ASSIGNMENT - 8.5

AI Assisted Coding

Name : Gonela Rishika

Roll No: 2303A52197

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:

workspace > username_validator.py > is_valid_username

```python
"""
Username Validator - Test-Driven Development Implementation
Task: Validate username based on specific requirements
Requirements:
  - Length: 5-15 characters
  - Characters: Only alphabets and digits
  - Must not start with a digit
  - No spaces allowed
"""


def is_valid_username(username):
    """
    Validates if a username meets all requirements.

    Args:
        username (str): The username to validate

    Returns:
        bool: True if valid, False otherwise
    """
    # Check if username is a string
    if not isinstance(username, str):
        return False

    # Check length requirement (5-15 characters)
    if len(username) < 5 or len(username) > 15:
        return False

    # Check if it starts with a digit
    if username[0].isdigit():
        return False

    # Check if it contains only alphabets and digits (no spaces or special characters)
    if not username.isalnum():
        return False

    return True
```

```python
# ============================================================================
# TEST CASES - Generated using TDD principles
# ============================================================================

print("=" * 70)
print("RUNNING USERNAME VALIDATOR TEST CASES")
print("=" * 70)

# TEST CASE 1: Valid usernames
print("\n[TEST SET 1] Valid Usernames")
assert is_valid_username("User123") == True, "Valid: 'User123' should be True"
print("✓ assert is_valid_username('User123') == True")

assert is_valid_username("Admin") == True, "Valid: 'Admin' should be True"
print("✓ assert is_valid_username('Admin') == True")

assert is_valid_username("John2024") == True, "Valid: 'John2024' should be True"
print("✓ assert is_valid_username('John2024') == True")

assert is_valid_username("alice42") == True, "Valid: 'alice42' should be True"
print("✓ assert is_valid_username('alice42') == True")

assert is_valid_username("Test5") == True, "Valid: 'Test5' (min length) should be True"
print("✓ assert is_valid_username('Test5') == True")

assert is_valid_username("ValidUser12345") == True, "Valid: 'ValidUser12345' (max length) should be True"
print("✓ assert is_valid_username('ValidUser12345') == True")


# TEST CASE 2: Invalid - Starts with digit
print("\n[TEST SET 2] Invalid - Starts with Digit")
assert is_valid_username("12User") == False, "Invalid: '12User' starts with digit"
print("✓ assert is_valid_username('12User') == False")

assert is_valid_username("1Admin") == False, "Invalid: '1Admin' starts with digit"
print("✓ assert is_valid_username('1Admin') == False")

assert is_valid_username("9john") == False, "Invalid: '9john' starts with digit"
print("✓ assert is_valid_username('9john') == False")
```

```
print("✓ assert is_valid_username('User Name') == False")

assert is_valid_username("Admin 123") == False, "Invalid: 'Admin 123' contains space"
print("✓ assert is_valid_username('Admin 123') == False")


# TEST CASE 4: Invalid - Too short (less than 5 characters)
print("\n[TEST SET 4] Invalid - Too Short")
assert is_valid_username("User") == False, "Invalid: 'User' is too short (4 chars)"
print("✓ assert is_valid_username('User') == False")

assert is_valid_username("a") == False, "Invalid: 'a' is too short (1 char)"
print("✓ assert is_valid_username('a') == False")

assert is_valid_username("ab12") == False, "Invalid: 'ab12' is too short (4 chars)"
print("✓ assert is_valid_username('ab12') == False")


# TEST CASE 5: Invalid - Too long (more than 15 characters)
print("\n[TEST SET 5] Invalid - Too Long")
assert is_valid_username("ValidUsername123456") == False, "Invalid: too long (19 chars)"
print("✓ assert is_valid_username('ValidUsername123456') == False")

assert is_valid_username("ThisIsAVeryLongUsername") == False, "Invalid: too long (23 chars)"
print("✓ assert is_valid_username('ThisIsAVeryLongUsername') == False")


# TEST CASE 6: Invalid - Special characters and symbols
print("\n[TEST SET 6] Invalid - Special Characters")
assert is_valid_username("User@123") == False, "Invalid: 'User@123' contains @"
print("✓ assert is_valid_username('User@123') == False")

assert is_valid_username("User-Name") == False, "Invalid: 'User-Name' contains hyphen"
print("✓ assert is_valid_username('User-Name') == False")

assert is_valid_username("User.123") == False, "Invalid: 'User.123' contains dot"
print("✓ assert is_valid_username('User.123') == False")

assert is_valid_username("User_Name") == False, "Invalid: 'User_Name' contains underscore"
print("✓ assert is_valid_username('User_Name') == False")

