Python Object Oriented Programming

## 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

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

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.

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

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

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](https://www.programiz.com/python-programming/class)

## 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

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)

# instantiate the object

blu = Parrot("Blu", 10)

# call our instance methods

print(blu.sing("'Happy'"))

print(blu.dance())

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

# 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")

def whoisThis(self):

print("Penguin")

def run(self):

print("Run faster")

peggy = Penguin()

peggy.whoisThis()

peggy.swim()

peggy.run()

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

class Computer:

def \_\_init\_\_(self):

self.\_\_maxprice = 900

def sell(self):

print("Selling Price: {}".format(self.\_\_maxprice))

def setMaxPrice(self, price):

self.\_\_maxprice = price

c = Computer()

c.sell()

# change the price

c.\_\_maxprice = 1000

c.sell()

# using setter function

c.setMaxPrice(1000)

c.sell()

RUN

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

class Parrot:

def fly(self):

print("Parrot can fly")

def swim(self):

print("Parrot can't swim")

class Penguin:

def fly(self):

print("Penguin can't fly")

def swim(self):

print("Penguin can swim")

# common interface

def flying\_test(bird):

bird.fly()

#instantiate objects

blu = Parrot()

peggy = Penguin()

# passing the object

flying\_test(blu)

flying\_test(peggy)

RUN

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.\

# Python Static, Class and Abstract methods

Many times while code reviews, I have seen people defining their classes and methods but incorrectly using decorators that Python provides for methods. So today I am going to talk more about these decorators and how and why to use them. We will see couple of real world scenarios where we can use them.

In this blog post, we will first take an example and then will break it down to dig more into these concepts. So here’s our first example:

class ExampleClass(object):

def some\_method(self):

pass

@classmethod

def class\_method(cls):

pass

@staticmethod

def static\_method():

pass

# Instance / Object methods:

In our example above, some\_method() is called the object method or instance method. The method takes one paramater self, which points to an instance of class ExampleClass when the method is called.

As we already know, using self parameter, instance methods can access member functions and other member variables of the class. This is the method that you will use most of the times while programming in Python.

Let’s take a simplistic example:

Let us say we have a simple Animal class. Animal class has a member variable as count that stores the count of the total Animals.

class Animal(object):

count = 1

def get\_count(self):

return self.count

# access the get\_count() by instance of the class

animal = Animal()

animal.get\_count() # 1

So we can access the get\_count method by referencing class instance animal. What would happen if we try to do this:

class Animal(object):

count = 1

def get\_count(self):

return self.count

print(Animal.get\_count())

# TypeError: get\_count() missing 1 required positional argument: 'self'

When executing above, we get TypeError: get\_count() missing 1 required positional argument: 'self'. This means the methods of the class can only be called by referencing to instances of that class. If we want to use Animal.get\_count() as it is, we can make use of class methods.

# Class Methods:

Class methods in Python can be defined by assigning @classmethod to any method. The thing to note here is that the class methods take cls parameter that points to a class and not the instance of the class.

Well what does that mean? Let us break this down. class\_method() takes cls as argument that means any class that we specify here is able to access to its own members and not the instance state. I know it’s a bit hard to understand. Hence, let us take a simple example to see what exactly are class methods.

We have our Animal class defined as:

class Animal(object):

count = 1

def get\_count(self):

return self.count

# access the get\_count() by instance of the class

animal = Animal()

animal.get\_count() # 1

Now, if we want to increase the count by accessing the class itself and not by its instance, we can make use of class method like:

class Animal(object):

count = 1

def get\_count(self):

return self.count

@classmethod

def inc\_count(cls):

cls.count += 1 # increment the count one

return cls()

animal = Animal.inc\_count() # inc count by accessing class directly

animal.get\_count() # 2

animal.inc\_count() # inc count by referencing the class instance (not recommended)

animal.get\_count() # 3

So long story short, class methods give us the power to access a method using reference to class itself rather than instance of that class.

Another thing to note here that self and cls are just the naming conventions that are followed by most of the Python programmers. You can give any name for those parameters just the mandatory thing is that they should be placed first in the arguments list.

# Static Methods:

Now, let’s see what are static methods. Like any function that we call, static methods can be called in a similar way. Hence, static methods are like regular functions just with the fact that they belong to a class’s namespace.

Static methods can be defined by decorating methods with @staticmethod decorator.

As static methods do not take self and cls as parameters, they do not have access to class members and variables.

class Example(object):

@staticmethod

def just\_another\_method():

print('This is static method')

example = Example()

example.just\_another\_method() # This is static method

# Abstract Methods:

Abstract methods in Python are pretty much different than class methods and static methods. However, while writing Object Orientated programs, abstract methods are used often.

Abstract methods in Python are the methods that are defined in the base class, but do not have any implementation. The derived class must override these abstract methods in their definition. Failing to do so will cause NotImplementedError.

For those who are familiar with Java programming language, abstract method in Python are equivalent to interface methods in Java.

class BaseClass(object):

def do\_something(self):

raise NotImplementedError

Any class that will inherit BaseClass should override and implement the do\_something() method, otherwise an exception would be thrown.

## 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.