02_July_OOPs_inheritance_assignment 13

August 27, 2023

0.1 02_July_OOPs_inheritance

1. Explain what inheritance is in object-oriented programming and why it is used. Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class to inherit properties and behaviors from an existing class. The existing class is often referred to as the "base class" or "parent class," while the new class is called the "derived class" or "child class." This mechanism promotes code reuse and establishes a hierarchical relationship among classes. Inheritance is useful when you want to create new classes that share attributes and behaviors with existing classes. It should be used when there's a clear "is-a" relationship between the new class and the existing class.

2. Discuss the concept of single inheritance and multiple inheritance, highlighting their differences and advantages. Single inheritance refers to the concept where a class can inherit attributes and methods from only one parent class. In other words, a derived class can have only one immediate base class. Advantages of Single Inheritance:

Simplicity: Single inheritance simplifies class hierarchies, making them easier to understand and manage.

Avoiding Diamond Problem: Single inheritance inherently avoids the "diamond problem," a complication that can arise in multiple inheritance due to ambiguous method resolution.

Clearer Code: Since there's only one parent class, it's clear where the attributes and methods of the derived class are coming from.

Multiple inheritance allows a class to inherit attributes and methods from multiple parent classes. This means a derived class can have more than one immediate base class.

Advantages of Multiple Inheritance:

Code Reuse: Multiple inheritance enables a derived class to inherit features from multiple sources, facilitating greater code reuse.

Richer Functionality: You can combine traits from different parent classes to create a more comprehensive and specialized derived class.

Modeling Complex Relationships: Multiple inheritance is useful when modeling real-world situations involving multiple characteristics or behaviors.

Mixins: Multiple inheritance can be used to create mixins—small, reusable components that add specific behaviors to classes.

Differences Ambiguity: Multiple inheritance can lead to ambiguity if the same method is defined in multiple parent classes. This can result in conflicts during method resolution.

Complexity: Multiple inheritance can introduce complex class hierarchies, potentially making code harder to understand and maintain.

Diamond Problem: In multiple inheritance, the diamond problem occurs when a class inherits from two classes that share a common ancestor. This can lead to confusion in method resolution.

Method Resolution Order (MRO): Both single and multiple inheritance have a defined order in which the methods of parent classes are resolved in the derived class. In Python, the C3 Linearization algorithm is used to determine the MRO.

3. Explain the terms "base class" and "derived class" in the context of inheritance. Base Class (Parent Class): The class that provides the common attributes and methods to be inherited by other classes. Derived Class (Child Class): The class that inherits attributes and methods from a base class, and may also provide its unique attributes and methods.

4. What is the significance of the "protected" access modifier in inheritance? How does it differ from "private" and "public" modifiers? The "protected" access modifier allows class members to be accessed within the defining class and its subclasses (derived classes), but not from outside the class hierarchy. This means that only the base class and its derived classes can access and modify "protected" members, while other classes outside the hierarchy cannot. "private": Accessible only within the defining class. "protected": Accessible within the defining class and its derived classes. "public": Accessible from anywhere in the program.

5. What is the purpose of the "super" keyword in inheritance? Provide an example. The "super" keyword in object-oriented programming is used to access and call methods or attributes from the parent (base) class within a derived class. It is particularly useful when you want to extend or override methods in the derived class while still utilizing the behavior defined in the parent class. The "super" keyword helps maintain the inheritance hierarchy and ensures that both the parent and derived class functionalities are used together. The "super" keyword helps ensure that the parent class's methods and attributes are utilized in the derived class. It maintains the integrity of the inheritance hierarchy and promotes code reuse. By using "super," you can customize and extend the functionality of the derived class without losing the features defined in the parent class.

[13]:
```python
class Shape:
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

class Rectangle(Shape):
    def __init__(self, width, height):
        super().__init__(width, height)

    def area(self):
        return super().area()  # Reusing the area calculation logic from the␣
 ↪parent class

rectangle = Rectangle(5, 10)
print(rectangle.area())
```

```
50
```

6. Create a base class called "Vehicle" with attributes like "make", "model", and "year". Then, create a derived class called "Car" that inherits from "Vehicle" and adds an attribute called "fuel_type". Implement appropriate methods in both classes.

[14]:
```
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

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

class Car(Vehicle):
    def __init__(self, make, model, year, fuel_type):
        super().__init__(make, model, year)
        self.fuel_type = fuel_type

    def display_info(self):
        vehicle_info = super().display_info()
        return f"{vehicle_info}, Fuel Type: {self.fuel_type}"

# Create a Car instance
car = Car("Toyota", "Camry", 2022, "Gasoline")
print(car.display_info())
```

```
2022 Toyota Camry, Fuel Type: Gasoline
```

7. Create a base class called "Employee" with attributes like "name" and "salary." Derive two classes, "Manager" and "Developer," from "Employee." Add an additional attribute called "department" for the "Manager" class and "programming_language" for the "Developer" class.

