

AI ASSISTED CODING

2303A52454

B-34

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

Aim

To apply Test-Driven Development (TDD) using AI by generating test cases first and then implementing Python code that satisfies those tests.

Task Description Task 1 – Test-Driven Development for Even/Odd Number Validator Prompt

Generate unittest test cases for a Python function `is_even(n)` that checks whether a number is even. Handle integers, zero, negative values, and invalid inputs.

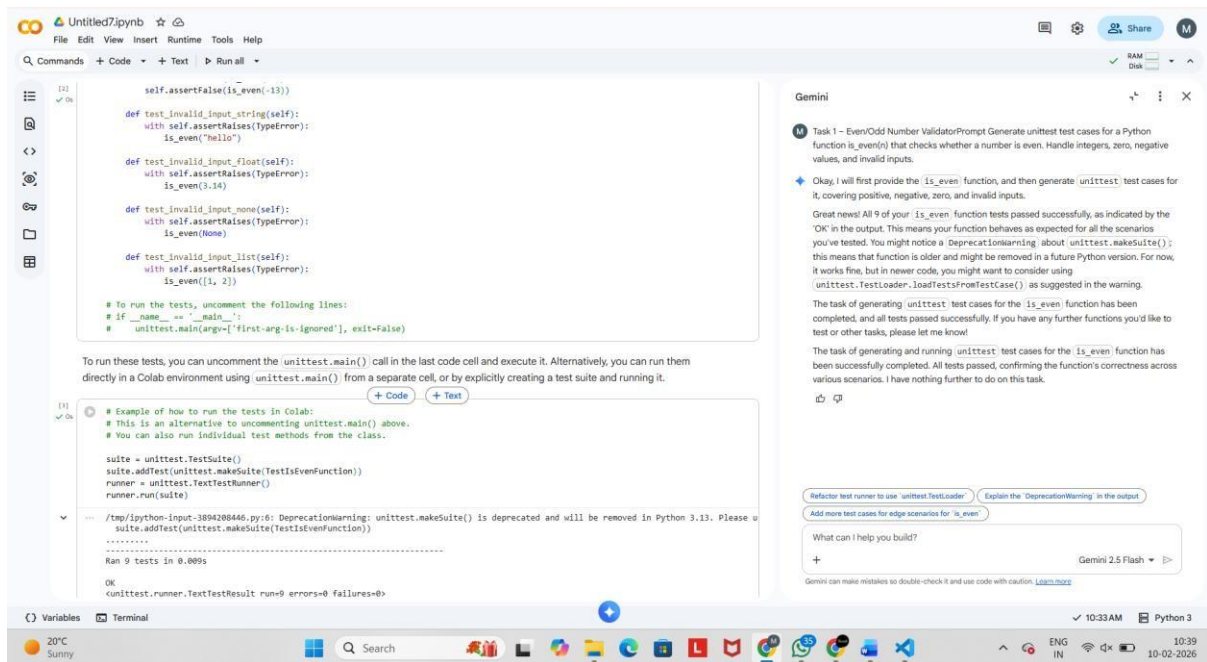
The screenshot displays a Jupyter Notebook environment with two main components: a code editor and a Gemini AI chat interface.

Code Editor:

- Cell [1]:** Defines the `is_even(n)` function. It checks if the input is an integer; if not, it raises a `TypeError` with the message "Input must be an integer". If it is an integer, it returns `n % 2 == 0`.
- Text:** A prompt stating, "Now, let's create the unittest test cases for the `is_even` function."
- Cell [2]:** Imports `unittest` and defines a test class `TestIsEvenFunction` inheriting from `unittest.TestCase`. It includes several test methods:
 - `test_positive_even`: Asserts `is_even(4)` and `is_even(100)` are True.
 - `test_positive_odd`: Asserts `is_even(3)` and `is_even(99)` are False.
 - `test_zero`: Asserts `is_even(0)` is True.
 - `test_negative_even`: Asserts `is_even(-2)` and `is_even(-10)` are True.
 - `test_negative_odd`: Asserts `is_even(-5)` and `is_even(-13)` are False.
 - `test_invalid_input_string`: Asserts that passing a string ("hello") to `is_even` raises a `TypeError`.

