

# Getting started with Python-

## Installing Python

# Contents

1. Installing Python on Windows, Mac, Linux OS.
2. Using the Conda Distribution
3. Installing Anaconda/Miniconda on Windows, OS X and Linux operating system
4. Using Jupyter as an IDE
5. Using Spyder as an IDE
6. The Spyder Interface
7. Using Google Colab as an IDE

# Installing Python on Windows, Mac and Linux OS

- Python is a high level, interpreted language that usually comes pre-installed on all computers.
- Because Python is widely used for various purposes, it has a lot of packages available which are all compatible with different versions of the language. This can be painful for a beginner to manage, right at the introduction to his or her journey.
- The easiest way of installing Python for data science is to use the Data Science Package Manager, Anaconda. It is the simplest way to install Python with all the libraries you need, it manages all package compatibility issues and doesn't interfere with the required system libraries.
- If you still wish to download python separately, you can go to <http://www.python.org> and select the version that is suitable for your Operating system and hardware. Be careful to select the latest stable version.

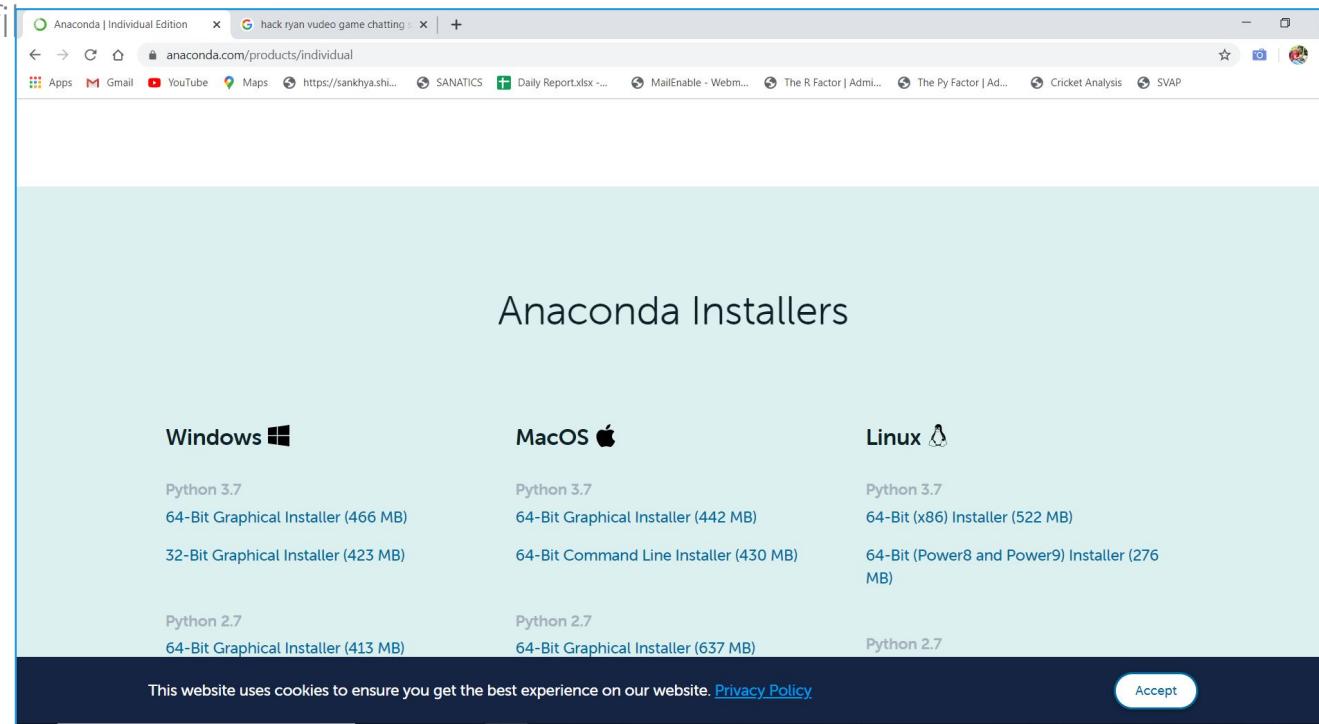
# Anaconda & the Conda Distribution

- The Conda distribution is an open data science platform powered by Python. It consists of two package managers – Anaconda & Miniconda.
- Anaconda package list includes:
  - easy installation of Python(2.7.12, 3.4.5, 3.5.2 and 3.6)
  - updates of over 100 pre-built and tested scientific and analytic Python packages
  - over 620 more packages available via a simple command:  
`“conda install <packagename>”.`
  - open source development environments such as Jupyter/IPython and Spyder and is supported by Sublime Text 2 and PyCharm.
- However, Anaconda takes up a lot of disk space which is why Miniconda comes in as an alternative – it includes only conda and Python, where all the other packages can be installed separately as per your needs.

# Installing Anaconda (Simulation)

Steps (for all OS):

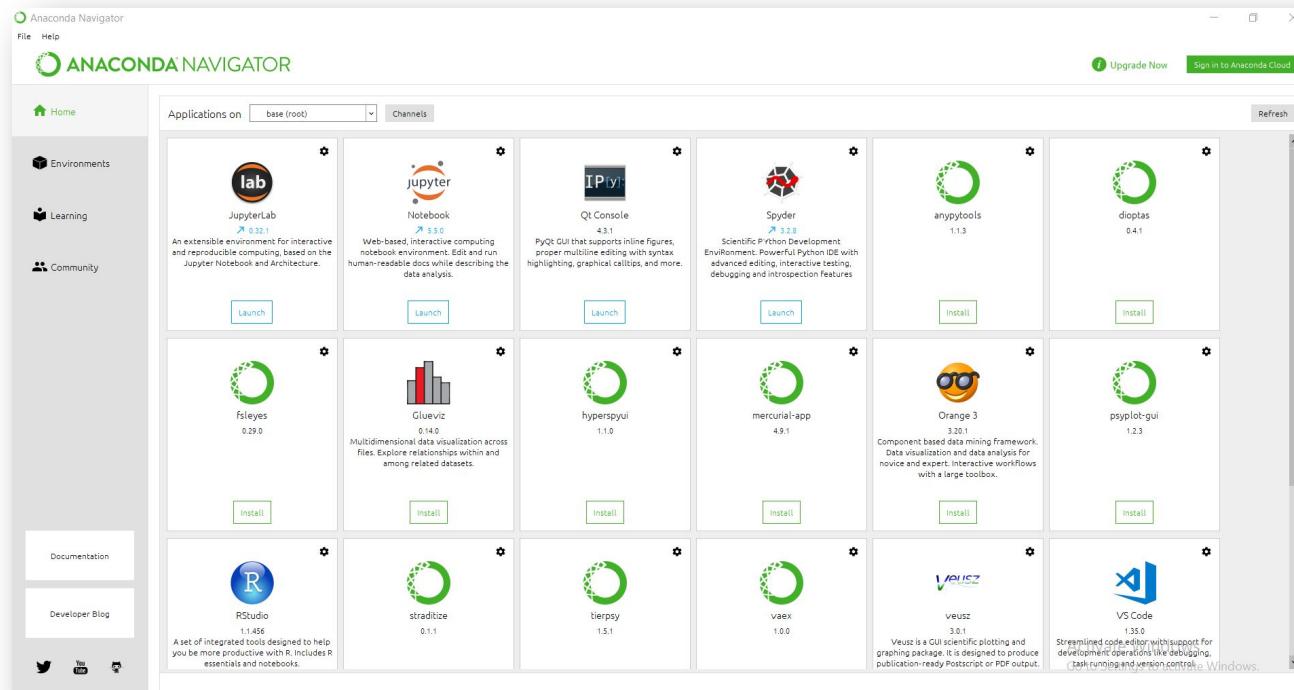
- Go to <https://www.anaconda.com/products/individual> to download the installer. Click on the installer link according to your system. It will take a while to download the .exe file.



Note : The interface of this site might change, but installer can be downloaded from this link.

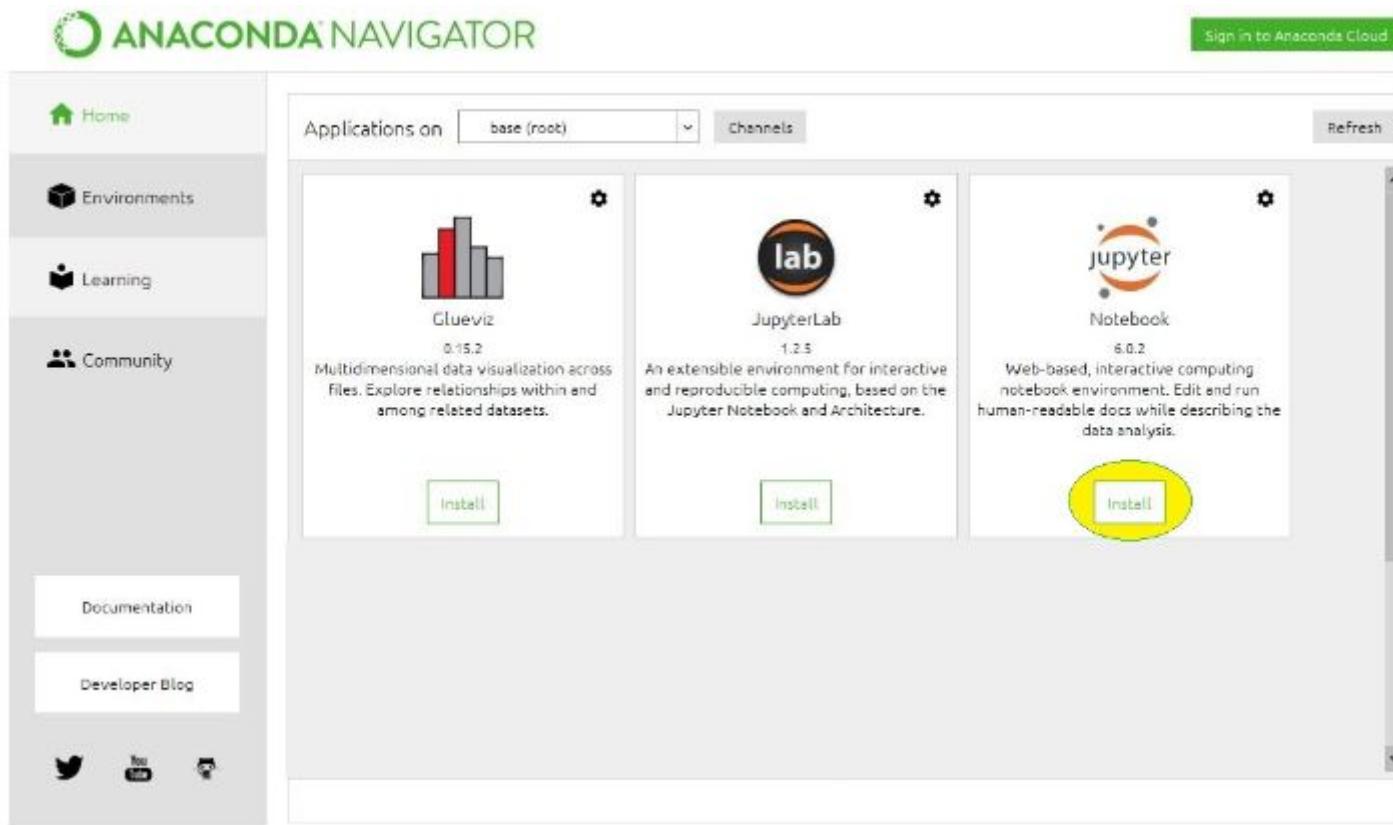
# Installing Anaconda (Simulation)

- After completing the installation process, launch Anaconda Navigator. The screen will appear as shown below.
- It provides Various python IDE(s) like Jupyter, Qt Console, Spyder etc. You can install the necessary Python IDE suitable for your application and system.



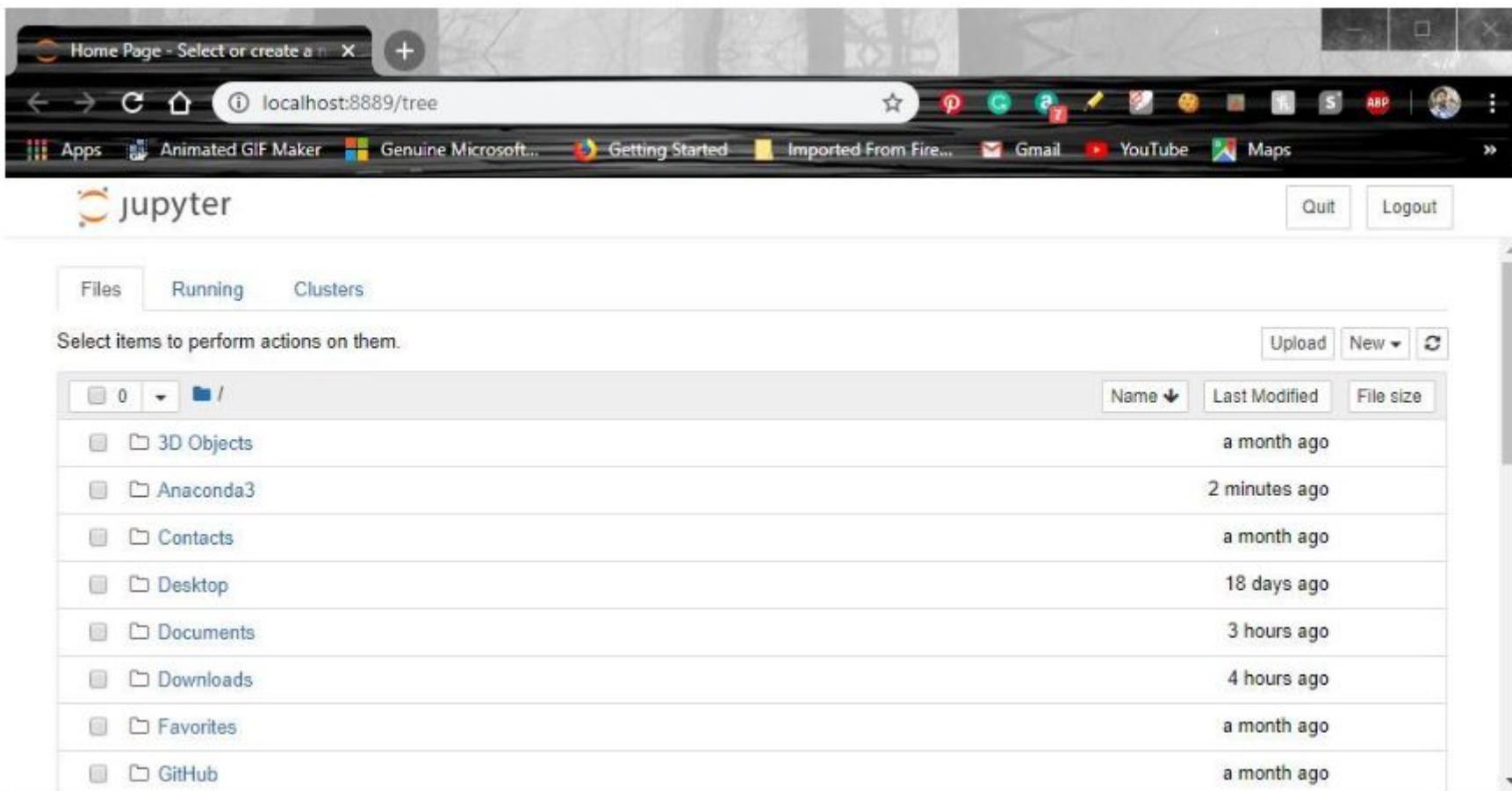
# Using Jupyter as an IDE

- To install Jupyter using Anaconda:
- Click on the Install Jupyter Notebook Button:



# Jupyter – User Interface (Simulation)

- Begin the Installation
- Load Packages
- Finish Installation
- Launch Jupyter



# Using Jupyter as an IDE

## Installing Jupyter Notebook using pip:

- PIP is a package management system used to install and manage software packages/libraries written in Python.
- These files are stored in a large “on-line repository” termed as Python Package Index (PyPI).
- pip uses PyPI as the default source for packages and their dependencies.
- To install Jupyter using pip, we need to first check if pip is updated in our system. Use the following command to update pip:

```
python -m pip install --upgrade pip
```

- After updating the pip version, follow the instructions provided below to install Jupyter:
- Command to install Jupyter

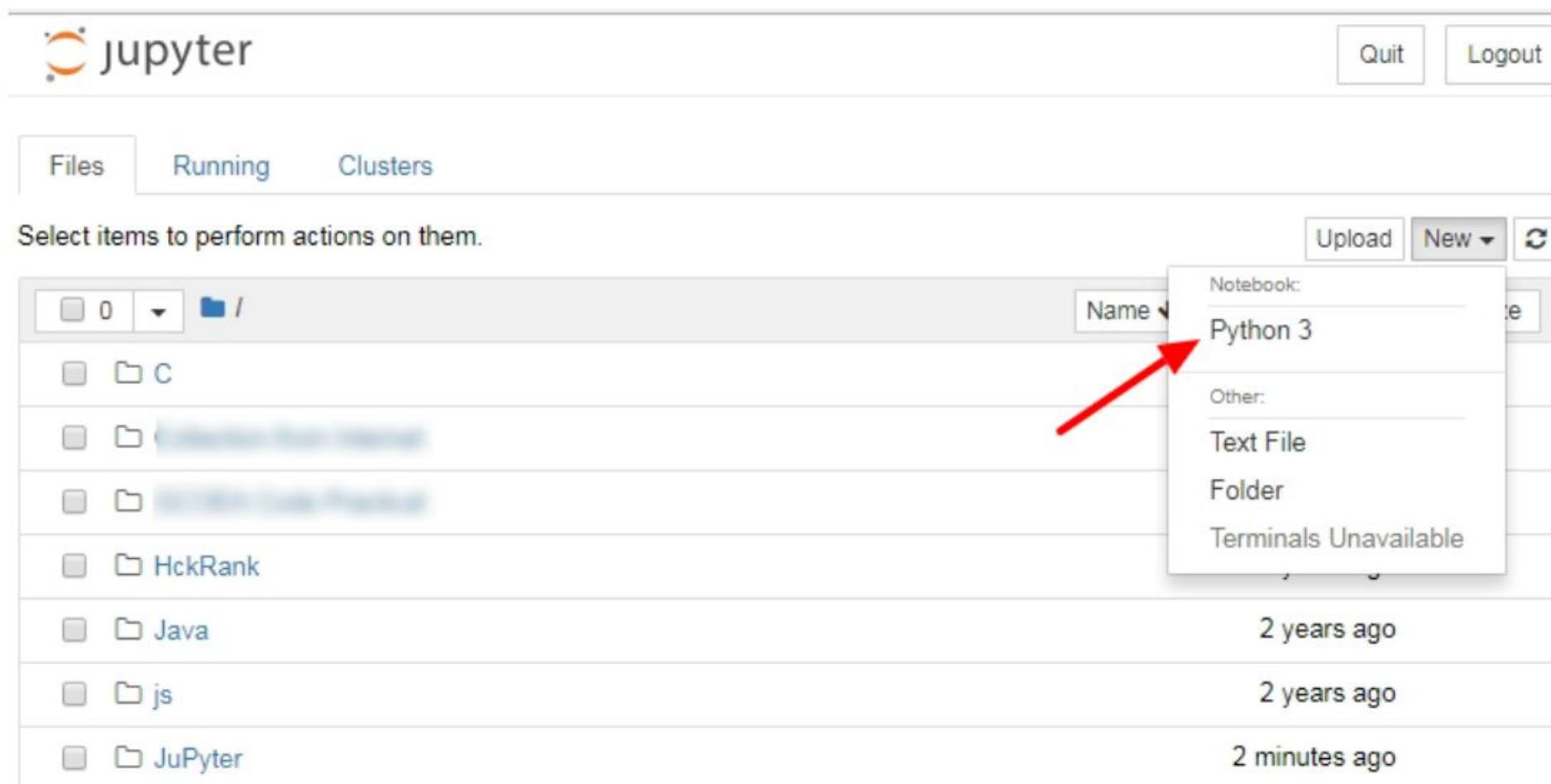
```
python -m pip install jupyter
```

- After installation is complete, use the following command to launch jupyter:

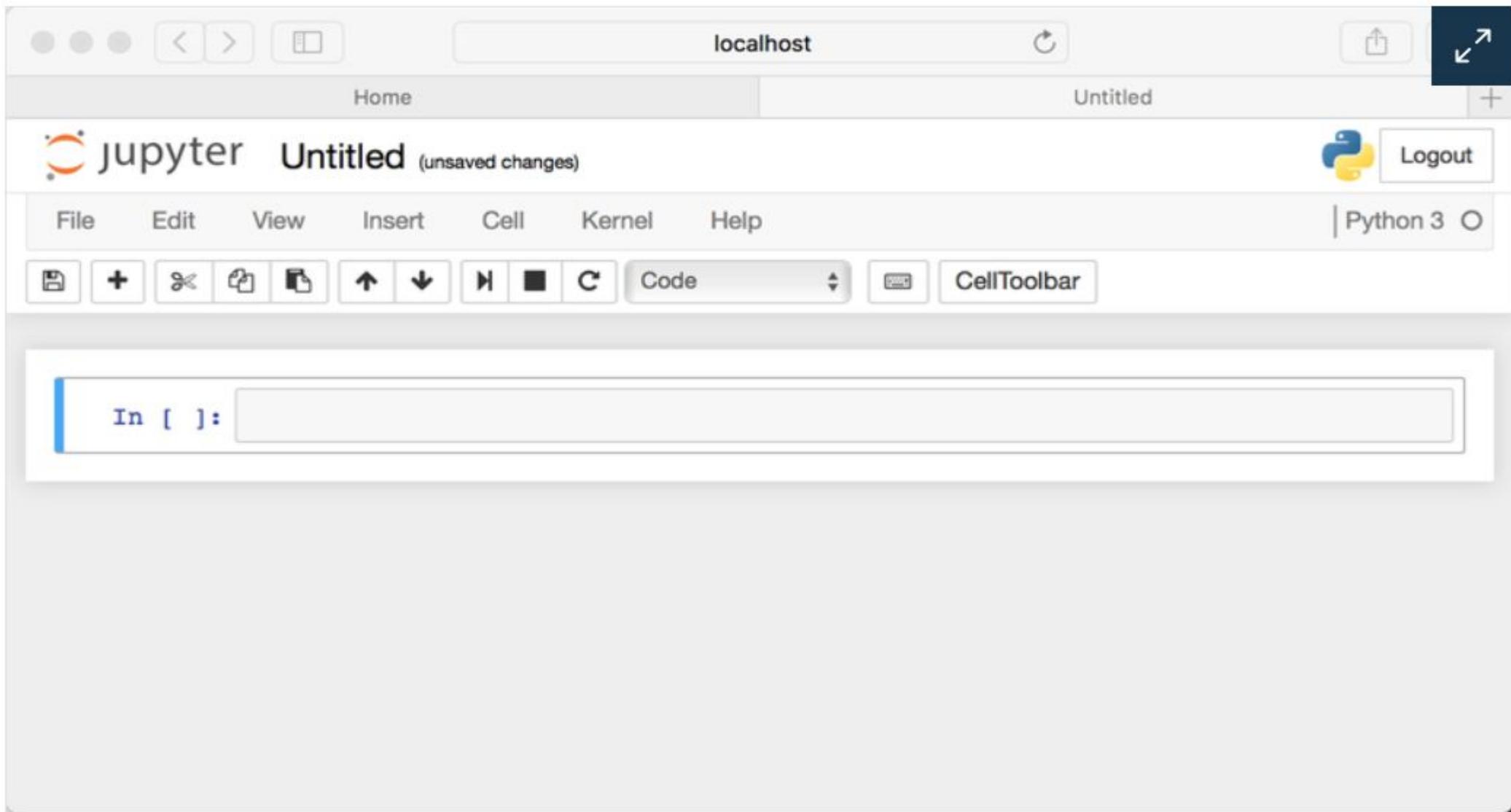
```
jupyter notebook
```

# Jupyter – User Interface (Simulation)

Click on the New button (upper right), and it will open up a list of choices, choose Python 3.



# Jupyter – User Interface (Simulation)

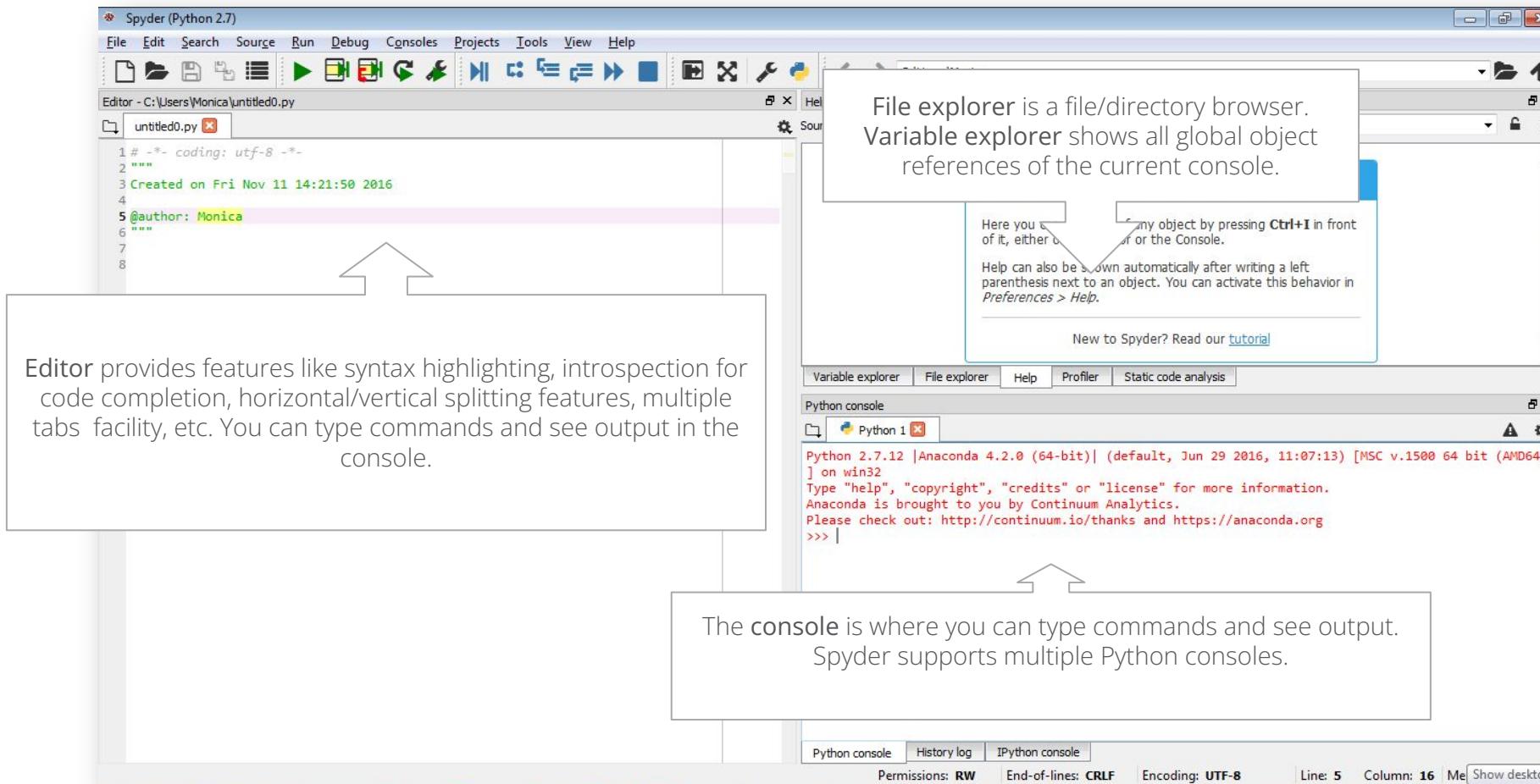


# Using Spyder as an IDE

- Spyder is an interactive development environment for Python, including an editor and comes pre-installed with Anaconda.
- The name ‘Spyder’ is derived from “Scientific Python Development Environment”
- One of its best features is its integration with Ipython – a command shell for interactive programming that is popular amongst Data-Scientists and Software Engineers alike as it improves productivity as it promotes an execute-explore workflow instead of an edit-compile-run workflow. Making it great for data exploration.
- R and MatLab users will be very comfortable using Spyder as its layout is very similar to the GUI available in these languages.

# Spyder – User Interface (Simulation)

After Anaconda is installed, find 'Spyder' in your programs menu. It loads all the python libraries, so it may take a while to start up.



# Using Google Colab as an IDE

Google Colab or “the Colaboratory” is a free cloud service hosted by Google to encourage Machine Learning and Artificial Intelligence research, where often the barrier to learning and success is the requirement of tremendous computational power.

## Benefits of Colab

- Python 2.7 and Python 3.6 support
- Free GPU acceleration
- Pre-installed libraries: All major Python libraries like TensorFlow, Scikit-learn, Matplotlib among many others are pre-installed and ready to be imported.
- Built on top of Jupyter Notebook
- Collaboration feature (works with a team just like Google Docs): Google Colab allows developers to use and share Jupyter notebook among each other without having to download, install, or run anything other than a browser.
- Supports bash commands
- Google Colab notebooks are stored on the drive

# Using Google Colab as an IDE

## Getting Started

1. To start working with Colab you first need to log in to your google account,
2. Then go to this link <https://colab.research.google.com>.

The screenshot shows the Google Colab interface. At the top, there's a navigation bar with the 'Welcome To Colaboratory' logo, file menu (File, Edit, View, Insert, Runtime, Tools, Help), a 'Share' button, and a settings gear icon. Below the navigation bar is a toolbar with 'Table of contents', a 'Copy to Drive' button, and a 'Connect' dropdown. On the left, a sidebar displays a 'Table of contents' with sections like 'Getting started', 'Data science', 'Machine learning', 'More Resources', 'Featured examples', and a 'Section' button. The main content area features a large heading 'Welcome to Colab!' and a descriptive text: 'If you're already familiar with Colab, check out this video to learn about interactive tables, the executed c history view, and the command palette.' Below the text is a video thumbnail titled '3 Cool Google Colab Features' featuring a man in a hoodie.

# Getting started with Python-

## Introduction to Python

# Contents

1. What is Python?
2. Why is Python used for Data Science?
3. Companies Using Python
4. Python's Community
5. Success Stories
6. Python Environment
7. IDE

# What is Python?

- Python is a general purpose, high level and powerful dynamic programming language.
- Python is an Object Oriented Programming language i.e. it is based on classes and objects where each object can communicate with other by exchanging messages and processing data.
- Its design philosophy is based on the importance of readability and simplicity
  - clean and easy to learn syntax
- It is Free and Open Source which means that it is free for download and the source code is freely available and open for modification and reuse.
- It compiles and runs on a wide variety of UNIX , Windows, MacOS, and various other platforms.

# Why Learn Python ?

Free and  
Open  
Source

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

Ease  
of  
Use

Python is easy to pickup and has easy readability.

Extensive  
support  
libraries

Large standard libraries include areas such as internet, string operations, operating system interface, web services.

Extensible

Python is portable and extensible. It allows to perform cross-language operations.

# Why Learn Python ?

Flexible

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

Embeddable

Python allows to add scripting capabilities to the code in other languages.

Great Data visualization

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

Speedy

It offers fast and effective support to the developers and programmers, it brings down the development time and cost.

Machine Learning

Easy to understand and use machine learning libraries.

# History of Python

1980

- Designed by Guido Van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands.

1990

- Derived from the television series Monty Python's Flying Circus.

1991

- First release of Python with version 0.9.0.

1994

- Python version 1.0 is released

2000

- Python version 2.0 is released

2008

- Python version 3.0 is released

2020

- Python version 3.8.3 is released

# Why Python is Used for Data Science?

- Python and R are among the most popular programming languages for Data Science and are often found opposite to each other in the room of debate on the fact that which one is more valuable. However, both the languages have their specialised features which makes them unique.

But what is making Python very popular

- Syntax simplicity: its simple and elegant syntax is easy to get along yet being so much powerful at the same time.
  - Operability on different environments: it is easy for programmers to write maintainable, large scale robust code
- Libraries: Python provides data scientists with a set of powerful and efficient libraries like **Scikit**, **Numpy**, **Pandas**, **Scipy** that assist data analysis and statistical computing.

# Companies Using Python

- There are around 4.3 million Python users.
- Python is widely used by:
  - Academics and Researchers.
  - Financial companies
  - Large tech companies
  - Social Media giants like Facebook and Instagram
  - Google, Netflix, Dropbox, Reddit, Quora

# Python's Community

- Python has an enthusiastic user base, dedicated to encouraging use of the language, and committed to being diverse and friendly.
- The Python Software Foundation (PSF) is a non profit corporation which promotes, protects, and advances the Python programming language and supports and facilitates the growth of a diverse and international community of Python programmers.
- The PSF manages the open source licensing of Python2.1 and later and own and protect the trademarks associated with Python.
- The PSF runs a North American Pycon Conference annually and supports other Python Conferences around the world.
- PyCon is the largest annual gathering of developers, application designers and business people for the community that uses and develops the open-source Python programming language.

Source: [python.org](http://python.org)

# Success Stories

- Python has been successfully implemented and is part of the winning formula for productivity, software quality, and maintainability at many companies and institutions around the world.
- Survey Monkey speeded up with Python:  
Survey monkey is the world's largest survey company, migrated from monolithic C# and .NET to Python as they were finding it slow to add, test and deploy new features. Python has sped up the process of adding new features and handled heavy traffic without problems and solved many more issues. This transition has been a great success.
- OpenERP is a full-featured enterprise resource planning suite written in Python. Python has played a strategic role in development of OpenERP, giving the company the flexibility to be able to adapt to market or design decision changes over the years.

# Success Stories

- ForecastWatch a service of Intellovations, is in the business of rating the accuracy of weather reports from companies such as Accuweather, MyForecast.com, and The Weather Channel. Python has provided high level functionality with very few lines of code, and its interactivity has made their testing code easy and made trying out new ideas or features painless and quick
- Likewise, there are many companies across industries from Software development to Arts to Business to Education to Government to Science and Engineering, where Python has given them a different shape altogether leading it to a big success.

# Python Environment

- Python has a rich and powerful external library support which comprises of external modules and packages. It has large set of extremely useful built-in functions.
- Automatic memory management system which uses a mix of reference counting and a cycle detecting garbage collector.
  - Now, how does reference counting works??

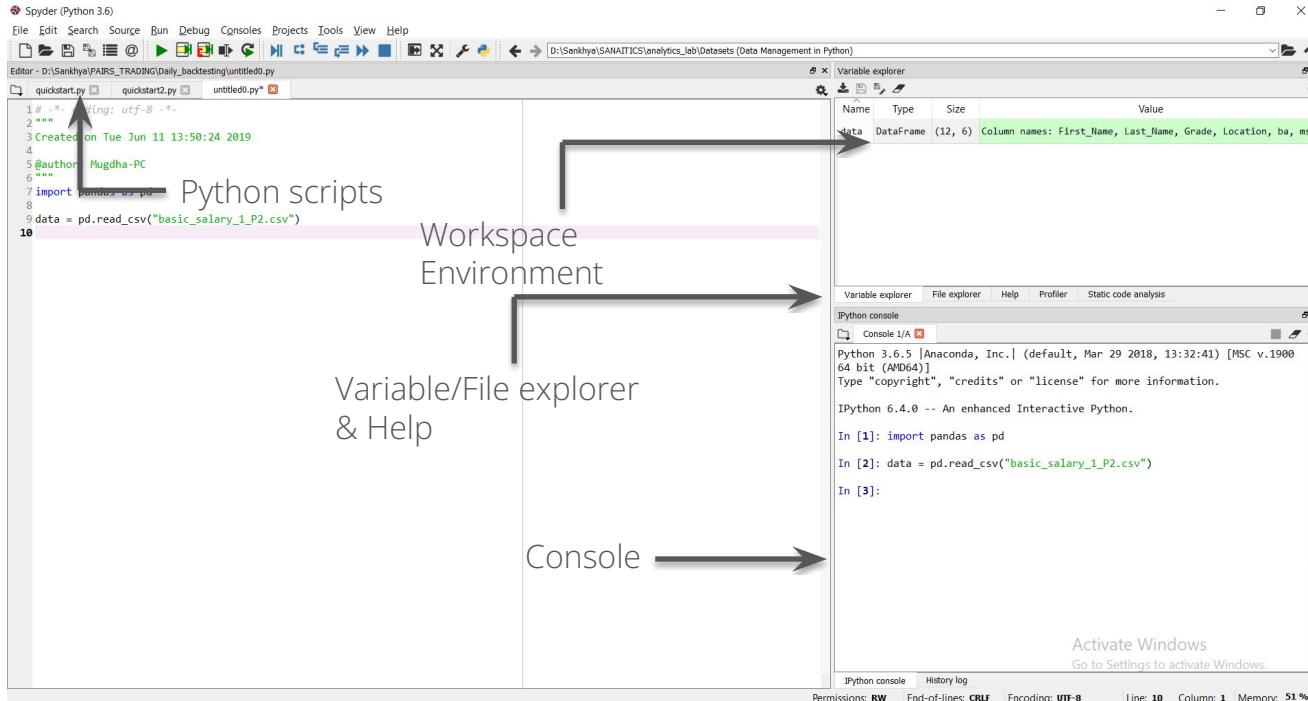
Every Python object has a reference counter. An extra counter is kept along with each object that is created. Anytime a reference is copied counter is incremented, and anytime a reference goes out of scope, or is reset, counter is decremented.
  - What does the Cycle Detection Garbage collector do?

It detects the objects which are no longer referenced and recycles the unused memory by freeing up the memory from those objects.
- Python commands are case sensitive.

# IDE

- Integrated Development Environment(IDE) is a software which provides programmers with an interface combined with all the tools at hand
- Selection of the right IDE influences the productivity and effectiveness of Python programming
- There are many IDE's for Python (for data science) such as Spyder (Scientific PYthon Development EnviRonment) and Rodeo.
- Most recommended and widely used among these is Spyder.
- Spyder includes a Console, Editor, History log, help, variable Explorer, File Explorer and Projects

# Spyder IDE



# Quick Recap

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

Python

- Python is an Object Oriented Programming language
- It is free and open source software
- It is built using packages, which contains Basic and advanced functions.
- Python Community is a global community with millions of software developers who interact online and offline in thousands of virtual and physical locations.

Spyder

- Spyder is most widely used user friendly IDE.

