

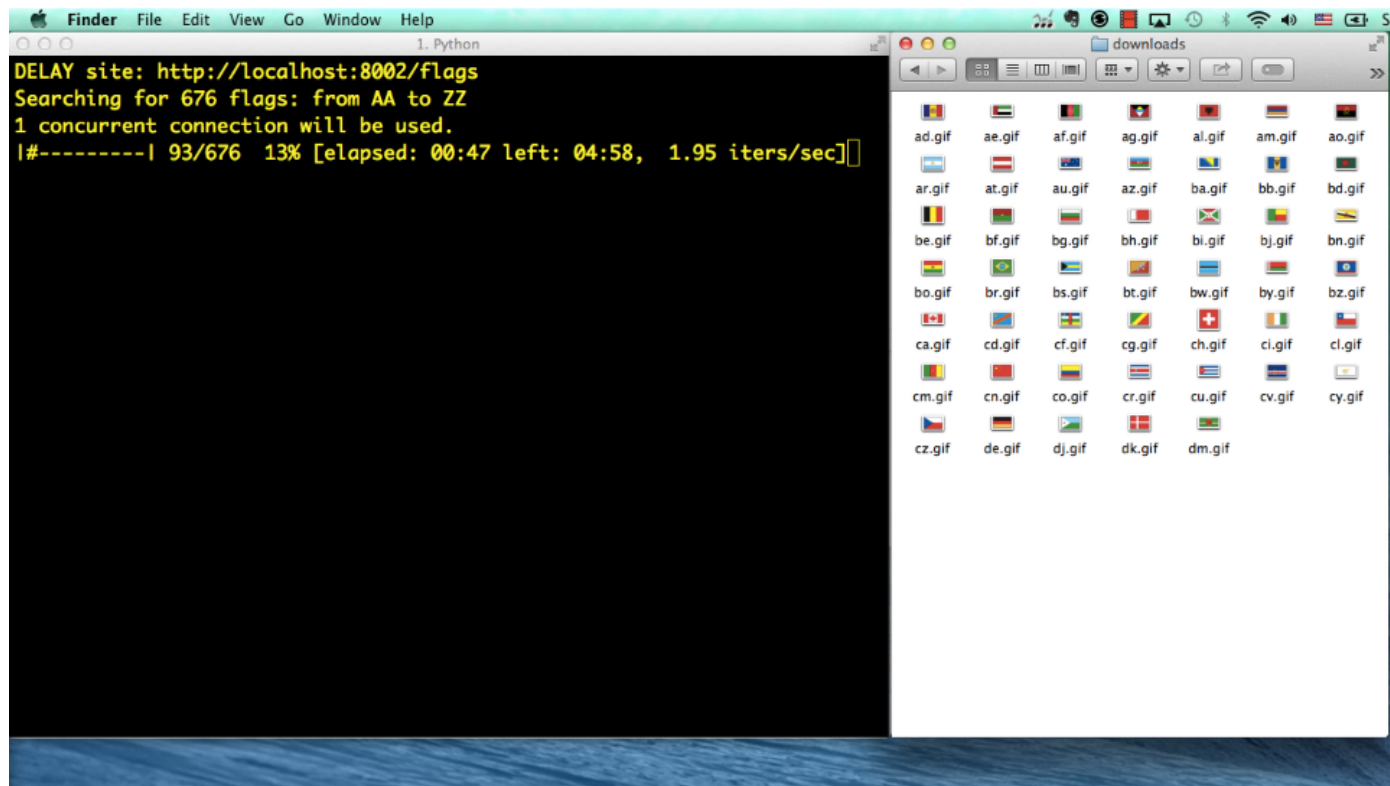
CONCURREN- CY WITH PYTHON 3.5 ASYNC & AWAIT

