ASSIGNMENT - 8.5

AI Assisted Coding

Name : G. Likhitha Rao

Roll No: 2303A52487

Batch:35

Task Description #1 (Username Validator – Apply AI in Authentication Context)

• Task: Use AI to generate at least 3 assert test cases for a function is_valid_username(username) and then implement the function using Test-Driven Development principles.

• Requirements:

o Username length must be between 5 and 15 characters.

o Must contain only alphabets and digits.

o Must not start with a digit.

o No spaces allowed.

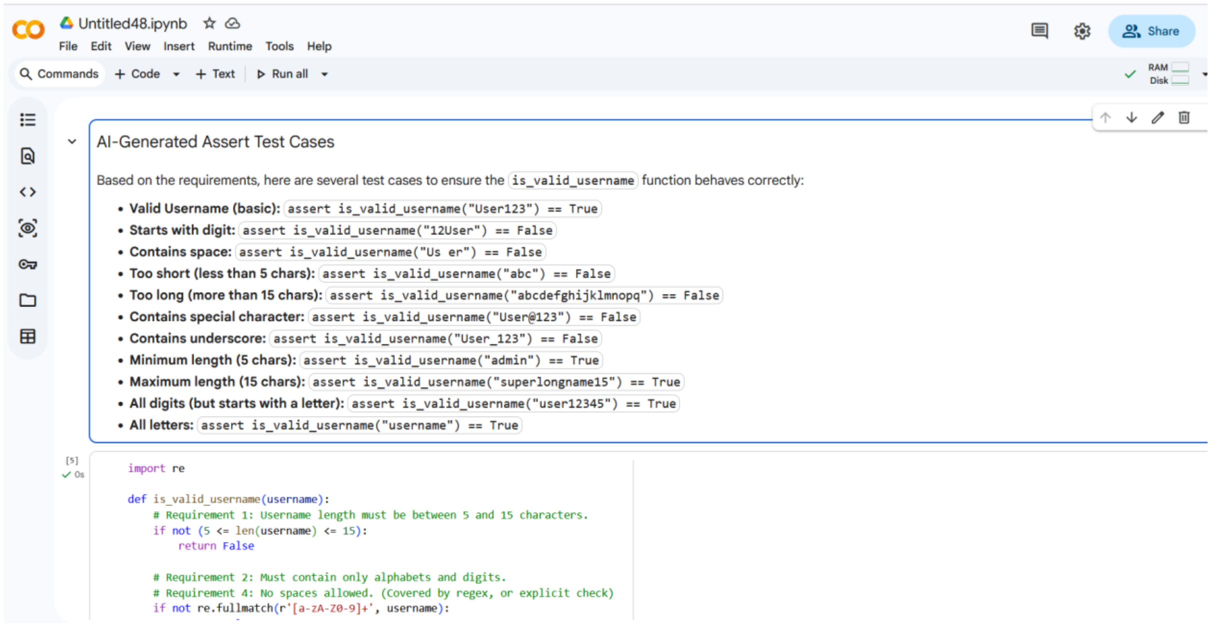
Example Assert Test Cases:

```
assert is_valid_username("User123") == True
assert is_valid_username("12User") == False
assert is_valid_username("Us er") == False
```

Expected Output #1:

• Username validation logic successfully passing all AI-generated test cases.

CODE:

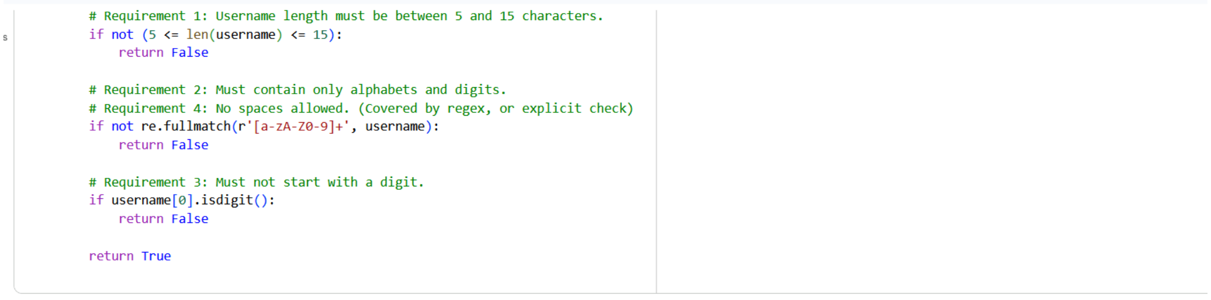

Untitled48.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
RAM
Disk

AI-Generated Assert Test Cases

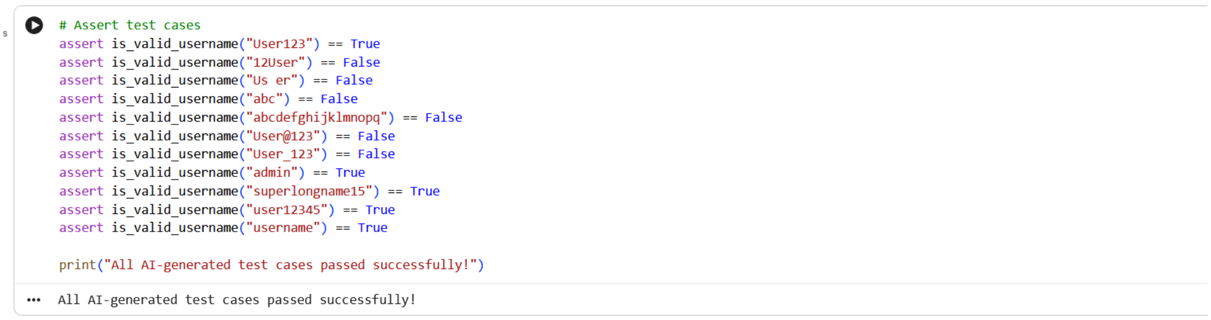
Based on the requirements, here are several test cases to ensure the `is_valid_username` function behaves correctly:

- **Valid Username (basic):** `assert is_valid_username("User123") == True`
- **Starts with digit:** `assert is_valid_username("12User") == False`
- **Contains space:** `assert is_valid_username("Us er") == False`
- **Too short (less than 5 chars):** `assert is_valid_username("abc") == False`
- **Too long (more than 15 chars):** `assert is_valid_username("abcdefghijklmnopq") == False`
- **Contains special character:** `assert is_valid_username("User@123") == False`
- **Contains underscore:** `assert is_valid_username("User_123") == False`
- **Minimum length (5 chars):** `assert is_valid_username("admin") == True`
- **Maximum length (15 chars):** `assert is_valid_username("superlongname15") == True`
- **All digits (but starts with a letter):** `assert is_valid_username("user12345") == True`
- **All letters:** `assert is_valid_username("username") == True`

```python
import re

def is_valid_username(username):
    # Requirement 1: Username length must be between 5 and 15 characters.
    if not (5 <= len(username) <= 15):
        return False

    # Requirement 2: Must contain only alphabets and digits.
    # Requirement 4: No spaces allowed. (Covered by regex, or explicit check)
    if not re.fullmatch(r'[a-zA-Z0-9]+', username):
```



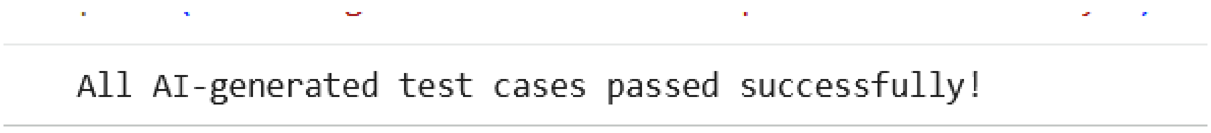
```python
    # Requirement 1: Username length must be between 5 and 15 characters.
    if not (5 <= len(username) <= 15):
        return False

    # Requirement 2: Must contain only alphabets and digits.
    # Requirement 4: No spaces allowed. (Covered by regex, or explicit check)
    if not re.fullmatch(r'[a-zA-Z0-9]+', username):
        return False

    # Requirement 3: Must not start with a digit.
    if username[0].isdigit():
        return False

    return True
```



```python
# Assert test cases
assert is_valid_username("User123") == True
assert is_valid_username("12User") == False
assert is_valid_username("Us er") == False
assert is_valid_username("abc") == False
assert is_valid_username("abcdefghijklmnopq") == False
assert is_valid_username("User@123") == False
assert is_valid_username("User_123") == False
assert is_valid_username("admin") == True
assert is_valid_username("superlongname15") == True
assert is_valid_username("user12345") == True
assert is_valid_username("username") == True

print("All AI-generated test cases passed successfully!")
```

```
All AI-generated test cases passed successfully!
```

OUTPUT:



```
All AI-generated test cases passed successfully!
```

Task Description #2 (Even–Odd & Type Classification – Apply AI for Robust Input Handling)

• Task: Use AI to generate at least 3 assert test cases for a function classify_value(x) and implement it using conditional logic and loops.

• Requirements:

o If input is an integer, classify as "Even" or "Odd".

o If input is 0, return "Zero".

o If input is non-numeric, return "Invalid Input".

