Corrotinas III

Live de Python #154

Roteiro

1. Controle de Fluxo
Entendendo de vez o yield

2. Geradores melhorados
Yield como expressão

3. Delegando para sub geradores
Cedendo a vez para outro gerador

4. asyncIO
A forma final das corrotinas

Roteiro

1. Contr[illegible]luxo
Entender [illegible] yield

2. Gera[illegible]lhorados
Yield como [illegible]ão

3. Delegando [illegible]ub geradores
Cedendo a vez p[illegible]rador

4. asyncIO
A forma final das corrotinas

picpay.me/dunossauro

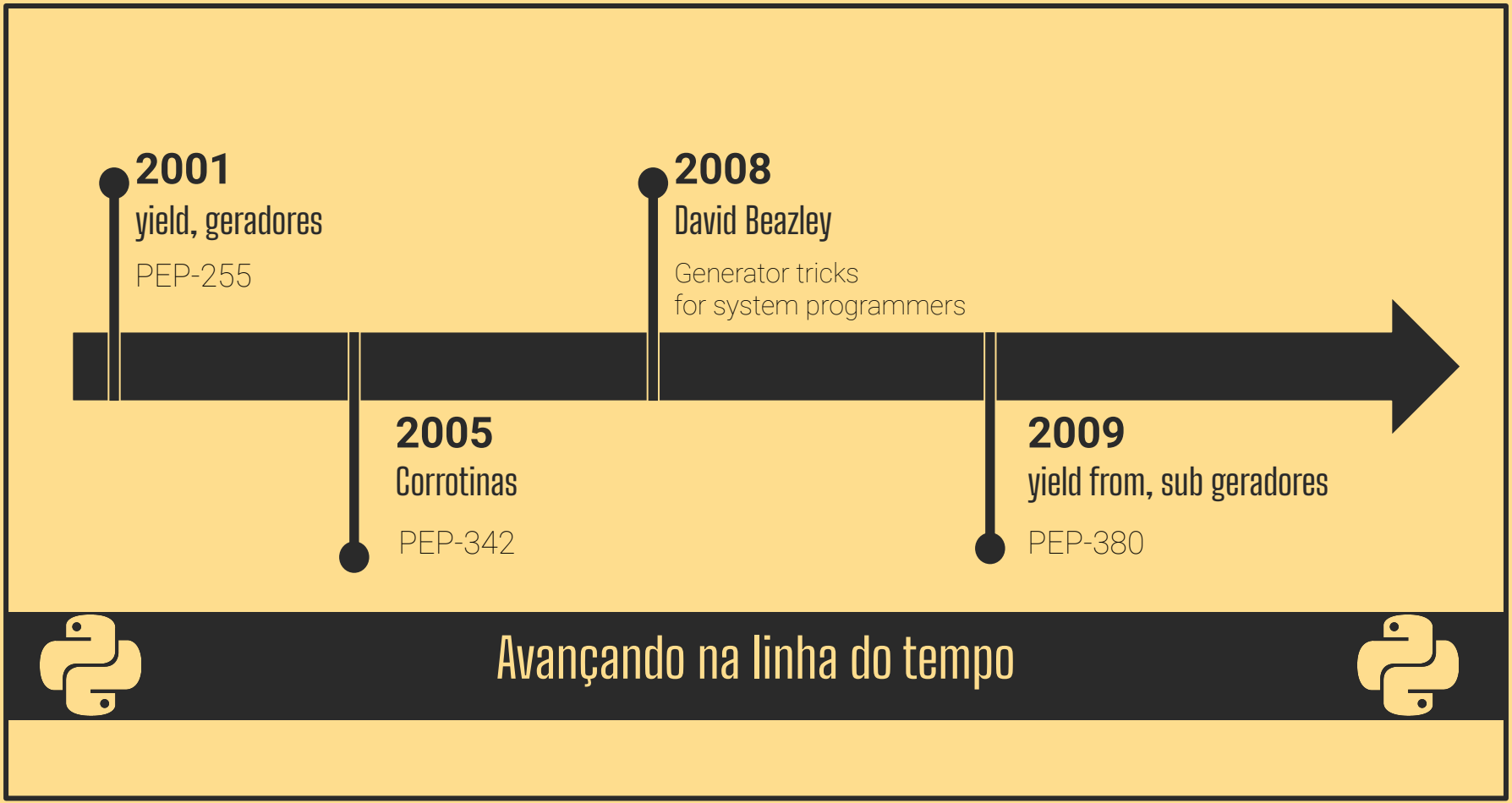
apoia.se/livedepython

PIX

Ajude o projeto

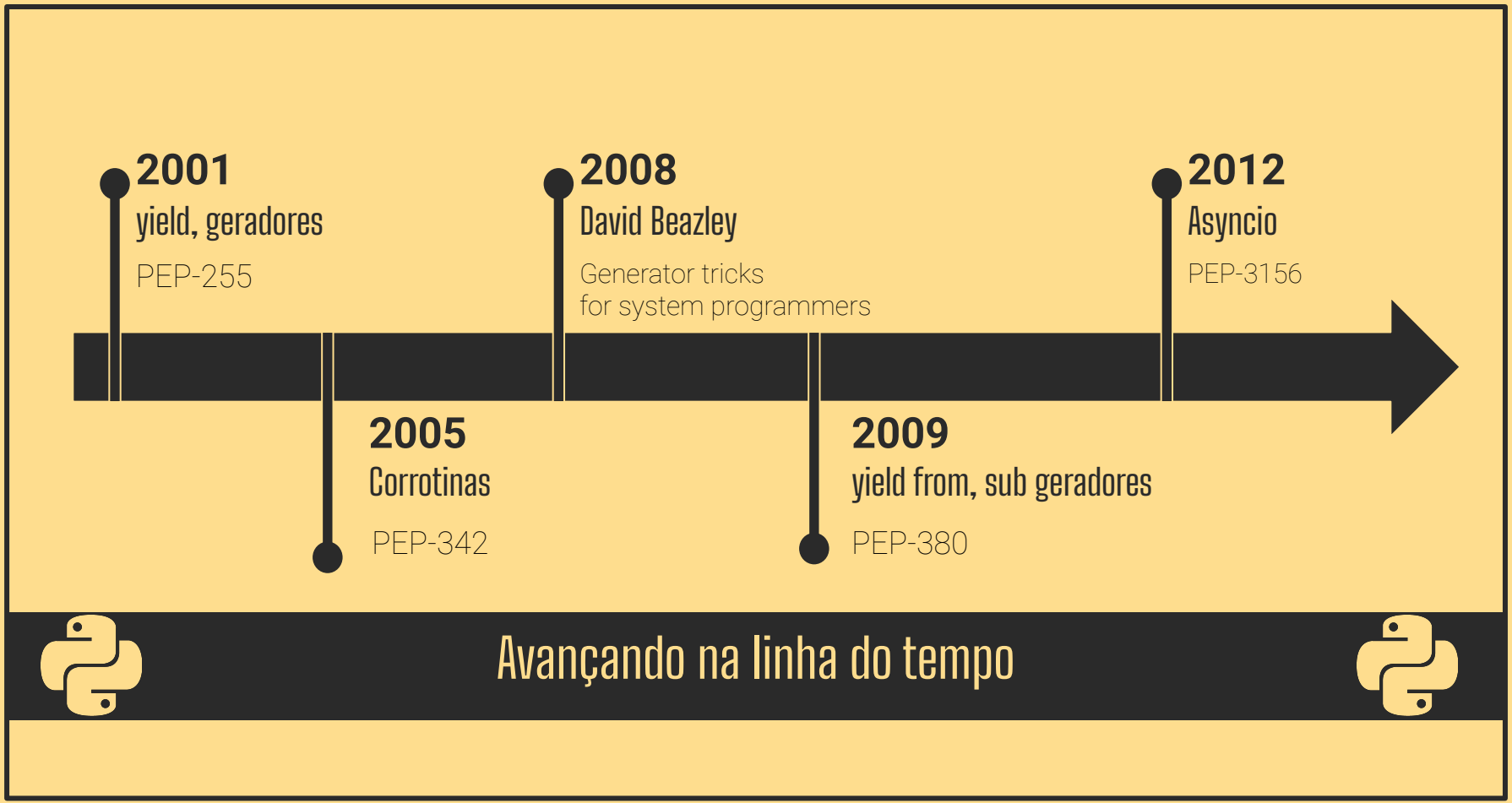
Ademar Peixoto, Alex Lima, Alexandre Harano, Alexandre Santos, Alexandre Souza, Alexandre Tsuno, Alysson Oliveira, Amaziles Carvalho, Anderson Araujo, Andre Rodrigues, André Rocha, Antonio Neto, Arnaldo Turque, Bianca Rosa, Bruno Oliveira, Caio Nascimento, Carlos Cardoso, Carlos Chiarelli, César Almeida, César Moreira, Davi Ramos, David Kwast, Diego Guimarães, Dilenon Delfino, Douglas Bastos, Edgard Sampaio, Elias Soares, Eugenio Mazzini, Everton Alves, Fabio Barros, Fabio Castro, Fabrício Coelho, Flavkaze Flavkaze, Franklin