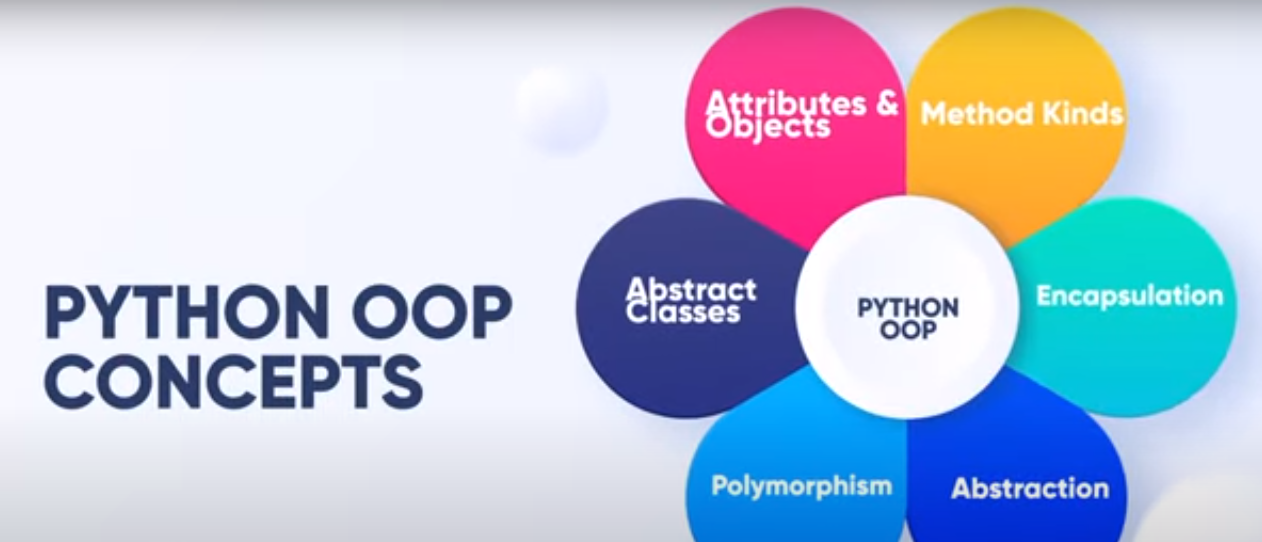
**OOP’s in python**

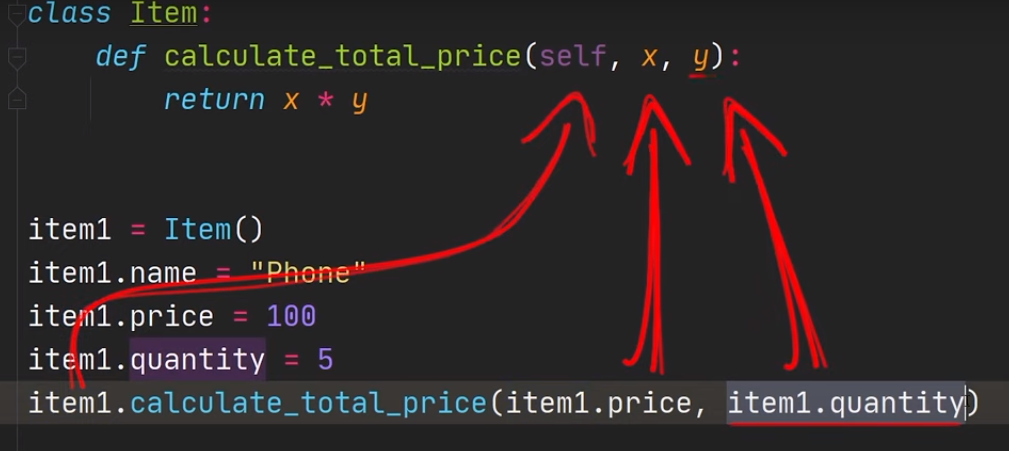
**Jim – jimShapeCoding-Youtube freeCodeCamp**

**(20-10-2023)**

Oops concepts:



Arrguments passing to class methos:



Constructor :

* Method that executes when we created the object for the class.
* It is called \_\_init\_\_ menthod.

