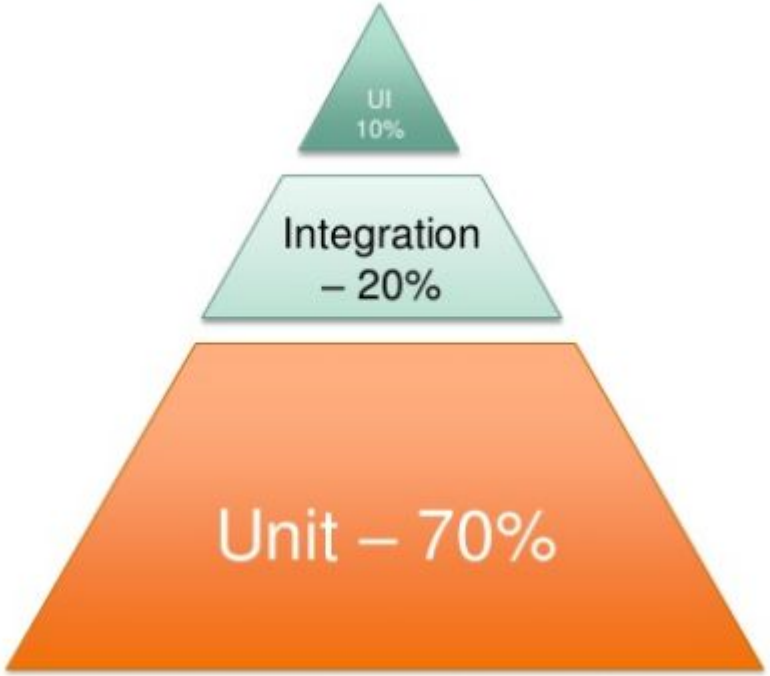
Python Testing using Mock

Marcelo Miranda

Tests

Testing Types



"TYPES" OF TESTS

- A good way to think about testing is to think about three different kinds of input:

"Good" Input

-What you would expect
-What the simplest implementation would be able to do

Invalid Input

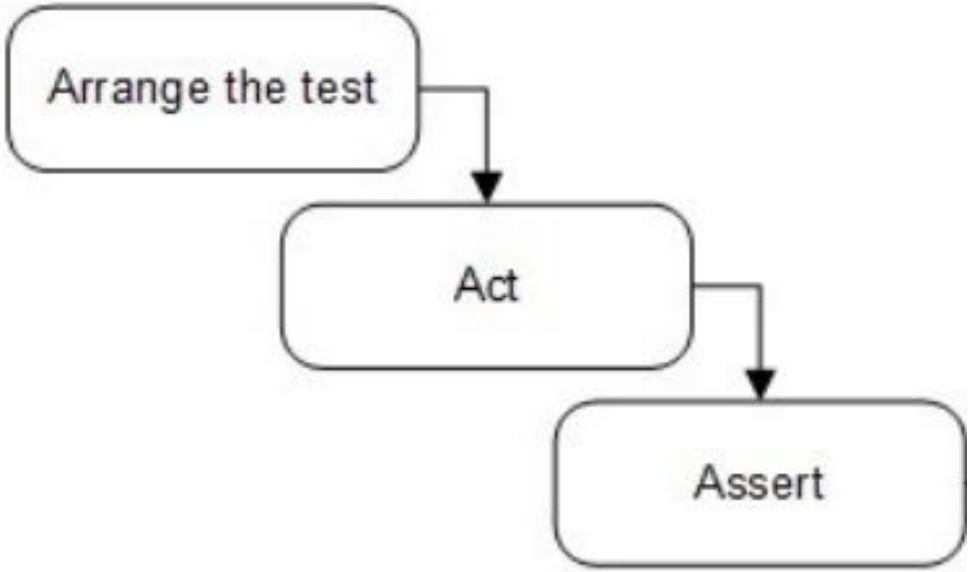
-Input that can't be used within the specs of the function
-Input that shouldn't get pass the initial checks in the code

Edge Cases

-Tricky version of "Good" input that may deviate from normal in how it is handled, or could potentially have unique behaviour

Tests Recipe

- **Arrange**: Initializes objects, creates mocks with arguments that are passed to the method under test and adds expectations.
- **Act**: Invokes the method or property under test with the arranged parameters.
- **Assert**: Verifies that the action of the method under test behaves as expected.



