Unit Testing All Versions

Examples

Test Setup and Teardown within a unittest.TestCase

Sometimes we want to prepare a context for each test to be run under. The `setUp` method is run prior to each test in the class. `tearDown` is run at the end of every test. These methods are optional. Remember that TestCases are often used in cooperative multiple inheritance so you should be careful to always call `super` in these methods so that base class's `setUp` and `tearDown` methods also get called. The base implementation of `TestCase` provides empty `setUp` and `tearDown` methods so that they can be called without raising exceptions:

```
import unittest


class SomeTest(unittest.TestCase):
    def setUp(self):
        super(SomeTest, self).setUp()
        self.mock_data = [1,2,3,4,5]

    def test(self):
        self.assertEqual(len(self.mock_data), 5)

    def tearDown(self):
        super(SomeTest, self).tearDown()
        self.mock_data = []


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

Note that in python2.7+, there is also the `addCleanup` method that registers functions to be called after the test is run. In contrast to `tearDown` which only gets called if `setUp` succeeds, functions registered via `addCleanup` will be called even in the event of an unhandled exception in `setUp` . As a concrete example, this method can frequently be seen removing various mocks that were registered while the test was running:

```
import unittest
import some_module


class SomeOtherTest(unittest.TestCase):
    def setUp(self):
        super(SomeOtherTest, self).setUp()

        # Replace `some_module.method` with a `mock.Mock`
        my_patch = mock.patch.object(some_module, 'method')
        my_patch.start()

        # When the test finishes running, put the original method back.
        self.addCleanup(my_patch.stop)
```

Another benefit of registering cleanups this way is that it allows the programmer to put the cleanup code next to the setup code and it protects you in the event that a subclasser forgets to call `super` in `tearDown` .

Asserting on Exceptions

You can test that a function throws an exception with the built-in unittest through two different methods.

Using a context manager

```
def division_function(dividend, divisor):
    return dividend / divisor


class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError):
            x = division_function(1, 0)
```

This will run the code inside of the context manager and, if it succeeds, it will fail the test because the exception was not raised. If the code raises an exception of the correct type, the test will continue.

You can also get the content of the raised exception if you want to execute additional assertions against it.

```
class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError) as ex:
```

```
            x = division_function(1, 0)

        self.assertEqual(ex.message, 'integer division or modulo by zero')
```

By providing a callable function

```
def division_function(dividend, divisor):
    """
    Dividing two numbers.

    :type dividend: int
    :type divisor: int

    :raises: ZeroDivisionError if divisor is zero (0).
    :rtype: int
    """
    return dividend / divisor


class MyTestCase(unittest.TestCase):
    def test_passing_function(self):
        self.assertRaises(ZeroDivisionError, division_function, 1, 0)
```

The exception to check for must be the first parameter, and a callable function must be passed as the second parameter. Any other parameters specified will be passed directly to the function that is being called, allowing you to specify the parameters that trigger the exception.

Choosing Assertions Within Unittests

While Python has an assert statement , the Python unit testing framework has better assertions specialized for tests: they are more informative on failures, and do not depend on the execution's debug mode.

Perhaps the simplest assertion is `assertTrue` , which can be used like this:

```
import unittest

class SimplisticTest(unittest.TestCase):
    def test_basic(self):
        self.assertTrue(1 + 1 == 2)
```

This will run fine, but replacing the line above with

```
        self.assertTrue(1 + 1 == 3)
```

will fail.

The `assertTrue` assertion is quite likely the most general assertion, as anything tested can be cast as some boolean condition, but often there are better alternatives. When testing for equality, as above, it is better to write

```
        self.assertEqual(1 + 1, 3)
```

When the former fails, the message is

```
======================================================================

FAIL: test (__main__.TruthTest)

----------------------------------------------------------------------

Traceback (most recent call last):

  File "stuff.py", line 6, in test

    self.assertTrue(1 + 1 == 3)

AssertionError: False is not true
```

but when the latter fails, the message is

```
======================================================================

FAIL: test (__main__.TruthTest)

