What is this async def and why is it aswesome

Radu Ciorba {radu@devrandom.ro}

June 20, 2018

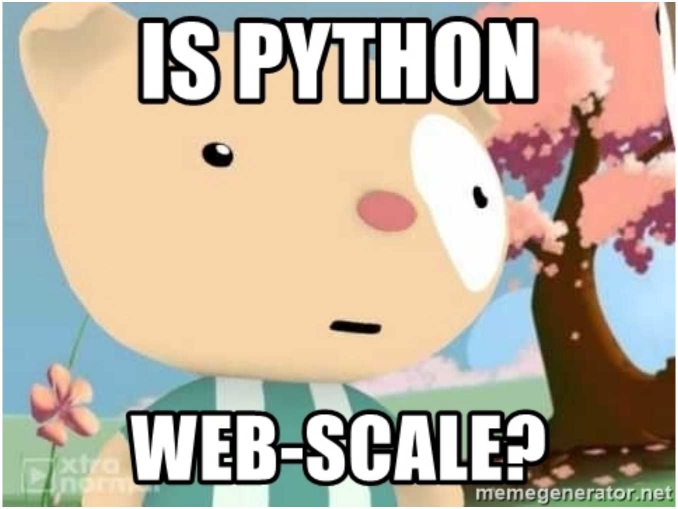
async def hello_world()

```
async def hello(name):
    return f"hello {name}" # P.S. how cool are format string!?

async def main():
    print(await hello("pytim"))
    print(await hello("pybalkan"))

print(main())
```

What is the problem we're trying to solve?

- Do stuff concurrently

What is the problem we're trying to solve?



CPU Bound vs IO Bound

- Mine Crypto-Currency - CPU heavy workload

CPU Bound vs IO Bound

A common use case: a python web application

- 1 Listen for incomming connections
- 2 Accept a connection
- 3 Read the request data from the connection
- 4 Parse the request
- 5 Call the request handler
- 6 Request handler queries the DB a few times
- 7 Render a template using the data we got in step 6
- 8 Write the response out to the connection
- Rinse, repeat.

Different approaches

- Threads
- Processes
- non blocking IO + event loop

Threads

Threads add overhead:

- Each thread has it's own stack
- Context switching
- Shared state - easy to introduce bugs
- The GIL - not a very big deal for IO bound problems
- Still - good enough in many cases

Event loop

- Single thread of execution
- Hand your code over to the EL to run it
- Don't do your own IO, ask the EL to do it for you
- Basically a form of cooperative concurrency
- No really, don't do your own IO, cause you'll block the EL

An Async Example

```
$("#ok-btn").on('click', function() {
  // do some stuff here then post some data
  $.post('/foo', json, function(data) {
      // on success continue doing stuff here
  })
  .fail(function(response) {
      // on error, do some cleanup here
  })
);
```

Also known as: Callback Hell

A Python Tornado Async Example

```python
class MessagesHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        """Display all messages."""
        self.write('<a href="/compose">Compose a message</a><br>')
        self.write('<ul>')
        db = self.settings['db']
        db.messages.find().sort([('_id', -1)]).each(self._got_message)

    def _got_message(self, message, error):
        if error:
            raise tornado.web.HTTPError(500, error)
        elif message:
            self.write('<li>%s</li>' % message['msg'])
        else:
            # Iteration complete
            self.write('</ul>')
            self.finish()
```

Async in Python

- 1999 - asyncore lands in the stdlib for 1.5.2
- 2002 - Twisted
- 2009 - Tornado
- 2009 - Gevent
- 2011 - PEP 3153 – Asynchronous IO support
- 2012 - Tulip & PEP 3156 – Asynchronous IO Support Rebooted
- 2013 - Tulip becomes AsyncIO in 3.4

AsyncIO goals

- Provide an API for event loops, allow interop between projects
- Offer a solution to avoid callbacks (not required)
- Work on python 3.4 as it was then, without changes to the language

Async without callbacks: Coroutines

- Provide an API for event loops, allow interop between projects
- Coroutines: a solution to avoid callbacks (not required)
- Work on python 3.4 as it was then, without changes to the language

Coroutines without changin the language?

```
def this_is_sort_of_a_coroutine():
    print("running")
    name = yield
    while True:
        name = yield f"hello {name}"
        if name is None:
            break
        print(f"hello {name}")
```

New language features: async def/await

```python
async def hello(name):
    return f"hello {name}"

async def main():
    print(await hello("pytim"))
    print(await hello("pybalkan"))
```

New language features: async for

```
import asyncio

import motor.motor_asyncio
from pymongo import ASCENDING, DESCENDING

client = motor.motor_asyncio.AsyncIOMotorClient('mongodb://localhost:27017'
db = client.clusterinfo

async def read_all():
    cursor = db.info.find().sort([("ta", ASCENDING), ])
    async for doc in cursor:
        await do_something_with(doc)
    # since 3.6 this also works:
    [doc async for doc in cursor()]
    {doc["_id"]: doc async for doc in cursor}

loop = asyncio.get_event_loop()
loop.run_until_complete(read_all())
```

New language features: async with

```python
import asyncio

class Cache:
    def __init__(self):
        self.lock = asyncio.Lock()
        self.cache = {}

    async def get(self, key):
        async with lock:
            self.cache.get(key)

    async def set(self, key, value):
        async with lock:
            self.cache[key] = value
```

New language features: yield in coroutines

```
# since 3.6 you can use yield in coroutines

async def async_generator():
    for n in range(10):
        yield n

async def main():
    async for n in async_generator():
        print(n)
```

Library support

- Sanic - Flask like web framework
- Motor - mongo driver, originally for tornado
- elasticsearcy-py-async - for ES
- aio-pika - for rabbitmq
- aio-libs project - aiomysql, aiopg, aioredis, aiohttp

Sanic + Motor

```python
from sanic import Sanic
from sanic.response import json

app = Sanic("pytim")
@app.listener("before_server_start")
def init_mongo(sanic, loop):
    # setup db ...

@app.route("svc_vt_info/search", methods=["POST"])
async def handle(request):
    _id = request.json.get("id")
    if _id is None:
        return json({"status": "error"}, status=400)
    result = await db.info.find({"_id": {"$in": _id}})\
      .to_list(length=None)
    if result is None:
        return json(None, status=404)
    return json({"data": result})

app.run(host="0.0.0.0", port=9090)
```

Where we are today

- Elegant way to write async code in Python
- A growing echosystem of asynio compattible libs

Alternatives

Gevent

- uses the greenlets package
- write code in blissful ignorance of the Event Loop
- gevent.monkey.patch_all() - replaces blocking IO calls with cooperative versions
- can't monkey patch C extensions - use pure python libs
- many 3rd party libs will work (requests, django, any pure python code should work)
- io in C extesions can still block the event loop
- unlike async/await your code can be suspended/resumed at any time (like threads)