Coroutines in Python

A Brief History from Iterators to Async Generators

Hello.

I’m Josh Marshall.
I work at uStudio (we’re hiring!)
I like event-driven things.

@joshmarshall | github.com/joshmarshall

(what we'll talk about)

What are **coroutines**?
Why should I care / **so what**?
How have they evolved in **Python**?
What's **next**?

(caveats)

This is from a user's perspective.

(Specifically event-driven / network services.)

Informal, leaky abstractions ahead.

Rabbit holes and limited time.

What is a **coroutine**?

Coroutines are a method of **cooperative multitasking** allowing execution to be explicitly **suspended** and **resumed**.

Coroutines also provide an alternate way of **reasoning** about **concurrent** operations.

“ Subroutines are special cases of … coroutines.

- Donald Knuth

Cooperative (vs preemptive)
Concurrent (vs parallel)
Explicit* (vs implicit)
**sometimes*

There’s also

Symmetric vs **Asymmetric,**

Stackless vs **Stackfull,**

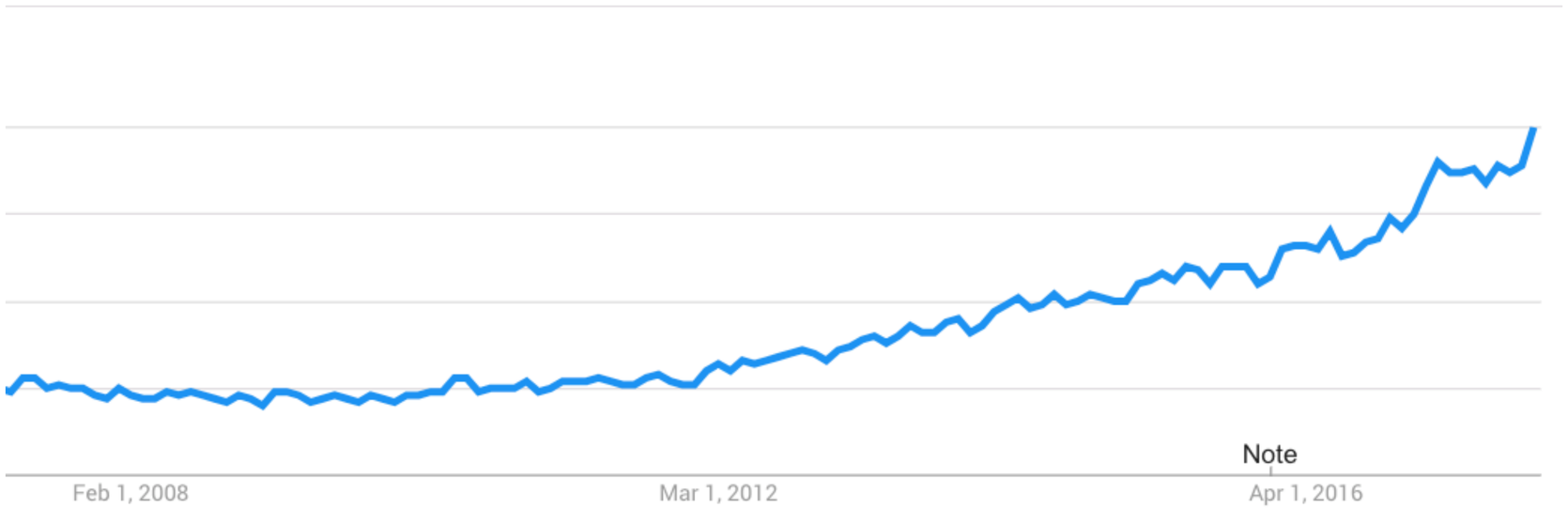
Lexical vs **Dynamic,**

etc.

Why should I **care?**

The rise of network-connected services requires us to be thinking about:

- Optimizing for **I/O**
- Accept || Read || Write
- **C10K** (C1M) Problem



This has led to the steady rise of asynchronous patterns / frameworks.

With asynchronous libraries, coroutines simplify **event-driven** logic and help **reduce bugs**.

