Introduction to Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design software. Objects represent real-world entities and are created from "classes." Python is an object-oriented language and supports all the major features of OOP.

Classes and Objects

Class

A class is a blueprint for creating objects. It defines a set of attributes and behaviors (methods) that the created objects will have.

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

    def display(self):
        print(f"Name: {self.name}, Roll No: {self.roll_no}")
```

Object

An object is an instance of a class. It holds actual data and can call methods defined in the class.

```
# Creating an object of Student class
s1 = Student("Alice", 101)
s1.display()
```

Encapsulation in Python

Encapsulation is one of the four fundamental concepts of Object-Oriented Programming (OOP), alongside abstraction, inheritance, and polymorphism. It refers to the practice of **bundling data (attributes)** and **methods (functions)** that operate on that data into a

single unit—typically a class—and **restricting direct access** to some of the object's components.

Purpose of Encapsulation

The primary goals of encapsulation include:

- **Data hiding: Prevent external components from accessing internal object details.**
- **Improved modularity: Encourages dividing code into manageable sections.**
- **Increased security: Prevents unintended interference and misuse of data.**
- **Ease of maintenance: Changes to internal implementations don't affect external code.**

Encapsulation in Python

Unlike some other languages like Java or C++, Python does not have strong access modifiers like private or protected. However, it uses **naming conventions** and **name mangling** to achieve a level of encapsulation.

Access Modifiers in Python (By Convention)

Modifier Type	Syntax	Accessibility
Public	**var_name**	**Accessible from anywhere**
Protected	**_var_name**	**Internal use only; accessible, but discouraged**
Private	**__var_name**	**Name mangling makes it harder to access**

Public Members

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

```
    def display(self):
        print("Name:", self.name)

s = Student("Alice")
s.display()
print(s.name)
```

Protected Members

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

    def _display(self):
        print("Name:", self._name)

s = Student("Bob")
s._display()
print(s._name)
```

Private Members (Name Mangling)

```
class Student:
    def __init__(self, name):
        self.__name = name  # private attribute

    def __display(self):
        print("Name:", self.__name)
```

```
    def show(self):
        self.__display()

s = Student("Carol")
s.show()
# print(s.__name)      # Error
# s.__display()        # Error

# Access via name mangling
print(s._Student__name)
```

Getter and Setter Methods

To access and modify private variables safely, use getter and setter methods:

```
class Student:
    def __init__(self):
        self.__name = ""

    def set_name(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

s = Student()
s.set_name("David")
print(s.get_name())
```

With Property Decorators

```
class Student:
    def __init__(self):
        self.__name = ""

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, value):
        self.__name = value

s = Student()
s.name = "Eva"
print(s.name)
```

Advantages of Encapsulation

- Prevents unintended data modification.
- Improves code readability and organization.
- Enhances software maintenance and evolution.
- Provides control over how important data is accessed and updated.

Inheritance in Python

Inheritance is one of the core concepts of Object-Oriented Programming (OOP). It allows

one class (child or derived class) to inherit attributes and methods from another class (parent or base class).

Why Use Inheritance?

- **Code Reusability**: Reuse code from existing classes.
- **Extensibility**: Add new features to existing classes without modifying them.
- **Data Organization**: Logical hierarchy of classes.
- **Polymorphism**: Use the same method name but with different behaviors in child classes.

Syntax of Inheritance in Python

```
class ParentClass:
    # attributes and methods

class ChildClass(ParentClass):
    # additional attributes and methods
```

Example:

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

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

# Using the classes
dog = Dog()
dog.speak()   # Inherited from Animal
dog.bark()    # Defined in Dog
```

Types of Inheritance in Python

Type of Inheritance	Description	Example Class Structure
Single	One child, one parent	A → B
Multiple	One child, multiple parents	A, B → C
Multilevel	Grandparent → Parent → Child	A → B → C
Hierarchical	One parent, multiple children	A → B, C
Hybrid	Combination of above	A, B → C → D

1. Single Inheritance

One child class inherits from one parent class.

```
class Parent:
    def show(self):
        print("Parent class")

class Child(Parent):
    def display(self):
        print("Child class")

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

2. Multiple Inheritance

A child class inherits from more than one parent class.

```
class Father:
```

```
    def skills(self):
        print("Programming")

class Mother:
    def hobbies(self):
        print("Painting")

class Child(Father, Mother):
    pass

obj = Child()
obj.skills()
obj.hobbies()
```

3. Multilevel Inheritance

A class inherits from a child class, making a chain.

```
class Grandparent:
    def house(self):
        print("Owns a house")

class Parent(Grandparent):
    def car(self):
        print("Owns a car")

class Child(Parent):
    def bike(self):
        print("Owns a bike")
```

```
c = Child()
c.house()
c.car()
c.bike()
```

