Fundamentals of Python

PYTHON OOPS

Shruti Bharat

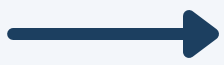
@shrutibharat0105

Table of Contents

Class and Object

- Everything in Python is an object.
- A **class** is like a **blueprint** for creating objects. An **object** is an **instance** of class.
- It defines a set of **attributes (variables)** and **methods (functions)** that describe the behavior and properties of objects created from it.
- Classes encapsulate data for the object and provide methods to manipulate that data.

```python
# Class definition
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        return f"Hello, my name is {self.name} and I am {self.age} years old."

# Creating an object of class Person
person1 = Person("Alice", 30)

# Accessing attributes and calling methods
print(person1.greet())  # Output: Hello, my name is Alice and I am 30 years old.
```



Class and Object

1. **Class Definition:**

- Person: A class with attributes name and age.

2. **Method:**

- greet(): Returns a greeting message including the person's name and age.

3. **Object Creation:**

- person1: An object of class Person representing a person named Alice who is 30 years old.

4. **Using the Object:**

- person1.greet(): Returns "Hello, my name is Alice and I am 30 years old."

Method Vs Function

Method:

- A method is a function that is defined inside a class and is associated with the objects of that class.
- Methods are called on objects and can access and modify the object's attributes.

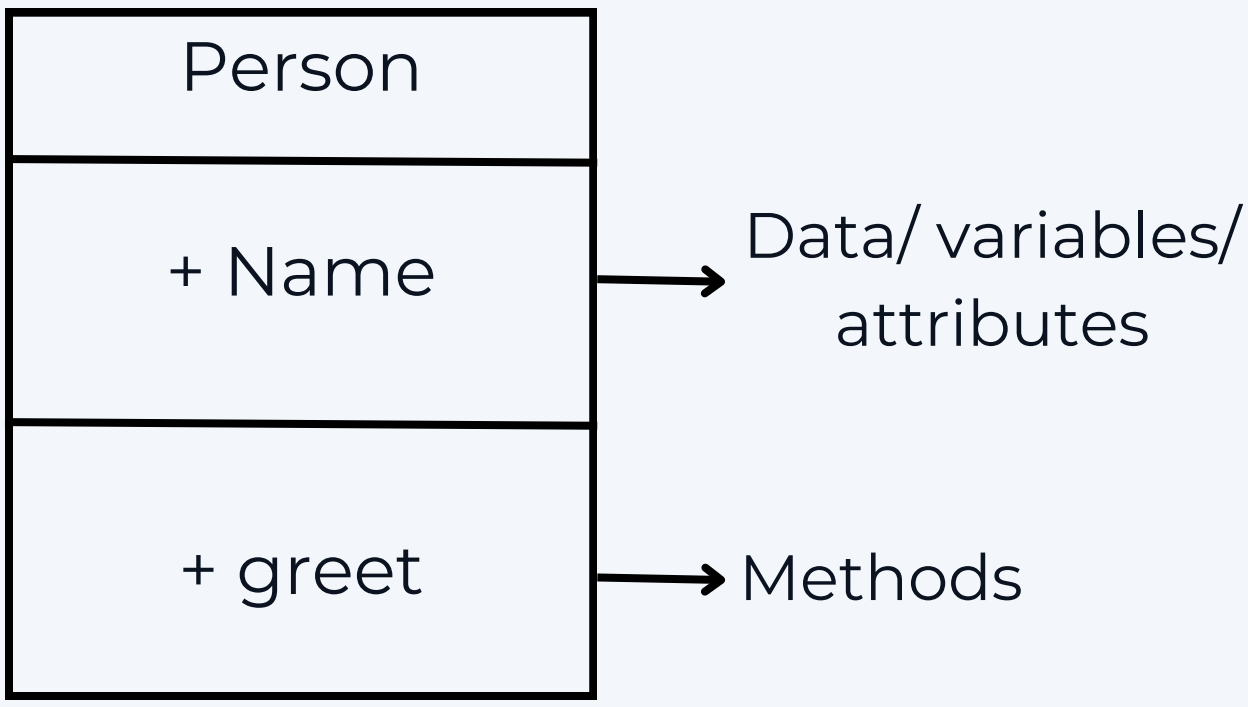
Function:

- In Python, a function is a block of reusable code that performs a specific task.
- It is defined using the def keyword and can be called from anywhere in the code.

