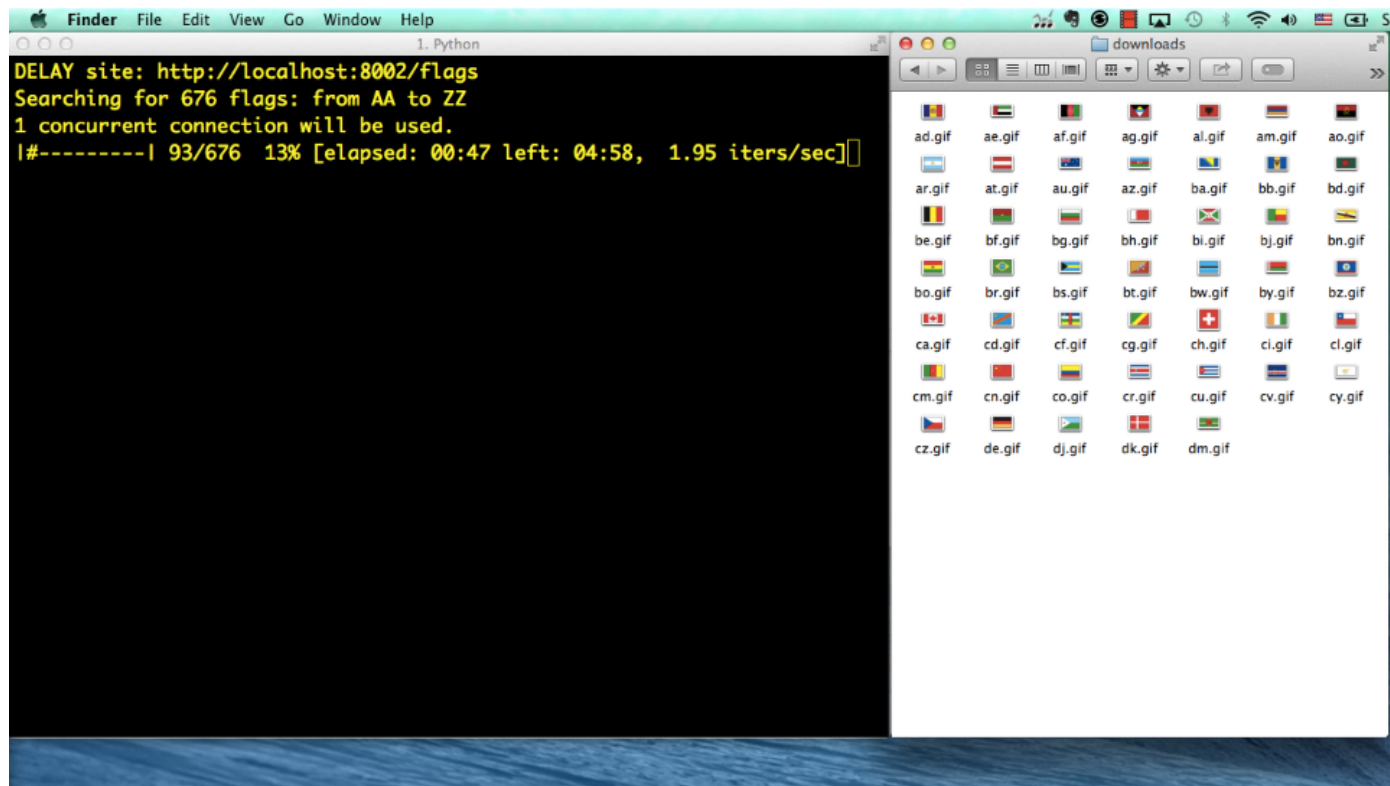


CONCURREN- NCY WITH PYTHON 3.5 ASYNC & AWAIT



Demo: downloading images

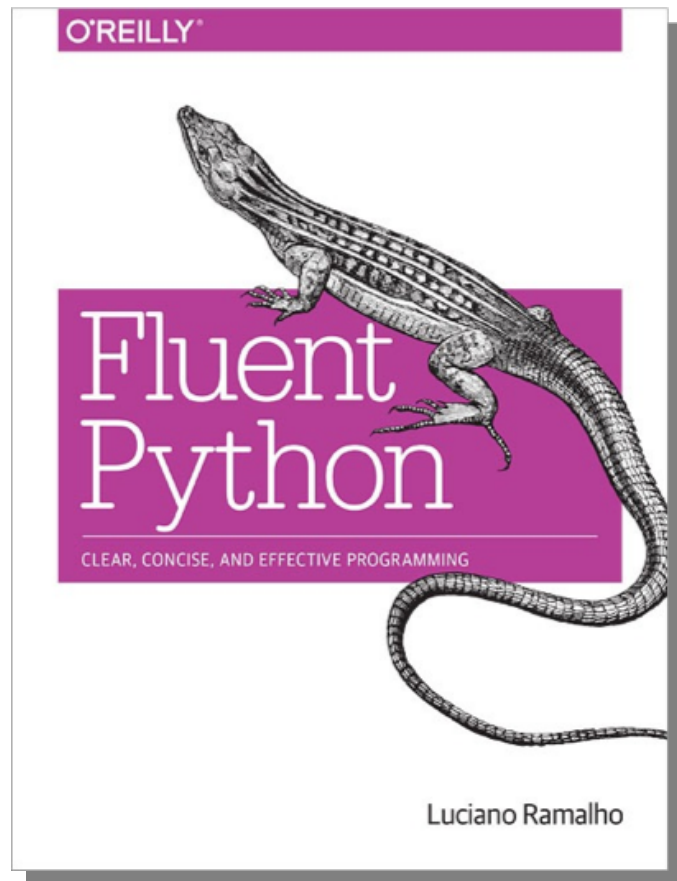
- Hitting 676 URLs, getting 194 flag pictures
- Sequential: 1.92 items/s
- Asynchronous: 150 items/s



<http://www.youtube.com/watch?v=M8Z65tA1514>

How the demo was made

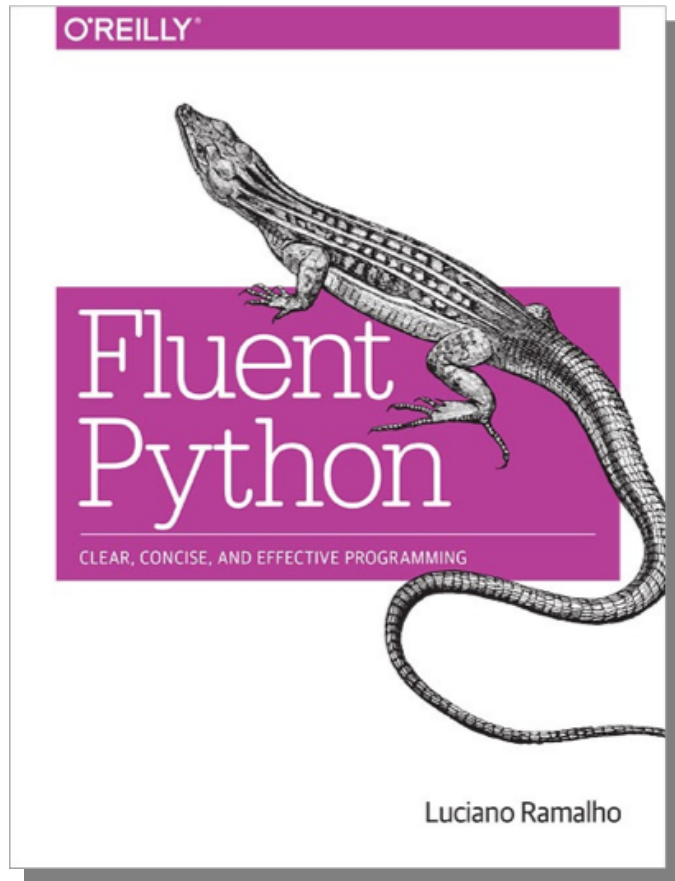
- Four versions of the script:
 - sequential
 - threaded using **`concurrent.futures.ThreadPoolExecutor`**
 - asynchronous using **`asyncio`**: with **`yield from`**
 - asynchronous using **`asyncio`**: with **`await`**
- Test harness:
 - local nginx server + vaurien proxy
- Full instructions on chapters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



