

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 shows a Jupyter Notebook interface with two code cells. The top cell contains the following Python code:

```
def is_even(n):
    """Checks if a number is even."""
    if not isinstance(n, int):
        raise TypeError("Input must be an integer")
    return n % 2 == 0
```

The bottom cell contains the generated unittest test cases:

```
import unittest

class TestIsEvenFunction(unittest.TestCase):

    def test_positive_even(self):
        self.assertTrue(is_even(4))
        self.assertTrue(is_even(100))

    def test_positive_odd(self):
        self.assertFalse(is_even(3))
        self.assertFalse(is_even(99))

    def test_zero(self):
        self.assertTrue(is_even(0))

    def test_negative_even(self):
        self.assertTrue(is_even(-2))
        self.assertTrue(is_even(-10))

    def test_negative_odd(self):
        self.assertFalse(is_even(-5))
        self.assertFalse(is_even(-13))

    def test_invalid_input_string(self):
        with self.assertRaises(TypeError):
            is_even("hello")
```

On the right side of the interface, there is a sidebar titled "Gemini" which displays AI-generated comments and suggestions:

- Task 1 – Even/Odd Number ValidatorPrompt 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.
- Okay, I will first provide the `is_even` function, and then generate `unittest` test cases for it, covering positive, negative, zero, and invalid inputs.
- Great news! All 9 of your `is_even` function tests passed successfully, as indicated by the 'OK' in the output. This means your function behaves as expected for all the scenarios you've tested. You might notice a `DeprecationWarning` about `unittest.makeSuite()`; this means that function is older and might be removed in a future Python version. For now, it works fine, but in newer code, you might want to consider using `unittest.TestLoader.loadTestsFromTestCase()` as suggested in the warning.
- The task of generating `unittest` test cases for the `is_even` function has been completed, and all tests passed successfully. If you have any further functions you'd like to test or other tasks, please let me know!
- The task of generating and running `unittest` test cases for the `is_even` function has been successfully completed. All tests passed, confirming the function is working correctly.
- Refactor test runner to use `'unittest.TestLoader'`
- Explain the `'DeprecationWarning'` in the output
- Add more test cases for edge scenarios for `is_even`

At the bottom of the sidebar, there is a message: "What can I help you build?".

The screenshot shows a Google Colab notebook titled "Untitled7.ipynb". The code cell contains test cases for the `is_even` function, including tests for integers, floats, and lists. A note at the bottom of the cell says: "To run these tests, you can uncomment the unittest.main() call in the last code cell and execute it. Alternatively, you can run them directly in a Colab environment using unittest.main() from a separate cell, or by explicitly creating a test suite and running it." Below this is another code cell with comments on how to run the tests in Colab.

The right side of the screen shows a Gemini AI interface. It has a sidebar with "Gemini" and a main area with a message from Gemini about generating test cases for the `is_even` function. Gemini suggests refactoring the code to use `unittest.TestLoader.loadTestsFromTestCase()` and explains the deprecation warning. It also provides a link to learn more about the warning.

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 cell containing Python code for string conversion functions and their unit tests. The code includes `to_uppercase` and `to_lowercase` methods with docstrings and assertions for various input types. A second cell shows the generated unit test code for these methods. To the right, the Gemini AI interface displays a task card for generating test cases, with notes about the success of the generated tests and a link to the generated code.

```

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("123!@#"), "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)

    def test_to_uppercase_invalid_input_list(self):
        with self.assertRaises(TypeError):
            to_uppercase(["a", "b"])

    # Tests for to_lowercase
    def test_to_lowercase_empty_string(self):
        self.assertEqual(to_lowercase(""), "")

    def test_to_lowercase_mixed_case(self):
        self.assertEqual(to_lowercase("HelloWorld"), "HelloWorld")
        self.assertEqual(to_lowercase("Python"), "python")

    def test_to_lowercase_already_lowercase(self):
        self.assertEqual(to_lowercase("lowercase"), "lowercase")

    def test_to_lowercase_already_uppercase(self):
        self.assertEqual(to_lowercase("UPPERCASE"), "uppercase")

    def test_to_lowercase_with_numbers_and_symbols(self):
        self.assertEqual(to_lowercase("Test123!@#"), "test123!@#")

    def test_to_lowercase_invalid_input_int(self):
        with self.assertRaises(TypeError):
            to_lowercase(123)

    def test_to_lowercase_invalid_input_none(self):
        with self.assertRaises(TypeError):
            to_lowercase(None)

    def test_to_lowercase_invalid_input_list(self):
        with self.assertRaises(TypeError):
            to_lowercase(["a", "b"])

