Object Oriented Programming- II

Inheritance

Inheritance is a fundamental feature of object-oriented programming (OOP) that allows you to define a new class based on an existing class. In Python, inheritance enables the creation of a hierarchy of classes, where the child or derived classes inherit attributes and methods from the parent or base class. This promotes code reusability and allows for the implementation of the "is-a" relationship between classes.

Here are some key points about inheritance in Python:

1. **Base class and derived class**

   The class from which inheritance is derived is called the base class, parent class, or superclass.
   The class that inherits from the base class is called the derived class, child class, or subclass.

2. **Syntax**

   To create a derived class, specify the base class in parentheses after the derived class name.
   The derived class inherits all the attributes and methods of the base class.
   Example:

```
class BaseClass:
      # Base class attributes and methods


     class DerivedClass(BaseClass):
```

3. Inheriting attributes and methods

The derived class automatically inherits all attributes and methods from the base class.

The derived class can add new attributes and methods or override the ones inherited from the base class.

4. Accessing base class functionality

The derived class can access the attributes and methods of the base class using the super() function.

The super() function returns a temporary object of the base class, allowing you to call its methods or access its attributes.

Example:

```
class BaseClass:
    def __init__(self):
        self.base_attribute = 10

    def base_method(self):
        return "This is a base method"

class DerivedClass(BaseClass):
    def __init__(self):
            super().__init__()      # Call the base class
constructor
        self.derived_attribute = 20

    def derived_method(self):
        return "This is a derived method"

obj = DerivedClass()
print(obj.base_attribute)      # Output: 10
```

```
print(obj.base_method())           # Output: This is a base
method
print(obj.derived_attribute)    # Output: 20
print(obj.derived_method())      # Output: This is a derived
method
```

__str__ method()

The __str__ method returns the human readable or informal string representation of an object This method is called by an inbuilt print() , str() or format() functions.

If you don't define a __str__() method for a class,then the builtin object implementation calls the __repr__() method instead.

Consider the below example:

```
class Ocean:
    def __init__(self, sea_creature_name, sea_creature_age):
        self.name = sea_creature_name
        self.age = sea_creature_age
    def __str__(self):
            return f'The creature type is {self.name} and the age is
{self.age}'
    def __repr__(self):
        return f'Ocean(\'{self.name}\', {self.age})'
c= Ocean('Jellyfish',5)
print(c)
print(str(c))
print(repr(c))
```

For the above code, **print(c) and print(str(c))** will call the __str__ method while repr(c) will call the __repr__() method

Types of Inheritance

1. **Single Inheritance**

   This is the simplest form of inheritance, where a class inherits from a single base class.

   Example:

```
class Vehicle:
    def move(self):
        print("Moving...")

class Car(Vehicle):
    def start(self):
        print("Car started")

car = Car()
car.move()  # Inherits 'move' method from the Vehicle class
car.start()  # Specific to the Car class
```

2. **Multiple inheritance**

   Python supports multiple inheritance, which means a class can inherit from multiple base classes. In this case, the derived class inherits attributes and methods from all the base classes.

Example:

```
class Flyable:
    def fly(self):
        print("Flying...")

class Swimmable:
    def swim(self):
        print("Swimming...")

class Amphibian(Flyable, Swimmable):
    pass

amphibian = Amphibian()
amphibian.fly()  # Inherits 'fly' method from Flyable class
amphibian.swim()   # Inherits 'swim' method from Swimmable
class
```

3. Multilevel inheritance

In multilevel inheritance, a derived class inherits from a base class, and another class derives from this derived class, forming a hierarchical structure.

Example:

```
class Animal:
    def eat(self):
        print("Eating...")

class Dog(Animal):
    def bark(self):
        print("Barking...")

class Bulldog(Dog):
    def guard(self):
        print("Guarding...")

bulldog = Bulldog()
bulldog.eat()  # Inherits 'eat' method from Animal class
bulldog.bark()  # Inherits 'bark' method from Dog class
bulldog.guard()  # Specific to the Bulldog class
```

4. Hierarchial Inheritance

In hierarchical inheritance, multiple classes inherit from a single base class. This means that a single base class is extended by multiple derived classes.

Example

```
class Shape:
    def draw(self):
        print("Drawing shape...")

class Circle(Shape):
    def draw(self):
        print("Drawing circle...")

class Square(Shape):
    def draw(self):
        print("Drawing square...")

circle = Circle()
circle.draw()  # Overrides the 'draw' method in Shape class

square = Square()
square.draw()  # Overrides the 'draw' method in Shape class
```

5. Hybrid Inheritance

Hybrid inheritance is a combination of multiple inheritance and multilevel inheritance. It involves multiple base classes and forms complex inheritance relationships.

