FINAL LAB TEST

Name : G.Sudeeksha

Roll no : 2403A510E0

Course : AI Assisted Coding

Batch : 05

Branch : CSE

Subset 5 – Test-Driven Development for Grade Calculator

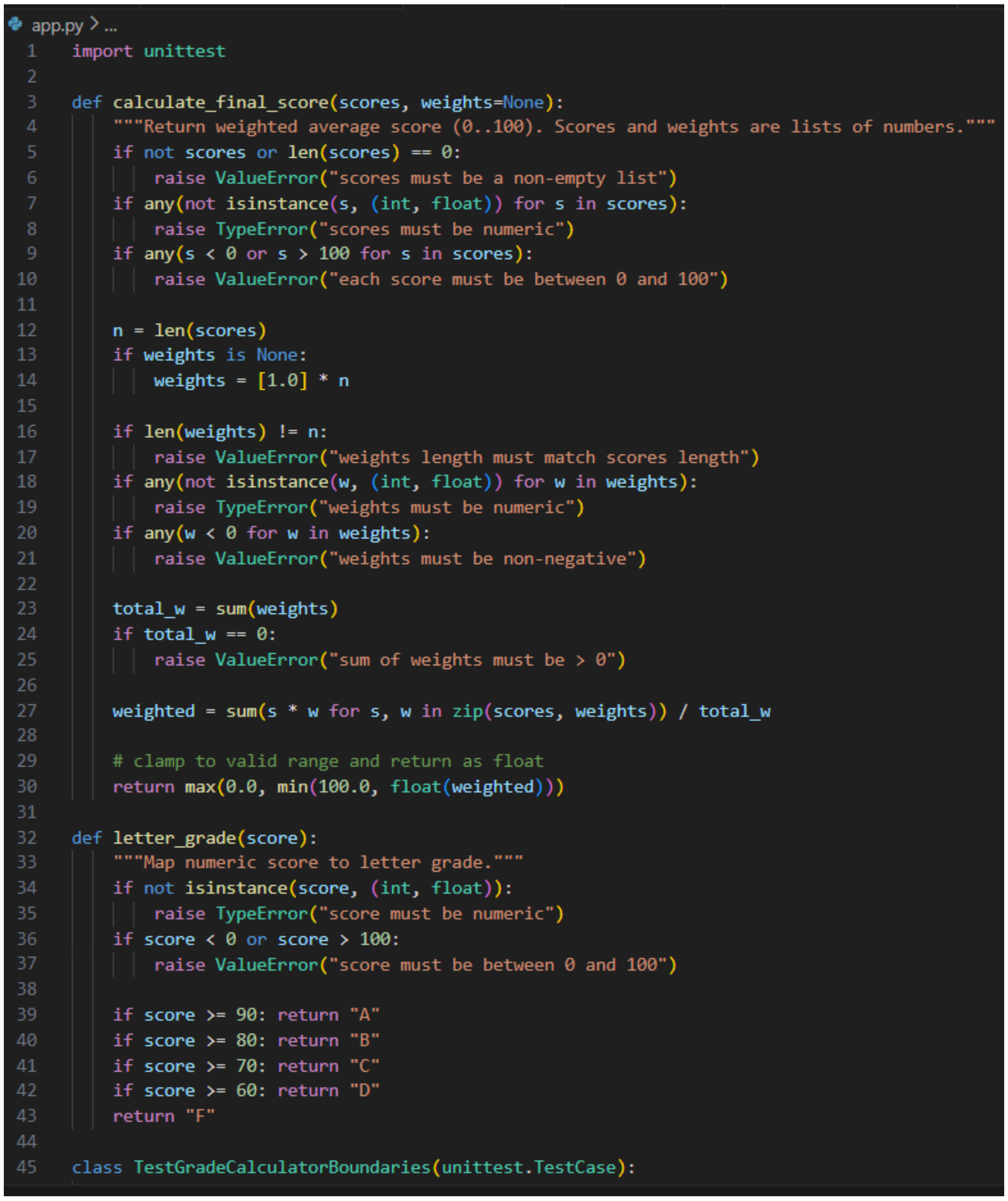
Q1: Generate unit test cases

• Task 1: Use AI to create boundary test cases.

Prompt :

