How to set class attribute with await in __init__

Asked 5 years, 7 months ago   Active 1 month ago   Viewed 48k times

115

44

How can I define a class with `await` in the constructor or class body?

For example what I want:

```python
import asyncio

# some code


class Foo(object):

    async def __init__(self, settings):
        self.settings = settings
        self.pool = await create_pool(dsn)

foo = Foo(settings)
# it raises:
# TypeError: __init__() should return None, not 'coroutine'
```

or example with class body attribute:

```python
class Foo(object):

    self.pool = await create_pool(dsn)  # Sure it raises syntax Error

    def __init__(self, settings):
        self.settings = settings

foo = Foo(settings)
```

My solution (But I would like to see a more elegant way)

```python
class Foo(object):

    def __init__(self, settings):
        self.settings = settings

    async def init(self):
        self.pool = await create_pool(dsn)

foo = Foo(settings)
await foo.init()
```

python   python-3.x   python-asyncio

78.4k 12 190 203   2,247 3 21 45

1 You might have some luck with `__new__`, although it might not be elegant – JBernardo Oct 14 '15 at 14:55

I don't have experience with 3.5, and in other languages this wouldn't work because of the viral nature of async/await, but have you tried defining an async function like `_pool_init(dsn)` and then calling it from `__init__`? It would preserve the init-in-constructor appearance. – a p Oct 14 '15 at 19:34

1 If you use curio: curio.readthedocs.io/en/latest/... – matsjoyce Jun 26 '17 at 16:04

3 use `@classmethod` 😎 it's an alternate constructor. put the async work there; then in `__init__`, just set the `self` attributes – grisaitis Jun 23 '20 at 20:24

7 Answers

Active Oldest Votes

138

Most magic methods aren't designed to work with `async def` / `await` - in general, you should only be using `await` inside the dedicated asynchronous magic methods - `__aiter__`, `__anext__`, `__aenter__`, and `__aexit__`. Using it inside other magic methods either won't work at all, as is the case with `__init__` (unless you use some tricks described in other answers here), or will force you to always use whatever triggers the magic method call in an asynchronous context.

Existing `asyncio` libraries tend to deal with this in one of two ways: First, I've seen the factory pattern used (`asyncio-redis`, for example):

```
import asyncio

dsn = "..."

class Foo(object):
    @classmethod
    async def create(cls, settings):
        self = Foo()
        self.settings = settings
        self.pool = await create_pool(dsn)
        return self

async def main(settings):
    settings = "..."
    foo = await Foo.create(settings)
```

Other libraries use a top-level coroutine function that creates the object, rather than a factory method:

```
import asyncio

dsn = "..."
```

```
        return foo

class Foo(object):
    def __init__(self, settings):
        self.settings = settings

    async def _init(self):
        self.pool = await create_pool(dsn)

async def main():
    settings = "..."
    foo = await create_foo(settings)
```

The `create_pool` function from `aiopg` that you want to call in `__init__` is actually using this exact pattern.

This at least addresses the `__init__` issue. I haven't seen class variables that make asynchronous calls in the wild that I can recall, so I don't know that any well-established patterns have emerged.

Share Improve this answer Follow   edited Jul 29 '20 at 20:59   answered Oct 14 '15 at 19:42
dano
78.4k 12 190 203

46

Another way to do this, for funsies:

```
class aobject(object):
    """Inheriting this class allows you to define an async __init__.

    So you can create objects by doing something like `await MyClass(params)`
    """
    async def __new__(cls, *a, **kw):
        instance = super().__new__(cls)
        await instance.__init__(*a, **kw)
        return instance

    async def __init__(self):
        pass

#With non async super classes

class A:
    def __init__(self):
        self.a = 1

class B(A):
    def __init__(self):
        self.b = 2
        super().__init__()

class C(B, aobject):
    async def __init__(self):
        super().__init__()
        self.c=3
    async def __init__(self, a):
        self.a = a

class E(D):
    async def __init__(self):
        self.b = 2
        await super().__init__(1)

# Overriding __new__

class F(aobject):
    async def __new__(cls):
        print(cls)
        return await super().__new__(cls)

    async def __init__(self):
        await asyncio.sleep(1)
        self.f = 6

async def main():
    e = await E()
    print(e.b) # 2
    print(e.a) # 1

    c = await C()
    print(c.a) # 1
    print(c.b) # 2
    print(c.c) # 3

    f = await F() # Prints F class
    print(f.f) # 6

import asyncio
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

Share Improve this answer Follow

edited Mar 11 '20 at 20:46

answered Jul 28 '17 at 4:02

khazhyk

1,080 15 13

3 This is currently the most clear and understandable implementation in my opinion. I really like how intuitively extensible it is. I was worried it would be necessary to delve into metaclasses. – Tankobot Feb 22 '18 at 19:05

2 This doesn't have correct `__init__` semantics if `super().__new__(cls)` returns a pre-existing instance - normally, this would skip `__init__`, but your code not. – Eric Dec 16 '18 at 6:51

Hmm, per `object.__new__` documentation, `__init__` should only be invoked if `isinstance(instance, cls)` ? This seems somewhat unclear to me... But I don't see the semantics you claim anywhere... – khazhyk Dec 17 '18 at 22:50

Thinking about this more, if you override `__new__` to return a pre-existing object, that new would need to be the outermost to make any sense, since other implementations of `__new__` would have no general way of knowing if you're returning a new uninitialized instance or not. – khazhyk Jul 16 '19 at 0:20

1 @khazhyk Well, there IS definitely something preventing you from defining `async def __init__(...)`, as showed by the OP, and I believe that `TypeError: __init__() should return None, not 'coroutine'`

invoked". Your new `__new__()` returns a coroutine, not a cls. That's why. Clever hack! – RayLuo Mar 14 '20 at 21:56

21

I would recommend a separate factory method. It's safe and straightforward. However, if you insist on a `async` version of `__init__()`, here's an example:

```
def asyncinit(cls):
    __new__ = cls.__new__

    async def init(obj, *arg, **kwarg):
        await obj.__init__(*arg, **kwarg)
        return obj

    def new(cls, *arg, **kwarg):
        obj = __new__(cls, *arg, **kwarg)
        coro = init(obj, *arg, **kwarg)
        #coro.__init__ = lambda *_1, **_2: None
        return coro

    cls.__new__ = new
    return cls
```

Usage:

```
@asyncinit
class Foo(object):
    def __new__(cls):
        '''Do nothing. Just for test purpose.'''
        print(cls)
        return super().__new__(cls)

    async def __init__(self):
        self.initialized = True


async def f():
    print((await Foo()).initialized)

loop = asyncio.get_event_loop()
loop.run_until_complete(f())
```

Output:

```
<class '__main__.Foo'>
True
```

Explanation:

Your class construction must return a `coroutine` object instead of its own instance.

11.7k 6 53 68 1,240 9 19

Couldn't you name your `new` `__new__` and use `super` (likewise for `__init__`, i.e. just let the client override that) instead? – Matthias Urlichs Oct 9 '18 at 2:11

13

Better yet you can do something like this, which is very easy:

```
import asyncio

class Foo:
    def __init__(self, settings):
        self.settings = settings

    async def async_init(self):
        await create_pool(dsn)

    def __await__(self):
        return self.async_init().__await__()

loop = asyncio.get_event_loop()
foo = loop.run_until_complete(Foo(settings))
```

Basically what happens here is `__init__()` gets called first as usual. Then `__await__()` gets called which then awaits `async_init()`.

Share Improve this answer Follow

edited Mar 20 '20 at 12:42
Spike Lee
35 2 8

answered Nov 21 '19 at 14:03
Vishnu shettigar
131 1 5

4

[Almost] canonical answer by @ojii

```
@dataclass
class Foo:
    settings: Settings
    pool: Pool

    @classmethod
    async def create(cls, settings: Settings, dsn):
        return cls(settings, await create_pool(dsn))
```

Share Improve this answer Follow

answered Apr 7 '20 at 7:26
Dima Tisnek
9,337 4 47 105

3 `dataclasses` for the win! so easy. – grisaitis Jun 23 '20 at 20:27

1

method.

```
import asyncio


class Foo(object):

    def __init__(self, settings):
        self.settings = settings
        loop = asyncio.get_event_loop()
        self.pool = loop.run_until_complete(create_pool(dsn))

foo = Foo(settings)
```

Important point to be noted is:

- This makes the async code work as sync(blocking)
- This is not the best way to run async code, but when it comes to only initiation via a sync method eg: `__init__` it will be a good fit.
- After initiation, you can run the async methods from the object with await. i.e `await foo.pool.get(value)`
- Do not try to initiate via an `await` call you will get `RuntimeError: This event loop is already running`

Share Improve this answer Follow

edited Apr 8 at 6:03

answered Mar 16 at 11:29

Ja8zyjits

1,203 14 26

0

Depending on your needs, you can also use `AwaitLoader` from: https://pypi.org/project/async-property/

From the docs:

> `AwaitLoader` will call await `instance.load()` , if it exists, before loading properties.

Share Improve this answer Follow

answered Jan 5 at 23:00

tehfink

367 5 7

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up