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

**ANS:-**

**Modules are about providing methods that you can use across multiple classes. Modules are about functions. You dont instantiate modules(e.g. Math module) you just use methods in them.Module is like library of methods.**

**Classes are about objects; it can hold its state (by keeping track of instance variables) and be duplicated as many times as you want.**

**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.**

**The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon.Explained below**

**class Student:**

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

**self.name = name**

**def \_\_str\_\_(self):**

**return f"This is {self.name} (an instance of a class Student)"**

**student1 = Student("Snehal Suryawanshi")**

**print(student1)**

**This is Snehal Suryawanshi (an instance of a class Student)**

**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.**

**class Student:**

**class\_teacher = 'Johnson' *# class attribute***

**def \_\_init\_\_(self,name, roll\_no): *# to bind attributes to class when object is created***

**self.name = name**

**self.roll\_no = roll\_no**

**def \_\_str\_\_(self):**

**return f"This is {self.name} (an instance of a class Student)"**

**student1 = Student("Akash Deep",25)**

**print(student1.class\_teacher)**

**Johnson**

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

**ANS:-**

**Unlike class attributes, instance attributes are not shared by objects. Every object has its own copy of the instance attribute. They are created in init method. How they are created is Explained below::**

**class Student:**

**def \_\_init\_\_(self,name, roll\_no): *# to bind attributes to class when object is created***

**self.name = name**

**self.roll\_no = roll\_no**

**def \_\_str\_\_(self):**

**return f"This is {self.name} (an instance of a class Student)"**

**student1 = Student("Snehal Suryawanshi",45)**

**print(student1.name) *# attributes are accessed by objects***

**print(student1.roll\_no)**

**Snehal Suryawanshi**

**45**

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

**ANS:-**

**Self represents the instance of the class. By using the “self” keyword 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.**

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

**ANS:-**

**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.**

***# + and \* operator is used for different purpose***

**print(1 + 2)**

***# concatenate two strings***

**print("abcd"+"efg")**

***# Product two numbers***

**print(13 \* 3)**

***# Repeat the String***

**print("Ineuron"\*4)**

**3**

**abcdefg**

**39**

**IneuronIneuronIneuronIneuron**

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

**ANS:-**

**Let us assumewe have an object called string1 which is a string object as defined below. Now, when we try to add a string to this string object, the compiler will throw an error. This is because the compiler doesn't know how to add them.**

***# declare our own string class***

**class String:**

**def \_\_init\_\_(self, string):**

**self.string = string**

**def \_\_repr\_\_(self):**

**return 'Object: {}'.format(self.string)**

**string1 = String('Hello')**

***# concatenate String object and a string***

**print(string1 +' world')**

**--------------------------------------------------------------**

**TypeError Traceback (most recent call last)**

**<ipython-input-6-57c4524b4029> in <module>**

**9**

**10 # concatenate String object and a string**

**---> 11 print(string1 +' world')**

**TypeError: unsupported operand type(s) for +: 'String' and 'str'**

### **This error can be avoided by adding the \_\_ add\_\_ method to the String class. This way, we are overloading the + operator to concatenate a string object with a string.**

***# declare our own string class***

**class String:**

**def \_\_init\_\_(self, string):**

**self.string = string**

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

**return self.string + other**

**def \_\_repr\_\_(self):**

**return 'Object: {}'.format(self.string)**

**string1 = String('Hello')**

***# concatenate String object and a string***

**print(string1 +' world')**

**Hello world**

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

**ANS:-**

**I feel, most popular form of operator overloading is of addition (+) operator. When two integers are passed to + operator, it will return the sum of two integers. When two strings are passed to + operator, it will return concatenation of two strings.**

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

**ANS:-**

1. **Inheritance.**
2. **Polymorphism.**