R Programming Tutorials

R is a **programming language** and software environment used for statistical analysis, data modeling, graphical representation and reporting. **R** is best tool for software programmers, statisticians and data miners who looking forward for to easily manipulate and present data in compelling ways.

1. Features

1. R is **platform-independent,** it can run alike on Windows, Mac OSX or Linux, and its **open source**.
2. R is an ***interpreted language,*** which means unlike Java or C you don't need a compiler to develop a program from your code. R take the program and converts it into low-level calls to pre-compiled code/functions.
3. R started with **statistical analysis tool** but after getting interests from programmer's community now you can do **nonstatistical tasks** like graph visualization, data pre-processing, etc.
4. It has effective **data handling** and **storage facilities**.
5. It supports a large pool of **operators** for performing operations on **arrays and matrices**.

2. Usage

a. Machine Learning : R has been used extensively in machine learning. And predictive model creation. It has various package for common ML tasks like linear and non-linear regression, linear and non-linear classification and many more.

b. Statistical Analysis : R is the most common programming language used amongst statisticians. In fact, it was initially built by statisticians for statisticians. It has a rich package repository with more than 9100 packages with every statistical function you can imagine. R's expressive syntax allows researchers – even those from non computer science backgrounds to quickly import, clean and analyze data from various data sources.

c. Data Visualization : Simply explained, R Programming provides a statistician with an extra asset: computer programming skills. Programming languages like R give a professionals superpowers that allow them to collect data in realtime, perform statistical and predictive analysis, create visualizations and communicate actionable results to stakeholders in a customized manner.

3. Concepts

i. Getting Started:

DECLARING VARIABLES

A variable provides program (code) the functionality to store or manipulate the data. In R, variables can store all the data types and doesn't need special declaration like we do in C or JAVA. A variable in R can contain letters, numbers, and the dot or underlines. Let us see the valid and invalid scenario of declaring a variable:

Valid Case:

Variable Name	Contains
varname	Letter
var_name var.name	Letter and Underline Letter and Dot
.var_name	Dot, Letter and Underline
var_1 var.1	Letter, Underline and Number Letter, Dot and Number
.var_num1	Dot, Letter, Underline and Number

Table 3.1: Valid Variable Name

Invalid Case:

Variable Name	Reason
_varname	Cannot start with underline
4varname	Cannot start with number
.4var_name	Dot is followed by number
var$name	No special character ($) is allowed Only Dot and Underline are allowed
1234	Cannot contain only numbers

Table 3.2: Invalid Variable Name

OPERATORS

We have the following types of operators in R Programming:-

1. Arithmetic Operators
2. Relational Operators
3. Assignment Operators
4. Logical Operators
5. Miscellaneous Operators

1. Arithmetic Operators

Operators that involves arithmetic operations like addition, subtraction, multiplication, division, etc are termed as **Arithmetic Operators.**

(a **+** b) **#Addition**

(a **-** b) **#Subtraction**

(a **/** b) **#Divide**

(a ***** b) **#Multiplication**

(a **%%** b) **#Modulus or Calculating Remainder**

(a **%/%** b) **#Calculating Quotient**

(a **^** b) **#a raise to power b**

2. Relational Operators

Operators that involves finding relations like greater than, less than, equal to, not equal to, etc. between two objects are termed as **Relational Operators.**

(a **>** b) **#a greater than b, if yes print TRUE else FALSE**

(a **<** b) **#a less than b, if yes print TRUE else FALSE**

(a **==** b) **#a equals to b, if yes print TRUE else FALSE**

(a **<=** b) **#a less than equal to b, if yes print TRUE else FALSE**

(a **>=** b) **#a greater than equal to b, if yes print TRUE else FALSE**

(a **!=** b) **#a not equal to b, if yes print TRUE else FALSE**

3. Assignment Operators:

Left Assignment

d <- c(0, 7, TRUE, 4i)
d <<- c(0, 7, TRUE, 4i)
d = c(0, 7, TRUE, 4i)

Right Assignment

c(0, 7, TRUE, 4i) -> f
c(0, 7, TRUE, 4i) ->> f
c(0, 7, TRUE, 4i) = f #Invalid Assignment

Exercise:

Interest Rate Comparison

Given,
P = 1000 (Principal amount)
N = 2 (No. of years)
RSimple = 15 (Simple Interest)
RCompound = 9 (Compound Interest)

Formulas:

Define variable **simple** and assign **P(1+RSimple*N)** value to it.
Define variable **complex** and assign $\mathbf{P(1+RSimple)^N}$ value to it.

Questions:

1. **Using arithmetic operators calculate simple and compound variables.**
2. **Using logical operators, find which one is greater, simple or compound?**
3. **Change value of RSimple = 10 and repeat question 2**

4. Logical Operators

(d **&** e) **#Element-wise Logical AND Operator: Compares corresponding element of the vectors.**
(d **&&** e) **#AND Operator: Compares only first element from both the vectors.**

(**!**d) **#Logical NOT Operator**

(d **|** e) **#Element-wise Logical OR Operator: Compares corresponding element of the vectors.**
(d || e) **#OR Operator: Compares only first element from both the vectors.**

5. Miscellaneous Operators

z <- 7:42 **#Colon Operator: Sequence of numbers**

x <- 10
y <- 50

(x **%in%** z) **#Belongs to Operator: returns TRUE if vector one is present in vector two else FALSE**
(y **%in%** z)

DECISION MAKING

A decision making statement consists of:

1. **Condition** (Mostly boolean (TRUE or FALSE) but can be arithmetic as well (x > 5 or x != 10))
2. **Conditional Code** (Depending upon the outcome which codes need to be executed)

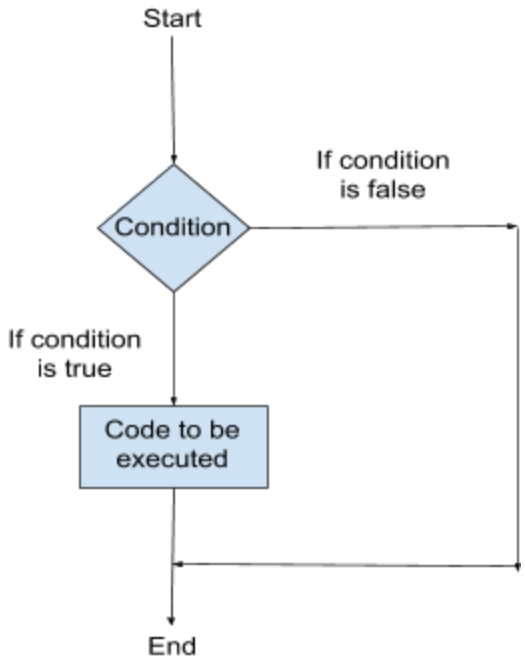

Fig 3.1: Flow Diagram for Decision Making Condition

Types of decision-making statements:

1. **IF Statement:** An **IF** statement consists of a condition followed by just one block of code to be executed.
   Syntax:
   if(condition( )){
   //block of code to be executed
   }
   Example:
   x <- 5
   if(x > 5){
   print("X is greater than 5")
   }

2. **IF...ELSE Statement:** An **IF...ELSE** Statement can consists of one or more conditions followed by one or more block of code to be executed depending on the number of else if statements.

