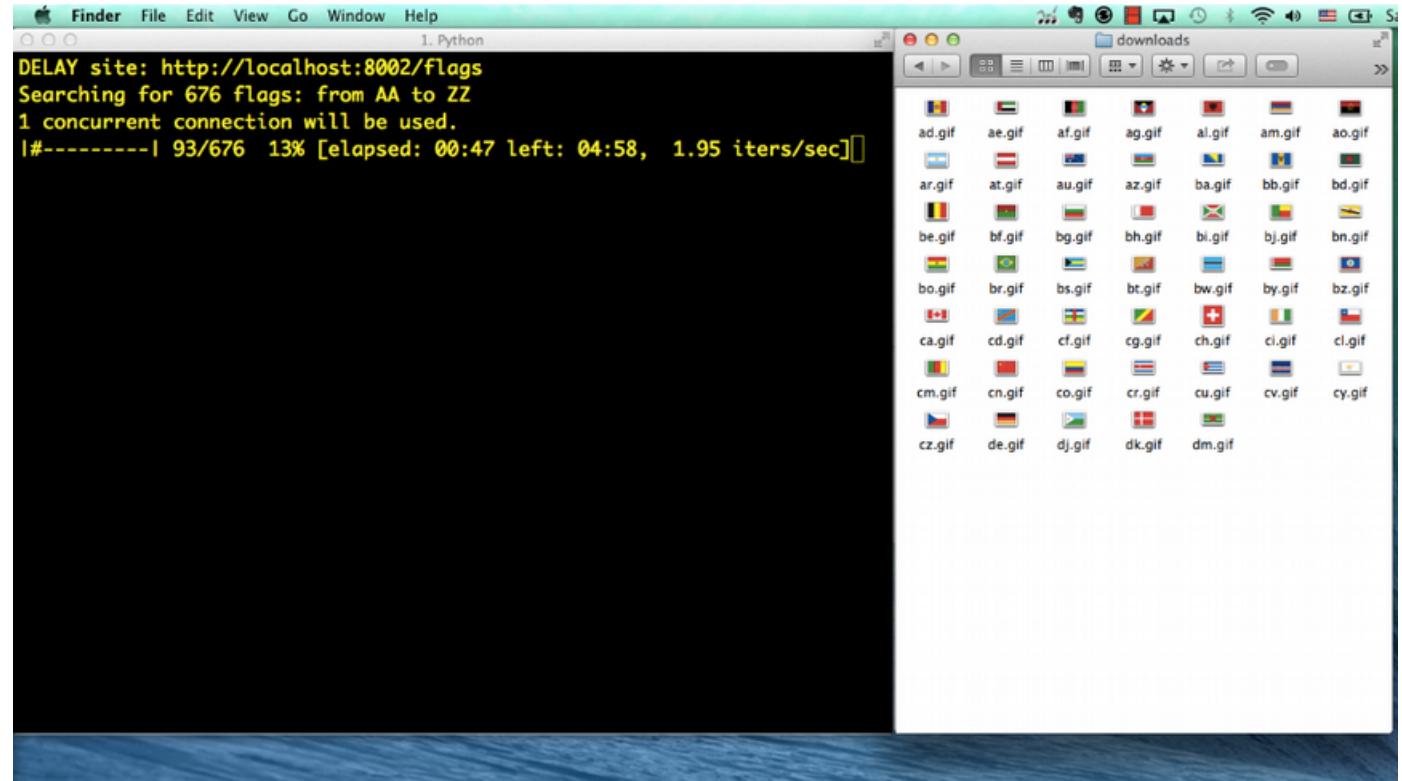


PLATE SPINNING: MODERN CONCURRENCY in PYTHON



Demo: downloading images

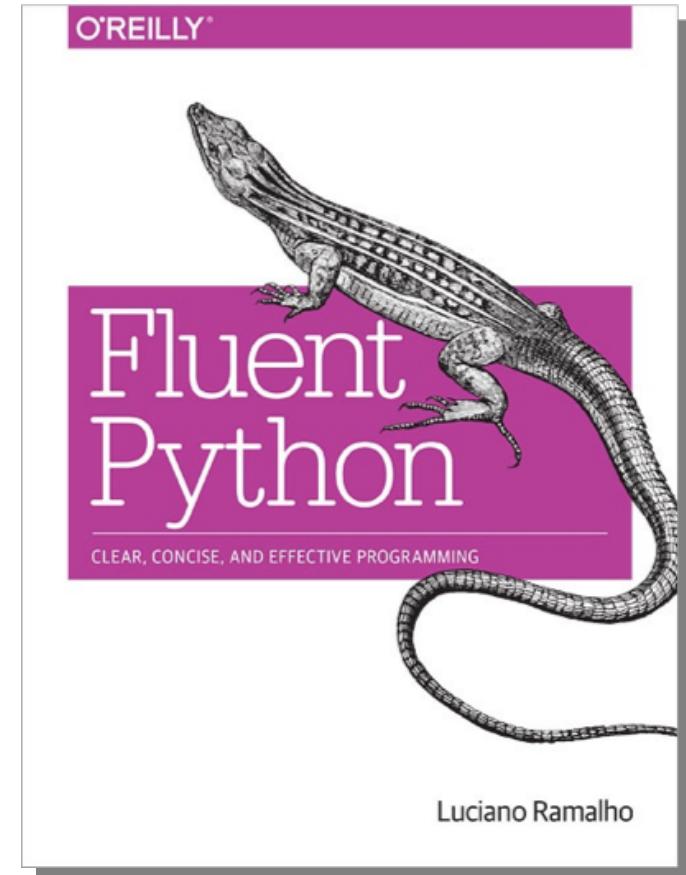
- Hitting 676 URLs, getting 194 flag pictures
- Sequential: 1.92 items/s
- Asynchronous: 150 items/s



