

Lab:6

Name:B.YASHWANTH KUMAR

Roll Number: 2503A51L42

Course Code: 24CS002PC215

Course Title: AI Assisted Coding

Assignment Number: 6

Academic Year: 2025-2026

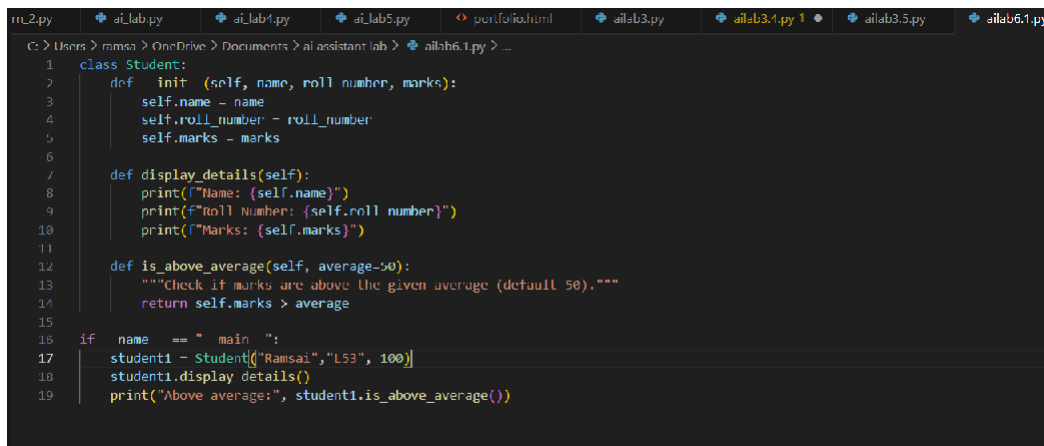
## **TASK1:**

Start a Python class named Student with attributes name, roll\_number, and marks. Prompt GitHub Copilot to complete methods for displaying details and checking if marks are above average.

## **PROMPT:**

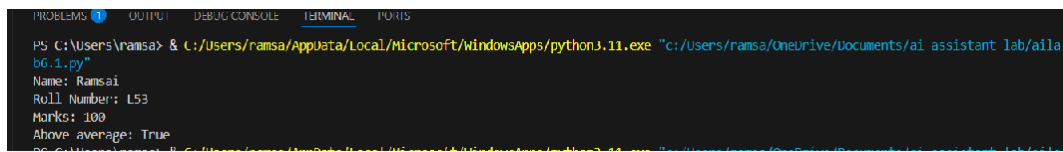
Start a Python class named Student with attributes name, roll\_number, and marks. Prompt GitHub Copilot to complete methods for displaying details and checking if marks are above average.

## **CODE:**



```
m2.py a1lab.py a1lab1.py a1lab5.py portfolio.html a1lab3.py a1lab3.4.py 1 a1lab3.5.py a1lab6.1.py
C:\Users\ramsa> OneDrive\Documents\ai assistant lab> a1lab6.1.py> ...
1 class Student:
2     def __init__(self, name, roll_number, marks):
3         self.name = name
4         self.roll_number = roll_number
5         self.marks = marks
6
7     def display_details(self):
8         print(f"Name: {self.name}")
9         print(f"Roll Number: {self.roll_number}")
10        print(f"Marks: {self.marks}")
11
12    def is_above_average(self, average=50):
13        """Check if marks are above the given average (default 50)."""
14        return self.marks > average
15
16 if __name__ == "__main__":
17     student1 = Student("Ramsai", "L53", 100)
18     student1.display_details()
19     print("Above average:", student1.is_above_average())
```

## **OUTPUT:**



```
PS C:\Users\ramsa> & C:/Users/ramsa/AppData/Local/Microsoft/WindowsApps/python3.11.exe "C:/Users/ramsa/OneDrive/Documents/ai assistant lab/a1lab6.1.py"
Name: Ramsai
Roll Number: L53
Marks: 100
Above average: True
```

