aiounittest Documentation

Release 1.4.0

Krzysztof Warunek

Jun 13, 2022

Contents

1 What? Why? Next? **3**
- 1.1 Why Not? . . . 3
- 1.2 What? . . . 3
- 1.3 Why? . . . 3
- 1.4 Next? . . . 4

2 Usage **5**
- 2.1 AsyncTestCase . . . 5
- 2.2 AsyncMockIterator . . . 6
- 2.3 async_test . . . 7
- 2.4 futurized . . . 8

3 License **11**

Index **13**

In the Python 3.8 (release note) and newer consider to use the unittest.IsolatedAsyncioTestCase. Builtin `unittest` module is now asyncio-featured.

CHAPTER 1

What? Why? Next?

1.1 Why Not?

In the Python 3.8 (release note) and newer consider to use the unittest.IsolatedAsyncioTestCase. Builtin `unittest` module is now asyncio-featured.

1.2 What?

This module is a set of helpers to write simple and clean test for asyncio-based code.

1.3 Why?

Actually this is not nothing new, it just wraps current test approach in the handy utils. There are couple libraries that try to solve this problem. This one:

- integrates nicely with standard `unittest` library,
- is as simple as possible, without a bunch of stuff that is straithforward with unittest (eg re-inveting `assertRaises` with *assertAsyncRaises*),
- supports both Python 3.5+ syntax and Python 3.4,
- it's well-documented (I think)

Among the others similar modules the best known is an extension pytest-asyncio. It provides couple extra features, but it cannot be used with *unittest.TestCase* (it does not support fixture injection).

Further reading:

- https://stackoverflow.com/questions/23033939/how-to-test-python-3-4-asyncio-code
- http://jacobbridges.github.io/post/unit-testing-with-asyncio/

1.4 Next?

- introduce `AsyncMock`, ... Probably not, the `unitest.mock.Mock` with `futurized` is pretty simple.
- it would be great if *unittest* could support async test > *python-ideas*

CHAPTER 2

Usage

To enable support for async tests just use `aiounittest.AsyncTestCase` instead of `unittest.TestCase` (or decorate async test coroutines with `async_test`). The `futurized` will help you to mock coroutines.

2.1 AsyncTestCase

Extends `unittest.TestCase` to support asynchronous tests. Currently the most common solution is to explicitly run `asyncio.run_until_complete` with test case. Aiounittest `AsyncTestCase` wraps it, to keep the test as clean and simple as possible.

class `aiounittest.`**`AsyncTestCase`**(*methodName='runTest'*)

AsyncTestCase allows to test asynchoronus function.

The usage is the same as `unittest.TestCase`. It works with other test frameworks and runners (eg. *pytest*, *nose*) as well.

AsyncTestCase can run:

- test of synchronous code (`unittest.TestCase`)
- test of asynchronous code, supports syntax with `async`/`await` (Python 3.5+) and `asyncio.coroutine`/`yield from` (Python 3.4)

Code to test:

```
import asyncio

async def async_add(x, y, delay=0.1):
    await asyncio.sleep(delay)
    return x + y

async def async_one():
    await async_nested_exc()

async def async_nested_exc():
```

(continues on next page)

(continued from previous page)

```
    await asyncio.sleep(0.1)
    raise Exception('Test')
```

Tests:

```
import aiounittest

class MyTest(aiounittest.AsyncTestCase):

    async def test_await_async_add(self):
        ret = await async_add(1, 5)
        self.assertEqual(ret, 6)

    async def test_await_async_fail(self):
        with self.assertRaises(Exception) as e:
            await async_one()
```

get_event_loop()

Method provides an event loop for the test

It is called before each test, by default `aiounittest.AsyncTestCase` creates the brand new event loop everytime. After completion, the loop is closed and then recreated, set as default, leaving asyncio clean.

Note: In the most common cases you don't have to bother about this method, the default implementation is a receommended one. But if, for some reasons, you want to provide your own event loop just override it. Note that `AsyncTestCase` won't close such a loop.

```
class MyTest(aiounittest.AsyncTestCase):

    def get_event_loop(self):
        self.my_loop = asyncio.get_event_loop()
        return self.my_loop
```

2.2 AsyncMockIterator

class aiounittest.mock.**AsyncMockIterator**(*seq*)

Allows to mock asynchronous for-loops.