<http://www.youtube.com/watch?v=M8Z65tAl5I4>

How the demo was made

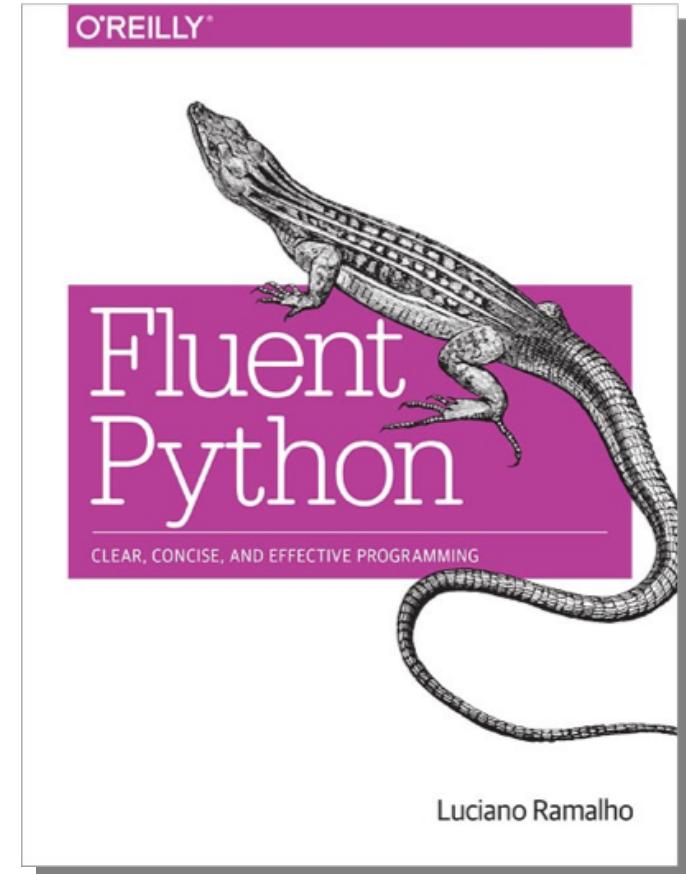
- Three versions of the script:
 - sequential
 - threaded using `concurrent.futures.ThreadPoolExecutor`
 - asynchronous using `asyncio` with `yield/from`
- Test harness:
 - local nginx server + vaurien proxy
- Full instructions on chapters 17 and 18 of Fluent Python



Pre-requisites

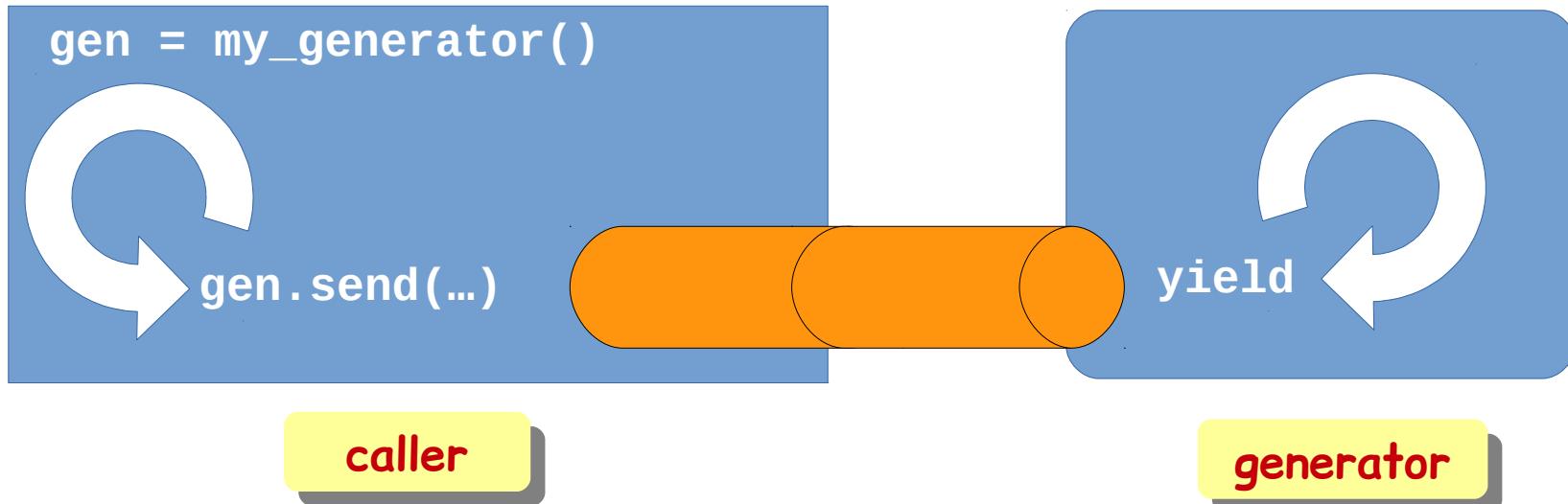
quick review
next...

