# Python Cheat Sheet-6. recursion(递归)、lambda(Anonymous Function, 匿名函数)、generator(生成器)

?> PCS-6（    ）：6.1 递归；6.2 匿名函数（lambda）；6.3 生成器（generator）；6.4 itertools库

知识点

描述

代码段

运算结果

备注

**6.1** 递归

通过简单的案例查看每一个循环，满足条件代码块运行变量值和返回值的变化是理解递归最好的途径。递归的核心是调用自身，重复同一个代码逻辑过程，实现返回值满足某一变化特征的叠加；并且需要有结束条件，以结束递归，即结束调用自身的循环；同时，给结束条件一个返回值，为最后一次调用自身的返回值。

下述两个案例一个为递归求和，一个为递归阶乘，通常用此来解释递归函数的方法。

* 递归求和

第1个解释案例是递归求和，其过程如图。用具体的语句和值解释上一段话为：开始输入值（开始值）为lst=[1,2,3,4,5], 不满足结束条件if not lst，执行else语句块，返回值语句为return lst[0]+recursive\_sum(lst[1:])，将值代入语句（表达式），为1+recursive\_sum([2,3,4,5])，其中包含调用自身语句recursive\_sum(lst[1:]，因此从函数名定义行重新开始执行该函数；因为输入参数值[2,3,4,5]不满足结束条件，再次执行else语句块，将值代入语句（表达式），为2+recursive\_sum([3,4,5])，继续调用自身，此时的输入值为[3,4,5]；以次类推，不断迭代，直至执行到5+recursive\_sum([])时，可以发现调用递归的输入值为[]，满足结束条件，因此执行if not lst语句块，返回值为0，结束递归迭代。当结束递归后，将每一次的返回值代入上一次迭代的返回值，计算获得最终结果为1+2+3+4+5+0=15。

从上述递归过程的描述中，可以确定递归的基本逻辑结构，无外乎调用自身即返回值计算逻辑和该逻辑能够产生结束条件，因此具有递归属性的问题都可以此点切入设计代码。

def recursive\_sum(lst):  
 if not lst:  
 print("if not lst:{}".format(lst))  
 return 0  
 else:  
 print(lst[1:])  
 return lst[0]+recursive\_sum(lst[1:])  
   
lst=[1,2,3,4,5]   
lst\_sum=recursive\_sum(lst)  
print("--"\*30)  
print(lst\_sum)

[2, 3, 4, 5]  
[3, 4, 5]  
[4, 5]  
[5]  
[]  
if not lst:[]  
------------------------------------------------------------  
15

可以通过小的语句核实确定某些语句的运行结果，辅助理解程序。

print(not [])  
print(not [5])  
print([5][1:])

True  
False  
[]

返回值是需要代入到上次迭代调用自身的部分，因此只要返回值满足计算要求，可以根据需要返回任何值。下述代码将结束条件返回值配置为入1000时，计算结果为1015。

def recursive\_sum(lst):  
 if not lst:  
 print("if not lst:{}".format(lst))  
 return 1000  
 else:  
 print(lst[1:])  
 return lst[0]+recursive\_sum(lst[1:])  
   
lst=[1,2,3,4,5]   
lst\_sum=recursive\_sum(lst)  
print("--"\*30)  
print(lst\_sum)

[2, 3, 4, 5]  
[3, 4, 5]  
[4, 5]  
[5]  
[]  
if not lst:[]  
------------------------------------------------------------  
1015

* 递归阶乘

直接可以替换递归求和图中对应的值，来表达递归阶乘的过程。可以解释为：开始输入值（开始值）为6, 不满足结束条件if n==1，执行else语句块，返回值语句为return n\*recursive\_factorial(n-1)，将值代入语句（表达式），为6×recursive\_factorial(5)，其中包含调用自身语句recursive\_factorial(5)，因此从函数名定义行重新开始执行该函数；因为输入参数值5不满足结束条件，再次执行else语句块，将值代入语句（表达式），为5×recursive\_factorial(4)，继续调用自身，此时的输入值为4；以次类推，不断迭代，直至执行到2×recursive\_factorial(1)时，可以发现调用递归的输入值为1，满足结束条件，因此执行if n==1语句块，返回值为1，结束递归迭代。当结束递归后，将每一次的返回值代入上一次迭代的返回值，计算获得最终结果为6×5×4×3×2×1=720。

def recursive\_factorial(n):  
 if n==1:  
 return 1  
 else:  
 return n\*recursive\_factorial(n-1)  
  
fatorial\_result=recursive\_factorial(6)  
print(fatorial\_result)