Method Vs Function

```python
# Function definition
def greet():
    return "Hello from a function!"

# Class definition
class Person:
    def __init__(self, name):
        self.name = name

    # Method definition
    def greet(self):
        return f"Hello from {self.name}!"

# Calling the function
print(greet())  # Output: Hello from a function!

# Creating an object of class Person and calling the method
person1 = Person("Shruti")
print(person1.greet())  # Output: Hello from Shruti!
```

Class diagram



Ignore the plus sign for now, as we will learn about it going forward. For now, just understand that it indicates public variables or methods. The minus sign indicates private variables or methods.

Magic(Dundar) methods & Constructor

- These methods allow you to define the behavior of objects for built-in operations and functions.
- They are used to implement operator overloading, initialize objects, and provide a way to customize the behavior of your classes

Constructor

- A constructor is a special method that is automatically called when an object is instantiated.
- Unlike other methods, the constructor is not controlled by the user and is invoked as soon as an object is created.

Magic(Dundar) methods & Constructor

What’s the use of a constructor:

- This makes it ideal for writing configuration code that should run immediately upon object creation, ensuring the object is set up correctly before any other methods are used.

Key Points:

- **Magic Methods:** Special methods with double underscores at the beginning and end of their names, enabling customization of object behavior.
- **Constructor (__init__):** Automatically called to initialize an object, making it perfect for configuration code that should not be controlled by the user.

Self keyword

- The self keyword in Python is used within a class to refer to the instance of the class itself.
- It allows access to the attributes and methods of the class in oop.
- Self is the default argument in every method.

An example of person class (Page 1)

- self is used in the __init__ method to assign the name and age attributes to the Person instance.
- self is also used in the greet method to access these attributes and return a greeting message.
- It is similar to this keyword in Java.

Reference variable

- The reference variable holds the objects.
- We can create objects without reference variables as well.
- An object can have multiple reference variables.
- Assigning a new reference variable to an existing object does not create a new object.

```python
# Class definition
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Creating an object of class Person without variable
Person("Shruti", 30)

p = Person("Shruti", 30)
# p is just variable name who has object address.
# p is not a object. It is reference variable.

q = p # Now p & q is pointing to same object.
```

Reference variable

Pass by reference:

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

def greet_person(person):
    # Create a new Person object
    new_person = Person("Shruti", 25)
    return new_person

# Create a Person object
person1 = Person("Shreya", 30)

# Pass the Person object to the function
another_person = greet_person(person1)

# Print the new Person object
print(another_person.name, another_person.age)  # Output: Shruti 25

# id(person1) = id(person) it is same as both are pointing to same object.
# We passed object reference ie. address of objects to function.
# Objects in python are mutable by default like list, set, dictionary.
```

Encapsulation

- Encapsulation in Python is the principle of bundling the data (attributes) and methods (functions) that operate on the data within a single class and controlling access to that class's components.
- It restricts direct access to some of the object's components and provides methods to manipulate the object's state, ensuring data integrity and preventing accidental modification.

Getter & setter methods

Private Keyword, Getter, and Setter:

- **Private Keyword (__):** In Python, attributes and methods can be marked as private by prefixing their names with double underscores (__). This makes them inaccessible from outside the class.

- **Getter Method:** A method used to access the value of a private attribute. It typically has the prefix get_ followed by the attribute name.

- **Setter Method:** A method used to set the value of a private attribute. It typically has the prefix set_ followed by the attribute name.

Getter & setter methods

```python
class BankAccount:
    def __init__(self, account_number, initial_balance):
        self.__account_number = account_number  # Private attribute
        self.__balance = initial_balance        # Private attribute

    # Getter method for account number
    def get_account_number(self):
        return self.__account_number

    # Getter method for balance
    def get_balance(self):
        return self.__balance

    # Method to deposit money
    def deposit(self, amount):
        self.__balance += amount

    # Method to withdraw money
    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print("Insufficient funds.")

# Creating an instance of BankAccount
account = BankAccount("123456789", 1000)

# Accessing account balance via getter method
print(account.get_balance())  # Output: 1000

# Depositing money
account.deposit(500)
print(account.get_balance())  # Output: 1500

# Withdrawing money
account.withdraw(200)
print(account.get_balance())  # Output: 1300
```

Static vs Instance Variables

Static Variables:

- Defined at the class level.
- Shared among all instances of the class.
- Incremented each time a new instance is created.

Instance Variables:

- Belong to a specific instance of a class.
- Each object has its own copy of these variables.

Static vs Instance Variables