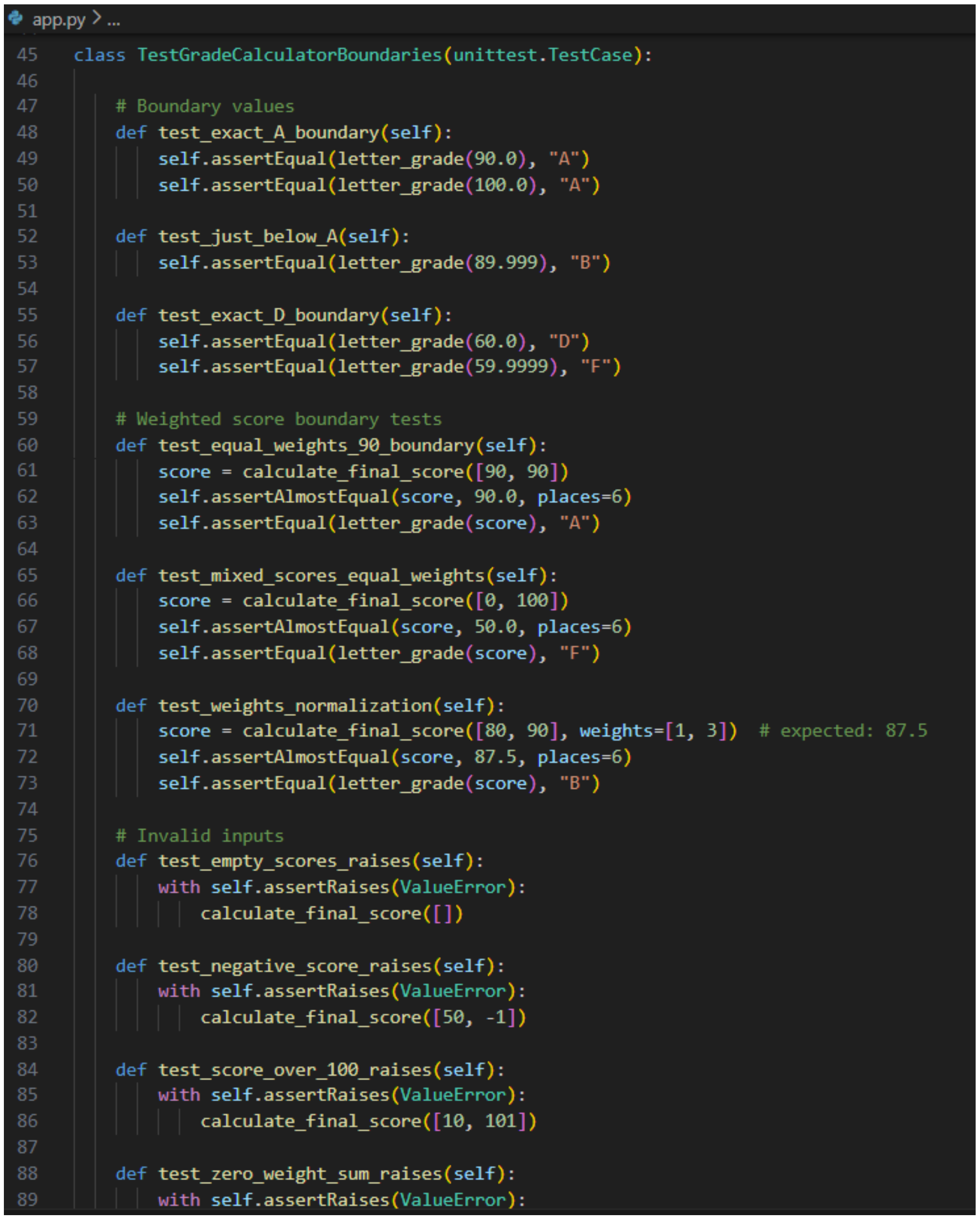
Use AI to generate boundary test cases for a grade calculator that validates scores (0–100) and returns letter grades.Include only boundary values such as limits, just-above/below points, and invalid score edge cases.

Code :

