## 迭代器

在python中许多对象都可以通过迭代工具（这些迭代工具包括for循环、列表解析、in成员关系测试以及map内置函数等）进行**遍历**，这些对象都可以被称为可迭代对象，基本上，这就是序列观念的通用化：**如果对象是实际保存的序列，或者可以在迭代工具环境中一次产生一个结果对象，就看做是可迭代的。**

迭代器对象要求支持迭代器协议的对象，在Python中，支持迭代器协议就是实现对象的\_\_iter\_\_()和next()方法。

\_\_next\_\_

返回下一个可用的元素，如果没有元素了，抛出 StopIteration 异常。

\_\_iter\_\_

返回 一个可迭代对象，以便在应该使用可迭代对象的地方使用迭代器，例如在 for 循环中。

### 1.1 文件迭代器

>>> f = open('script1.py')

>>> f.readline()

'import sys\n'

>>> f.readline()

'print sys.path\n'

>>> f.readline()

'x = 2\n'

>>> f.readline()

'print 2 \*\* 33\n'

>>> f.readline()

''

所有迭代工具工作起来都是在每次迭代中调用next（python3为\_\_next\_\_）方法，并且捕获StopIteration异常来确定何时离开。

>>> f = open('script1.py')

>>> f.next()

'import sys\n'

>>> f.next()

'print sys.path\n'

>>> f.next()

'x = 2\n'

>>> f.next()

'print 2 \*\* 33\n'

>>> f.next()

Traceback (most recent call last):

File "<pyshell#24>", line 1, in <module>

f.next()

StopIteration

### 1.2 手动迭代：iter和next

Python3.0还提供了内置函数next，它会自动调用一个对象的\_\_next\_\_方法。

从技术角度讲，当for循环开始时，会通过它传给iter内置函数，以便从可迭代对象中获得一个迭代器，返回的对象含有需要的next方法，而且会捕捉StopIteration异常，从而决定何时结束循环。

>>> f = open('script1.py')

>>> dir(f)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_doc\_\_', '\_\_enter\_\_', '\_\_exit\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'close', 'closed', 'encoding', 'errors', 'fileno', 'flush', 'isatty', 'mode', 'name', 'newlines', 'next', 'read', 'readinto', 'readline', 'readlines', 'seek', 'softspace', 'tell', 'truncate', 'write', 'writelines', 'xreadlines']

>>> dir([1,2,3])

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_delslice\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getslice\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_setslice\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>>

>>> dir(iter([1,2,3]))

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_doc\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_length\_hint\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'next']

>>>

对于列表以及其他的内置对象，不是自身的迭代器，我们必须调用iter来启动迭代。

### 1.3 列表解析

>>> L = [1,2,3]

>>> L = [x+10 for x in L]

>>> L

[11, 12, 13]

>>>

**列表解析比手动的for循环语句执行快（往往速度会快一倍），因为它们的迭代在解释器内部是以c语言的速度执行的，而不是以手动的python代码执行。**

>>> [line.upper() for line in open('script1.py') ]

['IMPORT SYS\n', 'PRINT SYS.PATH\n', 'X = 2\n', 'PRINT 2 \*\* 33\n']

>>>

>>> [x+y for x in 'abc' for y in 'lmn']

['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']

>>>

>>> [line.rstrip() for line in open('script1.py').readlines()]

['import sys', 'print sys.path', 'x = 2', 'print 2 \*\* 33']

>>> [line.rstrip() for line in open('script1.py')]

['import sys', 'print sys.path', 'x = 2', 'print 2 \*\* 33']

>>> map((lambda line: line.rstrip()), open('script1.py'))

['import sys', 'print sys.path', 'x = 2', 'print 2 \*\* 33']

## 其他迭代环境

map和列表解析是在解释器中以c语言的速度来运行的，比python的for循环代码在pvm中步进运行快的多。

### 2.1 map函数

在对象中从左到右扫描的每种工具都使用了迭代协议，列表解析、in成员关系测试、map内置函数以及像sorted和zip调用这样的内置函数都使用了迭代协议。

map(function, iterable, ...)