ThoughtWorks®



Demo: baixando imagens

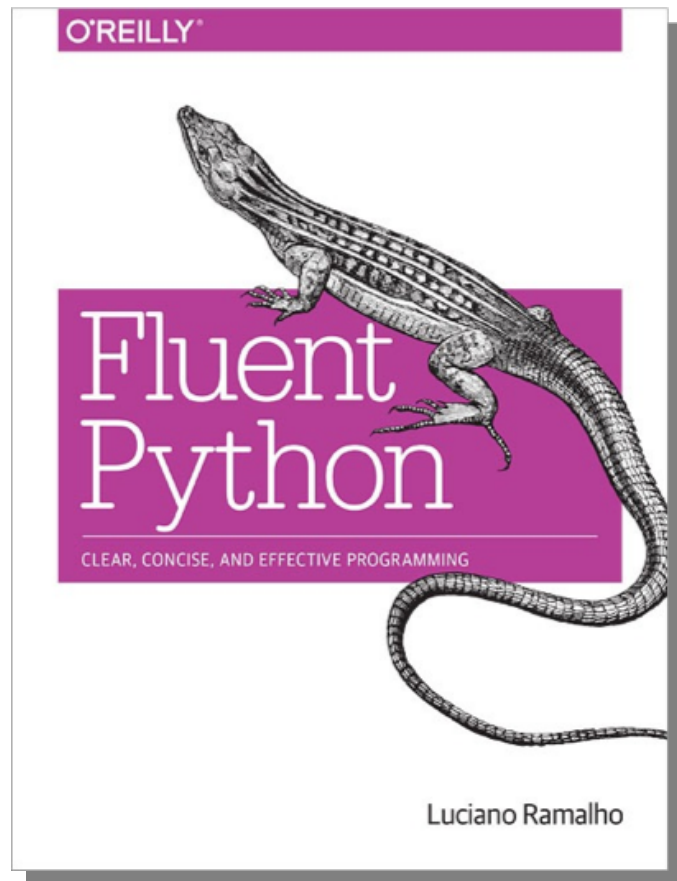
- Acessando 676 URLs, obtendo 194 images
- Sequencial:
< 2 itens/s
- Assíncrono:
150 itens/s



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

Como o demo foi feito

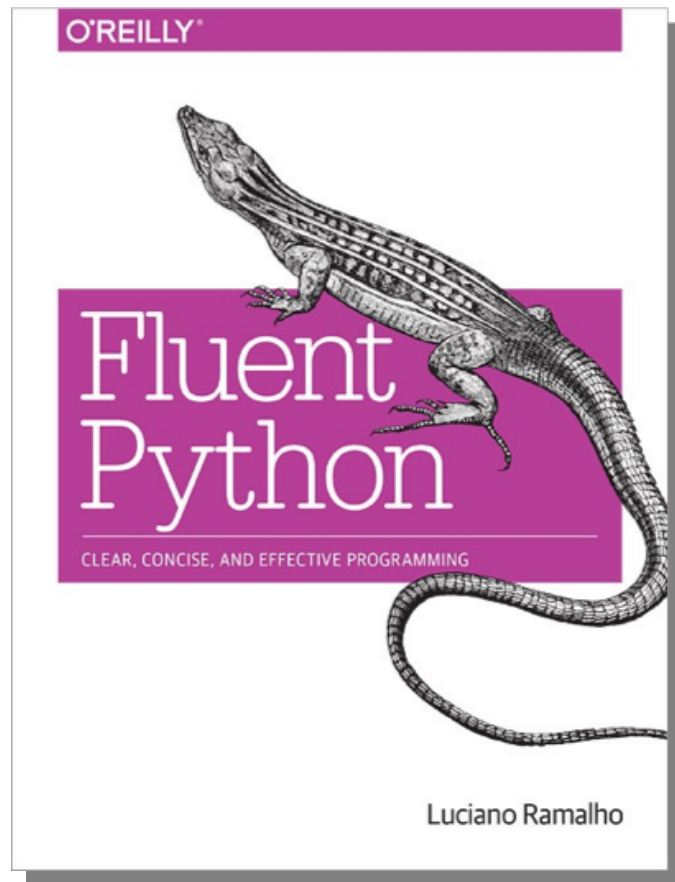
- Quatro versões do script:
 - sequencial
 - threaded com **concurrent.futures.ThreadPoolExecutor**
 - assíncrono com **asyncio**: usando **yield from**
 - assíncrono com **asyncio**: usando **await**
- Ambiente de testes:
 - local nginx server + vaurien proxy
- Instruções nos capítulos 17 e 18 do **Python Fluente**



