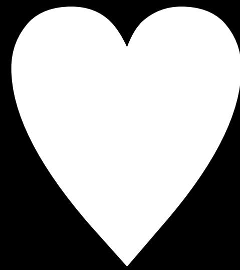
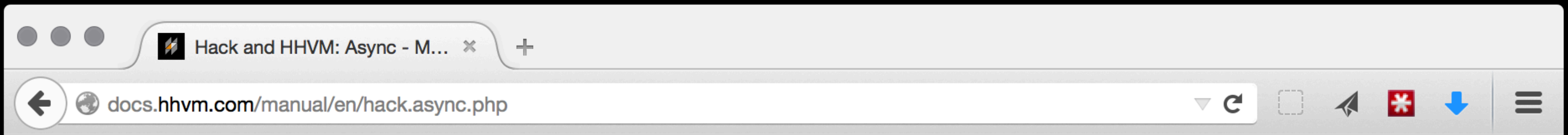
THINKING IN COROUTINES

Łukasz Langa
ambv on #python

fb.me/ambv
@llanga
lukasz@langa.pl


ASYNC



Async

Table of Contents

- async and await
- Type Annotating Async Functions
- Example: Coalesced Fetching
- Signature Examples
- Awaitables vs Continuations
- Async Lambdas
- Async Builtins
- External Resources

Asynchronous programming refers to a programming design pattern that

folly/folly/io/async at master...
GitHub, Inc. (US) https://github.com/facebook/folly/tree/master/folly/io/async

README.md

folly/io/async: An object-oriented wrapper around libevent

---

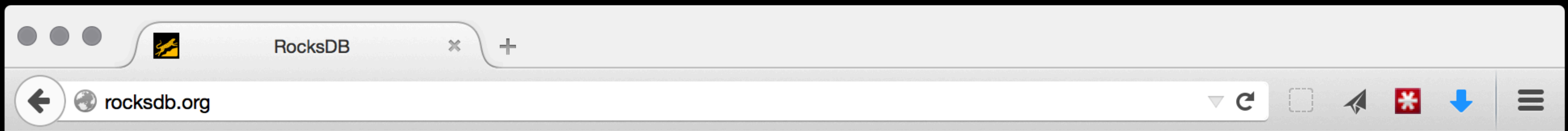
libevent is an excellent cross-platform eventing library. Folly's async provides C++ object wrappers for fd callbacks and event_base, as well as providing implementations for many common types of fd uses.

EventBase

The main libevent / epoll loop. Generally there is a single EventBase per thread, and once started, nothing else happens on the thread except fd callbacks. For example:

```
EventBase base;
auto thread = std::thread([&](){
  base.loopForever();
});
```

EventBase has built-in support for message passing between threads. To send a function to be run in the EventBase



BLOG OVERVIEW WIKI GITHUB

A persistent key-value store for fast storage environments

GET STARTED

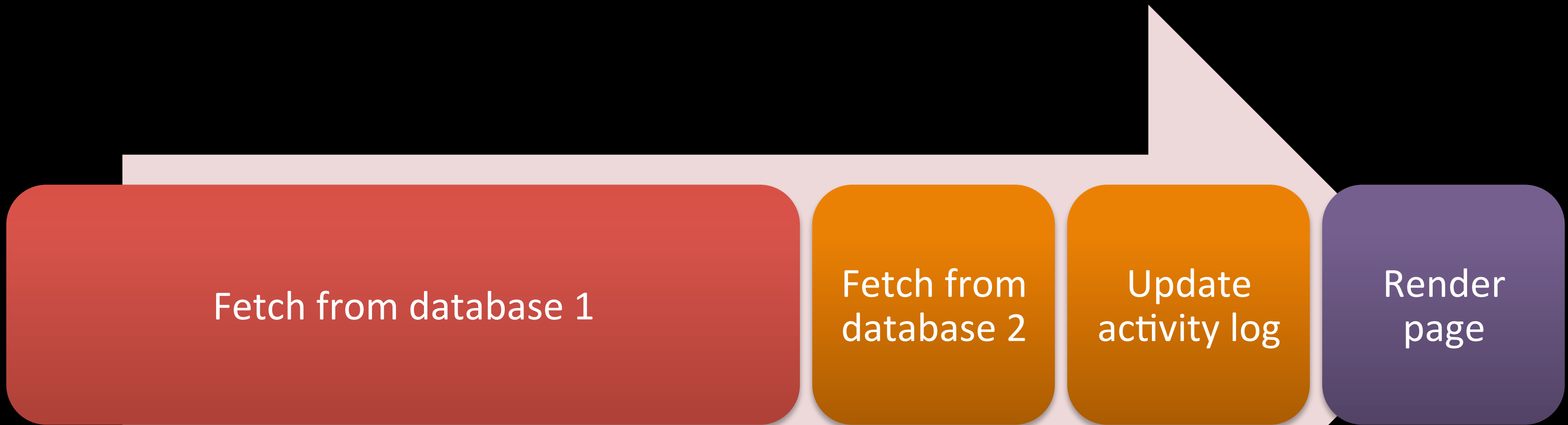
What is RocksDB?

RocksDB is an embeddable persistent key-value store for fast storage. RocksDB can also be the foundation for a client-server database but our current focus is on embedded workloads.

RocksDB builds on **LevelDB** to be scalable to run on servers with many CPU cores, to efficiently use fast storage, to support IO-bound, in-memory and write-once workloads, and to be flexible to allow for innovation.

For more background on RocksDB, see **Dhruba Borthakur's introductory talk** from the Data @ Scale 2013 conference.

BUT WHY?



Fetch from database 1

Render page

Fetch from database 2

Update activity log

Warning: Unresponsive script

A script on this page may be busy, or it may have stopped responding. You can stop the script now, or you can continue to see if the script will complete.

