Entendendo o Itertools

Live de Python # 241

Roteiro

1. Iteradores
Uma revisão básica

2. Lazy Evaluation
Os benefícios dos geradores

3. Funções iteráveis
Iteradores embutidos

4. Itertools
Expandindo os embutidos

5. Infinito e Combinação
Mostra o brilho das funções.

6. Terminação e diminuição
Tornando iteraveis mais fáceis de trabalhar.

Essa live contém inúmeras referências a outras lives. Pois muitos contextos já foram explicados previamente!

Aviso 1

Por contar com referências, para grande parte da live demonstrarei coisas de forma empírica.

Aviso 2

picpay.me/dunossauro

apoia.se/livedepython

pix.dunossauro@gmail.com

Ajude o projeto <3

Ademar Peixoto, Adilson Herculano, Adriano Ferraz, Alemao, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alex Lima, Alynne Ferreira, Alysson Oliveira, Ana Carneiro, Andre Azevedo, Andre Mesquita, Aquiles Coutinho, Arnaldo Turque, Aslay Clevisson, Aurelio Costa, Bernardo At, Bernardo Fontes, Bruno Almeida, Bruno Barcellos, Bruno Barros, Bruno Freitas, Bruno Lopes, Bruno Ramos, Caio Nascimento, Christiano Morais, Damianth, Daniel Freitas, Daniel Wojcickoski, Danilo Boas, Danilo Segura, Danilo Silva, David Couto, David Kwast, Davi Goivinho, Davi Souza, Delton Porfiro, Denis Bernardo, Diego Farias, Diego Guimarães, Dilenon Delfino, Diogo Paschoal, Diogo Silva, Edgar, Eduardo Silveira, Eduardo Tolmasquim, Elias Silva, Emerson Rafael, Eneas Teles, Erick Andrade, Érico Andrei, Everton Silva, Fabiano Tomita, Fabio Barros, Fábio Barros, Fabio Castro, Fábio Thomaz, Fabricio Patrocinio, Felipe Rodrigues, Fernanda Prado, Fernando Celmer, Firehouse, Flávio Meira, Francisco Neto, Francisco Silvério, Gabriel Espindola, Gabriel Mizuno, Gabriel Paiva, Gabriel Simonetto, Geandreson Costa, Geizielder, Gilberto Abrao, Giovanna Teodoro, Giuliano Silva, Guilherme Felitti, Guilherme Gall, Guilherme Silva, Guionardo Furlan, Gustavo Pereira, Gustavo Suto, Harold Gautschi, Heitor Fernandes, Helvio Rezende, Hugo Cosme, Igor Riegel, Italo Silva, Janael Pinheiro, Jean Victor, Joelson Sartori, Jônatas Oliveira, Jônatas Silva, Jon Cardoso, Jorge Silva, José Gomes, Joseíto Júnior, Jose Mazolini, Juan Felipe, Juan Gutierrez, Juliana Machado, Julio Franco, Júlio Gazeta, Julio Silva, Kaio Peixoto, Kálita Lima, Kaneson Alves, Leandro Miranda, Leandro Silva, Leo Ivan, Leonardo Mello, Leonardo Nazareth, Leon Solon, Luancomputacao Roger, Lucas Adorno, Lucas Carderelli, Lucas Mendes, Lucas Nascimento, Lucas Schneider, Lucas Simon, Lucas Valino, Luciano Filho, Luciano Ratamero, Luciano Teixeira, Luis Alves, Luis Eduardo, Luiz Duarte, Luiz Lima, Luiz Paula, Luiz Perciliano, Mackilem Laan, Marcelo Campos, Marcio Moises, Marco Mello, Marcos Gomes, Maria Clara, Marina Passos, Mateus Lisboa, Mateus Ribeiro, Mateus Silva, Matheus Silva, Matheus Vian, Mauricio Nunes, Mírian Batista, Mlevi Lsantos, Murilo Viana, Nathan Branco, Nicolas Teodosio, Otávio Carneiro, Patricia Minamizawa, Patrick Felipe, Paulo Tadei, Pedro Henrique, Pedro Pereira, Peterson Santos, Priscila Santos, Pydocs Pro, Pytonyc, Rafael Lopes, Rafael Romão, Rafael Veloso, Raimundo Ramos, Ramayana Menezes, Regis Santos, Renato Oliveira, Rene Bastos, Ricardo Silva, Riverfount, Rjribeiro, Robson Maciel, Rodrigo Barretos, Rodrigo Freire, Rodrigo Oliveira, Rodrigo Quiles, Rodrigo Ribeiro, Rodrigo Vaccari, Rodrigo Vieira, Rogério Nogueira, Ronaldo Silveira, Rui Jr, Samanta Cicilia, Téo Calvo, Thaynara Pinto, Thiago Araujo, Thiago Borges, Thiago Curvelo, Thiago Souza, Tiago Minuzzi, Tony Dias, Tyrone Damasceno, Uadson Emile, Valcilon Silva, Valdir Tegon, Vcwild, Vicente Marcal, Vinicius Stein, Vladimir Lemos, Walter Reis, William Vitorino, Willian Lopes, Wilson Duarte, Wilson Neto, Zeca Figueiredo

