Regular Expression :=

A Regular Expression, is a sequence of character that forms a search Pattern.

Regular Expression can be used to check if a string contains the specified search Pattern.

ReqEx Module.

Python has a build-in package called re, which can be used to work with Regular Expressions

Import the re module

```
Import re

str = "hello world"
result = re.match (r'hello', str)
Print (result.group())
```

re.search()

search the string to see if it starts with "the" and ends with "Spain":

```
str = "Python"
result = re.search (r"n", str)
Ping (result.group())
```

Special characters :-

re.findall() :-

the findall() function returns a list containing all matches.

```
txt = "the rain in spain"
result = re.findall ("ai", txt)
print (result)
```

characters:

① [] → a set of characters

Ex: "[a-m]"

Ex: txt = "the rain in spain"

```
x = re.findall ("[a-m]", txt)   # find all lower case characters
print (x)                       alphabetically b/w "a" and "m"
```

o/p :- ['h', 'e', 'a', 'i', 'i', 'a']

② \ → signals a special sequence (can also be used to escape special characters)

Ex: "\d"

Ex:- txt = "that will be 59 dollars"

```
x = re.findall ("\d", txt)   # find all digit
print (x)                    characters:
```

o/p : ['5', '9']

③ . → Any character (except newline character)

Example:- "he..o"

Ex:-
```
txt = "hello planet"
x = re.findall("he..o", txt)
print(x)        o/p:['hello']
```

④ ^ start with → Ex: "^hello"

Ex:
```
txt = "hello planet"
x = re.findall("^hello", txt)
if x:
    print("yes the string starts with 'hell'")
else:
    print("no match")
```

⑤ $ → End with → "planet$"

Ex:
```
tex = "hello planet"
x = re.findall("planet$", txt)
if x:
    print("yes, the string ends with 'planet'")
else:
    print("no match")
```

⑥ * → zero or more occurrence → "he*o"

Ex:-
```
txt = "hello planet"
x = re.findall("he.*o", txt")
print(x)
```

⑦ + → one or more occurrences → "he.+o"

Ex:-

```
txt = "hello planet"
x = re.findall ("he.+o", txt)
print (x)
```

⑧ ? → zero or one occurrence → "he.?o"

Ex:- txt = "hello planet"

x = re.findall ("he.?o", txt")

print (x)

⑨ {} Exactly the specified number of occurences. → "he.{2}o"

Ex:-

```
txt = "hello planet"
x = re.findall ("he.{2}o", txt)
print (x)
```

o/p: ["hello"]

Examples

① result = re.findall (r 'h.t', 'hat hit hot hen hut')

print (result)

o/p = ['hat', 'hit', 'hot', 'hut']

② result = re.findall (r '^H', 'hello world ! Hello again')

Print (result)

o/p = ['Hello']

③ result = re.findall (r 'd$', "Hello world")

print (result)

④ result = re.findall (r "^a...d$", "anand")
print (result)

⑤ result = re.findall (r "ho*", 'hot top hit hen')
print (result) o/p : ['ho', 'ho', 'h', 'h']

⑥ result = re.findall (r 'H+', "Hell Hello Hey H")
print (result)

Special Sequences :-

A special sequence is a \ followed by one of the characters in the list below and has a special meaning

① \d → Return a match where the string contains digits [numbers for 0-9] → "\d"

Ex:- result = re.findall (r "\d+", "phone : 123-456-7890")
print (result) o/p ['123', '456', '7890']

(or)

result = re.findall (r '\d+', "secret@ : ")
if (len (result) > 0 :
print ("valid")
else :
print ("invalid")

② \D → Return a match where the string Does NOT contains digits

Ex:-
result = re.findall (r '\D+', '-#Python-3.13 @ latest')
print (result) o/p : ['-#Python-.', '@ latest']

③ \s → Returns a match where the string contains a white space character "\s"

```
txt = "the rain in spain"
x = re.findall("\s", txt)
print(x)
if x :
    print("yeas, there is at least one match")
else :
    print("no match")
```

④ \S → Returns a match where the string DOES not contain a white space character → "\S"

Ex :- result = re.findall(r'\S+', '-# Python-3.13@ latest')
print(result)

⑤ \w → Returns a match where the string contains any word character [character from a to Z, digits from 0-9, and the underscore_ character. → "\w"

result = re.findall(r'\w+', 'Python-3.13')
Print(result o/p : ['Python_3', '13']

⑥ \W → Returns a match where the string DOES NOT contain any word character → "\W"

result = re.findall(r'\W+', 'Python-3.13')
Print(result)

Errors

1. Syntax errors
2. logical errors
3. runtime errors [exceptions]

1) Syntax error :-

```
if True
print ("Hi")
```

o/p : Syntax error

2. list = ["hi"]
print (list[5]) o/p : index error.

