



数据结构与算法 (Python版)

Python数据类型的性能 (下)

陈斌 北京大学 gischen@pku.edu.cn

list.pop的计时试验

❖ 我们注意到pop这个操作

pop()从列表末尾移除元素, $O(1)$

pop(i)从列表中部移除元素, $O(n)$

❖ 原因在于Python所选择的实现方法

从中部移除元素的话, 要把移除元素后面的元素全部向前挪位复制一遍, 这个看起来有点笨拙

但这种实现方法能够保证列表按索引取值和赋值的操作很快, 达到 $O(1)$

这也算是一种对常用和不常用操作的折衷方案

list.pop的计时试验

- ❖ 为了验证表中的大O数量级，我们把两种情况下的pop操作来实际计时对比
相对同一个大小的list，分别调用pop()和pop(0)
- ❖ 对不同大小的list做计时，期望的结果是
pop()的时间不随list大小变化，pop(0)的时间随着list变大而变长

```
import timeit
popzero = timeit.Timer("x.pop(0)", "from __main__ import x")
popend = timeit.Timer("x.pop()", "from __main__ import x")
```

list.pop的计时试验

❖ 首先我们看对比

对于长度2百万的列表，执行1000次

`pop()`时间是0.0007秒

`pop(0)`时间是0.8秒

相差1000倍

```
>>> x = list(range(2000000))
>>> popzero.timeit(number=1000)
0.7688910461739789
>>> x = list(range(2000000))
>>> popend.timeit(number=1000)
0.0007347123802041722
```

list.pop的计时试验

❖ 我们通过改变列表的大小来测试两个操作的增长趋势

```
import timeit
popzero = timeit.Timer("x.pop(0)", "from __main__ import x")
popend = timeit.Timer("x.pop()", "from __main__ import x")
print "pop(0)    pop()"
for i in range(1000000,100000001,1000000):
    x = list(range(i))
    pt = popend.timeit(number=1000)
    x = list(range(i))
    pz = popzero.timeit(number=1000)
    print "%15.5f, %15.5f" %(pz,pt)
```

list.pop的计时试验

❖ 我们通过改变列表的大小来测试两个操作的增长趋势

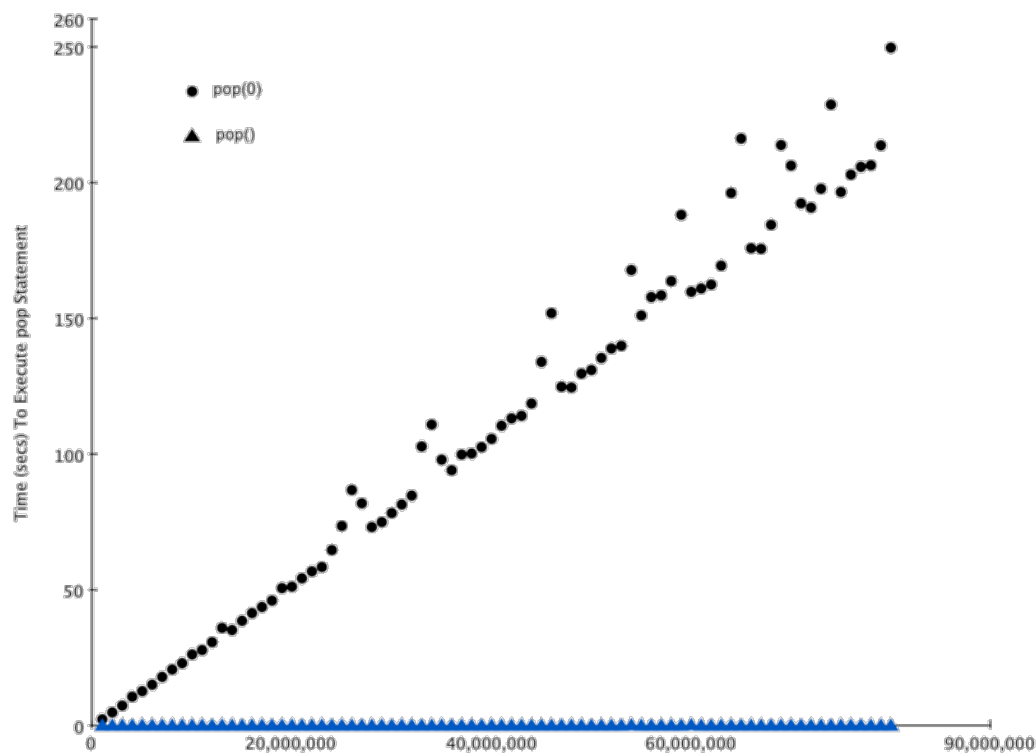
```
>>>
pop(0)    pop()
0.23149,  0.00078
0.68661,  0.00020
1.43575,  0.00045
2.00506,  0.00027
2.71711,  0.00032
3.32652,  0.00030
4.03600,  0.00032
4.56179,  0.00026
5.17211,  0.00034
5.75793,  0.00025
6.28499,  0.00028
6.63129,  0.00033
7.15702,  0.00027
```

