Contents

Class and Object in Python

In **Object-Oriented Programming (OOP)**, a **class** is a blueprint for creating objects, while an **object** is an instance of a class. Python uses classes to encapsulate data and methods that operate on the data.

---

1. Definition of Class and Object

- **Class:** A template or blueprint that defines attributes (variables) and behaviors (methods) of objects.
- **Object:** An instance of a class that has real values assigned to the attributes.

---

2. Syntax of a Class in Python

```
class ClassName:
    # Constructor method to initialize attributes
    def __init__(self, parameter1, parameter2):
        self.attribute1 = parameter1
        self.attribute2 = parameter2

    # Method (Function) inside class
    def display(self):
        print("Attribute 1:", self.attribute1)
        print("Attribute 2:", self.attribute2)

# Creating an Object
```

```
object_name = ClassName(value1, value2)

# Calling a Method
object_name.display()
```

---

3. Example of Class and Object Implementation

```
class Car:
    # Constructor method to initialize object properties
    def __init__(self, brand, model, year):
        self.brand = brand  # Attribute 1
        self.model = model  # Attribute 2
        self.year = year    # Attribute 3

    # Method to display car details
    def show_details(self):
        print(f"Car Brand: {self.brand}")
        print(f"Car Model: {self.model}")
        print(f"Manufacturing Year: {self.year}")

# Creating objects of class Car
car1 = Car("Toyota", "Camry", 2022)
car2 = Car("Honda", "Civic", 2021)

# Calling methods using objects
car1.show_details()
print()  # Blank line for better readability
car2.show_details()
```

---

4. Output of the Above Code

Car Brand: Toyota

Car Model: Camry

Manufacturing Year: 2022

Car Brand: Honda

Car Model: Civic

Manufacturing Year: 2021

---

5. Explanation of the Example

- **class Car:** → Defines a class named Car.
- **def __init__(self, brand, model, year):** → Constructor method that initializes object attributes.
- **self.brand = brand** → Assigns values to the object.
- **def show_details(self):** → Method to display object details.
- **car1 = Car("Toyota", "Camry", 2022)** → Creating an object car1.
- **car1.show_details()** → Calls the method to print car details.

Encapsulation and Access Modifiers in Python

1. Encapsulation in Python

Encapsulation is one of the fundamental principles of **Object-Oriented Programming (OOP)**. It refers to bundling the data (**variables**) and methods (**functions**) into a **single unit (class)** while **restricting direct access** to some details.

Advantages of Encapsulation:

- **Data Hiding:** Protects data from unintended modification.
- **Improves Security:** Restricts access to specific data.
- **Code Flexibility & Maintainability:** Methods control how data is modified.

---

2. Access Modifiers in Python

Python has **three types** of access modifiers:

1. **Public (public)** - Accessible anywhere.
2. **Protected (_protected)** - Accessible within the class and subclasses.

3. **Private (__private)** - Accessible only within the class.

---

3. Example of Encapsulation with Access Modifiers

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number  # Public attribute
        self._account_type = "Savings"        # Protected attribute
        self.__balance = balance              # Private attribute

    # Public method to display account details
    def display_info(self):
        print(f"Account Number: {self.account_number}")
        print(f"Account Type: {self._account_type}")
        print(f"Balance: ${self.__balance}")

    # Private method (Accessible only inside the class)
    def __update_balance(self, amount):
        self.__balance += amount

    # Public method to deposit money (Uses private method)
    def deposit(self, amount):
        if amount > 0:
            self.__update_balance(amount)
            print(f"Deposited: ${amount}")
        else:
            print("Invalid deposit amount!")

