AI ASSISTED CODING

NAME : N.JyoshnaSri

HALL.NO:2403A52106

LAB ASSIGNMENT – 6.4:

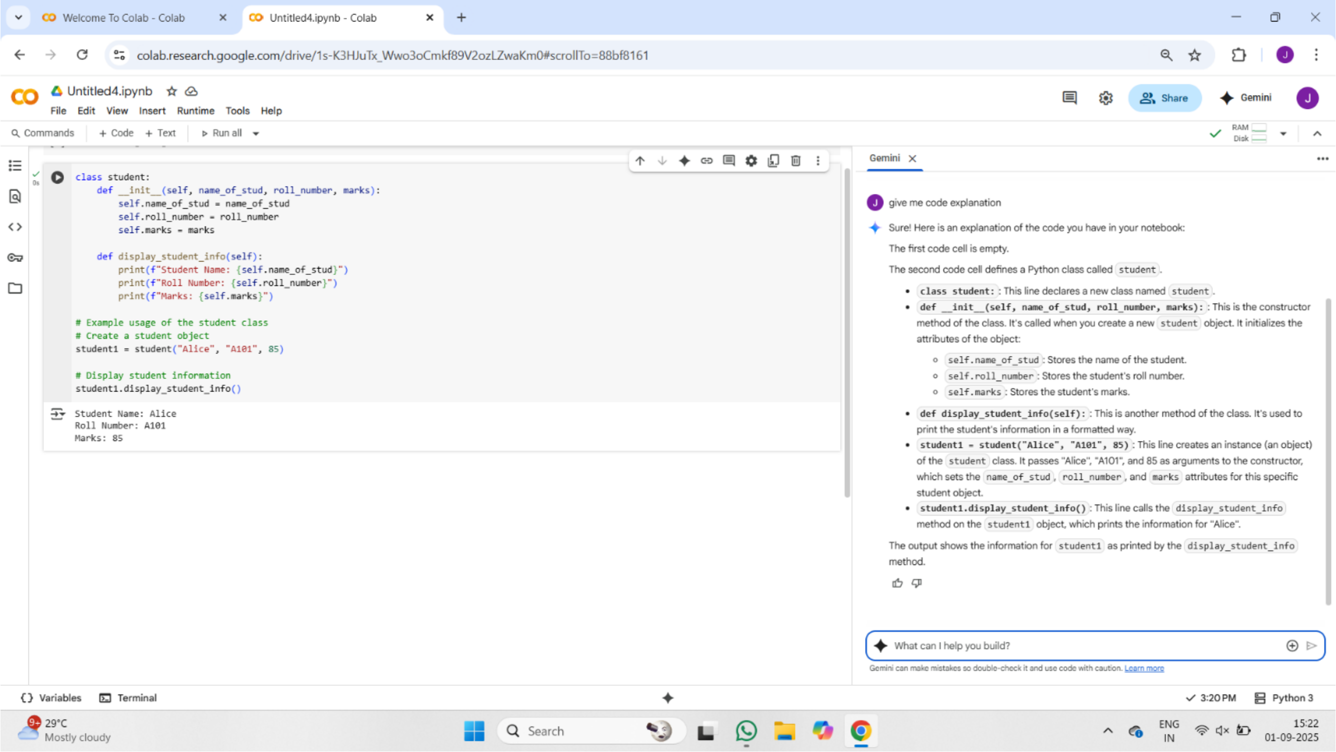
TASK – 1:

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

#prompt used here:

write a python program to create a class named student with attributes name of stud,roll_number and marks

Code screenshots:



```
class student:
    def __init__(self, name_of_stud, roll_number, marks):
        self.name_of_stud = name_of_stud
        self.roll_number = roll_number
        self.marks = marks

    def display_student_info(self):
        print(f"Student Name: {self.name_of_stud}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Marks: {self.marks}")

# Example usage of the student class
# Create a student object
student1 = student("Alice", "A101", 85)

# Display student information
student1.display_student_info()
```

```
Student Name: Alice
Roll Number: A101
Marks: 85
```

give me code explanation

Sure! Here is an explanation of the code you have in your notebook:

The first code cell is empty.

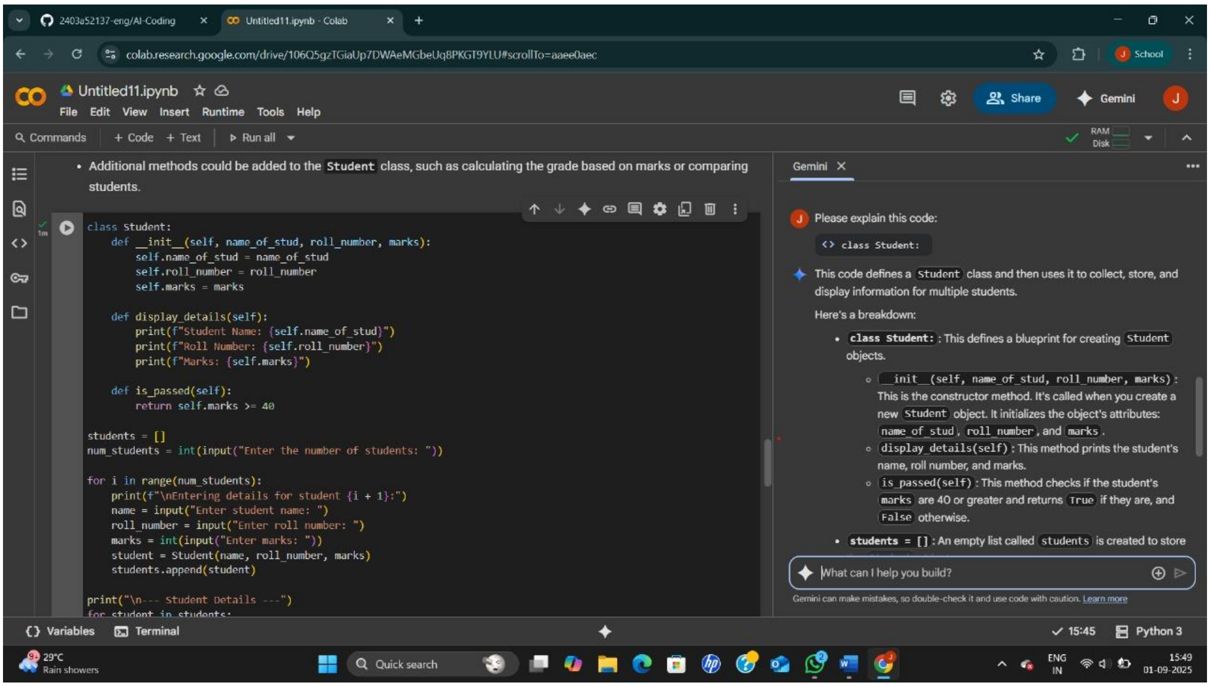
The second code cell defines a Python class called `student`.

- `class student:` : This line declares a new class named `student`.
- `def __init__(self, name_of_stud, roll_number, marks):` : This is the constructor method of the class. It's called when you create a new `student` object. It initializes the attributes of the object:
  - `self.name_of_stud` : Stores the name of the student.
  - `self.roll_number` : Stores the student's roll number.
  - `self.marks` : Stores the student's marks.
- `def display_student_info(self):` : This is another method of the class. It's used to print the student's information in a formatted way.
- `student1 = student("Alice", "A101", 85)` : This line creates an instance (an object) of the `student` class. It passes "Alice", "A101", and 85 as arguments to the constructor, which sets the `name_of_stud`, `roll_number`, and `marks` attributes for this specific student object.
- `student1.display_student_info()` : This line calls the `display_student_info` method on the `student1` object, which prints the information for "Alice".

The output shows the information for `student1` as printed by the `display_student_info` method.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more



- Additional methods could be added to the `Student` class, such as calculating the grade based on marks or comparing students.

```
class Student:
    def __init__(self, name_of_stud, roll_number, marks):
        self.name_of_stud = name_of_stud
        self.roll_number = roll_number
        self.marks = marks

    def display_details(self):
        print(f"Student Name: {self.name_of_stud}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Marks: {self.marks}")

    def is_passed(self):
        return self.marks >= 40

students = []
num_students = int(input("Enter the number of students: "))

for i in range(num_students):
    print(f"\nEntering details for student {i + 1}:")
    name = input("Enter student name: ")
    roll_number = input("Enter roll number: ")
    marks = int(input("Enter marks: "))
    student = Student(name, roll_number, marks)
    students.append(student)

print("\n--- Student Details ---")
for student in students:
```

Please explain this code:

`class Student:`

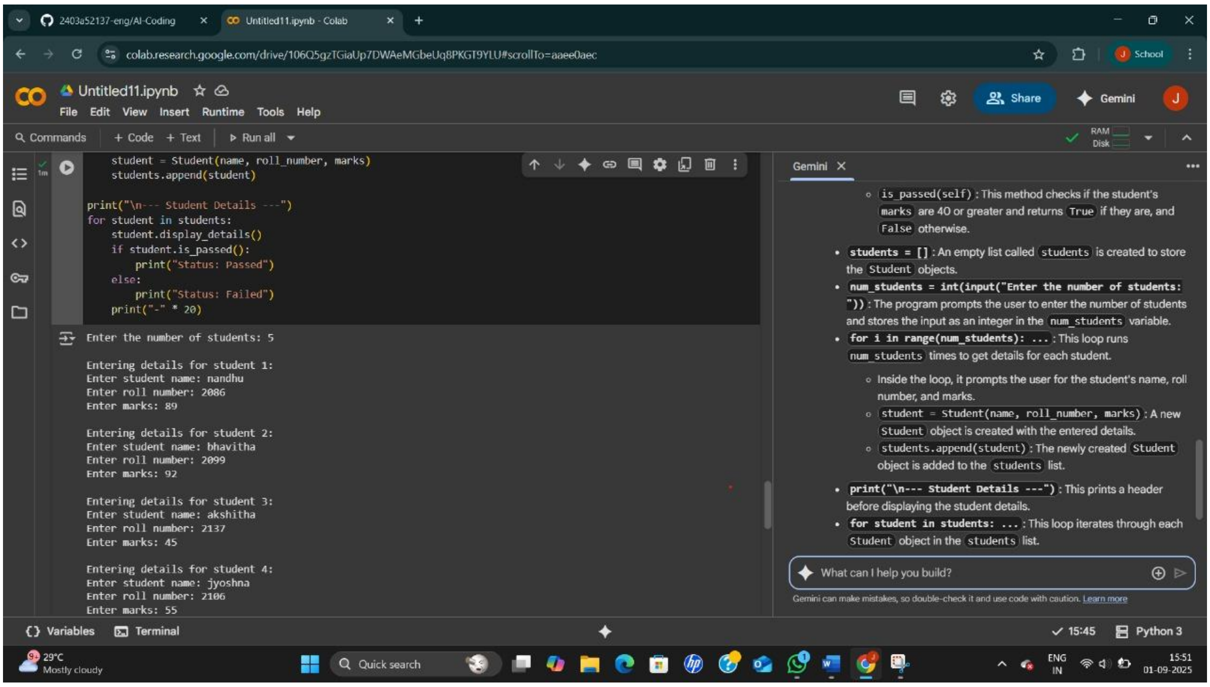
This code defines a `Student` class and then uses it to collect, store, and display information for multiple students.