```

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(TestStringCaseConverter))
runner = unittest.TextTestRunner()
runner.run(suite)

```

/tmp/ipython-Input-3015477513.py:3: DeprecationWarning: unittest.makeSuite() is deprecated and will be removed in Python 3.13. Please use unittest.suite.addTest(unittest.makeSuite(TestStringCaseConverter))

.....

Run 16 tests in 0.017s

OK

<unittest.runner.TextTestResult run=16 errors=0 failures=0>

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 screenshot shows a Jupyter Notebook interface with two main panes. The left pane displays Python test code for a `sum_list` function, including various test cases like empty lists, positive numbers, negative numbers, mixed types, floats, and non-numeric items. The right pane shows the results of the tests, including a warning about the deprecation of `makeSuite`. The bottom of the screen shows a Windows taskbar with various application icons.

```
File Edit View Insert Runtime Tools Help
File Commands + Code + Text Run all

Now, let's create the unittest test cases for the sum_list function.

[11] In [11]
Import unittest
class TestSumListFunction(unittest.TestCase):
    def test_empty_list(self):
        self.assertEqual(sum_list([]), 0)
    def test_list_with_positive_numbers(self):
        self.assertEqual(sum_list([1, 2, 3, 4, 5]), 15)
        self.assertEqual(sum_list([10, 20, 30]), 60)
    def test_list_with_negative_numbers(self):
        self.assertEqual(sum_list([-1, -2, -3]), -6)
        self.assertEqual(sum_list([-100]), -100)
    def test_list_with_mixed_numbers(self):
        self.assertEqual(sum_list([-1, 0, 1, 2, -5]), -3)
        self.assertEqual(sum_list([5, -2, 8, 4]), 7)
    def test_list_with_floats(self):
        self.assertEqual(sum_list([1.5, 2.5, 3.0]), 7.0)
        self.assertEqual(sum_list([-1.5, 2.0]), 0.5)
    def test_list_with_non_numeric_items_only(self):
        # With the fix, True/False should be ignored
        self.assertEqual(sum_list(['a', 'b', 'c']), 0)
        self.assertEqual(sum_list([None, True, False]), 0)
    def test_list_with_mixed_numeric_and_non_numeric_items(self):
        self.assertEqual(sum_list([1, 'a', 2, None, 3.5]), 6.5)
        self.assertEqual(sum_list(['hello', -5, 10, 'world', 0]), 5)
        self.assertEqual(sum_list([10, [], 20, 'test', -5]), 25)
    def test_list_with_zeros(self):
        self.assertEqual(sum_list([0, 0, 0]), 0)
        self.assertEqual(sum_list([0, 5, -5]), 0)

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

[12] In [12]
# Example of how to run the tests in Colab:
suite = unittest.TestSuite()
suite.addTest(unittest.makeSuite(TestSumListFunction))

Gemini 2.5 Flash ▾ ▷
Gemini can make mistakes so double-check it and use code with caution. Learn more

Gemini
I have updated the sum_list function in cell 5191a239 to correctly ignore boolean values, which were being inadvertently summed as integers. To confirm that this fix has resolved the test failure, please re-execute the sum_list function definition cell (5191a239) and then re-run the test suite in cell 2c08807e.

I have successfully identified and provided a fix for the sum_list function to correctly ignore boolean values when summing. The updated code in cell 5191a239 now explicitly checks for and excludes bool types. To verify that this fix resolves the test failure, please re-execute the updated sum_list function definition in cell 5191a239, and then re-run the test suite in cell 2c08807e. Once you've done that, I can confirm the results.

Excellent! The sum_list function (cell 5191a239) has been updated and re-executed. To confirm that the fix has resolved the test failure, please re-run the test suite in cell 2c08807e.

Task 3 - List Sum Calculator AI Prompt Used Generate test cases for sum_list(numbers) that handles empty lists, negative values, and ignores non-numeric items.

