

# Python list implementation



**Laurent Luce** written 10 years ago



This post describes the CPython implementation of the list object. CPython is the most used Python implementation.

Lists in Python are powerful and it is interesting to see how they are implemented internally.

Following is a simple Python script appending some integers to a list and printing them.

```
01 >>> l = []
02 >>> l.append(1)
03 >>> l.append(2)
04 >>> l.append(3)
05 >>> l
06 [1, 2, 3]
07 >>> for e in l:
08 ...     print e
09 ...
10 1
11 2
12 3
```

As you can see, lists are iterable.

## List object C structure

A list object in CPython is represented by the following C structure. ob\_item is a list of pointers to the list elements. allocated is the number of slots allocated in memory.

```
1 typedef struct {
2     PyObject_VAR_HEAD
3     PyObject **ob_item;
4     Py_ssize_t allocated;
5 } PyListObject;
```

## List initialization

Let's look at what happens when we initialize an empty list. e.g. l = [].

```
1 arguments: size of the list = 0
2 returns: list object = []
3 PyListNew:
4     nbytes = size * size of global Python object = 0
5     allocate new list object
6     allocate list of pointers (ob_item) of size nbytes = 0
7     clear ob_item
8     set list's allocated var to 0 = 0 slots
9     return list object
```

It is important to notice the difference between allocated slots and the size of the list. The size of a list is the same as len(l). The number of allocated slots is what has been allocated in memory. Often, you will see that allocated can be greater than size. This is to avoid needing calling realloc each time a new elements is appended to the list. We will see more about that later.

## Append

We append an integer to the list: l.append(1). What happens? The internal C function app1() is called:

```
1 arguments: list object, new element
2 returns: 0 if OK, -1 if not
3 app1:
4     n = size of list
```

### Recent Posts

- [Least frequently used cache eviction scheme with complexity O\(1\) in Python](#)
- [Cambridge city geospatial statistics](#)
- [Massachusetts Census 2010 Towns maps and statistics using Python](#)
- [Python, Twitter statistics and the 2012 French presidential election](#)
- [Twitter sentiment analysis using Python and NLTK](#)
- [Python dictionary implementation](#)
- [Python string objects implementation](#)
- [Python integer objects implementation](#)
- [Python and cryptography with pycrypto](#)
- [Python list implementation](#)

### Search



### Meta

- [Log in](#)
- [Entries feed](#)
- [Comments feed](#)
- [WordPress.org](#)

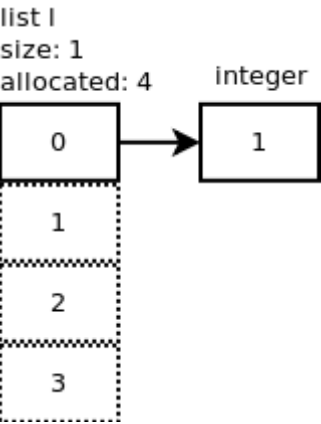
```
5 | call list_resize() to resize the list to size n+1 = 0 + 1 =
6 | 1
7 | list[n] = list[0] = new element
8 | return 0
```

Let’s look at list\_resize(). It over-allocates memory to avoid calling list\_resize too many time.  
The growth pattern of the list is: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...

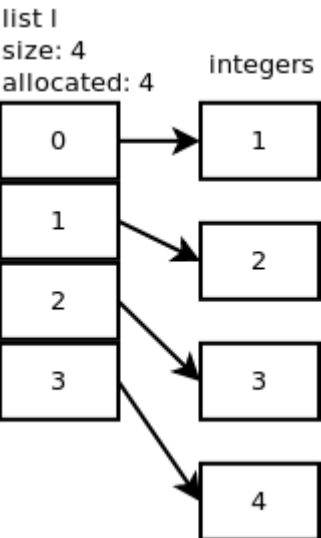
```
1 | arguments: list object, new size
2 | returns: 0 if OK, -1 if not
3 | list_resize:
4 |     new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6) = 3
5 |     new_allocated += newsize = 3 + 1 = 4
6 |     resize ob_item (list of pointers) to size new_allocated
7 |     return 0
```

4 slots are now allocated to contain elements and the first one is the integer 1. You can see on the following diagram that l[0] points to the integer object that we just appended. The dashed squares represent the slots allocated but not used yet.

Append operation amortized complexity is O(1).



