Object Oriented Programming

OOPs

Table of Contents

OOPs (Object Oriented Programming)

Object-Oriented Programming is a programming paradigm that revolves around the concept of objects, which are instances of classes. It allows you to model real-world entities as objects and define their behaviour through methods and attributes. OOP promotes modularity, reusability, and maintainability of code.

How OOPs is better?

1. **Modularity**: OOP promotes modular design, allowing you to break complex problems into manageable parts (objects) with distinct responsibilities.
2. **Reusability**: With features like inheritance, you can create new classes by extending existing ones, reducing redundant code and improving code reuse.
3. **Encapsulation**: Encapsulation hides the internal details of objects, preventing unintended interference and making code more robust and secure.
4. **Abstraction**: OOP allows you to model real-world concepts abstractly, focusing on what objects do rather than how they do it.
5. **Flexibility**: Polymorphism enables you to use different objects interchangeably, fostering adaptable and flexible code.
6. **Scalability**: OOP promotes scalability as your codebase grows. You can extend existing classes or create new ones without disrupting the existing code.
7. **Maintenance**: Changes in one part of the codebase have limited impact on other parts, leading to easier maintenance and updates.
8. **Collaboration**: Teams can work concurrently on different classes or modules without interfering with each other's work.
9. **Real-World Modelling**: OOP models real-world entities and their relationships, making the codebase more intuitive and closely mirroring the problem domain.
10. **Code Understandability**: Well-designed OOP code with clear class hierarchies and meaningful names enhances code readability and understandability.

Overall, OOP provides a structured approach that improves code organization, reusability, maintainability, and collaboration among developers.

Main features of OOPs

1. Encapsulation: Bundling data (attributes) and methods that operate on the data into a single unit (object) while hiding internal details.
   Eg: In a smartphone, there are various components such as the processor, memory, camera, and battery. These components are encapsulated within the device's outer shell, which serves as a protective barrier. Users interact with the smartphone through a limited set of well-defined interfaces, such as the touchscreen, buttons, and ports.

2. Abstraction: Abstraction is a OOPs concept to build the structure of real-world objects. It “shows” only essential attributes and “hides” unnecessary information from the outside.
   Eg: Abstraction is like using a TV remote without knowing its inner workings; you interact with its buttons (interface) to control the TV, without needing to understand the technical details inside.

3. Inheritance: Creating new classes by inheriting attributes and methods from existing ones, promoting code reuse and hierarchy.
   Eg: Think of a vehicle hierarchy: all vehicles share common traits like having wheels and engines. Inheritance is like having a "Vehicle" class as the base, and then creating subclasses like "Car," "Bike," and "Truck," inheriting the basic attributes from the "Vehicle" class while adding specific features for each type of vehicle.

4. Polymorphism: Treating different objects through a common interface, allowing flexibility and dynamic behaviour in code.
   Eg: Imagine a music player: different types of devices like phones, tablets, and laptops can all play music. Polymorphism allows you to control the music playback using the same play, pause, and stop buttons on these devices, even though the underlying mechanisms are different, making it easy to interact with different objects in a unified way.

Basic Terminologies:-

1. **Class:** A class is a building block of Object Oriented Programs. It is a user-defined data type that contains the data members and member functions that operate on the data members. It is like a blueprint or template of objects having common properties and methods.

2. **Object**: An object refers to the instance of the class, which contains the instance of the members and behaviors defined in the class template. In the real world, an object is an actual entity to which a user interacts, whereas class is just the blueprint for that object.

```
# Class keyword is used to create a class
class Car:
    def __init__(self, make, model, year):
        #self is used to access the attributes and methods of the class
        self.make = make
        self.model = model    #atributes
        self.year = year

    def start_engine(self):   #methods
        return f"{self.make} {self.model}'s engine started."

# Creating objects (instances) of the Car class
car1 = Car("Toyota", "Camry", 2021)
car2 = Car("Tesla", "Model S", 2022)
```

3. **Attributes and Methods**: Attributes are variables that hold data specific to each object, while methods are functions that define the behaviour of the object.

4. **Constructor:** A constructor is a block of code that initializes the newly created object. A constructor resembles an instance method but it's not a method as it doesn't have a return type.
   The __init__ method is a special method in Python classes that gets called automatically when an object is created from the class. It is used to initialize the object's attributes with the values provided during object creation.

