Bug Reports

Bug #1) app.py ID Creation

Summary

app.py incorrectly creates IDs upon task creation, which can result in ID collisions.

Description

app.py assigns new IDs by taking the number of current tasks and adding one to it, however there is already a function to create IDs in tasks.py that is not being used. This manner of ID creation in app.py can easily create IDs that already exist, as it relies on the wrong thing to create the ID.

Steps to reproduce (include browser)

1. Start the app
2. Ensure there is already at least two tasks in the list
    a. Create new task(s) if necessary
3. Delete a task that is not the newest
4. Create a new task (with any values for name, priority, type, etc.)
5. Streamlit then throws an error about Duplicate keys

Expected result

The deleted task disappears, and after that any newly created task pops up as normal.

Actual result

After deleting a task (that is not the newest) it is impossible to create new tasks, will always be a Duplicate key error thrown.

bad

Due: 2025-04-27 | Priority: High | Category: Other

streamlit.errors.StreamlitDuplicateElementKey: There are multiple elements with the same `key='complete_16'`. To fix this, please make sure that the `key` argument is unique for each element you create.

Traceback:

```
File "C:\Users\djcub\school_code\capstone\hw4\src\app.py", line 127, in <modul
    main()
File "C:\Users\djcub\school_code\capstone\hw4\src\app.py", line 88, in main
    if st.button(
       ^^^^^^^^^^
File "C:\Users\djcub\.virtualenvs\hw4-1FWPs0ss\Lib\site-packages\streamlit\run
    result = non_optional_func(*args, **kwargs)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\djcub\.virtualenvs\hw4-1FWPs0ss\Lib\site-packages\streamlit\ele
    return self.dg._button(
           ^^^^^^^^^^^^^^^^
File "C:\Users\djcub\.virtualenvs\hw4-1FWPs0ss\Lib\site-packages\streamlit\ele
    element_id = compute_and_register_element_id(
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```python
if submit_button and task_title:
    new_task = {
        "id": len(tasks) + 1,
        "title": task_title,
        "description": task_description,
        "priority": task_priority,
        "category": task_category,
        "due_date": task_due_date.strftime("%Y-%m-%d"),
        "completed": False,
        "created_at": datetime.now().strftime("%Y-%m-%d
        %H:%M:%S"),
    }       Imran, 2 weeks ago • init
    tasks.append(new_task)
    save_tasks(tasks)
```

Fix

I replaced the ID creation logic from `len(tasks) + 1` to a simple call to generate_unique_id(tasks) from tasks.py.

```python
if submit_button and task_title:
    new_task = {
        "id": generate_unique_id(tasks),      You, 2 mi
        "title": task_title,
        "description": task_description,
        "priority": task_priority,
        "category": task_category,
        "due_date": task_due_date.strftime("%Y-%m-%d"),
        "completed": False,
        "created_at": datetime.now().strftime("%Y-%m-%d
        %H:%M:%S"),
    }
    tasks.append(new_task)
    save_tasks(tasks)
```

Bug #2) No JSON validation

Summary

There is no validation to ensure that parsable JSON contains data in the correct format for the application.

Description

If you delete fields from tasks in the JSON file, tasks.py does not catch that the data is in the wrong format, causing errors in app.py when non-existent fields are read from.

Steps to reproduce (include browser)

1. Open tasks.json
2. Select a task object
3. Delete a one of it's fields ("priority" for example)
4. Run the streamlit app (streamlit run app.py)
5. Bug appears in webpage

Expected result

Upon loading a JSON file, data format should be checked and a new, empty tasks list should be used instead if there are any format issues.

Actual result

Improperly formatted JSON is loaded, and reads to non-existent fields are done, raising KeyErrors.

Fix

I created a set of key values that are required to be in each task. Then I go through each task in parsed JSON data and ensure that all of these required fields exist. If not, an empty list is returned.

```
try:
    with open(file_path, "r") as f:
        data: list[dict[str, Any]] = json.load(f)

    for task in data:
        assert REQUIRED_FIELDS.issubset(task.keys())

    return data
except (FileNotFoundError, AssertionError):
    return []
```

Bug #3) Wrong number of tasks displayed

Note, this is for a feature I added during TDD.

Summary

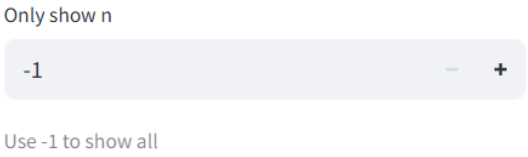
After adding the get_first_n feature, I noticed that it was always displaying one too few tasks.

