## 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

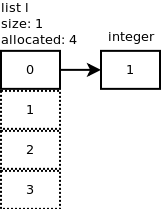
list\_resize:

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

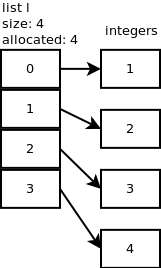
new\_allocated += newsize = 3 + 1 = 4

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

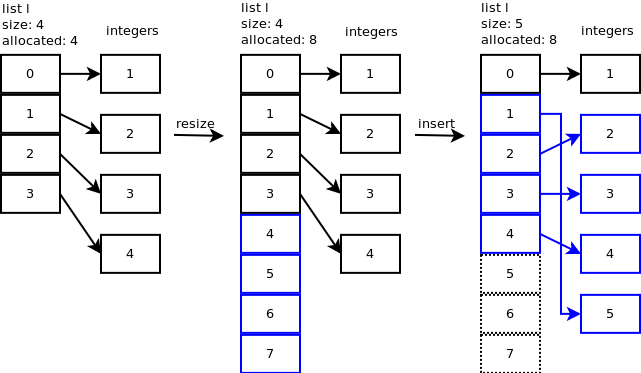
ins1:

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:

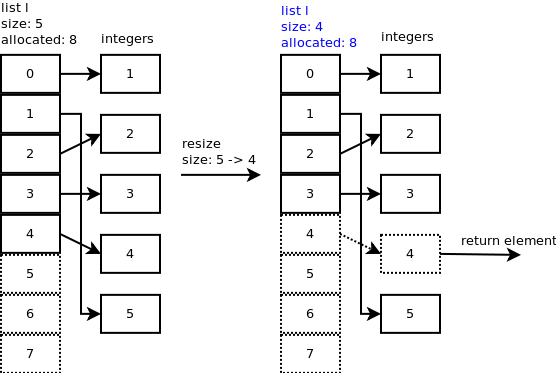
return null

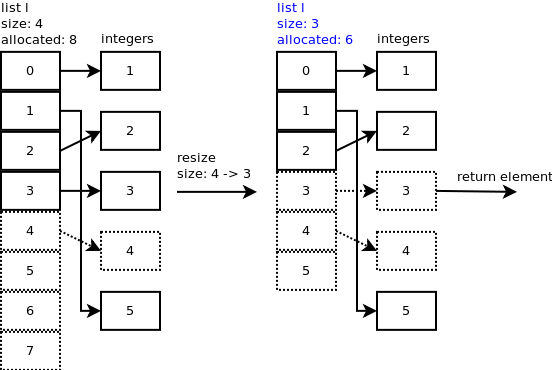
resize list with size 5 - 1 = 4. 4 is not less than 8/2 so no shrinkage

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

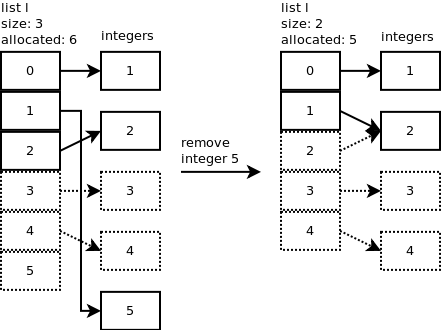
list\_ass\_slice:

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》