# Getting started with Python- Writing Your First Program in Python

# Contents

1. Opening Spyder
2. Create and Save Your File
3. Writing Your First Program in Python
4. Help in Python

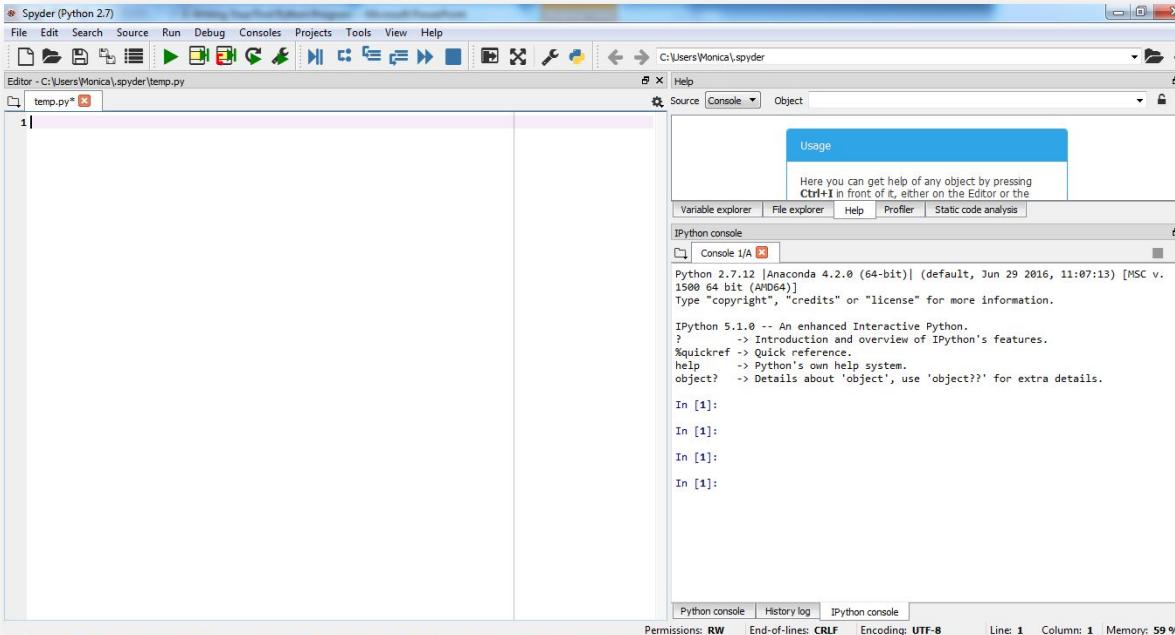
# Opening Spyder

In our last session we had installed Spyder and seen its features. Now we will program.

## Open Spyder

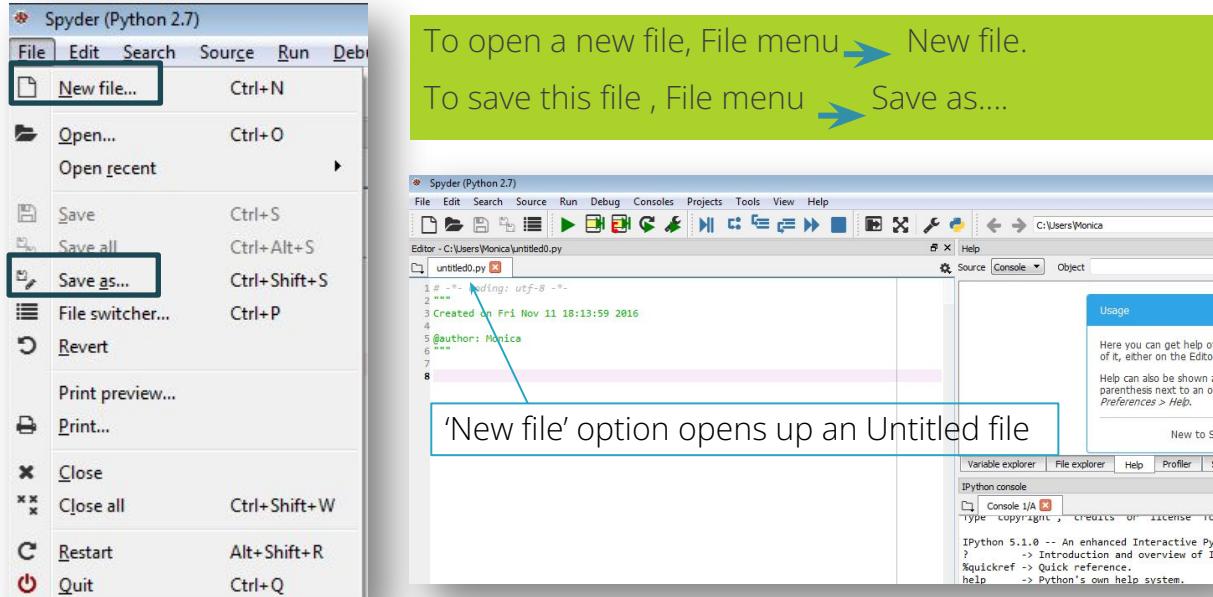
First go to Start menu, find Spyder, open it. We will see an environment like this.

Alternatively, you can open Spyder by typing spyder in your command shell.



# Create and Save Your File

- By default it opens a temp.py file. We can also open a new file, write commands, execute them. Save the file at a desired location. It helps to save the script for later use.



# Writing your First Program in Python

Creation of object (an object can store values and further can be used in multiple tasks):

```
# Create an object x and assign a value 20
x=20
x
20
```

- The '=' operator is used to assign the values to an object.
- Execute or run the command using the enter key in python console and F9 key in Editor. You can run a single command, bunch of commands by highlighting those, or the entire file at once.
- Comments are used to describe the code and can be added by preceding all comments with the # symbol.

# Writing your First Program in Python

Create another object and perform some mathematical calculations using both the objects

```
# Create an object y and assign a value 15  
y=15  
y  
15
```

```
# Add x and y  
x+y  
35
```

```
# Multiply x with y  
x*y  
300
```

- '+', and '-' are Arithmetic operators. There are more such operators for which we have a dedicated session at a later stage.

# Writing your First Program in Python

How to print a message?

```
print ("Welcome to Ask Analytics")  
Welcome to Ask Analytics
```

- `print()` is a python command to print a text message which must be enclosed in quotes.

Print numbers:

```
print (55)  
55  
print (4*5)  
20
```

# Writing your First Program in Python

We can use print command to join multiple words, strings(sequence of characters), numbers.

- To join multiple items to form a single sentence, separate the items to be printed with comma ( , ) operator.

```
print ("This is", "session no.", 3)
```

```
This is session no. 3
```

- Different operators and formatting options that can be used with print command are:

,	To print multiple strings in a line
+	To concatenate two strings into a single string
%	To concatenate strings and integers

# Writing your First Program in Python

There is more we can do with %.

Use %s to print string, %d to print integers, %f to print floating point numbers. For example:

```
x=14  
print ("He is %d years old" %x)  
He is 14 years old
```

# Help in Python

- Python has an interactive built-in help system consisting of a documentation for modules, objects or methods.
- To access help in Python, help() function is used.
  - Write help() with no argument in brackets and execute it, the interactive help system starts on the console. Type the name of module, function, class, method, or keyword in the console and get the built-in documentation printed on the console. To exit, type “quit”.
  - Another way of getting information is by writing:  
help(x) in Editor, execute it and the documentation is displayed on the console. 'x' can be any kind of object, module, function, package or keyword.

# Quick Recap

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

## Write a program

- To assign values to an object: Use operator '='.
- Press Enter & see the value of the object in console
- Add comments for reference using `#` symbol

## Help in Python

- If you know the topic but not the exact function:  
`help()` the interactive help system starts on the console.
- If you know the exact function:  
`help(functionname)`

# Getting started with Python- Using Indexing With Data Objects

# Objects and Zero Based Indexing

Python has 5 standard data types

# Python Data Types

## Numbers

Numeric values:

- integers (positive or negative whole numbers) : -2, 44, 4645, -245
- floating point numbers : 5.0, 22.2, 1.4556, -55555.675

## Strings

Sequence of characters represented in either pairs of single or double quotes.

## Lists

Sequence of elements which do not need to be of same data type. Elements and size can be updated.

## Tuples

Sequence data type similar to the list. Unlike lists, tuples are enclosed within parentheses ( () ) and cannot be updated.

## Dictionary

Unordered collection of key-value pairs. Dictionaries are defined within curly braces ( { } ). Each KEY must be unique, but the VALUE may be the same for two or more keys.

\*

Note : Use type() function to know the data type of the object.

# Numbers

Both integers and Floating point numbers are supported by Python and are defined as **int** and **float**.

```
x=10
x
10
type(x)
<type 'int'>
y= 15.20
type(y)
<type 'float'>
z= -32.54e100
type(z)
<type 'float'>
```

# Numbers

- To convert `float` to `int`, you need to use `int()` function.

```
x=int(-99999.675)
```

```
x
```

```
-99999
```

- To c

```
type(x)
```

```
<type 'int'>
```

```
y=float(25)
```

```
y
```

```
25.0
```

```
type(y)
```

```
<type 'float'>
```

# Strings

- Python strings are sequence (left-to-right order) of characters enclosed in single or double quotes.
- Python strings are immutable i.e. they cannot be modified but we can run expressions to make new string objects.

```
x='welcome to the Python world'  
x  
'welcome to the Python world'  
z='3948'  
z  
'3948'
```

```
type(z)
```

```
<class 'str'>  
x = x + ", John"  
x
```

```
'welcome to the Python wor
```

String outputs are displayed in quotes

Using ‘+’ operator you can concatenate two or more strings to make a new string object

# Strings

Accessing elements of the String:

- Use [ ] brackets with expression as index or indices.

```
x[0]
```

```
'w'
```

```
x[2:5]
```

```
'lco'
```

access multiple elements by giving a range of their index

- To check the length of the string, use `len()` function.

```
# length of the string object 'x'
```

```
len(x)
```

```
33
```

- Python methods for handling and processing strings are covered later.

# Strings

- To convert a value to a string type, you need to use str() function.

```
a=1567  
a  
1567  
y=str(a) ←  
type(y)  
<type 'str'>
```

Any value can be converted into string with the help of **str()** function.

# Lists

- List can store elements of different data types.
- List elements are separated by commas (,) and are enclosed within square brackets ( [ ] ).
- Lists are mutable unlike strings.

```
list1=['python', 1998, 'list', 12]
list1
['python', 1998, 'list', 12] ←
len(list1)
4 ←
list1 has elements of
number and string type
len() length of the list1
list2=[2001,2005,2010,2016]
list2
[2001, 2005, 2010, 2016]
list3=["red","blue", "white", "black"]
list3
['red', 'blue', 'white', 'black']
```

# Lists

Accessing elements of the List:

- Use [ ] brackets with expression as index or indices.

```
list1[0]  
'python'  
list2[1:4]  
[2005, 2010, 2016]
```

Updating Lists:

```
# Change the value of 3rd element of list2 to '2006'
```

```
list2[2]=2006
```

```
list2
```

```
[2001, 2005, 2006, 2016]
```

To change the value, you need to specify the index no. of the element enclosed in **[] brackets**.

```
# Add an element '2012' to list2
```

```
list2.append(2012)
```

```
list2
```

```
[2001, 2005, 2006, 2016, 2012]
```

**append()** function modifies the original list by adding a single element to the end of the list.

# Lists

Deleting List elements:

- To remove a list element if you know exactly which element(s) to be deleted, use **del** operator.

```
# Delete 2nd element of list2
del list2[1]
list2
[2001, 2006, 2016, 2012]
```

In case you do not know which element to delete use **remove()** method

```
# Delete '2012' from list2
list2.remove(2012)
list2
[2001, 2006, 2016]
```

# Lists

Some other common List functions:

```
# Insert new element at the given index  
list2.insert(1,2019)  
list2
```

```
[2001, 2019, 2006, 2016]
```

```
# Search index of an element from the list  
list2.index(2016)
```

```
3
```

```
# Sort the list in ascending order.  
list2.sort()  
list2
```

```
[2001, 2006, 2016, 2019]
```

```
# Reverse the list in place  
list2.reverse()  
list2
```

```
[2019, 2016, 2006, 2001]
```

insert() can insert an item at given position. Here 1 is the position 2019 is the item

For descending order specify reverse=True

# Tuples

- Lists are mutable and tuples are immutable.
- The main difference between mutable and immutable is memory
- Tuples are useful when you know that you are not frequently adding new elements.
- Tuples are heterogeneous data structures

# Tuples

Creating a Tuple:

```
# Create a tuple  
tuple1=('math','physics', 'chemistry')  
tuple1  
('math', 'physics', 'chemistry')
```

Accessing elements of the Tuple:

```
tuple1[0]
```

```
'math'
```

```
tuple1[0]="mathematics"
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support
```

Use [] brackets with expression as index or indices.

An exception is raised because modifying tuple is not allowed.

# Tuples

Updating Tuples:

- Tuples are immutable that means their elements cannot be changed. However, you can take portions of existing tuples to create new tuples.

```
tup1=('one', 'two', 'three')
tup2=(11,22,44.5)
tup3=tup1+tup2
      ←
tup3
('one', 'two', 'three', 11, 22, 44.5)
```

New tuple  
‘tup3’ is  
created  
containing  
elements of  
‘tup2’ and  
‘tup3’.

# Tuples

Deleting Tuple elements:

- Individual elements cannot be removed from a tuple. To explicitly remove an entire tuple, use `del` command.

```
del tuple1  
tuple1
```

```
Traceback (most recent call last):
```

```
  File "<ipython-input-7-97b79749fdf6>", line 1, in <module>  
    tuple1
```

```
NameError: name 'tuple1' is not defined
```

An exception is raised because tuple1 doesn't exists any more.

# Dictionary

- A dict (dictionary) type object can store a collections of elements just like a list, but the elements are in the form of key-value pairs and to retrieve the values, you can use keys. This way a dict can be treated like a database for storing and organising data
- Key-Value pairs are separated by a colon (:) and pairs themselves by commas (,) and all this is defined in a pair of curly braces ({}).
- The key-value pairs in dictionary objects are not ordered in any manner.
- Dictionary keys are Case Sensitive.

# Dictionary

Creating a Dictionary:

```
# Create a Dictionary  
Dict1={'Name': 'Ruchi', 'Age': '18', 'Class': 'Twelfth'}  
Dict1  
{'Name': 'Ruchi', 'Age': '18', 'Class': 'Twelfth'}
```

Accessing values of the Dictionary:

- Use [ ] brackets along with the key as index to obtain its value.

```
Dict1['Name']  
'Ruchi'
```



For the keys of the dictionary you can use only immutable objects but for the values you can use either mutable or immutable objects. Dictionaries can be nested with other dictionaries.

# Dictionary

Updating Dictionary:

- Dictionaries are mutable; you can add or modify elements. Just the way you access values using keys, you can modify values using keys.

```
# Change the value of 'Age' to '19'  
Dict1['Age']='19'  
Dict1  
{'Name': 'Ruchi', 'Age': '19', 'Class': 'Twelfth'}
```

```
# Adding a new element  
Dict1['School']='Kendriya Vidyalaya'  
Dict1  
{'Name': 'Ruchi',  
 'Age': '19',  
 'Class': 'Twelfth',  
 'School': 'Kendriya Vidyalaya'}
```

# Dictionary

Deleting Dictionary elements:

- You can delete the individual key-value pairs from the dictionary using **del**

command

```
del Dict1['Name']
```

```
Dict1
```

```
{'Age': '19', 'Class': 'Twelfth', 'School': 'Kendriya  
Vidyalaya'}
```

- Delete all key-value pairs at once using **clear()** function.

```
Dict1.clear()
```

```
Dict1
```

```
{}
```

# Dictionary

- Delete entire dictionary using **del** command.

```
del Dict1
Dict1
Traceback (most recent call last):<-->
  File "<ipython-input-24-6dd8ab4d7b02>",<br>
    Dict1
NameError: name 'Dict1' is not defined
```

An exception is raised because Dict1 doesn't exist anymore.

# Quick Recap

5 Standard Native Data Types: Numbers, Strings, Lists, tuples and Dictionary

Mutable  
Data Types

Lists and Dictionaries

Immutable  
Data Types

Strings and Tuples

Accessing

Use [ ] brackets along with the expression as index for Strings, Lists & Tuples and key for Dictionary

Updating

Specify index no. (for lists) and key (for Dictionary) between [ ] brackets and assign a new value

Deleting

Use **del** command for Strings, Lists, Tuples and Dictionaries

# Getting started with Python-

## Introduction to Libraries in Python

# What are Libraries?

- In Python, a library is used loosely to describe a collection of the core modules.
- A module is a .py file that contains certain functions and classes that can be used when called.
- The term ‘standard library’ in Python language refers to the collection of exact syntax, token and semantics of the Python language which comes bundled with the core Python distribution. There are more than 200 modules that forms a part of core Python. This library is written in the C language.
- “Additional libraries” are those libraries that can be added additionally to help with specific uses in Python. As an open source language, Python has a lot of additional libraries available.

# Contents

1. What are Libraries?
2. Importing Libraries in Python
3. Useful Libraries for Data Science

\

# Importing Libraries in Python

- You need to import a library into an environment before you can call it's functions in your program.
- To import any library just use any of the following codes -

```
import math as m  
from math import *
```

# Useful Libraries for Data Science

- There are a lot of Libraries available for scientific computations and data analysis in Python. Following is a list of Libraries that we will be using during this course :
- NumPy stands for Numerical Python. It is the fundamental package for scientific computing and is used to manipulate homogenous array based data. The most powerful feature of NumPy is n-dimensional array. This library also contains basic linear algebra functions, Fourier transforms, advanced random number capabilities and has an easy to use C API.
- Pandas contains high level data structures and manipulation tools which are used extensively for data munging and preparation. Pandas was added relatively recently to Python and have been instrumental in boosting Python's usage in data scientist community. It is built on top of NumPy.

# Useful Libraries for Data Science

- SciPy stands for Scientific Python. SciPy is built on NumPy. It is one of the most useful libraries for variety of high level science and engineering modules like discrete Fourier transform, Linear Algebra, Optimization and Sparse matrices.
- Matplotlib is used for plotting vast variety of graphs, starting from histograms to line plots to heat plots. You can also use Latex commands to add math to your plot.
- Scikit Learn is used for machine learning. Built on NumPy, SciPy and matplotlib, this library contains a lot of efficient tools for machine learning and statistical modelling including classification, regression, clustering and dimensionality reduction.
- Statsmodels for statistical modelling. Statsmodels is a Python module that allows users to explore data, estimate statistical models, and perform statistical tests. An extensive list of descriptive statistics, statistical tests, plotting functions, and result statistics are available for different types of data and each estimator.

# Quick Recap

NumPy	<ul style="list-style-type: none"><li>Manipulate homogenous array based data</li></ul>
Pandas	<ul style="list-style-type: none"><li>High level data munging and preparation</li></ul>
SciPy	<ul style="list-style-type: none"><li>High level science and engineering modules.</li><li>Discrete Fourier transform, Linear Algebra, Optimization and Sparse matrices.</li></ul>
Matplotlib	<ul style="list-style-type: none"><li>Plotting vast variety of graphs</li><li>Histograms, line plots, heat plots</li></ul>
Scikit Learn	<ul style="list-style-type: none"><li>Used for machine learning and statistical modeling.</li><li>Classification, regression, clustering and dimensionality reduction</li></ul>
Statsmodels	<ul style="list-style-type: none"><li>Explore data, estimate statistical models, and perform statistical tests</li></ul>

# Getting started with Python- Introduction to pandas Data Structures

# Contents

1. Introduction to pandas
2. Need for pandas
3. Introduction to pandas Data Structures
4. Series (Tasks: Creating, Indexing, Accessing elements)
5. DataFrame (Tasks: Creating, Indexing, Accessing elements, Slicing, Adding new column)

# Introduction to Pandas

pandas is a Python Package providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

- It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis / manipulation tool available in any language.
- It has a massive collection of many modules along with some unique features.

# Need for Pandas

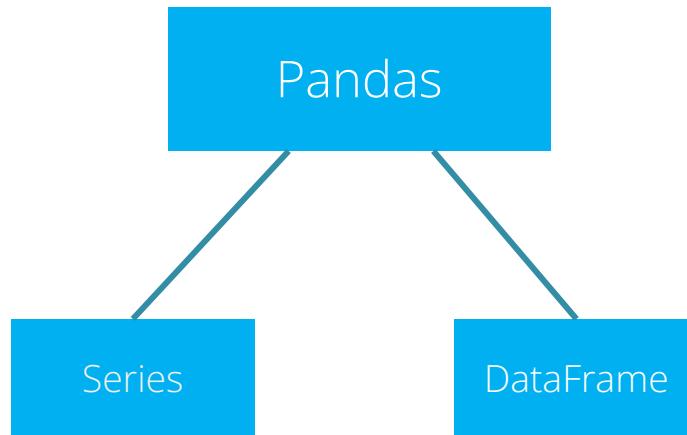
pandas makes data munging, preparing, analysis and modeling tasks easy and powerful with a few lines of code.

Some of the useful pandas techniques which gives it an edge over Python's built-in techniques:

- Creating DataFrames
- Loading data
- Crosstab
- Aggregating data
- Merging DataFrames
- Sorting DataFrames
- Plotting (Boxplot & Histogram)

# Introduction to pandas Data Structures

Pandas have 2 key data structure:



- Import pandas library for data management and modelling tasks.

```
# import pandas in python  
import pandas as pd
```

# Series

- Series is a one dimensional labeled array object similar to list or column in a table.
- It is capable of holding any sort of data type (integers, floating point numbers, strings, dictionaries, etc).

Create a series with an arbitrary list:

```
s=pd.Series([8, 'data', 5.36, -23455788675342648, 'structures'])  
s  
0          8  
1      data  
2      5.36  
3  -23455788675342648  
4  structures  
dtype: object
```

# Series

- You can explicitly define the index of Series, by default it starts from 'Zero'.

Define index with ‘`index=`’ argument:

```
s=pd.Series([8, 'data', 5.36, -23455788675342648, 'structures'],
index=['A', 'B', 'C', 'D', 'E'])
s
```

A	8
B	data
C	5.36
D	-23455788675342648
E	structures

dtype: object

# Series

- Series can convert a dictionary as well.
- If index values are not explicitly defined using index argument, it uses the keys of the dictionary as its index.

Pass a Dictionary object to Series:

```
d={'01' : 'Jan', '02':'Feb','03':'Mar', '04':'Apr'}  
d  
{'01': 'Jan', '02': 'Feb', '03': 'Mar', '04': 'Apr'}  
months=pd.Series(d)  
months  
01    Jan  
02    Feb  
03    Mar  
04    Apr  
dtype: object
```

# Series

Accessing elements of Series:

```
# Using Dictionary keys as its index
months['04']
'Apr'

# Using condition on value
months[months=='Jan']
01    Jan
dtype: object
```

# DataFrame

- DataFrame is a 2-dimensional labeled data structure, similar to a spreadsheet or SQL table or a dictionary of Series objects.
- It can hold any type of data (integers, floating point numbers, strings, series, dictionaries, another dataframe, etc)
- Create a 'basic\_salary' data with columns as 'First\_Name', 'Last\_Name', 'Grade', 'Location' and 'ba'.

# DataFrame

Pass a dictionary of lists to create a DataFrame

```
data={'First_Name': ['Alan', 'Agatha', 'Rajesh', 'Ameet', 'Neha'],
      'Last_Name': ['Brown', 'Williams', 'Kolte', 'Mishra', 'Rao'],
      'Grade': ['GR1', 'GR2', 'GR1', 'GR2', 'GR1'],
      'Location': ['DELHI', 'MUMBAI', 'MUMBAI', 'DELHI', 'MUMBAI'],
      'ba':[17990, 12390, 19250, 14780, 19235]}
basic_salary = pd.DataFrame(data, columns=['First_Name',
                                             'Last_Name', 'Grade', 'Location', 'ba'])
basic_salary
```

	First_Name	Last_Name	Grade	Location	ba
0	Alan	Brown	GR1	DELHI	17990
1	Agatha	Williams	GR2	MUMBAI	12390
2	Rajesh	Kolte	GR1	MUMBAI	19250
3	Ameet	Mishra	GR2	DELHI	14780
4	Neha	Rao	GR1	MUMBAI	19235

columns= argument allows us to give order of columns. By default they are alphabetically.  
Default indices

# DataFrame

- Indexing (getting slices or chunks of data) in DataFrame is similar to indexing in Series
- Give index to all the columns of 'basic\_salary'

Define index with '**index=**' argument

```
basic_salary = pd.DataFrame(data, index=['A','B','C','D','E'],
columns=['First_Name', 'Last_Name', 'Grade', 'Location', 'ba'])  
basic_salary
```

	First_Name	Last_Name	Grade	Location	ba
A	Alan	Brown	GR1	DELHI	17990
B	Agatha	Williams	GR2	MUMBAI	12390
C	Rajesh	Kolte	GR1	MUMBAI	19250
D	Ameet	Mishra	GR2	DELHI	14780
E	Neha	Rao	GR1	MUMBAI	19235

New indices

# DataFrame

## Accessing Columns

```
# Accessing columns using dictionary 'key' notation  
basic_salary['First_Name']  
# OR  
basic_salary.First_Name  
  
A      Alan  
B    Agatha  
C   Rajesh  
D   Ameet  
E     Neha  
  
Name: First_Name, dtype: object
```



Both the syntax above returns the same output as `basic_salary['First_Name']`. Using 'dot' notation also, columns can be accessed.

# DataFrame

## Accessing Rows

```
# Accessing rows using row index label  
basic_salary.loc[ 'B' , : ]
```

```
First_Name      Agatha  
Last_Name      Williams  
Grade          GR2  
Location       MUMBAI  
ba             12390  
Name: B, dtype: object
```



Use integers instead of rows index : basic\_salary.iloc[1,:]  
Use integer range : basic\_salary.iloc[1:3]

# DataFrame

- Show records of employees from Location MUMBAI and rows from index B to E.

Slicing – using condition and row index

```
basic_salary[basic_salary.Location=='MUMBAI']
```

	First_Name	Last_Name	Grade	Location	ba
B	Agatha	Williams	GR2	MUMBAI	12390
C	Rajesh	Kolte	GR1	MUMBAI	19250
E	Neha	Rao	GR1	MUMBAI	19235

```
# Slice along row indices
```

```
basic_salary.loc['B':'E']
```

	First_Name	Last_Name	Grade	Location	ba
B	Agatha	Williams	GR2	MUMBAI	12390
C	Rajesh	Kolte	GR1	MUMBAI	19250
D	Ameet	Mishra	GR2	DELHI	14780
E	Neha	Rao	GR1	MUMBAI	19235

# DataFrame

- Add a new column 'ms' to basic\_salary.

Adding a new Column using dictionary syntax

```
# Add a new column 'ms'  
basic_salary['ms']=[16070,6630,14960,9300,15200]  
basic_salary
```

	First_Name	Last_Name	Grade	Location	ba	ms
A	Alan	Brown	GR1	DELHI	17990	16070
B	Agatha	Williams	GR2	MUMBAI	12390	6630
C	Rajesh	Kolte	GR1	MUMBAI	19250	14960
D	Ameet	Mishra	GR2	DELHI	14780	9300
E	Neha	Rao	GR1	MUMBAI	19235	15200

# Quick Recap

## Pandas

- pandas is a Python Package providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.
- Import pandas in python : **import pandas as pd**

## Series

- Series is a one dimensional labeled array object similar to list or column in a table.
- **pd.Series():** Creates a series

## DataFrame

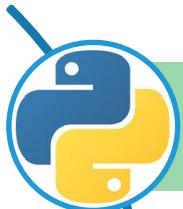
- DataFrame is a 2-dimensional labeled data structure, which can hold any type of data.
- **pd.DataFrame(data, columns=[], index=[]):** Creates a dataframe

# Getting started with Python- Working with Directories in Python

# Contents

1. Working with directories
  - i. Get current working directory
  - ii. Change working directory
  - iii. List the subdirectories
  - iv. Make directory
  - v. Rename or remove a file

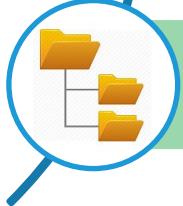
# Working with Directories



Python provides libraries containing methods that allow your programs to interact with files in your computer. Files and Directories form a Hierarchical structure in your system drive.



Files contain information. For example: csv files, word files, or python files.



Directories contain files and sub directories inside them. Directories can be helpful in managing large number of files in your program.

# Working with Directories

- Python has the 'os' module which provides us with several useful functions to work with files and directories.
- To get started, `import os` in your namespace

```
import os
```

- We can check current working directory or change working directory.
- `getcwd()` returns your current working directory in the form of a string.
- `chdir()` lets you change the working directory. The new path must be given as a string to this function.

```
# Check the current working directory
import os
os.getcwd()

'C:\\\\Users\\\\HP\\\\.spyder'

# Change the working directory
os.chdir("C:/Users/HP/Documents/")
```

Both forward (/) or backward slash(\) can be used.

# Working with Directories

- `listdir()` takes exactly 1 argument as a path and returns list of subdirectories and files in that path.

```
# List directories and files
os.listdir("c:/")

['$Recycle.Bin', 'Documents and Settings', 'hiberfil.sys',
'MSocache', 'pagefile.sys', 'PerfLogs', 'Program Files', 'Program
Files (x86)', 'ProgramData', 'Python27', 'Recovery', 'SWSetup',
'SWTOOLS', 'System Volume Information', 'Users', 'Windows']
```

# Working with Directories

- Create new directory

```
# Creating a new directory (using mkdir() function)
os.mkdir('mypython')
```

If the full path is not supplied, the new directory is created in the current working directory

- Rename a directory or file

```
# Renaming a directory or file (using rename() function)
os.rename('mypython','mynewpython')
```

First argument: old name  
Second argument: new name

- Remove a directory or file

```
# Removing directory or file (using rmdir() & remove() function)
os.rmdir('mynewpython')
os.remove('oldfile.txt')
```

rmdir() removes an empty directory  
remove() removes a file.

# Quick Recap

## Working with Directories

- **os.getcwd()** and **os.chdir()**: To check and set working directory respectively

## Create new directory

- **os.mkdir()**: To create a new directory

## Rename directory

- **os.rename()**: To rename a created a directory

## Remove directory & file

- **os.rmdir()** and **os.remove()**: To remove directory and file respectively

# Python Programming Basics

## Testing Data Types in Python

# Contents

## 1. Understanding Data Types in Python

- Numeric - Float
- Numeric - Integer
- Character
- Categorical
- Logical
- Vector
- Matrices
- Array
- Data Frame
- List

# Data Types in Python

Numeric -  
Float

Contains decimal numbers

Numeric -  
Integer

Contains only whole numbers

Character

Holds character strings

Categorical

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 Python

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 - Float

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
```

```
4.5
```

```
y = 3567
```

```
y
```

```
3567
```

```
type(x)
```



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

```
float
```

```
type(y)
```

```
int
```

# Numeric – Integer

Python shows type of object x as float. To convert it to an integer format, use `int()` function.

```
x =int(x)
```

```
x
```

```
4
```

```
type(x)
```

```
int
```

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

# Numeric – Integer

#To create an integer variable in Python use `int()` function.

```
f = int(22.5)  
f  
22
```

```
type(f)
```

```
int
```

```
x=8  
type(x)  
int
```

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

# 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 **str()** function.

```
z = "Welcome to Python Ready Reckon-er"
z
'Welcome to Python Ready Reckon-er'

x = "4.5"
x
'4.5'
type(z)
str
type(x)
str
```

Any value enclosed in quotes is stored as a character object

# Categorical

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

```
# Create an object x
```

```
import pandas as pd  
x = pd.Series(["high", "medium", "low", "low", "medium", "high",  
"high", "high", "medium", "low", "low"])
```

**pd.Series()** creates an  
one-dimensional ndarray.

```
# Check whether object x is
```

```
x.str.isalpha()
```

```
0    True  
1    True  
2    True  
3    True  
4    True  
5    True  
6    True  
7    True  
8    True  
9    True  
10   True  
dtype: bool
```

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

# Categorical

```
#Create a categorical object using Categorical() function
```

```
x = pd.Categorical(x)  
x.dtype
```

```
CategoricalDtype(categories=['high', 'low', 'medium'], ordered=False)
```

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

```
x.unique() ←
```

```
[high, medium, low]  
Categories (3, object):
```

categorical object x has 3 levels. By default the levels are **sorted chronologically**

# Categorical

```
#To specify the order of the factor
```

```
x = pd.Categorical(x, categories=['low', 'medium','high'])  
x  
[high, medium, low, low, medium, ..., high, high, medium, low, low]  
Length: 11  
Categories (3, object): [low, medium, high]
```

**categories=** 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  
isinstance(x,int)  
False
```

**isinstance()** returns true if the object argument is an instance of the classinfo argument

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

```
y = 4  
z = 7  
Result = y > z  
Result  
False
```

- With this kind of statement, you are asking Python 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 (bool standing for Boolean)
- You can check the class of the object using **type()**

# 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 = [1,2,5.3,6,-2,4]  
a  
[1, 2, 5.3, 6, -2, 4]
```

```
# Character vector
```

```
b = ["one","two","three"]  
b  
['one', 'two', 'three']
```

```
# Logical vector
```

```
import numpy as np  
d = np.array([4,24,6,4, 2,7])  
d > 5  
  
array([False,  True,  True, False, False,  True])
```

np.array() creates an array with one or multiple dimensions

# Matrices

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

```
# Create a matrix with 3 rows and 2 columns.
```

```
X = np.array([2, 3, 4, 5, 6, 7]).reshape(3,2)
```

```
X
```

```
array([[2, 3],  
       [4, 5],  
       [6, 7]])
```

- **np.array()** function is used to create a matrix (multi-dimensional array).
- **reshape(rows,cols)** is used to specify the dimensions of the matrix  
Note that the matrix is filled in by row-wise in Python.

# Matrices

It is possible to name the rows and columns of matrix during creation

```
import pandas as pd
x = pd.DataFrame(np.array([2, 3, 4, 5, 6, 7]).reshape(3,2),
index=[ "X", "Y", "Z"], columns=[ "A", "B"])
```

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

#Dimension names can be accessed or changed with three helpful functions **columns**, **index**, **rename()**

```
x.columns
Index(['A', 'B'], dtype='object')
x.index
Index(['X', 'Y', 'Z'], dtype='object')
x.rename(columns = {'A':'a', 'B':'b'}, inplace = True) ←
x.columns
Index(['a', 'b'], dtype='object')
```

rownames can be  
changed in similar  
manner

# 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 rows, two columns and three "tables".
```

```
a = np.arange(1,25).reshape(3,4,2) ←  
a
```

```
array([[[ 1,  2],  
       [ 3,  4],  
       [ 5,  6],  
       [ 7,  8]],  
  
      [[ 9, 10],  
       [11, 12],  
       [13, 14],  
       [15, 16]],  
  
      [[17, 18],  
       [19, 20],  
       [21, 22],  
       [23, 24]]])
```

**np.arange()** is one of the array creation routines based on numerical ranges. It creates an instance of ndarray with evenly spaced values and returns the reference to it.

**reshape (t,r,c)**

**t** = no. of tables

**r** = no. of rows

**c** = no. of columns

# 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 Python. To convert any object into a data frame type, use **pd.DataFrame()** function.

Create a data frame from different vectors

```
data = {'x':[12,23,45], 'y':[13,21,6], 'z':["a","b","c"]}  
df = pd.DataFrame(data)
```

	x	y	z
0	12	13	a
1	23	21	b
2	45	6	c

creating vectors x, y, z

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



DataFrames 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.

```
df.info() ←  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 3 entries, 0 to 2  
Data columns (total 3 columns):  
 x    3 non-null int64  
 y    3 non-null int64  
 z    3 non-null object  
dtypes: int64(2), object(1)  
memory usage: 152.0+ bytes
```

- **info()** shows the structure of an object
- **Note :** **z** is a character vector but by default Python stores it in the data frame as object

# Lists

List is a mutable, ordered sequence of elements. Each element inside a list is called items. Lists are defined by having values between square brackets. To convert any object into a list type, use **list()** function.

Create a list of 2 vectors - a numeric & other character

```
n=[2, 3, 5]
s=["aa", "bb", "cc", "dd", "ee"]
x=list(n+s)
x
[2, 3, 5, 'aa', 'bb', 'cc', 'dd', 'ee']
```

**list()** is used to create lists,

```
m = [2, 4, 6, 1, 3, 5]
p = ['even', 'odd']
y=list(m+p)
y
[2, 4, 6, 1, 3, 5, 'even', 'odd']
```

# Quick Recap

In this session, we discussed about various data types of Python and how to create them. Here is a quick recap:

## Data types

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

## Type Conversion Functions

`int(), str(), pd.DataFrame(), list(),  
np.array()`

# Python Programming Basics

## Numeric Functions and Operators in Python

# Contents

1. Introduction to Functions
2. General Functions
3. Introduction to Operators
4. Arithmetic Operators
5. Relational Operators
6. Miscellaneous Operators

# General Functions

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

```
#Absolute value of 'x'  
abs(-4)  
4  
x = [-4,4.5,-10.5,6]  
[abs(i) for i in x]  
[4, 4.5, 10.5, 6]`
```

```
#Square Root of 'x'  
import math  
math.sqrt(81)  
9.0
```

```
#Rounds to the nearest integer that's larger than x  
math.ceil(445.67)  
446
```

```
#Rounds to the nearest integer that's smaller than x  
math.floor(445.67)  
445
```

# General Functions

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

```
#Rounds to the nearest integer toward 0.  
math.trunc(445.67)  
445
```

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

```
round(44.5682,2)  
44.57
```

# General Functions

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

```
#Computes natural logarithms.
```

```
import numpy as np
```

```
np.log(50)
```

```
3.912023005428146
```

```
np.log([44,55])
```

```
array([3.78418963, 4.00733319])
```

```
math.log(50)
```

```
3.912023005428146
```

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

```
math.log(8,2)
```

```
3.0
```

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

```
math.log10(55)
```

```
1.7403626894942439
```

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

```
math.exp(6)
```

```
403.4287934927351
```

Natural log can be calculated using  
numpy as well as math library.

# Operators

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

# Arithmetic Operators

```
x = 7  
y = 18
```

```
#Addition  
x+y  
25
```

```
#Subtraction  
x-y  
-11
```

```
#Multiplication  
x*y  
126
```

# Arithmetic Operators

```
#Division
```

```
y/x
```

```
2.5714285714285716
```

```
#Exponentiation
```

```
y**x
```

```
612220032
```

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

```
y%x
```

```
4
```

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

```
y//x
```

```
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

Python

```
x = 7  
y = 18
```

```
#Less than
```

```
x < y
```

```
True
```

```
#Greater than
```

```
x > y
```

```
False
```

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

```
x <= 5
```

```
False
```

# Relational Operators

```
#Greater than or equal to  
y>=20  
False
```

```
#Equal to  
y==16  
False
```

```
#Not equal to  
x!=5  
True
```

# Miscellaneous Operators

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

```
x = [*range(1, 6, 1)]  
X  
1 2 3 4 5
```

**range()** represents an immutable sequence of numbers. Here range is from 1 to 6 with 1 unit interval. In python upper limit of range is always 1 less than the specified value.

```
x = 10  
t = [*range(1, 9, 1)]  
x in t  
False
```

**in** 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 Python. Here is a quick recap:

## General Functions

- **abs(), sqrt(), ceil(), floor(), trunc(), log(), log2(), log10(), exp()**

## Operators

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

# Python Programming Basics

## String Functions in Python

# Contents

1. Introduction to Strings
2. Understanding String Manipulation Functions
  - join()
  - format()
  - split()
  - lower()
  - upper()
  - casefold()
  - replace()

# Introduction to Strings

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

'A character string using single quotes'

"A character string using double quotes"

Internally Python stores every string within single quotes, even when you create them with double 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

```
a = "Hello"  
b = 'How'  
c = "are you?"
```

```
a+" "+b+" "+c  
'Hello How are you?'
```

In Python we can concatenate strings by using + and specify separator in between

```
a+"-"+b+"-"+c  
'Hello-How-are you?'
```

In this case it's "-".

```
[y" + str(i) for i in range(1,5)]
```

```
['y1', 'y2', 'y3', 'y4']
```

single character "y" is pasted with the sequence 1:4

# Concatenating Strings – join()

- **join()** is called on a string with argument as a list value and returns a single string value which is a concatenation of each string in the argument list.

