

PROGRAM : B.TECH/CSE
SPECIALIZATION : AIML
COURSE TITLE : AI ASSISTED CODING
COURSE CODE : 24CS101PC214
SEMESTER : 3RD
NAME OF THE STUDENT : KARNAKANTI GODADEVI
ENROLLMENT NO : 2403A52003
BATCH NO : 01

Task Description#1 (Classes)

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

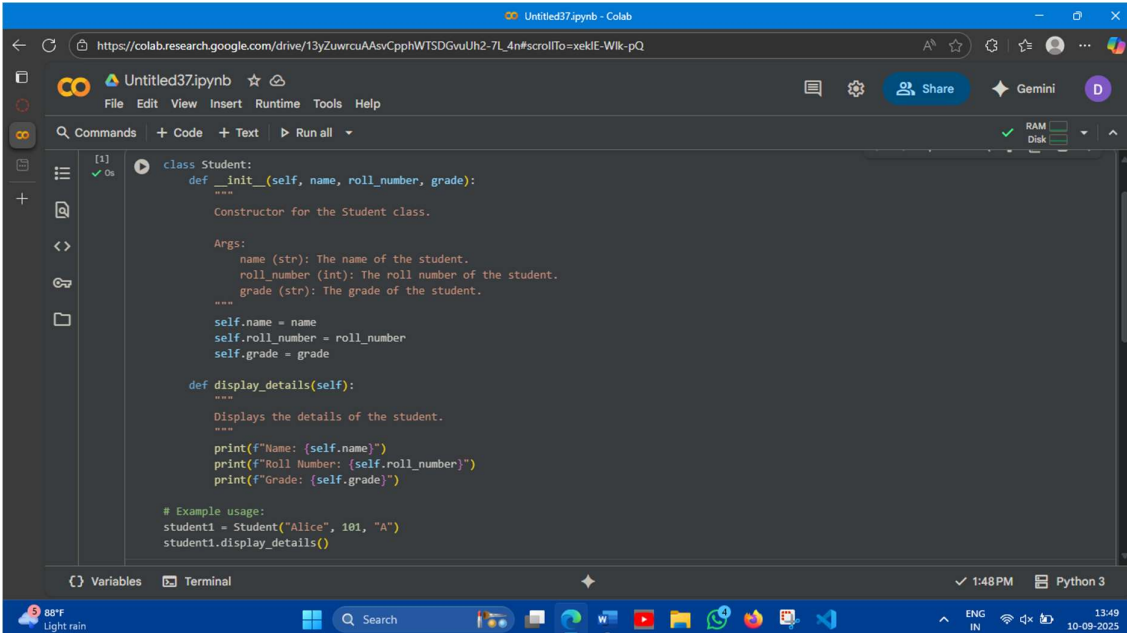
Expected Output#1

- Class with constructor and display_details() method

Prompt:

generate a code for student class with constructor , attributes and display_details() method

Code&output:



```
class Student:
    def __init__(self, name, roll_number, grade):
        """
        Constructor for the Student class.

        Args:
            name (str): The name of the student.
            roll_number (int): The roll number of the student.
            grade (str): The grade of the student.
        """
        self.name = name
        self.roll_number = roll_number
        self.grade = grade

    def display_details(self):
        """
        Displays the details of the student.
        """
        print(f"Name: {self.name}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Grade: {self.grade}")

# Example usage:
student1 = Student("Alice", 101, "A")
student1.display_details()
```

Name: Alice

Roll Number: 101

Grade: A

Code Analysis:

- **class Student::** This line defines a new class named Student. Classes are blueprints for creating objects (instances).
- **def __init__(self, name, roll_number, grade)::** This is the constructor method. It's called automatically when you create a new Student object.
 - **self:** Refers to the instance of the class being created.
 - **name, roll_number, grade:** These are parameters that you pass when creating a Student object.
 - **Inside the constructor, self.name = name, self.roll_number = roll_number, and self.grade = grade** assign the values passed as arguments to the object's attributes (properties).
- **def display_details(self)::** This defines a method named display_details within the Student class. Methods are functions that belong to a class.

- **self:** Again, refers to the instance of the class.
- This method prints the name, roll_number, and grade attributes of the Student object in a formatted string.
- **student1 = Student("Alice", 101, "A"):** This line creates an instance of the Student class.
 - Student("Alice", 101, "A") calls the constructor with the values "Alice", 101, and "A" for the name, roll_number, and grade parameters, respectively.
 - The resulting Student object is assigned to the variable student1.
- **student1.display_details():** This line calls the display_details method on the student1 object. This executes the code inside the display_details method for student1, printing its details

Task Description#2 (Loops)

- Prompt AI to complete a function that prints the first 10 multiples of a number using a

loop.

