Python Tutorial: Object-Oriented Programming system (OOPs) - Part 2

BEGINNER PROGRAMMING PYTHON

This article was published as a part of the Data Science Blogathon

Overview:

In my previous article on OOP, we discussed basic concepts like classes, objects, methods, etc. In this article, we are going to see the main concepts of OOP. They are:

1. Inheritance
   - Single-level Inheritance
   - Multi-level Inheritance
   - Multiple Inheritance
2. Polymorphism
   - Method Overriding
   - Operator Overloading
3. Encapsulation
4. Data Abstraction

I recommend you to read my article on part 1 of OOP that covered the basics of OOP.

Inheritance

Inheritance in programming is similar to biological inheritance. A child gets/inherits his parent's traits, but parents do not get child traits. Similarly, a class that inherits a class is called Sub/Child class. A class from which another class inherits is called the Super/Parent class.

Inheritance is of three types:

1. Single-level Inheritance
2. Multi-level Inheritance
3. Multiple Inheritance

Single-level Inheritance:

In Single-level Inheritance, the subclass inherits the properties and behaviour from a single parent class.

Syntax: If class2 is inheriting class1 -> **class class2(class1):**

Let's say we have 2 classes ClassA, ClassB. ClassA has __init__, classA_method() and classB has classB_method(). And ClassB inherits ClassA.

Below is how the code should look like.

```
class ClassA: def __init__(self): print("In class A init") def classA_method(self): print("Class A method")
class ClassB(ClassA): def classB_method(self): print("Class B method")
```

Now, if we create an object of ClassA, it can only access the methods of ClassA. And if we create an object of ClassB, it can access all the methods of both the classes.

Creating objects and calling methods:

```
objA = ClassA() objA.classA_method() objA.classB_method() objB = ClassB() objB.classA_method()
objB.classB_method()
```

Think of a min and guess the outputs for each of the above statements. I believe you give it a good go. Look at the below figure and check your answers.

```
objA = ClassA()
```

```
In class A init
```

```
objA.classA_method()
```

```
Class A method
```

```
objA.classB_method()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-48-5b9b942b7963> in <module>
----> 1 objA.classB_method()

AttributeError: 'ClassA' object has no attribute 'classB_method'
```

```
objB = ClassB()
```

```
In class A init
```

```
objB.classA_method()
```

```
Class A method
```

```
objB.classB_method()
```

```
Class B method
```

In the above example, the __init__ method is present only in the parent class, guess what happens if the child class has its own __init__ method.

```
class ClassA:


def __init__(self):print("In class A init")


def classA_method(self):print("Class A method")


class ClassB(ClassA):
```

```
def __init__(self):print("In class B init")


def classB_method(self):print("Class B method")
```

create an object of ClassB to see how it works.

```
objB = ClassB()
```

It first looks if ClassB has __init__, if not it calls the __init__ of the parent class.

How do we access the __init__ of the parent class in such a case?

> **super().__init__() is used to call the parent class' __init__ method.**

Let's see an example of super().__init__.

```
class ClassA: def __init__(self): print("In class A init") def classA_method(self): print("Class A method") class ClassB(ClassA): def __init__(self): print("In class B init") super().__init__() def classB_method(self): print("Class B method") Creating an object of ClassB objB = ClassB()
```

Multi-level Inheritance:

In Multi-level Inheritance, a sub-class inherits properties and behavior from another sub-class.

```
class ClassA:


def __init__(self):


print("In class A init")


def classA_method(self):


print("Class A method")


class ClassB(ClassA):
```

```
def __init__(self):

print("In Class B init")

super().__init__()

def classB_method(self):

print("Class B method")

class ClassC(ClassB):

def __init__(self):

print("In Class C init")

super().__init__()

def classC_method(self):

print("Class C method")
```

Here is a small challenge for you. Copy the above code and try creating objects for each class and access the methods to get a better understanding of how multi-level inheritance works.

I hope you've spent an ample amount of time understanding the above code.

Multiple Inheritance:

In multiple Inheritance, a sub-class will have more than one parent class.

