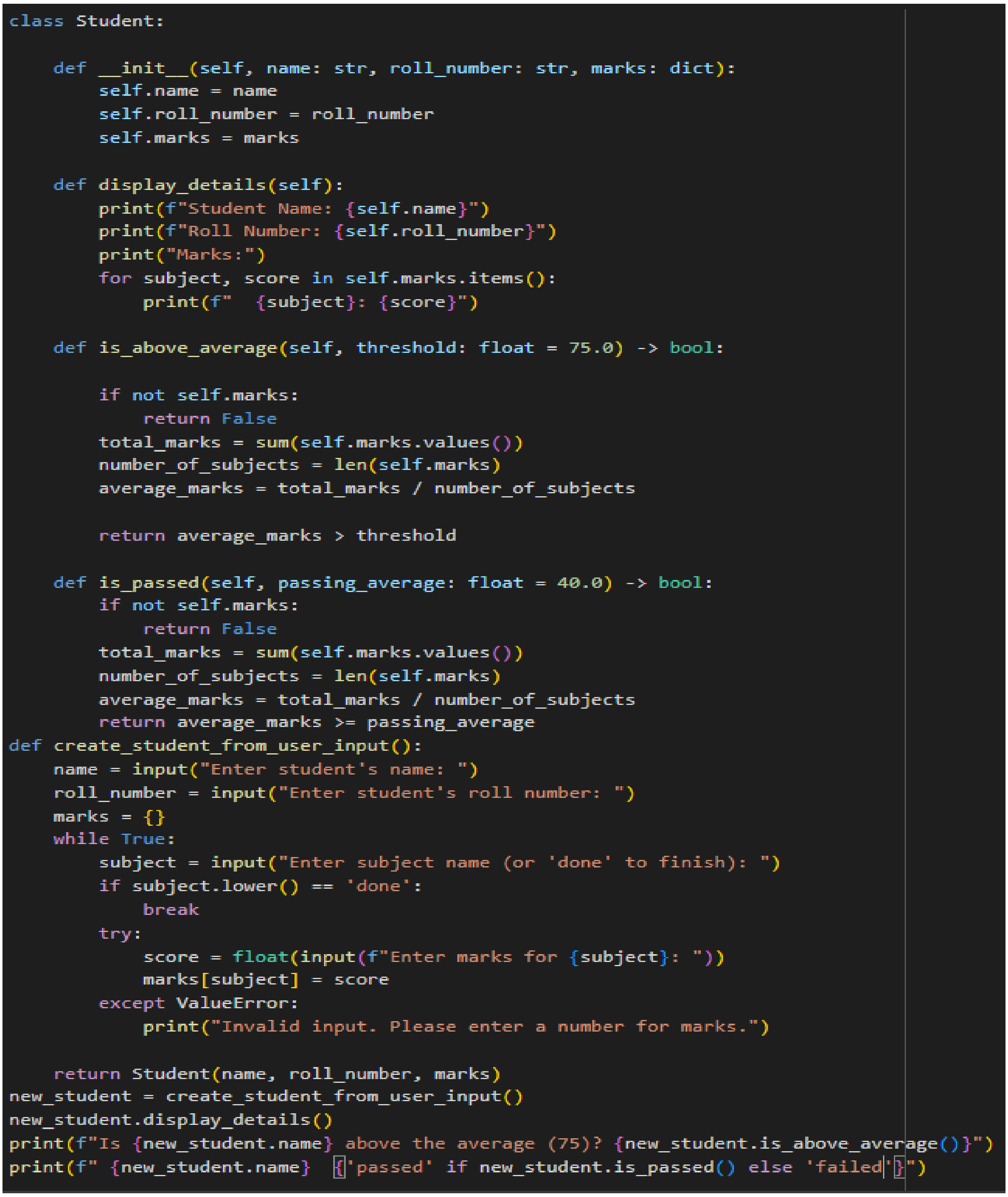
ASSIGNMENT 6

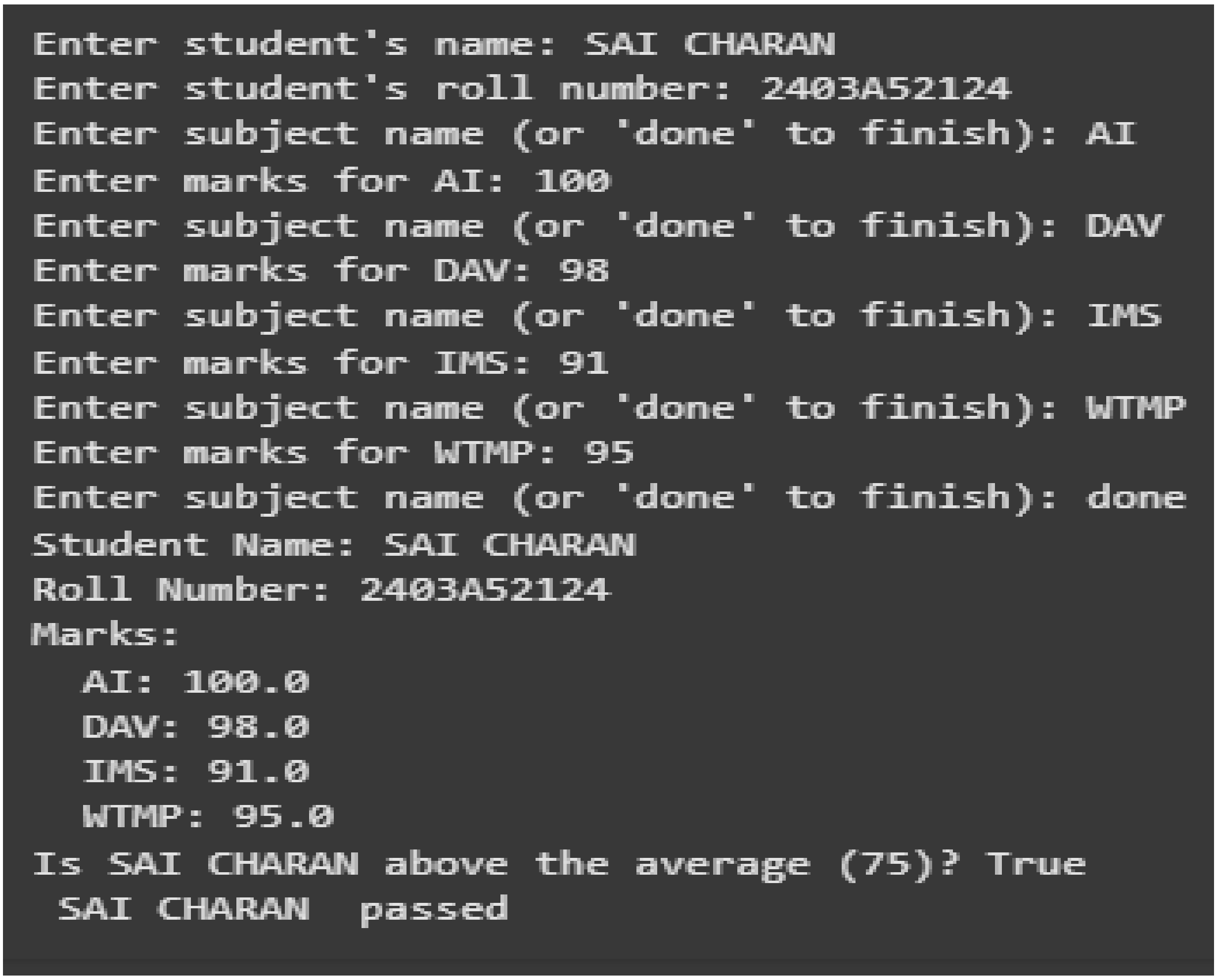
RAMIDI.SAI CHARAN

2403A52124

TASK 1 :

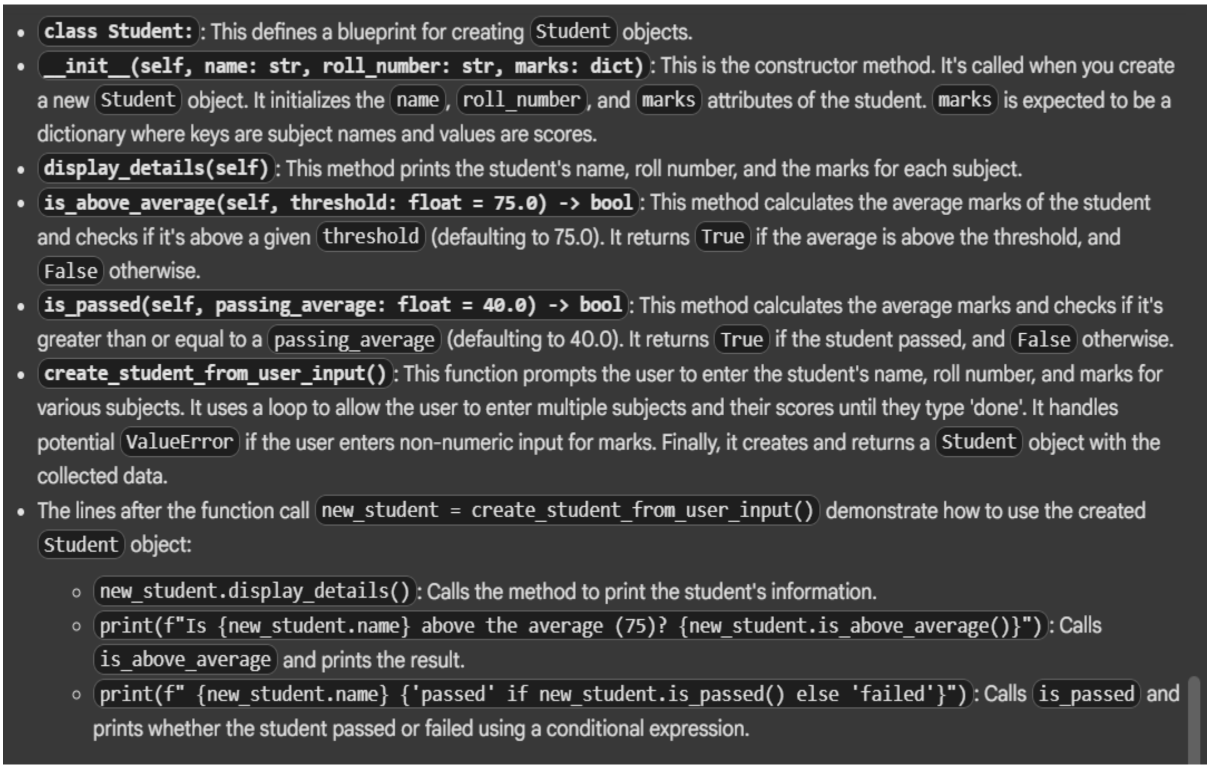


```python
class Student:

    def __init__(self, name: str, roll_number: str, marks: dict):
        self.name = name
        self.roll_number = roll_number
        self.marks = marks

    def display_details(self):
        print(f"Student Name: {self.name}")
        print(f"Roll Number: {self.roll_number}")
        print("Marks:")
        for subject, score in self.marks.items():
            print(f"  {subject}: {score}")

    def is_above_average(self, threshold: float = 75.0) -> bool:

        if not self.marks:
            return False
        total_marks = sum(self.marks.values())
        number_of_subjects = len(self.marks)
        average_marks = total_marks / number_of_subjects

        return average_marks > threshold

    def is_passed(self, passing_average: float = 40.0) -> bool:
        if not self.marks:
            return False
        total_marks = sum(self.marks.values())
        number_of_subjects = len(self.marks)
        average_marks = total_marks / number_of_subjects
        return average_marks >= passing_average
def create_student_from_user_input():
    name = input("Enter student's name: ")
    roll_number = input("Enter student's roll number: ")
    marks = {}
    while True:
        subject = input("Enter subject name (or 'done' to finish): ")
        if subject.lower() == 'done':
            break
        try:
            score = float(input(f"Enter marks for {subject}: "))
            marks[subject] = score
        except ValueError:
            print("Invalid input. Please enter a number for marks.")

    return Student(name, roll_number, marks)
new_student = create_student_from_user_input()
new_student.display_details()
print(f"Is {new_student.name} above the average (75)? {new_student.is_above_average()}")
print(f" {new_student.name}  {'passed' if new_student.is_passed() else 'failed'}")
```



