Generators و Iterators

در این بخش از دوره، تفاوت بین تکرار و تولید در پایتون را خواهیم آموخت و یاد خواهیم گرفت چگونه تولیدکننده‌های خود را با استفاده از دستور *yield* بسازیم. مولدها به ما امکان می‌دهند تا در طول اجرا، مقادیر را تولید کنیم به جای نگهداری همه چیز در حافظه.

قبلاً در بحثی درباره برخی از توابع پیش‌فرض پایتون مانند **range()**و **map()** و **filter()** به این موضوع اشاره کرده‌ایم.

بیایید کمی عمیق‌تر بررسی کنیم. یاد گرفتیم که چگونه تابعی با استفاده از دستور `def` و دستور `return` بنویسیم. توابع مولد به ما امکان می‌دهند تا یک مقدار را ارسال کنیم و سپس بعداً به آنجا برگردیم و از جایی که متوقف شده بود ادامه دهیم. این نوع تابع در پایتون یک تولیدکننده است که به ما امکان می‌دهد تا به مرور زمان یک دنباله از مقادیر را تولید کنیم. تفاوت اصلی در نحو ساختاری استفاده از دستور `yield` خواهد بود.

از نظر بیشتر جنبه‌ها، تابع مولد بسیار شبیه یک تابع عادی به نظر خواهد رسید. تفاوت اصلی این است که وقتی یک تابع تولیدکننده کامپایل می‌شود، آنها به شیءی تبدیل می‌شوند که پروتکل تکرار را پشتیبانی می‌کند. این بدان معناست که وقتی آنها در کد شما فراخوانی می‌شوند، در واقع یک مقدار برنمی‌گردانند و سپس خارج نمی‌شوند. به جای آن، توابع مولد به طور خودکار اجرای خود را متوقف و از نقطه آخر تولید مقدار ادامه می‌دهند. مزیت اصلی در اینجا این است که به جای محاسبه یک دنباله کامل از مقادیر به صورت پیش‌فرض، مولد یک مقدار را محاسبه کرده و سپس فعالیت خود را متوقف می‌کند و منتظر دستور بعدی می‌ماند. این ویژگی به عنوان *متوقف کردن وضعیت* شناخته می‌شود.

برای بهتر درک کردن مولدها، بیایید به ساخت آنها بپردازیم.

In [1]:

```
def cubes(n):
    c = []
    for num in range(n):
        c.append(num ** 3)
    return c

cubes(10)
```

Out[1]:

```
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

In [2]:

```
# Generator function for the cube of numbers (power of 3)
def gencubes(n):
    for num in range(n):
        yield num ** 3
        #print(num)
```

In [3]:

```
g = gencubes(5)
```

In [4]:

```
1 g
```

Out[4]:

```
<generator object gencubes at 0x000001FB44288190>
```

In [5]:

```
1 next(g)
```

Out[5]:

```
0
```

In [6]:

```
1 next(g)
```

Out[6]:

```
1
```

In [7]:

```
1 next(g)
```

Out[7]:

```
8
```

In [8]:

```
1 for x in gencubes(10):
2     print(x)
```

```
0
1
8
27
64
125
216
343
512
729
```

خوب! حالا که یک تابع مولد داریم، نیازی نیست که از هر مکعبی که ایجاد کرده‌ایم، پیگیری کنیم.

مولدها برای محاسبه مجموعه‌های بزرگی از نتایج (به ویژه در محاسباتی که شامل حلقه‌های خودشان هستند) در مواردی که نخواهیم حافظه را برای همه نتایج به صورت همزمان تخصیص دهیم، بهترین راه حل هستند.

بیایید یک مثال دیگر از یک تولیدکننده بسازیم که اعداد فیبوناچی (https://en.wikipedia.org/wiki/Fibonacci_number) را محاسبه می‌کند:

In [9]:

```
def genfibon(n):
    """
    Generate a fibonnaci sequence up to n
    """
    a = 1
    b = 1
    for i in range(n):
        yield a
        a,b = b,a+b
```

In [10]:

```
for num in genfibon(10):
    print(num)
```

```
1
1
2
3
5
8
13
21
34
55
```

اگر این یک تابع عادی بود، چگونه به نظر می‌رسید؟

In [11]:

```
def fibon(n):
    a = 1
    b = 1
    output = []

    for i in range(n):
        output.append(a)
        a,b = b,a+b

    return output
```

In [12]:

```
fibon(10)
```

Out[12]:

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

توجه کنید که اگر مقدار بسیار بزرگی از n را (مانند 100000) صدا بزنیم، تابع دوم باید از هر نتیجه‌ای پیگیری کند، در حالی که در مورد ما تنها اهمیت داریم که نتیجه قبلی را برای تولید نتیجه بعدی داشته باشیم!

درک بهتر مولدها

تا اینجا، درباره دو روش اصلی ایجاد مولدها آموخته‌اید: استفاده از توابع تولیدکننده و عبارات تولیدکننده. شاید حتی درکی شهودی از نحوه کار مولدها داشته باشید. بیایید یک لحظه وقت بگذاریم تا این دانش را یکمی صریح‌تر کنیم.

توابع مولد به نظر می‌رسند و همانند توابع عادی عمل می‌کنند، با این تفاوت که یک ویژگی تمایزدهنده دارند. توابع مولد از کلیدواژه yield Python به جای return استفاده می‌کنند. به خاطر بیاورید تابع مولدی که قبلاً نوشته بودید:

In [ ]:

```
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1
```

In [ ]:

```
for num in infinite_sequence():
    print(num)
```

این به نظر می‌رسد مانند یک تعریف تابع معمولی است، به استثنای عبارت yield Python و کدی که بعد از آن قرار دارد. yield نشان می‌دهد که کدام مقدار به call ارسال می‌شود، اما بر خلاف return، شما پس از آن از تابع خارج نمی‌شوید.

به جای آن، حالت تابع به یاد آورده می‌شود. در این صورت، وقتی next() بر روی یک شیء مولد (به صورت صریح یا ضمنی در یک حلقه for) فراخوانی می‌شود، متغیر num که قبلاً yield شده بود، افزایش می‌یابد و دوباره yield می‌شود. از آنجا که توابع تولیدکننده مانند سایر توابع به نظر می‌رسند و به شدت شبیه به آن‌ها عمل می‌کنند، می‌توانید فرض کنید که عبارات تولیدکننده بسیار شبیه به سایر توانایی‌های موجود در پایتون هستند.

ساخت مولد با استفاده از Generator Expressions

مانند عبارت‌های comprehensions، عبارت‌های مولد به شما اجازه می‌دهند تا در چند خط کد، به سرعت یک شیء تولیدکننده بسازید. آن‌ها همچنین در همان مواردی که عبارات comprehensions استفاده می‌شوند، مفید هستند با این تفاوت که شما می‌توانید آن‌ها را بدون ساخت و نگه‌داری کامل شیء در حافظه قبل از تکرار، ایجاد کنید. به عبارت دیگر، هیچ عواقب حافظه‌ای وجود ندارد هنگام استفاده از عبارات تولیدکننده. برای مثال، نگاهی به مربع کردن برخی اعداد بیندازید:

In [13]:

```
nums_squared_lc = [num**2 for num in range(5)]
nums_squared_gc = (num**2 for num in range(5))
```

هر دو nums_squared_lc و nums_squared_gc به طور کلی به یک شکل به نظر می‌رسند، اما یک تفاوت کلیدی وجود دارد. آیا می‌توانید آن را بیابید؟ نگاهی به اینکه در هنگام بررسی هر یک از این اشیاء چه اتفاقی می‌افتد:

In [14]:

```
1 nums_squared_lc
```

Out[14]:

```
[0, 1, 4, 9, 16]
```

In [15]:

```
1 nums_squared_gc
```

Out[15]:

```
<generator object <genexpr> at 0x000001FB44288B30>
```

شیء اول از براکت ها برای ساختن یک لیست استفاده کرد، در حالی که شیء دوم با استفاده از پرانتزها عبارت مولد ایجاد کرد. خروجی تأیید می‌کند که یک شیء تولیدکننده ایجاد کرده‌اید و که این شیء از یک لیست متمایز است.

In [16]:

```
1 next(nums_squared_gc)
```

Out[16]:

```
0
```

In [17]:

```
1 next(nums_squared_gc)
```

Out[17]:

```
1
```

In [18]:

```
1 next(nums_squared_gc)
```

Out[18]:

```
4
```

ارزیابی کارایی یک مولد

قبلاً یاد گرفتید که مولدها یک روش عالی برای بهینه‌سازی حافظه هستند. در حالی که یک تولیدکننده دنباله بی‌نهایت یک مثال مطلق از این بهینه‌سازی است، بیایید مثال‌های مربع کردن اعداد را که همین‌که دیدید را به سطح بالاتر ببریم و اندازه اشیاء حاصل را بررسی کنیم. شما می‌توانید این کار را با فراخوانی تابع sys.getsizeof() انجام دهید:

In [19]:

```
import sys
nums_squared_lc = [i ** 2 for i in range(10000)]
print(sys.getsizeof(nums_squared_lc))
```

```
85176
```

In [20]:

```
nums_squared_gc = (i ** 2 for i in range(10000))
print(sys.getsizeof(nums_squared_gc))
```

```
104
```

در این مورد، اندازه لیستی که از عبارت مولد لیست دریافت می‌کنید 87624 بایت است، در حالی که شیء مولد فقط 104 بایت است. این به این معنی است که اندازه لیست بیش از 842 برابر اندازه شیء مولد است!

اما یک نکته را به خاطر داشته باشید. اگر اندازه لیست کوچکتر از حافظه قابل دسترس ماشین باشد، آنگاه عبارات مولد لیست ممکن است سریع‌تر ارزیابی شوند نسبت به عبارات مولد معادل. برای بررسی این موضوع، بیایید جمع مقادیر به دست آمده از دو عبارت مولد را محاسبه کنیم. می‌توانید با استفاده از `cProfile.run()` خروجی را تولید کنید.

In [21]:

```
import cProfile
cProfile.run('sum([i * 2 for i in range(10000)])')
```

```
         5 function calls in 0.001 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001    0.001    0.001 <string>:1(<listcomp>)
        1    0.000    0.000    0.001    0.001 <string>:1(<module>)
        1    0.000    0.000    0.001    0.001 {built-in method builtins.ex
ec}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.su
m}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lspro
f.Profiler' objects}
```

In [22]:

