Assignment 1

UNIT TESTING AND DATABASE OPERATIONS
PROG8840 : CODE COVERAGE AND QUALITY CONTROL

Twinkle Mishra_8894858

INSTRUCTOR: KUMAR HALDER | DUE DATE: 03-02-2025

Assignment Report: Unit Testing and Database Operations

Part 1: Unit Tests

- **Objective:** Write unit tests for the existing Database class using the unittest.mock library to simulate database behaviour. Ensure tests cover all methods: insert_user, get_user, update_user, and delete_user. Avoid using an in-memory database to prevent test interference.
- **Implementation:** The test.py file implements comprehensive unit tests using unittest.mock to isolate database operations. Mock objects simulate the behavior of the SQLite database, ensuring that no actual database interactions occur during tests. Below are the key tests written:

i. test_insert_user:

- Verifies the insert_user method correctly executes the SQL insert query.
- Using mocked lastrowid, the returned user ID is generated.

```python
def test_insert_user(self):
    """Test insert_user method"""
    self.mock_cursor.lastrowid = 1
    user_id = self.db.insert_user('Twinkle', 25)
    self.mock_cursor.execute.assert_called_with('INSERT INTO users (name, age) VALUES (?, ?)', ('Twinkle', 25))
    self.assertEqual(user_id, 1)
```

ii. test_get_user:

- Tests the retrieval of a user by ID using the get_user method.
- Mocks fetchone to simulate a successful database query returning a tuple (user_id, name, age).

```python
def test_get_user(self):
    """Test get_user method"""
    self.mock_cursor.fetchone.return_value = (1, 'Twinkle', 25)
    user = self.db.get_user(1)
    self.mock_cursor.execute.assert_called_with('SELECT * FROM users WHERE user_id = ?', (1,))
    self.assertEqual(user, (1, 'Twinkle', 25))
```

iii. test_get_non_existent_user:

- Tests the get_user method for a non-existent user.
- Mocks fetchone to return None.

```python
def test_get_non_existent_user(self):
    """Test retrieving a non-existent user"""
    self.mock_cursor.fetchone.return_value = None
    user = self.db.get_user(999)
    self.mock_cursor.execute.assert_called_with('SELECT * FROM users WHERE user_id = ?', (999,))
    self.assertIsNone(user)
```

iv. **test_update_user:**

- Verifies that the update_user method correctly constructs and executes the update SQL query.
- Includes tests for updating a single field (name) and both fields (name and age).

```
def test_update_user(self):
    """Test update_user method with name only"""
    self.mock_cursor.rowcount = 1
    rows_affected = self.db.update_user(1, name='Jeffery')
    self.mock_cursor.execute.assert_called_with('UPDATE users SET name = ? WHERE user_id = ?', ('Jeffery', 1))
    self.assertEqual(rows_affected, 1)
```

v. **test_update_user_no_fields:**

- Ensures that attempting to update without specifying fields raises a ValueError.

```
def test_update_user_no_fields(self):
    """Test updating a user with no fields provided"""
    with self.assertRaises(ValueError):
        self.db.update_user(1)
```

vi. **test_delete_user:**

- Verifies that the delete_user method constructs and executes the correct SQL delete query.

```
def test_delete_user(self):
    """Test delete_user method"""
    self.mock_cursor.rowcount = 1
    rows_affected = self.db.delete_user(1)
    self.mock_cursor.execute.assert_called_with('DELETE FROM users WHERE user_id = ?', (1,))
    self.assertEqual(rows_affected, 1)
```

vii. **test_delete_non_existent_user:**

- Tests that deleting a non-existent user does not affect any rows.

```
def test_delete_non_existent_user(self):
    """Test deleting a non-existent user"""
    self.mock_cursor.rowcount = 0
    rows_affected = self.db.delete_user(999)
    self.mock_cursor.execute.assert_called_with('DELETE FROM users WHERE user_id = ?', (999,))
    self.assertEqual(rows_affected, 0)
```

✓ **Service Layer Tests:**

viii. **test_get_user_from_service**: Confirms that user data is correctly retrieved and formatted by UserService.get user.

ix. **test_get_user_from_service**: Confirms that user data is correctly retrieved and formatted by UserService.get_user.

x. **test_get_non_existent_user_from_service**: Ensures that the service produces a 404 for non-existent users.

