

Python与数据科学导论


C02



胡俊峰 2022/02/28
北京大学计算机学院



提纲

- 集合、字典 (dict)、容器类
 - 列表生成式、生成表达式 (generator)
 - 函数、生成器 (函数)
 - 函数闭包与函数的装饰器
 - 函数实例共享记忆缓存
- 

回顾一下上次课的核心内容：

- Python的程序框架：变量、表达式、流程控制 → 形式上的流程表达
- Python 的对象体系：对象、引用（引用计数）、对象内置方法→计算实现
- Python 的容器类型：list、tuple、set、dict
- 基于迭代器、生成器的循环控制
 - 容器、生成器表达式、生成器函数

set集合

- class set([iterable])
- **unordered** collection of distinct hashable objects

```

1 s1 = {1, 2, 3, 4, 5, 5, 1} ← 花括弧表示集合
2 s2 = set('abcaabbcc') #unique letters in s
3 print(s1, '\n', s2)

```

```

{1, 2, 3, 4, 5}
{'a', 'c', 'b'}

```

```

1 s1.add(7)
2 s1.remove(4)
3 s1

```

```
{1, 2, 3, 5, 7}
```

```

1 s1 = set([1, 2, 3, 3, 2]) ← 类型转换
2
3 s2 = set([2, 3, 4])
4
5 s1 & s2 # | & - ^ 并, 交, 差, 补

```

```
{2, 3}
```

容器类、可迭代对象、迭代器 与 迭代操作

- 容器 (container) 类 (如 : list , string , set , tuple , dict...) 都内建有一个操作函数来返回一个迭代器 (iterator) 。该迭代器被用来支持 for 循环语句 , 以及 in 操作。同时还支持 .next() 接口 , 返回下一个对象。

```
iter(object[, sentinel])
```

Return an **iterator** object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *object* must be a collection object which supports the **iterable** protocol (the **__iter__()** method), or it must support the sequence protocol (the **__getitem__()** method with integer arguments starting at 0). If it does not support either of those protocols, **TypeError** is raised. If the second argument, *sentinel*, is given, then *object* must be a callable object. The **iterator** created in this case will call *object* with no arguments for each call to its **__next__()** method; if the value returned is equal to *sentinel*, **StopIteration** will be raised, otherwise the value will be returned.

➤ Containers → List ; Dict (可迭代、可写、可直接存取 : Mutable Sequence)

Operation	Result
⁶ <code>s[i] = x</code>	item <i>i</i> of <i>s</i> is replaced by <i>x</i>
<code>s[i:j] = t</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> is replaced by the contents of the iterable <i>t</i>
<code>del s[i:j]</code>	same as <code>s[i:j] = []</code>
<code>s[i:j:k] = t</code>	the elements of <code>s[i:j:k]</code> are replaced by those of <i>t</i>
<code>del s[i:j:k]</code>	removes the elements of <code>s[i:j:k]</code> from the list
<code>s.append(x)</code>	same as <code>s[len(s):len(s)] = [x]</code>
<code>s.extend(t)</code> or <code>s += t</code>	for the most part the same as <code>s[len(s):len(s)] = t</code>
<code>s *= n</code>	updates <i>s</i> with its contents repeated <i>n</i> times
<code>s.count(x)</code>	return number of <i>i</i> 's for which <code>s[i] == x</code>
<code>s.index(x[, i[, j]])</code>	return smallest <i>k</i> such that <code>s[k] == x</code> and <code>i <= k < j</code>
<code>s.insert(i, x)</code>	same as <code>s[i:i] = [x]</code>
<code>s.pop([i])</code> ← 可以用来实现栈、队列	same as <code>x = s[i]; del s[i]; return x</code>
<code>s.remove(x)</code>	same as <code>del s[s.index(x)]</code>
<code>s.reverse()</code>	reverses the items of <i>s</i> in place
<code>s.sort([cmp[, key[, reverse]])</code>	sort the items of <i>s</i> in place

