Pytest and Unittest are both popular Python testing frameworks, each with distinct features and use cases.

Unittest:

* **Built-in:** Unittest is part of the Python Standard Library, meaning no external installation is required.
* **xUnit Style:** It follows the xUnit style of testing, requiring tests to be organized within classes that inherit from unittest.TestCase.
* **Explicit Setup/Teardown:** Uses setUp() and tearDown() methods for test preparation and cleanup.
* **Assertion Methods:** Provides a range of assertion methods (e.g., assertEqual, assertTrue) for verifying test outcomes.
* **Test Discovery:** Includes mechanisms for automatically discovering tests.

Pytest:

* **External Library:**

Requires installation via pip (e.g., pip install pytest).

* **Concise Syntax:**

Allows for writing tests as simple functions, reducing boilerplate code compared to Unittest's class-based structure.

* **Powerful Fixtures:**

Offers a flexible and powerful fixture system for managing test dependencies and state, promoting reusability.

* **Parameterization:**

Enables running the same test with multiple input values efficiently using the @pytest.mark.parametrize decorator.

* **Rich Plugin Ecosystem:**

Supports a wide array of plugins for extended functionality, such as parallel execution, coverage reporting, and integration with other tools.

* **Assertion Introspection:**

Provides detailed failure information without requiring specific assertion methods, making debugging easier.

Choosing between Pytest and Unittest:

* **Choose Unittest if:**

You are working on a legacy project already using Unittest, need compatibility with Python's built-in tools, or prefer a strict, class-based test organization.

* **Choose Pytest if:**

You are starting a new project, desire more concise and readable test code, require powerful features like fixtures and parameterization, need a rich plugin ecosystem, or value ease of use and scalability for larger projects.

It is also possible to use Pytest to run Unittest-based tests, allowing for a gradual transition or mixed approach in projects.

Simple test case:

# banking.py

class InsufficientBalanceError(Exception):

pass

class Account:

def \_\_init\_\_(self, owner, balance=0):

self.owner = owner

self.balance = balance

def deposit(self, amount):

if amount <= 0:

raise ValueError("Deposit must be positive")

self.balance += amount

return self.balance

def withdraw(self, amount):

if amount > self.balance:

raise InsufficientBalanceError("Not enough balance")

self.balance -= amount

return self.balance

def transfer(self, target\_account, amount):

self.withdraw(amount)

target\_account.deposit(amount)

return (self.balance, target\_account.balance)

**Your Tasks:**

Write **Pytest test cases** for the following requirements:

1. **Fixture Setup**
   * Use a Pytest fixture to create two accounts:
     + acc1 with owner "Alice" and balance 1000.
     + acc2 with owner "Bob" and balance 500.
2. **Deposit Tests**
   * Verify deposits increase balance correctly.
   * Ensure depositing 0 or negative values raises ValueError.
3. **Withdraw Tests**
   * Verify withdrawals decrease balance correctly.
   * Ensure withdrawing more than available balance raises InsufficientBalanceError.
4. **Transfer Tests**
   * Verify money transfer reduces balance in source account and increases in target account.
   * Ensure transfer fails if source account has insufficient funds (balances should remain unchanged).
5. **Parameterized Testing**
   * Use @pytest.mark.parametrize to test multiple withdrawal amounts (100, 500, 1000) from acc1.
6. **Exception Message Validation**
   * Check that the exception message "Not enough balance" is returned when overdraft happens.
7. **Edge Case**
   * Create a test where Alice tries to transfer her full balance to Bob. After the transfer:
     + Alice’s balance should be 0.
     + Bob’s balance should increase by Alice’s transferred amount.