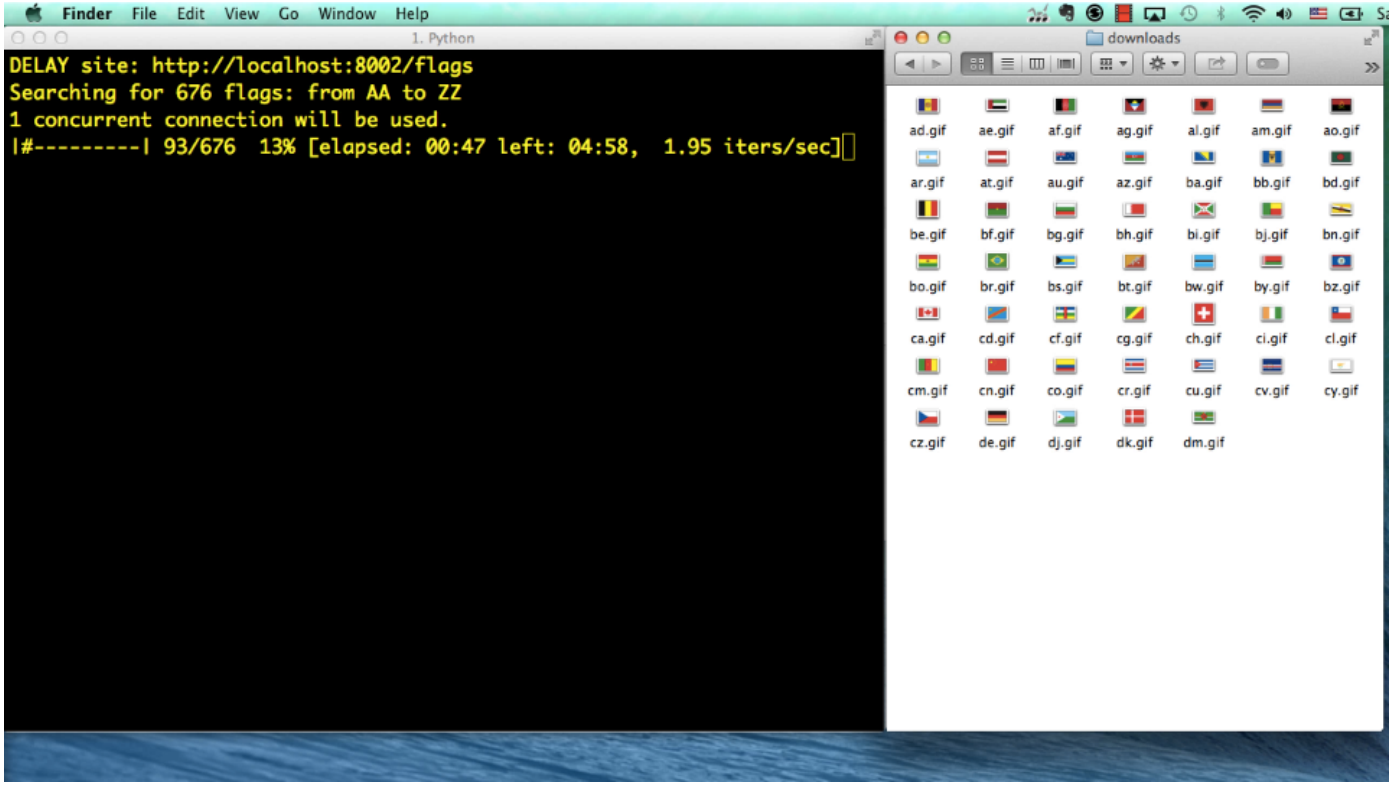
CONCURRENCY WITH PYTHON 3.5 ASYNC & AWAIT

