Python OOPS Inheritance,Polymorphism and Abstraction

Class Relationship

- Aggregation - HAS a relationship
- Inheritance
- example - customer has a address

In [19]:

```
# example
class Customer:

  def __init__(self,name,gender,address):
    self.name = name
    self.gender = gender
    self.address = address

  def print_address(self):
    print(self.address.get_city(),self.address.pin,self.address.state)

  def edit_profile(self,new_name,new_city,new_pin,new_state):
    self.name = new_name
    self.address.edit_address(new_city,new_pin,new_state)

class Address:

  def __init__(self,city,pin,state):
      self.__city = city
      self.pin = pin
      self.state = state

  def get_city(self):
    return self.__city

  def edit_address(self,new_city,new_pin,new_state):
    self.__city = new_city
    self.pin = new_pin
    self.state = new_state

add1 = Address('gurgaon',122011,'haryana')
cust = Customer('nitish','male',add1)

cust.print_address()

cust.edit_profile('ankit','mumbai',111111,'maharastra')
cust.print_address()
# method example
# what about private attribute
```

```
gurgaon 122011 haryana
mumbai 111111 maharastra
```

In aggregation if any attribute become private it cannot be access by another class

In that case we create getter method for another class

In [ ]:

```
+------------------+               +-----------------+
|    Customer      |<>------------>|    Address      |
+------------------+               +-----------------+
| - name: str      |               | - __city: str   |
| - gender: str    |               | - pin: int      |
| - address: Address|              | - state: str    |
+------------------+               +-----------------+
| + print_address(): void          | + get_city(): str |
| + edit_profile(...): void        | + edit_address(...): void |
+------------------+               +-----------------+
```

Inheritance

- Inheritance is an OOP's concept, in which there is a parent class and can have multiple child class, where child inheret the methods and attribute of parent class. It reduce the use of code resuability.

In [48]:

```
# Example

# parent
class User:

  def __init__(self):
    self.name = 'nitish'
    self.gender = 'male'

  def login(self):
    print('login')

# child
class Student(User):

  def __init__(self):
    super().__init__()
    self.rollno = 100

  def enroll(self):
    print('enroll into the course')

u = User()
s = Student()

print(s.name)
s.login()
s.enroll()
```

```
nitish
login
enroll into the course
```

In [ ]:

```
+------------------+               +-------------------+
|       User       |<--------------|     Student       |
+------------------+               +-------------------+
| - name: str      |               | - rollno: int      |
| - gender: str    |               +-------------------+
+------------------+               | + enroll(): void  |
| + login(): void |           +------------------+
+------------------+
```

What gets inherited?

- Constructor
- Non Private Attributes
- Non Private Methods

In [51]:

```python
# constructor example

class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass

s=SmartPhone(20000, "Apple", 13)
s.buy()
```

```
Inside phone constructor
Buying a phone
```

In [ ]:

```python
# constructor example 2

class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

class SmartPhone(Phone):
    def __init__(self, os, ram):
        self.os = os
        self.ram = ram
        print ("Inside SmartPhone constructor")

s=SmartPhone("Android", 2)
s.brand
# will throw error since we are not callinf Phone class constructor so brand never get intialized
```

In [ ]:

```
# child can't access private members of the class

class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    #getter
    def show(self):
        print (self.__price)

class SmartPhone(Phone):
    def check(self):
        print(self.__price)

s=SmartPhone(20000, "Apple", 13)
s.show()
s.check() # throw error because it cannot access private variable of parent class
```

In [57]:

```
class Parent:

    def __init__(self,num):
        self.__num=num

    def get_num(self):
        return self.__num

class Child(Parent):
    def show(self):
        print("This is in child class")

son=Child(100)
print(son.get_num())
son.show()
```

```
100
This is in child class
```

In [ ]:

```
class Parent:

    def __init__(self,num):
        self.__num=num

    def get_num(self):
        return self.__num

class Child(Parent):

    def __init__(self,val,num):
        self.__val=val

    def get_val(self):
        return self.__val

son=Child(100,10)
print("Parent: Num:",son.get_num()) # will throw error
print("Child: Val:",son.get_val())
```

