DCIT 201: PROGRAMMING I
ASSIGNMENT 3

INSTRUCTIONS: Answer one (1) question from each section by writing the appropriate Python code.

ENCAPSULATION

QUESTION 1.

You are required to create a `CommissionEmployee` class in Python to represent an employee who earns compensation based on a percentage of their total sales. The class must enforce encapsulation principles by restricting direct access to sensitive attributes.

Class

`CommissionEmployee`

Class Attributes

`first_name` (str): The employee's first name.
`last_name` (str): The employee's last name.
`social_security_number` (str): The employee's social security number.
`gross_sales` (float): The employee's total gross sales (must be ≥ 0.0).
`commission_rate` (float): The percentage of gross sales paid as commission (must be between 0.0 and 1.0).

Methods

Constructor (`__init__`) – Initializes all attributes .

Getters and Setters – Update and retrieve class attributees

`earnings()` - Returns the employee's earnings using the formula: `grossSales` * `commissionRate`

Implementation Tasks

A. Implement the `CommissionEmployee` class in Python with proper encapsulation.
B. Create an instance of the `CommissionEmployee` class.
C. Update the employee's `grossSales` and `commissionRate`, then display the updated details.
D.. Calculate and display the employee's earnings using the `earnings()` method.

QUESTION 2.

You are required to implement a Library Management System in Python using encapsulation principles. The system should manage books and library members, allowing members to borrow and return books while tracking book availability.

Class 1

`Book Class`

Class Attributes

`book_id` (str): Unique identifier for the book.

`title` (str): Title of the book.

`author` (str): Author of the book.

`_available_copies` (int): Number of available copies .

Methods

Constructor (`__init__`) – Initializes all attributes.

Getters and Setters – Retrieve and update attributes .

`borrow_book()` – Reduces `_available_copies` by 1 if a copy is available.

`return_book()` – Increases `_available_copies` by 1.

Class 2

`Member Class`

Class Attributes

`member_id` (str): Unique identifier for the member.

`name` (str): Name of the member.

`_borrowed_book` (`Book` or `None`): Tracks the borrowed book .

-

Methods

Constructor (`__init__`) – Initializes `member_id` and `name`.

Getters and Setters – Retrieve and update attributes .

`borrow_book(book)` – Allows borrowing if the member has no book and the book is available.

`return_book()` – Returns the borrowed book and updates availability.

Implementation Tasks

A.Implement the `Book` and `Member` classes in Python with proper encapsulation
B. Creates a book and a member.
C. Simulates borrowing and returning the book.

QUESTION 3.

You are required to implement a **Hospital Management System** in Python using **encapsulation principles**. The system should securely manage patient and doctor details while ensuring proper validation.

Class 1

`Patient Class`

Class Attributes:

`patient_id` (str): Unique identifier for the patient.
`name` (str): Name of the patient.
`_age` (int): Age of the patient (private attribute).
`_diagnosis` (str): Current diagnosis of the patient (private attribute).

Methods

Constructor (`__init__`) – Initializes all attributes .

Getters and Setters – Retrieve and update attributes while ensuring data integrity.

`set_age(age)` – Ensures age is greater than 0; otherwise, prints `"Invalid age."`

`set_diagnosis(diagnosis)` – Ensures diagnosis is not empty; otherwise, prints `"Diagnosis cannot be empty."`

Class 2

`Doctor Class`

Class Attributes

`doctor_id` (str): Unique identifier for the doctor.

`name` (str): Name of the doctor.

`specialization` (str): Doctor's field of specialization.

Methods

Constructor (`__init__`) – Initializes `doctor_id`, `name`, and `specialization`.

Getters and Setters – Retrieve and update attributes with proper validation.

