Python程序设计：三器语法

2023-2024

王斌辉 副教授

南开大学软件学院

“三器”语法

- 装饰器 Decorator
- 迭代器 Iterator
- 生成器 Generator

可迭代对象 iterable

- 每次能返回其一个成员的对象(An object capable of returning its members one at a time)，即实现了__iter__()或__getitem__()协议的对象
- Python 提供了两个通用迭代器对象：
  - 序列对象：list, str, tuple
  - 非序列对象： dict, file objects
- 可迭代对象可用于 for 循环，及其它需要序列的地方（如zip()、map() ...）
- 使用内置函数 iter()，或者__iter__()方法，可将可迭代对象转换为迭代器iterator

迭代器 iterator

- 用来表示一连串数据流的对象，称为迭代器
- 迭代器是实现迭代器协议的对象，它包含方法 __iter__() 和 __next__()
  - 迭代器的__iter__() 方法用来返回该迭代器对象自身，故迭代器必定是可迭代对象
  - 迭代器的 __next__() 方法（或将其传给内置函数 next()）将逐个返回数据流中的项，当没有数据可用时将引发 StopIteration 异常

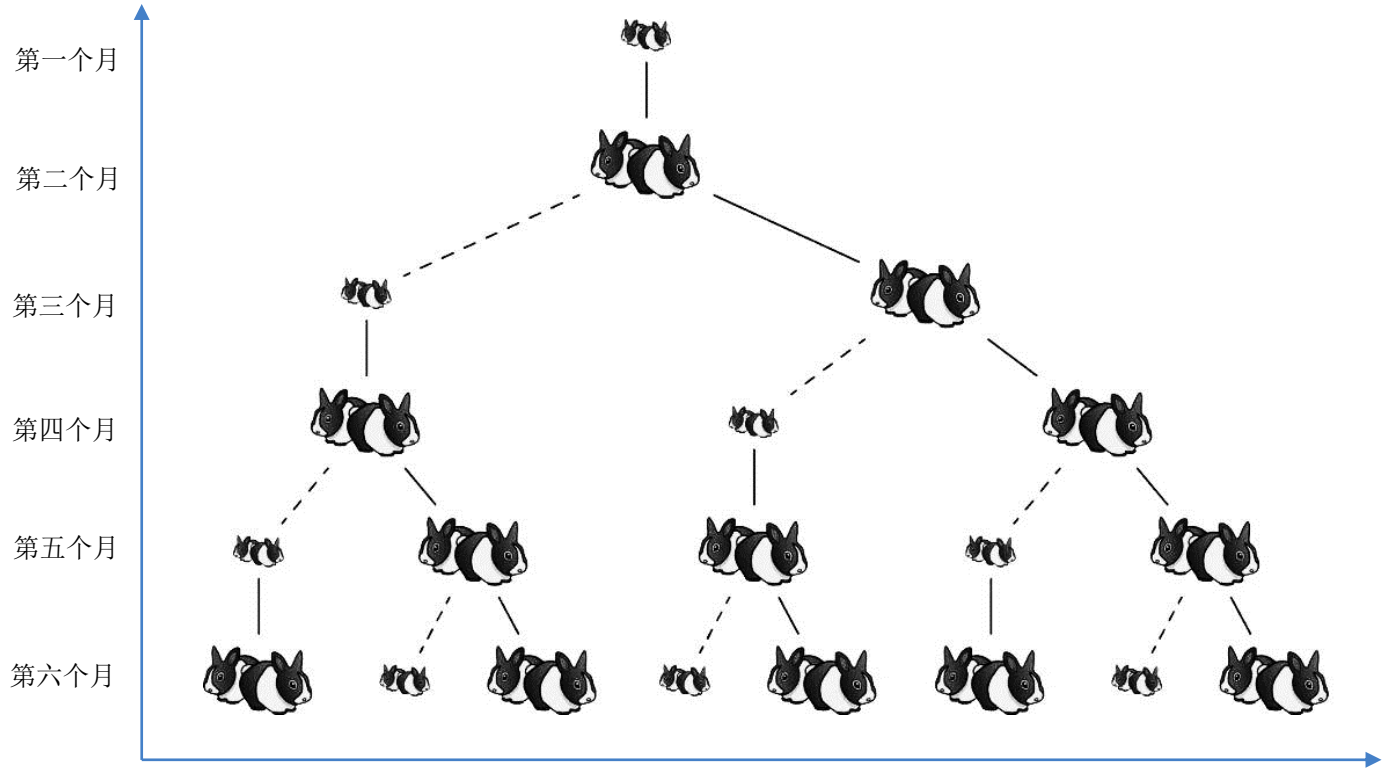
```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

迭代器 iterator

意大利数学家《算盘书》中兔子问题：假设一对初生兔子一个月到成熟期，一对成熟兔子每月生一对兔子，问6个月后会多少对兔子？




```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print(b)
        a, b = b, a + b
        n = n + 1
    return 'done'
```

当要扫描内存中放不下的大数据集时，如斐波拉契数列（Fibonacci），需要找到一种惰性获取数据项的方式，即按需一次获取一个数据项，而不是一次性收集全部数据

生成器 Generator

- 生成器表达式generator expression

```
[i * 2 for i in range(10) if i % 2 == 0]