Consider we have three classes ClassA, ClassB, and ClassC that inherits both ClassA, ClassB.

```
class ClassA:

def __init__(self):

print("In class A init")

def classA_method(self):

print("Class A method")
```

```
class ClassB:


def __init__(self):


print("In Class B init")


def classB_method(self):


print("Class B method")


class ClassC(ClassB, ClassA):


def __init__(self):


print("In Class C init")


super().__init__()


def classC_method(self):


print("Class C method")
```

Create an object of ClassC:

```
objectC = ClassC()
```

Observe one thing carefully here, when an object is created from ClassC that inherits from ClassA, ClassB. First, it searched for __init__ within ClassC, then it searched as per the order specified during the inheritance. This principle is called “Method Resolution Order (MRO).”

The Method Resolution of a Class can be checked using class.__mro__ or class.mro().

Checking the “MRO” for ClassC:

```
ClassC.mro()
```

First ClassC is searched, then ClassB, and at last ClassA.

Before reading further, have a play around in accessing the methods using the objectC.

Types of Inheritance (Image by Author)

Polymorphism

The word Polymorphism means having many forms. In OOP, the same function can be used for different data types.

The ways of implementing Polymorphism are:

- Method Overriding
- Operator Overloading

Method Overriding

Method overriding is the ability of OOP that allows a child/subclass to have its own set of rules to a method that is already present in the parent/superclass.

```
class A(): def detail(self): print("I am a class A method") class B(A): def detail(self): print("I overrode the detail method of Class A method")
```

The “detail” method of ClassB overrode the “detail” method ClassA.

Operator Overloading

It is the ability of a single operator to work with different types of input data. For example, the “+” operator can add numbers and can concatenate strings.

Assume we need to add two objects. Create a class named adder.

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

Create two objects of the **adder class** and try to add them using the + operator.

```
a = adder(2) b = adder(3)
```

It throws an error that “+” can’t be performed on two objects of the adder class.

“+” operator automatically invokes the built-in method __add__.

So, to add two objects, we need to modify the built-in __add__ method.

```
class Addition: def __init__(self, x): self.x = x def __add__(self, other): result = self.x + other.x
print("Result: ", result)
```

Now, if we create objects of the Addition class and apply + operation, it works.

```
c = Addition(2) d = Addition(3) c + d
```

In the statement “c+d”, + invokes the __add__ method, and “self” points to c and the “other” object is d.

Encapsulation

In OOP, encapsulation is the idea of putting restrictions on accessing variables and methods by wrapping them as a single entity. Thus, an object’s variable can only be changed by its method. Such variables are known as private variables and are prefixed with an underscore ‘_’

Accessing a private variable outside the class throws an error.

```
class A: def __init__(self): self._num = 10 a = A() a.num
```

Data Abstraction

Data Abstraction is the idea of hiding the functionality of a method or class from the users.

Abstract Method: A method that has only a pass statement. And the decorator @abstractmethod is used to declare abstract methods.

Abstract Class: In Python, A class that has at least one abstract method. To create abstract classes we need to use the **ABC** (Abstract Base Class) module.

We cannot create objects of an abstract class with abstract methods.

We can implement the abstract method in subclasses.

```
from abc import ABC, abstractmethod


class A(ABC):


@abstractmethod


def show(self):


print("I am an abstractmethod")


class B(A):
```

```
def show(self):


print("Implementing abstract method")
```

Creating an object of class A

objA = A()

Creating an object of Class B and calling show() method.

```
objB = B() objB.show()
```

End Notes:

I am glad you are determined to read till the conclusion. By the end of this article, we are familiar with all the important concepts of OOP in Python.

I hope this article is informative. Feel free to share it with your study buddies.

References:

Fork the complete code file from the GitHub repo

Read Part 1 of this article from here.

Other Blog Posts by me

Feel free to check out my other blog posts from my Analytics Vidhya Profile.

You can find me on LinkedIn, Twitter in case you would want to connect. I would be glad to connect with you.

For immediate exchange of thoughts, please write to me at harikabonthu96@gmail.com.

Article Url - https://www.analyticsvidhya.com/blog/2021/08/python-tutorial-object-oriented-programming-system-oops-part-2/

Harika Bonthu