Silva, Fábio Serrão, Gabriel Simonetto, Gabriel Soares, Gabriela Santiago, Geandreson Costa, Guilherme Felitti, Guilherme Marson, Guilherme Ostrock, Haroldo Júnior, Henrique Machado, Hélio Neto, Isaac Ferreira, Israel Fabiano, Italo Silva, Jeison Sanches, Johnny Tardin, Jonatas Leon, Jorge Plautz, José Prado, Jovan Costa, João Lugão, João Schiavon, Juan Gutierrez, Jônatas Silva, Júlia Kastrup, Kaneson Alves, Leonardo Cruz, Leonardo Galani, Leonardo Mello, Lorena Ribeiro, Lucas Barros, Lucas Ferreira, Lucas Mello, Lucas Mendes, Lucas Teixeira, Lucas Valino, Luiz Lima, Maiquel Leonel, Maiquel Leonel, Marcela Campos, Marcelo Rodrigues, Maria Clara, Natan Cervinski, Nicolas Teodosio, Nilo Pereira, Otavio Carneiro, Patric Lacouth, Patricia Minamizawa, Patrick Gomes, Paulo Tadei, Pedro Andrade, Pedro Pereira, Peterson Santos, Reinaldo Silva, Rodrigo Campos, Rodrigo Ferreira, Rodrigo Vaccari, Ronaldo Silva, Rubens Gianfaldoni, Sandro Mio, Silvio Xm, Thiago Araujo, Thiago Borges, Thiago Bueno, Tyrone Damasceno, Valdir Junior, Victor Geraldo, Vinícius Bastos, Vinícius Ferreira, Wesley Mendes, Willian Lopes, Willian Lopes, Willian Rosa, Wilson Duarte