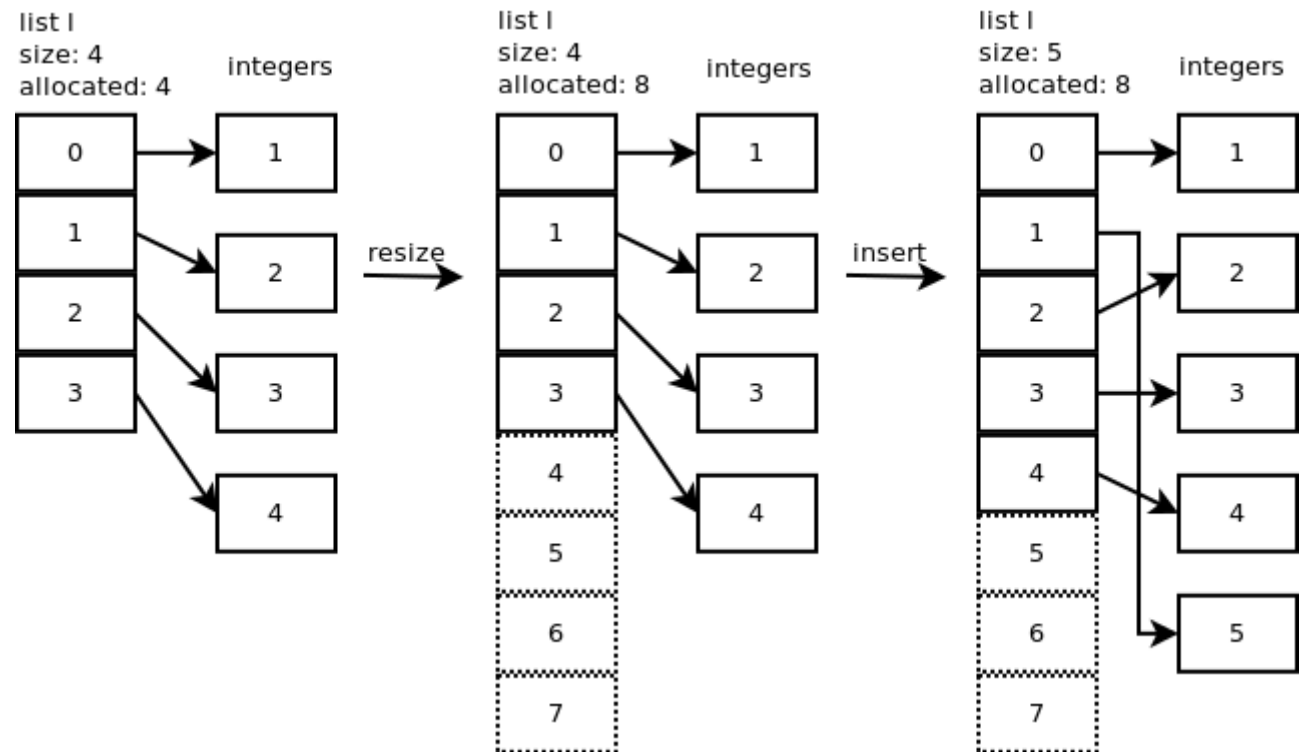
We continue by adding one more element: l.append(2). list\_resize is called with n+1 = 2 but because the allocated size is 4, there is no need to allocate more memory. Same thing happens when we add 2 more integers: l.append(3), l.append(4). The following diagram shows what we have so far.



## Insert

Let’s insert a new integer (5) at position 1: l.insert(1,5) and look at what happens internally. ins1() is called:

```
1 | arguments: list object, where, new element
2 | returns: 0 if OK, -1 if not
3 | ins1:
4 |     resize list to size n+1 = 5 -> 4 more slots will be
5 |     allocated
6 |     starting at the last element up to the offset where, right
7 |     shift each element
8 |     set new element at offset where
9 |     return 0
```



The dashed squares represent the slots allocated but not used yet. Here, 8 slots are allocated but the size or length of the list is only 5.