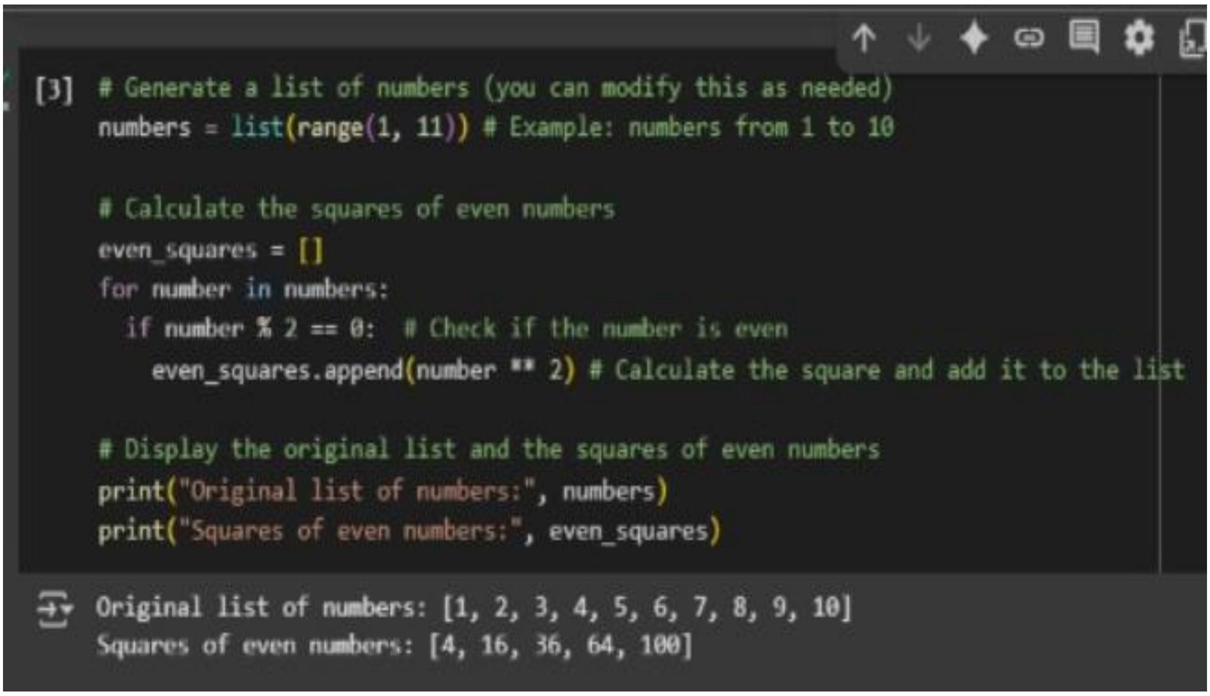
```
Enter student's name: SAI CHARAN
Enter student's roll number: 2403A52124
Enter subject name (or 'done' to finish): AI
Enter marks for AI: 100
Enter subject name (or 'done' to finish): DAV
Enter marks for DAV: 98
Enter subject name (or 'done' to finish): IMS
Enter marks for IMS: 91
Enter subject name (or 'done' to finish): WTMP
Enter marks for WTMP: 95
Enter subject name (or 'done' to finish): done
Student Name: SAI CHARAN
Roll Number: 2403A52124
Marks:
  AI: 100.0
  DAV: 98.0
  IMS: 91.0
  WTMP: 95.0
Is SAI CHARAN above the average (75)? True
 SAI CHARAN  passed
```

EXPLANATION:



- **`class Student:`**: This defines a blueprint for creating `Student` objects.
- **`__init__(self, name: str, roll_number: str, marks: dict)`**: This is the constructor method. It's called when you create a new `Student` object. It initializes the `name`, `roll_number`, and `marks` attributes of the student. `marks` is expected to be a dictionary where keys are subject names and values are scores.
- **`display_details(self)`**: This method prints the student's name, roll number, and the marks for each subject.
- **`is_above_average(self, threshold: float = 75.0) -> bool`**: This method calculates the average marks of the student and checks if it's above a given `threshold` (defaulting to 75.0). It returns `True` if the average is above the threshold, and `False` otherwise.
- **`is_passed(self, passing_average: float = 40.0) -> bool`**: This method calculates the average marks and checks if it's greater than or equal to a `passing_average` (defaulting to 40.0). It returns `True` if the student passed, and `False` otherwise.
- **`create_student_from_user_input()`**: This function prompts the user to enter the student's name, roll number, and marks for various subjects. It uses a loop to allow the user to enter multiple subjects and their scores until they type 'done'. It handles potential `ValueError` if the user enters non-numeric input for marks. Finally, it creates and returns a `Student` object with the collected data.
- The lines after the function call `new_student = create_student_from_user_input()` demonstrate how to use the created `Student` object:
    - `new_student.display_details()`: Calls the method to print the student's information.
    - `print(f"Is {new_student.name} above the average (75)? {new_student.is_above_average()}")`: Calls `is_above_average` and prints the result.
    - `print(f" {new_student.name} {'passed' if new_student.is_passed() else 'failed'}")`: Calls `is_passed` and prints whether the student passed or failed using a conditional expression.

