Python OOPs Concepts

Last Updated : 17 Mar, 2025

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 OOP capabilities to design elegant and efficient solutions to complex problems.

OOPs is a way of organizing code that uses objects and classes to represent real-world entities and their behavior. In OOPs, object has attributes thing that has specific data and can perform certain actions using methods.

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. Classes are blueprints for creating objects. A class defines a set of attributes and methods that the created objects (instances) can have.

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. Example: Myclass.Myattribute

Creating a Class

Here, the class keyword indicates that we are creating a class followed by name of the class (Dog in this case).

```
class Dog:
    species = "Canine"  # Class attribute

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

- **species:** A class attribute shared by all instances of the class.
- **__init__ method:** Initializes the name and age attributes when a new object is created.

Note: *For more information, refer to python classes.*

Python Objects

An Object is an instance of a Class. It represents a specific implementation of the class and holds its own data.

An object consists of:

- **State:** It is represented by the attributes and reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object and 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.

Creating Object

```
class Dog:
    species = "Canine"  # Class attribute

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

# Creating an object of the Dog class
dog1 = Dog("Buddy", 3)

print(dog1.name)
print(dog1.species)
```

Output

```
Buddy
Canine
```

Explanation:

- **dog1 = Dog("Buddy", 3):** Creates an object of the Dog class with name as "Buddy" and age as 3.
- **dog1.name:** Accesses the instance attribute name of the dog1 object.
- **dog1.species:** Accesses the class attribute species of the dog1 object.

Note: *For more information, refer to python objects.*

Self Parameter

self parameter is a reference to the current instance of the class. It allows us to access the attributes and methods of the object.

```
class Dog:
    species = "Canine"  # Class attribute
```

```
dog1 = Dog("Buddy", 3)  # Create an instance of Dog
dog2 = Dog("Charlie", 5)  # Create another instance of Dog

print(dog1.name, dog1.age, dog1.species)  # Access instance
and class attributes
print(dog2.name, dog2.age, dog2.species)  # Access instance
and class attributes
print(Dog.species)  # Access class attribute directly
```

Output

```
Buddy 3 Canine
Charlie 5 Canine
Canine
```

Explanation:

- **self.name:** Refers to the name attribute of the object (dog1) calling the method.
- **dog1.bark():** Calls the bark method on dog1.

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

__init__ Method

__init__ method is the constructor in Python, automatically called when a new object is created. It initializes the attributes of the class.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

dog1 = Dog("Buddy", 3)
print(dog1.name)
```

```
Buddy
```

Explanation:

- **__init__:** Special method used for initialization.
- **self.name and self.age:** Instance attributes initialized in the constructor.

Class and Instance Variables

In Python, variables defined in a class can be either class variables or instance variables, and understanding the distinction between them is crucial for object-oriented programming.

Class Variables

These are the variables that are shared across all instances of a class. It is defined at the class level, outside any methods. All objects of the class share the same value for a class variable unless explicitly overridden in an object.

Instance Variables

Variables that are unique to each instance (object) of a class. These are defined within the __init__ method or other instance methods. Each object maintains its own copy of instance variables, independent of other objects.

```
class Dog:
    # Class variable
    species = "Canine"

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

# Create objects
dog1 = Dog("Buddy", 3)
```

```
print(dog1.species)  # (Class variable)
print(dog1.name)     # (Instance variable)
print(dog2.name)     # (Instance variable)

# Modify instance variables
dog1.name = "Max"
print(dog1.name)     # (Updated instance variable)

# Modify class variable
Dog.species = "Feline"
print(dog1.species)  # (Updated class variable)
print(dog2.species)
```

Output

```
Canine
Buddy
Charlie
Max
Feline
Feline
```

Explanation:

- **Class Variable (species):** Shared by all instances of the class. Changing Dog.species affects all objects, as it's a property of the class itself.
- **Instance Variables (name, age):** Defined in the __init__ method. Unique to each instance (e.g., dog1.name and dog2.name are different).
- **Accessing Variables:** Class variables can be accessed via the class name (Dog.species) or an object (dog1.species). Instance variables are accessed via the object (dog1.name).
- **Updating Variables:** Changing Dog.species affects all instances. Changing dog1.name only affects dog1 and does not impact dog2.

Python Inheritance

promotes code reuse.

Types of Inheritance:

1. **Single Inheritance:** A child class inherits from a single parent class.
2. **Multiple Inheritance:** A child class inherits from more than one parent class.
3. **Multilevel Inheritance:** A child class inherits from a parent class, which in turn inherits from another class.
4. **Hierarchical Inheritance:** Multiple child classes inherit from a single parent class.
5. **Hybrid Inheritance:** A combination of two or more types of inheritance.

```
# Single Inheritance
class Dog:
    def __init__(self, name):
        self.name = name

    def display_name(self):
        print(f"Dog's Name: {self.name}")

class Labrador(Dog):  # Single Inheritance
    def sound(self):
        print("Labrador woofs")

# Multilevel Inheritance
class GuideDog(Labrador):  # Multilevel Inheritance
    def guide(self):
        print(f"{self.name}Guides the way!")

# Multiple Inheritance
class Friendly:
    def greet(self):
        print("Friendly!")
```

```
        print("Golden Retriever Barks")

# Example Usage
lab = Labrador("Buddy")
lab.display_name()
lab.sound()

guide_dog = GuideDog("Max")
guide_dog.display_name()
guide_dog.guide()

retriever = GoldenRetriever("Charlie")
retriever.display_name()
retriever.greet()
retriever.sound()
```

Explanation:

- **Single Inheritance:** Labrador inherits Dog's attributes and methods.
- **Multilevel Inheritance:** GuideDog extends Labrador, inheriting both Dog and Labrador functionalities.
- **Multiple Inheritance:** GoldenRetriever inherits from both Dog and Friendly.

