Is there a way to circumvent Python list.append() becoming progressively slower in a loop as the list grows?

Asked 10 years, 6 months ago Active 1 year, 3 months ago Viewed 38k times

54

27

I have a big file I'm reading from, and convert every few lines to an instance of an Object.

Since I'm looping through the file, I stash the instance to a list using list.append(instance), and then continue looping.

This is a file that's around ~100MB so it isn't too large, but as the list grows larger, the looping slows down progressively. (I print the time for each lap in the loop).

This is not intrinsic to the loop ~ when I print every new instance as I loop through the file, the program progresses at constant speed ~ it is only when I append them to a list it gets slow.

My friend suggested disabling garbage collection before the while loop and enabling it afterward & making a garbage collection call.

Did anyone else observe a similar problem with list.append getting slower? Is there any other way to circumvent this?

I'll try the following two things suggested below.

(1) "pre-allocating" the memory ~ what's the best way to do this? (2) Try using deque

Multiple posts (see comment by Alex Martelli) suggested memory fragmentation (he has a large amount of available memory like I do) ~ but no obvious fixes to performance for this.

To replicate the phenomenon, please run the test code provided below in the answers and assume that the lists have useful data.

gc.disable() and gc.enable() helps with the timing. I'll also do a careful analysis of where all the time is spent.

python class list performance append Edit tags

edited Mar 19 '10 at 21:53

asked Mar 18 '10 at 22:30

Deniz

1,291 2 14 20

2 How many items end up in the list? Show us some code. I suspect you may be doing something you don't realize and we're not getting the whole story. – FogleBird Mar 18 '10 at 23:30

9 S.Lott is always questioning the motives instead of just answering the question straight-forwardly. (Sometimes he may be justified but it's usually just annoying, like in this case.) – FogleBird Mar 18 '10 at 23:52

4 @FogleBird, Understanding the problem we are tasked to help with is *always* important. A large portion of the questions here—possibly a majority—are XY problems. Helping people, solving problems, and making better programs are all better than blindly answering questions. – Mike Graham Mar 19 '10 at 0:13

3 I don't have a philosophy of not understanding the underlying problem. Even on this question I asked for clarification in the form of code. What I don't like is frequently and condescendingly questioning people, particularly when they obviously aren't asking n00bish questions. – FogleBird Mar 19 '10 at 0:44

3 When responding to a comment in your own question, please *do* respond in a comment. It's impossible to follow along when answers to questions in comments aren't in a comment after it. If it's significant information, do update the question as well, of course. – Glenn Maynard Mar 19 '10 at 21:23

8 Answers

Active Oldest Votes

96

The poor performance you observe is caused by a bug in the Python garbage collector in the version you are using. Upgrade to Python 2.7, or 3.1 or above to regain the amoritized 0(1) behavior expected of list appending in Python.

If you cannot upgrade, disable garbage collection as you build the list and turn it on after you finish.

(You can also tweak the garbage collector's triggers or selectively call collect as you progress, but I do not explore these options in this answer because they are more complex and I suspect your use case is amenable to the above solution.)

Background:

See: https://bugs.python.org/issue4074 and also https://docs.python.org/release/2.5.2/lib/module-gc.html

The reporter observes that appending complex objects (objects that aren't numbers or strings) to a list slows linearly as the list grows in length.

The reason for this behavior is that the garbage collector is checking and rechecking every object in the list to see if they are eligible for garbage collection. This behavior causes the linear increase in time to add objects to a list. A fix is expected to land in py3k, so it should not apply to the interpreter you are using.