```python
class Car:
    # Static variable
    total_cars = 0

    def __init__(self, make, model):
        # Instance variables
        self.make = make
        self.model = model
        Car.total_cars += 1  # Increment the static variable

    def display_details(self):
        return f"Car make: {self.make}, model: {self.model}"

    @classmethod
    def get_total_cars(cls):
        return f"Total cars: {cls.total_cars}"

# Creating instances of Car
car1 = Car("Toyota", "Camry")
car2 = Car("Honda", "Accord")

# Accessing instance variables
print(car1.display_details())  # Output: Car make: Toyota, model: Camry
print(car2.display_details())  # Output: Car make: Honda, model: Accord

# Accessing static variable
print(Car.get_total_cars())    # Output: Total cars: 2
```

Aggregation

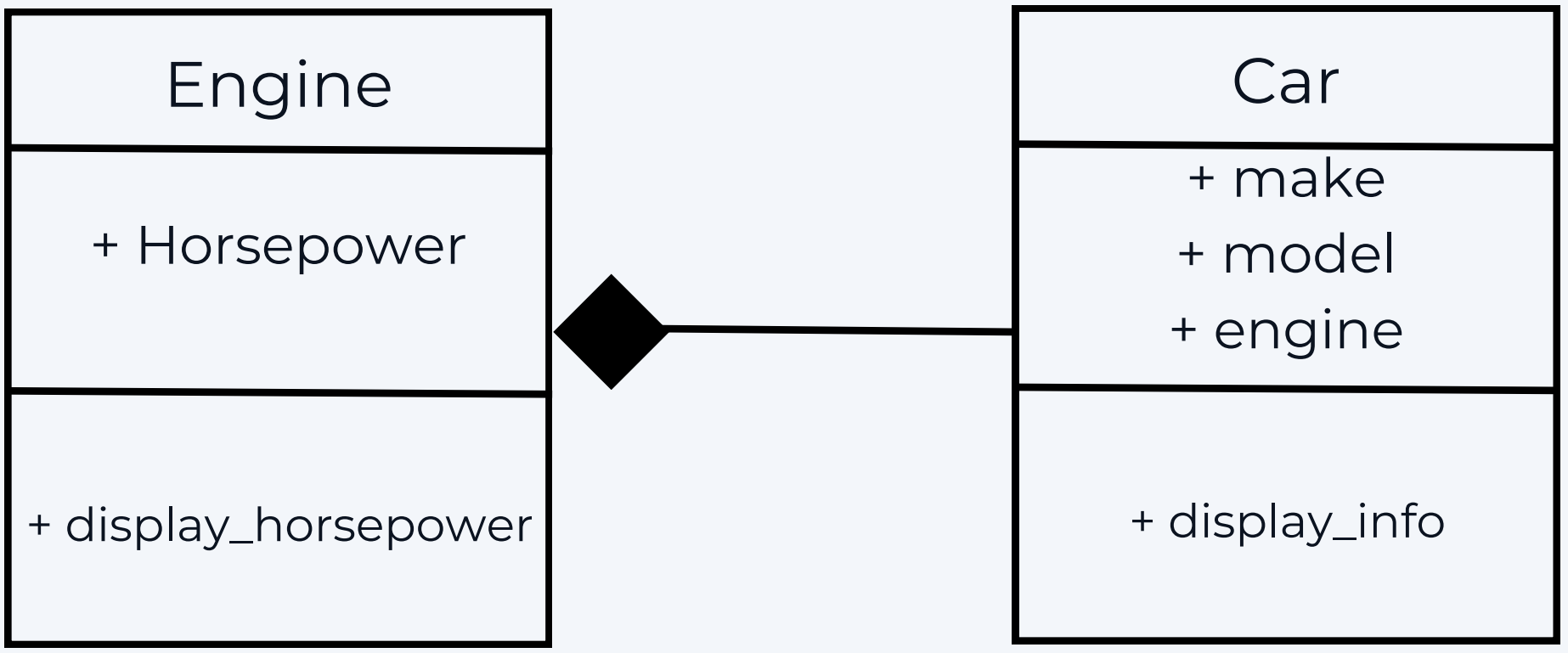
- A design principle where one class (the container class) contains objects of another class (the contained class).
- It represents a "has-a" relationship.

