BATCH 29

ASSIGNMENT 8.2

Task Description

Task 1 – Test-Driven Development for Even/Odd Number Validator

• Use AI tools to first generate test cases for a function is_even(n) and then implement the function so that it satisfies all generated tests.

Requirements:

• Input must be an integer

• Handle zero, negative numbers, and large integers

Example Test Scenarios:

is_even(2) → True

is_even(7) → False

is_even(0) → True

is_even(-4) → True

is_even(9) → False

Expected Output -1

• A correctly implemented is_even() function that passes all AI-generated test cases

PROMPT:

Generate assert-based test cases for function is_even(n) following TDD.
Requirements: handle zero, negative numbers, large integers, and invalid input.

CODE:

[2]

```python
def is_even(n):
    if not isinstance(n, int):
        raise TypeError("Input must be integer")
    return n % 2 == 0
print("Running Test Cases for is_even()...")

assert is_even(2) == True
assert is_even(7) == False
assert is_even(0) == True
assert is_even(-4) == True
assert is_even(9) == False
assert is_even(1000000) == True

try:
    is_even("10")
except TypeError:
    print("Passed invalid input test")

print("All is_even tests passed!\n")
```

```
Running Test Cases for is_even()...
Passed invalid input test
All is_even tests passed!
```

JUSTIFICATION:

I first defined expected behaviors using assert. This clarified how the function should respond to normal and edge cases. Writing tests before implementation made development focused and structured. The function now correctly handles negatives, zero, and invalid input.

Task Description

Task 2 – Test-Driven Development for String Case Converter

• Ask AI to generate test cases for two functions:

• to_uppercase(text)

• to_lowercase(text)

Requirements:

• Handle empty strings

• Handle mixed-case input

• Handle invalid inputs such as numbers or None

Example Test Scenarios:

to_uppercase("ai coding") → "AI CODING"

to_lowercase("TEST") → "test"

to_uppercase("") → ""

to_lowercase(None) → Error or safe handling

Expected Output -2

• Two string conversion functions that pass all AI-generated test cases with safe input handling.

CODE:

```
[3]
✓ 0s
def to_uppercase(text):
    if not isinstance(text, str):
        raise TypeError("Input must be string")
    return text.upper()

def to_lowercase(text):
    if not isinstance(text, str):
        raise TypeError("Input must be string")
    return text.lower()
print("Running Test Cases for String Case Converter...")

assert to_uppercase("ai coding") == "AI CODING"
assert to_lowercase("TEST") == "test"
assert to_uppercase("") == ""
assert to_lowercase("PyThOn") == "python"

try:
    to_lowercase(None)
except TypeError:
    print("Passed invalid input test")

print("All String Case tests passed!\n")
```

```
Running Test Cases for String Case Converter...
Passed invalid input test
All String Case tests passed!
```

JUSTIFICATION:

Using assert-based tests helped verify uppercase/lowercase conversions and error handling. TDD ensured empty strings and invalid inputs were considered before implementation.

Task Description

Task 3 – Test-Driven Development for List Sum Calculator

• Use AI to generate test cases for a function sum_list(numbers)

that calculates the sum of list elements.

Requirements:

• Handle empty lists

• Handle negative numbers

• Ignore or safely handle non-numeric values

Example Test Scenarios:

sum_list([1, 2, 3]) → 6

sum_list([]) → 0

sum_list([-1, 5, -4]) → 0

sum_list([2, "a", 3]) → 5

Expected Output 3

• A robust list-sum function validated using AI-generated test

Cases

CODE:

[4]
✓ 0s

```python
def sum_list(numbers):
    if not isinstance(numbers, list):
        raise TypeError("Input must be list")

    total = 0
    for item in numbers:
        if isinstance(item, (int, float)):
            total += item
    return total
print("Running Test Cases for sum_list()...")

assert sum_list([1, 2, 3]) == 6
assert sum_list([]) == 0
assert sum_list([-1, 5, -4]) == 0
assert sum_list([2, "a", 3]) == 5

print("All sum_list tests passed!\n")
```

```
Running Test Cases for sum_list()...
All sum_list tests passed!
```

JUSTIFICATION:

I ensured empty lists return 0 and non-numeric values are ignored safely. Writing tests first clarified the behavior for mixed data types.

Task Description

Task 4 – Test Cases for Student Result Class

• Generate test cases for a StudentResult class with the following methods:

• add_marks(mark)

• calculate_average()

• get_result()

Requirements:

• Marks must be between 0 and 100

• Average ≥ 40 → Pass, otherwise Fail

Example Test Scenarios:

Marks: [60, 70, 80] → Average: 70 → Result: Pass

Marks: [30, 35, 40] → Average: 35 → Result: Fail

Marks: [-10] → Error

Expected Output -4

• A fully functional StudentResult class that passes all AI-generated test

CODE:

[5]

```
            self.marks.append(mark)
        def calculate_average(self):
            if not self.marks:
                return 0
            return sum(self.marks) / len(self.marks)
        def get_result(self):
            return "Pass" if self.calculate_average() >= 40 else "Fail"
    print("Running Test Cases for StudentResult...")
    s1 = StudentResult()
    s1.add_marks(60)
    s1.add_marks(70)
    s1.add_marks(80)
    assert s1.calculate_average() == 70
    assert s1.get_result() == "Pass"
    s2 = StudentResult()
    s2.add_marks(30)
    s2.add_marks(35)
    s2.add_marks(40)
    assert s2.get_result() == "Fail"
    try:
        s3 = StudentResult()
        s3.add_marks(-10)
    except ValueError:
        print("Passed invalid marks test")
    print("All StudentResult tests passed!\n")
```

```
Running Test Cases for StudentResult...
Passed invalid marks test
All StudentResult tests passed!
```

JUSTIFICATION:

This task applied TDD in an object-oriented context. Test cases defined clear grading rules before writing class logic. Error handling ensures realistic validation.

Task Description

Task 5 – Test-Driven Development for Username Validator

Requirements:

• Minimum length: 5 characters

• No spaces allowed

• Only alphanumeric characters

Example Test Scenarios:

is_valid_username("user01") → True

is_valid_username("ai") → False

is_valid_username("user name") → False

is_valid_username("user@123") → False

Expected Output 5

A username validation function that passes all AI-generated test cases.

PROMPT:

Generate assert-based test cases for a Python function is_valid_username(username) following TDD.
Requirements:

- **Minimum 5 characters**
- **No spaces allowed**
- **Only alphanumeric characters allowed**

- **Return False for invalid inputs like None or non-string types**
  First generate test cases, then implement the function to satisfy all tests.
  Include print statements showing successful test execution.

CODE:

```
[6]
✓ 0s
def is_valid_username(username):
    if not isinstance(username, str):
        return False
    if len(username) < 5:
        return False
    if " " in username:
        return False
    if not username.isalnum():
        return False
    return True
print("Running Test Cases for is_valid_username()...")

assert is_valid_username("user01") == True
assert is_valid_username("ai") == False
assert is_valid_username("user name") == False
assert is_valid_username("user@123") == False

print("All Username tests passed!\n")
```

```
Running Test Cases for is_valid_username()...
All Username tests passed!
```

JUSTIFICATION:

This task emphasizes validation logic. Writing tests first helped define clear username rules. The function now strictly checks length, spaces, and allowed characters. This approach ensures clean input validation similar to real-world authentication systems.