It looks like the test cases for the sum_list function were removed. I will regenerate them for you, ensuring they cover empty lists, negative values, and ignoring non-numeric items.

Excellent! All 8 tests for your sum_list function have passed successfully, confirming that the fix to ignore boolean values worked as intended. The 'OK' in the output verifies that your function now correctly sums numeric items while ignoring non-numeric ones, including True and False. You still see the DeprecationWarning for unittest.makeSuite(), but as before, it doesn't prevent your tests from running correctly.

All tasks, including generating and testing the is_even, to_uppercase, to_lowercase, and sum_list functions, have been successfully completed and verified. All tests passed, confirming the correctness of the functions across various scenarios. I have nothing further to do at this moment. If you have any new tasks or questions, feel free to ask!
```

+ Refactor tests to use `unittest.TextLoader.loadFromTextCase()`

+ Generate more unit test 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

11:13 AM Python 3 10-02-2026

```
File Edit View Insert Runtime Tools Help
File Commands + Code + Text Run all

self.assertEqual(sum_list([10, 20, 30]), 60)

def test_list_with_negative_numbers(self):
    self.assertEqual(sum_list([-1, -2, -3]), -6)
    self.assertEqual(sum_list([-100]), -100)

def test_list_with_mixed_numbers(self):
    self.assertEqual(sum_list([-1, 0, 1, 2, -5]), -3)
    self.assertEqual(sum_list([5, -2, 8, -4]), 7)

def test_list_with_floats(self):
    self.assertEqual(sum_list([1.5, 2.5, 3.0]), 7.0)
    self.assertEqual(sum_list([-1.5, 2.0]), 0.5)

def test_list_with_non_numeric_items_only(self):
    # With the fix, True/False should be ignored
    self.assertEqual(sum_list(['a', 'b', 'c']), 0)
    self.assertEqual(sum_list([None, True, False]), 0)

def test_list_with_mixed_numeric_and_non_numeric_items(self):
    self.assertEqual(sum_list([1, 'a', 2, None, 3.5]), 6.5)
    self.assertEqual(sum_list(['hello', -5, 10, 'world', 0]), 5)
    self.assertEqual(sum_list([10, [], 20, 'test', -5]), 25)

def test_list_with_zeros(self):
    self.assertEqual(sum_list([0, 0, 0]), 0)
    self.assertEqual(sum_list([0, 5, -5]), 0)

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

[13] In [13]
# Example of how to run the tests in Colab:
suite = unittest.TestSuite()
suite.addTest(unittest.makeSuite(TestSumListFunction))
runner = unittest.TextTestRunner()
runner.run(suite)

/tmp/python-input-378467008.py:3: DeprecationWarning: unittest.makeSuite() is deprecated and will be removed in Python 3.13. Please use unittest.suite.addTest(unittest.makeSuite(testSumListFunction))
.....
Ran 8 tests in 0.006s

OK
<unittest.runner.TextTestResult run=8 errors=0 failures=0>

Gemini 2.5 Flash ▾ ▷
Gemini can make mistakes so double-check it and use code with caution. Learn more

Gemini
I have updated the sum_list function in cell 5191a239 to correctly ignore boolean values, which were being inadvertently summed as integers. To confirm that this fix has resolved the test failure, please re-execute the sum_list function definition cell (5191a239) and then re-run the test suite in cell 2c08807e.

I have successfully identified and provided a fix for the sum_list function to correctly ignore boolean values when summing. The updated code in cell 5191a239 now explicitly checks for and excludes bool types. To verify that this fix resolves the test failure, please re-execute the updated sum_list function definition in cell 5191a239, and then re-run the test suite in cell 2c08807e. Once you've done that, I can confirm the results.

Excellent! The sum_list function (cell 5191a239) has been updated and re-executed. To confirm that the fix has resolved the test failure, please re-run the test suite in cell 2c08807e.

Task 3 - List Sum Calculator AI Prompt Used Generate test cases for sum_list(numbers) that handles empty lists, negative values, and ignores non-numeric items.

It looks like the test cases for the sum_list function were removed. I will regenerate them for you, ensuring they cover empty lists, negative values, and ignoring non-numeric items.