`treat_patient(patient)` – Logs `patient_id` and `diagnosis`, then prints `"Patient <patient_id> treated for <diagnosis> successfully."`

Implementation Tasks

A. Implement the `Patient` and `Doctor` classes in Python with encapsulation.

B. Create a `Patient` object with the following details:

Patient ID: `"P001"`

Name: `"John Smith"`

Age: `45`

Diagnosis: `"Fever"`

C. Creates a `Doctor` object with the following details:

Doctor ID: `"D101"`

Name: `"Dr. Alice"`

Specialization: `"General Medicine"`

D. Performs the following operations:

Set the patient's diagnosis to "Flu".

Treat the patient and log the treated patient info.

QUESTION 4

You need to design an **Airline Reservation System** in Python using **encapsulation** to securely manage **flight details, passenger information, and reservation operations**..

Class 1
`Flight Class`

Class Attributes:
`flight_number` (str): Unique identifier for the flight.

`destination` (str): Flight destination.

`_capacity` (int): Total number of seats (private attribute).

`_booked_seats` (int): Seats currently booked (private attribute).

Methods

Constructor (`__init__`) – Initializes all attributes.

Getters and Setters – Retrieve and update attribute values with validation.

`set_capacity(capacity)` – Ensures capacity is greater than or equal to `booked_seats`; otherwise, prints `"Invalid capacity: must be at least the number of booked seats."`

`book_seat()` – Increases `_booked_seats` by 1 if seats are available; otherwise, prints `"No available seats."`

`cancel_seat()` – Decreases `_booked_seats` by 1 if at least one seat is booked; otherwise, prints `"No bookings to cancel."`

Class 2

`Passenger Class`

Class Attributes:

`passenger_id` (str): Unique identifier for the passenger.
`name` (str): Passenger's name.
`_contact_number` (str): Passenger's contact number (private attribute).
`_flight_booked` (str or None): Flight number of the booked flight (initially `None`).

Methods

Constructor (`__init__`) – Initializes all attributes.

Getters and Setters – Retrieve and update attributes with validation.

`set_contact_number(contact_number)` – Ensures contact number is exactly 10 digits; otherwise, prints `"Invalid contact number. Must be 10 digits."`

`book_flight(flight_number)` – Assigns `_flight_booked` to `flight_number` if not already booked; otherwise, prints `"Passenger already has a booked flight."`

`cancel_flight()` – Sets `_flight_booked` to `None` if a booking exists; otherwise, prints `"No booking exists to cancel."`

Implementation Tasks

A. Implement the `Flight` and `Passenger` classes in Python using encapsulation.
B. Creates a Flight object with
Flight Number: `"AI101"`
Destination: `"New York"`
Capacity: `200`
Booked Seats: `150`

C. Creates a Passenger object with
Passenger ID: `"P123"`
Name: `"Sarah Connor"`
Contact Number: `"9876543210"`

D. Performs the following operations
Book a seat for the passenger and update flight details.

Attempt to book again (**should not allow duplicate booking**).

Cancel the booking and update flight details.

Attempt to cancel again (**should not allow cancellation if no booking exists**).

Attempt to set an **invalid flight capacity** (**less than booked seats**).

Attempt to set an **invalid contact number** (**not 10 digits**).

QUESTION 5

You need to design a Banking System in Python using encapsulation principles. The system should securely manage customer accounts, transactions, and financial analytics.

Class 1

`BankAccount Class`

Class Attributes:

`_account_number (str)`: The account number of the bank account.

`_account_holder (str)`: The name of the account holder.
`_balance (float)`: The current balance in the account (must be ≥ 0.0).
`_interest_rate (float)`: The annual interest rate for the account (must be between 0.0 and 1.0).

Methods

Constructor: Initializes all attributes while ensuring valid balance and interest rate.
deposit(amount: float): Adds the specified amount to the balance.
withdraw(amount: float): Deducts the specified amount from the balance (if sufficient funds are available).
calculate_interest(): Returns the annual interest earned using the formula: `_balance * _interest_rate`.
get_balance(): Returns the current balance.