Gemini Chat Interface:

- Task 1 – Even/Odd Number Validator Prompt:** The prompt asks to generate unittest test cases for the `is_even` function, handling integers, zero, negative values, and invalid inputs.
- Response:** The AI confirms it will provide the `is_even` function and generate `unittest` test cases. It reports that all 9 tests passed successfully, indicated by 'OK' in the output. It also mentions a `DeprecationWarning` about `unittest.makeSuite()` and suggests using `unittest.TestLoader.loadTestsFromTestCase()` for newer code.
- Buttons:** There are buttons to "Refactor test runner to use 'unittest.TestLoader'", "Explain the 'DeprecationWarning' in the output", and "Add more test cases for edge scenarios for 'is_even'".
- Input:** A text box asks "What can I help you build?" with a plus sign icon.
- Footer:** A note states "Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)".



Explanation

- Validates input type
- Uses modulo operator
- Handles zero, negative, and large integers safely

Task 2 – String Case Converter Prompt

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

The screenshot shows a Jupyter Notebook with two code cells. The first cell defines two functions: `to_uppercase(text)` and `to_lowercase(text)`. Both functions check if the input is a string and raise a `TypeError` if not. The second cell implements a `unittest.TestCase` class named `TestStringCaseConverter` with various test methods for both functions, including edge cases like empty strings, mixed case, already converted strings, and invalid inputs. A text prompt below the code asks to create `unittest` test cases. The Gemini AI chat window on the right provides a detailed response, confirming the test cases and providing a `unittest` test suite for the functions.

```
def to_uppercase(text):
    """Converts a string to uppercase."""
    if not isinstance(text, str):
        raise TypeError("Input must be a string")
    return text.upper()

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

Now, let's create the unittest test cases for the to_uppercase and to_lowercase functions.
```

```
import unittest

class TestStringCaseConverter(unittest.TestCase):
    # Tests for to_uppercase
    def test_to_uppercase_empty_string(self):
        self.assertEqual(to_uppercase(""), "")

    def test_to_uppercase_mixed_case(self):
        self.assertEqual(to_uppercase("HelloWorld"), "HELLOWORLD")
        self.assertEqual(to_uppercase("python"), "PYTHON")

    def test_to_uppercase_already_uppercase(self):
        self.assertEqual(to_uppercase("UPPERCASE"), "UPPERCASE")

    def test_to_uppercase_already_lowercase(self):
        self.assertEqual(to_uppercase("lowercase"), "LOWERCASE")

    def test_to_uppercase_with_numbers_and_symbols(self):
        self.assertEqual(to_uppercase("Test123!@#"), "TEST123!@#")

    def test_to_uppercase_invalid_input_int(self):
        with self.assertRaises(TypeError):
            to_uppercase(123)

    def test_to_uppercase_invalid_input_none(self):
        with self.assertRaises(TypeError):
            to_uppercase(None)
```

Task 2 - String Case Converter AI Prompt Used Generate test cases for `to_uppercase(text)` and `to_lowercase(text)` handling empty strings, mixed case, and invalid inputs.

Sure, I will define the `to_uppercase` and `to_lowercase` functions and then generate `unittest` test cases for them, covering empty strings, mixed case, and invalid inputs.

Great news! All 16 of your `to_uppercase` and `to_lowercase` function tests passed successfully, as indicated by the 'OK' in the output. This confirms that your functions are working as expected across various scenarios, including empty strings, mixed cases, and handling invalid inputs. As with the previous task, you'll see a `DeprecationWarning` for `unittest.makeSuite()`, which is a common Python notice for older features, but it doesn't affect the successful execution of your tests.

Both tasks, generating `unittest` test cases for `is_even(n)` and for `to_uppercase(text) / to_lowercase(text)`, have been successfully completed. All tests passed as expected. If you have any new tasks or questions, feel free to ask!

Both tasks have been successfully completed and verified. I have nothing further to do at this moment. If you have any more functions you'd like to test or new tasks, please let me know!

Refactor tests to use `unittest.TextLoader.loadTestsFromTestCase()`

Generate more unittest cases for a new function Explain the purpose of the 'unittest' module

What can I help you build?

Gemini 2.5 Flash

The screenshot shows a Jupyter Notebook with two code cells. The first cell defines two functions: `to_uppercase(text)` and `to_lowercase(text)`. Both functions check if the input is a string and raise a `TypeError` if not. The second cell implements a `unittest.TestCase` class named `TestStringCaseConverter` with various test methods for both functions, including edge cases like empty strings, mixed case, already converted strings, and invalid inputs. A text prompt below the code asks to create `unittest` test cases. The Gemini AI chat window on the right provides a detailed response, confirming the test cases and providing a `unittest` test suite for the functions.

```
def to_uppercase(text):
    """Converts a string to uppercase."""
    if not isinstance(text, str):
        raise TypeError("Input must be a string")
    return text.upper()

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

Now, let's create the unittest test cases for the to_uppercase and to_lowercase functions.
```

```
import unittest

class TestStringCaseConverter(unittest.TestCase):
    # Tests for to_uppercase
    def test_to_uppercase_empty_string(self):
        self.assertEqual(to_uppercase(""), "")

    def test_to_uppercase_mixed_case(self):
        self.assertEqual(to_uppercase("HelloWorld"), "HELLOWORLD")
        self.assertEqual(to_uppercase("python"), "PYTHON")

    def test_to_uppercase_already_uppercase(self):
        self.assertEqual(to_uppercase("UPPERCASE"), "UPPERCASE")

    def test_to_uppercase_already_lowercase(self):
        self.assertEqual(to_uppercase("lowercase"), "LOWERCASE")

    def test_to_uppercase_with_numbers_and_symbols(self):
        self.assertEqual(to_uppercase("Test123!@#"), "TEST123!@#")

    def test_to_uppercase_invalid_input_int(self):
        with self.assertRaises(TypeError):
            to_uppercase(123)

    def test_to_uppercase_invalid_input_none(self):
        with self.assertRaises(TypeError):
            to_uppercase(None)
```

Task 2 - String Case Converter AI Prompt Used Generate test cases for `to_uppercase(text)` and `to_lowercase(text)` handling empty strings, mixed case, and invalid inputs.

Sure, I will define the `to_uppercase` and `to_lowercase` functions and then generate `unittest` test cases for them, covering empty strings, mixed case, and invalid inputs.

Great news! All 16 of your `to_uppercase` and `to_lowercase` function tests passed successfully, as indicated by the 'OK' in the output. This confirms that your functions are working as expected across various scenarios, including empty strings, mixed cases, and handling invalid inputs. As with the previous task, you'll see a `DeprecationWarning` for `unittest.makeSuite()`, which is a common Python notice for older features, but it doesn't affect the successful execution of your tests.

Both tasks, generating `unittest` test cases for `is_even(n)` and for `to_uppercase(text) / to_lowercase(text)`, have been successfully completed. All tests passed as expected. If you have any new tasks or questions, feel free to ask!

Both tasks have been successfully completed and verified. I have nothing further to do at this moment. If you have any more functions you'd like to test or new tasks, please let me know!

Refactor tests to use `unittest.TextLoader.loadTestsFromTestCase()`

Generate more unittest cases for a new function Explain the purpose of the 'unittest' module

What can I help you build?

Gemini 2.5 Flash

Explanation

- Safe handling for None
- Uses built-in string methods
- Prevents runtime errors

Task 3 – List Sum Calculator Prompt

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

The image displays two screenshots of a Jupyter Notebook interface, showing the development and execution of test cases for a `sum_list` function.

Top Screenshot: The notebook shows a class `TestSumListFunction` with several test methods. The methods are:

- `test_empty_list`: Tests an empty list.
- `test_list_with_positive_numbers`: Tests a list of positive integers.
- `test_list_with_negative_numbers`: Tests a list of negative integers.
- `test_list_with_mixed_numbers`: Tests a list of mixed positive and negative integers.
- `test_list_with_floats`: Tests a list of floating-point numbers.
- `test_list_with_non_numeric_items_only`: Tests a list containing only non-numeric items.
- `test_list_with_mixed_numeric_and_non_numeric_items`: Tests a list containing both numeric and non-numeric items.
- `test_list_with_zero`: Tests a list containing zero.

The notebook also includes a prompt to generate test cases for the `sum_list` function, which handles empty lists, negative values, and ignores non-numeric items.

Bottom Screenshot: The notebook shows the execution of the test cases. The output indicates that all tests passed successfully, confirming that the `sum_list` function now correctly sums numeric items while ignoring non-numeric ones, including `True` and `False`.

