Python Objects and Classes

In the last tutorial, we learned about Python OOP. We know that python also supports the concept of objects and classes.
An object is simply a collection of data (variables) and methods (functions). Similarly, a class is a blueprint for that object.

Before we learn about objects, let's first know about classes in Python.

Python Classes

A class is considered as a blueprint of objects. We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

Define Python Class

We use the `class` keyword to create a class in Python. For example,

```
class ClassName:
    # class definition
```

Here, we have created a class named `ClassName`.
Let's see an example,

```
class Bike:
    name = ""
    gear = 0
```

Here,

- `Bike` - the name of the class

- `name/gear` - variables inside the class with default values `""` and **0** respectively.

> **Note**: The variables inside a class are called attributes.

Python Objects

An object is called an instance of a class. For example, suppose `Bike` is a class then we can create objects like `bike1`, `bike2`, etc from the class.

Here's the syntax to create an object.

```
objectName = ClassName()
```

Let's see an example,

```
# create class
class Bike:
    name = ""
    gear = 0

# create objects of class
bike1 = Bike()
```

Here, `bike1` is the object of the class. Now, we can use this object to access the class attributes.

Access Class Attributes Using Objects

We use the `.` notation to access the attributes of a class. For example,

```
# modify the name attribute
```

```python
bike1.name = "Mountain Bike"

# access the gear attribute
bike1.gear
```

Here, we have used `bike1.name` and `bike1.gear` to change and access the value of `name` and `gear` attribute respectively.

Example 1: Python Class and Objects

```python
# define a class
class Bike:
    name = ""
    gear = 0

# create object of class
bike1 = Bike()

# access attributes and assign new values
bike1.gear = 11
bike1.name = "Mountain Bike"

print(f"Name: {bike1.name}, Gears: {bike1.gear} ")
```

Output

```
Name: Mountain Bike, Gears: 11
```

In the above example, we have defined the class named `Bike` with two attributes: `name` and `gear`.

We have also created an object `bike1` of the class `Bike`.

Finally, we have accessed and modified the attributes of an object using the `.` notation.

Create Multiple Objects of Python Class

We can also create multiple objects from a single class. For example,

```
# define a class
class Employee:
    # define an attribute
    employee_id = 0

# create two objects of the Employee class
employee1 = Employee()
employee2 = Employee()

# access attributes using employee1
employee1.employeeID = 1001
print(f"Employee ID: {employee1.employeeID}")

# access attributes using employee2
employee2.employeeID = 1002
print(f"Employee ID: {employee2.employeeID}")
```

Run Code

Output

```
Employee ID: 1001
Employee ID: 1002
```

In the above example, we have created two objects `employee1` and `employee2` of the `Employee` class.

Python Methods

We can also define a function inside a Python class. A Python Function defined inside a class is called a method.

Let's see an example,

```
# create a class
class Room:
    length = 0.0
    breadth = 0.0

    # method to calculate area
```

```
    def calculate_area(self):
        print("Area of Room =", self.length * self.breadth)

# create object of Room class
study_room = Room()

# assign values to all the attributes
study_room.length = 42.5
study_room.breadth = 30.8

# access method inside class
study_room.calculate_area()
```

Run Code

Output

```
Area of Room = 1309.0
```

In the above example, we have created a class named `Room` with:

- **Attributes**: `length` and `breadth`
- **Method**: `calculate_area()`

Here, we have created an object named `study_room` from the `Room` class. We then used the object to assign values to attributes: `length` and `breadth`.

Notice that we have also used the object to call the method inside the class,

```
study_room.calculate_area()
```

Here, we have used the `.` notation to call the method. Finally, the statement inside the method is executed.

Python Constructors

Earlier we assigned a default value to a class attribute,

```
class Bike:
```

```
    name = ""
...
# create object
bike1 = Bike()
```

However, we can also initialize values using the constructors. For example,

```
class Bike:

    # constructor function
    def __init__(self, name = ""):
        self.name = name

bike1 = Bike()
```

Here, `__init__()` is the constructor function that is called whenever a new object of that class is instantiated.
The constructor above initializes the value of the `name` attribute. We have used the `self.name` to refer to the `name` attribute of the `bike1` object.
If we use a constructor to initialize values inside a class, we need to pass the corresponding value during the object creation of the class.

```
bike1 = Bike("Mountain Bike")
```

Here, `"Mountain Bike"` is passed to the `name` parameter of `__init__()`.

Python Inheritance

Like any other OOP languages, Python also supports the concept of class inheritance.