```

```
print("✓ assert is_valid_username('User#123') == False")


# TEST CASE 7: Invalid - Non-alphabetic starting characters
print("\n[TEST SET 7] Invalid - Non-alphabetic Start")
assert is_valid_username("_User123") == False, "Invalid: '_User123' starts with underscore"
print("✓ assert is_valid_username('_User123') == False")

assert is_valid_username("-Admin") == False, "Invalid: '-Admin' starts with hyphen"
print("✓ assert is_valid_username('-Admin') == False")


# TEST CASE 8: Edge cases
print("\n[TEST SET 8] Edge Cases")
assert is_valid_username("") == False, "Invalid: empty string"
print("✓ assert is_valid_username('') == False")

assert is_valid_username("ALLCAPS") == True, "Valid: 'ALLCAPS' (all uppercase)"
print("✓ assert is_valid_username('ALLCAPS') == True")

assert is_valid_username("allsmall") == True, "Valid: 'allsmall' (all lowercase)"
print("✓ assert is_valid_username('allsmall') == True")

assert is_valid_username("MixedCase123") == True, "Valid: 'MixedCase123' (mixed case)"
print("✓ assert is_valid_username('MixedCase123') == True")


# ============================================================================
# TEST SUMMARY
# ============================================================================

print("\n" + "=" * 70)
print("ALL TESTS PASSED SUCCESSFULLY! ✓")
print("=" * 70)
print("\nValidation Rules Verified:")
print("  ✓ Length validation (5-15 characters)")
print("  ✓ Character validation (only alphabets and digits)")
print("  ✓ Starting character validation (must not start with digit)")
print("  ✓ No spaces allowed")
print("  ✓ No special characters allowed")
print("\n")
```

OUTPUT:

```
============================================================
[TEST SET 3] Invalid - Contains Spaces
✓ assert is_valid_username('Us er') == False
✓ assert is_valid_username('User Name') == False
✓ assert is_valid_username('Admin 123') == False

[TEST SET 4] Invalid - Too Short
✓ assert is_valid_username('User') == False
✓ assert is_valid_username('a') == False
✓ assert is_valid_username('ab12') == False

[TEST SET 5] Invalid - Too Long
✓ assert is_valid_username('ValidUsername123456') == False
✓ assert is_valid_username('ThisIsAVeryLongUsername') == False

[TEST SET 6] Invalid - Special Characters
✓ assert is_valid_username('User@123') == False
✓ assert is_valid_username('User-Name') == False
✓ assert is_valid_username('User.123') == False
✓ assert is_valid_username('User_Name') == False
✓ assert is_valid_username('User#123') == False

[TEST SET 7] Invalid - Non-alphabetic Start
✓ assert is_valid_username('_User123') == False
✓ assert is_valid_username('-Admin') == False

[TEST SET 8] Edge Cases
✓ assert is_valid_username('') == False
✓ assert is_valid_username('ALLCAPS') == True
✓ assert is_valid_username('allsmall') == True
✓ assert is_valid_username('MixedCase123') == True

============================================================
ALL TESTS PASSED SUCCESSFULLY! ✓
============================================================

Validation Rules Verified:
  ✓ Length validation (5-15 characters)
  ✓ Character validation (only alphabets and digits)
  ✓ Starting character validation (must not start with digit)
  ✓ No spaces allowed
  ✓ No special characters allowed
```

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"

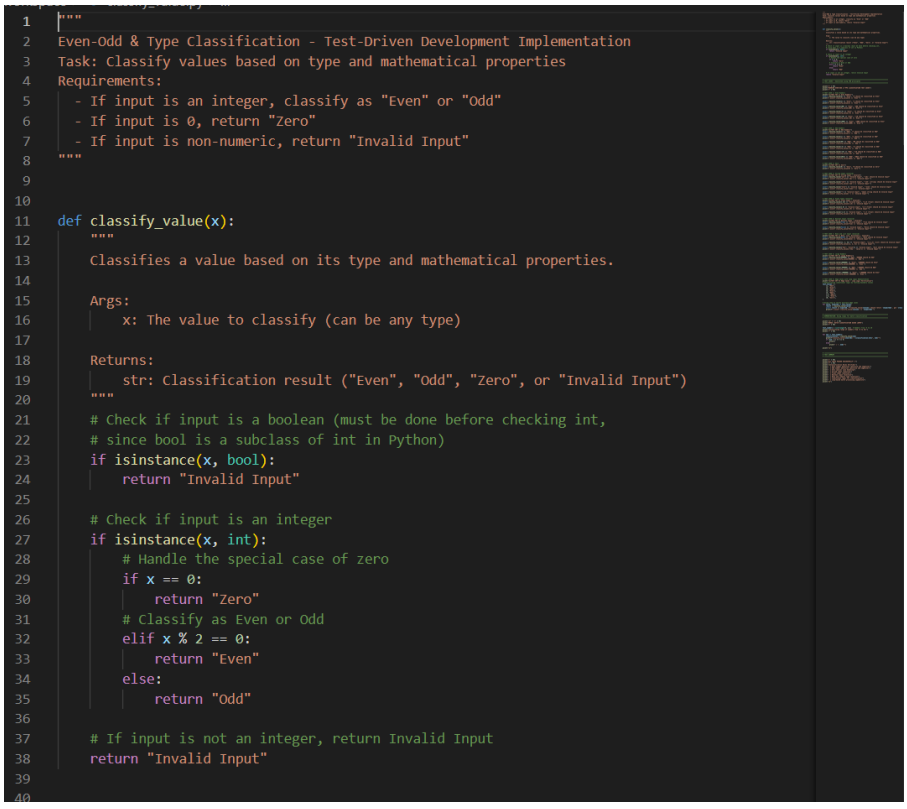
assert classify_value(7) == "Odd"

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

Expected Output #2:

• Function correctly classifying values and passing all test cases.

