AI Assisted Coding Lab Ass-6.1

Name: G.SAI GANESH

Batch-13

2303A51848

Task Description #1 (AI-Based Code Completion for Loops) Task:

Use an AI code completion tool to generate a loop-based

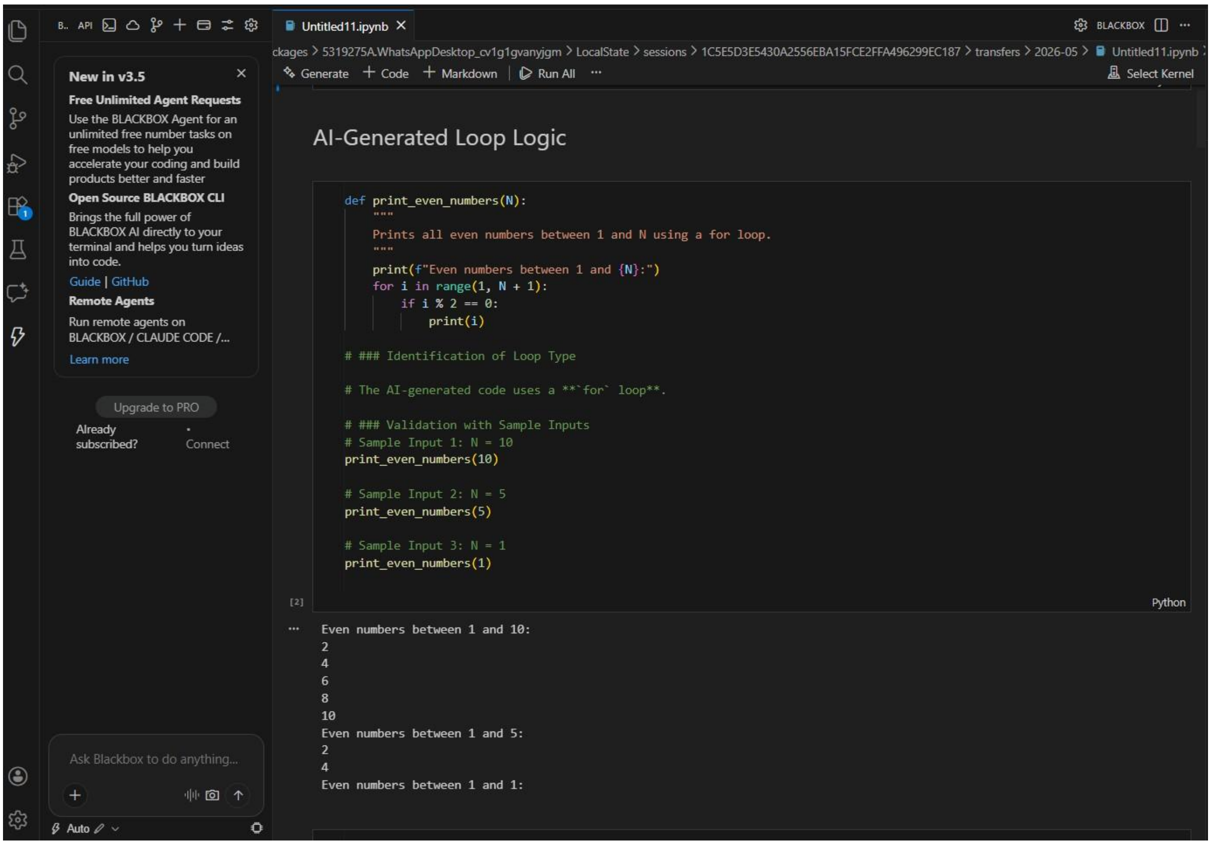
program.

Prompt:

“Generate Python code to print all even numbers between 1 and N using a loop.”

Expected Output:

- AI-generated loop logic.
- Identification of loop type used (for or while).
- Validation with sample inputs.


B.. API
Untitled11.ipynb
BLACKBOX
ckages > 5319275A.WhatsAppDesktop_cv1g1gvanyjgm > LocalState > sessions > 1C5E5D3E5430A2556EBA15FCE2FFA496299EC187 > transfers > 2026-05 > Untitled11.ipynb
Generate
+ Code
+ Markdown
Run All
Select Kernel
Upgrade to PRO
Already subscribed?
Connect
Ask Blackbox to do anything...
Auto
[2]
Python

New in v3.5

Free Unlimited Agent Requests

Use the BLACKBOX Agent for an unlimited free number tasks on free models to help you accelerate your coding and build products better and faster

Open Source BLACKBOX CLI

Brings the full power of BLACKBOX AI directly to your terminal and helps you turn ideas into code.

Guide | GitHub

Remote Agents

Run remote agents on BLACKBOX / CLAUDE CODE /...

Learn more

AI-Generated Loop Logic

```python
def print_even_numbers(N):
    """
    Prints all even numbers between 1 and N using a for loop.
    """
    print(f"Even numbers between 1 and {N}:")
    for i in range(1, N + 1):
        if i % 2 == 0:
            print(i)

# ### Identification of Loop Type

# The AI-generated code uses a **`for` loop**.

# ### Validation with Sample Inputs
# Sample Input 1: N = 10
print_even_numbers(10)

# Sample Input 2: N = 5
print_even_numbers(5)

# Sample Input 3: N = 1
print_even_numbers(1)
```

```
Even numbers between 1 and 10:
2
4
6
8
10
Even numbers between 1 and 5:
2
4
Even numbers between 1 and 1:
```

Task Description #2 (AI-Based Code Completion for Loop with

Conditionals)

