FEBRUARY 2, 2022 / #PYTHON

Object-Oriented Programming in Python

Ashutosh Krishna

Python is a fantastic programming language that allows you to use both functional and object-oriented programming paradigms.

Python programmers should be able to use fundamental object-oriented programming concepts, whether they are software developers, machine learning engineers, or something else.

All four core aspects of a generic OOP framework are supported by Python's object-oriented programming system: encapsulation, abstraction, inheritance, and polymorphism.

In this tutorial, we'll take a quick look at these features and get some practice with them.

Object-Oriented Programming Concepts in Python

What are Classes and Objects?

Python, like every other object-oriented language, allows you to define classes to create objects. In-built Python classes are the most common data types in Python, such as strings, lists, dictionaries, and so on.

A class is a collection of instance variables and related methods that define a particular object type. You can think of a class as an object's blueprint or template. Attributes are the names given to the variables that make up a class.

A class instance with a defined set of properties is called an object. As a result, the same class can be used to construct as many objects as needed.

Let's define a class named *Book* for a bookseller's sales software.

```
class Book:
    def __init__(self, title, quantity, author, price):
        self.title = title
        self.quantity = quantity
        self.author = author
        self.price = price
```

The `__init__` special method, also known as a **Constructor**, is used to initialize the Book class with attributes such as title, quantity, author, and price.

In Python, built-in classes are named in lower case, but user-defined classes are named in Camel or Snake case, with the first letter capitalized.

This class can be instantiated to any number of objects. Three books are instantiated in the following example code:

```
book1 = Book('Book 1', 12, 'Author 1', 120)
book2 = Book('Book 2', 18, 'Author 2', 220)
book3 = Book('Book 3', 28, 'Author 3', 320)
```

book1, book2 and book3 are distinct objects of the class *Book*. The term **self** in the attributes refers to the corresponding instances (objects).

```
print(book1)
print(book2)
print(book3)
```

Output:

```
<__main__.Book object at 0x00000156EE59A9D0>
<__main__.Book object at 0x00000156EE59A8B0>
<__main__.Book object at 0x00000156EE59ADF0>
```

The class and memory location of the objects are printed when they are printed. We can't expect them to provide specific information on the qualities, such as the title, author name, and so on. But we can use a specific method called `__repr__` to do this.

In Python, a special method is a defined function that starts and ends with two underscores and is invoked automatically when certain conditions are met.

```
class Book:
    def __init__(self, title, quantity, author, price):
        self.title = title
        self.quantity = quantity
        self.author = author
        self.price = price

    def __repr__(self):
        return f"Book: {self.title}, Quantity: {self.quantity}, Author: {self.autho

book1 = Book('Book 1', 12, 'Author 1', 120)
book2 = Book('Book 2', 18, 'Author 2', 220)
book3 = Book('Book 3', 28, 'Author 3', 320)

print(book1)
print(book2)
print(book3)
```

Output:

```
Book: Book 1, Quantity: 12, Author: Author 1, Price: 120
Book: Book 2, Quantity: 18, Author: Author 2, Price: 220
Book: Book 3, Quantity: 28, Author: Author 3, Price: 320
```

What is Encapsulation?

Encapsulation is the process of preventing clients from accessing certain properties, which can only be accessed through specific methods.

Private attributes are inaccessible attributes, and information hiding is the process of making particular attributes private. You use two underscores to declare private characteristics.

Let's introduce a private attribute called `__discount` in the *Book* class.

```
class Book:
    def __init__(self, title, quantity, author, price):
        self.title = title
        self.quantity = quantity
        self.author = author
        self.price = price
        self.__discount = 0.10

    def __repr__(self):
        return f"Book: {self.title}, Quantity: {self.quantity}, Author: {self.autho

book1 = Book('Book 1', 12, 'Author 1', 120)

print(book1.title)
print(book1.quantity)
print(book1.author)
print(book1.price)
print(book1.__discount)
```

Output:

```
Book 1
12
Author 1
120
Traceback (most recent call last):
  File "C:\Users\ashut\Desktop\Test\hello\test.py", line 19, in <module>
    print(book1.__discount)
AttributeError: 'Book' object has no attribute '__discount'
```

We can see that all the attributes are printed except the private attribute `__discount`. You use getter and setter methods to access private attributes.

