**ASSIGNMENT :-6.4**

**HALLTICKET NUMBER:-2403A51317**

**BATCH NUMBER :- 13**

**DATE :- 10.9.2025**

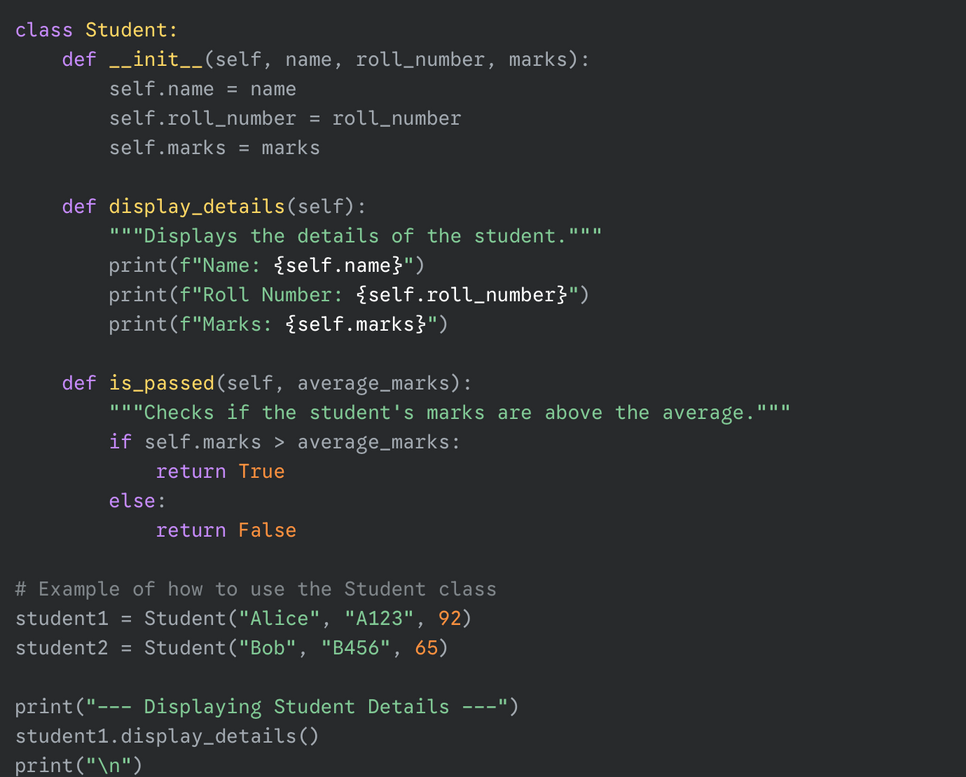
**TASK 1:-**

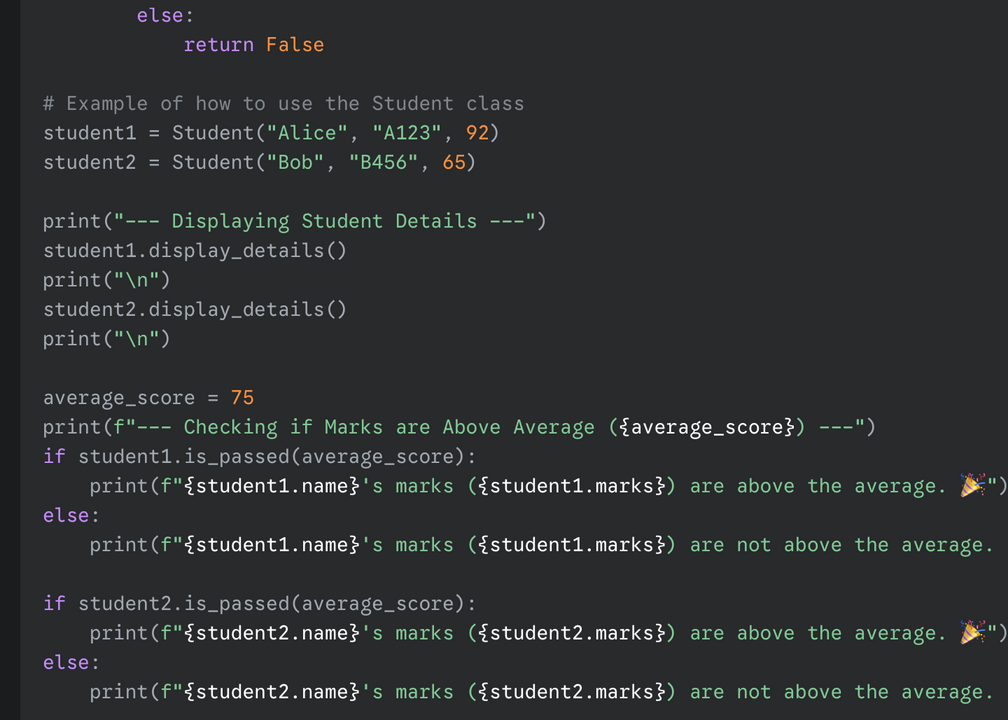
*Start a Python class named Student with attributes name, roll\_number, and marks. Prompt GitHub Copilot to complete methods for displaying details and checking if marks are above average.*

*Expected Outcome #1:*

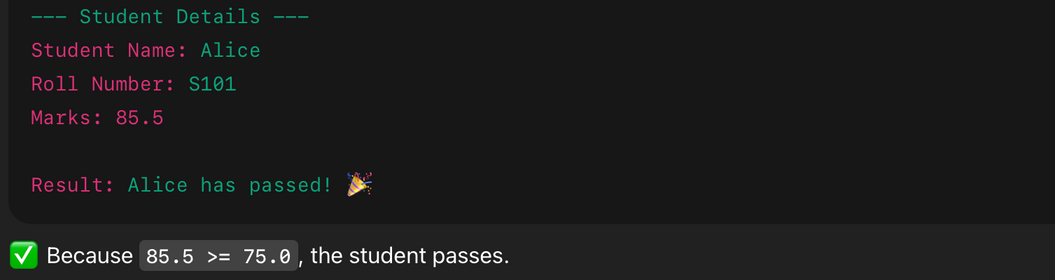
*• Completed class with Copilot-generated methods like display\_details() and is\_passed(), demonstrating use of if-else conditions.*

