only two elements long.

The test for a NULL value is is.null.

```
> d <- NULL
>is.null(d)
[1] TRUE

>is.null(7)
[1] FALSE
```

Since NULL cannot be a part of a vector, is.null is appropriately not vectorized.

Advanced Data Structures

1.8 data.frames

The data.frame is just like an Excel spreadsheet in that it has columns and rows. In statistical terms, each column is a variable and each row is an observation.

In terms of how R organizes data.frames, each column is actually a vector, each of which has the same length. That is very important because it lets each column hold a different type of data.This also implies that within a column each element must be of the same type, just like with vectors.

There are numerous ways to construct a data.frame, the simplest being to use the data.frame function. Let's create a basic data.frame using some of the vectors we have already introduced, namely x, y and q.

Example:

```
> x <- 10:1
> y <- -4:5
> q <- c("Hockey", "Football", "Baseball", "Curling", "Rugby",
+        "Lacrosse", "Basketball", "Tennis", "Cricket", "Soccer")
>df<- data.frame(x, y, q)
>df

   x   y          q
1 10  -4     Hockey
2  9  -3   Football
3  8  -2   Baseball
4  7  -1    Curling
5  6   0      Rugby
6  5   1   Lacrosse
7  4   2 Basketball
8  3   3     Tennis
9  2   4    Cricket
10 1   5     Soccer
```

Assigned names during the creation process

Example:

```
>df<- data.frame(First = x, Second = y, Sport = q)
>df

   First Second      Sport
1     10     -4     Hockey
2      9     -3   Football
3      8     -2   Baseball
4      7     -1    Curling
5      6      0      Rugby
6      5      1   Lacrosse
7      4      2 Basketball
8      3      3     Tennis
9      2      4    Cricket
10     1      5     Soccer
```

Functions on Data Frames

```
>nrow(df)
[1] 10

>ncol(df)
[1] 3

>dim(df)
[1] 10 3

>names(df)
[1] "First" "Second" "Sport"

>names(df)[3]

[1] "Sport"
```

Row names of a data.frame.

Example:

```
>rownames(df)

 [1]  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

>rownames(df) <- c("One", "Two", "Three", "Four", "Five", "Six",
+                       "Seven", "Eight", "Nine", "Ten")
>rownames(df)

 [1] "One"   "Two"  "Three" "Four"  "Five"  "Six"  "Seven"  "Eight"
 [9] "Nine"  "Ten"

># set them back to the generic index
>rownames(df) <- NULL
>rownames(df)
 [1]  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Usually a `data.frame` has far too many rows to print them all to the screen, so thankfully the `head` function prints out only the first few rows.

Example:

```
>head(df)

  First Second    Sport
1    10     -4   Hockey
2     9     -3 Football
3     8     -2 Baseball
4     7     -1  Curling
5     6      0    Rugby
6     5      1 Lacrosse

>head(df, n = 7)

  First Second      Sport
1    10     -4     Hockey
2     9     -3   Football
3     8     -2   Baseball
4     7     -1    Curling
5     6      0      Rugby
6     5      1   Lacrosse
7     4      2 Basketball

>tail(df)

  First Second      Sport
5     6      0      Rugby
6     5      1   Lacrosse
7     4      2 Basketball
8     3      3     Tennis
9     2      4    Cricket
10    1      5     Soccer
```

As we can with other variables, we can check the `class` of a `data.frame` using the `class` function.

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

Since each column of the `data.frame` is an individual `vector`, it can be accessed individually and each has its own `class`. Like many other aspects of `R`, there are multiple ways to access an individual column. There is the `$` operator and also the square brackets. Running `theDF$Sport` will give the third column in `theDF`. That allows us to specify one particular column by name.

Example:

```
>df$Sport

 [1] Hockey     Football   Baseball   Curling    Rugby      Lacrosse
 [7] Basketball Tennis     Cricket    Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis
```

Similar to `vector`s, `data.frame`s allow us to access individual elements by their position using square brackets, but instead of having one position two are specified. The first is the row number and the second is the column number. So to get the third row from the second column we use `theDF[3, 2]`.

```
>theDF[3, 2]

[1] -2
```

To specify more than one row or column use a `vector` of indices.

Example:

```
>df[3, 2:3]

  Second    Sport
3     -2 Baseball

>
>df[c(3, 5), 2]

[1] -2 0

>
>df[c(3, 5), 2:3]

  Second    Sport
3     -2 Baseball
5      0    Rugby
```

To access an entire row, specify that row while not specifying any column. Likewise, **to access an entire column,** specify that column while not specifying any row.

Example:

```
>df[, 3]

 [1] Hockey     Football   Baseball   Curling    Rugby      Lacrosse
 [7] Basketball Tennis     Cricket    Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis
>
>df[, 2:3]

  Second      Sport
1     -4     Hockey
2     -3   Football
3     -2   Baseball
4     -1    Curling
5      0      Rugby
6      1   Lacrosse
7      2 Basketball
8      3     Tennis
9      4    Cricket
10     5     Soccer

>
>df[2, ]

  First Second    Sport
2     9     -3 Football

>
>df[2:4, ]

  First Second    Sport
2     9     -3 Football
3     8     -2 Baseball
4     7     -1  Curling
```

1.9. Lists

`list in R` stores any number of items of any type. A `list` can contain all `numeric`s or `character`s or a mix of the two or `data.frame`s or, recursively, other `list`s.

`List`s are created with the `list` function where each argument to the function becomes an element of the `list`.

Example:

```
>#  creates a three element list
>list(1, 2, 3)

[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

>
>#  creates a single element list where the only element is a vector
>#  that has three elements
>list(c(1, 2, 3))

[[1]]
[1] 1 2 3

>
>#  creates a two element list
>#  the first element is a three element vector
>#  the second element is a five element vector
> list3 <- list(c(1, 2, 3), 3:7)

[[1]]
[1] 1 2 3

[[2]]
[1] 3 4 5 6 7
>
>#  two element list
>#  first element is a data.frame
>#  second element is a 10 element vector
>list4 <- list(df, 1:10)

>list4
[[1]]
   First Second      Sport
1     10     -4     Hockey
2      9     -3   Football
3      8     -2   Baseball
4      7     -1    Curling
5      6      0      Rugby
6      5      1   Lacrosse
7      4      2 Basketball
8      3      3     Tennis
9      2      4    Cricket
10     1      5     Soccer

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

.

Like `data.frame`s, `list`s can have names. Each element has a unique name that can be either viewed or assigned using `names`.

Example:

```
> names(list4)
NULL

> names(list4) <- c("data.frame", "vector")
> names(list4)
[1] "data.frame" "vector"

>names(list4)
NULL

>names(list4) <- c("data.frame", "vector", “list”)
>names(list4)
[1] "data.frame" "vector"     "list"

> list4
$`data.frame`
    x  y          q
1  10 -4     Hockey
2   9 -3   Football
3   8 -2   Baseball
4   7 -1    Curling
5   6  0      Rugby
6   5  1   Lacrosse
7   4  2 Basketball
8   3  3     Tennis
9   2  4    Cricket
10  1  5     Soccer

$vector
 [1]  1  2  3  4  5  6  7  8  9 10
```

Creating an empty list of a certain size is, perhaps confusingly, done with vector.

Example:

```
> (emptyList<- vector(mode = "list", length = 4))
[[1]]
NULL
[[2]]
NULL
[[3]]
NULL
[[4]]
NULL
```

To access an individual element of a `list`,

use double square brackets, specifying either the element number or name. Note that this allows access to only one element at a time.

Example:

```
> list4[1]
$`data.frame`
    x  y          q
1  10 -4     Hockey
2   9 -3   Football
3   8 -2   Baseball
4   7 -1    Curling
5   6  0      Rugby
6   5  1   Lacrosse
7   4  2 Basketball
8   3  3     Tennis
9   2  4    Cricket
10  1  5     Soccer


> list4[2]

$`vector`

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

Once an element is accessed it can be treated as if that actual element is being used, allowing nested indexing of elements.

Example:

```
> list4[[1]]$q

 [1] Hockey     Football   Baseball   Curling    Rugby      Lacrosse
Basketball Tennis     Cricket    Soccer

Levels: Baseball Basketball Cricket Curling Football Hockey Lacrosse Rugby
Soccer Tennis

> list4[[1]][,3,drop=FALSE]
           q
1      Hockey
2    Football
3    Baseball
4     Curling
5       Rugby
6    Lacrosse
7  Basketball
8      Tennis
9     Cricket
10     Soccer
>
```

It is possible to append elements to a `list` simply by using an index (either numeric or named) that does not exist.

Example:

```
>#  see how long it currently is
> length(list4)

[1] 2

>
>#  add a third element, unnamed
>  list4[[3]] <- 2
> list4
$`data.frame`
    x  y          q
1  10 -4     Hockey
2   9 -3   Football
3   8 -2   Baseball
4   7 -1    Curling
5   6  0      Rugby
6   5  1   Lacrosse
7   4  2 Basketball
8   3  3     Tennis
9   2  4    Cricket
10  1  5     Soccer

$vector
 [1]  1  2  3  4  5  6  7  8  9 10

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

Here number 2 is add to the list as a third element

1.10.Matrices

`matrix`is similar to a `data.frame` in that it is rectangular with rows and columns with numerical values and with same type. They also act similarly to `vector`s with element-by-element addition, multiplication, subtraction, division and equality. The `nrow`, `ncol` and `dim` functions work just like they do for `data.frame`s.

Example:

```
>#  create a 5x2 matrix
> A <- matrix(1:10, nrow = 5)


>#  create another 5x2 matrix
> B <- matrix(21:30, nrow = 5)


>#  create another 5x2 matrix
> C <- matrix(21:40, nrow = 2)
> A

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

```
> B

     [,1] [,2]
[1,]   21   26
[2,]   22   27
[3,]   23   28
[4,]   24   29
[5,]   25   30

> C

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   21   23   25   27   29   31   33   35   37    39
[2,]   22   24   26   28   30   32   34   36   38    40

>nrow(A)
[1] 5

>ncol(A)
[1] 2

>dim(A)

[1] 5 2

># addition
> A + B

     [,1] [,2]
[1,]   22   32
[2,]   24   34
[3,]   26   36
[4,]   28   38
[5,]   30   40

># multiplication
> A * B

     [,1] [,2]
[1,]   21  156
[2,]   44  189
[3,]   69  224
[4,]   96  261
[5,]  125  300
```

```
>#  see if the elements are equal
> A == B

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

`Matrix` multiplication is a commonly used operation in mathematics, requiring the number of columns of the left-hand `matrix` to be the same as the number of rows of the right-hand `matrix`. Both `A` and `B` are 5*X*2 so we will transpose `B` so it can be used on the right-hand side.

Example:

> A %*% **t**(B)

```
     [,1]  [,2] [,3]  [,4]  [,5]
[1,] 177  184  191  198  205
[2,] 224  233  242  251  260
[3,] 271  282  293  304  315
[4,] 318  331  344  357  370
[5,] 365  380  395  410  425
```

Another similarity with `data.frame`s is that `matrices` can also have row and column names.

Example:

```
>colnames(A)
NULL

>rownames(A)
NULL

>colnames(A) <- c("Left", "Right")
>rownames(A) <- c("1st", "2nd", "3rd", "4th", "5th")
>
>colnames(B)
NULL

>rownames(B)
NULL

>colnames(B) <- c("First", "Second")
>rownames(B) <- c("One", "Two", "Three", "Four", "Five")
>
>colnames(C)
NULL

>rownames(C)
NULL
```

```
>colnames(C) <- LETTERS[1:10]
>rownames(C) <- c("Top", "Bottom")
```

There are two special `vector`s, `letters` and `LETTERS`, that contain the lower-case and upper-case letters, respectively.

1.11. Arrays

An `array` is a multidimensional `vector`. It must all be of the same type and individual elements are accessed in a similar fashion using square brackets. The first element is the row index, the second is the column index and the remaining elements are for outer dimensions.