Python Try Except :

- the try block lets you test a block of code for errors
- the except block lets you handle the error.
- the else block lets you execute code when there is no error
- the finally block lets you execute code regardless of the result of the try - and except blocks.

Exception Handling :-

when an error occurs, or exception as we call it, Python will normally stop and generate an error message.
The exceptions can be handled using the try statement

Ex:-

① num = 2

try:

result = 10/num

except:

print ("cannot divide by zero")

print (result).

(00)

② try:

print (x) ## the try block will generate an exception, because x is not defind

except:

print ("An exception occurred")

Many Exceptions:

You can defines as many exception block as you want if you want to execute a special block of code for a special kind of error.

Eg:-

Print one message if the try block raises a NameError and another errors.

Ex:

try:

print (x)

except NameError:

print ("variable x is not defind")

except:

print ("something else went wrong!")

Else

You can use the else keyword to define a block of code to be executed if no errors were raised.

Example:-

In this example the try block does not generate any error.

```
try :
  print ("Hello")
except :
  print ("something went wrong")
else :
  print ("Nothing went wrong")
```

Finally

The finally block, if specified, will be executed regardless if the try block raised an error or not.

Example:-

```
try :
  print (x)
except :
  print ("something went wrong")
finally :
  print ("the 'try except' is finished.")
```

CSV - comma seperated Values

CSV : is a widely used file format for storing tabular data in Python the csv module provides functionality to read from write to csv files Here's a guide to working with csv files in Python.

1. Importing the csv Module :-

```
import csv
```

2. Reading csv files :

a) Reading csv as a list of Rows :-

Each row is a list of string

Ex:-

```
with open ('data.csv', mode='r') as file:
    reader = csv.reader (file)
    for row in reader:
        print (row)
```

b) Reading csv as a dictionary :-

Each row is mapped to a dictionary where keys are the column headers

Ex:-

```
with open ("data:csv', mode = 'r') as file:
    reader = csv.dictReader (file)
    for row in reader:
        Print (row)
```

② writing to csv files:

a) writing Rows from a List

```
data = [['Name', 'Age', 'City'], ['Alice', 25, 'New York'], ['Bob', 30, 'San Francisco']]

with open('output.csv', mode="w", newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data)        # write all rows
```

b) writing Rows from a Dictionary:

```
data = [{'Name': 'Alice', 'Age': 25, 'City': 'New York'},
        {'Name': 'Bob', 'Age': 30, 'City': 'San Francisco'}]

with open('output.csv', mode='w', newline='') as file:
    fieldnames = ['Name', 'Age', 'City']
    writer = csv.DictWriter(file, fieldnames = fieldnames)
    writer.writeheader()          # write the header row
    writer.writerows(data)        # writes all rows.
```

③ appending to a csv file:

To attend rows instead of overwriting the file:

```
new_data = [["Charlie", '35', 'Los Angeles']]

with open('output.csv', mode='a' newline='') as files:
    writer = csv.writer(file)
    writer.writerows(new_data)
```

Example :

```
import csv
# Read existing data
with open('data.csv', mode='r') as file:
    reader = csv.DictReader(file)
    data = [row for row in reader]

# modify data
data.append({'Name': 'Eve', 'Age': 22, 'City': 'Chicago'})

with open('data.csv', mode='w', newline='') as file:
    fieldnames = ['Name', 'Age', 'City']
    writer = csv.DictWriter(file, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow(data)
```

Numpy :

- Numpy is a Python library
- Numpy is used for working with arrays
- Numpy is short for "Numerical Python".

import Numpy

① import numpy

```
arr = numpy.array([1, 2, 3, 4, 5])
print(arr)
```

② import numpy as np

Now the numpy package can be referred to as np instead of numpy

```
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

Commands :-

pip install numpy

Python -m venv myenv

Py -m venv <name>

<name> / scripts / activate

numpy