Apply function to every item of iterable and return a list of the results. If additional iterable arguments are passed, function must take that many arguments and is applied to the items from all iterables in parallel. If one iterable is shorter than another it is assumed to be extended withNoneitems. If function isNone, the identity function is assumed; if there are multiple arguments, [map()](http://my.oschina.net/zyzzy/admin/" \l "map) returns a list consisting of tuples containing the corresponding items from all iterables (a kind of transpose operation). The iterable arguments may be a sequence or any iterable object; the result is always a list.

* 对可迭代函数'iterable'中的每一个元素应用‘function’方法，将结果作为list返回。

>>> def add100(x):

return x+100

>>> map(add100, [1,2,3])

[101, 102, 103]

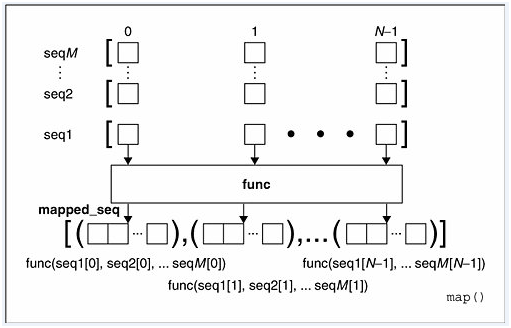
* 如果给出了额外的可迭代参数，则对每个可迭代参数中的元素‘并行’的应用‘function’。

>>> def abc(a,b,c):

return a+b+c

>>> map(abc, [1,2,3], [4,5,6], [7,8,9])

[12, 15, 18]



* 如果'function'给出的是‘None’，zip()开始取代map函数，目的是将多个列表相同位置的元素归并到一个元组。

>>> map(None, [1,2,3], [4,5,6])

[(1, 4), (2, 5), (3, 6)]

### 2.2 sorted函数

在python中，list的sort方法返回的是对已经存在的列表进行操作，而内建函数sorted方法返回的是一个新的list，而不是在原来的基础上进行的操作。

>>> help(sorted)

Help on built-in function sorted in module \_\_builtin\_\_:

sorted(...)

sorted(iterable, cmp=None, key=None, reverse=False) --> new sorted list

iterable：是可迭代类型;  
cmp：用于比较的函数，比较什么由key决定;  
key：用列表元素的某个属性或函数进行作为关键字，有默认值，迭代集合中的一项;  
reverse：排序规则. reverse = True  降序 或者 reverse = False 升序，有默认值。  
返回值：是一个经过排序的可迭代类型，与iterable一样。

class **Student**:

def **\_\_init\_\_**(*self*, name, age, score):

*self*.age = age

*self*.name = name

*self*.score = score

def **\_\_repr\_\_**(*self*):

return repr((*self*.name, *self*.score, *self*.age))

student\_some = [Student(*'Kel'*, 24, *'B'*), Student(*'Jun'*, 30, *'A'*), Student(*'Dave'*, 29, *'C'*)]

print sorted(student\_some, key = lambda student: student.age)

print student\_some

Output：

[('Kel', 'B', 24), ('Dave', 'C', 29), ('Jun', 'A', 30)]

[('Kel', 'B', 24), ('Jun', 'A', 30), ('Dave', 'C', 29)]

### 2.3 range迭代器

与python2.X不同，python3.0中range内置函数返回一个迭代器，该迭代器根据需要产生范围中数字，而不是内存中构建一个结果列表（取代了python2.x中的xrange）。

>>> range(10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> R = range(10)

>>> dir(R)

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_delslice\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getslice\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_setslice\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>> dir(iter(R))

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_doc\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_length\_hint\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'next']

>>> xrange(10)

xrange(10)

>>> list(xrange(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

xrange与range比较：

* xrange返回一个xrange对象，而不是一个列表。
* 与range相比，在循环中更快，内存效率更高。

### 2.4 zip内置函数

和range类似，map、zip以及filter内置函数在python3.0中也转变成迭代器以节约内存空间，而不再在内存中一次性生成结果列表。

zip()是Python的一个内建函数，它接受一系列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tuple（元组），然后返回由这些tuples组成的list（列表）。若传入参数的长度不等，则返回list的长度和参数中长度最短的对象相同。利用\*号操作符，可以将list unzip（解压）。

zip(...)

zip(seq1 [, seq2 [...]]) -> [(seq1[0], seq2[0] ...), (...)]

