

# Python Talk 9



这个指引文档在 知识共享 署名-相同方式共享 3.0协议之条款下提供

This guidance is available under the Creative Commons Attribution-ShareAlike License

# 函数式编程

函数式编程或称函数程序设计，又称泛函编程，是一种编程典范，它将电脑运算视为数学上的函数计算，并且避免使用程序状态以及易变物件。函数程式语言最重要的基础是 $\lambda$ 演算

（**lambda calculus**）。而且 $\lambda$ 演算的函数可以接受函数当作输入（引数）和输出（传出值）。

比起指令式编程，函数式编程更加强调程序执行的结果而非执行的过程，倡导利用若干简单的执行单元让计算结果不断渐进，逐层推导复杂的运算，而不是设计一个复杂的执行过程。

（以上内容来自维基百科）



# Python Build-in Functions

多 → 一

- ▶ `all`
- ▶ `any`
- ▶ `len`
- ▶ `min`
- ▶ `max`
- ▶ `sum`

多 → 多

- ▶ `enumerate`
- ▶ `filter`
- ▶ `map`
- ▶ `reversed`
- ▶ `sorted`
- ▶ `zip`

其他

- ▶ `range`
- ▶ `slice`
- ▶ **`yield`**



## all, any, len

`all` # 是否所有元素都为真

`all([1, 2, 3])` # True

`all([1, 2, ''])` # False

`any` # 是否有任何元素为真

`any([0, ''])` # False

`any([0, 1])` # True

`len` # 复杂数据类型的长度

`len([0, 'a', 9])` # 3



max, min, sum

max # 最大值

max([1, 2, 3]) # 3

min # 最小值

min([1, 2, 3]) # 1

sum # 总和

sum([1, 2, 3]) # 6



sorted, reversed

```
sorted                                # 排序元素  
k = lambda x: x[-1] # 根据最后一个元素排序  
sorted([(9, 3), (4, ), (2, 5)], key=k)  
  
reversed                             # 反转元素  
reversed([(9, 3), (4, ), (2, 5)])
```



# lambda表达式

**lambda** 参数列表：函数返回值

```
pow = lambda x, y: x ** y
```

```
f = lambda x: x ** 2 + 3 * x + 1
```

- ▶ 函数 `f` 在数学上相当于  $f(x) = x^2 + 3x + 1$
- ▶ 优点：代码简洁
- ▶ 缺点：无法进行复杂运算
- ▶ 提示：不要滥用**lambda**表达式，否则别人会很难理解你的代码



## reversed实践

```
>>> reversed([1, 2, 3])
<list_reverseiterator object at 0x123456789AB>
>>> # 这时返回的是一个iter的生成器，需要手动转换格式
>>> # 一般用list函数转换格式
>>> # 在特殊情况下可以用next
>>> list(reversed([1, 2, 3]))
[3, 2, 1]
>>>
```





# enumerate

- ▶ `enumerate` 插入序号
- ▶ 输入

```
['a', 'b', 'c']
```

- ▶ 输出

```
[  
    (0, 'a'),  
    (1, 'b'),  
    (2, 'c'),  
]
```

- ▶ 例：带序号打印列表

```
for i, j in enumerate(['a', 'b', 'c']):  
    print(i, j)
```



# zip

## ► zip可以拼接列表

### ► 输入

```
[1, 2, 3]  
[6, 7, 8]
```

### ► 输出

```
[  
    (1, 6),  
    (2, 7),  
    (3, 8),  
]
```

## ► 例：同时打印两个列表

```
A = [1, 2]  
B = ['a', 'b']  
for i, j in zip(a, b) :  
    print(i)  
    print(j)
```



# map

- ▶ `map` 可以将列表的每个元素分别用同一个函数执行

- ▶ 输入

```
lambda x: x ** 2  
[1, 2, 3, 4]
```

- ▶ 输出

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

- ▶ 练习

```
a = range(100)
```

使用 `map`、`sum` 和 `lambda`，用一程序求 `a` 的立方和



# 立方和解法

非函数式

```
s = 0
for i in range(a) :
    s += i ** 3
print(s)
```

函数式

```
sum(map(lambda x: x**3, a))
```

展开后

```
sum(
    map(
        lambda x: x**3,
        a
    )
)
```



# filter

## ▶ filter可筛选元素

### ▶ 输入

```
lambda x: x > 4  
[3, 4, 5, 6, 7]
```

### ▶ 输出

```
[5, 6, 7]
```

## ▶ 练习

```
a = range(1000)
```

不重复地打印出a中所有三或七的倍数



# 打印倍数解法

非函数式

```
for i in a :  
    if i % 3 == 0 :  
        print(i)  
    if i % 7 == 0 :  
        print(i)
```

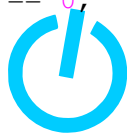
# 尝试找出以上程序的一处错误

函数式

```
filter(lambda x: x % 3 == 0  
        or x % 7 == 0, a)
```

展开后

```
filter(  
    lambda x: x % 3 == 0 or  
              x % 7 == 0,  
    a  
)
```



# range

- ▶ `range`可以快速得到一个等差整数数列

```
>>> range(1, 10, 2)
```

```
range(1, 10, 2)
```

```
>>> list(range(1, 10, 2))
```

```
[1, 3, 5, 7, 9]
```

```
>>> list(range(5, 3, -1))    # 反向
```

```
[5, 4]
```



# slice

- ▶ `slice`即切片，和 `[a:b:c]` 相同

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> a[2: 7: 2]
```

```
[3, 5, 7]
```

```
>>> a[slice(2, 7, 2)]
```

```
[3, 5, 7]
```





# yield

- ▶ **yield**在函数的返回值中添加元素

```
def f(x):  
    for i in range(x, 2 * x) :  
        yield(i)
```

```
def g(x):                # 不用yield的写法  
    ans = []  
    for i in range(x, 2 * x) :  
        ans.append(i)  
    return ans
```

```
list(f(3))                # [3, 4, 5]  
g(3)                     # [3, 4, 5]
```



# 练习

- ▶ 以下程序可以干什么？

```
sum(filter(lambda x: x % 3 == 1, map(lambda  
    x: x ** 2, range(20))))
```

- ▶ 展开后

```
sum(  
    filter(  
        lambda x: x % 3 == 1,  
        map(  
            lambda x : x ** 2,  
            range(20)  
        )  
    )  
)
```

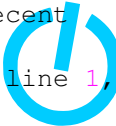


## iter和next

- ▶ `next` 可以将复杂数据类型转换为生成器
- ▶ `iter` 可以遍历一个函数式编程得到的生成器
- ▶ 也可以用 `__iter__` 和 `__next__`

```
>>> a = [1, 2]
>>> b = a.__iter__()
>>> b.__next__()
1
>>> b.__next__()
2
>>> b.__next__()
Traceback (most recent
    call last):
  File "<stdin>", line 1,
    in <module>
StopIteration
>>>
```

```
>>> a = [1, 2]
>>> b = iter(a)
>>> next(b)
1
>>> next(b)
2
>>> next(b)
Traceback (most recent
    call last):
  File "<stdin>", line 1,
    in <module>
StopIteration
>>>
```



# 感谢参加此次活动