Obrigado você


2001
yield, geradores
PEP-255
2008
David Beazley
Generator tricks
for system programmers
2005
Corrotinas
PEP-342
2009
yield from, sub geradores
PEP-380
Avançando na linha do tempo

Async IO

PEP-3156, corrotinas assíncronas (2012)


2001
yield, geradores
PEP-255
2005
Corrotinas
PEP-342
2008
David Beazley
Generator tricks
for system programmers
2009
yield from, sub geradores
PEP-380
2012
Asyncio
PEP-3156
Avançando na linha do tempo

O módulo asyncio (2012)

Em 2012, a PEP-3156 (Suporte IO assíncrono reiniciado: o módulo "asyncio"), nos apresenta a ideia de corrotinas assíncronas e mais alguns conceitos:

- Gather
- Loop de eventos
- Task
- Future

Corrotina assíncrona [exemplo_12.py]

Agora o “primer” da corrotina é feita com o decorador do módulo asyncio

```
from asyncio import coroutine

@coroutine
def coro():
    print(42)
```

Corrotina assíncrona [exemplo_12.py]

Para executar uma corrotina async, precisamos de um loop de eventos

```python
from asyncio import coroutine

@coroutine
def coro():
    print(42)
```

```python
from asyncio import get_evet_loop

loop = get_event_loop()

loop.run_until_complete(coro())
```

Corrotina assíncrona [exemplo_12.py]

Para executar uma corrotina async, precisamos de um loop de eventos

```python
from asyncio import coroutine

@coroutine
def coro():
    print(42)
```

```python
type(coro)  # function
type(coro())  # generator
```

```python
from asyncio import get_evet_loop

loop = get_event_loop()

loop.run_until_complete(coro())
```

Voltando a PEP-380

Agora que temos o conceito do cliente sendo o loop de eventos para execução de corrotinas, podemos encadear subgeradores.

Subgeradores [exemplo_13.py]

```python
from asyncio import get_event_loop, coroutine

@coroutine
def subgenerator():
    return 42

@coroutine
def coro():
    val1 = yield from subgenerator()
    val2 = yield from subgenerator()
    return val1 + val2


loop = get_event_loop()

loop.run_until_complete(coro())  # 84
```