- You should know how a Python generator function works
- Chapters 14 and 16 of Fluent Python cover this in detail.
- Tip: understand generators well before studying coroutines
- Otherwise: just relax and enjoy the high level overview



Generator: quick review

- Generator: any function that has the **yield** keyword in its body
- Caller sends values **or** generator yields values
- Most important: their progress is synchronized (e.g. loops in sync)



Spinner scripts demo

```
(.env35b3) $ python spinner_thread.py
spinner object: <Thread(Thread-1, initial)>
Answer: 42
(.env35b3) $ python spinner_yield.py
spinner object: <Task pending coro=<spin() running at spinner_yield.py:6>>
Answer: 42
```

Threaded spinner script: overview

- Uses threading library
- Main thread blocks waiting for slow function while spinner thread runs

```
import threading
import itertools
import time
import sys

class Signal: # Ⓛ
    go = True

def spin(msg, signal): # Ⓜ
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\''):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status)) # Ⓝ
        time.sleep(.1)
        if not signal.go: # Ⓞ
            break
    write(' ' * len(status) + '\x08' * len(status)) # Ⓟ

def slow_function(): # Ⓠ
    # pretend waiting a long time for I/O
    time.sleep(3) # Ⓡ
    return 42

def supervisor(): # Ⓢ
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner) # Ⓣ
    spinner.start() # Ⓤ
    result = slow_function() # Ⓥ
    signal.go = False # Ⓦ
    spinner.join() # Ⓧ
    return result

def main():
    result = supervisor() # Ⓨ
    print('Answer:', result)

if __name__ == '__main__':
    main()
```

spinner_thread.py

Thread... spinner bottom

- (K) Supervisor starts spinner thread
- (L) Calls `slow_function`, which blocks at (H)
- (M) Uses signal object to tell spinner thread to stop

```
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
def slow_function(): # G
    # pretend waiting a long time for I/O
    time.sleep(3) # H
    return 42

def supervisor(): # I
    signal = Signal()
    spinner = threading.Thread(target=spin,
                                args=('thinking!', signal))
    print('spinner object:', spinner) # J
    spinner.start() # K
    result = slow_function() # L
    signal.go = False # M
    spinner.join() # N
    return result

def main():
    result = supervisor() # O
    print('Answer:', result)

if __name__ == '__main__':
    main()
```

Threaded spinner: top

- Ⓐ spin gets Signal instance as second argument
- Ⓑ itertools.cycle() produces endless sequence of | / - \
- Ⓒ write backspaces ('\x08'), then sleep for 0.1s
- Ⓓ exit infinite loop if signal.go is False

```
1 import threading
2 import itertools
3 import time
4 import sys
5
6
7 class Signal: # Ⓛ
8     go = True
9
10
11 def spin(msg, signal): # Ⓜ
12     write, flush = sys.stdout.write, sys.stdout.flush
13     for char in itertools.cycle('|/-\\'): # Ⓝ
14         status = char + ' ' + msg
15         write(status)
16         flush()
17         write('\x08' * len(status)) # Ⓞ
18         time.sleep(.1)
19         if not signal.go: # Ⓟ
20             break
21         write(' ' * len(status) + '\x08' * len(status)) # Ⓠ
22
23
24 def slow_function(): # Ⓡ
25     # pretend waiting a long time for I/O
26     time.sleep(3) # Ⓢ
27     return 42
28
29
30 def supervisor(): # Ⓣ
```

Threaded spinner script: notes

- OS thread scheduler may switch active threads at any time – that's why threads cannot be cancelled from the outside
- Calling sleep() or I/O functions practically guarantees a switch
- **Every** standard library function that does I/O releases the GIL, allowing other Python bytecode to run

```
import threading
import itertools
import time
import sys

class Signal: # ①
    go = True

def spin(msg, signal): # ②
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'): # ③
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status)) # ④
        time.sleep(.1)
        if not signal.go: # ⑤
            break
    write(' ' * len(status) + '\x08' * len(status)) # ⑥

def slow_function(): # ⑦
    # pretend waiting a long time for I/O
    time.sleep(3) # ⑧
    return 42

def supervisor(): # ⑨
    signal = Signal()
    spinner = threading.Thread(target=spin,
                                args=('thinking!', signal))
    print('spinner object:', spinner) # ⑩
    spinner.start() # ⑪
    result = slow_function() # ⑫
    signal.go = False # ⑬
    spinner.join() # ⑭
    return result

def main():
    result = supervisor() # ⑮
    print('Answer:', result)

if __name__ == '__main__':
    main()
```