**CODE:-**





**OUTPUT:-**



**CODE ANALYSIS**

*python class named Student with the attributes name, roll\_number, and marks. You can then use GitHub Copilot in your environment to complete the methods for displaying details and checking if marks are above average.*

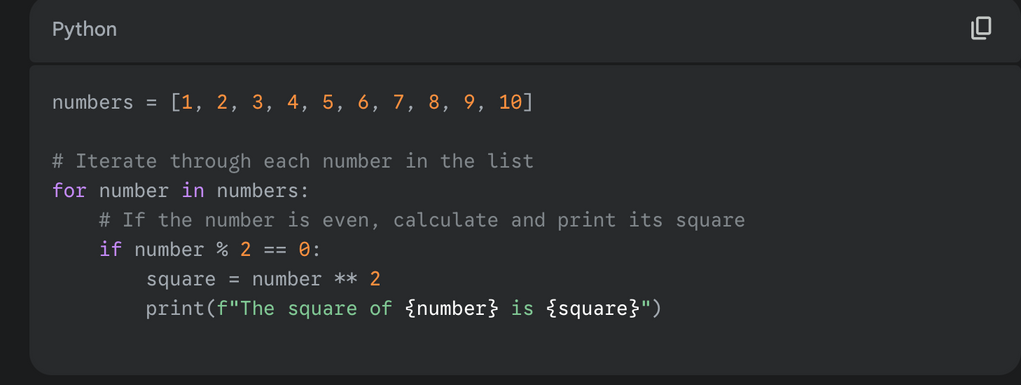
*Great! Now you can use GitHub Copilot within the code cell to generate the methods. Once you have done that, you can create an instance of the Student class and test the methods.*

