

## **ASSIGNMENT – 8.5**

**HT.NO: 2303A52283**

**BATCH: 45**

### **TASK - 01:**

#### **Question:**

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
```

#### **Prompt:**

Write at least 3 assert test cases for a function `is_valid_username(username)` with these rules:

Length between 5 and 15

Only letters and numbers

Cannot start with a number

No spaces

Then write the function so all test cases pass.

#### **Generated Code:**

```
def is_valid_username(username):
```

```
    """Check if the username is valid based on specific rules.
```

Rules:

- Length between 5 and 15 characters
- Only letters and numbers
- Cannot start with a number
- No spaces

Args:

username (str): The username to validate.

Returns:

bool: True if the username is valid, False otherwise.

"""

```
if not (5 <= len(username) <= 15):  
    raise ValueError("Username must be between 5 and 15 characters long.")  
  
if not username.isalnum():  
    raise ValueError("Username can only contain letters and numbers.")  
  
if username[0].isdigit():  
    raise ValueError("Username cannot start with a number.")  
  
if ' ' in username:  
    raise ValueError("Username cannot contain spaces.")  
  
return True  
  
# Test cases  
  
try:  
    assert is_valid_username("User123") == True  
    print("Test case 1 passed."  
  
    assert is_valid_username("123User") == False  
    print("Test case 2 passed."  
  
except AssertionError as e:
```

```

print(f"AssertionError: {e}")

try:
    assert is_valid_username("User 123") == False
    print("Test case 3 passed.")

except AssertionError as e:
    print(f"AssertionError: {e}")

try:
    assert is_valid_username("Us") == False
    print("Test case 4 passed.")

except AssertionError as e:
    print(f"AssertionError: {e}")

try:
    assert is_valid_username("ValidUser") == True
    print("Test case 5 passed.")

except AssertionError as e:
    print(f"AssertionError: {e}")

```

### **Output:**

```

Test case 1 passed.
Traceback (most recent call last):
  File "c:\Users\RIMSHA NAAZ\OneDrive\Documents\OneDrive\Documents\AAC-PY\AAC-8.5.py", line 37, in <module>
    assert is_valid_username("123User") == False
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\Users\RIMSHA NAAZ\OneDrive\Documents\OneDrive\Documents\AAC-PY\AAC-8.5.py", line 28, in is_valid_username
    raise ValueError("Username cannot start with a number.")
ValueError: Username cannot start with a number.

```

### **Explanation:**

This function validates a username using specific rules and raises a `ValueError` if any rule is broken (length, characters, starting digit, or spaces).

If the username follows all rules, it returns `True`.

The test cases use `assert` inside `try` blocks to check whether the function behaves correctly for valid and invalid usernames.

## **TASK - 02:**

### **Question:**

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:

- If input is an integer, classify as "Even" or "Odd".
- If input is 0, return "Zero".
- 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"
```

**Prompt:**

Create at least 3 assert test cases for a function `classify_value(x)` with these rules:

If input is an integer, return "Even" or "Odd"

If input is 0, return "Zero"

If input is not numeric, return "Invalid Input"

Then write the function using conditional statements and a loop so that all test cases pass.

check every test case step by step execute the next test case only when previous test case is executed successfully. If any test case fails, print the error message and stop further execution.

**Generated Code:**

```
def classify_value(x):  
    """Classify the input value based on specific rules.  
    """
```

Rules:

- If input is an integer, return "Even" or "Odd"
- If input is 0, return "Zero"
- If input is not numeric, return "Invalid Input"

Args:

x: The value to classify.

Returns:

str: The classification of the input value.

"""

```
if isinstance(x, int):
```

```
    if x == 0:
```

```
        return "Zero"
```

```
    elif x % 2 == 0:
```

```
        return "Even"
```

```
    else:
```

```
        return "Odd"
```

```
else:
```

```
    return "Invalid Input"
```

```
# Test cases
```

```
try:
```

```
    assert classify_value(4) == "Even"
```

```
    print("Test case 1 passed.")
```

```
assert classify_value(7) == "Odd"
```

```
print("Test case 2 passed.")
```

```
except AssertionError as e:
```

```
    print(f"AssertionError: {e}")
```

```
try:
```

```
    assert classify_value(0) == "Zero"
```

```
    print("Test case 3 passed.")
```

```
except AssertionError as e:
```

```
    print(f"AssertionError: {e}")
```

```
try:
```

```
    assert classify_value("Hello") == "Invalid Input"
```

```
print("Test case 4 passed.")

except AssertionError as e:
    print(f"AssertionError: {e}")

try:
    assert classify_value(3.14) == "Invalid Input"
    print("Test case 5 passed.")

except AssertionError as e:
    print(f"AssertionError: {e}")
```

#### Output:

```
Test case 1 passed.
Test case 2 passed.
Test case 3 passed.
Test case 4 passed.
Test case 5 passed.
```

#### Explanation:

This function checks the type of input. If it is an integer, it returns "Zero" for 0, "Even" for even numbers, and "Odd" for odd numbers.