In [59]:

```
class A:
    def __init__(self):
        self.var1=100

    def display1(self,var1):
        print("class A :", self.var1)
class B(A):

    def display2(self,var1):
        print("class B :", self.var1)
```

```
obj=B()
obj.display1(200)
```

```
class A : 100
```

In [61]:

```
# Method Overriding
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    def buy(self):
        print ("Buying a smartphone")

s=SmartPhone(20000, "Apple", 13)

s.buy()
```

```
Inside phone constructor
Buying a smartphone
```

Super Keyword

- Super keyword is a way to access parent method

In [65]:

```
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    def buy(self):
        print ("Buying a smartphone")
        # syntax to call parent ka buy method
        super().buy()

s=SmartPhone(20000, "Apple", 13)

s.buy()
```

```
Inside phone constructor
Buying a smartphone
Buying a phone
```

We cannot use Super outside the class

we cannot access parent data using super

In [ ]:

```
# can super access parent ka data?
# using super outside the class
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    def buy(self):
        print ("Buying a smartphone")
        # syntax to call parent ka buy method
        print(super().brand)

s=SmartPhone(20000, "Apple", 13)
```

```
s.buy()
super().buy()
```

In [76]:

```
# super -> constuctor
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

class SmartPhone(Phone):
    def __init__(self, price, brand, camera, os, ram):
        print('Inside smartphone constructor')
        super().__init__(price, brand, camera)
        self.os = os
        self.ram = ram
        print ("Inside smartphone constructor")

s=SmartPhone(20000, "Samsung", 12, "Android", 2)

print(s.os)
print(s.brand)
```

```
Inside smartphone constructor
Inside phone constructor
Inside smartphone constructor
Android
Samsung
```

Inheritance in summary

- A class can inherit from another class.
- Inheritance improves code reuse
- Constructor, attributes, methods get inherited to the child class
- The parent has no access to the child class

- Private properties of parent are not accessible directly in child class
- Child class can override the attributes or methods. This is called method overriding
- super() is an inbuilt function which is used to invoke the parent class methods and constructor

In [87]:

```
class Parent:

    def __init__(self,num):
      self.__num=num

    def get_num(self):
      return self.__num

class Child(Parent):

    def __init__(self,num,val):
      super().__init__(num)
      self.__val=val

    def get_val(self):
      return self.__val

son=Child(100,200)
print(son.get_num())
print(son.get_val())
```

```
100
200
```

In [89]:

```
class Parent:
    def __init__(self):
        self.num=100

class Child(Parent):

    def __init__(self):
        super().__init__()
        self.var=200
    def show(self):
        print(self.num)
        print(self.var)

son=Child()
son.show()
```

```
100
200
```

In [91]:

```
class Parent:
    def __init__(self):
        self.__num=100

    def show(self):
        print("Parent:",self.__num)

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.__var=10

    def show(self):
        print("Child:",self.__var)

obj=Child()
obj.show()
```

```
Child: 10
```

In [93]:

```
class Parent:
    def __init__(self):
        self.__num=100

    def show(self):
        print("Parent:",self.__num)

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.__var=10
    def show(self):
        print("Child:",self.__var)

obj=Child()
obj.show()
```

```
Child: 10
```

Types of Inheritance

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance(Diamond Problem)
- Hybrid Inheritance

Single Inheritance

In [96]:

```
# single inheritance
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass

SmartPhone(1000,"Apple","13px").buy()
```

```
Inside phone constructor
Buying a phone
```

Multilevel Inheritance

In [98]:

```
# multilevel
class Product:
    def review(self):
        print ("Product customer review")

class Phone(Product):
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass

s=SmartPhone(20000, "Apple", 12)

s.buy()
s.review()
```

```
Inside phone constructor
Buying a phone
Product customer review
```

