Software Testing using Pytest

Team Monday

Ahmed Mohamed Zakaria
Sec.1, BN.3
Email: ahmed.elkarashily98@eng-st.cu.edu.eg

Remonda Talaat Eskarous
Sec.1, BN.19
Email: remonda.bastawres99@eng-st.cu.edu.eg

Mohamed Ahmed Mohamed Ahmed
Sec.2, BN:10
Email: mohamed.dardir98@eng-st.cu.edu.eg

Mohamed Shawky Zaky
Sec.2, BN.15
Email: mohamed.sabae99@eng-st.cu.edu.eg

Mohamed Ramzy Helmy
Sec.2, BN.13
Email: mohamed.ibrahim98@eng-st.cu.edu.eg

Contents

1 System Overview

1.1 Software Under Test (SUT)

In this work, we explore the features of **pytest** unit testing tool. For the purpose of showing the true power of **pytest** and how it can scale to a large complex software. Our software under test *(SUT)* is a `distributed contour extraction system`. The software, basically, takes an input video through *input node*, which divides it into frames and sends them to *OTSU consumer node*. Then, the frames are binarized and sent to *collector node*, which collects the binarized frames and sends them to *Contour consumer node*. Then, the contours are extracted from the binarized frames and sent to the *output node*, which dumps the outputs in a file. The software modules communicate with each others through *TCP* sockets, and each module contains a set of functions that are required to obtain its output.

In our experiments, we focus on covering the main broad features of **pytest** on our code. We test multiple functionalities of different modules and use that to express the usage of **pytest** features and how it can scale to such complex functionalities. The main testing scheme uses *TCP* sockets to test nodes functionalities, however some tests, also, use direct function calls, in order to fairly cover the whole *SUT*.

1.2 Code Structure

Now, let's discuss the code structure in a bit of details. The submitted code is structured as follows :

- **back_machine folder** : contains the code for the first three nodes *(input, OTSU and collector)* and folders for input and configuration.
- **front_machine folder** : contains the code for the *contours* and *output* nodes and folders for output files.
- **test_scripts folder** : contains the main written test scripts with test cases and **pytest** functionalities.
- **sys_init.sh** : a shell script for running the system.
- **README.md** : contains software description, installation and usage guidelines.

2 PyTest Overview and Installation

2.1 Tool Overview

Pytest is a Python unit testing framework that makes it easy to write tests, yet scales to support complex functional testing for applications and libraries. In **Pytest**, tests are written within specific files and run through terminal. It, also, has external plugins for many environments.

2.2 Installation

```
python3 -m pip install --user -r requirements.txt
```

2.3 Usage

```
python -m pytest test_scripts/
```

3 Assert Statements

In PyTest, You can assert the correction of test statements using the keyword **assert**.
For example:

3.1 Basic Usage

```
def f():
    return 4

def test_assert():
    x = 4
    assert x == f()
```

In the SUT it has been useful in almost all test cases for example, When testing that the returned list of contours from the contours node is not empty in **test_contours_output()**:

```
def test_contours_output(self):
    ........
    thread.start()
    ........
    ret_message = self.out_socket.recv_pyobj()
    contours = ret_message['contours']

    assert len(contours) > 0, "Contours list are empty"
```

Another example when testing that the returned contours are the same as the expected contours and no data loss is happened when transmitting the data between the nodes as in **test_valid_contour()**

```
def test_valid_contour(self):
    .....
    assert contour1 == contour2
```

4 Assertions explanations

PyTest enables the user to give his own explanation for the failed assertions. This is useful for assertions that compare objects of the user-defined classes.
In the SUT when it is required to assert the 2 objects of class contours are the same. The user can define the explaination of the reason of failure which can be that the 2 lists of contours don't have the same size, or the 2 contours list have the same length but have different values. PyTest enables the user to define the explaination in **conftest.py** file as follows:

```
def pytest_assertrepr_compare(op, left, right):
    .....
    elif (isinstance(left, Contour) and isinstance(right, Contour)
    and op == "=="):
        if(len(left.contour_array) != len(right.contour_array)):
            return [
                "equality of 2 contour lists",
                "Lengths: {} of contour 1 != {} of contour 2".format(
                len(left.contour_array), len(right.contour_array))
            ]
        else:
            return [
                "equality of 2 contour lists",
                "Contour inner values are different"
            ]
```

5 Exceptions & Warnings

PyTest provides different methods to test some unexpected scenarios that throws exceptions and warnings. It helps to ensure that the exceptions and warnings are thrown correctly in the unexpected scenarios.

