AI ASSISTED CODING

ASSIGNMENT – 6.3

COURSE : B.TECH\CSe

SPECIALISATION : AIML

NAME : MD.HEENA

ENROLLMENT NUMBER : 2403A54067

COURSE : AI ASSISTED CODING

DATE : 16 AUGUST 2025

BATCH NO : 01

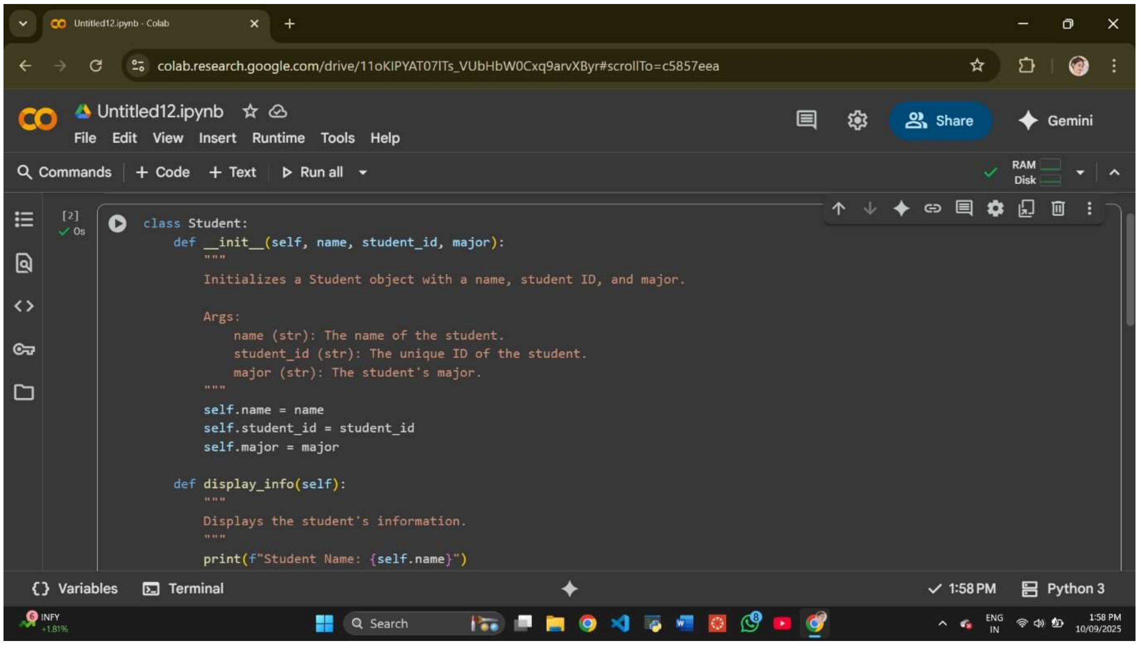
TASK DESCRIPTION#1 :

Use google gemini to generate a code to complete a Student class with attributes and a method

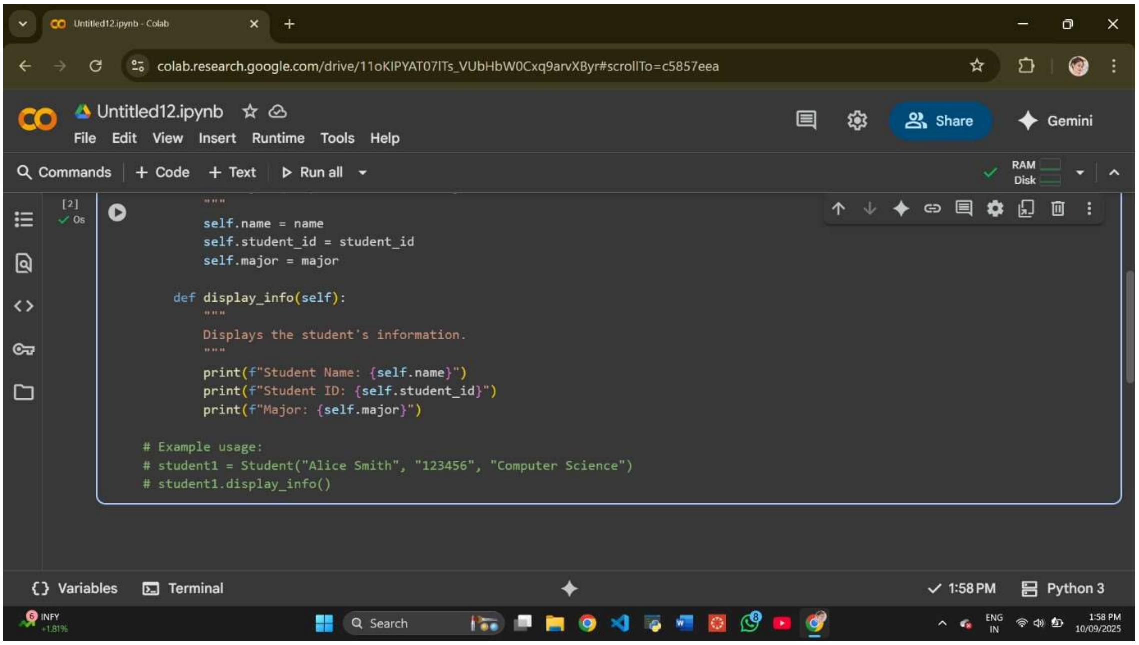
PROMPT :

Develop a python code to generate a code to complete a Student class with attributes and a method

SCREENSHOT AND EXPECTED OUTPUT :