We make the price property private in the following code example, and we use a setter method to assign the discount attribute and a getter function to get the price attribute.

```
quantity, author, price):

ty


count):
ount


e * (1-self.__discount)


itle}, Quantity: {self.quantity}, Author: {self.author}, Price: {self.get_price()}"
```

This time we'll create two objects, one for the purchase of single book and another for the purchase of books in bulk quantity. While purchasing books in bulk quantity, we get a discount of 20%, so we'll use the `set_discount()` method to set the discount to 20% in that case.

```
single_book = Book('Two States', 1, 'Chetan Bhagat', 200)

bulk_books = Book('Two States', 25, 'Chetan Bhagat', 200)
bulk_books.set_discount(0.20)

print(single_book.get_price())
print(bulk_books.get_price())
print(single_book)
print(bulk_books)
```

Output:

```
200
160.0
Book: Two States, Quantity: 1, Author: Chetan Bhagat, Price: 200
Book: Two States, Quantity: 25, Author: Chetan Bhagat, Price: 160.0
```

What is Inheritance?

Inheritance is regarded as the most significant characteristics of OOP. A class's ability to inherit methods and/or characteristics from another class is known as inheritance.

The subclass or child class is the class that inherits. The superclass or parent class is the class from which methods and/or attributes are inherited.

Two new classes have been added to our bookseller's sales software: a

Novel class and *Academic* class.

We can see that regardless of whether a book is classified as novel or academic, it may have some similar attributes like title and author, as well as common methods like `get_price()` and `set_discount()`. Rewriting all that code for each new class is a waste of time, effort, and memory.

```
class Book:
    def __init__(self, title, quantity, author, price):
        self.title = title
        self.quantity = quantity
        self.author = author
        self.__price = price
        self.__discount = None

    def set_discount(self, discount):
        self.__discount = discount

    def get_price(self):
        if self.__discount:
            return self.__price * (1-self.__discount)
        return self.__price

    def __repr__(self):
        return f"Book: {self.title}, Quantity: {self.quantity}, Author: {self.autho


class Novel(Book):
    def __init__(self, title, quantity, author, price, pages):
        super().__init__(title, quantity, author, price)
        self.pages = pages


class Academic(Book):
    def __init__(self, title, quantity, author, price, branch):
        super().__init__(title, quantity, author, price)
        self.branch = branch
```

Let's create objects for these classes to visualize them.

```
novel1 = Novel('Two States', 20, 'Chetan Bhagat', 200, 187)
novel1.set_discount(0.20)

academic1 = Academic('Python Foundations', 12, 'PSF', 655, 'IT')

print(novel1)
print(academic1)
```

Output:

```
Book: Two States, Quantity: 20, Author: Chetan Bhagat, Price: 160.0
Book: Python Foundations, Quantity: 12, Author: PSF, Price: 655
```

What is Polymorphism?

The term '**polymorphism**' comes from the Greek language and means '*something that takes on multiple forms.*'

Polymorphism refers to a subclass's ability to adapt a method that already exists in its superclass to meet its needs. To put it another way, a subclass can use a method from its superclass as is or modify it as needed.

```
class Academic(Book):
```

```
class Academic(Book):
    def __init__(self, title, quantity, author, price, branch):
        super().__init__(title, quantity, author, price)
        self.branch = branch

    def __repr__(self):
        return f"Book: {self.title}, Branch: {self.branch}, Quantity: {self.quantity
```

The *Book* superclass has a specific method called `__repr__`. This method can be used by subclass Novel so that it is called whenever an object is printed.

The *Academic* subclass, on the other hand, is defined with its own `__repr__` special function in the example code above. The *Academic* subclass will invoke its own method by suppressing the same method present in its superclass, thanks to polymorphism.

```
novel1 = Novel('Two States', 20, 'Chetan Bhagat', 200, 187)
novel1.set_discount(0.20)

academic1 = Academic('Python Foundations', 12, 'PSF', 655, 'IT')

print(novel1)
print(academic1)
```

Output:

```
Book: Two States, Quantity: 20, Author: Chetan Bhagat, Price: 160.0
Book: Python Foundations, Branch: IT, Quantity: 12, Author: PSF, Price: 655
```

What is Abstraction?

Abstraction isn't supported directly in Python. Calling a magic method, on the other hand, allows for abstraction.

If an abstract method is declared in a superclass, subclasses that inherit from the superclass must have their own implementation of the method.

A superclass's abstract method will never be called by its subclasses. But the abstraction aids in the maintenance of a similar structure across all subclasses.

In our parent class *Book*, we have defined the `__repr__` method. Let's make that method abstract, forcing every subclass to compulsorily have its own `__repr__` method.

```
from abc import ABC, abstractmethod


class Book(ABC):
    def __init__(self, title, quantity, author, price):
        self.title = title
        self.quantity = quantity
        self.author = author
        self.__price = price
        self.__discount = None

    def set_discount(self, discount):
        self.__discount = discount

    def get_price(self):
```

```
        if self.__discount:
            return self.__price * (1-self.__discount)
        return self.__price

    @abstractmethod
    def __repr__(self):
        return f"Book: {self.title}, Quantity: {self.quantity}, Author: {self.autho

class Novel(Book):
    def __init__(self, title, quantity, author, price, pages):
        super().__init__(title, quantity, author, price)
        self.pages = pages


class Academic(Book):
    def __init__(self, title, quantity, author, price, branch):
        super().__init__(title, quantity, author, price)
        self.branch = branch

    def __repr__(self):
        return f"Book: {self.title}, Branch: {self.branch}, Quantity: {self.quantity

novel1 = Novel('Two States', 20, 'Chetan Bhagat', 200, 187)
novel1.set_discount(0.20)

academic1 = Academic('Python Foundations', 12, 'PSF', 655, 'IT')

print(novel1)
print(academic1)
```

In the above code, we have inherited the *ABC* class to create the *Book* class. We've made the `__repr__` method abstract by adding the `@abstractmethod` decorator.