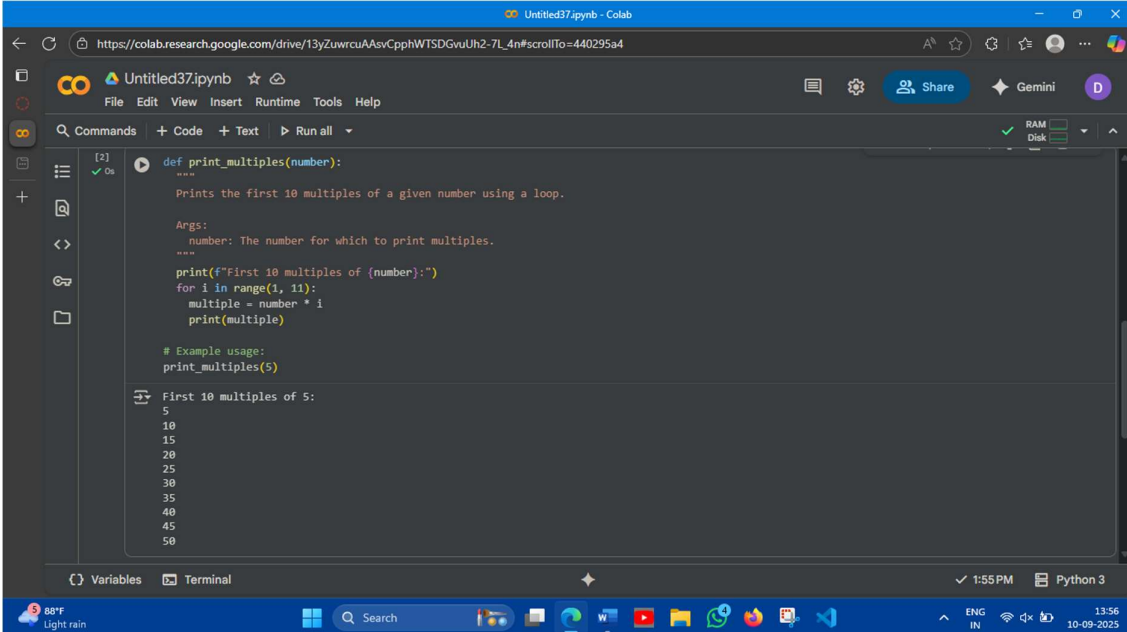
- Analyze the generated code
- Ask AI to generate code using other controlled looping

Expected Output#2

- Correct loop-based implementation

Prompt: generate code of function with loops that prints first 10 multiples of numbers

Screenshots:

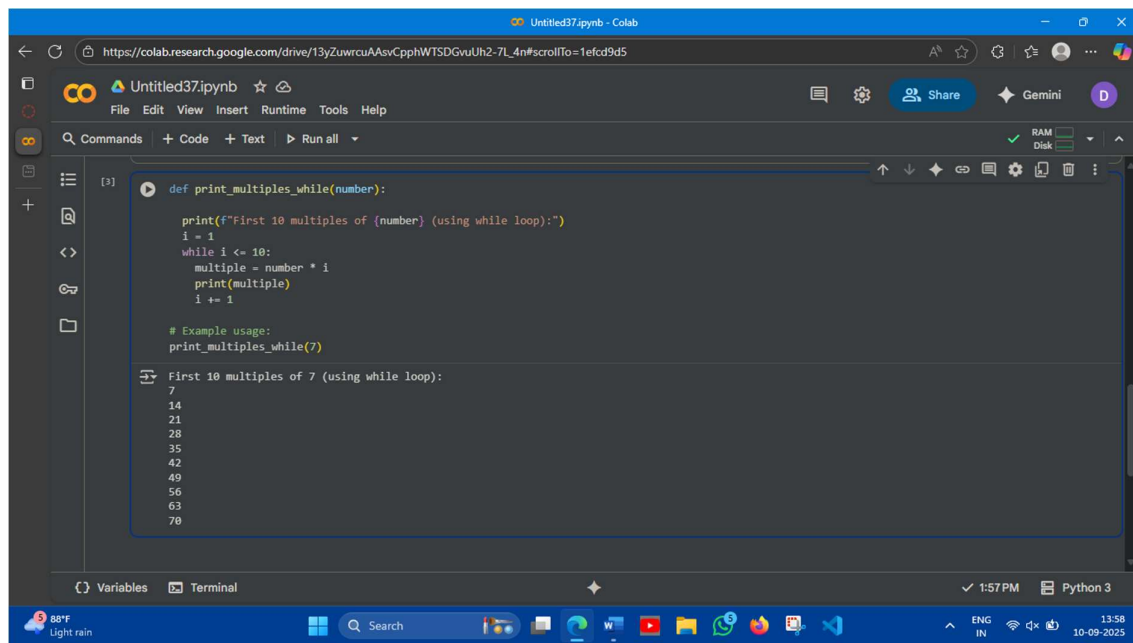


The screenshot shows a Google Colab notebook interface. The code cell contains a function definition and an example usage. The output of the code is displayed below the cell.

```
def print_multiples(number):  
    """  
    Prints the first 10 multiples of a given number using a loop.  
  
    Args:  
        number: The number for which to print multiples.  
    """  
    print(f"First 10 multiples of {number}:")  
    for i in range(1, 11):  
        multiple = number * i  
        print(multiple)  
  
# Example usage:  
print_multiples(5)
```