```
import numpy as np

list = [1, True, "hello", 123]

arr = np.array([1,2,3])
print(arr)

arr2 = np.array((1,2,3,"4"))
print(arr2)

arr3 = np.array(list)
print(arr3)
```

np.zeros()

create an array filled with zeros

```
arr = np.zeros((3,3))
print(arr)
```

(3 rows, 3 columns)

np.ones()

create an array filled with ones

```
arr = np.ones((2,4))
print(arr)
```

np.arange() :-

creates an array with evenly spaced values within a specified range

```
arr = np.arange(0, 10, 2)
print(arr)
```

np.linespace() :-

create an array with evenly spaced numbers over a specified interval

```
arr = np.linspace(1, 10, 4)
print(arr)
```

np.eye()

create an identity matrix (diagonal of ones, others are zeros)

```
arr = np.eye(3)
print(arr)
```

np.random.rand() :-

Generates an array of random number (uniform distribution) between 0 and 1

```
arr = np.random.rand(2,3)
print(arr)
```

np.random.randint()

Generates an array of random integers with in a specified range.

```
arr = np.random.randint(1, 10, (3,3))
print(arr)
```

2, Shape Manipulation Functions :-

np.reshape() :-

```
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped = arr.reshape((2,3))
```

o/p :- [[1, 2, 3],
[4, 5, 6]]

np.flatten():

Flattens a multi-dimensional array into a 1D array

1 Dimensional

```
arr = np.array([[1,2],[3,4]])
flat = arr.flatten()
print(flat)
```

np.transpose()

Transposes the dimensions of an array

```
arr = np.array([[1,2],[3,4]])
transposed = np.transpose(arr)
```

o/p : [[1,3],[2,4]]

③ Mathematical operations :-

np.sum()

calculates the sum of array elements.

```
arr = np.array([1,2,3,4])
print(np.sum(arr))  # 10
```

np.mean():

calculates the average of the elements.

```
arr = np.array([1,2,3,4])
print(np.mean(arr))  # 2.5
```

np.median() :-

finds the median of the array

```
arr = np.array([1,3,2,4])
print(np.median(arr))
```

np.std() :=

Calculates the standard deviation

```
arr = np.array([1,2,3,4])
print(np.std(arr))        # 1.118
```

np.sqrt() :=

Applies square root element-wise.

```
arr = np.array([1,4,9])
print(np.sqrt(arr))       # [1.0 2.0 3.0]
```

np.subtract()

```
l1 = [1,2,3]
l2 = [4,5,6]
arr = np.subtract(l1,l2)
print(arr)
```

np.dot() => Performs matrix multiplication

```
matrix 1 = np.array([[1,2],[3,4]])
matrix 2 = np.array([[5,6],[7,8]])
result = np.dot(matrix1, matrix2)
print(result)
```

④ Array Manipulation Function:

np.concatenate()

Join arrays along an existing axis

```
a = np.array([1, 2])
b = np.array([3,4])
concat = np.concatenate((a,b))      # [1,2,3,4]
```

.np.split()

splits an array into multiple sub-arrays.

```
arr = np.array([1,2,3,4,5,6])
result = np.split(arr, 3)      # [array([1,2]), array([3,4]), array([5,6])]
```

6., logical operations :-

np.any()

checks if any element is True

```
arr = np.array([0,0,1])
Print(np.any(arr))      # True.
```

np.all()

checks if all element are True.

```
arr = np.array([1,1,1])
Print(np.all(arr))      # True.
```

Pandas

Pandas is a powerful open-source library in Python primarily used for data manipulation and analysis. It provides easy-to-use data structure and functions designed to work seamlessly with structure data such as tabular data [similar to SQL tables or Excel spreadsheets].

Key features of Pandas

- Data structures
  - Series : A one-dimensional labeled array.
  - DataFrame : A two-dimensional labeled data structure similar to a table.

```
import Pandas as pd
```

pip install Pandas

①
```
list1 = [1,2,3,4]
ser = pd.Series(list)
Print(ser)
```

Adding custom index : preset index values

```
series = pd.series(list1, index = ["A","B","C","D"]
Print(series)
```

(or) dynamic index value.

