## 25. list是如何实现的。

在Python中list特别有用。让我们来看下list的内部是如何实现的。来看下面简单的程序，在list中添加一些整数并将他们打印出来。

>>> L = []

>>> L.append(1)

>>> L.append(2)

>>> L.append(3)

>>> L

[1, 2, 3]

>>> for e in L:

... print e

...

1

2

3

正如你所看到的，list是可以迭代的。

### List对象的C结构

Python中list是用下边的C语言的结构来表示的。ob\_item是用来保存元素的指针数组，allocated是ob\_item预先分配的内存总容量

typedef struct {

PyObject\_VAR\_HEAD

PyObject \*\*ob\_item;

Py\_ssize\_t allocated;

} PyListObject;

### List的初始化

让我们来看下当初始化一个空list的时候发生了什么 L = []

arguments: size of the list = 0

returns: list object = []

PyListNew:

nbytes = size \* size of global Python object = 0

allocate new list object

allocate list of pointers (ob\_item) of size nbytes = 0

clear ob\_item

set list's allocated var to 0 = 0 slots

return list object

非常重要的是知道list申请内存空间的大小（后文用allocated代替）的大小和list实际存储元素所占空间的大小(ob\_size)之间的关系，ob\_size的大小和len(L)是一样的，而allocated的大小是在内存中已经申请空间大小。通常你会看到allocated的值要比ob\_size的值要大。这是为了避免每次有新元素加入list时都要调用realloc进行内存分配。接下来我们会看到更多关于这些的内容。

### Append

我们在list中追加一个整数:L.append(1)。发生了什么？调用了内部的C函数app1()

arguments: list object, new element

returns: 0 if OK, -1 if not

app1:

n = size of list

call list\_resize() to resize the list to size n+1 = 0 + 1 = 1

list[n] = list[0] = new element

return 0

来让我们看下list\_resize(),list\_resize()会申请多余的空间以避免调用多次list\_resize()函数，list增长的模型是:0, 4, 8, 16, 25, 35, 46, 58, 72, 88, …

arguments: list object, new size

returns: 0 if OK, -1 if not

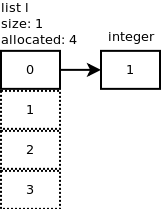
list\_resize:

new\_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6) = 3

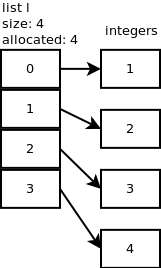
new\_allocated += newsize = 3 + 1 = 4

resize ob\_item (list of pointers) to size new\_allocated

return 0

开辟了四个内存空间来存放list中的元素，存放的第一个元素是1。你可以从下图中看到L[0]指向了我们刚刚加进去的元素。虚线的框代表了申请了但是还没有使用(存储元素)的内存空间  


我们继续加入一个元素：L.append(2)。调用list\_resize,同时n+1=2。但是因为allocated（译者注：已经申请的空间大小）是4。所以没有必要去申请新的内存空间。相同的事情发生在再次在list中添加两个元素的时候: L.append(3),L.append(4)。下图展示了到目前为止我们做了什么。



### Insert

现在我们在列表的第一个位置插入一个整数5:L.insert(1, 5),看看内部发生了什么。调用了ins1()

arguments: list object, where, new element

returns: 0 if OK, -1 if not

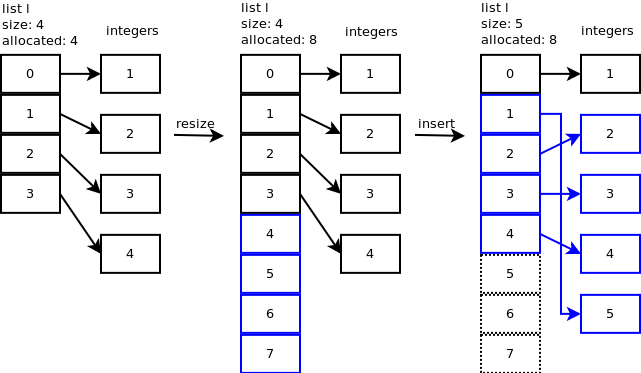
ins1:

resize list to size n+1 = 5 -> 4 more slots will be allocated

starting at the last element up to the offset where, right shift each element

set new element at offset where

return 0



虚线框表示已经申请但是没有使用的内存。申请了8个内存空间但是list实际用来存储元素只使用了其中5个内存空间。insert的时间复杂度是O(n)