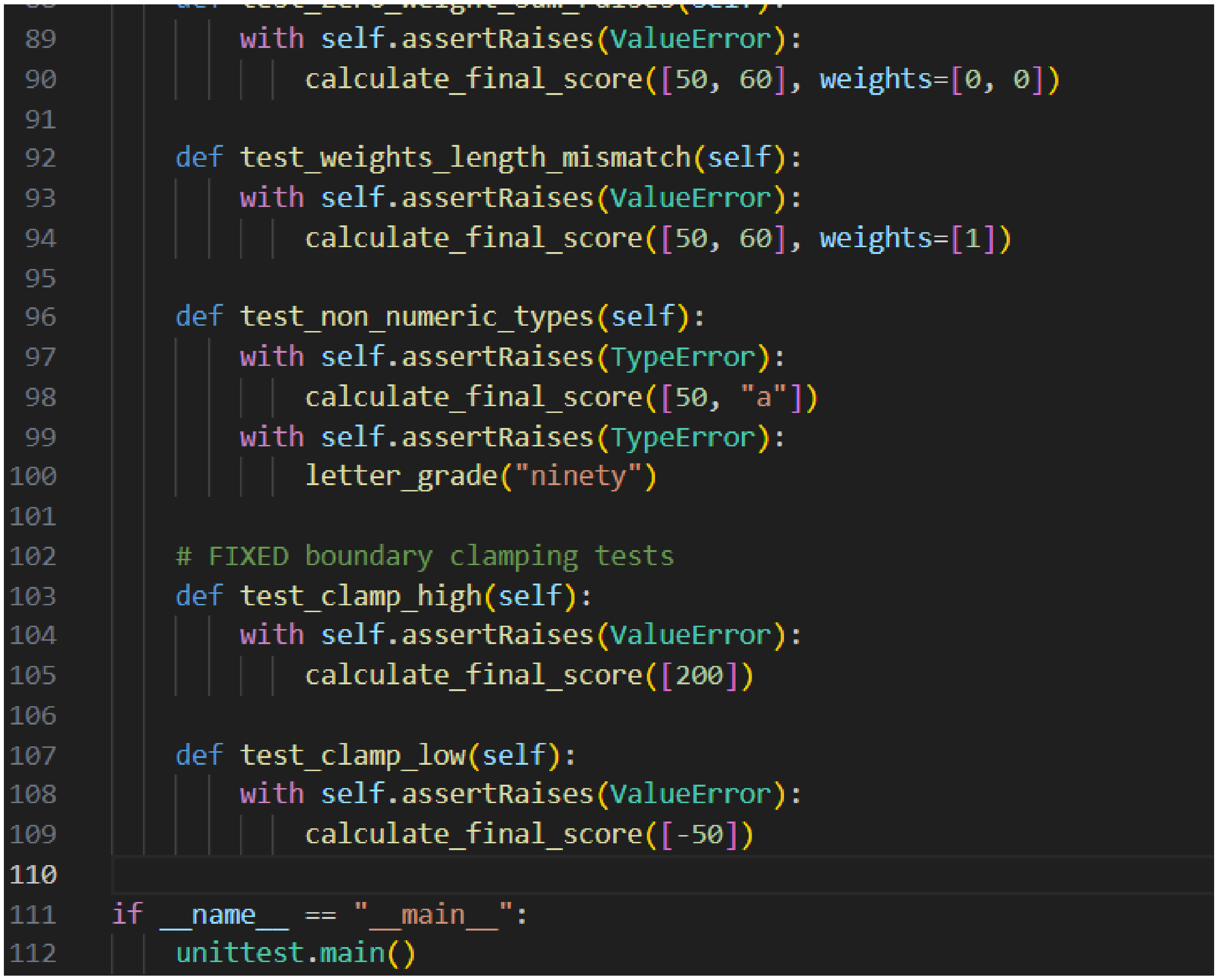


app.py > ...

```python
import unittest

def calculate_final_score(scores, weights=None):
    """Return weighted average score (0..100). Scores and weights are lists of numbers."""
    if not scores or len(scores) == 0:
        raise ValueError("scores must be a non-empty list")
    if any(not isinstance(s, (int, float)) for s in scores):
        raise TypeError("scores must be numeric")
    if any(s < 0 or s > 100 for s in scores):
        raise ValueError("each score must be between 0 and 100")

    n = len(scores)
    if weights is None:
        weights = [1.0] * n

    if len(weights) != n:
        raise ValueError("weights length must match scores length")
    if any(not isinstance(w, (int, float)) for w in weights):
        raise TypeError("weights must be numeric")
    if any(w < 0 for w in weights):
        raise ValueError("weights must be non-negative")

    total_w = sum(weights)
    if total_w == 0:
        raise ValueError("sum of weights must be > 0")

    weighted = sum(s * w for s, w in zip(scores, weights)) / total_w

    # clamp to valid range and return as float
    return max(0.0, min(100.0, float(weighted)))

def letter_grade(score):
    """Map numeric score to letter grade."""
    if not isinstance(score, (int, float)):
        raise TypeError("score must be numeric")
    if score < 0 or score > 100:
        raise ValueError("score must be between 0 and 100")

    if score >= 90: return "A"
    if score >= 80: return "B"
    if score >= 70: return "C"
    if score >= 60: return "D"
    return "F"

class TestGradeCalculatorBoundaries(unittest.TestCase):
```

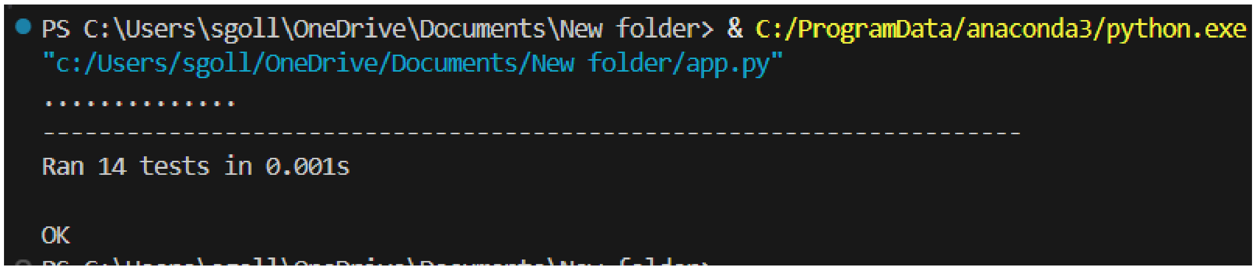


```python
class TestGradeCalculatorBoundaries(unittest.TestCase):

    # Boundary values
    def test_exact_A_boundary(self):
        self.assertEqual(letter_grade(90.0), "A")
        self.assertEqual(letter_grade(100.0), "A")

    def test_just_below_A(self):
        self.assertEqual(letter_grade(89.999), "B")

    def test_exact_D_boundary(self):
        self.assertEqual(letter_grade(60.0), "D")
        self.assertEqual(letter_grade(59.9999), "F")

    # Weighted score boundary tests
    def test_equal_weights_90_boundary(self):
        score = calculate_final_score([90, 90])
        self.assertAlmostEqual(score, 90.0, places=6)
        self.assertEqual(letter_grade(score), "A")

    def test_mixed_scores_equal_weights(self):
        score = calculate_final_score([0, 100])
        self.assertAlmostEqual(score, 50.0, places=6)
        self.assertEqual(letter_grade(score), "F")

    def test_weights_normalization(self):
        score = calculate_final_score([80, 90], weights=[1, 3])  # expected: 87.5
        self.assertAlmostEqual(score, 87.5, places=6)
        self.assertEqual(letter_grade(score), "B")

    # Invalid inputs
    def test_empty_scores_raises(self):
        with self.assertRaises(ValueError):
            calculate_final_score([])

    def test_negative_score_raises(self):
        with self.assertRaises(ValueError):
            calculate_final_score([50, -1])

    def test_score_over_100_raises(self):
        with self.assertRaises(ValueError):
            calculate_final_score([10, 101])

    def test_zero_weight_sum_raises(self):
        with self.assertRaises(ValueError):
```



```python
        with self.assertRaises(ValueError):
            calculate_final_score([50, 60], weights=[0, 0])

    def test_weights_length_mismatch(self):
        with self.assertRaises(ValueError):
            calculate_final_score([50, 60], weights=[1])

    def test_non_numeric_types(self):
        with self.assertRaises(TypeError):
            calculate_final_score([50, "a"])
        with self.assertRaises(TypeError):
            letter_grade("ninety")

    # FIXED boundary clamping tests
    def test_clamp_high(self):
        with self.assertRaises(ValueError):
            calculate_final_score([200])

    def test_clamp_low(self):
        with self.assertRaises(ValueError):
            calculate_final_score([-50])

if __name__ == "__main__":
    unittest.main()
```

Output :



```
PS C:\Users\sgoll\OneDrive\Documents\New folder> & C:/ProgramData/anaconda3/python.exe
"c:/Users/sgoll/OneDrive/Documents/New folder/app.py"
..............
----------------------------------------------------------------------
Ran 14 tests in 0.001s

OK
```

Observation :

