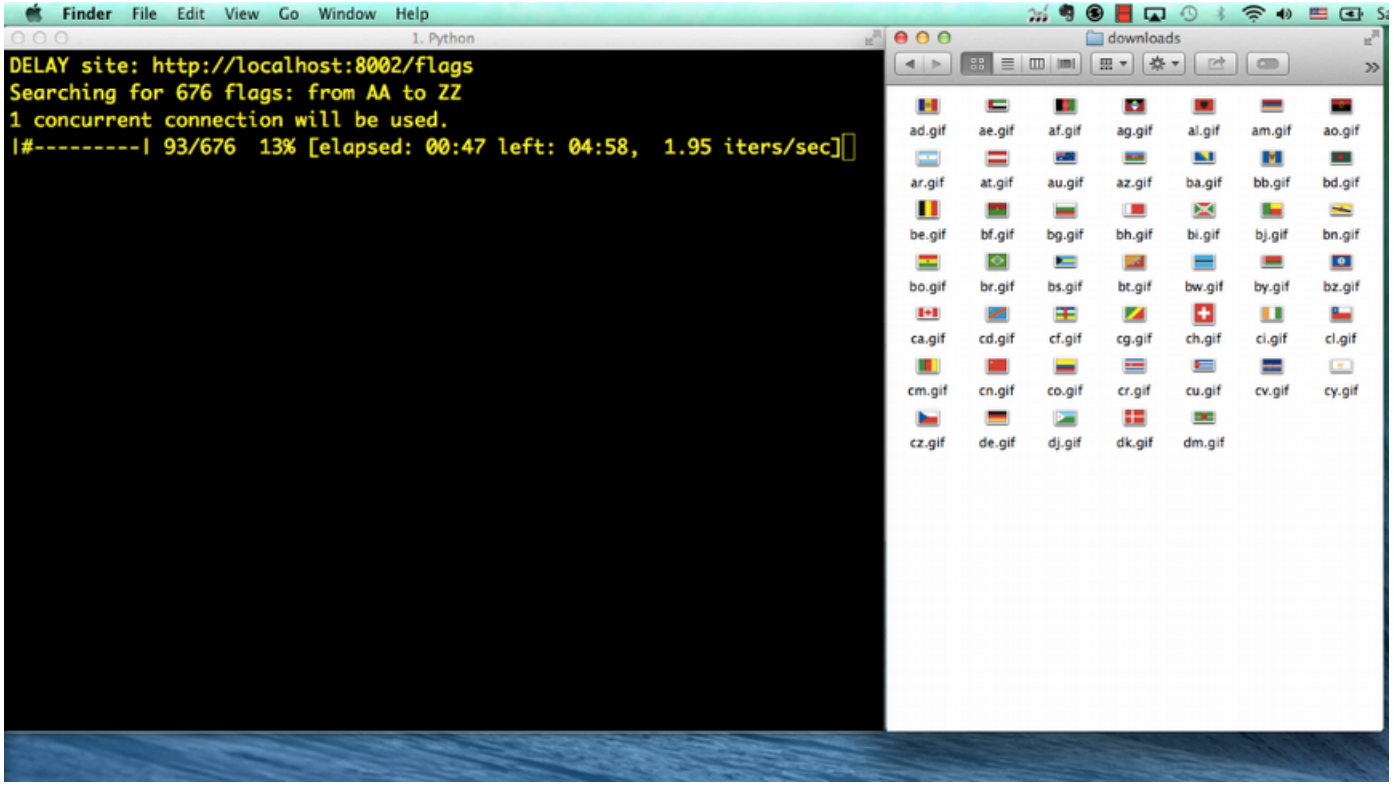
PLATE SPINNING: MODERN CONCURRENCY IN PYTHON