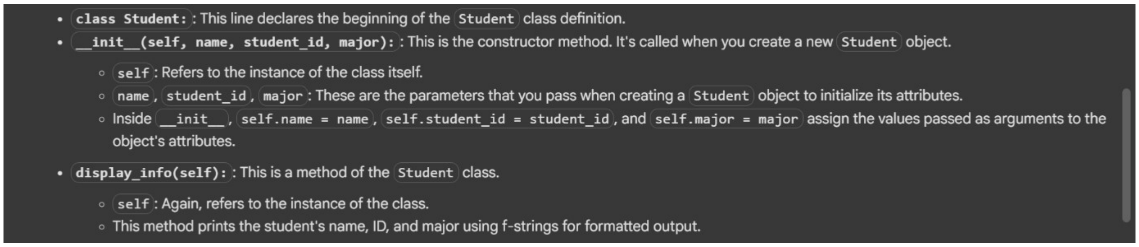

Untitled12.ipynb - Colab
colab.research.google.com/drive/11oKIPYAT07ITs_VUbHbW0Cxq9arvXByr#scrollTo=c5857eea
Untitled12.ipynb
File Edit View Insert Runtime Tools Help
Share
Gemini
Commands
+ Code
+ Text
Run all
RAM
Disk
[2]
0s
Variables
Terminal
1:58 PM
Python 3

```python
class Student:
    def __init__(self, name, student_id, major):
        """
        Initializes a Student object with a name, student ID, and major.

        Args:
            name (str): The name of the student.
            student_id (str): The unique ID of the student.
            major (str): The student's major.
        """
        self.name = name
        self.student_id = student_id
        self.major = major

    def display_info(self):
        """
        Displays the student's information.
        """
        print(f"Student Name: {self.name}")
```

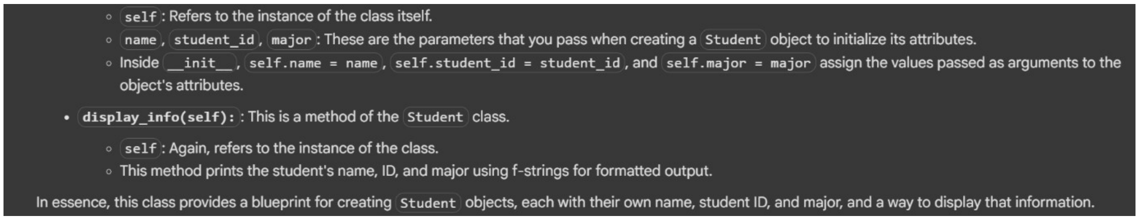

Untitled12.ipynb - Colab
colab.research.google.com/drive/11oKIPYAT07ITs_VUbHbW0Cxq9arvXByr#scrollTo=c5857eea
Untitled12.ipynb
File Edit View Insert Runtime Tools Help
Share
Gemini
Commands
+ Code
+ Text
Run all
RAM
Disk
[2]
0s
Variables
Terminal
1:58 PM
Python 3

```python
        """
        self.name = name
        self.student_id = student_id
        self.major = major

    def display_info(self):
        """
        Displays the student's information.
        """
        print(f"Student Name: {self.name}")
        print(f"Student ID: {self.student_id}")
        print(f"Major: {self.major}")

# Example usage:
# student1 = Student("Alice Smith", "123456", "Computer Science")
# student1.display_info()
```

EXPLANATION :



- `class Student:` : This line declares the beginning of the `Student` class definition.
- `__init__(self, name, student_id, major):` : This is the constructor method. It's called when you create a new `Student` object.
    - `self` : Refers to the instance of the class itself.
    - `name`, `student_id`, `major` : These are the parameters that you pass when creating a `Student` object to initialize its attributes.
    - Inside `__init__`, `self.name = name`, `self.student_id = student_id`, and `self.major = major` assign the values passed as arguments to the object's attributes.
- `display_info(self):` : This is a method of the `Student` class.
    - `self` : Again, refers to the instance of the class.
    - This method prints the student's name, ID, and major using f-strings for formatted output.



  - `self` : Refers to the instance of the class itself.
  - `name`, `student_id`, `major` : These are the parameters that you pass when creating a `Student` object to initialize its attributes.
  - Inside `__init__`, `self.name = name`, `self.student_id = student_id`, and `self.major = major` assign the values passed as arguments to the object's attributes.
- `display_info(self):` : This is a method of the `Student` class.
  - `self` : Again, refers to the instance of the class.
  - This method prints the student's name, ID, and major using f-strings for formatted output.