list.pop的计时试验

❖ 将试验数据画成图表，可以看出增长趋势

pop()是平坦的常数

pop(0)是线性增长的趋势



dict数据类型

❖ 字典与列表不同，根据关键码（key）找到数据项，而列表是根据位置（index）

最常用的取值get和赋值set，其性能为 $O(1)$

另一个重要操作contains(in)是判断字典中是否存在某个关键码（key），这个性能也是 $O(1)$

operation	Big-O Efficiency
copy	$O(n)$
get item	$O(1)$
set item	$O(1)$
delete item	$O(1)$
contains (in)	$O(1)$
iteration	$O(n)$

list和dict的in操作对比

- ❖ 设计一个性能试验来验证list中检索一个值，以及dict中检索一个值的计时对比
- 生成包含连续值的list和包含连续关键码key的dict，用随机数来检验操作符in的耗时。

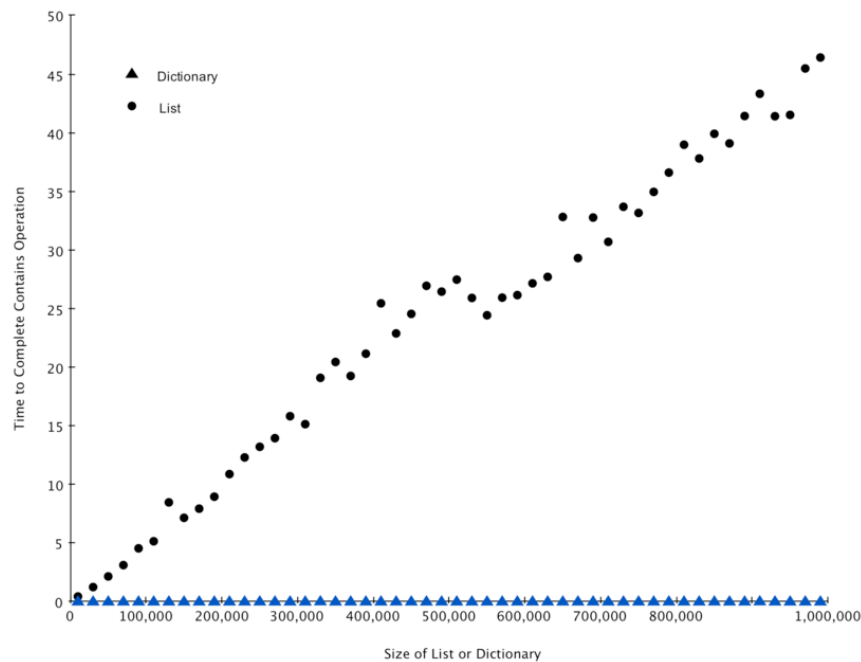
```
import timeit
import random

for i in range(10000,1000001,20000):
    t = timeit.Timer("random.randrange(%d) in x"%i,
                     "from __main__ import random,x")
    x = list(range(i))
    lst_time = t.timeit(number=1000)
    x = {j:None for j in range(i)}
    d_time = t.timeit(number=1000)
    print("%d,%10.3f,%10.3f" % (i, lst_time, d_time))
```

