AIAC LAB ASSIGNMENT 6.3

Name : Navadeep Munugoti

Roll.no: 2403A52015

Batch: AIAI 02

Subject: AI Assisted Coding

Task – 1

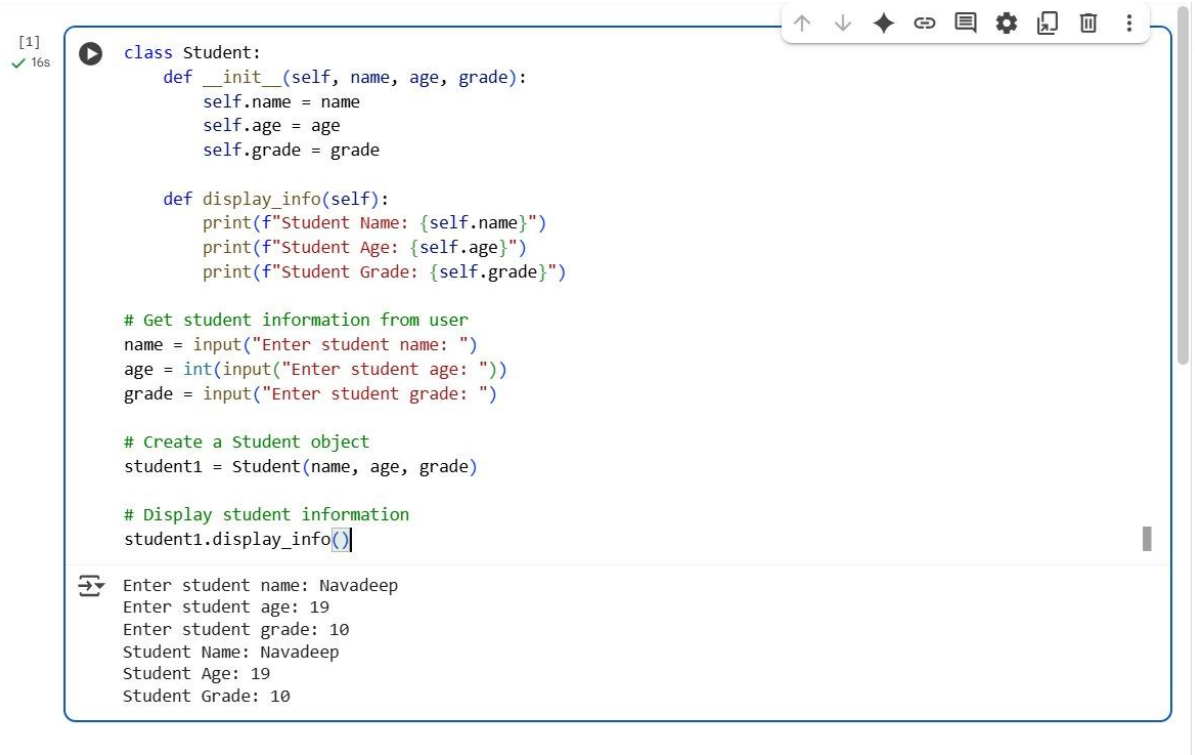
Task Description:

Use AI to complete a Student class with attributes and a method. Check output and analyze the code generated by AI tool.

Prompt:

Create a Python class named Student with the following attributes: name, age, and grade. Include a method called display_info that prints the student's name, age, and grade in a formatted string. Input should be taken from the user.

Code:



```
[1]
✓ 16s

class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def display_info(self):
        print(f"Student Name: {self.name}")
        print(f"Student Age: {self.age}")
        print(f"Student Grade: {self.grade}")

# Get student information from user
name = input("Enter student name: ")
age = int(input("Enter student age: "))
grade = input("Enter student grade: ")

# Create a Student object
student1 = Student(name, age, grade)

# Display student information
student1.display_info()

Enter student name: Navadeep
Enter student age: 19
Enter student grade: 10
Student Name: Navadeep
Student Age: 19
Student Grade: 10
```

Task – 2

Task Description:

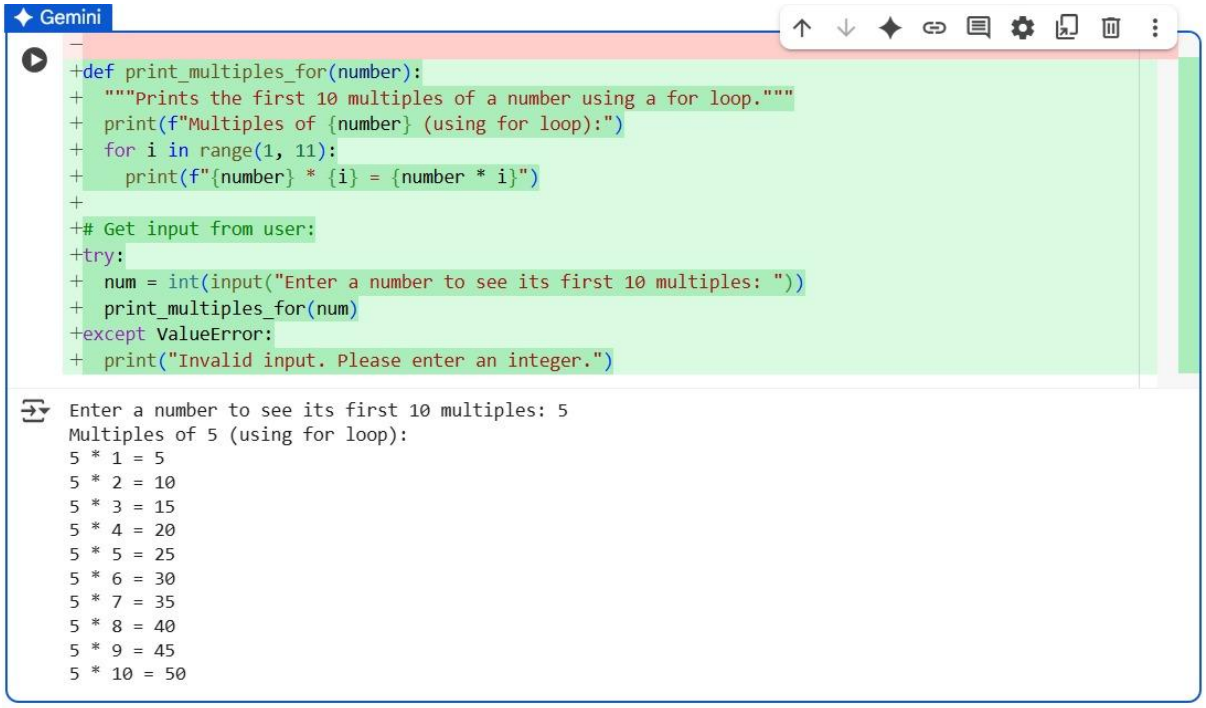
Prompt AI to complete a function that prints the first 10 multiples of a number using a
loop. Analyze the generated code and ask AI to generate code using other controlled looping.

Prompt :

Generate a Python function that takes a number as input and prints its first 10 multiples using a for loop.

After generating the code, analyze it, and then ask me to generate the same functionality using a while loop. Input should be taken from the user.

Code:



Task – 3

Task Description:

Ask AI to write nested if-elif-else conditionals to classify age groups. Analyze the generated code and ask AI to generate code using other conditional statements.

Prompt:

Generate Python code that uses nested if-elif-else statements to classify age into Child (0-12), Teenager (13-19), Adult (20-59), and Senior (60+). Take age input from the user.

Code:

Explanation:

This Python program classifies a person’s age into groups such as Child, Teenager, Adult, or Senior using nested if-elif-else conditions. The function classify_age(age) checks the given age and returns the

appropriate category. For example, ages 0–12 are classified as *Child*, 13–19 as *Teenager*, 20–59 as *Adult*, and 60+ as *Senior*. If an invalid or negative value is entered, it returns *Invalid age*. The program also includes error handling with try-except to ensure that only valid integers are accepted as input, and it displays the user's age group accordingly.

