Iteração e Generator Functions

Iteração e Generator Functions

▼ Iteráveis x Iteradores (Iterables x Iterators) + (**iter( )** e **next( )**)

Conceito	O que é?	Exemplo
Iterable	Um objeto que pode ser percorrido. Implementa `__iter__()`.	Lista, tupla, string, dicionário, conjunto
Iterator	Um objeto que sabe como retornar o próximo elemento. Implementa `__next__()`.	Criado com `iter(iterável)` ou implementado manualmente

- **Iterable (Iterável)**

❓

- Um **iterável** é qualquer objeto que pode ser percorrido elemento por elemento. Ele implementa o método especial `__iter__()`, que retorna um iterador.
- Exemplos de iteráveis: Listas (`list`), Tuplas (`tuple`), Strings (`str`), Dicionários(`dict`), Conjuntos (`set`)

```
lista = [1, 2, 3]  # Lista é um iterável

for elemento in lista:
```

```
    print(elemento)  # Pode percorrer normalmente usando um loop f
or
```

- Podemos verificar se um objeto é iterável usando `iter()`:

```
iterador = iter(lista)  # Transforma a lista em um iterador
print(iterador)  # <list_iterator object at ...(hexadecimal)>
```

- **Iterator (Iterador)**

> ❓ Um **iterador** é um objeto que mantém o estado da iteração e sabe como obter o próximo elemento. Ele implementa:
>
> - `__iter__()`: Retorna o próprio iterador.
> - `__next__()`: Retorna o próximo elemento ou levanta um erro `StopIteration` quando os elementos acabam.

- Criando um iterador manualmente:

```
lista = [1, 2, 3]
iterador = iter(lista)  # Criando um iterador a partir da lista

print(next(iterador))  # 1
print(next(iterador))  # 2
print(next(iterador))  # 3
print(next(iterador))  # StopIteration (porque não há mais elemento
s)
```

- De forma simples:
  - **Iterable (iterável)** → É um tipo de dado que pode ser percorrido (como listas, tuplas, strings).
  - **Iterator (iterador)** → Funciona como um **ponteiro** que mantém a posição atual dentro do iterável e sabe como obter o próximo elemento.

▼ Definição de generator e comparação com listas

> ❓ Generator é uma forma de criar sequências de valores sob demanda, sem armazená-los na memória. Ele gera cada elemento conforme necessário, economizando recursos. Pode ser criado com generator functions (usando yield) ou generator expressions (usando ( ) em vez de [ ] como em list comprehension).

- **Generator x List**
  - Um generator é uma forma de criar uma sequência de valores sob demanda, gerando cada elemento somente quando for necessário. Diferente de uma lista, que armazena todos os elementos na memória de uma vez, um generator economiza recursos, pois calcula e entrega os valores um por um, conforme forem solicitados.
  - Diferente de listas, generator não possui indexação direta (generator[2] não funciona, por exemplo). Isso ocorre pois como os dados são produzidos dinamicamente, o generator não sabe previamente quais valores ele terá, tornando impossível acessar um elemento específico sem iterar sobre ele.
  - Tabela de comparação:

Característica	**List (`list`)**	**Generator (`generator`)**
Armazena valores?	Sim (todos os elementos são armazenados na memória)	Não (gera os valores sob demanda)
Uso de memória	Alto (depende do tamanho da lista)	Baixo (só mantém um valor por vez)
Velocidade de criação	Mais lento (precisa armazenar todos os valores antes de usar)	Mais rápido (gera valores na hora)
Acesso por índice (`lista[i]`)	Sim	Não
Iteração (`for` `item` `in` `...`)	Sim	Sim

Modificável (`append`, `remove`, etc.)	Sim	Não (é imutável)
Reutilização	Sim (pode ser acessada várias vezes)	Não (após consumir, deve ser recriado)
Uso ideal	Pequenos e médios conjuntos de dados	Grandes volumes de dados ou processamento sob demanda

▼ Generator Expression

❓ Uma **Generator Expression** é uma forma compacta de criar um generator, semelhante a um **list comprehension**, mas que produz os valores sob demanda em vez de armazená-los em memória. Isso torna a execução mais eficiente em termos de uso de memória, especialmente para grandes volumes de dados.

- **Sintaxe**
  - A sintaxe de uma generator expression é semelhante à de um list comprehension, mas usa parênteses `()` em vez de colchetes `[]`.

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

print(next(generator))  # Exibe: 0
print(next(generator))  # Exibe: 1
```

  - Nesse exemplo, generator é um gerador que não armazena os valores na memória, mas gera cada um deles conforme necessário.
- Exemplo com mapeamento:

```
generator = (x ** 2 for x in range(5))

print(next(generator))  # 0
print(next(generator))  # 1
print(next(generator))  # 4
```

```
print(next(generator))  # 9
print(next(generator))  # 16
```

- Exemplo iterando com `for`:

```
generator = (x ** 2 for x in range(5))

for num in generator:
    print(num)  # 0 1 4 9 16
```

▼ Generator Functions (**yield**)

> ❓ Uma **generator function** é uma função que retorna um **iterador** usando a palavra-chave `yield` em vez de `return`. Ela gera valores sob demanda, economizando memória, pois não armazena todos os valores na memória, mas os produz conforme necessário.

- O `yield` é uma palavra-chave usada para **pausar** a execução de uma função e retornar um valor, permitindo que a função seja retomada posteriormente do ponto onde parou. Isso transforma a função em um **generator**, ou seja, um iterador que gera valores sob demanda.
  - Diferente do `return`, que finaliza a função e descarta seu estado, o `yield` **mantém o estado da função**, permitindo continuar a execução de onde parou ao chamar `next()` no gerador.

Característica	`return`	`yield`
Finaliza a função?	Sim	Não, apenas pausa
Armazena estado da função?	Não	Sim
Pode ser chamado múltiplas vezes?	Não	Sim, retoma do ponto onde parou
Tipo de função resultante	Função comum	Generator (iterador)

Uso de memória	Pode ser alto (armazena todos os valores)	Mais eficiente (gera valores sob demanda)

    - Em outras palavras, Generator Function é uma função que pode pausar.

- Exemplo de Generator Function:

```
def contador():
    for i in range(1, 6):
        yield i  # Pausa e retorna o valor atual de i

gen = contador()  # Cria o gerador

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

- Exemplo de Generator Function sem utilizar range():

```
def generator(n=0, maximum=10):
    while True:
        yield n  # Pausa a execução e retorna o valor atual de n
        n += 1  # Incrementa n quando a função é despausada

        if n >= maximum:
            return  # Interrompe quando atinge o limite

gen = generator()

for num in gen:  # Itera sobre o gerador para imprimir os números gera
    print(num)
```

▼ Delegar a iteração a outro iterável ou generator (**yield from**)

❓ O `yield from` é usado em **generators** para delegar a iteração a outro iterável ou **generator**. Ele permite que um **generator** retome a execução de outro **generator** após uma pausa em um `yield`, sem precisar escrever manualmente um loop `for`. Em outras palavras, ele simplifica a forma como você pode "passar" o controle de uma função geradora para outra, facilitando a composição de funções geradoras.

- Quando usamos `yield from`, ele faz o trabalho de iterar sobre o iterável (ou generator) e "retomar" a execução a partir de onde o `yield` foi chamado, passando os valores um a um.
  - Exemplo de delegar a iteração a **outro iterável:**

```
def generator():
    yield from [1, 2, 3]  # Pausa e delega para o iterável [1, 2, 3]

# Testando
for num in generator():
    print(num)  # Exibe: 1 2 3
```

    - Aqui, o `yield from` "pausa" a execução da função `generator` e delega a iteração para o iterável `[ 1, 2, 3 ]`, retornando um valor de cada vez, como se fosse um único fluxo contínuo.
- **Delegando para outro generator:**

❓ Quando usamos `yield from` para delegar a execução a outro **generator**, ele "retoma" a execução da função delegada, pausando a função principal até que o sub-generator termine sua execução.

  - Exemplo de delegar a iteração a **outro generator:**

```
def sub_generator():
    yield "A"
```

```
    yield "B"
    yield "C"

def main_generator():
    yield 0
    # Aqui, a execução do main_generator é pausada e a execução
    yield from sub_generator()  # O controle vai para sub_generator
    yield 1  # Após sub_generator terminar, a execução volta para m

# Testando
for item in main_generator():
    print(item)

# Exibe:
0
A
B
C
1
```