Return a list of tuples, where each tuple contains the i-th element

from each of the argument sequences. The returned list is truncated

in length to the length of the shortest argument sequence.

>>> Z = zip((1,2,3), (4,5,6))

>>> Z

[(1, 4), (2, 5), (3, 6)]

>>> Z = zip('ab', 'xyz')

>>> Z

[('a', 'x'), ('b', 'y')]

### 2.5 filter内置函数

>>> help(filter)

Help on built-in function filter in module \_\_builtin\_\_:

filter(...)

filter(function or None, sequence) -> list, tuple, or string

Return those items of sequence for which function(item) is true. If

function is None, return the items that are true. If sequence is a tuple

or string, return the same type, else return a list.

>>>

对sequence中的item依次执行function(item)，将执行结果为True的item组成一个List/String/Tuple（取决于sequence的类型）返回。

>>> def is\_odd(x):

return x % 2 == 1

>>> filter(is\_odd, [1,4,6,12,17])

[1, 17]

>>> filter(is\_odd, (2,3,4,67,8))

(3, 67)

>>>

>>> def is\_not\_empty(s):

return s and len(s.strip()) > 0

>>> filter(is\_not\_empty, ['test', None, '', 'str', ' ', 'END'])

['test', 'str', 'END']

>>>

>>> filter(None, [1,2,3])

[1, 2, 3]

>>>

### 2.6 enumerate函数

>>> help(enumerate)

Help on class enumerate in module \_\_builtin\_\_:

class enumerate(object)

| enumerate(iterable[, start]) -> iterator for index, value of iterable

|

| Return an enumerate object. iterable must be another object that supports

| iteration. The enumerate object yields pairs containing a count (from

| start, which defaults to zero) and a value yielded by the iterable argument.

| enumerate is useful for obtaining an indexed list:

| (0, seq[0]), (1, seq[1]), (2, seq[2]), ...

|

| Methods defined here:

|

| \_\_getattribute\_\_(...)

| x.\_\_getattribute\_\_('name') <==> x.name

|

| \_\_iter\_\_(...)

| x.\_\_iter\_\_() <==> iter(x)

|

| next(...)

| x.next() -> the next value, or raise StopIteration

|

| ----------------------------------------------------------------------

| Data and other attributes defined here:

|

| \_\_new\_\_ = <built-in method \_\_new\_\_ of type object>

| T.\_\_new\_\_(S, ...) -> a new object with type S, a subtype of T

enumerate函数用于遍历序列中的元素以及它们的下标,多用于在for循环中得到计数,enumerate参数为可遍历的变量，如 字符串，列表等。

>>> enumerate([4,56,7])

<enumerate object at 0x0000000002F713A8>

>>> for idx, value in enumerate([4,56,7]):

print idx, value

0 4

1 56

2 7

>>> print list(enumerate('abc'))

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

>>>

### 2.7 reduce函数

Help on built-in function reduce in module \_\_builtin\_\_:

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.

reduce()函数接收的参数和 map()类似，一个函数 f，一个list，但行为和 map()不同，reduce()传入的函数 f 必须接收两个参数，reduce()对list的每个元素反复调用函数f，并返回最终结果值。

reduce()还可以接收第3个可选参数，作为计算的初始值。如果把初始值设为100，计算：

|  |  |  |
| --- | --- | --- |
| 1 | reduce(f, [1, 3, 5, 7, 9], 100) |  |

结果将变为125，计算初始值和第一个元素：**f(100, 1)**，结果为**101**。

## 生成器

### 3.1 生成器概念

**生成器不会把结果保存在一个序列中，而是保存生成器的状态，在每次进行迭代时返回一个值，直到遇到StopIteration异常结束。**

生成器表达式： 通列表解析语法，只不过把列表解析的[]换成()  
生成器表达式能做的事情列表解析基本都能处理，只不过在需要处理的序列比较大时，**列表解析比较费内存。**

