Compreensão de listas e expressões geradoras

Muito do que estudamos até o momento em Python pode ser reproduzido de maneira parecida em outras linguagens. Comandos como `if`, `else`, `while` e `for`, bem como conceitos como criar funções, passar parâmetros e retornar valores são comuns a uma infinidade de linguagens de programação.

Porém, um dos objetivos da linguagem Python é realizar o máximo possível de trabalho com a menor quantidade possível de código, resultando em um código mais limpo e com menos efeitos colaterais.

Com isso, o Python traz maneiras diferentes e mais enxutas de resolver problemas que já lidávamos bem utilizando outras técnicas. Parte dessas técnicas foi inspirada em conceitos de **programação funcional**, que será explicada um pouco melhor em um capítulo futuro.

As **compreensões de listas** e **expressões geradoras** são algumas dessas ferramentas.

1. Compreensão de listas

1.1. Compreensão de listas contendo apenas um loop

Vamos considerar um problema simples: montar uma lista com os quadrados dos números de 1 até 10. Você provavelmente resolveria esse problema da seguinte maneira:

```
quadrados = []

for x in range(1, 11):
    quadrados.append(x**2)

print(quadrados)
print(x)
```

Note que utilizamos 3 linhas de código para criar uma lista: uma para declarar a lista, uma para percorrer alguns valores e uma para executar o cálculo e adicionar o resultado à lista.

Além disso, criamos uma variável extra, `x`, que segue existindo em nosso programa mesmo após o loop, como você pode observar pelo segundo `print` do exemplo.

A **compreensão de listas** resolve todos esses problemas: iremos resumir em uma única linha a criação da nova lista já com todos os valores desejados, e sem variáveis *sobrando* após a execução:

```
quadrados_compreensao = [num**2 for num in range(1, 11)]

print(quadrados_compreensao)
```

Note que a variável `x`, criada na versão extensa, ainda existe. A variável `num`, utilizada na compreensão, não:

```
print('x:', x) # imprime 10
print('num:', num) # erro de variável não existente
```

Vale destacar que você não precisa necessariamente utilizar `range` em suas compreensões. Você pode utilizar **qualquer** tipo de iterável, como listas, tuplas, strings etc.

O exemplo abaixo monta uma lista contendo a metade do valor de cada elemento de uma outra lista:

```
numeros = [1, 9, 4, 7, 6, 2]
```

```
metades = [n/2 for n in numeros]

print(metades)
```

1.2. Condicionais em compreensões

Podemos utilizar compreensões em nossas condicionais. Imagine que não fossem aceitos valores "quebrados" no exemplo anterior. Logo, não podemos dividir os números ímpares, apenas os pares. Fazemos isso usando um `if` na expressão:

```
metades_pares = [n/2 for n in numeros if n % 2 == 0]
print(metades_pares)
```

Podemos também utilizar `else` na expressão. Vejamos mais um exemplo:

Considere que são aceitos números quebrados no exemplo das metades. Porém, não queremos utilizar o tipo `float` desnecessariamente. Portanto, faremos uma divisão **inteira** quando o número for par (para que o resultado seja int) e uma divisão **real** quando o número for ímpar (para que o resultado seja float). A expressão ficaria assim:

```
metades_tipo = [n//2 if n % 2 == 0 else n/2 for n in numeros]

print(metades_tipo)
```

Note que quando usamos o `else`, a ordem da compreensão sofre uma alteração. Quando usamos apenas o `if`, ele vem após o `for`. Com o `else`, ambos vêm antes.

Outro ponto importante é que no caso do `else` passamos a ter uma segunda expressão. Quando a condição do `if` é verdadeira, a compreensão irá executar a expressão original. Caso contrário, ela irá executar a expressão do `else`.

Generalizando a sintaxe das compreensões de lista, temos as seguintes combinações:

```
lista = [expressao for item in colecao]

# equivale a:

for item in colecao:
    lista.append(expressao)
```

```
[expressao for item in colecao if condicao]

# equivale a:

for item in colecao:
    if condicao:
        lista.append(expressao)
```

```
[expressao if condicao else expressao_alternativa for item in colecao]

# equivale a:

for item in colecao:
    if condicao:
        lista.append(expressao)
```

```
    else:
        lista.append(expressao_alternativa)
```