subgerador

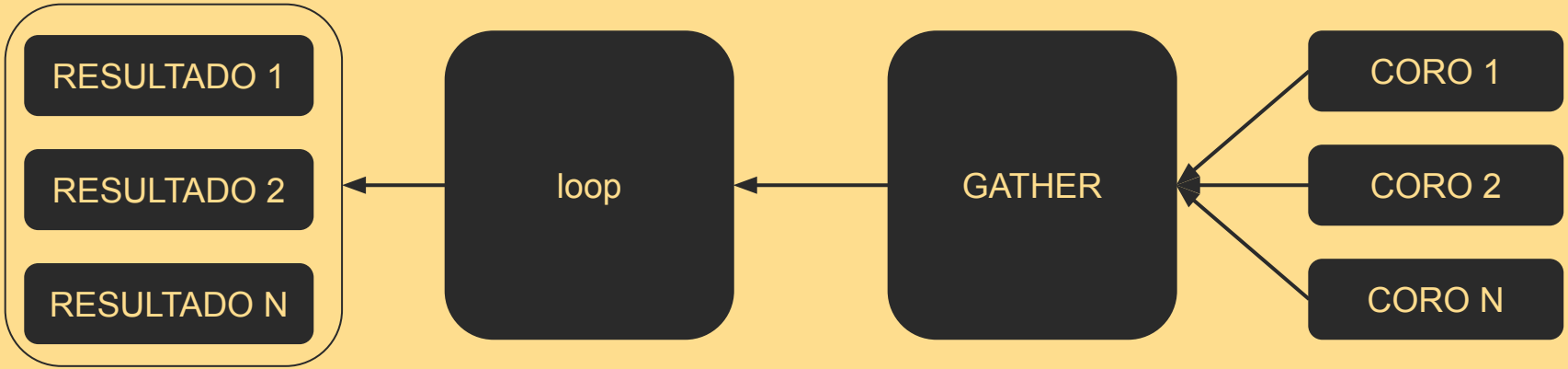
delegante

loop / client

O encadeamento de subgeradores permanece igual. Somente adicionamos agora o decorador de `coroutine`

Gather

O pacote asyncio trouxe consigo uma função muito poderosa para o uso em conjunto com as corrotinas, o **gather**. Que pode ser traduzido como “agregador”. Pense nele como um agregador de corrotinas.



Gather [exemplo_14.py]

Dessa forma, o loop fica responsável por rodar todas as tarefas de maneira concorrente.

A resposta do gather é uma lista, com todas as corrotinas que foram passadas, de maneira ordenada.

```python
from asyncio import (
    coroutine, get_event_loop, gather
)

@coroutine
def subgenerator():
    return 42

@coroutine
def coro():
    val1 = yield from subgenerator()
    val2 = yield from subgenerator()
    return val1 + val2


loop = get_event_loop()
grupo = gather(coro(), coro(), coro())
loop.run_until_complete(grupo)  # [84, 84, 84]
```

Tá, mas não vi nada de concorrência até agora

tá de sacanagem?

Um exemplo simples com gather [exemplo_15.py]

```python
from asyncio import (
    coroutine, get_event_loop, gather, sleep
)
```

```python
@coroutine
def subgenerator():
    yield from sleep(1)
    return 42


@coroutine
def coro():
    val1 = yield from subgenerator()
    val2 = yield from subgenerator()
    return val1 + val2
```

```python
loop = get_event_loop()
grupo = gather(*[coro() for i in range(20)])
result = loop.run_until_complete(grupo)
# [84, ..., 84]
```

Um salto no tempo

Sei que alguns conceitos ainda estão em aberto, como:

- EventLoop
- Task
- Future

Mas vamos avançar o tempo e voltaremos neles.

Async IO

PEP-492
Async / Await
(2015)


2001
yield, geradores
PEP-255
2005
Corrotinas
PEP-342
2008
David Beazley
Generator tricks
for system programmers
2009
yield from, sub geradores
PEP-380
2012
Asyncio
PEP-3156
2015
async / await
PEP-492
Avançando na linha do tempo

Corrotinas e asyncIO

Na PEP-429 (Corrotinas com sintaxe async e await), foram introduzidas duas palavras reservadas novas na linguagem: **async** e **await** em contrapartida ao modo **@coroutine** e **yield from**.

```
from asyncio import coroutine

@coroutine
def coro():
  result = yield from subgenerator()
  return result
```

```
async def coro():
  return await coroutine()
```

PEP-249

Embora nosso objeto de estudo nesse séria sejam as corrotinas. A PEP-249 introduz o conceito de **async** e **await** em diversas outras frentes. Como:

- Gerenciados de contextos assíncronos (**async with**)
- Iteradores assíncronos (**async for**)

Vale lembrar que essas chamadas só podem ser executadas dentro de corrotinas.

Aplicando a PEP-249 [exemplo_16.py]

Vamos usar como base nosso exemplo_15.py e aplicar a nova sintaxe do python para simplificar as chamad

```python
from asyncio import get_event_loop, gather, sleep


async def subgenerator():
    await sleep(1)
    return 42


async def coro():
    val1 = await subgenerator()
    val2 = await subgenerator()
    return val1 + val2


loop = get_event_loop()
grupo = gather(*[coro() for i in range(20)])
loop.run_until_complete(grupo)  # [84, ..., 84]
```

Uma comparação

```python
from asyncio import (
    coroutine, get_event_loop, gather
)

@coroutine
def subgenerator():
    return 42

@coroutine
def coro():
    val1 = yield from subgenerator()
    val2 = yield from subgenerator()
    return val1 + val2


loop = get_event_loop()
grupo = gather(coro(), coro(), coro())
loop.run_until_complete(grupo)  # [84, 84, 84]
```

```python
from asyncio import get_event_loop, gather, sleep


async def subgenerator():
    await sleep(1)
    return 42


async def coro():
    val1 = await subgenerator()
    val2 = await subgenerator()
    return val1 + val2


loop = get_event_loop()
grupo = gather(*[coro() for i in range(20)])
loop.run_until_complete(grupo)  # [84, ..., 84]
```

Finalmente o tipo corrotina

Agora, a instância de uma corrotina é de fato uma corrotina, não mais um generator

```
async def coro():
    ...

type(coro)    # function
type(coro())  # coroutine
```

BORA FAZER USO DESSAS CORROTINAS?