```python
class Engine:
    def __init__(self, horsepower):
        self.horsepower = horsepower

    def display_horsepower(self):
        return f"Horsepower: {self.horsepower}"

class Car:
    def __init__(self, make, model, engine):
        self.make = make
        self.model = model
        self.engine = engine  # Car "has-a" Engine

    def display_info(self):
        return f"Car: {self.make} {self.model}, {self.engine.display_horsepower()}"

# Creating an Engine object
engine = Engine(300)

# Creating a Car object with the Engine object
car = Car("Toyota", "Camry", engine)
print(car.display_info())  # Output: Car: Toyota Camry, Horsepower: 300
```

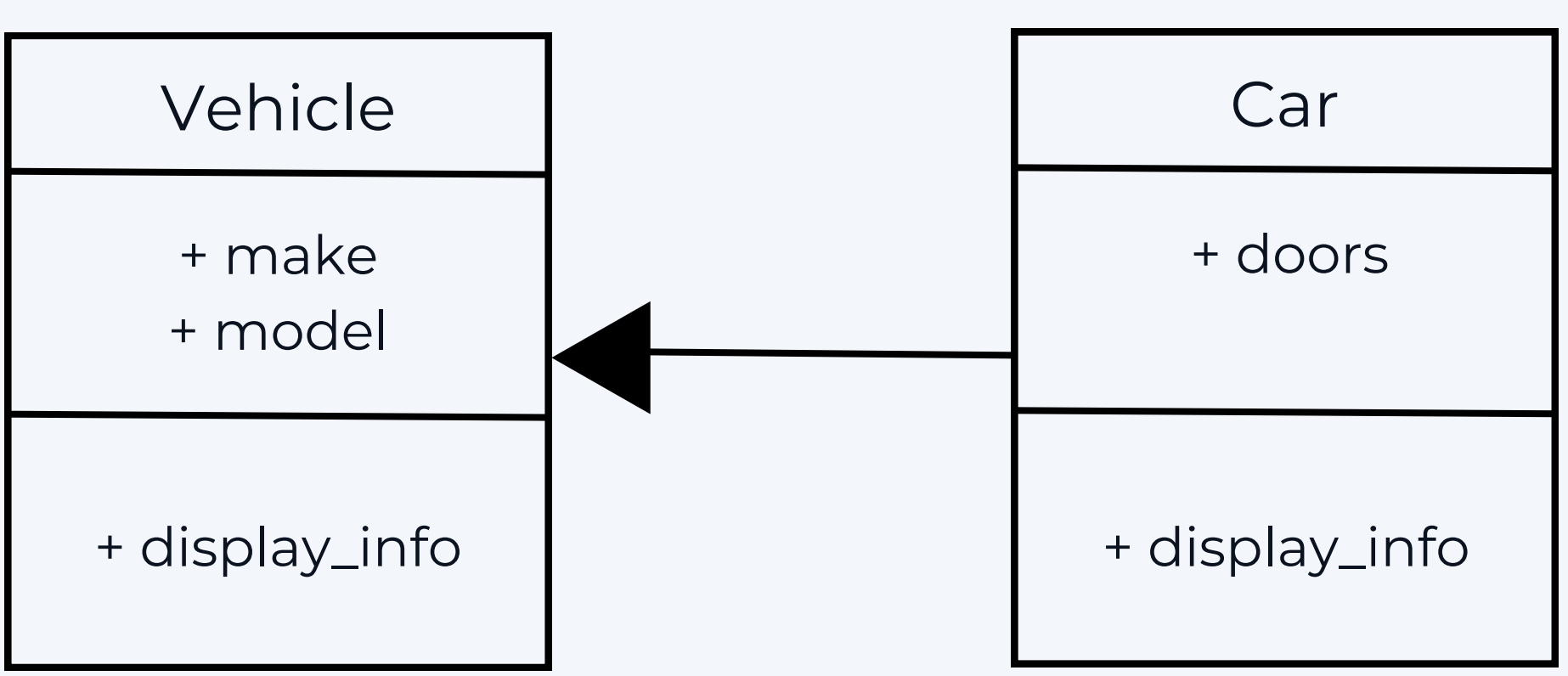
- The Car class contains an instance of the Engine class.
- This represents a "has-a" relationship where a Car has an Engine.
- The Car class uses the Engine class's display_horsepower method to include engine information in its display_info method.

Class diagram

Aggregation



Inheritance



Inheritance

- A mechanism where a new class (derived class) inherits attributes and methods from an existing class (base class).
- It represents an "is-a" relationship.