   Syntax:

```
if(condition( )){
//block of code to be executed
} else {
//block of code to be executed
}
```

   Example 1:

```
x <- 4
if( x > 5){
print("X is greater than 5")
} else {
print("X is less than 5")
}
```

   Example 2:

```
x <- 4
Y <- 11

if( x > 5 & y <10) {
        print("X is greater than 5")
        print("Y is less than 5")
} else if( x < 5 & y <10) {
        print("X is less than 5")
        print("Y is less than 10")
} else if( x < 5 & y >10) {
        print("X is less than 5")
        print("Y is greater than 10")
}
```

3. Switch Statement: To have good code readability, we can use **Switch** Statement instead of **IF...ELSE** Statement.

Syntax: switch(expression, case1, case2, case3....

Properties of Switch case:

1. The expression can be either an integer, character or string. When the expression is an integer, the value would be used as a key implicitly. When expression is as a character or string, the value of the expression should be used as key included in the case code.

For Example:

```
switch_code <- switch(1,
                      {
                        print("first")
                        print("second")
                      },
                      {
                        x <- 6

                        if (x > 5) {
                          print("X is greater than 5")
                        }
                      },
                      3,
                      "fourth")
```

Character or String as Expression

```
feature <- switch("name",
                  "name" = "Master Bridge",
                  "address" = "Vile Parle West",
                  "team" = 10)
```

2. If there is more than one match, the first matching element is returned.
3. No Default argument is available.
4. In the case of no match, NULL value is returned.

Exercise:
Calculator using switch case
UI design:

```
"Select operation.”
"1.Add"
"2.Subtract"
"3.Multiply"
"4.Divide"
```

Input:

1. Operation number
2. First Number
3. Second Number

Write a program to add, subtract, multiply and divide using switch case in R.
Hint: To take user inputs, use something like this
choice = as.integer(readline(prompt="Enter choice[1/2/3/4]: "))

LOOPS

In programming, there comes a time wherein you use your intellectual to write code for repetitive tasks, say thanks to **LOOPS.** Loops make sure you write your code once and let the machine take care of the execution of the repetitive tasks to be completed.

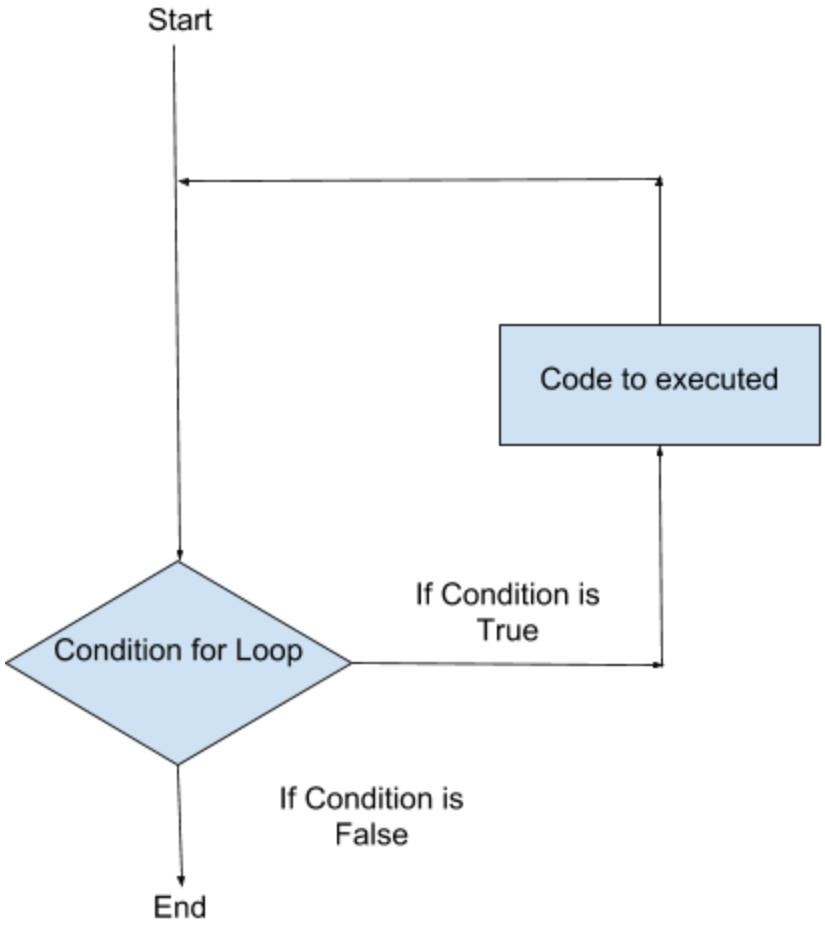

Fig 3.2 Flow Diagram for Loop Condition

Types of loops:

1. Repeat loop: Repeat loop repeats the code till the time the condition is not met. Remember, always check for stopping condition otherwise the program would run into infinite loop.

Syntax:

```
repeat {
  commands
  if(condition){
    break
  }
}
```

Example:

```
i <- 1
repeat {
  print("Hello")
  i <- i+1
  if(i > 100){
    break
  }
}
```

2. While loop: Repeats a statement or block of statements while a given condition is true. It tests the condition before the loop body executes.

Syntax:

```
while (test_expression) {
statement }
```

Example:

```
a <- c(“Hello”, “World”)
count <- 1
while( count < 5) {
  print(a)
  count <- count + 1
}
```

3. For loop: A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

```
for( value in data type){
  commands
}
```

Example:

Integer Example

```
v <- 1:5
for(i in v){
  print(i)
}
```

Character or String Example

```
v <- LETTERS[1:4]
for(i in v){
  print(i)
}
```

Functions

Function or a method is a named callable piece of code which performs some operations or command and optionally returns a value.

