Iteração: A história secreta - Naomi Ceder

Usar laços "for" e "list comprehensions é bastante comum e até básico em Python. Mas como a iteração realmente funciona em Python? As palavras "iterador" e "iterável" aparecem mais de quinhentas vezes na documentação do Python, mas o que um "iterador" realmente faz em comparação com um "iterável"? E como eles fazem isso? Nesta palestra você aprenderá os detalhes da história secreta da iteração, com um pouco de codificação ao vivo.

Iteração: A história secreta

De dentro do protocolo de iteração

Python Brasil 2020

Naomi Ceder

Este jupyter notebook está disponível em [https://github.com/nceder/workshops (https://github.com/nceder/workshops)](https://github.com/nceder/workshops)

Olá, sou Naomi e hoje vou falar um pouco sobre o processo de iteração em Python.

Em primeiro lugar, estou muito honrada em apresentar esta palestra. Há apenas 4 anos, fiz uma palestra principal na Python Brasil 12 - foi a primeira vez que visitei a América Latina e sempre penso nisso como o início de conhecer tantas pessoas incríveis em tantos países.

Pra mim esta palestra é muito especial por outro motivo - é a minha primeira em português.

Obrigada e espero que vocês gostem (e entendam) desta palestra.

A propósito, este notebook está disponível no githu neste link...

Iteração = a repetição com código e dados

Então, todos nós sabemos (ou pensamos que sabemos) o que é iteração: repetir código com dados ...

Mas embora o uso de iteração seja muito comum, diria que a história de seu funcionamento não é bem conhecida.

Na verdade, Dave Beazley, um grande homem do Python (e um colega aqui em Chicago) chamou a iteração de "Python's most powerful useful feature", ou seja, um recurso poderoso que todos nós usamos todos os dias.

Eu recomendo fortemente que você assista ao vídeo de sua palestra remota na PyCon Paquistão; conta a história do protocolo de iteração dos primeiros dias do Python ... e toca a música do Monty Python em seu trombone.

“Python's most powerful useful feature” - “o recurso útil mais poderoso do Python"”

-- Dave Beazley, “Iterations of Evolution: The Unauthorized Biography of the For-Loop (https://www.youtube.com/watch?v=2AXuhgid7E4)”

Coleções / séries repetitivas de dados estão em toda parte

por exemplo:

- temperaturas de um mês
- chaves para um dicionário de números de identificação de membro: informações do membro
- um arquivo CSV de um milhão de produtos
- o texto do romance *Dom Casmurro*
- os resultados de uma consulta de banco de dados para as vendas de ontem

E, claro, existem muitas coleções de dados em todos os lugares ... podem ser...

Elas têm pouco em comum

- diferentes tipos de séries
- diferentes tipos de elementos

Mas...

para todas, a mesma estrutura pode ser usada para iteração.

O laço `for` Pythonico

Ok ... vamos ver um laço `for` . É claro que laços `for` são usados em quase todas as linguagens de programação. Mas, como eu disse antes, nem todos funcionam da mesma maneira e Python foi um dos primeiros a iterar em uma sequência. Hoje, é claro, vários idiomas possuem esse recurso.

Mas deixe-me escrever um pequeno laço...

```
for elemento in uma_lista:
    print(elemento)
```

Isso funciona conforme o esperado e conta de um a quatro.

In [ ]:
```
# laço for (estilo Python)
uma_lista = [1, 2, 3, 4]
```

Pergunta: nesse laço simples, quantas exceções são lançadas?

- 0
- 1
- 4
- 2

Mas... tenho uma pergunta - nesse laço, quantas exeções são lançadas? Sabe?

A certa resposta é 2 ...

Você teve a resposta correta? Logo veremos por quê.

A certa resposta é 2 ...

Você teve a resposta correta? Logo veremos por quê.

Em 1994 esse laço foi surpreendente

Python and `for` loops

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or leaving the user completely free in the iteration test and step (as C), Python's `for` statement iterates over the items of any sequence (e.g., a list or a string), in the order that they appear in the sequence.

-- Python V 1.1 Docs, 1994

Não vou traduzir este parágrafo; É da documentação do Python 1.1, 25 anos atrás.