Implementation Tasks

A. Implement the BankAccount class with encapsulation in Python.
B. Create an instance of the BankAccount class.
C. Update the account balance and display updated details
D. Calculate and display annual interest earned.

INHERITANCE

QUESTION 1

Extend a CommissionEmployee class into a subclass called BasePlusCommissionEmployee. The system should manage employees who earn based on commission, and those who have a base salary in addition to commissions.

Base Class

`CommissionEmployee`

Class Attributes

`first_name (str)`: The employee's first name.
`last_name (str)`: The employee's last name.
`social_security_number (str)`: A unique identifier for the employee.
`gross_sales (float)`: The employee's total sales amount.
`commission_rate (float)`: The commission percentage (**between 0 and 1**).

Methods

`__init__`: Initializes all attributes.
`earnings()`: Returns commission earnings `(gross_sales * commission_rate)`.
`display_employee_details()`: Prints employee details and their earnings.

Derived Class

`BasePlusCommission`

Class Attributes

Inherits all fields from `CommissionEmployee`.
`_base_salary (float)`: A guaranteed base salary for the employee.

Methods

`__init__`: Calls the superclass constructor, initializes inherited attributes and `_base_salary`.
`earnings()`: Returns total earnings as `_base_salary + (_gross_sales * _commission_rate)`.
`set_base_salary(new_salary)`: Updates the base salary with validation (must be ≥ 0).

Implementation Taaks

A. Create an instance of Commission-Only Employees in Python
B. Create an instance of BasePlusCommission Employees
C. Calculate and Display Earnings on each employee
D. Update `baseSalary` for a `BasePlusCommissionEmployee` instance and print the update earnings.

QUESTION 2

You are tasked with developing a **Vehicle Rental Management System** using **inheritance** in Python. The system should manage different types of vehicles.

Base Class

`Vehicle`

Class Attributes

`_vehicle_id (str)`: Unique vehicle identifier.
`_brand (str)`: Brand name of the vehicle.
`_model (str)`: Model name.
`_is_available (bool)`: Vehicle availability status (**default: True**).

Methods

`__init__()`: Initializes all attributes and sets `_is_available = True`.
`rent_vehicle()`: If available, marks the vehicle as rented; otherwise, displays an error message.
`return_vehicle()`: Marks the vehicle as available and confirms the return.

Derived Class

`Car`

Class Attributes

`_seating_capacity (int)`: Number of seats in the car.

Constructor

Initializes all inherited attributes and `_seating_capacity`.

Methods

`calculate_rental_cost(days: int) -> float`: Calculates rental cost using the formula `1000 * days + _seating_capacity * 50.` Prints "Rental cost for <days> days: <calculated amount>"

Implementation Tasks

A. Create a Car Instance
B. Rent and return a vehicle
C. Calculate Rental Cost

QUESTION 3

You are tasked with designing an **E-Commerce System** to manage different types of users and orders using inheritance principles.

Base Class

`User`

Class Attributes

`_user_id (str)`: Unique identifier for the user.
`_name (str)`: User's full name.

Methods

`__init__()`: Initializes `_user_id` and `_name`.
`print_user_details()`: Displays the user's ID and name.

Derived Class

`Customer -  Extends User Class`

Class Attributes

`_email (str)`: Customer's email address.

`_cart (list[str])`: A list of items added to the cart.

Methods

`__init__()`: Calls the parent constructor and initializes `email` and `cart`.

`add_item_to_cart(item: str) : Adds an item to the cart.`

`view_cart() : Displays all cart items.`

Derived Class

`Order - Extends Customer Class`

Class Attributes

`_order_id (str): Unique order identifier.`

`_order_details (list[str]): List of ordered items.`

Methods

`__init__()`: Calls the parent constructor and initializes `order_id`.

`print_order_details() -> None`: Displays the order ID, customer details, and ordered items.