If the input is not an integer (like a string or float), it returns "Invalid Input".

The test cases use assert statements to verify that the function works correctly for different inputs.

## TASK - 03:

#### Question:

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

- Requirements:
  - Ignore case, spaces, and punctuation.
  - 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.

**Prompt:**

Generate at least 3 assert test cases for a function `is_palindrome(text)` with the following rules:

Ignore case, spaces, and punctuation

Handle edge cases like empty strings and single characters

Then implement the function Check each test case step by step, and if any test case fails, print the error message and stop further execution.

**Generated Code:**

```
import string  
  
def is_palindrome(text):  
    """Check if the input text is a palindrome, ignoring case, spaces, and punctuation.  
  
    Args:  
        text (str): The text to check.  
  
    Returns:  
        bool: True if the text is a palindrome, False otherwise.  
    """
```

Args:

`text (str): The text to check.`

Returns:

`bool: True if the text is a palindrome, False otherwise.`

""""

```
# Remove spaces and punctuation, and convert to lowercase  
cleaned_text = ''.join(char for char in text if char.isalnum()).lower()  
  
# Check if the cleaned text is equal to its reverse  
return cleaned_text == cleaned_text[::-1]
```

# Test cases

```

try:
    assert is_palindrome("A man, a plan, a canal, Panama") == True
    print("Test case 1 passed.")

    assert is_palindrome("Hello") == False
    print("Test case 2 passed.")

except AssertionError as e:
    print(f"AssertionError: {e}")

try:
    assert is_palindrome("") == True
    print("Test case 3 passed.")

    assert is_palindrome("a") == True
    print("Test case 4 passed.")

except AssertionError as e:
    print(f"AssertionError: {e}")

try:
    assert is_palindrome("No 'x' in Nixon") == True
    print("Test case 5 passed.")

except AssertionError as e:
    print(f"AssertionError: {e}")

```

### Output:

```

Test case 1 passed.
Test case 2 passed.
Test case 3 passed.
Test case 4 passed.
Test case 5 passed.

```

```

PS C:\Users\RIMSHA NAAZ\OneDrive\Documents\OneDrive\Documents\AAC-PY> python -m doctest -v AAC-8.5.PY
1 items had no tests:
  AAC-8.5.PY
0 tests in 1 items.
0 passed and 0 failed.
Test passed.

```

### **Explanation:**

This function checks whether a text is a palindrome by removing spaces and punctuation and converting everything to lowercase.

It then compares the cleaned text with its reverse to see if they are the same.

The test cases verify that the function works for normal sentences, single letters, empty strings, and non-palindromes.

## **TASK - 04:**

### **Question:**

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.

### **Prompt:**

Generate at least 3 assert-based test cases for a BankAccount class with the following methods:

```
deposit(amount)  
withdraw(amount)
```

get\_balance() and test the testcases using doctest testcases. Then implement the BankAccount class to ensure all test cases pass. If any test case fails, print the error message and stop further execution.

### **Generated Code:**

class BankAccount:

```
"""A simple BankAccount class to manage deposits, withdrawals, and balance."""
```

```
def __init__(self):
```

```
    """Initialize the bank account with a balance of 0."""
```

```
    self.balance = 0
```

```
def deposit(self, amount):
```

```
    """Deposit a specified amount into the bank account.
```

Args:

amount (float): The amount to deposit. Must be positive.

```
"""
```

```
if amount <= 0:
```

```
    raise ValueError("Deposit amount must be positive.")
```

```
    self.balance += amount
```

```
def withdraw(self, amount):
```

```
    """Withdraw a specified amount from the bank account.
```