Test:

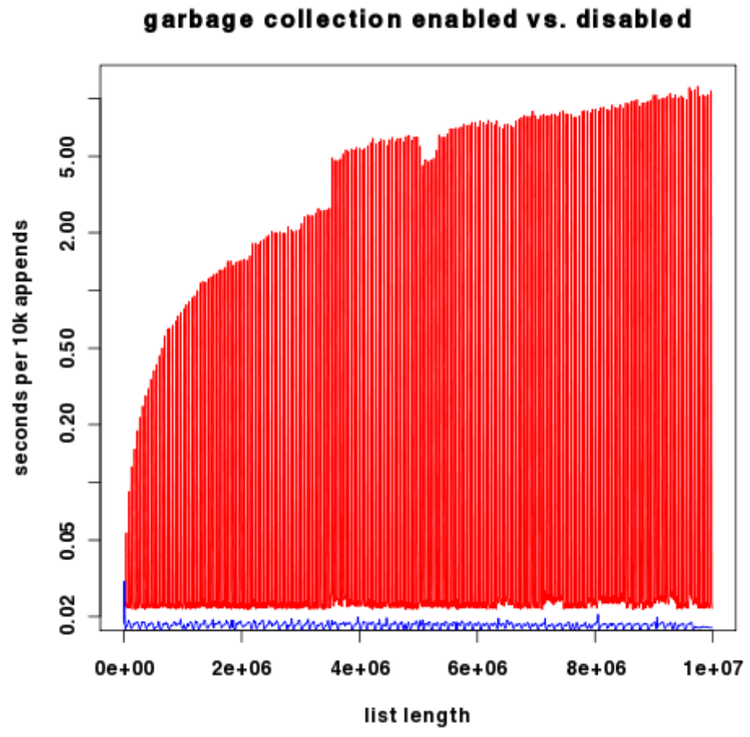
I ran a test to demonstrate this. For 1k iterations I append 10k objects to a list, and record the runtime for each iteration. The overall runtime difference is immediately obvious. With garbage collection disabled during the inner loop of the test, runtime on my system is 18.6s. With garbage collection enabled for the entire test, runtime is 899.4s.

This is the test:

```
import time
import gc

class A:
    def __init__(self):
        self.x = 1
        self.y = 2
        self.why = 'no reason'

def time_to_append(size, append_list, item_gen):
    t0 = time.time()
    for i in xrange(0, size):
        append_list.append(item_gen())
    return time.time() - t0

def test():
    x = []
    count = 10000
    for i in xrange(0,1000):
        print len(x), time_to_append(count, x, lambda: A())

def test_nogc():
    x = []
    count = 10000
    for i in xrange(0,1000):
        gc.disable()
        print len(x), time_to_append(count, x, lambda: A())
        gc.enable()
```

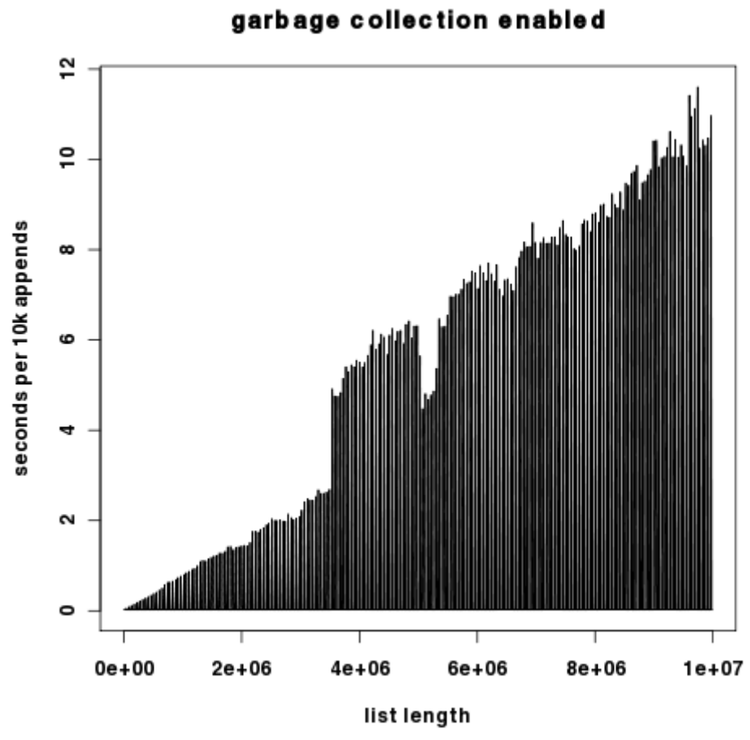
Full source: https://hypervolu.me/~erik/programming/python_lists/listtest.py.txt

Graphical result: Red is with gc on, blue is with gc off. y-axis is seconds scaled logarithmically.