Why should I care about anything except the new **hotness**?

Those who do not study the past are doomed to reinvent it, likely with a **partial**, **buggy** implementation.

Plus, I just like to see all the **turtles** on my way down.

Evolution of Python Coroutines

We begin our tale with a simple example use case.

(caveat auditorium)

Basic TCP-based service
Bidirectional JSON-RPC
Eventually talking to other services

PROTOCOL

```
{ int32 length; char[] message }
```

REQUEST

```
53{"jsonrpc": "2.0", "method": "foo", "id": “abcdefgh”}
```

RESPONSE

```
52{"jsonrpc": "2.0", "result": "ok", "id": "abcdefgh"}
```

```
class Server(object):

    def __init__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.bind((socket.gethostname(), 0))
        self.address, self.port = self.sock.getsockname()
        self.sock.listen(MAX_CLIENTS)

    def wait(self):
        client_sock, client_addr = self.sock.accept()
        while True:
            data = client_sock.recv(BUFFER_SIZE)
            result, _ = loads(data)
            print("Received {}".format(result))
            client_sock.sendall(msg({"result": "ok", "id": result["id"]}))
```

Example blocking server.

```
class Client(object):

    def __init__(self, address, port):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.address = address
        self.port = port

    def wait(self):
        print("Connecting to {} {}".format(self.address, self.port))
        self.sock.connect((self.address, self.port))
        self.sock.sendall(msg({"method": "connect"}))

        while True:
            data = self.sock.recv(BUFFER_SIZE)
            result, _ = loads(data)
            print("Received from server: {}".format(result))
            time.sleep(HEARTBEAT_INTERVAL)
            self.sock.sendall(msg({"method": "heartbeat"}))
```

Example blocking client.

Some of the problems

- Handles only **one client** at a time
- **Request / response** pattern
- Every network operation **blocks**

What are the **options** to solve the concurrent clients and operations?

When confronted with an I/O bound concurrency problem, some people think,

“Aha, I’ll use threads!”

Now you have
`ThreadError(“Timeout acquiring lock”)`
problems.

Use an evented system like
epoll / **kqueue** / **libuv** / etc…

…but now you are managing **callbacks** on **events**.

Callbacks

Iterators

Generators

Enhanced Generators

Subgenerator Delegation

Coroutines

Async Generators

```
class Server(object):

    def __init__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setblocking(0)
        self.sock.bind((socket.gethostname(), 0))
        self.address, self.port = self.sock.getsockname()
        self.recv_data = {}
        self.send_data = {}

    def register(self, selector):
        selector.add_fd(self.sock, READ, self.open_callback)
        print("Listening on {} {}".format(self.address, self.port))
        self.sock.listen(MAX_CLIENTS)

    def open_callback(self, sock, selector):
        client_sock, client_addr = self.sock.accept()
        print("New connection on {}".format(client_addr))
        selector.add_fd(client_sock, READ, self.read_callback)
        selector.add_fd(client_sock, WRITE, self.write_callback)
        self.recv_data.setdefault(client_sock.fileno(), b"")
        self.send_data.setdefault(client_sock.fileno(), b"")
```

```
    def read_callback(self, client_sock, selector):
        data = client_sock.recv(BUFFER_SIZE)
        self.recv_data[client_sock.fileno()] += data
        result, _ = loads(self.recv_data[client_sock.fileno()])
        print("Received from {}: {}".format(client_sock.fileno(), result))
        self.send_data[client_sock.fileno()] += msg(
            {"result": "ok", "id": result["id"]})
        self.recv_data[client_sock.fileno()] = b""

    def write_callback(self, client_sock, selector):
        data = self.send_data[client_sock.fileno()]
        client_sock.sendall(data)
        self.send_data[client_sock.fileno()] = b""

    def cleanup_client(self, client_sock, selector):
        del self.recv_data[client_sock.fileno()]
        del self.send_data[client_sock.fileno()]
        selector.remove_fd(client_sock, READ)
        selector.remove_fd(client_sock, WRITE)
```

