🏆 Python Challenge: Build an Expense Tracker with Mocked API Integration

🎯 Challenge

Create a basic Expense Tracker application in Python, using TDD principles to build and test each feature step-by-step. This application will allow users to manage expenses by adding, viewing, updating, and deleting records, but instead of storing data in a file, expenses will be managed through a simulated external API. Use mocking to simulate API responses, isolating tests from real network calls.

📚 Key Learnings

By completing this challenge, you will:

- Practice Test-Driven Development (TDD) by writing tests first and implementing code to satisfy them.
- Gain experience using unittest or pytest along with mocking techniques.
- Learn how to mock API calls using unittest.mock to simulate responses and control test conditions.
- Work with JSON data to structure expense records, simulating a real-world API integration without actual network calls.

👤 User Story

As a user of the Expense Tracker application, I want to manage my expenses by adding, viewing, updating, and deleting them through a command-line interface so that I can keep track of my spending.

✅ Acceptance Criteria

Functionality:

1. **Add an Expense**: Users can add an expense with a description, amount, and date.
2. **View Expenses**: Users can view a list of expenses retrieved from the API.
3. **Update an Expense**: Users can update an existing expense's details.
4. **Delete an Expense**: Users can remove an expense from the list.
5. **API Interaction**: All expense data should be managed through API calls, but the API should be simulated using mocks during testing.

Data Handling:

- Represent each expense as a dictionary with fields for description, amount, and date.
- Use JSON format for data interchange with the mocked API.

TDD and Mocking:

1. **Red-Green-Refactor (TDD)**:
   - Start by writing failing tests for each feature, then implement minimal code to pass each test, refactoring as needed.
2. **Mocking the API**:
   - Use unittest.mock to simulate API responses instead of making actual network calls.
   - Test interactions with the mocked API for functions like adding, viewing, updating, and deleting expenses.

🔄 Example Flow

1. **Step 1**: Write test cases for each feature (add, view, update, delete), using mocking to simulate API calls.
2. **Step 2**: Implement functions for each feature to satisfy the tests, using mocked responses.
3. **Step 3**: Set up API response structures as JSON data within test cases.
4. **Step 4**: Run and refactor the code, ensuring all tests pass with the mocked API.

📦 Sample API Response Structure

Each expense entry could have this structure in API responses:

json

Copy code

```
{
  "description": "Lunch",
  "amount": 10.5,
  "date": "2023-11-01"
}
```

🔄 Mocking API Calls

1. **Mocking GET, POST, PUT, DELETE Requests**:

    - Use unittest.mock.patch to replace API calls in the code with mocked responses for testing.

2. **Example Mock Setup**:
    - Mock requests.get, requests.post, requests.put, and requests.delete to simulate responses without network activity.
    - Control response data directly in each test, simulating various conditions and ensuring isolation from external dependencies.

🔄 **Example Mocking Test Flow**

1. **Setup Mocked Data**:
    - Create sample expense data as JSON to use in tests.
2. **Mock API Calls**:
    - Use patch("requests.get") or similar to simulate API responses for viewing expenses.
3. **Implement and Verify**:
    - Write tests for each function to check if they make the correct API calls and handle the simulated responses appropriately.