[17]:
```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class Manager(Employee):
    def __init__(self, name, salary, department):
        super().__init__(name, salary)
        self.department = department

    def display_info(self):
        return f"Name: {self.name}, Salary: {self.salary}, Department: {self.
 ↪department}"

class Developer(Employee):
    def __init__(self, name, salary, programming_language):
        super().__init__(name, salary)
        self.programming_language = programming_language

    def display_info(self):
        return f"Name: {self.name}, Salary: {self.salary}, Programming Language:
 ↪ {self.programming_language}"

# Creating instances of Manager and Developer classes
manager = Manager("Mansour", 180000, "IT")
developer = Developer("Ali", 170000, "Python")

# Displaying information using the display_info method
print(manager.display_info())     # Output: Name: Mansour, Salary: 180000,␣
 ↪Department: IT
print(developer.display_info())  # Output: Name: Ali, Salary: 170000,␣
 ↪Programming Language: Python
```

```
Name: Mansour, Salary: 180000, Department: IT
Name: Ali, Salary: 170000, Programming Language: Python
```

8. Design a base class called "Shape" with attributes like "colour" and "border_width." Create derived classes, "Rectangle" and "Circle," that inherit from "Shape" and add specific attributes like "length" and "width" for the "Rectangle" class and "radius" for the "Circle" class.

[18]:
```
class Shape:
    def __init__(self, colour, border_width):
        self.colour = colour
        self.border_width = border_width

class Rectangle(Shape):
    def __init__(self, colour, border_width, length, width):
        super().__init__(colour, border_width)
        self.length = length
        self.width = width

    def display_info(self):
        return f"Shape: Rectangle, Colour: {self.colour}, Border Width: {self.
 ↪border_width}, Length: {self.length}, Width: {self.width}"
```

```
class Circle(Shape):
    def __init__(self, colour, border_width, radius):
        super().__init__(colour, border_width)
        self.radius = radius

    def display_info(self):
        return f"Shape: Circle, Colour: {self.colour}, Border Width: {self.
 ↪border_width}, Radius: {self.radius}"

# Creating instances of Rectangle and Circle classes
rectangle = Rectangle("Green", 2, 10, 5)
circle = Circle("Red", 1, 7)

# Displaying information using the display_info method
print(rectangle.display_info())  # Output: Shape: Rectangle, Colour: Green,␣
 ↪Border Width: 2, Length: 10, Width: 5
print(circle.display_info())     # Output: Shape: Circle, Colour: Red, Border␣
 ↪Width: 1, Radius: 7
```

```
Shape: Rectangle, Colour: Green, Border Width: 2, Length: 10, Width: 5
Shape: Circle, Colour: Red, Border Width: 1, Radius: 7
```

9. Create a base class called "Device" with attributes like "brand" and "model." Derive two classes, "Phone" and "Tablet," from "Device." Add specific attributes like "screen_size" for the "Phone" class and "battery_capacity" for the "Tablet" class.

[19]:
```
class Device:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

class Phone(Device):
    def __init__(self, brand, model, screen_size):
        super().__init__(brand, model)
        self.screen_size = screen_size

    def display_info(self):
        return f"Device Type: Phone, Brand: {self.brand}, Model: {self.model},␣
 ↪Screen Size: {self.screen_size}"

class Tablet(Device):
    def __init__(self, brand, model, battery_capacity):
        super().__init__(brand, model)
        self.battery_capacity = battery_capacity

    def display_info(self):
        return f"Device Type: Tablet, Brand: {self.brand}, Model: {self.model},␣
 ↪Battery Capacity: {self.battery_capacity}"

# Creating instances of Phone and Tablet classes
phone = Phone("Apple", "iPhone 14", "6.1 inches")
tablet = Tablet("Samsung", "Galaxy Tab N20", "6000 mAh")

# Displaying information using the display_info method
print(phone.display_info())  # Output: Device Type: Phone, Brand: Apple, Model:␣
 ↪iPhone 14, Screen Size: 6.1 inches
print(tablet.display_info()) # Output: Device Type: Tablet, Brand: Samsung,␣
 ↪Model: Galaxy Tab N20, Battery Capacity: 6000 mAh
```

```
Device Type: Phone, Brand: Apple, Model: iPhone 14, Screen Size: 6.1 inches
Device Type: Tablet, Brand: Samsung, Model: Galaxy Tab N20, Battery Capacity:
6000 mAh
```

10. Create a base class called "BankAccount" with attributes like "account_number" and

"balance." Derive two classes, "SavingsAccount" and "CheckingAccount," from

"BankAccount." Add specific methods like "calculate_interest" for the

"SavingsAccount" class and "deduct_fees" for the "CheckingAccount" class.

[23]:
```
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number
        self.balance = balance

class SavingsAccount(BankAccount):
    def __init__(self, account_number, balance):
        super().__init__(account_number, balance)

    def calculate_interest(self, rate):
        interest = self.balance * rate / 100
        self.balance += interest
        return interest

class CheckingAccount(BankAccount):
    def __init__(self, account_number, balance):
        super().__init__(account_number, balance)

    def deduct_fees(self, fee):
        if self.balance >= fee:
            self.balance -= fee
            return fee
```

```python
        else:
            return 0

# Creating instances of SavingsAccount and CheckingAccount classes
savings_account = SavingsAccount("10945", 4000)
checking_account = CheckingAccount("16789", 200)

# Calculating interest and deducting fees
savings_interest = savings_account.calculate_interest(2.5)  # Assuming 2.5%␣
 ↪interest rate
checking_fee = checking_account.deduct_fees(20)  # Assuming a fee of $20

# Displaying account balances and results
print(f"Savings Account Balance: ${savings_account.balance}")
print(f"Savings Account Interest Earned: ${savings_interest}")
print(f"Checking Account Balance: ${checking_account.balance}")
print(f"Checking Account Fee Deducted: ${checking_fee}")
```

```
Savings Account Balance: $4100.0
Savings Account Interest Earned: $100.0
Checking Account Balance: $180
Checking Account Fee Deducted: $20
```