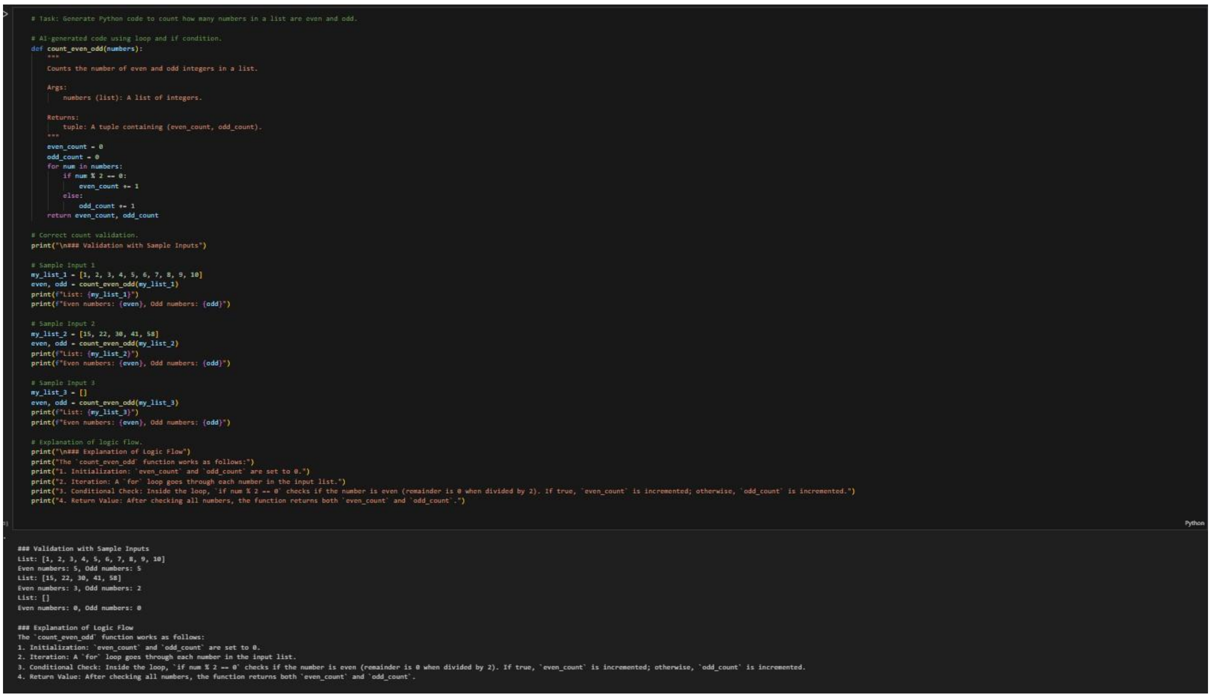
Task: Use an AI code completion tool to combine loops and conditionals.

Prompt:

“Generate Python code to count how many numbers in a list are even and odd.”

Expected Output:

- AI-generated code using loop and if condition.
- Correct count validation.
- Explanation of logic flow.



```python
# Task: Generate Python code to count how many numbers in a list are even and odd.

# AI-generated code using loop and if condition.
def count_even_odd(numbers):
    """
    Counts the number of even and odd integers in a list.

    Args:
        numbers (list): A list of integers.

    Returns:
        tuple: A tuple containing (even_count, odd_count).
    """
    even_count = 0
    odd_count = 0
    for num in numbers:
        if num % 2 == 0:
            even_count += 1
        else:
            odd_count += 1
    return even_count, odd_count

# Correct count validation.
print("\n### Validation with Sample Inputs")

# Sample Input 1
my_list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even, odd = count_even_odd(my_list_1)
print(f"List: {my_list_1}")
print(f"Even numbers: {even}, Odd numbers: {odd}")

# Sample Input 2
my_list_2 = [15, 22, 30, 41, 58]
even, odd = count_even_odd(my_list_2)
print(f"List: {my_list_2}")
print(f"Even numbers: {even}, Odd numbers: {odd}")

# Sample Input 3
my_list_3 = []
even, odd = count_even_odd(my_list_3)
print(f"List: {my_list_3}")
print(f"Even numbers: {even}, Odd numbers: {odd}")

# Explanation of logic flow.
print("\n### Explanation of Logic Flow")
print("The `count_even_odd` function works as follows:")
print("1. Initialization: `even_count` and `odd_count` are set to 0.")
print("2. Iteration: A `for` loop goes through each number in the input list.")
print("3. Conditional Check: Inside the loop, `if num % 2 == 0` checks if the number is even (remainder is 0 when divided by 2). If true, `even_count` is incremented; otherwise, `odd_count` is incremented.")
print("4. Return Value: After checking all numbers, the function returns both `even_count` and `odd_count`.")
```

```
### Validation with Sample Inputs
List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Even numbers: 5, Odd numbers: 5
List: [15, 22, 30, 41, 58]
Even numbers: 3, Odd numbers: 2
List: []
Even numbers: 0, Odd numbers: 0

### Explanation of Logic Flow
The `count_even_odd` function works as follows:
1. Initialization: `even_count` and `odd_count` are set to 0.
2. Iteration: A `for` loop goes through each number in the input list.
3. Conditional Check: Inside the loop, `if num % 2 == 0` checks if the number is even (remainder is 0 when divided by 2). If true, `even_count` is incremented; otherwise, `odd_count` is incremented.
4. Return Value: After checking all numbers, the function returns both `even_count` and `odd_count`.
```

Task Description #3 (AI-Based Code Completion for Class Attributes Validation)

Task: Use an AI tool to complete a Python class that validates user input.

