Asynchronous programming with Coroutines in Python
Asynchronous programming with Coroutines

in Python

Ewoud Van Craeynest

January 31, 2017
introduction

asyncio in Python 3.4 (briefly)

asyncio in Python 3.5

summary

(extra slides)
Introduction

What is async programming for us today?
What is async programming?

- writing concurrent applications
- without the use of threads or multiprocessing
- in a cooperative multitasking fashion

\(^1\) not to be confused with parallelism
Introduction

Not unique to Python

- many other languages also provide similar functionality
- .NET got credited by Guido
- Go, Dart, JavaScript, Scala, Clojure, Erlang, ...
- but also Perl and Ruby
Introduction

Not unique to Python

- C++17 was rumoured to include it
- some talk about it on conferences
- but targeted for a new TS
- and hopefully C++20
A style of writing code

- that doesn’t use blocking calls
- but rather an event loop
  - (that mustn’t be blocked)
Introduction

Blocking

What’s the issue with blocking api’s?

Why do we now dislike blocking?
Introduction
Blocking

What’s the issue with blocking api’s?
▶ your thread is ”busy”
▶ but not doing anything useful
  ▶ waiting for disk
  ▶ waiting for network/database
  ▶ waiting for serial port or other io

Why do we now dislike blocking?
▶ because now we have an alternative
Isn’t that why threading exists?

Why do we now frown upon threads?
Isn’t that why threading exists?

- yes, threads were designed for multitasking
- at operating system level
Why do we now frown upon threads?

- context switches are expensive
  - don't scale well
  - think large numbers of sockets (C10K)
- synchronisation is hard to get right
  - unpredictable scheduling
  - race conditions
  - deadlock
  - starvation
Introduction

Threading

**Threads**: the goto of our generation

- at least *for concurrency*
In the multicore age

- for **parallelism** however
- threads (or multiprocessing) are still a must
**Introduction**

**Threading**

**Threads no more?**

- No!
- just less of them
- one thread for all connections
  - i.s.o. one thread per connection
- one for all video stuff
- one for all screen io
- one for all ...
Introduction

Non blocking calls

Circling back

- we want code not to block a thread
- because we want to do things concurrently
Introduction
Non blocking calls

Wait a minute . . .

Isn’t all code blocking in a way?
Isn’t all code blocking in a way?

- indeed, but let’s make a distinction
  - executing code, crunching data
  - waiting for I/O operations to finish
- can’t do much about the first
  - except parallelize
- but the second is the subject of our attention
Non blocking calls

Non blocking I/O

- we want I/O code not to block a thread
- to do things concurrently

- we need new api’s
Not so new

- turns out, those api’s, aren’t all that new
- Python has a long history of async programming
- though not in the way we know it now
History of async programming api’s

- There were a few predecessors to what we got in Python3.5
  - gevent (greenlets, c stack hack)
  - tulip (now asyncio)
  - twisted (event driven)
  - tornado
  - ...
  - all a bit hacked on top of Python2

- asyncio provisional package in 3.4
Predecessors vs 3.5

- all rely on some form of select/poll loops
- so does Python asyncio
- but with nicer syntax
- supported from the language itself
- using new keywords
Let’s take a look
asyncio

- python 3.4 added asyncio
- Asynchronous I/O, event loop, coroutines and tasks
- this is where our story really starts
provisional in 3.4

- It was, however, a work in progress

Note

The asyncio package has been included in the standard library on a provisional basis. Backwards incompatible changes (up to and including removal of the module) may occur if deemed necessary by the core developers.
An example:

- note the `@asyncio.coroutine` decorator
- and `yield from` statement

```python
@asyncio.coroutine
def print_hello():
    while True:
        print("{} - Hello world!".format(int(time())))
        yield from asyncio.sleep(3)
```
coroutines

- are "special" functions
- which can be suspended and resumed

```
@asyncio.coroutine
def print_hello():
    while True:
        print("{} - Hello world!".format(int(time())))
        yield from asyncio.sleep(3)
```
new style

- using coroutines is considered "new style" async
- though we now consider this example code "old syntax"
  - see Python 3.5 coroutines later on

new style, not newest syntax

```python
@asyncio.coroutine
def print_hello():
    while True:
        print("{} - Hello world!".format(int(time())))
        yield from asyncio.sleep(3)
```
coroutine api

- async code that does use coroutines
- needs a **coroutine api**
- like `asyncio.open_connection` and its return objects

```python
@asyncio.coroutine
def tcp_echo_client(message, loop):
    reader, writer = yield from asyncio.open_connection('127.0.0.1', 8888,
                                                        loop=loop)

    print('Send: %r' % message)
    writer.write(message.encode())

    data = yield from reader.read(100)
    print('Received: %r' % data.decode())
    writer.close()
```
coroutines

- Python 3.5 added **new coroutine keywords**
- async def and await
- removing the need for the `@asyncio.coroutine` decorator and `yield from`
asyncio in Python 3.5
provisional

still provisional in 3.5

- The documentation has still the same note

Note
The asyncio package has been included in the standard library on a provisional basis. Backwards incompatible changes (up to and including removal of the module) may occur if deemed necessary by the core developers.