Inheritance allows us to create a new class from an existing class.

The new class that is created is known as **subclass** (child or derived class) and the existing class from which the child class is derived is known as **superclass** (parent or base class).

Python Inheritance Syntax

Here's the syntax of the inheritance in Python,

```
# define a superclass
class super_class:
    # attributes and method definition

# inheritance
class sub_class(super_class):
    # attributes and method of super_class
    # attributes and method of sub_class
```

Here, we are inheriting the `sub_class` class from the `super_class` class.

Example 1: Python Inheritance

```
class Animal:

    # attribute and method of the parent class
    name = ""

    def eat(self):
        print("I can eat")

# inherit from Animal
class Dog(Animal):

    # new method in subclass
    def display(self):
```

```
        # access name attribute of superclass using self
        print("My name is ", self.name)

# create an object of the subclass
labrador = Dog()

# access superclass attribute and method
labrador.name = "Rohu"
labrador.eat()

# call subclass method
labrador.display()
```
Run Code

Output

```
I can eat
My name is  Rohu
```

In the above example, we have derived a subclass `Dog` from a superclass `Animal`. Notice the statements,

```
labrador.name = "Rohu"

labrador.eat()
```

Here, we are using `labrador` (object of `Dog`) to access `name` and `eat()` of the `Animal` class. This is possible because the subclass inherits all attributes and methods of the superclass.

Also, we have accessed the `name` attribute inside the method of the `Dog` class using `self`.

---

is-a relationship

In Python, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an **is-a** relationship between two classes. For example,

1. **Car** is a **Vehicle**

2. **Apple** is a **Fruit**
3. **Cat** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Apple** can inherit from **Fruit**, and so on.

Example 2: Inheritance in Python

Let's take a look at another example of inheritance in Python,

A polygon is a closed figure with **3** or more sides. Say, we have a class called `Polygon` defined as follows,

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in 
range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

This class has data attributes to store the number of sides `n` and magnitude of each side as a list called `sides`.

- The `inputSides()` method takes in the magnitude of each side
- The `dispSides()` method displays these side lengths

A triangle is a polygon with **3** sides. So, we can create a class called `Triangle` which **inherits** from `Polygon`. This makes all the attributes of `Polygon` class available to the Triangle class.

We don't need to define them again **(code reusability)**. `Triangle` can be defined as follows.

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, the `Triangle` class has a new method `findArea()` to find and print the area of the triangle.

Now let's see the complete working code of the example above including creating an object,

```
class Polygon:
    # Initializing the number of sides
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in
range(self.n)]

    # method to display the length of each side of the polygon
    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])

class Triangle(Polygon):
    # Initializing the number of sides of the triangle to 3 by
    # calling the __init__ method of the Polygon class
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides

        # calculate the semi-perimeter
```

```
        s = (a + b + c) / 2

        # Using Heron's formula to calculate the area of the triangle
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)

# Creating an instance of the Triangle class
t = Triangle()

# Prompting the user to enter the sides of the triangle
t.inputSides()

# Displaying the sides of the triangle
t.dispSides()

# Calculating and printing the area of the triangle
t.findArea()
```
Run Code

Output

```
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0
The area of the triangle is 6.00
```

Here, we can see that even though we did not define methods like `inputSides()` or `dispSides()` for class `Triangle` separately, we were able to use them.

If an attribute is not found in the class itself, the search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

Method Overriding in Python Inheritance

In the previous example, we see the object of the subclass can access the method of the superclass.

However, what if the same method is present in both the superclass and subclass?

In this case, the method in the subclass overrides the method in the superclass. This concept is known as method overriding in Python.

Example: Method Overriding

```
class Animal:

    # attributes and method of the parent class
    name = ""

    def eat(self):
        print("I can eat")

# inherit from Animal
class Dog(Animal):

    # override eat() method
    def eat(self):
        print("I like to eat bones")

# create an object of the subclass
labrador = Dog()

# call the eat() method on the labrador object
labrador.eat()
```

Run Code

Output

```
I like to eat bones
```

In the above example, the same method `eat()` is present in both the `Dog` class and the `Animal` class.

Now, when we call the `eat()` method using the object of the `Dog` subclass, the method of the `Dog` class is called.
This is because the `eat()` method of the `Dog` subclass overrides the same method of the `Animal` superclass.

The super() Method in Python Inheritance

Previously we saw that the same method in the subclass overrides the method in the superclass.