Here's a breakdown:

- `class Student:` : This defines a blueprint for creating `Student` objects.
  - `__init__(self, name_of_stud, roll_number, marks)` : This is the constructor method. It's called when you create a new `Student` object. It initializes the object's attributes: `name_of_stud`, `roll_number`, and `marks`.
  - `display_details(self)` : This method prints the student's name, roll number, and marks.
  - `is_passed(self)` : This method checks if the student's `marks` are 40 or greater and returns `True` if they are, and `False` otherwise.
- `students = []` : An empty list called `students` is created to store

What can I help you build?

Gemini can make mistakes, so double-check it and use code with caution. Learn more



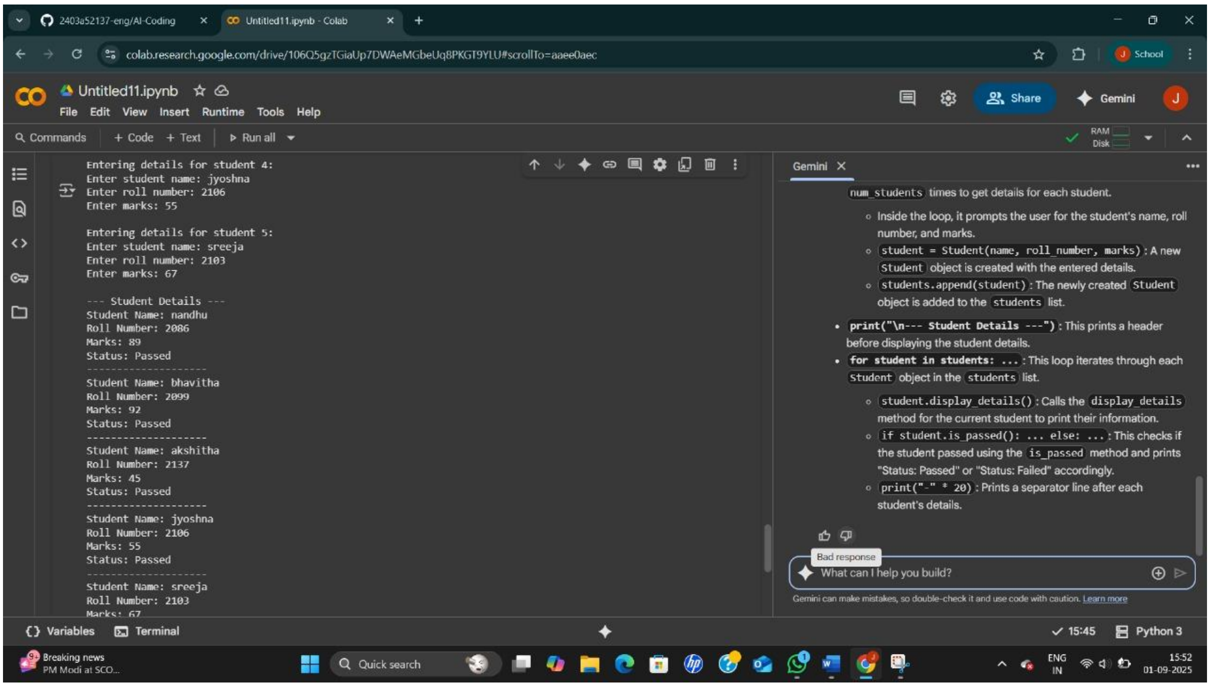
```
    student = Student(name, roll_number, marks)
    students.append(student)

print("\n--- Student Details ---")
for student in students:
    student.display_details()
    if student.is_passed():
        print("Status: Passed")
    else:
        print("Status: Failed")
    print("-" * 20)
```

```
Enter the number of students: 5

Entering details for student 1:
Enter student name: nandhu
Enter roll number: 2086
Enter marks: 89

Entering details for student 2:
Enter student name: bhavitha
Enter roll number: 2099
Enter marks: 92

Entering details for student 3:
Enter student name: akshitha
Enter roll number: 2137
Enter marks: 45

Entering details for student 4:
Enter student name: jyoshna
Enter roll number: 2106
Enter marks: 55
```

  - `is_passed(self)`: This method checks if the student's `marks` are 40 or greater and returns `True` if they are, and `False` otherwise.
- **`students = []`**: An empty list called `students` is created to store the `Student` objects.
- **`num_students = int(input("Enter the number of students: "))`**: The program prompts the user to enter the number of students and stores the input as an integer in the `num_students` variable.
- **`for i in range(num_students): ...`**: This loop runs `num_students` times to get details for each student.
  - Inside the loop, it prompts the user for the student's name, roll number, and marks.
  - `student = Student(name, roll_number, marks)`: A new `Student` object is created with the entered details.
  - `students.append(student)`: The newly created `Student` object is added to the `students` list.
- **`print("\n--- Student Details ---")`**: This prints a header before displaying the student details.
- **`for student in students: ...`**: This loop iterates through each `Student` object in the `students` list.

What can I help you build?

Gemini can make mistakes, so double-check it and use code with caution. Learn more

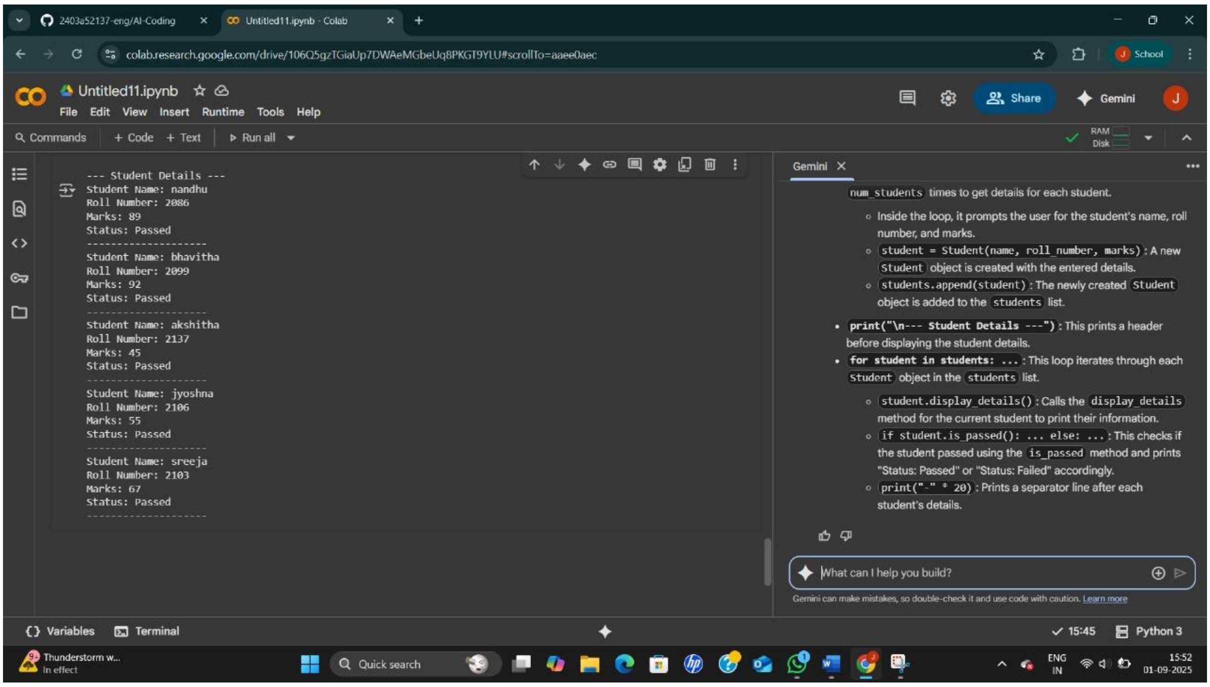


```
Entering details for student 4:
Enter student name: jyoshna
Enter roll number: 2106
Enter marks: 55

Entering details for student 5:
Enter student name: sreeja
Enter roll number: 2103
Enter marks: 67

--- Student Details ---
Student Name: nandhu
Roll Number: 2086
Marks: 89
Status: Passed
--------------------
Student Name: bhavitha
Roll Number: 2099
Marks: 92
Status: Passed
--------------------
Student Name: akshitha
Roll Number: 2137
Marks: 45
Status: Passed
--------------------
Student Name: jyoshna
Roll Number: 2106
Marks: 55
Status: Passed
--------------------
Student Name: sreeja
Roll Number: 2103
Marks: 67
```

`num_students` times to get details for each student.
  - Inside the loop, it prompts the user for the student's name, roll number, and marks.
  - `student = Student(name, roll_number, marks)`: A new `Student` object is created with the entered details.
  - `students.append(student)`: The newly created `Student` object is added to the `students` list.
- **`print("\n--- Student Details ---")`**: This prints a header before displaying the student details.
- **`for student in students: ...`**: This loop iterates through each `Student` object in the `students` list.
  - `student.display_details()`: Calls the `display_details` method for the current student to print their information.
  - `if student.is_passed(): ... else: ...`: This checks if the student passed using the `is_passed` method and prints "Status: Passed" or "Status: Failed" accordingly.
  - `print("-" * 20)`: Prints a separator line after each student's details.

What can I help you build?

