Await Thy Async

A Brief History of Coroutines in Python

Hello.

I’m Josh Marshall.
I work at uStudio.
I like event-driven things.

(what are we talking about)

Why should I care / **so what**?

What are **coroutines**?

How have they evolved in **Python**?

What **tools** are out there?

(caveats)

My experience is in frameworks.

(Specifically Tornado.)

Super informal. Mistakes ahead.

I don’t mind corrections!

(survey)

Who knows what a **coroutine** is?
Who has used coroutines in **Python**?
Who is using **Python 3**?
Who is using **Node** *(or similar)*?

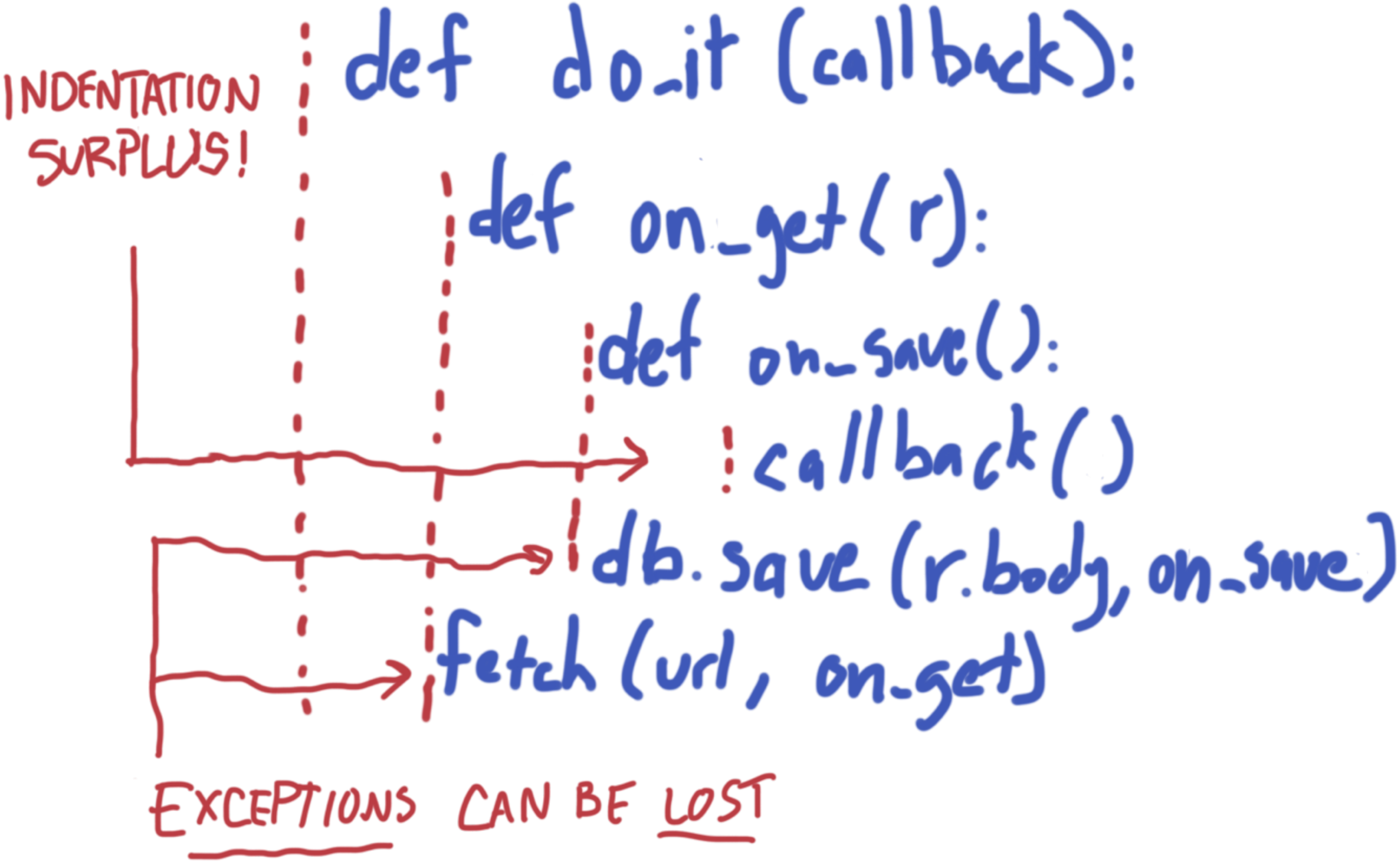
First, what are we trying to **solve**?

