Python - Iteradores

Iterador em Python é um objeto que representa um fluxo de dados. Ele segue o protocolo iterador que exige suporte aos métodos __iter__() e __next__(). O método integrado iter() do Python implementa o método __iter__(). Ele recebe um objeto iterável e retorna um objeto iterador. A função next() integrada chama internamente o método __next__() do iterador e retorna itens sucessivos no fluxo. Quando não há mais dados disponíveis, uma exceção StopIteration é gerada.

Python usa iteradores implicitamente ao trabalhar com tipos de dados de coleção, como lista, tupla ou string. É por isso que esses tipos de dados são chamados de iteráveis. Normalmente usamos o loop for para iterar por meio de um iterável da seguinte maneira -

```
for element in sequence:
 print (element)
```

O método integrado iter() do Python implementa o método __iter__(). Ele recebe um objeto iterável e retorna um objeto iterador.

Exemplo

O código a seguir obtém o objeto iterador da lista de tipos de sequência, string e tupla. A função iter() também retorna keyiterator do dicionário. No entanto, int id não é iterável, portanto produz TypeError.

```
print (iter("aa"))
print (iter([1,2,3]))
print (iter((1,2,3)))
print (iter({}))
print (iter(100))
```

Ele produzirá a seguinte **saída** -

```
<str_ascii_iterator object at 0x000001BB03FFAB60>
<list_iterator object at 0x000001BB03FFAB60>
<tuple_iterator object at 0x000001BB03FFAB60>
<dict_keyiterator object at 0x000001BB04181670>
Traceback (most recent call last):
  File "C:\Users\user\example.py", line 5, in <module>
    print (iter(100))
          ^^^^^^^^^
TypeError: 'int' object is not iterable
```

O objeto Iterator possui o método __next__(). Cada vez que é chamado, ele retorna o próximo elemento no fluxo do iterador. Quando o fluxo se esgota, o erro StopIteration é gerado. Chamar a função next() é equivalente a chamar o método __next__() do objeto iterador.

Exemplo

```
it = iter([1,2,3])
print (next(it))
print (it.__next__())
print (it.__next__())
print (next(it))
```

Ele produzirá a seguinte **saída** -

```
1
2
3
Traceback (most recent call last):
  File "C:\Users\user\example.py", line 5, in <module>
    print (next(it))
         ^^^^^^^^
StopIteration
```

Exemplo

Você pode usar o mecanismo de tratamento de exceções para capturar StopIteration.

```
it = iter([1,2,3, 4, 5])
print (next(it))
while True:
   try:
      no = next(it)
      print (no)
   except StopIteration:
      break
```

Ele produzirá a seguinte **saída** -

```
1
2
3
4
5
```

Para definir uma classe iteradora personalizada em Python, a classe deve definir os métodos __iter__() e __next__().

No exemplo a seguir, Oddnumbers é uma classe que implementa os métodos __iter__() e __next__(). Em cada chamada para __next__(), o número aumenta em 2, transmitindo assim números ímpares no intervalo de 1 a 10.

Exemplo

```
class Oddnumbers:

   def __init__(self, end_range):
      self.start = -1
      self.end = end_range

   def __iter__(self):
      return self

   def __next__(self):
      if self.start < self.end-1:
         self.start += 2
         return self.start
      else:
         raise StopIteration

countiter = Oddnumbers(10)
while True:
   try:
      no = next(countiter)
      print (no)
   except StopIteration:
      break
```

Ele produzirá a seguinte **saída** -

```
1
3
5
7
9
```

Iterador assíncrono

Duas funções integradas aiter() e anext() foram adicionadas na versão 3.10 do Python em diante. A função aiter() retorna um objeto iterador assíncrono. É uma contraparte assíncrona

do iterador clássico. Qualquer iterador assíncrono deve suportar os métodos __aiter__() e __anext__(). Esses métodos são chamados internamente pelas duas funções integradas.

Assim como o iterador clássico, o iterador assíncrono fornece um fluxo de objetos. Quando o fluxo se esgota, a exceção StopAsyncIteration é gerada.

No exemplo abaixo, uma classe iteradora assíncrona Oddnumbers é declarada. Ele implementa os métodos __aiter__() e __anext__(). A cada iteração, um próximo número ímpar é retornado e o programa aguarda um segundo para poder executar qualquer outro processo de forma assíncrona.

Ao contrário das funções regulares, as funções assíncronas são chamadas de corrotinas e são executadas com o método asyncio.run(). A corrotina main() contém um loop while que obtém sucessivamente números ímpares e gera StopAsyncIteration se o número exceder 9.

Exemplo

```
import asyncio

class Oddnumbers():
   def __init__(self):
      self.start = -1

   def __aiter__(self):
      return self

   async def __anext__(self):
      if self.start >= 9:
         raise StopAsyncIteration
      self.start += 2
      await asyncio.sleep(1)
      return self.start

async def main():
   it = Oddnumbers()
   while True:
      try:
         awaitable = anext(it)
         result = await awaitable
         print(result)
      except StopAsyncIteration:
         break

asyncio.run(main())
```

Ele produzirá a seguinte **saída** -

```
1
3
5
7
9
```