Example callback server.

```
class Server(object):

    def __init__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setblocking(0)
        self.sock.bind((socket.gethostname(), 0))
        self.address, self.port = self.sock.getsockname()
        self.recv_data = {}
        self.send_data = {}

    def register(self, selector):
        selector.add_fd(self.sock, READ, self.open_callback)
        print("Listening on {} {}".format(self.address, self.port))
        self.sock.listen(MAX_CLIENTS)

    def open_callback(self, sock, selector):
        client_sock, client_addr = self.sock.accept()
        print("New connection on {}".format(client_addr))
        selector.add_fd(client_sock, READ, self.read_callback)
        selector.add_fd(client_sock, WRITE, self.write_callback)
        self.recv_data.setdefault(client_sock.fileno(), b"")
        self.send_data.setdefault(client_sock.fileno(), b"")
```

```
    def read_callback(self, client_sock, selector):
        data = client_sock.recv(BUFFER_SIZE)
        self.recv_data[client_sock.fileno()] += data
        result, _ = loads(self.recv_data[client_sock.fileno()])
        print("Received from {}: {}".format(client_sock.fileno(), result))
        self.send_data[client_sock.fileno()] += msg(
            {"result": "ok", "id": result["id"]})
        self.recv_data[client_sock.fileno()] = b""

    def write_callback(self, client_sock, selector):
        data = self.send_data[client_sock.fileno()]
        client_sock.sendall(data)
        self.send_data[client_sock.fileno()] = b""

    def cleanup_client(self, client_sock, selector):
        del self.recv_data[client_sock.fileno()]
        del self.send_data[client_sock.fileno()]
        selector.remove_fd(client_sock, READ)
        selector.remove_fd(client_sock, WRITE)
```

Example callback server.

Callbacks are conceptually simple
but can create **unmaintainable** code,
lose stack context, **fail** silently, and are
often painful to **refactor**.

Callbacks

Iterators

Generators

Enhanced Generators

Subgenerator Delegation

Coroutines

Async Generators

Iterators are an important stepping stone on the way to **generators** (and beyond).

> [We propose] an **iteration** interface that objects can provide to control the behaviour of 'for' loops. Looping is customized by a method that produces an iterator object […] providing a '**get next value**' operation.

- PEP 234 (Jan 2001)

(iterator)

Returned with **`__iter__()`**

Iterate with **`x.next()`**

`next(x)` *for Py2.6+*

(iterator)

Iterators bring language features like

```
for x in b
```

```
[x for x in xs if x]
```

as well as helpers / idioms like

`map()` | **`filter()`** | **`“”.join()`**

However…

Iterators themselves do not solve the callback / concurrency problem.

(Meanwhile…)

Stackless Python introduced ~2000

Has coroutines, channels, etc.

Made everyone all jealous.

```
import stackless

ping_channel = stackless.channel()
pong_channel = stackless.channel()

def ping():
    while ping_channel.receive(): #blocks here
        print "PING"
        pong_channel.send("from ping")

def pong():
    while pong_channel.receive():
        print "PONG"
        ping_channel.send("from pong")

stackless.tasklet(ping)()
stackless.tasklet(pong)()
stackless.run()
```

Example Stackless tasks. (Grant Olson, 2006)

Callbacks
Iterators
Generators
Enhanced Generators
Subgenerator Delegation
Coroutines
Async Generators

> …provide a kind of function that can return an **intermediate result** ("the next value") to its caller, but maintaining the function's local state so that the function can be **resumed again** right where it left off.

- PEP 255 (May 2001)

(generator)

Uses **`yield`** in the body of the function

Re-entry after **`yield`** point

Callee must **`yield`** to caller

Python handles state + stack, not dev

(generator)

Can use generators to create coroutine-like workflows, using **trampolines** and **dispatchers**.