可迭代对象：

7

切片操作

Operation	Result
<code>x in s</code>	True if an item of <i>s</i> is equal to <i>x</i> , else False
<code>x not in s</code>	False if an item of <i>s</i> is equal to <i>x</i> , else True
<code>s + t</code>	the concatenation of <i>s</i> and <i>t</i>
<code>s * n, n * s</code>	equivalent to adding <i>s</i> to itself <i>n</i> times
<code>s[i]</code>	<i>i</i> th item of <i>s</i> , origin 0
<code>s[i:j]</code>	slice of <i>s</i> from <i>i</i> to <i>j</i>
<code>s[i:j:k]</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> with step <i>k</i>
<code>len(s)</code>	length of <i>s</i>
<code>min(s)</code>	smallest item of <i>s</i>
<code>max(s)</code>	largest item of <i>s</i>
<code>s.index(x)</code>	index of the first occurrence of <i>x</i> in <i>s</i>
<code>s.count(x)</code>	total number of occurrences of <i>x</i> in <i>s</i>

运算符优先级	描述
<code>:=</code>	赋值表达式
<code>lambda</code>	lambda 表达式
<code>if -- else</code>	条件表达式
<code>or</code>	布尔逻辑或 OR
<code>and</code>	布尔逻辑与 AND
<code>not x</code>	布尔逻辑非 NOT
<code>in, not in, is, is not, <, <=, >, >=, !=, ==</code>	比较运算，包括成员检测和标识号检测
<code> </code>	按位或 OR
<code>^</code>	按位异或 XOR
<code>&</code>	按位与 AND
<code><<, >></code>	移位
<code>+, -</code>	加和减
<code>*, @, /, //, %</code>	乘，矩阵乘，除，整除，取余 [5]
<code>+x, -x, ~x</code>	正，负，按位非 NOT
<code>**</code>	乘方 [6]

<https://docs.python.org/zh-cn/3/reference/expressions.html#boolean-operations>


循环与 enumerate（枚举） 函数:

```
1 animals = ['cat', 'dog', 'monkey']  
2 for idx, animal in enumerate(animals):  
3     print( animals[idx])
```

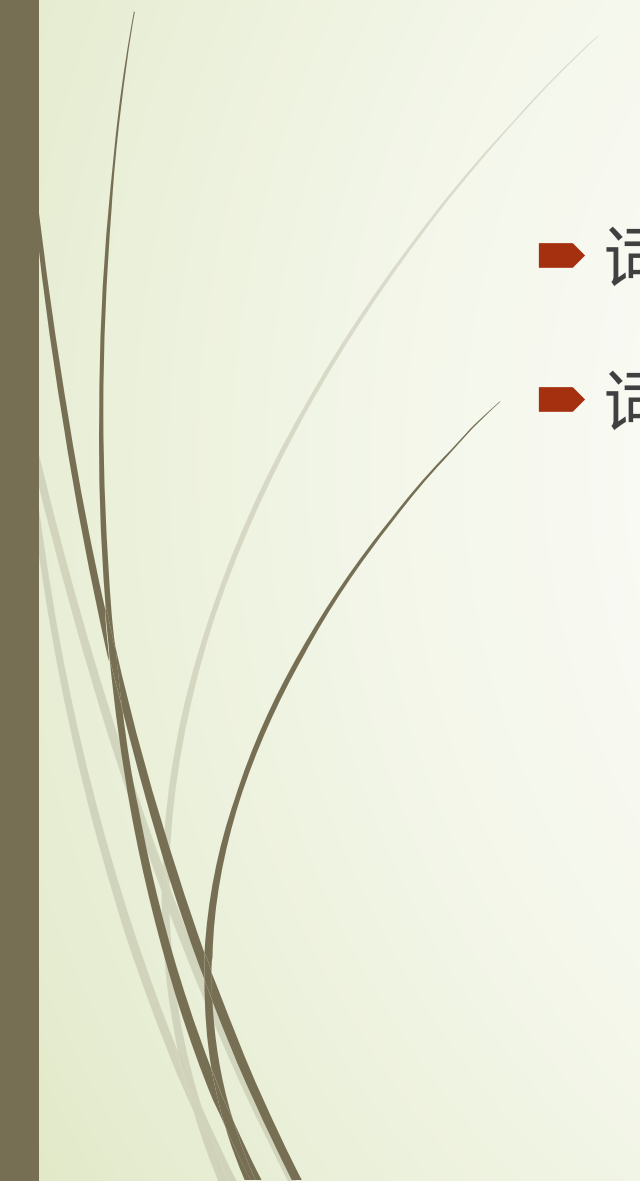
```
cat  
dog  
monkey
```

enumerate(*iterable*, *start*=0)

Return an **enumerate** object. *iterable* must be a sequence, an **iterator**, or some other object which supports iteration. The **__next__()** method of the iterator returned by **enumerate()** returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *iterable*.



