

# Install R and R Studio

## Getting Started With R

# Install R and R Studio

- Installing R on Windows platform
- Installing R on Mac & Linux
- R -studio Installation

# Installing R on Mac OS or Linux is similar to Windows.

R CRAN:

The Comprehensive R Archive Network - A network of global web servers storing identical & up-to-date versions of code and documentation for R.

It is the main repository for R packages. Currently, it features 14054 available packages.

Steps to Download and Install R :

Go to <http://cran.r-project.org/>

Select 'Download R for Windows'.

# Installing R on Windows Platform

- Click on 'Download R [latest version]'
- In our case it is R 4.2.3

A screenshot of a web browser window. The address bar shows 'cran.r-project.org/bin/windows/base/'. The page title is 'R-4.2.3 for Windows'. Below the title, there are three blue links: 'Download R-4.2.3 for Windows (77 megabytes, 64 bit)', 'README on the Windows binary distribution', and 'New features in this version'. The browser interface includes standard navigation buttons (back, forward, refresh), a search bar, and various extension icons.

[Download R-4.2.3 for Windows \(77 megabytes, 64 bit\)](#)

[README on the Windows binary distribution](#)

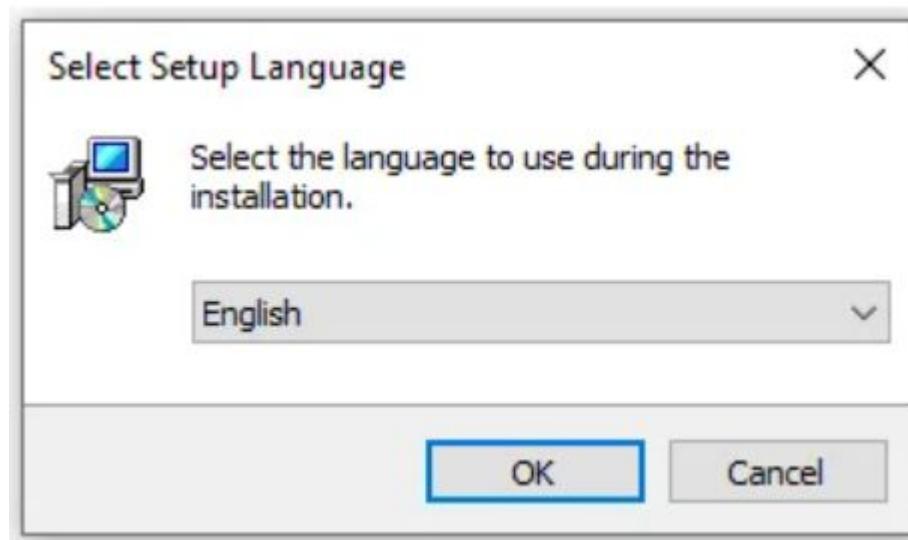
[New features in this version](#)

This build requires UCRT, which is part of Windows since Windows 10 and Windows Server 2016. On older systems, UCRT has to be installed manually from [here](#).

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.

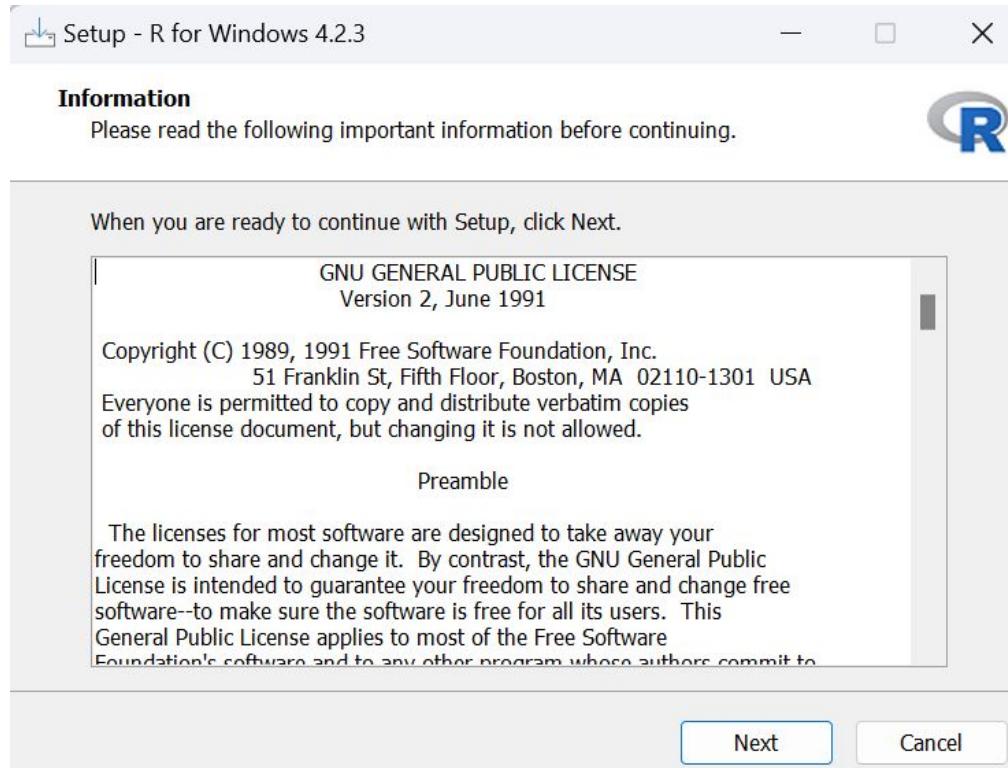
To install R on Windows OS:

1. Go to the [CRAN](#) website.
2. Click on "**Download R for Windows**".
3. Click on "**install R for the first time**" link to download the R executable (.exe) file.
4. Run the R executable file to start installation, and allow the app to make changes to your device.
5. Select the installation language.



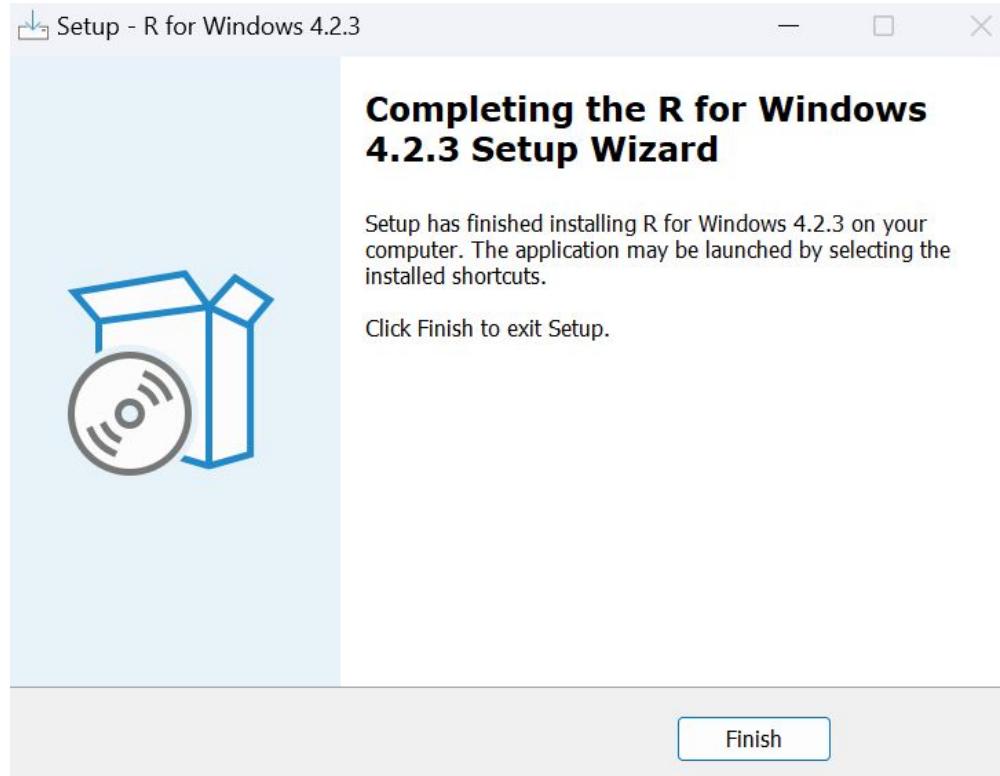
# Installing R on Windows Platform

## 6. Follow the installation instructions



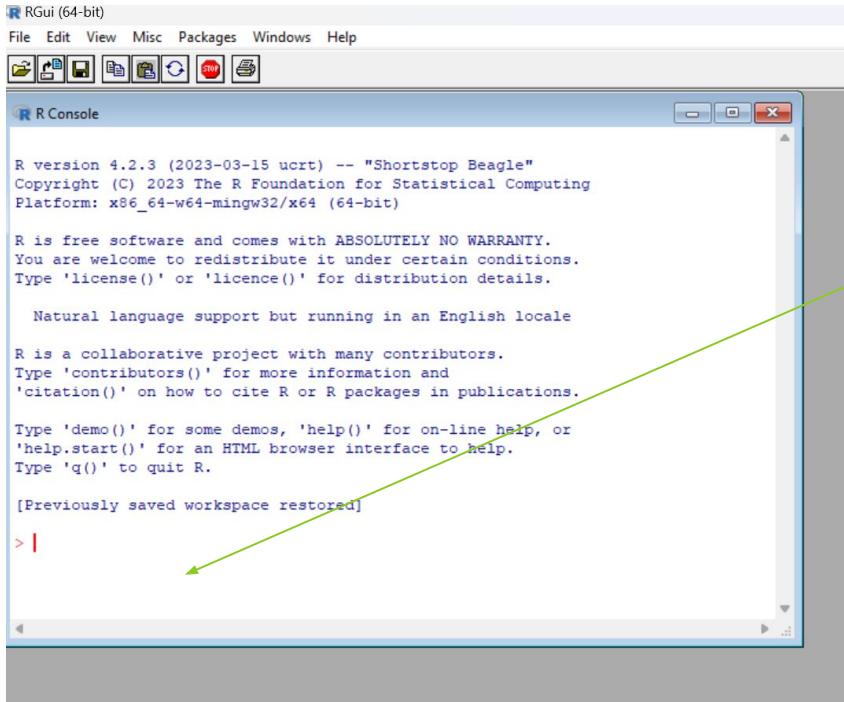
# Installing R on Windows Platform

7. Click on "**Finish**" to exit the installation setup



# Installing R on Windows Platform

- After it is installed, you should now have a R menu choice. Start the program by choosing either R i386 4.2.3 (32-bit) or R x64 4.2.3 (64-bit)
- This starts the UI



## The R console

- Type in the command
- Press Enter
- See the output

# Installing R on Mac or Linux Platform

Installing R on Mac OS or Linus is similar to Windows.

Steps to Download and Install R :

- Go to <http://cran.r-project.org/>
- Select 'Download R for Linux' or 'Download R for (Mac) OS X.'
- The next step is to click on the "R-4.2.3 pkg" (or newer version) file to begin the installation. You can leave the default options as is just like for Windows.

## R for macOS

This directory contains binaries for a base distribution and packages to run on macOS. Releases for old Mac OS X systems (through Mac OS X 10.5) and PowerPC Macs can be found in the [old](#) directory.

Note: Although we take precautions when assembling binaries, please use the normal precautions with downloaded executables.

Package binaries for R versions older than 3.2.0 are only available from the [CRAN archive](#) so users of such versions should adjust the CRAN mirror setting (<https://cran-archive.r-project.org>) accordingly.

R 4.2.3 "Shortstop Beagle" released on 2023/03/15

Please check the integrity of the downloaded package by checking the signature:

`pkutil --check-signature R-4.2.3.pkg`

in the *Terminal* application. If Apple tools are not available you can check the SHA1 checksum of the downloaded image:

`openssl sha1 R-4.2.3.pkg`

### Latest release:

[R-4.2.3-arm64.pkg](#) (notarized and signed)

SHA1 hash: 99d1ad04b0a6765d40cd019540ffe722f77b6b81  
(ca. 86MB) for M1 and higher Macs only!

**R 4.2.3** binary for macOS 11 (**Big Sur**) and higher, **Apple silicon arm64** build, signed and notarized package.

Contains R 4.2.3 framework, R.app GUI 1.79 for Apple silicon Macs (M1 and higher), Tcl/Tk 8.6.12 X11 libraries and Texinfo 6.8.

**Important: this version does NOT work on older Intel-based Macs** - see below for Intel version.

macOS Ventura users: there is a known bug in Ventura, if the installation fails, move the downloaded file away from the *Downloads* folder (e.g., to your home or Desktop)

# Installing R on Mac or Linux Platform

- RStudio is a free and open source integrated development environment (IDE) for R
- It is more user-friendly than using R directly since it keeps track of your script file, console, plots, and history, all in one place.
- It has an organized layout and several extra options.
- The usual RStudio screen has four windows:
  - Console
  - Workspace and history
  - Files, plots, packages and help
  - The R script(s) and data view.

To download RStudio go to <https://www.rstudio.com/> and follow the steps.

# RStudio - Installation

- To download RStudio go to
- <https://www.rstudio.com/products/rstudio/download/>
- Select Download of RStudio Desktop FREE Version

Choose Your Version of RStudio

RStudio is a set of integrated tools designed to help you be more productive with R. It includes a console, syntax-highlighting editor that supports direct code execution, and a variety of robust tools for plotting, viewing history, debugging and managing your workspace. [Learn More about RStudio features.](#)

Version	License	Cost	Action
RStudio Desktop	Open Source License	FREE	<a href="#">DOWNLOAD</a>
RStudio Desktop	Commercial License	\$995 per year	<a href="#">BUY</a>
RStudio Server	Open Source License	FREE	<a href="#">DOWNLOAD</a>
RStudio Server Pro	Commercial License	\$9,995 per year	<a href="#">DOWNLOAD</a>
RStudio Server Pro + RStudio Connect	Commercial License	\$29,995 per year	<a href="#">TALK</a>

# RStudio - Installation

- Choose and download installer as per your machine's platform
- Run the installer and follow instructions
- In our case, RStudio latest version - 1.1.447

The screenshot shows the RStudio website at <https://www.rstudio.com/products/rstudio/download/#download>. The page title is "RStudio Desktop 1.1.447 — Release Notes". The main content area starts with a note for Linux users: "Linux users may need to import RStudio's public code-signing key prior to installation, depending on the operating system's security policy." This note is circled in red. Below this, there is a section titled "Installers for Supported Platforms" which lists various installer files with their sizes, dates, and MD5 checksums.

Installers	Size	Date	MD5
RStudio 1.1.447 - Windows Vista/7/8/10	95.8 MB	2018-04-18	359df07f279db25c99d0f91449b0fc33
RStudio 1.1.447 - Mac OS X 10.6+ (64-bit)	74.5 MB	2018-04-18	13d679b7ec208bd31f22550c0dd6c99
RStudio 1.1.447 - Ubuntu 12.04-15.10/Debian 8 (32-bit)	89.3 MB	2018-04-18	8186c1b10793a6ff3138cd16af3ea433
RStudio 1.1.447 - Ubuntu 12.04-15.10/Debian 8 (64-bit)	97.4 MB	2018-04-18	a9d410a0c74eb68d4c34bd880fd103a6
RStudio 1.1.447 - Ubuntu 16.04+/Debian 9+ (64-bit)	64.5 MB	2018-04-18	1eecbf2c80fb329a8a1bc92a68e459e
RStudio 1.1.447 - Fedora 19+/Red Hat 7+/openSUSE 13.1+ (32-bit)	88.1 MB	2018-04-18	aac10abbcc8315ddd3bf2b25ce8d9914
RStudio 1.1.447 - Fedora 19+/Red Hat 7+/openSUSE 13.1+ (64-bit)	90.8 MB	2018-04-18	0dd507d4ece283a64b63f759dc9e4fad

# Getting Started With R

# Contents

- What is R?
- Why Learn R?
- History of R
- Users of R
- R Environment
- R community
- IDE (Integrated Development Environment )
- R Studio

# Why Learn R ?

Free and  
Open  
Source

A language where the original source code is freely available and can be modified.

Connects  
with other  
languages

Languages like C , C++, Java, Python, Fortran can be called from within R

Vast  
Engaged  
Community

R has a large community of more than 2 million users

Supports  
Extensions

Data structures in R consist of vectors, scalars, data frames, lists, etc

# Why Learn R ?

Flexible

It is very easy to write code enhancements, develop packages, develop apps , write your own functions and distribute your own software

Extremely  
Com-  
prehensive

R consists of over 18,000 CRAN, Bioconductor and GitHub packages

Great Data  
visualization

Varied plots such as boxplots, histograms, barplots, etc are available and are high in quality and self-explanatory.

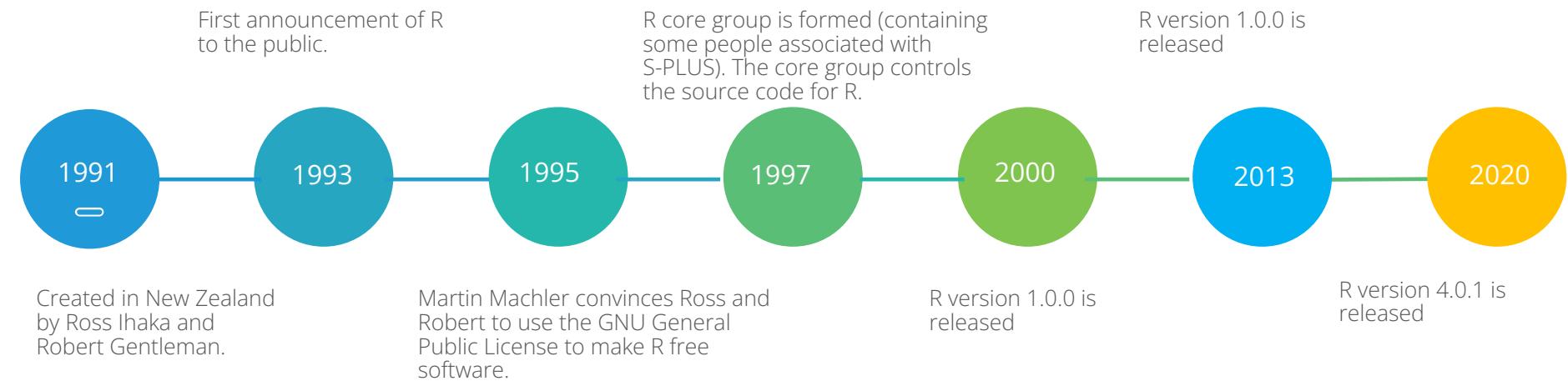
Advanced  
Statistical  
Language

Overall structure and syntax of R is exclusively developed for statistical computation

Cross  
Platform  
Compatible

You can run R on several O.S like Windows, Macintosh, GNU/Linux, UNIX and varied Software/Hardware

# History of R



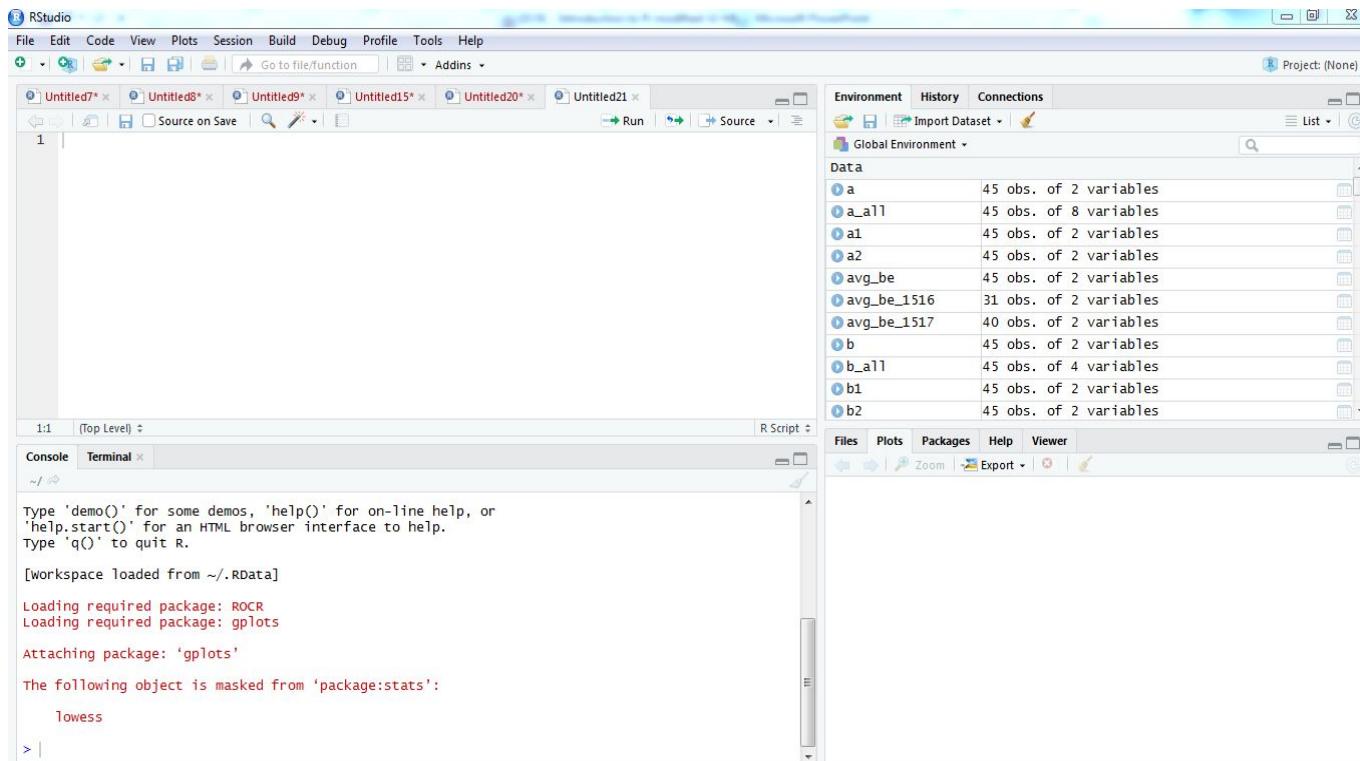
## Users of R

- R has more than 2 million Users and thousands of developers around the world as reported in 2014.
- R is widely used by:
  - Academicians and Researchers.
  - Banks
  - Regulators like FDA (Food and Drug Administration)
  - Social Media giants like Facebook and Twitter, Google, Mozilla, New York Times, Thomas Cook, Uber

## R Environment

- R Environment is a collection of objects like functions, variables.
- R can be extended via packages. Currently there are more than 18,000 packages available on CRAN, Bioconductor and GitHub
- All datasets created in the session remain in Memory
- Output can be used as input to other functions
- R commands are Case Sensitive.

# R Environment



## R Community

- R Community is a group of developers who maintain R and guide its evolution.
- Few of the contributors are: R-bloggers, Revolution Analytics Blog, R-statistics and Data Mining Blog. There are news, blogs, forums, research on new packages or advancements, on their websites. They enable you to connect with other users and you can also contribute to these communities to both help others and learn the material and strengthen your own understanding.
- Useful & most referred Links :
  1. Stack Overflow - <https://stackoverflow.com/>
  2. R Bloggers - <https://www.r-bloggers.com/>
  3. Revolution Analytics Blog - <https://blog.revolutionanalytics.com/>
  4. R Statistics - <https://www.r-statistics.com/>

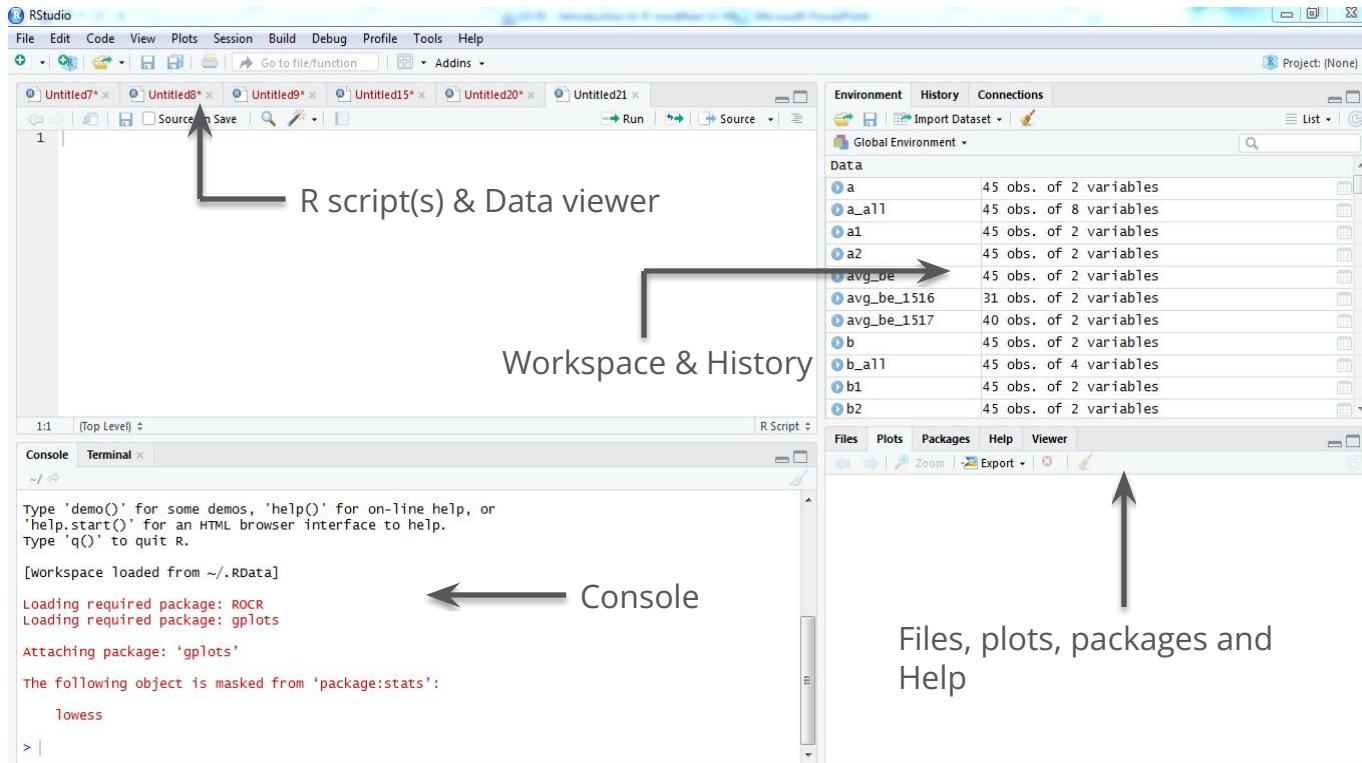
## IDE

- Integrated Development Environment (IDE) is a software which provides programmer with an interface combined with all the required tools at hand.
- There are many IDE's available for R. Few of them are RStudio , eclipse with StatET.
- Most recommended and widely used among these is RStudio. Microsoft R Open, which is the enhanced distribution of R from Microsoft Corporation, can be used with Rstudio and together they make a powerful combination. It is Open source & free to download, use and share.

# RStudio

- RStudio is an Integrated development environment (IDE) for R.
- The usual RStudio screen has four windows:
  - Console
  - Workspace and history
  - Files, plots, packages and help
  - The R script(s) and data view.
- RStudio is available in open source and commercial editions and runs on the desktop (Windows, Mac, and Linux) or in a browser connected to RStudio Server or RStudio Server Pro. RStudio supports authoring HTML, PDF, Word Documents, and slide shows.
- RStudio supports interactive graphics. RStudio make it easy to start new or find existing projects.

# RStudio



# Quick Recap

- In this session, we had an introduction about R. Here is the quick recap:



R

- It is a programming language and environment for statistical computing and graphics
- It is free and open source software
- Created by Ross Ihaka and Robert Gentleman
- It is built using packages, which contains Basic and advanced functions.
- R Community is a group of developers who maintain R and guide its evolution



RStudio

- RStudio is most widely used user friendly IDE.

# Packages in R



# Contents

1. R Package
2. Installing a Package from CRAN
3. How to Use a Package
4. Some Useful Commands
5. Some Useful R Packages

# R Package

- A Package is a collection of R functions with comprehensive documents.



- A core set of packages is included with the installation of R. Additional packages are required for specific requirements.
- These packages can be downloaded from various repositories like CRAN, or from bioconductor, GitHub, R Forge and many more.
- Packages extend the functionality of R by enabling additional visual capabilities, statistical methods, and discipline-specific functions, just to name a few.

# Installing a Package from CRAN?

To install a package



Type the following command in R console:

**`install.packages("pkg")`**

- `pkg` is the name of package whose current version should be downloaded from the repositories.
- For example :

**`install.packages("dplyr")`**

# How to Use a Package?

To use a package



Invoke the following command to load the package into the ongoing session: **library(<pkg>)**

Example :

```
library(dplyr)
```

# Some useful Commands

- Removing Packages:

**`remove.packages("pkg")`**

Removes a specific package

- Updating Packages:

**`update.packages()`**

Updates all installed packages

- Installed Packages:

**`installed.packages()`**

Returns a list of all installed packages

- Finding Available Packages:

**`available.packages()`**

Returns a list of all available packages

# Some Useful R Packages

For loading data	<ul style="list-style-type: none"><li>• readr, data.table, foreign, Hmisc</li></ul>
For data management	<ul style="list-style-type: none"><li>• data.table, dplyr, reshape2, tidyr</li></ul>
For data visualization	<ul style="list-style-type: none"><li>• ggplot2</li></ul>
For predictive modeling	<ul style="list-style-type: none"><li>• car, caret, e1071, party, ROCR</li></ul>
For Time Series	<ul style="list-style-type: none"><li>• forecast, zoo</li></ul>

# Quick Recap

In this session, we learnt the concept of packages and how to make use of it. Here is a quick recap:

R commands  
to work with  
packages

- `install.packages("pkg")`
- `library(<pkg>)`
- `remove.packages("pkg")`
- `available.packages()`
- `update.packages()`

Useful R Packages

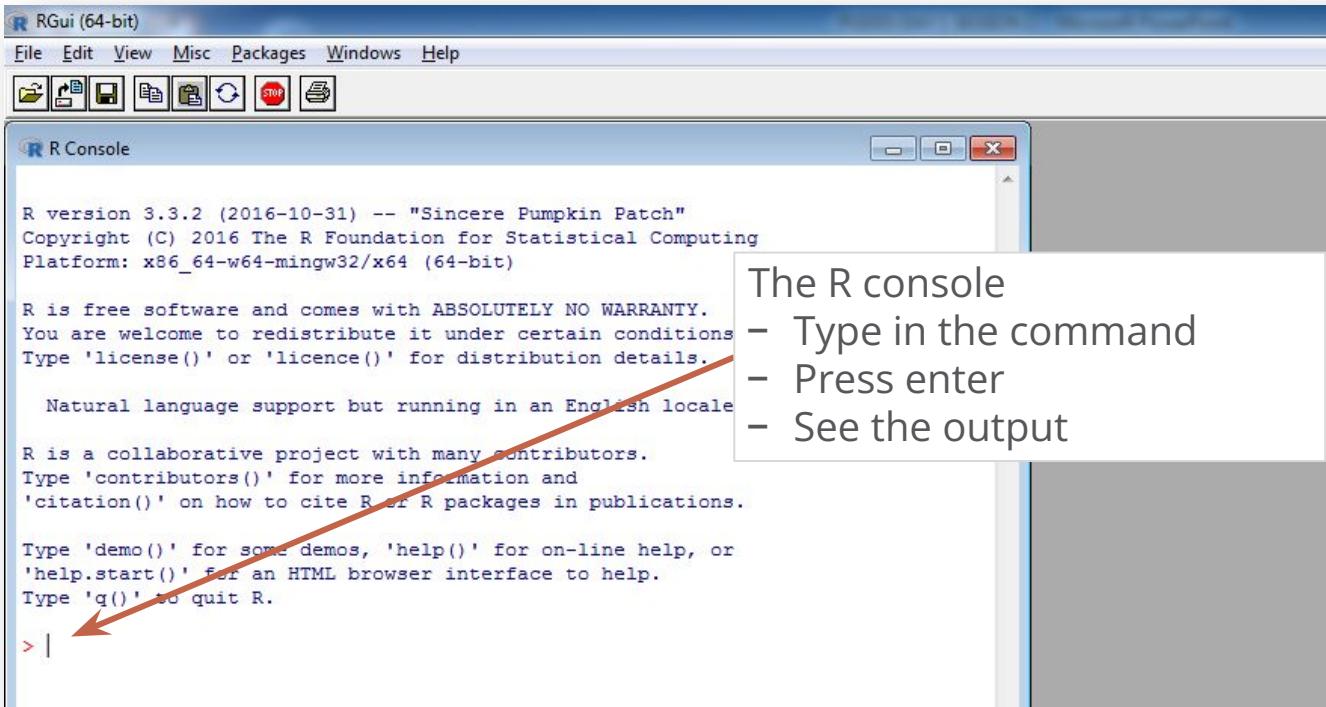
- `readr`, `data.table`, `foreign`, `Hmisc`,  
`dplyr`, `reshape2`, `tidyR`, `ggplot`,  
`car`, `caret`, `e1071`, `party`, `ROCR`  
`forecast` and `zoo`

# Writing your first Program in R

# Contents

- The R-UI
- Create and Save Your Script
- Write a Program
- R Workspace
- Help in R
- The Working Directory

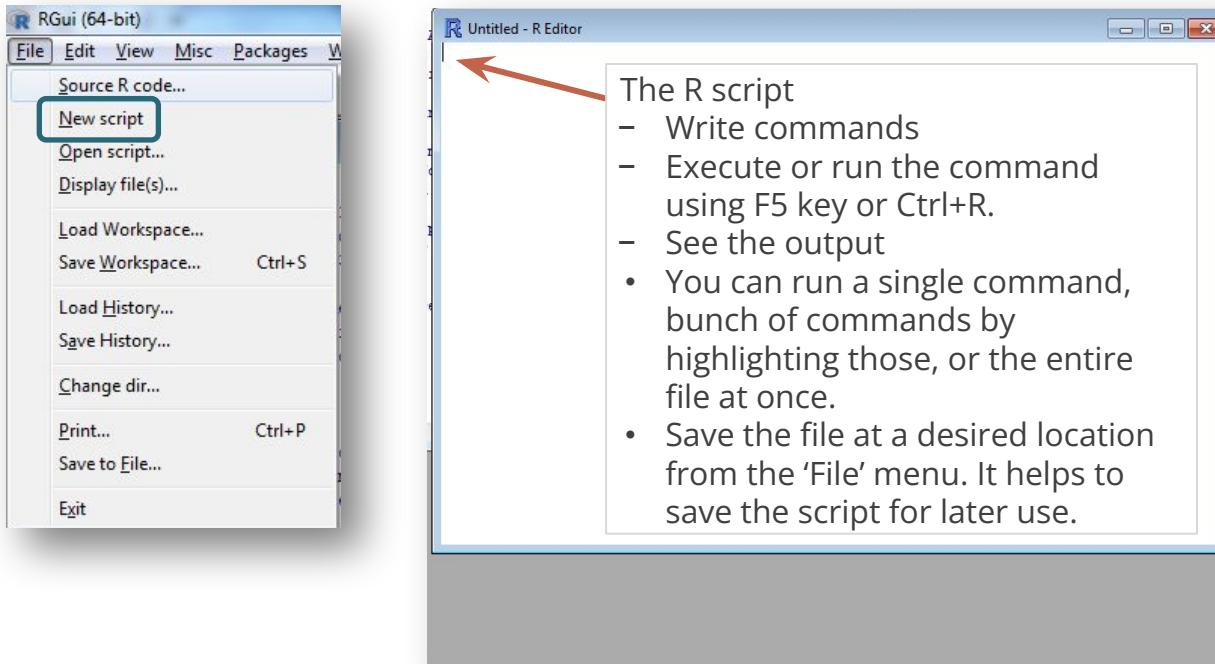
# The R-UI



# Create and Save Your Script

In your R console,

- Open a new script. **File > New script**



# Write a Program

Let's begin with writing our First program !

Create two objects 'x' and 'y' and perform mathematical operations on them.

```
# Create an object x and assign a value 10
```

```
x<-10
```

- The operators '<-' and '=' are used for assigning values to an object.

```
# Type x and press Enter key to see the value of the object
```

```
x
```

```
[1] 10
```



Note : Comments can be added to keep a record of what you did and why, by preceding all comments with the # symbol.

# Write a Program

```
# Create another object y and assign a value 7
```

```
y<-7  
y  
[1] 7
```

```
# Add the values of the objects x and y
```

```
x + y  
[1] 17
```

```
# Multiply x by y
```

```
x * y  
[1] 70
```



R is case sensitive i.e. it treats 'a' and 'A' as completely different objects.

# Write a Program

```
# To check which objects are already used
```

```
ls()
```

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

- Use up and down arrow keys to scroll through your command history.

# R Workspace

Objects that you create during an R session are held in memory, the collection of objects that you currently have is called the workspace

- This workspace is not saved on disk unless you tell R to do so.
- This means that your objects are lost when you close R and not save the objects, or worse when R or your system crashes on you during a session.
- If you have saved a workspace image and you start R the next time, it will restore the workspace.
- So all your previously saved objects are available again.
- You can also explicitly load a saved workspace i.e., that could be the workspace image of someone else. File > Load Workspace

# R Workspace

Some basic commands for using the command history

```
#To display all previous commands  
history()
```

```
#To display last 25 commands  
history(max.show=25)
```

```
#To save your command history to a file. Default is ".Rhistory"  
savehistory(file="myfile")  
#myfile is saved in My Documents
```

```
#To recall your command history. Default is ".Rhistory"  
loadhistory(file="myfile")
```

# Help in R

- R has a comprehensive built-in help system consisting of a documentation for packages and functions. Any of the following commands can be used to access help in R:
- If you know the topic but not the exact function:

```
help.search("regression")
```

OR

```
??"regression"
```

- If you know the exact function:

```
help(functionname)  
help(mean)
```

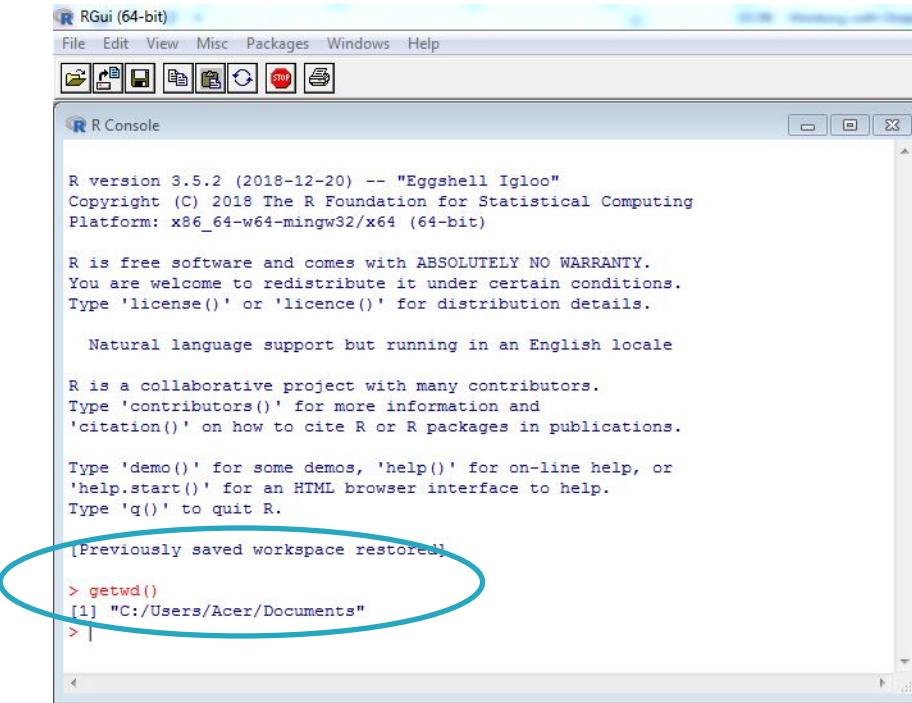
OR

```
?functionname  
?mean
```

# The Working Directory

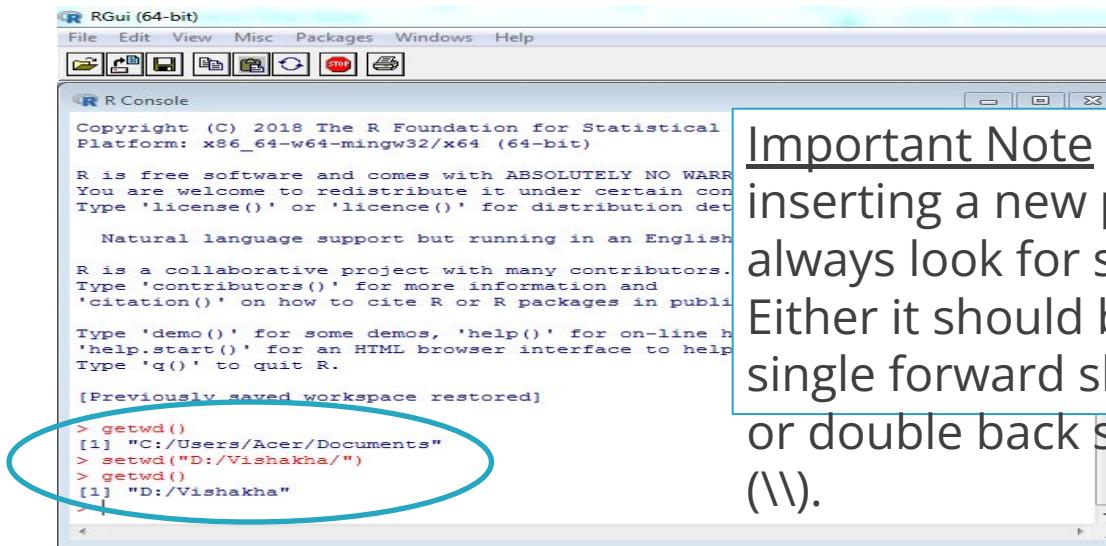
- One of the initial things that you want to do when you launch R for the first time is to set its working directory.
- A working directory is a default location on your hard disk where R looks without further instruction for any files you ask it to retrieve or write. It is important to select a location that is easy to find and remember, so you can access your files when you need them.
- In order to check what your R current working directory is, type command `getwd()`. `getwd()` returns an absolute file path representing the current working directory of the R process.
- If the directory it returns is not the directory you would like to use, then simply use the following function to set your working directory: `setwd('PATH')` replace **PATH** with the directory path that you would like to use
- Use `getwd()` again to verify that the change took place.

# The Working Directory



# The Working Directory

- If the directory it returns is not the directory you would like to use, then simply use the following function to set your working directory: `setwd('PATH')` replace **PATH** with the directory path that you would like to use
- Use `getwd()` again to verify that the change took place.



```
R Gui (64-bit)
File Edit View Misc Packages Windows Help
R Console
Copyright (C) 2018 The R Foundation for Statistical
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
environment.

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,
'help.start()' for an HTML browser interface to help
Type 'q()' to quit R.

[Previously saved workspace restored]

> getwd()
[1] "C:/Users/Acer/Documents"
> setwd("D:/Vishakha/")
> getwd()
[1] "D:/Vishakha"
>
```

Important Note : While inserting a new path, always look for slashes. Either it should be a single forward slash (/) or double back slash (\\).

# The Working Directory

- Steps to set working directory through R tools:
  - Go to the File menu
  - Select Change Working Directory
  - Select the appropriate folder/directory
- In case of Windows, R's default working directory is:  
"C:/Users/User Name/Documents"



You have the option to set the working directory at any time. Do this when you want to access files in a new location, such as when you are working on multiple projects at the same time or at the start of a new project.

# Quick Recap

In this session, we learnt how to work create and save scripts, create objects, see their output and perform calculations on them, manage R's workspace and get help in R:

## Write a program

- To assign values to an object: Use operators '<-' and '='.
- Press Enter & see the value of the object
- Add comments for reference using # symbol

## R Workspace

- Save workspace to save all objects created during the session so that next time you start R from the same working directory, the saved objects will be available for use
- Basic history commands: `history()`, `savehistory()`, `loadhistory()`

## Help in R

- If you know the topic but not the exact function:  
`help.search("regression")` or `??"regression"`
- If you know the exact function:  
`help(functionname)` or `?functionname`

## Working with Directories & scripts

- `getwd()` and `setwd()`: To check and set working directory respectively
- File > New script: To open a new script. Save the script from the 'File' menu. It helps to save the script for later use.

# Data Types in R

# Data Types in R

Numeric	Contains decimal as well as whole numbers
Integer	Contains only whole numbers
Character	Holds character strings
Factor	A vector that can contain only predefined values, and is used to store categorical data.
Logical	Can only take on two values, TRUE or FALSE.

# Data Types in R

Vector	1 dimension	Sequence of data elements of the same basic type.
Matrix	2 dimensions	Like a Vector but additionally contains the dimension attribute.
Array	2 or more dimensions	Hold multidimensional data. Matrices are a special case of two-dimensional arrays.
Data frame	2 dimensions	Table-like data object allowing different data types for different columns.
List		Collection of data objects, each element of a list is a data object.

# Numeric

Numeric type object can store decimal as well as whole numbers.

Create two numeric objects x and y and assign values 4.5 and 3567 respectively.

```
x<-4.5  
x  
[1] 4.5  
  
y<-3567  
y  
[1] 3567  
  
class(x)  
[1] "numeric"  
  
class(y)  
[1] "numeric"
```

class() function is used  
to check the data type  
of an object.

# Numeric

R shows class of object x as numeric. To convert it to an integer format, use `as.integer()` function.

```
x<-as.integer(x)
```

```
x
```

```
[1] 4
```

```
class(x)
```

```
[1] "integer"
```

So, new value of x is 4 after converting it into integer

# Integer

#To create an integer variable in R use `as.integer()` function.

```
f<-as.integer(22.5)  
f  
[1] 22
```

```
class(f)
```

```
[1] "integer"
```

```
x=8  
class(x)  
[1] "numeric"
```

**Note :** The default class of an integer is a numeric class

# Character

A character object is used to represent strings. A string is specified by using quotes (both single and double quotes will work). To convert any object into a character type, use `as.character()` function.

```
z<-"Welcome to R Ready Reckon-er"  
z  
[1] "Welcome to R Ready Reckon-er"
```

```
x<-"4.5"  
x  
[1] "4.5"  
class(z)
```

```
[1] "character"  
class(x)  
[1] "character"
```

Any value enclosed in quotes is stored as a character object

# Factor

Factor objects are used to categorize data and can store both strings and integers.

```
# Create an object x
```

```
x<-c("high", "medium", "low", "low", "medium", "high", "high", "high",
"medium", "low","low")
```

c() combines data of different types

```
# Check whether object x is a factor or character
```

```
is.factor(x) ←
```

```
[1] FALSE
```

is.factor() function returns True or False after checking whether the object is of type factor or not

```
is.character(x) ←
```

```
[1] TRUE
```

is.character() function returns True or False after checking whether the object is of type character or not

# Factor

```
#Create a factor object using factor() function
```

```
x<-factor(x)
```

```
x
```

```
[1] high   medium low    low    medium high   high   high  
[9] medium low    low  
Levels: high low medium
```

- A factor is a categorical variable that can take only one of a fixed, finite set of possibilities. Those possible categories are the levels.
- **Levels** are the unique data values. Above output tells how many levels are there in x.
- Alternatively, one can use **level()** function to check levels.

```
levels(x)
```

```
[1] "high" "low" "medium"
```

Factor object x has 11 elements and 3 levels. By default the levels are sorted alphabetically

# Factor

#To specify the order of the factor, use **ordered()** function

```
x_ordered<-ordered(x, levels=c("low", "medium","high"))  
x_ordered  
[1] high   medium low    low    medium high   high   high    medium low  
low  
Levels: low < medium < high
```

levels= takes the levels in the order in which you want them to be ordered



One of the most important uses of factors is in statistical modeling, since categorical variables enter into statistical models differently than continuous variables, storing data as factors insures that the modeling functions will treat such data correctly.

# Logical

Logical type objects can only take 2 values i.e. TRUE or FALSE.

```
# Create an object x and assign a value 4.5 and check whether it is an integer
```

```
x<-4.5
```

```
is.integer(x)
```

```
[1] FALSE
```

is.integer() function checks whether the object is integer or not.

```
# Create two numeric objects y and z  
# Check whether y is greater than z or not
```

```
y<-4
```

```
z<-7
```

```
Result <- y > z
```

```
Result
```

```
[1] FALSE
```

- With this kind of statement, you are asking R to evaluate the logical question “Is it true that y is greater than z?”
- The object(Result) storing the answer of above question is of type logical
- You can check the class of the object using class()

# Vector

A Vector is a Sequence of data elements of the same basic type.

Create three vectors: numeric, character and logical. All the elements of a vector must be of the same type.

```
# Numeric vector
```

```
a <- c(1,2,5.3,6,-2,4)  
a
```

```
[1] 1.0 2.0 5.3 6.0 -2.0 4.0
```

```
# Character vector
```

```
b <- c("one","two","three")  
b
```

```
[1] "one"    "two"    "three"
```

```
# Logical vector
```

```
d<-c(4,24,6,4, 2,7)  
d>5
```

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

# Matrices

Matrix is a two-dimensional data structure in R. It is similar to vectors but additionally contains the dimension attribute. To convert any object into a matrix type, use `as.matrix()` function.

Create a matrix with 3 rows and 2 columns.

```
x<-matrix(c(2, 3, 4, 5, 6, 7),nrow=3,ncol=2)  
x
```

```
 [,1] [,2]  
[1,] 2 5  
[2,] 3 6  
[3,] 4 7
```

- matrix() function is used to create a matrix.
- nrow= and ncol= is used to specify the dimension of the matrix

Note that the matrix is filled in by column-wise.

```
x<-matrix(c(2, 3, 4, 5, 6, 7),nrow=3,ncol=2,byrow=TRUE)  
x
```

```
 [,1] [,2]  
[1,] 2 3  
[2,] 4 5  
[3,] 6 7
```

byrow=TRUE fills the matrix row-wise

# Matrices

It is possible to name the rows and columns of matrix during creation by passing a 2-element list to the argument `dimnames`.

```
x<-matrix(c(2, 3, 4, 5, 6, 7), nrow=3, ncol=2, byrow=TRUE,  
           dimnames=list(c("X","Y","Z"), c("A","B")))
```

```
x  
      A B  
X 2 3  
Y 4 5  
Z 6 7
```

#Dimension names can be accessed or changed with two helpful functions `colnames()` and `rownames()`:

```
colnames(x)  
[1] "A" "B"  
rownames(x)  
[1] "X" "Y" "Z"  
colnames(x) <- c("a", "b")  
colnames(x)  
[1] "a" "b"
```

rownames can be changed in similar manner

# Matrices

- Another way of creating a matrix is by using functions **cbind()** and **rbind()** as in column bind and row bind.

```
# Alternative command for creating matrix column-wise and row-wise
```

```
cbind(c(2,3,4),c(5,6,7))
```

```
rbind(c(2,3),c(4,5), c(6,7))
```

# Arrays

- An array holds multidimensional rectangular data i.e. each row is the same length, and likewise for each column and other dimensions

#Create an array with four columns, three rows and two “tables”.

```
a<-array(1:24,dim=c(3,4,2))  
a
```

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

```
[1,]   13    16    19    22  
[2,]   14    17    20    23  
[3,]   15    18    21    24
```

array(data, dim = c(r,c,t) )

r = no. of rows

c = no. of columns

t = no. of tables

Note: Although the rows are given as the first dimension, the tables are filled column-wise. So, for arrays, R fills the columns, then the rows, and then the rest

# Data Frames

Data frame is a two-dimensional array like structure. It is a list of vectors of equal length. Data frames are the primary data structure in R. To convert any object into a data frame type, use `as.data.frame()` function.

Create a data frame from different vectors

```
x<-c(12,23,45)  
y<-c(13,21,6)  
z<-c("a","b","c")
```

creating vectors x, y, z

```
data<-data.frame(x,y,z)  
data
```

	x	y	z
1	12	13	a
2	23	21	b
3	45	6	c

- `data.frame()` function combines them in a table.
- object `data` is a dataframe containing three vectors x, y, z



Data.frames are distinct from matrices because they can include heterogeneous data types among columns/variables.

# Data Frames

Let's have a look at the structure of our data frame.

```
str(data) ←  
'data.frame': 3 obs. of 3 variables:  
$ x: num 12 23 45  
$ y: num 13 21 6  
$ z: Factor w/ 3 levels "a","b","c": 1 2 3
```

- str() shows the structure of an object
- Note : z is a character vector but by default R stores it in the data frame as factor..



By default, R always transforms character vectors to factors when creating a data frame with character vectors or converting a character matrix to a data frame. This can cause errors if you are unaware of it. To avoid this, you can specify **stringsAsFactors=FALSE** while creating a dataframe

# Lists

List is a data structure having components of mixed data types. To convert any object into a list type, use **as.list()** function.

Create a list of 2 vectors and a numeric value

```
n=c(2, 3, 5)
s=c("aa", "bb", "cc", "dd", "ee")
x=list(n, s, 3)
```

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

```
[[2]]
[1] "aa" "bb" "cc" "dd" "ee"
```

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

list() is used to create lists,

# Lists

We retrieve a list slice by enclosing the index of the vector single in a square bracket "[]" operator.

```
x[2]←
```

With an index vector, we can retrieve a slice with multiple members.

```
[[1]]
```

```
[1] "aa" "bb" "cc" "dd" "ee"
```

```
x[c(2, 3)]
```

```
[[1]]
```

```
[1] "aa" "bb" "cc" "dd" "ee"
```

```
[[2]]
```

```
[1] 3
```

# Quick Recap

1. Numeric
2. Integer
3. Character
4. Factor
5. Logical
6. Vector
7. Matrices
8. Array
9. Data Frame
10. List

Data Types

`as.integer()`,  
`as.matrix()`,  
`as.data.frame()`,  
`as.list()`,  
`as.logical()`, and  
`as.vector()`.

Type Conversion Functions

# Numeric Functions and Operators in R

# Contents

1. Introduction to Functions
2. General Functions
3. Introduction to Operators
4. Assignment Operators
5. Arithmetic Operators
6. Relational Operators
7. Logical Operators
8. Miscellaneous Operators

# Functions

In R, the operations that do all the work are called functions. R has a large number of in-built functions and the user can create their own functions. Most functions are in the following form:

`f(argument1, argument2,...)`

Where `f` is the name of the function, and `argument1, argument2, ...` are the arguments to the function.

In this tutorial we will be discussing General and Statistical Built-in Functions of R.

# General Functions

Note: 'x' here is a numeral or a vector of numerals.

```
#Absolute value of 'x'  
abs(-4)  
[1] 4  
abs(c(-4,4.5,-10.5,6))  
[1] 4.0 4.5 10.5 6.0
```

```
#Square Root of 'x'  
sqrt(81)  
[1] 9
```

```
#Rounds to the nearest integer that's larger than x  
ceiling(445.67)  
[1] 446
```

```
#Rounds to the nearest integer that's smaller than x  
floor(445.67)  
[1] 445
```

# General Functions

Note: 'x' here is a numeral or a vector of numerals.

```
#Rounds to the nearest integer toward 0.  
trunc(445.67)  
[1] 445  
trunc(c(445.67,19.567,33.09))  
[1] 445 19 33
```

```
#Rounds to the nearest possible value after mentioning how many digits  
#to keep after decimal point.  
round(44.5682,digits=2)  
[1] 44.57
```

```
#specify the number of significant digits to be retained
```

```
signif(44.5681,digits=4)  
[1] 44.57
```

Similar to round() function, signif() also specify number of significant digits regardless of the size of the number.

# General Functions

Note: 'x' here is a numeral or a vector of numerals.

```
#Computes natural logarithms.
```

```
log(50)
```

```
[1] 3.912023
```

```
log(c(44,55))
```

```
[1] 3.784190 4.007333
```

```
#Computes binary (base 2) logarithm.
```

```
log2(8)
```

```
[1] 3
```

```
#Computes logarithm to the base 10.
```

```
log10(55)
```

```
[1] 1.740363
```

```
#Computes the exponential value , ex.
```

```
exp(6)
```

```
[1] 403.4288
```

# Operators

An operator is a symbol which helps the user to command the compiler to perform specific mathematical or logical operations. R language is rich in built-in operators and are classified into the following categories:

- Assignment Operators
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Miscellaneous Operators

# Assignment Operators

These operators are used to assign values to objects.

```
#Left Assignment (<-, <<-, =) can be done in 3 ways:
```

```
x1<-45
```

```
x1=45
```

```
x1<<-45
```

```
x1
```

```
[1] 45
```

```
#Right Assignment (->, ->>) can be done in 2 ways
```

```
45->y1
```

```
45->>y1
```

```
y1
```

```
[1] 45
```

Different assignments  
all leading to same  
output

# Arithmetic Operators

These operators are used to perform mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R.

```
x<-7  
y<-18
```

```
#Addition  
x+y  
[1] 25
```

```
#Subtraction  
x-y  
[1] -11
```

```
#Multiplication  
x*y  
[1] 126
```

# Arithmetic Operators

```
#Division
```

```
y/x
```

```
[1] 2.571429
```

```
#Exponentiation
```

```
y^x
```

```
[1] 612220032
```

```
#Integer Division to get Remainder
```

```
y%%x
```

```
[1] 4
```

```
#Integer Division to get Quotient
```

```
y%/%x
```

```
[1] 2
```

# Relational Operators

These operators are used to compare values. The result of the comparison is the Boolean (True or False) value. Following table shows the relational operators available in R.

```
x<-7  
y<-18
```

```
#Less than
```

```
x<y  
[1] TRUE
```

```
#Greater than
```

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

```
#Less than or equal to
```

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

# Relational Operators

```
#Greater than or equal to
```

```
y>=20
```

```
[1] FALSE
```

```
#Equal to
```

```
y==16
```

```
[1] FALSE
```

```
#Not equal to
```

```
x!=5
```

```
[1] TRUE
```

# Logical Operators

Logical Operators are applicable only to vectors of logical or numeric type. They compare each element of the first vector with the corresponding element of the second vector. Below table describes the logical operators available in R with different expressions & their respective outcomes :

Expression	Outcome	Expression	Outcome
true AND true	TRUE	true OR true	TRUE
true AND false	FALSE	true OR false	TRUE
false AND false	FALSE	false OR false	FALSE
true AND missing	missing	true OR missing	TRUE
missing AND missing	missing	missing OR missing	missing
false AND missing	FALSE	false OR missing	missing

0 means False & any number >0 is True

# Logical Operators

Below table describes the logical operators available in R.

```
x<-c(FALSE,TRUE,2,5)
y<-c(TRUE,FALSE,FALSE,TRUE)
```

```
#Logical NOT
!x
[1] TRUE FALSE FALSE FALSE
```

```
#Element-wise logical AND
x&y
[1] FALSE FALSE FALSE TRUE
```

# Logical Operators

`&&` and `||` examines only the first element of the vector resulting into single length logical vector.

```
#Logical AND  
x&y  
[1] FALSE
```

```
#Element-wise logical OR  
x|y  
[1] TRUE TRUE TRUE TRUE
```

```
#Logical OR  
x||y  
[1] TRUE
```

# Miscellaneous Operators

These operators are used for specific purpose and not general mathematical or logical operations.

```
#Colon operator.
```

```
x<-1:5  
x  
[1] 1 2 3 4 5
```

It creates simple integer sequences

```
x<-10  
t<-1:8  
x%in%t  
[1] FALSE
```

This operator is used to identify if a value belongs to a vector or array

# Quick Recap

In this session, we learnt different types of Functions and Operators in R. Here is a quick recap:

## General Functions

- `abs()`, `sqrt()` `ceiling()`, `floor()`, `trunc()`,  
`signif()`, `log()`, `log2()`, `log10()`, `exp()`

## Operators

- Assignment Operators: `<-`, `<<-`, `=`, `->`, `->>`
- Arithmetic Operators: `+`, `-`, `*`, `/`, `^`, `%%`, `%%`
- Relational Operators: `<`, `>`, `<=`, `>=`, `==`, `!=`
- Logical Operators: `!`, `&`, `&&`, `|`, `||`
- Miscellaneous Operators: `%in%`, `:`

# String Functions in R

1. Introduction to Strings
2. Understanding String Manipulation Functions
  - paste()
  - format()
  - substring()
  - sub() & gsub()
  - strsplit()
  - abbreviate()
  - tolower()
  - toupper()
  - casefold()
  - chartr()

# Introduction to Strings

Any value written within a pair of single quote or double quotes in R is treated as a String.

