Python OOPs Concepts

Last Updated : 08 Apr, 2024

Object Oriented Programming is a fundamental concept in Python, empowering developers to build modular, maintainable, and scalable applications. By understanding the core OOP principles—classes, objects, inheritance, encapsulation, polymorphism, and abstraction—programmers can leverage the full potential of Python's OOP capabilities to design elegant and efficient solutions to complex problems.

What is Object-Oriented Programming in Python?

In Python object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of object-oriented Programming (OOPs) or oops concepts in Python is to bind the data and the functions that work together as a single unit so that no other part of the code can access this data.

OOPs Concepts in Python

- Class in Python
- Objects in Python
- Polymorphism in Python
- Encapsulation in Python
- Inheritance in Python
- Data Abstraction in Python

Python OOPs Concepts

Python Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

To understand the need for creating a class let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
    # Statement-1
    .
    .
    .
    # Statement-N
```

Creating an Empty Class in Python

Python

```python
# Python3 program to
# demonstrate defining
# a class

class Dog:
    pass
```

Python Objects

In object oriented programming Python, The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string “Hello, world” is an object, a list is an object that can hold other objects, and so on. You’ve been using objects all along and may not even realize it.

An object consists of:

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.
- State or Attributes can be considered as the breed, age, or color of the dog.
- The behavior can be considered as to whether the dog is eating or sleeping.

Creating an Object

This will create an object named obj of the class Dog defined above. Before diving deep into objects and classes let us understand some basic keywords that will be used while working with objects and classes.

Python3

```
obj = Dog()
```

The Python self

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.
3. This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

Note: For more information, refer to self in the Python class

The Python __init__ Method

The __init__ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object. Now let us define a class and create some objects using the self and __init__ method.

Creating a class and object with class and instance attributes

```
class Dog:

    # class attribute
    attr1 = "mammal"

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

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class attributes
print("Rodger is a {}".format(Rodger.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))

# Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

Output

```
Rodger is a mammal
Tommy is also a mammal
My name is Rodger
My name is Tommy
```

Creating Classes and objects with methods

Here, The Dog class is defined with two attributes:

- **attr1** is a class attribute set to the value "**mammal**". Class attributes are shared by all instances of the class.
- __init__ is a special method (constructor) that initializes an instance of the Dog class. It takes two parameters: self (referring to the instance being created) and name (representing the name of the dog). The name parameter is used to assign a name attribute to each instance of Dog. The speak method is defined within the Dog class. This method prints a

The driver code starts by creating two instances of the Dog class: Rodger and Tommy. The __init__ method is called for each instance to initialize their name attributes with the provided names. The speak method is called in both instances (Rodger.speak() and Tommy.speak()), causing each dog to print a statement with its name.

Python3

```
class Dog:

    # class attribute
    attr1 = "mammal"

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

    def speak(self):
        print("My name is {}".format(self.name))

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class methods
Rodger.speak()
Tommy.speak()
```

Output

```
My name is Rodger
My name is Tommy
```

Note: For more information, refer to Python Classes and Objects

Python Inheritance

In Python object oriented Programming, Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class

- It represents real-world relationships well.
- It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance

- **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.
- **Multilevel Inheritance:** Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.
- **Hierarchical Inheritance:** Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.
- **Multiple Inheritance:** Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

Inheritance in Python

In the above article, we have created two classes i.e. Person (parent class) and Employee (Child Class). The Employee class inherits from the Person class. We can use the methods of the person class through the employee class as seen in the display function in the above code. A child class can also modify the behavior of the parent class as seen through the details() method.

Python3

```
# Python code to demonstrate how parent constructors
# are called.

# parent class
class Person(object):

    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
```

```
        def details(self):
            print("My name is {}".format(self.name))
            print("IdNumber: {}".format(self.idnumber))

    # child class
    class Employee(Person):
        def __init__(self, name, idnumber, salary, post):
            self.salary = salary
            self.post = post

            # invoking the __init__ of the parent class
            Person.__init__(self, name, idnumber)

        def details(self):
            print("My name is {}".format(self.name))
            print("IdNumber: {}".format(self.idnumber))
            print("Post: {}".format(self.post))


    # creation of an object variable or an instance
    a = Employee('Rahul', 886012, 200000, "Intern")

    # calling a function of the class Person using
    # its instance
    a.display()
    a.details()
```

Output

```
Rahul
886012
My name is Rahul
IdNumber: 886012
Post: Intern
```

Note: For more information, refer to our Inheritance in Python tutorial.