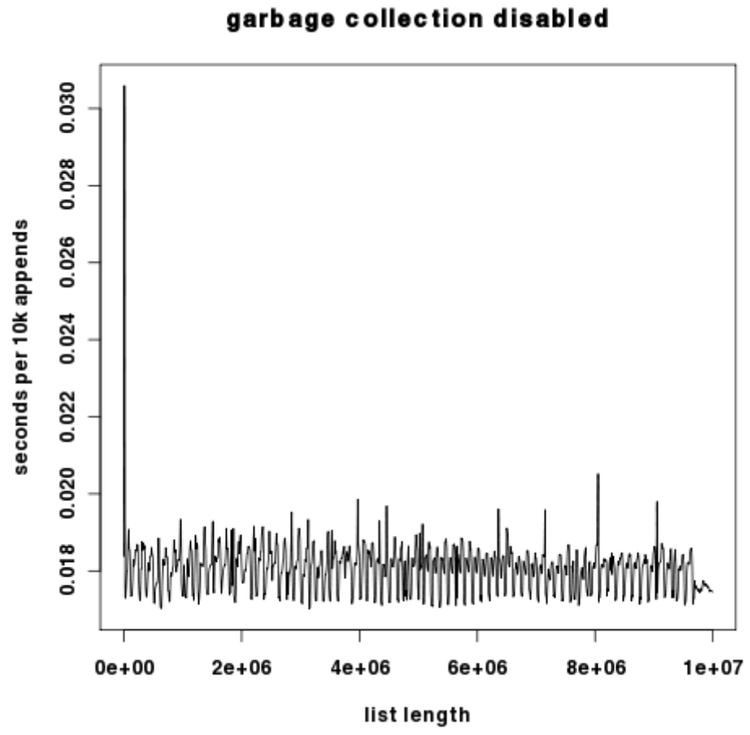


(source: hypervolu.me)

As the two plots differ by several orders of magnitude in the y component, here they are independently with the y-axis scaled linearly.



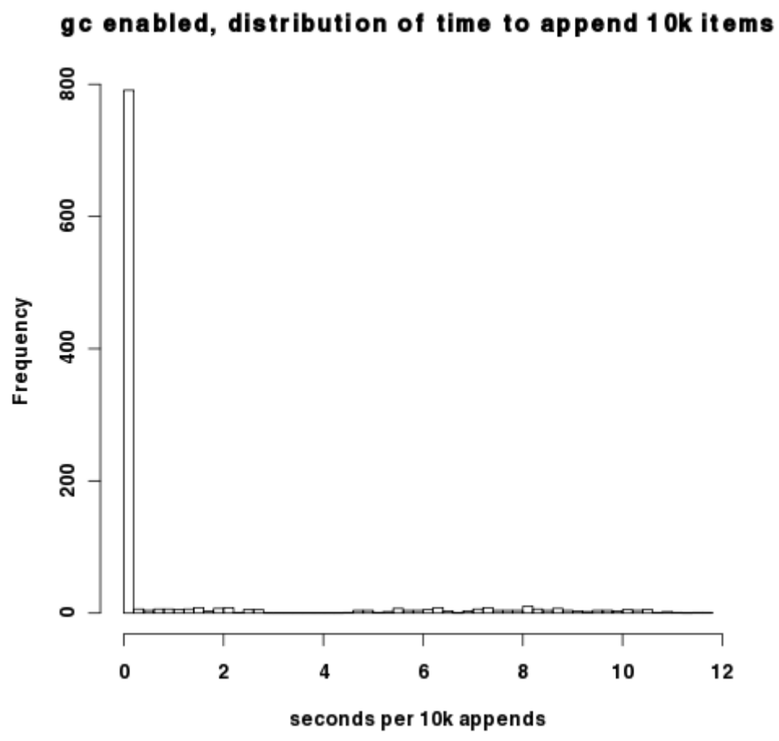
(source: hypervolu.me)



(source: hypervolu.me)

Interestingly, with garbage collection off, we see only small spikes in runtime per 10k appends, which suggests that Python's list reallocation costs are relatively low. In any case, they are many orders of magnitude lower than the garbage collection costs.

