Assignment 4: Testing

Streamlit App Link: https://brantbremer-cs4090-assignment-4-srcapp-e6dxfb.streamlit.app/

1. Unit Testing:

- **Goal:** Test all major functions in tasks.py to ensure correct behavior for valid and invalid inputs.
- **Tools Used:**
  - pytest for running tests
  - pytest-cov for measuring code coverage
- **Strategy:**
  - Identify critical functions (e.g., add_task, delete_task, update_task, etc.)
  - Write unit tests for each function
  - Include both normal cases and edge cases
  - Mock external dependencies (if any)
  - Use parameterized tests where the same logic is tested with multiple inputs
- **Coverage:**
  - Used pytest --cov to verify at least 90% line coverage.
  - Generated a detailed HTML report with --cov-report=html.
- **Validation:**
  - Ensured all unit tests pass
  - Ensured code coverage report shows 90%+ coverage
  - Reviewed uncovered lines and added additional tests as necessary

2. Bug Reporting and Fixing:

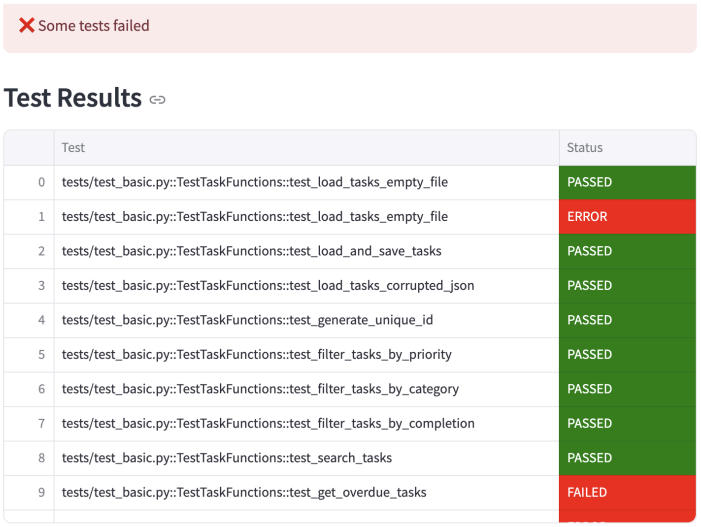
Bug 1: Temporary File Deletion Error

- **Summary**: After running a test that used a temporary file, a FileNotFoundError occurred when the test attempted to delete the file.
- **Location**: tests/test_basic.py — temp_tasks_file fixture
- **Error Message**: FileNotFoundError: [Errno 2] No such file or directory: '/path/to/tempfile.json'
- **Cause**: The temporary file may have already been deleted by the test, causing os.unlink() to fail.
- **Fix**: Wrapped the os.unlink() call in a try-except FileNotFoundError block to handle missing files gracefully.

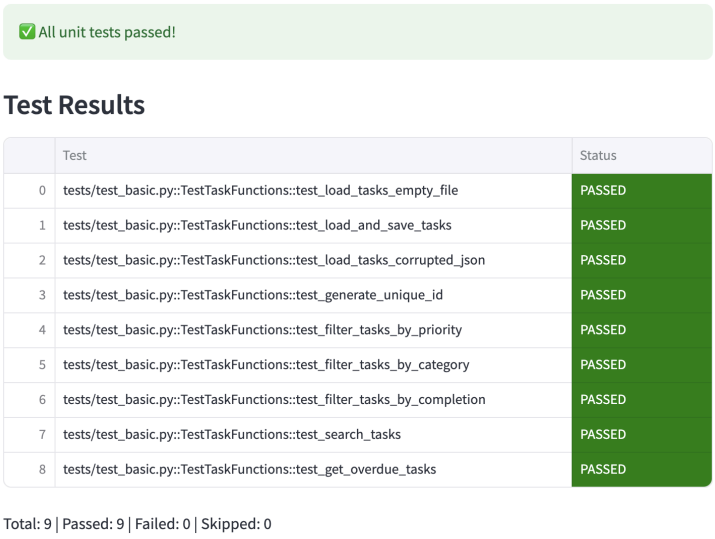
Bug 2: Incorrect Overdue Task Detection

- **Summary**: The get_overdue_tasks function incorrectly marked tasks without a due date as overdue.
- **Location**: src/tasks.py — get_overdue_tasks function
- **Error Message**: AssertionError: assert 2 == 1
- **Cause**: Tasks without a due_date field were still being processed, incorrectly increasing the count of overdue tasks.
- **Fix**: Modified the function to continue and skip any tasks without a due_date.

Before Bug Fixes:



After Bug Fixes:



4. Do Test-Driven Development (TDD):

Test Failure:

Test-Driven Development (TDD)

TDD tests demonstrate the process of developing features by writing tests first. Click the button below to run TDD tests.