C10K (C1M?)

Lots of waiting on **I/O bound** processes

Thread complexity when unnecessary

Callback hell is not great



INDENTATION SURPLUS!

```
def do_it (callback):
    def on_get(r):
        def on_save():
            callback()
        db.save(r.body, on_save)
    fetch(url, on_get)
```

EXCEPTIONS CAN BE LOST

So, what is a coroutine?

Coroutines provide a method of cooperative multitasking.

Coroutines allow delegation,
re-entry, and value passing.

With asynchronous libraries, coroutines simplify evented code and help reduce bugs.

(other implementations)

Goroutines - channels, green threads

Haskell - arb. suspend and resume

C - setjmp, longjmp, assembly

Ruby - fibers (continuations)

Node - promises, soon async / await

“ Subroutines are special cases of … coroutines.

- Donald Knuth

Cooperative (vs preemptive)

Concurrent (vs parallel)

Explicit* (vs implicit)

**sometimes*

How We (Python) Got Here

We’re going to start with **iterators** and **generators**.

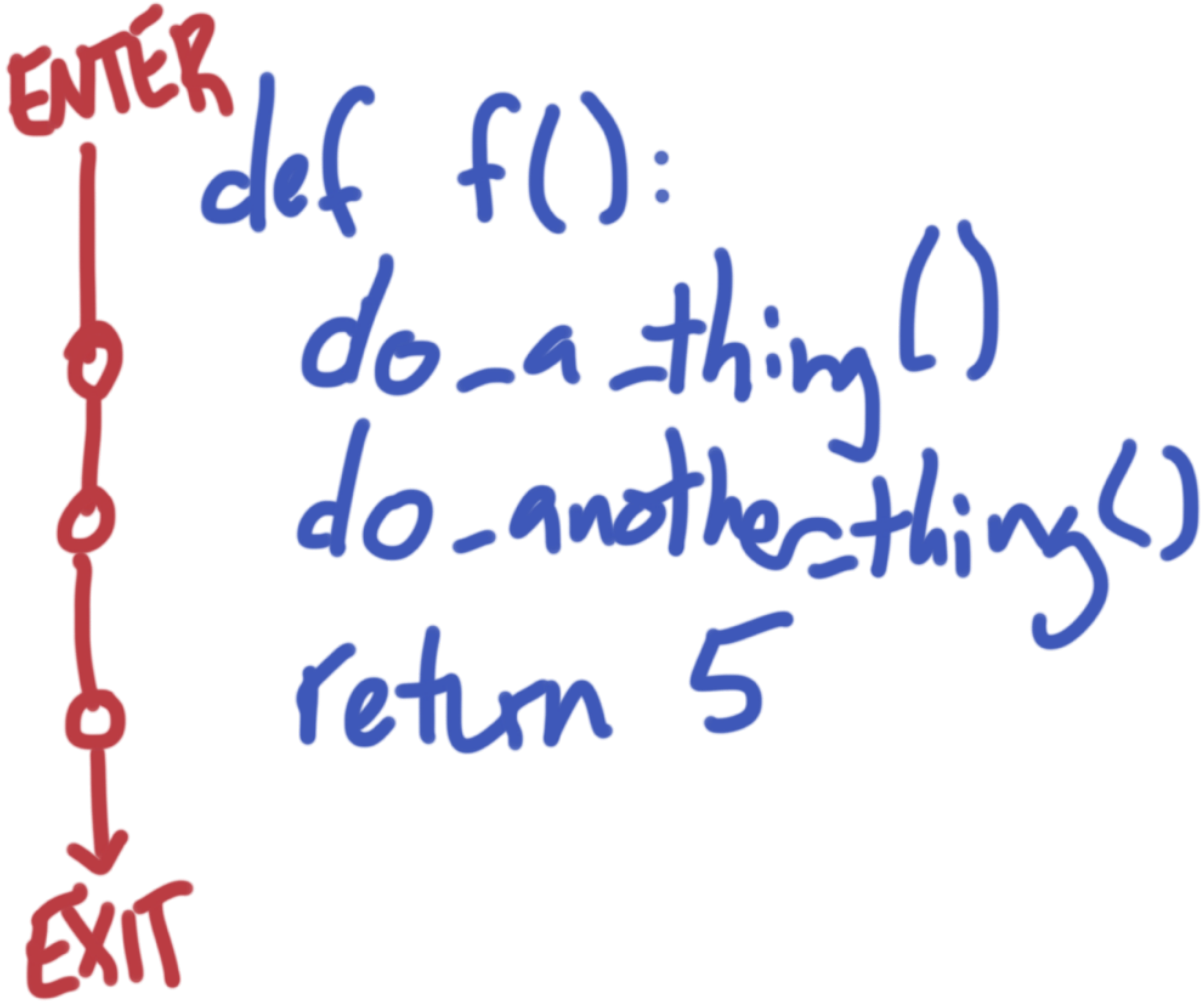
Iterators != Coroutines

Generators != Coroutines

…but they share some qualities.

We start the tale with…

Plain, unyieldable blocking calls.



```
def f():
    do_a_thing()
    do_another_thing()
    return 5
```

> “[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 Py3+*

(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()`**

Example!

(this will be referenced later)

However…

Writing complex iterators was difficult,
state management was wonky, etc.

(Meanwhile…)

Stackless Python introduced ~2000

Has microthreads, channels, etc.

Made everyone all jealous.

> “…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)

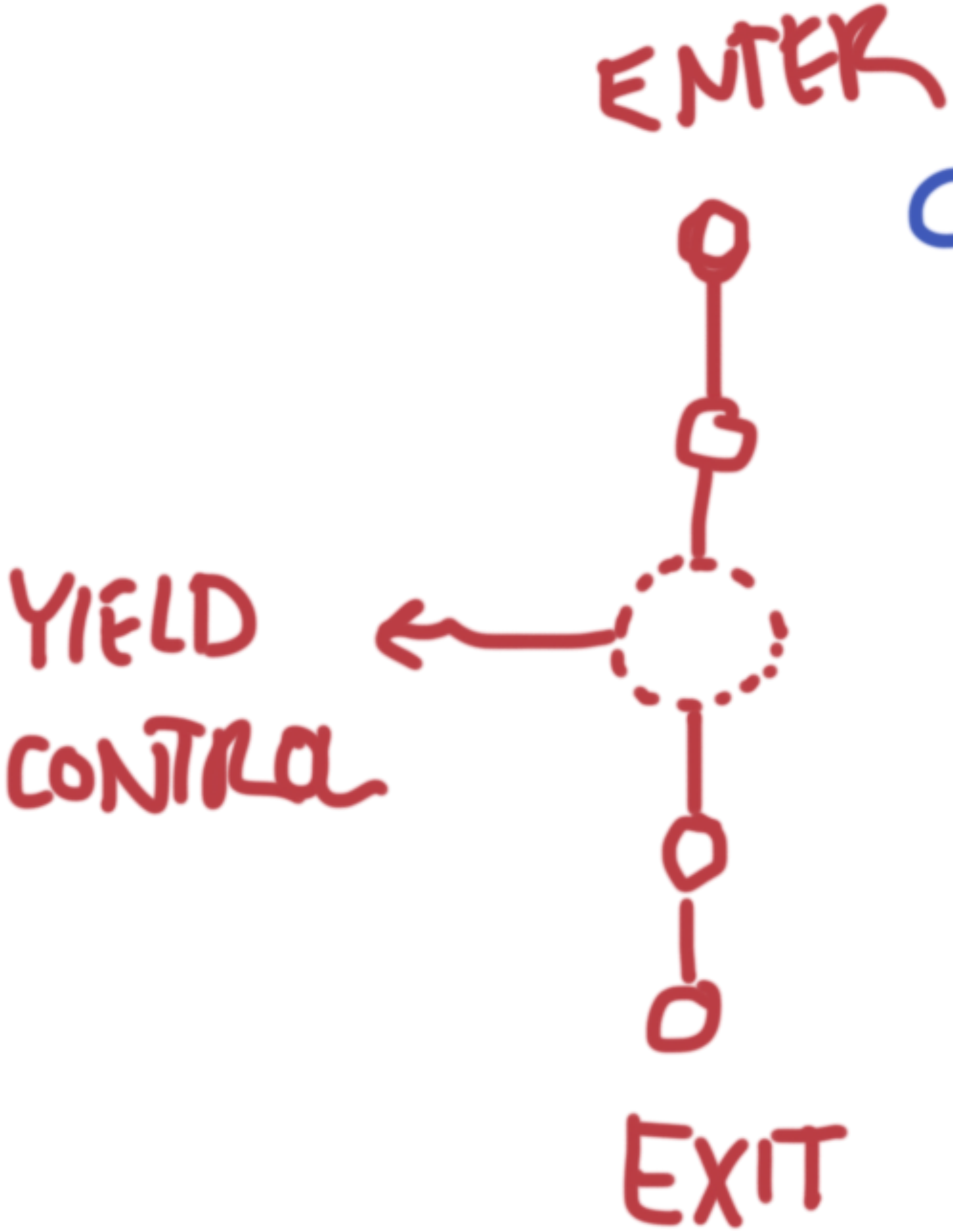
(generator)