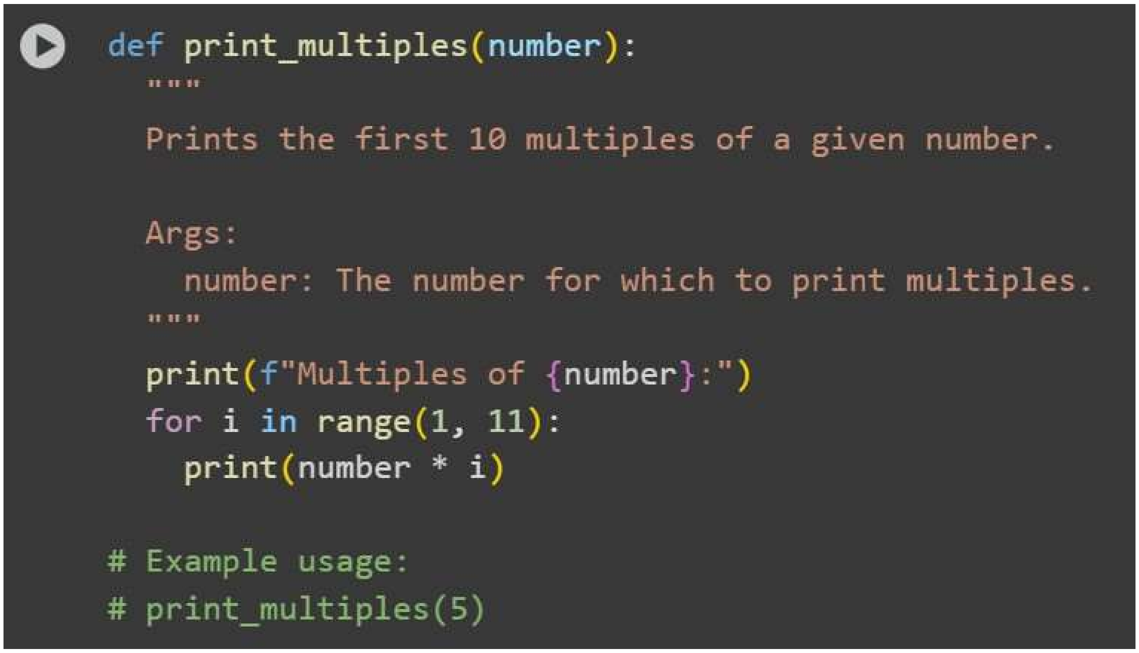
In essence, this class provides a blueprint for creating `Student` objects, each with their own name, student ID, and major, and a way to display that information.

Task Description#2

- **use google gemini to generate a python code to complete a function that prints the first 10 multiples of a number using a loop**

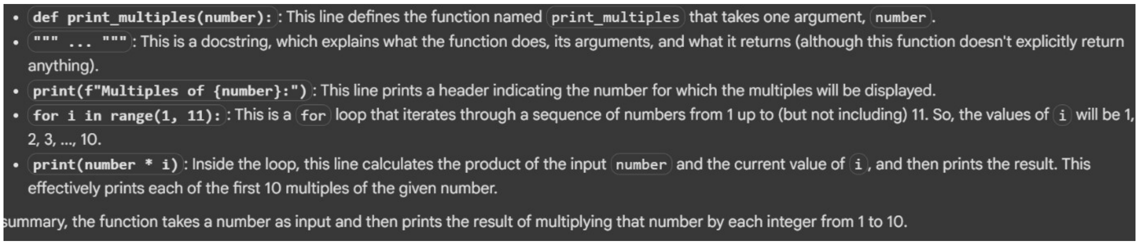
PROMPT :

Develop a python code to generate a python code to complete a function that prints the first 10 multiples of a number using a loop SCREENSHOT AND EXPECTED OUTPUT :



```
def print_multiples(number):
  """
  Prints the first 10 multiples of a given number.

  Args:
    number: The number for which to print multiples.
  """
  print(f"Multiples of {number}:")
  for i in range(1, 11):
    print(number * i)

# Example usage:
# print_multiples(5)
```

EXPLANATION :



- `def print_multiples(number):` : This line defines the function named `print_multiples` that takes one argument, `number`.
- `""" ... """` : This is a docstring, which explains what the function does, its arguments, and what it returns (although this function doesn't explicitly return anything).
- `print(f"Multiples of {number}:")` : This line prints a header indicating the number for which the multiples will be displayed.
- `for i in range(1, 11):` : This is a `for` loop that iterates through a sequence of numbers from 1 up to (but not including) 11. So, the values of `i` will be 1, 2, 3, ..., 10.
- `print(number * i)` : Inside the loop, this line calculates the product of the input `number` and the current value of `i`, and then prints the result. This effectively prints each of the first 10 multiples of the given number.

ummary, the function takes a number as input and then prints the result of multiplying that number by each integer from 1 to 10.

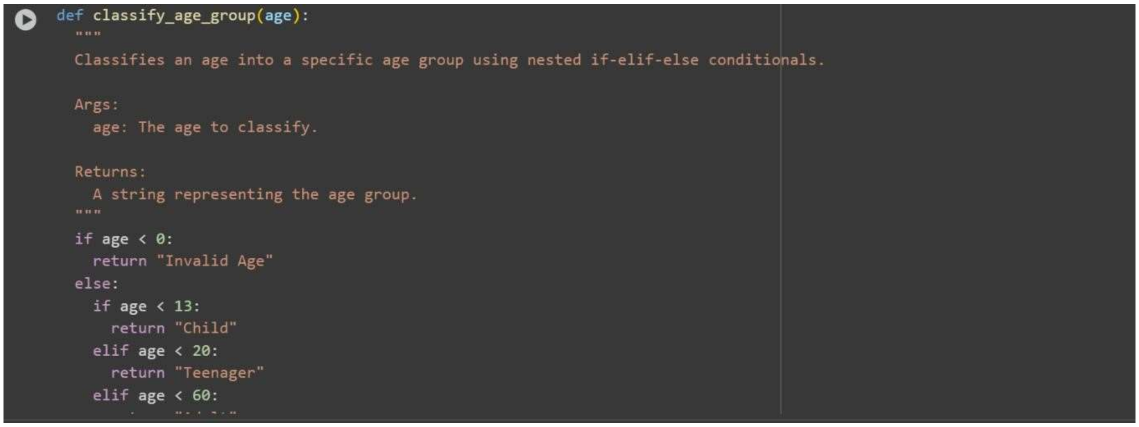
Task Description#3

- **Use google gemini to generate a python code to write nested if-elif-else conditionals to classify age groups**