### Pop

当你弹出list的最后一个元素：L.pop()。调用listpop()，list\_resize在函数listpop()内部被调用，如果这时ob\_size（译者注：弹出元素后）小于allocated（译者注：已经申请的内存空间）的一半。这时申请的内存空间将会缩小。

arguments: list object

returns: element popped

listpop:

if list empty:

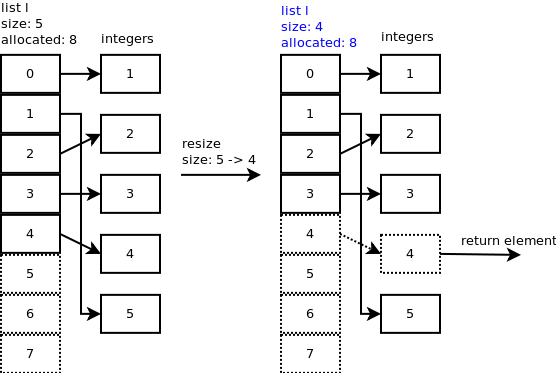
return null

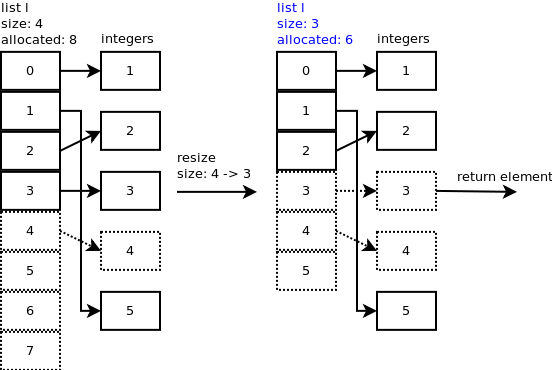
resize list with size 5 - 1 = 4. 4 is not less than 8/2 so no shrinkage

set list object size to 4

return last element

Pop的时间复杂度是O(1)



你可以发现4号内存空间指向还指向那个数值（译者注：弹出去的那个数值），但是很重要的是ob\_size现在却成了4. 让我们再弹出一个元素。在list\_resize内部，size – 1 = 4 – 1 = 3 比allocated（已经申请的空间）的一半还要小。所以list的申请空间缩小到6个，list的实际使用空间现在是3个(译者注：根据(newsize >> 3) + (newsize < 9 ? 3 : 6) = 3在文章最后有详述)你可以发现（下图）3号和4号内存空间还存储着一些整数，但是list的实际使用(存储元素)空间却只有3个了。  


### Remove

Python list对象有一个方法可以移除一个指定的元素。调用listremove()。

arguments: list object, element to remove

returns none if OK, null if not

listremove:

loop through each list element:

if correct element:

slice list between element's slot and element's slot + 1

return none

return null

切开list和删除元素，调用了list\_ass\_slice()（译者注：在上文slice list between element's slot and element's slot + 1被调用），来看下list\_ass\_slice()是如何工作的。在这里，低位为1 高位为2（译者注：传入的参数），我们移除在1号内存空间存储的数据5

arguments: list object, low offset, high offset

returns: 0 if OK

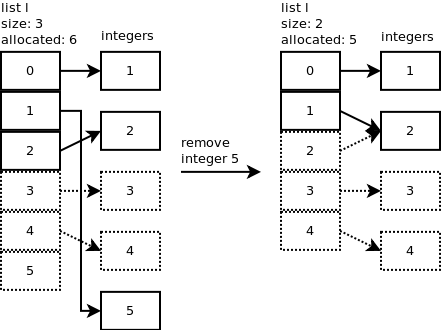
list\_ass\_slice:

copy integer 5 to recycle list to dereference it

shift elements from slot 2 to slot 1

resize list to 5 slots

return 0

Remove的时间复杂度为O(n)  