Example Assert Test Cases:

assert classify_value(8) == "Even"

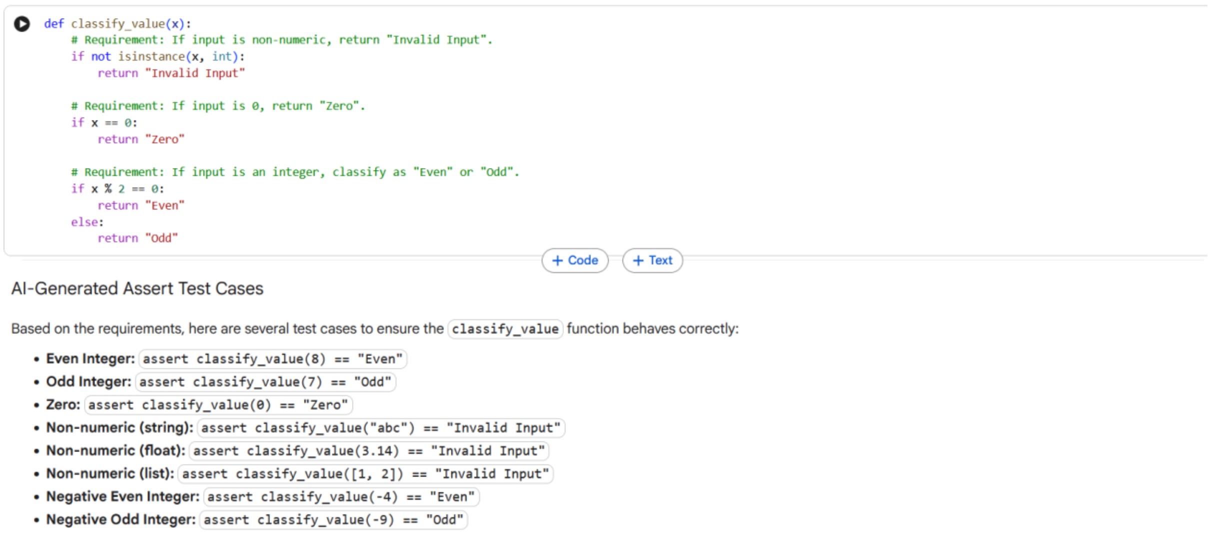
assert classify_value(7) == "Odd"

assert classify_value("abc") == "Invalid Input"

Expected Output #2:

• Function correctly classifying values and passing all test cases.

CODE:

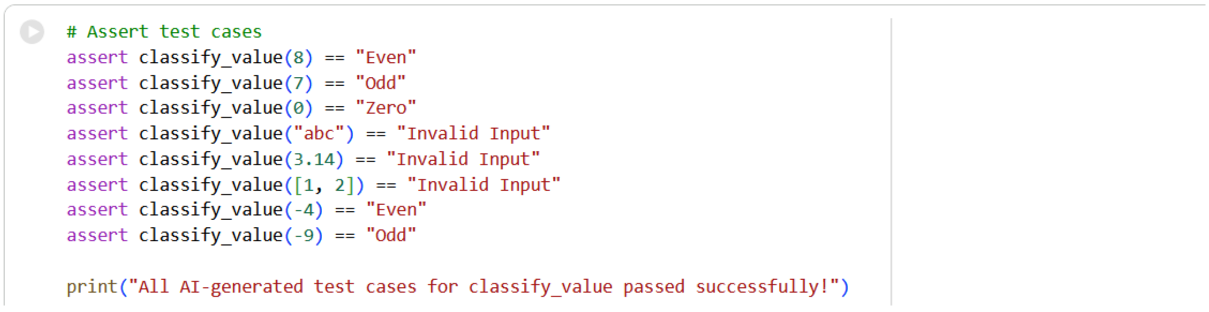


```python
def classify_value(x):
    # Requirement: If input is non-numeric, return "Invalid Input".
    if not isinstance(x, int):
        return "Invalid Input"

    # Requirement: If input is 0, return "Zero".
    if x == 0:
        return "Zero"

    # Requirement: If input is an integer, classify as "Even" or "Odd".
    if x % 2 == 0:
        return "Even"
    else:
        return "Odd"
```

AI-Generated Assert Test Cases

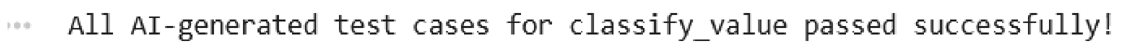
Based on the requirements, here are several test cases to ensure the `classify_value` function behaves correctly:

- **Even Integer:** `assert classify_value(8) == "Even"`
- **Odd Integer:** `assert classify_value(7) == "Odd"`
- **Zero:** `assert classify_value(0) == "Zero"`
- **Non-numeric (string):** `assert classify_value("abc") == "Invalid Input"`
- **Non-numeric (float):** `assert classify_value(3.14) == "Invalid Input"`
- **Non-numeric (list):** `assert classify_value([1, 2]) == "Invalid Input"`
- **Negative Even Integer:** `assert classify_value(-4) == "Even"`
- **Negative Odd Integer:** `assert classify_value(-9) == "Odd"`



```
# Assert test cases
assert classify_value(8) == "Even"
assert classify_value(7) == "Odd"
assert classify_value(0) == "Zero"
assert classify_value("abc") == "Invalid Input"
assert classify_value(3.14) == "Invalid Input"
assert classify_value([1, 2]) == "Invalid Input"
assert classify_value(-4) == "Even"
assert classify_value(-9) == "Odd"

print("All AI-generated test cases for classify_value passed successfully!")
```

OUTPUT:



```
All AI-generated test cases for classify_value passed successfully!
```

Task Description #3 (Palindrome Checker – Apply AI for String Normalization)

• Task: Use AI to generate at least 3 assert test cases for a function is_palindrome(text) and implement the function.