\(^4\) note is gone in Python3.6 docs
asyncio in Python 3.5
coroutines

same examples

▶ using the new syntax async/await

```
new syntax
async def print_hello():
    while True:
        print("{} - Hello world!".format(int(time())))
        await asyncio.sleep(3)
```
coroutines reiterated

- are "special" functions
- which can be suspended and resumed

**first coroutine**

async def print_hello():
    while True:
        print("{} - Hello world!".format(int(time())))
        await asyncio.sleep(3)
event loop

- an event loop will take care of starting and resuming tasks
- but in turn, it claims the thread you’re on

```python
loop = asyncio.get_event_loop()
loop.run_until_complete(print_hello())  # blocking!
```
Not all async code uses coroutines. In fact, many of the predecessors used callbacks triggered by certain events.

```python
async def process_input():
    text = sys.stdin.readline()
    n = int(text.strip())
    print('fib({}) = {}'.format(n, timed_fib(n)))

loop.add_reader(sys.stdin, process_input)
```
callback style async

- though used intensively in the past
- it **escalates quickly** in a cascade of callbacks and state machines
- becoming a bit of a **design anti-pattern** in itself
  - callback hell . . .
- but we didn’t really have another option
- and it did get us out of threading!
callback style async

- asyncio’s event loop supports scheduling regular callbacks
  - using a fifo queue of registered callbacks
  - in this case as soon as possible

**async using callback**

```python
asyncio

def hello_world(loop):
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()

loop.call_soon(hello_world, loop)  # <--

loop.run_forever()

loop.close()
```
asyncio callbacks

callback style async

- delayed callbacks are also possible
  - call_later
  - call_at

- event loop has own internal clock for computing timeouts

delayed async using callback

```python
loop.call_later(0.5, hello_world, loop)
```
asyncio in Python 3.5

coroutines

same examples

▶ using the new syntax asyncio/await with streams

```python
async def tcp_echo_client(message, loop):
    reader, writer = await asyncio.open_connection('127.0.0.1', 8888, loop=loop)

    print('Send: %r' % message)
    writer.write(message.encode())

    data = await reader.read(100)
    print('Received: %r' % data.decode())

    print('Close the socket')
    writer.close()
```
suspend on yield from

- coroutine will be suspended
- until open_connection has finished

coroutine api

```python
reader, writer = await asyncio.open_connection('127.0.0.1', 8888, loop=loop)
```
coroutine api

- also the objects returned by `open_connection` have coroutines
- though only for what blocks
  - `write` is documented not to block
- but we do want to suspend until `read` finishes
- without blocking the thread

```python
coroutine api
writer.write(message.encode())
data = await reader.read(100)
```
asyncio
callbacks in Python 3.5

coroutine api

- written as if it were synchronous code
- no callbacks and keeping state
- but nonblocking with suspend and resume behaviour

coroutine api

```python
async def tcp_echo_client(message, loop):
    reader, writer = await asyncio.open_connection('127.0.0.1', 8888, loop=loop)
    print('Send: %r' % message)
    writer.write(message.encode())

    data = await reader.read(100)
    print('Received: %r' % data.decode())
    writer.close()
```
asynchronous in Python 3.5

coroutines api

coroutine api

- as with Python 3.4
- we need alternatives for all blocking api’s we might want to use

- as usual Python comes with (some) batteries included
- additional batteries on PyPI
- though it must be mentioned that Twisted currently offers more
batteries include:

- low level socket operations
- streams & connections
- sleep
- subprocess
- synchronisation primitives
  - nonblocking, with coroutines
asyncio.org

- lists **PyPI** libraries to use with **asyncio**
- things like:
  - HTTP, ZMQ, DNS, Redis, memcache, Mongo, SQL, ...
  - REST, WebSockets, IRC, wsgi, Docker, ...
  - SIP, SSH, XMPP, SMTP, ...
  - files ⁵, queue, read-write lock
  - pyserial, cron, blender

⁵ workaround using threads
asyncio in Python 3.5

asyncserial

- coroutine api for serial port

asyncserial example

```python
async def foo():
    port = asyncserial.AsyncSerial('/dev/ttyUSB1')
    await port.write(somedata)
    response = await port.read(5)
    await some_handle_response(response)
```
aiozmq.rpc

