

# What is R?

- R is a programming language and software environment for statistical analysis and graphics representation.
- R is a dialect of the S language.
- R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.

# History of R

- S is a language that was developed by John Chambers and others at the old Bell Telephone Laboratories, originally part of AT&T Corp. at 1976.
- In 2004 Insightful purchased the S language from Lucent for \$2 million under the product name S-PLUS and built a number of fancy features.
- In 1991, Ross Ihaka and Robert Gentleman developed R as a free software environment for their teaching classes when they were colleagues at the University of Auckland in New Zealand.
- In addition, many other people have contributed new code and bug fixes to the project.

- Early 1990s: The development of R began.
- August 1993: The software was announced on the S-news mailing list.
- June 1995: The code was made available under the Free Software Foundation's GNU General Public License (GPL), Version 2.
- February 2000: The first version of R, version 1.0.0, was released.
- October 2004: Release of R version 2.0.0.
- April 2013: Release of R version 3.0.0.
- April 2015: Release of R-3.2.0
- December 2019 : Release of R-3.6.2

# Importance of R

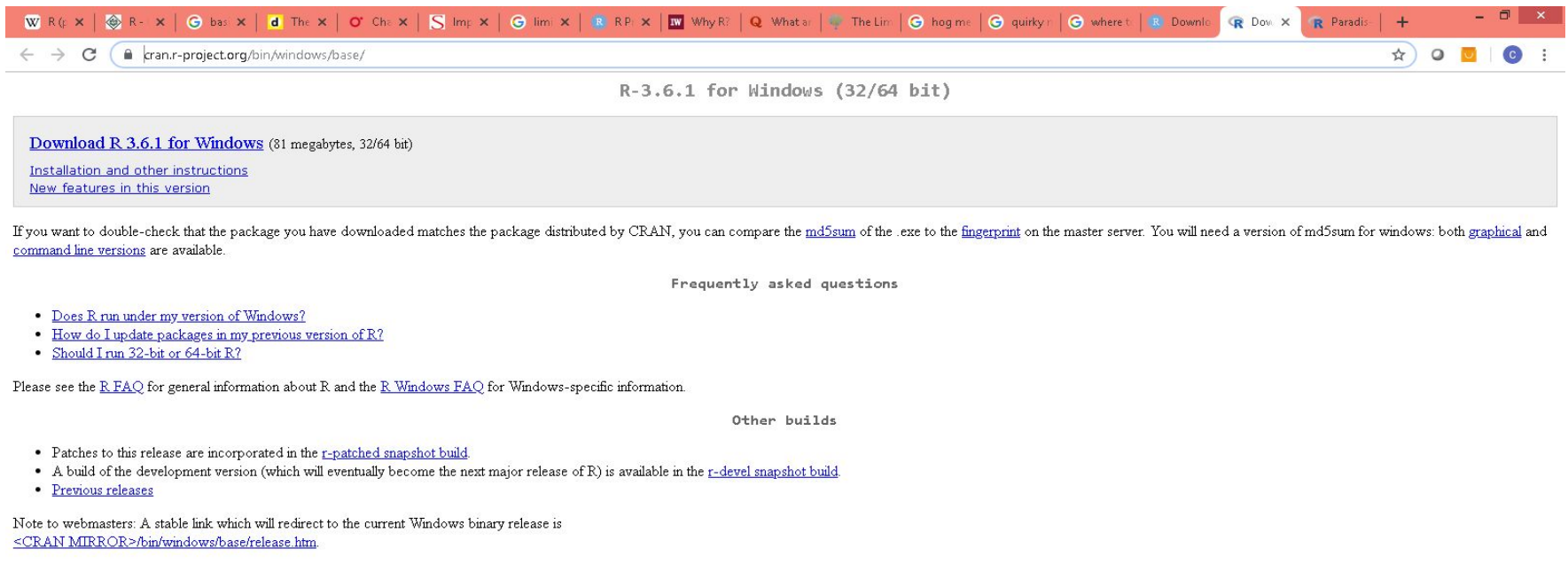
- R is free, open-source code
- R runs anywhere
- R supports extensions
- R provides an engaged community
- R connects with other languages
- Running code without a compiler
- The Ultimate Statistical Analysis Kit
- Benefits of Charting

# Limitation of R

- Lack of packages.
- R commands give little thought to memory management, and so R can consume all available memory.
- Memory management, speed, and efficiency are probably the biggest challenges R faces.
- R isn't just for advanced programmers.

# R Resources

- <https://cran.r-project.org>



The screenshot shows a web browser window with the address bar displaying `cran.r-project.org/bin/windows/base/`. The page title is "R-3.6.1 for Windows (32/64 bit)". The main content area has a light gray background and contains the following links:

- [Download R 3.6.1 for Windows](#) (81 megabytes, 32/64 bit)
- [Installation and other instructions](#)
- [New features in this version](#)

Below the links, a paragraph of text reads: "If you want to double-check that the package you have downloaded matches the package distributed by CRAN, you can compare the [md5sum](#) of the .exe to the [fingerprint](#) on the master server. You will need a version of md5sum for windows: both [graphical](#) and [command line versions](#) are available."

Next is a section titled "Frequently asked questions" with a bulleted list:

- [Does R run under my version of Windows?](#)
- [How do I update packages in my previous version of R?](#)
- [Should I run 32-bit or 64-bit R?](#)

Below this is a paragraph: "Please see the [R FAQ](#) for general information about R and the [R Windows FAQ](#) for Windows-specific information."

Then is a section titled "Other builds" with a bulleted list:

- Patches to this release are incorporated in the [r-patched snapshot build](#).
- A build of the development version (which will eventually become the next major release of R) is available in the [r-devel snapshot build](#).
- [Previous releases](#)

At the bottom of the content area, a note reads: "Note to webmasters: A stable link which will redirect to the current Windows binary release is [<CRAN MIRROR>/bin/windows/base/release.htm](#)."

Last change: 2019-07-05

Activate Windows  
Go to PC settings to activate Windows.

# R for Ubuntu

- There are two ways to install R in Ubuntu. One is through the terminal, and the other is through the Ubuntu Software Center.

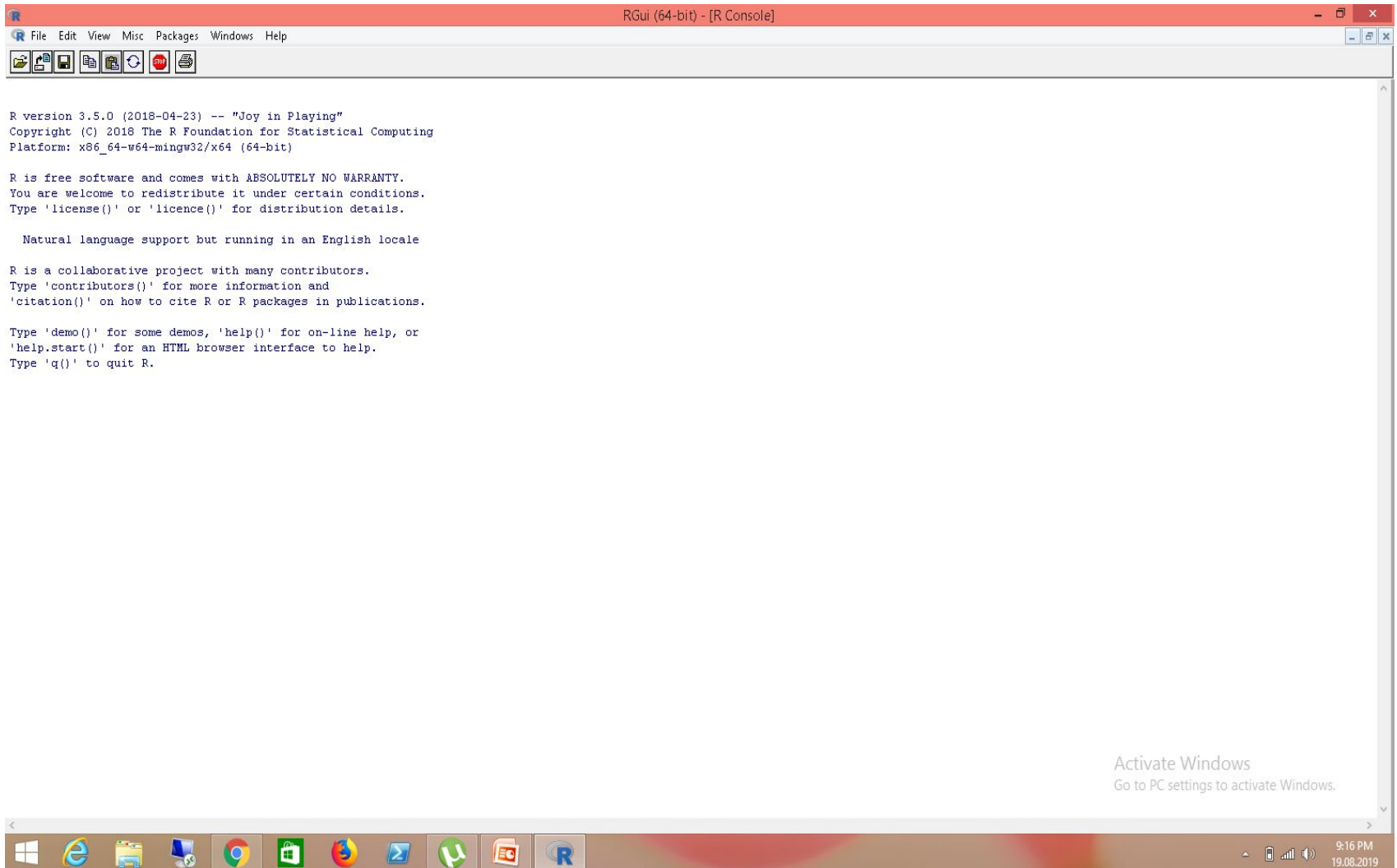
## 1. Through Terminal -

- Press Ctrl+Alt+T to open Terminal
- Then execute `sudo apt-get update`
- After that, `sudo apt-get install r-base`

## 2. Through Ubuntu Software Center -

- Open Ubuntu Software Center
- Search for r-base
- And click Install
- Then run R by executing **R** in the Terminal

# R console



The image shows a screenshot of the R console window. The window title is "RGui (64-bit) - [R Console]". The menu bar includes File, Edit, View, Misc, Packages, Windows, and Help. The toolbar contains icons for file operations and execution. The main text area displays the R version 3.5.0 startup screen, which includes the R logo, version information, copyright notice, platform details, and instructions for using R. The Windows taskbar is visible at the bottom, showing various application icons and the system clock.

```
R version 3.5.0 (2018-04-23) -- "Joy in Playing"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
```

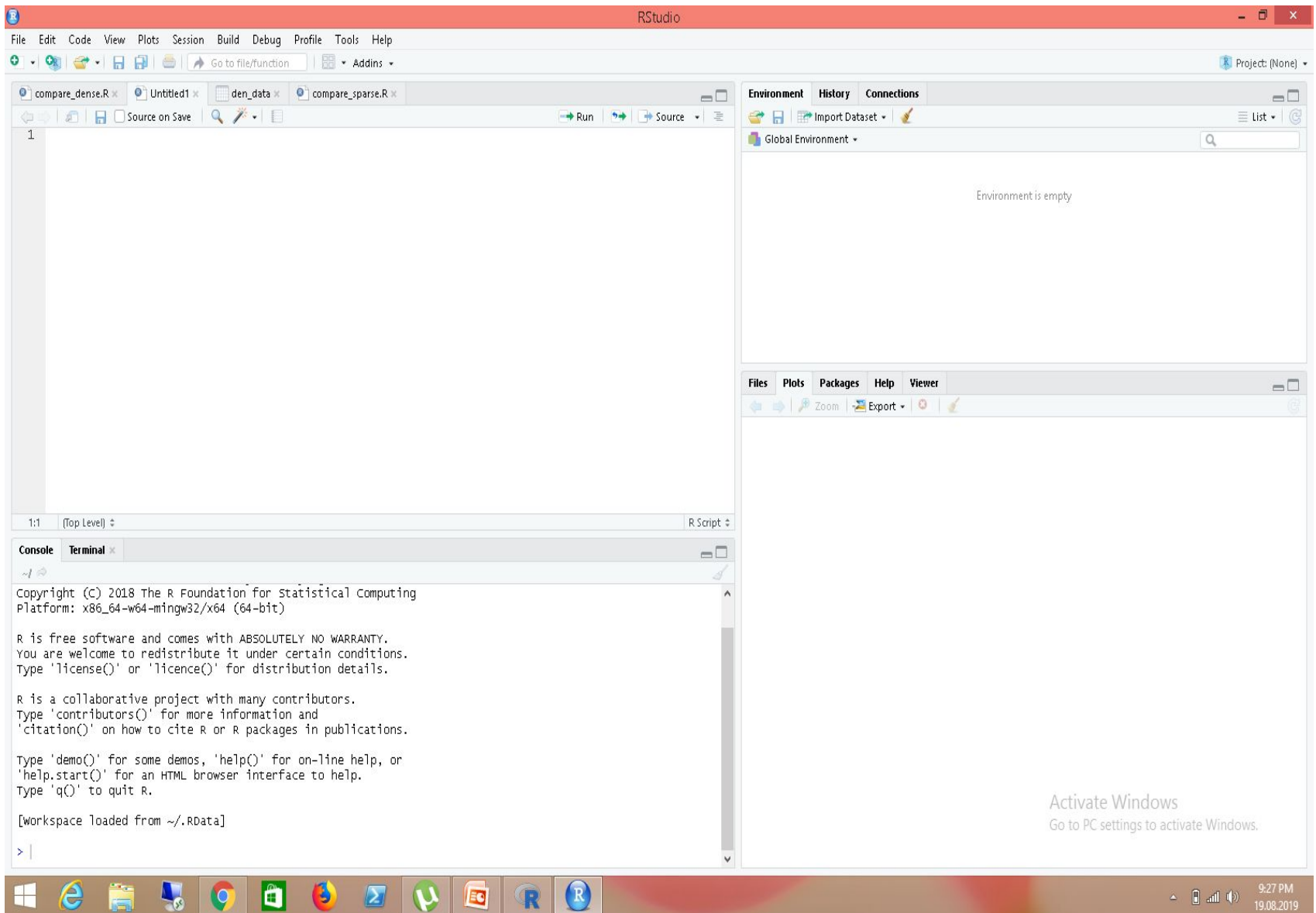
Activate Windows  
Go to PC settings to activate Windows.

9:16 PM  
19.08.2019



# R Studio

- RStudio is an integrated development environment for R.
- It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management.



# Arithmetic Operations

- Every statistical analysis involves a lot of calculations, and calculation is what R is designed for — the work that R does best.
1. Basic arithmetic operators-
    - These operators are used in just about every programming language.

# 1. Basic Arithmetic Operator

| <i>Operator</i>     | <i>Description</i>                               | <i>Example</i>          |
|---------------------|--|-------------------------|
| $x + y$             | y added to x                                     | $2 + 3 = 5$             |
| $x - y$             | y subtracted from x                              | $8 - 2 = 6$             |
| $x * y$             | x multiplied by y                                | $3 * 2 = 6$             |
| $x / y$             | x divided by y                                   | $10 / 5 = 2$            |
| $x ^ y$             | x raised to the power y                          | $2 ^ 5 = 32$            |
| $x \% y$            | remainder of x divided by y (x mod y)            | $7 \% 3 = 1$            |
| $x \mathrel{/\%} y$ | x divided by y but rounded down (integer divide) | $7 \mathrel{/\%} 3 = 2$ |

## 2. USING MATHEMATICAL FUNCTIONS

| <i>Function</i>               | <i>What It Does</i>  |
|-------------------------------|--|
| <code>abs(x)</code>           | Takes the absolute value of $x$  |
| <code>log(x, base = y)</code> | Takes the logarithm of $x$ with base $y$ ; if base is not specified, returns the natural logarithm |
| <code>exp(x)</code>           | Returns the exponential of $x$   |
| <code>sqrt(x)</code>          | Returns the square root of $x$   |
| <code>factorial(x)</code>     | Returns the factorial of $x$ ( $x!$ )  |

# 3. Relational Operators

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             |

## 4. Logical Operators

Logical Operators in R

| Operator | Description              |
|----------|--------------------------|
| !        | Logical NOT              |
| &        | Element-wise logical AND |
| &&       | Logical AND              |
|          | Element-wise logical OR  |
|          | Logical OR               |

## 5. Assignment operator-

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

## 6. Vector operations-

- Vector operations are functions that make calculations on a complete vector, like sum().
- Each result depends on more than one value of the vector.

