* . DATA ANALYSIS :

- Data Analysis is a process of obtaining raw data and converting it into information useful for decision-making by users.

* . DATA ANALYTICS :

- Data Analytics is the science of examining the raw data with the purpose of drawing conclusions about the information. (Graphical representation)
- Analytics defines the science behind the analysis.

* . DATA SCIENCE :

- Concepts that are better than software engineering involving more of statistics (mathematics).

* . Data - raw facts
Information - well organized data from collected raw data.

* . Seven characteristics that define data quality are :

1. ✓Accuracy - Conformity / definite at things.
Precision - Exactness of thing.

2. Legitimacy - lawfulness or authenticity.
Validity - validness or accurately useful.

3. Reliability - without fail.
✓Consistency - happen in same / similar way.

4. ✓ Timeliness - Time expectation for accessibility/availability

Relevance - valid data useful at a point of time.

5. ✓ Completeness - state of having / ~~data~~ satisfying requirement

Comprehensiveness - large amount of things.

6. Availability - State of being accesible

Accessibility - available for a moment.

7. Granularity - detailed view of things.

Uniqueness - existing without duplicity.

## *. STEPS INVOLVED IN DATA ANALYSIS :

1. Data Collection - Identify Sources
2. Data Cleaning - Remove noise
3. Data Analysis - Apply Algorithms
4. Data Interpretation - Explain results & Conclude.

## *. DATA MUNGING :

- Required for improving the quality of gathered data.
- Involves cleaning and transformation of messy data available with us.
- Also refered as Data Wrangling.

## *. REASONS for NOISE DATA :

- Data in real world is Dirty.
- lots of potentially incorrect data → Faulty instruments
  → Human / Computer error
  → Transmission error

## *. DATA CLEANING :

- Filling in missing values.
- Smooth noisy data.
- Identify or remove outliers.
- Resolve inconsistencies.

⇒ Ignore the tuple (rows).

⇒ Fill in missing value manually.

## *. Dealing with Noise :

- BINNING :

→ First sort data and partition into (equal-frequency) bins.

→ Then one can smooth by bin means, smooth by bin median, smooth by bin boundaries, etc...

- REGRESSION :

→ Smooth by fitting the data into regression functions.

## *. DATA TRANSFORMATION :

- Transformation is mapping the data to a new space.

Ex : Fourier Transform, Wavelet Transform.

## *. DATA SAMPLING :

- Sampling : obtaining a small sample s to represent the whole data set N.

- Types of Sampling :

→ Simple random sampling : There is an equal probability of selecting any particular item.

→ Stratified sampling : Partition the data set, and draw samples from each partition (proportionally i.e., approx the same percentage of the data.)

## *. MEASURES OF CENTRALITY :

- Mean = $\frac{\text{sum of observations}}{\text{no. of observations}} = \frac{\sum x_i}{n}$

- Mode = most frequent value in data. (may be multiple)

- Median = $\begin{cases} \text{Middle most value}, & n - \text{odd} \\ \text{average of middle most values}, & n - \text{even} \end{cases}$

$$\boxed{\text{Standard deviation} = \sqrt{\text{Variance}}} *$$

- Range = Highest observation − Lowest observation
- St. dev = $\begin{cases} s = \sqrt{\dfrac{\Sigma (x_i - \bar{x})^2}{n-1}} & \rightarrow \text{sample} \\ s = \sqrt{\dfrac{\Sigma (x_i - \bar{x})^2}{n}} & \rightarrow \text{population} \end{cases}$
- variance, $\begin{cases} s^2 = \dfrac{\Sigma (x_i - \bar{x})^2}{n-1} & \rightarrow \text{sample} \\ s^2 = \dfrac{\Sigma (x_i - \bar{x})^2}{n} & \rightarrow \text{population} \end{cases}$

*. Smoothing Noisy Data : BINNING

Ex : Sorted data for price : 4, 8, 15, 21, 21, 24, 25, 28, 34

- Partition into (equal − frequency) bins :

Bin 1 : 4, 8, 15

Bin 2 : 21, 21, 24

Bin 3 : 25, 28, 34

- Smoothing by bin means :

Bin 1 : 9, 9, 9

Bin 2 : 22, 22, 22

Bin 3 : 29, 29, 29

- Smoothing by bin boundaries :

Bin 1 : 4, 4, 15

Bin 2 : 21, 21, 24

Bin 3 : 25, 25, 34

## * DATA NORMALIZATION :

### (1) MIN - MAX NORMALIZATION :

- Min-Max Normalization performs a linear transformation on the original data.
- Suppose $min_A$ & $max_A$ are minimum & maximum values of an attribute.
- A min-max normalization maps a value, $v_i$ of A to $v_i'$ in the range $[new\_min_A, new\_max_A]$ by computing,

Normalized value * :

$$v_i' = \frac{v_i - min_A}{max_A - min_A} (new\_max_A - new\_min_A) + new\_min_A$$

Example : Suppose that the minimum and maximum values for the attribute income are $12,000 and $98,000 respectively. We would like to map income to the range [0.0, 1.0]. By min-max normalization, a value of $73,600 for income is transformed to :

$$v_i' = \frac{73,000 - 12,000}{98,000 - 12,000} (1.0 - 0) + 0 = 0.716$$

---

### (2) Z-SCORE NORMALIZATION :

- In Z-Score normalization (or zero-mean normalization), the values for an attribute A, are normalized based on the

mean (i.e. average) and standard deviation of A·A value, $v_i$, of A is normalized to $v_i'$ by computing.

$$v_i' = \frac{v_i - \bar{A}}{\sigma_A}$$

$\bar{A}$ - mean of attribute

$\sigma_A$ - st·dev of attribute

---

## ③ DECIMAL SCALE NORMALIZATION :

- Normalization by decimal scaling normalizes by moving the decimal point of values of attribute A.
- The no. of decimal points moved depends on the maximum absolute value of A.
- A value, $v_i$ of A is normalized to $v_i'$ by computing,

$$v_i' = \frac{v_i}{10^j}$$

, where j is the smallest integer such that $\max(|v_j'|) < 1$.

Ex: Suppose that the recorded values of A range from -986 to 917. The maximum absolute value A is 986. To normalize by decimal scaling, we therefore divide each value by 1000 (i.e. $j = 3$) so that -986 normalizes to -0.986 and 917 normalizes to 0.917.

$-986 \longrightarrow -0.986$, $j = 3$ ✓

$+917 \longrightarrow +0.917$, $j = 3$ ✓

---

(*) Data : Attribute % FAT : 9.5, 26.5, 7.8, 17.8, 31.4,

25.9, 27.4, 27.2, 31.2, 34.6, 42.5, 28.8,

33.4, 30.2, 34.1, 32.9, 41.2, 35.7

$n = 18$

Mean $= \frac{\sum x_i}{n} = \frac{\text{Sum of values}}{18} = \frac{518.1}{18} = 28.783$

Median = 30.7 (even data average)

Mode = no frequently occuring value

Range = (Highest - lowest) = 42.5 − 7.8 = 34.7

variance = 85.563

st. $= \sqrt{\text{variance}} = \sqrt{85.563} = 9.25002$

(*) 23, 23, 27, 27, 39, 41, 47, 49, 50, 52, 54, 54, 56, 57, 58, 58, 60, 61

Dividing into bins of **depth = 3**:

Bin 1 : 23, 23, 27

Bin 2 : 27, 39, 41

Bin 3 : 47, 49, 50

Bin 4 : 52, 54, 54

Bin 5 : 56, 57, 58

Bin 6 : 58, 60, 61

Smoothing by Bin Means:

Bin 1 : 24, 24, 24

Bin 2 : 36, 36, 36

Bin 3 : 49, 49, 49

Bin 4 : 53, 53, 53

Bin 5 : 57, 57, 57

Bin 6 : 60, 60, 60

Smoothing by Bin Medians:

Bin 1 : 23, 23, 23

Bin 2 : 39, 39, 39

Bin 3 : 49, 49, 49

Bin 4 : 54, 54, 54

Bin 5 : 57, 57, 57

Bin 6 : 60, 60, 60

Smoothing by Bin Boundaries:

Bin 1 : 23, 23, 27

Bin 2 : 27, 41, 41

Bin 3 : 47, 50, 50

Bin 4 : 52, 54, 54

Bin 5 : 56, 58, 58 (or) 56, 56, 58

Bin 6 : 58, 61, 61

# MODULE - 2

- R is a programming language and environment commonly used in statistical computing, data Analytics and scientific research.