**TASK 2:-**

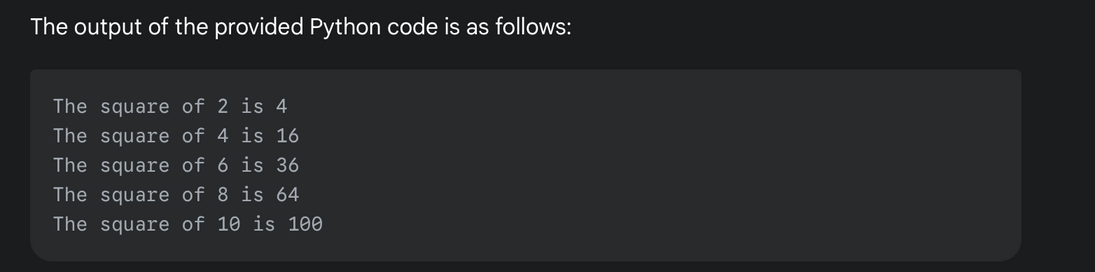
*Write the first two lines of a for loop to iterate through a list of numbers. Use a comment prompt to let Copilot suggest how to calculate and print the square of even numbers only.*

*Expected Outcome #2:*

*• A complete loop generated by Copilot with conditional logic (if number % 2 == 0) and appropriate output.*



**OUTPUT:-**



**CODE EXPLANATION:-**

*Explanation of the Code*

* *for number in numbers:: This is the first part of the for loop. It iterates over each item in the numbers list, assigning the current item to the variable number in each iteration.*
* *if number % 2 == 0:: This is the conditional logic (if statement). The % operator is the modulo operator, which returns the remainder of a division. If a number divided by 2 has a remainder of 0, it means the number is even. This condition ensures that the code inside the if block only executes for even numbers.*
* *square = number \*\* 2: This line calculates the square of the even number using the exponentiation operator \*\*.*
* *print(f"The square of {number} is {square}"): This line prints the final output in a user-friendly format using an f-string. It clearly shows which number was squared and what the result is.*