```
list2 = list(range(1, len(list1)+1))
series = Pd.Series(list1, index = list2)
print(series)
```

creating a series from a Dictionary :

```
data = {"Alice": 25, "Bob": 30, "charlie": 35}
series = Pd.series(data)
Print(series)
```

o/p :
```
Alice   25
Bob     30
charlie 35
```

Data Frame :=

creating a dataFrame from a Dictionary

```
data = {
    "Name": ["Alice", "Bob", "charlie"],
    "Age" : [25, 30, 35],
    "City" : ["New York", "Los Angeles", "Chicago"]
    }
df = Pd.DataFrame(data)
Print(df)
```

o/p :

	Name	age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	charlie	35	Chicago.

creating a DataFrame from a list of Lists :

```
data = [
    ["Alice", 25, "New York"],
    ["Bob", 30, "Los Angeles"],
    ["charlie", 35, "Chicago"]
]
```

```
df = pd.DataFrame(data, columns=["Name", "Age", "City"])
print(df)
```

o/p :

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

Creating an Empty Data Frame :=

```
empty_df = pd.DataFrame()
print(empty_df)
```

② Viewing and inspecting Data :

Basic Methods :=

Display the first few Rows : head()

```
list = [
  ["ajay", "raju", "ramu"],
  [20, 34, 23],
  [1, 2, 3]
  ]
df = pd.DataFrame(list)
print(df.head(2))
```

o/p :

	0	1	2
0	ajay	raju	ramu
1	20	34	23

Display the last few Rows : tail()

```
list :[
  ["ajay", "raju", "ramu"],
  [20, 34, 23],
  [1, 2, 3]
]
```

```
df = pd.DataFrame(list)
print(df.tail(2))
```

o/p

	0	1	2
0	20	34	23
1	1	2	3

Get data frame information : info()

```
list = [
    ["alok", "raju", "ramu"],
    [20, 34, 23],
    [1, 2, 3]
]
df = pd.DataFrame(list)
print(df.info())
```

o/p :

#	Column	Non Null	Dtype
0	0	3 non null	object
1	1	3 non null	"
2	2	"	"
		0	0

Get summary statistics :

describe()

```
data = [
    ["Alice", 25, "New York"],
    ["Bob", 30, "Los Angeles"],
    ["Charlie", 35, "Chicago"]
]
df = pd.DataFrame(data)
print(df.describe())
```

o/p

Get Column Names : columns

```
data = [
    ["Alice", 25, "NewYork"],
    ["Bob", 30, "LosAngles"],
    ["charlie" 35, "chicago"]
]
df = pd.DataFrame(data)
Print (df.columns)
```

Get index information : index No of rows.

```
data = [
    ["Alice", 25, "NewYork"],
    ["Bob", 30, "LosAngeles"],
    ["charlie" 35, "chicago"]
]
df = pd.DataFrame(data)
Print (df.index)
```

4. Filtering Data

Filtering Rows Based on a condition:

```
data = {
    "Name" : ["Alice", "Bob", "Charlie"],
    "Age"  : [25, 30, 35],
    "City" : ["New York", "Los Angeles", "Chicago"]
}
df = pd.DataFrame(data)
print(df[df["Age"]>25])
print(df[df["City"] == "New York"])
```

df

	Name	Age	City
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

Filtering Rows using Multiple conditions:

```
Filtered = df[(df["City"] == "New York") & (df["Age"]>30)]
print(Filtered)
```

⑤ Modifying Data:

Adding a New Column:

```
data = {
    "Name" : ["Alice", "Bob", "Charlie"],
    "Age"  : [25, 30, 35],
    "City" : ["New York", "Los Angeles", "Chicago"]
}
df = pd.DataFrame(data)
```

```
df["salary"] = [10000, 20000, 30000]
print(df)
```

updating a column :

```
df["salary"] = df["salary"] + 5000
print(df)
```

Delete a column :

```
print(df.drop("salary", axis=1))
```

axis is used to define whether to delete a row or column (axis=0 is for rows) (axis=1) for columns

```
df = df.drop("salary", axis=1)
print(df)
```

Delete a rows :

```
df = df.drop(0, axis=0)
print(df)
```

⑥ sorting data :

sorting by a column :

```
data = {
"name" : ["Alice", "Bob", "charlie"],
"Age" : [25, 30, 35],
"city" : ["Newyork", "Los Angeles", "chicago"],
```

"salary" : [30000, 20000, 30000]
}

df = df.sort_values("city")
print(df)

by default ascending the

(and)

df = df.sort_values("salary", ascending=False)
print(df)

Sorting by Multiple Columns:

df = df.sort_values(["Age", "salary"], ascending=False)
print(df)

⑦ Handling Missing Data:

Detecting Missing values:

df = pd.DataFrame([{"a":10, "b":20}, {"a":10, "b":20, "c":}])
print(df.isnull())

Filling Missing values:

print(df["c"].fillna(0))

Dropping Rows with Missing values:

print(df.dropna())

⑧ Grouping and Aggregating

Grouping Data by a column :

⑨ Combining Data Frames :

Concatenating DataFrames :-

