LAB ASSIGNMENT 6

TASK1

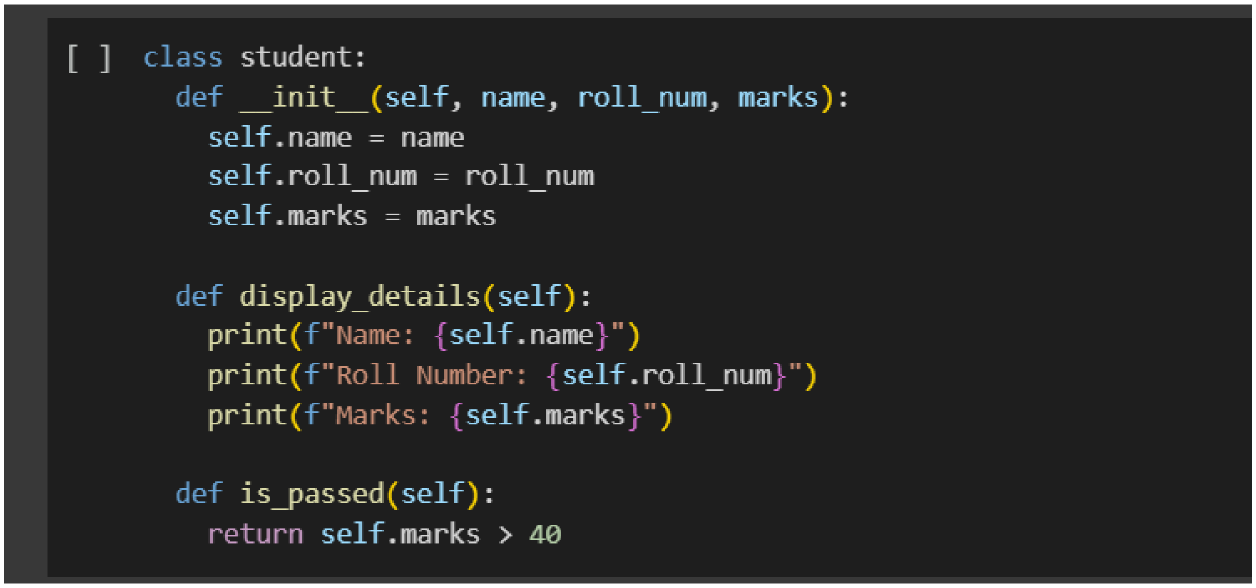
PROMPT:

#Create a python class called student with attributes name,roll no, and marks

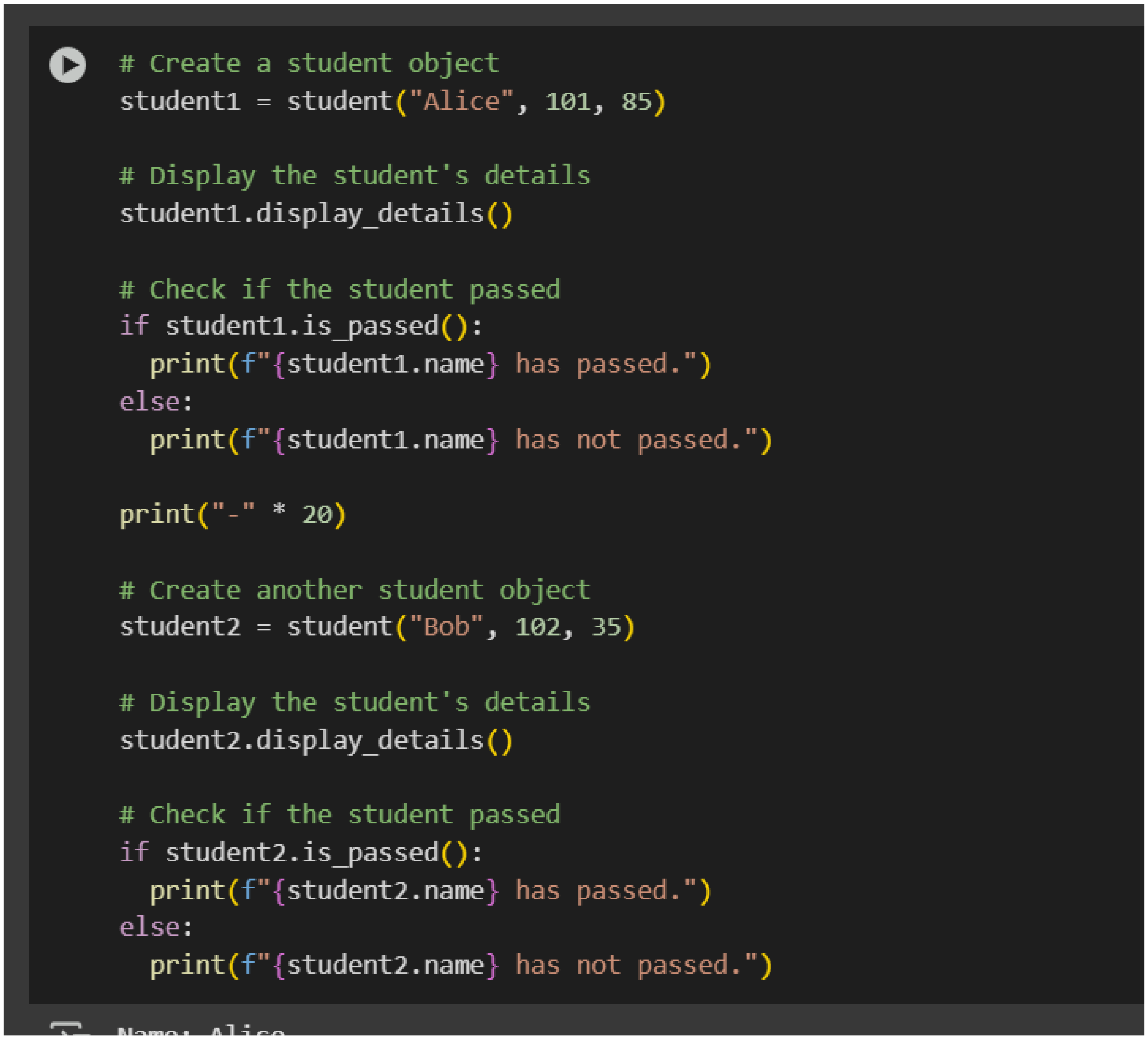
#include methods like display() and passed()