Demo: downloading images

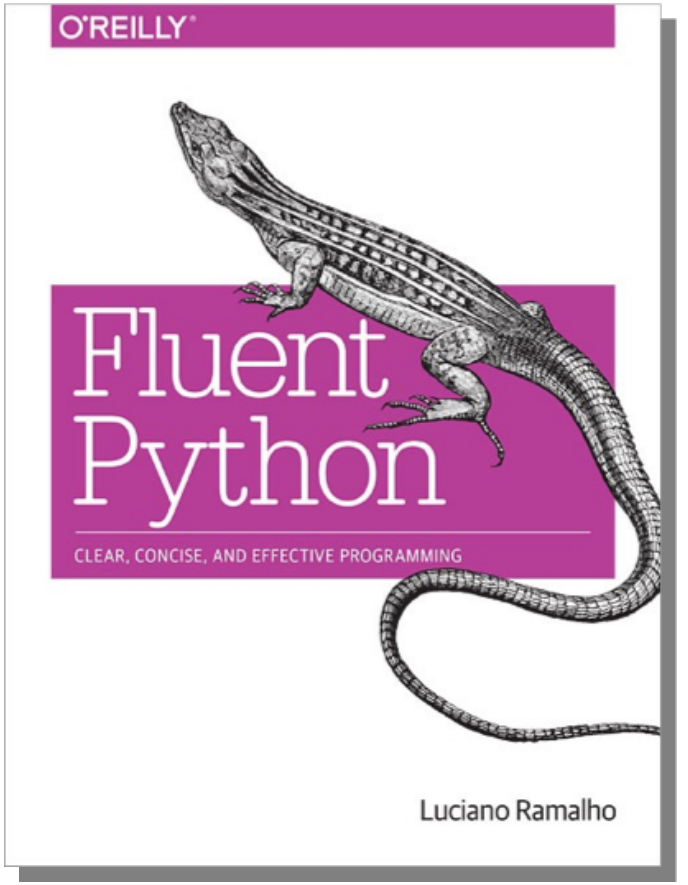
- Hitting 676 URLs, getting 194 flag pictures
- Sequential: 1.92 items/s
- Asynchronous: 150 items/s



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

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



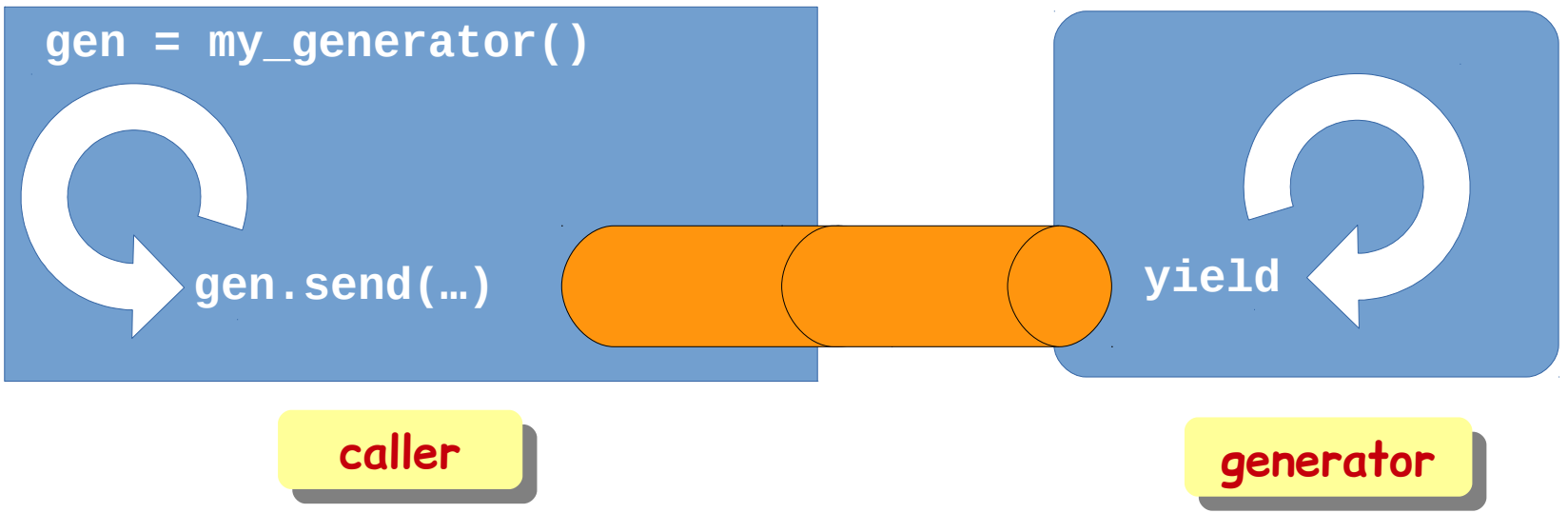
Pre-requisites

quick review next...