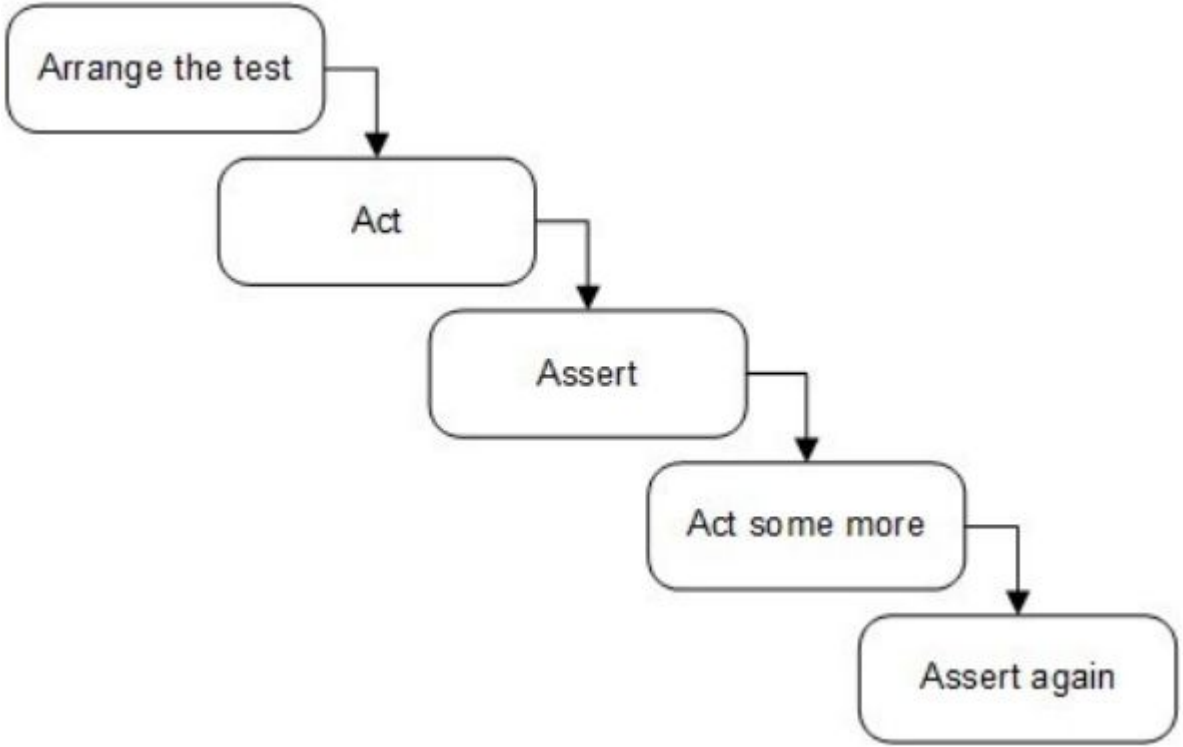
Setup dos dados

Setup expectativas

Exercitar o código

Realizar Verificações

AVOID



Tests Structure

github medium

```
project-root
├──src/
   └──package_name/
      └──__init__.py
├──tests/
   └──package_name/
├──.coveragerc
├──.gitignore
├──LICENSE
├──README.md
├──setup.cfg
└──setup.py
```

setup.py

```
setup_requires=(
    'pytest-runner',
),
tests_require=(
    'pytest-cov',
)
```

setup.cfg

```
[aliases]
test = pytest


[tool:pytest]
addopts = --cov --cov-report html --cov-report term-missing
```