```
",".join(["orange", "blue", "green"])  
'orange,blue,green'
```

Notice that **join()** is called on “,” (can be customised) which is inserted between each string of the list..

- **split()** is called on a string value and returns a list of strings

```
"split does the opposite of join".split()  
['split', 'does', 'the', 'opposite', 'of', 'join']
```

By default, it splits wherever the whitespace characters are found. You can specify a delimiter string to the **split()** function to split upon.

# Formatting Strings

String Formatting Operator : %

- Unique to strings
- C's printf() like facility
- Formatting specifier is %

List of symbols which can be used along with %:

```
print("Name is %s and score is %d" %('Sana', 44))
```

```
Name is Sana and score is 44
```

```
print("Name is %s and grade is %c" %('Sana', 'A'))
```

```
Name is Sana and grade is A
```

**%s** gives string conversion via str()  
prior to formatting.  
**%d** returns signed decimal integer.  
**%c** returns character.

# Formatting Strings - format()

String Formatting Function: format()

- Formatting specifier is { }

```
"This is a {}".format("new formatting style")
'This is a new formatting style'
```

- You can also use the positional index. This allows rearranging the order of display with no change in arguments.

```
"Learn {0} on {1}.format("Python", "Sanaitics Kit")
'Learn Python on Sanaitics Kit.'
```

# Extracting and Replacing

One common task when working with strings is the extraction and replacement of some characters. In Python extraction is same as subsetting. It can be done by simply using [ ] .

```
# Extract 'THE'
```

```
a = "WELCOME TO THE WORLD OF PYTHON"  
a[11:14]
```

```
'THE'
```

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

# Replacing Substrings – replace()

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

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

```
replace(old,new,string)
```

```
# Using replace()
```

```
string = "She is a data scientist. She works with an MNC"  
string.replace("She","Sharon")
```

```
'Sharon is a data scientist. Sharon works with an MNC'
```

**replace()** finds the first instance of the ‘She’ within string and replaces it with ‘Sharon’

# Replacing Substrings – replace()

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

```
Location = ["Mumbai", "Delhi", "Mumbai", "Kolkata", "Delhi"]
```

```
Location = [word.replace(word, word[0:3]) for word in Location] ←
```

```
Location
```

```
['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.
- ❑ **replace()** will replace a string with the new substring.

# Splitting a String - `split()`

Besides the tasks of extracting and replacing substrings, another common task is splitting a string based on a pattern. In Python we have a function `split()` 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 `split()`.

```
sentence = "break a string into individual components"  
sentence.split()
```

```
['break', 'a', 'string', 'into', 'individual', 'components']
```

`split()` is a regular expression pattern used for splitting

```
# split each date component
```

```
dates = ["12-10-2014", "01-05-2000", "26-06-2015"]  
dates = [date.split('-') for date in dates]  
dates
```

```
['12 10 2014', '01 05 2000', '26 06 2015']
```

Here, date components joined by a dash “-”, are split

# Other Basic String Manipulation Functions

```
#Converts any lower case characters into upper case  
[x.upper() for x in ["ASK ANALytics","data SCience"]]  
['ASK ANALYTICS', 'DATA SCIENCE']
```

```
#Converts any upper case characters into lower case  
[x.lower() for x in ["ASK ANALytics","data SCience"]]  
['ask analytics', 'data science']
```

```
#casifold function which is similar to lower function.
```

```
("all characters in LOWER case").casifold()  
'all characters in lower case'
```

It converts all characters to lower case

```
#replace function which replaces a character  
("This is a string").replace('a', 'A')  
'This is A string'
```

replace() function replaces characters with specified character.

# Quick Recap

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

Here is the quick recap:

## String Manipulation Functions

- **+** concatenates character strings
- **join()** concatenates list of strings with specified separator
- **format()** formats numbers and strings to a specified style
- **replace()** replaces all instances of a substring
- **split()** which splits the elements of a character vector into substrings
- **upper()** Converts any upper case characters into lower case
- **lower()** Converts any lower case characters into upper case
- **casefold()** Converts upper case characters into lower case

# Python Programming Basics

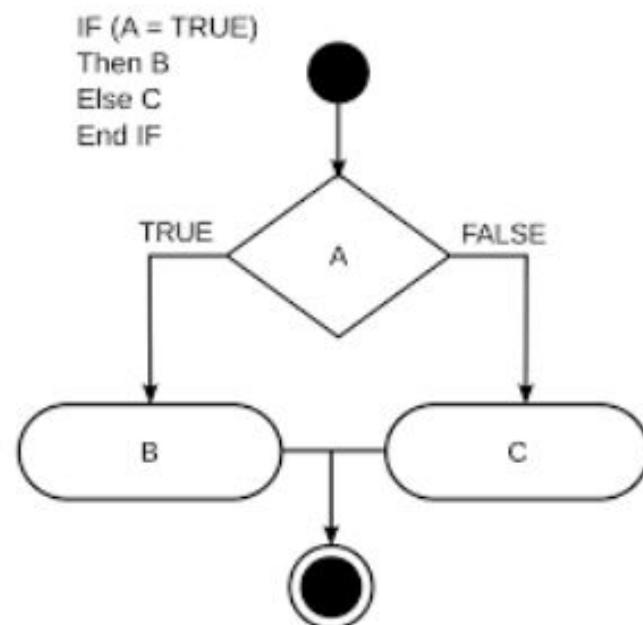
## if..else Conditional Statements

# Contents

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

# Introduction

- 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.

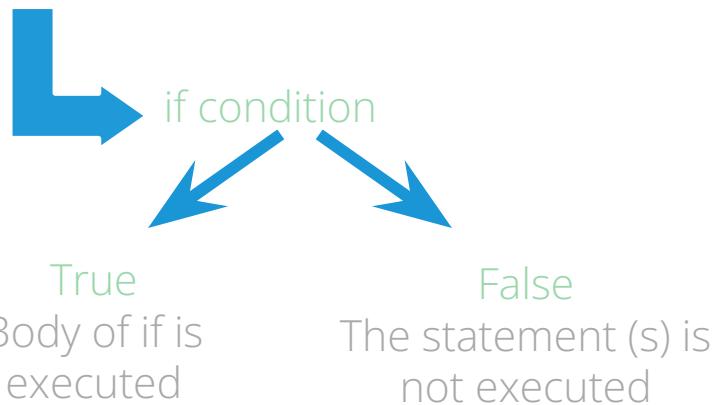


# if Statement

## Syntax:

```
if Boolean expression:  
    statement(s)
```

The program evaluates



# 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")
```

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 isinstance(x,str):  
    print(x)
```

Hello there

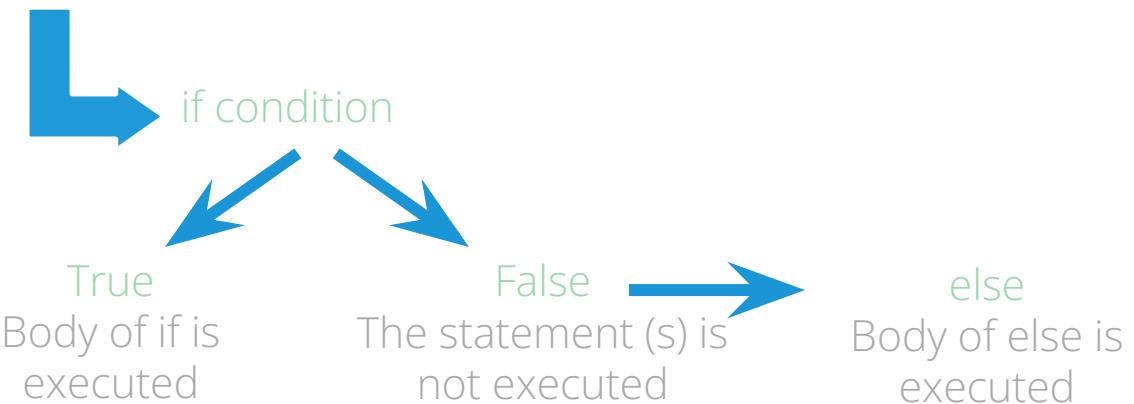
- ❑ **isinstance()** returns TRUE or FALSE depending on whether the argument is of string (str) type or not.
- ❑ Here **x** is a string, hence the condition returns TRUE and “Hello there” is printed

# if..else Statement

## Syntax:

```
if Boolean expression:  
    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')
```

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 **if..else()** function instead.

# if..else Statement

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

```
num = int(input("Enter a number: "))  
if num < 0: ←  
    print("Enter a positive number")  
else:  
    sum=(num * (num + 1)) / 2  
    print("The sum is", sum)
```

```
Enter a number: 5  
The sum is 15.0
```

❑ **input()** interactively takes the user input values  
Here the input is 5, hence the sum of 5 natural number is 15

# if..else inside print Function

Using **if..else** statement for vector, we can use for loop to execute **if..else** condition in a single statement

- If the condition is TRUE it returns x else y
- **if..else** cannot take a vector as an input in python but results into a vector. Hence we use for loop along with **if..else()**

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

```
a = [6,1,5,14]  
print([( "even" if x%2 == 0 else "odd") for x in a])  
['even', 'odd', 'odd', 'even']
```

# Nested if..else Statement

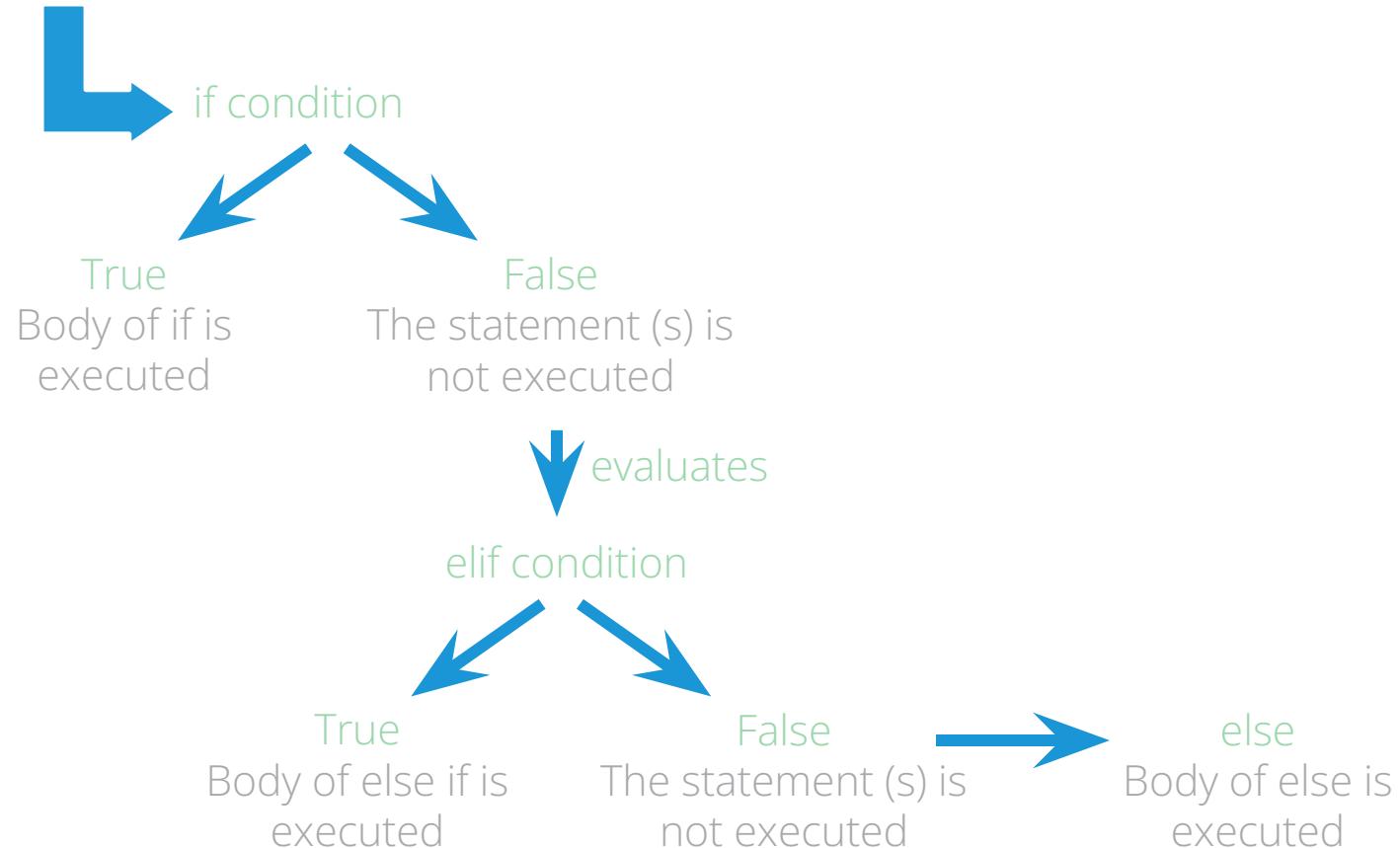
**elif** is short for "**else if**". It allows us to check multiple expressions. Similar to **else**, **elif** statement is optional.

## Syntax:

```
if Boolean expression1:  
    statement(s)  
elif Boolean expression2:  
    statement(s)  
elif Boolean expression3:  
    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

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

```
x=-11
if x>0:
    print('x is a positive number')
elif x==0:
    print('x is zero')
else:
    print('x is a negative number')
x is a negative number
```

The **if** condition and  
**elif** condition  
evaluates to False so  
it executes the  
statements in the  
block 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. **if..else()** 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 **elif** statement

# Python Programming Basics

## Loops in Python

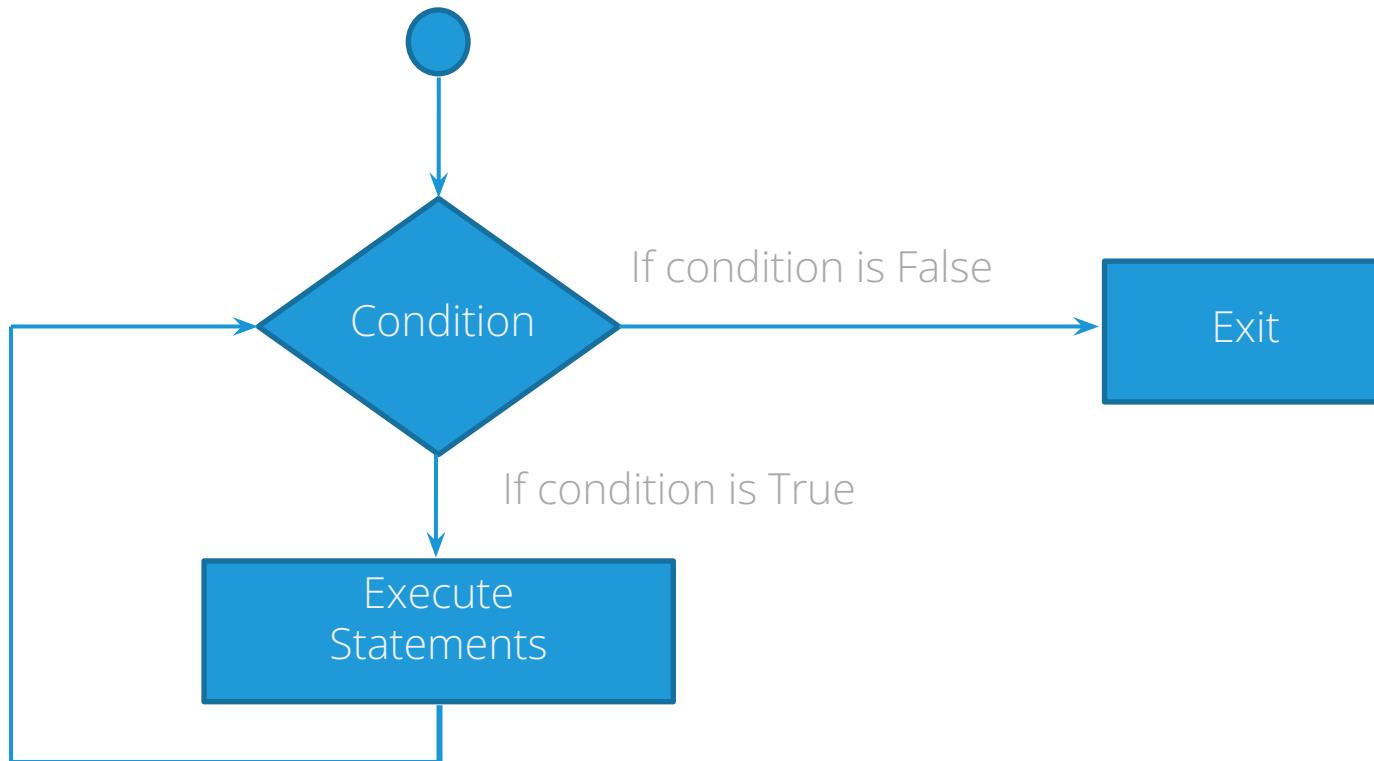
# Contents

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

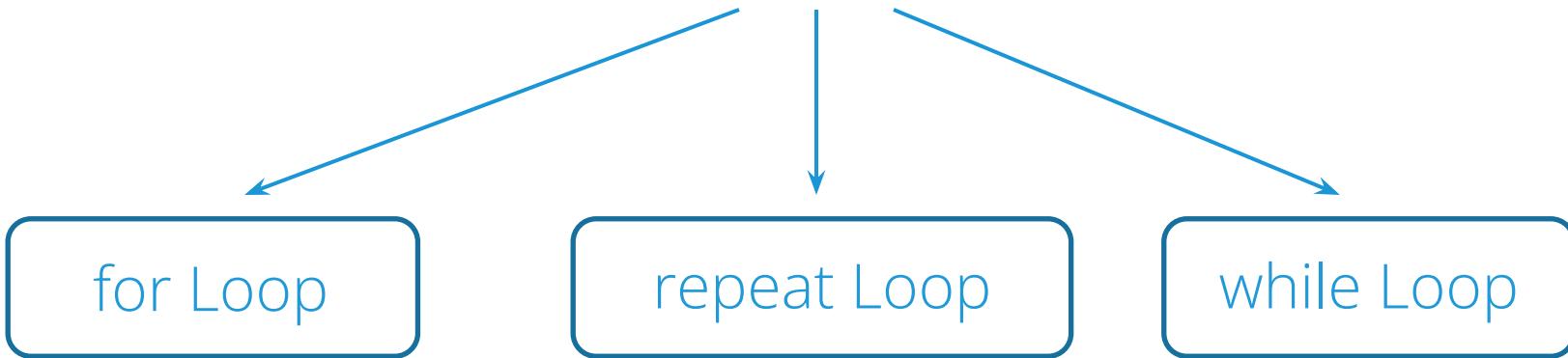
# Introduction

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 <b>for</b> consisting of block of program statements contained after :

# for Loop

- In **for** loop, the number of times the loop should be executed is typically defined in the initialisation 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 range(1,5):
    print(i)
```

```
1
2
3
4
```

# for Loop

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

```
Location = ["Mumbai","Delhi","Delhi","Mumbai","Mumbai","Delhi"]
count = 0

for i in Location:
    if (i == "Mumbai"):
        count = count + 1

countdelhi = len(Location)-count
print("Mumbai:",count)
print("Delhi:",countdelhi)
```

Mumbai: 3

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**.

# for Loop

- For loop is the most famous among all loops available in Python 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 next loops may come to the rescue.

# 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 <b>for</b> consisting of block of program statements contained after :

# 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 range(1,6):
    for j in range(1,3):
        print(i*j)
```

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

# Nested for Loop

## Important Note :

- While writing a nested for loop, always practice indentation.
- Python creates an indentation automatically, it follows indentation rules strictly.  
Hence, if there is some wrong indentation then Python gives indentation error while execution.

# while Loop

## Syntax:

```
while test_expression:  
    Statement 1  
    Statement 2  
    ....
```

Statement 1,  
Statement 2,...

Body of while consisting of  
block of program  
statements contained after :

# 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 < 11):
    print(i)
    i = i+2
```

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

```
1
3
5
7
9
```

# while Loop

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

```
number = int(input("Enter a number: "))  
if number < 0:  
    print("Enter a positive number")  
else:  
    sum = 0  
    while(number > 0):  
        sum = sum + number  
        number = number - 1  
    print("The sum is", sum)
```

```
Enter a number: 3  
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 : input() is explained in if..else Conditional Statements tutorial.

# repeat using while Loop

- In Python there is no separate function for repeat loop. This loop is similar to previous **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 using while Loop

## Syntax:

```
while True:  
    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 after :

# repeat using while 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 0 to 10
```

```
total = 0
while True:
    print(total)
    total = total + 2 ←
    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

```
0
2
4
6
8
```

# Interruption and Exit Statements in Python

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 to use **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 Python

```
# print numbers between 1 to 15 which are not divisible by 4
```

```
for i in range(1,15):
    if(i%4!=0):
        next
    else:
        print(i)
```

```
4
8
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.

# Interruption and Exit Statements in Python

```
# break the loop if a number in a range of 1 to 15 is divisible 4
```

```
for i in range(1,15):
    if(i%4==0):
        break
    else:
        print(i)
```

```
1
2
3
```

**if** statement checks whether value in **i** is divisible by 4 or not; if it returns TRUE then it exits the for loop and stops further 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

## Interruption and exit statements

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

## Python Programming Basics

### Testing Working with Dates and Time in Python

# Contents

1. Introduction
2. datetime library Functions
3. Merge Three Different Columns Into a Date in Python
4. Format a Vector With Inconsistent Date Formats

# Introduction

- Python has a range of date manipulation functions in datetime library 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 datetime library of Python, converts a variety of character date formats into Python dates. Once converted to dates, the following functions will return information about dates: **second, minute, hour, month, year**
- Pandas also has a Timestamp function. The inbuilt function offers a nice way to make easy parsing in dates and times.

# Base Package Functions

`datetime.strptime()` converts dates entered as strings into numeric dates.

`datetime.strptime(x, "%Y-%m-%d")`

x is a string object to be converted

"`%Y-%m-%d`" is the format (in which the date appears within the string) composed of codes such as:

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

# datetime library functions

```
# Formatting a date
```

```
from datetime import datetime
x = '5 jan 2010'
ndate = datetime.strptime(datetime.strptime(x, '%d %b %Y'), '%d %b %Y')
ndate
```

```
'05 Jan 2010'
```

```
type(ndate)
```

```
str
```

```
ndate2 = datetime.strptime(x, '%d %b %Y') ←
type(ndate2)
```

```
datetime.datetime
```

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

```
datetime.strptime(ndate, '%d %b %Y').strftime('%Y%b')
```

```
'2010January'
```

In Python we need to specify the format of the input date.

**datetime.strptime()** converts x to a datetime object

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

# datetime library functions

```
#To extract Day of the week.  
datetime.strptime(ndate, '%d %b %Y').strftime('%A')  
'Tuesday'
```

```
#To extract Month of the Year.  
datetime.strptime(ndate, '%d %b %Y').strftime('%B')  
'January'
```

Apart from datetime library, pandas also have functions that deal with timestamps.

```
#To extract Quarter no.  
import pandas as pd  
pd.Timestamp(ndate).quarter  
1
```

# datetime library functions

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

```
datetime.strptime("12-01-2015", "%m-%d-%Y").strftime("%Y-%m-%d")  
'2015-12-01'
```

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

```
date = datetime.now()  
date  
datetime.datetime(2019, 11, 13, 11, 46, 37, 306825)
```

- The letters **Y**, **m**, and **d** correspond to the year, month, and day elements of a date-time.
- strftime()** is used to specify the format in which one wants the date to be in.

# datetime library functions

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

```
date.hour
```

```
11
```

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

```
date.minute
```

```
46
```

---

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

```
date.second
```

```
37
```

# datetime library functions

```
# Capture current date
```

```
from datetime import date  
today = date.today()  
today  
  
datetime.date(2019, 11, 13)
```

**date.today()** returns the  
your system's current date

```
# Using operators with dates
```

```
d1 = datetime.date(datetime.strptime("20101201", '%Y%m%d'))  
d2 = datetime.date(datetime.strptime("10/7/04", '%m/%d/%y'))
```

```
d1
```

```
d2
```

```
datetime.date(2010, 12, 1)
```

```
datetime.date(2004, 10, 7)
```

```
d1-d2
```



```
datetime.timedelta(2246)
```

Different operators can be  
used with date objects

# Merge Three Different Columns Into a Date in Python

```
# My Dataframe
```

```
d = {'EmpID' : [101,102,103,104,105],  
      'year' : [1977,1989,2000,2012,2015],  
      'month' : [2,5,10,1,11],  
      'day' : [2,3,1,1,5]}  
  
datedf = pd.DataFrame(d)  
datedf
```

	EmpID	year	month	day
0	101	1977	2	2
1	102	1989	5	3
2	103	2000	10	1
3	104	2012	1	1
4	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 Python

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

```
# Merge 3 columns into one date column  
datedf['Date']=pd.to_datetime(datedf[['year','month','day']])  
datedf
```

	EmpID	year	month	day	Date
0	101	1977	2	2	1977-02-02
1	102	1989	5	3	1989-05-03
2	103	2000	10	1	2000-10-01
3	104	2012	1	1	2012-01-01
4	105	2015	11	5	2015-11-05

- ❑ new column date is created using []
- ❑ **to\_datetime()** converts date column into a date type

# Format a Vector With Inconsistent Date Formats

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

```
dates = ["12aug08", "01sep09", "7august06", "9august2007", "20july1999"]
ndates = pd.to_datetime(dates)

ndates
DatetimeIndex(['2008-08-12', '2009-09-01', '2006-08-07', '2007-08-09',
                '1999-07-20'],
               dtype='datetime64[ns]', freq=None)
```

Note that Pandas function `to_datetime()` is capable of handling such discrepancies as long as order of the date elements is consistent.

# Quick Recap

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

datetime functions

- **today(), now(), hour(), minute(), second(), quarter, month, year**

Pandas Timestamp  
functions

- **to\_datetime()**

Date manipulation  
tasks

- Merge Three Different Columns Into a Date in Python using **pandas()**

# Data Management in Python -

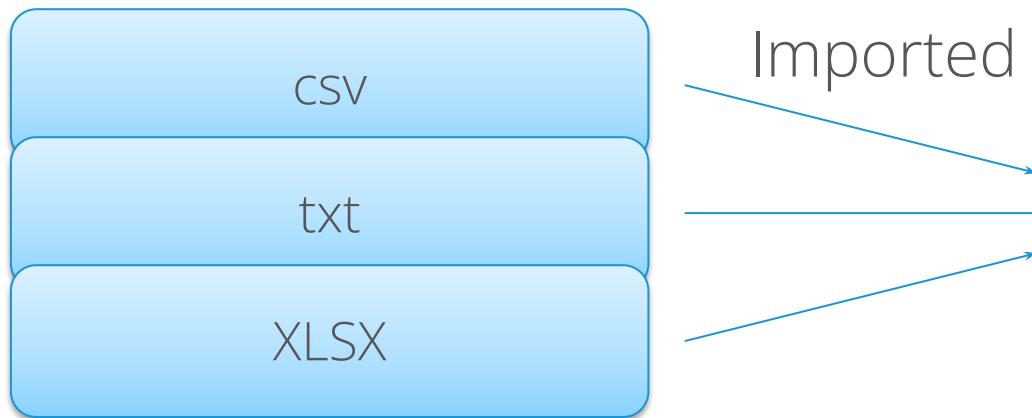
## Importing & Exporting Data

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

# Contents

1. Understanding the Data and Need for Importing Data
2. Common File Formats
3. Importing Files Using pandas
  - i. `read_csv()`
  - ii. `read_table()`
  - iii. `read_excel()`
4. Handle Missing Observations
5. Exporting CSV, Text and XLSX Files
6. Using SQLAlchemy in Pandas for creating engine
7. Importing & Exporting Files using SQLAlchemy Wrapper in pandas - `read_sql()`
8. Importing SAS files in pandas - `read_sas()`
9. Importing STATA files in Pandas - `read_stata()`

# Your 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.

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		

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		

# Using Pandas Library

- To import files of different formats
- pandas is a Python Package providing **high-performance, easy-to-use data structures**
- **data analysis tools**
- Pandas will be our preferred library for **data management**.

# 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

# read\_csv() Function

Importing a [.csv](#) file

```
import pandas as pd  
salary_data = pd.read_csv("C:/Users/Documents/basic_salary.csv")
```

`pd.read_csv()` assumes **header = TRUE** and **sep = ","** by default.



First locate your data file, whether it is saved in the default working directory of Python 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 Python. 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 [.txt](#) file

```
import pandas as pd  
salary_data = pd.read_table("C:/Users/Documents/basic_salary.txt")
```

**header = infer** (default) indicates that the first row of the file contains the names of the columns.  
Pass 0 if you wish to explicitly define column names.

**sep = "/t"** (default) specifies that the data is separated by tab.

**delim\_whitespace =** specifies whether whitespace is supposed to be considered as a delimiter.  
Default value is false.

**names =** array of column names you wish to define. Eg. **names = ['A', 'B', 'C']**



First locate your data file, whether it is saved in the default working directory of Python 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 Python. 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\_excel() Function

Importing a [.xlsx](#) file

```
import pandas as pd  
salary_data = pd.read_excel("C:/Users/Documents/basic_salary.xlsx")
```

**sheetname=** defines the excel sheet to import data from. Takes Sheet1 by default. "Sheet\_name" indicates the name of the sheet to be imported. number indicates the index number of the sheet to be imported. Starts from 0.

**header = infer** (default) indicates that the first row of the file contains the names of the columns. Pass 0 if you wish to explicitly define column names.

**index\_col=** number, helpful if you want to define a specific column as data index.



First locate your data file, whether it is saved in the default working directory of Python 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 Python. 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\_excel() Function

Importing multiple sheets from [.xlsx](#) file

- To facilitate working with multiple sheets from the same file, the **pd.ExcelFile** can be used to wrap the file and can be passed into **read\_excel()**

ExcelFile class is best for importing sheets using different arguments as specification.

```
salary_data={}
with pd.ExcelFile("C:/Users/Documents/basic_salary.xlsx") as xls:
    salary_data['Sheet1'] = pd.read_excel(xls, 'Sheet1', index_col=None)
    salary_data['Sheet2'] = pd.read_excel(xls, 'Sheet2', index_col=1)
```

# How Does Python Handle Missing Observations?

- By default, Pandas interprets the following values as null values –  
'-1.#IND', '1.#QNAN', '1.#IND', '-1.#QNAN', '#N/A N/A', '#N/A', 'N/A', '#NA', 'NULL', 'NaN', '-NaN', 'nan', '-nan', ''
- The read methods also accept the following arguments which deal with missing observations -
  - **na\_values**= Accepts strings, dictionaries of additional values to be considered null.
  - **na\_filter**= TRUE (default), detects missing value markers (like empty strings and the value of **na\_value** values).
  - **skip\_blank\_lines**= TRUE (default), skips blank lines and moves on to the next data entry.



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.

# Exporting CSV, Text and XLSX Files

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

```
# To a CSV
```

```
salary_data.to_csv('path_to_file/file_name.csv')
```

- ❑ **path\_to\_file** here is the path where the file needs to be saved.
- ❑ **file\_name** is the name you want to specify for that file.

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

```
salary_data.to_csv('path_to_table/file_name.txt', sep='\t',  
index=False)
```

```
# To an Excel Spreadsheet
```

```
salary_data.to_excel('path_to_file/file_name.xlsx')
```

# Using SQLAlchemy in Pandas

- To import files from different database management systems that require different API's such as MySQL, PostgreSQL and Oracle
- Pandas uses a database abstractor **SQLAlchemy**
- SQLAlchemy uses the **create\_engine** function to create an **Engine** object
- To create this object, **SQLAlchemy** uses the most prominent **DBAPI** driver
-

# create\_engine() Function

Create a MySQL, PostgreSQL and Oracle file

```
import pandas as pd
from sqlalchemy import create_engine

#Create Engine
#PostgreSQL
engine =
create_engine('postgresql://scott:tiger@localhost/mydatabase')

#MySQL
engine = create_engine('mysql://scott:tiger@localhost/foo')

#Oracle
engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')
```



Remember, the create function should point to the address of your database which includes your username for the database, the IP/hosting address, password as well as your database name.

# SQL Functions in SQLAlchemy

Importing a MySQL, PostGreSQL, Oracle database file

```
salary_data = pd.read_sql_table('data', engine)
salary_data = pd.read_sql_query('SELECT * FROM data', engine)
```

- ❑ 'data' here is the name of the table being imported.
- ❑ '**SELECT \* FROM data**' is the SQL query being passed to get the data.

Exporting a file to an SQL database

```
salary_data.to_sql('name of database', engine)
```

# Importing SAS files in Pandas

- Pandas can read SAS files but not write SAS xport (.XPT) and SAS7BDAT (.sas7bdat) format files.
- SAS files only contains two value types: ASCII text and floating point values (usually 8 bytes but sometimes truncated). For xport files, there is no automatic type conversion to integers, dates, or categoricals. For SAS7BDAT files, the format codes may allow date variables to be automatically converted to dates.

```
import pandas as pd  
salary_data = pd.read_sas('sas_data.sas7bdat')
```

# Importing STATA files in Pandas

Just like in the case of SAS files, Pandas can read STATA files but not write them, thus they cannot be exported as a STATA file.

```
import pandas as pd  
salary_data = pd.read_stata('file.stata')
```

- **convert\_categoricals** indicates whether value labels should be read and used to create a Categorical variable from them.
- **convert\_missing** indicates whether missing value representations in Stata should be preserved. If **False** (the default), missing values are represented as np.nan. If **True**, missing values are represented using **StataMissingValue** objects, and columns containing missing values will have object data type.

# Quick Recap

## Import Files Using Pandas

- **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()**
- **read\_excel** is used to import an Excel file.

## Handle Missing Values

- Use **na\_values** to explicitly define what values are to be considered null.
- Use **skip\_blank\_lines** to skip rows with blank spaces.

## Export Files Using Pandas

- **to\_csv** exports csv files
- **to\_csv(sep= '\t')** exports tab delimited txt files
- **to\_excel** exports excel file.

# Quick Recap

## Importing & Exporting MySQL, PostGreSQL and Oracle files

- DBAPI wrapper SQLAlchemy is used
- **read\_sql\_table()** imports the database while **read\_sql\_query()** imports the results of the specified query on the database.
- **data.to\_sql()** is used to read a file to an sql database

## Import SAS files

- Use **read\_sas()** function in native pandas. These files cannot be exported.

## Import STATA files

- Use **read\_stata()** function in native pandas. These files cannot be exported.

# Data Management in Python -

## Importing & Exporting Data

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

# 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. Derive a New Variable
12. Recode a Categorical Variable
13. Recode a Continuous Variable into Categorical Variable
14. Create a Decile Object
15. Remove Columns from a Data Frame
16. Remove Rows from a Data Frame

# Data Snapshot

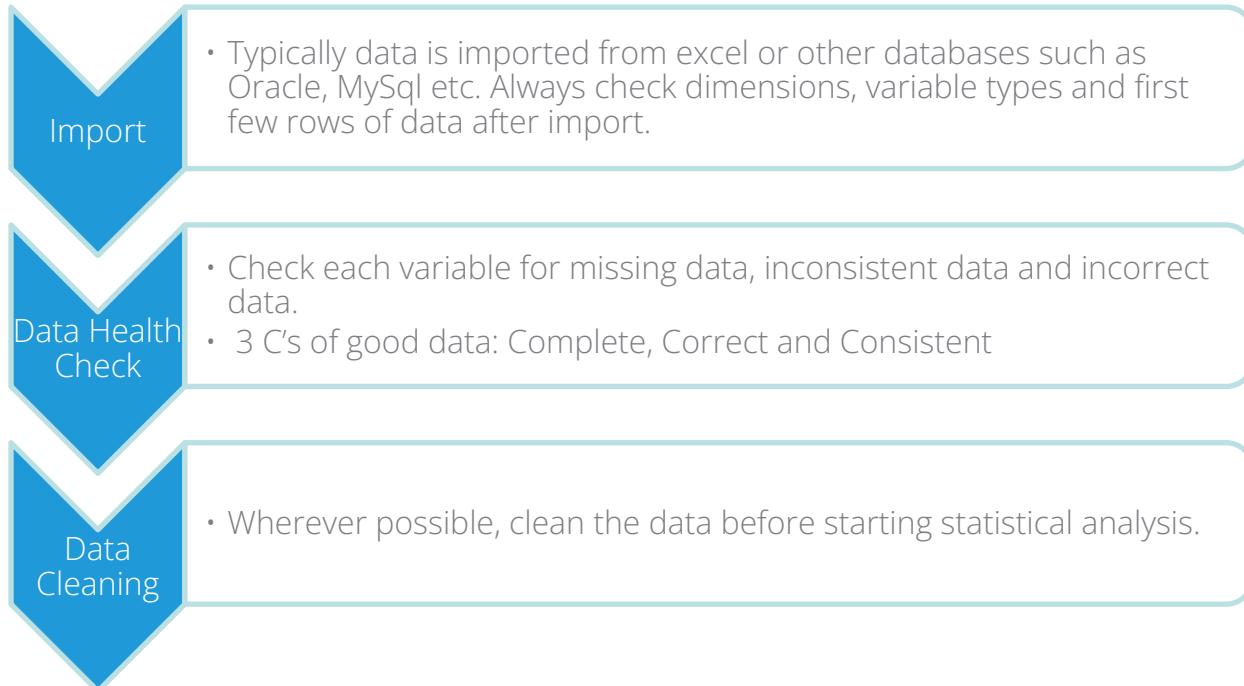
basic\_salary data consist salary of each employee with it's Location & Grade.

## Variables

Observations

First_Name	Last_Name	Grade	Location	ba	ms
Alan	Brown	GR1	DELHI	17990	16070
Agatha	Williams	GR2	MUMBAI	12390	6630
Rajesh	Kolte	GR1	MUMBAI	19250	14960
Columns		Description		Type	Measurement
First_Name	Last_Name	First Name		character	-
Last_Name		Last Name		character	-
Grade		Grade		character	GR1, GR2
Location		Location		character	DELHI, MUMBAI
ba		Basic Allowance		numeric	Rs.
ms		Management Supplements		numeric	Rs.

# First Look at the Data



# Why Data Checking is Important?

After importing the data into Python and before analysing the data, it is very important to understand your data & check that it has maintained the correct format and also to know how your data looks like, how are the variables treated, what types of variables data contains, how many missing observations are there in your data, how many rows and columns does it contain, etc. so that you get familiar with the data with which you are working. This preliminary step is the foundation for further data analysis, it helps you make your initial inferences about the data before you can start modelling/testing it.

```
# Import basic_salary data  
  
import pandas as pd  
# current directory is already specified  
salary_data_org= pd.read_csv('basic_salary.csv')
```

# Dimension of Data and Names of the Columns

Use the following commands to know how many rows and columns are there in our data and the names of the columns it contains:

```
# Retrieve the dimension of data
```

```
salary_data_org.shape  
←  
(12, 6)
```

- **shape** gives row and column dimension of the data.  
This data contains 12 rows and 6 columns.
- Alternatively, **data.shape[0]** and **data.shape[1]** can be used separately to know no. of rows and columns respectively.

```
# Get the Names of the columns
```

```
list(salary_data_org)  
↑  
['First_Name', 'Last_Name', 'Grade', 'Location', 'ba', 'ms']
```

- **list()** gives column names.
- You can also use **salary\_data.columns** instead to get the column names

# Internal Structure of Data

When Python reads data, it treats different variable types in different ways. `info()` compactly displays a dataframe's internal structure:

```
salary_data_org.info()
```

```
# Output
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12 entries, 0 to 11
Data columns (total 6 columns):
First_Name      12 non-null object
Last_Name       12 non-null object
Grade           12 non-null object
Location        12 non-null object
ba              12 non-null int64
ms              11 non-null float64
dtypes: float64(1), int64(1), object(4)
memory usage: 656.0+ bytes
```

Character variables are entered into a dataframe as object in Python

This gives us the following information:

- Type of the variable.
- Memory usage of the data

# Check Levels of a Categorical Variable

Our data has 4 object variables. A variable of data type 'object' is a categorical variable but in Python it has to be explicitly converted to the data type 'category' to be treated as one. Let's convert the variable Location to 'category' and check the number of levels it has using the `column.cat.categories` method:

```
salary_data_org['Location']=salary_data_org['Location'].astype('category')
salary_data_org['Location'].cat.categories
Index(['DELHI', 'MUMBAI'], dtype='object')
```

# 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 `memory_usage()` function to get an estimate in bytes.