4. Hierarchical Inheritance

Multiple child classes inherit from a single parent class.

```
class Parent:
    def show(self):
        print("Parent class")

class Child1(Parent):
    def display1(self):
        print("Child 1")

class Child2(Parent):
    def display2(self):
        print("Child 2")

c1 = Child1()
c1.show()
c1.display1()

c2 = Child2()
c2.show()
c2.display2()
```

5. Hybrid Inheritance

Combination of two or more types of inheritance.

```
class A:
    def methodA(self):
        print("Method A")
class B(A):
    def methodB(self):
        print("Method B")
class C:
    def methodC(self):
        print("Method C")
class D(B, C):
    def methodD(self):
        print("Method D")
d = D()
d.methodA()
d.methodB()
d.methodC()
d.methodD()
```

Method Overriding

When a child class provides a specific implementation of a method that is already defined in its parent class.

```
class Parent:
    def show(self):
        print("Parent's show")
```

```
class Child(Parent):
    def show(self):
        print("Child's show")

obj = Child()
obj.show()  # Child's show
```

The super() Function

Used to call the parent class's methods inside the child class.

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

class Child(Parent):
    def greet(self):
        super().greet()
        print("Hello from Child")

obj = Child()
obj.greet()
```

Key Points

- Python supports multiple and multilevel inheritance.
- The super() function is useful for calling parent methods.
- Inheritance enables modular, reusable, and extendable code.
- Method Resolution Order (MRO) is used in multiple inheritance to resolve

method calls.

Method Resolution Order (MRO)

In Python, **MRO (Method Resolution Order)** is the order in which base classes are searched when executing a method or looking up an attribute. It's especially relevant in **multiple inheritance**.

Example **with** MRO: Using super() properly

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

class B(A):
    def greet(self):
        print("Hello from B")
        super().greet()  # This follows MRO

class C(A):
    def greet(self):
        print("Hello from C")
        super().greet()  # This also follows MRO

class D(B, C):
    def greet(self):
        print("Hello from D")
        super().greet()  # MRO determines the next class

d = D()
d.greet()
```

Example **without** MRO: Not using super()

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

class B(A):
    def greet(self):
        print("Hello from B")
        A.greet(self)  # Direct call

class C(A):
    def greet(self):
        print("Hello from C")
        A.greet(self)  # Direct call

class D(B, C):
    def greet(self):
        print("Hello from D")
        B.greet(self)
        C.greet(self)

d = D()
d.greet()
```

Polymorphism in Python

Polymorphism is a core concept in object-oriented programming (OOP). The word

"polymorphism" comes from the Greek words *poly* (many) and *morph* (form). It refers to the ability of different objects to respond to the same function call or operation in different ways.

In **Python**, polymorphism allows functions, methods, or operators to behave differently based on the object or data type they are acting upon.

Types of Polymorphism

1. Duck Typing (Dynamic Typing)

Python is dynamically typed, which means types are determined at runtime. If an object implements a required method or behavior, it can be used, regardless of its actual class.

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

class Dog:
    def speak(self):
        return "Woof"

def animal_sound(animal):
    print(animal.speak())

c = Cat()
d = Dog()

animal_sound(c)
animal_sound(d)
```

This is an example of **duck typing**: "If it walks like a duck and quacks like a duck, it must

be a duck."

2. Operator Overloading

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

```
print(1 + 2)

# concatenate two strings
print("Geeks"+"For")

# Product two numbers
print(3 * 4)

# Repeat the String
print("Geeks"*4)
```

Output

```
3
GeeksFor
12
GeeksGeeksGeeksGeeks
```

How to overload the operators in Python?

Consider that we have two objects which are a physical representation of a class (user-

defined data type) and we have to add two objects with binary '+' operator it throws an error, because compiler don't know how to add two objects. So we define a method for an operator and that process is called operator overloading. We can overload all existing operators but we can't create a new operator. To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined.

Overloading binary + operator in Python:

When we use an operator on user-defined data types then automatically a special function or magic function associated with that operator is invoked. Changing the behavior of operator is as simple as changing the behavior of a method or function. You define methods in your class and operators work according to that behavior defined in methods. When we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined. Thereby changing this magic method's code, we can give extra meaning to the + operator.

How Does the Operator Overloading Actually work?

Whenever you change the behavior of the **existing operator** through operator overloading, you have to redefine the special function that is invoked automatically when the operator is used with the objects.