Pré-requisitos

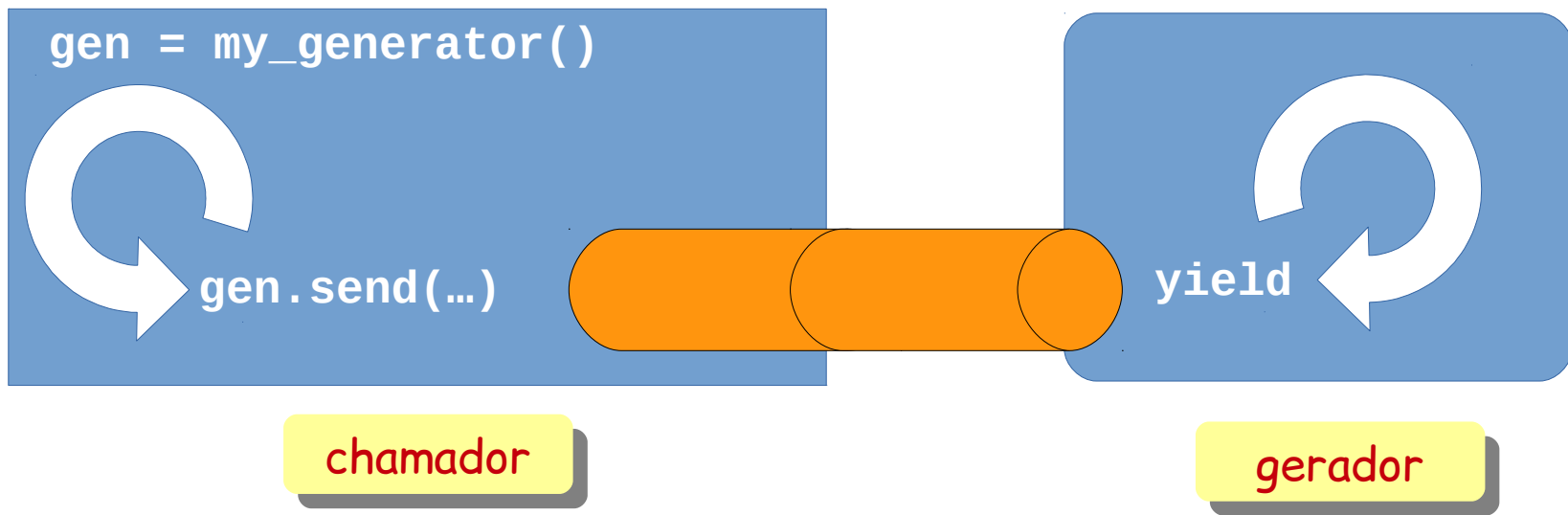
revisão rápida
em seguida...

- Como funciona uma função geradora
 - Capítulos 14 e 16 do Python Fluente abordam isso em detalhes
- Dica: entenda geradores *bem* antes de estudar corrotinas



Gerador: revisão rápida

- Gerador: qualquer função que tem a palavra **yield** em seu corpo
- Chamador: envia valores *e/ou* gerador produz valor
- Importante: seu progresso é sincronizada (i.e. laços sincronizados)



Demo: scripts com “spinner”

```
(.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

Spinner com threads: panorama

- Usa o pacote **threading**
- Thread principal inicia thread **spinner**
- Thread principal fica bloqueada esperando a **slow_function** enquanto a thread **spinner** executa
- Quando **slow_function** termina, thread principal sinaliza para thread **spinner** terminar

```
import threading
import itertools
import time
import sys

class Signal: # ④
    go = True

def spin(msg, signal): # ③
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'): # ⑤
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status)) # ⑥
        time.sleep(.1)
        if not signal.go: # ⑦
            break
    write(' ' * len(status) + '\x08' * len(status)) # ⑧

def slow_function(): # ⑨
    # pretend waiting a long time for I/O
    time.sleep(3) # ⑩
    return 42

def supervisor(): # ⑪
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner) # ⑫
    spinner.start() # ⑬
    result = slow_function() # ⑭
    signal.go = False # ⑮
    spinner.join() # ⑯
    return result

def main():
    result = supervisor() # ⑰
    print('Answer:', result)

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


Spinner com thread: trecho final

- Ⓚ supervisor inicia thread spinner
- Ⓛ Invoca `slow_function`, que bloqueia em Ⓜ
- Ⓜ Usa objeto `signal` para *sugerir* a thread `spinner` que pode parar

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

`sleep()` e
funções de I/O
liberam a 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()
```

Spinner com thread: trecho inicial

- Ⓑ **spin** recebe instância de **Signal** como segundo argumento
- Ⓒ **itertools.cycle()** produz série infinita de **| / - **
- Ⓓ escreve backspaces (`\x08`), e dorme por 0.1s
- Ⓔ encerra quando **signal.go** se torna **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
```

