asynctest Documentation

Release 0.12.3

Martin Richard

Apr 19, 2019

Contents

The package asynctest is built on top of the standard `unittest` module and cuts down boilerplate code when testing libraries for `asyncio`.

asynctest imports the standard unittest package, overrides some of its features and adds new ones. A test author can import asynctest in place of `unittest` safely.

It is divided in submodules, but they are all imported at the top level, so `asynctest.case.TestCase` is equivalent to *`asynctest.TestCase`*.

Currently, asynctest targets the "selector" model. Hence, some features will not (yet) work with Windows' proactor.

This documentation contains the reference of the classes and functions defined by asynctest, and an introduction guide.

CHAPTER 1

Tutorial

1.1 Introduction

Asynctest is a library which extends the standard pacakge `unittest` to support asyncio features.

This tutorial aims at gathering examples showing how to use *`asynctest`*. It is not a comprehensive documentation and doesn't explain the concepts of `asyncio`.

Some basic patterns of `unittest` are covered. However, if you are not familiar with `unittest`, it's probably a good idea to read its documentation first.

Note: This documentation has not yet been reviewed. The code samples and examples have been tested by the author but probably deserve (at least) a second look.

This tutorial can be improved and probably contains mistakes, typos and incorrect sentences. It can be considered as an "early release". We invite you to open issues or pull-requests on Github.

1.2 Test cases

1.2.1 Writing and running a first test

Tests are written in classes inheriting *`TestCase`*. A test case consists of:

- some set-up which prepares the environment and the resources required for the test to run,
- a list of assertions, usually as a list of checks that must be verified to mark the test as successful,
- some finalization code which cleans the resources used during the test. It should revert the environment back to its state before the set-up.

Let's look at a minimal example:

```
class MinimalExample(asynctest.TestCase):
    def test_that_true_is_true(self):
        self.assertTrue(True)
```

In this example, we created a test which contains only one assertion: it ensures that `True` is, well, true.

`assertTrue()` is a method of *`TestCase`*. If the test is successful, it does nothing. Else, it raises an `AssertionError`.

The documentation of `unittest` lists assertion methods implemented by `unittest.TestCase`. *`asynctest.TestCase`* adds some more for asynchronous code.

We can run it by creating an instance of our test case, its constructor takes the name of the test method as argument:

```
>>> test_case = MinimalExample("test_that_true_is_true")
>>> test_case.run()
<unittest.result.TestResult run=1 errors=0 failures=0>
```

To make things more convenient, `unittest` provides a test runner script. The runner discovers test methods in a module (or package, or class) by looking up methods with a name prefixed by `test_` in `TestCase` subclasses:

```
$ python -m unittest test_cases
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

The runner will create and run an instance of the test case (as shown in the code above) for each method that it finds. This means that you can add as many test methods to your *`TestCase`* class as you want.

1.2.2 Test setup

Let's work on a slightly more complex example:

```
class AnExampleWithSetup(asynctest.TestCase):
    async def a_coroutine(self):
        return "I worked"

    def test_that_a_coroutine_runs(self):
        my_loop = asyncio.new_event_loop()
        try:
            result = my_loop.run_until_complete(self.a_coroutine())
            self.assertIn("worked", result)
        finally:
            my_loop.close()
```

Here, we create a loop that will run a coroutine, ensure that the result of this coroutine is as expected (it should return an object containing the string `"worked"` somewhere). Then we close the loop, even if an exception was raised.

If we happen to write several test methods, the set-up and clean-up will likely be repeated several times. It's probably more convenient to move these parts into their own methods.

We can override two methods of the *`TestCase`* class: `setUp()` and `tearDown()` which will be respectively called before the test method and after the test method:

```
class AnExampleWithSetupMethod(asynctest.TestCase):
    async def a_coroutine(self):
        return "I worked"

    def setUp(self):
        self.my_loop = asyncio.new_event_loop()

    def test_that_a_coroutine_runs(self):
        result = self.my_loop.run_until_complete(self.a_coroutine())
        self.assertIn("worked", result)

    def tearDown(self):
        self.my_loop.close()
```

Both examples are very similar: *TestCase* will run *tearDown()* even if an exception is raised in the test method.

However, if an exception is raised in `setUp()`, the test execution is aborted and `tearDown()` will never run. If the setup fails in between the initialization of several resources, some of them will never be cleaned.

This problem can be solved by registering clean-up callbacks which will always be executed. A clean-up callback is a function without (required) arguments that is passed to `addCleanup()`.

Using this feature, we can rewrite our previous example:

```
class AnExampleWithSetupAndCleanup(asynctest.TestCase):
    async def a_coroutine(self):
        return "I worked"

    def setUp(self):
        self.my_loop = asyncio.new_event_loop()
        self.addCleanup(self.my_loop.close)

    def test_that_a_coroutine_runs(self):
        result = self.my_loop.run_until_complete(self.a_coroutine())
        self.assertIn("worked", result)
```

Tests should always run isolated from the others, this is why tests should only rely on local resources created for the test itself. This ensures that a test will not impact the execution of other tests, and can greatly help to get an accurate diagnostic when debugging a failing test.

It's also worth noting that the order in which tests are executed by the test runner is undefined. It can lead to unpredictable behaviors if tests share some resources.

1.2.3 Testing asynchronous code

Speaking of tests isolation, it's usually preferable to create one loop per test. If the loop is shared, one test could (for instance) schedule a task and never await its result, the task would then run (and possibly trigger unexpected side effects) in another test.

asynctest.TestCase will create (and clean) an event loop for each test that will run. This loop is set in the *loop* attribute. We can use this feature and rewrite the previous example:

```
class AnExampleWithTestCaseLoop(asynctest.TestCase):
    async def a_coroutine(self):
        return "I worked"

    def test_that_a_coroutine_runs(self):
```

(continues on next page)

(continued from previous page)

```
        result = self.loop.run_until_complete(self.a_coroutine())
        self.assertIn("worked", result)
```

Tests functions can be coroutines. *TestCase* will schedule them on the loop.

```
class AnExampleWithTestCaseAndCoroutines(asynctest.TestCase):
    async def a_coroutine(self):
        return "I worked"

    async def test_that_a_coroutine_runs(self):
        self.assertIn("worked", await self.a_coroutine())
```

`setUp()` and `tearDown()` can also be coroutines, they will all run in the same loop.

```
class AnExampleWithAsynchronousSetUp(asynctest.TestCase):
    async def setUp(self):
        self.queue = asyncio.Queue(maxsize=1)
        await self.queue.put("I worked")

    async def test_that_a_lock_is_acquired(self):
        self.assertTrue(self.queue.full())

    async def tearDown(self):
        while not self.queue.empty():
            await self.queue.get()
```

Note: The functions `setUpClass()`, `setUpModule()` and their `tearDown` counterparts can not be coroutine. This is because the loop only exists in an instance of *TestCase*.

In practice, these methods should be avoided because they will not allow to reset the environment between tests.

1.2.4 Automated checks

Asynchronous code introduces a class of subtle bugs which can be hard to detect. In particular, clean-up of resources is often performed asynchronously and can be missed in tests.

TestCase can check and fail if some callbacks or resources are still pending at the end of a test.

These checks can be configured with the decorator *fail_on()*.

```
class AnExempleWhichDetectsPendingCallbacks(asynctest.TestCase):
    def i_must_run(self):
        pass  # do something

    @asynctest.fail_on(active_handles=True)
    async def test_missing_a_callback(self):
        self.loop.call_later(1, self.i_must_run)
```

This test will fail because the test don't wait long enough or doesn't cancel the callback `i_must_run()`, scheduled to run in 1 second:

```
======================================================================
FAIL: test_missing_a_callback (tutorial.test_cases.
↪AnExempleWhichDetectsPendingCallbacks)
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/martius/Code/python/asynctest/asynctest/case.py", line 300, in run
    self._tearDown()
  File "/home/martius/Code/python/asynctest/asynctest/case.py", line 262, in _tearDown
    self._checker.check_test(self)
  File "/home/martius/Code/python/asynctest/asynctest/_fail_on.py", line 90, in check_
↪test
    getattr(self, check)(case)
  File "/home/martius/Code/python/asynctest/asynctest/_fail_on.py", line 111, in␣
↪active_handles
    case.fail("Loop contained unfinished work {!r}".format(handles))
AssertionError: Loop contained unfinished work (<TimerHandle when=3064.258340775␣
↪AnExempleWhichDetectsPendingCallbacks.i_must_run()>,)

----------------------------------------------------------------------
```

Some convenient decorators can be used to enable of disable all checks: *strict()* and *lenient()*.

All decorators can be used on a class or test function.

1.2.5 Conclusion

TestCase provides handy features to test coroutines and asynchronous code.

In the next section, we will talk about mocks. Mocks are objects simulating the behavior of other objects.

1.3 Mocking

Mocks are objects whose behavior can be controlled and which record how they are used. They are very commonly used to write tests. The next section presents the concept of a mock with an example. The rest of the chapter presents the features of *asynctest.mock*.

1.3.1 Using mocks

Let's have a look at a function to be tested.

```
def cache_users(client, cache):
    """
    Load the list of users from a distant server accessed with ``client``,
    add them to ``cache``.

    Notify the server about the number of new users put in the cache, and
    returns this number.

    :param client: a connection to the distant server
    :param cache: a dict-like object
    """
    users = client.get_users()

    nb_users_cached = 0
```

(continues on next page)

(continued from previous page)

```
    for user in users:
        if user.id not in cache:
            nb_users_cached += 1
            cache[user.id] = user

    client.increase_nb_users_cached(nb_users_cached)

    logging.debug("added %d users to the cache %r", nb_users_cached, cache)

    return nb_users_cached
```

Even if the implementation of this function is correct, it can fail. For instance, `client.get_users()` performs calls to a distant server, which can fail temporarily.

It would also be complicated to create multiple test cases if the result of `client.get_users()` can't be controlled inside the tests.

One can solve this problem by crafting a stub object:

```
class StubClient:
    User = collections.namedtuple("User", "id username")

    def __init__(self, *users_to_return):
        self.users_to_return = []
        self.users_to_return.extend(users_to_return)

        self.nb_users_cached = 0

    def add_user(self, user):
        self.users_to_return.append(user)

    def get_users(self):
        return self.users_to_return

    def increase_nb_users_cached(self, nb_cached):
        self.nb_users_cached += nb_cached
```

Tests can be written with this object.