- You should know how a Python generator function works
- Chapters 14 and 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
```

spinner_thread.py

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()
```

Thread... spinner bottom

Ⓚ Supervisor starts spinner thread

Ⓛ Calls slow_function, which blocks at Ⓗ

Ⓜ Uses signal object to tell spinner thread to stop

```
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():  # Ⓘ
31      signal = Signal()
32      spinner = threading.Thread(target=spin,
33                                 args=('thinking!', signal))
34      print('spinner object:', spinner)  # Ⓙ
35      spinner.start()  # Ⓚ
36      result = slow_function()  # Ⓛ
37      signal.go = False  # Ⓜ
38      spinner.join()  # Ⓝ
39      return result
40
41
42  def main():
43      result = supervisor()  # Ⓞ
44      print('Answer:', result)
45
46
47  if __name__ == '__main__':
48      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:  # Ⓐ
    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_asyncio.py

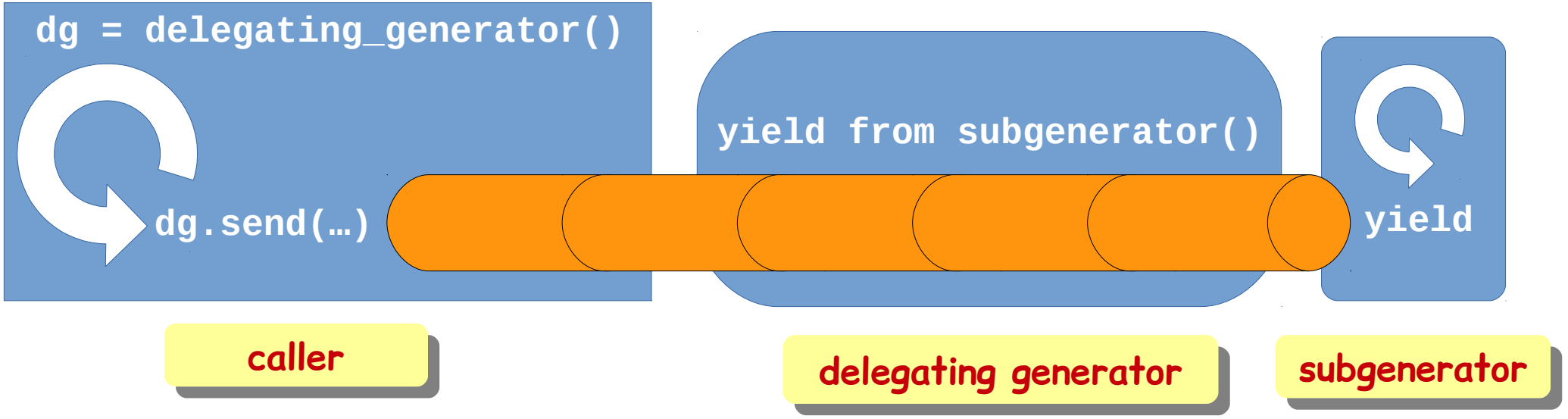
Coroutine spinner script: yield/from

- 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()  # Ⓚ
    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  @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():  # Ⓖ
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()
```

