Python Object Oriented Programming

In this article, you'll learn about the Object Oriented Programming (OOP) in Python and their fundamental concept with examples.

Table of Contents

Introduction to OOPs in Python

Python is a multi-paradigm programming language. Meaning, it supports different programming approach.

One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:

Parrot is an object,

- name, age, color are attributes
- singing, dancing are behavior

In Python, the concept of OOP follows some basic principles:

Inheritance	A process of using details from a new class without modifying existing class.
Encapsulation	Hiding the private details of a class from other objects.
Polymorphism	A concept of using common operation in different ways for different data input.

Class

A class is a blueprint for the object.

We can think of class as an sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, parrot is an object.

The example for class of parrot can be :

```
class Parrot:
    pass
```

Here, we use `class` keyword to define an empty class `Parrot` . From class, we construct instances. An instance is a specific object created from a particular class.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, `obj` is object of class `Parrot` .

Example 1: Creating Class and Object in Python

```
script.py      IPython Shell
 1    class Parrot:
 2
 3        # class attribute
 4        species = "bird"
 5
 6        # instance attribute
 7        def __init__(self, name, age):
 8            self.name = name
 9            self.age = age
10
11    # instantiate the Parrot class
12    blu = Parrot("Blu", 10)
13    woo = Parrot("Woo", 15)
14
15    # access the class attributes
16    print("Blu is a {}".format(blu.__class__.species))
17    print("Woo is also a {}".format(woo.__class__.species))
18
19    # access the instance attributes
20    print("{} is {} years old".format( blu.name, blu.age))
21    print("{} is {} years old".format( woo.name, woo.age))
```

Run ●

Powered by DataCamp

When we run the program, the output will be:

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

In the above program, we create a class with name Parrot . Then, we define attributes. The attributes are a characteristic of an object.

Then, we create instances of the Parrot class. Here, blu and woo are references (value) to our new objects.

Then, we access the class attribute using __class __.species . Class attributes are same for all instances of a class. Similarly, we access the instance attributes using blu.name and blu.age . However, instance attributes are different for every instance of a class.

To learn more about classes and objects, go to Python Classes and Objects

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Example 2 : Creating Methods in Python

script.py   IPython Shell

```python
1   class Parrot:
2
3       # instance attributes
4       def __init__(self, name, age):
5           self.name = name
6           self.age = age
7
8       # instance method
9       def sing(self, song):
10          return "{} sings {}".format(self.name, song)
11
12      def dance(self):
13          return "{} is now dancing".format(self.name)
14
15  # instantiate the object
16  blu = Parrot("Blu", 10)
17
18  # call our instance methods
19  print(blu.sing("'Happy'"))
20  print(blu.dance())
```

Run   ●

Powered by DataCamp

When we run program, the output will be:

```
Blu sings 'Happy'
Blu is now dancing
```

In the above program, we define two methods i.e `sing()` and `dance()`. These are called instance method because they are called on an instance object i.e `blu`.

Inheritance

Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Example 3: Use of Inheritance in Python

script.py   IPython Shell

```python
1   # parent class
```

```python
 5            print( Bird is ready )
 6
 7        def whoisThis(self):
 8            print("Bird")
 9
10        def swim(self):
11            print("Swim faster")
12
13    # child class
14    class Penguin(Bird):
15
16        def __init__(self):
17            # call super() function
18            super().__init__()
19            print("Penguin is ready")
20
21        def whoisThis(self):
22            print("Penguin")
23
24        def run(self):
25            print("Run faster")
26
27    peggy = Penguin()
28    peggy.whoisThis()
29    peggy.swim()
30    peggy.run()
```

Run   ⬤

Powered by DataCamp

When we run this program, the output will be:

```
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from swim() method. Again, the child class modified the behavior of parent class. We can see this from whoisThis() method. Furthermore, we extend the functions of parent class, by creating a new run() method.

Additionally, we use super() function before __init__() method. This is because we want to pull the content of __init__() method from the parent class into the child class.

Encapsulation

attribute using underscore as prefix i.e single " _ " or double " __".

Example 4: Data Encapsulation in Python

```python
script.py      IPython Shell
1    class Computer:
2
3        def __init__(self):
4            self.__maxprice = 900
5
6        def sell(self):
7            print("Selling Price: {}".format(self.__maxprice))
8
9        def setMaxPrice(self, price):
10            self.__maxprice = price
11
12   c = Computer()
13   c.sell()
14
15   # change the price
16   c.__maxprice = 1000
17   c.sell()
18
19   # using setter function
20   c.setMaxPrice(1000)
21   c.sell()
```

Run ●

Powered by DataCamp

When we run this program, the output will be:

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

In the above program, we defined a class Computer . We use __init__() method to store the maximum selling price of computer. We tried to modify the price. However, we can't change it because Python treats the __maxprice as private attributes. To change the value, we used a setter function i.e setMaxPrice() which takes price as parameter.

Polymorphism

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called

Example 5: Using Polymorphism in Python

```
script.py      IPython Shell
1    class Parrot:
2
3        def fly(self):
4            print("Parrot can fly")
5
6        def swim(self):
7            print("Parrot can't swim")
8
9    class Penguin:
10
11       def fly(self):
12           print("Penguin can't fly")
13
14       def swim(self):
15           print("Penguin can swim")
16
17   # common interface
18   def flying_test(bird):
19       bird.fly()
20
21   #instantiate objects
22   blu = Parrot()
23   peggy = Penguin()
24
25   # passing the object
26   flying_test(blu)
27   flying_test(peggy)
```

Run   ⬤

Powered by DataCamp

When we run above program, the output will be:

```
Parrot can fly
Penguin can't fly
```

In the above program, we defined two classes `Parrot` and `Penguin` . Each of them have common method `fly()` method. However, their functions are different. To allow polymorphism, we created common interface i.e `flying_test()` function that can take any object. Then, we passed the objects `blu` and `peggy` in the `flying_test()` function, it ran effectively.

Key Points to Remember:

- The programming gets easy and efficient.
- The class is sharable, so codes can be reused.
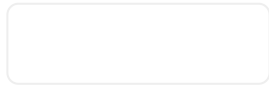
TUTORIALS

Python Tutorials

C Tutorials

Java Tutorials

Kotlin Tutorials

C++ Tutorials

Swift Tutorials

R Tutorials

Algorithms Tutorials

EXAMPLES

Python Examples

C Examples

Java Examples

Kotlin Examples

C++ Examples

R Examples

COMPANY

About

Advertising

Contact

LEGAL

Privacy Policy

Terms And Conditions