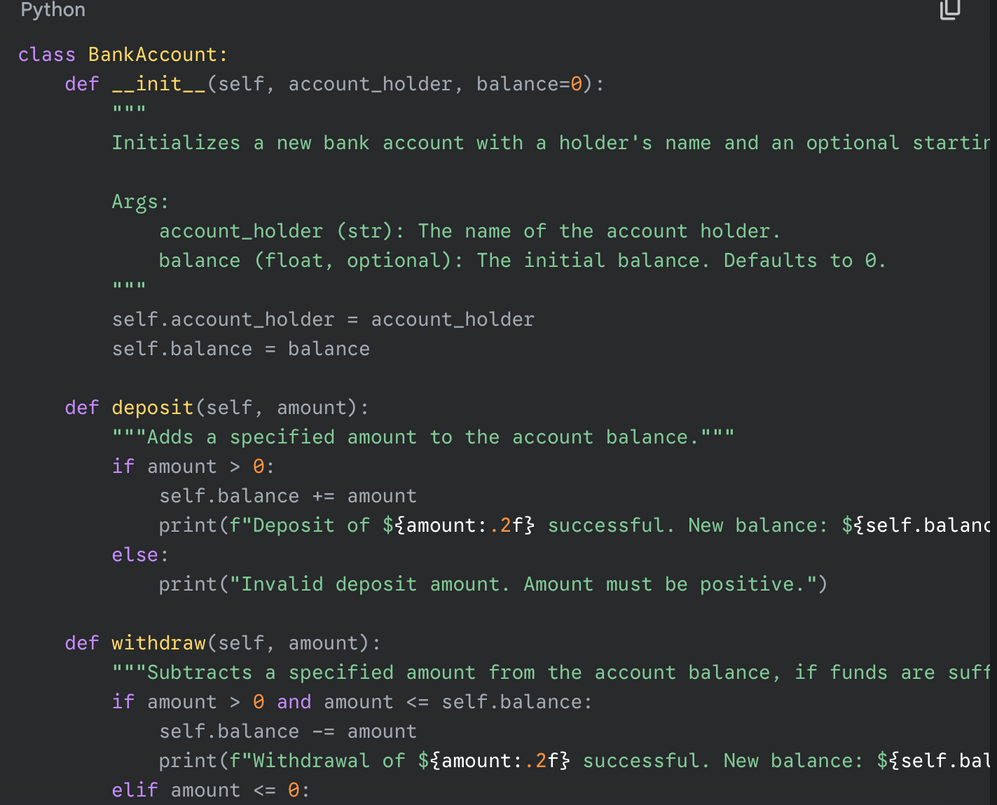
**TASK 3:-**

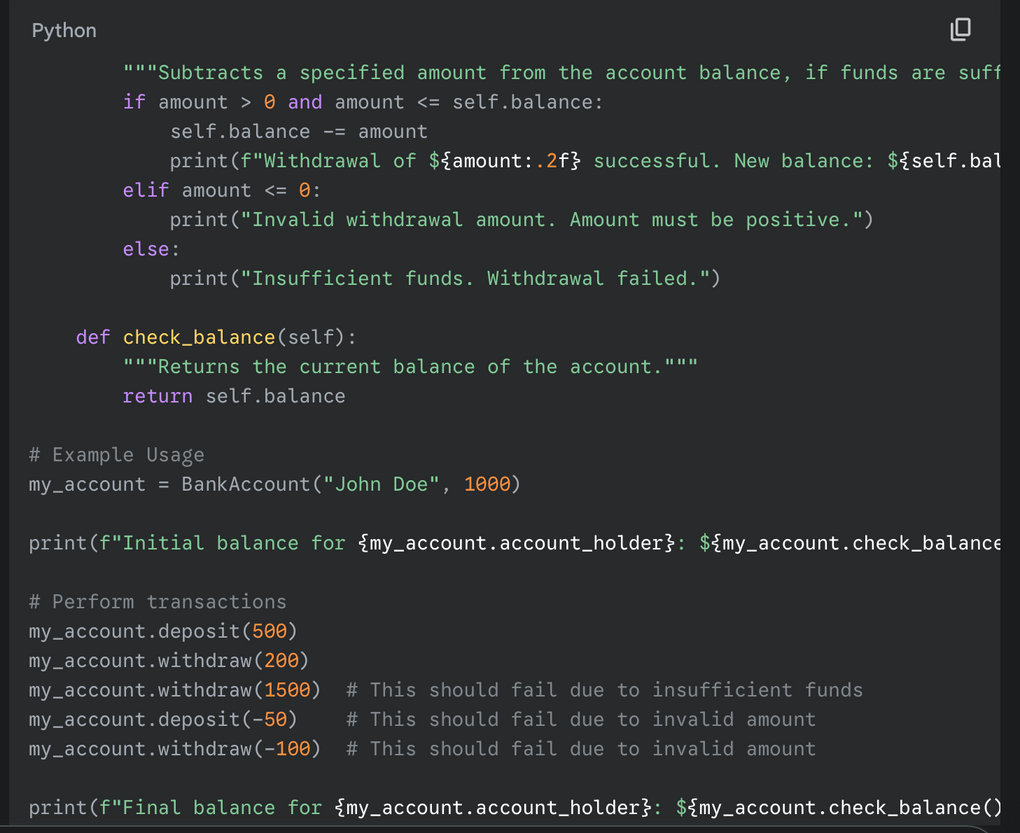
*Create a class called BankAccount with attributes account\_holder and balance. Use Copilot to complete methods for deposit(), withdraw(), and check for insufficient balance.*