**所有生成器都是迭代器，因为生成器完全实现了迭代器接口。**

>>> gen = (x\*\*2 for x in range(5))

>>> gen

<generator object <genexpr> at 0x00000000026AE828>

>>> dir(gen)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_doc\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_name\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'close', 'gi\_code', 'gi\_frame', 'gi\_running', 'next', 'send', 'throw']

>>>

>>> next(gen)

0

>>> next(gen)

1

>>> next(gen)

4

>>> next(gen)

9

>>> next(gen)

16

>>> next(gen)

Traceback (most recent call last):

File "<pyshell#10>", line 1, in <module>

next(gen)

StopIteration

>>>

### 3.2 生成器函数

  在函数中如果出现了yield关键字，那么该函数就不再是普通函数，而是生成器函数。

yield 的作用就是把一个函数变成一个 generator，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。

Python3.x中**yield from iterable本质上等于for item in iterable: yield item的缩写版 。**

>>> def gensqures(N):

for i in range(N):

yield i \*\* 2

>>> for i in gensqures(5):

print i , ": ",

0 : 1 : 4 : 9 : 16 :

>>>

>>> dir(gensqures(5))

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_doc\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_name\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'close', 'gi\_code', 'gi\_frame', 'gi\_running', 'next', 'send', 'throw']

>>>

>>> X = gensqures(5)

>>> X

<generator object gensqures at 0x00000000027F44C8>

>>> next(X)

0

>>> next(X)

1

>>> next(X)

4

>>> next(X)

9

>>> next(X)

16

>>> next(X)

Traceback (most recent call last):

File "<pyshell#22>", line 1, in <module>

next(X)

StopIteration

>>>

无穷生产奇数的生成器函数：

def **odd**():

n = 1

while True:

yield n

n += 2

odd\_num = odd()

count = 0

for odd in odd\_num:

if count >= 5:

break

print odd

count += 1

等价的迭代器代码：

class **Iter**:

def **\_\_init\_\_**(*self*):

*self*.start = -1

def **\_\_iter\_\_**(*self*):

return *self*

def **next**(*self*):

*self*.start += 2

return *self*.start

I = Iter()

count = 0

for odd in I:

if count >= 5:

break

print odd

count += 1

### 3.3 yield与return

在一个生成器中，如果没有return，则默认执行到函数完毕时返回StopIteration；

>>> def test():

yield 'a'

yield 'b'

>>> T = test()

>>> T

<generator object test at 0x00000000026F80D8>

>>> T.next()

'a'

>>> T.next()

'b'

>>> T.next()

Traceback (most recent call last):

File "<pyshell#20>", line 1, in <module>

T.next()

StopIteration

>>>

**如果遇到return,如果在执行过程中 return，则直接抛出 StopIteration 终止迭代。**

>>> def test():

yield 'a'

return

yield 'b'

>>> T = test()

>>> T.next()

'a'

>>> T.next()

Traceback (most recent call last):

File "<pyshell#28>", line 1, in <module>

T.next()

StopIteration

如果在return后返回一个值，那么这个值为StopIteration异常的说明，不是程序的返回值。生成器没有办法使用return来返回值。

Python2.X return不能带返回值

>>> def gen():

yield 'a'

return 'hello'

SyntaxError: 'return' with argument inside generator (<pyshell#2>, line 4)

Python3.x：

>>> def test():

yield 'a'

return "hello"

>>> T = test()

>>> next(T)

'a'

>>> next(T)

Traceback (most recent call last):

File "<pyshell#11>", line 1, in <module>

next(T)

StopIteration: hello

>>>

### 3.4 生成器支持的方法

**close()：手动关闭生成器函数，后面的调用会直接返回StopIteration异常。**

>>> def yid():

yield 'a'

yield 'b'

>>> Y = yid()

>>> next(Y)

'a'

>>> Y.close()

>>> next(Y)