Coro... spinner: bottom

Ⓜ Drive supervisor coroutine with event loop

Ⓗ Schedule Task with spin coroutine

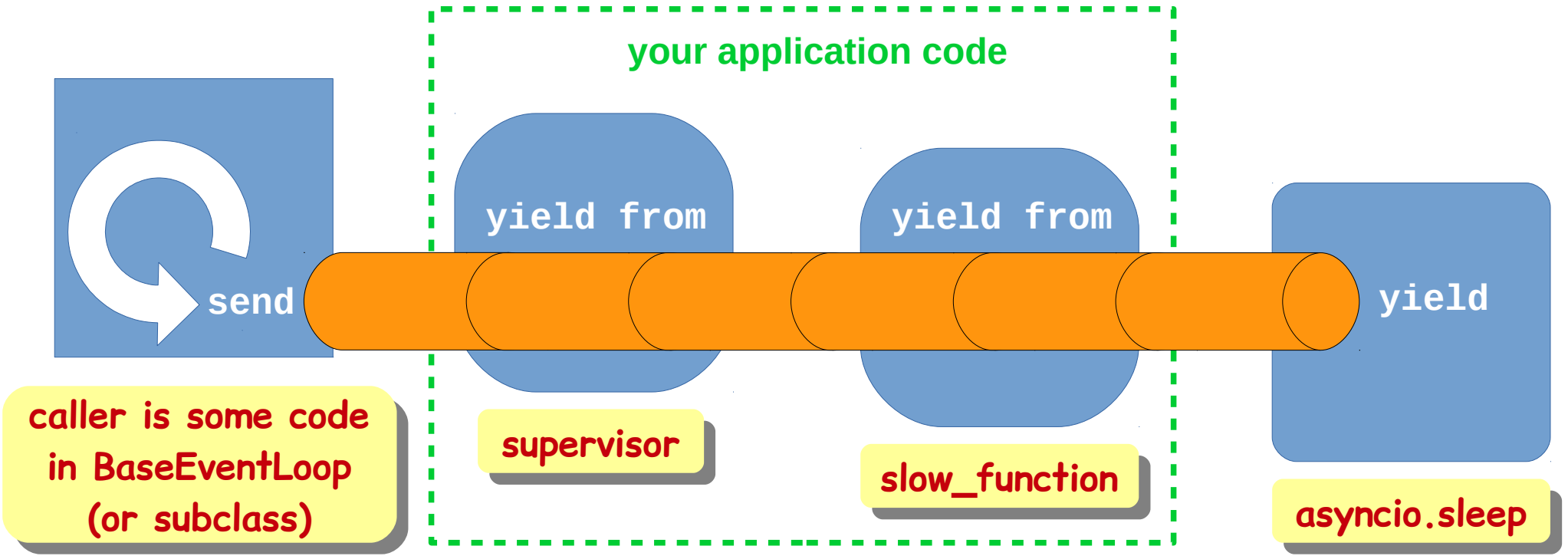
Ⓙ Wait for result from slow_function

```
20
21  @asyncio.coroutine
22  def slow_function():  # Ⓔ
23      # pretend waiting a long time for I/O
24      yield from asynci
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  @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():  # Ⓖ
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()
```

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  @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():  # Ⓖ
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()
```

slow_function is the subgenerator in this context

Coro... spinner: bottom

Ⓕ Delegate to asyncio.sleep

```
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():  # Ⓖ
30      spinner = asyncio.ensure_futu
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()
```

slow_function is the delegating generator here

Ⓗ

Coro... spinner: bottom

Ⓕ Delegate to asyncio.sleep

```
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():  # Ⓖ
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()
```

asyncio.sleep is the subgenerator here

Coro... spinner: bottom

Ⓕ Delegate to asyncio.sleep

```
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():  # Ⓖ
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()
```

yield from blocks slow_function

Coro spinner bottom

Ⓕ Delegate to asyncio.sleep

```
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():  #
30      spinner = asynci
31      print('spinner o
32      result = yield fr
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()
```

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  @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():  # Ⓖ
30      spinner = asyncio.e                                    Ⓗ
31      print('spinner objec
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()
```

when subgenerator returns, delegating generator resumes at yield from

Coro... spinner bottom

Ⓚ After slow_function returns, cancel spinner Task

```
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():  # Ⓖ
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_loo
39      result = loop.run_until_comp
40      loop.close()
41      print('Answer:', result)
42
43
44  if __name__ == '__main__':
45      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

threaded

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

async

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

- async 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
```

async

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

Threaded x async comparison

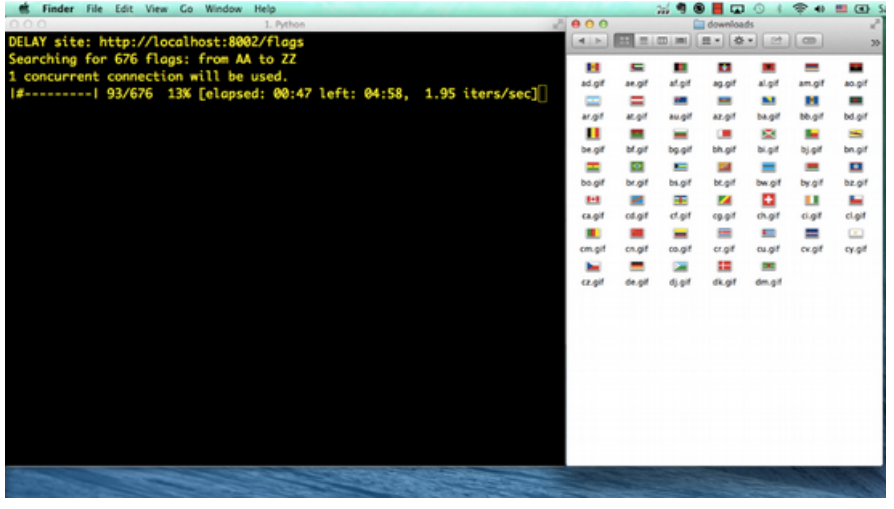
```
(.env35b3) $ python spinner_thread.py
spinner object: <Thread(Thread-1, initial)>
Answer: 42
(.env35b3) $ python spinner_yield.py
spinner object: <Task pending coro=<spin() running at spinner_yield.py:6>>
Answer: 42
```