Syntax:

Function_name <- **function**(**arg1**, **arg2**,…){
    Function **body**
}

Function consists of:

1. **Function Name** − This is the actual name of the function. R stores function as object in its environment with this name. So, it's easy to keep track of functions you've defined.

2. **Arguments** − An argument is the one which holds in the value when you call a function. They are optional; that is, a function may contain no arguments. Also, in R arguments can have default values as well.

3. **Function Body** − The function body contains a collection of commands that refers to function definition.

4. **Return Value** − The return value of a function is the last command in the function body to be evaluated. It is optional.

Types of Functions:

1. **Built-In Function:** As discussed earlier, *R takes the program and converts it into low-level calls to pre-compiled code/functions,* therefore we have built-in functions to carry out most commonly used calculus.

Example:

sum(4:11) #Prints the summation of series provided
median(4:8) #Prints the median of the series provided
seq(1:100) #Prints the sequence of the numbers provided
print( ) #Print statement

2. **User-defined Function:** Function that are created and defined by the user are called **user-defined functions.** Once created, they can be used like any other built-in functions.

Example:
Write a program to print your name ten times.

```
printNameTenTimes <- function(name){
  for(i in 1:10){
    print(paste(i,name,sep = " -> "))
  }
}

printNameTenTimes("Master Bridge")
```

Different Style of calling a Function:

Function:

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

Call the function by position of arguments: new.function(6,3,10)
Call the function by names of the arguments: new.function(b= 3,a= 10, c=9)

Exercise:

1. In the above mentioned example, introduce one more parameter **n** and call the function **printNameNTimes(name,n).** Print the names **n times.**
2. Change existing program of Calculator and incorporate functions wherever possible.

Strings

Any data written with a pair of single quote or double quotes in R is treated as a String. R represents String with double quotes even if you store it in single quote.

Valid Rules Applied in String Construction

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

Examples:

```
single_quote <- 'Start and end with single quote'
print(a)

double_quote <- "Start and end with double quotes"
print(b)

double_single_quote <- "single quote ' in between double quotes"
print(c)

single_double_quote <- 'Double quotes " in between single quote'
print(d)
```

Invalid Rules Applied in String Construction

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

Examples:

```
e <- 'Mixed quotes"
print(e)

f <- 'Single quote ' inside single quote'
print(f)

g <- "Double quotes " inside double quotes"
print(g)
```

String Manipulation
1. Concatenation:

For concatenating strings - **paste( )** function is used.

FORMULA = STRING1 + STRING2

In R, any number of strings can be combined with **paste( ).**
Note: It converts vectors into String and then concatenates it. (**Important**!!)

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

where,
… = strings to be concatenate.
sep = separator between the string. It is optional.
collapse = eliminate the space in between two strings within a list. It doesn't eliminate space between two words in a string.

2. Substring:

For extracting substring from a string - **substring( )** function is used.

FORMULA = ST**RING**

Syntax:
substring(x, first, last)

where,
x = character vector input.
first = position of the first character to be extracted.
last = position of the last character to be extracted.

3. Upper Case and Lower Case:

For changing characters to upper or lower case , **toupper( ) and tolower( )** functions are used respectively.

FORMULA = **UPPER and lower**

Syntax:
toupper(x)
tolower(x)

where,
x = vector input

VECTORS

Vectors are the most commonly used data types in R. There are different types of Vectors **(Single Element)**:

1. **Character**
2. **Double**
3. **Integer**
4. **Logical**
5. **Complex**
6. **Raw**

Multiple Element Vectors

Using colon operator with any numeric data.

Examples:

```
v <- 1:6
print(v)

v <- 1.6:10.6
print(v)

v <- 2.4:11.6
print(v)
```

Output:

```
[1] 1 2 3 4 5 6
[1]  1.6  2.6  3.6  4.6  5.6  6.6  7.6  8.6  9.6 10.6
[1]  2.4  3.4  4.4  5.4  6.4  7.4  8.4  9.4 10.4 11.4
```

```
#Create vector with elements from 1 to 100 incrementing by 10.
print(seq(1, 100, by = 10))
```

Output:

```
[1]  1 11 21 31 41 51 61 71 81 91
```

Vector Arithmetics

1. **Addition**
2. **Subtraction**
3. **Multiplication**
4. **Division**

Vector Element Sorting

```
element_vector <- c(1,4,9,2,5,19,8,34,-8,0,3)

# Sort the elements of the vector.
sort_result <- sort(element_vector)
print(sort_result)

# Sort the elements in the reverse order.
revsort_result <- sort(element_vector, decreasing = TRUE)
print(revsort_result)

# Sorting character vectors.
char_vector <- c("Z","A","Y","v", "a", "blackbuck")
sort_char_result <- sort(char_vector)
print(sort_char_result)

# Sorting character vectors in reverse order.
revsort_char_result <- sort(char_vector, decreasing = TRUE)
print(revsort_char_result)
```

Lists

Lists are the R objects which contain elements of different types like – numbers, strings, vectors or another list inside it.

Creating a list

For creating a list, we use **list( )** function.

Syntax:

list_data <- **list(**numbers, strings, vectors,... list)

Naming a list

```
company_data <- list("Master Bridge", 3, c("Borivali","Dadar","Vile Parle"))
names(company_data)
```

Accessing a list

#Adding a record to the existing list

```
company_data[4] <- "Monday - Sunday"

names_list <- names(company_data)
names_list[4] <- "Open"
names(company_data) <- names_list

print(company_data)
```

#Deleting a record from the existing list

```
company_data[4] <- NULL
print(company_data)
```

#Updating a record from the existing list

```
company_data[2] <- 5
print(company_data)
```

Exercise:

- Create a list named s containing sequence of 10 capital letters, starting with ‘D’.
- Create a vector named v which contains 100 random integer values between -1000 and +1000.

Matrices

Matrix are one of the data types in R which stores elements in two-dimensions (2D) rectangular layouts. Though you can store numeric, characters or logical (True/False) data in matrices, only numeric data is used in matrix manipulation.
A matrix is created by the function **matrix( ).**

Syntax:

```
matrix(
  data = NA,
  nrow = 1,
  ncol = 1,
  byrow = FALSE,
  dimnames = NULL
)
```

where,
data = input vector,
nrow = number of rows,
ncol = number of columns,
byrow = TRUE if the input vector elements needs to be arranged in row, else FALSE,
dimname = Names assigned to the rows and columns.