720

* 用循环语句替换递归的方法

递归的方法可以用循环的方式解决，例如下述代码改写的递归求和及阶乘。虽然循环的方式更加易读，但是递归的方法似乎也符合python的宗旨，保持简单，除非它必须复杂！因此，究竟是使用递归还是循环，可以由代码的作者自己决定。

lst=[1,2,3,4,5]   
lst\_sum=0  
while lst:  
 lst\_sum+=lst[0]  
 lst=lst[1:]  
print(lst\_sum)  
  
n=6  
fatorial\_result=1  
while n:  
 if n==1:break  
 else:  
 fatorial\_result\*=n  
 n=n-1  
print(fatorial\_result)

15  
720

* 递归列表展平

下述递归嵌套列表展平的代码，虽然代码行数不多，但是相对递归求和和阶乘要复杂。这里需要注意两个问题，一个是存在有三个条件，if是结束条件。elif和else分别针对两种不同情况采取的措施。结束条件不是像递归求和和阶乘最后执行一次结束，是对elif和else下不断调用自身，最终达到各个分枝下满足结束条件的输入参数，逐个结束各个分枝；另一个是，返回值中存在return flatten\_lst(lst[0])+flatten\_lst(lst[1:])，为调用自身的函数求和，需要从左到右一次计算，先返回flatten\_lst(lst[0])部分，待这部分递归完毕后，在执行flatten\_lst(lst[1:])部分，而各自部分同样有满足elif isinstance(lst[0],list)条件的对象，因此还会产生调用自身两个函数相加的情况，依旧从左到右依次计算，至递归结束。具体的过程可以从图中观察，一个是正序过程，由k-i,j的方式标注执行顺序；粉色部分则是逆序逐步代入值的过程。比较清晰的表述了该递归的整个过程。

当很难通过代码直接理解算法时，必然需要通过print()，打印各个变化值查看整个过程。如果打印的对象比较多时，尤其类似递归这样要不断循环的算法，最好给出些辅助标识，定位每一轮次迭代，方便查看每次迭代对应变量的变化，从而找出规律，理解算法。

i=0  
j=0  
k=0  
def flatten\_lst(lst):  
 global k,i,j  
   
 print("#"\*30,'%d-%d-%d'%(k,i,j))  
 print("lst:",lst)  
 if lst==[]:  
 print('\_'\*30,'if')  
 k+=1  
 return lst  
   
 elif isinstance(lst[0],list):  
 print('\_'\*30,"elif")  
 print(lst[0],';',lst[1:])  
 i+=1  
 return flatten\_lst(lst[0])+flatten\_lst(lst[1:])  
   
 else:  
 print('\_'\*30,'else')  
 print(lst[:1],";",lst[1:])  
 j+=1  
 return lst[:1]+flatten\_lst(lst[1:])  
   
nested\_lst=[['A','B',['C','D'],'E'],[6,[7,8,[9]]]]   
flatten\_nested\_lst=flatten\_lst(nested\_lst)  
print("--"\*30)  
print(flatten\_nested\_lst)

############################## 0-0-0  
lst: [['A', 'B', ['C', 'D'], 'E'], [6, [7, 8, [9]]]]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ elif  
['A', 'B', ['C', 'D'], 'E'] ; [[6, [7, 8, [9]]]]  
############################## 0-1-0  
lst: ['A', 'B', ['C', 'D'], 'E']  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ else  
['A'] ; ['B', ['C', 'D'], 'E']  
############################## 0-1-1  
lst: ['B', ['C', 'D'], 'E']  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ else  
['B'] ; [['C', 'D'], 'E']  
############################## 0-1-2  
lst: [['C', 'D'], 'E']  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ elif  
['C', 'D'] ; ['E']  
############################## 0-2-2  
lst: ['C', 'D']  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ else  
['C'] ; ['D']  
############################## 0-2-3  
lst: ['D']  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ else  
['D'] ; []  
############################## 0-2-4  
lst: []  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ if  
############################## 1-2-4  
lst: ['E']  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ else  
['E'] ; []  
############################## 1-2-5  
lst: []  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ if  
############################## 2-2-5  
lst: [[6, [7, 8, [9]]]]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ elif  
[6, [7, 8, [9]]] ; []  
############################## 2-3-5  
lst: [6, [7, 8, [9]]]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ else  
[6] ; [[7, 8, [9]]]  
############################## 2-3-6  
lst: [[7, 8, [9]]]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ elif  
[7, 8, [9]] ; []  
############################## 2-4-6  
lst: [7, 8, [9]]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ else  
[7] ; [8, [9]]  
############################## 2-4-7  
lst: [8, [9]]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ else  
[8] ; [[9]]  
############################## 2-4-8  
lst: [[9]]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ elif  
[9] ; []  
############################## 2-5-8  
lst: [9]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ else  
[9] ; []  
############################## 2-5-9  
lst: []  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ if  
############################## 3-5-9  
lst: []  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ if  
############################## 4-5-9  
lst: []  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ if  
############################## 5-5-9  
lst: []  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ if  
------------------------------------------------------------  
['A', 'B', 'C', 'D', 'E', 6, 7, 8, 9]