* *Copy Constructor is a type of constructor, whose purpose is to copy an object to another. What it means is that a copy constructor will clone an object and its values, into another object, is provided that both the objects are of the same class. Python don't have any copy constructor, though java and C++ have.

```python
# Example: Default Arguments in the Car class constructor
#__init__ is constructor
class Car:
    def __init__(self, make="Unknown", model="Unknown", year=2020):
        self.make = make
        self.model = model
        self.year = year

car1 = Car("Toyota", "Camry", 2021)
car2 = Car()  # Using default values for attributes
```

5. **Destructors:** Destructors are also special methods. But destructors free up the resources and memory occupied by an object. Destructors are automatically called when an object is being destroyed.
   Class destructors are useful for performing clean-up tasks, such as releasing resources like file handles or network connections, before an object is removed from memory.

```python
# Example: Using Class destructors(__del__) for resource release in the Car class
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.file_handle = open(f"{make}_{model}_data.txt", "w")

    def __del__(self):
        self.file_handle.close()
        print(f"{self.make} {self.model} object is being destroyed.")

car = Car("Toyota", "Camry")
# Assume other operations with the car object and file handling.
del car  # File handle will be closed, and the object is destroyed.
```

Encapsulation

Encapsulation refers to binding the data and the code that works on that together in a single unit. Class binds variables and methods to perform some task, hence implement encapsulation.

Access specifiers or **access modifiers** are keywords that determine the accessibility of methods, classes, etc in OOPs. These access specifiers allow the implementation of encapsulation. Three type of access modifiers are:-

1. Public: All the class members declared under the public specifier will be available to everyone.
2. Private: The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions(not available in python) are allowed to access the private data members of the class.
3. Protected: The protected access modifier is similar to the private access modifier in the sense that it can't be accessed outside of its class, however they can be accessed by any subclass (derived class) of that class.

Name	Accessibility from own class	Accessibility from derived class	Accessibility from world
Public	Yes	Yes	Yes
Private	Yes	No	No
Protected	Yes	Yes	No

Access Modifiers in Python

Python uses access modifiers to control the visibility of attributes and methods within a class. The three access modifiers are:

- Public: No underscore before the attribute/method name. Accessible from anywhere.
- Private: Double underscore prefix before the attribute/method name. Accessible only from within the class.
- Protected: Single underscore prefix before the attribute/method name. Accessible within the class and its subclasses.

The default access modifier for class members (attributes and methods) is public.

```
class Employee:
    def __init__(self, name, salary):
        self.name = name                # Public attribute
        self._salary = salary           # Protected attribute
        self.__bonus = 1000             # Private attribute

    def get_bonus(self):
        return self.__bonus

    def _calculate_total_salary(self):
        return self._salary + self.__bonus

employee = Employee("Alice", 50000)

# Accessing public and protected attributes directly
print(employee.name)       # Output: "Alice"
print(employee._salary)    # Output: 50000

# Attempting to access the private '__bonus' attribute directly
# This will raise an error: AttributeError: 'Employee' object has no attribute '__bonus'
print(employee.__bonus)   # Output: Attribute error

# Accessing the private '__bonus' attribute using the getter method
print(employee.get_bonus())  # Output: 1000

# Accessing the protected attribute and private attribute through a method
print(employee._calculate_total_salary())  # Output: 51000
```

Private methods: can not be accessed directly, but public methods of same function can call this.

```
class MyClass:
    def __init__(self):
        self.__private_var = 42

    def public_method(self):
        print("This is a public method.")
        self.__private_method()  # A public method can call a private method.

    def __private_method(self):
        print("This is a private method.")

# Create an instance of the class
obj = MyClass()

# Accessing a public method, which in turn calls the private method
obj.public_method()

# Attempting to access the private method directly (it's possible but discouraged)
obj.__private_method()      #raise an attribute error
```

Abstraction

Abstraction is a fundamental concept used to simplify complex systems by focusing on the essential details while hiding the unnecessary complexities. Abstraction allows us to build software that is easier to understand, maintain, and scale. At its core, abstraction involves:

1. **Hiding Implementation Details:** Abstraction allows us to hide the complex inner workings of a system or an object. This is crucial because it reduces complexity and allows us to work with high-level concepts.
2. **Exposing Only Necessary Information**: Abstraction exposes only the relevant features, properties, and behaviors of an object, making it easier for developers to interact with it.

Ways to achieve abstraction in python

- Use abstract classes to define a blueprint for group of related classes.
- Mark attributes and methods as private by prefixing their names with a double underscore __. This indicates they are intended for internal use within the class.
- Use getter and setter methods to provide controlled access to private attributes.
- Leverage Python modules and libraries to abstract complex functionality.