## 7. Matrix operations-

- These functions are used for operations and calculations on matrices.



# Objects

- To create new variables, you will need to use the assignment operator (<-).
- Instead of declaring data types, as done in C++ and Java, in R, the user assigns the variables with certain Objects in R, the most popular are:
  - Vectors
  - Factors
  - Lists
  - Data Frames
  - Matrices
- The data type of the object in R becomes the data type of the variable by definition.
- **R's** basic data types are character, numeric, integer, complex, and logical.

# vector

- A vector is the simplest type of data structure in R. A vector is a sequence of data elements of the same basic type.
- There are six data types of the simplest object - vector:
  1. Logical
  2. Numeric
  3. Integer
  4. Character
  5. Raw
  6. Complex
- If you want to check the variable type, use *class()*.

- *A vector is a sequence of elements that share the same data type.* These elements are known as components of a vector.
- R vector comes in two parts: **Atomic vectors** and **Lists**.
- All elements of an atomic vector must be of the same type, whereas the elements of a list can have different types.

# Atomic Vectors in R

- There are four common types of R atomic vectors:

## 1. Numeric Data Type

- Decimal values are referred to as numeric data types in R. If we assign a decimal value for any variable  $g$ ,  $g$  will become a numeric type.

## 2. Integer Data Type

- A numeric value with no fraction called integer data is represented by “Int”.
- -54 and 23 are two of the examples of an integer. Int size is 2 bytes while long Int size is 4 byte.
- In order to assign an integer to a variable, there are two ways:

1) The first way is to use the `as.integer()` function:

```
a <- as.integer(4)
```

- For checking data type:
- `typeof(a)`

2) The second way is the appending of L to the value:

```
b <- 4L
```

- For checking data type:
- `typeof(b)`

### 3) Character Data Type

- The character is held as the one-byte integer in memory. There are two ways to create a character data type value in R:

1. The first method is by typing a string between " "

- `x = "Rstudio"`
- For determining the type of x:
- `typeof(x)`

2. In order to convert a number into character, make use of `as.character()` function as follows:

- `> y = as.character(42)`
- For determining the type of y:
- `> typeof(y)`

## 4. Logical Data Type

- A logical data type returns either of the two values – TRUE or FALSE based on which condition is satisfied.
- **For example:**
- `a =3; b =6`
- `g = a>b`
- `g    #print the logical value`

# How to Create Vector in R?

- The `c()` function is used for creating a vector in R. This function returns a one-dimensional array, also known as vector.
- For example:
- `x <- c(1,2,3,4)`
- There are several other ways of creating a vector:

## 1. Using the Operator

- `x <- 1:5`
- For `y` operator:
- `y <- 5:-5`
- `y`



## 2. Create R vector using seq() function

- There are also two ways in this. The first way is to set the step size and the second method is by setting the length of the vector.

### 1) Setting step size with 'by' parameter:

- `seq(2,4, by = 0.4)`
- `(2.0,2.4,2.8,3.2,3.6,4.0)`

### 2) Specifying length of vector with the 'length.out' feature:

- `seq(1,4, length.out = 5)`
- `(1.00,1.75,2.50,3.25,4.00)`

# How to Access Elements of R Vectors?

- With the help of vector indexing, we can access the elements of vectors. Indexing denotes the position where the values in a vector are stored.

## 1. Indexing with Integer Vector

- Unlike many programming languages like Python, C++, Java etc. where the indexing starts from 0, the indexing of vectors in R starts with 1.
- `X(1,2,3)`
- `X[1]`
- O/p =?

## 2. Indexing with Character Vector

- Character vector indexing can be done as follows:
- `x <- c("One" = 1, "Two" = 2, "Three" = 3)`
- `x["Two"]`

## 3. Indexing with Logic Vector

- In logical indexing, the positions whose corresponding position has logical vector TRUE are returned.
- `a <- c(1,2,3,4)`
- `a[c(TRUE, FALSE, TRUE, FALSE)]`

# Operations in R Vector

## 1. Combining Vector in R

- Functions are used to combine vectors. In order to combine the two vectors in R, we will create two new vectors 'n' and 's'. Then, we will create another vector that will combine these two using `c(n,s)` as follows:
- `n = c(1, 2, 3, 4)`
- `s = c("Hadoop", "Spark", "HIVE", "Flink")`
- `c(n,s)`
- Output: ?

## 2. Arithmetic Operations on Vectors in R

- Arithmetic operations on vectors can be performed member-by-member.
- Suppose we have two vectors a and b:
- $a = c(1, 3)$
- $b = c(1, 3)$
- For Addition:
- $a + b$
- For subtraction:
- $a - b$
- For division:
- $a / b$
- For remainder operation:
- $a \% \% b$

### 3. Logical Index Vector in R -

- By using a logical index vector in R, we can form a new vector from a given vector, which has the same length as the original vector.
- If the corresponding members of the original vector are included in the slice, then vector members are TRUE and otherwise FALSE.
- `S = c("bb", "cc")`
- `L = c(TRUE, TRUE)`
- `S[L]`
- o/p= "bb", "cc".

## 4. Numeric Index

- For indexing a numerical value in R, we specify the index between square braces [ ].
- If our index is negative, then R will return us all the values except for the index that we have specified.
- `x <- c("aa", "bb", "cc", "dd", "ee")`
- `x[3] = ?`
- `x[-2] = ?`

## 5. Duplicate Index

- The index vector allows duplicate values. Hence, the following retrieves a member twice in one operation.
- `s = c("aa", "bb", "cc", "dd", "ee")`
- `s[c(2,3,3)] = ?`

## 6. Range Indexes

- To produce a vector slice between two indexes, we can use the colon operator `:`. It is convenient for situations involving large vectors.
- `s = c("aa", "bb", "cc", "dd", "ee")`
- `s[1:3]`

## 7. Out-of-order Indexes

- The index vector can even be out-of-order. Here is a vector slice with the order of first and second members reversed.
- **For example:**
- `> s [ c (2, 1, 3) ]`



# 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 R objects.
- A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number.

# 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.
- `var.1 = c(0,1,2,3)`
- `var.2 <- c("learn","R")`
- `c(TRUE) -> 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")`

# 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.
- `var_x <- "Hello"`
- `Class(var_x)=?`
- `var_x <- 34.5`
- `var_x <- 27L`

# Finding Variables

- 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())`
- `print(ls(pattern = "var"))`

# Deleting Variables

- Variables can be deleted by using the **rm()** function.
- Below we delete the variable var.3. On printing the value of the variable error is thrown.
- `rm(var.3)`
- `print(var.3)`

# Factor

- R factor is used to store categorical data as levels.
- It can store both character and integer types of data.
- These factors are created with the help of *factor()* functions, by taking a vector as input.
- R factors are variables. The factor is stored as integers.
- They have labels associated with these unique integers.
- Factor contains a predefined set value called levels. By default, R always sorts levels in alphabetical order.

# How to Create a Factor

- `directions <- c("North", "North", "West", "South")`
- `factor(directions)`
- o/p= levels: North, South, West
- In order to add this missing level to our factors, we use the “levels” attribute as follows:
- `factor(directions, levels= c("North", "East", "South", "West"))`
- In order to provide abbreviations or ‘labels’ to our levels, we make use of the labels argument as follows  
—
- `factor(directions, levels= c("North", "East", "South", "West"), labels=c("N", "E", "S", "W"))`

## Cont'd..

- if you want to exclude any level from your factor, you can make use of the exclude argument.
- `factor(directions, levels= c("North", "East", "South", "West"), exclude = "North")`
- There are various ways to access the elements of a factor in R. Some of the ways are as follows:
  - `data <- c("East", "West", "East", "North")`
  - `data[4]`
  - `data[c(2,3)]`
  - `data[-1]`
  - `data[c(TRUE, FALSE, TRUE, TRUE)]`



# How to Modify an R Factor?

- To modify a factor, we are only limited to the values that are not outside the predefined levels.
- `print(data)`
- `data[2] <- "North"`
- `data[3] <- "South"`

# Factor Functions in R

- *is.factor()* checks if the input is present in the form of factor and returns a Boolean value (TRUE or FALSE).
- *as.factor()* takes the input (usually a vector) and converts it into a factor.
- *is.ordered()* checks if the factor is ordered and returns boolean TRUE or FALSE.
- *as.ordered()* function takes an unordered function and returns a factor that is arranged in order.
- `f_directions <- factor(directions)`
- `is.factor(f_directions)`
- `as.factor(directions)`
- `is.ordered(f_directions)`
- `as.ordered(f_directions)`

# Reserved Words

- Reserved words in R programming are a set of words that have a special meaning and cannot be used as an identifier.
- The list of reserved words can be viewed by typing `?reserved` or `help(reserved)` at the R command prompt.

Reserved words in R

|             |          |             |               |          |
|-------------|----------|-------------|---------------|----------|
| if          | else     | repeat      | while         | function |
| for         | in       | next        | break         | TRUE     |
| FALSE       | NULL     | Inf         | NaN           | NA       |
| NA_integer_ | NA_real_ | NA_complex_ | NA_character_ | ...      |

# Comments in R

- Single comment is written using # at the beginning of the statement as follow.
- #Test program in R.
- R does not support multi-line comments as in C or python.

# Identifiers

- The unique name given to a variable like function or objects is known as an identifier.
- Following are the rules for naming an identifier.
  1. Identifiers can be a combination of letters, digits, period(.) and underscore.
  2. It must start with a letter or a period. If it starts with a period, it can not be followed by a digit.
  3. Reserved word in R can not be used as identifier.  
Ex. Total1,sum,.date.of.birth,Sum\_of\_two etc.

# Constants

- Constants or literals, are entities whose value cannot be altered. Basic types of constants are numeric constants and character constants.
- There are built-in constants also. All numbers fall under this category.
- They can be of type integer, double and complex.
- But it is not good to rely on these, as they are implemented as variables whose values can be changed,

# Reading Strings

- We can read strings from a keyboard using the `readline()` fun.
- It lets the user to enter a one-line string at the terminal.
- `Value <- readline(prompt="string")`
- Ex. `Print(n<-readline(prompt="enter the subject:"))`
- Enter the subject : R
- `[1] "R"`

# Data Types

- The variables are assigned with R objects and the data type of the R objects becomes the data type of the variable. There many type of R object.

1. Vectors
2. Lists
3. Matrices
4. Arrays
5. Factors
6. Data Frames



# Basic Data Types

- Numeric- Decimal values are called numeric in R.
- Integer
- Complex- A complex value in R is defined via the pure imaginary value  $i$ . A complex number will be in the form of  $a+bi$ .
- Logical- There are two logical values True and False.
- Character-used to represent character value in R.

# List

- A List is a generic vector containing other objects. Lists are the R objects which contain elements of different types like – numbers, strings, vectors and another list inside it.
- A list can also contain a matrix or a function as its elements.
- List is created using **list()** function.

# Creating a List

- `n <- list(c(2,3), c("a","b","c"), c(TRUE,FALSE,TRUE),3)`
- Output-
- `[[1]]`
- `[1] 2,3`
- `[[2]]`
- `[1] "a" "b" "c"`
- `[[3]]`
- `[1] TRUE FALSE TRUE`
- `[[4]]`
- `[1] 3`

# Naming List Elements

- The list elements can be given names and they can be accessed using these names.
- `list_data <- list(c("Jan","Feb","Mar"),  
matrix(c(3,9,5,1,-2,8), nrow = 2))`
- `names(list_data) <- c("1st Quarter",  
"A_Matrix")`
- `print(list_data)`

# Accessing List Elements

- Lists can be accessed in similar fashion to vectors. Integer, logical or character vectors can be used for indexing.
- Elements of the list can be accessed by the index of the element in the list.

- # Create a list containing a vector, a matrix and a list.
- `list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2), list("green",12.3))`
- # Give names to the elements in the list.
- `names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")`
- # Access the first element of the list.
- `print(list_data[1])`
- # Access the third element. As it is also a list, all its elements will be printed.
- `print(list_data[3])`
- # Access the list element using the name of the element.
- `print(list_data$A_Matrix)`

# Manipulating List Elements

- We can add, delete and update list elements as shown below.
- We can add and delete elements only at the end of a list. But we can update any element.

- `list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2), list("green",12.3))`
- `names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")`

`# Add element at the end of the list.`

- `list_data[4] <- "New element"`
- `print(list_data[4])`

`# Remove the last element.`

- `list_data[4] <- NULL`

`# Update the 3rd Element.`

```
list_data[3] <- "updated element"  
print(list_data[3])
```



# Merging Lists

- You can merge many lists into one list by placing all the lists inside `c()` function.
  - `list1 <- list(1,2,3)`
  - `list2 <- list("Sun","Mon","Tue")`
- # Merge the two lists.
- `merged.list <- c(list1,list2)`
- # Print the merged list.
- `print(merged.list)`

# Converting List to Vector

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

# Cont'd...

# Create lists.

- `list1 <- list(1:5)`
- `print(list1)`
- `list2 <- list(10:14)`
- `print(list2)`

# Convert the lists to vectors.

- `v1 <- unlist(list1)`
- `v2 <- unlist(list2)`
- `print(v1)`
- `print(v2)`

# Now add the vectors

- `result <- v1+v2`
- `print(result)`

# Matrices

- Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout.
- They contain elements of the same atomic types.
- Though we can create a matrix containing only characters or only logical values, they are not of much use.
- We use matrices containing numeric elements to be used in mathematical calculations.

- A Matrix is created using the `matrix()` function.
- Syntax-

`matrix(data, nrow, ncol, byrow, dimnames)`

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

- `M <- matrix(c(3:14), nrow = 4, byrow = TRUE)`
- `print(M)`
  
- `N <- matrix(c(3:14), nrow = 4, byrow = FALSE)`
- `print(N)`
  
- `rownames = c("row1", "row2", "row3", "row4")`
- `colnames = c("col1", "col2", "col3")`
- `P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames, colnames))`
- `print(P)`

[,1] [,2] [,3]

- [1,] 3 4 5
- [2,] 6 7 8
- [3,] 9 10 11
- [4,] 12 13 14

[,1] [,2] [,3]

- [1,] 3 7 11
- [2,] 4 8 12
- [3,] 5 9 13
- [4,] 6 10 14

col1 col2 col3

|      |    |    |    |
|------|----|----|----|
| row1 | 3  | 4  | 5  |
| row2 | 6  | 7  | 8  |
| row3 | 9  | 10 | 11 |
| row4 | 12 | 13 | 14 |

# Another way of creating a matrix

- By using `cbind()` and `rbind()` – If we are using `cbind()` function, elements will be filled column-wise and `rbind()` function fills the matrix elements row-wise.

- `M=cbind(c(1,2,3),c(4,5,6))`

- `M`

`[,1] [,2]`

- `[1,] 1 4`

- `[2,] 2 5`

- `[3,] 3 6`

- `M=rbind(c(1,2,3),c(4,5,6))`

- `M`



|        | [,1] | [,2] | [,3] |
|--------|------|------|------|
| • [1,] | 1    | 2    | 3    |
| • [2,] | 4    | 5    | 6    |

(3) By using `dim()` function- we can also create a matrix from a vector by setting its dimensions using `dim()`.

