Generators: The Final Frontier

David Beazley (@dabeaz)
http://www.dabeaz.com

Presented at PyCon'2014, Montreal

Previously on Generators

- *Generator Tricks for Systems Programmers* (2008)

  http://www.dabeaz.com/generators/

- A flying leap into generator awesomeness

Previously on Generators

- *A Curious Course on Coroutines and Concurrency (2009)*

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

- Wait, wait? There's more than iteration?

Today's Installment

- <u>Everything else</u> you ever wanted to know about generators, but were afraid to try
- Part 3 of a trilogy

Requirements

- You need Python 3.4 or newer
- No third party extensions
- Code samples and notes

    http://www.dabeaz.com/finalgenerator/

- Follow along if you dare!

Disclaimer

- This is an advanced tutorial
- Assumes general awareness of
    - Core Python language features
    - Iterators/generators
    - Decorators
    - Common programming patterns
- I learned a <u>LOT</u> preparing this

Will I Be Lost?

- Although this is the third part of a series, it's mostly a stand-alone tutorial
- If you've seen prior tutorials, that's great
- If not, don't sweat it
- Be aware that we're focused on a specific use of generators (you just won't get complete picture)

Focus

practical utility

bleeding edge

- Material in this tutorial is probably not immediately applicable to your day job
- More thought provoking and mind expanding
- from __future__ import future

Part I

Preliminaries - Generators and Coroutines

(rock)

Generators 101

- yield statement defines a generator function

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

- You typically use it to feed iteration

```
for x in countdown(10):
    print('T-minus', x)
```

- A simple, yet elegant idea

Under the Covers

- Generator object runs in response to next()

```
>>> c = countdown(3)
>>> c
<generator object countdown at 0x10064f900>
>>> next(c)
3
>>> next(c)
2
>>> next(c)
1
>>> next(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>>
```

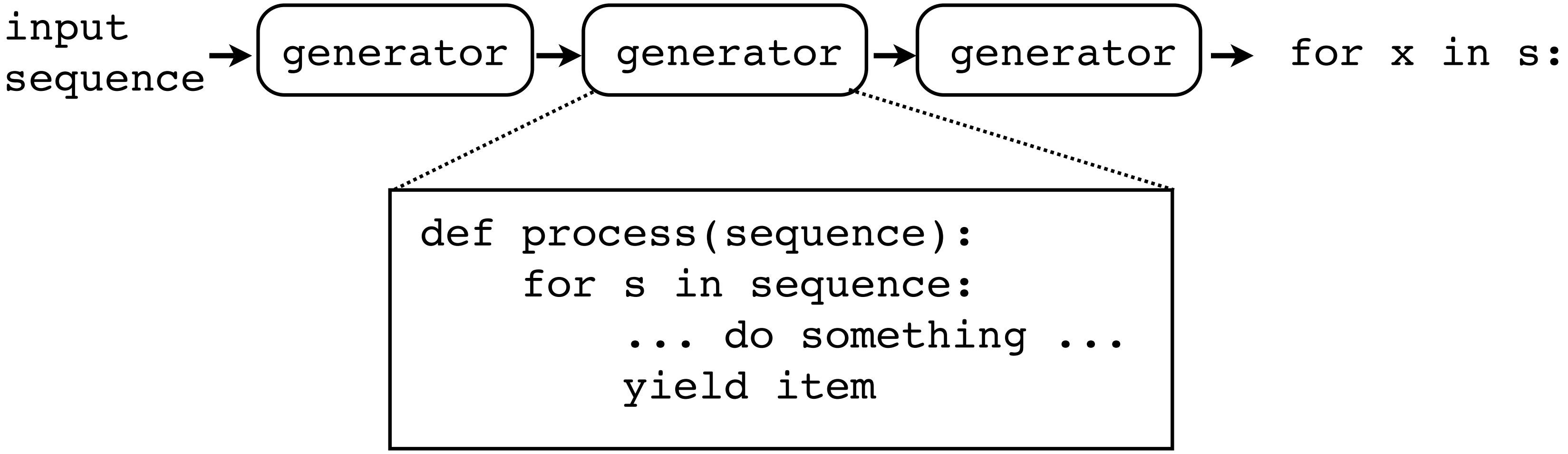
- StopIteration raised when function returns

Interlude

- Generators as "iterators" misses the big picture
- There is so much more to yield

Generators as Pipelines

- Stacked generators result in processing pipelines
- Similar to shell pipes in Unix



```
def process(sequence):
    for s in sequence:
        ... do something ...
        yield item
```

- Incredibly useful (see prior tutorial)

Coroutines 101

- yield can receive a value instead

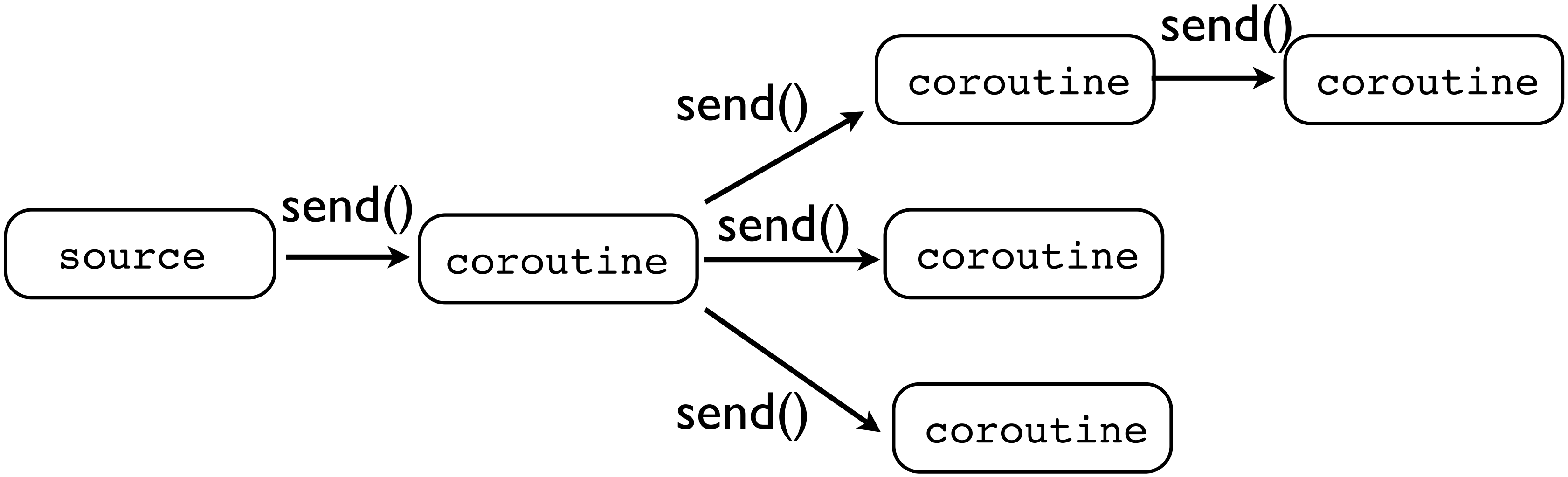
```
def receiver():
    while True:
        item = yield
        print('Got', item)
```

- It defines a generator that you send things to

```
recv = receiver()
next(recv)              # Advance to first yield
recv.send('Hello')
recv.send('World')
```

Coroutines and Dataflow

- Coroutines enable dataflow style processing



- Publish/subscribe, event simulation, etc.

Fundamentals

- The yield statement defines a generator function

```
def generator():
    ...
    ... yield ...
    ...
```

- The mere presence of yield anywhere is enough
- Calling the function creates a generator instance

```
>>> g = generator()
>>> g
<generator object generator at 0x10064f120>
>>>
```

Advancing a Generator

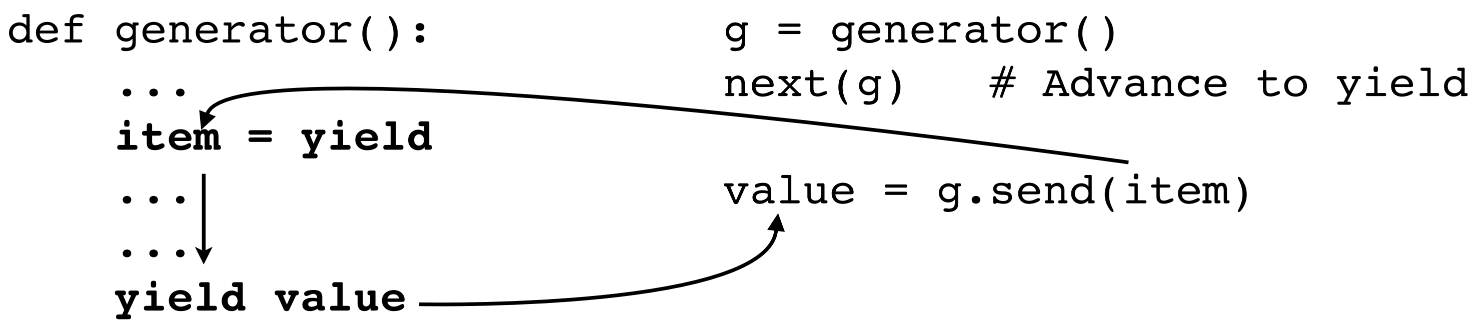
- next(gen) - Advances to the next yield

```
def generator():
    ...
    ...
    yield item
    ...
```

- Returns the yielded item (if any)
- It's the <u>only allowed operation</u> on a newly created generator
- Note: Same as gen.__next__()

Sending to a Generator

- gen.send(item) - Send an item to a generator



```
def generator():           g = generator()
    ...                    next(g)   # Advance to yield
    item = yield
    ...                    value = g.send(item)
    ...
    yield value
```

- Wakes at last yield, returns sent value
- Runs to the next yield and emits the value

Closing a Generator

- gen.close() - Terminate a generator

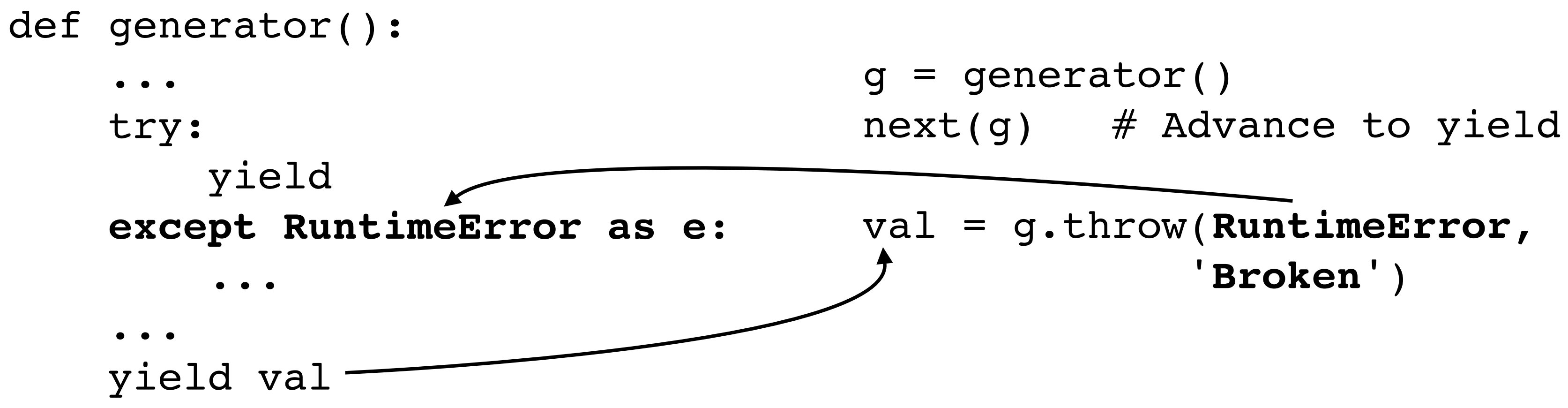
```
def generator():
    ...
    try:
        yield
    except GeneratorExit:
        # Shutting down
        ...
```

```
g = generator()
next(g)   # Advance to yield

g.close() # Terminate
```

- Raises GeneratorExit at the yield
- Only allowed action is to return
- If uncaught, generator silently terminates

Raising Exceptions

- gen.throw(typ [, val [,tb]]) - Throw exception



```
def generator():
    ...                           g = generator()
    try:                          next(g)   # Advance to yield
        yield
    except RuntimeError as e:     val = g.throw(RuntimeError,
        ...                                    'Broken')
    ...
    yield val
```

- Raises exception at yield
- Returns the next yielded value (if any)

Generator Return Values

- StopIteration raised on generator exit

```
def generator():
    ...
    yield
    ...
    return result
```

```
g = generator()
try:
    next(g)
except StopIteration as e:
    result = e.value
```

- Return value (if any) passed with exception
- Note: Python 3 only behavior (in Python 2, generators can't return values)

Generator Delegation

- yield from gen - Delegate to a subgenerator

```
def generator():
    ...
    yield value
    ...
    return result

def func():
    result = yield from generator()
```

- Allows generators to call other generators
- Operations take place at the current yield
- Return value (if any) is returned

Delegation Example

- Chain iterables together

```
def chain(x, y):
    yield from x
    yield from y
```

- Example:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> for x in chain(a, b):
...     print(x,end=' ')
...
1 2 3 4 5 6

>>> c = [7,8,9]
>>> for x in chain(a, chain(b, c)):
...     print(x, end=' ')
...
1 2 3 4 5 6 7 8 9
>>>
```

Mini-Reference

- Generator definition

```
def generator():
    ...
    yield
    ...
    return result
```

- Generator instance operations

```
gen = generator()

next(gen)                      # Advance to next yield
gen.send(item)                 # Send an item
gen.close()                    # Terminate
gen.throw(exc, val, tb)        # Raise exception
result = yield from gen        # Delegate
```

- Using these, you can do a lot of neat stuff

Part 2

and now for something completely different

A Common Motif

- Consider the following

```
f = open()
...
f.close()
```

……

```
lock.acquire()
...
lock.release()
```

……

```
db.start_transaction()
...
db.commit()
```

……

```
start = time.time()
...
end = time.time()
```

- It's so common, you'll see it everywhere!

Context Managers

- The 'with' statement