Spinner com threads: notas

- Escalonador do SO pode interromper uma thread a qualquer momento – por isso threads não podem ser canceladas por outras threads
- Invocar **sleep()** ou funções de E/S praticamente garante a priorização de outra thread
- *Todas* as funções da biblioteca padrão que fazem I/O liberam a GIL, permitindo a execução concorrente de outros bytecodes de Python

```
import threading
import itertools
import time
import sys

class Signal: # A
    go = True

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

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()
```

spinner_await.py

Coroutine spinner script: `async/await`

- Usa o pacote **asyncio**
- Thread principal (a *única* thread) inicia o loop de eventos para acionar as corrotinas
- **supervisor**, **spin** e **slow_function** são corrotinas
- corrotinas esperam resultados de outras corrotinas usando **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

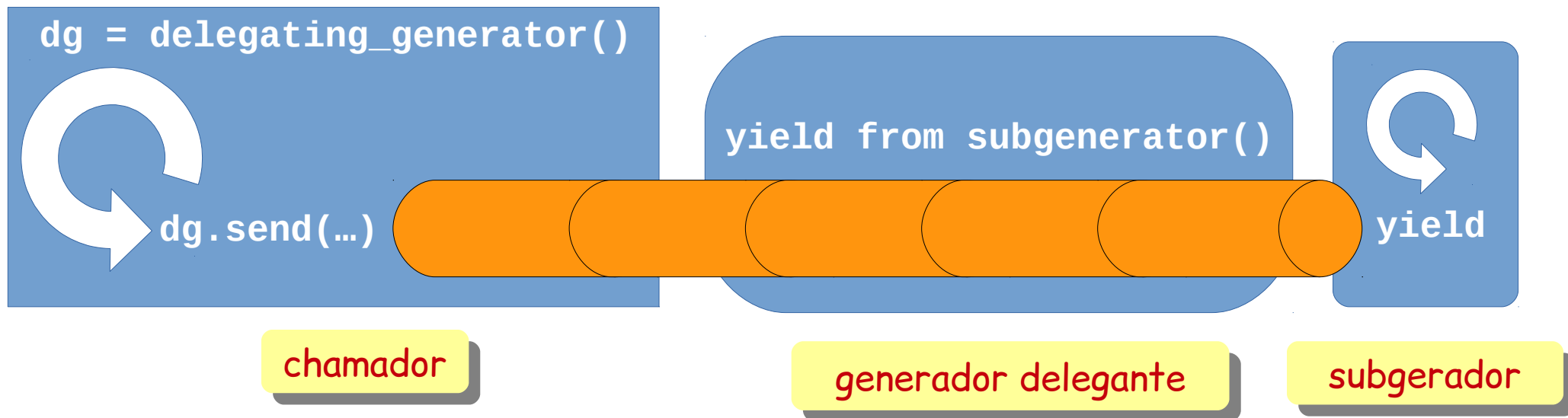
async def supervisor(): # ⑦
    spinner = asyncio.ensure_future(spin('thinking!')) # ⑧
    print('spinner object:', spinner) # ⑨
    result = await slow_function() # ⑩
    spinner.cancel() # ⑪
    return result