```
df1 = Pd.DataFrame({"A": [1,2], "B": [3,4]})
df2 = Pd.DataFrame({"A": [5,6], "B": [7,8]})
combined = Pd.concat([df1, df2])
Print(combined)
```

Merging DataFrames :

```
df1 = Pd.DataFrame({"Name": ["Alice", "Bob"], "Age": [25,30]})
df2 = Pd.DataFrame({"Name": ["Alice", "Bob"], "City": ["New York", "Chicago"]})
merged = Pd.merge(df1, df2, on = "Name")
Print(merged)
```

⑩ Exporting Data :

Writing to a CSV File :

```
df.to_CSV("output.CSV", index = False)
```

df.to_excel("out.

Matplotlib

Matplotlib is a comprehensive library in Python used for creating static, interactive and animated visualizations. It is particularly useful for data visualization, enabling users to create line plots, scatter plots, bar charts, histograms, pie charts and more.

Key Feature of matplotlib:

- High-quality Plots: Generates publication-quality figures.
- Customizable: Extensive options for customizing plots [colors, labels, legends, etc.]
- Integration: Works well with other Python libraries like Numpy and Pandas.

Commonly used Matplotlib Functions

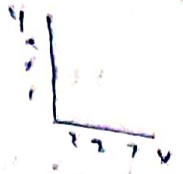
Here's breakdown of the most important functions in Matplotlib, primary from the Pyplot module matplotlib.pyplot.

Commands:

```
Py -m venv matplot
matplot/scripts/activate
pip install matplotlib.
```

import matplotlib.pyplot as plt

1. Basic Plotting Functions.



plot(x, y)

- create a line plot of x vs y.

```
x = [1,2]
y = [4,5]
plt.plot(x,y)
plt.show()
```

(or)

```
x=[1,2,3,4,5]
y=[50,100,50,20,250]
plt.plot(x,y)
plt.show()
```

(2)

```
x= [1,2,3,4]
y= [10,20,25,30]
plt.plot(x,y, color='blue', marker='o')
plt.show()
```

(or)

```
plt.plot(x,y, marker="o", ls="--")
plt.show()
```

(or)

```
plt.plot(x,y, marker="o", ls=":")
plt.show()
```

(or)

```
plt.plot(x,y, marker="o", ls="-.")
plt.show()
```

(or)

```
plt.plot(x,y, marker="o", ls="-.")
plt.grid()
plt.show()
```

(or)

bar chart

```
plt.bar(x,y)
plt.show()
```

2. Customizing Plots :

title(label)

: sets the title of the plot.

```
x = [10, 20, 30, 40, 50]
y = [50, 100, -50, 20, -200]
plt.bar(x,y)
plt.title("sample data")
plt.show()
```

```
x = [10000, 20000, 30000]
y = ["jan", "feb", "mar"]

plt.pie(x)
plt.show()
```

(and)

```
plt.pie(x, labels=y)
plt.show()
```

subplot

```
subplot(nrows, ncols, index)
```

- create a subplot in a grid of nrows x ncols and focuses on the index.

```
x = [1,2,3,4,5]
y = [10,20,30,40,50]

plt.plot(x,y)
plt.subplot(1,2,1)
plt.show()
```

scatter (x, y, ...)

- create a scatter plot of x vs. y

```
x = [1,2,3,4,5]
y = [10,20,30,40,50]
plt.scatter(x,y, color='red')
plt.show()
```

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

x = [10, 20, 30, 40, 50, 60, 70]

plt.title ("samplechart")

plt.xlabel ("x values")

plt.ylabel ("y values")

plt.plot (x, y)

plt.show ()
```

OOPS

Object-oriented Programming

OOPS in Python is a programming paradigm that organizes code around object. objects represent real-world entities and have attributes (data) and methods (function).

OOP helps in building modular, reusable, and maintainable code.