```
with open(filename) as f:
     statement
     statement
     ...

with lock:
     statement
     statement
     ...
```

- Allows control over entry/exit of a code block
- Typical use: everything on the previous slide

Context Management

- It's easy to make your own (@contextmanager)

```
import time
from contextlib import contextmanager

@contextmanager
def timethis(label):
    start = time.time()
    try:
        yield
    finally:
        end = time.time()
        print('%s: %0.3f' % (label, end-start)
```

- This times a block of statements

Context Management

- Usage

```
with timethis('counting'):
    n = 1000000
    while n > 0:
        n -= 1
```

- Output

```
counting: 0.023
```

Context Management

- Another example: temporary directories

```
import tempfile, shutil
from contextlib import contextmanager

@contextmanager
def tempdir():
    outdir = tempfile.mkdtemp()
    try:
        yield outdir
    finally:
        shutil.rmtree(outdir)
```

- Example

```
with tempdir() as dirname:
    ...
```

Whoa, Whoa, Stop!

- Another example: temporary directories

```
import tempfile, shutil
from contextlib import contextmanager

@contextmanager
def tempdir():
    outdir = tempfile.mkdtemp()
    try:
        yield outdir
    finally:
        shutil.rmtree(outdir)
```

What is this?

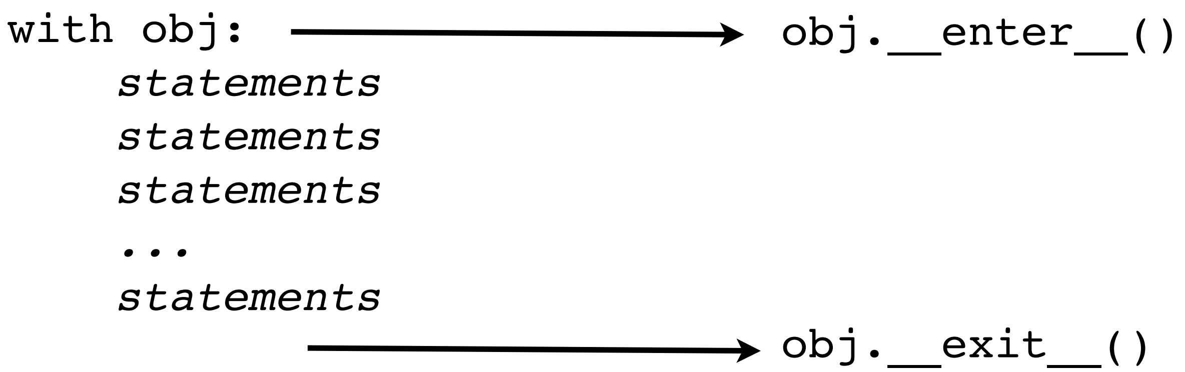
- Not iteration
- Not dataflow
- Not concurrency
- ????

- Example

```
with tempdir() as dirname:
    ...
```

Context Management

- Under the covers



```
with obj:  ──────────────→  obj.__enter__()
    statements
    statements
    statements
    ...
    statements
             ──────────────→  obj.__exit__()
```

- If an object implements these methods it can monitor entry/exit to the code block

Context Manager

- Implementation template

```
class Manager(object):
    def __enter__(self):
        return value
    def __exit__(self, exc_type, val, tb):
        if exc_type is None:
            return
        else:
            # Handle an exception (if you want)
            return True if handled else False
```

- Use:

```
with Manager() as value:
    statements
    statements
```

Context Manager Example

- Automatically deleted temp directories

```
import tempfile
import shutil

class tempdir(object):
    def __enter__(self):
        self.dirname = tempfile.mkdtemp()
        return self.dirname

    def __exit__(self, exc, val, tb):
        shutil.rmtree(self.dirname)
```

- Use:

```
with tempdir() as dirname:
     ...
```

Alternate Formulation

- @contextmanager is just a reformulation

```
import tempfile, shutil
from contextlib import contextmanager

@contextmanager
def tempdir():
    dirname = tempfile.mkdtemp()
    try:
        yield dirname
    finally:
        shutil.rmtree(dirname)
```

- It's the same code, glued together differently

Deconstruction

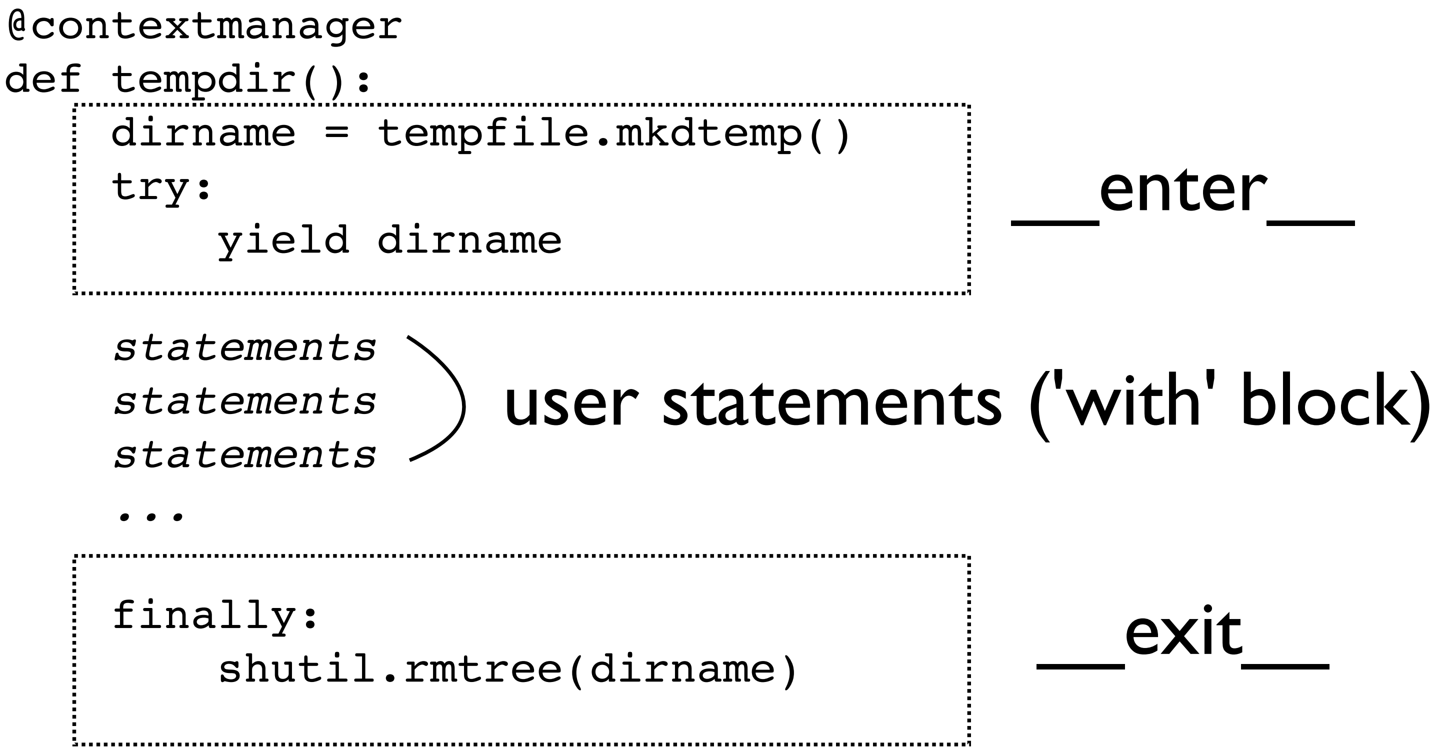
- How does it work?

```
@contextmanager
def tempdir():
    dirname = tempfile.mkdtemp()
    try:
        yield dirname
    finally:
        shutil.rmtree(dirname)
```

- Think of "yield" as scissors
- Cuts the function in half

Deconstruction

- Each half maps to context manager methods



```
@contextmanager
def tempdir():
    dirname = tempfile.mkdtemp()
    try:
        yield dirname

    statements
    statements
    statements
    ...

    finally:
        shutil.rmtree(dirname)
```

__enter__

user statements ('with' block)

__exit__

- yield is the magic that makes it possible

Deconstruction

- There is a wrapper class (Context Manager)

```
class GeneratorCM(object):
     def __init__(self, gen):
         self.gen = gen

     def __enter__(self):
        ...

     def __exit__(self, exc, val, tb):
        ...
```

- And a decorator

```
def contextmanager(func):
    def run(*args, **kwargs):
        return GeneratorCM(func(*args, **kwargs))
    return run
```

Deconstruction

- enter - Run the generator to the yield

```
class GeneratorCM(object):
     def __init__(self, gen):
         self.gen = gen

     def enter__(self):
         return next(self.gen)

     def __exit__(self, exc, val, tb):
        ...
```

- It runs a single "iteration" step
- Returns the yielded value (if any)

Deconstruction

- exit - Resumes the generator

```
class GeneratorCM(object):
    ...
    def __exit__(self, etype, val, tb):
        try:
            if etype is None:
                next(self.gen)
            else:
                self.gen.throw(etype, val, tb)
            raise RuntimeError("Generator didn't stop")
        except StopIteration:
            return True
        except:
            if sys.exc_info()[1] is not val: raise
```

- Either resumes it normally or raises exception

Full Disclosure

- Actual implementation is more complicated
- There are some nasty corner cases
    - Exceptions with no associated value
    - StopIteration raised inside a with-block
    - Exceptions raised in context manager
- Read source and see PEP-343

Discussion

- Why start with this example?
- A completely different use of yield
- Being used to reformulate control-flow
- It simplifies programming for others (easy definition of context managers)
- Maybe there's more... (of course there is)

Part 3

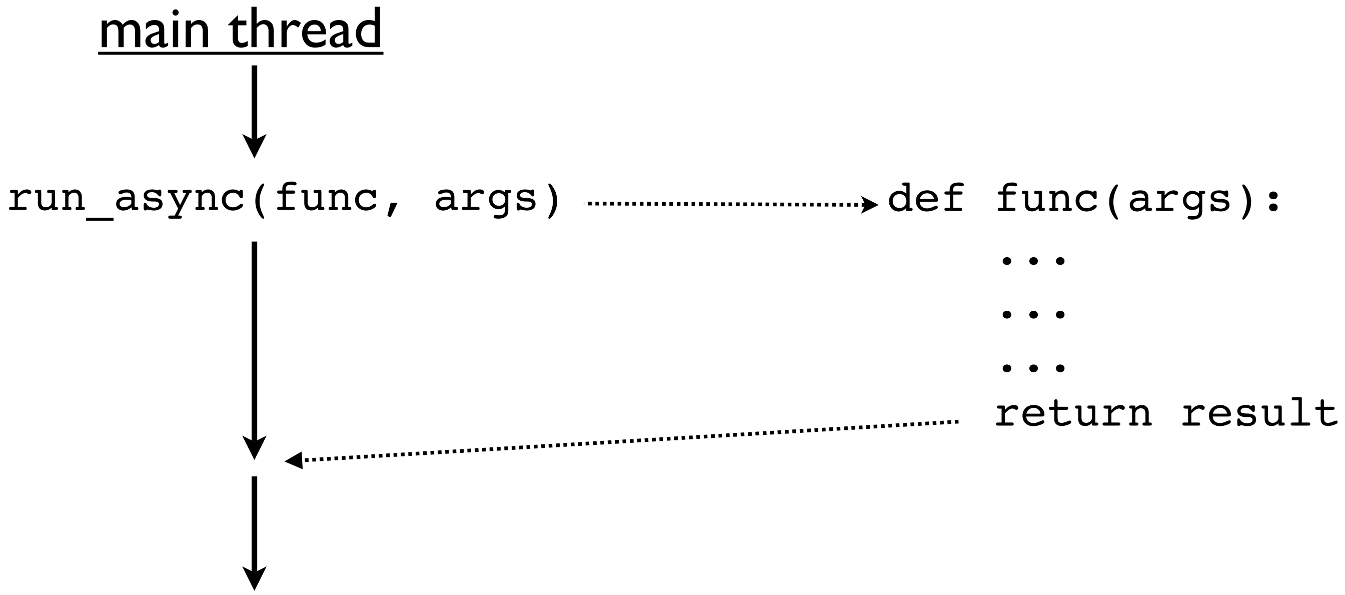
Call me, maybe

Part 3

Call me, maybe

Async Processing

- Consider the following execution model



main thread

```
run_async(func, args)         def func(args):
                                  ...
                                  ...
                                  ...
                                  return result
```

- Examples: Run in separate process or thread, time delay, in response to event, etc.

Example: Thread Pool

```
from concurrent.futures import ThreadPoolExecutor

def func(x, y):
    'Some function. Nothing too interesting'
    import time
    time.sleep(5)
    return x + y

pool = ThreadPoolExecutor(max_workers=8)
fut = pool.submit(func, 2, 3)
r = fut.result()
print('Got:', r)
```

- Runs the function in a separate thread
- Waits for a result

Futures

- Future - A result to be computed later

```
>>> fut = pool.submit(func, 2, 3)
>>> fut
<Future at 0x1011e6cf8 state=running>
>>>
```

- You can wait for the result to return

```
>>> fut.result()
5
>>>
```

- However, this blocks the caller

Futures

- Alternatively, you can register a callback

```
def run():
    fut = pool.submit(func, 2, 3)
    fut.add_done_callback(result_handler)

def result_handler(fut):
    result = fut.result()
    print('Got:', result)
```

- Triggered upon completion

Exceptions

```
>>> fut = pool.submit(func, 2, 'Hello')
>>> fut.result()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.4/concurrent/futures/_base.py",
line 395, in result
    return self.__get_result()
  File "/usr/local/lib/python3.4/concurrent/futures/_base.py",
line 354, in __get_result
    raise self._exception
  File "/usr/local/lib/python3.4/concurrent/futures/thread.py",
line 54, in run
    result = self.fn(*self.args, **self.kwargs)
  File "future2.py", line 6, in func
    return x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Futures w/Errors

- Error handling with callbacks

```
def run():
    fut = pool.submit(func, 2, 3)
    fut.add_done_callback(result_handler)

def result_handler(fut):
    try:
        result = fut.result()
        print('Got:', result)
    except Exception as e:
        print('Failed: %s: %s' % (type(e).__name__, e))
```

- Exception propagates out of fut.result() method

Interlude

- Consider the structure of code using futures

```
def run():
    fut = pool.submit(func, 2, 3)
    fut.add_done_callback(result_handler)

def result_handler(fut):
    try:
        result = fut.result()
        print('Got:', result)
    except Exception as e:
        print('Failed: %s: %s' % (type(e).__name__, e))
```

- Meditate on it... focus on the code.
- This seems sort of familiar

Callback Hell?