O'REILLY®
OSCON

Demo: downloading images

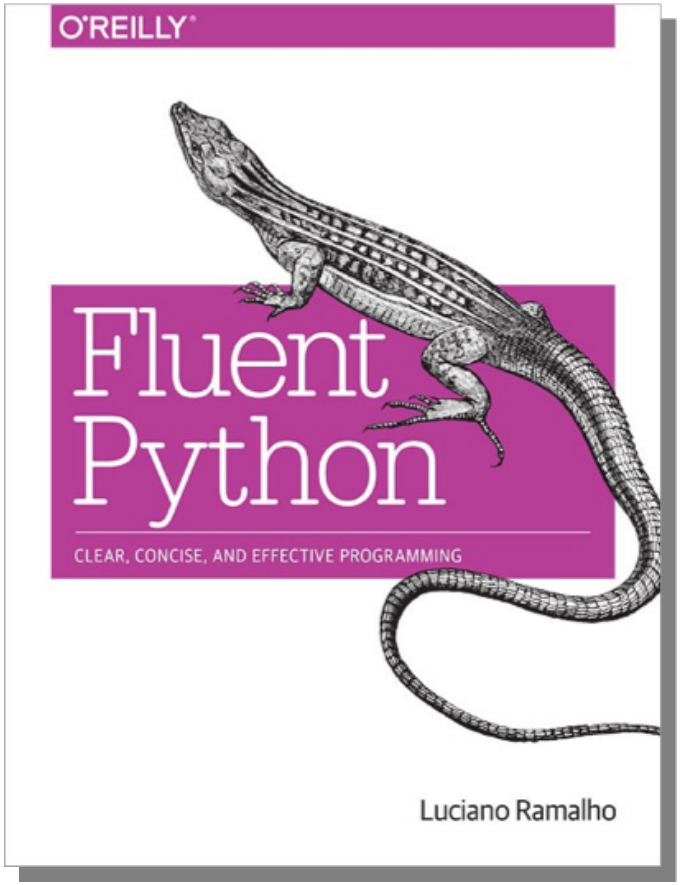
- Hitting 676 URLs, getting 194 flag pictures
- Sequential: 1.92 items/s
- Asynchronous: 150 items/s