Gemini can make mistakes, so double-check it and use code with caution. Learn more



```
--- Student Details ---
Student Name: nandhu
Roll Number: 2086
Marks: 89
Status: Passed
--------------------
Student Name: bhavitha
Roll Number: 2099
Marks: 92
Status: Passed
--------------------
Student Name: akshitha
Roll Number: 2137
Marks: 45
Status: Passed
--------------------
Student Name: jyoshna
Roll Number: 2106
Marks: 55
Status: Passed
--------------------
Student Name: sreeja
Roll Number: 2103
Marks: 67
Status: Passed
--------------------
```

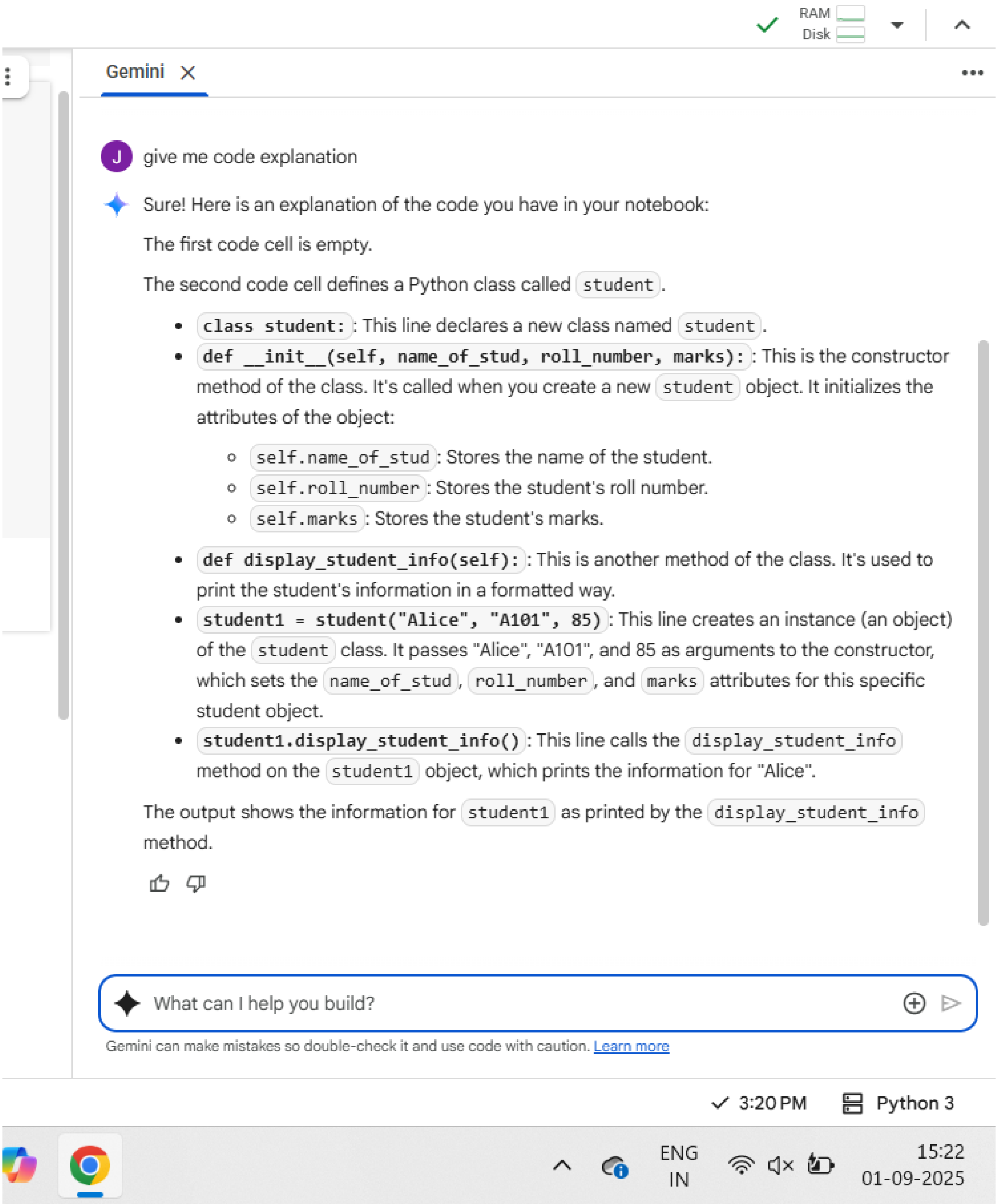
`num_students` times to get details for each student.

  - Inside the loop, it prompts the user for the student's name, roll number, and marks.
  - `student = Student(name, roll_number, marks)`: A new `Student` object is created with the entered details.
  - `students.append(student)`: The newly created `Student` object is added to the `students` list.
- `print("\n--- Student Details ---")`: This prints a header before displaying the student details.
- `for student in students: ...`: This loop iterates through each `Student` object in the `students` list.
  - `student.display_details()`: Calls the `display_details` method for the current student to print their information.
  - `if student.is_passed(): ... else: ...`: This checks if the student passed using the `is_passed` method and prints "Status: Passed" or "Status: Failed" accordingly.
  - `print("-" * 20)`: Prints a separator line after each student's details.

What can I help you build?

Gemini can make mistakes, so double-check it and use code with caution. Learn more

CODE EXPLANATION:



give me code explanation

Sure! Here is an explanation of the code you have in your notebook:

The first code cell is empty.

The second code cell defines a Python class called `student`.

- `class student:`: This line declares a new class named `student`.
- `def __init__(self, name_of_stud, roll_number, marks):`: This is the constructor method of the class. It's called when you create a new `student` object. It initializes the attributes of the object:
    - `self.name_of_stud`: Stores the name of the student.
    - `self.roll_number`: Stores the student's roll number.
    - `self.marks`: Stores the student's marks.
- `def display_student_info(self):`: This is another method of the class. It's used to print the student's information in a formatted way.
- `student1 = student("Alice", "A101", 85)`: This line creates an instance (an object) of the `student` class. It passes "Alice", "A101", and 85 as arguments to the constructor, which sets the `name_of_stud`, `roll_number`, and `marks` attributes for this specific student object.
- `student1.display_student_info()`: This line calls the `display_student_info` method on the `student1` object, which prints the information for "Alice".

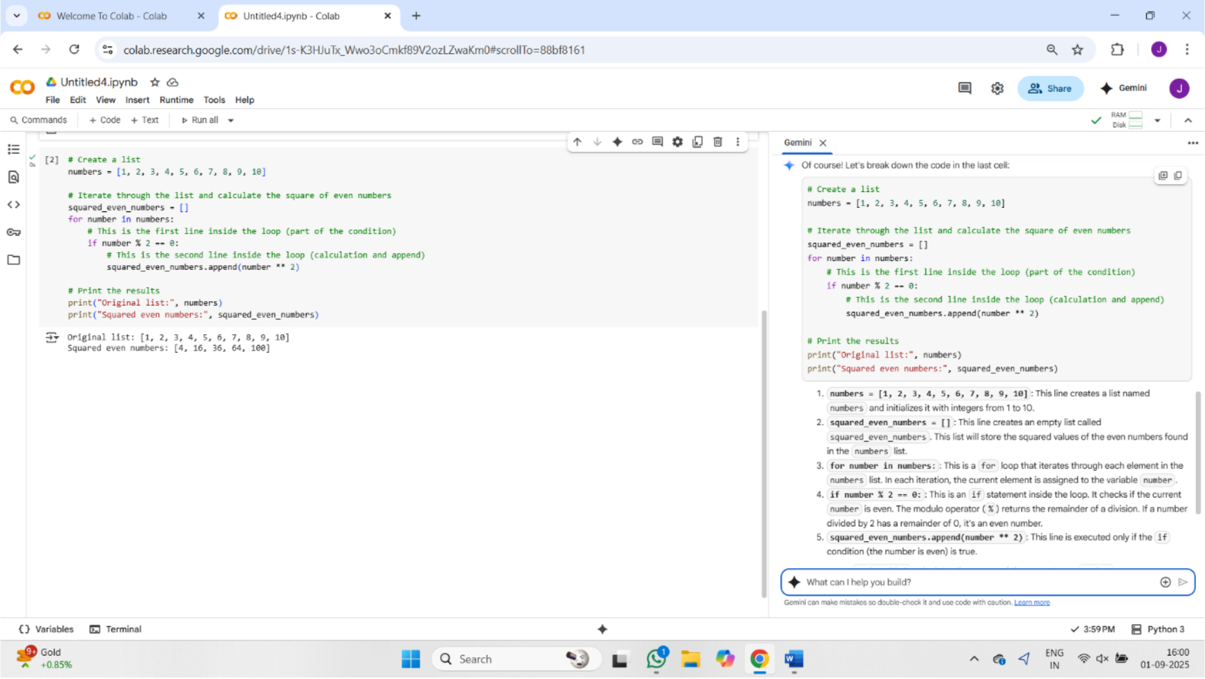
The output shows the information for `student1` as printed by the `display_student_info` method.

Gemini can make mistakes so double-check it and use code with caution. Learn more

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.**

CODE OUTPUT SCREENSHOTS:



```
[2] # Create a list
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

    # Iterate through the list and calculate the square of even numbers
    squared_even_numbers = []
    for number in numbers:
        # This is the first line inside the loop (part of the condition)
        if number % 2 == 0:
            # This is the second line inside the loop (calculation and append)
            squared_even_numbers.append(number ** 2)

    # Print the results
    print("Original list:", numbers)
    print("Squared even numbers:", squared_even_numbers)
```

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

Gemini

Of course! Let's break down the code in the last cell:

```
# Create a list
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Iterate through the list and calculate the square of even numbers
squared_even_numbers = []
for number in numbers:
    # This is the first line inside the loop (part of the condition)
    if number % 2 == 0:
        # This is the second line inside the loop (calculation and append)
        squared_even_numbers.append(number ** 2)

# Print the results
print("Original list:", numbers)
print("Squared even numbers:", squared_even_numbers)
```

1. `numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`: This line creates a list named `numbers` and initializes it with integers from 1 to 10.
2. `squared_even_numbers = []`: This line creates an empty list called `squared_even_numbers`. This list will store the squared values of the even numbers found in the `numbers` list.
3. `for number in numbers:`: This is a `for` loop that iterates through each element in the `numbers` list. In each iteration, the current element is assigned to the variable `number`.
4. `if number % 2 == 0:`: This is an `if` statement inside the loop. It checks if the current `number` is even. The modulo operator (`%`) returns the remainder of a division. If a number divided by 2 has a remainder of 0, it's an even number.
5. `squared_even_numbers.append(number ** 2)`: This line is executed only if the `if` condition (the number is even) is true.