- No, no, no.... keep focusing.

Interlude

- What if the function names are changed?

```
def entry():
    fut = pool.submit(func, 2, 3)
    fut.add_done_callback(exit)

def exit(fut):
    try:
        result = fut.result()
        print('Got:', result)
    except Exception as e:
        print('Failed: %s: %s' % (type(e).__name__, e))
```

- Wait! This is almost a context manager (yes)

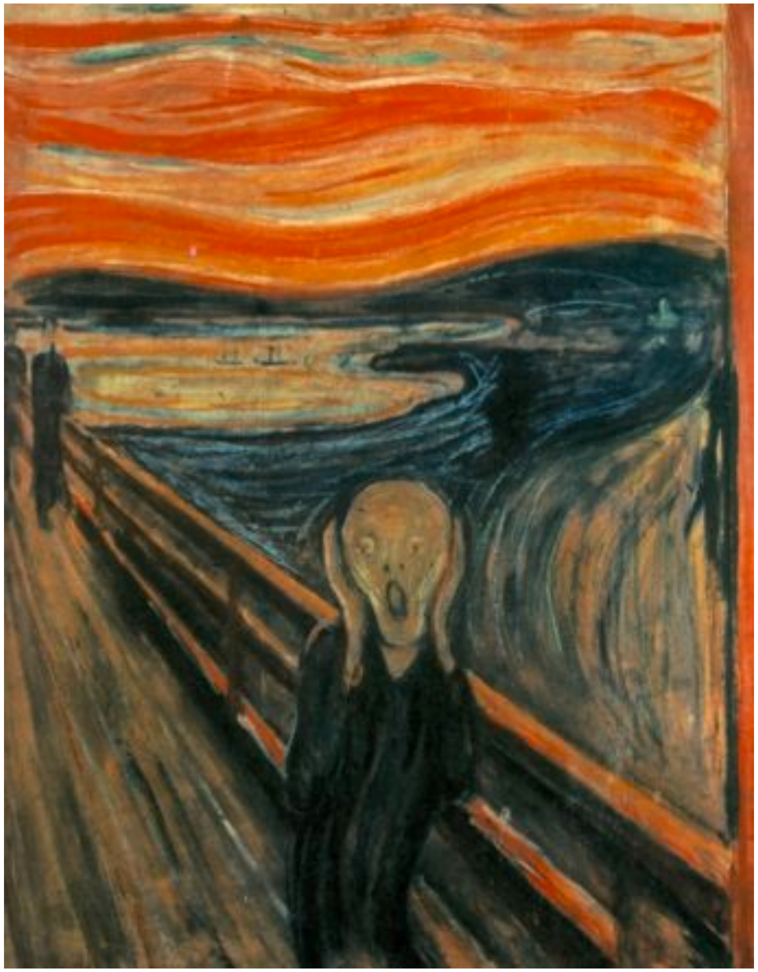
Inlined Futures

- Thought: Maybe you could do that yield trick

```
@inlined_future
def do_func(x, y):
    result = yield pool.submit(func, x, y)
    print('Got:', result)

run_inline_future(do_func)
```

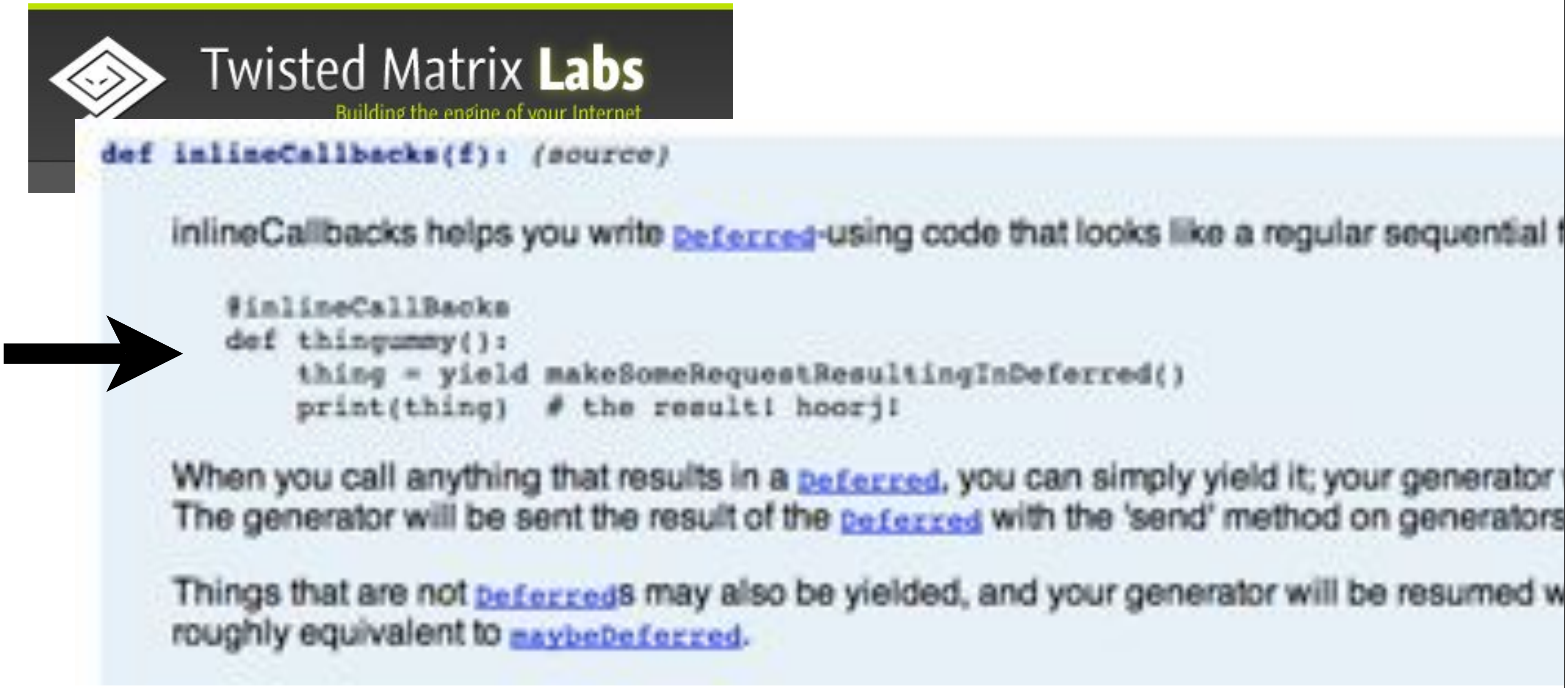
- The extra callback function is eliminated
- Now, just one "simple" function
- Inspired by @contextmanager

Déjà Vu



Déjà Vu

- This twisted idea has been used before...



```
def inlineCallbacks(f): (source)
```

inlineCallbacks helps you write Deferred-using code that looks like a regular sequential f

```
    #inlineCallBacks
    def thingummy():
        thing = yield makeSomeRequestResultingInDeferred()
        print(thing)  # the result! hoorj!
```

When you call anything that results in a Deferred, you can simply yield it; your generator
The generator will be sent the result of the Deferred with the 'send' method on generators

Things that are not Deferreds may also be yielded, and your generator will be resumed w
roughly equivalent to maybeDeferred.

Preview

- There are two separate parts
- Part 1: Wrapping generators with a "task"

```
t = Task(gen)
```

- Part 2: Implementing some runtime code

```
run_inline_future(gen)
```

- Forewarning: It will bend your mind a bit

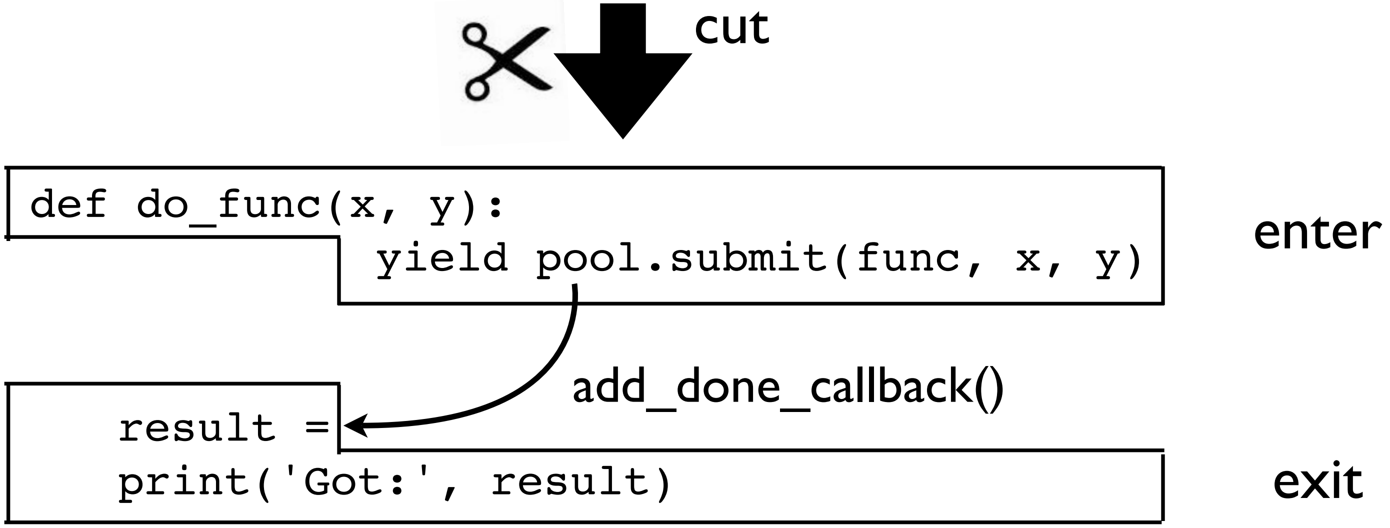
Commentary

- Will continue to use threads for examples
- Mainly because they're easy to work with
- And I don't want to get sucked into an event loop
- Don't dwell on it too much
- Key thing: There is some background processing

Running the Generator

- Problem: Stepping through a generator

```
def do_func(x, y):
    result = yield pool.submit(func, x, y)
    print('Got:', result)
```



```
def do_func(x, y):
            yield pool.submit(func, x, y)
```

```
    result =
    print('Got:', result)
```

- Involves gluing callbacks and yields together

Running the Generator

```
class Task:
    def __init__(self, gen):
        self._gen = gen

    def step(self, value=None):
        try:
            fut = self._gen.send(value)
            fut.add_done_callback(self._wakeup)
        except StopIteration as exc:
            pass

    def _wakeup(self, fut):
        result = fut.result()
        self.step(result)
```

Running the Generator

```
class Task:
    def __init__(self, gen):
        self._gen = gen

    def step(self, value=None):
        try:
            fut = self._gen.send(value)
            fut.add_done_callback(self._wakeup)
        except StopIteration as exc:
            pass

    def _wakeup(self, fut):
        result = fut.result()
        self.step(result)
```

Task class wraps around and represents a running generator.

Running the Generator

```
class Task:
    def __init__(self, gen):
        self._gen = gen

    def step(self, value=None):
        try:
            fut = self._gen.send(value)
            fut.add_done_callback(self._wakeup)
        except StopIteration as exc:
            pass

    def _wakeup(self, fut):
        result = fut.result()
        self.step(result)
```

Advance the generator to the next yield, sending in a value (if any)

Running the Generator

```
class Task:
    def __init__(self, gen):
        self._gen = gen

    def step(self, value=None):
        try:
            fut = self._gen.send(value)
            fut.add_done_callback(self._wakeup)
        except StopIteration as exc:
            pass

    def _wakeup(self, fut):
        result = fut.result()
        self.step(result)
```

Attach a callback to
the produced Future

Running the Generator

```
class Task:
    def __init__(self, gen):
        self._gen = gen

    def step(self, value=None):
        try:
            fut = self._gen.send(value)
            fut.add_done_callback(self._wakeup)
        except StopIteration as exc:
            pass

    def _wakeup(self, fut):
        result = fut.result()
        self.step(result)
```

Collect result and send back into the generator

Does it Work?

- Try it:

```
pool = ThreadPoolExecutor(max_workers=8)

def func(x, y):
    time.sleep(1)
    return x + y

def do_func(x, y):
    result = yield pool.submit(func, x, y)
    print('Got:', result)

t = Task(do_func(2, 3))
t.step()
```

Note: must initiate first step of the task to get it to run

- Output:

```
Got: 5
```

- Yes, it works

Does it Work?

- More advanced: multiple yields/looping

```
pool = ThreadPoolExecutor(max_workers=8)

def func(x, y):
    time.sleep(1)
    return x + y

def do_many(n):
    while n > 0:
        result = yield pool.submit(func, n, n)
        print('Got:', result)
        n -= 1

t = Task(do_many(10))
t.step()
```

- Yes, this works too.

