

Name: Ch Pushkar

HT No: 2303A52362

Lab 8: Test-Driven Development with AI

Course: AI Assisted Coding (23CS002PC304)

Assignment: 8.2

Topic: Generating and Working with Test Cases using Pytest

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

Task Description

The objective is to implement a function `is_even(n)` using a Test-Driven Development (TDD) approach with the pytest framework. We must identify edge cases (zero, negative numbers) and verify them using assertions.

Prompt

"I need to practice TDD using pytest. Generate a Python function `is_even(n)` and a corresponding set of pytest test functions. Ensure you test positive integers, negative integers, zero, and invalid inputs."

Code

```

def is_even(n):
    """
    Returns True if n is even, False otherwise.
    Validates that input is an integer.
    """
    if not isinstance(n, int):
        return "Invalid Input"
    return n % 2 == 0

# --- Pytest Test Cases ---
def test_positive_numbers():
    assert is_even(2) is True
    assert is_even(9) is False

def test_edge_cases():
    assert is_even(0) is True    # Zero is even
    assert is_even(-4) is True  # Negative even number

def test_invalid_input():
    assert is_even("string") == "Invalid Input"
    assert is_even(3.5) == "Invalid Input"

```

Output

```

PS C:\Users\chirr\Projects\AI Assistant Coding> pytest .\assign-8.py
=====
test session starts =====
platform win32 -- Python 3.14.2, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\chirr\Projects\AI Assistant Coding
collected 3 items

assign-8.py ...

```

Explanation

Using pytest makes TDD cleaner. Instead of manually printing "Input | Result", we define functions starting with `test_`. Inside these functions, we use the `assert` statement. If `is_even(2)` returns `True`, the test passes silently. If it returns `False`, pytest would shout with a detailed

error message. I separated tests into logical groups (positive, edge cases, invalid) to make debugging easier.

Task 2: Test-Driven Development for String Case Converter

Task Description

This task requires creating two functions, `to_uppercase` and `to_lowercase`. We will use pytest to handle "dirty" data—such as `None` values, empty strings, or numbers—ensuring the program doesn't crash.

Prompt

"Generate pytest test cases for string conversion functions `to_uppercase` and `to_lowercase`. Use assertions to verify they handle `None`, numbers, and empty strings safely without crashing."

Code

```

def to_uppercase(text):
    if text is None or not isinstance(text, str):
        return "Invalid Input"
    return text.upper()

def to_lowercase(text):
    if text is None or not isinstance(text, str):
        return "Invalid Input"
    return text.lower()

# --- Pytest Test Cases ---
def test_string_conversion():
    assert to_uppercase("ai coding") == "AI CODING"
    assert to_lowercase("TEST") == "test"

def test_empty_strings():
    assert to_uppercase("") == ""
    assert to_lowercase("") == ""

def test_invalid_types():
    assert to_lowercase(None) == "Invalid Input"
    assert to_uppercase(123) == "Invalid Input"

```

Output

```

PS C:\Users\chirr\Projects\AI Assistant Coding> pytest .\assign-8.py
=====
 test session starts =====
platform win32 -- Python 3.14.2, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\chirr\Projects\AI Assistant Coding
collected 13 items

assign-8.py ......

=====
 13 passed in 0.04s =====

```

Explanation

Here, the tests act as a "contract" for the code. The test `test_invalid_types` specifically asserts that passing `None` or `123` must return `"Invalid Input"` rather than raising an exception. This enforces robustness. If I had written the function to just return `text.upper()` without checking the type first, pytest would have failed this test immediately, signaling that my code was unsafe.

Task 3: Test-Driven Development for List Sum Calculator

Task Description

We need to write a function `sum_list(numbers)` that calculates the total of a list. We will use pytest to verify that the function safely ignores non-numeric values.

Prompt

"Write a Python function `sum_list` and use pytest to verify it. The tests should check that strings inside the list are ignored and that empty lists return 0."

