**Q1. What is the relationship between classes and modules?**

**Answer:**

* A **module** is a file containing Python code (functions, variables, and classes).
* A **class** is a blueprint for creating objects and can be defined inside a module.
* Modules help in **organizing and reusing** code, and classes within modules provide **structured and reusable objects**.

**Example:**

# mymodule.py

class Car:

def \_\_init\_\_(self, brand):

self.brand = brand

You can import the module and use the class:

from mymodule import Car

c = Car("Toyota")

print(c.brand) # Output: Toyota

**Q2. How do you make instances and classes?**

**Answer:**

* **Creating a Class:** Use the class keyword.
* **Creating an Instance:** Call the class as if it were a function.

**Example:**

class Dog:

def \_\_init\_\_(self, name):

self.name = name

dog1 = Dog("Buddy") # Creating an instance of the Dog class

print(dog1.name) # Output: Buddy

**Q3. Where and how should class attributes be created?**

**Answer:**

* **Class attributes** are shared across all instances and are **defined outside any method** within the class.

**Example:**

class Car:

wheels = 4 # Class attribute

c1 = Car()

c2 = Car()

print(c1.wheels, c2.wheels) # Output: 4 4

**Q4. Where and how are instance attributes created?**

**Answer:**

* **Instance attributes** are unique to each object and are created inside the \_\_init\_\_ method using self.

**Example:**

class Car:

def \_\_init\_\_(self, brand):

self.brand = brand # Instance attribute

c1 = Car("Toyota")

c2 = Car("Ford")

print(c1.brand) # Output: Toyota

print(c2.brand) # Output: Ford

**Q5. What does the term "self" in a Python class mean?**

**Answer:**

* self represents the **current instance of the class**.
* It allows instance methods to access and modify instance attributes.

**Example:**

class Car:

def \_\_init\_\_(self, brand):

self.brand = brand # `self.brand` refers to the instance attribute

def display(self):

print("Brand:", self.brand)

c1 = Car("Tesla")

c1.display() # Output: Brand: Tesla

**Q6. How does a Python class handle operator overloading?**

**Answer:**

* Operator overloading is done by defining **special methods (dunder methods)** such as \_\_add\_\_, \_\_sub\_\_, \_\_eq\_\_, etc.

**Example:**

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, other):

return Point(self.x + other.x, self.y + other.y)

p1 = Point(2, 3)

p2 = Point(4, 5)

p3 = p1 + p2 # Calls p1.\_\_add\_\_(p2)

print(p3.x, p3.y) # Output: 6 8

**Q7. When do you consider allowing operator overloading of your classes?**

**Answer:**

* When you want **intuitive behavior** for user-defined objects, making them behave like built-in types.
* For example, **adding two vectors, comparing objects, or implementing custom arithmetic operations**.

**Q8. What is the most popular form of operator overloading?**

**Answer:**

* **Arithmetic operator overloading** (+, -, \*, /) is the most commonly used.
* This allows mathematical operations on custom objects like \_\_add\_\_, \_\_mul\_\_, etc.

**Example:**

class Number:

def \_\_init\_\_(self, value):

self.value = value

def \_\_add\_\_(self, other):

return Number(self.value + other.value)

n1 = Number(10)

n2 = Number(20)

n3 = n1 + n2 # Calls n1.\_\_add\_\_(n2)

print(n3.value) # Output: 30

**Q9. What are the two most important concepts to grasp in order to comprehend Python OOP code?**

**Answer:**

1. **Classes and Instances** – Understanding how **classes define blueprints** and how **instances represent real objects**.
2. **Encapsulation and Inheritance** – Knowing how attributes and methods **work inside a class** and how **one class can inherit from another**.