spinner_asyncio.py

Coroutine spinner script: `yield from`

- Uses `asyncio` library
- Main thread (the *only* thread) starts event loop to drive coroutines
- **supervisor**, **spin** and **slow_function** are coroutines
- Coroutines wait for results from other coroutines using `yield from`

```
import asyncio
import itertools
import sys

@asyncio.coroutine # Ⓐ
def spin(msg): # Ⓑ
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))
    try:
        yield from asyncio.sleep(.1) # Ⓒ
    except asyncio.CancelledError: # Ⓓ
        break
    write(' ' * len(status) + '\x08' * len(status))

@asyncio.coroutine
def slow_function(): # Ⓔ
    # pretend waiting a long time for I/O
    yield from asyncio.sleep(3) # Ⓕ
    return 42

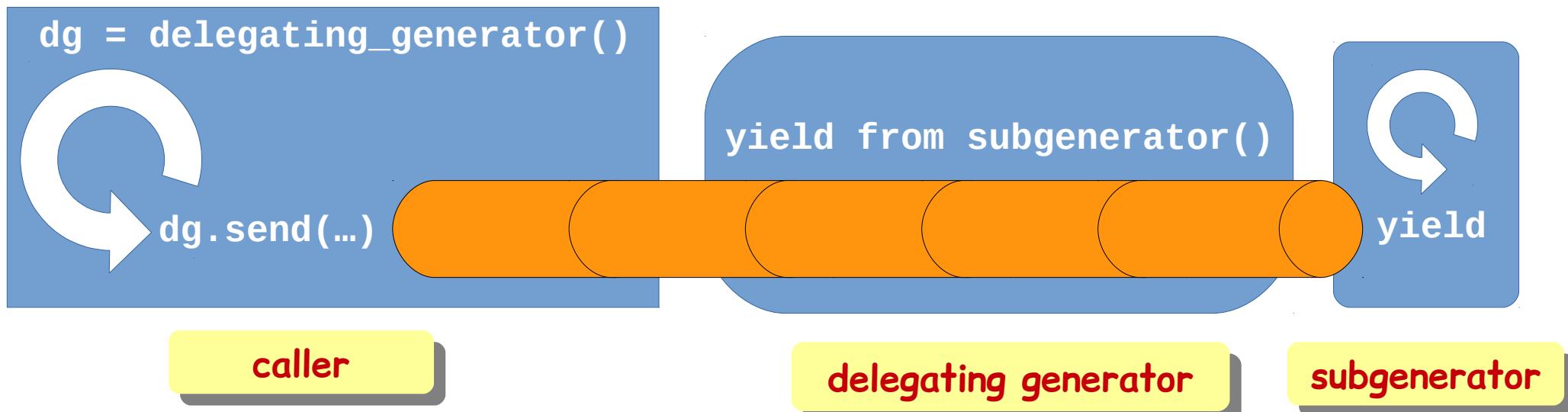
@asyncio.coroutine
def supervisor(): # Ⓖ
    spinner = asyncio.ensure_future(spin('thinking!')) # Ⓗ
    print('spinner object:', spinner) # Ⓘ
    result = yield from slow_function() # Ⓙ
    spinner.cancel() # ⓽
    return result

def main():
    loop = asyncio.get_event_loop() # Ⓛ
    result = loop.run_until_complete(supervisor()) # Ⓜ
    loop.close()
    print('Answer:', result)

if __name__ == '__main__':
    main()
```

yield/from concepts

- PEP-380: Syntax for Delegating to a Subgenerator



Coro... spinner: bottom

- Ⓜ Drive supervisor coroutine with event loop
- Ⓗ Schedule Task with spin coroutine
- Ⓡ Wait for result from slow_function

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
@asyncio.coroutine
def slow_function(): # Ⓛ
    # pretend waiting a long time for I/O
    yield from asyncio.sleep(3) # Ⓜ
    return 42

@asyncio.coroutine
def supervisor(): # Ⓝ
    spinner = asyncio.ensure_future(spin('thinking!')) # Ⓞ
    print('spinner object:', spinner) # Ⓟ
    result = yield from slow_function() # Ⓠ
    spinner.cancel() # Ⓡ
    return result

def main():
    loop = asyncio.get_event_loop() # Ⓢ
    result = loop.run_until_complete(supervisor()) # Ⓣ
    loop.close()
    print('Answer:', result)

if __name__ == '__main__':
    main()
```

Coro... spinner: bottom

- Ⓜ Drive supervisor coroutine with event loop
- Ⓗ Schedule Task with spin coroutine
- Ⓡ Wait for result from slow_function

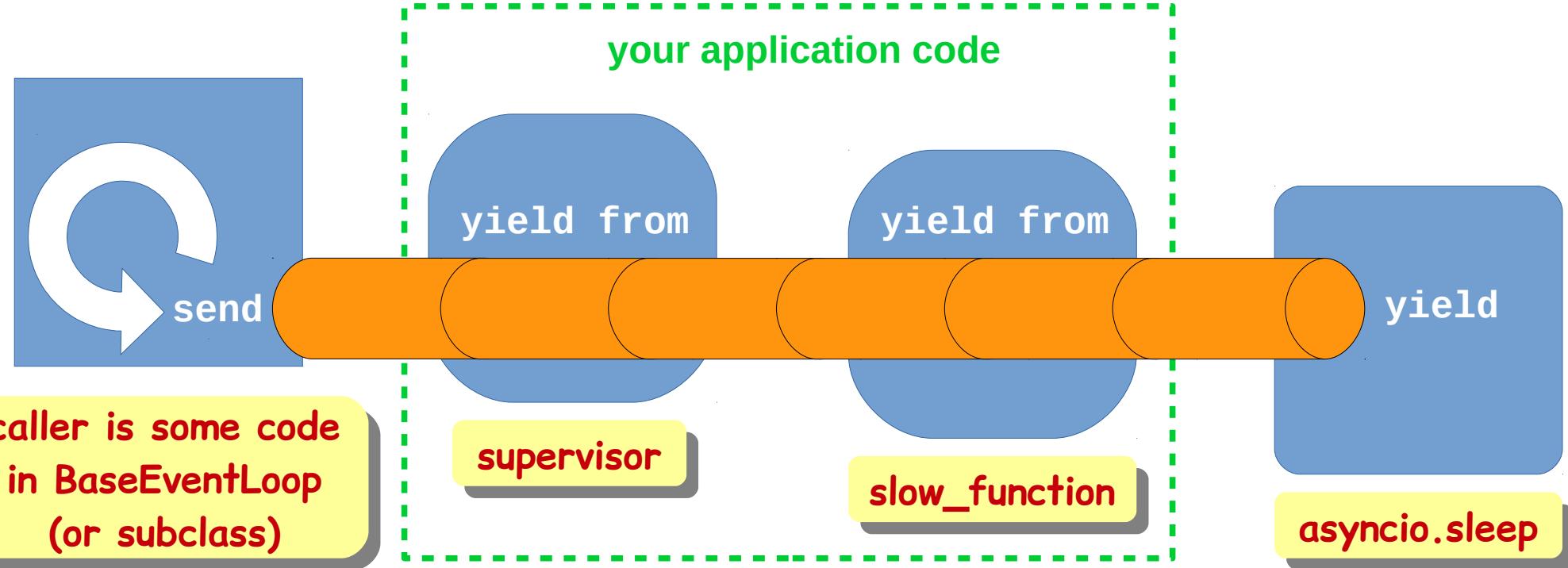
```
20  
21 @asyncio.coroutine  
22 def slow_function(): # Ⓛ  
23     # pretend waiting a long time for I/O  
24     yield from asyncio.sleep(1)  
25     return 42  
26  
27  
28 @asyncio.coroutine  
29 def supervisor(): # Ⓜ  
30     spinner = asyncio.ensure_future(spin('thinking!')) # Ⓝ  
31     print('spinner object:', spinner) # Ⓞ  
32     result = yield from slow_function() # Ⓟ  
33     spinner.cancel() # Ⓠ  
34     return result  
35  
36  
37 def main():  
38     loop = asyncio.get_event_loop() # Ⓡ  
39     result = loop.run_until_complete(supervisor()) # Ⓢ  
40     loop.close()  
41     print('Answer:', result)  
42  
43  
44 if __name__ == '__main__':  
45     main()
```

⌚3.4.3: `asyncio.async`
⌚3.4.4: `asyncio.ensure_future`



`yield/from` creates channel

- Channel connects event loop with last subgenerator in the delegating chain



Coro... spinner: bottom

- Ⓜ Drive supervisor coroutine with event loop
- Ⓗ Schedule Task with spin coroutine
- Ⓡ Wait for result from slow_function