{i * 2 for i in "abcd"}

{k: v for k, v in zip(("one", "two", "three"), (1, 2, 3))}

(i * i for i in range(10))
```

生成器 Generator

```
def func():
    for x in range(6):
        yield x
```

- 生成器函数: 使用 yield 语句的函数或方法
  - 当生成器函数被调用时，它返回一个名为生成器的迭代器（generator iterator）
  - 每个 yield 会临时暂停处理，记住当前位置执行状态（包括局部变量和挂起的 try 语句），当该生成器迭代器恢复时，它会从离开位置继续执行，这与普通函数调用(重新开始)差别很大）
  - 当yield表达式是赋值语句右侧的唯一表达式时，括号可以省略，即yield 语句在语义上等同于 yield 表达式，但建议总是加上，如右侧代码

```
val = (yield i)
```

  - yield 表达式和语句仅在定义 generator 函数时使用，并且仅被用于生成器函数的函数体内部
  - Python3.8 规定：禁止在实现推导式和生成器表达式的隐式嵌套作用域中使用 yield 表达式

生成器 Generator

- 生成器函数与函数，yield 与 return 的区别

  - 任何包含了 yield 关键字的函数都是生成器函数
  - 生成器函数是一类用来简化编写迭代器工作的特殊函数
  - 普通的函数计算并返回一个值，而生成器返回一个能返回数据流的迭代器
  - 当函数到达 return 表达式时，局部变量会被销毁然后把表达式返回给调用者
  - yield 和 return 最大区别：程序执行到 yield 时，生成器的执行状态会挂起并保留局部变量，在下一次调用生成器 __next__() 方法的时候，函数会恢复执行

生成器 Generator

```
def add(n, i):
    return n+i


def test():
    for i in range(4):
        yield i


g = test()
for n in [1, 10]:
    g = (add(n, i) for i in g)
print(list(g))
```

```
[20, 21, 22, 23]
```

生成器案例

```
def flatten_list(nested):
    if isinstance(nested, list):
        for sublist in nested:
            for item in flatten_list(sublist):
                yield item

    else:
        yield nested


def main():
    raw_list = ["a", ["b", "c", ["d"]]]
    g = flatten_list(raw_list)
    print(next(g))
    print(next(g))
    print(next(g))
    print(next(g))
    print("flatten_list is: ", list(flatten_list(raw_list)))


main()
```

```
a
b
c
d
flatten_list is:  ['a', 'b', 'c', 'd']
```

生成器案例

```
ccc
uuu
ppp
a ***
yyy
xxx
ccc
ccc
uuu
ppp
ppp
b ---
```

```python
def flatten_list(nested):
    if isinstance(nested, list):
        for sublist in nested:
            print("ccc")
            for item in flatten_list(sublist):
                print("ppp")
                yield item
                print("yyy")
    else:
        print("uuu")
        yield nested
        print("xxx")


def main():
    raw_list = ["a", ["b", "c", ["d"]]]
    g = flatten_list(raw_list)
    print(next(g), "***")
    print(next(g), "---")


main()
```

生成器 Generator

生成器方法

- .__next__():
  - 开始一个生成器函数的执行或从上次执行 yield 表达式的位置恢复执行
  - 当生成器函数通过 __next__() 方法恢复执行时，当前 yield 表达式总是取值为 None
  - 随后会继续执行到下一个 yield 表达式，生成器将再次挂起，而 expression_list 的值会被返回给 __next__() 的调用方
  - 若生成器没有产生下一个值就退出，则将引发 StopIteration 异常
- .send(value):
  - 恢复执行并向生成器函数“发送”一个值value，其将成为当前 yield 表达式的结果
  - 当调用 send() 来启动生成器时，它必须以 None 作为调用参数，因为这时没有可以接收值的 yield 表达式：即. **__next__() 方法相当于 .send(None)**

生成器 Generator

生成器方法

- .throw(value)
- .throw(type[, value[, traceback]])
  - 在生成器暂停的位置引发一个异常，并返回该生成器函数所产生的下一个值
  - 若生成器没有产生下一个值就退出，则将引发 StopIteration 异常
  - 若生成器函数没有捕获传入的异常，或是引发了另一个异常，则该异常会被传播给调用方
  - The *type* argument should be an exception class, and *value* should be an exception instance
- .close()：
  - 在生成器函数暂停的位置引发 GeneratorExit
  - 若生成器函数正常退出、关闭或引发 GeneratorExit (由于未捕获该异常) 则关闭并返回其调用者；若生成器产生了一个值，关闭会引发 RuntimeError；若生成器引发任何其他异常，它会被传播给调用者；若生成器已经由于异常或正常退出则 close() 不会做任何事

生成器

```
def echo(value=None):
    print("Execution starts when 'next()' is called for the first time.")
    try:
        while True:
            try:
                value = (yield value)
                print("*****")
            except Exception as e:
                value = e
    finally:
        print("Don't forget to clean up when 'close()' is called.")
```

```
generator = echo(1)
print(next(generator))
print(next(generator))
print(generator.send(2))
print(generator.throw(TypeError, "spam"))
generator.close()
```

```
Execution starts when 'next()' is called for the first time.
1
*****
None
*****
2
spam
Don't forget to clean up when 'close()' is called.
```