`M = c(1,2,3,4,5,6)`

`dim(M) = c(2,3)`

M

|        | [,1] | [,2] | [,3] |
|--------|------|------|------|
| • [1,] | 1    | 3    | 5    |
| • [2,] | 2    | 4    | 6    |

# Accessing Matrix Elements

- Matrix elements can be accessed in 3 different ways-
1. Integer vector as index- An element at the  $m$ th row and  $n$ th column of a matrix  $P$  can be accessed by the expression  $P[m,n]$ .
- We can use negative integers to specify rows or columns to be excluded.
  - If any field inside the bracket is left blank, it selects all.
  - For ex. the entire  $m$ th row of matrix  $P$  can be extracted as  $P[m,]$  and for column  $P[,n]$ .

- `M= matrix(c(1:12), nrow =4, byrow= TRUE)`

- `M`

      [,1] [,2] [,3]

- `[1,] 1 2 3`

- `[2,] 4 5 6`

- `[3,] 7 8 9`

- `[4,] 10 11 12`

1. `M[2,3]`

2. `M[2, ]`

3. `M[ ,3]`

4. `M[ , ]`

5. `M[ ,c(1,3)]`

6. `M[c(3,2) , ]`

7. `M[c(1,2) ,c(2,3)]`

8. `M[-1, ]`

1. [1] 6

2. [1] 4 5 6

3. [1] 3 6 9 12

4.     [,1] [,2] [,3]

[1,] 1    2    3

[2,] 4    5    6

[3,] 7    8    9

[4,] 10 11 12

5.     [,1] [,2]

[1,] 1    3

[2,] 4    6

[3,] 7    9

[4,] 10 12

6.      [,1] [,2] [,3]

[1,] 7    8    9

[2,] 4    5    6

7.      [,1] [,2]

[1,] 2    3

[2,] 5    6

8.      [,1] [,2] [,3]

[1,] 4    5    6

[2,] 7    8    9

[3,] 10 11 12

- Logical vector as index- Two logical vectors can be used to index a matrix. In such situation, rows and columns where the value is TRUE is returned.
- These indexing vectors are recycled if necessary and can be mixed with integers vectors.
- `M= matrix(c(1:12), nrow =4, byrow = TRUE)`
- `M[c(TRUE, FALSE,TRUE),c(TRUE,TRUE,FALSE)]`

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

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

```
[3,] 10  11
```

- Character vector as index – If we assign names to the rows and columns of a matrix, then we can access the elements by names.
- This can be mixed with integers or logical indexing.
- `M <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimname = list(c("r1", "r2", "r3", "r4"), c("c1", "c2", "c3")))`
- `M["r2", "c3"]` # elements at 2<sup>nd</sup> row, 3<sup>rd</sup> column
- `M[, "c1"]` # elements of the column named c1
- `M[TRUE, c("c1", "c2")]` # all rows and columns c1 & c2
- `M[2:3, c("c1", "c3")]` # 2<sup>nd</sup> & 3<sup>rd</sup> row, columns c1 & c3

[,1] [,2] [,3]

- [1,] 1 2 3
- [2,] 4 5 6
- [3,] 7 8 9
- [4,] 10 11 12

- [1] 6

- r1 r2 r3 r4  
1 4 7 10

c1 c2

- r1 1 2
- r2 4 5
- r3 7 8
- r4 10 11

c1 c2

- r1 4 6
- r2 7 9



# Matrix Arithmetic

- The dimensions ( no of rows and columns) should be same for the matrices involved in the operation.
- `Matrix1 <- matrix(c(10,20,30,40,50,60), nrow=2)`
- `Matrix2 <- matrix(c(1,2,3,4,5,6), nrow=2)`
- `Sum <- Matrix1 + Matrix2`
- `Difference <- Matrix1 - Matrix2`
- `Product <- Matrix1 * Matrix2`
- `Quotient <- Matrix1 / Matrix2`

[,1] [,2] [,3]

- [1,] 10 20 30
- [2,] 40 50 60

[,1] [,2] [,3]

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

# Matrix Manipulation

- We can modify a single element or elements based on a conditions.
- `Matrix1 <- matrix(c(10,20,30,40,50,60), nrow=2)`
- `Matrix1[2,2] <- 100`
- `Matrix1[ Matrix<40] <- 0`
- We can add row or column using `rbind()` and `cbind()`. Similarly it can be removed through reassignment.
- `cbind( Matrix1, c(1,2,3))`
- `rbind( Matrix1 , c(1,2))`
- `print( Matrix1 <- Matrix1[1:2, ]`

# Matrix Operations

1. Matrix Multiplication – Two matrices A of order MXN and B of order PXQ are eligible for multiplication only if N is equal to P.
  - The resultant matrix will be of the order MXQ.
  - Matrix multiplication is performed using the operator `A % * %B` where A and B are matrices.
  - `Matrix1 <- matrix(c(10,20,30,40,50,60), nrow=2)`
  - `Matrix2 <- matrix(c(1,2,3,4,5,6), nrow=2)`
  - `Product <- Matrix1 %*% Matrix2`
2. Transpose – The transpose of a matrix is an operation which flips a matrix over its diagonal, that is it switches the row and column indices of the matrix.
  - `Matrix1 <- matrix(c(10,20,30,40,50,60), nrow=2)`
  - `t( Matrix1)`

[,1] [,2]

- [1,] 10 40
- [2,] 20 50
- [3,] 30 60

[,1] [,2] [,3]

- [1,] 10 20 30
- [2,] 40 50 60

### 3. Cross product-

- `A<- matrix(c(10,20,30,40,50,60), nrow=2)`
- `B <- matrix(c(1,2,3,4,5,6), nrow=2)`
- `crossprod( A,B)`

### 4. Diagonal Matrix –

- `A <- matrix (1:9 , nrow =3)`
- `diag(A)` # prints the diagonal element
- `diag(3)` # create an identity matrix of order 3
- `diag( c(1,2,3) ,3)` # create a matrix of order 3 with diagonal elements 1,2,3.

[,1] [,2] [,3]

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

[,1] [,2] [,3]

- [1,] 1 0 0
- [2,] 0 1 0
- [3,] 0 0 1

[,1] [,2] [,3]

- [1,] 1 0 0
- [2,] 0 2 0
- [3,] 0 0 3

## 5. Row sum and column sum-

- `A<- matrix(c(10,20,30,40,50,60), nrow=2)`
- `rowSums(A)`
- `colSums(A)`

## 6. Row means and column means-

- `rowMeans(A)`
- `colMeans(A)`

## 7. Eigen values and eigen vectors-

- `Y <- eigen(A)`

## 8. Inverse –

- `solve(A)`



# Arrays

- Arrays are R data objects which can store data in more than two dimensions.
- Arrays can store only same data type.
- For ex. , if we create an array of dimension (2,4,5) then it creates 5 rectangular matrices each with 2 rows and 4 columns.
- An array is created using the `array()` function.
- It takes vectors as input. The function `dim()` defines the dimension of an array or use the values in the `dim` parameter to create an array.

- `V1= c(1,2,3)`
- `V2= c(10,20,30,40,50,60)`
- `A<- array(c(v1,v2),dim=c(3,3,2))`

, , 1

[,1] [,2] [,3]

- `[1,] 1 10 40`
- `[2,] 2 20 50`
- `[3,] 3 30 60`

, , 2

[,1] [,2] [,3]

- `[1,] 1 10 40`
- `[2,] 2 20 50`
- `[3,] 3 30 60`

- We can give names to the rows, columns and matrices in the array by using the `dimnames` parameter.
- `V1 = c(1,2,3)`
- `V2= c(10,20,30,40,50,60)`
- `Column.names <- c("col1","col2","col3")`
- `Row.names <- c("row1","row2","row3")`
- `Matrix.names <- c("matrix1","matrix2")`
- `A <- array(c(V1,V2), dim =c(3,3,2),dimnames = list(row.names,column.names,matrix.names))`

# Accessing Array Elements

- We can use the index position to access the array elements. Using index we can alter each and every individual element present in array.
- Syntax- `array_name [row_position, col_position, matrix_level]`
- `A <- array( 1:24, dim= c(3,4,2))`
- `A[1,2,1]` # 1<sup>st</sup> row 2<sup>nd</sup> col in matrix1.
- `A[3,4,2]`
- `A[3, , 1]` # only 3<sup>rd</sup> row in 1 matrix.
- `A[ , 4,2]` # 4<sup>th</sup> column in 2 matrix.
- `A[ , , 1]`
- `A[ , , 2]`

- |        | [,1] | [,2] | [,3] | [,4] |
|--------|------|------|------|------|
| • [1,] | 1    | 4    | 7    | 10   |
| • [2,] | 2    | 5    | 8    | 11   |
| • [3,] | 3    | 6    | 9    | 12   |

- |        | [,1] | [,2] | [,3] | [,4] |
|--------|------|------|------|------|
| • [1,] | 13   | 16   | 19   | 22   |
| • [2,] | 14   | 17   | 20   | 23   |
| • [3,] | 15   | 18   | 21   | 24   |

- [1] 4
- [1] 24
- [1] 3 6 9 12
- [1] 22 23 24

# Array Element Manipulation

- We can do calculations across the elements in an array using the `apply()` function.
- Syntax- `apply(x, margin, func)`
- X is an array, margin is the name of the dataset, func is function to be applied.
- `V1 <- c(1,2,3)`
- `V2 <- c(10,20,30,40,50,60)`
- `A<- array(c(V1,V2), dim=c(3,3,2))`
- `B <- apply(A, c(1), sum)`
- `C <- apply (C, c(2), sum)`

# Array Arithmetic

- To perform the arithmetic operations, we need to convert the multi-dimensional matrix into one dimensional matrix.
- `V1 <- c(1,2,3)`
- `V2 <- c(10,20,30,40,50,60)`
- `A<- array(c(V1,V2), dim=c(3,3,2))`
- `mat.a <- A[ , , 1]`
- `mat.b <- A[ , , 2]`
- `mat.a + mat.b`
- `mat.a - mat.b`
- `mat.a * mat.b`
- `mat.a / mat.b`

# Factors

- Factor is a data structure used for fields that takes only predefined finite number of values or categorical data.
- They are used to categorize the data and store it as levels.
- They can store both string and integers.
- For ex., A data field such as marital status may contain only values from single, married, separated, divorced and widowed. In such case, the possible values are predefined and distinct called levels.



# Creating factors

- factors are created with the help of *factor()* functions, by taking a vector as input.
- Factor contains a predefined set value called levels. By default, R always sorts levels in alphabetical order.
- `directions <- c("North", "North", "West", "South")`
- `factor(directions)`
- `o/p= levels: North, South, West`

# Accessing Factor

- There are various ways to access the elements of a factor in R. Some of the ways are as follows:
- `data <- c("East", "West", "East", "North")`
- `data[4]`
- `data[c(2,3)]`
- `data[-1]`
- `data[c(TRUE, FALSE, TRUE, TRUE)]`

# Modifying Factor

- To modify a factor, we are only limited to the values that are not outside the predefined levels.
- `print(data)`
- `data[2] <- "North"`
- `data[3] <- "South"`

# Data Frames

- A data frame is used for storing data tables.
- It is a list of vectors of equal length.
- A data frame is a table or a two-dimensional array like structure in which each column contains values of one variable and each row contains one set of values from each column.

# Characteristics of a data frame

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

# Creating Data Frames

- We can create data frames using the function `data.frame()`.
- The top line of the table called the header contains the column names.
- Each horizontal line afterward denotes a data row, which begins with the name of the row, and then followed by the actual data.
- Each data member of a row is called a cell.

- We can get the name of header using the function `names()`.
- No of rows using the function `nrow()`.
- No of column using the function `ncol()`.
- The `length()` function returns the length of the list which is same as that of no of columns.
- The structure of a data frame can be retrived using `str()` function.
- The statistical summary and nature of the data can be obtained by applying `summary()` function.

- `X <- data.frame("roll"=1:2,"name"=c("jack","jill"),"age"=c(20,22))`
- `print(X)`
- `names(X)`
- `nrow(X)`
- `ncol(X)`
- `str(X)`
- `summary(X)`



# Accessing Data Frame Components

- Components of data frame can be accessed like a list or like a matrix.
- (a) Accessing like a list – we can use either `[]` or `$` operator to access columns of data frame.
- Accessing with `[]` and `$` is similar.
- `X <-  
data.frame("roll"=1:2,"name"=c("jack","jill"),"age"=c(20,  
22))`
- `X$name`
- `X[["name"]]`
- `X[[3]]` # retrieves the value for the third col name as list

- (b) Accessing like a Matrix – Data frame can be accessed like a matrix by providing index for row and column.
- We can use the `[]` for indexing, this will return us a data frame unlike the other two `[[` and `$` will reduce it into a vector.
  - We can use the `head()` function to display first `n` rows.
  - Negative number for the index are also allowed in data frames.

- `X <-  
data.frame("roll"=1:3,"name"=c("jack","jill","Tom"),"age"=c(20,22,23))`
- `X["name"]`
- `X[1:2,]`
- `X[, 2:3]`
- `X[c(1,2),c(2,3)]`
- `X[,-1]`
- `X[-1,]`
- `X[X$age>21,]`
- `head(X,2)`

# Modifying Data Frames

- Data frames can be modified like we modified matrices through reassignment.
- `X <- data.frame("roll"=1:3,"name"=c("jack","jill","Tom"),"age"=c(20,22,23))`
- `X[1,"age"] <- 25`
- A data frame can be expanded by adding columns and rows.
- We can add the column vector using a new column name.
- Columns can also be added using the `cbind()` function.
- Similarly rows can be added using the `rbind()` function.
- Data frame columns can be deleted by assigning `NULL` to it.
- Similarly, rows can be deleted through reassignment.

- `print(X$bloodgroup <- c("A+", "B-", "AB+"))`

`# adding new column using cbind()`

- `print(X <-  
 cbind(X, city=c("delhi", "mumbai", "chennai")))`

`# adding new row using rbind()`

- `print(X <- rbind(X, c(4, "Jack", 24, "B+", "Delhi")))`

# Aggregating Data

- It is relatively easy to collapse data in R using one or more by variables and a defined function.
- When using the `aggregate()` function, the by variables must be in a list, even if there is only one column.
- The function can be built-in functions like `mean`, `max`, `min`, `sum` etc. or user provided function.

- `X <- data.frame("roll"=1:11,  
"name"=c("jack","jill","jeeva","smith","bob","smith","john",  
"mathew","charle","zen","yug"),  
"age"= c(20,20,30,21,19,21,19,18,22,25,21),  
"marks" = c(100,98,99,75,80,90,88,43,87,43,89))`
- `print(X)`
- `aggdata <- aggregate(X$marks,list(m=X$age),mean)`
- `print(aggdata)`
- `aggdata <- aggregate(X$marks,list(m=X$age),max)`
- `print(aggdata)`
- `aggdata <- aggregate(X$marks,list(m=X$age),sum)`
- `print(aggdata)`

# Sorting Data

- To sort a data frame in R, use the `order()` function.
- By default, sorting is ascending.
- We can sort in descending order by giving the sorting variable a minus sign in front.



- `X <-  
 data.frame("roll"=1:11,"name"=c("jack","jill","jeeva","smith",  
 "","bob","smith","john","mathew","charle","zen","yug"),  
 "age"= c(20,20,30,21,19,21,19,18,22,25,21),  
 "marks" = c(100,98,99,75,80,90,88,43,87,43,89))`

`#sort by name`

- `newdata <- X[order(X$name),]`

`# sort by age and within sort by name`

- `newdata <- X[order(X$age,X$name),]`

`# sort by age ascending and within age descending`

- `newdata <- X[order(X$name,-X$age),]`

# Merging Data

- We can merge two data frames(datasets) horizontally, by using the merge() function.
- In most cases, we can join two data frames by one or more common key variable(i.e. inner join).
- There are different types of join like inner join, outer join, left outer join, right outer join and cross join.
- Following are the points to be kept in mind while performing join operations-

1. An inner join of two data frames df1 and df2 returns only the rows in which the left table have matching keys in the right table.
2. An outer join of two data frames df1 and df2 returns all rows from both tables, join records from the left which have matching keys in the right table.
3. A left outer join(or simply left join) of two data frames df1 and df2 returns all rows from the left table, and any rows with matching keys from the right table.
4. A right outer join of two data frames df1 and df2 returns all rows from the right table, and any rows with matching keys from the left table.
5. A cross join of two data frames df1 and df2 returns a result set which is the number of rows in the first table multiplied by the no of rows in second table.