Tests Code

```
import unittest


class SimpleTest(unittest.TestCase):
    def test1(self):
        self.assertEqual(4+5,9)
    def test2(self):
        self.assertNotEqual(5*2,10)
    def test3(self):
        self.assertTrue(4+5==9,"The result is False")
    def test4(self):
        self.assertTrue(4+5==10,"assertion fails")
    def test5(self):
        self.assertIn(3,[1,2,3])
    def test6(self):
        self.assertNotIn(3, range(5))


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

TestCase Class

Object of this class represents the smallest testable unit. It holds the test routines and provides hooks for preparing each routine and for cleaning up thereafter.

The following methods are defined in the TestCase class:

setUp()	Method called to prepare the test fixture. This is called immediately before calling the test method
tearDown()	Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception,
setUpClass()	A class method called before tests in an individual class run.
tearDownClass()	A class method called after tests in an individual class have run.
run(*result=None*)	Run the test, collecting the result into the test result object passed as *result*.
skipTest(*reason*)	Calling this during a test method or setUp() skips the current test.
debug()	Run the test without collecting the result.
shortDescription()	Returns a one-line description of the test.

Asserts

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Assert for Collections

This set of assert functions are meant to be used with collection data types in Python, such as List, Tuple, Dictionary and Set.

assertListEqual (*list1*, *list2*, *msg=None*)	Tests that two lists are equal. If not, an error message is constructed that shows only the differences between the two.
assertTupleEqual (tuple1, tuple2, msg=None)	Tests that two tuples are equal. If not, an error message is constructed that shows only the differences between the two.
assertSetEqual (set1, set2, msg=None)	Tests that two sets are equal. If not, an error message is constructed that lists the differences between the sets.
assertDictEqual (expected, actual, msg=None)¶	Test that two dictionaries are equal. If not, an error message is constructed that shows the differences in the dictionaries.

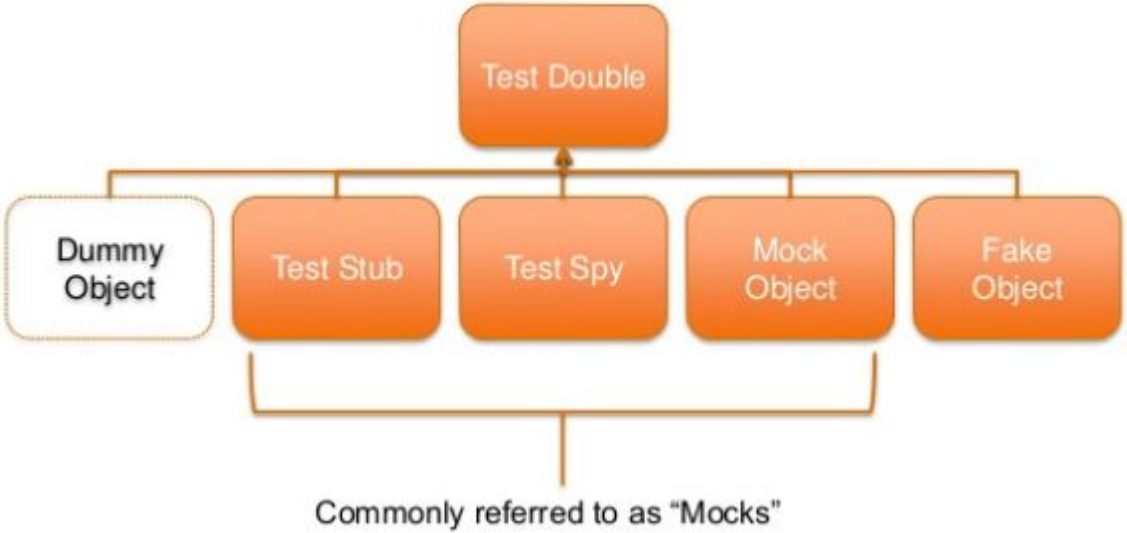
Skip – decorators

The following decorators implement test skipping and expected failures:

unittest.**skip**(*reason*)	Unconditionally skip the decorated test. *reason* should describe why the test is being skipped.
unittest.skipIf(condition, reason)	Skip the decorated test if condition is true.
unittest.skipUnless(condition, reason)	Skip the decorated test unless condition is true.
unittest.expectedFailure()	Mark the test as an expected failure. If the test fails when run, the test is not counted as a failure.

Mocks

What Are Mocks?



What Are Mocks?

- Stubs
  - Provide a canned response to method calls
- Spy
  - Real objects that behave like normal except when a specific condition is met
- Mocks
  - Verifies behavior (calls) to a method

Problems Mocks Solve

- Tests methods that have no return value