(Not as fun as normal trampolines.)

```
def accept(self, selector):
    print("Listening on {} {}".format(self.address, self.port))
    self.sock.listen(MAX_CLIENTS)
    while True:
        yield wait(selector, self.sock, READ)
        client_sock, client_addr = self.sock.accept()
        print("New connection on {}".format(client_addr))
        yield self.handle_client(selector, client_sock)
```

```
def main():
    server = Server()
    generators = [(None, server.accept()), (None, other())]
    while True:
        op, gen = generators.pop(0)
        if not op or not op.finished:
            generators.append((op, gen))
            continue
        op = gen.next() # StopIteration check
        generators.append((gen.next(), gen))
```

Example generator trampoline.

(Meanwhile…)

Twisted (2002) - deferred, networking

Greenlet / etc. emerge from Stackless

Callbacks
Iterators
Generators
Enhanced Generators
Subgenerator Delegation
Coroutines
Async Generators

> Python's generator functions are almost coroutines -- but not quite -- in that they allow **pausing execution** to produce a value, but do not provide for values or exceptions to be **passed in** when **execution resumes**.

- PEP 342 (2005)

So along came Python 2.5,
which gave generator objects
`send()`, **`throw()`**, and **`close()`**

(generator w/ send)

Introduces **`val = yield x`**

Caller is able to:

`gen.send(val)`

`gen.throw(exc)`

`gen.close()`

Lightweight coroutines are possible!

```python
@coroutine
def handle_client(self, selector, client_sock):
    while True:
        data = yield recv(selector, client_sock, BUFFER_SIZE)
        result, _ = loads(data)
        print("Received from client: {}".format(result))
        response = msg({"result": "ok", "id": result["id"]})
        yield sendall(selector, client_sock, response)
```

```python
@coroutine
def recv(selector, sock, numbytes):
    data = b""
    yield wait(selector, sock, READ)
    data = sock.recv(numbytes)
    yield result(data)


@coroutine
def sendall(selector, sock, data):
    yield wait(selector, sock, WRITE)
    sock.sendall(data)
    yield result(None)
```

Example generator with send

```python
@coroutine
def recv(selector, sock, numbytes):
    data = b""
    yield wait(selector, sock, READ)
    data = sock.recv(numbytes)
    yield result(data)


def wait(selector, fd, event):
    future = Future()

    def callback(fd, selector):
        selector.remove_fd(fd, event)
        future.set_result((fd, selector))

    selector.add_fd(fd, event, callback)
    return future
```

```python
def coroutine(fn):

    @functools.wraps(fn)
    def wrapped(*args, **kwargs):
        future = Future()
        context = {"gen": fn(*args, **kwargs)}

        def callback(result):
            try:
                result = context["future"].result()
            except Exception as exception:
                return context["gen"].throw(exception)
            try:
                context["future"] = context["gen"].send(result)
                context["future"].add_done_callback(callback)
            except StopIteration:
                future.set_result(result)

        # priming the pump
        context["future"] = context["gen"].send(None)
        context["future"].add_done_callback(callback)
        return future

    return wrapped
```

Example internals for generator with send

However…

Still constrained to caller-callee structure for yield control.

(Meanwhile…)

Mini-explosion of evented frameworks

Eventlet, gevent bring coroutines,
monkey-patching, etc.

(Also meanwhile…)

Python 3 is released! (2008)

Everyone immediately adopts.

Callbacks
Iterators
Generators
Enhanced Generators
Subgenerator Delegation
Coroutines
Async Generators

> A Python **generator** is a form of **coroutine**, but has the limitation that it can only yield to its immediate caller. […] A syntax is proposed for a generator to **delegate** part of its operations to another **generator**.

- PEP 380 (2011)

(yield from <subgenerator>)

Introduces **`val = yield from <gen>`**

Delegation to other coroutines

`return` from inside a generator

`Exceptions` from a saner context

Fewer trampolines!

(A rare positive in this case.)

(Meanwhile…)

asyncore, etc.

Competing, non-interop libraries

Node.js - callbacks revisited

Callbacks
Iterators
Generators
Enhanced Generators
Subgenerator Delegation
Coroutines
Async Generators

AsyncIO

> [The] current **lack of portability** between different async IO libraries causes a lot of **duplicated effort** for third party library developers. A sufficiently powerful abstraction could mean that asynchronous code gets written once, but **used everywhere**.

