Mayank Singh

3, Dec, 2022

Class Relationships

It is very ovious that Whenever an application is build it uses OOP paradigm and there are multiple classes that are created while develpoing those application. And to function the application/s properly different classes need to communicate with each other. For the coomincation to happen smoothly, classes have relationships. There are three types of relationship among classes:

1. Aggregation
2. Composition
3. Inheritance

Refer - https://www.geeksforgeeks.org/python-oops-aggregation-and-composition/ (https://www.geeksforgeeks.org/python-oops-aggregation-and-composition/) for more details.

Aggregation

Aggregation is a concept in which an object of one class can own or access another independent object of another class.

It represents **Has-A's** relationship. It is a unidirectional association i.e. a one-way relationship. For example, a department can have students but vice versa is not possible and thus unidirectional in nature. In Aggregation, both the entries can survive individually which means ending one entity will not affect the other entity.

Aggregation

In [1]:

```
## Aggregation example with code
class Customer: ## creating class number 1

    def __init__(self,name,gender,address): ## constructor of the class - paramet
        self.name = name
        self.gender = gender
        self.address = address ## address being a complex attribute, that can cor
                               ## therefore we will create altogether a new clas

    def print_address(self):
        print(self.address.city,self.address.pin,self.address.state) ## this func
                                                                      ## contain attributes

class Address:## creating class number 2

    def __init__(self,city,pin,state): ## constructor of the class - parameterize
        self.city = city
        self.pin = pin
        self.state = state

add1 = Address('gurgaon',122011,'haryana') # object of address class
cust = Customer('nitish','male',add1) ## object of customer class, that contains
                                      ##- this is known as aggregation

cust.print_address()
```

```
gurgaon 122011 haryana
```

In [2]:
```
## Aggregation example with code - # in case of private attribute
class Customer: ## creating class number 1

    def __init__(self,name,gender,address): ## constructor of the class - paramet
        self.name = name
        self.gender = gender
        self.address = address
    ## using the get method of Address class, we are accessing the city attribute
    def print_address(self):
        print(self.address.get_city(),self.address.pin,self.address.state)

class Address:## creating class number 2

    def __init__(self,city,pin,state): ## constructor of the class - parameterize
        self.__city = city ## making city a private attribute
        self.pin = pin
        self.state = state
## since private attributes can not be called outside the class directly, we have
    def get_city(self):
        return self.__city

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

cust.print_address()
```

```
gurgaon 122011 haryana
```

In [3]:
```
## Aggregation example with code - using methods smartly
class Customer: ## creating class number 1

    def __init__(self,name,gender,address): ## constructor of the class - paramet
        self.name = name
        self.gender = gender
        self.address = address

    def print_address(self):
        print(self.address._Address__city,self.address.pin,self.address.state)
# while updating the address, we are calling edit address function of Address cla
    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:## creating class number 2

    def __init__(self,city,pin,state): ## constructor of the class - parameterize
        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()
```

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

Composition

Composition is a type of Aggregation in which two entities are extremely reliant on one another. It indicates a relationship component. Both entities are dependent on each other in composition. The composed object cannot exist without the other entity when there is a composition between two entities.

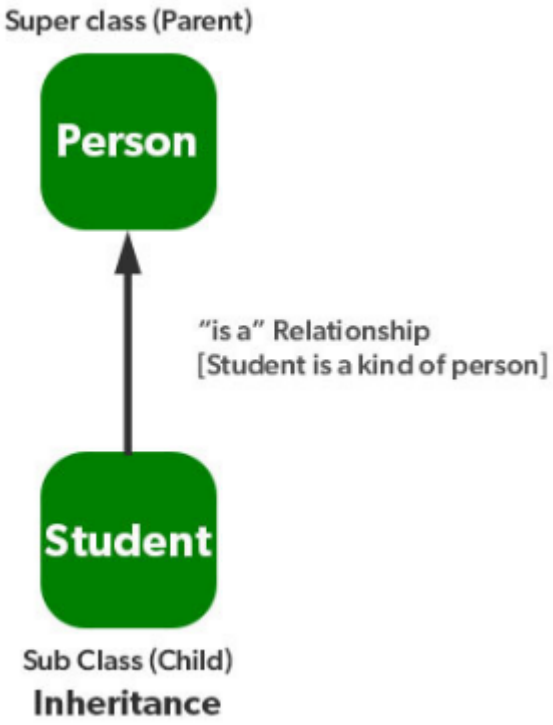
for composition code and explanation refer - [https://www.geeksforgeeks.org/python-oops-aggregation-and-composition/ (https://www.geeksforgeeks.org/python-oops-aggregation-and-composition/)](https://www.geeksforgeeks.org/python-oops-aggregation-and-composition/)

Inheritence

Inheritance allows us to define a class that inherits all the methods and properties from another class. Parent class is the class being inherited from, also called base class. Child class is the class that inherits from another class, also called derived class.

Inheritance is a mechanism that allows us to take all of the properties of another class and apply them to our own. The parent class is the one from which the attributes and functions are derived (also called as Base Class). Child Class refers to a class that uses the properties of another class (also known as a Derived class). An **Is-A Relation** is another name for inheritance.