Pre-requisites

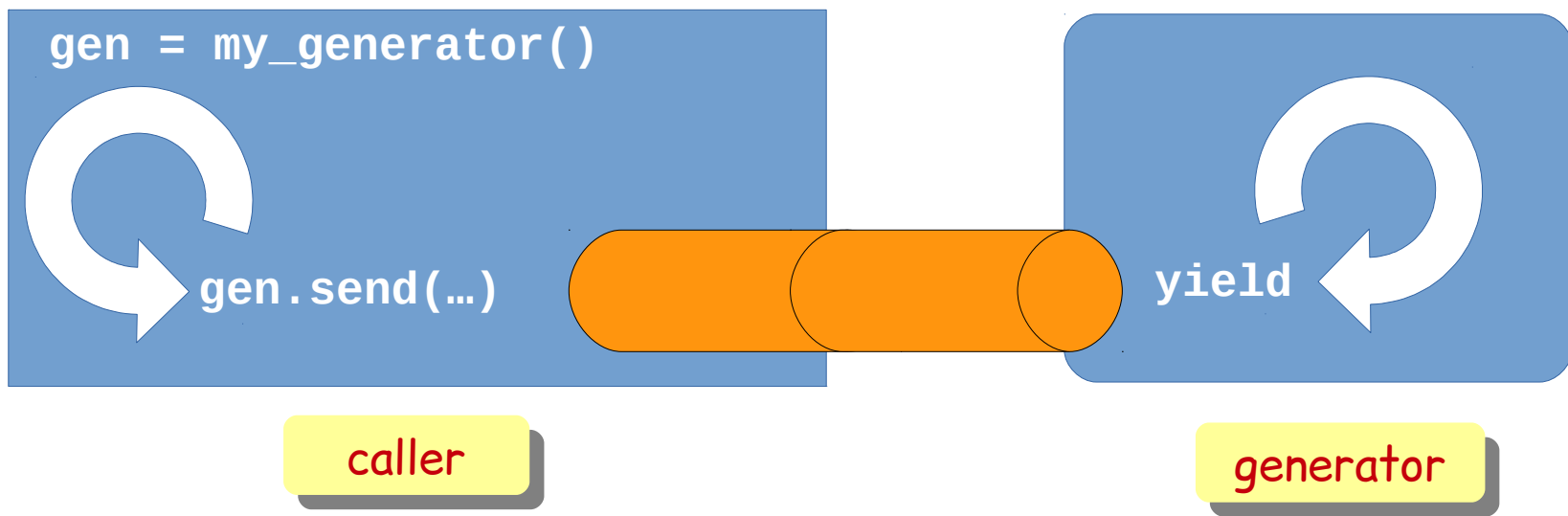
quick review
next...

- 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 *and/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
```

spinner_thread.py

Threaded spinner script: overview

- Uses **threading** library
- Main thread starts **spinner** thread
- Main thread blocks waiting for **slow_function** while **spinner** thread runs
- When **slow_function** returns, main thread signals **spinner** thread to stop

```
import threading
import itertools
import time
import sys

class Signal: # ④
    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)) # ⑤
        time.sleep(.1)
        if not signal.go: # ⑥
            break
    write(' ' * len(status) + '\x08' * len(status)) # ⑦

def slow_function(): # ⑧
    # pretend waiting a long time for I/O
    time.sleep(3) # ⑨
    return 42

def supervisor(): # ①
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner) # ⑩
    spinner.start() # ⑪
    result = slow_function() # ⑬
    signal.go = False # ⑭
    spinner.join() # ⑮
    return result

def main():
    result = supervisor() # ⑯
    print('Answer:', result)

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


Threaded spinner: bottom

- Ⓚ supervisor starts **spinner** thread
- Ⓛ Calls **slow_function**, which blocks at Ⓜ
- Ⓜ Uses **signal** object to tell **spinner** thread to stop

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

sleep() and
I/O functions
release the GIL

```
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()
```

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**

```
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
```

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 to another thread
- *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: # ④
    go = True

def spin(msg, signal): # ⑤
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'): # ⑥
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status)) # ⑦
        time.sleep(.1)
        if not signal.go: # ⑧
            break
    write(' ' * len(status) + '\x08' * len(status)) # ⑨

def slow_function(): # ⑩
    # pretend waiting a long time for I/O
    time.sleep(3) # ⑪
    return 42

def supervisor(): # ⑫
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner) # ⑬
    spinner.start() # ⑭
    result = slow_function() # ⑮
    signal.go = False # ⑯
    spinner.join() # ⑰
    return result

def main():
    result = supervisor() # ⑱
    print('Answer:', result)