In [100…

```
# Hierarchical
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")
```

```
class SmartPhone(Phone):
    pass

class FeaturePhone(Phone):
    pass

SmartPhone(1000,"Apple","13px").buy()
FeaturePhone(10,"Lava","1px").buy()
```

```
Inside phone constructor
Buying a phone
Inside phone constructor
Buying a phone
```

In [106…

```
# Multiple
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class Product:
    def review(self):
        print ("Customer review")

class SmartPhone(Phone, Product):
    pass

s=SmartPhone(20000, "Apple", 12)

s.buy()
s.review()
```

```
Inside phone constructor
Buying a phone
Customer review
```

In [108…

```
# the diamond problem
# https://stackoverflow.com/questions/56361048/what-is-the-diamond-problem-in-python-and-why-its-not-appear-in-python2
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class Product:
    def buy(self):
        print ("Product buy method")

# Method resolution order
class SmartPhone(Phone,Product):
    pass

s=SmartPhone(20000, "Apple", 12)

s.buy()
```

```
Inside phone constructor
Buying a phone
```

In [110…

```
class A:

    def m1(self):
        return 20

class B(A):

    def m1(self):
        return 30

    def m2(self):
        return 40
```

```
class C(B):

    def m2(self):
        return 20
obj1=A()
obj2=B()
obj3=C()
print(obj1.m1() + obj3.m1()+ obj3.m2())
```

```
70
```

In [ ]:

```
class A:

    def m1(self):
        return 20

class B(A):

    def m1(self):
        val=super().m1()+30
        return val

class C(B):

    def m1(self):
        val=self.m1()+20
        return val
obj=C()
print(obj.m1())
# infine recursion
```

Polymorphism

Having multiple faces

- Method Overriding :
- Method Overloading : Two method have same name but different parameter

- Operator Overloading : same operator works differently on types of input

Method Overloading

In Python, method overloading (with multiple methods having the same name but different parameters) doesn't work the same way as in languages like Java or C++. Python doesn't support defining multiple methods with the same name and different parameter lists directly.

If you define multiple methods with the same name in Python, the latest definition will overwrite the previous ones. Here's an example to show that:

Example: Attempting Method Overloading in Pytho

In [118…

```
class Shape:

  def area(self,a,b=0):
    if b == 0:
      return 3.14*a*a
    else:
      return a*b

s = Shape()

print(s.area(2))
print(s.area(3,4))
```

```
12.56
12
```

Operator Overloading

In [126…

```
'hello' + 'world'
```

Out[126…

```
'helloworld'
```

In [128…

```
4+6
```

Out[128…

```
10
```

In [134…

```
[1,2,3] + [4,5]
```

Out[134…

```
[1, 2, 3, 4, 5]
```

Abstraction

- Abstraction is a key concept in object-oriented programming (OOP) that allows you to hide complex implementation details and show only the essential features of an object. The main goal of abstraction is to simplify the interaction with complex systems by providing a clear interface while hiding unnecessary internal details.

In Python, abstraction can be achieved using:

- Abstract Classes
- Abstract Methods

Python provides the abc (Abstract Base Class) module to define abstract classes and methods. An abstract class is a class that cannot be instantiated directly and must be subclassed by other classes. It can contain abstract methods, which are methods that must be implemented by any subclass.

Abstract Class: A class that contains one or more abstract methods and cannot be instantiated. Abstract Method: A method that is declared but contains no implementation. It must be implemented by any subclass.

In [140…

```
from abc import ABC,abstractmethod
class BankApp(ABC):

  def database(self):
    print('connected to database')

  @abstractmethod
  def security(self):
    pass

  @abstractmethod
  def display(self):
    pass
```

In [142…

```
class MobileApp(BankApp):

  def mobile_login(self):
    print('login into mobile')

  def security(self):
    print('mobile security')

  def display(self):
    print('display')
```

In [144…

```
mob = MobileApp()
```

In [146…

```
mob.security()
```

```
mobile security
```

END