
threaded Documentation

Release 4.1.0

Alexey Stepanov

Nov 17, 2020

Contents

1	Usage	3
1.1	ThreadPooled	3
1.2	Threaded	4
1.3	AsyncIOTask	5
2	Testing	7
3	CI systems	9
4	CD system	11
4.1	API: Decorators: <i>ThreadPooled</i> , <i>threadpooled</i>	11
4.2	API: Decorators: <i>Threaded</i> class and <i>threaded</i> function.	12
4.3	API: Decorators: <i>AsyncIOTask</i> , <i>asynciotask</i>	13
5	Indices and tables	15
	Python Module Index	17
	Index	19

threaded is a set of decorators, which wrap functions in:

- `concurrent.futures.ThreadPool`
- `threading.Thread`
- `asyncio.Task` in Python 3.

Why? Because copy-paste of `loop.create_task`, `threading.Thread` and `thread_pool.submit` is boring, especially if target functions is used by this way only.

Pros:

- Free software: Apache license
- Open Source: <https://github.com/python-useful-helpers/threaded>
- PyPI packaged: <https://pypi.python.org/pypi/threaded>
- Tested: see bages on top
- Support multiple Python versions:

```
Python 3.4
Python 3.5
Python 3.6
Python 3.7
PyPy3 3.5+
```

Note: For python 2.7/PyPy you can use versions 1.x.x

Decorators:

- *ThreadPooled* - native `concurrent.futures.ThreadPool`.
- *threadpooled* is alias for *ThreadPooled*.
- *Threaded* - wrap in `threading.Thread`.
- *threaded* is alias for *Threaded*.
- *AsyncIOTask* - wrap in `asyncio.Task`. Uses the same API, as *ThreadPooled*.
- *asynctask* is alias for *AsyncIOTask*.

1.1 ThreadPooled

Mostly it is required decorator: submit function to `ThreadPoolExecutor` on call.

Note: API quite differs between Python 3 and Python 2.7. See API section below.

```
threaded.ThreadPooled.configure(max_workers=3)
```

Note: By default, if executor is not configured - it configures with default parameters: `max_workers=CPU_COUNT`
* 5

```
@threaded.ThreadPooled
def func():
    pass

concurrent.futures.wait([func()])
```

Python 3.5+ usage with `asyncio`:

Note: if `loop_getter` is not callable, `loop_getter_need_context` is ignored.

```
loop = asyncio.get_event_loop()
@threaded.ThreadPooled(loop_getter=loop, loop_getter_need_context=False)
def func():
    pass

loop.run_until_complete(asyncio.wait_for(func(), timeout))
```

Python 3.5+ usage with asyncio and loop extraction from call arguments:

```
loop_getter = lambda tgt_loop: tgt_loop
@threaded.ThreadPooled(loop_getter=loop_getter, loop_getter_need_context=True) #_
↳ loop_getter_need_context is required
def func(*args, **kwargs):
    pass

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait_for(func(loop), timeout))
```

During application shutdown, pool can be stopped (while it will be recreated automatically, if some component will request).

```
threaded.ThreadPooled.shutdown()
```

1.2 Threaded

Classic threading.Thread. Useful for running until close and self-closing threads without return.

Usage example:

```
@threaded.Threaded
def func(*args, **kwargs):
    pass

thread = func()
thread.start()
thread.join()
```

Without arguments, thread name will use pattern: 'Threaded: ' + func.__name__

Note: If func.__name__ is not accessible, str(hash(func)) will be used instead.

Override name can be don via corresponding argument:

```
@threaded.Threaded(name='Function in thread')
def func(*args, **kwargs):
    pass
```

Thread can be daemonized automatically:

```
@threaded.Threaded(daemon=True)
def func(*args, **kwargs):
    pass
```

Also, if no any addition manipulations expected before thread start, it can be started automatically before return:

```
@threaded.Threaded(started=True)
def func(*args, **kwargs):
    pass
```


1.3 AsyncIOTask

Wrap in `asyncio.Task`.

usage with `asyncio`:

```
@threaded.AsyncIOTask
def func():
    pass

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait_for(func(), timeout))
```

Provide event loop directly:

Note: if `loop_getter` is not callable, `loop_getter_need_context` is ignored.

```
loop = asyncio.get_event_loop()
@threaded.AsyncIOTask(loop_getter=loop)
def func():
    pass

loop.run_until_complete(asyncio.wait_for(func(), timeout))
```

Usage with loop extraction from call arguments:

```
loop_getter = lambda tgt_loop: tgt_loop
@threaded.AsyncIOTask(loop_getter=loop_getter, loop_getter_need_context=True)
def func(*args, **kwargs):
    pass

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait_for(func(loop), timeout))
```


CHAPTER 2

Testing

The main test mechanism for the package *threaded* is using *tox*. Available environments can be collected via *tox -l*

CHAPTER 3

CI systems

For code checking several CI systems is used in parallel:

1. [Travis CI](#): is used for checking: PEP8, pylint, bandit, installation possibility and unit tests. Also it's publishes coverage on coveralls.
2. [GitHub actions](#): is used for checking: PEP8, pylint, bandit, installation possibility and unit tests.
3. [coveralls](#): is used for coverage display.

Travis CI: is used for package delivery on PyPI.

Contents:

4.1 API: Decorators: *ThreadPooled*, *threadpooled*.

class `pooled.ThreadPooled`

Post function to ThreadPoolExecutor.

__init__ (*func*, *, *loop_getter*, *loop_getter_need_context*)

Parameters

- **func** (*typing.Optional[typing.Callable[... typing.Union[typing.Any, typing.Awaitable]]]*) – function to wrap
- **loop_getter** (*typing.Union[None, typing.Callable[... asyncio.AbstractEventLoop], asyncio.AbstractEventLoop]*) – Method to get event loop, if wrap in asyncio task
- **loop_getter_need_context** (*bool*) – Loop getter requires function context

Note: Attributes is read-only

loop_getter

typing.Union[None, typing.Callable[... asyncio.AbstractEventLoop], asyncio.AbstractEventLoop] Loop getter. If None: use `concurrent.futures.Future`, else use `EventLoop` for wrapped function.

loop_getter_need_context

bool - Loop getter will use function call arguments.

executor

`ThreadPoolExecutor` instance. Class-wide.

Return type *ThreadPoolExecutor*

_func

`typing.Optional[typing.Callable[... , typing.Union[typing.Any, typing.Awaitable]]]` Wrapped function. Used for inheritance only.

classmethod configure (*max_workers=None*)

Pool executor create and configure.

Parameters **max_workers** (`typing.Optional[int]`) – Maximum workers

Note: `max_workers=None` means `CPU_COUNT * 5`, it's default value.

classmethod shutdown ()

Shutdown executor.

__call__ (*args, **kwargs)

Decorator entry point.

Return type `typing.Union[concurrent.futures.Future, typing.Awaitable, typing.Callable[... , typing.Union[typing.Awaitable, concurrent.futures.Future]]]`

`pooled.threadpooled` (*func*, *, *loop_getter*, *loop_getter_need_context*)

Post function to `ThreadPoolExecutor`.

Parameters

- **func** (`typing.Optional[typing.Callable[... , typing.Union[typing.Any, typing.Awaitable]]]`) – function to wrap
- **loop_getter** (`typing.Union[None, typing.Callable[... , asyncio.AbstractEventLoop], asyncio.AbstractEventLoop]`) – Method to get event loop, if wrap in asyncio task
- **loop_getter_need_context** (*bool*) – Loop getter requires function context

Return type `typing.Union[ThreadPooled, typing.Callable[... , typing.Union[concurrent.futures.Future, typing.Awaitable]]]`

Not exported, but public accessed data type:

class `pooled.ThreadPoolExecutor` (*max_workers=None*)

Provide readers for protected attributes.

Simply extend `concurrent.futures.ThreadPoolExecutor`.

Parameters **max_workers** (`typing.Optional[int]`) – Maximum workers allowed. If none: `cpu_count() * 5`

max_workers

`int` - max workers variable.

is_shutdown

`bool` - executor in shutdown state.

4.2 API: Decorators: *Threaded* class and *threaded* function.

class `threaded.Threaded`

Run function in separate thread.

```
__init__ (name=None, daemon=False, started=False)
```

Parameters

- **name** (*typing.Optional[typing.Union[str, typing.Callable[... , typing.Union[typing.Any, typing.Awaitable]]]]*) – New thread name. If callable: use as wrapped function. If none: use wrapped function name.
- **daemon** (*bool*) – Daemonize thread.
- **started** (*bool*) – Return started thread

Note: Attributes is read-only.

name

typing.Optional[str] - New thread name. If none: use wrapped function name.

started

bool

daemon

bool

_func

Wrapped function. Used for inheritance only.

__call__ (*args, **kwargs)

Decorator entry point.

Return type typing.Union[threading.Thread, typing.Callable[... , threading.Thread]]**threaded.threaded** (name=None, daemon=False, started=False)

Run function in separate thread.

Parameters

- **name** (*typing.Optional[typing.Union[str, typing.Callable[... , typing.Union[typing.Any, typing.Awaitable]]]]*) – New thread name. If callable: use as wrapped function. If none: use wrapped function name.
- **daemon** (*bool*) – Daemonize thread.
- **started** (*bool*) – Return started thread

Return type typing.Union[*Threaded*, typing.Callable[... , threading.Thread]]

4.3 API: Decorators: *AsyncIOTask*, *asynciotask*.

class pooled.**AsyncIOTask**

Wrap to asyncio.Task.

__init__ (func, *, loop_getter, loop_getter_need_context)**Parameters**

- **func** (*typing.Optional[typing.Callable[... , typing.Awaitable]]*) – function to wrap
- **loop_getter** (*typing.Union[typing.Callable[... , asyncio.AbstractEventLoop], asyncio.AbstractEventLoop]*) – Method to get event loop, if wrap in asyncio task

- **loop_getter_need_context** (*bool*) – Loop getter requires function context

Note: Attributes is read-only

loop_getter

`typing.Union[typing.Callable[... , asyncio.AbstractEventLoop], asyncio.AbstractEventLoop]` Loop getter.

loop_getter_need_context

`bool` - Loop getter will use function call arguments.

_func

`typing.Optional[typing.Callable[... , typing.Awaitable]]` Wrapped function.
Used for inheritance only.

__call__ (**args, **kwargs*)

Decorator entry point.

Return type `typing.Union[AsyncIOTask, typing.Callable[... , asyncio.Task]]`

`pooled.asynciotask` (*func, *, loop_getter, loop_getter_need_context*)

Wrap to `asyncio.Task`.

Parameters

- **func** (`typing.Optional[typing.Callable[... , typing.Awaitable]]`) – function to wrap
- **loop_getter** (`typing.Union[typing.Callable[... , asyncio.AbstractEventLoop], asyncio.AbstractEventLoop]`) – Method to get event loop, if wrap in `asyncio` task
- **loop_getter_need_context** (*bool*) – Loop getter requires function context

Return type `typing.Union[AsyncIOTask, typing.Callable[... , asyncio.Task]]`

CHAPTER 5

Indices and tables

- `genindex`
- `modindex`
- `search`

p

pooled, [11](#)

t

threaded, [12](#)

Symbols

`__call__()` (*pooled.AsyncIOTask* method), 14
`__call__()` (*pooled.ThreadPooled* method), 12
`__call__()` (*threaded.Threaded* method), 13
`__init__()` (*pooled.AsyncIOTask* method), 13
`__init__()` (*pooled.ThreadPooled* method), 11
`__init__()` (*threaded.Threaded* method), 12
`_func` (*pooled.AsyncIOTask* attribute), 14
`_func` (*pooled.ThreadPooled* attribute), 12
`_func` (*threaded.Threaded* attribute), 13

A

`AsyncIOTask` (class in *pooled*), 13
`asynciotask()` (in module *pooled*), 14

C

`configure()` (*pooled.ThreadPooled* class method), 12

D

`daemon` (*threaded.Threaded* attribute), 13

E

`executor` (*pooled.ThreadPooled* attribute), 11

I

`is_shutdown` (*pooled.ThreadPoolExecutor* attribute),
12

L

`loop_getter` (*pooled.AsyncIOTask* attribute), 14
`loop_getter` (*pooled.ThreadPooled* attribute), 11
`loop_getter_need_context`
(*pooled.AsyncIOTask* attribute), 14
`loop_getter_need_context`
(*pooled.ThreadPooled* attribute), 11

M

`max_workers` (*pooled.ThreadPoolExecutor* attribute),
12

N

`name` (*threaded.Threaded* attribute), 13

P

`pooled` (module), 11, 13

S

`shutdown()` (*pooled.ThreadPooled* class method), 12
`started` (*threaded.Threaded* attribute), 13

T

`Threaded` (class in *threaded*), 12
`threaded` (module), 12
`threaded()` (in module *threaded*), 13
`ThreadPooled` (class in *pooled*), 11
`threadpooled()` (in module *pooled*), 12
`ThreadPoolExecutor` (class in *pooled*), 12