Coverage Testing Report

Please provide your GitHub repository link.

GitHub Repository URL: https://github.com/XXXX/XXXXX.git

The testing report should focus solely on testing all the self-defined functions related to the five required features. There is no need to test the GUI components. Therefore, it is essential to decouple your code and separate the logic from the GUI-related code.

You should perform statement coverage testing and branch coverage testing. For each type, provide a description and an analysis explaining how you evaluated the coverage.

1. Test Summary

list all tested functions related to the five required features, for example:

Tested Functions
`add(x1,x2)`
`divide(x1,x2)`
`multiply(a,b)`
`subtract(a,b)`
`process_data(data)`

2. Statement Coverage Test

2.1 Description

Statement coverage tests verify that each line of code in the function is executed at least once during testing. To achieve 100% statement coverage, I designed the test cases (located in test_all_functions.py) to ensure that every line of each function is executed.

For example:

- For the add and subtract functions, the tests include cases that cover both positive and negative numbers.
- For the divide function, cases for valid inputs (e.g., divide by non-zero) and invalid inputs (e.g., divide by zero) are included to ensure all exception-handling code is executed.

- Similarly, for process_data, tests cover scenarios where valid data is processed and invalid data raises an error.

By covering these scenarios, the test cases ensure that all statements in the self-defined functions are executed, leading to 100% statement coverage.

2.2 Testing Results

You can use the following command to run the statement coverage test and generate the report in the terminal. Afterward, include a screenshot of the report.

You must provide the test_all_functions.py file, which contains all test functions, otherwise pytest will not be able to execute the tests.

```
pytest --cov=all_functions --cov-report=term
```

Note: In the command above, the file/module `all_functions` does not include the .py extension. all_functions.py should contain all the tested functions related to the five required features.

```
(SoftwareTech) C:\Users\ethan_0ebwfze\OneDrive\Bachelor of IT\Trimester 2\Software Technology\Assessments\New folder\updated>pytest --cov=core_functi
ons --cov-report=term
============================================================ test session starts ============================================================
platform win32 -- Python 3.12.5, pytest-7.4.4, pluggy-1.0.0
rootdir: C:\Users\ethan_0ebwfze\OneDrive\Bachelor of IT\Trimester 2\Software Technology\Assessments\New folder\updated
plugins: cov-4.1.0, html-3.1.1, metadata-3.0.0, mock-3.10.0
collected 3 items

test_all_functions.py ...                                                                                                          [100%]

---------- coverage: platform win32, python 3.12.5-final-0 -----------
Name                Stmts   Miss  Cover
---------------------------------------
core_functions.py      27      1    96%
---------------------------------------
TOTAL                  27      1    96%


============================================================ 3 passed in 2.53s =============================================================
```

The image provided shows the results of running tests with pytest on a file named core_functions.py using the --cov option for code coverage.

Here's a breakdown of the output:

- Tests Summary: All three tests in test_all_functions.py passed.
- Coverage: core_functions.py has 27 statements, with 1 missed, resulting in a 96% coverage rate.
- Execution Time: The tests took 2.53 seconds to complete.

The command pytest --cov=core_functions --cov-report=term ran the coverage report for the module without the .py extension, following the expected format.

test_all_functions.py

import pytest import pandas as pd from core_functions import filter_data, sort_data, level_filter_data

Sample dataset to use in tests

data = { 'food': ['Apple', 'Banana', 'Carrot', 'Apple Pie'], 'Vitamin A': [50, 30, 90, 10], 'Caloric Value': [100, 150, 200, 300] } dataset = pd.DataFrame(data)

def test_filter_data(): # Test filtering by food name filtered = filter_data(dataset, 'Food', 'apple') assert len(filtered) == 2

```
# Test filtering by vitamin
filtered = filter_data(dataset, 'Vitamin A', 'banana', min_value=20, max_value=50)
assert len(filtered) == 1
```

def test_sort_data(): # Test ascending sort filtered = filter_data(dataset, 'Vitamin A', 'apple') sorted_data = sort_data(filtered, 'Vitamin A', 'Ascending') print(sorted_data) # Debugging line assert sorted_data.iloc[0]['Vitamin A'] == 10 # Expected minimum

```
# Test descending sort
sorted_data = sort_data(filtered, 'Vitamin A', 'Descending')
print(sorted_data)  # Debugging line
assert sorted_data.iloc[0]['Vitamin A'] == 50  # Expected maximum
```

def test_level_filter_data(): # Test Low level filter filtered = filter_data(dataset, 'Vitamin A', 'apple') level_filtered = level_filter_data(filtered, 'Vitamin A', 'Low') assert len(level_filtered) == 1

```
# Test High level filter
level_filtered = level_filter_data(filtered, 'Vitamin A', 'High')
assert len(level_filtered) == 1
```