```
salary_data_org.memory_usage()
```

```
Index      80
First_Name 96
Last_Name  96
Grade       96
Location    108
ba          96
ms          96
dtype: int64
```

# Number of Missing Observations

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

```
salary_data_org.isnull().sum()
```

```
First_Name      0  
Last_Name       0  
Grade           0  
Location        0  
ba              0  
ms              1  
dtype: int64
```

- ❑ **isnull()** returns a Boolean dataframe that returns TRUE for each null value'
- ❑ **sum()** displays the sum of missing observations.

# First n Rows of Data

To check how your data looks, without revealing the entire data set, which could have millions of rows and thousands of columns, we can use `head()` to obtain first n observations.

```
salary_data_org.head()
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
0	Alan	Brown	GR1	DELHI	17990	16070.0
1	Agatha	Williams	GR2	MUMBAI	12390	6630.0
2	Rajesh	Kolte	GR1	MUMBAI	19250	14960.0
3	Ameet	Mishra	GR2	DELHI	14780	9300.0
4	Neha	Rao	GR1	MUMBAI	19235	15200.0



By default, `head()` displays the first 5 rows

# First n Rows of Data

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

```
salary_data_org.head(n=2)
```

# Output

	First_Name	Last_Name	Grade	Location	ba	ms
0	Alan	Brown	GR1	DELHI	17990	16070.0
1	Agatha	Williams	GR2	MUMBAI	12390	6630.0

# Last n Rows of Data

Now we will see the last n rows of our data using `tail()`. By default, it displays last 5 rows.

```
salary_data_org.tail()
```

# Output

	First_Name	Last_Name	Grade	Location	ba	ms
7	John	Patil	GR2	MUMBAI	13500	10760.0
8	Sneha	Joshi	GR1	DELHI	20660	NaN
9	Gaurav	Singh	GR2	DELHI	13760	13220.0
10	Adela	Thomas	GR2	DELHI	13660	6840.0
11	Anup	Save	GR2	MUMBAI	11960	7880.0

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

```
salary_data_org.tail(n=2)
```

# Output

	First_Name	Last_Name	Grade	Location	ba	ms
10	Adela	Thomas	GR2	DELHI	13660	6840.0
11	Anup	Save	GR2	MUMBAI	11960	7880.0

# Summarising Data

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

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

```
salary_data_org.describe(include='all')
```

**describe()** is essentially applied to each column and it summarises all the columns.

It only provides summary of numeric variables until explicitly

programmed to include factor variables using `include = 'all'`.

	First_Name	Last_Name	Grade	Location	Da	Ms
count	12	12	12	12	12.0	11.0
unique	12	12	2	2	NaN	NaN
top	Rajesh	Kolte	GR2	MUMBAI	NaN	NaN
freq	1	1	7	7	NaN	NaN
mean	NaN	NaN	NaN	NaN	16154.58	11004.54
std	NaN	NaN	NaN	NaN	3739.37	3711.18
min	NaN	NaN	NaN	NaN	11960.0	6630.0
25%	NaN	NaN	NaN	NaN	13472.5	7360.0
50%	NaN	NaN	NaN	NaN	14270.0	10760.0
75%	NaN	NaN	NaN	NaN	19238.75	14225.0
max	NaN	NaN	NaN	NaN	23280.0	16070.0

# Change Variable Names – rename()

Our data is saved as an object named salary\_data.

Suppose we want to change the name of some variable (column) and its values. Let's rename the 'ba' variable to 'basic\_allowance' -

```
salary_data = salary_data_org.rename(columns={'ba':'basic_allowance'})  
list(salary_data)  
  
['First_Name', 'Last_Name', 'Grade', 'Location', 'basic_allowance',  
'ms']
```

- ❑ **rename()** uses name of the data object and assign **{'old name':'new name'}**.
- ❑ The result needs to be saved in an object because **rename()** doesn't modify the object directly.
- ❑ You can rename multiple column names like this:
- ❑ **salary\_data=salary\_data.rename(columns= {'ba':'basic\_allowance', 'ms':'management\_supplements'})**

# Derive a New Variable

Add a new variable to `salary_data` containing values as 5% of `ba`. We will use the `assign()` function to accomplish this:

```
salary_data=salary_data.assign(newvariable=salary_data['basic_allowance']
]*0.05)
salary_data.head(n=3)
```

# Output

	First_Name	Last_Name	Grade	Location	basic_allowance	ms
newvariable						
0	Alan	Brown	GR1	DELHI	17990	16070.0
1	Agatha	Williams	GR2	MUMBAI	12390	6630.0
2	Rajesh	Kolte	GR1	MUMBAI	19250	14960.0

# Recode a Categorical Variable - replace()

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.replace(to_replace=[ 'MUMBAI' , 'DELHI' ],value=[1,2],  
inplace=True)  
salary_data.head(n=3)
```

- ❑ **replace()** can be used to replace any part of a dataframe with another value.
- ❑ **to\_replace=** takes the value(s) to be replaced.
- ❑ **value=** takes the value(s) they need to be replaced with.
- ❑ **inplace= True** modifies the data directly i.e. you don't have to save the result back into salary\_data.

# Output

	First_Name	Last_Name	Grade	Location	basic_allowance	ms	newvariable
0	Alan	Brown	GR1	2	17990	16070.0	899.5
1	Agatha	Williams	GR2	1	12390	6630.0	619.5
2	Rajesh	Kolte	GR1	1	19250	14960.0	962.5



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

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

Categorise the employees on the basis of their ba in three categories, namely, Low, Medium and High. Converting a continuous variable to categorical is called binning. Pandas makes it efficient to bin variables through the `pd.cut()` function:

```
ba_labels = ['low','medium','high']
bins = [0,14000,19000,24000]
salary_data['Category'] = pd.cut(salary_data['basic_allowance'], bins,
labels=ba_labels)
salary_data.head()

# Output
```

	First_Name	Last_Name	Grade	Location	basic_allowance	ms	newvariable
0	Alan	Brown	GR1	2	17990	16070.0	899.50
1	Agatha	Williams	GR2	1	12390	6630.0	619.50
2	Rajesh	Kolte	GR1	1	19250	14960.0	962.50
3	Ameet	Mishra	GR2	2	14780	9300.0	739.00
4	Neha	Rao	GR1	1	19235	15200.0	961.75

	Category
0	medium
1	low
2	high
3	medium
4	high

# Create a Decile Object – quantile()

Interpret the deciles of ba

```
Decile_ba=salary_data[ 'basic_allowance' ].quantile(q=[0.1,0.2,0.3,0.4,0  
.5,0.6,0.7,0.8,0.9])  
Decile_ba
```

# Output

```
0.1    12490.0  
0.2    13412.0  
0.3    13548.0  
0.4    13700.0  
0.5    14270.0  
0.6    16706.0  
0.7    18861.5  
0.8    19247.0  
0.9    20519.0  
  
Name: basic_allowance, dtype: float64
```

## Interpretation:

- The first decile is 10%, implies that one-tenth of the '**basic\_allowance**' fall below or equal to 12490, and the remaining nine-tenth fall above 12490. 14270 is the median, thus half of the employees' '**basic\_allowance**' is

# Remove Columns from a Data Frame

Remove the column Last\_Name from salary\_data.

```
salary_data.drop('Last_Name',axis=1,inplace=True)  
salary_data.head()
```

# Output

	First_Name	Grade	Location	basic_allowance	ms	newvariable
<b>Category</b>						
0	Alan	GR1	2	17990	16070.0	899.50
medium						
1	Agatha	GR2	1	12390	6630.0	619.50
<b>low</b>						
2	Rajesh	GR1	1	19250	14960.0	962.50
high						
3	Ameet	GR2	2	14780	9300.0	739.00
medium						
4	Neha	GR1	1	19235	15200.0	961.75
high						

# 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 (i.e index 1,2 and 3)from salary\_data  
then we will write the following command:

```
salary_data.drop(salary_data.index[1:4], axis=0, inplace=True)  
salary_data.head(n=4)
```

# Output

	First_Name	Grade	Location	basic_allowance	ms	newvariable	Category
0	Alan	GR1	2	17990	16070.0	899.50	medium
4	Neha	GR1	1	19235	15200.0	961.75	high
5	Sagar	GR2	1	13390	6700.0	669.50	low
6	Aaron	GR1	1	23280	13490.0	1164.00	high

# Remove Rows from a Data Frame

Remove only rows which has Location as 'MUMBAI' i.e. 1

```
salary_data.drop(salary_data[salary_data.Location==1].index,  
inplace=True)  
salary_data
```

# Output

	First_Name	Grade	Location	basic_allowance	ms	newvariable
Category						
0	Alan	GR1	2	17990	16070.0	899.5
medium						
8	Sneha	GR1	2	20660	NaN	1033.0
high						
9	Gaurav	GR2	2	13760	13220.0	688.0
low						
10	Adela	GR2	2	13660	6840.0	683.0
low						

# Quick Recap

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

Dimensions and Variable Names

- `shape` returns the count of rows and columns
- `list(data)` returns variable names or column names.

Compact Internal Structure, Levels of Categorical Variable and Size of an Object

- `info()` returns many useful pieces of information like class of the object, data type of each column.
- `column.cat.categories()` returns the value of the levels of the object
- `memory_usage()` returns the size of an object in bytes.

Check the missing values (if any), First and Last n Rows

- `isnull()` returns a Boolean dataframe 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.

Summarise Data

- `describe()` summarises the data based on the type of variable it contains.

# Quick Recap

In this session, we learnt how to modify data in different ways.

## Change Variable Names

- `rename()` can be used to change names of single or multiple variables

## Derive a New Variable

- `assign()` adds a new variable in the dataframe

## Recoding

- `replace()` or `cut()` let you change the content of data

## Create a Decile Object

- `quantile()` is used for creating a decile object

## Remove Variables or Rows

- `drop()` is used to remove unwanted columns or rows from the data, either by specifying the variable names or using index

# Data Management in Python – Creating Subsets & Sorting Data

# Contents

1. The Need for Creating Subsets
2. Slicing Data Using Index
  - i. Row Subsetting
  - ii. Column Subsetting
  - iii. Row-Column Subsetting
3. Subsetting Data using Boolean Conditions
4. Introduction to Sorting
5. Sorting Data
  - i. Ascending Order
  - ii. Descending Order
  - iii. By Factor Variables
  - iv. By Multiple Variables; One Column with Characters / Factors and One with Numerals
  - v. By Multiple Variables and Multiple Ordering Levels

# 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



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.
- Python doesn't need any additional functions to slice its data,

# Indexing & Slicing in Pandas

- axis labelling function in Python helps identify observations and variables
- Python and NumPy indexing operators [ ] and attribute operator provide quick and easy access
- Pandas support 2 types of multi-indexing, **loc** and **iloc**.
- **loc** is used for label based indexing whereas **iloc** is primarily integer position based (from 0 to length -1 of the axis).

# Row Subsetting

- The **loc** function is used for label based indexing so it accepts labels and integers, provided that the integers are labels and not the index itself. However, note that python follows 0 index.

```
# Import data & Display rows from 5th to 10th  
  
import pandas as pd  
salary_data_org= pd.read_csv('basic_salary.csv')  
salary_data_org.loc[4:9]
```

# Output

	First_Name	Last_Name	Grade	Location	ba	ms
4	Neha	Rao	GR1	MUMBAI	19235	15200.0
5	Sagar	Chavan	GR2	MUMBAI	13390	6700.0
6	Aaron	Jones	GR1	MUMBAI	23280	13490.0
7	John	Patil	GR2	MUMBAI	13500	10760.0
8	Sneha	Joshi	GR1	DELHI	20660	NaN
9	Gaurav	Singh	GR2	DELHI	13760	13220.0

# Row Subsetting

```
# Display row numbers 1,3 and 5 only
```

```
salary_data_org.loc[[0,2,4]]
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
0	Alan	Brown	GR1	DELHI	17990	16070.0
2	Rajesh	Kolte	GR1	MUMBAI	19250	14960.0
4	Neha	Rao	GR1	MUMBAI	19235	15200.0



We use .loc[[ ]] when we want specific rows only.

# Column Subsetting

```
# Display columns 1 to 4  
salary_data_org.iloc[:,0:4] ←  
  
# Output
```

	First_Name	Last_Name	Grade	Location
0	Alan	Brown	GR1	DELHI
1	Agatha	Williams	GR2	MUMBAI
2	Rajesh	Kolte	GR1	MUMBAI
3	Ameet	Mishra	GR2	DELHI
4	Neha	Rao	GR1	MUMBAI
5	Sagar	Chavan	GR2	MUMBAI
6	Aaron	Jones	GR1	MUMBAI
7	John	Patil	GR2	MUMBAI
8	Sneha	Joshi	GR1	DELHI
9	Gaurav	Singh	GR2	DELHI
10	Adela	Thomas	GR2	DELHI
11	Anup	Save	GR2	MUMBAI

**iloc** helps use index by position. The row index is given first and the column index is added after a comma. Since a range of index is used here, the fact that all the rows have to be shown is denoted by the empty range.

# Row-Column Subsetting

```
# Display rows 1,5,8 and columns 1 and 2  
# With labels  
salary_data_org.loc[[0,4,7],['First_Name','Last_Name']]
```

# Output

	First_Name	Last_Name
0	Alan	Brown
4	Neha	Rao
7	John	Patil

# With Index

```
salary_data_org.iloc[[0,4,7],[0,1]]
```

# Output

	First_Name	Last_Name
0	Alan	Brown
4	Neha	Rao
7	John	Patil

# Subsetting Observations

```
# Create a subset with all details of employees of MUMBAI with ba  
# more than 15000  
  
salary_data_org[(salary_data_org.Location=='MUMBAI')  
&(salary_data_org.ba>15000)]
```

# Output

	First_Name	Last_Name	Grade	Location	ba	ms
2	Rajesh	Kolte	GR1	MUMBAI	19250	14960.0
4	Neha	Rao	GR1	MUMBAI	19235	15200.0
6	Aaron	Jones	GR1	MUMBAI	23280	13490.0

There is no limit on how many conditions may be combined to achieve the desired subset.

# Subsetting Observations

```
salary_data_org[(salary_data_org.Grade != 'GR1') &  
                 (salary_data_org.Location != "MUMBAI")]
```

# Output

	First_Name	Last_Name	Grade	Location	ba	ms
3	Ameet	Mishra	GR2	DELHI	14780	9300.0
9	Gaurav	Singh	GR2	DELHI	13760	13220.0
10	Adela	Thomas	GR2	DELHI	13660	6840.0

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

# 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
```

```
salary_data_org.loc[(salary_data_org.Grade=='GR1') &  
(salary_data_org.ba>15000), ['First_Name','Grade', 'Location']]
```

```
# Output
```

	First_Name	Grade	Location
0	Alan	GR1	DELHI
2	Rajesh	GR1	MUMBAI
4	Neha	GR1	MUMBAI
6	Aaron	GR1	MUMBAI
8	Sneha	GR1	DELHI

We're are combining the boolean conditions with **loc** function as we're trying to subset the dataframe by label positioning.

# Quick Recap

## Using loc, iloc

- Row Subsetting: By specifying the row labels using integers in [].
- Column Subsetting: By specifying the column labels in [].
- Row-Column Subsetting: By combining the above two methods.

## Using Boolean Conditions

- Subsetting observations: By giving conditions on columns using this function.
- Subsetting both observations and variables: By simply combining above two methods.

# Introduction

Sorting data is one of the common activities in preparing data for analysis

Sorting is storage of data in sorted order, it can be in ascending or descending order.

```
# Import Pandas and basic_salary data
```

```
import pandas as pd  
salary_data = pd.read_csv('basic_salary.csv')
```

# Ascending Data

```
# Sort salary_data by ba in Ascending order
```

```
ba_sorted_1=salary_data.sort_values(by=['ba']) ←  
ba_sorted_1.head()
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
11	Anup	Save	GR2	MUMBAI	11960	7880.0
1	Agatha	Williams	GR2	MUMBAI	12390	6630.0
5	Sagar	Chavan	GR2	MUMBAI	13390	6700.0
7	John	Patil	GR2	MUMBAI	13500	10760.0
10	Adela	Thomas	GR2	DELHI	13660	6840.0

By default,  
**sort\_values()**  
sorts data in  
ascending  
order

# Descending Order

```
# Sort salary_data by ba in Descending order
```

```
ba_sorted_2=salary_data.sort_values(by=['ba'], ascending = [0])  
ba_sorted_2.head()
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
6	Aaron	Jones	GR1	MUMBAI	23280	13490.0
8	Sneha	Joshi	GR1	DELHI	20660	NaN
2	Rajesh	Kolte	GR1	MUMBAI	19250	14960.0
4	Neha	Rao	GR1	MUMBAI	19235	15200.0
0	Alan	Brown	GR1	DELHI	17990	16070.0

Here, we are defining ascending as false by passing the Boolean argument 0.

# Sorting by Factor Variable

Sort data by column with characters / factors

```
# Sort salary_data by Grade
```

```
gr_sorted=salary_data.sort_values(by=['Grade'])  
gr_sorted.head()
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
0	Alan	Brown	GR1	DELHI	17990	16070.0
2	Rajesh	Kolte	GR1	MUMBAI	19250	14960.0
4	Neha	Rao	GR1	MUMBAI	19235	15200.0
6	Aaron	Jones	GR1	MUMBAI	23280	13490.0
8	Sneha	Joshi	GR1	DELHI	20660	NaN

Note that by default even with factor variables, **sort\_values()** sorts by ascending.

# 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.sort_values(by=['Grade'], ascending = [0])
gr_sorted.head()
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Agatha	Williams	GR2	MUMBAI	12390	6630.0
3	Ameet	Mishra	GR2	DELHI	14780	9300.0
5	Sagar	Chavan	GR2	MUMBAI	13390	6700.0
7	John	Patil	GR2	MUMBAI	13500	10760.0
9	Gaurav	Singh	GR2	DELHI	13760	13220.0

# 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.sort_values(by=['Grade','ba'])  
grba_sorted.head(10)
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
0	Alan	Brown	GR1	DELHI	17990	16070.0
4	Neha	Rao	GR1	MUMBAI	19235	15200.0
2	Rajesh	Kolte	GR1	MUMBAI	19250	14960.0
8	Sneha	Joshi	GR1	DELHI	20660	NaN
6	Aaron	Jones	GR1	MUMBAI	23280	13490.0
11	Anup	Save	GR2	MUMBAI	11960	7880.0
1	Agatha	Williams	GR2	MUMBAI	12390	6630.0
5	Sagar	Chavan	GR2	MUMBAI	13390	6700.0
7	John	Patil	GR2	MUMBAI	13500	10760.0
10	Adela	Thomas	GR2	DELHI	13660	6840.0

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
```

```
grms_sorted=salary_data.sort_values(by=['Grade','ms'],  
                                   ascending=[0,1])  
grms_sorted.head(10)
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Agatha	Williams	GR2	MUMBAI	12390	6630.0
5	Sagar	Chavan	GR2	MUMBAI	13390	6700.0
10	Adela	Thomas	GR2	DELHI	13660	6840.0
11	Anup	Save	GR2	MUMBAI	11960	7880.0
3	Ameet	Mishra	GR2	DELHI	14780	9300.0
7	John	Patil	GR2	MUMBAI	13500	10760.0
9	Gaurav	Singh	GR2	DELHI	13760	13220.0
6	Aaron	Jones	GR1	MUMBAI	23280	13490.0
2	Rajesh	Kolte	GR1	MUMBAI	19250	14960.0
4	Neha	Rao	GR1	MUMBAI	19235	15200.0

- Here, data is sorted by **Grade** in descending order and **ms** in ascending order.
- By default missing values in data are put last.
- You can put it first by adding an

# Quick Recap

In this session, we learnt sorting data using **sort\_values** in various ways.

Ascending/  
Descending Order

- **sort\_values** by default sorts in ascending order.
- For descending order: specify ascending=[**0**] for all variables.

Multiple Columns

- **sort\_values** allows us to sort by multiple columns of different type

Multiple Columns  
and Multiple  
Ordering Levels

- **sort\_values** provides flexibility to order by multiple columns with different ordering levels

# Data Management in Python – Merging / Appending & Aggregating Data

# Contents

1. Introduction to Merging
2. Types of Joins
  - i. Leftjoin
  - ii. Rightjoin
  - iii. Innerjoin
  - iv. Outerjoin
3. Appending Data Sets
4. Introduction to Aggregation
5. Aggregating Data Using
  - i. groupby() Function

# 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	Columns	Description
E-1004	Pr		Type
E-1005	Su	Employee_ID	Measurement
E-1007	R	First_Name	Possible values
E-1008	N	Last_Name	-
E-1009	Har	Basic_Salary	-

bonus\_data has information of only Bonus given to Employees.

→

Employee_ID	Bonus
E-1001	16070
E-1003	Columns
E-1004	Description
E-1006	Type
E-1008	Measurement
E-1010	Possible values

"Employee ID" is the common column in both datasets

# Merging

pandas provide various facilities for easily combining together Series, DataFrame objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

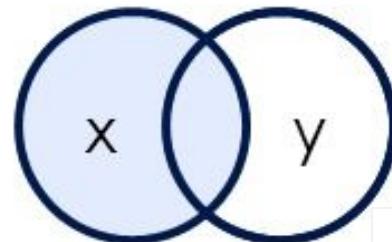
We can use the simple function `merge()` as our entry-point to merging data in pandas.

```
# Import sal_data and bonus_data  
import pandas as pd  
  
sal_data = pd.read_csv('sal_data.csv')  
bonus_data = pd.read_csv('bonus_data.csv')
```

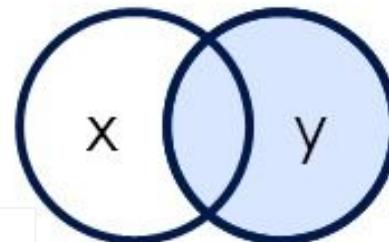
# Types of Joins

Consider  $\text{sal\_data} = x$  and  $\text{bonus\_data} = y$

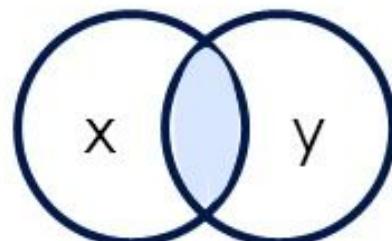
Left Join



Right Join

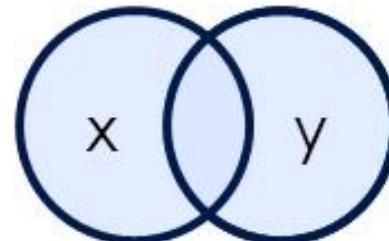


Inner Join



There are  
4 types of  
Joins

Outer Join



# Left Join

Left Join returns all rows from the left table, and any rows with matching keys from the right table.

```
# Display all the information(including bonus) of Employees from  
sal_data  
leftjoin=pd.merge(sal_data,bonus_data,how='left')  
leftjoin  
  
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
0	E-1001	Mahesh	Joshi	16860	16070.0
1	E-1002	Rajesh	Kolte	14960	NaN
2	E-1004	Priya	Jain	12670	13490.0
3	E-1005	Sneha	Joshi	15660	NaN
4	E-1007	Ram	Kanade	15850	NaN
5	E-1008	Nishi	Honrao	15950	15880.0
6	E-1009	Hameed	Singh	15120	NaN

**how=** is used to specify the type of join, in this case left.

# 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=pd.merge(sal_data,bonus_data,how='right')←  
rightjoin
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
0	E-1001	Mahesh	Joshi	16860.0	16070
1	E-1004	Priya	Jain	12670.0	13490
2	E-1008	Nishi	Honrao	15950.0	15880
3	E-1003	NaN	NaN	NaN	15200
4	E-1006	NaN	NaN	NaN	14200
5	E-1010	NaN	NaN	NaN	15120

To keep all rows from the data set y and only those from x that match, specify **how='right'**

# 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=pd.merge(sal_data,bonus_data)  
innerjoin
```

```
# Output
```

By default  
**merge** returns  
inner join.

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
0	E-1001	Mahesh	Joshi	16860	16070
1	E-1004	Priya	Jain	12670	13490
2	E-1008	Nishi	Honrao	15950	15880

# 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=pd.merge(sal_data,bonus_data,how='outer')  
outerjoin
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
0	E-1001	Mahesh	Joshi	16860.0	16070.0
1	E-1002	Rajesh	Kolte	14960.0	NaN
2	E-1004	Priya	Jain	12670.0	13490.0
3	E-1005	Sneha	Joshi	15660.0	NaN
4	E-1007	Ram	Kanade	15850.0	NaN
5	E-1008	Nishi	Honrao	15950.0	15880.0
6	E-1009	Hameed	Singh	15120.0	NaN
7	E-1003	NaN	NaN	NaN	15200.0
8	E-1006	NaN	NaN	NaN	14200.0
9	E-1010	NaN	NaN	NaN	15120.0

# Appending Data - Data Snapshot

basic\_salary - 1

Variables

basic\_salary - 1 data has 5 rows and 6 columns

basic\_salary - 2 data has 7 rows and 6 columns

First_Name	Last_Name	Grade	Location	ba	ms
Alan	Brown	GR1	DELHI	17990	16070
Agatha	Williams	GR2	MUMBAI	12300	5520
Rajesh	Kolte	GR1	MUMBAI	14500	6200
Ameet	Mishra	GR2	DELHI	13800	5800
Neha	Rao	GR1	MUMBAI	13200	5600

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

basic\_salary - 2

Variables

First_Name	Last_Name	Grade	Location	ba	ms
Sagar	Chavan	GR2	MUMBAI	20000	10000
Aaron	Jones	GR1	MUMBAI	18500	9200
John	Patil	GR2	MUMBAI	19000	9500
Sneha	Joshi	GR1	DELHI	20500	10200
Gaurav	Singh	GR2	DELHI	13760	6880
Adela	Thomas	GR2	DELHI	13660	6840
Anup	Save	GR2	MUMBAI	11960	7880

# Appending

- Append means adding cases/observations to a dataset.
- `concat()` appends data on one axis while computing the conditions on another.
- `frames=[Salary_1,Salary_2]` is the sequence that is passed as an object to `concat()` function.

# Appending Data Sets

```
# Import the data sets and append them using pd.concat()
```

```
Salary_1= pd.read_csv('basic_salary - 1.csv')
Salary_2= pd.read_csv('basic_salary - 2.csv')
frames=[Salary_1,Salary_2]
appendsalary=pd.concat(frames)
appendsalary
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
0	Alan	Brown	GR1	DELHI	17990	16070.0
1	Agatha	Williams	GR2	MUMBAI	12390	6630.0
2	Rajesh	Kolte	GR1	MUMBAI	19250	14960.0
3	Ameet	Mishra	GR2	DELHI	14780	9300.0
4	Neha	Rao	GR1	MUMBAI	19235	15200.0
0	Sagar	Chavan	GR2	MUMBAI	13390	6700.0
1	Aaron	Jones	GR1	MUMBAI	23280	13490.0
2	John	Patil	GR2	MUMBAI	13500	10760.0
3	Sneha	Joshi	GR1	DELHI	20660	NaN
4	Gaurav	Singh	GR2	DELHI	13760	13220.0
5	Adela	Thomas	GR2	DELHI	13660	6840.0
6	Anup	Save	GR2	MUMBAI	11960	7880.0

You can see that the original index of the dataframes has been maintained. Use argument **ignore\_index=True** E for 0 to n-1 index.

# Aggregating 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

# 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.

**groupby()** function in pandas carries out the process of taking numerous records and collapsing them into a single summary record.

```
# Import basic_salary data  
salary_data=pd.read_csv('basic_salary.csv')
```

# Aggregating Single Variable by Single Factor

```
# Calculate sum of variable 'ms' by variable 'Location'  
# In this example we are giving one variable and one factor
```

```
A=salary_data.groupby('Location')['ms'].sum()  
A
```

# Output

```
Location  
DELHI      45430.0  
MUMBAI     75620.0  
Name: ms, dtype: float64
```

- To aggregate, we need to create the **groupby** object first. In this case we are grouping **ms** by **Location**.
- ('**Location**') tells function to group according to the Location variables. This creates an instance where groupings for all variables are done.



groupby() by default ignores the missing data values.

# Aggregating Multiple Variables by Single Factor

```
# Calculate sum of variables 'ba' and 'ms' by variables 'Location'  
# In this example we are giving multiple variables and one factor
```

```
B=salary_data.groupby('Location')['ba','ms'].sum()
```

```
B
```

```
# Output
```

Location	ba	ms
DELHI	80850	45430.0
MUMBAI	113005	75620.0

To get the sum for both ba and ms, we're passing these labels to the index via [ ].

# Aggregating Multiple Variables by Multiple Factors

```
# Calculate sum of variable 'ms' and 'ba' by variables 'Location' and  
'Grade'  
# In this example we are giving two variables and two factors
```

```
C=salary_data.groupby(['Location', 'Grade'])['ba','ms'].sum()  
C
```

# Output

Location	Grade	ba	ms
DELHI	GR1	38650	16070.0
	GR2	42200	29360.0
MUMBAI	GR1	61765	43650.0
	GR2	51240	31970.0

Multiple factors are added as a dictionary, hence need to be contained within a [ ].

# Quick Recap

In this session, we learnt different ways of joining two data sets using `merge()` and `concat()` and aggregating data.. Here is the quick recap:

## Merging Data

4 types of joins:

- `left_join`
- `right join`
- `inner join`
- `outer join`

## Appending Data

`concat()` combines the vector, matrix or data frame by rows

## groupby()

- Creates a group as per the object passed, which can include multiple factors and multiple variables
- Allows you to run any function ranging from `sum()` to `mean()`, `median()` etc.
- Ignores NaN by default

# Data Management in Python – Handling Missing Values

Detecting, Excluding and Imputing NA's

# Contents

1. Introduction
2. Understanding Missing Data Mechanism
3. Replacing Missing Values with NA's while Importing
4. Detecting NA's
5. Excluding Missing Values from Analysis
6. Imputing Missing Values

# Introduction

- Missing values in data is a common phenomenon in real world problems and can create problems for simple and complicated analysis.
- You need to know the mechanism of missingness and how to treat them is a requirement to reduce the bias and to produce powerful models.
- Let's get familiar with the mechanisms 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 the variables in the dataset. For instance, if your equipment just flips out sometimes for no reason and doesn't record stuff, that will result in missing data that is MCAR.

- 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 instance, if men are more likely to tell you their weight than women, weight is MAR.

- Missing not at Random (MNAR):

MNAR happens when missingness is related to missing data in your dataset. For instance, single people are less likely to report marital status than married people.

# Missing Data Mechanism

- MNAR is 'non ignorable' because we have to include some model for why the data are missing and what the likely values are as we deal with the missing data.
- MCAR and MAR are both considered 'ignorable' because we don't have to include any information about the missing data itself when we deal with the missing data.

Let's go ahead with testing and dealing missing data

# 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	Variables					
	First_Name	Last_Name	Grade	Location	ba	ms
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 Python appears as NaN. NaN 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 Python.

```
# Import Data and check how Python treats missing data while importing
import pandas as pd
salary_data = pd.read_csv("basic_salary.csv")
```

# Output

	First_Name	Last_Name	Grade	Location	ba	ms
0	Alan	Brown	GR1	DELHI	17990	16070.0
1	Agatha	Williams	GR2	MUMBAI	12390	6630.0
2	Rajesh	Kolte	GR1	MUMBAI	19250	14960.0
3	Ameet	Mishra	GR2	DELHI	14780	9300.0
4	Neha	Rao	NaN	MUMBAI	19235	15200.0
5	Sagar	Chavan	GR2	MUMBAI	13390	6700.0
6	Aaron	Jones	GR1	MUMBAI	23280	13490.0
7	John	Patil	GR2	MUMBAI	13500	10760.0
8	Sneha	Joshi	GR1	DELHI	20660	NaN
9	Gaurav	Singh	GR2	DELHI	13760	13220.0
10	Adela	Thomas	GR2	DELHI	13660	6840.0
11	Anup	Save	GR2	MUMBAI	11960	7880.0

Note `read_csv()` replaces blank fields with NaN

# Detecting NA's

```
# Check whether our data has missing values or not
```

```
salary_data.isnull()
```

	First_Name	Last_Name	Grade	Location	ba	ms
0	False	False	False	False	False	False
1	False	False	False	False	False	False
2	False	False	False	False	False	False
3	False	False	False	False	False	False
4	False	False	True	False	False	False
5	False	False	False	False	False	False
6	False	False	False	False	False	False
7	False	False	False	False	False	False
8	False	False	False	False	False	True
9	False	False	False	False	False	False
10	False	False	False	False	False	False
11	False	False	False	False	False	False

`isnull()` returns logical matrix with the same dimensions as the data frame.

```
# Check total missing values
```

```
salary_data.isnull().sum()
```

First_Name	0
Last_Name	0
Grade	0
Location	0
ba	0
ms	1

- `isnull()` 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.
- `sum()` returns the total no. of missing values in the data by column.

# Detecting NA's

```
# Check Number of missing data per column
```

```
salary_data.info()
```

```
# Output
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12 entries, 0 to 11
Data columns (total 6 columns):
First_Name    12 non-null object
Last_Name     12 non-null object
Grade         11 non-null object
Location      12 non-null object
ba            12 non-null int64
ms            11 non-null float64
dtypes: float64(1), int64(1), object(4)
memory usage: 656.0+ bytes
```

Using `info()` we can check how many NaN's our data contains.

# Excluding Missing Values from Analysis

- Missing data is problematic because most statistical procedures require a value for each variable. When the data is incomplete, you have to decide how to deal with it.
- When Python encounters missing value, it attempts to perform the requested procedure and returns a missing (**NaN**) value as a result. One way of dealing with

```
import numpy as np
from statistics import *
x = [10, 30, 12, np.nan, 9]
mean(x)    ←
Nan
```

This output can be interpreted as: our vector contains missing value, so the requested statistic - the mean - is undefined for this data.

```
# We can calculate mean by dropping missing value like in the next example.
# remove missing value
```

```
np.nanmean(x)    ←
15.25
```

**nanmean** will remove all **NaN**'s while performing the requested procedure.

# 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. analysing only the cases with complete data.

```
# Case wise deletion
```

```
salary_data.dropna()
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
0	Alan	Brown	GR1	DELHI	17990	16070.0
1	Agatha	Williams	GR2	MUMBAI	12390	6630.0
2	Rajesh	Kolte	GR1	MUMBAI	19250	14960.0
3	Ameet	Mishra	GR2	DELHI	14780	9300.0
5	Sagar	Chavan	GR2	MUMBAI	13390	6700.0
6	Aaron	Jones	GR1	MUMBAI	23280	13490.0
7	John	Patil	GR2	MUMBAI	13500	10760.0
9	Gaurav	Singh	GR2	DELHI	13760	13220.0
10	Adela	Thomas	GR2	DELHI	13660	6840.0
11	Anup	Save	GR2	MUMBAI	11960	7880.0

- .dropna() is used for case deletion.
- Here, .dropna() 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									
Sn	Gender	Color	Weight	Shape	Camera	Ram	Processor	Internet	
1 M		3	3	4	3	3	3	3	
Observations	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	2 to 5		5	
	Shape	Shape			Categorical	3 to 5		5	
	Camera	Camera			Categorical	4 to 5		5	
	Ram	Ram			Categorical	5 to 5		5	
	Processor	Processor			Categorical	6 to 5		5	
	Internet	Internet			Categorical	7 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  
# Check number of missing values for each variable
```

```
consumer_pref = pd.read_csv("consumerpreference.csv")  
consumer_pref.isnull().sum()
```

# Output

Sn	0
Gender	0
Color	0
Weight	1
Shape	0
Camera	1
Ram	0
Processor	1
Internet	0
dtype: int64	

Our data has 3 missing values.

Here, we are calculating median of the values in variable 'Processor' using **median()** and replacing the **NaN** value with the median value. Now there are no missing values in variable '**Processor**'.

