Day 9 of Training at Ansh Info Tech

Topics Covered

Object Oriented Programming in Python

- **Inheritance in Python**
- **Single Inheritance**
- **Multiple Inheritance**
- **Multilevel Inheritance**
- **Hierarchical Inheritance**
- **Hybrid Inheritance**
- **12 Practice Questions on these Topics**

Summary

Inheritance in Python

Inheritance is a feature of object-oriented programming that allows a class to inherit attributes and methods from another class. This promotes code reusability and establishes a natural hierarchy between classes.

Single Inheritance

Single inheritance refers to a class inheriting from one parent class. This is the most straightforward form of inheritance where a subclass extends the functionality of a single superclass.

Multiple Inheritance

Multiple inheritance is when a class can inherit attributes and methods from more than one parent class. This allows for more complex relationships between classes but can also lead to ambiguity issues, which are handled by the method resolution order (MRO) in Python.

Multilevel Inheritance

Multilevel inheritance occurs when a class is derived from a class that is also derived from another class. This creates a chain of inheritance where a class inherits from a subclass of another class.

Hierarchical Inheritance

Hierarchical inheritance involves multiple subclasses inheriting from a single parent class. This type of inheritance is useful for defining a shared set of attributes and methods in the parent class that all subclasses will inherit.

Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. It includes a mix of single, multiple, multilevel, and hierarchical inheritance to create a more complex class structure.

Practice Questions on these Topics

1. Implement single inheritance with a base class and one derived class.
2. Demonstrate multiple inheritance with two parent classes and one child class.
3. Create a multilevel inheritance chain with three classes.
4. Use hierarchical inheritance to create several subclasses from a single parent class.
5. Illustrate hybrid inheritance with a combination of the above types.
6. **And many more...**

Introduction to Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows one class (child or subclass) to inherit the properties and methods of another class (parent or superclass). It promotes code reusability by enabling a new class to take on the attributes and behaviors of an existing class.

Key Concepts

1. **Parent Class (Superclass)**:
    - Also known as a base class or superclass.
    - It's the class whose attributes and methods are inherited by another class.
2. **Child Class (Subclass)**:
    - Also known as a derived class or subclass.
    - It inherits attributes and methods from its parent class and can also have its own additional attributes and methods.
3. **Inheritance Syntax in Python**:
    - In Python, inheritance is declared by specifying the parent class(es) in the definition of a child class.
    - Syntax: `class ChildClassName(ParentClassName):`
4. **Types of Inheritance**:
    - **Single Inheritance**: A child class inherits from only one parent class.
    - **Multiple Inheritance**: A child class inherits from multiple parent classes.
    - **Multilevel Inheritance**: One class is derived from another, which is itself derived from another class.
    - **Hierarchical Inheritance**: Multiple child classes inherit from the same parent class.
    - **Hybrid Inheritance**: Combination of two or more types of inheritance.

Advantages of Inheritance

- **Code Reusability**: Avoids redundant code by inheriting from existing classes.
- **Modularity**: Promotes a modular approach to software development.
- **Ease of Maintenance**: Changes made in the parent class automatically reflect in all child classes (depending on the design).

let's go through separate examples for each type of inheritance in Python: Single Inheritance and Multiple Inheritance.

Example 1: Single Inheritance Scenario:

Imagine a scenario where you have a base class Animal with generic attributes and methods, and a derived class Dog that inherits from Animal and adds specific attributes and methods related to dogs.

Single Inheritance Example:

```
# Parent class
class Animal:
    def __init__(self, species, legs):
        self.species = species
        self.legs = legs

    def make_sound(self):
        return "Some generic sound"

# Child class inheriting from Animal
class Dog(Animal):
    def __init__(self, species, legs, breed):
        super().__init__(species, legs)
        self.breed = breed

    def make_sound(self):
        return "Woof!"

    def describe(self):
        return f"A {self.breed} dog ({self.species}) with {self.legs} legs"

# Usage
my_dog = Dog("Canine", 4, "Labrador")
print(my_dog.make_sound())
print(my_dog.describe())
```

```
Woof!
A Labrador dog (Canine) with 4 legs
```

Example 2: Multiple Inheritance Scenario:

Consider a scenario where you have two parent classes, Father and Mother, and a child class Child that inherits from both Father and Mother, thereby inheriting attributes and methods from both parents.