```
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
  
@asyncio.coroutine  
def slow_function(): # Ⓛ  
    # pretend waiting a long time for I/O  
    yield from asyncio.sleep(3) # Ⓜ  
    return 42  
  
  
@asyncio.coroutine  
def supervisor(): # Ⓝ  
    spinner = asyncio.ensure_future(spin('thinking!')) # Ⓞ  
    print('spinner object:', spinner) # Ⓟ  
    result = yield from slow_function() # Ⓠ  
    spinner.cancel() # Ⓡ  
    return result  
  
  
def main():  
    loop = asyncio.get_event_loop()  
    result = loop.run_until_complete(supervisor()) # Ⓢ  
    loop.close()  
    print('Answer:', result)  
  
  
if __name__ == '__main__':  
    main()
```



**yield from blocks
delegating generator
(supervisor)**

Coro... spinner: bottom

- Ⓜ Drive supervisor coroutine with event loop
- Ⓗ Schedule Task with spin coroutine
- Ⓡ Wait for result from slow_function

```
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
  
@asyncio.coroutine  
def slow_function(): # Ⓛ  
    # pretend waiting a long time for I/O  
    yield from asyncio.sleep(3) # Ⓜ  
    return 42  
  
  
@asyncio.coroutine  
def supervisor(): # Ⓝ  
    spinner = asyncio.ensure_future(spin('thinking!')) # Ⓞ  
    print('spinner object:', spinner) # Ⓟ  
    result = yield from slow_function() # Ⓠ  
    spinner.cancel() # Ⓡ  
    return result  
  
  
def main():  
    loop = asyncio.get_event_loop()  
    result = loop.run_until_complete(supervisor()) # Ⓢ  
    loop.close()  
    print('Answer:', result)  
  
  
if __name__ == '__main__':  
    main()
```

slow_function is the subgenerator in this context

Coro... spinner: bottom

⑤ Delegate to
asyncio.sleep

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
@asyncio.coroutine
def slow_function(): # ⑥
    # pretend waiting a long time for I/O
    yield from asyncio.sleep(3) # ⑦
    return 42
@asyncio.coroutine
def supervisor(): # ⑧
    spinner = asyncio.ensure_future(slow_function())
    print('spinner object:', spinner) # ⑨
    result = yield from spinner # ⑩
    spinner.cancel() # ⑪
    return result
def main():
    loop = asyncio.get_event_loop() # ⑫
    result = loop.run_until_complete(supervisor()) # ⑬
    loop.close()
    print('Answer:', result)
if __name__ == '__main__':
    main()
```

slow_function is the delegating generator here

Coro... spinner: bottom

⑤ Delegate to
asyncio.sleep

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
@asyncio.coroutine
def slow_function(): # ⑥
    # pretend waiting a long time for I/O
    yield from asyncio.sleep(3) # ⑦
    return 42
@asyncio.coroutine
def supervisor(): # ⑧
    spinner = asyncio.ensure_future(spin('thinking!')) # ⑨
    print('spinner object:', spinner) # ⑩
    result = yield from slow_function() # ⑪
    spinner.cancel() # ⑫
    return result
def main():
    loop = asyncio.get_event_loop() # ⑬
    result = loop.run_until_complete(supervisor()) # ⑭
    loop.close()
    print('Answer:', result)
if __name__ == '__main__':
    main()
```

asyncio.sleep is the subgenerator here

Coro... spinner: bottom

⑤ Delegate to
asyncio.sleep

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
@asyncio.coroutine
def slow_function(): # ⑥
    # pretend waiting a long time for I/O
    yield from asyncio.sleep(3) # ⑦
    return 42
@asyncio.coroutine
def supervisor(): # ⑧
    spinner = asyncio.ensure_future(spin('thinking!')) # ⑨
    print('spinner object:', spinner) # ⑩
    result = yield from slow_function() # ⑪
    spinner.cancel() # ⑫
    return result
def main():
    loop = asyncio.get_event_loop() # ⑬
    result = loop.run_until_complete(supervisor()) # ⑭
    loop.close()
    print('Answer:', result)
if __name__ == '__main__':
    main()
```

yield from blocks
slow_function

Coro spinner bottom

⑤ Delegate to
asyncio.sleep

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
@asyncio.coroutine
def slow_function(): # ⑥
    # pretend waiting a long time for I/O
    yield from asyncio.sleep(3) # ⑦
    return 42