CODE:



```
"""
Even-Odd & Type Classification - Test-Driven Development Implementation
Task: Classify values based on type and mathematical properties
Requirements:
  - If input is an integer, classify as "Even" or "Odd"
  - If input is 0, return "Zero"
  - If input is non-numeric, return "Invalid Input"
"""


def classify_value(x):
    """
    Classifies a value based on its type and mathematical properties.

    Args:
        x: The value to classify (can be any type)

    Returns:
        str: Classification result ("Even", "Odd", "Zero", or "Invalid Input")
    """
    # Check if input is a boolean (must be done before checking int,
    # since bool is a subclass of int in Python)
    if isinstance(x, bool):
        return "Invalid Input"

    # Check if input is an integer
    if isinstance(x, int):
        # Handle the special case of zero
        if x == 0:
            return "Zero"
        # Classify as Even or Odd
        elif x % 2 == 0:
            return "Even"
        else:
            return "Odd"

    # If input is not an integer, return Invalid Input
    return "Invalid Input"
```

```python
# TEST CASES - Generated using TDD principles
# ============================================================================

print("=" * 70)
print("RUNNING EVEN-ODD & TYPE CLASSIFICATION TEST CASES")
print("=" * 70)

# TEST CASE 1: Even Numbers
print("\n[TEST SET 1] Even Numbers")
assert classify_value(8) == "Even", "8 should be classified as Even"
print("✓ assert classify_value(8) == 'Even'")

assert classify_value(2) == "Even", "2 should be classified as Even"
print("✓ assert classify_value(2) == 'Even'")

assert classify_value(100) == "Even", "100 should be classified as Even"
print("✓ assert classify_value(100) == 'Even'")

assert classify_value(-4) == "Even", "-4 should be classified as Even"
print("✓ assert classify_value(-4) == 'Even'")

assert classify_value(-10) == "Even", "-10 should be classified as Even"
print("✓ assert classify_value(-10) == 'Even'")

assert classify_value(1000) == "Even", "1000 should be classified as Even"
print("✓ assert classify_value(1000) == 'Even'")


# TEST CASE 2: Odd Numbers
print("\n[TEST SET 2] Odd Numbers")
assert classify_value(7) == "Odd", "7 should be classified as Odd"
print("✓ assert classify_value(7) == 'Odd'")

assert classify_value(1) == "Odd", "1 should be classified as Odd"
print("✓ assert classify_value(1) == 'Odd'")

assert classify_value(99) == "Odd", "99 should be classified as Odd"
print("✓ assert classify_value(99) == 'Odd'")

assert classify_value(-3) == "Odd", "-3 should be classified as Odd"
print("✓ assert classify_value(-3) == 'Odd'")
```

```python
assert classify_value(-15) == "Odd", "-15 should be classified as Odd"
print("✓ assert classify_value(-15) == 'Odd'")

assert classify_value(2023) == "Odd", "2023 should be classified as Odd"
print("✓ assert classify_value(2023) == 'Odd'")


# TEST CASE 3: Zero
print("\n[TEST SET 3] Zero")
assert classify_value(0) == "Zero", "0 should be classified as Zero"
print("✓ assert classify_value(0) == 'Zero'")


# TEST CASE 4: String Input (Invalid)
print("\n[TEST SET 4] String Input - Invalid")
assert classify_value("abc") == "Invalid Input", "'abc' should be Invalid Input"
print("✓ assert classify_value('abc') == 'Invalid Input'")

assert classify_value("123") == "Invalid Input", "'123' (string) should be Invalid Input"
print("✓ assert classify_value('123') == 'Invalid Input'")

assert classify_value("even") == "Invalid Input", "'even' should be Invalid Input"
print("✓ assert classify_value('even') == 'Invalid Input'")

assert classify_value("") == "Invalid Input", "empty string should be Invalid Input"
print("✓ assert classify_value('') == 'Invalid Input'")


# TEST CASE 5: Float Input (Invalid)
print("\n[TEST SET 5] Float Input - Invalid")
assert classify_value(3.14) == "Invalid Input", "3.14 (float) should be Invalid Input"
print("✓ assert classify_value(3.14) == 'Invalid Input'")

assert classify_value(2.0) == "Invalid Input", "2.0 (float) should be Invalid Input"
print("✓ assert classify_value(2.0) == 'Invalid Input'")

assert classify_value(-1.5) == "Invalid Input", "-1.5 (float) should be Invalid Input"
print("✓ assert classify_value(-1.5) == 'Invalid Input'")
```

```python
assert classify_value(1000000) == "Even", "1000000 should be Even"
print("✓ assert classify_value(1000000) == 'Even'")

assert classify_value(-999999) == "Odd", "-999999 should be Odd"
print("✓ assert classify_value(-999999) == 'Odd'")

assert classify_value(-1000000) == "Even", "-1000000 should be Even"
print("✓ assert classify_value(-1000000) == 'Even'")


# TEST CASE 9: Edge Cases with Loop Logic Demonstration
print("\n[TEST SET 9] Edge Cases - Loop Logic Verification")
# Demonstrating conditional logic by testing boundary values
test_values = [
    (1, "Odd"),
    (2, "Even"),
    (3, "Odd"),
    (4, "Even"),
    (5, "Odd"),
    (-1, "Odd"),
    (-2, "Even"),
    (0, "Zero"),
]

# Using loop to verify multiple edge cases
for value, expected in test_values:
    result = classify_value(value)
    assert result == expected, f"classify_value({value}) should return '{expected}', got '{res
    print(f"✓ assert classify_value({value}) == '{expected}'")


# ============================================================================
# DEMONSTRATION: Using loops for batch classification
# ============================================================================

print("\n" + "=" * 70)
print("BONUS: BATCH CLASSIFICATION USING LOOPS")
print("=" * 70)

test_numbers = list(range(-5, 11))  # Numbers from -5 to 10
print("\nClassifying range of numbers from -5 to 10:")
```

```python
print("-" * 70)

for num in test_numbers:
    classification = classify_value(num)
    print(f"classify_value({num:3d}) = {classification:15s}", end="")
    if (num + 1) % 4 == 0:
        print()
    else:
        print(" | ", end="")

print("\n")


# ============================================================================
# TEST SUMMARY
# ============================================================================

print("=" * 70)
print("ALL TESTS PASSED SUCCESSFULLY! ✓")
print("=" * 70)
print("\nClassification Rules Verified:")
print("  ✓ Even number detection (positive and negative)")
print("  ✓ Odd number detection (positive and negative)")
print("  ✓ Zero special case handling")
print("  ✓ String input rejection")
print("  ✓ Float input rejection")
print("  ✓ Boolean input rejection")
print("  ✓ None and complex type rejection")
print("  ✓ Conditional logic for classification")
print("  ✓ Loop-based batch processing capability")
print("\n")
```