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

spinner_await.py

Coroutine spinner script: `async/await`

- 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 **await**

```
import asyncio
import itertools
import sys

# ①
async def spin(msg): # ②
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))
        try:
            await asyncio.sleep(.1) # ③
        except asyncio.CancelledError: # ④
            break
    write(' ' * len(status) + '\x08' * len(status))

async def slow_function(): # ⑤
    # pretend waiting a long time for I/O
    await asyncio.sleep(3) # ⑥
    return 42

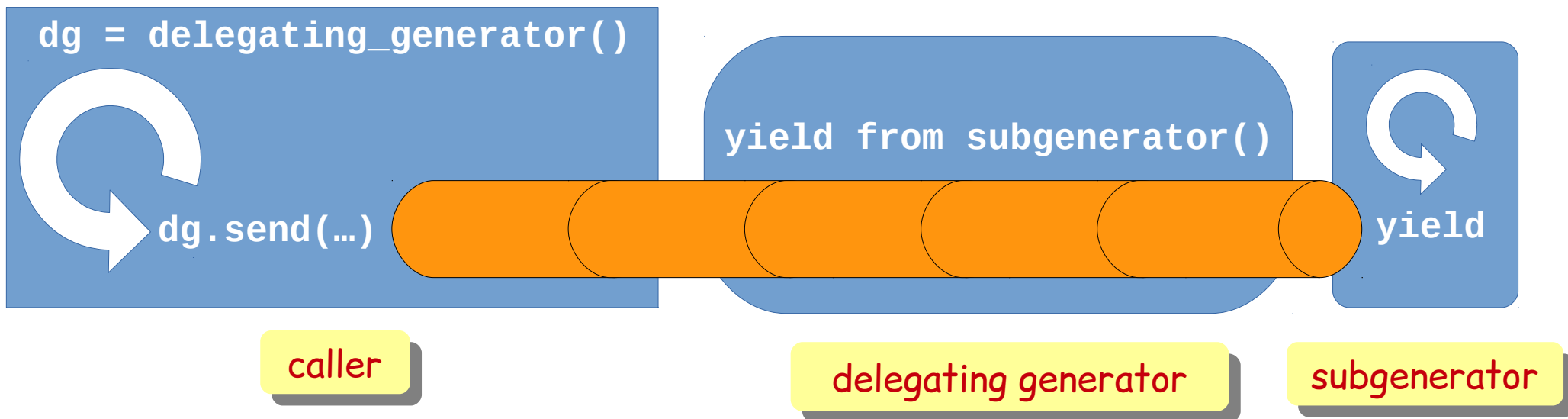
async def supervisor(): # ⑦
    spinner = asyncio.ensure_future(spin('thinking!')) # ⑧
    print('spinner object:', spinner) # ⑨
    result = await slow_function() # ⑩
    spinner.cancel() # ⑪
    return result

def main():
    loop = asyncio.get_event_loop() # ⑬
    result = loop.run_until_complete(supervisor()) # ⑭
    loop.close()
    print('Answer:', result)

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

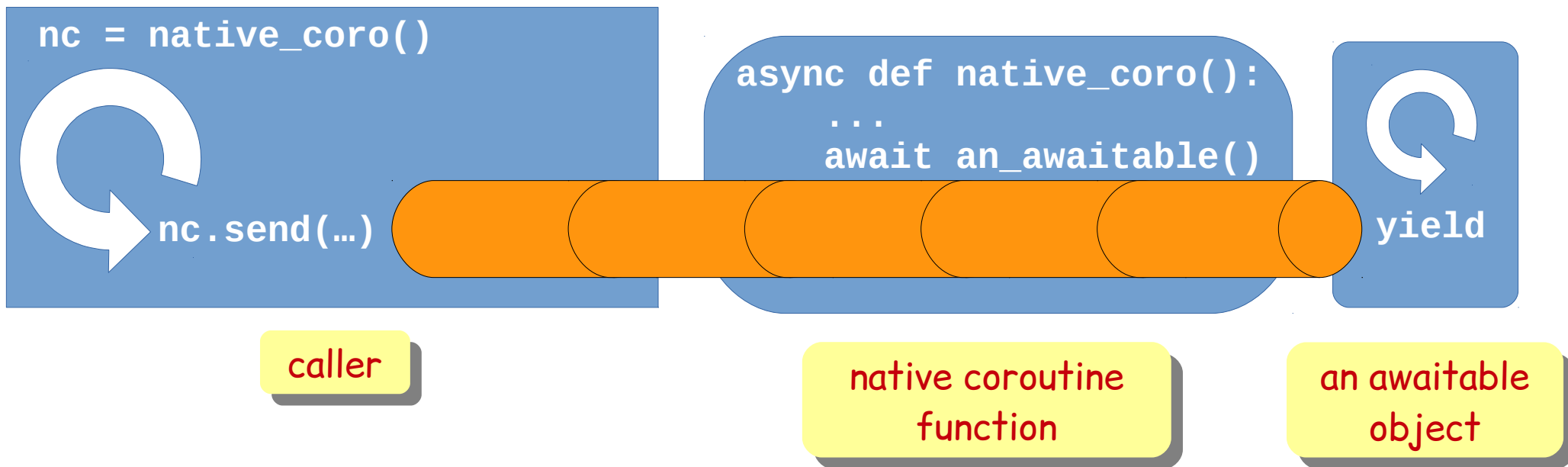
yield-from concepts

- PEP-380: Syntax for Delegating to a Subgenerator



async/await concepts

- PEP-492: Coroutines with async and await syntax
 - introduces *native coroutines* (\neq generator coroutines)



Coro... spinner: bottom

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