However, if we need to access the superclass method from the subclass, we use the `super()` method. For example,

```
class Animal:

    name = ""

    def eat(self):
        print("I can eat")

# inherit from Animal
class Dog(Animal):

    # override eat() method
    def eat(self):

        # call the eat() method of the superclass using super()
        super().eat()

        print("I like to eat bones")

# create an object of the subclass
labrador = Dog()

labrador.eat()
```
Run Code

Output

```
I can eat
I like to eat bones
```

In the above example, the `eat()` method of the `Dog` subclass overrides the same method of the `Animal` superclass.

Inside the `Dog` class, we have used

```
# call method of superclass
super().eat()
```

to call the `eat()` method of the `Animal` superclass from the `Dog` subclass.

So, when we call the `eat()` method using the `labrador` object

```
# call the eat() method
labrador.eat()
```

Both the overridden and the superclass version of the `eat()` method is executed.

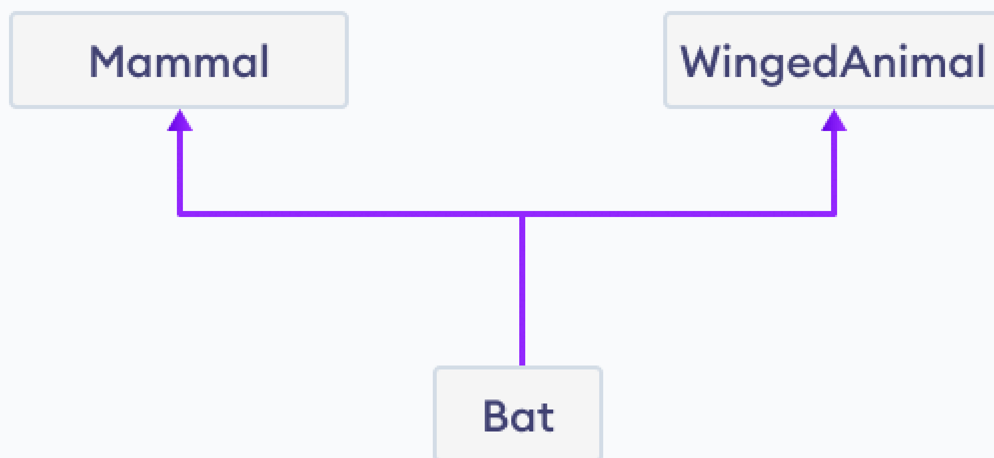
Uses of Inheritance

1. Since a child class can inherit all the functionalities of the parent's class, this allows code reusability.
2. Once a functionality is developed, you can simply inherit it. No need to reinvent the wheel. This allows for cleaner code and easier to maintain.
3. Since you can also add your own functionalities in the child class, you can inherit only the useful functionalities and define other required features.

Python Multiple Inheritance

A class can be derived from more than one superclass in Python. This is called multiple inheritance.

For example, A class `Bat` is derived from superclasses `Mammal` and `WingedAnimal`. It makes sense because bat is a mammal as well as a winged animal.



Multiple Inheritance

Python Multiple Inheritance Syntax

```
class SuperClass1:
    # features of SuperClass1

class SuperClass2:
    # features of SuperClass2

class MultiDerived(SuperClass1, SuperClass2):
    # features of SuperClass1 + SuperClass2 + MultiDerived class
```

Here, the `MultiDerived` class is derived from `SuperClass1` and `SuperClass2` classes.

Example: Python Multiple Inheritance

```
class Mammal:
    def mammal_info(self):
        print("Mammals can give direct birth.")

class WingedAnimal:
    def winged_animal_info(self):
        print("Winged animals can flap.")

class Bat(Mammal, WingedAnimal):
    pass

# create an object of Bat class
b1 = Bat()

b1.mammal_info()
b1.winged_animal_info()
```
Run Code

Output

```
Mammals can give direct birth.
Winged animals can flap.
```

In the above example, the `Bat` class is derived from two super classes: `Mammal` and `WingedAnimal`. Notice the statements,

```
b1 = Bat()
b1.mammal_info()
b1.winged_animal_info()
```

Here, we are using `b1` (object of `Bat`) to access `mammal_info()` and `winged_animal_info()` methods of the `Mammal` and the `WingedAnimal` class respectively.

Python Multilevel Inheritance

In Python, not only can we derive a class from the superclass but you can also derive a class from the derived class. This form of inheritance is known as **multilevel inheritance**.

Here's the syntax of the multilevel inheritance,

```
class SuperClass:
    # Super class code here

class DerivedClass1(SuperClass):
    # Derived class 1 code here

class DerivedClass2(DerivedClass1):
    # Derived class 2 code here
```

Here, the `DerivedClass1` class is derived from the `SuperClass` class, and the `DerivedClass2` class is derived from the `DerivedClass1` class.