• Requirements:

o Ignore case, spaces, and punctuation.

o Handle edge cases such as empty strings and single characters.

Example Assert Test Cases:

assert is_palindrome("Madam") == True

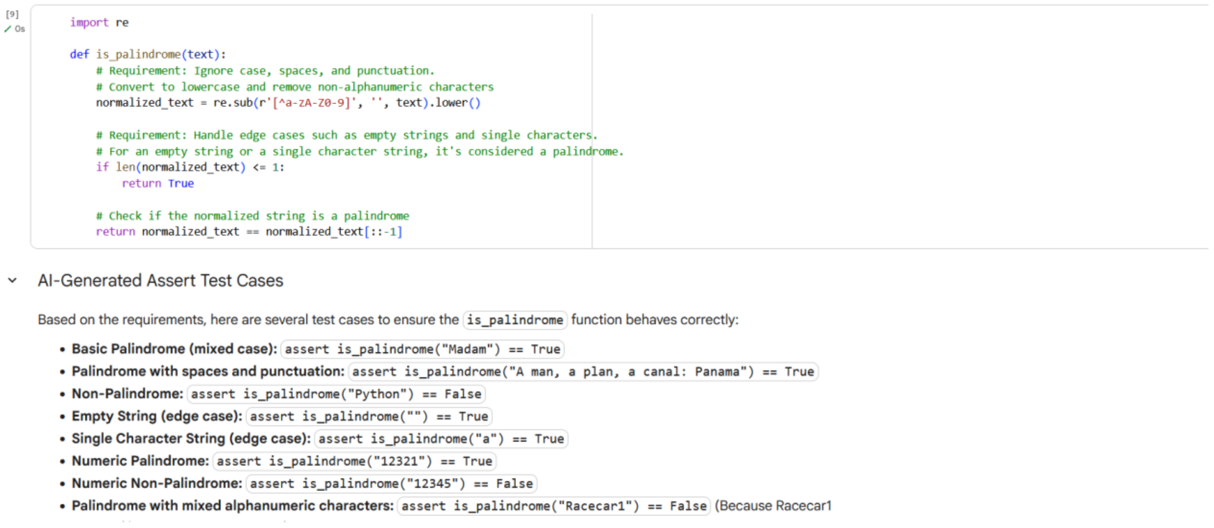
assert is_palindrome("A man a plan a canal Panama") == True

assert is_palindrome("Python") == False

Expected Output #3:

• Function correctly identifying palindromes and passing all AI-generated tests

CODE:

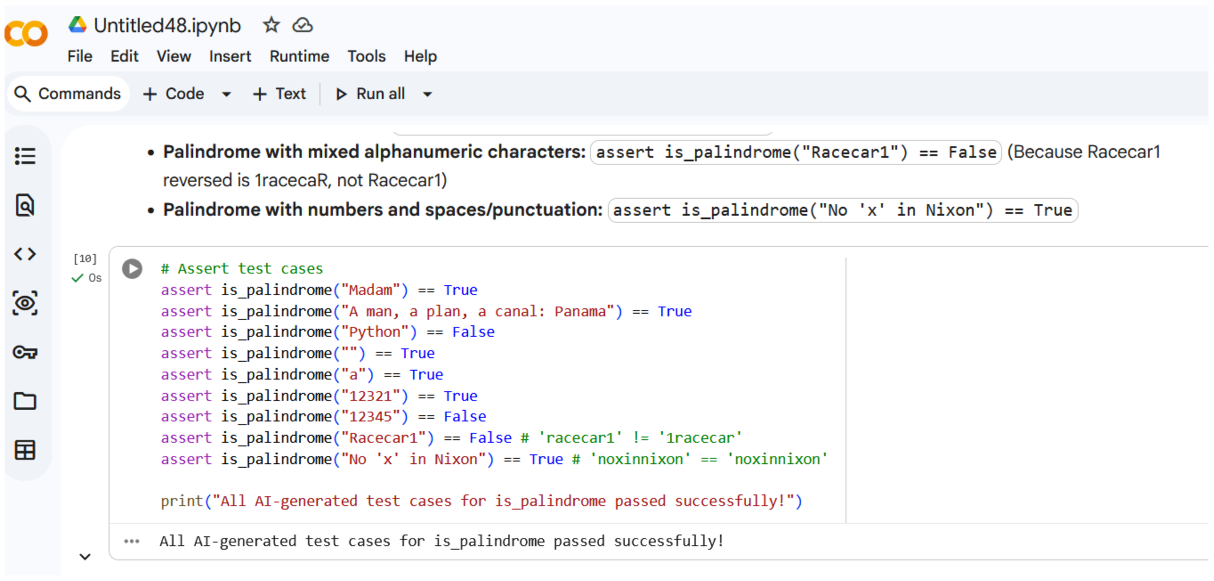


```
import re

def is_palindrome(text):
    # Requirement: Ignore case, spaces, and punctuation.
    # Convert to lowercase and remove non-alphanumeric characters
    normalized_text = re.sub(r'[^a-zA-Z0-9]', '', text).lower()

    # Requirement: Handle edge cases such as empty strings and single characters.
    # For an empty string or a single character string, it's considered a palindrome.
    if len(normalized_text) <= 1:
        return True

    # Check if the normalized string is a palindrome
    return normalized_text == normalized_text[::-1]
```