- PEP 3153 (2011)

> [A concrete proposal] which includes a **pluggable event loop**, transport and protocol abstractions similar to those in Twisted, and a higher-level scheduler based on **yield from**. The proposed package name is **asyncio**.

- PEP 3156 (2013)

(asyncio)

Originally a BDFL project (**tulip**)

A true **standard** lib for async Python

Familiar for Twisted / Tornado devs

Unifying direction for event-driven work

(asyncio)

BaseEventLoop <platform-specific>

Transport and protocol **separation**

Callback -> Future -> Coroutine

(it doesn't depend on gen)

(asyncio)

asyncio provides a variety of utilities
We can also combine (hybrid) libraries

So are projects like
Tornado / Twisted dead?
Absolutely **not**.

Asyncio and a standardized approach for building these libraries just makes them more valuable

(and now there are even *more* like aio, etc.)

Tasks, **Futures**, and **Loops**

(an asyncio aside)

An **Event Loop** *schedules*, *executes*, *continues*, and *cancels* coroutines*.

**which may be callbacks, generators, or first-class coroutines*

A **Future** represents an eventual result (or exception), used for async callbacks.

A **Task** is a subclass of Future, and schedules / tracks a single awaitable (coroutine).

```
class Server(object):

    def __init__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setblocking(0)
        self.sock.bind((socket.gethostname(), 0))
        self.address, self.port = self.sock.getsockname()

    @asyncio.coroutine
    def start(self, loop):
        print("Listening on {} {}".format(self.address, self.port))
        self.sock.listen(MAX_CLIENTS)
        while True:
            client_sock, client_addr = yield from loop.sock_accept(self.sock)
            loop.create_task(self.handle_client(loop, client_sock))

    @asyncio.coroutine
    def handle_client(self, loop, client_sock):
        while True:
            data = yield from loop.sock_recv(client_sock, BUFFER_SIZE)
            result, _ = loads(data)
            print("Received from client: {}".format(result))
            yield from loop.sock_sendall(
                client_sock, msg({"result": "ok", "id": result["id"]}))
```

```
def main():
    server = Server()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(server.start(loop))
```

Example sub generator delegation with asyncio

However…

Lots of nested decorators, runtime exceptions, lost futures, etc.

Callbacks
Iterators
Generators
Enhanced Generators
Subgenerator Delegation
Coroutines
Async Generators

> [We propose] to make coroutines a proper **standalone concept** in Python. The ultimate goal is to help establish a common, easily approachable, mental model of **asynchronous programming** in Python and make it as close to synchronous programming as possible.
>
> - PEP 0492 (2015)

> We believe [this] will help keep Python **relevant** and **competitive** in a quickly growing area of **asynchronous** programming, as many other languages have adopted, or are planning to adopt, similar features.

- PEP 0492 (2015)

(coroutines)

Available in Python 3.5

Native coroutine type(s)

Finally, unique from generators

New keywords - explicit and intuitive

(in my humble opinion)

(coroutines)

- `await` & `async`
- types.coroutine
- `__await__`,
- `async for (__aiter__, __anext__)`,
- `async with (__aenter__, __aexit__)`

(coroutines)

Get rid of all those decorators
Exceptions make (even more) sense
Syntax errors for mismatches
Runtime warnings for unused futures

(and a lot more stuff)

```
class Server(object):

    def __init__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setblocking(0)
        self.sock.bind((socket.gethostname(), 0))
        self.address, self.port = self.sock.getsockname()

    async def start(self, loop):
        print("Listening on {} {}".format(self.address, self.port))
        self.sock.listen(MAX_CLIENTS)
        while True:
            client_sock, client_addr = await loop.sock_accept(self.sock)
            loop.create_task(self.handle_client(loop, client_sock))

    async def handle_client(self, loop, client_sock):
        while True:
            data = await loop.sock_recv(client_sock, BUFFER_SIZE)
            result, _ = loads(data)
            print("Received from client: {}".format(result))
            await loop.sock_sendall(
                client_sock, msg({"result": "ok", "id": result["id"]}))
```