Through Inheritance relationship, child class can access the methods and atrributes of parent class.



https://www.youtube.com/watch?v=bEWwM4hXZg8&t=4s (https://www.youtube.com/watch?v=bEWwM4hXZg8&t=4s)

What gets inherited?

1. Constructor
2. Non Private Attributes
3. Non Private Methods

In [4]:
```
# 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):
        self.rollno = '20'

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

u = User()
s = Student()
## Using inheritence concept, the object of Student class can now access the attr
print(s.name) ## it will not be printed, beacuse Student class's contructor has r
s.login()
s.enroll()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [4], in <cell line: 24>()
     22 s = Student()
     23 ## Using inheritence concept, the object of Student class can now acces
s the attribute (name) and method (login) of User class.
---> 24 print(s.name) ## it will not be printed, beacuse Student class's contru
ctor has no attribute named as 'name'.
     25 s.login()
     26 s.enroll()

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

Important point to be noted:

In the above code, we can see that the attribute **name** is not getting called and the comiler shows an error. This is because while creating object of child class i.e. **student**, its constructor was called and there is no attribute named as **name**, and hence error appears as **'Student' object has no attribute 'name'**.

The child class will be able to call the parent class attribute only when there is no constructor in the child class, and hence by default parent class's constructor will be called at the very first place and **name** attribute will be called. *See below code*.

At a later stage we will see how to access the parent class constructor, even though child class has its own constructor.

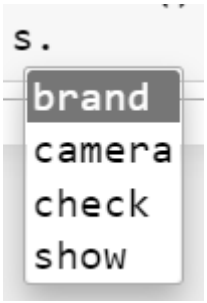
In [5]:
```
# Example

# parent
class User:

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

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

# child
class Student(User):

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

u = User()
s = Student()
## Using inheritence concept, the object of Student class can now access the attr
print(s.name) ## it will printed, beacuse Student class's contructor is NA, hence
s.login()
s.enroll()
```

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

In [6]:
```
#child can't access private members of the class. We cannot access __price attrib
#hence using a getter method to access it in child 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()
```

```
Inside phone constructor
20000
```

price attibute is not available to access as it is private. Simillar is the case with private method.



Method Overriding

If parent calss and child class has same methods (methods with same name), by default child method will be called.

In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class. In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as **Method Overriding**.

In [7]:
```
# Method Overriding
# constructor of parent is called, since child has no constructor and hence outpu
# method buy of child is called and hence output shows - Buying a smartphone
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

If parent and child has same methods, we can call parent method using **super** keyword. **It is always used inside the class**.

In [8]:
```
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
```

In [9]:
```python
# 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
```

In [10]:
```
## using super outside the class will not work
# 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")

s=SmartPhone(20000, "Apple", 13)
super().buy() ## Declaring super outside the class.
```

```
Inside phone constructor

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Input In [10], in <cell line: 18>()
     15         print ("Buying a smartphone")
     17 s=SmartPhone(20000, "Apple", 13)
---> 18 super().buy()

RuntimeError: super(): no arguments
```

In [11]:
```
## 1) super cannot access variables 2) super cannot be used outside the class 3)
```

In [12]:
```python
# super cannot access variables
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")
        print(super().brand) ## it will not work.

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

s.buy()
```

```
Inside phone constructor
Buying a smartphone

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [12], in <cell line: 19>()
     15         print(super().brand) ## it will not work.
     17 s=SmartPhone(20000, "Apple", 13)
---> 19 s.buy()

Input In [12], in SmartPhone.buy(self)
     13 def buy(self):
     14     print ("Buying a smartphone")
---> 15     print(super().brand)

AttributeError: 'super' object has no attribute 'brand'
```

Types of Inheritence

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance(Diamond Problem)
5. Hybrid Inheritance

In [13]:
```python
## One parent class with one child class
# 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
```

In [14]:
```python
# multilevel
# More than one parent - more than one child - like child has father, grand fathe
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 [15]:
```python
# Hierarchical
# one parent two child
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 [16]:
```python
# Multiple
# two parents one child
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 [17]:
```
# the diamond problem
# https://stackoverflow.com/questions/56361048/what-is-the-diamond-problem-in-pyt
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
```

Polymorphism

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

We see Polymorphism in programming through the below mentioned three use cases:

1. Method Overriding
2. Method Overloading
3. Operator Overloading

In [18]:
```
## Method Overriding:
## Explained Above
```

Method Overloading

If one class has methods with same name, but take different inputs. **In Python Method Overloading is not possible**.

In [19]:
```
class Shape:

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

    def area(self,l,b):
        return l*b

s = Shape()

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [19], in <cell line: 11>()
      7         return l*b
      9 s = Shape()
---> 11 print(s.area(2))
     12 print(s.area(3,4))

TypeError: Shape.area() missing 1 required positional argument: 'b'
```

In [20]:
```
# Although Method overloading is directly NA in Python, but we can implement it u
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

Use of same operator differently, based on the input.

In [21]:
```
# operator + can be used on integers, strings, list etc.
'hello' + 'world' # concatenation
```

Out[21]:
```
'helloworld'
```

In [22]:
```
4 + 5 # addition
```

Out[22]: 9

In [23]:
```
[1,2,3] + [4,5] # merging
```

Out[23]: [1, 2, 3, 4, 5]