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

Python Polymorphism

Polymorphism allows methods to have the same name but behave differently based on the object's context. It can be achieved through method overriding or overloading.

Types of Polymorphism

1. **Compile-Time Polymorphism**: This type of polymorphism is

input parameters or usage. It is commonly referred to as method or operator overloading.

2. **Run-Time Polymorphism**: This type of polymorphism is determined during the execution of the program. It occurs when a subclass provides a specific implementation for a method already defined in its parent class, commonly known as method overriding.

Code Example:

```
# Parent Class
class Dog:
    def sound(self):
        print("dog sound")  # Default implementation

# Run-Time Polymorphism: Method Overriding
class Labrador(Dog):
    def sound(self):
        print("Labrador woofs")  # Overriding parent method

class Beagle(Dog):
    def sound(self):
        print("Beagle Barks")  # Overriding parent method

# Compile-Time Polymorphism: Method Overloading Mimic
class Calculator:
    def add(self, a, b=0, c=0):
        return a + b + c  # Supports multiple ways to call add

# Run-Time Polymorphism
dogs = [Dog(), Labrador(), Beagle()]
for dog in dogs:
    dog.sound()  # Calls the appropriate method based on the ol
type


# Compile-Time Polymorphism (Mimicked using default arguments)
```

```
print(calc.add(5, 10, 15))  # Three arguments
```

Explanation:

1. Run-Time Polymorphism:

- Demonstrated using method overriding in the Dog class and its subclasses (Labrador and Beagle).
- The correct sound method is invoked at runtime based on the actual type of the object in the list.

2. Compile-Time Polymorphism:

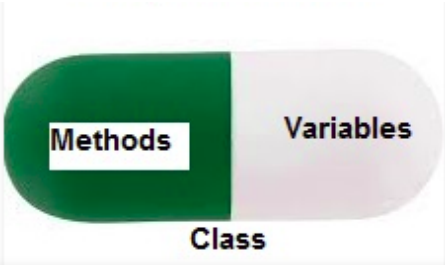
- Python does not natively support method overloading. Instead, we use a single method (add) with default arguments to handle varying numbers of parameters.
- Different behaviors (adding two or three numbers) are achieved based on how the method is called.

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

Python Encapsulation

Encapsulation is the bundling of data (attributes) and methods (functions) within a class, restricting access to some components to control interactions.

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



1. **Public Members**: Accessible from anywhere.
2. **Protected Members**: Accessible within the class and its subclasses.
3. **Private Members**: Accessible only within the class.

Code Example:

```
class Dog:
    def __init__(self, name, breed, age):
        self.name = name  # Public attribute
        self._breed = breed  # Protected attribute
        self.__age = age  # Private attribute

    # Public method
    def get_info(self):
        return f"Name: {self.name}, Breed: {self._breed},
Age: {self.__age}"

    # Getter and Setter for private attribute
    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Invalid age!")

# Example Usage
dog = Dog("Buddy", "Labrador", 3)

# Accessing public member
print(dog.name)  # Accessible

# Accessing protected member
print(dog._breed)  # Accessible but discouraged outside the
class
```

```
# Modifying private member using setter
dog.set_age(5)
print(dog.get_info())
```

Explanation:

- **Public Members:** Easily accessible, such as name.
- **Protected Members**: Used with a single _, such as _breed. Access is discouraged but allowed in subclasses.
- **Private Members:** Used with __, such as __age. Access requires getter and setter methods.

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

Data Abstraction

Abstraction hides the internal implementation details while exposing only the necessary functionality. It helps focus on "what to do" rather than "how to do it."

Types of Abstraction:

- **Partial Abstraction:** Abstract class contains both abstract and concrete methods.
- **Full Abstraction:** Abstract class contains only abstract methods (like interfaces).

Code Example:

```
from abc import ABC, abstractmethod

class Dog(ABC):  # Abstract Class
    def __init__(self, name):
        self.name = name
```

```
        pass

    def display_name(self):  # Concrete Method
        print(f"Dog's Name: {self.name}")

class Labrador(Dog):  # Partial Abstraction
    def sound(self):
        print("Labrador Woof!")

class Beagle(Dog):  # Partial Abstraction
    def sound(self):
        print("Beagle Bark!")

# Example Usage
dogs = [Labrador("Buddy"), Beagle("Charlie")]
for dog in dogs:
    dog.display_name()  # Calls concrete method
    dog.sound()  # Calls implemented abstract method
```

Explanation:

- **Partial Abstraction**: The `Dog` class has both abstract (`sound`) and concrete (`display_name`) methods.
- **Why Use It**: Abstraction ensures consistency in derived classes by enforcing the implementation of abstract methods.

OOPs Quiz:

- Python OOPS Quiz

Related Articles:

- Python Classes and Objects
- Constructors in Python
- Encapsulation in Python
- Abstract Classes in Python

- Method Overriding in Python
- Operator Overloading in Python

Recommended Problems:

- Design a class
- Constructor
- Encapsulation in Python
- Abstraction in Python
- Static Method In Python
- Inheritance in Python
- Multiple Inheritence in Python
- Method Overriding in Python
- Operator Overloading In Python

Similar Reads

1. Python Functions
2. Concrete Exceptions in Python
3. Python objects
4. Concurrency in Python
5. Python Crash Course
6. Python Naming Conventions
7. Python object
8. Python Docstrings
9. Python vs Cpython
10. Python | os.read() method