Multiple Inheritance Example:

```
# Parent class 1
class Father:
    def __init__(self, eye_color):
        self.eye_color = eye_color

    def play_game(self):
        return "Playing chess with dad"

# Parent class 2
class Mother:
    def __init__(self, hair_color):
        self.hair_color = hair_color

    def cooking(self):
        return "Cooking with mom"

# Child class inheriting from both Father and Mother
class Child(Father, Mother):
    def __init__(self, eye_color, hair_color, name):
        Father.__init__(self, eye_color)
        Mother.__init__(self, hair_color)
        self.name = name

    def play(self):
        return f"{self.name} likes {super().play_game()} and {super().cooking()}"

# Usage
my_child = Child("Blue", "Brown", "Emma")
print(my_child.play())
```

```
Emma likes Playing chess with dad and Cooking with mom
```

Multilevel Inheritance Scenario:

In multilevel inheritance, a derived class serves as the base class for another class. For instance, consider a scenario where you have a Base class representing a basic entity, a Derived class inheriting from Base, and a FurtherDerived class inheriting from Derived, adding more specific attributes and methods.

Multilevel Inheritance Example:

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

    def greet(self):
        return f"Hello, {self.name}!"

# Derived class inheriting from Base
class Derived(Base):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

    def describe(self):
        return f"{self.name} is {self.age} years old"

# FurtherDerived class inheriting from Derived
class FurtherDerived(Derived):
    def __init__(self, name, age, hobby):
        super().__init__(name, age)
        self.hobby = hobby

    def show_hobby(self):
        return f"{self.name}'s hobby is {self.hobby}"

# Usage
person = FurtherDerived("Alice", 30, "Painting")
print(person.greet())
print(person.describe())
print(person.show_hobby())
```

```
Hello, Alice!
Alice is 30 years old
Alice's hobby is Painting
```

Hierarchical Inheritance Scenario:

In hierarchical inheritance, multiple derived classes inherit from a single base class. Consider a scenario where you have a Shape base class and multiple derived classes like Rectangle, Circle, and Triangle, each inheriting from Shape and adding specific attributes and methods.

Hierarchical Inheritance Example:

```
# Base class
class Shape:
    def __init__(self, color):
        self.color = color

    def area(self):
        pass

# Derived classes inheriting from Shape
class Rectangle(Shape):
    def __init__(self, color, width, height):
        super().__init__(color)
        self.width = width
        self.height = height

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

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

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

class Triangle(Shape):
    def __init__(self, color, base, height):
        super().__init__(color)
        self.base = base
        self.height = height
```

```
    def area(self):
        return 0.5 * self.base * self.height

# Usage
rectangle = Rectangle("Red", 5, 10)
circle = Circle("Blue", 7)
triangle = Triangle("Green", 4, 6)

print(rectangle.area())
print(circle.area())
print(triangle.area())
```

```
50
153.86
12.0
```

Hybrid Inheritance Scenario:

In hybrid inheritance, a combination of two or more types of inheritance (e.g., single, multiple, or multilevel) is used. Consider a scenario where you have a LivingBeing base class and multiple derived classes like Animal, Bird, and Fish, each having specific attributes and methods, and Human class inheriting from both Animal and LivingBeing.

Hybrid Inheritance Example:

```
# Base class
class LivingBeing:
    def __init__(self, kingdom):
        self.kingdom = kingdom

    def breathe(self):
        return "Breathing..."