```
class TestUsingStub(asynctest.TestCase):
    def test_one_user_added_to_cache(self):
        user = StubClient.User(1, "a.dmin")
        client = StubClient(user)
        cache = {}

        # The user has been added to the cache
        nb_added = cache_users(client, cache)

        self.assertEqual(nb_added, 1)
        self.assertEqual(cache[1], user)

        # The user was already there
        nb_added = cache_users(client, cache)
        self.assertEqual(nb_added, 0)
        self.assertEqual(cache[1], user)

    def test_no_users_to_add(self):
```

(continues on next page)

(continued from previous page)

```
        cache = {}
        nb_added = cache_users(StubClient(), cache)

        self.assertEqual(nb_added, 0)
        self.assertEqual(len(cache), 0)
```

This will work correctly but has a few downsides. One of them is very practical: each time the interface of the stubbed class change, the stub must be updated.

There is also a bigger problem. In our example, `test_no_users_to_add()` might miss a bug. If `cache_users()` doesn't call `client.get_users()`, no user is added to the cache, yet all the assertions in the test are checked.

In this example, the bug would be detected thanks to the other test. However, it might not be the case with a more complex implementation. The key to write a a better test is to enforce all the assumtions and requirements stated in the documentation.

Currently, the test can be described this way:

> knowing that:
>
> - `client.get_users()` will return an empty result,
> - and that the cache is empty,
>
> a call to `cache_users()` must leave the cache empty.

Instead, it should be:

> knowing that:
>
> - `client.get_users()` will return an empty result,
> - and that the cache is empty,
>
> a call to `cache_users()` *must have queried the client* and must leaves the cache empty.

Mocks solve both of the issues discussed above. A mock can be configured to act like an actual object, and provides assertion methods to verify how the object has been used.

We can also leverage the mock to test another statement of the documentation and make the test even more accurate. We will verify that the server is indeed notified of the number of users added to the cache.

```
class TestUsingMock(asynctest.TestCase):
    def test_no_users_to_add(self):
        client = asynctest.Mock(Client())
        client.get_users.return_value = []
        cache = {}

        nb_added = cache_users(client, cache)

        client.get_users.assert_called()
        self.assertEqual(nb_added, 0)
        self.assertEqual(len(cache), 0)

        client.increase_nb_users_cached.assert_called_once_with(0)
```

In this example, client is a *Mock*. This mock will reproduce the interface of `Client()` (an instance of the `Client` class, ommited for simplicity, available in the example file *tutorial/mocking.py*).

By default, the attributes of a mock object are themselves mocks. We call them *child mocks*. In the above example, `client.get_users` is configured to return an empty list when called. By default, a new mock object would have been returned instead.

Later, `client.get_users.assert_called()` verifies that the method has been called. `client.increase_nb_users_cached.assert_called_once_with(1)` verifies that this method has been called, and that the right arguments have been provided.

Mocks are powerful and can be configured in many ways. Unfortunatly, they can be somewhat complex to use.

The next sections of this chapter will present the features of *asynctest.Mock* related to `asyncio`. It is recommended to be familiar with the module `unittest.mock` before reading the rest of this chapter.

1.3.2 Mocking of coroutines

Let's rewrite the previous example using asyncio.

```
async def cache_users_async(client, cache):
    users = await client.get_users()

    nb_users_cached = 0

    for user in users:
        if user.id not in cache:
            nb_users_cached += 1
            cache[user.id] = user

    await client.increase_nb_users_cached(nb_users_cached)

    logging.debug("added %d users to the cache %r", nb_users_cached, cache)

    return nb_users_cached
```

A mock object can not be awaited (with the `await` keyword). There are several ways to make `client.get_users()` awaitable. One approach is to configure the mock to return a `asyncio.Future` object:

```
class TestUsingFuture(asynctest.TestCase):
    async def test_no_users_to_add(self):
        client = asynctest.Mock(Client())

        client.get_users.return_value = asyncio.Future()
        client.get_users.return_value.set_result([])

        client.increase_nb_users_cached.return_value = asyncio.Future()
        client.increase_nb_users_cached.return_value.set_result(None)

        cache = {}

        nb_added = await cache_users_async(client, cache)

        client.get_users.assert_called()
        self.assertEqual(nb_added, 0)
        self.assertEqual(len(cache), 0)

        client.increase_nb_users_cached.assert_called_once_with(0)
```

`client.get_users()` returns is a future which yields an empty list. It works, but is fairly limited. For instance, if the original `get_users()` is a coroutine function, this is not the case of its mock counterpart.

This test can also miss a new bug now: what if `client.increase_nb_users_cached()` is never awaited? The method has been called, and since the result is a `Future`, this mistake will not be caught if the test runs with asyncio's Debug Mode.

asynctest.CoroutineMock is a type of mock which specializes in mocking coroutine functions (defined with `async def`). A *CoroutineMock* object is not awaitable, but it returns a coroutine instance when called.

It provides assertion methods to ensure it has been awaited, as shown in this example:

```
class TestUsingCoroutineMock(asynctest.TestCase):
    async def test_no_users_to_add(self):
        client = asynctest.Mock(Client())
        client.get_users = asynctest.CoroutineMock(return_value=[])
        client.increase_nb_users_cached = asynctest.CoroutineMock()
        cache = {}

        nb_added = await cache_users_async(client, cache)

        client.get_users.assert_awaited()
        self.assertEqual(nb_added, 0)
        self.assertEqual(len(cache), 0)

        client.increase_nb_users_cached.assert_awaited_once_with(0)
```

All the features of *asynctest.CoroutineMock* are decribed in the reference documentation.

1.3.3 Mocking of other objects

Mock can be configured with the arguments of its constructor. The value of `spec` defines the list of attributes of the mock. *asynctest.Mock* will also detect which attributes are coroutine functions and mock these attributes accordingly.

It means that in the previous example, it was not required to assign *CoroutineMock* objects to `get_users()` and `increase_nb_users_cached()`.

```
async def test_no_users_to_add(self):
    client = asynctest.Mock(AsyncClient())
    client.get_users.return_value = []
    cache = {}

    nb_added = await cache_users_async(client, cache)

    client.get_users.assert_awaited()
    self.assertEqual(nb_added, 0)
    self.assertEqual(len(cache), 0)

    client.increase_nb_users_cached.assert_awaited_once_with(0)
```

Note: *asynctest* will mock an attribute as a *CoroutineMock* if the function is a native coroutine (`async def` function) or a decorated generator (using `asyncio.coroutine()`, before Python 3.5).

Some libraries document function or methods as coroutines, while they are actually implemented as simple functions returning an awaitable object (like `asyncio.Future`).

In this case, *asynctest* can not detect that it should be mocked with *CoroutineMock*.

`spec` defines the attributes of the mock, but isn't passed to child mocks. In particular, using a class as `spec` will not reproduce the behavior of a constructor:

```
>>> ClientMock = asynctest.Mock(Client)
<Mock spec='Client' id='140657386768816'>
>>> ClientMock()
<Mock name='mock()' id='140657394808144'>
>>> ClientMock().get_users
<Mock name='mock().get_users' id='140657394808144'>
```

In this example, `ClientMock` should mock the `Client` class, but `ClientMock()` doesn't return a mock specified as a `Client` instance, and thus, `ClientMock().get_users` is not mocked as a coroutine. We need autospeccing to fix this.

1.3.4 Autospeccing

As the documentation of `unittest` says it, *`create_autospec()`* creates mock objects that have the same attributes and methods as the objects they are replacing. Any functions and methods (including constructors) have the same call signature as the real object.

It is the best solution to configure mocks to behave accurately like the object they replace.

The mock of a function or coroutine must be called with the right arguments:

```
async def test_functions_and_coroutines_arguments_are_checked(self):
    client = asynctest.Mock(Client())
    cache = {}

    cache_users_mock = asynctest.create_autospec(cache_users_async)

    with self.subTest("create_autospec returns a regular mock"):
        await cache_users_mock(client, cache)
        cache_users_mock.assert_awaited_once_with(client, cache)

    with self.subTest("an exception is raised when the mock is called "
                      "with the wrong number of arguments"):
        with self.assertRaises(TypeError):
            await cache_users_mock("wrong", "number", "of", "args")
```

Note: This example also shows the use of `assertRaises()`, which is successful only if an exception is raised in the `with` block.

`subTest()` is used to document in a human-readable format which case is tested. It doesn't change the outcome of the test. The message is displayed if an assertion fails, which is especially useful to understand faster which part of the test breaks.

`create_autospec()` will mock the constructor of a class as expected. When called, it returns a mock with the spec of the class:

```
async def test_create_autospec_on_a_class(self):
    AsyncClientMock = asynctest.create_autospec(AsyncClient)
    client = AsyncClientMock()

    with self.subTest("the mock of a class returns a mock instance of "
                      "the class"):
```

(continues on next page)

(continued from previous page)

```
        self.assertIsInstance(client, AsyncClient)

    with self.subTest("attributes of the mock instance are correctly "
                      "mocked as coroutines"):
        await client.increase_nb_users_cached(1)
```

1.3.5 Types of mocks

There are several types of mocks with slightly different features:

- *Mock* is the base mock type.
- *MagicMock*, it is very similar to *Mock*, except that magic methods are also mocks, and can be configured:

```
>>> asynctest.Mock().__hash__
<method-wrapper '__hash__' of Mock object at 0x7fb514e3a748>
>>> asynctest.MagicMock().__hash__
<MagicMock name='mock.__hash__' id='140415716319528'>
>>> asynctest.MagicMock().__hash__.return_value = "custom value"
```

- *NonCallableMock* and NonCallableMagicMock are their non-callable counterparts. It's usually better to use them when mocking objects or values.
- *CoroutineMock* mocks a coroutine function (or, more generaly, any callable object returning an awaitable).

As mentioned before, a *child mock* is a mock attached to another mock. The child mock is either an attribute of the parent mock, or the result of a call to the parent mock. This relationship enables some features documented in the documentation of unittest.mock.Mock.

Attaching a child mock is just a matter of setting the right attribute:

```
client_mock = asynctest.Mock()
# manually attaching a child mock to get_users
mock.get_users = asynctest.Mock()
# manually attaching the returned child mock to get_users()
mock.get_users.return_value = asynctest.NonCallableMock()
```

By default, the child mock is the result of the factory method _get_child_mock(), and its result depend on the type of mock:

parent mock	child mock
Mock	*Mock*
MagicMock	*MagicMock*
NonCallableMock	*Mock*
NonCallableMagicMock	*MagicMock*
CoroutineMock	*MagicMock*

1.3.6 Controlling the result of CoroutineMock

Calling a *CoroutineMock* returns a coroutine which can be awaited.

The result of this coroutine can be configured like the result of a call to a mock.

return_value

The simplest way to configure the result of a mock is to set its `return_value` attribute. This result will always be returned as it is.