词典 (dict)

- 词典的访问
 - 词典的遍历 (view to iter)
- 

Dict (key-value)

- 即为词典，按关键词查询，删除，修改。可以类比C++语言的map

11

dict根据key的内容来哈希，因此 key值不可变

```
1 d = {'Michael': 95, 'Bob': 75, 'l': 85}
2 print('Michael = ', d['Michael'])
3
4 d['Bob'] = 'fail'
5 print('Bob = ', d['Bob'])
```

关键词做下标，直接存取

按key值插入新元素或覆盖已有元素

```
Michael = 95
Bob = fail
```

```
1 d['Xiao Li'] = '90'
2 print ('Xiao Li' in d)
3
4 for k in d:
5     print (k, d[k])    # case sensitive
6
```

词典是sequence类型的么？

```
True
Michael 95
Bob fail
l 85
Laohu 80
Xiao Li 90
```

按关键字 (key) 直接访问

```
d = {'小李': 95, 'John': 75, 1: 35}  # 只要求是可hash对象  
print('小李: ', d['小李'])
```

```
d[1] = 'fail'  
print('1: ', d[1])
```

默认并非sequence类的

```
小李: 95  
1: fail
```

```
print(d.get(1))  
d.get('Lily', -1)  # -1, default None
```

```
fail
```

```
-1
```


增、删、查 键值对

```
d.pop((1))      # 删除
```

```
d['007'] = 80    # 增加
```

```
d
```

```
{'小李': 95, 'John': 90, '007': 80}
```

```
print(d.get('007'))    # 判断是否存在当前键值对
```

```
d.get('Lily', -1)      # -1, default None
```

```
80
```

```
-1
```

按关键字遍历

```
for k in d:                # 等效 for k in d.keys():  
    print (k, d[k])
```

```
d.keys()
```

```
小李 95
```


```
John 75
```

```
1 fail
```

↑
迭代器？

按内容 (values) 和条目 (items) 遍历

```
for val in d.values():    # values() 返回一个value的list (view)  
    print(val)
```



```
95  
75  
fail
```

```
for (key, val) in d.items():    # items() 返回一个dict_item的list  
    print(key, val)
```

```
小李 95  
John 75  
1 fail
```