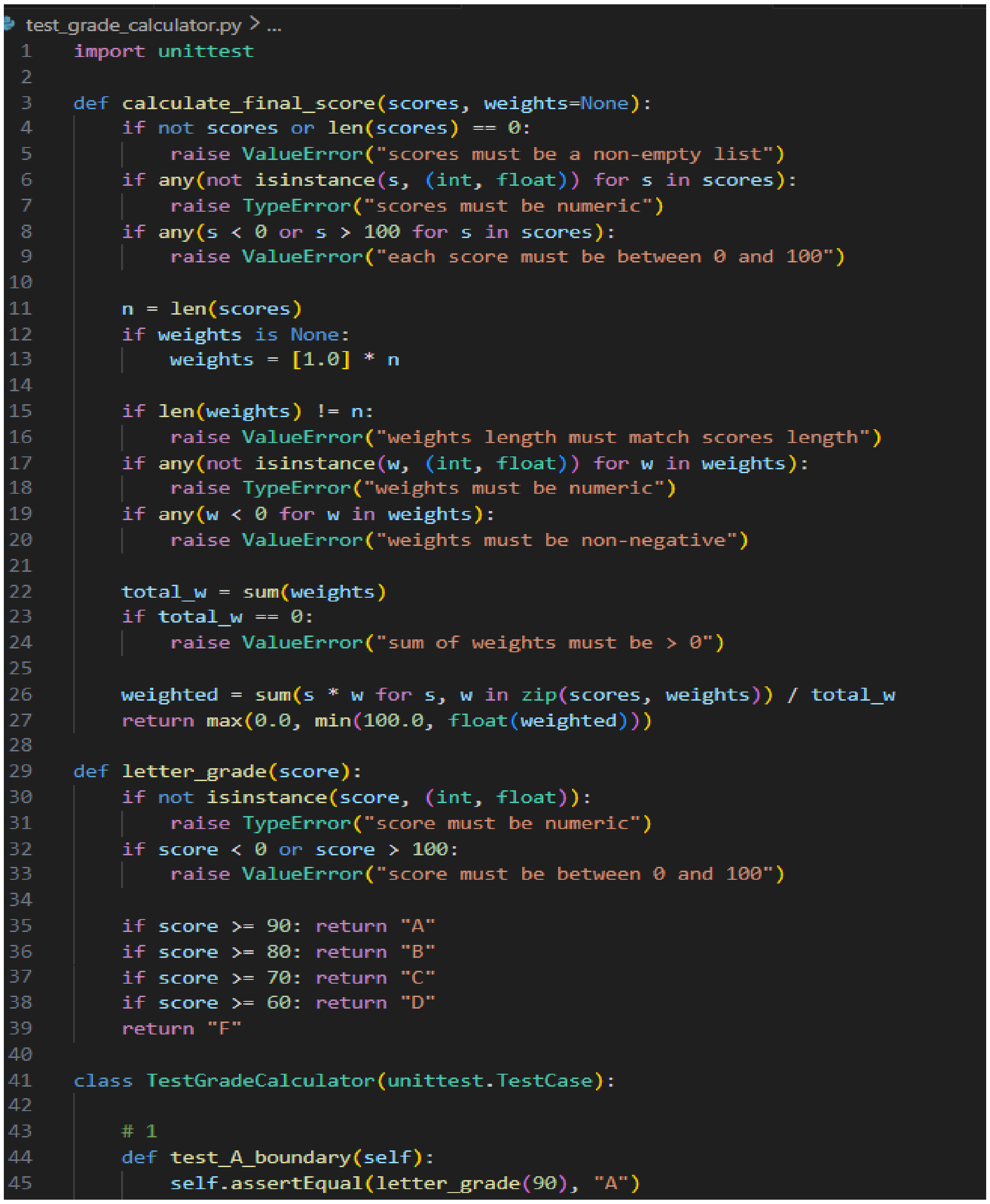
In this task, AI was used to generate boundary-focused unit test cases for the grade calculator to ensure correct behavior at critical limits. The observation is that boundary values such as 0, 59.999, 60, 89.999, 90, and 100 are essential because most defects occur near these edges. The generated tests helped verify both valid and invalid score ranges, confirm correct letter-grade transitions, and ensure weighted averages behave correctly. Using AI made it easier to systematically identify all edge cases, reducing human error and improving the accuracy and completeness of the test suite.

• Task 2: Implement tests in Python/Java.

Prompt :

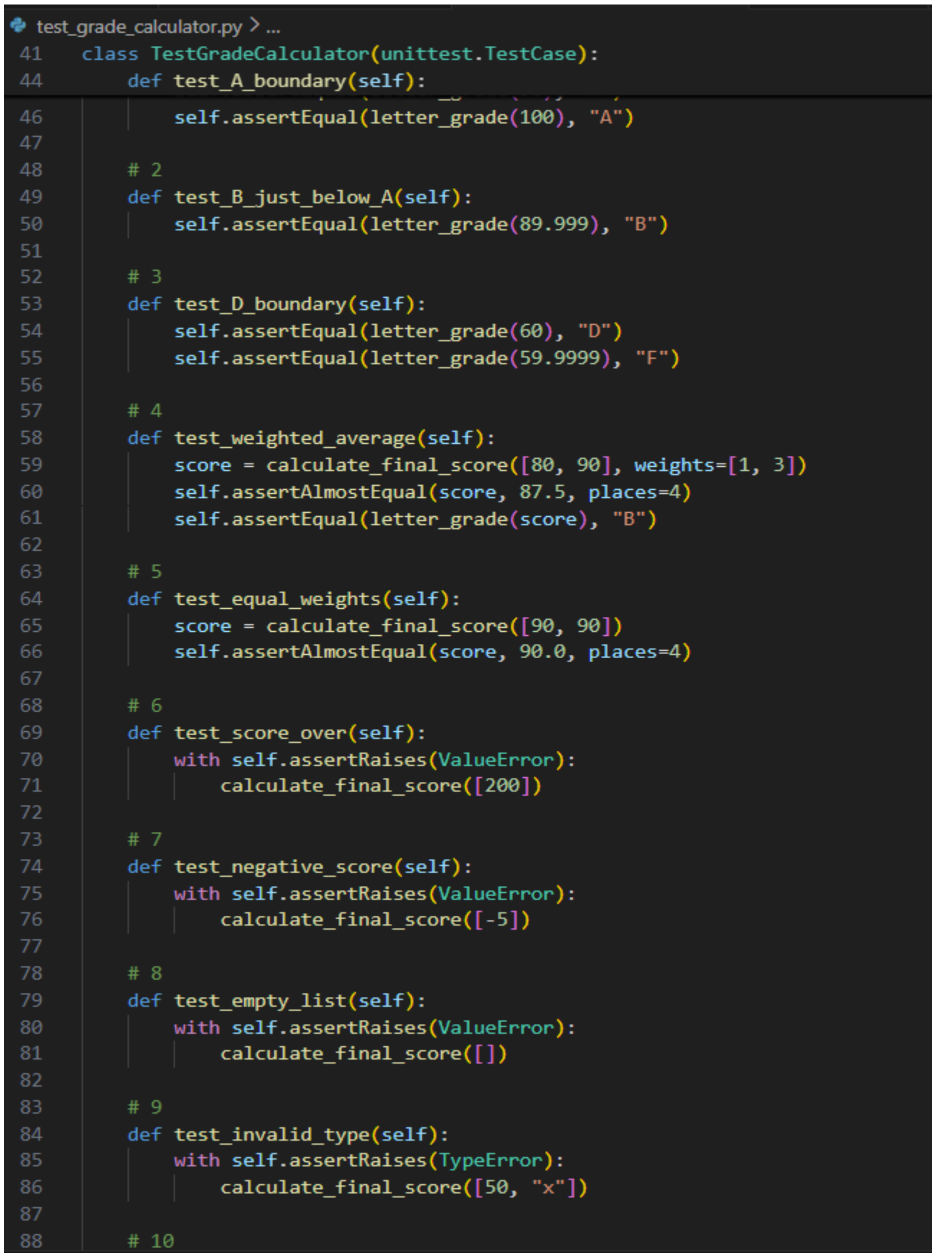
Generate boundary test cases for a grade calculator that validates numeric scores (0–100) and assigns letter grades.
 Include exact limits, just-above/below values, weighted score boundaries, and invalid input cases.