- If you are trying to analyze a dataset and present the findings in a research paper, then R is probably a better choice than python.

- But if you are writing a data Analysis program that runs in a distributed system & interacts with lot of other components, it would be preferable to work with python.

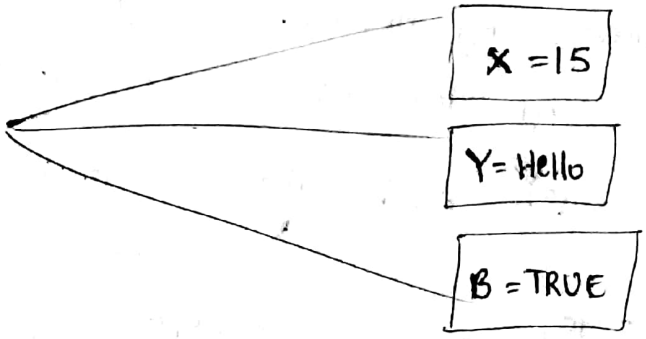
## *. R PROGRAMMING : VARIABLES :

- A variable in R can store Numeric values, Complex values, words, Matrices and even Table.

Ex:

```
x = 15
y ← "Hello"
TRUE → B
```

Memory Blocks




- R is called a dynamically typed language which means that we can change a data type of the same variable again & again when using it in program.

- There are 5 types of Atomic Vectors :

i. Character : 'a', "Hello", 'TRUE'

ii. Complex : 3 + 2i

iii. Integer : 2L, 34L, 0L (only +ve) (negative whole numbers)

iv. Numeric : 5, −3, −0.008, 3.14

v. Logical : TRUE, FALSE

---

*. CAT I CRUX POINTS ?

↔ Comment on which method you choose between min-max Normalization, z-score normalization, decimal scaling to use for given data, giving reasons as to why ?

Ans : ✓ Given the data, one may prefer decimal scaling for normalization as such a transformation would maintain the data distribution and be intuitive to interpret, while still allowing mining on specific datasets.

- Min-max normalization has undesired effect of not permitting any future values to fall outside the current minimum and maximum values without encountering an "out of bounds" error. As it is probable that such values may be present in future data, this method is less appropriate.

- Also, z score transforms values into measures that represent their distance from the mean, in terms of standard deviation. It is probable that this type of transformation would not increase the information value of the attribute in terms of usefulness of mining results.

## * Basic R programming syntax :

① Take user input :

```
n = as.integer(readline(prompt = "Enter n : "))
```

② Find the no. of elements in vector :

```
numEle = length(vectorName)
```

③ Mean : `mean(vectorName)`

④ Median : `median(vectorName)`

⑤ Variance : `var(vectorName)`

⑥ Standard deviation : `sd(vectorName)`

⑦ Define a vector

```
vectorName = c(1, 2, 3, )
```

⑧ Declare & define a list :

```
mylist = list(1, 5, 'a', "abc", -3, 0.5)
```

⑨ Declare vectors & convert to matrix :

```
> mylist = c(7:22)
> rnow = c("row1", "row2", "row3", "row4")
> col = c("Column1", "Column2", "Column3", "Column 4")
> m = matrix(mylist, nrow = 4, dimnames = list(rnow, col))
> print(m)
```

Output :

| | Column1 | Column2 | Column3 | Column4 |
|---|---|---|---|---|
| row1 | 7 | 11 | 15 | 19 |
| row2 | 8 | 12 | 16 | 20 |
| row3 | 9 | 13 | 17 | 21 |
| row4 | 10 | 14 | 18 | 22 |

(10)

```
vector1 = c(5,9,3)

vector2 = c(10,11,12,13,14,15)

result = array(c(vector1, vector2), dim = c(3,3,2))

print(result)
```

(no. of times to display this array)

output :

, , 1

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

, , 2

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

(11) DataFrame :

```
RegNo = c(10, 20, 30, 40, 50, 60)

Names = c("Ramesh", "Rakesh", "Shruti", "Subha", "Kamala", "Isha")

Grades = c(9.3, 5.2, 6.6, 7.8, 8.83, 9.11)

DataFrame1 = data.frame(RegNo, Names, Grades)

names(DataFrame1) = c('Registration Number', 'Names', 'Grades')

Print(DataFrame1)
```

output :

| | Registration Number | Names | Grades |
|---|---|---|---|
| 1 | 10 | Ramesh | 9.3 |
| 2 | 20 | Rakesh | 5.2 |
| 3 | 30 | Shruti | 6.6 |
| 4 | 40 | Subha | 7.8 |
| 5 | 50 | Kamala | 8.83 |
| 6 | 60 | Isha | 9.11 |

- Access only Grades:

> DataFrame1 $ Grade

- Access only RegNo :

> DataFrame1 $ RegNo

Similarly,

> DataFrame1 $ Names

(12) Calculate Average of Marks in DataFrame :

```
> AverageScore = rowSums( DataFrame[, 3:7]) / length(3:7)
```

(or)

```
> AverageScore = rowMeans( DataFrame[-1])
```

(13) Combining :

```
> x = 2:6
> y = c(2,5)
> rbind(x,y)
```

output :

| | [,1] | [,2] | [,3] | [,4] | [,5] |
|---|---|---|---|---|---|
| x | 2 | 3 | 4 | 5 | 6 |
| y | 2 | 5 | 2 | 5 | 2 |

(14) Adding column to dataframe :

```
> cbind( dataframe, ColumnName)
```

(15) Adding row to dataframe

```
> rbind (dataframe, rowName)
```

(16) Calculating grades ranges :

```
> j = 1
> Grades = 0
> for (i in dataFrame$AverageScore) {
    if (i > 90) {
      Grades[j] = 'S'
    }
    else if (i >= 81 && i <= 90) {
      Grades[j] <= 'A'
    }
    else if (i >= 71 && i <= 80) {
      Grades[j] = 'B'
    }
    else {
    } Grades[j] = 'F'
  } j = j+1
```

(17)

```
myFactorial = function (n) {
    fact = 1
    if (n < 0) {
        print ("Factorial doesn't exist")
    }
    else if (n == 0) {
        print ("The factorial of 0 is 1")
    }
    else {
        for (i in 1:n) {
            fact = fact * i
        }
        print (paste ("The factorial of", n, "is =", fact))
    }
}
n = as.integer (readline (prompt = "Enter a number :"))
myfactorial (n)
```

---

(18)

```
myArmstrong = function (n) {
    sum = 0
    flag = 1
    while (flag <= n) {
        temp = flag
        sum = 0
        while (temp > 0) {
            digit = temp %% 10
            sum = sum + (digit ^ 3)
            temp = floor (temp / 10)
        }
        if (flag == sum) {
            print (flag)
        }
        flag = flag + 1
    }
}
n = as.integer ( .... )
myArmstrong (n)
```

Armstrong Number between 1 to n

(19) Age calculation : (from DOB)

```
Age = floor (age_calc (givenData$BirthDate , enddate = Sys.Date(),
                       units = "years")) .
```