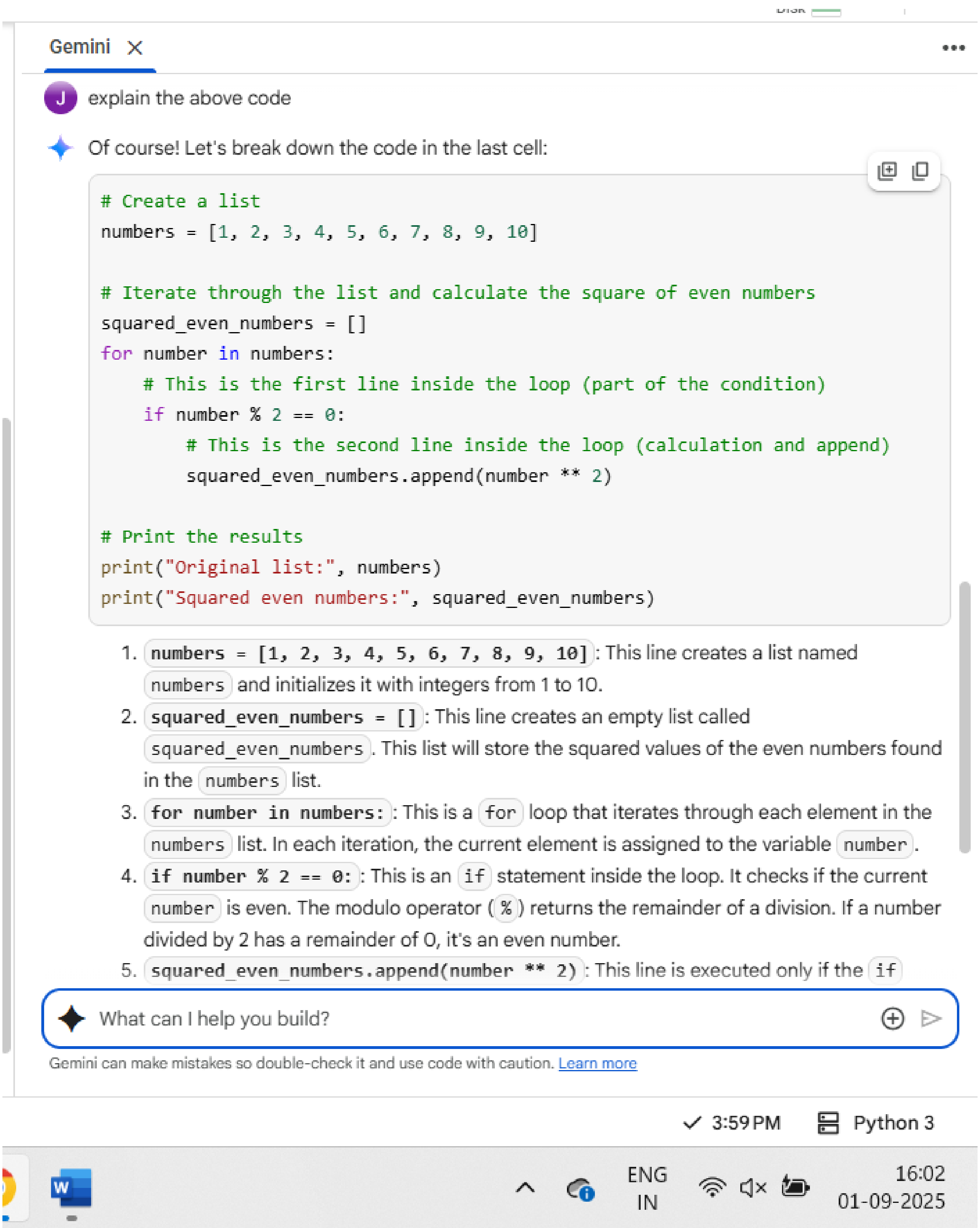
What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

#prompt used :

Write a python program to create a list and write first 2 lines in loop and calculate the square of even numbers only .

Code explanation:



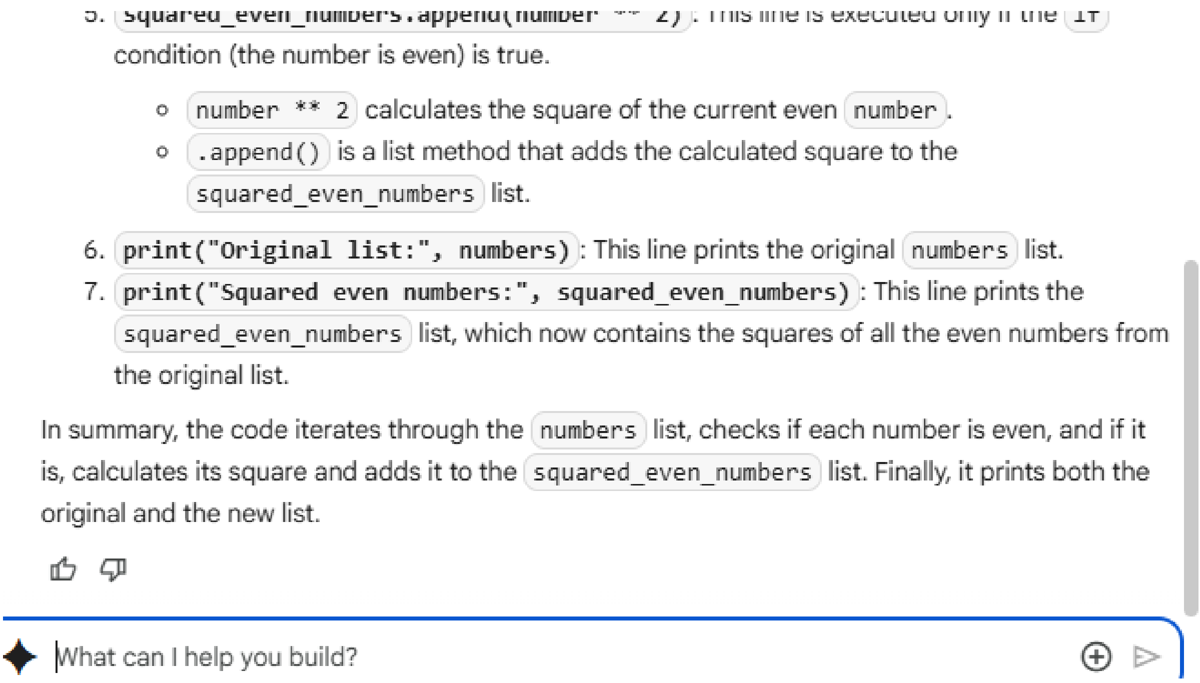
explain the above code

Of course! Let's break down the code in the last cell:

```
# Create a list
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Iterate through the list and calculate the square of even numbers
squared_even_numbers = []
for number in numbers:
    # This is the first line inside the loop (part of the condition)
    if number % 2 == 0:
        # This is the second line inside the loop (calculation and append)
        squared_even_numbers.append(number ** 2)

# Print the results
print("Original list:", numbers)
print("Squared even numbers:", squared_even_numbers)
```

1. `numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`: This line creates a list named `numbers` and initializes it with integers from 1 to 10.
2. `squared_even_numbers = []`: This line creates an empty list called `squared_even_numbers`. This list will store the squared values of the even numbers found in the `numbers` list.
3. `for number in numbers:`: This is a `for` loop that iterates through each element in the `numbers` list. In each iteration, the current element is assigned to the variable `number`.
4. `if number % 2 == 0:`: This is an `if` statement inside the loop. It checks if the current `number` is even. The modulo operator (`%`) returns the remainder of a division. If a number divided by 2 has a remainder of 0, it's an even number.
5. `squared_even_numbers.append(number ** 2)`: This line is executed only if the `if`

Gemini can make mistakes so double-check it and use code with caution. Learn more



5. `squared_even_numbers.append(number ** 2)`: This line is executed only if the `if` condition (the number is even) is true.
    - `number ** 2` calculates the square of the current even `number`.
    - `.append()` is a list method that adds the calculated square to the `squared_even_numbers` list.
6. `print("Original list:", numbers)`: This line prints the original `numbers` list.
7. `print("Squared even numbers:", squared_even_numbers)`: This line prints the `squared_even_numbers` list, which now contains the squares of all the even numbers from the original list.

In summary, the code iterates through the `numbers` list, checks if each number is even, and if it is, calculates its square and adds it to the `squared_even_numbers` list. Finally, it prints both the original and the new list.

What can I help you build?

Task-3:

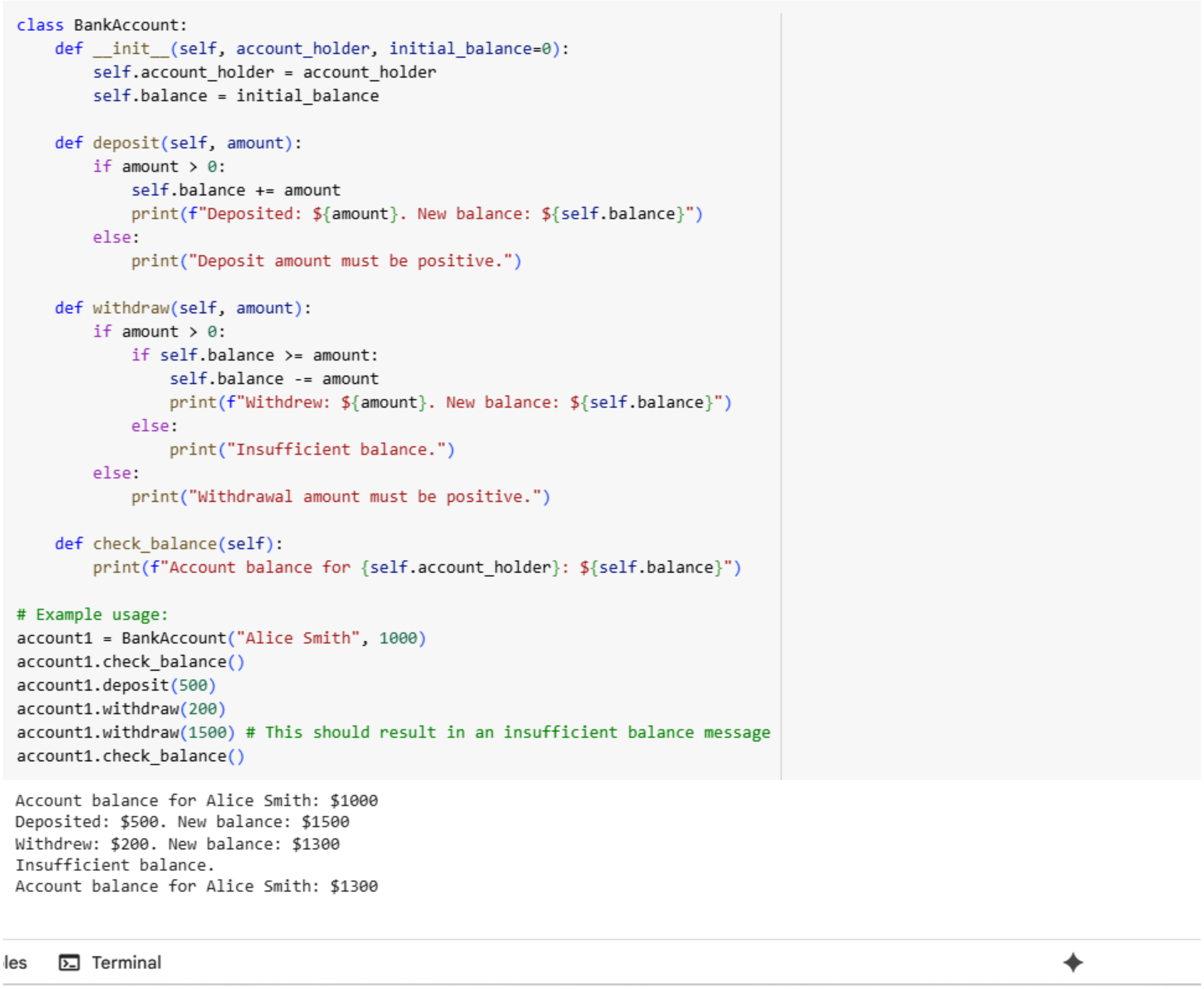
- **Create a class called BankAccount with attributes account_holder and balance. Use Copilot to complete methods for deposit(), withdraw(), and check for insufficient balance.**

#prompt used here:

Write a python program to create a class named bank account with attributes like account_holder,balance.