Implementation Tasks

A. Create Users:

```
Customer("C001", "Alice", "alice@example.com")
Customer("C002", "Bob", "bob@example.com")
```

B.Customers Add Items to Cart:

```
Alice: "Laptop", "Mouse"
Bob: "Smartphone", "Headphones"
```

C. Place Orders:

Alice places an order with **her cart items.**

Bob places an order with **his cart items.**

E. View Users and Orders:

Print details using `print_user_details()` and `print_order_details()`

QUESTION 4

You are tasked with designing a **Hospital Management System** to manage different types of staff and their roles using inheritance principles.

Base Class

`Staff`

Class Attributes

`staff_id (str)`: Unique identifier for each staff member.

`name (str)`: Name of the staff member.

`department (str)`: Department where the staff member works.

Constructor

Initializes `staff_id`, `name`, and `department`.

Methods

`display_details()`: Prints "Staff ID: <staff_id>, Name: <name>, Department: <department>"

Derived Class

`Doctor` - Inherits Staff, representing a doctor with a specialization.

Class Attributes

Inherits `staff_id`, `name`, and `department` from `Staff`.

`specialization (str)`: Doctor's area of expertise.

`years_of_experience (int)`: Number of years the doctor has practiced.

Constructor

Calls the superclass constructor to initialize inherited attributes.

Initializes `specialization` and `years_of_experience`

Methods

`display_details()`: Prints

"Doctor ID: <staff_id>, Name: <name>, Department: <department>, Specialization: <specialization>, Experience: <years_of_experience> years"

Derived Class

`Nurse` - Inherits `Staff`, representing a nurse with shift details..

Class Attributes

Inherits `staff_id`, `name`, and `department` from `Staff`.

`shift (str)`: Assigned shift (e.g., "Day", "Night").

`patients_assigned (int)`: Number of patients under care

Constructor

Calls the superclass constructor to initialize inherited attributes.

Initializes `shift` and `patients_assigned`

Methods

`display_details()`: Prints

"Nurse ID: <staff_id>, Name: <name>, Department: <department>, Shift: <shift>, Patients Assigned: <patients_assigned>"

Independent Class

`HospitalManagementSystem`

Methods

`register_doctor(doctor: Doctor)`: Calls `display_details()` on the `Doctor` instance.

`register_nurse(nurse: Nurse)`: Calls `display_details()` on the `Nurse` instance.

Implementation Tasks

A.Create Staff Members:

```
Doctor("S001", "Dr. Smith", "Cardiology", "Cardiology", 15")
Doctor("S002", "Dr. Lee", "Neurology", "Neurology", 8")
Nurse("S003", "Nurse Kelly", "Emergency", "Night", 5")
```

B. Register and Display Staff Details:

Call `register_doctor()` for each doctor.

Call `register_nurse()` for the nurse

QUESTION 5

You are tasked with designing a **Restaurant Management System** to manage various staff roles using inheritance principles. This system will focus on role-specific responsibilities and task delegation.

Base Class

`Employee - Represents a general restaurant employee.`

Class Attributes

`employee_id (str)`: Unique identifier for the employee.
`name (str)`: Name of the employee.

Constructor

Initializes `employee_id` and `name`

Methods

`display_details()`: Prints "Employee ID: <employee_id>, Name: <name>"

Derive Class

`Chef - Represents a chef with additional details.`

Class Attributes

Inherits `employee_id` and `name` from `Employee`.
`specialty (str)`: Type of cuisine the chef specializes in.

Constructor

Calls the superclass constructor to initialize inherited attributes.
Initializes `specialty`.

Methods

`display_details()`: Prints
"Chef ID: <employee_id>, Name: <name>, Specialty: <specialty>"
`prepare_dishes()`: Prints
"Chef <name> is preparing <specialty> dishes."

Derive Class

`Waiter` - Represents a Waiter with additional details.

Class Attributes

Inherits `employee_id` and `name` from `Employee`.

`assigned_section (str)`: Section of the restaurant assigned to the waiter.

Constructor

Calls the superclass constructor to initialize inherited attributes.

Initializes `assigned_section`

Methods

`display_details()`: Prints

"Waiter ID: <employee_id>, Name: <name>, Section: <assigned_section>"

`serve_customers()`: Prints

"Waiter <name> is serving customers in the <assigned_section> section."

Independent Class

`RestaurantManagementSystem` - Handles employee tasks.

Methods

`assign_chef_task(chef: Chef)`: Calls `prepare_dishes()` on the `Chef` instance.

`assign_waiter_task(waiter: Waiter)`: Calls `serve_customers()` on the `Waiter` instance.

Implementation Tasks

A. Create Employees:

```
Chef("E001", "Alice", "Italian")
Waiter("E002", "Bob", "Outdoor")
```

B. Assign Tasks:
Call `assign_chef_task()` for the chef.
Call `assign_waiter_task()` for the waiter.

POLYMORPHISM

QUESTION 1

You are tasked with designing a **Transportation Management System** to handle various types of vehicles and their operations using polymorphism. The system must demonstrate both **runtime polymorphism** (method overriding) and **compile-time polymorphism** (method overloading).

Abstract Class
`Vehicle`

Class Attributes

`vehicle_id (str)`: Unique identifier for the vehicle.

`model (str)`: Model name of the vehicle.

`fuel_level (float)`: Current fuel level in liters.

Constructor

Calls the superclass constructor to initialize inherited attributes.
Initializes `vehicle_id`, `model`, and `fuel_level`.

Methods

`refuel(liters: float)`: Adds fuel and prints the updated fuel level.

`calculate_range()`: **Abstract method** to be implemented in subclasses.

Derive Class

`Car - Extends Vehicle`

Class Attributes

Inherits `vehicle_id`, `model`, and `fuel_level` from `Vehicle`.

`fuel_efficiency (float)`: Kilometers per liter.

Constructor

Initializes `fuel_efficiency`.

Methods

Overrides `calculate_Range():` range = fuelLevel * fuelEfficiency. Prints "Car <model> can travel <range> km with current fuel level."

Independent Class

`TransportationManager`

Method

`operate_vehicle(vehicle: Vehicle)`: Calls `calculate_range()` on any `Vehicle` instance, demonstrating runtime polymorphism.

Implementation Tasks

A. Create Vehicles:

`Car("C001", "Sedan", 50, 15)`

B. Refuel Sedan using `refuel()`

C. Use `operate_Vehicle()` to process all

QUESTION 2

You are tasked with designing a **Banking System** that demonstrates **both compile-time (method overloading)** and **run-time polymorphism (method overriding)**. The system should handle different types of accounts and operations.

Base Class

`BankAccount`

Class Attributes

`account_Holder_Name` (str) – Name of the account holder

`account_Number` (str) – Unique account number

`balance` (float) – Current account balance

Constructor

Initializes all attributes

Methods

`deposit(amount: float)`: Increases balance and prints the updated balance.

`deposit(amount: float, note: str)`: Overloaded method that also prints the transaction note.

`withdraw(amount: float)`: Decreases balance if funds are available, else prints an error.

`display_Account_Details()`: Prints account details (overridden in subclasses).

Derive Class

`savingsAccount - Extends Bank Account`

Class Attributes

`interest_Rate` (float) – Annual interest rate

Constructor

Calls the superclass constructor to initialize inherited attributes.

Intializes `interest_Rate`

Methods

Overrides `withdraw(amount:float)`: Prevents withdrawal if balance falls below $100.

`calculate_Interest()`: Computes annual interest and displays it.