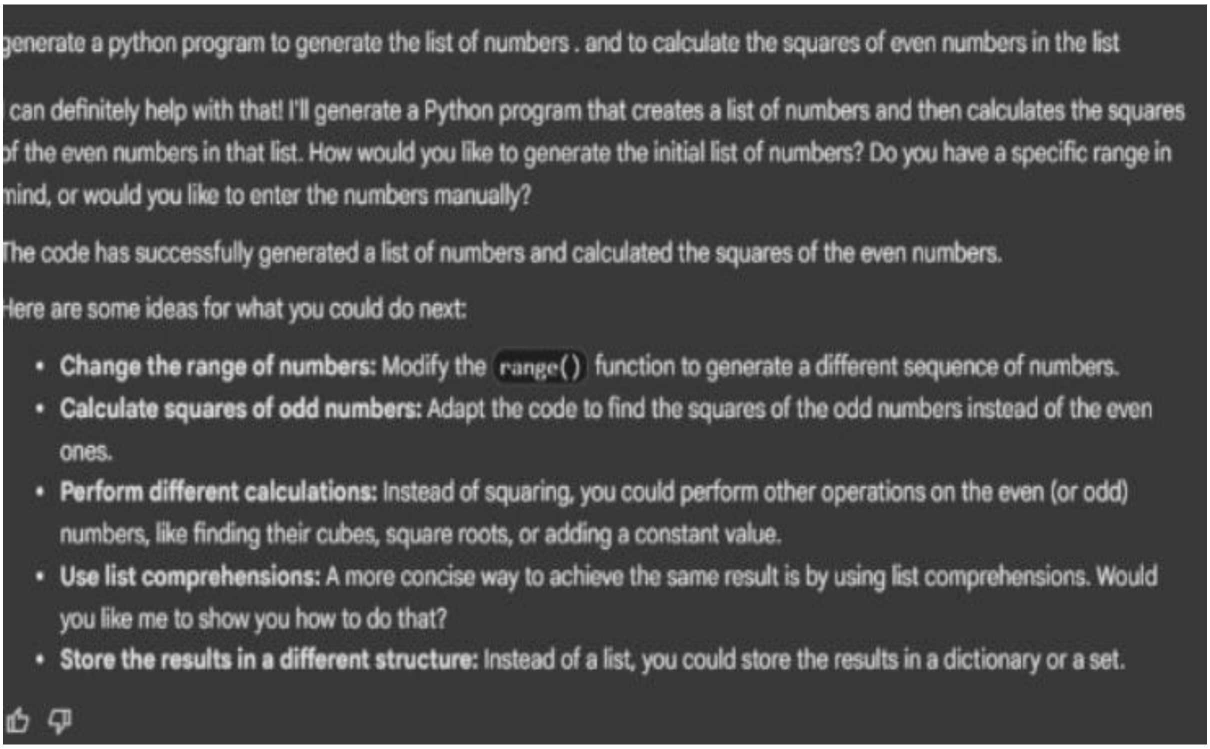
TASK 2 :



```python
# Generate a list of numbers (you can modify this as needed)
numbers = list(range(1, 11)) # Example: numbers from 1 to 10

# Calculate the squares of even numbers
even_squares = []
for number in numbers:
  if number % 2 == 0:  # Check if the number is even
    even_squares.append(number ** 2) # Calculate the square and add it to the list

# Display the original list and the squares of even numbers
print("Original list of numbers:", numbers)
print("Squares of even numbers:", even_squares)
```

```
Original list of numbers: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Squares of even numbers: [4, 16, 36, 64, 100]
```

EXPLANATION :



enerate a python program to generate the list of numbers . and to calculate the squares of even numbers in the list

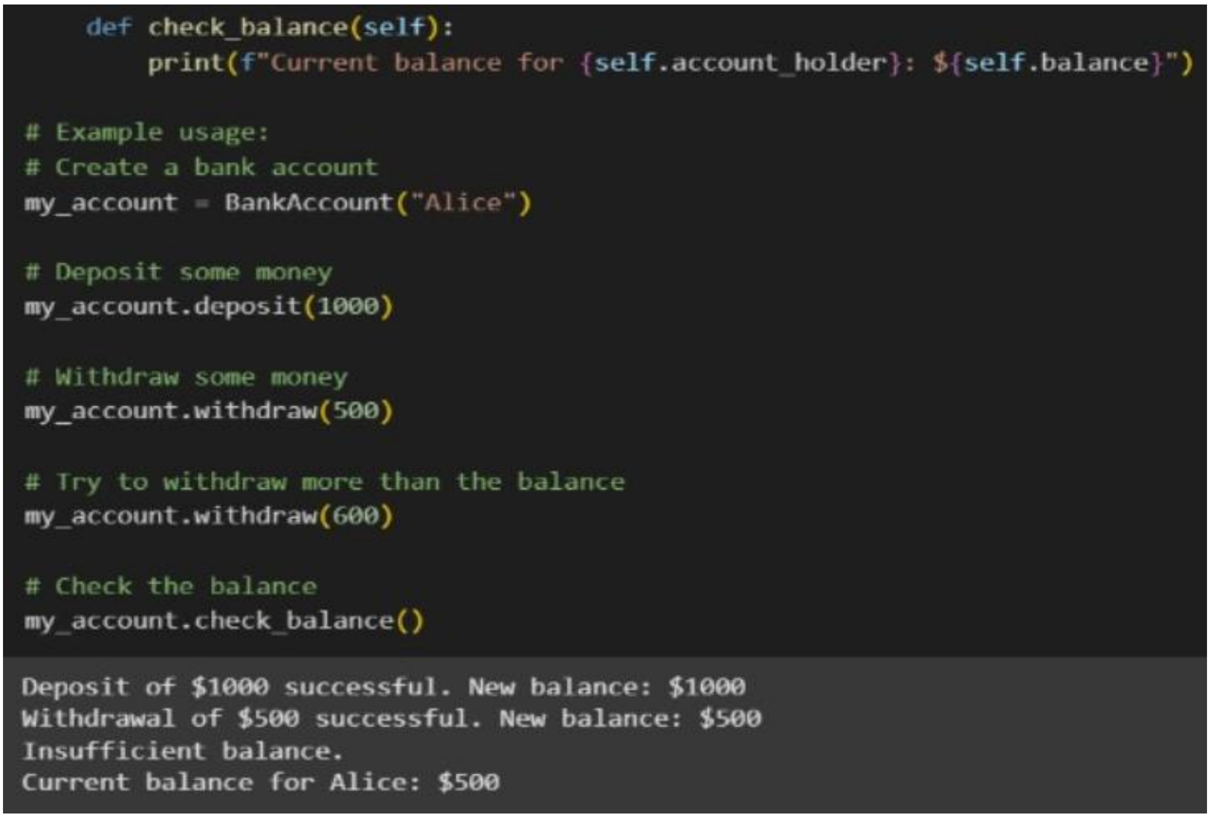
can definitely help with that! I'll generate a Python program that creates a list of numbers and then calculates the squares f the even numbers in that list. How would you like to generate the initial list of numbers? Do you have a specific range in ind, or would you like to enter the numbers manually?