Example:

```
class A:
    def method_a(self):
        print("Method A")

class B(A):
    def method_b(self):
        print("Method B")

class C(A):
    def method_c(self):
        print("Method C")

class D(B, C):
    def method_d(self):
        print("Method D")

d = D()
d.method_a()  # Inherits 'method_a' from A class
d.method_b()  # Inherits 'method_b' from B class
d.method_c()  # Inherits 'method_c' from C class
d.method_d()  # Specific to the D class
```

Method Resolution Order

Method Resolution Order (MRO) is the order in which methods are resolved or searched for in a class hierarchy. It defines how the inheritance hierarchy is traversed to find the appropriate method implementation when a method is called on an object.

To view the method resolution order for a class, you can use the __mro__ attribute or the mro() method. Here's an example:

Code:

```
class A:
    def method(self):
        print("A's method")

class B(A):
    def method(self):
        print("B's method")

class C(A):
    def method(self):
        print("C's method")

class D(B, C):
    pass

d = D()
d.method()
print(D.__mro__)
```

Output:

B's method

(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)

Explanation

In this example, the class D inherits from classes B and C, which in turn both inherit from class A. When d.method() is called, Python uses the MRO to find the appropriate method implementation. The MRO for class D is D, B, C, A, object. Python starts by searching for the method in class D, then B, then C, and finally in class A. Since class B is the first in the MRO that defines the method implementation, its method is called.

Polymorphism

The literal meaning of polymorphism is the condition of occurrence in different forms.

Inbuilt polymorphism in Python:

The same **len()** function can operate on different collections. It can either give you the length of the string or it can give you the length of the list.

Consider the another example mentioned below:

```
def add(x, y, z = 0):
    return x+y+z
print(add(5,3))
print(add(5,4,3))
```

When only two arguments are passed , value of third argument will be 0.'

When all the three arguments will be passed, the value of z will be 3.

This is also an example of polymorphism.

Method Overriding

Method overriding in Python is a feature that allows a subclass to provide a different implementation of a method that is already defined in its superclass. When a method in the subclass has the same name and signature as a method in the superclass, it is said to override the superclass method.

To override a method in Python, follow these steps:

1. Create a superclass (base class) with a method that you want to override.

```
class Animal:
    def make_sound(self):
        print("The animal makes a sound.")
```

2. Create a subclass (derived class) that inherits from the superclass.

```
class Dog(Animal):
    def make_sound(self):
        print("The dog barks.")
```

3. Define a method in the subclass with the same name and signature as the method you want to override in the superclass.

```
class Dog(Animal):
    def make_sound(self):
        print("The dog barks.")
```

4. In the subclass method, provide the implementation specific to the subclass. You can completely override the superclass method or modify its behaviour.

```
class Dog(Animal):
    def make_sound(self):
        print("Woof! Woof!")
```

5. Create objects of the subclass and call the overridden method. The subclass method will be executed instead of the superclass method.

```
animal = Animal()
animal.make_sound()  # Output: The animal makes a sound.

dog = Dog()
dog.make_sound()  # Output: Woof! Woof!
```

In this example, the Dog class overrides the make_sound method inherited from the Animal class. When make_sound is called on a Dog object, it prints "Woof! Woof!" instead of the default message from the Animal class.

Operator Overloading

Operator overloading in Python allows you to define the behaviour of operators such as +, -, *, /, ==, <, >, etc., for custom classes. It allows objects of a class to respond to operators in a way that is intuitive and meaningful based on the context of the class.

To overload an operator in Python, you need to define special methods within your class that correspond to the operator you want to overload. These special methods have predefined names and are called "magic" or "dunder" methods (short for "double underscore" methods).

Here's an example of how to overload the + operator:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        else:
            raise TypeError("Unsupported operand type for +")

# Usage
v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2
print(v3.x, v3.y)  # Output: 6 8
```

In the above example, the Vector class overloads the + operator by defining the __add__ method. This method is called when the + operator is used between two Vector objects. It adds the corresponding components of the two vectors and returns a new Vector object representing the sum.

Abstract class

In Python, an abstract class is a class that cannot be instantiated directly and is meant to be subclassed. It serves as a blueprint for other classes, defining a common interface and possibly some default implementations for its subclasses. Python provides the abc module (Abstract Base Classes) for creating abstract classes.

To create an abstract class in Python, follow these steps:

1. Import the ABC class and the abstract method decorator from the abc module.

```
from abc import ABC, abstractmethod
```

2. Create a class and inherit from ABC to make it an abstract class.

```
class AbstractClass(ABC):
    pass
```

3. Define abstract methods within the abstract class using the @abstractmethod decorator

```
class AbstractClass(ABC):
    @abstractmethod
    def abstract_method(self):
        pass
```

The @abstractmethod decorator marks a method as abstract, indicating that any subclass of the abstract class must implement this method. Abstract methods do not have any implementation in the abstract class itself; they only serve as placeholders for implementation in the derived classes.