Run TDD Tests

❌ Some TDD tests failed

```
============================= test session starts ==============================
platform darwin -- Python 3.13.0, pytest-8.3.5, pluggy-1.5.0 -- /Library/Framework
cachedir: .pytest_cache
metadata: {'Python': '3.13.0', 'Platform': 'macOS-15.3-arm64-arm-64bit-Mach-O', 'P
rootdir: /Users/brantbremer/Computer_Science/Capstone1_4090/cs4090-assignment-4
plugins: html-4.1.1, metadata-3.1.1, bdd-8.1.0, mock-3.14.0, cov-6.1.1, xdist-3.6.
collecting ... collected 0 items / 1 error

==================================== ERRORS ====================================
_____________________ ERROR collecting tests/test_tdd.py ______________________
ImportError while importing test module '/Users/brantbremer/Computer_Science/Capst
Hint: make sure your test modules/packages have valid Python names.
Traceback:
/Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/importlib/__init
    return _bootstrap._gcd_import(name[level:], package, level)
tests/test_tdd.py:6: in <module>
    from src.tasks import add_tags_to_task, get_tasks_by_tag
E   ImportError: cannot import name 'add_tags_to_task' from 'src.tasks' (/Users/br
=========================== short test summary info ============================
ERROR tests/test_tdd.py
!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!
```

Test Success:

✅ All TDD tests passed!

```
============================= test session starts ==============================
platform darwin -- Python 3.13.0, pytest-8.3.5, pluggy-1.5.0 -- /Library/Framework
cachedir: .pytest_cache
metadata: {'Python': '3.13.0', 'Platform': 'macOS-15.3-arm64-arm-64bit-Mach-O', 'P
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase(PosixPath('
rootdir: /Users/brantbremer/Computer_Science/Capstone1_4090/cs4090-assignment-4
plugins: html-4.1.1, metadata-3.1.1, hypothesis-6.131.9, bdd-8.1.0, mock-3.14.0, c
collecting ... collected 14 items

tests/test_tdd.py::TestTasks::test_add_subtask PASSED                    [  7%]
tests/test_tdd.py::TestTasks::test_add_tags_to_task PASSED               [ 14%]
tests/test_tdd.py::TestTasks::test_complete_subtask PASSED               [ 21%]
tests/test_tdd.py::TestTasks::test_filter_tasks_by_category PASSED       [ 28%]
tests/test_tdd.py::TestTasks::test_filter_tasks_by_completion PASSED     [ 35%]
tests/test_tdd.py::TestTasks::test_filter_tasks_by_priority PASSED       [ 42%]
tests/test_tdd.py::TestTasks::test_generate_next_occurrence PASSED       [ 50%]
tests/test_tdd.py::TestTasks::test_generate_unique_id PASSED             [ 57%]
tests/test_tdd.py::TestTasks::test_get_next_occurrence_date PASSED       [ 64%]
tests/test_tdd.py::TestTasks::test_get_overdue_tasks PASSED              [ 71%]
tests/test_tdd.py::TestTasks::test_get_tasks_by_tag PASSED               [ 78%]
tests/test_tdd.py::TestTasks::test_load_and_save_tasks PASSED            [ 85%]
tests/test_tdd.py::TestTasks::test_search_tasks PASSED                   [ 92%]
tests/test_tdd.py::TestTasks::test_set_task_recurrence PASSED            [100%]

============================== 14 passed in 0.09s ==============================
```

Feature 1: Task Tags/Labels

1. Initial Test Creation
    - I began by writing tests to verify the behavior of task tagging. The tests covered:
        - Adding tags to a task (test_add_tags_to_task).
        - Adding tags to a task that already has tags (test_add_tags_to_task_with_existing_tags).
        - Filtering tasks by a specific tag (test_get_tasks_by_tag).
2. Test Failure Demonstration
    - Initially, running the tests resulted in failures since the functionality to add and retrieve tags was not implemented yet.
3. Feature Implementation
    - I implemented the add_tags_to_task and get_tasks_by_tag functions in the src/tasks.py file.
        - add_tags_to_task: Adds tags to a task, ensuring no duplicate tags are added.
        - get_tasks_by_tag: Filters and returns tasks containing the specified tag.
4. Test Passing Verification
    - After implementing the functions, I re-ran the tests, and they passed successfully. The task could now be tagged and filtered based on tags.
5. Refactoring
    - No significant refactoring was required for this feature as the implementation was straightforward.

