OOPS CONCEPT IN PYTHON

- Classes is a template/blue-print for real-world entities . Classes is a user defined data type.*
- For example in class Mobile *
- Properties(colour,cost) of mobile are said as Attributes of Mobile class *
- Behaviour(playing games,calling) of Mobile are said as Methods of Mobile class *

- Objects - Objects are specific instances of a class *
- For example in Mobile class *
- Apple,Samsung,Vivo are the specific instance of Mobile class *

In [1]:
```
class Phone:   # Creating the 'Phone' class

    def make_call(self):   # defining method or behaviour of class
        print("Making phone call")

    def play_game(self):   # self is used to assigning the methods of class to th
        print("Playing Game")

p1 = Phone()    ## Instantiating the p1 object

p1.make_call()  ## Invoking methods through object
p1.play_game()
```

```
Making phone call
Playing Game
```

** ADDING PARAMETERS TO THE CLASS**

In [2]:
```
class Phone:

    def set_color(self,color):
        self.color = color

    def set_cost(self,cost):
        self.cost = cost

    def show_color(self):
        return self.color

    def show_cost(self):
        return self.cost

    def make_call(self):
        print("making phone call")

    def play_game(self):
        print("Playing Game")


## Creating Object

p1 = Phone()
p1.set_color("black")
p1.set_cost("20000")
```

In [3]:
```
p1.show_color()
```

Out[3]: 'black'

In [4]:
```
p1.show_cost()
```

Out[4]: '20000'

In [5]:
```
p1.make_call()
```

```
making phone call
```

In [6]:
```
p1.play_game()
```

```
Playing Game
```

** Creating a class with Constructor **

** Constructor is a special type of method.**

**When we create a object at the time of creation we can assign the values **

** In python it is defined as '**init**' *

In [7]:
```
class Employee:

    def __init__(self,name,age,salary,gender):

        self.name = name
        self.age = age
        self.salary = salary
        self.gender = gender

    def employee_details(self):
        print("Name of employee is: ",self.name)
        print("Age of employee is: ",self.age)
        print("Salary of employee is: ",self.salary)
        print("Gender of employee is: ",self.gender)

E1 = Employee('Mr.Sharma',35,60000,'male')
E1.employee_details()
```

```
Name of employee is:  Mr.Sharma
Age of employee is:  35
Salary of employee is:  60000
Gender of employee is:  male
```

INHERITENCE IN PYTHON

** With inheritence on class can derive the properties of another class**

** There are classes like superclass and subclass **

** Properties of superclass are inherited by subclass **

In [8]:
```
## Creating Superclass

class Vehicle:

    def __init__(self,mileage,cost):
        self.mileage = mileage
        self.cost = cost

    def show_details(self):
        print("I am a vehicle")
        print("Mileage of vehice is: ",self.mileage)
        print("Cost of vehicle is: ",self.cost)

# Initializing the object for superclass

v1 = Vehicle(500,500000)
v1.show_details()
```

```
I am a vehicle
Mileage of vehice is:  500
Cost of vehicle is:  500000
```

In [9]:
```
#Creating the Subclass

class Car(Vehicle):
    def show_car(self):
        print("I am a car")

# Intializing the object for subclass

c1 = Car(40,400000)
c1.show_details()
```

```
I am a vehicle
Mileage of vehice is:  40
Cost of vehicle is:  400000
```

In [10]:
```
c1.show_car()
```

```
I am a car
```

OVER-RIDING init METHOD

In [18]:
```
## Creating Superclass

class Vehicle:

    def __init__(self,mileage,cost):
        self.mileage = mileage
        self.cost = cost

    def show_details(self):
        print("I am a vehicle")
        print("Mileage of vehice is: ",self.mileage)
        print("Cost of vehicle is: ",self.cost)
```

In [15]:
```
class Car(Vehicle):

    ## Over riding init method

    def __init__(self,mileage,cost,tyres,hp):
        super().__init__(mileage,cost)  #By using super() we are invoking the met
        self.tyres = tyres
        self.hp = hp

    def show_car_details(self):
        print("I am a car")
        print("Number of tyres are ",self.tyres)
        print("Value of horse power is ",self.hp)
```

In [16]:
```
## Invoking show_details() method from parent class

c2 = Car(20,120000,4,300)
c2.show_details()
```

```
I am a vehicle
Mileage of vehice is:  20
Cost of vehicle is:  120000
```