#Create an object to check the class

CODE:



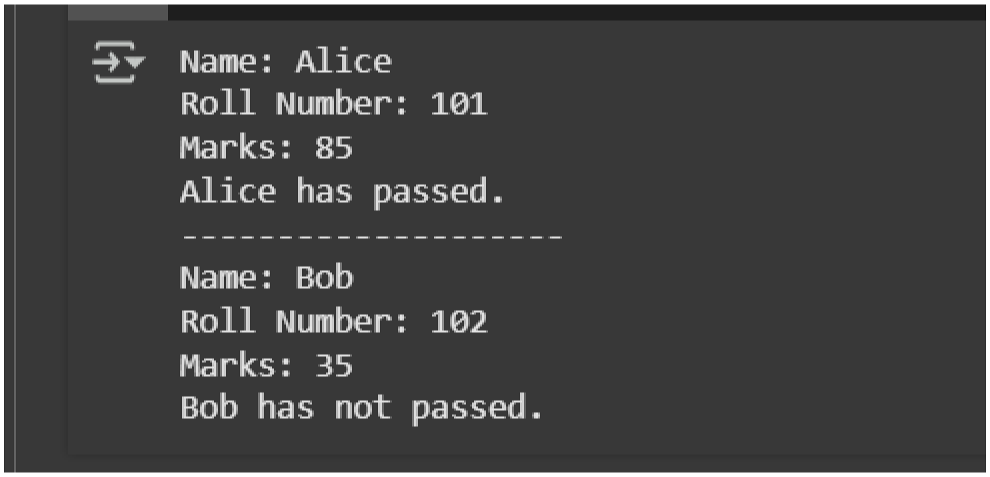
```
[ ]  class student:
       def __init__(self, name, roll_num, marks):
         self.name = name
         self.roll_num = roll_num
         self.marks = marks

       def display_details(self):
         print(f"Name: {self.name}")
         print(f"Roll Number: {self.roll_num}")
         print(f"Marks: {self.marks}")

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



```python
# Create a student object
student1 = student("Alice", 101, 85)

# Display the student's details
student1.display_details()

# Check if the student passed
if student1.is_passed():
  print(f"{student1.name} has passed.")
else:
  print(f"{student1.name} has not passed.")

print("-" * 20)

# Create another student object
student2 = student("Bob", 102, 35)

# Display the student's details
student2.display_details()

# Check if the student passed
if student2.is_passed():
  print(f"{student2.name} has passed.")
else:
  print(f"{student2.name} has not passed.")
```

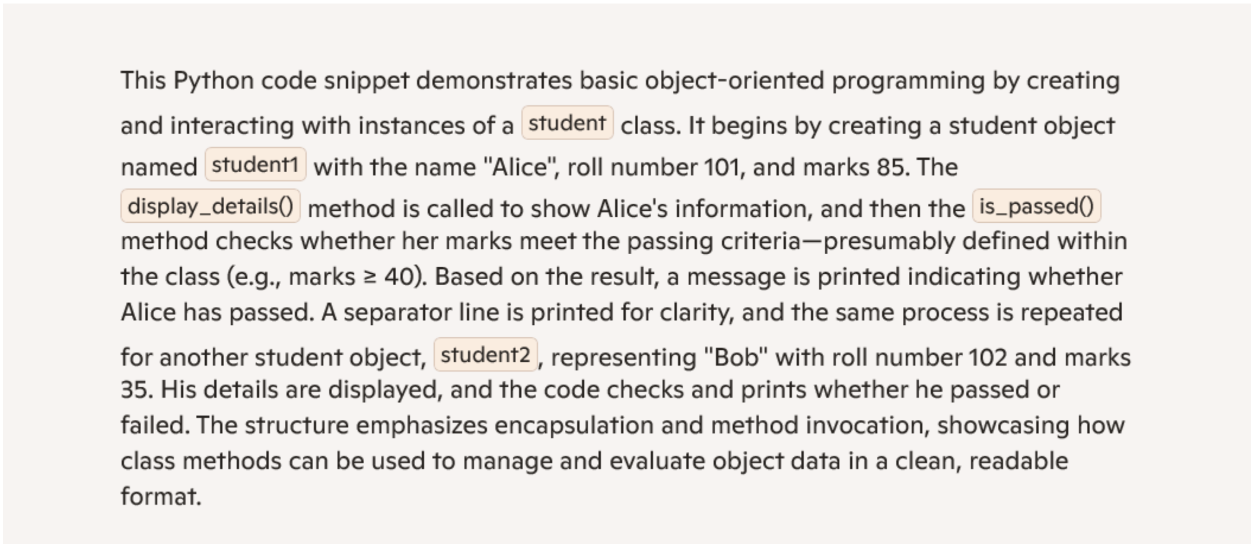
```
Name: Alice
```

OUTPUT:



```
Name: Alice
Roll Number: 101
Marks: 85
Alice has passed.
--------------------
Name: Bob
Roll Number: 102
Marks: 35
Bob has not passed.
```

EXPLANTION:



This Python code snippet demonstrates basic object-oriented programming by creating and interacting with instances of a `student` class. It begins by creating a student object named `student1` with the name "Alice", roll number 101, and marks 85. The `display_details()` method is called to show Alice's information, and then the `is_passed()` method checks whether her marks meet the passing criteria—presumably defined within the class (e.g., marks ≥ 40). Based on the result, a message is printed indicating whether Alice has passed. A separator line is printed for clarity, and the same process is repeated for another student object, `student2`, representing "Bob" with roll number 102 and marks 35. His details are displayed, and the code checks and prints whether he passed or failed. The structure emphasizes encapsulation and method invocation, showcasing how class methods can be used to manage and evaluate object data in a clean, readable format.

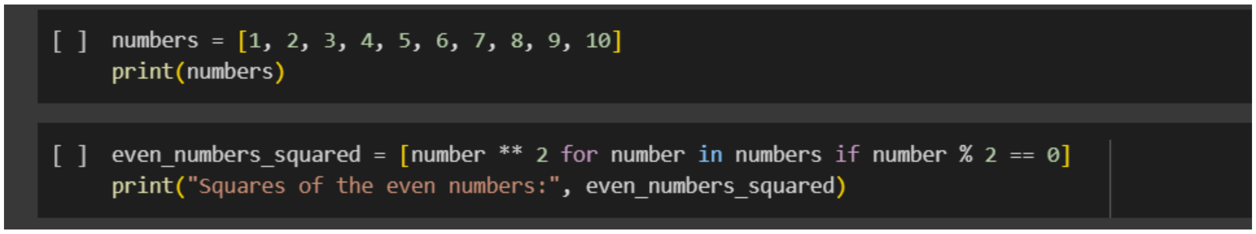
TASK 2

PROMPT:

#Write a python program to create a loop of random numbers

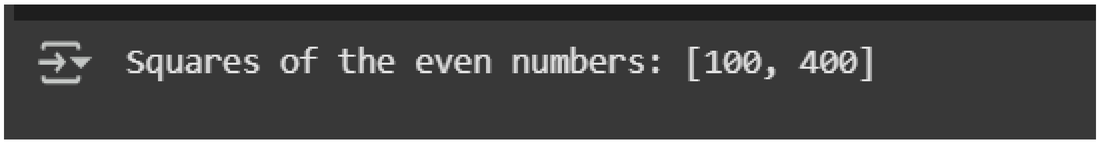
#ask the ai to generate a code that calculates and prints the squares of the even numbers in the list

CODE:



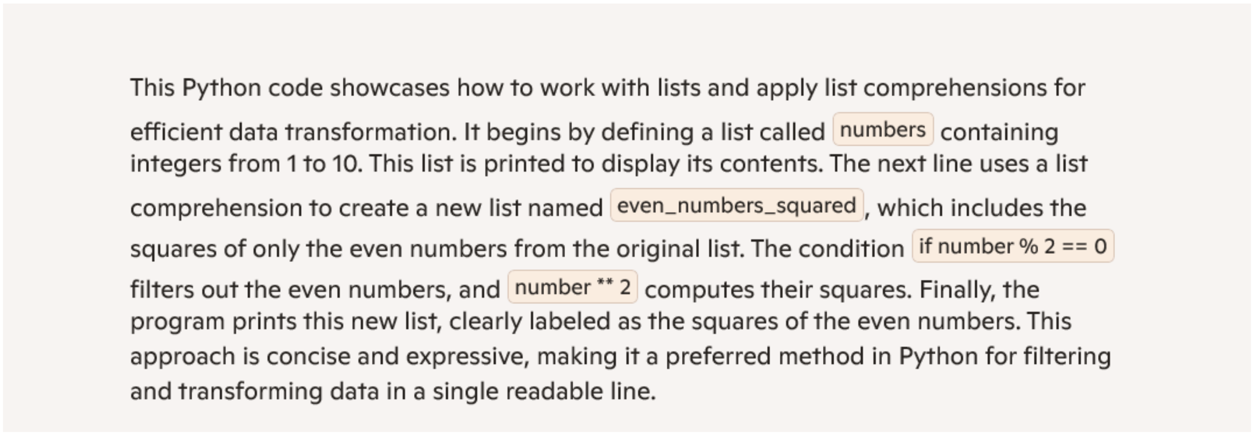
```
[ ] numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    print(numbers)

[ ] even_numbers_squared = [number ** 2 for number in numbers if number % 2 == 0]
    print("Squares of the even numbers:", even_numbers_squared)
```

OUTPUT:



```
Squares of the even numbers: [100, 400]
```

EXPLANTION:



This Python code showcases how to work with lists and apply list comprehensions for efficient data transformation. It begins by defining a list called `numbers` containing integers from 1 to 10. This list is printed to display its contents. The next line uses a list comprehension to create a new list named `even_numbers_squared`, which includes the squares of only the even numbers from the original list. The condition `if number % 2 == 0` filters out the even numbers, and `number ** 2` computes their squares. Finally, the program prints this new list, clearly labeled as the squares of the even numbers. This approach is concise and expressive, making it a preferred method in Python for filtering and transforming data in a single readable line.

TASK 3

PROMPT:

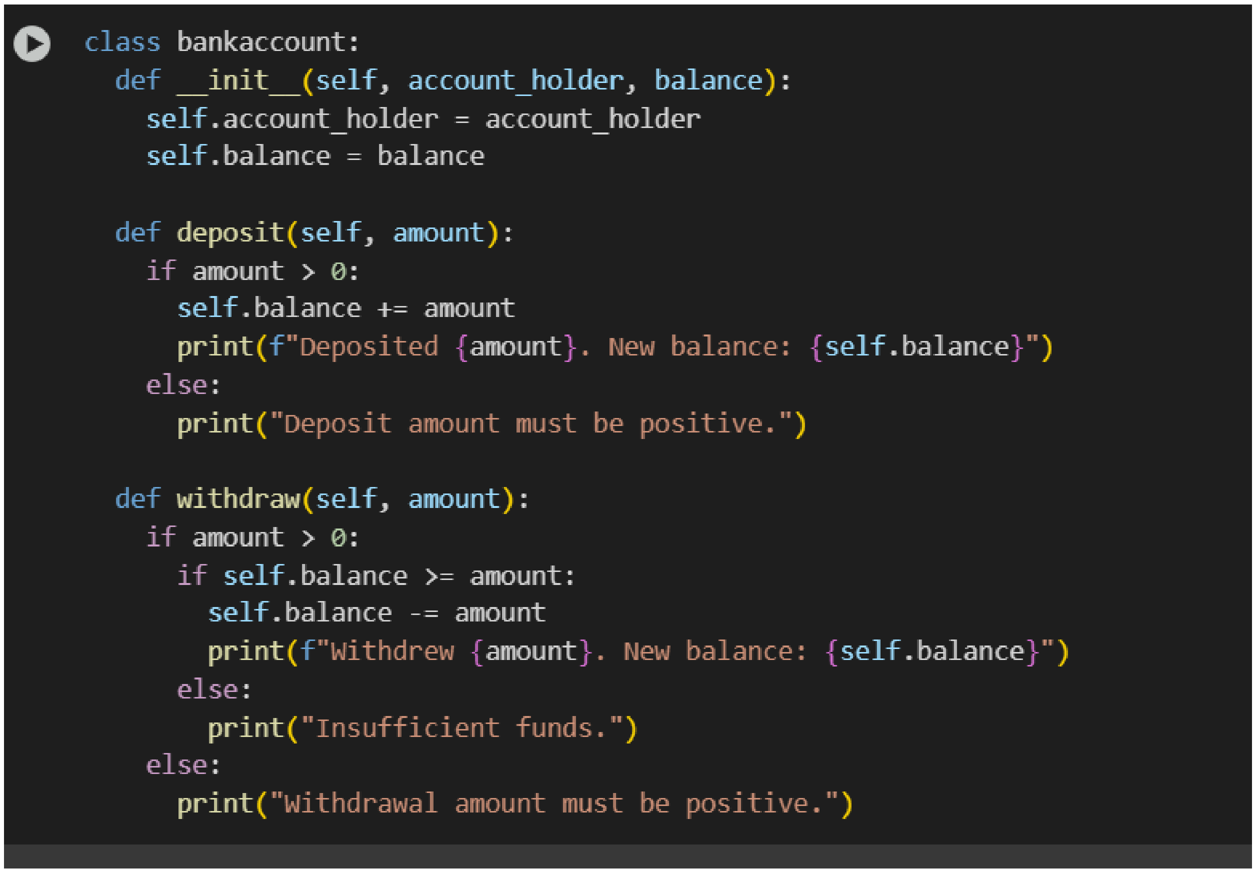
#create a class called bank account with attributes account

holder and balance

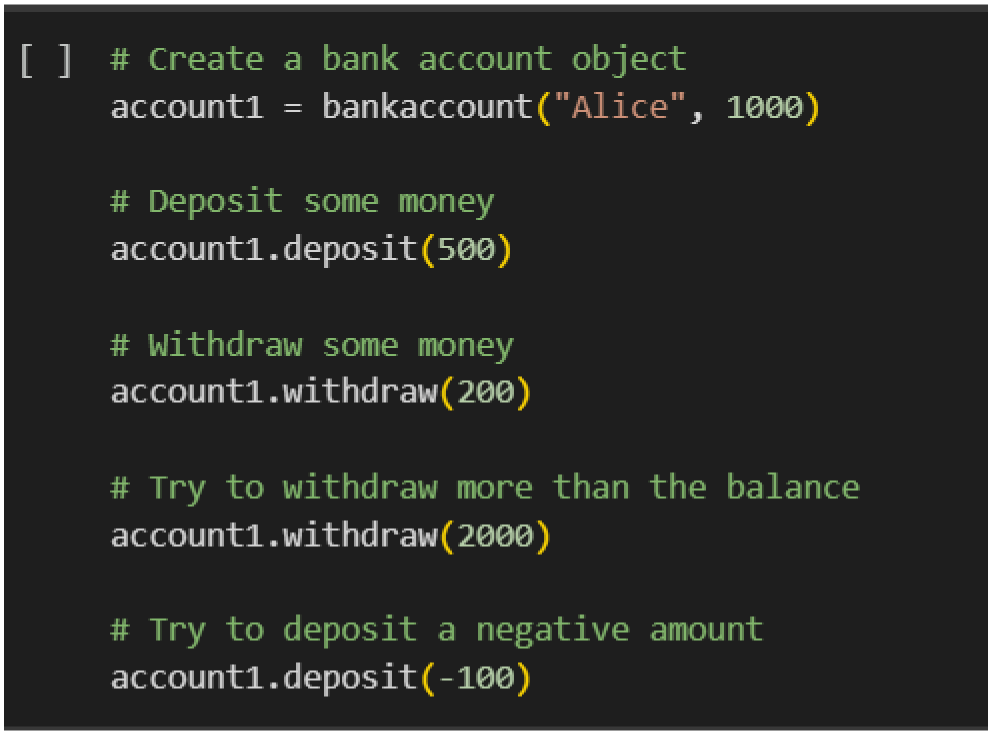
#method like deposits and withdraw

# create an object to check the class

CODE:

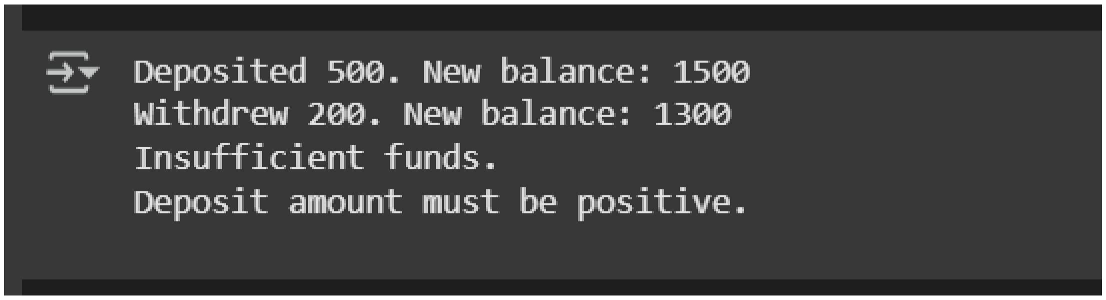


