AI Assisted Coding-6.3

Name:V.siri

Ht no:2303A51309

Batch:05

Task-01:

Prompt:

You are developing a simple student information management module.

complete a Student class.

- The class should include attributes such as name, roll number, and branch.
- Add a method display_details() to print student information.
- Execute the code and verify the output.
- Analyze the code generated by the AI tool for correctness and clarity

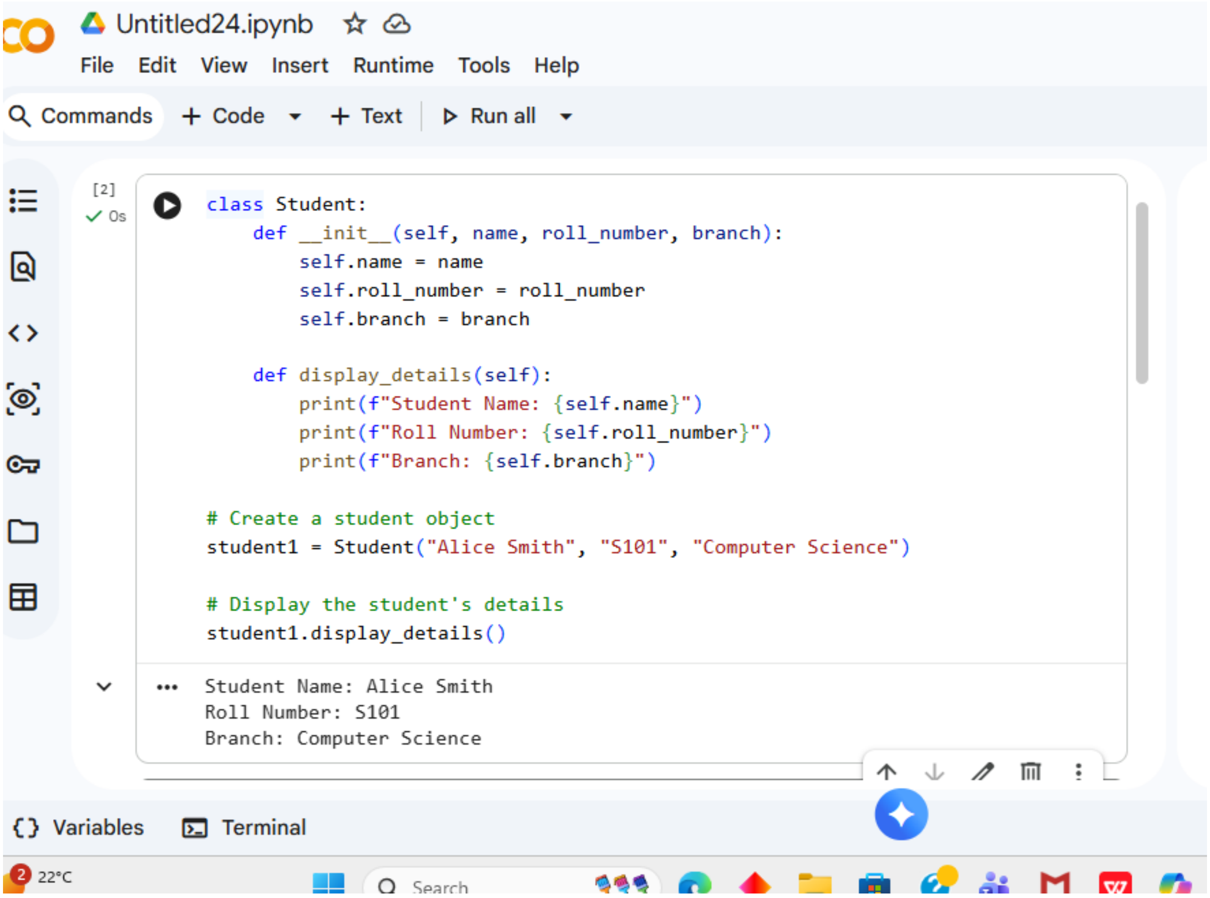
Code:

```
class Student:
    def __init__(self, name, roll_number, branch):
        self.name = name
        self.roll_number  = roll_number
        self.branch = branch
    def display_details(self):
        print(f"Student Name: {self.name}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Branch: {self.branch}")


# Create a student object
student1 = Student("Alice Smith", "S101", "Computer Science")


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

Explanation:

1. class Student:: This line defines a new class named Student.
2. def __init__(self, name, roll_number, branch):: This is the constructor method. It's automatically called when you create a new Student object. It takes name, roll_number, and branch as arguments and initializes the object's attributes with these values using self.attribute_name = value.
3. def display_details(self):: This is a method within the Student class. It prints the student's name, roll_number, and branch using an f-string for clear formatting.
4. student1 = Student("Alice Smith", "S101", "Computer Science"): This line creates an instance (an object) of the Student class. The values "Alice Smith", "S101", and "Computer Science" are passed to the __init__ method.
5. student1.display_details(): This line calls the display_details method on the student1 object, which then prints the student's information.

Implementation:



```
class Student:
    def __init__(self, name, roll_number, branch):
        self.name = name
        self.roll_number = roll_number
        self.branch = branch

    def display_details(self):
        print(f"Student Name: {self.name}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Branch: {self.branch}")

# Create a student object
student1 = Student("Alice Smith", "S101", "Computer Science")

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

```
Student Name: Alice Smith
Roll Number: S101
Branch: Computer Science
```

Task-02:

Prompt:

You are building a basic classification system based on age.

generate nested if-elif-else conditional statements to classify age groups

(e.g., child, teenager, adult, senior).

- Analyze the generated conditions and logic.

- Ask the AI to generate the same classification using alternative conditional structures (e.g.,

simplified conditions or dictionary-based logic).

Code:

```
def print_multiples_for(number):

    print(f"First 10 multiples of {number} (using for loop):")

    for i in range(1, 11):  # Loop 10 times, for multiples 1 through 10
```

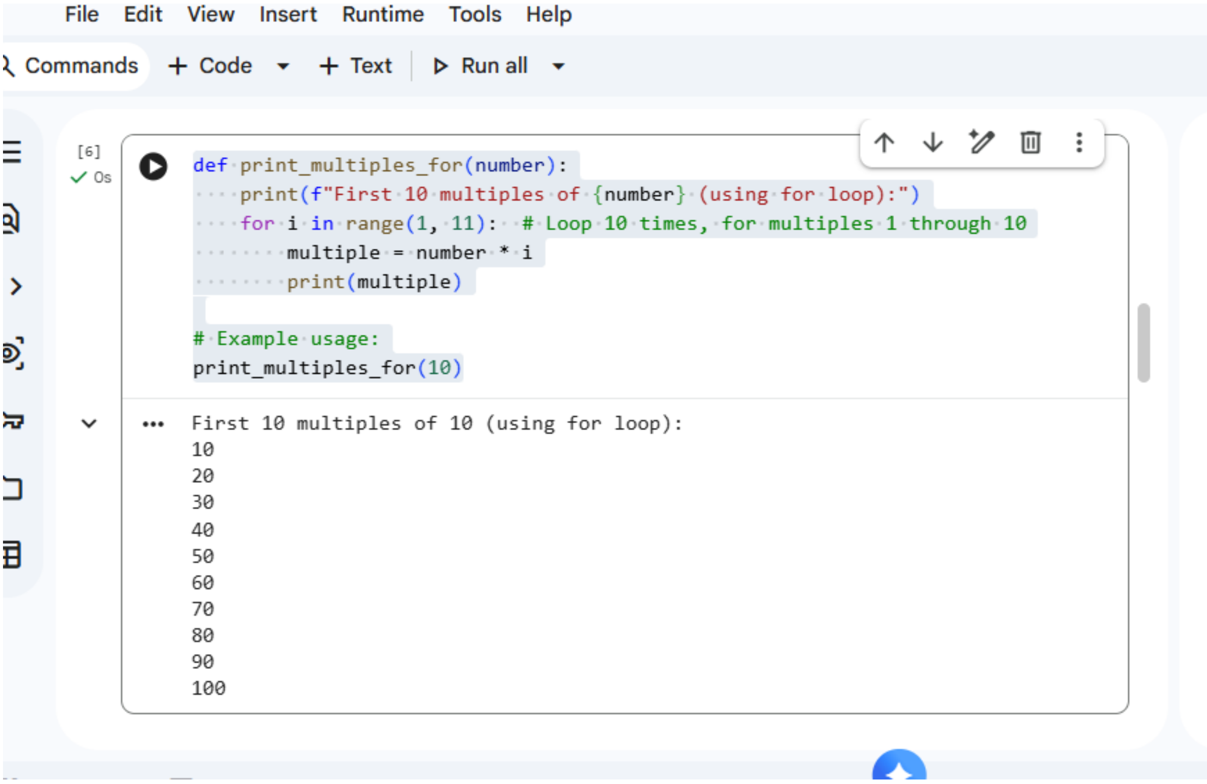
```
        multiple = number * i

        print(multiple)


# Example usage:

print_multiples_for(10)
```

