Generators و Iterators در پایتون

در پایتون، **Generators** و **Iterators** ابزارهایی برای تولید داده‌ها به‌صورت تدریجی و در صورت نیاز (lazy loading) هستند. این مفاهیم بسیار مفید در پردازش داده‌های بزرگ و یا اجرای کارهای پیچیده به‌صورت غیرهمزمان و موازی به شمار می‌آیند، چرا که به جای بارگذاری همه داده‌ها در حافظه، داده‌ها را فقط در زمان مورد نیاز تولید می‌کنند.

---

1. معرفی Generators به عنوان راهی برای تولید داده‌ها به صورت Lazy

یک **Generator** یک تابع است که به جای بازگرداندن یک لیست کامل از داده‌ها، با استفاده از دستور `yield` داده‌ها را به تدریج (lazy) تولید می‌کند. این یعنی شما تنها زمانی که به داده‌ها نیاز دارید، آن‌ها را دریافت می‌کنید. این روش حافظه‌ی کمتری مصرف می‌کند و عملکرد بهتری دارد، به‌ویژه هنگام کار با مجموعه‌های داده بزرگ.

مثال 1: تعریف یک Generator ساده

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

# استفاده از generator
counter = count_up_to(5)
for num in counter:
    print(num)
```

در این مثال، هر بار که `yield` فراخوانی می‌شود، مقدار جدیدی به خارج از تابع باز می‌گردد و تابع به حالت متوقف‌شده بازمی‌گردد تا دفعه بعدی که فراخوانی شود.

---

2. استفاده از `yield` برای ساختن Generatorها

کلمه‌کلیدی `yield` در پایتون برای تعریف generatorها استفاده می‌شود. وقتی از `yield` در تابع استفاده می‌کنید، آن تابع تبدیل به یک **generator function** می‌شود. هر بار که تابع فراخوانی شود، تابع تا دستور `yield` اجرا می‌شود و مقدار بازگشتی را می‌دهد، سپس وضعیت تابع حفظ می‌شود و هنگام فراخوانی بعدی از همان نقطه ادامه می‌یابد.

مثال 2: استفاده از `yield` برای ایجاد یک generator

```
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

# استفاده از generator برای دریافت دنباله فیبوناچی
for num in fibonacci(10):
    print(num)
```

در این مثال، با استفاده از `yield`، دنباله فیبوناچی به‌طور تدریجی تولید می‌شود.

---

3. تفاوت‌های بین Iterators و Generators

- Iterators
  :
  - یک شیء است که می‌تواند به‌طور پیوسته از داده‌ها عبور کند و داده‌ها را یک به یک بازگشت دهد.
  - باید متدهای `__iter__()` و `__next__()` را پیاده‌سازی کند.
  - مثالی از یک iterator می‌تواند یک شیء لیست باشد که می‌توانیم با استفاده از `for` از آن عبور کنیم.
- Generators
  :
  - یک نوع خاص از iterator است که از `yield` برای تولید داده‌ها به‌طور lazy استفاده می‌کند.
  - کد ساده‌تر و تمیزتری برای ایجاد یک generator دارید چون نیازی به پیاده‌سازی متدهای `__iter__()` و `__next__()` نیست.
  - Generators حافظه‌ی کمتری مصرف می‌کنند چرا که تنها یک مقدار از داده‌ها را در حافظه نگه می‌دارند و نه همه داده‌ها را به یک‌باره.

مثال 3: مقایسه Generator و Iterator

```
# استفاده از یک Iterator
numbers = [1, 2, 3, 4, 5]
iterator = iter(numbers)
print(next(iterator))  # 1
print(next(iterator))  # 2

# استفاده از یک Generator
def number_generator():
    yield 1
    yield 2
    yield 3

gen = number_generator()
print(next(gen))  # 1
print(next(gen))  # 2
```

در اینجا می‌بینیم که هر دو iterator و generator به‌طور مشابه عمل می‌کنند، اما تفاوت در نحوه پیاده‌سازی است. در generator، تنها نیاز به تابعی داریم که با `yield` مقادیر را تولید کند.

---

4. مزایای استفاده از Generators در برنامه‌های با داده‌های بزرگ

Generators به‌ویژه در پردازش داده‌های بزرگ مفید هستند زیرا:

- **کمتر مصرف حافظه**: به‌جای بارگذاری همه داده‌ها در حافظه، داده‌ها تنها در زمان نیاز تولید می‌شوند.
- **بهبود کارایی**: برای عملیات‌هایی که نیاز به پردازش تدریجی دارند، مانند خواندن فایل‌های بزرگ یا پردازش جریان‌های داده‌ای از شبکه، generators کارایی بهتری دارند.
- **خواندن داده‌ها از فایل یا شبکه**: می‌توان از generatorها برای خواندن داده‌ها از فایل‌ها یا شبکه به‌صورت تدریجی استفاده کرد، که باعث کاهش مصرف حافظه و زمان می‌شود.

مثال 4: استفاده از Generators برای خواندن فایل‌ها

```
def read_large_file(file_name):
    with open(file_name, 'r') as f:
        for line in f:
            yield line

# خواندن فایل به صورت lazy
for line in read_large_file('large_file.txt'):
    print(line.strip())
```

در این مثال، به‌جای خواندن تمام خطوط فایل در حافظه، از `yield` برای خواندن هر خط به‌صورت lazy استفاده می‌شود، که مصرف حافظه را کاهش می‌دهد.

---

5. Generators و پردازش‌های شبکه

می‌توان از generators برای پردازش داده‌ها از شبکه به‌صورت lazy استفاده کرد. به‌عنوان مثال، می‌توان داده‌ها را از یک API به‌طور تدریجی دریافت کرد.

مثال 5: استفاده از Generators برای دریافت داده‌ها از یک API

```
import requests

def fetch_data_from_api(url):
    response = requests.get(url)
    for item in response.json():
        yield item

# دریافت داده‌ها به صورت lazy
for data in fetch_data_from_api('https://api.example.com/data'):
    print(data)
```

در اینجا، داده‌ها به‌صورت تدریجی از API دریافت و پردازش می‌شوند.

نتیجه‌گیری

Generators ابزارهایی قدرتمند برای تولید داده‌ها به‌صورت lazy هستند و در برنامه‌های با داده‌های بزرگ یا هنگام نیاز به پردازش تدریجی داده‌ها بسیار مفیدند. آن‌ها با استفاده از `yield` منابع را به‌طور کارآمد مصرف کرده و پردازش‌های بهینه‌تری را فراهم می‌کنند. تفاوت‌های اصلی بین **Iterators** و **Generators** در نحوه پیاده‌سازی و استفاده از حافظه است که باعث می‌شود generators انتخاب بهتری برای پردازش‌های تدریجی باشند.