Syntax code:

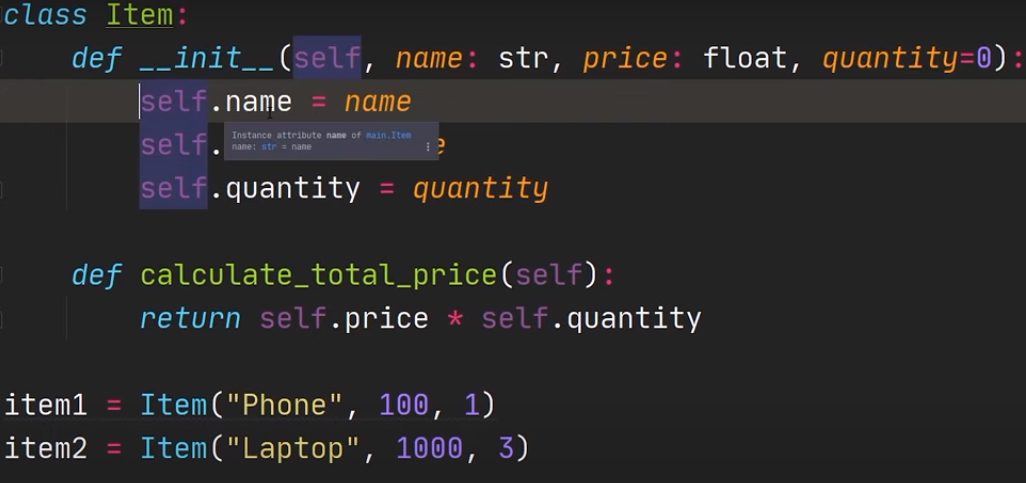
* class Item():
* #constructor for creating the variable and assigning the values.
* #it will execute while creating the object or instances
* def \_\_init\_\_(self,name,price,quan):
* self.name=name
* self.price=price
* self.quan=quan
* def get\_total\_price(self):
* return self.price\*self.quan
* item1=Item("iphone",40000,27)
* total=item1.get\_total\_price()
* print(total)
* item2=Item("samsung",23000,30)
* tota2=item2.get\_total\_price()
* print(total)

passing arguments:

* if you not passing the value for the parameter. You have to set the default value.
* class Item():
* # if you not passing the value while creating the object than it will take the default value
* def \_\_init\_\_(self,name=None,price=0,quan=0):
* self.name=name
* self.price=price
* self.quan=quan
* def get\_total\_price(self):
* return self.price\*self.quan
* item1=Item("iphone",40000)
* total=item1.get\_total\_price()
* print(total)
* item2=Item("samsung",30)
* tota2=item2.get\_total\_price()
* print(total)
* we can also add the variables after used cinstrctor also.
* class Item():
* # if you not passing the value while creating the object than it will take the default value
* def \_\_init\_\_(self,name=None,price=0,quan=0):
* self.name=name
* self.price=price
* self.quan=quan
* def get\_total\_price(self):
* return self.price\*self.quan
* item1=Item("iphone",40000)
* item1.colors="blue"
* total=item1.get\_total\_price()
* print(total)
* item2=Item("samsung",30)
* item1.colors="red"# here we adding the variable after initializing values
* tota2=item2.get\_total\_price()
* print(total)

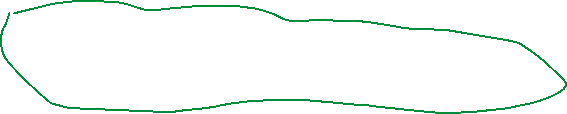
Validating the arguments data types:

* in this code the : is used to specify the data types also default value type is taken as type.