'A character string using single quotes'

"A character string using double quotes"

Internally R stores every string within double quotes, even when you create them with single quote.

We can insert single quotes in a string with double quotes, and vice versa:

"Welcome to 'Ask Analytics'"

'Welcome to "Ask Analytics"'

We cannot insert single quotes in a string with single quotes, neither we can insert double quotes in a string with double quotes:

"Welcome to "Ask" Analytics"

'Welcome to 'Ask' Analytics'

# Concatenating Strings - paste()

**paste()** is a very versatile function and is perhaps one of the most important functions that we can use to create and build strings.

**paste()** converts its arguments to character strings and concatenates (pastes) them to form one or several character strings.

```
a<- "Hello"  
b<- 'How'  
c<- "are you? "  
  
paste(a,b,c)  
[1] "Hello How are you?"
```

The default separator is a blank space  
(sep=" ")

```
paste(a,b,c,sep="-")  
[1] "Hello-How-are you?"
```

sep= takes a character string that is used as a separator. In this case it's "-".

```
paste("y",1:4,sep=".")  
[1] "y.1" "y.2" "y.3" "y.4"
```

single character "y" is pasted with the sequence 1:4 and separator sep=".."

# Concatenating Strings - `paste()`

If we want all the arguments to be collapsed into a single string, use the `collapse=` argument

```
# paste with collapsing  
paste("y",1:4,sep=".")  
[1] "y.1|y.2|y.3|y.4"
```

There is another function `paste0()` which is similar to `paste()`. The difference between them is that the argument `sep` by default is space (" ") in `paste()` and "" in `paste0()`.

```
# paste0 function  
paste0(a,b,c)  
[1] "HelloHoware you? "
```



`paste0()` is faster than `paste()` if our objective is concatenate strings without spaces because we don't have to specify the argument `sep`.

# Formatting Strings - format()

When you produce reports in R, you will want it to appear all nicely formatted to enhance the impact of your data and text on the viewer.

**format()** function formats numbers and strings to a specified style. It treats the elements of a vector as character strings using a common format. This is especially useful when printing numbers and quantities under different formats.

```
# default usage  
format(45.4)  
[1] "45.4"
```

Notice that the result is no longer a number but a character string.

This function takes a number of arguments to control the format of your result. We will see the use of some of the useful arguments like: nsmall, digits, width, justify

# Formatting Strings - format()

```
format(19.7,nsmall=3)
```

```
[1] "19.700"
```

nsmall= is used to specify the minimum number of digits after the decimal point.

```
format(13.567,digits=4)
```

```
[1] "13.57"
```

digits= is used to specify how many significant digits of numeric values to show

```
format(c("I","am","fine"),width=5)
```

```
[1] "I      " "am     " "fine   "
```

Here for each string the spaces are filled to make Length 5 for each string  
For e.g. "I "

width= is used to specify the minimum width of string  
By default, format() fills the strings with spaces so that they are all of the same length.

```
format(c("I","am","fine"),width=5,justify="centre")
```

```
[1] "  I  " " am  " "fine "
```

justify= controls how the string is displayed.Takes the values "left", "right", "centre", "none"

# Formatting Strings - format()

```
# Use of digits, widths and justify arguments  
format(format(10/1:5,digits=2),width=6,justify="centre")  
[1] " 10.0 " " 5.0 " " 3.3 " " 2.5 " " 2.0 "
```

For printing large numerical sequences we can use the arguments **big.mark**. For example, here is how we can print a number with sequences separated by a comma ","

```
format(145604500,big.mark=",")  
[1] "145,604,500"
```

# Extracting and Replacing - substring()

One common task when working with strings is the extraction and replacement of some characters. For such tasks we have the function **substring()** which extracts or replaces substrings in a character vector.

**substring(text, first, last)**



text is a character vector  
first indicates the first element to be replaced  
last indicates the last element to be replaced

```
# Extract 'THE'
```

```
substring("WELCOME TO THE WORLD OF R",12,14)
```

```
[1] "THE"
```



Characters from 12<sup>th</sup> to 14<sup>th</sup> position are extracted

# Extracting and Replacing - substring()

```
# Extract first 3 letters
```

```
Location<-c("Mumbai","Delhi","Mumbai","Kolkata","Delhi")  
substring(Location,1,3)
```

```
[1] "Mum" "Del" "Mum" "Kol" "Del"
```

```
# Extract each letter
```

```
substring("GOOD EVENING",1:12,1:12)
```

```
[1] "G" "O" "O" "D" " " "E" "V" "E" "N" "I" "N" "G"
```

```
# Replacing ':' with ' '
```

```
string=c("G1:E001","G2:E002","G3:E003")  
substring(string,3)<- " "  
string
```

```
[1] "G1 E001" "G2 E002" "G3 E003"
```

# Get an Edge!

- **substr()** function, similar to **substring()** is used to extract and replace substrings in a character vector.
  - Difference is **substr()** takes one start value and one end value, and returns one string. **substring()** takes n start values and the corresponding n end values, returning n strings, such that each string begins at  $n[i]$  and ends at  $n[i]$ , where i is in 1:n.
- The **substring()** function is a little more robust than **substr()**.

# Replacing Substrings – sub() & gsub()

Within a string, if you want to replace one substring with another:

- Use **sub()** to replace the first instance of a substring

**sub(old,new,string)**

- Use **gsub()** to replace all instances of a substring. The g in **gsub()** stands for global.

**gsub(old,new,string)**

```
# Using sub()
```

```
string<- "She is a data scientist. She works with an MNC"  
sub("She", "Sharon", string)
```

```
[1] "Sharon is a data scientist. She works with an MNC"
```

sub() finds the first instance of the 'She' within string and replaces it with 'Sharon'

# Replacing Substrings - sub() & gsub()

```
# Replace a string with its first 3 letters
```

```
Location<-c("Mumbai","Delhi","Mumbai","Kolkata","Delhi")
```

```
m<-sub("Mumbai","Mum",Location)
```

```
d<-sub("Delhi","Del",m)
```

```
k<-sub("Kolkata","Kol",d)
```

```
k
```

```
[1] "Mum" "Del" "Mum" "Kol" "Del"
```

- ❑ Here we are replacing 'Mumbai' with 'Mum', 'Delhi' with 'Del' and 'Kolkata' with 'Kol'.
- ❑ Location is a vector of strings.
- ❑ sub() will find the first instance of the old substring in each string of Location and replaces it with the new substring.

```
# Using gsub()
```

```
gsub("She","Sharon",string)
```

```
[1] "Sharon is a data scientist. Sharon works with an MNC"
```

gsub() does the same thing as sub(), but it replaces all instances of the substring(a global replace), not just the first

# Splitting a String - strsplit()

Besides the tasks of extracting and replacing substrings, another common task is splitting a string based on a pattern. In R we have a function **strsplit()** which splits the elements of a character vector into substrings.

If we want to break a string into individual components (.i.e. words), we can use **strsplit()**.

For example:

```
sentence<-c("break a string into individual components")  
strsplit(sentence, " ")
```

“ ” is a regular expression pattern used for splitting

```
[[1]]  
[1] "break"      "a"           "string"      "into"       "individual"  
[6] "components"
```

# Splitting a String - strsplit()

```
# split each date component
```

```
dates<-c("12-10-2014","01-05-2000","26-06-2015")  
strsplit(dates, "-")
```

```
[[1]]  
[1] "12"    "10"    "2014"
```

Here, date components joined by a dash "-", are split

```
[[2]]  
[1] "01"    "05"    "2000"
```

```
[[3]]  
[1] "26"    "06"    "2015"
```

# Abbreviate Strings - abbreviate()

Another useful function for basic manipulation of character strings is **abbreviate()**.

It abbreviate strings to at least minimum length characters, such that they remain unique.

```
states<-c("New York","ALABAMA","ARKANSAS","ARIZONA")
abbreviate(states,minlength=3)
```

New York	ALABAMA	ARKANSAS	ARIZONA
"NwY"	"ALA"	"ARK"	"ARI"

minlength= is used to specify the  
minimum length of the  
abbreviation

# Other Basic String Manipulation Functions

```
#Converts any upper case characters into lower case  
tolower(c("ASK ANALytics","data SCience"))  
[1] "ask analytics"  
[2] "data science"
```

```
#Converts any lower case characters into upper case  
toupper(c("ASK ANALytics","data SCience"))  
[1] "ASK ANALYTICS"  
[2] "DATA SCIENCE"
```

```
#Case-folding function which is a wrapper for both tolower() and toupper().
```

```
casefold("all characters in LOWER case")  
[1] "all characters in lower case"
```

By default, it converts all characters to lower case, but argument `upper = TRUE` can be included to indicate the opposite (characters in upper case)

```
#chartr function which replaces a character
```

```
chartr("a","A","This is a string")  
[1] "This is A string"
```

`chartr(old, new, x)`- translates each character in x that is specified in old to the corresponding character specified in new.

# Quick Recap

In this session, we learnt how to manipulate strings with functions in base R.

Here is the quick recap:

## String Manipulation Functions

- **paste()** converts its arguments to character strings and concatenates (pastes) them
- **format()** formats numbers and strings to a specified style
- **substring()** extracts or replaces substrings in a character vector.
- **sub()** replaces the first instance of a substring
- **gsub()** replaces all instances of a substring
- **strsplit()** which splits the elements of a character vector into substrings
- **abbreviate()** abbreviate strings to at least minimum length characters, such that they remain unique
- **toupper()** Converts any upper case characters into lower case
- **tolower()** Converts any lower case characters into upper case
- **casefold()** wrapper for both tolower() and toupper().
- **chartr()** stands for character translation.

# Functions

In R, the operations that do all the work are called functions. R has a large number of in-built functions and the user can create their own functions. Most functions are in the following form:

`f(argument1, argument2,...)`

Where f is the name of the function, and argument1, argument2, ... are the arguments to the function.

In this tutorial we will be discussing General and Statistical Built-in Functions of R.

# General Functions

Note: 'x' here is a numeral or a vector of numerals.

```
#Absolute value of 'x'  
abs(-4)  
[1] 4  
abs(c(-4,4.5,-10.5,6))  
[1] 4.0 4.5 10.5 6.0
```

```
#Square Root of 'x'  
sqrt(81)  
[1] 9
```

```
#Rounds to the nearest integer that's larger than x  
ceiling(445.67)  
[1] 446
```

```
#Rounds to the nearest integer that's smaller than x  
floor(445.67)  
[1] 445
```

# General Functions

Note: 'x' here is a numeral or a vector of numerals.

```
#Rounds to the nearest integer toward 0.  
trunc(445.67)  
[1] 445  
trunc(c(445.67,19.567,33.09))  
[1] 445 19 33
```

```
#Rounds to the nearest possible value after mentioning how many digits  
#to keep after decimal point.
```

```
round(44.5682,digits=2)  
[1] 44.57
```

```
#specify the number of significant digits to be retained
```

```
signif(44.5681,digits=4)  
[1] 44.57
```

Similar to round() function,  
signif() also specify number of  
significant digits regardless of  
the size of the number.

# General Functions

Note: 'x' here is a numeral or a vector of numerals.

```
#Computes natural logarithms.
```

```
log(50)
```

```
[1] 3.912023
```

```
log(c(44,55))
```

```
[1] 3.784190 4.007333
```

---

```
#Computes binary (base 2) logarithm.
```

```
log2(8)
```

```
[1] 3
```

---

```
#Computes logarithm to the base 10.
```

```
log10(55)
```

```
[1] 1.740363
```

---

```
#Computes the exponential value , ex.
```

```
exp(6)
```

```
[1] 403.4288
```

# Operators

An operator is a symbol which helps the user to command the compiler to perform specific mathematical or logical operations. R language is rich in built-in operators and are classified into the following categories:

- Assignment Operators
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Miscellaneous Operators

# Assignment Operators

These operators are used to assign values to objects.

```
#Left Assignment (<-, <<-, =) can be done in 3 ways:
```

```
x1<-45
```

```
x1=45
```

```
x1<<-45
```

```
x1
```

```
[1] 45
```

```
#Right Assignment (->, ->>) can be done in 2 ways:
```

```
45->y1
```

```
45->>y1
```

```
y1
```

```
[1] 45
```

Different assignments all leading to same output

# Arithmetic Operators

These operators are used to perform mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R.

```
x<-7  
y<-18
```

```
#Addition  
x+y  
[1] 25
```

```
#Subtraction  
x-y  
[1] -11
```

```
#Multiplication  
x*y  
[1] 126
```

# Arithmetic Operators

```
#Division
```

```
y/x
```

```
[1] 2.571429
```

```
#Exponentiation
```

```
y^x
```

```
[1] 612220032
```

```
#Integer Division to get Remainder
```

```
y%%x
```

```
[1] 4
```

```
#Integer Division to get Quotient
```

```
y%/%x
```

```
[1] 2
```

# Relational Operators

These operators are used to compare values. The result of the comparison is the Boolean (True or False) value. Following table shows the relational operators available in R.

```
x<-7  
y<-18
```

```
#Less than  
x<y  
[1] TRUE
```

```
#Greater than  
x>y  
[1] FALSE
```

```
#Less than or equal to  
x<=5  
[1] FALSE
```

# Relational Operators

```
#Greater than or equal to  
y>=20  
[1] FALSE
```

```
#Equal to  
y==16  
[1] FALSE
```

```
#Not equal to  
x!=5  
[1] TRUE
```

# Logical Operators

Logical Operators are applicable only to vectors of logical or numeric type. They compare each element of the first vector with the corresponding element of the second vector. Below table describes the logical operators available in R with different expressions & their respective outcomes :

Expression	Outcome	Expression	Outcome
true AND true	TRUE	true OR true	TRUE
true AND false	FALSE	true OR false	TRUE
false AND false	FALSE	false OR false	FALSE
true AND missing	missing	true OR missing	TRUE
missing AND missing	missing	missing OR missing	missing
false AND missing	FALSE	false OR missing	missing

0 means False & any number >0 is True

# Logical Operators

Below table describes the logical operators available in R.

```
x<-c(FALSE,TRUE,2,5)
y<-c(TRUE,FALSE,FALSE,TRUE)
```

```
#Logical NOT
!x
[1] TRUE FALSE FALSE FALSE
```

```
#Element-wise logical AND
x&y
[1] FALSE FALSE FALSE TRUE
```

# Logical Operators

`&&` and `||` examines only the first element of the vector resulting into single length logical vector.

```
#Logical AND  
x&&y  
[1] FALSE
```

```
#Element-wise logical OR  
x|y  
[1] TRUE TRUE TRUE TRUE
```

```
#Logical OR  
x||y  
[1] TRUE
```

# Miscellaneous Operators

These operators are used for specific purpose and not general mathematical or logical operations.

#Colon operator.

```
x<-1:5  
x  
[1] 1 2 3 4 5
```

It creates simple integer sequences

```
x<-10  
t<-1:8  
x%in%t  
[1] FALSE
```

This operator is used to identify if a value belongs to a vector or array

# Quick Recap

In this session, we learnt different types of Functions and Operators in R. Here is a quick recap:

## General Functions

- `abs()`, `sqrt()` `ceiling()`, `floor()`, `trunc()`,  
`signif()`, `log()`, `log2()`, `log10()`, `exp()`

## Operators

- Assignment Operators: `<-`, `<<-`, `=`, `->`, `->>`
- Arithmetic Operators: `+`, `-`, `*`, `/`, `^`, `%%`, `%/%`
- Relational Operators: `<`, `>`, `<=`, `>=`, `==`, `!=`
- Logical Operators: `!`, `&`, `&&`, `|`, `||`
- Miscellaneous Operators: `%in%`, `:`

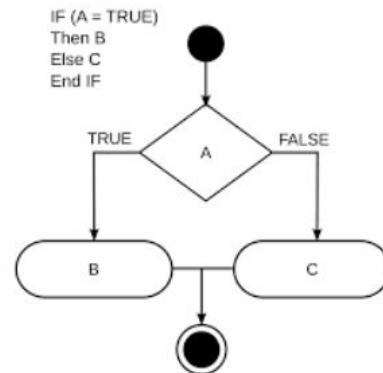
## If..else Conditional Statements

# Contents

1. Introduction
2. if Statement
3. if..else Statement
4. ifelse() Function
5. Nested if..else Statements

# Introduction

- Conditional Statements are useful in data science programming when we want to execute a code only if a certain condition is satisfied.
- Condition is referred as a logical expression where a decision is made to execute some code based on a Boolean (true or false) condition
- The most simplest form is **if** statement. **else** and **else if** statements can also be used depending on the number of conditions to be checked.



# if Statement

## Syntax:

```
if  (condition)
{
    statement(s)
}
```

The program evaluates



True  
Body of if is  
executed

False  
The statement (s) is  
not executed

# if Statement

If the condition is TRUE, the statement gets executed. But if it's FALSE, nothing happens.

```
# Check if x is a positive number or not
```

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

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

- Comparison operator '>' is used in condition.
- Here,  $x > 0$  returns TRUE; hence the print statement is executed

```
# Print the value of the object if it is of a character type
```

```
x<-"Hello there"  
if(is.character(x)) {  
  print(x)  
}
```

```
[1] "Hello there"
```

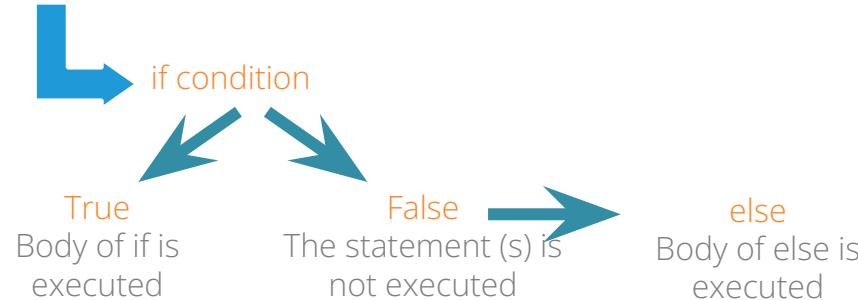
- `is.character()` returns TRUE or FALSE depending on whether the argument is of character type or not.
- Here  $x$  is a character, hence the condition returns TRUE and "Hello there" is printed

# if..else Statement

## Syntax:

```
if condition{  
    statement(s)  
} else {  
    statement(s)  
}
```

The program evaluates



The **else** statement is optional and there could be at most only one **else** statement following **if**.

# if..else Statement

**if..else** Statements are used when you want to execute one block of code when a condition is true, and another block of code when it is false.

```
# Check if x is a positive number or not and print the status
```

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

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

This condition evaluates to false; hence it will skip the body of if and the body of else will be executed.



There is an easier way to use **if..else** statement specifically for vectors. You can use **ifelse()** function instead.

# if..else Statement

```
# Take input as a number from user  
# Print the sum of natural numbers up to that number.
```

```
num=as.integer(readline(prompt="Enter Integer Value: "))  
if(num < 0) { ←  
  print("Enter a positive number")  
} else {  
  sum=(num * (num + 1)) / 2;  
  
  print(paste("The sum is", sum))  
}  
[1] "The sum is 15"
```

- ❑ readline() interactively reading a line from terminal.
- ❑ prompt= takes a string printed when prompting the user for input

Here the input is 5, hence the sum of 5 natural number is 15

# ifelse()

**ifelse()** Function is a vector equivalent form of the **if..else** statement

## Syntax:

```
ifelse(condition,x,y)
```

- If the condition is TRUE it returns x else y
- **ifelse()** function can take a vector as an input and results into a vector

```
#Create a vector & put a condition to check & print even or odd depending  
#on result.
```

```
a<-c(6,1,5,14)  
ifelse(a%%2==0,"even","odd")  
[1] "even" "odd" "odd" "even"
```

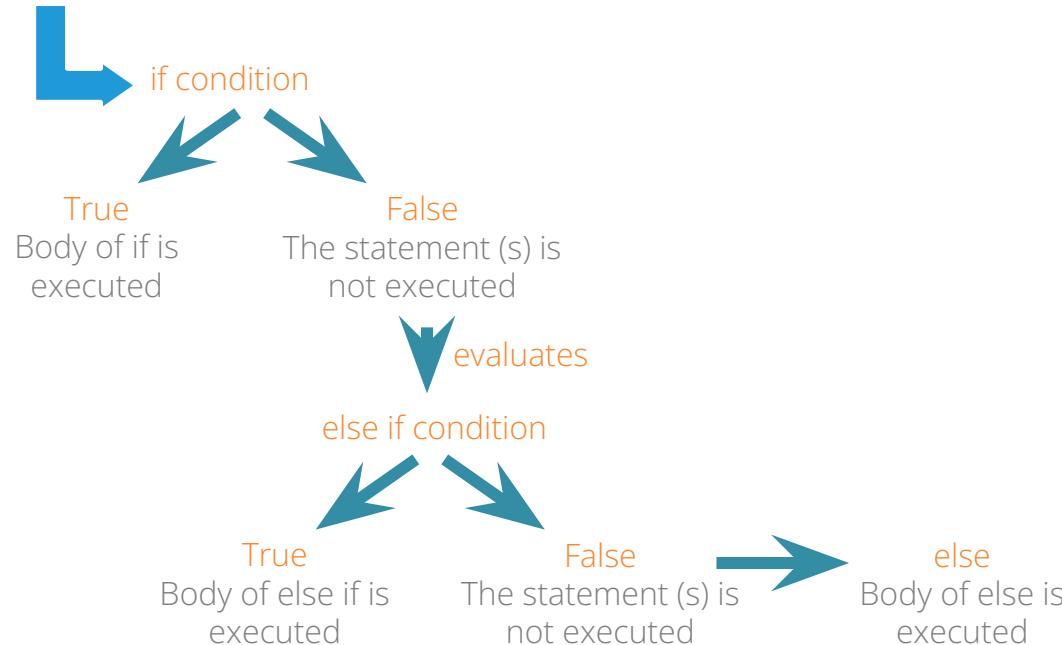
# Nested if..else Statement

## Syntax:

```
if (condition1) {  
    statement(s)  
} else if (condition2) {  
    statement(s)  
} else if (condition3) {  
    statement(s)  
} else {  
    statement(s)  
}
```

# Nested if..else Statement

The program evaluates



Only one statement gets executed depending on the conditions.

# Nested if..else Statement

Nested **if..else** Statements are used when there are several conditions. Any number of conditions can be added to **else if** statement after an **if** statement and before an **else** statement.

```
# check whether x is positive, negative or zero and print the status

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

The if condition and else if condition evaluates to FALSE so it executes the statements in body of else.

# Quick Recap

In this session, we learnt all the **if..else** conditional statements with the help of examples. Here is a quick recap:

## if statement

- If the condition is TRUE, the statement gets executed. But if it's FALSE, nothing happens.

## if..else statement

- Used when you want to execute a statement when the condition returns FALSE. **ifelse()** is the vector equivalent of the if ..else statement

## Nested if..else statement

- Used when there are several conditions
- Multiple conditions can be given using **else if** statement

# Loops in R

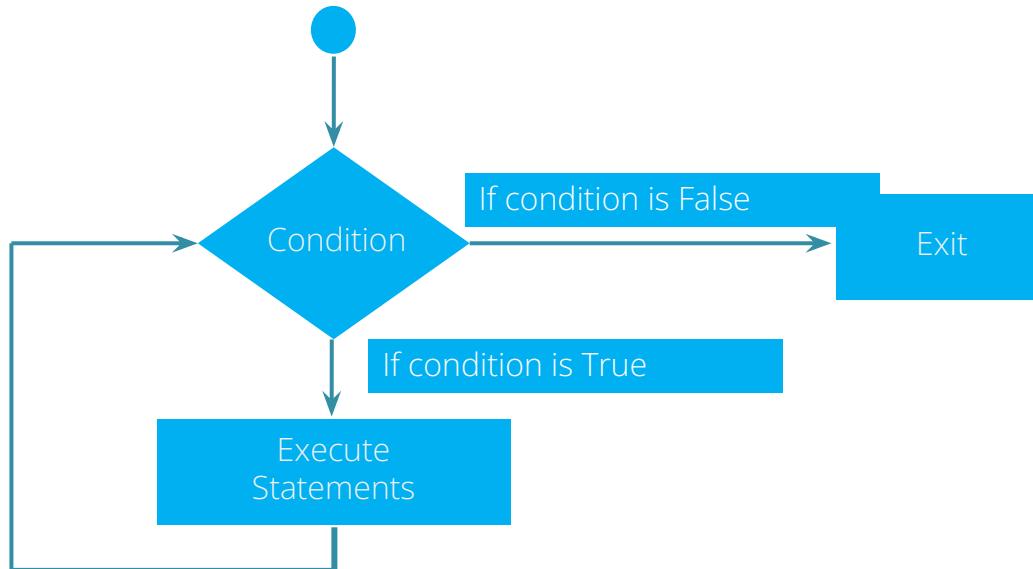
# Contents

1. Introduction
2. Types of Loops
3. for Loop
4. while Loop
5. repeat Loop
6. Interruption and Exit Statements in R

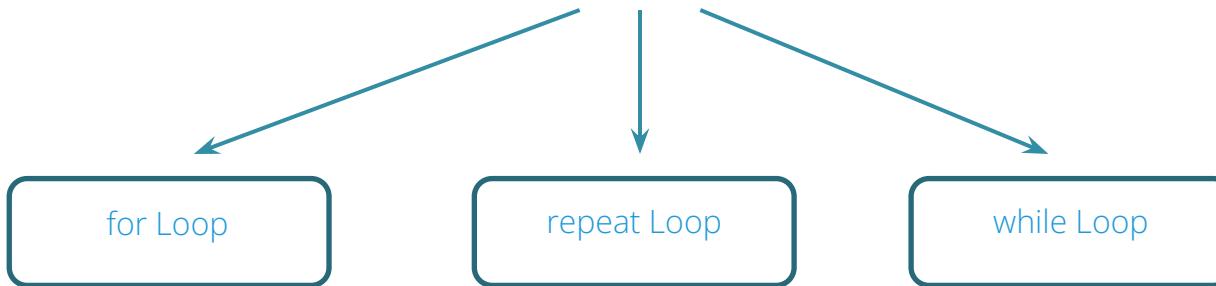
# Introduction

Loops are used when a block of codes needs to be executed multiple times (called iteration)

Flow Chart to illustrate a loop statement



# Types of Loops



- Iterates over the elements of any sequence (vector) till the condition defined is true
- Number of iterations are fixed and known in advance
- Infinite loop and used with break statement to exit the loop
- Number of iterations depends on the condition which is checked at the end of each iteration
- Repeats a statement or group of statements until some condition is met
- Number of iterations depends on the condition which is checked at the beginning of each iteration

# for Loop

## Syntax:

```
for (Loop_Variable in Sequence) {  
    Statement 1  
    Statement 2  
    ....  
}
```

Loop_Variable	sets a new value for each iteration of the loop.
Sequence	is a vector assigned to a Loop_Variable
Statement 1, Statement 2,...	Body of for consisting of block of program statements contained within curly braces

# for Loop

- In **for** loop, the number of times the loop should be executed is typically defined in the initialization part.
- Loop variable is created when the **for** statement runs.
- Loop variable is assigned the next element in the vector with each iteration and then the statements of the body are executed.

```
# Print numbers 1 to 4
```

```
for ( i in 1:4) {  
  print(i)  
}
```

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

# for Loop

```
# Print the how many times Mumbai and Delhi are appearing in the vector
```

```
Location<-c("Mumbai","Delhi","Delhi","Mumbai","Mumbai","Delhi")
count <- 0
for (i in Location) {
  if(i=="Mumbai")
    count=count+1
}
countdelhi<-length(Location)-count
print(paste("Mumbai:",count))
print(paste("Delhi:",countdelhi))
```

```
[1] "Mumbai: 3"
[1] "Delhi: 3"
```

- Here the loop iterates 6 times as the vector Location has 6 elements.
- In each iteration, i takes on the value of corresponding element of Location.
- Counter object count is used to count Mumbai in Location.
- if statement checks for Mumbai in Location and increases the count by 1.
- countdelhi is a object which has count of Delhi in Location.
- length() returns the no. of elements in the object.
- count of Mumbai is subtracted from length of Location, giving the count of Delhi.

# Nested for Loop

- The placing of one loop inside the body of another loop is called nesting.
- When you “nest” two loops, the outer loop takes control of the number of complete repetitions of the inner loop. Thus inner loop is executed N- times for every execution of Outer loop.

# Print numbers :

```
for(i in 1:5){  
  for(j in 1:2)  
{  
  print(i*j);  
 }  
 }
```

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

# Nested for Loop

## Syntax:

```
for (Loop_Variable1 in Sequence1){  
    Statement1  
    for(Loop_Variable2 in Sequence2){  
        Statement2  
        Statement3  
    }  
}
```

Loop_Variable1, Loop_Variable2	sets a new value for each iteration of the loop.
Sequence1, Sequence2	is a vector assigned to their respective Loop_Variable
Statement 1, Statement 2, Statement 3,....	Body of for consisting of block of program statements contained within curly braces

# Nested for Loop

## Important Note :

- While writing a nested for loop, always practice indentation.
- Like other languages R does not create an indentation automatically, so when you write a loop inside another loop always indent before writing a new loop so that it's easy to see & understand the code & one also understands placing of loops & where the brackets open & close.

# for Loop

- For loop is the most famous among all loops available in R and its construct implies that the number of iterations is fixed and known in advance, as in cases like "generate the first 100 prime numbers" or "enlist the 10 most important clients".
- But what if we do not know till when the loop should be iterated or control the number of iterations and one or several conditions may occur which are not predictable beforehand?
- In that case the while and repeat loops may come to the rescue.

# while Loop

## Syntax:

```
while (condition) {  
    Statement 1  
    Statement 2  
}
```

Statement 1,  
Statement 2,...

Body of while consisting of  
block of program  
statements contained within  
curly braces

# while Loop

- The initialization part defines a condition for the loop. The condition is checked every time at the beginning of the loop. The body of the loop is executed only if the condition evaluates to TRUE. This process is repeated until the condition evaluates to False.

```
# Print odd numbers between 1 to 10
```

```
i<-1  
while (i<10) {  
  print(i)  
  i=i+2  
}
```

```
[1] 1  
[1] 3  
[1] 5  
[1] 7  
[1] 9
```

Object i is incremented by 2 each time inside the loop.

# while Loop

```
# Take input as a number from user
# Print the sum of natural numbers up to that number.

number=as.integer(readline(prompt="Enter a number: "))
if(number < 0) {
  print("Enter a positive number")
} else {
  sum=0
  while(number > 0) {           ←
    sum=sum+number
    number = number - 1
  }
  print(paste("The sum is", sum))
}
[1] "The sum is 6"
```

- if condition checks whether the number is less than zero or not; if returns TRUE, it tells user to print a positive number.
- sum is a counter object; set to 0
- while loop is used to iterate until the number becomes 0 till then the value of number is added to sum and number is decremented by 1 with each iteration and when it becomes 0, the loop will stop and result will be printed.
- Here the input is 3, hence the sum of 5 natural number is 6



Note : readline() & prompt = are explained in if..else Conditional Statements tutorial.

# repeat Loop

- This loop is similar to **while** loop, but it is used when we want the blocks of statements to be executed at least once, no matter what the result of the condition.
- Note that you had to set a condition within the loop with a **break** statement to exit otherwise the loop would have executed infinite times. This statement introduces us to the notion of exiting or interrupting cycles within loops.

# repeat Loop

## Syntax:

```
repeat {  
    Statement 1  
    Statement 2  
    if (condition) {  
        break  
    }  
}
```

break	Terminates the loop statement and transfers execution to the statement immediately following the loop.
Statement 1, Statement 2,...	Body of repeat consisting of block of program statements contained within curly braces

# repeat Loop

- The condition is placed at the end of the loop body, so the statements inside the loop body are executed at least once, no matter what the result of condition and are repeated until the condition evaluates to FALSE.

```
# Print even numbers between 1 to 10
```

```
total<-0
repeat {
  total<-total+2
  print(total)
  if(total > 8)
    break
}
```

total is a counter object; set to 0.  
total will be incremented by 2 with each iteration and  
if it becomes greater than 8 the loop will stop

```
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
```

# Interruption and Exit Statements in R

How do you exit from a loop?

Can you stop or interrupt the loop, aside from the “natural” end, which occurs either because you reached the given number of iterations (for) or because you met a condition (while, repeat)?

And if yes how?

- **break** statement responds to the first question. It passes the control to the statement immediately after the end of the loop (if any). We have already seen how use of **break** statement in the last example.
- **next** statement discontinues the current iteration of the loop without terminating it and starts next iteration of the loop

# Interruption and Exit Statements in R

```
# print numbers between 1 to 15 which are divisible by 4
```

```
for (i in 1:15) {  
  if (i%4){  
    next  
  }  
  print(i)  
}
```

```
[1] 4  
[1] 8  
[1] 12
```

if statement checks whether value in i is divisible by 4 or not; if it returns TRUE then it skips all the statements after that and starts the next iteration of the loop.

# Quick Recap

In this session, we learnt how different types of loops are used in programming to repeat a specific block of code. Here is a quick recap .

for loop

- Number of iterations are defined in the beginning and runs till the condition defined is TRUE
- Counter is automatically incremented
- Nested for loop is used for multiple conditions

while loop

- Condition is checked in the beginning of the loop
- Counter is incremented inside the loop

repeat loop

- It's an infinite loop; a condition is set within the loop with the **break** statement to exit
- Counter is incremented inside the loop

Interruption and  
exit statements

- **break**: exits the loop
- **next**: discontinues the current iteration

# Working with Dates and Time in R

1. Introduction
2. Base Package Functions
3. Package lubridate
4. Package lubridate Functions
5. Merge Three Different Columns Into a Date in R
6. Format a Vector With Inconsistent Date Formats

# Introduction

- R has a range of built-in functions that allow us to work with dates and time.
- Working on dates and time can be tedious when the data come with date values in different format.
- The base function of R, **as.Date()** converts a variety of character date formats into R dates. Once converted to dates, the following functions will return information about dates: **weekdays()**, **months()**, **quarters()**, **seq()**.
- **as.Date()** handles only dates.
- For handling both dates and time there is a package called lubridate. The inbuilt function of this package offers a nice way to make easy parsing in dates and times.

In this tutorial we will see how base function **as.Date()** & other related functions and package lubridate works and also learn date manipulation tasks.

# Base Package Functions

`as.Date()` converts dates entered as strings into numeric dates.

`as.Date(x, format = "%Y-%m-%d")`

x is a string object to be converted

`format=` is the format (in which the date appears within the string) composed of codes such as:

	day as a number (01-31)	%d
Day	abbreviated weekday (Mon)	%a
	full weekday name (Monday)	%A
	abbreviated month (Jan)	%b
Month	full month name (January)	%B
	month as a number (01-12)	%m
Year	2-digit year (16)	%y
	4-digit year (2016)	%Y

# Base Package Functions

```
# Formatting a date
```

```
sdate1<- "5jan2010"  
ndate1<-as.Date(sdate1,format="%d%b%Y")  
ndate1
```

```
[1] "2010-01-05"
```

```
class(ndate1)
```

```
[1] "Date"
```

Default format for dates in as.Date() is YYYY-MM-DD — four digits for year, and two digits for month and day, separated by a hyphen.

```
# Using format argument to extract parts of date
```

```
format(ndate1,format="%Y")
```

```
[1] "2010"
```

```
format(ndate1,format="%Y%B")
```

```
[1] "2010January"
```

Format codes can also be used to extract parts of dates using format()

# Base Package Functions

Once you have converted to dates, the following functions will return information about dates:

```
#To extract Day of the week.
```

```
weekdays(ndate1)
```

```
[1] "Monday"
```

```
#To extract Month of the Year.
```

```
months(ndate1)
```

```
[1] "August"
```

```
#To extract Quarter no.
```

```
quarters(ndate1)
```

```
[1] "Q3"
```

```
#Generates dates sequences for Date object by 2 months.
```

```
x<-seq(ndate1,by="2 months",length.out=5)
```

```
x
```

```
[1] "2016-08-08" "2016-10-08" "2016-12-08" "2017-02-08" "2017-04-08"
```

# Base Package Functions

```
# Capture current date
```

```
today<-Sys.Date()  
today
```

```
[1] "2017-01-05"
```

Sys.Date() returns the your system's current date

```
# Using operators with dates
```

```
d1<-as.Date("20101201",format="%Y%m%d")  
d2<-as.Date("10/7/04",format="%m/%d/%y")
```

```
d1
```

```
d2
```

```
[1] "2010-12-01"
```

```
[1] "2004-10-07"
```

```
d1-d2
```

Time difference of 2246 days

```
d1-d2>365
```

Different operators can be used with date objects

```
[1] TRUE
```

# Package lubridate

- This package is developed by Garrett Grolemund and Hadley Wickham.
- lubridate offers many useful functions to work with date-times and timespans which makes basic date-time manipulations much more straightforward.
- It works for most of the popular date-time object classes (Date, POSIXt, chron, etc.), which is not always true for base R functions.

```
# Install and load package lubridate  
install.packages("lubridate")  
library(lubridate)
```

# Package lubridate Functions

```
#To parse character strings into dates.
```

```
mdy("12-01-2015")  
[1] "2015-12-01"
```

The letters `y`, `m`, and `d` correspond to the year, month, and day elements of a date-time. To read in a date, choose the function name that matches the order of elements in your date-time object.

```
Other functions are: ymd(), ydm(), dmy(), hm(), hms() and ymd_hms()
```

```
#To capture the Current date and time.
```

```
date<-now()  
date  
[1] "2019-04-11 12:48:00 IST"
```

# Package lubridate Functions

```
#To extract the hour component from the date object.
```

```
hour(date)
```

```
[1] 12
```

```
#To extract the minute component from the date object.
```

```
minute(date)
```

```
[1] 38
```

```
#To extract the second component from the date object.
```

```
second(date)
```

```
[1] 59.24696
```

# Merge Three Different Columns Into a Date in R

```
# My Dataframe
```

```
EmpID<-c(101,102,103,104,105)
year<-c(1977,1989,2000,2012,2015)
month<-c(2,5,10,1,11)
day<-c(2,3,1,1,5)

datedf<-data.frame(EmpID,year,month,day)
datedf
```

	EmpID	year	month	day
1	101	1977	2	2
2	102	1989	5	3
3	103	2000	10	1
4	104	2012	1	1
5	105	2015	11	5

Data: Employee ID (EmpID) and joining date (split into 3 columns: year month & day)

# Merge Three Different Columns Into a Date in R

We are having 3 separate columns as year, month, and day in our dataframe **datedf**.

```
# Merge 3 columns into one date column
```

```
datedf$date<-as.Date(paste(datedf$year,datedf$month,datedf$day,  
sep=' - '),format="%Y-%m-%d")  
datedf
```

	EmpID	year	month	day	date
1	101	1977	2	2	1977-02-02
2	102	1989	5	3	1989-05-03
3	103	2000	10	1	2000-10-01
4	104	2012	1	1	2012-01-01
5	105	2015	11	5	2015-11-05

- new column date is created using \$
- paste() combines the columns
- as.date() converts date column into a date type

# Format a Vector With Inconsistent Date Formats

Converting dates entered as strings into numeric dates in R is a little tricky if the date information is not represented consistently. Let's see how to deal with this kind of situation.

```
dates<-c("12aug08","01sep09","7august06","9august2007","20july1999")
ndates<-as.Date(dates,format="%d%b%y")

ndates
[1] "2008-08-12" "2009-09-01" "2006-08-07" "2020-08-09" "2019-07-20"
```

*In Vector `dates`, there are multiple date formats. We have specified a date format that appears appropriate for the first few dates including `%d`, which allows for days within a month to optionally have a leading zero when less than 10, and `%b`, which can match either an entire or an abbreviate month name. Unfortunately, we have some years that appear as 2 digits and some that appear as 4. As a result, we can see in our results that the first three dates are correct and the last two are not*

# Format a Vector With Inconsistent Date Formats

In the loop below, we go through our vector of numeric dates and see if any appear later than 2018. For these, we assume the date is presented with 4 digits and reread the string with the appropriate format.

```
for (i in 1:length(ndates)){
  if ((ndates[i])> as.Date("2018-01-01")){
    ndates[i] <- as.Date(dates[i],format="%d%B%Y")
  }
}
ndates
[1] "2008-08-12" "2009-09-01" "2006-08-07" "2007-08-09" "1999-07-20"
```

length() returns the length of an object  
for loop is executed from 1 to the length of  
ndates i.e. 5

# Format a Vector With Inconsistent Date Formats

The code we used here cannot be applied to many situations, but the steps we used can be:

- Use a format that is appropriate for as many dates as possible, focusing on the more flexible formats offered by R.
- Determine when the format did not work properly and define a rule for finding such cases.
- Use a format appropriate to the misread dates.

# Quick Recap

In this session, we learnt how to deal with dates and time using base package functions in R & package lubridate, how to merge 3 different columns into one date column and how to format a vector with inconsistent dates. Here is a quick recap:

## Base functions

- `as.date()`, `weekdays()`, `months()`,  
`quarters()`, `seq()`, `Sys.Date()`

## Package lubridate functions

- `ymd()` `series`, `now()`, `hour()`,  
`minute()`, `second()`

## Date manipulation tasks

- Merge Three Different Columns Into a Date in R using `paste()`

# Importing & Exporting Data

(CSV, TXT and XLSX, SAS, STATA, SPSS,  
MySQL, PostgreSQL and Oracle)



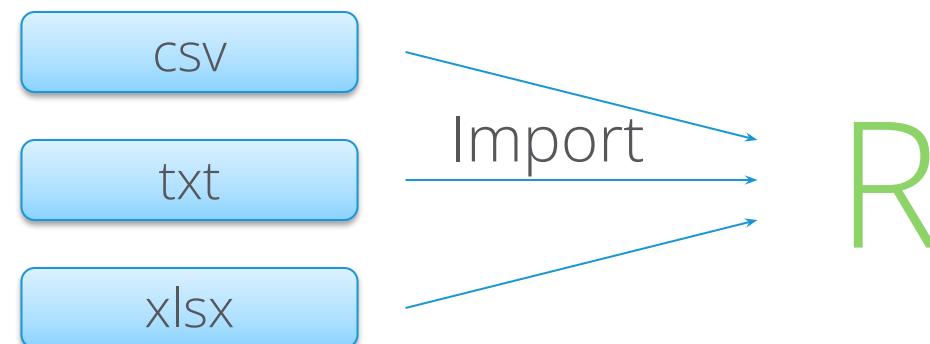
# Contents

1. Importing Files using Base Functions
  - `read.csv()`
  - `read.table()`
2. Handle Missing Observations
3. Importing Files Using different Packages
  - `readr`
  - `data.table`
  - `readxl`
4. Importing Files Interactively
5. Importing SAS, STATA and SPSS Files Using
6. Importing MySQL, PostgreSQL and Oracle database
7. Exporting data files of various formats



# About Importing Data

- To do any kind of data analysis in R, you first need to have a data in a file somewhere, either locally or on the Web. If the file is on the web, download and save the data in your system and load it into R to work on it.
- In R, the general term used for loading data is **Importing data**.
- The data is stored in the form of files and files can be in any format (csv, txt, xlxs,etc). There are several ways to import these files in R which we will learn in this tutorial.
- Let's first understand what are the most commonly used file formats to store data.



# Common File Formats

The most commonly used file formats for storing data

CSV:

Comma Separated Values file. Allows data to be saved in a table structured format; with commas acting as separators.

File	Edit	Format	View	Help
First_Name,Last_Name,Grade,Location,ba,ms				
Alan,Brown,GR1,DELHI,17990,16070				
Agatha,Williams,GR2,MUMBAI,12390,6630				
Rajesh,Kolte,GR1,MUMBAI,19250,14960				
Ameet,Mishra,GR2,DELHI,14780,9300				
Neha,Rao,missing,MUMBAI,19235,15200				
Sagar,Chavan,GR2,MUMBAI,13390,6700				
Aaron,Jones,GR1,MUMBAI,23280,13490				
John,Patil,GR2,MUMBAI,13500,10760				
Sneha,Joshi,GR1,DELHI,20660,missing				
Gaurav,Singh,GR2,DELHI,13760,13220				
Adela,Thomas,GR2,DELHI,13660,6840				
Anup,Save,GR2,MUMBAI,11960,7880				

TXT:

Text file. Stores data in plain text format, separated by different types of delimiters such as blank space, tab ("\t"), comma (","), pipe ("|"), etc.

File	Edit	Format	View	Help
First_Name	Last_Name	Grade	Location	ba ms
Alan	Brown	GR1	DELHI	17990 16070
Agatha	Williams	GR2	MUMBAI	12390 6630
Rajesh	Kolte	GR1	MUMBAI	19250 14960
Ameet	Mishra	GR2	DELHI	14780 9300
Neha	Rao	missing	MUMBAI	19235 15200
Sagar	Chavan	GR2	MUMBAI	13390 6700
Aaron	Jones	GR1	MUMBAI	23280 13490
John	Patil	GR2	MUMBAI	13500 10760
Sneha	Joshi	GR1	DELHI	20660 missing
Gaurav	Singh	GR2	DELHI	13760 13220
Adela	Thomas	GR2	DELHI	13660 6840
Anup	Save	GR2	MUMBAI	11960 7880



# Data Snapshot

basic\_salary data consist salary of each employee with it's Location & Grade.

**Variables**

First_Name	Last_Name	Grade	Location	ba	ms
Alan	Brown	GR1	DELHI	17990	16070
Columns	Description	Type	Measurement	Possible values	
First_Name	First Name	character	-	-	
Last_Name	Last Name	character	-	-	
Grade	Grade	character	GR1, GR2	2	
Location	Location	character	DELHI, MUMBAI	2	
ba	Basic Allowance	numeric	Rs.	positive values	
ms	Management Supplements	numeric	Rs.	positive values	

**Observations**

# read.csv() Function

One way of reading csv files is through **read.csv()**.

```
salary_data <- read.csv("C:/Users/Documents/basic_salary.csv")
```

**read.csv()** assumes **header = TRUE** and **sep = ","** by default.



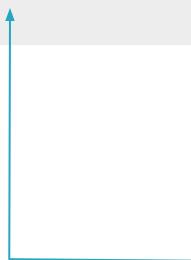
First locate your data file, whether it is saved in the default working directory of R or any other location in your system. If it is not stored in default working directory then you will have to give its path for importing it into R. If you copy file path from the folder, ensure it uses forward slash (/). Do not forget to accurately write the file name and extension.

# read.table() Function

Importing a .csv file

**read.table()** is almost identical to **read.csv()** and differ from it in two aspects: **sep**, & **header** arguments.

```
salary_data <- read.table("C:/Users/Documents/basic_salary.csv",  
header = TRUE, sep = ",")
```

- 
- ❑ **header = TRUE** (logical) indicates that the first row of the file contains the names of the columns. Default is **header = FALSE**
  - ❑ **sep = ","** specifies that the data is separated by comma. Without this command, R imports data in a single column.

# read.table() Function

Importing a [.txt](#) (tab delimited) file

```
salary_data <- read.table("C:/Users/Documents/basic_salary.txt",  
header = TRUE, sep = "\t", fill= TRUE)
```

**fill= TRUE** implicitly adds blank fields, in case the rows have unequal length

- Apart from the functions in the base package, R also has several additional packages to carry out efficient data import like **readr**, **data.table**, **readxl**.

# How does R Handle Missing Observations?

- Your data may contain some missing value(s) that need(s) to be handled effectively to reduce bias and to produce correct results.
- There are many ways of handling missing values in R. Missing values in R appears as NA.
- NA is not a string or a numeric value, but an indicator of missing values.
- While importing files using **read.table()** and **read.csv()** missing values are automatically filled with NA.



Sometimes data may contain random values which are considered as missing values. It is important to detect those values and overcome them. Therefore, we have a special tutorial for Handling missing values where you will learn to deal with different types of missing data.

# Package `readr`

- The package **readr** has functions for importing files and ensures faster imports, proving useful when the data under study is large.

```
# Install and load readr
```

```
install.packages("readr")  
library(readr)
```

```
# Importing a .csv file
```

```
salary_data<-read_csv("C:/Users/Documents/basic_salary.csv")
```

```
# Importing a .txt file (white space delimited)
```

```
salary_data<-read_table("C:/Users/Documents/basic_salary.txt")
```

```
# Importing a .txt file (tab delimited)
```

```
salary_data<-read_tsv("C:/Users/Documents/basic_salary.txt")
```

# Package data.table

- The package **data.table** provides many functions for data manipulation tasks. Its **fread()** function is meant to import data from regular(i.e. every row of your data needs to have the same number of columns) delimited files directly into R and is one of the most efficient means of dealing with very large data.
- It is much faster and convenient than other methods and one of the great things about this function is that all controls, expressed in arguments such as **sep** are automatically detected.

```
# Install and load data.table  
# Import file using fread()  
  
install.packages("data.table")  
library(data.table)  
salary_data<-fread("C:/Users/Documents/basic_salary.csv")
```

# Package readxl

- `readxl` is an R package that provides function to read Excel worksheets in both `.xls` and `.xlsx` formats.

```
# Install and load readxl  
# Import file using read_excel  
  
install.packages("readxl")  
library(readxl)  
salary_data<-read_excel("C:/Users/Documents/basic_salary.xlsx")
```

# Importing Files Interactively

- To manually select the directory and file where your dataset is located use **file.choose()** function .
- **file.choose()** can be used as an independent function or can be put inside a function with or without other arguments.

```
salary_data <- read.csv(file.choose())
```

This opens a dialog box that allows you to choose the file interactively.

# Importing SAS , STATA , SPSS using Package foreign

- The package `foreign` is used to import data from SAS, STATA and SPSS.

```
# Install and load package foreign
```

```
install.packages("foreign")
library(foreign)
```

```
# For SAS
```

```
# Save SAS dataset in transport format. Requires SAS on your system.
```

```
read.xport("dataset.xpt")
```

```
# For SPSS
```

```
read.spss("dataset.sav",use.value.labels=TRUE)
```

```
# For Stata binary
```

```
read.dta("dataset.dta")
```



`read.dta()` function reads a file in Stata version 5–12 binary format into a data frame

# Importing MySQL Database Package RMySQL

- To work with MySQL database in R, use package RMySQL:

```
install.packages("RMySQL")  
library(RMySQL)
```

```
# Create a database connection object.
```

```
mydb <- dbConnect(MySQL(), user='user', password='password',  
dbname='database_name', host='host')
```

```
# Save a results set object to retrieve data from the database
```

```
rs = dbSendQuery(mydb, "select * from some_table")
```



Queries can be run using the **dbSendQuery()**.

```
# Access the result in R
```

```
data = fetch(rs, n=-1)
```



To access the results in R, **fetch()** is used as it saves the results of the query as a data frame object.

**n=** specifies no. of records to be retrieved.  
**n=-1** retrieves all pending records.

# Importing PostgreSQL Database Using Package RPostgreSQL

- To work with PostgreSQL database in R, use package RPostgreSQL:

```
install.packages("RPostgreSQL")  
library(RPostgreSQL)
```

```
# Create a database connection object.
```

```
drv <- dbDriver("PostgreSQL")
```

This command establishes connection to PostgreSQL.

```
con <- dbConnect(drv, host='host', port='port', dbname='database_name',  
user='user', password='password')
```

```
# Obtain a table into R data frame
```

```
myTable <- dbReadTable(con, "tablename")
```

# Package RODBC

- There are several packages in R which allow the user to connect to an Oracle database. The most commonly used packages are: RODBC, RJDBC and ROracle.

RODBC – implements ODBC database connectivity.

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

```
# Create a database connection object.
```

```
con <- odbcConnect("data", uid="user", pwd="password")
```

```
# Query the database and put the results into a data frame
```

```
mydata <- sqlQuery(con, "SELECT * FROM TABLENAME.DATATABASE")
```



**sqlQuery(channel,query,errors)**

- **channel** = database connection object
- **errors** = logical : if TRUE halt's and display a character vector of error message(s), else return -1. Default is FALSE

# Importing Oracle Database Using Package RODBC, RJDBC and ROracle

Additionally, RJDBC package is based on the database interface (DBI) uses JDBC as the back-end connection to the database.

ROracle is an open source R package supporting a DBI-compliant Oracle driver based on the high performance OCI library. It requires Oracle Instant Client or Oracle Database Client to be installed on the client machine.

# Exporting CSV, TXT and XLSX Files

Sometimes you may want to export data saved as object from R workspace to different file formats. Methods for Exporting R objects into CSV, TXT and XLSX formats are given below:

```
# To a CSV File
```

```
write.csv(mydata, file = "MyData.csv")
```

mydata is object name which is saved in csv format with filename **MyData** in default working directory.

```
# To a Tab Delimited Text File
```

```
write.table(mydata, "c:/mydata.txt", sep="\t")
```

mydata object will be saved as tab delimited txt file as we have specified the **sep= "\t"**. It will be saved in C drive of the system.

```
# To a Excel Spreadsheet
```

```
install.packages("xlsx")
```

```
library(xlsx)
```

```
write.xlsx(mydata, "c:/mydata.xlsx")
```

package **xlsx** is used to import .xlsx files

# Exporting SPSS, SAS and STATA Files

Methods for Exporting R objects into SPSS, SAS and STATA formats using package `foreign`:

```
install.packages("foreign")
library(foreign)

write.foreign(df, datafile, codefile, package = c("SPSS", "Stata", "SAS"), ...)
# To SPSS
write.foreign(mydata, "c:/mydata.txt", "c:/mydata.sps", package="SPSS")

# To SAS
write.foreign(mydata, "c:/mydata.txt", "c:/mydata.sas", package="SAS")

# To STATA
write.dta(mydata, "c:/mydata.dta")
```

# Quick Recap

In this session, how to import and export data using different functions and packages.

Here is a quick recap:

## Import Files Using Base Functions

- **read.table()** is used for importing csv and txt files
- **read.csv()** is used specifically for importing csv files and is more efficient than **read.table()**
- **file.choose()** is used to manually select the file of any format from wherever it is located. This function can be put inside another function.

## Handle Missing Values

- Missing Values are filled with NA while importing files using **read.csv()** and **read.table()**.

## Import Files Using Packages

- **readr**, **data.table** and **readxl** provides functions importing data in different file formats. These packages are very helpful in case of large data.

## Export Files Using Base Functions and Packages

- **write.csv()** exports csv files
- **write.table()** exports txt files
- package **xlsx** provides function **write.xlsx()** to export excel files

# Quick Recap

In this session, we learnt the ways of importing SAS, SPSS, STATA files and MySQL, PostgreSQL and Oracle database Here is a quick recap:

Importing SAS,  
SPSS and STATA  
files

- Packages: **foreign** and **Hmisc**.
- For SPSS and SAS, **Hmisc** is recommended for ease and functionality

Importing MySQL  
database

- Package: **RMySQL**

Importing  
PostgreSQL  
database

- Package: **RPostgreSQL**

Importing Oracle  
database

- Packages: **RODBC**, **RJDBC**, **ROracle**

Exporting SAS,  
SPSS and STATA  
files

- Package: **foreign**

# Checking & Modifying Data

# Contents

1. Importance of Checking Data
2. Know the Dimensions of Data and Variable Names
3. Display the Internal Structure of Data
4. Check the levels of a Categorical Variable
5. Check the Size of an Object
6. Check the number of Missing Observations
7. Display First n Rows of Data
8. Display Last n Rows of Data
9. Summarise Your Data
10. Change Variable Names and Content of Data
11. Remove Columns from a Data Frame
12. Remove Rows from a Data Frame
13. Derive a New Variable
14. Recode a Categorical Variable
15. Recode a Continuous Variable into Categorical Variable

# Data Snapshot

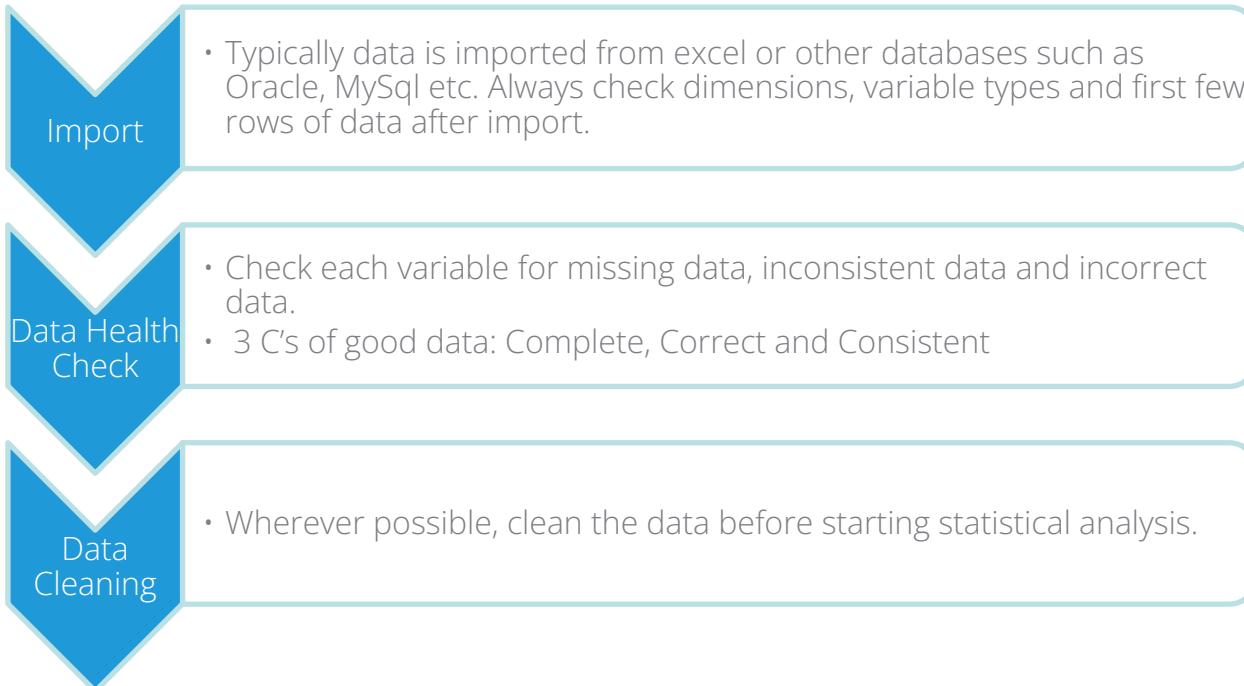
basic\_salary data consist salary of each employee with it's Location & Grade.

**Variables**

First_Name	Last_Name	Grade	Location	ba	ms
Alan	Brown	GR1	DELHI	17990	16070
Columns	Description	Type	Measurement	Possible values	
First_Name	First Name	character	-	-	
Last_Name	Last Name	character	-	-	
Grade	Grade	character	GR1, GR2	2	
Location	Location	character	DELHI, MUMBAI	2	
ba	Basic Allowance	numeric	Rs.	positive values	
ms	Management Supplements	numeric	Rs.	positive values	

**Observations**

# First Look at the Data



# Why Data Checking is Important?

After importing the data into R and before analyzing the data, it is very important to understand your data & check that it has maintained the correct format.

It is good practice to check number of rows/columns, variables types, number of missing values and first few rows of the data. .