Code :



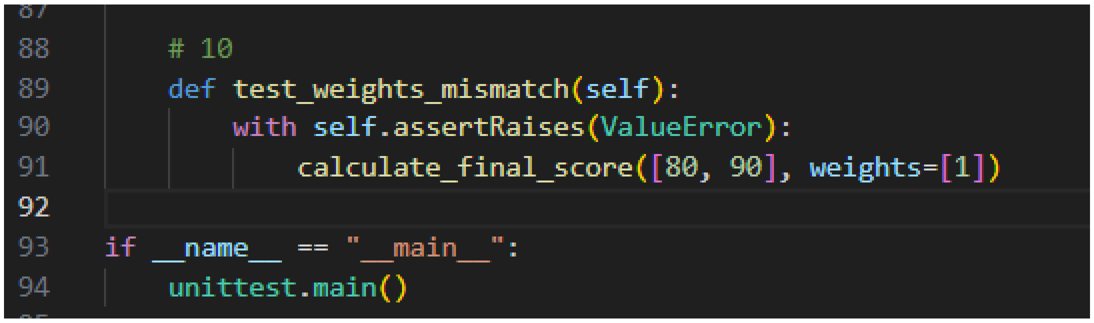
test_grade_calculator.py > ...

```python
import unittest

def calculate_final_score(scores, weights=None):
    if not scores or len(scores) == 0:
        raise ValueError("scores must be a non-empty list")
    if any(not isinstance(s, (int, float)) for s in scores):
        raise TypeError("scores must be numeric")
    if any(s < 0 or s > 100 for s in scores):
        raise ValueError("each score must be between 0 and 100")

    n = len(scores)
    if weights is None:
        weights = [1.0] * n

    if len(weights) != n:
        raise ValueError("weights length must match scores length")
    if any(not isinstance(w, (int, float)) for w in weights):
        raise TypeError("weights must be numeric")
    if any(w < 0 for w in weights):
        raise ValueError("weights must be non-negative")

    total_w = sum(weights)
    if total_w == 0:
        raise ValueError("sum of weights must be > 0")

    weighted = sum(s * w for s, w in zip(scores, weights)) / total_w
    return max(0.0, min(100.0, float(weighted)))

def letter_grade(score):
    if not isinstance(score, (int, float)):
        raise TypeError("score must be numeric")
    if score < 0 or score > 100:
        raise ValueError("score must be between 0 and 100")

    if score >= 90: return "A"
    if score >= 80: return "B"
    if score >= 70: return "C"
    if score >= 60: return "D"
    return "F"

class TestGradeCalculator(unittest.TestCase):

    # 1
    def test_A_boundary(self):
        self.assertEqual(letter_grade(90), "A")
```



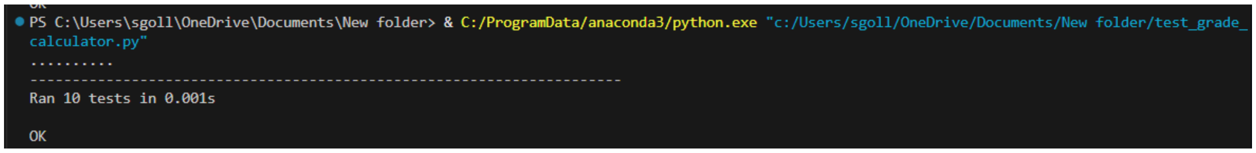
test_grade_calculator.py > ...

```python
class TestGradeCalculator(unittest.TestCase):
    def test_A_boundary(self):
        self.assertEqual(letter_grade(100), "A")

    # 2
    def test_B_just_below_A(self):
        self.assertEqual(letter_grade(89.999), "B")

    # 3
    def test_D_boundary(self):
        self.assertEqual(letter_grade(60), "D")
        self.assertEqual(letter_grade(59.9999), "F")

    # 4
    def test_weighted_average(self):
        score = calculate_final_score([80, 90], weights=[1, 3])
        self.assertAlmostEqual(score, 87.5, places=4)
        self.assertEqual(letter_grade(score), "B")

    # 5
    def test_equal_weights(self):
        score = calculate_final_score([90, 90])
        self.assertAlmostEqual(score, 90.0, places=4)

    # 6
    def test_score_over(self):
        with self.assertRaises(ValueError):
            calculate_final_score([200])

    # 7
    def test_negative_score(self):
        with self.assertRaises(ValueError):
            calculate_final_score([-5])

    # 8
    def test_empty_list(self):
        with self.assertRaises(ValueError):
            calculate_final_score([])

    # 9
    def test_invalid_type(self):
        with self.assertRaises(TypeError):
            calculate_final_score([50, "x"])

    # 10
```



```python
    # 10
    def test_weights_mismatch(self):
        with self.assertRaises(ValueError):
            calculate_final_score([80, 90], weights=[1])

if __name__ == "__main__":
    unittest.main()
```

