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

partnership, for example?

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

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

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

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

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

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

Q8. When, if ever, can self be included in a class&#39;s method definitions?

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

Q10. When is it necessary to use a reflection method? When do you not need it, even though you

support the operation in question?

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

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

behavior within a subclass?

### **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 single class can have multiple instances, each representing a distinct object created from that class. The class acts as a blueprint, and each instance is a concrete realization of that blueprint with its own set of data.

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

Data held only in an instance is known as **instance variables** or **instance attributes**. These variables store data that is unique to each instance of a class, meaning each object can have different values for these variables, even if they were created from the same class.

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

A class stores **class variables** or **class attributes**, which are shared among all instances of the class. It also holds **methods**, which are functions defined within the class that can operate on instance or class data. Class attributes represent knowledge or data that is common to all instances of the class.

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

A **method** is a function that is defined within a class and is intended to operate on objects of that class (i.e., instances). It differs from a regular function in that it is automatically passed the instance (typically named self) as its first argument when it is called. This allows methods to access and modify the instance's attributes.

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

Yes, inheritance is supported in Python. The syntax for defining a class that inherits from another class is:

python

class SubClassName(ParentClassName):

# class body

This allows the subclass to inherit attributes and methods from the parent class.

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

Python supports limited encapsulation. By convention, instance or class variables that are intended to be private are prefixed with an underscore (e.g., \_variable). However, this is only a convention and does not enforce strict access control. For stronger encapsulation, variables can be prefixed with a double underscore (e.g., \_\_variable), which triggers name mangling, making it harder to accidentally access the variable from outside the class.

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

* **Class variables** are defined within the class and outside any instance methods. They are shared among all instances of the class.
* **Instance variables** are defined within instance methods, typically within the \_\_init\_\_ method, using self. They are unique to each instance.

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

The self parameter must be included in a class's instance method definitions. It is the first parameter in the method and is automatically passed by Python when the method is called on an instance of the class. self refers to the specific instance that the method is being called on, allowing the method to access and modify the instance's attributes.

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

* The \_\_add\_\_ method is used to implement the addition operator (+) for instances of a class. It is called when the instance appears on the left-hand side of the operator (e.g., a + b).
* The \_\_radd\_\_ method is the reverse addition method. It is called when the instance appears on the right-hand side of the operator and the left-hand side does not support the addition operation (e.g., b + a where b does not have an \_\_add\_\_ method that can handle a).

### **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 like \_\_radd\_\_ are necessary when you want to ensure that your class can participate in operations where it appears on the right-hand side of a binary operator, and the left-hand operand does not support the operation. You do not need to implement reflection methods if the left-hand operand can handle the operation, or if your class instances are only expected to appear on the left-hand side.

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

The \_\_iadd\_\_ method is called the **in-place addition** method. It implements the += operation, which modifies an object in place, if possible, rather than 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. If you need to customize its behavior within a subclass, you can override the \_\_init\_\_ method in the subclass. To retain the functionality of the parent class’s \_\_init\_\_ method, you can call it explicitly using super():

python

class Parent:

def \_\_init\_\_(self, value):

self.value = value

class Child(Parent):

def \_\_init\_\_(self, value, extra):

super().\_\_init\_\_(value) # Call the parent class's \_\_init\_\_

self.extra = extra

This ensures that the initialization behavior from the parent class is included along with any additional behavior in the subclass.