AI ASSISTED CODING – 6.3

NAME : L.NAVYA

HALLTICKET NUMBER : 2403A51308

BATCH.NO : 01

YEAR AND SEMESTER : 2ND AND 3RD

COURSE CODE : 24CS002PC215

TASK-1

Ask Description#1 (Classes)

• Use AI to complete a Student class with attributes and a method.

• Check output

• Analyze the code generated by AI tool

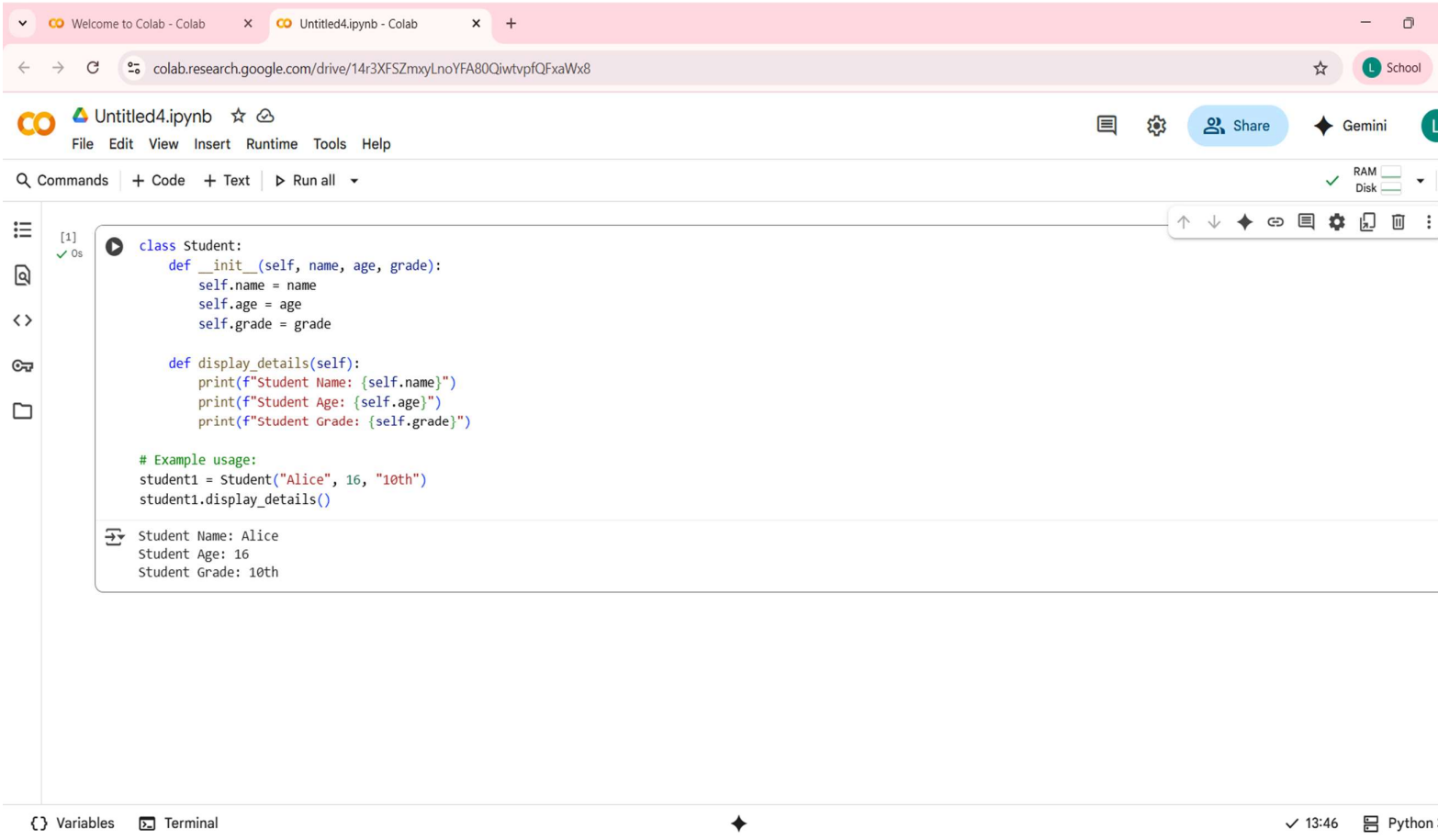
Expected Output#1

• Class with constructor and display_details() method

PROMT :

"Create a Python class called 'Student' with attributes for name, age, and grade. Include a method called 'display_details' that prints the student's information. Provide an example of how to create a Student object and call the display_details method."