http://www.youtube.com/watch?v=M8Z65tAl5l4

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



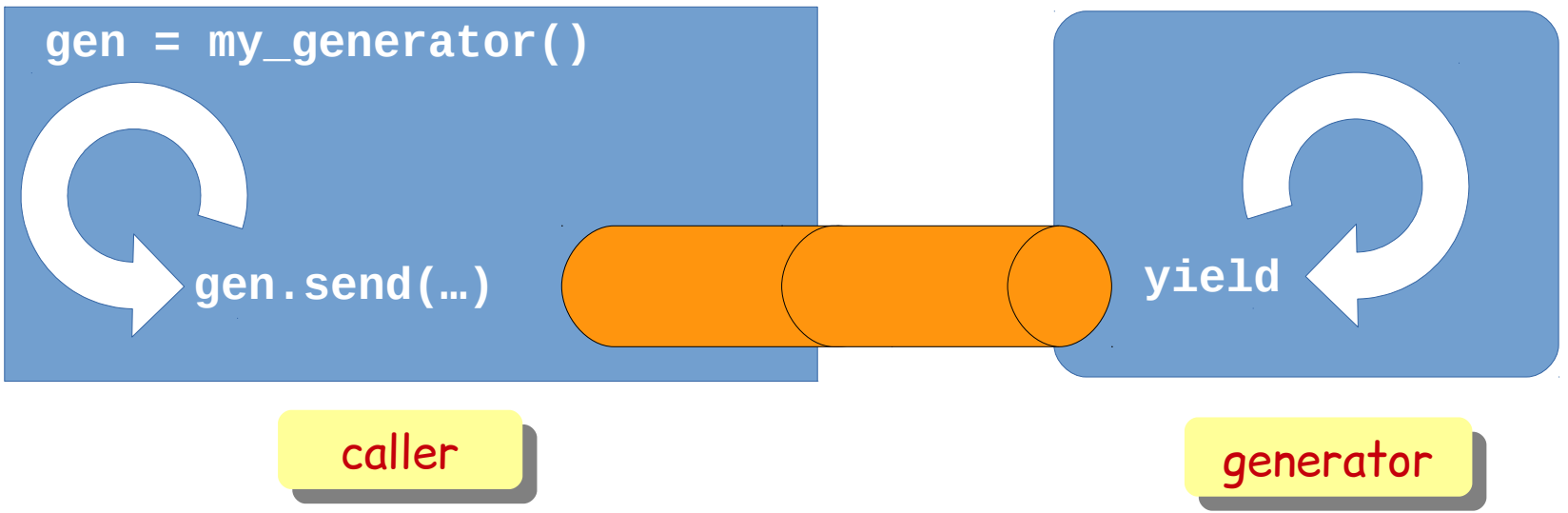
Pre-requisites

quick review next...

- You should know how a Python generator function works
  - Chapters 14 and 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 (i.e. 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:  # Ⓐ
    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()
```

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


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()
```

sleep() and I/O functions release the GIL

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:  # Ⓐ
    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_await.py

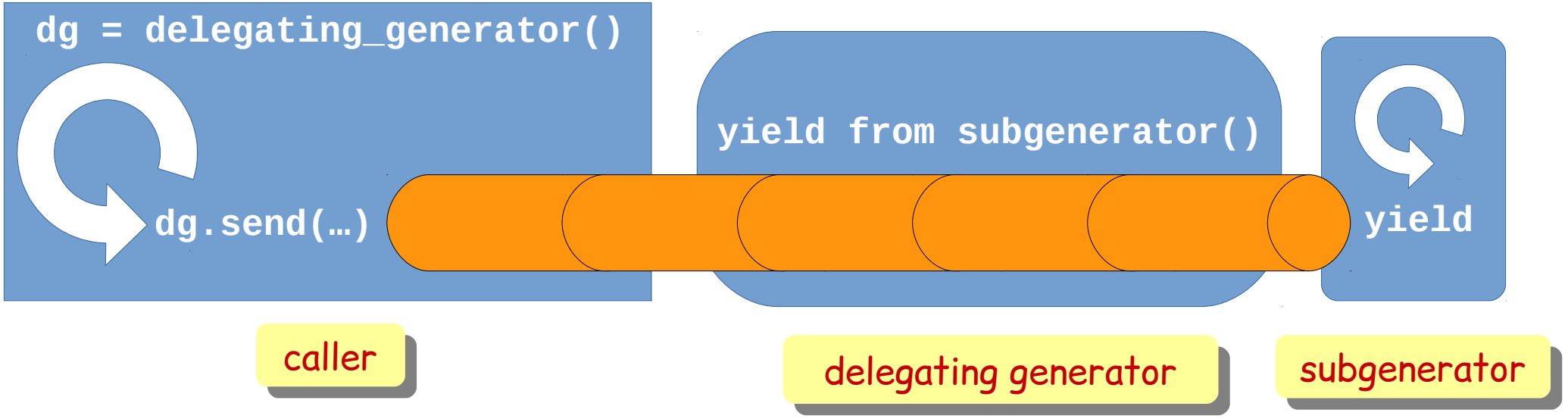
Coroutine spinner script: async/await

- 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**

```python
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


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()
```