5.1 Exceptions

PyTest can assert about the expected exceptions using **pytest.raises()** in the context manager. They are used on the SUT to test the failure when passing a non-valid input to the contours node or the otsu node and asserting that nothing will be returned from the contours node or the otsu node and **ZMQError** exception will be thrown in **test_contour_thread_alive() and test_otsu_thread_alive()**

```
def test_contour_thread_alive(self):
    ......
    with pytest.raises(zmq.ZMQError):
        ......
        in_message = { 'binary' : 0.5} #passing not valid input
        ......
        ret_message = self.out_socket.recv_pyobj(flags = zmq.NOBLOCK)
```

It throws a **ZMQError** within the context manager.

5.2 Warnings

In a similar way to asserting that exceptions are thrown. In PyTest you can assert that a specific warning happens using **pytest.warns** inside the SUT
For example:

```
def test_warning():
    with pytest.warns(UserWarning):
        warnings.warn("my warning", UserWarning)
```

In the the case of sending invalid inputs to the otsu and contours node and testing whether there is a UserWarning will show up, is evaluated in **test_contour_user_warning() and test_otsu_user_warning()**, For example:

```
def test_contour_user_warning(self):
    .....
    with pytest.warns(UserWarning):
        .....
        in_message = { 'binary' : 0.5} #sending invalid input
```

6 Fixtures

6.1 Introduction

Software Test Fixtures initialize test functions. They provide a fixed baseline so that tests execute reliably and produce consistent and repeatable results. **PyTest** Fixtures offer dramatic improvements over the classic **xUnit** style of setup/teardown functions:

1. Fixtures are activated by declaring their use from test functions, modules, classes or whole projects.
2. Fixtures are implemented in a modular manner, as each fixture can use other fixtures.
3. Fixture management scales from simple unit to complex functional testing, allowing to re-use fixtures across function, class, module or whole test session scopes.

6.2 SUT Termination testing

As our **SUT** generates different threads that communicate with each other and must not terminate before getting acknowledges from their receivers that they finished their work and terminated correctly, so I put test scripts that test the termination of each node (input, OTSU, collector, contour-extractor, output) in `test_scripts/fixtures_test.py` file.

6.3 Fixtures features

6.3.1 Autouse fixtures

to test each node function, we need to initialize the sockets that will communicate with it as its senders or receivers, so we used **init_sockets** fixture to do so, and we made its **autouse** parameter to be true to be called by default with each test function.

```
@pytest.fixture(autouse=True)
def _init_sockets(...):
    ...
```

6.3.2 Fixtures can introspect the requesting test context

different nodes have different socket types (bind / connect), so we need to initialize tests' sockets opposite of their types in nodes. to make **init_sockets** generic, we specified PyTest **markers** to test functions so that the fixture can access the **marker** type through **request** object.

```
def init_sockets(...):
    socket_type = request.node.get_closest_marker("type").args[0]

@pytest.mark.type('connect')
def test_input_terminate(...):
    ...
```

6.3.3 Fixture scope: sharing fixtures

Fixtures are created when first requested by a test, and are destroyed based on their scope:

1. **function**: the default scope, the fixture is destroyed at the end of the test.
2. **class**: the fixture is destroyed during teardown of the last test in the class.
3. **module**: the fixture is destroyed during teardown of the last test in the module.
4. **package**: the fixture is destroyed during teardown of the last test in the package.
5. **session**: the fixture is destroyed at the end of the test session.

we used module scope with fixtures that defines global objects that need to be instantiated once for all tests in the module like in **context** fixture, but used function scope with **init_sockets** fixture as it needs to be invoked for each test. PyTest allows us to use dynamic scopes that change for the same fixture.

