Programação assíncrona com corrotinas em Python

Exemplo: Judit Pólgar*

- 20 adversários
- 40s por jogada
  - Pólgar 5s, adversário 35s
- Média de 30 jogadas por partida (20 min)
- 20 partidas sequenciais: 400min = 6h40min

* Adaptado de Michael Grinberg em *Asynchronous Python for the Complete Beginner*. Foto de Ed Yourdon (CC BY-NC-SA 1992)

Exemplo: Judit Pólgar assíncrona

- Partidas “simultâneas”
- Pólgar dá o lance inicial e vai pro próximo tabuleiro
- Uma volta completa leva 20 × 5s = 40s
- 30 lances e voltas × 40s = 1200s
- 20 partidas concorrentes completadas em em 20 min!

Demo

```python
from gallery import Gallery

num_images = 10

urls = get_sample_urls(15_000_000, num_images)
gallery = Gallery(num_images)
gallery.display()

async def display_pics(urls):
    async with AsyncClient() as client:
        tasks = [fetch(client, url) for url in urls]
        for i, coroutine in enumerate(asyncio.as_completed(tasks)):
            img_rec = await coroutine
            gallery.update(i, img_rec.pixels, img_rec.name)

await display_pics(urls)

print(f'TOTAL BYTES: {gallery.size:_}')
```

```
( 1)  14_739_448 bytes | USS_Akron_%28ZRS-4%29_in_flight_over_Manhattan%2C_circa_1931-1933.jpg
( 2)  15_193_424 bytes | Lady_Elliot_Island_SVII.jpg
( 3)  15_283_554 bytes | Chehel_Sotoun_Inside%2C_Isfahan_Edit1.jpg
( 4)  14_632_560 bytes | Red_telephone_box%2C_St_Paul%27s_Cathedral%2C_London%2C_England%2C_GB%2C_IM
G_5182_edit.jpg
( 5)  14_984_578 bytes | Il_Ballo2.jpg
( 6)  15_476_995 bytes | IND-%28NethEastInd%29-1-Dutch_Administration-1_Gulden_%281815%29.jpg
( 7)  14_784_640 bytes | Zaandam2.jpg
( 8)  14_541_999 bytes | Pahit-Pahit_Manis_pamphlet_%28obverse%29.jpg
( 9)  14_724_731 bytes | Wang_Ximeng._A_Thousand_Li_of_Rivers_and_Mountains._%28Complete%2C_51%2C3x1
191%2C5_cm%29._1113._Palace_museum%2C_Beijing.jpg
(10)  15_479_621 bytes | Meripilus_giganteus_%28Karst_1882%29.jpg
TOTAL BYTES: 149_841_550
```

Latência de E/S: ordens de grandeza*

dispositivo	ciclos de CPU	escala "humana"
cache L1	3	3 segundos
cache L2	14	14 segundos
RAM	250	250 segundos
armazenagem local	41.000.000	1,3 anos
rede	240.000.000	7,6 anos

* Números citados por Ryan Dahl, criador do Node.js no vídeo *Introduction to Node.js*

Modelos de programação assíncrona

- Callbacks
  - Forma mais antiga
  - Código difícil de manter
    - "callback hell"

```
function1(param, function(err, result) {
    function2(param, function(err, result) {
        function3(param, function(err, result) {
            function4(param, function(err, result) {
                function5(param, function(err, result) {
                    function6(param, function(err, result) {
                        function7(param, function(err, result) {
                            // Do Something finally
                        })
                    })
                })
            })
        })
    })
})
```

- Corrotinas
  - em vez de "sub-rotinas", "co-rotinas"
  - funções que podem ser suspensas enquanto aguardam algum dado
  - evitam o problema conhecido como "pyramid of doom"

Tudo começa com geradores

- Um gerador é uma função que contem a palavra reservada yield, onde a execução é suspensa, para continuar depois

Um gerador muito simples

```
>>> def gen_123():
...     yield 1  # (1)
...     yield 2
...     yield 3
...
>>> gen_123  # doctest: +ELLIPSIS
<function gen_123 at 0x...>  # (2)
>>> gen_123()   # doctest: +ELLIPSIS
<generator object gen_123 at 0x...>  # (3)
>>> for i in gen_123():  # (4)
...     print(i)
1
2
3
```

```
>>> g = gen_123()  # (5)
>>> next(g)  # (6)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)  # (7)
Traceback (most recent call last):
  ...
StopIteration
```