he code has successfully generated a list of numbers and calculated the squares of the even numbers.

ere are some ideas for what you could do next:

- **Change the range of numbers:** Modify the `range()` function to generate a different sequence of numbers.
- **Calculate squares of odd numbers:** Adapt the code to find the squares of the odd numbers instead of the even ones.
- **Perform different calculations:** Instead of squaring, you could perform other operations on the even (or odd) numbers, like finding their cubes, square roots, or adding a constant value.
- **Use list comprehensions:** A more concise way to achieve the same result is by using list comprehensions. Would you like me to show you how to do that?
- **Store the results in a different structure:** Instead of a list, you could store the results in a dictionary or a set.

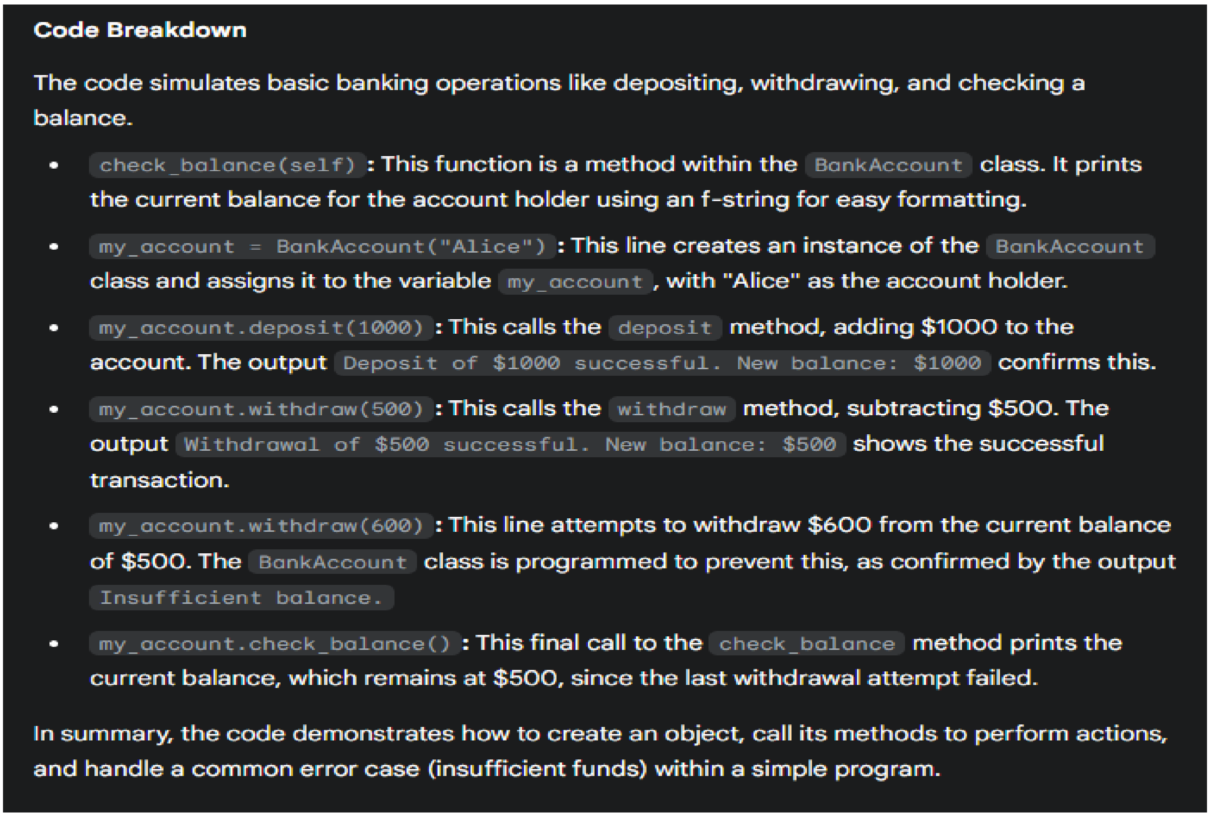
TASK 3 :



```
    def check_balance(self):
        print(f"Current balance for {self.account_holder}: ${self.balance}")

# Example usage:
# Create a bank account
my_account = BankAccount("Alice")

# Deposit some money
my_account.deposit(1000)

# Withdraw some money
my_account.withdraw(500)

# Try to withdraw more than the balance
my_account.withdraw(600)

# Check the balance
my_account.check_balance()
```

```
Deposit of $1000 successful. New balance: $1000
Withdrawal of $500 successful. New balance: $500
Insufficient balance.
Current balance for Alice: $500
```

EXPLANATION :