```python
class bankaccount:
  def __init__(self, account_holder, balance):
    self.account_holder = account_holder
    self.balance = balance

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



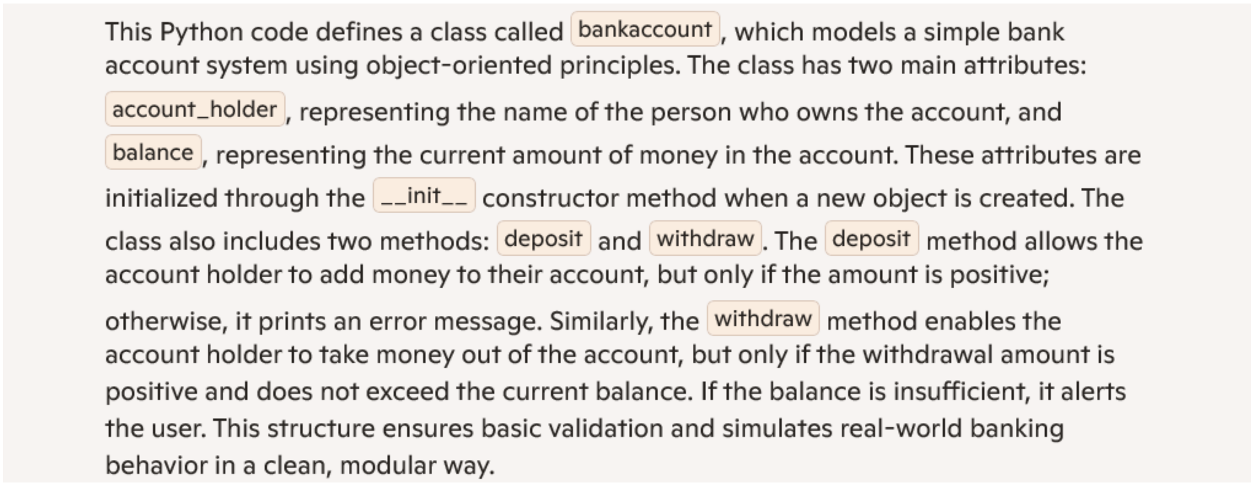
```python
# Create a bank account object
account1 = bankaccount("Alice", 1000)

# Deposit some money
account1.deposit(500)

# Withdraw some money
account1.withdraw(200)

# Try to withdraw more than the balance
account1.withdraw(2000)

# Try to deposit a negative amount
account1.deposit(-100)
```

OUTPUT:



```
Deposited 500. New balance: 1500
Withdrew 200. New balance: 1300
Insufficient funds.
Deposit amount must be positive.
```

EXPLANTION:



This Python code defines a class called `bankaccount`, which models a simple bank account system using object-oriented principles. The class has two main attributes: `account_holder`, representing the name of the person who owns the account, and `balance`, representing the current amount of money in the account. These attributes are initialized through the `__init__` constructor method when a new object is created. The class also includes two methods: `deposit` and `withdraw`. The `deposit` method allows the account holder to add money to their account, but only if the amount is positive; otherwise, it prints an error message. Similarly, the `withdraw` method enables the account holder to take money out of the account, but only if the withdrawal amount is positive and does not exceed the current balance. If the balance is insufficient, it alerts the user. This structure ensures basic validation and simulates real-world banking behavior in a clean, modular way.

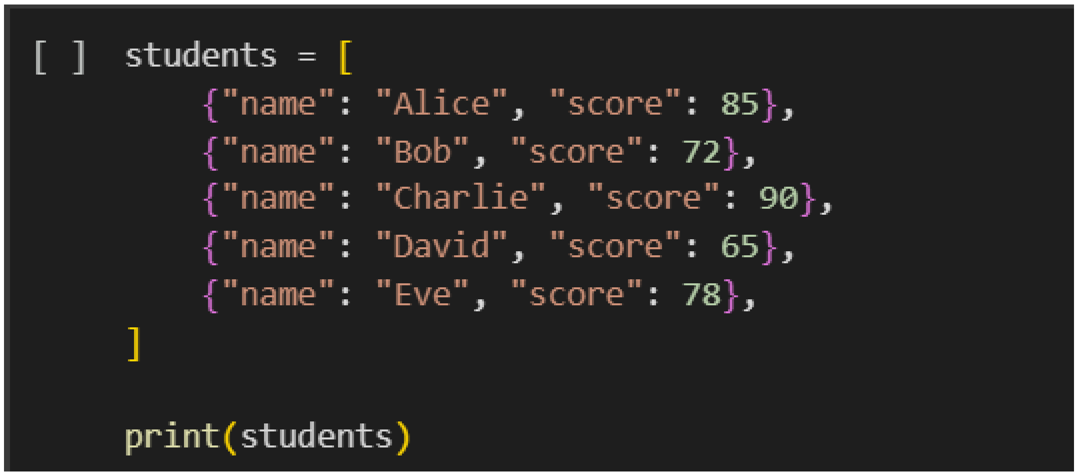
TASK 4

PROMPT:

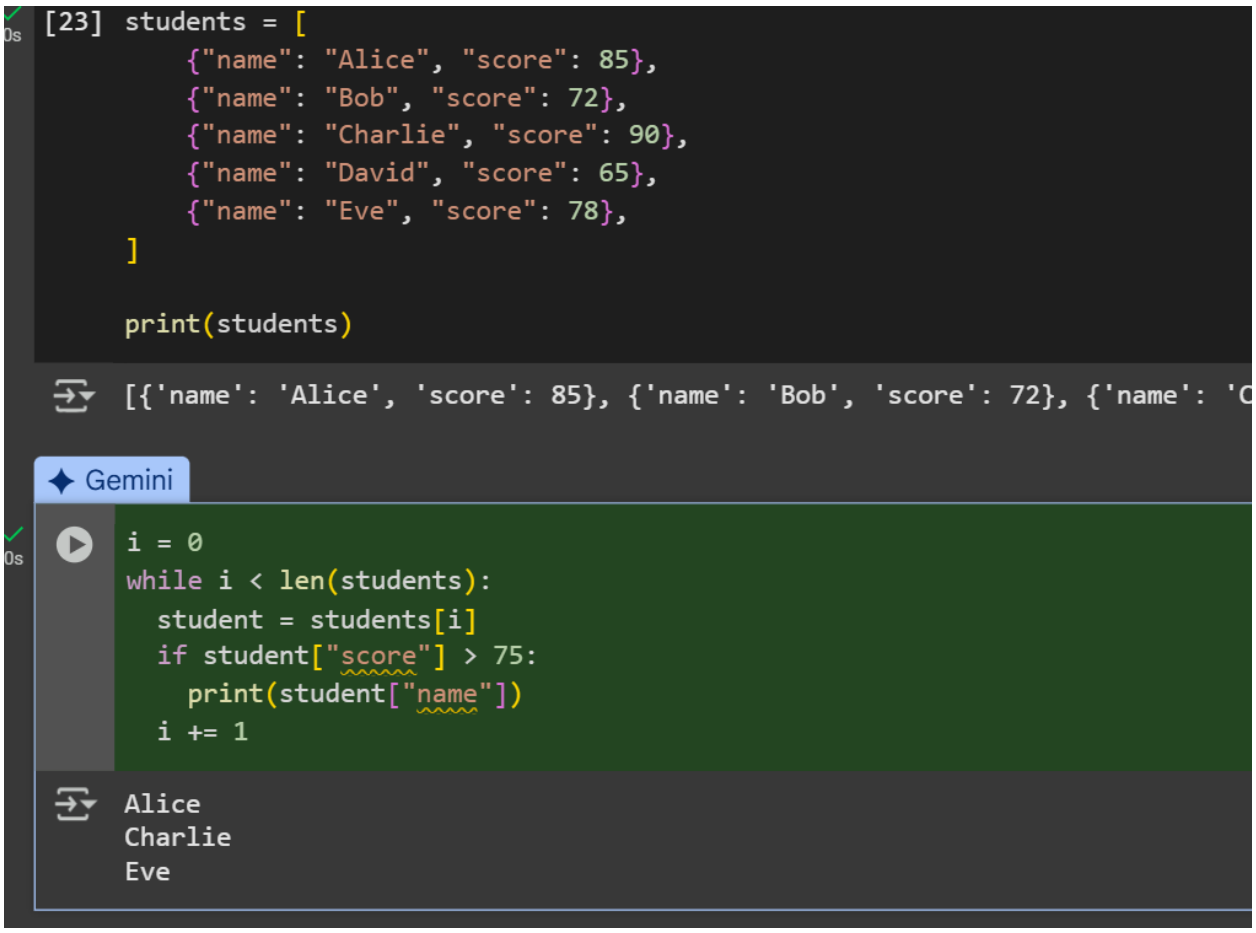
#defining a list of student dictionaries with name and score set up a while

loop skeleton and to print the names who are having more than 75

CODE:



```
[ ]  students = [
         {"name": "Alice", "score": 85},
         {"name": "Bob", "score": 72},
         {"name": "Charlie", "score": 90},
         {"name": "David", "score": 65},
         {"name": "Eve", "score": 78},
     ]

     print(students)
```



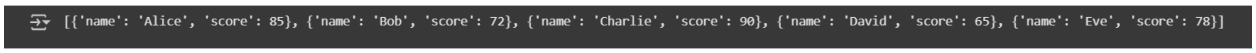
```
[23] students = [
         {"name": "Alice", "score": 85},
         {"name": "Bob", "score": 72},
         {"name": "Charlie", "score": 90},
         {"name": "David", "score": 65},
         {"name": "Eve", "score": 78},
     ]

     print(students)
```

```
[{'name': 'Alice', 'score': 85}, {'name': 'Bob', 'score': 72}, {'name': 'C
```

```
i = 0
while i < len(students):
  student = students[i]
  if student["score"] > 75:
    print(student["name"])
  i += 1
```

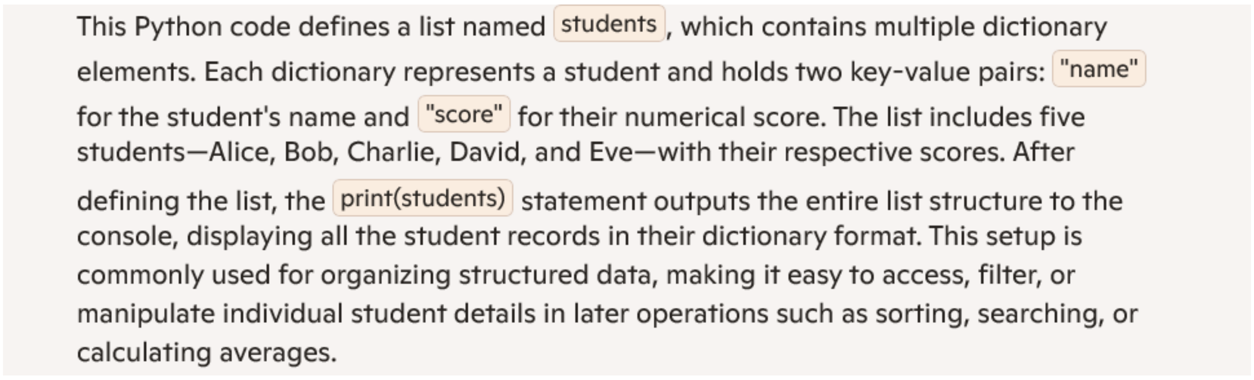
```
Alice
Charlie
Eve
```

OUTPUT:



```
[{'name': 'Alice', 'score': 85}, {'name': 'Bob', 'score': 72}, {'name': 'Charlie', 'score': 90}, {'name': 'David', 'score': 65}, {'name': 'Eve', 'score': 78}]
```

EXPLANTION:



This Python code defines a list named `students`, which contains multiple dictionary elements. Each dictionary represents a student and holds two key-value pairs: `"name"` for the student's name and `"score"` for their numerical score. The list includes five students—Alice, Bob, Charlie, David, and Eve—with their respective scores. After defining the list, the `print(students)` statement outputs the entire list structure to the console, displaying all the student records in their dictionary format. This setup is commonly used for organizing structured data, making it easy to access, filter, or manipulate individual student details in later operations such as sorting, searching, or calculating averages.

