

Python生成器

讲师：Wayne

从业十余载，漫漫求知路

生成器***

□ 生成器generator

- 生成器指的是生成器对象，可以由生成器表达式得到，也可以使用yield关键字得到一个生成器函数，调用这个函数得到一个生成器对象

□ 生成器函数

- 函数体中包含yield语句的函数，就是生成器函数，调用后返回生成器对象
- 生成器对象，是一个可迭代对象，是一个迭代器
- 生成器对象，是延迟计算、惰性求值的

生成器

□ 举例

```
def inc():
    for i in range(5):
        yield i
print(type(inc))
print(type(inc()))
x = inc()
print(type(x))
print(next(x))
for m in x:
    print(m, '*')
for m in x:
    print(m, '**')
```

□ 举例

```
y = (i for i in range(5))
print(type(y))
print(next(y))
print(next(y))
```

□ 普通的函数调用fn()，函数会立即执行完毕，但是生成器函数可以使用next函数多次执行

□ 生成器函数等价于生成器表达式，只不过生成器函数可以写得更加的复杂

生成器

□ 举例

```
def gen():  
    print('line 1')  
    yield 1  
    print('line 2')  
    yield 2  
    print('line 3')  
    return 3  
    yield 4
```

```
next(gen()) # line 1  
next(gen()) # line 1  
g = gen()  
print(next(g)) # line 1  
print(next(g)) # line 2  
print(next(g)) # StopIteration  
print(next(g, 'End')) # 没有元素给个缺省值
```

- 在生成器函数中，使用多个yield语句，执行一次后会暂停执行，把yield表达式的值返回
- 再次执行会执行到下一个yield语句
- return语句依然可以终止函数运行，但return语句的返回值不能被获取到
- return会导致当前函数返回，无法继续执行，也无法继续获取下一个值，抛出StopIteration异常
- 如果函数没有显式的return语句，如果生成器函数执行到结尾（相当于执行了return None），一样会抛出StopIteration异常

生成器

□ 生成器函数

- 包含yield语句的生成器函数调用生成 生成器对象 的时候，**生成器函数的函数体不会立即执行**
- next(generator) 会从函数的当前位置向后执行到之后碰到的第一个yield语句，会弹出值，并暂停函数执行
- 再次调用next函数，和上一条一样的处理过程
- 继续调用next函数，生成器函数如果结束执行了（显式或隐式调用了return语句），会抛出StopIteration异常

生成器应用

□ 无限循环

```
def counter():
```

```
    i = 0
```

```
    while True:
```

```
        i += 1
```

```
        yield i
```

```
def inc(c):
```

```
    return next(c)
```

```
c = counter()
```

```
print(inc(c))
```

```
print(inc(c))
```

```
def counter():
```

```
    i = 0
```

```
    while True:
```

```
        i += 1
```

```
        yield i
```

```
def inc():
```

```
    c = counter()
```

```
    return next(c)
```

```
print(inc()) # 是什么
```

```
print(inc()) # 是什么
```

```
print(inc()) # 是什么
```

生成器应用

□ 计数器

```
def inc():  
    def counter():  
        i = 0  
        while True:  
            i += 1  
            yield i  
    c = counter()  
    return lambda : next(c)  
  
foo = inc()  
print(foo())  
print(foo())
```

□ lambda表达式是匿名函数

□ return返回的是一个匿名函数

□ 等价于下面的代码

```
def inc():  
    def counter():  
        i = 0  
        while True:  
            i += 1  
            yield i  
  
    c = counter()  
  
    def _inc():  
        return next(c)  
    return _inc  
  
foo = inc()  
print(foo())  
print(foo())  
print(foo())
```

生成器应用

▣ 处理递归问题

```
def fib():  
    x = 0  
    y = 1  
    while True:  
        yield y  
        x, y = y, x+y
```

```
foo = fib()  
for _ in range(5):  
    print(next(foo))
```

```
for _ in range(100):  
    next(foo)  
    print(next(foo))
```


生成器应用

□ 协程coroutine

- 生成器的高级用法
- 比进程、线程轻量级
- 是在用户空间调度函数的一种实现
- Python3 asyncio就是协程实现，已经加入到标准库
- Python3.5 使用async、await关键字直接原生支持协程
- 协程调度器实现思路
 - 有2个生成器A、B
 - next(A)后，A执行到了yield语句暂停，然后去执行next(B)，B执行到yield语句也暂停，然后再再次调用next(A)，再调用next(B)在，周而复始，就实现了调度的效果
 - 可以引入调度的策略来实现切换的方式
- 协程是一种非抢占式调度

yield from

□ 举例

```
def inc():
```

```
    for x in range(1000):  
        yield x
```

```
foo = inc()
```

```
print(next(foo))
```

```
print(next(foo))
```

```
print(next(foo))
```

□ 等价于下面的代码

```
def inc():
```

```
    yield from range(1000)
```

```
foo = inc()
```

```
print(next(foo))
```

```
print(next(foo))
```

```
print(next(foo))
```

yield from

- yield from是Python 3.3出现的新的语法
- yield from iterable 是 for item in iterable: yield item 形式的语法糖

- 从可迭代对象中一个个拿元素

```
def counter(n): # 生成器, 迭代器
    for x in range(n):
        yield x
```

```
def inc(n):
    yield from counter(n)
```

```
foo = inc(10)
print(next(foo))
print(next(foo))
```

谢谢

咨询热线 400-080-6560