Script: https://fbstatic-a.akamaihd.net/rsrc.php/v2/yp/r/21bfDHaSBi6.js:68

☐ Don't ask me again

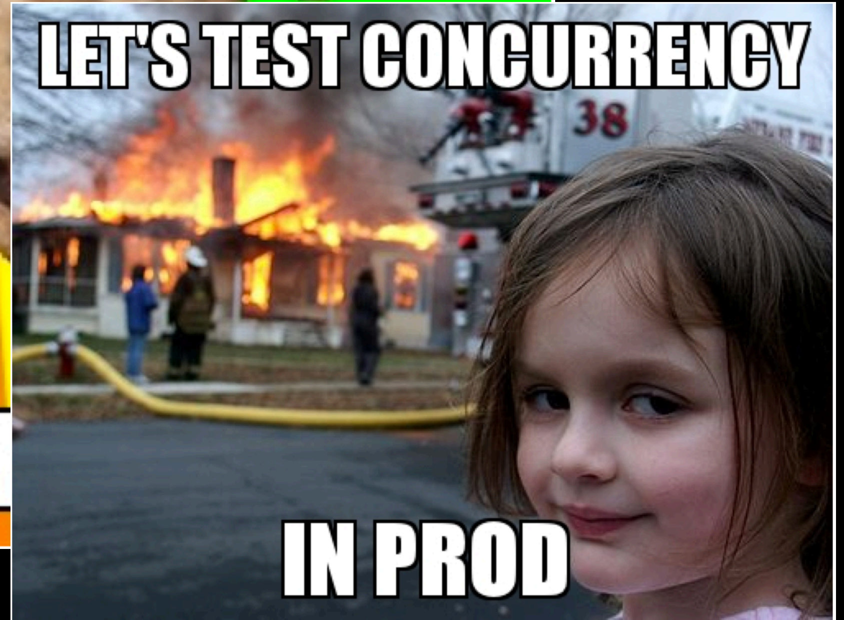
Continue | Stop script

JUST TAKE IT EASY
AND SPAWN A THREAD

I SEE DEADLOCKS

YOU SAY "THREADS"
I HEAR "JOB SECURITY"

I'M NOT SAYING USE THREADS
BUT USE THREADS


LET'S TEST CONCURRENCY
IN PROD

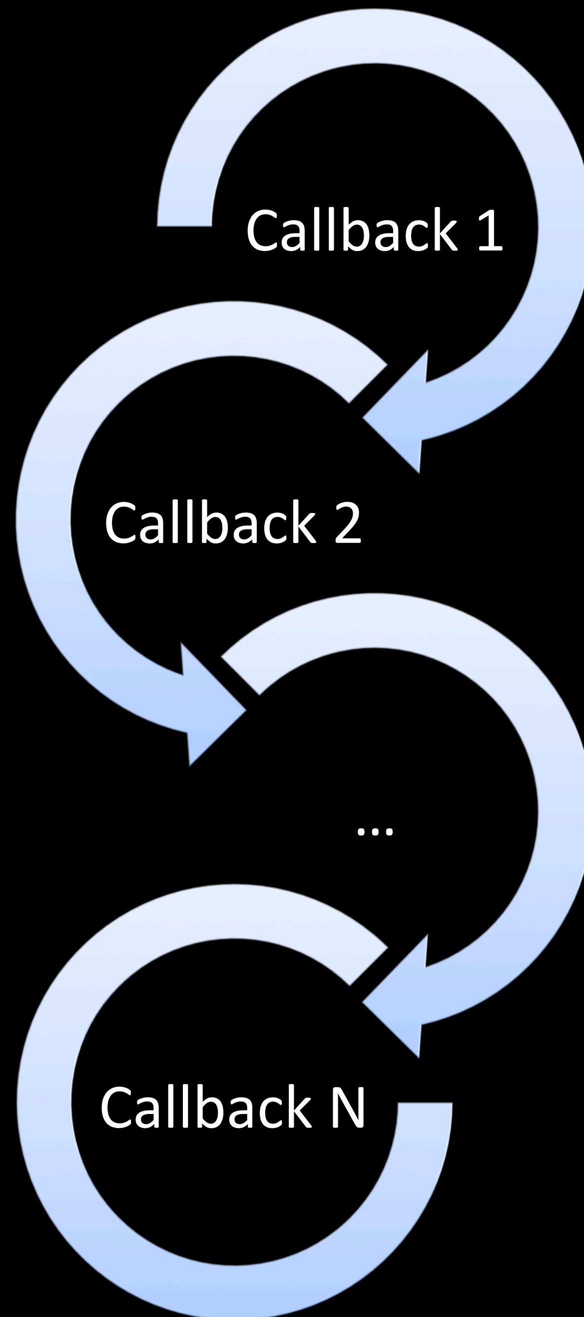
THREADS IN PYTHON?

THAT MUST BE SO CONCURRENT

GLOBAL INTERPRETER LOCK

SUDDENLY

ASYNCIO


Callback 1
Callback 2
...
Callback N

```python
import asyncio

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_forever()
```

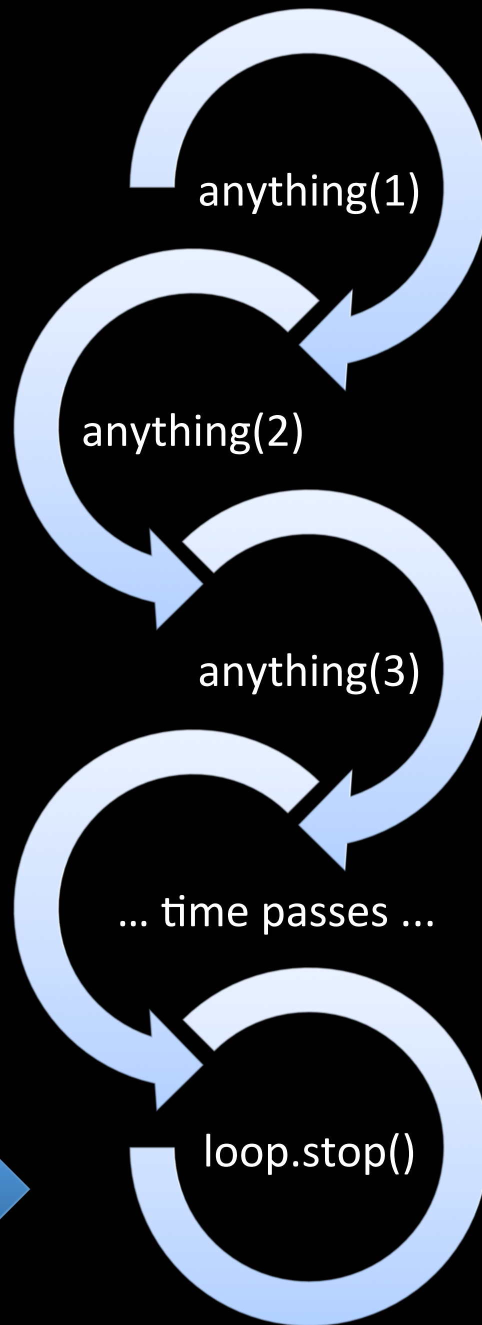
```python
class BaseEventLoop:
    ...
    def run_forever(self):
        """Run until stop() is called."""
        self._check_closed()
        self._running = True
        try:
            while True:
                try:
                    self._run_once()
                except _StopError:
                    break
        finally:
            self._running = False
```

WHAT DOES
THE LOOP
CALL?

```
def anything(i):
    print(i, datetime.datetime.now())

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.call_later(2, loop.stop)
    for i in range(1, 4):
        loop.call_soon(anything, i)
    try:
        loop.run_forever()
    finally:
        loop.close()
```

```
$ python3 exmpl.py
1 2015-03-10 22:19:49.508753
2 2015-03-10 22:19:49.508803
3 2015-03-10 22:19:49.508828
$
```