Code :



OUTPUT:

Student Name: Alice

Student Age: 16

Student Grade: 10th

EXPLANATION:

This code defines a Python class named Student. Classes are blueprints for creating objects. The __init__ method is the constructor. It initializes a student object with a name, age, and grade. The display_details method prints these attributes. self refers to the specific object instance. student1 = Student("Alice", 16, "10th") creates a student object named student1. It passes "Alice", 16, and "10th" to the constructor. student1.display_details() calls the method to show Alice's details. This demonstrates basic object-oriented programming with classes and objects.

OBSERVATION:

The code defines a simple Python class Student. It uses standard object-oriented programming principles. The init method is correctly used for object initialization. Attributes (name, age, grade) are accessed using self.
The display_details method provides a clear way to present object data. F-strings are used for formatted output, which is good practice. An example object student1 is created and used. The code is easy to understand and demonstrates basic class definition. It could be extended with more methods or attributes. This serves as a good basic example of class usage in Python.

TASK-2

Task Description#2 (Loops)

• Prompt AI to complete a function that prints the first 10 multiples of a number using a
loop.

• Analyze the generated code

• Ask AI to generate code using other controlled looping

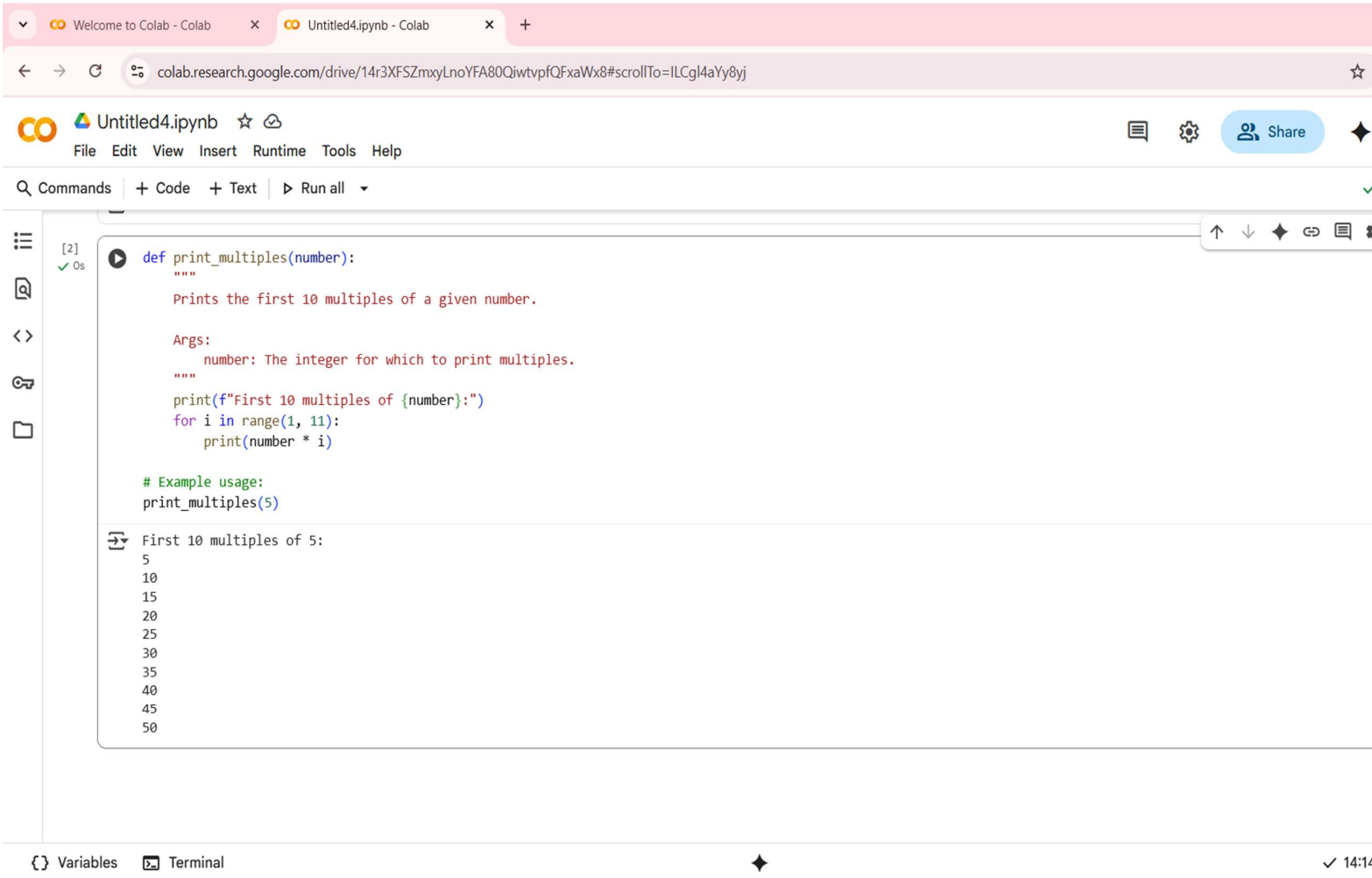
Expected Output#2

• Correct loop-based implementation.

PROMT:

"Write a Python function that takes an integer as input and prints its first 10 multiples. Use a loop to generate the multiples."

CODE:



OUTPUT:

First 10 multiples of 5:

5

10

15

20

25

30

35

40

45

50

EXPLANATION:

This code defines a Python function print_multiples. It takes one argument: number. The function calculates and prints multiples of this number. It uses a for loop to iterate 10 times. range(1, 11) generates numbers from 1 to 10. In each loop iteration, it multiplies the input number by the current loop counter. The result (the multiple) is then printed. This effectively prints the first 10 multiples. An example call print_multiples(5) demonstrates its usage. It's a simple function using a for loop for repetition.

OBSERVATION:

The function print_multiples is well-defined and clear. It takes a single integer argument as intended. A for loop is used effectively for iteration. range(1, 11) correctly generates the required sequence of numbers. The calculation number * i is straightforward and correct. The output is formatted clearly with a header. The docstring is helpful for understanding the function's purpose. The example usage demonstrates how to call the function. The code is concise and easy to read. It provides a basic but functional example of loop usage.

TASK-3

Task Description#3 (Conditional Statements)

• Ask AI to write nested if-elif-else conditionals to classify age groups.