# Intermediate base class
class Animal(LivingBeing):
    def __init__(self, kingdom, habitat):
        super().__init__(kingdom)
        self.habitat = habitat

    def sound(self):
        pass

# Another intermediate base class
class Mammal(Animal):
    def __init__(self, kingdom, habitat, warm_blooded=True):
        super().__init__(kingdom, habitat)
        self.warm_blooded = warm_blooded

    def has_fur(self):
        return "Has fur or hair"

# Derived class
class Human(Mammal):
    def __init__(self, kingdom, habitat, name):
        super().__init__(kingdom, habitat)
        self.name = name

    def sound(self):
        return "Speaking..."

    def walk(self):
        return "Walking on two legs"

# Usage
human = Human("Animalia", "Land", "Alice")
print(human.breathe())      # Output: Breathing...
print(human.sound())        # Output: Speaking...
print(human.has_fur())      # Output: Has fur or hair
print(human.walk())         # Output: Walking on two legs
```

```
Breathing...
Speaking...
Has fur or hair
Walking on two legs
```

1. Smart Home Automation System Question: Design a smart home automation system. Create a base class Device with common attributes like device_id, name, and methods like turn_on and turn_off. Create derived classes Light, Thermostat, and SecurityCamera with specific attributes and methods. Implement polymorphism to handle different devices.

- Requirements:

Device class with device_id, name, turn_on(), and turn_off(). Light class with brightness and color attributes. Thermostat class with temperature attribute and set_temperature() method. SecurityCamera class with resolution attribute and record() method. Use polymorphism to iterate over a list of devices and call their specific methods.

```
class Device:
    def __init__(self, device_id, name):
        self.device_id = device_id
        self.name = name
        self.is_on = False

    def turn_on(self):
        self.is_on = True
        print(f"{self.name} is now ON")

    def turn_off(self):
        self.is_on = False
        print(f"{self.name} is now OFF")

class Light(Device):
    def __init__(self, device_id, name, brightness, color):
        super().__init__(device_id, name)
        self.brightness = brightness
        self.color = color

    def set_brightness(self, brightness):
        self.brightness = brightness
        print(f"{self.name} brightness set to {self.brightness}")

    def set_color(self, color):
        self.color = color
        print(f"{self.name} color set to {self.color}")

class Thermostat(Device):
    def __init__(self, device_id, name, temperature):
        super().__init__(device_id, name)
        self.temperature = temperature

    def set_temperature(self, temperature):
        self.temperature = temperature
        print(f"{self.name} temperature set to {self.temperature}°C")

class SecurityCamera(Device):
    def __init__(self, device_id, name, resolution):
        super().__init__(device_id, name)
        self.resolution = resolution

    def record(self):
        if self.is_on:
            print(f"{self.name} is recording at {self.resolution} resolution")
        else:
            print(f"{self.name} is OFF and cannot record")

# Creating instances of the devices
light = Light("L001", "Living Room Light", 75, "Warm White")
thermostat = Thermostat("T001", "Living Room Thermostat", 22)
camera = SecurityCamera("C001", "Front Door Camera", "1080p")

# List of devices
devices = [light, thermostat, camera]

# Turning on all devices
for device in devices:
    device.turn_on()

# Performing specific actions on each device
light.set_brightness(85)
light.set_color("Cool White")
thermostat.set_temperature(24)
camera.record()

# Turning off all devices
for device in devices:
    device.turn_off()
```

2. Employee Management System Question: Develop an employee management system. Create a base class Employee with attributes like name, id, and salary. Create derived classes Manager, Developer, and Designer with specific attributes and methods. Implement method overriding and class-specific behavior.

Requirements:

Employee class with name, id, salary, and display_details() method. Manager class with team_size attribute and conduct_meeting() method. Developer class with programming_languages attribute and write_code() method. Designer class with design_tools attribute and create_design() method. Override display_details() in each subclass to include specific details.

```
class Employee:
  def __init__(self, name, id, salary):
    self.name = name
    self.id = id
    self.salary = salary

  def display_details(self):
    print("Name:", self.name)
    print("ID:", self.id)
    print("Salary:", self.salary)

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

  def conduct_meeting(self):
    print("Conducting meeting with team size:", self.team_size)

  def display_details(self):
    super().display_details()
    print("Team Size:", self.team_size)

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

  def write_code(self):
    print("Writing code in:", self.programming_languages)

  def display_details(self):
    super().display_details()
    print("Programming Languages:", self.programming_languages)

class Designer(Employee):
  def __init__(self, name, id, salary, design_tools):
    super().__init__(name, id, salary)
    self.design_tools = design_tools

  def create_design(self):
    print("Creating design using:", self.design_tools)

  def display_details(self):
    super().display_details()
    print("Design Tools:", self.design_tools)