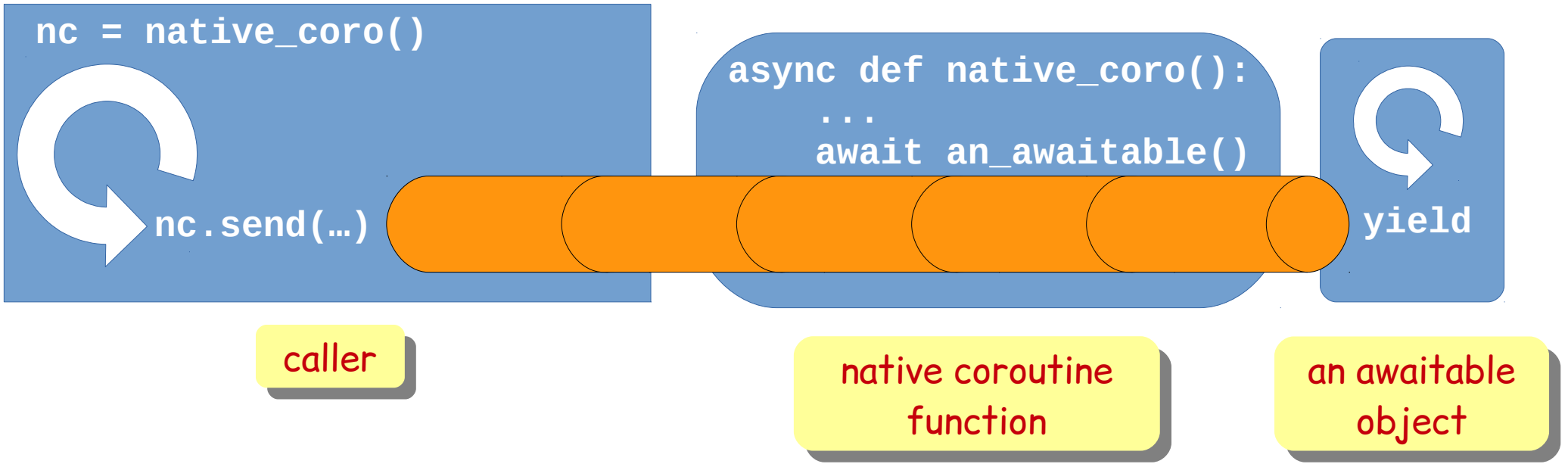
yield-from concepts

- PEP-380: Syntax for Delegating to a Subgenerator



async/await concepts

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



Coro... spinner: bottom

Ⓜ Drive **supervisor** coroutine with event loop

```
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()
```

main() blocks, waiting for result here

Coro... spinner: bottom

Ⓗ Schedule **Task** with **spin** coroutine

```python
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()
```

non-blocking: immediately returns Task object

Coro... spinner: bottom

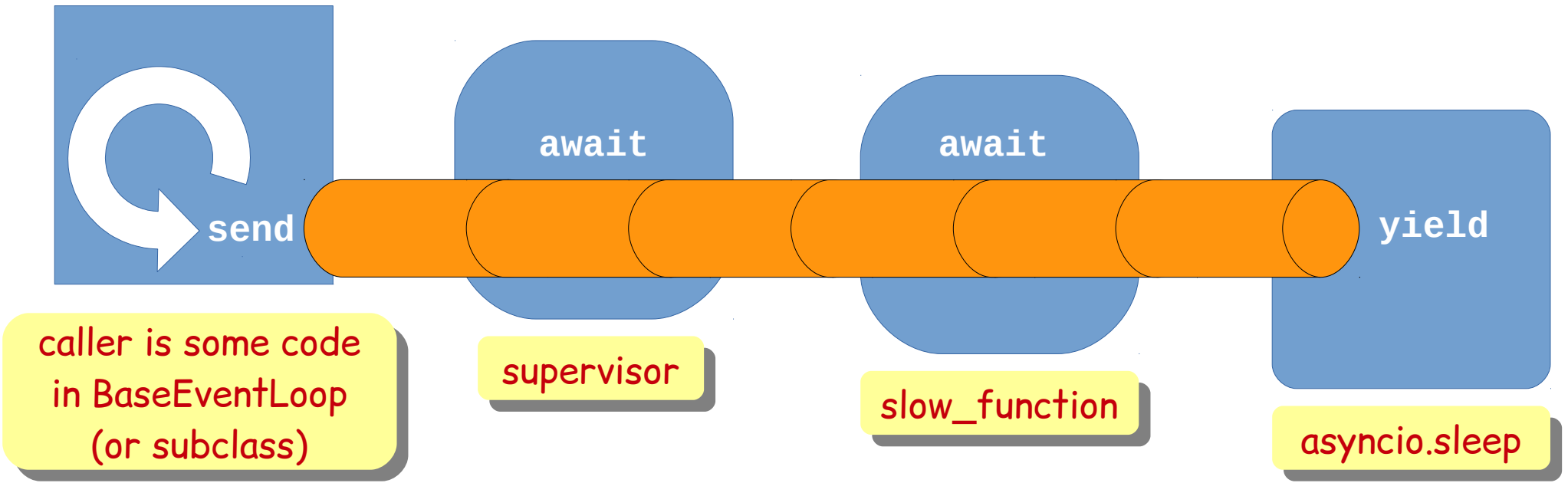
Ⓙ Wait for result from **slow_function**

```
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 blocks delegating coroutine supervisor()

await creates channel

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



Coro... spinner: bottom

Ⓙ 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()
```

