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

The relationship between a class and its instances is a one-to-many partnership. A class serves as a blueprint or template for creating multiple instances, also known as objects. Each object represents a distinct entity with its own set of attributes and behaviors, all defined by the class. Therefore, a class can have multiple instances, but each instance belongs to only one class.

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

An instance holds specific data that is unique to itself. This includes instance variables or attributes, which represent the state or characteristics of the object. Instance variables store data that differs from one instance to another, allowing each object to have its own values and maintain individuality. The values of these variables define the current state of the object and can be accessed and modified within the instance's scope.

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

A class stores knowledge related to the structure and behavior of objects. It encapsulates common attributes and methods that are shared among multiple instances. The class defines the blueprint or template, specifying the attributes and their initial values, as well as the methods or functions that define the behavior of the objects. It represents the collective knowledge or definition of the objects that will be instantiated from it.

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

A method is a function that is associated with a specific object or class in object-oriented programming. It operates on the data and behavior of that object or class. Unlike regular functions, methods have access to the internal state and properties of the object they belong to and can modify them directly.

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

Yes, inheritance is supported in Python. The syntax for inheritance involves specifying the parent class within parentheses after the name of the child class. For example, to create a child class called `ChildClass` that inherits from a parent class called `ParentClass`, the syntax would be: `class ChildClass(ParentClass):`. The child class will inherit the attributes and methods of the parent class.

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

In Python, encapsulation is supported to a certain extent. Conventionally, variables and methods intended to be private are prefixed with a single underscore (\_), indicating that they should not be accessed directly from outside the class. However, Python does not enforce strict privacy, and these variables can still be accessed and modified from outside the class, albeit with a naming convention as a gentle reminder.

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

In Python, a class variable is shared among all instances of a class. It is defined within the class but outside any methods and is accessed using the class name. An instance variable, on the other hand, is unique to each instance of a class. It is defined within a method using the `self` keyword and is accessed using the instance name.

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

The `self` parameter is included in a class's method definitions in Python to refer to the instance of the class itself. It is required to be the first parameter in most cases, allowing the method to access and manipulate the instance's attributes and methods. Whenever an instance method needs to interact with the instance's state or behavior, `self` is included in the method definition.

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

The `\_\_add\_\_` method is used to define the behavior of the `+` operator when applied to an object. It defines addition from the object's perspective. The `\_\_radd\_\_` method, on the other hand, is the reflected version and is invoked when the object is on the right side of the `+` operator. It allows the object to participate in addition when the left operand does not support the operation directly.

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

Reflection methods, such as `\_\_radd\_\_`, are necessary when an object needs to participate in an operation but the left operand does not directly support it. For example, if the left operand does not have an `\_\_add\_\_` method. If both operands support the operation, reflection methods are not needed, as the operation can be handled directly by the appropriate method, like `\_\_add\_\_`, without invoking the reflected version.

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

The `\_\_iadd\_\_` method is called the in-place addition method. It is used to define the behavior of the `+=` operator for an object. This method allows an object to be modified in place when the `+=` operation is performed on it, providing an efficient way to update the object's state without creating a new object.

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

Yes, the `\_\_init\_\_` method is inherited by subclasses in Python. If you need to customize its behavior within a subclass, you can override the method by defining it in the subclass with the desired modifications. By doing so, you can provide additional parameters or initialize attributes specific to the subclass while still utilizing the initialization logic from the parent class by calling `super().\_\_init\_\_()` within the subclass's `\_\_init\_\_` method.