```
async def test_result_set_with_return_value(self):
    coroutine_mock = asynctest.CoroutineMock()
    result = object()
    coroutine_mock.return_value = result

    # return the expected result
    self.assertIs(result, await coroutine_mock())
    # always return the same result
    self.assertIs(await coroutine_mock(), await coroutine_mock())
```

side_effect

The `side_effect` attribute of a mock enables more control over the result of the mock. If set, it has priority over `return_value`, which is ignored.

The value of `side_effect` can be a function. In this case, the call to the mock is forwarded to this function, and its result is returned.

```
async def test_result_with_side_effect_function(self):
    def uppercase_all(*args):
        return tuple(arg.upper() for arg in args)

    coroutine_mock = asynctest.CoroutineMock()
    coroutine_mock.side_effect = uppercase_all

    self.assertEqual(("FIRST", "CALL"),
                     await coroutine_mock("first", "call"))
    self.assertEqual(("A", "SECOND", "CALL"),
                     await coroutine_mock("a", "second", "call"))
```

If the side effect is an exception object or class, this exception is raised.

```
async def test_result_with_side_effect_exception(self):
    coroutine_mock = asynctest.CoroutineMock()
    coroutine_mock.side_effect = NotImplementedError

    # Raise an exception of the configured type
    with self.assertRaises(NotImplementedError):
        await coroutine_mock("any", "number", "of", "args")

    coroutine_mock.side_effect = Exception("an instance of exception")

    # Raise the exact specified object
    with self.assertRaises(Exception) as context:
        await coroutine_mock()

    self.assertIs(coroutine_mock.side_effect, context.exception)
```

Last but not least, `side_effect` can be any iterable object. In this case, the mock will return each value once, until the iterator is exhausted and `StopIteration` is raised to the caller.

`itertools.cycle()` allows to repeat the iterator.

```
async def test_result_with_side_effect_iterable(self):
    coroutine_mock = asynctest.CoroutineMock()
    coroutine_mock.side_effect = ["one", "two", "three"]

    self.assertEqual("one", await coroutine_mock())
    self.assertEqual("two", await coroutine_mock())
    self.assertEqual("three", await coroutine_mock())

    coroutine_mock.side_effect = itertools.cycle(["odd", "even"])
    self.assertEqual("odd", await coroutine_mock())
    self.assertEqual("even", await coroutine_mock())
    self.assertEqual("odd", await coroutine_mock())
    self.assertEqual("even", await coroutine_mock())
```

Important: If the value of `side_effect` is a coroutine function or a generator function, it is treated as a regular function.

The result of a call to this mock will be an instance of the coroutine or generator.

As of asynctest 0.12, specifying a coroutine function as the side effect of *`CoroutineMock`* is undefined and should be avoided. See Github issue #31.

Wrapped object

A mock can also wrap an object. This wrapped object is defined as an argument passed to the constructor of the mock.

When a mock or any of its attributes is called, the call is forwarded to the wrapped object, like if it was the value of `side_effect`. If `side_effect` or `return_value` are set for the mock, they will have priority over the wrapper.

In practice, this is equivalent to adding the features of a *`Mock`* to a stub object.

```
async def test_result_with_wrapped_object(self):
    stub = StubClient()
    mock = asynctest.Mock(stub, wraps=stub)
    cache = {}

    stub.add_user(StubClient.User(1, "a.dmin"))
    cache_users(mock, cache)

    mock.get_users.assert_called()
    self.assertEqual(stub.users_to_return, mock.get_users())
```

1.3.7 Asynchronous iterators and context managers

Python 3.5 introduced the support for asynchronous iterators and context managers. They can be implemented with the magic methods `__aiter__()`, `__anext__()`, `__aenter__()`, `__aexit__()` as described in **PEP 0492#asynchronous-context-managers-and-async-with**.

`MagicMock` will mock these methods and greatly simplify their configuration.

In the example we used so far, we assumed that `client.get_users()` loads all users from a database and store them in a list that it will return. This implementation may consume a lot of memory if there are a lot of users to return. We can instead use a *cursor*.

A cursor is an object *pointing to* the result of the query *get all users* on the database. It keeps an open connection to the database and fetches the objects lazily (only when they are really needed). It allows to load the users one by one from the database, and avoid filling the memory with all users at once.

It is also common to wrap several related queries to a database in a transaction to ensure the sequence of calls is consistent. A better implementation of `cache_users()` should keep the calls to `get_users()` and `increase_nb_users_cached()` in the same transaction.

The `cache_users()` implementation will look like this:

```
async def cache_users_with_cursor(client, cache):
    nb_users_cached = 0

    async with client.new_transaction() as transaction:
        users_cursor = transaction.get_users_cursor()

        async for user in users_cursor:
            if user.id not in cache:
                nb_users_cached += 1
                cache[user.id] = user

        await transaction.increase_nb_users_cached(nb_users_cached)

    logging.debug("added %d users to the cache %r", nb_users_cached, cache)

    return nb_users_cached
```

`client.new_transaction()` returns a transaction object. Under the hood, `async with` calls its coroutine method `__aenter__()` and the result is stored in the variable `transaction`.

`users_cursor` is an asynchronously iterable object. It implements the method `__aiter__()`, which returns an asynchronous iterator. `__aiter__()` is a function, not a coroutine. For each iteration of the `async for` loop, the coroutine method `__anext__()` of the asynchronous iterator is called and its result is assigned to `user`.

When the interpreter leaves the `async with` block, `__aexit__()` is called.

A partial implementation of this logic can be found in the example file *tutorial/mocking.py*.

The next sections show how to use *MagicMock* to test this method.

Asynchronous context manager

MagicMock mocks `__aenter__` with a *CoroutineMock* returning a new child mock.

If an exception is raised in an `async with` block, this exception is passed to `__aexit__()`. In this case, the return value defines wether the interpreter suppresses or propagates the exception, as described in the documentation of `object.__exit__()`.

MagicMock mocks `__aexit__` with a *CoroutineMock* returning `False` by default, which means that the exception is propagated.

By default, we can use a *MagicMock* in an `async with` block without configuration, exceptions raised in this block are propagated:

```
async def test_context_manager(self):
    with self.assertRaises(AssertionError):
        async with asynctest.MagicMock() as context:
            # context is a MagicMock
            context.assert_called()
```

However, in the example above, the `transaction` object exposes the same methods as `client`. In particular, We must configure this mock so `transaction.increase_nb_users_cached()` is a coroutine.

Asynchronous iterator

The method `__aiter__()` of a *MagicMock* returns an asynchronous iterator. By default, this iterator is empty.

```
async def test_empty_iterable(self):
    loop_iterations = 0
    async for _ in asynctest.MagicMock():
        loop_iterations += 1

    self.assertEqual(0, loop_iterations)
```

The values yielded by the iterator can be configured by setting the `return_value` of `__aiter__`. This value must be an iterable object, such as a list or a generator:

```
async def test_iterable(self):
    loop_iterations = 0
    mock = asynctest.MagicMock()
    mock.__aiter__.return_value = range(5)
    async for _ in mock:
        loop_iterations += 1

    self.assertEqual(5, loop_iterations)
```

Note: As of asynctest 0.12, it is not possible to use an asynchronously iterable object as `return_value` for `__aiter__()`.

Setting `side_effect` allows to override the behavior of *MagicMock*.

Putting it all together

We can leverage several features of *asynctest* when testing `cache_users_with_cursor()`:

```
class TestCacheWithMagicMethods(asynctest.TestCase):
    async def test_one_user_added_to_cache(self):
        user = StubClient.User(1, "a.dmin")

        AsyncClientMock = asynctest.create_autospec(AsyncClient)

        transaction = asynctest.MagicMock()
        transaction.__aenter__.side_effect = AsyncClientMock

        cursor = asynctest.MagicMock()
        cursor.__aiter__.return_value = [user]

        client = AsyncClientMock()
        client.new_transaction.return_value = transaction
        client.get_users_cursor.return_value = cursor

        cache = {}

        # The user has been added to the cache
```

(continues on next page)

(continued from previous page)

```
        nb_added = await cache_users_with_cursor(client, cache)

        self.assertEqual(nb_added, 1)
        self.assertEqual(cache[1], user)

        # The user was already there
        nb_added = await cache_users_with_cursor(client, cache)
        self.assertEqual(nb_added, 0)
        self.assertEqual(cache[1], user)
```

This example deserve some explanation.

First, we use *create_autospec()* to build a mock of the class *AsyncClient*.

`transaction` will be the object configured as a context manager. When called with `async with`, it must return an object with an interface as `client`. We set `AsyncClientMock` as a side effect to `transaction.__aenter__`, which means that a new mock of an instance of `AsyncClient` will be issued each time `transaction` is used in an `async width` block.

`cursor` will be used in the `async for` loop. The iterator will yield the values of `cursor.__aiter__.return_value`. We set to a list containing a single `User` object. A new iterator is created each time an `async for` loop is called upon the cursor, it is safe to use this mock several times.

We then create `client`, a mock created from `AsyncClientMock`. We configure it so the return values of `client.new_transaction()` and `client.get_users_cursor()` are the mocks we created above.

Note that we configured the behavior of `client`'s attributes, not those of `AsyncClientMock`. This is because the child mock of an autospecced class will not inherit the behavior of the parent mock, only its spec.

1.3.8 Patching

Patching is a mechanism allowing to temporarily replace a symbol (class, object, function, attribute, ...) by a mock, in-place. It is especially useful when one need a mock, but can't pass it as a parameter of the function to be tested.

For instance, if `cache_users()` didn't accept the `client` argument, but instead created a new client, it would not be possible to replace it by a mock like in all the previous examples.

When an object is hard to mock, it sometimes shows a limitation in the design: a coupling that is too tight, the use of a global variable (or a singleton), etc. However, it's not always possible or desirable to change the code to accomodate the tests. A common situation where tight coupling is almost invisible is when performing logging or monitoring. In this case, patching will help.

A *patch()* can be used as a context manager. It will replace the target (`logging.debug()`) with a mock during the lifetime of the `with` block.

