Making a flat list out of list of lists in Python [duplicate]

> **Possible Duplicates:**
> Flattening a shallow list in Python
> Comprehension for flattening a sequence of sequences?

I wonder whether there is a shortcut to make a simple list out of list of lists in Python.

I can do that in a for loop, but maybe there is some cool "one-liner"? I tried it with *reduce*, but I get an error.

Code

```
l = [[1,2,3],[4,5,6], [7], [8,9]]
reduce(lambda x,y: x.extend(y),l)
```

Error message

> Traceback (most recent call last): File "", line 1, in File "", line 1, in AttributeError: 'NoneType' object has no attribute 'extend'

python list

edited Nov 2 '10 at 22:51
Peter Mortensen
9,112 10 63 98

asked Jun 4 '09 at 20:30
Emma
3,610 3 10 9

marked as duplicate by S.Lott, Paolo Bergantino, David Z, SilentGhost, dF. Jun 4 '09 at 22:36

This question has been asked before and already has an answer. If those answers do not fully address your question, please ask a new question.

1 Duplicate: stackoverflow.com/questions/406121/…, stackoverflow.com/questions/457215/…, stackoverflow.com/questions/120886/… – S.Lott Jun 4 '09 at 20:39

2 There's an in-depth discussion of this here: rightfootin.blogspot.com/2006/09/more-on-python-flatten.html, discussing several methods of flattening arbitrarily nested lists of lists. An interesting read! – RichieHindle Jun 4 '09 at 20:41

44 Reopen this because it has the most concise answer of the duplicates, maybe it should be merged? – Peer Stritzinger Sep 12 '11 at 14:09

5 I agree with @PeerStritzinger, the top two answers in this one are more helpful than answers in the other three questions – Izkata Feb 17 '12 at 20:07

1 Some other answers are better but the reason yours fails is that the 'extend' method always returns None. For a list with length 2, it will work but return None. For a longer list, it will consume the first 2 args, which returns None. It then continues with None.extend(<third arg>), which causes this erro – mehtunguh Jun 11 '13 at 21:48

8 Answers

`[item for sublist in l for item in sublist]` is faster than the shortcuts posted so far. (`l` is the list to flatten.)

For evidence, as always, you can use the `timeit` module in the standard library:

```
$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99' '[item for sublist in l for item
in sublist]'
10000 loops, best of 3: 143 usec per loop
$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99' 'sum(l, [])'
1000 loops, best of 3: 969 usec per loop
$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99' 'reduce(lambda x,y: x+y,l)'
1000 loops, best of 3: 1.1 msec per loop
```

Explanation: the shortcuts based on `+` (including the implied use in `sum`) are, of necessity, `O(L**2)` when there are L sublists -- as the intermediate result list keeps getting longer, at each step a new intermediate result list object gets allocated, and all the items in the previous intermediate result must be copied over (as well as a few new ones added at the end). So (for simplicity and without actual loss of generality) say you have L sublists of I items each: the first I items are copied back and forth L-1 times, the second I items L-2 times, and so on; total number of copies is I times the sum of x for x from 1 to L excluded, i.e., `I * (L**2)/2` .

The list comprehension just generates one list, once, and copies each item over (from its original place of residence to the result list) also exactly once.

edited Jun 10 at 23:58
bcdan
1,065 1 5 21

answered Jun 4 '09 at 20:37
Alex Martelli
423k 71 794 1094

188 I tried a test with the same data, using `itertools.chain.from_iterable` : `$ python -mtimeit -s'from itertools import chain; l=[[1,2,3],[4,5,6], [7], [8,9]]*99' 'list(chain.from_iterable(l))'` . It runs a bit more than twice as fast as the nested list comprehension that's the fastest of the alternatives shown here. – intuited Oct 15 '10 at 1:21

89 I found the syntax hard to understand until I realized you can think of it exactly like nested for loops. for sublist in l: for item in sublist: yield item – Rob Crowell Jul 27 '11 at 16:43

10 @BorisChervenkov: Notice that I wrapped the call in `list()` to realize the iterator into a list. – intuited May 20 '12 at 22:56

34 [leaf for tree in forest for leaf in tree] might be easier to comprehend and apply. – John Mee Aug 29 '13 at 1:38

6 @Joel, actually nowadays `list(itertools.chain.from_iterable(l))` is best -- as noticed in other comments and Shawn's answer. – Alex Martelli Jan 4 at 15:40