Obrigado você

Iterad ores

Uma revisão sutil

Compreendendo Iteradores

Iterador é um objeto que pode ser **iterado** (ou seja, você pode "percorrer" o objeto, um elemento de cada vez).

No Python, um iterador é qualquer objeto que implementa o método `**__iter__**` (que deve retornar o próprio objeto) e o método `**__next__**` (que deve retornar o próximo valor do iterador).

Como os Iteradores funcionam

A melhor maneira de entender um iterador é ver um em ação.

```
1   my_list = [1, 2, 3, 4, 5]
2
3   # Agora, vamos criar um iterador a partir da lista
4   my_iterator = iter(my_list)
5
6   # Vamos imprimir os elementos do iterador
7   next(my_iterator)  # 1
8   next(my_iterator)  # 2
```

Sequências

Todos os tipos de sequência em python, são iteráveis:

- listas
- tuplas
- strings
- dicionários
- conjuntos
- bytes
- ...

```
1   a = [1, 2, 3]
2   for x in a:
3       print(x)
```

https://docs.python.org/3/library/collections.abc.html

Meu for [meu_for.py]

```python
a = [1, 2, 3]

it = iter(a)  # chama __iter__

while True:
    try:
        x = next(it)  # chama __next__
    except StopIteration:
        break
    print(x)
```

Paro por aqui!

https://www.youtube.com/live/Axbiz2q2iPA

https://www.youtube.com/live/Xj5LICeW0m0

Avaliação preguiçosa

Lazy ness

Avaliação preguiçosa

Uma estratégia de avaliação que **atrasa o cálculo de uma expressão até que seu valor seja realmente necessário**. Isso contrasta com a avaliação ansiosa (ou "eager evaluation"), onde as expressões são avaliadas assim que são declaradas.

```
>>> range(1_000)
range(0, 1000)
```

Geradores [count.py]

São um tipo de iterador que gera valores sob demanda. Quando você cria um gerador, o código não é executado imediatamente. Em vez disso, ele é executado cada vez que você solicita o próximo valor.

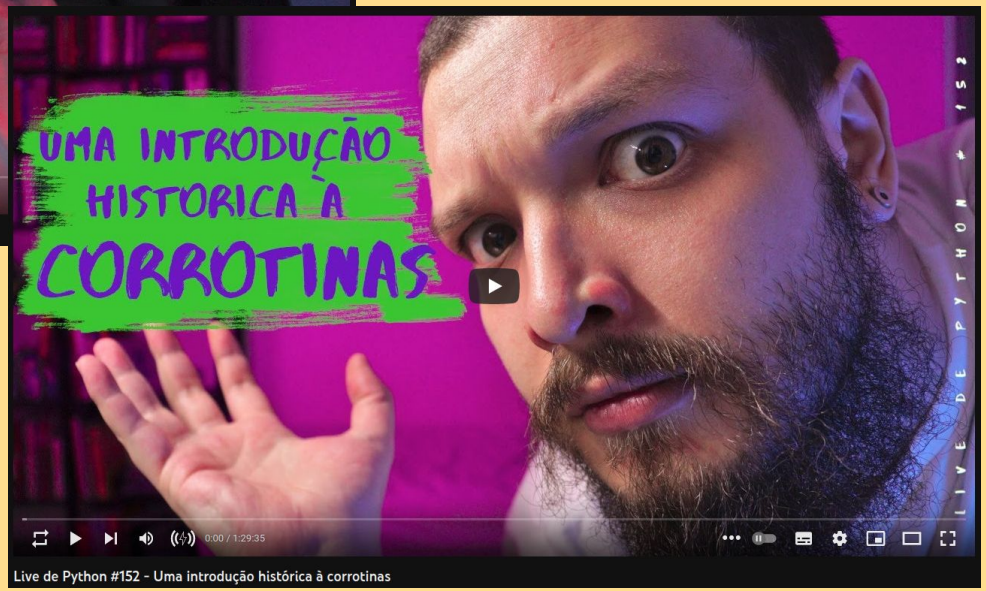
```python
def count(start, step=1):
    count = start
    while True:
        yield count
        count += step

contador = count(1)

print(next(contador))  # 1
print(next(contador))  # 2
print(next(contador))  # 3
```

Por que usar avaliação preguiçosa?

- Pode ser muito útil quando estamos lidando com grandes conjuntos de dados
  - Ao usar a avaliação preguiçosa, podemos reduzir o uso de memória, pois os valores são gerados sob demanda em vez de todos de uma vez.
- Permite que criemos estruturas de dados infinitas, já que os valores não precisam ser calculados até que sejam necessários.
- Pode nos ajudar a escrever código mais limpo e fácil de entender
  - Especialmente quando estamos encadeando várias operações juntas.

Chega por agora!

https://www.youtube.com/live/ZjwZ9nfhsk4



https://www.youtube.com/live/b6LT601s3_8

Sobre o consumo de memória

https://www.youtube.com/watch?v=cHraQ2l0Xg

Embutidos

Funções que são iteraveis e lazy

Funções embutidas

Existem diversas funções geradoras e iteráveis no namespace embutido.

```
>>> zip([1], [2])
<zip object at 0x7f4a7c59f240>

>>> range(1_000)
range(0, 1000)

>>> map(float, '123456')
<map object at 0x7f4a7c3f5bd0>

>>> reversed([1, 2, 3])
<list_reverseiterator object at 0x7f4a7c5df400>

>>> filter(None, [0, 0, 1])
<filter object at 0x7f4a7c3f4d60>
```

Aquela sensação estranha...

Usarei como exemplo uma das minhas funções preferidas **zip**

```
z = zip([1, 2, 3, 4], [3, 2, 1])
list(z)  # [(1, 3), (2, 2), (3, 1)]  cadê o 4?
list(z)  # []  só pode ser consumido uma vez
z = zip([1, 2, 3, 4], [3, 2, 1])
z[0]  # TypeError: 'zip' object is not subscriptable
```

De que algo está faltando

Agora usarei **map**

```
m = map(float, [1, 2, 3])
list(m)  # [1.0, 2.0, 3.0]

from operator import add
m = map(add, [1, 2, 3], [3, 2, 1])
list(m)  # [4, 4, 4]

m = map(add, zip([1, 2, 3], [3, 2, 1]))
list(m)  # TypeError: add expected 2 arguments, got 1
```

Saiu do caso previsto ...

Usando **filter** para ilustrar

```python
# Todos os verdadeiros
f = filter(None, [0, False, None, 1, 2, True])
list(f)  # [1, 2, True]

# E se quiser os falsos?
def not_bool(val): return not bool(val)  # ????
f = filter(not_bool, [0, False, None, 1, 2, True])
list(f)  # [0, False, None]
```

Melhor tacar compreensão em tudo

```python
# map
(float(x) for x in '1234567')

# filter
(x for x in [None, 0, False, 1, True] if bool(x))

# zip?      # range?
????        ????

# reversed
(x for x in [1, 2, 3, 4][::-1])
```

Caso func(func) tenha feito confusão mental

https://www.youtube.com/watch?v=IL9amfaaRws

Expandido horizontes

iter tools

Itertools

fornece **funções para criar iteradores para tarefas de looping eficientes e de alto desempenho**. Em palavras simples, ela é uma biblioteca que contém várias funções que retornam iteradores, que podem ser usados para percorrer elementos de forma eficiente.

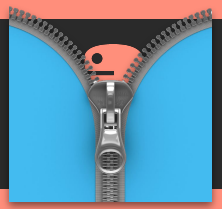
- **Eficiência de memória**: itertools manipula os iteradores de maneira "preguiçosa", o que significa que gera os elementos à medida que são necessários, em vez de gerar todos de uma vez e armazená-los na memória.
- **Eficiência de tempo**: itertools fornece funções de looping muito rápidas, o que pode economizar tempo de processamento.
- **Flexibilidade**: As funções em itertools podem ser combinadas de várias maneiras para atingir uma ampla gama de tarefas de looping.

Expandindo embutidos

Itertools conta com algumas ferramentas simples para resolver problemas codidianos, quando os embutidos deixam um pouco a desejar no caso padrão.

- **zip_longest**: Atribui valores padrões para zipar sequências de tamanhos diferentes
- **startmap**: Facilita a chamada de N iteráveis em uma única função
- **filterfalse**: Usa a mesma função do filter, para condições negativas
- **islice**: Adiciona as funções do slice, de forma preguiçosa, a iteradores

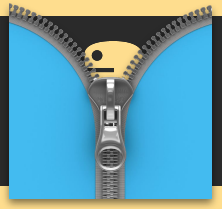
zip e zip_longest [zips.py]



Suponhamos que você precise calcular as médias das temperaturas dos estados sudeste nos últimos 5 dias.

```
from statistics import mean

ES = (22, 26, 18)
MG = (16, 20, 29, 18, 14)
RJ = (22, 17, 26, 19, 24)
SP = (31, 27, 29, 17, 19)

for dia in zip(ES, MG, RJ, SP):
  print(mean(dia))  # 22.75, 22.5, 25.5
```

zip e zip_longest [zips.py]



```
from statistics import mean
from itertools import zip_longest

ES = (22, 26, 18)
MG = (16, 20, 29, 18, 14)
RJ = (22, 17, 26, 19, 24)
SP = (31, 27, 29, 17, 19)

for dia in zip_longest(ES, MG, RJ, SP, fillvalue=None):
    print(mean(filter(None, dia)))
    # 22.75, 22.5, 25.5, 18 19
```

filter e filterfase [filters.py]

Geralmente, por filter ser uma hof, passamos uma função a ela. Só que se precisarmos inverter o filtro, temos que criar outra função.

```python
from itertools import filterfalse

ordens = [
    {"id": 1, "status": "pago"},
    {"id": 2, "status": "cancelado"},
    {"id": 3, "status": "pendente"},
    {"id": 4, "status": "pago"},
    {"id": 5, "status": "cancelado"},
]

def foi_pago(orden):
    return orden["status"] == "pago"

ordens_pagas = filter(foi_pago, ordens)
ordens_não_pagas = filterfalse(foi_pago, ordens)
```

starmap

Diferente do map, que aguada que os argumentos sejam passados de forma distribuída em vários iteráveis. O starmap desempacota o iterável

```python
from itertools import starmap
from operator import mul

pairs = [(2, 5), (3, 4), (8, 6), (7, 3)]

products = starmap(mul, pairs)
# 10, 12, 48, 21
```

islice

A função `itertools.islice()` que permite fazer uma "fatia" de um iterador, da mesma maneira que você faria uma fatia de uma sequência.

```
1   from itertools import islice
2
3   it = (x for x in [1, 2, 3, 4, 5])
4
5   next(
6       islice(it, 0)
7   )  # 1
```

Funções infinitas e combinatórias

+iter tools

Count

A função `itertools.count()` retorna um iterador que produz números consecutivos infinitamente. Aqui está um exemplo simples:

```python
from itertools import count

for i in count(0):
    print(i)
    if i >= 5:
        break
```

Essa função é útil quando você precisa de uma sequência infinita de números.

Exemplos de debug

```
>>> count(0, 10)
# 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, ...
>>> count(.1, .1)
# 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.10, ...
>>> count(0, -10)
# 0, -10, -20, -30, -40, -50, -60, -70, -80, -90, ...
```

Cycle

A função `itertools.cycle()` recebe um iterável e retorna um iterador que repete os elementos do iterável infinitamente. Aqui está um exemplo:

```python
from itertools import cycle

colors = cycle(['red', 'green', 'blue'])

for i, color in zip(range(7), colors):
    print(color)
    # 'red', 'green', 'blue', 'red', 'green', 'blue', 'red'
```

Exemplo para debug

```python
from itertools import cycle
atendimento = cycle(['Monica', 'Pedro', 'Ana', 'Sergio'])
pacientes = range(10)


for paciente in pacientes:
    print(f'{paciente=} - Atendimento {next(atendimento)}')

# paciente=0 - Atendimento Monica
# paciente=1 - Atendimento Pedro
# paciente=2 - Atendimento Ana
# paciente=3 - Atendimento Sergio
# paciente=4 - Atendimento Monica
# paciente=5 - Atendimento Pedro
# paciente=6 - Atendimento Ana
```

repeat

A função `itertools.repeat()` **retorna um iterador que produz o mesmo valor repetidamente**. A menos que você especifique um número de vezes para repetir o valor, a função `repeat()` irá produzi-lo infinitamente.

```python
from itertools import repeat

for i in repeat('Python', 5):
    print(i)
# Python Python Python Python Python
```

Exemplos para debug

```
>>> list(map(pow, range(0, 10, 2), repeat(2)))
[0, 4, 16, 36, 64]

>>> list(map(pow, repeat(2), range(12)))
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048]
```

Exemplos de combinações

O itertools fornece 4 funções para trabalhar com combinações:

- **product**: Quando precisamos evitar for dentro de for
- **permutation**: Produz todas as permutações entre N valores em tuplas
- **combinations**: Produz as combinações
- **combinations_with_replacement**: Combinações com repetição

Combinadores

```python
from itertools import product, permutations, combinations

letters = ['a', 'b']
numbers = [1, 2]

for letter, number in product(letters, numbers):
    print(f'{letter}{number}')  # 'a1', 'a2', 'b1', 'b2'.

for perm in permutations('ABC', 2):
    print(''.join(perm))  # 'AB', 'AC', 'BA', 'BC', 'CA', 'CB'

for combo in combinations('ABC', 2):
    print(''.join(combo))  # 'AB', 'AC', 'BC'
```

Iter tools

Terminação e diminuição

Funções que terminam (não infinitas)

Diferente dos iteradores infinitos ou combinatórios. A ideia de funções que terminam, é estabelecer um limite para os iteradores. Como:

- **accumulate**: itera em algo acumulando seus valores
- **takewhile**: Retorna os valores de um iterável enquanto forem verdadeidos
- **dropwhile**: Ignora valores de um iterável enquanto forem falsos

Acummulate

A função `itertools.accumulate()` **retorna um iterador que acumula valores**. Por padrão, acumula somas. Porém, outras funções podem ser passadas para alterar o comportamento de acumulação.

```python
from itertools import accumulate
import operator

data = [1, 2, 3, 4, 5]
result = accumulate(data, operator.mul)
for each in result:
    print(each)  # '1', '2', '6', '24', '120'
```

Um caso de uso

```python
from itertools import accumulate

# Lista de transações bancárias (depósitos e retiradas)
transactions = [200, -100, 300, -200, 100, -50, 300, -150, 200]

# Use accumulate para calcular o saldo após cada transação
balances = list(accumulate(transactions))

# Imprime todas as transações e o saldo após cada uma
for i, balance in enumerate(balances):
    print(f"Após a transação {i+1} de {transactions[i]:+}, o saldo é {balance}")
```

Take e Drop while

São funções que percorrem um iterável enquanto uma condição for verdadeira ou falsa:

```python
from itertools import takewhile, dropwhile

numbers = [1, 2, 3, 4, 5, 6]
result = takewhile(lambda x: x < 5, numbers)
for number in result:
    print(number)  # '1', '2', '3', '4'

result = dropwhile(lambda x: x<5, numbers)
for number in result:
    print(number)  # '5', '6'
```

Um caso de uso

Leitura de arquivo

```
1   from itertools import takewhile, dropwhile
2
3   for line in takewhile(  # um arquivo que termina com comentários
4       lambda x: not x.startswith('#'), open('meu_arquivo.txt', 'r')
5   ):
6        print(line)
7
8   for line in dropwhile(  # ignora os comentários no início do arquivo
9       lambda x: x.startswith('#'), open('meu_arquivo.txt', 'r')
10  ):
11       print(line)
```

Diminuição

Categorizei como dinuição, todas as funções que tornam um iterável mais simples

- **chain**: Junta N sequências para formarem uma única
- **compress**: Filtra valores de um interável usando outro iterável como base
- **groupby**:

Chain

aceita um ou mais iteráveis e retorna um iterador que efetivamente **concatena os iteráveis**

```python
from itertools import chain

letters = ['a', 'b', 'c']
numbers = [1, 2, 3]

for item in chain(letters, numbers):
    print(item)  # 'a', 'b', 'c', '1', '2', '3'
```

Compress

filtra um iterável usando outro iterável de booleanos como uma e**spécie de máscara**

```
1   from itertools import compress
2
3   data = ['a', 'b', 'c', 'd', 'e']
4   selectors = [True, False, True, False, True]
5
6   for item in compress(data, selectors):
7       print(item)  # 'a', 'c', 'e'
```

groupby

Agrupa valores em iteradores de iteradores.

```
1   from itertools import groupby
2   from operator import itemgetter
3
4   data = [
5     ('apple', 'fruit'), ('carrot', 'vegetable'),
6     ('pear', 'fruit'), ('broccoli', 'vegetable')
7   ]
8
9   # é necessário ordenar os dados primeiro
10  data.sort(key=itemgetter(1))
11
12  for key, group in groupby(data, itemgetter(1)):
13      print(key, list(group))
```

Algumas menções que podem ajudar

Além da biblioteca padrão, existem outras coleções de objetos para lidar com iteráveis!

- more-itertools: https://github.com/more-itertools/more-itertools
- toolz: https://github.com/pytoolz/toolz/
- aioitertools: https://github.com/omnilib/aioitertools

picpay.me/dunossauro

apoia.se/livedepython

pix.dunossauro@gmail.com

Ajude o projeto <3