Eg1: **Getter and Setter Methods:** To provide controlled access to private attributes, you can use getter and setter methods. Getter methods retrieve the value of private attributes, and setter methods modify their values.

```
class MyClass:
    def __init__(self):
        self.__private_var = 42

    def get_private_var(self):
        return self.__private_var

    def set_private_var(self, value):
        if value > 0:
            self.__private_var = value
```

Eg2: **Modules and Libraries**: Python's standard library and third-party libraries often provide abstracted interfaces or classes that hide the underlying complexity of various operations.

For example, the math module provides a range of mathematical functions and constants, abstracting the low-level implementation details.

Abstract Classes

An **abstract class** in Python is a class that cannot be instantiated, meaning you cannot create objects directly from it. Instead, abstract classes serve as blueprints for other classes. They typically contain one or more **abstract methods** that must be implemented by subclasses.

Abstract classes and methods help ensure that specific behaviors are implemented consistently across different classes in your program. Other languages use interface for same functionality (Interfaces specify the method signatures (names, parameters, return types) that classes must provide implementations for).

Defining Abstract Classes

To define an abstract class in Python, you can use the abc module (Abstract Base Classes). Here's how you define an abstract class:

```
from abc import ABC, abstractmethod

class MyAbstractClass(ABC):
    @abstractmethod
    def my_abstract_method(self):
        pass
```

In this example, MyAbstractClass is an abstract class, and my_abstract_method is an abstract method that must be implemented by any subclass.

Subclassing and Implementing Abstract Methods

When you create a subclass of an abstract class, you must provide implementations for all its abstract methods. Failure to do so will result in a TypeError.

```
class MyConcreteClass(MyAbstractClass):
    def my_abstract_method(self):
        return "Implemented abstract method in MyConcreteClass"
```

Here, MyConcreteClass is a concrete class that inherits from MyAbstractClass and provides an implementation for my_abstract_method.

Inheritance

Inheritance allows one class to inherit properties (attributes and methods) from another class. The class that inherits is called the subclass or derived class, and the class from which it inherits is known as the superclass or base class.

```python
#Base class
class Animal:
    def __init__(self, species):
        self.species = species

    def make_sound(self):
        return "Some generic sound"

    def get_species(self):
        print("My species is",self.species)

#Subclass
class Dog(Animal):
    def __init__(self, breed):
        super().__init__("Dog")
        self.breed = breed

    def make_sound(self):
        return "Woof!"

# Creating objects of the classes
animal_obj = Animal("Unknown")
dog_obj = Dog("Labrador")

print(animal_obj.species)  # Output: "Unknown"
print(dog_obj.species)     # Output: "Dog"
print(dog_obj.make_sound())  # Output: "Woof!"

#subclass obj can use methods of base class too
print(dog_obj.get_species()) # My species is Dog
```

In the above example, the Dog class inherits from the Animal class. The Dog class has its own attribute breed and overrides the make_sound method with its implementation.

Super() keyword

To access the superclass methods inside the subclass, you can use the super() function. It allows you to call the superclass methods and access their attributes.

```python
# Example: Using super() to call the superclass method
class Animal:
    def __init__(self, species):
        self.species = species

    def make_sound(self):
        return "Some generic sound"

class Dog(Animal):
    def __init__(self, breed):
        super().__init__("Dog")
        self.breed = breed

    def make_sound(self):
        generic_sound = super().make_sound()
        return f"{generic_sound} but also Woof!"

dog_obj = Dog("Labrador")
print(dog_obj.make_sound())  # Output: "Some generic sound but also Woof!"
```