```
# Treating missing values in variable 'Processor'  
# Median imputation
```

```
consumer_pref['Processor'].fillna((consumer_pref['Processor'].median()),  
), inplace=True)  
consumer_pref['Processor'].isnull().sum()
```

0

# Imputing Missing Values

```
# Imputing missing value with forward fill
```

```
forward_fill = consumer_pref.fillna(method='ffill')  
forward_fill.head(10)
```

```
# Output
```

	Sn	Gender	Color	Weight	Shape	Camera	Ram	Processor	Internet
0	1	M	3	3.0	4	3.0	3	3.0	3
1	2	M	3	4.0	3	3.0	4	4.0	4
2	3	M	1	2.0	2	5.0	4	4.0	4
3	4	M	1	1.0	2	5.0	5	5.0	5
4	5	F	4	3.0	4	5.0	5	5.0	5
5	6	F	4	3.0	4	4.0	3	3.0	3
6	7	F	4	4.0	4	1.0	2	2.0	2
7	8	F	5	4.0	5	1.0	1	1.0	1
8	9	M	1	1.0	1	1.0	4	3.0	4
9	10	M	5	1.0	4	3.0	2	3.0	2

- **fillna()** allows to fill the NaN values with the value specified or by predefined methods.
- **method = ‘ffill’** allows to forward fill the NaN values i.e. it fills the missing value with the previous value.

# Imputing Missing Values

```
# Imputing missing value with backward fill
```

```
backward_fill = consumer_pref.fillna(method='bfill') ←  
backward_fill.head(10)
```

```
# Output
```

	Sn	Gender	Color	Weight	Shape	Camera	Ram	Processor	Internet
0	1	M	3	3.0	4	3.0	3	3.0	3
1	2	M	3	4.0	3	3.0	4	4.0	4
2	3	M	1	2.0	2	5.0	4	4.0	4
3	4	M	1	1.0	2	5.0	5	5.0	5
4	5	F	4	3.0	4	5.0	5	5.0	5
5	6	F	4	3.0	4	4.0	3	3.0	3
6	7	F	4	4.0	4	1.0	2	2.0	2
7	8	F	5	4.0	5	1.0	1	1.0	1
8	9	M	1	1.0	1	3.0	4	3.0	4
9	10	M	5	3.0	4	3.0	2	3.0	2

- ❑ **fillna()** allows to fill the NaN values with the value specified or by predefined methods.
- ❑ **method = 'bfill'** allows to backward fill the NaN values i.e. it fills the missing value with the next value.

# Quick Recap

In this session, we learnt how to deal with different types of missing values. Here is the quick recap:

## Replacing missing values

- Blank fields in numeric & character column are replaced with NaN while importing data.
- **fillna()** is used to replace missing values with various methods.

## Recoding values to missing

- **isnull()**: returns the logical matrix which indicates which elements are missing.
- **sum()**: using this function we can calculate the count of NA's per column
- **info()**: check number of NA's per variable

## Excluding missing values from analysis

- **dropna()**: 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

# Descriptive Statistics

# Contents

1. Data Measurement Scales
2. Measures of Central Tendency and Variation- Recap
3. Measures of Central Tendency in Python
4. Measures of Variation in Python
5. Measures of Skewness and Kurtosis-Recap
6. Measures of Skewness and Kurtosis in Python

# 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

It is a single value Most commonly used measures of central tendency are :

Mean	<p>Arithmetic Mean. Commonly known as Average.</p> <p>It is the sum of all values of the variable divided by the total number of values.</p>
Median	<p>Arrange the data in ascending order, Median is the middle value, if N is odd.</p> <p>If N is even, it is average of two middle values.</p>
Mode	<p>It is the most frequently occurring observation in a set of data.</p>



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.

# 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 of Variable	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

In addition to a measure of central tendency, it is desirable to have a measure of dispersion (variation) of data.

Measure of Dispersion :

- 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)

# 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 Python, we shall consider the below case as an example.

## Background

Data of 100 retailers in platinum segment of FMCG companies.

## 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

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

Observations

# Describing Variables in Python

```
#Importing Data
```

```
import pandas as pd  
retail_data =pd.read_csv('Retail_Data.csv')
```

```
#Checking the variable features using summary function
```

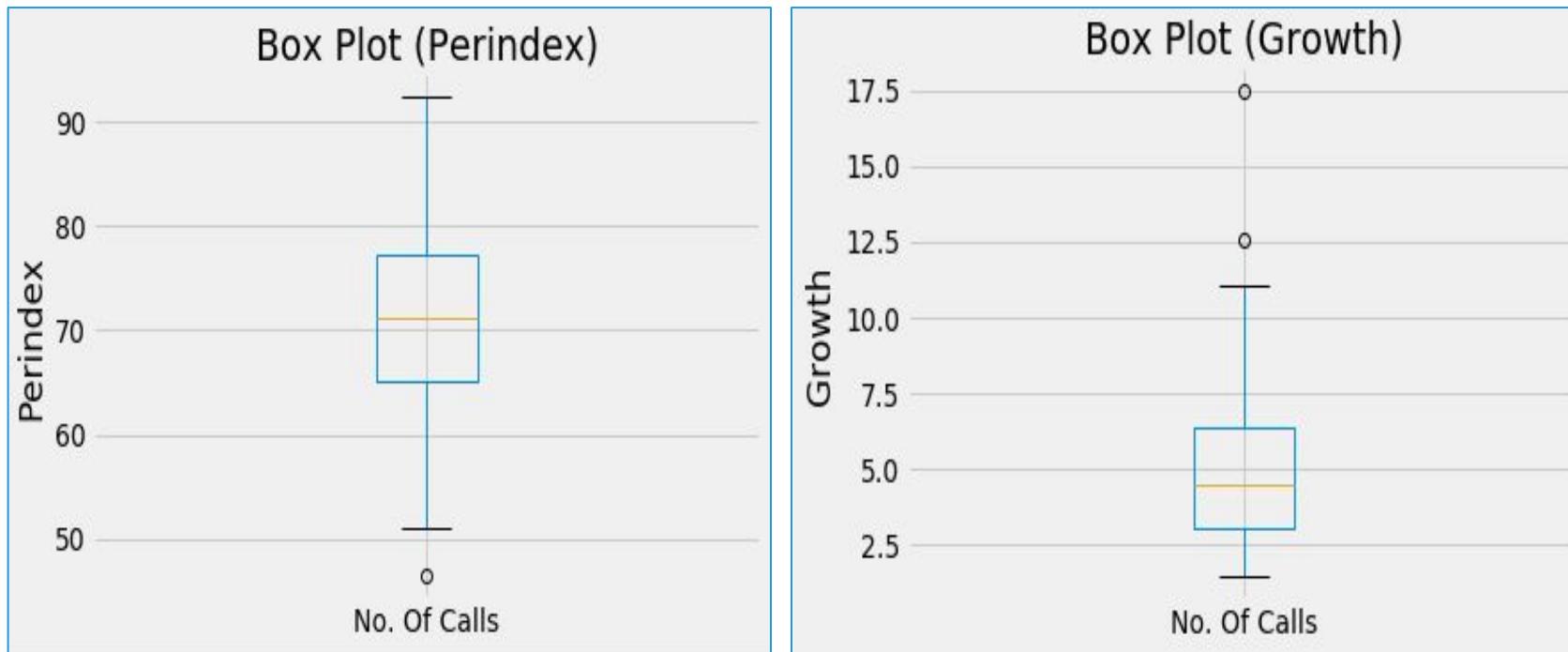
```
retail_data.describe(include = 'all')
```

```
# Output
```

	Retailer	Zone	Retailer_Age	Perindex	Growth	NPS_Category
min	1	NaN	NaN	46.53	1.47	NaN
25%	25.75	NaN	NaN	65.08	3.0575	NaN
std	29.01149	NaN	NaN	9.569232	2.620525	NaN
mean	50.5	NaN	NaN	70.49697	5.1528	NaN
50%	50.5	NaN	NaN	71.15	4.495	NaN
75%	75.25	NaN	NaN	77.175	6.34	NaN
count	100	100	100	99	100	100
max	100	NaN	NaN	92.49	17.5	NaN
unique	NaN	4	3	NaN	NaN	3
top	NaN	South	>5	NaN	NaN	Passive
freq	NaN	32	56	NaN	NaN	41

# Understanding Data through Visualisation

```
from matplotlib import pyplot as plt  
retail_data.Perindex.plot.box(label='No. Of Calls');plt.title('Box  
Plot (Perindex)');plt.ylabel('Perindex')  
retail_data.Growth.plot.box(label='No. Of Calls');plt.title('Box Plot  
(Growth)');plt.ylabel('Growth')
```



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  
How to plot a box plot is explained in detail in Data Visualisation Module

# Measures of Central Tendency in Python

```
# Mean for Perindex & Growth Variables
```

```
retail_data.Perindex.mean()
```

```
70.4969696969697
```

**mean()** in Python, gives mean of the variable. It excludes NAs by default

```
retail_data.Growth.mean()
```

```
5.152800000000002
```

```
# Median for Perindex & Growth Variables
```

```
retail_data.Perindex.median()
```

```
71.15
```

**median()** in Python, gives median of the variable.

```
retail_data.Growth.median()
```

```
4.495
```

So as we have seen, Perindex Variable is symmetric, hence its mean value is appropriate whereas for Growth Variable which is Positively Skewed, Median would be a better measure.

# Measures of Central Tendency in Python

```
# Import stats from scipy library
```

```
from scipy import stats
```

scipy is a python library used for advanced scientific operations.  
stats includes statistical operations

```
# Trimmed Mean
```

```
trimmed_mean_PI = stats.trim_mean(retail_data['Perindex'], 0.1)  
trimmed_mean_PI
```

```
70.76162500000001
```

Using 0.1 in the `trim_mean()`, excludes 10% observations from each side of the data from the mean

```
trimmed_mean_G = stats.trim_mean(retail_data['Growth'], 0.1)
```

```
trimmed_mean_G
```

```
4.825
```

```
# Mode
```

```
retail_data.Perindex.mode()
```

```
67.71
```

```
68.00
```

- In Python we can find mode directly by using `mode()` function.
- Here 67.71 and 68.00 has equal highest frequencies, hence 2 modes.

# Measures of Central Tendency in Python

```
# Measure of Central Tendency for Categorical Variable  
# Mode using Frequency Table
```

```
freq = retail_data['Zone'].value_counts()  
freq
```

# Output

South	32
West	28
North	25
East	15

**value\_counts()** in Python, gives the frequency of counts of the variable mentioned.

Here Mode is 32 as the frequency is highest for South Zone.

# Measures of Dispersion in Python

```
# Standard Deviation
```

```
retail_data['Perindex'].std()
```

```
9.56923188593669
```

**std()** in Python, gives standard deviation of the variable

```
# Variance
```

```
retail_data['Perindex'].var()
```

```
91.57019888682746
```

**var()** in Python, gives variance of the variable

```
# Coefficient of Variation
```

```
cv_PI = retail_data['Perindex'].std()/ retail_data.Perindex.mean()
```

```
cv_PI
```

```
0.135739620115161
```

We calculate CV manually by definition.

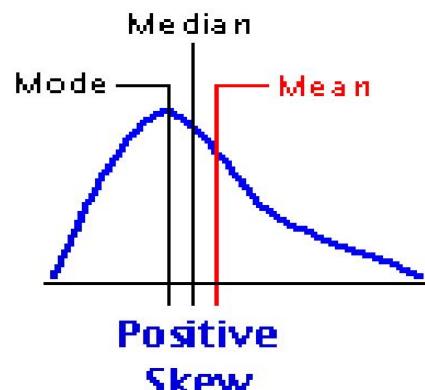
# Skewness

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

$$\text{Mode} < \text{Median} < \text{Mean}$$

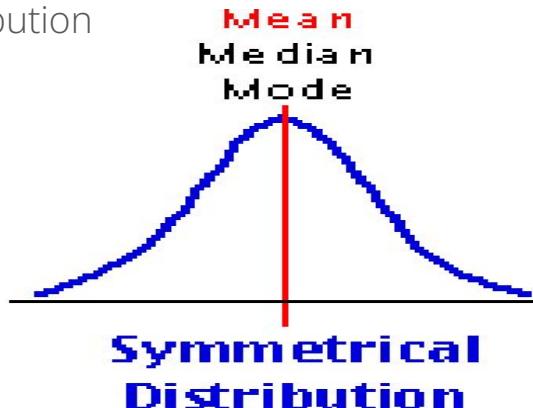


## Symmetric

- Both tails are equal
- Mass of the distribution is equally distributed

$$\text{Mean} = \text{Median} = \text{Mode}$$

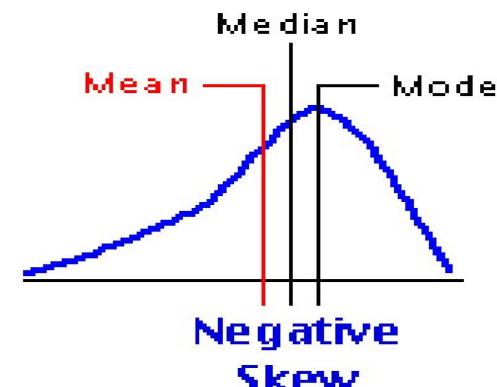
Normal Distribution is symmetric distribution



## Negatively Skewed

- Left Tail is longer
- Mass of the distribution is concentrated on the right

$$\text{Mean} < \text{Median} < \text{Mode}$$



# 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

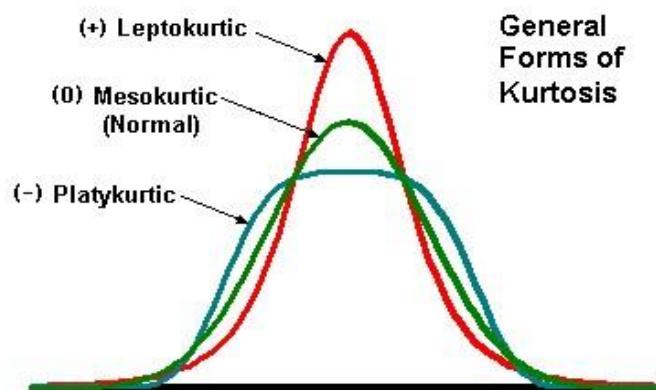
Normal distribution is termed as mesokurtic distribution

Leptokurtic

A leptokurtic distribution has a more acute peak. (positive kurtosis)

Platykurtic

A platykurtic distribution has a flatter peak. (negative kurtosis)



# Skewness and Kurtosis in Python

```
#Importing Data  
import pandas as pd  
retail_data =pd.read_csv('Retail_Data.csv')
```

We have already seen that Growth variable is Positively Skewed, so we'll find out skewness & kurtosis value for the same

```
from scipy import stats
```

Using package “scipy” in Python is the easiest way to find skewness and kurtosis

```
# Skewness
```

```
retail_data['Growth'].skew()
```

```
1.5912357812381297
```

▫ **skew()** gives skewness of the variable.

```
# Kurtosis
```

```
retail_data['Growth'].kurtosis()
```

```
4.283885801046328
```

▫ **kurtosis()** gives kurtosis of the variable.

# Quick Recap

In this session, we covered descriptive statistics using Python

## Measures of Central Tendency/Variation

- Mean, Median, Mode
- Standard Deviation, CV

## Measures of Skewness and Kurtosis

- Skewness and Kurtosis

## Working in Python

- Python codes for descriptive statistics

# Descriptive Statistics

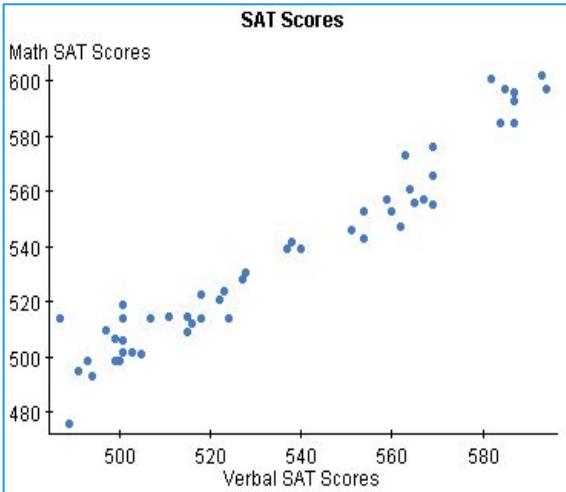
Bivariate Relationships in Python

# Contents

1. Interpreting Scatterplot
2. Describing Bivariate Relationship
3. Pearson's Coefficient of Correlation
4. Simple Linear Regression
5. Application Areas
6. Scatterplot in Python
7. Pearson's Coefficient of Correlation in Python
8. Simple Linear Regression in Python
9. Summarising Three Categorical Variables

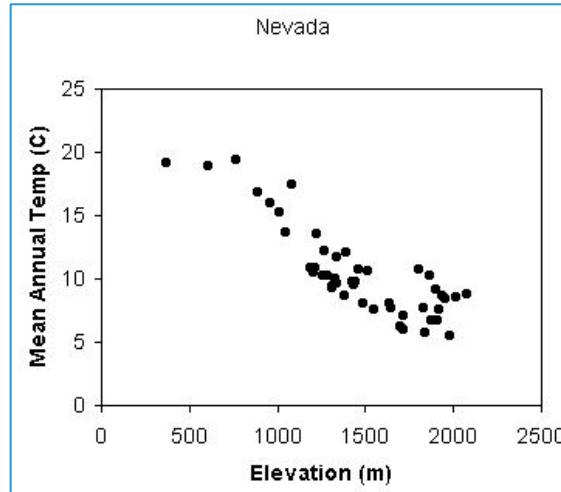
# Interpreting a Scatterplot

Positive Correlation



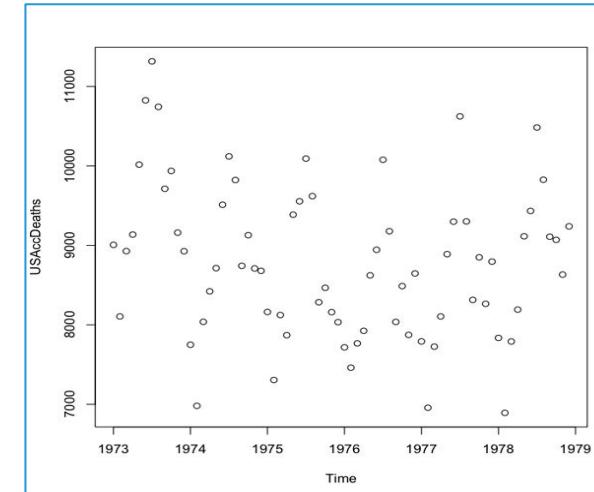
This is a positive sloping (upward) graph.  
As the value of one variable increases, the value of other variable also increases.

Negative Correlation



This is a negative sloping (downward) graph.  
As the value of one variable increases, the value of other variable tends to decrease.

No Correlation



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	
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

# 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 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

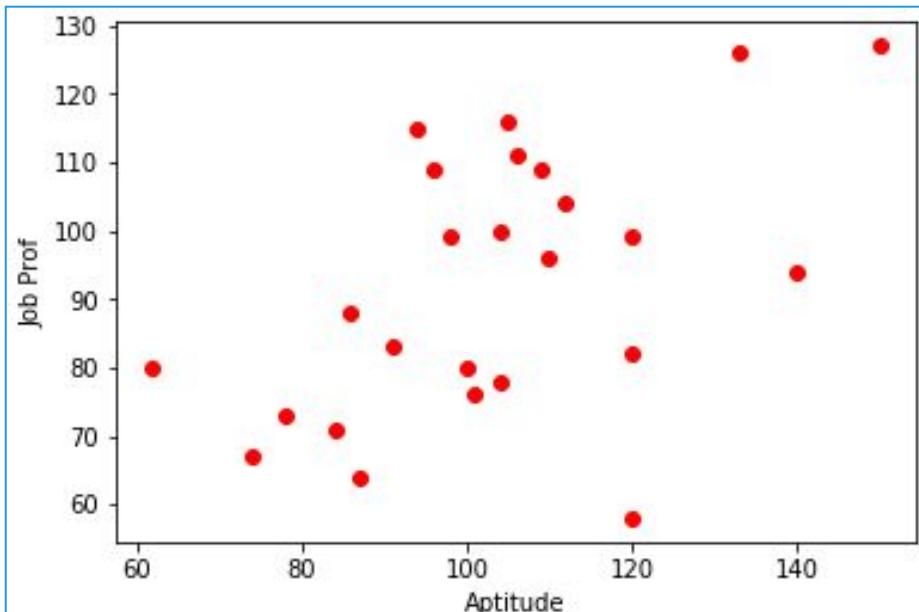
# Scatter Plot in Python

```
# Importing Data and necessary libraries
```

```
import pandas as pd  
import matplotlib.pyplot as plt  
job= pd.read_csv("Job_Proficiency.csv")
```

```
# Scatterplot
```

```
plt.scatter(job.aptitude,job.job_prof, color='red');  
plt.xlabel('Aptitude'); plt.ylabel('Job Prof')
```



- ❑ `plt.scatter()` gives a scatterplot of the two variables mentioned.
- ❑ `color=` provides color to the points.

# Pearson Correlation Coefficient in Python

```
# Correlation  
import numpy as np  
np.corrcoef(job.aptitude, job.job_prof) ←
```

```
array([[1.          , 0.51441069],  
       [0.51441069, 1.        ]])
```

**corrcoef()** 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 Python

```
# Simple Linear Regression
```

```
import statsmodels.formula.api as smf
modell= smf.ols("job_prof ~ aptitude", data = job).fit()
modell.summary()
```

OLS Regression Results						
Dep. Variable:	job_prof	R-squared:	0.265			
Model:	OLS	Adj. R-squared:	0.233			
Method:	Least Squares	F-statistic:	8.276			
Date:	Fri, 18 Oct 2019	Prob (F-statistic):	0.00852			
Time:	10:46:39	Log-Likelihood:	-105.28			
No. Observations:	25	AIC:	214.6			
Df Residuals:	23	BIC:	217.0			
Df Model:	1					
Covariance Type:	nonrobust					
-----						
	coef	std err	t	P> t	[0.025	0.975]
Intercept	41.3216	18.010	2.294	0.031	4.065	78.578
aptitude	0.4922	0.171	2.877	0.009	0.138	0.846
-----						
Omnibus:		1.110	Durbin-Watson:		2.409	
Prob(Omnibus):		0.574	Jarque-Bera (JB):		0.746	
Skew:		-0.416	Prob(JB):		0.689	
Kurtosis:		2.845	Cond. No.		557.	
-----						

- ❑ **ols()** gives us the linear regression model.
- ❑ **summary()** gives us the summary statistics

# Inferences : Simple Linear Regression

Dependent Variable : Job Proficiency

Independent Variable : Aptitude

Intercept	Aptitude
41. 3216	0.4922

$$\text{Equation : Job Proficiency} = 41.3216 + 0.4922 * \text{Aptitude}$$

Here Job Proficiency changes by 0.4992 units with a unit change in aptitude.

# Case Study - 2

To learn more Descriptive Statistics in Python, we shall consider the below case as an example.

## Background

Data of 100 retailers in platinum segment of an 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

Columns	Description	Type	Measurement	Possible values
Retailer	Retailer ID	numeric	-	-
Zone	Location of the retailer	character	North, East, West, 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

Observations

# 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 = pd.read_csv('Retail_Data.csv')
```

# Frequency Tables

```
Freq = pd.crosstab(index=retail_data["Zone"],  
columns=retail_data["NPS_Category"])  
Freq
```

NPS_Category	Detractor	Passive	Promoter
Zone			
East	5	9	1
North	5	13	7
South	7	9	16
West	6	10	12

**crosstab()** in Python, gives the frequency of counts of the two variables mentioned.

# Summarizing Two Categorical Variables

```
# Percentage Frequency Tables
```

```
Freq = pd.crosstab(index=retail_data["Zone"],  
columns=retail_data["NPS_Category"], normalize=True)  
Freq
```

NPS_Category	Detractor	Passive	Promoter
Zone			
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

By specifying **normalize=True** we can get percentage frequency

```
Freq = pd.crosstab(index=retail_data["Zone"],  
columns=retail_data["NPS_Category"], normalize='index')  
Freq
```

NPS_Category	Detractor	Passive	Promoter
Zone			
East	0.333333	0.600000	0.066667
North	0.200000	0.520000	0.280000
South	0.218750	0.281250	0.500000
West	0.214286	0.357143	0.428571

- By using **normalize = 'index'** we can get row wise distribution.
- Similarly for columns use **normalize = 'columns'**

# Summarizing Three Categorical Variables

```
# Three Way Frequency Table
```

```
table1 = pd.crosstab([retail_data.Zone, retail_data.NPS_Category],  
                     retail_data.Retailer_Age, margins = False)
```

```
table1
```

		2 to 5	<=2	>5
Retailer_Age	Zone			
Zone	NPS_Category			
East	Detractor	2	2	1
	Passive	3	3	3
	Promoter	0	0	1
North	Detractor	2	2	1
	Passive	6	1	6
	Promoter	0	1	6
South	Detractor	2	1	4
	Passive	4	2	3
	Promoter	3	3	10
West	Detractor	3	1	2
	Passive	1	1	8
	Promoter	1	0	11

**crosstab()** in Python, gives the frequency of counts of the three variables in one table itself.

# Quick Recap

In this session, we covered bivariate data analysis using Python.

## 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

## (v2) Data Visualisation – I

# Contents

- 1.** About Data Visualisation
2. Important Principles of Data Visualisation
3. Summarizing Data in Diagrams
  - i.** Bar Diagram
    - Simple Bar Chart
    - Sub Divided/Stacked Bar Chart
    - Multiple Bar Chart
  - ii.** Pie Chart
4. Summarizing Data in Diagrams using Python

# 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.

# 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 visualise the distribution of their 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			

# 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

# Diagrams in Python

```
#Importing the  
Libraries  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns
```

```
#Importing Data
```

```
telecom = pd.read_csv("telecom.csv")
```

```
#Aggregating Data
```

```
telecom1 = telecom.groupby('Age_Group')['Calls'].sum()  
telecom1
```

Age_Group	Calls
18-30	943187
30-45	798721
>45	128870

For plotting a bar chart in Python, it is important to aggregate the data using **groupby()** to get required vector/matrix]

# Simple Bar Chart in Python

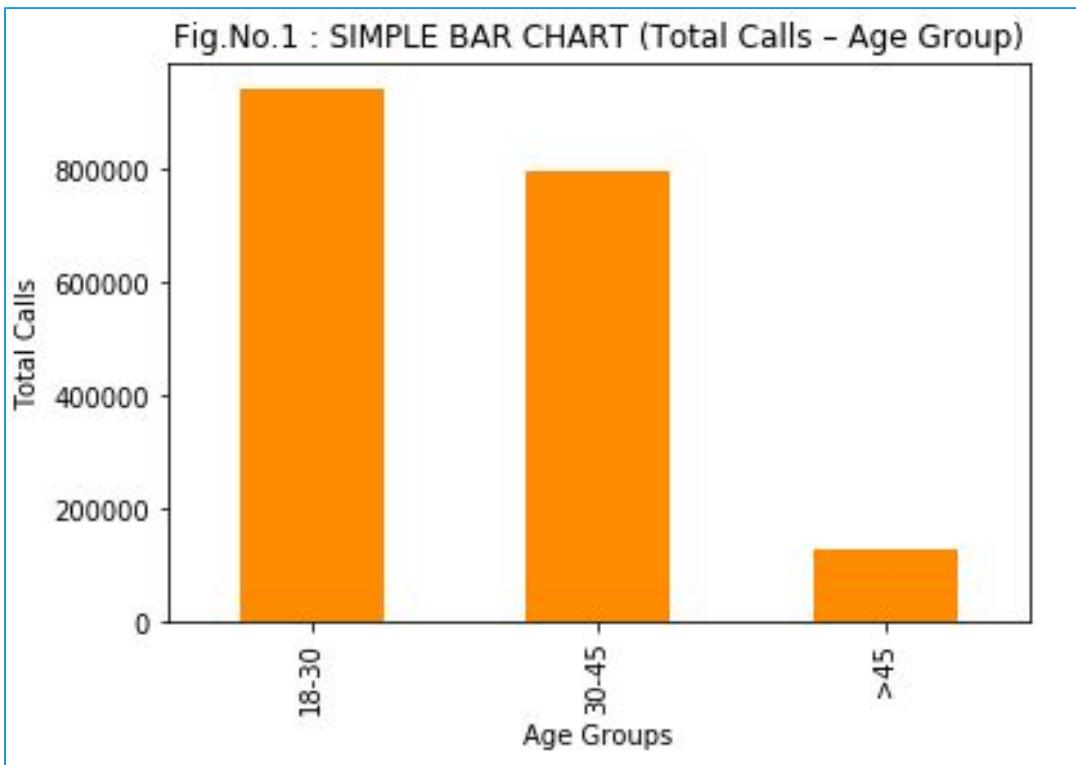
```
#Simple Bar Chart - Total Calls for different Age Groups
```

```
plt.figure(); telecom1.plot.bar(title='Fig.No.1 : SIMPLE BAR CHART (Total  
Calls - Age Group)', color='darkorange'); plt.xlabel('Age Groups');  
plt.ylabel('Total Calls')
```

- ❑ `plt.figure()` function is a convenient method to plot all columns with labels.
- ❑ `Plot.bar()` plots a bar chart. Can also be called by passing the argument `kind ='bar'` in plot.
- ❑ `title` is a string argument to give the plot a title.
- ❑ `color` argument specifies the plot colour. Accepts strings, hex numbers and colour code.
- ❑ `plt.xlabel` function/method to specify the x label.
- ❑ `plt.ylabel` function/method to specify the y label.

# Simple Bar Chart in Python

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 Python

```
# Simple Bar Chart - Mean Calls for different Age Groups
```

```
telecom2 = telecom.groupby('Age_Group')['Calls'].mean()  
telecom2
```

Age_Group	Calls
18-30	1882.608782
30-45	1866.170561
>45	1815.070423

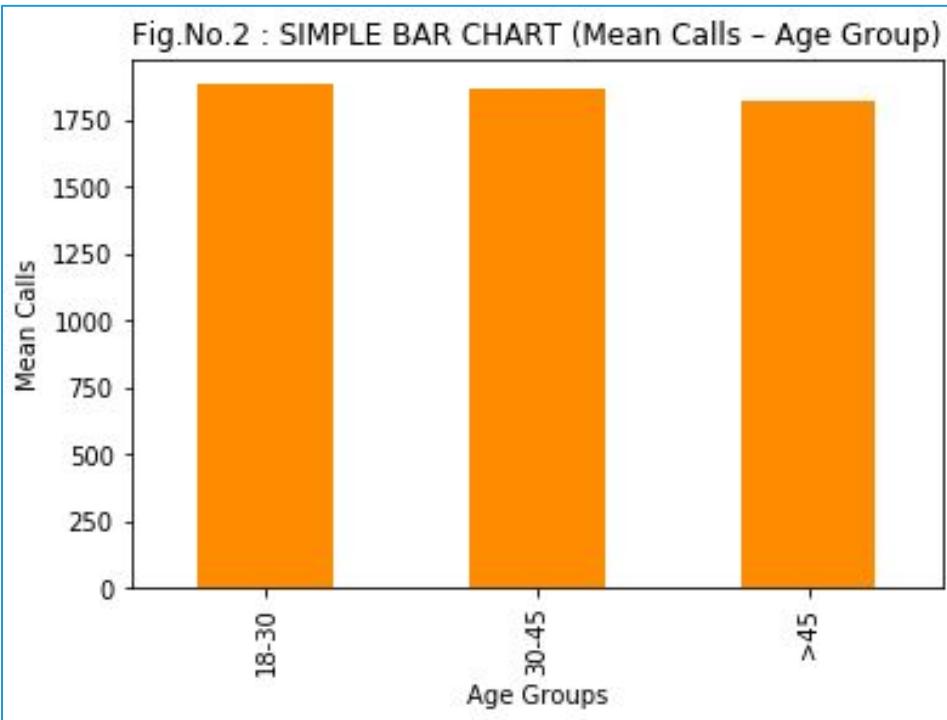
```
plt.figure(); telecom2.plot.bar(title='Fig.No.2 : SIMPLE BAR CHART (Mean  
Calls - Age Group)', color='darkorange'); plt.xlabel('Age Groups');  
plt.ylabel('Mean Calls')
```

**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 Python

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 Python

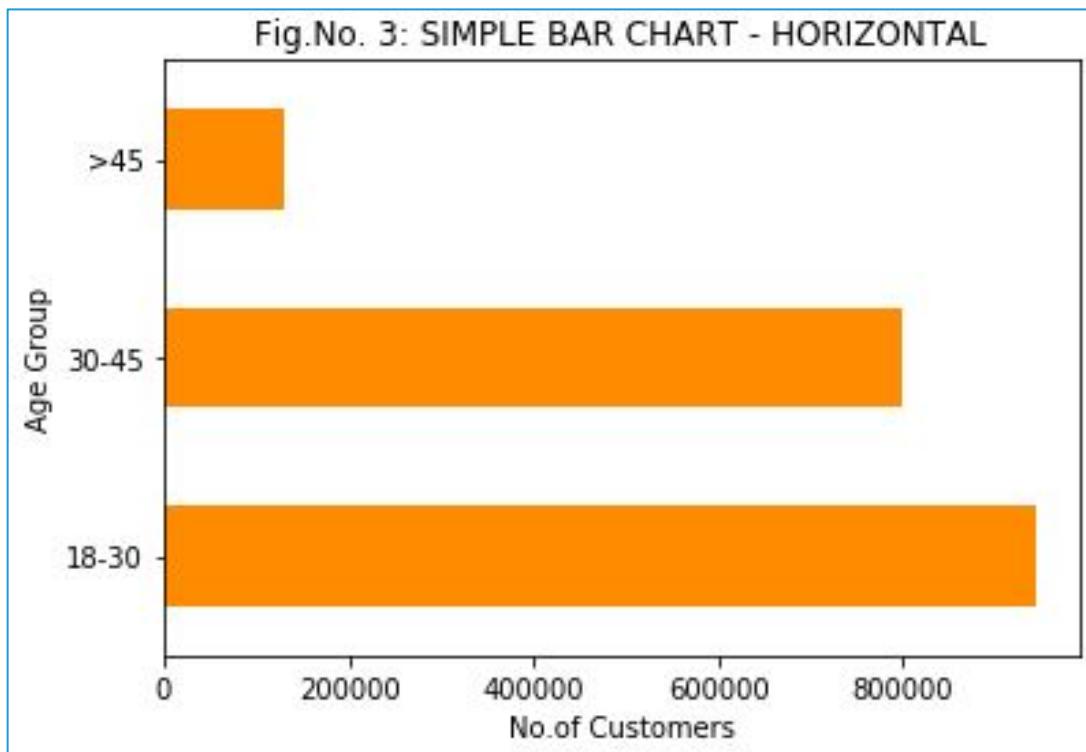
```
# Simple Bar Chart in Horizontal orientation
```

```
plt.figure(); telecom1.plot.bahr(title='Fig.No. 3: SIMPLE BAR CHART  
- HORIZONTAL', color='darkorange'); plt.xlabel('No.of Customers');  
plt.ylabel('Age Group')
```

- ❑ **bahr()** gives horizontal orientation to the bars.

# Simple Bar Chart in Python

This graph displays the number of customers across age group.



## Interpretation :

- This is horizontal view of figure 1. Both these graphs are describing the same thing 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 Python

```
# Stacked Bar Chart
```

```
telecom3=pd.pivot_table(telecom, index=['Age_Group'], columns=['Gender'],
values=['CustID'], aggfunc='count')
telecom3
```

Gender	CustID	
	F	M
Age_Group		
18-30	256	245
30-45	221	207
>45	32	39

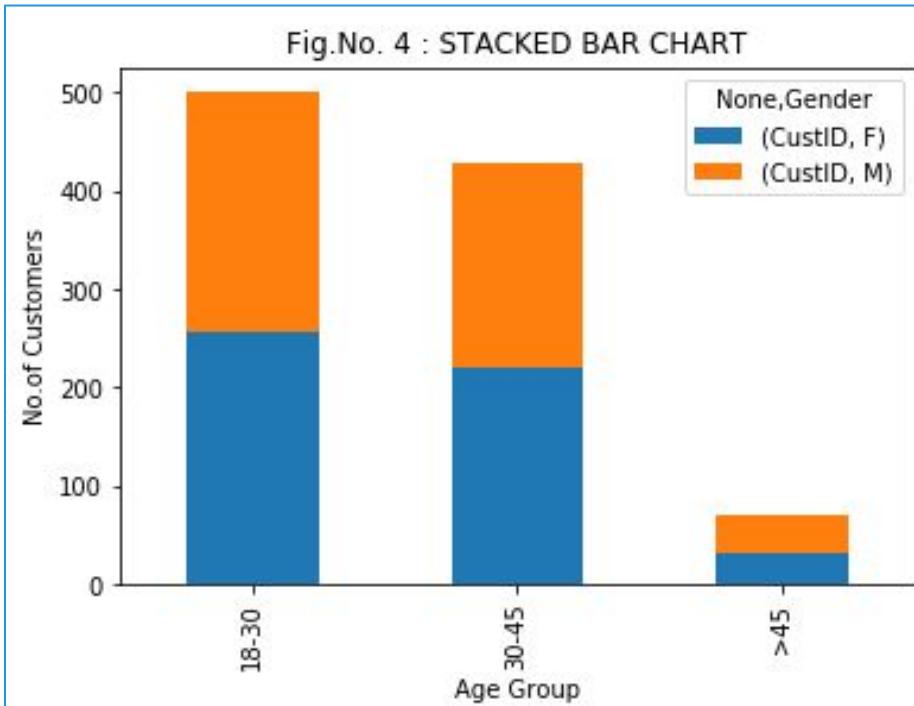
- ❑ **pivot\_table()** reshapes the data and aggregates according to function specified. Here, we are aggregating the number of calls made by gender and age group.
- ❑ **index** is the column or array to group by on the x axes (pivot table rows).
- ❑ **columns** is the column or array to group by on the y axes (pivot table column).
- ❑ **values** is the column to aggregate
- ❑ **aggfunc** specifies a function to aggregate by.

```
plt.figure(); telecom3.plot.bar(title='Fig.No. 4 : STACKED BAR CHART',
stacked=True); plt.xlabel('Age Group'); plt.ylabel('No.of Customers')
```

- ❑ **Stacked** returns a stacked chart. Default is False.

