KPM Power

Internal Training Document

Python Testing

Frameworks and Usage

Ashar Latif

ashar@kpmpower.com

September 1, 2021

Contents

This document gives a brief overview of the preferred testing framework we use for Python, as well as some general implementations.

1 Setup

1.1 Dependencies

```
pip install pytest
```

1.2 Config

Global fixtures file is called `conftest.py` and should be placed in the root of the project. All global fixtures go in this file. See Fixtures section for more info.

Pytest detects test automatically if in the following formats:

```
test_*.py
*_test.py
```

This is the recommended folder format (just for organization, `pytest` will detect tests based solely on file name):

```
<project name>/
|- tests/
   |- integration/
      |- # integration tests go here
   |- unit/
      |- # unit tests go here
```

Project-level configurations can be entered into a `pytest.ini` file in the root of the project. A full list of options can be found here and a sample `pytest.ini` can be found at the bottom of this file.

2 Running Tests

To run all defined tests, just invoke the command:

```
pytest
```

If we want to run a specific test file (lets call it `sample_test.py`), run the following command:

```
pytest sample_test.py
```

To run tests whose names contain a specific substring, invoke the following:

```
pytest -k <substring>
```

To run a specific test from a specific file, invoke the following:

```
pytest <test_file.py>::<test_function_name>
```

If we want to only run tests defined with specific **marks**, (for example `mark1`), run the following command:

```
pytest -m mark1
```

See Marks section for more info.

2.1 Verbose Mode

It is recommended to always run `pytest` in verbose mode, which can be invoked with the `-v` flag. Alternatively, you can make this option always-on by adding `export PYTEST_ADDOPTS="-v"` to your environment variables. Alternatively, you can add the following lines to `pytest.ini` (in the root of the project):

```
[pytest]
addopts = -v
```

3 Concurrent Tests

We can run tests in parallel using `pytest`. First we need to install the dependencies by invoking:

```
pip install pytest-xdist
```

Then, we can run multiple tests concurrently with the following command:

```
pytest -n <number_of_tests>
```

The `-n` flag is equivalent to the `--numprocesses` flag. We can also run as many parallel tests as we have CPU cores by running:

```
pytest -n auto
```

This setting can be added into you `pytest.ini`

4 Test Basics

4.1 Simple Passing Test

```
def hello_world():
    return 'hello world'

def test_function():
    assert hello_world() == 'hello world'
```

4.2 Test for Exceptions

```python
import pytest

def hello_error():
    raise NotImplementedError

def test_function():
    with pytest.raises(NotImplementedError):
        hello_error()

def test_context():
    with pytest.raises(NotImplementedError) as e:
        hello_error()
    assert e.xyz == abc
    # note that the exception context object is referenced *outside* the `with`
        block
```

5 Fixtures

Fixtures are indicated by the `@pytest.fixture` decorator. Best practice is to put these in `conftest.py` so that any test file can use it. A quick example:

```python
# conftest.py

import pytest

@pytest.fixture
def supply_AA_BB_CC():
    aa = 25
    bb = 35
    cc = 45
    return [aa,bb,cc]
```

```python
# basic_test.py

import pytest

def test_comparewithAA(supply_AA_BB_CC):
    zz = 35
    assert supply_AA_BB_CC[0]==zz,"aa and zz comparison failed"

def test_comparewithBB(supply_AA_BB_CC):
    zz = 35
    assert supply_AA_BB_CC[1]==zz,"bb and zz comparison failed"

def test_comparewithCC(supply_AA_BB_CC):
    zz = 35
    assert supply_AA_BB_CC[2]==zz,"cc and zz comparison failed"
```

6 Marks

Marks are indicated with a decorator in the format:

```
@pytest.mark.<mark_name>
```

For example, given the following test file:

```
import pytest

@pytest.mark.set1
def test_file1_method1():
    x=5
    y=6
    assert x+1 == y,"test failed"
    assert x == y,"test failed because x=" + str(x) + " y=" + str(y)

@pytest.mark.set2
def test_file1_method2():
    x=5
    y=6
    assert x+1 == y,"test failed"
```

Running `py.test -m set1` will run only `test_file_method1()`. It doesn't matter if marks in in separate files, all matching marks will still be run.