Write a python program to check deposit,withdrawand check insufficient balance.

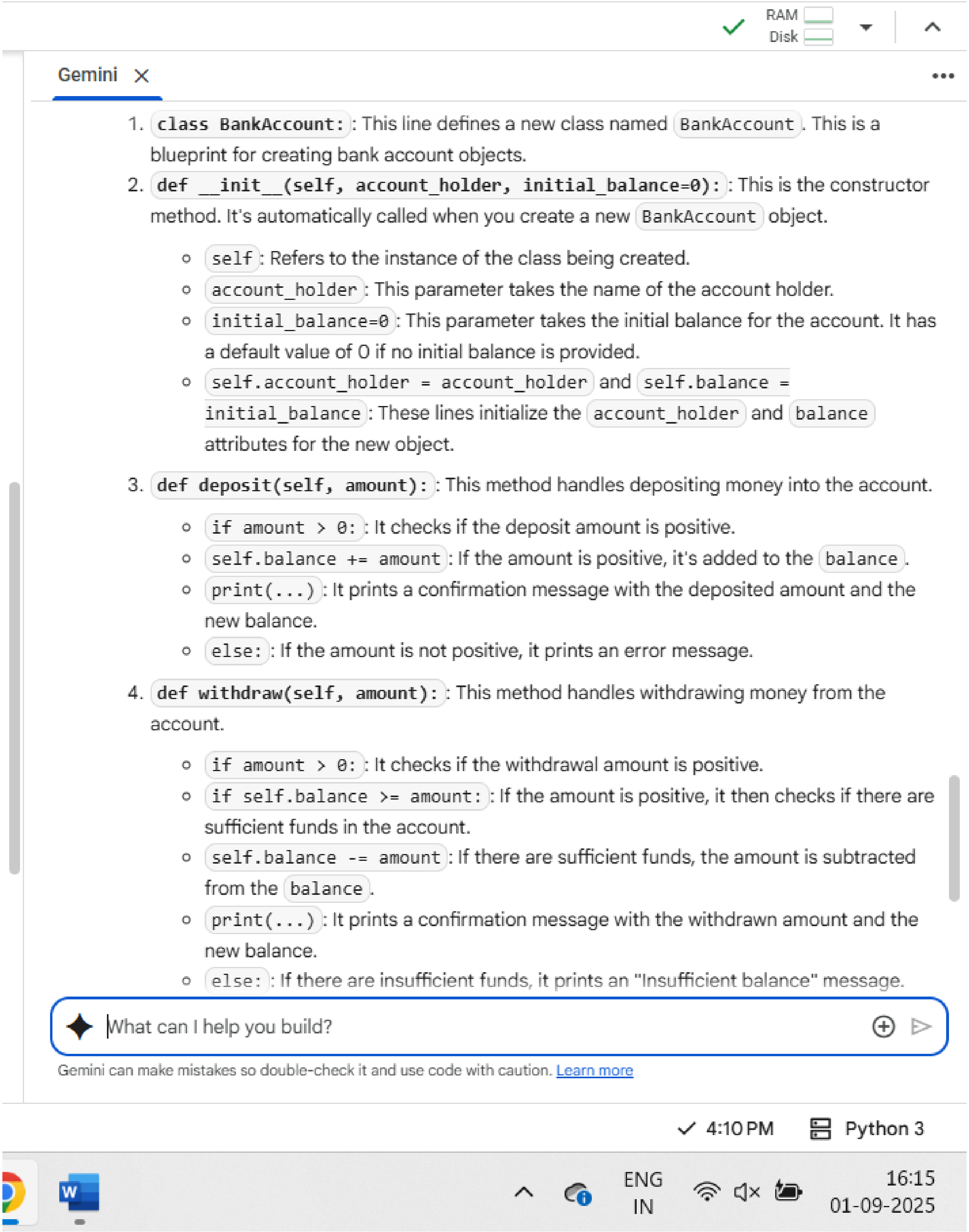
Code screenshots:



```python
class BankAccount:
    def __init__(self, account_holder, initial_balance=0):
        self.account_holder = account_holder
        self.balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"Deposited: ${amount}. New balance: ${self.balance}")
        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}. New balance: ${self.balance}")
            else:
                print("Insufficient balance.")
        else:
            print("Withdrawal amount must be positive.")

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

# Example usage:
account1 = BankAccount("Alice Smith", 1000)
account1.check_balance()
account1.deposit(500)
account1.withdraw(200)
account1.withdraw(1500) # This should result in an insufficient balance message
account1.check_balance()
```

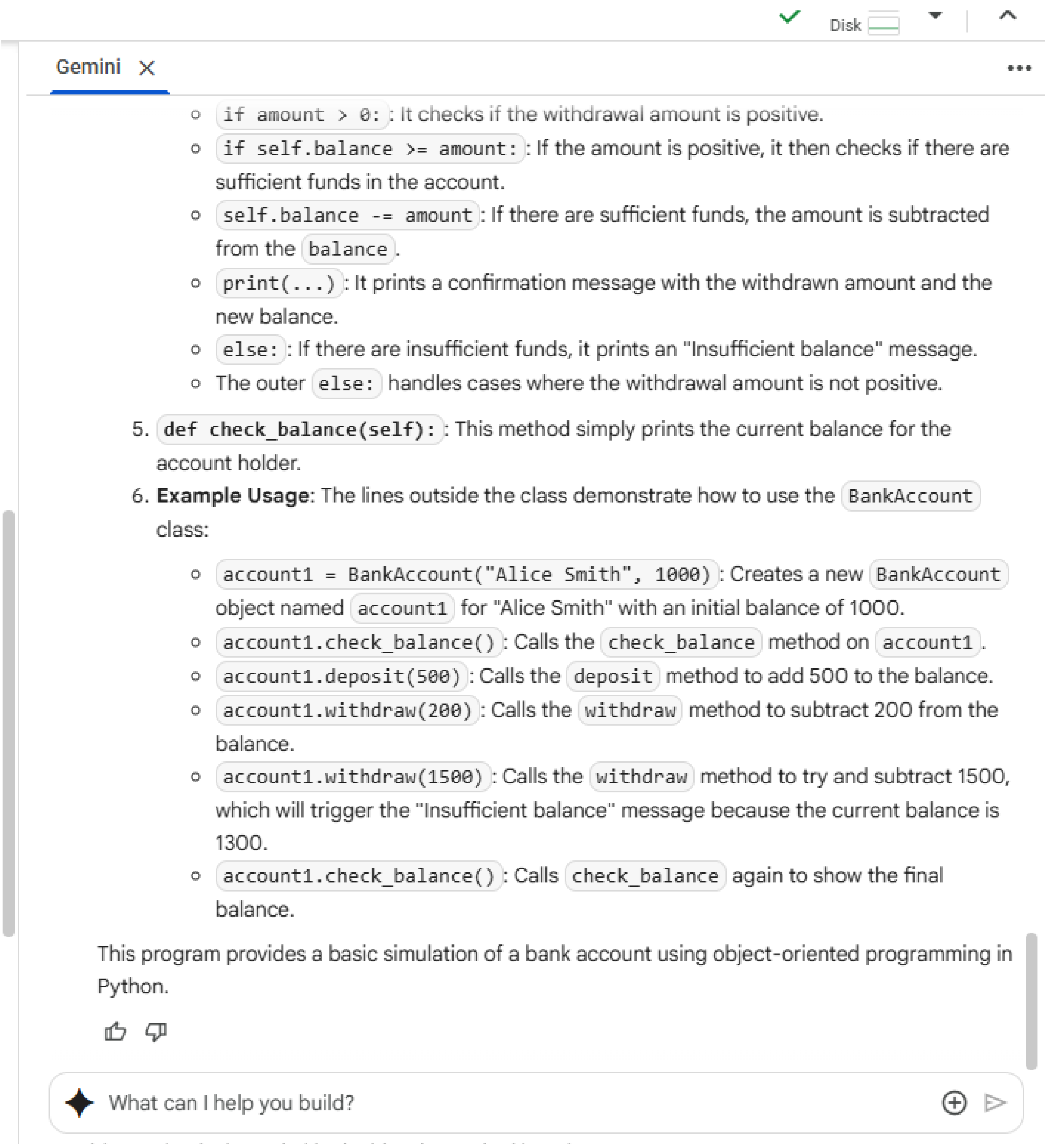
```
Account balance for Alice Smith: $1000
Deposited: $500. New balance: $1500
Withdrew: $200. New balance: $1300
Insufficient balance.
Account balance for Alice Smith: $1300
```

Code explanation:



1. `class BankAccount:`: This line defines a new class named `BankAccount`. This is a blueprint for creating bank account objects.
2. `def __init__(self, account_holder, initial_balance=0):`: This is the constructor method. It's automatically called when you create a new `BankAccount` object.
    - `self`: Refers to the instance of the class being created.
    - `account_holder`: This parameter takes the name of the account holder.
    - `initial_balance=0`: This parameter takes the initial balance for the account. It has a default value of 0 if no initial balance is provided.
    - `self.account_holder = account_holder` and `self.balance = initial_balance`: These lines initialize the `account_holder` and `balance` attributes for the new object.
3. `def deposit(self, amount):`: This method handles depositing money into the account.
    - `if amount > 0:`: It checks if the deposit amount is positive.
    - `self.balance += amount`: If the amount is positive, it's added to the `balance`.
    - `print(...)`: It prints a confirmation message with the deposited amount and the new balance.
    - `else:`: If the amount is not positive, it prints an error message.
4. `def withdraw(self, amount):`: This method handles withdrawing money from the account.
    - `if amount > 0:`: It checks if the withdrawal amount is positive.
    - `if self.balance >= amount:`: If the amount is positive, it then checks if there are sufficient funds in the account.
    - `self.balance -= amount`: If there are sufficient funds, the amount is subtracted from the `balance`.
    - `print(...)`: It prints a confirmation message with the withdrawn amount and the new balance.
    - `else:`: If there are insufficient funds, it prints an "Insufficient balance" message.