# Creating instances of each type of employee
manager = Manager("Alice", "M001", 80000, 5)
developer = Developer("Bob", "D001", 70000, ["Python", "JavaScript", "C++"])
designer = Designer("Charlie", "DS001", 65000, ["Photoshop", "Illustrator"])

# List of employees
employees = [manager, developer, designer]

# Displaying details of each employee
for employee in employees:
    employee.display_details()
    print()

# Performing specific actions for each type of employee
manager.conduct_meeting()
developer.write_code()
designer.create_design()
```

```
Name: Alice
ID: M001
Salary: 80000
Team Size: 5
```

```
Name: Bob
ID: D001
Salary: 70000
Programming Languages: ['Python', 'JavaScript', 'C++']

Name: Charlie
ID: DS001
Salary: 65000
Design Tools: ['Photoshop', 'Illustrator']

Conducting meeting with team size: 5
Writing code in: ['Python', 'JavaScript', 'C++']
Creating design using: ['Photoshop', 'Illustrator']
```

3. E-Learning Platform Question: Create an e-learning platform. Design a base class Course with common attributes like course_id, title, and methods like enroll_student and unenroll_student. Create derived classes ProgrammingCourse, DesignCourse, and MathCourse with specific attributes and methods.

Requirements:

Course class with course_id, title, students, enroll_student(), and unenroll_student(). ProgrammingCourse class with languages attribute and start_project() method. DesignCourse class with software attribute and assign_design_task() method. MathCourse class with difficulty_level attribute and assign_homework() method. Use polymorphism to manage courses and their specific tasks.

```
class Course:
    def __init__(self, course_id, title):
        self.course_id = course_id
        self.title = title
        self.students = []

    def enroll_student(self, student_name):
        if student_name not in self.students:
            self.students.append(student_name)
            print(f"{student_name} has been enrolled in {self.title}")
        else:
            print(f"{student_name} is already enrolled in {self.title}")

    def unenroll_student(self, student_name):
        if student_name in self.students:
            self.students.remove(student_name)
            print(f"{student_name} has been unenrolled from {self.title}")
        else:
            print(f"{student_name} is not enrolled in {self.title}")

    def display_details(self):
        print(f"Course ID: {self.course_id}")
        print(f"Course Title: {self.title}")
        print(f"Enrolled Students: {', '.join(self.students) if self.students else 'No students enrolled'}")


class ProgrammingCourse(Course):
    def __init__(self, course_id, title, languages):
        super().__init__(course_id, title)
        self.languages = languages

    def start_project(self):
        print(f"Starting a project in {', '.join(self.languages)} for {self.title}")

    def display_details(self):
        super().display_details()
        print(f"Programming Languages: {', '.join(self.languages)}")


class DesignCourse(Course):
    def __init__(self, course_id, title, software):
        super().__init__(course_id, title)
        self.software = software

    def assign_design_task(self):
        print(f"Assigning a design task using {self.software} for {self.title}")

    def display_details(self):
        super().display_details()
        print(f"Design Software: {self.software}")


class MathCourse(Course):
    def __init__(self, course_id, title, difficulty_level):
        super().__init__(course_id, title)
        self.difficulty_level = difficulty_level

    def assign_homework(self):
        print(f"Assigning homework for {self.title} at {self.difficulty_level} level")

    def display_details(self):
        super().display_details()
        print(f"Difficulty Level: {self.difficulty_level}")


# Creating instances of each type of course
programming_course = ProgrammingCourse("P101", "Intro to Python", ["Python", "Django"])
design_course = DesignCourse("D101", "Graphic Design Basics", "Photoshop")
math_course = MathCourse("M101", "Calculus I", "Intermediate")

# List of courses
courses = [programming_course, design_course, math_course]

# Displaying details of each course
for course in courses:
    course.display_details()
    print()
```

```
# Enrolling and unenrolling students
programming_course.enroll_student("Alice")
design_course.enroll_student("Bob")
math_course.enroll_student("Charlie")

print()