Prompt:

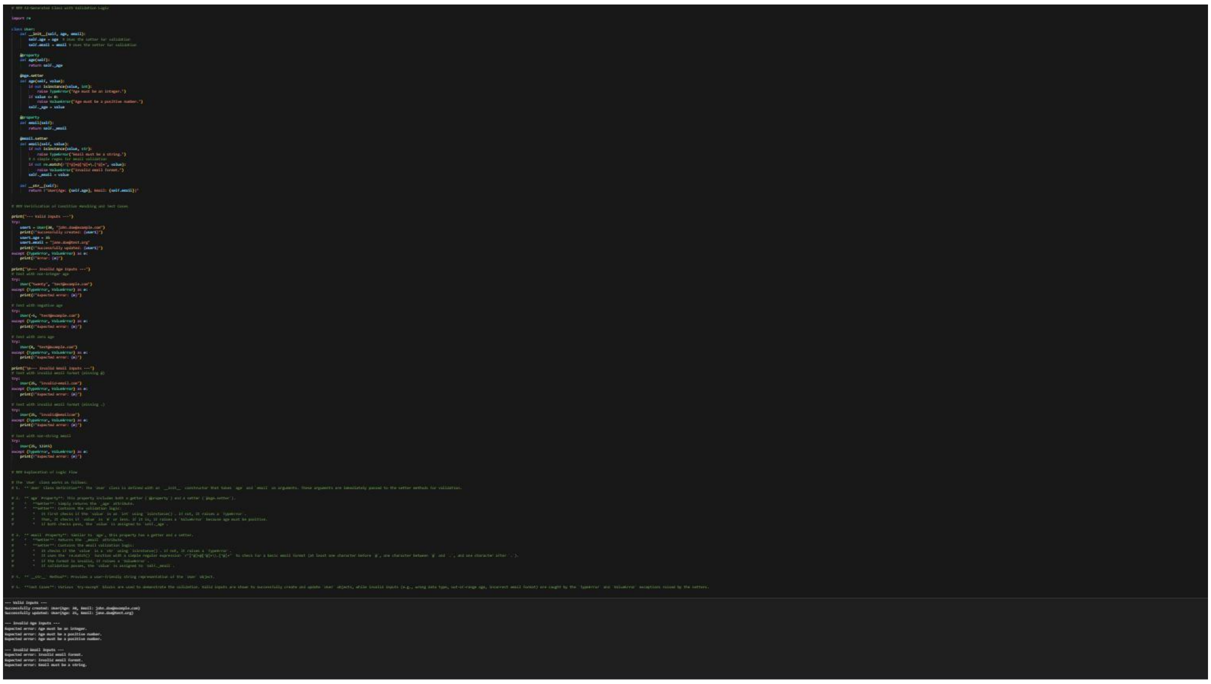
“Generate a Python class User that validates age and email using conditional statements.”

Expected Output:

- AI-generated class with validation logic.

- Verification of condition handling.

- Test cases for valid and invalid inputs.



```python
# ### AI-Generated Class with Validation Logic

import re

class User:
    def __init__(self, age, email):
        self.age = age  # Uses the setter for validation
        self.email = email # Uses the setter for validation

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if not isinstance(value, int):
            raise TypeError("Age must be an integer.")
        if value <= 0:
            raise ValueError("Age must be a positive number.")
        self._age = value

    @property
    def email(self):
        return self._email

    @email.setter
    def email(self, value):
        if not isinstance(value, str):
            raise TypeError("Email must be a string.")
        # A simple regex for email validation
        if not re.match(r"[^@]+@[^@]+\.[^@]+", value):
            raise ValueError("Invalid email format.")
        self._email = value

    def __str__(self):
        return f"User(Age: {self.age}, Email: {self.email})"


# ### Verification of Condition Handling and Test Cases

print("--- Valid Inputs ---")
try:
    user1 = User(30, "john.doe@example.com")
    print(f"Successfully created: {user1}")
    user1.age = 35
    user1.email = "jane.doe@test.org"
    print(f"Successfully updated: {user1}")
except (TypeError, ValueError) as e:
    print(f"Error: {e}")

print("\n--- Invalid Age Inputs ---")
# Test with non-integer age
try:
    User("twenty", "test@example.com")
except (TypeError, ValueError) as e:
    print(f"Expected error: {e}")

# Test with negative age
try:
    User(-5, "test@example.com")
except (TypeError, ValueError) as e:
    print(f"Expected error: {e}")

# Test with zero age
try:
    User(0, "test@example.com")
except (TypeError, ValueError) as e:
    print(f"Expected error: {e}")

print("\n--- Invalid Email Inputs ---")
# Test with invalid email format (missing @)
try:
    User(25, "invalid-email.com")
except (TypeError, ValueError) as e:
    print(f"Expected error: {e}")

# Test with invalid email format (missing .)
try:
    User(25, "invalid@emailcom")
except (TypeError, ValueError) as e:
    print(f"Expected error: {e}")

# Test with non-string email
try:
    User(25, 12345)
except (TypeError, ValueError) as e:
    print(f"Expected error: {e}")


# ### Explanation of Logic Flow

# The `User` class works as follows:
# 1.  **`User` Class Definition**: The `User` class is defined with an `__init__` constructor that takes `age` and `email` as arguments. These arguments are immediately passed to the setter methods for validation.

# 2.  **`age` Property**: This property includes both a getter (`@property`) and a setter (`@age.setter`).
#     *   **Getter**: Simply returns the `_age` attribute.
#     *   **Setter**: Contains the validation logic:
#         *   It first checks if the `value` is an `int` using `isinstance()`. If not, it raises a `TypeError`.
#         *   Then, it checks if `value` is `0` or less. If it is, it raises a `ValueError` because age must be positive.
#         *   If both checks pass, the `value` is assigned to `self._age`.

# 3.  **`email` Property**: Similar to `age`, this property has a getter and a setter.
#     *   **Getter**: Returns the `_email` attribute.
#     *   **Setter**: Contains the email validation logic:
#         *   It checks if the `value` is a `str` using `isinstance()`. If not, it raises a `TypeError`.
#         *   It uses the `re.match()` function with a simple regular expression `r"[^@]+@[^@]+\\.[^@]+"` to check for a basic email format (at least one character before `@`, one character between `@` and `.`, and one character after `.`).
#         *   If the format is invalid, it raises a `ValueError`.
#         *   If validation passes, the `value` is assigned to `self._email`.

# 4.  **`__str__` Method**: Provides a user-friendly string representation of the `User` object.

# 5.  **Test Cases**: Various `try-except` blocks are used to demonstrate the validation. Valid inputs are shown to successfully create and update `User` objects, while invalid inputs (e.g., wrong data type, out-of-range age, incorrect email format) are caught by the `TypeError` and `ValueError` exceptions raised by the setters.
```

```
--- Valid Inputs ---
Successfully created: User(Age: 30, Email: john.doe@example.com)
Successfully updated: User(Age: 35, Email: jane.doe@test.org)

--- Invalid Age Inputs ---
Expected error: Age must be an integer.
Expected error: Age must be a positive number.
Expected error: Age must be a positive number.

--- Invalid Email Inputs ---
Expected error: Invalid email format.
Expected error: Invalid email format.
Expected error: Email must be a string.
```