OUTPUT:

```
======================================================================
RUNNING EVEN-ODD & TYPE CLASSIFICATION TEST CASES
======================================================================

[TEST SET 1] Even Numbers
✓ assert classify_value(8) == 'Even'
✓ assert classify_value(2) == 'Even'
✓ assert classify_value(100) == 'Even'
✓ assert classify_value(-4) == 'Even'
✓ assert classify_value(-10) == 'Even'
✓ assert classify_value(1000) == 'Even'

[TEST SET 2] Odd Numbers
✓ assert classify_value(7) == 'Odd'
✓ assert classify_value(1) == 'Odd'
✓ assert classify_value(99) == 'Odd'
✓ assert classify_value(-3) == 'Odd'
✓ assert classify_value(-15) == 'Odd'
✓ assert classify_value(2023) == 'Odd'

[TEST SET 3] Zero
✓ assert classify_value(0) == 'Zero'

[TEST SET 4] String Input - Invalid
✓ assert classify_value('abc') == 'Invalid Input'
✓ assert classify_value('123') == 'Invalid Input'
✓ assert classify_value('even') == 'Invalid Input'
✓ assert classify_value('') == 'Invalid Input'

[TEST SET 5] Float Input - Invalid
✓ assert classify_value(3.14) == 'Invalid Input'
✓ assert classify_value(2.0) == 'Invalid Input'
✓ assert classify_value(-1.5) == 'Invalid Input'

[TEST SET 6] Boolean Input - Invalid
✓ assert classify_value(True) == 'Invalid Input'
✓ assert classify_value(False) == 'Invalid Input'

[TEST SET 7] None, List, Dictionary - Invalid
✓ assert classify_value(None) == 'Invalid Input'
```

```
✓ assert classify_value(2) == 'Even'
✓ assert classify_value(3) == 'Odd'
✓ assert classify_value(4) == 'Even'
✓ assert classify_value(5) == 'Odd'
✓ assert classify_value(-1) == 'Odd'
✓ assert classify_value(-2) == 'Even'
✓ assert classify_value(0) == 'Zero'

======================================================================
BONUS: BATCH CLASSIFICATION USING LOOPS
======================================================================

Classifying range of numbers from -5 to 10:
----------------------------------------------------------------------
classify_value( -5) = Odd
classify_value( -4) = Even           | classify_value( -3) = Odd           | classify_value( -2) =
 Even           | classify_value( -1) = Odd
classify_value(  0) = Zero           | classify_value(  1) = Odd           | classify_value(  2) =
 Even           | classify_value(  3) = Odd
classify_value(  4) = Even           | classify_value(  5) = Odd           | classify_value(  6) =
 Even           | classify_value(  7) = Odd
classify_value(  8) = Even           | classify_value(  9) = Odd           | classify_value( 10) =
 Even           |

======================================================================
ALL TESTS PASSED SUCCESSFULLY! ✓
======================================================================

Classification Rules Verified:
  ✓ Even number detection (positive and negative)
  ✓ Odd number detection (positive and negative)
  ✓ Zero special case handling
  ✓ String input rejection
  ✓ Float input rejection
  ✓ Boolean input rejection
  ✓ None and complex type rejection
  ✓ Conditional logic for classification
  ✓ Loop-based batch processing capability
```

