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

**Ans:** Classes in python act as a blueprint based on which objects are created. Objects are the very basis for object-oriented programming. The objects are the real-world entities, and class acts as a template that defines those objects. Classes are blueprints that allow you to create instances with attributes and bound functionality. Classes support inheritance, metaclasses, and descriptors.

A module can have zero or one or multiple classes. A class can be implemented in one or more **.py** files (modules). But often, we can organize a set of variables and functions into a class definition or just simply put them in a **.py** file and call it a module.

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

**Ans:** To create instances of a class, you call the class using class name and pass in whatever arguments its *\_\_init\_\_* method accepts.

**Example**: emp1 = Employee("Male", 25000), Here emp1 is an instance of class employee with attriubutes 'Male' and 25000.

Whereas for creating a class, we use the **Class** keyword. Class keyword is followed by classname and semicolon.

**Example:** Here Employee is a class created with class keyword with arguments gender and salary.

class Employee: def \_\_init\_\_(self, gender,salary): self.gender = gender self.salary = salary

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

**Ans:** Class attributes belong to the class itself they will be shared by all the instances. Such attributes are defined in the class body parts usually at the top, for legibility.

Like you will see in the examples in this section, class attributes are defined outside the \_\_init\_\_() function.

On the other hand, **instance attributes**, which are defined in the \_\_init\_\_() function, are class variables that allow us to define different values for each object of a class.

**Example:**

class Student:

school = "freeCodeCamp.org"

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

self.name = name

self.course = course

Student1 = Student("Jane", "JavaScript")

Student2 = Student("John", "Python")

print(Student1.name) # Jane

print(Student2.name) # John

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

**Ans:** Instance attributes are attributes or properties attached to an instance of a class.

The instance attribute is a variable that is unique to each object (instance). Every object of that class has its own copy of that variable. Any changes made to the variable don’t reflect in other objects of that class.

**Example:** In the case of our Car() class, each car has a specific color and style.

# A class with two instance attributes

class Car:

# initializer with instance attributes

def \_\_init\_\_(self, color, style):

self.color = color

self.style = style

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

**Ans:** self represents the instance of the class. By using the “self” we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.

The reason you need to use self. is because Python does not use the @ syntax to refer to instance attributes. Python decided to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically: the first parameter of methods is the instance the method is called on.

**Example:**

class food():

# init method or constructor

def \_\_init\_\_(self, fruit, color):

self.fruit = fruit

self.color = color

def show(self):

print("fruit is", self.fruit)

print("color is", self.color )

apple = food("apple", "red")

grapes = food("grapes", "green")

apple.show()

grapes.show()

**Output:**

Fruit is apple

color is red

Fruit is grapes

color is green

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

**Ans:** Suppose the user has two objects which are the physical representation of a user-defined data type class. The user has to add two objects using the "+" operator, and it gives an error. This is because the compiler does not know how to add two objects. So, the user has to define the function for using the operator, and that process is known as "operator overloading". The user can overload all the existing operators by they cannot create any new operator. Python provides some special functions, or we can say magic functions for performing operator overloading, which is automatically invoked when it is associated with that operator. Such as, when the user uses the "+" operator, the magic function \_\_add\_\_ will automatically invoke in the command where the "+" operator will be defined.

**Example:**

class complex\_1:

def \_\_init\_\_(self, X, Y):

self.X = X

self.Y = Y

# Now, we will add the two objects

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

return self.X + U.X, self.Y + U.Y

Object\_1 = complex\_1(23, 12)

Object\_2 = complex\_1(21, 22)

Object\_3 = Object\_1 + Object\_2

print (Object\_3)

**Output:**

(44, 34)

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

**Ans:** In Python, we can change the way operators work for user-defined types.

**For example,** the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.

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

**Ans:** The most popular form of operator overloading in python is by special methods called **Magic methods**. Which usually beign and end with double underscore **\_\_<method name>\_\_**.

A very popular and convenient example is the **Addition (+) operator**. Just think how the ‘+’ operator operates on two numbers and the same operator operates on two strings. It performs **“Addition”** on numbers whereas it performs **“Concatenation”** on strings.

**Example:**

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))

**Output:**

3

GeeksFor

3

GeeksFor

3

GeeksFor

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

**Ans:** In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

## Main Concepts of Object-Oriented Programming (OOPs)

* Class
* Objects
* Polymorphism
* Encapsulation
* Inheritance
* Data Abstraction