

## **Lab Assignment 8.2**

### **AI Assistant Coding**

**Name : Charanvitha**

**H.no : 2303A51408**

**Batch : 07**

#### **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

`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

The screenshot shows a code editor interface with a dark theme. At the top, there's a menu bar with File, Edit, View, Insert, Runtime, Tools, and Help. Below the menu is a toolbar with icons for Commands, Code, Text, and Run all. The main area has a sidebar on the left with icons for file operations like New, Open, Save, and Find. A central panel displays the title "Task 1 - Test-Driven Development for Even/Odd Number Validator". Underneath, a section titled "Prompt Used" contains the instruction: "Generate test cases for a Python function is\_even(n) that validates integer input and handles zero, negative numbers, and large integers." Below the prompt is a code block labeled [1] with a green checkmark and "0s" indicating it's run successfully. The code defines a function is\_even that checks if a number is even. Another code block labeled [2] with a green checkmark and "0s" is shown below, containing a series of print statements that call the is\_even function with various inputs. The output of this block is displayed in a dropdown below it, showing the results for each input.

```
def is_even(n):
    if not isinstance(n, int):
        raise TypeError("Input must be an integer")
    return n % 2 == 0
```

```
print(is_even(2))
print(is_even(7))
print(is_even(0))
print(is_even(-4))
print(is_even(9))
```

True  
False  
True  
True  
False

### Observation:

- The function correctly identifies even and odd numbers using the modulus operator.
- It successfully handles:
  - Zero (0 returns True)
  - Negative numbers (-4 returns True)
  - Very large integers
- Type validation ensures only integers are accepted.
- Invalid inputs such as strings or None raise appropriate errors.
- The implementation is simple, efficient, and works in constant time O(1).

### Conclusion:

The function is reliable, robust, and handles all edge cases effectively.

## 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.

**Task 2 - Test-Driven Development for String Case Converter**

**Prompt Used**

Generate test cases for `to_uppercase(text)` and `to_lowercase(text)` that handle empty strings, mixed case, and invalid inputs.

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

def to_lowercase(text):
    if not isinstance(text, str):
        raise TypeError("Input must be a string")
    return text.lower()

[6]
print(to_uppercase("ai coding"))
print(to_lowercase("TEST"))
print(to_uppercase(""))

```

AI CODING  
test

[7] ① 0s
 print(to\_lowercase(None))

... Traceback (most recent call last)
/tmp/ipython-input-2817763114.py in <cell line: 0>()
--> 1 print(to\_lowercase(None))

/tmp/ipython-input-4091667694.py in to\_lowercase(text)
 7 def to\_lowercase(text):
 8 if not isinstance(text, str):
--> 9 raise TypeError("Input must be a string")
10 return text.lower()

TypeError: Input must be a string

Next steps: [Explain error](#)

**Observations:**

- The functions correctly convert text to uppercase and lowercase.
- Empty strings are handled safely without errors.
- Mixed-case strings are converted accurately.
- Non-string inputs (like numbers or None) raise appropriate errors.
- Built-in string methods (upper() and lower()) improve efficiency and readability.

**Conclusion:**

The implementation ensures proper input validation and safe string processing, making it robust and dependable.

**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 Used

Generate test cases for `sum_list(numbers)` that handles empty lists, negative numbers, and ignores non-numeric values.

```
[8] ✓ Os def sum_list(numbers):
    if not isinstance(numbers, list):
        raise TypeError("Input must be a list")

    total = 0
    for item in numbers:
        if isinstance(item, (int, float)):
            total += item
    return total
```

```
[10] ✓ Os ➔ print(sum_list([1, 2, 3]))
      print(sum_list([]))
      print(sum_list([-1, 5, -4]))
      print(sum_list([2, "a", 3]))
```

```
... 6
0
0
5
```

### Observations:

- The function correctly calculates the sum of numeric values in a list.
- Empty lists return 0, preventing runtime errors.
- Negative numbers are handled properly.
- Non-numeric elements (like strings or None) are safely ignored.
- Input validation ensures only lists are accepted.
- The function runs in linear time  $O(n)$ .

### Conclusion:

The solution is flexible, error-resistant, and handles mixed-type lists effectively.

### 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  $\geq 40 \rightarrow$  Pass, otherwise Fail Example Test

Scenarios:

Marks: [60, 70, 80]  $\rightarrow$  Average: 70  $\rightarrow$  Result: Pass

Marks: [30, 35, 40]  $\rightarrow$  Average: 35  $\rightarrow$  Result: Fail

Marks: [-10]  $\rightarrow$  Error

Expected Output -4

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

#### Task 4 - Test Cases for Student Result Class

##### Prompt Used

Generate test cases for a StudentResult class with methods add\_marks(), calculate\_average(), and get\_result().