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:

```python
import re


def is_palindrome(text):
    """
    Check if a string is a palindrome, ignoring case, spaces, and punctuation.

    Args:
        text (str): The string to check

    Returns:
        bool: True if the string is a palindrome, False otherwise
    """
    # Remove non-alphanumeric characters and convert to lowercase
    cleaned = re.sub(r'[^a-z0-9]', '', text.lower())

    # Handle empty strings
    if not cleaned:
        return True

    # Check if the cleaned string equals its reverse
    return cleaned == cleaned[::-1]


# Test cases
if __name__ == "__main__":
    # Basic palindrome examples
    assert is_palindrome("Madam") == True
    assert is_palindrome("A man a plan a canal Panama") == True
    assert is_palindrome("Python") == False

    # Edge cases - empty strings
    assert is_palindrome("") == True
    assert is_palindrome("   ") == True
    assert is_palindrome("!!!") == True

    # Edge cases - single characters
    assert is_palindrome("a") == True
    assert is_palindrome("A") == True
    assert is_palindrome("1") == True
```

```python
    assert is_palindrome("!!!") == True

    # Edge cases - single characters
    assert is_palindrome("a") == True
    assert is_palindrome("A") == True
    assert is_palindrome("1") == True

    # Additional test cases with various punctuation
    assert is_palindrome("race a car") == False
    assert is_palindrome("Was it a car or a cat I saw?") == True
    assert is_palindrome("12321") == True
    assert is_palindrome("12345") == False

    # Case sensitivity tests
    assert is_palindrome("RaceCar") == True
    assert is_palindrome("LEVEL") == True

    # Numbers and mixed characters
    assert is_palindrome("A1b1A") == True
    assert is_palindrome("1a2b2a1") == True

    print("✓ All test cases passed!")
```

OUTPUT:

```
✓ All test cases passed!
```

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

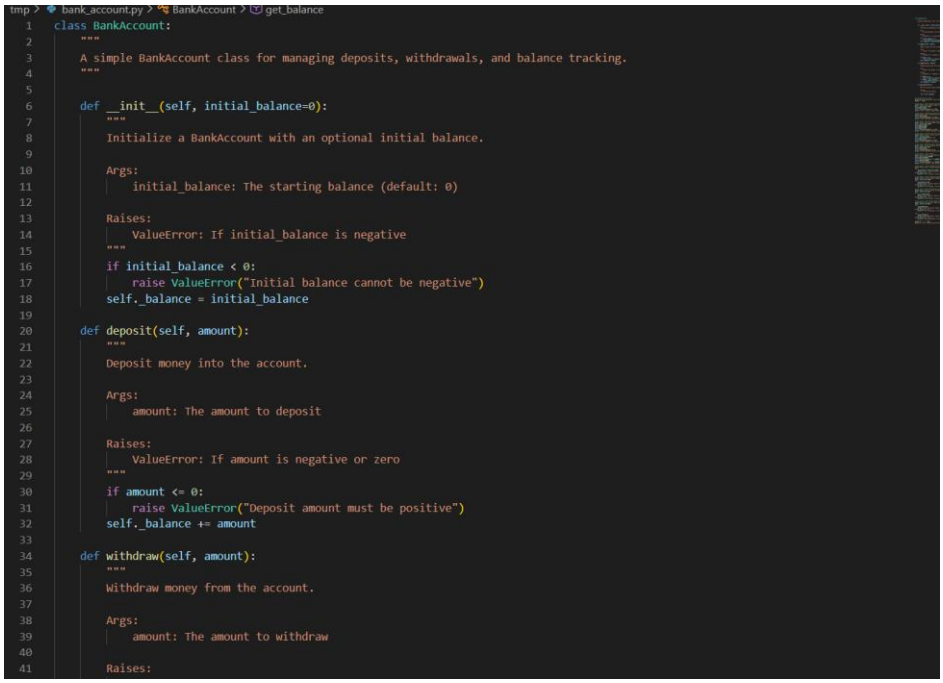
acc.withdraw(300)

assert acc.get_balance() == 1200

Expected Output #4:

• Fully functional class that passes all AI-generated assertions

CODE:



```
        if amount > self._balance:
            raise ValueError("Insufficient funds for withdrawal")
        self._balance -= amount

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

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


# AI-Generated Test Cases
print("Running BankAccount Class Tests...")
print("-" * 60)

# Test Case 1: Basic deposit and balance check
print("\nTest Case 1: Basic deposit and balance check")
acc1 = BankAccount(1000)
acc1.deposit(500)
assert acc1.get_balance() == 1500
print("✓ Test 1 passed: deposit(500) on balance 1000 = 1500")

# Test Case 2: Withdrawal and balance check
print("\nTest Case 2: Withdrawal and balance check")
acc1.withdraw(300)
assert acc1.get_balance() == 1200
print("✓ Test 2 passed: withdraw(300) on balance 1500 = 1200")

# Test Case 3: Multiple transactions
print("\nTest Case 3: Multiple transactions")
acc2 = BankAccount(2000)
acc2.deposit(1000)
acc2.withdraw(500)
acc2.deposit(250)
assert acc2.get_balance() == 2750
print("✓ Test 3 passed: Multiple transactions result in balance 2750")
```

```
78  print("\nTest Case 3: Multiple transactions")
79  acc2 = BankAccount(2000)
80  acc2.deposit(1000)
81  acc2.withdraw(500)
82  acc2.deposit(250)
83  assert acc2.get_balance() == 2750
84  print("✓ Test 3 passed: Multiple transactions result in balance 2750")
85
86  # Test Case 4: Account with zero initial balance
87  print("\nTest Case 4: Account with zero initial balance")
88  acc3 = BankAccount(0)
89  assert acc3.get_balance() == 0
90  acc3.deposit(100)
91  assert acc3.get_balance() == 100
92  print("✓ Test 4 passed: Zero-balance account initialized and deposit works")
93
94  # Test Case 5: Withdraw entire balance
95  print("\nTest Case 5: Withdraw entire balance")
96  acc4 = BankAccount(500)
97  acc4.withdraw(500)
98  assert acc4.get_balance() == 0
99  print("✓ Test 5 passed: Withdraw entire balance leaves 0")
00
01  # Test Case 6: Large amounts
02  print("\nTest Case 6: Large amounts handling")
03  acc5 = BankAccount(1000000)
04  acc5.deposit(500000)
05  assert acc5.get_balance() == 1500000
06  acc5.withdraw(250000)
07  assert acc5.get_balance() == 1250000
08  print("✓ Test 6 passed: Large amounts handled correctly")
09
10  # Test Case 7: Invalid negative initial balance
11  print("\nTest Case 7: Invalid negative initial balance")
12  try:
13      acc_invalid = BankAccount(-100)
14      print("✗ Test 7 failed: Should raise ValueError for negative balance")
15  except ValueError as e:
16      print(f"✓ Test 7 passed: Correctly raised ValueError: {e}")
17
18  # Test Case 8: Invalid negative deposit
```

```
try:
    acc_invalid = BankAccount(-100)
    print("✗ Test 7 failed: Should raise ValueError for negative balance")
except ValueError as e:
    print(f"✓ Test 7 passed: Correctly raised ValueError: {e}")

# Test Case 8: Invalid negative deposit
print("\nTest Case 8: Invalid negative deposit")
acc6 = BankAccount(1000)
try:
    acc6.deposit(-100)
    print("✗ Test 8 failed: Should raise ValueError for negative deposit")
except ValueError as e:
    print(f"✓ Test 8 passed: Correctly raised ValueError: {e}")

# Test Case 9: Invalid withdrawal exceeding balance
print("\nTest Case 9: Invalid withdrawal exceeding balance")
acc7 = BankAccount(100)
try:
    acc7.withdraw(200)
    print("✗ Test 9 failed: Should raise ValueError for insufficient funds")
except ValueError as e:
    print(f"✓ Test 9 passed: Correctly raised ValueError: {e}")

# Test Case 10: Zero amount operations
print("\nTest Case 10: Zero amount operations")
acc8 = BankAccount(500)
try:
    acc8.deposit(0)
    print("✗ Test 10a failed: Should raise ValueError for zero deposit")
except ValueError as e:
    print(f"✓ Test 10a passed: Correctly raised ValueError for zero deposit: {e}")

try:
    acc8.withdraw(0)
    print("✗ Test 10b failed: Should raise ValueError for zero withdrawal")
except ValueError as e:
    print(f"✓ Test 10b passed: Correctly raised ValueError for zero withdrawal: {e}")

print("\n" + "-" * 60)
print("All tests completed successfully! ✓")
```

OUTPUT:

```
Running BankAccount Class Tests...
------------------------------------------------------------

Test Case 1: Basic deposit and balance check
✓ Test 1 passed: deposit(500) on balance 1000 = 1500

Test Case 2: Withdrawal and balance check
✓ Test 2 passed: withdraw(300) on balance 1500 = 1200

Test Case 3: Multiple transactions
✓ Test 3 passed: Multiple transactions result in balance 2750

Test Case 4: Account with zero initial balance
✓ Test 4 passed: Zero-balance account initialized and deposit works

Test Case 5: Withdraw entire balance
✓ Test 5 passed: Withdraw entire balance leaves 0

Test Case 6: Large amounts handling
✓ Test 6 passed: Large amounts handled correctly

Test Case 7: Invalid negative initial balance
✓ Test 7 passed: Correctly raised ValueError: Initial balance cannot be negative

Test Case 8: Invalid negative deposit
✓ Test 8 passed: Correctly raised ValueError: Deposit amount must be positive

Test Case 9: Invalid withdrawal exceeding balance
✓ Test 9 passed: Correctly raised ValueError: Insufficient funds for withdrawal

Test Case 10: Zero amount operations
✓ Test 10a passed: Correctly raised ValueError for zero deposit: Deposit amount must be positive
✓ Test 10b passed: Correctly raised ValueError for zero withdrawal: Withdrawal amount must be positive

------------------------------------------------------------
All tests completed 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

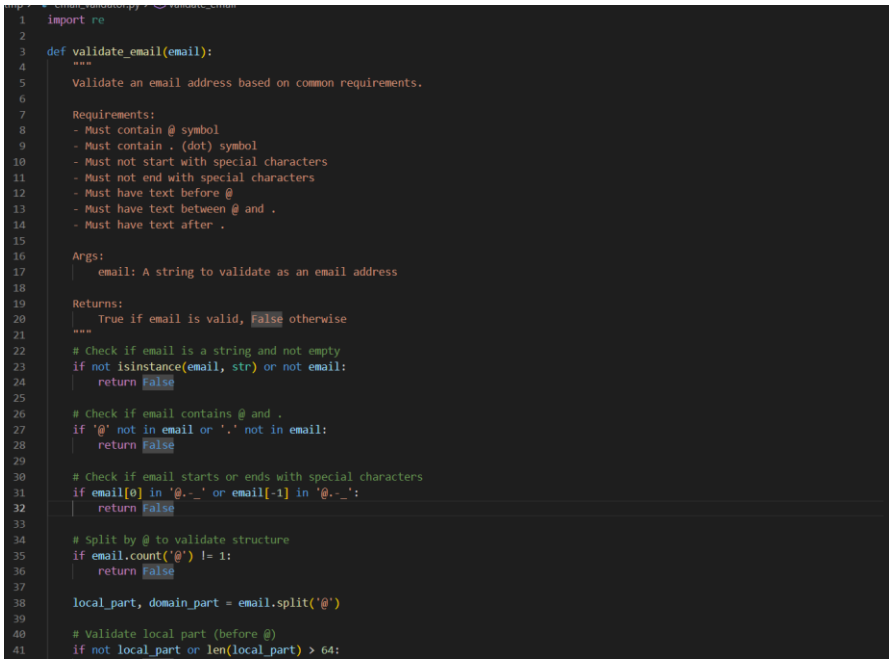
assert validate_email("userexample.com") == False

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

Expected Output #5:

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

CODE:



```python
import re

def validate_email(email):
    """
    Validate an email address based on common requirements.

    Requirements:
    - Must contain @ symbol
    - Must contain . (dot) symbol
    - Must not start with special characters
    - Must not end with special characters
    - Must have text before @
    - Must have text between @ and .
    - Must have text after .

    Args:
        email: A string to validate as an email address

    Returns:
        True if email is valid, False otherwise
    """
    # Check if email is a string and not empty
    if not isinstance(email, str) or not email:
        return False

    # Check if email contains @ and .
    if '@' not in email or '.' not in email:
        return False

    # Check if email starts or ends with special characters
    if email[0] in '@._' or email[-1] in '@._':
        return False

    # Split by @ to validate structure
    if email.count('@') != 1:
        return False

    local_part, domain_part = email.split('@')

    # Validate local part (before @)
    if not local_part or len(local_part) > 64:
```

```
        return False

    # Local part should not start/end with special chars (except . is allowed inside)
    if local_part[0] in '.-_' or local_part[-1] in '.-_':
        return False

    # Validate domain part (after @)
    if not domain_part or len(domain_part) > 255:
        return False

    # Domain must contain exactly one . at proper position
    if domain_part.count('.') < 1:
        return False

    # Split domain by .
    domain_parts = domain_part.split('.')

    # Check that domain parts are not empty
    for part in domain_parts:
        if not part or len(part) > 63:
            return False

    # Check that parts start/end with alphanumeric
    for part in domain_parts:
        if not part[0].isalnum() or not part[-1].isalnum():
            return False

    # Validate characters used in email
    # Allowed: alphanumeric, dot, hyphen, underscore in local part
    # Allowed: alphanumeric, dot, hyphen in domain part
    valid_local_chars = set('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789._-+')
    valid_domain_chars = set('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-')

    if not all(c in valid_local_chars for c in local_part):
        return False

    if not all(c in valid_domain_chars for c in domain_part):
        return False

    return True
```

```
# AI-Generated Test Cases
print("Running Email Validation Tests...")
print("-" * 70)

# Test Case 1: Valid email with standard format
print("\nTest Case 1: Valid email with standard format")
assert validate_email("user@example.com") == True
print("✓ Test 1 passed: validate_email('user@example.com') == True")

# Test Case 2: Invalid email - missing @
print("\nTest Case 2: Invalid email - missing @")
assert validate_email("userexample.com") == False
print("✓ Test 2 passed: validate_email('userexample.com') == False")

# Test Case 3: Invalid email - starts with @
print("\nTest Case 3: Invalid email - starts with @")
assert validate_email("@gmail.com") == False
print("✓ Test 3 passed: validate_email('@gmail.com') == False")

# Test Case 4: Valid email with numbers
print("\nTest Case 4: Valid email with numbers")
assert validate_email("user123@example456.com") == True
print("✓ Test 4 passed: validate_email('user123@example456.com') == True")

# Test Case 5: Valid email with underscore and dot in local part
print("\nTest Case 5: Valid email with underscore and dot in local part")
assert validate_email("john.doe_123@example.com") == True
print("✓ Test 5 passed: validate_email('john.doe_123@example.com') == True")

# Test Case 6: Invalid email - ends with dot
print("\nTest Case 6: Invalid email - ends with dot")
assert validate_email("user@example.") == False
print("✓ Test 6 passed: validate_email('user@example.') == False")

# Test Case 7: Invalid email - multiple @ symbols
print("\nTest Case 7: Invalid email - multiple @ symbols")
assert validate_email("user@exam@ple.com") == False
print("✓ Test 7 passed: validate_email('user@exam@ple.com') == False")

# Test Case 8: Invalid email - missing domain extension
print("\nTest Case 8: Invalid email - missing domain extension")
assert validate_email("user@example") == False
```

```
assert validate_email("") == False
print("✓ Test 9 passed: validate_email('') == False")

# Test Case 10: Invalid email - starts with dot
print("\nTest Case 10: Invalid email - starts with dot")
assert validate_email(".user@example.com") == False
print("✓ Test 10 passed: validate_email('.user@example.com') == False")