1. Creating and Naming a Matrix:

```
simple_matrix <- matrix(
  c(1, 2, 3, 4, 5, 6),
  nrow = 3,
  ncol = 2,
  byrow = TRUE,
  dimnames = list(c("row1", "row2", "row3"), c("col1", "col2"))
)
```

2. Accessing a Matrix:

#Print 1st Row and 2nd Column

```
print(simple_matrix[1,2])
```

#Print 3rd Row and 1st Column

```
print(simple_matrix[1,2])
```

#Print 2nd Row

```
print(simple_matrix[2,])
```

#Print 1st Column

```
print(simple_matrix[,1])
```

3. Matrix Arithmetics

1. Addition: matrix1 + matrix2
2. Subtraction: matrix1 - matrix2
3. Multiplication: matrix1 * matrix2
4. Division: matrix1 / matrix2

Arrays

Arrays are the only data types in R for storing elements in multi-dimensions. It can store data types like number, character or logic. An array is created by function **array( )** and **dim** parameter takes care of multi-dimensionality.

Syntax:

```
array(
  data,
  dim = (nrows, ncols, nmatrices),
  dimnames = c(row.names, col.names, matrix.names)
```

where,
data = input vectors,
dim = provides multi-dimensionality to the array,
dimnames = Names assigned to the rows, columns and matrices.

1. Creating and Naming an Array:

```
a <- c(1, 2, 3)
b <- c(4, 5, 6, 7, 8, 9)
column.names <- c("COL1", "COL2", "COL3")
row.names <- c("ROW1", "ROW2", "ROW3")
matrix.names <- c("Matrix1", "Matrix2")

simple_array <-
  array(
    c(a, b),
    dim = c(3, 3, 2),
    dimnames = list(row.names, column.names,
                    matrix.names)
)
```

2. Accessing an Array:

 # Print the second row of the second matrix of the array.

```
print(simple_array[2,,2])
```

 # Print the element in the 1st row and 1st column of the 1st matrix.

```
print(simple_array[1,1,1])
```

 # Print the 2nd Matrix.

```
print(simple_array[,,2])
```

3. Manipulating Array Elements:

 # Create two vectors of different lengths.

```
d <- c(5,9,3)
e <- c(10,11,12,13,14,15)
```

 # Take these vectors as input to the array.

```
array1 <- array(c(d,e),dim = c(3,3,2))
```

 # Create two vectors of different lengths.

```
a <- c(1, 2, 3)
b <- c(4, 5, 6, 7, 8, 9,10,11,12)
```

 # Take these vectors as input to the array.

```
array2 <- array(c(a,b),dim = c(3,3,2))
```

 # create matrices from these arrays.

```
matrix1 <- array1[,,2]
matrix2 <- array2[,,2]
```

 # Add the matrices.

```
result <- matrix1+matrix2
print(result)
```

Apply Function:

Calculations across the elements of an array can be achieved using the **apply( )** function.

Syntax:

apply(x, margin, fun)

where,
x = an array, including a matrix,
margin = a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. Where X has named dimnames, it can be a character vector selecting dimension names.
fun = the function to be applied like +, %*%, etc.

Example:

```
#Calculate row sums
apply(simple_array, 1, sum)

#Calculate col sums
apply(simple_array, 2, sum)
```

Exercise

Compute row and column sums for a matrix

Data Frame

A data frame is a table in which each column contains values of one feature and each row may or maynot contain one set of value from each column.

Following are the characteristics of a data frame:

- The row names should be unique.
- The column names should be defined.
- The data stored in a data frame can be of numeric, factor or character type.
- Dimensionality should be maintained across the table.

Syntax:

data.frame(vector1, vector2, vector3,..)

1. Creating a Data Frame:

```
#Create a data frame
company.data <- data.frame(
  company_id = c (1:5),
  comapny_name = c("Facebook", "Amazon", "Apple", "Netflix", "Google"),
  company_capital = c(523.3, 615.2, 911.0, 329.0, 843.25),
  company_start_date = as.Date(
    c(
      "2004-02-04",
      "1994-09-23",
      "1980-11-15",
      "1998-05-11",
      "1997-03-27"
    )
  ),
  stringsAsFactors = FALSE
)
```

 # Print the data frame
print(company.data)

#Print the structure of data frame (Data types used in the data frame)
str(company.data)

#Print the summary of data frame (Statistical Analysis of the data frame)
summary(company.data)

2. Naming a data frame:

Row Names and Column Names:
#Retrieve values of rows in the data frame
rownames(company.data)

#Change the row names
rownames(company.data) <- c("A","B","C","D")
rownames(company.data) <- c(1,2,3,4,5)

#Retrieve values of columns in the data frame
colnames(company.data)

#Change the row names
temp <- company.data$company_name
colnames(company.data) <- c("A","B","C","D")
colnames(company.data) <- temp

3. Accessing a data frame:

#Print Company Name and Capital - Specific Columns
company_details <-
  data.frame(Name = company.data$company_name,
             Capital = company.data$company_capital)
print(company_details)

#Access First Row
company.data[1,]

#Access First Column
company.data[,1]

#Access Multiple rows and columns
company.data[c(2,4),c(2,3)]

#Adding Columns - Same number of rows
cbind(data.frame, new.data.frame)
Or
data.frame$feature <- c(value1, value2, value3, …)

#Adding Rows - Same number of columns
rbind(data.frame, new.data.frame)

Exercise
1. Add 5 rows to the existing company.data
2. Add 1 column to the existing company.data
3. Change the Row to Company Names.

Packages

R packages consists of:

1. Compiled code,
2. R functions and
3. Sample data.

They are stored under a directory called "library" in the R environment. R installs some a set of default packages during installation.

To use additional packages, you can either:

1. Install directly from CRAN (Comprehensive R Archive Network)
2. Load Package to Library

Search Installed Packages: search( )

1. Install From CRAN

install.packages("**Package_Name**")

2. Install from Downloaded Zip, Tar or Gzip

```
install.packages(
  Path_to_downloaded_file,
  repos = NULL,
  type = "source"
)
```

Load Packages:
library("Package_Name")
Do search( ) and you can see the new package in the installed packages.
Note: You'll have to do load library using library( ) function, otherwise packages won't be visible in your search( ) function or installed packages.