O importante é que este texto assume que os leitores estavam esperando uma linguagem em que um laço `for` fosse apenas uma sequência de inteiros, como Pascal (em 1994 eu estava ensinando Pascal para alunos do ensino médio, então me lembro bem disso) ou uma linguagem na qual não há regras sobre os componentes da instrução `for` como C.

(Você sabe que a instrução `for` em C requer? Apenas parênteses e dois pontos e virgulas, nada mais.)

Por outro lado, Python era um pouco estranho porque a instrução `for` operava nos membros de uma sequência.

Como é que isso funciona?

- **Como um laço `for` sabe qual elemento é o "próximo"?**
- **Como os laços `for` podem usar tantos tipos diferentes?**
- **O que faz um objeto "funcionar" num laço `for` ?**

protocolo de iteração

- iteração em Python depende de um **protocolo**, não de tipos (desde Python 2.2)
- é um bom exemplo de "duck typing" em Python - qualquer objeto que siga o protocolo pode ser iterado.

A iteração em Python funciona com tantos tipos porque é um protocolo ... ou seja, é uma maneira especificada de iterar em sequências ou coisas que se comportam como sequências. O protocolo especifica como o seguinte é retornado e quais métodos um objeto precisa para iterar.

Este é um conceito muito importante em Python, a ideia de que a forma como um objeto funciona é mais importante do que seu tipo. Em Python, às vezes chamamos isso de "mais fácil pedir desculpas do que permissão" - não perguntamos se um objeto é do tipo correto ... apenas tentamos e vemos se faz o que queremos.

O protocolo:

1. para iteração precisa de um objeto **iterável**
2. e um **iterador** (que Python geralmente cria para você)

O protocolo tem duas partes...

Esses termos são mencionados mais de 500 vezes na documentação oficial do Python. Eles são muito semelhantes e fáceis de confundir, mas devemos entender como cada um funciona para entender a iteração em Python.

O que é um ‘iterável’?

- um objeto que pode retornar seus elementos **um após o outro**
- um objeto que retorna um **iterador** se a função `iter()` é chamada com ele.

ou seja

- **qualquier classe** com um método **`__iter__()`**
- ou **qualqier classe** com um método **`__getitem__()`** que implementa **semântica de sequência**.

Os iteráveis incluem **todos os tipos de sequência** (como `list` , `str` , e `tuple`) e **alguns tipos não sequência** como `dict` , arquivos, etc.

A propósito: os sublinhados duplos são pronunciados "dunder" em inglês (abreviação de "double underscore")

Os iteráveis podem ser criados de **duas** maneiras. Se ele contém um método `__iter__` que retorna um iterador, é (como veremos) um iterável. Mas se não contiver um método `__iter__` ele ainda pode ser iterável si contiver um método `__getitem__` com semântica de sequência. E muitos objetos, como listas e strings, contém ambos.

¿Semântica de sequência?

- um objeto acessa seus valores ou elementos em ordem
- acessa seus elementos com colchetes `[]` e índices de inteiros começando em 0
- lança uma execeção IndexError quando o índice está mais além do final da sequência

```
x[0]
x[1]
x[2]
```

Mas, o que é semântica de sequência? É quando um objeto...

Neste exemplo a lista `x` contém três elementos que podemos acessar com os índices 0, 1, e 2. Se tentarmos acessar a posição 3 (ou o quarto elemento) uma execeção IndexError é lançada, porque esse elemento não existe no objeto.

Vamos fazer um iterável - `Repeater`

Um objeto que pode ser iterado e retorna o mesmo valor o número especificado de vezes.

```
repeat = Repeater("olá", 4)

for i in repeat:
    print(i)

olá
olá
olá
olá
```

Vamos ver como um iterável funciona na prática... Vou criar uma classe que é um iterável o mais simples possível. Será muito simples, retornando o mesmo valor um determinado número de vezes.

Se criamos uma instância de Repeat com o valor "olá" e o número 4, esse iterável vai retornar "olá" 4 vezes.

Um iterável com `__getitem__()`

Porque não tem método `__iter__()`, temos que implementar a classe con `__getitem__`.