@asyncio.coroutine
def supervisor(): # ⑧
    spinner = asyncio.Task(
        print('spinner created'), # ⑨
        result = yield from spinner # ⑩
    )
    spinner.cancel() # ⑪
    return result

def main():
    loop = asyncio.get_event_loop() # ⑫
    result = loop.run_until_complete(supervisor()) # ⑬
    loop.close()
    print('Answer:', result)

if __name__ == '__main__':
    main()
```

asyncio.sleep() sets up a timer with loop.call_later, then yields to the main loop

Coro spinner bottom

- Ⓜ Drive supervisor coroutine with event loop
- Ⓗ Schedule Task with spin coroutine
- Ⓡ Attach slow_function to event loop

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
@asyncio.coroutine
def slow_function(): # Ⓛ
    # pretend waiting a long time for I/O
    yield from asyncio.sleep(3) # Ⓜ
    return 42
@asyncio.coroutine
def supervisor(): # Ⓝ
    spinner = asyncio.event_loop().spawn_subprocess_exec(
        'sh', '-c', 'stty sane')
    print('spinner object:', spinner)
    result = yield from slow_function() # Ⓞ
    spinner.cancel() # Ⓟ
    return result
def main():
    loop = asyncio.get_event_loop() # Ⓠ
    result = loop.run_until_complete(supervisor()) # Ⓡ
    loop.close()
    print('Answer:', result)
if __name__ == '__main__':
    main()
```

when subgenerator returns,
delegating generator resumes
at yield from

Coro... spinner bottom

K After
slow_function
returns, cancel
spinner Task

```
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
  
@asyncio.coroutine  
def slow_function(): # E  
    # pretend waiting a long time for I/O  
    yield from asyncio.sleep(3) # F  
    return 42  
  
  
@asyncio.coroutine  
def supervisor(): # G  
    spinner = asyncio.ensure_future(spin('thinking!')) # H  
    print('spinner object:', spinner) # I  
    result = yield from slow_function() # J  
    spinner.cancel() # K  
    return result  
  
  
def main():  
    loop = asyncio.get_event_loop()  
    result = loop.run_until_complete(  
        loop.close()  
    print('Answer:', result)  
  
  
if __name__ == '__main__':  
    main()
```

Tasks can be
cancelled because
cancellation happens
only at yield points

Coroutine spinner: top

- Ⓐ Each iteration waits for `asyncio.sleep(.1)` to and...
- Ⓑ Handles cancellation by terminating infinite loop, then clearing the status line

```
1 import asyncio
2 import itertools
3 import sys
4
5
6 @asyncio.coroutine # Ⓐ
7 def spin(msg): # Ⓑ
8     write, flush = sys.stdout.write, sys.stdout.flush
9     for char in itertools.cycle('|/-\\'):
10        status = char + ' ' + msg
11        write(status)
12        flush()
13        write('\x08' * len(status))
14        try:
15            yield from asyncio.sleep(.1) # Ⓒ
16        except asyncio.CancelledError: # Ⓓ
17            break
18        write(' ' * len(status) + '\x08' * len(status))
19
20
21 @asyncio.coroutine
22 def slow_function(): # Ⓔ
23     # pretend waiting a long time for I/O
24     yield from asyncio.sleep(3) # Ⓕ
25     return 42
26
27
28 @asyncio.coroutine
29 def supervisor(): # Ⓖ
```

Threaded x async main

```
def main():
    result = supervisor() # O
    print('Answer:', result)
```

threaded

```
def main():
    loop = asyncio.get_event_loop() # L
    result = loop.run_until_complete(supervisor()) # M
    loop.close()
    print('Answer:', result)
```

async

- async main manages the event loop
- note how supervisor() is called in each version

Threaded x async supervisor

```
def supervisor(): # I
    signal = Signal()
    spinner = threading.Thread(target=spin,
                                args=('thinking!', signal))
    print('spinner object:', spinner) # J
    spinner.start() # K
    result = slow_function() # L
    signal.go = False # M
    spinner.join() # N
    return result
```

threaded

```
@asyncio.coroutine
def supervisor(): # G
    spinner = asyncio.ensure_future(spin('thinking!')) # H
    print('spinner object:', spinner) # I
    result = yield from slow_function() # J
    spinner.cancel() # K
    return result
```

