Q1. What is the relationship between classes and modules?

Ans: Classes are always nested inside a module; they are attributes of a module object. Classes and modules are both namespaces, but classes correspond to statements (not entire files) and support the OOP notions of multiple instances, inheritance, and operator overloading (modules do not). In a sense, a module is like a single instance class, without inheritance, which corresponds to an entire file of code.

Q2. How do you make instances and classes?

Ans: Classes are made by running class statements; instances are created by calling a class as though it were a function.

Q3. Where and how should be class attributes created?

Ans: Class attributes are created by assigning attributes to a class object. They are normally generated by top-level assignments nested in a class statement—each name assigned in the class statement block becomes an attribute of the class object (technically, the class statement's local scope morphs into the class object's attribute namespace, much like a module). Class attributes can also be created, though, by assigning attributes to the class anywhere a reference to the class object exists—even outside the class statement.

Q4. Where and how are instance attributes created?

Ans: Instance attributes are created by assigning attributes to an instance object. They are normally created within a class's method functions coded inside the class statement, by assigning attributes to the self-argument (which is always the implied instance). Again, though, they may be created by assignment anywhere a reference to the instance appears, even outside the class statement. Normally, all instance attributes are initialized in the __init__ constructor method; that way, later method calls can assume the attributes already exist.

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

Ans: self is the name commonly given to the first (leftmost) argument in a class's method function; Python automatically fills it in with the instance object that is the implied subject of the method call. This argument need not be called self (though this is a very strong convention); its position is what is significant.

Q6. How does a Python class handle operator overloading?

Ans: Operator overloading is coded in a Python class with specially named methods; they all begin and end with double underscores to make them unique. These are not built-in or reserved names; Python just runs them automatically when an instance appears in the corresponding operation. Python itself defines the mappings from operations to special method names.

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

Ans: Operator overloading is useful to implement objects that resemble built-in types (e.g., sequences or numeric objects such as matrixes), and to mimic the built-in type interface expected by a piece of code. Mimicking built-in type interfaces enables you to pass in class instances that also have state information (i.e., attributes that remember data between operation calls). You shouldn’t use operator overloading when a simple named method will suffice, though.

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

Ans: The __init__ constructor method is the most commonly used; almost every class uses this method to set initial values for instance attributes and perform other startup tasks.

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

Ans: The special self argument in method functions and the __init__ constructor method are the two cornerstones of OOP code in Python; if we get these, we should be able to read the text of most OOP Python code—apart from these, it’s largely just packages of functions. The inheritance search matters too, of course, but self represents the automatic object argument, and __init__ is widespread.