Q1. Define the relationship between a class and its instances. Is it a one-to-one or a one-to-many partnership?

- Answer: The relationship between a class and its instances is a one-to-many partnership. A class defines a blueprint, and you can create multiple instances (objects) based on that blueprint.

Q2. What kind of data is held only in an instance?

- Answer: Instance-specific data is held in instance variables. Each instance can have its own unique values for these variables.

Q3. What kind of knowledge is stored in a class?

- Answer: A class stores shared knowledge and behaviors in the form of class variables and methods that can be used by all instances created from that class.

Q4. What exactly is a method, and how is it different from a regular function?

- Answer: A method is a function defined within a class, associated with instances of that class. The key difference is that methods are called on instances and have access to instance-specific data (via `self`), while regular functions are standalone and not tied to any specific instance.

Q5. Is inheritance supported in Python, and if so, what is the syntax?

- Answer: Yes, Python supports inheritance. To create a subclass that inherits from a superclass, you define the subclass with the superclass in parentheses, like this: `class Subclass(Superclass):`.

Q6. How much encapsulation (making instance or class variables private) does Python support?

- Answer: Python supports encapsulation to some extent. You can make variables "private" by prefixing their names with a single underscore (e.g., `\_private\_var`). While this indicates that the variable should be treated as non-public, it's more of a convention than strict enforcement.

Q7. How do you distinguish between a class variable and an instance variable?

- Answer: Class variables are shared among all instances of a class and are defined within the class itself. Instance variables are specific to each instance and are assigned values within the constructor method (`\_\_init\_\_`) or other instance methods.

Q8. When, if ever, can `self` be included in a class's method definitions?

- Answer: `self` should be included as the first parameter in a class's method definitions to refer to the instance that the method is called on. It's a convention in Python, and you should include it in all instance methods.

Q9. What is the difference between the `\_\_add\_\_` and the `\_\_radd\_\_` methods?

- Answer: `\_\_add\_\_` is called when using the `+` operator on an object, while `\_\_radd\_\_` is called when the object is on the right side of the `+` operator. `\_\_radd\_\_` is used for reverse addition when the left operand doesn't support the operation.

Q10. When is it necessary to use a reflection method? When do you not need it, even though you support the operation in question?

- Answer: A reflection method (e.g., `\_\_str\_\_`) is necessary to define how an object should be represented as a string when using functions like `str()`. You don't need it if the default representation is sufficient or if you don't plan to use the object in string contexts.

Q11. What is the `\_\_iadd\_\_` method called?

- Answer: The `\_\_iadd\_\_` method is called for the `+=` operator.

Q12. Is the `\_\_init\_\_` method inherited by subclasses? What do you do if you need to customize its behavior within a subclass?

- Answer: The `\_\_init\_\_` method is not inherited by subclasses by default. To customize its behavior within a subclass, you can define a new `\_\_init\_\_` method in the subclass and explicitly call the superclass's `\_\_init\_\_` method using `super().\_\_init\_\_()` if needed.