```
async def slow_function(): # Ⓜ
    # pretend waiting a long time for I/O
    await asyncio.sleep(3) # Ⓜ
    return 42

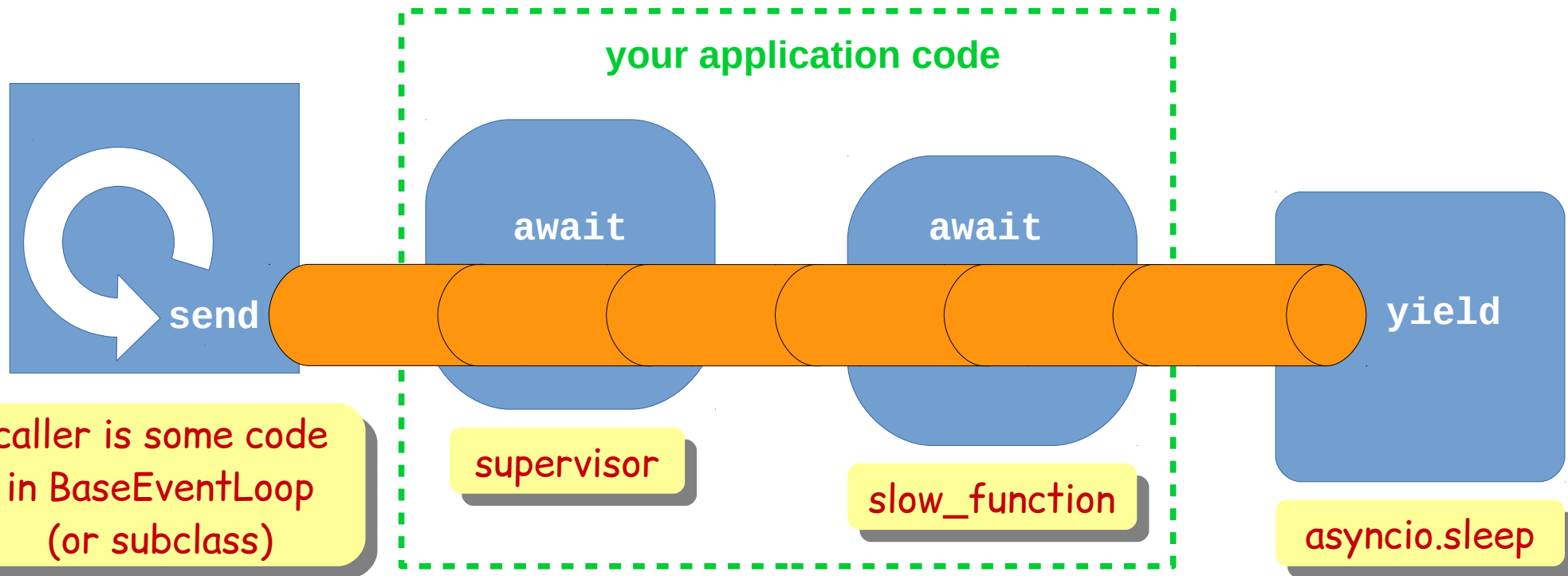
async def supervisor(): # Ⓜ
    spinner = asyncio.ensure_future(spin('thinking!')) # Ⓜ
    print('spinner object:', spinner) # Ⓜ
    result = await slow_function() # Ⓜ
    spinner.cancel() # Ⓜ
    return result