AI-Generated Assert Test Cases

Based on the requirements, here are several test cases to ensure the `is_palindrome` function behaves correctly:

- **Basic Palindrome (mixed case):** `assert is_palindrome("Madam") == True`
- **Palindrome with spaces and punctuation:** `assert is_palindrome("A man, a plan, a canal: Panama") == True`
- **Non-Palindrome:** `assert is_palindrome("Python") == False`
- **Empty String (edge case):** `assert is_palindrome("") == True`
- **Single Character String (edge case):** `assert is_palindrome("a") == True`
- **Numeric Palindrome:** `assert is_palindrome("12321") == True`
- **Numeric Non-Palindrome:** `assert is_palindrome("12345") == False`
- **Palindrome with mixed alphanumeric characters:** `assert is_palindrome("Racecar1") == False` (Because Racecar1



Untitled48.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all

- **Palindrome with mixed alphanumeric characters:** `assert is_palindrome("Racecar1") == False` (Because Racecar1 reversed is 1racecaR, not Racecar1)
- **Palindrome with numbers and spaces/punctuation:** `assert is_palindrome("No 'x' in Nixon") == True`

```
# Assert test cases
assert is_palindrome("Madam") == True
assert is_palindrome("A man, a plan, a canal: Panama") == True
assert is_palindrome("Python") == False
assert is_palindrome("") == True
assert is_palindrome("a") == True
assert is_palindrome("12321") == True
assert is_palindrome("12345") == False
assert is_palindrome("Racecar1") == False # 'racecar1' != '1racecar'
assert is_palindrome("No 'x' in Nixon") == True # 'noxinnixon' == 'noxinnixon'

print("All AI-generated test cases for is_palindrome passed successfully!")
```

```
All AI-generated test cases for is_palindrome passed successfully!
```

OUTPUT:

```
All AI-generated test cases for is_palindrome passed successfully!
```

Task Description #4 (BankAccount Class – Apply AI for Object-Oriented Test-Driven Development)

• Task: Ask AI to generate at least 3 assert-based test cases for a BankAccount class and then implement the class.

• Methods:

o deposit(amount)

o withdraw(amount)

o get_balance()

Example Assert Test Cases:

acc = BankAccount(1000)

acc.deposit(500)

assert acc.get_balance() == 1500

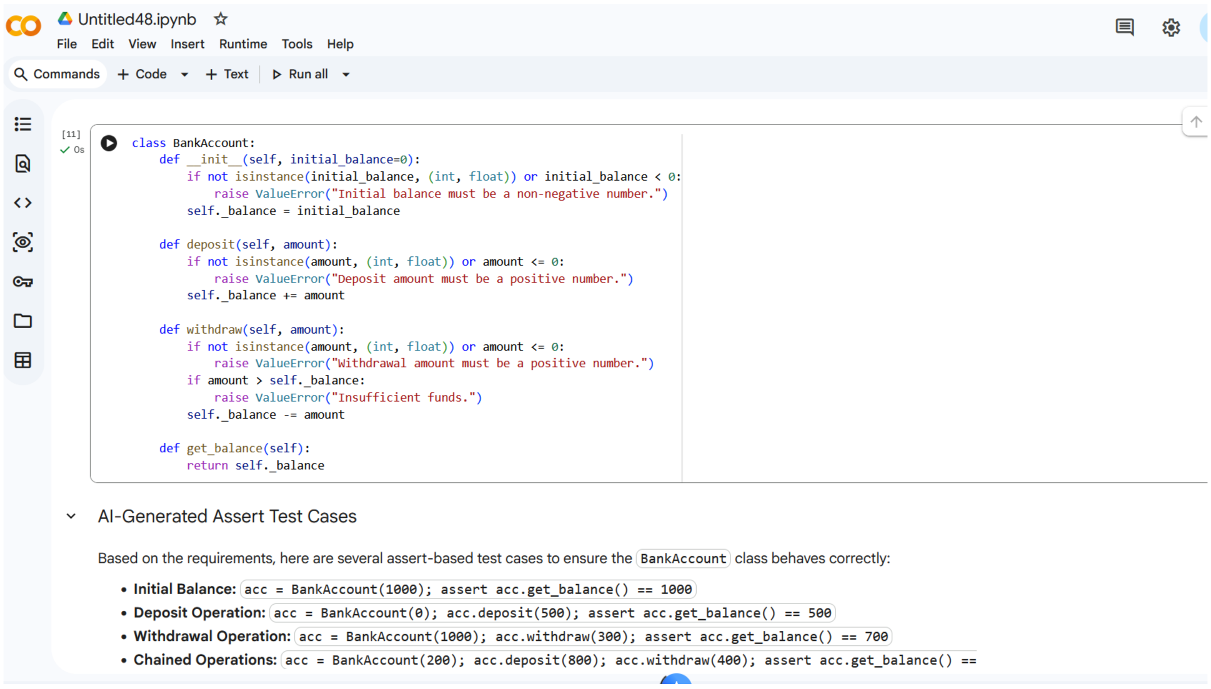
acc.withdraw(300)

