Exercise 6 Program 1 - Single Inheritance in Python

Question Statement

Create a Python program to demonstrate single inheritance using a `Dog` class that inherits from an `Animal` class.

Input: Dog

Output: Woof!

Concept Explanation: Single Inheritance

What is Inheritance?

Inheritance is a fundamental concept of Object-Oriented Programming (OOP) that allows a new class (child) to derive properties and behaviors from an existing class (parent). This helps in code reusability and organization.

What is Single Inheritance?

Single inheritance means a child class inherits from only one parent class. It allows the child class to access the attributes and methods of the parent class while also defining its own additional properties.

How Single Inheritance Works in Python?

1. **Define a Parent Class (`Animal`)**
   - Contains common attributes and methods.
2. **Define a Child Class (`Dog`)**
   - Inherits from the `Animal` class.
   - Can override or extend the functionality of the parent class.
3. **Create an Instance of the Child Class (`Dog`)**
   - Calls the inherited methods as well as its own methods.

Simple Method: Python Code for Single Inheritance

```python
# Parent class
class Animal:
    def speak(self):
        return "Some sound"

# Child class inheriting from Animal
class Dog(Animal):
    def speak(self):  # Method overriding
        return "Woof!"

# Creating an instance of Dog
dog = Dog()
print(dog.speak())  # Output: Woof!
```

Code Execution Flow

1. **Class Definition (`Animal`)**
   - Defines a `speak()` method returning `"Some sound"`.
2. **Class Definition (`Dog`)**
   - Inherits from `Animal` and overrides the `speak()` method.
3. **Object Creation (`dog = Dog()`)**
   - Creates an instance of `Dog`.
4. **Method Call (`dog.speak()`)**
   - Calls the overridden `speak()` method from `Dog`, returning `"Woof!"`.

Alternative Method: Using `super()` to Call Parent Method

```python
# Parent class
```

```
class Animal:
    def speak(self):
        return "Some generic animal sound"

# Child class inheriting from Animal
class Dog(Animal):
    def speak(self):
            parent_sound = super().speak()  # Calling the parent class method
        return f"{parent_sound} and Woof!"

# Creating an instance of Dog
dog = Dog()
print(dog.speak())  # Output: Some generic animal sound and Woof!
```

Key Differences in This Method
- Uses `super().speak()` to call the parent class method.
- The overridden `speak()` method in `Dog` extends the parent method's output.

Full Execution Explanation
1. **The `Animal` Class**
    - Defines a method `speak()` returning `"Some generic animal sound"`.
2. **The `Dog` Class**
  - Inherits from `Animal` and overrides `speak()`.
  - Calls `super().speak()` to include the parent class behavior.
3. **Object Creation (`dog = Dog()`)**
  - An instance of `Dog` is created.
4. **Calling `dog.speak()`**
    - Executes `super().speak()`, which returns `"Some generic animal

sound"`.

- Appends `" and Woof!"`, resulting in `"Some generic animal sound and Woof!"`.

Exercise 6 Program 2 - Multilevel Inheritance in Python

Question Statement

Develop a Python program to show multilevel inheritance using a `Cat` class that inherits from a `Mammal` class, which in turn inherits from an `Animal` class.

Input: Cat

Output: Meow!

Concept Explanation: Multilevel Inheritance

What is Multilevel Inheritance?

Multilevel inheritance is a type of inheritance where a class inherits from another class, which itself is derived from another class. This forms a chain of inheritance.

How Multilevel Inheritance Works in Python?

1. **Define a Grandparent Class (`Animal`)**
   - Contains general attributes and methods.
2. **Define a Parent Class (`Mammal`)**
   - Inherits from `Animal` and adds more specific features.
3. **Define a Child Class (`Cat`)**
   - Inherits from `Mammal` and specializes further.
4. **Create an Instance of the Child Class (`Cat`)**
   - Calls methods from its own class, as well as inherited ones.

Simple Method: Python Code for Multilevel Inheritance

```python
# Grandparent class
```

```
class Animal:
    def sound(self):
        return "Some sound"

# Parent class
class Mammal(Animal):
    def feature(self):
        return "Warm-blooded"

# Child class inheriting from Mammal (which inherits from Animal)
class Cat(Mammal):
    def sound(self):  # Method overriding
        return "Meow!"

# Creating an instance of Cat
cat = Cat()
print(cat.sound())  # Output: Meow!
print(cat.feature())  # Output: Warm-blooded
```

Code Execution Flow
1. **Class Definition (`Animal`)**
   - Defines a `sound()` method returning `"Some sound"`.
2. **Class Definition (`Mammal`)**
   - Inherits from `Animal` and defines `feature()`.
3. **Class Definition (`Cat`)**
   - Inherits from `Mammal` and overrides `sound()`.
4. **Object Creation (`cat = Cat()`)**
   - An instance of `Cat` is created.
5. **Method Calls (`cat.sound()` and `cat.feature()`)**
   - Calls the overridden `sound()` method, returning `"Meow!"`.
   - Calls `feature()` from `Mammal`, returning `"Warm-blooded"`.