```
# Import basic_salary data  
salary_data <- read.csv("basic_salary.csv", header=TRUE)
```

# Dimension of Data and Names of the Columns

Suppose we want to know how many rows and columns are there in our data and the names of the columns it contains, we could ask R like this:

```
# Retrieve the Dimension of your data using dim()
```

```
dim(salary_data)
```

```
[1] 12 6
```



Data contains 12 rows and 6 columns

- Also if one wants to know no. of rows and columns separately, `nrow()` and `ncol()` command can be used respectively.

```
# Get the Names of the columns using names()
```

```
names(salary_data)
```

```
[1] "First_Name"  "Last_Name"   "Grade"      "Location"
```

```
[5] "ba"          "ms"
```

# Internal Structure of Data

When R reads data, it treats different variable types in different ways. **str()** is the easiest way to inspect how R treats variables in our dataframe. It compactly displays a dataframe's internal structure like this:

```
str(salary_data)
```

Character variables are entered into a dataframe as factors in R

# Output

```
'data.frame': 12 obs. of 6 variables:  
 $ First_Name: Factor w/ 12 levels "Aaron","Adela",...: 4 3 10 5 9 11 1 8 12 7 ...  
 $ Last_Name : Factor w/ 12 levels "Brown","Chavan",...: 1 12 5 6 8 2 3 7 4 10 ...  
 $ Grade     : Factor w/ 2 levels "GR1","GR2": 1 2 1 2 1 2 1 2 1 2 ...  
 $ Location   : Factor w/ 2 levels "DELHI","MUMBAI": 1 2 2 1 2 2 2 2 1 1 ...  
 $ ba         : int 17990 12390 19250 14780 19235 13390 23280 13500 20660 13760 ...  
 $ ms         : int 16070 6630 14960 9300 15200 6700 13490 10760 NA 13220 ...
```

Structure gives the following information:

- Class of the object like in this case 'salary\_data' is of 'data.frame' class
- Type of the variable.
- Number of levels of each factor.
- Some values of the first few rows

# Check the levels

Our data has 4 factor variables. A factor is a categorical variable that can take only one of a fixed, finite set of possibilities. Those possible categories are the levels. We can check the levels using **levels()** function

```
levels(salary_data$Grade)
```

```
[1] "GR1" "GR2"
```

# Assigning individual levels :

```
levels(salary_data$Grade)[1]<-“G1”
```

```
levels(salary_data$Grade)
```

```
[1] "G1" "GR2"
```

# Assigning levels as a group :

```
levels(salary_data$Grade)<-c(“G1”, “G2”)
```

```
levels(salary_data$Grade)
```

```
[1] "G1" "G2"
```

# Check the Size of an Object

Suppose we want to know how much memory space is used to store `salary_data` object, we can use `object.size()` function to get an estimate in bytes.

```
object.size(salary_data)
```

```
4584 bytes
```

```
# Get the size of all the objects present in the current environment
```

```
for(i in ls()){
  message(i); print(object.size(get(i)))
}
```

```
a3
```

```
72 bytes
```

```
access_secret
```

```
136 bytes
```

```
access_token
```

```
.
```

```
.
```

This `for` loop prints all the objects in the current environment and their respective memory size in bytes

- `ls()` gives the vector of names of objects in the specified environment. With no argument, it shows what data sets and functions a user has defined.
- `message()` coerces the object/s to character (which is pasted together with no separator).
- `get()` returns the value of a named object.

# Number of Missing Observations

Our data might contain some missing values or observations. In R missing data are usually recorded as NA. We can check the number of missing observations like this:

```
nmiss<-sum(is.na(salary_data$ms))  
nmiss
```

```
[1] 1
```

- \$ operator after a data frame object helps in selecting a column from the data frame.
- is.na() returns a logical vector giving information about missing values.
- sum() displays the sum of missing observations.

# First 'n' Rows of Data

Now if we want to have an idea about how our data looks like without displaying the entire data set, which could have millions of rows and thousands of columns then we can use **head()** to obtain first n observations.

```
head(salary_data)
```

# Output

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
2	Agatha	Williams	GR2	MUMBAI	12390	6630
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
4	Ameet	Mishra	GR2	DELHI	14780	9300
5	Neha	Rao	GR1	MUMBAI	19235	15200
6	Sagar	Chavan	GR2	MUMBAI	13390	6700



By default, **head()** displays the first 6 rows

# First n Rows of Data

The no. of rows to be displayed can be customised to n

```
head(salary_data, n=2)
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
2	Agatha	Williams	GR2	MUMBAI	12390	6630
.						

# Last 'n' Rows of data

Now we will see the last n rows of our data using `tail()`. By default, it displays last 6 rows.

```
tail(salary_data)
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
7	Aaron	Jones	GR1	MUMBAI	23280	13490
8	John	Patil	GR2	MUMBAI	13500	10760
9	Sneha	Joshi	GR1	DELHI	20660	NA
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Anup	Save	GR2	MUMBAI	11960	7880

## Last n Rows of Data

The no. of rows to be displayed can be customised to n

```
tail(salary_data, n=2)
```

# Output

	First_Name	Last_Name	Grade	Location	ba	ms
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Anup	Save	GR2	MUMBAI	11960	7880

# Summarizing Data

We can also inspect our data using **summary()**. This function gives summary of objects including datasets, variables, linear models, etc.

```
# Variables are summarized based on their type
```

```
summary(salary_data)
```

First_Name	Last_Name	Grade	Location
Aaron	Brown	GR1:5	DELHI :5
Adela	Chavan	GR2:7	MUMBAI:7
Agatha	Jones		
Alan	Joshi		
Ameet	Kolte		
Anup	Mishra		
(Other):6	(Other):6		
ba	ms		
Min. :11960	Min. : 6630		
1st Qu.:13472	1st Qu.: 7360		
Median :14270	Median :10760		
Mean :16155	Mean :11005		
3rd Qu.:19239	3rd Qu.:14225		
Max. :23280	Max. :16070		
	NA's :1		

When **summary()** is applied to a dataframe, it is essentially applied to each column, and it summarizes all the columns.

- For a continuous variable, it gives summary in the form of : min, 1<sup>st</sup> quantile, median, mean, 3<sup>rd</sup> quantile, max and count of NA's (if any).
- For a factor(categorical) variable, it gives the frequency of each level or category like in this case: Grade has 2 levels-GR1 and GR2 occurring 5 and 7 times respectively.

# Quick Recap

In this session, we learnt how to check data features in R and why we should do it.

Here is the quick recap of functions used for checking data features:

Check the dimensionality and variable names or column names

Check the compact internal structure, levels of categorical variable and size of an object

Check the missing values (if any), first and last n rows

Summaries data

- **dim()** returns the count of rows and columns
- **names()** returns variable names or column names.
- **str()** returns many useful pieces of information like class of the object, data type of each column.
- **levels()** returns the value of the levels of the object
- **object.size()** returns the size of an object in bytes.
- **is.na()** returns a logical vector telling whether the data has missing values or not.
- **head()** returns the first n rows of data.
- **tail()** returns the last n rows of data.
- **summary()** summarizes the data based on the type of variable it contains.

# Change Variable Names and Content – fix()

```
# Import basic_salary data  
salary_data <- read.csv("basic_salary.csv", header=TRUE)
```

In case we want to change the name of some variable or column and its values. We can use the following command which lets us make changes interactively.

```
fix(salary_data) ←
```

fix() function will open the data in a separate, editable Data Editor like the screenshot below. You can go to the desired column name or cell and make the changes manually. Complete the task by closing the dialogue box File >Close.

	First_Name	Last_Name	Grade	Location	ba	ms	var7	var8
1	Alan	Brown	GR1	DELHI	17990	16070		
2	Agatha	Williams	GR2	MUMBAI	12390	6630		
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960		
4	Ameet	Mishra	GR2	DELHI	14780	9300		
5	Neha	Rao	GR1	MUMBAI	19235	15200		
6	Sagar	Chavan	GR2	MUMBAI	13390	6700		
7	Aaron	Jones	GR1	MUMBAI	23280	13490		



Alternative function for editing data interactively is edit(), similar to fix(). The difference is: edit() lets you edit an object and returns the new version and fix() lets you edit an object and modifies the original.

# Change Variable Names – names()

```
#Renaming column names with R's built-in function names() :
```

```
names(salary_data)[names(salary_data)=="ba"] <- "basic allowance"  
names(salary_data)
```

```
[1] "First_Name" "Last_Name" "Grade" "Location" "basic allowance"  
[6] "ms"
```



Note that this modifies `salary_data` directly; i.e. you don't have to save the result back into `salary_data`.

# Remove Columns from a Data Frame

```
# To remove column Last_Name from salary_data.
```

```
salary_data$Last_Name<-NULL  
head(salary_data)
```

To remove a column, set it to **NULL**

	First_Name	Grade	Location	ba	ms	newvariable
1	Alan	GR1	DELHI	17990	16070	899.50
2	Agatha	GR2	MUMBAI	12390	6630	619.50
3	Rajesh	GR1	MUMBAI	19250	14960	962.50
4	Ameet	GR2	DELHI	14780	9300	739.00
5	Neha	GR1	MUMBAI	19235	15200	961.75
6	Sagar	GR2	MUMBAI	13390	6700	669.50

You can remove columns using integer indexing also like this:

```
salary_data[6]<-NULL
```



Index is the position of column. It starts from 1.  
In this case index number of `First_Name` is 1, `Grade` is 2 and so on and so forth.

# Remove Rows from a Data Frame

We can remove unwanted rows from our data by using their index nos.

Suppose we want to remove rows 2, 3 and 4 from salary\_data then we will write the following command:

```
salary_data[ -(2:4), ]
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
5	Neha	Rao	GR1	MUMBAI	19235	15200
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
7	Aaron	Jones	GR1	MUMBAI	23280	13490
8	John	Patil	GR2	MUMBAI	13500	10760
9	Sneha	Joshi	GR1	DELHI	20660	NA
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Anup	Save	GR2	MUMBAI	11960	7880

# Remove Rows from a Data Frame

```
# Remove only rows which has Location as 'MUMBAI'
```

```
salary_data[!(salary_data$Location=="MUMBAI"),]
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
4	Ameet	Mishra	GR2	DELHI	14780	9300
9	Sneha	Joshi	GR1	DELHI	20660	NA
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840

Negation (!) symbol used to remove the row which satisfies the condition.

```
# You can also specify multiple conditions like this:
```

```
salary_data[!(salary_data$Location=="MUMBAI" & salary_data$Grade=="GR2"),]
```

# Derive a New Variable

```
#Add a new variable serial_no to salary_data .
```

```
salary_data$serial_no <- c(1:nrow(salary_data))  
head(salary_data,n=3)
```

Specify new variable name after \$ operator and assign values to it

	First_Name	Last_Name	Grade	Location	basic	allowance	ms	serial_no
1	Alan	Brown	GR1	DELHI	17990	16070		1
2	Agatha	Williams	GR2	MUMBAI	12390	6630		2
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960		3



Variables are always added horizontally in a dataframe. One can use operators like \* for multiplying, + for addition, - for subtraction, and / for division to create new variables.

# Recode a Categorical Variable – ifelse()

One data manipulation task that you need to do in pretty much any data analysis is recode data. It's almost never the case that the data are set up exactly the way you need them for your analysis.

Let's recode Location 'MUMBAI' as 1 and 'DELHI' as 2

```
salary_data$Location<-ifelse(salary_data$Location=="MUMBAI", 1, 2)  
head(salary_data)
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	2	17990	16070
2	Agatha	Williams	GR2	1	12390	6630
3	Rajesh	Kolte	GR1	1	19250	14960
4	Ameet	Mishra	GR2	2	14780	9300
5	Neha	Rao	GR1	1	19235	15200
6	Sagar	Chavan	GR2	1	13390	6700

Location column is recoded using ifelse statement.



Make a note of this syntax. It's great for recoding within R programs

# Recode a Continuous Variable into Categorical Variable – ifelse()

Let's categorise the employees on the basis of their basic allowance(ba) in three categories, namely, Low, Medium and High

```
salary_data$category<-ifelse(salary_data$ba <14000, "Low",
ifelse(salary_data$ba <19000, "Medium", "High"))
head(salary_data)
```

	First_Name	Last_Name	Grade	Location	ba	ms	category
1	Alan	Brown	GR1		2 17990	16070	Medium
2	Agatha	Williams	GR2		1 12390	6630	Low
3	Rajesh	Kolte	GR1		1 19250	14960	High
4	Ameet	Mishra	GR2		2 14780	9300	Medium
5	Neha	Rao	GR1		1 19235	15200	High
6	Sagar	Chavan	GR2		1 13390	6700	Low

Nested **ifelse** statement is used

```
class(salary_data$category)
```

```
[1] "character"
```

Note that column **category** is of character type. You will have to convert it to a factor using **as.factor()**.

# Get an Edge!

- Another way of recoding continuous variable to categorical variable is through `cut()` function. Only difference is you don't need to convert its output into factor as it by default produces output of factor type.
- Previous example using `cut()`:

```
• salary_data$category<-cut(salary_data$ba,breaks=c(0,14000,19000,Inf),labels=c("Low","Medium","High"))
```

- Note that 14000 is assigned to "Low", 19000 to "Medium" and 0 to <NA>
- Cut points giving the number of intervals into which `ba` is to be cut are defined using `breaks=`
- Labels for the levels of the resulting category. are defined using `levels=`

# Quick Recap

In this session, we learnt how to modify data in different ways. Here is the quick recap:

## Change variable names

- `edit()` lets you edit an object and returns the new version
- `fix()` lets you edit an object and modifies the original.
- `names()` is a built-in function which lets you rename the column and modifies the original object

## Change the content of data

- `edit()` and `fix()` lets you change the content of data
- Specify column name after `$` operator and assign `NULL` to it.
- Specify the index in `[]` brackets and assign `NULL` to it

## Remove a column

- Specify new variable name after `$` operator and assign values to it

## Add a new column

- Categorical Variable using `ifelse` statement
- Continuous variable to categorical variable using nested `ifelse` statement

## Recoding

# Creating Subsets & Sorting Data

# Contents

1. Understanding the Need for Creating Subsets
2. Using Index
  - Row Subsetting
  - Column Subsetting
  - Row-Column Subsetting
3. Using subset() Function
  - Subsetting Observations
  - Subsetting Variable
  - Subsetting both Observations & Variables
  - Subsetting using Not Operator

# Contents

## 4. Introduction to Sorting

## 5. Sorting Data

- Ascending Order
- Descending Order
- By Factor Variables
- By Multiple Variables; one column with characters / factors and one with numerals
- By Multiple Variables and Multiple Ordering Levels

# Data Snapshot

basic\_salary data consists salary of each employee with it's Location & Grade.

## Variables

Observations

First_Name	Last_Name	Grade	Location	ba	ms
Alan	Brown	GR1	DELHI	17990	16070
Agatha	Williams	GR2	MUMBAI	12390	6630
Columns	Description	Type	Measurement	Possible values	
First_Name	First Name	character	-	-	
Last_Name	Last Name	character	-	-	
Grade	Grade	character	GR1, GR2	2	
Location	Location	character	DELHI, MUMBAI	2	
ba	Basic Allowance	numeric	Rs.	positive values	
ms	Management Supplements	numeric	Rs.	positive values	



Here we continue to use previous data for our further analysis.



Here we continue to use previous data for our further analysis.

# Need for Creating Subsets

- Sometimes we want to view filtered snippet, or to extract just the data we are interested in from a data frame, for obtaining this we will be using some of R's built-in subsetting functions.
- In terms of R, Subsetting is extracting the needed rows (observations) and columns (variables) from a dataset.
- This tutorial aims to teach the ways in which it is possible to subset data set in R.

# Row Subsetting

- We can use the [] bracket notation to access the indices of rows and columns like this: [R, C]. Here, the first index is for **Rows** and the second is for **Columns**.

```
# Import basic_salary data
```

```
salary_data <- read.csv("basic_salary.csv", header=TRUE)
```

```
# Display rows from 5th to 10th & save it as new object data
```

```
salary1 <- salary_data [5:10, ]  
salary1
```

```
# Output
```

Colon notation(:) used since rows have consecutive positions.

	First_Name	Last_Name	Grade	Location	ba	ms
5	Neha	Rao	GR1	MUMBAI	19235	15200
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
7	Aaron	Jones	GR1	MUMBAI	23280	13490
8	John	Patil	GR2	MUMBAI	13500	10760
9	Sneha	Joshi	GR1	DELHI	20660	NA
10	Gaurav	Singh	GR2	DELHI	13760	13220



When we only want to subset rows we use the first index and leave the second index blank. Leaving an index blank indicates that you want to keep all the elements in that dimension. Use names() to know which column corresponds to which no. in the index.

# Row Subsetting

```
# Display only selected rows & save it as new object data
```

```
salary2 <- salary_data[c(1,3,5), ] ←  
salary2
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
5	Neha	Rao	GR1	MUMBAI	19235	15200

c() can be used to give a list of row indices if the rows are not in sequential order

# Column Subsetting

- As we know, we can subset variables using [] bracket notation but now we use the second index and leave the first index blank. This indicates that we want all the rows for specific columns.

```
# Display columns 1 to 4 & save it as new object data
```

```
salary3<-salary_data[ ,1:4]  
head(salary3)
```

```
# Output
```

	First_Name	Last_Name	Grade	Location
1	Alan	Brown	GR1	DELHI
2	Agatha	Williams	GR2	MUMBAI
3	Rajesh	Kolte	GR1	MUMBAI
4	Ameet	Mishra	GR2	DELHI
5	Neha	Rao	GR1	MUMBAI
6	Sagar	Chavan	GR2	MUMBAI

# Row-Column Subsetting

```
# Display rows 1,5,8 and columns 1 and 2 & save it as new object  
data salary4<-salary_data[c(1,5,8),c(1,2)]  
salary4
```

# Output

	First_Name	Last_Name
1	Alan	Brown
5	Neha	Rao
8	John	Patil

We can also subset the columns by name as in select for subset() like this:

```
salary6<-salary_data[c(1,5,8),c("First_Name", "Last_Name")]
```

# Subsetting Observations

- The **subset()** function with conditions made from logical operators involving the columns of the data frame will let you subset the data frame by observations.
- Our data is saved as an object named **salary\_data** which is of class **data.frame**.

```
# Create a subset with all details of employees of MUMBAI with ba  
more than 15000
```

```
salary5<-subset(salary_data,Location=="MUMBAI" & ba>15000)  
salary5
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
5	Neha	Rao	GR1	MUMBAI	19235	15200
7	Aaron	Jones	GR1	MUMBAI	23280	13490

- **subset()** will return a data frame and so this newly created object can be used as an argument in **subset()**
- **Note :** There is no limit on how many



R assumes variable names are within the data frame being subset, so there is no need to tell R where to look for **location** and **ba** in the above example.

hieve

# Subsetting Observations

- We can also specify which columns we would like **subset()** to return by adding a **select** argument

```
# Create a subset of only Location, First name and ba of salary5  
data, created in the first example
```

```
salary6<-subset(salary5,select=c(Location,First_Name,ba))  
salary6
```

# Output

	Location	First_Name	ba
3	MUMBAI	Rajesh	19250
5	MUMBAI	Neha	19235
7	MUMBAI	Aaron	23280

- ❑ **select=** takes columns to be selected from a data frame.
- ❑ The order of columns in the output totally depends on how you place the column names in **select** argument.

# Subsetting Both Observations and Variables

- We can subset observations and variables by simply combining the previous two methods of subsetting.

```
# Select First_Name, Grade and Location of employees of GR1 with ba  
more than 15000
```

```
salary7<-subset(salary_data, Grade=="GR1" & ba>15000,  
select= c(First_Name, Grade, Location))
```

```
salary7
```

```
# Output
```

	First_Name	Grade	Location
1	Alan	GR1	DELHI
3	Rajesh	GR1	MUMBAI
5	Neha	GR1	MUMBAI
7	Aaron	GR1	MUMBAI
9	Sneha	GR1	DELHI

# Subsetting Using Not Operator

Suppose we want all the details of employees not having GR1 and not from MUMBAI, we will write the following command.

```
salary8<-subset(salary_data,! (Grade=="GR1") & !(Location=="MUMBAI"))  
salary8
```

# Output

	First_Name	Last_Name	Grade	Location	ba	ms
4	Ameet	Mishra	GR2	DELHI	14780	9300
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840

**Not Equal To (!)**  
operator is used to  
give condition.

# Quick Recap

In this session, we learnt many ways to subset data using **subset()** and row/column index. Here is the quick recap of how we can create subsets :

## Using index

- **Row Subsetting:** By specifying the row indices using `[]` notation
- **Column Subsetting:** By specifying the column indices using `[]` notation or column names using `c()`.
- **Row-Column Subsetting:** By combining the above two methods.

## Using **subset()**

- **Subsetting observations:** By giving conditions on columns using this function.
- **Subsetting variables:** By specifying columns in **select** argument.
- **Subsetting both observations and variables:** By simply combining above two methods.
- **Subsetting using Not Operator :** By giving conditions on those columns which we do not want using this function.

# Introduction to Sorting

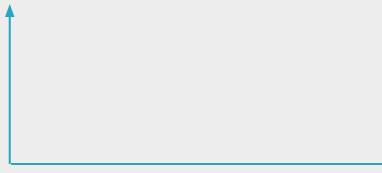
Sorting data is one of the common activity in preparing data for analysis.

Sorting is storage of data in sorted order, it can be in ascending or descending order.

We will be exploring all the ways in which sorting can be done.

```
# Import and attach basic_salary data
```

```
salary_data <- read.csv("basic_salary.csv",header=TRUE)  
attach(salary_data)
```



**attach()** attaches the database to the R search path, so the variables in the database can be accessed by simply giving their names

# Ascending Data

```
# Sort salary_data by ba in Ascending order
```

```
ba_sorted_1<-salary_data[order(ba),]  
ba_sorted_1
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
12	Anup	Save	GR2	MUMBAI	11960	7880
2	Agatha	Williams	GR2	MUMBAI	12390	6630
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
8	John	Patil	GR2	MUMBAI	13500	10760
11	Adela	Thomas	GR2	DELHI	13660	6840
10	Gaurav	Singh	GR2	DELHI	13760	13220
4	Ameet	Mishra	GR2	DELHI	14780	9300
1	Alan	Brown	GR1	DELHI	17990	16070
5	Neha	Rao	GR1	MUMBAI	19235	15200
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
9	Sneha	Joshi	GR1	DELHI	20660	NA
7	Aaron	Jones	GR1	MUMBAI	23280	13490

**order()** is used to sort a vector, matrix or data frame. By default, it sorts in ascending order

# Descending Order

```
# Sort salary_data by ba in Descending order
```

```
ba_sorted_2<-salary_data[order(-ba),]  
ba_sorted_2
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
7	Aaron	Jones	GR1	MUMBAI	23280	13490
9	Sneha	Joshi	GR1	DELHI	20660	NA
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
5	Neha	Rao	GR1	MUMBAI	19235	15200
1	Alan	Brown	GR1	DELHI	17990	16070
4	Ameet	Mishra	GR2	DELHI	14780	9300
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840
8	John	Patil	GR2	MUMBAI	13500	10760
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
2	Agatha	Williams	GR2	MUMBAI	12390	6630
12	Anup	Save	GR2	MUMBAI	11960	7880

The '-' sign before a numeric column reverses the default order.

Alternatively , you can also use decreasing=TRUE

```
ba_sorted_2<-salary_data[order(ba, decreasing = TRUE),]
```

# Sorting by Factor Variable

Sort data by column with characters / factors

```
# Sort salary_data by Grade
```

```
gr_sorted<-salary_data[order(Grade),]  
gr_sorted
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
5	Neha	Rao	GR1	MUMBAI	19235	15200
7	Aaron	Jones	GR1	MUMBAI	23280	13490
9	Sneha	Joshi	GR1	DELHI	20660	NA
2	Agatha	Williams	GR2	MUMBAI	12390	6630
4	Ameet	Mishra	GR2	DELHI	14780	9300
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
8	John	Patil	GR2	MUMBAI	13500	10760
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Anup	Save	GR2	MUMBAI	11960	7880

Note that by default  
`order()` sorts in ascending  
order

# Sorting by Factor Variable

Sort data by column with characters / factors in Descending order

```
# Sort salary_data by Grade in Descending order
```

```
gr_sorted<-salary_data[order(Grade,decreasing=TRUE),]  
gr_sorted
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
2	Agatha	Williams	GR2	MUMBAI	12390	6630
4	Ameet	Mishra	GR2	DELHI	14780	9300
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
8	John	Patil	GR2	MUMBAI	13500	10760
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Anup	Save	GR2	MUMBAI	11960	7880
1	Alan	Brown	GR1	DELHI	17990	16070
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
5	Neha	Rao	GR1	MUMBAI	19235	15200
7	Aaron	Jones	GR1	MUMBAI	23280	13490
9	Sneha	Joshi	GR1	DELHI	20660	NA

In case of **factor type variables**, if the ordering is to be made descending, then the logical argument of **decreasing= TRUE** needs to be included.

# Sorting Data by Multiple Variables

- Sort data by giving multiple columns; one column with characters / factors and one with numerals

```
# Sort salary_data by Grade and ba
```

```
grba_sorted<-salary_data[order(Grade,ba),]  
grba_sorted
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
5	Neha	Rao	GR1	MUMBAI	19235	15200
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
9	Sneha	Joshi	GR1	DELHI	20660	NA
7	Aaron	Jones	GR1	MUMBAI	23280	13490
12	Anup	Save	GR2	MUMBAI	11960	7880
2	Agatha	Williams	GR2	MUMBAI	12390	6630
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
8	John	Patil	GR2	MUMBAI	13500	10760
11	Adela	Thomas	GR2	DELHI	13660	6840
10	Gaurav	Singh	GR2	DELHI	13760	13220
4	Ameet	Mishra	GR2	DELHI	14780	9300

Here, data is first sorted in increasing order of **Grade** then **ba**.

# Multiple Variables & Multiple Ordering Levels

- Sort data by giving multiple columns; one column with characters / factors and one with numerals and multiple ordering levels

```
# Sort salary_data by Grade in Descending order and then by ms in  
# Ascending order
```

```
grba_sorted<-salary_data[order(Grade,decreasing=TRUE,ms),] ←  
grba_sorted
```

# Output

	First_Name	Last_Name	Grade	Location	ba	ms
10	Gaurav	Singh	GR2	DELHI	13760	13220
8	John	Patil	GR2	MUMBAI	13500	10760
4	Ameet	Mishra	GR2	DELHI	14780	9300
12	Anup	Save	GR2	MUMBAI	11960	7880
11	Adela	Thomas	GR2	DELHI	13660	6840
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
2	Agatha	Williams	GR2	MUMBAI	12390	6630
1	Alan	Brown	GR1	DELHI	17990	16070
5	Neha	Rao	GR1	MUMBAI	19235	15200
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
7	Aaron	Jones	GR1	MUMBAI	23280	13490
9	Sneha	Joshi	GR1	DELHI	20660	NA

- Here, data is sorted by **Grade** and **ms** in decreasing order.
- By default missing values in data are put last.
- You can put it first by adding an argument **na.last=FALSE** in **order()**.

# Quick Recap

In this session, we learnt sorting data using **order()** in various ways. Here is a quick recap

## Ascending/ Descending order

- **order()** by default sorts in ascending order.
- For descending order: specify **decreasing=TRUE** for factor type variable and put '-' sign before numeric type variable

## Multiple Columns

- **order()** allows us to sort by multiple columns of different type

## Multiple columns and multiple ordering levels

- **order()** provides flexibility to order by multiple columns with different ordering levels

# Merging / Appending & Aggregating Data

# Contents

1. Introduction to Merging
2. Types of Joins
3. Understanding the types of joins with examples
4. Introduction to Package dplyr
5. Understanding types of joins offered by dplyr
6. Appending Data Sets
7. Introduction to Aggregation
8. Aggregating Data Using
  - aggregate() Function
  - Package dplyr
  - Package data.table

# Data Snapshot

sal\_data consist information about Employee's Basic Salary, their ID & full Name

→

Employee_ID	First_Name	Last_Name	Basic_Salary
E-1001	Mahesh	Joshi	16860
E-1002	Raj	Prabhu	
E-1004	Priya		
E-1005	Sneha	Employee_ID	Employee ID
E-1007	Ram	First_Name	First Name
E-1008	Nisha	Last_Name	Last Name
E-1009	Hari	Basic_Salary	Basic Salary

	Columns	Description	Type	Measurement	Possible values
E-1005	Employee_ID	Employee ID	character	-	-
E-1007	First_Name	First Name	character	-	-
E-1008	Last_Name	Last Name	character	-	-
E-1009	Basic_Salary	Basic Salary	numeric	Rs.	positive values

bonus\_data has information of only Bonus given to Employees.

→

Employee_ID	Bonus
E-1001	16070
E-1003	
E-1004	
E-1006	
E-1008	
E-1010	

	Columns	Description	Type	Measurement	Possible values
E-1006	Employee_ID	Employee ID	character	-	-
E-1010	Bonus	Bonus amount	numeric	Rs.	Positive values

"Employee ID" is the common column in both datasets

# Merging

Join (Merge) two datasets having one or more common columns with identical names.

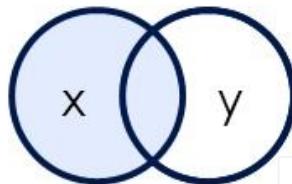
**merge()** function is used to join two data sets. To perform join operations import the data sets.

```
# Import sal_data and bonus_data  
sal_data <- read.csv("sal_data.csv",header=TRUE)  
  
bonus_data <- read.csv("bonus_data.csv",header=TRUE)
```

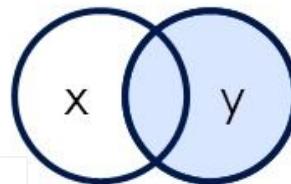
# Types of Joins

Consider  $\text{sal\_data} = x$  and  $\text{bonus\_data} = y$

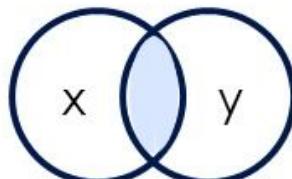
Left Join



Right Join

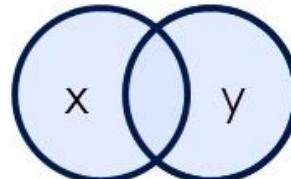


Inner Join



There are  
4 types of  
Joins

Outer Join



# Left Join

Left Join returns all rows from the left table, and rows with matching keys from the right table.

```
# Display all the information(including bonus) of Employees from sal_data  
leftjoin<-merge(sal_data,bonus_data,by=c("Employee_ID"),all.x=TRUE)  
leftjoin  
  
# Output
```

Emp	Tooyee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1002	Rajesh	Kolte	14960	NA
3	E-1004	Priya	Jain	12670	13490
4	E-1005	Sneha	Joshi	15660	NA
5	E-1007	Ram	Kanade	15850	NA
6	E-1008	Nishi	Honrao	15950	15880
7	E-1009	Hameed	Singh	15120	NA

- ❑ **all.x= true** is specified to include all rows from the data set **x**(i.e. first data frame) and only those from **y**(i.e. second dataframe) that match
- ❑ **by=** is used to specify common columns.

# Right Join

Right Join returns all rows from the right table, and any rows with matching keys from the left table.

```
# Display all the information of employees who are receiving bonus
```

```
rightjoin<-merge(sal_data,bonus_data,by="Employee_ID",all.y=TRUE)  
rightjoin
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1004	Priya	Jain	12670	13490
3	E-1008	Nishi	Honrao	15950	15880
4	E-1003	<NA>	<NA>	NA	15200
5	E-1006	<NA>	<NA>	NA	14200
6	E-1010	<NA>	<NA>	NA	15120

**all.y= true** is specified to keep all rows from the data set **y** and only those from **x** that match

# Inner Join

Inner Join returns only the rows in which the x have matching keys in the y.

```
# Display all the information about employees which are common in both  
# the tables
```

```
innerjoin<-merge(sal_data,bonus_data,by=("Employee_ID")) ←  
innerjoin
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1004	Priya	Jain	12670	13490
3	E-1008	Nishi	Honrao	15950	15880

By default **all.x** and **all.y** is  
equal to **FALSE**

# Outer Join

Outer Join returns all rows from x and y, join records from x which have matching keys in the y.

```
# Combine sal_data and bonus_data
```

```
outerjoin<-merge(sal_data,bonus_data,by=c("Employee_ID"),all=TRUE)  
outerjoin
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1002	Rajesh	Kolte	14960	NA
3	E-1004	Priya	Jain	12670	13490
4	E-1005	Sneha	Joshi	15660	NA
5	E-1007	Ram	Kanade	15850	NA
6	E-1008	Nishi	Honrao	15950	15880
7	E-1009	Hameed	Singh	15120	NA
8	E-1003	<NA>	<NA>	NA	15200
9	E-1006	<NA>	<NA>	NA	14200
10	E-1010	<NA>	<NA>	NA	15120

all= true is specified to include all rows from both data sets.

# Package dplyr

- Using `merge()` in R on big tables can be time consuming. The join functions in the package **dplyr** are much faster.
- **dplyr** is the next iteration of package **plyr**, developed by Hadley Wickham and Romain Francois, focused on tools for working with data frames (hence the **d** in the name).
- It is built to be fast, highly expressive, and open-minded about how your data is stored.
- The package offers six different joins:
  - inner join
  - left join
  - right join
  - full join
  - semi join
  - anti join

# inner\_join()

- **inner\_join()**: similar to **merge()** with **all.x=F** and **all.y=F**.
- It returns all rows from x where there are matching values in y, and all columns from x and y.

```
install.packages("dplyr")
library(dplyr)
```

```
inner_join(sal_data,bonus_data,by="Employee_ID")
```

# Output

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1004	Priya	Jain	12670	13490
3	E-1008	Nishi	Honrao	15950	15880

# left\_join()

- **left\_join()**: similar to **merge()** with **all.x=T**
  - It returns all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns.

```
left_join(sal_data,bonus_data,by="Employee_ID")
```

# Output

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1002	Rajesh	Kolte	14960	NA
3	E-1004	Priya	Jain	12670	13490
4	E-1005	Sneha	Joshi	15660	NA
5	E-1007	Ram	Kanade	15850	NA
6	E-1008	Nishi	Honrao	15950	15880
7	E-1009	Hameed	Singh	15120	NA

# right\_join()

- **right\_join**: similar to **merge()** with **all.y=TRUE**, only difference is that it sorts the result by the common column.
- It returns all rows from y, and all columns from x and y. Rows in y with no match in x will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

```
right_join(sal_data,bonus_data,by="Employee_ID")
```

# Output

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1003	<NA>	<NA>	NA	15200
3	E-1004	Priya	Jain	12670	13490
4	E-1006	<NA>	<NA>	NA	14200
5	E-1008	Nishi	Honrao	15950	15880
6	E-1010	<NA>	<NA>	NA	15120
.	.	.	.	.	.

# full\_join()

- **full\_join**: similar to `merge()` with `all=TRUE`.
- It returns all rows and all columns from both x and y. where there are not matching values, returns NA for the one missing.

```
full_join(sal_data,bonus_data,by="Employee_ID") ←—————
```

# Output

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1002	Rajesh	Kolte	14960	NA
3	E-1004	Priya	Jain	12670	13490
4	E-1005	Sneha	Joshi	15660	NA
5	E-1007	Ram	Kanade	15850	NA
6	E-1008	Nishi	Honrao	15950	15880
7	E-1009	Hameed	Singh	15120	NA
8	E-1003	<NA>	<NA>	NA	15200
9	E-1006	<NA>	<NA>	NA	14200
10	E-1010	<NA>	<NA>	NA	15120

data is merged  
keeping the  
original order  
of x

# semi\_join

- **semi\_join**: return all rows from x where there are matching values in y, keeping just columns from x.

```
# Keep the sal_data in the same format, but only keep the records that  
# also have a match in the bonus_data
```

```
semi_join(sal_data,bonus_data,by="Employee_ID")
```

# Output

	Employee_ID	First_Name	Last_Name	Basic_Salary
1	E-1001	Mahesh	Joshi	16860
2	E-1004	Priya	Jain	12670
3	E-1008	Nishi	Honrao	15950



With **semi\_join()** we get a similar result as with **inner\_join()** but the semi\_join result contains only the rows originally found in x

## anti\_join()

- **anti\_join**: returns all rows from x where there are not matching values in y, keeping just columns from x.

```
# Display the records of employees who are not receiving bonus
```

```
anti_join(sal_data,bonus_data,by="Employee_ID")
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary
1	E-1002	Rajesh	Kolte	14960
2	E-1005	Sneha	Joshi	15660
3	E-1007	Ram	Kanade	15850
4	E-1009	Hameed	Singh	15120

# Appending Data- Data Snapshot

basic\_salary data consist salary of each employee with it's Location & Grade.

**Variables**

Columns	Description	Type	Measurement	Possible values
First_Name	First Name	character	-	-
Last_Name	Last Name	character	-	-
Grade	Grade	character	GR1, GR2	2
Location	Location	character	DELHI, MUMBAI	2
ba	Basic Allowance	numeric	Rs.	positive values
ms	Management Supplements	numeric	Rs.	positive values

**Observations**

# Appending

- Append means adding cases/observations to a dataset.
- Appending two datasets using `rbind()` function requires both the datasets with exactly the same number of variables with exactly the same names.
- If datasets do not have the same number of variables, variables can be either dropped or created so both match.

# Appending Data Sets

```
# Import the data sets and append them using rbind() function
```

```
salary_1<-read.csv("basic_salary - 1.csv",header=TRUE)
salary_2<-read.csv("basic_salary - 2.csv",header=TRUE)
rbindsalary<-rbind(salary_1,salary_2)
rbindsalary
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
2	Agatha	Williams	GR2	MUMBAI	12390	6630
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
4	Ameet	Mishra	GR2	DELHI	14780	9300
5	Neha	Rao	GR1	MUMBAI	19235	15200
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
7	Aaron	Jones	GR1	MUMBAI	23280	13490
8	John	Patil	GR2	MUMBAI	13500	10760
9	Sneha	Joshi	GR1	DELHI	20660	NA
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Anup	Save	GR2	MUMBAI	11960	7880

rbind() combines the vector, matrix or data frame by rows

# Quick Recap

In this session, we learnt different ways of joining two data sets using `merge()` and package dplyr. Here is the quick recap:

Base function in R :  
`merge()`

4 types of joins:

- `left_join`
- `right join`
- `inner join`
- `outer join`

package `dplyr`-  
recommended in case  
of large data as it is  
much faster than  
`merge()`

6 types of joins:

- `inner_join()`
- `left_join()`
- `right_join()`
- `full_join()`
- `semi_join()`
- `anti_join()`

Appending Data

- `rbind()` combines the vector, matrix or data frame by rows

# Aggregating Data - Data Snapshot

basic\_salary data consist salary of each employee with it's Location & Grade.

## Variables

Observations

First_Name	Last_Name	Grade	Location	ba	ms
Alan	Brown	GR1	DELHI	17990	16070
Columns	Description		Type	Measurement	Possible values
First_Name	First Name	character	-	-	-
Last_Name	Last Name	character	-	-	-
Grade	Grade	character	GR1, GR2	2	
Location	Location	character	DELHI, MUMBAI	2	
ba	Basic Allowance	numeric	Rs.	positive values	
ms	Management Supplements	numeric	Rs.	positive values	

# Introduction to Aggregation

Aggregating data means splitting data into subsets, computing summary statistics on each subset and displaying the results in a conveniently summarised form.

Suppose a database has millions of rows. It can be aggregated and summarised on various levels depending on the type of information needed.

Aggregation is necessary to manage complexity of data.

In R, we have a built-in function **aggregate()** which carries out the process of taking numerous records and collapsing them into a single summary record.

```
# Import basic_sal data  
salary_data<-read.csv("basic_salary.csv",header=TRUE)
```

# aggregate()

```
# Calculate sum of variable 'ms' by variable 'Location'  
# In this example we are giving one variable and one factor
```

```
A<-aggregate(ms~Location,data=salary_data,FUN=sum)  
A
```

```
# Output
```

Location	ms
DELHI	45430
MUMBAI	75620

- $\sim$  is used to give a formula such as  $y \sim x$ , where  $y$  are the numeric type variables to be split into groups according to the grouping  $x$  variables.
- FUN=** argument takes the function to compute summary statistics.

```
# The above command can also be written as:
```

```
aggregate(salary_data$ms, by=list(salary_data$Location), FUN=sum) ←
```

**by=** takes a list of variables by which grouping is to be done



aggregate() by default ignores the missing data values.

# aggregate()

```
# Calculate sum of variable 'ms' by variables 'Location' and 'Grade'  
# In this example we are giving one variable and two factors
```

```
C<-aggregate(ms~Location+Grade,data=salary_data,FUN=sum)  
C
```

# Output

	Location	Grade	ms
1	DELHI	GR1	16070
2	MUMBAI	GR1	43650
3	DELHI	GR2	29360
4	MUMBAI	GR2	31970

Multiple factors can be added using  
‘+’ operator.

```
# Calculate sum of variables 'ba' and 'ms' by variables 'Location' and 'Grade'  
# In this example we are giving multiple variables and one factor
```

```
B<-aggregate(cbind(ba,ms)~Location,data=salary_data,FUN=sum)  
B
```

# Output

	Location	ba	ms
1	DELHI	60190	45430
2	MUMBAI	113005	75620

**cbind()** combines vector, matrix or data frame by columns



When aggregated with cbind(), it ignores the rows corresponding to the row having missing values. So, we recommend not to use cbind()(in case of missing values) with aggregate(), you can aggregate these variables separately.

# Aggregating using Package dplyr

- The package dplyr provides a well structured set of functions for manipulating data and performing typical operations with standard, easy to remember syntax. It is also very fast, even with large collections
- Major strength of dplyr is the ability to group the data by a variable or variables and then operate on the data "by group". It uses "split-apply-combine" paradigm approach i.e.split the data into groups, apply some analysis to each group, and then combine the results with the help of **group\_by()** and **summarize()**.

```
# Install and load the package dplyr  
install.packages("dplyr")  
library(dplyr)
```

# Aggregating using Package dplyr

```
# Calculate sum for variable 'ms' by variable 'Location'
```

```
loc<-group_by(salary_data,Location)
dplyr_agg<-summarize(loc,sum=sum(ms,na.rm=TRUE))
dplyr_agg
```

# Output

```
# A tibble: 2 x 2
  Location     sum
  <fct>    <int>
1 DELHI      45430
2 MUMBAI     75620
```

- group\_by()** splits the data into groups upon which some operations can be run. Multiple factors can be specified  
eg.: **group\_by(salary\_data, Location, Grade)**
- summarize()** which collapses each group into a single-row summary of that group. **You can also summarise multiple variables at the same time.**
- sum** is the name of the new column with summarised data which is specified as an argument **sum=**
- na.rm=TRUE** ignores the missing values. If it is not specified it gives **NA** instead of 45430 because by default **na.rm=False**.

# Aggregating using Package data.table

- In case of large data sets, a data.frame can be limiting, the time it takes to do certain things is just too long.
- The package data.table solves this for you by reducing computing time.
- This package provide many features like fast aggregation, fast ordered joins, fast data manipulation, fast file import, etc.
- It is much faster than R's built-in function **aggregate()**.

```
# Install and load package data.table  
install.packages("data.table")  
library(data.table)
```

# Aggregating using Package data.table

```
# Calculate sum for variable 'ms' by variable 'Location'  
# First convert dataframe salary_data into a datatable using data.table()
```

```
salary_data_DT<-data.table(salary_data)  
setkey(salary_data_DT,Location)  
DT_agg<-salary_data_DT[,.(sumofms=sum(ms,na.rm=TRUE)),by=Location]  
DT_agg
```

# Output

	Location	sumofms
1:	DELHI	45430
2:	MUMBAI	75620

- ❑ **setkey()** sorts the rows of a datatable by the columns provided by reference, always in ascending order. It speeds up operations on keyed datatables.
- ❑ **by=** takes the factors by which data will be aggregated. Multiple factors can be specified with **list()**.
- ❑ **na.rm=TRUE** ignores the missing values. It works the same way like in **dplyr**.
- ❑ **.()** notation allows you to rename the columns inside datatable.



Both **data.table** and **dplyr** are very efficient when it comes to large data. If you're looking for pure speed, **data.table** is the clear winner. However, **dplyr**'s syntax is easier than **data.table**'s.

# Quick Recap

In this session, we learnt different methods of aggregating data.

Here is the quick recap:

## aggregate()

- It is a built-in function of R, which aggregates the inputted **data.frame** by applying a function specified by the **FUN** argument to the column defined before **~** by the columns defined after **~**

## Package dplyr

- It provides two functions **groupby()** and **summarise()** for aggregation of data

## Package data.table

- It provides fast and more efficient aggregation in case of large data sets.
- **data.frame** need to be converted to **data.table**

Package dplyr

Data Manipulation and Summarization

# Contents

1. Introduction to dplyr
2. dplyr verbs:
  - filter()
  - select()
  - arrange()
  - mutate()
  - summarise()
  - group\_by()
3. Drawing Random Numbers
4. Merging Data
5. Pipe Operator
6. More Functions from dplyr

# Introduction

- dplyr is a powerful package to transform and summarize a data frame or data frame like object.
- The package contains a set of functions that are very handy and easy when performing exploratory data analysis and data manipulation.
- dplyr is part of core tidyverse.
- dplyr is developed by Hadley Wickham.
- dplyr is known as grammar of data management and functions in this package as verbs.

# Data Snapshot

basic\_salary data consist salary of each employee with it's Location & Grade.

**Variables**

First_Name	Last_Name	Grade	Location	ba	ms
Alan	Brown	GR1	DELHI	17990	16070
Columns	Description	Type	Measurement	Possible values	
First_Name	First Name	character	-	-	
Last_Name	Last Name	character	-	-	
Grade	Grade	character	GR1, GR2	2	
Location	Location	character	DELHI, MUMBAI	2	
ba	Basic Allowance	numeric	Rs.	positive values	
ms	Management Supplements	numeric	Rs.	positive values	

**Observations**

# Creating Subset - filter()

- `filter()` allows us to select a subset of rows in the data frame.
- Multiple conditions may be applied to the data frame to obtain the desired subset.

```
# Install and load package dplyr  
# Import basic_salary data  
  
install.packages("dplyr")  
library(dplyr)  
  
data<-read.csv("basic_salary.csv",header=TRUE)
```

# Creating Subset - filter()

```
# To create a subset of GR1 employees in MUMBAI
```

```
head(filter(data,Grade=="GR1",Location=="MUMBAI"))
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Rajesh	Kolte	GR1	MUMBAI	19250	14960
2	Neha	Rao	GR1	MUMBAI	19235	15200
3	Aaron	Jones	GR1	MUMBAI	23280	13490

filter() creates a subset of the dataframe with rows that meet the required specifications

# Creating Subset - select()

- We often work with large data frames with many variables. Sometimes it may happen that only a few variables turn out to be useful.
- `select ()` helps us get subset of only the useful variables.

```
# To create a subset having variables First_Name,Grade,Location
```

```
s1<-select(data,First_Name,Grade,Location) ←  
head(s1)
```

```
# Output
```

	First_Name	Grade	Location
1	Alan	GR1	DELHI
2	Agatha	GR2	MUMBAI
3	Rajesh	GR1	MUMBAI
4	Ameet	GR2	DELHI
5	Neha	GR1	MUMBAI
6	Sagar	GR2	MUMBAI

Only the variables in the list will  
be selected

# Creating Subset - select()

```
# To create a subset of some columns together
```

```
s2<-select(data,First_Name:Grade)  
head(s2)
```

```
# Output
```

	First_Name	Last_Name	Grade
1	Alan	Brown	GR1
2	Agatha	Williams	GR2
3	Rajesh	Kolte	GR1
4	Ameet	Mishra	GR2
5	Neha	Rao	GR1
6	Sagar	Chavan	GR2

To select all columns between two columns **semicolon (:)**  can be used. This command will select all columns between **First\_Name** and **Grade** (Both inclusive)

# Creating Subset - select()

```
# To create a subset with not having some specified columns
```

```
s3<-select(data,-(Grade:ba))  
head(s3)
```

```
# Output
```

	First_Name	Last_Name	ms
1	Alan	Brown	16070
2	Agatha	Williams	6630
3	Rajesh	Kolte	14960
4	Ameet	Mishra	9300
5	Neha	Rao	15200
6	Sagar	Chavan	6700

All variables except those between **Grade** and **ba** (both inclusive) are selected.

# Sorting Data - arrange()

- `arrange()` is used to reorder rows.
- assign the column names in the data frame that are to be sorted.

```
# Sort ba in ascending order
```

```
a1<-arrange(data,ba) ←  
head(a1)
```

Use `desc()` to order a column in descending order.

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Anup	Save	GR2	MUMBAI	11960	7880
2	Agatha	Williams	GR2	MUMBAI	12390	6630
3	Sagar	Chavan	GR2	MUMBAI	13390	6700
4	John	Patil	GR2	MUMBAI	13500	10760
5	Adela	Thomas	GR2	DELHI	13660	6840
6	Gaurav	Singh	GR2	DELHI	13760	13220

# Sorting Data - arrange()

```
# Sort by multiple variables
```

```
a2<-arrange(data,Grade,Location,ba)  
head(a2)
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
2	Sneha	Joshi	GR1	DELHI	20660	NA
3	Neha	Rao	GR1	MUMBAI	19235	15200
4	Rajesh	Kolte	GR1	MUMBAI	19250	14960
5	Aaron	Jones	GR1	MUMBAI	23280	13490
6	Adela	Thomas	GR2	DELHI	13660	6840

Here data is sorted by Grade. Within a particular Grade, the data is sorted by Location. Within a particular Grade and Location, the data is sorted by ba. All sorting is done in ascending order.

# Modifying Data - mutate()

- `mutate()` is used to add new columns to the data frame.
- These new columns are functions of the existing columns.

```
# To create a new column
```

```
m1<-mutate(data, tot=ba+ms)  
head(m1)
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms	tot
1	Alan	Brown	GR1	DELHI	17990	16070	34060
2	Agatha	williams	GR2	MUMBAI	12390	6630	19020
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960	34210
4	Ameet	Mishra	GR2	DELHI	14780	9300	24080
5	Neha	Rao	GR1	MUMBAI	19235	15200	34435
6	Sagar	Chavan	GR2	MUMBAI	13390	6700	20090

If we want only the new column, use `transmute()`.

# Summarizing Data - summarise()

`summarise()` function will give the summary statistics for a given column in the data frame.

```
# To get summary statistic of a variable
```

```
summarise(data,meanba=mean(ba,na.rm=TRUE),  
          medianba=median(ba,na.rm=TRUE))
```

```
# Output
```

	meanba	medianba
1	16154.58	14270

Some of the summary statistics could be `mean`, `median`, `max`, `min`, `sd`, `sum` .

# Grouping Data - group\_by()

- **group\_by()** verb splits the data frame by assigned variables
- We can now apply functions to these individual groups

```
# To create group  
# Get summary statistic
```

```
loc_wise<-group_by(data,Location)  
summarise(loc_wise,count=n(),mean=mean(ba,na.rm=TRUE))
```

```
# Output
```

```
# A tibble: 2 x 3  
  Location count   mean  
  <fct>     <int>   <dbl>  
1 DELHI       5 16170  
2 MUMBAI      7 16144.
```

Here, first use **group\_by()** to get the distinct locations then use the function **summarise()** to get the count and meanba in the distinct locations



Incase your summarise code gives an error, just execute the following :  
`dplyr::summarise(loc_wise,count = n(),mean=mean(ba,na.rm=TRUE))`

# Drawing Random Samples

- `sample_n()` and `sample_frac()` are used to draw random samples of rows from the rows of data frame
- `sample_n()` gives a fixed number of rows
- `sample_frac()` gives a fixed fraction

```
# To draw a random sample of 5 rows
```

```
data_5<-sample_n(data,5)  
head(data_5)
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Neha	Rao	GR1	MUMBAI	19235	15200
2	Alan	Brown	GR1	DELHI	17990	16070
3	Sagar	Chavan	GR2	MUMBAI	13390	6700
4	Sneha	Joshi	GR1	DELHI	20660	NA
5	John	Patil	GR2	MUMBAI	13500	10760

# Drawing Random Samples

```
# To draw a random sample of size 10% (0.10) of data
```

```
data_0.1<-sample_frac(data,0.1)  
data_0.1
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Aaron	Jones	GR1	MUMBAI	23280	13490

# Merging Data – join()

- `join()` in dplyr will work on data frames, tibbles and tbl references.
- The different operations could be –
  - `left_join()`: All rows from first data are included
  - `right_join()`: All rows from second data are included
  - `inner_join()`: Included rows which are matched on common variable
  - `full_join()`: All rows from two data are included with appropriate NA's



We have already studied these functions in detail in DM 04 - Merging, Appending \_ Aggregating Data

# Pipe Operator

- dplyr imports Pipe operator (%>%) from another package (magrittr)
- Its performance is similar to that of nested function.
- This operator allows the output of one function to be taken as input for another function.
- The chain of functions are read from left to right.

```
# Syntax for pipe operator to get first 6 rows of selected variables
```

```
data %>%  
  select(First_Name,Grade,Function) %>%  
  head
```

```
# Output
```

	First_Name	Grade	Location
1	Alan	GR1	DELHI
2	Agatha	GR2	MUMBAI
3	Rajesh	GR1	MUMBAI
4	Ameet	GR2	DELHI
5	Neha	GR1	MUMBAI
6	Sagar	GR2	MUMBAI

# Pipe Operator

```
# Syntax for pipe operator to get data of selected variables belonging  
to MUMBAI
```

```
data %>%  
  select(First_Name,Grade,Location) %>%  
  filter(Location=="MUMBAI")
```

```
# Output
```

	First_Name	Grade	Location
1	Agatha	GR2	MUMBAI
2	Rajesh	GR1	MUMBAI
3	Neha	GR1	MUMBAI
4	Sagar	GR2	MUMBAI
5	Aaron	GR1	MUMBAI
6	John	GR2	MUMBAI
7	Anup	GR2	MUMBAI

# Some More Functions-n\_distinct()

- `n_distinct()` is used to count the number of unique values in a set of vectors.
- It is much faster than using `length(unique())`.

```
# To get the unique length
```

```
n_distinct(data)
```

```
[1] 41
```

```
n_distinct(data$Grade,na.rm=TRUE) ←
```

```
[1] 2
```

If we do not want to include **NA**, we use `na.rm=TRUE`

# Some More Functions-distinct()

- **distinct()** retains only unique/distinct rows from an input tbl.
- This is similar to **unique.data.frame()**, but considerably faster.

```
# To get the unique categories in Grade  
distinct(data,Grade)
```

# Output

	Grade
1	GR1
2	GR2

# Some More Functions-dense\_rank()

- Sometimes we may want not only to rank the dataset based on a variable but also display the rank along with the dataset.
- `dense_rank()` gives the ranking with no gaps between ranks.

```
# To sort on basis of a variable and to display rank alongwith
```

```
data %>%
  mutate(Rank=dense_rank(desc(ba)))%>%
  arrange(Rank) %>%
  head()
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms	Rank
1	Aaron	Jones	GR1	MUMBAI	23280	13490	1
2	Sneha	Joshi	GR1	DELHI	20660	NA	2
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960	3
4	Neha	Rao	GR1	MUMBAI	19235	15200	4
5	Alan	Brown	GR1	DELHI	17990	16070	5
6	Ameet	Mishra	GR2	DELHI	14780	9300	6

Other than `dense_rank()`, the following functions also provide great help : `row_number()`, `min_rank()`, `ntile()`, `percent_rank()`, `cume_dist()`

# Some More Functions-rowwise()

- Standard data frame operations in R are done column-wise rather than row-wise.
- `rowwise()` function is used when we want to perform row-wise operations (For instance, calculating row mean).

```
# To get the row means
```

```
data %>%
  rowwise()%>%
  mutate(meanval=mean(c(ba,ms))) %>%
  head(3)
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms	meanval
1	Alan	Brown	GR1	DELHI	17990	16070	17030
2	Agatha	Williams	GR2	MUMBAI	12390	6630	9510
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960	17105

# Some More Functions-select\_if()

- `select_if()` drops variables that are not in the selection list
- This function is a variant of `select()`.

