

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 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

The screenshot shows a code editor with a dark theme. At the top, there is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu bar is a toolbar with 'Commands', '+ Code', '+ Text', and 'Run all'. On the left side, there is a sidebar with icons for file explorer, search, and other tools. The main area displays the following code:

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

Prompt Used
Generate test cases for a Python function is_even(n) that validates integer input and handles zero, negative numbers, and large integers.

[1]
✓ Os
def is_even(n):
    if not isinstance(n, int):
        raise TypeError("Input must be an integer")
    return n % 2 == 0

[2]
✓ Os
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.

The screenshot shows a Jupyter Notebook with the following content:

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] 0s
print(to_lowercase(None))
```

```
...
-----
TypeError                                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]
✓ 0s
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]
✓ 0s
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] ✓ Os
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] ✓ Os
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] ⓘ Os
student.add_marks(-10)

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

/tmp/ipynthon-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 ≥ 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:

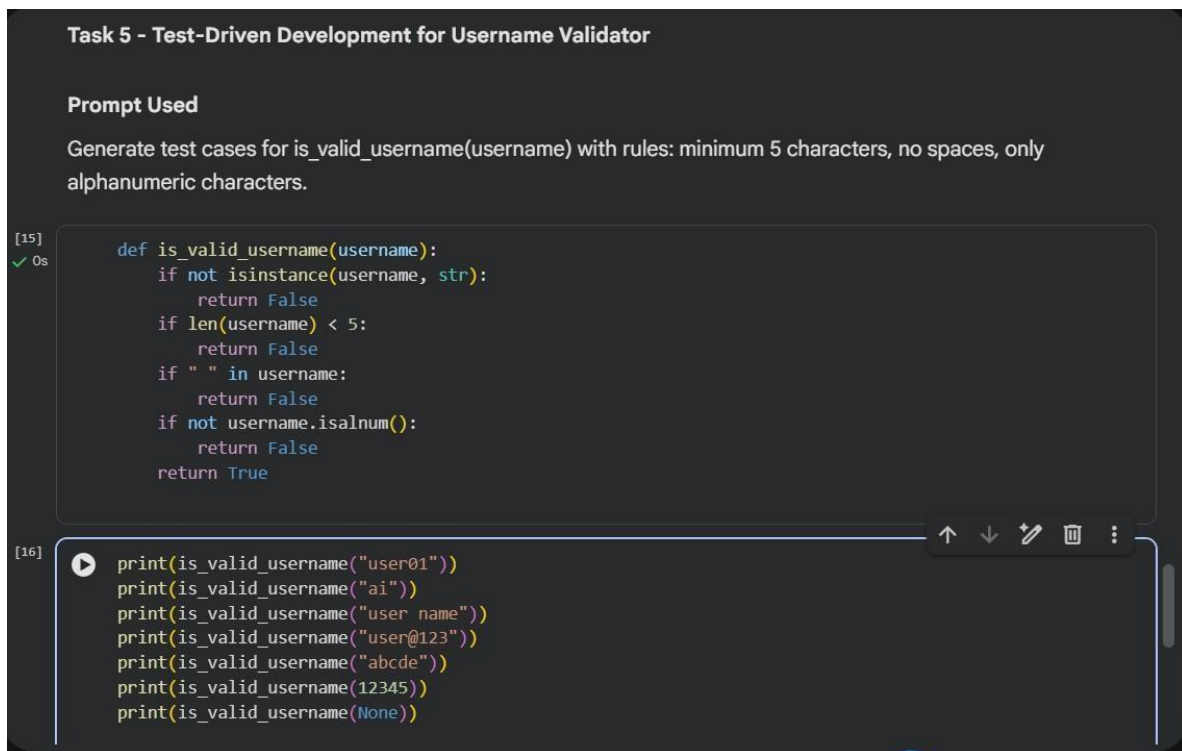
`is_valid_username("user01") → True`

`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.



The screenshot shows a code editor with a dark theme. At the top, the title is "Task 5 - Test-Driven Development for Username Validator". Below it, the "Prompt Used" section contains the text: "Generate test cases for `is_valid_username(username)` with rules: minimum 5 characters, no spaces, only alphanumeric characters." The main code area contains two blocks of Python code. The first block, starting at line 15, defines the `is_valid_username` function. It checks if the input is a string, if its length is at least 5, if it contains no spaces, and if it consists of only alphanumeric characters. The second block, starting at line 16, contains a series of `print` statements that test the function with various inputs: "user01", "ai", "user name", "user@123", "abcde", 12345, and None. The code is syntax-highlighted, and the editor includes standard icons for navigation and editing.

```
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] def is_valid_username(username):
[15]     if not isinstance(username, str):
[15]         return False
[15]     if len(username) < 5:
[15]         return False
[15]     if " " in username:
[15]         return False
[15]     if not username.isalnum():
[15]         return False
[15]     return True

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



```
[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))

... 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.