(20) ~~Date~~ Experience of employees : (from DOJ)

```
Experience = floor (age_calc (given$DateofJoin , enddate = Sys.Date(),
                              units = "years")) .
```

(21) Display details of Gender = Female (F) :

```
print ( given [ given$Gender == 'F' , ])
```

(22) Display details of Gender = Male and Married (M) :

```
print ( given [ given$ Gender == 'M'  &  given$MaritageStatus == 'Mx' , ])
```

(23) Count no. of person who are single & title as 'ABC' :

```
print( nrow (given [ given$title == 'ABC'  &  given$Status == 'Single' , ]))
```

---

(24) Print Series :

```
n = as.Integer (readline (prompt = 'Enter number :'))

Summ = 0

myFactorial = function (i) {
    facto = 1
    for (j in 1:i) {
      facto = facto * j
    }
    return (facto)
}

for (i in 1:n) {
  Summ = Summ + i / myFactorial(i)
}

print (summ)
```

(25) Display only Female having MA degree from DataFram

```
> display = subset (DataFrame , DataFrame$ degree == 'MA' & DataFrame$ge
                                                    == 'female')
> print (display)
```

(26) Create another dataframe D2 from D1 where age is below 5

```
> D2 = subset (D1, D1$age < 50)
> print (D2)
```

(27) Program to display Fiberacci sequence upto nth term

```
recursive_func ← function (n) {
    if (n <= 1) {
      return (n)
    }
    else {
      return (recursive_func (n-1) + recursive_func (n-2))
    }
}
nterms = as.Integer (readline (prompt = " upto which term"))
print ("Fiberacci sequence :")
for (i in 0 : (nterms -1)) {
    print (recursive_func (i))
}
```

(28) Check variable exists or not ?

```
is.defined = function (sym) {
    sym = deparse (substitute (sym))
    env = parent.frame ()
    exists (sym, env)
}
is.defined (abc)          # FALSE
abc = 10
is.defined (abc)          # TRUE
```

[OR]

```
exists = function (variablename) {
#print (ls (env = globalenv ()))
return (1 == length (ls (pattern = paste ("^", variablename, "$",
                                          sep = ""), env = globalenv ())))
}
```

Ex :-

```
c = as.Date ("2016-02-01")
D = as.Date ("2016-06-15")
Seq.Date (D, C, by = "-1 month")
```

output :-

"2016-06-15" "2016-05-15" "2016-04-15" "2016-03-15"

"2016-02-15"

# MODULE 3 : DATA MANIPULATION

## *. MtCars Dataset :

- Full form of mtCars is Motor Trend Car Road Tests.
- The data was extracted from 1974 Motor Trend US Magazine.

## *. Description of mtCars :

- A dataframe with 32 observations on 11 (numeric) variables.

- [,1] mpg — Miles/(US) gallon
- [,2] cyl — Number of cylinders
- [,3] disp — Displacement
- [,4] hp — Gross Horsepower
- [,5] drat — Rear axle ratio
- [,6] wt — Weight (1000 lbs)
- [,7] qsec — time 1/4 mile
- [,8] vs — Engine {0 - V-shaped, 1 - straight}
- [,9] am — Transmission {0 - Auto, 1 - Manual}
- [,10] gear — Forward gears
- [,11] carb — Carburetors Count

## *. SORTING DATA :

```
# Sorting examples using the mtcars dataset
> attach(mtcars)

# sort by mpg
> newdata = mtcars[order(mpg),]
```

```
# sort by mpg and cyl
> newdata = mtcars [order(mpg, cyl),]

# sort by mpg (ascending) and cyl (descending)
> newdata = mtcars [order(mpg, -cyl),]

detach (mtcars)
```

*. DATA MANIPULATION:

- Finding and removing Duplicate Records.

*. DATA REDUNDANCY, DUPLICACY:

- Storage administrators are struggling to handle spiraling volumes of documents, audio, video, images and large email attachments.
- Adding storage is not always the best solution.
- Many companies are turning to data reduction technologies such as data deduplication.
- Entries that have been added by a system user multiple times.
- Example, Re registering because you have forgotten your details.
- It is one of the problem which causes inconsistency in databases.

- Some of data is stored at multiple locations or tables.
- Data redundancy is costly to address as it requires
  - additional storage,
  - synchronization between databases.
  - design work to align the information represented by different presentation of the same data.
- Storing the information several times, it leads to wastage of storage space.

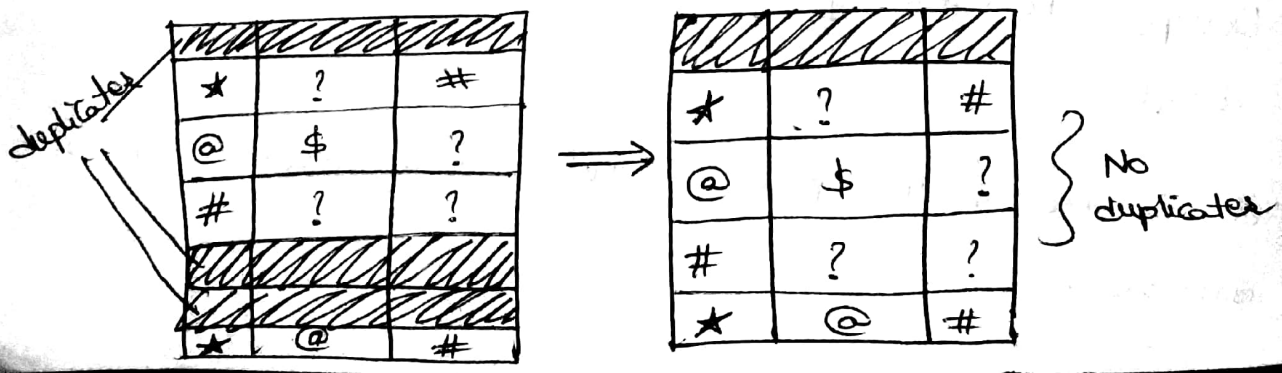
*. problems with Data Redundancy & Duplication:

1. Wasted storage space.
2. More difficult Database updates.
3. Possibility of Inconsistent data.

*. DATA DEDUPLICATION:

- Data deduplication, also called intelligent compression or single-instance storage.

*. Removing duplicated data in R:



- duplicated() : Find duplicate elements (R base)
- unique() : Keep only unique elements (R base)
- dplyr :: distinct() : Keep only unique elements (more efficient than unique())

"R base and dplyr"

- unique() - for extracting unique elements
- duplicated() - for identifying duplicated elements
  ↳ returns a logical vector where TRUE specifies which elements of vector or dataframe are duplicates.

Example() :

```
> x = c(1,1,4,5,4,6)
> duplicated(x)
```

o/p : FALSE TRUE FALSE FALSE TRUE FALSE

Example :

```
> x = c(1,1,4,5,4,6)
```

- Write a command to extract duplicate elements.
- Write a command to remove duplicate elements.

## * - DATA FRAME EXAMPLE : (IMP)

- How can I remove duplicate rows from this example data frame ?

| | |
|---|---|
| A | 1 |
| A | 1 |
| A | 2 |
| B | 4 |
| B | 1 |
| B | 1 |
| C | 2 |
| C | 2 |

I would like to remove the duplicates based on both the columns

| | |
|---|---|
| A | 1 |
| A | 2 |
| B | 4 |
| B | 1 |
| C | 2 |

order is not important.

