**Harshavardhan**

**2403A51235**

**Batch-CS11**

**Task 1:**

Task 1: Auto-Complete a Python Class for Bank Account  
Task Description #1  
• Write a class definition comment and start the constructor for a class called BankAccount with  
account\_holder and balance attributes. Use GitHub Copilot to auto-complete the rest of the  
class, including methods to deposit, withdraw, and display balance.

**Code:**

# BankAccount class represents a simple bank account with basic operations.

# Attributes:

#   account\_holder (str): Name of the account holder

#   balance (float): Current account balance

# Methods:

#   deposit(amount): Adds amount to balance

#   withdraw(amount): Subtracts amount from balance if sufficient funds

#   display\_balance(): Prints the current balance

class BankAccount:

    def \_\_init\_\_(self, account\_holder, balance=0.0):

        self.account\_holder = account\_holder

        self.balance = balance

    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 > 0:

            if self.balance >= amount:

                self.balance -= amount

                print(f"Withdrew ${amount:.2f}. New balance: ${self.balance:.2f}")

            else:

                print("Insufficient funds.")

        else:

            print("Withdrawal amount must be positive.")

    def display\_balance(self):

        print(f"Account Holder: {self.account\_holder}")

        print(f"Current Balance: ${self.balance:.2f}")

# Take user input for account details and operations

name = input("Enter account holder name: ")

try:

    initial\_balance = float(input("Enter initial balance: "))

except ValueError:

    print("Invalid input. Setting initial balance to 0.")

    initial\_balance = 0.0

account = BankAccount(name, initial\_balance)

try:

    deposit\_amount = float(input("Enter amount to deposit: "))

    account.deposit(deposit\_amount)

except ValueError:

    print("Invalid deposit amount.")

try:

    withdraw\_amount = float(input("Enter amount to withdraw: "))

    account.withdraw(withdraw\_amount)

except ValueError:

    print("Invalid withdrawal amount.")

account.display\_balance()

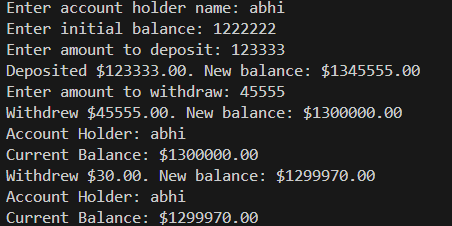
# Withdraw money

account.withdraw(30)

# Display balance

account.display\_balance()

**output:**

****

**Explanation:**

1. **Class Definition**
   * The class BankAccount is created to represent a bank account.
   * It has two **attributes**:
     + account\_holder → Name of the person holding the account.
     + balance → Current balance in the account (default is 0.0).
2. **Constructor (\_\_init\_\_)**
   * Runs when a new object is created.
   * Takes account\_holder name and an optional balance (if not given, it starts at 0).
3. **Deposit Method** (deposit)
   * Accepts an amount.
   * If amount is positive, it adds it to the balance and prints the new balance.
   * If not positive, it shows an error message.
4. **Withdraw Method** (withdraw)
   * Accepts an amount.
   * If amount is positive **and** enough balance exists, it deducts from balance.
   * If insufficient funds, it shows an error message.
   * If invalid amount (≤ 0), it rejects.
5. **Display Balance Method** (display\_balance)
   * Prints account holder name and current balance.
6. **User Input Section**
   * Asks the user for account holder name.
   * Asks for an initial balance (if invalid, sets it to 0).
   * Creates an account object (account = BankAccount(name, initial\_balance)).
7. **Deposit Operation**
   * Takes user input for deposit amount.
   * Calls account.deposit(deposit\_amount).
8. **Withdraw Operation**
   * Takes user input for withdrawal amount.
   * Calls account.withdraw(withdraw\_amount).
9. **Final Display**
   * Shows the account details with account.display\_balance().
10. **Extra Transactions at End**

* Withdraws 30 from the account (hardcoded).
* Shows updated balance again.

**Task 2:**

Task 2: Auto-Complete a For Loop to Sum Even Numbers in a List  
Task Description #2  
• Write a comment and the initial line of a loop to iterate over a list. Allow GitHub Copilot to  
complete the logic to sum all even numbers in the list.  
Expected Output #2  
• Code that:  
• Iterates over a list  
• Checks if the number is even using % 2 == 0  
• Accumulates the sum  
• Sample input/output

**Code:**

# Take input from user for how many numbers to sum

count = int(input("How many numbers do you want to sum? "))

numbers = []

for i in range(count):

    num = int(input(f"Enter number {i+1}: "))

    numbers.append(num)

even\_sum = 0

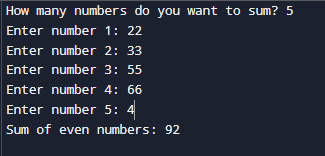
for num in numbers:

    if num % 2 == 0:

        even\_sum += num

print(f"Sum of even numbers: {even\_sum}")

**output:**

****

**Explanation:**

1. The program first asks the user **how many numbers** they want to input (count).
2. It then runs a **for loop** that takes count numbers from the user and stores them in a list called numbers.
3. A variable even\_sum is initialized to 0 to keep track of the sum of even numbers.
4. Another loop goes through each number in the list numbers.
5. Inside the loop, the program checks if the number is even (num % 2 == 0).
6. If it is even, the number is **added to even\_sum**.
7. After all numbers are checked, the program prints the final sum of even numbers.