Gemini can make mistakes so double-check it and use code with caution. Learn more



- `if amount > 0:`: It checks if the withdrawal amount is positive.
- `if self.balance >= amount:`: If the amount is positive, it then checks if there are sufficient funds in the account.
- `self.balance -= amount`: If there are sufficient funds, the amount is subtracted from the `balance`.
- `print(...)`: It prints a confirmation message with the withdrawn amount and the new balance.
- `else:`: If there are insufficient funds, it prints an "Insufficient balance" message.
- The outer `else:` handles cases where the withdrawal amount is not positive.

5. **`def check_balance(self):`**: This method simply prints the current balance for the account holder.
6. **Example Usage**: The lines outside the class demonstrate how to use the `BankAccount` class:
    - `account1 = BankAccount("Alice Smith", 1000)`: Creates a new `BankAccount` object named `account1` for "Alice Smith" with an initial balance of 1000.
    - `account1.check_balance()`: Calls the `check_balance` method on `account1`.
    - `account1.deposit(500)`: Calls the `deposit` method to add 500 to the balance.
    - `account1.withdraw(200)`: Calls the `withdraw` method to subtract 200 from the balance.
    - `account1.withdraw(1500)`: Calls the `withdraw` method to try and subtract 1500, which will trigger the "Insufficient balance" message because the current balance is 1300.
    - `account1.check_balance()`: Calls `check_balance` again to show the final balance.

This program provides a basic simulation of a bank account using object-oriented programming in Python.

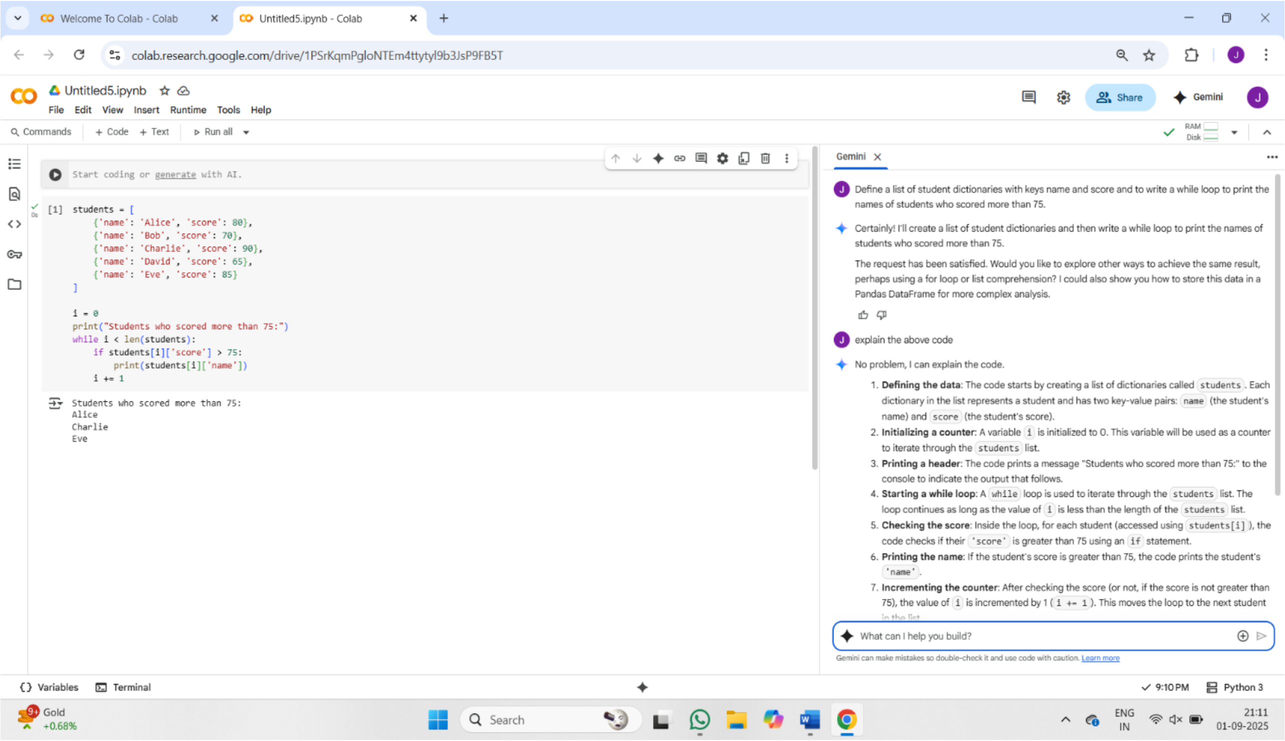
Task-4:

• 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

#PROMPT USED:

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

CODE SCREENSHOTS:


Welcome To Colab - Colab
Untitled5.ipynb - Colab
colab.research.google.com/drive/1PSrKqmPgloNTEm4ttytyl9b3JsP9FB5T
Untitled5.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
Gemini
RAM
Disk
Start coding or generate with AI.
Variables Terminal
9:10 PM
Python 3
Gold
+0.68%
Search
ENG
IN
21:11
01-09-2025

```
[1] students = [
        {'name': 'Alice', 'score': 80},
        {'name': 'Bob', 'score': 70},
        {'name': 'Charlie', 'score': 90},
        {'name': 'David', 'score': 65},
        {'name': 'Eve', 'score': 85}
    ]

    i = 0
    print("Students who scored more than 75:")
    while i < len(students):
        if students[i]['score'] > 75:
            print(students[i]['name'])
        i += 1
```

```
Students who scored more than 75:
Alice
Charlie
Eve
```

Gemini

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

Certainly! I'll create a list of student dictionaries and then write a while loop to print the names of students who scored more than 75.

The request has been satisfied. Would you like to explore other ways to achieve the same result, perhaps using a for loop or list comprehension? I could also show you how to store this data in a Pandas DataFrame for more complex analysis.

explain the above code

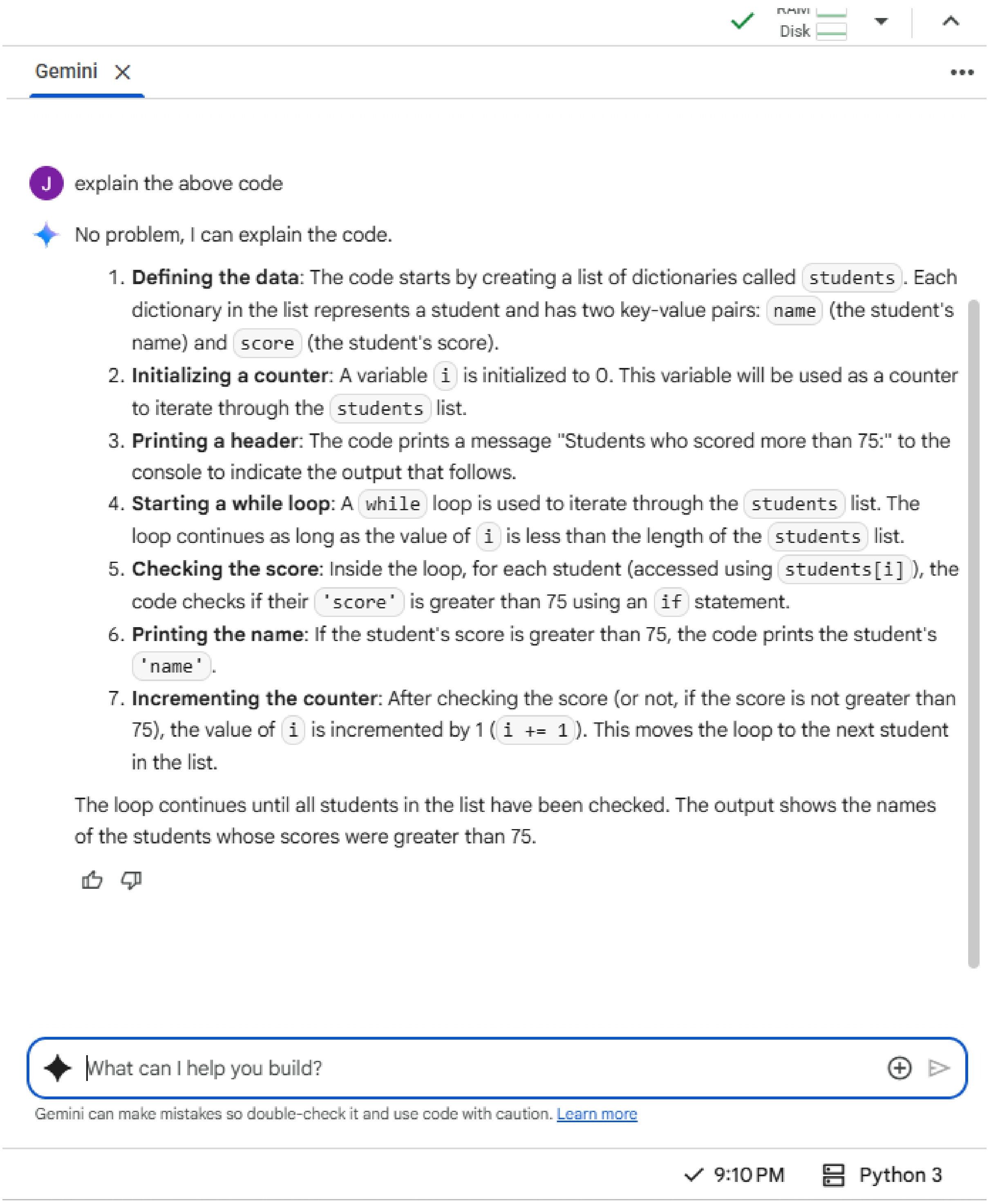
No problem, I can explain the code.