```
async def test_with_context_manager(self):
    client = asynctest.Mock(AsyncClient())
    cache = {}

    with asynctest.patch("logging.debug") as debug_mock:
        await cache_users_async(client, cache)

    debug_mock.assert_called()
```

Alternatively, *patch()* can be used to decorate a test or a test class (inheriting *TestCase*). This second example is roughly equivalent to the previous one. The main difference is that for all tests affected by the patch (the decorated

method or all test methods in a decorated test class) must accept an additional argument which will receive the mock object used by the patch.

Note that when using multiple decorators on a single method, the order of the arguments is inversed compared to the order of the decorators. This is due to the way decorators work in Python, a topic which we don't cover in this documentation.

```
@asynctest.patch("logging.error")
@asynctest.patch("logging.debug")
async def test_with_decorator(self, debug_mock, error_mock):
    client = asynctest.Mock(AsyncClient())
    cache = {}

    await cache_users_async(client, cache)

    debug_mock.assert_called()
    error_mock.assert_not_called()
```

Note: In practice, we should have used `unittest.TestCase.assertLogs()`. It asserts that a given message have been logged and makes more sense than manually patching `logging`.

There are variants of *`patch()`*:

- *`asynctest.patch.object()`* patches the attribute of a given object,
- *`asynctest.patch.multiple()`* patches several attributes of a given object,
- *`asynctest.patch.dict()`* patches the values in a `dict` for the given indices.

The official python documentation provide extensive details about how to define the target of a patch in its section Where to patch.

Scope of the patch

There is one hidden catch in the examples above: what happens to the patch when the interpreter reaches the `await` statement and pauses the coroutine?

When patch is used as a context manager, the patch stays active until the interpreter reached the end of the `with` block.

When used as a decorator, the patch is activated right before the function (or coroutine) is executed, and deactivated once it returned. This is equivalent to englobing the body of the function in a `with` statement instead of using the decorator.

However, since couroutines are asynchronous, the work performed by the interpreter while the coroutine is paused is unpredictable. In some cases, the patch can conflict with something else, and must only be active when the patched coroutine is running.

It is possible to control when a *`asynctest.patch()`* must be active when applied to a coroutine with the argument `scope`.

If `scope` is set to *`asynctest.LIMITED`*, the patch is active only when the coroutine is running.

This situation is illustrated in the example bellow. The test case `TestMustBePatched` runs a task in background which fails if some patch is active. It contains two tests: one which shows the test conflicting, and one which uses the *`LIMITED`* `scope` to deactivate the patch outside of the test coroutine.

```
class TestMustBePatched(asynctest.TestCase):
    async def setUp(self):
        # Event used to track if the background task checked if the patch
        # is active
        self.checked = asyncio.Event()

        # This task checks if the object is patched continuously, and sets
        # the checked event everytime it does so.
        self.background_task = asyncio.create_task(
            must_be_patched.crash_if_patched(self.checked))

        # Any test will fail if the background task raises an exception
        self.addCleanup(terminate_and_check_task, self.background_task)

    @asynctest.patch.object(must_be_patched, "is_patched",
                            return_value=True)
    async def test_patching_conflicting(self, _):
        # This call blocks until the check happened once in background
        await happened_once(self.checked)
        self.assertTrue(await must_be_patched.is_patched())
        await happened_once(self.checked)

    @asynctest.patch.object(must_be_patched, "is_patched",
                            return_value=True, scope=asynctest.LIMITED)
    async def test_patching_not_conflicting(self, _):
        await happened_once(self.checked)
        self.assertTrue(await must_be_patched.is_patched())
        await happened_once(self.checked)
```

In this example, `happened_once()` pauses the coroutine until the background task checked once that the patch is not active. The code of `must_be_patched`, `happened_once()` and `terminate_and_check_task()` is available in the example file *tutorial/patching.py*.

`test_patching_conflicting()` fails because the patch is still active when it is paused and aways the `self.checked` event. While paused, the background task runs, and crashes because the patch is still active.

In `test_patching_not_conflicting()`, the patch is set with a *`LIMITED`* scope, and is active only when the coroutine runs. When `await must_be_patched.is_patched()` runs, the patch is still active. This coroutine runs in the scope of the outher coroutine (the test): indeed, `must_be_patched.is_patched()` is scheduled in the task running the test.

1.3.9 Conclusion

This chapter showed most of the concepts and features of mock relevant when testing asynchronous code. There are plenty of other features and subtleties which are covered in the documentation of `unittest.mock`.

1.4 Advanced Features

This chapter describes miscellaneous features of *`asynctest`* which can be leveraged in specific use cases.

1.4.1 Controlling time

Tests running calls to `asyncio.sleep()` will take as long as the sum of all these calls. These calls are frequent when testing for timeouts, for instance.

In many cases, this will add a useless delay to the execution of the test suite, and encourage us to deactivate or ignore these tests.

`ClockedTestCase` is a subclass of *`TestCase`* which allows to advance the clock of the loop in a test with the coroutine *`advance()`*.

This will not affect the wall clock: functions like `time.time()` or `datetime.datetime.now()` will return the regular date and time of the system.

```
class TestAdvanceTime(asynctest.ClockedTestCase):
    async def test_advance_time(self):
        base_loop_time = self.loop.time()
        base_wall_time = time.time()

        await self.advance(10)

        self.assertEqual(base_loop_time + 10, self.loop.time())
        self.assertTrue(is_time_around(base_wall_time))
```

This example is pretty self-explanatory: we verified that the clock of the loop advanced as expected, without awaiting 10 actual seconds and changing the time of the wall clock.

Internally, *`ClockedTestCase`* will ensure that the loop is executed as if time was passing *fast*, instead of jumping the clock to the target time.

```
class TestWithClockAndCallbacks(asynctest.ClockedTestCase):
    results = None

    def runs_at(self, expected_time):
        self.results.append(is_time_around(expected_time, self.loop))

    @asynctest.fail_on(active_handles=True)
    async def test_callbacks_executed_when_expected(self):
        self.results = []

        base_time = self.loop.time()
        self.loop.call_later(1, self.runs_at, base_time + 1)
        self.loop.call_at(base_time + 7, self.runs_at, base_time + 7)

        # This shows that the callback didn't run yet
        self.assertEqual(0, len(self.results))

        await self.advance(10)

        # This shows that the callbacks ran...
        self.assertEqual(2, len(self.results))
        # ...when expected
        self.assertTrue(all(self.results))
```

This example schedules function calls to be executed later by the loop. Each call will verify that it runs at the expected time. `@fail_on(active_handles=True)` ensures that the callbacks have been executed when the test finishes.

The source code of `is_time_around()` can be found in the example file *tutorial/clock.py*.

1.4.2 Mocking I/O

Testing libraries or functions dealing with low-level IO objects may be complex: these objects are outside of our control, since they are owned by the kernel. It can be impossible to exactly predict their behavior and simulate edge-cases, such as the ones happening in a real-world scenario in a large network.

Even worse, using mocks in place of files will often raise `OSError` because these obhjects are not compatible with the features of the system used by the loop.

asynctest provides special mocks which can be used in place of actual file-like objects. They are supported by the loop provided by *TestCase* if the loop uses a standard implementation with a selector (Window's Proactor loop or uvloop are not supported).

These mocks are configured with a spec matching common file-like objects.

Mock	spec
FileMock	a file object, implements `fileno()`
SocketMock	`socket.socket`
SSLSocketMock	`ssl.SSLSocket`

We can use *asynctest.isfilemock()* to differenciate mocks from regular objects.

As of *asynctest* 0.12, these mocks don't provide other features, and must be configured to return expected values for calls to methods like `read()` or `recv()`.

When configured, we still need to force the loop to detect that I/O is possible on these mock files.

This is done with *set_read_ready()* and *set_write_ready()*.

```
class TestMockASocket(asynctest.TestCase):
    async def test_read_and_write_from_socket(self):
        socket_mock = asynctest.SocketMock()
        socket_mock.type = socket.SOCK_STREAM

        recv_data = iter((
            b"some data read",
            b"some other",
            b" ...and the last",
        ))

        recv_buffer = bytearray()

        def recv_side_effect(max_bytes):
            nonlocal recv_buffer

            if not recv_buffer:
                try:
                    recv_buffer.extend(next(recv_data))
                    asynctest.set_read_ready(socket_mock, self.loop)
                except StopIteration:
                    # nothing left
                    pass

            data = recv_buffer[:max_bytes]
            recv_buffer = recv_buffer[max_bytes:]

            if recv_buffer:
                # Some more data to read
```

(continues on next page)

(continued from previous page)

```
                asynctest.set_read_ready(socket_mock, self.loop)

            return data

        def send_side_effect(data):
            asynctest.set_read_ready(socket_mock, self.loop)
            return len(data)

        socket_mock.recv.side_effect = recv_side_effect
        socket_mock.send.side_effect = send_side_effect

        reader, writer = await asyncio.open_connection(sock=socket_mock)