- Consider An example here :

```
> a = c(rep("A", 3), rep("B", 3), rep("C", 2))
> b = c(1, 1, 2, 4, 1, 1, 2, 2)
> df = data.frame(a, b)
# Extract duplicate element
> df[duplicated(df),]
# Remove duplicate element
> df[!duplicated(df),]
# Display the duplicate records based on the column "colx"
  in the dataframe mydata.
> mydata[duplicated(mydata$colx),]
```

# Remove the duplicate records based on the column "colx" in the dataframe mydata.

```
> mydata [ ! duplicated (mydata $ colx), ]
```

# Find and drop duplicate elements

```
> unique (x)        # x = c (1,1,4,5,4,6)
```

o/p : 1 4 5 6

## * DATA MANIPULATION — DATA CLEANING :

- In R, missing values are represented by the symbol NA (not available).
- Impossible values (e.g., dividing by zero) are represented by the symbol NaN (not a number)
- Unlike SAS, R uses the same symbol for character and numeric data.

# Returns TRUE of x is missing

```
> is.na(x)
```

```
> y = c (1,2,3, NA)
> is.na (y)        # returns a vector (F F F T)
```

# print index of NA

```
> y = c (1, 2, 3, NA)
> for (i in 1 : length (y)) {
    if ( is.na (y[i])) {
      cat (i, "\n")
    }
  }
```

# using which function

```
> which (is.na (y))
```

# Testing for Missing values in Dataframe :

```
> df = data.frame (col1 = c (1:3, NA),
                   Col2 = c ("this", NA, "is", "text"),
                   Col3 = c (TRUE, FALSE, TRUE, TRUE),
                   col4 = c (2.5, 4.2, 3.2, NA),
                   StringsAsFactors = FALSE)
```

# identify NAs in full dataframe

```
> is.na (df)
```

# identify NAs in specific dataframe column

```
> is.na (df $Col2)
```

# Print NAs in row 3 in given data frame "df"

```
> is.na (df [3,])
```

# identify location of NAs in vector

```
> which (is.na(df))
```

o/p : 4 6 16

# identify count of NAs in dataframe

```
> sum (is.na (df))
```

o/p : 3

# Print the count of NAs in row 3 in given dataframe

```
> sum ( is.na ( df [3, ]))
```

# Print count of NAs in col4 in given dataframe 'df'

> sum(is.na(df[,4]))

# Arithmetic functions on missing values yield missing values.

> x = c(1, 2, NA, 3)

> mean(x) # returns NA

> mean(x, na.rm = TRUE) # returns 2

↳ 1, 2, NA, 3

→ 1, 2, 3

→ $\frac{1+2+3}{3} = \frac{6}{3} = 2$

rm - remove
na - NA value

# Print the median of vector x

> median(x, na.rm = TRUE)

# Print the median of col1 in dataframe "df"

> median(df$col1, na.rm = TRUE)

## *. IDENTIFYING COMPLETE CASES:

- The function complete.cases() returns a logical vector indicating which cases are complete.

# list rows of data without missing values

> df[complete.cases(df),]

## *. LISTWISE Deletion of Missing values:

- The functions na.omit() or na.exclude() returns the object with listwise deletion of missing values.

```
# create new dataset without missing data
> newdata = na.omit(df)

# Print the records without missing values
> na.exclude(df)
```

O/P:

| | Col1 | Col2 | Col3 | Col4 |
|---|---|---|---|---|
| 1 | 1 | this | TRUE | 2.5 |
| 3 | 3 | is | TRUE | 3.2 |

## *. DATA RECODING:

- One data manipulation task that you need to do in pretty much any data analysis is recode data.
- Replacing data in an existing field or recording into a new field based on criteria you specify.
- Recoding is also known as Replacing or Imputation.

Ex:

```
> x = c(3,4,5,6,7,8)        # numeric vector

# Recode the values less than 6 with 0.
> x[x<6] = 0                # O/P: 0 0 0 6 7 8
```

# Write a command to recode the values between 4 and 8 with 100 in the vector x.

```
> x = c(3,4,5,6,7,8)
> x[x>4 & x<8] = 100
> x
```

o/p : 3 4 100 100 100 8

## * Recoding Missing values (NA) :

- Values of a vector or dataframe can be recoded to NA if required

Ex :

```
> x = c(3,4,5,6,7,8)
> x[x==6] = NA
> x
```

o/p : 3 4 5 NA 7 8

# Write a program to recode the values grater than 6 with NA in vector x

```
> x[x>6] = NA
> x
```

o/p : 3 4 5 6 NA NA

# Recoding missing values by another value :

```
> A = c (3, 2, NA, 5, 3, 7, NA, NA, 5, 2, 6)
```

# Recoding missing values by 0 :

```
> A [is.na (A)] = 0
> A
```

o/p : 3 2 0 5 3 7 0 0 5 2 6

- Recoding tasks are more complex, particularly when you wish to re-code a categorical variable or factor

Ex :

```
> gender = c ("MALE", "FEMALE", "FEMALE", "MALE", "MALE")
> gender [ gender == "MALE" ] = 1
> gender
```

o/p : "1" "FEMALE" "FEMALE" "1" "1"

# Recode MALE by 1 & FEMALE by 0 using ifelse()

```
> ifelse ( gender == "MALE", 1, 0)
```

o/p : 1 0 0 1 1

# Recode MALE by 1, FEMALE by 2, UNKNOWN by 3 using ifelse()

```
> gender = c ("M", "F", "F", "UNKNOWN", "M")
```

```
> ifelse ( gender == "M" , 1 , ifelse( gender == "F", 2,3))
```

op: 1 2 2 3 1

*. Recoding values in dataframe :

```
> A = data.frame ( Gender = c ('F', 'F', 'M', 'F', 'B', 'M', 'M'
                   Height = c (154, 167, 178, 145, 169, 183,
                               176)
                 )
```

```
> A
```

op:

| | Gender | Height |
|---|---|---|
| 1 | F | 154 |
| 2 | F | 167 |
| 3 | M | 178 |
| 4 | F | 145 |
| 5 | B | 169 |
| 6 | M | 183 |
| 7 | M | 176 |

```
# This one gets re-coded to the value 99.
# Note that the Gender variable is located in the
  first column, or A[,1]
> A[,1] = ifelse ( A[,1] == 'M', 1, ifelse (A[,1]
                                    == 'F', 2, 99))
> A
```

o/p :

| | Gender | Height |
|---|---|---|
| 1 | 2 | 154 |
| 2 | 2 | 167 |
| 3 | 1 | 178 |
| 4 | 2 | 145 |
| 5 | 99 | 169 |
| 6 | 1 | 183 |
| 7 | 1 | 176 |

# Recode Data in existing Field :

```
> dataframe $ column = value
> dataframe $ column = 'value'
> dataframe $ column = NA
```

# Recode into a new field :

```
> dataframe $ copyColumn = NA
> dataframe $ copyColumn = dataframe $ column
```

## *. DATA MERGING :

- One of the challenges in data analytics is merging two datasets that share atleast one common column.
- Merging data involves in :
  - Adding Columns
  - Adding Rows

## *. Adding Rows :

- To join two dataframes (datasets) vertically, use the rbind function
- The two dataframes must have the same variables, but they doesn't have to be in the same order.
- Syntax :

```
> total = rbind ( dataframeA , dataframeB )
```

Ex :

df1

| | Key | field1 |
|---|---|---|
| 1 | aaa | 3 |
| 2 | bbb | 1 |
| 3 | ccc | 4 |

df2

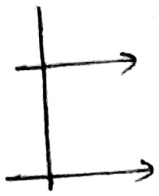
| | field2 | key |
|---|---|---|
| 1 | 2 | aaa |
| 2 | 1 | bbb |
| 3 | 7 | ccc |
| 4 | 8 | ddd |

```
> rbind (df1, df2)
```

| | Key | field1 |
|---|---|---|
| 1 | aaa | 3 |
| 2 | bbb | 1 |
| 3 | ccc | 4 |
| 4 | aaa | 2 |
| 5 | bbb | 1 |
| 6 | ccc | 7 |
| 7 | ddd | 8 |

output : ↗

- If dataframeA has variables that dataframe B doesnot, then either :



→ Delete the extra variables in data-frame A

→ Create the additional variables in dataframeB and set them to NA (missing) before joining them with rbind().