For Example:

```
# Python Program illustrate how
# to overload an binary + operator
# And how it actually works

class A:
    def __init__(self, a):
```

```
        self.a = a

    # adding two objects
    def __add__(self, o):
        return self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("Geeks")
ob4 = A("For")

print(ob1 + ob2)
print(ob3 + ob4)
# Actual working when Binary Operator is used.
print(A.__add__(ob1 , ob2))
print(A.__add__(ob3,ob4))
#And can also be Understand as :
print(ob1.__add__(ob2))
print(ob3.__add__(ob4))
```

3. Method Overriding (Runtime Polymorphism)

When a child class defines a method that already exists in the parent class with the same name and parameters, it **overrides** the parent method.

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

class Dog(Animal):
```

```
    def speak(self):
        print("Dog barks")

a = Animal()
d = Dog()

a.speak()
d.speak()
```

This is **runtime polymorphism**, where the method call is resolved at runtime.

Polymorphism with Functions and Objects

Python supports polymorphism with functions and classes.

```
class Bird:
    def fly(self):
        print("Bird can fly")

class Airplane:
    def fly(self):
        print("Airplane can fly")

def lets_fly(thing):
    thing.fly()

b = Bird()
a = Airplane()
```

```
lets_fly(b)
lets_fly(a)
```

The same function lets_fly() behaves differently based on the passed object's class.

Polymorphism with Inheritance

```
class Vehicle:
    def start(self):
        print("Starting vehicle...")

class Car(Vehicle):
    def start(self):
        print("Starting car...")

class Bike(Vehicle):
    def start(self):
        print("Starting bike...")

vehicles = [Car(), Bike()]

for v in vehicles:
    v.start()
```

The base class reference can point to child class objects. This is classical polymorphism in OOP.

Advantages of Polymorphism

- Increases **code reusability**.
- Makes code **more readable and maintainable**.
- Supports **extensibility**—new classes can be added without changing existing code.
- Promotes **loose coupling** between components.

Real-world Analogy

A person can act as:

- A student in a classroom
- A customer in a shop
- A player in a game

Each role has a different behavior, even though it is the same person. Similarly, in programming, one interface (method name) can behave differently for different object types.

What is Python Constructor

In Python, a constructor is a special method called when an object is created. Its purpose is to assign values to the data members within the class when an object is initialized. The name of the constructor method is always **__init__**.

Here is an example of a simple class with a constructor:

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

person = Person("John", 30)
print(person.name)
print(person.age)
```

Output:

John

30

In this example, the **__init__** method is called when the ***Person*** object is created, and it sets the ***name*** and ***age*** attributes of the object.

The **__init__** method is commonly referred to as the "constructor" because it is responsible for constructing the object. It is called automatically when the object is created, and it is used to initialize the object's attributes.

Types of Python Constructor

In Python, there are two types of constructors:

- **Default Constructor:** A default constructor is a constructor that takes no arguments. It is used to create an object with default values for its attributes.
- **Parameterized Constructor:** A parameterized constructor is a constructor that takes one or more arguments. It is used to create an object with custom values for its attributes.
- **Non-Parameterized Constructor:** A non-parameterized constructor is a constructor that does not take any arguments. It is a special method in Python that is called when you create an instance of a class. The non-parameterized constructor is used to initialize the default values for the instance variables of the object.

Default Constructors

Default constructors are useful when you want to create an object with a predefined set of attributes, but you don't want to specify the values of those attributes when the object is created.

Here is an example of a default constructor:

```
class Person:
    def __init__(self):
```

```
        self.name = "John"
        self.age = 30

person = Person()
print(person.name)
print(person.age)
```

Output:

John

30

In this example, the **__init__** method is the default constructor for the ***Person*** class. It is called automatically when the object is created, and it sets the default values for the ***name*** and ***age*** attributes.

Parameterized Constructors

Parameterized constructors are useful when you want to create an object with custom values for its attributes. They allow you to specify the values of the object's attributes when the object is created, rather than using default values.

Here is an example of a class with a parameterized constructor:

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

person = Person("Alice", 25)
print(person.name)
print(person.age)
```

Output:

Alice

In this example, the **__init__** method is the parameterized constructor for the ***Person*** class. It takes two arguments, ***name*** and ***age***, and it sets the values of the *name* and *age* attributes of the object to the values of these arguments.

Non-Parameterized Constructors

There is not necessarily a need for a non-parameterized constructor in Python. It is up to the programmer to decide whether to include a non-parameterized constructor in a class.

However, a non-parameterized constructor can be useful in the following cases:

- When you want to initialize the default values for the instance variables of an object.
- When you want to perform some operations when an object is created, such as opening a file or establishing a connection to a database.
- When you want to create a “skeleton” object that can be used as a template for creating other objects.

For example, consider the following class:

```
class MyClass:
    def __init__(self):
        self.arg1 = 10
        self.arg2 = 20
```

In this case, the non-parameterized constructor is used to initialize the default values for the instance variables ***arg1*** and ***arg2***. If you create an instance of the ***MyClass*** class without passing any arguments, the default values will be used.