Multilevel Inheritance in Python

Example: Python Multilevel Inheritance

```
class SuperClass:

    def super_method(self):
        print("Super Class method called")

# define class that derive from SuperClass
class DerivedClass1(SuperClass):
    def derived1_method(self):
```

```
        print("Derived class 1 method called")

# define class that derive from DerivedClass1
class DerivedClass2(DerivedClass1):

    def derived2_method(self):
        print("Derived class 2 method called")

# create an object of DerivedClass2
d2 = DerivedClass2()

d2.super_method()  # Output: "Super Class method called"

d2.derived1_method()  # Output: "Derived class 1 method called"

d2.derived2_method()  # Output: "Derived class 2 method called"
```

Run Code

Output

```
Super Class method called
Derived class 1 method called
Derived class 2 method called
```

In the above example, `DerivedClass2` is derived from `DerivedClass1`, which is derived from `SuperClass`.

It means that `DerivedClass2` inherits all the attributes and methods of both `DerivedClass1` and `SuperClass`.

Hence, we are using `d2` (object of `DerivedClass2`) to call methods from `SuperClass`, `DerivedClass1`, and `DerivedClass2`.

Python Operator Overloading

In Python, we can change the way operators work for user-defined types.

For example, the `+` operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called **operator overloading**.

Python Special Functions

Class functions that begin with double underscore `__` are called special functions in Python.
The special functions are defined by the Python interpreter and used to implement certain features or behaviors.

They are called **"double underscore"** functions because they have a double underscore prefix and suffix, such as `__init__()` or `__add__()`.
Here are some of the special functions available in Python,

Function	Description
`__init__()`	initialize the attributes of the object
`__str__()`	returns a string representation of the object
`__len__()`	returns the length of the object
`__add__()`	adds two objects
`__call__()`	call objects of the class like a normal function

Example: + Operator Overloading in Python

To overload the `+` operator, we will need to implement `__add__()` function in the class.

With great power comes great responsibility. We can do whatever we like inside this function. But it is more sensible to return the `Point` object of the coordinate sum.

Let's see an example,

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)


p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)

# Output: (3,5)
```

Run Code

In the above example, what actually happens is that, when we use `p1 + p2`, Python calls `p1.__add__(p2)` which in turn is `Point.__add__(p1,p2)`. After this, the addition operation is carried out the way we specified.

Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	Expression	Internally
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`

Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder (modulo)	`p1 % p2`	`p1.__mod__(p2)`
Bitwise Left Shift	`p1 << p2`	`p1.__lshift__(p2)`
Bitwise Right Shift	`p1 >> p2`	`p1.__rshift__(p2)`
Bitwise AND	`p1 & p2`	`p1.__and__(p2)`
Bitwise OR	`p1 | p2`	`p1.__or__(p2)`
Bitwise XOR	`p1 ^ p2`	`p1.__xor__(p2)`
Bitwise NOT	`~p1`	`p1.__invert__()`

Overloading Comparison Operators

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well.

Here's an example of how we can overload the `<` operator to compare two objects the `Person` class based on their `age`:

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # overload < operator
    def __lt__(self, other):
        return self.age < other.age

p1 = Person("Alice", 20)
p2 = Person("Bob", 30)

print(p1 < p2)  # prints True
print(p2 < p1)  # prints False
```
Run Code

Output

```
True
False
```

Here, `__lt__()` overloads the `<` operator to compare the `age` attribute of two objects.

The `__lt__()` method returns,

- `True` - if the first object's `age` is less than the second object's `age`
- `False` - if the first object's `age` is greater than the second object's `age`

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Operator	Expression	Internally
Less than	`p1 < p2`	`p1.__lt__(p2)`
Less than or equal to	`p1 <= p2`	`p1.__le__(p2)`
Equal to	`p1 == p2`	`p1.__eq__(p2)`

Not equal to	`p1 != p2`	`p1.__ne__(p2)`
Greater than	`p1 > p2`	`p1.__gt__(p2)`
Greater than or equal to	`p1 >= p2`	`p1.__ge__(p2)`

Advantages of Operator Overloading

Here are some advantages of operator overloading,

- Improves code readability by allowing the use of familiar operators.
- Ensures that objects of a class behave consistently with built-in types and other user-defined types.
- Makes it simpler to write code, especially for complex data types.
- Allows for code reuse by implementing one operator method and using it for other operators.