- RPC mechanism on top of ZeroMQ using coroutines

```python
client = await aiozmq.rpc.connect_rpc(connect='tcp://127.0.0.1:5555')
ret = await client.call.get_some_value()
await client.call.start_some_remote_action(some_calc(ret))
await asyncio.sleep(42)
await client.call.stop_some_remote_action()
```
asyncio in Python 3.5

aiohttp

- HTTP using coroutines
- even with and for can use coroutines

```python
async def fetch(session, url):
    with aiohttp.Timeout(10):
        async with session.get(url) as response:
            return await response.text()

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    with aiohttp.ClientSession(loop=loop) as session:
        html = loop.run_until_complete(fetch(session, 'http://python.org'))
        print(html)
```
asycnio in Python 3.5

coroutines

AsyncSSSH

```python
coroutines
async def run_client():
    async with asyncssh.connect('localhost') as conn:
        stdin, stdout, stderr = await conn.open_session('echo "Hello!"

        output = await stdout.read()
        print(output, end='')

        await stdout.channel.wait_closed()

        status = stdout.channel.get_exit_status()
        if status:
            print('Program exited with status %d' % status, file=sys.stderr)
        else:
            print('Program exited successfully')

asyncio.get_event_loop().run_until_complete(run_client())
```
asyncio in Python 3.5

blocking stuff

- blocking functions should not be called directly
- it will block the loop and all other tasks
- if no high level async API available
  - run in executor, like ThreadPoolExecutor

blocking stuff in executor

```python
await loop.run_in_executor(None, my_blocking_func, arg1, arg2)
```
asyncio in Python 3.5

coroutines

aiofiles

- file IO is blocking
- not easily made asynchronous
- aiofiles delegates to thread pool
  - unblocking your event loop
  - using the future mechanism
- discussion with Guido on GitHub about asynchronous files

blocking stuff behind the scenes

```python
async with aiofiles.open('filename', mode='r') as f:
    contents = await f.read()
```
### asyncio in Python 3.5

**Testing**

#### asynctest

- You’ll want to test coroutines
- But that requires a loop running
  - Loop aware test
  - asynctest module

```python
import asynctest
import aiozmq.rpc

class MyRpcTest(asynctest.TestCase):
    async def setUp(self):
        self.client = await aiozmq.rpc.connect_rpc(
            connect='tcp://127.0.0.1:5555')
```

---

54
don’t write your unit tests this way!

```python
async def test_some_remote_action(self):
    cc = self.client.call
    r = await cc.get_some_value()
    self.assertGreater(someValue, r)
    await cc.start_some_remote_action(some_calc(ret))
    for _ in range(5):
        await time.sleep(0.5)
        newr = await cc.get_some_value()
        self.assertGreater(newr, r)
        r = newr
    await cc.stop_some_remote_action()
```
asyncio in Python 3.5

testing

**asynctest**

- loop aware test
- ideally run unrelated tests concurrently on the same loop
  - realistic?
    - perhaps not

```python
import asynctest
```
asynctest: other features

- ClockedTestCase
  - allows to control the loop clock
  - run timed events
  - without waiting for the wall clock
    - accelerated tests anyone?

```python
import asynctest
```
asyncio in Python 3.5

testing

asyncio: other features

- CoroutineMock
- FileMock
- SocketMock

*asynctest*

```python
import asyncio.selector
```
pytest-asyncio

- for those on pytest iso unittest
- haven’t tried it ...
- claim custom event loop support
- monkey patching coroutines allowed

```python
@pytest.mark.asyncio
def test_some_asyncio_code():
    res = await library.do_something()
    assert b'expected result' == res
```
some applications might require stopping the loop
basically any `await` statement is an opportunity for the loop to stop
it will warn you about unfinished scheduled tasks on the loop

```
loop.stop()
```
cancelling a task

- sometimes not required to stop whole loop
- a single task might suffice

```python
sometask = loop.create_task(my_coroutine())
...
sometask.cancel()
```
asyncio in Python 3.5

stopping loop

threadsafety

- the whole thing isn’t threadsafe
- why would it?
- so take precautions from other threads

```
loop.call_soon_threadsafe(loop.stop)
loop.call_soon_threadsafe(sometask.cancel)
```
asyncio in Python 3.5

exceptions

- raised exceptions from a coroutine
- get set on the internal future object
- and reraised when awaited on

```python
async def foo():
    raise Exception()

async def bar():
    await foo()  # Exception time
```
asyncio in Python 3.5

exceptions

- but if never awaited
- aka exception never consumed
- it’s logged with traceback ⁶

```
async def foo():
    raise Exception()

asyncio.ensure_future(foo())  # will log unconsumed exception
```