- `df1 =data.frame(CustomerId = c(1:6), product= c(rep("toaster",3),rep("radio",3)))`
- `df2 =data.frame(CustomerId = c(2,4,6), state= c(rep("alabama",2),rep("ohio",1)))`
- `print(df1)`
- `print(df2)`
- `# inner join`
- `merge(df1,df2, by= "CustomerId")`
- `# outer join`
- `merge(x=df1,y=df2, by= "CustomerId",all=TRUE)`
- `# left outer join`
- `merge(x=df1,y=df2, by= "CustomerId",all.x=TRUE)`
- `#Right outer join`
- `merge(x=df1,y=df2, by= "CustomerId",all.y=TRUE)`
- `#cross join`
- `merge(x=df1,y=df2, by= NULL)`

# Reshaping Data

- R provides a variety of methods for reshaping data prior to analysis.
- Two important functions for reshaping data are the `melt()` and `cast()` functions.
- These functions are available in reshape package.
- Before using these functions, make sure that the package is properly installed in your system.
- We can “melt” the data so that each row is a unique id-variable combination. Then we can “cast” the melted data into any shape we would like.

- `y <- data.frame("id"=c(1,2,1,2,1), "age"=c(20,20,21,21,19),  
"marks1"=c(80,60,70,80,90),"marks2"=c(100,98,99,75,80))`
- `print(y)`
- `#melting data`
- `mdata= melt(y, id=c("id","age"))`
- `# cast( data, formula, function)`
- `# mean marks for each id`
- `markmeans <- cast(mdata,id~variable,mean)`
- `# mean mark for each group`
- `agemmeans <- cast(mdata,age~variable,mean)`

# Subsetting Data

- The `subset()` function is the easier way to select variables and observations.
- In the following ex., we select all rows that have a value of age greater than or equal to 20 or age less than 10.
- Similarly we select all rows with `name="smith"` or `name="John"`.

- `X <- data.frame ("roll"=1:11,  
"name"=c("jack","jill","jeeva","smith","bob","smith","john",  
"mathew","charle","zen","yug"),  
"age"= c(20,20,30,21,19,21,19,18,22,25,21))`
- `print(X)`
- `newdata <-  
subset(X,age>=25&age<30,select=c(roll,name,age))`
- `print(newdata)`
- `newdata <-  
subset(X,name=="smith" | name=="john",select=roll:age)`
- `print(newdata)`



# Data Type Conversion

- We can convert one data type to another data type as in any programming language.
- We can convert any basic data type to numeric using the function `as.numeric()`.
- Similarly `as.integer()` converts to integer, `as.character()` converts to character, `as.logical()` converts to logical and `as.complex()` converts to complex data types.

# Unit 3

## Conditions and loops

- Decision making structures are used by the programmer to specify one or more conditions to be evaluated or tested by the program.
- A statement or statements need to be executed if the condition is TRUE and optionally other statements to be executed if the condition is FALSE.

# Decision Making

- R provides the following types of decision making statements which includes if statement, if..else statement, nested if...else statement, ifelse() function and switch statement.

# if Statement

- An if statement consists of a boolean expression followed by one or more statements. The syntax is-
- If( boolean\_expression)  
{  
    // statement will execute if the boolean expression is true.  
}

- If the `boolean_expression` evaluates to `TRUE`, then the block of code inside the `if` statement will be executed.
- If `boolean_expression` evaluates to `FALSE`, then the first set of code after the end of `if` statement will be executed.
- Here `boolean expression` can be a logical or numeric vector, but only the first element is taken into consideration.
- In the case of numeric vector, zero is taken as `FALSE`, rest as `TRUE`.

- `x <- 10`  
 if (`x > 0`)  
 {  
 cat(x, " is a positive number\n")  
 }

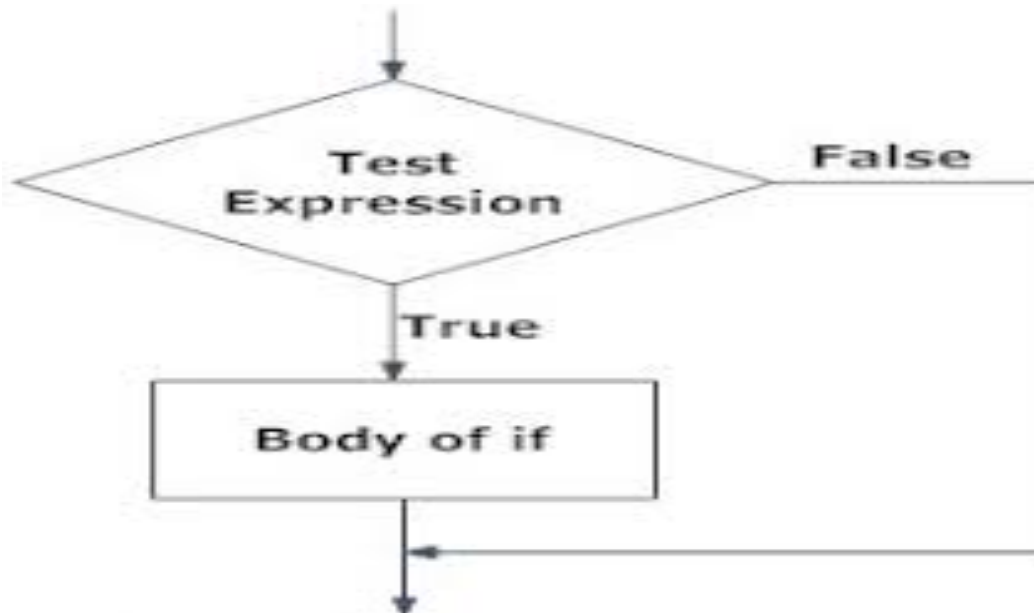
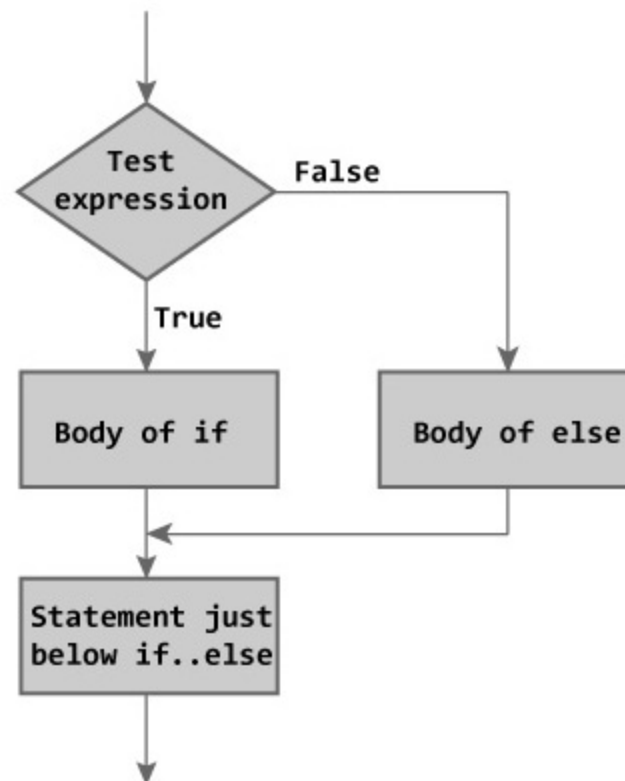


Fig: Operation of if statement

# if....else Statement

- An if statement can be followed by an optional else statements which executes when the boolean expression is FALSE.
- The syntax of if...else is-  
if (boolean\_expression)  
{  
    // if expression is true  
}  
else  
{  
    // if expression is false  
}

- If the `boolean_expression` evaluates to be `TRUE`, then if block of code will be executed, otherwise else block of code will be executed.





- `X <- -5`  
`if(x > 0){`  
`cat( x, "is a positive number\n")`  
`} else {`  
`cat( x, "is a negative number\n")`  
`}`
- We can write the if...else statement in a single line if the “if and else” block contains only one statement as follows.
- `if( x>0) cat ( x, "is a positive no\n") else cat(x, "is a negative no\n")`

# Nested if...else Statement

- An if statement can be followed by an optional else if..else statement, which is very useful to test various conditions using single if...else if statement.
- We can nest as many if..else statement as we want.
- Only one statement will get executed depending upon the boolean\_expression.

- ```
if( boolean_expression 1) {  
    // execute when expression 1 is true.  
} else if(boolean_expression 2) {  
    // execute when expression 2 is true.  
} else if(boolean_expression 3) {  
    // execute when expression 3 is true.  
} else {  
    // execute when none of the above condition is  
    true.  
}
```

- `X <- 19`  
`if (x < 0)`  
`{`  
 `cat(x, "is a negative number")`  
`} else if (x > 0)`  
`{`  
 `cat(x, "is a positive number")`  
`}`  
`else`  
 `print("zero")`

# ifelse() function

- Most of the function in R take vector as input and output a resultant vector.
- This vectorization of code, will be much faster than applying the same function to each element of the vector individually.
- There is an easier way to use if..else statement specifically for vectors in R.
- We can use if...else() function instead which is the vector equivalent form of the if..else statement.

- `ifelse(boolean_expression, x, y)`
- Here, `boolean_expression` must be a logical vector.
- The return value is a vector with the same length as `boolean_expression`.
- This returned vector has element from `x` if the corresponding value of `boolean_expression` is `TRUE` or from `Y` if the corresponding value of `boolean_expression` is `FALSE`.
- For example, the *i*th element of result will be `x[i]`, if `boolean_expression[i]` is `TRUE` else it will take the value of `y[i]`.
- The vectors `x` and `y` are recycled whenever necessary.

- `a = c(5,7,2,9)`  
`ifelse( a %% 2 == 0 , "even" ,"odd")`
- `o/p = ?`
- In the above example, the `boolean_expression` is `a %% 2 ==0` which will result into the `vector(FALSE, FALSE,TRUE,FALSE)`.
- Similarly, the other two vectors in the function argument gets recycled to `("even", "even", "even", "even")` and `("odd", "odd", "odd", "odd")` respectively.
- Hence the result is evaluated accordingly.

# switch Statement

- A switch statement allows a variable to be tested for equality against a list of values.
- Each value is called a case, and the variable being switched on is checked for each case.
- `switch( expression, case1, case2, case3....)`
- If the value of expression is not a character string, it is coerced to integer.
- We can have any no of case statements within a switch.
- Each case is followed by the value to be compared to and a colon.



- If the value of the integer is between 1 and `nargs()-1` { the max no of arguments} then the corresponding element of case condition is evaluated and the result is returned.
- If expression evaluates to a character string then the string is matched(exactly) to the names of the elements.
- If there is more than one match, the first matching element is returned.
- No default argument is available.

- Switch( 2, “red”, “green”, “blue”)
- Switch(“color”, “color” = “red”, “shape” = “ square” ,  
”length “=5)
- Output- [1] “green”  
              [2] “red”
- If the value evaluated is a number, that item of the list is returned.
- In the above example, “red”, “green”, ”blue” from a three item list. The switch() function returns the corresponding item to the numeric value evaluated.
- In the above example, green is returned.
- The result of the statement can be a string as well.
- In this case, the matching named item’s value is returned.
- In the above example, “color” is the string that is matched and its value “red” is returned.

# Loops

- In General, statements are executed sequentially.
- Loops are used in programming to repeat a specific block of code.
- R provides various looping structures like for loop, while loop and repeat loop.

# for loop

- A for loop is a repetition control structure that allow us to efficiently write a loop that needs to execute a specific number of times.
- A for loop is used to iterate over a vector in R programming.

```
for ( value in sequence)  
{  
    statements  
}
```

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

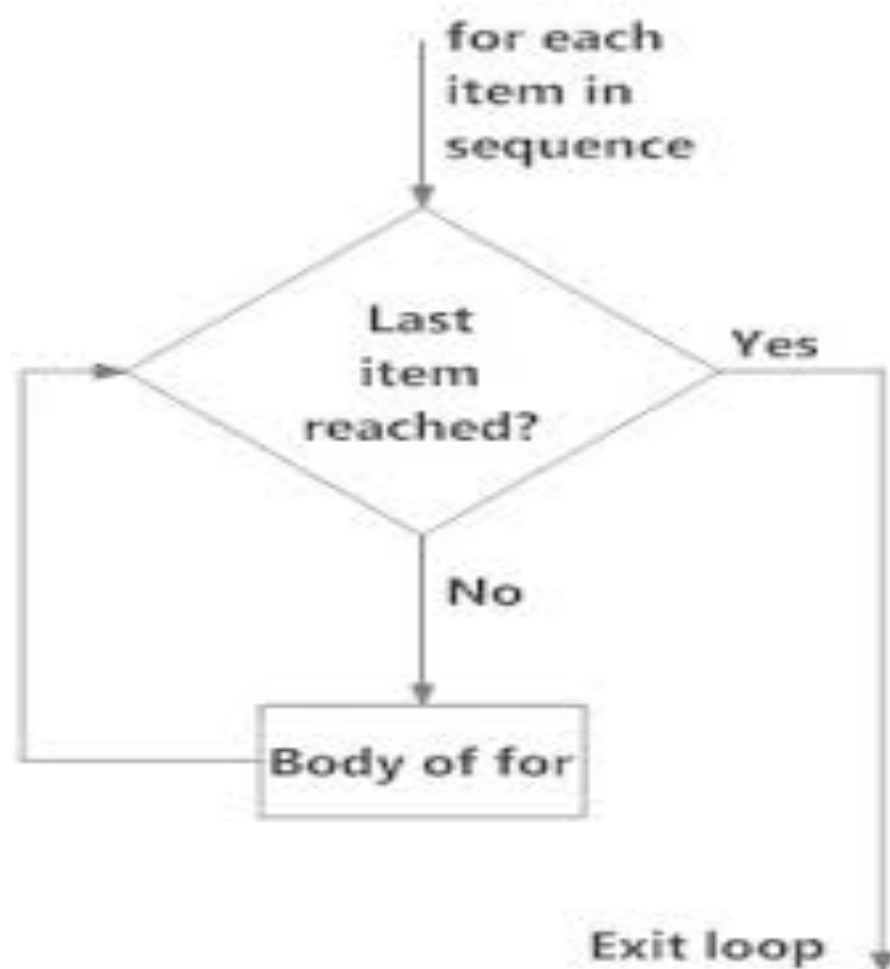


Fig: operation of for loop

- `X <- c(2,5,3,9,8,11,6)`

`count <- 0`

`for(val in X)`

`{`

`if (val %% 2 == 0)`

`count = count+1`

`}`

`cat( "no of even numbers in", X, "is", count, "\n")`

- o/p = ?

- The for loop in R is flexible that they are not limited to integers in the input.
- We can pass character vector, logical vector, lists or expressions.
- Ex-
- ```
V <- c( "a", "e", "i", "o", "u")  
for ( vowel in V)  
{  
  print(vowel)  
}
```
- o/p- ?