Note: Supported only in Python 3.6 and newer, uses async/await syntax.

```
from aiounittest import AsyncTestCase
from aiounittest.mock import AsyncMockIterator
from unittest.mock import Mock


async def fetch_some_text(source):
    res = ''
    async for txt in source.paginate():
        res += txt
```

(continues on next page)

(continued from previous page)

```
    return res


class MyAsyncMockIteratorTest(AsyncTestCase):

    async def test_add(self):
        source = Mock()
        mock_iter = AsyncMockIterator([
            'asdf', 'qwer', 'zxcv'
        ])
        source.paginate.return_value = mock_iter

        res = await fetch_some_text(source)

        self.assertEqual(res, 'asdfqwerzxcv')
        mock_iter.assertFullyConsumed()
        mock_iter.assertIterCount(3)
```

assertFullyConsumed()

Whenever *async for* reached the end of the given sequence.

assertIterCount(*expected*)

Checks whenever a number of a mock iteration matches expected.

Parameters **int** (*expected*) – Expected number of iterations

2.3 async_test

aiounittest.**async_test**(*func=None*, *loop=None*)

Runs synchonously given function (coroutine)

Parameters

- **func** (*callable*) – function to run (mostly coroutine)
- **loop** (*event loop of None*) – event loop to use to run *func*

By default the brand new event loop will be created (old closed). After completion, the loop will be closed and then recreated, set as default, leaving asyncio clean.

Note: `aiounittest.async_test` is an alias of `aiounittest.helpers.run_sync`

Function can be used like a *pytest.mark.asyncio* (implementation differs), but it's compatible with `unittest.TestCase` class.

```
import asyncio
import unittest
from aiounittest import async_test

async def add(x, y):
    await asyncio.sleep(0.1)
    return x + y

class MyAsyncTestDecorator(unittest.TestCase):

    @async_test
    async def test_async_add(self):
```

(continues on next page)

(continued from previous page)

```
        ret = await add(5, 6)
        self.assertEqual(ret, 11)
```

Note: If the loop is provided, it won't be closed. It's up to you.

This function is also used internally by `aiounittest.AsyncTestCase` to run coroutines.

2.4 futurized

`aiounittest.futurized(o)`

Makes the given object to be awaitable.

Parameters **o** (*any*) – Object to wrap

Returns awaitable that resolves to provided object

Return type asyncio.Future

Anything passed to `futurized` is wrapped in `asyncio.Future`. This makes it awaitable (can be run with `await` or `yield from`) as a result of await it returns the original object.

If provided object is a Exception (or its sublcass) then the *Future* will raise it on await.

```
fut = aiounittest.futurized('SOME TEXT')
ret = await fut
print(ret)  # prints SOME TEXT

fut = aiounittest.futurized(Exception('Dummy error'))
ret = await fut  # will raise the exception "dummy error"
```

The main goal is to use it with `unittest.mock.Mock` (or `MagicMock`) to be able to mock awaitable functions (coroutines).

Consider the below code

```
from asyncio import sleep

async def add(x, y):
    await sleep(666)
    return x + y
```

You rather don't want to wait 666 seconds, you've gotta mock that.

```
from aiounittest import futurized, AsyncTestCase
from unittest.mock import Mock, patch

import dummy_math

class MyAddTest(AsyncTestCase):

    async def test_add(self):
        mock_sleep = Mock(return_value=futurized('whatever'))
        patch('dummy_math.sleep', mock_sleep).start()
        ret = await dummy_math.add(5, 6)
```

(continues on next page)

(continued from previous page)

```
        self.assertEqual(ret, 11)
        mock_sleep.assert_called_once_with(666)

    async def test_fail(self):
        mock_sleep = Mock(return_value=futurized(Exception('whatever')))
        patch('dummy_math.sleep', mock_sleep).start()
        with self.assertRaises(Exception) as e:
            await dummy_math.add(5, 6)
        mock_sleep.assert_called_once_with(666)
```

CHAPTER 3

License


```
MIT License

Copyright (c) 2017-2019 Krzysztof Warunek

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

Index

A

`assertFullyConsumed()` (*aiounittest.mock.AsyncMockIterator method*), 7
`assertIterCount()` (*aiounittest.mock.AsyncMockIterator method*), 7
`async_test()` (*in module aiounittest*), 7
`AsyncMockIterator` (*class in aiounittest.mock*), 6
`AsyncTestCase` (*class in aiounittest*), 5

F

`futurized()` (*in module aiounittest*), 8

G

`get_event_loop()` (*aiounittest.AsyncTestCase method*), 6