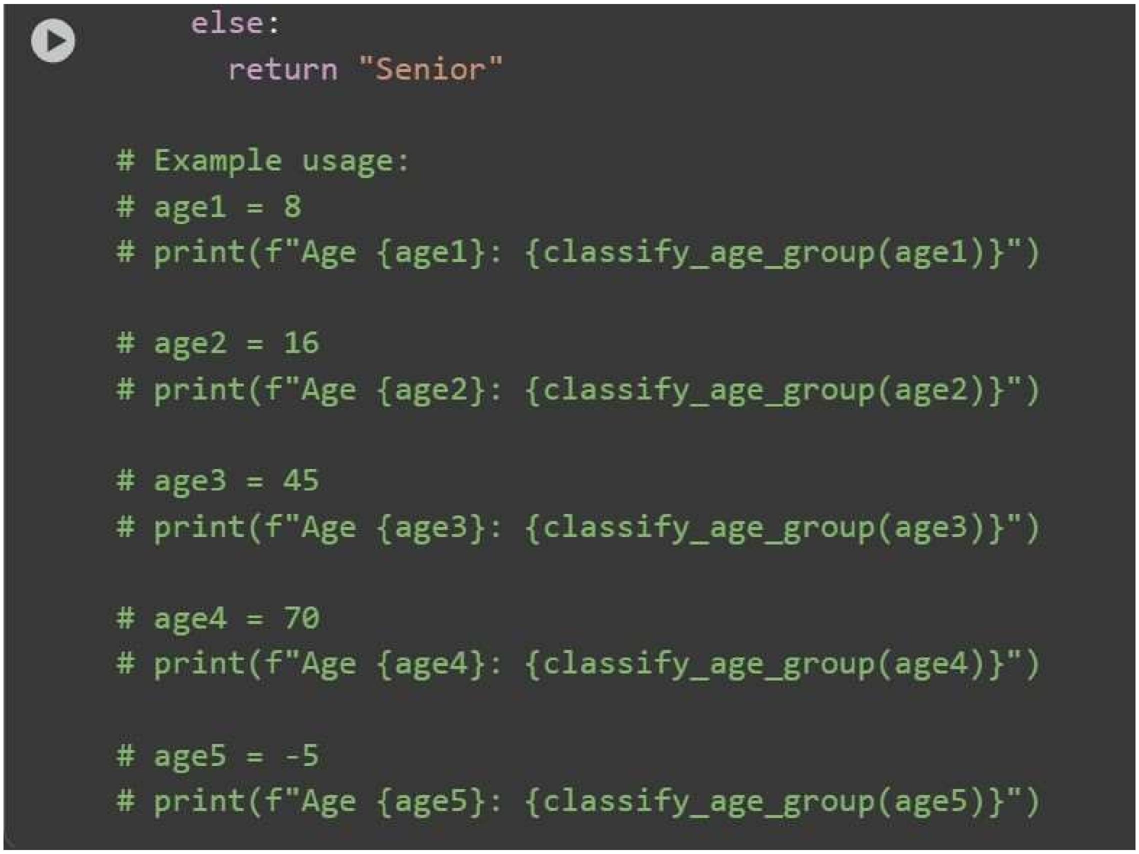
PROMPT :

Develop a python code to generate a python code to write nested if-elifelse conditionals to classify age groups

SCREENSHOT AND EXPECTED OUTPUT :

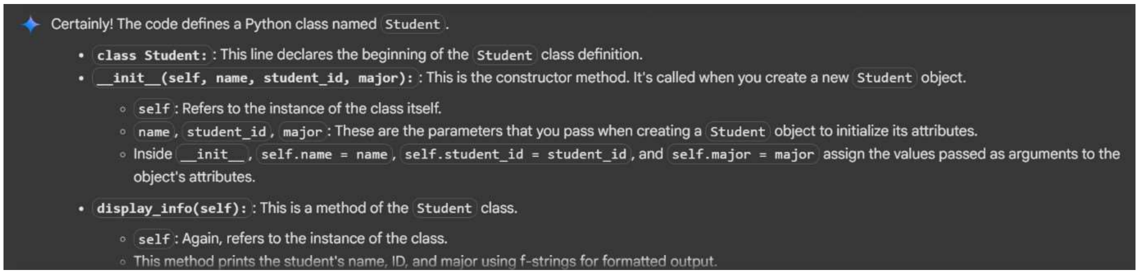


```python
def classify_age_group(age):
  """
  Classifies an age into a specific age group using nested if-elif-else conditionals.

  Args:
    age: The age to classify.

  Returns:
    A string representing the age group.
  """
  if age < 0:
    return "Invalid Age"
  else:
    if age < 13:
      return "Child"
    elif age < 20:
      return "Teenager"
    elif age < 60:
```



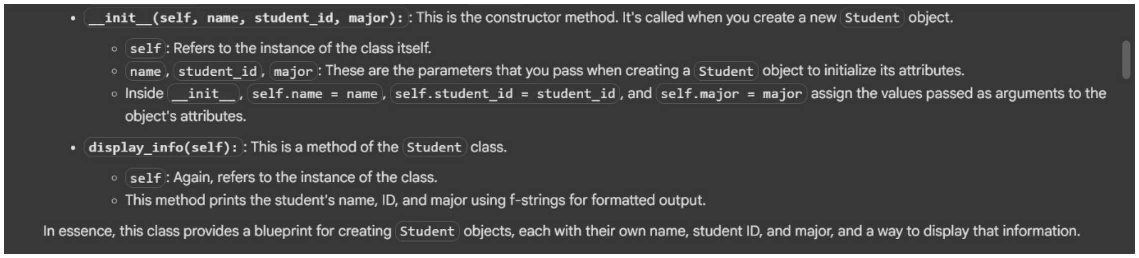
```python
    else:
      return "Senior"

# Example usage:
# age1 = 8
# print(f"Age {age1}: {classify_age_group(age1)}")

# age2 = 16
# print(f"Age {age2}: {classify_age_group(age2)}")

# age3 = 45
# print(f"Age {age3}: {classify_age_group(age3)}")

# age4 = 70
# print(f"Age {age4}: {classify_age_group(age4)}")

# age5 = -5
# print(f"Age {age5}: {classify_age_group(age5)}")
```

EXPLANATION :



Certainly! The code defines a Python class named `Student`.

