# 07_Oct_Python oops assignment

October 8, 2023

[ ]:
```python
#Problem 1: Bank Account
#Question 1:
#Create a class representing a bank account with attributes like account␣
 ↪number, account holder name, and balance. Implement methods to deposit and␣
 ↪withdraw money from the account.


class BankAccount:
    def __init__(self, account_number, account_holder_name, balance=0):
        self.account_number = account_number
        self.account_holder_name = account_holder_name
        self.balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            return f"Deposited ${amount:.2f}. Current balance: ${self.balance:.
 ↪2f}"
        else:
            return "Deposit amount must be greater than zero."

    def withdraw(self, amount):
        if amount > 0:
            if amount <= self.balance:
                self.balance -= amount
                return f"Withdrew ${amount:.2f}. Current balance: ${self.
 ↪balance:.2f}"
            else:
                return "Insufficient funds for withdrawal."
        else:
            return "Withdrawal amount must be greater than zero."

    def get_balance(self):
        return f"Account balance for {self.account_holder_name}: ${self.balance:
 ↪.2f}"

    def __str__(self):
```

```
        return f"Account Number: {self.account_number}\nAccount Holder: {self.
 ↪account_holder_name}\nBalance: ${self.balance:.2f}"

# Example usage:
if __name__ == "__main__":
    # Create a bank account
    account1 = BankAccount("12345", "John Doe", 1000.00)

    # Deposit and withdraw money
    print(account1.deposit(500.50))
    print(account1.withdraw(200.75))
    print(account1.withdraw(1500.00))  # Should show "Insufficient funds for␣
 ↪withdrawal."

    # Get the current balance
    print(account1.get_balance())

    # Display account information
    print("\nAccount Information:")
    print(account1)

#Problem 2: Employee Management
#Question 2:
#Create a class representing an employee with attributes like employee ID,␣
 ↪name, and salary. Implement methods to calculate the yearly bonus and␣
 ↪display employee details.


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

    def calculate_yearly_bonus(self, bonus_percentage):
        bonus = (bonus_percentage / 100) * self.salary
        return f"Yearly Bonus: ${bonus:.2f}"

    def display_employee_details(self):
        return f"Employee ID: {self.employee_id}\nName: {self.name}\nSalary:␣
 ↪${self.salary:.2f}"

# Example usage:
if __name__ == "__main__":
    employee1 = Employee("001", "Alice Smith", 50000.00)
    print(employee1.calculate_yearly_bonus(10))
    print(employee1.display_employee_details())
```

```python
#Problem 3: Vehicle Rental

#Question 3:
#Create a class representing a vehicle rental system. Implement methods to rent a vehicle, return a vehicle, and display available vehicles.


class VehicleRental:
    def __init__(self):
        self.available_vehicles = []

    def rent_vehicle(self, vehicle):
        if vehicle in self.available_vehicles:
            self.available_vehicles.remove(vehicle)
            return f"Rented {vehicle}"
        else:
            return f"{vehicle} is not available for rent."

    def return_vehicle(self, vehicle):
        self.available_vehicles.append(vehicle)
        return f"Returned {vehicle}"

    def display_available_vehicles(self):
        return "Available Vehicles: " + ", ".join(self.available_vehicles)

# Example usage:
if __name__ == "__main__":
    rental_system = VehicleRental()
    rental_system.available_vehicles = ["Car", "Truck", "Motorcycle"]
    print(rental_system.rent_vehicle("Car"))
    print(rental_system.display_available_vehicles())

#Problem 4: Library Catalog
#Question 4:
#Create classes representing a library and a book. Implement methods to add books to the library, borrow books, and display available books.


class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.available = True

class LibraryCatalog:
    def __init__(self):
```

```
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def borrow_book(self, title):
        for book in self.books:
            if book.title == title and book.available:
                book.available = False
                return f"Borrowed {book.title}"
        return f"Book '{title}' is not available for borrowing."

    def display_available_books(self):
        available_books = [book.title for book in self.books if book.available]
        return "Available Books: " + ", ".join(available_books)

# Example usage:
if __name__ == "__main__":
    book1 = Book("The Great Gatsby", "F. Scott Fitzgerald")
    book2 = Book("To Kill a Mockingbird", "Harper Lee")
    library = LibraryCatalog()
    library.add_book(book1)
    library.add_book(book2)
    print(library.borrow_book("To Kill a Mockingbird"))
    print(library.display_available_books())


#Problem 5: Product Inventory
#Question 5:
#Create classes representing a product and an inventory system. Implement␣
 ↪methods to add products to the inventory, update product quantity, and␣
 ↪display available products.


class Product:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity

class InventorySystem:
    def __init__(self):
        self.products = []

    def add_product(self, product):
        self.products.append(product)
```

```
    def update_product_quantity(self, product_name, new_quantity):
        for product in self.products:
            if product.name == product_name:
                product.quantity = new_quantity
                return f"Updated quantity of {product_name} to {new_quantity}"
        return f"Product '{product_name}' not found."

    def display_available_products(self):
        available_products = [f"{product.name} (${product.price:.2f}): {product.
 ↪quantity} available" for product in self.products]
        return "Available Products:\n" + "\n".join(available_products)

# Example usage:
if __name__ == "__main__":
    product1 = Product("Laptop", 800.00, 10)
    product2 = Product("Smartphone", 400.00, 20)
    inventory = InventorySystem()
    inventory.add_product(product1)
    inventory.add_product(product2)
    print(inventory.update_product_quantity("Laptop", 5))
    print(inventory.display_available_products())
```

[ ]:
```
#Problem 6: Shape Calculation

#Create a class representing a shape with attributes like length, width, and␣
 ↪height.
#Implement methods to calculate the area and perimeter of the shape.
class Shape:
    def __init__(self, length, width, height):
        self.length = length
        self.width = width
        self.height = height

    def calculate_area(self):
        raise NotImplementedError("Subclasses must implement this method")

    def calculate_perimeter(self):
        raise NotImplementedError("Subclasses must implement this method")

class Rectangle(Shape):
    def calculate_area(self):
        return self.length * self.width

    def calculate_perimeter(self):
        return 2 * (self.length + self.width)

class Triangle(Shape):
```

```
    def calculate_area(self):
        # Using Heron's formula to calculate the area of a triangle
        s = (self.length + self.width + self.height) / 2
        return (s * (s - self.length) * (s - self.width) * (s - self.height))␣
 ↪** 0.5

    def calculate_perimeter(self):
        return self.length + self.width + self.height

#Problem 7: Student Management

#Create a class representing a student with attributes like student ID, name,␣
 ↪and grades.
#Implement methods to calculate the average grade and display student details.
class Student:
    def __init__(self, student_id, name, grades):
        self.student_id = student_id
        self.name = name
        self.grades = grades

    def calculate_average_grade(self):
        if self.grades:
            return sum(self.grades) / len(self.grades)
        else:
            return 0.0

    def display_student_details(self):
        return f"Student ID: {self.student_id}\nName: {self.name}\nAverage␣
 ↪Grade: {self.calculate_average_grade():.2f}"

#Problem 8: Email Management

#Create a class representing an email with attributes like sender, recipient,␣
 ↪and subject.
#Implement methods to send an email and display email details.
class Email:
    def __init__(self, sender, recipient, subject, message):
        self.sender = sender
        self.recipient = recipient
        self.subject = subject
        self.message = message

    def send_email(self):
        return f"Email sent from {self.sender} to {self.recipient} with subject:
 ↪ {self.subject}"

    def display_email_details(self):
```

```
        return f"Sender: {self.sender}\nRecipient: {self.recipient}\nSubject:␣
 ↪{self.subject}\nMessage: {self.message}"

#Problem 9: Social Media Profile

#Create a class representing a social media profile with attributes like␣
 ↪username and posts.
#Implement methods to add posts, display posts, and search for posts by keyword.
class SocialMediaProfile:
    def __init__(self, username):
        self.username = username
        self.posts = []

    def add_post(self, post_content):
        self.posts.append(post_content)

    def display_posts(self):
        return "\n".join(self.posts)

    def search_posts_by_keyword(self, keyword):
        return [post for post in self.posts if keyword in post]

#Problem 10: ToDo List

#Create a class representing a ToDo list with attributes like tasks and due␣
 ↪dates.
#Implement methods to add tasks, mark tasks as completed, and display pending␣
 ↪tasks.
class ToDoList:
    def __init__(self):
        self.tasks = []

    def add_task(self, task, due_date):
        self.tasks.append({"task": task, "due_date": due_date, "completed":␣
 ↪False})

    def mark_task_as_completed(self, task):
        for t in self.tasks:
            if t["task"] == task:
                t["completed"] = True

    def display_pending_tasks(self):
        pending_tasks = [t for t in self.tasks if not t["completed"]]
        return "\n".join([f"Task: {t['task']}, Due Date: {t['due_date']}" for t␣
 ↪ in pending_tasks])
```