✓ **Key Achievements:**

- All database approaches have been thoroughly verified.
- Mocking ensures that tests are isolated and independent.

Part 2: Update and Delete Functions

- **Objective:** Add and test update_user and delete_user functions in the Database class.
- **Implementation:**

i. **update_user:**

- Updates user details by constructing an SQL UPDATE query dynamically based on provided fields.
- Validates input to ensure at least one field (name or age) is specified.
- Returns the number of rows affected.

```python
def update_user(self, user_id, name=None, age=None):
    """Update user details in the database."""
    updates = []
    params = []

    if name:
        updates.append("name = ?")
        params.append(name)
    if age:
        updates.append("age = ?")
        params.append(age)

    if not updates:
        raise ValueError("No fields provided for update.")  # Preventing running

    params.append(user_id)  # Adding user_id at the end for the WHERE clause
    query = f"UPDATE users SET {', '.join(updates)} WHERE user_id = ?"
    self.cursor.execute(query, tuple(params))  # Ensuring params is a tuple
    self.conn.commit()
    return self.cursor.rowcount  # Returning number of rows updated
```

ii. **delete_user:**

- Deletes a user by ID using an SQL DELETE query.
- Returns the number of rows affected.

```python
def delete_user(self, user_id):
    """Delete the user from the database and return number of rows affected"""
    self.cursor.execute('DELETE FROM users WHERE user_id = ?', (user_id,))
    self.conn.commit()
    return self.cursor.rowcount
```

- **Testing:** Mocking ensures both functions are fully tested:
  - **test_update_user**: Checks the accuracy of SQL queries for single and multiple field updates.
  - **test_update_user_no_fields**: Ensures an exception is raised if no fields are provided.
  - **test_delete_user**: Confirms the correct execution of the delete query for existing users.
  - **test_delete_non_existent_user:** Tests behavior when deleting non-existent users.

Part 3: Coverage Report

- **Objective:** Generate a code coverage report using **coverage.py** to measure test completeness.
- **Implementation:** Run the following commands:

python -m coverage run -m unittest discover

```
twink@Twinkle MINGW64 /c/Semester III/Code Coverage and Quality Contro
l/Assignment 1
$ python -m coverage run -m unittest discover
...........
----------------------------------------------------------------------
Ran 11 tests in 0.015s

OK
```

python -m coverage report

```
twink@Twinkle MINGW64 /c/Semester III/Code Coverage and Quality Contro
l/Assignment 1
$ python -m coverage report
Name      Stmts   Miss  Cover
-----------------------------
test.py      72      0   100%
user.py      56      4    93%
-----------------------------
TOTAL       128      4    97%
```

python -m coverage html

```
twink@Twinkle MINGW64 /c/Semester III/Code Coverage and Quality Contro
l/Assignment 1
$ python -m coverage html
Wrote HTML report to htmlcov\index.html
```