- `class Student:` : This line declares the beginning of the `Student` class definition.
- `__init__(self, name, student_id, major):` : This is the constructor method. It's called when you create a new `Student` object.
  - `self` : Refers to the instance of the class itself.
  - `name`, `student_id`, `major` : These are the parameters that you pass when creating a `Student` object to initialize its attributes.
  - Inside `__init__`, `self.name = name`, `self.student_id = student_id`, and `self.major = major` assign the values passed as arguments to the object's attributes.
- `display_info(self):` : This is a method of the `Student` class.
  - `self` : Again, refers to the instance of the class.
  - This method prints the student's name, ID, and major using f-strings for formatted output.



- `__init__(self, name, student_id, major):` : This is the constructor method. It's called when you create a new `Student` object.
  - `self` : Refers to the instance of the class itself.
  - `name`, `student_id`, `major` : These are the parameters that you pass when creating a `Student` object to initialize its attributes.
  - Inside `__init__`, `self.name = name`, `self.student_id = student_id`, and `self.major = major` assign the values passed as arguments to the object's attributes.
- `display_info(self):` : This is a method of the `Student` class.
  - `self` : Again, refers to the instance of the class.
  - This method prints the student's name, ID, and major using f-strings for formatted output.

In essence, this class provides a blueprint for creating `Student` objects, each with their own name, student ID, and major, and a way to display that information.

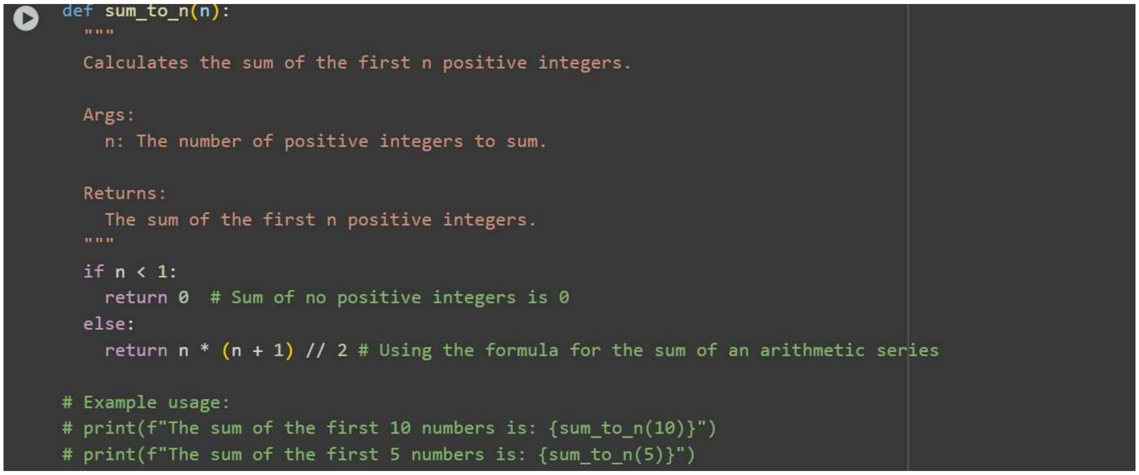
Task Description#4

Use google gemini to generate a python code to write a sum_to_n() function to calculate sum of first n numbers

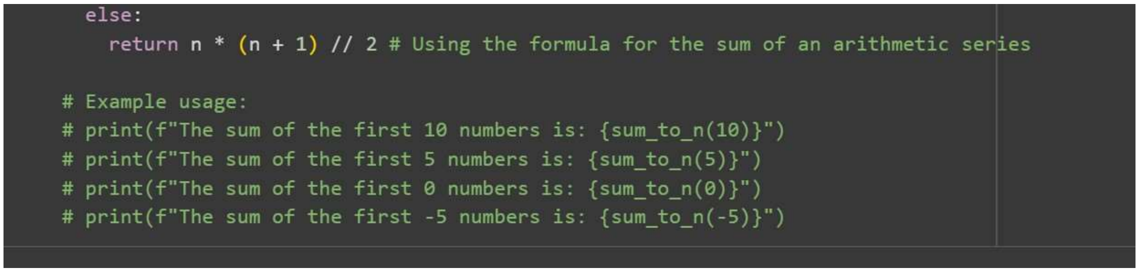
PROMPT :

Develop a a python code to write a sum_to_n() function to calculate sum of first n numbers

SCREENSHOT AND EXPECTED OUTPUT :