6.1 Special Marks

The two most important special marks are `xfail` and `skip`. Marking a test with `skip` will make `pytest` skip that test. `xfail` is much more interesting. We use this for test that are expected to fail. For example, running the following test:

```
import pytest
@pytest.mark.skip
def test_add_1():
    assert 100+200 == 400,"failed"

@pytest.mark.xfail
def test_add_2():
    assert 15+13 == 28,"failed"

@pytest.mark.xfail
def test_add_3():
    assert 15+13 == 100,"failed"

def test_add_4():
    assert 3+2 == 6,"failed"
```

Gives the following output:

```
test_addition.py::test_add_1 SKIPPED
test_addition.py::test_add_2 XPASS
test_addition.py::test_add_3 xfail
test_addition.py::test_add_4 FAILED

============================================== FAILURES
    ==========================================
```

```
_____________________________________________ test_add_4
    ______________________________________________
    def test_add_4():
>       assert 3+2 == 6,"failed"
E    AssertionError: failed
E    assert (3 + 2) == 6
test_addition.py:24: AssertionError

================ 1 failed, 1 skipped, 1 xfailed, 1 xpassed in 0.07 seconds
    =================
```

7 Parameterized Tests

`pytest` allows us to use many arguments at once without rewriting functions. To do this you have to use the `@pytest.mark.parametrize` decorator. An example:

```
import pytest

@pytest.mark.parametrize("input1, input2, output",[(5,5,10),(3,5,12)])
def test_add(input1, input2, output):
    assert input1+input2 == output,"failed"
```

8 Mocking

There are a few ways of implementing mocks with `pytest`. This document will cover 2 ways: `pytest-mock` and `monkeypatch`. They have non-overlapping domains so there are certain situations where one will be preferred over the other. In general, either is acceptable to use. Some examples of comparative usage are in this article

8.1 pytest-mock

This requires `pytest-mock` to be installed, which can be done with the following command:

```
pip install pytest-mock
```

Variables

Variable mocking is mainly used for mocking globals (outside of function scope). Say you have the following file that contains a Lambda handler:

```
# lambda_handler.py

is_cold_start = True

def handler():
    if is_cold_start:
        # do stuff
        is_cold_start = False
    # do more stuff
```

```
    return True  # arbitrary return value for illustration
```

You want to be able to test `handler()` when `is_cold_start = False`; to do so, you must mock `is_cold_start`.

```
# handler_test.py

import pytest
import lambda_handler

def test_handler(mocker):
    mocker.patch.object(lambda_handler, 'is_cold_start', False)
    assert handler()
```

The signature is as follows:

```
mocker.patch.object(
    module,      # this is NOT a string
    'variable',  # this IS a string
    value        # this is whatever
)
```

The module name follows the import name. For example, if `lambda_handler.py` were in a folder `src/`, the module would then be `src.lambda_handler`.

Functions

For the following file:

```
# hello_world.py

import os

def say_passphrase():
    passphrase = os.environ.get('PASSPHRASE')
    return passphrase or 'I need somebody (Help!)'
```

We can mock the `os.environ.get()` call by using the fully qualified method name like so:

```
# hello_world_test.py

import pytest
from hello_world import say_passphrase

def test_passphrase(mocker):
    mocker.patch('hello_world.os.environ.get', return_value='hello world')
    assert say_passphrase() == 'hello world'
```

8.2 monkeypatch

This is native to `pytest` so has no further dependencies. More info on `monkeypatch` can be found here, and a general tutorial on its functionality is here

9 Test Coverage

We will be using the `coverage` library to check for test coverage. To install, invoke the following:

```
pip install coverage
```

Testing for coverage is extremely simple with this tool and does not require any modifications to our standard testing invocations. Instead, you simply precede your standard command with `coverage run -m` as show below:

```
coverage run -m pytest sample_test.py
```

To view the coverage report, simply run:

```
coverage report -m
```

This will return something that looks like the following:

```
$ coverage report -m
Name                      Stmts   Miss  Cover   Missing
-------------------------------------------------------
my_program.py                20      4    80%   33-35, 39
my_other_module.py           56      6    89%   17-23
-------------------------------------------------------
TOTAL                        76     10    87%
```

10 Sample pytest.ini

This `pytest.ini` will always have verbose output and will run as many parallel tests as there are CPU cores on the host device. The full list of command line flags can be found here.

```
[pytest]
addopts = -v --numprocesses auto
```