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

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

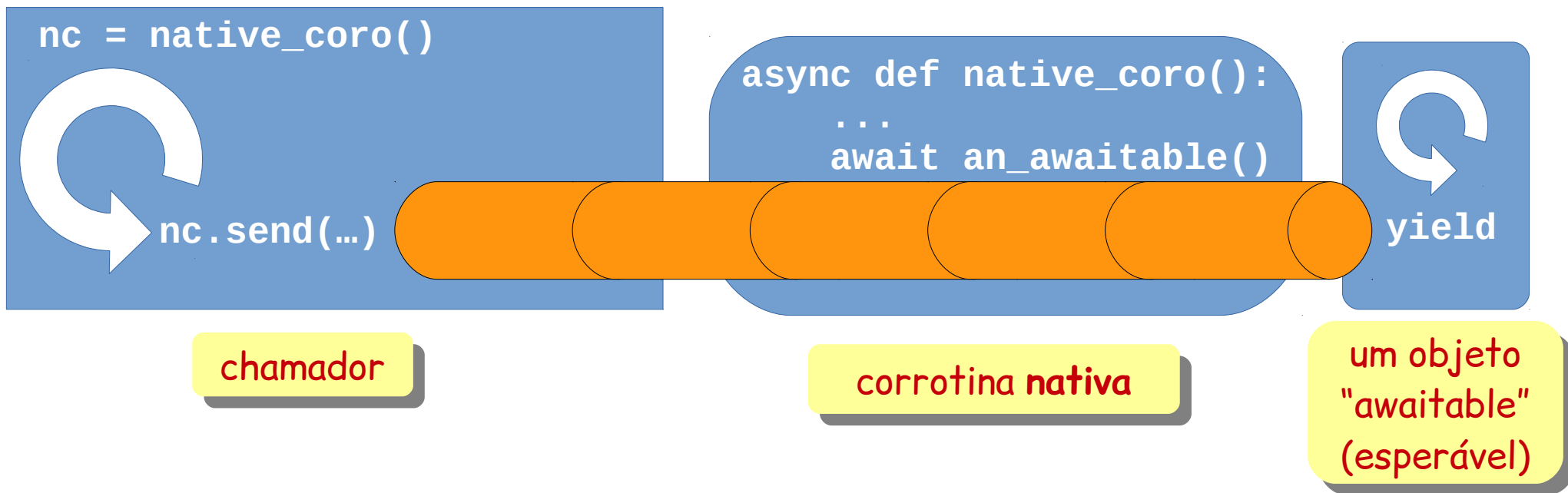
yield-from: conceitos

- PEP-380: Syntax for Delegating to a Subgenerator



async/await: conceitos

- PEP-492: Coroutines with async and await syntax
 - introduziu *corrotinas nativas* (\neq geradores-corrotinas)



Coro... spinner: final


Ⓜ Aciona corrotina
supervisor
usando o loop de
eventos

```
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() bloqueia e
aguarda resultado aqui

Coro... spinner: final

Ⓜ Agenda tarefa
(Task) **spinner**
com a corrotina
spin

```
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()
```

não bloqueante:
ensure_future
devolve Task
imediatamente

Coro... spinner: final


ⓐ Aguarda resultado
de `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)

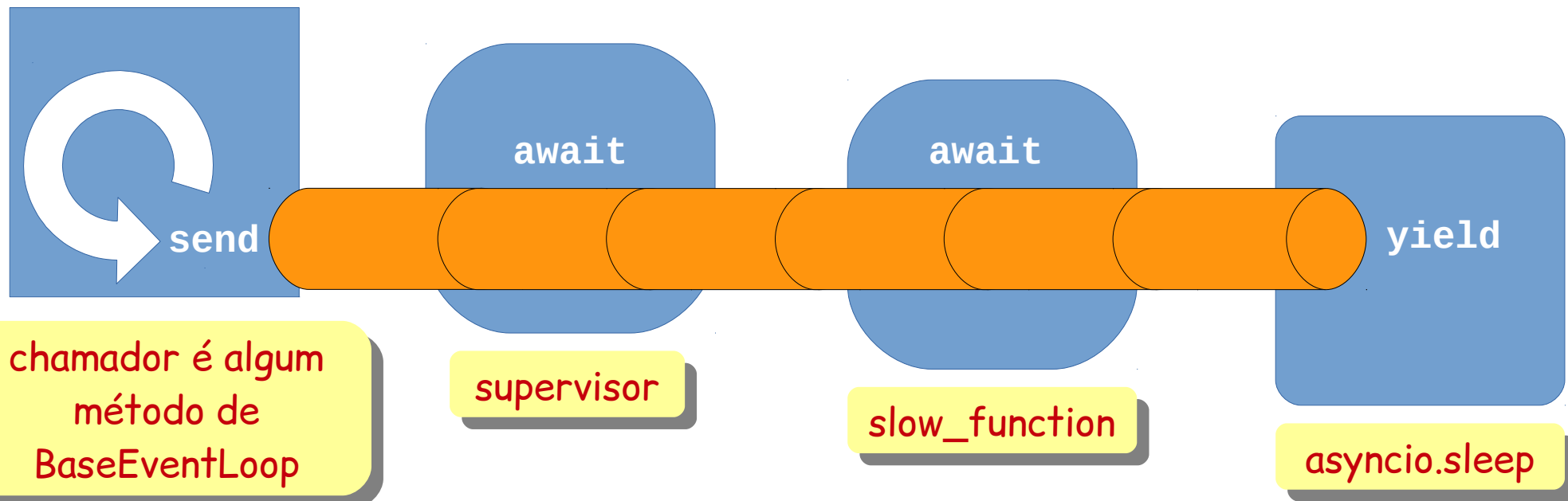
if __name__ == '__main__':
    main()
```



