Python Advanced Assignment-19

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

Ans.

The relationship between a class and its instances is one-to-many. A class defines a blueprint for creating multiple instances (objects), each with its own state and behavior. You can create many instances from a single class, but each instance is independent of others.

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

Ans.

An instance holds instance variables (or attributes), which store data that is specific to that particular object. Each instance can have different values for these variables, even if they come from the same class.

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

Ans.

A class stores class variables and methods. Class variables are shared among all instances of the class, while methods define behaviors (functions) that can operate on instance or class data. A class may also include static methods and class methods, which don't operate on individual instances.

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

Ans.

A method is a function that is defined inside a class and is bound to the class or its instances. Methods typically take `self` as their first parameter (in the case of instance methods), allowing them to operate on instance data. A regular function is not bound to any class or instance and operates independently.

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

Ans.

Yes, Python supports inheritance. You define inheritance by specifying the parent class in parentheses when defining the child class.

Syntax:

```
class Parent:
    pass


class Child(Parent):  # Child inherits from Parent
    pass
```

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

Ans.

Python supports limited encapsulation. Variables prefixed with a single underscore (`_`) are treated as "protected" (indicating that they shouldn't be accessed directly), while variables prefixed with two underscores (`__`) are name-mangled to make them harder to access from outside the class. However, true privacy is not enforced, and private variables can still be accessed if needed.

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

Ans.

- Class variables are defined outside of methods and are shared among all instances of the class.

- Instance variables are defined inside methods (typically inside `__init__`) and are unique to each instance.

Example:

```
class MyClass:
    class_variable = 10  # Class variable


    def __init__(self):
        self.instance_variable = 20  # Instance variable
```

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

Ans.

`self` is included as the first parameter in instance methods of a class. It refers to the instance calling the method and is used to access instance variables and methods.

Example:

```
class MyClass:
    def my_method(self):
        print("This is an instance method.")
```

Q9: What is the difference between the `__add__` and the `__radd__` methods?

Ans.

- `__add__`: Implements addition (`+`) when the instance is on the left side of the operator.

- `__radd__`: Implements addition when the instance is on the right side of the operator (or if the left-side operand does not support addition).

Example:

```
class MyClass:
    def __add__(self, other):
        return "Left-side addition"

    def __radd__(self, other):
        return "Right-side addition"

# Usage
a = MyClass()
print(a + 1)  # Calls __add__
print(1 + a)  # Calls __radd__
```

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

Ans.

Reflection methods (like `__getattr__`, `__setattr__`) are necessary when you need to dynamically access or manipulate attributes or methods that are not explicitly defined in the class.

You don’t need reflection if all attributes or methods can be accessed directly through the class or instance. Reflection is typically used when the structure of objects is flexible or unknown at runtime.

Q11: What is the `__iadd__` method called?

Ans.

The `__iadd__` method is called for in-place addition (the `+=` operator). It allows modifying the object itself, rather than creating a new object.

Example:

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def __iadd__(self, other):
        self.value += other
        return self

# Usage
a = MyClass(5)
a += 10  # Calls __iadd__
print(a.value)  # Output: 15
```

Q12: Is the `__init__` method inherited by subclasses? What do you do if you need to customize its behavior within a subclass?

Ans.

Yes, the `__init__` method is inherited by subclasses. If you need to customize its behavior in the subclass, you can override it by defining a new `__init__` method in the subclass. You can also call the parent class's `__init__` method using `super()` to retain some of its behavior.

Example:

```
class Parent:
    def __init__(self, name):
        self.name = name


class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # Call Parent's __init__
        self.age = age  # Add new behavior
```