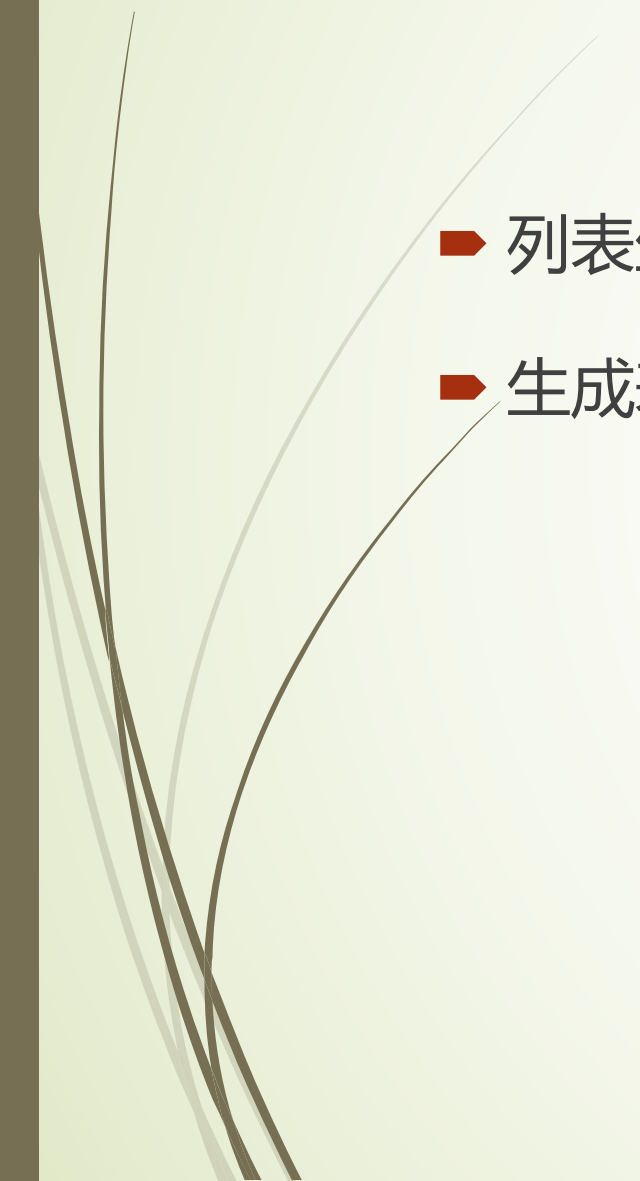
类型的相互转换

<code>print(int(10.6))</code>	<code># 类型转换函数只接受一个输入参数</code>
<code>print(set([1, 2, 3, 3, 2]))</code>	<code># set类型会自动去重</code>
<code>print(list('hello'))</code>	<code># 字符串也可以看作是一个容器类型</code>
<code>print(list({1:2, 'a':3}))</code>	<code># 返回词典的index list</code>
<code>print(dict([[1, 2], [3, 4]]))</code>	<code># 一种初始化词典的方法</code>

```
10
{1, 2, 3}
['h', 'e', 'l', 'l', 'o']
[1, 'a']
{1: 2, 3: 4}
```




列表生成式、生成表达式 (generator expression)

- ➡ 列表生成式
 - ➡ 生成表达式
- 

列表生成式

- 通过生成表达式定义（初始化）的列表

```
ls = [i*2 for i in range(5)]    # 通过迭代器生成列表
```

```
print(type(ls), ls)
```

```
<class 'list'> [0, 2, 4, 6, 8]
```

```
ls2 = [m + n  
       for m in 'ABC'          # 多个容器中取内容进行拼装  
       for n in '123'  
       ]  
ls2
```

```
['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']
```

列表生成式也可以生成集合等容器类对象

```
s1 = {5, 6}           # 初始化集合对象
s2 = {i*2 for i in range(5)} # 通过迭代器生成集合

print(type(s2))

s = s1 ^ s2           # | & - ^ 并, 交, 差, 补
for i in s:
    print(i)          # 集合元素值保证唯一
else:
    print('end of loop')
```

```
<class 'set'>
0
2
4
5
8
end of loop
```

列表生成式生成词典

```
d = { m+n : k
      for m in 'ABC'
      for n in '123'
      for k in 'xyz'
    }
```

```
print(type(d))
```

生成词典过程中会强制key是唯一不重复的，因此只会保留最后一条

```
for i in d:
    print(i, d[i])
```

```
<class 'dict'>
```

```
A1 z
```

```
A2 z
```

```
A3 z
```

```
B1 z
```

```
B2 z
```

```
B3 z
```

```
C1 z
```

```
C2 z
```

```
C3 z
```


生成表达式 (generator expression) → 生成器

```
c = (i*2 for i in range(5))    # 不是元组, 是生成器
```

可以用tuple()来转换

```
print(type(c), c)
```

```
<class 'generator'> <generator object <genexpr> at 0x000001736E0556C8>
```

```
g = (x * x for x in range(1, 11) if x% 2 == 0)    # 偶数的平方
```

```
print(next(g))
```

输出第一个元素

```
#ls3 = [i for i in g]    # 生成一个列表
```

```
ls3 = list(g)
```

list函数接受一个生成器

```
print(ls3)
```

4

[16, 36, 64, 100]



列表生成式基本语法：

List comprehension:

```
[expression for target1 in iterable1 [if condition1]
    for target1 in iterable2 [if condition2]
    for target1 in iterable3 [if condition3]
    for target1 in iterable4 [if condition4]
    .....
    for targetN in iterableN [if conditionN]]
```

```
d = { m+n : k
      for m in 'ABC'
      for n in '123'
      for k in 'xyz' if k < 'y'
    }

print(d)
```

```
{ 'A1' : 'x', 'A2' : 'x', 'A3' : 'x', 'B1' : 'x', 'B2' : 'x', 'B3' : 'x', 'C1' : 'x', 'C2' : 'x', 'C3' : 'x' }
```

zip() 函数用于将可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的列表。

```
1 str = [[i,j] for i,j in zip('abc','bcd')]  
2 print (str)
```

