**HTNO:2403A52134**

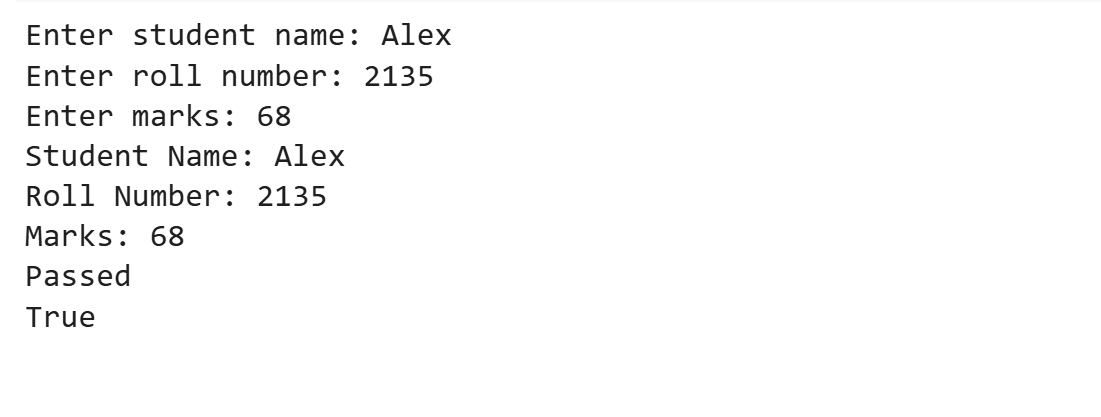
**Assignment-6:**

**Task-1**:

Code:



Output:

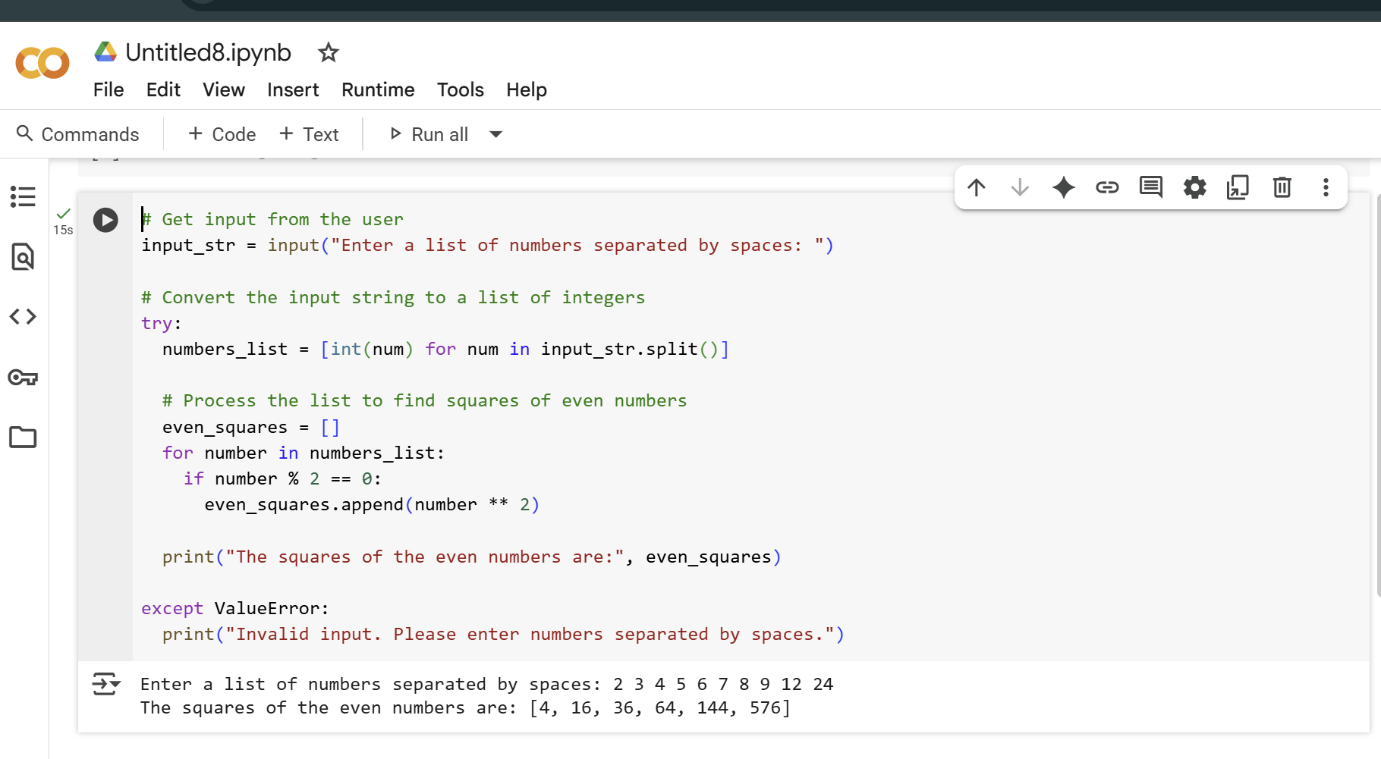


Explanation:

* class Student:: This line starts the definition of a class named Student.
* def \_\_init\_\_(self, name, roll\_number, marks, address=None):: This is the constructor method (\_\_init\_\_) of the class. It's called when you create a new Student object. It takes name, roll\_number, and marks as required arguments, and an optional address argument which defaults to None.
* self.name = name: This line assigns the value passed for name to the name attribute of the Student object.
* self.roll\_number = roll\_number: This line assigns the value passed for roll\_number to the roll\_number attribute of the Student object.
* self.marks = marks: This line assigns the value passed for marks to the marks attribute of the Student object.
* self.address = address # Added address attribute: This line assigns the value passed for address to the address attribute of the Student object.
* def display\_details(self):: This defines a method named display\_details for the Student class. The self parameter refers to the instance of the class.
* print(f"Student Name: {self.name}"): This line prints the student's name using an f-string to embed the self.name attribute.
* print(f"Roll Number: {self.roll\_number}"): This line prints the student's roll number.
* print(f"Marks: {self.marks}"): This line prints the student's marks.
* if self.address:: This checks if the address attribute is not None.
* print(f"Address: {self.address}") # Display address if available: If the address is not None, this line prints the student's address.
* def is\_passed(self):: This defines a method named is\_passed.
* average\_marks = 50 # Example average: This line sets a variable average\_marks to 50.
* if self.marks > average\_marks:: This line checks if the student's marks are greater than average\_marks.
* print("Passed"): If the condition is true, it prints "Passed".
* return True: If the condition is true, it returns True.
* else:: If the condition in the if statement is false, the code in this block is executed.
* print("Failed"): If the condition is false, it prints "Failed".
* return False: If the condition is false, it returns False.
* def update\_marks(self, new\_marks): # Added method to update marks: This defines a new method named update\_marks that takes new\_marks as an argument.
* self.marks = new\_marks: This line updates the marks attribute of the Student object with the value of new\_marks.
* print(f"Marks updated to: {self.marks}"): This line prints a confirmation that the marks have been updated.
* # Example usage with the updated class: This is a comment indicating the start of example usage.
* student3 = Student("Charlie", "C003", 60, "123 Main St"): This line creates a new Student object named student3 with a name, roll number, marks, and an address.
* student3.display\_details(): This calls the display\_details method on student3.
* student3.update\_marks(85): This calls the update\_marks method on student3 to change the marks to 85.
* student3.is\_passed(): This calls the is\_passed method on student3 to check if they passed after the marks update.

**Task-2:**

Code & Output:



Explanation:

# Get input from the user  
input\_str = input("Enter a list of numbers separated by spaces: ")

This line prompts the user to enter a list of numbers separated by spaces and stores their input as a string in the variable input\_str.

# Convert the input string to a list of integers  
try:  
  numbers\_list = [int(num) for num in input\_str.split()]

This is a try block, which is used to handle potential errors. Inside the try block:

* input\_str.split() splits the input string into a list of substrings based on spaces.
* [int(num) for num in ...] is a list comprehension that attempts to convert each substring in the list to an integer and creates a new list called numbers\_list with these integers.