```
# To select specific variables
```

```
data %>%  
  select_if(is.numeric) %>%  
  head()
```

```
# Output
```

	ba	ms
1	17990	16070
2	12390	6630
3	19250	14960
4	14780	9300
5	19235	15200
6	13390	6700

# Some More Functions-`rename_if()`

- `rename_if()` changes the column names of only the variables in the selected list.
- This function is a variant of `rename()`.

```
# To rename the specified variables
```

```
data %>%
  rename_if(is.numeric,toupper) %>%
  head(3)
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	BA	MS
1	Alan	Brown	GR1	DELHI	17990	16070
2	Agatha	Williams	GR2	MUMBAI	12390	6630
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960

Columns which are of type numeric, their names are changed to upper case



`na.if()` is also a useful function that can convert particular values from the data to NAs. `na_if()` is a translation of the SQL command `NULL_IF`. For instance, sometimes missing values are assigned values 999 or 0 and it may be required to recode them as NA.

# Quick Recap

In this session, we learnt about package dplyr which is known as the Grammar of Data Management. This package contains a set of functions that are very handy and easy when performing exploratory data analysis and data manipulation

## Imp verbs in dplyr package

- filter(), select() is used for creating subsets.
- arrange() for sorting data
- mutate() for modifying data
- summarise() for summarising data.
- group\_by() splits the data frame by assigned variables

## Drawing Random Sample

- sample\_n(), sample\_fact() are functions used for drawing random numbers

## Merging Data

- left\_join(), right\_join(), inner\_join(), full\_join()

## Pipe Operator

- dplyr imports (`%>%`) this operator from another package (magrittr). Its performance is comparable to nested function

## Miscellaneous Functions

- n\_distinct(), distinct(), tally(), dense\_rank(), row\_wise(), select\_if(), rename\_if() are some more functions which are easy to understand and use for data manipulation.

# Package data.table

# Contents

1. Working on Large Data
2. Introduction to package data.table
3. Importing Data
4. Creating Data
5. Merging Data
6. Creating Subsets
7. Sorting Data
8. Modifying Data
9. Aggregating Data

# Working on Large Data

Many times R users struggle helplessly while dealing with large data sets. When your machine fail to work on large data sets, it gives repetitive warnings, error messages of insufficient memory usage.

Below are some practices which impede R's performance on large data sets:

- Using **read.csv()** to load large files.
- Using **aggregate()** to perform aggregation on large datasets.
- Using web browser : Opening several tabs in chrome consumes a significant amount of system's memory.
- Machine Specification : R reads objects into RAM. Your system should have enough free RAM space available which could seamlessly work with large data.

There are more such factors and functions which hinder R's performance.



In case of large data sets, a data.frame can be limiting, the time it takes to do certain things is too long. The package **data.table** solves this by reducing computing time.

# Introduction to Package data.table

The R package data.table is written by Matt Dowle in year 2008.

It provides an enhanced version of **data.frame** that allows you to do :

- Fast aggregation of large data (e.g. 100GB in RAM)
- Fast ordered joins
- Fast add/modify/delete of columns by group using no copies at all
- List columns
- Fast file import (**fread**).

How to use package **data.table**?

- Import and load the package
- You can create your own data using **data.table()** exactly the same way you create dataframe or convert a dataframe into a datatable using **data.table()**

# Data Snapshot

sal\_data consists information about Employee's Basic Salary, their ID & full Name

→ 

Employee_ID	First_Name	Last_Name	Basic_Salary
E-1001	Mahesh	Joshi	16860
E-1002	Raj		
E-1004	Pr		
E-1005	Sn	Employee_ID	Employee ID
E-1007	R	First_Name	First Name
E-1008	N	Last_Name	Last Name
E-1009	Har	Basic_Salary	Basic Salary

Columns	Description	Type	Measurement	Possible values
Employee_ID	Employee ID	character	-	-
First_Name	First Name	character	-	-
Last_Name	Last Name	character	-	-
Basic_Salary	Basic Salary	numeric	Rs.	positive values

bonus\_data has information of only Bonus given to Employees.

→ 

Employee_ID	Bonus
E-1001	16070
E-1003	
E-1004	
E-1006	
E-1008	
E-1010	

Columns	Description	Type	Measurement	Possible values
Employee_ID	Employee ID	character	-	-
Bonus	Bonus amount	numeric	Rs.	Positive values

"Employee ID" is the common column in both datasets

# Importing Data

The **fread()** of package **data.table** loads large datasets faster and more conveniently than other base functions in R. **fread()** imports data in data.table format.

```
# Install and load package data.table  
# Compare read.csv() with fread()  
  
install.packages("data.table")  
library(data.table)  
  
system.time(dt<-read.csv("data.csv"))  
  
system.time(dt<-fread("data.csv"))
```

Here **data.csv** is a dummy file. Run these speed test codes on large data, you will see loading data with **fread()** is much faster than the base function **read.csv()**.  
**system.time()** is used to measure the execution time for an R function.



**fread()** is faster than **read.csv()** because, **read.csv()** first read rows as character and then converts them into integer and factor as data types. On the other hand, **fread()** simply reads everything as character.

# Merging Data

```
# Import data :  
  
sal_data <- fread("sal_data.csv")  
bonus_data <- fread("bonus_data.csv")
```

```
# Set key as ID for both data tables  
  
setkey(sal_data,Employee_ID)  
setkey(bonus_data,Employee_ID)
```

→ **setkey()** sorts the rows of the data table by the column(s) provided in ascending order. Keys in data table delivers incredibly fast results. You can set keys on any type of column i.e. numeric, factor, integer, character.

# Merging Data

```
# Merge data1 and data2
```

```
merged <- merge(sal_data,bonus_data,all = TRUE)
```



- ❑ **merge()** function in data.table is same as base R function, the only difference is that we don't have to mention "**by**" which variable we want to do the merging as we have already mentioned that in **setkey()** function.
- ❑ Also, here we have performed outer join by specifying **all = TRUE**
- ❑ We can perform others as follows:  
**all.x = TRUE** for left join, **all.y = TRUE** for right join &  
**merge(data1,data2)** for inner join

Here, **merge()** is the function of package data.table used for quick merging of data tables based on common keys (by default).

This function acts very similarly to that of data.frame's, with the major exception being that the default columns used to merge two datatables are the shared key columns, and not the shared columns with the same names.

# Creating Data

Create a large datatable to perform data manipulation tasks

```
# dt1 with 70,00,000 rows and 4 columns using random sampling
```

```
dt1<- data.table(ID=1:7000000,  
                  Capacity=sample(100:1000,size=50,replace=F),  
                  Code=sample(LETTERS[1:4],50,replace=T),  
                  State=rep(c("Alabama","Indiana","Texas","Nevada")))
```

```
head(dt1, 4)
```

	ID	Capacity	Code	State
1:	1	918	C	Alabama
2:	2	836	C	Indiana
3:	3	488	B	Texas
4:	4	647	A	Nevada

- **data.table()** is used to create a datatable and convert a dataframe into a datatable
- **sample(x, size, replace = FALSE)**
- **x** = either a vector of one or more elements from which to choose, or a positive integer
- **size** = a non-negative integer giving the number of items to choose.
- **replace** = should sampling be with replacement?

- **Capacity=sample(100:1000,size=50,replace=F)** : For column "Capacity" a sample of **size 50** from numbers 100 to 1000 is generated & sampling is done without replacement.
- **Code=sample(LETTERS[1:4],50,replace=T)**: Similarly, for column "Code" a sample of size 50 from 1st 4 letters(A:D) is generated & sampling is done with replacement.
- And the process is going to be repeated till 7000000 observations are generated.
- **State=rep(c("Alabama", "Indiana", "Texas", "Nevada"))** : Column "State" uses rep() function where the values are repeated in same order as mentioned till all 7000000 observations are generated.



Note : Here the Output will vary for each user as we are generating random sample.

# Creating Subsets

```
# Subsetting rows by numbers
```

```
dt1[4:6,]
```

#Output

	ID	Capacity	Code	State
1:	4	647	A	Nevada
2:	5	428	D	Alabama
3:	6	758	A	Indiana

```
# Use column names to select rows
```

```
sub_rows<-dt1[Code=="C" & State=="Alabama"]  
head(sub_rows,3)
```

#Output

	ID	Capacity	Code	State
1:	1	918	C	Alabama
2:	25	330	C	Alabama
3:	41	855	C	Alabama

# Creating Subsets

```
# Subsetting columns
```

```
sub_columns<-dt1[,.(ID,Capacity)]  
head(sub_columns, 3)
```

```
#Output
```

	ID	Capacity
1:	1	918
2:	2	836
3:	3	488

```
# Subset all rows using key columns where first key column “Code” matches  
“C” and second key column “State” matches “Alabama”
```

```
setkey(dt1,Code,State)  
sub_key<-dt1[ .("C","Alabama")]  
head(sub_key, 2)
```

```
#Output
```

	ID	Capacity	Code	State
1:	1	918	C	Alabama
2:	25	330	C	Alabama

Once the key is set, we no longer need to provide the column name again and again.

# Creating Subsets

```
# Subset all rows where just the first key column “Code” matches “C”
```

```
sub_key2<-dt1[.(“C”)]  
head(sub_key2, 2)
```

#Output

	ID	Capacity	Code	State
1:	1	918	C	Alabama
2:	25	330	C	Alabama

```
# Subset all rows where just the second key column “State” matches “Alabama”
```

```
sub_key3 <- dt1[.(unique(Code), “Alabama”)]  
head(sub_key3, 2)
```

#Output

	ID	Capacity	Code	State
1:	33	484	A	Alabama
2:	73	393	A	Alabama

We can not skip the values of key columns before. Therefore we provide all unique values from key column Code.

# Sorting Data

```
# Sort dt1 by variable 'Code' in ascending order and variable 'State'  
# in descending order
```

```
dt_order1<-dt1[order(Code,-State)]  
head(dt_order1,4)
```

#Output

	ID	Capacity	Code	State
1:	23	393	A	Texas
2:	27	865	A	Texas
3:	83	484	A	Texas
4:	123	393	A	Texas

By default **order()** sorts variables in ascending order.  
‘-’ sign results in descending order.

**NOTE :** **order()** in package `data.table` is much faster than base function `order()` because it uses **radix order sort** which imparts additional boost.

```
# Sort dt1 by variables 'Code' and 'Capacity' in descending order
```

```
dt_order2<-dt1[order(-Code,-Capacity)]  
head(dt_order2,3)
```

#Output

	ID	Capacity	Code	State
1:	17	990	D	Alabama
2:	117	990	D	Alabama
3:	217	990	D	Alabama

# Modifying Data

```
# Add a new column 'new_capacity' to dt1
```

```
dt1[,new_capacity:=Capacity+5]  
head(dt1,4)
```

#Output

	ID	Capacity	Code	State	new_capacity
1:	33	484	A	Alabama	489
2:	73	393	A	Alabama	398
3:	77	865	A	Alabama	870
4:	133	484	A	Alabama	489



Using '`:=`' operator, new columns can be added and assigned values.

**Remember** while sorting we had set the key `dt1` on 'Code' and 'State' so any operation on `dt1` will return the output sorted on keys.

```
# Update row values 'Alabama' to 'A1' in column 'State'
```

```
dt1[State=="Alabama",State:="A1"]  
head(dt1,3)
```

#Output

	ID	Capacity	Code	State	new_capacity
1:	33	484	A	A1	489
2:	73	393	A	A1	398
3:	77	865	A	A1	870



Columns are updated using '`:=`' operator

# Modifying Data

```
# Delete column 'Capacity' from dt1
```

```
dt1[,c("Capacity"):=NULL]  
head(dt1,2)
```

```
#Output
```

	ID	Code	State	new_capacity
1:	33	A	A1	489
2:	73	A	A1	398



Column can be removed, by assigning **NULL** to it, using '**:=**' operator.

```
# Concept of Chaining of commands
```

```
# Execute last three commands together in one command
```

```
dt1[,new_capacity:=Capacity + 5] [State=="Alabama",State=="A1"]  
[,"Capacity":=NULL]
```

# Modifying Data

```
# Renaming column 'new_capacity' from dt1  
  
setnames(dt1,old="new_capacity" , new = "New_Capacity")  
head(dt1,2)
```

#Output

	ID	Capacity	Code	State	New_Capacity
1:	1	851	A	A1	856
2:	5	862	A	A1	867



Note : Multiple columns can be renamed just in one code

```
setnames(data, old=c("old_name",...),new=c("new_names",...))
```

# Aggregating Data

```
# To calculate sum of variable 'New_Capacity' by variable 'State'
```

```
setkey(dt1, New_Capacity, State)  
DT_agg <- dt1[, sum(New_Capacity), by=State]
```

```
DT_agg
```

```
#Output
```

	State	v1
1:	Indiana	778680000
2:	Nevada	778680000
3:	Al	1017590000
4:	Texas	1017590000

- New\_Capacity** and **State** are set as keys for faster aggregation.
- by=** takes the factors by which data will be aggregated. Multiple factors can be specified with **list()**
- A new column **V1** is created with sum of **New\_Capacity**, **State** wise.

```
# To change the name of the column while aggregating, see the below command.
```

```
# Rename the column 'V1' to 'Totalcapacity'
```

```
DT_agg <- dt1[, .(Totalcapacity = sum(New_Capacity)), by=State]
```

```
DT_agg
```

```
#Output
```

	State	Totalcapacity
1:	Indiana	778680000
2:	Nevada	778680000
3:	Al	1017590000
4:	Texas	1017590000

.() notation allows you to rename the columns inside datatable.

# Quick Recap

In this session, we learnt how useful package **data.table** is for working on large data.

Here is the quick recap:

Importing, creating  
data and merging  
data

- **fread()** is used to import data. It is faster than read.csv()
- **data.table()** is used to create data
- **merge()** is used to merge data sets on common columns

Sub setting

- Subsetting can be done by rows and columns.
- **setkey()** sets the variables as keys and speeds up the subsetting

Sorting

- **order()** is used to sort data.table in ascending/  
descending order

Modifying

- Adding new column, updating values, deleting columns,  
chaining command & renaming columns for all these  
functions are possible with **data.table**

Aggregating

- **setkey()** is used to set keys on variables as it provides  
faster aggregation.
- Aggregation through package **data.table** is much faster  
than through R's built-in function **aggregate()**.

Package reshape2

Converting Data from Wide to Long Format

# Contents

1. Understanding the difference between Long Format and Wide Format
2. Introduction to reshape2
3. Reshaping data using cast and melt functions

# Long format vs Wide format

Some data have wide format and some have long; for different analysis you may choose one of the either.

In **Long Format** data, one column contains all the possible variables, another column contains their respective values. For example:

ID	Names	Subject	variable	value
1	Rohit	Maths	I_semester	78
2	John	Physics	I_semester	56
3	Vivek	Statistics	I_semester	76
4	Martina	Physics	I_semester	89
5	Agatha	Statistics	I_semester	67
1	Rohit	Maths	II_semester	88
2	John	Physics	II_semester	59
3	Vivek	Statistics	II_semester	81
4	Martina	Physics	II_semester	73
5	Agatha	Statistics	II_semester	80

In **Wide Format** data, each column represents a different variable. For example:

ID	Names	Subject	I_semester	II_semester
1	Rohit	Maths	78	88
2	John	Physics	56	59
3	Vivek	Statistics	76	81
4	Martina	Physics	89	73
5	Agatha	Statistics	67	80

# Long format vs Wide format

- Long data is mainly used in data visualization (plotting graphs) and statistical modeling. Also it is easy to perform operations on subsets of data.
- Wide data is used in the calculation of growth, for example, we have a company data with list of companies and their revenue for 4 Quarters. Our data is in wide format with 5 columns: Company, Quarter1, Quarter2, Quarter3, Quarter4. Since our data is in a wide format it is easy to calculate Quarter wise growth.
- In practical scenario, while wide format is more readable, long format is easier to analyze. Therefore, it is useful to know how to convert between the two.

# Introduction to reshape2

- We know that data comes in many forms. Hence, we are required to reshape it according to our need.
- The process of reshaping data in R is tedious. R base functions for 'aggregation' reduces and rearranges the data into smaller forms, but with reduction in amount of information. Also, these functions don't solve the problem of converting data from wide to long format & vice versa. The package **reshape2** overcomes these problems.
- **reshape2** is an R package written and maintained by Hadley Wickham.
- **reshape2** makes it easy to convert data from long to wide formats & vice versa.

Package reshape2 has two key functions:

melt – converts data from wide format to long format

cast – converts data from long format to wide format

# Data Snapshot

stud\_data consists student names & their marks scored in Math's, Economics & Statistics. It has 5 rows & 5 columns.

Variables					
Observations	Student_ID	Names	Maths	Economics	Statistics
	1	Rohan	67	56	
	2	John	89	88	79
Columns	Description	Type	Measurement	Possible values	
Student_ID	Student ID	character	-	-	
Names	Student names	character	-	-	
Maths	Marks scored in Maths	numeric	-	positive values	
Economics	Marks scored in Economics	numeric	-	positive values	
Statistics	Marks scored in Statistics	numeric	-	positive values	

# Tasks to be Performed

stud\_data is originally in a wide format. We will perform following activities using package reshape2:

- Convert data format from wide to long & Changing the names of new columns that will be created in the conversion process.
- Convert data format from long to wide. While doing this conversion perform subsetting and aggregation.

# melt()

- `melt()` converts data from wide format to long format. It's a kind of restructuring where categorical variables are 'melted' into unique observations.
- Here variables are divided into 2 groups: identifier and measured variables.
- Identifier variables(id) identify the unit or column that measurements takes place on.

```
# Install and load package reshape2  
# Import stud_data  
# Convert wide data into long  
  
install.packages("reshape2")  
library(reshape2)  
stud_data<-read.csv("stud_data.csv",header=TRUE)
```

# melt()

```
melt(stud_data)
```

#Output

Using Names as id variables		
	Names	variable value
1	Rohan	Student_ID 1
2	John	Student_ID 2
3	Anisha	Student_ID 3
4	Agatha	Student_ID 4
5	Ashima	Student_ID 5
6	Rohan	Maths 67
7	John	Maths 89
8	Anisha	Maths 69
9	Agatha	Maths 79
10	Ashima	Maths 77
11	Rohan	Economics 56
12	John	Economics 88
13	Anisha	Economics NA
14	Agatha	Economics 92
15	Ashima	Economics 67
16	Rohan	Statistics NA
17	John	Statistics 79
18	Anisha	Statistics 88
19	Agatha	Statistics 89
20	Ashima	Statistics 89

- ❑ By default, **melt()** assumes factor and character variables are id variables, and all others are measured. Here, **Names** is assumed as id variable and rest of the columns as measured variables.
- ❑ Column “**variable**” contains measured variables and Column “**value**” contains their respective values.
- ❑ All measured variables must be of same type, (e.g. numeric, factor, date) because melted data is stored in a R data frame, and the column **value** can assume only one type.
- ❑ Note that **stud\_data** has 2 missing values. To remove missing values from the melted data set, include **na.rm=TRUE**.

# melt() with id.vars

ID variables can be separately specified

```
# Change the ID variables to 'Student_ID' and 'Names'  
melt(stud_data,id.vars=c('Student_ID','Names'))
```

#Output

	Student_ID	Names	variable	value
1	1	Rohan	Maths	67
2	2	John	Maths	89
3	3	Anisha	Maths	69
4	4	Agatha	Maths	79
5	5	Ashima	Maths	77
6	1	Rohan	Economics	56
7	2	John	Economics	88
8	3	Anisha	Economics	NA
9	4	Agatha	Economics	92
10	5	Ashima	Economics	67
11	1	Rohan	Statistics	NA
12	2	John	Statistics	79
13	3	Anisha	Statistics	88
14	4	Agatha	Statistics	89
15	5	Ashima	Statistics	89

With **id.vars=** we can give multiple id variables & we can specify them in a list of variables enclosed in **c()**. If we only specify id variables, by default non id variables will be taken as measured variables.

See the next example for setting measured variables.

# melt() with measure.vars

```
# Specify measured variables as Maths and Economics
```

```
melt(stud_data,id.vars=c('Student_ID','Names'),measure.vars=c('Maths',  
'Economics'))
```

#Output

	Student_ID	Names	variable	value
1		1 Rohan	Maths	67
2		2 John	Maths	89
3		3 Anisha	Maths	69
4		4 Agatha	Maths	79
5		5 Ashima	Maths	77
6		1 Rohan	Economics	56
7		2 John	Economics	88
8		3 Anisha	Economics	NA
9		4 Agatha	Economics	92
10		5 Ashima	Economics	67

With **measure.vars=** one can specify a vector of measured variables.

# melt() with variable.name and value.name

What if we want to control the column names in our long format data? We can do that in just one step:

```
# Change the names of new columns created after melting stud_data  
long_format<-melt(stud_data,id.vars=c('Student_ID','Names'),  
variable.name='Subjects', value.name='Marks')  
long_format
```

#Output

	Student_ID	Names	Subjects	Marks
1		1 Rohan	Maths	67
2		2 John	Maths	89
3		3 Anisha	Maths	69
4		4 Agatha	Maths	79
5		5 Ashima	Maths	77
6		1 Rohan	Economics	56
7		2 John	Economics	88
8		3 Anisha	Economics	NA
9		4 Agatha	Economics	92
10		5 Ashima	Economics	67
11		1 Rohan	Statistics	NA
12		2 John	Statistics	79
13		3 Anisha	Statistics	88
14		4 Agatha	Statistics	89
15		5 Ashima	Statistics	89

Using **variable.name=** and **value.name=** arguments, we can change the names of the columns “**variable**” and “**value**”.

# cast function

- **cast** function converts data from long format to wide format. It is a complement of **melt()** function. There are two types of **cast** functions: **dcast()** and **acast()**
  - dcast()** – returns a dataframe as the output
  - acast()** – returns a vector/matrix/array as the output

The basic arguments of \*cast is the data in long format and a formula of the form:

**x1 + x2 ~ y1 + y2**

id variables are specified on the left and measured variables on the right.

The order of the variables matter, the first varies slowest, and the last fastest.

- Since dataframe objects are the most common, we will explore the **dcast()** function.

Let's take the **long-format** data object which we created in the previous example and cast it into some different wide formats.

# dcast() with Formula

```
# Reshape long_format object into wide format
```

```
dcast(long_format,Student_ID+Names~Subjects)
```

```
#Output
```

```
Using Marks as value column: use value.var to override.
```

	Student_ID	Names	Maths	Economics	Statistics
1	1	Rohan	67	56	NA
2	2	John	89	88	79
3	3	Anisha	69	NA	88
4	4	Agatha	79	92	89
5	5	Ashima	77	67	89

dcast() uses a formula to describe the shape of the data.

```
# There are different ways in which you can dcast data.
```

```
# See the below commands :
```

```
dcast(long_format, Names+Student_ID~Subjects)
```

```
dcast(long_format,Student_ID~Subjects)
```

```
dcast(long_format,Names ~ Subjects)
```

# dcast() with subset

You can subset variables while converting data from long to wide format.

```
# Subsetting  
# Reshape long_format object into wide format displaying columns:  
# Student_ID, Names and Maths column
```

```
install.packages("plyr")  
library(plyr)  
dcast(long_format,Student_ID+Names~Subjects,  
subset=.(Subjects=="Maths"))
```

#Output

Using Marks as value column: use value.var to override.

	Student_ID	Names	Maths
1	1	Rohan	67
2	2	John	89
3	3	Anisha	69
4	4	Agatha	79
5	5	Ashima	77

Here, we have sub setted **Maths** from the variable **Subjects**.

- ❑ **subset=** is used to subset data using `.()`
- ❑ package **plyr** is needed to access `'.'`

# dcast() with fun.aggregate

One confusing “mistake” you might make is casting a dataset in which there is more than one value per data cell. For example, this time we won’t include any measured variable.

```
dcast(long_format,Student_ID+Names ~ .)
```

#Output

```
Using Marks as value column: use value.var to override.  
Aggregation function missing: defaulting to length
```

Student_ID	Names	.
1	1	Rohan
2	2	John
3	3	Anisha
4	4	Agatha
5	5	Ashima

- “.” represents no variable.
- Notice a warning message: Aggregation function missing: defaulting to length
- If you look at the output, the cells are filled with the number of data rows for each **Student\_ID-Names** combination.
- The numbers we’re seeing are the number of **Subjects** for each **Student\_ID-Names** combination .

So, when you cast your data and there are multiple values per cell, you also need to tell **dcast** how to aggregate the data. See the next example for using aggregation function.

# dcast() with fun.aggregate

Calculate total marks for all students

```
# Aggregation
```

```
dcast(long_format, Student_ID+Names ~ .,fun.aggregate=sum,na.rm=TRUE)
```

Using Marks as value column: use value.var to override.

	Student_ID	Names	.
1	1	Rohan	123
2	2	John	256
3	3	Anisha	157
4	4	Agatha	260
5	5	Ashima	233

- ❑ **fun.aggregate** is used to specify the aggregation function
- ❑ **na.rm=TRUE** is to remove NA values.
- ❑ In base R, **aggregate()** also has similar formula where variables on left hand side are continuous and on right hand side are categorical.

In this case, an aggregation function reduces multiple values to a single one.

# Quick Recap

In this session, we learnt how to change the data from long to wide format and vice-versa. Here is the quick recap:

## Long vs Wide Format

- In Long Format data, one column contains all the possible variables, another column contains their respective values.
- In Wide Format data, each column represents a different variable.

## melt()

- melt() converts data from wide format to long format.
- Usage: `melt(data,id.vars,measure.vars,variable.name="variable",na.rm=FALSE,value.name="value")`

## cast()

- Use **acast** or **dcast** depending on whether you want vector/matrix/array output or data frame output.
- Usage: `dcast(data, formula,fun.aggregate=NULL,subset = NULL)`

# Package sqldf

Importing, Sorting, Subsetting, Modifying,  
Aggregating and Merging Data

# Contents

1. Introduction
2. Importing Data
3. Sorting Data
4. Renaming Data
5. Subsetting Data
6. Aggregating Data
7. Merging Data

# sqldf

- For those who are learning R and may be well-versed in SQL, package sqldf is very useful because it enables us to use SQL commands in R.
- One who has basic SQL skills can manipulate data frames and data tables in R using their SQL skills.
- SQL stands for Structured Query Language, with data stored as tables in a database. There are number of database types, which are reasonably similar and sqldf uses SQLite as default.
- In this tutorial, we will see how to import, sort, modify, subset, aggregate and merge data in R using SQL.
- Let's install the package and load the package to run SQL queries on R Data frames and data tables.

```
install.packages("sqldf")
library(sqldf)
```

# Data Snapshot

basic\_salary data consist salary of each employee with it's Location & Grade.

Variables

Observations

Columns	Description	Type	Measurement	Possible values
First_Name	First Name	character	-	-
Last_Name	Last Name	character	-	-
Grade	Grade	character	GR1, GR2	2
Location	Location	character	DELHI, MUMBAI	2
ba	Basic Allowance	numeric	Rs.	positive values
ms	Management Supplements	numeric	Rs.	positive values

# Importing data – read.csv.sql()

- **read.csv.sql()** function is used to import the datasets into R, filtering it with an SQL statement.

```
salary_data<-read.csv.sql("basic_salary.csv",sql="select * from file",header=TRUE)
```

- **file** refers to the file given as first argument i.e 'basic\_salary'.
- This function is very similar to other functions that allow importing datasets into R, with the sole exception that the second argument you pass is an SQL statement.
- Only the filtered portion is processed by R so that files larger than R can otherwise handle & can be accommodated.
- This function replaces the NA values with 0.

# Sorting Data – order by

Sort salary\_data by 'ba' in Descending order

```
# Sorting by one column
```

```
sortdata<-"select * from salary_data order by ba desc"  
sorteddf<-sqldf(sortdata)  
sorteddf
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Aaron	Jones	GR1	MUMBAI	23280	13490
2	Sneha	Joshi	GR1	DELHI	20660	0
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
4	Neha	Rao	GR1	MUMBAI	19235	15200
5	Alan	Brown	GR1	DELHI	17990	16070
6	Ameet	Mishra	GR2	DELHI	14780	9300
7	Gaurav	Singh	GR2	DELHI	13760	13220
8	Adela	Thomas	GR2	DELHI	13660	6840
9	John	Patil	GR2	MUMBAI	13500	10760
10	Sagar	Chavan	GR2	MUMBAI	13390	6700
11	Agatha	Williams	GR2	MUMBAI	12390	6630
12	Anup	Save	GR2	MUMBAI	11960	7880

- order by clause is used to sort data base on one or more columns
- For sorting in descending order **desc** is to be specified after that particular column name. By default, it sorts data in ascending order.

# Sorting Data – order by

Sort salary\_data by 'First\_Name' in Descending order and 'ba' in Ascending order

```
# Sorting by multiple columns with different ordering levels
```

```
sortdata2<-"select * from salary_data order by First_Name desc,ba"
sorteddf2<-sqldf(sortdata2)
sorteddf2
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Sneha	Joshi	GR1	DELHI	20660	0
2	Sagar	Chavan	GR2	MUMBAI	13390	6700
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
4	Neha	Rao	GR1	MUMBAI	19235	15200
5	John	Patil	GR2	MUMBAI	13500	10760
6	Gaurav	Singh	GR2	DELHI	13760	13220
7	Anup	Save	GR2	MUMBAI	11960	7880
8	Ameet	Mishra	GR2	DELHI	14780	9300
9	Alan	Brown	GR1	DELHI	17990	16070
10	Agatha	Williams	GR2	MUMBAI	12390	6630
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Aaron	Jones	GR1	MUMBAI	23280	13490

Column names are specified by which data is sorted with **order by** clause.

# Renaming Columns - as

```
# Display columns First_Name,Grade,Location,ba of salary data and  
# Rename column 'ba' as Basic_Salary
```

```
renamecols<-"select First_Name,Grade,Location,ba as Basic_Salary from ←  
salary_data"  
renamecolsdf<-sqldf(renamecols)  
renamecolsdf
```

```
# Output
```

	First_Name	Grade	Location	Basic_Salary
1	Alan	GR1	DELHI	17990
2	Agatha	GR2	MUMBAI	12390
3	Rajesh	GR1	MUMBAI	19250
4	Ameet	GR2	DELHI	14780
5	Neha	GR1	MUMBAI	19235
6	Sagar	GR2	MUMBAI	13390
7	Aaron	GR1	MUMBAI	23280
8	John	GR2	MUMBAI	13500
9	Sneha	GR1	DELHI	20660
10	Gaurav	GR2	DELHI	13760
11	Adela	GR2	DELHI	13660
12	Anup	GR2	MUMBAI	11960

as clause is used with select statement to rename the columns in the output.

# Subsetting Data

Display columns 'First\_Name' and 'ba' of salary\_data

```
# Column Subsetting
```

```
subcols<-"select First_Name,ba from salary_data"  
subset_cols<-sqldf(subcols)  
subset_cols
```

```
# Output
```

	First_Name	ba
1	Alan	17990
2	Agatha	12390
3	Rajesh	19250
4	Ameet	14780
5	Neha	19235
6	Sagar	13390
7	Aaron	23280
8	John	13500
9	Sneha	20660
10	Gaurav	13760
11	Adela	13660
12	Anup	11960

**Column Subsetting** is done by modifying the **select** statement by just giving the columns to be subsetted.

# Subsetting Data

Show the records of employees having 'ba' more than 15000.

```
# Row Subsetting
```

```
subdata<- "select * from salary_data where ba>15000"  
subset_ba<-sqldf(subdata)  
subset_ba
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
2	Rajesh	Kolte	GR1	MUMBAI	19250	14960
3	Neha	Rao	GR1	MUMBAI	19235	15200
4	Aaron	Jones	GR1	MUMBAI	23280	13490
5	Sneha	Joshi	GR1	DELHI	20660	0

**Row Subsetting** is done by using **where** clause with a condition in SQL query

# Subsetting Data

Display the 'First\_Name' and ba of only those employees who are from MUMBAI 'Location' and having GR1 'Grade'.

```
sqldf("select First_Name,ba from salary_data where Location='MUMBAI'  
and Grade='GR1'")
```

# Output

	First_Name	ba
1	Rajesh	19250
2	Neha	19235
3	Aaron	23280

By using **and** clause, we can give multiple conditions for subsetting data.

# Subsetting Data

Calculate Total Salary of employees and display records of employees whose Total salary is greater than the median of Total salary of all employees

# Calculating Total Salary

```
new_saldata<-sqldf("select  
First_Name,Last_name,Grade,Location,ba,ms,sum(ba+ms) as TS from  
salary_data group by First_Name") ←  
new_saldata
```

# Output

	First_Name	Last_Name	Grade	Location	ba	ms	TS
1	Aaron	Jones	GR1	MUMBAI	23280	13490	36770
2	Adela	Thomas	GR2	DELHI	13660	6840	20500
3	Agatha	Williams	GR2	MUMBAI	12390	6630	19020
4	Alan	Brown	GR1	DELHI	17990	16070	34060
5	Ameet	Mishra	GR2	DELHI	14780	9300	24080
6	Anup	Save	GR2	MUMBAI	11960	7880	19840
7	Gaurav	Singh	GR2	DELHI	13760	13220	26980
8	John	Patil	GR2	MUMBAI	13500	10760	24260
9	Neha	Rao	GR1	MUMBAI	19235	15200	34435
10	Rajesh	Kolte	GR1	MUMBAI	19250	14960	34210
11	Sagar	Chavan	GR2	MUMBAI	13390	6700	20090
12	Sneha	Joshi	GR1	DELHI	20660	0	20660

Here, we have used **sum()** function to calculate total salary and named that column as TS

# Subsetting Data

```
# Record of employees whose TS > median TS of all the employees
```

```
sqldf("select * from new_saldata where TS>(select median(TS) from  
new_saldata)")
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms	TS
1	Aaron	Jones	GR1	MUMBAI	23280	13490	36770
2	Alan	Brown	GR1	DELHI	17990	16070	34060
3	Gaurav	Singh	GR2	DELHI	13760	13220	26980
4	John	Patil	GR2	MUMBAI	13500	10760	24260
5	Neha	Rao	GR1	MUMBAI	19235	15200	34435
6	Rajesh	Kolte	GR1	MUMBAI	19250	14960	34210

We can use multiple **select** statements inside the another.  
Here, we have used a **select** statement to calculate  
median of **TS** inside another **select** statement

# Subsetting Data

```
# Who are the Top 3 highest paid employees?  
# Display their 'First_Name', 'Grade' and 'TS'
```

```
sqldf("select First_Name,Grade,TS from new_saldata order by TS desc  
limit 3") ←
```

```
# Output
```

	First_Name	Grade	TS
1	Aaron	GR1	36770
2	Neha	GR1	34435
3	Rajesh	GR1	34210

**limit** statement is used to retrieve records and limit the number of records returned based on a **limit** value.

## Interpretation :

- Above command sorts the data in the descending order of **TS** using **order by** clause and retrieves first 3 records from the data using **limit** statement.

# Subsetting Data – More Examples

Few of the SQL queries for subsetting data:

```
# Subsetting rows using Logical operators (SQL and)
```

```
query1<- "select * from sal_data where Basic_Salary>14000 and  
First_Name like 'r%'"  
sqldf(query1)
```

This command will return the rows having First Name starting with ‘r’ and basic Salary greater than 14000. **SQL ‘or’ operator** can also be used if we want either the first condition OR the second condition to be true.

```
# Subsetting rows by value range (SQL between)
```

```
query2<- "select * from sal_data where Basic_Salary between 16000 and  
17000"  
sqldf(query2)
```

This command will return all the records of employees whose Basic Salary is between 16000-17000.

# Aggregating Data – group by

```
# Calculate sum of variable 'ba' by variable 'Location'
```

```
agg_sum<-"select Location,sum(ba) as sum_of_ba from salary_data group  
by Location"  
aggdf<-sqldf(agg_sum)  
aggdf
```

```
# Output
```

	Location	sum_of_ba
1	DELHI	80850
2	MUMBAI	113005

- **as** clause is used to change the name of the column in the output.
- **group by** clause is used with select statement for aggregation in SQL.
- Aggregation functions like **sum()**, **count()** , **avg()** can be used with **select** statement.

```
# Display No. of Employees Location wise
```

```
agg_count<-"select Location,count(*) as No_of_employees from  
salary_data group by Location"  
countdf<-sqldf(agg_count)  
countdf
```

```
# Output
```

	Location	No_of_employees
1	DELHI	5
2	MUMBAI	7

# Aggregating Data – group by

```
# Aggregate TS(Total Salary) and show the percentage share of TS by  
# 'Location'
```

```
sum_ts<-sqldf("select Location,sum(TS) as Total_Salary from  
new_saldata group by Location")  
sum_ts
```

```
# Output
```

	Location	Total_Salary
1	DELHI	126280
2	MUMBAI	188625

- This command calculates the sum of TS by Location and displays it with Location.
- Here we have used the data new\_saldata which we created while subsetting based on TS.

```
sqldf("select Location,Total_Salary,(Total_Salary*100/(select  
sum(Total_Salary)from sum_ts))as Percent_share from sum_ts")
```

```
# Output
```

	Location	Total_Salary	Percent_share
1	DELHI	126280	40
2	MUMBAI	188625	59

This command displays **Location**, **Total Salary** & percentage share of **TS** by **Location**

# Data Snapshot

sal\_data consist information about Employee's Basic Salary, their ID & full Name

Employee_ID	First_Name	Last_Name	Basic_Salary
E-1001	Mahesh	Joshi	16860
E-1002	Raj		
E-1004	Priya		
E-1005	Sneha		
E-1007	Ravi		
E-1008	Nisha		
E-1009	Hari		

Columns	Description	Type	Measurement	Possible values
Employee_ID	Employee ID	character	-	-
First_Name	First Name	character	-	-
Last_Name	Last Name	character	-	-
Basic_Salary	Basic Salary	numeric	Rs.	positive values

bonus\_data has information of only Bonus given to Employees.

"Employee ID" is the common column in both datasets

Employee_ID	Bonus
E-1001	16070
E-1003	
E-1004	
E-1006	
E-1008	
E-1010	

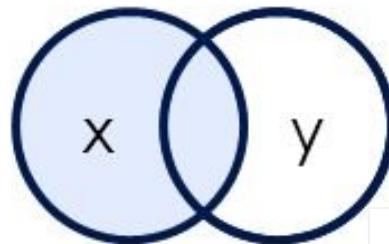
  

Columns	Description	Type	Measurement	Possible values
Employee_ID	Employee ID	character	-	-
Bonus	Bonus amount	numeric	Rs.	Positive values

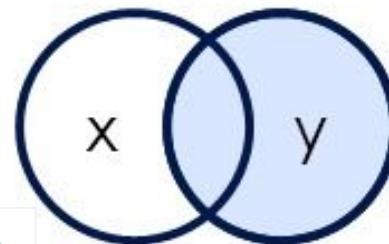
# Merging Data-Types of Joins

Consider `sal_data =x` and `bonus_data=y`

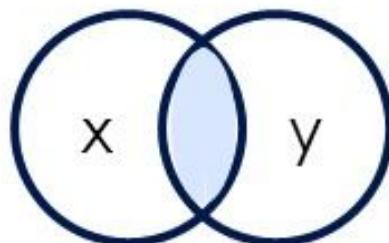
Left Join



Right Join

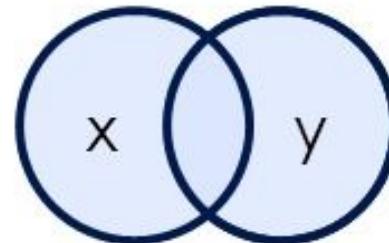


Inner Join



There are  
4 types of  
Joins

Outer Join



# Merging Data

- Performing joins is one of the most common operations in SQL.
- **Left Join** returns all rows from the left table, and any rows with matching keys from the right table whereas **Right Join** returns all rows from the right table, and any rows with matching keys from the left table.
- **Inner joins** return only rows with matching data for the common column, and **full outer joins** return all rows in all data sets, even if there are rows without matches.
- Currently, **sqldf** does not support right joins or full outer joins.
- Let's perform **left** and **inner join** on **sal\_data** and **bonus\_data**.
- Before performing joins first import two data sets **sal\_data** and **bonus\_data** and store them in objects as **sal\_data** and **bonus\_data**.

```
sal_data<-read.csv.sql("sal_data.csv",sql="select * from file",
header=TRUE)
```

```
bonus_data<-read.csv.sql("bonus_data.csv",sql="select * from file",
header=TRUE)
```

# Merging Data – Left Join

```
# Left Join
```

```
leftjoin_string<-"select sal_data.* , bonus_data.Bonus from sal_data  
left join bonus_data on sal_data.Employee_ID = bonus_data.Employee_ID"  
  
sal_join_bonus<-sqldf(leftjoin_string)  
sal_join_bonus
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1002	Rajesh	Kolte	14960	NA
3	E-1004	Priya	Jain	12670	13490
4	E-1005	Sneha	Joshi	15660	NA
5	E-1007	Ram	Kanade	15850	NA
6	E-1008	Nishi	Honrao	15950	15880
7	E-1009	Hameed	Singh	15120	NA

A new data frame, `sal_join_bonus`, will be created using `sqldf()`.  
`sqldf()` at minimum, requires a character string with the SQL operation to be performed..



`merge()` in base R performs the equivalent of inner and left joins, as well as right and full outer joins, which are unavailable in `sqldf`.

# Merging Data – Inner Join

```
# Inner Join
```

```
innerjoin_string<-"select sal_data.*,bonus_data.Bonus from sal_data  
inner join bonus_data on sal_data.Employee_ID=bonus_data.Employee_ID"  
  
sal_join_bonus2<-sqldf(innerjoin_string)  
sal_join_bonus2
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1004	Priya	Jain	12670	13490
3	E-1008	Nishi	Honrao	15950	15880

# Data management tasks using sqldf

## Data management tasks using sqldf

- Importing data: Using `read.csv.sql()`
- Sorting data: Using `order by` clause
- Renaming columns: Using `as` clause
- Subsetting data: Using `where` clause
- Aggregating data: Using aggregation function on columns and `group by` clause
- Merging data: Left and inner join

# tidyverse package

Transposing, Splitting and Concatenating

# Contents

1. Introduction to `tidyverse`
2. Reshaping Data Using the Following Functions:
  - `gather()`
  - `spread()`
  - `separate()`
  - `unite()`

# Introduction

- The way package **reshape2** is used to reshape data, **tidyR** is also one of the options to do the same.
- **tidyR** is developed by Hadley Wickham which provides four main functions: **gather()**, **spread()**, **separate()**, **unite()** which are similar to functions in package reshape2. We will see the difference and similarity between the two packages while performing the following tasks:
  - Convert data from long format to wide & vice versa
  - Split one character column into multiple columns
  - Concatenate multiple columns into one column

# Data Snapshot

stud\_data consist student names & their marks scored in Math's, Economics & Statistics. It has 5 rows & 5 columns.

Variables					
Observations	Student_ID	Names	Maths	Economics	Statistics
	1	Rohan	67	56	
	2	John	89	88	79
	Columns	Description	Type	Measurement	Possible values
	Student_ID	Student ID	character	-	-
	Names	Student names	character	-	-
	Maths	Marks scored in Maths	numeric	-	positive values
	Economics	Marks scored in Economics	numeric	-	positive values
	Statistics	Marks scored in Statistics	numeric	-	positive values

# gather()

- **gather()** converts data from wide format to long format. It takes multiple columns and collapses into key-value pairs.

key is the name of new “key” column (made from names of data columns)

value is the name of new “value” column

The diagram illustrates the transformation of a wide-format data table into a long-format data table using the `gather()` function. On the left, a wide-format table shows student IDs (1-5) with their names (Rohan, John, Anisha, Agatha, Ashima) and scores in three subjects: Maths, Economics, and Statistics. A blue rounded rectangle highlights the columns for Maths, Economics, and Statistics. A large blue arrow points from this wide table to a long-format table on the right. The long-format table has four columns: Student\_ID, Names, Subjects, and Marks. The data is now organized by student and subject, with each row representing a specific score. The original subject names (Maths, Economics, Statistics) have been converted into the 'Subjects' column, and the scores are in the 'Marks' column.

Student_ID	Names	Maths	Economics	Statistics
1	Rohan	67	56	
2	John	89	88	79
3	Anisha	69		88
4	Agatha	79	92	89
5	Ashima	77	67	89

Student_ID	Names	Subjects	Marks
1	Rohan	Maths	67
2	John	Maths	89
3	Anisha	Maths	69
4	Agatha	Maths	79
5	Ashima	Maths	77
1	Rohan	Economics	56
2	John	Economics	88
3	Anisha	Economics	NA
4	Agatha	Economics	92
5	Ashima	Economics	67
1	Rohan	Statistics	NA
2	John	Statistics	79
3	Anisha	Statistics	88
4	Agatha	Statistics	89
5	Ashima	Statistics	89

# gather()

```
# Install and load package tidyR  
# Import stud_data data  
# Convert the stud_data format to long format
```

```
install.packages("tidyR")  
library(tidyR)  
stud_data<-read.csv("stud_data.csv",header=TRUE)  
  
gather(stud_data)
```



Wide Format data: each column represents a different variable.  
Long Format data: one column contains all the possible variables, another column contains their respective values  
Detailed discussion of these two formats is done in our tutorial for package **reshape2**

# gather()

```
gather(stud_data) ←
```

	key	value
1	Student_ID	1
2	Student_ID	2
3	Student_ID	3
4	Student_ID	4
5	Student_ID	5
6	Names	Rohan
7	Names	John
8	Names	Anisha
9	Names	Agatha
10	Names	Ashima
11	Maths	67
12	Maths	89
13	Maths	69
14	Maths	79
15	Maths	77
16	Economics	56
17	Economics	88
18	Economics	<NA>
19	Economics	92
20	Economics	67
21	Statistics	<NA>
22	Statistics	79
23	Statistics	88
24	Statistics	89
25	Statistics	89

Warning message:

attributes are not identical across measure variables;  
they will be dropped

In our tutorial for Package **reshape2**, we have seen how **melt()** function works. **gather()** is similar to it.

- ❑ **gather()** converts data from wide to long format with a warning by treating all columns as key while **melt()** treats **Names** as id variables (Id columns are the columns that contain the identifier of the observation that is represented as a row in our data set).
- ❑ Indeed, if we don't specify any id variables to **melt()**, then it will use the factor or character columns as id variables whereas **gather()** requires mentioning the columns that needs to be treated as key-value pairs.

# gather() with key:value Pair

```
# Here, we are adding new variables Subjects and Marks.
```

```
longformat<-gather(stud_data,Subjects,Marks,Maths,Economics,Statistics)  
longformat
```

```
# Output
```

	Student_ID	Names	Subjects	Marks
1		1 Rohan	Maths	67
2		2 John	Maths	89
3		3 Anisha	Maths	69
4		4 Agatha	Maths	79
5		5 Ashima	Maths	77
6		1 Rohan	Economics	56
7		2 John	Economics	88
8		3 Anisha	Economics	NA
9		4 Agatha	Economics	92
10		5 Ashima	Economics	67
11		1 Rohan	Statistics	NA
12		2 John	Statistics	79
13		3 Anisha	Statistics	88
14		4 Agatha	Statistics	89
15		5 Ashima	Statistics	89

In the syntax of **gather()**, after the data is mentioned:

- 1<sup>st</sup> mentioned variable is key
- 2<sup>nd</sup> mentioned variable is value
- and the subsequent variables are the ones' to be gathered.

This command can be alternatively written as:

```
longformat<-gather(stud_data,  
Subjects,Marks, Maths:Statistics)
```

Note that this data has 2 missing values

To remove rows from output where the value column is **NA**, include **na.rm=TRUE**.



**gather()** cannot handle matrices or arrays, while **melt()** can.

# spread()

- **spread()** converts data from long format to wide format.
- The **spread()** function spreads a key-value pair across multiple columns. It's a complement of **gather()**.

The diagram illustrates the transformation of data from a long format table to a wide format table using the `spread()` function. A large blue arrow points from the left table to the right table, indicating the direction of the transformation. The left table (long format) has columns: `Student_ID`, `Names`, `Subjects`, and `Marks`. The right table (wide format) has columns: `Student_ID`, `Names`, `Maths`, `Economics`, and `Statistics`. The `Subjects` column from the long table is mapped to the `Maths`, `Economics`, and `Statistics` columns in the wide table. The `Maths` column in the wide table is highlighted with a blue border. The `Names` column is also highlighted with a blue border in both tables.

Student_ID	Names	Subjects	Marks
1	Rohan	Maths	67
2	John	Maths	89
3	Anisha	Maths	69
4	Agatha	Maths	79
5	Ashima	Maths	77
1	Rohan	Economics	56
2	John	Economics	88
3	Anisha	Economics	NA
4	Agatha	Economics	92
5	Ashima	Economics	67
1	Rohan	Statistics	NA
2	John	Statistics	79
3	Anisha	Statistics	88
4	Agatha	Statistics	89
5	Ashima	Statistics	89

Student_ID	Names	Maths	Economics	Statistics
1	Rohan	67	56	
2	John	89	88	79
3	Anisha	69		88
4	Agatha	79	92	89
5	Ashima	77	67	89

# spread() with fill

```
# Convert the longformat data from last example to wide format
```

```
spread(longformat, Subjects, Marks, fill=0) ←
```

```
# Output
```

	Student_ID	Names	Economics	Maths	Statistics
1	1	Rohan	56	67	0
2	2	John	88	89	79
3	3	Anisha	0	69	88
4	4	Agatha	92	79	89
5	5	Ashima	67	77	89

- ❑ **spread()** two columns(key-value pair)are spread into multiple columns, making 'long' data wider.
- ❑ **fill=** is used to replace NA's with the value provided to it.
- ❑ **Note** that there are **two types of missingness** in the input: explicit missing values (i.e. NA), and implicit missing rows that simply aren't present. Both types of missing value will be replaced by fill.



**dcast()** from the package reshape2 performs the same task as **spread()**. Also, **dcast()** allows to perform aggregation which is not possible in package **tidyR**

# separate()

**separate()** splits a single character column into multiple columns.

```
# Create a data frame empdata with columns as empid, location,  
# address and date.  
# Split the column date in 3 columns
```

```
empid<-c(101,102,103,104)  
location<-c("Mumbai","Delhi","Delhi","Mumbai")  
address<-c("4/Churchgate","12/Rohini","8/Pitampura", "21/Andheri")  
date<-c("2016-10-09","2010-11-01","2009-09-23","1990-02-30")  
empdata<-data.frame(empid,location,address,date) ←
```

```
empdata
```

```
# Output
```

	empid	location	address	date
1	101	Mumbai	4/Churchgate	2016-10-09
2	102	Delhi	12/Rohini	2010-11-01
3	103	Delhi	8/Pitampura	2009-09-23
4	104	Mumbai	21/Andheri	1990-02-30

**employee data:**

columns: **empid**(Employee ID), **Location**,  
**address**(sector and area), **date**(Date of  
joining)

# separate() with into and convert

```
sep_date<-separate(empdata,date,into=c("Year","Month","Date"))  
sep_date
```

# Output

	empid	location	address	Year	Month	Date
1	101	Mumbai	4/Churchgate	2016	10	09
2	102	Delhi	12/Rohini	2010	11	01
3	103	Delhi	8/Pitampura	2009	09	23
4	104	Mumbai	21/Andheri	1990	02	30

into= takes names of new columns to create character vector.

```
class(sep_date$Year)←
```

```
[1] "character"
```

By default, new columns created will be of the type of original column. Here, since date is of type character, columns Year, Month and Date will of the same type.

```
sep_date<-separate(empdata,date,into=c("Year","Month","Date"),  
convert=TRUE)
```

```
class(sep_date$Year)
```

```
[1] "integer"
```

convert=TRUE will run type.convert with as.is=TRUE on new columns. This is useful if the component columns are integer, numeric or logical.

# separate() with sep

```
# Split the column address in 2 columns
```

```
sep_address<-separate(empdata,address,into=c("sector","area"),sep="/",
convert=TRUE)
sep_address
```

```
# Output
```

	empid	location	sector	area	date
1	101	Mumbai	4	Churchgate	2016-10-09
2	102	Delhi	12	Rohini	2010-11-01
3	103	Delhi	8	Pitampura	2009-09-23
4	104	Mumbai	21	Andheri	1990-02-30

- Here, we have split the column **address** into two new columns: **sector** and **area** with separator as '/'.
  - **sep=** is used to specify separator between columns. The default value is a regular expression that matches any sequence of non-alphanumeric values. Here, separator is assumed as '-'.



\* **colsplit()** from package **reshape2** give the same result as you get with **separate()**. Difference is **colsplit()** works only on a single column, we need to use **cbind()** to combine split columns with original data while **separate()** performs all the operations at once reducing the possibility of making mistakes.

# unite() with sep

**unite()** is a complement of **separate()**. It unites multiple columns into single column.

```
# Unite the 3 date columns into one
```

```
unite_date<-unite(sep_date,date,c(Year,Month,Date),sep="/") ←  
unite_date
```

```
# Output
```

	empid	Location	address	date
1	101	Mumbai	4/Churchgate	2016/10/09
2	102	Delhi	12/Rohini	2010/11/01
3	103	Delhi	8/Pitampura	2009/09/23
4	104	Mumbai	21/Andheri	1990/02/30

unite() takes the dataframe, name of the column to add, vector of columns to combine and a separator to use between the values as arguments

# tidyr vs reshape2

- As we have seen tidyr and reshape2 functions perform similar operations.
- reshape2 functions can do aggregation which is not possible with tidyr.
- tidyr is designed specifically for tidying data, while reshape2 is designed with a wider purpose of reshaping and aggregating.
- Therefore, we use **gather()** and **separate()** functions from tidyr to quickly tidy our data and **dcast()** function from reshape2 to aggregate them.

# Quick Recap

In this session, we learnt how to tidy our data using `tidyr` functions and what is the difference between package `tidyr` and `reshape2`. Here is the quick recap:

tidy functions

- **`gather()`**: converts wide data to longer format. It is similar to the **`melt()`** function from **`reshape2`** but can handle only dataframes.
- **`spread()`**: converts long data to wider format. It is similar to the **`dcast()`** function from **`reshape2`**.
- **`separate()`**: splits one column into two or more columns. it is similar to **`colsplit()`** function from **`reshape2`**.
- **`unite()`**: combines two or more columns into a single column.

apply Family Functions

# Contents

1. Introduction
2. apply()
3. lapply()
4. sapply()
5. mapply()
6. rapply()
7. tapply()
8. Related Functions

# Introduction

- The 'apply' family belongs to R base package and is populated with functions to manipulate slices of data from vector, matrices, array, lists and dataframes in a repetitive way.
- This family contains seven functions, all ending with apply: apply, lapply , sapply, vapply, mapply, rapply, and tapply.
- 'apply' family functions can be used as an alternative to **for loops**. These functions provide a compact syntax for sometimes rather complex tasks that is more readable and faster than poorly written loops.
- But how and when should we use this, depends on the structure of data we wish to operate on, and the format of the output we need.
- In this tutorial, we will demonstrate how to use these functions in R. They are extremely helpful, as you will see.

# apply()

This function returns a vector or array or matrix or list of values obtained by applying a function to margins of an array or matrix or dataframe.

**apply(X, MARGIN, FUN, ...)**

**X** is an array (a matrix if the dimension of the array is 2)

**MARGIN** is a vector defining how the function is applied: **MARGIN=1** indicate rows,

**MARGIN=2** indicates columns, **MARGIN=c(1,2)** indicates rows and columns

**FUN** is the function to be applied; can be any R function, including a User

Defined Function

**...** optional arguments to **FUN**

# apply()

Let's create a matrix and use **apply()** to find sum of each row

```
m<-matrix(c(10:19, 15:24), nrow=5, ncol=4)  
m
```

# Output

	[,1]	[,2]	[,3]	[,4]
[1,]	10	15	15	20
[2,]	11	16	16	21
[3,]	12	17	17	22
[4,]	13	18	18	23
[5,]	14	19	19	24

```
apply(m, 1, sum)  
[1] 60 64 68 72 76
```

- This command basically means, apply the function '**sum**' to the matrix **m** along margin **1**, i.e. by row, summing up the values of each row.
- A line vector is obtained containing the sum of the values of each row.

# lapply()

- This function is similar to **apply()**, but it takes a list or vector or dataframe as an input and returns a list as an output. The "l" in "lapply" stands for "list".
- **lapply()** becomes especially useful when dealing with data frames. R dataframe is considered as a list having variables as its elements. We can therefore apply a function to all the variables in a data frame by using the **lapply()** function.

Let's create a dataframe and obtain the sum of each variable in that dataframe.

```
a1<-c(10,20,30,40)
a2<-c(1,2,3,4)
a3<-c(100,200,300,400)
df<-data.frame(a1,a2,a3)
df
```

# Output

	a1	a2	a3
1	10	1	100
2	20	2	200
3	30	3	300
4	40	4	400

# lapply()

```
is.list(df) ←  
[1] TRUE  
  
x<-lapply(df,sum)  
x  
# Output  
$a1  
[1] 100  
  
$a2  
[1] 10  
  
$a3  
[1] 1000
```

**is.list()** returns TRUE or FALSE depending on whether its argument is a list or not.

Note that unlike in the **apply()** there is no margin argument since we are just applying the function to each component of the list.

# sapply()

- This function is a user friendly version; it works as `lapply()` but tries to simplify the output to the most elementary data structure that is possible. The "s" in "sapply" stands for "simplify".
- `sapply()` is a wrapper for `lapply()`, takes vector or list or dataframe as an input and returns a vector or matrix or list.

Let's create a list and apply `sapply()` to each element in the list.

```
list1<-list(a=10:14,b=15:19,c=20:24)
list1
```

# Output

```
$a
[1] 10 11 12 13 14

$b
[1] 15 16 17 18 19

$c
[1] 20 21 22 23 24
```

# sapply()

```
sapply(list1,sum) ← _____
```

a	b	c
60	85	110

- If we include arguments: **simplify=FALSE**, it will return the output as a list.
- **simplify="array"**, it will return the output as an array.

```
# Using a user defined function with multiple arguments
```

```
sapply(list1,function(x, y) sum(x) + y,y=5) ← _____
```

a	b	c
65	90	115

- Here, we have defined a function where we are adding **y** to sum of **x**.
- **x** refers to **list1** so it will add **y** i.e. 5 to all the elements of **list1**.