Description

If there were 5 tasks in the list, for example, only 4 would be displayed.

Steps to reproduce (include browser)

1. Count how many tasks are in you list when you type in -1 for n:



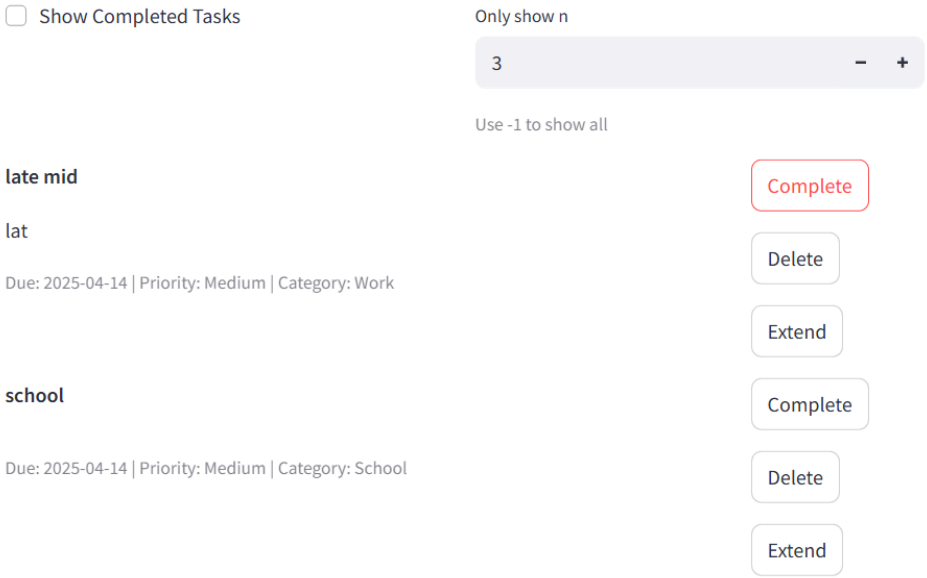
   a.
2. Change n to be the number of tasks you just counted

Expected result

All tasks should still be shown

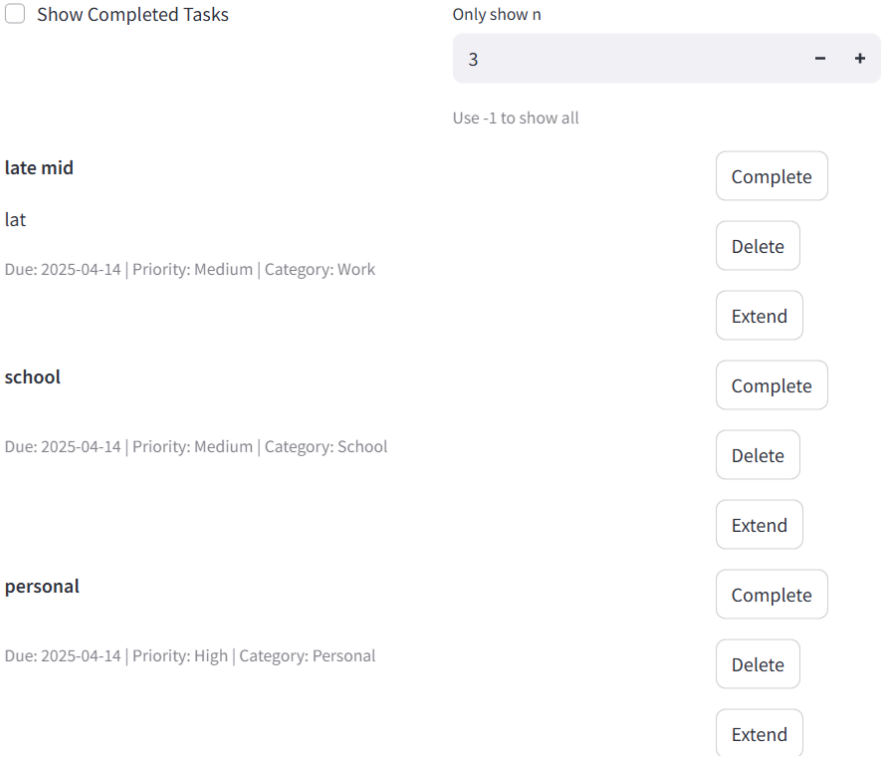
Actual result

One task disappears. Only n-1 tasks are shown



Fix

Turns out it was only an off by one error, because I forgot Python indexes are inclusive at the start, and exclusive at the end.