  # Process the list to find squares of even numbers  
  even\_squares = []  
  for number in numbers\_list:  
    if number % 2 == 0:  
      even\_squares.append(number \*\* 2)

This section processes the numbers\_list:

* even\_squares = [] initializes an empty list called even\_squares to store the squares of the even numbers.
* for number in numbers\_list: starts a for loop that iterates through each number in the numbers\_list.
* if number % 2 == 0: checks if the current number is even by using the modulo operator (%). If the remainder when divided by 2 is 0, the number is even.
* even\_squares.append(number \*\* 2) if the number is even, its square (number \*\* 2) is added to the even\_squares list.

  print("The squares of the even numbers are:", even\_squares)

After the loop finishes, this line prints the final even\_squares list to the console, along with a descriptive message.

except ValueError:  
  print("Invalid input. Please enter numbers separated by spaces.")

This is the except block, which catches a ValueError. If the code inside the try block fails to convert any of the input substrings to an integer (e.g., if the user enters text instead of numbers), a ValueError occurs, and this block is executed, printing an error message.

**Task-3:**

Code :

class BankAccount:

    def \_\_init\_\_(self, account\_holder, balance=0):

        self.account\_holder = account\_holder

        self.balance = balance

        print(f"Account created for {self.account\_holder} with initial balance: ${self.balance:.2f}")

    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 balance.")

        else:

            print("Withdrawal amount must be positive.")

# --- User Interaction ---

# Create a bank account instance

account\_name = input("Enter account holder's name: ")

my\_account = BankAccount(account\_name)

# Get deposit amount from user and make a deposit

try:

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

    my\_account.deposit(deposit\_amount)

except ValueError:

    print("Invalid input for deposit amount.")

# Get withdrawal amount from user and make a withdrawal

try:

    withdrawal\_amount = float(input("Enter withdrawal amount: "))

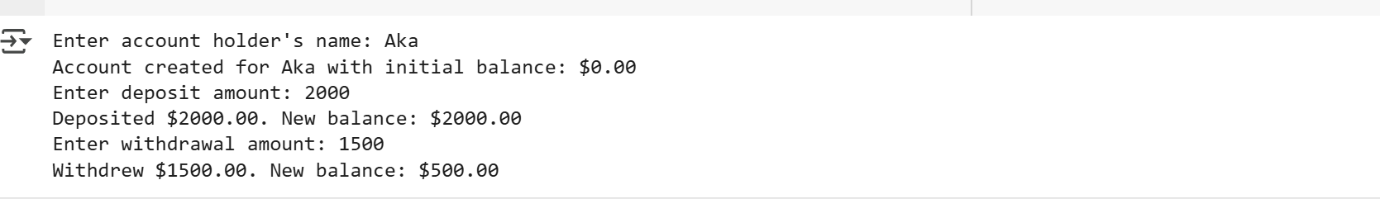
    my\_account.withdraw(withdrawal\_amount)

except ValueError:

    print("Invalid input for withdrawal amount.")

# You can add more interactions here if you like

Output:



Explanation:

lass BankAccount:

This line defines a new class named BankAccount. Classes are blueprints for creating objects (instances).

    def \_\_init\_\_(self, account\_holder, balance=0):

This is the constructor method (\_\_init\_\_). It's a special method that is automatically called when you create a new BankAccount object.

* self refers to the instance of the class being created.
* account\_holder is a parameter to store the name of the account holder.
* balance=0 is a parameter for the initial balance, with a default value of 0 if no balance is provided.

        self.account\_holder = account\_holder  
        self.balance = balance

These lines assign the values passed to the constructor to the account\_holder and balance attributes of the BankAccount object.

        print(f"Account created for {self.account\_holder} with initial balance: ${self.balance:.2f}")

This line prints a confirmation message to the console indicating that the account has been created, including the account holder's name and initial balance formatted to two decimal places.

    def deposit(self, amount):

This defines a method named deposit within the BankAccount class. It takes self (the instance) and amount as parameters.

        if amount > 0:

This line checks if the deposit amount is a positive number.

            self.balance += amount

If the amount is positive, this line adds the amount to the current balance of the account.

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

This line prints a confirmation message showing the deposited amount and the new balance.

        else:  
            print("Deposit amount must be positive.")