# while loop

- while loops used to loop until a specific condition is met.
- Syntax-  

```
while ( test_expression)  
{  
    statement  
}
```
- Here, test expression is evaluated and the body of the loop is entered if the result is TRUE.
- The statements inside the loop are executed and the flow returns to evaluate the test\_expression again.
- This is repeated each time until test\_expression evaluated to FALSE, in which case, the loop exits.



num=5

sum=0

while(num>0)

{ sum= sum + num

num= num - 1

} cat( "the sum is", sum, "\n")

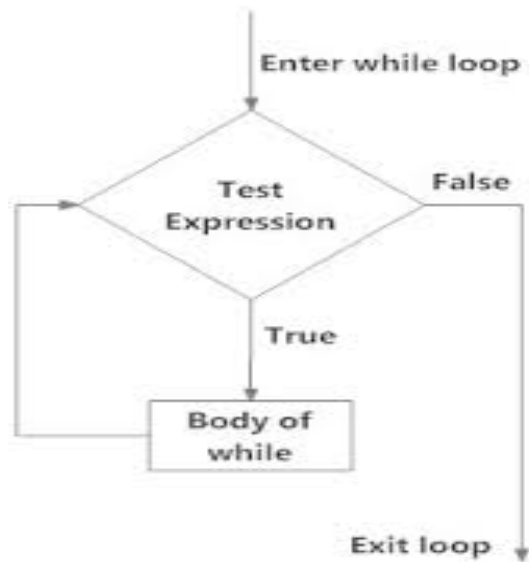
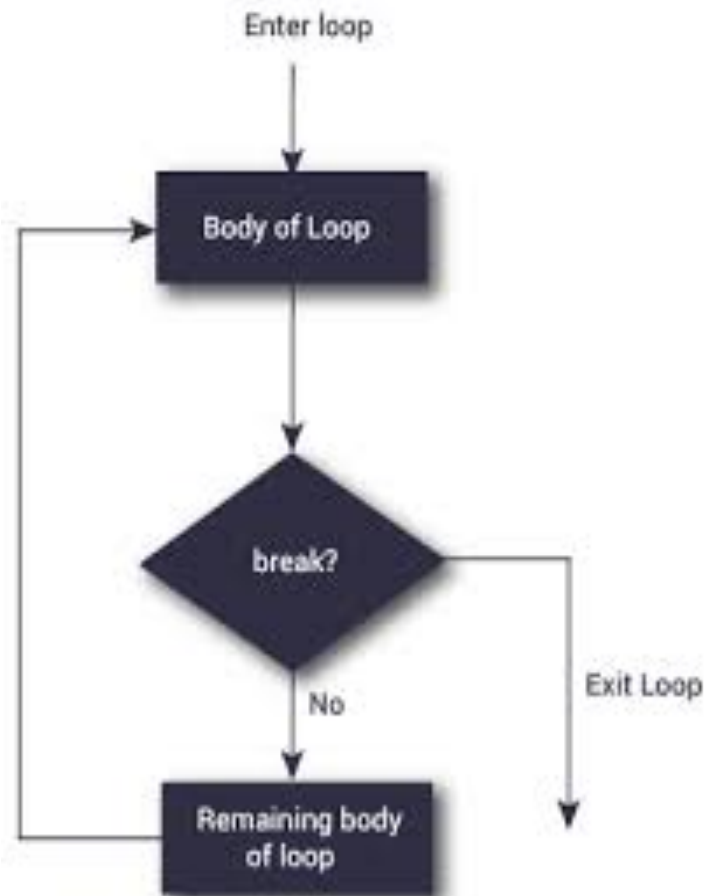


Fig: operation of while loop

# repeat loop

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

```
repeat {  
    Statements  
if( condition)  
    {  
        Break  
    }  
}
```



# Loop Control Statements

- Loop control statements are also known as jump statements.
- Loop control statements change execution from its normal sequence.
- When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- The loop control statements in R are break statement and next statement.

# 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 exists from the innermost loop that is being evaluated.

- `x<- 1:10`  
`for( val in x) {`  
    `if (val == 3) {`  
        `break`  
    `}`  
    `print(val)     }`
- o/p = ?
- In the above example, we iterate over the vector x, which has consecutive numbers from 1 to 10.
- Inside the for loop we have used an if condition to break if the current value is equal to 3.

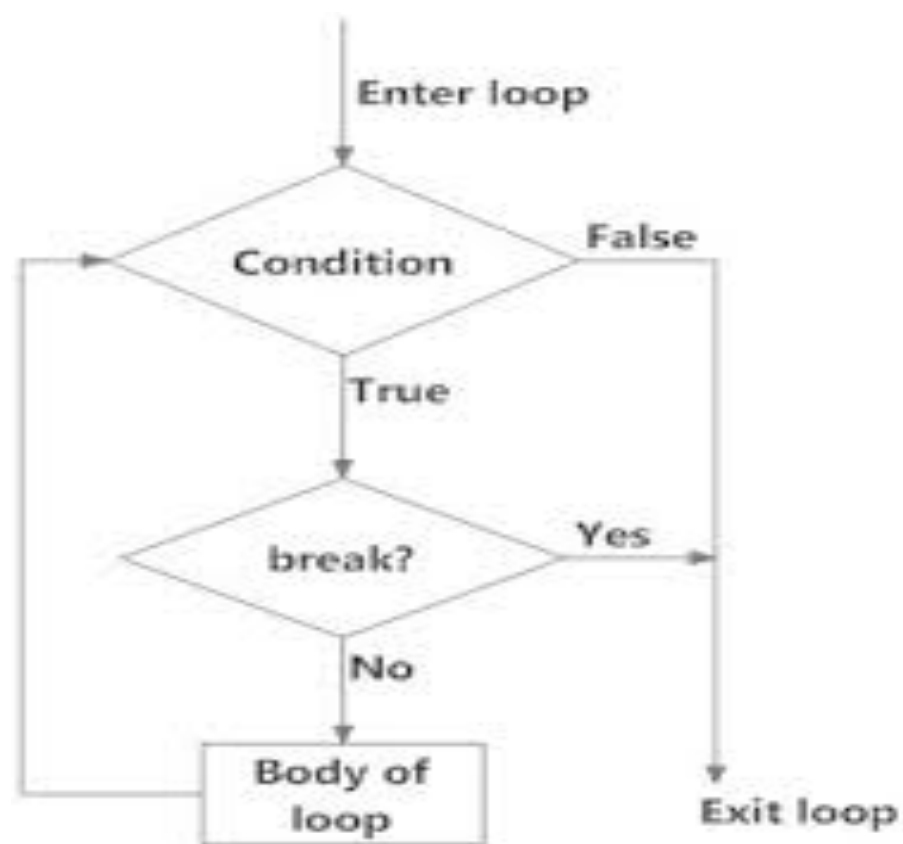


Fig: flowchart of break

# 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 loop.
- This is equivalent to the continue statement in C, java and python.



- `X <- 1:10`  
`for( val in X) {`  
    `if ( val == 3) {`  
        `next`  
    `}`  
    `print( val)`  
}
- 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.

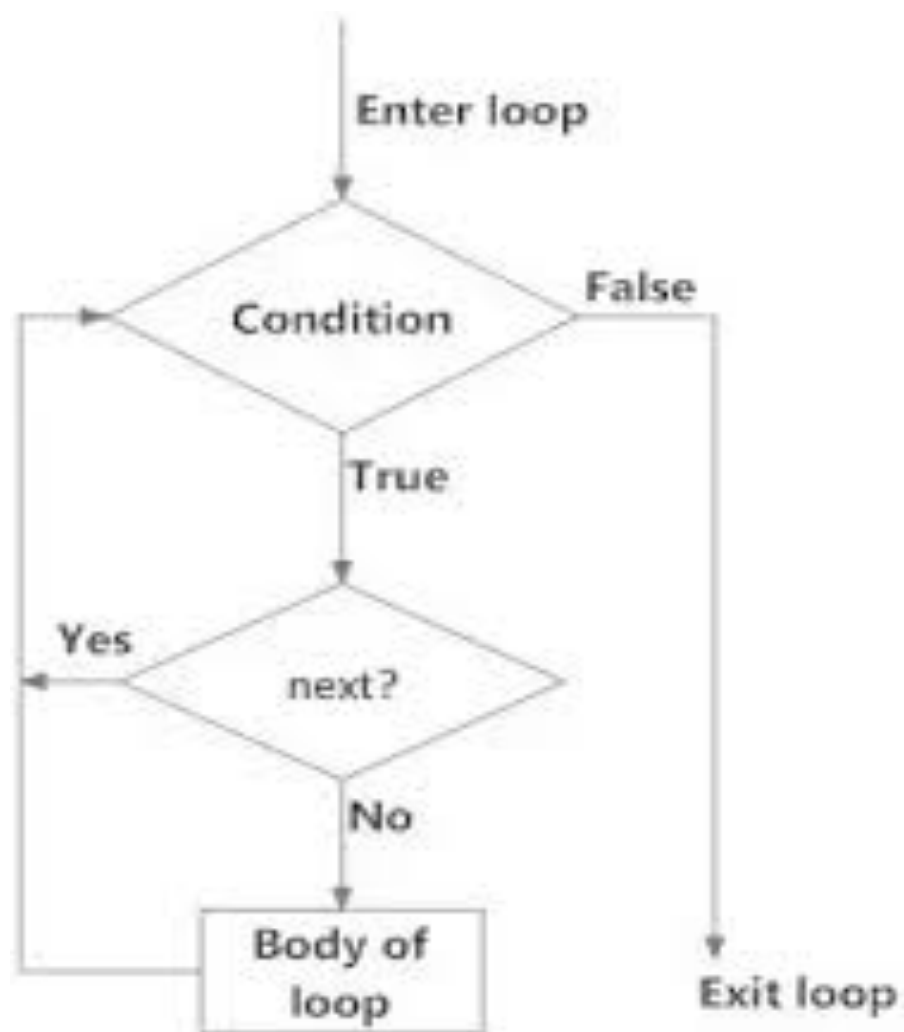


Fig: flowchart of next

# Functions

- Functions are used to logically break our code into simpler parts which becomes easy to maintain and understand.
- A function is a set of statements organized together to perform a specific task.
- R has a large no of built-in functions and the user can create their own functions.
- A function is an object, with or without arguments.

# Function Definition

- The reserved word function is used to declare a function in R.
- `func_name <- function(argument)`  
  {  
    Statement  
  }
- Here, the reserved word function is used to declare a function in R.
- This function object is given a name by assigning it to a variable, `func_name`.
- The statements within the curly braces form the body of the function. These braces are optional if the body contains only a single expression.

- Following are the components of a function in R-
  1. Function Name – This is the actual name of the function. It is stored in R environment as an object with this name.
  2. Arguments – When a function is invoked, we can pass values to the arguments. Arguments are optional. A function may or may not contain arguments. The arguments can also have default values.
  3. Function Body – The function body contains a collection of statements that defines what the function does.

# Function Calling

- We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like build-in functions.
- ```
power <- function(x,y)
{
  result <- x^y
  cat( x, "raised to the power", y, "is", result, "\n")
}
```
- ```
power(2,3)
```
- Here, the arguments used in the function declaration `x` and `y` are called formal arguments and those used while calling the function are called actual argument.

# Function without Arguments

- It is possible to create a function in R without arguments.
- ```
square <- function()  
{  
  for( i in 1:5)  
    cat("square of", i, "is", (i*i), "\n")  
}
```
- ```
square()
```

# Function with named Arguments

- When calling a function in this way, the order of the actual arguments does not matter or we can pass the arguments in a shuffled order.
- For example, all the function calls given below are equivalent.
- ```
power <- function(x,y)
{
  result <- x^y
  cat( x, "raised to the power", y, "is", result, "\n")
}
```
- ```
power(2,3)
```
- ```
Power(x=2,y=3)
```
- ```
Power(y=3,x=2)
```



- Further we can use named and unnamed arguments in a single function call.
- In such case, all the named arguments are matched first and then the remaining unnamed arguments are matched in a positional order
- `power( x=2,3)`
- `power(2, y=3)`

# Function with default Arguments

- We can assign default values to arguments in a function in R. This is done by providing an appropriate value to the formal argument in the function declaration.
- The function named `power` is defined with a default value for `Y` in the following example program. If no value is passed for `Y`, then the default value is taken.
- If the value is passed for `Y`, then the default value will be overridden.

- ```
power <- function(x,y=2)
{
  result <- x^y
  cat( x, "raised to the power", y, "is", result,
    "\n")
}
```
- ```
power(2)
```
- ```
Power(2,3)
```

# Built-in Functions

- There are several built-in functions available in R. These functions can be directly used in user written program.
- The built-in functions can be grouped into mathematical functions, character functions, statistical functions, probability functions, date functions, time functions and other useful functions.

# Mathematical functions

1. `abs()`- this function computes the absolute value of numeric data.
  - The syntax is `abs(x)`, where `x` is any numeric value, array or vector.
  - `abs(-1)`
  - `x <- c( -2,4,0,45,9,-4)`
  - `abs(x)`
  - `x <- matrix (c( -3,5,-7,1,-9,4), nrow= 3, ncol=2, byrow=TRUE)`
  - `abs(x[1, ])`
  - `abs (x[, 1])`

2. Sin(), cos() and tan()- the function sin() computes the sine value, cos() computes the cosine value and tan() computes the tangent value of numeric data in radians.

- Syntax is sin(x), cos(x), tan(x), where x is any numeric, array or vector.
- sin(10) , cos(90) , tan(50)
- x <- c( -2,4,0,45,9,-4)
- sin(x) , cos(x) , tan(x)
- x <- matrix (c( -3,5,-7,1,-9,4), nrow= 3, ncol=2, byrow=TRUE)
- sin(x[1, ]) ,cos(x[,1 ]), tan(x[1,])

3. `asin()`, `acos()` and `atan()` – the `asin()` computes the inverse sine value, `acos()` computes inverse cosine value and `atan()` computes inverse tangent value of numeric data in radians.

- `asin(1)`, `acos(1)`, `atan(50)`

4. `exp(x)` – the function computes the exponential value of a number or number vector,  $e^x$ .

- `x=5` , `exp(x)`

5. `ceiling`- This function returns the smallest integer larger than the parameter.

- `x <- 2.5`
- `Ceiling(x)`
- 3

6. floor- This function returns the largest integer not greater than the giving number.

- `x <- 2.5`
- `floor(x)`

7. round()- This function returns the integer rounded to the giving number.

- The syntax is `round( x, digits=n)`, where `x` is numeric variable or a vector and `digit` specifies the number of digits to be rounded.
- `x<- 2.587888`
- `round(x,3)`



7. `trunc()`- This function returns the integer truncated with decimal part.

- `x <- 2.99`
- `trunc(x)`

8. `signif(x, digits=n)`- This function rounds the values in its first argument to the specified number of significant digits.

- `x <- 2.587888`
- `Signif (x,3)`
- `2.59`

10. `log()`, `log10()`, `log2()`, `log(x,b)`- `log()` function computes natural logarithms for a number or vector.

11. `max()` and `min()` – `max()` function computes the maximum value of a vector and `min()` function computes the minimum value of a vector.

- `x <- c(10, 289, -100, 8000)`
- `max(x)` , `min(x)`

12. `beta()` and `lbeta()`- `beta()` function returns the beta value and `lbeta()` returns the natural logarithm of the beta function.

- `beta(4,9)`
- `lbeta(4,9)`

o/p - 0.0005050, -7.590852

13. `gamma()`- this function returns the gamma function  $\Gamma x$ .

- `x=5`
- `gamma(x)`
- o/p – 24

14. `factorial ()`- this function computes factorial of a number or a numeric vector.

- `x=5`
- `factorial(x)`

# Character Function

- These functions are used for string handling operations like extracting characters from a string, extracting substrings from a string, concatenation of strings, matching strings, inserting strings, converting strings from one case to another and so on.
1. `agrep()`- this function searches for approximate matches to pattern within each element of the string.
- `agrep( pattern, x, ignore.case=FALSE, value=FALSE, max.distance=0.1, useBytes= FALSE)`

- `x <- c("R language", "and", "SAND")`
- `agrep("an", x)`
- `agrep("an", x, ignore.case=TRUE)`
- `agrep("uag", x, ignore.case=TRUE, max=1)`
- `agrep("uag", x, ignore.case=TRUE, max=2)`

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

2. `char.expand()`- This function seeks for a unique match of its first argument among the elements of its second.

- If successful, it returns this element, otherwise, it performs an action specified by the third argument. The syntax is as follow-

```
char.expand( input, target, nomatch= stop("no match"), warning())
```

- Where `input` is the character string to be expanded, `target` is the character vector with the values to be matched against, `nomatch` is an R expression to be evaluated in case expansion was not possible and `warning` function prints the warning message in case there is no match.
- The match string searches only in the beginning.

