Python 中的黑暗角落（一）：理解 yield 关键字

https://liam.page/2017/06/30/understanding-yield-in-python/

本文作者： Liam Huang

Tue Oct, 27 15:01

Python 是非常灵活的语言，其中 `yield` 关键字是普遍容易困惑的概念。

此篇将介绍 `yield` 关键字，及其相关的概念。

迭代、可迭代、迭代器

迭代（iteration）与可迭代（iterable）

> 迭代是一种操作；可迭代是对象的一种特性。

很多数据都是「容器」；它们包含了很多其他类型的元素。实际使用容器时，我们常常需要逐个获取其中的元素。**逐个获取元素的过程，就是「迭代」**。

```
1
2 a_list = [1, 2, 3]
3 for i in a_list:
4     print(i)
```

如果我们可以从一个对象中，逐个地获取元素，那么我们就说这个对象是「可迭代的」。

Python 中的顺序类型，都是可迭代的（`list`，`tuple`，`string`）。其余包括 `dict`，`set`，`file` 也是可迭代的。对于用户自己实现的类型，如果提供了 `__iter__()` 或者 `__getitem__()` 方法，那么该类的对象也是可迭代的。

迭代器（iterator）

> 迭代器是一种对象。

迭代器抽象的是一个「数据流」，是只允许迭代一次的对象。对迭代器不断调用 `next()` 方法，则可以依次获取下一个元素；当迭代器中没有元素时，调用 `next()` 方法会抛出 `StopIteration` 异常。迭代器的 `__iter__()` 方法返回迭代器自身；因此迭代器也是可迭代的。

迭代器协议（iterator protocol）

> 迭代器协议指的是容器类需要包含一个特殊方法。

如果一个容器类提供了 `__iter__()` 方法，并且该方法能返回一个能够逐个访问容器内所有元素的迭代器，则我们说该容器类实现了迭代器协议。

Python 中的迭代器协议和 Python 中的 `for` 循环是紧密相连的。

```

for x in something:
    print(x)
```

Python 处理 `for` 循环时，首先会调用内建函数 `iter(something)`，它实际上会调用 `something.__iter__()`，返回 `something` 对应的迭代器。而后，`for` 循环会调用内建函数 `next()`，作用在迭代器上，获取迭代器的下一个元素，并赋值给 `x`。此后，Python 才开始执行循环体。

生成器、`yield` 表达式

生成器函数（generator function）和生成器（generator）

> 生成器函数是一种特殊的函数；生成器则是特殊的迭代器。

如果一个函数包含 `yield` 表达式，那么它是一个生成器函数；调用它会返回一个特殊的迭代器，称为生成器。

```
def func():
    return 1

def gen():
    yield 1

print(type(func))
print(type(gen))

print(type(func()))
print(type(gen()))
```

如上，生成器 `gen` 看起来和普通的函数没有太大区别。仅只是将 `return` 换成了 `yield`。用 `type()` 函数打印二者的类型也能发现，`func` 和 `gen` 都是函数。然而，二者的返回值的类型就不同了。`func()` 是一个 `int` 类型的对象；而 `gen()` 则是一个迭代器对象。

`yield` 表达式

如前所述，如果一个函数定义中包含 `yield` 表达式，那么该函数是一个生成器函数（而非普通函数）。实际上，`yield` 仅能用于定义生成器函数。

与普通函数不同，生成器函数被调用后，其函数体内的代码并不会立即执行，而是返回一个生成器（generator-iterator）。当返回的生成器调用成员方法时，相应的生成器函数中的代码才会执行。

```
def square():
    for x in range(4):
        yield x ** 2
square_gen = square()
for x in square_gen:
    print(x)
```

前面说到，`for` 循环会调用 `iter()` 函数，获取一个生成器；而后调用 `next()` 函数，将生成器中的下一个值赋值给 `x`；再执行循环体。因此，上述 `for` 循环基本等价于：

```
genitor = square_gen.__iter__()
while True:
    x = geniter.next()
    print(x)
```

注意到，`square` 是一个生成器函数；作为它的返回值，`square_gen` 已经是一个迭代器；迭代器的 `__iter__()` 返回它自己。因此 `geniter` 对应的生成器函数，即是 `square`。