First 10 multiples of 5:
5
10
15
20
25
30
35
40
45
50

Generate code using other controlled looping



```
def print_multiples_while(number):  
    print(f"First 10 multiples of {number} (using while loop):")  
    i = 1  
    while i <= 10:  
        multiple = number * i  
        print(multiple)  
        i += 1  
  
# Example usage:  
print_multiples_while(7)  
  
First 10 multiples of 7 (using while loop):  
7  
14  
21  
28  
35  
42  
49  
56  
63  
70
```

Code Analysis:

- **def print_multiples_while(number)::** This line defines a function named `print_multiples_while` that takes one argument, `number`.
- **print(f"First 10 multiples of {number} (using while loop):"):** This line prints a descriptive header indicating the output that follows.
- **i = 1:** This initializes a variable `i` to 1. This variable will act as a counter for the multiples.
- **while i <= 10::** This is the while loop condition. The code block inside the while loop will continue to execute as long as the value of `i` is less than or equal to 10.

- **multiple = number * i:** Inside the loop, this line calculates the current multiple by multiplying the input number by the counter *i*.
- **print(multiple):** This line prints the calculated multiple.
- **i += 1:** This is a crucial step in a while loop. It increments the value of *i* by 1 in each iteration. This ensures that the loop will eventually terminate when *i* becomes greater than 10. If this line were missing, the loop would run indefinitely (an infinite loop).
- **print_multiples_while(7):** This line calls the `print_multiples_while` function with the argument 7, triggering the execution of the code within the function to print the first 10 multiples of 7.

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

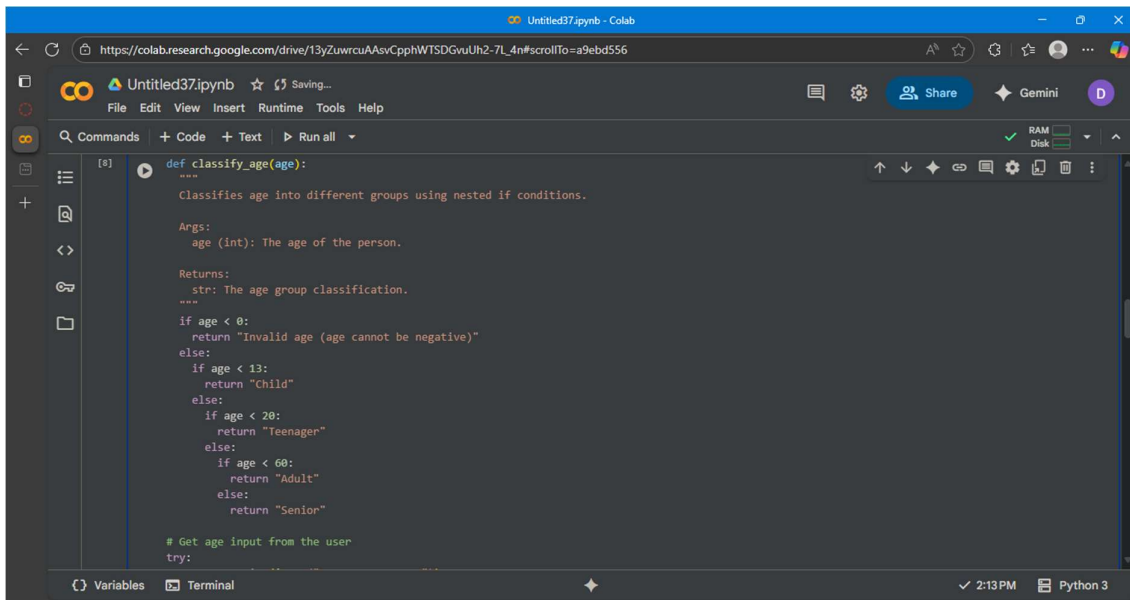
Expected Output#3

- Age classification function with appropriate conditions and with explanation

Prompt:

generate code for Age classification function with appropriate conditions and with explanation using if,elif,else conditionals to classify age groups with user given input