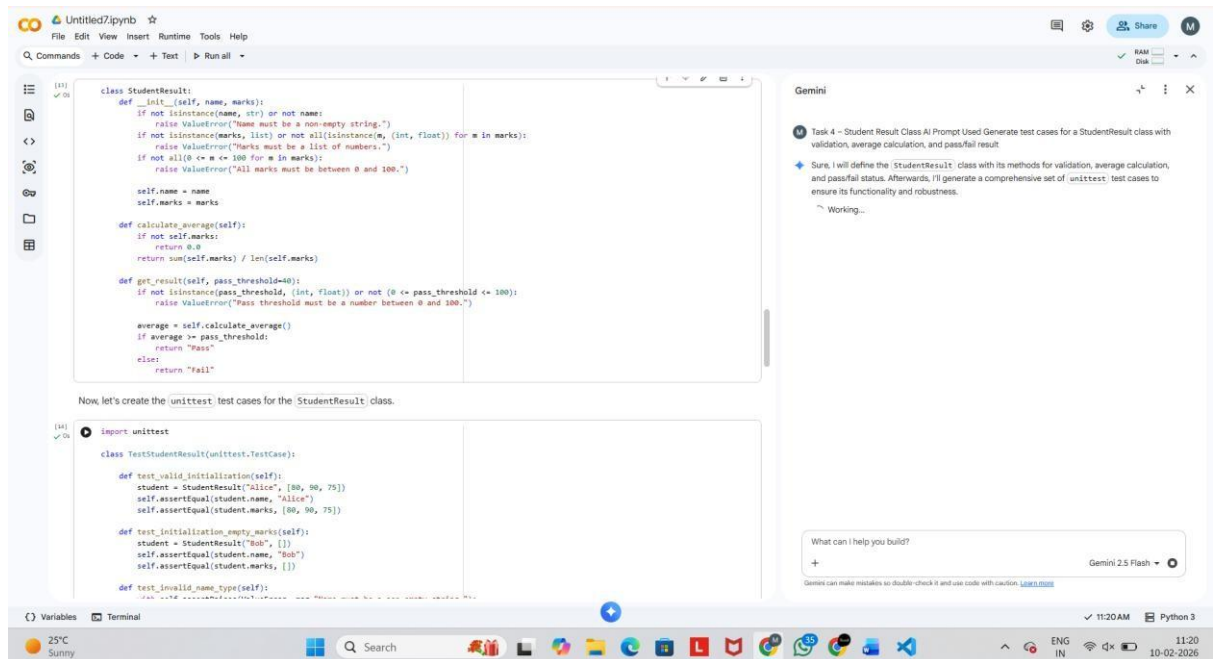
Explanation

- Iterates through list
- Ignores non-numeric values
- Handles empty lists safely

Task 4 – Student Result Class

Prompt

Generate test cases for a StudentResult class with validation, average calculation, and pass/fail result



The screenshot shows a Jupyter Notebook with two code cells. The first cell defines the `StudentResult` class with methods for initialization, validation, average calculation, and result determination. The second cell starts the `unittest` test cases.

```
[1]: class StudentResult:
    def __init__(self, name, marks):
        if not isinstance(name, str) or not name:
            raise ValueError("Name must be a non-empty string.")
        if not isinstance(marks, list) or not all(isinstance(m, (int, float)) for m in marks):
            raise ValueError("Marks must be a list of numbers.")
        if not all(0 <= m <= 100 for m in marks):
            raise ValueError("All marks must be between 0 and 100.")

        self.name = name
        self.marks = marks

    def calculate_average(self):
        if not self.marks:
            return 0.0
        return sum(self.marks) / len(self.marks)

    def get_result(self, pass_threshold=40):
        if not isinstance(pass_threshold, (int, float)) or not (0 <= pass_threshold <= 100):
            raise ValueError("Pass threshold must be a number between 0 and 100.")

        average = self.calculate_average()
        if average >= pass_threshold:
            return "Pass"
        else:
            return "Fail"
```