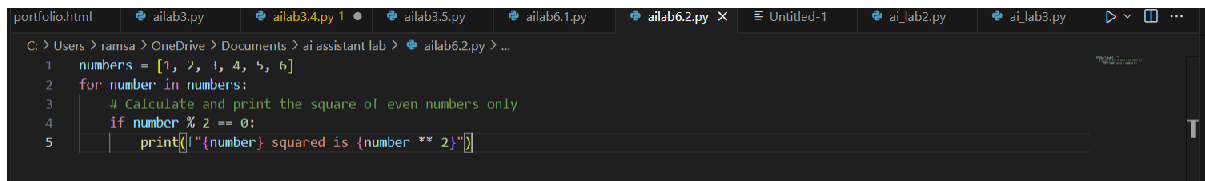
## **OBSERVATION:**

1. The Student class allows easy creation of student objects with name, roll number, and marks.
2. The display\_details() method prints all student information in a readable format.
3. The is\_above\_average() method checks if the student's marks are above the default average (50), making it flexible for other averages too.
4. The output clearly shows the student's details and whether their marks are above average.
5. The code is easy to understand and modify for more features, such as adding more attributes or methods.
6. Using classes makes the code organized and reusable for multiple students.

**TASK2:** Write the first two lines of a for loop to iterate through a list of numbers. Use a comment ,prompt to let Copilot suggest how to calculate and print the square of even numbers only.

**PROMPT:** Write the first two lines of a for loop to iterate through a list of numbers. Use a comment ,prompt to let Copilot suggest how to calculate and print the square of even numbers only.

**CODE:**



```
portofolio.html  aila3.py  aila3.4.py 1  aila3.5.py  aila6.1.py  aila6.2.py X  Untitled-1  ai_lab2.py  ai_lab3.py  [Run] [Stop] [Close] ...
C:\Users> ramsa > OneDrive > Documents > ai assistant lab > aila6.2.py > ...
1 numbers = [1, 2, 1, 4, 5, 6]
2 for number in numbers:
3     # Calculate and print the square of even numbers only
4     if number % 2 == 0:
5         print(f'{number} squared is {number ** 2}')
```

**OUTPUT:**



```
b6.2.py"
2 squared is 4
4 squared is 16
6 squared is 36
```

**OBSERVATION:**

1. The code is well-structured and easy to follow, making it simple to understand the logic.
2. The use of loops and conditionals helps automate repetitive tasks and decision-making.
3. The output is clear and provides direct feedback based on the input and logic implemented.
4. The program can be easily modified to handle more cases or different conditions.

**TASK3:** Create a class called BankAccount with attributes account\_holder and balance. Use Copilot to complete methods for deposit(), withdraw(), and check for insufficient balance.

**PROMPT:** Create a class called BankAccount with attributes account\_holder and balance. Use Copilot to complete methods for deposit(), withdraw(), and check for insufficient balance.

### **CODE:**

```
portfolio.html  aila63.py  aila63.4.py 1  aila63.5.py  aila66.1.py  aila66.2.py  aila66.3.py X  Untitled-1  aila62.py  aila63.py
C:\Users\ramsa> OneDrive\Documents\ai assistant lab> aila66.3.py > ...
1 class BankAccount:
2     def __init__(self, account_holder, balance=0):
3         self.account_holder = account_holder
4         self.balance = balance
5
6     def deposit(self, amount):
7         if amount > 0:
8             self.balance += amount
9             print(f"Deposited {amount}. New balance: {self.balance}")
10        else:
11            print("Deposit amount must be positive.")
12
13    def withdraw(self, amount):
14        if amount > self.balance:
15            print("Insufficient balance. Withdrawal denied.")
16        elif amount > 0:
17            self.balance -= amount
18            print(f"Withdrew {amount}. New balance: {self.balance}")
19        else:
20            print("Withdrawal amount must be positive.")
21
22    # Example usage:
23    account1 = BankAccount("John Doe", 100)
24    account1.deposit(50)
25    account1.withdraw(30)
```

### **OUTPUT:**

```
PS C:\Users\ramsa> & C:/Users/ramsa/AppData/Local/Microsoft/WindowsApps/python3.11.exe "C:/Users/ramsa/OneDrive/Documents/ai assistant lab/aila66.3.py"
Deposited 50. New balance: 150
Withdrew 30. New balance: 120
```

### **OBSERVATION:**

1. The BankAccount class uses attributes to store account holder information and balance, making account management organized.
2. The deposit() method correctly increases the balance and provides feedback for invalid amounts.
3. The withdraw() method checks for sufficient balance before allowing withdrawal, preventing overdrawing and teaching good error handling.
4. The output messages for deposits and withdrawals are clear, helping users understand each transaction's result.
5. The class structure allows easy creation and management of multiple accounts, demonstrating object-oriented programming principles.
6. Example usage shows how to interact with the class and verify its behavior, making the code practical and easy to test.