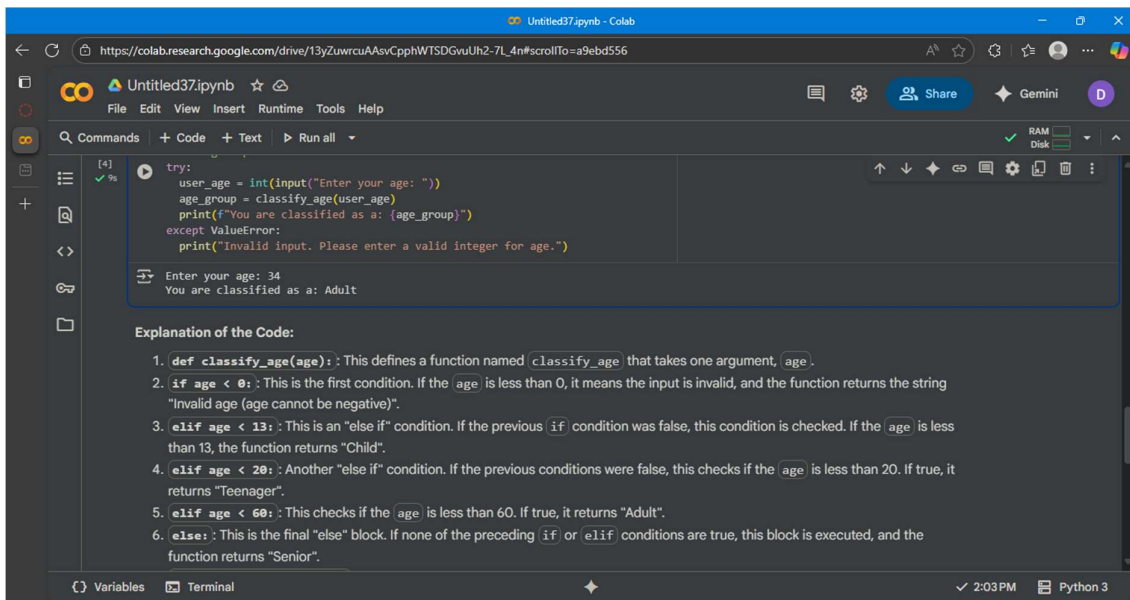
Screenshots:



The screenshot shows a Google Colab notebook titled 'Untitled37.ipynb'. The code cell contains a function `def classify_age(age):` with a docstring explaining its purpose and arguments. The function uses nested `if` and `elif` statements to classify age into groups: 'Invalid age (age cannot be negative)', 'Child', 'Teenager', 'Adult', and 'Senior'. A `try:` block is at the bottom, intended for user input.

```
[8] def classify_age(age):  
    """  
    Classifies age into different groups using nested if conditions.  
  
    Args:  
        age (int): The age of the person.  
  
    Returns:  
        str: The age group classification.  
    """  
    if age < 0:  
        return "Invalid age (age cannot be negative)"  
    elif age < 13:  
        return "Child"  
    elif age < 20:  
        return "Teenager"  
    elif age < 60:  
        return "Adult"  
    else:  
        return "Senior"  
  
    # Get age input from the user  
    try:
```

Explanation:



The screenshot shows the same Google Colab notebook after execution. The code cell now includes a `try:` block that prompts the user for their age and prints the classification. The output shows the user entered '34' and was classified as 'Adult'. Below the code, an 'Explanation of the Code:' section provides a numbered list of six points explaining the logic of the `classify_age` function.

```
[4] try:  
    user_age = int(input("Enter your age: "))  
    age_group = classify_age(user_age)  
    print(f"You are classified as a: {age_group}")  
except ValueError:  
    print("Invalid input. Please enter a valid integer for age.")  
Enter your age: 34  
You are classified as a: Adult
```

Explanation of the Code:

1. `def classify_age(age):` : This defines a function named `classify_age` that takes one argument, `age`.
2. `if age < 0:` : This is the first condition. If the `age` is less than 0, it means the input is invalid, and the function returns the string "Invalid age (age cannot be negative)".
3. `elif age < 13:` : This is an "else if" condition. If the previous `if` condition was false, this condition is checked. If the `age` is less than 13, the function returns "Child".
4. `elif age < 20:` : Another "else if" condition. If the previous conditions were false, this checks if the `age` is less than 20. If true, it returns "Teenager".
5. `elif age < 60:` : This checks if the `age` is less than 60. If true, it returns "Adult".
6. `else:` : This is the final "else" block. If none of the preceding `if` or `elif` conditions are true, this block is executed, and the function returns "Senior".

```
4. elif age < 20: : Another "else if" condition. If the previous conditions were false, this checks if the age is less than 20. If true, it returns "Teenager".
5. elif age < 60: : This checks if the age is less than 60. If true, it returns "Adult".
6. else: : This is the final "else" block. If none of the preceding if or elif conditions are true, this block is executed, and the function returns "Senior".
7. try...except ValueError: : This block handles potential errors.
    o try: : The code inside the try block is executed first.
    o user_age = int(input("Enter your age: ")) : This line prompts the user to enter their age and attempts to convert the input into an integer.
    o age_group = classify_age(user_age) : If the conversion to integer is successful, this line calls the classify_age function with the user's age.
    o print(f"You are classified as a: {age_group}") : This line prints the classified age group.
    o except ValueError: : If the user enters something that cannot be converted to an integer (e.g., text), a ValueError occurs, and the code inside the except block is executed.
    o print("Invalid input. Please enter a valid integer for age.") : This line prints an error message for invalid input.

This code effectively uses if, elif, and else to create a decision-making structure for classifying age based on different ranges.
```

This code defines a function `classify_age` that categorizes an input age. It uses `if`, `elif`, and `else` to check different age ranges. The `try...except` block handles potential invalid input from the user. It returns a string indicating the age group or an error message for negative/invalid ages.

