ASSIGNMENT-8.2

2303A52441
Batch-37

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:-

write the code based on the test cases

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

Code:-

```
"""
write the code based on the test cases
    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
"""
def is_even(num):
    if not isinstance(num, int):
        return "Error: Input must be an integer."
    return num % 2 == 0
# Example usage
if __name__ == '__main__':
    print(is_even(2))    # True
    print(is_even(7))    # False
    print(is_even(0))    # True
    print(is_even(-4))   # True
    print(is_even(9))    # False
    print(is_even(3.5)) # Error: Input must be an integer.
```

Explain code:-

This code checks whether the given number `n` is even or not. First, it makes sure the input is an integer, otherwise it returns an error message. Then it uses `n % 2` to check the remainder—if the remainder is `0`, it returns `True` (even), else it returns `False` (odd).

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.

Prompt:-

generate test cases for two functions:to_uppercase(text) to_lowercase(text) Handle empty strings Handle mixed-case input Handle invalid inputs such as numbers or None

Code:-

```
# generate test cases for two functions:to_uppercase(text)
to_lowercase(text) 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



"""

def to_uppercase(text):

    if not isinstance(text, str):

        return "Error: Input must be a string."

    return text.upper()

def to_lowercase(text):

    if not isinstance(text, str):
```

```
        return "Error: Input must be a string."

    return text.lower()

# Example usage

if __name__ == '__main__':

    print(to_uppercase("ai coding"))   # "AI CODING"

    print(to_lowercase("TEST"))        # "test"

    print(to_uppercase(""))            # ""

    print(to_lowercase(None))          # Error: Input must be a string.

    print(to_uppercase(123))           # Error: Input must be a string.

    print(to_lowercase(45.6))         # Error: Input must be a string
```

Explain code:

This code defines two functions: `to_uppercase()` converts a text into all capital letters, and `to_lowercase()` converts a text into all small letters. Before converting, both functions check if the input is a string, and if not (like None or numbers), they return an error message. In the main part, different test cases are printed to show that the functions work correctly for normal text, empty strings, and invalid inputs.

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.

Prompt:-

#generate test cases for a function sum_list(numbers) that calculates the sum of list elements. Handle empty lists Handle negative numbers Ignore or safely handle non-numeric values

Code:-

```
#generate test cases for a function sum_list(numbers) that calculates the
sum of list elements. 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

"""

def sum_list(numbers):

    if not isinstance(numbers, list):

        return "Error: Input must be a list."

    total = 0

    for num in numbers:

        if isinstance(num, (int, float)):

            total += num

        else:

            print(f"Warning: Ignoring non-numeric value '{num}'")

    return total

# Example usage

if __name__ == '__main__':

    print(sum_list([1, 2, 3]))          # 6

    print(sum_list([]))                  # 0

    print(sum_list([-1, 5, -4]))         # 0

    print(sum_list([2, "a", 3]))         # Warning: Ignoring non-numeric
value 'a' \n 5

    print(sum_list("not a list"))         # Error: Input must be a list.

    print(sum_list([1, 2, None, 4]))     # Warning: Ignoring non-numeric
value 'None' \n 7
```

Explain code:-

This code adds all numeric values int from the given list and returns the total sum. If the list contains non-numeric values like strings or None, it safely ignores them instead of causing an error.

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

Prompt:-

#Generate test cases for a StudentResult class with the following methods: add_marks(mark) calculate_average() get_result() Marks must be between 0 and 100 Average ≥ 40 → Pass, otherwise Fail

Code:-

```
#Generate test cases for a StudentResult class with the following methods:
add_marks(mark) calculate_average() get_result() 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
"""
class StudentResult:
    def __init__(self):
        self.marks = []
    def add_marks(self, mark):
        if not isinstance(mark, (int, float)):
            return "Error: Mark must be a number."
        if mark < 0 or mark > 100:
            return "Error: Mark must be between 0 and 100."
        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):
        average = self.calculate_average()
        return "Pass" if average >= 40 else "Fail"
# Example usage
if __name__ == '__main__':
    student = StudentResult()
    student.add_marks(60)
    student.add_marks(70)
    student.add_marks(80)
    print(f"Average: {student.calculate_average()}")  # Average: 70.0
    print(f"Result: {student.get_result()}")          # Result: Pass
    student2 = StudentResult()
    student2.add_marks(30)
    student2.add_marks(35)
    student2.add_marks(40)
    print(f"Average: {student2.calculate_average()}") # Average: 35.0
    print(f"Result: {student2.get_result()}")         # Result: Fail
    student3 = StudentResult()
    print(student3.add_marks(-10))                    # Error: Mark must
be between 0 and 100.
```

Explain code:-

This code creates a `StudentResult` class that stores marks and only allows values between 0 and 100, otherwise it returns an error message. It then calculates the average of all marks and prints Pass if the average is 40 or more, otherwise it prints Fail.

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.

Note: Report should be submitted a word document for all tasks in a

single document with prompts, comments & code explanation, and

output and if required, screenshots

Prompt —

#generate test cases for Username Validator function validate_username(username) that checks if a username is valid. Handle empty strings Handle special characters Handle length constraints (e.g., 3-15 characters) Minimum length: 5 characters No spaces allowed Only alphanumeric characters

Code:-

```
#generate test cases for Username Validator function
validate_username(username) that checks if a username is valid. Handle
empty strings Handle special characters Handle length constraints (e.g.,
3-15 characters) 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

"""

import re

def validate_username(username):

    if not isinstance(username, str):

        return "Error: Input must be a string."

    if len(username) < 5 or len(username) > 15:

        return False
```

```
    if ' ' in username:
        return False
    if not re.match("^[a-zA-Z0-9]+$", username):
        return False
    return True
# Example usage
if __name__ == '__main__':
    print(validate_username("user01"))      # True
    print(validate_username("ai"))          # False
    print(validate_username("user name"))   # False
    print(validate_username("user@123"))    # False
    print(validate_username(""))             # False
    print(validate_username("validUser123")) # True
    print(validate_username("short"))        # False
    print(validate_username("thisisaverylongusername")) # False
    print(validate_username(12345))         # Error: Input must be a
string.
```

Explain code:-

This code checks whether a username is valid by making sure it is a string, has a length between 5 and 15 characters, and does not contain spaces. It also uses a regular expression (`re.match`) to allow only alphanumeric characters (letters and numbers), so special symbols like @ are rejected.