Exception Handling

```
class Task:
    def __init__(self, gen):
        self._gen = gen

    def step(self, value=None, exc=None):
        try:
            if exc:
                fut = self._gen.throw(exc)
            else:
                fut = self._gen.send(value)
            fut.add_done_callback(self._wakeup)
        except StopIteration as exc:
            pass

    def _wakeup(self, fut):
        try:
            result = fut.result()
            self.step(result, None)
        except Exception as exc:
            self.step(None, exc)
```

Exception Handling

```
class Task:
    def __init__(self, gen):
        self._gen = gen

    def step(self, value=None, exc=None):
        try:
            if exc:
                fut = self._gen.throw(exc)
            else:
                fut = self._gen.send(value)
            fut.add_done_callback(self._wakeup)
        except StopIteration as exc:
            pass

    def _wakeup(self, fut):
        try:
            result = fut.result()
            self.step(result, None)
        except Exception as exc:
            self.step(None, exc)
```

send() or throw()
depending on
success

Exception Handling

```
class Task:
    def __init__(self, gen):
        self._gen = gen

    def step(self, value=None, exc=None):
        try:
            if exc:
                fut = self._gen.throw(exc)
            else:
                fut = self._gen.send(value)
            fut.add_done_callback(self._wakeup)
        except StopIteration as exc:
            pass

    def _wakeup(self, fut):
        try:
            result = fut.result()
            self.step(result, None)
        except Exception as exc:
            self.step(None, exc)
```

Catch exceptions
and pass to next
step as appropriate

Error Example

- Try it:

```
def do_func(x, y):
    try:
        result = yield pool.submit(func, x, y)
        print('Got:', result)
    except Exception as e:
        print('Failed:', repr(e))

t = Task(do_func(2, 'Hello'))
t.step()
```

- Output:

```
Failed: TypeError("unsupported operand type(s) for +:
'int' and 'str'",)
```

- Yep, that works too.

Commentary

- This whole thing is rather bizarre
- Execution of the inlined future takes place all on its own (concurrently with other code)
- The normal rules don't apply

Consider

- Infinite recursion?

```
def recursive(n):
    yield pool.submit(time.sleep, 0.001)
    print('Tick:', n)
    Task(recursive(n+1)).step()

Task(recursive(0)).step()
```

- Output:

```
Tick: 0
Tick: 1
Tick: 2
...
Tick: 1662773
Tick: 1662774
...
```

Part 4

source: @UrsulaWJ

yield from yield from yield from yield from future
(maybe)

A Singular Focus

- Focus on the future
- Not the past
- Not now
- Yes, the future.
- No, really, the future.

(but not the singularity)

A Singular Focus

```
class Task:
    def __init__(self, gen):
        self._gen = gen

    def step(self, value=None, exc=None):
        try:
            if exc:
                fut = self._gen.throw(exc)
            else:
                fut = self._gen.send(value)
            fut.add_done_callback(self._wakeup)
        except StopIteration as exc:
            pass
...
```

generator must only produce Futures

Puzzler

- Can you make library functions?

```
def after(delay, gen):
    '''
    Run an inlined future after a time delay
    '''
    yield pool.submit(time.sleep, delay)
    yield gen

Task(after(10, do_func(2, 3))).step()
```

- It's trying to delay the execution of a user-supplied inlined future until later.

Puzzler

- Can you make library functions?

```
def after(delay, gen):
    '''
    Run an inlined future after a time delay
    '''
    yield pool.submit(time.sleep, delay)
    yield gen

Task(after(10, do_func(2, 3))).step()
```

- No

```
Traceback (most recent call last):
...
AttributeError: 'generator' object has no attribute
'add_done_callback'
```

Puzzler

- Can you make library functions?

```
def after(delay, gen):
    '''
    Run an inlined future after a time delay
    '''
    yield pool.submit(time.sleep, delay)
    yield gen

Task(after(10, do_func(2, 3))).step()
```

- This is busted
- gen is a generator, not a Future

Puzzler (2nd Attempt)

- What about this?

```
def after(delay, gen):
    '''
    Run an inlined future after a time delay
    '''
    yield pool.submit(time.sleep, delay)
    for f in gen:
        yield f

Task(after(10, do_func(2, 3))).step()
```

- Idea: Just iterate the generator manually
- Make it produce the required Futures

Puzzler (2nd Attempt)

- What about this?

```
def after(delay, gen):
    '''
    Run an inlined future after a time delay
    '''
    yield pool.submit(time.sleep, delay)
    for f in gen:
        yield f

Task(after(10, do_func(2, 3))).step()
```

- No luck. The result gets lost somewhere

```
Got: None
```

- Hmmm.

Puzzler (3rd Attempt)

- Obvious solution (duh!)

```
def after(delay, gen):
    yield pool.submit(time.sleep, delay)
    result = None
    try:
        while True:
            f = gen.send(result)
            result = yield f
    except StopIteration:
        pass

Task(after(10, do_func(2, 3))).step()
```

- Hey, it works!

```
Got: 5
```

Puzzler (3rd Attempt)

- Obvious solution (duh!)

```
def after(delay, gen):
    yield pool.submit(time.sleep, delay)
    result = None
    try:
        while True:
            f = gen.send(result)
            result = yield f
    except StopIteration:
        pass

Task(after(10, do_func(2, 3))).step()
```

manual running of generator with results (ugh!)

- Hey, it works!

```
Got: 5
```

Puzzler (4th Attempt)

- A better solution: yield from

```
def after(delay, gen):
    yield pool.submit(time.sleep, delay)
    yield from gen

Task(after(10, do_func(2, 3))).step()
```

- 'yield from' - Runs the generator for you
- And it works! (yay!)

```
Got: 5
```

- Awesome

PEP 380

- yield from gen - Delegate to a subgenerator

```
def generator():
    ...
    yield value
    ...
    return result

def func():
    result = yield from generator()
```

- Transfer control to other generators
- Operations take place at the current yield
- Far more powerful than you might think

Conundrum

- "yield" and "yield from"?

```
def after(delay, gen):
    yield pool.submit(time.sleep, delay)
    yield from gen
```

- Two different yields in the same function
- Nobody will find <u>that</u> confusing (NOT!)

Puzzler (5th Attempt)

- Maybe this will work?

```
def after(delay, gen):
    yield from pool.submit(time.sleep, delay)
    yield from gen

Task(after(10, do_func(2, 3))).step()
```

- Just use 'yield from'- always!

Puzzler (5th Attempt)

- Maybe this will work?

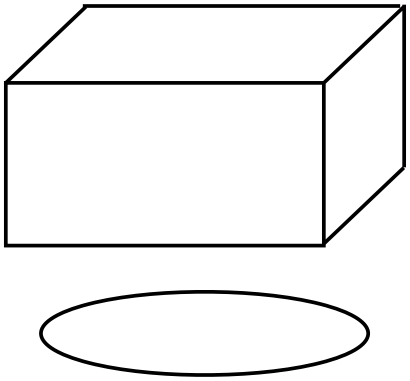
```
def after(delay, gen):
    yield from pool.submit(time.sleep, delay)
    yield from gen

Task(after(10, do_func(2, 3))).step()
```

- Just use 'yield from'- always!
- No. 'yield' and 'yield from' not interchangeable:

```
Traceback (most recent call last):
...
TypeError: 'Future' object is not iterable
>>>
```

??????

(Can it be made to work?)



Iterable Futures

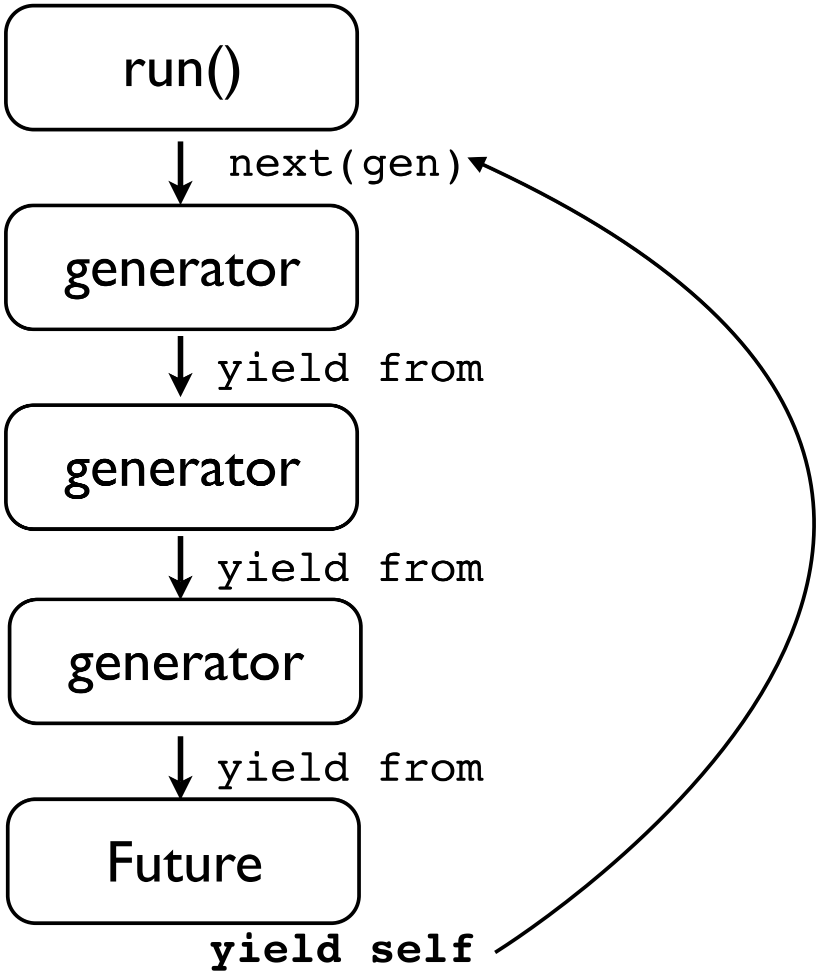
- A simple ingenious patch

```
def patch_future(cls):
    def __iter__(self):
        if not self.done():
            yield self
        return self.result()
    cls.__iter__ = __iter__

from concurrent.futures import Future
patch_future(Future)
```

- It makes all Future instances iterable
- They simply produce themselves and the result
- It magically makes 'yield from' work!

All Roads Lead to Future

- Future is the only thing that actually yields
- Everything else delegates using 'yield from'
- Future terminates the chain



run()

`next(gen)`

generator

`yield from`

generator

`yield from`

generator

`yield from`

Future

`yield self`

The Decorator

- Generators yielding futures is its own world
- Probably a good idea to have some demarcation

```
import inspect
def inlined_future(func):
    assert inspect.isgeneratorfunction(func)
    return func
```

- Does nothing much at all, but serves as syntax

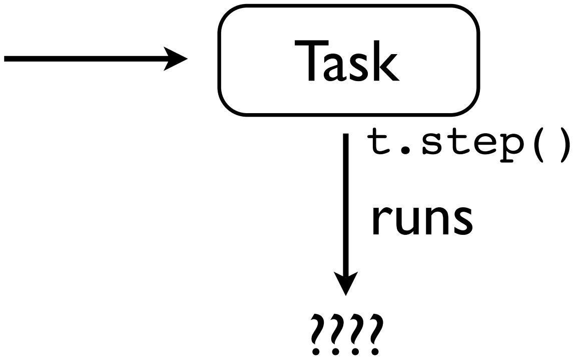
```
@inlined_future
def after(delay, gen):
    yield from pool.submit(time.sleep, delay)
    yield from gen
```

- Alerts others about what you're doing

Task Wrangling

- The "Task" object is just weird

```
t = Task(gen)
t.step()
```



- No way to obtain a result
- No way to join with it
- Or do much of anything useful at all

Tasks as Futures

- This tiny tweak makes it much more interesting

```
class Task(Future):
    def __init__(self, gen):
      super().__init__()
      self._gen = gen

    def step(self, value=None, exc=None):
        try:
            if exc:
                fut = self._gen.throw(exc)
            else:
                fut = self._gen.send(value)
            fut.add_done_callback(self._wakeup)
        except StopIteration as exc:
            self.set_result(exc.value)
```

Tasks as Futures

- This tiny tweak makes it <u>much</u> more interesting

```
class Task(Future):
    def __init__(self, gen):
        super().__init__()
        self._gen = gen

    def step(self, value=None, exc=None):
        try:
            if exc:
                fut = self._gen.throw(exc)
            else:
                fut = self._gen.send(value)
            fut.add_done_callback(self._wakeup)
        except StopIteration as exc:
            self.set_result(exc.value)
```

A Task is a Future

Set its result upon completion

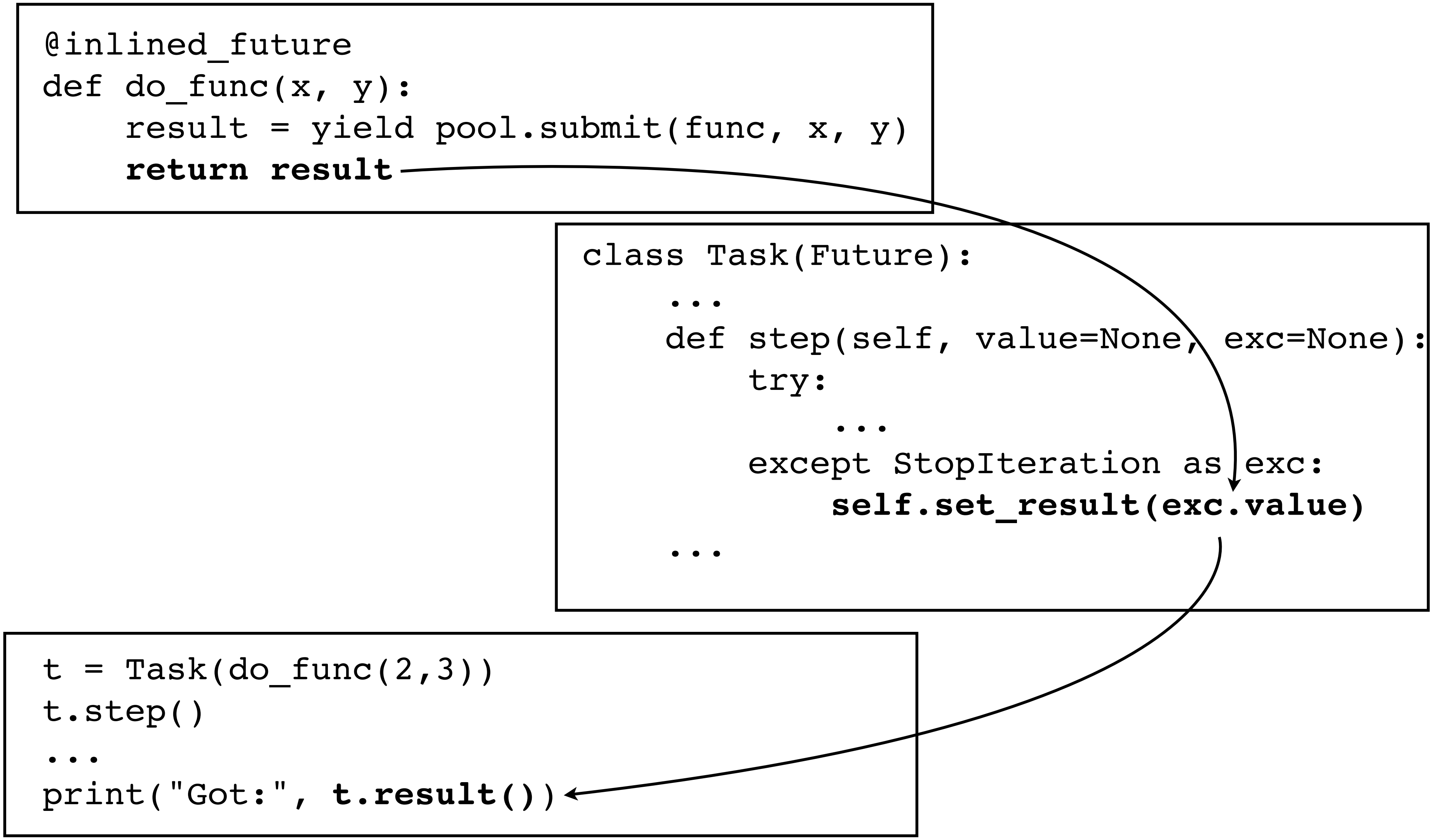
Example

- Obtaining the result of task

```
@inlined_future
def do_func(x, y):
    result = yield pool.submit(func, x, y)
    return result

t = Task(do_func(2,3))
t.step()
...
print("Got:", t.result())
```

- So, you create a task that runs a generator producing Futures
- The task is also a Future
- Right. Got it.