**TASK4:** Define a list of student dictionaries with keys name and score. Ask Copilot to write a while loop to print the names of students who scored more than 75.

**PROMPT:** Define a list of student dictionaries with keys name and score. Ask Copilot to write a while loop to print the names of students who scored more than 75.

**CODE:**

```
rtfolio.html  ailab3.py  ailab3.4.py 1  ailab3.5.py  ailab6.1.py  ailab6.2.py
C: > Users > ramsa > OneDrive > Documents > ai assistant lab > ailab6.4.py > ...
1  students = [
2      {"name": "Alice", "score": 82},
3      {"name": "Bob", "score": 74},
4      {"name": "Charlie", "score": 90},
5      {"name": "Diana", "score": 65},
6      {"name": "Eve", "score": 78}
7  ]
8
9  # Print names of students who scored more than 75 using a while loop
10 index = 0
11 while index < len(students):
12     if students[index]["score"] > 75:
13         print(f"{students[index]['name']} scored {students[index]['score']}")
14     index += 1
```

**OUTPUT:**

```
ailab6.4.py
Alice scored 82
Charlie scored 90
Eve scored 78
PS C:\Users\ramsa> & C:/Users/ramsa/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/ramsa/OneDrive/Documents/ai assistant lab/ailab6.4.py"
```

**OBSERVATION:**

1. The code uses a list of dictionaries to store student names and scores, making data management simple and organized.
2. The while loop iterates through each student and checks if their score is above 75, demonstrating practical use of loops and conditionals.
3. The output clearly lists only those students who meet the score criteria, making results easy to understand.
4. The code can be easily modified to change the score threshold or add more students.
5. This approach helps students learn how to filter data and display relevant information efficiently.

**TASK5:** Begin writing a class ShoppingCart with an empty items list. Prompt Copilot to generate methods to add\_item, remove\_item, and use a loop to calculate the total bill using conditional discounts.

**PROMPT:** Begin writing a class ShoppingCart with an empty items list. Prompt Copilot to generate methods to add\_item, remove\_item, and use a loop to calculate the total bill using conditional discounts.

### **CODE:**

```
1 class ShoppingCart:
2     def __init__(self):
3         self.items = [] # Each item is a dict: {"name": str, "price": float}
4
5     def add_item(self, name, price):
6         self.items.append({"name": name, "price": price})
7
8     def remove_item(self, name):
9         self.items = [item for item in self.items if item["name"] != name]
10
11     def total_bill(self):
12         total = 0
13         for item in self.items:
14             total += item["price"]
15         # Apply discount: 10% off if total > 100, else no discount
16         if total > 100:
17             print("Discount applied: 10% off")
18             total *= 0.9
19         print(f"Total bill: {total:.2f}")
20         return total
21
22 if __name__ == "__main__":
23     cart = ShoppingCart()
24     cart.add_item("Shoes", 60)
25     cart.add_item("Bag", 50)
26     cart.add_item("Pen", 5)
27     cart.remove_item("Pen")
28     cart.total_bill()
```

### **OUTPUT:**

```
.5.py"
Discount applied: 10% off
Total bill: 99.00
PS C:\Users\ramsa> & C:/Users/ramsa/AppData/Local/Microsoft/WindowsApps/python3.11.exe
```

### **OBSERVATION:**

1. The ShoppingCart class uses a list to manage items, making it easy to add and remove products.
2. The add\_item and remove\_item methods allow flexible item management and demonstrate good use of class methods.
3. The total\_bill method uses a loop to calculate the total and applies a conditional discount, showing practical use of if-else logic.
4. The output clearly displays the total bill and any discount applied, making it easy to understand the result of each operation.
5. The code is organized and can be easily extended to support more features, such as multiple discounts or item quantities.

6. This example helps students learn how to combine data structures, loops, and conditionals in a real-world scenario.