# Creating an object of BankAccount
account = BankAccount("12345678", 1000)

# Accessing Public Attribute
print("Public Attribute:", account.account_number)  # Works

# Accessing Protected Attribute (Not recommended but possible)
print("Protected Attribute:", account._account_type)  # Works but should be
accessed carefully

# Accessing Private Attribute (Will cause an error)
# print("Private Attribute:", account.__balance)  # AttributeError
```

```
# Accessing Private Attribute using name mangling
print("Private Attribute (Using Name Mangling):",
account._BankAccount__balance)  # Works

# Calling Public Method
account.display_info()

# Calling Public Method that uses Private Method
account.deposit(500)
```

---

4. Output of the Above Code

Public Attribute: 12345678

Protected Attribute: Savings

Private Attribute (Using Name Mangling): 1000

Account Number: 12345678

Account Type: Savings

Balance: $1000

Deposited: $500

---

5. Explanation of Access Modifiers

Modifier	Syntax	Accessibility
Public	self.variable	Accessible from anywhere
Protected	self._variable	Should be accessed only within the class and subclasses
Private	self.__variable	Cannot be accessed directly outside the class

- **Public attributes/methods** are accessible anywhere.

- **Protected attributes/methods** are **not strictly private**, but they are intended to be used **within the class or subclasses**.
- **Private attributes/methods** use **name mangling** (_ClassName__variable) to prevent direct access.

Inheritance in Python

1. What is Inheritance?

Inheritance is one of the fundamental concepts of **Object-Oriented Programming (OOP)**. It allows a class (**child class**) to inherit attributes and methods from another class (**parent class**). This helps in **code reusability** and **reducing redundancy**.

---

2. Types of Inheritance in Python

1. **Single Inheritance** – One child class inherits from one parent class.
2. **Multiple Inheritance** – A child class inherits from multiple parent classes.
3. **Multilevel Inheritance** – A class inherits from another 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 multiple types of inheritance.

---

3. Example of Different Types of Inheritance

(A) Single Inheritance

```
# Parent Class
class Animal:
    def make_sound(self):
        print("Animal makes a sound")

# Child Class (Inheriting from Animal)
class Dog(Animal):
    def bark(self):
```

```
        print("Dog barks!")

# Creating an object of Dog
dog = Dog()
dog.make_sound()  # Inherited method from Animal class
dog.bark()        # Method of Dog class
```

Output:

Animal makes a sound

Dog barks!

(B) Multiple Inheritance

```
# Parent Class 1
class Engine:
    def start(self):
        print("Engine started")

# Parent Class 2
class Wheels:
    def rotate(self):
        print("Wheels are rotating")

# Child Class (Inheriting from both Engine and Wheels)
class Car(Engine, Wheels):
    def drive(self):
        print("Car is moving")

# Creating an object of Car
car = Car()
car.start()    # From Engine class
car.rotate()   # From Wheels class
car.drive()    # From Car class
```

Output:

Engine started

Wheels are rotating

Car is moving

(C) Multilevel Inheritance

```
# Grandparent Class
class LivingBeing:
    def breathe(self):
        print("Breathing...")

# Parent Class (Inheriting from LivingBeing)
class Animal(LivingBeing):
    def eat(self):
        print("Eating...")

# Child Class (Inheriting from Animal)
class Dog(Animal):
    def bark(self):
        print("Barking...")

# Creating an object of Dog
dog = Dog()
dog.breathe()  # From LivingBeing
dog.eat()      # From Animal
dog.bark()     # From Dog
```

Output:

Breathing...

Eating...

Barking...

(D) Hierarchical Inheritance

```
# Parent Class
class Vehicle:
    def move(self):
        print("Vehicle is moving")
```

```
# Child Class 1
class Car(Vehicle):
    def drive(self):
        print("Car is driving")

# Child Class 2
class Bike(Vehicle):
    def ride(self):
        print("Bike is riding")

# Creating objects of Car and Bike
car = Car()
bike = Bike()

car.move()  # Inherited from Vehicle
car.drive()  # From Car

bike.move()  # Inherited from Vehicle
bike.ride()  # From Bike
```

Output:

Vehicle is moving

Car is driving

Vehicle is moving

Bike is riding

(E) Hybrid Inheritance

Hybrid inheritance is a mix of multiple types of inheritance.