You can use `itertools.chain()` :

```
>>> import itertools
>>> list2d = [[1,2,3],[4,5,6], [7], [8,9]]
>>> merged = list(itertools.chain(*list2d))
```

or, on Python >=2.6, use `itertools.chain.from_iterable()` which doesn't require unpacking the list:

```
>>> import itertools
>>> list2d = [[1,2,3],[4,5,6], [7], [8,9]]
>>> merged = list(itertools.chain.from_iterable(list2d))
```

This approach is arguably more readable than `[item for sublist in l for item in sublist]` and appears to be faster too:

```
[me@home]$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99;import itertools'
'list(itertools.chain.from_iterable(l))'
10000 loops, best of 3: 24.2 usec per loop
[me@home]$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99' '[item for sublist in l
for item in sublist]'
10000 loops, best of 3: 45.2 usec per loop
[me@home]$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99' 'sum(l, [])'
1000 loops, best of 3: 488 usec per loop
[me@home]$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99' 'reduce(lambda x,y:
x+y,l)'
1000 loops, best of 3: 522 usec per loop
[me@home]$ python --version
Python 2.7.3
```

edited Mar 12 '13 at 17:15

answered Jun 4 '09 at 21:06
Shawn Chin
37.7k 7 81 133

2 And if someone wants to know why other solutions based on `+` operator (concatenation) are inefficient, see How not to Flatten a List of Lists – Mathieu Larose Jun 26 '13 at 18:26

3 awesome, itertools are even faster than Martelli's lists – qarma Sep 14 '13 at 15:29

3 @ShawnChin BTW, piece of hardware you had when answering this question, my current workstation is half as fast and is been 4 years. – Manuel Gutierrez Sep 24 '13 at 15:18

@alexandre see docs.python.org/2/tutorial/… – Shawn Chin Jul 23 '14 at 20:15

2 The `*` is the tricky thing that makes `chain` less straightforward than the list comprehension. You have to know that chain only joins together the iterables passed as parameters, and the * causes the top-level list to be expanded into parameters, so `chain` joins together all those iterables, but doesn't descend further. I think this makes the comprehension more readable than the use of chain in this case. – Tim Dierks Sep 3 '14 at 14:13

```
>>> sum(l, [])
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note that only works on lists of lists. For lists of lists of lists, you'll need another solution.

edited Jun 4 '09 at 20:42

answered Jun 4 '09 at 20:35
Triptych
93.8k 19 109 147

46 What a cleverly implicit use of the overloaded (+) operator! – Nick Retallack Jun 4 '09 at 20:39

28 that's pretty neat and clever but I wouldn't use it because it's confusing to read. – andrewrk Jun 15 '10 at 18:55

4 @curious: This just sums the elements of iterable passed in the first argument, treating second argument as the initial value of the sum (if not given, `0` is used instead and this case will give you an error). Because you are summing nested lists, you actually get `[1,3]+[2,4]` as a result of `sum([[1,3],[2,4]],[])`, which is equal to `[1,3,2,4]`. – Tadeck Mar 23 '12 at 18:21

22 This is a Shlemiel the painter's algorithm joelonsoftware.com/articles/fog0000000319.html -- unnecessarily inefficient as well as unnecessarily ugly. – Mike Graham Apr 25 '12 at 18:24

4 The append operation on lists forms a `Monoid`, which is one of the most convenient abstractions for thinking of a `+` operation in a general sense (not limited to numbers only). So this answer deserves a +1 from me for (correct) treatment of lists as a monoid. *The performance is concerning though...* – ulidtko Dec 3 '14 at 10:35

@Nadia: You have to use much longer lists. Then you see the difference quite strikingly! My results for `len(l) = 1600`

```
A took 14.323 ms
B took 13.437 ms
C took 1.135 ms
```

where:

```
A = reduce(lambda x,y: x+y,l)
B = sum(l, [])
C = [item for sublist in l for item in sublist]
```

edited Jun 4 '09 at 21:07

community wiki
2 revs
jacob

It only gets worse and worse, as the algorithmic complexity of A and B are different (worse) than that of C. – Mike Graham Apr 25 '12 at 18:27

```
>>> l = [[1,2,3],[4,5,6], [7], [8,9]]
>>> reduce(lambda x,y: x+y,l)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The `extend()` method in your example modifies `x` instead of returning a useful value (which `reduce()` expects).

