## NAME:HARSHA

## ROLL NO:2403A51352

## BATCH:14

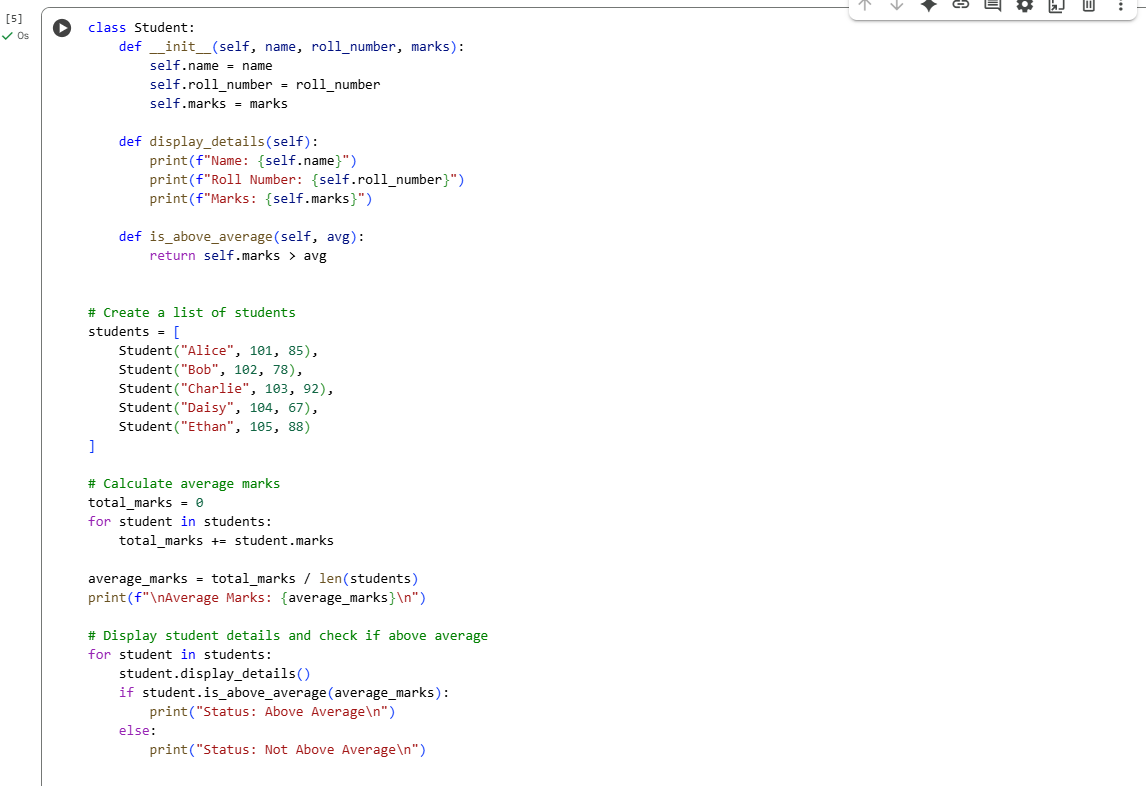
## DATE:10-09-2025

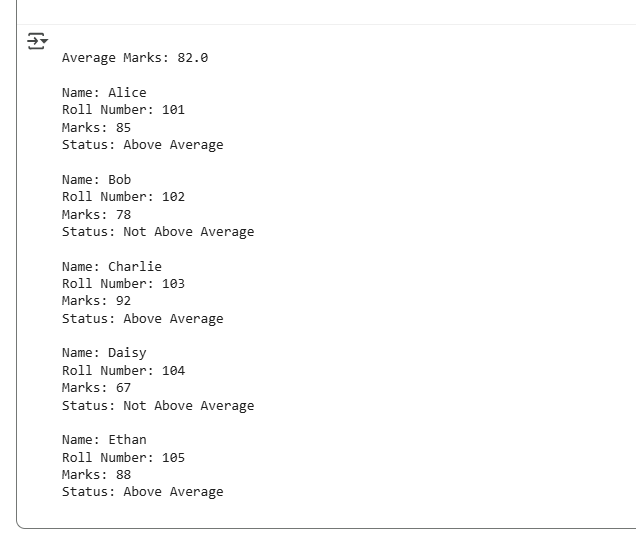
ASSIGNMENT-6.4

Task Description #1:

• 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&OUTPUT:





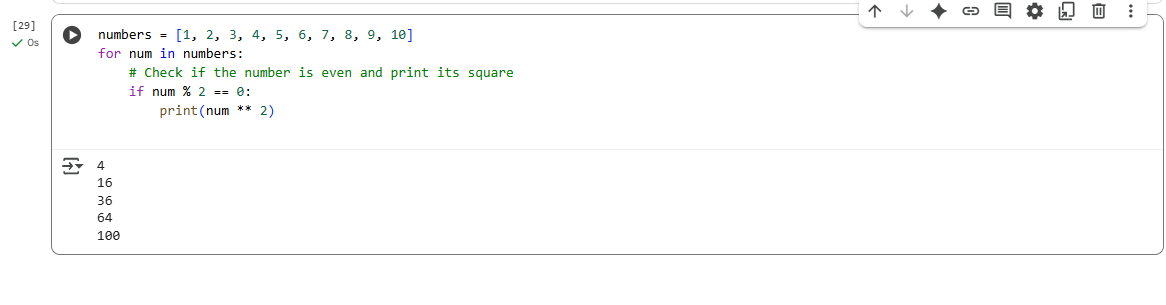
### OBSERVATION:

1. The Student class effectively encapsulates student data and behavior.
2. Constructor (\_\_init\_\_) initializes each student's name, roll number, and marks.
3. The display\_details() method neatly prints individual student information.
4. The is\_above\_average() method compares marks against a calculated average.
5. A list of Student objects allows for easy iteration and batch processing.
6. Average marks are calculated using a loop and then used for comparison.
7. Students with marks greater than 82.0 are identified as above average.
8. The logic separates data handling (attributes) from behavior (methods).
9. Looping through objects demonstrates object-oriented programming in practice.
10. Output is clear, structured, and useful for performance evaluation.

Task Description #2:

• 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 &OUTPUT:



## OBSERVATION:

1.The for loop is used to iterate over a list of integers from 1 to 10.

2.A comment is added to guide Copilot to generate the correct logic.

3.Copilot can understand the context from comments and suggest relevant code.

4.The condition if num % 2 == 0 correctly identifies even numbers.

5.num \*\* 2 computes the square of the number.

6.Only even numbers are processed, odd numbers are ignored.

7.Output shows the squares of 2, 4, 6, 8, and 10.

8.This approach separates data (list) from logic (loop and condition).

9.Comment-driven development helps guide AI-assisted code suggestions.

10.The example demonstrates clean and readable code for beginners.

**Task Description #3:**

**•** 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&OUTPUT:



## OBSERVATION:

1.The class BankAccount demonstrates basic principles of **object-oriented programming** in Python.

2. It defines two core attributes: account\_holder (a string) and balance (a numeric value initialized with a default of 0).

3. The \_\_init\_\_ constructor ensures that each instance starts with a specific account holder and an optional initial balance.

4. The deposit() method handles increasing the balance and includes a check to ensure only positive values are allowed.

5. If a non-positive deposit is attempted, the method responds with a validation message instead of silently failing.

6. The withdraw() method implements both the deduction logic and a safety check to avoid negative balances.

7. When an attempted withdrawal exceeds the available balance, an appropriate error message is printed, demonstrating **defensive programming**.

8. The display() method provides a clear snapshot of the current state of the account.

9.A list named transactions is used to simulate a batch of banking actions, with each item containing the operation type and amount.

10. The for loop iterates through this list, using conditionals to decide whether to call deposit() or withdraw() based on the action type.

11.This loop-driven approach mimics real-world transaction processing where multiple operations may be queued.

12.The structure can be easily extended to accept user input or read transactions from a file or database.

13.The final print statement, after the loop, calls the display() method to show the end result, offering clear visibility into account status.

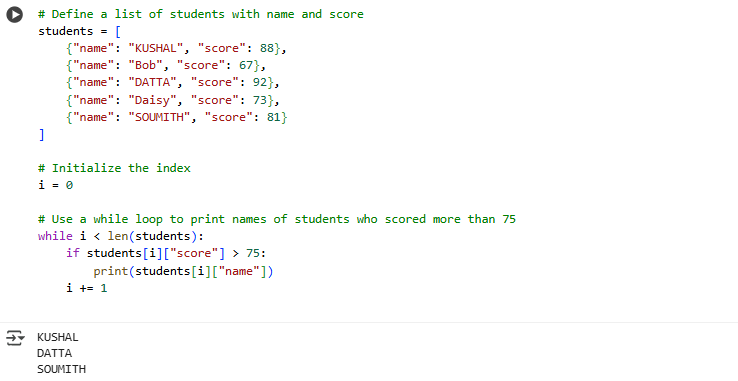
14.The example uses readable and descriptive print statements, making the program's flow easy to follow.

15.The code avoids hardcoding logic inside the loop, instead relying on class methods, showing **good encapsulation**.