```
# Parent Class
class A:
    def method_A(self):
        print("Method from class A")

# Child Class 1 (Single Inheritance)
class B(A):
    def method_B(self):
```

```
        print("Method from class B")

# Child Class 2 (Hierarchical Inheritance)
class C(A):
    def method_C(self):
        print("Method from class C")

# Child Class 3 (Multiple Inheritance)
class D(B, C):
    def method_D(self):
        print("Method from class D")

# Creating an object of D
obj = D()
obj.method_A()  # Inherited from A
obj.method_B()  # Inherited from B
obj.method_C()  # Inherited from C
obj.method_D()  # From D
```

Output:

Method from class A

Method from class B

Method from class C

Method from class D

5. Method Overriding in Inheritance

If a child class has the **same method** as the parent class, the **child class method overrides** the parent's method.

```
class Animal:
    def sound(self):
        print("Animals make sound")

class Dog(Animal):
    def sound(self):  # Overriding the parent class method
        print("Dog barks!")

dog = Dog()
```

```
dog.sound()  # Calls the overridden method in the child class
```

Output:

Dog barks!

---

6. super() Function in Inheritance

The **super()** function allows a child class to access methods of its parent class.

```
class Parent:
    def show(self):
        print("This is the Parent class")

class Child(Parent):
    def show(self):
        super().show()  # Calls the parent class method
        print("This is the Child class")

obj = Child()
obj.show()
```

Output:

This is the Parent class

This is the Child class

What is Data Abstraction?

Data Abstraction is an **Object-Oriented Programming (OOP)** concept that **hides the internal implementation** of a class and only exposes **essential features** to the user. It helps in simplifying complex systems by breaking them into smaller, more manageable parts.

Key Benefits of Abstraction:

✓ **Hides unnecessary details** from the user.
✓ **Improves code modularity and maintainability.**
✓ **Protects data from direct modification.**
✓ **Enhances security and usability.**

---

2. How is Abstraction Achieved in Python?

Python provides **abstraction** using:

1. **Abstract Classes**
2. **Abstract Methods**
3. **The abc module** (ABC and @abstractmethod)

---

3. Abstract Classes and Methods in Python

- An **abstract class** is a class that **cannot be instantiated** (i.e., objects cannot be created from it).
- An **abstract method** is a method **declared in an abstract class but without implementation**. Child classes **must override** it.

(A) Syntax for Abstraction

```
from abc import ABC, abstractmethod  # Importing ABC module

# Abstract Class
class Vehicle(ABC):

    @abstractmethod
    def start(self):
        pass  # Abstract method (must be implemented in child class)

    @abstractmethod
    def stop(self):
        pass  # Abstract method (must be implemented in child class)

# Concrete Class (Inheriting from Vehicle)
class Car(Vehicle):
    def start(self):
        print("Car starts with a key")

    def stop(self):
        print("Car stops when brakes are applied")
```

```
# Creating an object of Car
car = Car()
car.start()
car.stop()
_____
```

4. Output of the Above Code

Car starts with a key

Car stops when brakes are applied

---

5. Explanation

Concept	Description
ABC class	Used to define an abstract class.
@abstractmethod	Used to declare a method as abstract (must be implemented in the child class).
Abstract Class (Vehicle)	Defines abstract methods start() and stop() but does not implement them.
Concrete Class (Car)	Inherits Vehicle and provides implementations for start() and stop().
Object Creation	We create an object of Car, not Vehicle (because Vehicle is abstract).

---

6. Example: ATM Machine (Real-Life Abstraction)

```
from abc import ABC, abstractmethod

# Abstract Class
class ATM(ABC):
    @abstractmethod
    def withdraw(self, amount):
        pass
```

```
    @abstractmethod
    def deposit(self, amount):
        pass

# Concrete Class
class BankATM(ATM):
    def withdraw(self, amount):
        print(f"Withdrew ${amount} from the ATM")

    def deposit(self, amount):
        print(f"Deposited ${amount} into the ATM")

# Creating an object of BankATM
atm = BankATM()
atm.deposit(500)
atm.withdraw(200)
```

Output:

Deposited $500 into the ATM

Withdrew $200 from the ATM

Python Top 30 Questions

1. What is a class in Python?

- **Answer:** A class is a blueprint for creating objects. It defines attributes and methods.
- **Example:**

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def display(self):
        print(f"{self.brand} {self.model}")
```