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

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


await creates channel

- Channel connects event loop with last awaitable object in the delegating chain



Coro... spinner: bottom

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

```
async def slow_function(): # Ⓜ  
    # pretend waiting a long time for I/O  
    await asyncio.sleep(3) # Ⓜ  
    return 42
```

```
async def supervisor(): # Ⓜ  
    spinner = asyncio.ensure_future(spin('thinking!')) # Ⓜ  
    print('spinner object:', spinner) # Ⓜ  
    result = await slow_function() # Ⓜ  
    spinner.cancel() # Ⓜ  
    return result
```

await blocks
delegating coroutine
(supervisor)

```
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**

```
async def slow_function(): # Ⓜ  
    # pretend waiting a long time for I/O  
    await asyncio.sleep(3) # Ⓜ  
    return 42
```

```
async def supervisor(): # Ⓜ  
    spinner = asyncio.ensure_future(spin('thinking!')) # Ⓜ  
    print('spinner object:', spinner) # Ⓜ  
    result = await slow_function() # Ⓜ  
    spinner.cancel() # Ⓜ  
    return result
```

slow_function is
driven directly by
the event loop

```
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

Ⓕ Delegate to
asyncio.sleep

```
async def slow_function(): # Ⓔ
    # pretend waiting a long time for I/O
    await asyncio.sleep(3) # Ⓕ
    return 42
```

slow_function
blocks here

```
async def supervisor(): # Ⓖ
    spinner = asyncio.ensure_future(spin('thinking!')) # Ⓗ
    print('spinner object:', spinner) # Ⓘ
    result = await slow_function() # ⓵
    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

Ⓕ Delegate to
`asyncio.sleep`

```
async def slow_function(): # Ⓔ
    # pretend waiting a long time for I/O
    await asyncio.sleep(3) # Ⓕ
    return 42
```

```
async def supervisor():
    spinner = asyncio.
    print('spinner obj
    result = await slow
    spinner.cancel() # Ⓖ
    return result
```

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

```
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


Ⓚ After
slow_function
returns, cancel
spinner task

```
async def slow_function(): # Ⓔ
    # pretend waiting a long time for I/O
    await asyncio.sleep(3) # Ⓕ
    return 42

async def supervisor(): # Ⓖ
    spinner = asyncio.ensure_future(spin('thinking!')) # Ⓖ
    print('spinner object:', spinner) # Ⓘ
    result = await 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()
```



Tasks can be
cancelled because
cancellation happens
only at yield points

Coroutine spinner: top

- Ⓒ Each iteration waits for **asyncio.sleep(.1)** and...
- Ⓓ Handles cancellation by terminating infinite loop, then clearing the status line

```
import asyncio
import itertools
import sys

# Ⓐ
async 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:
            await asyncio.sleep(.1) # Ⓒ
        except asyncio.CancelledError: # Ⓓ
            break
    write(' ' * len(status) + '\x08' * len(status))

async def slow_function(): # Ⓔ
    # pretend waiting a long time for I/O
    await asyncio.sleep(3) # Ⓕ
    return 42
```

Threaded x async main

threaded

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

asynchronous

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

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

Threaded x async supervisor

```
def supervisor(): # ①
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner) # ②
    spinner.start() # ③
    result = slow_function() # ④
    signal.go = False # ⑤
    spinner.join() # ⑥
    return result
```

threaded

```
async def supervisor(): # ⑦
    spinner = asyncio.ensure_future(spin('thinking!')) # ⑧
    print('spinner object:', spinner) # ⑨
    result = await slow_function() # ⑩
    spinner.cancel() # ⑪
    return result
```

asynchronous

Threaded x async comparison

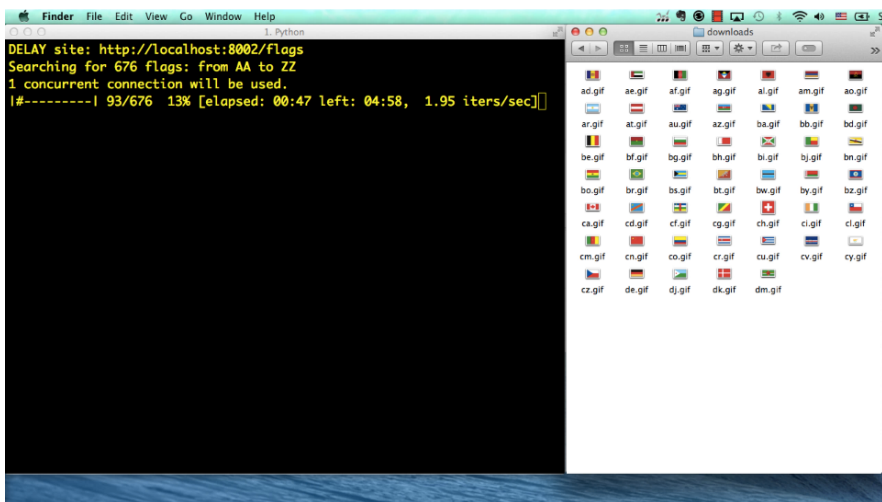
```
(.env35b3) $ python spinner_thread.py  
spinner object: <Thread(Thread-1, initial)>  
Answer: 42  
(.env35b3) $ python spinner_yield.py  
spinner object: <Task pending coro=<spin() running at spinner_yield.py:6>>  
Answer: 42
```