1.3. Aninhando compreensões

É possível aninhar compreensões de lista. Ao colocarmos mais de um `for` consecutivo, o primeiro for será considerado o mais externo, e o seguinte, mais interno. O exemplo abaixo mostra todas as combinações possíveis entre alguns nomes e sobrenomes:

```
nomes = ['Ana', 'Bruno', 'Carla', 'Daniel', 'Emília']
sobrenomes = ['Silva', 'Oliveira']

combinacoes = [nome + ' ' + sobrenome for nome in nomes for sobrenome in sobrenomes]
print(combinacoes)
```

A linha `combinacoes = [nome + ' ' + sobrenome for nome in nomes for sobrenome in sobrenomes]` equivale a:

```
combinacoes = []

for nome in nomes:
    for sobrenome in sobrenomes:
        combinacoes.append(nome + ' ' + sobrenome)
```

Inclusive podemos utilizar essa forma para trabalhar com matrizes. O exemplo abaixo lê pelo teclado a quantidade de vitórias, empates e derrotas para cada time em um grupo:

```
times = ['Atlético Python', 'JavaScript United', 'C Seniors', 'Javeiros do Norte']
entradas = ['V', 'E', 'D']

tabela = [[int(input(f'Digite a quantidade de {tipo} do time {time}: ')) for tipo in entradas] for time in times]

print(tabela)
```

1.4. Compreensão de dicionários

Da mesma forma que utilizamos compreensão para listas, podemos utilizá-la para dicionários. A diferença é que precisamos, obrigatoriamente, passar um par chave-valor. O exemplo abaixo parte de uma lista de notas e uma lista de alunos e chega em um dicionário associando cada aluno a uma nota.

```
alunos = ['Ana', 'Bruno', 'Carla', 'Daniel', 'Emília']
medias = [9.0, 8.0, 8.0, 6.5, 7.0]

cadastro = {alunos[i]:medias[i] for i in range(len(alunos))}

print(cadastro)
```

Saída na tela:

```
{'Ana': 9.0, 'Bruno': 8.0, 'Carla': 8.0, 'Daniel': 6.5, 'Emília': 7.0}
```

Talvez você não tenha achado esse código tão *pythonico*, e você tem razão. Não gostamos de percorrer listas por índices dessa maneira. Uma estratégia melhor é utilizar o `zip`, estudado em capítulos anteriores:

```
alunos = ['Ana', 'Bruno', 'Carla', 'Daniel', 'Emília']
medias = [9.0, 8.0, 8.0, 6.5, 7.0]

cadastro = {aluno:media for aluno, media in zip(alunos, medias)}
```

```
print(cadastro)
```

Observação: se você tentou fazer uma compreensão de dicionário e esqueceu de utilizar um par chave-valor, talvez você tenha se surpreendido ao notar que não gerou um erro. Isso ocorre porque existe *outra* estrutura de dados em Python que não estudamos no curso que utiliza os símbolos `{` e `}`: o `set` (conjunto). Ele é uma coleção **mutável** de elementos (como a lista), mas ele não possui índice (porque a ordem não importa) e ele não aceita elementos repetidos. Caso tenha curiosidade, segue material de referência com o básico de como trabalhar com conjuntos: https://www.programiz.com/python-programming/set

2. Expressões geradoras

2.1. Criando uma expressão geradora

Se você executar o código abaixo, não notará erros de execução. Ambas as linhas executam com sucesso:

```
colchetes = [x for x in range(10)]

parenteses = (x for x in range(10))
```

Listas são representadas por colchetes, e fazemos compreensão de listas utilizando colchetes. Dicionários utilizam chaves (`{` e `}`), e utilizamos chaves para fazer compreensão de dicionários. A expressão entre parênteses só pode ser uma tupla, certo?

Errado. Não existe compreensão de tuplas em Python. Quando colocamos uma expressão semelhante a uma compreensão de lista entre parênteses, estamos criando uma expressão geradora. Note que podemos iterar uma expressão geradora:

```
gerador_quadrados = (x**2 for x in range(10))

for quadrado in gerador_quadrados:
    print(quadrado)
```

Também podemos convertê-la para outras estruturas, como uma lista ou uma tupla:

```
gerador_impares = (x for x in range(20) if x % 2 == 1)

lista_impares = list(gerador_impares)

print(lista_impares)
```

Porém, não podemos utilizar nosso gerador uma segunda vez. Execute o código abaixo:

```
gerador_quadrados = (x**2 for x in range(10))

for quadrado in gerador_quadrados:
    print(quadrado)

lista_quadrados = list(gerador_quadrados)
print(lista_quadrados) # imprime uma lista vazia
```

Para entender porque o resultado foi uma lista vazia, precisamos entender a diferença entre um **iterável** e um **iterador** em Python.