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

```
A = [[1, 2, 3], [3, 4, 5], [6, 7, 8]]  
print (A)
```

```
B = [x for x in zip(*A)]  
print(B)
```

* 算符为序列解包运算分离出向量
zip再把向量中对应位置元素打包成元组

```
C = [sum(i) for i in B]  
print(C)
```

实现对A的按列求和

```
[[1, 2, 3], [3, 4, 5], [6, 7, 8]]  
[(1, 3, 6), (2, 4, 7), (3, 5, 8)]  
[10, 13, 16]
```

Zip函数 配和 解包运算符 * 实现矩阵转置

总结一下前面的内容：

- 词典不是顺序类型但可以通过生成序列视图来变成可迭代对象
- 列表生成式本质上是生成表达式内容的实例化。等价于list(生成器)
 - 生成表达式式直接实例化就是生成器
- 列表生成式可以嵌套、组合，用来实现批量数据的筛选、重组、拆分、变换等加工



Python的函数

- Python函数的定义与参数传递
- 迭代器函数
- 函数嵌套、函数变量的作用域、闭包
- 函数装饰器

Python的函数定义：def 函数名(参数列表)：
程序块
return

```
1  def fib(max):  
2  
3      n, a, b = 0, 0, 1  
4  
5      while n < max:  
6  
7          print(b, end = ' —> ')  
8  
9          a, b = b, a + b  
10  
11         n = n + 1  
12  
13     return 'done'  
14  
15 fib(5)
```

1 —> 1 —> 2 —> 3 —> 5 —>

'done'

参数传对象的引用：

27

```
def demo(num1, num2, num_list): # 参数 传引用
```

```
    print("函数内部")
```

```
    # 赋值语句
```

```
    num1, num2 = num2, num1 # 分别指向新对象
```

```
    print("num2的id= ", id(num2))
```

```
    num_list[2] = 10 # 修改外部对象
```

```
    num_list = [1, 2, 3, 4] # 赋值语句导致指向新的对象
```

```
    print(num1, num2)
```

```
    print(num_list)
```

```
a = 99
```

```
b = 101
```

```
lst = [4, 5, 6]
```

```
print("a的id = ", id(a))
```

```
demo(a, b, lst)
```

```
print('函数外部')
```

```
print(a, b)
```

```
print(lst)
```

```
a的id = 94518506617216
```

```
函数内部
```

```
→ num2的id= 94518506617216
```

```
101 99
```

```
[1, 2, 3, 4]
```

```
函数外部
```

```
99 101
```

```
[4, 5, 10]
```

函数返回值也是引用：

28

```
def demo(num1, num2, num_list): # 参数 传引用
```

```
    print("函数内部")
```

```
    # 赋值语句
```

```
    num1, num2 = num2, num1 # 分别指向新对象
```

```
    num_list[2] = 10 # 修改外部对象
```

```
    num_list = [1, 2, 3, 4] # 赋值语句导致指向新的对象
```

外部的引用及数据对象并不影响

```
    print(num1, num2)
```

```
    print(num_list)
```

```
    return num_list, num1 # 返回多个值（都是引用）
```

```
a = 99
```

```
b = 101
```

```
lst = [4, 5, 6]
```

```
print("b的id = ", id(b))
```

```
lst, a = demo(a, b, lst) # lst指向新的列表 a指向新的对象
```

```
print('函数外部')
```

```
print(a, b)
```

```
print("a的新id = ", id(a))
```

```
print(lst)
```

```
b的id = 94518506617280
```

```
函数内部
```

```
101 99
```

```
[1, 2, 3, 4]
```

```
函数外部
```

```
101 101
```

```
a的新id = 94518506617280
```

```
[1, 2, 3, 4]
```

python函数的可变参数列表

**args and **kwargs*

- 主要用于函数定义。支持将不定数量的参数传递给一个函数。
- *args 是用来发送一个非键值对的可变数量的参数列表给当前函数。 **kwargs是用来接受一个键值对的可变数量的参数列表给当前函数

Python函数的参数传递格式：

def fun (arg , ... , *args , **kwargs) :