Example:

```
>theArray<- array(1:12, dim = c(2, 3, 2))
>theArray

, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

>the Array [1, , ] [,1] [,2]


     [1,]    1    7
     [2,]    3    9
     [3,]    5   11

>theArray[1, , 1]

[1] 1 3 5

>theArray[, , 1]

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

The main difference between an `array` and a `matrix` is that `matrices` are restricted to two dimensions while `array`s can have an arbitrary number.

STATISTICS WITH R PROGRAMMING UNIT-2

2.1 R Programming Structures :

Control Statements - Loops

Looping Over Non vector

Sets

2.2 If-Else

2.3 Arithmetic and Boolean Operators and values

2.4 Default Values for Argument

2.5 A. Return Values

B. Deciding Whether to explicitly call return

C. Returning Complex Objects

D. Functions are Objective,

2.6 No Pointers in R

2.7 Recursion –Example -1: A Quicksort Implementation

Example -2: ABinary Search Tree.

2.1 R PROGRAMMING STRUCTURES : CONTROL STATEMENTS

Control statements in R look very similar to C- Language.

A. LOOPS

Looping is the process of executing the statement(s) repeatedly until the condition is satisfy.
Every looping statement has four features. They are

i. Initialization
ii. Condition
iii. Statement(s)
iv. Increment or Decrement

Every loop has group of iterations. Each iteration has condition, statement(s) and Increment and decrement.

R- Language provides three looping statements. They are

i. for statement
ii. while statement
iii. repeat statement

i. for statement

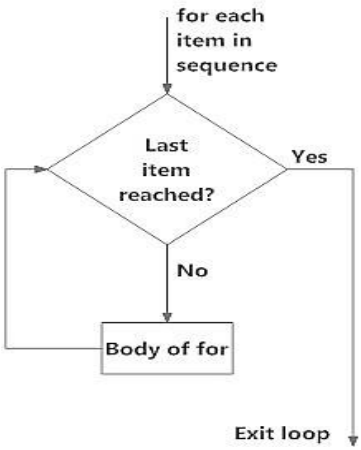
A for loop is used to iterate over a vector in R programming.

Syntax

```
for(val in sequence)
{
statement
}
```

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

Flowchart of for loop



Example-1:

```
# printing numbers from 1 to 10
for( iin 1:10 ) print(i)
```

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

Example-2 :

```
# Finding squares for given vector
> x <- c(5,12,13)
>for (n in x) print(n^2) [1]
25
[1] 144
[1] 169
```

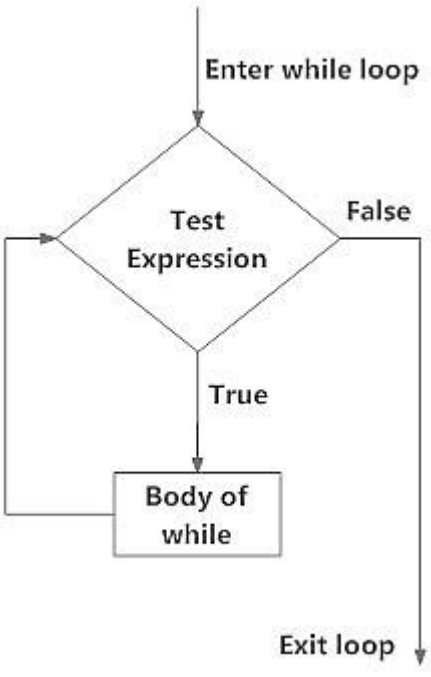
ii. while statement

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

Syntax:

```
Inititalization
while (condition) {

Statement(s)
Increment/ Decrement
}
```

Flowchart of while Loop



Example-1:

```
# printing numbers from 1 to 10
i<- 1
while(i<=10){
print(i)
i<-i+1
}

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

Example-2 :

```
# Finding squares for given vector x <-
c(5,12,13)
i<-1
     while(i<=length(x)){
          print(x[i]^2)
          i<-i+1
}


Output [1]

25
[1] 144
[1] 169
```

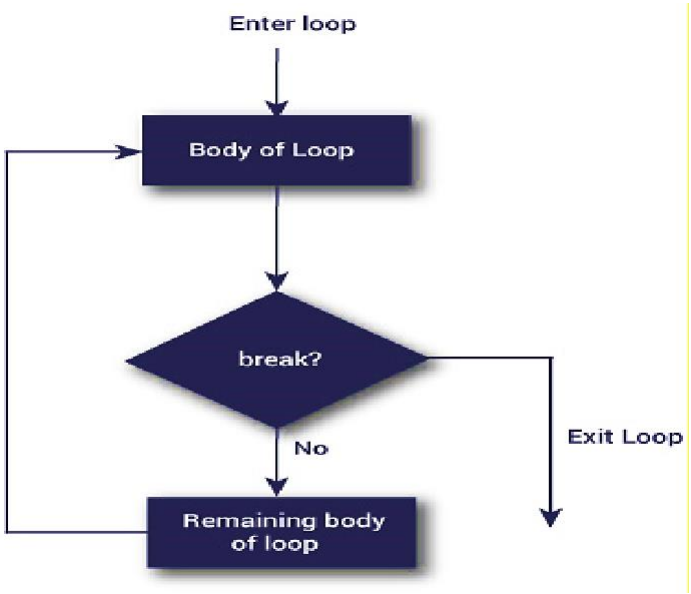
iii. repeat statement

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:

```
Inititalization
repeat {

if(condition) break
Statement(s)
Increment/ Decrement
}
```

Flowchart of repeat loop



Example-1:

```
# printing numbers from 1 to 10
i<- 1
repeat{
if(i>10) break
print(i)
i<-i+1
}

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

Example-2 :

```
# Finding squares for given vector x<-
c(5,12,13)
i<-1
repeat{
          if(i>length(x)) break
          print(x[i]^2)
          i<-i+1
}

Output [1]

25
[1] 144
[1] 169
```

In addition to the Looping, R language provides two statement for looping. They are

i. break statement
ii. next statement

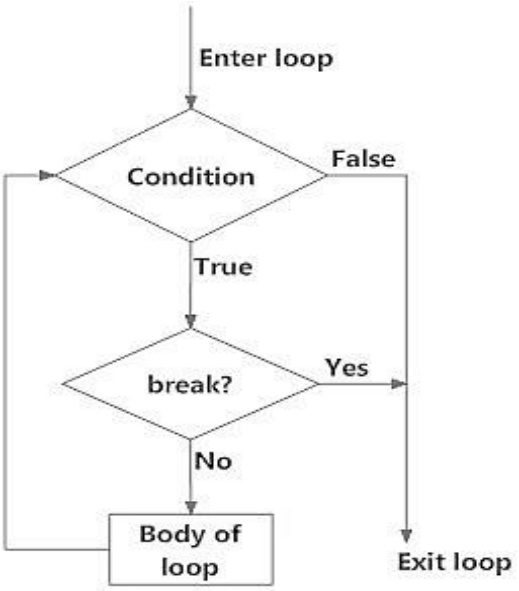
i. break statement

A `break` statement is used inside a loop (repeat, for, while) 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

```
break
```

Flowchart of break statement



Example:

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

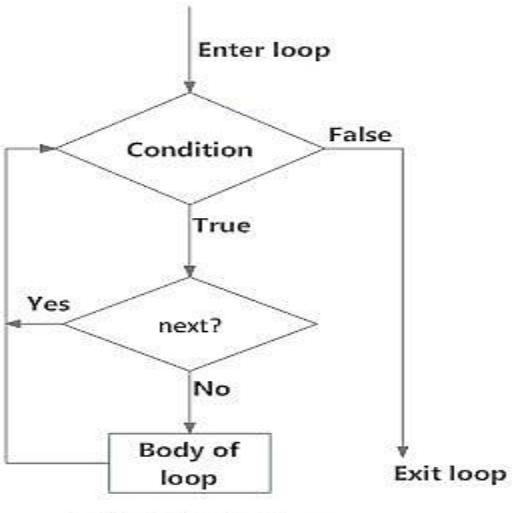
ii. next statement

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

```
next
```

Flowchart of next statement



Example:

```
x <- 1:5
for (val in 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.

2.1.B: Looping Over Non-vector Sets : lapply() function

R does not directly support iteration over nonvector sets, but there are a couple of indirect easy ways to accomplish it:

• Use lapply(), assuming that the iterations of the loop are independent of each other, thus allowing them to be performed in any order.

Example:

```
a<-c(1:10)

b<-c(-5:4)

x<-list(a,b)

# compute the list mean for each list element

mean1<-lapply(x,mean)

print(mean1)
```

output

```
[[1]]
[1] 5.5

[[2]]
[1] -0.5
```

2.2 if-else Statement

An **if** statement can be followed by an optional **else** statement which executes when the boolean expression is false.

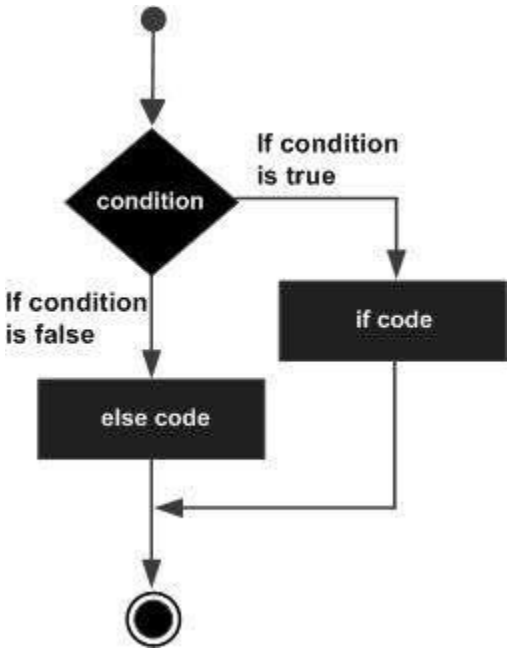
Syntax

The basic syntax for creating an **if...else** statement in R is −

```
if(boolean_expression) {
   // statement(s) will execute if the boolean expression is true.
} else {
   // statement(s) will execute if the boolean expression is false.
}
```

If the Boolean expression evaluates to be **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

Flow Diagram



Example

```
x <-10

if(x<=10){
print(100)
}else{
print(200)
}
```

Output

```
100
```

Here x is equal to 10, the condition x<=10 is true, so if statement is printing 100.

```
x <- 10

if(x>10){
print(100)
}else{
print(200)
}
```

Output

```
200
```

Here x is equal to 10, the condition x>10 is false so else block will be executed and it prints 200 on the screen.

2.3 Arithmetic and Boolean Operators and Values

Operation	Description
x + y	Addition
x – y	Subtraction
x * y	Multiplication
x / y	Division
x ^ y	Exponentiation
x %% y	Modular arithmetic
x %/% y	Integer division
x == y	Test for equality
x <= y	Test for less than or equal to
x >= y	Test for greater than or equal to
x && y	Boolean AND for scalars
x || y	Boolean OR for scalars
x & y	Boolean AND for vectors (vector x,y,result)
x | y	Boolean OR for vectors (vector x,y,result)
!x	Boolean negation

Examples:

```
>x<-c(TRUE, FALSE,TRUE)
>x
[1] TRUE FALSE TRUE
>y<-c(TRUE, TRUE, FALSE)
>y
[1] TRUE TRUE FALSE
>x& y
[1] TRUE FALSE FALSE


>x[1] && y[1]
[1] TRUE
>x&& y # looks at just the first elements of each vector
[1] TRUE
>if (x[1] && y[1]) print("both TRUE")
[1] "both TRUE"
>if (x & y) print("both TRUE")
[1] "both TRUE"
Warning message:
In if (x & y) print("both TRUE") :
The condition has length > 1 and only the first element will be
used
```

Here &(AND) will be applied on “first element in x” and “first element in y”. For that it prints the warning message on the screen.

To avoid the warning message just use && on x and y

The Boolean values “TRUE can be represented as T” , similarly “FALSE can be represented as F” (both must be capitalized). These values change to 1 and 0 in arithmeticexpressions:

```
     > T&T
[1] TRUE
     > T&F
[1] FALSE
     > F&F
[1] FALSE
     > T*5
[1] 5
     > F*5
[1] 0
```

We can apply the relational operator on numerical values.

```
>1 < 2
[1] TRUE
     > (1 < 2) * (3 < 4)
[1] 1
```

Here the comparison 1 < 2 returns TRUE, and 3 < 4 yields TRUE as well. Both values are treated as 1 values, so the product is 1(1 * 1 = 1)

```
     > (1 < 2) * (3 < 4) * (5 < 1)
[1] 0
     > (1 < 2) == TRUE
[1] TRUE
     > (1 < 2) == 1
[1] TRUE
```

2.4 Default Values for Arguments

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**.