slow_function() is driven directly by the event loop

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.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()
```

slow_function() blocks here

Coro... spinner: bottom

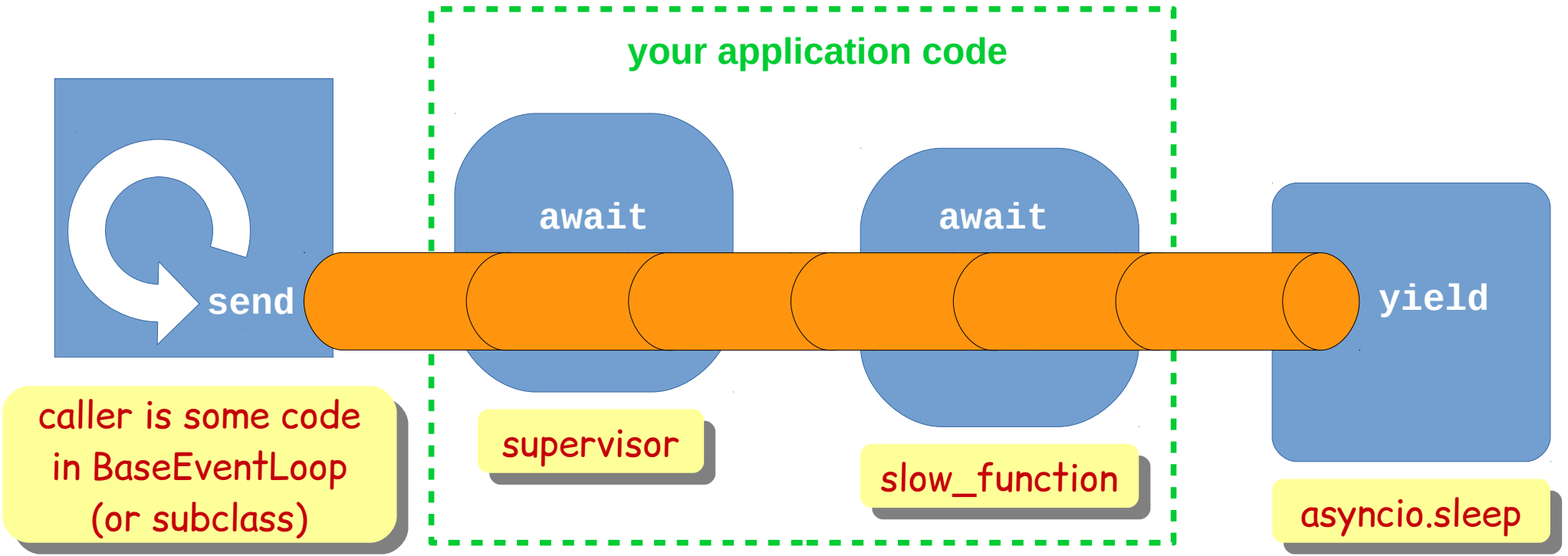
Ⓕ Delegate to **asyncio.sleep**