assert acc.get_balance() == 1200

Expected Output #4:

• Fully functional class that passes all AI-generated assertions

CODE:

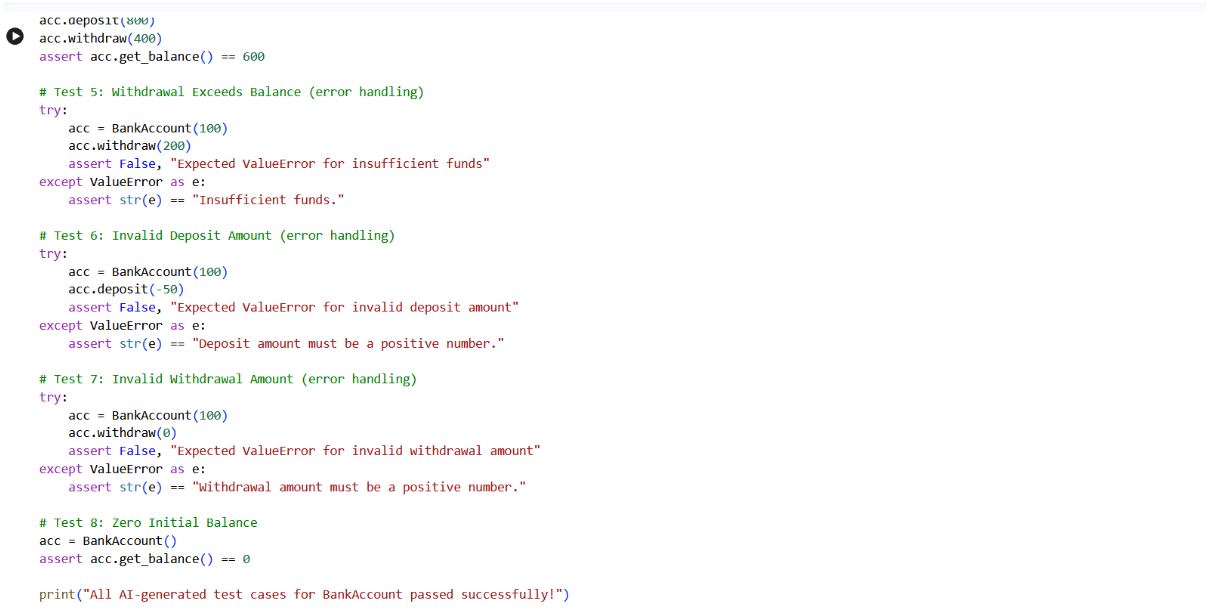

Untitled48.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
[11]
0s

```python
class BankAccount:
    def __init__(self, initial_balance=0):
        if not isinstance(initial_balance, (int, float)) or initial_balance < 0:
            raise ValueError("Initial balance must be a non-negative number.")
        self._balance = initial_balance

    def deposit(self, amount):
        if not isinstance(amount, (int, float)) or amount <= 0:
            raise ValueError("Deposit amount must be a positive number.")
        self._balance += amount

    def withdraw(self, amount):
        if not isinstance(amount, (int, float)) or amount <= 0:
            raise ValueError("Withdrawal amount must be a positive number.")
        if amount > self._balance:
            raise ValueError("Insufficient funds.")
        self._balance -= amount

    def get_balance(self):
        return self._balance
```

AI-Generated Assert Test Cases

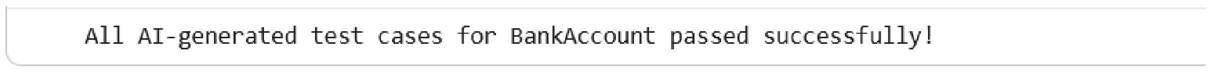
Based on the requirements, here are several assert-based test cases to ensure the `BankAccount` class behaves correctly:

- **Initial Balance:** `acc = BankAccount(1000); assert acc.get_balance() == 1000`
- **Deposit Operation:** `acc = BankAccount(0); acc.deposit(500); assert acc.get_balance() == 500`
- **Withdrawal Operation:** `acc = BankAccount(1000); acc.withdraw(300); assert acc.get_balance() == 700`
- **Chained Operations:** `acc = BankAccount(200); acc.deposit(800); acc.withdraw(400); assert acc.get_balance() ==`



```
acc.deposit(800)
acc.withdraw(400)
assert acc.get_balance() == 600

# Test 5: Withdrawal Exceeds Balance (error handling)
try:
    acc = BankAccount(100)
    acc.withdraw(200)
    assert False, "Expected ValueError for insufficient funds"
except ValueError as e:
    assert str(e) == "Insufficient funds."

# Test 6: Invalid Deposit Amount (error handling)
try:
    acc = BankAccount(100)
    acc.deposit(-50)
    assert False, "Expected ValueError for invalid deposit amount"
except ValueError as e:
    assert str(e) == "Deposit amount must be a positive number."

# Test 7: Invalid Withdrawal Amount (error handling)
try:
    acc = BankAccount(100)
    acc.withdraw(0)
    assert False, "Expected ValueError for invalid withdrawal amount"
except ValueError as e:
    assert str(e) == "Withdrawal amount must be a positive number."

# Test 8: Zero Initial Balance
acc = BankAccount()
assert acc.get_balance() == 0

print("All AI-generated test cases for BankAccount passed successfully!")
```

OUTPUT:



```
All AI-generated test cases for BankAccount passed successfully!
```

Task Description #5 (Email ID Validation – Apply AI for Data Validation)

• Task: Use AI to generate at least 3 assert test cases for a function validate_email(email) and implement the function.

• Requirements:

o Must contain @ and .

o Must not start or end with special characters.

o Should handle invalid formats gracefully.

Example Assert Test Cases:

assert validate_email("user@example.com") == True

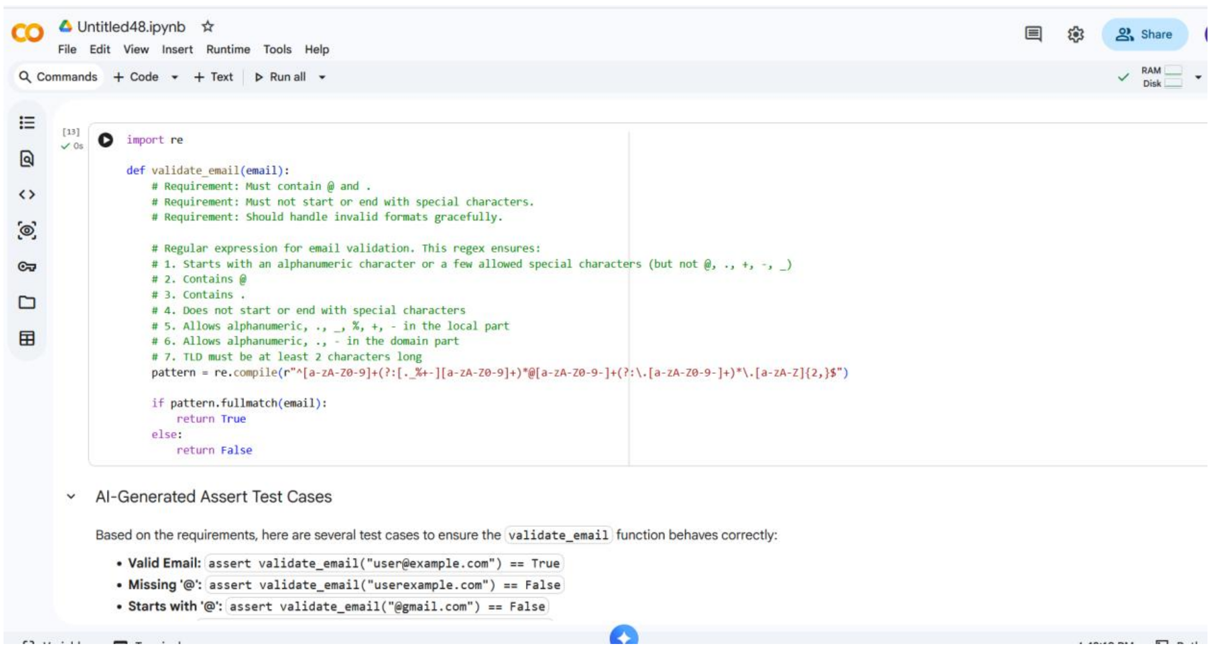
assert validate_email("userexample.com") == False

assert validate_email("@gmail.com") == False

Expected Output #5:

• Email validation function passing all AI-generated test cases and handling edge cases correctly.