        writer.write(b"a request?")
        self.assertEqual(b"some", await reader.read(4))
        self.assertEqual(b" data read", await reader.read(10))
        self.assertEqual(b"some other ...and the last", await reader.read())
```

In this example, we configure a socket mock to simulate a simple request-response scenario with a TCP (stream) socket. Some data is available to read on the socket once a request has been written. `recv_side_effect()` makes as if the data is received in several packets, but it has no impact on the high level `StreamReader`.

It's common that while a read operation blocks until data is available, a write is often successful. Thus, we didn't bother simulating the case where the congestion control would block the write operation.

1.4.3 Testing with event loop policies

Advanced users may not be able to use the loop provided by `TestCase` because they use a customized event loop policy (see Policies). It is often the case when using an alternative implementation (like uvloop) or if the tests are integrated within a framework hidding the scheduling and management of the loop.

It is possible to force the *`TestCase`* to use the loop provided by the policy by setting the class attribute *`use_default_loop`*.

Conversely, authors of libraries may not want to assume which loop they should use and let users explicitly pass the loop as argument to a function call. For instance, most of the high-level functions of `asyncio` (see Streams, for instance) allow the caller to specify the loop to use if it needs this kind of flexibility.

`forbid_get_event_loop` forbids the use of `asyncio.get_event_loop()`. An exception is raised if the method is called while a test is running. It helps developers to ensure they don't rely on the default loop this their library.

Note: The behavior of `asyncio.get_event_loop()` changed over time. Explicitly passing the loop is not the recommended practice anymore.

CHAPTER 2

Reference

2.1 Module `case`

Enhance `unittest.TestCase`:

- a new loop is issued and set as the default loop before each test, and closed and disposed after,
- if the loop uses a selector, it will be wrapped with *`asynctest.TestSelector`*,
- a test method in a TestCase identified as a coroutine function or returning a coroutine will run on the loop,
- `setUp()` and `tearDown()` methods can be coroutine functions,
- cleanup functions registered with `addCleanup()` can be coroutine functions,
- a test fails if the loop did not run during the test.

2.1.1 class-level set-up

Since each test runs in its own loop, it is not possible to run `setUpClass()` and `tearDownClass()` as coroutines.

If one needs to perform set-up actions at the class level (meaning once for all tests in the class), it should be done using a loop created for this sole purpose and that is not shared with the tests. Ideally, the loop shall be closed in the method which creates it.

If one really needs to share a loop between tests, `TestCase.use_default_loop` can be set to `True` (as a class attribute). The test case will use the loop returned by `asyncio.get_event_loop()` instead of creating a new loop for each test. This way, the event loop or event loop policy can be set during class-level set-up and tear down.

2.1.2 TestCases

class asynctest.**TestCase**(*methodName='runTest'*)

A test which is a coroutine function or which returns a coroutine will run on the loop.

Once the test returned, one or more assertions are checked. For instance, a test fails if the loop didn't run. These checks can be enabled or disabled using the *fail_on()* decorator.

By default, a new loop is created and is set as the default loop before each test. Test authors can retrieve this loop with *loop*.

If *use_default_loop* is set to `True`, the current default event loop is used instead. In this case, it is up to the test author to deal with the state of the loop in each test: the loop might be closed, callbacks and tasks may be scheduled by previous tests. It is also up to the test author to close the loop and dispose the related resources.

If *forbid_get_event_loop* is set to `True`, a call to `asyncio.get_event_loop()` will raise an `AssertionError`. Since Python 3.6, calling `asyncio.get_event_loop()` from a callback or a coroutine will return the running loop (instead of raising an exception).

These behaviors should be configured when defining the test case class:

```
class With_Reusable_Loop_TestCase(asynctest.TestCase):
    use_default_loop = True

    forbid_get_event_loop = False

    def test_something(self):
        pass
```

If *setUp()* and *tearDown()* are coroutine functions, they will run on the loop. Note that `setUpClass()` and `tearDownClass()` can not be coroutines.

New in version 0.5: attribute *use_default_loop*.

New in version 0.7: attribute *forbid_get_event_loop*. In any case, the default loop is now reset to its original state outside a test function.

New in version 0.8: `ignore_loop` has been deprecated in favor of the extensible *fail_on()* decorator.

setUp()
Method or coroutine called to prepare the test fixture.

see `unittest.TestCase.setUp()`

tearDown()
Method called immediately after the test method has been called and the result recorded.

see `unittest.TestCase.tearDown()`

addCleanup(*function*, **args*, ***kwargs*)
Add a function, with arguments, to be called when the test is completed. If function is a coroutine function, it will run on the loop before it's cleaned.

assertAsyncRaises(*exception*, *awaitable*)
Test that an exception of type `exception` is raised when an exception is raised when awaiting `awaitable`, a future or coroutine.

See `unittest.TestCase.assertRaises()`

assertAsyncRaisesRegex(*exception*, *regex*, *awaitable*)
Like *assertAsyncRaises()* but also tests that `regex` matches on the string representation of the raised exception.

See `unittest.TestCase.assertRaisesRegex()`

assertAsyncWarns(*warning*, *awaitable*)
Test that a warning is triggered when awaiting `awaitable`, a future or a coroutine.

See `unittest.TestCase.assertWarns()`

assertAsyncWarnsRegex(*warning*, *regex*, *awaitable*)
Like *assertAsyncWarns()* but also tests that regex matches on the message of the triggered warning.

See unittest.TestCase.assertWarnsRegex()

doCleanups()
Execute all cleanup functions. Normally called for you after tearDown.

forbid_get_event_loop = False
If true, the value returned by asyncio.get_event_loop() will be set to None during the test. It allows to ensure that tested code use a loop object explicitly passed around.

loop = None
Event loop created and set as default event loop during the test.

use_default_loop = False
If true, the loop used by the test case is the current default event loop returned by asyncio.get_event_loop(). The loop will not be closed and recreated between tests.

class asynctest.**FunctionTestCase**(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)
Enables the same features as *TestCase*, but for *FunctionTestCase*.

class asynctest.**ClockedTestCase**(*methodName='runTest'*)
Subclass of *TestCase* with a controlled loop clock, useful for testing timer based behaviour without slowing test run time.

advance(*seconds*)
Fast forward time by a number of seconds.

Callbacks scheduled to run up to the destination clock time will be executed on time:

```
>>> self.loop.call_later(1, print_time)
>>> self.loop.call_later(2, self.loop.call_later, 1, print_time)
>>> await self.advance(3)
1
3
```

In this example, the third callback is scheduled at `t = 2` to be executed at `t + 1`. Hence, it will run at `t = 3`. The callback as been called on time.

2.1.3 Decorators

@asynctest.**fail_on**(***checks*)
Enable checks on the loop state after a test ran to help testers to identify common mistakes.

Enable or disable a check using a keywork argument with a boolean value:

```
@asynctest.fail_on(unused_loop=True)
class TestCase(asynctest.TestCase):
    ...
```

Available checks are:

- unused_loop: disabled by default, checks that the loop ran at least once during the test. This check can not fail if the test method is a coroutine. This allows to detect cases where a test author assume its test will run tasks or callbacks on the loop, but it actually didn't.

- `active_selector_callbacks`: enabled by default, checks that any registered reader or writer callback on a selector loop (with `add_reader()` or `add_writer()`) is later explicitly unregistered (with `remove_reader()` or `remove_writer()`) before the end of the test.
- `active_handles`: disabled by default, checks that there is not scheduled callback left to be executed on the loop at the end of the test. The helper *`exhaust_callbacks()`* can help to give a chance to the loop to run pending callbacks.

The decorator of a method has a greater priority than the decorator of a class. When *`fail_on()`* decorates a class and one of its methods with conflicting arguments, those of the class are overriden.

Subclasses of a decorated *`TestCase`* inherit of the checks enabled on the parent class.

New in version 0.8.

New in version 0.9: `active_handles`

New in version 0.12: `unused_loop` is now deactivated by default to maintain compatibility with non-async test inherited from `unittest.TestCase`. This check is especially useful to track missing `@asyncio.coroutine` decorators in a codebase that must be compatbible with Python 3.4.

`@asynctest.`**`strict`**

Activate strict checking of the state of the loop after a test ran.

It is a shortcut to *`fail_on()`* with all checks set to `True`.

Note that by definition, the behavior of *`strict()`* will change in the future when new checks will be added, and may break existing tests with new errors after an update of the library.

New in version 0.8.

`@asynctest.`**`lenient`**

Deactivate all checks performed after a test ran.

It is a shortcut to *`fail_on()`* with all checks set to `False`.

New in version 0.8.

`@asynctest.`**`ignore_loop`**

By default, a test fails if the loop did not run during the test (including set up and tear down), unless the *`TestCase`* class or test function is decorated by *`ignore_loop()`*.

Deprecated since version 0.8: Use *`fail_on()`* with `unused_loop=False` instead.

2.2 Module `mock`

Wrapper to unittest.mock reducing the boilerplate when testing asyncio powered code.

A mock can behave as a coroutine, as specified in the documentation of `Mock`.

2.2.1 Mock classes

`class` `asynctest.`**`Mock`**(*spec=None*, *side_effect=None*, *return_value=sentinel.DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *parent=None*, *_spec_state=None*, *_new_name=''*, *_new_parent=None*, ***kwargs*)

Enhance `unittest.mock.Mock` so it returns a *`CoroutineMock`* object instead of a *`Mock`* object where a method on a `spec` or `spec_set` object is a coroutine.

For instance:

```
>>> class Foo:
...     @asyncio.coroutine
...     def foo(self):
...         pass
...
...     def bar(self):
...         pass
```

```
>>> type(asynctest.mock.Mock(Foo()).foo)
<class 'asynctest.mock.CoroutineMock'>
```

```
>>> type(asynctest.mock.Mock(Foo()).bar)
<class 'asynctest.mock.Mock'>
```

The test author can also specify a wrapped object with `wraps`. In this case, the *Mock* object behavior is the same as with an `unittest.mock.Mock` object: the wrapped object may have methods defined as coroutine functions.

If you want to mock a coroutine function, use *CoroutineMock* instead.

See *NonCallableMock* for details about *asynctest* features, and `unittest.mock` for the comprehensive documentation about mocking.

class asynctest.**NonCallableMock**(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, *is_coroutine=None*, *parent=None*, ***kwargs*)

Enhance `unittest.mock.NonCallableMock` with features allowing to mock a coroutine function.

If `is_coroutine` is set to `True`, the *NonCallableMock* object will behave so `asyncio.iscoroutinefunction()` will return `True` with `mock` as parameter.

If `spec` or `spec_set` is defined and an attribute is get, *CoroutineMock* is returned instead of *Mock* when the matching spec attribute is a coroutine function.

The test author can also specify a wrapped object with `wraps`. In this case, the *Mock* object behavior is the same as with an `unittest.mock.Mock` object: the wrapped object may have methods defined as coroutine functions.

See `unittest.mock.NonCallableMock`

is_coroutine

True if the object mocked is a coroutine

class asynctest.**MagicMock**(**args*, ***kwargs*)

Enhance `unittest.mock.MagicMock` so it returns a *CoroutineMock* object instead of a *Mock* object where a method on a `spec` or `spec_set` object is a coroutine.

If you want to mock a coroutine function, use *CoroutineMock* instead.

MagicMock allows to mock `__aenter__`, `__aexit__`, `__aiter__` and `__anext__`.

When mocking an asynchronous iterator, you can set the `return_value` of `__aiter__` to an iterable to define the list of values to be returned during iteration.

You can not mock `__await__`. If you want to mock an object implementing __await__, *CoroutineMock* will likely be sufficient.

see *Mock*.

New in version 0.11: support of asynchronous iterators and asynchronous context managers.

class asynctest.**CoroutineMock**(**args*, ***kwargs*)

Enhance `Mock` with features allowing to mock a coroutine function.

The *CoroutineMock* object will behave so the object is recognized as coroutine function, and the result of a call as a coroutine:

```
>>> mock = CoroutineMock()
>>> asyncio.iscoroutinefunction(mock)
True
>>> asyncio.iscoroutine(mock())
True
```

The result of `mock()` is a coroutine which will have the outcome of `side_effect` or `return_value`:

- if `side_effect` is a function, the coroutine will return the result of that function,
- if `side_effect` is an exception, the coroutine will raise the exception,
- if `side_effect` is an iterable, the coroutine will return the next value of the iterable, however, if the sequence of result is exhausted, `StopIteration` is raised immediately,
- if `side_effect` is not defined, the coroutine will return the value defined by `return_value`, hence, by default, the coroutine returns a new *CoroutineMock* object.

If the outcome of `side_effect` or `return_value` is a coroutine, the mock coroutine obtained when the mock object is called will be this coroutine itself (and not a coroutine returning a coroutine).

The test author can also specify a wrapped object with `wraps`. In this case, the *Mock* object behavior is the same as with an `unittest.mock.Mock` object: the wrapped object may have methods defined as coroutine functions.

assert_any_await(**args*, ***kwargs*)
Assert the mock has ever been awaited with the specified arguments.

New in version 0.12.

assert_awaited()
Assert that the mock was awaited at least once.

New in version 0.12.

assert_awaited_once(**args*, ***kwargs*)
Assert that the mock was awaited exactly once.

New in version 0.12.

assert_awaited_once_with(**args*, ***kwargs*)
Assert that the mock was awaited exactly once and with the specified arguments.

New in version 0.12.

assert_awaited_with(**args*, ***kwargs*)
Assert that the last await was with the specified arguments.

New in version 0.12.

assert_has_awaits(*calls*, *any_order=False*)
Assert the mock has been awaited with the specified calls. The *await_args_list* list is checked for the awaits.

If *any_order* is False (the default) then the awaits must be sequential. There can be extra calls before or after the specified awaits.

If *any_order* is True then the awaits can be in any order, but they must all appear in *await_args_list*.

New in version 0.12.

assert_not_awaited()
Assert that the mock was never awaited.

New in version 0.12.

await_args

await_args_list

await_count
Number of times the mock has been awaited.

New in version 0.12.

awaited
Property which is set when the mock is awaited. Its `wait` and `wait_next` coroutine methods can be used to synchronize execution.

New in version 0.12.

reset_mock(**args*, ***kwargs*)
See `unittest.mock.Mock.reset_mock()`

2.2.2 Autospeccing

asynctest.**create_autospec**(*spec*, *spec_set=False*, *instance=False*, *_parent=None*, *_name=None*, ***kwargs*)
Create a mock object using another object as a spec. Attributes on the mock will use the corresponding attribute on the spec object as their spec.

`spec` can be a coroutine function, a class or object with coroutine functions as attributes.

If `spec` is a coroutine function, and `instance` is not `False`, a `RuntimeError` is raised.

New in version 0.12.

2.2.3 Patch

asynctest.**GLOBAL = <PatchScope.GLOBAL: 2>**
An enumeration.

asynctest.**LIMITED = <PatchScope.LIMITED: 1>**
An enumeration.

asynctest.**patch**(*target*, *new=sentinel.DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, *scope=<PatchScope.GLOBAL: 2>*, ***kwargs*)
A context manager, function decorator or class decorator which patches the target with the value given by the `new` argument.

`new` specifies which object will replace the `target` when the patch is applied. By default, the target will be patched with an instance of *`CoroutineMock`* if it is a coroutine, or a *`MagicMock`* object.

It is a replacement to `unittest.mock.patch()`, but using *`asynctest.mock`* objects.

When a generator or a coroutine is patched using the decorator, the patch is activated or deactivated according to the `scope` argument value:

- *`asynctest.GLOBAL`*: the default, enables the patch until the generator or the coroutine finishes (returns or raises an exception),
- *`asynctest.LIMITED`*: the patch will be activated when the generator or coroutine is being executed, and deactivated when it yields a value and pauses its execution (with `yield`, `yield from` or `await`).

The behavior differs from `unittest.mock.patch()` for generators.

When used as a context manager, the patch is still active even if the generator or coroutine is paused, which may affect concurrent tasks:

```
@asyncio.coroutine
def coro():
    with asynctest.mock.patch("module.function"):
        yield from asyncio.get_event_loop().sleep(1)

@asyncio.coroutine
def independent_coro():
    assert not isinstance(module.function, asynctest.mock.Mock)

asyncio.create_task(coro())
asyncio.create_task(independent_coro())
# this will raise an AssertionError(coro() is scheduled first)!
loop.run_forever()
```

Parameters **scope** – *asynctest.GLOBAL* or *asynctest.LIMITED*, controls when the patch is activated on generators and coroutines

When used as a decorator with a generator based coroutine, the order of the decorators matters. The order of the `@patch()` decorators is in the reverse order of the parameters produced by these patches for the patched function. And the `@asyncio.coroutine` decorator should be the last since `@patch()` conceptually patches the coroutine, not the function:

```
@patch("module.function2")
@patch("module.function1")
@asyncio.coroutine
def test_coro(self, mock_function1, mock_function2):
    yield from asyncio.get_event_loop().sleep(1)
```

see `unittest.mock.patch()`.

New in version 0.6: patch into generators and coroutines with a decorator.

`asynctest.patch.`**`object`**(*target*, *attribute*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, *scope=asynctest.GLOBAL*, ***kwargs*)

Patch the named member (`attribute`) on an object (`target`) with a mock object, in the same fashion as *patch()*.

See *patch()* and `unittest.mock.patch.object()`.

`asynctest.patch.`**`multiple`**(*target*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, *scope=asynctest.global*, ***kwargs*)

Perform multiple patches in a single call. It takes the object to be patched (either as an object or a string to fetch the object by importing) and keyword arguments for the patches.

See *patch()* and `unittest.mock.patch.multiple()`.

`asynctest.patch.`**`dict`**(*in_dict*, *values=()*, *clear=False*, *scope=asynctest.GLOBAL*, ***kwargs*)

Patch a dictionary, or dictionary like object, and restore the dictionary to its original state after the test.

Its behavior can be controlled on decorated generators and coroutines with `scope`.

New in version 0.8: patch into generators and coroutines with a decorator.

Parameters

- **in_dict** – dictionary like object, or string referencing the object to patch.
- **values** – a dictionary of values or an iterable of (key, value) pairs to set in the dictionary.
- **clear** – if `True`, in_dict will be cleared before the new values are set.
- **scope** – *asynctest.GLOBAL* or *asynctest.LIMITED*, controls when the patch is activated on generators and coroutines

See *patch()* (details about `scope`) and `unittest.mock.patch.dict()`.

2.2.4 Helpers

`asynctest.`**mock_open**(*mock=None*, *read_data=''*)

A helper function to create a mock to replace the use of `open()`. It works for `open()` called directly or used as a context manager.

Parameters

- **mock** – mock object to configure, by default a *MagicMock* object is created with the API limited to methods or attributes available on standard file handles.
- **read_data** – string for the `read()` and `readlines()` of the file handle to return. This is an empty string by default.

`asynctest.`**return_once**(*value*, *then=None*)

Helper to use with `side_effect`, so a mock will return a given value only once, then return another value.

When used as a `side_effect` value, if one of `value` or `then` is an `Exception` type, an instance of this exception will be raised.

```
>>> mock.recv = Mock(side_effect=return_once(b"data"))
>>> mock.recv()
b"data"
>>> repr(mock.recv())
'None'
>>> repr(mock.recv())
'None'
```

```
>>> mock.recv = Mock(side_effect=return_once(b"data", then=BlockingIOError))
>>> mock.recv()
b"data"
>>> mock.recv()
Traceback BlockingIOError
```

Parameters

- **value** – value to be returned once by the mock when called.
- **then** – value returned for any subsequent call.

New in version 0.4.

2.3 Module `selector`

Mock of `selectors` and compatible objects performing asynchronous IO.

This module provides classes to mock objects performing IO (files, sockets, etc). These mocks are compatible with *TestSelector*, which can simulate the behavior of a selector on the mock objects, or forward actual work to a real selector.

2.3.1 Mocking file-like objects

class asynctest.**FileMock**(**args*, ***kwargs*)
Mock a file-like object.

A FileMock is an intelligent mock which can work with TestSelector to simulate IO events during tests.

fileno()
Return a *FileDescriptor* object.

class asynctest.**SocketMock**(*side_effect=None*, *return_value=sentinel.DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *parent=None*, ***kwargs*)
Bases: asynctest.selector.FileMock

Mock a socket.

See *FileMock*.

class asynctest.**SSLSocketMock**(*side_effect=None*, *return_value=sentinel.DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *parent=None*, ***kwargs*)
Bases: asynctest.selector.SocketMock

Mock a socket wrapped by the ssl module.

See *FileMock*.

New in version 0.5.

class asynctest.**FileDescriptor**
Bases: int

A subclass of int which allows to identify the virtual file-descriptor of a *FileMock*.

If *FileDescriptor()* without argument, its value will be the value of *next_fd*.

When an object is created, *next_fd* is set to the highest value for a *FileDescriptor* object + 1.

next_fd = 0

Helpers

asynctest.**fd**(*fileobj*)
Return the *FileDescriptor* value of fileobj.

If fileobj is a *FileDescriptor*, fileobj is returned, else fileobj.fileno() is returned instead.

Note that if fileobj is an int, ValueError is raised.

Raises **ValueError** – if fileobj is not a *FileMock*, a file-like object or a *FileDescriptor*.

asynctest.**isfilemock**(*obj*)
Return True if the obj or obj.fileno() is a *asynctest.FileDescriptor*.

2.3.2 Mocking the selector

class asynctest.**TestSelector**(*selector=None*)
A selector which supports IOMock objects.

It can wrap an actual implementation of a selector, so the selector will work both with mocks and real file-like objects.

A common use case is to patch the selector loop:

```
loop._selector = asynctest.TestSelector(loop._selector)
```

Parameters **selector** – optional, if provided, this selector will be used to work with real file-like objects.

close()
Close the selector.

Close the actual selector if supplied, unregister all mocks.

See selectors.BaseSelector.close().

modify(*fileobj*, *events*, *data=None*)
Shortcut when calling *TestSelector.unregister()* then *TestSelector.register()* to update the registration of a an object to the selector.

See selectors.BaseSelector.modify().

register(*fileobj*, *events*, *data=None*)
Register a file object or a *FileMock*.

If a real selector object has been supplied to the *TestSelector* object and fileobj is not a *FileMock* or a *FileDescriptor* returned by *FileMock.fileno()*, the object will be registered to the real selector.

See selectors.BaseSelector.register().

select(*timeout=None*)
Perform the selection.

This method is a no-op if no actual selector has been supplied.

See selectors.BaseSelector.select().

unregister(*fileobj*)
Unregister a file object or a *FileMock*.

See selectors.BaseSelector.unregister().

Helpers

asynctest.**set_read_ready**(*fileobj*, *loop*)
Schedule callbacks registered on loop as if the selector notified that data is ready to be read on fileobj.

Parameters

- **fileobj** – file object or *FileMock* on which the event is mocked.
- **loop** – asyncio.SelectorEventLoop watching for events on fileobj.