Example -1:

```
# 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()
```

output

```
18
```

Example -2:

Here default values (a=3 and b=6) will be passed to new.function

```
# Call the function with giving new value of the argument.
new.function(9)
output
```

```
54
```

Here default value (b=6) will be passed to new.function because b is not specified.

Example -3:

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

```
output
```

```
54
```

Here no default values will be passed to new.function because a and b are specified.

2.5. A. Return Values

Function is a block statements which performs specific task. Function can take the values through the formal arguments and may return the value if is required.

The return value of a function can be numeric, character, date, Boolean, vector, data frame, matrix, list or an array.

Function return the value through the return() function.

Syntax

return(value)

return() in R is optional. We can return the value by specifying the variable name.

Example:

```
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) # function returns the value k with return()
}
```

```
Output
```

```
>oddcount(1:10) 5
```

Here the function explicitly using the return() in oddcount.

The following R- Code return the value by specifying the variable name

```
Oddcount<- function(x)
{
 k <- 0 # assign 0 to k for
(n in x)
{
if (n %% 2 == 1) k <- k+1        # %% is the modulo operator
}
k         # function returns the value k without
return()
}

Output
>oddcount(1:10) 5
```

So using return() in the function is optional. Based on the program complexity we can use return() explicitly.

2.5.B. Deciding Whether to Explicitly Call return()

Using return() explicitly in the function is based on the program size and complexity. If the program size is short then no need use return() explicitly. For example.

```
Oddcount<- function(x) { k <-
0 # assign 0 to k for (n in
x) {
if (n %% 2 == 1) k <- k+1        # %% is the modulo operator
}
k         # function returns the value k without return()
}

Output
>oddcount(1:10) 5
```

In the above example. It is simple to return the value. A call to return() wasn't necessary For

Good software design, it is better to use the return() explicitly for complex programs.

2.5.C Returning Complex Objects

We know that R-function return any object like numerical, character, date, vector, list etc. A function can return the complex object like function.

Here is an example of a function being returned:

```
g<- function() {
t <- function(x) return(x^2) # Here t is a function like g
return(t)
}
```

Output

```
>g()
function(x) return(x^2)
<environment: 0x8aafbc0>
```

Here tis a function because t is created with function() t<-function(x) After that the function g is returning t. That means g is returning functions.

2.5.D Functions are Objects

R functions are *first-class objects* that means they can be used for the most part just like other objects. Consider the following example

```
 g <- function(x) {
return(x+1)
      }
```

Here, function() is a built-in R function whose job is to create functions. Every function() has two arguments they are

1. The formal argument list- here it is x
2. The function body – here { return(x+1) }

We can access these two arguments can be accessed through formal() and body() functions

formals() function returns the formal arguments of given function.
Example:
>formals(g) x

body() function returns the body of given function Example:
>body(g)
{
return(x+1)
}

Function assignment

Since functions are objects, we can also assign them, use them as argumentsto other functions, and so on.

```
> f1 <- function(a,b) return(a+b)
> f2 <- function(a,b) return(a-b)

> f <- f1  #f1 is assign to f
```

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

     > f <- f2  #f2 is assign to f
>f(3,2)
[1] 1
```

Passing function as an argument

Since functions are objects, we can also pass them as an argument.
Example:

```
     > g <- function(h,a,b) h(a,b)
>g(f1,3,2)
[1] 5
>g(f2,3,2)
[1] 1
```

Here g is a function which takes three arguments, they are

1. h – is a function which is taking two arguments as h(a,b)
2. a- is a numerical
3. b- is a numerical

When we pass f1 function to g, immediately f1(a,b) will be called and the result a+b (3+2=5) is printed on the screen

2.6: No Pointers in R

R does not have variables corresponding to *pointers* or *references* like those of, say, the C language. This can make programming more difficult in some cases. (As of this writing, the current version of R has an experimental feature called *reference classes*)

Example-1:

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

The function sort() does not change x. If we do want x to change in this R code, the solution is to reassign the arguments:

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

Here R uses the reference classes to store the address of x and performing the sort on x and reassign values to x.

Example-2:

An example is the following function, which determines the indices ofodd and even numbers in a vector of integers:

```
Odds_evens<-function(v){
odds<- which(v %% 2 == 1)
```

```
evens<- which(v %% 2 == 1)
list(o=odds,e=evens)
}
```

Output

```
>x<-c(1:11)
>odds_evens(x)
$o 6
$e 5
```

```
Here odd numbers are 6 (1,3,5,7,9,11) and even numbers are 5
(2,4,6,8,10)
```

2.7 Recursion

A *recursive* function calls itself. To solve a problem of type X by writing a recursive function f():

1. Break the original problem of type X into one or more smaller problems of type X.
2. Within f(), call f() on each of the smaller problems.
3. Within f(), piece together the results of (b) to solve the original problem.

2.7.A : Example - A Quicksort Implementation

A classic example is Quickso1rt, an algorithm used to sort a vector of numbers from smallest to largest. For instance, suppose we wish to sort the vector(5,4,12,13,3,8,88).

We first compare everything to the first element, 5 (pivot element) ,to form two sub vectors: one consisting of the elements less than 5 and the other consisting of the elements greater than or equal to 5. That gives us sub vectors (4,3) and (12,13,8,88). We then call the function on the sub vectors, returning (3,4) and (8,12,13,88). We string those together with the 5, yielding (3,4,5,8,12,13,88), as desired.

R's vector-filtering capability and its c() function make implementation of Quick sort quite easy.

This example is for the purpose of demonstrating recursion. R's own sort function, sort(), is much faster, as it is written in C.

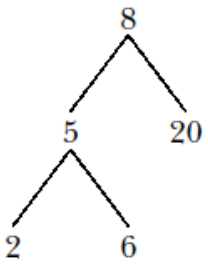
```
qs<- function(x) {
if (length(x) <= 1) return(x)
pivot<- x[1]
therest<- x[-1]
left<- therest[therest< pivot]
right<- therest[therest>= pivot]
left<- qs(left)
right<- qs(right)
return(c(left,pivot,right))
}
```

Output

```
>x<-c(5,4,12,13,3,8,88)
>qs(x)
[1] 3          4    5    8    12   13   88
```

2.7.B: Extended Example: A Binary Search Tree

Consider the following



R-code for Binary search tree is as follows

```
1 # routines to create trees and insert items into them are included 2 #
below; a deletion routine is left to the reader as an exercise
3
4 # storage is in a matrix, say m, one row per node of the tree; if row 5 # i
contains (u,v,w), then node i stores the value w, and has left and 6 # right links
to rows u and v; null links have the value NA
7
        8 # the tree is represented as a list (mat,nxt,inc), where mat is the
        9 # matrix, nxt is the next empty row to be used, and inc is the number of
        10 # rows of expansion to be allocated whenever the matrix becomes full
        11
12 # print sorted tree via in-order traversal 13
printtree<- function(hdidx,tr) {
        14 left <- tr$mat[hdidx,1]
        15 if (!is.na(left)) printtree(left,tr)
        16 print(tr$mat[hdidx,3]) # print root
        17 right <- tr$mat[hdidx,2]
18 if (!is.na(right)) printtree(right,tr) 19 }
20
21 # initializes a storage matrix, with initial stored value firstval 22
newtree<- function(firstval,inc) {
23 m <- matrix(rep(NA,inc*3),nrow=inc,ncol=3) 24
m[1,3] <- firstval
25 return(list(mat=m,nxt=2,inc=inc))
26 }
27
        28 # inserts newval into the subtree of tr, with the subtree's root being
        29 # at index hdidx; note that return value must be reassigned to tr by the
        30 # caller (including ins() itself, due to recursion)
        31 ins <- function(hdidx,tr,newval) {
        32 # which direction will this new node go, left or right?
        33 dir<- if (newval<= tr$mat[hdidx,3]) 1 else 2
34 # if null link in that direction, place the new node here, otherwise 35 #
recurse
```

```
        36 if (is.na(tr$mat[hdidx,dir])) {
        37 newidx<- tr$nxt # where new node goes
        38 # check for room to add a new element
        39 if (tr$nxt == nrow(tr$mat) + 1) {
        40 tr$mat<-
        41 rbind(tr$mat, matrix(rep(NA,tr$inc*3),nrow=tr$inc,ncol=3))
42 }
        43 # insert new tree node
        44 tr$mat[newidx,3] <- newval
        45 # link to the new node
        46 tr$mat[hdidx,dir] <- newidx
        47 tr$nxt<- tr$nxt + 1 # ready for next insert
        48 return(tr)
49 } else tr<- ins(tr$mat[hdidx,dir],tr,newval) 50 }
```

Output

```
#tree building using its routines
     > x <- newtree(8,3)
>x
$mat
[,1] [,2] [,3]
[1,] NA NA 8
[2,] NA NANA
[3,] NA NANA
$nxt
[1] 2
$inc
[1] 3

     > x <- ins(1,x,5)
>x
$mat
      [,1] [,2] [,3]
[1,] 2 NA 8
[2,] NA NA 5
[3,] NA NANA
$nxt
[1] 3
$inc
[1] 3

     > x <- ins(1,x,6)
>x
$mat
[,1] [,2] [,3]
[1,] 2 NA 8
```

```
[2,] NA 3 5
[3,] NA NA 6
$nxt
[1] 4
$inc
[1] 3

     > x <- ins(1,x,2)
>x
$mat
[,1] [,2] [,3]
[1,] 2 NA 8
[2,] 4 3 5
[3,] NA NA 6
[4,] NA NA 2
[5,] NA NANA
[6,] NA NANA
$nxt
[1] 5
$inc
[1] 3

     > x <- ins(1,x,20)
>x
$mat
[,1] [,2] [,3]
      [1,] 2 5 8
[2,] 4 3 5
[3,] NA NA 6
[4,] NA NA 2
[5,] NA NA 20
[6,] NA NANA
$nxt
[1] 6
$inc
[1] 3
```

Execution Explanation

First, the command containing our call newtree(8,3) creates a new tree, assigned to x, storing the number 8. The argument 3 specifies that we allocate storage room three rows at a time.

The result is that the matrix component of the list x is now as follows:

```
[,1] [,2] [,3]
[1,] NA NA 8
[2,] NA NANA
[3,] NA NANA
```

Three rows of storage are indeed allocated, and our data now consists just of the number 8. The two NA values in that first row indicate that this node of the tree currently has no children.

We then make the call ins(1,x,5) to insert a second value, 5, into thetree x. The argument 1 specifies the root. In other words, the call says,“Insert 5 in the subtree of x whose root is in row 1.” Note that we need toreassign the return value of this call back to x. Again, this is due to the lackof pointer variables in R. The matrix now looks like this:

```
[,1] [,2] [,3]
[1,] 2 NA 8
[2,] NA NA 5
[3,] NA NANA
```

The element 2 means that the left link out of the node containing 8 ismeant to point to row 2, where our new element 5 is stored.
The session continues in this manner. Note that when our initial allotmentof three rows is full, ins() allocates three new rows, for a total of six. Inthe end, the matrix is as follows:

