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'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?

Answers

## 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 for creating multiple instances (objects), each of which can have its own unique state and behavior. While a single class can produce many instances, each instance is tied to the class that created it, and each instance can have different values for its instance variables.

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

Data that is held only in an instance is referred to as \*\*instance variables\*\*. These variables are specific to each instance of a class and can hold different values for different instances. For example, in a class representing a `Car`, instance variables might include `color`, `model`, and `year`, which can vary between different `Car` instances.

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

A class stores \*\*class variables\*\* and \*\*methods\*\* that define the shared behavior and characteristics of all instances of the class. Class variables are shared across all instances, meaning they have the same value for every instance unless explicitly overridden. Methods define functions that can be called on instances of the class, encapsulating the behavior that instances of the class can perform. Additionally, a class can contain metadata about its structure and relationships with other classes.

## 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 associated with instances of that class. The key difference between a method and a regular function is that methods implicitly take the instance (or class) as their first argument, typically named `self` for instance methods or `cls` for class methods. This allows methods to access and modify instance or class variables. In contrast, regular functions are standalone and do not have access to the instance or class context unless explicitly passed.

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

Yes, inheritance is supported in Python. It allows a class (subclass) to inherit attributes and methods from another class (superclass). The syntax for inheritance is as follows:

```python

class ParentClass:

# Parent class code

class ChildClass(ParentClass):

# Child class code

```

In this example, `ChildClass` inherits from `ParentClass`, gaining access to its methods and attributes.

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

Python supports a form of encapsulation, but it is not as strict as in some other programming languages. In Python, you can indicate that a variable is intended to be private by prefixing its name with an underscore (`\_`) for a weak "internal use" indicator or double underscores (`\_\_`) for name mangling, which makes it harder to access from outside the class. However, this is more of a convention than enforced access control, as Python relies on the principle of "we are all consenting adults here," allowing access if needed.

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

- \*\*Instance Variables\*\*: These are defined within methods (usually the `\_\_init\_\_` method) and are specific to each instance of the class. They are accessed using `self` and can hold different values for different instances.

```python

class MyClass:

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

self.instance\_var = value # Instance variable

```

- \*\*Class Variables\*\*: These are defined directly within the class body and are shared across all instances of the class. They are accessed using the class name or through an instance, but changes to a class variable via an instance do not affect the class variable itself unless explicitly done.

```python

class MyClass:

class\_var = 0 # Class variable

```

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

The `self` parameter must be included in the method definitions of instance methods to refer to the instance calling the method. It allows access to instance variables and other methods within the class. While `self` is not a keyword, it is a strong convention in Python to use `self` as the first parameter name for instance methods. It is not used in class methods or static methods, which use `cls` or no specific first parameter, respectively.

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

- \*\*`\_\_add\_\_`\*\*: This method is called when the left operand of the addition operator (`+`) is an instance of the class. It defines how to add two objects when the instance is on the left side.

```python

class MyClass:

def \_\_add\_\_(self, other):

return MyClass(self.value + other.value)

```

- \*\*`\_\_radd\_\_`\*\*: This method is called when the left operand is not an instance of the class, and the right operand is. It defines how to add two objects when the instance is on the right side of the addition operator. This is useful for handling cases where the left operand does not know how to handle the addition.

```python

class MyClass:

def \_\_radd\_\_(self, other):

return MyClass(other.value + self.value)

```

## 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 `\_\_getattr\_\_`, `\_\_setattr\_\_`, etc.) are necessary when you want to customize the behavior of attribute access or modification for a class. You would use them when you need to implement dynamic behavior, such as handling attributes that do not exist or enforcing specific rules for setting attributes.

However, you do not need to use reflection methods if the default behavior (direct attribute access) is sufficient for your use case. For example, if you are simply storing and retrieving instance variables without any special logic, you can rely on standard attribute access without implementing reflection methods.

## 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 a class. When you use `+=`, Python will call the `\_\_iadd\_\_` method if it is defined, allowing you to modify the instance in place rather than creating a new instance.

```python

class MyClass:

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

self.value = value

def \_\_iadd\_\_(self, other):

self.value += other.value

return self # Return the modified instance

```

## 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 `\_\_init\_\_` method in the subclass. To ensure that the parent class's `\_\_init\_\_` method is also called (to initialize inherited attributes), you should use the `super()` function.

Example:

```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 # Additional initialization for Child

```

In this example, the `Child` class customizes the `\_\_init\_\_` method while still invoking the parent class's initialization logic.