The density of the above plots make it difficult to see that with the garbage collector on, most intervals actually have good performance; it's only when the garbage collector cycles that we encounter the pathological behavior. You can observe this in this histogram of 10k append time. Most of the datapoints fall around 0.02s per 10k appends.



(source: hypervolu.me)

The raw data used to produce these plots can be found at http://hypervolu.me/~erik/programming/python_lists/

edited Jul 3 '19 at 7:06

Glorfindel
19k 11 62 85

answered Mar 19 '10 at 19:20

Erik Garrison
1,547 12 12

I didn't mean to add to the community wiki yet... but somehow the post has been added. Does anyone know how to remove it? – Erik Garrison Mar 19 '10 at 22:18

You can't undo community wiki. And it get's triggered automatically when you make eight edits (meta.stackexchange.com/questions/11740). (I feel your pain. Community wiki is so strange.) – Jon-Eric Mar 19 '10 at 22:26

1 Oh no! After more than a year of lurking on this site, I finally make a contribution and I get bitten by a hidden feature! I wouldn't have made so many small formatting edits had I known of this. – Erik Garrison Mar 20 '10 at 2:00

4 Such detail. This must have taken. Many hours. Thank you. – Paul Draper Mar 7 '13 at 5:47

4 Tested this in Python 3.4 and CPython and the bug is solved (with GC-990000 0.020383834838867188 & wihtout GC 9990000 0.013748407363891602) times are similar – Alex Punnen Jul 1 '15 at 6:36

Use a set instead then convert it to a list at the end

1

```
my_set=set()
with open(in_file) as f:
    # do your thing
    my_set.add(instance)

my_list=list(my_set)
my_list.sort() # if you want it sorted
```

I had the same problem and this solved the time problem by several orders.

answered Jul 30 '16 at 1:20

Kahiga
11 1

I encountered this problem while using Numpy arrays, created as follows:

1

```
import numpy
theArray = array([],dtype='int32')
```

Appending to this array within a loop took progressively longer as the array grew, which was a deal-breaker given that I had 14M appends to make.

The garbage collector solution outlined above sounded promising but didn't work.

What did work was creating the array with a predefined size as follows:

```
theArray = array(arange(limit),dtype='int32')
```

Just make sure that **limit** is bigger than the array you need.

You can then set each element in the array directly:

```
theArray[i] = val_i
```

And at the end, if necessary, you can remove the unused portion of the array

```
theArray = theArray[:i]
```

This made a HUGE difference in my case.

answered Nov 29 '11 at 9:50
Nathan Labenz
489 3 7

2 Why initialize the array with `{0,1,2,3,..,limit-1}` ? That is extremely inefficient if you are anyhow going to change the values afterwards. `numpy.zeros(limit)` is about 30x faster, `numpy.empty(limit)` is.... almost infinitely faster. – Bart Nov 1 '17 at 21:29

There is nothing to circumvent: **appending to a list is O(1) amortized.**

13

A list (in CPython) is an array at least as long as the list and up to twice as long. If the array isn't full, appending to a list is just as simple as assigning one of the array members (O(1)). Every time the array is full, it is automatically doubled in size. This means that on occasion an O(n) operation is required, but *it is only required every n operations*, and it is increasingly seldom required as the list gets big. O(n) / n ==> O(1). (In other implementations the names and details could potentially change, but the same time properties are bound to be maintained.)

