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

Parrot is an object,

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

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

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

The example for object of parrot class can be:

```
obj = Parrot()
```

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

Suppose we have details of parrot. Now, we are going to show how to build the class and objects of parrot.

Example 1: Creating Class and Object in Python

script.py IPython Shell

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
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
```

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

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

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)
```

```
# call our instance methods
print(blu.sing("'Happy'"))
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

```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")

    def whoisThis(self):
        print("Bird")

    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):

    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

```

```
24      def run(self):
25          print("Run faster")
26
27  peggy = Penguin()
28  peggy.whoisThis()
29  peggy.swim()
```

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

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single " _ " or double " __".

Example 4: Data Encapsulation in Python

script.py IPython Shell

```
1   class Computer:
2
3       def __init__(self):
4           self.__maxprice = 900
5
```

```
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 Polymorphism.

Example 5: Using Polymorphism in Python

script.py IPython Shell

```
1   class Parrot:
2
```

```
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)
```

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.
- The productivity of programmars increases
- Data is safe and secure with data abstraction.

PREVIOUS
PYTHON USER-DEFINED EXCEPTION

NEXT
PYTHON OBJECTS AND CLASS

Python Tutorial