Python - Geradores

Um gerador em Python é um tipo especial de função que retorna um objeto iterador. Parece semelhante a uma função Python normal, pois sua definição também começa com a palavra-chave def. No entanto, em vez da instrução return no final, o gerador usa a palavra-chave yield.

Sintaxe

```
def generator():
 . . .
 . . .
 yield obj
it = generator()
next(it)
. . .
```

A instrução return no final da função indica que a execução do corpo da função terminou, todas as variáveis locais da função saem do escopo. Se a função for chamada novamente, as variáveis locais serão reinicializadas.

A função do gerador se comporta de maneira diferente. Ela é invocada pela primeira vez como uma função normal, mas quando sua instrução yield chega, sua execução é temporariamente pausada, transferindo o controle de volta. O resultado produzido é consumido pelo chamador. A chamada para a função interna next() reinicia a execução do gerador a partir do ponto em que foi pausado e gera o próximo objeto para o iterador. O ciclo se repete conforme o rendimento subsequente fornece o próximo item no iterador que está esgotado.

Exemplo 1

A função no código abaixo é um gerador que produz números inteiros sucessivamente de 1 a 5. Quando chamada, ela retorna um iterador. Cada chamada para next() transfere o controle de volta para o gerador e busca o próximo número inteiro.

```
def generator(num):
   for x in range(1, num+1):
      yield x
   return

it = generator(5)
while True:
   try:
      print (next(it))
```

```
   except StopIteration:
      break
```

Ele produzirá a seguinte **saída** -

```
1
2
3
4
5
```

A função geradora retorna um iterador dinâmico. Conseqüentemente, é mais eficiente em termos de memória do que um iterador normal obtido de um objeto de sequência Python. Por exemplo, se você deseja obter os primeiros n números da série Fibonacci. Você pode escrever uma função normal e construir uma lista de números de Fibonacci e, em seguida, iterar a lista usando um loop.

Exemplo 2

Abaixo está a função normal para obter uma lista de números de Fibonacci -

```
def fibonacci(n):
   fibo = []
   a, b = 0, 1
   while True:
      c=a+b
      if c>=n:
         break
      fibo.append(c)
      a, b = b, c
   return fibo
f = fibonacci(10)
for i in f:
   print (i)
```

Ele produzirá a seguinte saída -

```
1
2
3
5
8
```

O código acima coleta todos os números da série Fibonacci em uma lista e então a lista é percorrida usando um loop. Imagine que queremos que a série de Fibonacci atinja um grande

número. Nesse caso, todos os números devem ser reunidos em uma lista que exige muita memória. É aqui que o gerador é útil, pois gera um único número na lista e o fornece para consumo.

Exemplo 3

O código a seguir é a solução baseada em gerador para a lista de números de Fibonacci -

```
def fibonacci(n):
   a, b = 0, 1
   while True:
      c=a+b
      if c>=n:
         break
      yield c
      a, b = b, c
   return

f = fibonacci(10)
while True:
   try:
      print (next(f))
   except StopIteration:
      break
```

Gerador Assíncrono

Um gerador assíncrono é uma corrotina que retorna um iterador assíncrono. Uma corrotina é uma função Python definida com a palavra-chave async e pode agendar e aguardar outras corrotinas e tarefas. Assim como um gerador normal, o gerador assíncrono produz itens incrementais no iterador para cada chamada para a função anext(), em vez da função next().

Sintaxe

```
async def generator():
. . .
. . .
yield obj
it = generator()
anext(it)
. . .
```

Exemplo 4

O código a seguir demonstra um gerador de corrotina que produz números inteiros incrementais em cada iteração de um loop **for assíncrono** .

```
import asyncio

async def async_generator(x):
   for i in range(1, x+1):
      await asyncio.sleep(1)
      yield i

async def main():
   async for item in async_generator(5):
      print(item)

asyncio.run(main())
```

Ele produzirá a seguinte **saída** -

```
1
2
3
4
5
```

Exemplo 5

Vamos agora escrever um gerador assíncrono para números de Fibonacci. Para simular alguma tarefa assíncrona dentro da corrotina, o programa chama o método sleep() por 1 segundo antes de gerar o próximo número. Como resultado, você obterá os números impressos na tela após um atraso de um segundo.

```
import asyncio

async def fibonacci(n):
   a, b = 0, 1
   while True:
      c=a+b
      if c>=n:
         break
      await asyncio.sleep(1)
      yield c
      a, b = b, c
   return

async def main():
```

```
    f = fibonacci(10)
    async for num in f:
        print (num)

asyncio.run(main())
```

Ele produzirá a seguinte **saída** -

```
1
2
3
5
8
```