```python
# Base class
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def display_info(self):
        return f"Vehicle: {self.make} {self.model}"

# Derived class
class Car(Vehicle):
    def __init__(self, make, model, doors):
        super().__init__(make, model)
        self.doors = doors

    def display_info(self):
        return f"Car: {self.make} {self.model}, Doors: {self.doors}"


# Creating an object of Car
car = Car("Toyota", "Camry", 4)
print(car.display_info())  # Output: Car: Toyota Camry, Doors: 4
```

- The Car class inherits from the Vehicle class.
- The Car class can access the attributes and methods of the Vehicle class.
- The display_info method is overridden in the Car class to include additional information about doors.

Types of inheritance

1. **Single Inheritance**
    - In single inheritance, a class inherits from one base class.

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

    def display(self):
        print(f"Parent name: {self.name}")

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

    def display(self):
        print(f"Child name: {self.name}, Age: {self.age}")

# Example usage
child = Child("John", 10)
child.display()  # Output: Child name: John, Age: 10
```

Types of inheritance

2. **Multiple Inheritance**
   - In multiple inheritance, a class inherits from more than one base class.

```python
class Father:
    def __init__(self, father_name):
        self.father_name = father_name

class Mother:
    def __init__(self, mother_name):
        self.mother_name = mother_name

class Child(Father, Mother):
    def __init__(self, father_name, mother_name, child_name):
        Father.__init__(self, father_name)
        Mother.__init__(self, mother_name)
        self.child_name = child_name

    def display(self):
        print(f"Child name: {self.child_name}, Father: {self.father_name}, Mother: {self.mother_name}")

# Example usage
child = Child("Mark", "Sara", "John")
child.display()  # Output: Child name: John, Father: Mark, Mother: Sara
```

Diamond Problem:

- The diamond problem occurs in multiple inheritance when a class inherits from two classes that both inherit from a single common base class.
- This can lead to ambiguity about which version of a method to inherit from the common base class.
- The Method Resolution Order (MRO) determines the order in which base classes are looked up when searching for a method.

Types of inheritance

Diamond Problem:

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

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

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

class D(B, C):
    pass

# Creating instance of class D
d = D()
d.greet()
```

Python follows the MRO, which in this case is:
D → B → C → A
Thus, it uses the greet method from B.

Types of inheritance

3. **Multilevel inheritance**
   - In multilevel inheritance, a class inherits from a derived class, forming a chain of inheritance.

```python
class Grandparent:
    def __init__(self, grandparent_name):
        self.grandparent_name = grandparent_name

class Parent(Grandparent):
    def __init__(self, grandparent_name, parent_name):
        super().__init__(grandparent_name)
        self.parent_name = parent_name

class Child(Parent):
    def __init__(self, grandparent_name, parent_name, child_name):
        super().__init__(grandparent_name, parent_name)
        self.child_name = child_name

    def display(self):
        print(f"Child name:{self.child_name},Parent:{self.parent_name},Grandparent:{self.grandparent_name}")

# Example usage
child = Child("Edward", "Mark", "John")
child.display()  # Output: Child name: John, Parent: Mark, Grandparent: Edward
```

The **super keyword** in Python is used to call a method from the parent class. This is especially useful for extending the functionality of inherited methods.

Types of inheritance

4. **Hierarchical Inheritance**
   - In hierarchical inheritance, multiple derived classes inherit from a single base class.

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

    def display(self):
        print(f"Parent name: {self.name}")

class Child1(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

    def display(self):
        print(f"Child1 name: {self.name}, Age: {self.age}")

class Child2(Parent):
    def __init__(self, name, grade):
        super().__init__(name)
        self.grade = grade

    def display(self):
        print(f"Child2 name: {self.name}, Grade: {self.grade}")

# Example usage
child1 = Child1("John", 10)
child2 = Child2("Anna", "A")
child1.display()  # Output: Child1 name: John, Age: 10
child2.display()  # Output: Child2 name: Anna, Grade: A
```