Task Description #4 (AI-Based Code Completion for Classes) Task:

Use an AI code completion tool to generate a Python class for

managing student details.

Prompt:

“Generate a Python class Student with attributes (name, roll number,

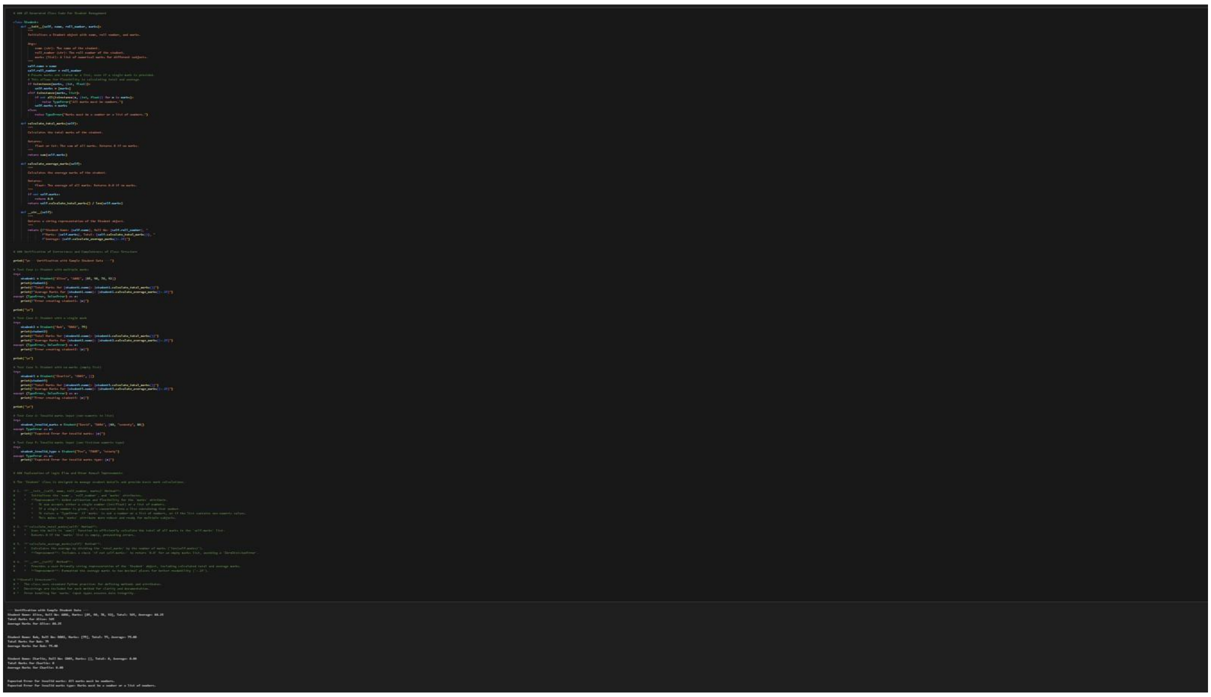
marks) and methods to calculate total and average marks.”

Expected Output:

- AI-generated class code.

- Verification of correctness and completeness of class structure.