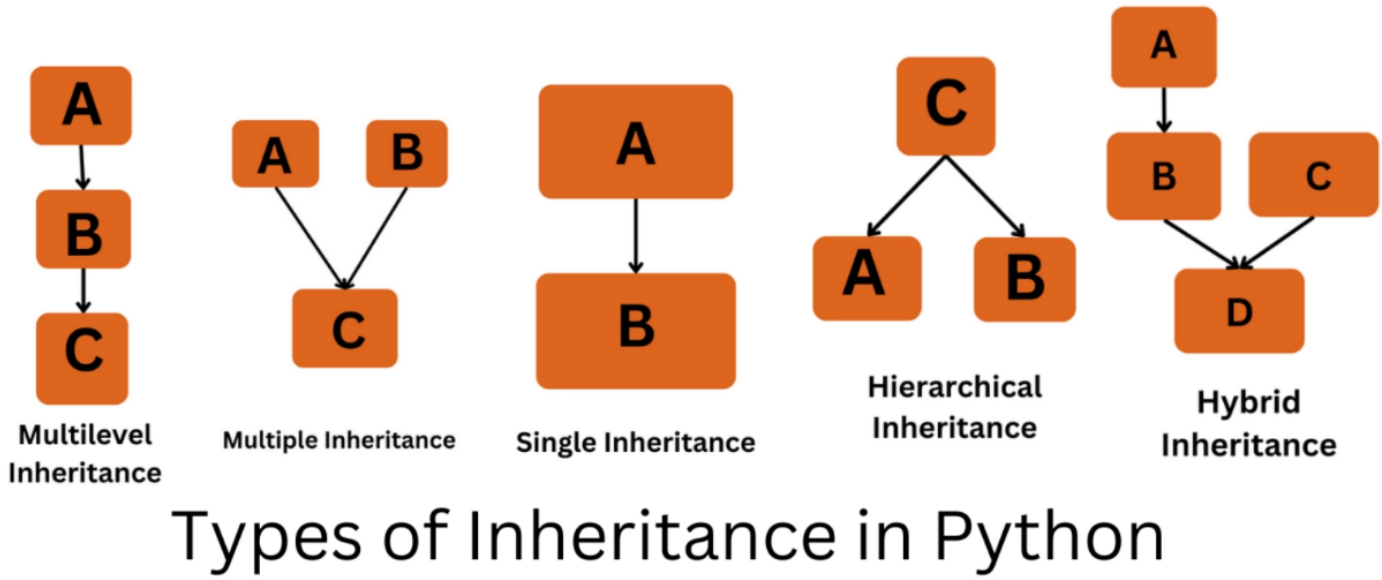
Method Overriding

Method overriding occurs when a subclass provides its implementation for a method that is already defined in the superclass. The subclass method with the same name as the superclass method will override the superclass method.

```python
# Example: Method Overriding in the Car class
class Vehicle:
    def start(self):
        return "Vehicle starting..."

class Car(Vehicle):
    def start(self):
        return "Car starting..."


vehicle_obj = Vehicle()
car_obj = Car()

print(vehicle_obj.start())  # Output: "Vehicle starting..."
print(car_obj.start())      # Output: "Car starting..."
```

Car class updates the functionality of start method, hence overrides it.

Types of inheritance



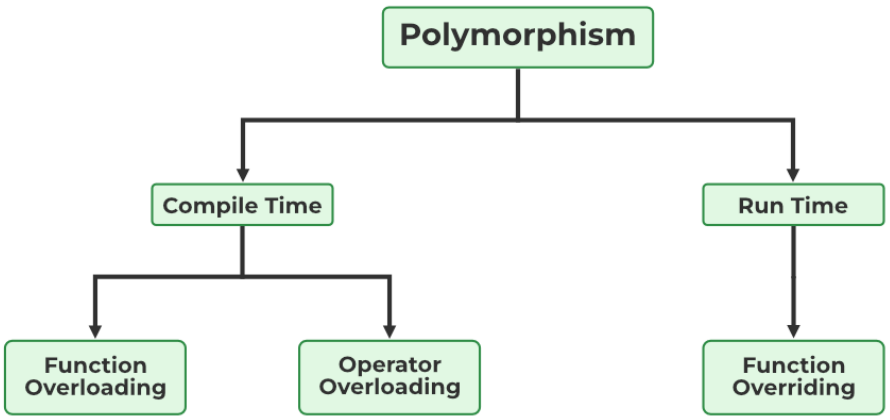
To implement multiple inheritance

```
# multiple inheritance

# Base class1
class Mother:
    mothername = ""

    def mother(self):
        print(self.mothername)

# Base class2
class Father:
    fathername = ""

    def father(self):
        print(self.fathername)

# Derived class
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)


# Driver's code
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

Polymorphism

Polymorphism refers to the process by which some code, data, method, or object behaves differently under different circumstances or contexts. Compile-time polymorphism and Run time polymorphism are the two types of polymorphisms in OOPs languages.

- **Compile-Time Polymorphism**: Compile time polymorphism, also known as static polymorphism or early binding is the type of polymorphism where the binding of the call to its code is done at the compile time. Method overloading or operator overloading are examples of compile-time polymorphism.
- **Runtime Polymorphism**: Also known as dynamic polymorphism or late binding, runtime polymorphism is the type of polymorphism where the actual implementation of the function is determined during the runtime or execution. Method overriding is an example of this method.



Method overloading

Method overloading allows you to define multiple methods with the same name in a class, but with different parameters.

Python does not natively support method overloading as some other languages do (e.g., Java or C++).

But you can achieve similar functionality using default arguments or variable-length argument lists. The method called depends on the number and types of arguments provided when you invoke it

```python
class Calculator:
    def add(self, a, b=None):
        if b is None:
            return a
        else:
            return a + b

calc = Calculator()
result1 = calc.add(5)
result2 = calc.add(2, 3)

print(result1)  # Output: 5
print(result2)  # Output: 5
```

Cant do this.