Outro gerador bem simples

```
>>> def gen_AB():
...     print('start')
...     yield 'A'           # (1)
...     print('continue')
...     yield 'B'           # (2)
...     print('end.')       # (3)
```

```
>>> for c in gen_AB():     # (4)
...     print('-->', c)    # (5)
...
start     (6)
--> A     (7)
continue  (8)
--> B     (9)
end.      (10)
>>>       (11)
```

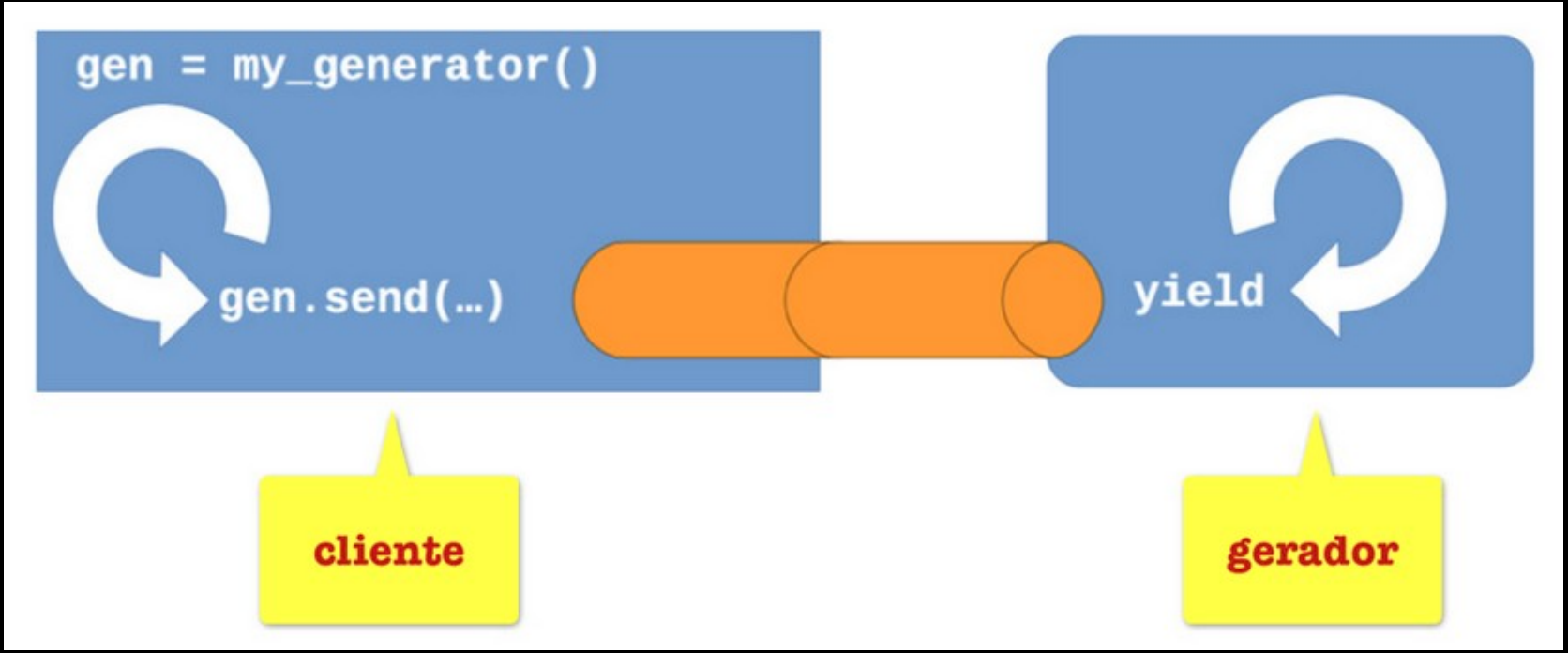
Geradores como corrotinas simples

- Usando gen.send() em vez de next(gen)