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

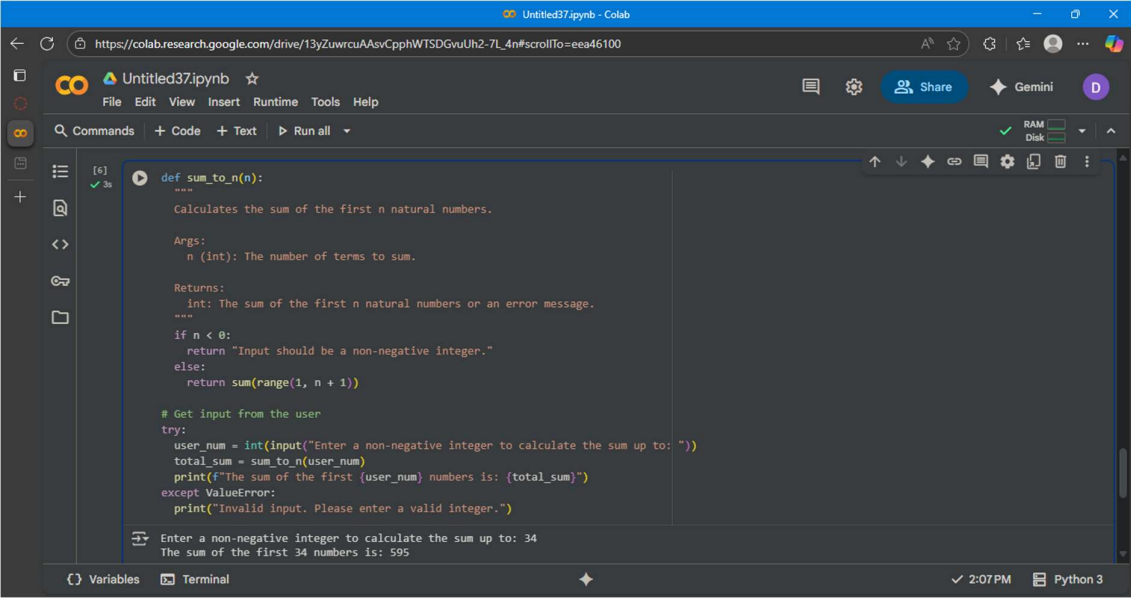
Expected Output#4

- Python code with explanation

Prompt:

Generate a `sum_to_n()` function to calculate sum of first `n` numbers with user given input

Screenshots:

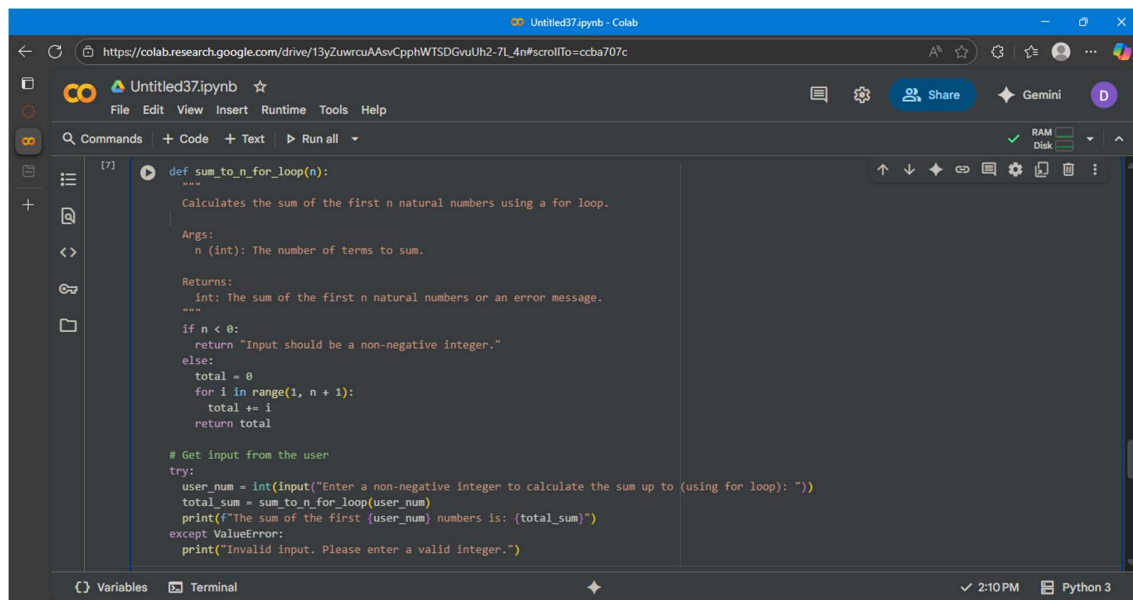


The screenshot shows a Google Colab notebook titled 'Untitled37.ipynb'. The code defines a function `sum_to_n(n)` that calculates the sum of the first `n` natural numbers. The function includes a docstring, arguments, and returns. It also includes a try-except block to handle user input errors. The output shows the user entering 34, and the function returning 595.