Vou fazer assim...

```
        def __getitem__(self, indice):
            if 0 <= indice < self.límite:
                return self.valor
            else:
                raise IndexError
```

se `0 <= indice < self.límite` retornamos o valor... senão... lançamos uma exceção IndexError

In [ ]:
```
class Repeater:
    def __init__(self, valor, limite):
        self.valor = valor
        self.limite = limite

    def __getitem__(self, indice):
```

In [ ]:
```
repeat = Repeater("olá", 4)
```

Vamos criar uma instância desta classe... com o valor 'olá' e 4 como o número de repetições.

Vamos tentar a semântica da sequência ... quais índices funcionam? e quais lançam exceções?

e que retorna `iter()` ?

In [ ]:
```
# __getitem__ com semântica de sequência?

repeat[1]
```

In [ ]:
```
# retorna a função iter() um iterador?

iter(repeat)
```

In [ ]:
```
# funciona num laço?
for elemento in repeat:
    print(elemento)
```

Funciona num laço?

Funciona numa lista de compreensão?

In [ ]:
```
# numa lista de compreensão?

[x for x in repeat]
```

O que realmente aconteceu

- o objeto `repeat` é um **iterável**
  - pode retornar seus elementos com índices inteiros começando em 0
  - continua até que uma exceção IndexError seja gerada
- um **iterador** é criado do objeto `repeat` pela instrução `for`
  - o iterador obtém cada elemento do iterável
  - o iterador captura a exceção IndexError de iterável e para a iteração
- toda vez que um novo processo de iteração é iniciado, um novo iterador é criado e a iteração começa do início

Observação: Nosso objeto `repeat` lançou uma exceção IndexError ... como quase sempre em laços como este

No caso da instrução `for` , ele realmente chama `iter()` para obter um iterador, e mantém o iterador em uma variável anônima.

O que é um iterador?

- O laço `for` em Python depende de ser capaz de obter o elemento **next** (próximo)
- **o próprio laço não sabe e não se importa** exatamente onde na série um elemento está (ou que tipo ele é)
- **o próprio iterável não sabe** qual elemento é o **next**
- é o **iterador** que sabe qual elemento é o **next**

Um **iterador** tem um método `__next__()` que retorna o próximo elemento na sequência.

Mas o iterável é só uma parte do protocolo de iteração... a outra parte, igualmente importante, mesmo que muitas vezes seja criado automaticamente, é o iterador. E que é um iterador?

Um iterador

- representa **um fluxo de dados**.
- cada **chamada do método `__next__()` do iterador** (ou a função `next()`) **retornará o próximo elemento** no fluxo.
- quando **não há mais elementos, uma exceção StopIteration** é lançada.
- depois o **iterador está esgotado** e se você chamar o método `__next__()` novamente, a exceção StopIteration é lançada...

Em outras palavras um iterador...

Todos os iteradores

- deve ter um método `__iter__()` que retorna o próprio iterador
  - portanto, **todos os iteradores também são iteráveis**
  - todos os iteradores podem ser usados em locais onde outros iteráveis são aceitos.

Lembre-se que um objeto pode ser um iterável se tiver um método `__getitem__` com sequência de semântica **ou** se tiver um método `__iter__` que retorna um iterador... e todos os iteradores...

Múltiplas Iteraçôes

- A maioria dos iteráveis (por exemplo, listas) produzem **um novo iterador por vez**.
- **um iterador sempre retornará o mesmo iterador esgotado** da iteração anterior, como um contêiner vazio.

Uma diferença importante entre um iterável e um iterador é como funcionam com múltiplas iterações...

Vamos fazer um iterador - `RepeatIterator`

Funcionará quase da mesma maneira, mas vamos implementá-lo como um iterador

- método `__next__()` para retornar o próximo elemento
- método `__iter__()` para retornar o próprio iterador

Então... agora vamos criar um iterador

como iterador, deve ter um método `__next__` e um método `__iter__` ...