If the deposit amount is not positive, this line prints an error message.

    def withdraw(self, amount):

This defines a method named withdraw within the BankAccount class. It takes self (the instance) and amount as parameters.

        if amount > 0:

This line checks if the withdrawal amount is a positive number.

            if self.balance >= amount:

If the amount is positive, this line checks if the current balance is greater than or equal to the withdrawal amount.

                self.balance -= amount

If the balance is sufficient, this line subtracts the amount from the current balance.

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

This line prints a confirmation message showing the withdrawn amount and the new balance.

            else:  
                print("Insufficient balance.")

If the balance is insufficient, this line prints an error message.

        else:  
            print("Withdrawal amount must be positive.")

If the withdrawal amount is not positive, this line prints an error message.

# --- User Interaction ---

This is a comment indicating the start of the user interaction part of the code.

# Create a bank account instance  
account\_name = input("Enter account holder's name: ")  
my\_account = BankAccount(account\_name)

These lines prompt the user to enter the account holder's name, store it in the account\_name variable, and then create a new BankAccount object named my\_account using the provided name.

# Get deposit amount from user and make a deposit  
try:  
    deposit\_amount = float(input("Enter deposit amount: "))  
    my\_account.deposit(deposit\_amount)  
except ValueError:  
    print("Invalid input for deposit amount.")

This block handles user input for the deposit amount:

* It prompts the user to enter the deposit amount.
* It attempts to convert the input to a floating-point number using float().
* If successful, it calls the deposit method of the my\_account object with the entered amount.
* The try...except ValueError block catches errors if the user enters something that cannot be converted to a number and prints an error message.

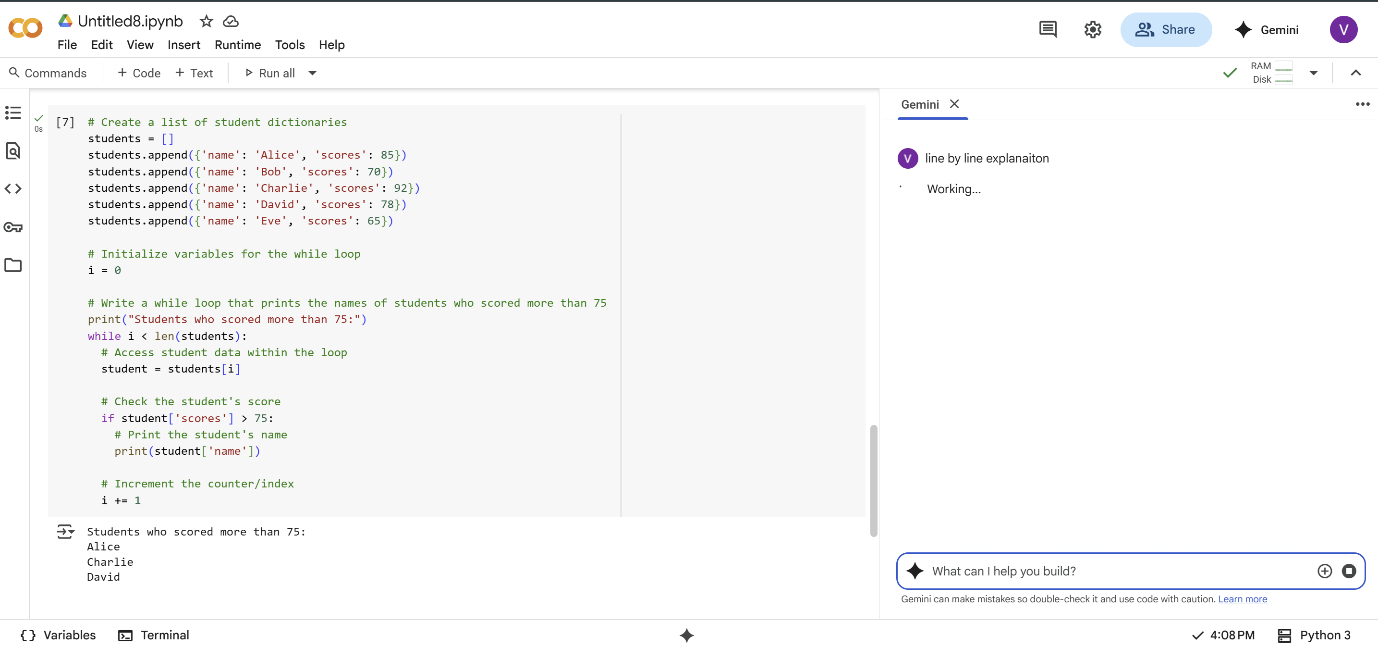
# Get withdrawal amount from user and make a withdrawal  
try:  
    withdrawal\_amount = float(input("Enter withdrawal amount: "))  
    my\_account.withdraw(withdrawal\_amount)  
except ValueError:  
    print("Invalid input for withdrawal amount.")

