# Asyncio Extras

*Release 1.3.0*

**Sep 27, 2017**

# Contents

This library provides some "missing" features for the asyncio ([PEP 3156](#)) module:

- decorator for making asynchronous context managers (like `contextmanager()`)
- decorator and context manager for running a function or parts of a function in a thread pool
- helpers for calling functions in the event loop from worker threads and vice versa
- helpers for doing non-blocking file i/o

versionhistory

asyncio_extras.contextmanager

asyncio_extras.contextmanager.**async_contextmanager**(*func*)

Transform a coroutine function into something that works with `async with`.

This is an asynchronous counterpart to `contextmanager()`. The wrapped function can either be a native async generator function (`async def` with `yield`) or, if your code needs to be compatible with Python 3.5, you can use `yield_()` instead of the native `yield` statement.

The generator must yield *exactly once*, just like with `contextmanager()`.

Usage in Python 3.5 and earlier:

```python
@async_contextmanager
async def mycontextmanager(arg):
    context = await setup_remote_context(arg)
    await yield_(context)
    await context.teardown()


async def frobnicate(arg):
    async with mycontextmanager(arg) as context:
        do_something_with(context)
```

The same context manager function in Python 3.6+:

```python
@async_contextmanager
async def mycontextmanager(arg):
    context = await setup_remote_context(arg)
    yield context
    await context.teardown()
```

> **Parameters func** (`Callable[..., Coroutine]`) – an async generator function or a coroutine function using `yield_()`
>
> **Return type** `Callable`
>
> **Returns** a callable that can be used with `async with`

asyncio_extras.file

**class** `asyncio_extras.file.`**AsyncFileWrapper**(*path: str, args: tuple, kwargs: dict, executor: typing.Union[concurrent.futures._base.Executor, NoneType]*) → None

Wraps certain file I/O operations so they're guaranteed to run in a thread pool.

The wrapped methods work like coroutines when called in the event loop thread, but when called in any other thread, they work just like the methods of the `file` type.

This class supports use as an asynchronous context manager.

The wrapped methods are:

- `flush()`

- `read()`

- `readline()`

- `readlines()`

- `seek()`

- `truncate()`

- `write()`

- `writelines()`

**async_readchunks**(*size*)

Read data from the file in chunks.

> **Parameters** `size` (`int`) – the maximum number of bytes or characters to read at once
>
> **Returns** an asynchronous iterator yielding bytes or strings

`asyncio_extras.file.`**open_async**(*file*, *\*args*, *executor=None*, *\*\*kwargs*)

Open a file and wrap it in an *AsyncFileWrapper*.

Example:

```python
async def read_file_contents(path: str) -> bytes:
    async with open_async(path, 'rb') as f:
        return await f.read()
```

The file wrapper can also be asynchronously iterated line by line:

```python
async def read_file_lines(path: str):
    async for line in open_async(path):
        print(line)
```

> **Parameters**
>
> - **file** (Union[str, Path]) – the file path to open
>
> - **args** – positional arguments to open()
>
> - **executor** (Optional[Executor]) – the executor argument to *AsyncFileWrapper*
>
> - **kwargs** – keyword arguments to open()
>
> **Return type** *AsyncFileWrapper*
>
> **Returns** the wrapped file object

asyncio_extras.threads

asyncio_extras.threads.**threadpool**(*arg=None*)

> Return a decorator/asynchronous context manager that guarantees that the wrapped function or `with` block is run in the given executor.

> If no executor is given, the current event loop's default executor is used. Otherwise, the executor must be a PEP 3148 compliant thread pool executor.

> Callables wrapped with this must be used with `await` when called in the event loop thread. They can also be called in worker threads, just by omitting the `await`.

> Example use as a decorator:

```python
@threadpool
def this_runs_in_threadpool():
    return do_something_cpu_intensive()


async def request_handler():
    result = await this_runs_in_threadpool()
```

> Example use as an asynchronous context manager:

```python
async def request_handler(in_url, out_url):
    page = await http_fetch(in_url)

    async with threadpool():
        data = transform_page(page)

    await http_post(out_url, page)
```

> > **Parameters arg** (`Union`[Executor, `Callable`, None]) – either a callable (when used as a decorator) or an executor in which to run the wrapped callable or the `with` block (when used as a context manager)

asyncio_extras.threads.**call_in_executor**(*func*, *\*args*, *executor=None*, *\*\*kwargs*)

> Call the given callable in an executor.

This is a nicer version of the following:

```
get_event_loop().run_in_executor(executor, func, *args)
```

If you need to pass keyword arguments named `func` or `executor` to the callable, use `functools.partial()` for that.

> **Parameters**
>
> - **func** (`Callable`) – a function
> - **args** – positional arguments to call with
> - **executor** (`Optional`[Executor]) – the executor to call the function in
> - **kwargs** – keyword arguments to call with
>
> **Return type** `Future`
>
> **Returns** a future that will resolve to the function call's return value

`asyncio_extras.threads.`**`call_async`**(*loop*, *func*, *\*args*, *\*\*kwargs*)

> Call the given callable in the event loop thread.
>
> If the call returns an awaitable, it is resolved before returning to the caller.
>
> If you need to pass keyword arguments named `loop` or `func` to the callable, use `functools.partial()` for that.
>
> > **Parameters**
> >
> > - **func** (`Callable`) – a regular function or a coroutine function
> > - **args** – positional arguments to call with
> > - **loop** (`AbstractEventLoop`) – the event loop in which to call the function
> > - **kwargs** – keyword arguments to call with
> >
> > **Returns** the return value of the function call

# Python Module Index

## a

# Index

## A

async_contextmanager() (in module asyncio_extras.contextmanager), 3
async_readchunks() (asyncio_extras.file.AsyncFileWrapper method), 5
AsyncFileWrapper (class in asyncio_extras.file), 5
asyncio_extras.contextmanager (module), 3
asyncio_extras.file (module), 5
asyncio_extras.threads (module), 7

## C

call_async() (in module asyncio_extras.threads), 8
call_in_executor() (in module asyncio_extras.threads), 7

## O

open_async() (in module asyncio_extras.file), 5

## P

Python Enhancement Proposals
    PEP 3156, 1

## T

threadpool() (in module asyncio_extras.threads), 7