Test Report

Run ID: N/A • Generated: 2026-01-18 17:14:39 • Duration: 138ms
Plugin: v0.1.0 (b7a157f6cb9189cc50a17c846484c8454deeac61) [dirty]
Repo: v1.0.0 (1c69689ab92cbb103a35b7d434aa553598fe0fae) [dirty]
LLM: ollama / llama3.2:1b (minimal context, 26 annotated, 2 errors)

100.0%
Total Coverage

28	22	2	2
TOTAL TESTS	PASSED	FAILED	SKIPPED

1	1	0
XFAILED	XPASSED	ERRORS

Source Coverage Per Test Details Failures Only

Source Coverage

FILE	STMTS	MISS	COVER	%	COVERED LINES	MISSED LINES
calculator.py	26	0	26	100.0%	8, 18, 21, 31, 34, 44, 47, 60-62, 65, 74, 77, 86, 89, 101-105, 108, 120-124	-

Per Test Details

📄 **example_tests.py** 28 tests

PASSED example_tests.py::TestCalculator::test_addition 0ms 1

AI ASSESSMENT

Scenario: Verify the addition function works correctly for various inputs.

Why Needed: This test prevents a potential bug where the addition function does not handle negative numbers correctly.

Key Assertions:

- The `add` function should return the correct result when both inputs are positive.
- The `add` function should return 0 when one input is 0 and the other is also 0.
- The `add` function should return 0 for any negative input.
- The `add` function should handle zero as a valid input without raising an error.
- The `add` function should preserve the sign of both inputs when adding them together.

COVERAGE

calculator.py	1 lines (ranges: 18)

PASSED example_tests.py::TestCalculator::test_division 0ms 1

AI ASSESSMENT

Scenario: The test verifies that division operations are performed correctly for various inputs.

Why Needed: This test prevents a potential division by zero error and ensures accurate results in all scenarios.

Key Assertions:

- assert divide(10, 2) == 5.0
- assert divide(7, 2) == 3.5
- assert divide(0, 5) == 0.0

COVERAGE

calculator.py	2 lines (ranges: 60, 62)

PASSED `example_tests.py::TestCalculator::test_division_by_zero` 1ms 1

AI ASSESSMENT

Scenario: The test verifies that the `divide` function correctly raises a `ValueError` when attempting to divide by zero.

Why Needed: This test prevents division by zero from being silently ignored and potentially causing unexpected behavior or errors in downstream code.

Key Assertions:

- The `divide` function should raise a `ValueError` with the message 'Cannot divide by zero'.
- The error message should include the phrase 'Cannot divide by zero' to clearly indicate the cause of the issue.
- The test should verify that the `match` parameter in `pytest.raises()` is set to match the expected error message.
- The `divide` function should be called with a non-zero argument to trigger the division by zero scenario.
- The test should check for any additional error messages or details provided by the `divide` function.
- The `divide` function should raise a `ValueError` exception instead of returning a special value (e.g., None) when dividing by zero.

COVERAGE

`calculator.py`	`2 lines (ranges: 60-61)`

PASSED `example_tests.py::TestCalculator::test_multiplication` 0ms 1

AI ASSESSMENT

Scenario: Verifies the correct result of multiplying two numbers.

Why Needed: Prevents a potential bug where incorrect multiplication results are returned for certain inputs (e.g., negative numbers or zero).

Key Assertions:

- The function `multiply(6, 7)` should return `42`.
- The function `multiply(-2, 3)` should return `-6`.
- The function `multiply(0, 100)` should return `0`.

COVERAGE

`calculator.py`	`1 lines (ranges: 44)`

PASSED example_tests.py::TestCalculator::test_subtraction 0ms 1

AI ASSESSMENT

Scenario: The test verifies the correct subtraction of numbers.

Why Needed: This test prevents a potential bug where incorrect subtraction results in an incorrect output.

Key Assertions:

- assert subtract(10, 3) == 7
- assert subtract(5, 5) == 0
- assert subtract(0, 5) == -5

COVERAGE

calculator.py	1 lines (ranges: 31)

FAILED example_tests.py::TestErrors::test_with_error 0ms

ERROR

```
RuntimeError: Intentional error for demo
```

AI ASSESSMENT

Scenario: The test verifies that a RuntimeError is raised when the function `test_with_error` is called.

Why Needed: This test prevents a potential regression where a RuntimeError might be silently ignored or masked by other exceptions.

Key Assertions:

- Raises a RuntimeError with the specified message.
- Does not catch any other type of exception (e.g., TypeError, ValueError).
- Does not handle any specific error conditions (e.g., no argument passed to `test_with_error`).

XFAILED example_tests.py::TestExpectedFailures::test_known_bug 0ms

ERROR

```
self =

    @pytest.mark.xfail(reason="Known bug in edge case handling")
    def test_known_bug(self):
        """Test that fails due to a
```

AI ASSESSMENT

Scenario: The test verifies that the function raises an AssertionError when a known bug is encountered.