Code

```

def sum_list(numbers):
    total = 0
    for item in numbers:
        # Check if item is an integer or float
        if isinstance(item, (int, float)):
            total += item
    return total

# --- Pytest Test Cases ---
def test_standard_integers():
    assert sum_list([1, 2, 3]) == 6

def test_empty_list():
    assert sum_list([]) == 0

def test_mixed_types():
    # Should ignore "a" and sum 2 + 3
    assert sum_list([2, "a", 3]) == 5

def test_negative_numbers():
    assert sum_list([-1, 5, -4]) == 0

```

Output

```

PS C:\Users\chirr\Projects\AI Assistant Coding> pytest .\assign-8.py
=====
test session starts =====
platform win32 -- Python 3.14.2, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\chirr\Projects\AI Assistant Coding
collected 4 items

assign-8.py ....
=====
4 passed in 0.03s =====

```

Explanation

The logic for `sum_list` is verified by four distinct tests. `test_mixed_types` is the most interesting one; it asserts that `[2, "a", 3]` equals 5. This confirms that our logic successfully filters out the

string "a". If we used the built-in sum() function, this specific test would fail with a TypeError, guiding us to implement the manual loop with isinstance checks.

Task 4: Test Cases for Student Result Class

Task Description

This task involves Object-Oriented TDD. We will use pytest **fixtures** to create a fresh StudentResult instance for each test, ensuring clean state.

Prompt

"Create a StudentResult class. Use pytest fixtures to instantiate the class and write tests for adding marks, calculating averages, and determining Pass/Fail results."

Code

```
class StudentResult:
    def __init__(self):
        self.marks = []

    def add_marks(self, mark):
        if mark < 0 or mark > 100:
            raise ValueError("Mark must be 0-100")
        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):
        avg = self.calculate_average()
        return "Pass" if avg >= 40 else "Fail"
```

```

# --- Pytest Fixtures & Tests ---
@pytest.fixture
def student():
    return StudentResult()

def test_pass_scenario(student):
    for m in [60, 70, 80]:
        student.add_marks(m)
    assert student.calculate_average() == 70.0
    assert student.get_result() == "Pass"

def test_fail_scenario(student):
    for m in [30, 35, 40]:
        student.add_marks(m)
    assert student.calculate_average() == 35.0
    assert student.get_result() == "Fail"

def test_invalid_mark_raises_error(student):
    # Verify that invalid input raises an exception
    with pytest.raises(ValueError):
        student.add_marks(-10)

```

Output

```

PS C:\Users\chirr\Projects\AI Assistant Coding> pytest .\assign-8.py
=====
 test session starts =====
platform win32 -- Python 3.14.2, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\chirr\Projects\AI Assistant Coding
collected 3 items

assign-8.py ...

=====
 3 passed in 0.02s =====

```

Explanation

In OOP testing, state is everything. If we reused the same student object for all tests, marks

from the first test might mess up the second test. I used the `@pytest.fixture` decorator to create a fresh student object automatically for every test function. I also used `pytest.raises(ValueError)`, which is a professional way to test that code *correctly* errors out when given bad input (like -10).

Task 5: Test-Driven Development for Username Validator

Task Description

The final task uses `pytest.mark.parametrize` to run the same test logic on multiple different inputs (valid and invalid usernames) efficiently.

Prompt

"Write a username validator function. Use `pytest.mark.parametrize` to test valid inputs ('user01') against invalid ones ('ai', 'user name', 'user@123')."

Code

```

import pytest

def is_valid_username(username):
    # Rule 1: Length >= 5
    if len(username) < 5:
        return False
    # Rule 2: Alphanumeric only (no spaces, no symbols)
    if not username.isalnum():
        return False
    return True

# --- Pytest Parametrized Testing ---
@pytest.mark.parametrize("username, expected_result", [
    ("user01", True),           # Valid
    ("ai", False),             # Too short
    ("user name", False),      # Contains space
    ("user@123", False),       # Contains symbol
    ("admin123", True)         # Valid
])
def test_username_validation(username, expected_result):
    assert is_valid_username(username) == expected_result

```

Output

```

PS C:\Users\chirr\Projects\AI Assistant Coding> pytest -q assign-8.py
.....
5 passed in 0.02s

```

Explanation

Instead of writing five separate test functions, I used `@pytest.mark.parametrize`. This allows us to feed a list of inputs and expected outputs into a single test function. It makes the code much cleaner and easier to expand. If we wanted to test a new case, like "super_user", we would just add one line to the list, and pytest would automatically generate a new test case for it.