## *- Adding Columns :

- You can append two data-frames using cbind() function
- No. of observations in two dataframes must be same
- Appending is different from merging.

Ex :

df1

| | Key | field1 |
|---|---|---|
| 1 | aaa | 3 |
| 2 | bbb | 1 |
| 3 | ccc | 4 |

+

df2

| | Key | field2 |
|---|---|---|
| 1 | aaa | 2 |
| 2 | ccc | 1 |
| 3 | eee | 7 |

> cbind(df1, df2)

| | Key | field1 | Key | field2 |
|---|---|---|---|---|
| 1 | aaa | 3 | aaa | 2 |
| 2 | bbb | 1 | ccc | 1 |
| 3 | ccc | 4 | eee | 7 |

> rbind(df1, df2)

- Error in match.names : Names don't match previous names

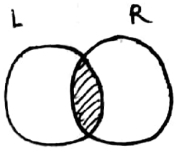
## *- MERGING DATA :

- Merging is joining two datasets that share at least one common column.
- Merging is also known as Join.

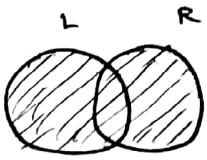
- Joins are of mainly three types :
  - Inner Join or Join
  - outer Join or Full Join → Left outer join or Left join
    → Right outer join or Right join
  - Cross Join

*. Types of Join :

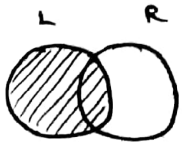
- Inner Join : Returns records that have matching values in both tables.
- Left outer Join : Returns all records from left table and matched records from the right table.
- Right outer Join : Returns all records from right table, and matched record from left table.
- Full outer Join : Returns all records when there is a match in either left or right table.



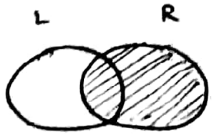
INNER JOIN



FULL JOIN



LEFT JOIN



RIGHT JOIN

- Cross Join : The cross Join is used to generate a paired combination of each row of the first table with each row of the second table. (Cartesian Join)

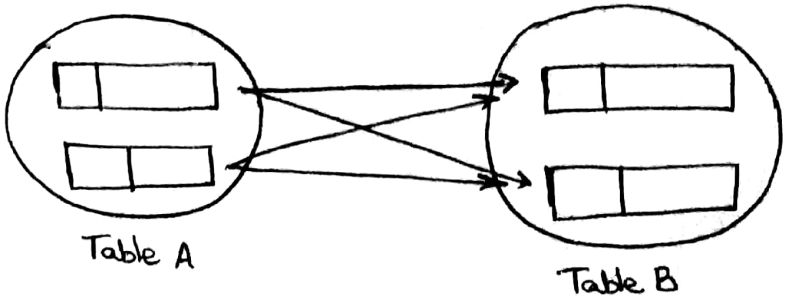


Table A

Table B

All combination of A, B

# Outer Join

```
> merge(x = df1, y = df2, by = "Id", all = TRUE)
```

# Left Join

```
> merge(x = df1, y = df2, by = "Id", all.x = TRUE)
```

# Right Join

```
> merge(x = df1, y = df2, by = "Id", all.y = TRUE)
```

# Cross Join

```
> merge(x = df1, y = df2, by = NULL)
```

| ID | X1 |
|---|---|
| 1 | a1 |
| 2 | a2 |

| ID | X2 |
|---|---|
| 2 | b1 |
| 3 | b2 |

- A semi join returns the rows of the first table where it can find a match in the second table
- An anti join returns the rows of the first table where it cannot find a match in the second table.

inner_join

| ID | X1 | X2 |
|---|---|---|
| 2 | a2 | b1 |

left_join

| ID | X1 | X2 |
|---|---|---|
| 1 | a1 | NA |
| 2 | a2 | b1 |

right_join

| ID | X1 | X2 |
|---|---|---|
| 2 | a2 | b1 |
| 3 | NA | b2 |

full_join

| ID | X1 | X2 |
|---|---|---|
| 1 | a1 | NA |
| 2 | a2 | b1 |
| 3 | NA | b2 |

semi_join

| ID | X1 |
|---|---|
| 2 | a2 |

anti_join

| ID | X1 |
|---|---|
| 1 | a1 |

# MODULE - 4

## * READING AND WRITING DATA IN R :

- There are a few principal functions reading data into R.
- Read.table, read.csv — for reading Tabular data.
- readLines — for reading lines of text file.
- source — for reading in R code files (inverse of dump).
- dget — for reading R code files (inverse of dput).
- load — for reading in saved workspaces.
- unserialize — single Robjects into binary form reading.

Similarly,

- write.table, write.csv
- writeLines, dump, dput, save, serialize

## *. READING DATA INTO R :

- R can read data from a variety of file formats — for example, files created as text, or in Excel, SPSS or Stata.
- We will mainly be reading files in text format .txt or .csv (Excel type).
- To read an entire dataframe directly, the external file will normally have a special form.
- The 1st line of file should have a name for each variable in dataframe.

- Each additional line of the file has as its first item a row label and the values for each variable.

EX :

```
> airqual = read.table("C:/Desktop/airquality.txt")
> airqual = read.csv("C://Desktop//airquality.csv")

# Command to assign column names
> Colnames(dataframe) = c("col1", "col2", "col3")

# Command to assign row names
> rownames(dataframe) = c("Row1", "Row2", "Row3", "Row4")

# Creating a new textfile textdf to save
> write.table(dataframe, "textdf.txt")
> read.table("textdf.txt")
```

* WORKING DIRECTORY :

```
# setting up working directory
> Setwd("C:/Desktop")

# get the current working directory
> getwd()
```

## *. IMPORTING DATA IN R :

- Data contained in external text files can be imported in R using one of the following functions :

→ scan()
→ read.table()
→ read.csv()
→ read.csv2()
→ read.delim()
→ read.delim2()

- The requirements on the form of data set are sometimes quite strict, so it is better to modify input files to satisfy this requirements.

---

## *. The function scan() :

- This function is the most flexible : It can be used to read logical, integer, numeric, complex, character, raw data, lists.

```
> scan ( file = " ", what = double(0), n = -1, sep = " ",
        dec = ".", skip = 0, na.strings = "NA")
```

# file → name of the file to be read.
# what → logical, integer, numeric, list, type of data.
# n → maximum data values to be read.
# sep → white space seperator.
# dec → decimal points character.
# skip → no. of lines to skip before scanning dataset

* The function read.table() :

> read.table ( file, header = FALSE, sep = " ", dec = ".", row.names, col.names )

# If only column labels are present, header = TRUE.

* The function read.csv() & read.csv2() :

IMP read.csv() ⟶ .CSV ("comma" separated files) decimal is "."

IMP read.csv2() ⟶ decimal is "," seperater ";"

> read.csv ( file, header = TRUE, sep = ",", dec = "." )

> read.csv2 ( file, header = TRUE, sep = ";", dec = "," )

* The functions read.delim() & read.delim2() :

- They are intended to read TAB seperated files.

> read.delim ( file, header = TRUE, sep = "/t", dec = "." )

> read.delim2 ( file, header = TRUE, sep = "\t", dec = "," )

* Exporting data :

- R objects can be exported into text file using cat() function :

> cat ( x, file = " ", sep = " ", fill = FALSE, labels = NULL, append = FALSE )

## x = R object

## *. WRITING DATAFRAMES :

- often it is useful to write a matrix or a data frame to a file as a grid of elements.

- write() – writes out a matrix or vector in a specified no. of columns.

- write.table() – writes out dataframe (or an object that can be coerced to a data frame) with row and column labels.