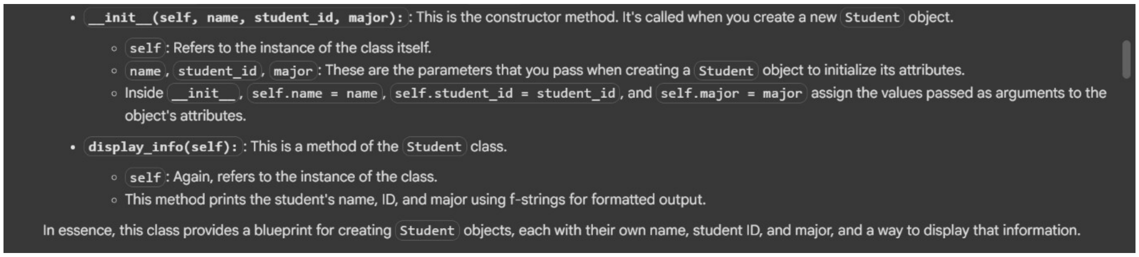


```python
def sum_to_n(n):
  """
  Calculates the sum of the first n positive integers.

  Args:
    n: The number of positive integers to sum.

  Returns:
    The sum of the first n positive integers.
  """
  if n < 1:
    return 0  # Sum of no positive integers is 0
  else:
    return n * (n + 1) // 2 # Using the formula for the sum of an arithmetic series

# Example usage:
# print(f"The sum of the first 10 numbers is: {sum_to_n(10)}")
# print(f"The sum of the first 5 numbers is: {sum_to_n(5)}")
```



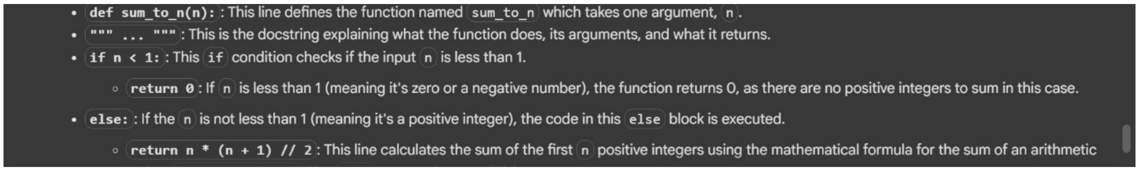
```
 else:
  return n * (n + 1) // 2 # Using the formula for the sum of an arithmetic series

# Example usage:
# print(f"The sum of the first 10 numbers is: {sum_to_n(10)}")
# print(f"The sum of the first 5 numbers is: {sum_to_n(5)}")
# print(f"The sum of the first 0 numbers is: {sum_to_n(0)}")
# print(f"The sum of the first -5 numbers is: {sum_to_n(-5)}")
```

EXPLANATION :



- `__init__(self, name, student_id, major):` : This is the constructor method. It's called when you create a new `Student` object.
  - `self` : Refers to the instance of the class itself.
  - `name`, `student_id`, `major` : These are the parameters that you pass when creating a `Student` object to initialize its attributes.
  - Inside `__init__`, `self.name = name`, `self.student_id = student_id`, and `self.major = major` assign the values passed as arguments to the object's attributes.
- `display_info(self):` : This is a method of the `Student` class.
  - `self` : Again, refers to the instance of the class.
  - This method prints the student's name, ID, and major using f-strings for formatted output.

In essence, this class provides a blueprint for creating `Student` objects, each with their own name, student ID, and major, and a way to display that information.



- `def sum_to_n(n):` : This line defines the function named `sum_to_n` which takes one argument, `n`.
- `""" ... """` : This is the docstring explaining what the function does, its arguments, and what it returns.
- `if n < 1:` : This `if` condition checks if the input `n` is less than 1.
  - `return 0` : If `n` is less than 1 (meaning it's zero or a negative number), the function returns 0, as there are no positive integers to sum in this case.
- `else:` : If the `n` is not less than 1 (meaning it's a positive integer), the code in this `else` block is executed.
  - `return n * (n + 1) // 2` : This line calculates the sum of the first `n` positive integers using the mathematical formula for the sum of an arithmetic

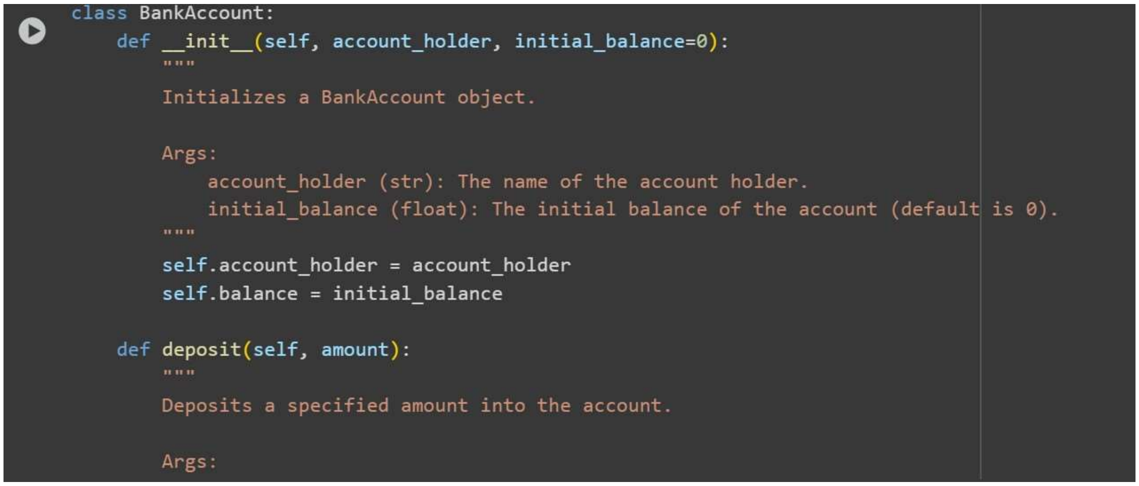
Task Description#5

Use google gemini to generate a python code to build a BankAccount class with deposit, withdraw, and balance methods