Alternative Method: Using `super()` to Call Parent Methods
```python
# Grandparent class
class Animal:
    def sound(self):
        return "Some general animal sound"

# Parent class
class Mammal(Animal):
    def sound(self):
            parent_sound = super().sound()  # Calling the grandparent method
        return f"{parent_sound} but specific to mammals"

# Child class
class Cat(Mammal):
    def sound(self):
        parent_sound = super().sound()  # Calling the parent method
        return f"{parent_sound}, and for a cat, it's Meow!"

# Creating an instance of Cat
cat = Cat()
print(cat.sound())  # Output: Some general animal sound but specific to mammals, and for a cat, it's Meow!
```

Key Differences in This Method
- Uses `super().sound()` to call methods from both `Mammal` and `Animal`.
- Allows the child class to include behavior from both parent and grandparent classes.

Full Execution Explanation

1. **The `Animal` Class**
   - Defines `sound()`, returning `"Some general animal sound"`.
2. **The `Mammal` Class**
   - Inherits `Animal` and overrides `sound()`, appending `"but specific to mammals"`.
3. **The `Cat` Class**
   - Inherits `Mammal` and overrides `sound()`, appending `"and for a cat, it's Meow!"`.
4. **Object Creation (`cat = Cat()`)**
   - An instance of `Cat` is created.
5. **Calling `cat.sound()`**
   - Executes `super().sound()`, calling `Mammal.sound()`, which itself calls `Animal.sound()`.
   - The final output is `"Some general animal sound but specific to mammals, and for a cat, it's Meow!"`.

Exercise 6 Program 3 - Hierarchical Inheritance in Python

Question Statement

Design a Python program to illustrate hierarchical inheritance using a `Dog` and `Cat` class that both inherit from an `Animal` class.

Input: Dog, Cat

Output: Woof!, Meow!

Concept Explanation: Hierarchical Inheritance

What is Hierarchical Inheritance?

Hierarchical inheritance is a type of inheritance where multiple child classes inherit from a single parent class.

This allows different child classes to share common functionality from the parent while also implementing their own unique behaviors.

How Hierarchical Inheritance Works in Python?

1. **Define a Parent Class (`Animal`)**
   - Contains common attributes and methods.
2. **Define Multiple Child Classes (`Dog` and `Cat`)**
   - Both inherit from `Animal` and implement their own methods.
3. **Create Instances of the Child Classes (`Dog` and `Cat`)**
   - Calls inherited and overridden methods.

Simple Method: Python Code for Hierarchical Inheritance

```python
# Parent class
class Animal:
```

```
    def speak(self):
        return "Some sound"

# Child class 1
class Dog(Animal):
    def speak(self):
        return "Woof!"

# Child class 2
class Cat(Animal):
    def speak(self):
        return "Meow!"

# Creating instances
dog = Dog()
cat = Cat()

print(dog.speak())  # Output: Woof!
print(cat.speak())  # Output: Meow!
```

Code Execution Flow
1. **Class Definition (`Animal`)**
   - Defines a `speak()` method returning `"Some sound"`.
2. **Class Definition (`Dog`)**
   - Inherits from `Animal` and overrides `speak()` to return `"Woof!"`.
3. **Class Definition (`Cat`)**
   - Inherits from `Animal` and overrides `speak()` to return `"Meow!"`.
4. **Object Creation (`dog = Dog()`, `cat = Cat()`)**
   - Instances of both `Dog` and `Cat` are created.
5. **Method Calls (`dog.speak()` and `cat.speak()`)**
   - Calls overridden `speak()` methods from respective classes.

Alternative Method: Using `super()` to Call Parent Method
```python
# Parent class
class Animal:
    def speak(self):
        return "Some general animal sound"

# Child class 1
class Dog(Animal):
    def speak(self):
            parent_sound = super().speak()  # Calling the parent class method
        return f"{parent_sound}, but for a dog, it's Woof!"

# Child class 2
class Cat(Animal):
    def speak(self):
            parent_sound = super().speak()  # Calling the parent class method
        return f"{parent_sound}, but for a cat, it's Meow!"

# Creating instances
dog = Dog()
cat = Cat()

print(dog.speak())  # Output: Some general animal sound, but for a dog, it's Woof!
print(cat.speak())  # Output: Some general animal sound, but for a cat, it's Meow!
```

Key Differences in This Method
- Uses `super().speak()` to call the parent method.

