Learn to use OOP (Class and Object)

Task 1: Create a class BankAccount to model a bank account.

- **Attributes**:
  - Instance attributes: account_holder_name, account_number, and balance (initialize to 0).
  - Class attribute: bank_name (e.g., "ABC Bank").
- **Methods**:
  - Instance methods:
  - deposit(amount) - Adds the amount to the balance.
  - withdraw(amount) - Deducts the amount from the balance if sufficient funds are available. Otherwise, print an error message.
  - display_balance() - Prints the current balance.
  - Class methods:
  - change_bank_name(new_name) - Changes the class attribute bank_name.
- Use a @property for balance to restrict direct access and validate any modifications.

Reference Code

```python
class BankAccount:
    # Class attribute
    bank_name = "ABC Bank"

    def __init__(self, account_holder_name, account_number):
        # Instance attributes
        self.account_holder_name = account_holder_name
        self.account_number = account_number
        self._balance = 0  # Protected attribute for balance

    # Instance methods
    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            print(f"Deposited ${amount:.2f}. New balance: ${self._balance:.2f}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount <= self._balance:
            self._balance -= amount
            print(f"Withdrew ${amount:.2f}. Remaining balance: ${self._balance:.2f}")
        else:
            print("Insufficient funds!")
```

```
    def display_balance(self):
        print(f"Account Balance: ${self._balance:.2f}")

    # Class method
    @classmethod
    def change_bank_name(cls, new_name):
        cls.bank_name = new_name
        print(f"Bank name changed to: {cls.bank_name}")

    # Property for balance
    @property
    def balance(self):
        return self._balance
```

```
    @balance.setter
```

```
    def balance(self, value):
        if value < 0:
            raise ValueError("Balance cannot be negative!")
        self._balance = value
```

```
# Example Usage
if __name__ == "__main__":
    # Creating a bank account
    account1 = BankAccount("Alice", "123456")

    # Using instance methods
    account1.deposit(1000)
    account1.withdraw(500)
    account1.display_balance()

    # Using class method
    BankAccount.change_bank_name("XYZ Bank")
    print(f"Bank Name: {BankAccount.bank_name}")

    # Testing @property
    print(f"Current Balance: ${account1.balance}")
    account1.balance = 2000  # Updating balance using setter
    print(f"Updated Balance: ${account1.balance}")
```

Explanation of @property Usage

1. **Why Use @property for balance?**
    - Prevent direct modification of the _balance attribute to ensure data integrity.
    - Add validation when setting the balance (e.g., ensuring it isn't negative).
2. **When to Use @property?**
    - Use it whenever you want to control how an attribute is accessed or modified.
    - Example: Prevent users from setting invalid values directly or dynamically calculate a value based on other attributes.