2. What is an object in Python?

- **Answer:** An object is an instance of a class.
- **Example:**

```
my_car = Car("Tesla", "Model S")
my_car.display()  # Output: Tesla Model S
```

3. What is the __init__ method?

- **Answer:** The __init__ method is a constructor used to initialize object attributes.
- **Example:**

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


4. What is inheritance in Python?

- **Answer:** Inheritance allows a class to inherit attributes and methods from another class.
- **Example:**

class Animal:

```
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

dog = Dog()
dog.speak()  # Output: Animal speaks
dog.bark()   # Output: Dog barks
```

5. What is method overriding?

- **Answer:** Method overriding allows a subclass to provide a specific implementation of a method already defined in its superclass.
- **Example:**

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

dog = Dog()
dog.speak()  # Output: Dog barks
```

6. What is polymorphism in Python?

- **Answer:** Polymorphism allows objects of different classes to be treated as objects of a common superclass.
- **Example:**

```
class Cat:
    def speak(self):
        print("Meow")

class Dog:
    def speak(self):
        print("Bark")

def animal_sound(animal):
    animal.speak()

animal_sound(Cat())  # Output: Meow
animal_sound(Dog())  # Output: Bark
```

7. What is encapsulation in Python?

- **Answer:** Encapsulation is the bundling of data and methods that operate on the data into a single unit (class). It restricts direct access to some of the object's components.
- **Example:**

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance
```

8. What are private members in Python?

- **Answer:** Private members are attributes or methods prefixed with __ (double underscore). They cannot be accessed directly outside the class.
- **Example:**

```
class Person:
    def __init__(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

p = Person("Alice")
print(p.get_name())  # Output: Alice
# print(p.__name)    # Error: AttributeError
```

9. What is the difference between __init__ and __new__?

- **Answer:**
    - __new__: Creates a new instance of the class.
    - __init__: Initializes the instance after it is created.
- **Example:**

```
class MyClass:
    def __new__(cls):
```

```
        print("Creating instance")
        return super().__new__(cls)

    def __init__(self):
        print("Initializing instance")

obj = MyClass()
```

10. What is a static method?

- **Answer:** A static method belongs to the class rather than an instance. It does not require a self parameter.
- **Example:**

```
class Math:
    @staticmethod
    def add(a, b):
        return a + b

print(Math.add(2, 3))  # Output: 5
```


11. What is a class method?

- **Answer:** A class method belongs to the class and takes cls as the first parameter. It can modify class state.
- **Example:**

```
class MyClass:
    count = 0

    @classmethod
    def increment_count(cls):
        cls.count += 1

MyClass.increment_count()
print(MyClass.count)  # Output: 1
```

12. What is the difference between @staticmethod and @classmethod?

- **Answer:**

- @staticmethod: Does not take self or cls as a parameter.
- @classmethod: Takes cls as a parameter and can modify class state.

---

13. What is method chaining in Python?

- **Answer:** Method chaining allows calling multiple methods in a single statement.
- **Example:**

```
class Calculator:
    def __init__(self, value=0):
        self.value = value

    def add(self, num):
        self.value += num
        return self

    def multiply(self, num):
        self.value *= num
        return self

calc = Calculator()
calc.add(5).multiply(2)
print(calc.value)  # Output: 10
```

---

14. What is the super() function?

- **Answer:** The super() function is used to call a method from the parent class.
- **Example:**

```
class Parent:
    def greet(self):
        print("Hello from Parent")

class Child(Parent):
    def greet(self):
        super().greet()
```

```
        print("Hello from Child")

c = Child()
c.greet()
# Output:
# Hello from Parent
# Hello from Child
```

---

15. What is multiple inheritance?

- **Answer:** Multiple inheritance allows a class to inherit from more than one parent class.
- **Example:**

```
class A:
    def greet(self):
        print("Hello from A")

class B:
    def greet(self):
        print("Hello from B")

class C(A, B):
    pass

c = C()
c.greet()  # Output: Hello from A (due to method resolution order)
```