**6.2** 匿名函数（lambda）

lambda的基本语法为lambda argument1,argument2,argument3,...,argumentN:expression using arguments，需要注意的是lambda使用的是表达式（expression）不是语句（statement），返回一个函数表达式，不需要定义函数名（因此称为匿名函数）。lambda的表达式中可以加入if条件语句，语法为lambda <arguments>:<return value if condition is True> if <condition> else <return value if condition is False>；如果包含类似elif的结构，则语法为lambda <args>:<return value> if <condition> else (return value if <condition> else <return value>)。

对于较为简单的函数定义，使用lambda为内联函数（inline function），可以简化代码，而def方法则会稍显繁琐。

def sum\_func(x,y,z):  
 return x+y+z  
lst=[1,2,3]  
print(sum\_func(\*lst))  
  
sum\_lambda=lambda x,y,z:x+y+z  
print(sum\_lambda(\*lst))  
  
print("-"\*60)  
comparison\_operation\_A=lambda x:True if x>10 else False #只有一个条件  
print(comparison\_operation\_A(20))  
  
CompOpera\_B=lambda x:True if (x>10 and x<20) else False #包含多个条件，需要括起  
print(CompOpera\_B(18))  
  
CompOpera\_C=lambda x:x>10 and x<20 #不使用if else的方式  
print(CompOpera\_C(18))  
  
print("-"\*60)   
CompOpera\_D=lambda x:1 if x<10 else (2 if x<20 else 0) #类似elif  
print(CompOpera\_D(7))  
print(CompOpera\_D(15))  
print(CompOpera\_D(30))

6  
6  
------------------------------------------------------------  
True  
True  
True  
------------------------------------------------------------  
1  
2  
0

* 嵌套的匿名函数

与嵌套函数类似。

nested\_sum=lambda x:lambda y:x+y  
sum\_instance=nested\_sum(10) #输入值对应参数x  
print(sum\_instance(20)) #输入值对应参数y  
print(sum\_instance(50))

30  
60

* lambda 与filter(), map(),reduce(),sorted

lambda经常配合其它以函数作为输入参数的函数，可以非常便捷的以inline function方式行内完成代码。通过help()非常方便的确定filter(), map(),reduce(),sorted输入参数包含使用函数作为参数的部分。

print(help(filter))  
print("+"\*60)   
print(help(map))  
print("+"\*60)   
from functools import reduce  
print(help(reduce))  
print("+"\*60)   
print(help(sorted))

Help on class filter in module builtins:  
  
class filter(object)  
 | filter(function or None, iterable) --> filter object  
 |   
 | Return an iterator yielding those items of iterable for which function(item)  
 | is true. If function is None, return the items that are true.  
 |   
 | Methods defined here:  
 |   
 | \_\_getattribute\_\_(self, name, /)  
 | Return getattr(self, name).  
 |   
 | \_\_iter\_\_(self, /)  
 | Implement iter(self).  
 |   
 | \_\_next\_\_(self, /)  
 | Implement next(self).  
 |   
 | \_\_reduce\_\_(...)  
 | Return state information for pickling.  
 |   
 | ----------------------------------------------------------------------  
 | Static methods defined here:  
 |   
 | \_\_new\_\_(\*args, \*\*kwargs) from builtins.type  
 | Create and return a new object. See help(type) for accurate signature.  
  