每次执行到 `x = geniter.next()` 时，`square` 函数会从上一次暂停的位置开始，一直执行到下一个 `yield` 表达式，将 `yield` 关键字后的表达式列表返回给调用者，并再次暂停。注意，**每次从暂停恢复时，生成器函数的内部变量、指令指针、内部求值栈等内容和暂停时完全一致**。

生成器的方法

生成器有一些方法。调用这些方法可以控制对应的生成器函数；不过，若是生成器函数已在执行过程中，调用这些方法则会抛出 `ValueError` 异常。

- `generator.next()`：从上一次在 `yield` 表达式暂停的状态恢复，继续执行到下一次遇见 `yield` 表达式。当该方法被调用时，当前 `yield` 表达式的值为 `None`，下一个 `yield` 表达式中的表达式列表会被返回给该方法的调用者。若没有遇到 `yield` 表达式，生成器函数就已经退出，那么该方法会抛出 `StopIterator` 异常。
- `generator.send(value)`：和 `generator.next()` 类似，差别仅在与它会将当前 `yield` 表达式的值设置为 `value`。
- `generator.throw(type[, value[, traceback]])`：向生成器函数抛出一个类型为 `type` 值为 `value` 调用栈为 `traceback` 的异常，而后让生成器函数继续执行到下一个 `yield` 表达式。其余行为与 `generator.next()` 类似。
- `generator.close()`：告诉生成器函数，当前生成器作废不再使用。

举例和说明

如果你看不懂生成器函数

如果你还是不太能理解生成器函数，那么大致上你可以这样去理解。

- 在函数开始处，加入 `result = list()`；
- 将每个 `yield` 表达式 `yield expr` 替换为 `result.append(expr)`；
- 在函数末尾处，加入 `return result`。

关于「下一个」`yield` 表达式

介绍「生成器的方法」时，我们说当调用 `generator.next()` 时，生成器函数会从当前位置开始执行到下一个 `yield` 表达式。这里的「下一个」指的是执行逻辑的下一个。因此

```
def f123():
    yield 1
    yield 2
    yield 3

for item in f123():
    print(item)

def f13():
    yield 1
    while False:
        yield 2
    yield 3

for item in f13():
    print(item)
```

使用 `send()` 方法与生成器函数通信

```
def func():
    x = 1
    while True:
        y = (yield x)
        x += y

geniter = func()
geniter.next()
geniter.send(3)
geniter.send(10)
```

此处，生成器函数 `func` 用 `yield` 表达式，将处理好的 `x` 发送给生成器的调用者；与此同时，生成器的调用者通过 `send` 函数，将外部信息作为生成器函数内部的 `yield` 表达式的值，保存在 `y` 当中，并参与后续的处理。

这一特性是使用 `yield` 在 Python 中使用协程的基础。

`yield` 的好处

Python 的老用户应该会熟悉 Python 2 中的一个特性：内建函数 `range` 和 `xrange`。其中，`range` 函数返回的是一个列表，而 `xrange` 返回的是一个迭代器。

> 在 Python 3 中，`range` 相当于 Python 2 中的 `xrange`；而 Python 2 中的 `range` 可以用 `list(range())` 来实现。

Python 之所以要提供这样的解决方案，是因为在很多时候，我们只是需要逐个顺序访问容器内的元素。大多数时候，我们不需要「一口气获取容器内所有的元素」。比方说，顺序访问容器内的前 5 个元素，可以有两种做法：

- 获取容器内的所有元素，然后取出前 5 个；
- 从头开始，逐个迭代容器内的元素，迭代 5 个元素之后停止。

显而易见，如果容器内的元素数量非常多（比如有 `10 ** 8` 个），或者容器内的元素体积非常大，那么后一种方案能节省巨大的时间、空间开销。

现在假设，我们有一个函数，其产出（返回值）是一个列表。而若我们知道，调用者对该函数的返回值，只有逐个迭代这一种方式。那么，如果函数生产列表中的每一个元素都需要耗费非常多的时间，或者生成所有元素需要等待很长时间，则使用 `yield` 把函数变成一个生成器函数，每次只产生一个元素，就能节省很多开销了。