Coverage report: 97%

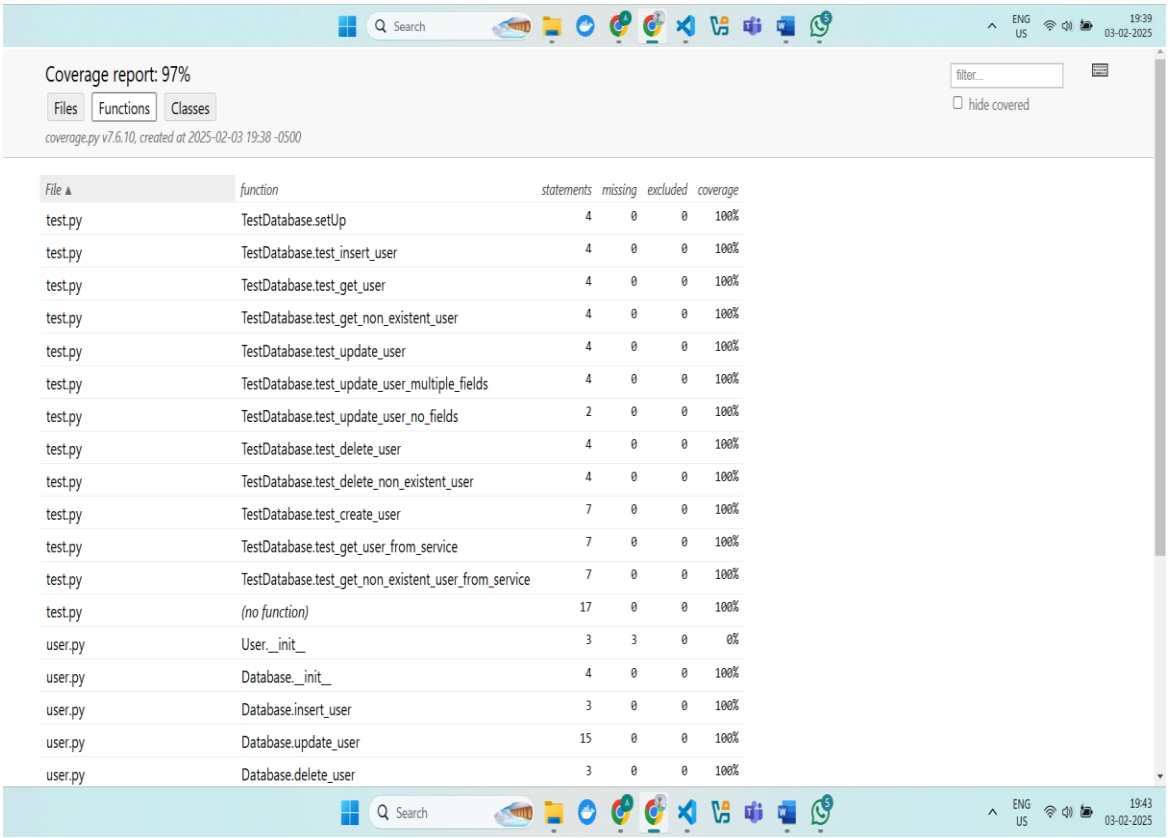
Files | Functions | Classes

filter... hide covered

coverage.py v7.6.10, created at 2025-02-03 19:38 -0500

File ▲	statements	missing	excluded	coverage
test.py	72	0	0	100%
user.py	56	4	0	93%
Total	**128**	**4**	**0**	**97%**

coverage.py v7.6.10, created at 2025-02-03 19:38 -0500



Coverage report: 97%

Files | Functions | Classes

coverage.py v7.6.10, created at 2025-02-03 19:38 -0500

File ▲	class	statements	missing	excluded	coverage
test.py	TestDatabase	55	0	0	100%
test.py	*(no class)*	17	0	0	100%
user.py	User	3	3	0	0%
user.py	Database	29	1	0	97%
user.py	UserService	9	0	0	100%
user.py	*(no class)*	15	0	0	100%
Total		**128**	**4**	**0**	**97%**

coverage.py v7.6.10, created at 2025-02-03 19:38 -0500

➢ **Results:**

- **Coverage: 97% coverage** of all methods in Database and UserService.