- `x<- c("sand", "and", "land")`
- `char.expand("an", x, warning("no expand"))`
- `char.expand("a", x, warning("no expand"))`

3. `charmatch()`- This function finds matches between two arguments and returns the index position.

- `charmatch( x, table, nomatch= NA_integer_)`
- Where `x` gives the value to be matched, `table` gives the value to be matched against and `nomatch` gives the value to be returned at non matching positions.

- `charmatch("an", c("and", "sand"))`
- `charmatch("an", "sand")`
- `[1] 1`
- `[1] NA`

4. `charToRaw` – This function converts character to ASCII or “raw” objects.

- `x <- charToRaw("a")`
- `Y <- charToRaw("AB")`
- `[1] 61`
- `[1] 41 42`



5. `chartr()` – this function is used for character substitutions.

- `chartr(old, new, x)`
- `x <- "apples are red"`
- `chartr("a", "g", x)`

6. `dquote()`- this function is used for putting double quotes on a text.

- `x <- '2013-06-12'`
- `dquote(x)`

7. `format()`- numbers and strings can be formatted to a specific style using `format()` function.

- Ex- `format(x, digits, nsmall, scientific, width, justify= c("left", "right", "centre", "none"))`

8. `gsub()`- this function replaces all matches of a string, if the parameter is a string vector, returns a string vector of the same length and with the same attributes.

- `gsub(pattern, replacement, x, ignore.case=FALSE)`

Ex- `x<- "apples are red"`

`gsub("are", "were", x)`

o/p- `"apples were red"`

9. `nchar()` & `nzchar()`- This function determines the size of each elements of a character vector. `nzchar()` tests whether elements of a character vector are non-empty strings.

Syn- `nchar(x, type="chars", allowNA= FALSE)`

syn- `nzchar()`

10. `noquote()`- This function prints out strings without quotes. The syntax is `noquote(x)` where `x` is a character vector.

Ex- letters

`noquotes(letters)`

11. paste()- Strings in R are combined using the paste() function. It can take any number of string arguments to be combined together.

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

- Where.... Represents any number of arguments to be combined, sep represents any separator between the arguments. It is optional.
- Collapse is used to eliminate the space in between two strings but not the space within two words of one string.
- Ex- a <- "hello"
- b <- "everyone"
- print(paste(a,b,c))
- print( paste(a,b,c, sep = "-" ))
- print( paste(a,b,c, sep = "", collapse = ""))

12. `replace()`- This function replaces the values in `X` with indices given in `list` by those given in `values`. If necessary, the values in 'values' are recycled.

syn- `replace( x, list, values)`

Ex- `x <- c("green", "red", "yellow")`

`y <- replace(x,1,"black")`

13. `sQuote()`- This function is used for putting single quote on a text.

`X <- "2013-06-12 19:18:05"`

`sQuote(X)`

14. `strsplit()`- This function splits the elements of a character vector `x` into substrings according to the matches to substring split within them.

Syn- `strsplit( x, split)`

15. substr()- This function extracts or replace substrings in a character vector.

Syn- substr( x, start, stop)

substr( x, start, stop) <- value

Ex- substr( "programming", 2,3)

x= c("red", "blue", "green", "yellow")

Substr(x,2,3) <- "gh"

16. tolower() – This function converts string to its lower case.

Syn- tolower("R Programming")

17. toString – This function produces a single character string describing an R object.

Syn- toString(x)

toString( x, width = NULL)

18. toupper- This function converts string to its upper case.

Syn- toupper("r programming")

# Statistical Function

1. `mean()`- The function `mean()` is used to calculate average or mean in R.

Syn- `mean(x, trim= 0, na.rm = FALSE)`

Trim is used to drop some observation from both end of the sorted vector and `na.rm` is used to remove the missing values from the input vector.

2. `median()`- the middle most value in a data series is called the median. The `median()` function is used in R to calculate this value.

Syn- `median(x, na.rm= FALSE)`



3. `var()`- returns the estimated variance of the population from which the no in vector x are sampled.

Syn- `x<- c(10,2,30,2,5,8)`  
`var(x, na.rm= TRUE)`

4. `sd()`- returns the estimated standard deviation of the population from which the no in vector x are sampled.

Syn- `sd(x, na.rm= TRUE)`

5. `scale()`- returns the standard scores(z-score) for the no in vector in x. Used to standardizing a matrix.

Syn- `x<- matrix(1:9, 3,3)`  
`scale(x)`

6. `sum()`- adds up all elements of a vector.

Syn- `sum(X)`

`sum(c(1:10))`

7. `diff(x,lag=1)`- returns suitably lagged and iterated differences.

Syn- `diff(x, lag, differences)`

Where `X` is a numeric vector or matrix containing the values to be differenced, `lag` is an integer indicating which lag to use and `differences` is an integer indicating the order of the difference.

- For ex., if `lag=2`, the difference between third and first value, between the fourth and the second value are calculated.
- The attribute `differences` returns the differences of differences.

8. range()- returns a vector of the minimum and maximum values.

Syn- `x<- c(10,2,14,67,86,54)`

`range(x)`

o/p- 2 86

9. rank()- This function returns the rank of the numbers( in increasing order) in vector x.

Syn- `rank(x, na.last = TRUE)`

10. Skewness- how much differ from normal distribution.

Syn- `skewness(x)`

# Date and Time Functions

- R provides several options for dealing with date and date/time.
  - Three date/time classes commonly used in R are Date, POSIXct and POSIXlt.
1. Date – date() function returns a date without time as character string. Sys.Date() and Sys.time() returns the system's date and time.

```
Syn <- date()
```

```
    Sys.Date()
```

```
    Sys.time()
```

- We can create a date as follows-
- Dt <- as.Date("2012-07-22")

- While creating a date, the non-standard must be specified.
- `Dt2 <- as.Date("04/20/2011" , format = "%m%d%Y")`
- `Dt3 <- as.Date("October 6, 2010", format = "%B %d,%Y")`

|    |                                                                                      |                       |
|----|--------------------------------------------------------------------------------------|-----------------------|
| %Y | Year with century<br>on input:<br>00 to 68 prefixed by 20<br>69 to 99 prefixed by 19 | 1984, 2005            |
| %C | Century                                                                              | 19, 20                |
| %D | Date formatted %m/%d/%y                                                              | 05/27/84,<br>07/07/05 |
| %u | Weekday<br>1-7<br>Monday is 1                                                        | 7, 4                  |

## Date Formats

| Conversion specification | Description                                           | Example          |
|--------------------------|-------------------------------------------------------|------------------|
| %a                       | Abbreviated weekday                                   | Sun, Thu         |
| %A                       | Full weekday                                          | Sunday, Thursday |
| %b or %h                 | Abbreviated month                                     | May, Jul         |
| %B                       | Full month                                            | May, July        |
| %d                       | Day of the month<br>01-31                             | 27, 07           |
| %j                       | Day of the year<br>001-366                            | 148, 188         |
| %m                       | Month<br>01-12                                        | 05, 07           |
| %U                       | Week<br>01-53<br>with Sunday as first day of the week | 22, 27           |
| %w                       | Weekday<br>0-6<br>Sunday is 0                         | 0, 4             |
| %W                       | Week<br>00-53<br>with Monday as first day of the week | 21, 27           |
| %x                       | Date, locale-specific                                 |                  |
| %y                       | Year without century<br>00-99                         | 84, 05           |

2. POSIXct- If we have times in your data, this is usually the best class to use. In POSIXct, “ct” stands for calendar time.

- We can create some POSIXct objects as follows.

```
Tm1<- as.POSIXct("2013-07-24 23:55:26")
```

o/p – “2013-07-24 23:55:26 PDT”

```
Tm2 <- as.POSIXct("25072012 08:32:07", format=
"%d%m%Y %H:%M:%S")
```

- We can specify the time zone as follows.

```
Tm3<-      as.POSIXct("2010-12-01      11:42:03",
tz="GMT")
```

- Times can be compared as follows.
  - $Tm2 > Tm1$
  - We can add or subtract seconds as follows.
  - $Tm1 + 30$
  - $Tm1 - 30$
  - $Tm2 - Tm1$
3. POSIXlt- This class enables easy extraction of specific components of a time. In POSIXlt, “lt” stands for local time.
- “lt” also helps one remember that POSIXlt objects are lists.
  - $Tm1.ltm1 <- as.POSIXlt("2013-07-24 23:55:26")$
  - o/p- “2013-07-24 23:55:26”



- We can extract the components in time as follows.

- `unlist(Tm1.lt)`

`sec min hour mday mon year wday yday isdat`

`26 55 23 24 6 113 3 204 1`

- `mday`, `wday`, `yday` stands for day of the month, day of the week and day of year resp.
- A particular component of a time can be extracted as follows.
- `Tm1.lt$sec`
- we can truncate or round off the times as given below.
- `trunc(Tm1.lt, "days")` o/p - "2013-07-24"
- `trunc(Tm1.lt, "mins")` o/p - "2013-07-24 23:55:00"

# Other Functions

1. `rep( x, ntimes)` – This function repeats `x` `n` times.

Ex.- `rep( 1:3,4)`

2. `cut( x,n)`- divide continuous variable in factor with `n` levels.

`X<- c(1,2,3,1,2,3,1)`

`cut( X,2)`

# Recursive Function

- A function that calls itself is called a recursive function and this technique is known as recursion.
- This special programming technique can be used to solve problems by breaking them into smaller and simpler sub- problems.
- Recursive functions call themselves. They break down the problem into the smallest possible components.
- The function() calls itself within the original function() on each of the smaller components. After this, the results will be put together to solve the original problem.

- recursive.factorial <- function(x)
- {
- if ( x == 0)
- return (1)
- else
- return ( X \* recursive.factorial( X-1))
- }
- recursive.factorial (5)

## Convert decimal number to binary-

- `convert_to_binary <- function(n)`
- `{`
- `if ( n>1)`
- `{`
- `convert_to _binary(as.integer(n/2))`
- `}`
- `cat ( n%%2)`
- `}`
- `convert_to_binary(5)`

# Classes and objects

- Everything in R is an object.
- An object is a data structure having some attributes and methods which act on its attributes.
- Class is a blueprint for the object. We can think of class like a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house.
- House is the object. As, many houses can be made from a description, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called instantiation.
- While most programming languages have a single class system, R has three class systems. Namely, S3, S4 and more recently Reference class systems.
- They have their own features and peculiarities and choosing one over the other is a matter of preference.

# S3 Class

- S3 class is somewhat primitive in nature. It lacks a formal definition and object of this class can be created simply by adding a class attribute to it.
- This simplicity accounts for the fact that it is widely used in R programming language. In fact most of the R built-in classes are of this type.
- S3 is used to overload any function. Therefore, we can call different names of the function. And, it depends upon the type of input parameter or the number of a parameter.
- The class of an object is determined by its class attribute, a character vector of class names.

- S3 is different from conventional programming languages like Java, C++, and C# that implement message passing OO. This makes S3 easier to implement.
- The below ex. shows how to define a function that will create and return an object of a given class. A list is created with the relevant members, the list's class is set, and a copy of the list is being returned. The components of the list become the member variables of the object.
- `s <- list(name = "ABC", age = 29, GPA = 4.0)`
- `class(s) <- "student"`
- `S`
- o/p- `$name [1] "ABC"`  
`$age [1] 29`  
`$GPA [1] 4.0`  
`attr(,"class")`  
`[1] "student"`



- How to use constructors to create objects?
- It is a good practice to use a function with the same name as class (not a necessity) to create objects.
- This will bring some uniformity in the creation of objects and make them look similar.
- We can also add some integrity check on the member attributes. Here is an example. Note that in this example we use the attr() function to set the class attribute of the object.
- # a constructor function for the "student" class

```
student <- function(n,a,g)
{
  if(g>4 || g<0) stop("GPA must be between 0 and 4")
  value <- list(name = n, age = a, GPA = g)
  # class can be set using class() or attr() function
  attr(value, "class") <- "student"
  value }
```

# Methods and Generic Functions

- In the above example, when we simply write the name of the object, its internals get printed. In interactive mode, writing the name alone will print it using the `print()` function.
- How does `print()` know how to print these variety of dissimilar looking object?
- The answer is, `print()` is a generic function. Actually, it has a collection of a number of methods. You can check all these methods with `methods(print)`.
- Printing our object of class "student" looks for a method of the form `print.student()`, but there is no method of this form.
- So, which method did our object of class "student" call? It called `print.default()`. This is the fallback method which is called if no other match is found. Generic functions have a default method.

- Now, we will make a generic function – GPA  
GPA <- **function**(obj)  
{  
  **UseMethod**("GPA")  
}
- Now, we will make a new method for the class “student”  
GPA.student <- **function**(obj)  
{  
  cat("Total GPA is", obj\$GPA, "\n")  
}

# S4 class

- S4 class are an improvement over the S3 class. They have a formally defined structure which helps in making object of the same class look more or less similar.
- Class components are properly defined using the `setClass()` function and objects are created using the `new()` function.
- We specify a function to verify that the data is consistent (validation) and also specify the default values (the prototype).
- `setClass ("student", slots=list(name="character", age="numeric", GPA="numeric"))`

- There are two functions *is.object* and the *isS4* commands.
- We use *is.object* command to determine whether a variable refers to an object or not.
- We use the *isS4* command to determine whether a variable is an S4 object or not.
- The importance of the commands is that the *isS4* command alone cannot determine that a variable is an S3 object. First, we need to determine whether the variable is an object and then decide if it is S4 or not.
- In an object, we use a set of commands to get information about the data elements, or slots within an object. The first is the **slotNames** command which can take either an object or the name of a class. We obtain names of slots that are related to the class as strings.

- **How to create S4 objects?**
- S4 objects are created using the new() function.
- `s <- new("student", name="John", age=21, GPA=3.5)`
- `s`
- We can check if an object is an S4 object through the function isS4().
- `isS4(s)`
- `[1] TRUE`
- The function setClass() returns a generator function.
- This generator function (usually having same name as the class) can be used to create new objects. It acts as a constructor.
- `student <- setClass("student", slots=list(name="character", age="numeric", GPA="numeric"))`
- `student`

- **How to access and modify slot?**
- Just as components of a list are accessed using \$, slot of an object are accessed using @.
- **Accessing slot**
- `s@name`
- `[1] "John"`
- **Modifying slot directly**
- A slot can be modified through reassignment.
- `s@GPA <- 3.7`
- `s`
- **Modifying slots using slot() function**
- Similarly, slots can be access or modified using the slot() function.
- `slot(s,"name")`
- `[1] "John"`
- `slot(s,"name") <- "Paul"`

# Reference Class

- Reference class were introduced later, compared to the other two. It is more similar to the object oriented programming we are used to seeing in other major programming languages.
- Reference classes are basically S4 classed with an environment added to it.
- `setRefClass("student")`
- Member variables of a class, if defined, need to be included in the class definition. Member variables of reference class are called fields (analogous to slots in S4 classes).
- `setRefClass("student", fields = list(name = "character", age = "numeric", GPA = "numeric"))`



- **How to create a reference objects?**
- The function `setRefClass()` returns a generator function which is used to create objects of that class.
- `student <- setRefClass("student", fields = list(name = "character", age = "numeric", GPA = "numeric"))`
- `s <- student(name = "John", age = 21, GPA = 3.5) > s`
- **How to access and modify fields?**
- Fields of the object can be accessed using the `$` operator.
- `s$name [1] "John"`
- Similarly, it is modified by reassignment.
- `s$name <- "Paul"`

# Reference Methods

- Methods are defined for a reference class and do not belong to generic functions as in S3 and S4 classes.
- All reference class have some methods predefined because they all are inherited from the superclass `envRefClass`.
- We can create our own methods for the class.
- This can be done during the class definition by passing a list of function definitions to methods argument of `setRefClass()`.