⁶Get more logging by enabling asyncio debug mode
asyncio in Python 3.5

logging

- asyncio logs information on the logging module in logger 'asyncio'
- useful to redirect this away from frameworks that steal stdin and stdout
  - like robotframework
asynchronous in Python 3.5

alternatives

alternatives to asyncio

- as is to be expected ...
- not everyone completely agrees on Python’s implementation
- and offer partial or complete improvement over asyncio
asyncio in Python 3.5
alternatives to asyncio

other loops

▶ we can use loops other than the standard one
▶ like uvloop
  ▶ a fast, drop-in replacement of asyncio event loop
    ▶ implements asyncio.AbstractEventLoop
    ▶ promises Go-like performance
▶ expect others . . .

uvloop

```python
import asyncio
import uvloop
loop = uvloop.new_event_loop()
asyncio.set_event_loop(loop)
```

7https://github.com/MagicStack/uvloop
asyncio in Python 3.5
alternatives

curio: an alternative to asyncio

▶ by David Beazly
▶ based on task queueing
  ▶ not callback based event loop
▶ not just the loop
▶ complete async I/O library
  ▶ sockets, files, sleep, signals, synchronization, processes, ssl, ipc
  ▶ interactive monitoring
▶ claims 75 to 150% faster than asyncio
▶ claims 5 to 40% faster than uvloop
  ▶ and about the same speed as gevent
alternatives to asyncio

- I like standard stuff
- but benefits promised by others make them enticing . . .
summary
asynchronous programming

- concurrency without threading
- write suspendable functions
- as if it was synchronous code
asynchronous programming

- with callbacks in any version
- with `@asyncio.coroutine` in 3.4
- with `async def` coroutines in 3.5
asynchronous programming

- needs nonblocking api’s
- expect to see many of them
- even replacing blocking ones
  - as they can also be used blocking
what about **Python 3.6**?

- a *christmas present*
- minor *asyncio* improvements
- `run_coroutine_threadsafe`
  - submit coroutines to event loops in other threads
- `timeout()` context manager
  - simplifying timeouts handling code
- all changes backported to 3.5.x
Python 3.6

- deserves a presentation of its own
- but do checkout formatted string literals

```python
>>> name = "Fred"
>>> f"He said his name is {name}."
'He said his name is Fred.'
```
Thank you for joining!
KEEP CALM AND WRITE COROUTINES
asyncio in Python 3.5
behind the screens

extra slides
asyncio in Python 3.5
behind the screens

How to make your library coroutine enabled?

▶ it’s about operations that happen asynchronously
▶ often in hardware or network
▶ that finish somewhere in the future
usecase: asyncify pyserial

- use (part of) existing api
- use "everything is a file" to get async behaviour
- use future objects
asyncio in Python 3.5
behind the screens

**usecase**: asyncify pyserial

- use (part of) existing api

```python
reuse existing api
class AsyncSerialBase:
    def __init__(self, port=None, loop=None, timeout=None, write_timeout=None, inter_byte_timeout=None, **kwargs):
        if (timeout is not None or write_timeout is not None or inter_byte_timeout is not None):
            raise NotImplementedError("Use asyncio timeout features")
        self.ser = serial.serial_for_url(port, **kwargs)
        if loop is None:
            loop = asyncio.get_event_loop()
        self._loop = loop
```
asyncio in Python 3.5
behind the screens

**usecase**: asyncify pyserial

- use "everything is a file" to get async behaviour
  - async by callback that is

```python
go to async
def self._loop.add_reader(self.fileno(),
  self._read_ready, n)
```
usecase: asyncify pyserial

- use future objects
  - to replace callback api by coroutines

```python
def read(self, n):
    assert self.read_future is None or self.read_future.cancelled()
    future = asyncio.Future(loop=self._loop)
    ... # add_reader ...
    return future
```
use case: asyncify pyserial

- use future objects
  - to replace callback api by coroutines

```python
def _read_ready(self, n):
    self._loop.remove_reader(self.fileno())
    if not self.read_future.cancelled():
        try:
            res = os.read(self.fileno(), n)
        except Exception as exc:
            self.read_future.set_exception(exc)
        else:
            self.read_future.set_result(res)
    self.read_future = None
```
asyncio in Python 3.5
behind the screens

**usecase**: asyncify pyserial

- use **future** objects
- because a future returned by a regular function
- can be awaited on
- as if it was a coroutine

---

**future objects**

**return future**
asyncio in Python 3.5
behind the screens

usecase: asyncify pyserial

- cleanup

reuse existing api

```python
def close(self):
    if self.read_future is not None:
        self._loop.remove_reader(self.fileno())
    if self.write_future is not None:
        self._loop.remove_writer(self.fileno())
    self.ser.close()
```