6.3.4 Fixture finalization and teardown code

teardown code is the code that is executed when the fixture is destroyed. to include teardown code we need to use **yield** statement instead of **return** and place teardown code after **yield** statement. we used this feature to close all sockets at the end of each test (i.e. when **init_sockets** is destroyed) to be reused again.

```
def init_sockets(...):
    ...
    yield sockets

    sockets.in_socket.close()
    sockets.out_socket.close()
```

6.3.5 Modularity

In addition to using fixtures in test functions, fixture functions can use other fixtures themselves. This contributes to a modular design of your fixtures as classes and allows re-use of framework-specific fixtures across many projects.

we used this feature to make two classes to implement the logic of **connect** and **bind** types of sockets call objects from these classes in **init_sockets** fixture.

```
class BindSockets:
    def __init__(self,...):
        self.init_in_socket = self.init_in_socket_bind()
        self.init_out_socket = self.init_out_socket_bind()

    def init_in_socket_bind(self):
        ...

    def init_out_socket_bind(self):
        ...

class ConnectSockets:
    def __init__(self,...):
        self.init_in_socket = self.init_in_socket_connect()
        self.init_out_socket = self.init_out_socket_connect()

    def init_in_socket_connect(self):
        ...

    def init_out_socket_connect(self):
        ...

@pytest.fixture(scope = 'function')
def init_sockets(...):
    if socket_type == 'connect':
        sockets = ConnectSockets(...)
    elif socket_type == 'bind':
        sockets = BindSockets(...)
```

7 Skip and Xfail

7.1 Introduction

Any software under test *(SUT)* can have a set of infeasible test cases that can arise from dependencies issue or unimplemented functionalities. **pytest** offers some robust features to handle such cases using *skip* and *xfail* markers. In this section, we discuss both features in a bit of details. The scripts can be found under `test_scripts/skip_xfail_test.py`, where the *OTSU node* of the software under test is covered using both *node* and *edge* **coverage**. The script contains 5 test cases :

1. **test_connection()** : node connection with outer sockets *[SKIPPED]*.
2. **test_output()** : node output validity *[PASSED]*.
3. **test_terminate()** : termination behaviour of node *[XPASSED]*.
4. **test_wrong_input()** : node behaviour with wrong inputs *[FAILED]*.
5. **test_wrong_port()** : node behaviour with wrong ports *[XFAILED]*.

7.2 PyTest Skip Marker

7.2.1 Description

Skip marker is usually used with tests that are expected to pass only under certain conditions. In other words, this feature is used by **pytest** to execute test cases, only if some conditions are met. This can be very useful for :

1. Skipping test cases of unimplemented functionalities, which is crucial for *test-driven development*.
2. Skipping test cases on platform migration. This is used when migrating the software into another platform, where some implementations are not expected to run properly.
3. Skipping test cases with unmet dependencies, where requirements for running code are unmet, due to incompatible versions or missing dependencies.

7.2.2 Usage

Skip marker is mainly used through *@pytest.mark.skip()* or *@pytest.mark.skipif()* python decorators. It can be used to skip a certain test case and even a complete class or module. This feature is covered in the scripts by using both markers on *test functions* or *text classes*.

@pytest.mark.skip() is used to skip a test case immediately without consideration :

```
@pytest.mark.skip(reason="unimplemented functionality")
def test_case():
    ...
```

Meanwhile, *@pytest.mark.skipif()* is used to skip a test case when a certain condition is met :

```
import sys
@pytest.mark.skipif(sys.version_info < (3, 7),
reason="requires python3.7 or higher")
def test_case():
    ...
```

7.3 PyTest XFail Marker

7.3.1 Description

XFail marker is usually used to indicate that a test is expected to fail. A common example is a test for a feature not yet implemented, or a bug not yet fixed. However, if the test passes, it will be reported as *xpass*. Moreover, if the test fails, it will not be reported as a failure. The tester can mark a test as *strict non-runnable xfail*, which will skip running the test. This can be useful to clean test report from any tests that are expected to fail, in order to only focus on meaningful failures.