# mapply()

- **mapply()** is a multivariate version of **sapply()**.
- **mapply()** applies a function to multiple list or multiple vector arguments. It starts with first element of each argument first, followed by the second element, and so on

Let's create two vectors and apply **mapply()** on them.

```
a<-10:14  
b<-15:19  
mapply(sum,a,b)    ←  
[1] 25 27 29 31 33
```

Here, **mapply()** adds 10 with 15, 11 with 16, and so on.

# rapply()

- **rapply()** stands for recursive apply, and as the name suggests it applies function to all elements of a list recursively.

Let's understand this with a simple list example.

```
l<-list(a=1:5,b=6:10)  
rapply(l,log2) ←
```

- ❑ This command returns the log2 of each element of the list.
- ❑ Notice that the output is in the form of a vector, check the next example if you want to preserve the original structure.

a1	a2	a3	a4	a5	b1	b2
0.000000	1.000000	1.584963	2.000000	2.321928	2.584963	2.807355
b3	b4	b5				
3.000000	3.169925	3.321928				

```
rapply(l,log2,how="list") ←
```

```
$a  
[1] 0.000000 1.000000 1.584963 2.000000 2.321928  
  
$b  
[1] 2.584963 2.807355 3.000000 3.169925 3.321928
```

**how="list"(or "replace")** preserves the original structure. Otherwise, the default is to use **how="unlist"**, which results in a vector.

# tapply()

- **tapply()** can be thought of as a generalization to **apply()** that allows for “ragged” arrays, arrays where each row can have a different number of columns.  
This is very helpful when you’re trying to summarise a data set.
- **tapply()** takes input as a vector, tells R how to group the elements in the vector and applies specified function to each group of variable

Let's understand this with a simple example.

```
x <-c(10, 15, 20, 100, 150, 200, 1000, 1500, 2000)
groups<-c("x", "x", "x", "y", "y", "y", "z", "z", "z")
tapply(x, groups, sum)
```

x	y	z
45	450	4500

**x** is a numeric vector  
**groups** is a grouping variable(3 groups)

# Related Functions - rep()

## rep() Function:

This function replicates its values a specified number of times. It is often used in conjunction with 'apply' functions.

```
z<-c(1,3,6,8,10)
repz=rep(z,c(3,1,2,1,2)) ←
repz
```

```
[1] 1 1 1 3 6 6 8 10 10
```

Here, `rep()` replicates values of `z` as specified `c(3,1,2,1,2)`: three times the first, one time the second, two times the third and so on.

```
# create 3 x 3 matrix
```

```
m=mapply(rep,1:3,3) ←
```

```
m
```

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

`mapply()` vectorizes the action of the function `rep`.  
This command is a concise form of :  
`matrix(c(rep(1,3),rep(2,3),rep(3,3)),nrow=3,ncol=3)`

# Related Functions - sweep()

## sweep() Function:

This function allows you to “sweep” out the values of a summary statistic. It is often used with **apply()** to standardize arrays.

```
m<-matrix(2:13,4,3)
```

```
m
```

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

4 x 3 matrix is created

```
m_mean<-apply(m,2,mean)
```

```
m_mean
```

```
[1] 3.5 7.5 11.5
```

using **apply()** finding the means per column

## Related Functions - sweep()

```
m_sweep<-sweep(m,2,m_mean,FUN="-")  
m_sweep
```

```
[,1] [,2] [,3]  
[1,] -1.5 -1.5 -1.5  
[2,] -0.5 -0.5 -0.5  
[3,]  0.5  0.5  0.5  
[4,]  1.5  1.5  1.5
```

**sweep()** takes an array, MARGIN, summary statistics which is to be swept out, the function to be used to carry out the sweep as its arguments.

Using sweep(), we are centering each column of the matrix around mean i.e. taking the elements of the columns of the matrix m and subtracting the mean m\_mean from each of them.

# Quick Recap

In this session, we learnt how to use apply family in R and why we should do it. Here is the quick recap of functions used :

## apply family

- 'apply' family is populated with functions to manipulate slices of data from vector, matrices, array, lists and dataframes in a repetitive way.
- This family contains seven functions, all ending with apply & it is used as an alternative to for loops.

## apply()

- Returns a vector or array or matrix or list of values obtained by applying a function to margins of an array or matrix or dataframe.
- **apply(X, MARGIN, FUN, ...)**

## lapply()

- It takes a list or vector or dataframe as an input and returns a list as an output. The "l" in "lapply" stands for "list".

## sapply()

- **sapply()** is a wrapper for **lapply()**, takes vector or list or dataframe as an input and returns a vector or matrix or list. The "s" in "sapply" stands for "simplify."

# Quick Recap

`mapply()`

- **`mapply()`** is a multivariate version of **`sapply()`**.
- **`mapply()`** applies a function to multiple list or multiple vector arguments.

`rapply()`

- **`rapply()`** stands for recursive apply, it applies function to all elements of a list recursively.

`tapply()`

- **`tapply()`** can be thought of as a generalisation to **`apply()`**

Related Functions

- **`rep()`** replicates its values a specified number of times.
- **`sweep()`** allows you to “sweep” out the values of a summary statistic.

# Handling Missing Values

Detecting, Excluding and Imputing NA's

# Contents

1. Introduction
2. Replacing Missing Values with NA's while Importing
3. Detecting NA's
4. Excluding Missing Values from Analysis
5. Imputing Missing Values

# Introduction

- Missing values in data is a common phenomenon in real world problems and can impact the statistical analysis if not treated properly.
- You need to know the pattern of missingness and how to treat them is a requirement to reduce the bias and to produce accurate inference from data.
- Lets get familiar with the pattern of missingness and explore various options of how to deal with them.

# Missing Data Mechanism

Three types of missing data:

- Missing Completely at Random (MCAR):

MCAR happens when missingness is totally unrelated to other variables in the dataset. Example: Missing data on Gender

- Missing at Random (MAR):

MAR happens when the missingness is related to the information in your study.

Other variables (but not the variable that is missing itself) in the dataset can be used to predict missingness. For example: Missing data for variable income can be estimated using other variables such as experience and designation.

Most missing data imputation methods are based on MAR

- Missing not at Random (MNAR):

MNAR happens when data is missing for a specific segment. For example: Missing data for variable income for all senior managers.

# Data Snapshot

basic salary data consist salary of each employee with it's Location & Grade. The data has 12 rows and 6 columns with 2 missing values.

Variables								
Observations	First_Name	Last_Name	Grade	Location	ba	ms		
	Alan	Brown	GR1	DELHI	17990	16070		
	Agatha	Williams	GR2	MUMBAI	12390	6630		
	Columns	Description	Type	Measurement	Possible values			
	First_Name	First Name	character	-	-			
	Last_Name	Last Name	character	-	-			
	Grade	Grade	character	GR1, GR2	2			
Location	Location	character	DELHI, MUMBAI		2			
ba	Basic Allowance	numeric	Rs.	positive values				
ms	Management Supplements	numeric	Rs.	positive values				

# Replacing Missing Values with NA while Importing the Data

A missing value is one whose value is unknown. Missing values in R appears as NA. NA is not a string or a numeric value, but an indicator of missingness. Our data has two missing values, let's see what happens when we import this data in R.

```
# Import Data and check how R treats missing data while importing
```

```
salary_data<-read.csv("basic salary.csv",header=TRUE)  
salary_data
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
2	Agatha	Williams	GR2	MUMBAI	12390	6630
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
4	Ameet	Mishra	GR2	DELHI	14780	9300
5	Neha	Rao		MUMBAI	19235	15200
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
7	Aaron	Jones	GR1	MUMBAI	23280	13490
8	John	Patil	GR2	MUMBAI	13500	10760
9	Sneha	Joshi	GR1	DELHI	20660	NA
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Anup	Save	GR2	MUMBAI	11960	7880

Note `read.csv()` replaces blank fields in numeric column with NA and in character column it is kept as blank only.  
If any field contains string “NA”, then `read.csv()` assumes that the value is missing and converts it into NA.

# Replacing Missing Values with NA while Importing the Data

If we want R to recognize all missing values the same way( the "correct" way i.e. **NA**) when we import the data, we will use **na.strings=** argument.

```
salary_data<-read.csv("basic salary.csv",header=TRUE,na.strings="")  
salary_data
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
2	Agatha	Williams	GR2	MUMBAI	12390	6630
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
4	Ameet	Mishra	GR2	DELHI	14780	9300
5	Neha	Rao	<NA>	MUMBAI	19235	15200
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
7	Aaron	Jones	GR1	MUMBAI	23280	13490
8	John	Patil	GR2	MUMBAI	13500	10760
9	Sneha	Joshi	GR1	DELHI	20660	NA
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Anup	Save	GR2	MUMBAI	11960	7880

- ❑ **read.csv()** by default leaves blank fields in character variable as is, so we need to specify a blank string "" to **na.strings=** for R to treat it as missing value and convert it to **NA**.
- ❑ **na.strings=** takes a character vector of strings which are to be interpreted as **NA** values.
- ❑ The **<NA>** vs **NA** just means that some of our columns are character and some are numeric.

# Replacing Missing Values with NA while Importing the Data

Our data might employ a different string to signal missing values like this:

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
2	Agatha	Williams	GR2	MUMBAI	12390	6630
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
4	Ameet	Mishra	GR2	DELHI	14780	9300
5	Neha	Rao	missing	MUMBAI	19235	15200
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
7	Aaron	Jones	GR1	MUMBAI	23280	13490
8	John	Patil	GR2	MUMBAI	13500	10760
9	Sneha	Joshi	GR1	DELHI	20660	missing
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Anup	Save	GR2	MUMBAI	11960	7880

To proceed with the analysis we will have to convert these missing values to correct missing value notation i.e. **NA** for R to recognise these as missing values.

# Replacing Missing Values with NA while Importing the Data

```
# Convert 'missing' value to 'NA'
```

```
salary_data<-read.csv("basic salary.csv", header=TRUE,  
na.strings="missing")  
salary_data
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
2	Agatha	Williams	GR2	MUMBAI	12390	6630
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
4	Ameet	Mishra	GR2	DELHI	14780	9300
5	Neha	Rao	<NA>	MUMBAI	19235	15200
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
7	Aaron	Jones	GR1	MUMBAI	23280	13490
8	John	Patil	GR2	MUMBAI	13500	10760
9	Sneha	Joshi	GR1	DELHI	20660	NA
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Anup	Save	GR2	MUMBAI	11960	7880

**na.strings=** converts  
**'missing'** value to  
**'NA'**

# Detecting NA's

```
# Check whether our data has missing values or not
```

```
na.fail(salary_data)
```

```
Error in na.fail.default(salary_data) : missing values in object
```

**na.fail()** returns the object only if it contains no missing values otherwise it returns an error message indicating there is one or more missing values in the data.

```
# Check total missing values
```

```
sum(is.na(salary_data))
```

```
[1] 2
```

**is.na** for dataframe returns a logical matrix with the same dimensions as the data frame, and with dimnames taken from the row and column names of the data frame. Here, **sum()** returns the total no. of missing values in the data.

# Detecting NA's

```
# Check the count of NA's for each column
```

```
na_count<-sapply(salary_data,function(y) sum(is.na(y)))  
na_count
```

First_Name	Last_Name	Grade	Location	ba	ms
0	0	1	0	0	1

**sapply()** is a part of 'apply' family functions. It applies the given user defined function on the data specified.

```
# Check Number of missing data per column
```

```
summary(salary_data)
```

```
# Output
```

First_Name	Last_Name	Grade	Location	ba	ms
Aaron :1	Brown :1	GR1 :4	DELHI :5	Min. :11960	Min. : 6630
Adela :1	Chavan :1	GR2 :7	MUMBAI:7	1st Qu.:13472	1st Qu.: 7360
Agatha :1	Jones :1	NA's:1		Median :14270	Median :10760
Alan :1	Joshi :1			Mean :16155	Mean :11005
Ameet :1	Kolte :1			3rd Qu.:19239	3rd Qu.:14225
Anup :1	Mishra :1			Max. :23280	Max. :16070
(Other):6	(Other):6				NA's :1

Using **summary()** we can check how many NA's our data contains.

# Excluding Missing Values from Analysis

- When R encounters missing value, it attempts to perform the requested procedure and returns a missing (**NA**) value as a result. One way of dealing with missing values is to remove them while performing that procedure.

```
x<-c(10,30,12,NA, 9)  
mean(x)
```

```
[1] NA
```

```
# We can calculate mean by dropping missing value like in the next example.  
# remove missing value
```

```
mean(x,na.rm=TRUE)  
[1] 15.25
```

**na.rm=** if set to **TRUE** will remove all **NA**'s while performing the requested procedure. By default, it is set to **FALSE**.



na.rm= can be used with many R functions, such as mean, median, sd, var, and so forth.

# Excluding Missing Values from Analysis

Case wise deletion (complete case analysis) is the easiest way to deal with missing data. It simply removes all the cases with missing data anywhere in the data i.e. analyzing only the cases with complete data.

```
# Case wise deletion  
  
na.omit(salary_data)  
  
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
2	Agatha	Williams	GR2	MUMBAI	12390	6630
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
4	Ameet	Mishra	GR2	DELHI	14780	9300
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
7	Aaron	Jones	GR1	MUMBAI	23280	13490
8	John	Patil	GR2	MUMBAI	13500	10760
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Anup	Save	GR2	MUMBAI	11960	7880

**na.omit()** is used for case deletion.  
Here, **na.omit()** removes 2 rows that contains missing values



Complete case analysis is widely used method for handling missing data, and is a default method in many statistical packages. But it has limitations like it may introduce bias and some useful information will be omitted from analysis.

# Data Snapshot

Consumer preference data consist information about 73 respondents & their preferences about 7 attributes on scale of 1-Least Important to 5-Most Important.

Variables									
Observations	Sn	Gender	Color	Weight	Shape	Camera	Ram	Processor	Internet
	1	M	3	3	4	3	3	3	3
	Columns	Description		Type	Measurement		Possible values		
	Sn	Serial No. (Index Variable)		Character			-		
	Gender	Gender		Categorical	M;F		2		
	Color	Color		Categorical	1 to 5		5		
	Weight	Weight		Categorical	1 to 5		5		
	Shape	Shape		Categorical	1 to 5		5		
	Camera	Camera		Categorical	1 to 5		5		
	Ram	Ram		Categorical	1 to 5		5		
	Processor	Processor		Categorical	1 to 5		5		
	Internet	Internet		Categorical	1 to 5		5		

# Imputing Missing Values

- If the amount of missing data is very small relative to the size of the data then case wise deletion may be the best strategy in order not to bias the analysis, however deleting available data points deprives the data of some amount of information.
- Basically, we need to decide how we're going to use our missing data, if at all, then either remove cases from our data or impute missing values before wiping out potentially useful data points from our data and proceed with our analysis.
- Single imputation can be done by replacing missing values with the mean / median / mode (or any other procedure) of the other values in the variable.

In this tutorial we will primarily focus on performing single imputation .



Mean/ Median / mode imputations are simple but, like complete case analysis, can introduce bias on mean and deviation. Furthermore, they ignore relationship with other variables

# Imputing Missing Values

Treating missing values in the variable 'Processor'

```
# Import the data and  
# Check number of missing values for each variable
```

```
consumer_pref<-read.csv("consumerpreference.csv",header=TRUE)  
sapply(consumer_pref,function(y) sum(is.na(y)))
```

Sn	Gender	Color	Weight	Shape	Camera
0	0	0	1	0	1
Ram	Processor	Internet			
0	1	0			

Our data has 3 missing values.

```
# Treating missing values in variable 'Processor'  
# Median imputation
```

```
consumer_pref$Processor[is.na(consumer_pref$Processor)]<-median(consum  
er_pref$Processor,na.rm=TRUE)  
sum(is.na(consumer_pref$Processor))
```

[1] 0

Here, we are calculating median of the values in variable 'Processor' using **median()** and replacing the **NA** value with the median value. Now there are no missing values in variable 'Processor'.

# Imputing Missing Values

Treating missing values in the variable 'Camera'

```
# Check the 'Gender' value corresponding to the NA value
```

```
g<-subset(consumer_pref,is.na(consumer_pref$Camera),select=c(Gender))  
g  
9 M
```

Corresponding 'Gender' value to the **NA** in Variable 'Camera' is 'M'.

```
# Calculate median of values in variable 'Camera' whose corresponding  
# 'Gender' value is 'M'
```

```
g_subset_camera<-subset(consumer_pref,consumer_pref$Gender=="M",  
select=c(Camera))  
consumer_pref$Camera[is.na(consumer_pref$Camera)]<-apply(g_subset_camera,  
2,function(x) median(x,na.rm=TRUE))  
sum(is.na(consumer_pref$Camera))
```

[1] Using **subset()** we have subsetted the values of variable 'Camera' whose corresponding 'Gender' value is 'M'. Then, using **apply()** (because **g\_subset\_camera** is a dataframe), calculated the median of the subsetted values and replaced **NA** with that value. As a result, we are left with no missing values in variable 'Camera'.

# Imputing Missing Values

Treating missing values in the variable 'Weight'

```
# Replace the NA value in 'Weight' with the corresponding value  
# in the variable which is highly correlated with 'Weight'
```

```
cor(consumer_pref[3:9],use="pairwise.complete.obs")
```

	Color	Weight	Shape	Camera	Ram
Color	1.0000000	0.9172664	0.9421043	-0.2749990	-0.2779061
Weight	0.9172664	1.0000000	0.8656048	-0.3434737	-0.2978006
shape	0.9421043	0.8656048	1.0000000	-0.2297692	-0.2751586
Camera	-0.2749990	-0.3434737	-0.2297692	1.0000000	0.8326233
Ram	-0.2779061	-0.2978006	-0.2751586	0.8326233	1.0000000
Processor	-0.2319926	-0.2650182	-0.2296934	0.8522504	0.9049753
Internet	-0.2556812	-0.2839609	-0.2320419	0.8601564	0.8957823
	Processor	Internet			
Color	-0.2319926	-0.2556812			
Weight	-0.2650182	-0.2839609			
shape	-0.2296934	-0.2320419			
Camera	0.8522504	0.8601564			
Ram	0.9049753	0.8957823			
Processor	1.0000000	0.8631503			
Internet	0.8631503	1.0000000			

- Here, we are computing the correlation between the 7 attributes using **cor()** giving the vector of indices of those 7 attributes..
- use="pairwise.complete.obs"** uses the non-NA values when calculating the correlation between 7 attributes.

## Interpretation :

- We have found that 'Weight' is highly correlated with 'Color' as they have the highest correlation coefficient.

# Imputing Missing Values

```
s<-subset(consumer_pref,is.na(consumer_pref$Weight))  
s
```

	Sn	Gender	Color	Weight	Shape	Camera	Ram	Processor	Internet
10	10	M	5	NA	4	3	2	3	2

This command subsets the row which contains NA in the variable ‘Weight’.

```
consumer_pref$Weight[is.na(consumer_pref$Weight)]<-s$Color  
consumer_pref[10, ]
```

	Sn	Gender	Color	Weight	Shape	Camera	Ram	Processor	Internet
10	10	M	5	5	4	3	2	3	2



This command replaces NA in ‘Weight’ with corresponding value of highly correlated variable i.e. ‘Color’

# Quick Recap

In this session, we learnt how to deal with different types of missing values. Here is the quick recap:

## Replacing missing values with NA

- Blank fields in numeric column are replaced with NA and in character column it is kept as blank only while importing data.
- **na.strings=**: to specify the character vector of strings which are to be treated as NA

## Recoding values to missing

- **na.fail**: returns the object if it does not contain any missing values otherwise it gives error indicating the missingness in data.
- **is.na()**: returns the logical matrix which indicates which elements are missing.
- **sapply()**: using this function we can calculate the count of NA's per column
- **summary()**: check number of NA's per variable

## Excluding missing values from analysis

- **na.rm=TRUE**: drops missing value
- **na.omit()**: performs case wise deletion

## Imputing missing values

- Single imputation can be done by replacing missing values with the mean / median / mode (or any other procedure) of the other values in the variable

# Measures of Central Tendency & Variation

# Contents

1. Sources & Types of Data
2. Data Measurement Scales
3. Measures of Central Tendency
  - i. Mean, Median, Mode
  - ii. Trimmed Mean
5. Measures of Variation
  - i. Range, Inter Quartile Range, Standard Deviation
  - ii. Coefficient of Variation
7. Measures of Central Tendency in R
8. Measures of Variation in R

# Sources of Data

## 1. Primary data

- Data collected by the investigator himself/herself for a specific purpose.
- Direct method of data collection.
- Eg. Data collected for research through questionnaires, interviews.

## 2. Secondary data

- Data collected by someone else for some other purpose (but being used by the investigator for another purpose).
- Indirect method of data collection.
- Eg. Census data being used to study the impact of education on income.

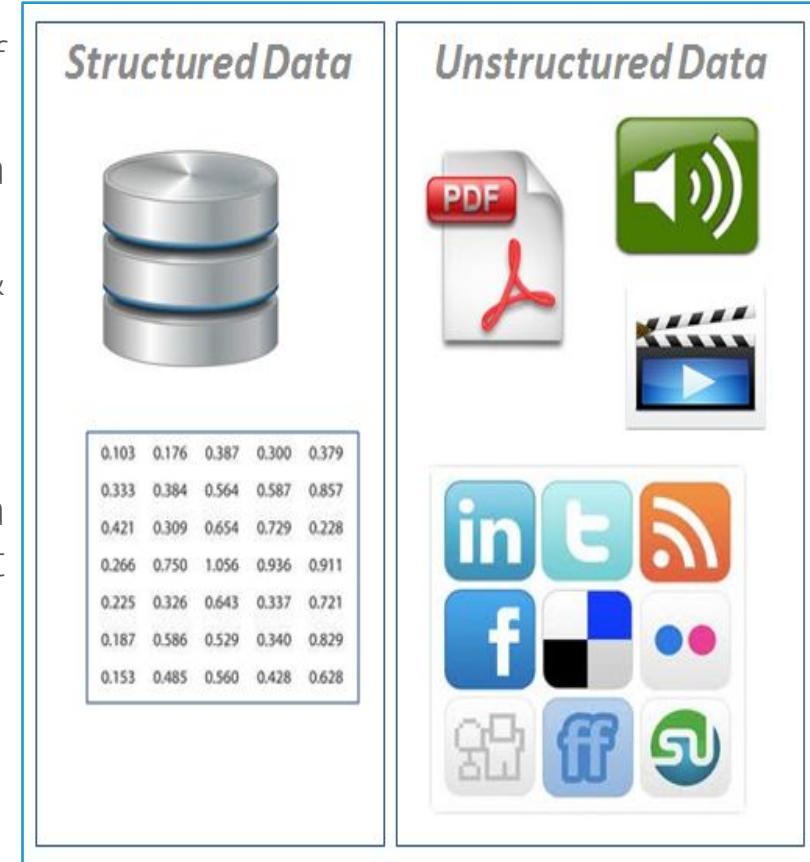
# Types of Data

## 1. Structured data

- Information is stored with high degree of organisation.
- Contains qualitative data, quantitative data or a mixture of both.
- Eg. Data arranged in Excel file in rows & columns

## 2. Unstructured data

- Information that either does not have a pre-defined data model and/or is not organized in a pre-defined manner.
- Eg. E-mails, tweets, blogs etc.



# Measurement Scales

## 1. Nominal scale

- Placing of data into categories without any order or structure.
- No numerical relationship between categories even if numbers are used for representation.
- Eg. Gender, nationality, language, region etc.

## 2. Ordinal scale

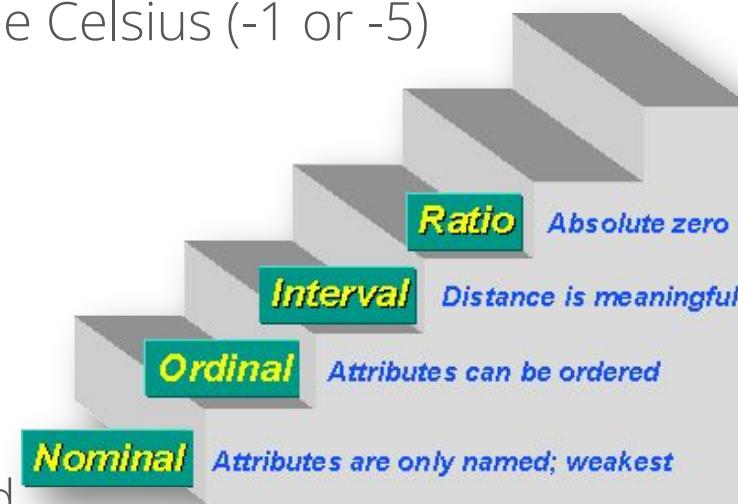
- Placing of data into categories such that order of values is meaningful but relative degree of difference is not known.
- Eg. Ranking the features of a product on a scale of 1 to 5.
- Likert scale: Psychometric scale commonly used in questionnaires.

Highly Dissatisfied	Dissatisfied	Neutral	Satisfied	Highly Satisfied
1	2	3	4	5

# Measurement Scales

## 3. Interval scale

- Numeric scale in which the order as well as the relative difference between values is known.
- No “true zero”
- Eg. Temperature can be below 0 degree Celsius (-1 or -5)



## 4. Ratio scale

- Numeric scale with an absolute “zero”.
- Addition, subtraction, multiplication and division are all valid operations.
- Eg. Height, Weight, Age ,etc will always be measured from 0 to maximum not below 0.

# Measurement Scales

Respondent	Gender	Region	Age	Satisfaction Level
1	M	1	23	3
2	M	2	45	4
3	M	2	33	3
4	F	2	25	4
5	F	3	37	2
6	M	1	35	1
7	M	2	41	5
8	F	3	27	2

Data

Region	1	Mumbai
	2	Delhi
	3	Kolkata
Satisfaction Level	1	Highly dissatisfied
	2	dissatisfied
	3	Neutral
	4	Satisfied
	5	Highly satisfied

Description

Gender: Nominal  
Region: Nominal  
Age: Ratio  
Satisfaction Level: Ordinal

# Measures of Central Tendency

Measure of Central Tendency (a.k.a. Measures of Central Location) :

- It is a single value that describe a set of data by identifying the central position within that set of data.

Most commonly used measures of central tendency are :

Mean	Arithmetic Mean. Commonly known as Average.  It is the sum of all values of the variable divided by the total number of values.
Median	Arrange the data in ascending order, Median is the middle value, if N is odd.  If N is even, it is average of two middle values.
Mode	It is the most frequently occurring observation in a set of data.



Note : The mean, median and mode are all valid measures of central tendency, but under different conditions, some measures of central tendency become more appropriate to use than others.

# Calculating Mean, Median, Mode

Consider marks of 12 students in an examination

13, 20, 16, 17, 09, 18, 17, 11, 08, 17, 12, 20

Now, Mean is the sum of all values of the variable divided by the total number of values

$$\frac{\sum_{i=1}^n x_i}{n} = \frac{13+20+16+17+09+18+17+11+08+17+12+20}{12} = \frac{178}{12} = 14.83$$

Here, N is even, Median is average of two middle values after arranging the data in ascending order

Data in Ascending order : 08, 09, 11, 12, 13, 16, 17, 17, 17, 18, 20, 20

Average of middle two values :  $\frac{16+17}{2} = 16.5$

Mode is the most frequently occurring observation in a set of data. Therefore, here, Mode is  
17

MEAN 14.83 Average marks scored by the students

MEDIAN 16.5 Half of the students have scored above and half below this

MODE 17 Marks scored by maximum students

# Trimmed Mean

It is recommended to report 'Trimmed Mean' along with mean if outliers are present in the data.

Trimmed mean excludes extreme data points for the calculation of mean. Typically, 5% data points (5% at each end) are excluded.

Note that trimmed mean will give robust estimate if underlying distribution is symmetric.

# Get an Edge!

## Best Measure of Central Tendency

Type <b>of Variable</b>	Best Measure
Nominal	Mode
Ordinal	Median
Interval/Ratio (Symmetric)	Mean
Interval/Ratio (Not Symmetric)	Median

- Mean is appropriate when the distribution is symmetric. For symmetric distribution, the mean is at the centre.
- For a skewed (not symmetric) distribution, mean is generally not at the centre. Median is better measure of central tendency for a skewed distribution.

# Measures of Variation

Measure of Dispersion : In addition to a measure of central tendency, it is desirable to have a measure of dispersion (variation) of data.

- A measure of dispersion is an indication of the spread of measurements around the center of the distribution.
- Two data sets can have equal mean (measure of central tendency) but vastly different variability.
- Eg. Score of Batsman A = (78,62,73,54,76,77) & Score of Batsman B = (92,8,78,34,109,99)

So Average scores of two batsmen in 6 innings is equal(=70) whereas Spread around mean is not identical.

Most commonly used measures of variation are :

- Range
- Inter-Quartile Range (IQR)
- Standard Deviation

# Range, IQR, SD

Range	Inter Quartile Range (IQR)	Variance and Standard Deviation (SD)
Most simple measure of variation. The difference between the highest and lowest values is termed as the range.	The interquartile range is the range between the lower quartile and the upper quartile.	The variance is based on "Sum of squares of deviations from mean"(Say SS). The Variance is SS divided by n.
Range is a crude measure as it does not take into account all values (Except the highest and the lowest). It has same units as the original values.	Quartiles are the values which divide the data in 4 equal parts. The values that divide each part are called the first, second, and third quartiles; and they are denoted by Q1, Q2, and Q3, respectively It has same units as the original values.	The standard deviation is the positive square root of the variance. It has same units as the original values.

# Calculating Range, IQR, SD

Consider, the distribution of marks of 12 students in an examination

13, 20, 16, 17, 09, 18, 17, 11, 08, 17, 20, 12

$$\text{Range} = X_{max} - X_{min} : 20 - 08 = 12$$

Data in Ascending order : 08, 09, 11, 12, 13, 16, 17, 17, 17, 18, 20, 20

$Q_1$  is the  $n/4^{\text{th}}$  value : 3<sup>rd</sup> Value= 11

$Q_3$  is the  $3(n/4)^{\text{th}}$  value : 9<sup>th</sup> Value= 18

$$IQR = Q_3 - Q_1 : 18 - 11 = 7$$

Here :  $\bar{x}$  is the mean = 14.83

$$S^2 = \frac{\sum(x_i - \bar{x})^2}{n} = \frac{(13 - 14.83)^2 + \dots + (20 - 14.83)^2 + (12 - 14.83)^2}{12} = 15.47$$

Therefore, Standard Deviation :  $S = \sqrt{S^2} = 3.93$

# Coefficient of Variation (CV)

As variance has same units as that of the variable, it is inappropriate to use variance to compare two data sets having different units. Hence, there is a need of a quantity without unit like Coefficient of Variation (CV) for effective comparison.

CV is a relative measure of variation and is used to compare variability in two data sets.

The CV is defined as "Standard Deviation divided by Mean" and is generally expressed as a percentage.

Higher the value of CV, more is the variability.  
CV is sometimes referred to as "Relative Standard Deviation".

# Case Study - 1

## Objective

- To compare the performance of two batsmen using the measures of central tendency and measure of variation

## Available Information

- Runs scored by two batsman A and B in 6 matches

Runs Scored

Batsman A	Batsman B
78	92
62	8
73	78
54	34
76	109
77	99

# Observation and Conclusion

Batsman A	Batsman B
78	92
62	8
73	78
54	34
76	109
77	99
<b>MEAN = 70</b>	<b>MEAN = 70</b>
<b>CV = 13.97%</b>	<b>CV = 57.32%</b>

- Average scores of two batsmen in 6 innings is equal(=70) but the spread around mean is not identical.
- We can see that variability in performance of Batsman B is more than that of Batsman A. Hence, we can infer that Batsman A is a more consistent performer than Batsman B.

# Case Study - 2

To learn Descriptive Statistics in R, we shall consider the below case as an example.

## Background

Data of 100 retailers in platinum segment of the FMCG company.

## Objective

To describe the variables present in the data

## Sample Size

Sample size: 100

Variables: Retailer, Zone, Retailer\_Age, Perindex, Growth, NPS\_Category

# Data Snapshot

## Retail Data

### Variables

Retailer	Zone	Retailer_Age	Perindex	Growth	NPS_Category
1	North	<=2	81.84	3.04	Promoter
<b>Observations</b>					
Columns	Description	Type	Measurement	Possible values	
Retailer	Retailer ID	numeric	-	-	
Zone	Location of the retailer	character	East, West, North, South	4	
Retailer_Age	Number of years doing business with the company	character	<=2, 2 to 5, >5	3	
Perindex	Index of performance based on sales, buying frequency and buying recency	numeric	-	positive values	
Growth	Annual sales growth	numeric	-	positive values	
NPS_Category	Category indicating loyalty with the company	character	Detractor, Passive, Promoter	3	

# Describing Variables in R

```
#Importing Data
```

```
retail_data <-read.csv("Retail_Data.csv" header=TRUE)
```

```
#Checking the variable features using summary function
```

```
summary(retail_data)
```

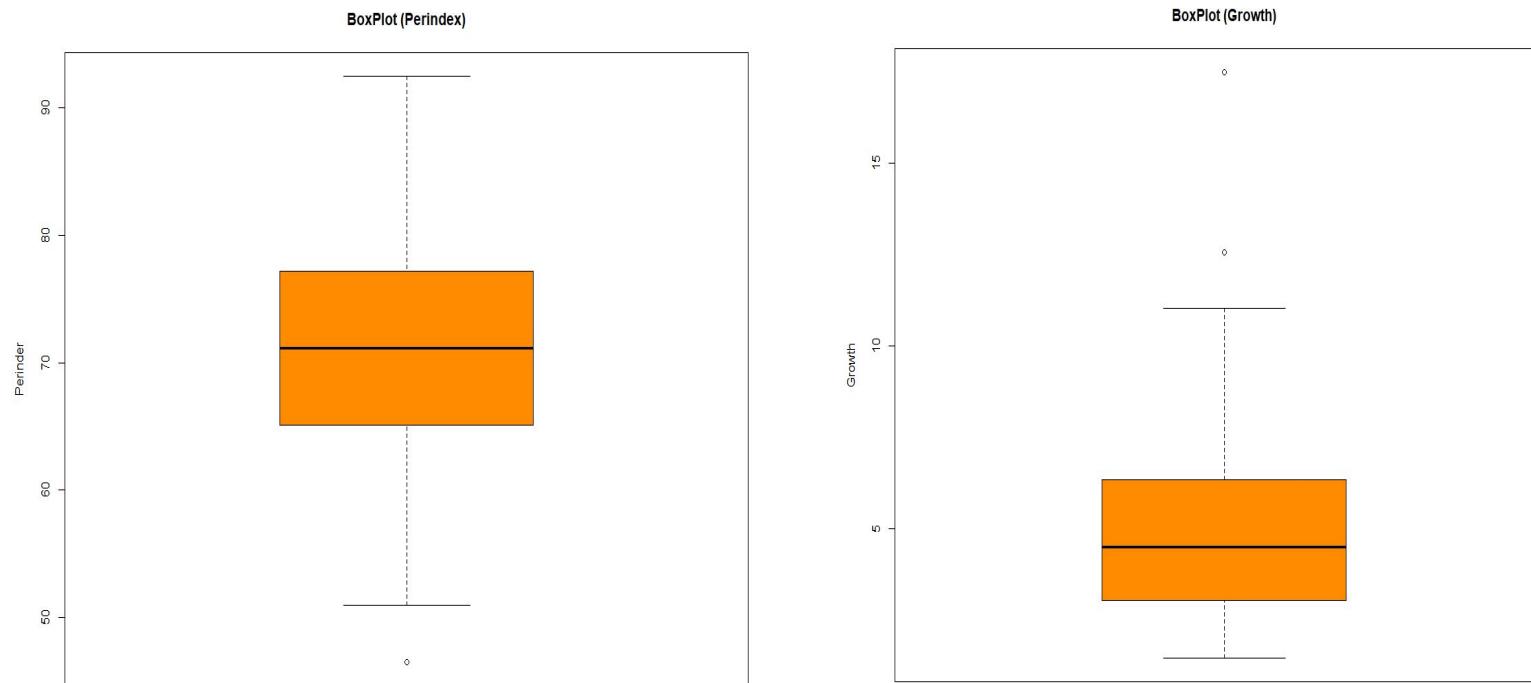
```
# Output
```

Retailer	Zone	Retailer_Age	Perindex	Growth	NPS_Category
Min. : 1.00	East :15	<=2 :17	Min. :46.53	Min. : 1.470	Detractor:23
1st Qu.: 25.75	North:25	>5 :56	1st Qu.:65.08	1st Qu.: 3.058	Passive :41
Median : 50.50	South:32	2 to 5:27	Median :71.15	Median : 4.495	Promoter :36
Mean : 50.50	West :28		Mean :70.50	Mean : 5.153	
3rd Qu.: 75.25			3rd Qu.:77.17	3rd Qu.: 6.340	
Max. :100.00			Max. :92.49	Max. :17.500	
			NA's :1		

# Understanding Data through Visualisation

```
boxplot(retail_data$Perindex, data= retail_data, main = "BoxPlot  
(Perindex)",ylab = "Perindex",col = "darkorange")
```

```
boxplot(retail_data$Growth, data= retail_data, main = "BoxPlot  
(Growth)",ylab = "Growth",col = "darkorange")
```



Here we can see that Perindex variable is distributed symmetrically whereas Growth variable is Positively Skewed.



The concept of Skewness is explained in detail in next presentation

# Measures of Central Tendency in R

```
# Mean for Perindex & Growth Variables
```

```
mean(retail_data$Perindex)
```

```
[1] NA
```

mean() in R, gives mean of the variable.

```
mean(retail_data$Perindex,na.rm = T)
```

```
[1] 70.49697
```

Using **na.rm=T** excludes the missing values from the mean

```
mean(retail_data$Growth,na.rm = T)
```

```
[1] 5.1528
```

```
# Median for Perindex & Growth Variables
```

```
median(retail_data$Perindex,na.rm = T)
```

```
[1] 71.15
```

median() in R, gives median of the variable.

```
median(retail_data$Growth,na.rm = T)
```

```
[1] 4.495
```

So as we have seen, Perindex Variable is symmetric, hence it's mean value is considered whereas for Growth Variable which is Positively Skewed, Median would be a better measure.

# Measures of Central Tendency in R

```
# Trimmed Mean
```

```
trimmed_mean_PI <- mean(retail_data$Perindex,0.10,na.rm=T)  
trimmed_mean_PI
```

```
[1] 70.5842
```

Using 0.10 in the **mean()**, excludes 10% observations from each side of the data from the mean

```
trimmed_mean_G <- mean(retail_data$Growth,0.10,na.rm = T)  
trimmed_mean_G
```

```
[1] 4.825
```

# Measures of Central Tendency in R

```
# Measure of Central Tendency for Categorical Variable  
# Mode using Frequency Table
```

```
freq <- table(retail_data$Zone) ← table() in R, gives the frequency of counts of the  
variable mentioned.
```

```
freq
```

	East	North	South	West
freq	15	25	32	28

Here Mode is 32 as the frequency is highest for South Zone.

# Measures of Dispersion in R

```
# Range, Difference & Inter Quartile Range
```

```
r_PI <- range(retail_data$Perindex,na.rm = T)  
r_PI  
[1] 46.53 92.49
```

range() in R, gives minimum and maximum values of that variable

```
r_G <- range(retail_data$Growth,na.rm = T)  
r_G  
[1] 1.47 17.50
```

```
diff(r_PI)  
[1] 45.96
```

diff() calculates difference between all values of that vector

```
diff(r_G)  
[1] 16.03
```

```
IQR(retail_data$Perindex,na.rm = T)  
[1] 12.095
```

IQR() in R gives the Inter-Quartile range of the variable

```
IQR(retail_data$Growth,na.rm = T)  
[1] 3.2825
```

# Measures of Dispersion in R

```
# Standard Deviation/ Variance
```

```
sd(retail_data$Perindex,na.rm = T) ←
```

```
[1] 9.569232
```

sd() in R, gives standard deviation of the variable

```
sd(retail_data$Growth)
```

```
[1] 2.620525
```

```
var(retail_data$Perindex,na.rm = T) ←
```

```
[1] 91.5702
```

var() in R, gives variance of the variable

```
var(retail_data$Growth)
```

```
[1] 6.867152
```

```
# Coefficient of Variation
```

```
cv_PI <- sd(retail_data$Perindex,na.rm = T)/
```

```
mean(retail_data$Perindex,na.rm = T)
```

```
cv_PI
```

```
[1] 0.1357396
```

There is no standard function for CV in R. Hence we calculate it by definition.

```
cv_G <- sd(retail_data$Growth)/mean(retail_data$Growth)
```

```
cv_G
```

```
[1] 0.5085633
```

# Quick Recap

In this session, we learnt the basics of Descriptive Statistics :

## Measures of Central Tendency

- Mean
- Median
- Mode

## Measures of Variation

- Range
- Inter-Quartile Range
- Variance/ Standard Deviation

## Appropriate Measures to Report

- Trimmed Mean when symmetric data has outliers
- Median when data is not symmetric and has outliers
- Coefficient of Variation

# Beyond Mean & Variance

# Contents

- 1. Skewness
  - i. Visualizing Skewness
  - ii. How to Calculate Skewness
- 1. Kurtosis
  - ii. Visualizing Kurtosis
  - iii. How to Calculate Kurtosis
- 1. Moments
- 2. Skewness and Kurtosis in R
- 3. Application Areas

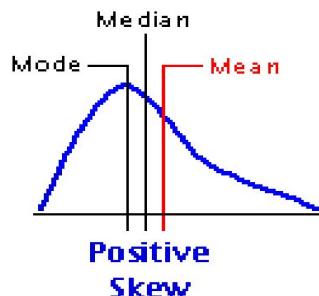
# Skewness

Skewness gives us the Shape of the data. It is the 'Lack of Symmetry'

## Positively Skewed

- Right Tail is longer
- Mass of the distribution is concentrated on the left

$$\text{Mode} < \text{Median} < \text{Mean}$$

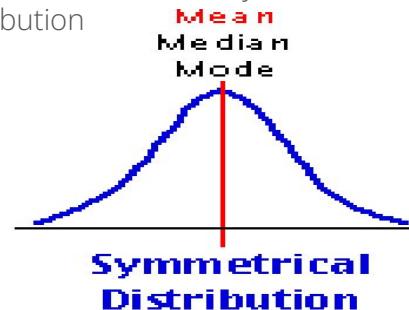


## Symmetric

- Both tails are equal
- Mass of the distribution is equally distributed

$$\text{Mean} = \text{Median} = \text{Mode}$$

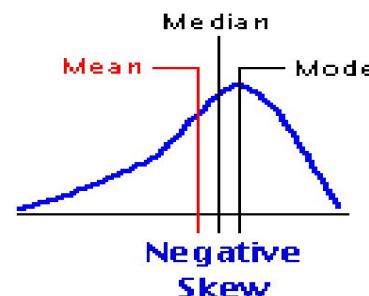
Normal Distribution is symmetric distribution



## Negatively Skewed

- Left Tail is longer
- Mass of the distribution is concentrated on the right

$$\text{Mean} < \text{Median} < \text{Mode}$$



# How to Calculate Skewness

$$\frac{(Mean - Mode)}{Standard Deviation}$$

Pearson Measure of Skewness

$$\frac{3(Mean - Median)}{Standard Deviation}$$

Pearson Measure of Skewness  
(Alternative Form)

\* where  $\frac{(Q_3 - Q_2) - (Q_2 - Q_1)}{Q_3 - Q_1} = \frac{Q_1 - 2Q_2 + Q_3}{Q_3 - Q_1}$  and  
Quartile,  $Q_1$ =First Quartile,  $Q_2$ =Second Quartile,  $Q_3$ = Third Quartile

\* where  $Q_1$ =First Quartile,  $Q_2$ =Second Quartile,  $Q_3$ = Third Quartile

Bowley's Coefficient of Skewness

## Skewness Based on Third Moment

- The most widely used measure of skewness is based on the third moment.

$$\frac{n}{(n - 1)(n - 2)} \sum \left( \frac{x_j - \bar{x}}{s} \right)^3$$

- Any threshold or rule of thumb is arbitrary, but here is one: If the skewness is greater than 1.0 (or less than -1.0), the skewness is substantial and the distribution is far from symmetrical. Value 'zero' indicates symmetric distribution.

# Kurtosis

Kurtosis is defined as a measure of 'peakedness'. It is generally measured relative to Normal distribution. (Which means 'excess of kurtosis' is measured)

Mesokurtic

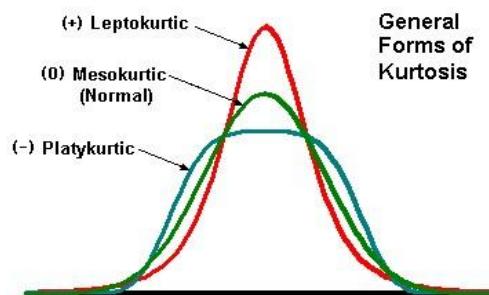
Normal distribution is termed as mesokurtic distribution

Leptokurtic

A leptokurtic distribution has a more acute peak.  
(positive kurtosis)

Platykurtic

A platykurtic distribution has a flatter peak.  
(negative kurtosis)



# How to Calculate Kurtosis

$$\frac{n(n + 1) \sum_{i=1}^n (x_i - \bar{x})^4}{(n - 1)(n - 2)(n - 3)s^4} - \frac{3(n - 1)^2}{(n - 2)(n - 3)}$$

Measure of Kurtosis

Value zero indicates peakedness is same as that of normal distribution.

# Moments

Moments are the constants which help us in knowing the characteristics of population and the graphic shape of a data.

$k^{th}$  raw moment ( $\mu'_k$ ) =

$$\sum_{i=1}^n \frac{x_i^k}{n}$$

$k^{th}$  central moment ( $\mu_k$ ) =

$$\sum_{i=1}^n \frac{(x_i - \bar{x})^k}{n}$$

The moments about zero are called RAW MOMENTS

The moments about mean are called CENTRAL MOMENTS

# Get an Edge!

- Skewness :
  - If bulk of the data is at the left and the right tail is longer, distribution is skewed right or positively skewed.
  - If bulk of the data is at the right and the left tail is longer, distribution is skewed left or negatively skewed.
  - If skewness is less than  $-1$  or greater than  $+1$ , the distribution is highly skewed.
  - If skewness is between  $-1$  and  $-\frac{1}{2}$  or between  $+\frac{1}{2}$  and  $+1$ , the distribution is moderately skewed.
  - If skewness is between  $-\frac{1}{2}$  and  $+\frac{1}{2}$ , the distribution is approximately symmetric.
- Kurtosis :
  - A distribution with kurtosis  $\approx 3$  (excess  $\approx 0$ ) is called mesokurtic.
  - A distribution with kurtosis  $< 3$  (excess kurtosis  $< 0$ ) is called platykurtic.
  - A distribution with kurtosis  $> 3$  (excess kurtosis  $> 0$ ) is called leptokurtic.

# Case Study

To learn Descriptive Statistics in R, we shall consider the below case as an example.

## Background

Data of 100 retailers in platinum segment of the FMCG company.

## Objective

To describe the variables present in the data

## Sample Size

Sample size: 100

Variables: Retailer, Zone, Retailer\_Age, Perindex, Growth,  
NPS\_Category

# Data Snapshot

Variables							
Retail Data		Retailer	Zone	Retailer_Age	Perindex	Growth	NPS_Category
		1	North	<=2	81.84	3.04	Promoter
Observations		Columns	Description	Type	Measurement	Possible values	
		Retailer	Retailer ID	numeric	-	-	
		Zone	Location of the retailer	character	East, West, North, South	4	
		Retailer_Age	Number of years doing business with the company	character	<=2, 2 to 5, >5	3	
		Perindex	Index of performance based on sales, buying frequency and buying recency	numeric	-	positive values	
		Growth	Annual sales growth	numeric	-	positive values	
		NPS_Category	Category indicating loyalty with the company	character	Detractor, Passive, Promoter	3	

# Skewness and Kurtosis in R

```
#Importing Data  
retail_data <-read.csv("Retail_Data.csv", header=TRUE)
```

We have already seen that Growth variable is Positively Skewed, so we'll find out skewness & kurtosis value for the same

```
library(e1071) ← Using package "e1071" in R is the easiest way  
# Skewness to find skewness and kurtosis
```

```
skewness(retail_data$Growth,type = 2) ←  
[1] 1.591236
```

❑ **skewness()** gives skewness of the variable.  
❑ **type=2** uses moment based formula.

```
# Kurtosis
```

```
kurtosis(retail_data$Growth,type = 2) ←  
[1] 4.283886
```

❑ **kurtosis()** gives kurtosis of the variable.  
❑ **type=2** uses moment based formula.

```
# Skewness and Kurtosis by Zone
```

```
f <- function(x)c(skew = skewness(x,type = 2),kurt = kurtosis(x,type = 2))  
aggregate(Growth~Zone,data = retail_data,FUN = f)
```

	Zone	Growth.skew	Growth.kurt
1	East	1.15293909	0.75091158
2	North	-0.04046698	-1.06571086
3	South	2.36833028	6.88519638
4	West	0.64121875	-0.58961827

# Application Areas

Questions like :

- What is the shape of my data? Where are my data values concentrated and what is its spread?

## Skewness

It is generally used to check how close the data is to normal distribution.  
- It is used to decide appropriate statistical measure

## Kurtosis

It is used to see the extent to which the data is concentrated about its mean.  
- Not commonly reported but very helpful to assess the distribution of variable under study.

# Quick Recap

In this session, we learnt the basics of knowing the Shape of the Data.

## Skewness

- Measures of Skewness
- Visualizing and Interpreting Skewness

## Kurtosis

- Measures of Kurtosis
- Visualizing and Interpreting Kurtosis

# Bivariate Relationships

# Contents

1. Describing a Bivariate Relationship
2. Scatterplot
  - i. What is Scatterplot?
  - ii. Interpreting Scatterplot
3. Pearson's Coefficient of Correlation
4. Line of Best Fit : Regression Line
5. Relationships and r
6. Simple Linear Regression
7. Application Areas
8. Scatterplot in R
9. Pearson's Coefficient of Correlation in R
10. Simple Linear Regression in R
11. Summarising two categorical variables

# Describing a Bivariate Relationship

We have so far studied how do we describe and study Univariate Data, that is data having only one variable.

Now we shall study to describe a Bivariate data, that is a data having two variables. The Bivariate data can either have :

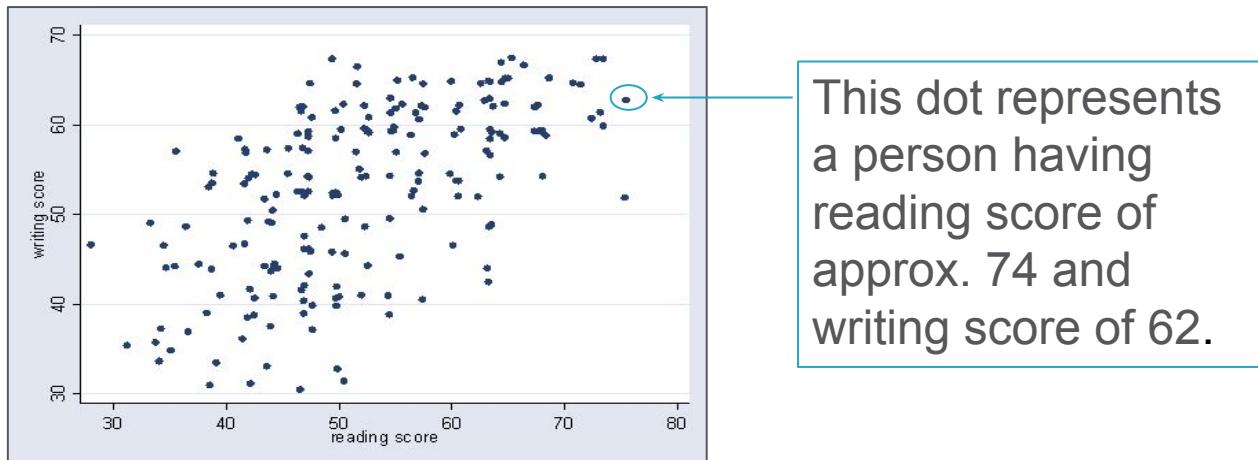
- Two Numeric Variables
- Two Categorical Variables
- One Numeric and One Categorical Variable

The relationship between two numeric continuous variables can be described using :

- **Scatter Plot** : Scatter plot provides nature of relationship graphically
- **Co-Relation Coefficient** : Correlation coefficient measures degree of linear relationship
- **Simple Linear Regression** : Simple Linear Regression gives equation of the type  
$$Y=a +bX$$
 where in you can also predict the value Y for any given value of X.

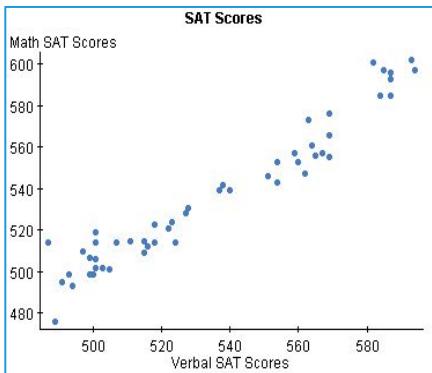
# What is a Scatterplot?

- A scatter plot consists of a X axis (the horizontal axis), a Y axis (the vertical axis), and a series of dots.
- The X-axis and Y-axis represent the values of one variable each.
- Each dot on the scatterplot is one observation from a data set representing the corresponding variable value on X and Y axis respectively
- This plot can be used only for two numeric continuous variables

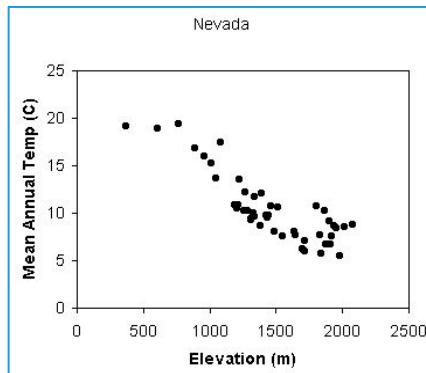


# Interpreting a Scatterplot

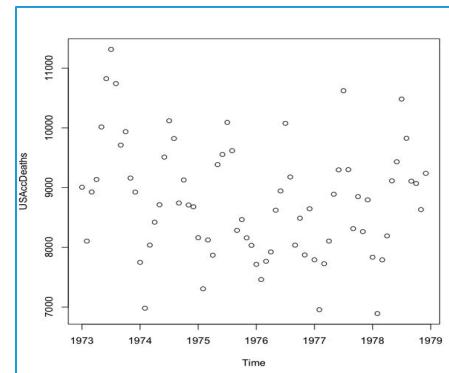
Positive Correlation



Negative Correlation



No Correlation



This is a positive sloping (upward) graph.  
As the value of one variable increases, the value of other variable also increases.

This is a negative sloping (downward) graph.  
As the value of one variable increases, the value of other variable tends to decrease.

This is a graph with random pattern.  
There is no connection between the two variables. If value of one variable increases, other might increase/decrease.

# Pearson's Coefficient of Correlation

The Pearson's correlation coefficient numerically measures the strength of a linear relation between two variables

$$r = \frac{\sum(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum(X_i - \bar{X})^2} \sqrt{\sum(Y_i - \bar{Y})^2}} = \frac{\text{cov}(X, Y)}{sd(x)sd(y)}$$

RANGE $-1 \leq r \leq 1$	
Positive Correlation	$r > 0$
Negative Correlation	$r < 0$
No Correlation	$r = 0$

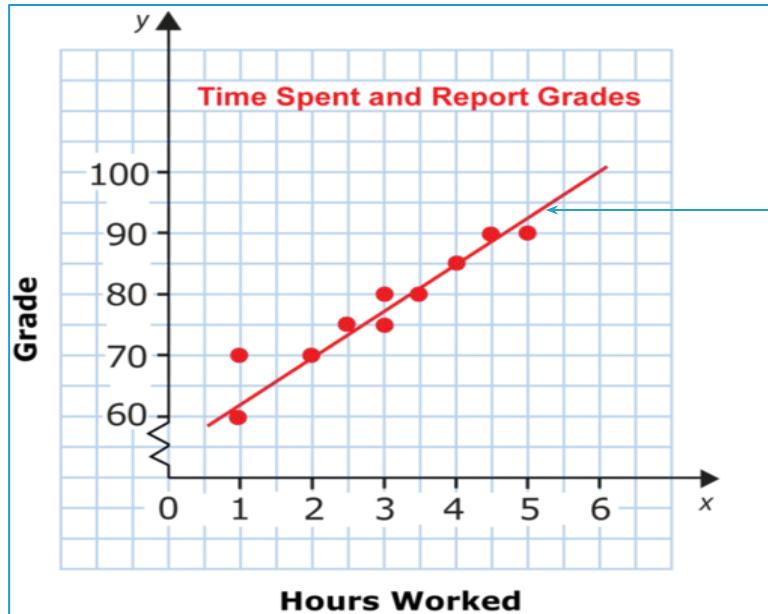
- The two variables can be measured in entirely different units.
- Example, you could correlate a person's age with their blood sugar levels. Here, the units are completely different.
- It is not affected by change of Origin and Scale



Both Covariance and Pearson's correlation coefficient can be used only for continuous Numeric variables

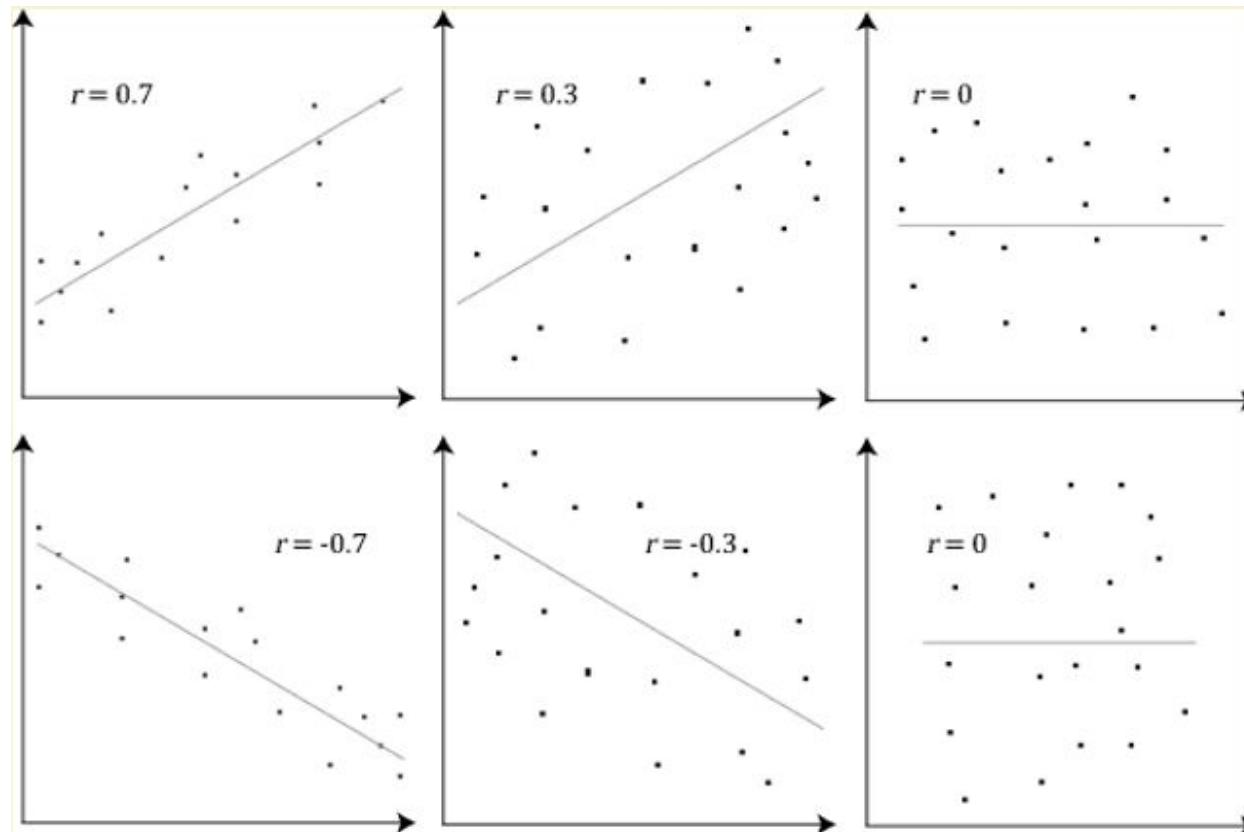
# Line of Best Fit : Regression Line

A line of best fit (or "trend" line) is a straight line that best represents the data on a scatter plot.