In [17]:
```
## Invoking show_car_details from child class

c2.show_car_details()
```

```
I am a car
Number of tyres are  4
Value of horse power is  300
```

Types of Inheritence

```
** These are the types of inheritence in python**

* Single Inheritence - In this type child class inherits from parent class
```

```
* Multiple Inheritence -In this type , the child inherits from more than 1 parent class

* Multi-level Inheritence - In this type, we have Parent,child,grand-child relationship

* Hybrid Inheritence
```

MULTIPLE INHERITENCE IN PYTHON

In [20]:
```
## Parent class one

class Parent1():
    def assign_string_one(self,str1):
        self.str1 = str1

    def show_string_one(self):
        return self.str1
```

In [22]:
```
## Parent class two

class Parent2():
    def assign_string_two(self,str2):
        self.str2 = str2

    def show_string_two(self):
        return self.str2
```

In [24]:
```
## Child class

class Derived(Parent1,Parent2):
    def assign_string_three(self,str3):
        self.str3 = str3

    def show_string_three(self):
        return self.str3

## Intializing Child class

d1 = Derived()

d1.assign_string_one("one")
d1.assign_string_two("two")
d1.assign_string_three("three")
```

In [25]:
```
d1.show_string_one()
```

Out[25]: 'one'

In [26]:
```
d1.show_string_two()
```

Out[26]: 'two'

In [27]:
```
d1.show_string_three()
```

Out[27]:
```
'three'
```

MULTI LEVEL INHERITENCE IN PYTHON

In [29]:
```
## Parent class

class Parent():
    def assign_name(self,name):
        self.name = name

    def show_name(self):
        return self.name
```

In [35]:
```
## Cild class

class Child(Parent):
    def assign_age(self,age):
        self.age = age

    def show_age(self):
        return self.age
```

In [36]:
```
## Grand-Child class

class GrandChild(Child):
    def assign_gender(self,gender):
        self.gender = gender

    def show_gender(self):
        return self.gender
```

In [37]:
```
gc = GrandChild()
gc.assign_name("Jasmine")
gc.assign_age('20')
gc.assign_gender('female')
```

In [39]:
```
gc.show_name()
```

Out[39]:
```
'Jasmine'
```

In [40]:
```
gc.show_age()
```

Out[40]:
```
'20'
```

In [41]:
```
gc.show_gender()
```

Out[41]:
```
'female'
```

ENCAPSULATION

Encapsulation refers to wrapping a data in a single unit and it is a mechanism that binds
code and the data it manipulates.

Or, It is a protective sheild that prevents the data from being accessed by the code outside this
shield and in this the variable or data of a class is hidden from any other class and can be accessed
only through any member function of the own class in which that they are declared.

** Benefits of ENCAPSULATION **

* Data Hiding
* Flexibility
* Reusability

In [45]:
```
class Student:

    def __init__(self,name):

        #Object Attribute
        self.__name = name

    def display(self):
        print('name =',self.__name)

## Creating object of Student class

obj = Student('Jhon')

## Accessing the attribute from class method

obj.display()

## Accessing the attribute from outside

print('name =',obj.__name)
```

```
name = Jhon

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [45], in <cell line: 21>()
     17 obj.display()
     19 ## Accessing the attribute from outside
---> 21 print('name =',obj.__name)

AttributeError: 'Student' object has no attribute '__name'
```

Here double underscore is the prefix of name attribute denotes that it is a private member

private member
soo it can be accessed from display() method but not from outside, so it will raise
attribute error for obj.__name

POLYMORPHISM

Polymorphism is the ability to take more than one ctasks. It is achieved by using
function overloading in which a function can have same name but different functionalities

In [48]:
```
class Add1:
    def __init__(self):
        # object attributes
        self.x = 10
        self.y = 20
    def add(self):
        #add integer values
        addition = self.x + self.y
        print(f'Addition of (self.x) and (self.y) is : {addition}')
class Add2(Add1):
    def __init__(self):
        # object attributes
        self.x = 15.89
        self.y = 30.67
    def add(self):
        #add float values
        addition = self.x + self.y
        print(f'Addition of (self.x) and (self.y) is : {addition}')

# Create objects
obj1 = Add1()
obj2 = Add2()
# call add method
obj1.add()
obj2.add()
```

```
Addition of (self.x) and (self.y) is : 30
Addition of (self.x) and (self.y) is : 46.56
```