Unit Test Guide

What is Unit Testing?

Unit testing is a way to test individual parts (or "units") of your code to ensure they work as expected. In Python, the `unittest` module provides a built-in framework for writing and running tests.

Why Use Unit Tests?

- **Catch bugs early**: Identify issues before deploying your program.
- **Confidence in code changes**: Ensure new updates don't break existing functionality.
- **Reusable tests**: Once written, tests can be reused to verify future changes.

Setting Up Your Tests

1. **Import `unittest` and the Classes to Test**: At the top of your test file, import the `unittest` module and the classes you want to test:

   ```
   import unittest
   from library import Library
   from book import Book
   ```

2. **Create a Test Class**: Your test class must inherit from `unittest.TestCase`. This ensures your test class has access to assertion methods like `assertEqual`, `assertIn`, etc.
3. **Use the `setUp` Method**: The `setUp` method runs **before every test**. Use it to create the objects you'll need for testing.

   ```
   def setUp(self):
       self.library = Library()
       self.book = Book("1234567890", "Sample Title", "Sample Author", 2022)
       self.library.add_book(self.book)
   ```

Writing Test Cases

Each test case should:

- Be a **method** in your test class.
- Start with the word `test_` (e.g., `test_add_book`).

- Focus on **one specific feature or scenario**.

Common Assertion Methods

Assertion	Description
`self.assertEqual(a, b)`	Verifies that `a == b`.
`self.assertNotEqual(a, b)`	Verifies that `a != b`.
`self.assertIn(item, collection)`	Verifies that `item` is in `collection`.
`self.assertNotIn(item, collection)`	Verifies that `item` is not in `collection`.
`self.assertRaises(exception, callable, *args)`	Ensures an exception is raised.

Example Test Case

```
def test_add_book(self):
    book = Book("9876543210", "New Book", "New Author", 2023)
    self.library.add_book(book)
    self.assertIn(book, self.library)  # Check the book was added successfully
```

Running Your Tests

1. Save your test file, e.g., `tests.py`.
2. Run the tests from the command line:

   ```
   python -m unittest tests.py
   ```

3. If all tests pass, you'll see:

   ```
   ....
   ----------------------------------------------------------------------
   Ran 4 tests in 0.002s

   OK
   ```

4. If a test fails, you'll see a detailed error message showing:
   - The test that failed.
   - The reason for failure.

- The expected and actual outputs.

Debugging Failed Tests

When a test fails:

- Read the error message to identify the issue.
- Use `print` statements or a debugger to inspect variable values.
- Fix the code and rerun the tests.

Extending Your Tests

Once your initial tests pass, consider adding:

1. Edge Cases
   :
   - Empty inputs (e.g., an empty library).
   - Invalid data (e.g., a missing ISBN).
2. Performance Tests
   :
   - Adding or searching for many books.
3. Boundary Cases
   :
   - Removing the only book in the library.
   - Adding books with similar titles or authors.

Good Practices

- **Write clear test names**: Use descriptive names to indicate what each test checks.
- **Test incrementally**: Write one test at a time and ensure it passes before moving on.
- **Document your code**: Add comments to explain complex logic in your tests.

Sample Workflow

1. Implement a feature (e.g., `add_book`).
2. Write a test for the feature (e.g., `test_add_book`).
3. Run the test.
    - If it passes: Move to the next feature.
    - If it fails: Debug and fix the issue.
4. Repeat for all features.

Common Issues and Solutions

1. **"No module named library or book"**:
    - Ensure your test file is in the same directory as `library.py` and `book.py`.
    - Ensure correct import statements: `from library import Library`.
2. **"AssertionError: X not equal to Y"**:
    - Check the expected and actual values.
    - Ensure your code correctly implements the functionality being tested.