- write (x, file = "data", ncolumns = , append = FALSE, sep = " ")

  x – the data to be written out

  file – the file to write to

  ncolumns – the no. of columns to write the data in.

  append – if TRUE the data x are appended to file.

  sep – a string used to seperate columns. sep = "/t".

- write.table (x, file = " ", append = FALSE, sep = " ", na = "NA", dec = ".", row.names = TRUE, col.names = TRUE)

- write.csv (x, file = " ")
- write.csv2 (x, file = " ")

## *. DPLYR Package :

| dplyr Function | Description | Equivalent SQL |
|---|---|---|
| Select() | Selecting columns (variables) | SELECT |
| filter() | Filter (subset) rows | WHERE |
| group_by() | Group the data | GROUPBY |
| summarize() | Summarize (or aggregate) data | - |
| arrange() | Sort the data | ORDER BY |
| join() | Joining Data Frames (tables) | JOIN |
| mutate() | Creating New Variable | COLUMN ALIAS |

Practice :

(#) Load Data :

```
> mydata = read.csv("../sampledata.csv")
```

# Selecting Random N Rows :

```
> sample_n (mydata, 3)        # no. of rows to select
```

# Selecting Random Fraction of Rows :

```
> sample_frac (mydata, 0.1)   # 10% of rows
```

(#) Remove duplicate rows based on all variables

```
> x1 = distinct (mydata)
```

# Remove duplicate rows based on variable

> x2 = distinct (mydata, Index, .keep_all = TRUE)
(Index → variable)

## Remove duplicate rows based on multiple variables

> x2 = distinct (mydata, Index, Y2010, .keep_all = TRUE)
(Index → variable, Y2010 → variable)

# select() Function :

> mydata2 = select (mydata, Index, state : Y2008)

# Dropping variables :

> mydata = select (mydata, -Index, - state)

(or)

> mydata = select (mydata, - c (Index, state)

## selecting or Dropping variables starts with 'Y' :

> mydata 3 = select (mydata, starts_with("Y")) ~select

> mydata33 = Select (mydata, - starts_with ("Y")) ~drop

# Selecting variables contain 'I' in their names :

> mydata4 = select (mydata, contains ("I"))

# Reorder variables :

> mydata5 = select (mydata, state, everything())

## Rename from "Index" to "Index1" variables :

> mydata6 = rename (mydata, Index1 = Index)

# Filter Rows :

```
> mydata7 = filter(mydata, Index == "A")
```

# Multiple Selection Criteria :

```
> mydata7 = filter(mydata6, Index %in% c("A","C"))
```

# 'AND' Condition selection criteria :

```
> mydata8 = filter(mydata6, Index %in% c("A","C")
                           & Y2002 >= 1300000)
```

# 'OR' Condition selection criteria :

```
> mydata9 = filter(mydata6, Index %in% c("A","C")
                           | Y2002 >= 1300000)
```

# 'NOT' Condition selection criteria :

```
> mydata10 = filter(mydata6, !Index %in% c("A","C"))
```

# Contains Condition :

```
> mydata10 = filter(mydata6, grepl("Ar", State))
```

↓

the grepl function is used to search pattern "Ar" for the record contain "Ar" in state names.

# Summarize selected variables :

```
> summarize (mydata, Y2015_mean = mean (Y2015),
                     Y2015_med = median (Y2015))
```

# Summarize multiple variables :

```
> summarize_at (mydata, vars (Y2005, Y2006),
                        list (~n(), ~mean(.), ~median(.)))
```

# Summarize all numeric variables :

```
> summarize_if (mydata, is.numeric, list (~n(), ~mean(.),
                                          ~median(.)))
```

[OR]

```
> numdata = mydata [sapply (mydata, is.numeric)]

> summarize_all (numdata, list (~n(), ~mean(.), ~median(.)))
```

# Summarize Factor variable :

# checking the no. of levels / categories and count of missing observations in a variable.

```
> summarize_all (mydata ["Index"], funs (nlevels (.),
                                         nmiss = sum (is.na(.))))
```

| nlevels | nmiss |
|---|---|
| 1 | 19 | 0 |

## Arrange() Function :

# Sort data by multiple variables

```
> arrange (mydata, Index, Y2011)
> arrange (mydata, desc (Index), Y2011)
```

## Pipe operator %>% :

**Example :** We are selecting 10 random observations of two variables "Index" "State" from the data frame "mydata".

```
dt = sample_n (select (mydata, Index, State), 10)
```

[OR]

```
dt = mydata %>% select (Index, State) %>% sample_n (10)
```

| | Index | State |
|---|---|---|
| 44 | T | Texas |
| 32 | N | New Mexico |
| 51 | W | Wyoming |
| 9 | D | D Columbia |
| 5 | C | California |
| 40 | R | Rhode |
| 22 | M | MIT |
| 4 | A | Atlanta |
| 42 | S | South D |
| 46 | V | Vermont |

## Groupby() Function :

# Summarize data by Categorical value variable :

```
> t = summarise_at ( group-by (mydata, Index),
                     vars (Y2011, Y2012), funs (n() ...)))
```

[OR]

```
> t = mydata %>% group_by (Index) %>%
      summarise_at ( vars (Y2011 : Y2015), funs (...))
```

# Filter data with Categorical variable :

```
> t = mydata %>% filter (Index %in% c("A", "C", "I"))
      %>% group_by (Index) %>% do (head(:, 2))
```

→ It extracts top 2 rows from 'A', 'C' and 'I' categories of variable index.

# Selecting 3rd maximum value by Categorical Variable :

```
> t = mydata %>% select ( Index, Y2015) %>%
      filter (Index %in% c('A', 'C', 'I')) %>%
      group_by (Index) %>%
      do (arrange (., desc (Y2015))) %>% slice(3)
```

| | Index | Y2015 |
|---|---|---|
| 1 | A | 1647724 |
| 2 | C | 1330736 |
| 3 | I | 1583516 |

# Summarize, Group, Sort together :

```
> t = mydata %>% group_by (Index) %>%
      summarise (Mean_2014 = mean (Y2014, na.rm = TRUE))
      Mean_2015 = mean (Y2015, na.rm = TRUE)) %>%
      arrange (desc (Mean_2015))
```

# Select state that generated highest income among the variable 'Index' :

```
> out = mydata %>% group_by (Index) %>%
      filter (min_rank (desc (Y2015)) == 1) %>%
      select (Index, State, Y2015)
```

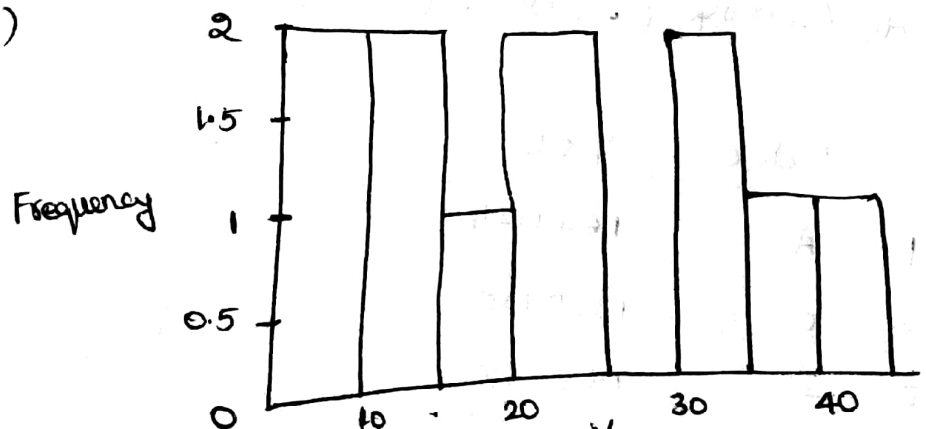
..... so many functions are there . .....