• Analyze the generated code

• Ask AI to generate code using other conditional statements

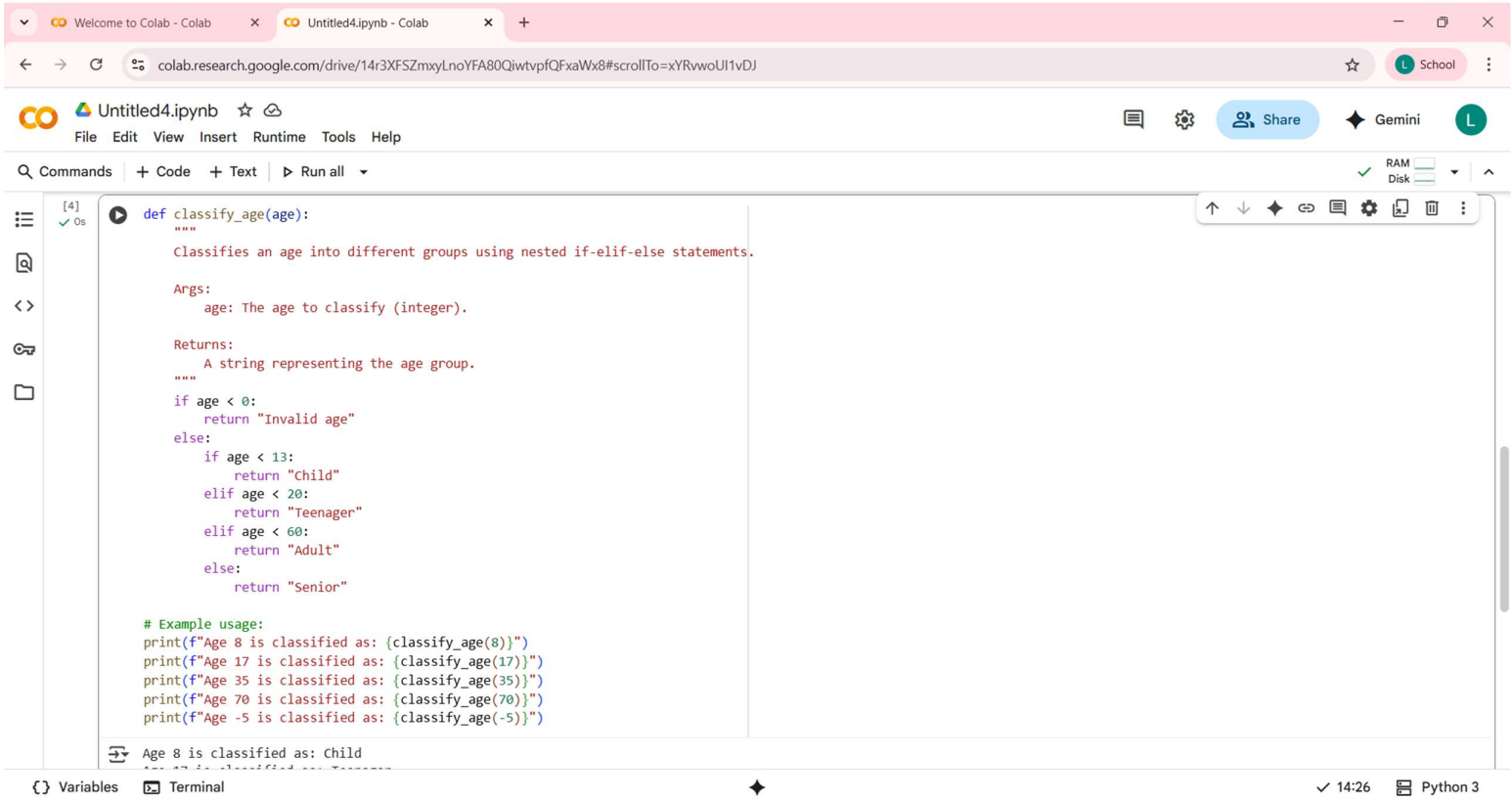
Expected Output#3

• Age classification function with appropriate conditions and with explanation

PROMT:

"Write a Python function called 'classify_age' that takes an integer representing age as input. The function should use nested if-elif-else statements to classify the age into the following groups: 'Invalid age' for negative ages, 'Child' for ages less than 13, 'Teenager' for ages less than 20, 'Adult' for ages less than 60, and 'Senior' for ages 60 or older. The function should return a string indicating the age group. Include example usage of the function."

CODE:



OUTPUT:

Age 8 is classified as: Child

Age 17 is classified as: Teenager

Age 35 is classified as: Adult

Age 70 is classified as: Senior

Age -5 is classified as: Invalid age

EXPLANATION:

This code defines a Python function classify_age. It takes an integer age as input. The function categorizes age into groups. It first checks if the age is negative (Invalid age). If not negative, it enters a nested conditional block. Inside, it checks for Child (under 13). Then Teenager (13-19). Next Adult (20-59). Finally, Senior (60 or over) is the default. The function returns the corresponding age group string.

OBSERVATION:

The code effectively classifies ages using nested conditionals. The initial check for negative age handles invalid input well. The nested structure is a valid way to represent the logic. However, a series of if-elif-else without nesting could achieve the same result. The current structure might be slightly harder to read for some. The return statements correctly exit the function once a condition is met. The age boundaries (13, 20, 60) are clearly defined in the logic. The docstring explains the function's purpose. Example usage covers various age cases, including an invalid one. This is a functional example of using conditional logic for categorization.

TASK-4

Task Description#4 (For and While loops)