- spinner 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




```python
import asyncio

import aiohttp  ❶

from flags import BASE_URL, save_flag, show, main  ❷


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


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


def download_many(cc_list):
    loop = asyncio.get_event_loop()  ❽
    to_do = [download_one(cc) for cc in sorted(cc_list)]  ❾
    wait_coro = asyncio.wait(to_do)  ❿
    res, _ = loop.run_until_complete(wait_coro)  ⓫
    loop.close()  ⓬

    return len(res)


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

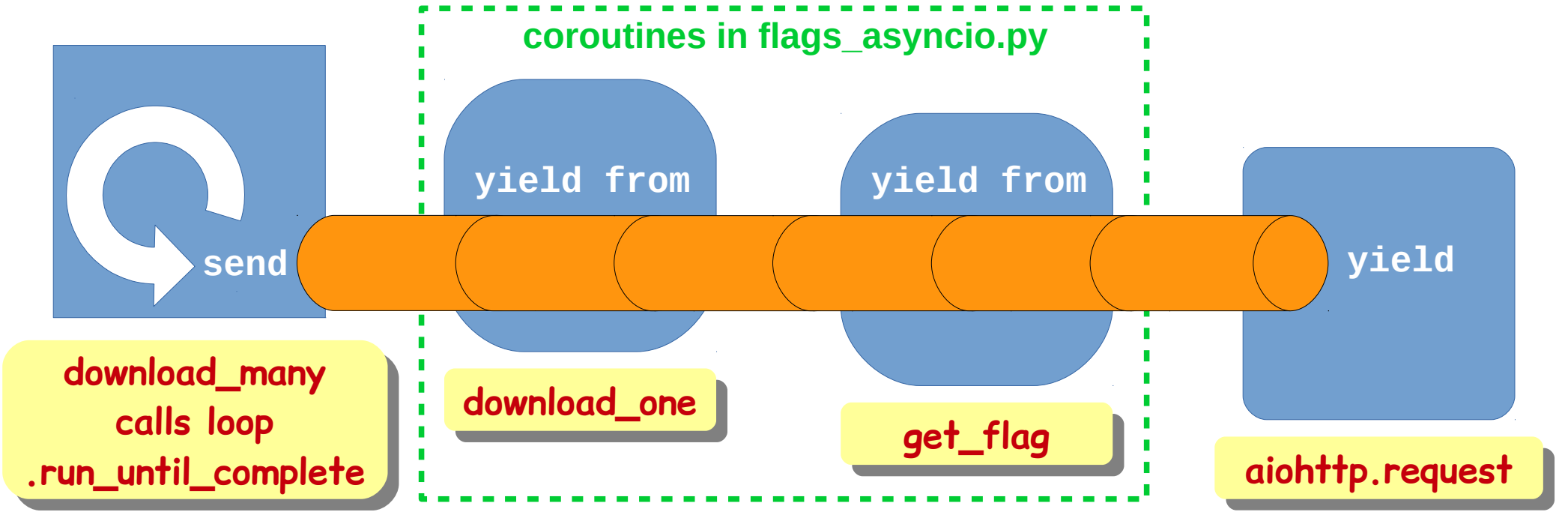
Zoom in...

- **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    ❸
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)    ❹
    image = yield from resp.read()    ❺
    return image


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


def download_many(cc_list):
    loop = asyncio.get_event_loop()    ❽
    to_do = [download_one(cc) for cc in sorted(cc_list)]    ❾
    wait_coro = asyncio.wait(to_do)    ❿
    res, _ = loop.run_until_complete(wait_coro)    ⓫
    loop.close()    ⓬

    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... and squint

```
@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
```

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

Zoom further... and squint

```
@asyncio.coroutine
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 **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

Escape (squinting)