**await bloqueia
corrotina delegante
supervisor()**

await cria um canal

- Canal conecta laço de eventos com o último objeto *awaitable* na cadeia de delegações



Coro... spinner: final


ⓐ Aguarda resultado
de **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() é
acionada
diretamente pelo
loop de eventos

Coro... spinner: final

Ⓕ Delega para
asyncio.sleep

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

slow_function()
bloqueia aqui

```
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: final

Ⓕ Delega para
`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
```

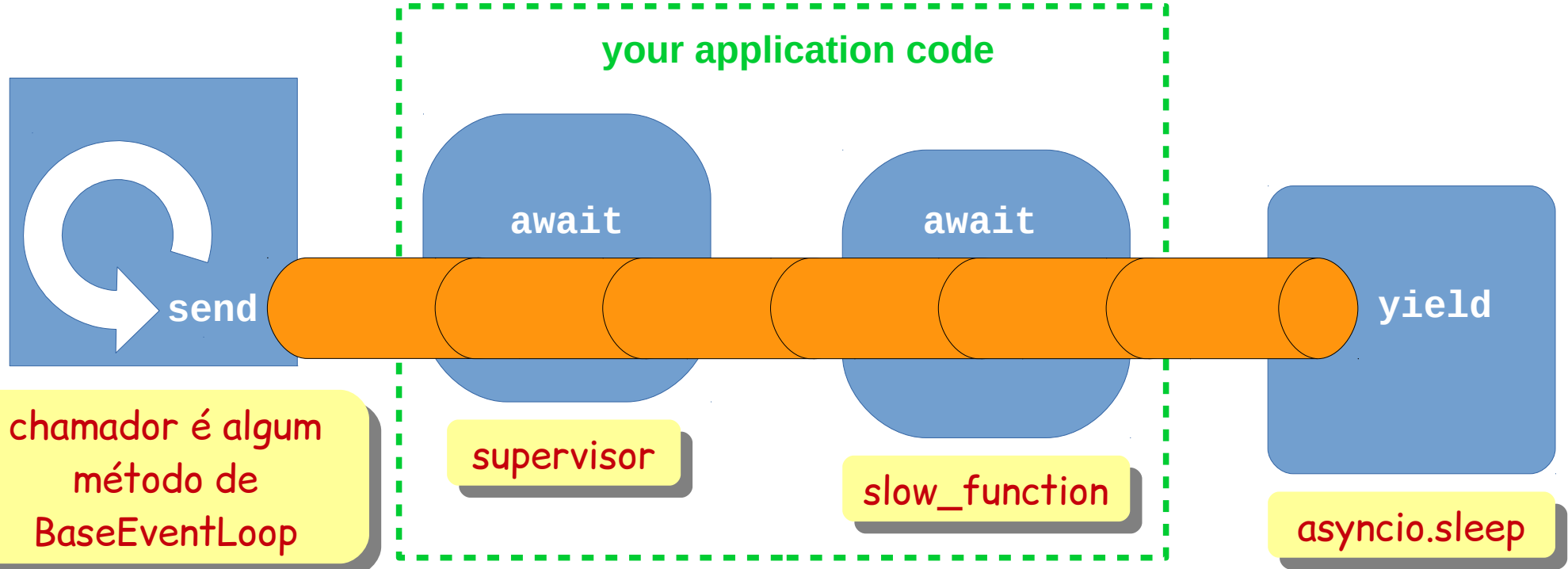
`asyncio.sleep()` configura um timer com `loop.call_later`, e passa o controle para o event loop

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

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

Programação assíncrona

- Desenvolvedora escreve funções que conectam o loop de eventos às funções da biblioteca que realizam I/O (ou “dormem”, nesse caso)



Coro... spinner: final

Ⓚ Quando **slow_function** retorna, cancelamos a tarefa **spinner**

```
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 podem ser canceladas com segurança porque só podem ser interrompidas nos pontos de suspensão (yield ou await)

corrotina spinner: início

Ⓒ Cada iteração é
suspensa por
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
```

corrotina spinner: início

- Ⓓ Trata a exceção de cancelamento saindo do laço infinito

```
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)
```