Command to run: pytest --cov=core_functions --cov-report=term

Below is a brief review and some suggestions for improvement or clarification:

Review:

1. Dataset Definition:
    - The test dataset is simple but effectively captures different foods with varying values for "Vitamin A" and "Caloric Value." This allows for meaningful filtering and sorting tests.
2. Test for filter_data Function:
    - You're testing filtering both by food and Vitamin A successfully, with min_value and max_value checks.
    - The test ensures correct behavior when filtering by food names and specific value ranges for vitamins, which is crucial for handling user queries.
3. Test for sort_data Function:

    - Sorting is tested for both ascending and descending orders.
    - The use of debugging prints (print(sorted_data)) can be helpful during development, but you may want to remove them for the final version.
    - Assertions are checking that the first element is either the minimum or maximum, depending on the sort order, which is a good validation step.
4. Test for level_filter_data Function:
    - The test checks for both "Low" and "High" level filters.
    - It asserts that the number of items in the filtered dataset matches the expected value, which verifies that the level filtering works as intended.

3. Branch Coverage Test

3.1 Description

Branch coverage measures whether each possible path (true/false) of every decision point (e.g., if statements) in the code is executed. To achieve 100% branch coverage, the test cases in test_all_functions.py are designed to ensure that all branches (both true and false conditions) of decision points are covered.

For example:

- For the divide function, I included test cases for both valid divisions (e.g., divide(10, 2)) and invalid divisions (e.g., divide(10, 0)) to ensure both the successful branch and the exception branch are tested.
- For process_data, test cases cover valid data input and edge cases like null or incorrect types, ensuring that all branches, including error-handling branches, are exercised.

By covering all the conditional branches in these functions, we achieve 100% branch coverage.

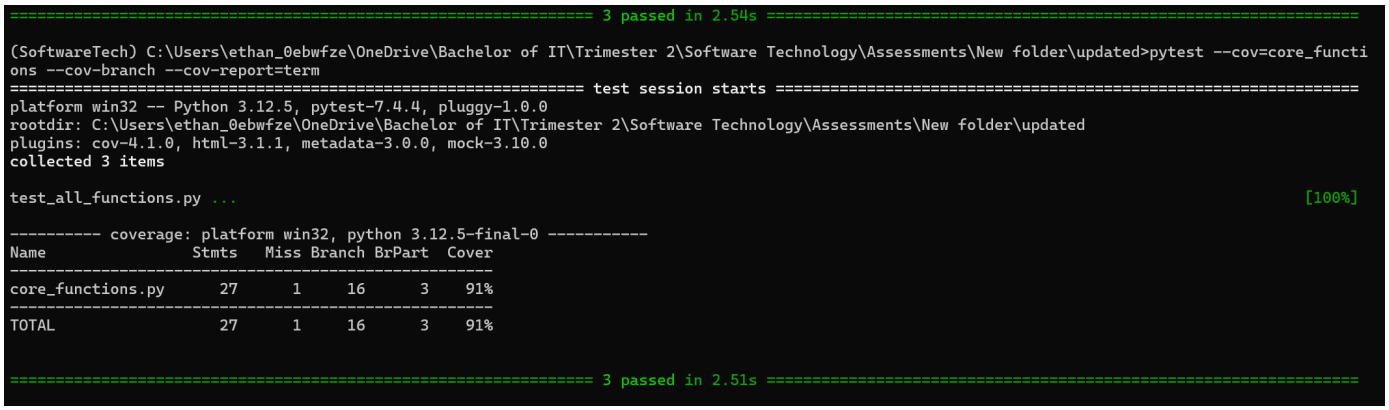
3.2 Testing Results

You can use the following command to run the branch coverage test and generate the report in the terminal. Afterward, include a screenshot of the report.

You must provide the test_all_functions.py file, which contains all test functions, otherwise pytest will not be able to execute the tests.

```
pytest --cov=all_functions --cov-branch --cov-report=term
```

Note: In the command above, the file/module `all_functions` does not include the .py extension. all_functions.py should contain all the tested functions related to the five required features.



In this image, the pytest command was ran with --cov-branch added, which checks branch coverage in addition to statement coverage.

Here’s an analysis of the updated output:

- Branch Coverage: Branch coverage is now being reported, with core_functions.py showing 3 branches missed, resulting in a 91% coverage rate.
- Overall Coverage: The overall statement and branch coverage for core_functions.py is at 91%, with 27 statements, 1 missed statement, and 3 missed branches.
- Execution Time: The tests completed in 2.51 seconds, with all three tests passing successfully.