Excellent! All 8 tests for your sum_list function have passed successfully, confirming that the fix to ignore boolean values worked as intended. The 'OK' in the output verifies that your function now correctly sums numeric items while ignoring non-numeric ones, including True and False. You still see the DeprecationWarning for unittest.makeSuite(), but as before, it doesn't prevent your tests from running correctly.

All tasks, including generating and testing the is_even, to_uppercase, to_lowercase, and sum_list functions, have been successfully completed and verified. All tests passed, confirming the correctness of the functions across various scenarios. I have nothing further to do at this moment. If you have any new tasks or questions, feel free to ask!
```

+ Refactor tests to use `unittest.TextLoader.loadFromTextCase()`

+ Generate more unit test 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

11:13 AM Python 3 10-02-2026

```
File Edit View Insert Runtime Tools Help
File Commands + Code + Text Run all

25°C Sunny
Terminal
Windows Search
ENG IN Wi-Fi Disk
11:13 AM Python 3 10-02-2026
```

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 interface with two code cells. The top cell contains the definition of the `StudentResult` class, which includes methods for initialization, calculating average, and getting results based on a pass threshold. The bottom cell contains test cases for the `TestStudentResult` class, using `assertEqual` and `assertRaises` to validate various scenarios like empty name, invalid marks type, and failing the average check.

```
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"

Now, let's create the unittest test cases for the StudentResult class.

class TestStudentResult(unittest.TestCase):

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

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

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

This screenshot shows a more comprehensive set of test cases for the `StudentResult` class. It includes tests for invalid names (empty string, non-string), invalid mark types (non-list, non-numeric), non-numeric elements in marks, marks below zero, marks above hundred, calculate average with valid and empty marks, and various get_result scenarios including borderlines and custom thresholds. The tests use `assertEqual`, `assertRaises`, and `assertAlmostEqual`.

```
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", [90, "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, 101, 80])

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

def test_calculate_average_empty_marks(self):
    student = StudentResult("Bob", [])
    self.assertAlmostEqual(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", [30, 30, 30]) # Average 30
    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")
```

```

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=40), "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")
    self.assertEqual(student.get_result(pass_threshold=75), "Fail")

def test_get_result_all_marks(self):
    student = StudentResult("Liam", [80, 80, 80])
    self.assertEqual(student.get_result(pass_threshold=80), "Pass")
    self.assertEqual(student.get_result(pass_threshold=90), "Pass")

def test_get_result_empty_marks(self):
    student = StudentResult("Nina", [100, 100, 100])
    self.assertEqual(student.get_result(pass_threshold=90), "Pass")
    self.assertEqual(student.get_result(pass_threshold=100), "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

M 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 `c2059488` to expect the correct rounded value.

I've updated the `test_calculate_average_valid_marks` test case in cell `c2059488` to correctly handle the floating-point precision. To verify that this resolves the previous failure, please re-run the `TestStudentResult` test suite in cell `dfe5bae22`.

You're right, let's continue. I have updated the `TestStudentResult` class in cell `c2059488` to fix the floating-point precision issue. Please execute cell `c2059488` first to apply this change, and then execute cell `dfe5bae22` to re-run the test suite and confirm that all tests now pass.

Working...

What can I help you build?

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

Variables Terminal

25°C Sunny

11:20 AM Python 3 ENG IN 10-02-2026

```

student = StudentResult("Bob", [])
self.assertEqual(student.get_result(pass_threshold=40), "Fail")
self.assertEqual(student.get_result(pass_threshold=0), "Pass")

def test_get_result_all_hundreds(self):
    student = StudentResult("Grace", [100, 100, 100])
    self.assertEqual(student.get_result(pass_threshold=90), "Pass")

def test_get_result_empty_marks(self):
    student = StudentResult("Heidi", [ ])
    self.assertEqual(student.get_result(pass_threshold=40), "Fail")
    self.assertEqual(student.get_result(pass_threshold=0), "Pass")

def test_get_result_invalid_threshold(self):
    student = StudentResult("Ivan", [80])
    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")

```

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-2330757480.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)
Traceback (most recent call last):
File "/tmp/ipython-Input-1004771171.py", line 41, in test_calculate_average_valid_marks
 self.assertAlmostEqual(student.calculate_average(), 81.666, places=3)
AssertionError: 81.66666666666667 != 81.666 within 3 places (0.00066666666745886 difference)

Ran 19 tests in 0.01s
FAILED (failures=1)
`unittest.runner.TextTestRunner().run(19 errors=0 failures=1)`

Refactor tests to use `unittest.TextTestRunner().loadTestsFromTestCase()`
Generate more unit tests for a new function Explain the purpose of the `unittest` module

What can I help you build?

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

Variables Terminal

25°C Sunny

11:20 AM Python 3 ENG IN 10-02-2026

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.