Example



```
@inlined_future
def do_func(x, y):
    result = yield pool.submit(func, x, y)
    return result
```

```
class Task(Future):
    ...
    def step(self, value=None, exc=None):
        try:
            ...
        except StopIteration as exc:
            self.set_result(exc.value)
    ...
```

```
t = Task(do_func(2,3))
t.step()
...
print("Got:", t.result())
```

Task Runners

- You can make utility functions to hide details

```
def start_inline_future(fut):
    t = Task(fut)
    t.step()
    return t

def run_inline_future(fut):
    t = start_inline_future(fut)
    return t.result()
```

- Example: Run an inline future to completion

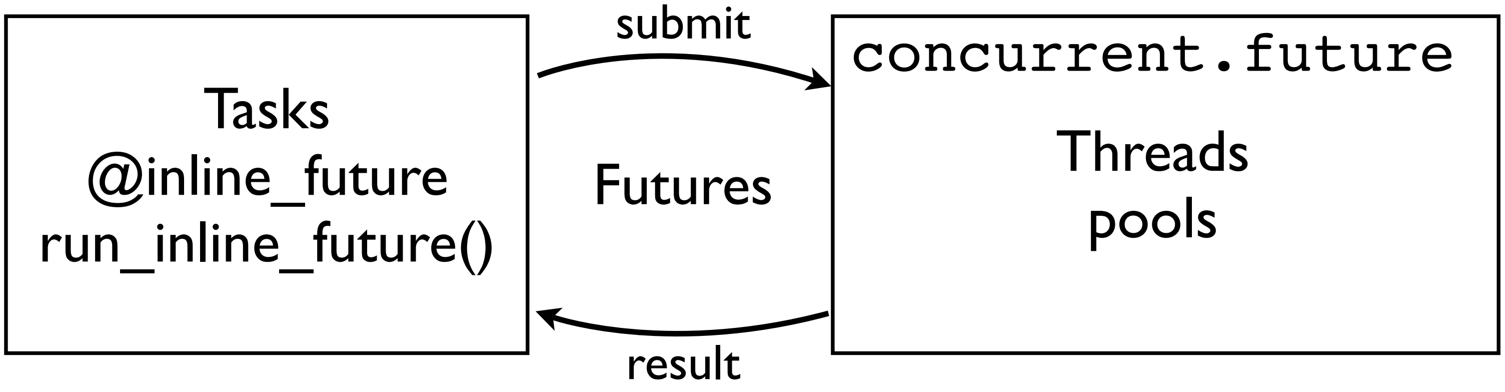
```
result = run_inline_future(do_func(2,3))
print('Got:', result)
```

- Example: Run inline futures in parallel

```
t1 = start_inline_future(do_func(2, 3))
t2 = start_inline_future(do_func(4, 5))
result1 = t1.result()
result2 = t2.result()
```

Step Back Slowly

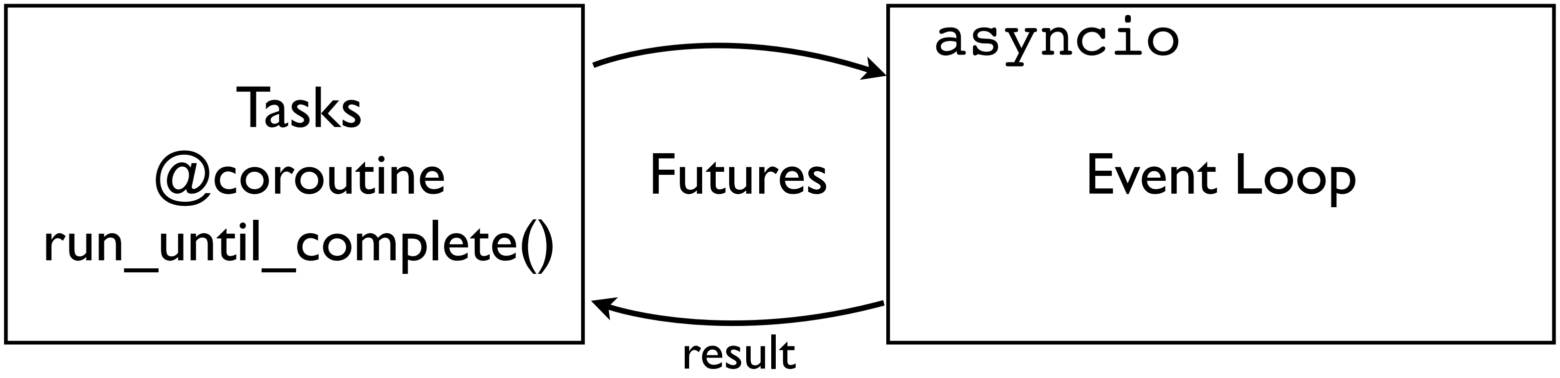
- Built a generator-based task system for threads



- Execution of the future hidden in background
- Note: that was on purpose (for now)

asyncio

- Ideas are the foundation asyncio coroutines



- In fact, it's almost exactly the same
- Naturally, there are some details with event loop

Simple Example

- asyncio "hello world"

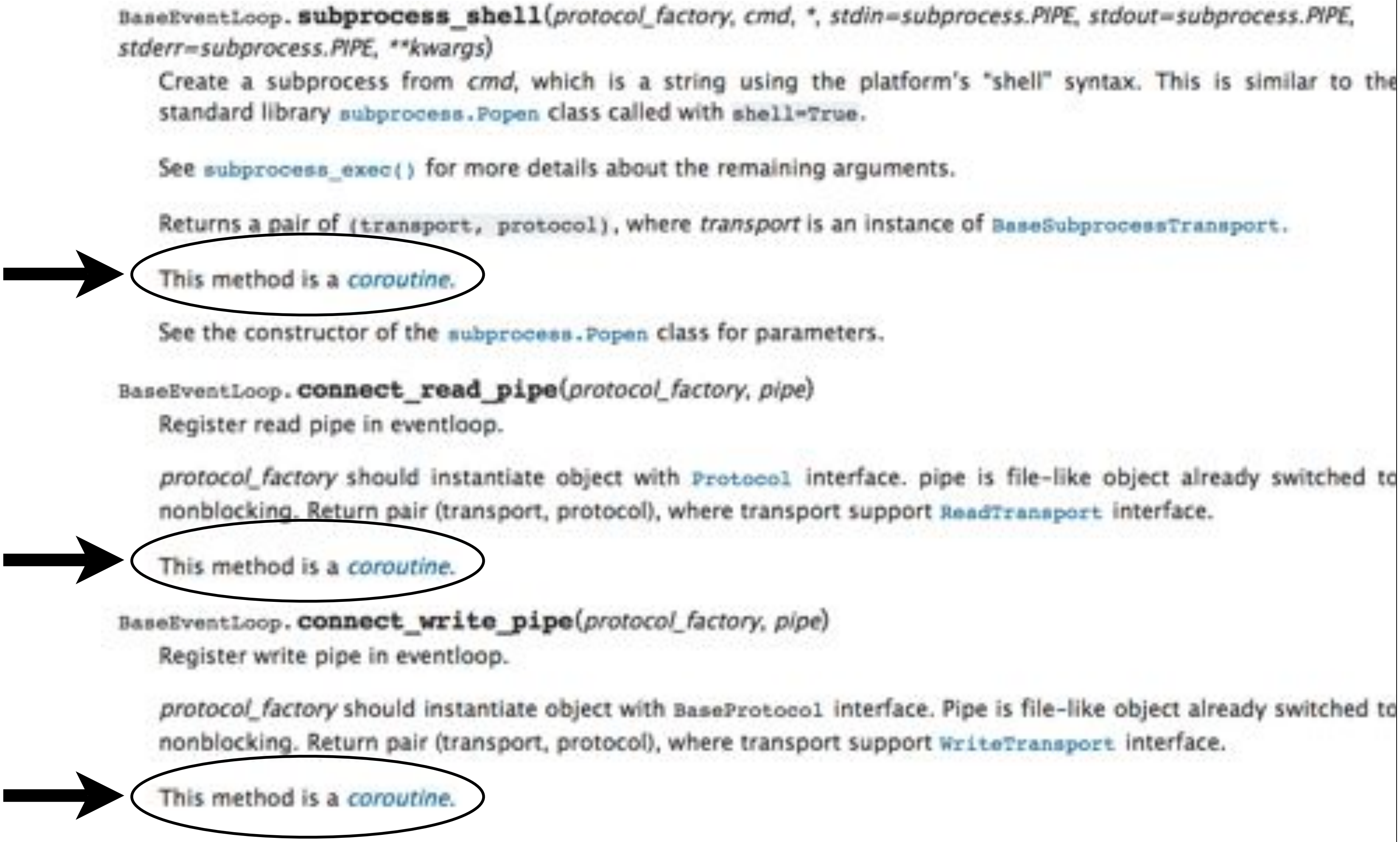
```
import asyncio

def func(x, y):
    return x + y

@asyncio.coroutine
def do_func(x, y):
    yield from asyncio.sleep(1)
    return func(x, y)

loop = asyncio.get_event_loop()
result = loop.run_until_complete(do_func(2,3))
print("Got:", result)
```

Advanced Example

- asyncio - Echo Server

```
import asyncio

@asyncio.coroutine
def echo_client(reader, writer):
    while True:
        line = yield from reader.readline()
        if not line:
            break
        resp = b'Got:' + line
        writer.write(resp)
    writer.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(
   asyncio.start_server(echo_client, host='', port=25000)
)
loop.run_forever()
```

Be on the Lookout!



`BaseEventLoop.`**`subprocess_shell`**(*protocol_factory, cmd, *, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, **kwargs*)

Create a subprocess from *cmd*, which is a string using the platform's "shell" syntax. This is similar to the standard library `subprocess.Popen` class called with `shell=True`.

See `subprocess_exec()` for more details about the remaining arguments.

Returns a pair of `(transport, protocol)`, where *transport* is an instance of `BaseSubprocessTransport`.

This method is a *coroutine*.

See the constructor of the `subprocess.Popen` class for parameters.

`BaseEventLoop.`**`connect_read_pipe`**(*protocol_factory, pipe*)

Register read pipe in eventloop.

protocol_factory should instantiate object with `Protocol` interface. pipe is file-like object already switched to nonblocking. Return pair (transport, protocol), where transport support `ReadTransport` interface.

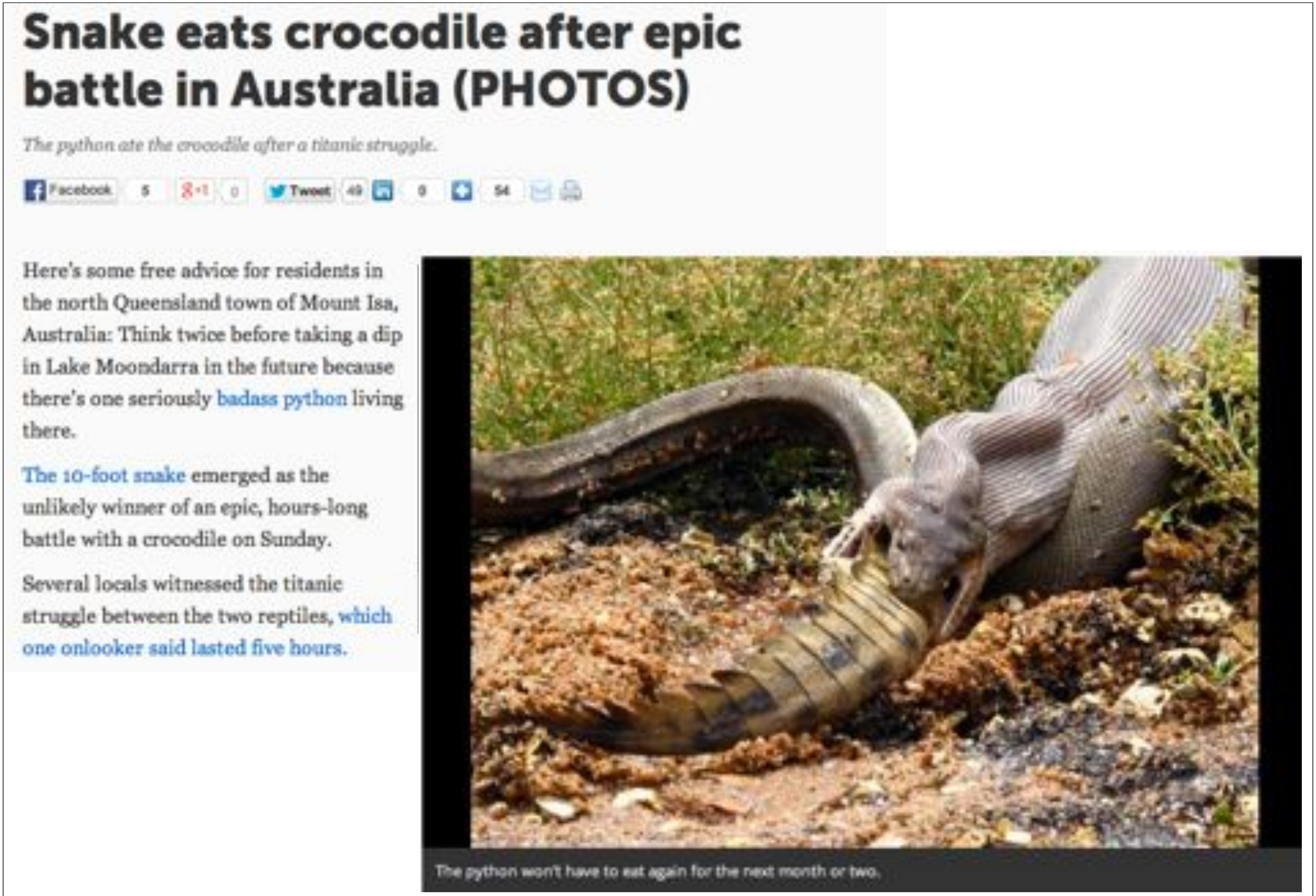
This method is a *coroutine*.