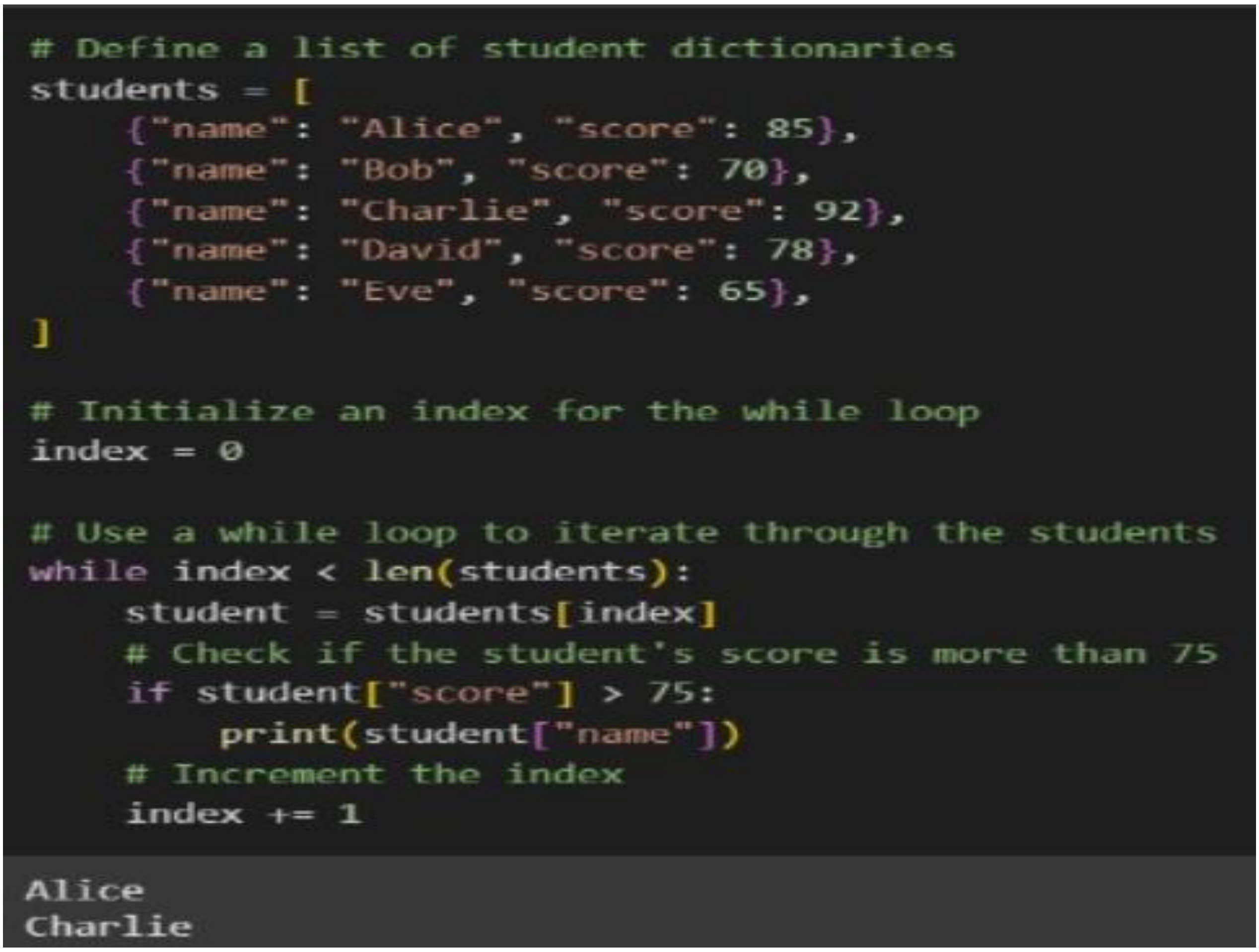
Code Breakdown

The code simulates basic banking operations like depositing, withdrawing, and checking a balance.

- `check_balance(self)` : This function is a method within the `BankAccount` class. It prints the current balance for the account holder using an f-string for easy formatting.
- `my_account = BankAccount("Alice")` : This line creates an instance of the `BankAccount` class and assigns it to the variable `my_account`, with "Alice" as the account holder.
- `my_account.deposit(1000)` : This calls the `deposit` method, adding $1000 to the account. The output `Deposit of $1000 successful. New balance: $1000` confirms this.
- `my_account.withdraw(500)` : This calls the `withdraw` method, subtracting $500. The output `Withdrawal of $500 successful. New balance: $500` shows the successful transaction.
- `my_account.withdraw(600)` : This line attempts to withdraw $600 from the current balance of $500. The `BankAccount` class is programmed to prevent this, as confirmed by the output `Insufficient balance.`
- `my_account.check_balance()` : This final call to the `check_balance` method prints the current balance, which remains at $500, since the last withdrawal attempt failed.

In summary, the code demonstrates how to create an object, call its methods to perform actions, and handle a common error case (insufficient funds) within a simple program.

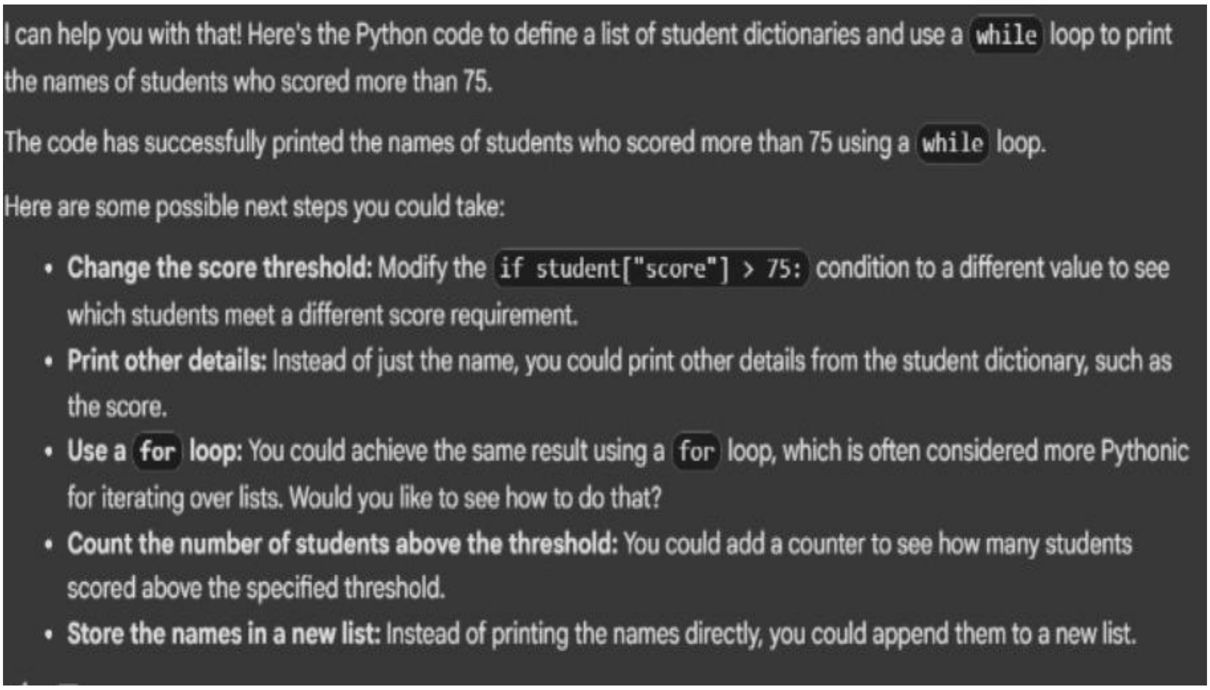
TASK 4 :



```python
# Define a list of student dictionaries
students = [
    {"name": "Alice", "score": 85},
    {"name": "Bob", "score": 70},
    {"name": "Charlie", "score": 92},
    {"name": "David", "score": 78},
    {"name": "Eve", "score": 65},
]

# Initialize an index for the while loop
index = 0

# Use a while loop to iterate through the students
while index < len(students):
    student = students[index]
    # Check if the student's score is more than 75
    if student["score"] > 75:
        print(student["name"])
    # Increment the index
    index += 1
```

```
Alice
Charlie
```