# Test Case 11: Invalid email - ends with special character
print("\nTest Case 11: Invalid email - ends with special character")
assert validate_email("user@example.com-") == False
print("✓ Test 11 passed: validate_email('user@example.com-') == False")

# Test Case 12: Valid email with hyphen in domain
print("\nTest Case 12: Valid email with hyphen in domain")
assert validate_email("user@ex-ample.com") == True
print("✓ Test 12 passed: validate_email('user@ex-ample.com') == True")

# Test Case 13: Invalid email - space in email
print("\nTest Case 13: Invalid email - space in email")
assert validate_email("user @example.com") == False
print("✓ Test 13 passed: validate_email('user @example.com') == False")

# Test Case 14: Invalid email - non-alphanumeric in domain start
print("\nTest Case 14: Invalid email - hyphen at domain start")
assert validate_email("user@-example.com") == False
print("✓ Test 14 passed: validate_email('user@-example.com') == False")

# Test Case 15: Valid email with plus sign in local part
print("\nTest Case 15: Valid email with plus sign in local part")
assert validate_email("user+tag@example.com") == True
print("✓ Test 15 passed: validate_email('user+tag@example.com') == True")

# Test Case 16: Invalid email - None type
print("\nTest Case 16: Invalid email - None type")
assert validate_email(None) == False
print("✓ Test 16 passed: validate_email(None) == False")

# Test Case 17: Invalid email - missing local part
print("\nTest Case 17: Invalid email - missing local part")
assert validate_email("@example.com") == False
print("✓ Test 17 passed: validate_email('@example.com') == False")
```

```
print("\nTest Case 16: Invalid email - None type")
assert validate_email(None) == False
print("✓ Test 16 passed: validate_email(None) == False")

# Test Case 17: Invalid email - missing local part
print("\nTest Case 17: Invalid email - missing local part")
assert validate_email("@example.com") == False
print("✓ Test 17 passed: validate_email('@example.com') == False")

# Test Case 18: Valid email with subdomain
print("\nTest Case 18: Valid email with subdomain")
assert validate_email("user@mail.example.co.uk") == True
print("✓ Test 18 passed: validate_email('user@mail.example.co.uk') == True")

# Test Case 19: Invalid email - consecutive dots
print("\nTest Case 19: Invalid email - consecutive dots in domain")
assert validate_email("user@example..com") == False
print("✓ Test 19 passed: validate_email('user@example..com') == False")

# Test Case 20: Valid simple email
print("\nTest Case 20: Valid simple email")
assert validate_email("a@b.c") == True
print("✓ Test 20 passed: validate_email('a@b.c') == True")

print("\n" + "-" * 70)
print("All 20 test cases completed successfully! ✓")
```

OUTPUT:

```
PS C:\tmp> python -c "exec(open('/tmp/email_validator.py').read())"
Running Email Validation Tests...
----------------------------------------------------------------------

Test Case 1: Valid email with standard format
âœ“ Test 1 passed: validate_email('user@example.com') == True

Test Case 2: Invalid email - missing @
âœ“ Test 2 passed: validate_email('userexample.com') == False

Test Case 3: Invalid email - starts with @
âœ“ Test 3 passed: validate_email('@gmail.com') == False

Test Case 4: Valid email with numbers
âœ“ Test 4 passed: validate_email('user123@example456.com') == True

Test Case 5: Valid email with underscore and dot in local part
âœ“ Test 5 passed: validate_email('john.doe_123@example.com') == True

Test Case 6: Invalid email - ends with dot
âœ“ Test 6 passed: validate_email('user@example.') == False

Test Case 7: Invalid email - multiple @ symbols
âœ“ Test 7 passed: validate_email('user@exam@ple.com') == False

Test Case 8: Invalid email - missing domain extension
âœ“ Test 8 passed: validate_email('user@example') == False

Test Case 9: Invalid email - empty string
âœ“ Test 9 passed: validate_email('') == False

Test Case 10: Invalid email - starts with dot
âœ“ Test 10 passed: validate_email('.user@example.com') == False

Test Case 11: Invalid email - ends with special character
âœ“ Test 11 passed: validate_email('user@example.com-') == False

Test Case 12: Valid email with hyphen in domain
âœ“ Test 12 passed: validate_email('user@ex-ample.com') == True

Test Case 13: Invalid email - space in email
âœ“ Test 13 passed: validate_email('user @example.com') == False
```

```
PS C:\tmp> python -c "exec(open('/tmp/email_validator.py').read())"
âœ“ Test 14 passed: validate_email('user@-example.com') == False

Test Case 15: Valid email with plus sign in local part
âœ“ Test 15 passed: validate_email('user+tag@example.com') == True

Test Case 16: Invalid email - None type
âœ“ Test 16 passed: validate_email(None) == False

Test Case 17: Invalid email - missing local part
âœ“ Test 17 passed: validate_email('@example.com') == False

Test Case 18: Valid email with subdomain
âœ“ Test 18 passed: validate_email('user@mail.example.co.uk') == True

Test Case 19: Invalid email - consecutive dots in domain
âœ“ Test 19 passed: validate_email('user@example..com') == False

Test Case 20: Valid simple email
âœ“ Test 20 passed: validate_email('a@b.c') == True

----------------------------------------------------------------------
All 20 test cases completed successfully! âœ“
PS C:\tmp>
```