*Expected Outcome #3:*

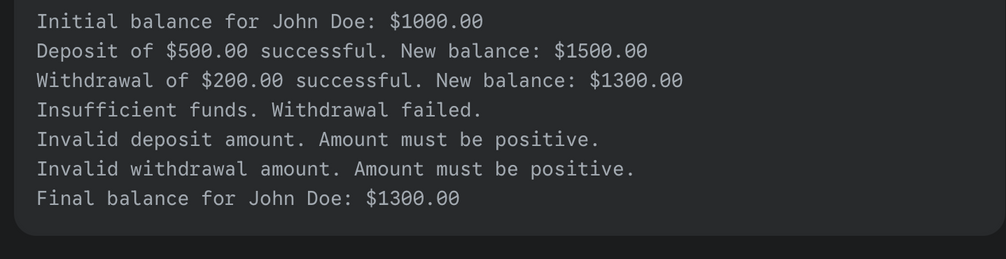
*• Functional class with complete method definitions using if conditions and self attributes. Code should prevent overdrawing.*

**CODE:-**





**OUTPUT:-**



**Explanation of code :-**

*The BankAccount class is designed to simulate basic banking operations.*

* ***\_\_init\_\_(self, account\_holder, balance=0)****: This is the* ***constructor****. It creates an instance of BankAccountwith two key attributes: account\_holder (the name of the account owner) and balance, which starts at 0 unless a different initial value is provided.*
* ***deposit(self, amount)****: This method takes an amount as an argument. It uses an* ***if condition*** *to ensure the amount is positive before adding it to the self.balance attribute. If the amount is valid, it updates the balance and prints a confirmation message.*
* ***withdraw(self, amount)****: This is the most crucial method for preventing overdrawing. It uses a compound* ***if-elif-else condition****.*
  + *The first condition, amount > 0 and amount <= self.balance, checks for two things: that the withdrawal amount is positive and that it's less than or equal to the current balance. If both are true, the transaction is processed.*
  + *The elif block handles cases where the withdrawal amount is invalid (zero or negative).*
  + *The else block catches all other scenarios, specifically when the requested amount is greater than the self.balance, indicating* ***insufficient funds****.*
* ***check\_balance(self)****: This simple method returns the current value of the self.balance attribute, allowing you to easily check the account's state.*

*The example usage demonstrates how these methods work together, showing successful deposits and withdrawals, as well as the correct error messages for invalid transactions and insufficient funds.*