7.3.2 Usage

XFail marker is mainly used through *@pytest.mark.xfail()* python decorator. However, a set of parameters can be specified for extra functionalities :

1. **condition** : defines a certain condition upon which the test is expected to fail.
2. **reason** : defines the reason why test is expected to fail.
3. **raises** : defines a set of exceptions that will be raised if test fails.
4. **run** : a boolean that defines whether to run the test or totally ignore it.
5. **strict** : a boolean that defines whether the test is a strict *xfail*.

```
import sys
@pytest.mark.xfail(sys.platform == "win32",
reason="bug in a 3rd party library", raises=RuntimeError,
run=False, strict=True)
def test_case():
    ...
```

8 Plugins

9 Hooks

9.1 Description

pytest test suite reports all test results, whether it is a pass or a failure. There can be a huge number of test cases, which makes the failures very difficult to track through the terminal. However, **pytest** calls *hook* functions from registered *plugins* for any given hook specification. Hook functions can be used to organize tests execution, but, more importantly, it can be used to dump all test failures into `failures.txt` file, which can be much easier to search through. **pytest** can automatically pick up the configured hooks and execute the desired tests, in order to generate the failures file.

9.2 Usage

Hook functions are defined in **pytest** configuration file, which is named `conftest.py`. This file can be detected directly by **pytest** for both *plugins* and *hooks*. To define a hook function, *@pytest.hookimpl()* python decorator is used. Hooks are covered in our test scripts under `test_scripts/conftest.py`.

To define a hook function, start test session and link the failures text file :

```
@pytest.hookimpl()
def pytest_sessionstart(session: Session):
    ...
```

To define a hook wrapper function that run tests and make the report :

```
@pytest.hookimpl(hookwrapper=True)
def pytest_runtest_makereport(item: Item, call: CallInfo):
    ...
```

10 Mock Property

10.1 Description

In *Python*, an *API* call can $1,000$ seconds to run. Consequently, when testing an interface (or wrapper) that calls an *API* function, it would consume a lot of time to perform a simple test. One way to overcome this is to fix a correct return of the *API* call for each test, in order to avoid long call times. This is the main use of the **mocks** in **pytest**, which allow the tester to set a fixed correct return of the *API* calls. This enables fast and effective run of the tests, which is important for continuous integration and development *(CI/CD)*. We are motivated to include **mocks** in our experiments, as our *SUT* contains a decent set of *API* calls of external libraries. We include some tests that use **mocks** in our scripts, in order to speed up execution. However, not all tests use **mocks**, as we need to test *dynamic behaviour* and for the purpose of *integration testing*.

10.2 Usage

We can define **mocks** for **pytest** by using *@mock.patch()* python decorator. This takes the target *API* function as a parameter and define a single or multiple fixed outputs of it later in the code. We include the use of **mocks** in our test scripts under `test_scripts/mock_test.py`.

The exact syntax of **mocks** definition in **pytest** is shown as follows :

```
@mock.patch("api_name.function_name" ,autospec = True)
    def test_function(self ,mock_variable):
    ...
    mock_variable.side_effect = [..]
    ...
```

We notice that a **mock** variable is passed to the test function, which is used to set the *side effect* of the **mock**. The *side effect* of the **mock** is a list of fixed returns of the *API* function call.

11 Work Load Distribution

Name	Work
Ahmed Mohamed Zakaria	Plugins
Remonda Talaat Eskarous	Hooks and Mock Property
Mohamed Ahmed Mohamed Ahmed	Fixtures
Mohamed Shawky Zaky	Skip and Xfail
Mohamed Ramzy Helmy	Assertions explanations, Exceptions & Warnings

References

[1] Pytest documentation: docs.pytest.org.

[2] Pytest repository: pytest-dev/pytest.