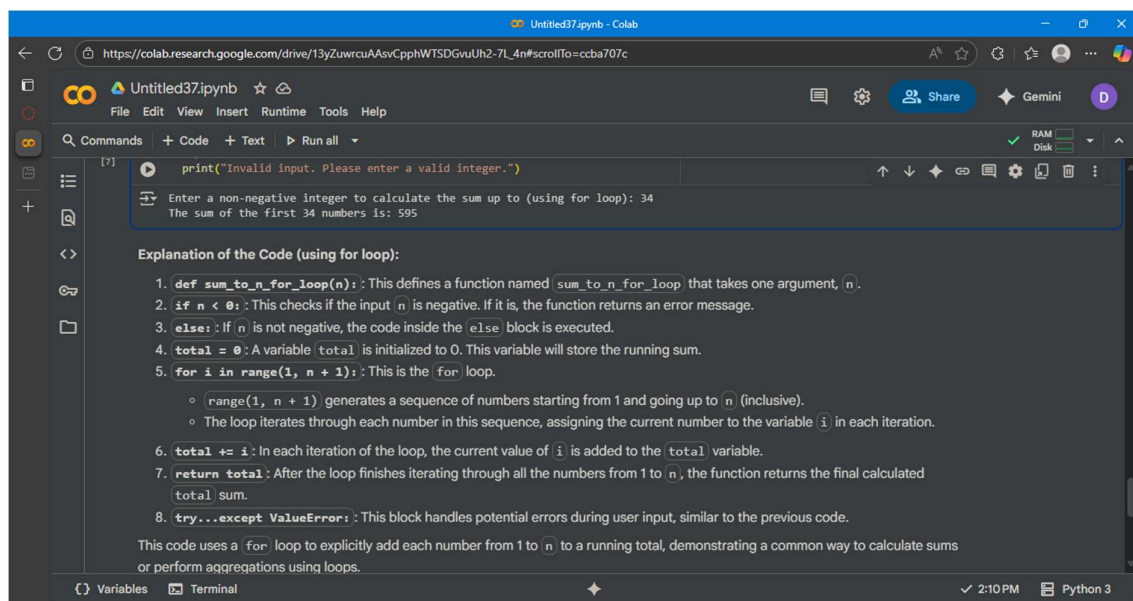
```
def sum_to_n(n):  
    """  
    Calculates the sum of the first n natural numbers.  
  
    Args:  
        n (int): The number of terms to sum.  
  
    Returns:  
        int: The sum of the first n natural numbers or an error message.  
    """  
    if n < 0:  
        return "Input should be a non-negative integer."  
    else:  
        return sum(range(1, n + 1))  
  
    # Get input from the user  
    try:  
        user_num = int(input("Enter a non-negative integer to calculate the sum up to: "))  
        total_sum = sum_to_n(user_num)  
        print(f"The sum of the first {user_num} numbers is: {total_sum}")  
    except ValueError:  
        print("Invalid input. Please enter a valid integer.")
```

Enter a non-negative integer to calculate the sum up to: 34
The sum of the first 34 numbers is: 595

generate other controlled looping with explanation



```
def sum_to_n_for_loop(n):  
    """  
    Calculates the sum of the first n natural numbers using a for loop.  
  
    Args:  
        n (int): The number of terms to sum.  
  
    Returns:  
        int: The sum of the first n natural numbers or an error message.  
    """  
    if n < 0:  
        return "Input should be a non-negative integer."  
    else:  
        total = 0  
        for i in range(1, n + 1):  
            total += i  
        return total  
  
    # Get input from the user  
    try:  
        user_num = int(input("Enter a non-negative integer to calculate the sum up to (using for loop): "))  
        total_sum = sum_to_n_for_loop(user_num)  
        print(f"The sum of the first {user_num} numbers is: {total_sum}")  
    except ValueError:  
        print("Invalid input. Please enter a valid integer.")
```



```
print("Invalid input. Please enter a valid integer.")  
Enter a non-negative integer to calculate the sum up to (using for loop): 34  
The sum of the first 34 numbers is: 595
```

Explanation of the Code (using for loop):