A faster way to do the `reduce` version would be

```
>>> import operator
>>> l = [[1,2,3],[4,5,6], [7], [8,9]]
>>> reduce(operator.add, l)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

edited Sep 24 '11 at 10:14
agf
60.9k 16 142 162

answered Jun 4 '09 at 20:35
Greg Hewgill
451k 98 797 992

3 `reduce(operator.add, l)` would be the correct way to do the `reduce` version. Built-ins are faster than lambdas. – agf Sep 24 '11 at 10:04

1 @agf here is how: * `timeit.timeit('reduce(operator.add, l)', 'import operator; l=[[1, 2, 3], [4, 5, 6, 7, 8], [1, 2, 3, 4, 5, 6, 7]]', number=10000)` **0.017956018447875977** * `timeit.timeit('reduce(lambda x, y: x+y, l)', 'import operator; l=[[1, 2, 3], [4, 5, 6, 7, 8], [1, 2, 3, 4, 5, 6, 7]]', number=10000)` **0.025218963623046875** – lukmdo Mar 20 '12 at 22:13

4 This is a Shlemiel the painter's algorithm joelonsoftware.com/articles/fog0000000319.html – Mike Graham Apr 25 '12 at 18:26

this can use only for `integers`. But what if list contains `string`? – Freddy Sep 11 at 7:16

@Freddy: The `operator.add` function works equally well for both lists of integers and lists of strings. – Greg Hewgill Sep 11 at 7:38

I take my statement back. sum is not the winner. Although it is faster when the list is small. But the performance degrades significantly with larger lists.

```
>>> timeit.Timer(
        '[item for sublist in l for item in sublist]',
        'l=[[1, 2, 3], [4, 5, 6, 7, 8], [1, 2, 3, 4, 5, 6, 7]] * 10000'
    ).timeit(100)
2.0440959930419922
```

The sum version is still running for more than a minute and it hasn't done processing yet!

For medium lists:

```
>>> timeit.Timer(
        '[item for sublist in l for item in sublist]',
        'l=[[1, 2, 3], [4, 5, 6, 7, 8], [1, 2, 3, 4, 5, 6, 7]] * 10'
    ).timeit()
20.126545906066895
>>> timeit.Timer(
        'reduce(lambda x,y: x+y,l)',
        'l=[[1, 2, 3], [4, 5, 6, 7, 8], [1, 2, 3, 4, 5, 6, 7]] * 10'
    ).timeit()
22.242258071899414
>>> timeit.Timer(
        'sum(l, [])',
        'l=[[1, 2, 3], [4, 5, 6, 7, 8], [1, 2, 3, 4, 5, 6, 7]] * 10'
    ).timeit()
16.449732065200806
```

Using small lists and timeit: number=1000000

```
>>> timeit.Timer(
        '[item for sublist in l for item in sublist]',
        'l=[[1, 2, 3], [4, 5, 6, 7, 8], [1, 2, 3, 4, 5, 6, 7]]'
    ).timeit()
2.4598159790039062
>>> timeit.Timer(
        'reduce(lambda x,y: x+y,l)',
        'l=[[1, 2, 3], [4, 5, 6, 7, 8], [1, 2, 3, 4, 5, 6, 7]]'
    ).timeit()
1.5289170742034912
>>> timeit.Timer(
        'sum(l, [])',
        'l=[[1, 2, 3], [4, 5, 6, 7, 8], [1, 2, 3, 4, 5, 6, 7]]'
    ).timeit()
1.0598428249359131
```

edited Dec 23 '13 at 9:14
devsaw
378 2 12

answered Jun 4 '09 at 20:46
Nadia Alramli
50.7k 12 125 137

13 for a truly miniscule list, e.g. one with 3 sublists, maybe -- but since sum's performance goes with O(N**2) while the list comprehension's goes with O(N), just growing the input list a little will reverse things -- indeed the LC will be "infinitely faster" than sum at the limit as N grows. I was responsible for designing sum and doing its first implementation in the Python runtime, and I still wish I had found a way to effectively restrict it to summing numbers (what it's really good at) and block the "attractive nuisance" it offers to people who want to "sum" lists;-). – Alex Martelli Jun 4 '09 at 21:07

Why do you use extend?

```
reduce(lambda x, y: x+y, l)
```

This should work fine.

edited Dec 31 '14 at 12:07
MERose
889 2 9 27

answered Jun 4 '09 at 20:38
Andrea Ambu
9,023 10 38 65

The reason your function didn't work: the extend extends array in-place and doesn't return it. You can still return x from lambda, using some trick:

```
reduce(lambda x,y: x.extend(y) or x, l)
```

Note: extend is more efficient than + on lists.

answered Jun 4 '09 at 20:47
Igor Krivokon
7,634 1 20 37

3 `extend` is better used as `newlist = []`, `extend = newlist.extend`, `for sublist in l: extend(l)` as it avoids the (rather large) overhead of the `lambda`, the attribute lookup on `x`, and the `or`. – agf Sep 24 '11 at 10:12

---

protected by Ashwini Chaudhary Jan 11 '13 at 13:22

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site.

Would you like to answer one of these unanswered questions instead?