```
[,1] [,2] [,3]
[1,] 2 5 8
[2,] 4 3 5
[3,] NA NA 6
[4,] NA NA 2
[5,] NA NA 20
[6,] NA NANA
This represents the tree we graphed for this example.
```

Program Explanation

There is recursion in both printtree() and ins(). The former is definitelythe easier of the two, so let’s look at that first. It prints out the tree, in sortedorder.

Recall our description of a recursive function f() that solves a problemof category X: We have f() split the original X problem into one or moresmaller X problems, call f() on them, and combine the results.

In this case,our problem’s category X is to print a tree, which could be a subtree of alarger one. The role of the function on line 13 is to print the given tree,which it does by calling itself in lines 15 and 18. There, it prints first the leftsubtree and then the right subtree, pausing in between to print the root.

This thinking—print the left subtree, then the root, then the rightsubtree—forms the intuition in writing the code, but again we must makesure to have a proper termination mechanism.
This mechanism is seen inthe if() statements in lines 15 and 18.

When we come to a null link, we do not continue to recursion.The recursion in ins() follows the same principles but is considerablymore delicate. Here, our “category X” is an insertion of a value into a subtree.

We start at the root of a tree, determine whether our new value mustgo into the left or right subtree (line 33), and then call the function againon that subtree. Again, this is not hard in principle, but a number of detailsmust be attended to, including the expansion of the matrix if we run out ofroom (lines 40–41).

One difference between the recursive code in printtree() and ins() isthat the former includes two calls to itself, while the latter has only one. Thisimplies that it may not be difficult to write the latter in a nonrecursive form.

STATISTICS WITH R PROGRAMMING

UNIT-3

3.1 Doing Math and Simulation in R:

A. Math Function

B. Extended Example Calculating Probability

C. Cumulative Sums and Products

D. Minima and Maxima

E. Calculus,

3.2 Functions for Statistical Distribution

3.3 Sorting

3.4 Linear Algebra Operation on Vectors and Matrices

A. Extended Example: Vector cross Product

B. Extended Example: Finding Stationary Distribution of Markov Chains

3.5 Set Operation

3.6 Input /out put : Accessing the Keyboard and Monitor

3.7 Reading and writer Files

3.1 Doing Math and Simulation in R:

A. Math Function

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

1. **exp(): Exponential function, base e**

Example:

```
>exp(1)
[1] 2.718282
>
```

2. **log(): Natural logarithm**

Example:

```
> log(1)
[1] 0.6931472
>
```

3. **log10(): Logarithm base 10**

Example:

```
> log10(3)
[1] 0.4771213
>
```

4. **sqrt(): Square root**

Example:

```
> sqrt(5)
[1] 2.236068
>
```

5. **abs(): Absolute value**

Example:

```
> abs(-5)
[1] 5
>
```

6. **sin(), cos(), and so on: Trig functions**

Example:

```
> sin(30)
[1] -0.9880316
> cos(30)
[1] 0.1542514
> tan(30)
[1] -6.405331
>
```

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

Example:

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

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

Example:

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

9. **pmin() and pmax(): Element-wise minima and maxima of several vectors Example: 1**

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

Example: 2

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

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

Example:

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


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

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

Example:

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

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

Example: round()

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

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

Example: ceiling()

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

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

Example:

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

B. Extended Example: Calculating a Probability

R language provides the facility to calculate the probability. Suppose we have *n* independent events, and the *i*th event has the probability *pi* of occurring.
What is the probability of exactly one of these events occurring?
Suppose first that *n* = 3 and our events are named A, B, and C. Then we break down the computation as follows:
P(exactly one event occurs) = P(A and not B and not C) +
P(not A and B and not C) +
P(not A and not B and C)

P(A and not B and not C) would be $pA(1 - pB)(1 - pC)$, and so on.

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

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

Output

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

C. Cumulative Sums and Products

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

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

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

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

D. Minima and Maxima

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

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

E. Calculus

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

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

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

R-code for the above equation is as follows

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

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

3.2: 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 Common R Statistical Distribution Functions

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

Example:
Let's simulate 1,000 chi-square variates with 2 degrees of freedom and find their mean.

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

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

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

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

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

Here, we used q to indicate quantile—in this case, the 0.95 quantile, or the 95th percentile.

3.3 Sorting

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

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

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

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

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

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

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

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

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

3.4 Linear Algebra Operations on Vectors and Matrices

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

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

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

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

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

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

The solve() function

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

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

Its matrix form is as follows:

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

Here's the R- code:

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

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

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

The sweep() function

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

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

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

A. Extended Example: Vector Cross Product

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

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

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

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

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

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

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

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

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

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

B. Extended Example: Finding Stationary Distributions of Markov Chains

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

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

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

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

Since R vector and matrix indices start at 1 rather than 0, it will be convenient to relabel our states here as 1, 2, and 3 rather than 0, 1, and 2. For example, state 3 now means that we currently have two consecutive heads. Let *pij* denote the *transition probability* of moving from state *i* to state *j* during a time step. In the game example, for instance, *p*23 = 0.5, reflecting the fact that with probability 1/2, we will toss a head and thus move from
having one consecutive head to two. On the other hand, if we toss a tail while we are in state 2, we go to state 1, meaning 0 consecutive heads; thus *p*21 = 0.5.

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

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

$\pi = \pi P$ (8.4)

which is equivalent to Equation 8.5:

$(I - PT)\pi = 0$ (8.5)

Here *I* is the identity matrix and *PT* denotes the transpose of *P*.

Any single one of the equations in the system of Equation 8.5 is redundant. We thus eliminate one of them, by removing the last row of $I-P$ in Equation 8.5. That also means removing the last 0 in the 0 vector on the right-hand side of Equation 8.5.

But note that there is also the constraint shown in Equation 8.6.

$$\sum_i \pi_i = 1$$

In matrix terms, this is as follows:

$$1_n^T \pi = 1$$

where 1*n* is a vector of *n* 1s. So, in the modified version of Equation 8.5, we replace the removed row with a row of all 1s and, on the right-hand side, replace the removed 0 with a 1. We can then solve the system.

All this can be computed with R's solve() function, as follows:

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

Here are the main steps:

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

3.5 Set Operations

R includes some handy set operations, including these:

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

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

Here are some simple examples of using these functions:

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

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

```
 > c32 <- combn(1:3,2)
 > c32
[,1] [,2] [,3]
[1,] 1 1 2
[2,] 2 3 3
 > class(c32)
[1] "matrix"
```

The results are in the columns of the output. We see that the subsets of {1,2,3} of size 2 are (1,2), (1,3), and (2,3). The function also allows you to specify a function to be called by combn() on each combination. For example, we can find the sum of the numbers in each subset, like this:

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

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

3.6 INPUT/OUTPUT : Accessing the Keyboard and Monitor

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

A. Accessing the Keyboard

i. Using the scan() Function

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

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

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

.

ii. Using the readline() Function

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

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

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

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

iii. Using the readLines() Function

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

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

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

B. Accessing the Monitor

i. The print() function

print() function prints the data on the monitor

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

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

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

```
>print("abc")
[1] "abc"
```

ii. The cat() function

cat() function prints the data on the monitor

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

Note that we needed to supply our own end-of-line character, "\n", in the call to cat(). Without it, our next call would continue to write to the same line.

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

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

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

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

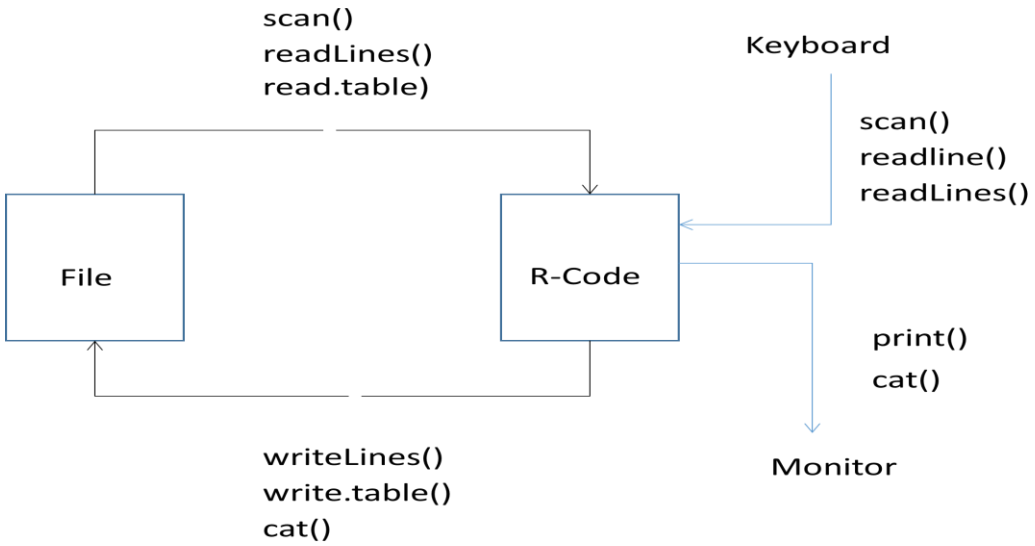
```
>cat(x,"abc","de\n",sep="\n")
1
2
3
abc
de
```

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

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

3.7: Reading and Writing Files

The details of functions involved in the reading and writing files



A. Reading data from a file

i. Introduction to Connections

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

Connecting the file with read mode

Suppose the file z1.txt has following code

Ravi 25
Nani 22
Babu 23

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


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

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

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

>readLines(c,n=1)
character(0)
```

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

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

```
c <- file("z","r")
while(TRUE) {
  rl<- readLines(c,n=1)
  if (length(rl) == 0) {
```

```
 print("reached the end")
break
} else print(rl)
}
```

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

Rewinding File

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

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

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

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

ii. Using the readLines() Function

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

hello
how
are you

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

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

iii. Using read.table() function

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

name	age
Ravi	25
Nani	22
Babu	23

We can read the z.txt file as

```
>z<- read.table("z.txt")
>z
```

[1]	name	age
[2]	Ravi	25
[3]	Nani	22
[4]	Babu	23

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

```
> z <- read.table("z",header=TRUE)
```

	name	age
[1]	Ravi	25
[2]	Nani	28
[3]	Babu	19

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

You can use scan() to read in a vector, whether numeric or character, from a file or the keyboard.

Suppose we have files named *z1.txt*, *z2.txt*, *z3.txt*, and *z4.txt*.

The z1.txt file contains the following:

123
4 5
6

The z2.txt file contents are as follows:

123
4.2 5 6

The z3.txt file contains this:

abc
de f g

And finally, the z4.txt file has these contents:

abc
123 6
y

Using the scan() function to read the data

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

Here we got a vector of four integers

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

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

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

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

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

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

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

Reading Matrix through scan() function

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

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

We can read it into a matrix this way:

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

B. Writing data to a file

i. write.table() function

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

```
>kids<- c("Jack","Jill")
>ages<- c(12,10)
 > d <- data.frame(kids,ages)
>d
kids ages
 1 Jack 12
 2 Jill 10
```

```
>write.table(d,"kds.txt")
```

```
The file kds.txt will now have these contents: "kids"
"ages"
"1" "Jack" 12
"2" "Jill" 10
```

ii. writeLines() function

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

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

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

iii. cat() function

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

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

The first call to cat() creates the file *u*, consisting of one line with contents "abc". The second call appends a second line

We can write multiple fields to the file as follows.

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

UNIT-4

4.1 Graphics

Creating Graphs

4.1.1 The Workhorse of R Base Graphics, the plot() Function

A. plot() function

B. Starting New Graph- x11() function

C. points() function

D. legend() function

E. text() function

F. locator() function

4.1.2 barplot() function

4.1.3 boxplot() function

4.1.4 Histogram - hist() function

4.1.5 PIE-CHART : pie() function

4.1.6 Strip- Chart : stripchart() function

4.2 Customizing Graphs

4.3 Saving Graphs to Files

4.4 R- Colors

4.1 Graphics

R has a very rich set of graphics facilities. The R home page (*http://www.r-project.org/*) has a few colorful examples, but to really appreciate R's graphical power, browse through the R Graph Gallery at *http://addictedtor.free.fr/graphiques*.

Creating Graphs

4.1.1 The Workhorse of R Base Graphics, the plot() Function

A. plot() function

A Plot 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. Plots are usually used in identifying the trends in data.

Plots are also called as Line Charts

The **plot()** function in R is used to creates the plot.

Syntax

```
plot(x, y, type, main, sub, xlab, ylab, asp , ...)
```

Arguments

`x` The x coordinates of points in the plot.

`y` the y coordinates of points in the plot,

`type`
: what type of plot should be drawn. Possible types are
- `"p"` for **p**oints,
- `"l"` for **l**ines,
- `"b"` for **b**oth,
- `"c"` for the lines part alone of `"b"`,
- `"o"` for both '**o**verplotted',
- `"h"` for '**h**istogram' like (or 'high-density') vertical lines,
- `"s"` for stair **s**teps,
- `"S"` for other **s**teps, see 'Details' below,
- `"n"` for no plotting.

`main`
: an overall title for the plot:

`sub`
: a sub title for the plot: see.

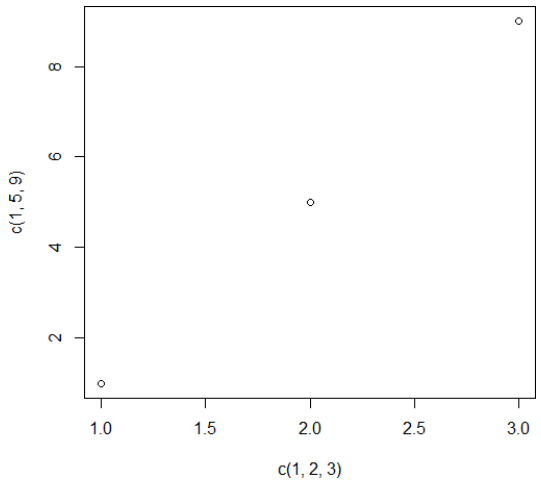
`xlab`
: a title for the x axis:

`ylab`
: a title for the y axis:

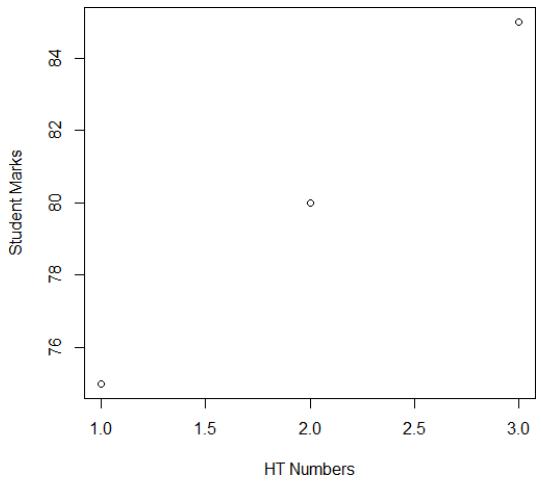
`asp`
: the *y/x* aspect ratio.

Example:

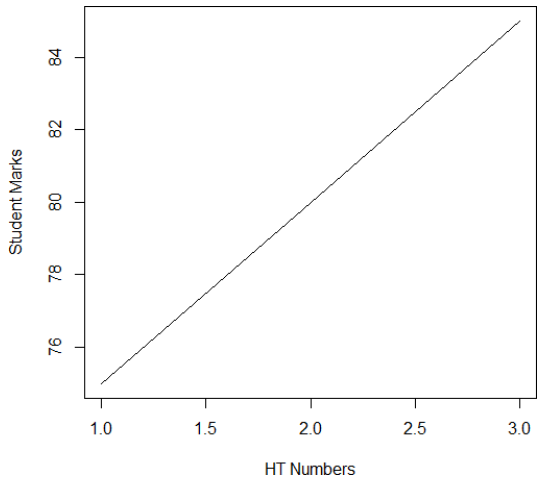
```
>plot(c(1,2,3),c(1,5,9))
```



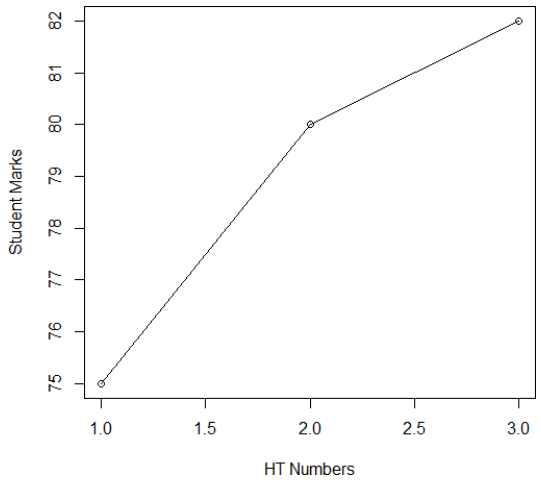

```
>#  Changing X axis and Y axis Names
>plot(c(1,2,3),c(75,80,85),xlab="HT Numbers",ylab="Student
Marks")
```



```
># Changing plot type as “l”
>plot(c(1,2,3),c(75,80,85),xlab="HT Numbers",ylab="Student
Marks", type=”l”)
```



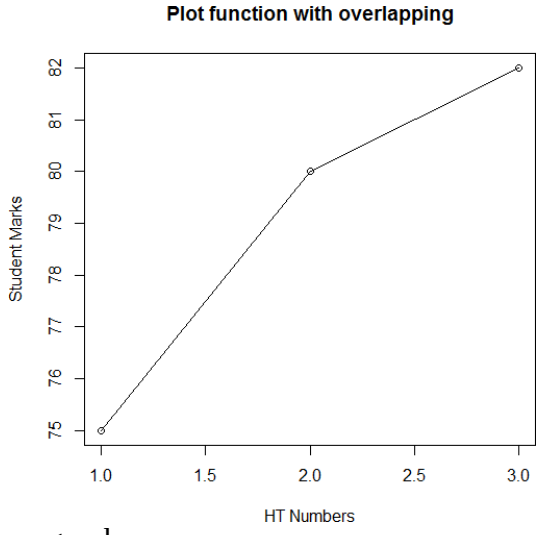

```
># Changing plot type as “o”
>plot(c(1,2,3),c(75,80,82),xlab="HT Numbers",ylab="Student
Marks", type=”o”)
```



```
>#  Changing title of the page- main

>plot(c(1,2,3),c(75,80,82),xlab="HT Numbers",ylab="Student
Marks", type=”o”, main=” Plot function with overlapping”)
```

A.

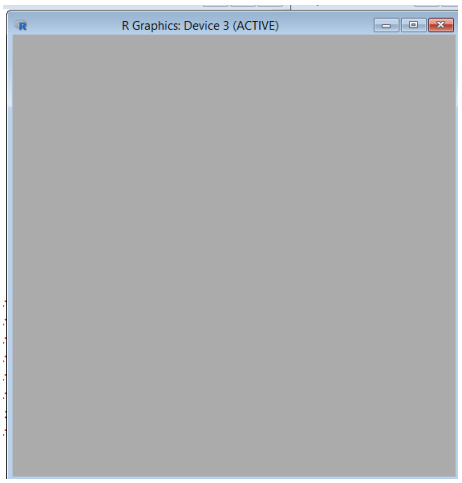
Each time when we call plot(), directly or indirectly, the current graph window will be replaced by the new



one. To create new window or new graph

B. <u>Starting New Graph- x11() function</u>

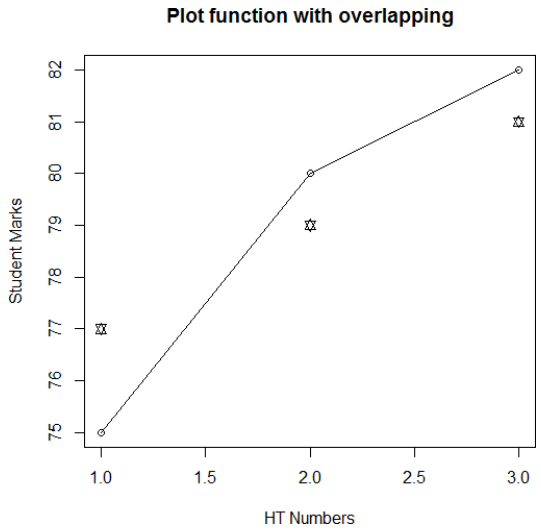
just use the x11() as follows
>x11()



C. points() function

The points() function adds a set of (*x*,*y*) points, with labels for each, to the currently displayed graph.
Example:

```
>points(c(1,2,3),c(77,79,81),pch=11)
```



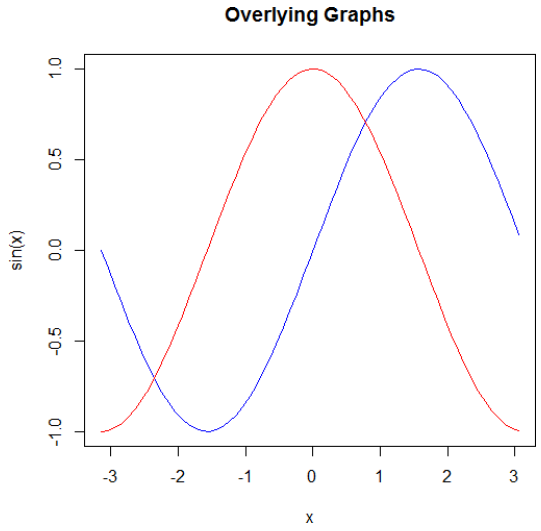
Here pch means Plot Character. pch=11 means, the character will be printed in plot diagram based on x and y axis coordinates.

D. legend() function

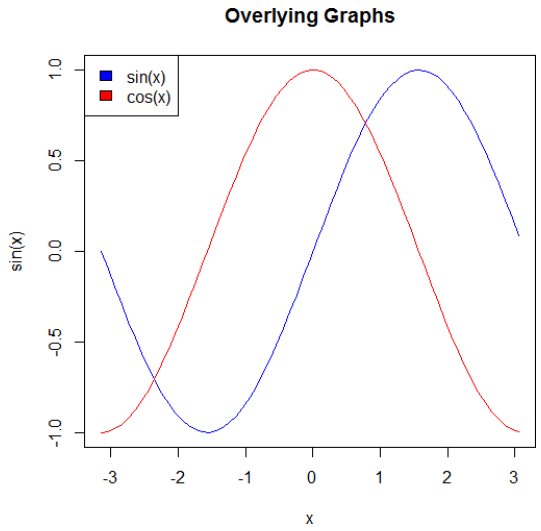
The legend() function is used to add extra information to the plot.
The following plot has two curved lines in the plot diagram. sin(x) and cos(x) depicted as blue and red colors in the plot diagram. The legend function gives an explanation about curved lines in the rectangle box with respective colors.

Example:

```
>x<-seq(-pi,pi,0.1)
>plot(x,sin(x), main= “Overlying Graphs”, type= “l”, col=“blue”)
>lines(x,cos(x), col=“red”)
```



>legend(“topleft”, c(“sin(x)”, “cos(x)”), fill=c(“blue”, “red”))

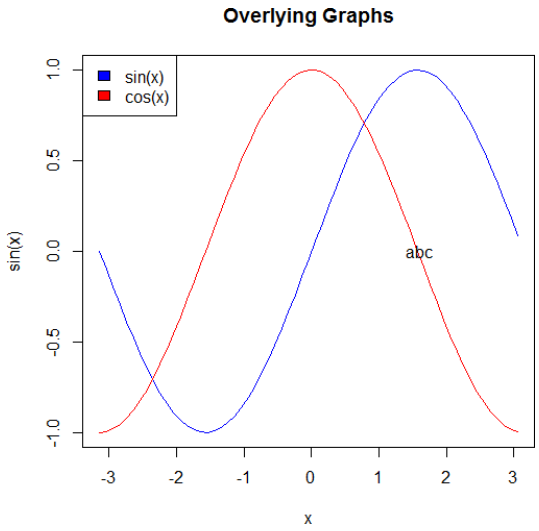


E. text() function

The text() function is used to place some text anywhere in the current graph.
Example:

```
>text(1.6, 0, "abc")
```

This writes the text “abc” at the point (1.6, 0) in the graph. The center of the string, in this case “b,” would go at that point.



F. locator() function

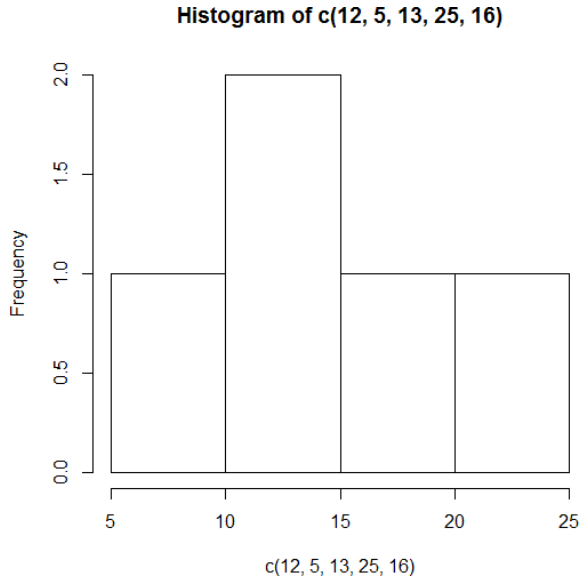
The locator() function retrieves the *x*- and *y*-coordinates where mouse pointer is selected in the graph. Simply call the function locator(1)and then click the mouse at the desired spot in the graph. The function returns the *x* and *y*-coordinates of your click point. Specifically, typing the following will tell R that you will click in one place in the graph:

```
>locator(1)
```

Once we click, R will tell us the exact coordinates of the point mouse clicked.
Call locator(2) to get the locations of two places, and so on.