Types of inheritance

5. Hybrid Inheritance

- Hybrid inheritance is a combination of two or more types of inheritance.

```
class Base:
    def __init__(self, base_name):
        self.base_name = base_name

class Derived1(Base):
    def __init__(self, base_name, derived1_name):
        super().__init__(base_name)
        self.derived1_name = derived1_name

class Derived2(Base):
    def __init__(self, base_name, derived2_name):
        super().__init__(base_name)
        self.derived2_name = derived2_name

class Derived3(Derived1, Derived2):
    def __init__(self, base_name, derived1_name, derived2_name, derived3_name):
        Derived1.__init__(self, base_name, derived1_name)
        Derived2.__init__(self, base_name, derived2_name)
        self.derived3_name = derived3_name

    def display(self):
        print(f"Derived3 name: {self.derived3_name}, Base: {self.base_name},
        Derived1: {self.derived1_name}, Derived2: {self.derived2_name}")

# Example usage
derived3 = Derived3("BaseClass", "Derived1Class", "Derived2Class", "Derived3Class")
derived3.display()  # Output: Derived3 name: Derived3Class, Base: BaseClass,
Derived1: Derived1Class, Derived2: Derived2Class
```

Types of inheritance

Inheritance Summary:

- **Reusability:** Inheritance allows classes to reuse methods and attributes from parent classes.
- **Hierarchical Classification:** Supports creating a natural hierarchy through base and derived classes.
- **Overriding:** Derived classes can provide specific implementations for base class methods.
- **Extensibility:** New functionality can be added to existing classes through derived classes.
- **Transitivity:** Inherited traits are passed down the inheritance chain.
- **Accessibility:** Public and protected members of base classes are accessible to derived classes.
- **Multiple Inheritance:** A class can inherit from more than one base class.
- **Single Root Hierarchy:** All classes ultimately inherit from a single root class, usually Object.

Polymorphism

It refers to the ability of different objects to respond to the same method calls, typically resulting in different behavior based on their individual implementations.

Types of polymorphism:

- **Compile-Time Polymorphism (Method Overloading):** This isn't natively supported in Python but can be simulated using default arguments or variable-length arguments.

- **Run-Time Polymorphism (Method Overriding):** This is supported in Python through inheritance and overriding methods in derived classes.

- **Operator overloading:** Having different functionality of the operator based on the operands. eg. ‘+’ operator

Polymorphism

```python
# Run-Time Polymorphism (Method Overriding)
class Animal:
    def sound(self):
        raise NotImplementedError("Subclass must implement abstract method")

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

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

# Demonstrating Run-Time Polymorphism
def make_sound(animal):
    return animal.sound()

# Creating instances
dog = Dog()
cat = Cat()


# Compile-Time Polymorphism (Simulated Method Overloading)
class MathOperations:
    def add(self, a, b, c=0):
        return a + b + c

# Demonstrating Compile-Time Polymorphism
math_op = MathOperations()

# Output
print(make_sound(dog))          # Output: Woof!
print(make_sound(cat))          # Output: Meow!
print(math_op.add(1, 2))        # Output: 3
print(math_op.add(1, 2, 3))     # Output: 6

# Operator overloading
print('Hello'+'World')      # Output: HelloWorld
print(3 + 4)                # Output: 7
print([1,2,3] + [4,5])      # Output: [1,2,3,4,5]
```

Abstraction

- To illustrate abstraction in Python, we'll use the abc module, which provides the infrastructure for defining abstract base classes.
- Abstraction supports encapsulation by keeping the implementation details hidden and exposing only the necessary methods.
- It is achieved through abstract classes and interfaces.
- Abstract classes can serve as a blueprint for multiple subclasses, promoting code reuse.
- Abstraction helps in reducing complexity and increasing the reusability of code.
- Abstract classes and methods make it easier to use polymorphism, where different subclasses can be treated as instances of the abstract base class, allowing for flexible and interchangeable code.

Abstraction

```python
from abc import ABC, abstractmethod

# Abstract base class
class Shape(ABC):

    @abstractmethod
    def area(self):
        pass


# Concrete class inheriting from abstract class
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height


# Concrete class inheriting from abstract class
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2


# Creating instances of concrete classes
rectangle = Rectangle(3, 4)
circle = Circle(5)

# Demonstrating abstraction
print(f"Rectangle area: {rectangle.area()}")  # Output: Rectangle area: 12
print(f"Circle area: {circle.area()}")        # Output: Circle area: 78.5
```

Coding problems

- Python program to build flashcard using class in Python

- Shuffle a deck of card with OOPS in Python

- How to create an empty class in Python?

- Student management system in Python

🚀 Ready to Level Up?

CONNECT FOR MORE INSIGHTS AND FUTURE UPDATES!

Shruti Bharat

@shrutibharat0105