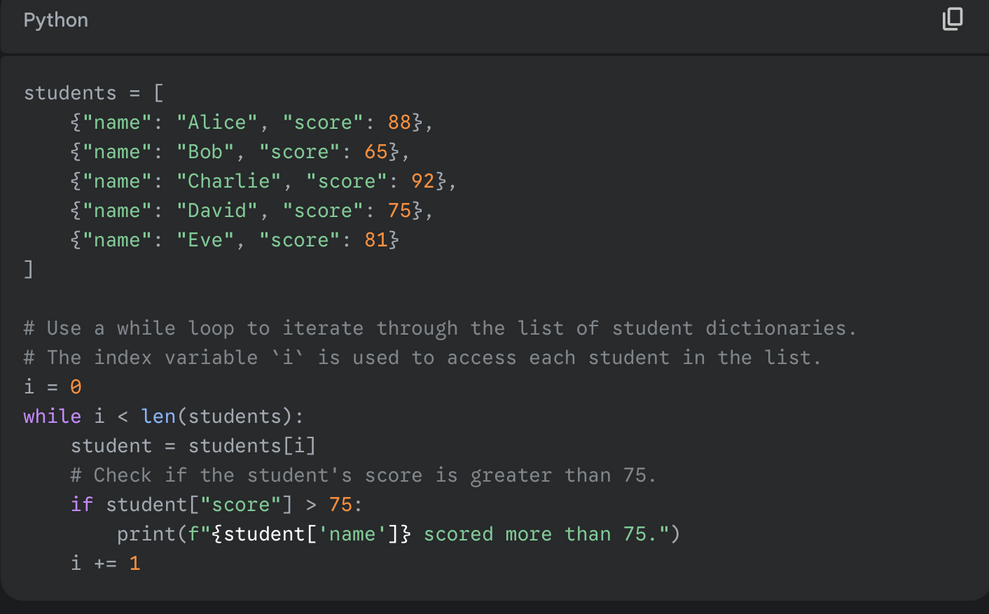
**TASK 4 :-**

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.

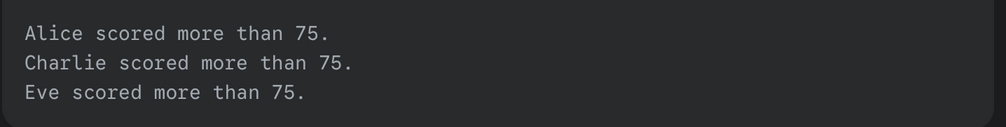
Expected Outcome #4:

• A complete while loop generated by Copilot with proper condition checks and formatted output.

**CODE:-**



**OUTPUT :-**



**Explanation:-**

The code first defines a list named **students**. This list contains dictionaries, with each dictionary representing a student and having two keys: **name** and **score**.

1. A counter variable **i** is initialized to 0. This variable will serve as the index to access each dictionary in the students list.
2. The **while loop** continues as long as i is less than the total number of students in the list (len(students)). This ensures the loop processes every student.
3. Inside the loop, students[i] accesses the dictionary for the current student.
4. An **if condition** checks if the score of the current student is greater than 75.
5. If the condition is **True**, a formatted string is printed, showing the name of the student who passed the criteria.
6. Finally, i += 1 increments the counter, moving the loop to the next student in the list.