- Minor manual improvements (if needed) with justification.



```
# ### AI-Generated Class Code for Student Management

class Student:
    def __init__(self, name, roll_number, marks):
        """
        Initializes a Student object with name, roll number, and marks.

        Args:
            name (str): The name of the student.
            roll_number (str): The roll number of the student.
            marks (list): A list of numerical marks for different subjects.
        """
        self.name = name
        self.roll_number = roll_number
        # Ensure marks are stored as a list, even if a single mark is provided.
        # This allows for flexibility in calculating total and average.
        if isinstance(marks, (int, float)):
            self.marks = [marks]
        elif isinstance(marks, list):
            if not all(isinstance(m, (int, float)) for m in marks):
                raise TypeError("All marks must be numbers.")
            self.marks = marks
        else:
            raise TypeError("Marks must be a number or a list of numbers.")

    def calculate_total_marks(self):
        """
        Calculates the total marks of the student.

        Returns:
            float or int: The sum of all marks. Returns 0 if no marks.
        """
        return sum(self.marks)

    def calculate_average_marks(self):
        """
        Calculates the average marks of the student.

        Returns:
            float: The average of all marks. Returns 0.0 if no marks.
        """
        if not self.marks:
            return 0.0
        return self.calculate_total_marks() / len(self.marks)

    def __str__(self):
        """
        Returns a string representation of the Student object.
        """
        return (f"Student Name: {self.name}, Roll No: {self.roll_number}, "
                f"Marks: {self.marks}, Total: {self.calculate_total_marks()}, "
                f"Average: {self.calculate_average_marks():.2f}")


# ### Verification of Correctness and Completeness of Class Structure

print("\n--- Verification with Sample Student Data ---")

# Test Case 1: Student with multiple marks
try:
    student1 = Student("Alice", "S001", [85, 90, 78, 92])
    print(student1)
    print(f"Total Marks for {student1.name}: {student1.calculate_total_marks()}")
    print(f"Average Marks for {student1.name}: {student1.calculate_average_marks():.2f}")
except (TypeError, ValueError) as e:
    print(f"Error creating student1: {e}")

print("\n")

# Test Case 2: Student with a single mark
try:
    student2 = Student("Bob", "S002", 75)
    print(student2)
    print(f"Total Marks for {student2.name}: {student2.calculate_total_marks()}")
    print(f"Average Marks for {student2.name}: {student2.calculate_average_marks():.2f}")
except (TypeError, ValueError) as e:
    print(f"Error creating student2: {e}")

print("\n")

# Test Case 3: Student with no marks (empty list)
try:
    student3 = Student("Charlie", "S003", [])
    print(student3)
    print(f"Total Marks for {student3.name}: {student3.calculate_total_marks()}")
    print(f"Average Marks for {student3.name}: {student3.calculate_average_marks():.2f}")
except (TypeError, ValueError) as e:
    print(f"Error creating student3: {e}")

print("\n")

# Test Case 4: Invalid marks input (non-numeric in list)
try:
    student_invalid_marks = Student("David", "S004", [80, "seventy", 90])
except TypeError as e:
    print(f"Expected Error for invalid marks: {e}")

# Test Case 5: Invalid marks input (non-list/non-numeric type)
try:
    student_invalid_type = Student("Eve", "S005", "ninety")
except TypeError as e:
    print(f"Expected Error for invalid marks type: {e}")


# ### Explanation of Logic Flow and Minor Manual Improvements

# The `Student` class is designed to manage student details and provide basic mark calculations.

# 1.  **`__init__(self, name, roll_number, marks)` Method**:
#     *   Initializes the `name`, `roll_number`, and `marks` attributes.
#     *   **Improvement**: Added validation and flexibility for the `marks` attribute.
#         *   It now accepts either a single number (int/float) or a list of numbers.
#         *   If a single number is given, it's converted into a list containing that number.
#         *   It raises a `TypeError` if `marks` is not a number or a list of numbers, or if the list contains non-numeric values.
#         *   This makes the `marks` attribute more robust and ready for multiple subjects.

# 2.  **`calculate_total_marks(self)` Method**:
#     *   Uses the built-in `sum()` function to efficiently calculate the total of all marks in the `self.marks` list.
#     *   Returns 0 if the `marks` list is empty, preventing errors.

# 3.  **`calculate_average_marks(self)` Method**:
#     *   Calculates the average by dividing the `total_marks` by the number of marks (`len(self.marks)`).
#     *   **Improvement**: Includes a check `if not self.marks:` to return `0.0` for an empty marks list, avoiding a `ZeroDivisionError`.

# 4.  **`__str__(self)` Method**:
#     *   Provides a user-friendly string representation of the `Student` object, including calculated total and average marks.
#     *   **Improvement**: Formatted the average marks to two decimal places for better readability (`:.2f`).

# **Overall Structure**:
# *   The class uses standard Python practices for defining methods and attributes.
# *   Docstrings are included for each method for clarity and documentation.
# *   Error handling for `marks` input types ensures data integrity.
```

```
--- Verification with Sample Student Data ---
Student Name: Alice, Roll No: S001, Marks: [85, 90, 78, 92], Total: 345, Average: 86.25
Total Marks for Alice: 345
Average Marks for Alice: 86.25


Student Name: Bob, Roll No: S002, Marks: [75], Total: 75, Average: 75.00
Total Marks for Bob: 75
Average Marks for Bob: 75.00


Student Name: Charlie, Roll No: S003, Marks: [], Total: 0, Average: 0.00
Total Marks for Charlie: 0
Average Marks for Charlie: 0.00


Expected Error for invalid marks: All marks must be numbers.
Expected Error for invalid marks type: Marks must be a number or a list of numbers.
```

Task Description 5 (AI-Assisted Code Completion Review) Task:

Use an AI tool to generate a complete Python program using

classes, loops, and conditionals together.