Explanation:

1. def print_multiples_while(number):: Defines a function that accepts one argument, number.
2. count = 1: An initialization step is crucial for a while loop. A variable count is set to 1 to keep track of which multiple we are currently calculating.
3. while count <= 10:: This is the loop's condition. The code inside the while block will continue to execute as long as count is less than or equal to 10.
4. multiple = number * count: Similar to the for loop, the number is multiplied by the current count to get the multiple.
5. print(multiple): The calculated multiple is printed.
6. count += 1: This is the critical update step. The count is incremented by 1 in each iteration. Without this, the while loop would run indefinitely (an infinite loop) if the condition remained true.

This while loop achieves the same result but requires manual management of the loop counter and its increment, which is suitable when the number of iterations isn't precisely known beforehand, but rather depends on a condition.

Implementation:



```
def print_multiples_for(number):
    print(f"First 10 multiples of {number} (using for loop):")
    for i in range(1, 11):  # Loop 10 times, for multiples 1 through 10
        multiple = number * i
        print(multiple)

# Example usage:
print_multiples_for(10)
```

```
First 10 multiples of 10 (using for loop):
10
20
30
40
50
60
70
80
90
100
```

Task-03:

Prompt:

You are building a basic classification system based on age.

generate nested if-elif-else conditional statements to classify age groups

(e.g., child, teenager, adult, senior).

- Analyze the generated conditions and logic.

- Ask the AI to generate the same classification using alternative conditional structures (e.g.,

simplified conditions or dictionary-based logic).

Code:

```
def classify_age_nested(age):

    if age >= 0:

        if age <= 12:

            return "Child"

        elif age <= 19:
            return "Teenager"
        elif age <= 64:
            return "Adult"
        else:
            return "Senior"
    else:
        return "Invalid Age"

# Test cases
print(f"Age 5: {classify_age_nested(5)}")
print(f"Age 15: {classify_age_nested(15)}")
print(f"Age 30: {classify_age_nested(30)}")
print(f"Age 70: {classify_age_nested(70)}")
print(f"Age -2: {classify_age_nested(-2)}")
```