TDD #1 - extend_task_due_date

Allow the user to extend the due date of already created tasks

Initial test creation

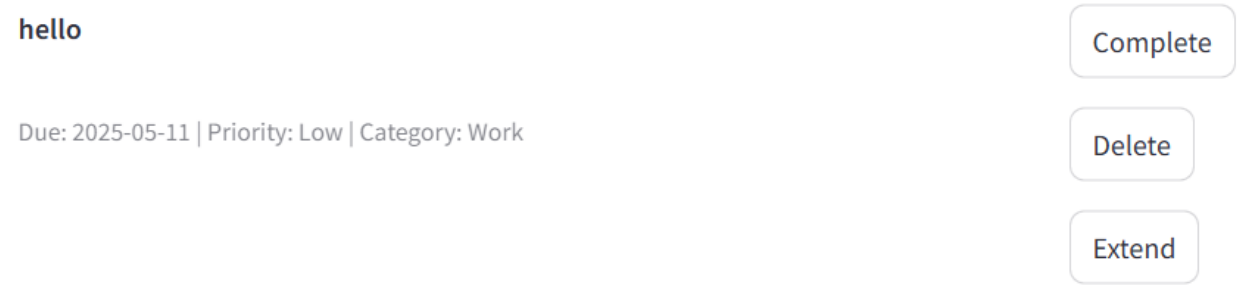
```python
def test_delete_completed_tasks():
    task = {
        "id": 1,
        "title": "First",
        "description": "first desc",
        "priority": "Low",
        "category": "Work",
        "due_date": "2025-04-10",
        "completed": False,
        "created_at": "2025-01-10 17:54:10",
    }

    extend_task_due_date(task, 2)
    assert task.get("due_date") == "2025-04-12"
```

Initial failure

```
tests\test_tdd_feature_1.py F                                            [100%]

================================== FAILURES ===================================
_____________________________ test_delete_completed_tasks _____________________________

    def test_delete_completed_tasks():
        task = {
            "id": 1,
            "title": "First",
            "description": "first desc",
            "priority": "Low",
            "category": "Work",
            "due_date": "2025-04-10",
            "completed": False,
            "created_at": "2025-01-10 17:54:10",
        }

        extend_task_due_date(task, 2)
>       assert task.get("due_date") == "2025-04-12"
E       AssertionError: assert '2025-04-10' == '2025-04-12'
E
E         - 2025-04-12
E         ?         ^
E         + 2025-04-10
E         ?         ^

tests\test_tdd_feature_1.py:19: AssertionError
```

Implementation

```python
def extend_task_due_date(task, days):
    due_date = datetime.strptime(task.get("due_date"), "%Y-%m-%d")
    new_date = due_date + timedelta(days)
    task["due_date"] = new_date.strftime("%Y-%m-%d")
```



Pass Verification

```
$ pytest tests/test_tdd_feature_1.py
============================= test session starts ==============================
platform win32 -- Python 3.11.0, pytest-8.3.5, pluggy-1.5.0
rootdir: C:\Users\djcub\school_code\capstone\hw4
plugins: hypothesis-6.131.0, bdd-8.1.0, cov-6.1.1, html-4.1.1, metadata-3.1.1, mock-3.1
4.0, xdist-3.6.1
collected 1 item

tests\test_tdd_feature_1.py .                                            [100%]

============================== 1 passed in 0.34s ===============================
```

TDD #2 - sort_by_due_date

I want to change the order of tasks where those with the soonest due date are listed first.

Initial test creation

The test makes use of a testing fixture to provide a tasks list.

```python
def test_sort_by_due_date(tasks):
    original_len = len(tasks)
    sorted_tasks = sort_by_due_date(tasks)

    assert len(sorted_tasks) == original_len

    for i in range(original_len - 1):
        assert sorted_tasks[i].get("due_date") < sorted_tasks[i + 1].
        get("due_date")
```

Initial failure

```
$ pytest tests/test_tdd_feature_2.py
============================= test session starts ==============================
platform win32 -- Python 3.11.0, pytest-8.3.5, pluggy-1.5.0
rootdir: C:\Users\djcub\school_code\capstone\hw4
plugins: hypothesis-6.131.0, bdd-8.1.0, cov-6.1.1, html-4.1.1, metadata-3.1.1, mock-3.1
4.0, xdist-3.6.1
collected 1 item

tests\test_tdd_feature_2.py F                                            [100%]

=================================== FAILURES ===================================
_____________________________ test_sort_by_due_date _____________________________

tasks = [{'category': 'Work', 'completed': False, 'created_at': '2025-01-10 17:54:10',
'description': 'first desc', ...}, {'ca....}, {'category': 'School', 'completed': True,
 'created_at': '2025-01-10 17:54:16', 'description': 'the last one', ...}]

    def test_sort_by_due_date(tasks):
        original_len = len(tasks)
        sorted_tasks = sort_by_due_date(tasks)

>       assert len(sorted_tasks) == original_len
E       TypeError: object of type 'NoneType' has no len()

tests\test_tdd_feature_2.py:56: TypeError
```