Prompt:

“Generate a Python program for a simple bank account system using

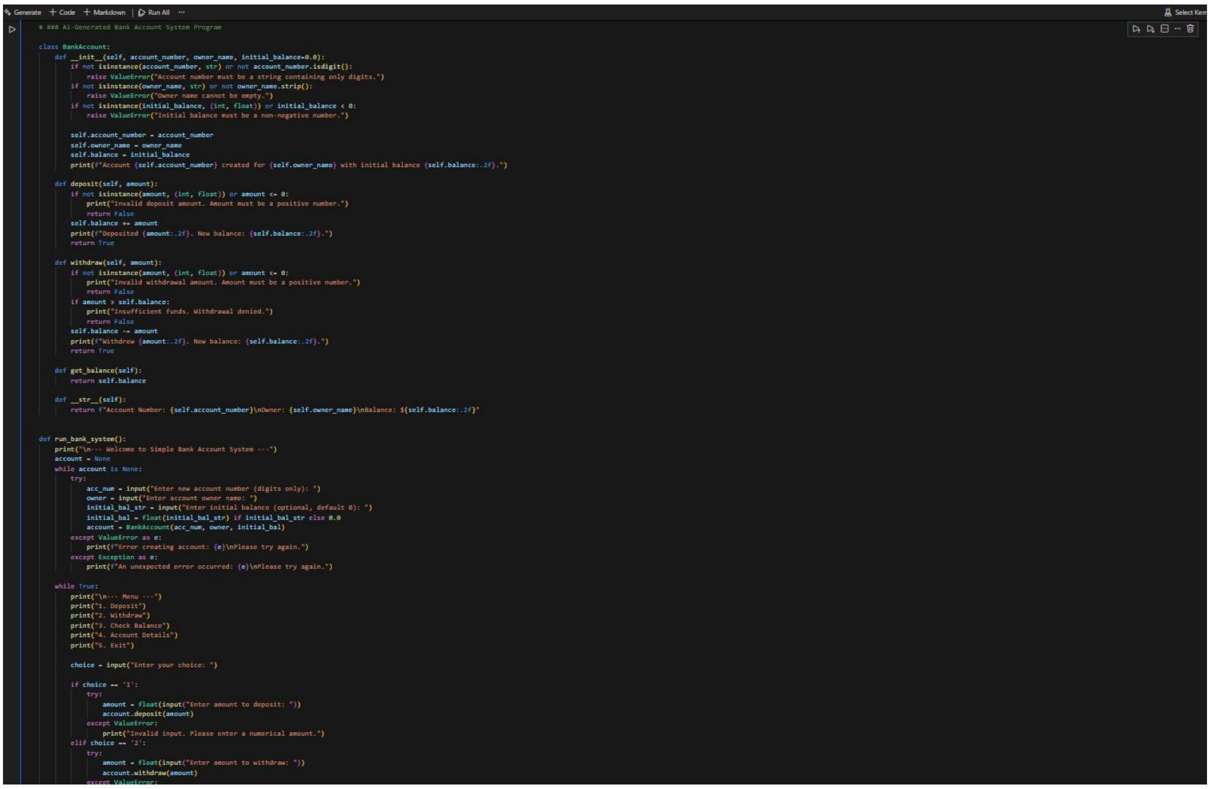
class, loops, and conditional statements.”

Expected Output:

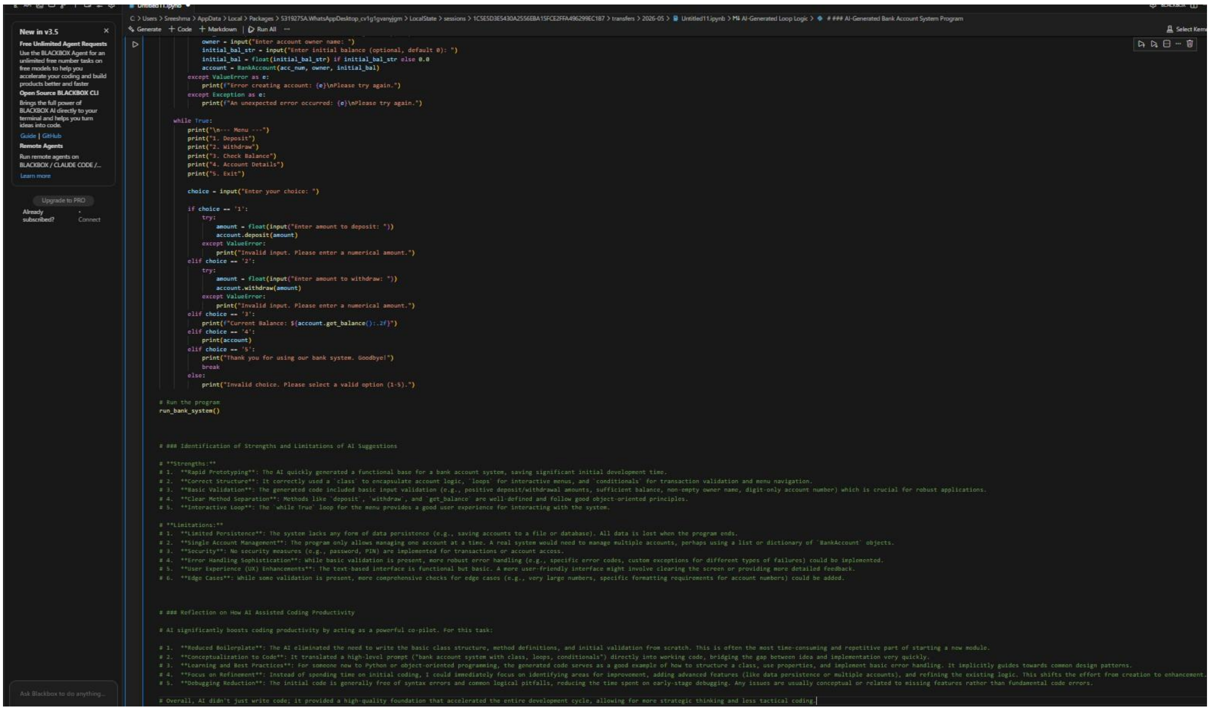
• Complete AI-generated program.