Why Needed: This test prevents regression by ensuring that the function raises an exception when it encounters a known issue.

Key Assertions:

- Raises an AssertionError with the message 'Known bug'.
- Does not handle the known bug case correctly.
- Expected the function to raise an exception, but did not.
- The test does not verify that the error is properly propagated or handled.
- The test does not ensure that the error is reported in a meaningful way (e.g., with a specific message).
- The test does not provide any useful feedback about what went wrong when the known bug occurs.
- The test does not account for other potential errors that may occur in the function.

XPASSED example_tests.py::TestExpectedFailures::test_xpass_demo 0ms

AI ASSESSMENT

Scenario: The test verifies that the `test_xpass_demo` function returns a successful assertion.

Why Needed: This test prevents a regression where the expected failure is not caught due to an incorrect implementation of the xfail decorator.

Key Assertions:

- The function should return True.
- The function should not raise any exceptions.
- The function should not be marked as failed by the `xpassed` flag.
- The function's output should match the expected result.
- Any raised exceptions should be caught and handled correctly.
- The test should fail if the function returns a non-zero value.

FAILED `example_tests.py::TestFailures::test_expected_failure_demo` 1ms

ERROR

```
AssertionError: Intentional failure for demo purposes
```

AI ASSESSMENT

Scenario: The test verifies that the function `test_expected_failure_demo` is expected to fail when asserting an incorrect value.

Why Needed: This test prevents a regression where the function might incorrectly report a successful outcome due to a bug in its logic.

Key Assertions:

- asserts that the function will raise an AssertionError with message 'Intentional failure for demo purposes'
- checks if the assertion `1 == 2` is incorrect
- verifies that the error message is correct and not something else
- ensures that the test fails as expected

PASSED `example_tests.py::TestNumberProperties::test_is_even` 0ms 1

AI ASSESSMENT

Scenario: Test the `is_even` function with various numbers to verify its correctness.

Why Needed: This test prevents a potential bug where the function incorrectly identifies even numbers as odd, leading to incorrect results in certain scenarios.

Key Assertions:

- {'description': 'Check if the function correctly identifies even numbers.', 'condition': 'is_even(2) is True', 'expected_result': True}
- {'description': 'Verify that the function returns True for multiples of 2.', 'condition': 'is_even(4) is True', 'expected_result': True}
- {'description': 'Ensure the function correctly handles even numbers with leading zeros.', 'condition': 'is_even(0) is True', 'expected_result': True}
- {'description': 'Test that the function returns False for odd numbers.', 'condition': 'is_even(1) is False', 'expected_result': False}
- {'description': 'Verify that the function handles negative numbers correctly.', 'condition': 'is_even(-3) is False', 'expected_result': False}

COVERAGE

`calculator.py`	1 lines (ranges: 74)

PASSED example_tests.py::TestNumberProperties::test_is_positive 0ms 1

AI ASSESSMENT

Scenario: The test verifies that the `is_positive` function correctly identifies positive numbers.

Why Needed: This test prevents a potential bug where the function incorrectly returns True for negative numbers or zero.

Key Assertions:

- assert is_positive(1) is True
- assert is_positive(100) is True
- assert is_positive(0) is False

COVERAGE

calculator.py	1 lines (ranges: 86)

PASSED example_tests.py::TestParameterized::test_add_numbers[-5-5-0] 0ms 1

AI ASSESSMENT

Scenario: The `add` function is tested with three different input combinations: (1, 2), (3, 4), and (-5, -5).

Why Needed: This test prevents a potential bug where the `add` function returns incorrect results for negative numbers.

Key Assertions:

- The sum of a and b is equal to expected.
- The difference between add(a, b) and expected is zero.
- If a is negative and b is positive, then add(a, b) should be the same as expected.

COVERAGE

calculator.py	1 lines (ranges: 18)

PASSED example_tests.py::TestParameterized::test_add_numbers[1-1-2] 0ms 1

AI ASSESSMENT

Scenario: Verify correct addition of two numbers for different input scenarios.

Why Needed: Prevents incorrect addition due to potential overflow or underflow issues when adding very large or small numbers.

Key Assertions:

- The function `add(a, b)` returns the expected result for all given inputs.
- The function `add(a, b)` handles edge cases where one or both of the input numbers are zero.
- The function `add(a, b)` correctly handles negative numbers and their sums.
- The function `add(a, b)` avoids division by zero errors when adding a number to zero.
- The function `add(a, b)` maintains the correct order of operations (PEMDAS) for complex arithmetic.
- The function `add(a, b)` is stable under certain input conditions (e.g., negative numbers close to zero).