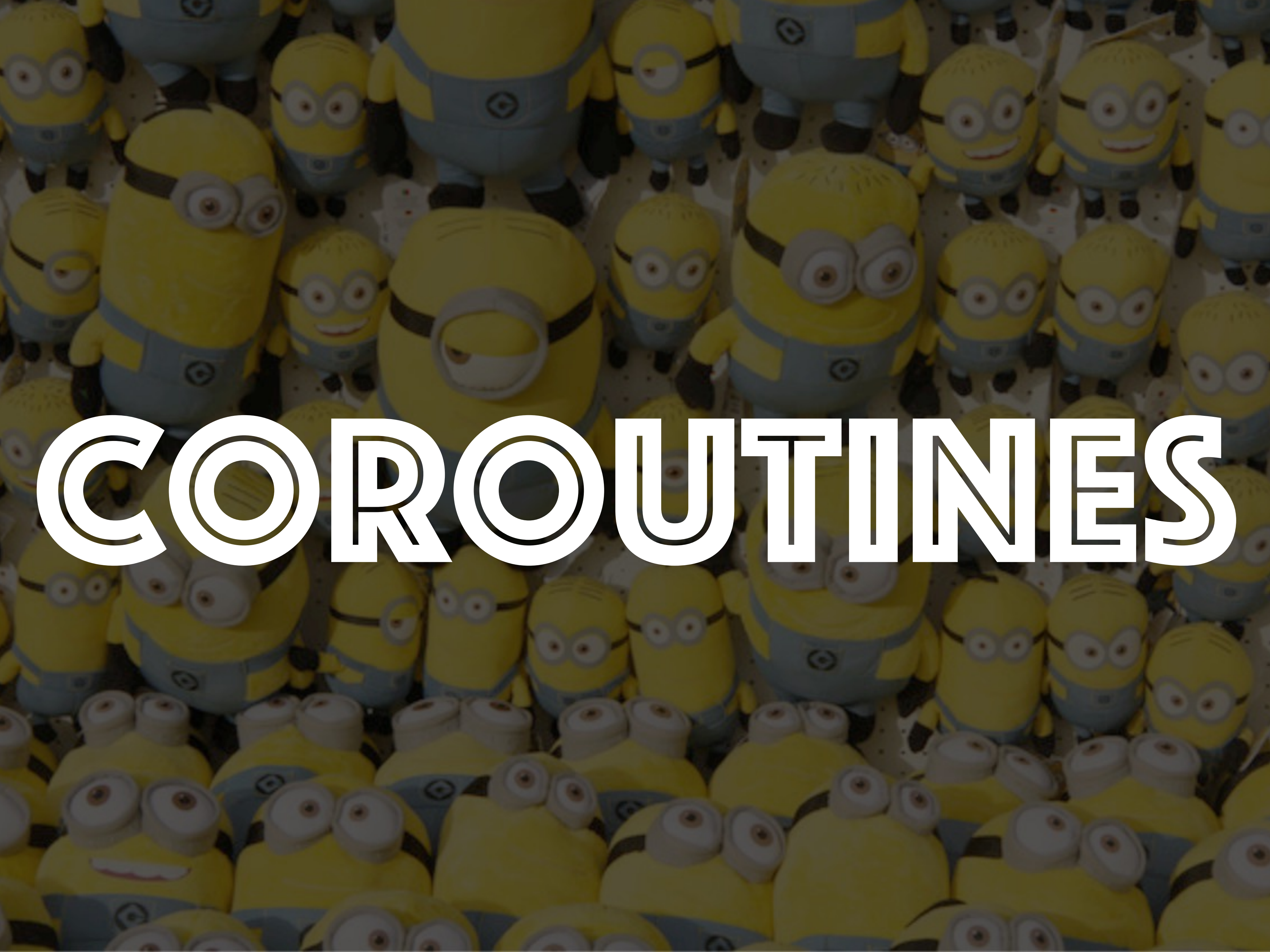
No busy looping,
rather something
like select(2)



```
def anything(i):
    print(i, datetime.datetime.now())
    time.sleep(i)

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.call_later(2, loop.stop)
    for i in range(1, 4):
        loop.call_soon(anything, i)
    try:
        loop.run_forever()
    finally:
        loop.close()
```

```
$ python3 exmpl.py
1 2015-03-10 22:36:11.733630
2 2015-03-10 22:36:12.737269
3 2015-03-10 22:36:14.742384
$
```

```
$ PYTHONASYNCIODEBUG=1 python3 exmpl.py
1 2015-03-10 22:34:48.553062
Executing <Handle anything(1) at example.py:5
created at example.py:13> took 1.001 seconds
2 2015-03-10 22:34:49.554451
Executing <Handle anything(2) at example.py:5
created at example2.py:13> took 2.005 seconds
3 2015-03-10 22:34:51.559950
Executing <Handle anything(3) at example.py:5
created at example2.py:13> took 3.003 seconds
$
```


COROUTINES

```
# plain old blocking function
def anything(i):
    print(i, datetime.datetime.now())
    time.sleep(i)
```

```
# a coroutine function
async def anything(i):
    print(i, datetime.datetime.now())
    await asyncio.sleep(i)
```

```python
async def anything(i):
    print(i, datetime.datetime.now())
    await asyncio.sleep(i)
```