Verify Package installed:

any(grepl("**Package_Name**", installed.packages( )))

CSV

In R, we can read data from many sources stored outside the R environment like:

- **CSV**
- **Excel**
- **Binary Files**
- **XML File**
- **JSON File**
- **Web Data**
- **Database**

We can also write data into respective files. In this module we will learn to read data from a csv file and then write data into a csv file.
Note: It is preferred if the file should is present in current working directory so that R can read it. We can also set our own directory and read files from there.

1. Reading from CSV

Syntax:
read.csv(filepath, header = TRUE, sep = ",", ...)

where,
filepath = path of the file,
header = a logical value indicating whether the file contains the names of the variables as its first line,
sep = the field separator character.

2. Writing into CSV

In R, we can create csv file from existing data frame. The **write.csv( )** function creates the csv file in the working directory or the path provided by the user.

```
# Write Company data into a new file.
write.csv(company.data,"output.csv",row.names = FALSE)
```

Excel

 # Load the library into R workspace.
library("xlsx")
If not installed

1. Reading from Excel

Syntax:
read.xlsx(filepath, sheetName = "", sheetIndex = "")

where,
filepath = path of the file,
sheetName = Name of the sheet (if you multiple sheets)
sheetIndex = Index of the sheet (if you multiple sheets)

2. Writing into Excel

In R, we can create excel file from existing data frame. The **write.xlsx( )** function creates the excel file in the working directory or the path provided by the user.

 # Write data into a new file.
write.xlsx(data frame, file, sheetName="Sheet1",
col.names=TRUE, row.names=TRUE, append=FALSE, showNA=TRUE)

#Writing a Data frame into Excel
write.xlsx(people_data, "output.xls")

#Ignore or Don't write Row Names
write.xlsx(data, "output.xls", row.names = FALSE)

#Ignore or Don't write Column Names
write.xlsx(data, "output.xls", row.names = FALSE, col.names = FALSE)

XML Files

XML is a file format which shares both:

1. File format and
2. Data

It stands for Extensible Markup Language (XML). It is similar to HTML as it contains markup tags. But unlike HTML where the markup tag describes structure of the page, in **XML** the markup tags describe the meaning of the data contained into the file.

 # Load the library into R workspace.
library("XML")
If not installed,
install.packages("XML")

1. Reading from XML

Syntax:
xmlParse(filepath)

where,
filepath = path of the file

2. XML to Data Frame
To handle the data effectively in large files we read the data in the xml file as a data frame. Then process the data frame for data analysis**.**

 # Convert the XML file to a data frame.
xml_dataframe <- xmlToDataFrame(filepath_XML_File)
print(xml_dataframe)

JSON

1. Read the JSON File

The JSON file is read by R using the function from JSON(). It is stored as a list in R.

```
# Load the package required to read JSON files.
library("rjson")

# Give the input file name to the function.
result <- fromJSON(file = "input.json")

# Print the result.
print(result)
```

2. Convert JSON to a Data Frame

We can convert the extracted data above to a R data frame for further analysis using the as.data.frame() function.

```
# Load the package required to read JSON files.
library("rjson")

# Give the input file name to the function.
result <- fromJSON(file = "input.json")

# Convert JSON file to a data frame.
json_data_frame <- as.data.frame(result)

print(json_data_frame)
```

R - Databases

In R, you can connect to different databases using different packages:

1. The RODBC package provides access to Microsoft Access and Microsoft SQL Server databases through an ODBC interface.
2. The RMySQL package provides an interface to MySQL.
3. The ROracle package provides an interface for Oracle.
4. The RJDBC package provides access to databases through a JDBC interface.

RODBC:

install.packages("RODBC")
library("RODBC")

#Load connection settings with server and database name

connection <- odbcDriverConnect('driver={SQL Server}; server = server_name; database = database_name; trusted_connection=true')

#Read a table from an ODBC database into a data frame

sqlFetch(connection, table_name)

#Submit a query to an ODBC database and return the results

sqlQuery(connection, query)

#Write or update (append=True) a data frame to a table in the ODBC database

sqlSave(connection, df_name, tablename = table_name, append = FALSE)

#Remove a table from the ODBC database

sqlDrop(connection, table_name)

#Close the connection

close(connection)

R - Pie Charts

R Programming language has numerous libraries for data visualization. Pie-chart is one of the most common chart used to represent data. A pie-chart is a representation of data as portion of a circle with different colors. The slices are labeled and the numbers corresponding to each slice can be represented either values or percentages 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
pie(x, labels, radius, main, col, clockwise)

Where,
x = input vector containing the numeric values used in the pie chart,
labels = description to the slices,
radius = radius of the circle of the pie chart. (value between −1 and +1),
main = main title of the chart.
col = color palette,
clockwise = logical value indicating if the slices are drawn clockwise or anti clockwise.

Different Types of Pie-Chart:

1. Pie-Chart with default color and name
2. Pie-Chart with customized color and percentage
3. Pie-Chart with customized color, value and legend

1. Pie-Chart with default color and name

```
# Create data for the graph.
college_admits <- c(100, 162, 210, 113)
colleges_labels <-
  c(
    "Arizon State University",
    "New York University",
    "University of Maryland",
    "Syracuse Unviersity"
  )

# Give the chart file a name.
png(file = "college_admits.jpg")

# Plot the chart with default colors and name
pie(college_admits, colleges_labels)

# Save the file.
dev.off()
```

