asynctest Documentation

Release 0.12.2

Martin Richard

Aug 28, 2018

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

Reference

1.1 Module asynctest.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.

1.2 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.

1.2.1 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.

1.2.2 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.case.`**`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.case.`**`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.

1.3 Mock objects

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`.

1.3.1 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()`

1.3.2 Patch

asynctest.**GLOBAL = <PatchScope.GLOBAL: 2>**
Value of `scope`, activating a patch until the decorated generator or coroutine returns or raises an exception.

asynctest.**LIMITED = <PatchScope.LIMITED: 1>**
Value of `scope`, deactivating a patch when a decorated generator or a coroutine pauses (`yield` or `await`).

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():
```

(continues on next page)

(continued from previous page)

```
    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()`.

1.3.3 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.

1.4 Mocking of 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.

1.4.1 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*.

1.4.2 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()

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.

1.5 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 2

Contribute

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

CHAPTER 3

Documentation indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

P

R

S

T

U