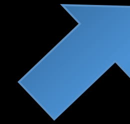
```python
anything
```

coroutine function

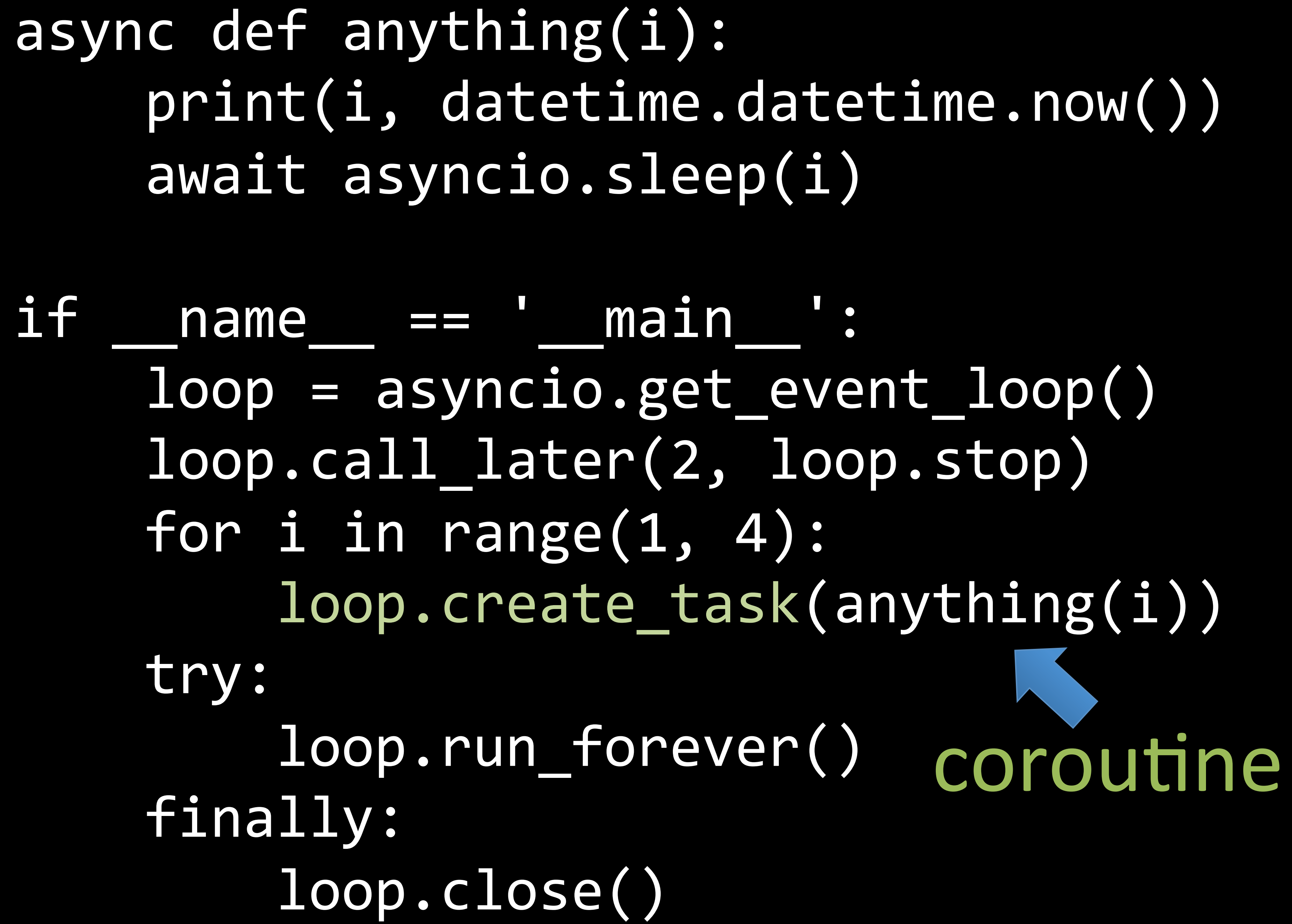
```python
anything(1)
```

coroutine

```
async def anything(i):
    print(i, datetime.datetime.now())
    await asyncio.sleep(i)
```



coroutine, too!


async def anything(i):
    print(i, datetime.datetime.now())
    await asyncio.sleep(i)

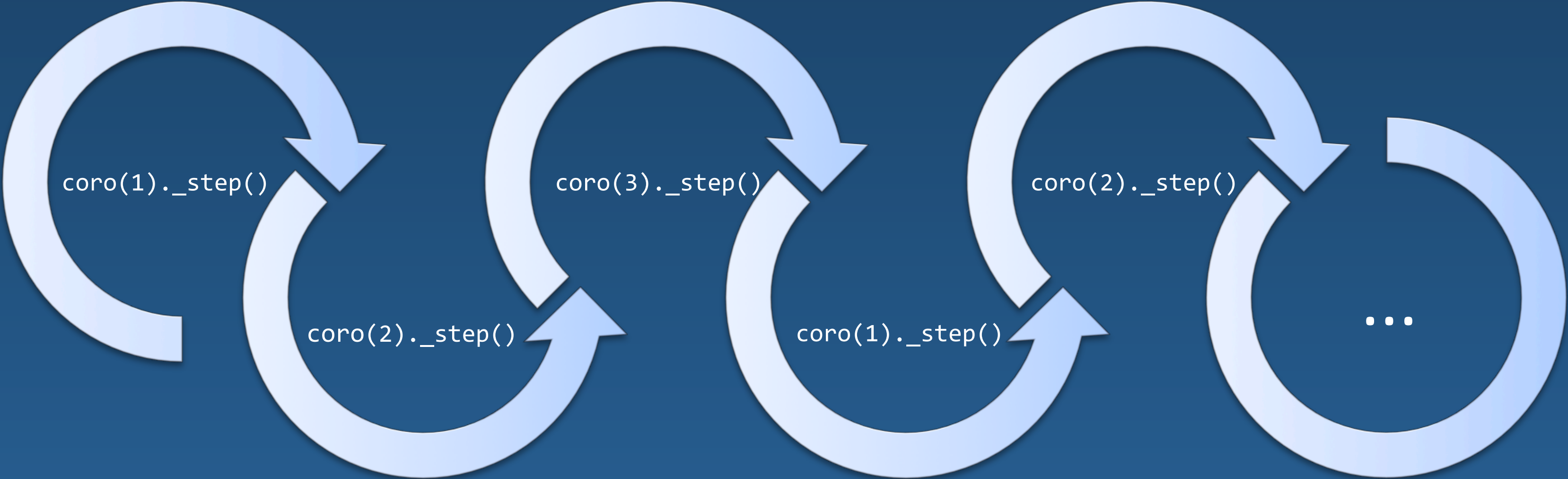
if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.call_later(2, loop.stop)
    for i in range(1, 4):
        loop.create_task(anything(i))
    try:
        loop.run_forever()
    finally:
        loop.close()
coroutine

```
$ python3 exmpl.py
1 2015-03-11 01:29:17.045832
2 2015-03-11 01:29:17.045921
3 2015-03-11 01:29:17.045964
$
```