```
mock = asynctest.SocketMock()
mock.recv.return_value = b"Data"

def read_ready(sock):
    print("received:", sock.recv(1024))

loop.add_reader(mock, read_ready, mock)

set_read_ready(mock, loop)

loop.run_forever() # prints received: b"Data"
```

New in version 0.4.

asynctest.**set_write_ready**(*fileobj*, *loop*)

Schedule callbacks registered on `loop` as if the selector notified that data can be written to `fileobj`.

Parameters

- **fileobj** – file object or *FileMock* on which th event is mocked.
- **loop** – `asyncio.SelectorEventLoop` watching for events on `fileobj`.

New in version 0.4.

2.4 Module helpers

Helper functions and coroutines for *asynctest*.

asynctest.helpers.**exhaust_callbacks**(*loop*)

Run the loop until all ready callbacks are executed.

The coroutine doesn't wait for callbacks scheduled in the future with `call_at()` or `call_later()`.

Parameters **loop** – event loop

CHAPTER 3

Code examples

3.1 List of code examples

3.1.1 `tutorial/clock.py`

```
# coding: utf-8
import time

import asynctest


def is_time_around(expected_time, loop=None, delta=.01):
    """
    Checks that the time is equal to ``expected_time`` (within the range of +/-
    ``delta``).

    If ``loop`` is provided, the clock of the loop is used.
    """
    now = loop.time() if loop else time.time()
    return (expected_time - delta) <= now <= (expected_time + delta)


class TestAdvanceTime(asynctest.ClockedTestCase):
    async def test_advance_time(self):
        base_loop_time = self.loop.time()
        base_wall_time = time.time()

        await self.advance(10)

        self.assertEqual(base_loop_time + 10, self.loop.time())
        self.assertTrue(is_time_around(base_wall_time))
```

(continues on next page)

(continued from previous page)

```
class TestWithClockAndCallbacks(asynctest.ClockedTestCase):
    results = None

    def runs_at(self, expected_time):
        self.results.append(is_time_around(expected_time, self.loop))

    @asynctest.fail_on(active_handles=True)
    async def test_callbacks_executed_when_expected(self):
        self.results = []

        base_time = self.loop.time()
        self.loop.call_later(1, self.runs_at, base_time + 1)
        self.loop.call_at(base_time + 7, self.runs_at, base_time + 7)

        # This shows that the callback didn't run yet
        self.assertEqual(0, len(self.results))

        await self.advance(10)

        # This shows that the callbacks ran...
        self.assertEqual(2, len(self.results))
        # ...when expected
        self.assertTrue(all(self.results))
```

3.1.2 tutorial/mocking.py

```
# coding: utf-8
# pylama: ignore=C0103, ignore camel case variable name (AsyncClientMock)

import asyncio
import collections
import itertools
import logging

import asynctest


class Client:
    def add_user(self, user):
        raise NotImplementedError

    def get_users(self):
        raise NotImplementedError

    def increase_nb_users_cached(self, nb_cached):
        raise NotImplementedError


class AsyncClient:
    async def add_user(self, user, transaction=None):
        raise NotImplementedError

    async def get_users(self, transaction=None):
        raise NotImplementedError
```

(continues on next page)

(continued from previous page)

```
    async def increase_nb_users_cached(self, nb_cached, transaction=None):
        raise NotImplementedError

    def get_users_cursor(self, transaction=None):
        return self.Cursor(transaction or self)

    def new_transaction(self):
        return self.Transaction(self)

    class Transaction:
        def __init__(self, client):
            self.client = client

        def __call__(self, funcname, *args, **kwargs):
            """
            Forwards the call to the client, with the argument ``transaction ``
            set.
            """
            method = getattr(self.client, funcname)
            return method(*args, transaction=self, **kwargs)

        async def __aenter__(self):
            return self

        async def __aexit__(self, *args):
            pass

    class Cursor:
        def __init__(self, transaction):
            self.transaction = transaction

        def __aiter__(self):
            return self

        async def __anext__(self):
            # if the request has not been started, do it there
            raise NotImplementedError


def cache_users(client, cache):
    """
    Load the list of users from a distant server accessed with ``client``,
    add them to ``cache``.

    Notify the server about the number of new users put in the cache, and
    returns this number.

    :param client: a connection to the distant server
    :param cache: a dict-like object
    """
    users = client.get_users()

    nb_users_cached = 0

    for user in users:
        if user.id not in cache:
            nb_users_cached += 1
```

(continues on next page)

(continued from previous page)

```
            cache[user.id] = user

    client.increase_nb_users_cached(nb_users_cached)

    logging.debug("added %d users to the cache %r", nb_users_cached, cache)

    return nb_users_cached


class StubClient:
    User = collections.namedtuple("User", "id username")

    def __init__(self, *users_to_return):
        self.users_to_return = []
        self.users_to_return.extend(users_to_return)

        self.nb_users_cached = 0

    def add_user(self, user):
        self.users_to_return.append(user)

    def get_users(self):
        return self.users_to_return

    def increase_nb_users_cached(self, nb_cached):
        self.nb_users_cached += nb_cached


class TestUsingStub(asynctest.TestCase):
    def test_one_user_added_to_cache(self):
        user = StubClient.User(1, "a.dmin")
        client = StubClient(user)
        cache = {}

        # The user has been added to the cache
        nb_added = cache_users(client, cache)

        self.assertEqual(nb_added, 1)
        self.assertEqual(cache[1], user)

        # The user was already there
        nb_added = cache_users(client, cache)
        self.assertEqual(nb_added, 0)
        self.assertEqual(cache[1], user)

    def test_no_users_to_add(self):
        cache = {}
        nb_added = cache_users(StubClient(), cache)

        self.assertEqual(nb_added, 0)
        self.assertEqual(len(cache), 0)


class TestUsingMock(asynctest.TestCase):
    def test_no_users_to_add(self):
        client = asynctest.Mock(Client())
        client.get_users.return_value = []
```

(continues on next page)

(continued from previous page)

```
        cache = {}

        nb_added = cache_users(client, cache)

        client.get_users.assert_called()
        self.assertEqual(nb_added, 0)
        self.assertEqual(len(cache), 0)

        client.increase_nb_users_cached.assert_called_once_with(0)


async def cache_users_async(client, cache):
    users = await client.get_users()

    nb_users_cached = 0

    for user in users:
        if user.id not in cache:
            nb_users_cached += 1
            cache[user.id] = user

    await client.increase_nb_users_cached(nb_users_cached)

    logging.debug("added %d users to the cache %r", nb_users_cached, cache)

    return nb_users_cached


class TestUsingFuture(asynctest.TestCase):
    async def test_no_users_to_add(self):
        client = asynctest.Mock(Client())

        client.get_users.return_value = asyncio.Future()
        client.get_users.return_value.set_result([])

        client.increase_nb_users_cached.return_value = asyncio.Future()
        client.increase_nb_users_cached.return_value.set_result(None)

        cache = {}

        nb_added = await cache_users_async(client, cache)

        client.get_users.assert_called()
        self.assertEqual(nb_added, 0)
        self.assertEqual(len(cache), 0)

        client.increase_nb_users_cached.assert_called_once_with(0)


class TestUsingCoroutineMock(asynctest.TestCase):
    async def test_no_users_to_add(self):
        client = asynctest.Mock(Client())
        client.get_users = asynctest.CoroutineMock(return_value=[])
        client.increase_nb_users_cached = asynctest.CoroutineMock()
        cache = {}

        nb_added = await cache_users_async(client, cache)
```

(continues on next page)

(continued from previous page)

```
        client.get_users.assert_awaited()
        self.assertEqual(nb_added, 0)
        self.assertEqual(len(cache), 0)

        client.increase_nb_users_cached.assert_awaited_once_with(0)


class TestUsingCoroutineMockAndSpec(asynctest.TestCase):
    async def test_no_users_to_add(self):
        client = asynctest.Mock(AsyncClient())
        client.get_users.return_value = []
        cache = {}

        nb_added = await cache_users_async(client, cache)

        client.get_users.assert_awaited()
        self.assertEqual(nb_added, 0)
        self.assertEqual(len(cache), 0)

        client.increase_nb_users_cached.assert_awaited_once_with(0)


class TestAutoSpec(asynctest.TestCase):
    async def test_functions_and_coroutines_arguments_are_checked(self):
        client = asynctest.Mock(Client())
        cache = {}

        cache_users_mock = asynctest.create_autospec(cache_users_async)

        with self.subTest("create_autospec returns a regular mock"):
            await cache_users_mock(client, cache)
            cache_users_mock.assert_awaited_once_with(client, cache)

        with self.subTest("an exception is raised when the mock is called "
                          "with the wrong number of arguments"):
            with self.assertRaises(TypeError):
                await cache_users_mock("wrong", "number", "of", "args")

    async def test_create_autospec_on_a_class(self):
        AsyncClientMock = asynctest.create_autospec(AsyncClient)
        client = AsyncClientMock()

        with self.subTest("the mock of a class returns a mock instance of "
                          "the class"):
            self.assertIsInstance(client, AsyncClient)

        with self.subTest("attributes of the mock instance are correctly "
                          "mocked as coroutines"):
            await client.increase_nb_users_cached(1)


class TestCoroutineMockResult(asynctest.TestCase):
    async def test_result_set_with_return_value(self):
        coroutine_mock = asynctest.CoroutineMock()
        result = object()
        coroutine_mock.return_value = result
```

(continues on next page)

(continued from previous page)

```
        # return the expected result
        self.assertIs(result, await coroutine_mock())
        # always return the same result
        self.assertIs(await coroutine_mock(), await coroutine_mock())

    async def test_result_with_side_effect_function(self):
        def uppercase_all(*args):
            return tuple(arg.upper() for arg in args)

        coroutine_mock = asynctest.CoroutineMock()
        coroutine_mock.side_effect = uppercase_all

        self.assertEqual(("FIRST", "CALL"),
                         await coroutine_mock("first", "call"))
        self.assertEqual(("A", "SECOND", "CALL"),
                         await coroutine_mock("a", "second", "call"))

    async def test_result_with_side_effect_exception(self):
        coroutine_mock = asynctest.CoroutineMock()
        coroutine_mock.side_effect = NotImplementedError