```
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


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


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

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

Driving async operations

- Application programmer writes functions that connect event loop to library functions that perform async I/O (or just sleep, in this case)



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_comple
    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)**

```
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
```

Coroutine spinner: top

Ⓓ Handle cancellation by terminating infinite loop, then clearing the console 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

threaded

```
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
```

asynchronous

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

Threaded x async comparison

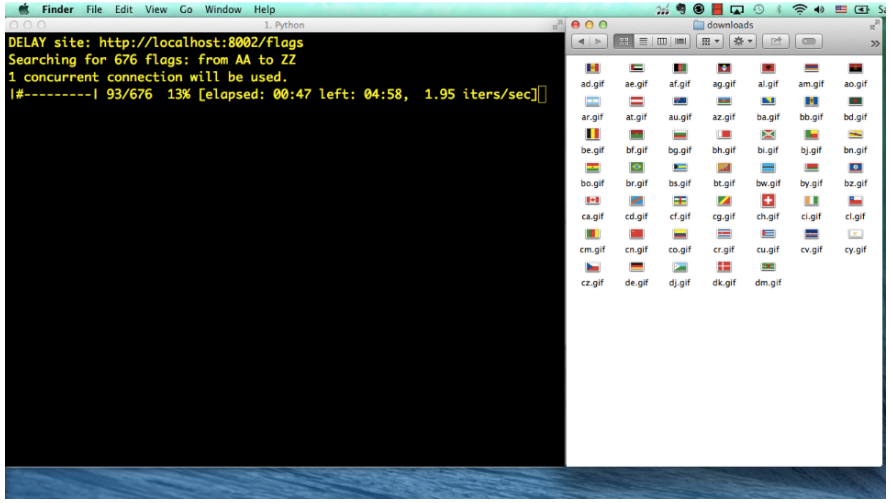
```
(.env35b3) $ python spinner_thread.py
spinner object: <Thread(Thread-1, initial)>
Answer: 42
(.env35b3) $ python spinner_yield.py
spinner object: <Task pending coro=<spin() running at spinner_yield.py:6>>
Answer: 42
```

- **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




```python
import asyncio

import aiohttp  # Ⓐ

from flags import BASE_URL, save_flag, show, main  # Ⓑ


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


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


def download_many(cc_list):
    loop = asyncio.get_event_loop()  # Ⓗ
    to_do = [download_one(cc) for cc in sorted(cc_list)]  # Ⓘ
    wait_coro = asyncio.wait(to_do)  # Ⓙ
    res, _ = loop.run_until_complete(wait_coro)  # Ⓚ
    loop.close() # Ⓛ

    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**

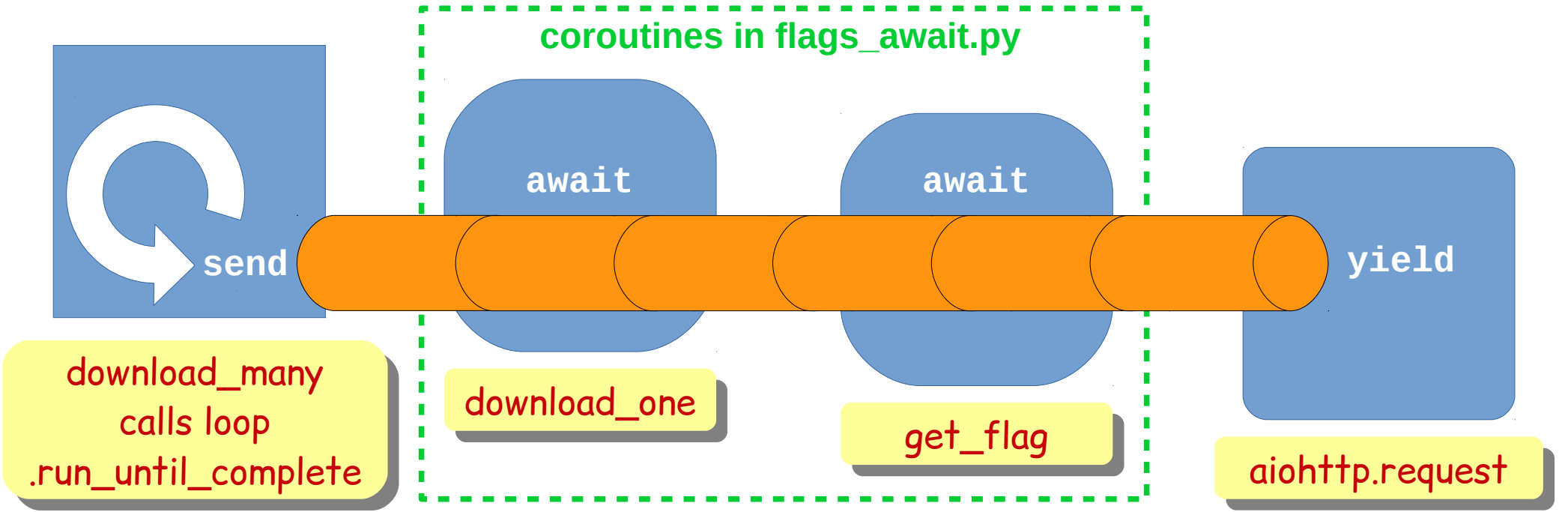
Ⓓ, Ⓔ

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

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


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


def download_many(cc_list):
    loop = asyncio.get_event_loop()  # Ⓗ
    to_do = [download_one(cc) for cc in sorted(cc_list)]  # Ⓘ
    wait_coro = asyncio.wait(to_do)  # Ⓙ
    res, _ = loop.run_until_complete(wait_coro)  # Ⓚ
    loop.close() # Ⓛ

    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...

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

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 the **await** (or **yield from**) keywords

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
```