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

flags_await.py

flags_await.py

- Simplified implementation of demo script



```
import asyncio
```

```
import aiohttp # ④
```

```
from flags import BASE_URL, save_flag, show, main # ⑤
```

```
async def get_flag(cc): # ③
```

```
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
```

```
    resp = await aiohttp.request('GET', url) # ⑥
```

```
    image = await resp.read() # ⑦
```

```
    return image
```

```
async def download_one(cc): # ⑧
```

```
    image = await get_flag(cc) # ⑨
```

```
    show(cc)
```

```
    save_flag(image, cc.lower() + '.gif')
```

```
    return cc
```

```
def download_many(cc_list):
```

```
    loop = asyncio.get_event_loop() # ⑩
```

```
    to_do = [download_one(cc) for cc in sorted(cc_list)] # ⑪
```

```
    wait_coro = asyncio.wait(to_do) # ⑫
```

```
    res, _ = loop.run_until_complete(wait_coro) # ⑬
```

```
    loop.close() # ⑭
```

```
    return len(res)
```

```
if __name__ == '__main__':
```

```
    main(download_many)
```

Zoom in...

①...⑫

download_many
schedules many
instances of
download_one

⑦

download_one
delegates to **get_flag**

④, ⑤

get_flag delegates to
aiottp.request() and
response.read()

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

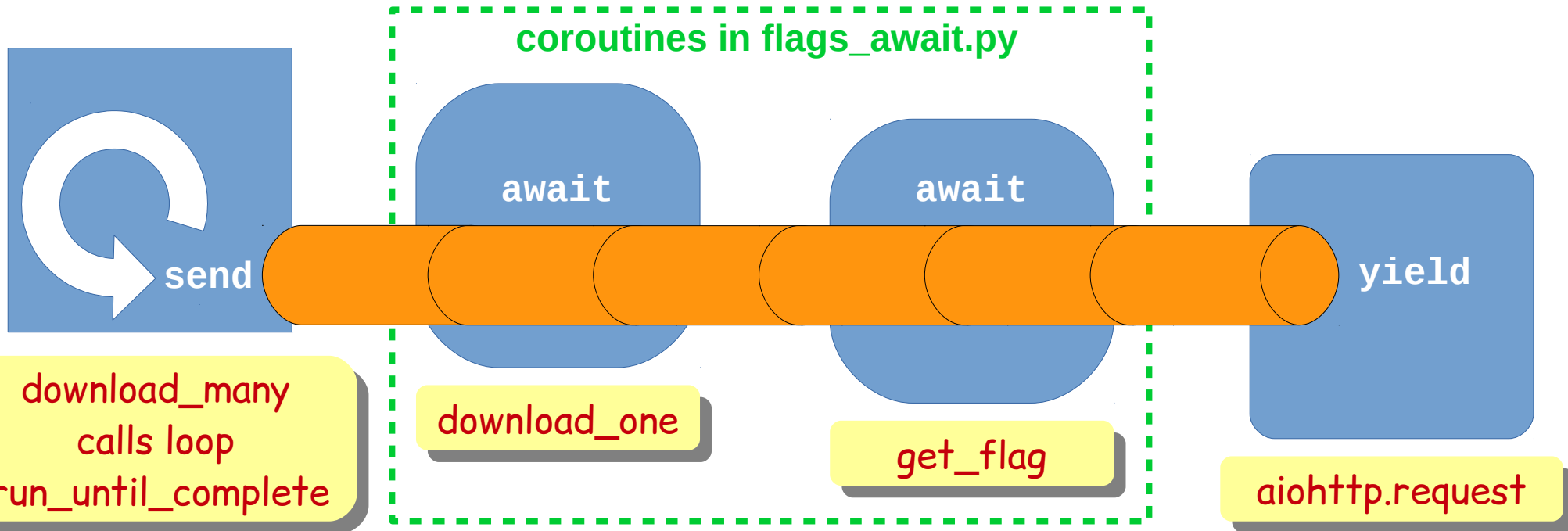
```
async def download_one(cc): # ⑥
    image = await get_flag(cc) # ③
    show(cc)
    save_flag(image, cc.lower() + '.gif')
    return cc
```

```
def download_many(cc_list):
    loop = asyncio.get_event_loop() # ⑧
    to_do = [download_one(cc) for cc in sorted(cc_list)] # ①
    wait_coro = asyncio.wait(to_do) # ⑨
    res, _ = loop.run_until_complete(wait_coro) # ⑫
    loop.close() # ⑩

    return len(res)
```