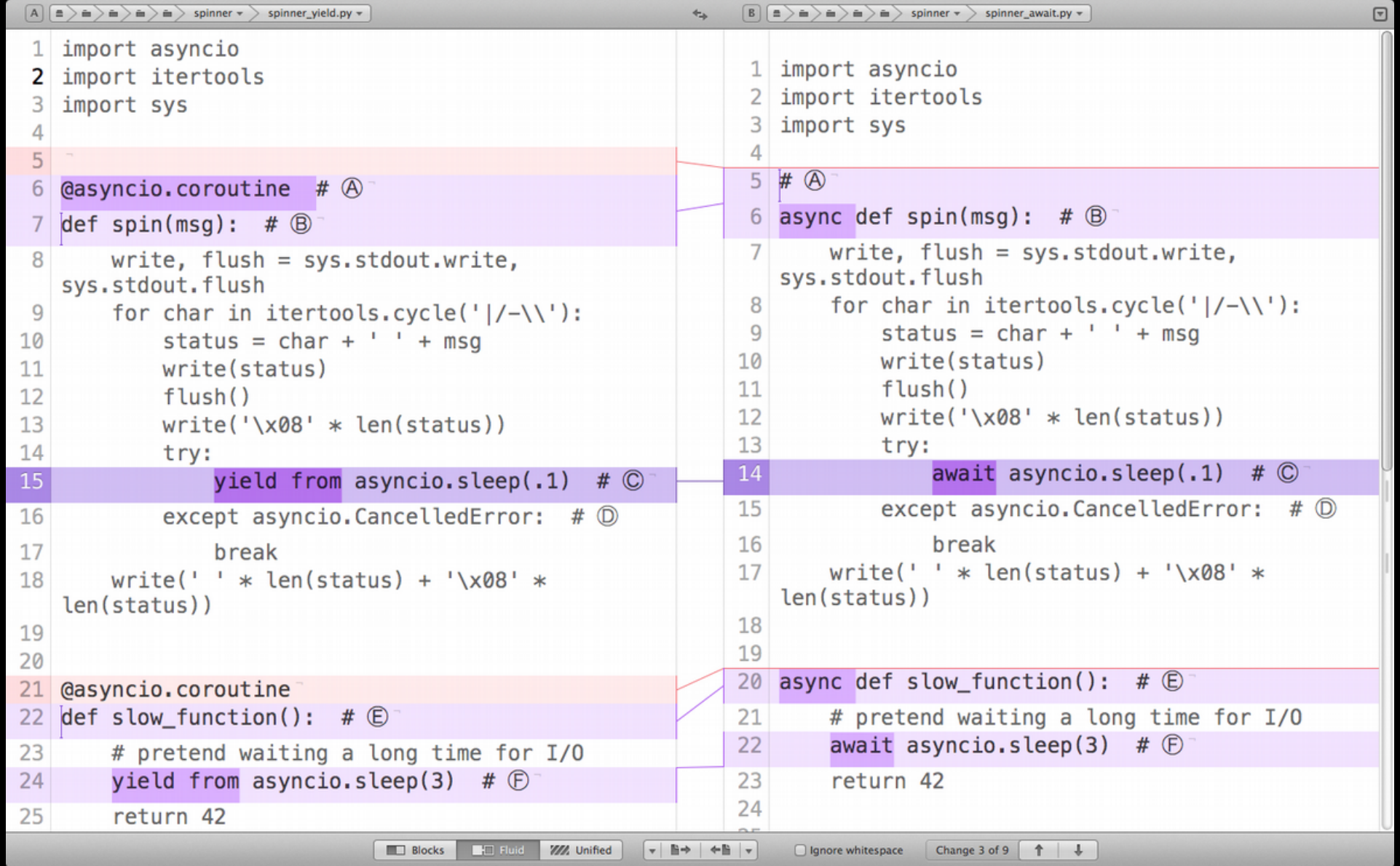
```
@asyncio.coroutine
def three_stages(request1):
    response1 =             api_call1(request1)
    # stage 1
    request2 = step1(response1)
    response2 =             api_call2(request2)
    # stage 2
    request3 = step2(response2)
    response3 =             api_call3(request3)
    # stage 3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

# 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!**


spinner
spinner_yield.py
spinner
spinner_await.py
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
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
Blocks
Fluid
Unified
Ignore whitespace
Change 3 of 9

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 **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 + 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