- The child classes (`Dog` and `Cat`) extend the behavior by appending their specific sounds.

Full Execution Explanation

1. **The `Animal` Class**
   - Defines `speak()`, returning `"Some general animal sound"`.
2. **The `Dog` Class**
   - Inherits `Animal` and overrides `speak()`, appending `", but for a dog, it's Woof!"`.
3. **The `Cat` Class**
   - Inherits `Animal` and overrides `speak()`, appending `", but for a cat, it's Meow!"`.
4. **Object Creation (`dog = Dog()`, `cat = Cat()`)**
   - Instances of `Dog` and `Cat` are created.
5. **Calling `dog.speak()` and `cat.speak()`**
   - Executes `super().speak()`, appending respective animal sounds.
   - The final outputs are:
     - `"Some general animal sound, but for a dog, it's Woof!"`
     - `"Some general animal sound, but for a cat, it's Meow!"`

Exercise 6 Program 4 - Method Overriding in Python

Question Statement

Create a Python program to demonstrate polymorphism using method overriding in a `Dog` and `Cat` class.

Input: Dog, Cat

Output: Woof!, Meow!

Concept Explanation: Polymorphism and Method Overriding

What is Polymorphism?

Polymorphism is an Object-Oriented Programming (OOP) concept that allows different classes to define methods with the same name but with different implementations.

What is Method Overriding?

Method overriding occurs when a child class provides a specific implementation for a method that is already defined in its parent class.

- The method in the child class has the same name as in the parent class.
- The child class's method overrides the parent's method.

How Method Overriding Works in Python?

1. **Define a Parent Class (`Animal`)**
   - Contains a method `speak()` with a general implementation.
2. **Define Multiple Child Classes (`Dog` and `Cat`)**
   - Both inherit from `Animal` and override `speak()`.
3. **Create Instances of the Child Classes (`Dog` and `Cat`)**
   - Calls the overridden `speak()` method from respective classes.

Simple Method: Python Code for Method Overriding
```python
# Parent class
class Animal:
    def speak(self):
        return "Some sound"

# Child class 1
class Dog(Animal):
    def speak(self):
        return "Woof!"

# Child class 2
class Cat(Animal):
    def speak(self):
        return "Meow!"

# Creating instances
dog = Dog()
cat = Cat()

print(dog.speak())  # Output: Woof!
print(cat.speak())  # Output: Meow!
```

Alternative Method: Using `super()` to Call Parent Method
```python
# Parent class
class Animal:
    def speak(self):
```

```
        return "Some general animal sound"

# Child class 1
class Dog(Animal):
    def speak(self):
        parent_sound = super().speak()  # Calling the parent method
        return f"{parent_sound}, but for a dog, it's Woof!"

# Child class 2
class Cat(Animal):
    def speak(self):
        parent_sound = super().speak()  # Calling the parent method
        return f"{parent_sound}, but for a cat, it's Meow!"

# Creating instances
dog = Dog()
cat = Cat()

print(dog.speak())  # Output: Some general animal sound, but for a dog, it's Woof!
print(cat.speak())  # Output: Some general animal sound, but for a cat, it's Meow!
```

Full Execution Explanation

1. **The `Animal` Class**
   - Defines `speak()`, returning `"Some general animal sound"`.
2. **The `Dog` Class**
   - Inherits `Animal` and overrides `speak()`, appending `", but for a dog, it's Woof!"`.
3. **The `Cat` Class**
   - Inherits `Animal` and overrides `speak()`, appending `", but for a

cat, it's Meow!"`.

4. **Object Creation (`dog = Dog()`, `cat = Cat()`)**

**   - Instances of `Dog` and `Cat` are created.**

5. **Calling `dog.speak()` and `cat.speak()`**

**   - Executes `super().speak()`, appending respective animal sounds.**

**   - The final outputs are:**

**   - `"Some general animal sound, but for a dog, it's Woof!"`**

**   - `"Some general animal sound, but for a cat, it's Meow!"`**

Exercise 6 Program 5 - Method Overloading in Python

Question Statement

Develop a Python program to show polymorphism using method overloading in a `Shape` class with `Circle` and `Rectangle` subclasses.

Input: Circle, Rectangle

Output: Area of Circle, Area of Rectangle

Concept Explanation: Polymorphism and Method Overloading

What is Polymorphism?