Example:

```
> hist(c(12,5,13,25,16))
```




```
> locator(1)
$x
[1] 6.239237
$y
[1] 1.221038
```

4.1.2 barplot() function

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

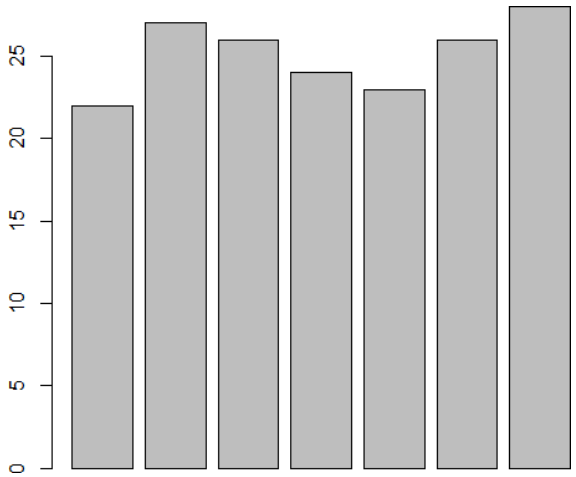
The basic syntax to create a bar-chart in R is:

```
>barplot(H,xlab,ylab,main, names.arg,col)
```

Following is the description of the parameters used:

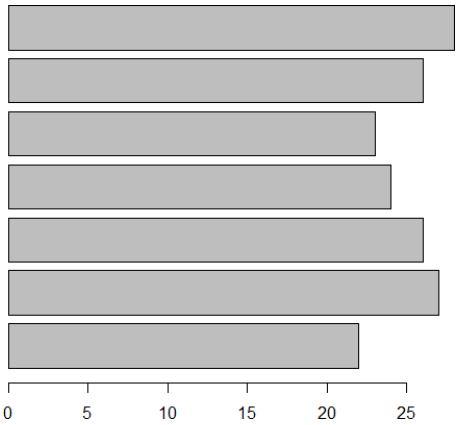
- **H** is a vector or matrix containing numeric values used in bar chart.
- **xlab** is the label for x axis.
- **ylab** is the label for y axis.
- **main** is the title of the bar chart.
- **names.arg** is a vector of names appearing under each bar.
- **col** is used to give colors to the bars in the graph.

```
> temp<-c(22,27,26,24,23,26,28)

> temp

[1] 22 27 26 24 23 26 28

> barplot(temp)
```

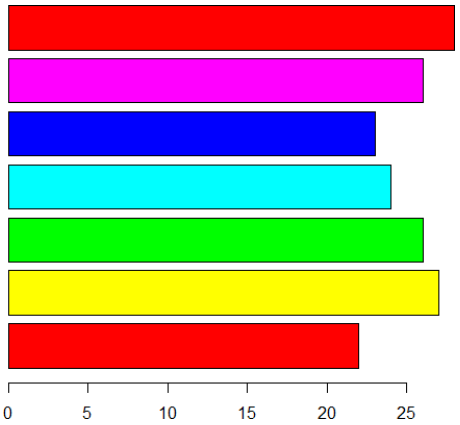


The bars can be pictured horizontally in the graph using the argument horiz=TRUE

> barplot(temp,horiz=TRUE)



Adding colors to the bars through rainbow() function
>barplot(temp,horiz=TRUE,col=rainbow(6))



4.1.3 boxplot() function

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.

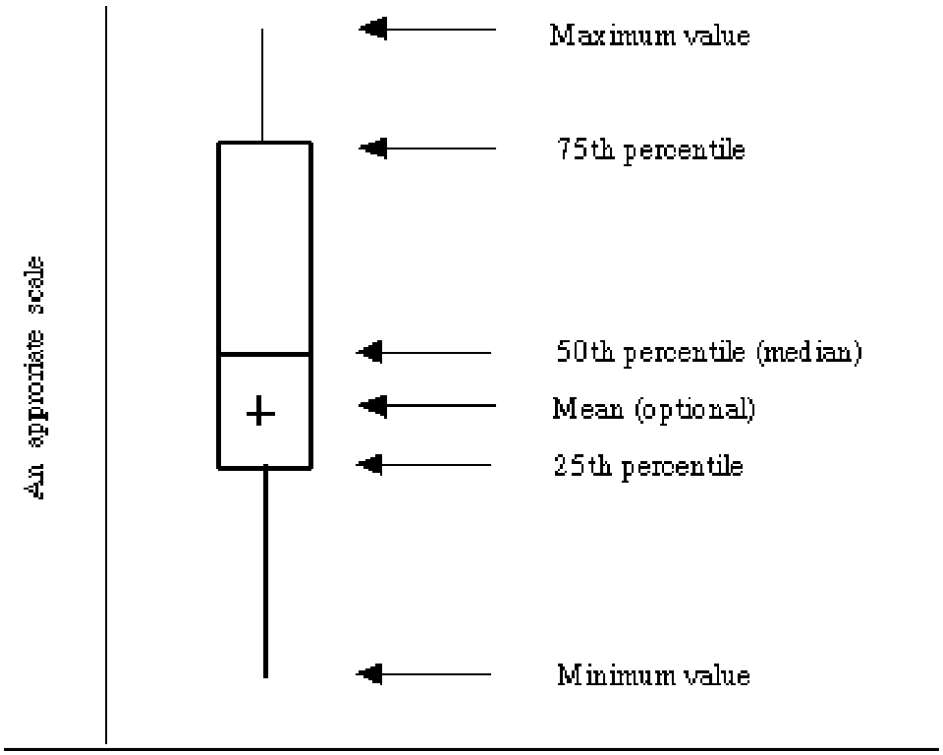
Syntax

The basic syntax to create a boxplot in R is :

boxplot(x,data,notch,varwidth,names,main)

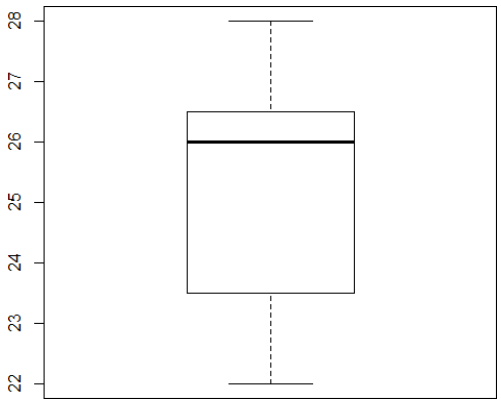
Following is the description of the parameters used:

- **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.



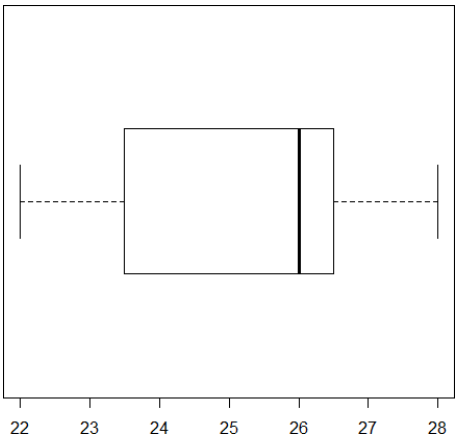
Example

```
> temp<-c(22,27,26,24,23,26,28)
> temp
[1] 22 27 26 24 23 26 28
> boxplot(temp)
```



Placing boxplot as horizontal

```
> boxplot(temp, horizontal=TRUE)
```



4.1.4 Histogram - hist() function

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:

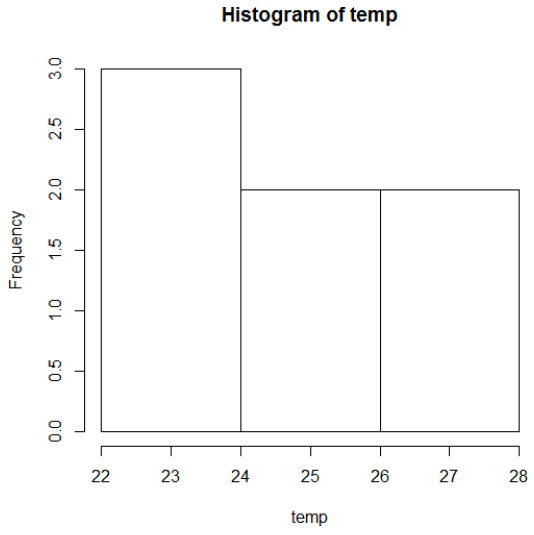
```
hist(v,main,xlab,xlim,ylim,breaks,col,border)
```

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

```
> temp<-c(22,27,26,24,23,26,28)
> temp
[1] 22 27 26 24 23 26 28
> hist(temp)
>
```



4.1.5 PIE-CHART : pie() function

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)
```

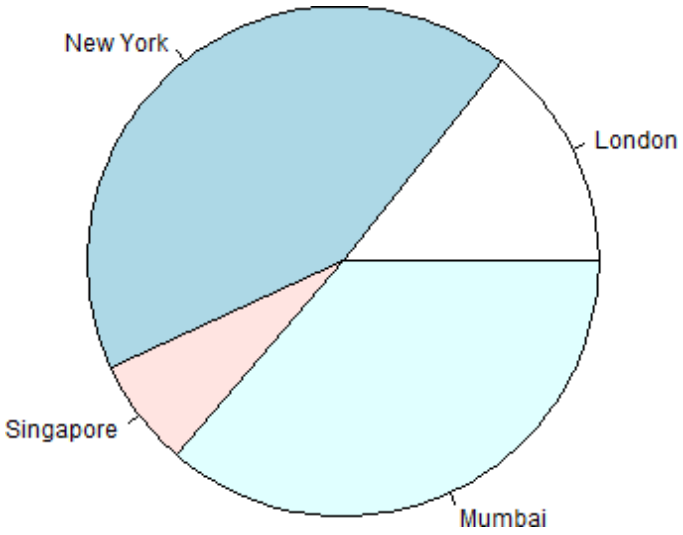
Following is the description of the parameters used:

- **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()
```



4.1.6 Strip- Chart : stripchart() function

stripchart produces one dimensional scatter plots (or dot plots) of the given data. These plots are a good alternative to boxplots when sample sizes are small.

Usage

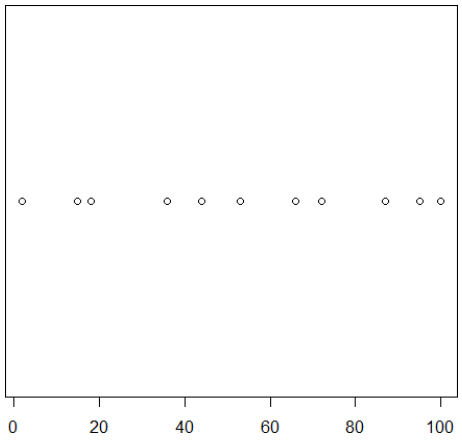
stripchart(x, ...)