# Stacked Bar Chart in Python

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 Python

```
# Percentage Bar Chart
```

```
telecom4=telecom3.div(telecom3.sum(1).astype(float), axis=0)  
telecom4
```

	CustID	
Gender	F	M
Age_Group		
18-30	0.510978	0.489022
30-45	0.516355	0.483645
>45	0.450704	0.549296

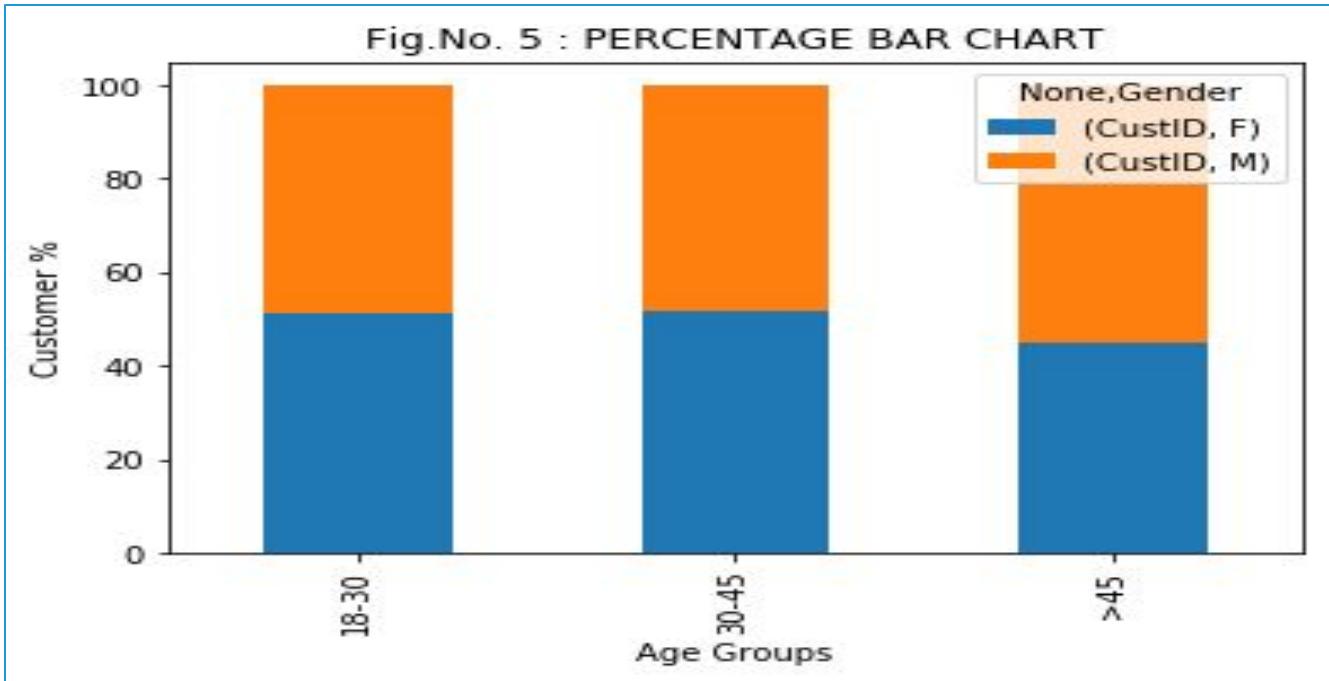
- ❑ **div()** creates percentage values by dividing the count data by column sum.

```
plt.figure();(telecom4*100).plot.bar(title='Fig.No. 5 : PERCENTAGE BAR  
CHART', stacked=True); plt.xlabel('Age Groups'); plt.ylabel('Customer %')
```

- ❑ **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 Python

```
# 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 Python

```
# Multiple Bar Chart
```

```
telecom5=pd.pivot_table(telecom, index=['Age_Group'], columns=['Gender'],
values=['Calls'], aggfunc='sum')
telecom5
```

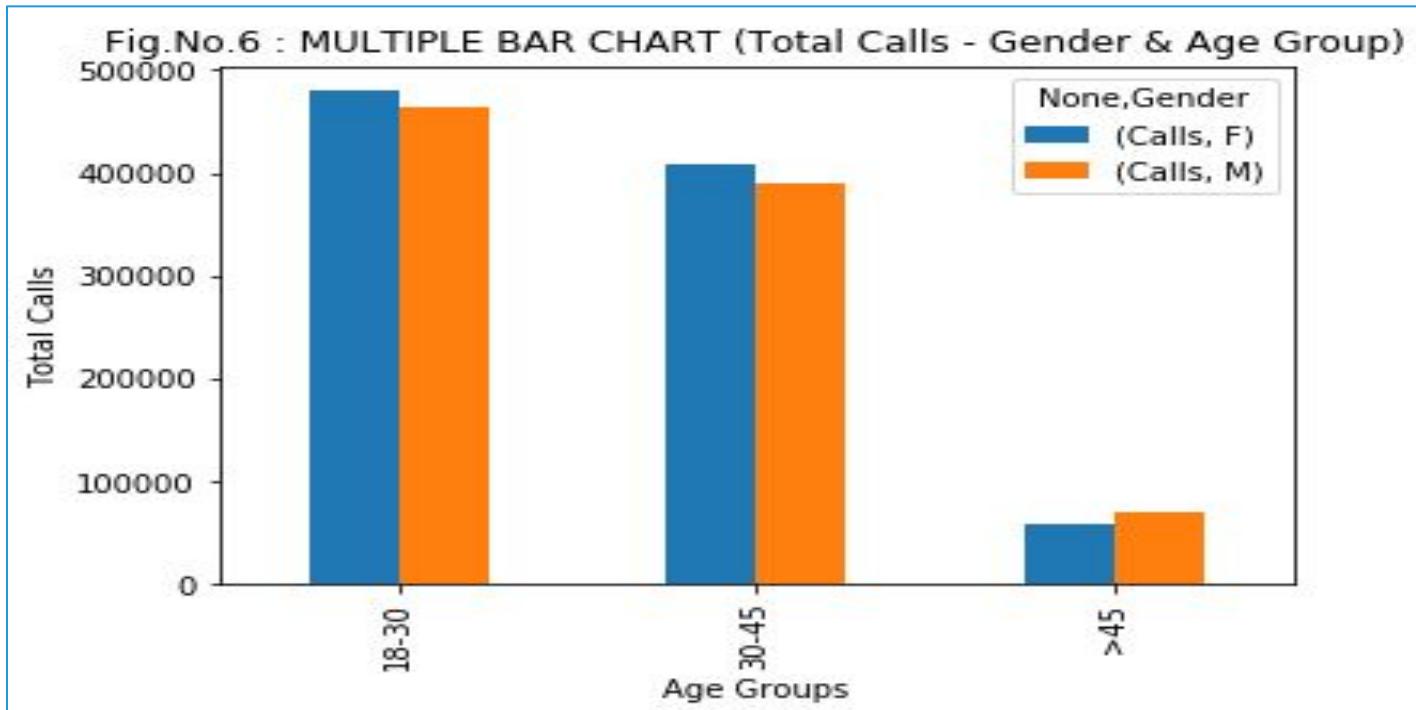
Gender	Calls	
	F	M
Age_Group		
18-30	480235	462952
30-45	408184	390537
>45	58310	70560

- ☐ **pivot\_table()** is used to cross tabulate the categories of more than one variables using another numeric variable which results in total of each category

```
plt.figure(); telecom5.plot.bar(title='Fig.No.6 : MULTIPLE BAR CHART
(Total Calls - Gender & Age Group)); plt.xlabel('Age Groups');
plt.ylabel('No. of Calls')
```

# Multiple Bar Chart in Python

```
# 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 in Python

```
# Pie Chart
```

```
telecom6 = telecom.groupby('Age_Group')['Calls'].sum()  
telecom6 = telecom6.div(telecom6.sum().astype(float)).round(2)*100  
telecom6
```

Age_Group	
18-30	50.0
30-45	43.0
>45	7.0

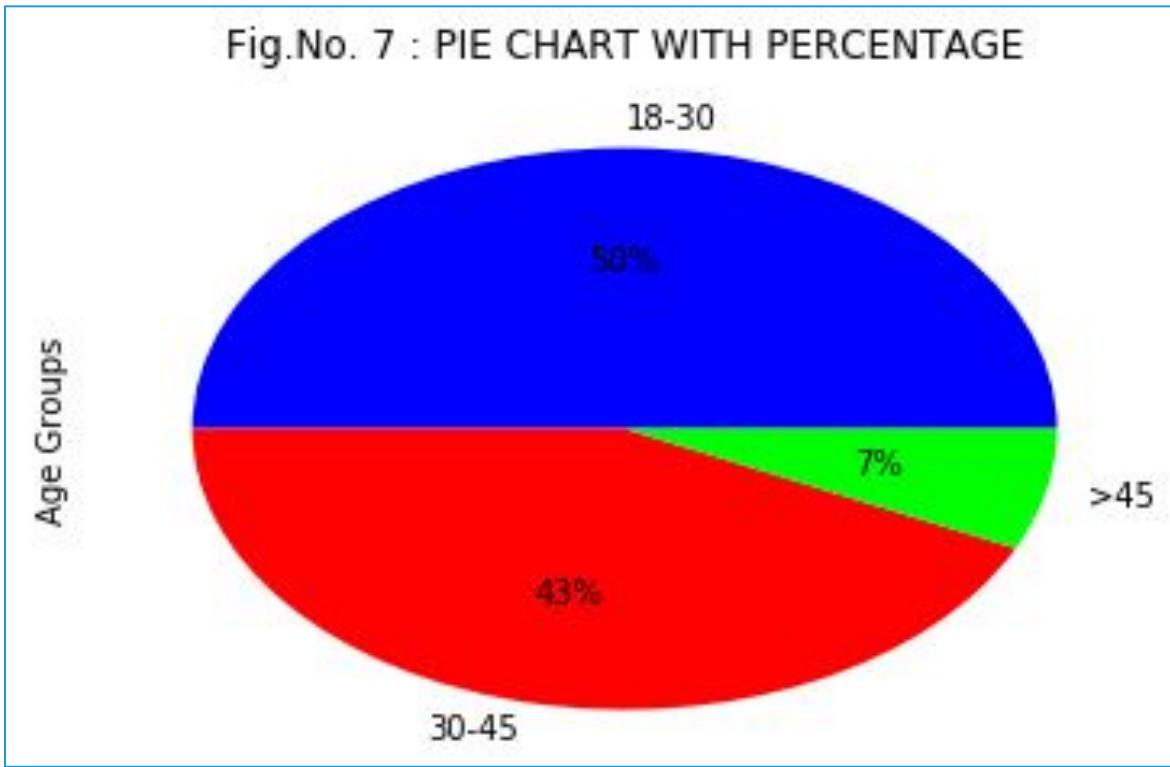
Here, we calculate the proportions for each category using **div()** function

```
telecom6.plot.pie(label='Age Groups', title = "Fig.No. 7 : PIE CHART WITH  
PERCENTAGE", colormap='brg', autopct='%1.0f%%')
```

- ❑ **pie()** Used with plot create a pie chart
- ❑ **autopct** is used to display percentage values
- ❑ **label=** provides a user defined label for the variable on X axis
- ❑ **title=** gives title of the plot
- ❑ **colormap=** can be used to input your choice of colors

# Pie Chart in Python

```
# Output of 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.

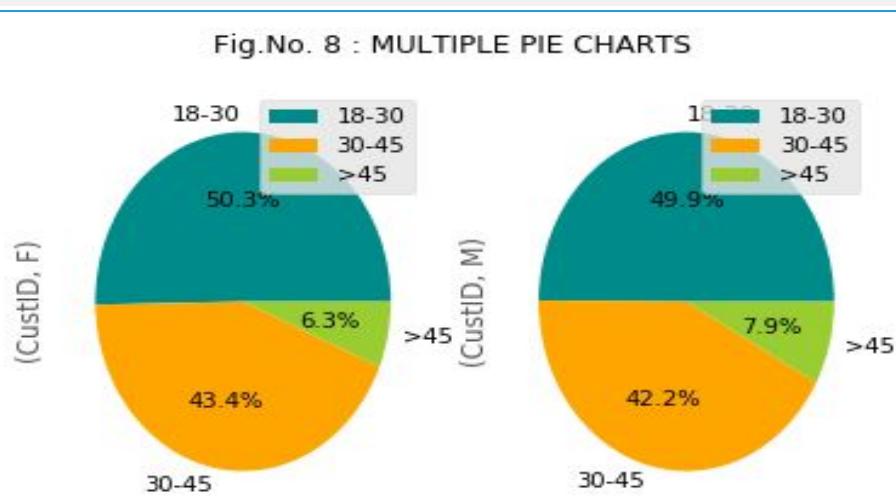
# Multiple Pie Chart in Python

```
#Pie Bar Chart - More than one
```

```
telecom7 = pd.pivot_table(telecom, index=['Age_Group'], columns=['Gender'],
values=['CustID'], aggfunc='count')
telecom7
```

	CustID	
Gender	F	M
Age_Group		
18-30	256	245
30-45	221	207
>45	32	39

```
plt.figure(); telecom7.plot.pie(title='Fig.No. 8 : MULTIPLE PIE CHARTS',
colors=['darkcyan','orange','yellowgreen'], autopct='%.1f%%', subplots=True)
```



- ❑ subplots() is default false, when 'True' plots multiple pie charts \

# Get an Edge!

## Important Principles of Data Visualisation

ACCENT is the principle of Data Visualization given for effective graphical display by D.A. Burn

Apprehension

Does the graph maximize the ability to correctly perceive relations among variables.?

Clarity

Is the graph able to visually distinguish all the elements of a graph and show the most important ones prominently?

Consistency

Are the elements, symbol shapes, and colors consistent with the previous graphs?

Efficiency

Is the graph able to portray complex relation in a simple and easy to interpret way?

Necessity

Is the graph more useful than the other ways to represent the data like a table/text?

Truthfulness

Are they accurately positioned and scaled such that the true values determinable by magnitude in terms of scale

# Contents

- 1. About Data Visualisation**
- 2. Important Principles of Data Visualisation**
- 3. Summarizing Data in Diagrams**
  - i. Bar Diagram**
    - Simple Bar Chart
    - Sub Divided/Stacked Bar Chart
    - Multiple Bar Chart
  - ii. Pie Chart**
- 4. Summarizing Data in Diagrams using Python**

# 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.

# 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 visualise the distribution of their 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	

# 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

# Diagrams in Python

```
#Importing the  
Libraries  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns
```

```
#Importing Data
```

```
telecom = pd.read_csv("telecom.csv")
```

```
#Aggregating Data
```

```
telecom1 = telecom.groupby('Age_Group')['Calls'].sum()  
telecom1
```

Age_Group	Calls
18-30	943187
30-45	798721
>45	128870

For plotting a bar chart in Python, it is important to aggregate the data using **groupby()** to get required vector/matrix

# Simple Bar Chart in Python

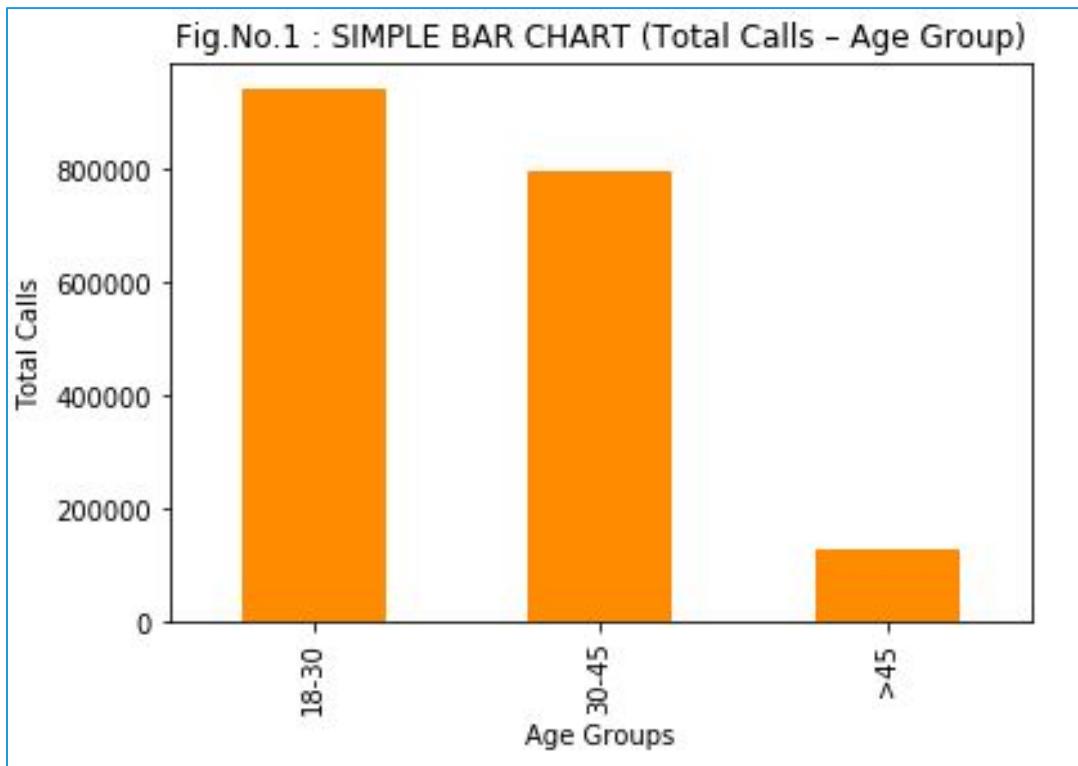
```
#Simple Bar Chart - Total Calls for different Age Groups
```

```
plt.figure(); telecom1.plot.bar(title='Fig.No.1 : SIMPLE BAR CHART (Total  
Calls - Age Group)', color='darkorange'); plt.xlabel('Age Groups');  
plt.ylabel('Total Calls')
```

- ❑ `plt.figure()` function is a convenient method to plot all columns with labels.
- ❑ `Plot.bar()` plots a bar chart. Can also be called by passing the argument `kind ='bar'` in plot.
- ❑ `title` is a string argument to give the plot a title.
- ❑ `color` argument specifies the plot colour. Accepts strings, hex numbers and colour code.
- ❑ `plt.xlabel` function/method to specify the x label.
- ❑ `plt.ylabel` function/method to specify the y label.

# Simple Bar Chart in Python

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 Python

```
# Simple Bar Chart - Mean Calls for different Age Groups
```

```
telecom2 = telecom.groupby('Age_Group')['Calls'].mean()  
telecom2
```

Age_Group	Calls
18-30	1882.608782
30-45	1866.170561
>45	1815.070423

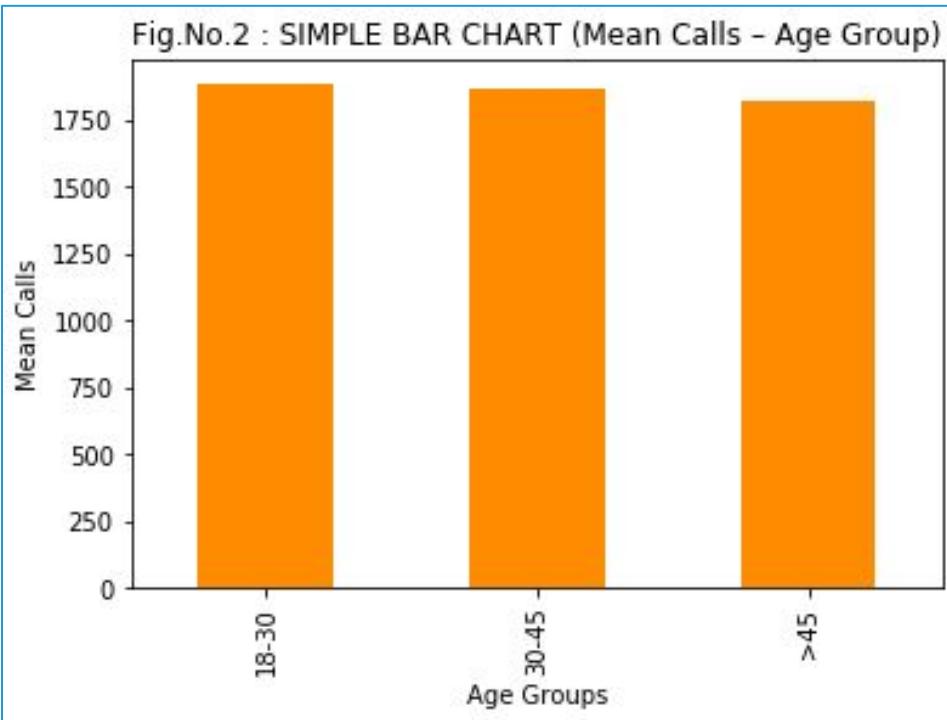
```
plt.figure(); telecom2.plot.bar(title='Fig.No.2 : SIMPLE BAR CHART (Mean  
Calls - Age Group)', color='darkorange'); plt.xlabel('Age Groups');  
plt.ylabel('Mean Calls')
```

**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 Python

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 Python

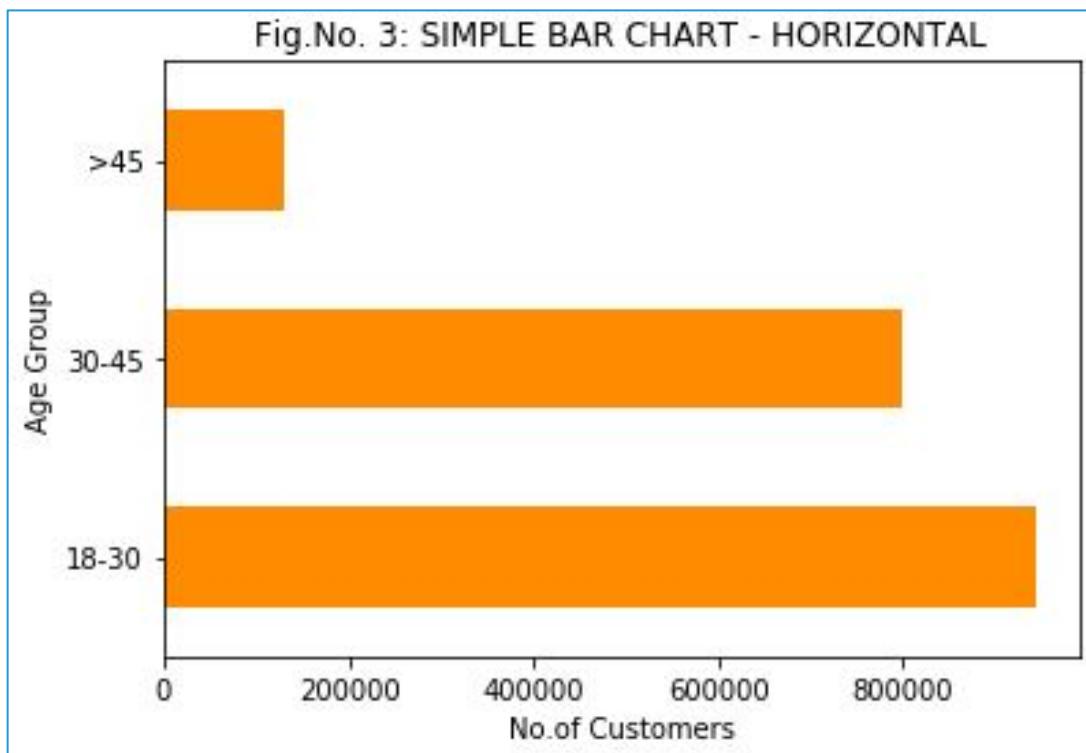
```
# Simple Bar Chart in Horizontal orientation
```

```
plt.figure(); telecom1.plot.barh(title='Fig.No. 3: SIMPLE BAR CHART  
- HORIZONTAL', color='darkorange'); plt.xlabel('No.of Customers');  
plt.ylabel('Age Group')
```

- ❑ **barh()** gives horizontal orientation to the bars.

# Simple Bar Chart in Python

This graph displays the number of customers across age group.



## Interpretation :

- This is horizontal view of figure 1. Both these graphs are describing the same thing 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 Python

```
# Stacked Bar Chart
```

```
telecom3=pd.pivot_table(telecom, index=['Age_Group'], columns=['Gender'],
values=['CustID'], aggfunc='count')
telecom3
```

Gender	CustID	
	F	M
Age_Group		
18-30	256	245
30-45	221	207
>45	32	39

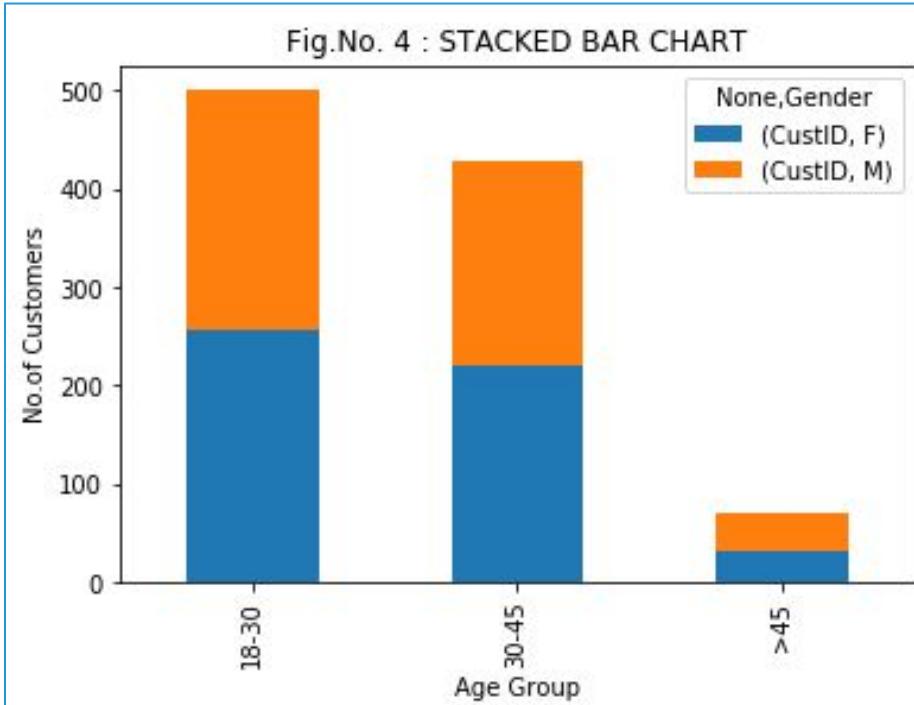
- ❑ **pivot\_table()** reshapes the data and aggregates according to function specified. Here, we are aggregating the number of calls made by gender and age group.
- ❑ **index** is the column or array to group by on the x axes (pivot table rows).
- ❑ **columns** is the column or array to group by on the y axes (pivot table column).
- ❑ **values** is the column to aggregate
- ❑ **aggfunc** specifies a function to aggregate by.

```
plt.figure(); telecom3.plot.bar(title='Fig.No. 4 : STACKED BAR CHART',
stacked=True); plt.xlabel('Age Group'); plt.ylabel('No.of Customers')
```

- ❑ **Stacked** returns a stacked chart. Default is False.

# Stacked Bar Chart in Python

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 Python

```
# Percentage Bar Chart
```

```
telecom4=telecom3.div(telecom3.sum(1).astype(float), axis=0)  
telecom4
```

	CustID	
Gender	F	M
Age_Group		
18-30	0.510978	0.489022
30-45	0.516355	0.483645
>45	0.450704	0.549296

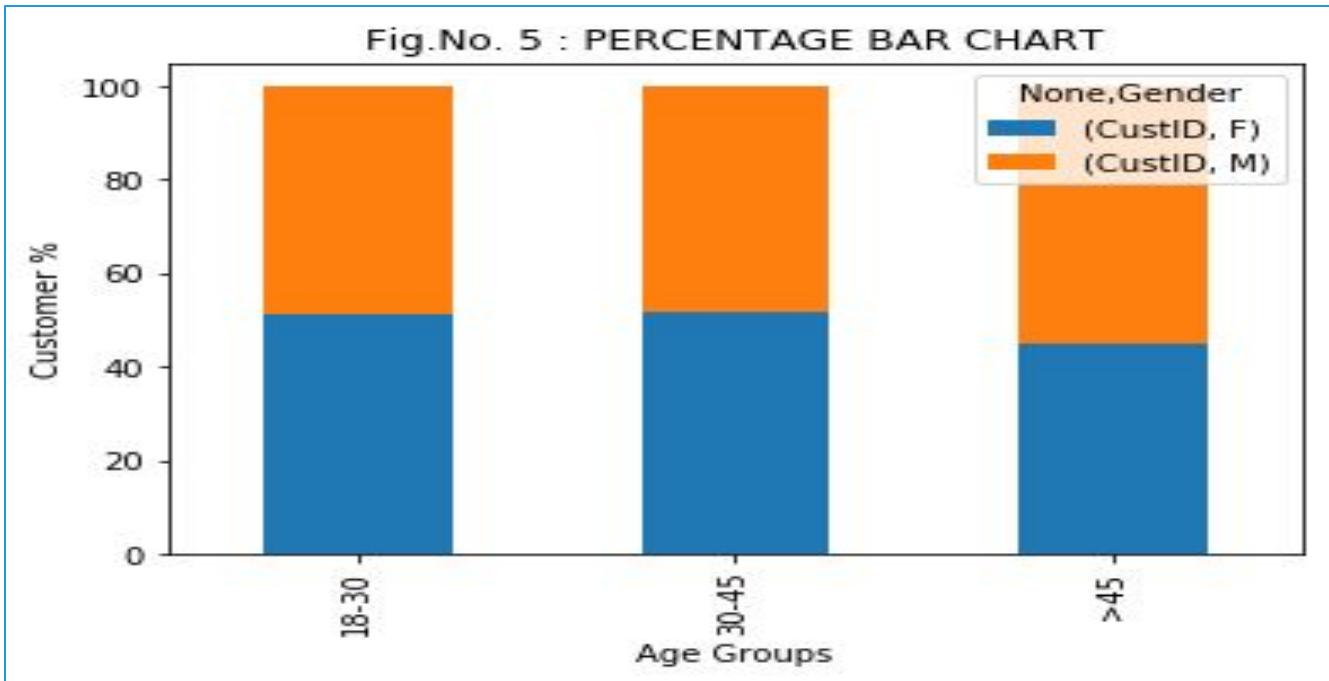
- ❑ **div()** creates percentage values by dividing the count data by column sum.

```
plt.figure();(telecom4*100).plot.bar(title='Fig.No. 5 : PERCENTAGE BAR  
CHART', stacked=True); plt.xlabel('Age Groups'); plt.ylabel('Customer %')
```

- ❑ **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 Python

```
# 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 Python

```
# Multiple Bar Chart
```

```
telecom5=pd.pivot_table(telecom, index=['Age_Group'], columns=['Gender'],
values=['Calls'], aggfunc='sum')
telecom5
```

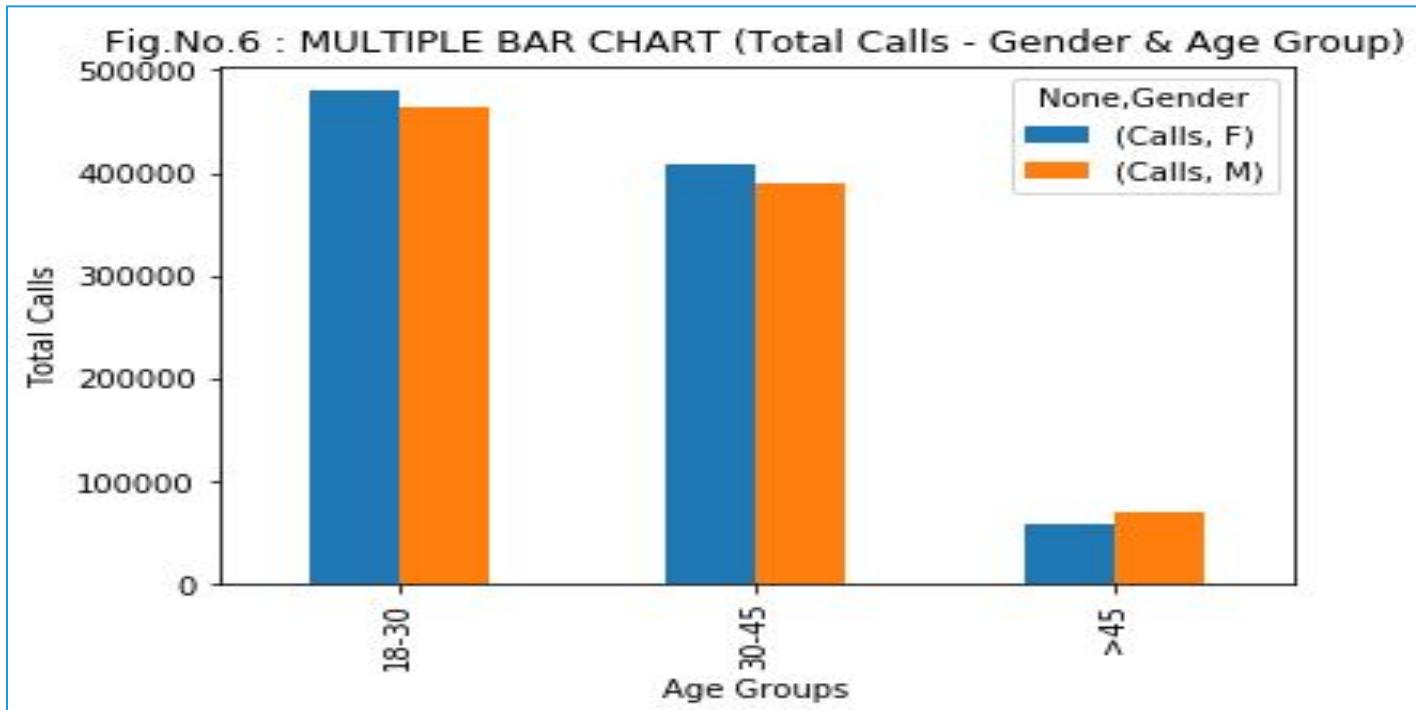
Gender	Calls	
	F	M
Age_Group		
18-30	480235	462952
30-45	408184	390537
>45	58310	70560

- ❑ **pivot\_table()** is used to cross tabulate the categories of more than one variables using another numeric variable which results in total of each category

```
plt.figure(); telecom5.plot.bar(title='Fig.No.6 : MULTIPLE BAR CHART
(Total Calls - Gender & Age Group)); plt.xlabel('Age Groups');
plt.ylabel('No. of Calls')
```

# Multiple Bar Chart in Python

```
# 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 in Python

```
# Pie Chart
```

```
telecom6 = telecom.groupby('Age_Group')['Calls'].sum()  
telecom6 = telecom6.div(telecom6.sum()).astype(float)).round(2)*100  
telecom6
```

Age_Group	
18-30	50.0
30-45	43.0
>45	7.0

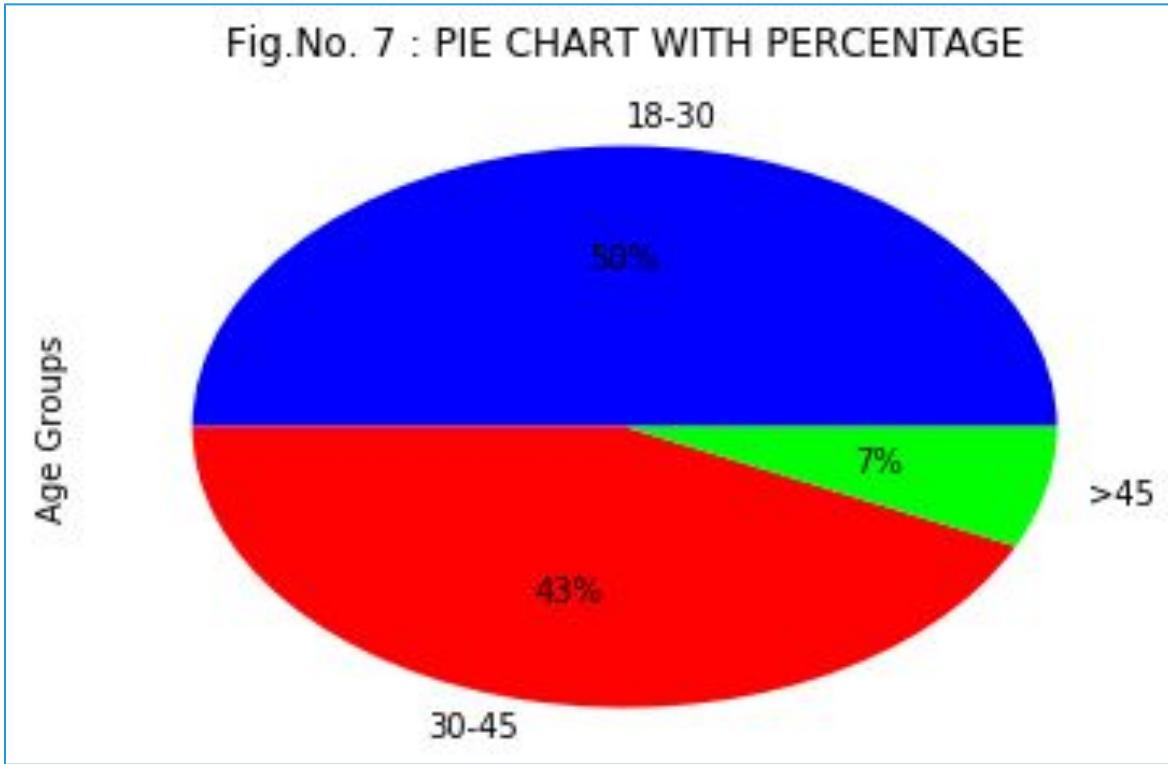
Here, we calculate the proportions for each category using **div()** function

```
telecom6.plot.pie(label='Age Groups', title = "Fig.No. 7 : PIE CHART WITH  
PERCENTAGE", colormap='brg', autopct='%1.0f%')
```

- ❑ **pie()** Used with plot create a pie chart
- ❑ **autopct** is used to display percentage values
- ❑ **label=** provides a user defined label for the variable on X axis
- ❑ **title=** gives title of the plot
- ❑ **colormap=** can be used to input your choice of colors

# Pie Chart in Python

```
# Output of 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.

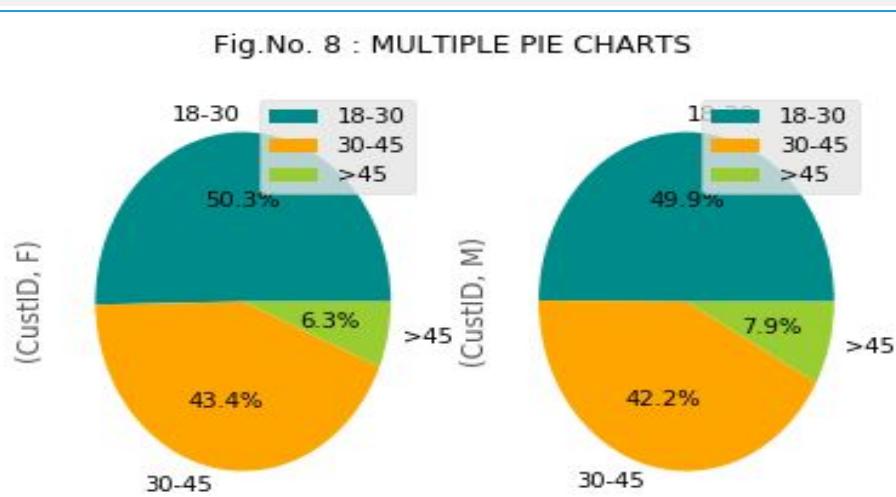
# Multiple Pie Chart in Python

```
#Pie Bar Chart - More than one
```

```
telecom7 = pd.pivot_table(telecom, index=['Age_Group'], columns=['Gender'],
values=['CustID'], aggfunc='count')
telecom7
```

	CustID	
Gender	F	M
Age_Group		
18-30	256	245
30-45	221	207
>45	32	39

```
plt.figure(); telecom7.plot.pie(title='Fig.No. 8 : MULTIPLE PIE CHARTS',
colors=['darkcyan','orange','yellowgreen'], autopct='%.1f%%', subplots=True)
```



- ❑ **subplots()** is default false, when 'True' plots multiple pie charts

# Get an Edge!

## Important Principles of Data Visualisation

ACCENT is the principle of Data Visualization given for effective graphical display by D.A. Burn

Apprehension

Does the graph maximize the ability to correctly perceive relations among variables.?

Clarity

Is the graph able to visually distinguish all the elements of a graph and show the most important ones prominently?

Consistency

Are the elements, symbol shapes, and colors consistent with the previous graphs?

Efficiency

Is the graph able to portray complex relation in a simple and easy to interpret way?

Necessity

Is the graph more useful than the other ways to represent the data like a table/text?