This block handles user input for the withdrawal amount in a similar way to the deposit block:

* It prompts the user to enter the withdrawal amount.
* It attempts to convert the input to a floating-point number.
* If successful, it calls the withdraw method of the my\_account object with the entered amount.
* The try...except ValueError block catches errors if the user enters something that cannot be converted to a number and prints an error message.

**Task-4:**

Code & Output:



Explanation:

# Create a list of student dictionaries  
students = []

This section starts with a comment indicating the purpose. The line students = [] initializes an empty list called students. This list will hold dictionaries, where each dictionary represents a student.

students.append({'name': 'Alice', 'scores': 85})  
students.append({'name': 'Bob', 'scores': 70})  
students.append({'name': 'Charlie', 'scores': 92})  
students.append({'name': 'David', 'scores': 78})  
students.append({'name': 'Eve', 'scores': 65})

These lines add dictionaries to the students list. Each dictionary has two key-value pairs: 'name' for the student's name and 'scores' for their score. This populates the list with example student data.

# Initialize variables for the while loop  
i = 0

This section prepares for the while loop. The line i = 0 initializes a variable i to 0. This variable will serve as an index to iterate through the students list.

# Write a while loop that prints the names of students who scored more than 75  
print("Students who scored more than 75:")

This comment describes the purpose of the following code block. The print() statement outputs a header to the console before the list of names is printed.

while i < len(students):

This is the while loop. It will continue to execute the code block below it as long as the condition i < len(students) is true. len(students) returns the number of elements in the students list, so this condition ensures the loop iterates through all elements using the index i.

  # Access student data within the loop  
  student = students[i]

Inside the loop, this line accesses the dictionary at the current index i from the students list and assigns it to the variable student.

  # Check the student's score  
  if student['scores'] > 75:

This line checks the value associated with the key 'scores' in the current student dictionary. If the score is greater than 75, the code block inside the if statement will be executed.

    # Print the student's name  
    print(student['name'])

If the student's score is greater than 75, this line prints the value associated with the key 'name' from the current student dictionary, which is the student's name.

  # Increment the counter/index  
  i += 1

This line increments the value of i by 1. This is crucial for the while loop to eventually terminate as it moves the index to the next student in the list in each iteration.

**Task-5:**

Code:

class ShoppingCart:

    def \_\_init\_\_(self):

        self.items = []

    def add\_item(self, item\_name, price, quantity=1):

        """Adds an item to the shopping cart."""

        self.items.append({"name": item\_name, "price": price, "quantity": quantity})

        print(f"Added {quantity} x {item\_name} to the cart.")

    def remove\_item(self, item\_name):

        """Removes an item from the shopping cart."""

        initial\_item\_count = len(self.items)

        self.items = [item for item in self.items if item["name"] != item\_name]

        if len(self.items) < initial\_item\_count:

            print(f"Removed {item\_name} from the cart.")

        else:

            print(f"{item\_name} not found in the cart.")

    def calculate\_total\_bill(self):

        """Calculates the total bill with conditional discounts."""

        total\_bill = 0

        for item in self.items:

            item\_total = item["price"] \* item["quantity"]

            # Apply conditional discounts

            if item["quantity"] >= 5:

                item\_total \*= 0.9  # 10% discount for buying 5 or more of an item

                print(f"Applied 10% discount on {item['name']} for buying {item['quantity']}.")

            elif item["quantity"] >= 3:

                item\_total \*= 0.95  # 5% discount for buying 3 or 4 of an item

                print(f"Applied 5% discount on {item['name']} for buying {item['quantity']}.")

            total\_bill += item\_total

        # Further conditional discounts based on total bill

        if total\_bill > 100:

            total\_bill \*= 0.9  # 10% discount for total bill over $100

            print("Applied 10% discount on total bill over $100.")