1. `def sum_to_n_for_loop(n):` : This defines a function named `sum_to_n_for_loop` that takes one argument, `n`.
2. `if n < 0:` : This checks if the input `n` is negative. If it is, the function returns an error message.
3. `else:` : If `n` is not negative, the code inside the `else` block is executed.
4. `total = 0:` : A variable `total` is initialized to 0. This variable will store the running sum.
5. `for i in range(1, n + 1):` : This is the `for` loop.
 - `range(1, n + 1)` generates a sequence of numbers starting from 1 and going up to `n` (inclusive).
 - The loop iterates through each number in this sequence, assigning the current number to the variable `i` in each iteration.
6. `total += i:` : In each iteration of the loop, the current value of `i` is added to the `total` variable.
7. `return total:` : After the loop finishes iterating through all the numbers from 1 to `n`, the function returns the final calculated `total` sum.
8. `try...except ValueError:` : This block handles potential errors during user input, similar to the previous code.

This code uses a `for` loop to explicitly add each number from 1 to `n` to a running total, demonstrating a common way to calculate sums or perform aggregations using loops.

Code analysis:

This function calculates the sum of the first n non-negative integers. It initializes a total to zero. A for loop iterates from 1 up to n. In each iteration, the current number is added to total. Finally, the accumulated total is returned

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

Prompt:

generate a python code for building a BankAccount class with deposit, withdraw, and balance methods with comments and explanations

screenshots:

```
class BankAccount:
    def __init__(self, account_holder, initial_balance=0):
        """
        Constructor for the BankAccount class.

        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
        print(f"Account created for {self.account_holder} with initial balance ${self.balance:.2f}")

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

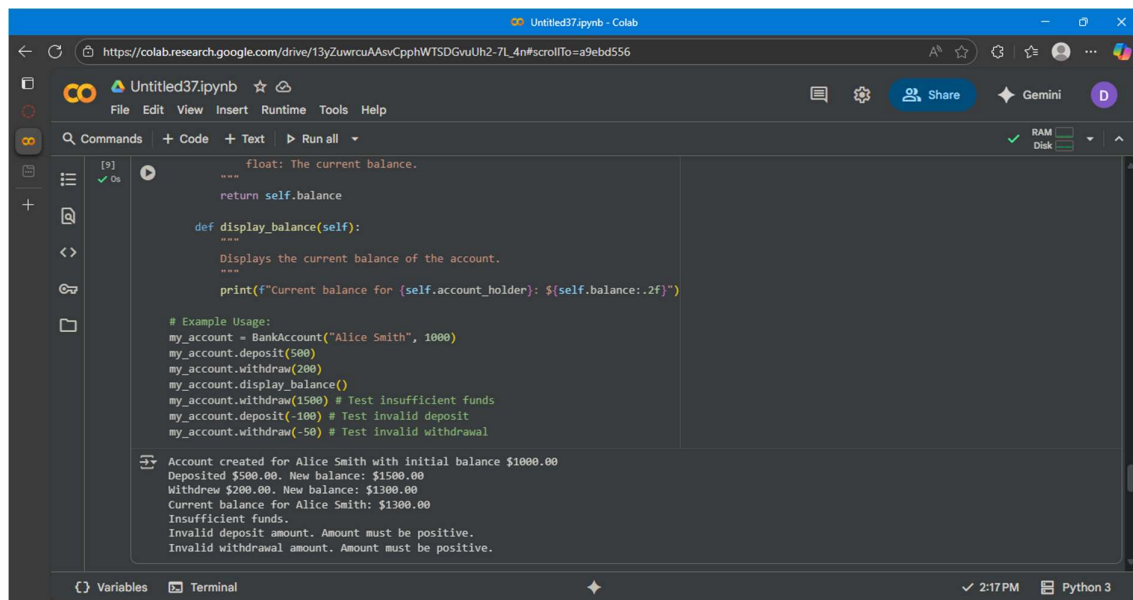
        Args:
            amount (float): The amount to deposit.
        """
        if amount > 0:
            self.balance += amount
            print(f"Deposited ${amount:.2f}. New balance: ${self.balance:.2f}")
        else:
            print("Invalid deposit amount. Amount must be positive.")
```

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

        Args:
            amount (float): The amount to withdraw.
        """
        if 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("Invalid withdrawal amount. Amount must be positive.")