EXPLANATION :



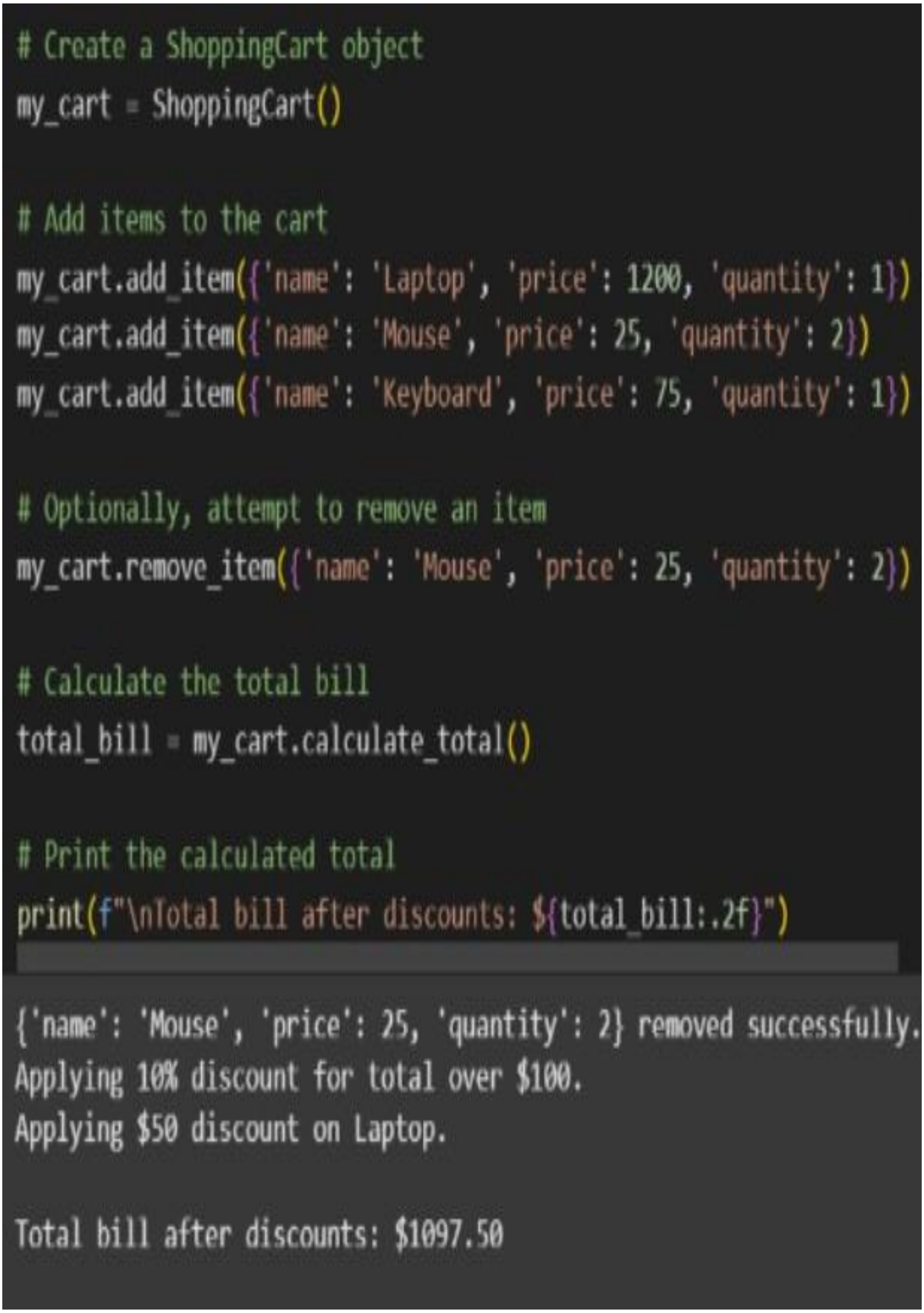
I can help you with that! Here's the Python code to define a list of student dictionaries and use a `while` loop to print the names of students who scored more than 75.

The code has successfully printed the names of students who scored more than 75 using a `while` loop.

Here are some possible next steps you could take:

- **Change the score threshold:** Modify the `if student["score"] > 75:` condition to a different value to see which students meet a different score requirement.
- **Print other details:** Instead of just the name, you could print other details from the student dictionary, such as the score.
- **Use a `for` loop:** You could achieve the same result using a `for` loop, which is often considered more Pythonic for iterating over lists. Would you like to see how to do that?
- **Count the number of students above the threshold:** You could add a counter to see how many students scored above the specified threshold.
- **Store the names in a new list:** Instead of printing the names directly, you could append them to a new list.