```
class MyClass:
    def __init__(self):
        self.arg1 = 10
        self.arg2 = 20
```

```
obj = MyClass()
print(obj.arg1)
print(obj.arg2)
```

Output:

```
10
20
```

Note: If you do not define a non-parameterized constructor in your class, Python will automatically provide one for you. However, it is a good practice to define a constructor for your class, even if it is a non-parameterized constructor so that you have control over the initialization of your objects.

Rules of Python Constructor

Here are some rules for defining constructors in Python:

- The constructor method must be named **__init__**. This is a special name that is recognized by Python as the constructor method.
- The first argument of the constructor method must be **self**. This is a reference to the object itself, and it is used to access the object's attributes and methods.
- The constructor method must be defined inside the class definition. It cannot be defined outside the class.
- The constructor method is called automatically when an object is created. You don't need to call it explicitly.
- You can define both default and parameterized constructors in a class. If you define both, the parameterized constructor will be used when you pass arguments to the object constructor, and the default constructor will be used when you don't pass any arguments.

Multiple Constructors in Single Class

You can have more than one constructor in a single Python class. This is known as "method overloading". To do this, you will need to use the same method name (in this case, the name of the method will be the same as the name of the class) but define the method with different numbers or types of arguments.

Here is an example of a class with two constructors:

```
class MyClass:
    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2

    def __init__(self, arg1):
        self.arg1 = arg1
        self.arg2 = None
```

The first constructor takes two arguments and sets them as instance variables, while the second constructor takes only one argument and sets it as an instance variable. When you create an instance of the ***MyClass*** class, Python will use the appropriate constructor based on the number of arguments that you pass.

For example:

```
class MyClass:
    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2

    def __init__(self, arg1):
        self.arg1 = arg1
        self.arg2 = None
```

obj1 = MyClass(10, 20)

obj2 = MyClass(30)

In the first case, the first constructor will be called and ***arg1*** will be set to 10 and ***arg2*** will be set to 20. In the second case, the second constructor will be called and ***arg1*** will be set to 30 and ***arg2*** will be set to **None**.

Note: Python does not have true method overloading like some other programming languages. When you define multiple methods with the same name in a single class, only the last one will be used. However, you can use default values for arguments to achieve a similar effect.

Difference Between Constructor and Destructor

Parameters	Constructor	Destructor
Purpose	Initializes an object when it is created	Cleans up the object before it is destroyed
Method Name	__init__()	__del__()
Execution	Called automatically when an object is created	Called automatically when an object is deleted
Arguments	Can take arguments to initialize variables	Does not take arguments
Use Case	Used to set up an object's properties	Used to release resources like files or memory

What is a Destructor?

A destructor is a special method in Python that is executed when an object is deleted. It is used to free resources, such as closing files or database connections. The destructor method is named __del__().

Syntax of Destructor

```
class ClassName:
    def __del__(self):
        # Code to execute when the object is destroyed
```

Example of Destructor

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print("Student object created!")
    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")
    def __del__(self):
        print("Student object destroyed!")
# Creating an object
student1 = Student("Alice", 20)
student1.display()
# Deleting the object
del student1
```

Output:

```
Student object created!
Name: Alice, Age: 20
Student object destroyed!
```

Explanation:

- The __del__() method executes when the object is deleted.

- The del student1 statement explicitly destroys student1, triggering __del__().

- This ensures proper resource cleanup when objects are no longer needed.

Example of Constructor and Destructor Together

Below is an example demonstrating both a constructor and a destructor in a Python class:

```
class FileHandler:
    def __init__(self, filename, mode):
        self.file = open(filename, mode)
        print("File opened successfully.")
    def write_data(self, data):
        self.file.write(data)
        print("Data written to file.")
    def __del__(self):
        self.file.close()
        print("File closed successfully.")
# Creating an object
file = FileHandler("sample.txt", "w")
file.write_data("Hello, Python!")
# Deleting the object
del file
```

Output:

```
File opened successfully.
```

Data written to file.

File closed successfully.

Explanation:

- The constructor (__init__()) opens a file for writing.

- The write_data() method writes data to the file.

- The destructor (__del__()) ensures the file is closed when the object is deleted.

Python Classes/Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class:

Example

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

Create Object

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example

Create a class named Person, use the __init__() function to assign values for name and age:

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Note: The __init__() function is called automatically every time the class is being used to create a new object.

The __str__() Function

The __str__() function controls what should be returned when the class object is

represented as a string.

If the __str__() function is not set, the string representation of the object is returned:

Example

The string representation of an object WITHOUT the __str__() function:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1)
```

Example

The string representation of an object WITH the __str__() function:

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

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

p1 = Person("John", 36)

print(p1)
```

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the

object.

Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

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

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Note: The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example

Use the words *mysillyobject* and *abc* instead of *self*:

```
class Person:
  def __init__(mysillyobject, name, age):
```

```
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Modify Object Properties

You can modify properties on objects like this:

Example

Set the age of p1 to 40:

```
p1.age = 40
```

Delete Object Properties

You can delete properties on objects by using the del keyword:

Example

Delete the age property from the p1 object:

```
del p1.age
```

Delete Objects

You can delete objects by using the del keyword:

Example

Delete the p1 object:

```
del p1
```

The pass Statement

class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

Example

```
class Person:
  pass
```

Instance Variables vs Class Variables

➤ **Instance Variables:**

- Belong to **individual objects**.
- Defined using self inside the class.
- Unique to each object.

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

car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")
print(car1.brand)   # Output: Toyota
print(car2.brand)   # Output: Honda
```

Class Variables:

- Shared **across all instances** of the class.
- Defined **outside of any method** in the class.

```
class Car:
    wheels = 4   # class variable
```

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

car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")

print(car1.wheels)  # Output: 4
print(car2.wheels)  # Output: 4
```

If you change the class variable using the class name, it affects all instances:

```
Car.wheels = 6
print(car1.wheels)  # Output: 6
print(car2.wheels)  # Output: 6
```

But if you override it using an object:

```
car1.wheels = 8
print(car1.wheels)  # Output: 8 (only for car1)
print(car2.wheels)  # Output: 6
```

Composition in python

In Python, "composition" is a fundamental concept in object-oriented programming (OOP) where a class is composed of other classes. Instead of inheriting behavior (like in inheritance), a class *contains* instances of other classes and uses their functionality to achieve its own purpose. This creates a "has-a" relationship, as opposed to the "is-a" relationship of inheritance.

Think of it like building a car: a car "has a" engine, "has" wheels, "has" seats, etc. The car

class doesn't inherit from an Engine class; rather, it uses an instance of an Engine to perform its driving functions.

Key Characteristics of Composition:

- **"Has-A" Relationship:** The primary indicator of composition. One object "has" or "contains" another object.
- **Delegation:** The outer class often delegates tasks to its contained objects.
- **Loose Coupling:** Changes in the contained objects are less likely to affect the outer class, as long as the interface remains consistent. This makes your code more flexible and easier to maintain.
- **Reusability:** You can reuse smaller, specialized classes to build more complex ones.
- **Flexibility:** You can easily swap out or change the contained objects without modifying the core logic of the outer class.

Example of Composition in Python:

Imagine a Computer that **has a** CPU and **has** RAM.

```
class CPU:
    def __init__(self, cores):
        self.cores = cores

    def process(self):
        return f"CPU with {self.cores} cores is processing data."

class RAM:
    def __init__(self, size_gb):
        self.size_gb = size_gb

    def store(self):
```

```
        return f"{self.size_gb}GB RAM is storing data."

class Computer:
    def __init__(self, cpu_cores, ram_size):
        # Composition: Computer has a CPU
        self.cpu = CPU(cpu_cores)
        # Composition: Computer has RAM
        self.ram = RAM(ram_size)

    def boot_up(self):
        cpu_status = self.cpu.process()
        ram_status = self.ram.store()
        return f"Computer booting up: {cpu_status} and {ram_status}"