- ➡ 单个参数的列表
- ➡ *args 列表形式的参数向量
- ➡ **kwargs 键值对的可变数量的参数列表

➤ 非键值对的可变数量的参数列表

31

```
1  def test_var_args(f_arg, *argv):    #起始参数（引用），后继序列
2
3      print("first normal arg:", f_arg)
4
5      for arg in argv:
6
7          print("another arg through *argv:", arg)
8
9
10 test_var_args('2018', 'python', 'eggs', 'test')
```

```
first normal arg: 2018
another arg through *argv: python
another arg through *argv: eggs
another arg through *argv: test
```

```
1 def greet_me(**kwargs):  # 参数-值 列表
2
3     for key, value in kwargs.items():
4
5         print("{0} = {1}".format(key, value))
6
7 greet_me(name="python", room = '108', place = ['science', 'building' ])
```

```
name = python
```

```
room = 108
```

```
place = ['science', 'building']
```

混合参数列表示例：

```
def test_args(arg1, arg2, *argv, **argd):    #起始参数（引用），后继序列

    print("参数2:", arg2)

    for arg in argv:
        print("参数列表", arg)

    for k, v in argd.items():
        print(k, ':', v)

test_args(2021, 'python', 'data science', 'deep learning',
          classroom = '二教207', time = 'Mon 5-6' )
```

参数2: python

参数列表 data science

参数列表 deep learning

classroom : 二教207

time : Mon 5-6

函数应用举例：递归函数实现快排

实现快速排序算法

```
def quicksort(arr):  
    if len(arr) <= 1:          #递归返回条件  
        return arr  
    pivot = arr[len(arr) // 2]  
    left = [x for x in arr if x < pivot]      #生成小于pivot的左列表  
    middle = [x for x in arr if x == pivot]   #middle列表, 支持有重复值  
    right = [x for x in arr if x > pivot]     #生成大于pivot的右列表  
    return quicksort(left) + middle + quicksort(right) #返回一个新的列表  
  
print (quicksort([3, 6, 8, 10, 1, 2, 1]))
```


递归函数+lambda表达式实现快排

```
def qsort(a):  
    if len(a) <= 1:  
        return a  
    else:  
        return (qsort(list(filter(lambda x: x <= a[0], a[1:]))) # a[0]是哨兵  
                + [a[0]]  
                + qsort(list(filter(lambda x: x > a[0], a[1:]))) # 多加了一层
```

```
print (qsort([3, 6, 8, 10, 1, 2, 1]))
```

```
[1, 1, 2, 3, 6, 8, 10]
```

Sorted()函数：

接受一个可迭代对象，返回一个排好序的list

```
In [10]: ls = [5, 2, 3, 1, 4]
```

```
new_ls = sorted(ls)  
ls
```

```
Out[10]: [5, 2, 3, 1, 4]
```

```
In [11]: sorted?
```

Signature: `sorted(iterable, /, *, key=None, reverse=False)`

Docstring:

Return a new list containing all items from the iterable in ascending order.

A custom key function can be supplied to customize the sort order, and the reverse flag can be set to request the result in descending order.

Type: `builtin_function_or_method`

对象自带的Sort方法：

```
In [13]: ls.sort?  
ls.
```

- append
- clear
- copy
- count
- extend
- index
- insert
- pop
- remove
- reverse

← Tab键进行代码提示和补全

Signature: `ls.sort(*, key=None, reverse=False)`

Docstring: Stable sort *IN PLACE*.

Type: builtin_function_or_method

接受函数作参数的函数 —— 高阶函数

- map()函数接收两个参数，一个是函数，一个是Iterable的对象，map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。

```
1  def f(x):  
2  
3      return x * x  
4  
5  r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])  
6  
7  list(r)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

reduce()函数 —— 目前已经不在标准库中

- reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算。
- $\text{reduce}(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)$

```
1  from functools import reduce
2
3  def fn(x, y):
4
5      return x * 10 + abs(y)
6
7  reduce(fn, [1, 3, 5, -7, 9])
```

13579

filter()函数

- 过滤序列，filter()接收一个函数和一个序列。把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

```
1 def is_odd(n):  
2  
3     return n % 2 == 1  
4  
5 list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15]))
```

[1, 5, 9, 15]