        elif total\_bill > 50:

            total\_bill \*= 0.95  # 5% discount for total bill over $50

            print("Applied 5% discount on total bill over $50.")

        return total\_bill

# Example Usage:

cart = ShoppingCart()

cart.add\_item("Laptop", 1000, 1)

cart.add\_item("Mouse", 25, 3)

cart.add\_item("Keyboard", 75, 5)

cart.add\_item("Monitor", 200, 2)

print("\nItems in cart:")

for item in cart.items:

    print(f"- {item['name']}: ${item['price']} x {item['quantity']}")

total = cart.calculate\_total\_bill()

print(f"\nTotal bill: ${total:.2f}")

cart.remove\_item("Mouse")

print("\nItems in cart after removal:")

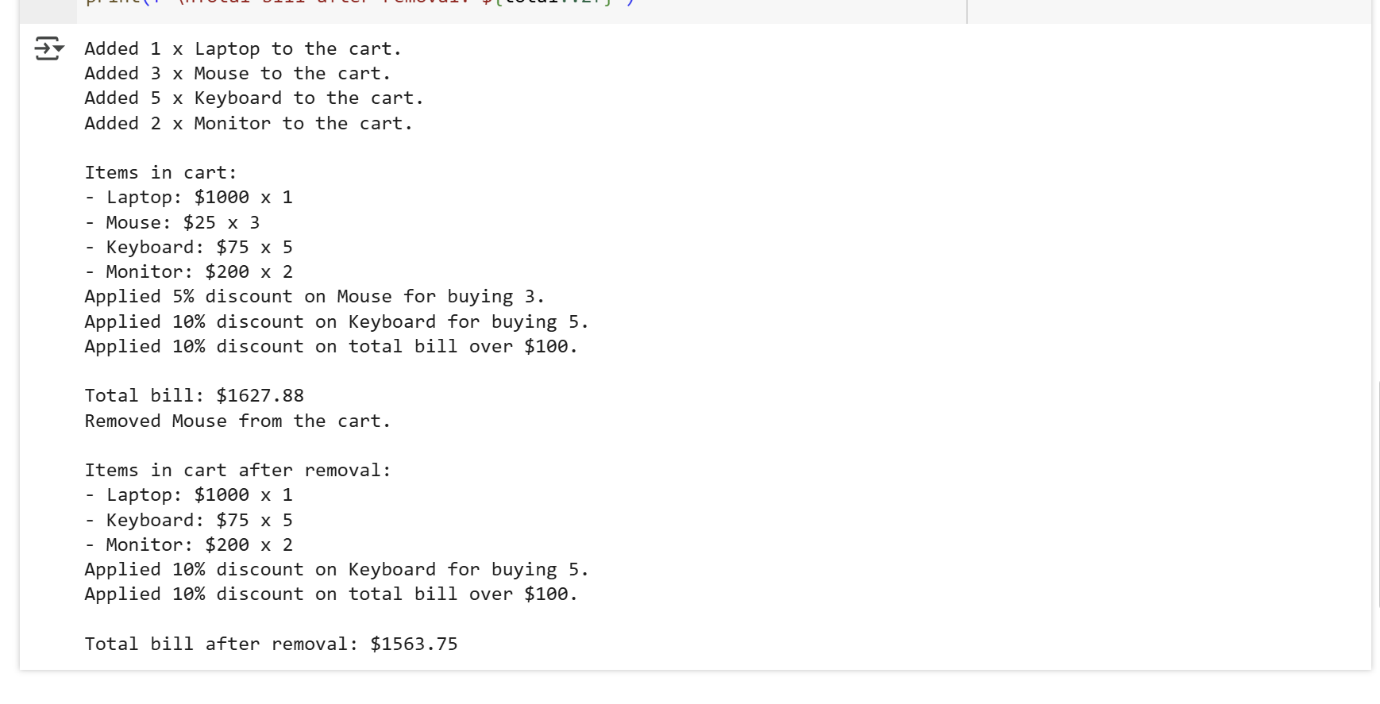
for item in cart.items:

    print(f"- {item['name']}: ${item['price']} x {item['quantity']}")

total = cart.calculate\_total\_bill()

print(f"\nTotal bill after removal: ${total:.2f}")

Output:



Explanation:

class ShoppingCart:  
    def \_\_init\_\_(self):  
        self.items = []

This defines a class named ShoppingCart. The \_\_init\_\_ method is the constructor, which is called when you create a new ShoppingCart object. It initializes an empty list called items to store the items in the cart.

    def add\_item(self, item\_name, price, quantity=1):  
        """Adds an item to the shopping cart."""  
        self.items.append({"name": item\_name, "price": price, "quantity": quantity})  
        print(f"Added {quantity} x {item\_name} to the cart.")