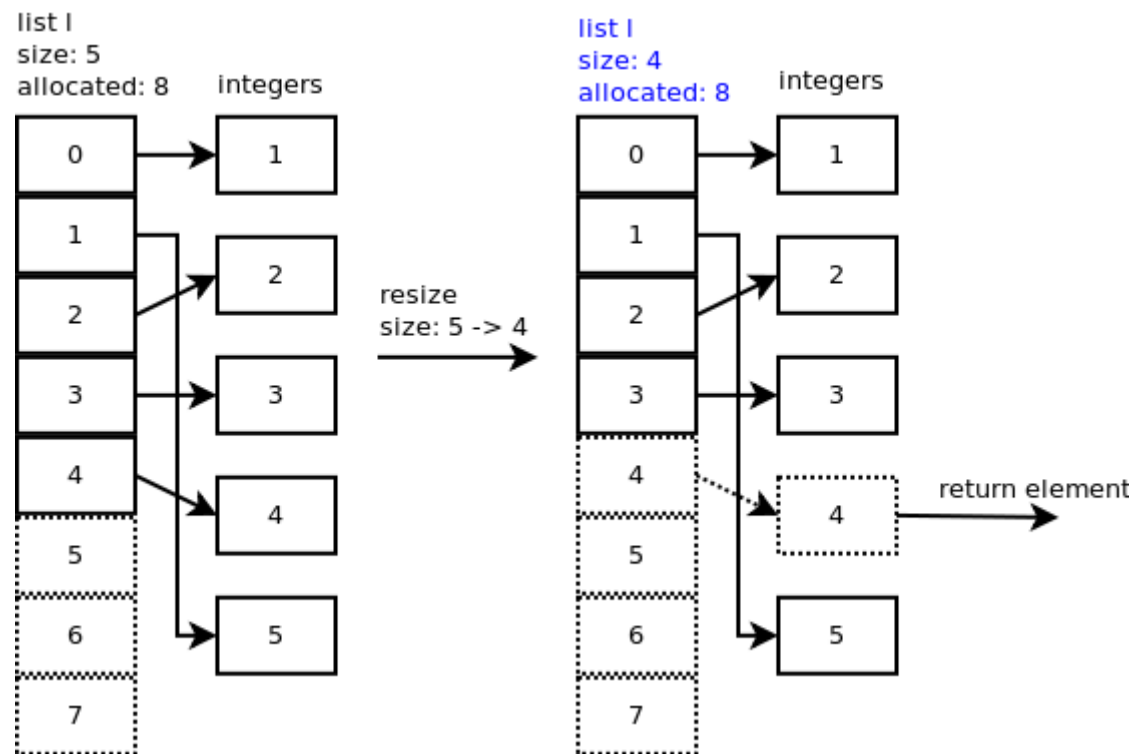
Insert operation complexity is  $O(n)$ .

## Pop

When you pop the last element: `l.pop()`, `listpop()` is called. `list_resize` is called inside `listpop()` and if the new size is less than half of the allocated size then the list is shrunk.

```
1 arguments: list object
2 returns: element popped
3 listpop:
4     if list empty:
5         return null
6     resize list with size 5 - 1 = 4. 4 is not less than 8/2 so
no shrinkage
7     set list object size to 4
8     return last element
```

Pop operation complexity is  $O(1)$ .