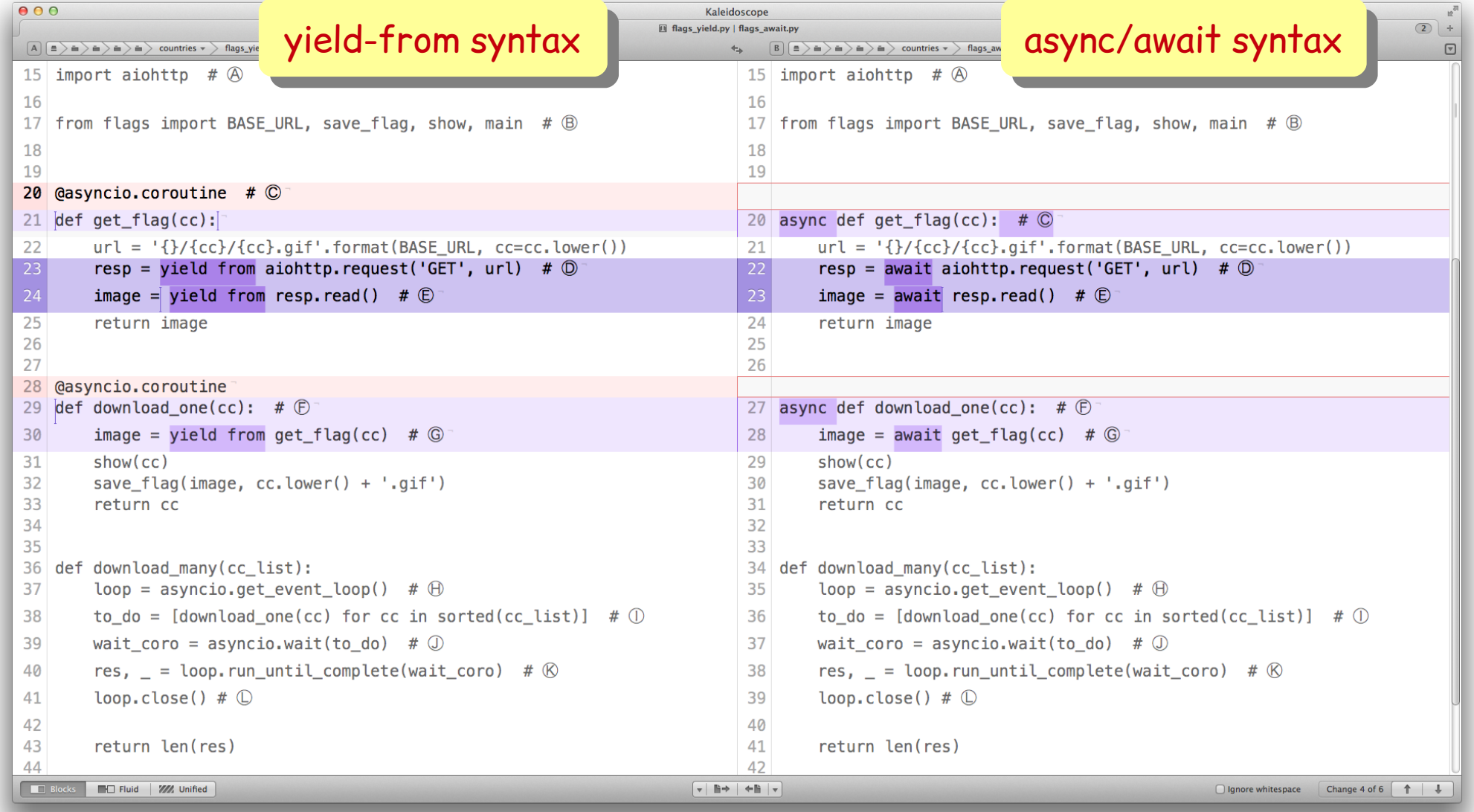
context is preserved through all stages: it's all in the local scope of the native coroutine