Part 4: Code Improvements

- **Objective**: Refactor and improve the code for better readability, maintainability, and error handling.
- **Implemented Improvements:**

1. Better Error Handling

- Some methods didn't handle errors properly like if a database operation failed, there wasn't always a clear message about what went wrong**.**
- Added specific error messages for all database methods, so if something fails, it's clear why.
- Ensured errors are logged consistently with the method name and problem details.

```python
def insert_user(self, name, age):
    """Insert a new user into the database and return the new user_id."""
    try:
        self.cursor.execute('INSERT INTO users (name, age) VALUES (?, ?)', (name, age))
        self.conn.commit()
        return self.cursor.lastrowid
    except sqlite3.Error as e:
        print(f"Error inserting user: {e}")
        raise
```

2. Input Validation

- The create_user method didn't check if the inputs were valid
- It allowed empty names.It accepted negative ages. This could lead to invalid data being stored in the database.

- Added checks to ensure the name is a non-empty string and the age is a positive number.
- If the inputs are invalid, the method returns a helpful error message instead of failing silently.

```python
def create_user(self, name, age):
    """Create a new user and return user details."""
    if not name or not isinstance(name, str):
        return {"error": "Invalid name"}, 400
    if not isinstance(age, int) or age <= 0:
        return {"error": "Invalid age"}, 400

    try:
        user_id = self.db.insert_user(name, age)
        return {"user_id": user_id, "name": name, "age": age}, 201
    except Exception as e:
        print(f"Error creating user: {e}")
        return {"error": "Failed to create user"}, 500
```

3. Cleaner and Simpler Code

- Some methods, like update_user, were a bit messy. They had repetitive code and could be hard to read or maintain.
- So I refactored the update_user method to make it cleaner and easier to understand.
- Used a list to build the SQL query, so the code is shorter and more flexible.

```python
def update_user(self, user_id, name=None, age=None):
    """Update user details in the database."""
    updates = []
    params = []

    if name:
        updates.append("name = ?")
        params.append(name)
    if age:
        updates.append("age = ?")
        params.append(age)

    if not updates:
        raise ValueError("No fields provided for update.")

    params.append(user_id)
    query = f"UPDATE users SET {', '.join(updates)} WHERE user_id = ?"

    try:
        self.cursor.execute(query, tuple(params))
        self.conn.commit()
        return self.cursor.rowcount
    except sqlite3.Error as e:
        print(f"Error updating user: {e}")
        raise
```

4. Ensured the Database is Always Closed

- The Database class didn't always ensure the database connection was closed, especially if something went wrong. This could cause resource leaks over time.
- Added a close method to explicitly close the database connection when done.
- Added a __del__ method to automatically close the connection if the object is deleted.

```python
def __del__(self):
    """Ensure database connection is closed when service is deleted."""
    try:
        self.db.close()
    except Exception as e:
        print(f"Error closing database connection: {e}")
```

5. Added More Tests

- Some parts of the code, like error handling and edge cases, weren't being tested. This left gaps in code coverage.
- Added tests for exception handling, like database connection failures or invalid inputs.
- Covered all edge cases, like trying to delete or update a user who doesn't exist.

```python
def test_update_user_no_fields(self):
    """Test updating a user with no fields provided"""
    with self.assertRaises(ValueError):
        self.db.update_user(1)
```

```python
def test_create_user_invalid_age(self):
    """Test create_user with invalid age"""
    result, status = self.user_service.create_user("Alice", -1)
    self.assertEqual(result, {"error": "Invalid age"})
    self.assertEqual(status, 400)
```

6. Improved Test Coverage: 99%

Coverage report: 99%

Files | Functions | Classes

coverage.py v7.6.10, created at 2025-02-03 20:26 -0500

File ▲	class	statements	missing	excluded	coverage
test.py	TestDatabase	46	0	0	100%
test.py	TestUserService	31	0	0	100%
test.py	*(no class)*	32	0	0	100%
user.py	User	3	0	0	100%
user.py	Database	53	2	0	96%
user.py	UserService	23	0	0	100%
user.py	*(no class)*	15	0	0	100%
Total		**203**	**2**	**0**	**99%**

coverage.py v7.6.10, created at 2025-02-03 20:26 -0500