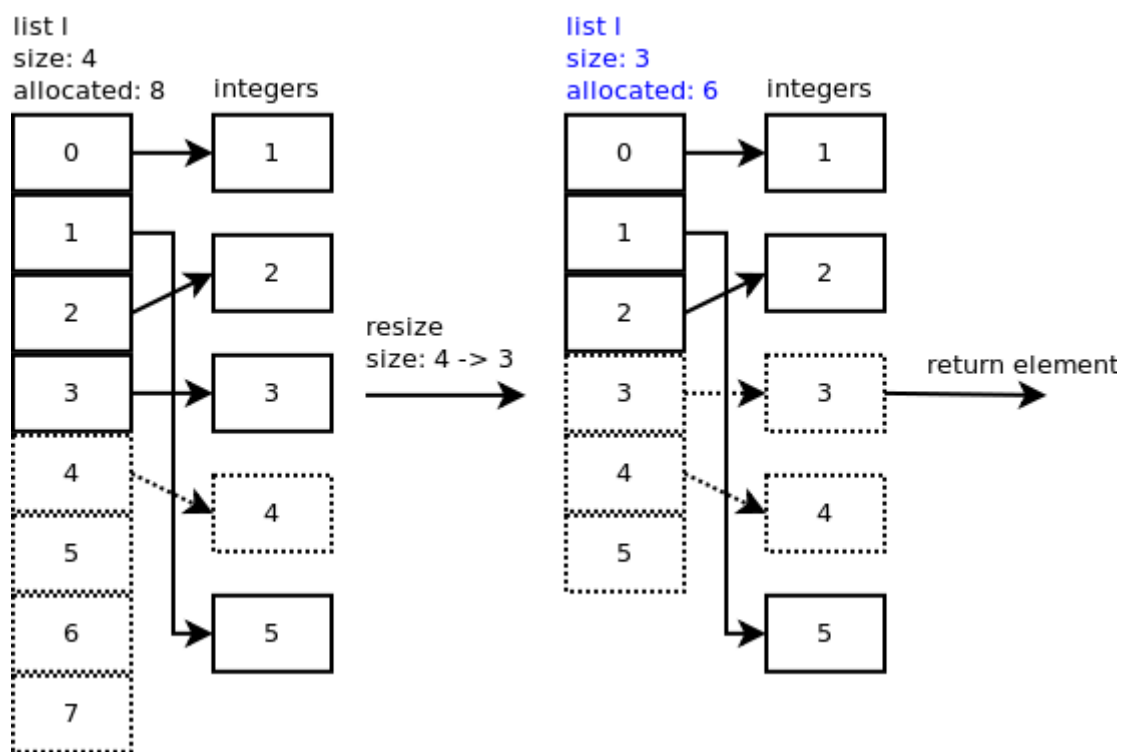


You can observe that slot 4 still points to the integer but the important thing is the size of the list which is now 4.

Let's pop one more element. In `list_resize()`,  $size - 1 = 4 - 1 = 3$  is less than half of the allocated slots so the list is shrunk to 6 slots and the new size of the list is now 3.

You can observe that slot 3 and 4 still point to some integers but the important thing is the size of the list which is now 3.





## Remove

Python list object has a method to remove a specific element: `l.remove(5)`. `listremove()` is called.

```

1 arguments: list object, element to remove
2 returns none if OK, null if not
3 listremove:
4     loop through each list element:
5         if correct element:
6             slice list between element's slot and element's slot
7 + 1             return none
8             return null

```

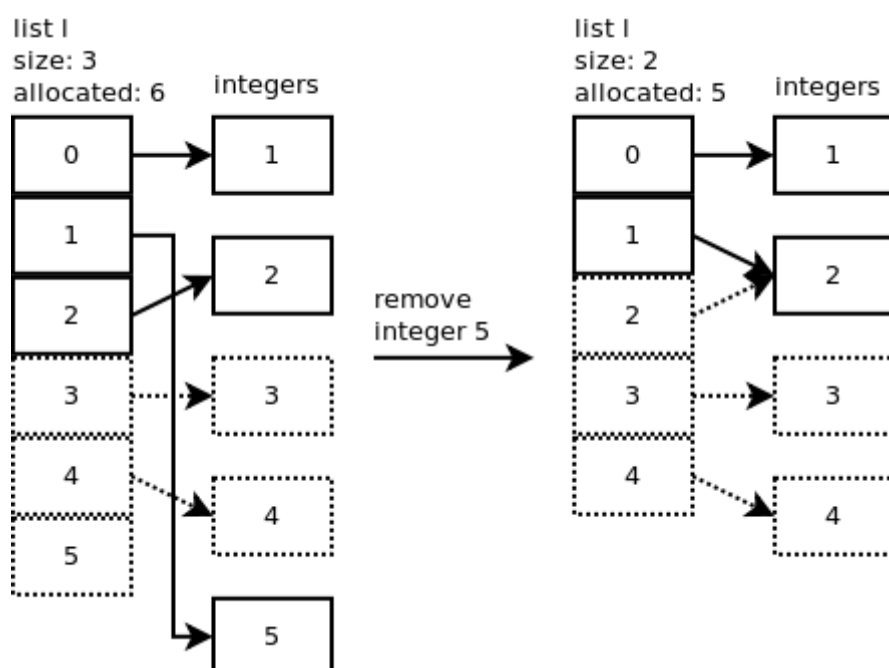
To slice the list and remove the element, `list_ass_slice()` is called and it is interesting to see how it works. Here, low offset is 1 and high offset is 2 as we are removing the element 5 at position 1.

```

1 arguments: list object, low offset, high offset
2 returns: 0 if OK
3 list_ass_slice:
4     copy integer 5 to recycle list to dereference it
5     shift elements from slot 2 to slot 1
6     resize list to 5 slots
7     return 0

```


Remove operation complexity is  $O(n)$ .



That's it for now. I hope you enjoyed the article. Please write a comment if you have any feedback. If you need help with a project written in Python or with building a new web service, I am available as a freelancer: [LinkedIn profile](#). Follow me on Twitter [@laurentluce](#).