Arguments

x the data from which the plots are to be produced

Example

```
>temp<-c(2,100,95,15,72,87,66,44,53,36,18)
> stripchart(temp,pch=21)
```

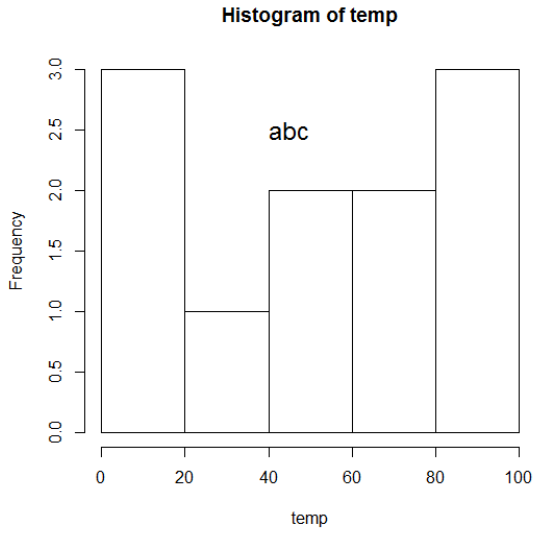


4.2 Customizing Graphs

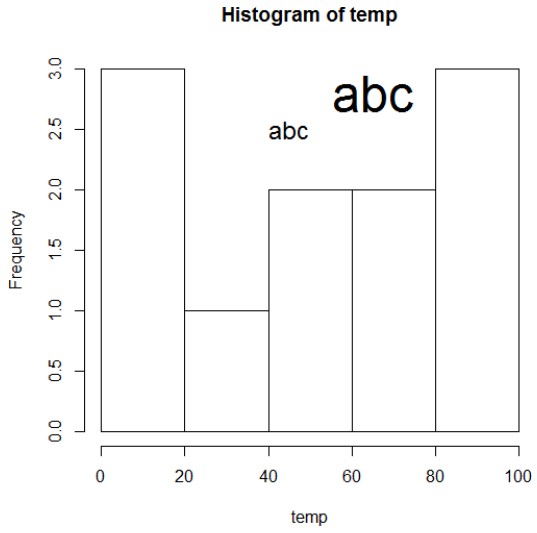
4.2.1 Changing Character Sizes: The cex Option

The cex (for *character expand*) function allows you to expand or shrink characters within a graph, which can be very useful. 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)
```




```
>text(65,2.8,"abc",cex = 3.0)
```



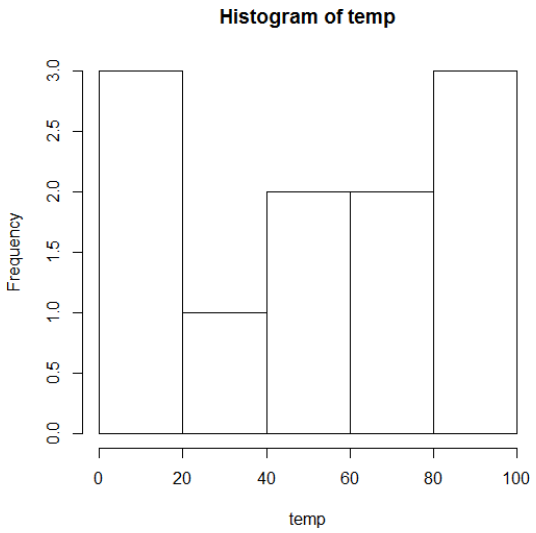
This prints the same text as in our earlier example but with characters 1.5 times the normal size.

4.2.2 Changing the Range of Axes: The xlim and ylim Options

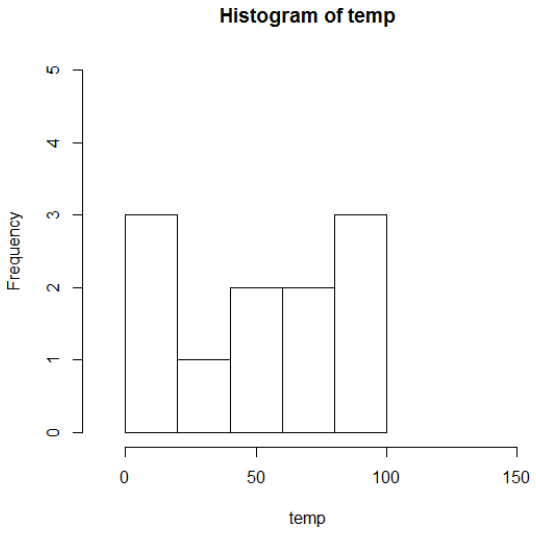
Xlim and ylim is used to set the ranges of x- and y- axis of plot.

Example:

Before using xlim and ylim
>hist(temp)



After using xlim and ylim
> hist(temp, xlim=c(-10,150),ylim=c(0,5))



12.2.3 Adding a Polygon: The polygon() Function

We can use polygon() to draw arbitrary polygonal objects.

Syntax

>polygon(x,y,…)

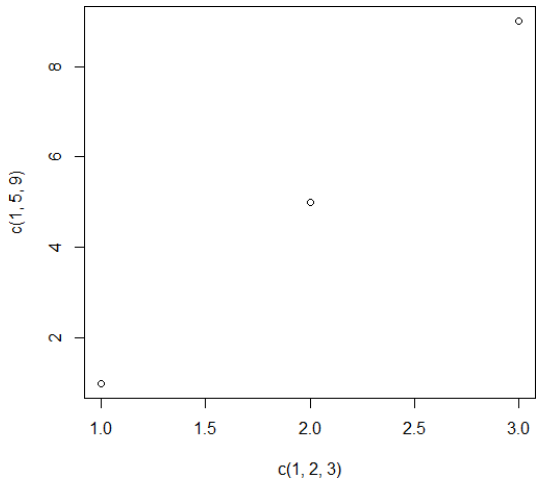
X – x axis coordinates

Y – y axis coordinates

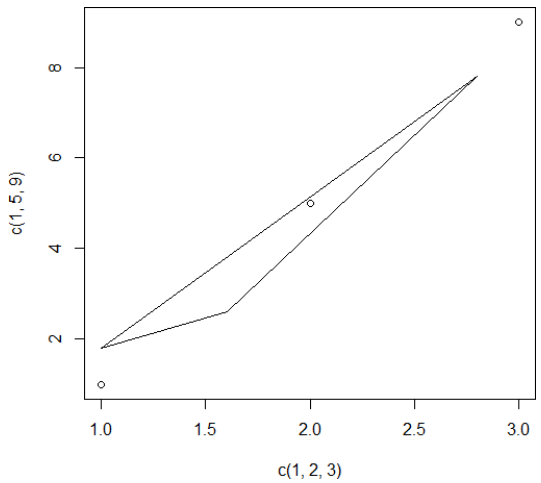
For example, the following code draws the polygon in the plot diagram

Before Polygon()

>plot(c(1,2,3),c(1,5,9))



>polygon(c(1,1.6,2.8),c(1.8,2.6,7.8))



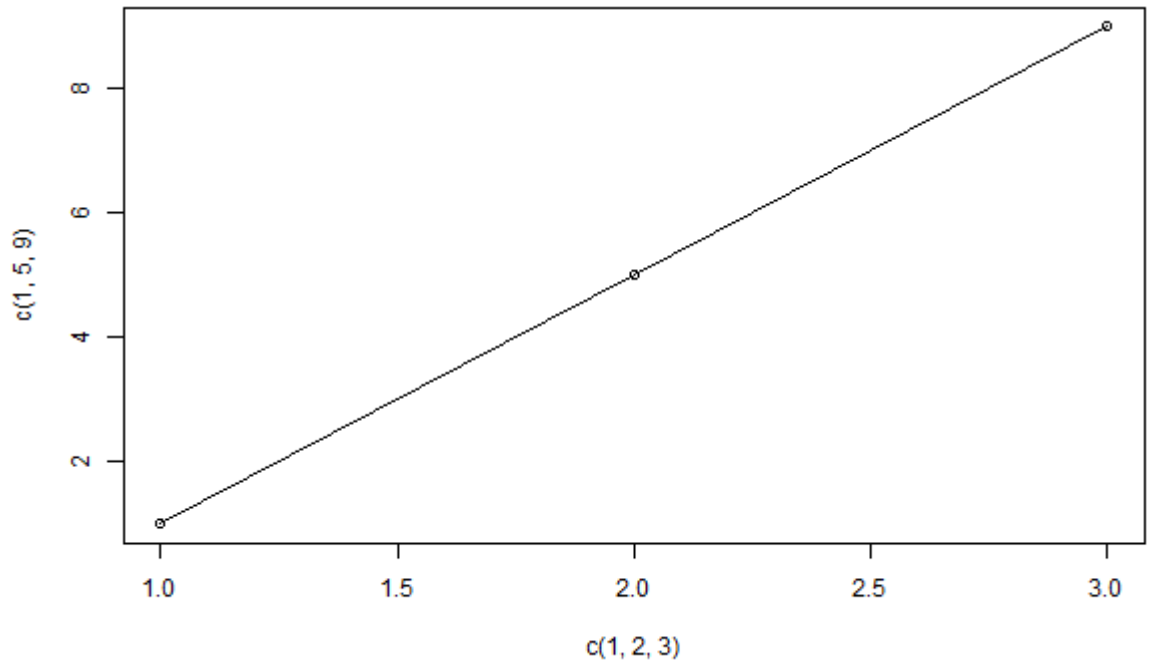
4.3 Saving Graphs to Files

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.

By default, all the graphs are displayed on the screen. We can save the graphs with special functions in R.

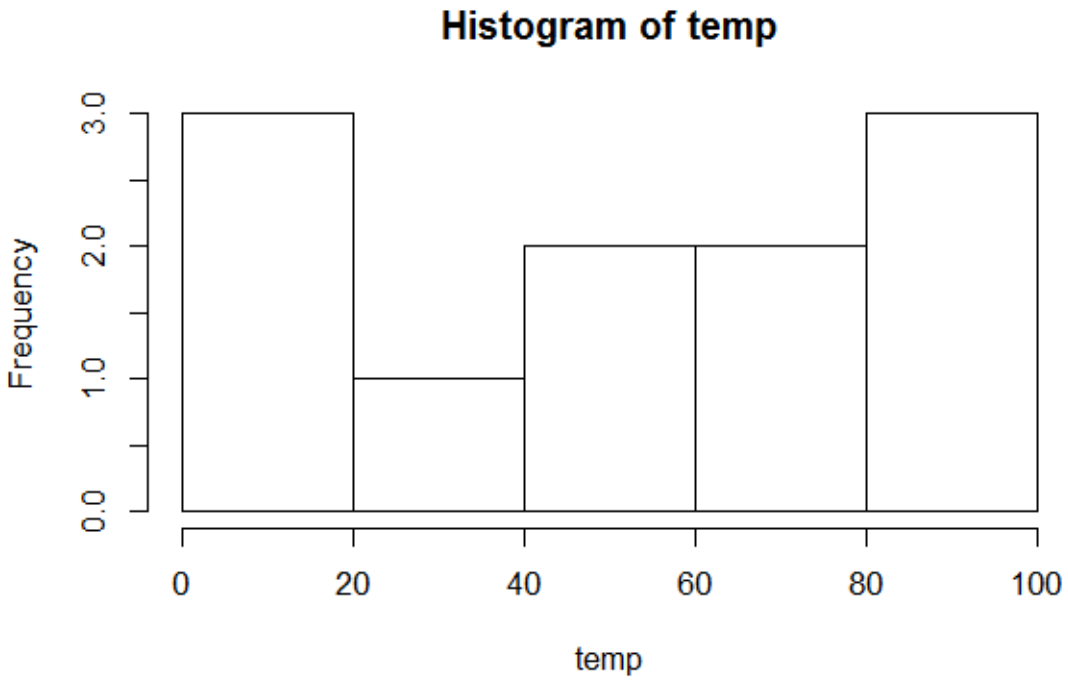
1. To save graph as .png format

```
> png(file="plot1.png", width=600, height=400)
> plot(c(1,2,3),c(1,5,9), type="o")
> dev.off()
null device
          1
>
```



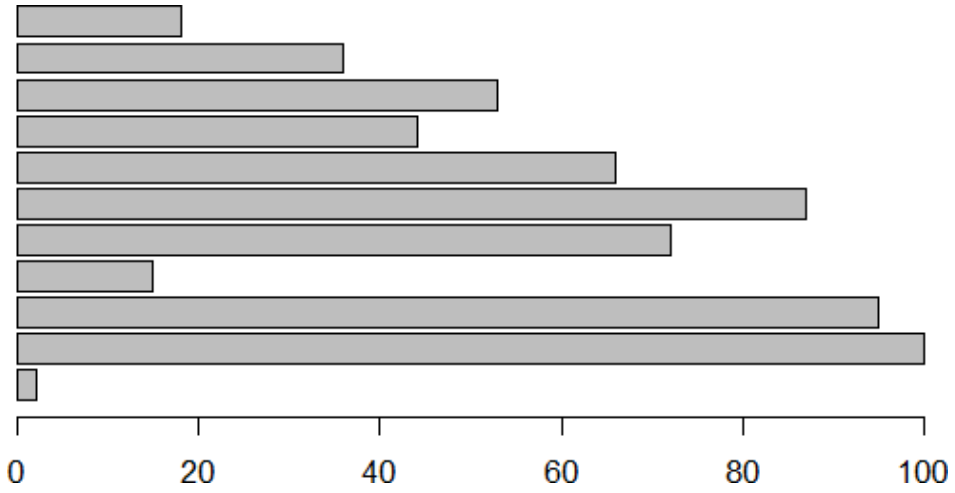
2. To save graph as .bmp format