Overrides `display_Account_Details()`: Includes `interest_Rate` in account details

Implementation Tasks

A. Create Accounts:

`SavingsAccount("Alice", "SA123", 500, 3%)`

B. Deposit to savings accounts using one and two arguments.

C. Withdraw from `SavingsAccount`, testing the minimum balance limit.

D. Display account details for both.

QUESTION 3

You are tasked with designing an **E-Commerce System** to handle various types of products and dynamic pricing using polymorphism.

Abstract Class
`Product`

Class Attributes

`product_Id` (str) – Unique product ID

`product_Name` (str) – Name of the product

`base_Price` (float) – Original price of the product

Constructor

Intializes all attributes

Methods

`apply_Discount(percentage: float)`: Reduces price by a given percentage.

`calculate_Final_Price()`: Abstract method for product-specific price calculations.

Derive Class
`Electronics`

Class Attributes
`warranty_period (int)`: Warranty duration in months.

Constructor

Calls the superclass constructor to initialize inherited attributes.

Intializes `warranty_period`

Methods

Overrides `calculate_final_price()`: Computes final price after warranty-based adjustments.

Derive Class

`Clothing`

Class Attributes

`size (str)`: Size of the clothing item.

`fabric_charge (float)`: Additional cost based on fabric quality.

Constructor

Calls the superclass constructor to initialize inherited attributes.

Intializes `size` and `fabric_charge`

Methods

Overrides `calculate_final_price()`: Computes final price by adding fabric charges.

Derive Class

`Cart`

Methods

`add_Product(product: Product)`: Adds a product to the cart.

`calculateTotalPrice(...products)`: Overloaded method to calculate total cost for multiple products.

Implementation Tasks

A. Create Products

```
Electronics("E001", "Laptop", 1000.0, 24)
Clothing("C001", "Winter Jacket", 200.0, "M", 20.0)
```

B. Apply 10% discount to Laptop

C. Calculate final price of Laptop and Winter Jacket.

D. Add both to Cart and calculate total price

QUESTION 4

You are tasked with designing a **Staff Management System** for an organization. The system must demonstrate **polymorphism** to calculate staff Annual salary

Abstract Class

`StaffMember`

Class Attributes

`name` (str) – Staff member's name

`id` (str) – Unique identifier

Constructor

Initializes all attributes

Methods

`get_Annual_Salary()`: Abstract method to calculate annual pay.

`toString()`: Returns staff details.

Derive Class
`Staff`

Constructor

Calls the superclass constructor to initialize inherited attributes.

Methods

`add_Staff(staff: StaffMember)`: Adds a staff member.

`get_Annual_Salary(monthly_Salary: int)`: Computes total monthly salary.

`display_Staff()`: Prints staff details.

Implementation Tasks
A. Create a staff Member
B. Display Staff Details
C. Calculate total Annual Salary

QUESTION 5

You are tasked with designing a Church Management System that demonstrates method overriding (runtime polymorphism) and method overloading (compile-time polymorphism).

Abstract Class
`StaffMember`

Class Attributes

`name` (str): Member's name.

`member_Id` (str): Unique identifier.

Constructor

Initializes `name` and `member_Id`.

Methods

`get_Contribution()`: Returns `0.0` (default for general members).

`give_Offering( amount: double)`: Prints `"Offering given: <amount>"`.

Derive Class

`Pastor - Inherits ChurchMember`

Class Attributes

`tithe` (double): Monthly tithe contribution.

Constructor

Initializes `name`, `member_Id`, and `tithe`

Methods

Override `get_Contribution()`: Returns `tithe`.

Overload `give_Offering(amount: double, message: str)`: Prints `"Offering given: <amount>. Note: <message>"`.

Implementation Tasks

A. Create Staff Members

```
StaffMember("John Doe", "M001") for a general memnber
Pastor("Rev. Smith", "P001", 500.0)
```

B. Check Contributions : `get_contribution()` for both members.

C. Give Offerings:

General member calls `give_offering(100.0)`.

Pastor calls `give_offering(200.0, "For the church renovation")`.