E agora?

Corrotinas + HTTPX

Vamos montar uma corrotina que faz um request para baixar pokemons?

```
async def subgenerator(number):
    async with AsyncClient() as client:
        response = await client.get(
            base_url.format(number=number),
            timeout=None
        )
        print(number)
        return number, response.json()['name']
```

Agora vamos unir isso ao gather

Para gerar corrotinas de requests de maneira concorrente

```
async def coro():
    return await gather(
        *[subgenerator(number) for number in range(1, 10)]
    )
```

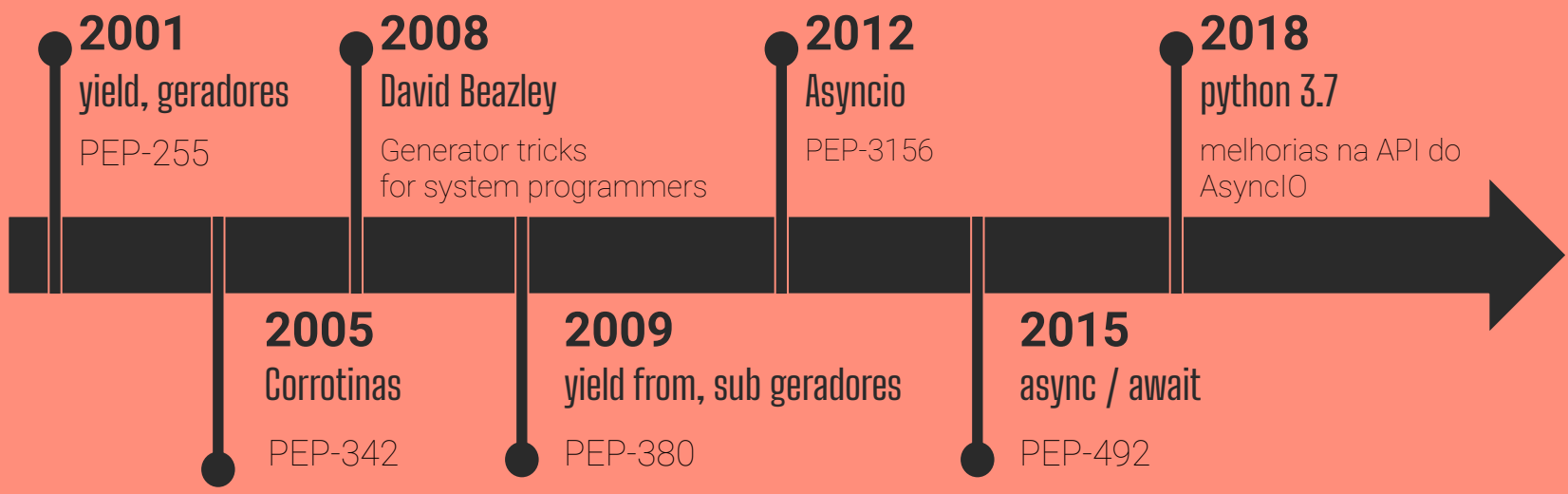
Tudo junto e rodando

```
/Live153 on master*
$ python exemplo_17.py
3
8
9
5
4
1
2
6
7
[(1, 'bulbasaur'),
 (2, 'ivysaur'),
 (3, 'venusaur'),
 (4, 'charmander'),
 (5, 'charmeleon'),
 (6, 'charizard'),
 (7, 'squirtle'),
 (8, 'wartortle'),
 (9, 'blastoise')]
```

```python
from asyncio import get_event_loop, gather
from httpx import AsyncClient

base_url = 'https://pokeapi.co/api/v2/pokemon/{number}'

async def subgenerator(number):
    async with AsyncClient() as client:
        response = await client.get(
            base_url.format(number=number),
            timeout=None
        )
        print(number)
        return number, response.json()['name']

async def coro():
    return await gather(
        *[subgenerator(number) for number in range(1, 10)]
    )


loop = get_event_loop()
result = loop.run_until_complete(coro())
print(result)
```

Melhorias na versão 3.7 (2018)

Async IO



Avançando na linha do tempo

Simplificação para rodar corrotinas

Agora, embora de maneira provisória, a biblioteca asyncio ganhou novas chamadas que são interessantes para o nosso contexto:

- **asyncio.run**
  - Para executar uma corrotina
- loop
  - loop.call_soon
    - chama no inicio do loop
  - loop.call_later
    - chama depois de X segundos
  - loop.call_at
    - chama em um horário pré programado

asyncio.run

Com o **run**(), agora não é mais preciso, para executar corrotinas, fazer a chamada explícita do loop de eventos.

```python
from asyncio import run

async def coro():
    return 1

run(coro())  # 1
```

Simplificando com run

[exemplo_18.py]

Agora as chamadas de controle do loop não não mais necessárias.