While creating the Novel class, we intentionally missed the implementation of the `__repr__` method to see what happens.

Output:

```
Traceback (most recent call last):
  File "C:\Users\ashut\Desktop\Test\hello\test.py", line 40, in <module>
    novel1 = Novel('Two States', 20, 'Chetan Bhagat', 200, 187)
TypeError: Can't instantiate abstract class Novel with abstract method __repr__
```

We get a **TypeError** saying we cannot instantiate object of the Novel class. Let's add the implementation of the `__repr__` method and see what happens now.

```
 quantity, author, price, pages):
, quantity, author, price)


itle}, Quantity: {self.quantity}, Author: {self.author}, Price: {self.get_price()}"
```

Output:

```
Book: Two States, Quantity: 20, Author: Chetan Bhagat, Price: 160.0
Book: Python Foundations, Branch: IT, Quantity: 12, Author: PSF, Price: 655
```

Now it works fine.

Method Overloading

The concept of method overloading is found in almost every well-known programming language that follows object-oriented programming concepts. It simply refers to the use of many methods with the same name that take various numbers of arguments within a single class.

Let's now understand method overloading with the help of the following code:

```
class OverloadingDemo:
    def add(self, x, y):
        print(x+y)

    def add(self, x, y, z):
        print(x+y+z)


obj = OverloadingDemo()
obj.add(2, 3)
```

Output:

```
Traceback (most recent call last):
  File "C:\Users\ashut\Desktop\Test\hello\setup.py", line 10, in <module>
    obj.add(2, 3)
TypeError: add() missing 1 required positional argument: 'z'
```

You're probably wondering why this happened. As a result, the error is displayed because Python only remembers the most recent definition of add(self, x, y, z), which takes three parameters in addition to self. As a result, three arguments must be passed to the `add()` method when it is called. To put it another way, it disregards the prior definition of add().

Thus, Python **doesn't support** Method Overloading by default.

Method Overriding

When a method with the same name and arguments is used in both a derived class and a base or super class, we say that the derived class method "overrides" the method provided in the base class.

When the overridden method gets called, the derived class's method is always invoked. The method that was used in the base class is now hidden.

```
class ParentClass:
    def call_me(self):
        print("I am parent class")


class ChildClass(ParentClass):
    def call_me(self):
        print("I am child class")

pobj = ParentClass()
pobj.call_me()

cobj = ChildClass()
cobj.call_me()
```

Output:

```
I am parent class
I am child class
```

In the above code, when the `call_me()` method was called on the child object, the `call_me()` from the child class was invoked. We can invoke the parent class's `call_me()` method from the child class using `super()`, like this:

```
class ParentClass:
    def call_me(self):
        print("I am parent class")


class ChildClass(ParentClass):
    def call_me(self):
        print("I am child class")
        super().call_me()

pobj = ParentClass()
pobj.call_me()

cobj = ChildClass()
cobj.call_me()
```

Output:

```
I am parent class
I am child class
I am parent class
```

Wrapping Up

In this article, we covered what classes and objects mean. We also covered the four pillars of the Object-Oriented Programming.

Apart from that we also touched two important topics – Method Overloading and Method Overriding.

Thanks for reading!

Ashutosh Krishna

Application Developer at Thoughtworks India

Trending Guides

Exponents in Python
sizeof Operator in C
Dijkstra's Algorithm
enumerate() in Python
What is an Algorithm?
Parse a Boolean in JS
Python Datetime.now()
Int to String in Java
Python Length of List
Escape a String in JS
What is SQL?
HTML New Line
Python Slicing
JavaScript Range
JS Check for Null
Java List Example
Merge Arrays in JS
JS Modulo Operator
What is a REST API?
JavaScript Encoding
JavaScript Get Request
JS .reverse() Function
Visual Studio vs VSCode
Python Lambda Functions
Python Remove from String
Python String to Datetime
What is Prim's Algorithm?
Different Casings Explained
Principle of Least Privilege
Data Structures & Algorithms

About Alumni Network Open Source Shop Support Sponsors Academic Honesty Code of Conduct Privacy Policy Terms of Service Copyright Policy