Appending to a list already scales.

Is it possible that when the file gets to be big you are not able to hold everything in memory and you are facing problems with the OS paging to disk? Is it possible it's a different part of your algorithm that doesn't scale well?

edited Mar 19 '10 at 6:53

answered Mar 18 '10 at 23:37

Mike Graham
60.7k 12 84 119

Thanks for clarifying ~ so once in a while, I have an O(n) operation, but the next time the loop happens, it is back to the usual? I will do a detailed timing analysis of my code in the next few days and post again. – Deniz Mar 18 '10 at 23:50

Yes, the O(n) operations only occur occasionally. The number of appends between them grows at the same rate as n, so it averages out to having only a constant effect. – Mike Graham Mar 19 '10 at 5:17

256 GIGABYTES of ram or 128 or 64 but nothing lower than 64 GIGABYTES. For example: top - 02:36:31 up 36 days, 11:21, 7 users, load average: 0.84, 0.31, 0.11 Tasks: 274 total, 2 running, 272 sleeping, 0 stopped, 0 zombie Cpu(s): 6.2%us, 0.1%sy, 0.0%ni, 93.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st Mem: 132370600k total, 7819100k used, 124551500k free, 481084k buffers Swap: 2031608k total, 3780k used, 2027828k free, 5256144k cached – Deniz Mar 19 '10 at 6:37

Thanks Mike ~ below you mention that turning gc off helped you ~ is there any side effects to that? When should I turn it back on? – Deniz Mar 19 '10 at 6:53

@Deniz, Turning of the GC means that any reference cycles will not be discovered and collected. If what you are storing lots of strings, it looks like this doesn't help you. (At least from running a test along the lines of FogleBird's on my machine really quick, it looks like it does not suffer the penalty detected for creating millions of lists. Presumably Python knows it needn't search strings for reference cycles.) – Mike Graham Mar 19 '10 at 6:58

A lot of these answers are just wild guesses. I like Mike Graham's the best because he's right about how lists are implemented. But I've written some code to reproduce your claim and look into it further. Here are some findings.

6

Here's what I started with.

```
import time
x = []
for i in range(100):
    start = time.clock()
    for j in range(100000):
        x.append([])
    end = time.clock()
    print end - start
```

I'm just appending empty lists to the list `x`. I print out a duration for every 100,000 appends, 100 times. It does slow down like you claimed. (0.03 seconds for the first iteration, and 0.84 seconds for the last... quite a difference.)

Obviously, if you instantiate a list but don't append it to `x`, it runs way faster and doesn't scale up over time.

But if you change `x.append([])` to `x.append('hello world')`, there's no speed increase at all. The same object is getting added to the list 100 * 100,000 times.

What I make of this:

- The speed decrease has nothing to do with the size of the list. It has to do with the number of live Python objects.
- If you don't append the items to the list at all, they just get garbage collected right away and are no longer being managed by Python.
- If you append the same item over and over, the number of live Python objects isn't increasing. But the list does have to resize itself every once in a while. But this isn't the source of the performance issue.
- Since you're creating and adding lots of newly created objects to a list, they remain live and are not garbage collected. The slow down probably has something to do with this.

As far as the internals of Python that could explain this, I'm not sure. But I'm pretty sure the list data structure is not the culprit.

answered Mar 18 '10 at 23:50

FogleBird
62.1k 23 117 128

If you are forming a list of 100000 * 100 empty lists at 36 bytes a piece, that's 360 MB of lists you are trying to store on top of the big list and other overhead. Are you sure you aren't running into memory fragmentation or disk swapping? – Mike Graham Mar 19 '10 at 0:06

For figuring out which of two similar pieces of code is faster, timing is key (in Python, you'd use the `timeit` module to help you measure.) For determining how things scale, you have to understand mathematically what is going on to characterize the properties of your algorithm; timing can be important but is secondary. One of the most important things when timing an operation is to be sure you control your test to catch exactly what you want. – Mike Graham Mar 19 '10 at 0:09