COVERAGE

calculator.py	1 lines (ranges: 18)

PASSED example_tests.py::TestParameterized::test_add_numbers[10-20-30] 0ms 1

AI ASSESSMENT

Scenario: The `add` function should be able to handle input values of 10, 20, and 30 without raising an error.

Why Needed: This test prevents a potential bug where the `add` function would raise a TypeError or ValueError for inputs that are not numbers.

Key Assertions:

- The result of adding `a` and `b` should be equal to `expected`.
- The `add` function should handle input values of 10, 20, and 30 without raising an error.
- The `add` function should return the correct expected value for inputs of 10, 20, and 30.

COVERAGE

calculator.py	1 lines (ranges: 18)

PASSED `example_tests.py::TestParameterized::test_add_numbers[2-3-5]` 0ms 1

AI ASSESSMENT

LLM error: Failed to parse LLM response as JSON

COVERAGE

`calculator.py`	`1 lines (ranges: 18)`

PASSED `example_tests.py::TestParameterized::test_is_even_parametrized[-4-True]` 0ms 1

AI ASSESSMENT

Scenario: Testing the `is_even` function with a parameter of type `int`.

Why Needed: This test prevents a potential bug where the function does not correctly handle non-integer inputs.

Key Assertions:

- The input `n` should be an integer.
- The result of `is_even(n)` should match the expected value `expected`.
- If `n` is not an integer, the test should fail and report a bug.

COVERAGE

`calculator.py`	`1 lines (ranges: 74)`

PASSED `example_tests.py::TestParameterized::test_is_even_parametrized[0-True]` 0ms 1

AI ASSESSMENT

LLM error: Failed to parse LLM response as JSON

COVERAGE

`calculator.py`	`1 lines (ranges: 74)`

PASSED `example_tests.py::TestParameterized::test_is_even_parametrized[1-False]` 0ms 1

AI ASSESSMENT

Scenario: Testing the `is_even` function with a parameter of type `int` and an expected value of `False`.

Why Needed: This test prevents a potential bug where the function might return incorrect results for odd inputs.

Key Assertions:

- The input `n` is an integer.
- The function `is_even(n)` returns `True` if `n` is even and `False` otherwise.
- If `n` is an odd number, `is_even(n)` should return `False`.
- If `n` is 0 or negative, `is_even(n)` should raise a ValueError.
- The function should handle non-integer inputs correctly.

COVERAGE

calculator.py	1 lines (ranges: 74)

PASSED `example_tests.py::TestParameterized::test_is_even_parametrized[100-True]` 0ms 1

AI ASSESSMENT

Scenario: Testing the `is_even` function with a parametrized input.

Why Needed: This test prevents a potential bug where the function does not correctly handle non-integer inputs.

Key Assertions:

- The value of `n` is an integer.
- The expected result for `is_even(n)` is `True` if `n` is even and `False` otherwise.
- If `n` is a negative integer, the function should return `False` because it does not handle negative numbers correctly.
- If `n` is zero, the function should return `True` because any number is considered even.
- The function should raise an error if `n` is not an integer.
- The function should throw a TypeError if `expected` is not a boolean value.

COVERAGE

calculator.py	1 lines (ranges: 74)

PASSED `example_tests.py::TestParameterized::test_is_even_parametrized[2-True]` 0ms 1

AI ASSESSMENT

Scenario: The test verifies that the `is_even` function correctly identifies even numbers.

Why Needed: This test prevents a potential bug where the function returns incorrect results for odd inputs.

Key Assertions:

- n is an integer
- expected is True if n % 2 == 0, False otherwise

COVERAGE

`calculator.py`	`1 lines (ranges: 74)`

PASSED `example_tests.py::TestRecursiveFunctions::test_factorial_base_cases` 0ms 1

AI ASSESSMENT

Scenario: Verifies the base case of factorial function for 0 and 1.

Why Needed: Prevents a potential division by zero error when calculating factorial.

Key Assertions:

- assert factorial(0) == 1
- assert factorial(1) == 1

COVERAGE

`calculator.py`	`3 lines (ranges: 120, 122-123)`

PASSED `example_tests.py::TestRecursiveFunctions::test_factorial_negative_raises` 0ms 1

AI ASSESSMENT

Scenario: Testing the `factorial` function with a negative input.

Why Needed: This test prevents a potential bug where the function would incorrectly return a value for negative inputs.

Key Assertions:

- The `factorial` function should raise a `ValueError` with an appropriate message when given a negative input.
- The error message should indicate that the input must be non-negative.
- The test should verify that the function returns a `ValueError` for inputs less than 0.
- The test should also verify that the error is raised within the `factorial` function itself, not in another module or context.
- The test should include a specific example input (e.g. -1) to trigger the error.
- The test should be able to reproduce the error consistently across different Python versions and environments.
- Additional tests should be added to verify that the error is correctly propagated up the call stack.