None  
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  
Help on class map in module builtins:  
  
class map(object)  
 | map(func, \*iterables) --> map object  
 |   
 | Make an iterator that computes the function using arguments from  
 | each of the iterables. Stops when the shortest iterable is exhausted.  
 |   
 | Methods defined here:  
 |   
 | \_\_getattribute\_\_(self, name, /)  
 | Return getattr(self, name).  
 |   
 | \_\_iter\_\_(self, /)  
 | Implement iter(self).  
 |   
 | \_\_next\_\_(self, /)  
 | Implement next(self).  
 |   
 | \_\_reduce\_\_(...)  
 | Return state information for pickling.  
 |   
 | ----------------------------------------------------------------------  
 | Static methods defined here:  
 |   
 | \_\_new\_\_(\*args, \*\*kwargs) from builtins.type  
 | Create and return a new object. See help(type) for accurate signature.  
  
None  
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  
Help on built-in function reduce in module \_functools:  
  
reduce(...)  
 reduce(function, sequence[, initial]) -> value  
   
 Apply a function of two arguments cumulatively to the items of a sequence,  
 from left to right, so as to reduce the sequence to a single value.  
 For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates  
 ((((1+2)+3)+4)+5). If initial is present, it is placed before the items  
 of the sequence in the calculation, and serves as a default when the  
 sequence is empty.  
  
None  
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  
Help on built-in function sorted in module builtins:  
  
sorted(iterable, /, \*, key=None, reverse=False)  
 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.  
  
None

如果某一函数或者方法的输入参数有函数参数，则首先考虑使用lambda的方式，这经常用于[pandas](https://pandas.pydata.org/)下配合方法apply，根据行数据增加新的列数据。

下述案例将以字典保存的保龄球大赛得分转换为DataFrame数据格式，并用apply方法，以lambda方式计算标准计分（Standard Score，z\_score，代表原始数值和平均值之间的距离，并以标准差为单位计算，即z-score是从感兴趣的点到均值之间有多少个标准差，这样就可以在不同组数据间比较某一数值的重要程度。）

浅试pandas的DataFrame数据结构。pandas数据处理方法异常丰富，是数据处理的核心应用库，可以查看pandas手册，或相关说明。

vals=[2,3,4,10,15,17,19,30,50]  
filtered\_vals=filter(lambda x:x>10, vals) #返回的是一个迭代器（iterator）  
print(filtered\_vals)  
print(list(filtered\_vals))  
  
print("-"\*60)   
mapped\_vals=map(lambda x:x\*\*2,vals)  
print(list(mapped\_vals))  
  
print("-"\*60)   
reduced\_vals=reduce(lambda x,y:x+y, vals) #相当于求和  
print(reduced\_vals)  
  
print("-"\*60)   
scores=[('Mason',90),('Reece',81),('A',73),('B',97),('C',85)]  
sorted\_vals=sorted(scores,key=lambda score:score[1])  
print(sorted\_vals)

<filter object at 0x0000020982DF1730>  
[15, 17, 19, 30, 50]  
------------------------------------------------------------  
[4, 9, 16, 100, 225, 289, 361, 900, 2500]  
------------------------------------------------------------  
150  
------------------------------------------------------------  
[('A', 73), ('Reece', 81), ('C', 85), ('Mason', 90), ('B', 97)]

bowlingContest\_scores\_dic={'Barney':86,'Harold':73,'Chris':124,'Neil':111,'Tony':90,'Simon':38,  
 'Jo':84,'Dina':71,'Graham':103,'Joe':85,'Alan':90,'Billy':89,  
 'Gordon':229,'Wade':77,'Cliff':59,'Arthur':95,'David':70,'Charles':88}  
   
print(bowlingContest\_scores\_dic)  
  
import pandas as pd  
bowlingContest\_scores\_df=pd.DataFrame.from\_dict(bowlingContest\_scores\_dic,orient='index',columns=['score'])  
print(bowlingContest\_scores\_df)  
scores\_mean=bowlingContest\_scores\_df.score.mean()  
scores\_std=bowlingContest\_scores\_df.score.std()  
print("-"\*60)   
print('scores\_mean=%.2f,scores\_std=%.2f'%(scores\_mean,scores\_std))  
bowlingContest\_scores\_df['z\_score']=bowlingContest\_scores\_df.score.apply(lambda score:round((score-scores\_mean)/scores\_std,3))  
print(bowlingContest\_scores\_df)