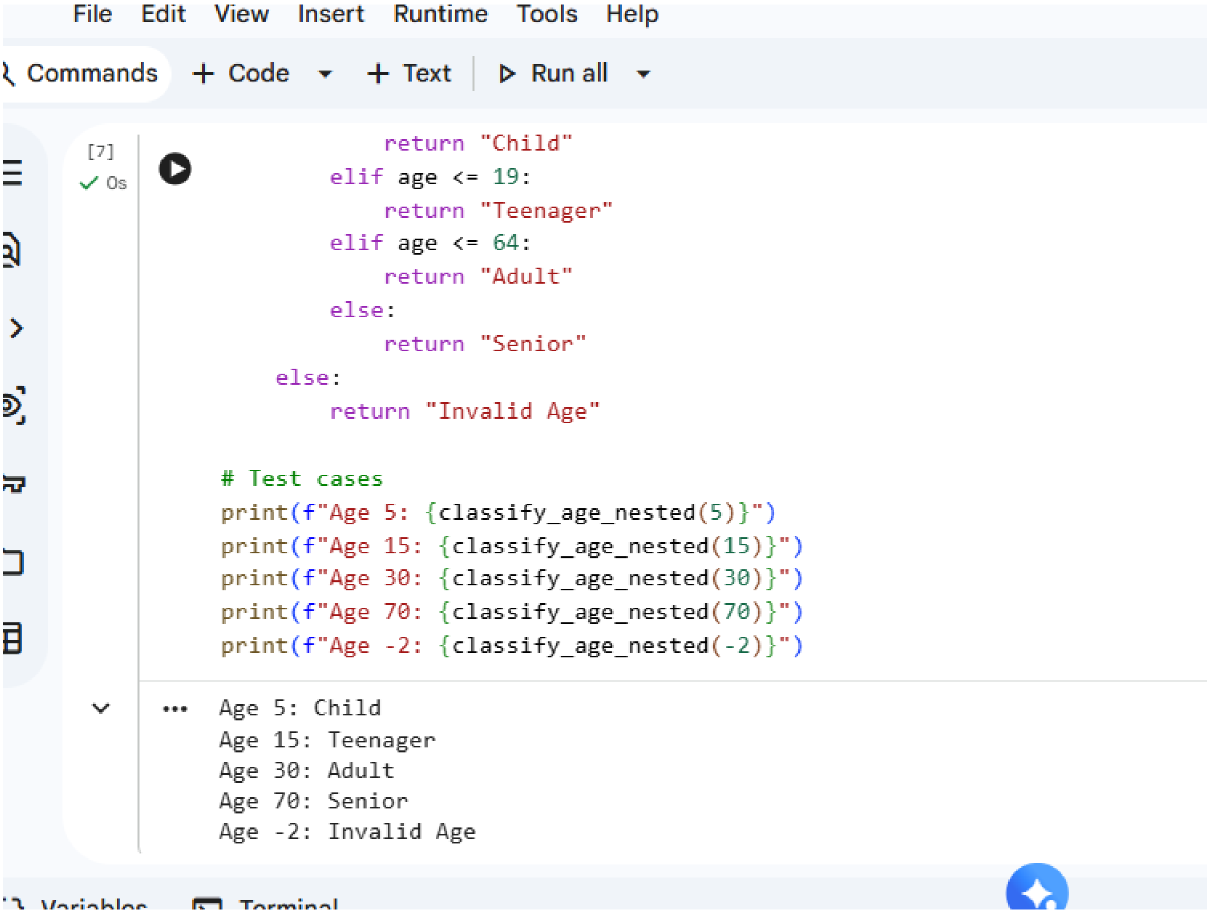
Explanation:

1. def classify_age_nested(age):: Defines a function that takes one argument, age.
2. if age >= 0:: The outermost condition checks if the age is non-negative. This acts as a primary guard against invalid negative ages.
3. if age <= 12:: If the age is non-negative, the first nested condition checks if it's 12 or younger, classifying it as a "Child".
4. elif age <= 19:: If not a child, this condition checks if the age is 19 or younger, classifying it as a "Teenager". Since the previous if already handled ages up to 12, this elif effectively covers ages 13 to 19.
5. elif age <= 64:: If not a teenager, this checks if the age is 64 or younger, classifying it as an "Adult". This effectively covers ages 20 to 64.
6. else: (inside the age >= 0 block): If none of the above conditions are met (meaning age is greater than 64), it's classified as a "Senior".

7. else: (matching if age >= 0): If the initial age >= 0 condition is false (i.e., age is negative), it returns "Invalid Age".

This structure uses nesting to sequentially narrow down the age range. While functional, excessive nesting can sometimes make code harder to read and maintain.