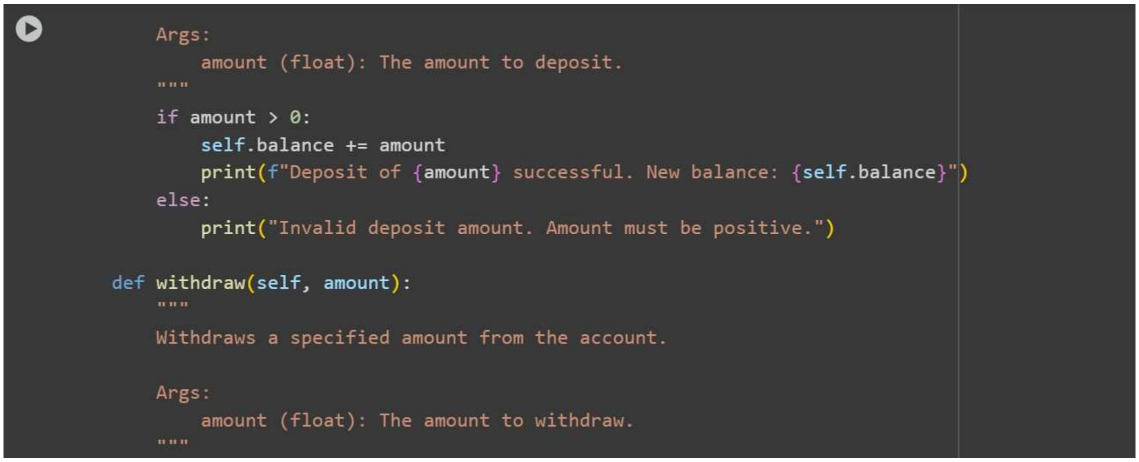
PROMPT :

Develop a python code to build a BankAccount class with deposit, withdraw, and balance methods

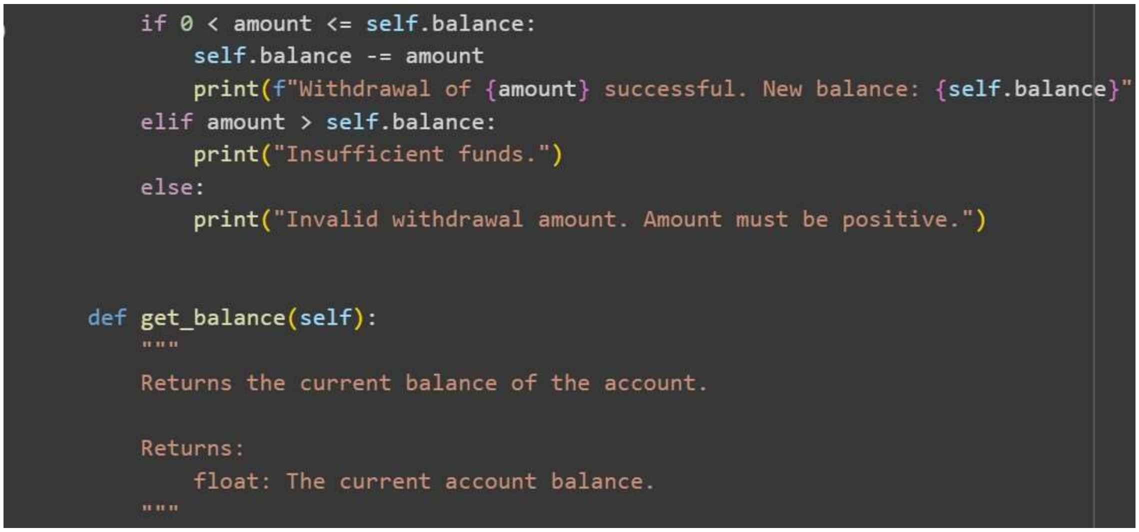
SCREENSHOT AND EXPECTED OUTPUT :



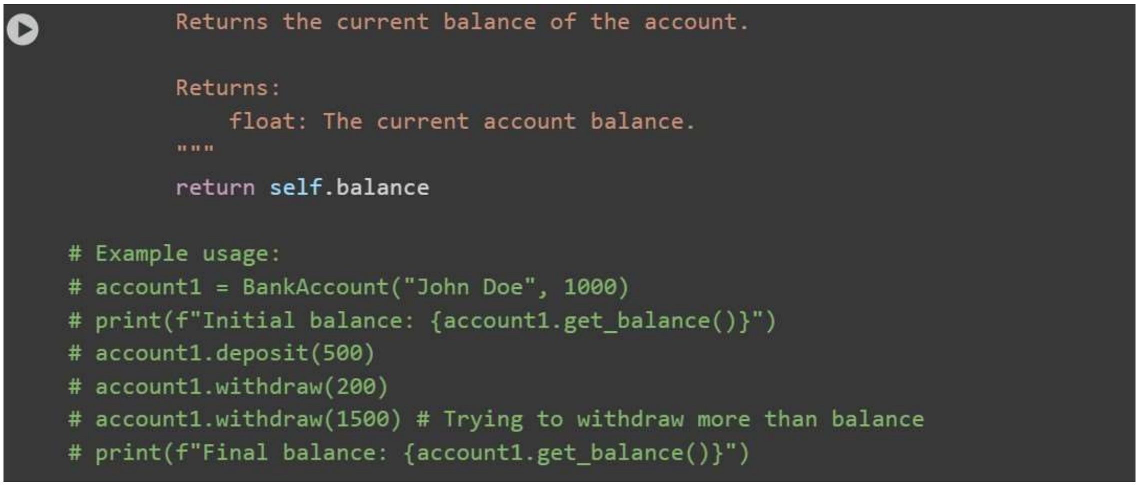
```
class BankAccount:
    def __init__(self, account_holder, initial_balance=0):
        """
        Initializes a BankAccount object.

        Args:
            account_holder (str): The name of the account holder.
            initial_balance (float): The initial balance of the account (default is 0).
        """
        self.account_holder = account_holder
        self.balance = initial_balance

    def deposit(self, amount):
        """
        Deposits a specified amount into the account.

        Args:
```



```
        Args:
            amount (float): The amount to deposit.
        """
        if amount > 0:
            self.balance += amount
            print(f"Deposit of {amount} successful. New balance: {self.balance}")
        else:
            print("Invalid deposit amount. Amount must be positive.")

    def withdraw(self, amount):
        """
        Withdraws a specified amount from the account.

        Args:
            amount (float): The amount to withdraw.
        """
```