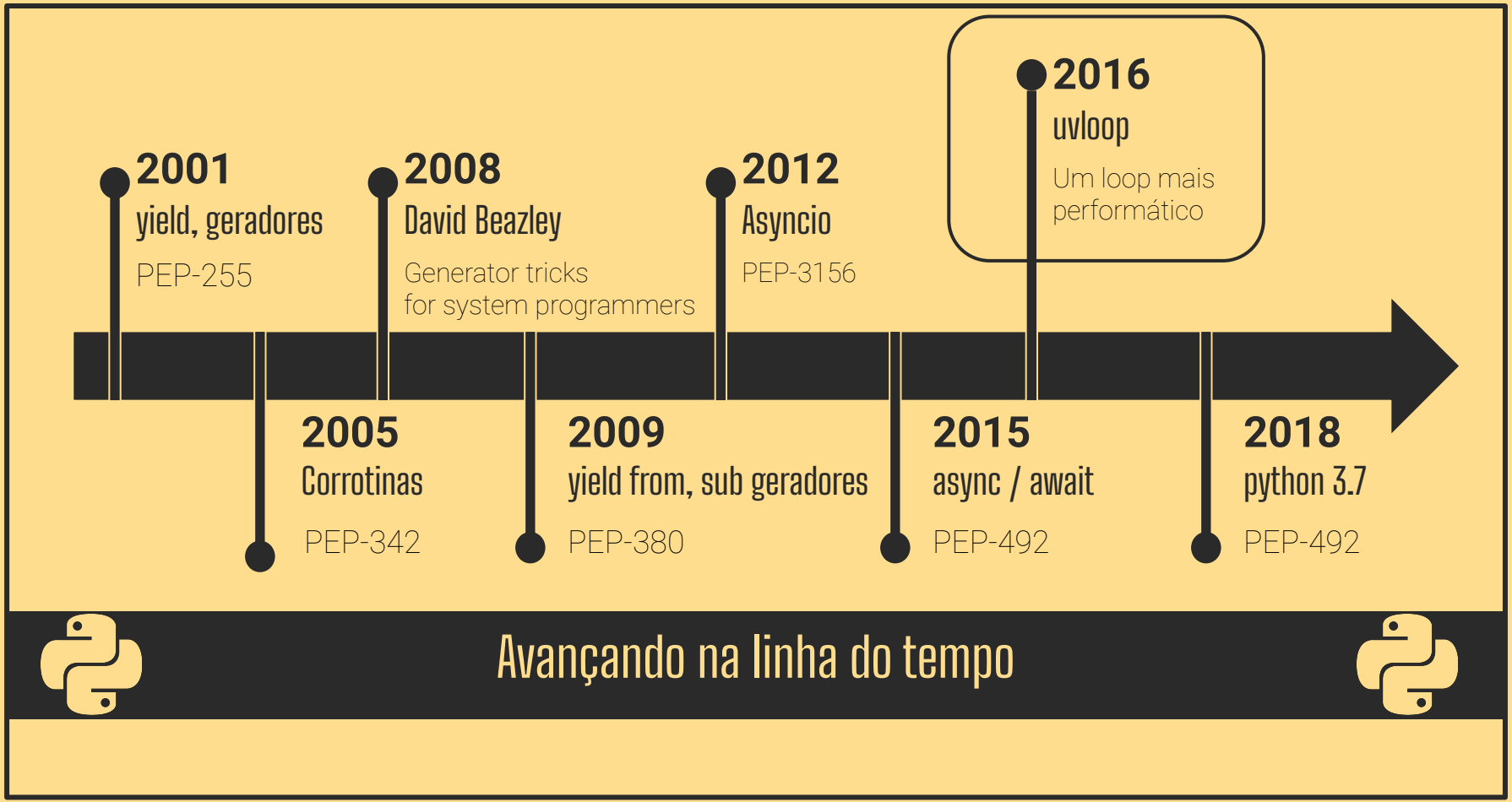
Porém ...

```python
from asyncio import gather, run
from httpx import AsyncClient

base_url = 'https://pokeapi.co/api/v2/pokemon/{number}'

async def downlaod(number):
    async with AsyncClient() as client:
        response = await client.get(
            base_url.format(number=number),
            timeout=None
        )
        print(number)
        return number, response.json()['name']

async def coro(start, stop):
    return await gather(
        *[downlaod(number) for number in range(start, stop)]
    )

run(coro(1, 5))
```