Uses `yield` in the body of the function

Re-entry after `yield` point

Callee must `yield` to caller

Python handles state + stack, not dev



```
def f():
    do_a_thing()
    wait_on_a_thing()
    do_another_thing()
    return 42
```

Example!

(generator)

People used generators to create coroutine-like workflows, using **trampolines** and **dispatchers**.

(It’s not great fun.)

Example!

(Meanwhile…)

Twisted (2002) - deferred, networking

Greenlet / etc. emerge from Stackless

> “ 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 generators **`send()`**

(generator w/ send)

Introduces `val = yield x`

Caller is able to:

```
gen.send(val)
gen.throw(exc)
gen.close()
```

Lightweight coroutines are possible!

Example!

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.

We are all using it today.

> “ 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)

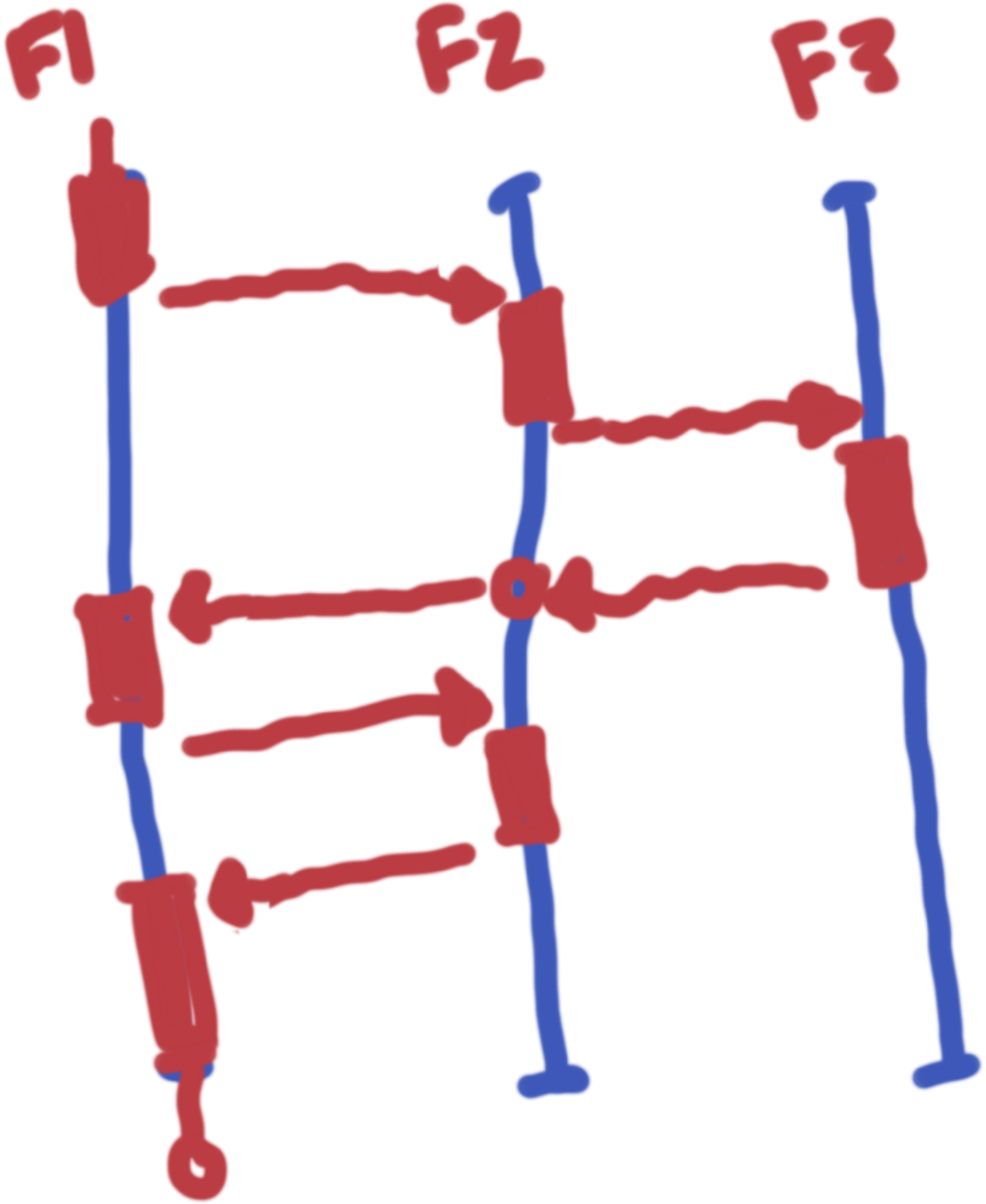
(yield from <subgenerator>)