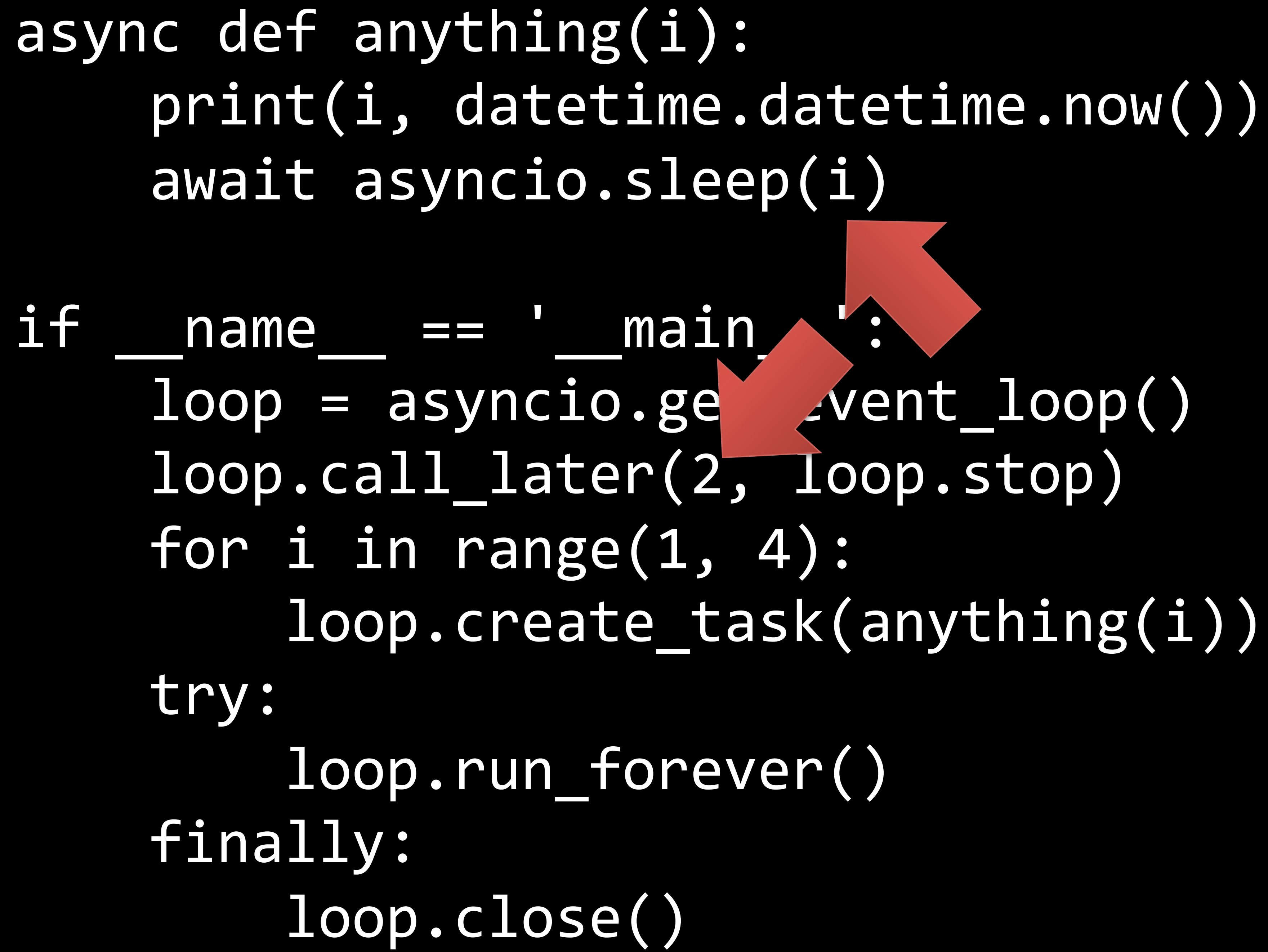
```
async def anything(i):
    print(i, datetime.datetime.now())
    await asyncio.sleep(i)

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.call_later(2, loop.stop)
    for i in range(1, 4):
        loop.create_task(anything(i))
    try:
        loop.run_forever()
        loop.close()
```

```
class Task(futures.Future):
  def __init__(self, coro, loop=None):
    super().__init__(loop=loop)
    ...
    self._loop.call_soon(self._step)
```

```
class Task(futures.Future):
  def _step(self):
    ...
    try:
      ...
      result = next(self._coro)
    except StopIteration as exc:
      self.set_result(exc.value)
    except BaseException as exc:
      self.set_exception(exc)
      raise
    else:
      ...
      self._loop.call_soon(self._step)
```


coro(1)._step()
coro(2)._step()
coro(3)._step()
coro(1)._step()
coro(2)._step()
...

Blair



async def anything(i):
print(i, datetime.datetime.now())
await asyncio.sleep(i)
if __name__ == '__main__':
loop = asyncio.ge event_loop()
loop.call_later(2, loop.stop)
for i in range(1, 4):
loop.create_task(anything(i))
try:
loop.run_forever()
finally:
loop.close()

```
$ PYTHONASYNCIODEBUG=1 python3 exmpl.py
1 2015-03-11 01:51:33.264004
2 2015-03-11 01:51:33.264498
3 2015-03-11 01:51:33.265810
Task was destroyed but it is pending!
Object created at (most recent call last):
  File "exmpl.py", line 14, in <module>
    loop.create_task(anything(i))
task: <Task pending coro=<anything() running at
exmpl.py:8> wait_for=<Future pending cb=[Task._wakeup()]
created at exmpl.py:14>
Task was destroyed but it is pending!
Object created at (most recent call last):
  File "exmpl.py", line 14, in <module>
    loop.create_task(anything(i))
task: <Task pending coro=<anything() running at
exmpl.py:8> wait_for=<Future pending cb=[Task._wakeup()]
created at exmpl.py:14>
```

```
loop.run_until_complete(anything(1))
```

coroutine
or task

```python
async def anything(i):
  print(i, datetime.datetime.now())
  await asyncio.sleep(i)

if __name__ == '__main__':
  loop = asyncio.get_event_loop()
  tasks = [loop.create_task(anything(i))
           for i in range(1, 4)]
  try:
    loop.run_until_complete(
      asyncio.wait(tasks))
  finally:
    loop.close()
```

```
$ PYTHONASYNCIODEBUG=1 python3 exmpl.py
1 2015-03-11 02:25:14.785569
2 2015-03-11 02:25:14.787152
3 2015-03-11 02:25:14.787581
$
```

```python
async def anything(i):
  print(i, datetime.datetime.now())
  await asyncio.sleep(i)


if __name__ == '__main__':
  loop = asyncio.get_event_loop()
  tasks = [loop.create_task(anything(i))
           for i in range(1, 4)]
  try:
    loop.run_until_complete(
      asyncio.wait(tasks))
  finally:
    loop.close()
```

```python
async def anything(i):
  print(i, datetime.datetime.now())
  await asyncio.sleep(i)
  return i, datetime.datetime.now()

if __name__ == '__main__':
  loop = asyncio.get_event_loop()
  tasks = [loop.create_task(anything(i))
           for i in range(1, 4)]
  try:
    loop.run_until_complete(
      asyncio.wait(tasks))
    for task in tasks:
        print(*task.result())
  finally:
    loop.close()
```

```
$ PYTHONASYNCIODEBUG=1 python3 exmpl.py
1 2015-03-11 15:03:14.701144
2 2015-03-11 15:03:14.702612
3 2015-03-11 15:03:14.703101
1 2015-03-11 15:03:15.702948
2 2015-03-11 15:03:16.703643
3 2015-03-11 15:03:17.708134
```

```
if __name__ == '__main__':
  loop = asyncio.get_event_loop()
  task = loop.create_task(anything(3))
  try:
    result = loop.run_until_complete(task)
    print(*result)
  finally:
    loop.close()
```

```
if __name__ == '__main__':
  loop = asyncio.get_event_loop()
  try:
    result = loop.run_until_complete(
      anything(3))
    print(*result)
  finally:
    loop.close()
```

```
if __name__ == '__main__':
  loop = asyncio.get_event_loop()
  task = loop.create_task(anything('g'))
  try:
    result = loop.run_until_complete(task)
  except TypeError:
    print('Type error: ', task.exception())
  else:
    print(*result)
  finally:
    loop.close()
```

```
$ PYTHONASYNCIODEBUG=1 python3 exmpl.py
g 2015-03-11 15:07:47.862128
Type error:  unsupported operand type(s) for
+: 'float' and 'str'
```

```
async def anything(i):
  print(i, datetime.datetime.now())

  await asyncio.sleep(i)

  return i, datetime.datetime.now()
```

```
async def anything(i):
  print(i, datetime.datetime.now())
  try:
    await asyncio.sleep(i)
  except TypeError:
    i = 0
  return i, datetime.datetime.now()
```

```
$ PYTHONASYNCIODEBUG=1 python3 exmpl.py
g 2015-03-11 15:09:06.617283
0 2015-03-11 15:09:06.617661
```