`BaseEventLoop.`**`connect_write_pipe`**(*protocol_factory, pipe*)

Register write pipe in eventloop.

protocol_factory should instantiate object with `BaseProtocol` interface. Pipe is file-like object already switched to nonblocking. Return pair (transport, protocol), where transport support `WriteTransport` interface.

This method is a *coroutine*.



Snake eats crocodile after epic battle in Australia (PHOTOS)

The python ate the crocodile after a titanic struggle.

Here's some free advice for residents in the north Queensland town of Mount Isa, Australia: Think twice before taking a dip in Lake Moondarra in the future because there's one seriously badass python living there.

The 10-foot snake emerged as the unlikely winner of an epic, hours-long battle with a crocodile on Sunday.

Several locals witnessed the titanic struggle between the two reptiles, which one onlooker said lasted five hours.

The python won't have to eat again for the next month or two.

(source: globalpost.com)

Part 5

Python Threads

- Threads, what are they good for?
- Answer: Nothing, that's what!
- Damn you GIL!!

Actually...

- Threads are great at doing nothing!

```
time.sleep(2)              # Do nothing for awhile

data = sock.recv(nbytes)  # Wait around for data
data = f.read(nbytes)
```

- In fact, great for I/O!
- Mostly just sitting around

CPU-Bound Work

- Threads are weak for computation
- Global interpreter lock only allows 1 CPU
- Multiple CPU-bound threads fight each other
- Could be better

http://www.dabeaz.com/GIL

A Solution

- Naturally, we must reinvent the one thing that threads are good at
- Namely, waiting around.
- Event-loops, async, coroutines, green threads.
- Think about it: These are focused on I/O

(yes, I know there are other potential issues with threads, but work with me here)

CPU-Bound Work

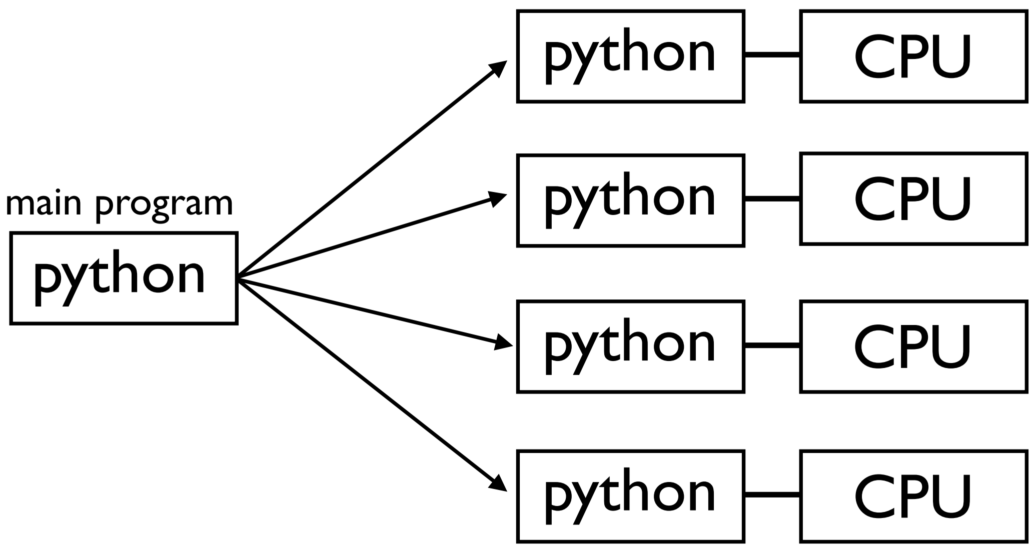
- Event-loops have their own issues

(source: chicagotribune.com)

- Don't bug me, I'm blocking right now

Standard Solution

- Delegate the work out to a process pool



- multiprocessing, concurrent.futures, etc.

Thought Experiment

- Didn't we just do this with inlined futures?

```
def fib(n):
    return 1 if n <= 2 else (fib(n-1) + fib(n-2))

@inlined_future
def compute_fibs(n):
    result = []
    for i in range(n):
        val = yield from pool.submit(fib, i)
        result.append(val)
    return result

pool = ProcessPoolExecutor(4)
result = run_inline_future(compute_fibs(35))
```

- It runs without crashing (let's ship it!)

Thought Experiment

- Sequential execution

```
run_inline_future(compute_fibs(34))
run_inline_future(compute_fibs(34))
```

- Can you launch tasks in parallel?

```
t1 = start_inline_future(compute_fibs(34))
t2 = start_inline_future(compute_fibs(34))
result1 = t1.result()
result2 = t2.result()
```

- Recall (from earlier)

```
def start_inline_future(fut):
    t = Task(fut)
    t.step()
    return t
```

Thought Experiment

- Sequential execution

```
run_inline_future(compute_fibs(34))
run_inline_future(compute_fibs(34))
```

→ 9.56s

- Can you launch tasks in parallel?

```
t1 = start_inline_future(compute_fibs(34))
t2 = start_inline_future(compute_fibs(34))
result1 = t1.result()
result2 = t2.result()
```

- Recall (from earlier)

```
def start_inline_future(fut):
    t = Task(fut)
    t.step()
    return t
```

Thought Experiment

- Sequential execution

```
run_inline_future(compute_fibs(34))
run_inline_future(compute_fibs(34))
```

→ 9.56s

- Can you launch tasks in parallel?

```
t1 = start_inline_future(compute_fibs(34))
t2 = start_inline_future(compute_fibs(34))
result1 = t1.result()
result2 = t2.result()
```

→ 4.78s

Inlined tasks running outside confines of the GIL?

- Recall (from earlier)

```
def start_inline_future(fut):
    t = Task(fut)
    t.step()
    return t
```

Execution Model

- The way in which it works is a little odd

```
@inlined_future
def compute_fibs(n):
    result = []
    for i in range(n):
        print(threading.current_thread())
        val = yield from pool.submit(fib, i)
        result.append(val)
    return result
```

add this

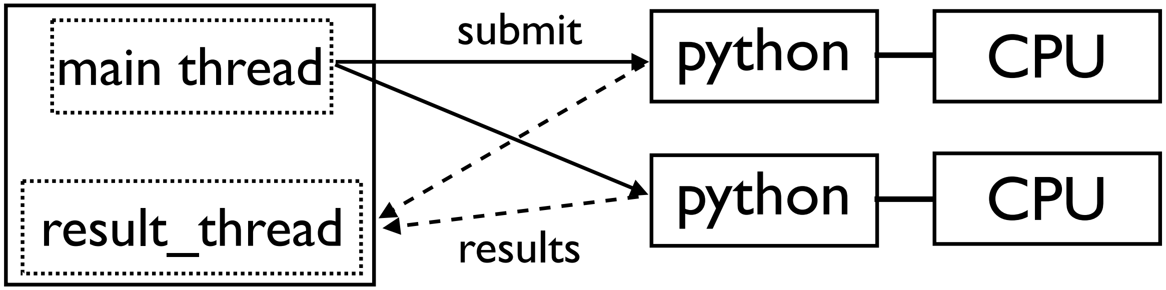
- Output: (2 Tasks)

```
<_MainThread(MainThread, started 140735086636224)>
<_MainThread(MainThread, started 140735086636224)>
<Thread(Thread-1, started daemon 4320137216)>
<Thread(Thread-1, started daemon 4320137216)>
<Thread(Thread-1, started daemon 4320137216)>
...
```

????

Process Pools

- Process pools involve a hidden result thread



- result thread reads returned values
- Sets the result on the associated Future
- Triggers the callback function (if any)

The Issue

- Our inlined future switches execution threads

```
@inlined_future
def compute_fibs(n):
    result = []
    for i in range(n):
        val = yield from pool.submit(fib, i)
        result.append(val)
    return result
```

main thread

result thread

- Switch occurs at the first yield
- All future execution occurs in result thread
- That could be a little weird (especially if it blocked)

Important Lesson

- If you're going to play with control flow, you must absolutely understand possible implications under the covers (i.e., switching threads across the yield statement).

Insight

- The yield is <u>not</u> implementation

```
@inlined_future
def compute_fibs(n):
    result = []
    for i in range(n):
        val = yield from pool.submit(fib, i)
        result.append(val)
    return result
```

- You can implement different execution models
- You don't have to follow a formulaic rule

Inlined Thread Execution

- Variant: Run generator entirely in a single thread

```
def run_inline_thread(gen):
    value = None
    exc = None
    while True:
        try:
            if exc:
                fut = gen.throw(exc)
            else:
                fut = gen.send(value)
            try:
                value = fut.result()
                exc = None
            except Exception as e:
                exc = e
        except StopIteration as exc:
            return exc.value
```

- It just steps through... no callback function

New Execution

- Try it again with a thread pool (because why not?)

```
@inlined_future
def compute_fibs(n):
    result = []
    for i in range(n):
        print(threading.current_thread())
        val = yield from pool.submit(fib, i)
        result.append(val)
    return result

tpool = ThreadPoolExecutor(8)
t1 = tpool.submit(run_inline_thread(compute_fibs(34)))
t2 = tpool.submit(run_inline_thread(compute_fibs(34)))
result1 = t1.result()
result2 = t2.result()
```

New Execution

- Output: (2 Threads)

```
<Thread(Thread-1, started 4319916032)>
<Thread(Thread-2, started 4326428672)>
<Thread(Thread-1, started 4319916032)>
<Thread(Thread-2, started 4326428672)>
<Thread(Thread-1, started 4319916032)>
<Thread(Thread-2, started 4326428672)>
...
```

(works perfectly)

4.60s

(a bit faster)

- Processes, threads, and futures in perfect harmony
- Uh... let's move along. Faster. Must go faster.

Big Idea

- You can mold and adapt generator execution

- That yield statement: magic!

Part 6

Fake it until you make it

Actors

- There is a striking similarity between coroutines and actors (i.e., the "actor" model)
- Features of Actors
    - Receive messages
    - Send messages to other actors
    - Create new actors
    - No shared state (messages only)
- Can coroutines serve as actors?

Example

- A very simple example

```
@actor
def printer():
    while True:
        msg = yield
        print('printer:', msg)

printer()
n = 10
while n > 0:
    send('printer', n)
    n -=1
```

idea: use generators to define a kind of "named" actor task

Attempt 1

- Make a central coroutine registry and a decorator

```
_registry = { }

def send(name, msg):
    _registry[name].send(msg)

def actor(func):
    def wrapper(*args, **kwargs):
        gen = func(*args, **kwargs)
        next(gen)
        _registry[func.__name__] = gen
    return wrapper
```

- Let's see if it works...

Example

- A very simple example

```
@actor
def printer():
    while True:
        msg = yield
        print('printer:', msg)

printer()
n = 10
while n > 0:
    send('printer', n)
    n -=1
```

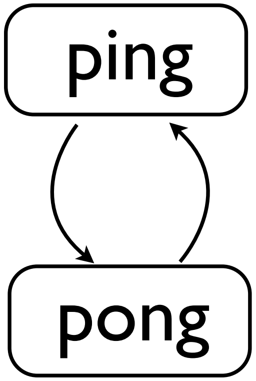
- It seems to work (maybe)

```
printer: 10
printer: 9
printer: 8
...
printer: 1
```

Advanced Example

- Recursive ping-pong (inspired by Stackless)

```
@actor
def ping():
    while True:
        n = yield
        print('ping %d' % n)
        send('pong', n + 1)

@actor
def pong():
    while True:
        n = yield
        print('pong %d' % n)
        send('ping', n + 1)

ping()
pong()
send('ping', 0)
```



Advanced Example

- Alas, it does not work

```
ping 0
pong 1
Traceback (most recent call last):
  File "actor.py", line 36, in <module>
    send('ping', 0)
  File "actor.py", line 8, in send
    _registry[name].send(msg)
  File "actor.py", line 24, in ping
    send('pong', n + 1)
  File "actor.py", line 8, in send
    _registry[name].send(msg)
  File "actor.py", line 31, in pong
    send('ping', n + 1)
  File "actor.py", line 8, in send
    _registry[name].send(msg)
ValueError: generator already executing
```

Problems

- Important differences between actors/coroutines
    - Concurrent execution
    - Asynchronous message delivery
- Although coroutines have a "send()", it's a normal method call
    - Synchronous
    - Involves the call-stack
    - Does not allow recursion/reentrancy

Solution 1

- Wrap the generator with a thread

```
class Actor(threading.Thread):
    def __init__(self, gen):
        super().__init__()
        self.daemon = True
        self.gen = gen
        self.mailbox = Queue()
        self.start()

    def send(self, msg):
        self.mailbox.put(msg)

    def run(self):
        while True:
            msg = self.mailbox.get()
            self.gen.send(msg)
```

- Err..... no.