{'Barney': 86, 'Harold': 73, 'Chris': 124, 'Neil': 111, 'Tony': 90, 'Simon': 38, 'Jo': 84, 'Dina': 71, 'Graham': 103, 'Joe': 85, 'Alan': 90, 'Billy': 89, 'Gordon': 229, 'Wade': 77, 'Cliff': 59, 'Arthur': 95, 'David': 70, 'Charles': 88}  
 score  
Barney 86  
Harold 73  
Chris 124  
Neil 111  
Tony 90  
Simon 38  
Jo 84  
Dina 71  
Graham 103  
Joe 85  
Alan 90  
Billy 89  
Gordon 229  
Wade 77  
Cliff 59  
Arthur 95  
David 70  
Charles 88  
------------------------------------------------------------  
scores\_mean=92.33,scores\_std=39.09  
 score z\_score  
Barney 86 -0.162  
Harold 73 -0.495  
Chris 124 0.810  
Neil 111 0.477  
Tony 90 -0.060  
Simon 38 -1.390  
Jo 84 -0.213  
Dina 71 -0.546  
Graham 103 0.273  
Joe 85 -0.188  
Alan 90 -0.060  
Billy 89 -0.085  
Gordon 229 3.496  
Wade 77 -0.392  
Cliff 59 -0.853  
Arthur 95 0.068  
David 70 -0.571  
Charles 88 -0.111

* 用lambda定义递归展平嵌套列表

前述递归嵌套列表是根据索引为0的项值是否为列表给出elif和else的处理路径，如果列表为空则执行if，其为终止条件。此次给出的方法是使用lambda函数，lambda函数组合了列表推导式的方法，递归lambda自身完成嵌套列表的展平。直接看包括列表推导式和递归的lambda函数理解递归的过程，因为无法print()变量，很难查看整个计算流程，因此需要将lambda转换为def的形式，并将列表推导式转换为for循环。

图解释了整个计算流程。对循环的子列表执行递归操作，通过判断该列表是否为列表，执行if或者else语句块。else语句块为终止条件，为列表项值不再是子列表，而是单个值时的情况，则返回包含该一个值的列表，通过extend的方法，将其追加到变量lst\_collection中。注意，追加的过程，是根据循环从左到右，从外到内（子列表）迭代过程的逆序，例如先['A','B',['C','D'],'E'] 子列表，然后[6,[7,8,[9]]]子列表；前者子列表仍旧从左到右，先A，再B，在调用自身位置追加到列表为['A','B']；到['C','D']时，先执行嵌套子列表，从左到右，在调用到自身位置处追加到列表为['C','D']；在A,B和['C','D']齐平的位置合并列表为['A','B','C','D','E']，依次类推。

终止条件的返回值是返回递归调用自身的位置，例如满足终止条件的A，返回值为['A']，该值对应到递归调用的位置lst\_collection.extend(flatten\_lst\_loop(n\_lst))，而被追加到lst\_collection列表中，因为返回的是含一个值的列表，因此使用extend方法。

思考：如果将else终止条件返回值改为lst，而不是[lst]，并将调用位置语句改为lst\_collection.append(flatten\_lst\_loop(n\_lst))，即将extend改为append，为什么不可以？

flatten\_lst=lambda lst: [m for n\_lst in lst for m in flatten\_lst(n\_lst)] if type(lst) is list else [lst]  
  
nested\_lst=[['A','B',['C','D'],'E'],[6,[7,8,[9]]]]   
print(flatten\_lst(nested\_lst))

['A', 'B', 'C', 'D', 'E', 6, 7, 8, 9]

将lambda转换为def定义的形式，并拆解列表推导式，方便通过print方式查看变量值的变化。

i=0  
def flatten\_lst\_loop(lst):   
 global i  
   
 print("-"\*50,i)  
 lst\_collection=[]  
 if type(lst) is list:  
 i+=1  
 print('\*\*',lst)  
 for n\_lst in lst:  
 print("##",n\_lst)   
 lst\_collection.extend(flatten\_lst\_loop(n\_lst))  
 print(':',lst\_collection)  
 else:  
 i+=1  
 print('++',lst)  
 return [lst]  
 return lst\_collection  
   
flattened\_nestedLst=flatten\_lst\_loop(nested\_lst)  
print("-"\*60)  
print(flattened\_nestedLst)