Invoking coroutines

- Outside a coroutine:

```
task = loop.create_task(coro())
result = loop.run_until_complete(
  coro())
```

- Inside a coroutine:

```
task = loop.create_task(coro())
result = await coro()
```

WHAT'S
INCLUDED?
eneloop

18.5.1.5. Creating connections

coroutine `BaseEventLoop.`**`create_connection`**(*protocol_factory, host=None, port=None, *, ssl=None, family=0, proto=0, flags=0, sock=None, local_addr=None, server_hostname=None*)

Create a streaming transport connection to a given Internet *host* and *port*: socket family `AF_INET` or `AF_INET6` depending on *host* (or *family* if specified), socket type `SOCK_STREAM`. *protocol_factory* must be a callable returning a *protocol* instance.

This method is a *coroutine* which will try to establish the connection in the background. When successful, the coroutine returns a `(transport, protocol)` pair.

The chronological synopsis of the underlying operation is as follows:

1. The connection is established, and a *transport* is created to represent it.
2. *protocol_factory* is called without arguments and must return a *protocol* instance.
3. The protocol instance is tied to the transport, and its `connection_made()` method is called.
4. The coroutine returns successfully with the `(transport, protocol)` pair.

The created transport is an implementation-dependent bidirectional stream.

> **Note:** *protocol_factory* can be any kind of callable, not necessarily a class. For example, if you want to use a pre-created protocol instance, you can pass `lambda: my_protocol`.

Options allowing to change how the connection is created:

- *ssl*: if given and not false, a SSL/TLS transport is created (by default a plain TCP transport is created). If *ssl* is a `ssl.SSLContext` object, this context is used to create the transport; if *ssl* is `True`, a context with some unspecified default settings is used.

See also: *SSL/TLS security considerations*

18.5.1.6. Creating listening connections

coroutine `BaseEventLoop.`**`create_server`**(*protocol_factory, host=None, port=None, *, family=socket.AF_UNSPEC, flags=socket.AI_PASSIVE, sock=None, backlog=100, ssl=None, reuse_address=None*)

Create a TCP server (socket type `SOCK_STREAM`) bound to *host* and *port*.

Return a `Server` object, its `sockets` attribute contains created sockets. Use the `Server.close()` method to stop the server: close listening sockets.

Parameters:

- If *host* is an empty string or `None`, all interfaces are assumed and a list of multiple sockets will be returned (most likely one for IPv4 and another one for IPv6).
- *family* can be set to either `socket.AF_INET` or `AF_INET6` to force the socket to use IPv4 or IPv6. If not set it will be determined from host (defaults to `socket.AF_UNSPEC`).
- *flags* is a bitmask for `getaddrinfo()`.
- *sock* can optionally be specified in order to use a preexisting socket object. If specified, *host* and *port* should be omitted (must be `None`).
- *backlog* is the maximum number of queued connections passed to `listen()` (defaults to 100).
- *ssl* can be set to an `SSLContext` to enable SSL over the accepted connections.
- *reuse_address* tells the kernel to reuse a local socket in TIME_WAIT state, without waiting for its natural timeout to expire. If not specified will automatically be set to True on UNIX.

This method is a *coroutine*.

On Windows with `ProactorEventLoop`, SSL/TLS is not supported.

See also: The function `start_server()` creates a (`StreamReader`, `StreamWriter`) pair and calls back a function with this pair.

18.5.1.7. Watch file descriptors

On Windows with `SelectorEventLoop`, only socket handles are supported (ex: pipe file descriptors are not supported).

On Windows with `ProactorEventLoop`, these methods are not supported.

`BaseEventLoop.`**`add_reader`**(*fd*, *callback*, **args*)

Start watching the file descriptor for read availability and then call the *callback* with specified arguments.

Use functools.partial to pass keywords to the callback.

`BaseEventLoop.`**`remove_reader`**(*fd*)

Stop watching the file descriptor for read availability.

`BaseEventLoop.`**`add_writer`**(*fd*, *callback*, **args*)

Start watching the file descriptor for write availability and then call the *callback* with specified arguments.

Use functools.partial to pass keywords to the callback.

`BaseEventLoop.`**`remove_writer`**(*fd*)

Stop watching the file descriptor for write availability.

The *watch a file descriptor for read events* example uses the low-level `BaseEventLoop.add_reader()` method to register the file descriptor of a socket.

18.5.1.8. Low-level socket operations

coroutine `BaseEventLoop.`**`sock_recv`**(*sock*, *nbytes*)

18.5.6. Subprocess — Pyth... ×

Python Software Foundation (US) https://docs.python.org/3.4/library/asyncio-subprocess.html

Table Of Contents

18.5.6. Subprocess
- 18.5.6.1. Windows event loop
- 18.5.6.2. Create a subprocess: high-level API using Process
- 18.5.6.3. Create a subprocess: low-level API using subprocess.Popen
- 18.5.6.4. Constants
- 18.5.6.5. Process

18.5.7. Subprocess and threads

18.5.8. Subprocess examples
- 18.5.8.1. Subprocess using transport and protocol
- 18.5.8.2. Subprocess using streams

Previous topic

18.5.5. Streams (high-level API)

Next topic

18.5.9. Synchronization primitives

This Page

Report a Bug
Show Source

18.5.6. Subprocess

18.5.6.2. Create a subprocess: high-level API using Process ¶

coroutine `asyncio.`**`create_subprocess_exec`**(**args, stdin=None, stdout=None, stderr=None, loop=None, limit=None, **kwds*)

Create a subprocess.