# Performing specific actions for each type of course
programming_course.start_project()
design_course.assign_design_task()
math_course.assign_homework()

print()

# Displaying details of each course again to see changes
for course in courses:
    course.display_details()
    print()

# Unenrolling a student and showing updated details
programming_course.unenroll_student("Alice")
design_course.unenroll_student("Bob")
math_course.unenroll_student("Charlie")

print()

# Displaying details of each course again to see changes
for course in courses:
    course.display_details()
    print()
```

4. Restaurant Management System Question: Design a restaurant management system. Create a base class Restaurant with common attributes like name, location, and methods like open_restaurant and close_restaurant. Create derived classes FastFoodRestaurant, FineDiningRestaurant, and Cafe with specific attributes and methods.

Requirements:

Restaurant class with name, location, open_restaurant(), and close_restaurant(). FastFoodRestaurant class with drive_thru attribute and serve_fast_food() method. FineDiningRestaurant class with dress_code attribute and serve_gourmet_dishes() method. Cafe class with coffee_types attribute and serve_coffee() method. Override open_restaurant() and close_restaurant() in each subclass.

```
class Restaurant:
    def __init__(self, name, location):
        self.name = name
        self.location = location
        self.is_open = False

    def open_restaurant(self):
        if not self.is_open:
            self.is_open = True
            print(f"{self.name} at {self.location} is now open.")
        else:
            print(f"{self.name} at {self.location} is already open.")

    def close_restaurant(self):
        if self.is_open:
            self.is_open = False
            print(f"{self.name} at {self.location} is now closed.")
        else:
            print(f"{self.name} at {self.location} is already closed.")

    def display_details(self):
        status = "Open" if self.is_open else "Closed"
        print(f"Restaurant Name: {self.name}")
        print(f"Location: {self.location}")
        print(f"Status: {status}")


class FastFoodRestaurant(Restaurant):
    def __init__(self, name, location, drive_thru):
        super().__init__(name, location)
        self.drive_thru = drive_thru

    def serve_fast_food(self):
        print(f"Serving fast food at {self.name}.")

    def open_restaurant(self):
        super().open_restaurant()
        print(f"Drive-thru at {self.name} is now open.")

    def close_restaurant(self):
        super().close_restaurant()
        print(f"Drive-thru at {self.name} is now closed.")

    def display_details(self):
        super().display_details()
        print(f"Drive-thru: {'Available' if self.drive_thru else 'Not Available'}")


class FineDiningRestaurant(Restaurant):
    def __init__(self, name, location, dress_code):
        super().__init__(name, location)
        self.dress_code = dress_code

    def serve_gourmet_dishes(self):
        print(f"Serving gourmet dishes at {self.name}.")

    def open_restaurant(self):
        super().open_restaurant()
        print(f"Please adhere to the dress code: {self.dress_code}.")

    def close_restaurant(self):
        super().close_restaurant()
        print(f"{self.name} is now closed. We hope you enjoyed our gourmet dishes.")

    def display_details(self):
        super().display_details()
        print(f"Dress Code: {self.dress_code}")


class Cafe(Restaurant):
    def __init__(self, name, location, coffee_types):
        super().__init__(name, location)
        self.coffee_types = coffee_types

    def serve_coffee(self):
        print(f"Serving coffee at {self.name}. Available types: {', '.join(self.coffee_types)}")
```

```
    def open_restaurant(self):
        super().open_restaurant()
        print(f"Come enjoy our coffee at {self.name}.")

    def close_restaurant(self):
        super().close_restaurant()
        print(f"{self.name} is now closed. See you tomorrow for your coffee fix.")

    def display_details(self):
        super().display_details()
        print(f"Coffee Types: {', '.join(self.coffee_types)}")


# Creating instances of each type of restaurant
fast_food = FastFoodRestaurant("Fast Bites", "123 Fast Lane", drive_thru=True)
fine_dining = FineDiningRestaurant("Elegant Eats", "456 Luxury Blvd", dress_code="Formal")
cafe = Cafe("Coffee Corner", "789 Brew Street", coffee_types=["Espresso", "Latte", "Cappuccino"])