Solution 2

- Write a tiny message scheduler

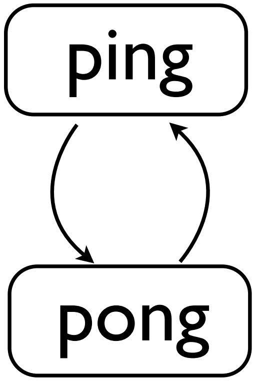
```
_registry = { }
_msg_queue = deque()

def send(name, msg):
    _msg_queue.append((name, msg))

def run():
    while _msg_queue:
        name, msg = _msg_queue.popleft()
        _registry[name].send(msg)
```

- send() simply drops messages on a queue
- run() executes as long as there are messages

Advanced Example

- Recursive ping-pong (reprise)

```
@actor
def ping():
    while True:
        n = yield
        print('ping %d' % n)
        send('pong', n + 1)

@actor
def pong():
    while True:
        n = yield
        print('pong %d' % n)
        send('ping', n + 1)

ping()
pong()
send('ping', 0)
run()
```



Advanced Example

- It works!

```
ping 0
pong 1
ping 2
pong 3
ping 4
ping 5
ping 6
pong 7
...
... forever
```

- That's kind of amazing

Comments

- It's still kind of a fake actor
  - Lacking in true concurrency
  - Easily blocked
- Maybe it's good enough?
- I don't know
- Key idea: you can bend space-time with yield

Part 7

A Terrifying Visitor

Let's Write a Compiler

- Well, an extremely simple one anyways...
- Evaluating mathematical expressions

```
2 + 3 * 4 - 5
```

- Why?
- Because eval() is for the weak, that's why

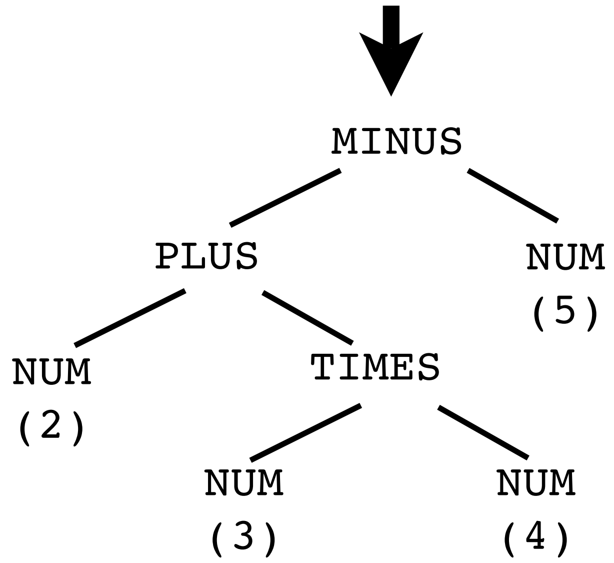
Compilers 101

- Lexing : Make tokens

`2 + 3 * 4 - 5` ⟶ `[NUM,PLUS,NUM,TIMES,NUM,MINUS,NUM]`

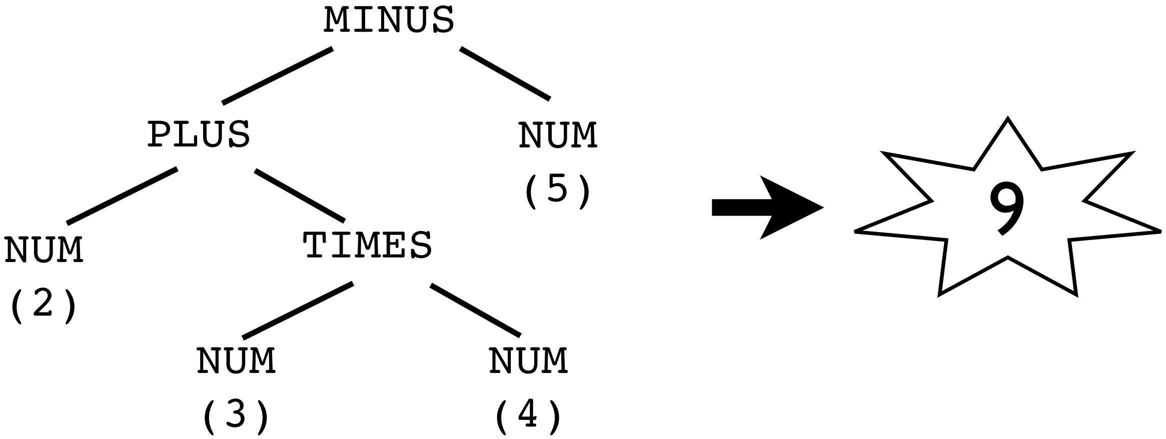
- Parsing : Make a parse tree

`[NUM,PLUS,NUM,TIMES,NUM,MINUS,NUM]`



Compilers 101

- Evaluation : Walk the parse tree



MINUS
PLUS
NUM
(5)
NUM
(2)
TIMES
NUM
(3)
NUM
(4)
9

- It's almost too simple

Tokenizing

```
import re
from collections import namedtuple
tokens = [
    r'(?P<NUM>\d+)',
    r'(?P<PLUS>\+)',
    r'(?P<MINUS>-)',
    r'(?P<TIMES>\*)',
    r'(?P<DIVIDE>/)',
    r'(?P<WS>\s+)',
    ]

master_re = re.compile('|'.join(tokens))
Token = namedtuple('Token', ['type','value'])

def tokenize(text):
    scan = master_re.scanner(text)
    return (Token(m.lastgroup, m.group())
            for m in iter(scan.match, None)
            if m.lastgroup != 'WS')
```

Tokenizing

- Example:

```
text = '2 + 3 * 4 - 5'
for tok in tokenize(text):
    print(tok)
```

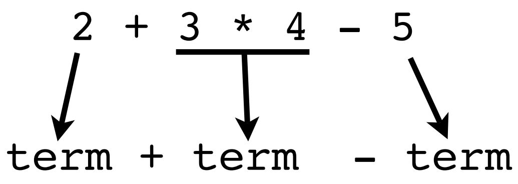
```
Token(type='NUM', value='2')
Token(type='PLUS', value='+')
Token(type='NUM', value='3')
Token(type='TIMES', value='*')
Token(type='NUM', value='4')
Token(type='MINUS', value='-')
Token(type='NUM', value='5')
```

Parsing

- Must match the token stream against a grammar

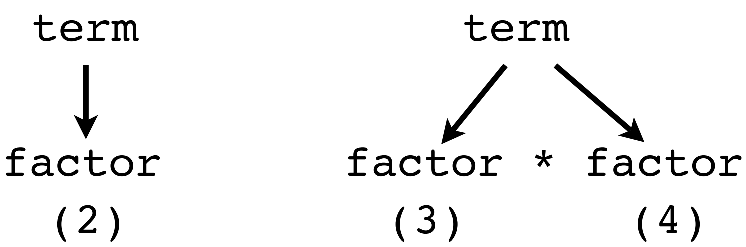
```
expr ::= term { +|- term }*
term ::= factor { *|/ factor}*
factor ::= NUM
```

- An expression is just a bunch of terms



```
2 + 3 * 4 - 5
term + term  - term
```

- A term is just one or more factors



```
term            term
factor          factor * factor
(2)             (3)      (4)
```

Recursive Descent Parse

```
expr ::= term { +|- term }*
term ::= factor { *|/ factor}*
factor ::= NUM
```

Encode the grammar as a collection of functions

```
def expr():
    term()
    while accept('PLUS','MINUS'):
        term()
    print('Matched expr')

def term():
    factor()
    while accept('TIMES','DIVIDE'):
        factor()
    print('Matched term')

def factor():
    if accept('NUM'):
        print('Matched factor')
    else:
        raise SyntaxError()
```

Each function steps through the rule

Recursive Descent Parse

```
def parse(toks):
    lookahead, current = next(toks, None), None
    def accept(*toktypes):
        nonlocal lookahead, current
        if lookahead and lookahead.type in toktypes:
            current, lookahead = lookahead, next(toks, None)
            return True

    def expr():
        term()
        while accept('PLUS','MINUS'):
            term()
        print('Matched expr')
    ...
    expr()
```

Tree Building

- Need some tree nodes for different things

```
class Node:
    _fields = []
    def __init__(self, *args):
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

class BinOp(Node):
    _fields = ['op', 'left', 'right']

class Number(Node):
    _fields = ['value']
```

- Example:

```
n1 = Number(3)
n2 = Number(4)
n3 = BinOp('*', n1, n2)
```

Tree Building

```
def parse(toks):
    ...
    def expr():
        left = term()
        while accept('PLUS','MINUS'):
            left = BinOp(current.value, left)
            left.right = term()
        return left

    def term():
        left = factor()
        while accept('TIMES','DIVIDE'):
            left = BinOp(current.value, left)
            left.right = factor()
        return left

    def factor():
        if accept('NUM'):
            return Number(int(current.value))
        else:
            raise SyntaxError()
    return expr()
```

Building nodes and hooking them together

Our Little Parser

- Story so far...

```
text = '2 + 3*4 - 5'
toks = tokenize(text)
tree = parse(toks)
```

```
BinOp('-',
      BinOp('+',
            Number(2),
            BinOp('*',
                  Number(3),
                  Number(4)
                 )
            ),
      Number(5)
)
```

Evaluation

- The "Visitor" pattern

```
class NodeVisitor:
    def visit(self, node):
        return getattr(self,
                       'visit_' + type(node).__name__)(node)
```

- Example:

```
class MyVisitor(NodeVisitor):
    def visit_Number(self, node):
        print(node.value)
    def visit_BinOp(self, node):
        self.visit(node.left)
        self.visit(node.right)
        print(node.op)

MyVisitor().visit(tree)
```

output

```
2
3
4
*
+
5
-
```

Evaluation

- An Expression Evaluator

```
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_BinOp(self, node):
        leftval = self.visit(node.left)
        rightval = self.visit(node.right)
        if node.op == '+':
            return leftval + rightval
        elif node.op == '-':
            return leftval - rightval
        elif node.op == '*':
            return leftval * rightval
        elif node.op == '/':
            return leftval / rightval

print(Evaluator().visit(tree))
```

→ 9

Digression

- Last 12 slides a whole graduate CS course
- Plus at least one additional Python tutorial
- Don't worry about it
- Left as an exercise...

Death Spiral

- And it almost works...

```
# Make '0+1+2+3+4+...+999'
text = '+'.join(str(x) for x in range(1000))
toks = tokenize(text)
tree = parse(toks)
val = Evaluate().visit(tree)
```

```
Traceback (most recent call last):
  File "compiler.py", line 100, in <module>
    val = Evaluator().visit(tree)
  File "compiler.py", line 63, in visit
    return getattr(self, 'visit_' + type(node).__name__)(node)
  File "compiler.py", line 80, in visit_BinOp
    leftval = self.visit(node.left)
  ...
RuntimeError: maximum recursion depth exceeded while calling a
Python object
```

Evaluation

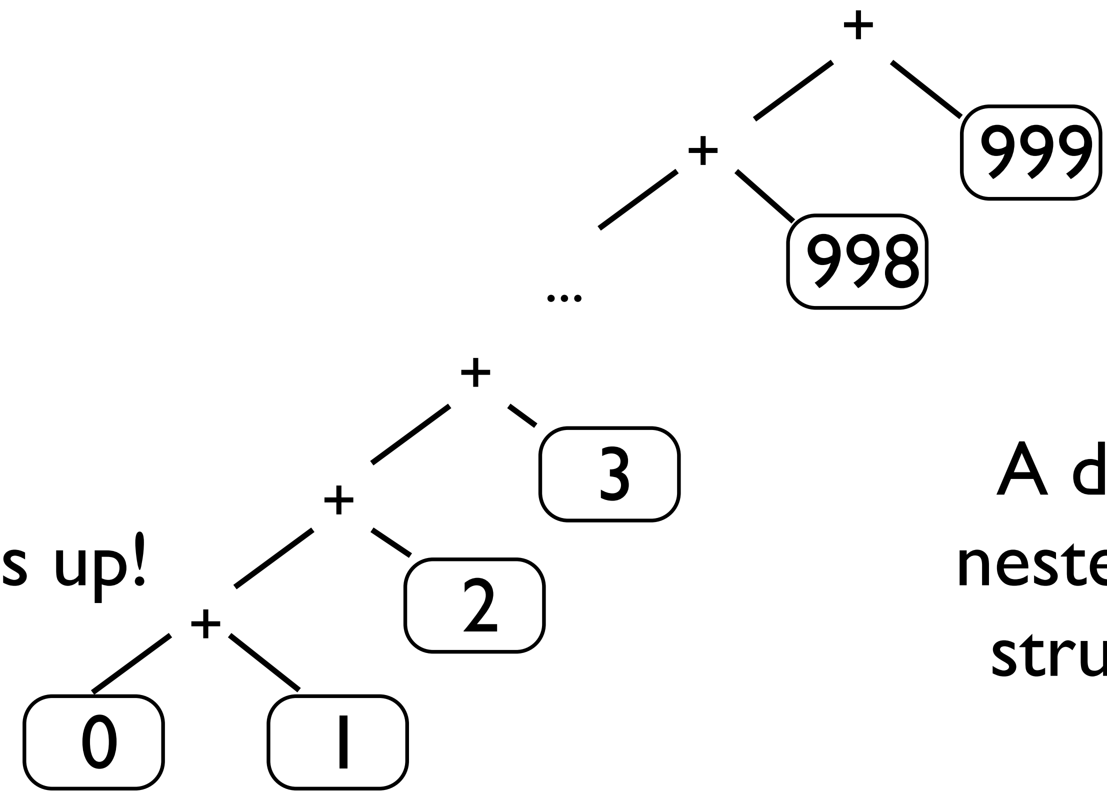
- An Expression Evaluator

```
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_BinOp(self, node):
        leftval = self.visit(node.left)
        rightval = self.visit(node.right)
        if node.op == '+':
            return leftval + rightval
        elif node.op == '-':
            return leftval - rightval
        elif node.op == '*':
            return leftval * rightval
        elif node.op == '/':
            return leftval / rightval

print(Evaluator().visit(tree))
```

!%*@*^#^#
Recursion
(damn you to hell)

Evaluation

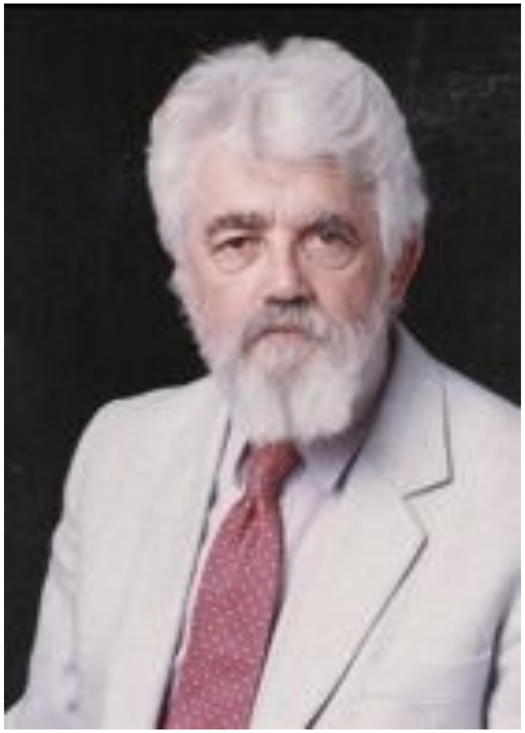
0 + 1 + 2 + 3 + 4 ... + 999



Blows up!