```
#!/usr/bin/env python

def foo(x):
    if x > 10:
        bar(x)
    else:
        something_else(x)
```

How do we know that bar(x) has been called?

Problems Mocks Solve

- Tests error handling

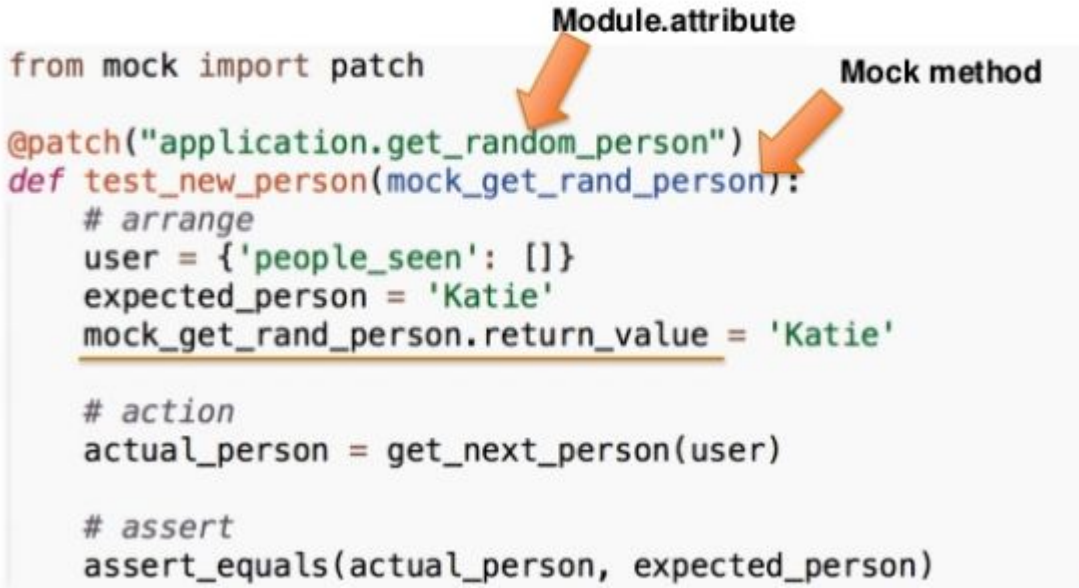
```
#!/usr/bin/env python

def foo(filename):
    try:
        return parse_large_file(filename)
    except MemoryError:
        return ""
```

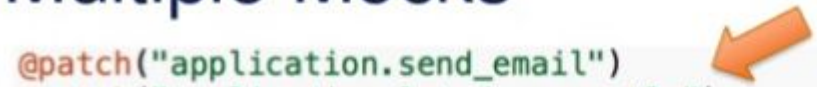
How do we generate this exception???

Mocks – patching

Patching


Module.attribute
Mock method
from mock import patch
@patch("application.get_random_person")
def test_new_person(mock_get_rand_person):
# arrange
user = {'people_seen': []}
expected_person = 'Katie'
mock_get_rand_person.return_value = 'Katie'
# action
actual_person = get_next_person(user)
# assert
assert_equals(actual_person, expected_person)

Multiple Mocks