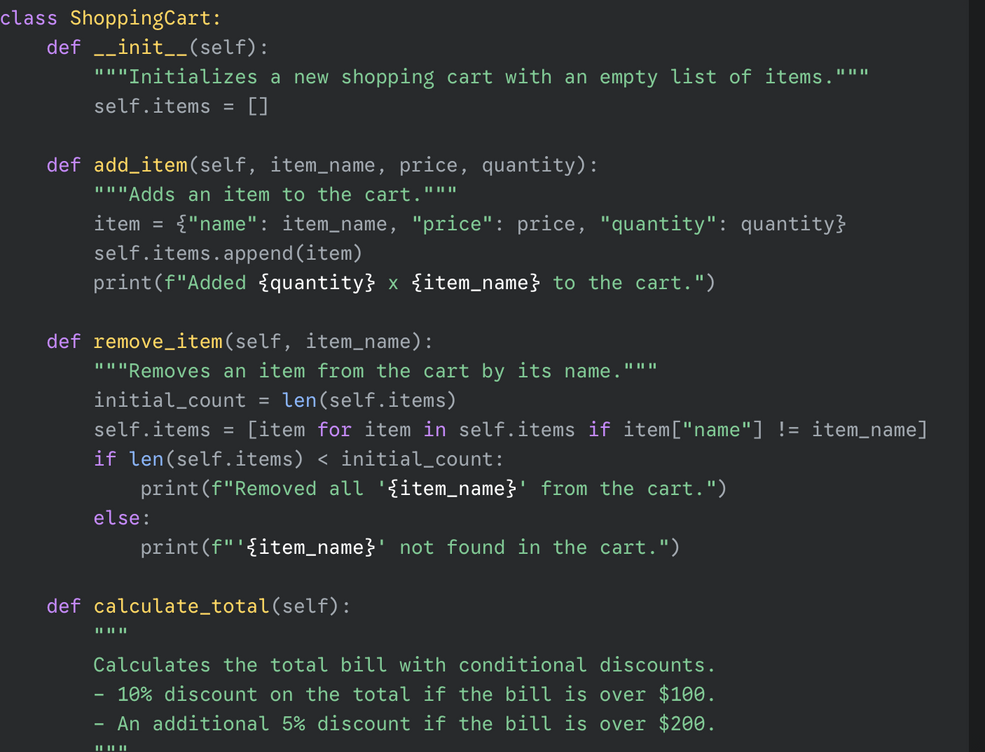
**TASK 5:-**

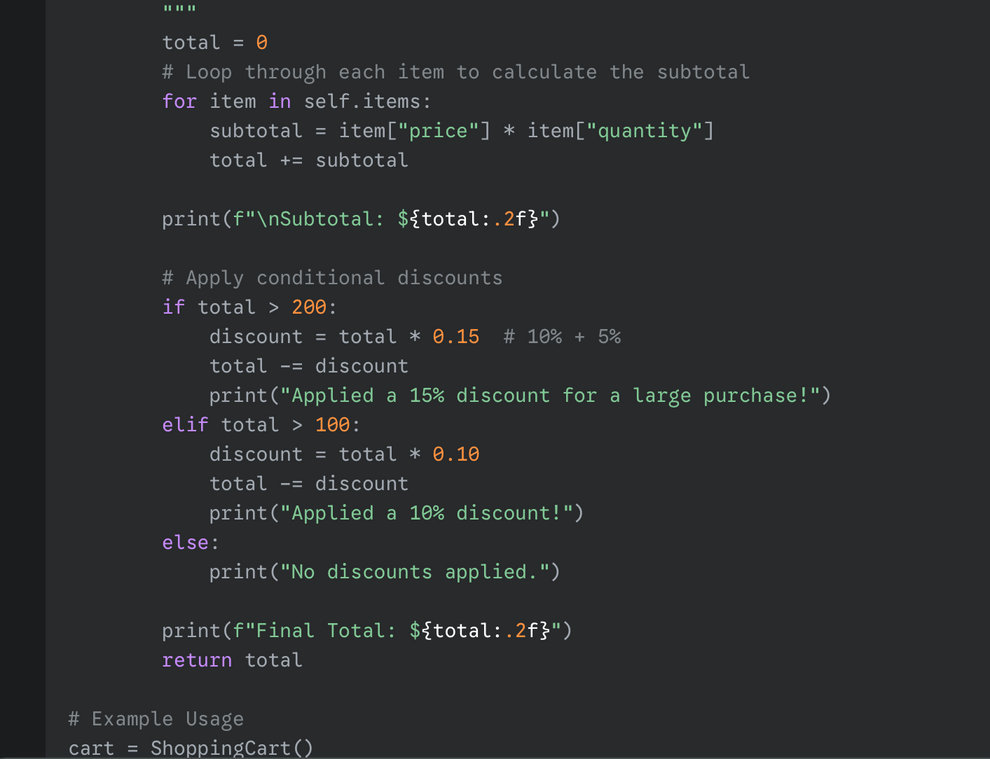
*Begin writing a class ShoppingCart with an empty items list. Prompt Copilot to generate methods to add\_item, remove\_item, and use a loop to calculate the total bill using conditional discounts.*