**Line of Best Fit :**  
This line may pass through some of the points, none of the points, or all of the points.

# Relationships and r



# Simple Linear Regression

The equation of line of best fit is used to describe relationship between two variables

Mathematical form of simple linear regression :

$$Y = aX + b + e$$

Where,

a : Intercept (The value at which the fitted line crosses the y-axis i.e. X=0)

b : Slope of the Line

e : error which is assumed to be a random variable

NOTE : a and b are population parameters which are estimated using sample

Here, variable Y is known as a 'Dependent' variable, that 'depends on' X which is known as the 'Independent' variable.

# Application Areas

## Scatter Plot

It is useful in visualising the relationship between any two variables as an initial step.

- Life expectancy and the number of cigarettes smoked per day
- Literacy rate and life expectancy in a particular region

## Correlation Coefficient

It gives the exact numeric measure of the extent of bivariate relationship.

- Distance between home & office and the time taken to get there
- Size of car engine and cost of car insurance

## Simple Linear Regression

It is very useful in predicting the value of one variable given the value of another in a bivariate scenario.

- Number of bedrooms and cost of home insurance
- Scores in the final exam given the scores in mock test

# Case Study - 1

## Background

- A company conducts different written tests before recruiting employees. The company wishes to see if the scores of these tests have any relation with post-recruitment performance of those employees.

## Objective

- To study the correlation between Aptitude and Job Proficiency.
- Predict the Job proficiency for a given Aptitude score.

## Available Information

- Sample size is 33
- Independent Variables: Scores of tests conducted before recruitment on the basis of four criteria – Aptitude, Test of English, Technical Knowledge, General Knowledge
- Dependent Variable job\_prof: Job Performance Index calculated after an employee finishes probationary period (6 months)

# Data Snapshot

Job\_Proficiency

**Variables**

empno	aptitude	testofen	tech_	g_k	job_prof
1	86	110	100	87	88
2	62	62	99	100	80
3	110	107	103	103	96
4	101	117	93	95	76
5	100	101	95	88	80
6	78	85	95	84	73
7	120	77	80	74	58
8	105	122	116	102	116

Columns	Description	Type	Measurement	Possible values
Empno	Employee Number	numeric	-	positive values
aptitude	Aptitude Score of the Employee	numeric	-	positive values
Testofen	Test of English	numeric	-	positive values
tech_	Technical Score	numeric	-	positive values
g_k	General Knowledge Score	numeric	-	positive values
Job_prof	Job Proficiency Score	numeric	-	positive values

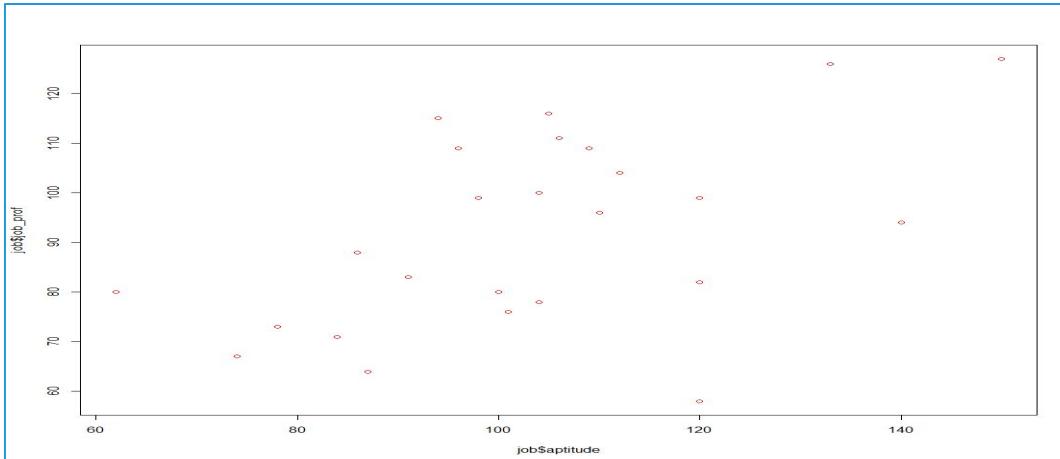
# Scatter Plot in R

```
# Importing Data  
job<-read.csv("Job_Proficiency.csv",header=T)
```

```
# Scatterplot  
plot(job$aptitude,job$job_prof,col="red")
```

```
# Output
```

- ❑ **plot()** gives a scatterplot of the two variables mentioned.
- ❑ **col=** provides color to the points.



# Pearson Correlation Coefficient in R

```
# Correlation  
cor(job$aptitude, job$job_prof) ←  
[1] 0.5144107
```

**cor()** calculates Pearson Correlation Coefficient for the two variables mentioned.

Pearson Correlation Coefficient	0.5144
---------------------------------	--------

There is positive relation between aptitude and job proficiency but the relation is of moderate degree.

# Simple Linear Regression in R

```
# Simple Linear Regression
```

```
model1<-lm(job_prof~aptitude, data=job)  
model1
```

```
# Output
```

```
Call:  
lm(formula = job_prof ~ aptitude, data = job)
```

```
Coefficients:  
(Intercept)      aptitude  
        41.3216          0.4922
```

lm() gives the linear regression model

# Inferences : Simple Linear Regression

Dependent Variable : Job Proficiency

Independent Variable : Aptitude

Intercept	Aptitude
41. 3216	0.4922

Equation : Job Proficiency = 41. 3216 + 0.4922 \* Aptitude

Here Job Proficiency changes by 0.4992 units with a unit change in aptitude.

# Case Study - 2

To learn more Descriptive Statistics in R, we shall consider the below case as an example.

## Background

Data of 100 retailers in platinum segment of the FMCG company.

## Objective

To describe bivariate relationships in the data

## Sample Size

Sample size: 100

Variables: Retailer, Zone, Retailer\_Age, Perindex, Growth,  
NPS\_Category

# Data Snapshot

Retail Data

**Variables**

Retailer	Zone	Retailer_Age	Perindex	Growth	NPS_Category
1	North	<=2	81.84	3.04	Promoter

Columns	Description	Type	Measurement	Possible values
Retailer	Retailer ID	numeric	-	-
Zone	Location of the retailer	character	East, West, North, South	4
Retailer_Age	Number of years doing business with the company	character	<=2, 2 to 5, >5	3
Perindex	Index of performance based on sales, buying frequency and buying recency	numeric	-	positive values
Growth	Annual sales growth	numeric	-	positive values
NPS_Category	Category indicating loyalty with the company	character	Detractor, Passive, Promoter	3

# Summarizing Two Categorical Variables

Using Frequency/Cross Tables describing the counts, percentages, etc. is a very basic and most useful way in summarizing two categorical variables.

```
#Importing Data
```

```
retail_data <- read.csv("Retail_Data.csv", header=TRUE)
```

```
# Frequency Tables
```

```
freq <- table(retail_data$Zone, retail_data$NPS_Category)  
freq
```

	Detractor	Passive	Promoter
East	5	9	1
North	5	13	7
South	7	9	16
West	6	10	12

**table()** in R, gives the frequency of counts of the two variables mentioned.

```
# Percentage Frequency Tables
```

```
prop.table(freq)
```

	Detractor	Passive	Promoter
East	0.05	0.09	0.01
North	0.05	0.13	0.07
South	0.07	0.09	0.16
West	0.06	0.10	0.12

**prop.table()** in R, gives the frequency expressed as percentage of total count.

# Summarizing Two Categorical Variables

```
prop.table(freq,1)
```

	Detractor	Passive	Promoter
East	0.33333333	0.60000000	0.06666667
North	0.20000000	0.52000000	0.28000000
South	0.21875000	0.28125000	0.50000000
West	0.21428571	0.35714286	0.42857143

- ❑ **prop.table()** in R, gives the frequency expressed as percentage of total count.
- ❑ (,1) expresses the frequency as percentage of row count whereas (,2) would express it as percentage of column count.

# Summarizing Two Categorical Variables

```
# Installing package - "gmodels"  
  
install.packages("gmodels")  
library(gmodels)  
  
# Frequency Table using "gmodels" package  
CrossTable(retail_data$Zone, retail_data$NPS_Category) ←
```

Cell Contents				
N				
Chi-square contribution				
N / Row Total				
N / Col Total				
N / Table Total				
Total Observations in Table: 100				
retail_data\$Zone		retail_data\$NPS_Category	Promoter	Row Total
Detractor		Passive		
East		5	9	15
0.696		1.321	3.585	
0.333		0.600	0.067	0.150
0.217		0.220	0.028	
0.050		0.090	0.010	
North		5	13	25
0.098		0.738	0.444	
0.200		0.520	0.280	0.250
0.217		0.317	0.194	
0.050		0.130	0.070	
South		7	9	32
0.018		1.294	1.742	
0.219		0.281	0.500	0.320
0.304		0.220	0.444	
0.070		0.090	0.160	
West		6	10	28
0.030		0.191	0.366	
0.214		0.357	0.429	0.280
0.261		0.244	0.333	
0.060		0.100	0.120	
Column Total		23	41	100
		0.230	0.410	0.360

**CrossTable()**  
in R, gives the frequency of counts of the two variables mentioned

# Summarizing Two Categorical Variables

```
# Frequency Table using 'gmodels' package
```

```
CrossTable(retail_data$Zone, retail_data$NPS_Category, prop.r = FALSE,  
prop.c = FALSE)
```

Cell Contents  
|-----  
| N  
| Chi-square contribution  
N / Table Total

Total Observations in Table: 100

retail_data\$Zone	retail_data\$NPS_Category			Row Total
	Detractor	Passive	Promoter	
East	5 0.696 0.050	9 1.321 0.090	1 3.585 0.010	15
North	5 0.098 0.050	13 0.738 0.130	7 0.444 0.070	25
South	7 0.018 0.070	9 1.294 0.090	16 1.742 0.160	32
West	6 0.030 0.060	10 0.191 0.100	12 0.366 0.120	28
Column Total	23	41	36	100

**prop.r=** removes the row proportion from the output and  
**prop.c=** removes the column proportion

# Summarizing Three Categorical Variables

```
# Three Way Frequency Table
```

```
table1 <-  
table(retail_data$Zone, retail_data$NPS_Category, retail_data$Retailer_Age)  
ftable(table1)←
```

		<=2	>5	2 to 5
East	Detractor	2	1	2
	Passive	3	3	3
	Promoter	0	1	0
North	Detractor	2	1	2
	Passive	1	6	6
	Promoter	1	6	0
South	Detractor	1	4	2
	Passive	2	3	4
	Promoter	3	10	3
West	Detractor	1	2	3
	Passive	1	8	1
	Promoter	0	11	1

**ftable()** in R, gives the frequency of counts of the three variables in one table itself.

# Quick Recap

In this session, we learnt the basics of Bivariate Relationships

## Bivariate Data

- Bivariate data can either have :
  - Two Numeric Variables
  - Two Categorical Variables
  - One Numeric and One Categorical Variable

## Scatter Plot

- Each dot on the scatterplot is one observation from a data set representing the corresponding variable value on X and Y axis respectively. Here X & Y are continuous variables.

## Pearson's Correlation Coefficient

- Numerically measures the strength of a linear relation between two variables

## Simple Linear Regression

- The equation of the line of best fit used to describe relationship between two variables

## Cross Tables

- Tables for summarizing categorical variables.

# v2 Basic Data Visualisation in R

# Contents

1. About Data Visualisation
2. Application Areas
3. Summarizing Data in Diagrams
  - i. Bar Diagram
    - Simple Bar Chart
    - Sub Divided/Stacked Bar Chart
    - Multiple Bar Chart
  - ii. Pie Chart
5. Summarizing Data in Diagrams using R

# About Data Visualisation

## What is Data Visualisation?

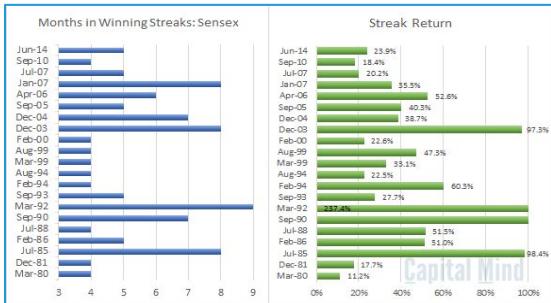
It is the visual representation of data in the form of graphs and plots.

## Why is it important?

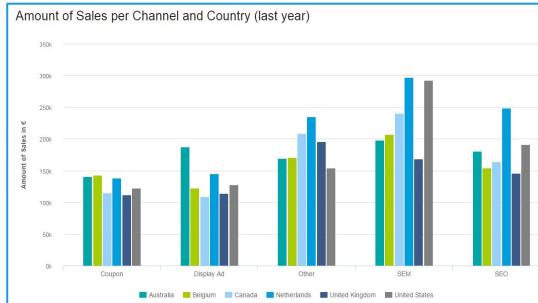
It enables us to

- See the data and get insights in one glance
- Allows us to grasp difficult / complex data in an easy manner
- Helps us to identify patterns or trends easily. Also shows distribution, correlation and causality in data.

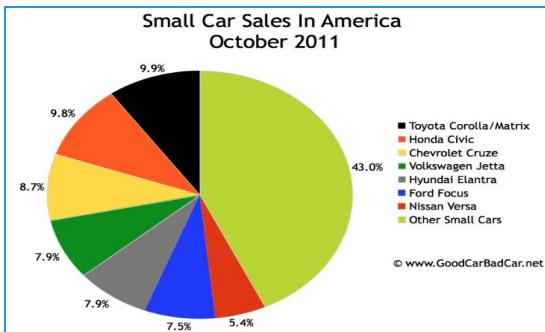
# Application Areas



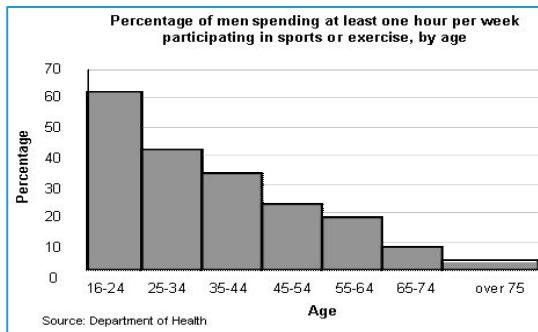
Sensex Charts



Sales Charts



Sales Charts



Survey Results

# Case Study

To get a better understanding of the subject, we shall consider the below case as an example.

## Background

A telecom service provider has the Demographic and Transactional information of their customers

## Objective

To visualize the data using usage variables and customer demographic information for generating business insights.

## Sample Size

1000

# Data Snapshot

telecom data

Variables

Observations

CustID	Age	Gender	PinCode	Active	Calls	Minutes	Amt	AvgTime	Age_Group	
1001	29	F	186904	Yes	2247	18214	3168.76	8.105919	18-30	
Columns										
<u>CustID</u>		Customer ID			Numeric		-		-	
<u>Age</u>		Age of the Customer			Numeric		-		-	
Gender	Gender of the Customer			Categorical		M, F		2		
<u>PinCode</u>		<u>Pincode</u> of area			Numeric		-		-	
Active	Age of the Customer			Categorical		Yes, No		2		
Calls	Number of Calls made			Numeric		-		positive values		
Minutes	Number of minutes spoken			Numeric		minutes		positive values		
Amt	Amount charged			Continuous		Rs.		positive values		
<u>AvgTime</u>		Mean Time per call			Continuous		minutes		positive values	
<u>Age_Group</u>		Age Group of the Customer			Categorical		18-30, 30-45, >45		3	

# Simple Bar Diagram

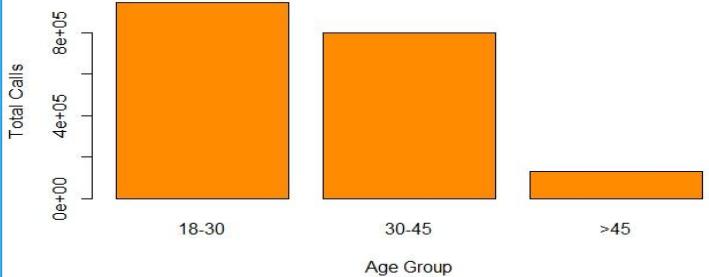
A **Bar Chart** is the simplest and the most basic form of graph. In this graph, for each data item, we simply draw a 'bar' showing its value.

**Simple Bar Chart:** It is a type of chart which shows the values of different categories of data as rectangular bars with different lengths. The values are generally :

- Frequency
- Mean
- Totals
- Percentages

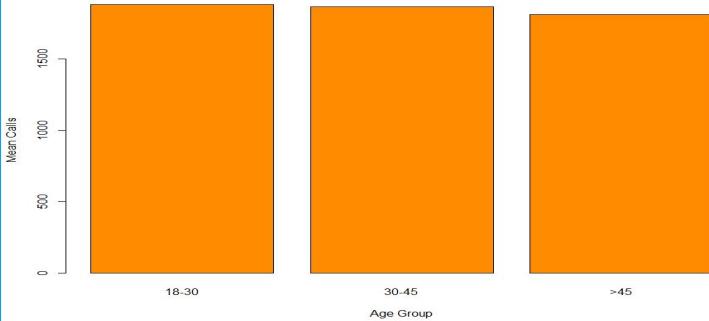
# Simple Bar Diagram

Fig.No.1 : SIMPLE BAR CHART (Total Calls - Age Group)



This graph simply gives the total number of calls for each age group.

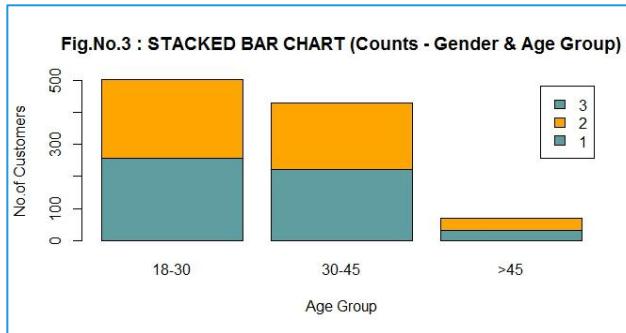
Fig.No.2 : SIMPLE BAR CHART (Mean Calls – Age Group)



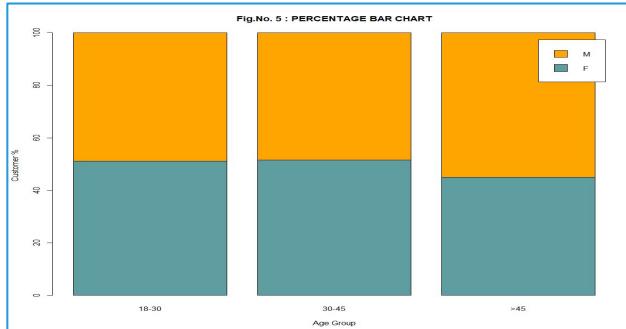
By plotting the average calls we can see that, though there is quite a difference in total calls in each age group, the average number of calls across age groups is similar.

# Stacked Bar Diagram

Sub Divided or Stacked Bar Chart: It further divides the bar into different categories within the variable.



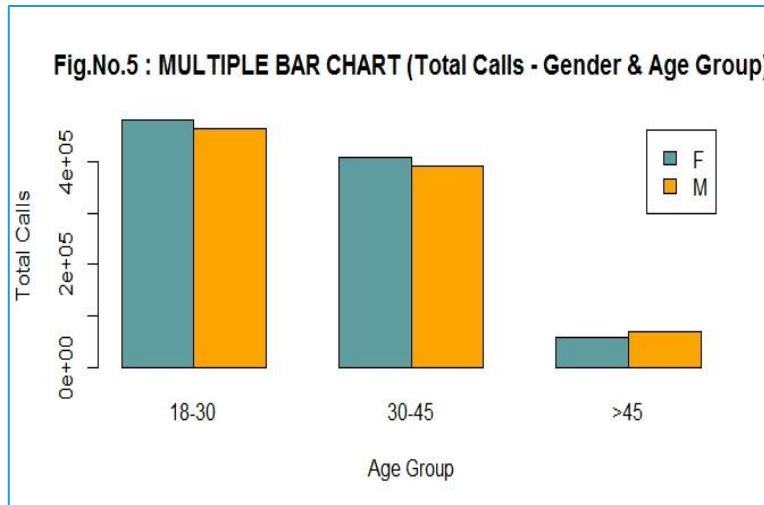
This graph divides the number of customers in each age group by Gender. However, this graph has absolute counts and is difficult to compare the gender wise distribution across the Age Groups.



Plotting a percentage stacked graph makes it efficient to compare the gender wise distribution of the number of customers across the Age Groups.

# Multiple Bar Diagram

**Multiple or Grouped Bar Chart:** It divides the bar into different categories within the variable and places it one besides the other. By multiple bars diagram two or more sets of inter-related data are represented.



This bar chart can be used when we wish to see the gender-wise distribution of number of calls across age groups.

# Diagrams in R

```
#Importing Data
```

```
telecom<-read.csv("telecom.csv", header=TRUE)
```

```
#Aggregating Data
```

```
telecom1<-aggregate(Calls~Age_Group,data = telecom, FUN=sum)
```

```
telecom1
```

	Age_Group	Calls
1	>45	128870
2	18-30	943187
3	30-45	798721

For plotting a bar chart in R,  
it is important to aggregate  
the data to get required  
vector/matrix

# Simple Bar Chart in R

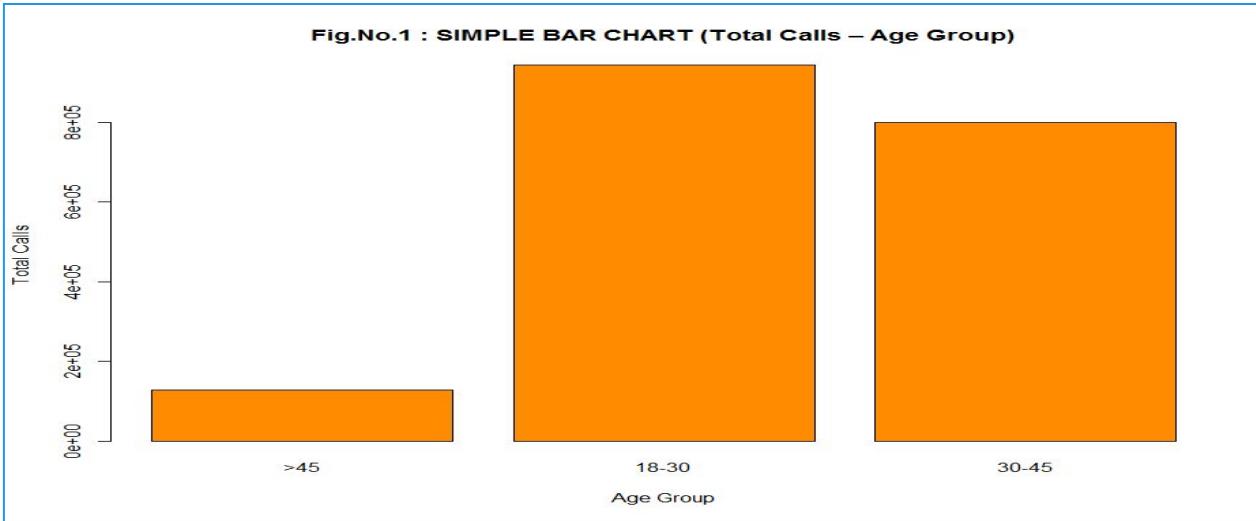
```
#Simple Bar Chart - Total Calls for different Age Groups
```

```
barplot(telecom1$Calls, main= "Fig.No.1 : SIMPLE BAR CHART (Total Calls -  
Age Group)", names.arg = telecom1$Age_Group,  
       xlab = "Age Group", ylab="Total Calls", col = "darkorange")
```

- barplot()** in base R yields different types of bar chart
- telecom1\$Calls** has to be a vector or matrix for which the bar chart needs to be plotted
- main=** provides the user defined name of the chart. It has to be put in double quotes
- names.arg=** specifies the names given to each bar
- xlab=** provides a user defined label for the variable on X axis
- ylab=** provides a user defined label for the variable on Y axis
- col=** can be used to input your choice of color to the bars

# Simple Bar Chart in R

This is the output that you get on running the previous code

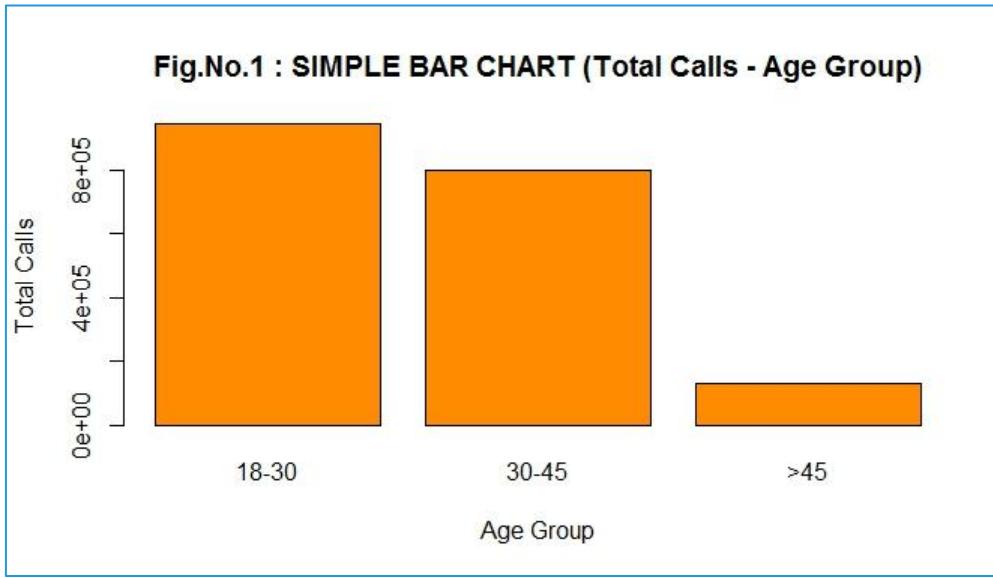


To get the bars in proper order, we will have to re-order the levels of column “Age\_Group” in telecom data as follows & then run the same R codes again :

```
telecom$Age_Group <- factor(telecom$Age_Group, levels = c("18-30", "30-45", ">45"))
```

# Simple Bar Chart in R

This graph simply gives the distribution of the **Total number of calls** across different **Age Groups**.



## Interpretation :

- Number of calls made by young age group (18-30) is slightly higher than mid age group (30-45) and very high than age group >45.

# Simple Bar Chart in R

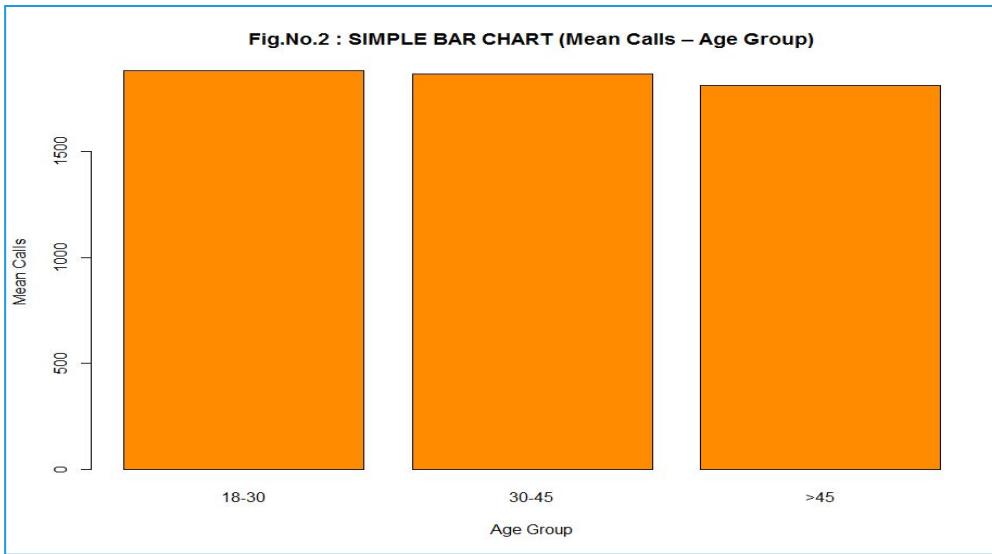
```
# Simple Bar Chart - Mean Calls for different Age Groups  
  
telecom2<-aggregate(Calls~Age_Group,data = telecom, FUN=mean)  
Telecom2  
  
Age_Group      Calls  
1     18-30 1882.609  
2     30-45 1866.171  
3     >45 1815.070  
  
barplot(telecom2$Calls, main= "Fig.No.2 : SIMPLE BAR CHART (Mean Calls -  
Age Group)", names.arg = telecom2$Age_Group, xlab = "Age Group",  
ylab="Mean Calls", col = "darkorange")
```

## Note :

- The barplot code remains the same with respect to previous barplot code, the only difference is while aggregating the data.
- In previous plot aggregation function was “**sum**” & in this plot aggregation function is “**mean**”.

# Simple Bar Chart in R

This graph simply gives the distribution of the **Mean calls** across different **Age Groups**.



## Interpretation :

- By plotting the average calls we can see that, though there is quite a difference in total calls in each age group, **the average number of calls across age groups is similar.**

# Simple Bar Chart in R

```
# Simple Bar Chart in Horizontal orientation
```

```
tele1<-table(telecom$Age_Group)  
tele1
```

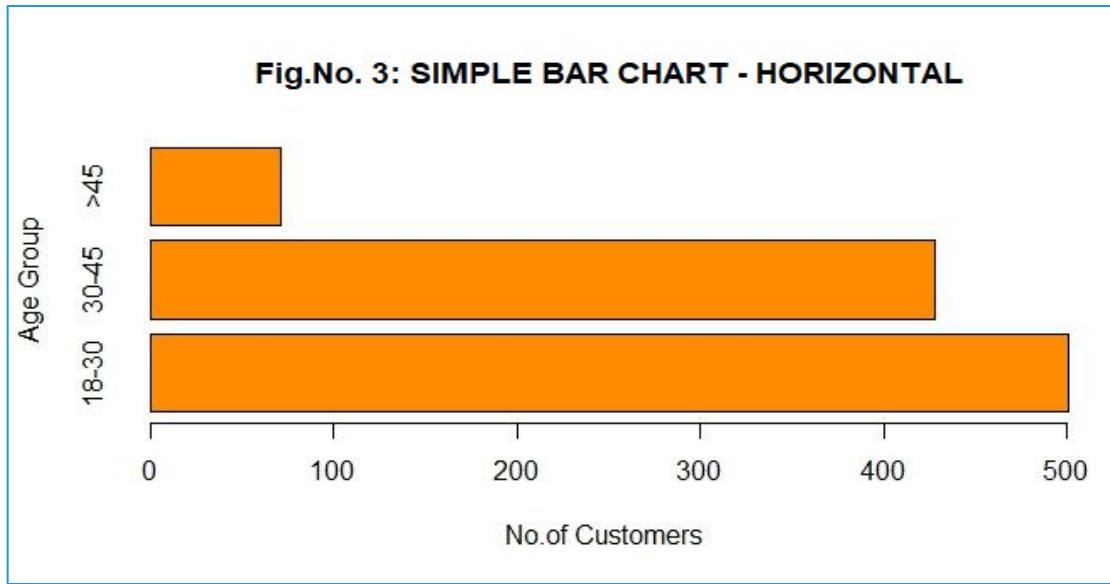
18-30	30-45	>45
501	428	71

```
barplot(tele1, main= "Fig.No. 3: SIMPLE BAR CHART - HORIZONTAL",  
xlab="No.of Customers",ylab = "Age Group",col = "darkorange",horiz = TRUE)
```

- ❑ **horiz =** gives horizontal orientation to the bars.  
It takes the frequency on the X axis

# Simple Bar Chart in R

This graph displays the number of customers across age group.



## Interpretation :

- This is horizontal view, which indicates that there are very few customers for age group >45 as compared to other two age groups.
- This graph is generally useful when there are negative frequency values in the data.

# Stacked Bar Chart in R

```
# Stacked Bar Chart
```

```
telecom3<-table(telecom$Gender,telecom$Age_Group)
```

```
telecom3
```

	18-30	30-45	>45
F	256	221	32
M	245	207	39

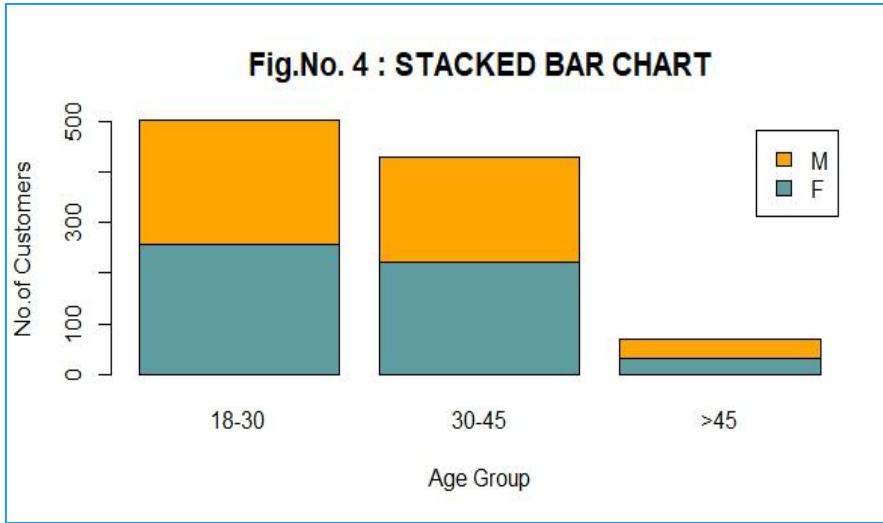
- ❑ **table()** inputting two variables gives a matrix having their counts in each category

```
barplot(telecom3,main="Fig.No. 4 : STACKED BAR CHART",  
xlab ="Age Group",ylab ="No.of Customers", col=c("cadetblue","orange"),  
legend=rownames(telecom3))
```

- ❑ **legend=rownames()** displays the legend on the graph output

# Stacked Bar Chart in R

This graph divides the number of customers in each age group by Gender.



## Interpretation :

- This graph shows that, though there are more young customers in data but, almost equal number of Males and Females are present in each age group.

# Percentage Bar Chart in R

```
# Percentage Bar Chart
```

```
telecom4<-prop.table(telecom3,2)
telecom4
```

	18-30	30-45	>45
F	0.5109789	0.5163551	0.45070
M	0.4890220	0.4836449	0.54929

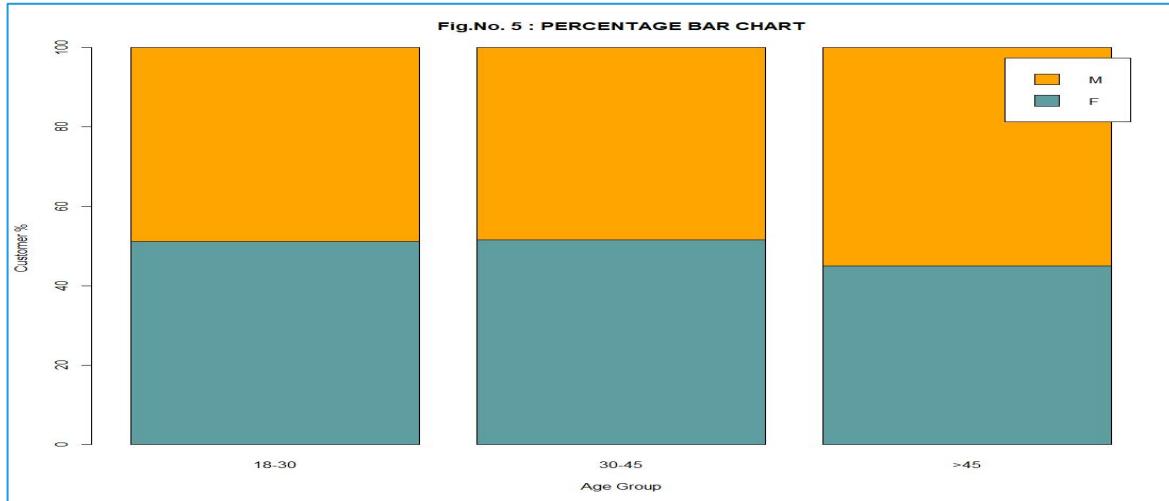
- **prop.table()** helps us create data frame with percentage values
- **(,2)** gives percentage as per column count

```
barplot(telecom4*100, main= "Fig.No. 5 : PERCENTAGE BAR CHART",
xlab = "Age Group", ylab="Customer %", col = c("cadetblue","orange"),
legend= rownames(telecom4))
```

- **telecom4\*100** has to be a vector or matrix for which the bar chart needs to be plotted. \*100 would display percentage scale on y-axis.

# Percentage Bar Chart in R

```
# Output for gender wise distribution of number of customers across the  
Age Groups.
```



## Interpretation :

- Data contains almost equal proportion of Male and Female callers across three different age groups.
- Plotting a percentage stacked graph makes it efficient to compare the gender wise distribution of the number of customers across the Age Groups.

# Multiple Bar Chart in R

```
# Multiple Bar Chart
```

```
telecom5<-aggregate(Calls~Age_Group+Gender, data= telecom, FUN = sum)  
telecom5
```

```
telecom6<-xtabs(Calls~Gender+Age_Group,telecom5)
```

		18-30	30-45	45
		F	408184	583
		M	462952	390537

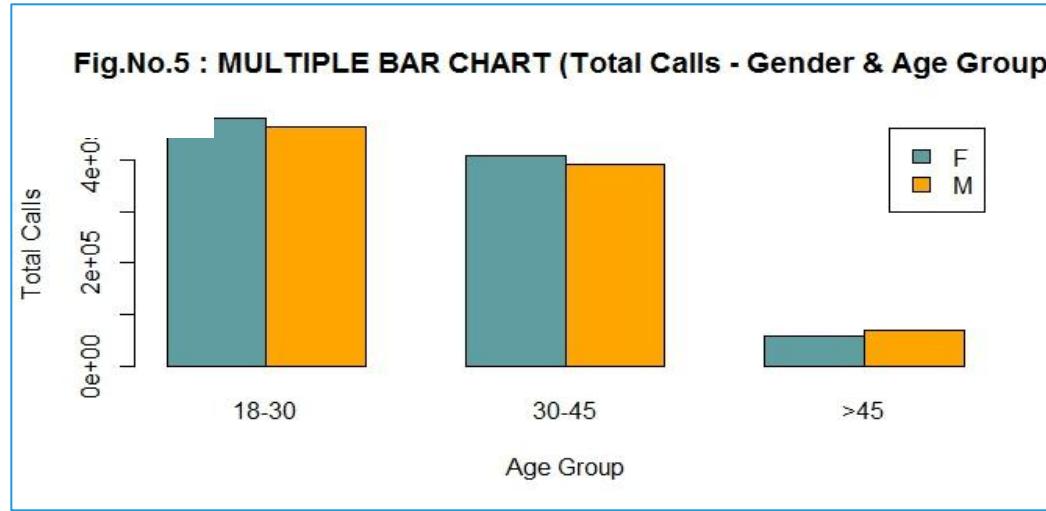
- ❑ **xtabs()** is used to cross tabulate the categories of more than one variables using another numeric variable which results in total of each category

```
barplot(telecom6,main="Fig.No.6 : MULTIPLE BAR CHART (Total Calls - Gender  
& Age Group)", xlab = "Age Group", ylab="Total Calls",  
col=c("cadetblue","orange"), legend=rownames(telecom6), beside = TRUE) ←
```

- ❑ **beside=TRUE** enables us to show the different class of the same bar one beside the other

# Multiple Bar Chart in R

```
# Output for gender-wise distribution of number of calls across age groups
```



## Interpretation :

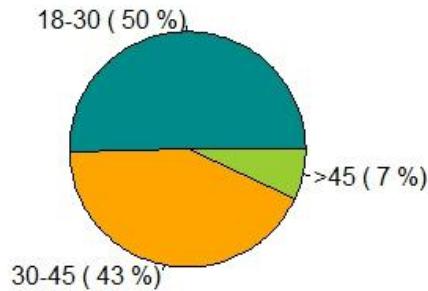
- There is no significant difference between Male and Female in terms of number of calls made across three different age groups, the only difference is that, age group >45 has slightly more male customers than female customers as compared to other age groups.
- This can be used as an alternative way of representing a stacked bar graph.

# Pie Chart

Pie charts are generally used to show percentage or proportional data.

In this graph the entire circle (pie) is sliced proportional to the values of each category.

**Fig.No. 6 : Pie Chart with Percentage**



The above Pie Chart show how the total number of Calls are proportionally distributed amongst age groups.

# Pie Chart in R

```
# Pie Chart
```

```
telecom7<-aggregate(Calls~Age_Group,data = telecom, FUN=sum)
telecom7$pct <- round(telecom7$Calls/sum(telecom7$Calls)*100)
```

	Age_Group	Calls	pct
1	18-30	943187	50
2	30-45	798721	43
3	>45	128870	7

Here, we calculate the proportions for each category using formula

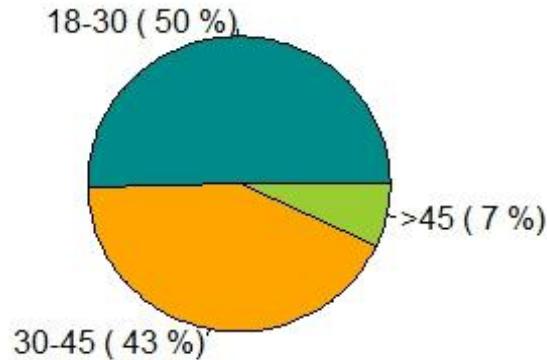
```
pie(telecom7$Calls,labels = paste(telecom7$Age_Group,"(",telecom7$pct,"%"),
col=c("darkcyan","orange","yellowgreen"),main="Fig.No. 7 : PIE CHART WITH
PERCENTAGE")
```

- pie()** in base R yields a pie chart
- telecom7\$Calls** has to be a vector or matrix for which the pie chart needs to be plotted
- labels=** provides a user defined label for the variable on X axis
- paste()** labels each category using string values separated by commas
- col=** can be used to input your choice of color to the bars

## Pie Chart in R

# Output of Pie chart with percentage

**Fig.No. 7 : PIE CHART WITH PERCENTAGE**



### Interpretation :

- **50%** of calls are made by Age\_Group 18-30, **43%** by 30-45 & **only 7%** by >45 Age\_Group.

# Quick Recap

In this session, we learnt data visualisation using basic graphs

Chart Types and  
Functions in R

- Bar Diagrams - `barplot()`
- Pie Chart - `pie()`

## v2 Basics of Data Visualisation in R

### Other Basic Graphs with R

# Contents

## 1. Summarizing Data in Diagrams

1. Box-Whisker Plot
2. Histogram
3. Density Plot
4. Stem and Leaf Diagram
5. Pareto Chart

## 2. Summarizing Data in Diagrams using R

# Box – Whisker Plot

Box and Whisker plot summarizes data graphically using 5 measures:

- Minimum
- The Three Quartiles : Q1, Q2 (i.e. Median) and Q3
- Maximum

Describing a Box-Plot :

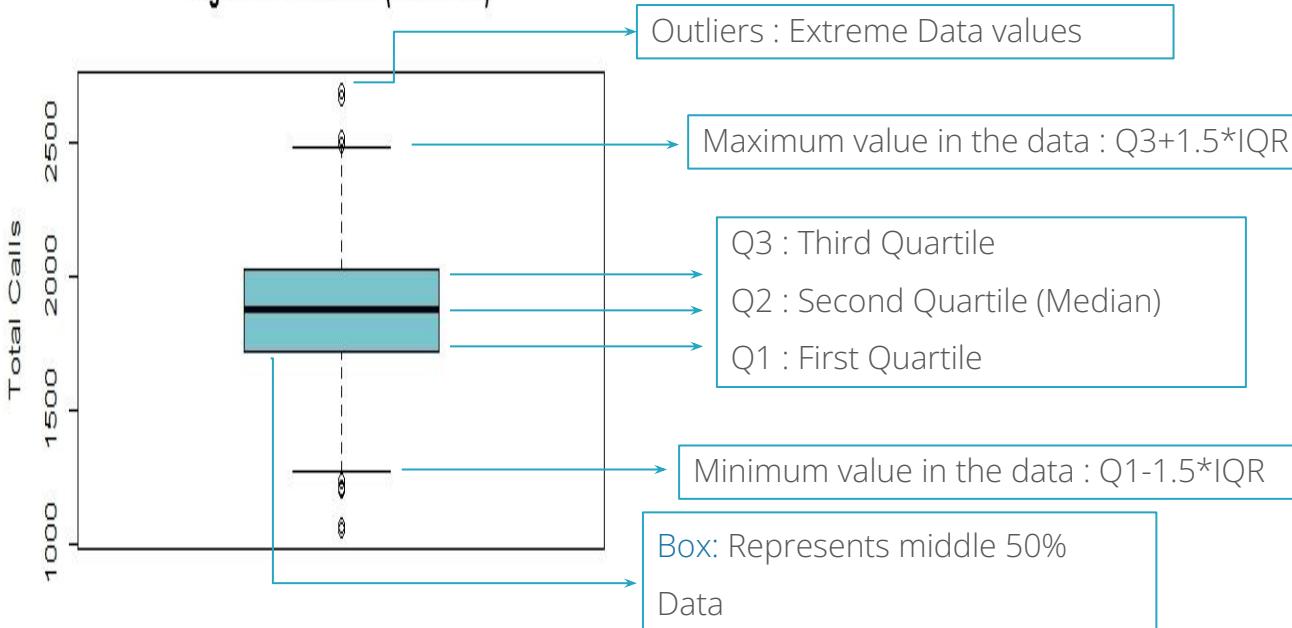
- The rectangle (box) in the middle represents the middle 50% of the data (between the values that are  $\frac{1}{4}$  and  $\frac{3}{4}$  of the way through the data).
- The lines (whiskers) extend from the box to the smallest and largest values.
- The diagram also shows the middle value (i.e. The Median).
- The outliers which are plotted outside the plot (The observations which are outside 1.5 times the interquartile range above the upper quartile and below the lower quartile)

Advantages of a Box Plot :

- A boxplot is particularly effective when comparing two sets of data
- It shows us the shape of the data.

# Box – Whisker Plot

Fig.No.8 : BOX PLOT (Total Calls)



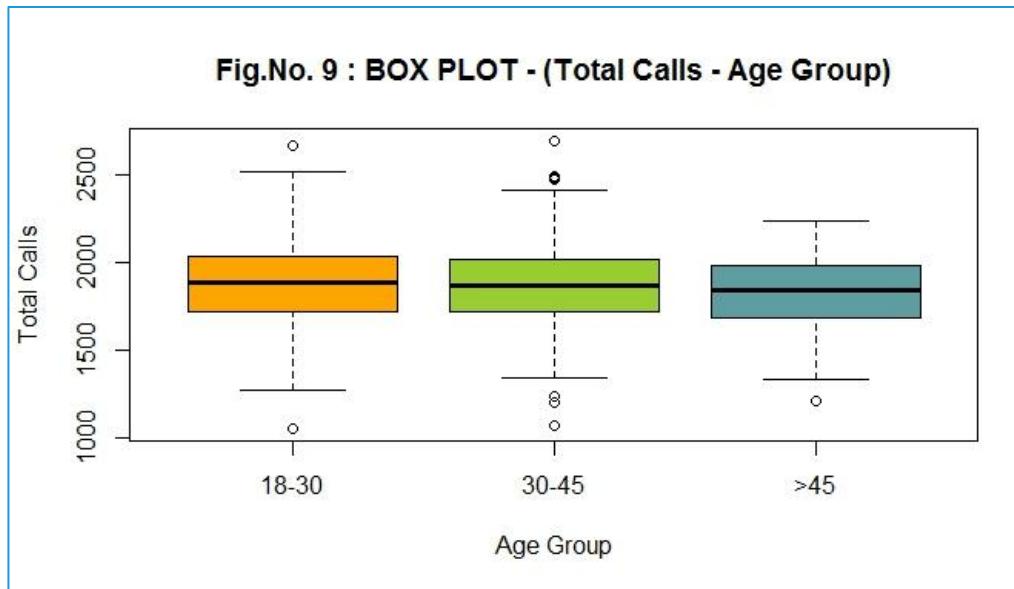
This plot shows that the distribution of total call is very much symmetric & there exists few outliers in the data.



The minimum and maximum values are the ones excluding the outliers

## Box – Whisker Plot

Here, plotting box plots for each categories of a variable gives us a good comparison of how 'total calls' is distributed for various age groups.



- We can observe that the distribution is almost symmetric amongst various age groups, but variability is least in >45 age group.
- Also, the age group 30-45 has many outliers.

# Case Study

To get a better understanding of the subject, we shall consider the below case as an example.

## Background

A telecom service provider has the Demographic and Transactional information of their customers

## Objective

To visualize the data using usage variables and customer demographic information for generating business insights.

## Sample Size

1000

\*

Here we continue to use previous data for our further analysis.

# Data Snapshot

telecom data

Variables

CustID	Age	Gender	PinCode	Active	Calls	Minutes	Amt	AvgTime	Age_Group
1001	29	F	186904	Yes	2247	18214	3168.76	8.105919	18-30
Observations	Columns	Description			Type	Measurement		Possible values	
	CustID	Customer ID		Numeric	-	-			
	Age	Age of the Customer		Numeric	-	-			
	Gender	Gender of the Customer		Categorical	M, F	2			
	PinCode	Pincode of area		Numeric	-	-			
	Active	Age of the Customer		Categorical	Yes, No	2			
	Calls	Number of Calls made		Numeric	-	positive values			
	Minutes	Number of minutes spoken		Numeric	minutes	positive values			
	Amt	Amount charged		Continuous	Rs.	positive values			
	AvgTime	Mean Time per call		Continuous	minutes	positive values			
	Age_Group	Age Group of the Customer		Categorical	18-30, 30-45, >45	3			

# Box Plot in R

```
#Importing Data
```

```
telecom<-read.csv("telecom.csv", header=TRUE)
```

```
#BoxPlot - Total Calls
```

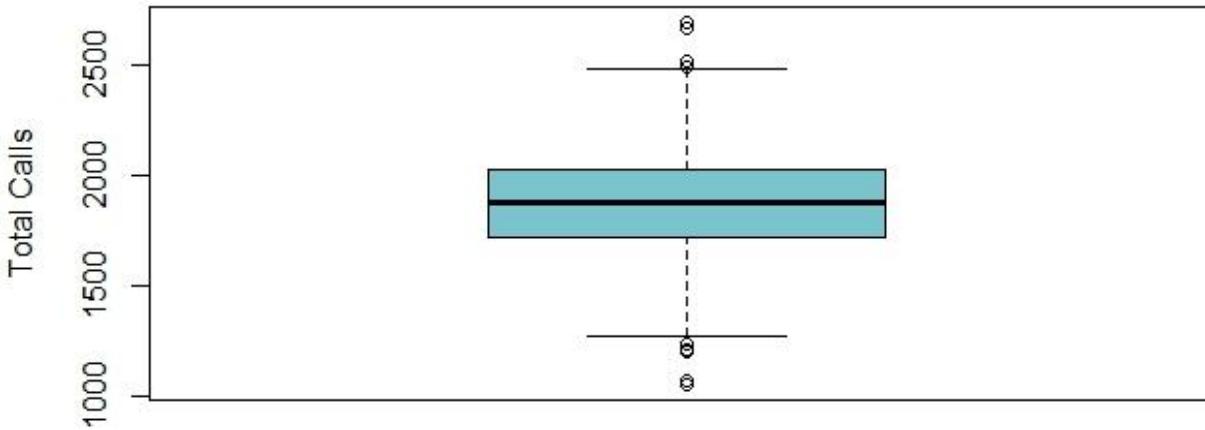
```
boxplot(telecom$Calls, data= telecom, main="Fig.No.8 : BOX PLOT (Total  
Calls)", ylab= "Total Calls", col= "cadetblue3")
```

- boxplot()** in base R yields a different types of box chart
- telecom\$Calls** specifies vector (variable) for which the box plot needs to be plotted
- data=** calls the data out of which the variable needs to be plotted
- main=** provides the user defined name of the chart. It has to be put in double quotes
- ylab=** provides a user defined label for the variable on Y axis
- col=** can be used to input your choice of color to the bodies of the box plots.

# Box Plot in R

```
# Output
```

**Fig.No. 8 : BOX PLOT (Total Calls)**



## Interpretation :

- While we see a few outliers , the data of the number of calls overall is symmetric

# Box Plot in R

```
#BoxPlot for different categories of Age_Group
```

```
boxplot(Calls~Age_Group,data=telecom,  
        main="Fig.No. 9 : BOX PLOT - (Total Calls - Age Group)",  
        xlab="Age Group",ylab="Total Calls",  
        col=c("orange","green","cadetblue"))
```

Difference between previous boxplot & this boxplot code is,

- ❑ **Calls~Age\_Group** specifies vector (variable) for which the box plot for different categories of a variable is to be plotted.
- ❑ **xlab=** provides a user defined label for the variable on X axis .
- ❑ **col=** can be used to input your choice of color to the bodies of the box plots. Here we have mentioned 3 colors as the variable has 3 categories.

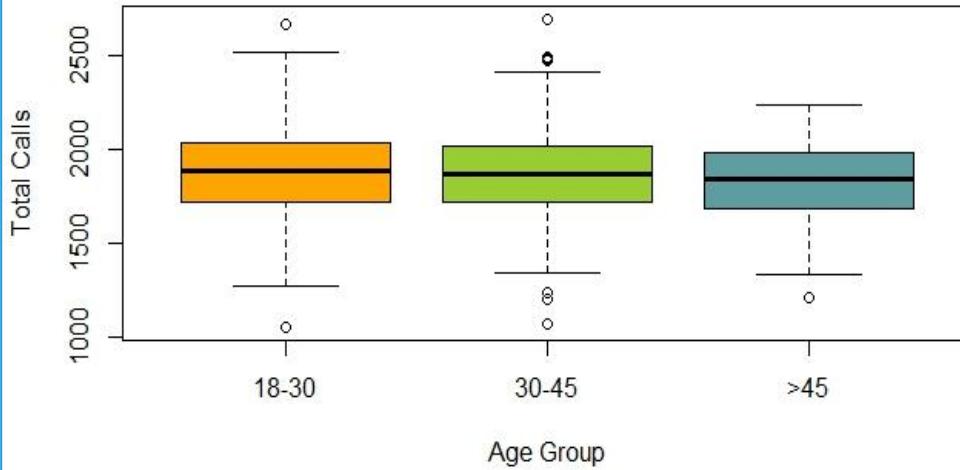


Note : Re – order the levels of Variable Age\_Group as explained in previous ppt before you execute the boxplot code, Age\_Group wise.

# Box Plot in R

```
# Output
```

**Fig.No. 9 : BOX PLOT - (Total Calls - Age Group)**

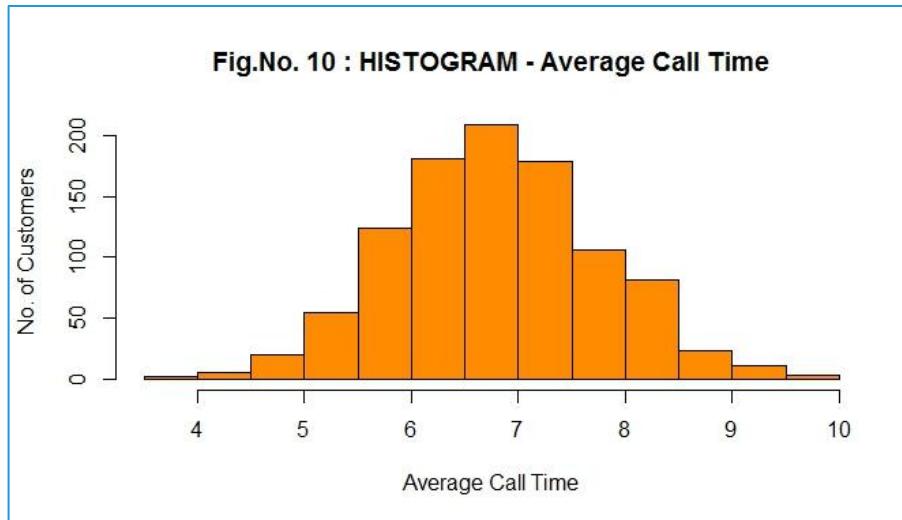


## Interpretation :

- Here we can observe that the spread of total calls is higher in the age group 18-30.
- The number of outliers is higher in 30 – 45 age group.
- However, there is symmetry between all age groups.