In [ ]:
```
class RepeatIterator:
    def __init__(self, valor, limite):
        self.valor = valor
        self.limite = limite

    def __next__(self):

    def __iter__(self):
```

```
class RepeatIterator:
    def __init__(self, valor, limite):
        self.valor = valor
        self.limite = limite
        self.conta = 0
```

```
        def __next__(self):
            if self.conta < self.limite:
                self.conta += 1
                return self.valor
            else:
                raise StopIteration

        def __iter__(self):
            return self
```

Adicionamos uma variável de instância "conta" para que o objeto saiba onde está na sequência.

E, neste caso, após retornar o último item, o objeto não lança uma exceção IndexError (porque não é acessado por índice), mas sim uma exceção StopIteration.

E, claro, existe um método `__iter__()` que retorna o próprio objeto.

In [ ]:
```
# é criada uma instância de RepeatIterator

repeat_iter = RepeatIterator("Olá", 4)

repeat_iter
```

Bom, temos uma instância de RepeatIterator. Um iterador deve retornar o próximo valor com a função `next()` ...

e nossa instância faz isso

In [ ]:
```
# a função next() retorna o próximo elemento?
next(repeat_iter)
```

In [ ]:
```
# quando iter() e chamada com ela,
# retorna o objeto mesmo?
print(repeat_iter)

repeat_iter_iter = iter(repeat_iter)
print(repeat_iter_iter)
```

a próxima pergunta é quando `iter()` é chamada com ela, retorna o objeto mesmo?

Sim...

Por outro lado quando chamamos `iter()` com nosso iterável, retorna um objeto diferente a cada vez.

In [ ]:
```
# chamar iter() com iterável
# sempre retorna una iterador novo
print(iter(repeat))

print(iter(repeat))
```

In [ ]:
```
# você pode usá-lo em um laço for?
for elemento in repeat_iter:
    print(elemento)
```

Finalmente, podemos usá-lo em um laço?

Claro, funciona perfeitamente... mas sabe por que tem só três valores?

é por que a chamada de `next()` acima usou o primeiro valor...

In [ ]:
```
# pode ser reutilizado? (com next())
next(repeat_iter)
```

Mas... se o iterador é esgotado y usarmos `next()` novamente... o que acontece?

Lança uma exceção StopIteration... o que esperaríamos

In [ ]:
```
# podemos usar-lo novamente num laço for?
for elemento in repeat_iter:
    print(elemento)
```

y se usarmos o iterador esgotado num laço?

Nada... exatamente como se o objeto estivesse vazio.

O que realmente aconteceu

- o `for` chamou `iter()` com `repeat_iter` , que retornou si mesmo (em uma variável anônima)
- o `for` chamou `next()` com o iterador para obter os valores do laço
- o `for` pegou `StopIteration` e parou a iteração

o que realmente aconteceu aqui é... Vimos como funcionam os iteráveis e iteradores ... precisamos de ambos, mas normalmente vemos apenas os iteráveis.

É por isso que havia duas exceções na minha pergunta anterior ...

1. uma exceção IndexError é lançada pelo iterável (e capturada pelo iterador)
2. uma exceção StopIteration é lançada pelo iterador (e capturada pelo laço `for`)

(se usarmos um iterador como iterável, será apenas uma exceção)

É por isso que havia duas exceções na minha pergunta anterior ...

uma exceção IndexError é lançada pelo iterável (e capturada pelo iterador)

e, por sua vez uma exceção StopIteration é lançada pelo iterador (e capturada pelo laço for) que para a iteração.

A diferença entre um iterável e um iterador é importante

- para iterar sobre uma sequência, um iterável é usado e um novo iterador é criado a cada vez ...
- para iterar sobre um arquivo ou sobre os resultados de um banco de dados, e tal, um iterador é usado e esse iterador é **o mesmo objeto** todas as vezes (a menos que você obtenha explicitamente um novo) e **o iterador esgotado parece vazio.**

Deve-se enfatizar que há uma diferença básica entre iteráveis e iteradores - um novo iterador é criado cada vez que um iterável é usado, mas um iterador pode ser usado como iterável apenas uma vez.

Finalmente, depois de ver como criar um iterador como classe, devo dizer que existe uma maneira mais simples e comum de criar um iterador.