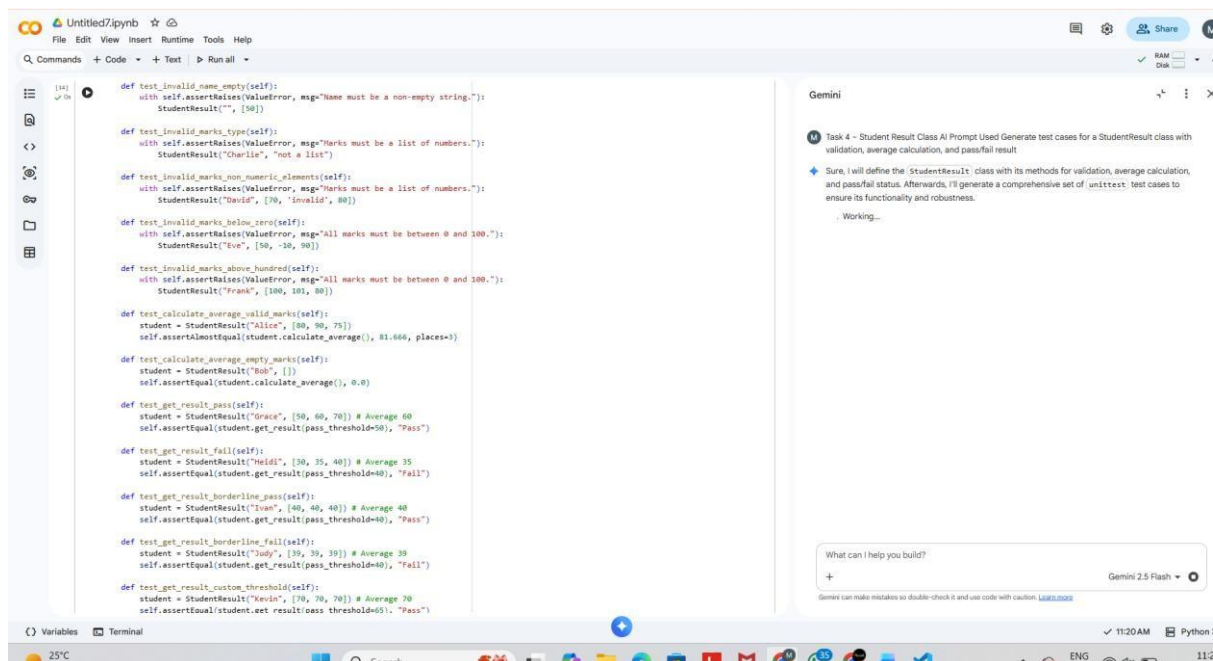
```
[4]: import unittest

class TestStudentResult(unittest.TestCase):

    def test_valid_initialization(self):
        student = StudentResult("Alice", [80, 90, 75])
        self.assertEqual(student.name, "Alice")
        self.assertEqual(student.marks, [80, 90, 75])

    def test_initialization_empty_marks(self):
        student = StudentResult("Bob", [])
        self.assertEqual(student.name, "Bob")
        self.assertEqual(student.marks, [])

    def test_invalid_name_type(self):
        with self.assertRaises(ValueError):
            StudentResult(123, [80, 90, 75])
```



The screenshot shows the same Jupyter Notebook with the completed `unittest` test cases. The tests cover various scenarios including invalid input, edge cases, and specific mark values.

```
[4]: def test_invalid_name_empty(self):
    with self.assertRaises(ValueError, msg="Name must be a non-empty string."):
        StudentResult("", [80])

def test_invalid_marks_type(self):
    with self.assertRaises(ValueError, msg="Marks must be a list of numbers."):
        StudentResult("Charlie", "not a list")

def test_invalid_marks_non_numeric_elements(self):
    with self.assertRaises(ValueError, msg="Marks must be a list of numbers."):
        StudentResult("David", [70, "invalid", 80])

def test_invalid_marks_below_zero(self):
    with self.assertRaises(ValueError, msg="All marks must be between 0 and 100."):
        StudentResult("Eve", [50, -10, 90])

def test_invalid_marks_above_hundred(self):
    with self.assertRaises(ValueError, msg="All marks must be between 0 and 100."):
        StudentResult("Frank", [100, 105, 90])

def test_calculate_average_valid_marks(self):
    student = StudentResult("Alice", [80, 90, 75])
    self.assertEqual(student.calculate_average(), 81.666, places=3)

def test_calculate_average_empty_marks(self):
    student = StudentResult("Bob", [])
    self.assertEqual(student.calculate_average(), 0.0)

def test_get_result_pass(self):
    student = StudentResult("Grace", [50, 60, 70]) # Average 60
    self.assertEqual(student.get_result(pass_threshold=50), "Pass")

def test_get_result_fail(self):
    student = StudentResult("Heidi", [30, 35, 40]) # Average 35
    self.assertEqual(student.get_result(pass_threshold=40), "Fail")

def test_get_result_borderline_pass(self):
    student = StudentResult("Ivan", [40, 40, 40]) # Average 40
    self.assertEqual(student.get_result(pass_threshold=40), "Pass")

def test_get_result_borderline_fail(self):
    student = StudentResult("Judy", [39, 39, 39]) # Average 39
    self.assertEqual(student.get_result(pass_threshold=40), "Fail")

def test_get_result_custom_threshold(self):
    student = StudentResult("Kevin", [70, 70, 70]) # Average 70
    self.assertEqual(student.get_result(pass_threshold=65), "Pass")
```