# List of restaurants
restaurants = [fast_food, fine_dining, cafe]

# Displaying details of each restaurant
for restaurant in restaurants:
    restaurant.display_details()
    print()

# Opening and closing restaurants
for restaurant in restaurants:
    restaurant.open_restaurant()
    print()
    restaurant.close_restaurant()
    print()

# Performing specific actions for each type of restaurant
fast_food.serve_fast_food()
fine_dining.serve_gourmet_dishes()
cafe.serve_coffee()

print()

# Displaying details of each restaurant again to see changes
for restaurant in restaurants:
    restaurant.display_details()
    print()
```

5. Hospital Management System Question: Develop a hospital management system. Create a base class Person with common attributes like name, age, and methods like display_info. Create derived classes Doctor, Nurse, and Patient with specific attributes and methods. Implement polymorphism and method overriding.

Requirements:

Person class with name, age, and display_info(). Doctor class with specialization attribute and diagnose() method. Nurse class with shift attribute and assist() method. Patient class with ailment attribute and receive_treatment() method. Override display_info() in each subclass to include specific details.

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

    def display_info(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")


class Doctor(Person):
    def __init__(self, name, age, specialization):
        super().__init__(name, age)
        self.specialization = specialization

    def diagnose(self):
        print(f"Dr. {self.name} is diagnosing a patient.")

    def display_info(self):
        super().display_info()
        print(f"Specialization: {self.specialization}")


class Nurse(Person):
    def __init__(self, name, age, shift):
        super().__init__(name, age)
        self.shift = shift

    def assist(self):
        print(f"Nurse {self.name} is assisting in the {self.shift} shift.")

    def display_info(self):
        super().display_info()
        print(f"Shift: {self.shift}")


class Patient(Person):
    def __init__(self, name, age, ailment):
        super().__init__(name, age)
        self.ailment = ailment

    def receive_treatment(self):
        print(f"Patient {self.name} is receiving treatment for {self.ailment}.")

    def display_info(self):
        super().display_info()
        print(f"Ailment: {self.ailment}")


# Creating instances of each type of person
doctor = Doctor("Alice Smith", 45, "Cardiology")
nurse = Nurse("Bob Johnson", 30, "Night")
patient = Patient("Charlie Brown", 25, "Flu")

# List of people in the hospital
hospital_people = [doctor, nurse, patient]

# Displaying details of each person
for person in hospital_people:
    person.display_info()
    print()

# Performing specific actions for each type of person
doctor.diagnose()
nurse.assist()
patient.receive_treatment()

print()

# Displaying details of each person again to see changes
for person in hospital_people:
    person.display_info()
    print()
```

6. Online Booking System Question: Create an online booking system. Design a base class Booking with common attributes like booking_id, date, and methods like confirm_booking and cancel_booking. Create derived classes HotelBooking, FlightBooking, and EventBooking with specific attributes and methods.

Requirements:

Booking class with booking_id, date, confirm_booking(), and cancel_booking(). HotelBooking class with hotel_name attribute and reserve_room() method. FlightBooking class with flight_number attribute and book_seat() method. EventBooking class with event_name attribute and reserve_ticket() method. Use polymorphism to manage different bookings.

```
class Booking:
    def __init__(self, booking_id, date):
        self.booking_id = booking_id
        self.date = date
        self.is_confirmed = False

    def confirm_booking(self):
        if not self.is_confirmed:
            self.is_confirmed = True
            print(f"Booking {self.booking_id} for {self.date} is confirmed.")
        else:
            print(f"Booking {self.booking_id} for {self.date} is already confirmed.")

    def cancel_booking(self):
        if self.is_confirmed:
            self.is_confirmed = False
            print(f"Booking {self.booking_id} for {self.date} is cancelled.")
        else:
            print(f"Booking {self.booking_id} for {self.date} is not confirmed yet.")

    def display_info(self):
        status = "Confirmed" if self.is_confirmed else "Not Confirmed"
        print(f"Booking ID: {self.booking_id}")
        print(f"Date: {self.date}")
        print(f"Status: {status}")


class HotelBooking(Booking):
    def __init__(self, booking_id, date, hotel_name):
        super().__init__(booking_id, date)
        self.hotel_name = hotel_name

    def reserve_room(self):
        print(f"Reserving a room at {self.hotel_name} for booking {self.booking_id}.")

    def display_info(self):
        super().display_info()
        print(f"Hotel Name: {self.hotel_name}")


class FlightBooking(Booking):
    def __init__(self, booking_id, date, flight_number):
        super().__init__(booking_id, date)
        self.flight_number = flight_number

    def book_seat(self):
        print(f"Booking a seat on flight {self.flight_number} for booking {self.booking_id}.")

    def display_info(self):
        super().display_info()
        print(f"Flight Number: {self.flight_number}")


class EventBooking(Booking):
    def __init__(self, booking_id, date, event_name):
        super().__init__(booking_id, date)
        self.event_name = event_name

    def reserve_ticket(self):
        print(f"Reserving a ticket for event {self.event_name} for booking {self.booking_id}.")

    def display_info(self):
        super().display_info()
        print(f"Event Name: {self.event_name}")


# Creating instances of each type of booking
hotel_booking = HotelBooking("HB001", "2024-06-20", "Grand Hotel")
flight_booking = FlightBooking("FB002", "2024-06-21", "AA123")
event_booking = EventBooking("EB003", "2024-06-22", "Rock Concert")

# List of bookings
bookings = [hotel_booking, flight_booking, event_booking]

# Displaying details of each booking
for booking in bookings:
    booking.display_info()
```

```
    print()

# Confirming and cancelling bookings
for booking in bookings:
    booking.confirm_booking()
    print()
    booking.cancel_booking()
    print()

# Performing specific actions for each type of booking
hotel_booking.reserve_room()
flight_booking.book_seat()
event_booking.reserve_ticket()

print()

# Displaying details of each booking again to see changes
for booking in bookings:
    booking.display_info()
    print()
```

7. University Course Management Question: Design a university course management system. Create a base class UniversityMember with common attributes like name, id, and methods like show_details. Create derived classes Professor, Student, and Administrator with specific attributes and methods.

Requirements:

UniversityMember class with name, id, and show_details(). Professor class with department attribute and teach_course() method. Student class with major attribute and enroll_course() method. Administrator class with role attribute and manage_operations() method. Override show_details() in each subclass to include specific details.

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

    def show_details(self):
        print(f"Name: {self.name}")
        print(f"ID: {self.id}")


class Professor(UniversityMember):
    def __init__(self, name, id, department):
        super().__init__(name, id)
        self.department = department

    def teach_course(self):
        print(f"Professor {self.name} is teaching a course in the {self.department} department.")

    def show_details(self):
        super().show_details()
        print(f"Department: {self.department}")


class Student(UniversityMember):
    def __init__(self, name, id, major):
        super().__init__(name, id)
        self.major = major

    def enroll_course(self):
        print(f"Student {self.name} is enrolling in a course for their major in {self.major}.")

    def show_details(self):
        super().show_details()
        print(f"Major: {self.major}")


class Administrator(UniversityMember):
    def __init__(self, name, id, role):
        super().__init__(name, id)
        self.role = role

    def manage_operations(self):
        print(f"Administrator {self.name} is managing operations as a {self.role}.")

    def show_details(self):
        super().show_details()
        print(f"Role: {self.role}")


# Creating instances of each type of university member
professor = Professor("Dr. Smith", "P001", "Computer Science")
student = Student("Alice Johnson", "S001", "Biology")
administrator = Administrator("John Doe", "A001", "Registrar")

# List of university members
members = [professor, student, administrator]

# Displaying details of each university member
for member in members:
    member.show_details()
    print()

# Performing specific actions for each type of university member
professor.teach_course()
student.enroll_course()
administrator.manage_operations()

print()

# Displaying details of each university member again to see changes
for member in members:
    member.show_details()
    print()
```

8. Transport System Question: Develop a transport system. Create a base class Transport with common attributes like id, capacity, and methods like start and stop. Create derived classes Bus, Train, and Airplane with specific attributes and methods. Implement method overriding and polymorphism.

Requirements:

Transport class with id, capacity, start(), and stop(). Bus class with route_number attribute and pick_up_passengers() method. Train class with number_of_coaches attribute and depart() method. Airplane class with flight_code attribute and take_off() method. Use polymorphism to manage different transport types.

