

# AI Assisted Coding

Week4 -Tuesday

**Name: Lalu prasad Aroori**

**Roll No.: 2303A51948**

**Batch No.: 07**

**Lab 8: Test-Driven Development with AI – Generating and Working with Test Cases**

**Week 4 – Wednesday**

---

## **Task 1 – TDD for Even/Odd Number Validator**

---

### **1) Prompt to be given (Humanized – VS Code AI Tool)**

Generate Python unittest test cases for a function `is_even(n)`.

Requirements:

- Input must be an integer
- Handle zero
- Handle negative numbers
- Handle large integers
- Raise appropriate error for invalid inputs

Do not implement the function yet. Only generate test cases.

---

### **2) Code**

#### **AI-Generated Test Cases**

```
import unittest
```

```
class TestIsEven(unittest.TestCase):

    def test_positive_even(self):

        self.assertTrue(is_even(2))


    def test_positive_odd(self):

        self.assertFalse(is_even(7))


    def test_zero(self):

        self.assertTrue(is_even(0))


    def test_negative_even(self):

        self.assertTrue(is_even(-4))


    def test_large_number(self):

        self.assertTrue(is_even(1000000))


    def test_invalid_input(self):

        with self.assertRaises(TypeError):

            is_even("a")
```

---

### Implementation to Pass Tests

```
def is_even(n):

    if not isinstance(n, int):

        raise TypeError("Input must be an integer")

    return n % 2 == 0
```

---

### 3) Output

.....

-----  
Ran 6 tests in 0.001s

OK

---

#### 4) Remarks

- Tests were written before implementation (TDD).
- Function satisfies all test conditions.
- Handles edge cases properly.
- Input validation improves reliability.

---

#### Task 2 – TDD for String Case Converter

---

##### 1) Prompt

Generate unittest test cases for two functions:

- to\_uppercase(text)
- to\_lowercase(text)

Requirements:

- Handle empty strings
- Handle mixed-case input
- Handle None and invalid inputs

---

##### 2) Code

###### Test Cases

```
class TestStringCase(unittest.TestCase):
```

```
    def test_uppercase(self):
```

```
self.assertEqual(to_uppercase("ai coding"), "AI CODING")
```

```
def test_lowercase(self):
```

```
    self.assertEqual(to_lowercase("TEST"), "test")
```

```
def test_empty(self):
```

```
    self.assertEqual(to_uppercase(""), "")
```

```
    self.assertEqual(to_lowercase(""), "")
```

```
def test_invalid(self):
```

```
    with self.assertRaises(TypeError):
```

```
        to_lowercase(None)
```

---

### ✅ Implementation

```
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()
```

---

### 3) Output

```
....
```

```
-----  
Ran 4 tests in 0.001s
```

OK

---

#### 4) Remarks

- TDD ensures safe input handling.
  - Empty string handled correctly.
  - Raises proper error for invalid input.
  - Clean and maintainable implementation.
- 

### Task 3 – TDD for List Sum Calculator

---

#### 1) Prompt

Generate unittest test cases for a function `sum_list(numbers)`.

Requirements:

- Handle empty list
  - Handle negative numbers
  - Ignore non-numeric values
- 

#### 2) Code

##### Test Cases

```
class TestSumList(unittest.TestCase):
```

```
    def test_normal(self):
```

```
        self.assertEqual(sum_list([1,2,3]), 6)
```

```
    def test_empty(self):
```

```
        self.assertEqual(sum_list([]), 0)
```

```
def test_negative(self):  
    self.assertEqual(sum_list([-1,5,-4]), 0)
```

```
def test_mixed(self):  
    self.assertEqual(sum_list([2,"a",3]), 5)
```

---

### ✓ Implementation

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

---

### 3) Output

....

-----

Ran 4 tests in 0.001s

OK

---

### 4) Remarks

- Function ignores invalid elements safely.
  - Handles empty list properly.
  - Demonstrates robust test-driven design.
- 

### ✓ Task 4 – StudentResult Class (TDD)

---

#### 1) Prompt

Generate unittest test cases for a StudentResult class with:

- add\_marks(mark)
- calculate\_average()
- get\_result()

Requirements:

- Marks between 0 and 100
  - Average  $\geq 40$  -> Pass
  - Otherwise Fail
  - Raise error for invalid marks
- 

## 2) Code

### Test Cases

```
class TestStudentResult(unittest.TestCase):

    def test_pass(self):
        s = StudentResult()
        s.add_marks(60)
        s.add_marks(70)
        s.add_marks(80)
        self.assertEqual(s.calculate_average(), 70)
        self.assertEqual(s.get_result(), "Pass")

    def test_fail(self):
        s = StudentResult()
        s.add_marks(30)
        s.add_marks(35)
        s.add_marks(40)
```

```
self.assertEqual(s.get_result(), "Fail")
```

```
def test_invalid(self):  
    s = StudentResult()  
    with self.assertRaises(ValueError):  
        s.add_marks(-10)
```

---

### ✓ Implementation

```
class StudentResult:  
    def __init__(self):  
        self.marks = []  
  
    def add_marks(self, mark):  
        if not (0 <= mark <= 100):  
            raise ValueError("Marks 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):  
        return "Pass" if self.calculate_average() >= 40 else "Fail"
```

---

### 3) Output

...

-----



Ran 3 tests in 0.001s

OK

---

#### 4) Remarks

- Ensures marks validation.
  - Clear pass/fail logic.
  - Follows OOP principles.
  - AI-generated tests improved correctness.
- 

### Task 5 – Username Validator (TDD)

---

#### 1) Prompt

Generate unittest test cases for a function `is_valid_username(username)`.

Requirements:

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

#### 2) Code

##### Test Cases

```
class TestUsername(unittest.TestCase):
```

```
    def test_valid(self):  
        self.assertTrue(is_valid_username("user01"))
```

```
    def test_short(self):  
        self.assertFalse(is_valid_username("ai"))
```

```
def test_space(self):  
    self.assertFalse(is_valid_username("user name"))
```

```
def test_special_char(self):  
    self.assertFalse(is_valid_username("user@123"))
```

---

### ✓ Implementation

```
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
```

---

### 3) Output

....

-----

Ran 4 tests in 0.001s

OK

---

### 4) Remarks

- Enforces validation rules strictly.
- Handles invalid cases clearly.

- TDD improves security and reliability.
  - Clean and readable logic.
- 

#### **Overall Lab Conclusion**

- Test cases were generated first (TDD approach).
- Implementation written to satisfy tests.
- Edge cases handled properly.
- AI-assisted TDD improves code reliability and validation quality.
- Writing tests first ensures structured development.