```
> bmp(file="hist1.bmp", width=6, height=4, units="in",
res=100)
> hist(temp)
> dev.off()
null device
          1
>
```



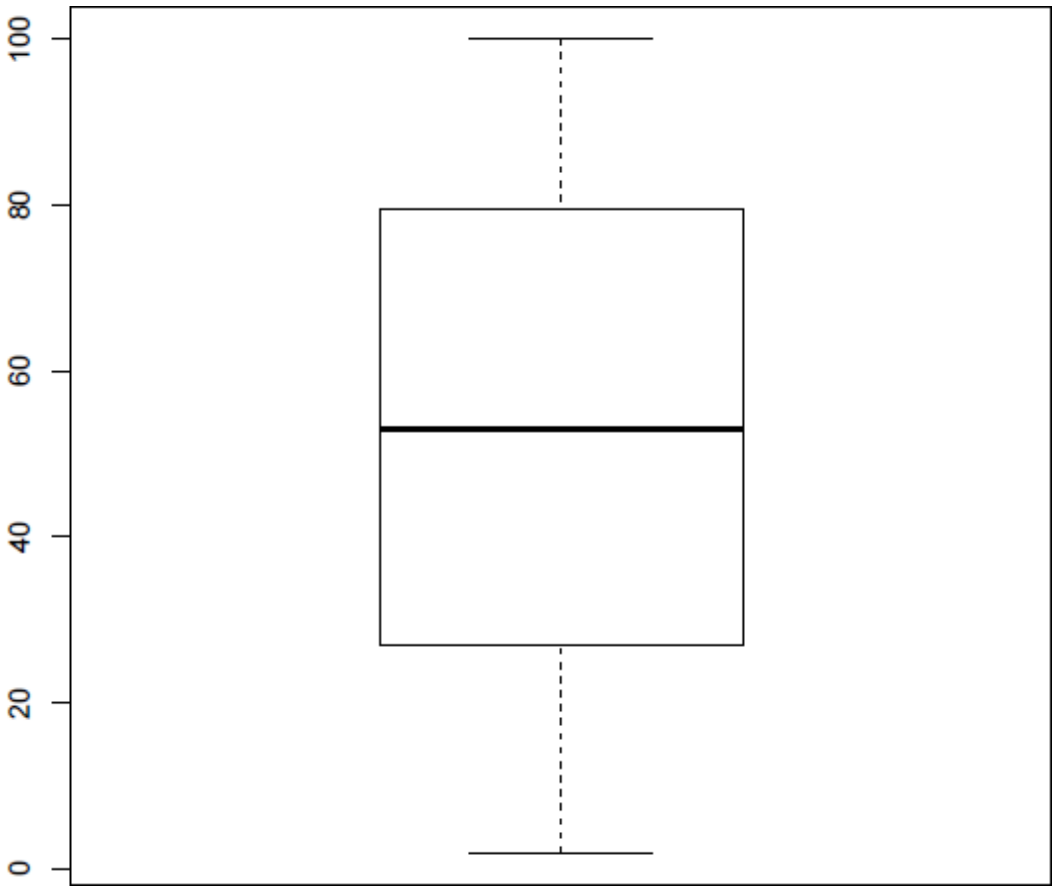
3. To save graph as .tiff format

```
> tiff(file="barplot1.tiff", width=6, height=4, units="in",
res=100)
> barplot(temp,horiz=TRUE)
> dev.off()
tiff
   2
>
```



4. To save graph as .pdf format

```
> pdf(file="boxplot.pdf")
> boxplot(temp)
> dev.off()
tiff
   2
```



4.3 R – Colors

Colors in R can be specified in three ways

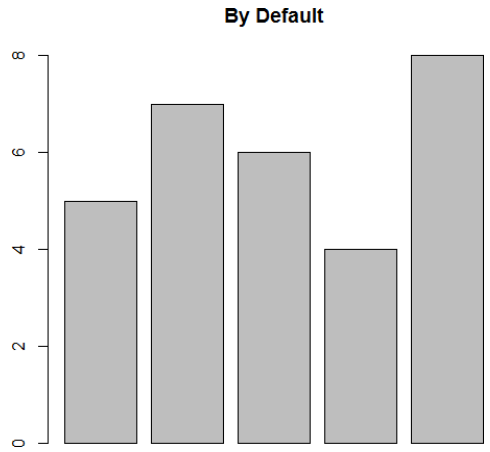
1. Specifying through the names
2. Specifying through the Hexa Decimal values
3. Specifying through the rainbow() function

1. Specifying through the names

We can specify the color through color name. R has 657 colors to specify color name
We can get the color names through colors() function

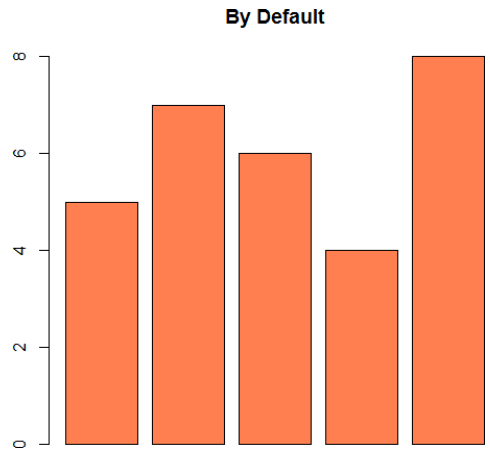
```
> colors()
  [1] "white"           "aliceblue"       "antiquewhite"
  [4] "antiquewhite1"   "antiquewhite2"   "antiquewhite3"
  [7] "antiquewhite4"   "aquamarine"      "aquamarine1"
 [10] "aquamarine2"     "aquamarine3"     "aquamarine4"
 [13] "azure"           "azure1"          "azure2"
 [16] "azure3"          "azure4"          "beige"
 [19] "bisque"          "bisque1"         "bisque2"
 [22] "bisque3"         "bisque4"         "black"
```

```
> temp<-c(5,7,6,4,8)
> barplot(temp,main="By Default")
```



Specifying color through col option in barplot() function. Here color name is “coral”

```
barplot(temp,main="By Default", col="coral")
```

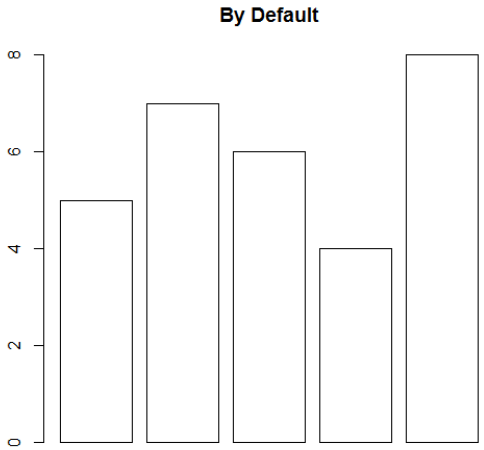


2. Specifying through the Hexa Decimal values

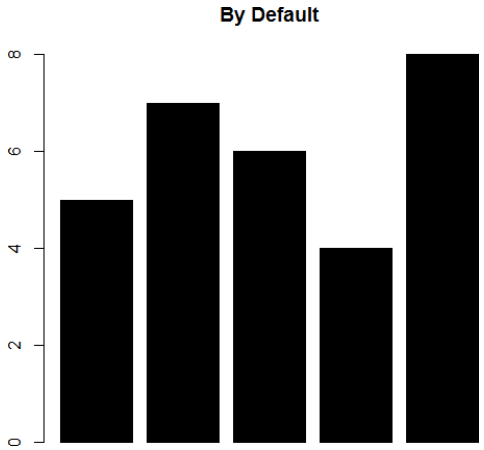
We can also specify the color name through hexadecimal value. The hexadecimal digit number has 6 numerical values followed by #. It is a six hexadecimal digit number of the form #RRGGBB. Here RR stands for RED, GG for GREEN and BB for BLUE.
The permissible range of value is from 00 to FF.
Sample hexadecimal codes for colors
Red = #FF0000
Green = #00FF00
Blue=#0000FF
Black=#000000
White=#FFFFFF

Example

```
> barplot(temp,main="By Default", col="#FFFFFF") #White
```




```
> barplot(temp,main="By Default", col="#000000") # Black
```



3. Specifying through the Color Palette

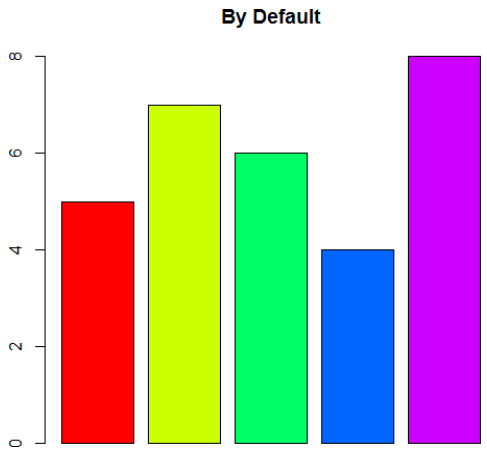
We can apply colors to the diagram through Color Palette
Color Palette has five built- in functions. They are

> rainbow() function
> terrain.colors()
> cm.colors()
> heat.colors()
> colors()

Example

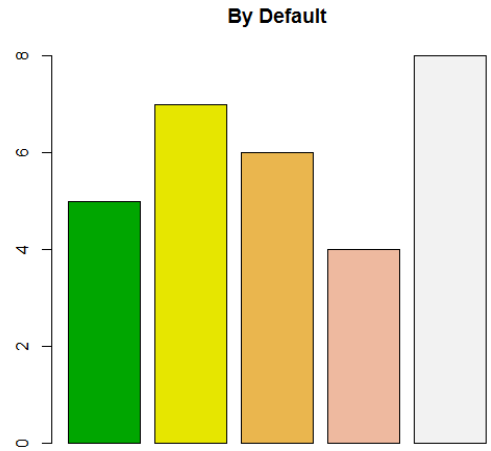
1. rainbow() function

>barplot(temp,main="By Default", col=rainbow(5))



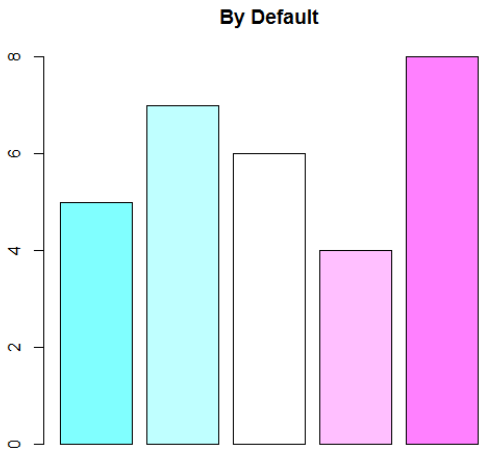
2. terrain.colors()

>barplot(temp,main="By Default", col=terrain.colors(5))



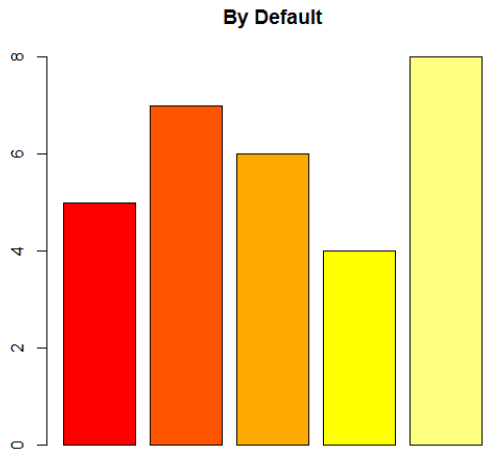
3. cm.colors()

>barplot(temp,main="By Default", col= cm.colors (5))



4. heat.colors()

>barplot(temp,main="By Default", col=heat.colors(5))



5. colors()

>barplot(temp,main="By Default", col=colors(5))