        # Raise an exception of the configured type
        with self.assertRaises(NotImplementedError):
            await coroutine_mock("any", "number", "of", "args")

        coroutine_mock.side_effect = Exception("an instance of exception")

        # Raise the exact specified object
        with self.assertRaises(Exception) as context:
            await coroutine_mock()

        self.assertIs(coroutine_mock.side_effect, context.exception)

    async def test_result_with_side_effect_iterable(self):
        coroutine_mock = asynctest.CoroutineMock()
        coroutine_mock.side_effect = ["one", "two", "three"]

        self.assertEqual("one", await coroutine_mock())
        self.assertEqual("two", await coroutine_mock())
        self.assertEqual("three", await coroutine_mock())

        coroutine_mock.side_effect = itertools.cycle(["odd", "even"])
        self.assertEqual("odd", await coroutine_mock())
        self.assertEqual("even", await coroutine_mock())
        self.assertEqual("odd", await coroutine_mock())
        self.assertEqual("even", await coroutine_mock())

    async def test_result_with_wrapped_object(self):
        stub = StubClient()
        mock = asynctest.Mock(stub, wraps=stub)
        cache = {}

        stub.add_user(StubClient.User(1, "a.dmin"))
        cache_users(mock, cache)

        mock.get_users.assert_called()
```

(continues on next page)

(continued from previous page)

```
        self.assertEqual(stub.users_to_return, mock.get_users())


async def cache_users_with_cursor(client, cache):
    nb_users_cached = 0

    async with client.new_transaction() as transaction:
        users_cursor = transaction.get_users_cursor()

        async for user in users_cursor:
            if user.id not in cache:
                nb_users_cached += 1
                cache[user.id] = user

        await transaction.increase_nb_users_cached(nb_users_cached)

    logging.debug("added %d users to the cache %r", nb_users_cached, cache)

    return nb_users_cached


class TestWithMagicMethods(asynctest.TestCase):
    async def test_context_manager(self):
        with self.assertRaises(AssertionError):
            async with asynctest.MagicMock() as context:
                # context is a MagicMock
                context.assert_called()

    async def test_empty_iterable(self):
        loop_iterations = 0
        async for _ in asynctest.MagicMock():
            loop_iterations += 1

        self.assertEqual(0, loop_iterations)

    async def test_iterable(self):
        loop_iterations = 0
        mock = asynctest.MagicMock()
        mock.__aiter__.return_value = range(5)
        async for _ in mock:
            loop_iterations += 1

        self.assertEqual(5, loop_iterations)


class TestCacheWithMagicMethods(asynctest.TestCase):
    async def test_one_user_added_to_cache(self):
        user = StubClient.User(1, "a.dmin")

        AsyncClientMock = asynctest.create_autospec(AsyncClient)

        transaction = asynctest.MagicMock()
        transaction.__aenter__.side_effect = AsyncClientMock

        cursor = asynctest.MagicMock()
        cursor.__aiter__.return_value = [user]
```

(continues on next page)

(continued from previous page)

```
        client = AsyncClientMock()
        client.new_transaction.return_value = transaction
        client.get_users_cursor.return_value = cursor

        cache = {}

        # The user has been added to the cache
        nb_added = await cache_users_with_cursor(client, cache)

        self.assertEqual(nb_added, 1)
        self.assertEqual(cache[1], user)

        # The user was already there
        nb_added = await cache_users_with_cursor(client, cache)
        self.assertEqual(nb_added, 0)
        self.assertEqual(cache[1], user)


class TestCachingIsLogged(asynctest.TestCase):
    async def test_with_context_manager(self):
        client = asynctest.Mock(AsyncClient())
        cache = {}

        with asynctest.patch("logging.debug") as debug_mock:
            await cache_users_async(client, cache)

        debug_mock.assert_called()

    @asynctest.patch("logging.error")
    @asynctest.patch("logging.debug")
    async def test_with_decorator(self, debug_mock, error_mock):
        client = asynctest.Mock(AsyncClient())
        cache = {}

        await cache_users_async(client, cache)

        debug_mock.assert_called()
        error_mock.assert_not_called()
```

3.1.3 tutorial/mocking_io.py

```
# coding: utf-8
import asyncio
import socket

import asynctest


class TestMockASocket(asynctest.TestCase):
    async def test_read_and_write_from_socket(self):
        socket_mock = asynctest.SocketMock()
        socket_mock.type = socket.SOCK_STREAM

        recv_data = iter((
            b"some data read",
```

(continues on next page)

(continued from previous page)

```
            b"some other",
            b" ...and the last",
        ))

        recv_buffer = bytearray()

        def recv_side_effect(max_bytes):
            nonlocal recv_buffer

            if not recv_buffer:
                try:
                    recv_buffer.extend(next(recv_data))
                    asynctest.set_read_ready(socket_mock, self.loop)
                except StopIteration:
                    # nothing left
                    pass

            data = recv_buffer[:max_bytes]
            recv_buffer = recv_buffer[max_bytes:]

            if recv_buffer:
                # Some more data to read
                asynctest.set_read_ready(socket_mock, self.loop)

            return data

        def send_side_effect(data):
            asynctest.set_read_ready(socket_mock, self.loop)
            return len(data)

        socket_mock.recv.side_effect = recv_side_effect
        socket_mock.send.side_effect = send_side_effect

        reader, writer = await asyncio.open_connection(sock=socket_mock)

        writer.write(b"a request?")
        self.assertEqual(b"some", await reader.read(4))
        self.assertEqual(b" data read", await reader.read(10))
        self.assertEqual(b"some other ...and the last", await reader.read())
```

3.1.4 tutorial/patching.py

```
# coding: utf-8
import asyncio
import unittest.mock

import asynctest


class MustBePatched:
    async def is_patched(self):
        """
        return ``False``, unless patched.
        """
        return False
```

(continues on next page)

(continued from previous page)

```
    async def crash_if_patched(self, ran_event):
        """
        Verify that the method is not patched. The coroutine is put to sleep
        for a duration of 0, meaning it let the loop schedule other coroutines
        concurrently.

        Each time the check is performed, ``ran_event`` is set.
        """

        try:
            while True:
                try:
                    is_patched = await self.is_patched()
                    assert not is_patched

                    await asyncio.sleep(0)
                finally:
                    ran_event.set()
        except asyncio.CancelledError:
            pass


async def terminate_and_check_task(task):
    task.cancel()
    await task


async def happened_once(event):
    await event.wait()
    event.clear()


must_be_patched = MustBePatched()  # noqa


class TestMustBePatched(asynctest.TestCase):
    async def setUp(self):
        # Event used to track if the background task checked if the patch
        # is active
        self.checked = asyncio.Event()

        # This task checks if the object is patched continuously, and sets
        # the checked event everytime it does so.
        self.background_task = asyncio.create_task(
            must_be_patched.crash_if_patched(self.checked))

        # Any test will fail if the background task raises an exception
        self.addCleanup(terminate_and_check_task, self.background_task)

    @asynctest.patch.object(must_be_patched, "is_patched",
                            return_value=True)
    async def test_patching_conflicting(self, _):
        # This call blocks until the check happened once in background
        await happened_once(self.checked)
        self.assertTrue(await must_be_patched.is_patched())
        await happened_once(self.checked)
```

(continues on next page)

(continued from previous page)

```
    @asynctest.patch.object(must_be_patched, "is_patched",
                            return_value=True, scope=asynctest.LIMITED)
    async def test_patching_not_conflicting(self, _):
        await happened_once(self.checked)
        self.assertTrue(await must_be_patched.is_patched())
        await happened_once(self.checked)
```

3.1.5 tutorial/test_cases.py

```
# coding: utf-8
import asyncio
import asynctest


class MinimalExample(asynctest.TestCase):
    def test_that_true_is_true(self):
        self.assertTrue(True)


class AnExampleWithSetup(asynctest.TestCase):
    async def a_coroutine(self):
        return "I worked"

    def test_that_a_coroutine_runs(self):
        my_loop = asyncio.new_event_loop()
        try:
            result = my_loop.run_until_complete(self.a_coroutine())
            self.assertIn("worked", result)
        finally:
            my_loop.close()


class AnExampleWithSetupMethod(asynctest.TestCase):
    async def a_coroutine(self):
        return "I worked"

    def setUp(self):
        self.my_loop = asyncio.new_event_loop()

    def test_that_a_coroutine_runs(self):
        result = self.my_loop.run_until_complete(self.a_coroutine())
        self.assertIn("worked", result)

    def tearDown(self):
        self.my_loop.close()


class AnExampleWithSetupAndCleanup(asynctest.TestCase):
    async def a_coroutine(self):
        return "I worked"

    def setUp(self):
        self.my_loop = asyncio.new_event_loop()
        self.addCleanup(self.my_loop.close)
```

(continues on next page)

(continued from previous page)

```
    def test_that_a_coroutine_runs(self):
        result = self.my_loop.run_until_complete(self.a_coroutine())
        self.assertIn("worked", result)


class AnExampleWithTestCaseLoop(asynctest.TestCase):
    async def a_coroutine(self):
        return "I worked"

    def test_that_a_coroutine_runs(self):
        result = self.loop.run_until_complete(self.a_coroutine())
        self.assertIn("worked", result)


class AnExampleWithTestCaseAndCoroutines(asynctest.TestCase):
    async def a_coroutine(self):
        return "I worked"

    async def test_that_a_coroutine_runs(self):
        self.assertIn("worked", await self.a_coroutine())


class AnExampleWithAsynchronousSetUp(asynctest.TestCase):
    async def setUp(self):
        self.queue = asyncio.Queue(maxsize=1)
        await self.queue.put("I worked")

    async def test_that_a_lock_is_acquired(self):
        self.assertTrue(self.queue.full())

    async def tearDown(self):
        while not self.queue.empty():
            await self.queue.get()


class AnExempleWhichDetectsPendingCallbacks(asynctest.TestCase):
    def i_must_run(self):
        pass  # do something

    @asynctest.fail_on(active_handles=True)
    async def test_missing_a_callback(self):
        self.loop.call_later(1, self.i_must_run)
```

CHAPTER 4

Contribute

Development of `asynctest` is on github: Martiusweb/asynctest. Patches, feature requests and bug reports are welcome.

CHAPTER 5

Documentation indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index