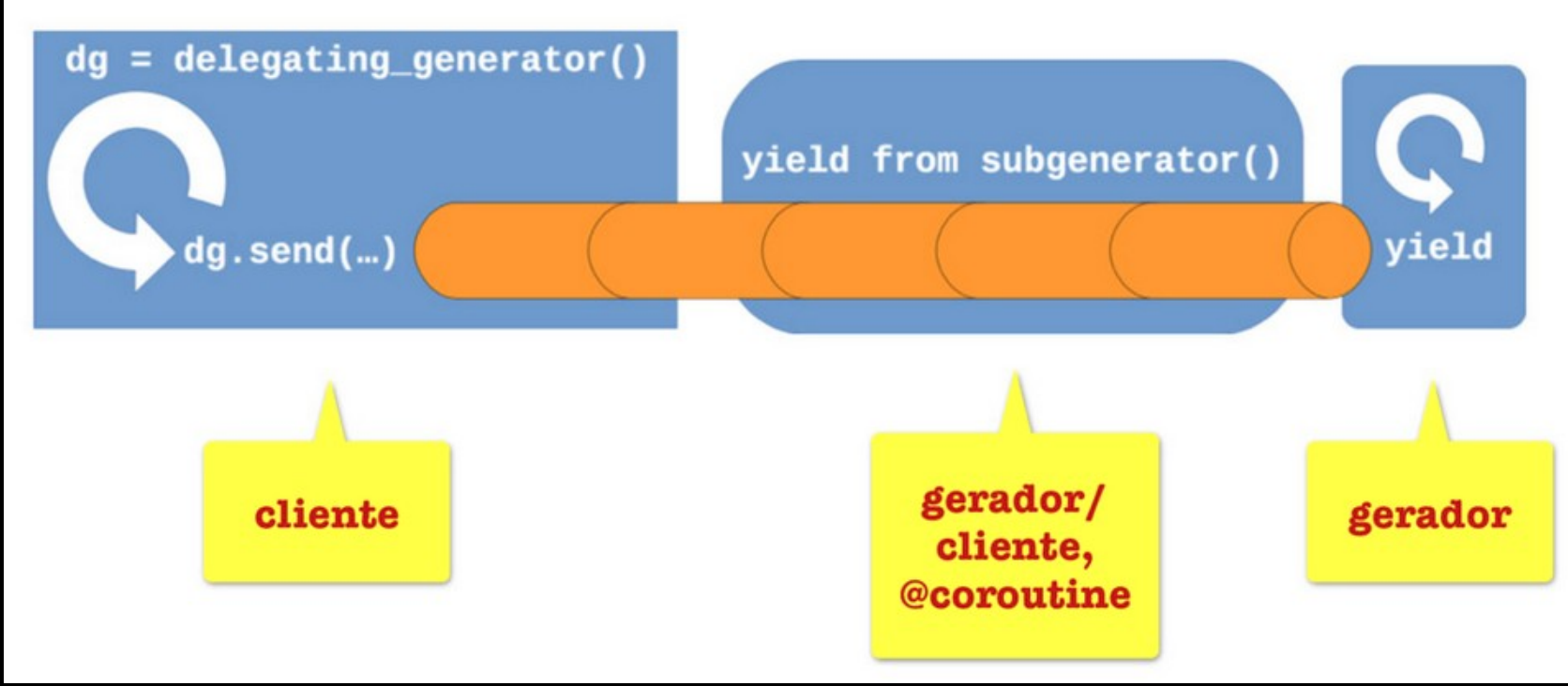
```
def averager():  # (1)
    total = 0.0
    count = 0
    average = 0.0
    while True:  # (2)
        term = yield average  # (3)
        total += term
        count += 1
        average = total/count
```

```
>>> coro_avg = averager()  # (1)
>>> next(coro_avg)  # (2)
0.0
>>> coro_avg.send(10)  # (3)
10.0
>>> coro_avg.send(30)
20.0
>>> coro_avg.send(5)
15.0
```

