hrldcpr / **tree.md**

Last active 3 days ago • Report abuse

one-line tree in python

tree.md

One-line Tree in Python

Using Python's built-in defaultdict we can easily define a tree data structure:

```
def tree(): return defaultdict(tree)
```

That's it!

It simply says that a tree is a dict whose default values are trees.

(If you're following along at home, make sure to `from collections import defaultdict`)

(Also: Hacker News reader @zbuc points out that this is called autovivification. Cool!)

Examples

JSON-esque

Now we can create JSON-esque nested dictionaries without explicitly creating sub-dictionaries—they magically come into existence as we reference them:

```
users = tree()
users['harold']['username'] = 'hrldcpr'
users['handler']['username'] = 'matthandlersux'
```

We can print this as json with `print(json.dumps(users))` and we get the expected:

```
{"harold": {"username": "hrldcpr"}, "handler": {"username": "matthandlersux"}}
```

Without assignment

We can even create structure with no assignment at all, since merely referencing an entry creates it:

```
taxonomy = tree()
taxonomy['Animalia']['Chordata']['Mammalia']['Carnivora']['Felidae']['Felis']['cat']
taxonomy['Animalia']['Chordata']['Mammalia']['Carnivora']['Felidae']['Panthera']['lion']
taxonomy['Animalia']['Chordata']['Mammalia']['Carnivora']['Canidae']['Canis']['dog']
taxonomy['Animalia']['Chordata']['Mammalia']['Carnivora']['Canidae']['Canis']['coyote']
taxonomy['Plantae']['Solanales']['Solanaceae']['Solanum']['tomato']
taxonomy['Plantae']['Solanales']['Solanaceae']['Solanum']['potato']
taxonomy['Plantae']['Solanales']['Convolvulaceae']['Ipomoea']['sweet potato']
```

We'll prettyprint this time, which requires us to convert to standard dicts first:

```
def dicts(t): return {k: dicts(t[k]) for k in t}
```

Now we can prettyprint the structure with `pprint(dicts(taxonomy))`:

```
{'Animalia': {'Chordata': {'Mammalia': {'Carnivora': {'Canidae': {'Canis': {'coyote': {},
                                                                             'dog': {}}},
                                                      'Felidae': {'Felis': {'cat': {}},
                                                                  'Panthera': {'lion': {}}}}}}},
 'Plantae': {'Solanales': {'Convolvulaceae': {'Ipomoea': {'sweet potato': {}}},
                           'Solanaceae': {'Solanum': {'potato': {},
                                                      'tomato': {}}}}}}
```

So the substructures we referenced now exist as dicts, with empty dicts at the leaves.

Iteration

This tree can be fun to iteratively walk through, again because structure comes into being simply by referring to it.

For example, suppose we are parsing a list of new animals to add to our taxonomy above, so we want to call a function like:

```
add(taxonomy,
    'Animalia>Chordata>Mammalia>Cetacea>Balaenopteridae>Balaenoptera>blue whale'.split('>'))
```

We can implement this simply as:

```
def add(t, path):
  for node in path:
    t = t[node]
```

Again we are never assigning to the dictionary, but just by referencing the keys we have created our new structure:

```
{'Animalia': {'Chordata': {'Mammalia': {'Carnivora': {'Canidae': {'Canis': {'coyote': {},
                                                                             'dog': {}}},
                                                      'Felidae': {'Felis': {'cat': {}},
                                                                  'Panthera': {'lion': {}}}},
                                           'Cetacea': {'Balaenopteridae': {'Balaenoptera': {'blue whale': {
 'Plantae': {'Solanales': {'Convolvulaceae': {'Ipomoea': {'sweet potato': {}}},
                           'Solanaceae': {'Solanum': {'potato': {},
                                                      'tomato': {}}}}}}
```

Conclusion

This probably isn't very useful, and it makes for some perplexing code (see `add()` above).

But if you like Python then I hope this was fun to think about or worthwhile to understand.

There was a good discussion of this gist on Hacker News.

Ceasar commented on Apr 23, 2012

Nice write-up! I actually had discovered this with a friend (@**AlexeyMK**) last summer. It's a fun trick. The only thing I dislike about this code is that defaultdicts print in an ugly manner, so I prefer the solution offered in the SO post, but it's the same thing in the end.

hrldcpr commented on Apr 23, 2012 Owner

Nice! Defining it using `__missing__` is pretty awesome, I didn't know about that.
I figured other people must have done this before, but I'm glad to be spreading the word regardless ;)
(In fact it turns out the exact code is already in the autovivification wikipedia article...)

I agree the unprintability is annoying. But I really like how concise the one-line definition is, so for a writeup like this clarity wins.

md2perpe commented on Apr 24, 2012

Giving Python what has existed in PHP since the dinosaur era...

pferreir commented on Apr 24, 2012

I didn't know about `__missing__` . Great!

cblp commented on Apr 24, 2012

`tree = lambda: defaultdict (tree)` is shorter

JakubOboza commented on Apr 24, 2012

Isn't tree in hash both memory waste and making access of element slow. You lose both *features* of this two data structures.
In my opinion its not the best idea :)

jul commented on Apr 24, 2012

Funny coincidence ; I began a package that is oriented as using defaultdict as a k-ary unordered rooted trees (the specific kind of trees you build with defaultdict)
http://readthedocs.org/docs/vectordict/en/latest/

WoLpH commented on Apr 24, 2012

@**JakubOboza**: it might be a memory waste but it's not really making the access of the element slower.

If you want a recursive structure you should be prepared to handle the results of that. If you have a tree with `n` total items and `m` levels deep and you want to get an item that's `m` levels deep than you'll have a time complexity of `O(m)` which is pretty good if you ask me. And if you're going to get multiple items from depth `m-1` than you can just store the intermediate result to make all lookups `O(1)` again.

JakubOboza commented on Apr 24, 2012

@**WoLpH** not that i'm expert on algorithms but...
It depends on a tree type if it is AVL, BST or RB tree then access should be O(logn) whish is fast. with BST in worst case you get O(n) so it is not always "O(m)". Depends if the tree is balanced or not.

Lets think about how hash works you get in theory O(1) but it is O(k) where k is time spend on hashing function. this 1 in ( ) is marking CONTANT time, it is not changing while data set is growing.

Now space and memory. Hashtables have hash function that gives you index of the element. lets do a very basic example Char -> Int
so for A index of 65 or J you get 74. So you have potentially 24 keys A-Z.

Now important thing you have to allocate for each dict 24 spaces in memory even if you store there 1 key-value pair.

Now if you build tree in hash you get memory cluster fuck and access is much worse because you have tree layers and hashing function for key and this tree is not balanced so in worst scenario you are still at O(n)

Do you now understand what i said in previous comment ?

With regular tree in list / array or something you would be right. But if you will put it into hash you are screwed.

WoLpH commented on Apr 24, 2012

@**JakubOboza**: I'm far from an expert either. I think you and I are talking about different datastructures here.
This structure is more like a btree right now, it is a very simple recursive storage object. AVL/BS/RB trees are sorted structures, since this is based on a dictionary/hash it is not and cannot be sorted.

This data structure is much simpler than AVL/BS/RB trees, although it could be used as a building-block for that, I wouldn't recommend it.

The way I understood it is that this data structure gives you simple access to a recursive data structure without having to define it in advance and accessing it won't be "give me the largest element" or anything but more something like "give me element `x`" where `x` is a multi-dimensional key. In that case the lookup time is the amount of dimensions (see http://wiki.python.org/moin/TimeComplexity#dict for more info).

JakubOboza commented on Apr 24, 2012

I'm not saying tree is bad i'm saying this way of implementation is massively suboptimal in terms of memory and access time :)

jul commented on Apr 24, 2012

Well, a (more than) suboptimal solutions that can be implemented easily can still fit in the functionnal domain in which a software is supposed to work. Therefore it may often be no big deals. (And the computer being very cheap slaves that feel no pain, I strictly feel no remorse in making them burn their CPU on suboptimal code if it significiantly lowers the very expensive developper fees).
Thinking of complexity is nice. But overthinking about complexity when you have largely enough cheap resources is very close to procrastination.

JakubOboza commented on Apr 24, 2012

@**jul** no.
I think people should build things in right way :) Not say something is cheap, if you will build server software it will not be cheap anymore i see where we have 64 gb ram servers and we see this kind of things, refactor and save few gigs of ram instantly.

hrldcpr commented on Apr 24, 2012 Owner

@**JakubOboza** I agree that this tree implementation would be a bad choice for e.g. a filesystem, but at the same time, implementing a B-tree for storing your small JSON objects is also a bad idea. There is sometimes a tradeoff between code complexity and performance, and if you don't need high performance then the simplest solution is often the best.

WoLpH commented on Apr 24, 2012

@**JakubOboza**: agreed, but that's not the point here I think.

If you want to have a search tree, than this is the wrong structure. If you just want to store data which you always find by entering the key (say... settings, json data, etc...) than this is a good data structure.

hrldcpr commented on Apr 24, 2012 Owner

@**JakubOboza** and more generally, "the right way" does not equate to "the fastest way" — when building real software one of the skills is knowing when *not* to waste time optimizing.

JakubOboza commented on Apr 24, 2012

Why B-tree is bad for representing JSON objects ?

You have same interface then in case of hash
object["key"] = value, it is only under the hood implemented in a different way.
Performance wise on small hashes you will be even faster then on hashes because of the hashing function. So you have option to be faster and more space efficient :)

JakubOboza commented on Apr 24, 2012

@**hrldcpr**
You are right it is wrong, same as using 1 line hacks :)

WoLpH commented on Apr 24, 2012

JakubOboza: a B-tree is 1-dimensional from the users perspective, this object is multi-dimensional from the users perspective. They are 2 completely different things.

JakubOboza commented on Apr 24, 2012

@**WoLpH** now you talk about syntax :) i'm not hater or something i just pointed out that i like this in form of hack but i don't like when people use this type of hacks in real software :)

If Python would be like Ocaml or Scheme you could implement multi dimensional syntax also :) So i don't wanna talk about syntax :>

Cheers!

WoLpH commented on Apr 24, 2012

@**JakubOboza** it is not about syntax, it is about a completely different data structure and matching usage.

But please do amuse me a little here, what kind of data structure would you use to store recursive objects which don't require sorting? I am genuinely interested, I think this structure is fairly optimal for that usage pattern.

aJanuary commented on Apr 24, 2012

@**JakubOboza** You're talking about searching the tree, while @**WoLpH** was talking about walking it.

Walking from root to a particular node without use of an auxiliary data structure will take O(m) (where m is the distance from root to node) no matter how the tree is sorted.

If you want to find a node that meets a particular condition you can sort the tree in such a way as to not need to walk the entire tree (worst case O(n) where n is the number of nodes).

ghost commented on Apr 24, 2012

I recently implemented a similar dict inside of dict data structure for a simple search program. @**JakubOboza** is correct, it was a memory disaster.

crazy4groovy commented on Apr 24, 2012

Nice inspiration for a similar Groovy implementation at: https://gist.github.com/2478499

jul commented on Apr 24, 2012

@JakubOboza : abstract data structures are mainly APIs with properties (index, keys, values, parent, next .... whatever). And API is a view on an implementation.
If you badly need a property and it worths it, then API matters.

Complexity is mostly an implementation problem,which performance will vary according to your real case of use. With python's duck typing and ctypes, performance will mostly not lie in the API but in the implementation.

For instance matrix are either 1D array accessed by [index % line_length + int(index/line_length * line_length)] or a 2D array, or a dict (sparse matrix). Performance is (mostly) not a matter of data API but a matter of use case and implementation. o(n) are only best case implementation. Life is not always best case, even with well known data structures.

So, I still do think that reasoning on best case complexity instead of API is a premature optimisation.

API matters, since what you mean matters. If you need, it worths it. Complexity is just an implementation detail, which is a concern for 0.01% of *real* developpers.

markwatson commented on Apr 25, 2012

Here's some code I'm using for pretty printing without json in older versions of python (that don't have dictionary comprehensions):

```
def dicts(t):
    try:
        return dict((k, dicts(t[k])) for k in t)
    except TypeError:
        return t
```

westurner commented on Jan 3, 2013

You can add a field delimiter and 'flatten' the keys if you would like to avoid the additional lookup overhead.

e.g. instead of data['one']['two']['three'], it's just data['one.two.three'], or data[('one','two','three')]

@JakubOboza : a trie would be faster. There are many implementations of tries for Python.

westurner commented on Jan 3, 2013

There exist implementations of DefaultOrderedDict : http://stackoverflow.com/a/6190500

westurner commented on Jan 3, 2013

This works well for serializing to Unicode:

```
json.dumps(obj, indent=2)
```

A custom JSONEncoder with support for ISO datetimes can be helpful: http://stackoverflow.com/a/2680060

domenkozar commented on Jan 3, 2013

Exactly the trick that I use in mr.bob (http://mrbob.readthedocs.org/en/latest/) to make namespace of keys :)

bj0 commented on Feb 18, 2013

Adding a simple subclass:

```
class mydefaultdict(defaultdict):
    def __getattr__(self, attr):
        return self[attr]
    def __setattr__(self, attr, val):
        self[attr] = val

def tree(): return mydefaultdict(tree)
```

will turn:

```
taxonomy['Animalia']['Chordata']['Mammalia']['Carnivora']['Felidae']['Felis']['cat']
```

into:

```
taxonomy.Animalia.Chordata.Mammalia.Carnivora.Felidae.Felis.cat
```

dgagnon commented on Nov 1, 2013

You can use the same trick to edit the files as well:

```
def add(t, tkeys, value):
  i = 0
  keys = tkeys.split('.')
  for key in keys:
    i = i + 1
    if i == len(keys) and value != None:
      t[key] = value
    t = t[key]
```

ashumeow commented on Nov 5, 2013

Good! =)

metaperl commented on Mar 11, 2014

Very cool. I tend to resort to objects and methods for everything. And you did this in 1-2 lines.

gVallverdu commented on Mar 26, 2014

Nice solution !
I wrote that lines to print the tree :

```
def ptr(t, depth = 0):
    """ print a tree """
    for k in t.keys():
        print("%s %2d %s" % ("".join(depth * ["    "]), depth, k))
        depth += 1
        ptratree(t[k], depth)
        depth -= 1
```

using your example it gives :

```
0 Animalia
    1 Chordata
        2 Mammalia
            3 Carnivora
                4 Canidae
                    5 Canis
                        6 coyote
                        6 dog
                4 Felidae
                    5 Felis
                        6 cat
                    5 Panthera
                        6 lion
0 Plantae
    1 Solanales
        2 Convolvulaceae
            3 Ipomoea
                4 sweet potato
        2 Solanaceae
            3 Solanum
                4 tomato
                4 potato
```

drasko commented on May 18, 2014

In case someone needs it off-the-shelf, code to create JSON representation of the file tree using this technique:

```
import os
import collections
import json

def tree(): return collections.defaultdict(tree)

def add(t, keys):
  for key in keys:
    t = t[key]

s = tree()

# Set the directory you want to start from
path = 'path/to/directory'

for root, dirs, files in os.walk(path):
    for f in files:
        l = root.split('/')
        l.append(f)

        add(s, l)

print(json.dumps(s, indent=4, sort_keys=True))
```

kzinglzy commented on Jun 6, 2014

Wow! It is amazing. Thanks

baocaixiong commented on Oct 11, 2014

amazing. Thanks!

adugin commented on Dec 26, 2014

```
def tree(levels=0, func=None):
    if levels > 0 and func:
```

```
        return defaultdict(reduce(lambda f, i: lambda: defaultdict(f), xrange(levels-1), func))
    else:
        return defaultdict(tree)
```

Usage:

```
>>> d = tree()
>>> d[1][2][3][4][5] = 6
>>> d = tree(3, float)
>>> d
defaultdict(<function <lambda> at 0x0280E3F0>, {})
>>> d[1][2][3]
0.0
>>> d[7][8][9] += 10
>>> d[7][8][9]
10.0
>>> d = tree(0, int)
>>> d
defaultdict(<function tree at 0x0280C830>, {})
>>> d = tree(1, int)
>>> d
defaultdict(<type 'int'>, {})
>>> d[1]
0
>>>
```

adugin commented on Dec 30, 2014

```
def tree(func=lambda: tree(), depth=0):
    return defaultdict(reduce(lambda f, i: lambda: defaultdict(f), xrange(depth-1), func))
```

uralbash commented on Feb 12, 2016

Same with ordered tree:

```
from collections import OrderedDict, defaultdict

def tree():
    """
    Autovivication ordered hash
    """
    return OrderedDict(defaultdict(tree))
```

nehemiahjacob commented on May 17, 2016

I guess this attracts more appreciation than it deserves. In the reality it will break for a very basic case.

```
>>> from collections import defaultdict
>>> def tree(): return defaultdict(tree)
...
>>> t = tree()
>>> t['a']['b']['c'] = 'hello'
>>> t['a']['b']['c']['d'] = 'world'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

I believe I don't have to explain why it fails.

ionFreeman commented on Aug 9, 2016 • edited

@nehemiahjacob Yes, you can break it by assigning to it. Don't assign to it.

```
t = tree()
t['a']['b']['c']['hello']
t['a']['b']['c']['d']['world']
```

sshadmand commented on Dec 3, 2016 • edited

This is great! Question though: How would I programmatically create a dynamic set list of keys into a tree with depth?

For example, I have a variable passed in as a list of terms: ['a', 'b', 'c'], and, based on that list I want to create a tree where each item is the child of the previous item.

Based on your example, the output would be equivalent to the output generated by this assignment:

```
t = tree()
t["a"]["b"]["c"]
```

But again, I want to create that tree dynamically based on the list of keys I am sent.

Any ideas? :)

UPDATE

I found the answer on S.O. here:

http://stackoverflow.com/questions/14692690/access-nested-dictionary-items-via-a-list-of-keys

c0fec0de commented on Mar 14, 2017

There is a powerful python tree library http://anytree.readthedocs.io/

martinym commented on Dec 27, 2017 • edited

There's a **very** good answer discussion on stackoverflow about the different ways of doing this and in particular one very simple one that doesn't use `defaultdict`. It's very portable (Python 2.5+ & 3.x) and another potentially very nice thing about it is the resulting subclass instances `print` just like a normal `dict`s.

If you're too lazy to look (**`TL;DR`**), here's the code (all of it):

```
class Vividict(dict):
    def __missing__(self, key):
        value = self[key] = type(self)() # retain local pointer to value
        return value                     # faster to return than dict lookup
```

jedie commented on May 3

made another variant of a tree: https://gist.github.com/jedie/523dc31dbbd22968b58a0f4798876e3c