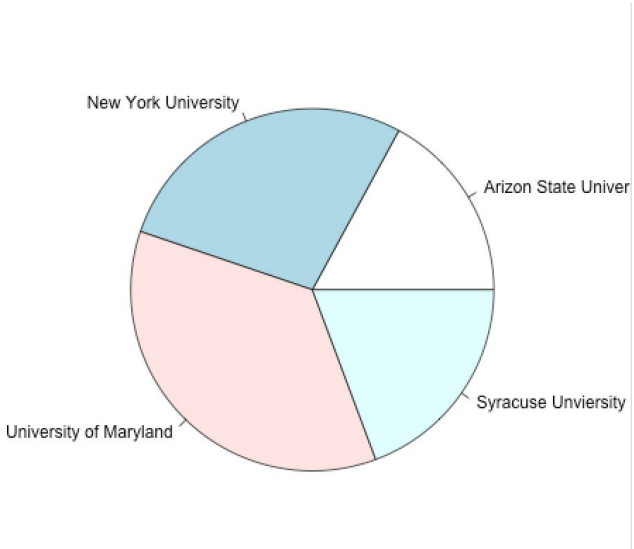

Fig 3.3 Pie-Chart with default Color and Name

2. Pie-Chart with customized Color and Percentage

```
# Create data for the graph.
college_admits <- c(100, 162, 210, 113)
colleges_labels <-
  c(
    "Arizon State University",
    "New York University",
    "University of Maryland",
    "Syracuse Unviersity"
  )

admit_percentage <- 100*college_admits/sum(college_admits)
admit_percentage <- round(admit_percentage,1)

# Give the chart file a name.
png(file = "college_admits_color.jpg")

#Plot the chart with colors and percentage
pie(college_admits, admit_percentage, col = rainbow(length(college_admits)))

# Save the file.
dev.off()
```

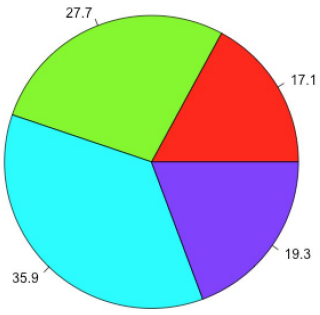

Fig 3.4 Pie Chart with customized Color and Percentage

2. Pie-Chart with customized Color, Value and Legend

```
# Create data for the graph.
college_admits <- c(100, 162, 210, 113)
colleges_labels <-
  c(
    "Arizon State University",
    "New York University",
    "University of Maryland",
    "Syracuse Unviersity"
  )

# Give the chart file a name.
png(file = "college_admits_color_legend.jpg")

#Plot the chart with color, value and legend
pie(college_admits, college_admits, col = rainbow(length(college_admits)))
legend("topright",
       c(
         "Arizon State University",
         "New York University",
         "University of Maryland",
         "Syracuse Unviersity"
       ),
       cex = 0.8,
       fill = rainbow(length(college_admits)))

# Save the file.
dev.off()
```

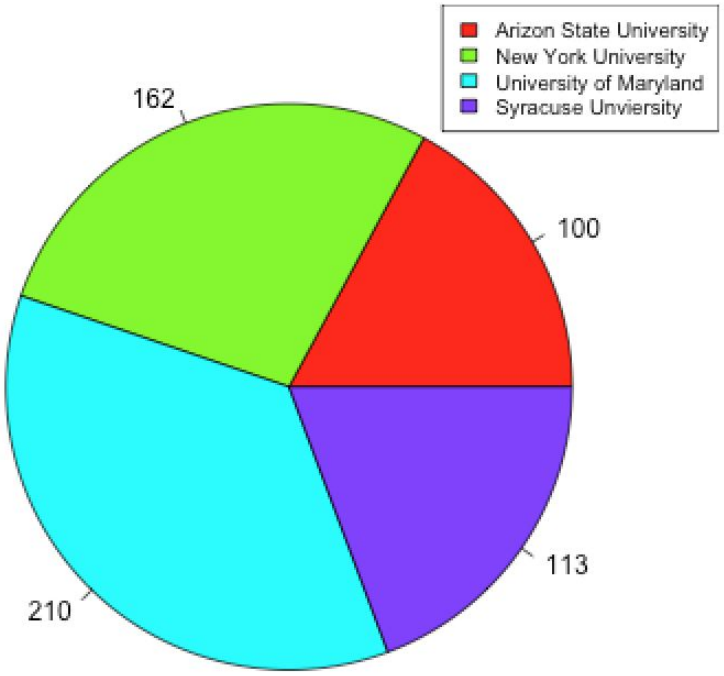

Fig 3.5: Pie-Chart with customized Color, Value and Legend

R - Bar Charts

A bar chart represents data in rectangular bars with length of the bar proportional to the value of the variable. R uses the function **barplot()** to create bar charts. R can draw both Vertical and Horizontal bars in the bar chart. In bar chart each of the bars can be given different colors.

Syntax

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

Where,
H = vector or matrix containing numeric values used in bar chart,
xlab = label for x axis,
ylab = label for y axis,
main = main title of the bar chart,
names.arg = vector of names appearing under each bar,
col = to give colors to the bars in the graph,
horiz = logical value changes orientation to Horizontal if True, otherwise Vertical.

Different Types of Bar Plots:

1. Simple Bar Chart (1-Dimensional)
2. Bar Chart Labels, Title and Colors
3. Group Bar Chart and Stacked Chart

1. Simple Bar Chart (1-Dimensional)

```
# Create the data for the chart
college_admits <- c(100, 162, 210, 113)

# Give the chart file a name
png(file = "college_admits_bar.jpg")

# Plot the bar chart & Save
barplot(college_admits)
dev.off()
```

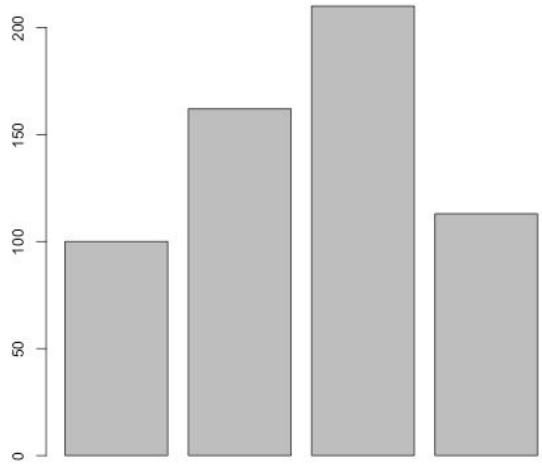

Fig 3.6: Simple Bar Chart

2. Bar Chart Labels, Title and Colors

```
# Create the data for the chart
college_admits <- c(100, 162, 210, 113)
colleges_labels <-
  c(
    "Arizon State University",
    "New York University",
    "University of Maryland",
    "Syracuse Unviersity"
  )

# Give the chart file a name
png(file = "college_admits_color_bar.jpg")

# Plot the bar chart & Save
barplot(
  college_admits,
  names.arg = colleges_labels,
```

```
  xlab = "Colleges",
  ylab = "Admits",
  col = "blue",
  main = "College Admit chart",
  border = "red"
)
dev.off()
```