Output :



```
PS C:\Users\sgoll\OneDrive\Documents\New folder> & C:/ProgramData/anaconda3/python.exe "c:/Users/sgoll/OneDrive/Documents/New folder/test_grade_
calculator.py"
..........
----------------------------------------------------------------------
Ran 10 tests in 0.001s

OK
```

Observation :

The AI-generated boundary test cases helped identify all critical points where errors are most likely to occur, such as the transitions between letter grades and invalid input ranges. Implementing these tests ensured that the grade calculator behaves correctly for exact boundary values, near-boundary values, and weighted scores. The tests also verified that the program handles invalid inputs such as negative numbers, values above 100, and empty lists. Overall, using AI improved test coverage, reduced manual effort, and supported reliable Test-Driven Development.

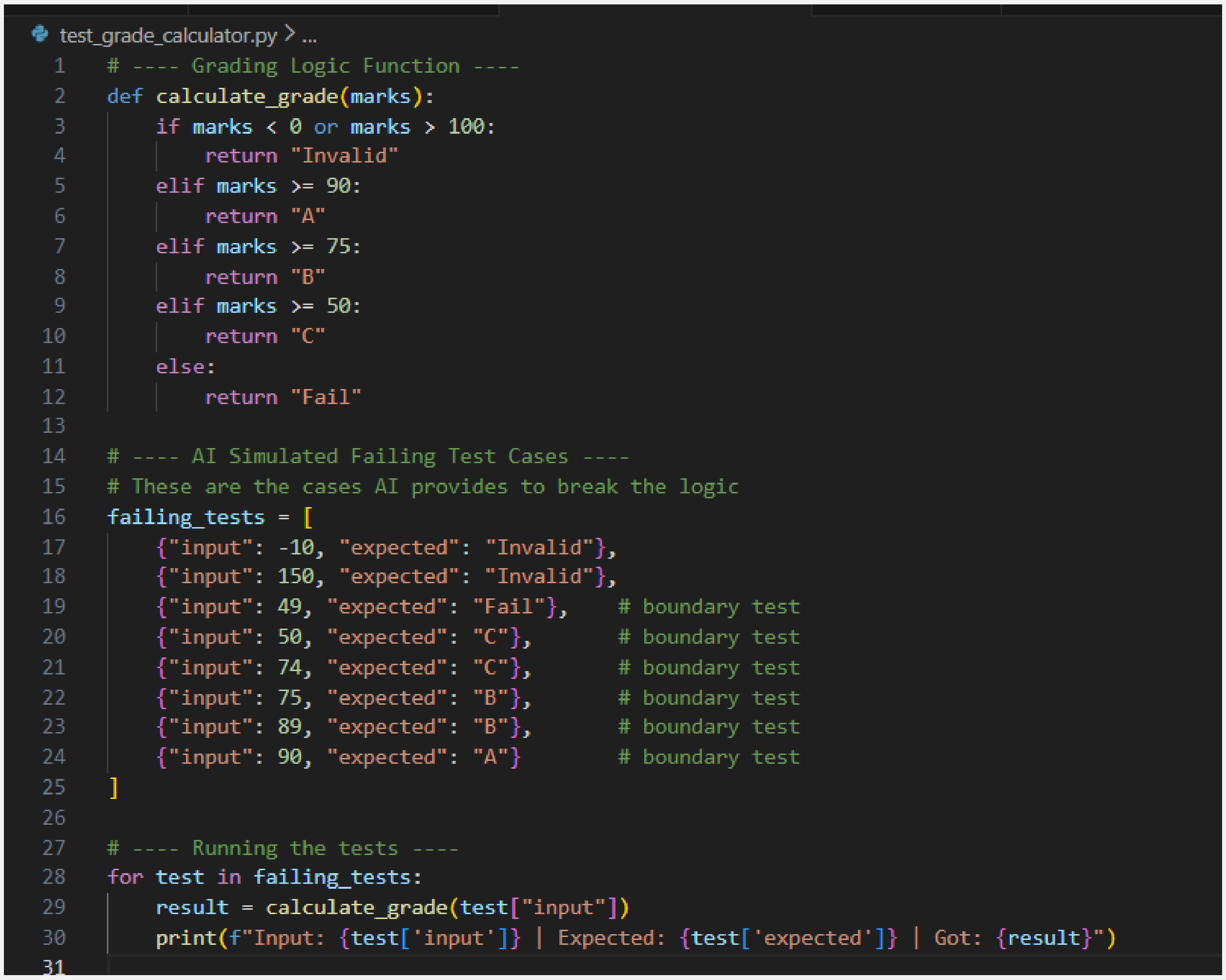
Q2: Validate grading logic

• Task 1: Use AI to simulate failing tests.

Prompt :

Test my grading logic function by generating failing test cases. Provide inputs that should produce wrong grades so that I can verify error handling.