**Task Description #4:**

**•** 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&OUTPUT:



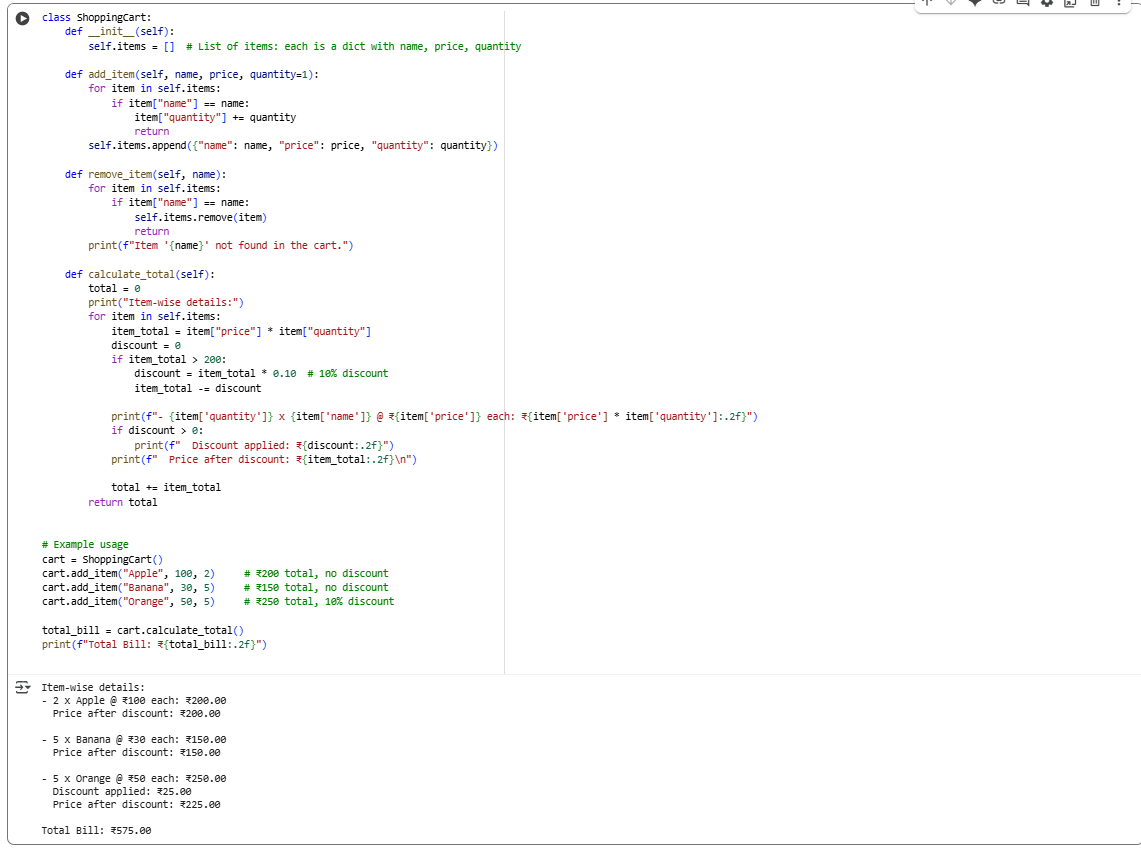
## OBSERVATION:

1. A list of dictionaries is used to represent student data, with each dictionary containing name and score keys.
2. This structure allows for easy and scalable storage of multiple students' records.
3. The while loop is used instead of a for loop, demonstrating a fundamental control structure in Python.
4. The loop uses an index variable i, initialized to 0, to iterate through the list.
5. The condition i < len(students) ensures the loop stops after the last student.
6. Inside the loop, an if condition checks whether the current student's score is greater than 75.
7. Only names of students who meet the condition are printed, showcasing **conditional filtering**.
8. The i += 1 statement ensures that the index increments with each iteration, preventing an infinite loop.
9. The output reflects only those students who scored above the threshold, confirming correct logic.
10. The approach is clean, efficient, and easy to modify for different score thresholds or additional conditions.
11. This task reinforces the importance of loop control, indexing, and conditionals in Python programming.
12. The code is suitable for beginner-level understanding of list traversal and dictionary access.

**Task Description #5:**

**•** 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&OUTPUT:



## OBSERVATION:

1.The ShoppingCart class uses an empty list items to hold all shopping cart entries as dictionaries.

2. The add\_item() method checks if an item already exists; if so, it increases the quantity, preventing duplicates.

3. The remove\_item() method allows for removing an item by its name, with a message if the item is not found.

4. The calculate\_total() method iterates through all items, calculating the subtotal for each (price × quantity).

5. A **10% discount** is applied individually to items whose subtotal exceeds ₹200, demonstrating conditional discounts.

6. The method clearly prints detailed billing information per item: original price, discount amount (if any), and price after discount.

7. This detailed breakdown improves transparency and helps users understand how the final bill is computed.

8. The discount logic is simple yet effective, encouraging bulk purchases of higher-value items.

9. The code properly sums the discounted totals to provide the final payable amount.

10. Overall, the design promotes good encapsulation, readability, and is easily extendable for future features like multiple discount rules or coupon codes.

11.The sample usage with multiple items showcases realistic shopping scenarios and tests discount application well.

12.User-friendly print statements make it clear how individual items contribute to the total cost and discounts applied.