Usando o loop a nosso favor [exemplo_19.py]

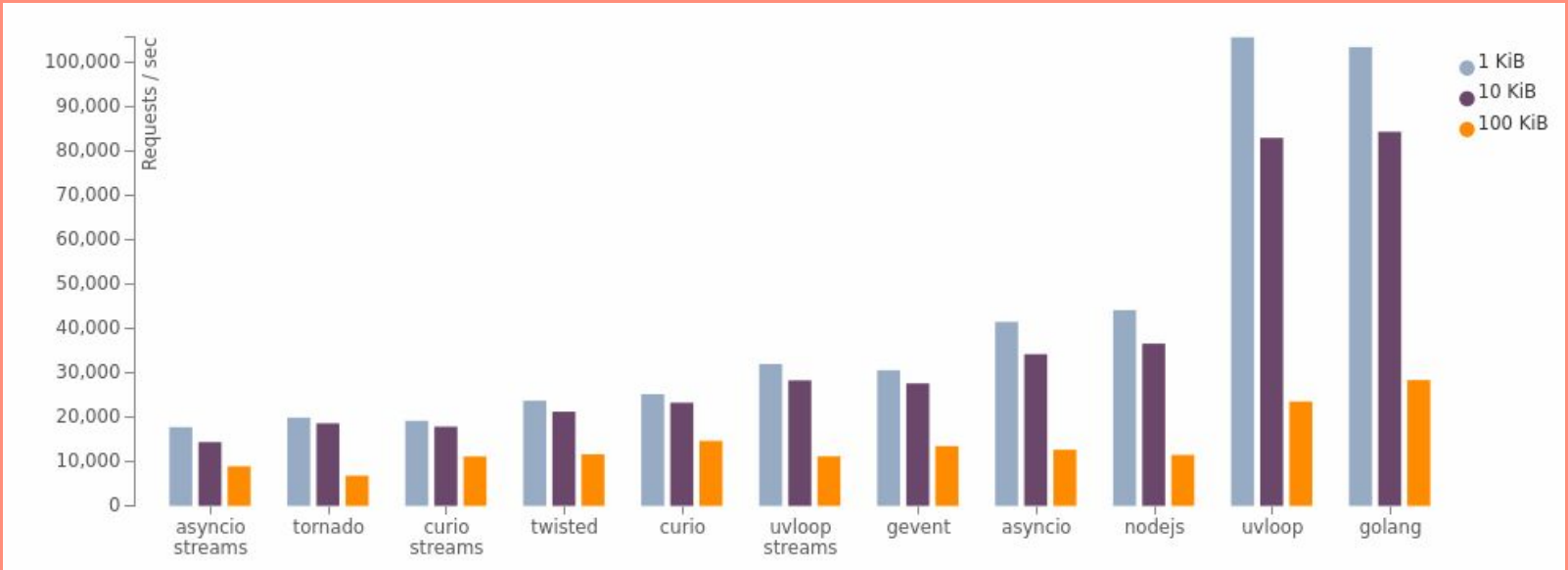
Com as novas chamadas do loop, podemos criar schedulers para rodar nossos requests

```python
loop = get_event_loop()

loop.call_soon(
    gather,
    *[download(number) for number in range(5, 10)]
)

loop.call_later(
    10,
    gather,
    *[download(number) for number in range(1, 5)]
)
loop.call_later(
    10,
    gather,
    *[download(number) for number in range(90, 100)]
)

loop.run_forever()
```


2001
yield, geradores
PEP-255
2005
Corrotinas
PEP-342
2008
David Beazley
Generator tricks
for system programmers
2009
yield from, sub geradores
PEP-380
2012
Asyncio
PEP-3156
2015
async / await
PEP-492
2016
uvloop
Um loop mais
performático
2018
python 3.7
PEP-492
Avançando na linha do tempo

uvloop

uvloop é uma alternativa ao loop padrão do python, bem mais performático. Recomendado para uso em produção