CODE:

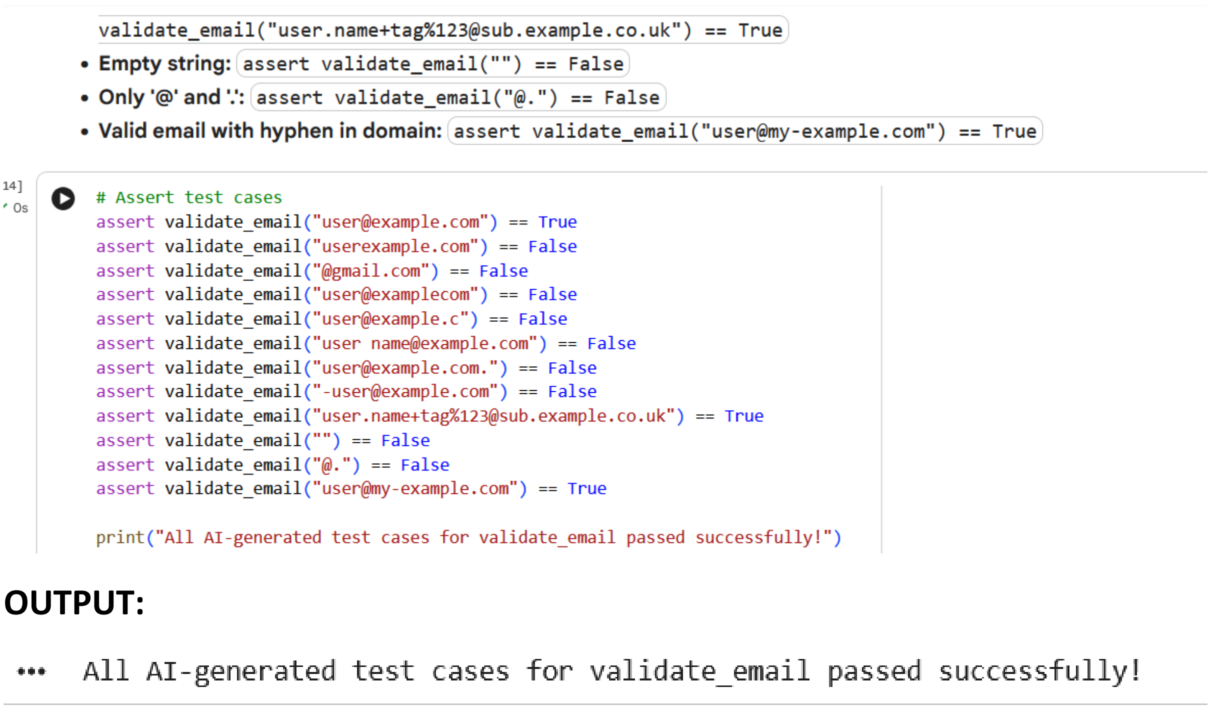

Untitled48.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
RAM Disk

```
[13]
import re

def validate_email(email):
    # Requirement: Must contain @ and .
    # Requirement: Must not start or end with special characters.
    # Requirement: Should handle invalid formats gracefully.

    # Regular expression for email validation. This regex ensures:
    # 1. Starts with an alphanumeric character or a few allowed special characters (but not @, ., +, -, _)
    # 2. Contains @
    # 3. Contains .
    # 4. Does not start or end with special characters
    # 5. Allows alphanumeric, ., _, %, +, - in the local part
    # 6. Allows alphanumeric, ., - in the domain part
    # 7. TLD must be at least 2 characters long
    pattern = re.compile(r"^[a-zA-Z0-9]+(?:[._%+-][a-zA-Z0-9]+)*@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*\.[a-zA-Z]{2,}$")

    if pattern.fullmatch(email):
        return True
    else:
        return False
```

AI-Generated Assert Test Cases

Based on the requirements, here are several test cases to ensure the `validate_email` function behaves correctly:

- **Valid Email:** `assert validate_email("user@example.com") == True`
- **Missing '@':** `assert validate_email("userexample.com") == False`
- **Starts with '@':** `assert validate_email("@gmail.com") == False`



`validate_email("user.name+tag%123@sub.example.co.uk") == True`
- **Empty string:** `assert validate_email("") == False`
- **Only '@' and '.':** `assert validate_email("@.") == False`
- **Valid email with hyphen in domain:** `assert validate_email("user@my-example.com") == True`

```
14]
# Assert test cases
assert validate_email("user@example.com") == True
assert validate_email("userexample.com") == False
assert validate_email("@gmail.com") == False
assert validate_email("user@examplecom") == False
assert validate_email("user@example.c") == False
assert validate_email("user name@example.com") == False
assert validate_email("user@example.com.") == False
assert validate_email("-user@example.com") == False
assert validate_email("user.name+tag%123@sub.example.co.uk") == True
assert validate_email("") == False
assert validate_email("@.") == False
assert validate_email("user@my-example.com") == True

print("All AI-generated test cases for validate_email passed successfully!")
```

OUTPUT:

```
All AI-generated test cases for validate_email passed successfully!
```