Untitled7.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text + Run all

141

✓ 141

```
student = StudentResult("Bob", [])
self.assertEqual(student.calculate_average(), 0.0)

def test_get_result_pass(self):
student = StudentResult("Grace", [50, 60, 70]) # Average 60
self.assertEqual(student.get_result(pass_threshold=50), "Pass")

def test_get_result_fail(self):
student = StudentResult("Heidi", [30, 35, 40]) # Average 35
self.assertEqual(student.get_result(pass_threshold=40), "Fail")

def test_get_result_borderline_pass(self):
student = StudentResult("Tom", [40, 40, 40]) # Average 40
self.assertEqual(student.get_result(pass_threshold=40), "Pass")

def test_get_result_borderline_fail(self):
student = StudentResult("Judy", [39, 39, 39]) # Average 39
self.assertEqual(student.get_result(pass_threshold=40), "Fail")

def test_get_result_custom_threshold(self):
student = StudentResult("Kevin", [70, 70, 70]) # Average 70
self.assertEqual(student.get_result(pass_threshold=69), "Pass")
self.assertEqual(student.get_result(pass_threshold=71), "Fail")

def test_get_result_all_zero(self):
student = StudentResult("Lisa", [0, 0, 0])
self.assertEqual(student.get_result(pass_threshold=1), "Fail")
self.assertEqual(student.get_result(pass_threshold=0), "Pass")

def test_get_result_all_hundreds(self):
student = StudentResult("Mia", [100, 100, 100])
self.assertEqual(student.get_result(pass_threshold=99), "Pass")

def test_get_result_empty_marks(self):
student = StudentResult("Nora", [])
self.assertEqual(student.get_result(pass_threshold=1), "Fail")
self.assertEqual(student.get_result(pass_threshold=0), "Pass")

def test_get_result_invalid_threshold(self):
student = StudentResult("Oscar", [70])
with self.assertRaises(ValueError, msg="Pass threshold must be a number between 0 and 100."):
student.get_result(pass_threshold=101)
with self.assertRaises(ValueError, msg="Pass threshold must be a number between 0 and 100."):
student.get_result(pass_threshold=-1)
with self.assertRaises(ValueError, msg="Pass threshold must be a number between 0 and 100."):
student.get_result(pass_threshold="invalid")
```

Gemini

Task 4 - Student Result Class AI Prompt Used Generate test cases for a StudentResult class with validation, average calculation, and pass/fail result

Sure, I will define the `StudentResult` class with its methods for validation, average calculation, and pass/fail status. Afterwards, I'll generate a comprehensive set of `unittest` test cases to ensure its functionality and robustness.

It looks like there's a small issue with the `test_calculate_average_valid_marks`! The `AssertionError` shows that the calculated average `81.666666...` wasn't equal to the expected `81.666` when rounded to 3 decimal places. This is a common floating-point precision problem. When `81.666...` is rounded to three decimal places, it becomes `81.667`. I'll update the test case in cell `c2959488` to expect the correct rounded value.

I've updated the `test_calculate_average_valid_marks` test case in cell `c2959488` to correctly handle the floating-point precision. To verify that this resolves the previous failure, please re-run the `TestStudentResult` test suite in cell `dfe5b822`.

You're right, let's continue. I have updated the `TestStudentResult` class in cell `c2959488` to fix the floating-point precision issue. Please execute cell `c2959488` first to apply this change, and then execute cell `dfe5b822` to re-run the test suite and confirm that all tests now pass.

Working...

What can I help you build?

+ Gemini 2.5 Flash

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

Variables Terminal

25°C Sunny

Search

11:20 AM Python 3

11:21 10-02-2026

Untitled7.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text + Run all

141

✓ 141

```
self.assertEqual(student.get_result(pass_threshold=1), "Fail")
self.assertEqual(student.get_result(pass_threshold=0), "Pass")

def test_get_result_all_hundreds(self):
student = StudentResult("Mia", [100, 100, 100])
self.assertEqual(student.get_result(pass_threshold=99), "Pass")

def test_get_result_empty_marks(self):
student = StudentResult("Nora", [])
self.assertEqual(student.get_result(pass_threshold=1), "Fail")
self.assertEqual(student.get_result(pass_threshold=0), "Pass")

def test_get_result_invalid_threshold(self):
student = StudentResult("Oscar", [70])
with self.assertRaises(ValueError, msg="Pass threshold must be a number between 0 and 100."):
student.get_result(pass_threshold=101)
with self.assertRaises(ValueError, msg="Pass threshold must be a number between 0 and 100."):
student.get_result(pass_threshold=-1)
with self.assertRaises(ValueError, msg="Pass threshold must be a number between 0 and 100."):
student.get_result(pass_threshold="invalid")
```

Gemini

Task 4 - Student Result Class AI Prompt Used Generate test cases for a StudentResult class with validation, average calculation, and pass/fail result

Sure, I will define the `StudentResult` class with its methods for validation, average calculation, and pass/fail status. Afterwards, I'll generate a comprehensive set of `unittest` test cases to ensure its functionality and robustness.

