Volume 8, Issue 11, pp. 116-120, 2024.

International Journal of Scientific Engineering and Science
ISSN (Online): 2456-7361

Testing and Verification of Complex Systems Using Unittest and Pytest in Python

Kokalko Mykola
Senior Software Engineer at Uvik
Hollywood, FL

Abstract— *This paper discusses the features of using the unittest and pytest libraries for testing and verifying complex systems in the Python programming language. The purpose of the study is to analyze the testing methodology and compare the various approaches to modular and integration testing used by these frameworks. The main focus is on the methods of writing, organizing and executing tests, ensuring flexibility, repeatability and efficiency of testing, as well as the possibility of asynchronous and parameterized testing. The paper describes the advantages and limitations of each of the libraries: unittest provides a strict test structure, which contributes to standardization and increases code readability, which is especially important for team development. At the same time, pytest offers more flexible features due to extensibility, plugin support and parallelization of test execution, which makes it more suitable for complex systems that require integration with external components and faster testing. The analysis also examines examples of using fixtures and plug-ins, such as pytest-mock and pytest-asyncio, which expand testing capabilities. It is concluded that the choice of the tool should be based on the specifics of the project and testing requirements, and for scalable systems and process automation it is preferable to use pytest, while for small and strictly structured projects unittest is suitable.*

Keywords— *Testing, verification, unittest, pytest, Python, unit testing, integration testing, automation, parameterization, asynchronous testing.*

I. Introduction

Software testing and verification are critical stages in the development of complex systems, upon which the stability and reliability of the final product depend. Modern systems comprise numerous interrelated modules, and their correct interaction necessitates a comprehensive approach to testing. Traditional methods, which typically involve manual testing, fail to provide the required level of reliability and do not meet scalability and flexibility demands. Under these circumstances, automated testing tools take precedence, enabling the creation and execution of tests with minimal time and resource investment. Among such tools, the unittest and pytest libraries stand out, widely adopted in Python development due to their functional capabilities and adaptability to diverse testing scenarios.

The relevance of this topic is underscored by the need for a high level of reliability in complex software products used across various sectors, including finance, healthcare, industry, and information technology. Automated testing tools facilitate the timely identification and correction of errors, significantly accelerating the development process and improving the quality of the final product. However, the choice of an appropriate tool depends on the project's characteristics, its structure, and its scope. Unittest and pytest are two popular solutions, each with its advantages and limitations, warranting detailed examination and analysis.

The aim of this study is to analyze the testing methodology using the unittest and pytest libraries, identify their features and comparative advantages, and establish approaches to optimizing testing for complex systems based on project-specific characteristics and development requirements.

II. Materials and Methods

The theoretical aspects of testing and verifying complex systems using unittest and pytest in Python have been explored in the works of authors such as García de la Barrera A., Yuan Z., Erni N., Lukasczyk S., Kroiß F., Fraser G., and Devroey X [1].

Software testing, in turn, is a process of analyzing the functionality of the final product version to assess its compliance with established specifications. Due to the potential inclusion of thousands of lines of code and numerous interconnected components, even a single coding error can trigger cascading failures in other parts of the system. Therefore, it is crucial to conduct testing to verify the program's correct functioning according to specified requirements.

The complexity of modern software solutions necessitates multiple testing levels to verify various aspects of their correctness. According to the ISTQB Certified Test Foundation Level certification program, there are four main levels of testing:
- Unit Testing — assessment of individual lines of code;
- Integration Testing — verification of interaction between individual modules;
- System Testing — testing of the complete system;
- Acceptance Testing — validation against business requirements.

Developing a testing strategy requires a clear understanding of the application domain to be tested. It is essential to determine which parts of the software system should undergo testing. Complete testing of all possible scenarios is unfeasible, so testing priorities should be established based on risk assessment.

Once the areas to be tested are defined, attention can be turned to the characteristics that high-quality unit tests should possess [1].

In Yuan Z.'s study [2], the capabilities and limitations of using ChatGPT for the automatic generation of unit tests are examined in detail. The main objective of this work was to evaluate the quality of generated tests and identify weaknesses in the automated approach. The authors proposed improvement strategies aimed at enhancing the accuracy and completeness of the generated tests through the application of machine learning methods and program code analysis techniques.