```

def is_valid_username(username):
    """Validates a username based on specific criteria."""
    if not isinstance(username, str):
        raise TypeError("Username must be a string.")

    # Rule 1: Minimum length of 4 characters
    if len(username) < 4:
        return False

    # Rule 2: No spaces allowed
    if ' ' in username:
        return False

    # Rule 3: Alphanumeric characters only
    if not username.isalnum():
        return False

    return True

```

Now, let's create the `unittest` test cases for the `is_valid_username` function.

```

import unittest

class TestisValidUsername(unittest.TestCase):

    def test_is_valid_username(self):
        self.assertTrue(is_valid_username("user123"))
        self.assertTrue(is_valid_username("AlphOmega"))
        # self.assertFalse(is_valid_username("test_user")) # This will fail if not fixed to alphanumeric only - Removed as '_' is not alphanumeric
        self.assertTrue(is_valid_username("ValidUser"))
        self.assertTrue(is_valid_username("9Only")) # Exactly 6 chars, assumed min_len >= 4

    def test_too_short_username(self):
        self.assertFalse(is_valid_username("abc"))
        self.assertFalse(is_valid_username("ab"))
        self.assertFalse(is_valid_username(""))

    def test_username_with_spaces(self):
        self.assertFalse(is_valid_username("user name"))
        self.assertFalse(is_valid_username(" user"))
        self.assertFalse(is_valid_username("user "))

    def test_username_with_non_alphanumeric_characters(self):
        self.assertFalse(is_valid_username("user$name"))

```

The screenshot shows the Jupyter Notebook interface with the code editor, a terminal at the bottom, and the Gemini extension sidebar on the right. The sidebar provides feedback on the code changes and offers suggestions for running tests.

```

        self.assertFalse(is_valid_username("user "))

    def test_username_with_non_alphanumeric_characters(self):
        self.assertFalse(is_valid_username("username"))
        self.assertFalse(is_valid_username("user@123"))
        self.assertFalse(is_valid_username("user#"))
        self.assertFalse(is_valid_username("user$"))

    def test_username_with_mixed_violations(self):
        self.assertFalse(is_valid_username("a b")) # Too short and space
        self.assertFalse(is_valid_username("ab")) # Too short and non-alphanumeric
        self.assertFalse(is_valid_username("ab c")) # Space and non-alphanumeric

    def test_invalid_input_type(self):
        with self.assertRaises(TypeError):
            is_valid_username(1)
        with self.assertRaises(TypeError):
            is_valid_username(None)
        with self.assertRaises(TypeError):
            is_valid_username(['user'])

    def test_username_with_mixed_violations(self):
        self.assertFalse(is_valid_username("a b")) # Too short and space
        self.assertFalse(is_valid_username("ab")) # Too short and non-alphanumeric
        self.assertFalse(is_valid_username("ab c")) # Space and non-alphanumeric

```

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(TestisValidUsername))
runner = unittest.TextTestRunner()
runner.run(suite)

# /tmp/python-input-1778473121.py:3: DeprecationWarning: unittest.makeSuite() is deprecated and will be removed in Python 3.13. Please use unittest.TestLoader.loadTestsFromTestCase() instead.
...
FAIL: test_valid_username (<main_.TestisValidUsername.test_valid_username>)
...
Traceback (most recent call last):
File "/tmp/python-input-1411063963.py", line 8, in test_valid_username
    self.assertTrue(is_valid_username("test_user")) # This will fail if not fixed to alphanumeric only
AssertionError: False is not true
Ran 6 tests in 0.004s
FAILED (failures=1)
unittest.runner.TextTestResult run=6 errors=0 failures=1>

```

The screenshot shows the Jupyter Notebook interface with the code editor, a terminal at the bottom, and the Gemini extension sidebar on the right. The sidebar provides feedback on the code changes and offers suggestions for running tests.

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