```
cProfile.run('sum((i * 2 for i in range(10000)))')
```

```
         10005 function calls in 0.002 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    10001    0.001    0.000    0.001    0.000 <string>:1(<genexpr>)
        1    0.000    0.000    0.002    0.002 <string>:1(<module>)
        1    0.000    0.000    0.002    0.002 {built-in method builtins.ex
ec}
        1    0.001    0.001    0.002    0.002 {built-in method builtins.su
m}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lspro
f.Profiler' objects}
```

در اینجا می‌بینید که جمع کردن همه مقادیر در عبارت مولدی لیست تقریباً برابر زمان جمع کردن در مولد انجام شد. اما بصورت کلی، اگر سرعت مهم است و حافظه نه، آن‌گاه یک عبارت list comprehension ابزار بهتری برای این کار است.

توجه: این اندازه‌گیری‌ها فقط برای اشیاء ساخته شده با عبارات مولد معتبر نیستند. آن‌ها همچنین برای اشیاء ساخته شده از تابع مولد مشابه هستند، زیرا مولدهای حاصل معادل هستند.

بیاد داشته باشید که عبارات مولد لیست‌های کامل را برمی‌گردانند، در حالی که عبارات مولد مجموعه‌ها را برمی‌گردانند. مولدها بدون تفاوت کار می‌کنند، بدون اینکه از تابع یا عبارت ساخته شوند. استفاده از یک عبارت فقط به شما امکان می‌دهد تا مولدهای ساده را در یک خط تعریف کنید، با فرض وجود یک yield در انتهای هر حلقه داخلی.

یادآوری: عبارت yield پایه‌ای است که تمام قابلیت‌های مولدها بر آن تکیه دارد، بنابراین بیایید به بررسی نحوه کارکرد yield در پایتون بپردازیم.

درک بهتر دستور yeild

در کل، yield یک عبارت بسیار ساده است. وظیفه اصلی آن کنترل جریان یک تابع مولد به گونه‌ای است که شبیه به عبارت‌های return است. همانطور که در بالا به طور خلاصه اشاره شد، عبارت yield در پایتون چند ترفند ویژه دارد.

وقتی یک تابع مولد را فراخوانی می‌کنید یا از یک عبارت مولد استفاده می‌کنید، یک مولد ویژه به نام مولد (generator) برگردانده می‌شود. می‌توانید این مولد را به یک متغیر اختصاص دهید تا از آن استفاده کنید. وقتی روش‌های ویژه‌ای مانند next() را بر روی مولد فراخوانی می‌کنید، کد درون تابع تا عبارت yield اجرا می‌شود.

زمانی که عبارت yield پایتون اجرا می‌شود، برنامه اجرای تابع را متوقف می‌کند و مقداری که به آن yield شده است را به فراخواننده برمی‌گرداند. (در مقابل، عبارت return اجرای تابع را به طور کامل متوقف می‌کند.) وقتی یک تابع متوقف می‌شود، وضعیت آن تابع ذخیره می‌شود. این شامل هر اتصال متغیری که محلی در مولد است، نشانگر دستور، پشته داخلی و هرگونه مدیریت استثناء است.

این قابلیت به شما امکان می‌دهد تا هر زمان که یکی از متدهای مولد را فراخوانی کنید، اجرای تابع را ادامه دهید. به این صورت، ارزیابی تابع به‌صورت کامل از جایی که عبارت yield قرار دارد، ادامه پیدا می‌کند. این را می‌توانید با استفاده از چندین عبارت yield در عمل مشاهده کنید.

In [23]:

```
def multi_yield():
    yield_str = "This will print the first string"
    yield yield_str
    yield_str = "This will print the second string"
    yield yield_str
```

In [24]:

```
multi_obj = multi_yield()
```

In [25]:

```
print(next(multi_obj))
```

```
This will print the first string
```

In [26]:

```
print(next(multi_obj))
```

```
This will print the second string
```

In [27]:

```
print(next(multi_obj))
```

```
---------------------------------------------------------------------------
-
StopIteration                             Traceback (most recent call las
t)
Cell In[27], line 1
----> 1 print(next(multi_obj))

StopIteration:
```

نگاهی دقیق‌تر به آخرین فراخوانی `next()` بیندازید. می‌توانید ببینید که اجرا با یک `traceback` متوقف شده است. دلیل این امر این است که تولیدکننده‌ها، مانند سایر تکرارکننده‌ها، قابل استنفاده هستند. مگر اینکه تولیدکننده شما بی‌نهایت باشد، می‌توانید فقط یک بار از آن عبور کنید. پس از ارزیابی تمامی مقادیر، تکرار متوقف شده و حلقه `for` خاتمه خواهد یافت. اگر از `next()` استفاده کردید، در عوض یک استثناء صریح `StopIteration` دریافت خواهید کرد.

`yield` می‌تواند برای کنترل جریان اجرای تولیدکننده شما به شکل‌های مختلفی استفاده شود. استفاده از چندین دستور `yield` در پایتون می‌تواند تا جایی که خلاقیت شما اجازه دهد، بهره‌برداری شود.

استفاده از متدهای پیشرفته مولدها

شما روش‌ها و ساختارهای متداول ترین مولدها را دیده‌اید، اما چند ترفند دیگر برای پوشش وجود دارد. علاوه بر yield، اشیاء مولد می‌توانند از متدهای زیر استفاده کنند:

- .send()
- .throw()

.send()

اینجا یک مثال از استفاده از `.send()` در مولدهای پایتون است:

```
def generator():
    while True:
        x = yield
        print(x)

g = generator()
next(g)  # شروع مولد
g.send(1)  # چاپ 1
g.send(2)  # چاپ 2
```

در این مثال، تابع `generator()` به عنوان یک مولد تعریف شده است که مقادیر را با استفاده از دستور `yield` باز می‌گرداند. دستور `.send()` برای ارسال مقدار به داخل مولد و ادامه اجرای آن استفاده می‌شود. در این حالت، مقداری که به داخل مولد فرستاده شده، چاپ می‌شود.

In [28]:

```
def generator():

    while True:
        x = yield
        print(x)
```

In [29]:

```
g = generator()
```

In [34]:

```
next(g)
```

None

In [35]:

```
g.send(10)
```

10

In [36]:

```
g.send(2)
```

2

.throw()

اینجا یک مثال از استفاده از `.throw()` در مولدهای پایتون است:

```
def generator():
    try:
        while True:
            x = yield
            print(x)
    except ValueError:
        print("Caught a ValueError")

g = generator()
next(g)  # شروع مولد
g.send(1)  # چاپ 1
g.throw(ValueError)  # چاپ "Caught a ValueError"
```

در این مثال، تابع `generator()` به عنوان یک مولد تعریف شده است که مقادیر را با استفاده از دستور `yield` باز می‌گرداند. دستور `.throw()` برای پرتاب استثناء در داخل مولد و ادامه اجرای آن استفاده می‌شود. در این حالت، استثناء پرتاب شده توسط بلوک `try` دریافت شده و پیام "Caught a ValueError" چاپ می‌شود.

In [37]:

```
def generator():
    try:
        while True:
            x = yield
            print(x)
    except ValueError:
        print("Caught a ValueError")
```

In [38]:

```
g = generator()
```

In [39]:

```
next(g)
```

In [40]:

```
g.send(1)
```

```
1
```

In [41]:

```
g.throw(ValueError)
```

```
Caught a ValueError

---------------------------------------------------------------------------
-
StopIteration                             Traceback (most recent call las
t)
Cell In[41], line 1
----> 1 g.throw(ValueError)

StopIteration:
```

.close()

در این مثال، یک تابع مولد به نام my_generator() تعریف شده است که به طور نامحدود حلقه می‌زند و هر بار یک مقدار را دریافت می‌کند. روش close() برای متوقف کردن تکرار مولد و پرتاب یک استثناء GeneratorExit در داخل مولد استفاده می‌شود. در این حالت، استثناء پرتاب شده توسط بلوک try مشخص شده و پیام "Generator has been closed" چاپ می‌شود.

```
def my_generator():
    try:
        while True:
            x = yield
    except GeneratorExit:
        print("Generator has been closed")

g = my_generator()
next(g)
g.close()  # Output: "Generator has been closed"
```

In [42]:

```
def generator():
    try:
        while True:
            x = yield
            print(x)
    except GeneratorExit:
        print("Generator has been closed")

g = generator()

```

In [44]:

```
next(g)
```

```
None
```

In [45]:

```
1 g.close()
```

```
Generator has been closed
```

In [46]:

```
1 next(g)
```

```
---------------------------------------------------------------------------
-
StopIteration                             Traceback (most recent call las
t)
Cell In[46], line 1
----> 1 next(g)

StopIteration:
```