• Generate a sum_to_n() function to calculate sum of first n numbers

• Analyze the generated code

• Get suggestions from AI with other controlled looping

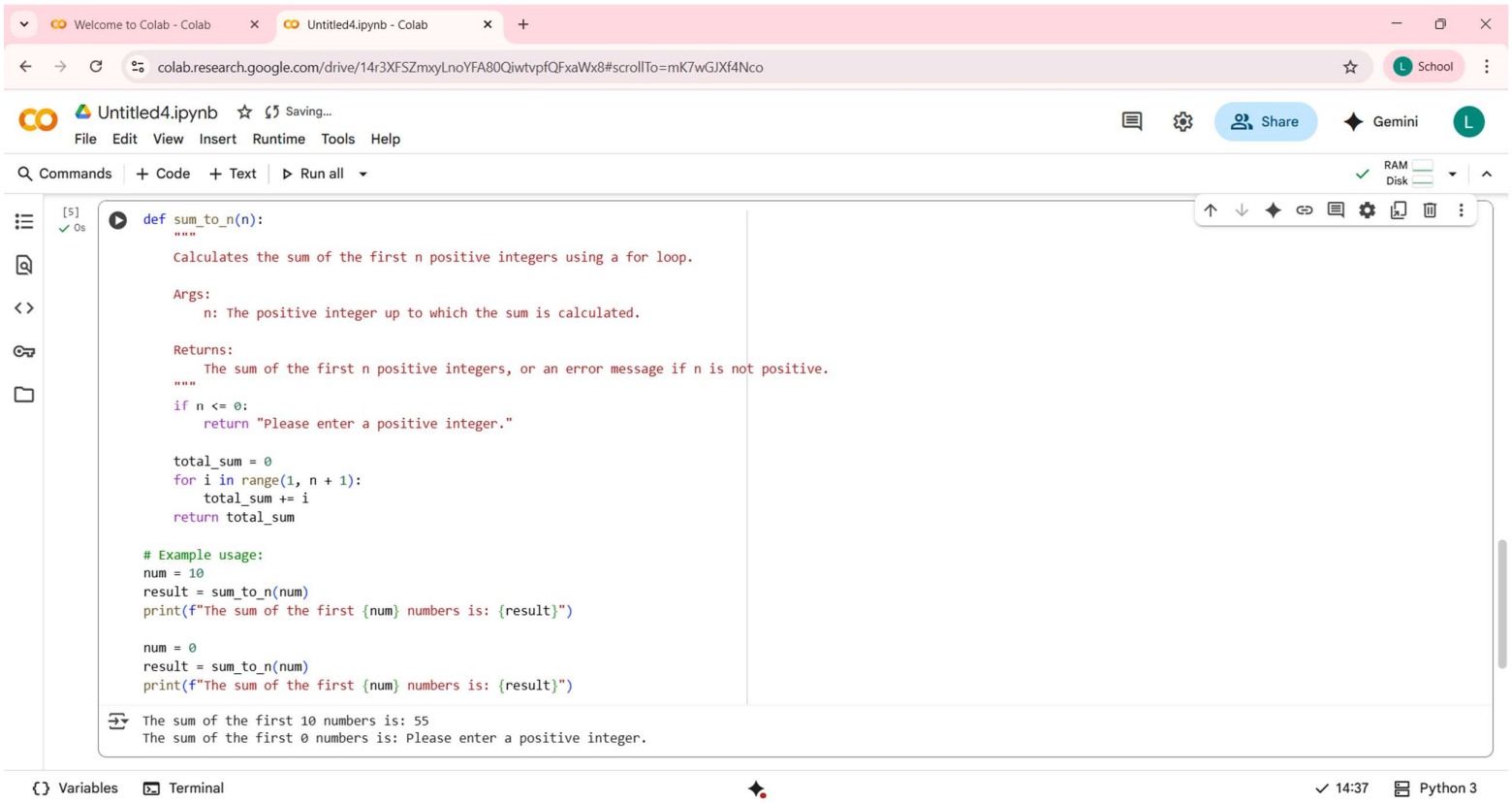
Expected Output#4

• Python code with explanation.

PROMT:

"Write a Python function called 'sum_to_n' that takes a positive integer 'n' as input. The function should calculate the sum of all positive integers from 1 up to 'n' using a loop. Handle the case where 'n' is not positive by returning an appropriate message. Include an example of how to use the function."

CODE:



OUTPUT:

The sum of the first 10 numbers is: 55

The sum of the first 0 numbers is: Please enter a positive integer.

EXPLANATION:

This code defines a Python function sum_to_n. It calculates the sum of positive integers up to n. It first checks if n is positive; otherwise, it returns an error message. A variable total_sum is initialized to zero. A for loop iterates from 1 up to n (inclusive). In each iteration, the current number is added to total_sum. This process accumulates the sum iteratively. Finally, the function returns the calculated total_sum. Example usage demonstrates calls with positive and non-positive inputs. This is a standard way to sum a range of numbers using a loop.

OBSERVATION:

The function correctly calculates the sum of the first n positive integers. It includes essential input validation for non-positive numbers. A for loop is used appropriately for iteration.