# Histogram

- A Histogram shows frequency for each bin or bucket created based on range of values of a variable. The histogram is recommended for a continuous variable and is generally used to check the Normality of the data.



- This plot shows that the distribution of Average Call Time is very much symmetric.

# Histogram in R

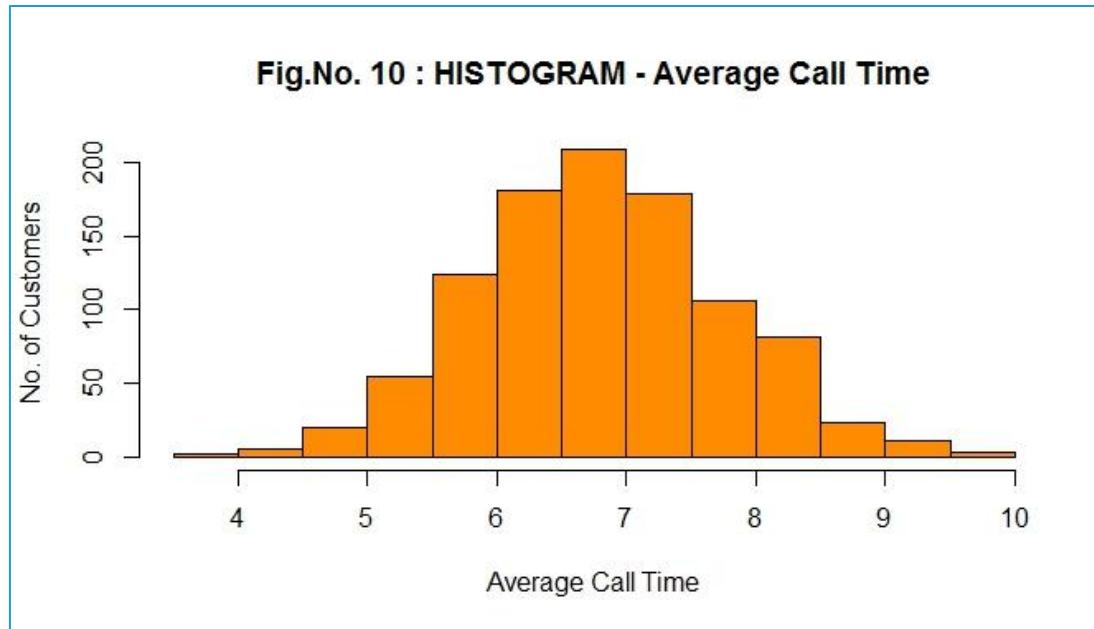
```
# Histogram - Average Call Time
```

```
hist(telecom$AvgTime, breaks=12, main = "Fig.No. 10 : HISTOGRAM – Average  
Call Time", xlab = "Average Call Time", ylab = "No. of Customers",  
col="darkorange")
```

- hist()** in base R yields a histogram
- telecom\$AvgTime** specifies vector (variable) for which the histogram needs to be plotted
- breaks=** specifies the number of bins in the histogram
- main=** provides the user defined name of the chart. It is to be put in double quotes
- xlab=** provides a user defined label for the variable on X axis
- ylab=** provides a user defined label for the variable on Y axis
- col=** can be used to input your choice of color to the bars

# Histogram in R

This plot shows the distribution of Average Call Time

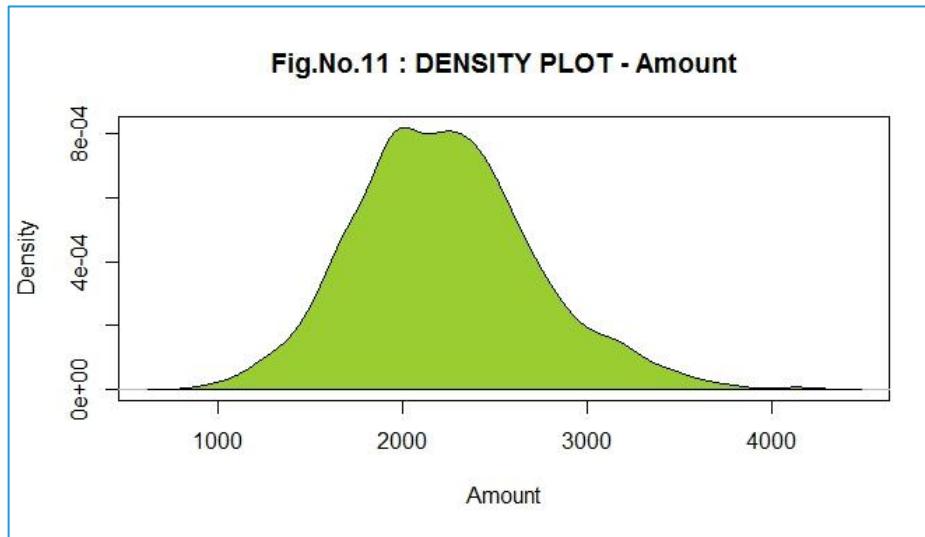


## Interpretation :

- This plot shows that the distribution of Average Call Time is quite symmetric.

# Density Plot

- A Density Plot is similar to a histogram which plots the probability.
- It is generally used to check the Normality of the data when there are higher data points.



- This plot shows that the distribution of amount is slightly positively skewed.

# Density Plot in R

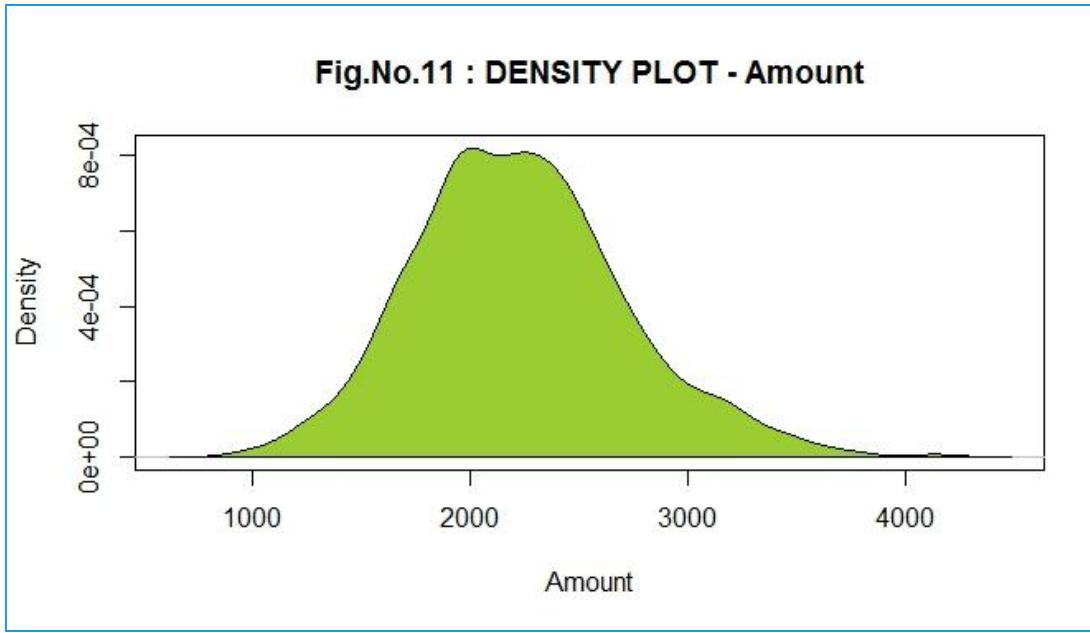
```
# Density Plot - Amount
```

```
Telecom_den<-density(telecom$Amt)
plot(Telecom_den, main="Fig.No.11 : DENSITY PLOT - Amount",xlab="Amount")
polygon(Telecom_den, col="yellowgreen")
```

- density()** returns the density values of the variable
- plot()** plots the line graph taking object returned by function density
- main=** provides the user defined name of the chart. It is to be put in double quotes
- xlab=** provides a user defined label for the variable on X axis
- polygon()** shows the area covered under the curve
- col=** can be used to input your choice of color to the polygon

# Density in R

This plot shows the distribution of Amount



## Interpretation :

- This plot shows that the distribution of Amount is slightly positively skewed.

# Stem and Leaf Plot

- A Stem and Leaf diagram can, again, be an alternative to a histogram.
- It is a special table where each numeric value is split into a stem (First digit(s)) and a leaf (last Digit)
- Stem and leaf diagrams show the shape of the distribution (like bar charts) but have the advantage of not losing the detail of the original data.
- Arranging the leaves in numerical order, will allow us to use the diagram to find the middle value (the median) and the values that are a quarter and three-quarters of the way through the data (the lower and upper quartiles).

The decimal point is 2 digit(s) to the right of the	
10	56
11	
12	014799
13	2223444556778
14	00011112233334445666666666678888888888889999999
15	001111123333444455666777777888888888899999
16	0000000000111112222222222222333333333444444445555555555555666666+34
17	00000000001111112222222222222233333333334444444444555+53
18	0000000000000111111111111111111122222222333333333333333333333+93
19	000000000000000000000000000011111111112222222233333333333333333333+89
20	000000000000000000000011111111111222222222222223333333344444+54
21	00000000111111111111223333344444555555566666666666666677888888888
22	0000111111223333334444445555555555556778899
23	000011222344445555556777799
24	112237789999
25	2
26	7
27	0

## Stem and Leaf Plot in R

```
# Stem and Leaf Plot in R  
stem(telecom$Calls)
```

- ❑ **stem()** in base R yields a stem and leaf chart
- ❑ **telecom\$Calls** specifies vector (variable) for which the stem and leaf plot needs to be plotted

# Stem and Leaf Plot in R

# Output

```
The decimal point is 2 digit(s) to the right of the |

10 | 56
11 |
12 | 014799
13 | 2223444556778
14 | 00011112233334445666666666678888888888889999999
15 | 00111112333344445566667777778888888888999999
16 | 00000000000111112222222222222333333333444444445555+34
17 | 0000000000011111122222222222222333333333344444444555+53
18 | 00000000000001111111111111111111222222222333333333333333+93
19 | 00000000000000011111111111111111111222222222333333333333333+89
20 | 0000000000000000000001111111111112222222222222333333334444+54
21 | 000000001111111111122333344444555555556666666666677888888888
22 | 0000111111122333333444444455555555556778899
23 | 000011222344445555556777799
24 | 112237789999
25 | 2
26 | 7
27 | 0
```

## Interpretation :

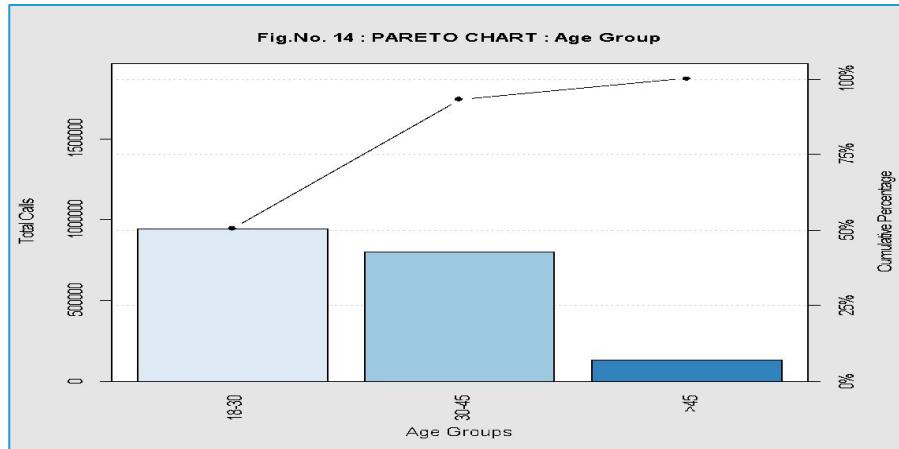
- The stem and leaf plot of overall calling data shows that, calls values are symmetrically distributed and there exists few outliers also in the data.



The output of the stem and leaf diagram is produced in the R console

# Pareto Chart

- Pareto chart, named after Vilfredo Pareto, is a type of chart that contains both a bar and a line graph, where individual values are represented in descending order by bars. In this way the chart visually depicts which categories are more significant. The cumulative total is represented by the line.
- There needs to be at least one categorical variable to plot this chart.



- From the above chart we can interpret that 50% of the Total calls made come from age group 18-30.
- Another 42% calls are made by age group 30-45, only 8% calls are made by customers > 45 .

# Get an Edge!

## RColorBrewer

RColorBrewer is a package that uses [www.colorbrewer2.org](http://www.colorbrewer2.org) to help choose colour schemes for graphics in R

The colours are split into 3 groups :

1. Sequential : Light colours for low data, dark for high data
2. Diverging : Light colours for mid-range data, low and high contrasting dark colours
3. Qualitative: Colours designed to give maximum visual difference between classes

```
install.packages("RColorBrewer")
library(RColorBrewer)←
col=brewer.pal(n,"palette")
```

We need to install the  
“RColorBrewer” package to use  
the color brewer in R

- ❑ **brewer.pal()** is the function to be used in “**col=**” argument.
- ❑ **n** specifies the number of colors to be used
- ❑ **palette** is the name of the color palette which can be chosen by running **display.brewer.all()** function



Refer to [www.stat.auckland.ac.nz](http://www.stat.auckland.ac.nz) for details

# Pareto Chart in R

```
# Pareto Chart - Age Group
```

```
install.packages("qcc")
library(qcc)
library(RColorBrewer)
```

Using "qcc" package is the easiest way to plot  
a Pareto Chart in R

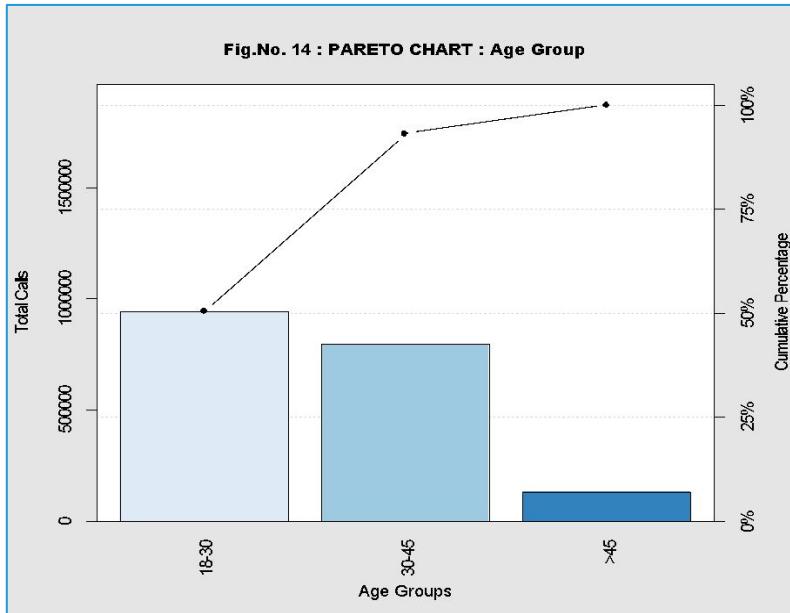
```
telecom1<-aggregate(Calls~Age_Group,data = telecom, FUN=sum)
```

```
pareto.chart(telecom1$Calls, xlab= "Age Groups" ,ylab= "Total Calls" ,
main = "Fig.No. 14 : PARETO CHART : Age Group",
col=brewer.pal(3,"Blues"), names.arg=telecom1$Age_Group)
```

- ❑ **pareto.chart()** is the function in “qcc” package used to plot a Pareto Chart
- ❑ **telecom1\$Calls** specifies vector (variable) for which the Pareto chart needs to be plotted
- ❑ **names.arg=** is the argument that allows the bars to be named according the row names in the variable mentioned
- ❑ **main=** provides the user defined name of the chart. It has to be put in double quotes
- ❑ **xlab=** provides a user defined label for the variable on X axis
- ❑ **col=** can be used to input your choice of color to the bars
- ❑ **brewer.pal** uses the RColorBrewer to colour the bars

# Pareto Chart in R

# Output



## Interpretation :

- 50% of the Total calls made come from age group 18-30.
- Another 42% calls are made by age group 30-45, only 8% calls are made by customers > 45 years of age

# Quick Recap

In this session, we learnt data visualisation using basic graphs

## Chart Types and Functions in R

- .. Box-Whisker Plot - `boxplot()`
- .. Histogram - `hist()`
- .. Density Plot - `plot() + polygon()`
- .. Stem and Leaf Diagram - `stem()`
- .. Pareto Chart - `pareto.chart()` in package "qcc"

v2 Data Visualization

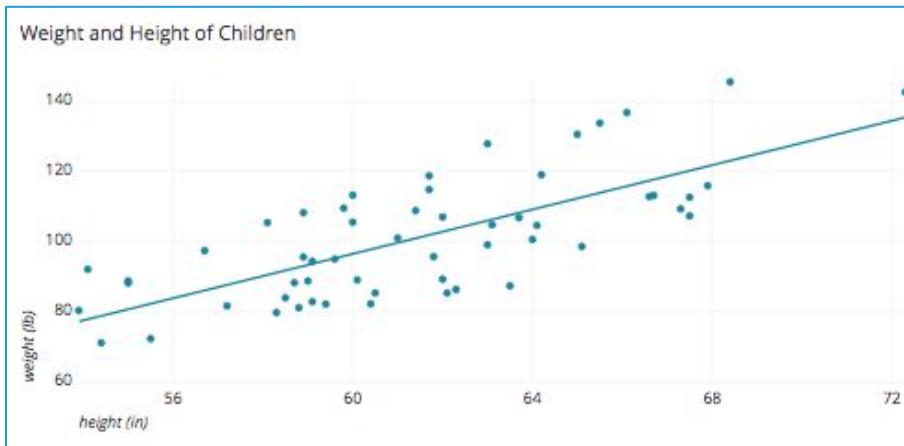
Visualizing Relationships in R

# Contents

- 1.** Summarizing Data in Diagrams
  - i.** Scatterplot with Regression Line
  - ii.** Scatterplot Matrix
  - iii.** Bubble Chart
  - iv.** Heat Map
  - v.** Trend Line
  - vi.** Motion Chart
- 2.** Application Areas

# Scatter Plot

A **scatter plot** is a two-dimensional data visualization that uses dots to represent the values obtained for two different variables - one plotted along the x-axis and the other plotted along the y-axis. For example this scatter plot shows the height and weight for a set of children.



Each dot represents one child with his or her height measured along the x-axis and weight measured along the y-axis

Scatter plots are used when you want to see how two variables are correlated. In the height and weight e.g., the chart wasn't just a simple log of the height and weight of a set of children, but it also visualized the relationship between height and weight - namely that weight increases as height increases. Notice that the relationship isn't perfect, some taller children weight less than some shorter children, but the general trend is pretty strong and we can see that weight is correlated with height.

# Case Study

Let us try and see the correlation between Aptitude score of an employee and how is job performance/Proficiency

## Background

A company has the scores of various attribute tests of their employees

## Objective

To understand the factors contributing to the Job Proficiency of an employee.

To see the relationship between these various factors

## Sample Size

25

# Data Snapshot

JOB PROFICIENCY  
DATA Variables

Observations

empno	aptitude	testofen	tech_	g_k_	job_prof
1	86	110	100	87	88
2	62	62	99	100	80
3	110	107	103	103	96
--	--	--	--	--	--

Columns	Description	Type	Measurement	Possible values
empno	Employee No	Numeric	-	-
aptitude	Aptitude	Numeric	-	positive values
testofen	Test of English	Numeric	-	positive values
tech_	Technical Score	Numeric	-	positive values
g_k_	General Knowledge	Numeric	-	positive values
job_prof	Job Proficiency	Numeric	-	positive values

# ScatterPlot with Regression Line in R

```
# Importing Data
```

```
job<-read.csv("JOB PROFICIENCY DATA.csv", header=TRUE)  
attach(job)
```

`attach()` is used to call the data in R with help of which in further codes specifying the data repetitively can be avoided .

```
#Scatterplot with Regression Line
```

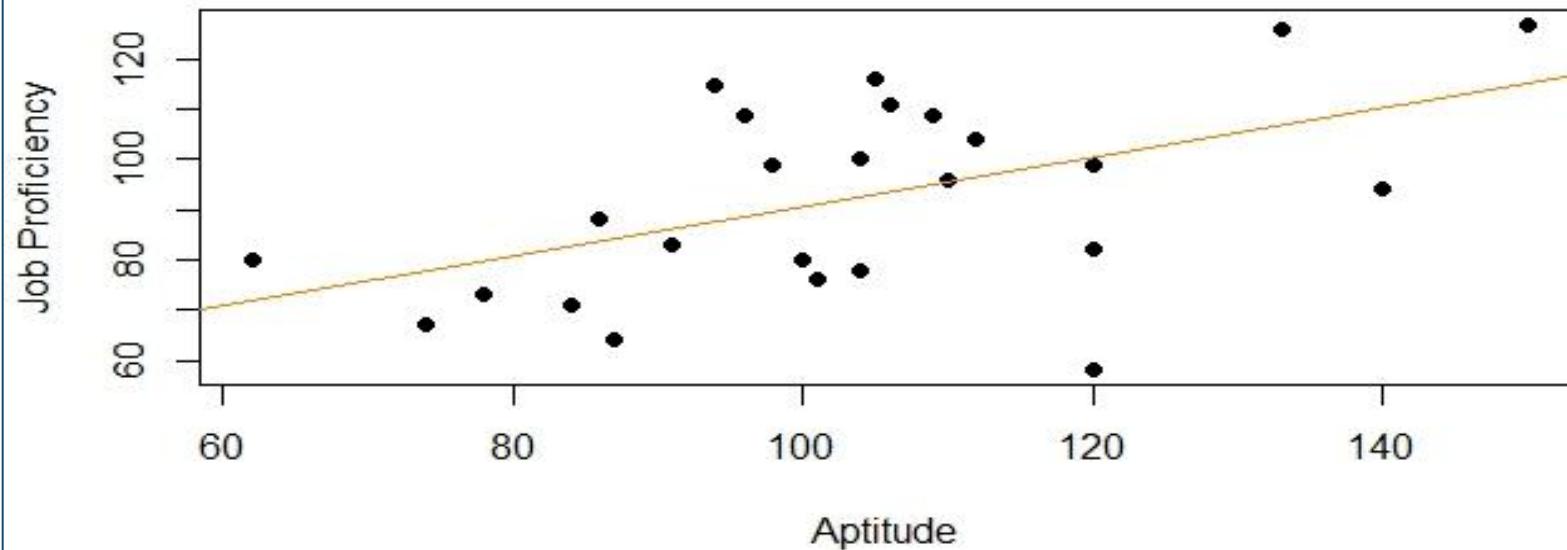
```
plot(aptitude,job_prof, main="Fig.No. 1 : ScatterPlot with Regression  
Line", xlab="Aptitude ", ylab="Job Proficiency", pch=19)  
abline(lm(job_prof~aptitude), col="darkorange")
```

- ❑ `plot()` in base R yields different types of plots
- ❑ `aptitude` is one of the variable for plot
- ❑ `job_prof` is another variable to be plotted
- ❑ `main=` provides the user defined name of the chart. It has to be put in double quotes
- ❑ `xlab=` provides a user defined label for the variable on X axis
- ❑ `ylab=` provides a user defined label for the variable on Y axis
- ❑ `pch=` gives various shapes for the data points on the plot

- ❑ `abline()` in base R yields different types of lines on plot
- ❑ `lm()` provides the liner regression line of the first variable mentioned on the second
- ❑ `col=` provides the color of the line plotted

# ScatterPlot with Regression Line in R

**Fig.No. 1 : ScatterPlot with Regression Line**

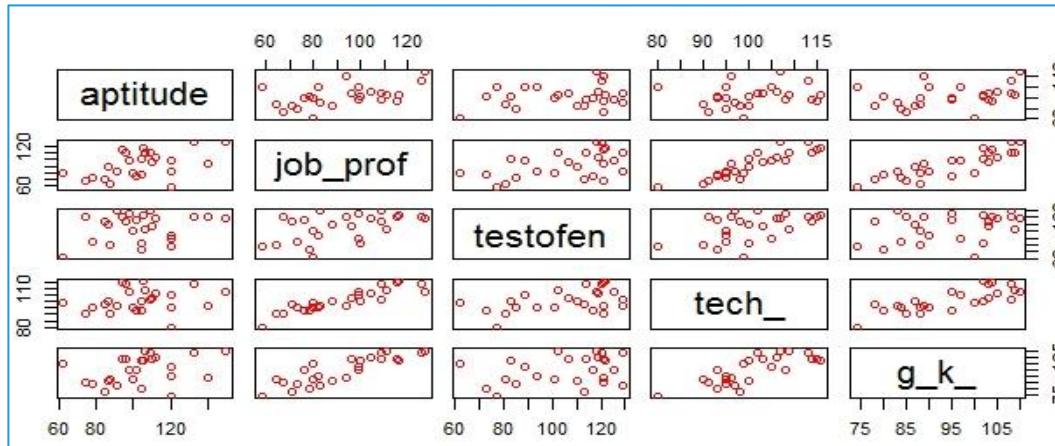


## Interpretation :

- Scatter plot above shows that, as the aptitude score increases job proficiency also increases.
- For a given aptitude score, the job proficiency can be estimated and vice-a-versa using the regression line.

# Scatter Plot Matrix

Scatter Plot Matrix gives the Scatterplot diagram of multiple variables with each other, all in one chart. It is used to determine if you have a linear correlation between multiple variables.



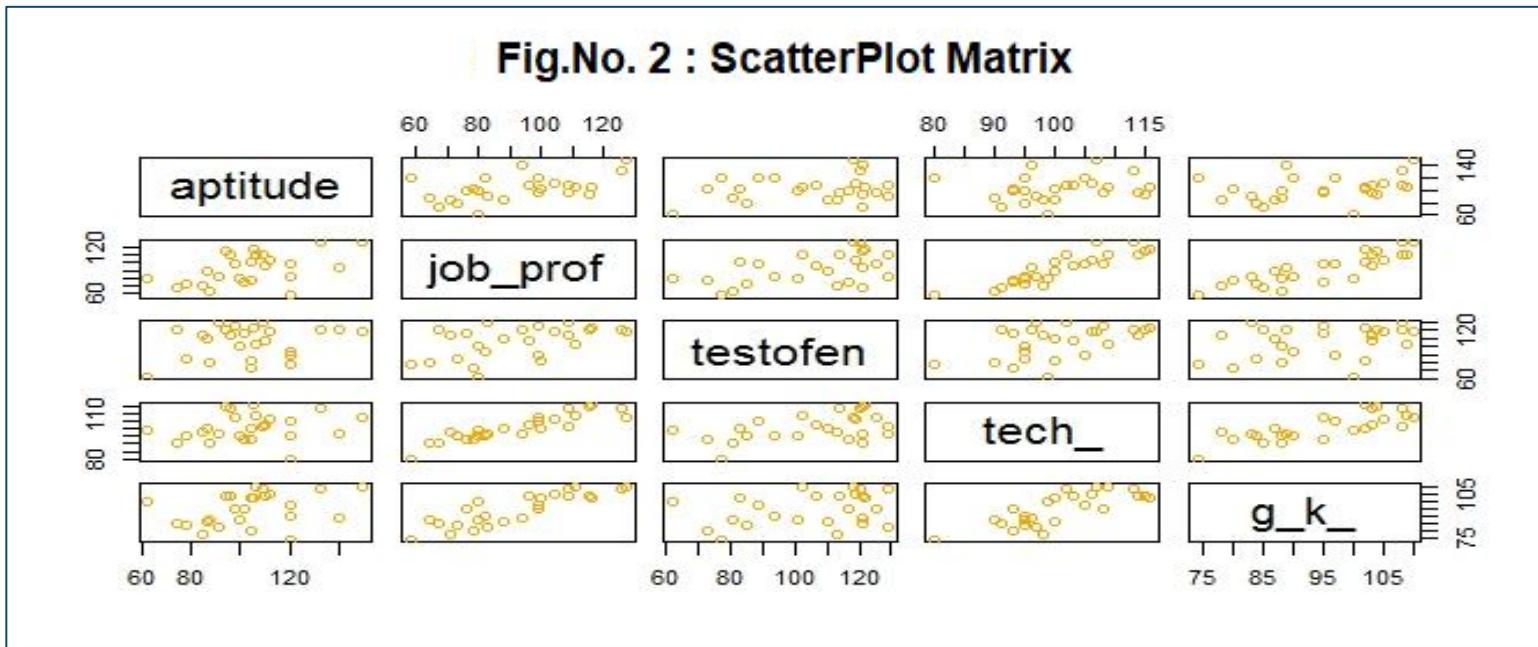
```
# ScatterPlot Matrix
```

```
pairs(~aptitude+job_prof+testofen+tech_+g_k_, data=job, main="Fig.No. 2 :  
ScatterPlot Matrix", col="darkorange")
```

- ❑ **pairs()** in base R are used to plot pairwise comparison
- ❑ ~ each variable name to be plotted followed by a “+” sign needs to mentioned
- ❑ **main=** provides the user defined name of the chart.

# Scatter Plot Matrix in R

```
# Output
```



## Interpretation :

- Scatter plot matrix above shows that, as the aptitude score, English language score, technical score and general knowledge score increases job proficiency also increases.
- Technical score and GK score has slight positive relation but other variables are not related to each other.

# Scatter Plot Matrix in R using package “GGally”

```
#Installing and calling the package “Ggally” :
```

```
install.packages("GGally")
library(GGally)
```

GGally is the best package we can use to plot an effective  
Scatter Plot in R

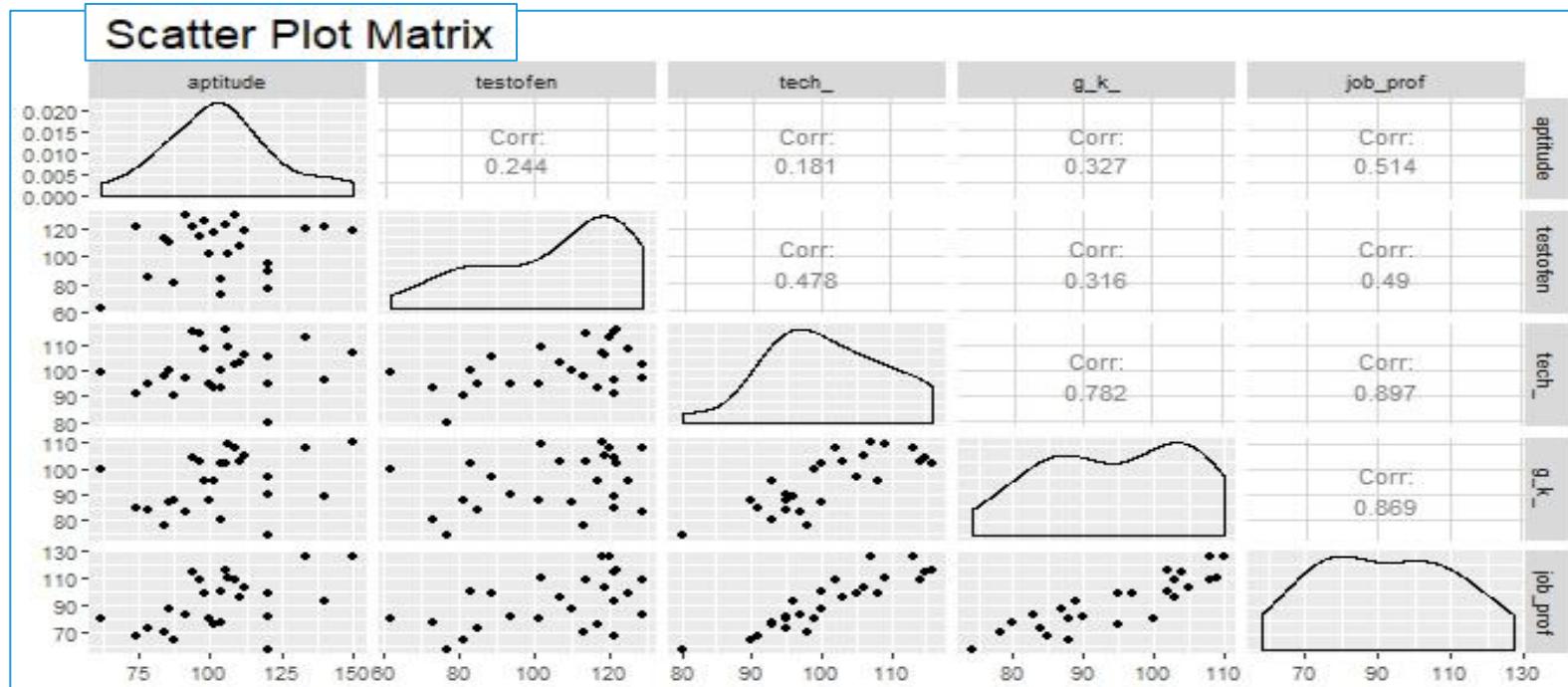
```
#ScatterPlot Matrix
```

```
ggpairs(job[,c("aptitude","testofen","tech_","g_k_","job_prof")],
title = "Scatter Plot Matrix")
```

- **ggpairs()** is the function used to call the variables for which the pairwise comparison chart needs to be plotted.
- **job[]** is the name of the data of which the variables need to be plotted
- **title =** provides the user defined name of the chart

# Scatter Plot Matrix in R using package “GGally”

This plot shows the strength of relation through correlation coefficient and also the distribution of each variable.



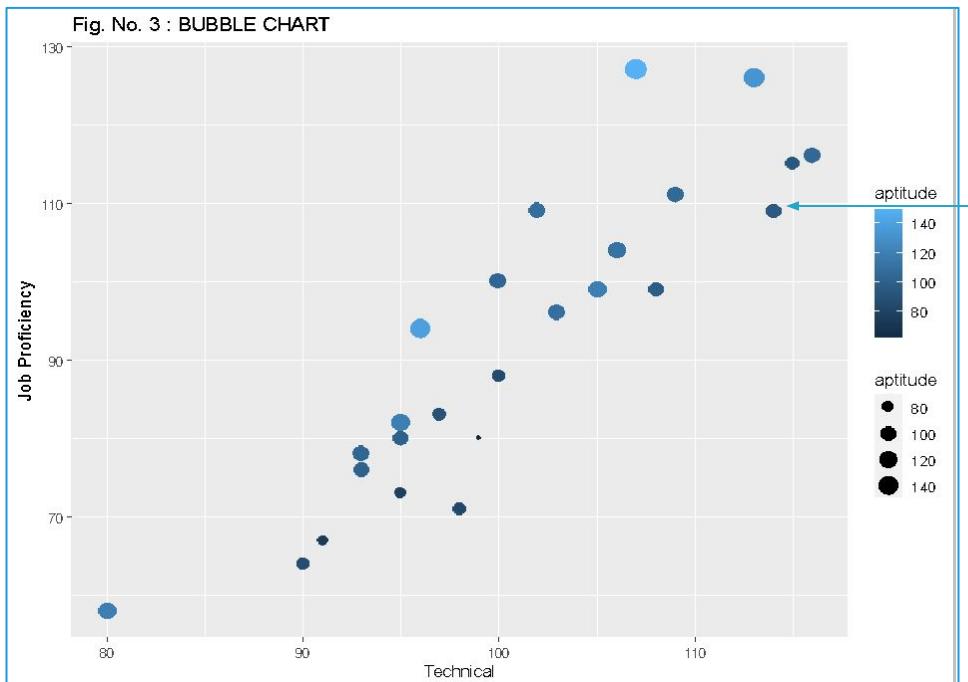
## Interpretation :

- Technical and GK score have high correlation with job proficiency as compared to other variables.
- Technical and GK score also share high positive relation with each other.
- Aptitude score graph is symmetric.

# Bubble Chart

Bubble chart is generally used instead of a scatter plot if your data has three data series that each contain a set of values.

The sizes of the bubbles are determined by the values in the third data series.



## Interpretation :

- Here we observe that as Technical score increases Job Proficiency also increases , however, Aptitude score does not show any such consistent direction.

# Bubble Chart in R

Package `ggplot2` will be used to create a bubble chart.

```
# Bubble Chart
```

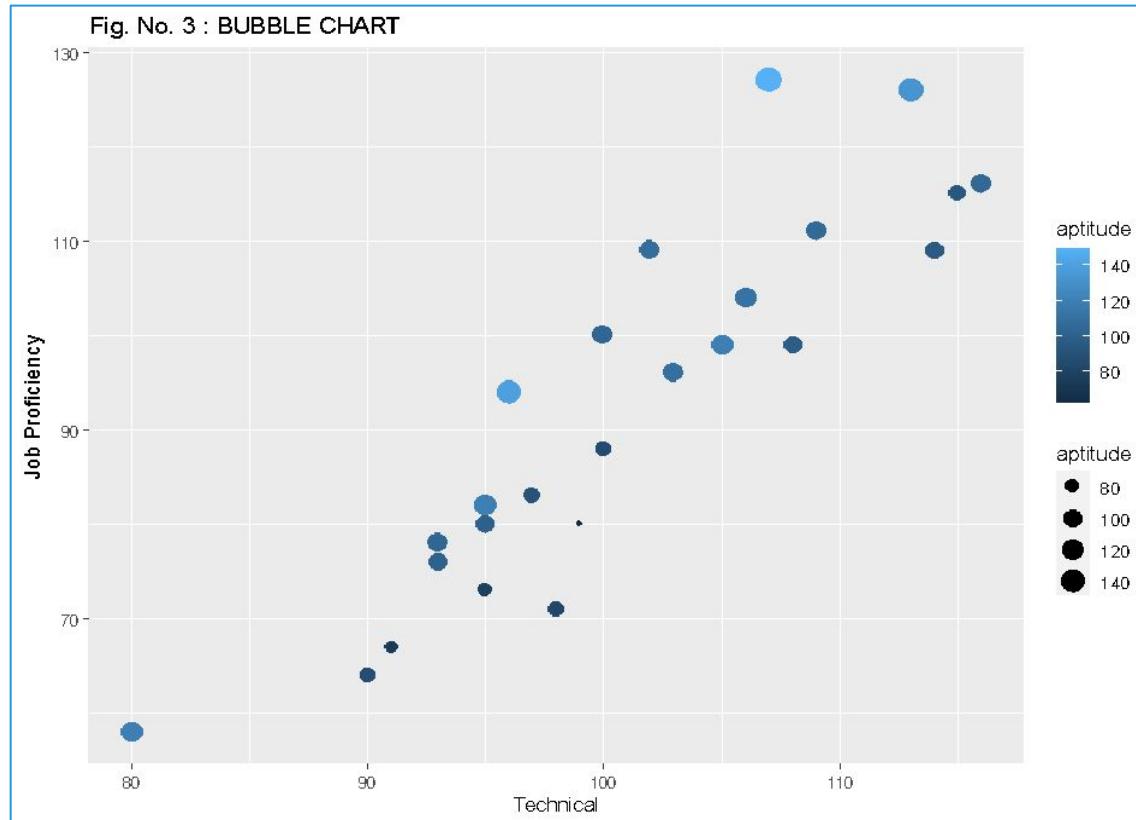
```
install.packages("ggplot2")
library(ggplot2)

qplot(x=tech_, y=job_prof, data=job, color=aptitude, size=aptitude,
xlab="Technical", ylab="Job Proficiency", main="Fig. No. 3 :
BUBBLE CHART")
```

- ❑ `qplot()` in package `ggplot2` is used to plot any ‘quick plot’
- ❑ `x,y` variables to be plotted on x and y axis
- ❑ `data=` data to be used for plotting
- ❑ `color=` variable to be considered for the colour of the bubble
- ❑ `size=` variable to be considered for the size of the bubble
- ❑ `xlab,ylab` labels for x and y axes
- ❑ `main=` provides the user defined name of the chart

# Bubble Chart in R

# Output



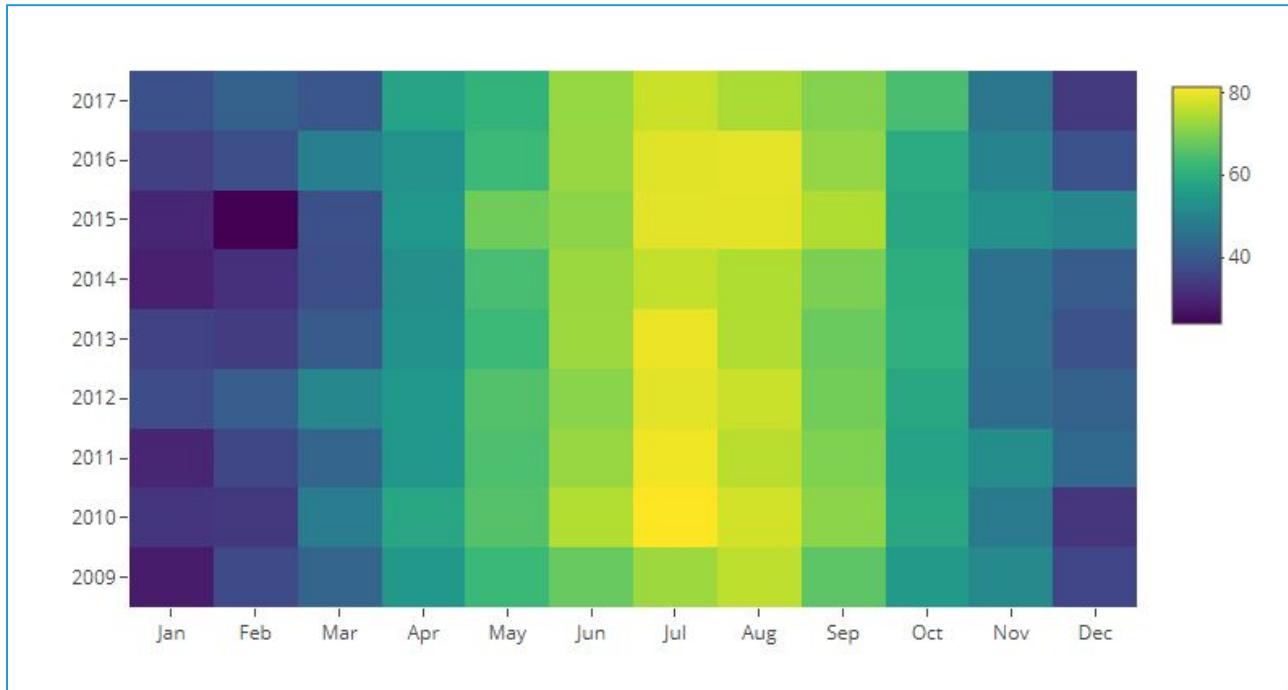
## Interpretation :

- Here we observe that as Technical score increases Job Proficiency also increases, however, Aptitude score does not show any such consistent direction.

# Heat Map

A Heat Map is a graphical representation of data where the individual values contained in a matrix are represented as colors.

It gives us quick information through color patterns.



In the example given above , we can see the temperature fluctuation in NY across months over the years

# Case Study

To get a better understanding of the subject, we shall consider the below case as an example.

## Background

NY Temperature varies across months over the years

## Objective

To visually see the hottest months in the years  
To see how temperature has fluctuated over the years

## Sample Size

108

# Data Snapshot

## Average Temperatures in NY

Variables

Observations

Year	Month	Temperature
2009	Jan	27.9
2009	Feb	36.7
2009	Mar	42.4
2009	Apr	54.5
2009	May	62.5
2009	Jun	67.5

Columns	Description	Type	Measurement	Possible values
Year	Years listed from 2009-2017	Categorical	2009 – 2017	9
Month	Months of the year	Categorical	Jan - Dec	12
Temperature	Average Temperature in degree Fahrenheit	Numeric	-	-

# Heat Map in R

```
# Installing and calling the package
```

```
install.packages("plotly")
library(plotly)
```

```
# Importing Data and Arranging the Months in the right order :
```

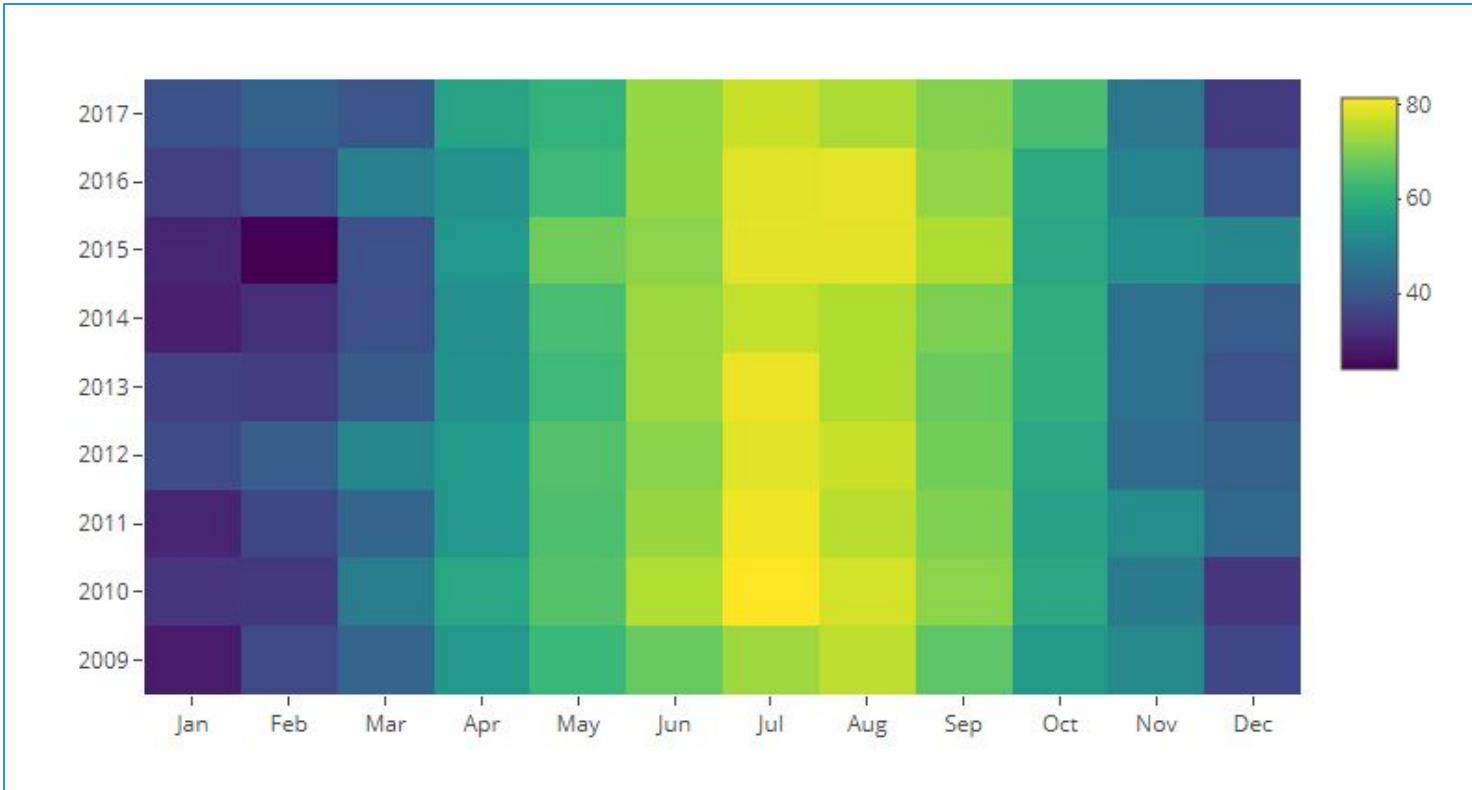
```
heatmapdata<-read.csv("Average Temperatures in NY.csv", header=TRUE)
heatmapdata$Month<-factor(heatmapdata$Month, level=unique(heatmapdata$Month))
```

```
# Heat Map
```

```
plot_ly(heatmapdata, x=heatmapdata$Month, y=heatmapdata$Year,
z=heatmapdata$Temperature,
type="heatmap", connectgaps=FALSE, showscale=T)
```

# Heat Map in R

# Output for Heat Map :

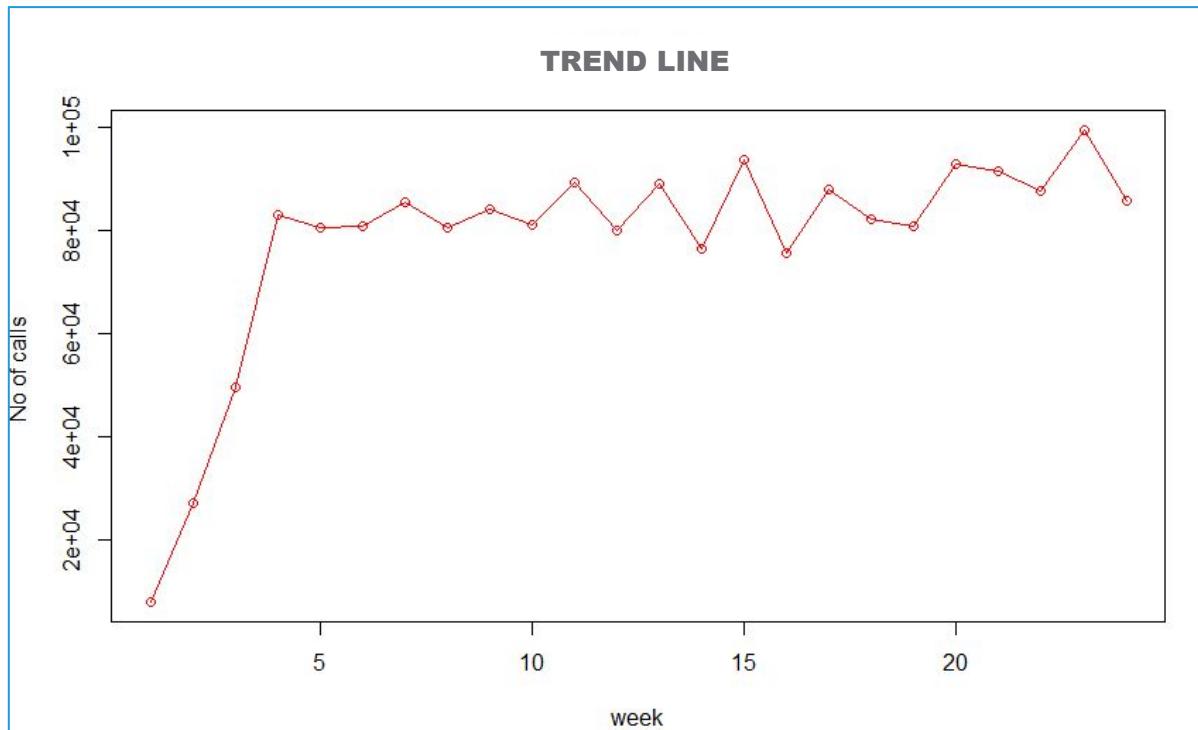


## Interpretation :

- Heat map above shows that July is the hottest season across the year .
- 2015 showed a longer hot period as compared to other years extending from may to September

# Trend Line

- A Trend Line is a straight line that connects two or more data points and then extends into the future to act as a line of support or resistance.
- It is usually used to plot something over time . It can be used to estimate the future values too



We can observe the increase and decrease in the total number of calls over a period of 24 weeks

# Case Study

To get a better understanding of the subject, we shall consider the below case as an example.

## Background

Telecom Weekly Data for 24 weeks

## Objective

To visually observe the trend of total calls over 24 weeks

## Sample Size

21902

# Data Snapshot

Plotting a trendline requires time-element. Consider the following datasets. Week can be taken as the time element.

Variables					
Observations	CustID	Week	Calls	Minutes	Amt
	1001	1	56	292	70.4
Columns	Description	Type	Measurement	Possible values	
CustID	Customer ID	Numeric	-	-	
Week	Week no.	Numeric	1-24	24	
Calls	No. of Calls	Numeric	-	positive values	
Minutes	Total Minutes	Numeric	Minutes	positive values	
Amt	Amount Charged	Numeric	Rs.	positive values	

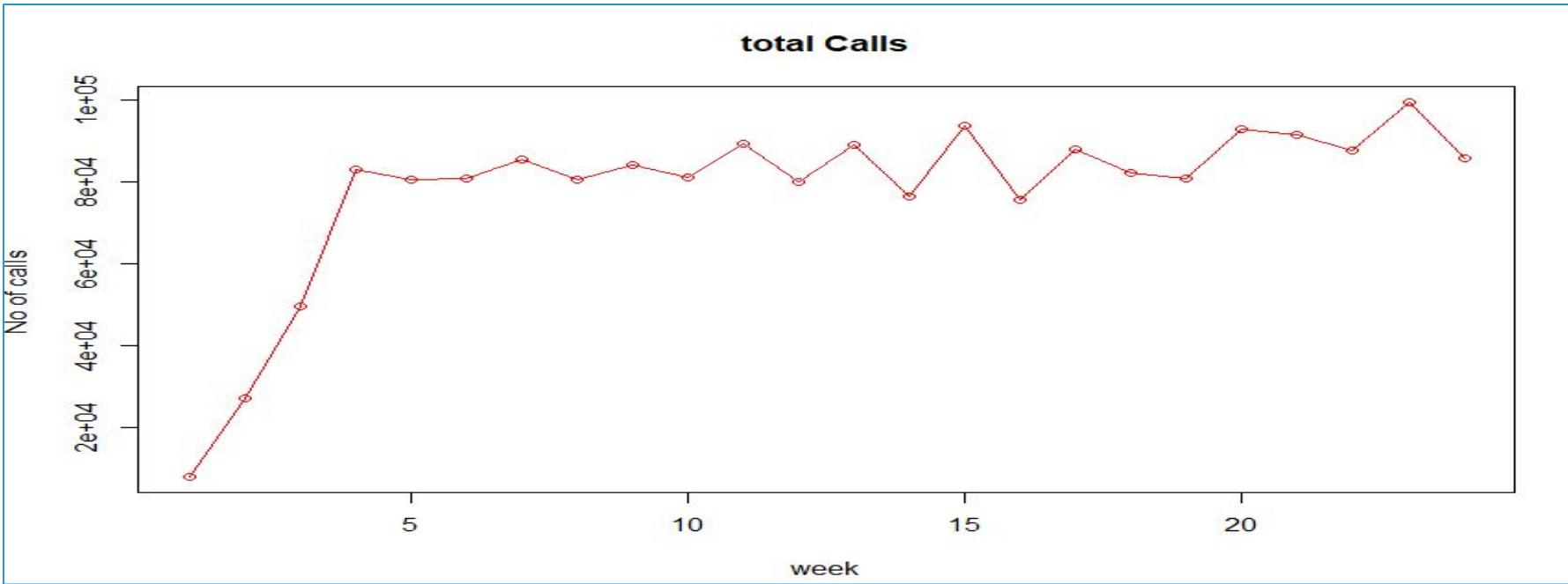
# Trend Line in R

```
# Importing Data  
transaction<-read.csv("TelecomData_WeeklyData.csv", header=TRUE)  
  
# Merging and Formatting Data  
trend<-aggregate(Calls~Week, data=transaction, FUN=sum)  
  
# Trend Line  
plot(trend, type = "o", col = "red", xlab = "week", ylab = "No of calls",  
main = "total Calls")
```

- The basic function is `plot(v,type,col,xlab,ylab )`
- `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.
- `col` is used to give colors to both the points and lines.
- `xlab()` is the label for x axis.
- `ylab()` is the label for y axis.
- `main` is the Title of the chart.

# Trend Line in R

# Output

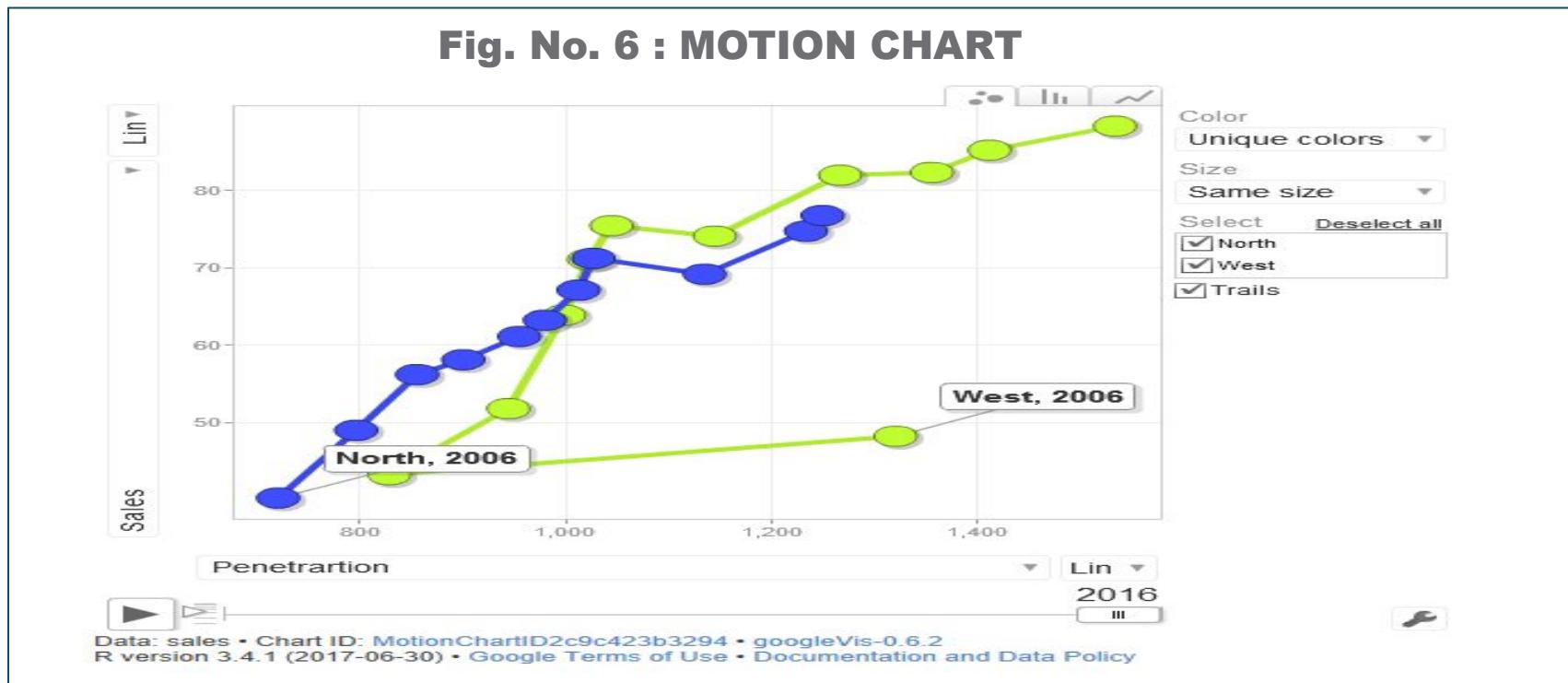


## Interpretation :

- Upto first 4 weeks, number of calls increases continuously. After 5<sup>th</sup> week there are more ups and down in number of calls among customers.