Scientific work by Erni N. [3] describes the results of the SBFT (Search-Based and Fuzz Testing) tool competition, emphasizing tasks related to test generation for Python. The primary contribution of the study lies in a comparative analysis of various tools and methods aimed at the automated generation of test cases. The authors assessed the performance and accuracy of each method, which helped identify the most effective approaches.

The publication by Lukasczyk S., Kroiß F., and Fraser G. [4] presents an empirical study on the automated generation of unit tests for Python, focusing on the issues of test incompleteness and non-determinism, and proposes solutions to enhance the quality of automatic generation.

The work of Devroey X. [5] introduces the JUGE infrastructure, designed for comparative analysis of unit test generators in Java. The primary goal of this work was to improve the transparency and reliability of research related to automated test generation and to propose metrics for objective result evaluation.

Practical aspects of testing Python applications are extensively discussed across several electronic resources. Reference [6] covers various approaches to testing CLI applications, including the use of unittest and pytest libraries. Source [7] is a guide to the fundamentals of unit testing, focusing on essential concepts and tools. For a deeper understanding of the capabilities of the pytest framework, "Pytest Beginner's Guide" [8] is recommended, where effective practices for writing and organizing tests with this tool are thoroughly outlined.

Collectively, the reviewed sources indicate substantial progress in the field of software testing automation. The development of tools and methodologies for generating unit tests in Python contributes to improved software quality and greater efficiency in the development process.

III. Results and Discussion

In testing complex systems, the importance lies not only in the correctness of individual components but also in their interaction with each other. Here, testing frameworks for unit testing in Python are examined in detail. The Python unittest module enables structured and efficient testing by providing tools for creating and organizing tests [2].

Unit testing with the unittest library offers a convenient way to verify the correct operation of classes and methods. Consider the example of testing a class BankAccount:

```
import unittest

class BankAccount:
    def __init__(self, id):
        self.id = id
        self.balance = 0

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            return True
        return False

    def deposit(self, amount):
        self.balance += amount
        return True
```

This class allows deposits and withdrawals, with the withdrawal operation limited by the current balance. To ensure the class correctly handles such cases, the following test should be written:

```
class TestBankOperations(unittest.TestCase):
    def test_insufficient_deposit(self):
        # Setup
        account = BankAccount(1)
        account.deposit(100)
        # Action
        result = account.withdraw(200)
        # Verification
        self.assertFalse(result)
```

A TestBankOperations class is then created, inheriting from unittest.TestCase, which allows for the writing of test functions. Within the test function, it is verified that an attempt to withdraw more than the available balance returns False. When an error occurs, a new test is added. If the test fails, the output will be as follows:

```
def test_negative_deposit(self):
    # Setup
    account = BankAccount(1)
    # Action
    result = account.deposit(-100)
    # Verification
    self.assertFalse(result)

FAIL: test_negative_deposit (example.TestBankOperations)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "example.py", line 35, in test_negative_deposit
    self.assertFalse(result)
AssertionError: True is not false
```

This result indicates a failure in handling a negative deposit, signaling a need for code correction. Using unittest, test cases are developed to cover various aspects of module functionality, including boundary tests, exception handling, and performance testing. The structured nature of unittest facilitates the implementation of standardized tests, which is especially valuable in team-based development where a unified testing style is essential.

For example, if a data processing module needs testing, a class can be created with methods to verify the handling of both valid and invalid data, thus ensuring module reliability across different scenarios:

```
import unittest
from data_processor import DataProcessor

class TestDataProcessor(unittest.TestCase):
    def setUp(self):
        self.processor = DataProcessor()

    def test_process_valid_data(self):
        result = self.processor.process("valid_input")
        self.assertEqual(result, "processed_output")

    def test_process_invalid_data(self):
        with self.assertRaises(ValueError):
            self.processor.process("invalid_input")

if __name__ == '__main__':
    unittest.main()
```

In this example, the TestDataProcessor class verifies correct behavior for both valid and invalid data, helping prevent production errors.