```
class Transport:
    def __init__(self, id, capacity):
        self.id = id
        self.capacity = capacity

    def start(self):
        print(f"Transport {self.id} is starting.")

    def stop(self):
        print(f"Transport {self.id} is stopping.")

    def show_details(self):
        print(f"ID: {self.id}")
        print(f"Capacity: {self.capacity}")


class Bus(Transport):
    def __init__(self, id, capacity, route_number):
        super().__init__(id, capacity)
        self.route_number = route_number

    def pick_up_passengers(self):
        print(f"Bus {self.id} on route {self.route_number} is picking up passengers.")

    def show_details(self):
        super().show_details()
        print(f"Route Number: {self.route_number}")


class Train(Transport):
    def __init__(self, id, capacity, number_of_coaches):
        super().__init__(id, capacity)
        self.number_of_coaches = number_of_coaches

    def depart(self):
        print(f"Train {self.id} with {self.number_of_coaches} coaches is departing.")

    def show_details(self):
        super().show_details()
        print(f"Number of Coaches: {self.number_of_coaches}")


class Airplane(Transport):
    def __init__(self, id, capacity, flight_code):
        super().__init__(id, capacity)
        self.flight_code = flight_code

    def take_off(self):
        print(f"Airplane {self.id} with flight code {self.flight_code} is taking off.")

    def show_details(self):
        super().show_details()
        print(f"Flight Code: {self.flight_code}")


# Creating instances of each type of transport
bus = Bus("B001", 50, "10A")
train = Train("T001", 200, 10)
airplane = Airplane("A001", 150, "AA123")

# List of transports
transports = [bus, train, airplane]

# Displaying details of each transport
for transport in transports:
    transport.show_details()
    print()

# Starting and stopping each transport
for transport in transports:
    transport.start()
    transport.stop()
    print()

# Performing specific actions for each type of transport
bus.pick_up_passengers()
train.depart()
```

```
airplane.take_off()
```

9. Smart City Infrastructure Question: Create a smart city infrastructure management system. Design a base class Infrastructure with common attributes like location, status, and methods like activate and deactivate. Create derived classes TrafficLight, StreetLight, and SurveillanceCamera with specific attributes and methods.

Requirements:

Infrastructure class with location, status, activate(), and deactivate(). TrafficLight class with intersection_id attribute and change_light() method. StreetLight class with lumens attribute and adjust_brightness() method. SurveillanceCamera class with resolution attribute and start_recording() method. Override activate() and deactivate() in each subclass.

```
class Infrastructure:
    def __init__(self, location, status="inactive"):
        self.location = location
        self.status = status

    def activate(self):
        self.status = "active"
        print(f"Infrastructure at {self.location} is now active.")

    def deactivate(self):
        self.status = "inactive"
        print(f"Infrastructure at {self.location} is now inactive.")

    def show_details(self):
        print(f"Location: {self.location}")
        print(f"Status: {self.status}")


class TrafficLight(Infrastructure):
    def __init__(self, location, intersection_id):
        super().__init__(location)
        self.intersection_id = intersection_id

    def change_light(self, color):
        print(f"Traffic light at intersection {self.intersection_id} is now {color}.")

    def activate(self):
        super().activate()
        print(f"Traffic light at intersection {self.intersection_id} is now active.")

    def deactivate(self):
        super().deactivate()
        print(f"Traffic light at intersection {self.intersection_id} is now inactive.")

    def show_details(self):
        super().show_details()
        print(f"Intersection ID: {self.intersection_id}")


class StreetLight(Infrastructure):
    def __init__(self, location, lumens):
        super().__init__(location)
        self.lumens = lumens

    def adjust_brightness(self, brightness):
        print(f"Street light at {self.location} is now set to {brightness} lumens.")

    def activate(self):
        super().activate()
        print(f"Street light at {self.location} is now active.")

    def deactivate(self):
        super().deactivate()
```