```python
class Calculator:
    def add(self, a, b=None):
        return a+b

    def add(self,a,b,c):    #gets type error when tried to use
        return a+b+c
```

Operator overloading

Operator overloading in Python allows you to define custom behavior for standard operators when applied to objects of your class. This is achieved by defining special methods with double underscores (e.g., __add__() for addition) in your class. These special methods are also known as "**magic methods**" or "**dunder methods**" (short for "double underscore").

Special methods are enclosed in double underscores (__) at the beginning and end of their names. They provide a way to define specific behaviors for objects, making Python classes more powerful and flexible

1. **The __str__ method** is used to provide a human-readable string representation of an object. It is called when you use the str() function or the print statement with the object.
2. **The __add__ method** allows you to define addition between objects of your class. It is called when you use the + operator with the objects.
3. **The __eq__ Method** enables you to define the equality comparison between objects of your class. It is called when you use the == operator with the objects.

```python
# Example: Special Methods in Python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"Point({self.x}, {self.y})"

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

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

point1 = Point(1, 2)
point2 = Point(3, 4)

print(point1)               # Output: "Point(1, 2)"
print(point1 + point2)      # Output: "Point(4, 6)"
print(point1 == point2)     # Output: False
```

Eg2: Use of sub and mul methods

```python
# Example: Other Common Special Methods
class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return ComplexNumber(self.real + other.real, self.imag + other.imag)

    def __sub__(self, other):
        return ComplexNumber(self.real - other.real, self.imag - other.imag)

    def __mul__(self, other):
        return ComplexNumber(self.real * other.real - self.imag * other.imag,
                             self.real * other.imag + self.imag * other.real)

    def __str__(self):
        return f"{self.real} + {self.imag}i"

num1 = ComplexNumber(2, 3)
num2 = ComplexNumber(1, 4)

print(num1 + num2)   # Output: "3 + 7i"
print(num1 - num2)   # Output: "1 - 1i"
print(num1 * num2)   # Output: "-10 + 11i"
```

Advance OOPs concept

Method Resolution Order (MRO's)

When a class inherits from multiple base classes, Python uses a method resolution order (MRO) to determine the order in which the base classes' methods are called.

```
# Example: Method Resolution Order (MRO) in Multiple Inheritance

class A:
    def greet(self):
        return "Hello from A."

class B(A):
    def greet(self):
        return "Hello from B."

class C(A):
    def greet(self):
        return "Hello from C."

class D(B, C):
    pass

d = D()
print(d.greet())  # Output: "Hello from B."
```

In this example, the D class inherits from both B and C, and since B is listed first in the inheritance chain, its method is called when we call the greet method on an object of the D class.

Mixins

Mixins are a way to share functionalities among classes without using multiple inheritance. A mixin is a class that is not intended to be instantiated but is designed to be mixed into other classes, enhancing their functionalities.

In below example, we define two mixins (JSONMixin and XMLMixin) that provide functionalities to convert an object's attributes to JSON and XML formats, respectively. The Employee class inherits from the Person class and mixes in the functionalities of both mixins.

```python
# Example: Using Mixins in Python
class JSONMixin:
    def to_json(self):
        import json
        return json.dumps(self.__dict__)

class XMLMixin:
    def to_xml(self):
        xml_str = f"<{self.__class__.__name__}>"
        for key, value in self.__dict__.items():
            xml_str += f"<{key}>{value}</{key}>"
        xml_str += f"</{self.__class__.__name__}>"
        return xml_str

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

class Employee(Person, JSONMixin, XMLMixin):
    def __init__(self, name, age, emp_id):
        super().__init__(name, age)
        self.emp_id = emp_id

employee = Employee("John", 30, "EMP123")

print(employee.to_json())
# Output: '{"name": "John", "age": 30, "emp_id": "EMP123"}'

print(employee.to_xml())
# Output: '<Employee><name>John</name><age>30</age><emp_id>EMP123</emp_id></Employee>'
```

Interview Questions

Q1. How much memory does a class occupy?

Classes do not consume any memory. They are just a blueprint based on which objects are created. Now when objects are created, they actually initialize the class members and methods and therefore consume memory.