yield e funções geradoras

Live de python #151

Roteiro

1. Yield
De onde vem e o que significa?

2. Geradores
Do que se alimentam?

3. Produzindo
Gerando os valores cremosos

4. Sub geradores
Geradores com geradores

5. Expressões geradoras
Encurtando o caminho

6. Gerenciadores de contexto
Criando contextos

7. Corrotinas (breve)
Conversando com yield

picpay.me/dunossauro

apoia.se/livedepython

PIX

Ajude o projeto

Ademar, Alex Menezes, Alexandre Fernandes, Alexandre Harano, Alexandre Souza, Alexandre Tsuno, Alysson Oliveira, Amaziles Jose, Andre Rodrigues, André Almeida, Antonio Cassiano, Bianca Rosa, Bruno Fernandes, Bruno Rocha, Caio Vinicius, Carlos Alberto, César Moreira, César Túlio, Davi Alves, David Kwast, Diego Moreira, Dilenon Stefan, Douglas Bastos, Edgard Sampaio, Edivaldo Venancio, Edson Braga, Eduardo Marcos, Eduardo Sidney, Elias Da, Eugenio Mazzini, Everton Alves, Fabio Castro, Fabrício Vilela, Faricio Lima, Fernando Lanfranchi, Flavkaze, Franklin Sousa, Fábio Serrão, Gabriel Simonetto, Gabriela Santiago, Geandreson Costa, Gladson Araujo, Guilherme Felitti, Guilherme Marson, Guilherme Ostrock, Henrique Machado, Hélio De, Isaac Ferreira, Israel Azevedo, Italo Bruno, Jeison Sanches, Johnny Tardin, Jonatas Baldin, Jonatas Leon, Jones Ferreira, Jorge Luiz, José Willia, Jovan Costa, João Lugão, João Paulo, Juan Ernesto, Jônatas Silva, Júlia Kastrup, Kaneson Alves, Leonardo Cordeiro, Leonardo Galani, Leonardo Ribeiro, Lorena Carla, Lucas Barreto, Lucas Barros, Lucas Ferreira, Lucas Sartor, Luiz Lima, Maiquel Leonel, Maiquel Leonel, Marcela M, Marcelo Pontes, Maria Clara, Natan Cervinski, Nicolas Teodosio, Otavio Carneiro, Patric Lacouth, Patrick Henrique, Paulo Henrique, Paulo Henrique, Pedro Baesse, Pedro Martins, Peterson W, Rafael De, Reinaldo Chaves, Renan Gomes, Renne Rocha, Rodrigo Ferreira, Rodrigo Vaccari, Ronaldo Fraga, Rubens Gianfaldoni, Sandro Roberto, Silvio Xm, Thiago Dias, Thiago Martins, Tyrone Damasceno, Valdir Junior, Victor Matheus, Vinícius Bastos, Vinícius Borba, Wesley Mendes, Willian Lopes, Willian Lopes, Willian Vieira, Wilson Beirigo

Obrigado você

De onde vem e o que significa?

Yield

Yield

O yield foi introduzido na PEP-255. A motivação para sua introdução é facilitar o sistema “respostas” de maneira preguiçosa, sem a necessidade de um sistema de callbacks. O yield foi introduzido na versão 2.3 da linguagem

A tradução da palavra yield é **produzir** ou **produção**.

Yield

A ideia central é ter uma função que sabe produzir valores. Como assim?

```
1 def função_geradora():
2     yield 1
```

Yield

A ideia central é ter uma função que sabe produzir valores. Como assim?

```
1 def função_geradora():
2     yield 1
3
4 >>> função_geradora()
5 # <generator object função_geradora at 0x7f567ef09040>
```

Yield

A ideia central é ter uma função que sabe produzir valores. Como assim?

```
1 def função_geradora():
2     yield 1
3
4 >>> função_geradora()
5 # <generator object função_geradora at 0x7f567ef09040>
```

Yield

A função geradora é um objeto função, porém quando “chamada” ela se comporta como um gerador. Toda função a palavra ‘yield’ no seu corpo é um gerador

```
1 type(função_geradora)
2 # function
3
4 type(função_geradora())
5 # generator
```

Geradores

Do que se alimentam?