TASK 5

PROMPT:

#create a class called shopping cart and attributes

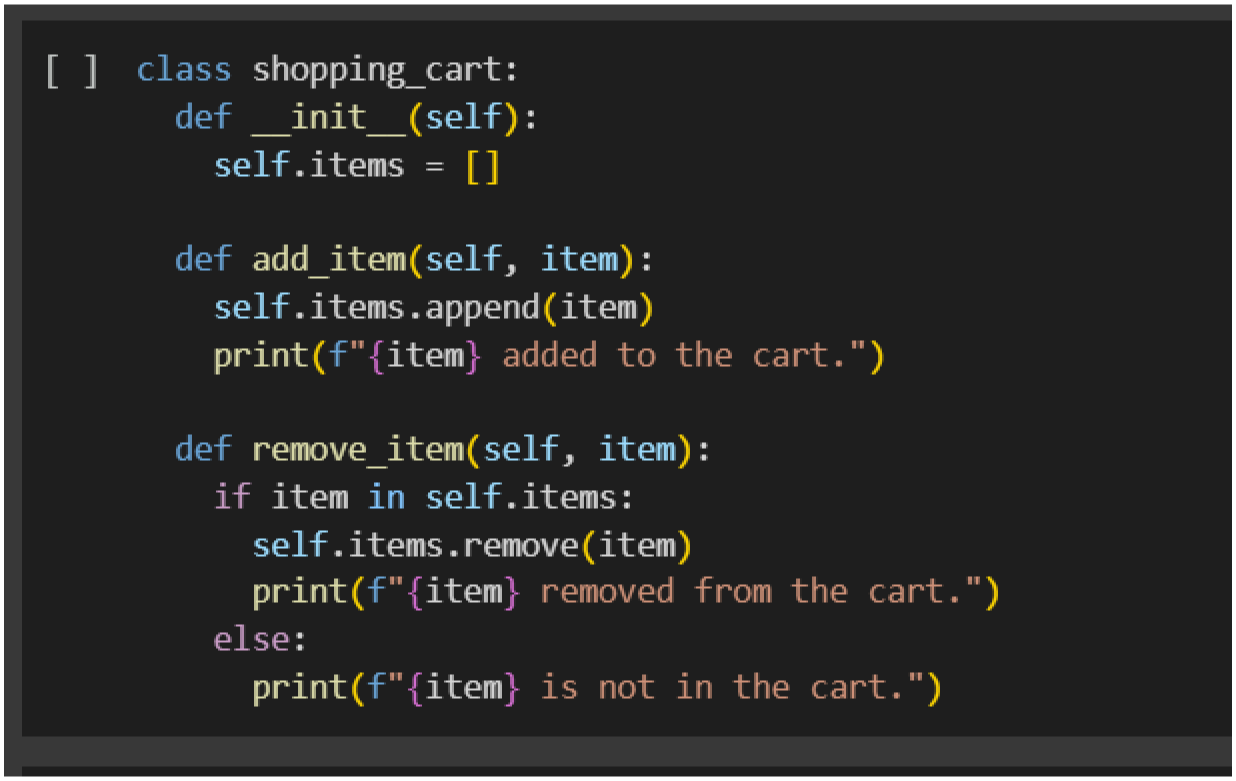
like items

#Add methods like add_item and remove_item

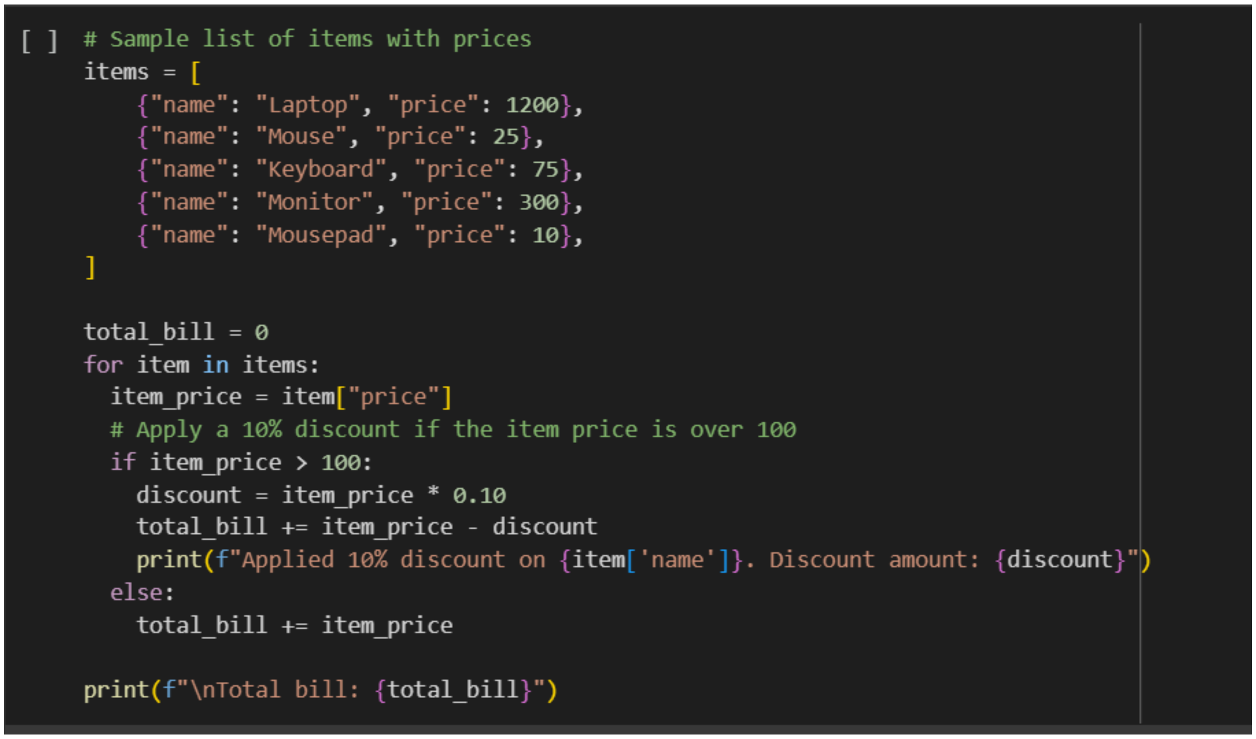
#use a loop to calculate the total bill

#use an object to check the class

CODE:

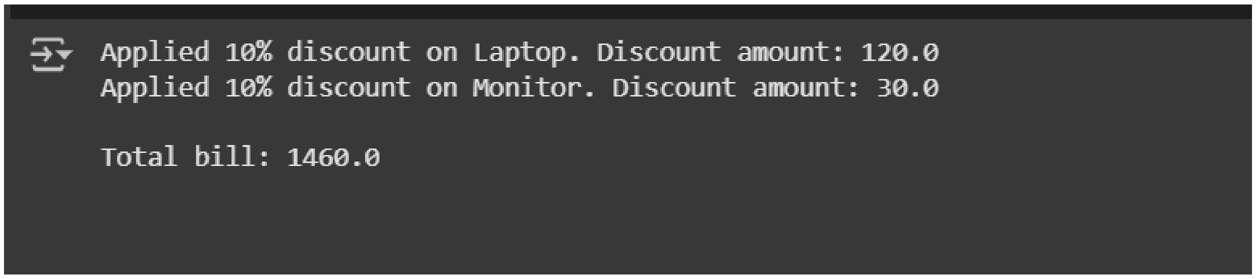


```
[ ]  class shopping_cart:
       def __init__(self):
         self.items = []

       def add_item(self, item):
         self.items.append(item)
         print(f"{item} added to the cart.")

       def remove_item(self, item):
         if item in self.items:
           self.items.remove(item)
           print(f"{item} removed from the cart.")
         else:
           print(f"{item} is not in the cart.")
```



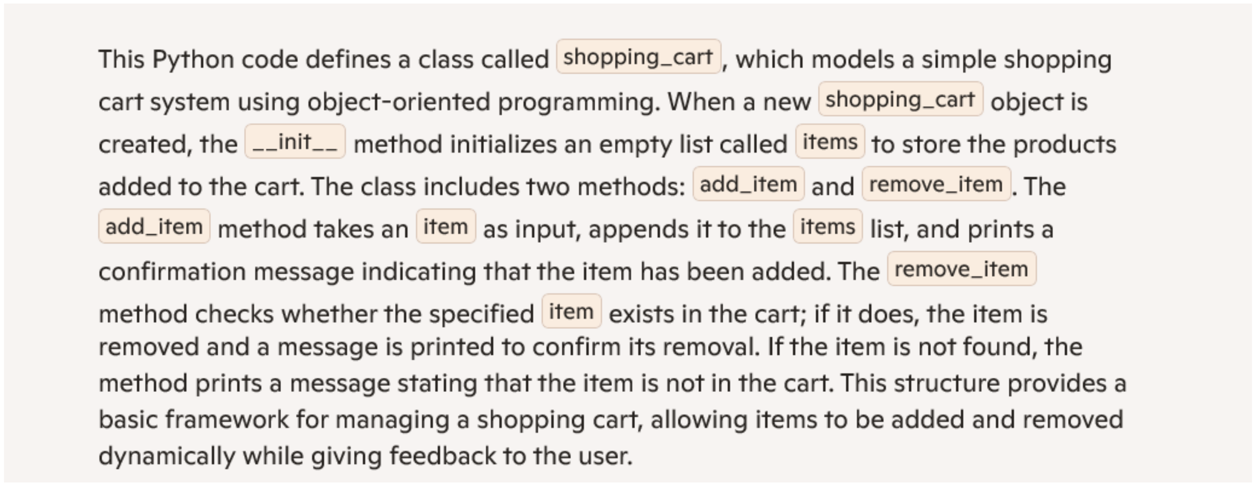
```
[ ]  # Sample list of items with prices
     items = [
         {"name": "Laptop", "price": 1200},
         {"name": "Mouse", "price": 25},
         {"name": "Keyboard", "price": 75},
         {"name": "Monitor", "price": 300},
         {"name": "Mousepad", "price": 10},
     ]

     total_bill = 0
     for item in items:
       item_price = item["price"]
       # Apply a 10% discount if the item price is over 100
       if item_price > 100:
         discount = item_price * 0.10
         total_bill += item_price - discount
         print(f"Applied 10% discount on {item['name']}. Discount amount: {discount}")
       else:
         total_bill += item_price

     print(f"\nTotal bill: {total_bill}")
```

OUTPUT:



```
Applied 10% discount on Laptop. Discount amount: 120.0
Applied 10% discount on Monitor. Discount amount: 30.0

Total bill: 1460.0
```

EXPLANTION:



This Python code defines a class called `shopping_cart`, which models a simple shopping cart system using object-oriented programming. When a new `shopping_cart` object is created, the `__init__` method initializes an empty list called `items` to store the products added to the cart. The class includes two methods: `add_item` and `remove_item`. The `add_item` method takes an `item` as input, appends it to the `items` list, and prints a confirmation message indicating that the item has been added. The `remove_item` method checks whether the specified `item` exists in the cart; if it does, the item is removed and a message is printed to confirm its removal. If the item is not found, the method prints a message stating that the item is not in the cart. This structure provides a basic framework for managing a shopping cart, allowing items to be added and removed dynamically while giving feedback to the user.