It's not disk swapping or anything. It scales up right away and quite linearly. Not sure what point you're trying to make in your second comment? Feel free to run the examples I posted for yourself to see how it behaves. Do you disagree with my conclusions? – FogleBird Mar 19 '10 at 0:28

@FogleBird, How are you determining that you aren't suffering from memory fragmentation or swapping on your computer? These are usually things your OS does without really telling you much about it. If this is occurring, it makes this example represent that more than anything. The number of life Python objects isn't somehow inherently unscalable, and you haven't proven that it is. You've proven that something about this test slows down your computer. – Mike Graham Mar 19 '10 at 1:00

3 A variant of this benchmark measuring only the append time (not the time to create an empty list) shows it's very stable. One measuring just the time to create an empty list (and not the time to append) shows it growing... but only if the append is also there (present though not getting measured), otherwise the time to create an empty list (as the old ones get recycled), per se, is also stable (saving the lists to a preallocated list also makes list-creation slow down). Looks like memory fragmentation (costlier and costlier to allocate). (MacOSX 10.5, 4G RAM, tried Python 2.5 and 2.6). – Alex Martelli Mar 19 '10 at 2:07

|

2

Can you try ~~http://docs.python.org/release/2.5.2/lib/deque-objects.html~~ allocating expected number of required elements in your list? ? I would bet that list is a contiguous storage that has to be reallocated and copied every few iterations. (similar to some popular implementations of std::vector in c++)

EDIT: Backed up by http://www.python.org/doc/faq/general/#how-are-lists-implemented

edited Mar 18 '10 at 22:49

answered Mar 18 '10 at 22:39

Tomasz Zieliński
15k 7 52 70

How would I go about the allocation? I know how many items there are before hand, I could try this very easily. – Deniz Mar 18 '10 at 23:19

Not every few operations—every *n* operations. (The FAQ simplifies to "few" which hides the implementation of the "clever".) This allows `append` to have a complexity of O(1) amortized. – Mike Graham Mar 18 '10 at 23:42

I understand that it's increased *2. So it's O(1) with "spikes", right? – Tomasz Zieliński Mar 19 '10 at 9:50

-2

This post is hidden. It was deleted 10 years ago by the post author.

One thing you could try is to prepend the new value onto the beginning of the list, and then when the list is complete, do a reverse. I don't know Python syntax, but I'm sure you could do something like this.

This way you avoid having to traverse the list each time you append.

Maybe this is harder than I expected in Python.

OK. Use a dictionary with the key being a count that you increment every time in the loop. That way, no appending onto a list.

edited Mar 18 '10 at 22:47

answered Mar 18 '10 at 22:39

Matthew Farwell
56.8k 16 117 164

2 Prepending is not good for lists, only for containers like deques. Also, list knows it's length. – Tomasz Zieliński Mar 18 '10 at 22:44

Appending to a `list` is O(1). Prepending to a `list` is O(n). (A "list" in Python isn't a linked list, if that's what you were thinking.) – Mike Graham Mar 18 '10 at 23:41

1 Prepending is **BAD** for lists ... it will copy the whole list each time!! – John Machin Mar 18 '10 at 23:43

Cool. I learnt something today :-) Thanks for your help. – Matthew Farwell Mar 19 '10 at 7:51

comments disabled on deleted / locked posts / reviews

-1

This post is hidden. It was deleted 10 years ago by user124626.

Yea, you could unroll the loop. In other words try appending items i, i+1, i+2, i+3 at the same time.

answered Mar 18 '10 at 22:33

user124626

1 That won't help because the problem is list.append, not the loop: "This is not intrinsic to the loop ~ when I print every new instance as I loop through the file, the program progresses at constant speed ~ it is only when I append them to a list it gets slow." – Barthelemy Mar 18 '10 at 22:39

Youre right I must have been thinking of other applications where this had been useful, such as building a string or query, etc. – user124626 Mar 18 '10 at 22:50

comments disabled on deleted / locked posts / reviews