----------------------------------------------------------------------

Traceback (most recent call last):

  File "stuff.py", line 6, in test
```

```
    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3
```

which is more informative (it actually evaluated the result of the left hand side).

You can find the list of assertions in the standard documentation . In general, it is a good idea to choose the assertion that is the most specifically fitting the condition. Thus, as shown above, for asserting that `1 + 1 == 2` it is better to use `assertEqual` than `assertTrue` . Similarly, for asserting that `a is None` , it is better to use `assertIsNone` than `assertEqual` .

Note also that the assertions have negative forms. Thus `assertEqual` has its negative counterpart `assertNotEqual` , and `assertIsNone` has its negative counterpart `assertIsNotNone` . Once again, using the negative counterparts when appropriate, will lead to clearer error messages.

Testing Exceptions

Programs throw errors when for instance wrong input is given. Because of this, one needs to make sure that an error is thrown when actual wrong input is given. Because of that we need to check for an exact exception, for this example we will use the following exception:

```
class WrongInputException(Exception):
    pass
```

This exception is raised when wrong input is given, in the following context where we always expect a number as text input.

```
def convert2number(random_input):
    try:
        my_input = int(random_input)
    except ValueError:
        raise WrongInputException("Expected an integer!")
    return my_input
```

To check whether an exception has been raised, we use `assertRaises` to check for that exception. `assertRaises` can be used in two ways:

1. Using the regular function call. The first argument takes the exception type, second a callable (usually a function) and the rest of arguments are passed to this callable.
2. Using a `with` clause, giving only the exception type to the function. This has as advantage that more code can be executed, but should be used with care since multiple functions can use the same exception which can be problematic. An example: with self.assertRaises(WrongInputException): convert2number("not a number")

This first has been implemented in the following test case:

```
import unittest

class ExceptionTestCase(unittest.TestCase):

    def test_wrong_input_string(self):
        self.assertRaises(WrongInputException, convert2number, "not a number")

    def test_correct_input(self):
        try:
            result = convert2number("56")
            self.assertIsInstance(result, int)
        except WrongInputException:
            self.fail()
```

There also may be a need to check for an exception which should not have been thrown. However, a test will automatically fail when an exception is thrown and thus may not be necessary at all. Just to show the options, the second test method shows a case on how one can check for an exception not to be thrown. Basically, this is catching the exception and then failing the test using the `fail` method.

Mocking functions with unittest.mock.create_autospec

One way to mock a function is to use the `create_autospec` function, which will mock out an object according to its specs. With functions, we can use this to ensure that they are called appropriately.

With a function `multiply` in `custom_math.py` :

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

And a function `multiples_of` in `process_math.py` :

```
from custom_math import multiply
```

```
def multiples_of(integer, *args, num_multiples=0, **kwargs):
    """
    :rtype: list
    """
    multiples = []

    for x in range(1, num_multiples + 1):
        """
        Passing in args and kwargs here will only raise TypeError if values were
        passed to multiples_of function, otherwise they are ignored. This way we can
        test that multiples_of is used correctly. This is here for an illustration
        of how create_autospec works. Not recommended for production code.
        """
        multiple = multiply(integer,x, *args, **kwargs)
        multiples.append(multiple)

    return multiples
```

We can test `multiples_of` alone by mocking out `multiply` . The below example uses the Python standard library unittest, but this can be used with other testing frameworks as well, like pytest or nose:

```
from unittest.mock import create_autospec
import unittest

# we import the entire module so we can mock out multiply
import custom_math
custom_math.multiply = create_autospec(custom_math.multiply)
from process_math import multiples_of


class TestCustomMath(unittest.TestCase):
    def test_multiples_of(self):
        multiples = multiples_of(3, num_multiples=1)
        custom_math.multiply.assert_called_with(3, 1)

    def test_multiples_of_with_bad_inputs(self):
        with self.assertRaises(TypeError) as e:
            multiples_of(1, "extra arg",  num_multiples=1) # this should raise a TypeError
```

Unit tests with pytest

installing pytest:

```
pip install pytest
```

getting the tests ready:

```
mkdir tests
touch tests/test_docker.py
```

Functions to test in `docker_something/helpers.py` :

```
from subprocess import Popen, PIPE
# this Popen is monkeypatched with the fixture `all_popens`

def copy_file_to_docker(src, dest):
    try:
        result = Popen(['docker','cp', src, 'something_cont:{}'.format(dest)], stdout=PIPE, stde
        err = result.stderr.read()
        if err:
            raise Exception(err)
    except Exception as e:
        print(e)
    return result

def docker_exec_something(something_file_string):
    fl = Popen(["docker", "exec", "-i", "something_cont", "something"], stdin=PIPE, stdout=PIPE,
    fl.stdin.write(something_file_string)
    fl.stdin.close()
    err = fl.stderr.read()
    fl.stderr.close()
    if err:
        print(err)
        exit()
    result = fl.stdout.read()
    print(result)
```

The test imports `test_docker.py` :

```
import os
```

```
import os
from tempfile import NamedTemporaryFile
import pytest
from subprocess import Popen, PIPE

from docker_something import helpers
copy_file_to_docker = helpers.copy_file_to_docker
docker_exec_something = helpers.docker_exec_something
```

mocking a file like object in `test_docker.py` :

```
class MockBytes():
    '''Used to collect bytes
    '''
    all_read = []
    all_write = []
    all_close = []

    def read(self, *args, **kwargs):
        # print('read', args, kwargs, dir(self))
        self.all_read.append((self, args, kwargs))

    def write(self, *args, **kwargs):
        # print('wrote', args, kwargs)
        self.all_write.append((self, args, kwargs))

    def close(self, *args, **kwargs):
        # print('closed', self, args, kwargs)
        self.all_close.append((self, args, kwargs))

    def get_all_mock_bytes(self):
        return self.all_read, self.all_write, self.all_close
```

Monkey patching with pytest in `test_docker.py` :

```
@pytest.fixture
def all_popens(monkeypatch):
    '''This fixture overrides / mocks the builtin Popen
        and replaces stdin, stdout, stderr with a MockBytes object

        note: monkeypatch is magically imported
    '''
    all_popens = []

    class MockPopen(object):
        def __init__(self, args, stdout=None, stdin=None, stderr=None):
            all_popens.append(self)
            self.args = args
            self.byte_collection = MockBytes()
            self.stdin = self.byte_collection
            self.stdout = self.byte_collection
            self.stderr = self.byte_collection
            pass
    monkeypatch.setattr(helpers, 'Popen', MockPopen)

    return all_popens
```

Example tests, must start with the prefix `test_` in the `test_docker.py` file:

```
def test_docker_install():
    p = Popen(['which', 'docker'], stdout=PIPE, stderr=PIPE)
    result = p.stdout.read()
    assert 'bin/docker' in result

def test_copy_file_to_docker(all_popens):
    result = copy_file_to_docker('asdf', 'asdf')
    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert mock_read
    assert result.args == ['docker', 'cp', 'asdf', 'something_cont:asdf']


def test_docker_exec_something(all_popens):

    docker_exec_something(something_file_string)

    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert len(mock_read) == 3
    something_template_stdin = mock_write[0][1][0]
    these = [os.environ['USER'], os.environ['password_prod'], 'table_name_here', 'test_vdm', 'co
    assert all([x in something_template_stdin for x in these])
```

running the tests one at a time:

```
py.test -k test_docker_install tests
py.test -k test_copy_file_to_docker tests
py.test -k test_docker_exec_something tests
```

```
py.test -k test_ tests
```

running all the tests in the `tests` folder:

```
py.test -k test_ tests
```

Syntax

Parameters

Remarks

There are several unit testing tools for Python. This documentation topic describes the basic `unittest` module. Other testing tools include `py.test` and `nosetests`. This python documentation about testing compares several of these tools without going into depth.