Polymorphism allows a single interface (method name) to represent different functionalities across different classes.

What is Method Overloading?

Method overloading allows a class to define multiple methods with the same name but different parameters. Python does not support true method overloading like Java or C++, but we can achieve similar behavior using:

1. **Default Arguments**
2. **Variable-Length Arguments (`*args` and `**kwargs`)**
3. **Function Overloading using `@staticmethod` or `@classmethod`**

How Method Overloading Works in Python?

1. **Define a Parent Class (`Shape`)**
   - Contains a generic `area()` method.
2. **Define Multiple Child Classes (`Circle` and `Rectangle`)**
   - Override `area()` to provide specific implementations.
3. **Create Instances of the Child Classes (`Circle` and `Rectangle`)**

- Calls their respective `area()` methods.

Simple Method: Python Code for Method Overloading Using Default Arguments

```python
import math

# Parent class
class Shape:
    def area(self, length, breadth=None):
        if breadth is None:  # Circle case (one argument)
            return math.pi * length * length
        else:  # Rectangle case (two arguments)
            return length * breadth

# Creating instances
shape = Shape()

# Finding areas
print("Area of Circle:", shape.area(5))  # Output: Area of Circle: 78.5398
print("Area of Rectangle:", shape.area(5, 10))  # Output: Area of Rectangle: 50
```

Alternative Method: Using `@staticmethod` for Overloading

```python
import math

# Parent class
```

```
class Shape:
    @staticmethod
    def area(*args):
        if len(args) == 1:  # Circle case
            return math.pi * args[0] * args[0]
        elif len(args) == 2:  # Rectangle case
            return args[0] * args[1]
        else:
            return "Invalid number of arguments"

# Finding areas
print("Area of Circle:", Shape.area(5))  # Output: Area of Circle: 78.5398
print("Area of Rectangle:", Shape.area(5, 10))  # Output: Area of Rectangle: 50
```

Full Execution Explanation

1. **The `Shape` Class**
   - Defines `area()` as a `@staticmethod`.
   - Uses `*args` to accept varying numbers of arguments.
   - Checks argument count to determine shape:
     - **1 argument** -> Circle (`pir²`).
     - **2 arguments** -> Rectangle (`length × breadth`).
     - **Other cases** -> Invalid input.
2. **Calling `Shape.area(5)`**
   - Executes the circle formula and returns `78.5398`.
3. **Calling `Shape.area(5, 10)`**
   - Executes the rectangle formula and returns `50`.

Polymorphism using Operator Overloading in Vector Class

Question

```
Question 6(f):
Prepare a Python program to demonstrate polymorphism using operator overloading in a Vector class.
Input: Vector1, Vector2
Output: Vector addition
```

Concept Explanation

```
Concept Explanation:

Polymorphism:
Polymorphism means "many forms". It allows the same operator or function to behave differently based on context.

Operator Overloading:
Operator overloading enables custom behavior for Python's built-in operators. We can override special methods like __add__() to change how "+" works for class objects.

In this example, we will create a Vector class and override the '+' operator to perform vector addition.
```

Code (Simple Method)

```
Code (Simple Method):
```

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"

v1 = Vector(3, 4)
v2 = Vector(5, 6)
v3 = v1 + v2

print("Vector1:", v1)
print("Vector2:", v2)
print("Vector Addition Result:", v3)
```

Code Execution Flow and Output

Polymorphism using Operator Overloading in Vector Class

```
Code Execution Flow:

1. v1 = Vector(3, 4) -> Creates a vector with components (3, 4)
2. v2 = Vector(5, 6) -> Creates a vector with components (5, 6)
3. v3 = v1 + v2     -> Calls v1.__add__(v2), returns Vector(8, 10)
4. print(...)       -> Outputs the vectors using __str__()

Output:
Vector1: (3, 4)
Vector2: (5, 6)
Vector Addition Result: (8, 10)
```

Alternative Method

```
Alternative Method (n-Dimensional Vectors):

class Vector:
    def __init__(self, values):
        self.values = values

    def __add__(self, other):
        result = [a + b for a, b in zip(self.values, other.values)]
        return Vector(result)

    def __str__(self):
        return str(tuple(self.values))

v1 = Vector([1, 2, 3])
v2 = Vector([4, 5, 6])
v3 = v1 + v2

print("Vector1:", v1)
print("Vector2:", v2)
print("Vector Addition Result:", v3)

Output:
Vector1: (1, 2, 3)
Vector2: (4, 5, 6)
Vector Addition Result: (5, 7, 9)
```