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

Delegation to other coroutines!

Less trampolines!

(A rare positive in this case.)


F1
F2
F3

Example!

(Meanwhile…)

asyncore, Twisted, Tornado, gevent

Competing, non-interop libraries

Node.js - callbacks are okay now?

> “[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)

A true **standard** lib for async Python

Familiar for Twisted / Tornado devs

Unifying direction for event-driven work

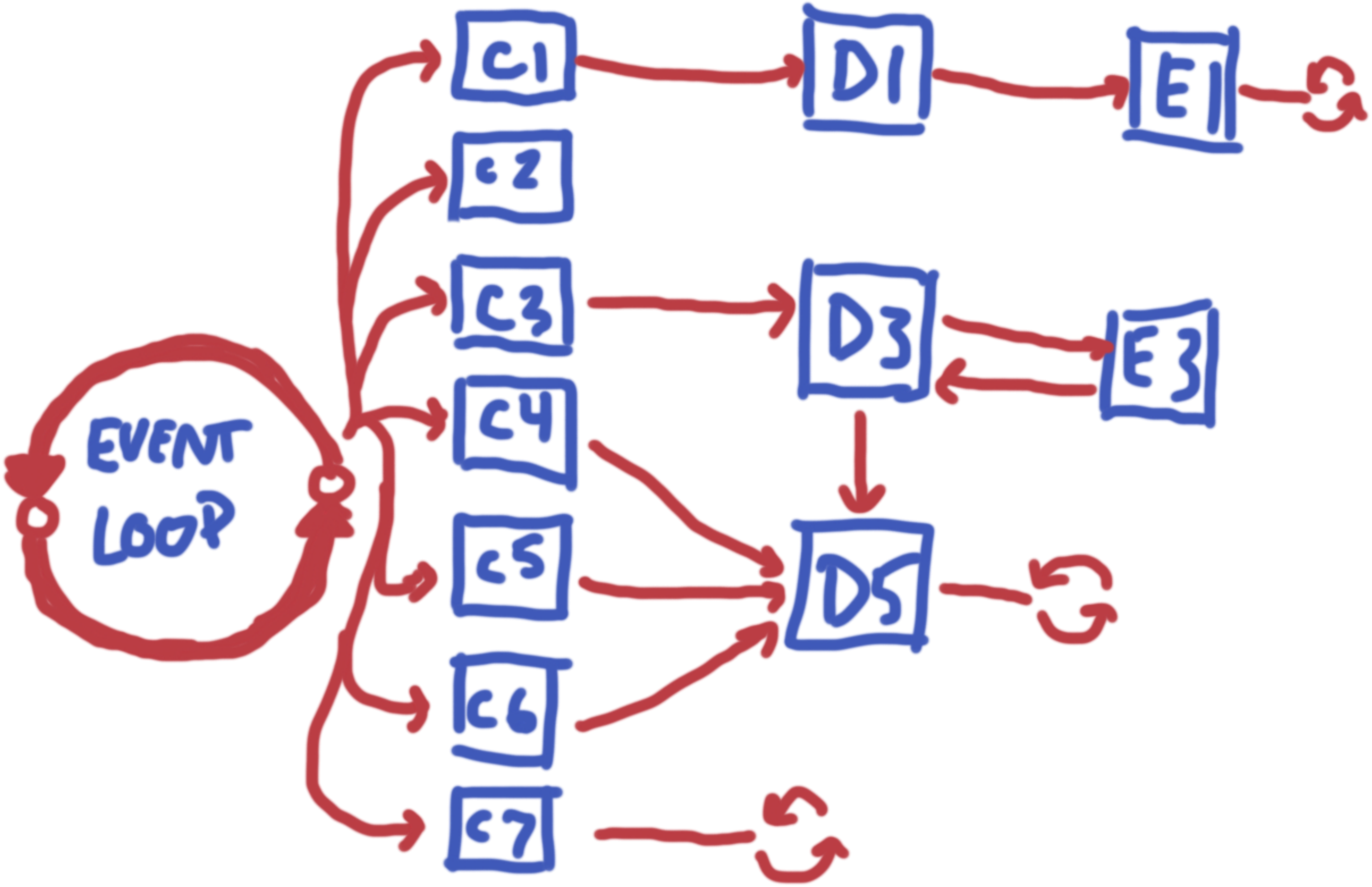
(asyncio)

BaseEventLoop <platform-specific>

Transport and protocol **separation**

Callback -> Future -> Coroutine

(it doesn't depend on gen)


EVENT LOOP
C1
C2
C3
C4
C5
C6
C7
D1
D3
D5
E1
E3

Examples!

(asyncio)

asyncio provides a variety of utilities

We can also combine (hybrid) libraries

A few more examples ahead

(I also recommend you dig in yourself)

However…

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

> “[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)

(coroutines)

Native coroutine type(s)

Finally, unique from generators

New keywords - explicit and intuitive

(in my humble opinion)

COROUTINES!

```
async def fetch(url):
    response = await get(url)
    return response.code
```

CONTEXTS!

```
async with db.connect() as session:
```

FOR LOOPS!

```
    async for record in session.find():
        await record.update(foo="bar")
```

WAITING ON STUFF!

Examples!

(asyncio)

Let’s combine some stuff.

Terminal stdin with asyncio

Binary streaming to stdout

Command line tool for beeps

Examples!

So is Tornado / Twisted dead?

Absolutely **not**.

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

Asyncio **doesn’t** provide:

- HTTP clients and server frameworks
- Asynchronous database drivers
- Popular wire protocols for TCP and UDP
- Testing helpers to isolate business logic
- Etc.

Asyncio was created with the explicit goal of encouraging library interop.

You’re going to need all the libraries you can get.

(other 3.x benefits)

Working with strings / unicode / binary
Type annotations for great success(?)
Working SSL, sane(r) libraries
Wheels / packaging improvements

Let’s talk about testing!
(I’ll walk through examples.)

Tasks, Futures, and Loops

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).

Examples!

(gotchas)

Support **explicit** loops.

Make sure you handle **result()**.

async + await > yield from > callbacks

Wrap those hybrid **futures**!

Probably standardize on testing framework

Python **3.5**(.2) isn’t everywhere

Python deployment is *sooo* easy!

(some references / links)

David Beazley’s Coroutine Guide

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

Coroutines on Wikipedia

https://en.wikipedia.org/wiki/Coroutine

Coroutines in C

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

Concurrency in Python

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