Key OOPS Concept in Python:

1) class

1) A class is a blueprint for creating objects It defines the attributes and methods that its objects will have.

```
class Mobile :
    ram = "8gb"
    rom = "128gb"
    screen = "2K"
    battery = "6000mah"

mob1 = Mobile()          # mob1 is the instance of the Mobile class
print(mob1)
```

* A class can have a attributes and method.

```
class Mobile :
    def __init__(self):      # constructor
        print("constructor is called", self)

mob = Mobile()
print(mob)
```

Constructor (__init__)

- A special method used to initialize a object with attributes when it's created.

① Class Mobile:

```
def __init__(self, ram, rom, screen, cam):
    self.ram = ram
    self.rom = rom
    self.screen = screen
    self.cam = cam
```

→ # Instance attribute

```
mob1 = Mobile("12gb", "256gb", "2k", "108mp")
mob2 = Mobile("16gb", "512gb", "4k", "200mp")

print(mob1.cam)
print(mob2.ram)
```

② Class Mobile:

```
brand = "samsung"
def __init__(self, ram, rom, screen, cam):
    self.ram = ram
    self.rom = rom
    self.screen = screen
    self.cam = cam

def getFullInfo(self):
    return(f"this mobile is having cam of {self.cam}")

mob1 = Mobile("12gb", "256gb", "2k", "108mp")
mob2 = Mobile("16gb", "512gb", "4k", "200mp")
```

```
print (mob1.brand)
print (mob2.brand)
print (mob2.getCamInfo)
```

③ class Account :

```
    def __init__ (self, name, bal):
        self.name = name
        self.balance = bal

    def deposit (self, amount):
        self.balance += amount
        return (f"Amount added, current balance is {self.balance}")

    def withdraw (self, amount):
        if (self.balance >= amount):
            self.balance -= amount
            return (self.balance)
        else :
            return ("insufficient funds")

acc1 = Account ("moni", 10000)
Print (acc1.deposit (5000))
Print (acc1.withdraw (15000))
```

④ Class Mobile :

```
    brand = "xiomi"
    def __init__(self, model):
        self.model = model
        print("Hello from constructor")
    def info(self):
        return(f"model is {self.model}")

m1 = Mobile("ultra")
print(m1.info())
```

2) object :

- An object is an instance of a class. It has attributes and can use methods defined in the class.

① Class car :

```
    wheels = 4          # Attributes

    def start(self):    # Methods
        print("car is starting ...")

my_car = car()          # creating an object
print(my_car.wheels)    # Accessing an attribute.
my_car.start()          # Calling a Method.
```

3) Encapsulation :

- Encapsulation means bundling data (attributes) and methods together in a class. You can restrict direct access to some data using private attributes

```
class BankAccount :
    def __init__(self, balance):
        self.__balance = balance          # Private attribute.

    def deposit(self, amount):
        self.balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount(100)
account.deposit(50)
Print(account.get_balance())   # Accessing balance through a method
```

4) Inheritance :

- Inheritance allows one class (child) to inherit attributes and methods from another class (Parent). It promotes code reuse.

④

```
class Animal:
    def speak(self):
        print("Animal speaks")
class Dog(Animal):          # Dog inherits from Animal
    def bark(self):
        print("Dog barks")
d = Dog()
d.speak()                   # Inherited method
d.bark()                    # Own method
```

⑤ Polymorphism :-

- Polymorphism allows different classes to have methods with the same name, but potentially different behaviors.

```
class Bird:
    def speak(self):
        print("chirp, chirp")
class Cat:
    def speak(self):
        print("Meow")
# Using the same method name
for animal in [Bird(), Cat()]:
    animal.speak
```

⑥ Abstraction :-

- Abstraction hides unnecessary detail and shows only the essential features. In Python you can use abstract classes to achieve abstraction.

```
from abc import ABC, abstractmethod

class shape(ABC):          # Abstract class
    @abstractmethod
    def area(self):
        pass

class circle(shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

c = circle(5)
Print(c.area())
```

⑦ Method overriding :-

- A child class can override a method from the parent class

```
class vehicle:
    def start(self):
        print("vehicle starting...")

class Bike(vehicle):
    def start(self):
        print("Bike starting...")      # overriding the parent method.

b = Bike()
b.start()
```

⑧ Method overloading :

- Python doesn't directly support method overloading. You can achieve similar functionality by using default arguments

```
class calculator:
    def add(self, a, b=0):
        return a+b

c = calculator()
print(c.add(10))        # Single argument
print(c.add(10,20))     # Two argument
```