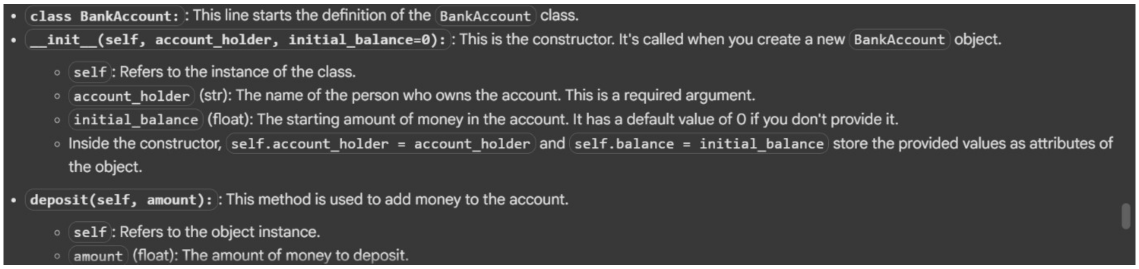


```
        if 0 < amount <= self.balance:
            self.balance -= amount
            print(f"Withdrawal of {amount} successful. New balance: {self.balance}"
        elif amount > self.balance:
            print("Insufficient funds.")
        else:
            print("Invalid withdrawal amount. Amount must be positive.")


    def get_balance(self):
        """
        Returns the current balance of the account.

        Returns:
            float: The current account balance.
        """
```

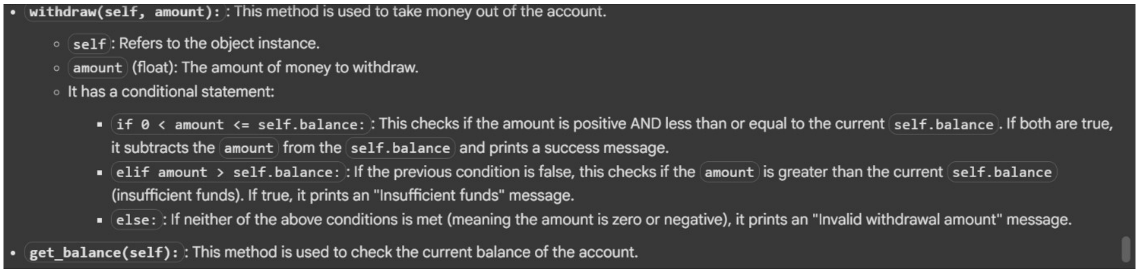


```
        Returns the current balance of the account.

        Returns:
            float: The current account balance.
        """
        return self.balance

# Example usage:
# account1 = BankAccount("John Doe", 1000)
# print(f"Initial balance: {account1.get_balance()}")
# account1.deposit(500)
# account1.withdraw(200)
# account1.withdraw(1500) # Trying to withdraw more than balance
# print(f"Final balance: {account1.get_balance()}")
```

EXPLANATION :



- **`class BankAccount:`**: This line starts the definition of the `BankAccount` class.
- **`__init__(self, account_holder, initial_balance=0):`**: This is the constructor. It's called when you create a new `BankAccount` object.
  - `self`: Refers to the instance of the class.
  - `account_holder` (str): The name of the person who owns the account. This is a required argument.
  - `initial_balance` (float): The starting amount of money in the account. It has a default value of 0 if you don't provide it.
  - Inside the constructor, `self.account_holder = account_holder` and `self.balance = initial_balance` store the provided values as attributes of the object.
- **`deposit(self, amount):`**: This method is used to add money to the account.
  - `self`: Refers to the object instance.
  - `amount` (float): The amount of money to deposit.



- **`withdraw(self, amount):`**: This method is used to take money out of the account.
  - `self`: Refers to the object instance.
  - `amount` (float): The amount of money to withdraw.
  - It has a conditional statement:
    - `if 0 < amount <= self.balance:`: This checks if the amount is positive AND less than or equal to the current `self.balance`. If both are true, it subtracts the `amount` from the `self.balance` and prints a success message.
    - `elif amount > self.balance:`: If the previous condition is false, this checks if the `amount` is greater than the current `self.balance` (insufficient funds). If true, it prints an "Insufficient funds" message.
    - `else:`: If neither of the above conditions is met (meaning the amount is zero or negative), it prints an "Invalid withdrawal amount" message.
- **`get_balance(self):`**: This method is used to check the current balance of the account.