*Last modified: May 30, 2020*

Author




[Laurent Luce](#)

Comments




[Sundaram](#) commented 5 years ago

Nice write up, helped me understand the list internals. Thank you!



[Malhar Vora](#) commented 6 years ago

Very nice and informative article. Thanks Laurent.




[winnie](#) commented 6 years ago

Hello, Laurent.

You wrote following:  
>>You can observe that slot 4 still points to the integer but the important thing is the size of the list which is now 4.


Does that mean that Integer 4 will still be accessible to GC and won't be collected till list will be really shrinked?



[Laurent Luce](#) commented 6 years ago

Author

@winnie: The integer 4 object will be collected when there is no reference to it. That means when there is nothing pointing to it in the list and when there are no references to it outside the list.




[arete](#) commented 7 years ago

hii Laurent Luce'

Most used feature in python

Really exciting reading through without a doubt, thank you a great deal I found a write-up in relation to record complexity, however these kind of memory pics are actually beneficial from the point of view of rendering.



[Laurent](#) commented 7 years ago

Question just for the sake of understanding the internals: does it make any difference between calling `l += [4]` and `l.append(4)` ?

It looks like the former is a slightly heavier (albeit not significantly) as it's creating a new list and merging the two lists whereas the latter directly calls `INPLACE_ADD` (from what `dis.dis` is saying)



Leo Dirac commented 7 years ago

I don't see how append can always be  $O(1)$ . When resize is needed, sometimes it will need to copy the old list of pointers into the newly allocated space, right? The C function realloc can sometimes just expand the block of memory in place, but sometimes needs to copy it. I think append will be  $O(N)$  sometimes, with an amortized rate something less than that. I don't exactly follow the resize formula, but it looks like it's growing quadratically rather than exponentially which means the amortized cost would be  $O(\sqrt{N})$  rather than  $O(\log N)$ .



Laurent Luce commented 7 years ago

Author

@Leo: From the Python wiki on operations time complexity: "These operations rely on the Amortized part of Amortized Worst Case. Amortized worst case of append is  $O(1)$ . Individual actions may take surprisingly long, depending on the history of the container. Internally, a list is represented as an array; the largest costs come from growing beyond the current allocation size (because everything must move), or from inserting or deleting somewhere near the beginning (because everything after that must move). If you need to add/remove at both ends, consider using a collections.deque instead."



dan commented 8 years ago

tem: agreed. I think the python wiki article can be misleading if you don't read carefully. pop(0) becomes equivalent to remove, which is  $O(n)$  for a basic list.



Jan commented 9 years ago

Very interesting reading indeed, thanks a lot 😊 I found an article about list complexity, but these memory pictures are really helpful from the point of view of implementation



Ganesh commented 9 years ago

Can you explain how is list implemented so as it can hold different datatypes.



Laurent Luce commented 9 years ago

Author

A list object contains a list of pointers to PyObjects. A PyObject contains a pointer to the corresponding type object. See object.h in the Python source code for more details.



tem commented 10 years ago

L.pop([index]) -> item — remove and return item at index.

I'm guessing pop is  $O(n)$  if you pop the first element.



Ivan commented 10 years ago

Hi,

Could you give us an idea of the running time of these operations?

$O(1)$  for append ?

$O(n)$  for insert ?



And what would be the total running time to append 1000 elements to a [] taking into account operations wasted on growing the list?



Laurent Luce commented 10 years ago

Author

@Ivan: I updated the article with the following information. Append is  $O(1)$ , Insert is  $O(n)$ , Pop is  $O(1)$ , Remove is  $O(n)$ . Sort is  $O(n \log n)$ . You can see more of those [here](#).

Appending  $n$  elements is  $O(n)$ . For  $n = 1000$ , the list is going to be resized 27 times. The growth pattern is the following: 4, 8, 16, 25, 35, 46, 58, 72, 88, 106, 126, 148, 173, 201, 233, 269, 309, 354, 405, 462, 526, 598, 679, 771, 874, 990, 1120. This means 27 calls to realloc. It doesn't change the complexity. It is still  $O(n)$ .



klonuo commented 10 years ago

Very interesting article

I read it in my RSS reader as part of "Planet Python" feeds and was then "forced" to launch Firefox and see your blog 😊

I hope for other interesting post in future

Cheers



TechnoBits.net commented 10 years ago

Very good article, uncovers the most used feature in python.

Comments are closed.

- ◀ [Solving mazes using Python: Simple recursivity and A\\* search](#)
- ▶ [Python and cryptography with pycrypto](#)