---

## *. EXPLORATORY DATA ANALYSIS :

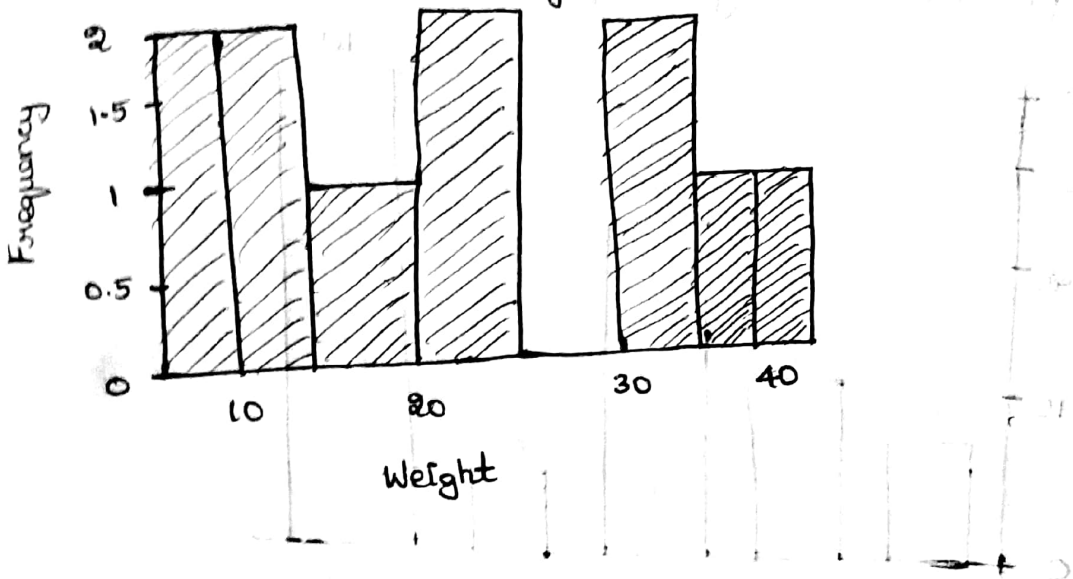
# Simple Histogram

```
> x ← c (9, 13, 21, 8, 36, 22, 12, 41, 31, 33, 19)
> hist (x)
```



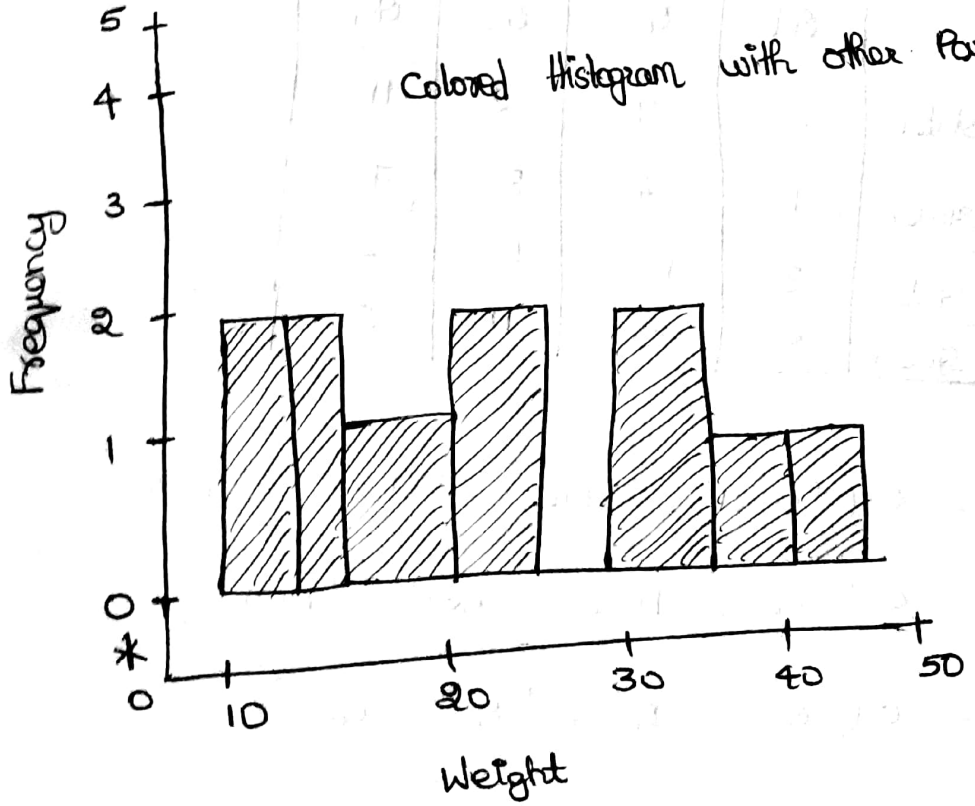
# colored Histogram

```
> x ← c(9, 13, 21, 8, 36, 22, 12, 41, 31, 33, 19)
> hist(x, xlab = "Weight", col = "yellow", border = "blue",
       main = "Colored Histogram")
```



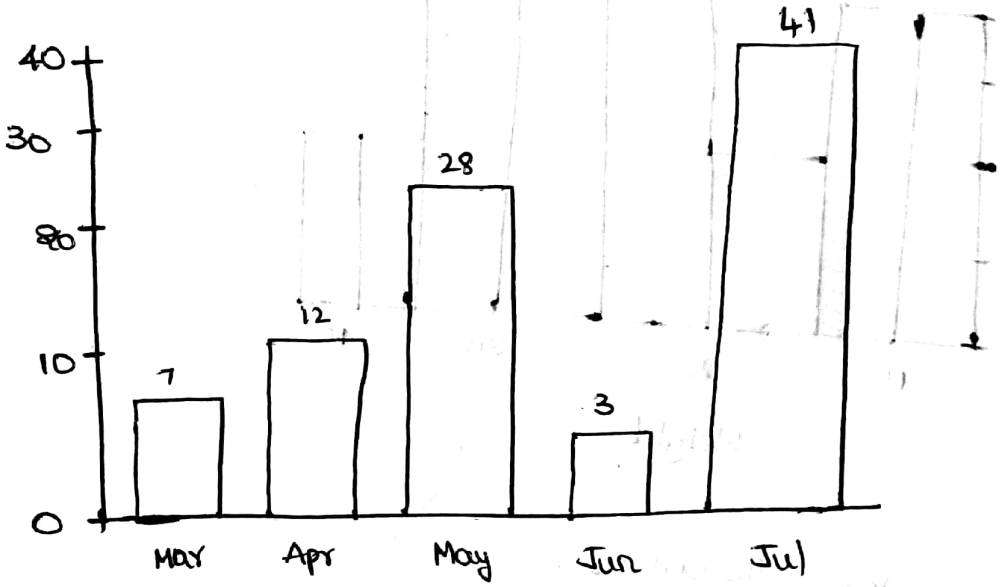

# Parameters Histogram :

```
> hist(x, xlab = "Weight", col = "black", border = "black",
       main = "Colored Histogram with other Parameters",
       xlim = c(0,50), ylim = c(0,5))
```



## *. BAR CHART :

```
> x = c ("Mar", "Apr", "May", "Jun", "Jul")
> y = c (7, 12, 28, 3, 41)
> barplot (names.arg = x, y)
```