```
1 fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]  
2  
3 odd_numbers = list(filter(lambda x: x % 2, fibonacci))  
4  
5 odd_numbers
```

[1, 1, 3, 5, 13, 21, 55]

生成器函数：generator function

—— **yield**关键字

- 带有 yield 的函数不再是一个普通函数，Python 解释器会将其视为一个 generator，调用 fab(5) 不会执行 fab 函数，而是返回一个 iterable 对象。每次next调用，执行到yield位置函数返回next元素

➡ 函数实现一个 generator

42

```
1  def fib(max = 5): ← 设置缺省参数值
2
3      n, a, b = 0, 0, 1
4
5      while n < max:
6
7          yield b    #此时返回下一个序列元素
8
9          a, b = b, a + b
10
11         n = n + 1
12
13 f = fib()
14 f
```

<generator object fib at 0x0000019DEB305138>

```
1  for i in f:
2
43     print(i, out = ' ', '')
```

- 也可以手动调用全局的 `next()` 方法（因为 `fab(5)` 是一个 generator 对象），这样我们就可以更清楚地看到 `fab` 的执行流程：

```
1  f = fib(5)
2
3  next(f)
```

1

```
1  next(f)
```

1

```
1  next(f)
```

2

➡ 生成器函数的定义 (generator) :

```
1  def fib(max = 5): ← 设置缺省参数值
2
3      n, a, b = 0, 0, 1
4
5      while n < max:
6
7          yield b    #此时返回下一个序列元素
8
9          a, b = b, a + b
10
11         n = n + 1
12
13  f = fib()
14  f
```

带有 yield 的函数Python 解释器会将其视为一个 generator , 调用 fab(5) 不会执行 fab 函数 , 而是返回一个 iterable 对象。每次next调用 , 执行到yield位置函数返回next元素

<generator object fib at 0x00000019DEB305138>

生成器函数：派生多个生成器实例

```
f1 = fib(5)  # 生成器1
```

```
f2 = fib(9)
```

```
print(next(f1), next(f1), next(f1), next(f2))
```

```
1 1 2 1
```

```
print([i for i in f1])
```

```
print([i for i in f2])  # f1, f2 相互独立
```

```
[3, 5]
```

```
[1, 2, 3, 5, 8, 13, 21, 34]
```



嵌套函数、函数闭包、装饰器

- 核心知识点：
 - 变量的作用域
 - 函数的私有运行环境
 - 函数的功能切片与组装

嵌套函数的定义：

```
: def fo(par=0): ← 缺省参数  
    i = par + 1  
    def fi(x = 0):      # 外部不可见  
        return x + i  
    return fi()  
print(fo(10))
```

变量的作用域：local-nonlocal-global

nonlocal & global

- python引用变量的顺序为：当前作用域局部变量->外层作用域变量->当前模块中的全局变量->python内置变量
- global关键字可以用在任何地方，包括最上层函数中和嵌套函数中，**即使之前未定义该变量**，global修饰后也可以直接使用
- nonlocal关键字只能用于嵌套函数中，并且外层函数中定义了相应的局部变量


```

1  def scope_test():
2      def do_local():
3          spam = "local spam No.1"  # 1号 ← 局部变量
4      def do_nonlocal():
5          nonlocal spam
6          spam = "nonlocal spam No.2 " # 2号
7      def do_global():
8          global spam
9          spam = "global spam No.3" #3号
10     spam = "test spam"           # 2号设置
11     do_local()
12     print("After local assignment:", spam)  # 打印 2号 spam
13     do_nonlocal()
14     print("After nonlocal assignment:", spam)
15     do_global()
16     print("After global assignment:", spam)
17     scope_test()
18     print("In global scope:", spam) # 注意缩进 ← 引用全局变量

```

外层变量

全局变量

局部变量

引用全局变量

```

After local assignment: test spam
After nonlocal assignment: nonlocal spam No.2
After global assignment: nonlocal spam No.2
In global scope: global spam No.3

```

函数闭包：内部定义了一个函数，然后把该函数作为返回值