## Comparison between S3 vs S4 vs Reference Class

| S3 Class                                           | S4 Class                                     | Referene Class                                 |
|----------------------------------------------------|----------------------------------------------|------------------------------------------------|
| Lacks formal definition                            | Class defined using <code>setClass()</code>  | Class defined using <code>setRefClass()</code> |
| Objects are created by setting the class attribute | Objects are created using <code>new()</code> | Objects are created using generator functions  |
| Attributes are accessed using <code>\$</code>      | Attributes are accessed using <code>@</code> | Attributes are accessed using <code>\$</code>  |
| Methods belong to generic function                 | Methods belong to generic function           | Methods belong to the class                    |
| Follows copy-on-modify semantics                   | Follows copy-on-modify semantics             | Does not follow copy-on-modify semantics       |

# Debugging

- A grammatically correct program may give us incorrect results due to logical errors. In case, if such errors (i.e. bugs) occur, we need to find out why and where they occur so that you can fix them. *The procedure to identify and fix bugs is called “**debugging**”.*
- There are a number of R debug functions, such as:
  - traceback()
  - debug()
  - browser()
  - trace()
  - recover()

# Fundamental Principles of R Debugging

1. The Essence of Debugging - The principle of confirmation: Fixing a bugging program is a process of confirming, one by one, that many things you believe to be true about code are actually true. When we find one of our assumptions is not true, we have found a clue to the location of a bug.
2. Start Small - Stick to small simple test cases, at least at the beginning of the R debug process. Working with large data objects may make it harder to think about the problem. Of course, we should eventually test our code in large, complicated cases, but start small.

### 3. Debug in a Modular-

- Top-Down Manner: Most professional software developers agree that code should be written in a modular manner. Our first-level code should not be long enough with much of it consisting of functions calls. And those functions should not be too lengthy and should call another function if necessary. This makes code easier at the writing stage and also for others to understand when the time comes for the code to be extended.
- We should debug in a top-down manner.

### 4. Antibugging - If we have a section of a code in which a variable $x$ should be positive, then we can insert this line: ***Stopifnot( $x > 0$ )***

- If there is a bug in the code earlier that renders  $x$  equals to, say  $-3$ , the call to *stopifnot()* will bring things right there, with an error message like this: ***Error:  $x > 0$  is not TRUE***

# R Debug Functions

- 1. `traceback()` - If our code has already crashed and we want to know where the offending line is, then try *traceback()*. This will (sometimes) show whereabouts in the code of the problem occurred.
- When an R function fails, an error is printed to the screen. Immediately after the error, you can call `traceback()` to see in which function the error occurred. The `traceback()` function prints the list of functions that were called before the error occurred. The functions are printed in reverse order.

2. `debug()`- The function *debug()* in R allows the user to step through the execution of a function, line by line. At any point, we can print out values of variables or produce a graph of the results within the function. While debugging, we can simply type “c” to continue to the end of the current section of code. *traceback()* does not tell us where the error occurred in the function. In order to know which line causes the error, we will have to step through the function using `debug()`.



3. `browser()` - The R debug function *browser()* stops the execution of a function until the user allows it to continue. This is useful if we don't want to step through the complete code, line-by-line, but we want it to stop at a certain point so we can check out what is going on. Inserting a call to the `browser()` in a function will pause the execution of a function at the point where the `browser()` is called. Similar to using `debug()` except we can control where execution gets paused.
4. `trace()` - Calling *trace()* on a function allows the user to insert bits of code into a function. The syntax for R debug function `trace()` is a bit strange for first-time users. It might be better off using `debug()`.

5. `recover()` - When we are debugging a function, *recover()* allows us to check variables in upper-level functions.
- By typing a number in the selection, we are navigated to the function on the call stack and positioned in the browser environment.
  - We can use `recover()` as an error handler, set using *options()* (e.g. `options(error=recover)`).
  - When a function throws an error, execution is halted at the point of failure. We can browse the function calls and examine the environment to find the source of the problem.
  - In `recover`, we use the previous `f()`, `g()` and `h()` functions for debugging.

# Error Handling & Recovery in R

- Exception or Error handling is a process of responding to anomalous occurrences in the code that disrupt the flow of the code. In general, the scope for the exception handlers begins with `try` and ends with a `catch`. R provides `try()` and `trycatch()` function for the same.
- The `try()` function is a wrapper function for `trycatch()` which prints the error and then continues. On the other hand, `trycatch()` gives you the control of the error function and also optionally, continues the process of the function.

# Unit-4

- **Functions for Reading Data into R:**
- Usually we will be using data already in a file that we need to read into R in order to work on it. R can read data from a variety of file formats—for example, files created as text, or in Excel.
- We will mainly be reading files in text format .txt or .csv (comma-separated, usually created in Excel).
- To read an entire data frame directly, the external file will normally have a special form -
  1. The first line of the file should have a *name* for each variable in the data frame.
  2. Each additional line of the file has as its first item a *row label* and the values for each variable.

1. CSV files – The csv file is a text file in which the values in the columns are separated by a comma. CSV stands for “comma-separated values”.
  - The file should be present in current working directory so that R can read it.
  - We can also set the directory from which files are to be read or written.

# get and print current working directory

`getwd()`

# Set current working directory

`setwd("D:/R programs")`

1. Reading from a csv file- we can read from a csv file using read.csv() function.

- We can check for the type, number of rows and columns in the data frame.
- Data <- read.csv("C:/Desktop/airquality.csv")
- The function **read.table()** can then be used to read the data frame directly.
- air <- read.table("C:/Desktop/airquality.txt")
- We can use the colnames() command to assign column names to the dataset.
- **Reading CSV Files with Pandas**
- result = pandas.read\_csv('X:\data.csv')

2. Reading from excel file- Microsoft Excel is the most widely used spreadsheet program which stores data in the .xls or .xlsx format.

- R can read directly from these files using some excel specific packages. Few such packages are – XLConnect, xlsx, gdata etc. We will be using “xlsx” package.
- `install.packages("xlsx")`
- `mydata = read.xlsx("D:/myexcel.xlsx", sheetindex = 1)`
- `mydata = read.xlsx ("D:/myexcel.xlsx", sheetName = "mysheet1")`

3. Reading data from json- the function `fromJSON()` will return a list.

- `Install.packages(rjson)`
- `File1 <- fromJSON("emp.json")`
- We can again convert the list into a dataframe using `as.data.frame()` function.
- `Json_data <- as.data.frame(File1)`

4. Reading data from XML file- The xml file is read by R and converted to data frames using the function `xmlToDataFrame()`.

- The result is stored as a list in R.
- `library("XML")`
- `Result <- xmlToDataFrame("XMLREAD.xml")`



5. Reading from binary files- we can read from binary files using the function `readBin()`.

- The syntax of `readBin()` function is `readBin(con,what,n)` where `con` is the connection object to read from the binary file, `what` is the mode like `char`, `int` etc. representing the bytes to be read and `n` is the number to bytes to read from the file.

# creating connection object

- `read.filename <- file("emp.dat", "rb")`

# reading column name

- `Col <- readBin(read.filename, character(), n=3)`

6. Reading data from HTML table-

- `readHTMLTable(doc, header = NA, colClasses = NULL, skip.rows = integer(), trim = TRUE, elFun = xmlValue, as.data.frame = TRUE,)`

- There are a few very useful functions for reading data into R.
- **read.table()** and **read.csv()** are two popular functions used for reading tabular data into R.
- **readLines()** is used for reading lines from a text file.
- **source()** is a very useful function for reading in R code files from a another R program.
- **dget()** function is also used for reading in R code files.
- **load()** function is used for reading in saved workspaces.
- **unserialize()** function is used for reading single R objects in binary format.

1. Writing to a CSV file- We can create csv file from existing data frame in R.
  - The `write.csv()` function is used to create the csv file.
  - `data <- read.csv("employee.csv")`
  - `hr <- subset( data, dept == "HR")`
  - `write.csv(hr, "hr.csv")`
  - The R base function **`write.table()`** can be used to export a data frame or a matrix to a file.
  - `write.table(x, file, append = FALSE, sep = " ", dec = ".", row.names = TRUE, col.names = TRUE)`

## 2. Writing to an excel file-

```
write.xlsx( x, file, sheetname = "Sheet1", col.names = TRUE, row.names = TRUE, append = FALSE)
```

- X is data frame to be written into workbook, file is the path to the output file.

## 3. Writing to binary file- we can write to binary files from R using the function writeBin().

- The syntax of writeBin is writeBin(object,con) where object is the binary file to be written and con is the connection object to write the binary file.
- emp <- read.csv("emp.csv")
- write.file <- file("emp.dat", "wb")
- writeBin(colnames(empdata), write.file)
- close(write.filename)

- **Functions for Writing Data to Files:**
- There are similar functions for writing data to files
- **write.table()** is used for writing tabular data to text files (i.e. CSV).
- **writeLines()** function is useful for writing character data line-by-line to a file or connection.
- **dump()** is a function for dumping a textual representation of multiple R objects.
- **dput()** function is used for outputting a textual representation of an R object.
- **save()** is useful for saving an arbitrary number of R objects in binary format to a file.
- **serialize()** is used for converting an R object into a binary format for outputting to a connection (or file).

# Handling large data sets in R

- **The Problem with large data sets in R-**
- R reads entire data set into RAM all at once. Other programs can read file sections on demand.
- R Objects live in memory entirely.
- Does not have int64 data type  
Not possible to index objects with huge numbers of rows & columns even in 64 bit systems (2 Billion vector index limit) . Hits file size limit around 2-4 GB.

- How big is a large data set:
- We can categorize large data sets in R across two broad categories:
- Medium sized files that can be loaded in R ( within memory limit but processing is cumbersome (typically in the 1-2 GB range )
- Large files that cannot be loaded in R due to R / OS limitations as discussed above . we can further split this group into 2 sub groups
  - Large files - (typically 2 - 10 GB) that can still be processed locally using some work around solutions.
  - Very Large files - ( > 10 GB) that needs distributed large scale computing.

- **Medium sized datasets (< 2 GB)**

1. Try to reduce the size of the file before loading it into R

- If you are loading xls files , you can select specific columns that is required for analysis instead of selecting the entire data set.
- You can not select specific columns if you are loading csv or text file - you might want to pre-process the data in command line using cut or awk commands and filter data required for analysis.

2. **Pre-allocate number of rows and pre-define column classes**

- Read optimization example :
- read in a few records of the input file , identify the classes of the input file and assign that column class to the input file while reading the entire data set
- calculate approximate row count of the data set based on the size of the file , number of fields in the column ( or using wc in command line ) and define nrow= parameter
- define comment.char parameter



- Alternately, use **fread** option from package data.table.
- “fast and friendly file finagler”, the popular data.table package is an extremely useful and easy to use. Its fread() function is meant to import data from *regular* delimited files directly into R, without any detours or nonsense.
- One of the great things about this function is that all controls, expressed in arguments such as sep, colClasses and nrows are automatically detected.
- Also, bit64::integer64 types are also detected and read directly without needing to read as character before converting.
- **ff** - ff is another package dealing with large data sets similar to bigmemory. It uses a pointer as well but to a flat binary file stored in the disk, and it can be shared across different sessions.
- One advantage ff has over bigmemory is that it supports multiple data class types in the data set unlike bigmemory.

- **Parallel Processing**-Parallelism approach runs several computations at the same time and takes advantage of multiple cores or CPUs on a single system or across systems. Following R packages are used for parallel processing in R.
- **Bigmemory** - bigmemory is part of the “big” family which consists of several packages that perform analysis on large data sets. bigmemory uses several matrix objects but we will only focus on big.matrix.
- big.matrix is a R object that uses a pointer to a C++ data structure. The location of the pointer to the C++ matrix can be saved to the disk or RAM and shared with other users in different sessions.
- By loading the pointer object, users can access the data set without reading the entire set into R.

- **Very Large datasets -**

- There are two options to process very large data sets ( > 10GB) in R.
- Use integrated environment packages like Rhipe to leverage Hadoop MapReduce framework.
- Use RHadoop directly on hadoop distributed system.
- Storing large files in databases and connecting through DBI/ODBC calls from R is also an option worth considering.
-

# Unit-5

- Regular Expressions- Regular Expressions (regex) are a set of pattern matching commands used to detect string sequences in a large text data. These commands are designed to match a family (alphanumeric, digits, words) of text which makes them versatile enough to handle any text / string class.
- In short, using regular expressions you can get more out of text data while writing shorter codes.

- **String Manipulation-** In R, we have packages such as stringr and stringi which are loaded with all string manipulation functions.
- In addition, R also comprises several base functions for string manipulations. These functions are designed to complement regular expressions.
- The practical differences between string manipulation functions and regular expressions are
- We use string manipulation functions to do simple tasks such as splitting a string, extracting the first three letters, etc. We use regular expressions to do more complicated tasks such as extract email IDs or date from a set of text.
- String manipulation functions are designed to respond in a certain way. They don't deviate from their natural behavior. Whereas, we can customize regular expressions in any way we want.

# List of String Manipulation Functions

| Functions                 | Description                                                                                                                                                                                                   |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>nchar()</code>      | It counts the number of characters in a string or vector. In the <code>stringr</code> package, it's substitute function is <code>str_length()</code>                                                          |
| <code>tolower()</code>    | It converts a string to the lower case. Alternatively, you can also use the <code>str_to_lower()</code> function                                                                                              |
| <code>toupper()</code>    | It converts a string to the upper case. Alternatively, you can also use the <code>str_to_upper()</code> function                                                                                              |
| <code>chartr()</code>     | It is used to replace each character in a string. Alternatively, you can use <code>str_replace()</code> function to replace a complete string                                                                 |
| <code>substr()</code>     | It is used to extract parts of a string. Start and end positions need to be specified. Alternatively, you can use the <code>str_sub()</code> function                                                         |
| <code>setdiff()</code>    | It is used to determine the difference between two vectors                                                                                                                                                    |
| <code>setequal()</code>   | It is used to check if the two vectors have the same string values                                                                                                                                            |
| <code>abbreviate()</code> | It is used to abbreviate strings. The length of abbreviated string needs to be specified                                                                                                                      |
| <code>strsplit()</code>   | It is used to split a string based on a criterion. It returns a list. Alternatively, you can use the <code>str_split()</code> function. This function lets you convert your list output to a character matrix |
| <code>sub()</code>        | It is used to find and replace the first match in a string                                                                                                                                                    |
| <code>gsub()</code>       | It is used to find and replace all the matches in a string / vector. Alternatively, you can use the <code>str_replace()</code> function                                                                       |

# List of Regular Expression Commands

- In regex, there are multiple ways of doing a certain task. Therefore, while learning, it's essential for you to stick to a particular method to avoid confusion.

| Function   | Description                                                                                           |
|------------|-------------------------------------------------------------------------------------------------------|
| grep       | returns the index or value of the matched string                                                      |
| grepl      | returns the Boolean value (True or False) of the matched string                                       |
| regexpr    | return the index of the first match                                                                   |
| gregexpr   | returns the index of all matches                                                                      |
| regexec    | is a hybrid of regexpr and gregexpr                                                                   |
| regmatches | returns the matched string at a specified index. It is used in conjunction with regexpr and gregexpr. |

- Regular expressions in R can be divided into 5 categories:
- Meta characters
- Sequences
- Quantifiers
- Character Classes
- POSIX character classes



**1. Meta characters** – Meta characters comprises a set of special operators which regex doesn't capture. These characters include: . \ | ( ) [ ] { } \$ \* + ?