**译者注：**

文中list的sort部分没有进行翻译  
核心部分

我们能看到 Python 设计者的苦心。在需要的时候扩容,但又不允许过度的浪费,适当的内存回收是非常必要的。

这个确定调整后的空间大小算法很有意思。

调整后大小 (new\_allocated) = 新元素数量 (newsize) + 预留空间 (new\_allocated)

调整后的空间肯定能存储 newsize 个元素。要关注的是预留空间的增长状况。将预留算法改成 Python 版就更清楚了:(newsize // 8) + (newsize < 9 and 3 or 6)。当 newsize >= allocated,自然按照这个新的长度 "扩容" 内存。而如果 newsize < allocated,且利用率低于一半呢?

allocated newsize new\_size + new\_allocated

10 4 4 + 3

20 9 9 + 7

很显然,这个新长度小于原来的已分配空间长度,自然会导致 realloc 收缩内存。(不容易啊)

引自《深入Python编程》

## Python的is

is是对比地址,==是对比值

## read,readline和readlines

* read 读取整个文件
* readline 读取下一行,使用生成器方法
* readlines 读取整个文件到一个迭代器以供我们遍历

## 26. Python2和3的区别

### print函数

***Python 2***

*print 'Python', python\_version()*

*print 'Hello, World!'*

*print('Hello, World!')*

*print "text", ; print 'print more text on the same line'*

*run result:*

Python 2.7.6  
Hello, World!  
Hello, World!  
text print more text on the same line

***Python 3***

*print('Python', python\_version())*

*print('Hello, World!')*

*print("some text,", end="")*

*print(' print more text on the same line')*

*run result:*

Python 3.4.1  
Hello, World!  
some text, print more text on the same line

print 'Hello, World!'

run result

File ““, line 1  
　　print ‘Hello, World!’  
　　　　　　　　　　^  
SyntaxError: invalid syntax

**Note**

以上通过 Python 2 使用 Printing "Hello, World" 是非常正常的，尽管如此，如果你有多个对象在小括号中，我们将创建一个元组，因为 print在 Python 2 中是一个声明，而不是一个函数调用。

print 'Python', python\_version()

print('a', 'b')

print 'a', 'b'

run result:

Python 2.7.7  
(‘a’, ‘b’)  
a b

### 整除

如果你正在移植代码，这个变化是特别危险的。或者你在 Python 2 上执行 Python 3 的代码。因为这个整除的变化表现在它会被忽视（即它不会抛出语法异常）。  
　　因此，我还是倾向于使用一个 float(3)/2 或 3/2.0 代替在我的 Python 3 脚本保存在 Python 2 中的 3/2 的一些麻烦（并且反而过来也一样，我建议在你的 Python 2 脚本中使用 from \_\_future\_\_ import division）

Python2

print 'Python', python\_version()

print '3 / 2 =', 3 / 2

print '3 // 2 =', 3 // 2

print '3 / 2.0 =', 3 / 2.0

print '3 // 2.0 =', 3 // 2.0

run result:

Python 2.7.6  
3 / 2 = 1  
3 // 2 = 1  
3 / 2.0 = 1.5  
3 // 2.0 = 1.0

Python3

print('Python', python\_version())

print('3 / 2 =', 3 / 2)

print('3 // 2 =', 3 // 2)

print('3 / 2.0 =', 3 / 2.0)

print('3 // 2.0 =', 3 // 2.0)

run result:

Python 3.4.1  
3 / 2 = 1.5  
3 // 2 = 1  
3 / 2.0 = 1.5  
3 // 2.0 = 1.0

### Unicode

Python 2 有 ASCII str() 类型，unicode() 是单独的，不是 byte 类型。现在，在 Python3，我们最终有了 Unicode (utf-8) 字符串，以及一个字节类：byte 和 bytearrays。

***Python 2***

print 'Python', python\_version()

run result:

Python 2.7.6

print type(unicode('this is like a python3 str type'))

run result:

< type ‘unicode’ >

print type(b'byte type does not exist')

run result:

< type ‘str’ >

print 'they are really' + b' the same'

run result:

they are really the same

print type(bytearray(b'bytearray oddly does exist though'))

*run result:*

< type ‘bytearray’ >

***Python 3***

print('Python', python\_version())

print('strings are now utf-8 \u03BCnico\u0394é!')

run result:

Python 3.4.1

strings are now utf-8 μnicoΔé!

print('Python', python\_version(), end="")

print(' has', type(b' bytes for storing data'))

*run result:*

Python 3.4.1 has < class ‘bytes’ >

print('and Python', python\_version(), end="")

print(' also has', type(bytearray(b'bytearrays')))

run result:

and Python 3.4.1 also has < class ‘bytearray’>

## 27. super

一、问题的发现与提出

　　在Python类的方法（method）中，要调用父类的某个方法，在Python 2.2以前，通常的写法如代码段1：

class A:

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

print "enter A"

print "leave A"

class B(A):

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

print "enter B"

A.\_\_init\_\_(self)

print "leave B"

>>> b = B()

enter B

enter A

leave A

leave B

即，使用非绑定的类方法（用类名来引用的方法），并在参数列表中，引入待绑定的对象（self），从而达到调用父类的目的。

　　这样做的缺点是，当一个子类的父类发生变化时（如类B的父类由A变为C时），必须遍历整个类定义，把所有的通过非绑定的方法的类名全部替换过来，例如代码段2，

 代码段2：

class B(C): # A --> C

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

print "enter B"

C.\_\_init\_\_(self) # A --> C

print "leave B"

如果代码简单，这样的改动或许还可以接受。但如果代码量庞大，这样的修改可能是灾难性的。

　　因此，自Python 2.2开始，Python添加了一个关键字super，来解决这个问题。下面是Python 2.3的官方文档说明：

 super(type[, object-or-type])

  Return the superclass of type. If the second argument is omitted the super object  
  returned is unbound. If the second argument is an object, isinstance(obj, type)   
  must be true. If the second argument is a type, issubclass(type2, type) must be   
  true. super() only works for new-style classes.

  A typical use for calling a cooperative superclass method is:

   class C(B):  
       def meth(self, arg):  
           super(C, self).meth(arg)

  New in version 2.2.

　　从说明来看，可以把类B改写如代码段3：

 代码段3：

class A(object): # A must be new-style class

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

print "enter A"

print "leave A"

class B(C): # A --> C

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

print "enter B"

super(B, self).\_\_init\_\_()

print "leave B"

尝试执行上面同样的代码，结果一致，但修改的代码只有一处，把代码的维护量降到最低，是一个不错的用法。因此在我们的开发过程中，super关键字被大量使用，而且一直表现良好。

　　在我们的印象中，对于super(B, self).\_\_init\_\_()是这样理解的：super(B, self)首先找到B的父类（就是类A），然后把类B的对象self转换为类A的对象（通过某种方式，一直没有考究是什么方式，惭愧），然后“被转换”的类A对象调用自己的\_\_init\_\_函数。考虑到super中只有指明子类的机制，因此，在多继承的类定义中，通常我们保留使用类似代码段1的方法。

　　有一天某同事设计了一个相对复杂的类体系结构（我们先不要管这个类体系设计得是否合理，仅把这个例子作为一个题目来研究就好），代码如代码段4：

 代码段4：

class A(object):

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

print "enter A"

print "leave A"

class B(object):

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

print "enter B"

print "leave B"

class C(A):

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

print "enter C"

super(C, self).\_\_init\_\_()

print "leave C"

class D(A):

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

print "enter D"

super(D, self).\_\_init\_\_()

print "leave D"

class E(B, C):

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

print "enter E"

B.\_\_init\_\_(self)

C.\_\_init\_\_(self)

print "leave E"

class F(E, D):

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

print "enter F"

E.\_\_init\_\_(self)

