EXPERIMENT 4

SUNIT SABLE

211070023

BATCH A

ST

AIM: To **design, implement, and test a Library Management System** to ensure its functionality and reliability using automated testing. This includes:

* Developing a program that allows adding, borrowing, and returning books.
* Writing a test suite to validate correct system behavior.
* Running and refining tests to handle edge cases.
* Automating testing using CI/CD pipelines to maintain code quality.

THEORY & IMPLEMENTATION:

**Test Suite Generation for Library Management System**

**Step 1: Implement a Practical Program**

Before generating test suites, we need a program to test. Instead of a simple function, we will create a **Library Management System** that allows users to add books, borrow books, return books, and check book availability.

**Key Features:**

* A user can add a book to the library.
* A user can borrow a book if it is available.
* The system prevents borrowing if the book is unavailable.
* A user can return a borrowed book.
* The system provides information on book availability.

**Code Implementation:**

Create a file named library.py and add the following code:

class Library:

def \_\_init\_\_(self):

"""Initialize a library with an empty book catalog."""

self.books = {}

def add\_book(self, title, copies=1):

"""Add a book to the library with a specified number of copies."""

if copies <= 0:

raise ValueError("Number of copies must be positive.")

self.books[title] = self.books.get(title, 0) + copies

def borrow\_book(self, title):

"""Borrow a book if available."""

if title not in self.books or self.books[title] == 0:

raise ValueError("Book not available.")

self.books[title] -= 1

def return\_book(self, title):

"""Return a borrowed book."""

if title not in self.books:

self.books[title] = 0

self.books[title] += 1

def is\_book\_available(self, title):

"""Check if a book is available."""

return self.books.get(title, 0) > 0

**Step 2: Set Up the Testing Environment**

Now, we need to test the program. We will use unittest, a built-in Python testing framework, to verify that the library behaves as expected.

**How does the test environment work?**

* The test script imports the Library class from library.py.
* It creates test cases to check book addition, borrowing, returning, and error handling.
* The setUp() method initializes a library before each test runs.
* The tests run automatically, and errors are reported.

**Create a Test File:**

Save the following as test\_library.py:

import unittest

from library import Library

class TestLibrary(unittest.TestCase):

def setUp(self):

"""Create a library instance for testing."""

self.library = Library()

self.library.add\_book("Python Programming", 2)

def test\_add\_book(self):

"""Test if adding books increases the count correctly."""

self.library.add\_book("Data Structures", 3)

self.assertTrue(self.library.is\_book\_available("Data Structures"))

def test\_borrow\_book(self):

"""Test if borrowing a book decreases availability."""

self.library.borrow\_book("Python Programming")

self.assertTrue(self.library.is\_book\_available("Python Programming"))

def test\_borrow\_unavailable\_book(self):

"""Test that borrowing an unavailable book raises an error."""

with self.assertRaises(ValueError):

self.library.borrow\_book("Machine Learning")

def test\_return\_book(self):

"""Test if returning a book increases availability."""

self.library.borrow\_book("Python Programming")

self.library.return\_book("Python Programming")

self.assertTrue(self.library.is\_book\_available("Python Programming"))

def test\_negative\_copies\_addition(self):

"""Test that adding a negative number of copies raises an error."""

with self.assertRaises(ValueError):

self.library.add\_book("Algorithms", -1)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Key Concepts:**

* The setUp() method creates a fresh library for each test.
* Each test case checks a specific function of the program.
* Invalid transactions (e.g., borrowing an unavailable book, adding negative copies) are properly handled using assertRaises().

**Step 3: Run the Test Suite**

Now, we execute the test suite to check if everything is working as expected.

**Command to Run Tests:**

python -m unittest test\_library.py

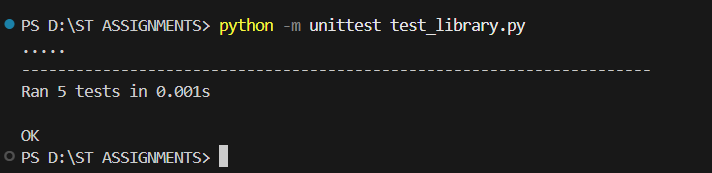
**Expected Output:**

If all tests pass, you'll see something like this:

Ran 5 tests in 0.001s

OK

**My Output:**

****

If a test fails, unittest will indicate the failure and show where the error occurred.

**Step 4: Update the Test Suite with More Cases**

To improve coverage, we should add more test cases for special scenarios:

* Borrowing the last copy of a book (should make it unavailable).
* Returning a book that was never borrowed (should still work).
* Performing multiple borrow and return actions in sequence.

**Updated test\_library.py with new cases:**

def test\_borrow\_last\_copy(self):

"""Test that borrowing the last copy makes the book unavailable."""

self.library.borrow\_book("Python Programming")

self.library.borrow\_book("Python Programming")

self.assertFalse(self.library.is\_book\_available("Python Programming"))

def test\_return\_unborrowed\_book(self):

"""Test that returning a book that wasn't borrowed increases availability."""

self.library.return\_book("Data Science")

self.assertTrue(self.library.is\_book\_available("Data Science"))

def test\_multiple\_transactions(self):

"""Test multiple borrow and return operations in sequence."""

self.library.borrow\_book("Python Programming")

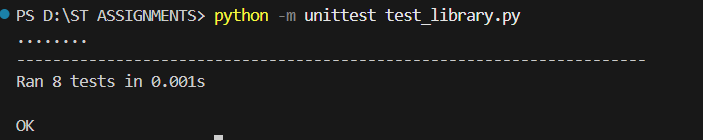
self.library.return\_book("Python Programming")

self.library.borrow\_book("Python Programming")

self.assertTrue(self.library.is\_book\_available("Python Programming"))

**Run the tests again:**

python -m unittest test\_library.py



**Why is updating test cases important?**

* It ensures that new edge cases are covered.
* It increases confidence in the correctness of the program.
* It prevents future code changes from introducing new bugs (regression testing).

**Step 5: Automate Testing with CI/CD**

To ensure that our tests run automatically whenever the code is updated, we can use GitHub Actions.

**Create a GitHub Actions Workflow:**

1. Inside your project, create a folder .github/workflows/.
2. Inside that folder, create a file test.yml and add the following:

name: Run Library Tests

on: [push, pull\_request]

jobs:

test:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v2

- name: Set up Python

uses: actions/setup-python@v2

with:

python-version: '3.9'

- name: Install dependencies

run: pip install pytest

- name: Run tests

run: pytest

**Why use CI/CD for testing?**

* Ensures code is always tested before deployment.
* Prevents merging broken code into the main branch.
* Saves time by automating testing.

CONCLUSION: Through this experiment, we successfully:  
✅ Implemented a **Library Management System** with key functionalities.  
✅ Created a **test suite** using Python’s unittest framework to verify system operations.  
✅ Ran tests and identified **potential issues** (e.g., borrowing unavailable books).  
✅ Enhanced test coverage by adding **edge cases** and verifying system robustness.  
✅ Automated the testing process with **GitHub Actions**, ensuring continuous validation of code updates.

This experiment demonstrates the importance of **automated testing and CI/CD integration** in software development, helping to **maintain software reliability and prevent regressions**.