Assertion (validating the values):

* using assert statement key word we can validate data and data types
* class Item():
* def \_\_init\_\_(self,name:str,price:int,quan=0):
* #Assert validation
* assert type(name)==str,"name type should be string"
* assert type(price)==int and price>=0,"price should be int and greater than 0"



* assert type(quan)==int and quan>=0,"quantity should be int and greater than 0"
* self.name=name
* self.price=price
* self.quan=quan
* def get\_total(self):
* return self.price\*self.quan
* item1=Item("iphone",50000,-9)
* total=item1.get\_total()
* print(total)
* item2=Item("iphone",50000,5)
* total=item2.get\_total()
* print(total)

class level attributes (variables):

* class level attributes is can be accessed across all the instances. By same value.
* It is like a global variable to the all instances
* class Item():
* pay\_rate=0.8 #class attributes. accessed by all instances
* def \_\_init\_\_(self,name:str,price:int,quan=0):
* #Assert validation
* assert type(name)==str,"name type should be string"
* assert type(price)==int and price>=0,"price should be int and greater than 0"
* assert type(quan)==int and quan>=0,"quantity should be int and greater than 0"
* self.name=name
* self.price=price
* self.quan=quan
* def get\_total(self):
* return self.price\*self.quan
* item1=Item("iphone",50000,-9)
* item2=Item("iphone",50000,5)
* print(item1.pay\_rate)# both instances can be accesed the attributes
* print(item2.pay\_rate)
* print(Item.\_\_dict\_\_)# it is used to print all class attributes that avilable
* #\_\_dict\_\_ is called magic methodes (inbult methods). it converts the all variable to dictonaries.
* print(item1.\_\_dict\_\_)# it is used to print all instances attributes
* print(item2.\_\_dict\_\_)

out put:

0.8

0.8

{'\_\_module\_\_': '\_\_main\_\_', 'pay\_rate': 0.8, '\_\_init\_\_': <function Item.\_\_init\_\_ at 0x0000020956BD8EA0>, 'get\_total': <function Item.get\_total at 0x0000020956BD9DA0>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Item' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Item' objects>, '\_\_doc\_\_': None}

{'name': 'iphone', 'price': 50000, 'quan': 9}

{'name': 'iphone', 'price': 50000, 'quan': 5}

Dynamic class attributes:

* we can change the class attributes values dynamically.

Code:

class Item():

pay\_rate=0.8 #class attributes. accessed by all instances

def \_\_init\_\_(self,name:str,price:int,quan=0):

#Assert validation

assert type(name)==str,"name type should be string"

assert type(price)==int and price>=0,"price should be int and greater than 0"

assert type(quan)==int and quan>=0,"quantity should be int and greater than 0"

self.name=name

self.price=price

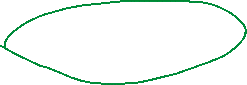
self.quan=quan

def get\_total(self):

return self.price\*self.quan

def get\_price(self):

return self.price\*self.pay\_rate# here the pay rate is taken in class level first isf that is modified in instances level that it takes that.



item1=Item("iphone",100,9)

print(item1.get\_price())

item2=Item("iphone",100,5)

item2.pay\_rate=0.9

print(item2.get\_price())

Out put:

80.0

90.0

Tracking of instances:

* We can have a list and track the instances that created.

Code :

class Item():

pay\_rate=0.8 #class attributes. accessed by all instances

all\_instances=[]



def \_\_init\_\_(self,name:str,price:int,quan=0):

#Assert validation

assert type(name)==str,"name type should be string"

assert type(price)==int and price>=0,"price should be int and greater than 0"

assert type(quan)==int and quan>=0,"quantity should be int and greater than 0"

#assiging the valus to attributes

self.name=name

self.price=price

self.quan=quan

#do some addtional actions

Item.all\_instances.append(self)#appending the instances into list every time

def get\_total(self):

return self.price\*self.quan

#we can control the what should be used by self or object name

def \_\_repr\_\_(self):

return self.name

item1=Item("iphone",50000,9)

item2=Item("samsung",10000,5)

item3=Item("laptop",30000,5)

item4=Item("glass",200,5)

item5=Item("pen",5,5)

print(Item.all\_instances)# this line will print the instances adresses

for item in Item.all\_instances:#this can print all instances name

print(item.name)

out:

[<\_\_main\_\_.Item object at 0x000001CAE2A7F550>, <\_\_main\_\_.Item object at 0x000001CAE2A7F590>, <\_\_main\_\_.Item object at 0x000001CAE2A7F5D0>, <\_\_main\_\_.Item object at 0x000001CAE2A7F750>, <\_\_main\_\_.Item object at 0x000001CAE2A7F790>]

iphone

samsung

laptop

glass

pen

class methode :

* Class methods are used to call by the class name. not for instances name.

class Item():

pay\_rate=0.8 #class attributes. accessed by all instances

all\_instances=[]

def \_\_init\_\_(self,name:str,price:int,quan=0):

#Assert validation

assert type(name)==str,"name type should be string"

assert type(price)==int and price>=0,"price should be int and greater than 0"

assert type(quan)==int and quan>=0,"quantity should be int and greater than 0"

#assiging the valus to attributes

self.name=name

self.price=price

self.quan=quan

#do some addtional actions

Item.all\_instances.append(self)#appending the instances into list every time

def get\_total(self):

return self.price\*self.quan

#we can control the what should be used by self or object name

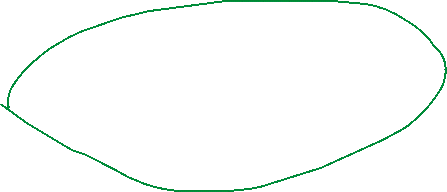
def \_\_repr\_\_(self):

return self.name

@classmethod#it is used to call by class name or class method

def create\_instances(cls,data):

i=0



for \_ in data:

Item(data["name"][i],

data["price"][i],

data["quan"][i]

)

i+=1

data={"name":["iphone","mobile","laptop"],

"price":[100,400,500],

"quan":[4,10,30]

}

Item.create\_instances(data)

print(Item.all\_instances)# this line will print the instances name

Out put:

[iphone, mobile, laptop]

**(31-10-2023)**

Ex 2 for class method (accessing the data from csv file):

Code :

import csv

class Item():

pay\_rate=0.8 #class attributes. accessed by all instances

all\_instances=[]

def \_\_init\_\_(self,name:str,price:int,quan=0):

#Assert validation

assert type(name)==str,"name type should be string"

assert type(price)==int and price>=0,"price should be int and greater than 0"

assert type(quan)==int and quan>=0,"quantity should be int and greater than 0"

#assiging the valus to attributes

self.name=name

self.price=price

self.quan=quan

print(self)

#do some addtional actions

Item.all\_instances.append(self)#appending the instances into list every time

def get\_total(self):

return self.price\*self.quan

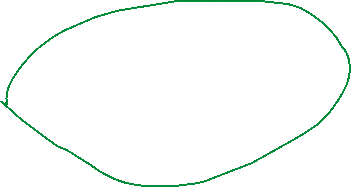
#we can control the what should be used by self or object name

def \_\_repr\_\_(self):

return self.name

@classmethod#it is used to call by class name or class method

def create\_instances(cls,data):



for i in data:

Item(i[0],int(i[1]),int(i[2]))

print("completed")

data=[]

with open('F:\PYTHON PRACTICE\python cource\oops in python\oops\_python\data.csv','r') as dt:

file=csv.DictReader(dt)

field=file.fieldnames

for i in file:

data.append([i[field[0]],i[field[1]],i[field[2]]])

Item.create\_instances(data)

print(Item.all\_instances[0].price)

out:

iphone

samsung

nokia

blackbery

lava

completed

2000

Static method:

* This method is defined using @staticmethod on top of the functions.
* The static method is used to call by class name. which is not belonging to any instances.
* It like a regular function not accepting any compulsory aruguments.

