Oops_Assignment

September 10, 2024

[1]:
```
#Q1. What are the five key concepts of Object-Oriented Programming (OOP)?

#1. Key Concepts of Object-Oriented Programming (OOP)
#The five key concepts of Object-Oriented Programming are:

#Encapsulation: Bundling the data (attributes) and methods (functions) that␣
 ↪operate on the data into a single unit, or class. Access to this data is␣
 ↪controlled through access modifiers.

#Abstraction: Hiding the complex implementation details and showing only the␣
 ↪necessary features of an object. This is achieved using abstract classes and␣
 ↪methods.

#Inheritance: A mechanism that allows one class (child class) to inherit␣
 ↪attributes and methods from another class (parent class), promoting code␣
 ↪reuse.

#Polymorphism: The ability of different objects to respond to the same method␣
 ↪in different ways. This is typically achieved through method overriding.

#Class: A blueprint for creating objects (instances). It defines attributes and␣
 ↪methods that the created objects will have.
```

[2]:
```
#Q2. Write a Python class for a `Car` with attributes for `make`, `model`, and␣
 ↪`year`. Include a method to display the car's information.
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        print(f"Car: {self.year} {self.make} {self.model}")

# Example usage
car1 = Car("Toyota", "Camry", 2020)
car1.display_info()
```

```
Car: 2020 Toyota Camry
```

[3]:
```python
#Q3) Explain the difference between instance methods and class methods. Provide an example of each.
'''
Instance Methods: These methods operate on an instance of the class. They can access and modify the object's attributes.

Class Methods: These methods operate on the class itself rather than an instance of the class. They are marked with the @classmethod decorator and take cls as the first parameter instead of self.
'''


class Example:
    class_variable = 0

    def __init__(self, value):
        self.instance_variable = value

    def instance_method(self):
        return f"Instance method called with {self.instance_variable}"

    @classmethod
    def class_method(cls):
        return f"Class method called. Class variable: {cls.class_variable}"

# Example
obj = Example(10)
print(obj.instance_method())  # Calls instance method
print(Example.class_method())  # Calls class method
```

```
Instance method called with 10
Class method called. Class variable: 0
```

[4]:
```python
#Q#4) How does Python implement method overloading? Give an example.
'''
Python does not support traditional method overloading (i.e., multiple methods with the same name but different parameters). Instead, method overloading is typically handled by default arguments or variable arguments (*args).
'''

class Example:
    def greet(self, name=None):
        if name:
            print(f"Hello, {name}!")
        else:
```

```
            print("Hello!")


obj = Example()
obj.greet("Alice")
obj.greet()
```

```
Hello, Alice!
Hello!
```

[ ]:
```
#Q5) What are the three types of access modifiers in Python? How are they␣
 ↪denoted?

#Python uses the following conventions to denote access modifiers:

#Public: Accessible from outside the class. No underscore is used. Example:␣
 ↪self.attribute.
#Protected: Indicated by a single underscore (_). It's a convention to suggest␣
 ↪that it should not be accessed directly outside the class.
#Private: Indicated by a double underscore (__). Name mangling is used to␣
 ↪prevent access to this attribute outside the class.
```

[5]:
```
#Q6) Describe the five types of inheritance in Python. Provide a simple example␣
 ↪of multiple inheritance.
'''
1. Single Inheritance: A child class inherits from one parent class.
2. Multiple Inheritance: A child class inherits from multiple parent classes.
3. Multilevel Inheritance: A child class inherits from a parent class, which in␣
 ↪turn inherits from another class.
4. Hierarchical Inheritance: Multiple child classes inherit from a single␣
 ↪parent class.
5. Hybrid Inheritance: A combination of two or more types of inheritance
'''
class Parent1:
    def method1(self):
        print("Method from Parent1")

class Parent2:
    def method2(self):
        print("Method from Parent2")

class Child(Parent1, Parent2):
    pass


child = Child()
child.method1()
```

```
child.method2()
```

```
Method from Parent1
Method from Parent2
```

[ ]:
```
#Q7)  What is the Method Resolution Order (MRO) in Python? How can you retrieve it programmatically?
'''
The Method Resolution Order (MRO) is the order in which Python looks for a method in a hierarchy of classes during multiple inheritance.

You can retrieve it using the mro() method or __mro__ attribute.
'''

class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

print(D.mro())
```

[ ]:
```
#Q8) Create an abstract base class `Shape` with an abstract method `area()`. Then create two subclasses `Circle` and `Rectangle` that implement the `area()` method.
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

class Rectangle(Shape):
    def __init__(self, length, width):
```

```
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

circle = Circle(5)
rectangle = Rectangle(4, 6)
print(circle.area())  # Output: 78.5
print(rectangle.area())  # Output: 24
```

[9]:
```
#Q9) Demonstrate polymorphism by creating a function that can work with␣
 ↪different shape objects to calculate and print their areas.
def print_area(shape):
    print(f"The area is: {shape.area()}")

shapes = [Circle(5), Rectangle(4, 6)]
for shape in shapes:
    print_area(shape)
#output
The area is: 78.5
The area is: 24
```

[10]:
```
#Q10) Implement encapsulation in a `BankAccount` class with private attributes␣
 ↪for `balance` and `account_number`. Include methods for deposit, withdrawal,␣
 ↪and balance inquiry.
class BankAccount:
    def __init__(self, account_number, balance=0):
        self.__account_number = account_number
        self.__balance = balance

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount > self.__balance:
            print("Insufficient balance!")
        else:
            self.__balance -= amount

    def get_balance(self):
        return self.__balance

account = BankAccount("123456789")
account.deposit(1000)
account.withdraw(500)
print(account.get_balance())
```

```
500
```

[11]:
```python
#Q11)  Write a class that overrides the `__str__` and `__add__` magic methods.␣
 ↪What will these methods allow you to do?
class MyNumber:
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return f"MyNumber({self.value})"

    def __add__(self, other):
        if isinstance(other, MyNumber):
            return MyNumber(self.value + other.value)
        return NotImplemented


num1 = MyNumber(10)
num2 = MyNumber(20)
num3 = num1 + num2
print(num3)
```

```
MyNumber(30)
```

[12]:
```python
#Q12) Create a decorator that measures and prints the execution time of a␣
 ↪function.
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Execution time: {end_time - start_time} seconds")
        return result
    return wrapper

@timer
def slow_function():
    time.sleep(2)
    print("Function finished")

slow_function()
```

```
Function finished
Execution time: 2.0017247200012207 seconds
```

[13]:
```
#Q13)Explain the concept of the Diamond Problem in multiple inheritance. How does Python resolve it?
class A:
    def method(self):
        print("Method from A")

class B(A):
    def method(self):
        print("Method from B")

class C(A):
    def method(self):
        print("Method from C")

class D(B, C):
    pass


d = D()
d.method()
```

```
Method from B
```

[15]:
```
#Q14)  Write a class method that keeps track of the number of instances created from a class.
class InstanceCounter:
    count = 0

    def __init__(self):
        InstanceCounter.count += 1

    @classmethod
    def get_instance_count(cls):
        return cls.count


obj1 = InstanceCounter()
obj2 = InstanceCounter()
print(InstanceCounter.get_instance_count())
```

```
2
```

[16]:
```
#Q15) Implement a static method in a class that checks if a given year is a leap year.
class Year:
    @staticmethod
    def is_leap_year(year):
```

```
        return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)

# Example usage
print(Year.is_leap_year(2024))
```

```
True
```

[ ]:

[ ]:

[ ]:

[ ]: