MONDAY, SEPTEMBER 04, 2006

» More on python flatten

The standard (but for some unknown reason, not built-in) python **`flatten`** method looks something like this:

```
def flatten(l):
  out = []
  for item in l:
    if isinstance(item, (list, tuple)):
      out.extend(flatten(item))
    else:
      out.append(item)
  return out
```

This obviously runs into recursion errors pretty quickly for highly nested lists; what suprised me is that it can't grok even relatively shallow nesting, e.g., 10 levels deep:

```
a = []
for i in xrange(10):
  a = [a, i]
a = flatten(a)

Traceback (most recent call last):
  File "test.py", line 13, in ?
    a = flatten(a)
  File "test.py", line 5, in flatten
    out.extend(flatten(l))
  ...
  File "test.py", line 5, in flatten
    out.extend(flatten(l))
RuntimeError: maximum recursion depth exceeded
```

Someone smarter than I am (viz., Danny Yoo) wrote a better method using an iterator, which looks basically like this:

```
def iter_flatten(iterable):
  it = iter(iterable)
  for e in it:
    if isinstance(e, (list, tuple)):
      for f in iter_flatten(e):
        yield f
    else:
      yield e

a = []
for i in xrange(300):
  a = [a, i]
a = [i for i in iter_flatten(a)]
```

As you can see, this works with deeply nested arrays (up to 499 levels on my box). He also wrote a very opaque version using a form of tail recursion (via continuation passing),

About Me

Name: MonkeeSage
Location: San Antonio, Texas, United States

View my complete profile

Previous Posts

» No builtin flatten in python...
» Hyper-fatugly!
» Of Rocks and Reptiles...
» Favorite editor
» Doing it the Ruby Way™
» Write it in C or face 10,000 slow deaths!
» Kazehakase...ruby in your browser?!!
» ELER...Ruby-style
» A Matter of Semantics
» Non-Recursive Factorial

I Power Blogger

Contact me
Atom feed

which is "not meant to be read by humans". That version can handle lists nested as deeply as the system recursion limit (1000 on my box)! That's pretty cool (even though my brain implodes when I try to understand it, heh)!

Just for fun, I decided to see what I could come up with. Here is my offering:

```
def flatten(l, limit=1000, counter=0):
  for i in xrange(len(l)):
    if (isinstance(l[i], (list, tuple)) and
        counter < limit):
      for a in l.pop(i):
        l.insert(i, a)
        i += 1
      counter += 1
      return flatten(l, limit, counter)
  return l
```

Nothing fancy. It's about as fast as Yoo's continuation version, but it breaks at 499 levels like the iterator version (but the iterator version is slower and requires the extra list-comprehension syntax if used for assignment). I also added an (optional) **`limit`** argument, to specify the maximum number of levels to flatten (*ala* ruby's **`Array#flatten`**). Don't worry about the **`counter`** argument, that's just to internally track state across recursions.

Addendum: Wow! I just came across a wonderful version of **`flatten`** in a cookbook comment. I saw it a few days ago, but I didn't really think about it, just kind of thought "yeah, another recursive flatten method," but this one whoops all of the others for speed, nesting level support and elegance! This is from Mike C. Fletcher's BasicTypes library. The method looks something like this (I've altered it a bit—see comments below):

```
def flatten(l, ltypes=(list, tuple)):
    ltype = type(l)
    l = list(l)
    i = 0
    while i < len(l):
        while isinstance(l[i], ltypes):
            if not l[i]:
                l.pop(i)
                i -= 1
                break
            else:
                l[i:i + 1] = l[i]
        i += 1
    return ltype(l)

a = []
for i in xrange(2000):
  a = [a, i]
a = flatten(a)
```

Freakin' genius!

** Fixed for empty lists/tuples based on Noah's comment*
** Fixed again based on Greg's comment*
** Fixed yet again based on John Y's comment*

Addendum: I got to wondering how Mr. Fletcher's version would stack up against the built-in **`Array#flatten`** method in ruby. Granted, his doesn't have a flatten limit, and I think it would be kind of hard to add one, but then I have never really needed that feature. So here is the ruby version with times:

```
def flatten(l)
    i = 0
    while i < l.size
        while l[i].is_a? Array
            if l[i].empty?
                l.delete_at(i)
                i -= 1
                break
```

```
            else
                l[i...i+1] = l[i]
            end
        end
        i += 1
    end
    l
end
```

And now the times (tested with 1.8.6 final, best out of three). First, Fletcher's version:

```
p flatten(1500.times.inject { | m, i | m = [m, i] })

# time ruby test.rb
#
# real 0m0.038s
# user 0m0.028s
# sys 0m0.004s
```

Then the built-in version:

```
p 1500.times.inject { | m, i | m = [m, i] }.flatten

# time ruby test.rb
#
# real 0m0.027s
# user 0m0.016s
# sys 0m0.004s
```

Wow! That's pretty neat! A **`flatten`** implemented *in ruby* that is competitive with the C backend, heh! Now I know there are reasons for this, and like I said, Fletcher's version will just smash everything at every level, without regard. But still, it's always cool to find a bit of interpreted code that breaks out a can of Chuck Norris on the interpreter! (ruby is so manly it can *almost* beat *itself* up! heh!)

Labels: programming, python

posted by MonkeeSage @ Monday, September 04, 2006 37 comments links to this post

37 Comments:

Anonymous said...

i don't know if this is because of the simplification, but the version you have there has problems with empty list as the last element.

flatten([[]])

November 16, 2006 at 10:44 AM

MonkeeSage said...

Good catch! I've fixed it now.

January 25, 2007 at 10:28 PM

Marcin said...

Here's a fixed point implementation:
http://murthercity.blogspot.com/2007/05/python-fixed-point-based-flatten.html
The auxiliary functions are complex.

May 15, 2007 at 6:23 AM

MonkeeSage said...

Interesting solution. :) It is kind of slow, but it gets points for creativity.

May 17, 2007 at 8:31 AM

Marcin said...

The point is that it's not so much creative, as a basic application of functional programming techniques.

In fact, my main problem with python for implementing efficient code is that it can't

optimise things like this properly, and it doesn't provide any simple way to write an efficient implementation oneself of many things.

I suppose I could try to write something equivalent in ruby, and see how it stacks up, but I'd rather persuade you to do that.

May 17, 2007 at 11:22 AM

Marcin said...

Of course, if I had my way, I'd have it recursive, but I know that python deals with recursion poorly.

May 17, 2007 at 11:31 AM

MonkeeSage said...

Yes, neither python nor ruby is tail-call optimized out of the box like scheme, so we are faced with stack-imposed recursion limits and other standard imperative programming limitations. Basically, we are stuck with stateful programming techniques in these languages. The upshot is that we are able to use side-effects to their full potential, which many functional paradigms cannot. So, while it might be more complex in these languages to implement a generic flatten function, at the same time it is easier to loop over a list of variables and act according to their arbitrary values. Six of one, and half-dozen of the other. But in any case, it's always good to question the status quo; you never know where an optimization might occur!

May 23, 2007 at 6:33 AM

Marcin said...

"The upshot is that we are able to use side-effects to their full potential"

What? How does tail-call optimisation prevent the use of side effects?

"which many functional paradigms cannot"

Name one. Just one.

May 24, 2007 at 12:33 PM

MonkeeSage said...

Well, the most obvious example is Haskell and monads, but Ocaml/SML have similar problem areas. But I wasn't saying that functional languages can't compete with imperative languages; I was only saying that each are best suited to a particular application. With imperative programming it is easier to do some things, but with functional programming it is easier to do others. Both have their own domains in which they proform the best.

June 11, 2007 at 7:52 AM

sfllaw said...

I've just started programming with Python and I was also surprised that there was no flatten() builtin.

So I whipped together this iterative algorithm that can take an infinite number of items and returns a flat list.

A similar generator version is slightly slower, due to the function call overhead. But it does save a lot of space! Implementing that is left as an exercise for the reader.

Here's the code. Note that the code is a tad verbose, in order to optimize for the non-recursive case.

```
def flatten(sequence, recursive=True):
    """flatten(sequence[, recursive]) -> list

    Returns a flat list containing all elements in the sequence in
    depth-first order.

    By default, it flattens all the sequence recursively.  Set
    recursive to false, in order to only flatten one level of
    elements.

    Examples:
    >>> flatten([1, 2, [3, 4], (5,6), [7, [8, [9, [10]]]]])
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> flatten([1, 2, [3, 4], (5,6), [7, [8, [9, [10]]]]], recursive=False)
[1, 2, 3, 4, 5, 6, 7, [8, [9, [10]]]]
"""
result = []                     # Result array
if recursive:
    # We use an iterative O(n) algorithm here.  By keeping a stack
    # of iter objects, we don't need to recurse down Python's call
    # stack.
    stack = []
    i = iter(sequence)
    while True:
        try:
            e = i.next()
            if hasattr(e, "__iter__") and not isinstance(e, basestring):
                stack.append(i)
                i = iter(e)
            else:
                result.append(e)
        except StopIteration:
            try:
                i = stack.pop()
            except IndexError:
                return result
else:
    for e in sequence:
        if hasattr(e, "__iter__") and not isinstance(e, basestring):
            result.extend(e)
        else:
            result.append(e)
    return result
```

August 19, 2007 at 1:59 AM

MonkeeSage said...

Nice! Very good algorithm. :)

September 18, 2007 at 12:20 AM

witsch said...

how about this one? :)

```
def flatten(l, ltypes=(list, tuple)):
    n = 0
    for i in iter(l):
        if isinstance(i, ltypes):
            l[n:n+1] = []
            n += 1
            l[n:n] = list(i)
    return l

a = []
for i in xrange(2000):
    a = [a, i]
a = flatten(a)
```

November 1, 2007 at 11:21 AM

Forrest said...

FYI, the first example has a problem. Where there is 'l' in the if statement there should be 'item'.

November 11, 2007 at 9:57 PM

John Y. said...

I know this is an old thread, but I'm commenting anyway because it appears near the top of Google searches on "flatten list python". (The ASPN Cookbook is higher, but I don't feel like registering.)

Your last Cookbook fix (posted 2007/05/14) still has a problem. It successfully flattens [[[]]] but fails on [1, 2, [3, []]], which was the other problem case presented on ASPN. The problem is that you've shortened the list (by popping) but haven't adjusted i to reflect this.

I believe the following works. (I have tested it successfully, but maybe I have not been imaginative enough in coming up with test cases.)

```
def flatten(L, containers=(list, tuple)):
    i = 0
    while i < len(L):
        while isinstance(L[i], containers):
            if not L[i]:
                L.pop(i)
                i -= 1
                break
            else:
                L[i:i + 1] = (L[i])
        i += 1
    return L
```

April 23, 2008 at 12:45 AM

John Y. said...

One more thing that might trip some people up: The function in my previous comment modifies the input list in place; that is, destructively. This also means you will get an error if you try to pass in a tuple. To make the function safer (but also a bit slower), make a copy:

```
def flatten(S, containers=(list, tuple)):
    L = list(S)
    i = 0
    etc.
```

April 23, 2008 at 1:27 AM

MonkeeSage said...

Thanks for your comments John Y! I've updated the post and the ASPN recipe based on your code. Nicely done. :)

December 13, 2008 at 3:01 AM

Juliano said...

For the same reasons John Y. above, I decided to post here...

I've been using Python for Natural Language Processing and, as I don't have a formal programming background, I just developed a VERY simple way of flattening my recursive lists...

```
def flatten(lol):
    lol = str(lol)
    lol = lol.replace('[', '')
    lol = lol.replace(']', '')
    lol = lol.replace(', ,', ',,')
    lol = '[' + lol + ']'
    lol = lol.replace('[,', '[')
    return eval(lol)
```

Besides, I tried to use this with some of the tests posted here and it worked as well...

Would be glad to get some feedback...

Thanks.

Juliano

January 25, 2009 at 11:26 PM

Andreas Zeidler said...

i'm not so sure about the "whoops all of the others for speed" — check the version i posted above again, it's 33% faster than mr. fletcher's.

see http://blog.zitc.de/2009/01/flattened.html for more details... :)

January 26, 2009 at 5:03 AM

MonkeeSage said...

@Juliano:

That's a creative use of python's ability to introspect lists, but it has the same problem that the naive implementation does: it runs into the recursion limit ("maximum recursion depth exceeded while getting the repr of a list"). It's also not very efficient in terms of memory use, since it creates a string copy of representation of the list, and a new list. I haven't timed it, but I'd also guess it is slower, since it involves the object creation I just spoke of, and because of the machinery to coerce the list to a string representation, and the function call to eval.

@Andreas:

I somehow missed your previous comment. Your version looks very interesting. I'll run some tests and update the post if your version is indeed faster.

April 14, 2009 at 5:55 PM

MonkeeSage said...

@Andreas:

I've posted a comment on your blog post explaining why I'm not going to recommend using your version yet, even though it is faster. Please leave a note if you come up with a fixed version that is still faster than Mr. Fletcher's version and I'll update this post.

April 14, 2009 at 6:17 PM

Bruce said...

I have a nested list whose contents are non-iterable objects. The function thus fails. How would I adjust it to work for my case?

July 8, 2009 at 10:57 AM

Bruce said...

Answer to my own question: add an __iter__ method to my class that just yields self.

July 8, 2009 at 1:47 PM

MonkeeSage said...

Yup, you need to implement the iterator protocol:

http://docs.python.org/library/stdtypes.html#container.__iter__

July 10, 2009 at 3:13 AM

Caolan said...