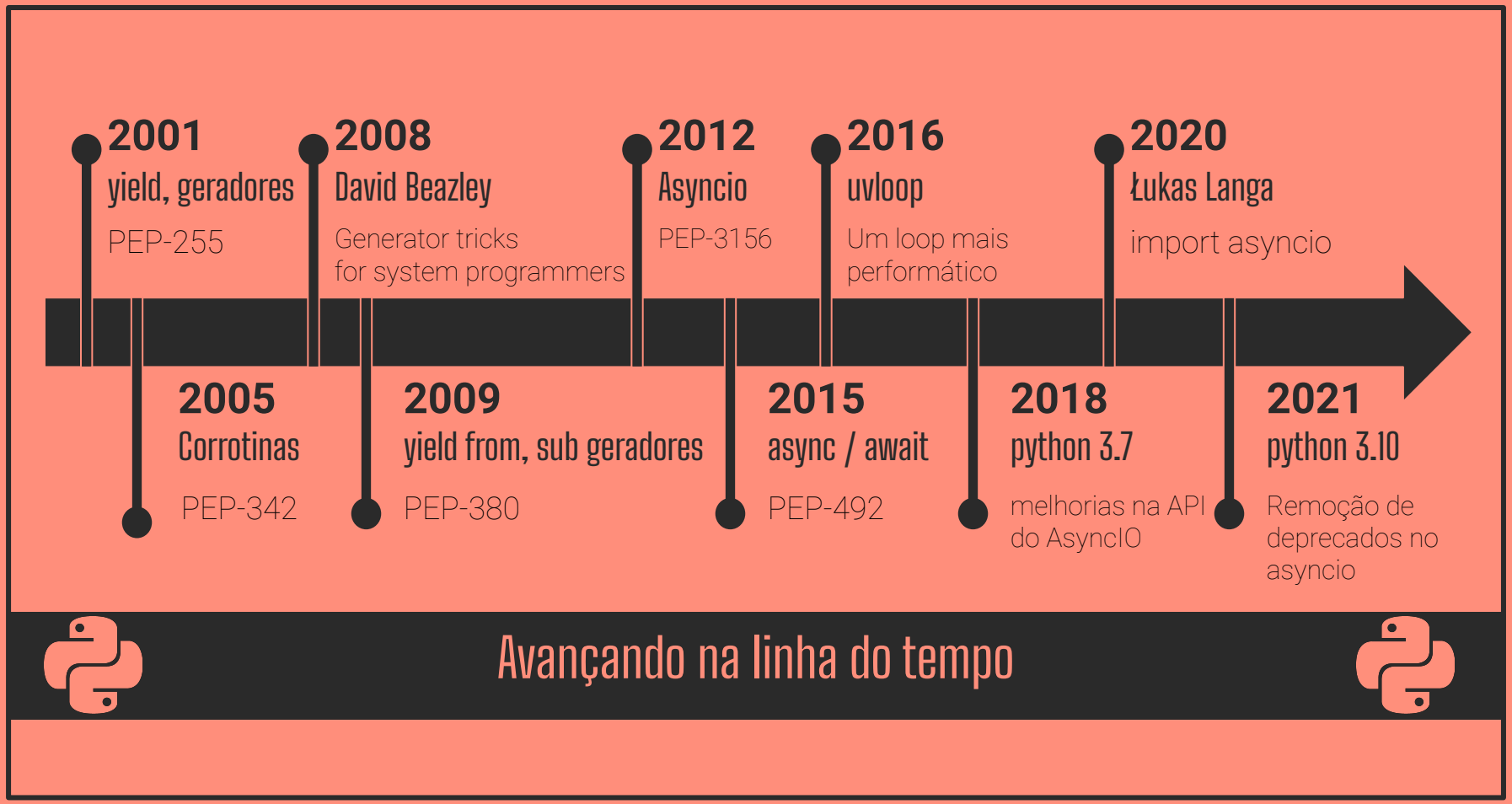


https://uvloop.readthedocs.io/

uvloop

Para usar o uvloop você precisa somente mudar uma chamada do asyncio. O que torna a modificação uma coisa simples.

```
>>> import asyncio

>>> type(asyncio.get_event_loop())
# asyncio.unix_events._UnixSelectorEventLoop

>>> import uvloop

>>> asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())

>>> type(asyncio.get_event_loop())
# uvloop.Loop
```


2001
yield, geradores
PEP-255
2005
Corrotinas
PEP-342
2008
David Beazley
Generator tricks
for system programmers
2009
yield from, sub geradores
PEP-380
2012
Asyncio
PEP-3156
2015
async / await
PEP-492
2016
uvloop
Um loop mais
performático
2018
python 3.7
melhorias na API
do AsyncIO
2020
Łukas Langa
import asyncio
2021
python 3.10
Remoção de
deprecados no
asyncio
Avançando na linha do tempo

E o que faltou?

Faltaram mais detalhamentos de alguns tópicos:

- Loop de eventos
  - Detalhamento do funcionamento interno
  - Alternativas como uvloop e trio
- Tasks / Futures
  - Explicação concreta de como se manifestam os wrappers

picpay.me/dunossauro

apoia.se/livedepython

PIX

Ajude o projeto