Args:

amount (float): The amount to withdraw. Must be positive and less than or equal to the current balance.

```
"""
```

```
if amount <= 0:
```

```
    raise ValueError("Withdrawal amount must be positive.")
```

```
if amount > self.balance:
```

```
    raise ValueError("Insufficient funds.")
```

```
    self.balance -= amount

def get_balance(self):
    """Return the current balance of the bank account."""
    return self.balance

# Test cases using doctest
import doctest

def test_bank_account():
    """
    >>> account = BankAccount()
    >>> account.get_balance()
    0
    >>> account.deposit(100)
    >>> account.get_balance()
    100
    >>> account.withdraw(30)
    >>> account.get_balance()
    70
    >>> account.withdraw(100)

Traceback (most recent call last):
...
ValueError: Insufficient funds.

>>> account.deposit(-50)

Traceback (most recent call last):
...     ValueError: Deposit amount must be positive.

>>> account.withdraw(-20)

Traceback (most recent call last):
...     ValueError: Withdrawal amount must be positive.

    """
```

```

pass

if __name__ == "__main__":
    doctest.testmod()

```

### **Output:**

```

File "c:\Users\RIMSHA NAAZ\OneDrive\Documents\OneDrive\Documents\AAC-PY\AAC-8.py", line 208, in __main__.test_bank_account
Failed example:
    account.withdraw(-20)
Exception raised:
Traceback (most recent call last):
  File "C:\Users\RIMSHA NAAZ\AppData\Local\Programs\Python\Python311\Lib\doctest.py", line 1350, in __run
    exec(compile(example.source, filename, "single",
  File "<doctest __main__.test_bank_account[8]>", line 1, in <module>
    account.withdraw(-20)
  File "c:\Users\RIMSHA NAAZ\OneDrive\Documents\OneDrive\Documents\AAC-PY\AAC-8.py", line 180, in withdraw
    raise ValueError("Withdrawal amount must be positive.")
ValueError: Withdrawal amount must be positive.
*****
1 items had failures:
  2 of   9 in __main__.test_bank_account
***Test Failed*** 2 failures.

```

### **Explanation:**

This BankAccount class manages a simple bank account with methods to deposit, withdraw, and check the balance. It raises a ValueError if the deposit/withdraw amount is not positive or if there are insufficient funds. The doctest section automatically tests the class methods by running the examples written inside the docstring.

## **TASK - 05:**

### **Question:**

- Task: Use AI to generate at least 3 assert test cases for a function validate\_email(email) and implement the function.
- Requirements:
  - Must contain @ and .
  - Must not start or end with special characters.
  - 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.

**Prompt:**

Generate at least 3 assert test cases for a function validate\_email(email) with these rules:

Must contain @ and .

Must not start or end with special characters

Should handle invalid formats properly and test cases in pytest framework. Then implement the function to ensure all test cases pass. If any test case fails, print the error message and stop further execution.

**Generated Code:**

```
import re

def validate_email(email):
    """Validate an email address based on specific rules.
```

Rules:

- Must contain @ and .
- Must not start or end with special characters.
- Should handle invalid formats properly.

Args:

email (str): The email address to validate.

Returns:

bool: True if the email is valid, False otherwise.

"""

# Check for presence of @ and .

if '@' not in email or '.' not in email:

return False

# Check for valid format using regex

```

pattern = r'^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'

if re.match(pattern, email):
    return True
else:
    return False

# Test cases using pytest

def test_validate_email():

    assert validate_email("user@example.com") == True
    print("Test case 1 passed.")

    assert validate_email("invalid.email") == False
    print("Test case 2 passed.")

    assert validate_email("@example.com") == False
    print("Test case 3 passed.")

    assert validate_email("user@.com") == False
    print("Test case 4 passed.")

    assert validate_email("user@example") == False
    print("Test case 5 passed.")

if __name__ == "__main__":
    test_validate_email()

```

### Output:

```

PS C:\Users\RIMSHA NAAZ\OneDrive\Documents\OneDrive\Documents\AAC-PY> python -m pytest -v AAC-8.py
=====
test session starts =====
platform win32 -- Python 3.11.0, pytest-9.0.2, pluggy-1.6.0 -- C:\Users\RIMSHA NAAZ\AppData\Local\Programs\Python\Python311\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\RIMSHA NAAZ\OneDrive\Documents\OneDrive\Documents\AAC-PY
collected 2 items

AAC-8.py::test_bank_account PASSED [ 50%]
AAC-8.py::test_validate_email PASSED [100%]

===== 2 passed in 0.08s =====

```

### Explanation:

This function checks whether an email address is valid. It first ensures the email contains @ and ., then uses a regular expression (regex) to verify the correct format (proper username, domain, and extension). The test cases confirm that valid emails return True and incorrectly formatted emails return False.