2.2 Iteráveis e iteradores

Um **iterável** é um objeto em Python que podemos percorrer utilizando um loop. Geralmente pensamos em iteráveis como algum tipo de coleção. Listas, tuplas, dicionários e strings são todos iteráveis.

Porém, o que o loop realmente utiliza não é o iterável, mas o **iterador**. Quando tentamos percorrer um iterável, é criado um iterador a partir dele utilizando a função `iter`. Em cada passo da iteração (do loop), a função `next` é chamada, e ela irá retornar o próximo elemento. Quando os elementos são esgotados, ela irá lançar a exceção (uma espécie de erro sinalizado, que estudaremos em um capítulo futuro) `StopIteration`.

Veja o exemplo abaixo:

```
lista = [1, 3, 5]

iterador = iter(lista)

print(iterador)

print(next(iterador))
print(next(iterador))
print(next(iterador))
print(next(iterador)) # Erro: StopIteration
```

Uma diferença fundamental entre um **iterável** e um **iterador** é que o iterador já possui todos os exemplos salvos em algum lugar. O iterável não. Ele irá gerar/buscar cada elemento no momento que a função `next` é chamada, e ele não irá salvar os elementos anteriores.

Uma expressão geradora não é um **iterável**, ela é um **iterador**. Uma lista é um **iterável**.

Ou seja, quando nós fazemos uma compreensão de lista, a expressão é avaliada na mesma hora e todos os elementos são gerados e salvos na memória.

Quando utilizamos uma expressão geradora, cada elemento é gerado apenas quando solicitado, e os elementos não ficam salvos.

```
gerador = (x for x in range(5))

print(next(gerador))
print(next(gerador))
print(next(gerador))
print(next(gerador))
print(next(gerador))
print(next(gerador)) # Erro: StopIteration
```

Podemos utilizar expressões geradoras quando:

> Iremos trabalhar com uma base de dados tão grande que a geração da lista pode ser excessivamente lenta ou consumir memória demais.
> Quando desejamos obter dados infinitos (uma sequência numérica sem fim, ou então um *stream* de dados que pode estar chegando por um sensor ou pela internet, por exemplo).
> Quando sabemos com certeza absoluta que só precisaremos iterar uma única vez por cada elemento e não precisaremos deles posteriormente.

Caso você precise dos dados mais de uma vez, a expressão geradora deixa de ser atrativa e compensa mais utilizar compreensão de listas.

3. Funções geradoras

Expressões geradoras são uma forma compacta para criar iteradores. Uma das formas mais completas envolve utilizar programação orientada a objeto para definir uma classe com alguns métodos específicos para que os objetos se comportem como geradores. A outra envolve utilizar uma função geradora.

Funções geradoras lembram bastante funções convencionais, mas em vez de `return` elas utilizarão a palavra `yield`.

A função irá retornar um iterador, e iremos sempre chamar `next` passando esse gerador.

Cada vez que o `next` for chamado, o iterador irá executar a função até encontrar o `yield`. O estado da função é salvo e o valor do `yield` é retornado. Quando chamarmos `next` novamente, a função irá executar do ponto que parou até encontrar novamente o `yield`. Quando não houver mais `yield`, a exceção `StopIteraction` será lançada.

Vejamos um exemplo:

```
def funcao_geradora():
    yield 1
    yield 3
```

```
    yield 5

meu_gerador = funcao_geradora()

print(next(meu_gerador))
print(next(meu_gerador))
print(next(meu_gerador))
print(next(meu_gerador)) # erro: StopIteration
```

A nossa função geradora pode, inclusive, ter malhas de repetição:

```
def gerador_de_sequencia(limite:int):
    contador = 0
    while contador < limite:
        yield contador
        contador += 1

iterador_sequencia = gerador_de_sequencia(10)

for x in iterador_sequencia:
    print(x)
```

Caso tenha interesse em se aprofundar nos assuntos deste capítulo e ver alguns experimentos envolvendo tamanho e performance de cada um, segue algumas boas referências:

https://djangostars.com/blog/list-comprehensions-and-generator-expressions/

https://towardsdatascience.com/comprehensions-and-generator-expression-in-python-2ae01c48fc50

https://docs.python.org/3/howto/functional.html#generator-expressions-and-list-comprehensions