Geradores

Um gerador é uma coisa que você pode iterar sobre ele. Com uma produção única (rever live de python #86)

```
1 gerador = função_geradora()
2
3 for valor in gerador:
4     print(valor)
5
6 # 1
```

Geradores

Para produzir o próximo item na iteração sem a necessidade de um laço for, podemos usar a função **next()**

```
1 gerador = função_geradora()
2
3 next(gerador)
4 # 1
```

Geradores

Quando não houverem mais valores para next, o python retornará uma exception **StopIteration**

```
1 gerador = função_geradora()
2
3 next(gerador)
4 # 1
5
6 next(gerador)
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9 StopIteration
```

Condição de parada

Toda vez que a função next chama o próximo valor do gerador ele produz (yield) o próximo valor

```python
def função_geradora():
    yield 1
    yield 2
    yield 3

gerador = função_geradora()

next(gerador)     # 1
next(gerador)     # 2
next(gerador)     # 3
next(gerador)     # StopIteration
```

Gerando valores de fato.

Produ zindo

Entendendo melhor as paradas

O python irá executar todo o código até a próxima aparição de yield

```
1 def gen_print():
2     print('Inicio da geração')
3     print('antes do 1')
4     yield 1
5     print('depois do 1')
6     print('antes do dois')
7     yield 2
8     print('depois do dois')
9     print('fim da função geradora')
```

Entendendo melhor as paradas

```
g = gen_print()

next(g)
Inicio da geração
antes do 1
# 1

next(g)
depois do 1
antes do dois
# 2

next(g)
depois do dois
fim da função geradora
# StopIteration:
```

```
def gen_print():
    print('Inicio da geração')
    print('antes do 1')
    yield 1
    print('depois do 1')
    print('antes do dois')
    yield 2
    print('depois do dois')
    print('fim da função geradora')
```

Me dá um exemplo prático?

Imagine que eu precise gerar uma sequência infinita de números ímpares. Como você faria?

```
1 def impares():
2     valor = 1
3     while True:
4         yield valor
5         valor += 2
```

Sub geradores

Geradores de geradores

Sub geradores

Sub geradores foram inseridos no python 2.3 com a sintaxe **yield from**.

O que podemos traduzir como ‘produza de’

```
1 def impares(r):
2     yield from (n for n in range(r) if n % 2 == 1)
```

gen expr

Entendendo as expressões geradoras

Expressões geradoras

Expressões geradoras são a forma mais sucinta de criar um gerador. Por exemplo, vamos usar nossa função de números primos como exemplo

```python
def impares():
    valor = 1
    while True:
        yield valor
        valor += 2
```

```python
# Números ímpares até 100
g = (n for n in range(100) if n % 2 == 1)

next(g)
# 1

next(g)
# 3
```

contexto

Gerenciando contextos com yield

Gerenciamento de contexto

Embora tenhamos uma live dedicada a esse tema (live de python #43) é legal expandir o uso do yield no caso de context managers

```
1 @contextmanager
2 def contador_de_tempo():
3     t0 = datetime.now()
4     yield
5     t1 = datetime.now()
6     delta = t1 - t0
7     print(delta.seconds)
```

Gerenciamento de contexto

```python
@contextmanager
def contador_de_tempo():
    t0 = datetime.now()
    yield
    t1 = datetime.now()
    delta = t1 - t0
    print(delta.seconds)
```

```python
with contador_de_tempo():
    print('oi')
    sleep(3)
```

Só pra não dizer que não falei das flores

Corrotinas

Corrotinas

Corrotinas são o estado da arte quando se fala no statement yield. Ela mantém uma forma de “comunicar” com um “gerador”.

Só daremos um leve exemplo, acredito que precisaríamos de uma live só sobre elas.

Corrotinas

```python
def coroutine():
    print('Comecei')
    while True:
        value = yield
        yield from (x*10 for x in value)
        print('Acabou a sequência')

c = coroutine()
next(c) # Comecei
c.send([1, 2, 3, 4])

next(c)   # 10
next(c)   # 20
next(c)   # 30
next(c)   # 40
next(c)   # Acabou a sequência
```