Range(1, n + 1) is the correct range for summing from 1 to n. The total_sum variable correctly accumulates the sum. The code is clear, concise, and easy to understand. It effectively demonstrates the use of a for loop for summation. An alternative approach could use a while loop. For larger n, a mathematical formula (n * (n + 1) / 2) would be more efficient. This code provides a solid basic example of loop-based summation.

TASK-5

Task Description#5 (Class)

• Use AI to build a BankAccount class with deposit, withdraw, and balance methods.

• Analyze the generated code

• Add comments and explain code

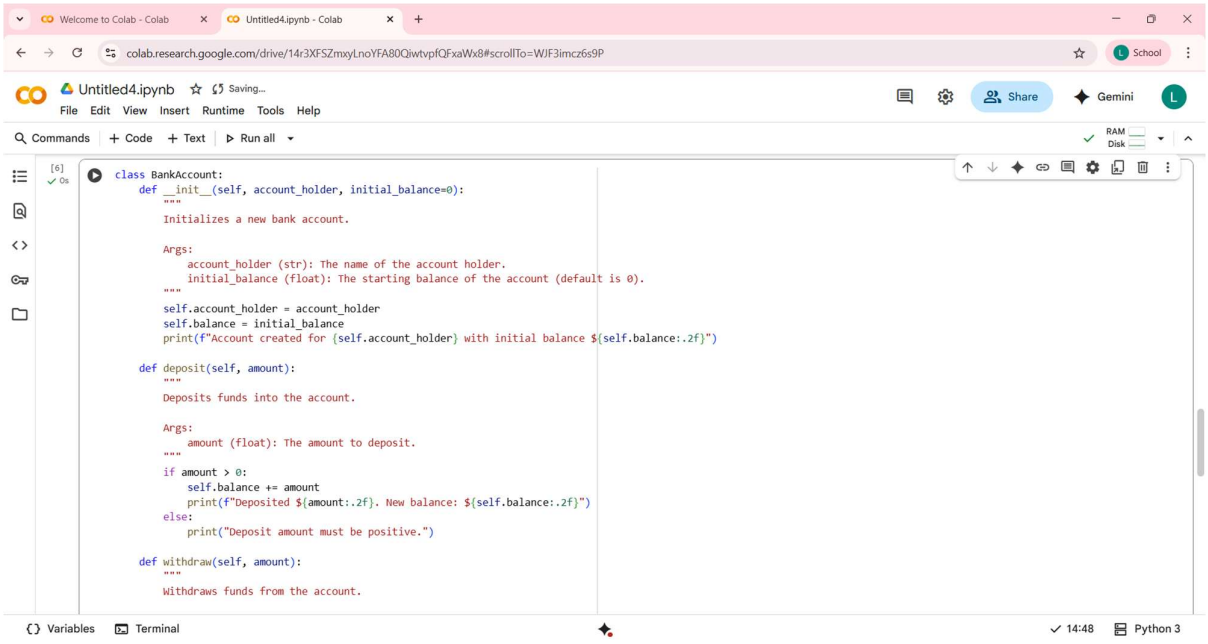
Expected Output#5

• Python code with explanation.

PROMT:

"Create a Python class called 'BankAccount'. The class should have a constructor that initializes the account with an account holder's name and an optional initial balance (defaulting to 0). Implement the following methods: 'deposit' to add funds, 'withdraw' to remove funds (checking for sufficient balance), and 'get_balance' to display the current balance. Include example usage of the class and its methods."

CODE:



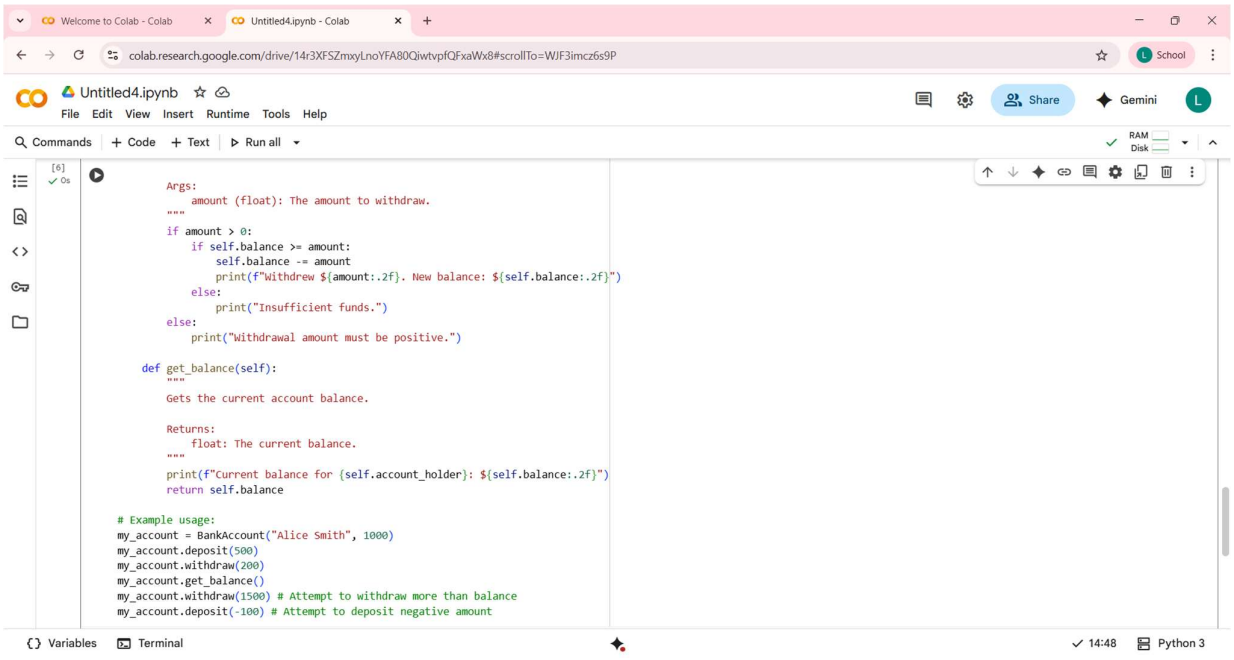
```
class BankAccount:
    def __init__(self, account_holder, initial_balance=0):
        """
        Initializes a new bank account.

        Args:
            account_holder (str): The name of the account holder.
            initial_balance (float): The starting balance of the account (default is 0).
        """
        self.account_holder = account_holder
        self.balance = initial_balance
        print(f"Account created for {self.account_holder} with initial balance ${self.balance:.2f}")

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

        Args:
            amount (float): The amount to deposit.
        """
        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):
        """
        Withdraws funds from the account.
```



```
        Args:
            amount (float): The amount to withdraw.
        """
        if amount > 0:
            if self.balance >= amount:
                self.balance -= amount
                print(f"Withdrew ${amount:.2f}. New balance: ${self.balance:.2f}")
            else:
                print("Insufficient funds.")
        else:
            print("Withdrawal amount must be positive.")

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

        Returns:
            float: The current balance.
        """
        print(f"Current balance for {self.account_holder}: ${self.balance:.2f}")
        return self.balance

# Example usage:
my_account = BankAccount("Alice Smith", 1000)
my_account.deposit(500)
my_account.withdraw(200)
my_account.get_balance()
my_account.withdraw(1500) # Attempt to withdraw more than balance
my_account.deposit(-100) # Attempt to deposit negative amount
```

OUTPUT:

"Create a Python class called 'BankAccount'. The class should have a constructor that initializes the account with an account holder's name and an optional initial balance (defaulting to 0). Implement the following methods: 'deposit' to add funds, 'withdraw' to remove funds (checking for sufficient balance), and 'get_balance' to display the current balance. Include example usage of the class and its methods."

EXPLANATION:

This code defines a Python class BankAccount. It models a simple bank account. The __init__ method creates an account with a holder and balance. The deposit method adds funds, checking for positive amounts. The withdraw method removes funds, checking for positive amounts and sufficient balance. The get_balance method displays and returns the current balance. Methods use self to access object attributes (account_holder, balance). The code demonstrates object encapsulation of data and behavior. Example usage shows creating an account and performing transactions. It's a clear example of class definition and method implementation.

OBSERVATION:

The BankAccount class is well-structured and easy to read. It correctly encapsulates account data and operations.
The __init__ method handles initial balance and account holder name. Input validation for positive deposit/withdrawal amounts is included. The withdraw method correctly checks for sufficient funds. Print statements within methods provide immediate feedback. The get_balance method serves its purpose. The code is a good basic example of object-oriented design. Potential improvements could include handling fees or transaction history. Overall, it's a functional and understandable basic bank account model.