• Identification of strengths and limitations of AI suggestions.

• Reflection on how AI assisted coding productivity.



```python
# ### AI-Generated Bank Account System Program

class BankAccount:
    def __init__(self, account_number, owner_name, initial_balance=0.0):
        if not isinstance(account_number, str) or not account_number.isdigit():
            raise ValueError("Account number must be a string containing only digits.")
        if not isinstance(owner_name, str) or not owner_name.strip():
            raise ValueError("Owner name cannot be empty.")
        if not isinstance(initial_balance, (int, float)) or initial_balance < 0:
            raise ValueError("Initial balance must be a non-negative number.")

        self.account_number = account_number
        self.owner_name = owner_name
        self.balance = initial_balance
        print(f"Account {self.account_number} created for {self.owner_name} with initial balance {self.balance:.2f}.")

    def deposit(self, amount):
        if not isinstance(amount, (int, float)) or amount <= 0:
            print("Invalid deposit amount. Amount must be a positive number.")
            return False
        self.balance += amount
        print(f"Deposited {amount:.2f}. New balance: {self.balance:.2f}.")
        return True

    def withdraw(self, amount):
        if not isinstance(amount, (int, float)) or amount <= 0:
            print("Invalid withdrawal amount. Amount must be a positive number.")
            return False
        if amount > self.balance:
            print("Insufficient funds. Withdrawal denied.")
            return False
        self.balance -= amount
        print(f"Withdrew {amount:.2f}. New balance: {self.balance:.2f}.")
        return True

    def get_balance(self):
        return self.balance

    def __str__(self):
        return f"Account Number: {self.account_number}\nOwner: {self.owner_name}\nBalance: ${self.balance:.2f}"


def run_bank_system():
    print("\n--- Welcome to Simple Bank Account System ---")
    account = None
    while account is None:
        try:
            acc_num = input("Enter new account number (digits only): ")
            owner = input("Enter account owner name: ")
            initial_bal_str = input("Enter initial balance (optional, default 0): ")
            initial_bal = float(initial_bal_str) if initial_bal_str else 0.0
            account = BankAccount(acc_num, owner, initial_bal)
        except ValueError as e:
            print(f"Error creating account: {e}\nPlease try again.")
        except Exception as e:
            print(f"An unexpected error occurred: {e}\nPlease try again.")

    while True:
        print("\n--- Menu ---")
        print("1. Deposit")
        print("2. Withdraw")
        print("3. Check Balance")
        print("4. Account Details")
        print("5. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':
            try:
                amount = float(input("Enter amount to deposit: "))
                account.deposit(amount)
            except ValueError:
                print("Invalid input. Please enter a numerical amount.")
        elif choice == '2':
            try:
                amount = float(input("Enter amount to withdraw: "))
                account.withdraw(amount)
            except ValueError:
```


Untitled11.ipynb
C: > Users > Sreeshma > AppData > Local > Packages > 5319275A.WhatsAppDesktop_cv1g1gvanyjgm > LocalState > sessions > 1C5E5D3E5430A2556EBA15FCE2FFA496299EC187 > transfers > 2026-05 > Untitled11.ipynb > M↓ AI-Generated Loop Logic > # ### AI-Generated Bank Account System Program
Generate
+ Code
+ Markdown
Run All
Select Kern

New in v3.5

Free Unlimited Agent Requests

Use the BLACKBOX Agent for an unlimited free number tasks on free models to help you accelerate your coding and build products better and faster

Open Source BLACKBOX CLI

Brings the full power of BLACKBOX AI directly to your terminal and helps you turn ideas into code.

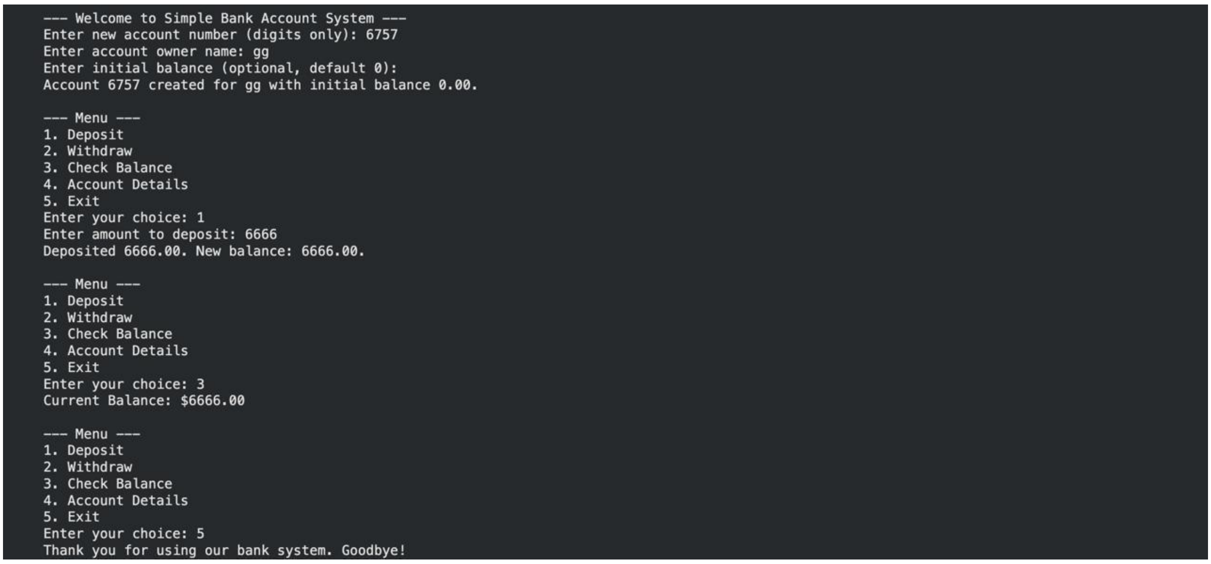
Guide | GitHub

Remote Agents

Run remote agents on BLACKBOX / CLAUDE CODE /...