- If any of these characters are available in a string, regex won't detect them unless they are prefixed with double backslash (\) in R.
- From a given vector, we want to detect the string "percent%." We'll use the base grep() function used to detect strings given a pattern. Also. we'll use the gsub() function to make the replacements.
- ```
dt <- c("percent%", "percent")
grep(pattern = "percent\\%", x = dt, value = T)
[1] "percent%"
```

## 2. **Quantifiers** - Quantifiers are the shortest to type, but these tiny atoms are immensely powerful.

- One position here and there can change the entire output value.
- Quantifiers are mainly used to determine the length of the resulting match.
- Always remember, that quantifiers exercise their power on items to the immediate left of it.
- Following is the list of quantifiers commonly used in detecting patterns in text: It matches everything except a newline.
- These quantifiers can be used with metacharacters, sequences, and character classes to return complex patterns. Combinations of these quantifiers help us match a pattern. The nature of these quantifiers is better known in two ways:
- **Greedy Quantifiers** : The symbol `.*` is known as a greedy quantifier. It says that for a particular pattern to be matched, it will try to match the pattern as many times as its repetition are available.
- **Non-Greedy Quantifiers** : The symbol `.?` is known as a non-greedy quantifier. Being non-greedy, for a particular pattern to be matched, it will stop at the first match.

| Quantifier | Description  |
|------------|--|
| .          | It matches everything except a newline.  |
| ?          | The item to its left is optional and is matched at most once.  |
| *          | The item to its left will be matched zero or more times.   |
| +          | The item to its left is matched one or more times.   |
| {n}        | The item to its left is matched exactly n times. The item must have a consecutive repetition at place. e.g. Anna |
| {n, }      | The item to its left is matched n or more times.   |
| {n,m}      | The item to its left is matched at least n times but not more than m times.                                      |

- Let's look at an example of greedy vs. non-greedy quantifier. From the given number, apart from the starting digit, we want to extract this number till the next digit '1' is detected. The desired result is 101.
- `number <- "101000000000100"`
- `#greedy`  
`regmatches(number, gregexpr(pattern = "1.*1",text = number))`  
`[1] "1010000000001"`
- `#non greedy`  
`regmatches(number, gregexpr(pattern = "1.?1",text = number))`  
`[1] "101"`
- It works like this: the greedy match starts from the first digit, moves ahead, and stumbles on the second '1' digit. Being greedy, it continues to search for '1' and stumbles on the third '1' in the number. Then, it continues to check further but couldn't find more. Hence, it returns the result as "1010000000001." On the other hand, the non-greedy quantifier, stops at the first match, thus returning "101."

- Let's look at a few more examples of quantifiers:
- `names <-  
c("anna","crissy","puerto","cristian","garcia","steven","alex",  
"rudy")`
- `#doesn't matter if e is a match  
grep(pattern = "e*",x = names,value = T)  
[1] "anna" "crissy" "puerto" "cristian" "garcia" "steven"  
"alex" "rudy"`
- `#must match t one or more times  
grep(pattern = "t+",x = names,value = T)  
[1] "puerto" "cristian" "steven"`
- `#must match n two times  
grep(pattern = "n{2}",x = names,value = T)  
[1] "anna"`

**3. Sequences** - As the name suggests, sequences contain special characters used to describe a pattern in a given string. Following are the commonly used sequences in R:

| Sequences       | Description                   |
|-----------------|-------------------------------|
| <code>\d</code> | matches a digit character     |
| <code>\D</code> | matches a non-digit character |
| <code>\s</code> | matches a space character     |
| <code>\S</code> | matches a non-space character |
| <code>\w</code> | matches a word character      |
| <code>\W</code> | matches a non-word character  |
| <code>\b</code> | matches a word boundary       |
| <code>\B</code> | matches a non-word boundary   |

- `gsub(pattern = "\\d", "_", "I'm working in RStudio v.0.99.484")`
- `[1] "I'm working in RStudio v._._.____"`
- # substitute any non-digit with an underscore
- `gsub(pattern = "\\D", "_", "I'm working in RStudio v.0.99.484")`
- `[1] "_____0_99_484"`
- # substitute any whitespace with underscore
- `gsub(pattern = "\\s", "_", "I'm working in RStudio v.0.99.484")`
- `[1] "I'm_working_in_RStudio_v.0.99.484"`
- # substitute any wording with underscore
- `gsub(pattern = "\\w", "_", "I'm working in RStudio v.0.99.484")`
- `[1] "_ _ _ _ _ _ . . . _ "`

- Let's look at some examples:
- `string <- "I have been to Paris 20 times"`
- #match a digit  
`gsub(pattern = "\\d+",replacement = " ",x = string)`  
`regmatches(string,regexpr(pattern = "\\d+",text = string))`
- #match a non-digit  
`gsub(pattern = "\\D+",replacement = " ",x = string)`  
`regmatches(string,regexpr(pattern = "\\D+",text = string))`
- #match a space - returns positions  
`gregexpr(pattern = "\\s+",text = string)`
- #match a non space  
`gsub(pattern = "\\S+",replacement = "app",x = string)`
- #match a word character  
`gsub(pattern = "\\w",replacement = "k",x = string)`
- #match a non-word character  
`gsub(pattern = "\\W",replacement = "k",x = string)`



## 4. Character Classes - Character classes refer to a set of characters enclosed in a square bracket [ ].

- These classes match only the characters enclosed in the bracket. These classes can also be used in conjunction with quantifiers.
- The use of the caret (^) symbol in character classes is interesting. It negates the expression and searches for everything except the specified pattern. Following are the types of character classes used in regex:

| Characters   | Description                       |
|--------------|-----------------------------------|
| [aeiou]      | matches lower case vowels         |
| [AEIOU]      | matches upper case vowels         |
| [0123456789] | matches any digit                 |
| [0-9]        | same as the previous class        |
| [a-z]        | match any lower case letter       |
| [A-Z]        | match any upper case letter       |
| [a-zA-Z0-9]  | match any of the above classes    |
| [^aeiou]     | matches everything except letters |
| [^0-9]       | matches everything except digits  |

- Let's look at some examples using character classes:
- `string <- "20 people got killed in the mob attack. 14 got severely injured"`
- `#extract numbers`  
`regmatches(x = string,gregexpr("[0-9]+",text = string))`
- `#extract without digits`  
`regmatches(x = string,gregexpr("[^0-9]+",text = string))`

**5. POSIX Character Classes** - In R, these classes can be identified as enclosed within a double square bracket (`[[ ]]`).

- They work like character classes. A caret ahead of an expression negates the expression value. Following are the posix character classes available in R:

| POSIX Characters | Description   |
|------------------|---|
| [:lower:]        | matches lower case letter   |
| [:upper:]        | matches upper case letter   |
| [:alpha:]        | matches letters   |
| [:digit:]        | matches digits  |
| [:space:]        | matches space characters eg. tab, newline, vertical tab, space, etc   |
| [:blank:]        | matches blank characters (same as previous) such as space, tab  |
| [:alnum:]        | matches alphanumeric characters, e.g. AB12, ID101, etc  |
| [:cntrl:]        | matches control characters. Control characters are non-printable characters such as \t (tab), \n (new line), \e (escape), \f (form feed), etc |
| [:punct:]        | matches punctuation characters  |
| [:xdigit:]       | matches hexadecimal digits (0 - 9 A - E)  |
| [:print:]        | matches printable characters ([:alpha:] [:punct:] and space)  |
| [:graph:]        | matches graphical characters. Graphical characters comprise [:alpha:] and [:punct:]   |

- `x <- "I like beer! #beer, @wheres_my_beer, I like R (v3.2.2) #rrrrrrrr2015"`
- `# remove space or tabs`
- `gsub(pattern = "[[:blank:]]", replacement = "", x)`
- `[1]`  
`"Ilikebeer!#beer,@wheres_my_beer,IlikeR(v3.2.2)#rrrrrrrr2015"`
- `# replace punctuation with whitespace`
- `gsub(pattern = "[[:punct:]]", replacement = " ", x)`
- `[1]` `"I like beer beer wheres my beer I like R v3 2 2 rrrrrrr2015"`
- `# remove alphanumeric characters`
- `gsub(pattern = "[[:alnum:]]", replacement = "", x)`
- `[1]` `" ! #, @__, (..) #"`

- Let's look at some of the examples of this regex class:
- `string <- c("I sleep 16 hours\n, a day","I sleep 8 hours\n a day.", "You sleep how many\t hours ?")`
- `#get digits`  
`unlist(regmatches(string,gregexpr("[[:digit:]]+",text = string)))`
- `#remove punctuations`  
`gsub(pattern = "[[:punct:]]+",replacement = "",x = string)`
- `#remove spaces`  
`gsub(pattern = "[[:blank:]]",replacement = "-",x = string)`
- `#remove control characters`  
`gsub(pattern = "[[:cntrl:]]+",replacement = " ",x = string)`
- `#remove non graphical characters`  
`gsub(pattern = "[^[:graph:]]+",replacement = "",x = string)`

# 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.

- The basic syntax to create a bar-chart in R is –
- **barplot(H,xlab,ylab,main, names.arg,col)** Following is the description of the parameters used –
- **H** is a vector or matrix containing numeric values used in bar chart.
- **xlab** is the label for x axis.
- **ylab** is the label for y axis.
- **main** is the title of the bar chart.
- **names.arg** is a vector of names appearing under each bar.
- **col** is used to give colors to the bars in the graph.
- Example- `H <- c(7,12,28,3,41)`
- `barplot(H)`



# Bar Chart Labels, Title and Colors

- The features of the bar chart can be expanded by adding more parameters.
- The **main** parameter is used to add **title**.  
The **col** parameter is used to add colors to the bars.
- The **args.name** is a vector having same number of values as the input vector to describe the meaning of each bar.
- `H <- c(7,12,28,3,41)`
- `M <- c("Mar","Apr","May","Jun","Jul")`
- `barplot(H,names.arg=M,xlab="Month",ylab="Revenue",col="blue", main="Revenue chart",border="red")`

# Group Bar Chart and Stacked Bar Chart

- We can create bar chart with groups of bars and stacks in each bar by using a matrix as input values.
- More than two variables are represented as a matrix which is used to create the group bar chart and stacked bar chart.
- `colors = c("green","orange","brown")`
- `months <- c("Mar","Apr","May","Jun","Jul")`
- `regions <- c("East","West","North")`
- `Values <- matrix(c(2,9,3,11,9,4,8,7,3,12,5,2,8,10,11), nrow = 3, ncol = 5, byrow = TRUE)`
- `barplot(Values, main = "total revenue", names.arg = months, xlab = "month", ylab = "revenue", col = colors)`
- `legend("topleft", regions, cex = 1.3, fill = colors)`

# Histogram

- A histogram represents the frequencies of values of a variable bucketed into ranges. Histogram is similar to bar chart 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.

- The basic syntax for creating a histogram using R is –
- `hist(v,main,xlab,xlim,ylim,breaks,col,border)`
- **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.
- A simple histogram is created using input vector, label, col and border parameters.
- `v <- c(9,13,21,8,36,22,12,41,31,33,19)`
- `hist(v,xlab = "Weight",col = "yellow",border = "blue")`

# Box Plot

- 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.

- 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.
- We use the data set "mtcars" available in the R environment to create a basic boxplot. Let's look at the columns "mpg" and "cyl" in mtcars.
- `input <- mtcars[,c('mpg','cyl')]`
- `boxplot(mpg ~ cyl, data = mtcars, xlab = "Number of Cylinders", ylab = "Miles Per Gallon", main = "Mileage Data")`

- **Boxplot with Notch –**

- We can draw boxplot with notch to find out how the medians of different data groups match with each other.
- The below script will create a boxplot graph with notch for each of the data group.
- `boxplot(mpg ~ cyl, data = mtcars,  
xlab = "Number of Cylinders",  
ylab = "Miles Per Gallon",  
main = "Mileage Data",  
notch = TRUE, varwidth = TRUE,  
col = c("green","yellow","purple"),  
names = c("High","Medium","Low"))`

# Scatter Plot

- Scatterplots show many points plotted in the Cartesian plane. Each point represents the values of two variables. One variable is chosen in the horizontal axis and another in the vertical axis.
- The simple scatterplot is created using the **plot()** function.
- **Scatterplot Matrices** - When we have more than two variables and we want to find the correlation between one variable versus the remaining ones we use scatterplot matrix. We use **pairs()** function to create matrices of scatterplots.
- `pairs(formula, data)`
- `pairs(~wt+mpg+disp+cyl,data = mtcars, main = "Scatterplot Matrix")`



- Syntax of Scatterplot
- `plot(x, y, main, xlab, ylab, xlim, ylim, axes)`
- **x** is the data set whose values are the horizontal coordinates.
- **y** is the data set whose values are the vertical coordinates.
- **main** is the title of the graph.
- **xlab** is the label in the horizontal axis.
- **ylab** is the label in the vertical axis.
- **xlim** is the limits of the values of x used for plotting.
- **ylim** is the limits of the values of y used for plotting.
- **axes** indicates whether both axes should be drawn on the plot.
- `input <- mtcars[,c('wt','mpg')]`
- `plot(x = input$wt, y = input$mpg, xlab = "Weight", ylab = "Milage", xlim = c(2.5,5), ylim = c(15,30), main = "Weight vs Milage" )`

- **3D Scatterplot-** You can create a 3D scatterplot with the scatterplot3d package. Use the function **scatterplot3d(x, y, z)**.
- library(scatterplot3d)
- attach(mtcars)
- scatterplot3d(wt,disp,mpg, main="3D Scatterplot")
- # 3D Scatterplot with Coloring and Vertical Drop Lines
- library(scatterplot3d)
- attach(mtcars)
- scatterplot3d(wt,disp,mpg, pch=16, highlight.3d=TRUE, type="h", main="3D Scatterplot")

# Strip Chart

- Strip charts can be created using the `stripchart()` function in R programming language.
- This function takes in a numeric vector or a list of numeric vectors, drawing a strip chart for each vector.
- Let us use the built-in dataset `airquality` which has “Daily air quality measurements.”
- `stripchart(airquality$Ozone)`
- **Multiple Strip Charts-** `stripchart(x, main="Multiple stripchart for comparision",  
xlab="Degree Fahrenheit",  
ylab="Temperature",  
method="jitter",  
col=c("orange","red"), pch=16 )`

# Dot plot

- Create dotplots with the **dotchart(x, labels=)** function, where *x* is a numeric vector and **labels** is a vector of labels for each point. You can add a **groups=** option to designate a factor specifying how the elements of *x* are grouped. If so, the option **gcolor=** controls the color of the groups label. **cex** controls the size of the labels.
- `Dotchart(mtcars$mpg, labels=row.names(mtcars), cex=.7, main="Gas Milage for Car Models", xlab="Miles Per Gallon")`

# Density Plots

- Kernel density plots are usually a much more effective way to view the distribution of a variable.
- It is created using **plot(density(x))** where *x* is a numeric vector.
- `d <- density(mtcars$mpg)`
- `plot(d)`

# Line Graph

- 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.

- `plot(v,type,col,xlab,ylab)` Following is the description of the parameters used –
- **v** is a vector containing the numeric values.
- **type** takes the value "p" to draw only the points, "l" to draw only the lines and "o" to draw both points and lines.
- **xlab** is the label for x axis.
- **ylab** is the label for y axis.
- **main** is the Title of the chart.
- **col** is used to give colors to both the points and lines.
- `v <- c(7,12,28,3,41)`
- `plot(v,type = "o")`

# Pie 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.
- `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.
- `x <- c(21, 62, 10, 53)`
- `labels <- c("London", "New York", "Singapore", "Mumbai")`
- `pie(x,labels)`



- **Slice Percentages and Chart Legend –**
- `x <- c(21, 62, 10, 53)`
- `labels <- c("London", "NewYork", "Singapore", "Mumbai")`
- `piepercent <- round(100*x/sum(x), 1)`
- `png(file = "city_percentage_legends.jpg")`
- `pie(x, labels = piepercent, main = "City pie  
chart", col = rainbow(length(x)))`  
`legend("topright", c("London", "New  
York", "Singapore", "Mumbai"), cex = 0.8, fill =  
rainbow(length(x)))`

- **3D Pie Chart –**

- A pie chart with 3 dimensions can be drawn using additional packages. The package **plotrix** has a function called **pie3D()** that is used for this.
- `x <- c(21, 62, 10,53)`
- `lbl <- c("London", "New York", "Singapore", "Mumbai")`
- `png(file = "3d_pie_chart.jpg")`
- `pie3D(x,labels = lbl,explode = 0.1, main = "Pie Chart of Countries ")`