It looks like there's a small issue with the `test_calculate_average_valid_marks`! The `AssertionError` shows that the calculated average `81.666666...` wasn't equal to the expected `81.666` when rounded to 3 decimal places. This is a common floating-point precision problem. When `81.666...` is rounded to three decimal places, it becomes `81.667`. I'll update the test case in cell `c2959488` to expect the correct rounded value.

I've updated the `test_calculate_average_valid_marks` test case in cell `c2959488` to correctly handle the floating-point precision. To verify that this resolves the previous failure, please re-run the `TestStudentResult` test suite in cell `dfe5b822`.

You're right, let's continue. I have updated the `TestStudentResult` class in cell `c2959488` to fix the floating-point precision issue. Please execute cell `c2959488` first to apply this change, and then execute cell `dfe5b822` to re-run the test suite and confirm that all tests now pass.

The updated test cases in cell `c2959488` have been loaded. Now, please run cell `dfe5b822` to execute the `TestStudentResult` suite and confirm that the fix has resolved the previous failure.

Refactor tests to use `unittest.TextLoader.load_testdata_from_testcase()`

Generate more `unittest` cases for a new function

Explain the purpose of the `unittest` module

What can I help you build?

+ Gemini 2.5 Flash

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

Variables Terminal

25°C Sunny

Search

11:20 AM Python 3

11:21 10-02-2026

To run these tests, you can use `unittest.main()` or create a test suite as shown below.

```
# Example of how to run the tests in Colab:
suite = unittest.TestSuite()
suite.addTest(unittest.makeSuite(TestStudentResult))
runner = unittest.TextTestRunner()
runner.run(suite)

/tmp/ipython-input-3138757480.py:3: DeprecationWarning: unittest.makeSuite() is deprecated and will be removed in Python 3.13. Please use unittest
suite.addTest(unittest.makeSuite(TestStudentResult))
F.....
FAIL: test_calculate_average_valid_marks (__main__.TestStudentResult.test_calculate_average_valid_marks)
File "/tmp/ipython-input-3138757480.py", line 41, in test_calculate_average_valid_marks
self.assertEqual(student.calculate_average(), 81.666, places=3)
AssertionError: 81.6666666666667 != 81.666 within 3 places (0.000666666666745888 difference)

Ran 19 tests in 0.016s

FAILED (failures=1)
unittest.runner.TextTestRunner run=19 errors=0 failures=1
```