```
def print_msg(msg):  
    def printer(st=''):# 定义了一个内部函数  
        print(st+msg) ←  
    return printer      # 返回该函数  
  
another = print_msg("老板，来一碗") # 接收到一个printer函数的实例  
theOther = print_msg("再来一碗")    # 另一个printer函数的实例  
enough = print_msg("待会还要开车呢") # 每个都是被外层函数定制的  
  
another()      # 不输入参数为默认参数值  
theOther()  
enough('别喝了') # 加入参数
```

老板，来一碗

再来一碗

别喝了 待会还要开车呢

函数闭包还可以实现类似
函数模板的功能

```
def inc(x):  
    return x + 1
```

```
def dec(x):  
    return x - 1
```

```
def make_operate_of(func): # 要求输入一个函数  
    def operate(x): # 要求输入一个参数  
        return func(x)  
    return operate
```

```
addone = make_operate_of(inc) # 输出一个指定的计算模式  
minusone = make_operate_of(dec)
```

```
print(addone(2))  
print(minusone(2))
```

函数装饰器：一般用于抽取共性操作作为功能切片，对一类函数进行包装

```
def decorator(func):  
    def dechouse():  
        print("木地板，", "吊顶", end = "") # end替换换行符  
        func()  
    return dechouse  
  
def house():  
    print("房子")  
  
def classroom():  
    print("教室")  
  
ordinary() # 原本的样子  
  
newhouse = decorator(house)  
newhouse()  
newclassroom = decorator(classroom)  
newclassroom()
```

房子
木地板， 吊顶房子
木地板， 吊顶教室

@ 号加装饰器函数名放在函数定义的前面，
表明定义了一个装饰后的函数

```
@decorator
def officeroom():
    print("办公室")

officeroom()
```

木地板， 吊顶办公室

```
def smart_divide(func): #对除法操作进行安全检查
    def inner(a,b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("除数不能为0")
            return

        return func(a, b)
    return inner

@smart_divide # 更加安全的除法
def divide(a,b):
    return a/b

divide(4, 3)
divide(4, 0)
```

I am going to divide 4 and 3
I am going to divide 4 and 0
除数不能为0

通过函数闭包对装饰器进行定制

```
def specify(req):  
    def decorator(func):  # 装饰器函数  
        def dechouse():  
            print( req + "木地板, ", "吊顶", end = "") # end 替换换行符  
            func()  
        return dechouse  
    return decorator  # 函数闭包, 返回定制的装饰器函数  
  
@specify(' 高档')  # 通过参数实现定制化的装饰器  
def house():  
    print("房子")  
  
@specify(' 普通')  # 通过参数实现定制化的装饰器  
def officeroom():  
    print("办公室")  
  
house()  
officeroom()
```

高档木地板, 吊顶房子
普通木地板, 吊顶办公室

装饰器可以嵌套

```
def star(func):  
    def inner(*args):  
        print("*" * 30)          # 重复30次  
        func(*args)  
        print("*" * 30)  
    return inner
```

```
def percent(func):  
    def inner(*args):  
        print("%" * 30)  
        func(*args)  
        print("%" * 30)  
    return inner
```

@star

@percent # 较下层对应的是更内层

```
def printer(msg):  
    print(msg)
```

```
printer("今天沙尘太大啦!!!")
```

```
*****  
%%%%%%%%%%%%%%%%%%%%%%%%  
今天沙尘太大啦!!!  
%%%%%%%%%%%%%%%%%%%%%%%%  
*****
```

```
# https://www.geeksforgeeks.org/memoization-using-decorators-in-python/
def memoize_factorial(f): # 输入被装饰的操作函数f
    memory = {}          # 所有被装饰的函数实例共享一个公共缓存（记忆化存储）

    def inner(num):
        if num not in memory: # 查询缓存
            memory[num] = f(num) # 递归调用操作函数f，返回结果进行缓存写入
            print('%d not in ' % (num), end='')
        return memory[num] # 如果缓存有内容，则直接返回结果，对递归函数进行优化

    return inner

@memoize_factorial
def facto(num):
    if num == 1:
        return 1
    else:
        return num * facto(num-1)

print(facto(4))
print(facto(5))
print(facto(4))
```

装饰器实现记忆化存储

```
1 not in 2 not in 3 not in 4 not in 24
5 not in 120
24
```



关于作业

- 课程网发布
- 本次作业截止时间为周四中午12点（周一讲评）
 - 时间可以安排过来最好能周二晚上提交，以方便助教批改
 - 在作业讲评中被引用的优质、有创意作业会有额外加分