This defines a method called add\_item which takes the item's name, price, and an optional quantity (defaulting to 1) as arguments. It appends a dictionary representing the item to the items list and prints a confirmation message.

    def remove\_item(self, item\_name):  
        """Removes an item from the shopping cart."""  
        initial\_item\_count = len(self.items)  
        self.items = [item for item in self.items if item["name"] != item\_name]  
        if len(self.items) < initial\_item\_count:  
            print(f"Removed {item\_name} from the cart.")  
        else:  
            print(f"{item\_name} not found in the cart.")

This defines the remove\_item method, which takes an item's name as input. It creates a new list self.items containing all items except the one to be removed using a list comprehension. It then checks if any items were actually removed and prints a corresponding message.

    def calculate\_total\_bill(self):  
        """Calculates the total bill with conditional discounts."""  
        total\_bill = 0  
        for item in self.items:  
            item\_total = item["price"] \* item["quantity"]  
            # Apply conditional discounts  
            if item["quantity"] >= 5:  
                item\_total \*= 0.9  # 10% discount for buying 5 or more of an item  
                print(f"Applied 10% discount on {item['name']} for buying {item['quantity']}.")  
            elif item["quantity"] >= 3:  
                item\_total \*= 0.95  # 5% discount for buying 3 or 4 of an item  
                print(f"Applied 5% discount on {item['name']} for buying {item['quantity']}.")  
  
            total\_bill += item\_total

This defines the calculate\_total\_bill method. It initializes total\_bill to 0. It then iterates through each item in the self.items list. Inside the loop, it calculates the item\_total and applies conditional discounts based on the item's quantity (10% for 5 or more, 5% for 3 or 4).

        # Further conditional discounts based on total bill  
        if total\_bill > 100:  
            total\_bill \*= 0.9  # 10% discount for total bill over $100  
            print("Applied 10% discount on total bill over $100.")  
        elif total\_bill > 50:  
            total\_bill \*= 0.95  # 5% discount for total bill over $50  
            print("Applied 5% discount on total bill over $50.")  
  
        return total\_bill

This part of the calculate\_total\_bill method applies further conditional discounts based on the running total\_bill. If the total is over $100, a 10% discount is applied. If it's over $$100, a 10% discount is applied. If it's over $50 (but not over $100), a 5% discount is applied. Finally, it returns the calculated total\_bill.

# Example Usage:  
cart = ShoppingCart()  
cart.add\_item("Laptop", 1000, 1)  
cart.add\_item("Mouse", 25, 3)  
cart.add\_item("Keyboard", 75, 5)  
cart.add\_item("Monitor", 200, 2)

This is the example usage section. It creates an instance of the ShoppingCart class and adds several items with different quantities and prices to demonstrate the functionality.

print("\nItems in cart:")  
for item in cart.items:  
    print(f"- {item['name']}: ${item['price']} x {item['quantity']}")

This prints a header "Items in cart:" and then iterates through the cart.items list, printing the name, price, and quantity of each item.

total = cart.calculate\_total\_bill()  
print(f"\nTotal bill: ${total:.2f}")

This calls the calculate\_total\_bill method to get the total bill and prints it, formatted to two decimal places.

cart.remove\_item("Mouse")

This calls the remove\_item method to remove the item named "Mouse" from the cart.

print("\nItems in cart after removal:")  
for item in cart.items:  
    print(f"- {item['name']}: ${item['price']} x {item['quantity']}")

This prints a header "Items in cart after removal:" and then iterates through the updated cart.items list, printing the details of the remaining items.

total = cart.calculate\_total\_bill()  
print(f"\nTotal bill after removal: ${total:.2f}")

Finally, this calls calculate\_total\_bill again to get the total bill after removing an item and prints the new total, formatted to two decimal places.