# Create a computer
my_pc = Computer(cpu_cores=8, ram_size=16)

# Use the computer's functionality
print(my_pc.boot_up())
```

How It Works

- The CPU and RAM classes define simple components.
- The Computer class doesn't inherit from CPU or RAM. Instead, in its __init__ method, it **creates instances** of CPU and RAM and assigns them to self.cpu and self.ram.
- When you call my_pc.boot_up(), the Computer class **delegates** the "processing" and "storing" tasks to its contained cpu and ram objects.

Benefits in this Example:

- **Clearer Relationships:** It's obvious that a car is made up of an engine and tires, not that it *is* an engine or a tire.
- **Modularity:** Engine and Tire are separate, self-contained units. You could use them in other contexts (e.g., an Engine in a boat, a Tire in a bicycle).
- **Flexibility:** If you wanted to change the type of engine (e.g., an electric engine), you could create a new ElectricEngine class and simply change which Engine instance is created in the Car's __init__ without modifying the Car's drive() or park() methods significantly (as long as the start() and stop() methods have compatible interfaces).
- **Reduced Complexity:** Each class focuses on its own specific responsibility.

When to Choose Composition Over Inheritance:

- **"Has-A" vs. "Is-A":** If the relationship between two classes is "A has a B" (e.g., a car has an engine), use composition. If it's "A is a B" (e.g., a dog is an animal), use inheritance.
- **Code Reuse:** Composition promotes code reuse by allowing you to assemble objects with existing functionality.
- **Flexibility and Maintainability:** Composition generally leads to more flexible and maintainable code because it reduces tight coupling between classes.
- **Avoiding the "Diamond Problem":** Multiple inheritance can lead to ambiguity issues (the "diamond problem"). Composition avoids this by not relying on a strict inheritance hierarchy.

In essence, composition is about building complex objects from simpler, interchangeable parts, leading to more robust and adaptable software designs.

Method Overloading and Method Overriding in Python

Comparison Table

Method Overloading	Method Overriding
Refers to defining multiple methods with the same name but different parameters	Refers to defining a method in a subclass that has the same name as the one in its superclass
Can be achieved in Python using default arguments	Can be achieved by defining a method in a subclass with the same name as the one in its superclass
Allows a class to have multiple methods with the same name but different behaviors based on the input parameters	Allows a subclass to provide its own implementation of a method defined in its superclass
The choice of which method to call is determined at compile-time based on the number and types of arguments passed to the method	The choice of which method to call is determined at runtime based on the actual object being referred to
Not supported natively in Python	Supported natively in Python

Method Overloading In Python

Method overloading in Python is a concept that allows a class to have multiple methods with the same name but different parameter lists. Unlike other programming languages like Java, Python does not directly support method overloading, where you can define multiple methods with different parameter lists.

How do you overload a method in Python?

In Python, method overloading is achieved using default or variable-length arguments (*args, **kwargs) to handle different parameter lists. The idea is to define a single method with a generic name that can accept multiple combinations of arguments.

Then, the appropriate logic inside the method will handle different cases based on the arguments passed. Here's how you can overload a method in Python:

Example using default arguments:

python Copy code

```python
class MathOperations:
    def add(self, a, b, c=0, d=0):
        return a + b + c + d

math_op = MathOperations()

# You can call the 'add' method with different numbers of arguments, and the
print(math_op.add(2, 3))          # Output: 5 (2 + 3 + 0 + 0)
print(math_op.add(2, 3, 4))       # Output: 9 (2 + 3 + 4 + 0)
print(math_op.add(2, 3, 4, 5))    # Output: 14 (2 + 3 + 4 + 5)
```

Example using variable-length arguments:

python Copy code

```python
class MathOperations:
    def add(self, *args):
        return sum(args)

math_op = MathOperations()

# The 'add' method can accept any number of arguments using *args, and you
print(math_op.add(2, 3))          # Output: 5 (2 + 3)
print(math_op.add(2, 3, 4))       # Output: 9 (2 + 3 + 4)
print(math_op.add(2, 3, 4, 5))    # Output: 14 (2 + 3 + 4 + 5)
```

Example using variable-length keyword arguments:

```python
class MathOperations:
    def add(self, **kwargs):
        a = kwargs.get('a', 0)
        b = kwargs.get('b', 0)
        c = kwargs.get('c', 0)
        d = kwargs.get('d', 0)
        return a + b + c + d

math_op = MathOperations()

# The 'add' method can accept keyword arguments using **kwargs, and you can
print(math_op.add(a=2, b=3))           # Output: 5 (2 + 3 + 0 + 0)
print(math_op.add(a=2, b=3, c=4))      # Output: 9 (2 + 3 + 4 + 0)
print(math_op.add(a=2, b=3, c=4, d=5)) # Output: 14 (2 + 3 + 4 + 5)
```

Using default or variable-length arguments, you can create methods in Python that can handle multiple argument combinations, effectively achieving method overloading-like behaviour.

Method Overriding

Method overriding is a concept in Object-Oriented Programming (OOP) where a subclass provides a specific implementation for a method already defined in its superclass. When an object of the subclass calls the overridden method, Python will use the implementation from the subclass instead of the one in the superclass.

Method overriding is based on inheritance, which is one of the critical features of OOP. Inheritance allows a class (subclass) to inherit properties and behaviors from another class (superclass). The subclass can then extend or modify the functionality of the superclass to create a more specialised version of the class.

Here's a step-by-step explanation of how to override a method in Python:

Superclass and Subclass Relationship:

- In Python, we define classes using the class keyword. The class being inherited is called the superclass (or parent class), and the class inheriting from the superclass is called the subclass (or child class).
- To create a subclass, include the superclass's name inside parentheses after the subclass name, like this: class SubclassName(SuperclassName):

Method Definition in Superclass:

- The superclass contains a method that you want to override in the subclass.
- The method in the superclass must have the same name as the method you want to override in the subclass. This is essential for method overriding to work.
- The method in the superclass can have any implementation you want to share among its subclasses.

Method Definition in Subclass:

- In the subclass, define a method with the same name and number of parameters as the method you want to override in the superclass.
- The method definition in the subclass will replace the implementation of the method in the superclass. This means when you call the method on an instance of the subclass, Python will use the overridden method from the subclass, not the one from the superclass.

Calling the Overridden Method:

- When you create an instance of the subclass and call the method, Python first looks for the process in the subclass.
- If Python finds the method in the subclass, it executes the overridden method from the subclass.
- If Python does not find the method in the subclass, it looks for the method in the superclass and executes it.

Here's an example of method overriding:

python Copy code