Implementation

```python
def sort_by_due_date(tasks) -> list:
    """Get the tasks list sort by soonest due date first."""
    return sorted(tasks, key=lambda t: t["due_date"])
```

dfsdf

Due: 2025-04-26 | Priority: Low | Category: Work

pm

Due: 2025-04-27 | Priority: Medium | Category: Personal

dfsdf

Due: 2025-04-28 | Priority: Low | Category: Work

Pass Verification

```
$ pytest tests/test_tdd_feature_2.py
============================= test session starts ==============================
platform win32 -- Python 3.11.0, pytest-8.3.5, pluggy-1.5.0
rootdir: C:\Users\djcub\school_code\capstone\hw4
plugins: hypothesis-6.131.0, bdd-8.1.0, cov-6.1.1, html-4.1.1, metadata-3.1.1, mock-3.1
4.0, xdist-3.6.1
collected 1 item

tests\test_tdd_feature_2.py .                                            [100%]

============================== 1 passed in 0.41s ===============================
```

TDD #3 - get_first_n

Add the ability to just see a shorter list, only showing the first so many tasks. Combined with sort_by_due_date means the user can focus on the immediate tasks and worry less about future ones.

Initial test creation

I made use of test fixtures, just like before. However, I also used parameterization to better catch edge cases. In these tests, the task list `tasks` has 4 elements, so 0, 1, 3, 4, and 5 are all cases that should be under scrutiny.

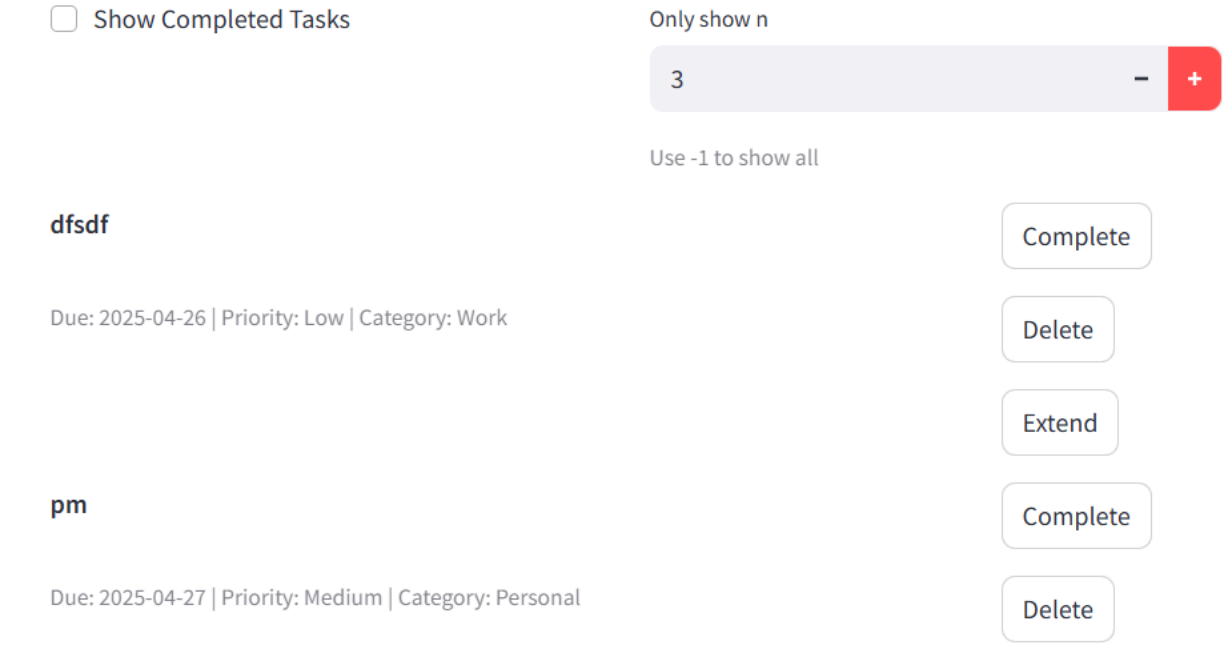
```python
@pytest.mark.parametrize("n", [0, 1, 2, 3])
def test_get_first_n_always_short(tasks, n):
    short_list = get_first_n(tasks, n)
    assert len(short_list) == n


@pytest.mark.parametrize("n", [4, 5, 6])
def test_get_first_n_same_or_longer(tasks, n):
    original_len = len(tasks)
    short_list = get_first_n(tasks, n)
    assert len(short_list) == original_len
```