Feature 2: Task Subtasks

1. Initial Test Creation
    - I created tests to ensure the subtask functionality was working correctly:
        - Adding a subtask to a task (test_add_subtask).
        - Marking a subtask as complete (test_complete_subtask).
2. Test Failure Demonstration
    - Initially, the tests failed because the functionality to add subtasks and mark them as complete did not exist.
3. Feature Implementation
    - I implemented the add_subtask and complete_subtask functions in src/tasks.py.
        - add_subtask: Adds a subtask to a task, generating a unique ID for each subtask.
        - complete_subtask: Marks a subtask as completed by its ID.
4. Test Passing Verification

    - Once the functions were implemented, I ran the tests again. They passed, confirming that tasks could now have subtasks and their completion status could be updated.
5. Refactoring
    - The implementation was simple, and no refactoring was required. The functions were kept clean and focused on their tasks.

Feature 3: Task Recurrence

1. Initial Test Creation
    - I wrote tests to handle task recurrence, which included:
        - Setting recurrence on a task (test_set_recurrence).
        - Calculating the next occurrence date for a recurring task (test_get_next_occurrence_date).
        - Generating the next occurrence for a recurring task (test_generate_next_occurrence).
2. Test Failure Demonstration
    - Before implementation, the tests failed because recurrence logic was missing, and the task objects did not have recurrence functionality.
3. Feature Implementation
    - I implemented the set_task_recurrence, get_next_occurrence_date, and generate_next_occurrence functions in src/tasks.py.
        - set_task_recurrence: Sets a recurrence pattern (daily, weekly, monthly, yearly) for a task.
        - get_next_occurrence_date: Calculates the next occurrence date based on the task's recurrence pattern.
        - generate_next_occurrence: Creates a new task for the next occurrence, resetting the completion status of subtasks and updating the due date.
4. Test Passing Verification
    - After implementing the features, I ran the tests, and all tests passed successfully. The tasks now correctly handled recurrence, and the next occurrence was calculated and generated.
5. Refactoring
    - The implementation was relatively simple, but some logic was adjusted to handle edge cases (e.g., handling different recurrence patterns and generating unique task IDs). However, no major refactoring was needed.

5. Do Behavior-Driven Development (BDD):

BDD focuses on the behavior of the application from the user's perspective. The tests are written in natural language, and scenarios are described in a format that stakeholders can easily understand.

- **Tests**: I wrote feature files using Gherkin syntax to describe the application's expected behavior for various tasks, such as adding tasks, marking them as completed, filtering by priority, and searching for tasks.

- **Approach**: Each feature file includes multiple scenarios, and I implemented the corresponding steps in Python to ensure the behavior is correct. For example, I defined steps for adding a task, checking if it was added, and verifying its attributes.

- **Tools**: I used **pytest-bdd**, a BDD testing framework, along with pytest for execution. The feature files and step definitions are located in the features folder.

6. Do Property-Based Testing:

Property-based testing verifies that the application behaves correctly under a wide range of inputs by generating random data and checking certain properties of the code. This is helpful in detecting edge cases that might not be covered by regular example-based testing.

- **Tests**: Using **hypothesis**, I created property-based tests that generate random data for tasks (e.g., task titles, priorities, etc.) to ensure that the app can handle various inputs and edge cases.

- **Approach**: I defined properties that tasks should satisfy, such as ensuring tasks with the same title don't exist, or that tasks marked as completed should be updated properly. Hypothesis automatically generates different inputs to test these properties.

- **Tools**: The tests are implemented using pytest and hypothesis. The property-based tests are located in the test/test_property.py file.

Lessons Learned

Unit Testing

Writing unit tests for key functions using pytest ensured core functionality worked as expected, achieving over 90% code coverage. Integrating Streamlit buttons for test execution made testing more accessible for users.

Bug Reporting and Fixing

Documenting and fixing bugs systematically improved the app's stability. Clear before/after evidence confirmed that issues were resolved without introducing new ones.

Pytest Features

I utilized pytest-cov, parameterization, mocking, and HTML reports to enhance testing. These features improved test coverage, made tests more efficient, and provided clear, actionable results.

Test-Driven Development (TDD)

By writing tests before implementation for new features, I ensured the app met functional requirements and passed tests before finalizing the features. TDD guided feature development and improved code quality through refactoring.

Behavior-Driven Development (BDD)

Writing user-centric scenarios in Gherkin syntax ensured features aligned with real-world usage. BDD tests were executed through Streamlit buttons, making the process interactive and user-friendly.

Property-Based Testing

Adding Hypothesis for property-based testing uncovered edge cases and ensured robustness. This bonus testing method added valuable coverage and improved the app's reliability.

General Takeaways

- Automation (via Streamlit buttons) made testing seamless.
- TDD and BDD provided clear functional guidance, improving design.
- Comprehensive testing (unit, BDD, property-based) ensured thorough coverage and app stability.