- **main** assíncrono gerencia o loop de eventos
- observe como **supervisor()** é invocado em cada implementação

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: comparando

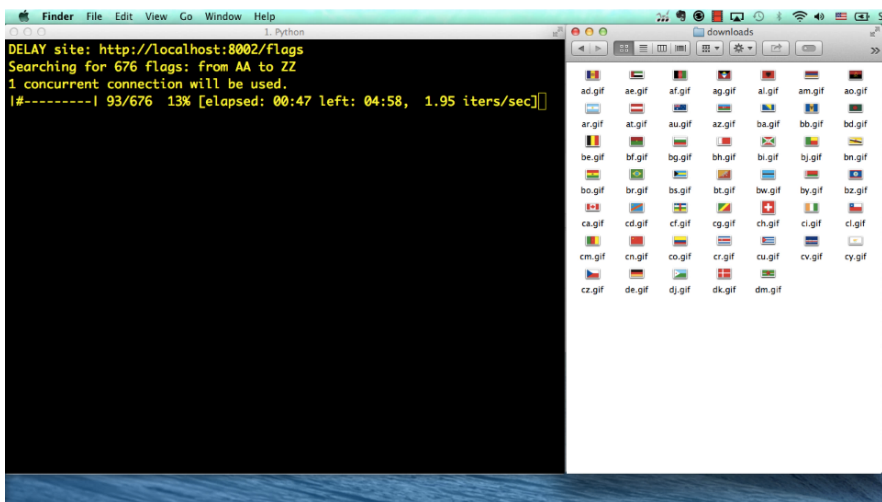
```
(.env35b3) $ python spinner_thread.py  
spinner object: <Thread(Thread-1, initial)>  
Answer: 42  
(.env35b3) $ python spinner_yield.py  
spinner object: <Task pending coro=<spin() running at spinner_yield.py:6>>  
Answer: 42
```

- ação **spinner** é implementada como **Thread** ou como **Task**
- **Task** assíncrona é semelhante a green thread
 - uma thread cooperativa gerenciada pelas bibliotecas da sua aplicação (e não pelo SO)
- **Task** embrulha uma corrotina
- Corrotina consome muito menos memória que thread (kilobytes, not megabytes)

flags_await.py

flags_await.py

- Implementação simplificada do demo de download de bandeiras



```
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

aciona muitas
instâncias de

download_one

⑦

download_one

delega para **get_flag**

④, ⑤

get_flag delega para
aiottp.request() e
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
```

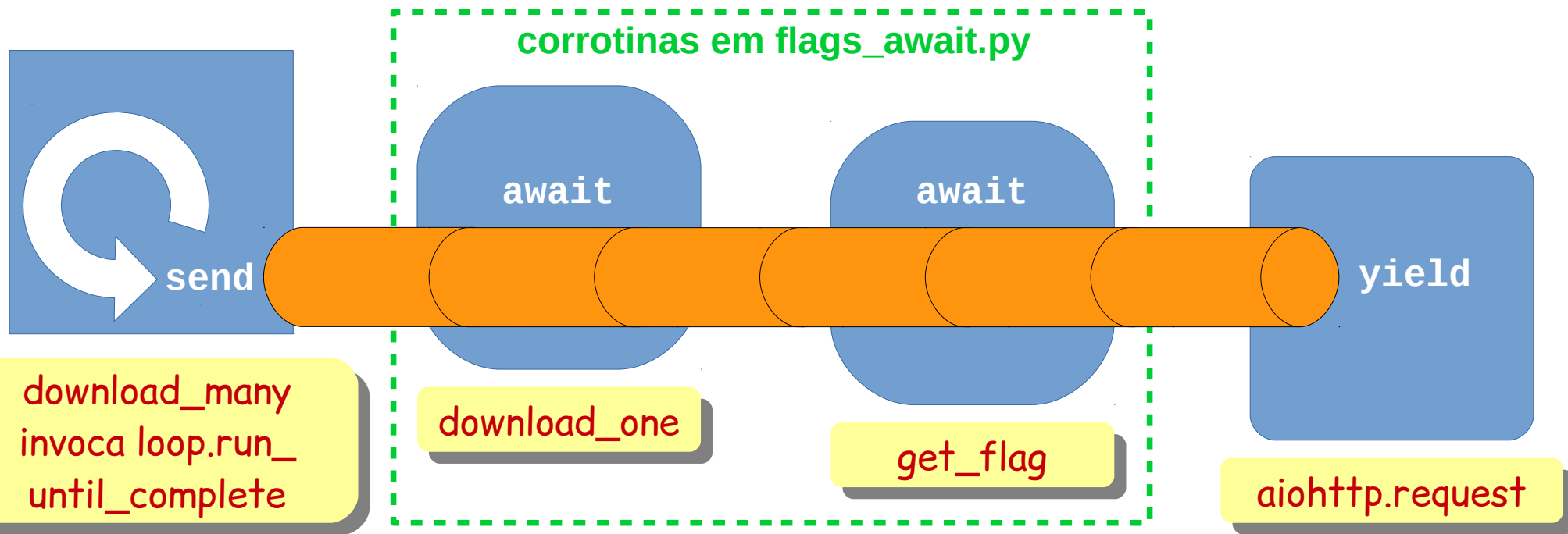
```
async def download_one(cc): # ⑥
    image = await get_flag(cc) # ③
    show(cc)
    save_flag(image, cc.lower() + '.gif')
    return cc
```

```
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 em ação

- Código da usuária encadeia corrotinas para fazer o laço de evento acionar corrotinas que realizam I/O de forma assíncrona



Mais zoom...