1. **Defining the data**: The code starts by creating a list of dictionaries called `students`. Each dictionary in the list represents a student and has two key-value pairs: `name` (the student's name) and `score` (the student's score).
2. **Initializing a counter**: A variable `i` is initialized to 0. This variable will be used as a counter to iterate through the `students` list.
3. **Printing a header**: The code prints a message "Students who scored more than 75:" to the console to indicate the output that follows.
4. **Starting a while loop**: A `while` loop is used to iterate through the `students` list. The loop continues as long as the value of `i` is less than the length of the `students` list.
5. **Checking the score**: Inside the loop, for each student (accessed using `students[i]`), the code checks if their `'score'` is greater than 75 using an `if` statement.
6. **Printing the name**: If the student's score is greater than 75, the code prints the student's `'name'`.
7. **Incrementing the counter**: After checking the score (or not, if the score is not greater than 75), the value of `i` is incremented by 1 (`i += 1`). This moves the loop to the next student in the list

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

CODE EXPLANATION:



explain the above code

No problem, I can explain the code.

1. **Defining the data**: The code starts by creating a list of dictionaries called `students`. Each dictionary in the list represents a student and has two key-value pairs: `name` (the student's name) and `score` (the student's score).
2. **Initializing a counter**: A variable `i` is initialized to 0. This variable will be used as a counter to iterate through the `students` list.
3. **Printing a header**: The code prints a message "Students who scored more than 75:" to the console to indicate the output that follows.
4. **Starting a while loop**: A `while` loop is used to iterate through the `students` list. The loop continues as long as the value of `i` is less than the length of the `students` list.
5. **Checking the score**: Inside the loop, for each student (accessed using `students[i]`), the code checks if their `'score'` is greater than 75 using an `if` statement.
6. **Printing the name**: If the student's score is greater than 75, the code prints the student's `'name'`.
7. **Incrementing the counter**: After checking the score (or not, if the score is not greater than 75), the value of `i` is incremented by 1 (`i += 1`). This moves the loop to the next student in the list.

The loop continues until all students in the list have been checked. The output shows the names of the students whose scores were greater than 75.

What can I help you build?
Gemini can make mistakes so double-check it and use code with caution. Learn more
9:10 PM
Python 3

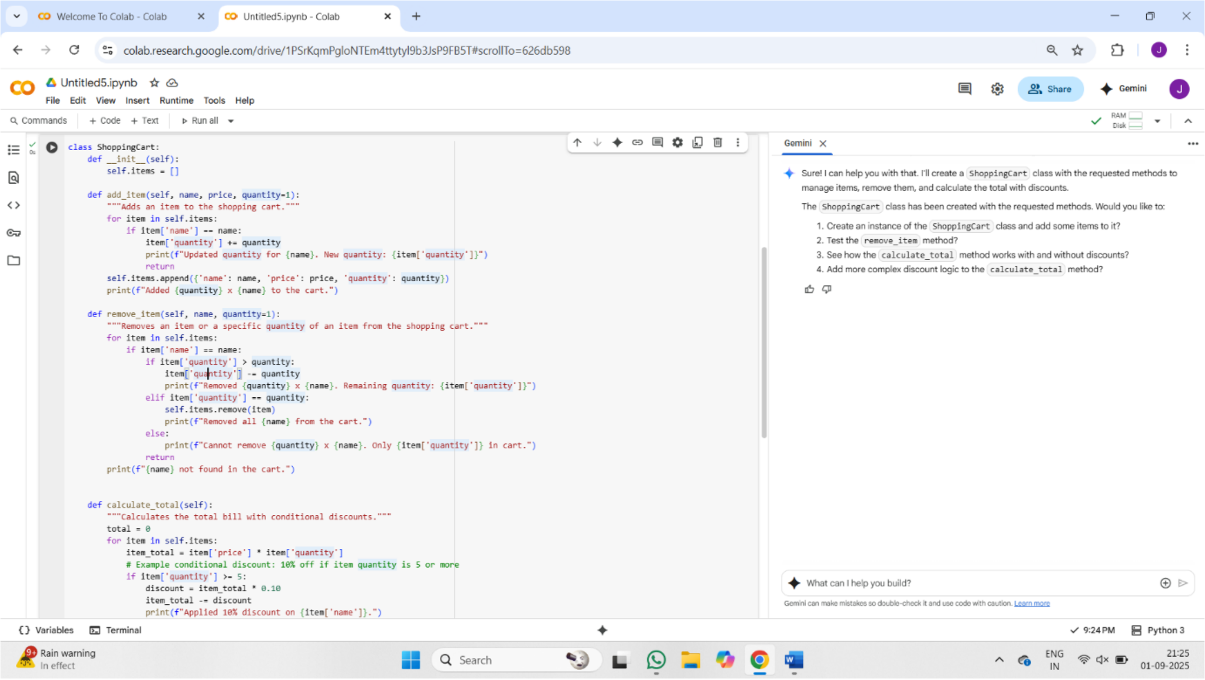
TASK-5:

- **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.**

#PROMPT USED:

Begin writing a class ShoppingCart with an empty items list to generate methods to add_item, remove_item, and use a loop to calculate the total bill using conditional discounts.

CODE SCREENSHOTS:



```python
class ShoppingCart:
    def __init__(self):
        self.items = []

    def add_item(self, name, price, quantity=1):
        """Adds an item to the shopping cart."""
        for item in self.items:
            if item['name'] == name:
                item['quantity'] += quantity
                print(f"Updated quantity for {name}. New quantity: {item['quantity']}")
                return
        self.items.append({'name': name, 'price': price, 'quantity': quantity})
        print(f"Added {quantity} x {name} to the cart.")

    def remove_item(self, name, quantity=1):
        """Removes an item or a specific quantity of an item from the shopping cart."""
        for item in self.items:
            if item['name'] == name:
                if item['quantity'] > quantity:
                    item['quantity'] -= quantity
                    print(f"Removed {quantity} x {name}. Remaining quantity: {item['quantity']}")
                elif item['quantity'] == quantity:
                    self.items.remove(item)
                    print(f"Removed all {name} from the cart.")
                else:
                    print(f"Cannot remove {quantity} x {name}. Only {item['quantity']} in cart.")
                return
        print(f"{name} not found in the cart.")


    def calculate_total(self):
        """Calculates the total bill with conditional discounts."""
        total = 0
        for item in self.items:
            item_total = item['price'] * item['quantity']
            # Example conditional discount: 10% off if item quantity is 5 or more
            if item['quantity'] >= 5:
                discount = item_total * 0.10
                item_total -= discount
                print(f"Applied 10% discount on {item['name']}.")
```

Gemini

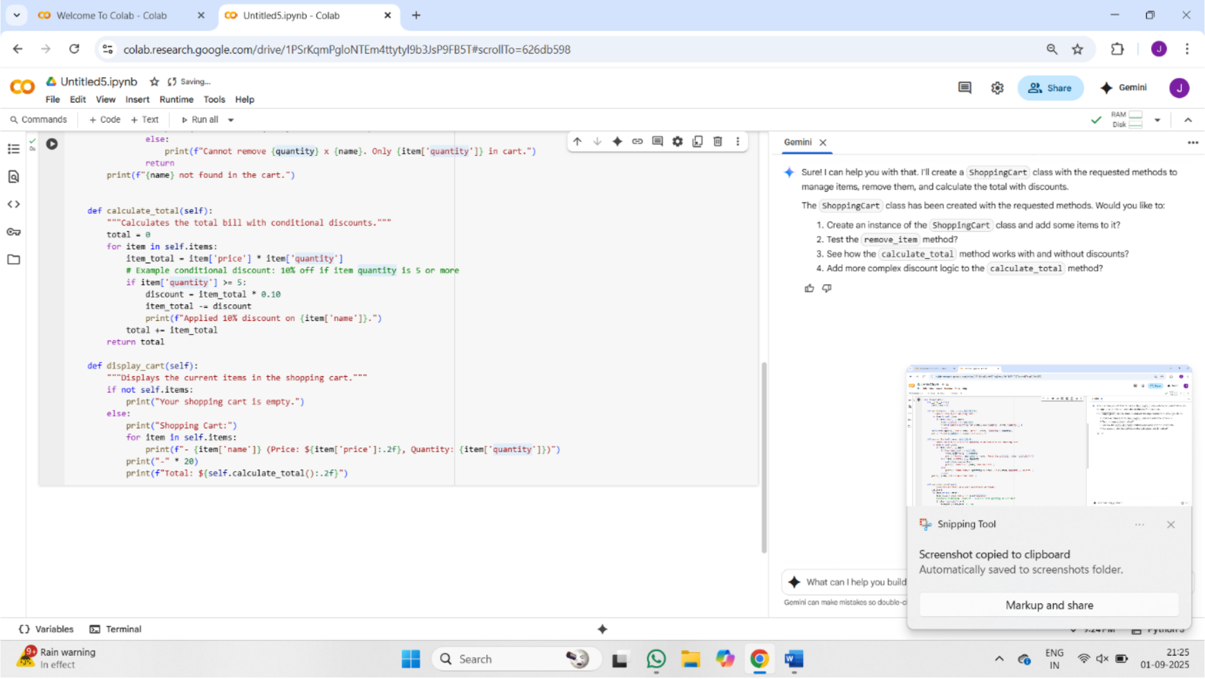
Sure! I can help you with that. I'll create a `ShoppingCart` class with the requested methods to manage items, remove them, and calculate the total with discounts.

The `ShoppingCart` class has been created with the requested methods. Would you like to:

1. Create an instance of the `ShoppingCart` class and add some items to it?
2. Test the `remove_item` method?
3. See how the `calculate_total` method works with and without discounts?
4. Add more complex discount logic to the `calculate_total` method?

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more



```python
                else:
                    print(f"Cannot remove {quantity} x {name}. Only {item['quantity']} in cart.")
                return
        print(f"{name} not found in the cart.")


    def calculate_total(self):
        """Calculates the total bill with conditional discounts."""
        total = 0
        for item in self.items:
            item_total = item['price'] * item['quantity']
            # Example conditional discount: 10% off if item quantity is 5 or more
            if item['quantity'] >= 5:
                discount = item_total * 0.10
                item_total -= discount
                print(f"Applied 10% discount on {item['name']}.")
            total += item_total
        return total

    def display_cart(self):
        """Displays the current items in the shopping cart."""
        if not self.items:
            print("Your shopping cart is empty.")
        else:
            print("Shopping Cart:")
            for item in self.items:
                print(f"- {item['name']} (Price: ${item['price']:.2f}, Quantity: {item['quantity']})")
            print("-" * 20)
            print(f"Total: ${self.calculate_total():.2f}")
```

Gemini

Sure! I can help you with that. I'll create a `ShoppingCart` class with the requested methods to manage items, remove them, and calculate the total with discounts.