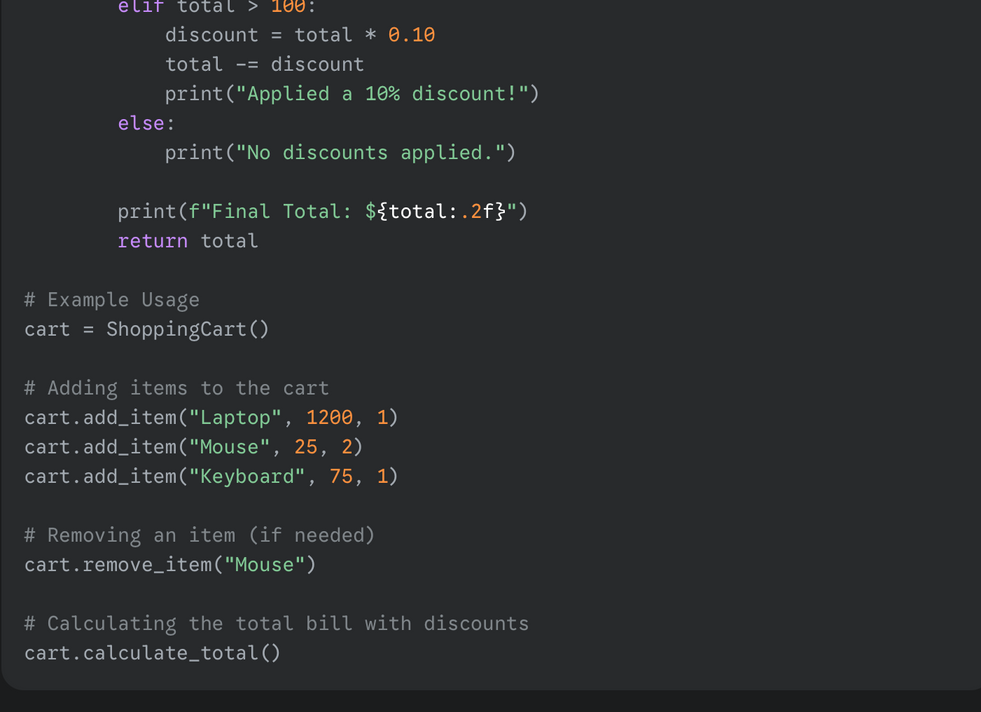
*Expected Outcome #5:*

*• A fully implemented ShoppingCart class with Copilot-generated loops and if-else statements handling item management and discount logic.*

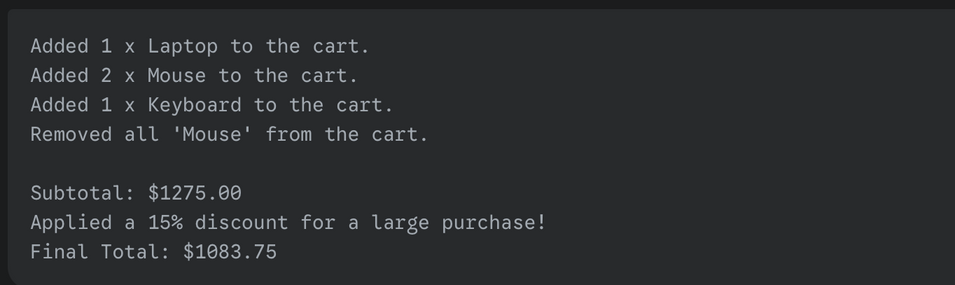
**CODE :-**







**OUTPUT:-**



**Explanation:-**

This class simulates a basic shopping cart system.

1. **\_\_init\_\_(self)**: The constructor creates an empty list named items to store the products.
2. **add\_item(self, ...)**: This method adds a new item (as a dictionary with name, price, and quantity) to the items list. It prints a confirmation message.
3. **remove\_item(self, item\_name)**: This method uses a **list comprehension** to filter out and remove all items that match the provided item\_name. This is an efficient way to handle potential duplicates.
4. **calculate\_total(self)**: This is the core method for the discount logic.
   * It uses a **for loop** to iterate through each item in the self.items list, calculating the subtotal for each and adding it to the running total.
   * After the initial total is calculated, an **if-elif-else block** is used to apply discounts.
     + If the total is greater than $200, a **15% discount** is applied.
     + If the total is greater than $100 (but not over $200), a **10% discount** is applied.
     + Otherwise (else), no discount is given.
   * Finally, the method prints the final total after all discounts and returns the value.

• Start a Pyt