Implementaion:



```
            return "Child"
        elif age <= 19:
            return "Teenager"
        elif age <= 64:
            return "Adult"
        else:
            return "Senior"
    else:
        return "Invalid Age"

# Test cases
print(f"Age 5: {classify_age_nested(5)}")
print(f"Age 15: {classify_age_nested(15)}")
print(f"Age 30: {classify_age_nested(30)}")
print(f"Age 70: {classify_age_nested(70)}")
print(f"Age -2: {classify_age_nested(-2)}")
```

```
Age 5: Child
Age 15: Teenager
Age 30: Adult
Age 70: Senior
Age -2: Invalid Age
```

Task-04:

Prompt:

You need to calculate the sum of the first n natural numbers.

generate a sum_to_n() function using a for loop.

- Analyze the generated code.

- Ask the AI to suggest an alternative implementation using a while loop or a mathematical formula.

Code:

```
def sum_to_n_for(n):
    """Calculates the sum of the first n natural numbers using a for loop."""
    if n < 1:
        return "Input must be a positive integer"
    total_sum = 0
    for i in range(1, n + 1):
        total_sum += i
    return total_sum


# Test cases
print(f"Sum of first 5 natural numbers (for loop): {sum_to_n_for(5)}") # Expected: 15
print(f"Sum of first 10 natural numbers (for loop): {sum_to_n_for(10)}") # Expected: 55
print(f"Sum of first 1 natural number (for loop): {sum_to_n_for(1)}") # Expected: 1
print(f"Sum of first 0 natural numbers (for loop): {sum_to_n_for(0)}") # Expected: Input must be a positive integer
```

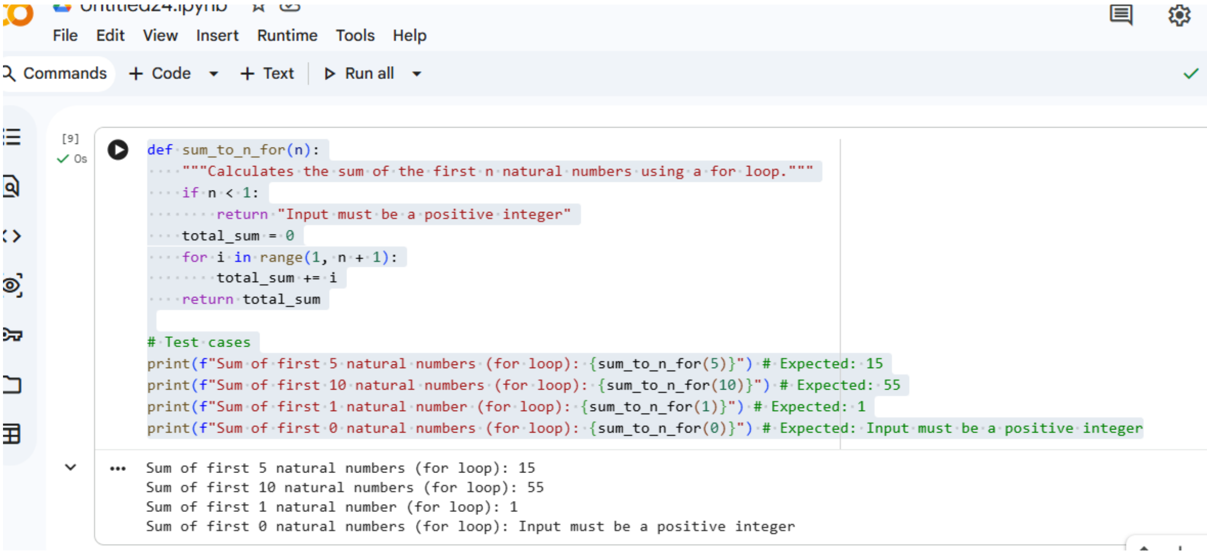
Explanation:

1. def sum_to_n_for(n):: Defines a function that takes one argument, n, representing the number of natural numbers to sum.
2. if n < 1:: A guard clause to handle invalid inputs (non-positive integers), as natural numbers typically start from 1.
3. total_sum = 0: Initializes a variable total_sum to 0. This variable will accumulate the sum of numbers.

4. for i in range(1, n + 1):: This is the core of the for loop. range(1, n + 1) generates a sequence of numbers from 1 up to and including n. In each iteration, i takes on the value of the next natural number.
5. total_sum += i: In each iteration, the current value of i is added to total_sum.
6. return total_sum: After the loop completes, the final accumulated total_sum is returned.