Truthfulness

Are they accurately positioned and scaled such that the true values determinable by magnitude in terms of scale

# Quick Recap

In this session, we learnt data visualisation using basic graphs

Chart Types and  
Functions in Python

- Bar Diagrams - `plot.bar()`
- Pie Chart - `plot.pie()`

# Other Basic Graphs with Python

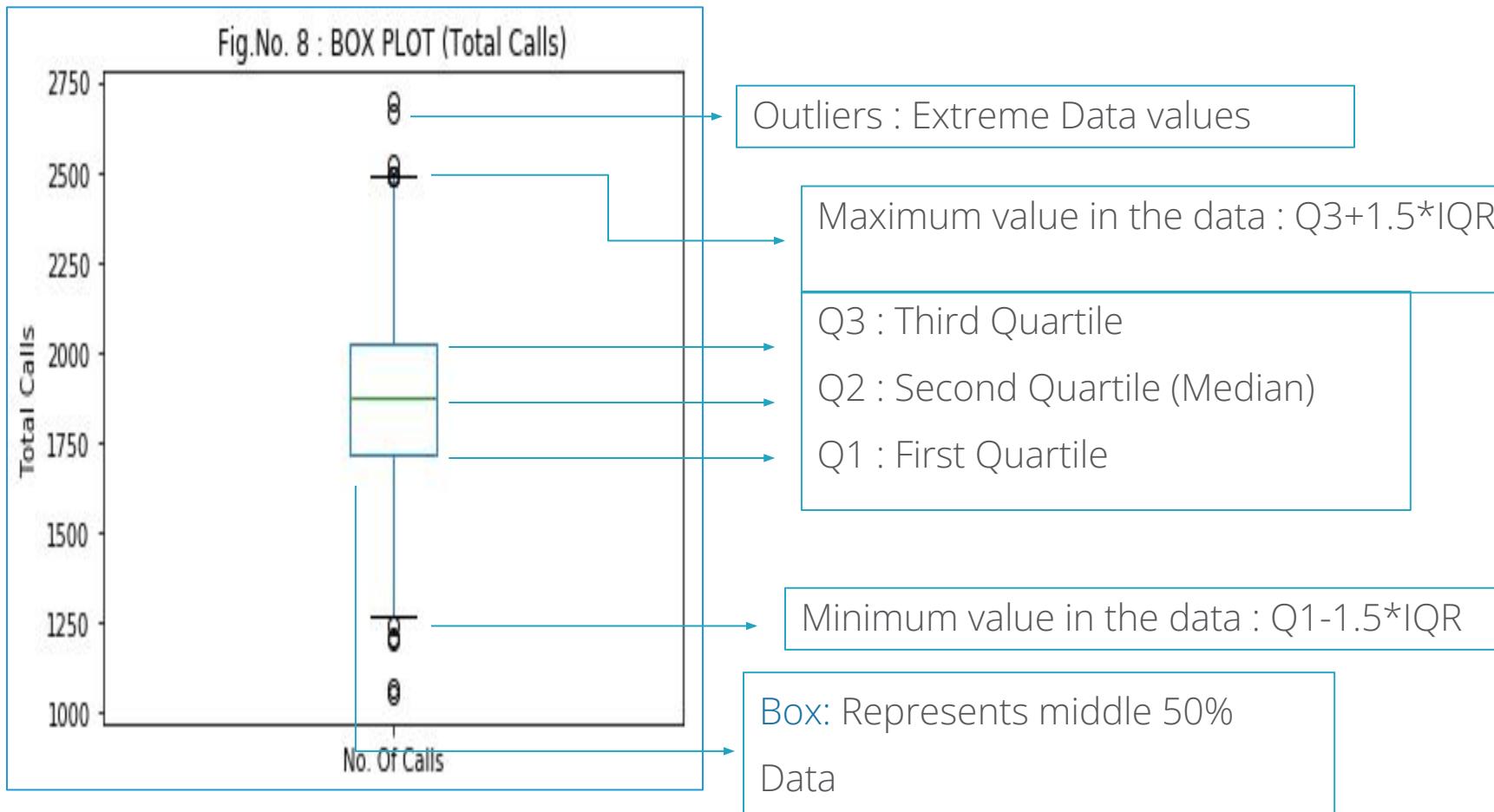
# Contents

## 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 Python

# Box – Whisker Plot



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

# 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 visualise the distribution of their customer database  
To see how the Calls and Amount are distributed across customers

## Sample Size

1000

\*

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

# 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			

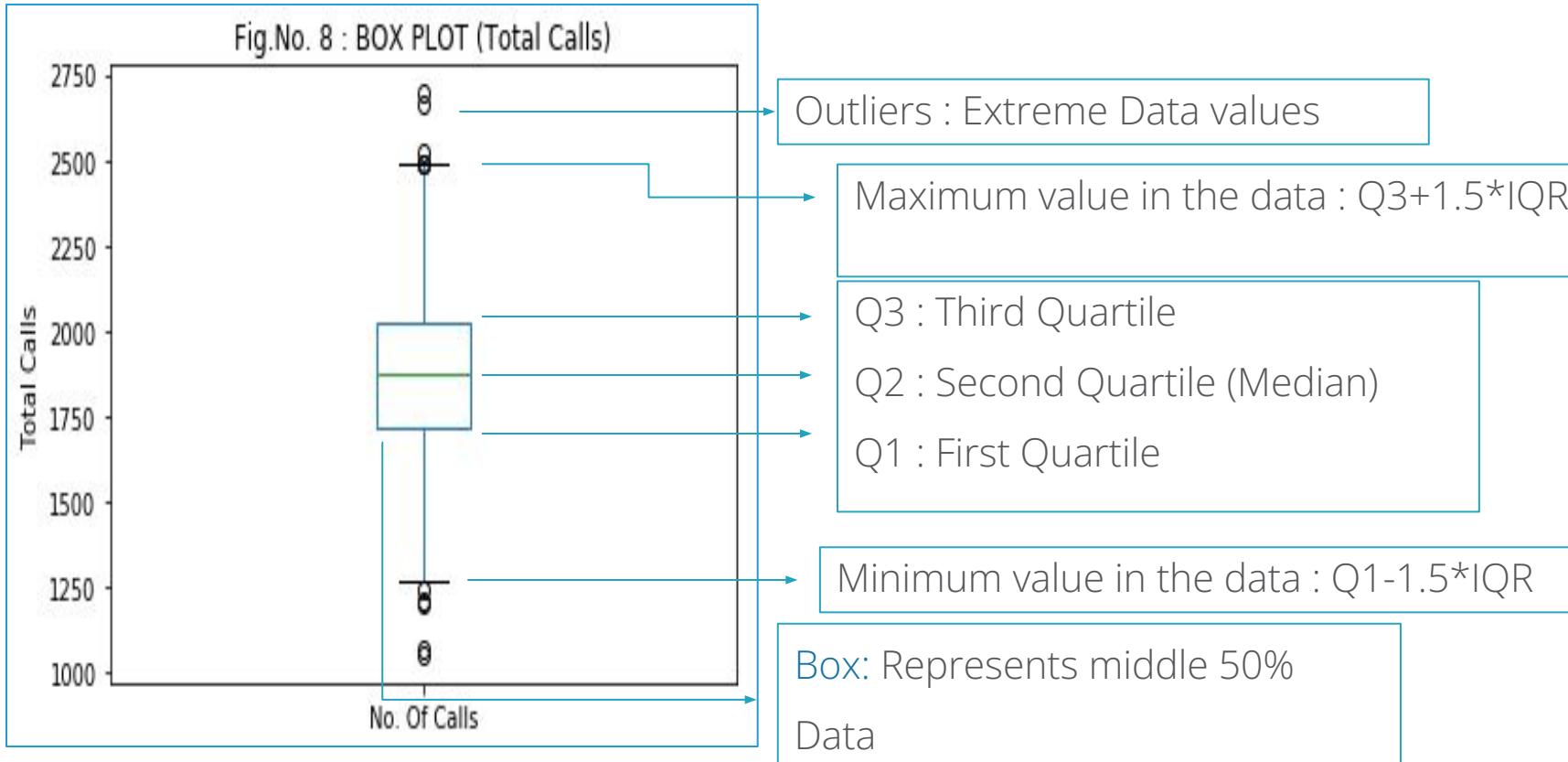
# Box Plot in Python

```
#Importing Data  
  
import pandas as pd  
telecom = pd.read_csv("telecom.csv")  
  
#BoxPlot - Total Calls  
  
import matplotlib.pyplot as plt  
telecom.Calls.plot.box(label='No. Of Calls');plt.title('Fig.No. 8 :  
BOX PLOT (Total Calls)');plt.ylabel('Total Calls')
```

- **box()** in pandas yields a different types of box chart
- **Calls** specifies vector (column) for which the box plot needs to be plotted
- **label=** provides a user defined label for the variable on X axis
- **ylabel** provides a user defined label for the variable on Y axis

# Box Plot in Python

# Output



## Interpretation :

- This plot shows that the distribution of total call is very much symmetric & there exists few outliers in the data.

# Box Plot in Python

```
#BoxPlot for different categories of Age_Group  
telecom.boxplot(column='Calls', by='Age_Group', grid=False,  
patch_artist=True);plt.title('Fig.No. 9 : BOXPLOT - Average Call  
Time');plt.suptitle('');plt.ylabel('Total Calls')
```

Difference between previous boxplot & this boxplot code is,

- **boxplot()** in pandas yields different types of box chart.  
It's a different way of writing **plot.box()**
- **column** specifies vector (variable) for which the box plot needs to be plotted
- **by** Specifies the vector (column) by which the distribution should be plotted.
- **ylabel** provides a user defined label for the variable on

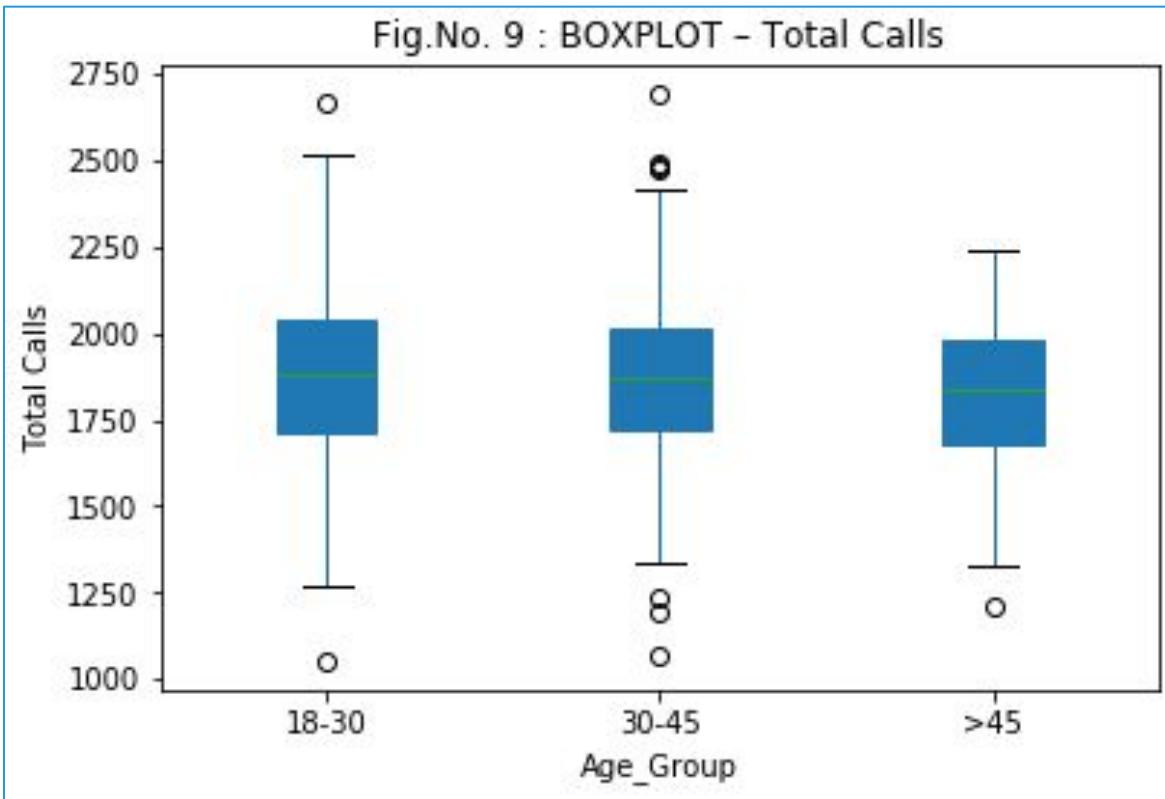
X axis

- 

\* 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 Python

```
# Output
```

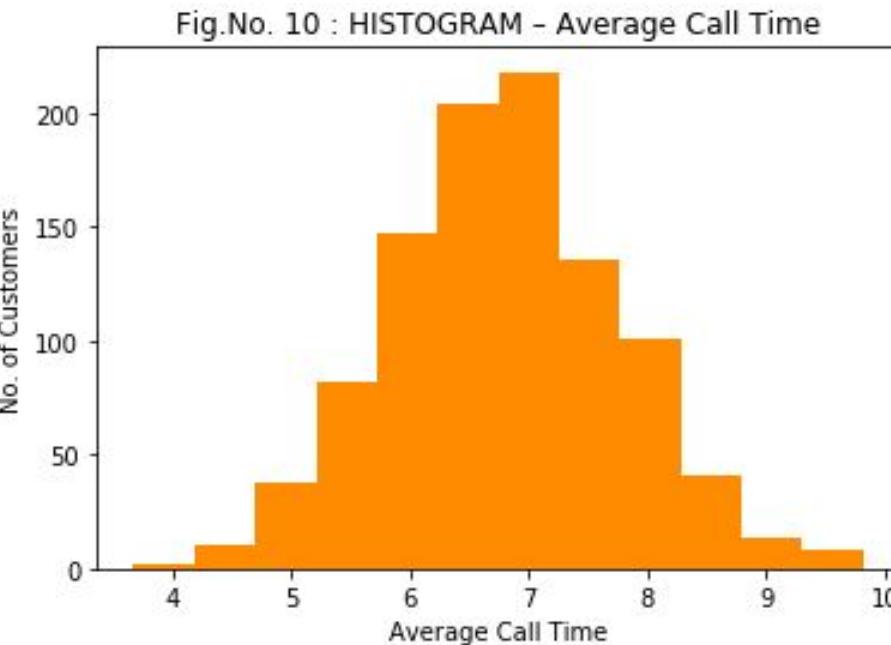


## Interpretation :

- 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.

# Histogram

- A Histogram is similar to a bar chart but is used to display continuous data. Therefore we will use a continuous scale with no 'gaps' between the bars.
- It 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 Python

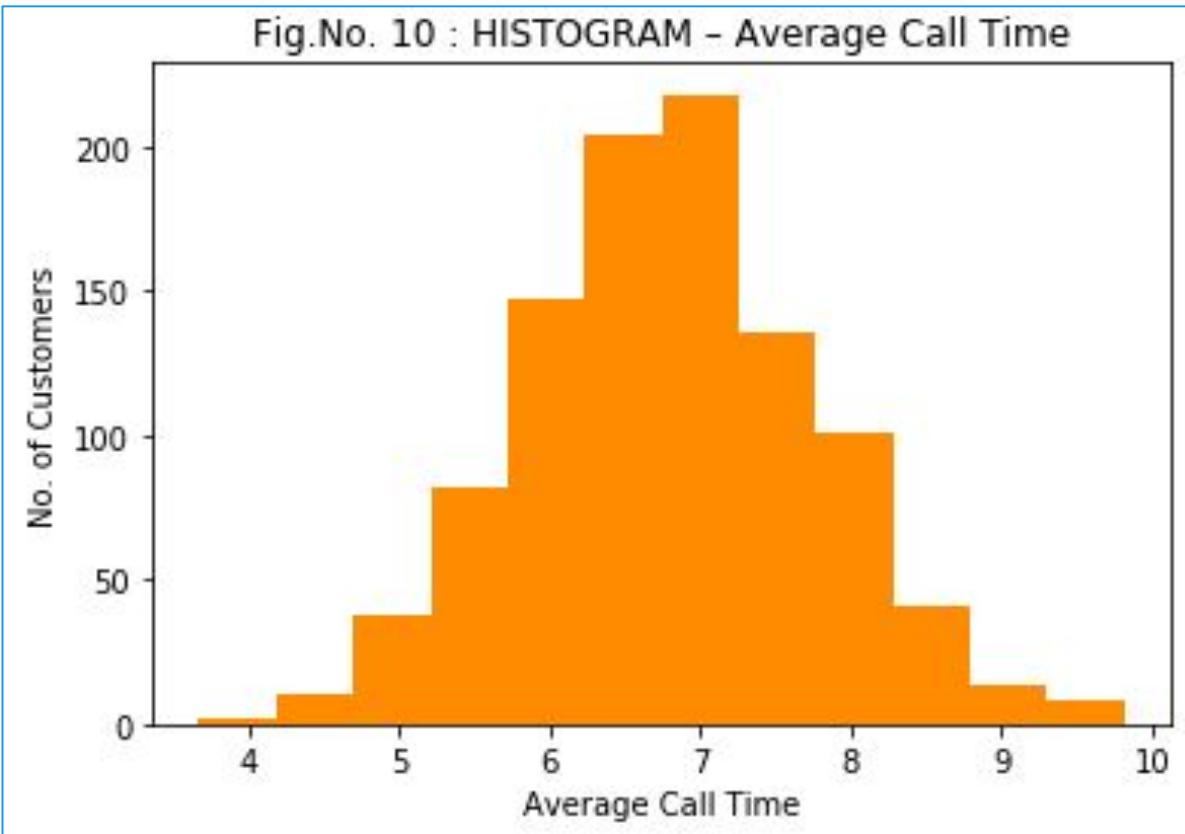
```
# Histogram - Average Call Time
```

```
telecom.AvgTime.hist(bins=12,grid=False, color = 'darkorange');  
plt.title('Fig.No. ↑ 10 : HISTOGRAM - Average Call Time');  
plt.xlabel('Average Call Time');plt.ylabel('No. of Customers')
```

- hist()** yields a histogram
- bins** specifies the width of each bar
- xlabel** provides a user defined label for the variable on X axis
- ylabel** provides a user defined label for the variable on Y axis
- color** can be used to input your choice of color to the bars

# Histogram in Python

This plot shows the distribution of Average Call Time

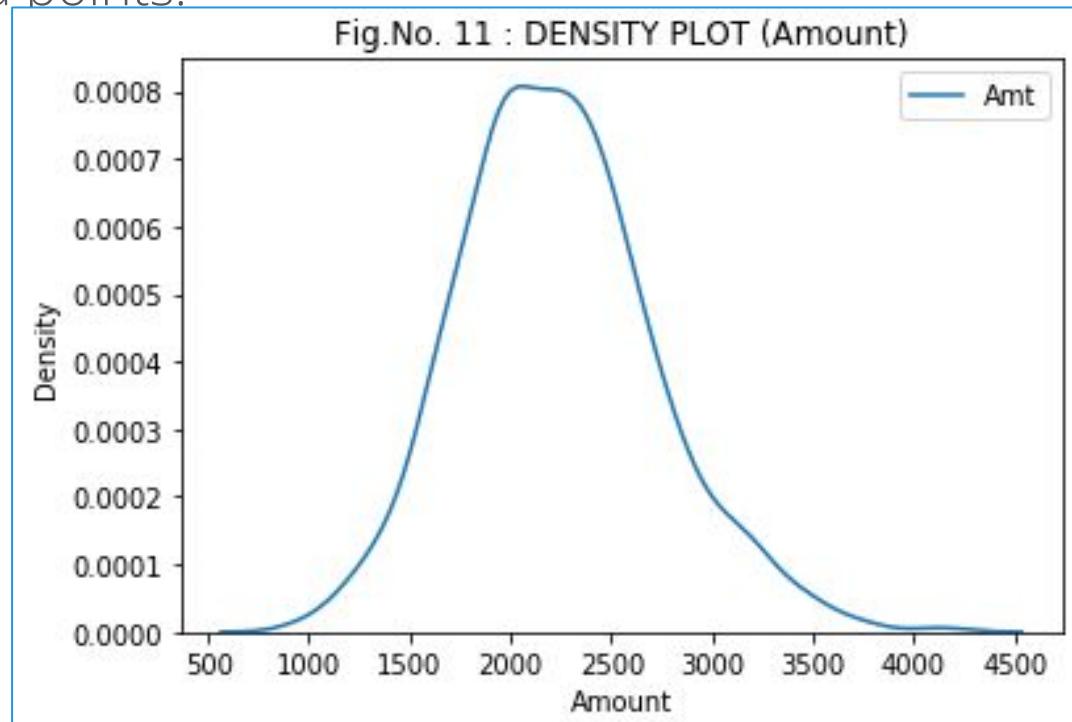


## Interpretation :

- 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 very slightly positively skewed.

# Density Plot in Python

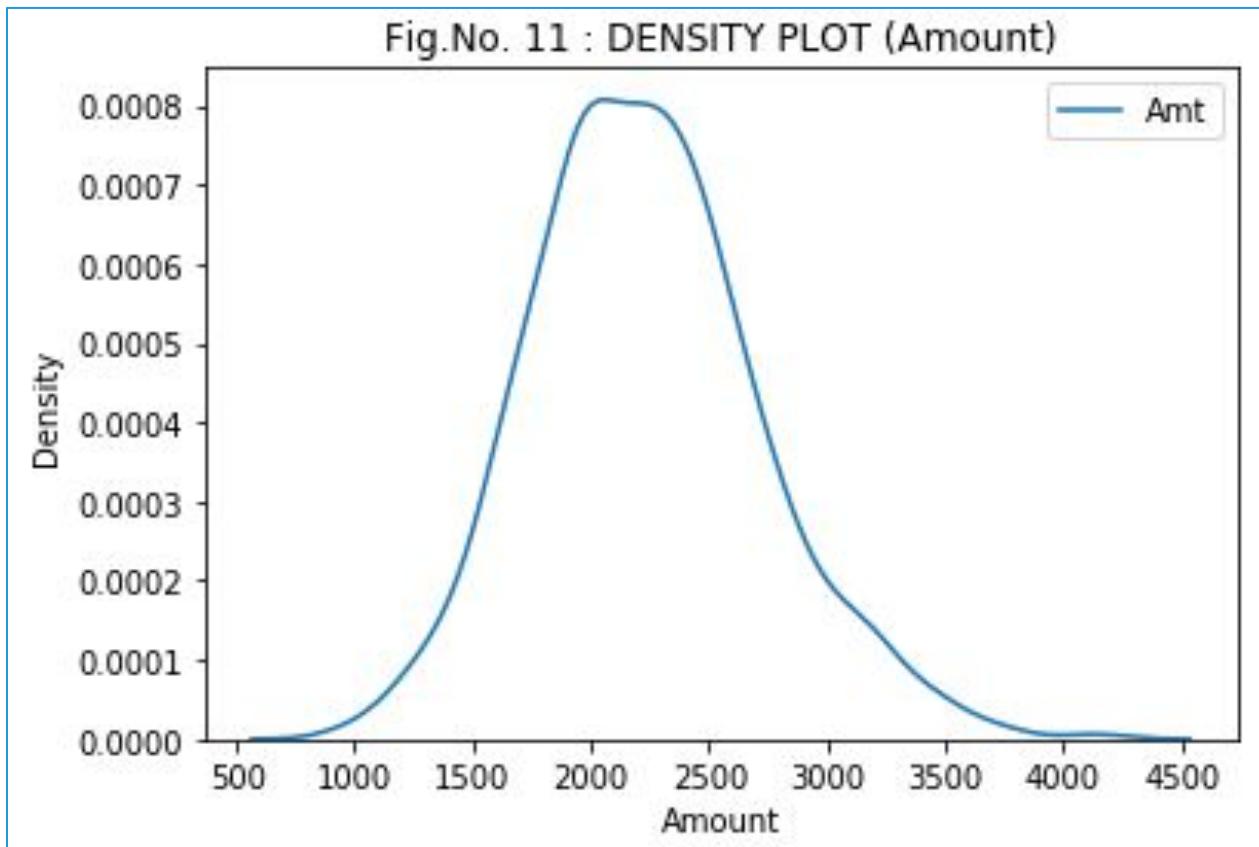
```
# Density Plot - Amount
```

```
(telecom['Amt'].plot.kde();plt.title('Fig.No. 11 : DENSITY PLOT  
(Amount)');plt.xlabel('Amount'))
```

- kde()** returns the density values of the variable (kernel density estimation)
- plot()** plots the line graph of the specified variable
- title** provides the user defined name of the chart. It is to be put in double quotes
- xlabel** provides a user defined label for the variable on X axis

# Density Plot in Python

This plot shows the distribution of Amount

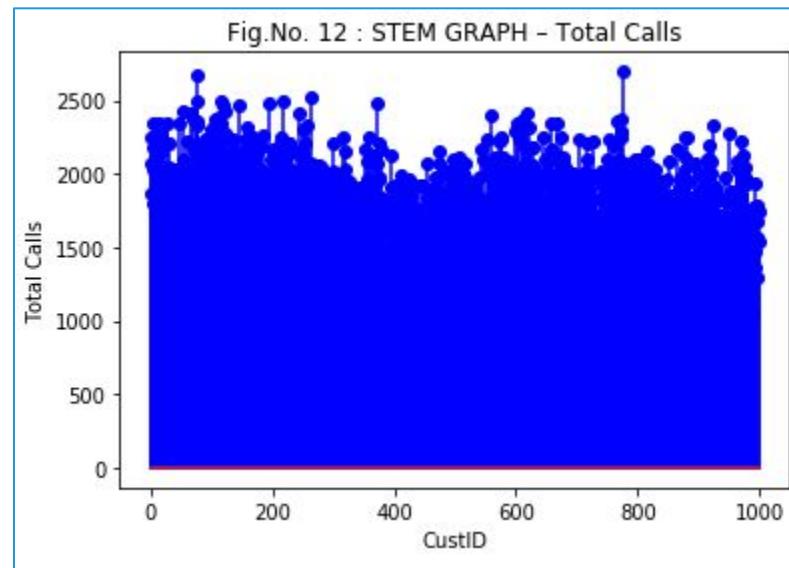


## Interpretation :

- This plot shows that the distribution of Amount is slightly positively symmetric as smaller amount has slightly high frequency count of customers.

# 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).



# Stem and Leaf Plot in Python

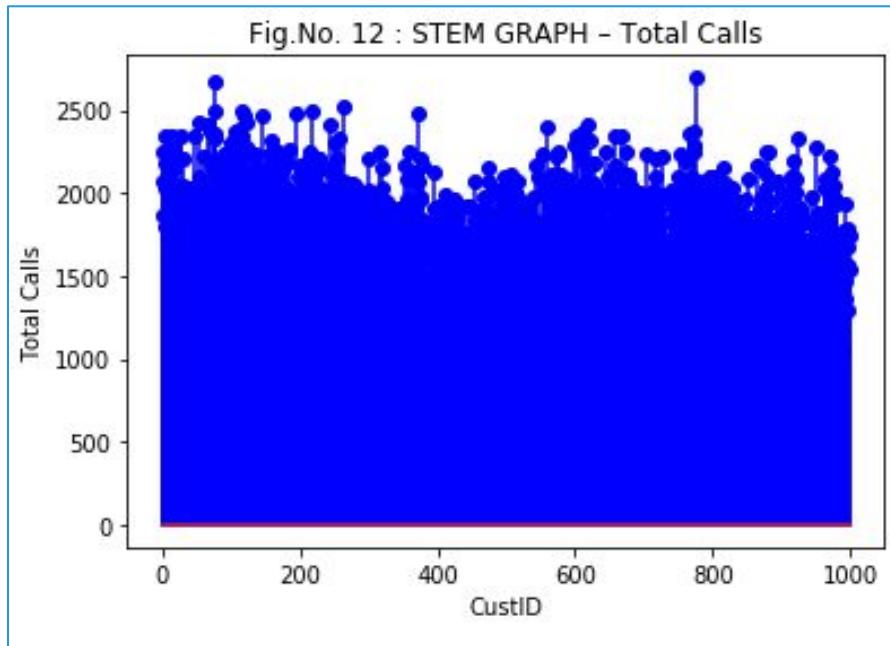
```
# Stem Plot in Python
```

```
plt.stem(telecom.Calls);plt.title('Fig.No. 12 : STEM GRAPH – Total  
Calls'); plt.xlabel('CustID'); plt.ylabel('Total Calls')
```

- stem()** in base Python yields a stem chart
- telecom.Calls** specifies vector (variable) for which the stem plot needs to be plotted

# Stem and Leaf Plot in Python

```
# Output
```

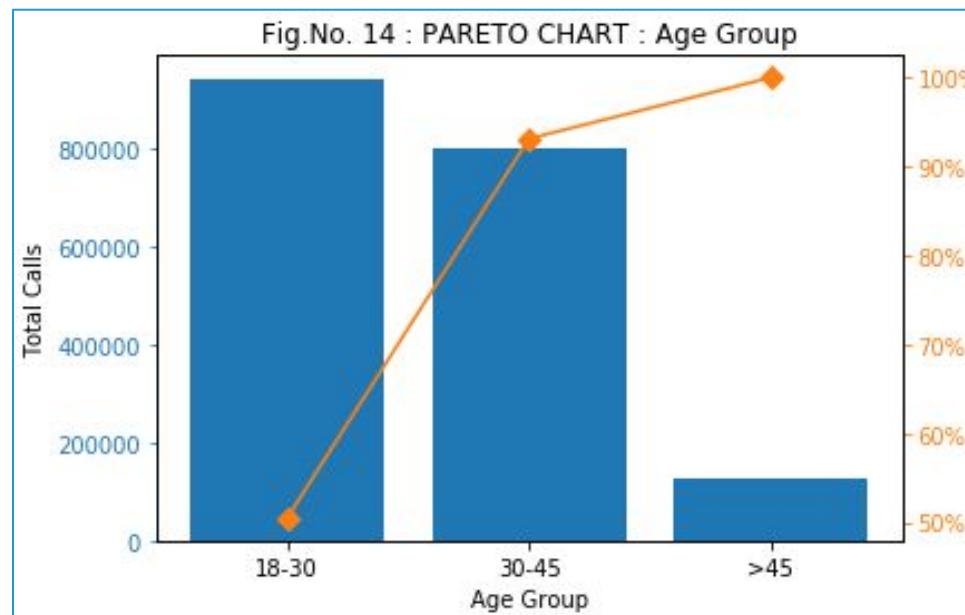


## Interpretation :

- 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.

# 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 .

# Pareto Chart in Python

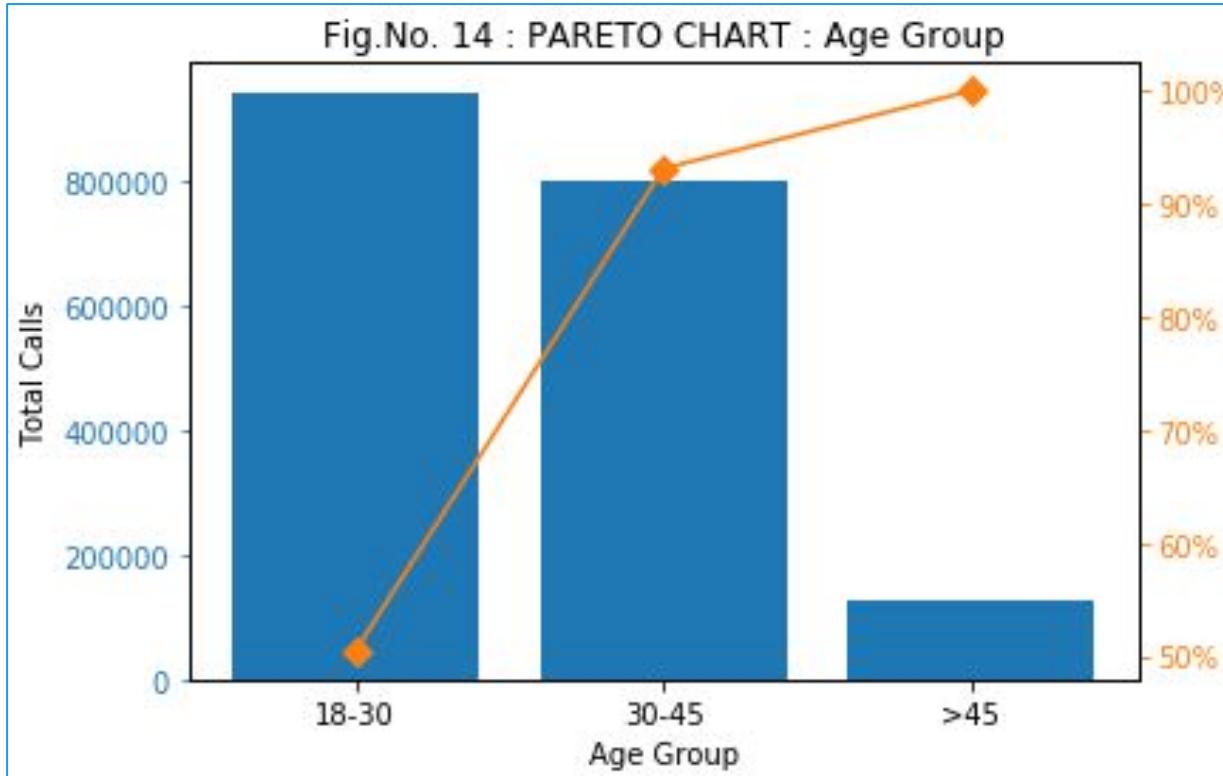
```
# Pareto Chart - Age Group
```

```
telecom1 = telecom.groupby('Age_Group')['Calls'].sum()
telecom1
telecom1 = telecom1.to_frame()
telecom1["cumpercentage"] = telecom1["Calls"].cumsum() / telecom1["Calls"].sum() * 100
fig, ax = plt.subplots()
ax.bar(telecom1.index, telecom1['Calls'])
ax2 = ax.twinx()
ax2.plot(telecom1.index, telecom1['cumpercentage'], color='red', ms=7)
ax2.yaxis.set_major_formatter(PercentFormatter())
ax.tick_params(axis="y", colors="C1")
ax.set_xlabel("Age Group")
ax.set_ylabel("Calls")
ax.set_title("Pareto Chart")
```

- ❑ `to_frame()` function is used to convert the given series object to a dataframe
- ❑ `plt.subplots` method provides a way to plot multiple plots on a single figure. Given the number of rows and columns , it returns a tuple ( `fig , ax` ), giving a single figure `fig` with an array of axes `ax`
- ❑ `telecom1.index` is the argument that allows the bars to be named according the row names in the variable mentioned
- ❑ `telecom1["Calls"]` specifies vector (variable) for which the Pareto chart needs to be plotted
- ❑ `ax2.twinx()` Create a twin Axes sharing the X axis
- ❑ `set_major_formatter(PercentFormatter())` sets percentage format on y axis for pareto chart
- ❑ `ax.tick_params` provides axis ticks the chart. It has to be put in double quotes
- ❑ `colors` can be used to input your choice of color to the bars
- ❑ `ax.set_xlabel, ax.set_ylabel` provides a user defined label for the variable on X and Y axes

# Pareto Chart in Python

# 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

# Get an Edge!

Graphs for different types of Variables

Type of Variable	Chart
Discrete	Bar Graph
Continuous	Histogram/Boxplot/Density Plot
Categorical	Bar Graph/Pie Chart/Pareto Chart
Dichotomous	Multiple/ Stacked Bar Chart

# Quick Recap

In this session, we learnt data visualisation using basic graphs

## Chart Types and Functions in Python

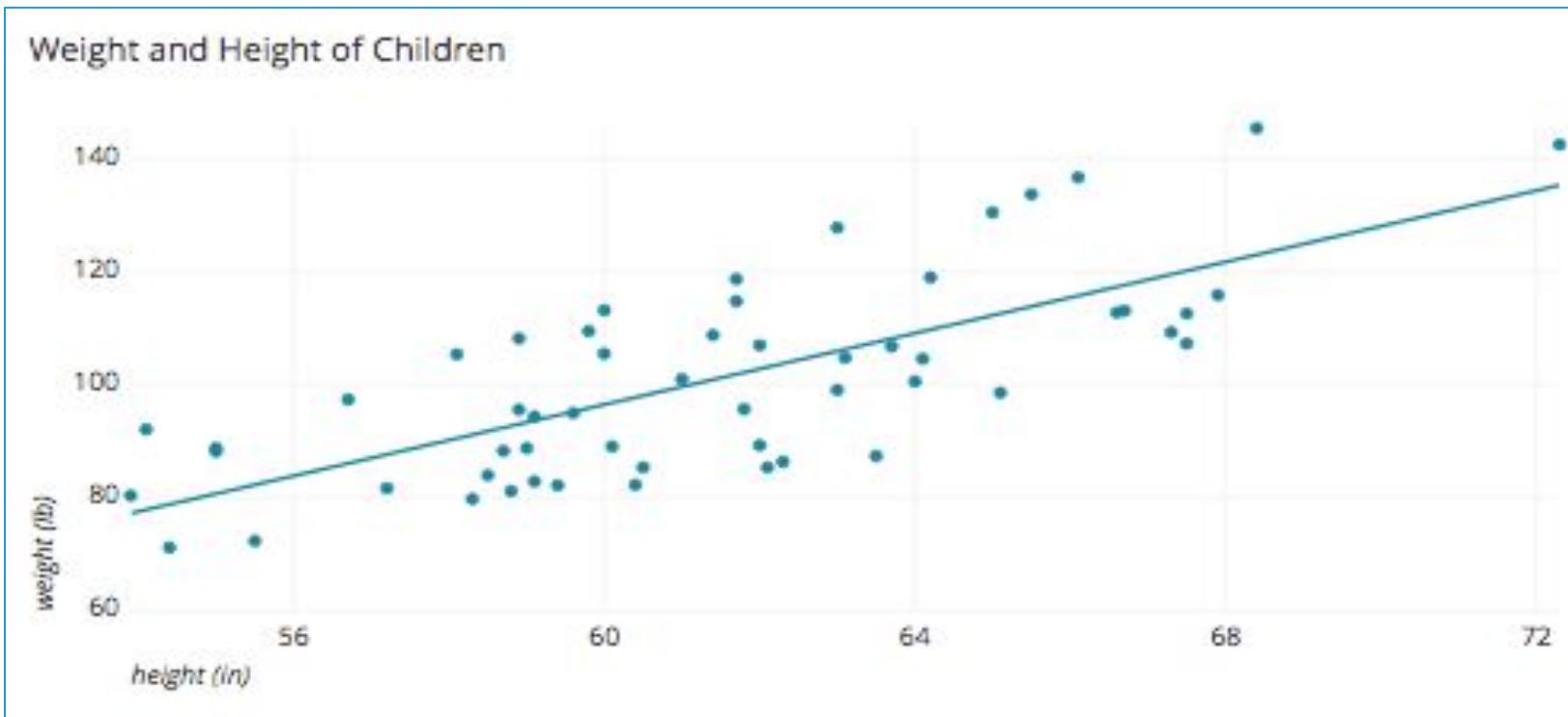
- .. Box-Whisker Plot – `box()`,`boxplot()`
- .. Histogram - `hist()`
- .. Density Plot - `kde()`
- .. Stem Plot- `stem()`
- .. Pareto Chart – `bar()` + `twinx()` + `plot()`

# Visualizing Relationships in Python

# 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. Summarizing Data in Diagrams using Python

# Scatter Plot



Each dot represents one child with his or her height measured along the x-axis and weight measured along the y-axis

# Case Study

## 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
--	--	--	--	--	--

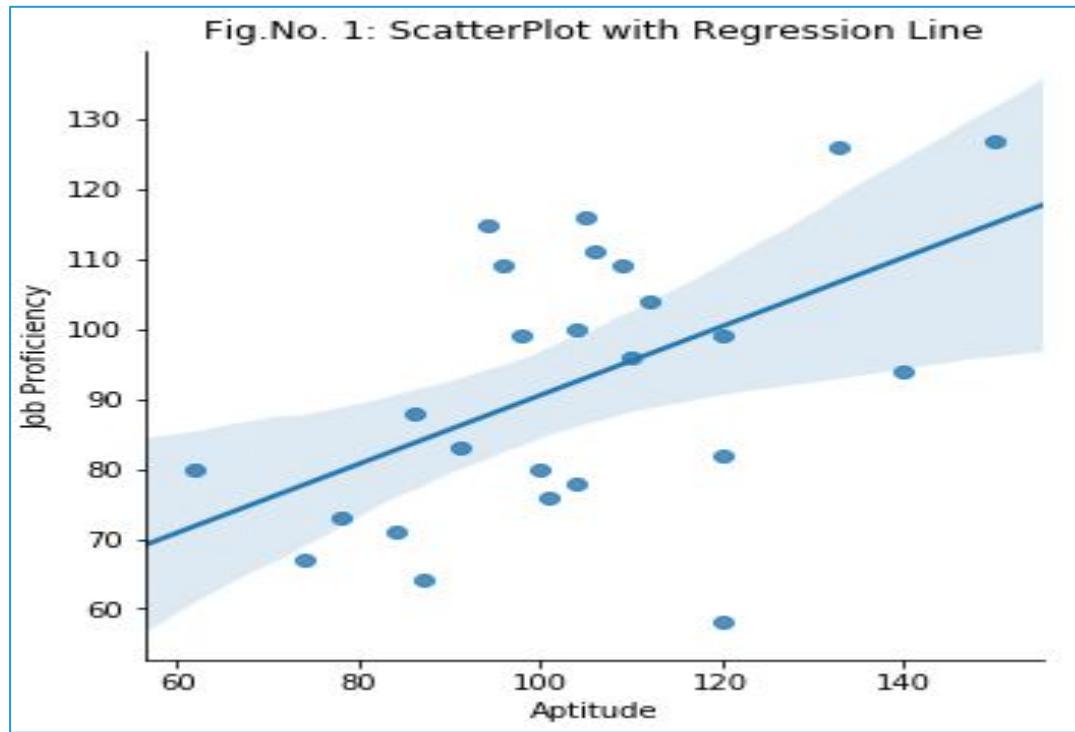
Columns	Description	Type	Measurement	Possible values
empno	Employee No	Numeric	-	-
aptitude	Aptitude	Numeric	-	positive values
testofen	Test of English	Numeric	-	positive values
tech_	Technical Score	Numeric	-	positive values
g_k_	General Knowledge	Numeric	-	positive values
job_prof	Job Proficiency	Numeric	-	positive values

# ScatterPlot with Regression Line in Python

```
#Importing Data  
  
import pandas as pd  
job=pd.read_csv('JOB PROFICIENCY DATA.csv', index_col=0) ← index_col= 0  
instead of  
None (take  
first column  
as index by  
default)  
  
#Importing Library Seaborn  
  
import seaborn as sns  
import matplotlib.pyplot as plt  
  
#Scatterplot of job proficiency against aptitude with Regression  
Line  
sns.lmplot('aptitude','job_prof',data=job);plt.xlabel('Aptitude');plt.  
ylabel('Job Proficiency');plt.title('Fig.No. 1: ScatterPlot with  
Regression Line')
```