Code:

import csv

class Item():

pay\_rate=0.8 #class attributes. accessed by all instances

all\_instances=[]

def \_\_init\_\_(self,name:str,price:int,quan=0):

#Assert validation

assert type(name)==str,"name type should be string"

assert type(price)==int and price>=0,"price should be int and greater than 0"

assert type(quan)==int and quan>=0,"quantity should be int and greater than 0"

#assiging the valus to attributes

self.name=name

self.price=price

self.quan=quan

print(self)

#do some addtional actions

Item.all\_instances.append(self)#appending the instances into list every time

def get\_total(self):

return self.price\*self.quan

#we can control the what should be used by self or object name

def \_\_repr\_\_(self):

return self.name

@classmethod#it is used to call by class name or class method

def create\_instances(cls,data):

for i in data:

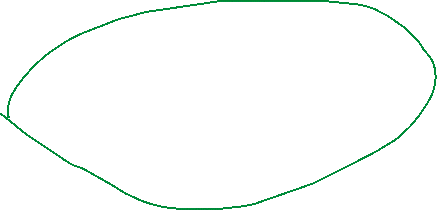
Item(i[0],int(i[1]),int(i[2]))

print("completed")

# static methode is checking the value is integer or not

@staticmethod

def val\_int(num):



if isinstance(num,float):

if num.is\_integer():

return True

else:

return False

else:

return True

#getting the datas from the csv file

data=[]

with open("F:\PYTHON PRACTICE\python cource\oops in python\oops\_python\data.csv",'r') as dt:

file=csv.DictReader(dt)

field=file.fieldnames

for i in file:

data.append([i[field[0]],i[field[1]],i[field[2]]])

# creating the instances for all the datas in csv file

Item.create\_instances(data)

# accessing the particular instances atributes using list of instances

print(Item.all\_instances[0].price)

print(Item.val\_int(5.1)) # calling the static method.

Out:

iphone

samsung

nokia

blackbery

lava

completed

2000

False

Different between @staticmethod and @classmethos:

In Python, both @classmethod and @staticmethod are decorators used to define methods within a class, but they serve different purposes and have distinct use cases:

@classmethod:

A @classmethod is used to define a method that operates on the class itself, rather than on instances of the class.

It takes the class as its first argument (usually named cls) and can access and modify class-level attributes and call other class methods.

Class methods are often used for alternative constructors or for methods that need to access or manipulate class-level data.

@staticmethod:

A @staticmethod is used to define a method that does not depend on the class or its instances. It is a regular function defined within the class's namespace.

Static methods do not receive any special first argument (neither self nor cls). They are mainly used for utility functions that are related to the class but do not require access to instance-specific or class-specific data.

In summary:

Use @classmethod when you need to define methods that operate on class-level data or when you want to create alternative constructors.

Use @staticmethod when you want to define utility methods that are related to the class but do not depend on instance-specific or class-specific data.

The choice between class methods and static methods depends on the specific requirements of your class and the behavior you want to achieve.

Inheritance:

* Using inheritance we can use the repeated functionalites of other class to the another class.
* We can add another class to add extra methodes or attributes which can also use the parent class functionalites.

Code:

# class for getting items

class Item():

pay\_rate=0.8 #class attributes. accessed by all instances

all\_instances=[]

def \_\_init\_\_(self,name:str,price:int,quan=0):

#Assert validation

assert type(name)==str,"name type should be string"

assert type(price)==int and price>=0,"price should be int and greater than 0"

assert type(quan)==int and quan>=0,"quantity should be int and greater than 0"

#assiging the valus to attributes

self.name=name

self.price=price

self.quan=quan

#do some addtional actions

Item.all\_instances.append(self)#appending the instances into list every time

def get\_total(self):

return self.price\*self.quan

# static methode is checking the value is integer or not

@staticmethod

def val\_int(num):

if isinstance(num,float):

if num.is\_integer():

return True

else:

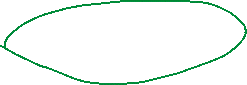
return False

else:

return True

# class for adding extra method for phone only

class Phone(Item):



total\_phones=0

def \_\_init\_\_(self,name:str,price:int,quan=0,broken\_phones=0):

super().\_\_init\_\_(name,price,quan)

assert broken\_phones>=0 and type(broken\_phones)==int,"broken phone should be int"

self.broken=broken\_phones



Phone.total\_phones+=1

def get\_good\_phones(self):

return self.quan-self.broken

# using class method to calculate all objects broken phone

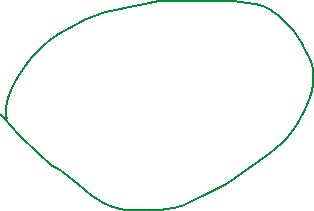
@classmethod

def get\_total\_broken(cls):

total=0

for i in Phone.all\_instances:

try:



total+=i.broken

except:

pass

return total

i1=Item("laptop",3000,20)

i2=Item("bike",30000,10)

i3=Item("car",4000,30)

# printing the name atribute by object reference

for i in Item.all\_instances:

print(i.name)

p1=Phone("lava",3000,4,2)

p2=Phone("micro",3000,12,3)

p3=Phone("iphone",4000,40,4)

#using the same methods in two classes

print(Phone.all\_instances)

print(Phone.get\_good\_phones(p1))

print(Phone.get\_total\_broken())#printing the total broken phones

Out:

laptop

bike

car

[<\_\_main\_\_.Item object at 0x000001CE669E88D0>, <\_\_main\_\_.Item object at 0x000001CE669E8910>, <\_\_main\_\_.Item object at 0x000001CE669E8950>, <\_\_main\_\_.Phone object at 0x000001CE669E8A10>, <\_\_main\_\_.Phone object at 0x000001CE669E8A50>, <\_\_main\_\_.Phone object at 0x000001CE669E89D0>]

2

9

Encapsulation:

* Encapsulation helps in hiding the internal details of an object from the outside world and providing a controlled interface for interacting with that object. In Python, you can achieve encapsulation using classes and access modifiers.
* Access Modifiers: Python provides three access modifiers to control the visibility of class members (attributes and methods):
* Public (default): Members are accessible from anywhere.
* Protected (\_ prefix): Members should not be accessed outside the class, but it's not enforced by the language.
* Private (\_\_ double underscore prefix): Members should not be accessed outside the class, and their names are name-mangled (name mangling is a mechanism for making the name more unique).

Note:

* In your provided code, you have implemented encapsulation using single and double underscores to indicate protected and private attributes. However, it's important to note that Python's name mangling for private attributes isn't preventing you from accessing and modifying the attribute. The double underscore name mangling is mainly a mechanism to avoid accidental name clashes in the inheritance chain.
* Let me clarify a few points from your code:
* Protected Attribute with a Single Underscore:
* A single underscore prefix (e.g., \_name) is a common convention to indicate that an attribute is intended to be protected. However, it doesn't provide strong encapsulation; it's more of a naming convention.
* You can still access and modify \_name from outside the class if you choose to do so. It's not truly private.
* Private Attribute with Double Underscores:
* Attributes with a double underscore prefix (e.g., \_\_price) undergo name mangling. This means their names are changed to include the class name to prevent accidental name clashes in the inheritance chain.
* While it makes it slightly harder to access the attribute from outside the class, it's not entirely private, and you can still access it using the mangled name, as you've observed.

Code :

# class for getting items

class Item():

pay\_rate=0.8 #class attributes. accessed by all instances

all\_instances=[]

def \_\_init\_\_(self,name:str,price:int,quan=0):

#Assert validation

assert type(name)==str,"name type should be string"

assert type(price)==int and price>=0,"price should be int and greater than 0"

assert type(quan)==in t and quan>=0,"quantity should be int and greater than 0"

#assiging the valus to attributes

self.\_name=name# making this as a encpsulated attribute by adding \_ at front(protected)

self.\_\_price=price# making this as private atribute using \_\_

self.quan=quan

#do some addtional actions

Item.all\_instances.append(self)#appending the instances into list every time

# making the name attribute as a encpsualted attri using @property decorater.

@property

def name(self):

return self.\_name

#we can able to protect the assigning the value.