```python
class Animal:
    def make_sound(self):
        return "Some generic sound"

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Create instances of the subclasses
dog = Dog()
cat = Cat()

# Call the overridden method for each subclass
print(dog.make_sound())  # Output: "Woof!"
print(cat.make_sound())  # Output: "Meow!"
```

In this example, Animal is the superclass, and Dog and Cat are subclasses. Both Dog and Cat override the make_sound() method defined in the Animal class with their specific implementations.

Method overriding allows you to create more specialised and flexible classes in your code. It promotes code reuse, as you can define common behaviours in the superclass and customise them in the subclasses. This is one of the powerful tools in OOP that enables you to build complex and scalable applications in Python.

Abstraction in Programming: A Detailed Academic Note

Abstraction is a fundamental concept in computer science and object-oriented programming (OOP) that focuses on showing only essential information while hiding complex implementation details. This approach simplifies the system, improves efficiency, and enhances maintainability.

There are primarily two types of abstraction:

1. **Data Abstraction:** This type of abstraction hides the intricate internal structure and organization of data, presenting only the necessary information to the user. It allows users to interact with data in a simplified form without needing to know how it is stored or manipulated internally.
2. **Process Abstraction:** Also known as control abstraction, this involves hiding the underlying implementation steps of a particular process or function. Users can invoke a process by its name and provide inputs, without needing to understand the detailed sequence of operations that occur internally.

Abstraction in Object-Oriented Programming (Python Examples)

In object-oriented programming, abstraction is heavily utilized through concepts like abstract classes and abstract methods.

Abstract Classes

An **abstract class** is a blueprint for other classes. It cannot be instantiated directly, meaning you cannot create an object from an abstract class. Its primary purpose is to serve as a base class for other "concrete" classes that will inherit from it. Abstract classes often define a common interface or a set of methods that all its subclasses must implement.

In Python, to create an abstract class, you need to:

- Inherit from the ABC (Abstract Base Class) class, which is part of Python's abc module.
- At least one method within the class must be declared as an abstract method

using the @abstractmethod decorator.

Abstract Methods

An **abstract method** is a method declared in an abstract class but does not contain an implementation. It acts as a placeholder that *must* be overridden (implemented) by any concrete subclass that inherits from the abstract class. If a concrete class fails to implement all abstract methods of its abstract parent, it will also be considered an abstract class and cannot be instantiated.

Example in Python:

Consider the following Python code snippet demonstrating an abstract class and its usage:

```
from abc import ABC, abstractmethod

# Define an abstract class 'democlass'
class democlass(ABC):
    # Abstract method - must be implemented by subclasses
    @abstractmethod
    def method1(self):
        pass # No implementation here

    # Concrete method - can be implemented or overridden by subclasses
    def method2(self):
        print("Concrete method2 of democlass")

# Attempting to instantiate democlass directly will raise a TypeError
# obj = democlass() # This line would cause an error
```

```
# Define a concrete class 'concreteclass' that inherits from 'democlass'
class concreteclass(democlass):
    # Override the abstract method 'method1'
    def method1(self):
        print("Implemented method1 in concreteclass")

# Create an object of the concrete class
obj = concreteclass()

# Call the methods
obj.method1() # Output: Implemented method1 in concreteclass
obj.method2() # Output: Concrete method2 of democlass
```

Explanation of the Example:

- The democlass is declared as an abstract class by inheriting from ABC.
- method1 is an abstract method, marked with @abstractmethod. It has no body (pass), indicating that it must be implemented by subclasses.
- method2 is a concrete method, with an actual implementation. Subclasses can use it as is or override it.
- If you uncomment obj = democlass(), it would result in a TypeError because democlass is abstract and cannot be instantiated.
- The concreteclass inherits from democlass. To become a concrete (instantiable) class, it *must* provide an implementation for method1, which it does.
- An object obj is successfully created from concreteclass.
- Both method1 (the overridden version) and method2 (inherited from the abstract class) can be called on the obj instance.

This example clearly illustrates how abstraction in Python, through abstract classes and

methods, enforces a structure where common functionalities are defined at a higher level (abstract class), but their specific implementations are delegated to the concrete subclasses. This promotes modularity, reusability, and a clear contract between the base class and its derived classes.

Python - Interfaces

In software engineering, an **interface** is a software architectural pattern. It is similar to a class but its methods just have prototype signature definition without any executable code or implementation body. The required functionality must be implemented by the methods of any class that inherits the interface.

The method defined without any executable code is known as abstract method.

Interfaces in Python

In languages like Java and Go, there is keyword called interface which is used to define an interface. Python doesn't have it or any similar keyword. It uses **abstract base classes** (in short ABC module) and **@abstractmethod** decorator to create interfaces.

NOTE: In Python, abstract classes are also created using ABC module.

An abstract class and interface appear similar in Python. The only difference in two is that the abstract class may have some non-abstract methods, while all methods in interface must be abstract, and the implementing class must override all the abstract methods.

Rules for implementing Python Interfaces

We need to consider the following points while creating and implementing interfaces in Python −

- Methods defined inside an interface must be abstract.
- Creating object of an interface is not allowed.
- A class implementing an interface needs to define all the methods of that interface.

- In case, a class is not implementing all the methods defined inside the interface, the class must be declared abstract.

Ways to implement Interfaces in Python

We can create and implement interfaces in two ways −

- Formal Interface
- Informal Interface

Formal Interface

Formal interfaces in Python are implemented using abstract base class (ABC). To use this class, you need to import it from the **abc** module.

Example

In this example, we are creating a formal interface with two abstract methods.