Initial failure

```
============================= short test summary info ==============================
FAILED tests/test_tdd_feature_3.py::test_get_first_n_always_short[0] - TypeError: objec
t of type 'NoneType' has no len()
FAILED tests/test_tdd_feature_3.py::test_get_first_n_always_short[1] - TypeError: objec
t of type 'NoneType' has no len()
FAILED tests/test_tdd_feature_3.py::test_get_first_n_always_short[2] - TypeError: objec
t of type 'NoneType' has no len()
FAILED tests/test_tdd_feature_3.py::test_get_first_n_always_short[3] - TypeError: objec
t of type 'NoneType' has no len()
FAILED tests/test_tdd_feature_3.py::test_get_first_n_same_or_longer[4] - TypeError: obj
ect of type 'NoneType' has no len()
FAILED tests/test_tdd_feature_3.py::test_get_first_n_same_or_longer[5] - TypeError: obj
ect of type 'NoneType' has no len()
FAILED tests/test_tdd_feature_3.py::test_get_first_n_same_or_longer[6] - TypeError: obj
ect of type 'NoneType' has no len()
============================== 7 failed in 0.72s ===============================
```

Implementation

```python
def get_first_n(tasks, n) -> list:
    """Get a shortened list of tasks."""
    return tasks[:n]        You, 6 minutes
```



Note that when user enters -1, the function is just skipped over entirely and never called.

Pass Verification

```
$ pytest tests/test_tdd_feature_3.py
================================ test session starts ================================
platform win32 -- Python 3.11.0, pytest-8.3.5, pluggy-1.5.0
rootdir: C:\Users\djcub\school_code\capstone\hw4
plugins: hypothesis-6.131.0, bdd-8.1.0, cov-6.1.1, html-4.1.1, metadata-3.1.1, mock-3.1
4.0, xdist-3.6.1
collected 7 items

tests\test_tdd_feature_3.py .......                                          [100%]

================================= 7 passed in 0.55s =================================
```

Methodology

Unit testing

To start, I went though the tasks.py file and looked for the functions that seemed most critical. I read through their code and tried to understand the function's purpose, then wrote some unit tests to check that the function completes that purpose. If a function has branches (if statements), I tried to write a test covering each branch, aiming for the 90% coverage goal. As this is just basic unit testing, I tried to limit all tests to simple cases, as more complex and strategic tests will be written in the next parts.

Parameterized Tests

For coming up with parameterized tests, I went back through my basic tests and found the ones that had to use several assert statements to test each input case for a function, as these could easily be simplified with parameterization. Thus, it seemed to me best to test the filter functions by parameterization. Specifically I wrote parameterized tests for filter_tasks_by_priority and filter_tasks_by_category. Each test had a constant task list with one of each type of task in there, and then the test would take the parameter of the priority or category to filter by.

Fixture Tests

I chose to implement tests using a fixture since I found many of my test functions starting with defining some list of tasks for the specific tests. Those lists of tasks tended to be pretty short and unrealistic (didn't contain all the normal fields). After writing the tests with the fixture I saw the huge benefit that fixtures provide and how it would make my previous basic tests better.

Lessons Learned

There are a variety of things I learned from this assignment. However, I think the most important thing to learn from it is that there is no type of testing that fits all. Different techniques are more useful for different things. For example, I found property based testing most useful for when many different numbers can be used, or when a list can be a variety of sizes. However, for situations where different specific values (like strings) can be used, parameterization was more helpful.

Another lesson I learned was that having coverage over an entire file doesn't necessarily mean you have tested all of its functionality. There may still be specific cases and scenarios where something breaks down and its important to consider the different contexts the code could run with.

I learned a lot about BDD testing specifically. At first I started by just repeating some basic tests I wrote earlier in the assignment, but in the new testing format. Through this I learned how a lot of operations are reused across tests. I also saw how tests can be generalized using pytest_bdd's parsers, and learned that

this way of generalizing is a lot easier to see and understand for humans, than using pytest's parameterization. By the end of it I saw how you can test functionality from many different standpoints using different scenarios in a feature. For example I did this in generate_unique_id.feature to show how the edge case of starting with no existing tasks is meant to work.