When testing interactions among multiple components of a complex system, pytest proves to be a more suitable tool due to its flexibility and support for integration testing. Pytest enables testing of integrations between various modules or services, such as databases, web services, external APIs, or message queues [6].

An example involves testing microservices architecture, where different services interact with one another through APIs. In such systems, it is crucial to test not only the functionality of individual services but also their integration, including data exchange and error handling.

To illustrate Pytest usage, consider a basic example. Pytest automatically identifies test functions in files starting with "test_" or ending in "_test.py." As an example, a function `reverse_text()` is created to reverse a string, and it is tested using `test_reverse_text()`. The test asserts that the output of `reverse_text('python')` should equal 'nohtyp.' If this condition is met, the test passes; otherwise, Pytest reports a failure.

```
# test_reversal.py

def reverse_text(text):
    return text[::-1]

def test_reverse_text():
    assert reverse_text('python') == 'nohtyp'
```

Fixtures in Pytest are intended to create and provide fixed conditions for tests, necessary for their correct execution. These can include, for example, database connections, file system setups, or the preparation of complex objects. Such fixtures are used to enhance the reproducibility and reliability of testing. An example is the fixture `my_fixture`, which returns a list of numbers:

```
import pytest

@pytest.fixture
def my_fixture():
    return [1, 2, 3]

def test_sum(my_fixture):
    assert sum(my_fixture) == 6
```

However, despite the importance of fixtures, their use may be unwarranted in certain cases. This applies to situations where the setup is overly simple, used only once, or when a fixture complicates code readability, reducing clarity. They may also introduce unnecessary dependencies between tests, compromising isolation and complicating debugging [4]. Effective management of fixtures becomes essential in large projects. For example, a `conftest.py` file can contain shared fixtures accessible across multiple tests, improving convenience and reducing code redundancy. It is important to define the scope of a fixture correctly — whether for a single function or an entire module — to optimize the testing process.

In more complex scenarios, such as testing class methods, fixtures play a crucial role [5]. For instance, a `Calculator` class, which includes methods for arithmetic operations, can be tested using Pytest as follows:

```
# calculator.py

class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Cannot divide by zero.")
        return a / b
```

Tests for this class include verifying each operation:

```
# test_calculator.py

import pytest
from calculator import Calculator

@pytest.fixture
def calc():
    return Calculator()

def test_addition(calc):
    assert calc.add(2, 3) == 5

def test_subtraction(calc):
    assert calc.subtract(5, 3) == 2

def test_multiplication(calc):
    assert calc.multiply(3, 4) == 12

def test_division(calc):
    assert calc.divide(8, 2) == 4

def test_division_by_zero(calc):
    with pytest.raises(ValueError):
        calc.divide(10, 0)
```

Practical use of pytest for integration testing may involve fixtures to set up the testing environment, simulating third-party services with mock objects, and using plugins for database testing.

An example fixture for database setup in tests:

```
import pytest
from myapp import create_app, db

@pytest.fixture(scope='module')
def test_client():
    app = create_app()
    with app.test_client() as client:
        with app.app_context():
            db.create_all()
        yield client
        with app.app_context():
            db.drop_all()

def test_database_integration(test_client):
    response = test_client.post('/create_record',
data=dict(name="Test"))
    assert response.status_code == 200
    assert b"Record created" in response.data
```

In this example, the `test_client` fixture creates a test environment with a database where API integration with the database can be verified, as well as the accuracy of record handling. This approach allows for testing the entire system’s functionality rather than individual components.

One of the significant advantages of pytest is the ability to use parameterized tests. In real-world systems, especially complex ones, it is necessary to test a wide range of inputs and their effects on the system. For example, testing a payment system might involve verifying different currencies, payment methods, transaction statuses, etc.

An example of using parameterization in pytest to check different data sets:

```
import pytest
from payment_processor import PaymentProcessor

@pytest.mark.parametrize("payment_method, amount,
expected_status", [
    ("credit_card", 100, "approved"),
    ("debit_card", 50, "approved"),
    ("crypto", 200, "pending"),
])
def test_process_payment(payment_method, amount,
expected_status):
    processor = PaymentProcessor()
    status = processor.process(payment_method, amount)
    assert status == expected_status
```

In this case, a single test function evaluates different payment processing scenarios, significantly reducing code volume and increasing test coverage.