The `ShoppingCart` class has been created with the requested methods. Would you like to:

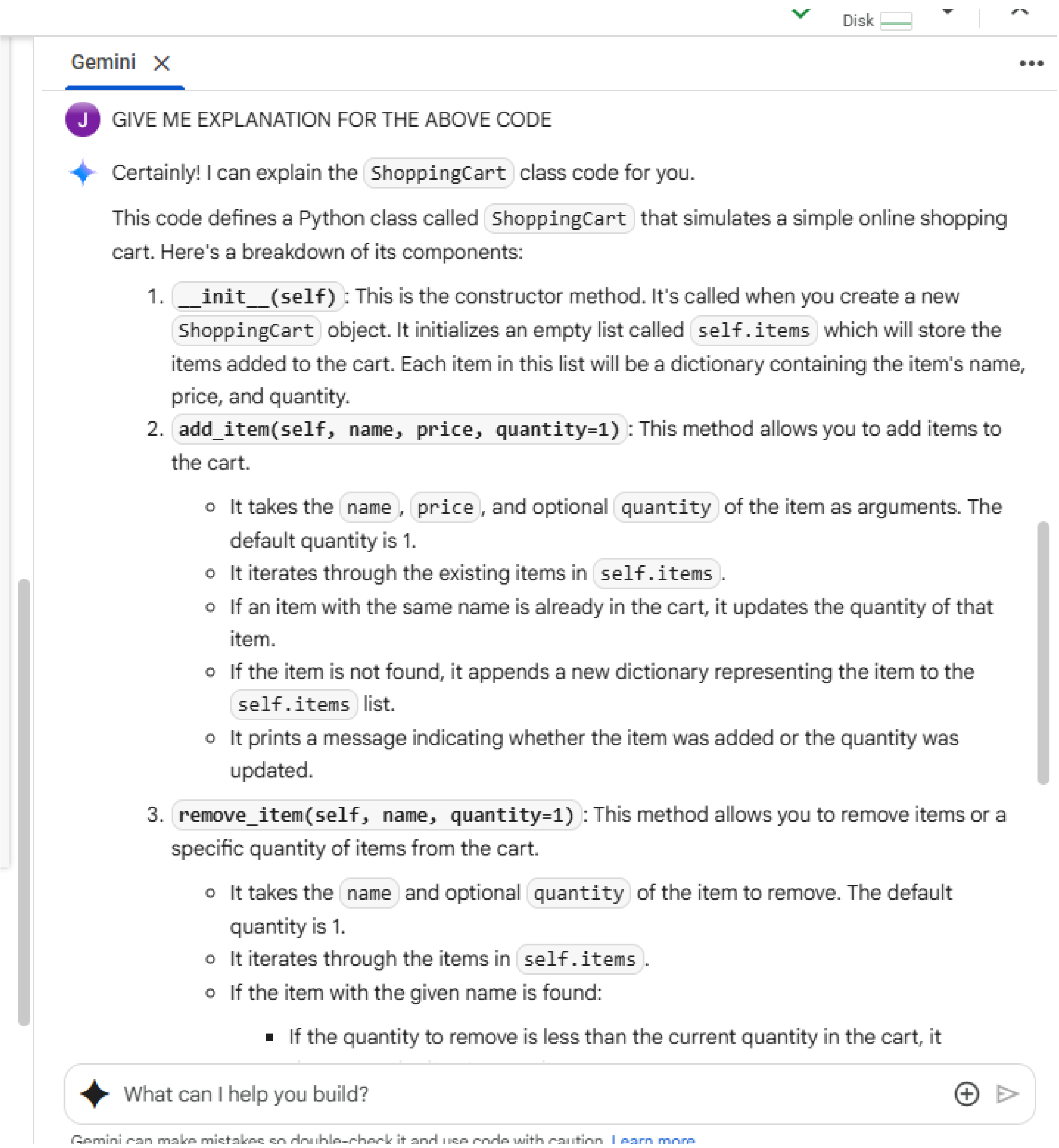
1. Create an instance of the `ShoppingCart` class and add some items to it?
2. Test the `remove_item` method?
3. See how the `calculate_total` method works with and without discounts?
4. Add more complex discount logic to the `calculate_total` method?

Snipping Tool

Screenshot copied to clipboard
Automatically saved to screenshots folder.

Markup and share

CODE EXPLANATION:



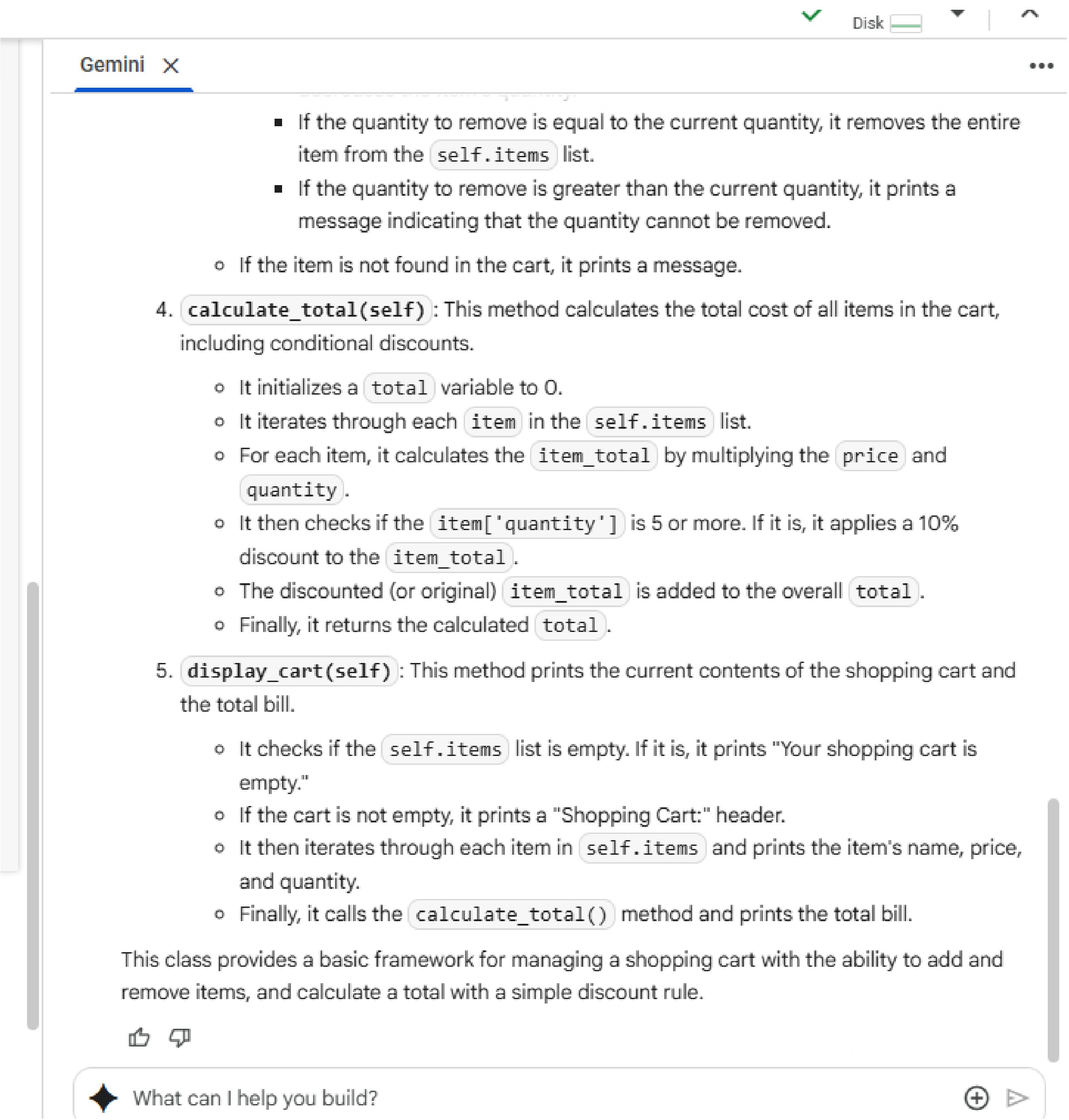
J GIVE ME EXPLANATION FOR THE ABOVE CODE

Certainly! I can explain the `ShoppingCart` class code for you.

This code defines a Python class called `ShoppingCart` that simulates a simple online shopping cart. Here's a breakdown of its components:

1. `__init__(self)`: This is the constructor method. It's called when you create a new `ShoppingCart` object. It initializes an empty list called `self.items` which will store the items added to the cart. Each item in this list will be a dictionary containing the item's name, price, and quantity.
2. `add_item(self, name, price, quantity=1)`: This method allows you to add items to the cart.
   - It takes the `name`, `price`, and optional `quantity` of the item as arguments. The default quantity is 1.
   - It iterates through the existing items in `self.items`.
   - If an item with the same name is already in the cart, it updates the quantity of that item.
   - If the item is not found, it appends a new dictionary representing the item to the `self.items` list.
   - It prints a message indicating whether the item was added or the quantity was updated.
3. `remove_item(self, name, quantity=1)`: This method allows you to remove items or a specific quantity of items from the cart.
   - It takes the `name` and optional `quantity` of the item to remove. The default quantity is 1.
   - It iterates through the items in `self.items`.
   - If the item with the given name is found:
     - If the quantity to remove is less than the current quantity in the cart, it

Gemini can make mistakes so double-check it and use code with caution. Learn more



        - If the quantity to remove is equal to the current quantity, it removes the entire item from the `self.items` list.
        - If the quantity to remove is greater than the current quantity, it prints a message indicating that the quantity cannot be removed.
    - If the item is not found in the cart, it prints a message.

4. **`calculate_total(self)`**: This method calculates the total cost of all items in the cart, including conditional discounts.
    - It initializes a `total` variable to 0.
    - It iterates through each `item` in the `self.items` list.
    - For each item, it calculates the `item_total` by multiplying the `price` and `quantity`.
    - It then checks if the `item['quantity']` is 5 or more. If it is, it applies a 10% discount to the `item_total`.
    - The discounted (or original) `item_total` is added to the overall `total`.
    - Finally, it returns the calculated `total`.
5. **`display_cart(self)`**: This method prints the current contents of the shopping cart and the total bill.
    - It checks if the `self.items` list is empty. If it is, it prints "Your shopping cart is empty."
    - If the cart is not empty, it prints a "Shopping Cart:" header.
    - It then iterates through each item in `self.items` and prints the item's name, price, and quantity.
    - Finally, it calls the `calculate_total()` method and prints the total bill.

This class provides a basic framework for managing a shopping cart with the ability to add and remove items, and calculate a total with a simple discount rule.