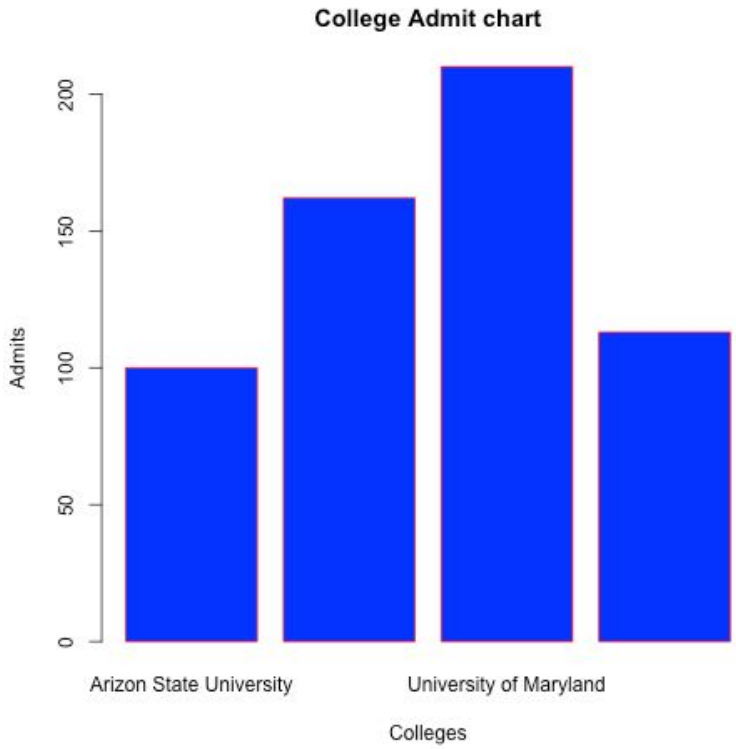

Fig 3.7: Bar Chart Labels, Title and Colors

3. Group Bar Chart and Stacked Bar Chart

```
# Create the input vectors.
colleges_labels <-
  c(
    "Arizon State University",
    "New York University",
    "University of Maryland",
    "Syracuse Unviersity"
```

```
  )
colors = c("green","orange","brown","blue")
regions <- c("East","West","North","South")

# Create the matrix of the values.
Values <- matrix(c(2,9,3,11,9,4,8,7,3,12,5,2,8,10,11,16), nrow = 4, ncol = 4, byrow = TRUE)

# Give the chart file a name
png(file = "barchart_stacked.png")

# Create the bar chart
barplot(Values, main = "College Admits", names.arg = colleges_labels, xlab = "Colleges", ylab = "Admits", col = colors)

# Add the legend to the chart
legend("topleft", regions, cex = 1.3, fill = colors)

# Save the file
dev.off()
```

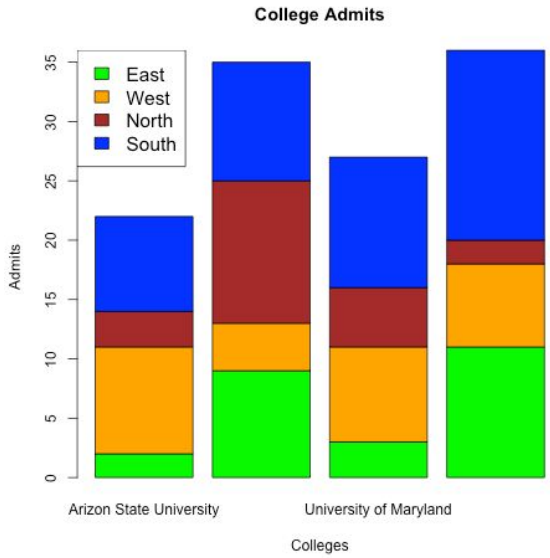

Fig 3.8: Stacked Bar Chart

R - Line Graphs

A line chart is a graph that connects a series of points by drawing line segments between them. These points are ordered in one of their coordinate (usually the x-coordinate) value. Line charts are usually used in identifying the trends in data.

The **plot()** function in R is used to create the line graph.

Syntax

plot(v,type,col,xlab,ylab)

Where,
v = vector containing the numeric values,
type = "p" to draw only the points,
"l" to draw only the lines and
"o" to draw both points and lines,
xlab = label for x axis,
ylab = label for y axis,
main = main title of the chart,
col = used to give colors to both the points and lines.

Different Types of Line Graphs:

1. Simple Line Graph
2. Color Line Graph with Labels and Title
3. Multiple Line in a Line Graph

1. Simple Line Graph

```
# Create the data for the chart.
college_admits <- c(100, 162, 210, 303)

# Give the chart file a name.
png(file = "college_admits_line.jpg")
```

 # Plot the bar chart.
plot(college_admits,type = "o")

 # Save the file.
dev.off()

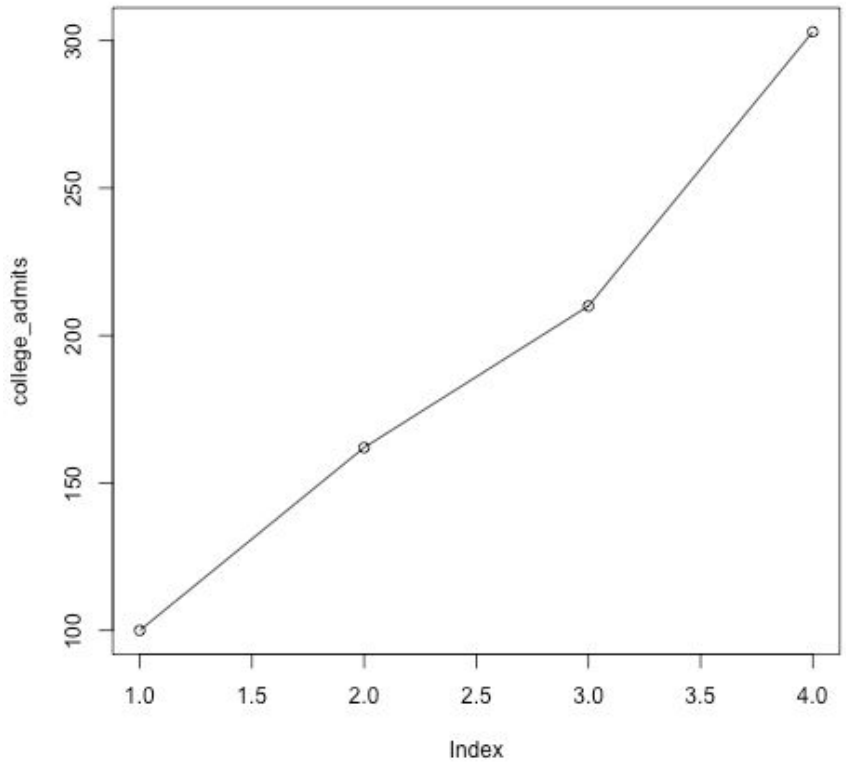

Fig 3.9: Simple Line Chart

2. Colorful Line Graph

 # Give the chart file a name.
png(file = "college_admits_colored_line.jpg")

 # Plot the bar chart.
plot(college_admits ,type = "o", col = "red", xlab = "Year", ylab = "College Admits", main = "College Admits chart")