list和dict的in操作对比

- ❖ 可见字典的执行时间与规模无关，是常数
- ❖ 而列表的执行时间则随着列表的规模加大而线性上升

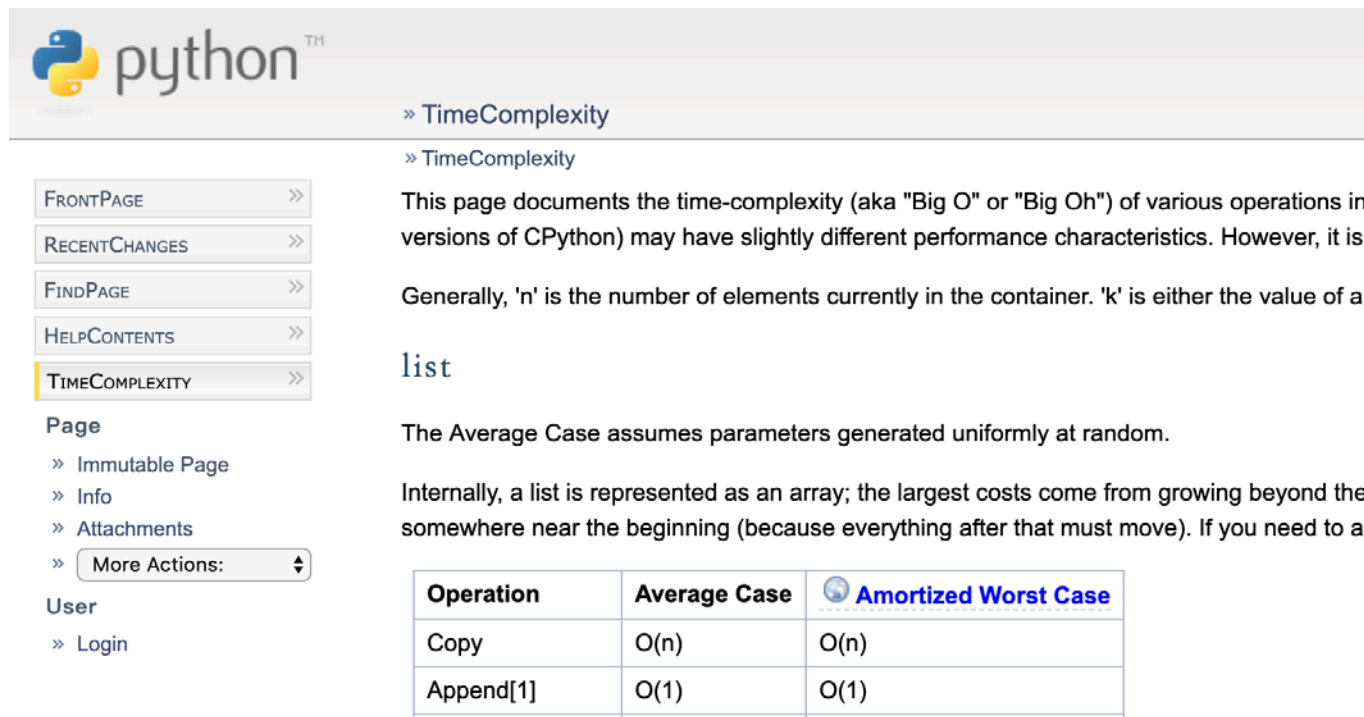
10000,	0.062,	0.001
30000,	0.189,	0.001
50000,	0.329,	0.001
70000,	0.440,	0.001
90000,	0.582,	0.001
110000,	0.710,	0.001
130000,	0.840,	0.001
150000,	0.951,	0.001
170000,	1.077,	0.001
190000,	1.227,	0.001
210000,	1.377,	0.001
230000,	1.499,	0.001
250000,	1.618,	0.001
270000,	1.738,	0.001
290000,	1.892,	0.001
310000,	2.055,	0.001




更多Python数据类型操作复杂度

❖ Python官方的算法复杂度网站：

<https://wiki.python.org/moin/TimeComplexity>



The screenshot shows the Python Wiki page for TimeComplexity. On the left is a sidebar with navigation links: FRONTPAGE, RECENTCHANGES, FINDPAGE, HELPCONTENTS, and TIMECOMPLEXITY (highlighted). Below these are links for Page (Immutable Page, Info, Attachments, More Actions) and User (Login). The main content area has a breadcrumb trail: » TimeComplexity. The text explains that the page documents the time-complexity (aka "Big O" or "Big Oh") of various operations in versions of CPython, noting that performance characteristics may vary. It defines 'n' as the number of elements in the container and 'k' as a value. A section for 'list' states that the Average Case assumes parameters generated uniformly at random. It explains that internally, a list is represented as an array, and the largest costs come from growing beyond the end, requiring elements to move. A table follows, comparing the Average Case and Amortized Worst Case for Copy and Append[1] operations.

Operation	Average Case	 Amortized Worst Case
Copy	$O(n)$	$O(n)$
Append[1]	$O(1)$	$O(1)$