This for loop provides a clear and iterative way to sum numbers when the number of iterations is known (from 1 to n)

Implementation:



```python
def sum_to_n_for(n):
    """Calculates the sum of the first n natural numbers using a for loop."""
    if n < 1:
        return "Input must be a positive integer"
    total_sum = 0
    for i in range(1, n + 1):
        total_sum += i
    return total_sum

# Test cases
print(f"Sum of first 5 natural numbers (for loop): {sum_to_n_for(5)}") # Expected: 15
print(f"Sum of first 10 natural numbers (for loop): {sum_to_n_for(10)}") # Expected: 55
print(f"Sum of first 1 natural number (for loop): {sum_to_n_for(1)}") # Expected: 1
print(f"Sum of first 0 natural numbers (for loop): {sum_to_n_for(0)}") # Expected: Input must be a positive integer
```

```
Sum of first 5 natural numbers (for loop): 15
Sum of first 10 natural numbers (for loop): 55
Sum of first 1 natural number (for loop): 1
Sum of first 0 natural numbers (for loop): Input must be a positive integer
```

Task-05:

Prompt:

You are designing a basic banking application.

generate a Bank Account class with methods such as deposit(), withdraw(),

and check_balance().

- Analyze the AI-generated class structure and logic.
- Add meaningful comments and explain the working of the code.

Code:

```
class BankAccount:
    """A simple class to represent a bank account."""

    def __init__(self, account_holder_name, initial_balance=0):
        """
        Initializes a new BankAccount instance.

        Args:
            account_holder_name (str): The name of the account holder.
            initial_balance (float): The starting balance of the account. Defaults to 0.
        """
        self.account_holder = account_holder_name
        if initial_balance >= 0:
            self.balance = initial_balance
        else:
            print("Initial balance cannot be negative. Setting to 0.")
            self.balance = 0
        print(f"Account for {self.account_holder} created 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("Deposit amount must be positive.")

    def withdraw(self, amount):
        """
        Withdraws a specified amount from the account if sufficient funds are available.

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

    def check_balance(self):
        """
```

```python
        Returns the current balance of the account.

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

    def display_account_info(self):
        """
        Prints the account holder's name and current balance.
        """
        print(f"\n--- Account Information ---")
        print(f"Account Holder: {self.account_holder}")
        print(f"Current Balance: ${self.balance:.2f}")
        print(f"---------------------------")

# --- Demonstration ---
# Create a bank account
my_account = BankAccount("John Doe", 100.50)
my_account.display_account_info()

# Perform a deposit
my_account.deposit(200)
my_account.display_account_info()

# Perform a withdrawal
```

```
my_account.withdraw(75.25)
my_account.display_account_info()

# Attempt an overdraft withdrawal
my_account.withdraw(500)
my_account.display_account_info()

# Attempt a negative deposit
my_account.deposit(-50)

# Attempt a negative withdrawal
my_account.withdraw(-20)
```