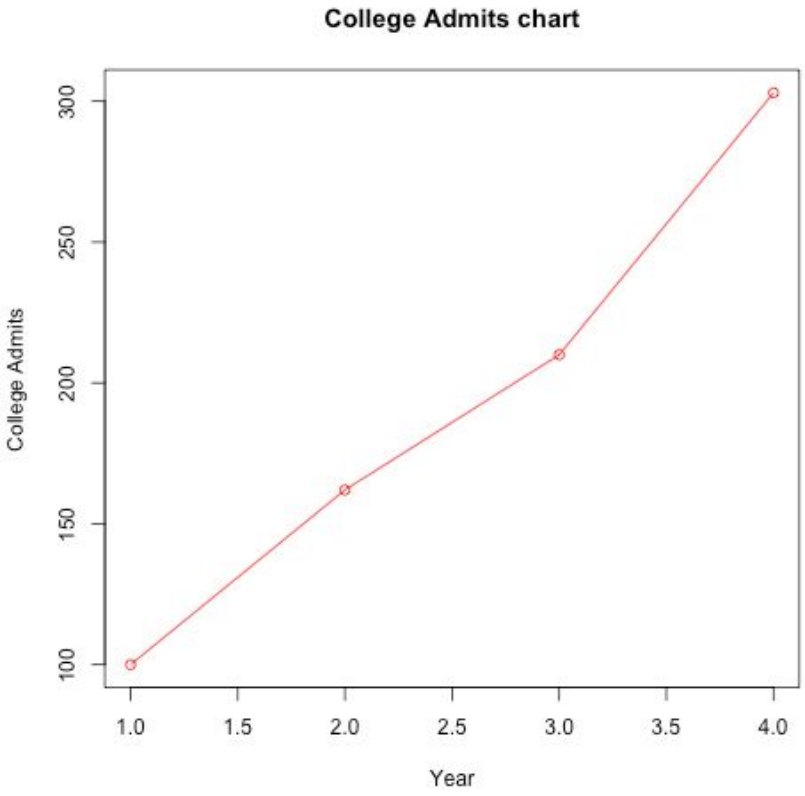

Fig 3.10: Color Line Chart with Labels and Title

3. Multiple Line in a Line Graph:

```
#Add to create data
college_rejects <- c(180, 162, 110, 93)

# Give the chart file a name.
png(file = "college_admits_multiple_line.jpg")

# Plot the bar chart.
plot(college_admits ,type = "o", col = "blue", xlab = "Year", ylab = "College Admits", main = "College Admits chart")
lines(college_rejects, type = "o", col = "red")

# Save the file.
dev.off()
```

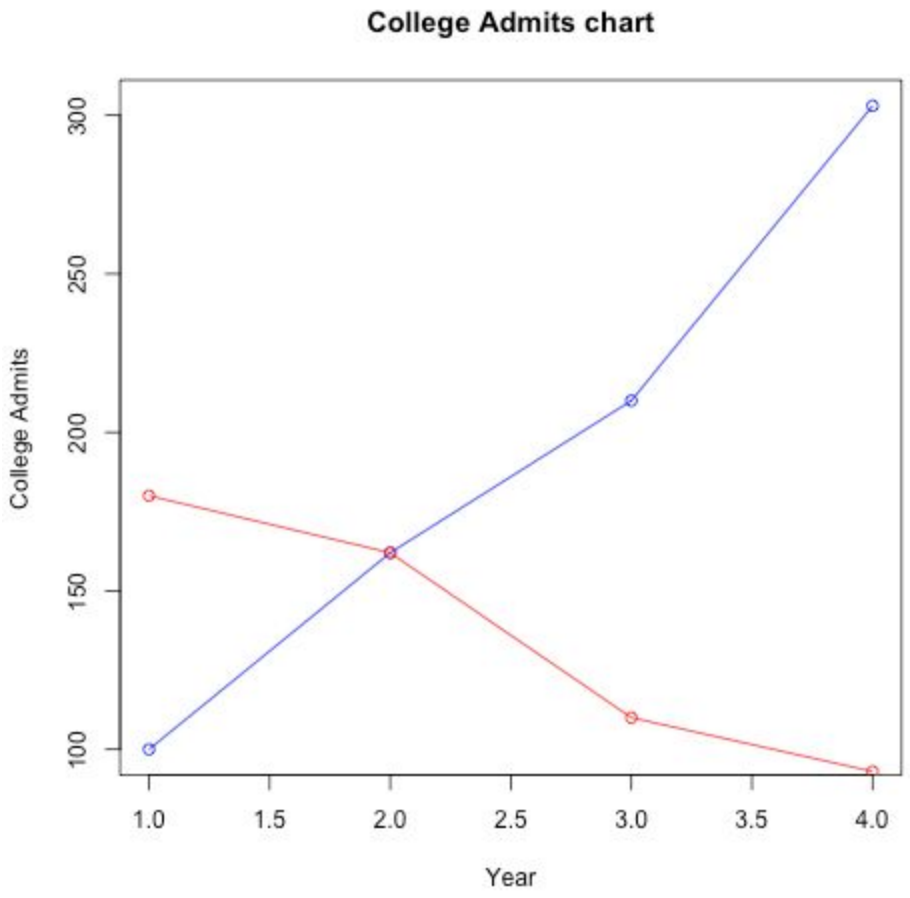

Fig 3.11: Multiple Line in a Line Graph

Activities:

1. **Connect to your local database and read any table into a dataframe.**
2. **Drop some table in the local database from the connection opened in RStudio Session.**
3. **Find correlated data and plot it onto Pie Chart, Bar Chart and Line Chart.**