@property

def price(self):

return self.\_\_price

@price.setter

def price(self,value):

self.\_\_price=value

def get\_total(self):

return self.price\*self.quan

# static methode is checking the value is integer or not

@staticmethod

def val\_int(num):

if isinstance(num,float):

if num.is\_integer():

return True

else:

return False

else:

return True

i1=Item("laptop",3000,20)

i2=Item("bike",30000,10)

# we can change other attributes.

i1.price=5000# if we use \_\_ it will not change the value.

print(i1.price)

# but we cant able to change the encpsulated attribute value

#i1.name="lap2" # throws error

# printing the encpsulated attributes asusual

i1.\_name="changed"# single under score will change the value.

print(i1.name)

print(i2.name)

Out:

3000

5000

changed

bike

Abstraction:

* Abstraction is one of the fundamental concepts in Object-Oriented Programming (OOP) and is supported in Python. Abstraction involves simplifying complex reality by modeling classes based on real-world objects or concepts. It allows you to hide the complex implementation details of an object and only expose the essential features and functionalities that are relevant to the user.

Code:

class Item:

def \_\_init\_\_(self,name:str,price:int,quan:int):

assert type(name)==str," name should be string"

assert type(price)==int," price should be int"

assert type(quan)==int," quantity should be int"

self.name=name

self.price=price

self.quan=quan

def get\_price(self):

return self.price

# by using \_\_ we can make method as a private one.

def \_\_connect(self):

print(f"""

conected to email server

for {self.name} product""")

def \_\_send(self):

print("sent")

def send\_mail(self):

# the methods are only accessed inside the class

self.\_\_connect()

self.\_\_send()

i1=Item("mobile",30000,5)

i2=Item("laptop",50000,7)

i3=Item("car",40000,2)

i1.send\_mail()

Out:

conected to email server

for mobile product

sent

Polymorphism:

In Python, polymorphism is one of the fundamental concepts of object-oriented programming (OOP). It allows objects of different classes to be treated as objects of a common superclass, which simplifies code and makes it more flexible. Polymorphism is often achieved through the use of method overriding and method overloading.

Here are the two main types of polymorphism in Python:

Compile-Time (Static) Polymorphism:

This is also known as method overloading. It occurs when two or more methods in the same class have the same name but different parameters (the number or types of parameters). Python does not support method overloading like some other languages (e.g., Java or C++), where you can define multiple methods with the same name but different parameter lists. However, you can achieve similar behavior by using default arguments or variable-length argument lists. Here's an example:

Code:

class Calculator:

def add(self, a, b):

return a + b

def add(self, a, b, c):

return a + b + c

calc = Calculator()

result = calc.add(2, 3) # This will raise a TypeError

Run-Time (Dynamic) Polymorphism:

This is also known as method overriding. It occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The overridden method in the subclass should have the same name, the same number of parameters, and compatible types. Here's an example:

Code:

class Animal:

def speak(self):

pass

class Dog(Animal):

def speak(self):

return "Woof!"

class Cat(Animal):

def speak(self):

return "Meow!"

def animal\_sound(animal):

return animal.speak()

dog = Dog()

cat = Cat()

print(animal\_sound(dog)) # Outputs "Woof!"

print(animal\_sound(cat)) # Outputs "Meow!"

In this example, the animal\_sound function takes an instance of the Animal class as a parameter, and it can work with both Dog and Cat instances due to polymorphism.

Polymorphism is a powerful concept in object-oriented programming, as it allows for more flexible and extensible code by allowing different classes to share common interfaces and be used interchangeably when appropriate.