- ❑ `sns.lmplot()` calls a scatter plot from sns object with regression line
- ❑ `plt.xlabel` provides a user defined label for the variable on x axis
- ❑ `plt.ylabel` provides a user defined label for the variable on y axis
- ❑ `plt.title` gives title to the plot

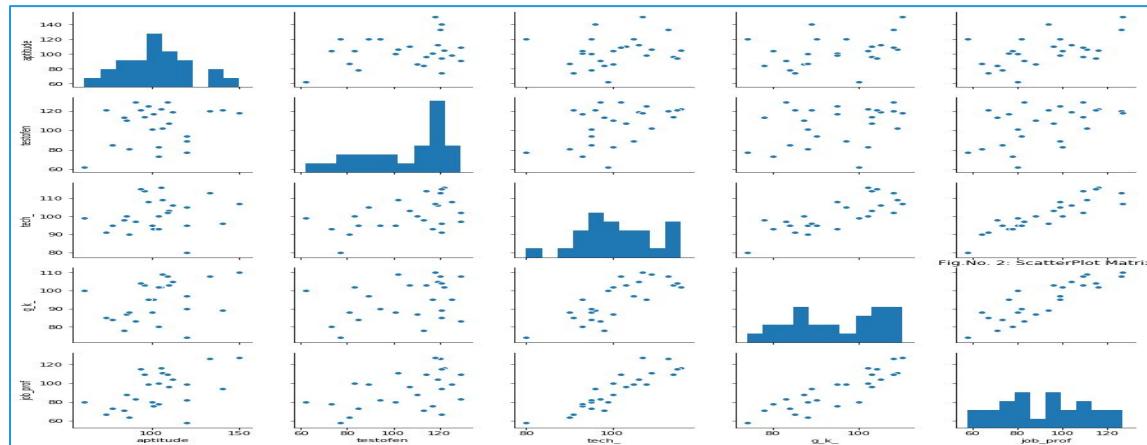
# ScatterPlot with Regression Line in Python



## 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

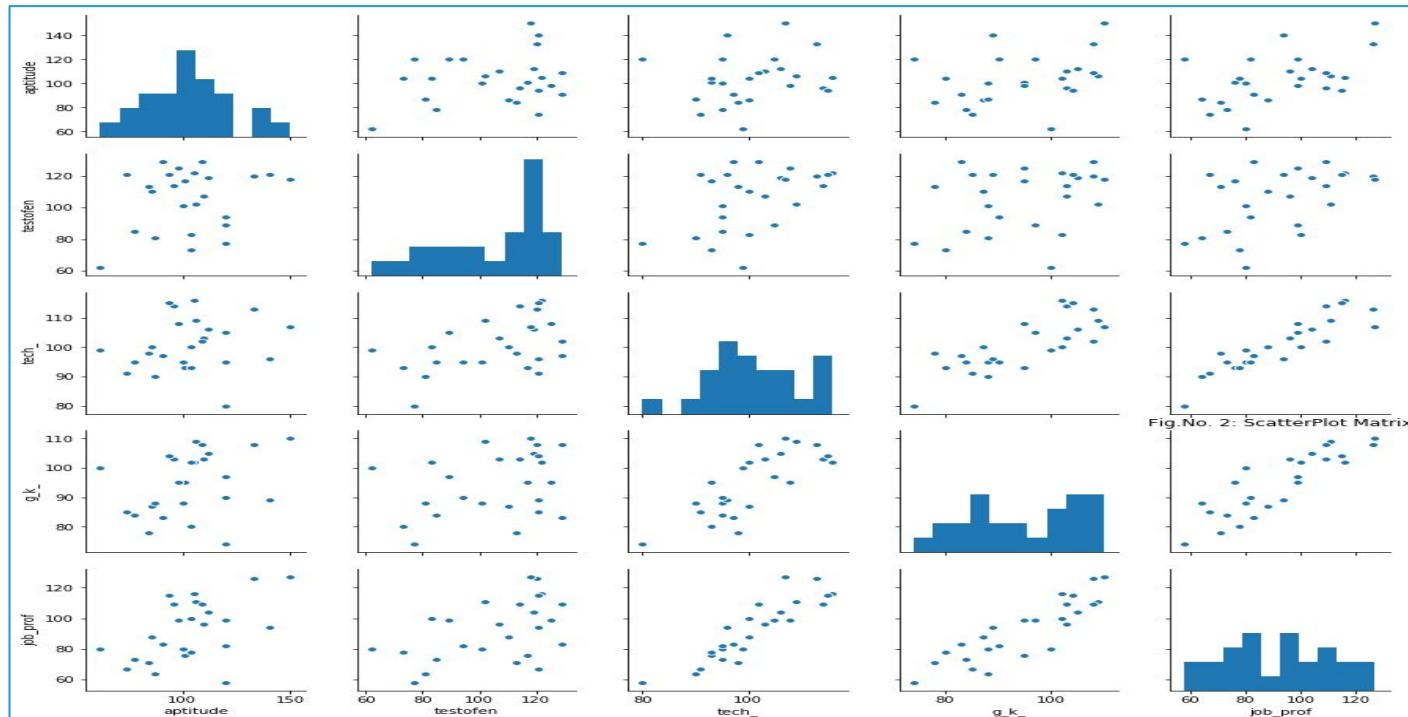


```
# ScatterPlot Matrix  
sns.pairplot(job);plt.title('Fig.No. 2: ScatterPlot Matrix')
```

❑ pairplot() from sns is used to plot pairwise comparison

# Scatter Plot Matrix in Python

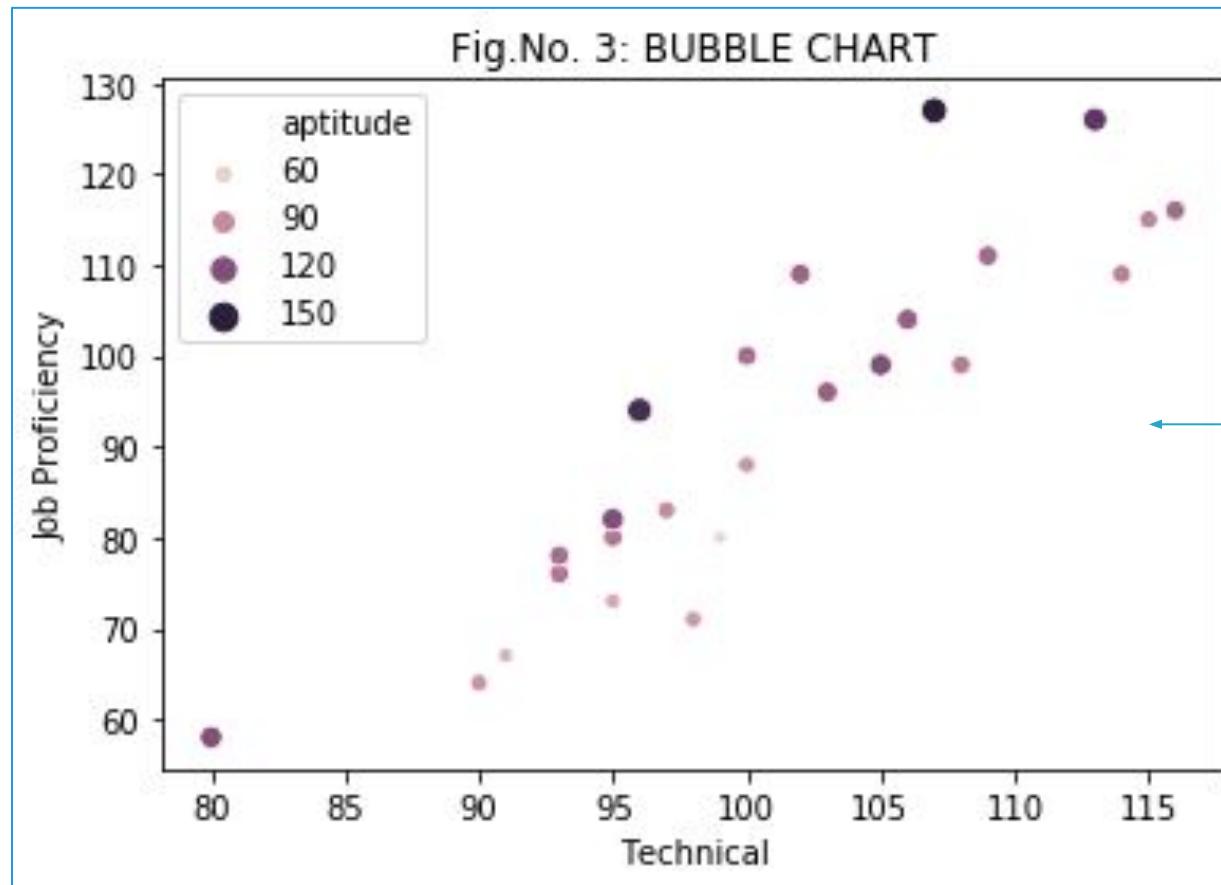
# Output



## 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.

# Bubble Chart



## 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 Python

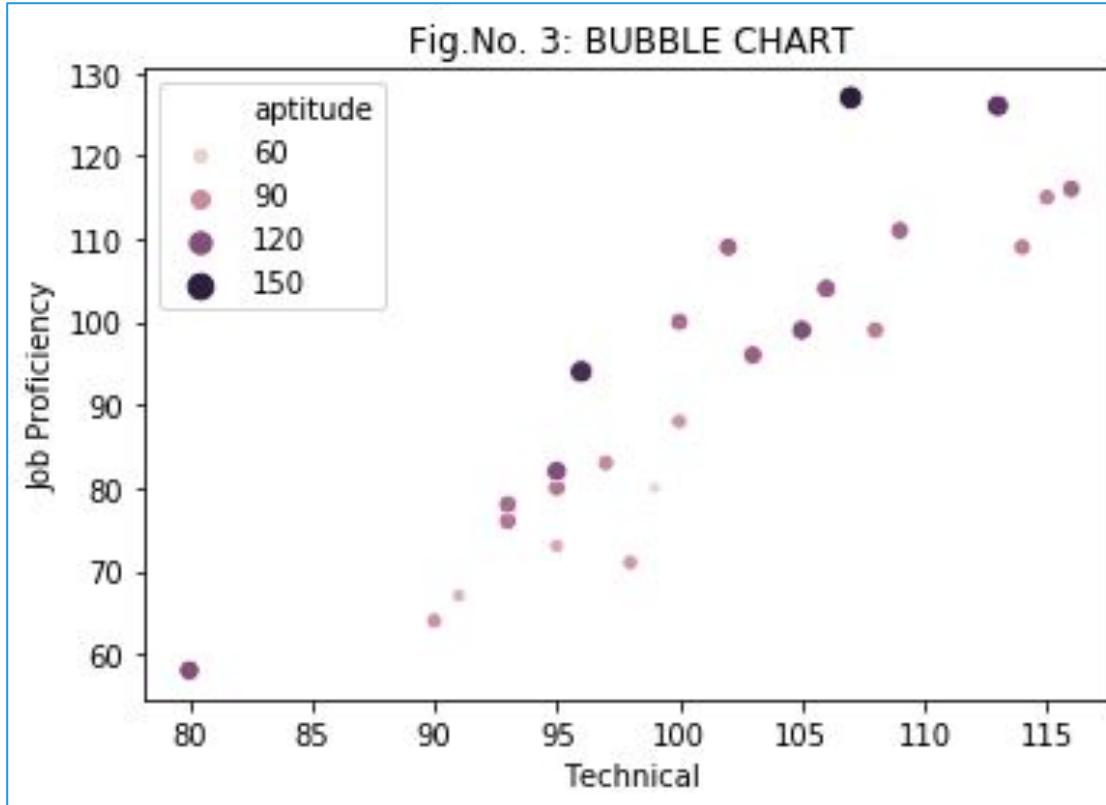
```
# Bubble Chart
```

```
sns.scatterplot('tech_', 'job_prof', data=job,  
hue='aptitude', size='aptitude'); plt.title('Fig.No. 3: BUBBLE CHART');  
plt.xlabel('Technical'); plt.ylabel('Job Proficiency')
```

- sns.scatterplot() calls a scatter plot from sns object
- tech\_, job\_prof are variables to be plotted on x and y axis
- hue gives colors based on aptitude score
- size assigns the size to the bubble based on aptitude score

# Bubble Chart in Python

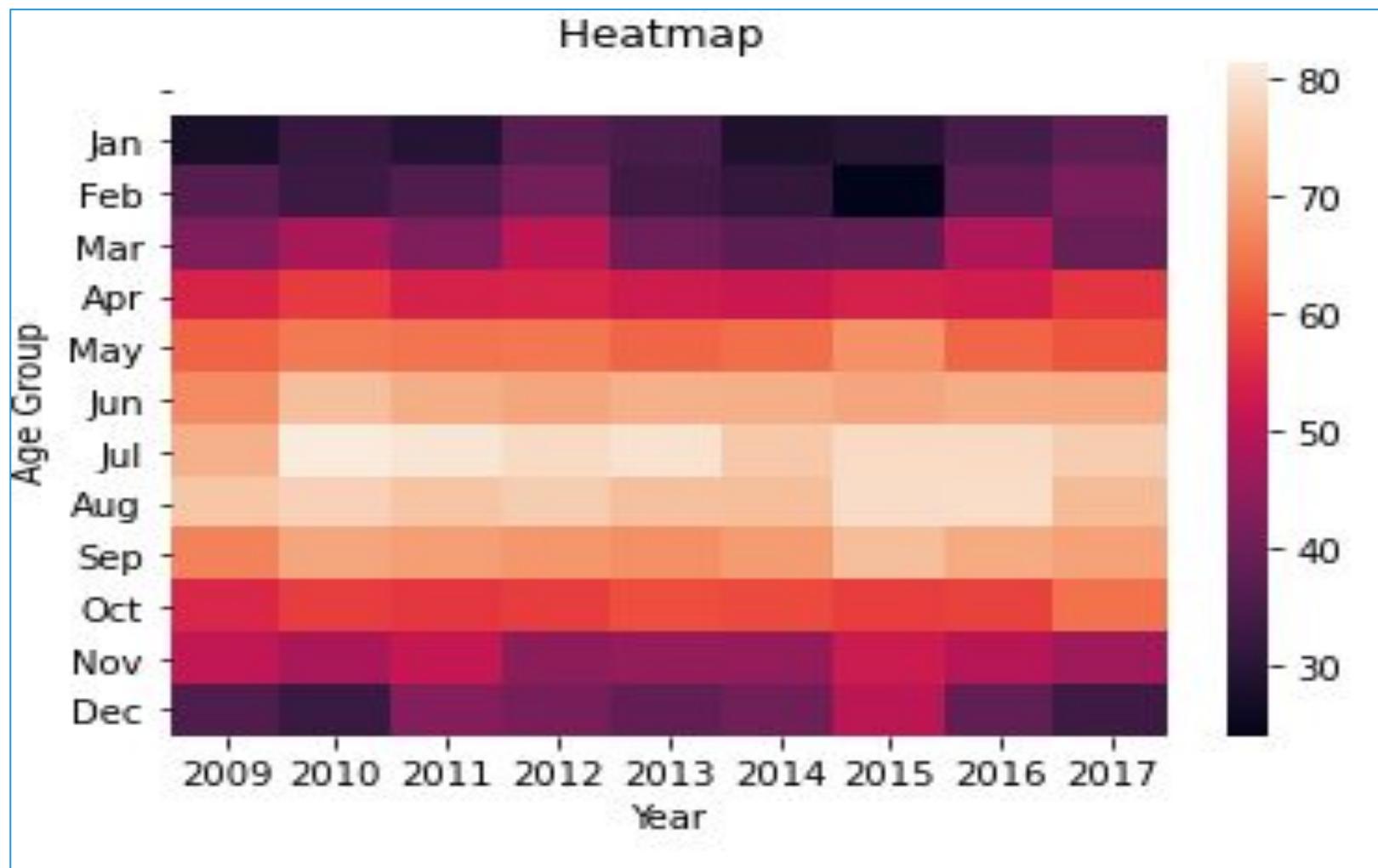
# Output



## 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



# 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



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	-	-

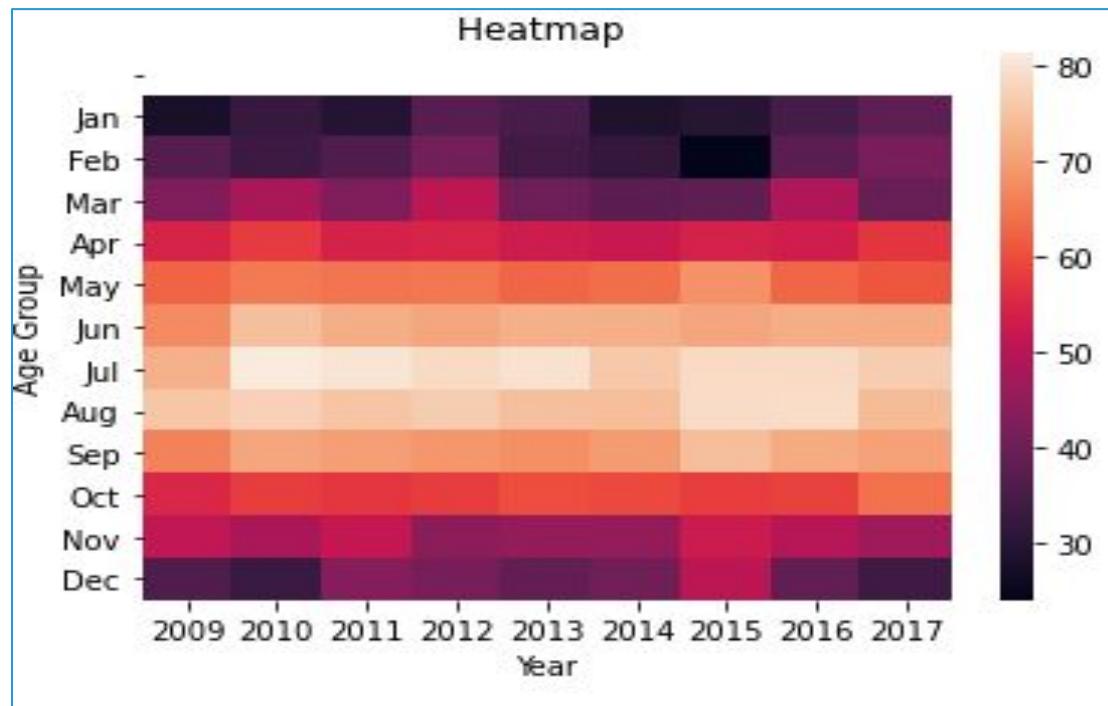
# Heat Map in Python

```
# Installing and calling the package  
import seaborn as sns  
import calendar  
  
# Importing Data and Arranging the Months in the right order :  
heatmapdata=pd.read_csv('Average Temperatures in NY.csv')  
agg=pd.pivot_table(heatmapdata, index=['Month'], columns=['Year'])  
agg.columns = (heatmapdata['Year']).unique()  
agg = agg.reindex(list(calendar.month_abbr))  
  
# Heat Map  
plt.show(); ax=sns.heatmap(agg);ax.set(xlabel='Year', ylabel='Age  
Group',title='Heatmap ')
```

- 
- ❑ **calendar** library gives the functions related to calendar manipulations such as Year, Month.

# Heat Map in Python

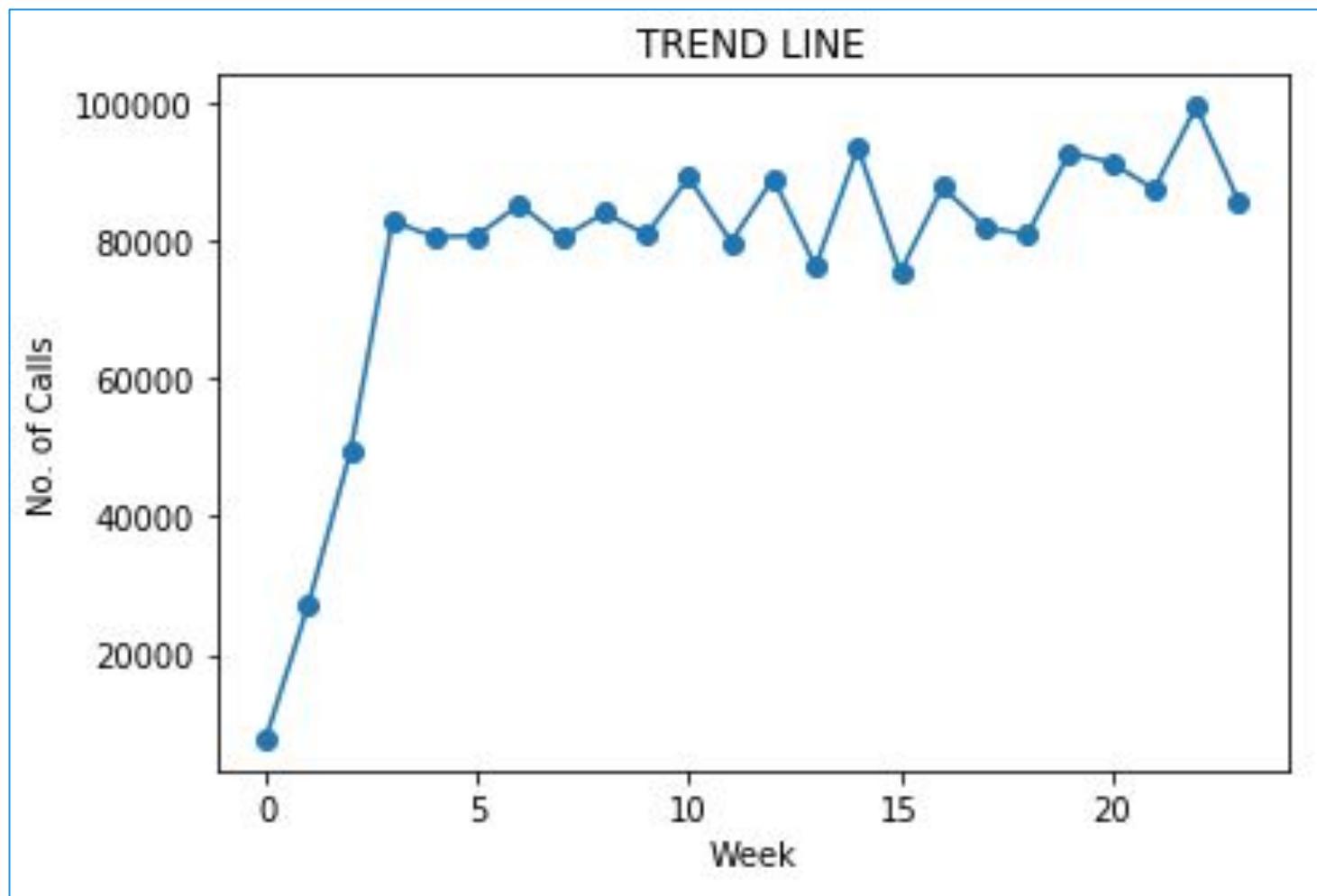
# Output for Heat Map :



## 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



# 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.

TelecomData\_WeeklyData

Variables					
Observations	CustID	Week	Calls	Minutes	Amt
Columns	1001	1	56	292	70.4
CustID	Description	Type	Measurem ent	Possible values	-
Week	Customer ID	Numeric	-	-	24
Calls	Week no.	Numeric	1-24	positive values	
Minutes	No. of Calls	Numeric	-		
Minutes	Total Minutes	Numeric	Minutes	positive values	
Amt	Amount Charged	Numeric	Rs.	positive values	20

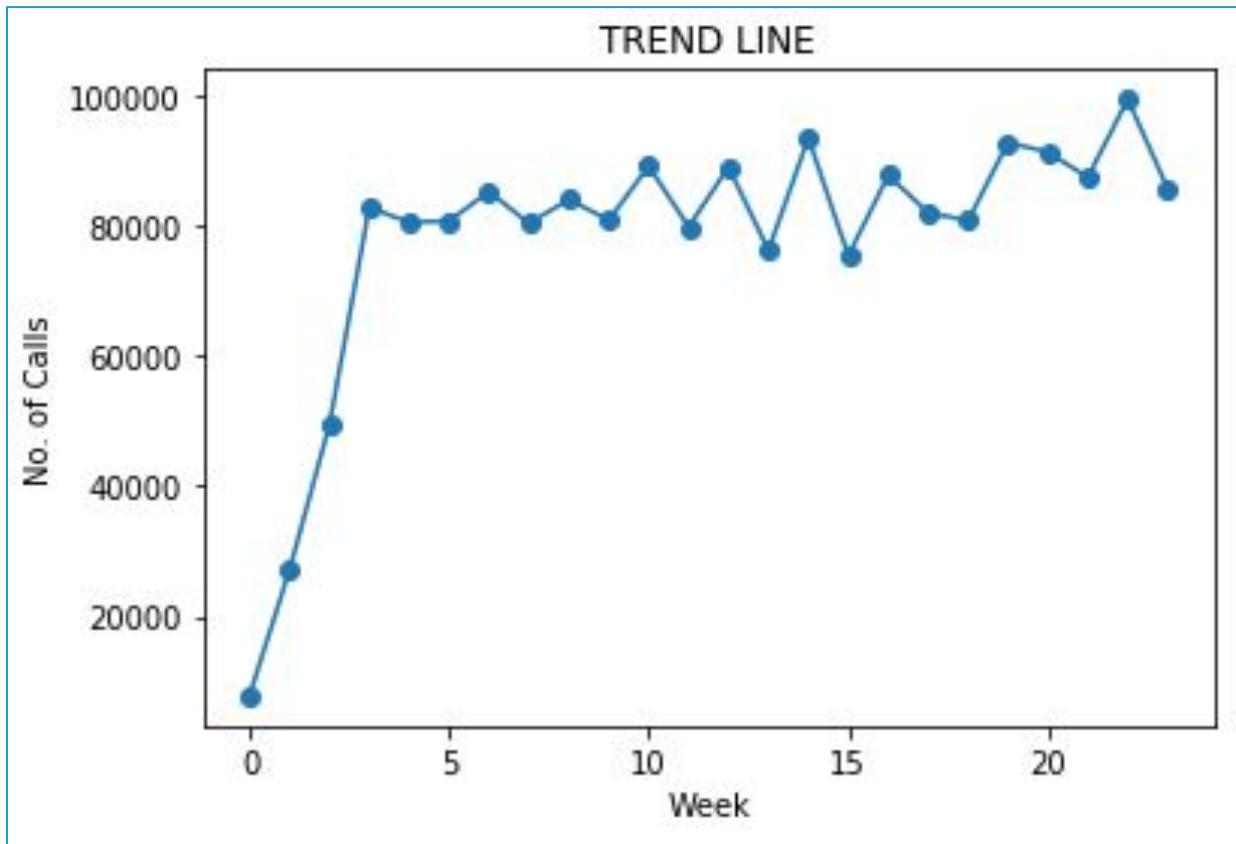
# Trend Line in Python

```
# Importing Data  
transaction = pd.read_csv("TelecomData_WeeklyData.csv")  
  
# Merging and Formatting Data  
trend=(transaction.groupby('Week')['Calls'].sum().to_frame()).reset_index()  
)  
# Trend Line  
plt.plot(trend['Calls'], marker='o');plt.xlabel('Week');plt.ylabel('No.  
of Calls');plt.title('TREND LINE')
```

- The basic function is `plot(x, data, marker, color)`
- `x` is a vector containing the numeric values.
- `marker` plots simple line, "o" used to draw both points and lines.
- `plt.xlabel` is the label for x axis.
- `plt.ylabel` is the label for y axis.
- `plt.title` is the Title of the chart.
- `color` is used to give colors to both the points and lines.

# Trend Line in Python

# Output

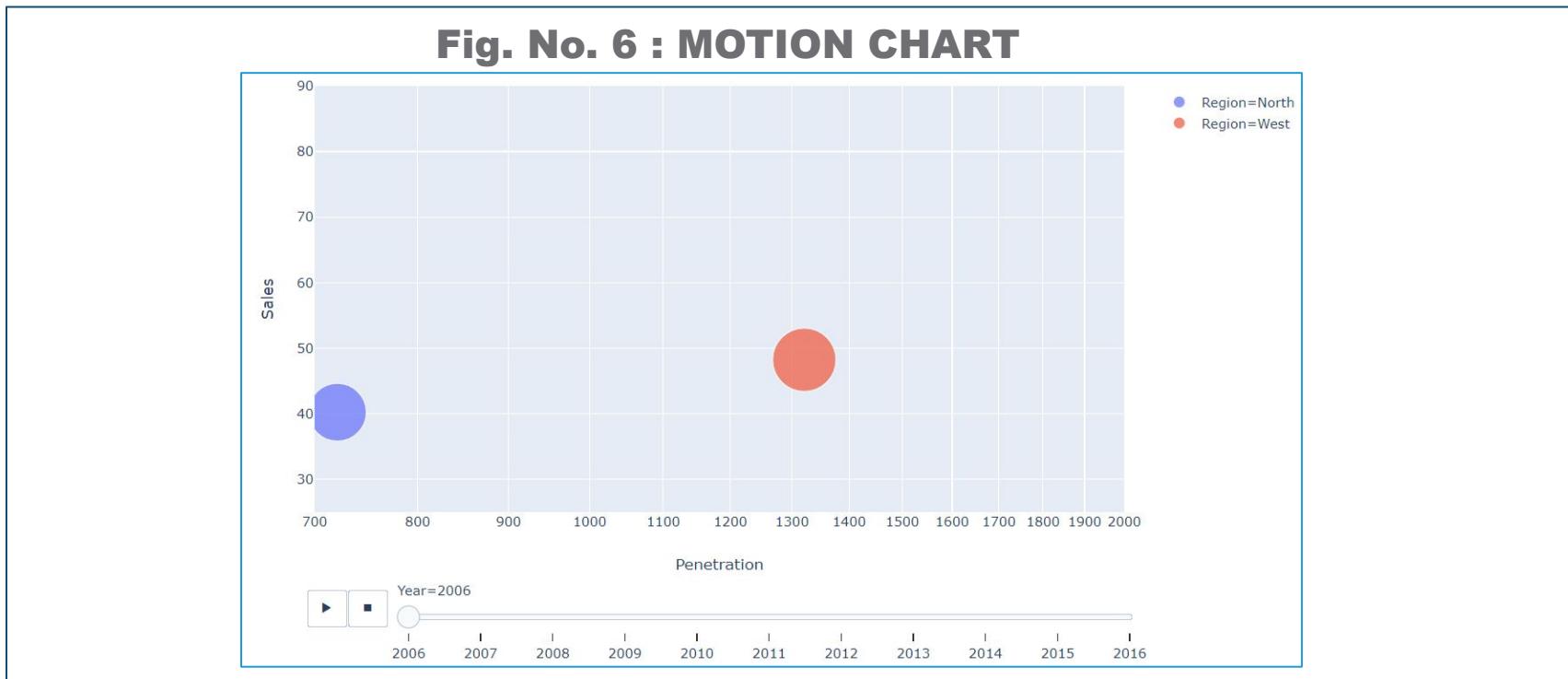


## Interpretation :

- 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 Python

To create a motion chart in python execute the following code in Jupyter Notebook.

```
#Importing Data
```

```
sales = pd.read_csv("Sales Data (Motion Chart).csv")
```

```
#Installing plotly-express
```

```
pip install plotly-express
```

Install **plotly-express** using pip installer with this command in anaconda prompt

```
#Installing
```

```
import plotly.express as px
```

**plotly-express** is the best package we can use to plot an effective Motion Chart in Python

```
# Motion Chart
```

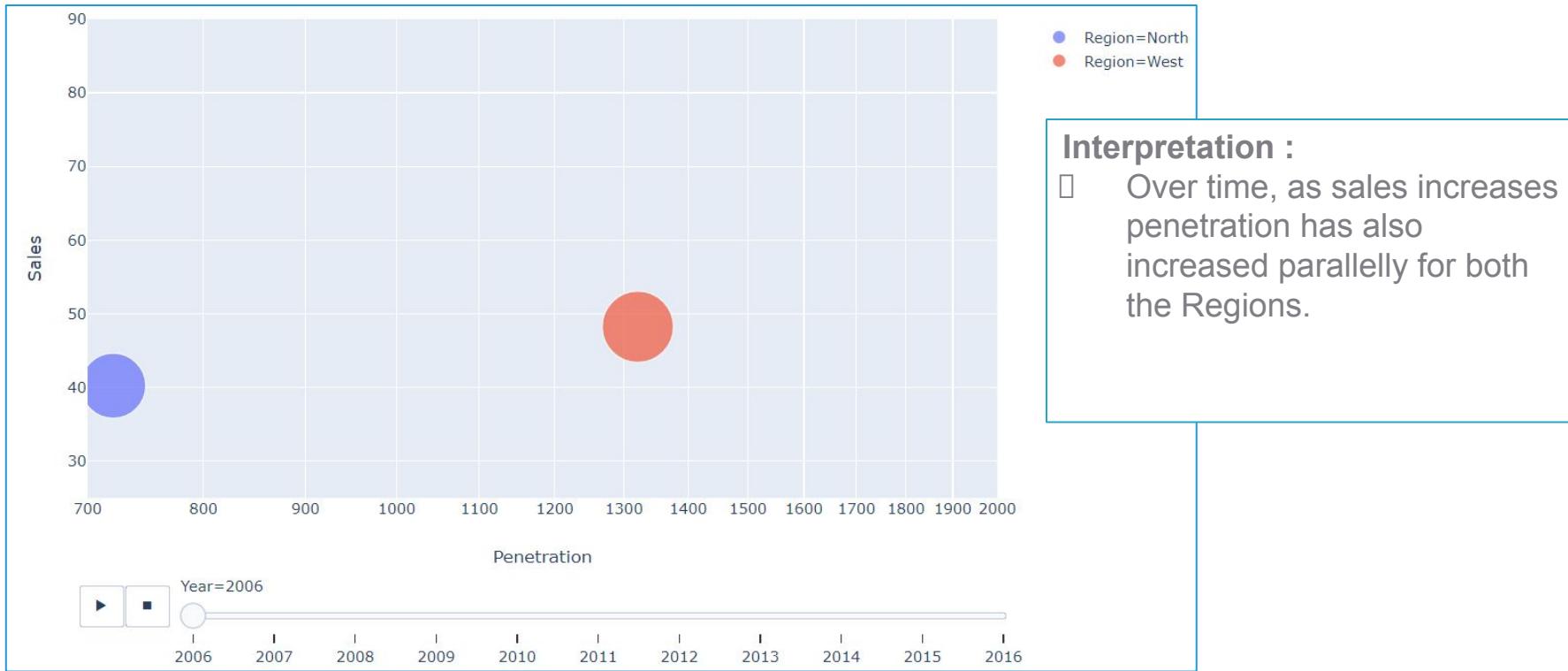
```
px.scatter(sales, x="Penetration", y="Sales", animation_frame="Year",
           animation_group="Region", size="Sales", color="Region",
           hover_name="Region", log_x=True, size_max=55, range_x=[700,2000],
```

```
range
```

- ❑ **px.scatter** is the function used to create a motion chart
- ❑ **sales** is the data that is used
- ❑ **animation\_frame**= inputs time variable
- ❑ **animation\_group**= inputs of categorical variable
- ❑ **log\_x**=(default False)If True, the x-axis is log-scaled in cartesian coordinates.

# Motion Chart in Python

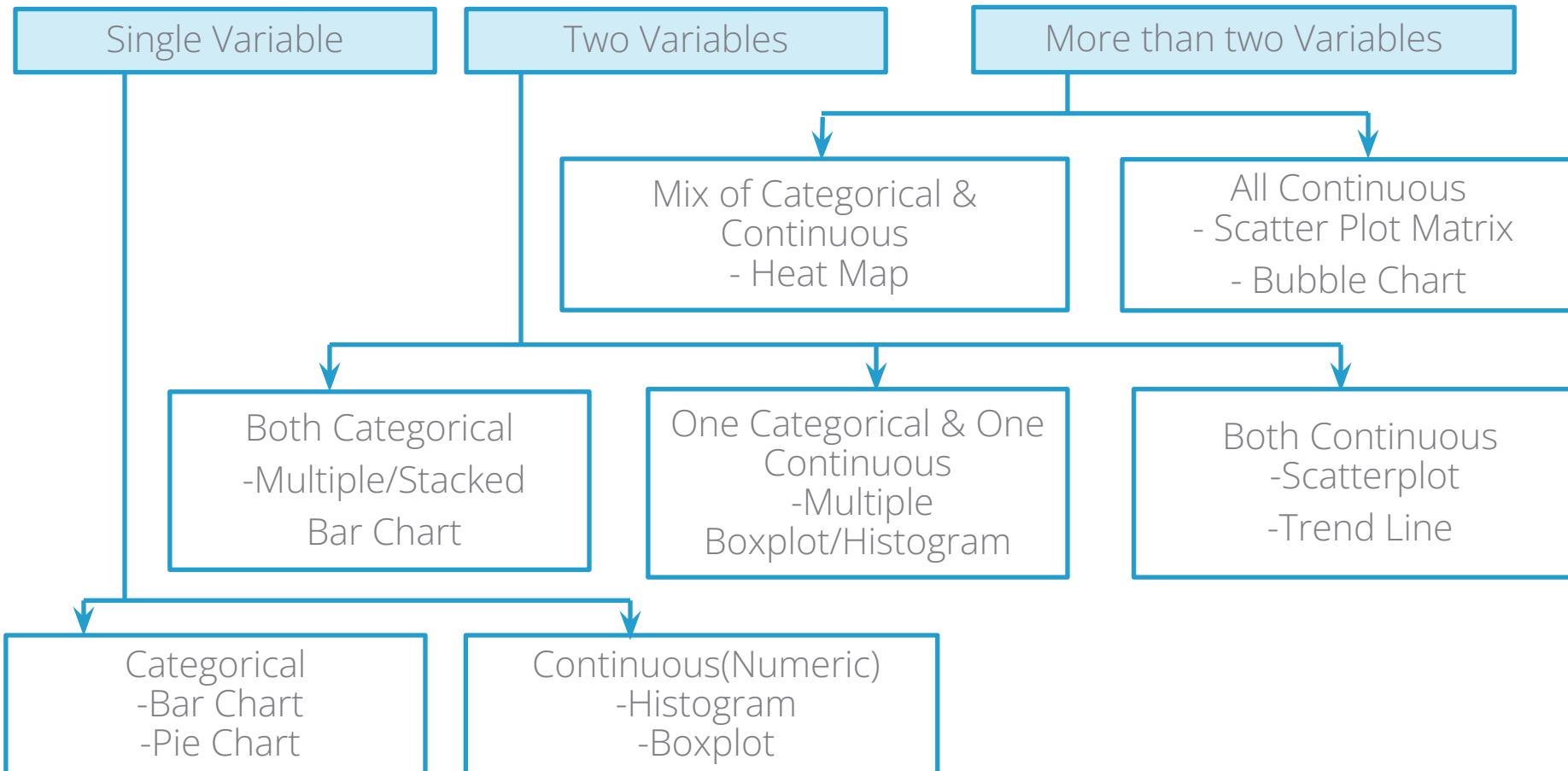
# Output



Motion chart can be run only with Jupyter Notebook using plotly-express

# Get an Edge!

Choosing the right graph



# Quick Recap

In this session, we learnt data visualisation using basics graphs

Chart Types and Functions in Python

- Scatterplot with Regression Line `sns.lmplot()`
- Scatterplot Matrix – `sns.pairplot()`
- Bubble Chart – `sns.lmplot()`
- Heat Map – `sns.heatmap()`
- Trend Line – `plot()`
- Motion Chart – `px.scatter` from package "plotly.express"