# Motion Chart

- A Motion Chart is a dynamic bubble chart which allows efficient and interactive exploration and visualization of longitudinal multivariate Data.
- It allows you to plot the dimension values in your report against up to four metrics across time.



# Case Study

To get a better understanding of the subject, we shall consider the below case as an example.

## Background

Sales Data & its penetration in each Region over the years

## Objective

To visually observe the sales & penetration in motion over the years

## Sample Size

22

# Data Snapshot

Sales Data (Motion Chart)

Variables

Year	Region	Sales	Penetrartion
2006	North	40.23	721

Observations

Columns	Description	Type	Measurement	Possible values
Year	Year	Numeric	2006-2016	11
Region	Region	Categorical	North,West	2
Sales	Sales in a particular Year	Numeric	Rs.	Positive values
Penetration	Penetration in a particular Year	Numeric	-	Positive values

# Motion Chart in R

```
#Importing Data
```

```
sales<-read.csv("Sales Data (Motion Chart).csv", header=TRUE)
```

```
#Installing and calling the package
```

```
install.packages("googleVis")
library(googleVis) ↑
```

googleVis is the best package we can use to plot an effective Motion Chart in R

```
# Motion Chart
```

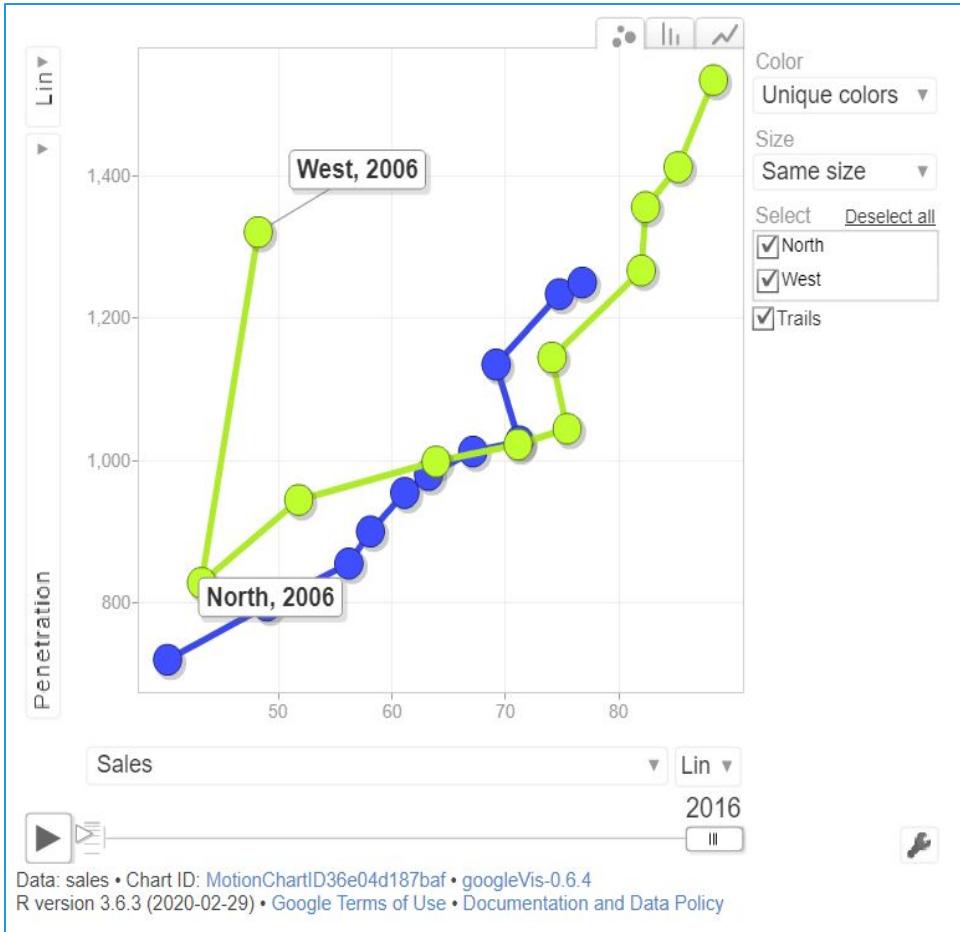
```
mchart<-gvisMotionChart(sales, idvar="Region", timevar="Year")
plot(mchart)
```

- **gvisMotionChart()** is the function used to create a motion chart
- **sales** is the data that is used
- **idvar=** inputs the id variable
- **timevar=** inputs the time variable

plot() is used to plot the final Motion Chart

# Motion Chart in R

```
# Output
```



## Interpretation :

- Over time, as sales increases penetration has also increased parallelly for both the Regions.

# Browser Settings for Motion Chart

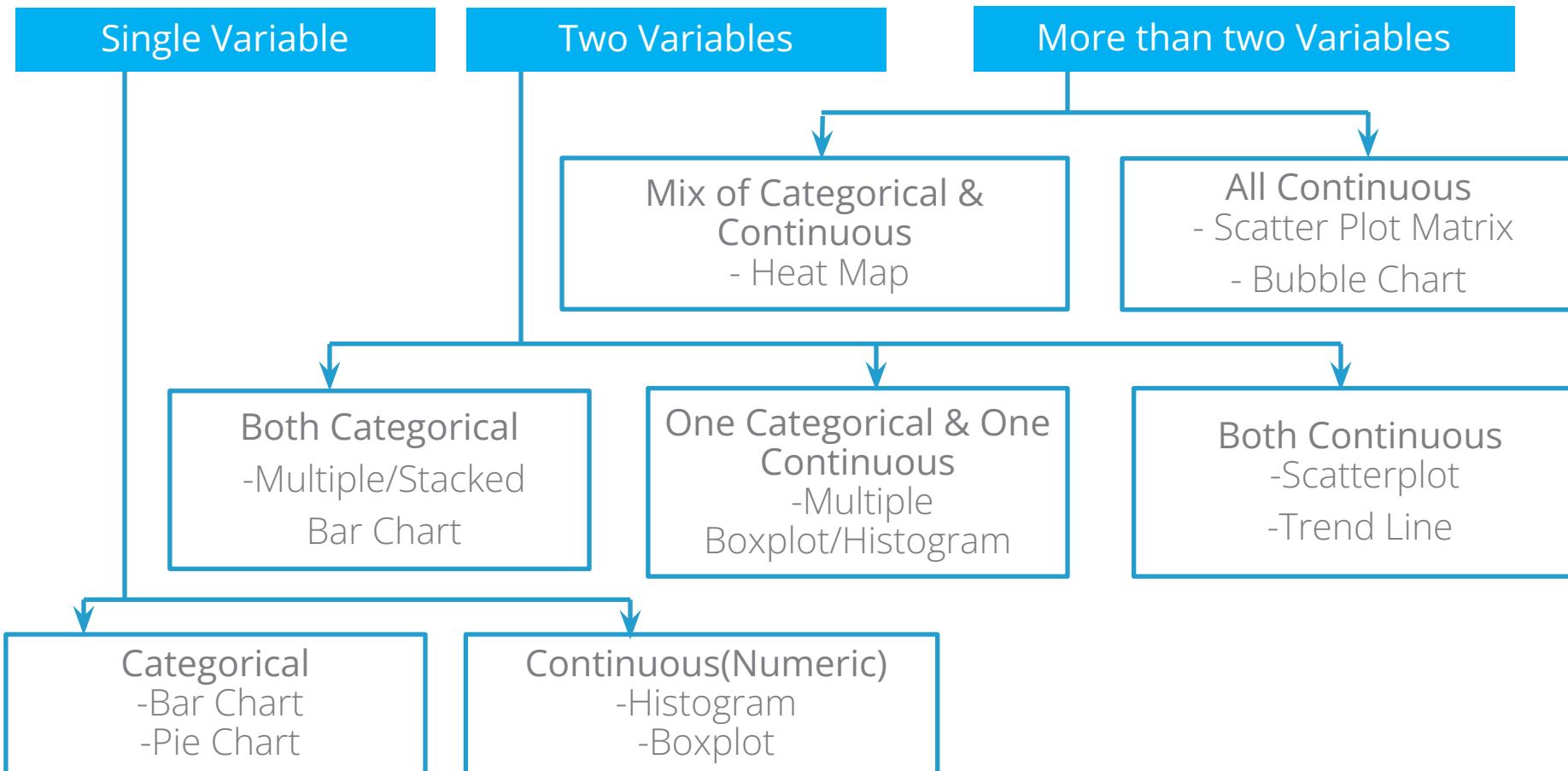
Incase you do not get the output for Motion Chart you will have to do changes in your Chrome Browser settings as follows :

1. Go to the website which opened when you executed the Motion Chart code.
2. To the left of the web address, click the icon that you see: Lock  , Info  or Dangerous  .
3. Click Site settings.
4. In permission setting change Flash to "allow". Your changes will save automatically.
5. Then go back to Motion Chart web page & Reload it, you will be able to see the Motion Chart.

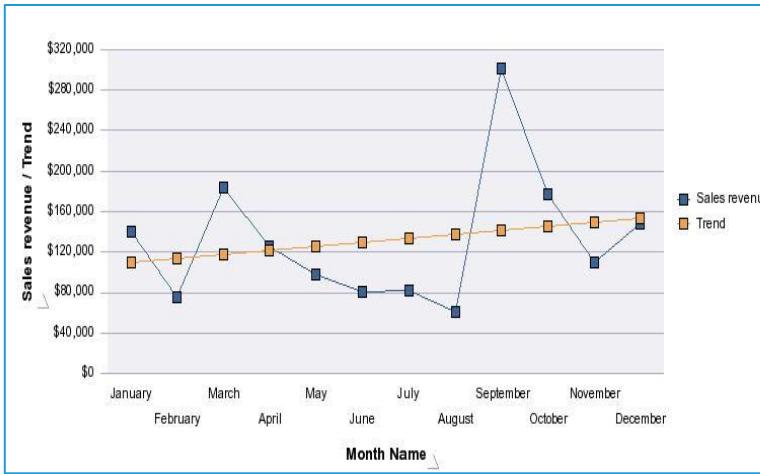
If you are using any other browser then make sure that flash player is enabled and updated.

# Get an Edge!

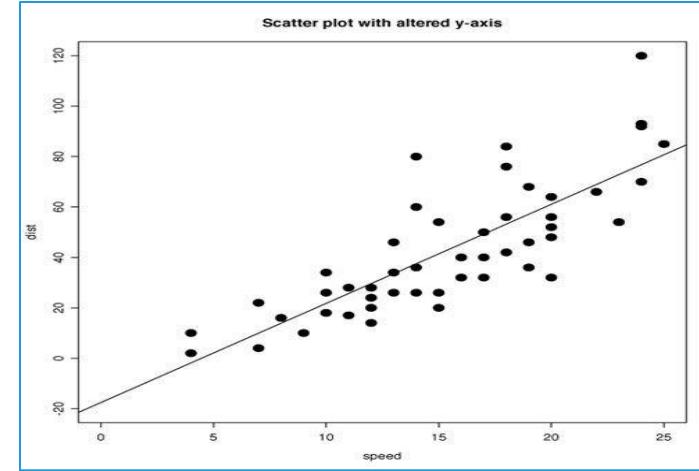
Choosing the right graph



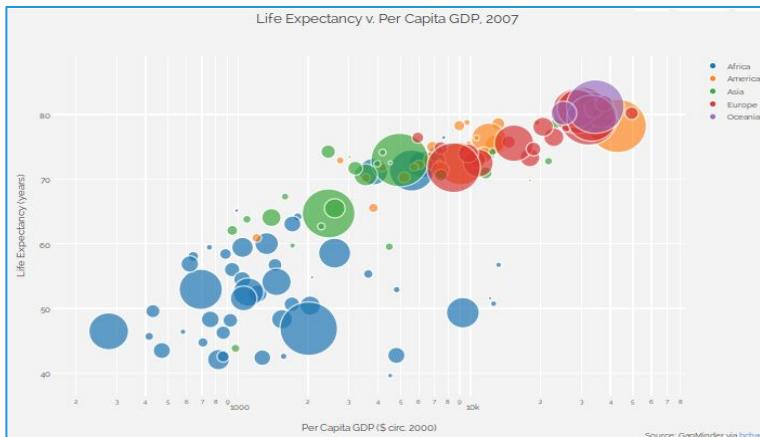
# Application Areas



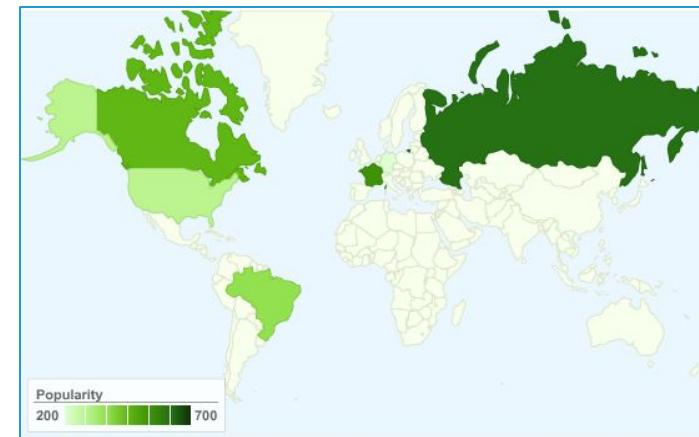
## Sales Trends



## Distance v/s Speed



## Economics



## Geographical Studies

# Quick Recap

## Chart Types and Functions in R

- Scatterplot with Regression Line – **plot()** + **abline()**
- Scatterplot Matrix – **pairs()** or **ggpairs()** from package "GGally"
- Bubble Chart – **qplot()** from package "ggplot2"
- Heat Map – **plot\_ly()** from package "plot.ly"
- Trend Line – **plot()**
- Motion Chart - **gvisMotionChart()** from package "googleVis"

# v2 Visualisations

## using ggplot2 in R

# Contents

- 1.** What is ggplot2?
2. Summarizing Data in Diagrams using ggplot2
  - i.** Bar Charts
  - ii.** Pie Chart
  - iii.** Box-Whisker Plot
  - iv.** Histogram
  - v.** Scatterplot with Regression Line
  - vi.** Trend Line

# What is ggplot2?

- The ggplot2 package, created by Hadley Wickham, offers a powerful graphics language for creating elegant and complex plots.
- Originally based on Leland Wilkinson's The Grammar of Graphics, ggplot2 allows you to create graphs that represent both univariate and multivariate, numerical and categorical data in a straightforward manner.
- Grouping can be represented by color, symbol, size, and transparency.
- This package is available from CRAN via
  - `install.packages("ggplot2")`
  - `library(ggplot2)`

\*Website: <http://ggplot2.org> (better documentation)

# Case Study - 1

To get a better understanding of the subject, we shall consider the below case as an example.

## Background

A telecom service provider has the Demographic and Transactional information of their customers

## Objective

To look at the distribution of customer database  
To see how the Calls and Amount are distributed across customers

## Sample Size

1000

# Data Snapshot

telecom data

Variables

Observations

CustID	Age	Gender	PinCode	Active	Calls	Minutes	Amt	AvgTime	Age_Group
1001	29	F	186904	Yes	2247	18214	3168.76	8.105919	18-30
Columns	Description			Type	Measurement		Possible values		
CustID	Customer ID			Numeric	-		-		
Age	Age of the Customer			Numeric	-		-		
Gender	Gender of the Customer			Categorical	M, F		2		
PinCode	Pincode of area			Numeric	-		-		
Active	Active usage of telecom			Categorical	Yes, No		2		
Calls	Number of Calls made			Numeric	-		positive values		
Minutes	Number of minutes spoken			Numeric	minutes		positive values		
Amt	Amount charged			Continuous	Rs.		positive values		
AvgTime	Mean Time per call			Continuous	minutes		positive values		
Age Group	Age Group of the Customer			Categorical	18-30, 30-45, >45		3		

# Diagrams in R

```
# Importing Data
```

```
telecom<-read.csv("telecom.csv", header=TRUE)
```

```
# Installing and calling the package
```

```
install.packages("ggplot2")
library(ggplot2)
```

# Simple Bar Chart in R

```
# Simple Bar Chart (Age Group)
```

```
ggplot(telecom,aes(x=Age_Group,y=Calls))+  
geom_bar(stat="identity",fill="darkorange")+labs(x="Age Groups",y="Total  
Calls",title="Fig. No. 1 : Simple Bar Diagram(Age Group)")
```

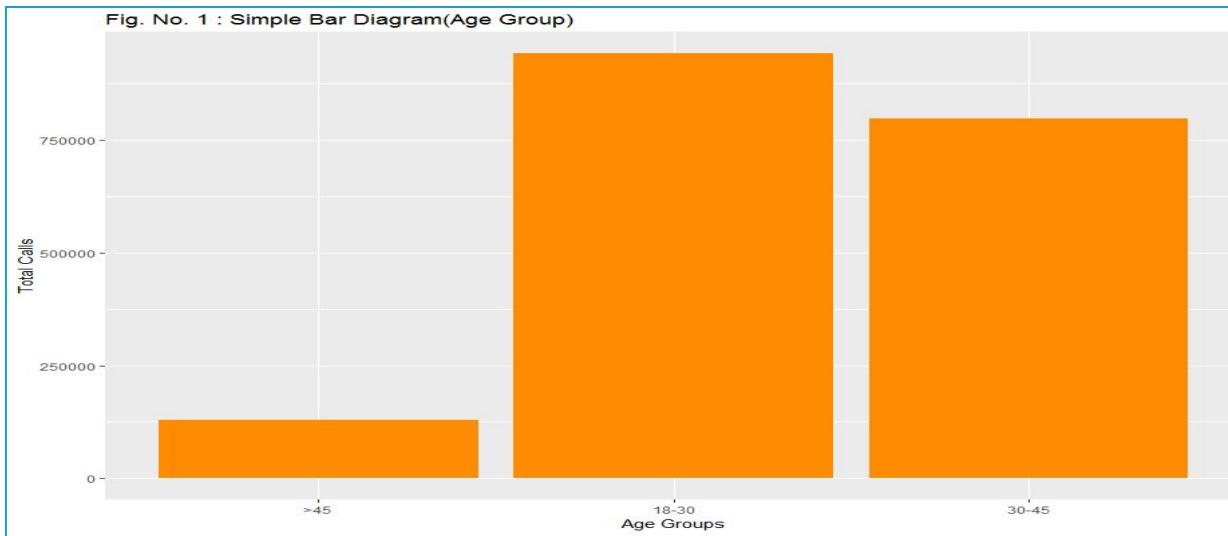
- ❑ **ggplot()** is a function in ggplot2 which yields different types of plots
- ❑ **telecom** is the data that is used
- ❑ **aes()** specifies the variables to be used on each axis
- ❑ **geom\_bar()** makes the height of the bar proportional to the number of cases in each group
- ❑ **stat="identity"** is used to represent the height of the bar which represent values in the data
- ❑ **labs()** is used to label the various features of the graph



In **geom\_bar()**, **stat="bin"** is the default function which represents count of cases in each group

# Simple Bar Chart in R

This is the output that you get on running the previous code



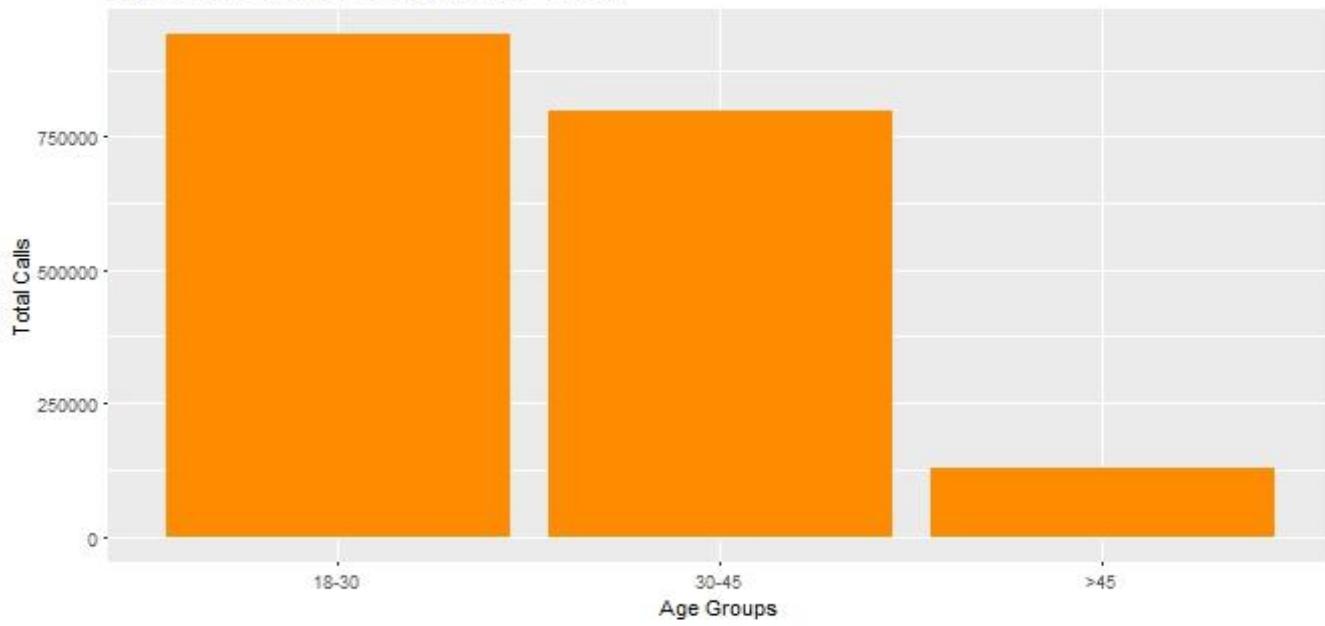
To get the bars in proper order, we will have to re-order the levels of column "Age\_Group" in telecom data as follows & then run the same ggplot code :

```
telecom$Age_Group <- factor(telecom$Age_Group, levels = c("18-30", "30-45", ">45"))
```

# Simple Bar Chart in R

```
# Output is a ordered bar graph :
```

Fig. No. 1 : Simple Bar Diagram(Age Group)



# Simple Bar Chart in R

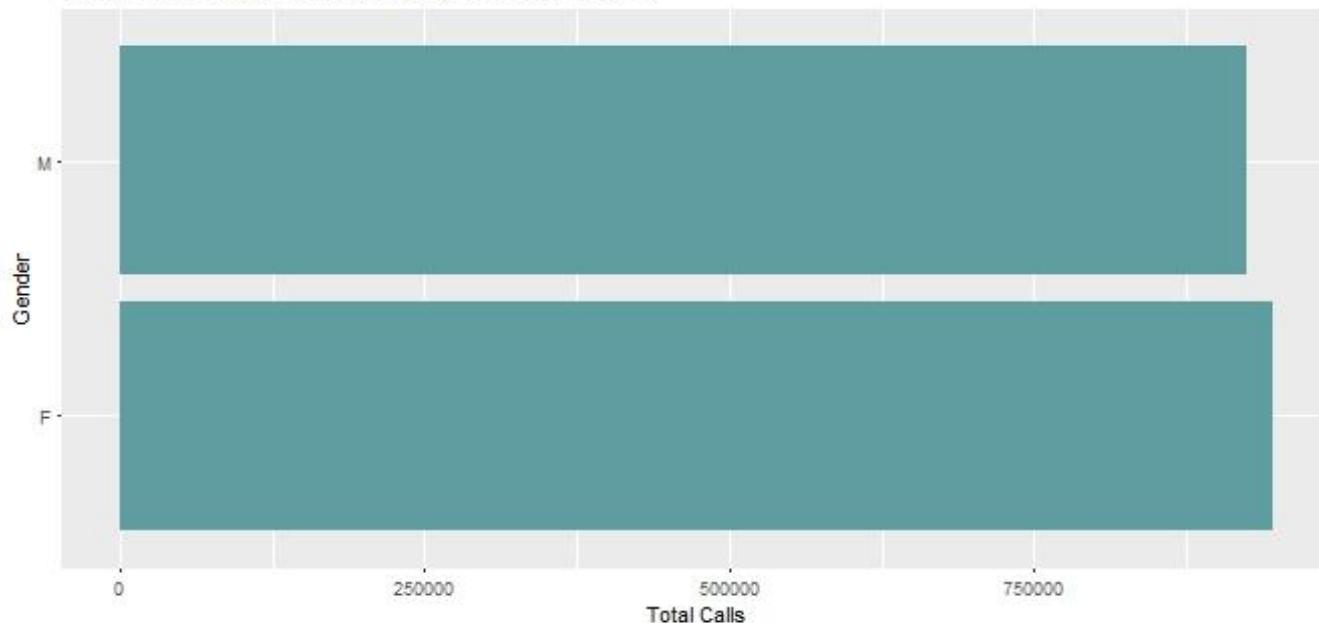
```
# Simple Bar Chart (Gender) - Horizontal  
  
ggplot(telecom, aes(x=Gender, y=Calls))+  
  geom_bar(stat="identity",fill="cadetblue") +  
  labs(x="Gender",y="Total Calls",  
       title="Fig. No. 2 : Simple Bar Diagram(Gender)-Horizontal") +  
  coord_flip()
```

- ❑ **coord\_flip()** gives us horizontal bars by flipping the co-ordinates.

# Simple Bar Chart in R

```
# Output
```

Fig. No. 2 : Simple Bar Diagram(Gender)-Horizontal



# Stacked Bar Chart in R

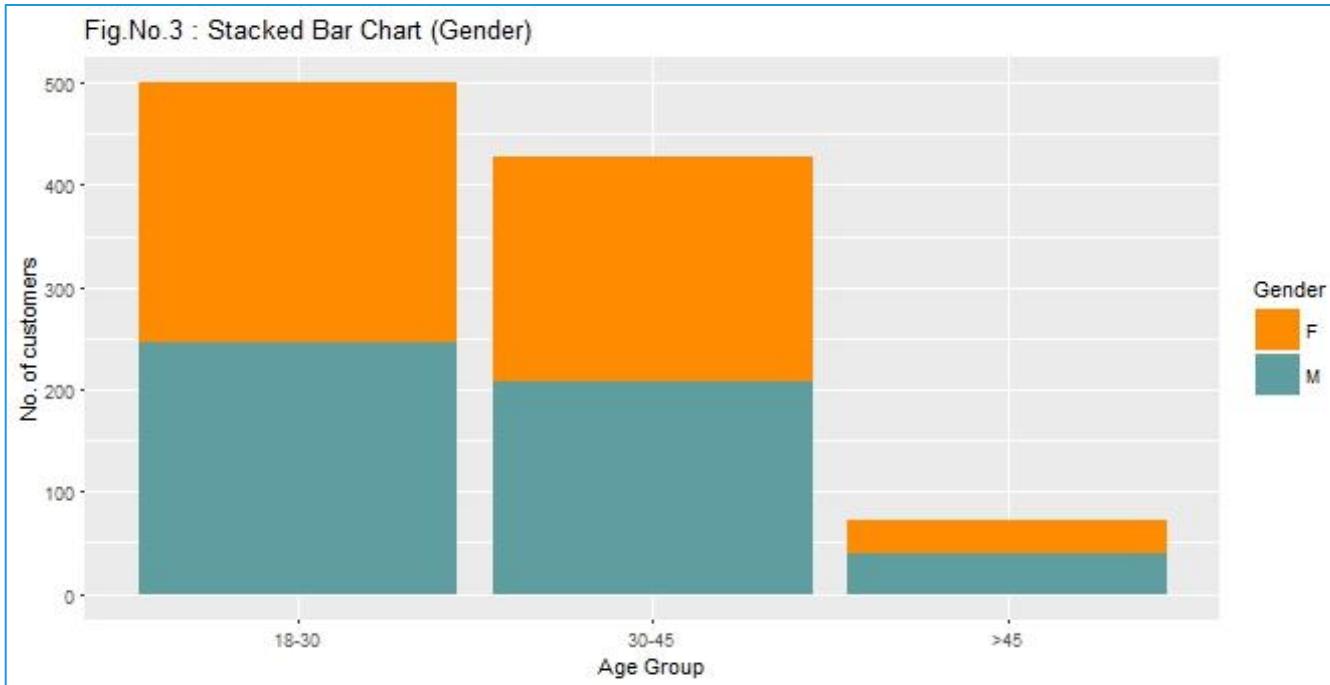
```
# Stacked (or Sub-Divided) Bar Chart
```

```
ggplot(telecom, aes(x=Age_Group))+ geom_bar(aes(fill=Gender))+  
  labs(x="Age Group", y="No. of customers",title="Fig.No.3 : Stacked Bar  
  Chart (Gender)")+scale_fill_manual(values=c("darkorange","cadetblue"))
```

- **aes()** function in **geom\_bar()** divides each bar as per the input variable using **fill= Gender**
- **scale\_fill\_manual()** allows to use the user defined colors for the sub divided bar

# Stacked Bar Chart in R

```
# Output
```



# Multiple Bar Chart in R

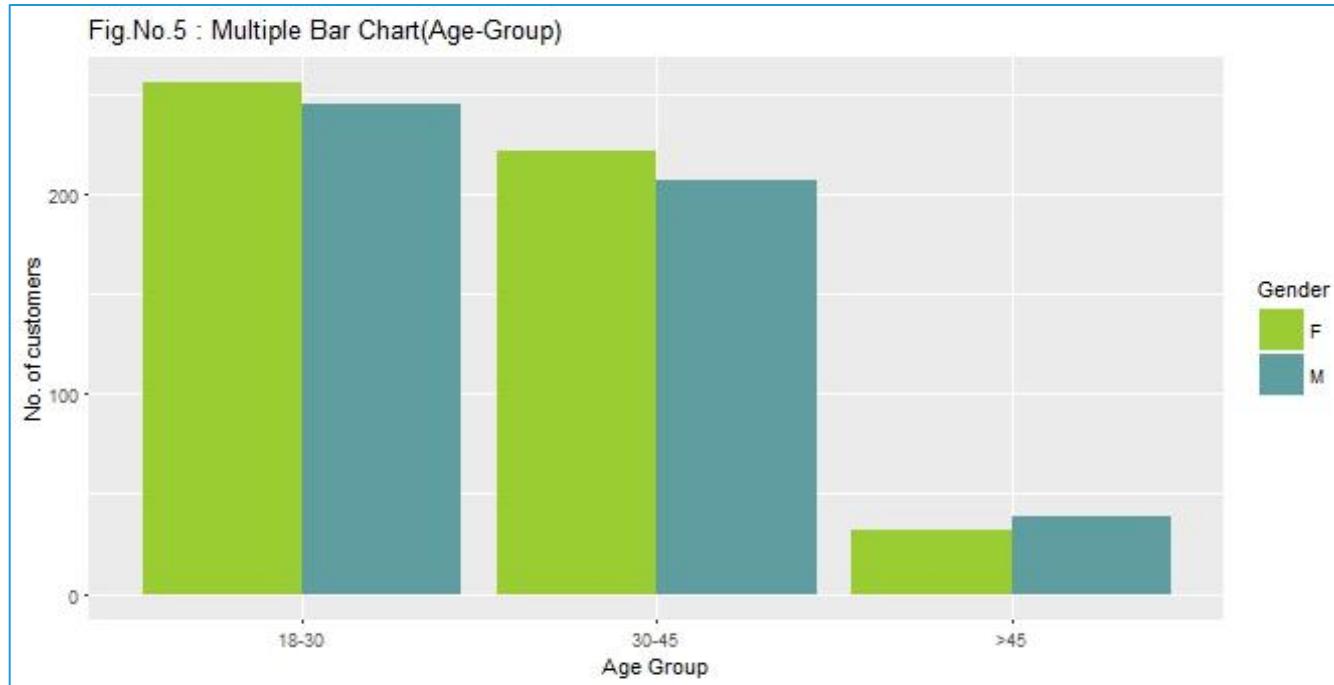
```
# Multiple (or Grouped) Bar Chart
```

```
ggplot(telecom, aes(x=Age_Group))+geom_bar(aes(fill=Gender),position="dodge")  
+ labs(x="Age Group", y="No. of customers",title="Fig.No.5 : Multiple Bar  
Chart(Age-Group)")+ scale_fill_manual(values=c("yellowgreen","cadetblue"))
```

- ❑ **position="dodge"** gives us the divided bars one beside the other

# Multiple Bar Chart in R

# Output



# Pie Chart in R

```
# Pie Chart
```

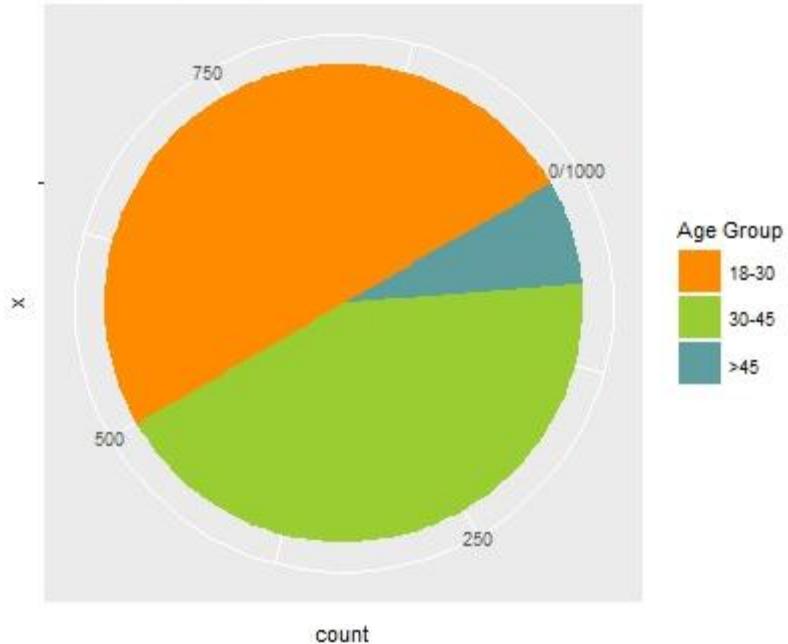
```
ggplot(telecom, aes(x="", fill=Age_Group))+ geom_bar(width=1)+  
coord_polar(theta="y", start=pi/3)+labs(title="Fig.No. 7 : Pie chart",  
fill="Age Group")+scale_fill_manual(values=c("darkorange","yellowgreen",  
"cadetblue"))
```

- ❑ **coord\_polar()** it transforms stacked bar charts into circular pie chart
- ❑ **theta="y"** uses Y axis scale for proportion
- ❑ **start=pi/3** it starts the first proportion of pie from pi/3 angle

# Pie Chart in R

```
# Output
```

Fig.No. 7 : Pie chart



# Box Plot in R

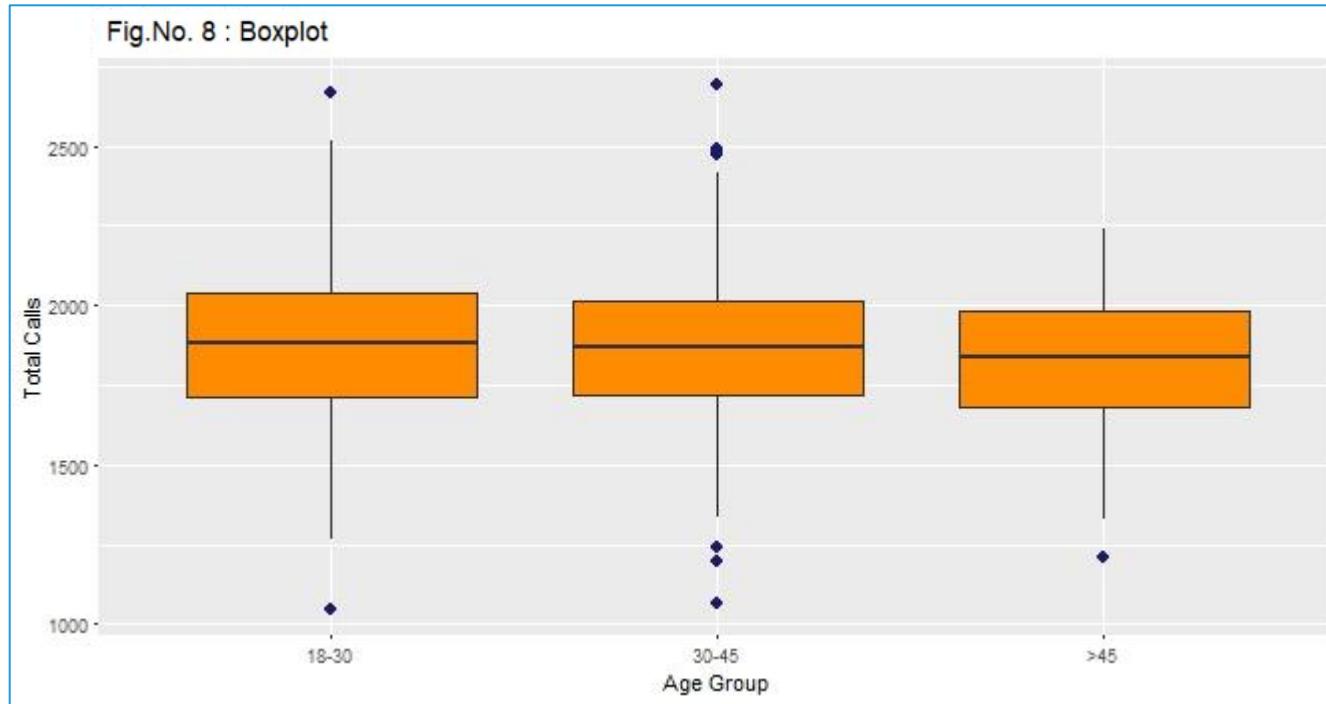
```
# Box Plot
```

```
ggplot(telecom, aes(x=Age_Group, y=Calls))+ geom_boxplot(fill="darkorange",  
outlier.colour="midnightblue", outlier.size=2.5)+labs(x="Age Group", y="Total  
Calls", title="Fig.No. 8 : Boxplot")
```

- geom\_boxplot ()** calls the boxplot function

# Box Plot in R

```
# Output
```



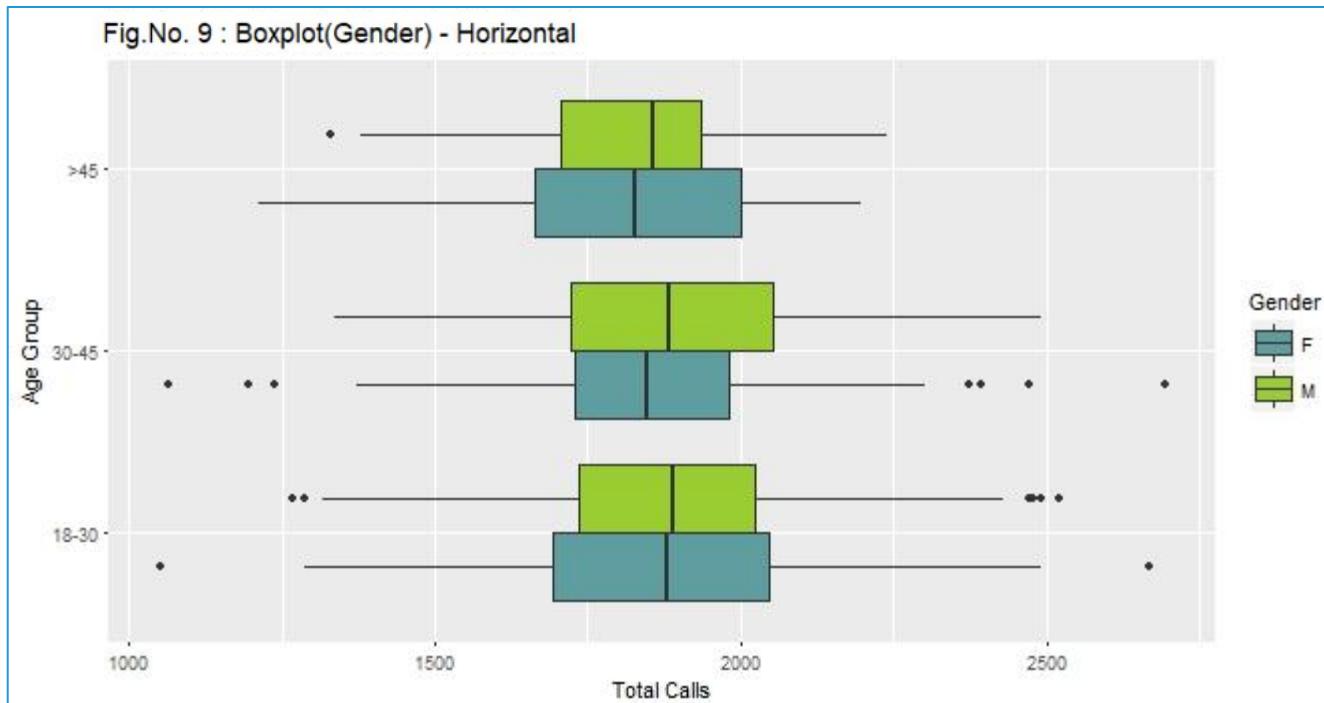
# Box Plot in R

```
# Box Plot (Gender) - Horizontal  
  
ggplot(telecom, aes(x=Age_Group, y=Calls))+ geom_boxplot(aes(fill=Gender))+  
  labs(y="Total Calls", x="Age Group",title="Fig.No. 9 : Boxplot(Gender) -  
  Horizontal") +scale_fill_manual(values=c("cadetblue","yellowgreen"))  
+coord_flip()
```

- ❑ **aes()function in geom\_boxplot()** gives multiple boxplot one beside the other using **fill= Gender**

# Box Plot in R

```
# Output
```



# Histogram in R

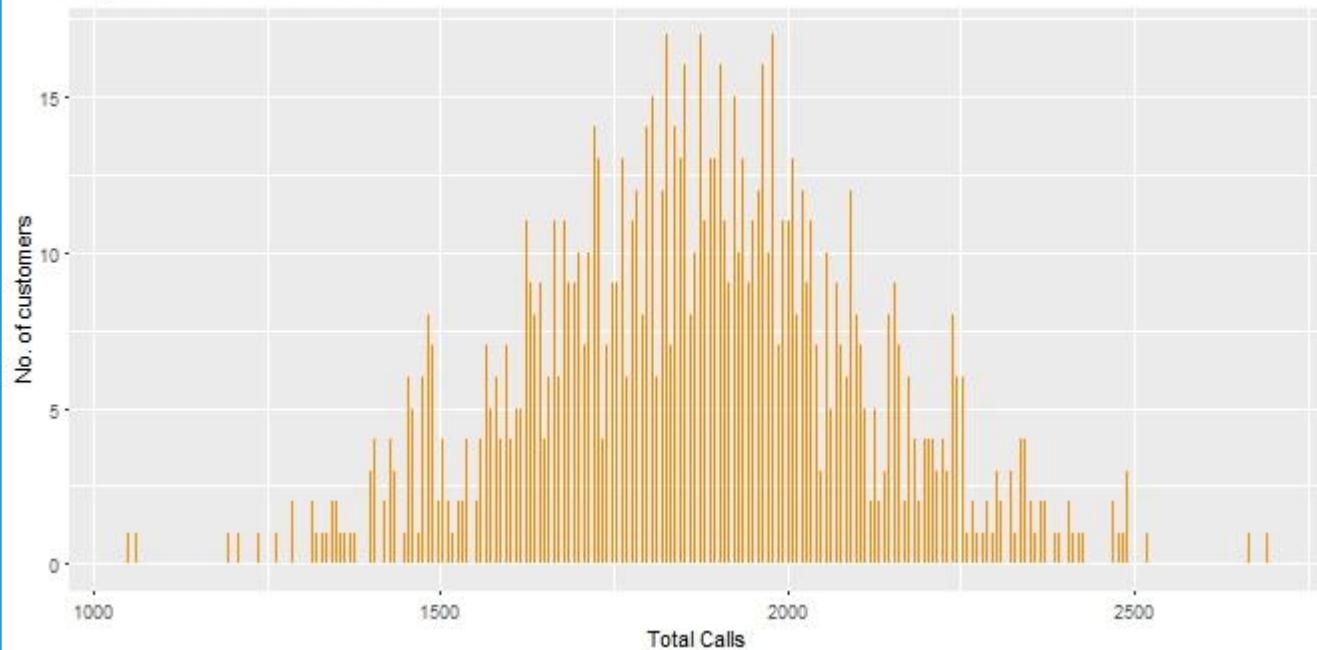
```
# Histogram  
ggplot(telecom, aes(x=Calls))+ geom_histogram(binwidth=2,  
fill="darkorange")+labs(x="Total Calls", y="No. of customers", title="Fig. No.  
10 : HISTOGRAM")
```

- **geom\_histogram()** is used to plot histogram
- **binwidth=** gives size to each bar in the graph

# Histogram in R

```
# Output
```

Fig. No. 10 : HISTOGRAM



# Histogram in R

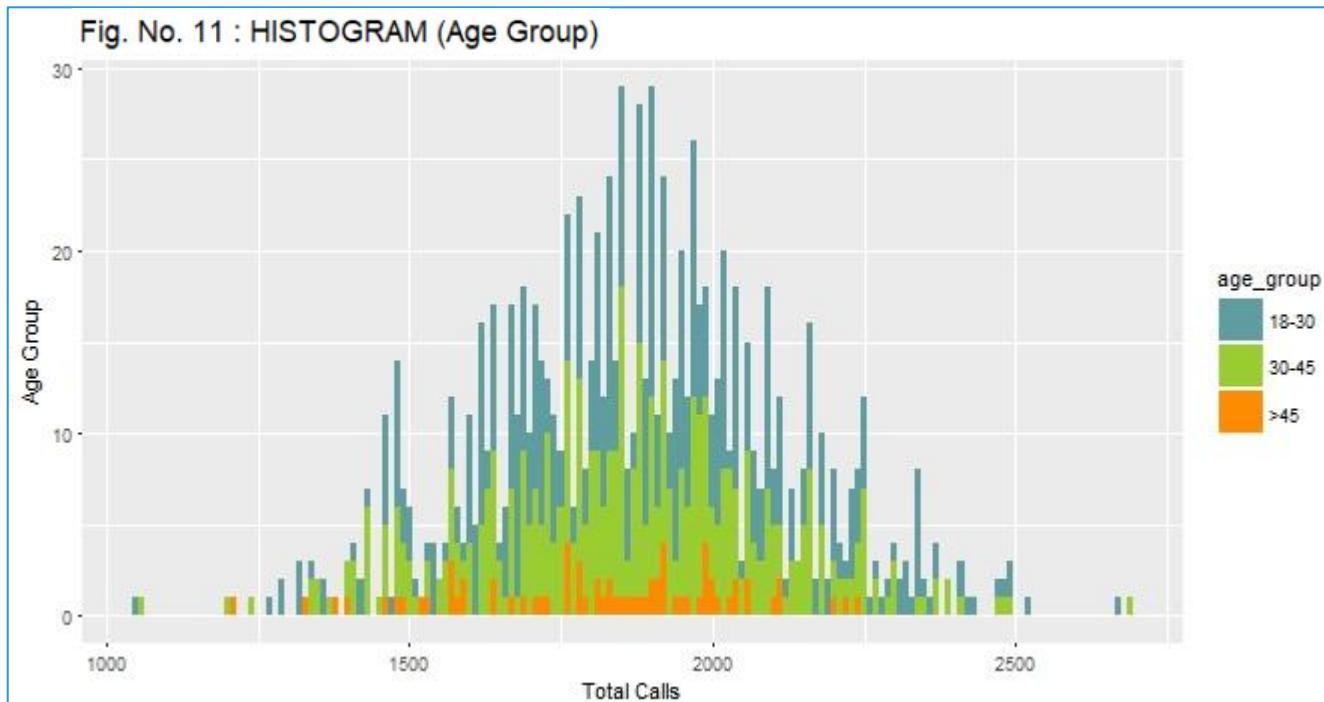
```
# Histogram with Age Group
```

```
ggplot(telecom, aes(x=Calls))+ geom_histogram(aes(fill=Age_Group),  
binwidth=10)+ labs(x="Total Calls", y="Age Group", title="Fig. No. 11 :  
HISTOGRAM (Age Group)", colour="Age Group") +  
scale_fill_manual(values=c("cadetblue", "yellowgreen", "darkorange"))
```

- ❑ **aes()** function in **geom\_histogram()** gives multiple bar one over the other using **fill= Age\_Group**

# Histogram in R

```
# Output
```



# Data Snapshot

JOB PROFICIENCY DATA  
Variables

Columns	Description	Type	Measurement	Possible values
empno	Employee No	Numeric	-	-
aptitude	Aptitude	Numeric	-	positive values
testofen	Test of English	Numeric	-	positive values
tech_	Technical Score	Numeric	-	positive values
g_k_	General Knowledge	Numeric	-	positive values
job_prof	Job Proficiency	Numeric	-	positive values

Observations

## Case Study - 2

To get a better understanding of the subject, we shall consider the below case as an example.

### Background

A company has the scores of various attribute tests of their employees

### Objective

To study the correlation between Aptitude and Job Proficiency.

### Sample Size

25

# ScatterPlot with Regression Line in R

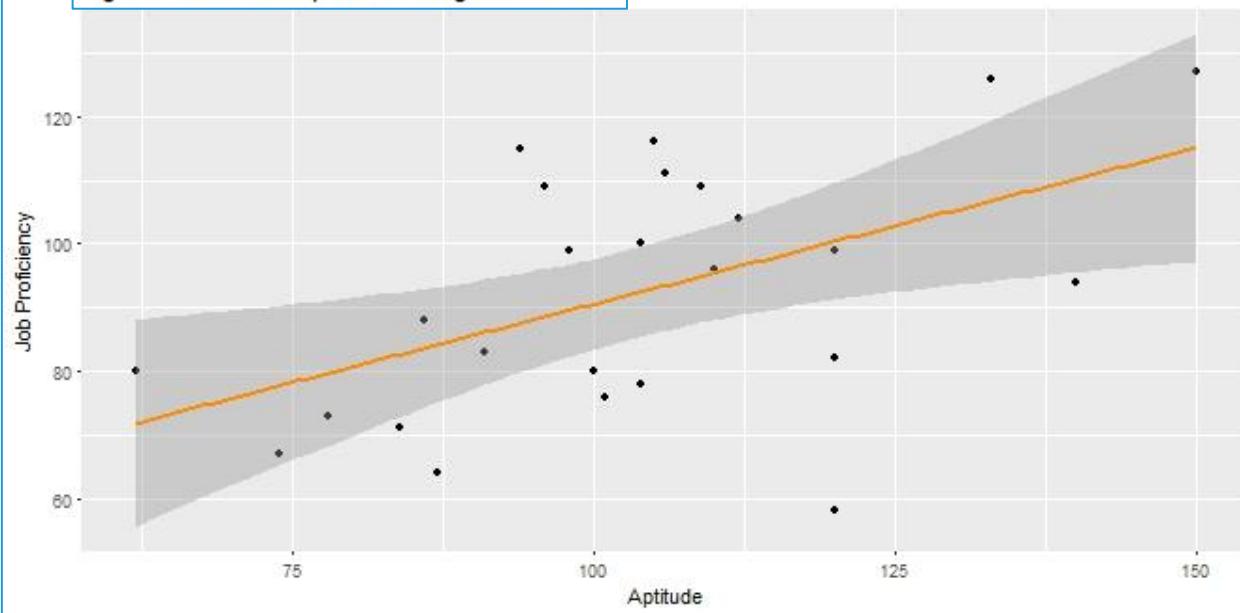
```
# Importing Data  
job<-read.csv("JOB PROFICIENCY DATA.csv", header=TRUE)  
  
# Scatterplot with Regression Line  
ggplot(job, aes(x=aptitude, y=job_prof))+  
  geom_point()+geom_smooth(method="lm", col="darkorange")  
  labs(x="Aptitude", y="Job Proficiency", title="Fig. No. 13 :Scatterplot  
with Regression Line")
```

- **geom\_point ()** is used to plot the data points, in this case it's a scatter plot
- **geom\_smooth ()** is used to plot the curve
- **method="lm"** is used to get a linear regression line

# ScatterPlot with Regression Line in R

```
# Output
```

Fig. No. 13 :Scatterplot with Regression Line



# Data Snapshot

Plotting a trendline requires time-element.

Consider the following two datasets. Week can be taken as the time element.

Variables				
<b>CustID</b>	<b>Age</b>	<b>Gender</b>	<b>PinCode</b>	<b>Active</b>
1001	29	F	186904	Yes
Columns	Description	Type	Measure ment	Possible values
<u>CustID</u>	Customer ID	Numeric	-	-
Age	Age	Numeric	-	18-51
Gender	Gender	Categorical	-	M,F
<u>PinCode</u>	Area's PinCode	Numeric	-	-
Active	usage of telecom	Categorical	-	Y,N

Variables				
<b>CustID</b>	<b>Week</b>	<b>Calls</b>	<b>Minutes</b>	<b>Amt</b>
1001	1	56	392	78.4
Columns	Description	Type	Measure ment	Possible values
<u>CustID</u>	Customer ID	Numeric	-	-
Week	Week no.	Numeric	-	1-24
Calls	No. of Calls	Numeric	-	positive values
Minutes	Total Minutes	Numeric	Minutes	positive values
<u>Amt</u>	Amount Charged	Numeric	Rs.	positive values

# Trend Line in R

```
# Importing Data
demographic<-read.csv("TelecomData_CustDemo.csv", header=TRUE)
transaction<-read.csv("TelecomData_WeeklyData.csv", header=TRUE)

# Merging and Formatting Data
# Creating new variable Age_Group & aggregating

working<-merge(demographic, transaction, by=("CustID"), all=TRUE)
working$Age_Group<-cut(working$Age, breaks= c(0,30,45,Inf), labels= c("18-
30","30-45",">45"))
trend<-aggregate(Calls~Week+Age_Group, data=working, FUN=sum)

# Observing Age_group wise Trend
ggplot(trend, aes(x=Week, y=Calls, colour=Age_Group))+
  geom_line(size=1)+ geom_point(size=3)+labs(y="Calls", title="Fig. No. 14 :
  TREND LINE")
```

- ❑ **geom\_line()** is used to call the trend line
- ❑ **geom\_point()** is used to plot the data points

# Trend Line in R

```
# Output
```



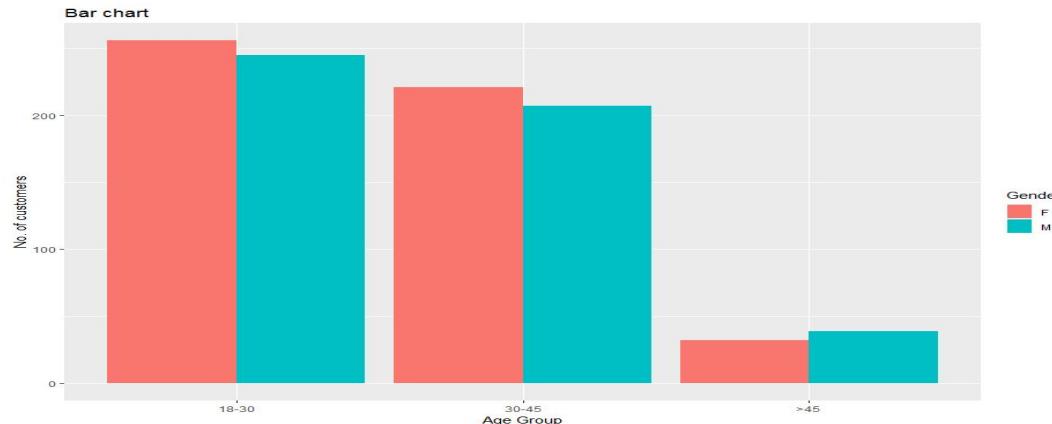
# Get an Edge!

Multiple Bar Chart in R (CAUTION)

```
ggplot(telecom, aes(x=Age_Group,fill=Gender))+geom_bar(position="dodge",  
fill="darkorange")+labs(x="Age Group", y="No. of customers", title="Bar  
chart")
```

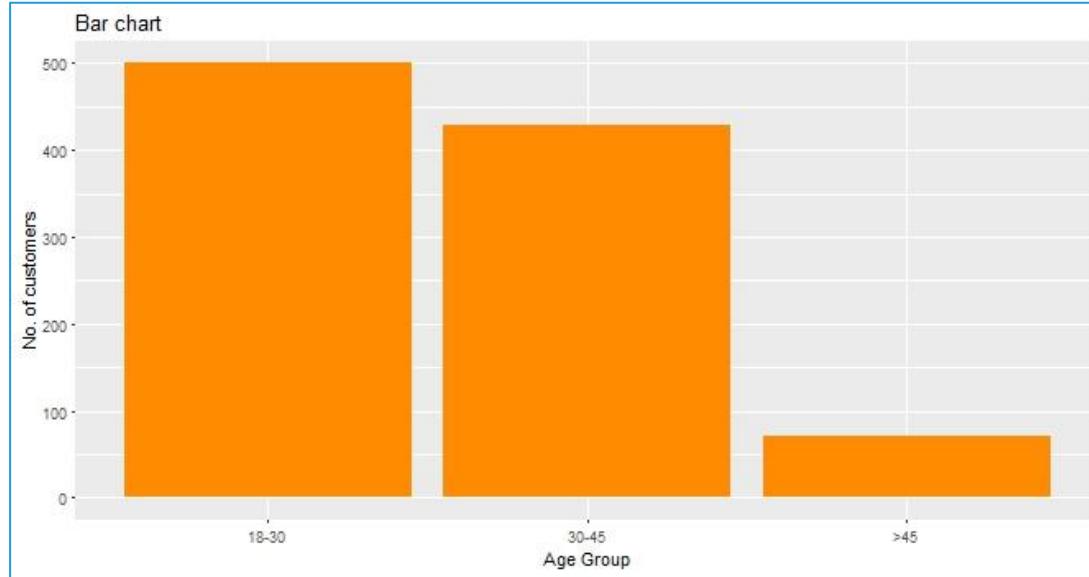
Caution: fill="darkorange" in geom\_bar() overrides the fill=Gender in ggplot()

So instead of getting this output :



# Get an Edge!

You get this output only because of fill = "darkorange" argument in geom\_bar() function.



# Quick Recap

Using ggplot  
package in R

- i. Bar Charts
- ii. Pie Chart
- iii. Box-Whisker Plot
- iv. Histogram
- v. Scatterplot with Regression Line
- vi. Trend Line