16. What is the method resolution order (MRO)?

- **Answer:** MRO defines the order in which base classes are searched for a method or attribute.
- **Example:**

```
class A:
    pass

class B(A):
    pass
```

```
class C(A):
    pass

class D(B, C):
    pass

print(D.mro())  # Output: [D, B, C, A, object]
```

17. What is the __str__ method?

- **Answer:** The __str__ method returns a string representation of an object.
- **Example:**

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

    def __str__(self):
        return f"Person: {self.name}"

p = Person("Alice")
print(p)  # Output: Person: Alice
```

18. What is the __repr__ method?

- **Answer:** The __repr__ method returns an unambiguous string representation of an object, often used for debugging.
- **Example:**

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

    def __repr__(self):
        return f"Person(name='{self.name}')"

p = Person("Alice")
print(repr(p))  # Output: Person(name='Alice')
```

19. What is the difference between __str__ and __repr__?

- **Answer:**
  - __str__: User-friendly string representation.
  - __repr__: Unambiguous string representation for debugging.

20. What is the __del__ method?

- **Answer:** The __del__ method is called when an object is about to be destroyed.
- **Example:**

```
class MyClass:
    def __del__(self):
        print("Object destroyed")

obj = MyClass()
del obj  # Output: Object destroyed
```

21. What is the __call__ method?

- **Answer:** The __call__ method allows an object to be called like a function.
- **Example:**

```
class Adder:
    def __call__(self, a, b):
        return a + b

add = Adder()
print(add(2, 3))  # Output: 5
```

22. What is the __getitem__ method?

- **Answer:** The __getitem__ method allows an object to support indexing.
- **Example:**

```
class MyList:
    def __init__(self, items):
```

```
        self.items = items

    def __getitem__(self, index):
        return self.items[index]

my_list = MyList([1, 2, 3])
print(my_list[1])  # Output: 2
```

---

23. What is the __setitem__ method?

- **Answer:** The __setitem__ method allows an object to support item assignment.
- **Example:**

```
class MyList:
    def __init__(self, items):
        self.items = items

    def __setitem__(self, index, value):
        self.items[index] = value

my_list = MyList([1, 2, 3])
my_list[1] = 5
print(my_list.items)  # Output: [1, 5, 3]
```

---

24. What is the __len__ method?

- **Answer:** The __len__ method allows an object to define its length.
- **Example:**

```
class MyList:
    def __init__(self, items):
        self.items = items

    def __len__(self):
        return len(self.items)

my_list = MyList([1, 2, 3])
print(len(my_list))  # Output: 3
```

---

25. What is the __iter__ method?

- **Answer:** The __iter__ method allows an object to be iterable.
- **Example:**

```
class MyRange:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __iter__(self):
        return iter(range(self.start, self.end))

for i in MyRange(1, 4):
    print(i)  # Output: 1 2 3
```

---

26. What is the __contains__ method?

- **Answer:** The __contains__ method allows an object to support the in operator.
- **Example:**

```
class MyList:
    def __init__(self, items):
        self.items = items

    def __contains__(self, item):
        return item in self.items

my_list = MyList([1, 2, 3])
print(2 in my_list)  # Output: True
```

---

27. What is the __add__ method?

- **Answer:** The __add__ method allows an object to support the + operator.
- **Example:**

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
print(p3.x, p3.y)  # Output: 4 6
```

28. What is the __eq__ method?

- **Answer:** The __eq__ method allows an object to support the == operator.
- **Example:**

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

p1 = Point(1, 2)
p2 = Point(1, 2)
print(p1 == p2)  # Output: True
```

---

29. What is the __hash__ method?

- **Answer:** The __hash__ method allows an object to be hashable, enabling it to be used as a dictionary key.
- **Example:**

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

    def __hash__(self):
        return hash(self.name)

p = Person("Alice")
print(hash(p))
```

30. What is the __slots__ attribute?

- **Answer:** The __slots__ attribute restricts the creation of new attributes in an object, saving memory.
- **Example:**

```
class Person:
    __slots__ = ['name', 'age']

    def __init__(self, name, age):
        self.name = name
        self.age = age

p = Person("Alice", 25)
# p.address = "123 Street"  # Error: AttributeError
```