Q1. What is the relationship between classes and modules?

A module in Python is a file that can contain code, including class definitions. Classes are defined within modules, and a module can contain multiple classes along with other code. Modules allow you to organize related classes and functions into separate files, promoting code modularity and reusability.

Q2. How do you make instances and classes?

To create instances of a class, you call the class name followed by parentheses, which invokes the class's constructor (\_\_init\_\_ method). For example, to create an instance of a class named Person, you would write person\_instance = Person(). To create classes, you use the class keyword followed by the class name and a block of code that defines attributes and methods.

Q3. Where and how should class attributes be created?

Class attributes are created directly within the class block, but outside of any method. They are shared among all instances of the class and can be accessed using either the class name or instance name. Class attributes are typically defined above the \_\_init\_\_ method and can be used to store values that are common to all instances of the class.

Q4. Where and how are instance attributes created?

Instance attributes are created within the \_\_init\_\_ method of a class. They are specific to each instance of the class and are used to store individual data for each instance. Instance attributes are usually defined using the self parameter (e.g., self.attribute\_name = value) within the constructor.

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

In a Python class, self refers to the instance of the class itself. It is the first parameter passed to instance methods and points to the instance that the method is being called on. By convention, self is the name used for this parameter, but you can use any name you prefer (although sticking to the convention is recommended).

Q6. How does a Python class handle operator overloading?

Operator overloading in Python allows you to define special methods (e.g., \_\_add\_\_, \_\_sub\_\_, \_\_mul\_\_, etc.) that specify how instances of your class should behave when used with built-in operators. For example, you can define the \_\_add\_\_ method to specify the behavior of the + operator when applied to instances of your class.

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

Operator overloading is considered when you want to provide intuitive and meaningful behavior for built-in operators when used with instances of your class. For example, if you have a custom numeric class, you might want to define \_\_add\_\_ to enable addition of instances using the + operator.

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

One of the most popular forms of operator overloading is defining the \_\_str\_\_ method, which is used to provide a string representation of your object when it is used with the str() function or printed.

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

The two most important concepts to grasp in order to comprehend Python OOP code are:

* Classes and Instances: Understanding how classes are defined and how instances are created from classes. This includes knowing how to define attributes and methods, and how to access them using instances.
* Inheritance and Polymorphism: Understanding the concept of inheritance, where one class can inherit attributes and methods from another. Additionally, understanding polymorphism, which allows different classes to be used interchangeably through common interfaces like method names and attributes.