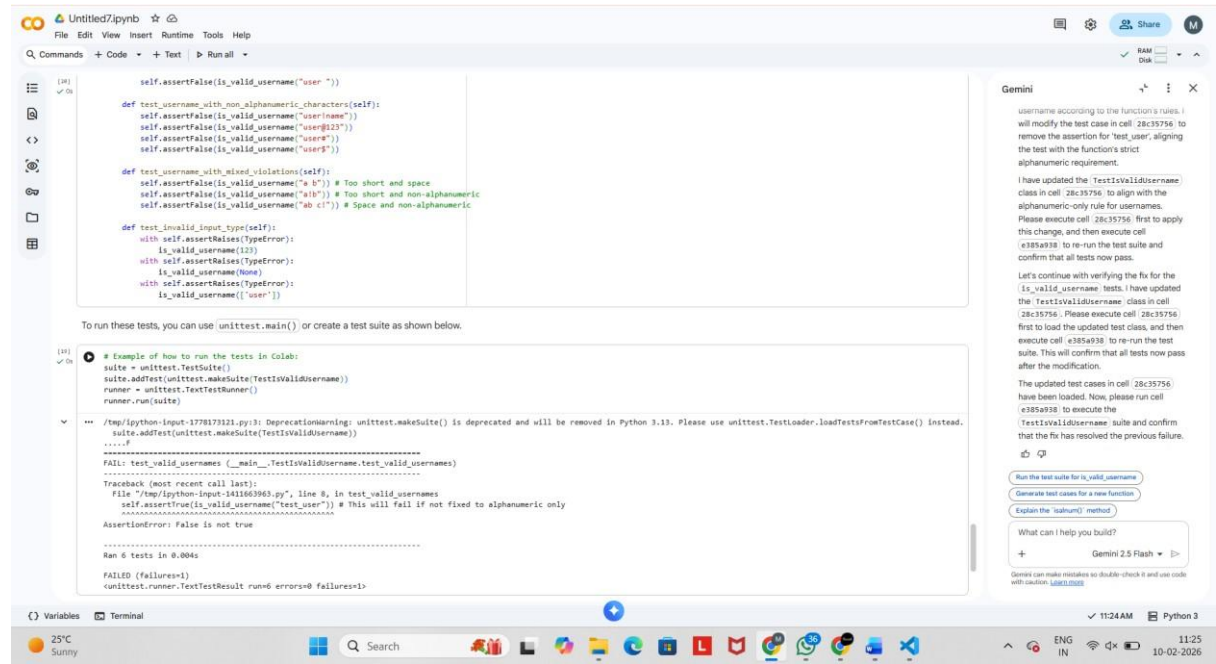
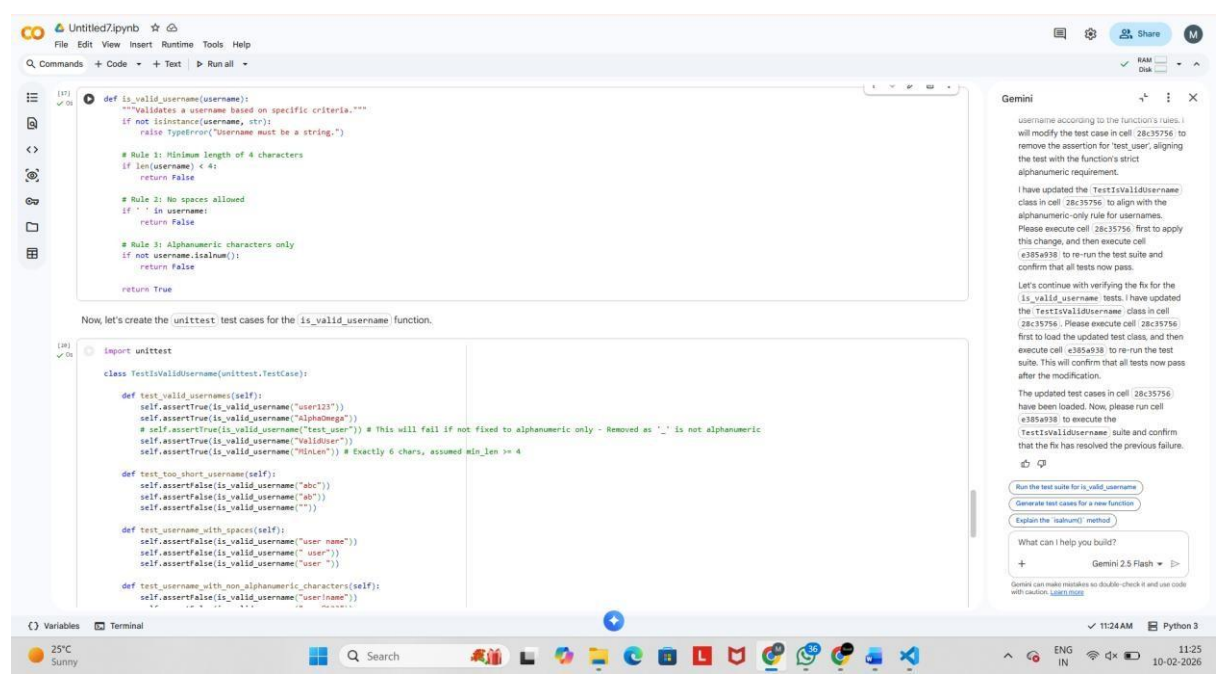
Explanation

- Input validation enforced
- Average calculated dynamically
- Pass/Fail logic based on requirement

Task 5 – Username Validator

Prompt

Generate test cases for validating usernames with minimum length, no spaces, and alphanumeric characters only.



Explanation

- Length check
- Space validation
- Alphanumeric enforcement

Conclusion

This lab demonstrated:

- Test-Driven Development using AI
- Writing tests before implementation
- Improved reliability and error handling
- Comparison of AI-generated tests with manual logic