await in action

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



Zoom further... and squint

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

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

Zoom further... and squint

```
async def get_flag(cc): # ©
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = aiohttp.request('GET', url) # Ⓓ
    image = resp.read() # Ⓔ
    return image
```

- Guido van Rossum's tip for reading async code:
 - squint and ignore **await** 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

```
async def three_stages(request1):  
    response1 = await api_call1(request1)  
    # stage 1  
    request2 = step1(response1)  
    response2 = await api_call2(request2)  
    # stage 2  
    request3 = step2(response2)  
    response3 = await api_call3(request3)  
    # stage 3  
    step3(response3)
```

```
# ...
```

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

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

Escape (squinting)

```
async def three_stages(request1):  
    response1 = api_call1(request1)  
    # stage 1  
    request2 = step1(response1)  
    response2 = api_call2(request2)  
    # stage 2  
    request3 = step2(response2)  
    response3 = api_call3(request3)  
    # stage 3  
    step3(response3)
```

```
# ...
```

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

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

Before considering
another language for
asynchronous jobs,
try Python 3.3 or later!



supports yield from!

Python 3.5 `async/await`

- New keywords introduced for the first time since Python 3.0 (2008)
- PEP-492 *very* briefly:
 - **`async def`** is used to build *native coroutines*
 - **`await`** is used to delegate to *awaitable objects*
 - *native coroutines*; *decorated generator-based coroutines*; *implementers* of **`__await__`** protocol
 - new constructs available only inside *native coroutines*:
 - **`async for`**: supports asynchronous **`__aiter__`** and **`__anext__`**
 - **`async with`**: supports asynchronous **`__aenter__`** and **`__aexit__`**

Proper language
support for
coroutines, finally!

yield-from syntax

```
15 import aiohttp # ①
16
17 from flags import BASE_URL, save_flag, show, main # ②
18
19
20 @asyncio.coroutine # ③
21 def get_flag(cc):
22     url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
23     resp = yield from aiohttp.request('GET', url) # ④
24     image = yield from resp.read() # ⑤
25     return image
26
27
28 @asyncio.coroutine
29 def download_one(cc): # ⑥
30     image = yield from get_flag(cc) # ⑦
31     show(cc)
32     save_flag(image, cc.lower() + '.gif')
33     return cc
34
35
36 def download_many(cc_list):
37     loop = asyncio.get_event_loop() # ⑧
38     to_do = [download_one(cc) for cc in sorted(cc_list)] # ⑨
39     wait_coro = asyncio.wait(to_do) # ⑩
40     res, _ = loop.run_until_complete(wait_coro) # ⑪
41     loop.close() # ⑫
42
43     return len(res)
44
```

async/await syntax

```
15 import aiohttp # ①
16
17 from flags import BASE_URL, save_flag, show, main # ②
18
19
20 async def get_flag(cc): # ③
21     url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
22     resp = await aiohttp.request('GET', url) # ④
23     image = await resp.read() # ⑤
24     return image
25
26
27 async def download_one(cc): # ⑥
28     image = await get_flag(cc) # ⑦
29     show(cc)
30     save_flag(image, cc.lower() + '.gif')
31     return cc
32
33
34 def download_many(cc_list):
35     loop = asyncio.get_event_loop() # ⑧
36     to_do = [download_one(cc) for cc in sorted(cc_list)] # ⑨
37     wait_coro = asyncio.wait(to_do) # ⑩
38     res, _ = loop.run_until_complete(wait_coro) # ⑪
39     loop.close() # ⑫
40
41     return len(res)
42
```


Summary



- 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 **await** (or **yield from**) 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 + Q & A



- **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://bit.ly/oscon-spin>
- Me:
 - Twitter: @ramalhoorg