Task – 4

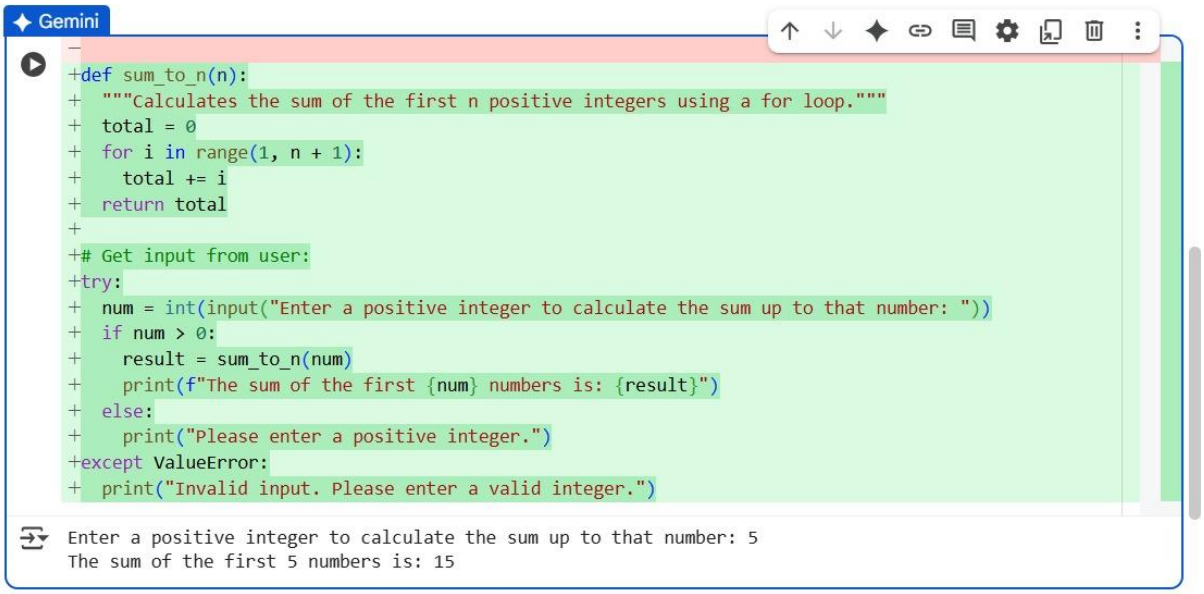
Task Description:

Generate a sum_to_n() function to calculate sum of first n numbers. Analyze the generated code and get suggestions from AI with other controlled looping.

Prompt:

Generate a Python function called sum_to_n that takes a positive integer n as input and calculates the sum of all integers from 1 to n using a loop.

Code:



Explaination:

This Python program calculates the sum of the first n positive integers using a for loop. The user enters a number, and if it’s valid and positive, the program displays the sum; otherwise, it shows an error message.

Task – 5

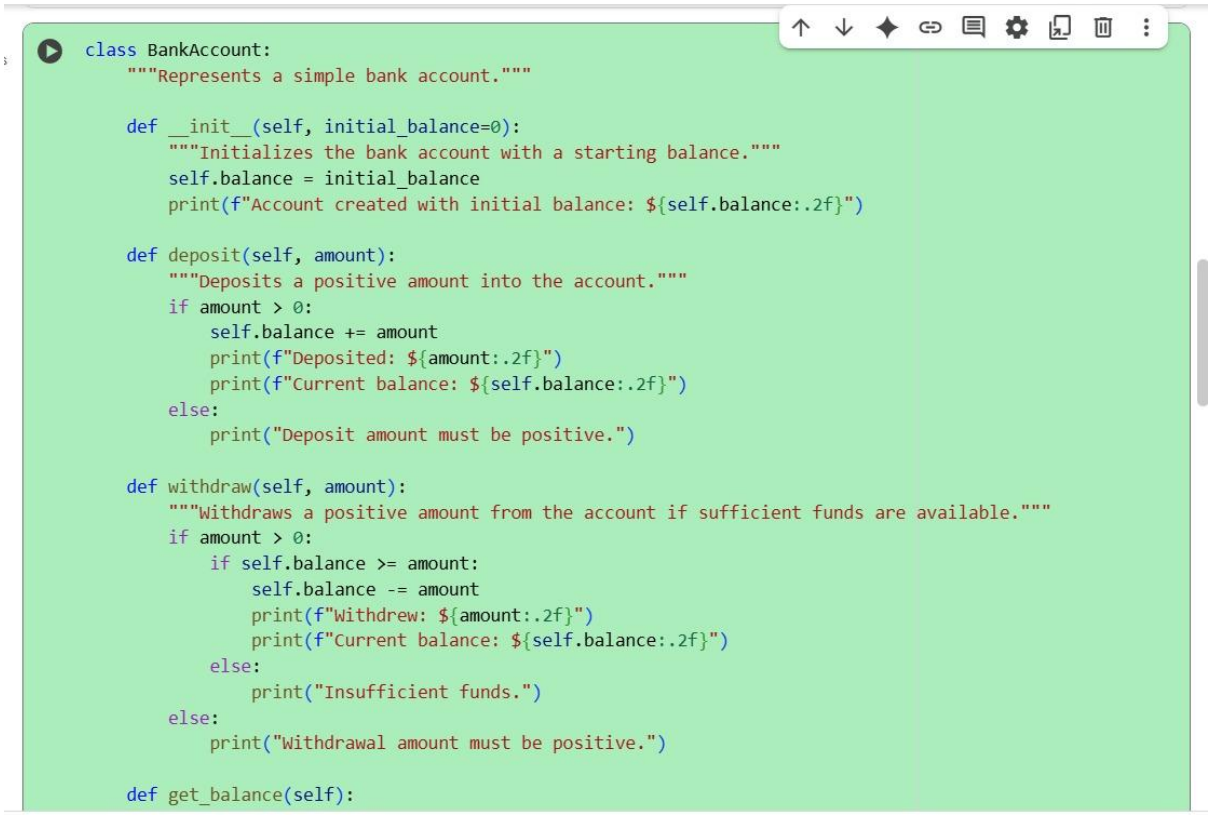
Task Description:

Use AI to build a BankAccount class with deposit, withdraw, and balance methods. Analyze the generated code and add comments and explain code.

Prompt:

Generate a Python BankAccount class with methods: deposit, withdraw, and get_balance. After generating the code, add comments and explain how it works. Take input from the user.

Code:



```python
class BankAccount:
    """Represents a simple bank account."""

    def __init__(self, initial_balance=0):
        """Initializes the bank account with a starting balance."""
        self.balance = initial_balance
        print(f"Account created with initial balance: ${self.balance:.2f}")

    def deposit(self, amount):
        """Deposits a positive amount into the account."""
        if amount > 0:
            self.balance += amount
            print(f"Deposited: ${amount:.2f}")
            print(f"Current balance: ${self.balance:.2f}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        """Withdraws a positive amount from the account if sufficient funds are available."""
        if amount > 0:
            if self.balance >= amount:
                self.balance -= amount
                print(f"Withdrew: ${amount:.2f}")
                print(f"Current balance: ${self.balance:.2f}")
            else:
                print("Insufficient funds.")
        else:
            print("Withdrawal amount must be positive.")

    def get_balance(self):
```

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

# --- User Interaction ---

# Create a bank account
account = BankAccount()

# Deposit funds
try:
    deposit_amount = float(input("Enter amount to deposit: "))
    account.deposit(deposit_amount)
except ValueError:
    print("Invalid input. Please enter a valid number for deposit.")

# Withdraw funds
try:
    withdraw_amount = float(input("Enter amount to withdraw: "))
    account.withdraw(withdraw_amount)
except ValueError:
    print("Invalid input. Please enter a valid number for withdrawal.")

# Display final balance
print(f"Final balance: ${account.get_balance():.2f}")
```

```
Account created with initial balance: $0.00
Enter amount to deposit: 500
Deposited: $500.00
Current balance: $500.00
Enter amount to withdraw: 120
```

```
# --- User Interaction ---

# Create a bank account
account = BankAccount()

# Deposit funds
try:
    deposit_amount = float(input("Enter amount to deposit: "))
    account.deposit(deposit_amount)
except ValueError:
    print("Invalid input. Please enter a valid number for deposit.")

# Withdraw funds
try:
    withdraw_amount = float(input("Enter amount to withdraw: "))
    account.withdraw(withdraw_amount)
except ValueError:
    print("Invalid input. Please enter a valid number for withdrawal.")

# Display final balance
print(f"Final balance: ${account.get_balance():.2f}")
```

```
Account created with initial balance: $0.00
Enter amount to deposit: 500
Deposited: $500.00
Current balance: $500.00
Enter amount to withdraw: 120
Withdrew: $120.00
Current balance: $380.00
Final balance: $380.00
```

Explanation:

Code Explanation:

- **`BankAccount` class:** This class defines the structure and behavior of a bank account.
- **`__init__(self, initial_balance=0)`:** This is the constructor method. It's called when you create a new `BankAccount` object. It initializes the `balance` attribute to the `initial_balance` provided (or 0 if none is given).
- **`deposit(self, amount)`:** This method handles deposits. It checks if the deposit `amount` is positive and, if so, adds it to the `balance`.
- **`withdraw(self, amount)`:** This method handles withdrawals. It checks if the withdrawal `amount` is positive and if there are enough `balance` funds. If both conditions are met, it subtracts the amount from the `balance`. Otherwise, it prints an error message.
- **`get_balance(self)`:** This method simply returns the current value of the `balance` attribute.
- **User Interaction Section:**
  - An instance of the `BankAccount` class is created.
  - The code then prompts the user to enter amounts for deposit and withdrawal using `input()`.
  - `try-except` blocks are used to handle potential `ValueError` if the user enters non-numeric input.
  - The `deposit` and `withdraw` methods are called with the user-provided amounts.
  - Finally, the `get_balance` method is called to get the final balance, which is then printed.