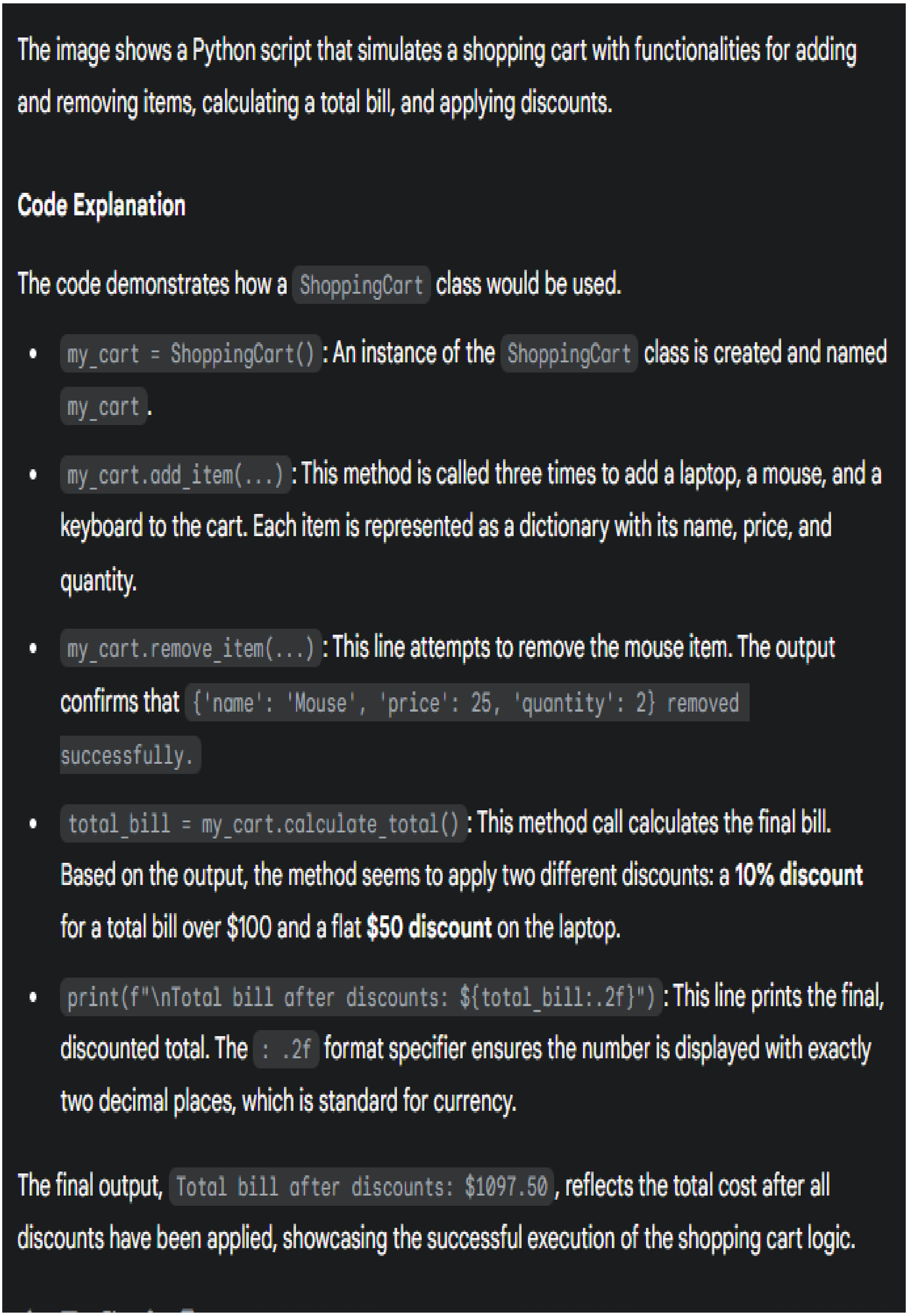
TASK 5 :



```
# Create a ShoppingCart object
my_cart = ShoppingCart()

# Add items to the cart
my_cart.add_item({'name': 'Laptop', 'price': 1200, 'quantity': 1})
my_cart.add_item({'name': 'Mouse', 'price': 25, 'quantity': 2})
my_cart.add_item({'name': 'Keyboard', 'price': 75, 'quantity': 1})

# Optionally, attempt to remove an item
my_cart.remove_item({'name': 'Mouse', 'price': 25, 'quantity': 2})

# Calculate the total bill
total_bill = my_cart.calculate_total()

# Print the calculated total
print(f"\nTotal bill after discounts: ${total_bill:.2f}")
```

```
{'name': 'Mouse', 'price': 25, 'quantity': 2} removed successfully.
Applying 10% discount for total over $100.
Applying $50 discount on Laptop.

Total bill after discounts: $1097.50
```

EXPLANATION :



The image shows a Python script that simulates a shopping cart with functionalities for adding and removing items, calculating a total bill, and applying discounts.

Code Explanation

The code demonstrates how a `ShoppingCart` class would be used.

- `my_cart = ShoppingCart()` : An instance of the `ShoppingCart` class is created and named `my_cart`.
- `my_cart.add_item(...)` : This method is called three times to add a laptop, a mouse, and a keyboard to the cart. Each item is represented as a dictionary with its name, price, and quantity.
- `my_cart.remove_item(...)` : This line attempts to remove the mouse item. The output confirms that `{'name': 'Mouse', 'price': 25, 'quantity': 2} removed successfully.`
- `total_bill = my_cart.calculate_total()` : This method call calculates the final bill. Based on the output, the method seems to apply two different discounts: a **10% discount** for a total bill over $100 and a flat **$50 discount** on the laptop.
- `print(f"\nTotal bill after discounts: ${total_bill:.2f}")` : This line prints the final, discounted total. The `:.2f` format specifier ensures the number is displayed with exactly two decimal places, which is standard for currency.

The final output, `Total bill after discounts: $1097.50`, reflects the total cost after all discounts have been applied, showcasing the successful execution of the shopping cart logic.