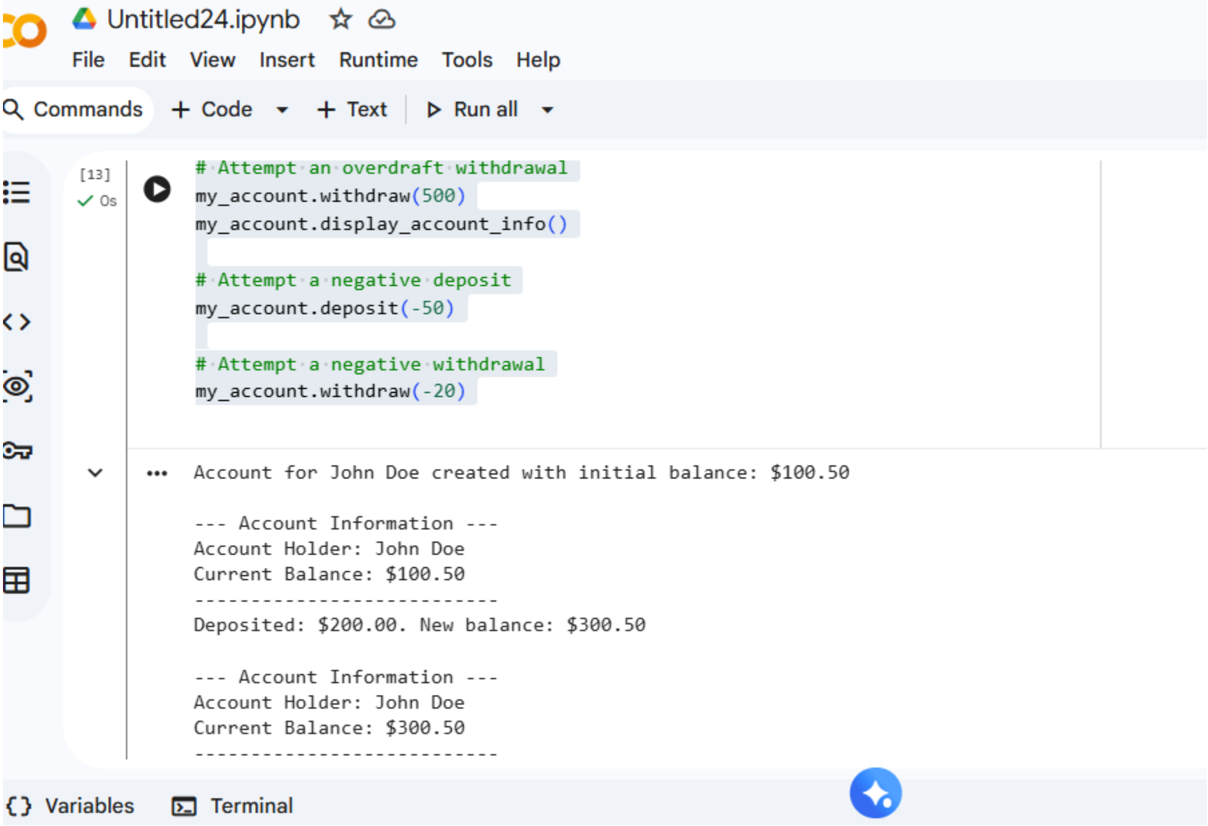
Explanation:

1. class BankAccount:: Defines a new class named BankAccount. This is the blueprint for creating individual bank account objects.
2. __init__(self, account_holder_name, initial_balance=0):: This is the constructor method. It's automatically called when a new BankAccount object is created.
    - It takes account_holder_name (string) and an optional initial_balance (float, defaults to 0) as arguments.
    - It initializes two instance attributes:
        - self.account_holder: Stores the name of the account owner.
        - self.balance: Stores the current money in the account. It includes a check to ensure the initial_balance is not negative.
3. deposit(self, amount):: This method handles adding money to the account.
    - It takes an amount (float) as an argument.

- It checks if the amount is positive. If so, it adds the amount to self.balance and prints a confirmation message.
- If the amount is not positive, it prints an error message.

4. withdraw(self, amount):: This method handles removing money from the account.
    - It takes an amount (float) as an argument.
    - It first checks if the amount is positive. If not, it prints an error.
    - Then, it checks if there are sufficient funds (i.e., if amount is less than or equal to self.balance).
    - If funds are sufficient, it subtracts the amount from self.balance and prints a confirmation.
    - If funds are insufficient, it prints an error message.
5. check_balance(self):: This simple method returns the current value of self.balance.
6. display_account_info(self):: A utility method to print a formatted summary of the account holder's name and current balance..

Implementation:



```
# Attempt an overdraft withdrawal
my_account.withdraw(500)
my_account.display_account_info()

# Attempt a negative deposit
my_account.deposit(-50)

# Attempt a negative withdrawal
my_account.withdraw(-20)
```

```
Account for John Doe created with initial balance: $100.50

--- Account Information ---
Account Holder: John Doe
Current Balance: $100.50
---------------------------
Deposited: $200.00. New balance: $300.50

--- Account Information ---
Account Holder: John Doe
Current Balance: $300.50
---------------------------
```