**Task 3:**

Task 3: Auto-Complete Conditional Logic to Check Age Group  
Task Description #3  
• Start a function that takes age as input and returns whether the person is a child, teenager,  
adult, or senior using if-elif-else. Use Copilot to complete the conditionals.  
Expected Output #3  
• Function like:  
Output for age\_group(45) ➝ "Adult"

**Code :**

def age\_group(age):

    if age < 13:

        return "Child"

    elif age < 20:

        return "Teenager"

    elif age < 60:

        return "Adult"

    else:

        return "Senior"

# Take age as input from user and print the age group

user\_age = int(input("Enter your age: "))

print(age\_group(user\_age))

**output:**

****

**Explanation:**

This Python code defines a function and then uses it to classify a person’s age into groups. Here’s what it does:

* The function age\_group(age) checks the given age and returns a category:
  + If age < 13 → returns "Child".
  + If age < 20 → returns "Teenager".
  + If age < 60 → returns "Adult".
  + Otherwise (60 and above) → returns "Senior".
* Then the program asks the user to **enter their age** using input().
* int(input(...)) converts the input string to a number.
* Finally, it calls age\_group(user\_age) and prints the result.

**Task 4:**

Task 4: Auto-Complete a While Loop to Reverse Digits of a Number  
Task Description #4  
• Write a comment and start a while loop to reverse the digits of a number. Let Copilot  
complete the loop logic.  
Expected Output #4  
• Functional loop: Output: 4321

**Code:**

# Reverse the digits of a number using a while loop

num = 1234

reversed\_num = 0

while num > 0:

    digit = num % 10

    reversed\_num = reversed\_num \* 10 + digit

    num //= 10

print(reversed\_num)  # Output: 4321

**output:**

**4321**

**Explaination:**

* num = 1234 → starting number.
* reversed\_num = 0 → a variable to store the reversed number.

The **while loop** runs as long as num > 0:

1. digit = num % 10 → extracts the last digit of num.
   * For 1234, digit = 4.
2. reversed\_num = reversed\_num \* 10 + digit → shifts the current reversed number one place left and adds the new digit.
   * Initially 0 \* 10 + 4 = 4.
3. num //= 10 → removes the last digit from num.
   * 1234 // 10 = 123.

This repeats:

* Next loop: digit = 3, reversed\_num = 43, num = 12.
* Next loop: digit = 2, reversed\_num = 432, num = 1.
* Next loop: digit = 1, reversed\_num = 4321, num = 0.

When num becomes 0, the loop ends.

Finally, print(reversed\_num) → **4321**

**Task 5:**

Task 5: Auto-Complete Class with Inheritance (Employee → Manager)  
Task Description #5  
• Begin a class Employee with attributes name and salary. Then, start a derived class Manager  
that inherits from Employee and adds department. Let GitHub Copilot complete the methods  
and constructor chaining.  
Expected Output #5  
• Auto-generated code like:  
Name: John, Salary: 50000, Dept: IT

**Output:**

# Employee class with name and salary

class Employee:

    def \_\_init\_\_(self, name, salary):

        self.name = name

        self.salary = salary

    def display(self):

        print(f"Name: {self.name}, Salary: {self.salary}")

# Manager class inherits from Employee and adds department

class Manager(Employee):

    def \_\_init\_\_(self, name, salary, department):

        super().\_\_init\_\_(name, salary)

        self.department = department

    def display(self):

        print(f"Name: {self.name}, Salary: {self.salary}, Dept: {self.department}")

# Example usage:

mgr = Manager("John", 50000, "IT")

mgr.display()  # Output: Name: John, Salary: 50000, Dept: IT

**output:**

**Name: John, Salary: 50000, Dept: IT**

**Explaination:**

 Employee is a base (parent) class with attributes **name** and **salary**, and a method display() to print them.

 Manager is a derived (child) class that **inherits** from Employee.

 Manager adds a new attribute **department** in addition to name and salary.

 super().\_\_init\_\_(name, salary) is used in Manager’s constructor to call the parent class constructor so code is not repeated.

 Manager overrides the display() method to also show the department along with name and salary.

 When mgr = Manager("John", 50000, "IT") is created, John’s name, salary, and department are stored.

 Calling mgr.display() prints:

Name: John, Salary: 50000, Dept: IT

 This demonstrates **inheritance, constructor chaining, and method overriding** in object-oriented programming.