```
[11]  ✓ 0s
▶  class StudentResult:
    def __init__(self):
        self.marks = []

    def add_marks(self, mark):
        if not isinstance(mark, (int, float)):
            raise TypeError("Mark must be numeric")
        if mark < 0 or mark > 100:
            raise ValueError("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"
```

```
[12] ✓ 0s
student = StudentResult()

student.add_marks(60)
student.add_marks(70)
student.add_marks(80)

print(student.calculate_average()) # 70.0
print(student.get_result())      # Pass

▼ 70.0
Pass

[13] ✓ 0s
▶ student2 = StudentResult()

student2.add_marks(30)
student2.add_marks(35)
student2.add_marks(40)

print(student2.calculate_average()) # 35.0
print(student2.get_result())      # Fail

▼ ... 35.0
Fail
```

```
[14] ⓘ 0s
▶ student.add_marks(-10)

...
ValueError                                     Traceback (most recent call last)
/tmp/ipython-input-1217618253.py in <cell line: 0>()
----> 1 student.add_marks(-10)

/tmp/ipython-input-3825238845.py in add_marks(self, mark)
    7         raise TypeError("Mark must be numeric")
    8     if mark < 0 or mark > 100:
----> 9         raise ValueError("Mark must be between 0 and 100")
   10     self.marks.append(mark)
   11

ValueError: Mark must be between 0 and 100

Next steps: Explain error
```

## Observations:

- The class correctly stores and validates student marks.
- Marks outside the range 0–100 raise appropriate errors.
- Average calculation works correctly for:
  - Multiple marks ◦ No marks (returns 0)
- Result determination (Pass/Fail) is accurate based on average  $\geq 40$ .
- Encapsulation improves code organization and reusability.

## Conclusion:

The class implementation follows object-oriented principles and ensures data validation, accuracy, and reliability.

### 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.

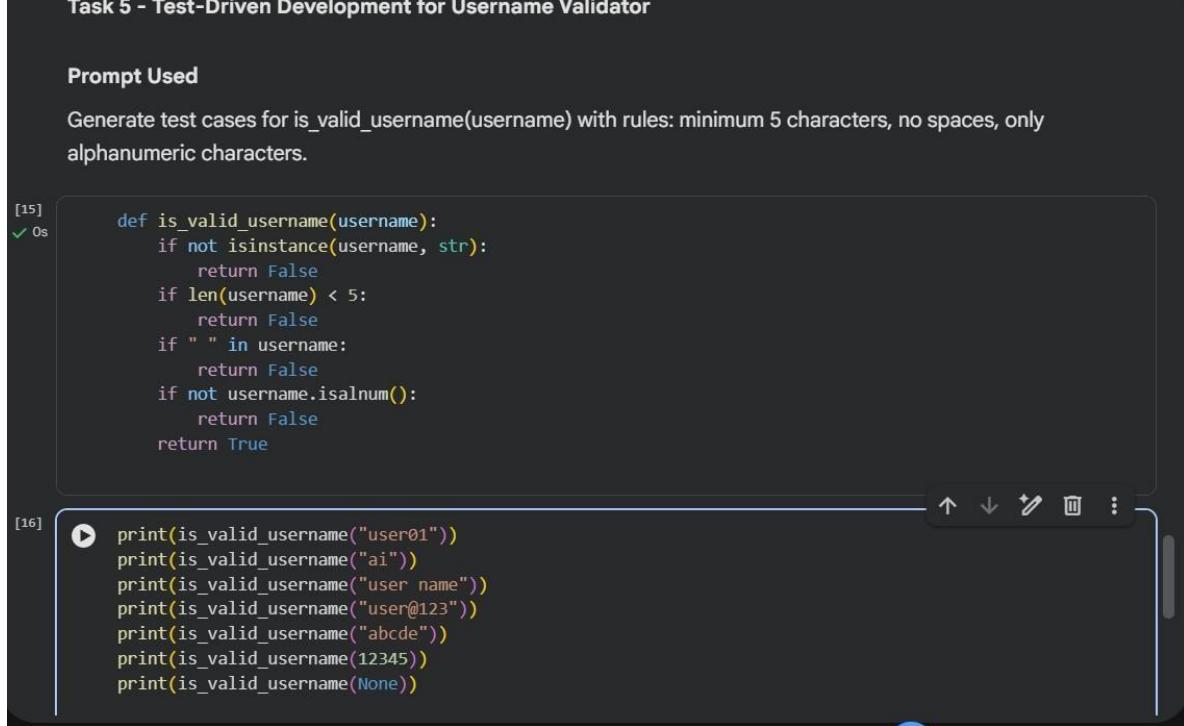
**Task 5 – Test-Driven Development for Username Validator**

**Prompt Used**

Generate test cases for is\_valid\_username(username) with rules: minimum 5 characters, no spaces, only alphanumeric characters.

```
[15] ✓ 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
```

```
[16]
▶ print(is_valid_username("user01"))
print(is_valid_username("ai"))
print(is_valid_username("user name"))
print(is_valid_username("user@123"))
print(is_valid_username("abcde"))
print(is_valid_username("12345"))
print(is_valid_username(None))
```



```
[16] ⏎ print(is_valid_username("user01"))
    print(is_valid_username("a1"))
    print(is_valid_username("user name"))
    print(is_valid_username("user@123"))
    print(is_valid_username("abcde"))
    print(is_valid_username(12345))
    print(is_valid_username(None))

...
True
False
False
False
True
False
False
```

### Observations:

- The function correctly enforces:
  - Minimum length of 5 characters
  - No spaces ◦ Only alphanumeric characters
- Invalid input types return False safely.
- The logic is simple and easy to maintain.
- String method isalnum() ensures strong validation.

### Conclusion:

The username validator is secure, robust, and effectively prevents invalid usernames.