Python Polymorphism

In object oriented Programming Python, Polymorphism simply means having many forms. For example, we need to determine if the given species of birds fly or not, using polymorphism we can do this using a single function.

This code demonstrates the concept of Python oops inheritance and method overriding in Python classes. It shows how subclasses can override methods defined in their parent class to provide specific behavior while still inheriting other methods from the parent class.

Python3

```
class Bird:

    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):

    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):

    def flight(self):
        print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```


Output

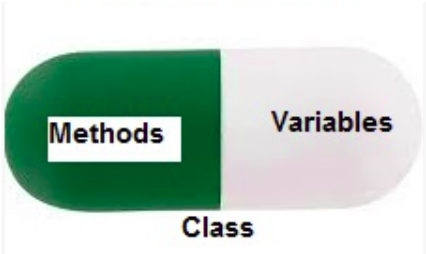
```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

Note: For more information, refer to our Polymorphism in Python Tutorial.

Python Encapsulation

In Python object oriented programming, Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.



Encapsulation in Python

In the above example, we have created the c variable as the private attribute. We cannot even access this attribute directly and can't even change its value.

Python3

```
# Python program to
# demonstrate private members
# "__" double underscore represents private attribute.
# Private attributes start with "__".

# Creating a Base class
class Base:
    def __init__(self):
        self.a = "GeeksforGeeks"
        self.__c = "GeeksforGeeks"

# Creating a derived class
class Derived(Base):
    def __init__(self):
```

```
            print(self.__c)


    # Driver code
    obj1 = Base()
    print(obj1.a)

    # Uncommenting print(obj1.c) will
    # raise an AttributeError

    # Uncommenting obj2 = Derived() will
    # also raise an AtrributeError as
    # private member of base class
    # is called inside derived class
```

Output

```
GeeksforGeeks
```

Note: for more information, refer to our Encapsulation in Python Tutorial.

Data Abstraction

It hides unnecessary code details from the user. Also, when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.

Data Abstraction in Python can be achieved by creating abstract classes.

Object Oriented Programming in Python | Set 2 (Data Hiding and Object Printing)

Similar Reads

Shuffle a deck of card with OOPS in Python

The objective is to distribute a deck of cards among two players. The code for the Shuffle Deck of Cards in Python can be used to shuffle the cards. The shuffle method, which is a...

3 min read

__invert__ and __abs__ magic functions in Python OOPS

The __invert__ and __abs__ magic methods are used for implementing unary operators in Python. In this article, we will see the concept of __invert__ and __abs__ magic functions...

4 min read

In this article, we will compare and highlight the features of aggregation and Composition in Python OOPS. [caption width="800"] [/caption]Concept of Inheritance Inheritance is a...

5 min read

30 OOPs Interview Questions and Answers (2024)

Object-Oriented Programming, or OOPs, is a programming paradigm that implements the concept of objects in the program. It aims to provide an easier solution to real-world...

16 min read

Top 10 Advance Python Concepts That You Must Know

Python is a high-level, object-oriented programming language that has recently been picked up by a lot of students as well as professionals due to its versatility, dynamic...

6 min read

View More Articles

Article Tags : python-oop-concepts Python

Practice Tags : python

GeeksforGeeks
Sanchhaya Education Private Limited
A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305

Mathematics
Physics
Chemistry
Biology
Social Science
English Grammar

Accountancy
Business Studies
Economics
Management
HR Management
Finance
Income Tax

Databases

SQL
MYSQL
PostgreSQL
PL/SQL
MongoDB

Preparation Corner

Company-Wise Recruitment Process
Resume Templates
Aptitude Preparation
Puzzles
Company-Wise Preparation
Companies
Colleges

Competitive Exams

JEE Advanced
UGC NET
UPSC
SSC CGL
SBI PO
SBI Clerk
IBPS PO
IBPS Clerk

More Tutorials

Software Development
Software Testing
Product Management
Project Management
Linux
Excel
All Cheat Sheets

Free Online Tools

Typing Test
Image Editor
Code Formatters
Code Converters
Currency Converter
Random Number Generator
Random Password Generator

Write & Earn

Write an Article
Improve an Article
Pick Topics to Write
Share your Experiences
Internships