Criar um iterador com uma função geradora

- usa `yield` em vez de `return`
- a função retorna um objeto, que é um generador
- os generadores são iteradores

Uma função geradora é um tipo especial de função... a instrução "return" não é usada dentro de um gerador, mas `yield` . Se uma função contém 'yield', ela é um gerador, e quando o código é executado, um iterador é criado

```
conta = 0
while conta < limite:
    yield valor
    conta += 1
```

In [ ]:
```
def repeat_gen(valor, limite):


for elemento in repeat_gen("olá", 4):
    print(elemento)
```

O que realmente aconteceu

- `repeat_gen("olá", 4)` **retornou um objeto gerador que também é um iterador**
- `for` chamou `iter()` com o gerador anônimo
- `for` chamou `next()` com este gerador para obter os valores do laço (os valores foram retornados por `yield`)

- o gerador retornou `StopIteration` ao terminar
- `for` capturou a `StopIteration` e interrompeu a iteração

Observação sobre funções geradoras

Mesmo que a função esteja no laço for, é o objeto gerador retornado pela função que é o iterador

Neste exemplo o processo é explicito y podemos ver que não é a função, mas o gerador que é o iterador.

In [ ]:
```
def repeat_gen(valor, limite):
    conta = 0
    while conta < limite:
        yield valor
        conta += 1

gen_iterator = repeat_gen("olá", 4)

for elemento in gen_iterator:
    print(elemento)
```

In [ ]:
```
repeat_gen
```

Aqui podemos ver a diferença entre a função que cria o gerador e o própio gerador, que é o objeto que funciona como um iterador

Como eu disse, e mais comum e mais Pythonico usar funções geradoras para criar iteradores.

In [ ]:
```
gen_iterator
```

In [ ]:
```
iter(gen_iterator)
```

Iteração em Python

- é um **protocolo** (desde Python 2.2)
- precisa de um **iterável**
- também precisa de um **iterador** (muitas vezes anônimo y criado automaticamente) para obter o **next**
- **iteradores podem ser usados como iteráveis,** mas não são "atualizados"
- geradores também são iteradores e são comumente usados em vez de criar uma classe

Bom... para resumir... a iteração em Python...

- é um **protocolo** (desde Python 2.2)
- precisa de um **iterável**

- também precisa de um **iterador** (muitas vezes anônimo y criado automaticamente) para obter o **next**
- **iteradores podem ser usados como iteráveis,** mas não são "atualizados"
- geradores também são iteradores e são comumente usados em vez de criar uma classe

É isso ... Espero que agora você tenha um melhor entendimento do que realmente acontece na iteração em Python e que você nunca se deixe enganar pela diferença entre um iterável e um iterador.

Muito obrigada! Perguntas?

@NaomiCeder • naomi@naomiceder.tech (mailto:naomi@naomiceder.tech) • www.naomiceder.tech (http://www.naomiceder.tech)

Este jupyter notebook está disponível em https://github.com/nceder/workshops (https://github.com/nceder/workshops)

Recursos para iteração

- O Tutorial Python - https://docs.python.org/pt-br/3.8/tutorial/index.html (https://docs.python.org/pt-br/3.8/tutorial/index.html)
  - Comandos `for` (https://docs.python.org/pt-br/3.8/tutorial/controlflow.html#for-statements)
  - Iteradores (https://docs.python.org/pt-br/3.8/tutorial/classes.html#iterators)
  - Geradores (https://docs.python.org/pt-br/3.8/tutorial/classes.html#generators) & Expressões geradoras (https://docs.python.org/pt-br/3.8/tutorial/classes.html#generator-expressions)
- Tipos de iteradores documentação (https://docs.python.org/pt-br/dev/library/stdtypes.html#iterator-types)
- Iteradores, Programação Funcionado COMOFAZER (https://docs.python.org/pt-br/dev/howto/functional.html#iterators)
- Iterations of Evolution: The Unauthorized Biography of the For-Loop (https://www.youtube.com/watch?v=2AXuhgid7E4) - Dave Beazley, PyCon Pakistan 2017

Se você tiver alguma pergunta, eu adoraria respondê-la...