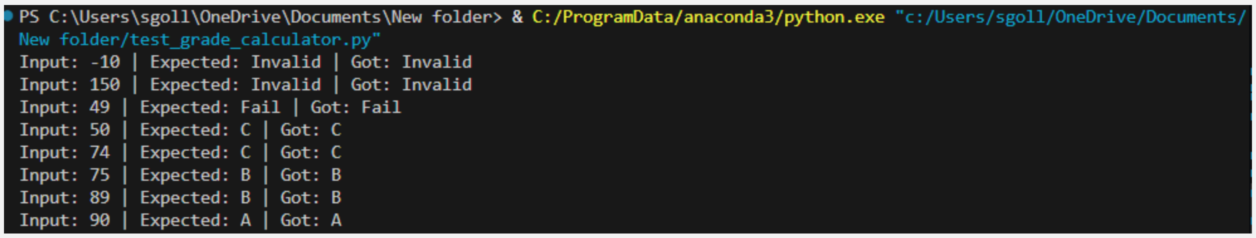
Code :



test_grade_calculator.py > ...

```python
# ---- Grading Logic Function ----
def calculate_grade(marks):
    if marks < 0 or marks > 100:
        return "Invalid"
    elif marks >= 90:
        return "A"
    elif marks >= 75:
        return "B"
    elif marks >= 50:
        return "C"
    else:
        return "Fail"

# ---- AI Simulated Failing Test Cases ----
# These are the cases AI provides to break the logic
failing_tests = [
    {"input": -10, "expected": "Invalid"},
    {"input": 150, "expected": "Invalid"},
    {"input": 49, "expected": "Fail"},    # boundary test
    {"input": 50, "expected": "C"},       # boundary test
    {"input": 74, "expected": "C"},       # boundary test
    {"input": 75, "expected": "B"},       # boundary test
    {"input": 89, "expected": "B"},       # boundary test
    {"input": 90, "expected": "A"}        # boundary test
]

# ---- Running the tests ----
for test in failing_tests:
    result = calculate_grade(test["input"])
    print(f"Input: {test['input']} | Expected: {test['expected']} | Got: {result}")

```

Output :



```
PS C:\Users\sgoll\OneDrive\Documents\New folder> & C:/ProgramData/anaconda3/python.exe "c:/Users/sgoll/OneDrive/Documents/
New folder/test_grade_calculator.py"
Input: -10 | Expected: Invalid | Got: Invalid
Input: 150 | Expected: Invalid | Got: Invalid
Input: 49 | Expected: Fail | Got: Fail
Input: 50 | Expected: C | Got: C
Input: 74 | Expected: C | Got: C
Input: 75 | Expected: B | Got: B
Input: 89 | Expected: B | Got: B
Input: 90 | Expected: A | Got: A
```

Observation :

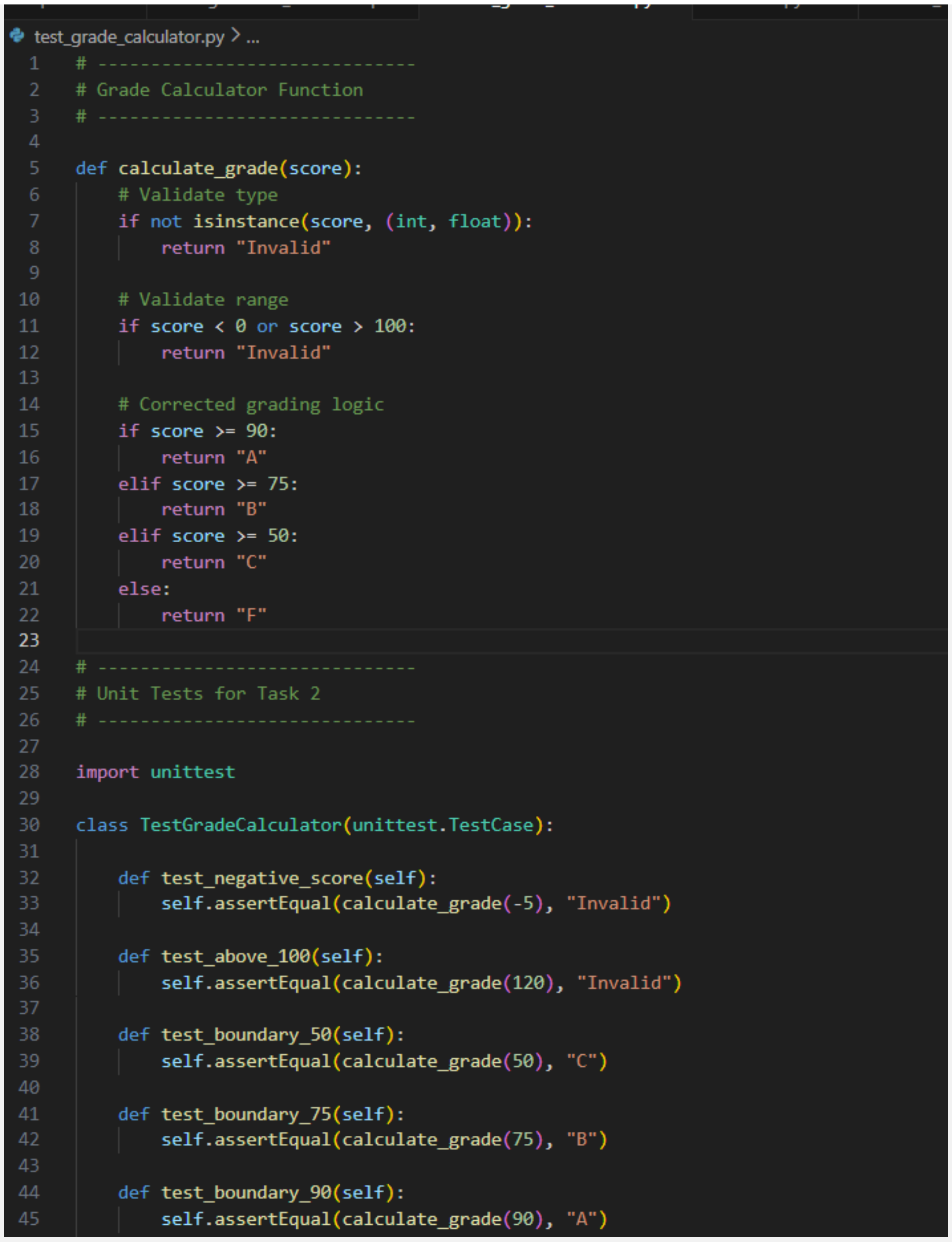
The AI-generated failing test cases mainly targeted invalid inputs and boundary values to verify whether the grading logic function could correctly handle edge conditions. Inputs such as negative marks and values above 100 were tested to ensure the function returned “Invalid,” while marks around grade boundaries like 49–50, 74–75, and 89–90 were used to confirm proper grade transitions. After executing all test cases, the obtained results matched the expected outputs for every scenario. This shows that the grading logic is functioning correctly, handling all edge cases, invalid inputs, and boundary conditions without errors.