```
@patch("application.send_email")
@patch("application.let_down_gently")
@patch("application.give_it_time")
def test_person2_dislikes_person1(mock_give_it_time,
                                  mock_let_down,
                                  mock_send_email):
    # arrange
    person1 = 'Bill'
    person2 = {'likes': ['Sam'], 'dislikes': ['Bill'] }

    # action
    evaluate(person1, person2)

    # assert
    mock_let_down.assert_called_once_with(person1)
    assert_equals(mock_give_it_time.call_count, 0)
    assert_equals(mock_send_email.call_count, 0)
```

Patch Multiple

```
@patch.multiple("application",
    send_email=DEFAULT,
    let_down_gently=DEFAULT,
    give_it_time=DEFAULT)
def test_person2_dislikes_person1_multi(send_email,
                                        let_down_gently,
                                        give_it_time):
    # arrange
    person1 = 'Bill'
    person2 = {'likes': ['Sam'], 'dislikes': ['Bill'] }

    # action
    evaluate(person1, person2)

    # assert
    let_down_gently.assert_called_once_with(person1)
    assert_equals(give_it_time.call_count, 0)
    assert_equals(send_email.call_count, 0)
```

Mocks – side_effect

Use *side_effect*

```
@patch.object(Application, "get_random_person")
def test_experienced_user(mock_get_rand_person):
    # arrange
    app = Application()
    user = {'people_seen': ['Sarah', 'Mary']}
    expected_person = 'Katie'
    mock_get_rand_person.side_effect = ['Mary', 'Sarah', 'Katie']

    # action
    actual_person = app.get_next_person(user)

    # assert
    assert_equals(actual_person, expected_person)
```

Test runners

Choosing a Test Runner

01 **unittest**

- Contains both a testing framework and a test runner.
- Requires
  - Put your tests into classes as methods.
  - Use a serie of special assertion methods instead of the built-in assert statement..

```python
import unittest


class TestSum(unittest.TestCase):

    def test_sum(self):
        self.assertEqual(sum([1, 2, 3]), 6, "Should be 6")

    def test_sum_tuple(self):
        self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")

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

```
$ python test_sum_unittest.py
.F
======================================================================
FAIL: test_sum_tuple (__main__.TestSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum_unittest.py", line 9, in test_sum_tuple
    self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")
AssertionError: Should be 6

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Choosing a Test Runner

02

nose or nose2

- Nose is compatible with any tests written using the unittest framework.

```
$ pip install nose2
$ python -m nose2
.F
======================================================================
FAIL: test_sum_tuple (__main__.TestSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum_unittest.py", line 9, in test_sum_tuple
    self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")
AssertionError: Should be 6

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Choosing a Test Runner

03

pytest

- Support the execution of unittest test cases.
- Advantage of Pytest comes by writing pytest test cases.
- Features:
  - Support for the built-in assert statement instead of using special self.assert*().
  - Support for filtering for test cases.
  - Ability to rerun from the last failing test.
  - An ecosystem of hundreds of plugins to extend the functionality

```
def test_sum():
    assert sum([1, 2, 3]) == 6, "Should be 6"

def test_sum_tuple():
    assert sum((1, 2, 2)) == 6, "Should be 6"
```

Pytest

pytest

pytest supports execution of unittest test cases. The real advantage of pytest comes by writing pytest test cases. pytest test cases are a series of functions in a Python file starting with the name test_.

pytest has some other great features:

- Support for the built-in assert statement instead of using special self.assert*() methods
- Support for filtering for test cases
- Ability to rerun from the last failing test
- An ecosystem of hundreds of plugins to extend the functionality

Writing the TestSum test case example for pytest would look like this:

Python

```python
def test_sum():
    assert sum([1, 2, 3]) == 6, "Should be 6"

def test_sum_tuple():
    assert sum((1, 2, 2)) == 6, "Should be 6"
```

You have dropped the TestCase, any use of classes, and the command-line entry point.

Fixtures

sample

Examples

example1
example2

Challenge

challenge

ciandt.com

THANK YOU

Driven by **Impact**