Complex systems often use multithreading or asynchronous processes to enhance performance. Pytest supports testing asynchronous functions via the pytest-asyncio plugin, which is especially useful for testing web services that rely on asynchronous requests or systems interacting with external APIs [7].

Example of testing asynchronous functions with pytest:

```
import pytest
import asyncio
from myasyncapp import async_process_data

@pytest.mark.asyncio
async def test_async_data_processing():
    result = await async_process_data("input_data")
    assert result == "processed_output"
```

The main distinction between unittest and pytest lies in flexibility and extensibility. While unittest is more traditional and structured, pytest offers greater freedom in organizing tests. For instance, pytest allows for test writing based on advanced data structures and context managers, enabling more accurate modeling of complex scenarios and conditions.

Additionally, pytest supports an extensive plugin ecosystem, including pytest-django, pytest-flask, and pytest-mock, making it more suitable for verifying complex systems involving databases, networks, or external services. This enables the creation of more detailed and precise tests that cover interactions with various system components [8].

Example of using a fixture to test database interactions:

```
import pytest
from myapp import create_app, db

@pytest.fixture(scope='module')
def test_client():
    app = create_app()
    with app.test_client() as client:
        with app.app_context():
            db.create_all()
        yield client
        with app.app_context():
            db.drop_all()
```

IV. Conclusion

The analysis of using unittest and pytest libraries for testing and verifying complex systems in Python has confirmed their relevance and importance for modern development processes. This study has identified the key features and differences between these tools. Unittest is characterized by its structured format, making it ideal for standardizing tests in team-based projects where adherence to a unified development style is required. It provides tools for creating unit tests, ensuring a high degree of isolation and test repeatability.

Conversely, pytest has demonstrated its effectiveness in offering flexibility and extensibility. The ability to use plugins, such as pytest-asyncio for asynchronous function testing or pytest-xdist for test parallelization, enables adaptation of testing to specific project requirements. With its support for parameterization and user-friendly syntax, pytest accelerates the process of writing and executing tests, which is especially crucial when working with large, complex systems.

Thus, the choice between unittest and pytest should be based on the project's specific features and needs. Unittest is preferable for smaller, well-structured projects that require strict adherence to standards. In contrast, pytest, with its flexibility and integration capability with external tools, is the optimal choice for large, scalable systems that require automation and quick adaptability to changes. The general conclusion is that the proper application of these tools enhances software reliability, reduces development time, and minimizes errors in the final stages of implementation.

References

[1] García de la Barrera A. et al. Quantum software testing: State of the art //Journal of Software: Evolution and Process. – 2023. – Vol. 35. – No. 4. – p. e2419.
[2] Yuan Z. et al. Evaluating and improving chatgpt for unit test generation //Proceedings of the ACM on Software Engineering. - 2024. – Vol. 1. – No. FSE. – pp. 1703-1726.
[3] Erni N. et al. Softool competition 2024-python test case generation track //Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing. – 2024. – pp. 37-40.
[4] Lukasczyk S., Kroiß F., Fraser G. An empirical study of automated unit test generation for Python //Empirical Software Engineering. – 2023. – Vol. 28. – No. 2. – p. 36.
[5] Devroey X. et al. JUGE: An infrastructure for benchmarking Java unit test generator //Software Testing, Verification and Reliability. – 2023. – vol. 33. – No. 3. – p. e1838.
[6] 4 methods for testing Python applications with the command line. [Electronic resource] Access mode: https://habr.com/ru/companies/otus/articles/755460 / (accessed 10/15/2024).
[7] Python Unit Tests: A quick start. [Electronic resource] Access mode: https://habr.com/ru/companies/otus/articles/481806 / (accessed 10/15/2024).
[8] Pytest beginners guide. [Electronic resource] Access mode: https://medium.com/plusteam/pytest-beginners-guide-9fb9451706bf (accessed 10/15/2024).