Example coroutines with asyncio

```
class Server(object):

    def __init__(self, address, port):
        self.address = address
        self.port = port

    async def start(self, loop):
        print("Listening on {} {}".format(self.address, self.port))
        await asyncio.start_server(self.handle_client, self.address, self.port)

    async def handle_client(self, reader, writer):
        while True:
            data = await reader.read(BUFFER_SIZE)
            result, _ = loads(data)
            print("Received from client: {}".format(result))
            await writer.write(msg({"result": "ok", "id": result["id"]}))
```

Example coroutines with asyncio

```python
class Server(object):

    def __init__(self, address, port):
        self.address = address
        self.port = port

    async def start(self, loop):
        print("Listening on {} {}".format(self.address, self.port))
        await asyncio.start_server(self.handle_client, self.address, s

    async def handle_client(self, reader, writer):
        while True:
            data = await reader.read(BUFFER_SIZE)
            result, _ = loads(data)
            print("Received from client: {}".format(result))
            await writer.write(msg({"result": "ok", "id": result["id"]}))
```

```python
class Server(object):

    def __init__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.bind((socket.gethostname(), 0))
        self.address, self.port = self.sock.getsockname()
        self.sock.listen(MAX_CLIENTS)

    def wait(self):
        client_sock, client_addr = self.sock.accept()
        while True:
            data = client_sock.recv(BUFFER_SIZE)
            result, _ = loads(data)
            print("Received {}".format(result))
            client_sock.sendall(msg({"result": "ok", "id": result["id"]}))
```

Comparison with blocking “noscale” service

Parallel read / write coroutines for each asyncio server client

```
class Server(object):

    def __init__(self, address, port):
        self.address = address
        self.port = port

    async def start(self, loop):
        print("Listening on {} {}".format(self.address, self.port))
        await asyncio.start_server(
            partial(self.handle_client, loop), self.address, self.port)

    async def watch_reads(self, reader, outq):
        while True:
            data = await reader.read(BUFFER_SIZE)
            result, _ = loads(data)
            print("Received from client: {}".format(result))
            await outq.put(msg({"result": "ok", "id": result["id"]}))

    async def watch_writes(self, writer, outq):
        while True:
            message = await outq.get()
            writer.write(message)

    async def handle_client(self, loop, reader, writer):
        outq = asyncio.Queue()
        f1 = loop.create_task(self.watch_reads(reader, outq))
        f2 = loop.create_task(self.watch_writes(writer, outq))
        await asyncio.gather(f1, f2)
```

