Asynchronous programming with Coroutines

in Python

Intelligent Systems | ALTRAN

Asynchronous programming with Coroutines

in Python

Ewoud Van Craeynest

January 31, 2017

Table of Contents

Introduction

What is async programming for us today?

Introduction

What is async programming?

- writing concurrent applications [1]
- without the use of threads or multiprocessing
- in a **cooperative multitasking** fashion

[1] not to be confused with parallelism

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

Not unique to Python

- C++17 was rumoured to include it
- some talk about it on conferences
- but targetted 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

Introduction

Threading

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 [2]

- at least **for concurrency**

[2] doesn't mean they can't be useful if used correctly, like goto's

In the multicore age

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

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 . . .

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 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 where 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** [3]
- Asynchronous I/O, event loop, coroutines and tasks

- this is where our story really starts

[3] get it from PyPI for Python 3.3

asyncio

provisional

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

first coroutine

```
@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

first coroutine

```
@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 newst syntax

```
@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

coroutine api

```
@asyncio.coroutine
def tcp_echo_client(message, loop):
    reader, writer = yield from asyncio.open_connection('127.0.0.1
                                                        loop=loop)
    print('Send: %r' % message)
    writer.write(message.encode())

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

asyncio in Python 3.5

asyncio in Python 3.5

new keywords

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 [4]

- 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

running the event loop

```
loop = asyncio.get_event_loop()
loop.run_until_complete(print_hello())  # blocking!
```

old style async

- not all async code uses coroutines
- in fact, many of the predecessors used callbacks
  - triggered by certain events

async using callback

```
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

```
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()
```

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

```
loop.call_later(0.5, hello_world, loop)
```

same examples

- using the new syntax `async/await` with streams

new syntax

```
async def tcp_echo_client(message, loop):
    reader, writer = await asyncio.open_connection('127.0.0.1', 88
                                                   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

```
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

coroutine api

```
writer.write(message.encode())

data = await reader.read(100)
```

coroutine api

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

coroutine api

```
async def tcp_echo_client(message, loop):
    reader, writer = await asyncio.open_connection('127.0.0.1', 88
                                                   loop=loop)
    print('Send: %r' % message)
    writer.write(message.encode())

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

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

asyncio in Python 3.5

coroutines api

batteries include:

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

asyncio.org

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

asyncserial

- coroutine api for serial port

asyncserial example

```
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

RPC with coroutines

```
client = await aiozmq.rpc.connect_rpc(connect='tcp://127.0.0.1:555

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()
```

aiohttp

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

aiohttp

```
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)
```

AsyncSSH

aiohttp

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

        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=s
        else:
            print('Program exited successfully')

asyncio.get_event_loop().run_until_complete(run_client())
```

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

```
await loop.run_in_executor(None, my_blocking_func, arg1, arg2)
```

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

```
async with aiofiles.open('filename', mode='r') as f:
    contents = await f.read()
```

asynctest

- you'll want to test coroutines
- but that requires a loop running
  - loop aware test
  - asynctest module

asynctest

```
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')
```

asyncio in Python 3.5

testing

don't write your unit tests this way!

asynctest testcase

```
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()
```

asynctest

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

asynctest

```
import asynctest
```

asynctest: other features

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

asynctest

```
import asynctest
```

asynctest: other features

- `CoroutineMock`
- `FileMock`
- `SocketMock`

asynctest

```
import asynctest.selector
```

pytest-asyncio

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

pytest-asyncio

```
@pytest.mark.asyncio
async def test_some_asyncio_code():
    res = await library.do_something()
    assert b'expected result' == res
```

stopping the loop

- 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

stopping the loop

```
loop.stop()
```

cancelling a task

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

stopping the loop

```
sometask = loop.create_task(my_coroutine())
. . .
sometask.cancel()
```

threadsafety

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

stopping threadsafe

```
loop.call_soon_threadsafe(loop.stop)

loop.call_soon_threadsafe(sometask.cancel)
```

exceptions

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

exceptions

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

async def bar():
    await foo()   # Exception time
```

exceptions

- but if never awaited
- aka exception never consumed
- it's logged with traceback [6]

exceptions

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

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

[6] Get more logging by enabling asyncio debug mode

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

alternatives to asyncio

- as is to be expected . . .
- not everyone completely agrees on Python's implementation
- and offer partial or complete improvement over `asyncio`

other loops

- we can use loops other than the standard one
- like `uvloop` [7]
  - a fast, drop-in replacement of asyncio event loop
    - implements `asyncio.AbstractEventLoop`
  - promises Go-like performance
- expect others . . .

uvloop

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

[7] https://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

summary

summary

asynchronous programming

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

summary

asynchronous programming

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

summary

asynchronous programming

- needs nonblocking api's
- expect to see many of them
- even replacing blocking ones
  - as they can also be used blocking

summary

Python 3.6

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

formatted string literal

```
>>> name = "Fred"
>>> f"He said his name is {name}."
'He said his name is Fred.'
```

summary

Thank you for joining!

summary

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**

asyncio in Python 3.5

behind the screens

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

reuse existing api

```
class AsyncSerialBase:
    def __init__(self, port=None, loop=None, timeout=None, write_t
                 **kwargs):
        if (timeout is not None
                or write_timeout is not None
                or inter_byte_timeout is not None):
            raise NotImplementedError("Use asyncio timeout feature
        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

going async

```
self._loop.add_reader(self.fileno(),
                      self._read_ready, n)
```

asyncio in Python 3.5

behind the screens

usecase: asyncify `pyserial`

- use **future** objects
  - to replace callback api by coroutines

going async

```
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
```

asyncio in Python 3.5

behind the screens

usecase: asyncify `pyserial`

- use **future** objects
  - to replace callback api by coroutines

future objects

```
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

```
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()
```