D.\_\_init\_\_(self)

print "leave F"

enter F

enter E

enter B

leave B

enter C

enter D

enter A

leave A

leave D

leave C

leave E

enter D

enter A

leave A

leave D

leave F

　明显地，类A和类D的初始化函数被重复调用了2次，这并不是我们所期望的结果！我们所期望的结果是最多只有类A的初始化函数被调用2次——其实这是多继承的类体系必须面对的问题。我们把代码段4的类体系画出来，如下图：

    object  
   |       \  
   |        A  
   |      / |  
   B  C  D  
    \   /   |  
      E    |  
        \   |  
          F

　　按我们对super的理解，从图中可以看出，在调用类C的初始化函数时，应该是调用类A的初始化函数，但事实上却调用了类D的初始化函数。好一个诡异的问题！

　　也就是说，mro中记录了一个类的所有基类的类类型序列。查看mro的记录，发觉包含7个元素，7个类名分别为：

 F E B C D A object

　　从而说明了为什么在C.\_\_init\_\_中使用super(C, self).\_\_init\_\_()会调用类D的初始化函数了。 ???

　　我们把代码段4改写为：

 代码段9：

class A(object):

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

print "enter A"

super(A, self).\_\_init\_\_() # new

print "leave A"

class B(object):

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

print "enter B"

super(B, self).\_\_init\_\_() # new

print "leave B"

class C(A):

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

print "enter C"

super(C, self).\_\_init\_\_()

print "leave C"

class D(A):

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

print "enter D"

super(D, self).\_\_init\_\_()

print "leave D"

class E(B, C):

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

print "enter E"

super(E, self).\_\_init\_\_() # change

print "leave E"

class F(E, D):

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

print "enter F"

super(F, self).\_\_init\_\_() # change

print "leave F"

f = F() result:

enter F

enter E

enter B

enter C

enter D

enter A

leave A

leave D

leave C

leave B

leave E

leave F

明显地，F的初始化不仅完成了所有的父类的调用，而且保证了每一个父类的初始化函数只调用一次。

　　再看类结构：

object

/ \

/ A

| / \

B-1 C-2 D-2

\ / /

E-1 /

\ /

F

E-1,D-2是F的父类，其中表示E类在前，即F（E，D）。

所以初始化顺序可以从类结构图来看出 ： F－>E->B -->C --> D --> A

由于C，D有同一个父类，因此会先初始化D再是A。

三、延续的讨论

　　我们再重新看上面的类体系图，如果把每一个类看作图的一个节点，每一个从子类到父类的直接继承关系看作一条有向边，那么该体系图将变为一个有向图。不能发现mro的顺序正好是该有向图的一个拓扑排序序列。

　　从而，我们得到了另一个结果——Python是如何去处理多继承。支持多继承的传统的面向对象程序语言（如C++）是通过虚拟继承的方式去实现多继承中父类的构造函数被多次调用的问题，而Python则通过mro的方式去处理。

　　但这给我们一个难题：对于提供类体系的编写者来说，他不知道使用者会怎么使用他的类体系，也就是说，不正确的后续类，可能会导致原有类体系的错误，而且这样的错误非常隐蔽的，也难于发现。

四、小结

　　1. super并不是一个函数，是一个类名，形如super(B, self)事实上调用了super类的初始化函数，  
       产生了一个super对象；  
　　2. super类的初始化函数并没有做什么特殊的操作，只是简单记录了类类型和具体实例；  
　　3. super(B, self).func的调用并不是用于调用当前类的父类的func函数；  
　　4. Python的多继承类是通过mro的方式来保证各个父类的函数被逐一调用，而且保证每个父类函数  
       只调用一次（如果每个类都使用super）；  
　　5. 混用super类和非绑定的函数是一个危险行为，这可能导致应该调用的父类函数没有调用或者一  
       个父类函数被调用多次。

## 28. Python range和xrange的区别和联系

range产生的是一个列表，而xrange产生的是一个类似迭代器的。

所以对于较大的集合时候，xrange比range性能好。

因为range一次把所以数据都返回，而xrange每次调用返回其中的一个值