ABSTRACTION

QUESTION 1

You are tasked with implementing **abstraction** in a payroll system using Python. The system should define an abstract `Employee` class as a base for different types of employees.

Abstract Class

`Employee`

Encapsulated Class Attributes

`_name (str)`: Employee's name.

`_employee_id (str)`: Unique identifier.

Constructor

Initializes `_name` and `_employee_id`.

Methods

Getter methods: Provide access to `_name` and `_employee_id`.

Abstract method `calculate_pay()`: Must be implemented by subclasses.

Derive Class

`FullTimeEmployee - Extends Employee`

Encapsulated Class Attributes

`_salary (float)`: The full-time employee's salary.

Constructor

Initializes `_name`, `_employee_id`, and `_salary`.

Methods

`get_salary()`: Returns `_salary`

Implement `calculate_pay()` to return: `"FullTimeEmployee Pay: <salary>"`.

Implementation Tasks

A. Create the abstract `Employee` class with the specified attributes and methods in Python

B. Implement the `FullTimeEmployee` subclass and define `calculate_pay()`.

C. Instantiate a `FullTimeEmployee` object.

D. Display the employee's details using the getter methods.

QUESTION 2

You are tasked with designing a **Medical Record Management System** using Python. The system should enforce **abstraction** by defining a base class for different types of medical personnel while allowing specific roles and specializations to vary.

Abstract Class

`MedicalPersonnel`

Encapsulated Class Attributes

`_name (str)`: The name of the medical personnel.

`_id (str)`: A unique identifier for the personnel.

Constructor

Initializes `_name` and `_id`.

Methods

Getter methods: Provide access to `_name` and `_id`.

`perform_duties()`: Defines the duties of medical personnel (must be implemented in subclasses).

`get_specialization()`: Defines the specialization of the personnel (must be implemented in subclasses).

`display_details()`: Prints the name and ID of the personnel.

.

Derive Class

`Doctor - Extends MedicalPersonnel`

Class Attributes

`_specialization (str)`: The medical specialty (e.g., Cardiologist, Pediatrician).

Constructor

Initializes `_name`, `_id`, and `_specialization`.

Methods

Implement `perform_duties()` to return:

"Doctor <name>: Diagnoses patients, prescribes medication, and conducts surgeries."

Implement `get_specialization()` to return `_specialization`.

Derive Class

`Nurse - Extends MedicalPersonnel`

Class Attributes

`_department` (str): The department the nurse works in (e.g., ICU, Emergency).

Constructor

Initializes `_name`, `_id`, and `_department`.

Methods

Implement `perform_duties()` to return:

"Nurse <name>: Provides patient care, administers medications, and assists doctors."

Implement `get_specialization()` to return `_department`

Implementation Tasks

A. Create the `MedicalPersonnel` abstract class with the required methods.

B. Implement the `Doctor` subclass, ensuring they define `perform_duties()` and `get_specialization()`.

C. Write statements to:

Create a list of at least one `Doctor`, one `Nurse`.

Use a loop to:

Call `display_details()` for each object.

Call `perform_duties()` for each object.

Call `get_specialization()` for each object.

QUESTION 3

You are tasked with designing a **Device Management System** for a tech company that manages different types of devices used by employees. The system must use **abstraction** to provide a blueprint for handling various device operations while allowing specific implementations for different device types.

Abstract Class

`Device`

Class Attributes

`_device_id (str)`: Unique identifier.

`_brand (str)`: Device brand.

`_model (str)`: Device model.

Constructor

Initializes `_device_id`, `_brand`, and `_model`

Abstract Methods

`calculate_power_consumption()`: Computes power consumption in kWh.

`calculate_maintenance_cost()`: Computes yearly maintenance cost.

`getDetails()`: Returns device details.

Derive Class

`Laptop - Extends Device`

Class Attributes

`_processor_power (float)`: Processor power in watts.