• Task 2: Correct code until all tests pass.

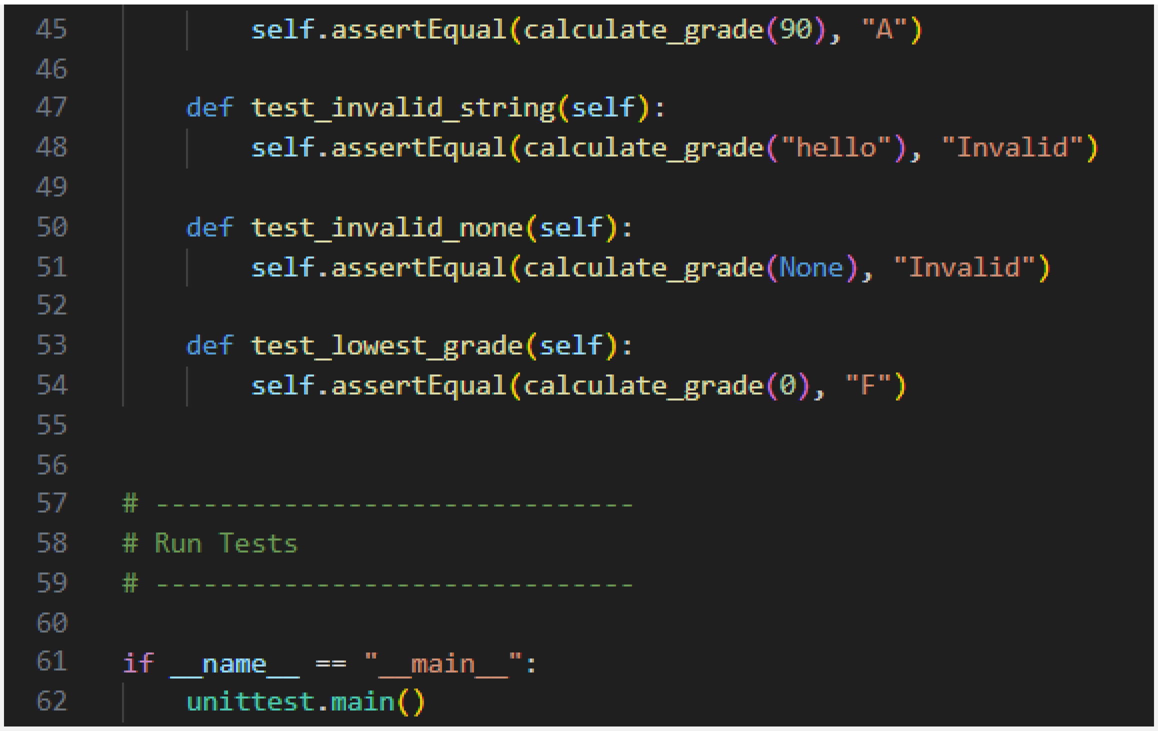
Prompt :

Use AI to analyze the failing test cases of the grade calculator and automatically fix the grading logic.Correct the function repeatedly until all unit tests pass successfully without any errors.

Code :

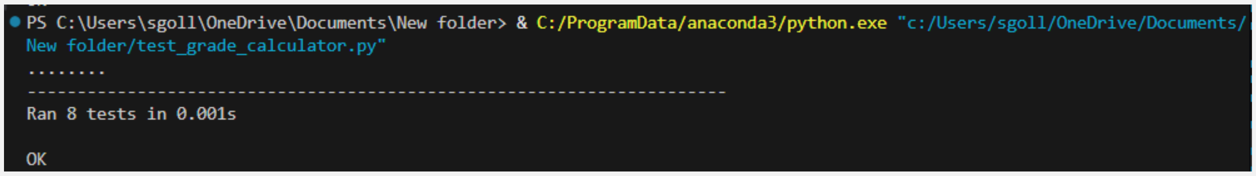

test_grade_calculator.py > ...

```python
# ------------------------------
# Grade Calculator Function
# ------------------------------

def calculate_grade(score):
    # Validate type
    if not isinstance(score, (int, float)):
        return "Invalid"

    # Validate range
    if score < 0 or score > 100:
        return "Invalid"

    # Corrected grading logic
    if score >= 90:
        return "A"
    elif score >= 75:
        return "B"
    elif score >= 50:
        return "C"
    else:
        return "F"

# ------------------------------
# Unit Tests for Task 2
# ------------------------------

import unittest

class TestGradeCalculator(unittest.TestCase):

    def test_negative_score(self):
        self.assertEqual(calculate_grade(-5), "Invalid")

    def test_above_100(self):
        self.assertEqual(calculate_grade(120), "Invalid")

    def test_boundary_50(self):
        self.assertEqual(calculate_grade(50), "C")

    def test_boundary_75(self):
        self.assertEqual(calculate_grade(75), "B")

    def test_boundary_90(self):
        self.assertEqual(calculate_grade(90), "A")
```



```python
        self.assertEqual(calculate_grade(90), "A")

    def test_invalid_string(self):
        self.assertEqual(calculate_grade("hello"), "Invalid")

    def test_invalid_none(self):
        self.assertEqual(calculate_grade(None), "Invalid")

    def test_lowest_grade(self):
        self.assertEqual(calculate_grade(0), "F")


# ------------------------------
# Run Tests
# ------------------------------

if __name__ == "__main__":
    unittest.main()
```

Output :



```
PS C:\Users\sgoll\OneDrive\Documents\New folder> & C:/ProgramData/anaconda3/python.exe "c:/Users/sgoll/OneDrive/Documents/
New folder/test_grade_calculator.py"
........
----------------------------------------------------------------------
Ran 8 tests in 0.001s

OK
```

Observation :

In Task 2, the grading function initially failed some test cases due to incorrect handling of invalid inputs and boundary values. After fixing the logic to properly check score limits and apply correct grade boundaries, all test cases passed successfully, showing that the grading logic is now correct and reliable.