Learn more

```python
            owner = input("Enter account owner name: ")
            initial_bal_str = input("Enter initial balance (optional, default 0): ")
            initial_bal = float(initial_bal_str) if initial_bal_str else 0.0
            account = BankAccount(acc_num, owner, initial_bal)
        except ValueError as e:
            print(f"Error creating account: {e}\nPlease try again.")
        except Exception as e:
            print(f"An unexpected error occurred: {e}\nPlease try again.")

    while True:
        print("\n--- Menu ---")
        print("1. Deposit")
        print("2. Withdraw")
        print("3. Check Balance")
        print("4. Account Details")
        print("5. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':
            try:
                amount = float(input("Enter amount to deposit: "))
                account.deposit(amount)
            except ValueError:
                print("Invalid input. Please enter a numerical amount.")
        elif choice == '2':
            try:
                amount = float(input("Enter amount to withdraw: "))
                account.withdraw(amount)
            except ValueError:
                print("Invalid input. Please enter a numerical amount.")
        elif choice == '3':
            print(f"Current Balance: ${account.get_balance():.2f}")
        elif choice == '4':
            print(account)
        elif choice == '5':
            print("Thank you for using our bank system. Goodbye!")
            break
        else:
            print("Invalid choice. Please select a valid option (1-5).")

# Run the program
run_bank_system()



# ### Identification of Strengths and Limitations of AI Suggestions

# **Strengths:**
# 1.  **Rapid Prototyping**: The AI quickly generated a functional base for a bank account system, saving significant initial development time.
# 2.  **Correct Structure**: It correctly used a `class` to encapsulate account logic, `loops` for interactive menus, and `conditionals` for transaction validation and menu navigation.
# 3.  **Basic Validation**: The generated code included basic input validation (e.g., positive deposit/withdrawal amounts, sufficient balance, non-empty owner name, digit-only account number) which is crucial for robust applications.
# 4.  **Clear Method Separation**: Methods like `deposit`, `withdraw`, and `get_balance` are well-defined and follow good object-oriented principles.
# 5.  **Interactive Loop**: The `while True` loop for the menu provides a good user experience for interacting with the system.

# **Limitations:**
# 1.  **Limited Persistence**: The system lacks any form of data persistence (e.g., saving accounts to a file or database). All data is lost when the program ends.
# 2.  **Single Account Management**: The program only allows managing one account at a time. A real system would need to manage multiple accounts, perhaps using a list or dictionary of `BankAccount` objects.
# 3.  **Security**: No security measures (e.g., password, PIN) are implemented for transactions or account access.
# 4.  **Error Handling Sophistication**: While basic validation is present, more robust error handling (e.g., specific error codes, custom exceptions for different types of failures) could be implemented.
# 5.  **User Experience (UX) Enhancements**: The text-based interface is functional but basic. A more user-friendly interface might involve clearing the screen or providing more detailed feedback.
# 6.  **Edge Cases**: While some validation is present, more comprehensive checks for edge cases (e.g., very large numbers, specific formatting requirements for account numbers) could be added.



# ### Reflection on How AI Assisted Coding Productivity

# AI significantly boosts coding productivity by acting as a powerful co-pilot. For this task:

# 1.  **Reduced Boilerplate**: The AI eliminated the need to write the basic class structure, method definitions, and initial validation from scratch. This is often the most time-consuming and repetitive part of starting a new module.
# 2.  **Conceptualization to Code**: It translated a high-level prompt ("bank account system with class, loops, conditionals") directly into working code, bridging the gap between idea and implementation very quickly.
# 3.  **Learning and Best Practices**: For someone new to Python or object-oriented programming, the generated code serves as a good example of how to structure a class, use properties, and implement basic error handling. It implicitly guides towards common design patterns.
# 4.  **Focus on Refinement**: Instead of spending time on initial coding, I could immediately focus on identifying areas for improvement, adding advanced features (like data persistence or multiple accounts), and refining the existing logic. This shifts the effort from creation to enhancement.
# 5.  **Debugging Reduction**: The initial code is generally free of syntax errors and common logical pitfalls, reducing the time spent on early-stage debugging. Any issues are usually conceptual or related to missing features rather than fundamental code errors.

# Overall, AI didn't just write code; it provided a high-quality foundation that accelerated the entire development cycle, allowing for more strategic thinking and less tactical coding.
```



```
--- Welcome to Simple Bank Account System ---
Enter new account number (digits only): 6757
Enter account owner name: gg
Enter initial balance (optional, default 0):
Account 6757 created for gg with initial balance 0.00.

--- Menu ---
1. Deposit
2. Withdraw
3. Check Balance
4. Account Details
5. Exit
Enter your choice: 1
Enter amount to deposit: 6666
Deposited 6666.00. New balance: 6666.00.

--- Menu ---
1. Deposit
2. Withdraw
3. Check Balance
4. Account Details
5. Exit
Enter your choice: 3
Current Balance: $6666.00

--- Menu ---
1. Deposit
2. Withdraw
3. Check Balance
4. Account Details
5. Exit
Enter your choice: 5
Thank you for using our bank system. Goodbye!
```