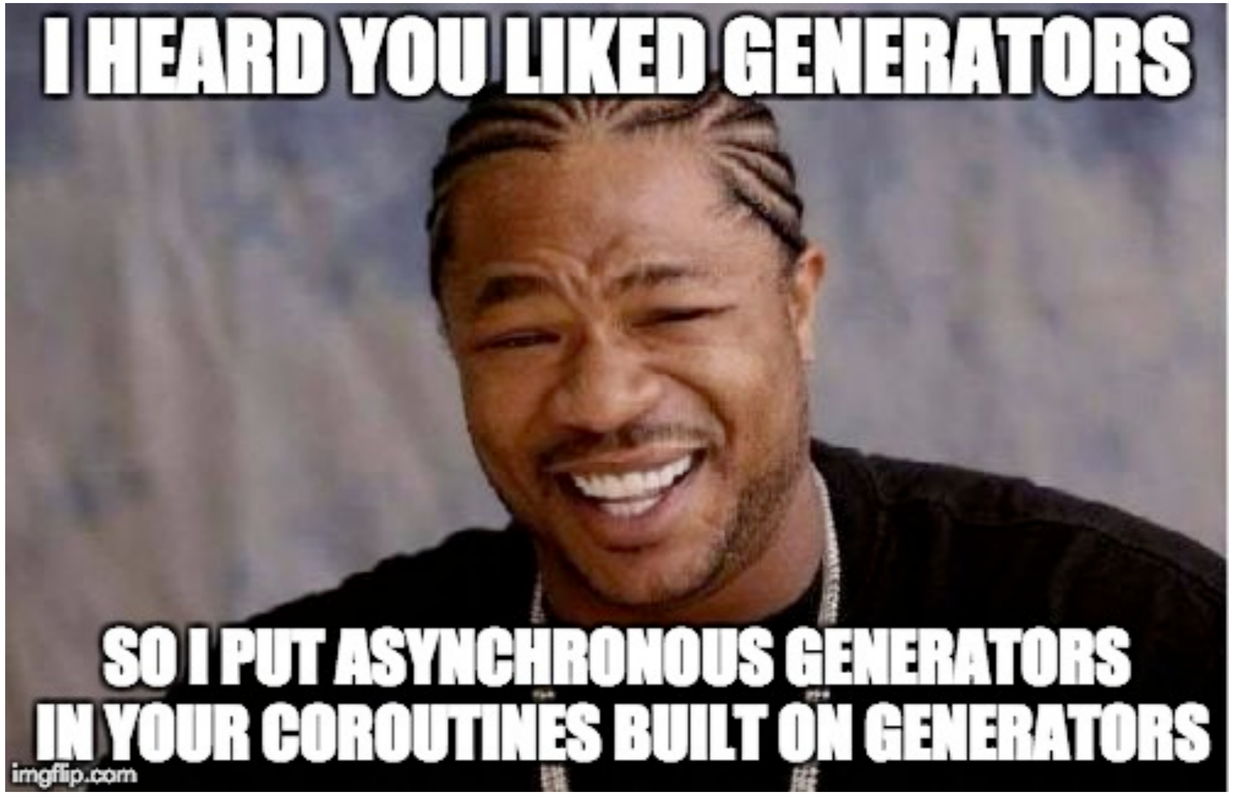
```
async def watch_reads(self, reader, outq):
    while True:
        data = await reader.read(BUFFER_SIZE)
        result, _ = loads(data)
        print("Received from client: {}".format(result))
        await outq.put(msg({"result": "ok", "id": result["id"]}))

async def watch_writes(self, writer, outq):
    while True:
        message = await outq.get()
        writer.write(message)
```

Parallel read / write coroutines for each asyncio server client

Callbacks
Iterators
Generators
Enhanced Generators
Subgenerator Delegation
Coroutines
Async Generators

> [...] Currently there is no equivalent concept for the **asynchronous iteration protocol**. Essentially, the goals and rationale for PEP 255, applied to the **asynchronous** execution case, hold true for this **proposal** as well.

- PEP 0525 (2016)


I HEARD YOU LIKED GENERATORS
SO I PUT ASYNCHRONOUS GENERATORS
IN YOUR COROUTINES BUILT ON GENERATORS
imgflip.com

(asynchronous generators)

`yield` inside `async def` in Python 3.6.

`aiter()` and `anext()` builtins

`.asend()`, `.throw()`, `.aclose()`

`await`, `yield`, and `return`

Closing **Thoughts**

(gotchas)

Support **explicit** loops in signatures.
Make sure you handle **result()**.
async + await > yield from > callbacks
Wrap those hybrid **futures**!
Standardize on an async testing framework
Python **3.6** isn't everywhere
(Docker is your friend)

(other implementations and concepts)

C - setjmp, longjmp

C++ - Boost coroutines

Node - promises, soon async / await

Goroutines - channels, green threads

Ruby - fibers

Continuations, actor model

(some references / links)

David Beazley's Coroutine Guide

http://www.dabeaz.com/coroutines/

AsyncIO Libraries

https://github.com/python/asyncio/wiki/ThirdParty

Coroutines in C

http://bit.ly/2azh52u | http://bit.ly/2ayfCcW

Concurrency in Python

https://blog.gevent.org/2010/02/27/why-gevent/

Guido Explaining Twisted's Deferred

http://bit.ly/2zW6wiT