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

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:** suports asynchronous __**aiter**__ and __**anext**__
    - **async with:** suports asynchronous __**aenter**__ and __**aexit**__

Proper language support for coroutines, finally!


yield-from syntax
async/await syntax
Kaleidoscope
flags_yield.py | flags_await.py
countries
flags_yie
countries
flags_aw
import aiohttp # Ⓐ
from flags import BASE_URL, save_flag, show, main # Ⓑ
@asyncio.coroutine # Ⓒ
def get_flag(cc):
url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
resp = yield from aiohttp.request('GET', url) # Ⓓ
image = yield from resp.read() # Ⓔ
return image
@asyncio.coroutine
def download_one(cc): # Ⓕ
image = yield from get_flag(cc) # Ⓖ
show(cc)
save_flag(image, cc.lower() + '.gif')
return cc
def download_many(cc_list):
loop = asyncio.get_event_loop() # Ⓗ
to_do = [download_one(cc) for cc in sorted(cc_list)] # Ⓘ
wait_coro = asyncio.wait(to_do) # Ⓙ
res, _ = loop.run_until_complete(wait_coro) # Ⓚ
loop.close() # Ⓛ
return len(res)
import aiohttp # Ⓐ
from flags import BASE_URL, save_flag, show, main # Ⓑ
async def get_flag(cc): # Ⓒ
url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
resp = await aiohttp.request('GET', url) # Ⓓ
image = await resp.read() # Ⓔ
return image
async def download_one(cc): # Ⓕ
image = await get_flag(cc) # Ⓖ
show(cc)
save_flag(image, cc.lower() + '.gif')
return cc
def download_many(cc_list):
loop = asyncio.get_event_loop() # Ⓗ
to_do = [download_one(cc) for cc in sorted(cc_list)] # Ⓘ
wait_coro = asyncio.wait(to_do) # Ⓙ
res, _ = loop.run_until_complete(wait_coro) # Ⓚ
loop.close() # Ⓛ
return len(res)
Blocks
Fluid
Unified
Ignore whitespace
Change 4 of 6

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 overhead (compared to OS threads)

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!
- Twitter feed for me and the book:
  - @ramalhoorg, @fluentpython