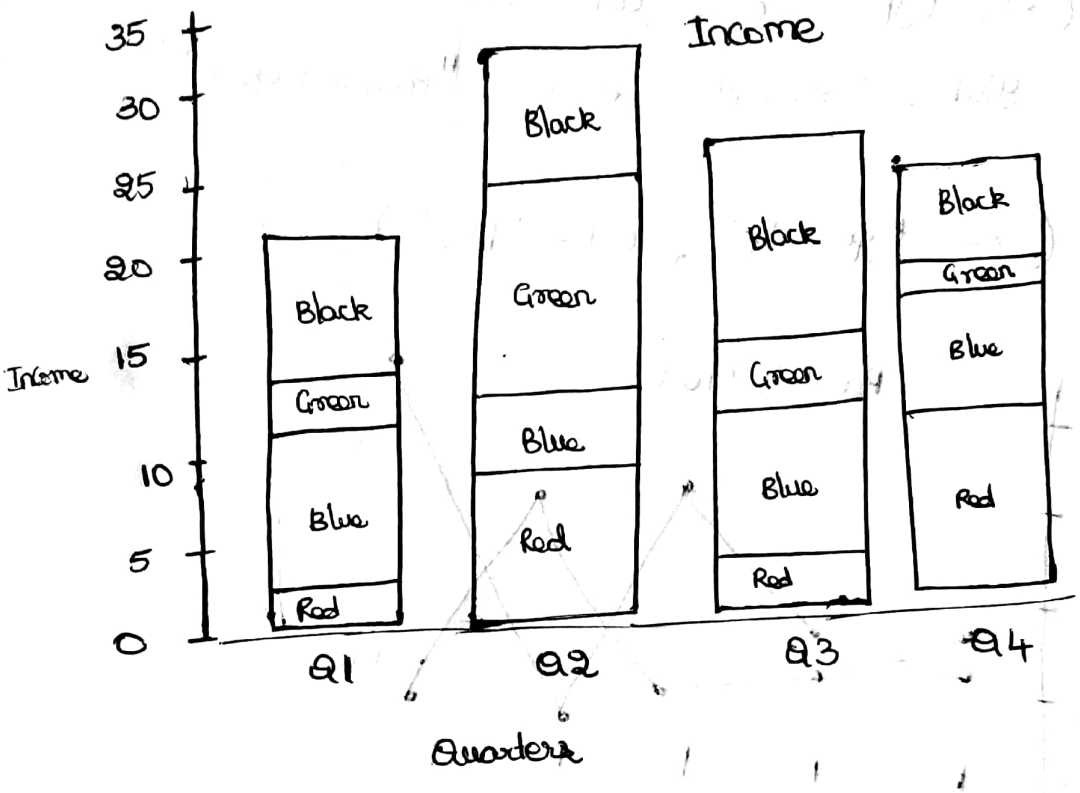

*. Ex : Draw a stacked bar chart with the following data.

| | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|
| State1 | 2 | 9 | 3 | 11 |
| State2 | 9 | 4 | 8 | 7 |
| State3 | 3 | 12 | 5 | 2 |
| State4 | 8 | 10 | 11 | 5 |

```
> states = c ("state1", "state2", "state3", "state4")
> colors = c ("red", "blue", "green", "black")
> quarters = c ("Q1", "Q2", "Q3", "Q4")
```

> values = matrix (c (2,9,3,11,9,4,8,7,13,12,5,2,8,10,11,15)
nrow = 4, ncol = 4, byrow = TRUE

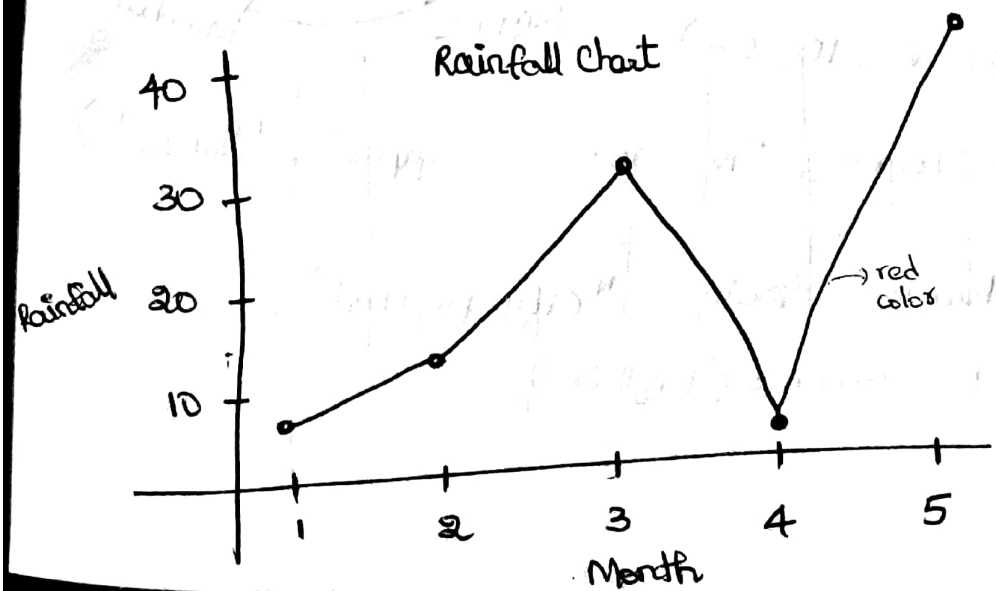
> barplot (values, main = "Income", names.arg = quarters,
xlab = "Quarters", ylab = "Income", col = colors)




# Draw a Line graph single line

> x = c (7, 12, 28, 3, 41)

> plot (x, type = "o", col = "red", xlab = "Month",
ylab = "Rainfall", main = "Rainfall chart")



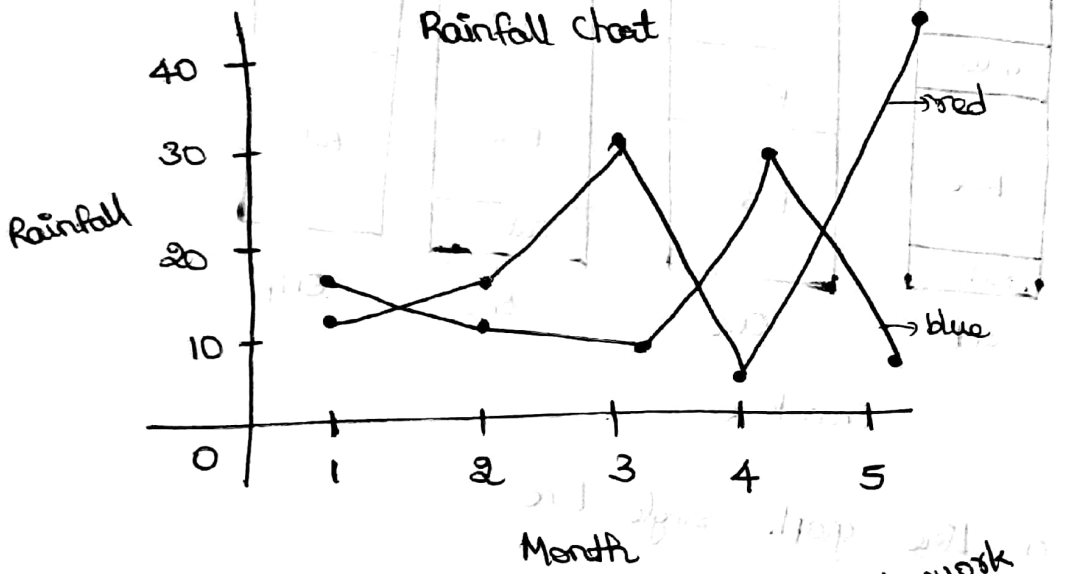
# Draw multiple line graphs :

```
> x1 = c(7, 12, 28, 3, 41)

> x2 = c(14,7, 6, 19, 3)

> plot(x1, type = "o", col = "red", xlab = "Month",
       ylab = "Rainfall", main = "Rainfall chart")

> lines(x2, type = "o", col = "blue")
```




## *. Piechart :

# Ex :

```
> x = c(21, 62, 10, 53)

> labels = c("London", "New York", "Singapore", "Mumbai")

> pie(x, labels, main = "City pie chart",
      col = rainbow(length(x)))
```