`_daily_usage_hours (float)`: Daily usage in hours.

`_maintenance_cost_per_year (float)`: Fixed yearly maintenance cost.

Constructor

Initializes all class fields including inherited ones

Methods

Override `calculate_power_consumption()`: `(processorPower * dailyUsageHours * 365) / 1000`

Override `calculate_maintenance_cost()`: returns `maintenanceCostPerYear`

Override `getDetails()`: Returns device details.

Implementation Tasks

A. Create a Laptop instance

B. Display Device details.

C. Display Power consumption.

D. Display Maintenance cost.

QUESTION 4

You are tasked with creating a **2D Shape Management System** for a design application that allows users to manage, resize, and render various 2D shapes. Use **abstraction** to define the core operations for all shapes and provide specific implementations for different shape types.

Interface

`shape2D`

Methods:

`draw()`: Displays the shape's details.

`resize(factor: float)`: Scales the shape by a given factor.

`move(delta_x: float, delta_y: float)`: Adjusts the shape's position

Class

`Rectangle - Implements 2D interface`

Class Attributes

`color: str` → Defines the rectangle's color.

`position_x: float` → X-coordinate of the rectangle.

`position_y: float` → Y-coordinate of the rectangle.

`width: float` → Width of the rectangle.

`height: float` → Height of the rectangle.

Constructor:
Initializes all attributes.

Methods:
`draw()`: Prints the rectangle's color, position, width, and height.
`resize(factor: float)`: Scales `width` and `height` by `factor`.
`move(delta_x: float, delta_y: float)`: Updates position based on the given deltas.

Class
`shapeManager`

Class Atrributes
`Shape: shape2D`: Stores `Shape2D` object.

Methods
`add_shape(shape: Shape2D)`: Adds a new shape to the manager.

`draw_shape()`: Calls `draw()` on all stored shapes.

`resize_shape(factor: float)`: Resizes all shapes by the given factor.

`move_shape(delta_x: float, delta_y: float)`: Moves all shapes by the given deltas.

Implementation Tasks

A. Create Shapes:

Rectangle: Red, `(2,3)`, `5,10`.

B. Add to ShapeManager and perform:

- Draw the rectangle.
- Resize rectangle by `2.0x`.
- Move rectangle `(-1.0,1.0)`, then draw again.

QUESTION 5

Design a University Management System in Python for managing departments, hostels, and students. The system should use interfaces for abstraction

Interface

`Department`

Attributes

`dept_name: str` → Name of the department.

`dept_head: str` → Head of the department.

Methods:

`print_department_details()`: Displays department details.

Class

`Student - Implements department`

Class Atrributes

`student_name: str` → Student's full name.

`regd_no: str` → Student's registration number.

`elective_subject: str` → Elective subject of the student.

`avg_marks: float` → Student's average marks.

`hostel_name: str` → Name of the assigned hostel.

`hostel_location: str` → Location of the hostel.

`number_of_rooms: int` → Number of rooms in the hostel.

Methods

`set_student_details()`: Inputs and assigns student details, including department and hostel.

`get_student_details()`: return all student information.

Implements `print_department_details()`: Prints department details.

`migrate_hostel(new_hostel: str, new_location: str, new_rooms: int)`: Updates the student's hostel details.

Class

`UniversityManager`

Class Atrributes

`student_record: Student` → Stores a single student's information at a time.

Methods

`admit_student(student: Student)`: Assigns the given student as the current record.

`display_student_details()`: Prints the details of the stored student.

`update_hostel(new_hostel: str, new_location: str, new_rooms: int)`: Modifies the hostel details of the stored student.

Implementation Tasks

A. Define Interface for Departments

B. Create a `Student` class that implements `Department`.

C. Create a `UniversityManager` class to handle student-related operations.

D. Admit a new student by providing all necessary details.

E. Migrate a student by updating their hostel details.

F. Display student details using `displayStudentDetails()`