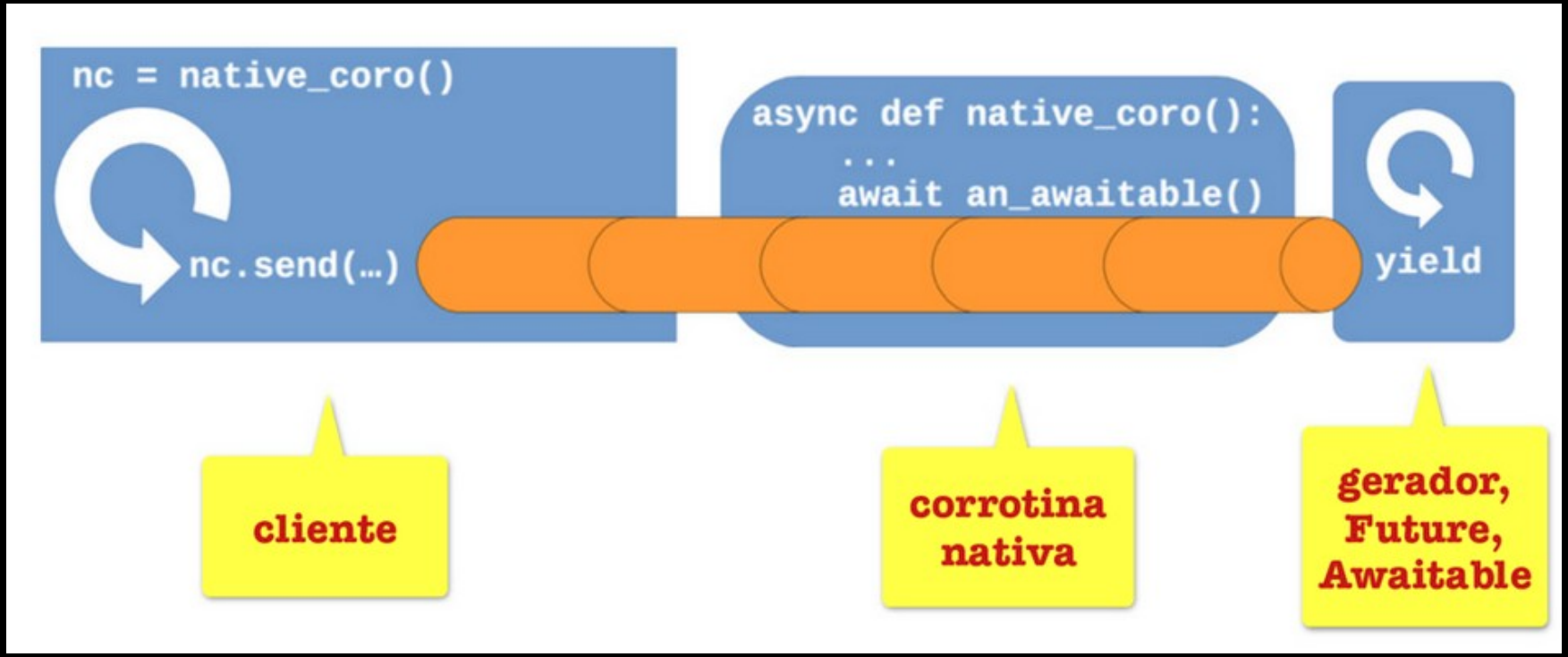
Concorrência com corrotinas (1)



Concorrência com corrotinas (2)



Concorrência com corrotinas (3)



Instruções async with e async for

```python
import asyncio
import aiopg

dsn = 'dbname=aiopg user=aiopg password=passwd host=127.0.0.1'

async def go():
    async with aiopg.create_pool(dsn) as pool:
        async with pool.acquire() as conn:
            async with conn.cursor() as cur:
                await cur.execute("SELECT 1")
                ret = []
                async for row in cur:
                    ret.append(row)
                assert ret == [(1,)]

loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

Métodos especiais assíncronos

instrução	sequencial	async
for	__iter__, __next__	__aiter__, __anext__
with	__enter__, __exit__	__aenter__, __aexit__

- Nas variantes async, os métodos em vermelho são corrotinas, para não bloquear o laço de eventos
- O __aiter__ não é corrotina, mas deve retornar um iterador assíncrono que implementa uma corrotina __anext__

Threads x processos x corrotinas

	threads	processos	corrotinas
uso de CPU em código Python	limitado a 1 núcleo da CPU para todas as threads executando bytecode Python	cada processo pode usar um núcleo da CPU	limitado a uma thread, mas pode usar executores de thread ou processo
uso de memória	memória alocada para um processo + cerca de 4MB por thread no mínimo	maior uso de memória: memória isolada para cada processo + threads	menor uso de memória
comunicação entre unidades de execução	todas as threads podem acessar os mesmos objetos na memória	comunicação entre processos é mais lenta e complicada	todas as corrotinas podem acessar os mesmos objetos na memória
custo de inicialização	criar uma thread custa menos que criar um processo	custo maior: multiplicado pelo número de processos	menor custo de inicialização por usar só uma thread por padrão