The *limit* parameter sets the buffer limit passed to the `StreamReader`. See `BaseEventLoop.subprocess_exec()` for other parameters.

Return a `Process` instance.

This function is a *coroutine*.

coroutine `asyncio.`**`create_subprocess_shell`**(*cmd, stdin=None, stdout=None, stderr=None, loop=None, limit=None, **kwds*)

Run the shell command *cmd*.

The *limit* parameter sets the buffer limit passed to the `StreamReader`. See `BaseEventLoop.subprocess_shell()` for other parameters.

Return a `Process` instance.

It is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities. The `shlex.quote()` function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

This function is a *coroutine*.

Use the `BaseEventLoop.connect_read_pipe()` and `BaseEventLoop.connect_write_pipe()` methods to connect pipes.

Table Of Contents
- 18.5.9. Synchronization primitives
  - 18.5.9.1. Locks
    - 18.5.9.1.1. Lock
    - 18.5.9.1.2. Event
    - 18.5.9.1.3. Condition
  - 18.5.9.2. Semaphores
    - 18.5.9.2.1. Semaphore
    - 18.5.9.2.2. BoundedSemaphore

Previous topic
18.5.6. Subprocess

Next topic
18.5.10. Queues

This Page
Report a Bug
Show Source

Quick search
Go
Enter search terms or a module, class or function name.

18.5.9. Synchronization primitives ¶

Locks:

- `Lock`
- `Event`
- `Condition`

Semaphores:

- `Semaphore`
- `BoundedSemaphore`

asyncio lock API was designed to be close to classes of the `threading` module (`Lock`, `Event`, `Condition`, `Semaphore`, `BoundedSemaphore`), but it has no *timeout* parameter. The `asyncio.wait_for()` function can be used to cancel a task after a timeout.

18.5.9.1. Locks

18.5.9.1.1. Lock

class asyncio.**Lock**(**, loop=None*)

Primitive lock objects.

A primitive lock is a synchronization primitive that is not owned by a particular coroutine when locked. A primitive lock is in one of two states, 'locked' or 'unlocked'.

It is created in the unlocked state. It has two basic methods, `acquire()` and `release()`. When the state is

18.5.10. Queues — Python ...

Python Software Foundation (US) https://docs.python.org/3.4/library/asyncio-queue.html

Table Of Contents

- 18.5.10. Queues
  - 18.5.10.1. Queue
  - 18.5.10.2. PriorityQueue
  - 18.5.10.3. LifoQueue
    - 18.5.10.3.1. JoinableQueue
    - 18.5.10.3.2. Exceptions

Previous topic

18.5.9. Synchronization primitives

Next topic

18.5.11. Develop with asyncio

This Page

Report a Bug
Show Source

Quick search

Go

Enter search terms or a module, class or function name.

18.5.10. Queues

Queues:

- `Queue`
- `PriorityQueue`
- `LifoQueue`
- `JoinableQueue`

asyncio queue API was designed to be close to classes of the `queue` module (`Queue`, `PriorityQueue`, `LifoQueue`), but it has no *timeout* parameter. The `asyncio.wait_for()` function can be used to cancel a task after a timeout.

18.5.10.1. Queue

class `asyncio.`**`Queue`**(*maxsize=0*, ***, *loop=None*)

A queue, useful for coordinating producer and consumer coroutines.

If *maxsize* is less than or equal to zero, the queue size is infinite. If it is an integer greater than `0`, then `yield from put()` will block when the queue reaches *maxsize*, until an item is removed by `get()`.

Unlike the standard library `queue`, you can reliably know this Queue's size with `qsize()`, since your single-threaded asyncio application won't be interrupted between calling `qsize()` and doing an operation on the Queue.

This class is *not thread safe*.

EXECUTORS

```python
def anything(i):
  print(i, datetime.datetime.now())
  time.sleep(i)

if __name__ == '__main__':
  loop = asyncio.get_event_loop()
  loop.call_later(2, loop.stop)
  with ThreadPoolExecutor(max_workers=8) as e:
    for i in range(1, 4):
      loop.run_in_executor(e, anything, i)
    try:
      loop.run_forever()
    finally:
      loop.close()
```

```
$ PYTHONASYNCIODEBUG=1 python3 exmpl.py
1 2015-03-11 00:35:33.193047
2 2015-03-11 00:35:33.194048
3 2015-03-11 00:35:33.195291
$
```

AVAILABLE EXECUTORS

ThreadPoolExecutor

- Less overhead
- GIL still there
- Passes arbitrary arguments
- Based on threading

ProcessPoolExecutor

- More overhead
- No GIL
- Passes only picklable arguments
- Based on multiprocessing

