匿名函数

匿名：隐藏名字，即没有名称

匿名函数：没有名字的函数。

函数没有名字该如何定义？函数没有名字如何调用？

Lambda表达式

Python中，使用Lambda表达式构建匿名函数。

```
lambda x: x ** 2 # 定义
(lambda x: x ** 2)(4) # 调用

foo = lambda x,y: (x+y) ** 2 # 定义函数
foo(1, 2)
# 等价于
def foo(x,y):
    return (x+y) ** 2
```

- 使用lambda关键字定义匿名函数，格式为`lambda [参数列表]: 表达式`
- 参数列表不需要小括号。无参就不写参数
- 冒号用来分割参数列表和表达式部分
- 不需要使用return。表达式的值，就是匿名函数的返回值。表达式中不能出现等号
- lambda表达式（匿名函数）**只能写在一行上**，也称为单行函数

匿名函数往往用在为高阶函数传参时，使用lambda表达式，往往能简化代码

```
# 返回常量的函数
print((lambda :0)())
print((lambda x:100)(1))

# 加法匿名函数，带缺省值
print((lambda x, y=3: x + y)(5))
print((lambda x, y=3: x + y)(5, 6))
# keyword-only参数
print((lambda x, *, y=30: x + y)(5))
print((lambda x, *, y=30: x + y)(5, y=10))

# 可变参数
print((lambda *args: (x for x in args))(*range(5)))
print((lambda *args: [x+1 for x in args])(*range(5)))
print((lambda *args: {x%2 for x in args})(*range(5)))
```

应用

```
# 需求，构建一个字典，所有key对应的值是一个列表，创建新的kv对的值也是空列表
d = {c:[] for c in 'abcde'}
d['a'].append(10) # {'a': [10], 'b': [], 'c': [], 'd': [], 'e': []}
d['f'] = [] # 新增key，value是空列表
d['f'].append(20)

```

```python
# defaultdict
from collections import defaultdict
d = defaultdict(list) # lambda : list()
d['a'].append('10') # d['a'] = list()
d['f'].append('20')

# sorted
x = ['a', 1, 'b', 20, 'c', 32]
print(sorted(x, key=str))
# 如果按照数字排序怎么做？
x = ['a', 1, 'b', 20, 'c', 32]
print(sorted(x, key=lambda x: x if isinstance(x, int) else int(x, 16)))
```

生成器函数

Python中有2种方式构造生成器对象：

1. 生成器表达式
2. 生成器函数
    - 函数体代码中包含yield语句的函数
    - 与普通函数调用不同，生成器函数调用返回的是生成器对象

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


def inc():
    for i in range(5):
        yield i

print(type(inc))
print(type(inc())) # 生成器函数一定要调用，返回生成器对象

g = inc() # 返回新的生成器对象
print(next(g))
for x in g:
    print(x)
print('--------------------')
for x in g: # 还能迭代出元素吗？
    print(x)
```

普通函数调用，函数会立即执行直到执行完毕。

生成器函数调用，并不会立即执行函数体，而是返回一个生成器对象，需要使用next函数来驱动这个生成器对象，或者使用循环来驱动。

生成器表达式和生成器函数都可以得到生成器对象，只不过生成器函数可以写更加复杂的逻辑。

执行过程

```
def gen():
    print(1)
    yield 2
    print(3)
    yield 4
    print(5)
    return 6
    yield 7

print(next(gen())) # 看到什么
print(next(gen())) # 看到什么
g = gen()
print(next(g))
print(next(g))
print(next(g)) # return的值可以拿到吗？
print(next(g, 'End')) # 没有元素不想抛异常，给个缺省值
```

- 在生成器函数中，可以多次yield，每执行一次yield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语句又会暂停执行
- 函数返回
    - 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

生成器函数

- 包含yield语句的生成器函数调用后，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

应用

1、无限循环

```
def counter():
    i = 0
    while True:
        i += 1
        yield i

c = counter()
print(next(c)) # 打印什么
print(next(c)) # 打印什么
print(next(c)) # 打印什么
```

2、计数器

```
def inc():
    def counter():
        i = 0
        while True:
            i += 1
            yield i

    c = counter()
    return next(c)

print(inc()) # 打印什么?
print(inc()) # 打印什么?
print(inc()) # 打印什么?为什么?如何修改
```

修改上例

```
def inc():
    def counter():
        i = 0
        while True:
            i += 1
            yield i

    c = counter()
    def inner():
        return next(c)
    return inner # return lambda : next(c)

foo = inc()
print(foo()) # 打印什么?
print(foo()) # 打印什么?
print(foo()) # 打印什么?为什么?
```

代码中的inner函数可以由lambda表达式替代。

3、斐波那契数列

```
def fib():
    a = 0
    b = 1
    while True:
        yield b
        a, b = b, a + b

f = fib()
for i in range(1, 102):
    print(i, next(f))
```

4、协程Coroutine

- 生成器的高级用法
- 它比进程、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语法

从Python 3.3开始增加了yield from语法，使得`yield from iterable`等价于`for item in iterable: yield item`。

yield from就是一种简化语法的语法糖。

```
def inc():
    for x in range(1000):
        yield x
# 使用yield from 简化
def inc():
    yield from range(1000) # 注意这个函数出现了yield，也是生成器函数

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

本质上yield from的意思就是，从from后面的可迭代对象中拿元素一个个yield出去。

作业

- 编写一个函数，能够实现内建函数map的功能
  - 函数签名`def mymap(func, iterable, /)`