async

Threaded x async comparison

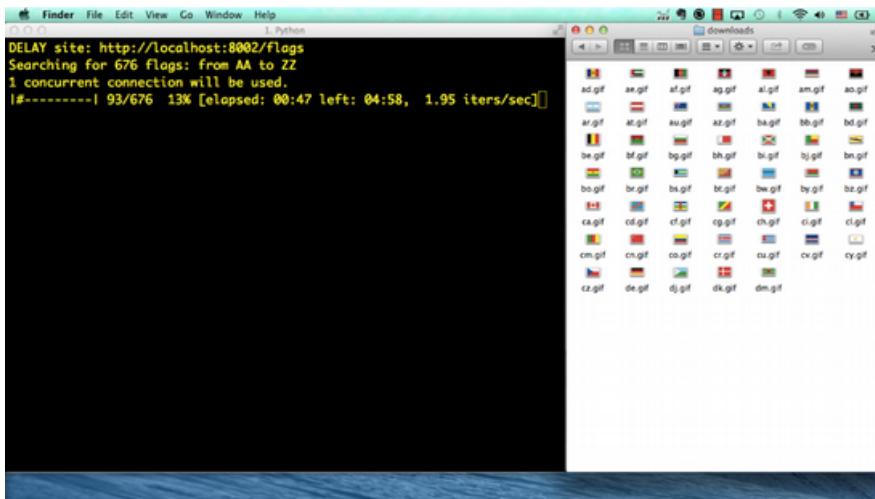
```
(.env35b3) $ python spinner_thread.py
spinner object: <Thread(Thread-1, initial)>
Answer: 42
(.env35b3) $ python spinner_yield.py
spinner object: <Task pending coro=<spin() running at spinner_yield.py:6>>
Answer: 42
```

- spinner activity implemented as Thread or Task
- async Task is similar to a green thread (an application-level thread)
- Task wraps a coroutine
- Each coroutine uses much less memory than an OS thread (kilobytes, not megabytes)

flags_asyncio.py

flags_asyncio.py

- Simplified implementation of demo script



```
import asyncio
import aiohttp    ①
from flags import BASE_URL, save_flag, show, main    ②

async def get_flag(cc):    ③
    url = '{}/{}{}.gif'.format(BASE_URL, cc.lower())
    resp = yield from aiohttp.request('GET', url)    ④
    image = yield from resp.read()    ⑤
    return image

@asyncio.coroutine
def download_one(cc):    ⑥
    image = yield from get_flag(cc)    ⑦
    show(cc)
    save_flag(image, cc.lower() + '.gif')
    return cc

def download_many(cc_list):    ⑧
    loop = asyncio.get_event_loop()
    to_do = [download_one(cc) for cc in sorted(cc_list)]    ⑨
    wait_coro = asyncio.wait(to_do)
    res, _ = loop.run_until_complete(wait_coro)    ⑪
    loop.close()    ⑫

    return len(res)

if __name__ == '__main__':
    main(download_many)
```

Zoom in...

- **download_many** schedules many instances of **download_one**
- **download_one** delegates to **get_flag**
- **get_flag** delegates to **aiottp.request()** and **response.read()**

```
@asyncio.coroutine    ③
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)    ④
    image = yield from resp.read()      ⑤
    return image

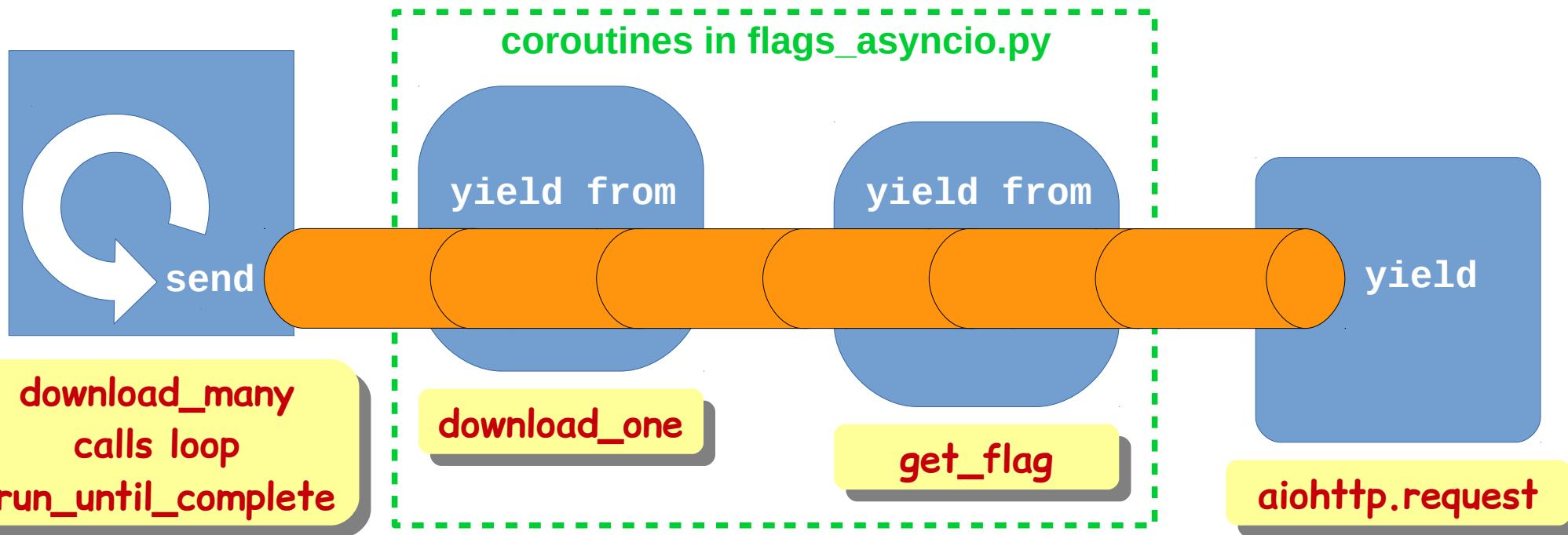
@asyncio.coroutine
def download_one(cc):    ⑥
    image = yield from get_flag(cc)    ⑦
    show(cc)
    save_flag(image, cc.lower() + '.gif')
    return cc

def download_many(cc_list):
    loop = asyncio.get_event_loop()    ⑧
    to_do = [download_one(cc) for cc in sorted(cc_list)]    ⑨
    wait_coro = asyncio.wait(to_do)
    res, _ = loop.run_until_complete(wait_coro)    ⑪
    loop.close()    ⑫

    return len(res)
```

yield/from in action

- User code creates chain of coroutines connecting event loop to library functions that perform asynchronous I/O