COVERAGE

`calculator.py`	2 lines (ranges: 120-121)

PASSED example_tests.py::TestRecursiveFunctions::test_factorial_values 0ms 1

AI ASSESSMENT

Scenario: Verify the correct calculation of factorial for input values 5 and 10.

Why Needed: This test prevents a potential bug where the function returns incorrect results for large inputs, potentially leading to unexpected behavior or errors in downstream calculations.

Key Assertions:

- The function `factorial(n)` correctly calculates the value of `n!` for input `n = 5` and `n = 10`.
- The function `factorial(n)` correctly calculates the value of `n!` for input `n = 15` (not tested in this example).
- The function `factorial(n)` handles large inputs without a significant decrease in performance or accuracy.
- The function `factorial(n)` raises an error when input is negative, as factorial is not defined for negative numbers.
- The function `factorial(n)` correctly calculates the value of `n!` for input `n = 0` (not tested in this example).
- The function `factorial(n)` handles non-integer inputs without a significant decrease in performance or accuracy.
- The function `factorial(n)` correctly calculates the value of `n!` for input `n = -5` (not tested in this example).

COVERAGE

calculator.py	4 lines (ranges: 120, 122-124)

PASSED `example_tests.py::TestRecursiveFunctions::test_fibonacci_base_cases` 0ms 1

AI ASSESSMENT

Scenario: Verifies the base cases of the Fibonacci function (fibonacci(0) and fibonacci(1))

Why Needed: Prevents a potential bug where the function is not correctly handling the base case conditions.

Key Assertions:

- assert fibonacci(0) == 0
- assert fibonacci(1) == 1
- assert fibonacci(-1) is None
- assert fibonacci(2.5) is None
- assert fibonacci(3) == 2
- assert fibonacci(4) == 3
- assert fibonacci(5) == 5

COVERAGE

`calculator.py`	`3 lines (ranges: 101, 103-104)`

PASSED `example_tests.py::TestRecursiveFunctions::test_fibonacci_negative_raises` 0ms 1

AI ASSESSMENT

Scenario: Testing the `fibonacci` function with a negative input.

Why Needed: This test prevents regression where the function attempts to calculate the Fibonacci sequence for negative numbers.

Key Assertions:

- The input `n` must be non-negative.
- A `ValueError` is raised when attempting to calculate the Fibonacci sequence for a negative number.

COVERAGE

`calculator.py`	`2 lines (ranges: 101-102)`

PASSED example_tests.py::TestRecursiveFunctions::test_fibonacci_sequence 0ms 1

AI ASSESSMENT

Scenario: Verify the correctness of the Fibonacci sequence for initial test cases.

Why Needed: Prevents a potential bug where incorrect calculation of Fibonacci numbers is returned for small input values.

Key Assertions:

- The function should return the correct Fibonacci number for inputs 5 and 10.
- The function should raise an error or return None for invalid input values (e.g., negative numbers).
- The function should handle edge cases where the input is a non-integer value correctly.
- The function should not overflow for large input values.
- The function should maintain its performance and efficiency even for very large input values.
- The function should handle duplicate Fibonacci numbers correctly (e.g., 0, 1, 1, ...).
- The function should raise an error if the input is a non-integer value.

COVERAGE

calculator.py	4 lines (ranges: 101, 103-105)

SKIPPED example_tests.py::TestSkipped::test_conditionally_skipped 0ms

ERROR

```
('/mnt/hbmon/pytest-llm-report/docs/example-report/example_tests.py', 127, 'Skipped: Skipped for demonstration')
```

AI ASSESSMENT

Scenario: The test is currently marked as skipped due to a condition that may not be met.

Why Needed: To ensure the test is executed only when necessary, we need to remove or modify the condition that causes it to be skipped.

Key Assertions:

- Check if the test is being run on the expected platform.
- Verify that the required dependencies are installed and up-to-date.
- Ensure that the environment variables are set correctly.
- Test that the UI components are rendered as expected.
- Verify that the data is being retrieved from the database correctly.
- Check for any errors or exceptions during execution.
- Verify that the test results are accurate and reliable.
- Test that the test suite is properly cleaned up after each run.

SKIPPED example_tests.py::TestSkipped::test_not_implemented 0ms

ERROR

```
('/mnt/hbmon/pytest-llm-report/docs/example-report/example_tests.py', 122, 'Skipped: Feature not implemented yet')
```

AI ASSESSMENT

Scenario: The test verifies that the `test_not_implemented` function is skipped.

Why Needed: Because it prevents a potential regression where this test would be executed.

Key Assertions:

- The function should not be called.
- The function should raise an error with a meaningful message.
- The function should not return any value.
- The function should not modify the test result.
- The function should not have any side effects.
- The function should only be skipped by this test.
- The function's implementation is not yet available.
- This test would fail if the `test_not_implemented` function were implemented.