-------------------------------------------------- 0  
\*\* [['A', 'B', ['C', 'D'], 'E'], [6, [7, 8, [9]]]]  
## ['A', 'B', ['C', 'D'], 'E']  
-------------------------------------------------- 1  
\*\* ['A', 'B', ['C', 'D'], 'E']  
## A  
-------------------------------------------------- 2  
++ A  
: ['A']  
## B  
-------------------------------------------------- 3  
++ B  
: ['A', 'B']  
## ['C', 'D']  
-------------------------------------------------- 4  
\*\* ['C', 'D']  
## C  
-------------------------------------------------- 5  
++ C  
: ['C']  
## D  
-------------------------------------------------- 6  
++ D  
: ['C', 'D']  
: ['A', 'B', 'C', 'D']  
## E  
-------------------------------------------------- 7  
++ E  
: ['A', 'B', 'C', 'D', 'E']  
: ['A', 'B', 'C', 'D', 'E']  
## [6, [7, 8, [9]]]  
-------------------------------------------------- 8  
\*\* [6, [7, 8, [9]]]  
## 6  
-------------------------------------------------- 9  
++ 6  
: [6]  
## [7, 8, [9]]  
-------------------------------------------------- 10  
\*\* [7, 8, [9]]  
## 7  
-------------------------------------------------- 11  
++ 7  
: [7]  
## 8  
-------------------------------------------------- 12  
++ 8  
: [7, 8]  
## [9]  
-------------------------------------------------- 13  
\*\* [9]  
## 9  
-------------------------------------------------- 14  
++ 9  
: [9]  
: [7, 8, 9]  
: [6, 7, 8, 9]  
: ['A', 'B', 'C', 'D', 'E', 6, 7, 8, 9]  
------------------------------------------------------------  
['A', 'B', 'C', 'D', 'E', 6, 7, 8, 9]

**6.3** 生成器（generator）

生成器包括生成器函数和生成器表达式。生成器不会一次性计算所有结果，例如以列表形式返回所有计算值（如果为海量数据，列表形式会非常耗内存），而是根据需要提取数值，这样可以增加计算的效率，节约内存空间。例如filter()，map()，zip()等函数的返回值均为迭代器（iterator）（可迭代对象），即生成器返回迭代器（对象）。

生成器函数，就是在def定义函数时，返回值以yield的方式一次返回一个结果；生成器表达式，就是将列表推导式的[]改为()就可。或者使用iter()函数将可迭代对象转换为迭代器（iterator）。

通过help()查看生成器说明文件，有\_\_next\_\_方法，即next(iterable)实现逐个读取生成器返回的迭代对象。

* 生成器函数

def squared\_generator(range\_start,range\_stop,range\_step=1):  
 for i in range(range\_start,range\_stop,range\_step):  
 yield i\*\*2  
squared\_iterable=squared\_generator(1,5)  
print(squared\_iterable)  
  
for i in squared\_iterable:print(i)  
print("-"\*60)  
for i in squared\_iterable:print(i) #当迭代完毕后，在执行则为空，需重新调用生成器函数

<generator object squared\_generator at 0x000001E013AF4430>  
1  
4  
9  
16  
------------------------------------------------------------

squared\_iterable=squared\_generator(1,5)  
for i in squared\_iterable:print(i)

1  
4  
9  
16

def computing\_generator(x):  
 x+=1  
 print('Performed addition')  
 yield x  
   
 x\*=2  
 print('Performed multiplication')  
 yield x  
  
computing\_iterable=computing\_generator(5)  
print(next(computing\_iterable))  
print(next(computing\_iterable))

Performed addition  
6  
Performed multiplication  
12

def infinite\_generator():  
 i=0  
 while True:  
 i+=1  
 yield i  
infinite\_iterable=infinite()   
print(next(infinite\_iterable))  
print(next(infinite\_iterable))  
print(next(infinite\_iterable))  
  
help(infinite\_iterable)