```
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
```

Mais zoom... e aperte os olhos

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

- Dica de Guido van Rossum's para ler código assíncrono com corrotinas:
 - aperte os olhos e ignore as palavras **await** (ou **yield from**)

Vale a pena tudo isso?

- Concorrência é sempre difícil
- A nova biblioteca **asyncio** e as novas palavras reservadas oferecem uma alternativa eficaz para as abordagens tradicionais
 - gerenciar threads e locks na unha
 - sobreviver ao inferno de callbacks (callback hell)

Callback hell em 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);  
        });  
    });  
});
```

contexto do estágio
1 não disponível

contextos dos estágios
1 e 2 não disponíveis

Callback hell em 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)
```

contexto do estágio
1 não disponível

contextos dos estágios
1 e 2 não disponíveis

Fuga do inferno dos callbacks

```
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
```

contexto é
preservado
por todos os
estágios:
tudo
acontece no
escopo local
da corrotina

Fuga (apertando os olhos)

```
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
```

contexto é
preservado
por todos os
estágios:
tudo
acontece no
escopo local
da corrotina

Antes de complicar
seu stack com uma
linguagem diferente,
experimente Python 3.3
ou superior



já tinha yield from!

Python 3.5 async/await

- Novas palavras reservadas, primeiras desde o Python 3.0 (2008)
- PEP-492 *muito resumidamente:*
 - **async def** para definir *corrotinas nativas*
 - **await** para delegar para objetos *awaitable*
 - corrotinas nativas; geradores-corrotinas decoradas; implementações do protocolo `__await__`
 - novas instruções disponíveis somente dentro de *corrotinas nativas*:
 - **async for**: métodos assíncronos `__aiter__` e `__anext__`
 - **async with**: métodos assíncronos `__aenter__` e `__aexit__`

Suporte nativo e
de verdade para
corrotinas!

sintaxe yield-from

```
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
```

sintaxe async/await

```
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
```

- I/O concorrente pode ser feito sem threads ou callbacks
 - sem threads ou callbacks no **seu** código, pelo menos
- Instâncias de **asyncio.Task** embrulham corrotinas
 - permitem cancelamento, esperar resultados e verificar o status da tarefa
- corrotinas acionadas com **await** (ou **yield from**) comportam-se como threads leves cooperativas
 - pontos de suspensão explícitos facilitam o raciocínio, a corretude e a depuração
 - milhares de corrotinas podem ser agendadas ao mesmo tempo, graças ao baixo *overhead* de memória (comparando com threads do OS)

Links & perguntas

ThoughtWorks®

- Repositório de código do **Fluent Python**:
 - <https://github.com/fluentpython/example-code>
 - novo exemplo com `async-await` no diretório **17-futures/countries/**
 - novo diretório **18b-async-await/** com os exemplos de **18-asyncio/** reescritos na nova sintaxe
- Slides para esta palestra (e muitas outras):
 - <https://speakerdeck.com/ramalho/>
- Contas no Twitter:
 - @ramalhoorg
 - @fluentpython, @pythonfluente