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

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



```
class BankAccount:
    """
    float: The current balance.
    """
    def __init__(self, account_holder, initial_balance=0):
        self.account_holder = account_holder
        self.balance = initial_balance

    def deposit(self, amount):
        """
        Deposits the current balance of the account.
        """
        self.balance += amount

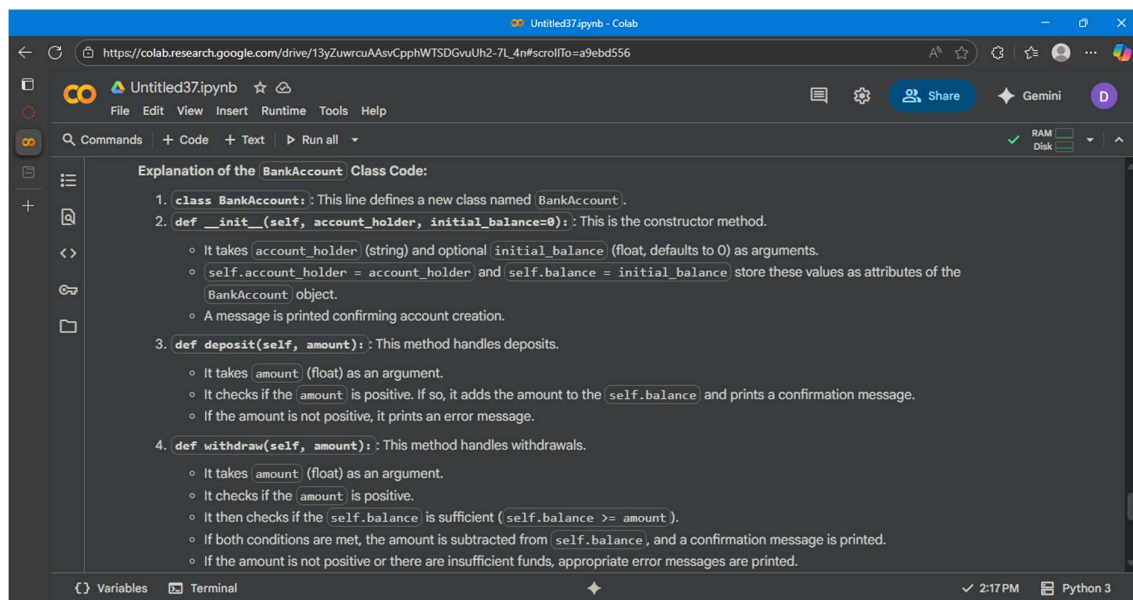
    def withdraw(self, amount):
        """
        Withdraws the current balance of the account.
        """
        self.balance -= amount

    def display_balance(self):
        """
        Displays the current balance of the account.
        """
        print(f"Current balance for {self.account_holder}: ${self.balance:.2f}")

# Example Usage:
my_account = BankAccount("Alice Smith", 1000)
my_account.deposit(500)
my_account.withdraw(200)
my_account.display_balance()
my_account.withdraw(1500) # Test insufficient funds
my_account.deposit(-100) # Test invalid deposit
my_account.withdraw(-50) # Test invalid withdrawal

Account created for Alice Smith with initial balance $1000.00
Deposited $500.00. New balance: $1500.00
Withdrew $200.00. New balance: $1300.00
Current balance for Alice Smith: $1300.00
Insufficient funds.
Invalid deposit amount. Amount must be positive.
Invalid withdrawal amount. Amount must be positive.
```

Explanation:



```
Explanation of the BankAccount Class Code:

1. class BankAccount: : This line defines a new class named BankAccount .

2. def __init__(self, account_holder, initial_balance=0): : This is the constructor method.
   o It takes account_holder (string) and optional initial_balance (float, defaults to 0) as arguments.
   o self.account_holder = account_holder and self.balance = initial_balance store these values as attributes of the BankAccount object.
   o A message is printed confirming account creation.

3. def deposit(self, amount): : This method handles deposits.
   o It takes amount (float) as an argument.
   o It checks if the amount is positive. If so, it adds the amount to the self.balance and prints a confirmation message.
   o If the amount is not positive, it prints an error message.

4. def withdraw(self, amount): : This method handles withdrawals.
   o It takes amount (float) as an argument.
   o It checks if the amount is positive.
   o It then checks if the self.balance is sufficient (self.balance >= amount ).
   o If both conditions are met, the amount is subtracted from self.balance , and a confirmation message is printed.
   o If the amount is not positive or there are insufficient funds, appropriate error messages are printed.
```

```
3. def deposit(self, amount): : This method handles deposits.
    ◦ It takes amount (float) as an argument.
    ◦ It checks if the amount is positive. If so, it adds the amount to the self.balance and prints a confirmation message.
    ◦ If the amount is not positive, it prints an error message.

4. def withdraw(self, amount): : This method handles withdrawals.
    ◦ It takes amount (float) as an argument.
    ◦ It checks if the amount is positive.
    ◦ It then checks if the self.balance is sufficient ( self.balance >= amount ).
    ◦ If both conditions are met, the amount is subtracted from self.balance , and a confirmation message is printed.
    ◦ If the amount is not positive or there are insufficient funds, appropriate error messages are printed.

5. def get_balance(self): : This method simply returns the current value of the self.balance attribute.
6. def display_balance(self): : This method prints the current balance in a formatted string.
7. # Example Usage: : This section demonstrates how to create a BankAccount object and call its methods ( deposit , withdraw ,
   display_balance ). It also includes examples of invalid operations (insufficient funds, invalid deposit/withdrawal amounts) to show
   the error handling.
```