1  
2  
3  
Help on generator object:  
  
infinite = class generator(object)  
 | Methods defined here:  
 |   
 | \_\_del\_\_(...)  
 |   
 | \_\_getattribute\_\_(self, name, /)  
 | Return getattr(self, name).  
 |   
 | \_\_iter\_\_(self, /)  
 | Implement iter(self).  
 |   
 | \_\_next\_\_(self, /)  
 | Implement next(self).  
 |   
 | \_\_repr\_\_(self, /)  
 | Return repr(self).  
 |   
 | close(...)  
 | close() -> raise GeneratorExit inside generator.  
 |   
 | send(...)  
 | send(arg) -> send 'arg' into generator,  
 | return next yielded value or raise StopIteration.  
 |   
 | throw(...)  
 | throw(typ[,val[,tb]]) -> raise exception in generator,  
 | return next yielded value or raise StopIteration.  
 |   
 | ----------------------------------------------------------------------  
 | Data descriptors defined here:  
 |   
 | gi\_code  
 |   
 | gi\_frame  
 |   
 | gi\_running  
 |   
 | gi\_yieldfrom  
 | object being iterated by yield from, or None

* 迭代对象的send()方法

send()方法可以向yield表达式传递参数，作为yield表达式的值。

def double\_inputs():  
 while True:  
 x=yield  
 print("-"\*10,x)  
 yield x\*2  
   
di\_gen= double\_inputs()  
print(next(di\_gen))  
print(di\_gen.send(5))  
next(di\_gen)  
print(di\_gen.send(7))

None  
---------- 5  
10  
---------- 7  
14

def squared\_generator():  
 for i in range(1,10):  
 print("-"\*10,i)  
 yield i\*\*2  
sg=squared\_generator()  
print(next(sg))  
print(next(sg))  
print(next(sg))  
print("-"\*60)  
print(sg.send(7))  
print(sg.send(9))

---------- 1  
1  
---------- 2  
4  
---------- 3  
9  
------------------------------------------------------------  
---------- 4  
16  
---------- 5  
25

需要注意的是，下述计数求和代码，需要先运行next()，然后执行send()，否则提示TypeError: can't send non-None value to a just-started generator错误。send()将值传递给的是yield表达式，因此可以看到下述代码，在send(1)之后，先运行delta=yield countNtotal\_nt(count,total)语句，此时delta对象的值为1，执行if语句块，然后，依次迭代。

from collections import namedtuple  
  
countNtotal\_nt=namedtuple('countNsum',['count','total'])  
  
def countNtotal(count=0,total=0):  
 i=0  
 while True:  
 print('-'\*10,i)  
 delta=yield countNtotal\_nt(count,total)  
 print("###",delta)  
 if delta:  
 print('+'\*10,i)  
 count+=1  
 total+=delta  
 i+=1  
   
vals=[1,2,3,4,None,7,8,9]  
countNtotal\_gen=countNtotal()  
print(next(countNtotal\_gen))  
print("-"\*60)  
for v in vals:print(countNtotal\_gen.send(v))

---------- 0  
countNsum(count=0, total=0)  
------------------------------------------------------------  
### 1  
++++++++++ 0  
---------- 1  
countNsum(count=1, total=1)  
### 2  
++++++++++ 1  
---------- 2  
countNsum(count=2, total=3)  
### 3  
++++++++++ 2  
---------- 3  
countNsum(count=3, total=6)  
### 4  
++++++++++ 3  
---------- 4  
countNsum(count=4, total=10)  
### None  
---------- 4  
countNsum(count=4, total=10)  
### 7  
++++++++++ 4  
---------- 5  
countNsum(count=5, total=17)  
### 8  
++++++++++ 5  
---------- 6  
countNsum(count=6, total=25)  
### 9  
++++++++++ 6  
---------- 7  
countNsum(count=7, total=34)

* 生成器表达式

plus10\_generator=(i+10 for i in [1,2,3,4,5])  
print(plus10\_generator)  
print(list(plus10\_generator))  
  
print("-"\*60)  
lst\_iter=iter([1,2,3,4,5])  
print(lst\_iter)  
print(next(lst\_iter))  
print(next(lst\_iter))

<generator object <genexpr> at 0x000001E013AE7430>  
[11, 12, 13, 14, 15]  
------------------------------------------------------------  
<list\_iterator object at 0x000001E013BB9670>  
1  
2