A deeply nested tree structure

I Told You So



- The visitor pattern is bad idea
- Better: Functional language with pattern matching and tail-call optimization

QUESTION

How do you NOT do something?

QUESTION

How do you NOT do something?

(yield?)

Evaluation

- An Expression Evaluator

```
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_BinOp(self, node):
        leftval = yield node.left
        rightval = yield node.right
        if node.op == '+':
            return leftval + rightval
        elif node.op == '-':
            return leftval - rightval
        elif node.op == '*':
            return leftval * rightval
        elif node.op == '/':
            return leftval / rightval

print(Evaluator().visit(tree))
```

Nope. Not doing that recursion.

Generator Wrapping

- Step 1: Wrap "visiting" with a generator

```
class NodeVisitor:
    def genvisit(self, node):
        result = getattr(self,
                         'visit_' + type(node).__name__)(node)
        if isinstance(result, types.GeneratorType):
            result = yield from result
        return result
```

- Thinking: No matter what the visit_() method produces, the result will be a generator
- If already a generator, then just delegate to it

Generator Wrapping

- Example: A method that simply returns a value

```
>>> v = Evaluator()
>>> n = Number(2)
>>> gen = v.genvisit(n)
>>> gen
<generator object genvisit at 0x10070ab88>
>>> gen.send(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 2
>>>
```

- Result: Carried as value in StopIteration

Generator Wrapping

- A method that yields nodes (iteration)

```
>>> v = Evaluator()
>>> n = BinOp('*', Number(3), Number(4))
>>> gen = v.genvisit(n)
>>> gen
<generator object genvisit at 0x10070ab88>
>>> gen.send(None)
<__main__.Number object at 0x1058525c0>
>>> gen.send(_.value)
<__main__.Number object at 0x105852630>
>>> gen.send(_.value)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 12
>>>
```

Again, note the return mechanism

Generator Wrapping

- A method that yields nodes

```
>>> v = Evaluator()
>>> n = BinOp('*', Number(3), Number(4))
>>> gen = v.genvisit(n)
>>> gen
<generator object genvisit at 0x10070ab88>
>>> gen.send(None)
<__main__.Number object at 0x1058525c0>
>>> gen.send(_.value)
<__main__.Number object at 0x105852630>
>>> gen.send(_.value)
Traceback (most recent call las
  File "<stdin>", line 1, in <
StopIteration: 12
>>>
```

Manually carrying out this method in the example

```
def visit_Number(self, node):
    return node.value
```

Running Recursion

- Step 2: Run depth-first traversal with a stack

```
class NodeVisitor:
    def visit(self, node):
        stack = [ self.genvisit(node) ]
        result = None
        while stack:
            try:
                node = stack[-1].send(result)
                stack.append(self.genvisit(node))
                result = None
            except StopIteration as exc:
                stack.pop()
                result = exc.value
        return result
```

- Basically, a stack of running generators

Transcendence

- Does it work?

```
# Make '0+1+2+3+4+...+999'
text = '+'.join(str(x) for x in range(1000))
toks = tokenize(text)
tree = parse(toks)
val = Evaluate().visit(tree)
print(val)
```

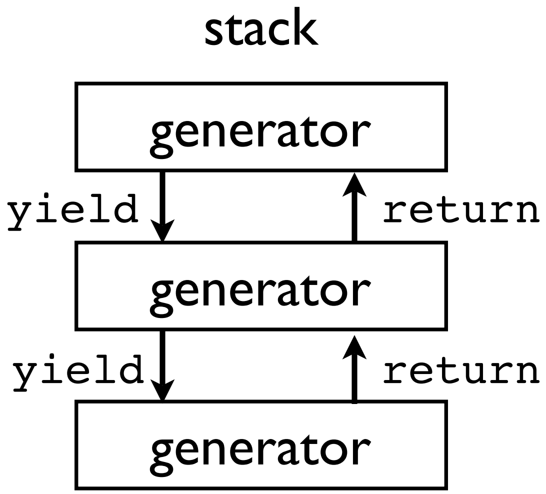
- Yep

```
499500
```

- Yow!

Running Recursion

```
class Evaluator(NodeVisitor):
    def visit_BinOp(self, node):
        leftval = yield node.left
        rightval = yield node.right
        if node.op == '+':
            result = leftval + rightval
        ...
        return result
```



- Each yield creates a new stack entry
- Returned values (via StopIteration) get propagated as results

Running Recursion

```
>>> v = Evaluator()
>>> n = BinOp('*', Number(3), Number(4))
>>> stack = [ v.genvisit(n) ]
>>> stack[-1].send(None)
<__main__.Number object at 0x1058525c0>
>>> stack.append(v.genvisit(_))
>>> stack[-1].send(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 3
>>> stack.pop()
>>> stack[-1].send(3)
<__main__.Number object at 0x105852630>
>>> stack.append(v.genvisit(_))
>>> stack[-1].send(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 4
>>> stack.pop()
>>> stack[-1].send(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 12
>>>
```

Nodes are visited and generators pushed onto a stack

Running Recursion

```
>>> v = Evaluator()
>>> n = BinOp('*', Number(3), Number(4))
>>> stack = [ v.genvisit(n) ]
>>> stack[-1].send(None)
<__main__.Number object at 0x1058525c0>
>>> stack.append(v.genvisit(_))
>>> stack[-1].send(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 3
>>> stack.pop()
>>> stack[-1].send(3)
<__main__.Number object at 0x105852630>
>>> stack.append(v.genvisit(_))
>>> stack[-1].send(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 4
>>> stack.pop()
>>> stack[-1].send(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 12
>>>
```

Results propagate via StopIteration

12 (Final Result)

Final Words

Historical Perspective

- Generators seem to have started as a simple way to implement iteration (Python 2.3)
- Took an interesting turn with support for coroutines (Python 2.5)
- Taken to a whole new level with delegation support in PEP-380 (Python 3.3).

Control Flow Bending

- yield statement allows you to bend control-flow to adapt it to certain kinds of problems
    - Wrappers (context managers)
    - Futures/concurrency
    - Messaging
    - Recursion
- Frankly, it blows my mind.

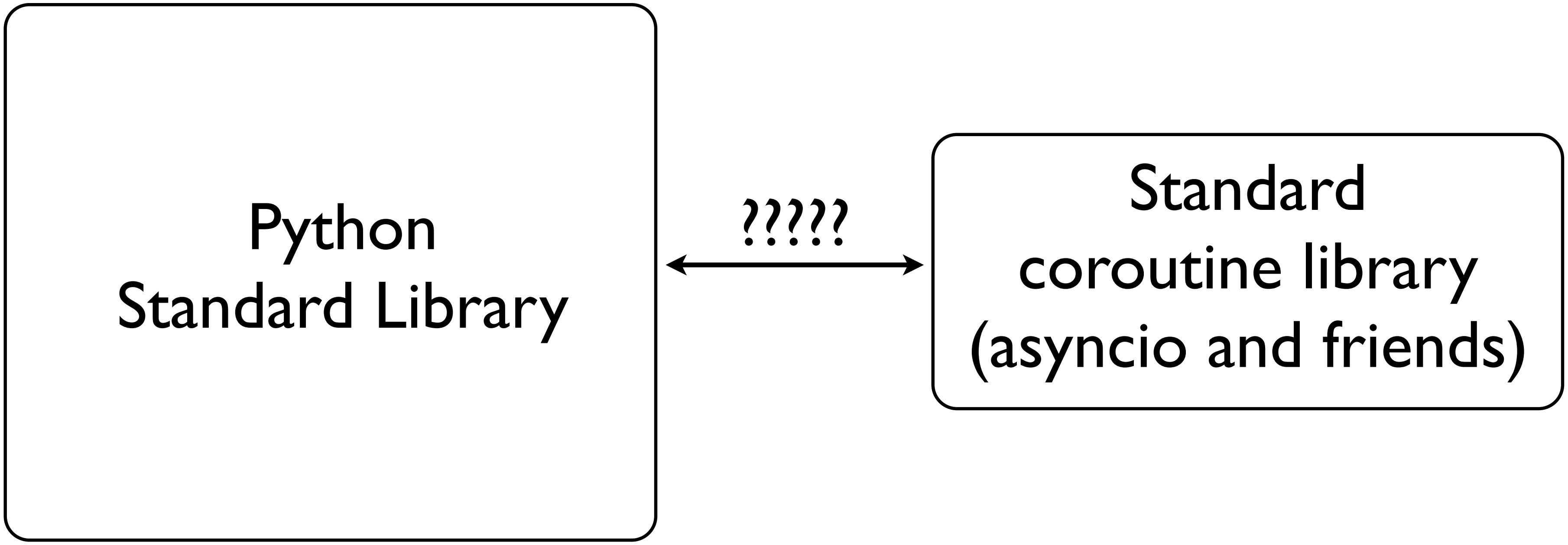
asyncio

- Inclusion of asyncio in standard library may be a game changer
- To my knowledge, it's the only standard library module that uses coroutines/generator delegation in a significant manner
- To really understand how it works, need to have your head wrapped around generators
- Read the source for deep insight

Is it Proper?

- Are coroutines/generators a good idea or not?
- Answer: <u>I still don't know</u>
- Issue: Coroutines seem like they're "all in"
- Fraught with potential mind-bending issues
- Example: Will there be two standard libraries?

Two Libraries?



Python
Standard Library

?????

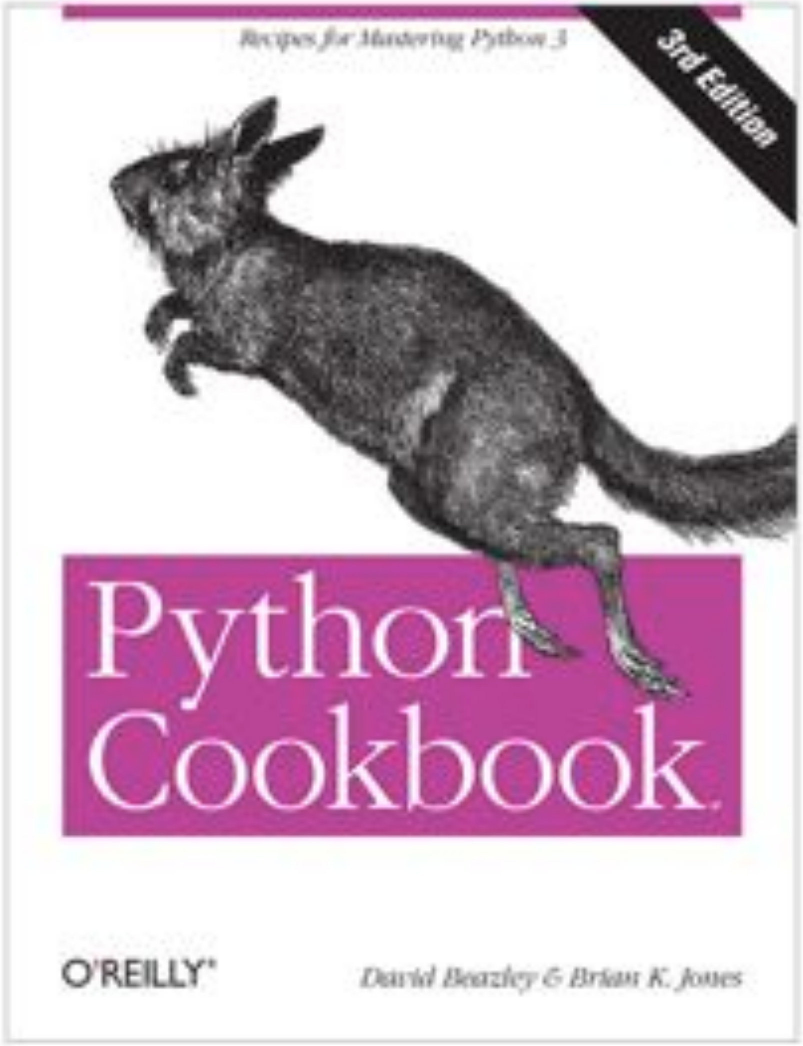
Standard
coroutine library
(asyncio and friends)

- If two different worlds, do they interact?
- If so, by what rules?

Personal Use

- My own code is dreadfully boring
- Generators for iteration: Yes.
- Everything else: Threads, recursion, etc. (sorry)
- Nevertheless: There may be something to all of this advanced coroutine/generator business

A Bit More Information


Recipes for Mastering Python 3
3rd Edition
Python Cookbook
O'REILLY
David Beazley & Brian K. Jones

Thanks!

- I hope you got some new ideas
- Please feel free to contact me

@dabeaz (Twitter)

http://www.dabeaz.com

- Also, I teach Python classes (shameless plug)
- Special Thanks:

Brian Curtin, Ken Izzo, George Kappel, Christian Long,
Michael Prentiss, Vladimir Urazov, Guido van Rossum