Creating class and object: example

class Person: #Capital letter of first word of class, this style is known as camel case. def **init**(self,name,age,company): #instance attribute and constructor self.name=name self.age=age self.company=company

In [2]:
```
#lets create first object because without object, programming will not run

person1=Person("nick",21,"FaceBook")
```

In [3]:
```
print(person1.age)
```

```
21
```

In [4]:
```
print(person1.__dict__)
```

```
{'name': 'nick', 'age': 21, 'company': 'FaceBook'}
```

In [5]:
```
#lets create class attribute
#Creating class and object: example

class Person:  #Capital letter of first word of class, this style is known as camel case.

    country="USA"        #class variable

    def __init__(self,name,age,company):        #instance attribute and constructor
        self.name=name
        self.age=age
        self.company=company

person2=Person("Slim",34,"instagram")
person1=Person("nick",21,"FaceBook")
```

In [6]:
```
print(person1.age)
print(person2.name)
```

```
21
Slim
```

In [7]:
```
#But, class varibale can be use by any object
print(person1.country)
print(person2.country)
```

```
USA
USA
```

In [8]:
```
#Function/Methods
```

In [15]:
```
class Person:  #Capital letter of first word of class, this style is known as camel case.

    country="USA"        #class variable

    def __init__(self,name,age,company):        #instance attribute and constructor
        self.name=name
        self.age=age
        self.company=company

    def display(self):    # methods
        print(f'{self.name} is my name and my age is {self.age}, I work in {self.company}')

person2=Person("Slim",34,"instagram")
person1=Person("nick",21,"FaceBook")
```

In [16]:
```
person2.display()
```

```
Slim is my name and my age is 34, I work in instagram
```

In [18]:
```
person1.display()
```

```
nick is my name and my age is 21, I work in FaceBook
```

In [19]:
```
#Class inheritance in python
```

Inheritance is a process by which a class takes on the attributes and methods of another class. However, the class can also have its own attributes and methods.

In the case of inheritance, the original class is referred to as the parent class, while the class that inherits is referred to as the child class.

In [8]:
```
class Company:         #parent class

    name="ITCompany"

    def __init__(self,numofdept=int,address=str,avgsalary=int):
```

```
        self.numofdept=numofdept
        self.address=address
        self.avgsalary=avgsalary

    def display(self):
        print("Company name is",self.name,"and number of department is",self.numofdept)

class Department(Company):        #child class
    def __init__(self,numofdept,address,avgsalary,deptname):
        self.deptname=deptname
        super().__init__(numofdept,address,avgsalary)

    def method1(self):
        print("inheritance")
```

In [9]:
```
#object

hr=Department(1,"Baneshwor",40000,"HR")
```

In [27]:
```
print(hr.avgsalary)
```

```
40000
```

In [28]:
```
hr.display()
```

```
Company name is ITCompany and number of department is 1
```

In [30]:
```
hr.method1()
```

```
inheritance
```

In [31]:
```
#polymorphism in python
```

The concept of polymorphism builds on the concept of inheritance. While it can be helpful to define child objects, these child objects may operate slightly differently. Polymorphism allows you to define a child object but create and use its own methods.

In [23]:
```
class Person:
    def __init__(self,name,gender):
        self.name=name
        self.gender=gender

    def display(self):
        print("I am a person and I can be any gender type")

class Man(Person):
    def __init__(self,name,gender,age):
        self.age=age
        super().__init__(name,gender)

    def display(self):
        print("I am a Man and my gender is male")

class Women(Person):
    def __init__(self,name,gender,salary):
        self.salary=salary
        super().__init__(name,gender)

    def display(self):
        print("I am a women and my gender is female")
```

In [2]:
```
#Here we use inheritance and also 3 display method to show example of polymorphism
```

In [24]:
```
#create object
person1=Person("jack","Male")
```

In [25]:
```
man1=Man("Dean","Male",21)
```

In [27]:
```
man1.display()
```

```
I am a Man and my gender is male
```

In [28]:
```
girl1=Women("rita","female",2000)
```

In [29]:
```
girl1.display()
```

```
I am a women and my gender is female
```

In [30]:
```
#see different class method is behaving differently, however name of all method is same. This is called overrid
```

Overriding: Overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The method name and the number and type of its parameters are the same in both the superclass and the subclass. In your code, the display method is being overridden in the Man and Women

In [31]:
```
#encapusltaion in Python
```

In [31]: #encapsulation in Python

The idea behind encapsulation is that all properties and methods of an object are kept private and safe from being inherited by another object. This allows you to define both public and private methods and attributes.

In [32]:
```
class Vechicle:
    def __init__(self,types,milege,cost):
        self.types=types          #public attribute
        self.milege=milege
        self.__cost=cost            #private attribute

    def display(self):
        print("type of Vechile is",self.types)
        print("Milege of Vechicle is",self.milege)
        print("cost of vechicle is",self.__cost)
```

In [33]:
```
#create object
vechicle1=Vechicle("Car",2000,300000)
```

In [34]:
```
print(vechicle1.types)
```

```
Car
```

In [43]:
```
print(vechicle1.cost)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[43], line 1
----> 1 print(vechicle1.cost)

AttributeError: 'Vechicle' object has no attribute 'cost'
```

In [36]:
```
#its does not shows because it is private attribute, it is make private because accidental error may not occur
```

In [37]:
```
#anyway, we can also access private attribute, we have seen in previous tutorial, noting in python is exactly p
```

In [39]:
```
print(vechicle1.milege)
```

```
2000
```

In [40]:
```
vechicle1.display()
```

```
type of Vechile is Car
Milege of Vechicle is 2000
cost of vechicle is 300000
```

In [44]:
```
#But, may get confused, in display method cost get printed, how?
```

We can print the __cost attribute in the display method because we are accessing it from within the same class. Private attributes in Python can still be accessed within the class they are defined in.

In [46]:
```
#But we can print cost directly with object too by name mangling:
print(vechicle1._Vechicle__cost)
```

```
300000
```

In [47]:
```
#now, in the same way let see what is @property
```

In [1]:
```
class Vechicle:
    def __init__(self,types,milege,cost):
        self.types=types          #public attribute
        self.milege=milege
        self.__cost=cost            #private attribute

    def display(self):
        print("type of Vechile is",self.types)
        print("Milege of Vechicle is",self.milege)
        print("cost of vechicle is",self.__cost)

    @property
    def cost(self):
        return self.__cost

    @cost.setter
    def cost(self, value):
        if value >= 0:
            self.__cost = value
        else:
            print("Cost cannot be negative.")
```

In [2]:
```
vechicle1=Vechicle("Car",2000,300000)
```

In [3]:
```
print(vechicle1.cost)
```

```
300000
```

In [4]:
```
#see, now we do not need to do name mangling, client and user can directly access it.
```

In [5]:
```
#so do we need if else in setter?
class Vechicle:
    def __init__(self,types,milege,cost):
        self.types=types          #public attribute
        self.milege=milege
        self.__cost=cost            #private attribute

    def display(self):
        print("type of Vechile is",self.types)
        print("Milege of Vechicle is",self.milege)
        print("cost of vechicle is",self.__cost)

    @property
    def cost(self):
        return self.__cost

    @cost.setter
    def cost(self, value):
        self.__cost = value
```

In [6]:
```
vechicle3=Vechicle("Honda",400,1000)
```

In [7]:
```
print(vechicle3.cost)
```

```
1000
```

In [8]:
```
#no we do not need it.
```

In [9]:
```
print(vechicle3.__dict__)
```

```
{'types': 'Honda', 'milege': 400, '_Vechicle__cost': 1000}
```