* 用生成器递归展平列表

该方法逻辑同用lambda定义递归展平嵌套列表，只是这里有些处理方法的替换，1是，将if type(lst) is list:判断条件用try,except异常处理替换，当lst变量不为列表时，就会引发异常，从而执行except语句块，为结束条件；2是，用yield替换lst\_collection.extend(flatten\_lst\_loop(n\_lst))值追加到列表中，而是直接构建生成器，产生迭代对象。

def flatten\_lst\_generator(lst):  
 try: #使用语句try/except捕捉异常  
 for n\_lst in lst:   
 for m in flatten\_lst\_generator(n\_lst):  
 yield m  
   
 except:   
 yield lst  
   
nested\_lst=[['A','B',['C','D'],'E'],[6,[7,8,[9]]]]   
flatten\_nestedLst=flatten\_lst\_generator(nested\_lst)  
print(list(flatten\_nestedLst))

['A', 'B', 'C', 'D', 'E', 6, 7, 8, 9]

**6.4** itertools库

[itertools](https://docs.python.org/3/library/itertools.html)①库可以实现一系列迭代器（iterator），可以简洁而高效的完成相关的算法，而避免自行重新编写相关的工具。理解与善用itertools库，可以提高代码书写效率，及高效利用内存。对itertools可以直接查看官网。下述摘抄了官网说明表格，记录如下，方便查阅。通过表格给出的参数，结果和示例很容易推断工具的使用方法和算法目的。

* Infinite iterators（无穷迭代器）:

| Iterator | Arguments | Results | Example |
| --- | --- | --- | --- |
| count() | start, [step] | start, start+step, start+2\*step, … | count(10) –> 10 11 12 13 14 … |
| cycle() | p | p0, p1, … plast, p0, p1, … | cycle(‘ABCD’) –> A B C D A B C D … |
| repeat() | elem [,n] | elem, elem, elem, … endlessly or up to n times | repeat(10, 3) –> 10 10 10 |

* Iterators terminating on the shortest input sequence（根据最短输入序列长度停止的迭代器）:

| accumulate() | p [,func] | p0, p0+p1, p0+p1+p2, … | accumulate([1,2,3,4,5]) –> 1 3 6 10 15 |
| --- | --- | --- | --- |
| chain() | p, q, … | p0, p1, … plast, q0, q1, … | chain(‘ABC’, ‘DEF’) –> A B C D E F |
| chain.from\_iterable() | iterable | p0, p1, … plast, q0, q1, … | chain.from\_iterable([‘ABC’, ‘DEF’]) –> A B C D E F |
| compress() | data, selectors | (d[0] if s[0]), (d[1] if s[1]), … | compress(‘ABCDEF’, [1,0,1,0,1,1]) –> A C E F |
| dropwhile() | pred, seq | seq[n], seq[n+1], starting when pred fails | dropwhile(lambda x: x<5, [1,4,6,4,1]) –> 6 4 1 |
| filterfalse() | pred, seq | elements of seq where pred(elem) is false | filterfalse(lambda x: x%2, range(10)) –> 0 2 4 6 8 |
| groupby() | iterable[, key] | sub-iterators grouped by value of key(v) |  |
| islice() | seq, [start,] stop [, step] | elements from seq[start:stop:step] | islice(‘ABCDEFG’, 2, None) –> C D E F G |
| pairwise() | iterable | (p[0], p[1]), (p[1], p[2]) | pairwise(‘ABCDEFG’) –> AB BC CD DE EF FG |
| starmap() | func, seq | func(*seq[0]), func(*seq[1]), … | starmap(pow, [(2,5), (3,2), (10,3)]) –> 32 9 1000 |
| takewhile() | pred, seq | seq[0], seq[1], until pred fails | takewhile(lambda x: x<5, [1,4,6,4,1]) –> 1 4 |
| tee() | it, n | it1, it2, … itn splits one iterator into n |  |
| zip\_longest() | p, q, … | (p[0], q[0]), (p[1], q[1]), … | zip\_longest(‘ABCD’, ‘xy’, fillvalue=‘-’) –> Ax By C- D- |

* Combinatoric iterators（排列组合迭代器）:

| Iterator | Arguments | Results |
| --- | --- | --- |
| product() | p, q, … [repeat=1] | cartesian product, equivalent to a nested for-loop |
| permutations() | p[, r] | r-length tuples, all possible orderings, no repeated elements |
| combinations() | p, r | r-length tuples, in sorted order, no repeated elements |
| combinations\_with\_replacement() | p, r | r-length tuples, in sorted order, with repeated elements |

注释（Notes）：

① itertools — Functions creating iterators for efficient looping，（<https://docs.python.org/3/library/itertools.html>）。

PC6-ipynb download