```
from abc import ABC, abstractmethod

# creating interface
class demoInterface(ABC):
  @abstractmethod
  def method1(self):
    print ("Abstract method1")
    return

  @abstractmethod
  def method2(self):
    print ("Abstract method1")
    return
```

Let us provide a class that implements both the abstract methods.

```
# class implementing the above interface
class concreteclass(demoInterface):
   def method1(self):
      print("This is method1")
      return

   def method2(self):
      print("This is method2")
      return



# creating instance
obj = concreteclass()

# method call
obj.method1()
obj.method2()
```

Output

When you execute this code, it will produce the following output −

```
This is method1
This is method2
```

Informal Interface

In Python, the **informal interface** refers to a class with methods that can be overridden. However, the compiler cannot strictly enforce the implementation of all the provided methods.

This type of interface works on the principle of **duck typing**. It allows us to call any method on an object without checking its type, as long as the method exists.

Example

In the below example, we are demonstrating the concept of informal interface.

```
class demoInterface:
   def displayMsg(self):
      pass



class newClass(demoInterface):
   def displayMsg(self):
      print("This is my message")



# creating instance
obj = newClass()

# method call
obj.displayMsg()
```

Output

On running the above code, it will produce the following output −

```
This is my message
```

Class Decorators in Python

Class decorators are a powerful feature in Python that allow you to modify or enhance

the behavior of classes. Similar to function decorators, class decorators are functions that take a class as input and return a modified class or a completely new class.

Basic Syntax

```
@decorator
class MyClass:
    pass
```

This is equivalent to:

```
class MyClass:
    pass
MyClass = decorator(MyClass)
```

Simple Example

Here's a basic class decorator that adds a method to a class:

```
def add_greet_method(cls):
    def greet(self):
        return f"Hello from {self.__class__.__name__}!"

    cls.greet = greet
    return cls

@add_greet_method
class Person:
    pass

p = Person()
print(p.greet())
```

```
# Output: Hello from Person!
```

More Practical Example: Singleton Pattern

Class decorators are often used to implement design patterns like Singleton:

```
def singleton(cls):
    instances = {}

    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return get_instance

@singleton
class Database:
    def __init__(self):
        print("Initializing database...")

db1 = Database()  # Prints "Initializing database..."
db2 = Database()  # No output - same instance returned
print(db1 is db2)  # True
```

Class Decorator with Arguments

You can also create decorators that accept arguments:

```
def add_attributes(**kwargs):
    def decorator(cls):
```

```
        for key, value in kwargs.items():
            setattr(cls, key, value)
        return cls
    return decorator

@add_attributes(version="1.0", author="John Doe")
class MyClass:
    pass

print(MyClass.version)  # Output: 1.0
print(MyClass.author)   # Output: John Doe
```

Built-in Class Decorators

Python includes several useful built-in class decorators:

1. @dataclass (from Python 3.7+):

```
from dataclasses import dataclass
@dataclass
class Point:
    x: float
    y: float
```

2. @property, @classmethod, @staticmethod (though these are typically used as method decorators)

When to Use Class Decorators

Class decorators are useful when you want to:

- Add or modify class attributes/methods

- Implement design patterns (Singleton, Factory, etc.)
- Register classes automatically
- Add logging or validation
- Modify class creation behavior

Important Considerations

1. Class decorators are applied after the class is created but before any instances are made.
2. The order of multiple decorators matters - they're applied from bottom to top.
3. Be careful with inheritance when using class decorators.

Class decorators provide a clean, Pythonic way to modify class behavior without using metaclasses, which are more complex.