Zoom further...

```
@asyncio.coroutine
def get_flag(cc):
    url = '{}/{}/{}/{}.gif'.format(BASE_URL, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)
    image = yield from resp.read()
    return image
```

- Guido van Rossum's tip for reading async code:
 - squint and ignore **yield from** for a moment...

Zoom further...

```
@asyncio.coroutine
def get_flag(cc):
    url = '{}/{}/{}/{}.gif'.format(BASE_URL, cc=cc.lower())
    resp = aiohttp.request('GET', url)
    image =
    return image
```

- Guido van Rossum's tip for reading async code:
 - squint and ignore **yield from** for a moment...

What is the point?

- Concurrency is always hard
- Python's new **asyncio** library and language features provide an effective alternative to:
 - managing threads and locks by hand
 - coping with callback hell

Callback hell in JavaScript

```
api_call1(request1, function (response1) {  
    // stage 1  
    var request2 = step1(response1);  
  
    api_call2(request2, function (response2) {  
        // stage 2  
        var request3 = step2(response2);  
  
        api_call3(request3, function (response3) {  
            // stage 3  
            step3(response3);  
        });  
    });  
});
```

context from stage 1 is gone

context from stage 2 is gone

Callback hell in Python

```
def stage1(response1):
    request2 = step1(response1)
    api_call2(request2, stage2)
```

```
def stage2(response2):
    request3 = step2(response2)
    api_call3(request3, stage3)
```

```
def stage3(response3):
    step3(response3)
```

```
api_call1(request1, stage1)
```

context from
stage 1 is gone

context from
stage 2 is gone

Escape from callback hell

```
@asyncio.coroutine
def three_stages(request1):
    response1 = yield from api_call1(request1)
    # stage 1
    request2 = step1(response1)
    response2 = yield from api_call2(request2)
    # stage 2
    request3 = step2(response2)
    response3 = yield from api_call3(request3)
    # stage 3
    step3(response3)

# schedule execution
loop.create_task(three_stages(request1))
```

context is preserved through all stages: it's all in the local scope of the delegating coroutine

Before considering
another language for
asynchronous jobs,
try Python 3.4 or 3.5!

Python 3.5 `async/await`

- New keywords introduced for the first time since Python 3.0 (2008)
- Very briefly:
 - **`async def`** is used to declare coroutines (yay!)
 - `asyncio.async()` function is now called `asyncio.ensure_future()`
 - **`await`** is used to delegate to subgenerators (hooray!)
 - other new constructs:
 - **`async for`**
 - **`async with`**
- Proper language support for coroutines, finally!

```
1 import asyncio
2 import itertools
3 import sys
4
5
6 @asyncio.coroutine # Ⓐ
7 def spin(msg): # Ⓑ
8     write, flush = sys.stdout.write,
9     sys.stdout.flush
10    for char in itertools.cycle('|/-\\'):
11        status = char + ' ' + msg
12        write(status)
13        flush()
14        write('\x08' * len(status))
15        try:
16            yield from asyncio.sleep(.1)
17        except asyncio.CancelledError: #
18            break
19        write(' ' * len(status) + '\x08' *
20              len(status))
21
22 @asyncio.coroutine
23 def slow_function(): # Ⓒ
24     # pretend waiting a long time for I/O
25     yield from asyncio.sleep(3) # Ⓓ
26
27     return 42
```

```
1 import asyncio
2 import itertools
3 import sys
4
5 # Ⓛ
6 async def spin(msg): # Ⓜ
7     write, flush = sys.stdout.write,
8     sys.stdout.flush
9     for char in itertools.cycle('|/-\\'):
10        status = char + ' ' + msg
11        write(status)
12        flush()
13        write('\x08' * len(status))
14        try:
15            await asyncio.sleep(.1) # Ⓝ
16        except asyncio.CancelledError: # Ⓞ
17            break
18        write(' ' * len(status) + '\x08' *
19        len(status))
20
21 async def slow_function(): # Ⓟ
22     # pretend waiting a long time for I/O
23     await asyncio.sleep(3) # Ⓠ
24
25     return 42
```

Summary

O'REILLY®

OSCON

- Concurrent I/O can be achieved without threads or callback hell
 - no threads or callbacks in **your** code, at least
- Asyncio **Task** instances wrap coroutines
 - allow cancellation, waiting for result and status checks
- Coroutines driven with **yield from** or **await** behave as cooperative lightweight threads
 - explicit switching points make it easier to reason and debug
 - thousands of coroutines can be scheduled at once thanks to low memory cost

Links



- **Fluent Python** example code repository:
 - <https://github.com/fluentpython/example-code>
 - new async/await example in directory 17-futures/countries/
 - new directory 18b-async-await with 18-asyncio examples rewritten with new syntax
- Slides for this talk (and others):
 - <https://speakerdeck.com/ramalho/>
- Please rate this talk:
 - <http://www.oscon.com/open-source-2015/public/schedule/detail/41211>
- Me:
 - Twitter: @ramalhoorg