AT FACEBOOK

```
/* pub_service.thrift */

include "common/fb303/if/fb303.thrift"
include "wormhole/common/types.thrift"

namespace py wormhole.monitoring.pub_service
namespace py.asyncio wormhole.monitoring_asyncio.pub_service

service PublisherService extends fb303.FacebookService {
  void startPublishers(1: string dataSourceUrl)
    throws (1: PublisherServiceException ex),

  ...
```

```python
# fake_publisher.py

import asyncio

from wormhole.monitoring_asyncio.pub_service import \
    PublisherService
from fb303_asyncio.FacebookBase import FacebookBase

class FakePublisherServer(FacebookBase, PublisherService.Iface):
    def __init__(self, version, *, pub_port, loop=None):
        super().__init__('fake-publisher-server')
        self._version = version
        self._pub_port = pub_port
        self.loop = loop or asyncio.get_event_loop()
        self.resetCounter('publisher.pub.port', self._pub_port)

    def getVersion(self):
        return self._version

    ...
```

```python
from thrift.server.TAsyncioServer import \
    ThriftAsyncServerFactory
...

if __name__ == '__main__':
    args = docopt.docopt(__doc__, argv)
    loop = asyncio.get_event_loop()
    handler = FakePublisherServer(
        version=args['__version__'],
        pub_port=args['--pub_port'],
        loop=loop,
    )
    server = loop.run_until_complete(
        ThriftAsyncServerFactory(
            handler, port=args['--fb303_port'], loop=loop,
        ),
    )
    try:
        loop.run_forever()
    finally:
        server.close()
        loop.close()
```

```
from wormhole.monitoring_asyncio.pub_service import PublisherService
from thrift.server.TAsyncioServer import ThriftClientProtocolFactory

class PublisherMonitor:
  ...

  @asyncio.coroutine
  def connectToPublisher(self):
    try:
      transport, protocol = yield from self.loop.create_connection(
        ThriftClientProtocolFactory(
          PublisherService.Client, self.loop, timeouts={'': 2},
        ),
        host='::1',
        port=self.port,
      )
      return protocol
    except OSError:
      self.log.error("Can't connect to port %d", self.port)
      return None
```

```
from wormhole.monitoring_asyncio.pub_service import PublisherService
from thrift.server.TAsyncioServer import ThriftClientProtocolFactory

class PublisherMonitor:
  ...

  @asyncio.coroutine
  def updatePublisherStatus(self):
    protocol = yield from self.connectToPublisher()
    try:
      self.pub_status = yield from protocol.client.getStatus()
    except (
      PublisherServiceException,
      TTransportException,
      TApplicationException,
    ):
      self.log.error("Can't talk to the Publisher at %d", self.port)
    finally:
      protocol.close()
```

```
class PublisherMonitor:
    ...

    @asyncio.coroutine
    def run(self):
        cmd_line = [
            'wormhole_publisher',
            '--pub_port={}'.format(self.port),
        ]
       self.proc = yield from asyncio.create_subprocess_exec(
            *cmd_line,
            stdout=asyncio.subprocess.PIPE,
            stderr=asyncio.subprocess.PIPE,
            preexec_fn=ensure_dead_with_parent
        )
        # it's running now
        self.loop.create_task(self.tail_logs(self.proc.stderr))
        self.loop.create_task(self.watchdog())
        # wait for it to die
        status_code = yield from self.proc.wait()
```

```
class PublisherMonitor:
    ...

    @asyncio.coroutine
    def run(self):
        cmd_line = [
            'wormhole_publisher',
            '--pub_port={}'.format(self.port),
        ]
       self.proc = yield from asyncio.create_subprocess_exec(
            *cmd_line,
            stdout=asyncio.subprocess.PIPE,
            stderr=asyncio.subprocess.PIPE,
            preexec_fn=ensure_dead_with_parent
        )
        # it's running now
        self.loop.create_task(self.tail_logs(self.proc.stderr))
        self.loop.create_task(self.watchdog())
        # wait for it to die
        status_code = yield from self.proc.wait()
```

```python
import ctypes
import signal

def ensure_dead_with_parent():
    """A last resort measure to make sure this
    process dies with its parent.

    Defensive programming for unhandled errors.
    """
    PR_SET_PDEATHSIG = 1 # include/uapi/linux/prctl.h
    libc = ctypes.CDLL(ctypes.util.find_library('c'))
    libc.prctl(PR_SET_PDEATHSIG, signal.SIGKILL)
```

```
for sig in [
    signal.SIGALRM, signal.SIGVTALRM,
    signal.SIGPROF, signal.SIGINT,
    signal.SIGTERM,
]:
    loop.add_signal_handler(sig, sighandler)
```

Welcome to aiomysql's docum… +

http://aiomysql.readthedocs.org/en/latest/

aiomysql

Search docs

`aiomysql` — API Reference

Connection

Cursor

Pool

Tutorial

`aiomysql.sa` — support for SQLAlchemy functional SQL layer

Examples of aiomysql usage

Glossary

Read the Docs v: latest

Welcome to aiomysql's documentation!

aiomysql is a library for accessing a *MySQL* database from the asyncio (PEP-3156/tulip) framework. It depends and reuses most parts of *PyMySQL* . **aiomysql** tries to be like awesome aiopg library and preserve same api, look and feel.

Internally **aiomysql** is copy of PyMySQL, underlying io calls switched to async, basically `yield from` and `asyncio.coroutine` added in proper places. *sqlalchemy* support ported from aiopg.

Features

- Implements *asyncio DBAPI like* interface for *MySQL*. It includes *Connection*, *Cursor* and *Pool* objects.
- Implements *optional* support for charming *sqlalchemy* functional sql layer.

Basics

aiomysql based on *PyMySQL* , and provides same api, you just need to use `yield from conn.f()` instead of just call `conn.f()` for every method.

ADVICE

USE PYTHON 3.5+

WRITE UNIT TESTS

SET UP DEBUGGING

```python
if __name__ == '__main__':
    import logging
    log = logging.getLogger('asyncio')
    log.setLevel(logging.DEBUG)
    import gc
    gc.set_debug(gc.DEBUG_UNCOLLECTABLE)
    loop = asyncio.get_event_loop()
    loop.set_debug(True)
    try:
        loop.run_forever()
    finally:
        loop.close()
```

```
$ PYTHONASYNCIODEBUG=1 python3 server.py
```

```python
if __name__ == '__main__':
  loop = asyncio.get_event_loop()
  anything(10)
  try:
    loop.run_until_complete(
      asyncio.sleep(1)
    )
  finally:
    loop.close()
```

```
$ PYTHONASYNCIODEBUG=1 python3 exmpl.py
<CoroWrapper anything() running at
exmpl.py:5, created at exmpl.py:12> was
never yielded from
Coroutine object created at (most recent
call last):
  File "exmpl.py", line 12, in <module>
    anything(10)
$
```

DO NOT USE STOPITERATION

```
def generator():
    yield 1
    yield 2
    raise StopIteration    # wrong!
                           # see PEP-479
```

```
def generator():
  yield 1
  yield 2
  return
```

PREFER PROCESSPOOL EXECUTORS

READ THE DOCS, DON'T BE AFRAID OF THE SOURCE

PY 3.4 - ASYNCIO

```
@asyncio.coroutine
def anything(i):
  print(i, datetime.datetime.now())
  try:
    yield from asyncio.sleep(i)
  except TypeError:
    i = 0
  return i, datetime.datetime.now()
```

PY 3.5 - PEP 492

```
async def anything(i):
  print(i, datetime.datetime.now())
  try:
    await asyncio.sleep(i)
  except TypeError:
    i = 0
  return i, datetime.datetime.now()
```

PEP 484
TYPE HINTS

```
def greeting(name: str) -> str:
        return 'Hello ' + name
```

Images used

- Memes approved by and used according to best practices of the #memepolice
- "Prison Planet" by Mark Rain https://www.flickr.com/photos/azrainman/1003163361/
- "Minions" by Richard Croft cc-by-sa 2.0 http://www.geograph.org.uk/photo/3666790
- Still from "The Fox (What Does The Fox Say?" by Ylvis (fair use)
- "Everybody Lies" by Alphanza1 http://alphanza1.deviantart.com/art/Everybody-Lies-362332275
- "Batteries not included" by Pete Slater https://www.flickr.com/photos/johnnywashngo/6200247250/
- A public domain image of a Mexican execution from 1914

This slidedeck as PDF:

fb.me/coroutines

Łukasz Langa
ambv on #python

fb.me/ambv
@llanga
lukasz@langa.pl