While this version is faster than mine for small lists, mine seems better able to handle very large lists (for me it starts becoming faster on lists nested around 25,000 deep, and improves significantly from there).

I imagine this version has a wider use-case ;)

If anyone is interested in my more highly scalable implementation:
http://caolanmcmahon.com/flatten_for_python

July 25, 2009 at 2:34 AM

MonkeeSage said...

@Caolan:

Nicely done!!! Very cool version of flatten!

August 8, 2009 at 10:45 PM

John said...

"what suprised me is that it can't grok even relatively shallow nesting, e.g., 10 levels deep"

I think the python recursion limit on your machine may be set to an abnormally low value. With the default installation on my machine, I can run the simple recursive version 500 levels deep.

September 21, 2009 at 5:28 PM

MonkeeSage said...

Hmmm. Try it on a data structure like the one produced by:

```
a = []
for i in xrange(10):
  a = [a, i]
```

October 16, 2009 at 1:55 AM

Thomas Figg said...

```
def flatten(input):
output = []
stack = []
stack.extend(reversed(input))
while stack:
top = stack.pop()
if isinstance(top, (list, tuple)):
stack.extend(reversed(top))
else:
output.append(top)

return output
```

I've tried this against 'the fastest' one listed in your main blog post, on a million elements:

Test: http://pastebin.ca/1649364

Mine:
$ time python flattentest.py flatten 1000000
1000000

real 0m5.209s
user 0m4.934s
sys 0m0.145s

The one that 'whoops all of the others for speed, nesting level support and elegance':
$ time python flattentest.py flatten2 1000000
1000000

real 10m32.296s
user 9m55.045s
sys 0m4.894s

October 30, 2009 at 1:28 PM

MonkeeSage said...

@Thomas:

Interesting solution! I can't reproduce your numbers, however. I'm seeing almost the same performance between the (final) version I posted and yours. Anyhow, great work rethinking the problem! Quite an interesting algorithm you've come up with!

November 3, 2009 at 12:29 AM

Anonymous said...

The reason it's not in the library is because of the retarded "isinstance() is evil" ppl. They are obsessed with not throwing exceptions.....they are simply wrong. so are the ppl that say gotos/labeled continues/labeled breaks are evil....please let these ppl find another language to stymie with propaganda

December 15, 2009 at 6:04 PM

Atlas Bergeron said...

This comment has been removed by the author.

April 3, 2012 at 11:08 AM

Atlas Bergeron said...

Well written. I have come up with a (better) solution that does not rely on recursion. There is a saying I have heard, "if you are using recursion in python, you are doing it wrong." Using objects and iteration is (generally) better. Here is my solution, you might get a kick out of it.

http://code.activestate.com/recipes/578092-flattening-an-arbitrarily-deep-list-or-any-iterato/

April 3, 2012 at 11:56 AM

Ben Mezger said...

def flatten(alist):
result = []
try:
for data in alist:
for nested in flatten(data):
result.append(nested)
except TypeError:
result.append(alist)
return result

June 5, 2012 at 4:08 PM

Python Larry said...

Thanks for this post! The link for the "very opaque version" no longer works; seems ActiveState has removed that list. Internet Archive to the rescue! :-)

July 24, 2013 at 10:52 AM

Ben Norman said...

This comment has been removed by the author.

November 8, 2013 at 7:11 PM

Ben Norman said...

This code should do the exact same thing but be slightly faster and more readable; I basically took out the, to me looking, redundant decrement then break bit and replaced it with a Pythonic(?) try: except: clause.

N.B. replaced spaces with "____", since blogger seems to be messing up the indentation.

```
func="""
def rewritten_flatten(l, ltypes=(list, tuple)): ##by me :)
____ltype = type(l)

____l = list(l)
____i = 0
____try: ##if it ends with a list then once it's discarded, it will check an index that doesn't exist, however this only happens at the end!
_________while i < len(l):
_____________while isinstance(l[i], ltypes):
_____________________if not l[i]: ##if l[i] is empty
_____________________________l.pop(i) ##discard l[i]
_____________________else:
_____________________________l[i:i + 1] = l[i] ##insert list into self - increasing len(l)
_____________i += 1
____except IndexError:
_________pass
____return ltype(l)
"""

exec(func.replace("____","    "))
```

November 8, 2013 at 7:17 PM

Anonymous said...

Thank you so much for this excellent post. Incidentally I found a few more very clever and interesting implementations on the link below.

https://wiki.python.org/moin/ProblemSets/99%20Prolog%20Problems%20Solutions#Problem_7:_Flatten_a_nested_list_structure

May 5, 2014 at 12:18 PM

Post a Comment

Links to this post:

Create a Link

<< Home