Traceback (most recent call last):

File "<pyshell#18>", line 1, in <module>

next(Y)

StopIteration

>>>

**send()：生成器函数最大的特点是可以接受外部传入的一个变量，并根据变量内容计算结果后返回。**

>>> def gen():

value = 0

while True:

receive = yield value

if receive == 'e':

break

value = "got: %s" % receive

>>> G = gen()

>>> print G.send(None) # TypeError: can't send non-None value to a just-started generator，如果生成器还没有执行，第一次send数据必须为None。

0

>>> print G.send('aaa')

got: aaa

>>> print G.send(3)

got: 3

>>> print G.send('e')

Traceback (most recent call last):

File "<pyshell#31>", line 1, in <module>

print G.send('e')

StopIteration

>>>  
生成器：

>>> def gen():

for i in range(3):

X = yield i

print "X = ", X

>>> G = gen()

>>> next(G)

0

>>> G.send(77)

X = 77

1

>>> G.send(88)

X = 88

2

>>> next(G)

X = None

Traceback (most recent call last):

File "<pyshell#54>", line 1, in <module>

next(G)

StopIteration

>>>

 throw()：用来向生成器函数送入一个异常，可以结束系统定义的异常，或者自定义的异常。  
throw()后直接跑出异常并结束程序，或者消耗掉一个yield，或者在没有下一个yield的时候直接进行到程序的结尾。

>>> def gen():

while True:

try:

yield 'normal value'

yield 'normal value 2'

print('here')

except ValueError:

print('we got ValueError here')

except TypeError:

break

>>> G = gen()

>>> print next(G)

normal value

>>> print G.throw(ValueError)

we got ValueError here

normal value

>>> print next(G)

normal value 2

>>> print next(G)

here

normal value

>>> print next(G)

normal value 2

>>> print next(G)

here

normal value

>>> print G.throw(TypeError)

Traceback (most recent call last):

File "<pyshell#65>", line 1, in <module>

print G.throw(TypeError)

StopIteration

>>>

### 3.5总结

1. 按照鸭子模型理论，生成器就是一种迭代器，可以使用for进行迭代。
2. 第一次执行next(generator)时，会执行完yield语句后程序进行挂起，所有的参数和状态会进行保存。再一次执行next(generator)时，会从挂起的状态开始往后执行。在遇到程序的结尾或者遇到StopIteration时，循环结束。
3. 可以通过generator.send(arg)来传入参数，这是协程模型。
4. 可以通过generator.throw(exception)来传入一个异常。**throw语句会消耗掉一个yield。**可以通过generator.close()来手动关闭生成器。
5. next()等价于send(None)

## 代码实例：对迭代的各种方法进行计时

Mytimer.py：

import time

reps = 1000

repslist = range(reps)

def **timer**(func, \*pargs, \*\*kargs):

start = time.clock()

for i in repslist:

ret = func(\*pargs, \*\*kargs)

elapsed = time.clock() - start

return (elapsed, ret)

Timeseqs.py:

import sys, mytimer

from compiler.ast import GenExpr

reps = 10000

repslist = range(reps)

def **forLoop**():

res = []

for x in repslist:

res.append(abs(x))

return res

def **listComp**():

return [abs(x) for x in repslist]

def **mapCall**():

return list(map(abs, repslist))

def **genExpr**():

return list(abs(x) for x in repslist)

def **genFunc**():

def **gen**():

for x in repslist:

yield abs(x)

return list(gen())

print sys.version

for test in (forLoop, listComp, mapCall, genExpr, genFunc):

elapsed, result = mytimer.timer(test)

print *"\_ "* \* 33

print *"%-9s:%.5f => [%s...%s]"* %(test.\_\_name\_\_, elapsed, result[0], result[-1])

结果：

2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:53:40) [MSC v.1500 64 bit (AMD64)]

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

forLoop :13.70444 => [0...99999]

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

listComp :8.22495 => [0...99999]

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

mapCall :5.34949 => [0...99999]

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

genExpr :10.41403 => [0...99999]

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

genFunc :10.31133 => [0...99999]