Date : 8th - 09 - 2020
Morning Session : 9am – 11.00 PM
By ~ Rohan Kumar

Topics: Python OOPs Day-2

Principles of OOP:

1) **Encapsulation**
2) **Inheritance**
3) **Abstraction**
4) **Polymorphism**

Encapsulation: Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.

```
class Mycar:
    def __init__(self,mil,model,speed):
        self.mil = mil
        self.model = model
        self.speed = speed

    def get_speed(self):
        return self.speed

    def get_mil(self):  # get or print data
        return self.speed

    def set_speed(self,value):   #update the data
        self.speed = value

bmw = Mycar(30,'bmw',200)
bmw.set_speed(300)
bmw.speed = 400
bmw.get_speed()
```

```
400
```

Real-life example of encapsulation : in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section". As using encapsulation also hides the data. In this example, the data of any of the sections like sales, finance or accounts are hidden from any other section.

Protected members : are those members of the class which cannot be accessed outside the class but can be accessed from within the class and it's subclasses. To accomplish this in Python, just follow **the convention** by prefixing the name of the member by a **single underscore "_"**.

```
class Test:
    def __init__(self):
        self.a  = 10
        self._b  = 20
        self.__c = 30  #two underscores makes the variable private and we can only access it in side the class,not outside of it

t1 = Test()
print(t1.a)
print(t1._b)
print(t1.__c)
```

```
10
20

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-12-2e41712b7c15> in <module>
      8 print(t1.a)
      9 print(t1._b)
---> 10 print(t1.__c)

AttributeError: 'Test' object has no attribute '__c'
```

Private members : Private members are similar to protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class. In Python, there is no existence of **Private** instance variables that cannot be accessed except inside a class. However, to define a private member prefix the member name with double underscore "__".

```
class Mycar:
    def __init__(self,mil,model,speed):
        self.mil = mil
        self.model = model
        self.__speed = speed

    def get_speed(self):
        return self.__speed

    def set_speed(self,value):    #update the data
        self.__speed = value

bmw = Mycar(30,'bmw',200)
bmw.set_speed(300)
bmw.__speed = 400
bmw.get_speed()
```

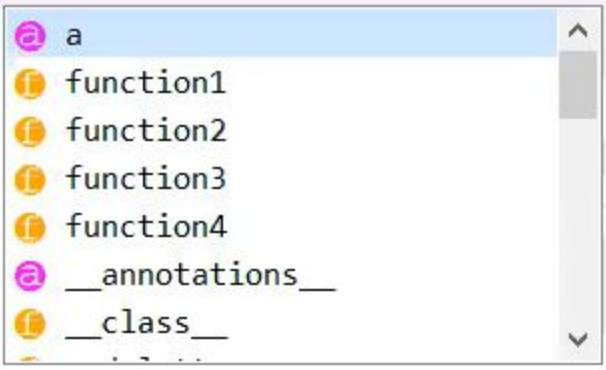
```
300
```

Inheritance:

Inheritance is the capability of one class to derive or inherit the properties from another class. The benefits of inheritance are:

1. It represents real-world relationships well.
2. It provides reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

```python
class A:    #parent
    def function1(self):
        print("this is func1 from A")

    def function2(self):
        print("this is funct2 from A")

class B(A): #child
    def function3(self):
        print("this is func3 from B")

    def function4(self):
        print("this is funct4 from B")



a = A()
b = B()b.
```

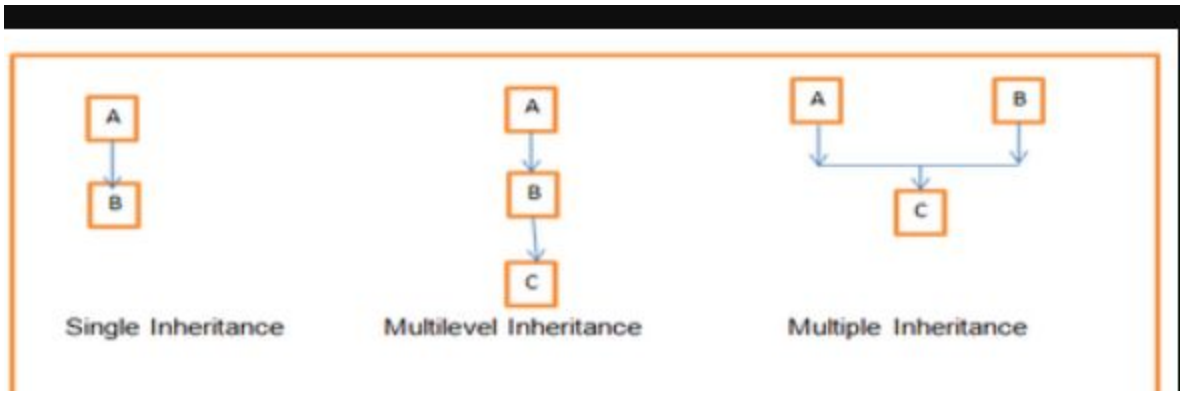


Different forms of Inheritance:

1. Single inheritance: When a child class inherits from only one parent class, it is called single inheritance. We saw an example above.

2. Multiple inheritance: When a child class inherits from multiple parent classes, it is called multiple inheritance. Unlike Java and like C++, Python supports multiple inheritance. We specify all parent classes as a comma-separated list in the bracket.

3. Multilevel inheritance: When we have a child and grandchild relationship.



Constructors in Inheritance:

```
class A:    #parent

    def __init__(self):
        print("i am in A constructor")

    def function1(self):
        print("this is func1 from A")

    def function2(self):
        print("this is funct2 from A")

class B(A): #child

    def function3(self):
        print("this is func3 from B")

    def function4(self):
        print("this is funct4 from B")



b = B()
```

When the child does not have a constructor function then it parents constructor function will execute.

```
class A:   #parent

    def __init__(self):
        print("i am in A constructor")

    def function1(self):
        print("this is func1 from A")

    def function2(self):
        print("this is funct2 from A")

class B(A): #child

    def __init__(self):
        print("i am in B constructor")

    def function3(self):
        print("this is func3 from B")

    def function4(self):
        print("this is funct4 from B")



b = B()
```

When you create an object of a class when the child class does not have a constructor its parents class will be executed and When the child has a constructor it's only child constructor will execute.

Super is a keyword used to excess parents properties.

```
class A:    #parent

    def __init__(self):
        print("i am in A constructor")

    def function1(self):
        print("this is func1 from A")

    def function2(self):
        print("this is funct2 from A")

class B(A): #child

    def __init__(self):
        super().__init__()
        print("i am in B constructor")

    def function3(self):
        print("this is func3 from B")

    def function4(self):
        print("this is funct4 from B")



b = B()
```

MCQ 1 :

MCQ 2:

MCQ 3:

```
class Demo:
    def __init__(self):
        self.model = 'car'
        self._wheels = 4
        self.__color = 'red'
```

Answer:

MCQ 4:

```
class Test1:
    def __init__(self):
        print("in Test1")

class Test2(Test1):
    def func1(self):
        print("in Test2")

t2 = Test2()
```

Answer: