Math ∪ Code

by Sahand Saba

BLOG COACHING GITHUB ABOUT

30 Python Language Features and Tricks You May Not Know About

MAR 05, 2014

1 Introduction

Since I started learning Python, I decided to maintain an often visited list of "tricks". Any time I saw a piece of code (in an example, on Stack Overflow, in open source software, etc.) that made me think "Cool! I didn't know you could do that!" I experimented with it until I understood it and then added it to the list. This post is part of that list, after some cleaning up. If you are an experienced Python programmer, chances are you already know most of these, though you might still find a few that you didn't know about. If you are a C, C++ or Java programmer who is learning Python, or just brand new to programming, then you might find quite a few of them surprisingly useful, like I did.

Each trick or language feature is demonstrated only through examples, with no explanation. While I tried my best to make the examples clear, some of them might still appear cryptic depending on your familiarity level. So if something still doesn't make sense after looking at the examples, the title should be clear enough to allow you to use Google for more information on it.

The list is very roughly ordered by difficulty, with the easier and more commonly known language features and tricks appearing first.

A table of contents is given at the end.

April 9th, 2014 update: As you can see the article has been growing with currently 38 items in it, mostly thanks to comments from readers. As such the number 30 in the title is no longer accurate. However, I chose to leave it as is since that's the original title the article was shared as, making it more recognizable and easier to find.

March 14th, 2014 update: Roy Keyes made a great suggestion of turning this article into a GitHub repository to allow readers to make improvements or additions through pull requests. The repository is now at https://github.com/sahands/python-by-example. Feel free to fork, add improvements or additions and submit pull requests. I will update this page periodically with the new additions.

March 8th, 2014 update: This article generated a lot of good discussion on Reddit, Hacker News, and in the comments below, with many readers suggesting great alternatives and improvements. I have updated the list below to include many of the improvements suggested, and added a few new items based on suggestions that made me have one of those "Cool! I didn't know you could do that!" moments. In particular, I did not know about `itertools.chain.from_iterable`, and dictionary comprehensions. There was also a very interesting discussion about the possibility of some of the techniques below leading to harder to debug code. My say on it is that as far as I can see, none of the items below are inherently harder to debug. But I can definitely see how they can be taken too far, resulting in hard to debug, maintain and understand code. Use your best judgment and if it feels like how short and smart your code is is outweighing how readable and maintainable it is, then break it down and simplify it. For example, I think list comprehensions can be very readable and rather easy to debug and maintain. But a list comprehension inside another list comprehension that is then passed to `map` and then to `itertools.chain`? Probably not the best idea!

1.1 Unpacking

```
>>> a, b, c = 1, 2, 3
>>> a, b, c
(1, 2, 3)
>>> a, b, c = [1, 2, 3]
>>> a, b, c
(1, 2, 3)
>>> a, b, c = (2 * i + 1 for i in range(3))
>>> a, b, c
(1, 3, 5)
>>> a, (b, c), d = [1, (2, 3), 4]
>>> a
1
>>> b
2
>>> c
3
>>> d
4
```

1.2 Unpacking for swapping variables

```
>>> a, b = 1, 2
>>> a, b = b, a
>>> a, b
(2, 1)
```

1.3 Extended unpacking (Python 3 only)

```
>>> a, *b, c = [1, 2, 3, 4, 5]
>>> a
1
>>> b
[2, 3, 4]
>>> c
5
```

1.4 Negative indexing

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[-1]
10
>>> a[-3]
8
```

1.5 List slices (`a[start:end]`)

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[2:8]
[2, 3, 4, 5, 6, 7]
```

1.6 List slices with negative indexing

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[-4:-2]
[7, 8]
```

1.7 List slices with step (`a[start:end:step]`)

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[::2]
[0, 2, 4, 6, 8, 10]
>>> a[::3]
[0, 3, 6, 9]
>>> a[2:8:2]
[2, 4, 6]
```

1.8 List slices with negative step

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[::-1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> a[::-2]
[10, 8, 6, 4, 2, 0]
```

1.9 List slice assignment

```
>>> a = [1, 2, 3, 4, 5]
>>> a[2:3] = [0, 0]
>>> a
[1, 2, 0, 0, 4, 5]
>>> a[1:1] = [8, 9]
>>> a
[1, 8, 9, 2, 0, 0, 4, 5]
>>> a[1:-1] = []
>>> a
[1, 5]
```

1.10 Naming slices (`slice(start, end, step)`)

```
>>> a = [0, 1, 2, 3, 4, 5]
>>> LASTTHREE = slice(-3, None)
>>> LASTTHREE
slice(-3, None, None)
>>> a[LASTTHREE]
[3, 4, 5]
```

1.11 Iterating over list index and value pairs (`enumerate`)

```
>>> a = ['Hello', 'world', '!']
>>> for i, x in enumerate(a):
...     print '{}: {}'.format(i, x)
...
0: Hello
1: world
2: !
```

1.12 Iterating over dictionary key and value pairs (`dict.iteritems`)

```
>>> m = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> for k, v in m.iteritems():
...     print '{}: {}'.format(k, v)
...
a: 1
c: 3
b: 2
d: 4
```

Note: use `dict.items` in Python 3.

1.13 Zipping and unzipping lists and iterables

```
>>> a = [1, 2, 3]
>>> b = ['a', 'b', 'c']
>>> z = zip(a, b)
>>> z
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> zip(*z)
[(1, 2, 3), ('a', 'b', 'c')]
```

1.14 Grouping adjacent list items using zip

```
>>> a = [1, 2, 3, 4, 5, 6]

>>> # Using iterators
>>> group_adjacent = lambda a, k: zip(*([iter(a)] * k))
>>> group_adjacent(a, 3)
[(1, 2, 3), (4, 5, 6)]
>>> group_adjacent(a, 2)
[(1, 2), (3, 4), (5, 6)]
>>> group_adjacent(a, 1)
[(1,), (2,), (3,), (4,), (5,), (6,)]


>>> # Using slices
>>> from itertools import islice
>>> group_adjacent = lambda a, k: zip(*(islice(a, i, None, k) for i in range(k)))
>>> group_adjacent(a, 3)
[(1, 2, 3), (4, 5, 6)]
>>> group_adjacent(a, 2)
[(1, 2), (3, 4), (5, 6)]
>>> group_adjacent(a, 1)
[(1,), (2,), (3,), (4,), (5,), (6,)]
```

1.15 Sliding windows (n -grams) using zip and iterators

```
>>> from itertools import islice
>>> def n_grams(a, n):
...     z = (islice(a, i, None) for i in range(n))
...     return zip(*z)
...
>>> a = [1, 2, 3, 4, 5, 6]
>>> n_grams(a, 3)
[(1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 6)]
>>> n_grams(a, 2)
[(1, 2), (2, 3), (3, 4), (4, 5), (5, 6)]
>>> n_grams(a, 4)
[(1, 2, 3, 4), (2, 3, 4, 5), (3, 4, 5, 6)]
```

1.16 Inverting a dictionary using zip

```
>>> m = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> m.items()
[('a', 1), ('c', 3), ('b', 2), ('d', 4)]
>>> zip(m.values(), m.keys())
[(1, 'a'), (3, 'c'), (2, 'b'), (4, 'd')]
>>> mi = dict(zip(m.values(), m.keys()))
>>> mi
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

1.17 Flattening lists:

```
>>> a = [[1, 2], [3, 4], [5, 6]]
>>> list(itertools.chain.from_iterable(a))
[1, 2, 3, 4, 5, 6]

>>> sum(a, [])
[1, 2, 3, 4, 5, 6]

>>> [x for l in a for x in l]
[1, 2, 3, 4, 5, 6]

>>> a = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
>>> [x for l1 in a for l2 in l1 for x in l2]
[1, 2, 3, 4, 5, 6, 7, 8]

>>> a = [1, 2, [3, 4], [[5, 6], [7, 8]]]
>>> flatten = lambda x: [y for l in x for y in flatten(l)] if type(x) is list else [x]
>>> flatten(a)
[1, 2, 3, 4, 5, 6, 7, 8]
```

Note: according to Python's documentation on `sum`, `itertools.chain.from_iterable` is the preferred method for this.

1.18 Generator expressions

```
>>> g = (x ** 2 for x in xrange(10))
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
>>> next(g)
9
>>> sum(x ** 3 for x in xrange(10))
2025
>>> sum(x ** 3 for x in xrange(10) if x % 3 == 1)
408
```

1.19 Dictionary comprehensions

```
>>> m = {x: x ** 2 for x in range(5)}
>>> m
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

>>> m = {x: 'A' + str(x) for x in range(10)}
>>> m
{0: 'A0', 1: 'A1', 2: 'A2', 3: 'A3', 4: 'A4', 5: 'A5', 6: 'A6', 7: 'A7', 8: 'A8', 9: 'A9'}
```

1.20 Inverting a dictionary using a dictionary comprehension

```
>>> m = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> m
{'d': 4, 'a': 1, 'b': 2, 'c': 3}
>>> {v: k for k, v in m.items()}
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

1.21 Named tuples (`collections.namedtuple`)

```
>>> Point = collections.namedtuple('Point', ['x', 'y'])
>>> p = Point(x=1.0, y=2.0)
>>> p
Point(x=1.0, y=2.0)
>>> p.x
1.0
>>> p.y
2.0
```

1.22 Inheriting from named tuples:

```
>>> class Point(collections.namedtuple('PointBase', ['x', 'y'])):
...     __slots__ = ()
...     def __add__(self, other):
...             return Point(x=self.x + other.x, y=self.y + other.y)
...
>>> p = Point(x=1.0, y=2.0)
>>> q = Point(x=2.0, y=3.0)
>>> p + q
Point(x=3.0, y=5.0)
```

1.23 Sets and set operations

```
>>> A = {1, 2, 3, 3}
>>> A
set([1, 2, 3])
>>> B = {3, 4, 5, 6, 7}
>>> B
set([3, 4, 5, 6, 7])
>>> A | B
set([1, 2, 3, 4, 5, 6, 7])
>>> A & B
set([3])
>>> A - B
set([1, 2])
>>> B - A
set([4, 5, 6, 7])
>>> A ^ B
set([1, 2, 4, 5, 6, 7])
>>> (A ^ B) == ((A - B) | (B - A))
True
```

1.24 Multisets and multiset operations (`collections.Counter`)

```
>>> A = collections.Counter([1, 2, 2])
>>> B = collections.Counter([2, 2, 3])
>>> A
Counter({2: 2, 1: 1})
>>> B
Counter({2: 2, 3: 1})
>>> A | B
Counter({2: 2, 1: 1, 3: 1})
>>> A & B
Counter({2: 2})
>>> A + B
Counter({2: 4, 1: 1, 3: 1})
>>> A - B
Counter({1: 1})
>>> B - A
Counter({3: 1})
```

1.25 Most common elements in an iterable (`collections.Counter`)

```
>>> A = collections.Counter([1, 1, 2, 2, 3, 3, 3, 3, 4, 5, 6, 7])
>>> A
Counter({3: 4, 1: 2, 2: 2, 4: 1, 5: 1, 6: 1, 7: 1})
>>> A.most_common(1)
[(3, 4)]
>>> A.most_common(3)
[(3, 4), (1, 2), (2, 2)]
```

1.26 Double-ended queue (`collections.deque`)

```
>>> Q = collections.deque()
>>> Q.append(1)
>>> Q.appendleft(2)
>>> Q.extend([3, 4])
>>> Q.extendleft([5, 6])
>>> Q
deque([6, 5, 2, 1, 3, 4])
>>> Q.pop()
4
>>> Q.popleft()
6
>>> Q
deque([5, 2, 1, 3])
>>> Q.rotate(3)
>>> Q
deque([2, 1, 3, 5])
>>> Q.rotate(-3)
>>> Q
deque([5, 2, 1, 3])
```

1.27 Double-ended queue with maximum length (`collections.deque`)

```
>>> last_three = collections.deque(maxlen=3)
>>> for i in xrange(10):
...     last_three.append(i)
...     print ', '.join(str(x) for x in last_three)
...
0
0, 1
0, 1, 2
1, 2, 3
2, 3, 4
3, 4, 5
4, 5, 6
5, 6, 7
6, 7, 8
7, 8, 9
```

1.28 Ordered dictionaries (`collections.OrderedDict`)

```
>>> m = dict((str(x), x) for x in range(10))
>>> print ', '.join(m.keys())
1, 0, 3, 2, 5, 4, 7, 6, 9, 8
>>> m = collections.OrderedDict((str(x), x) for x in range(10))
>>> print ', '.join(m.keys())
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
>>> m = collections.OrderedDict((str(x), x) for x in range(10, 0, -1))
>>> print ', '.join(m.keys())
10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

1.29 Default dictionaries (`collections.defaultdict`)

```
>>> m = dict()
>>> m['a']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'a'
>>>
>>> m = collections.defaultdict(int)
>>> m['a']
0
>>> m['b']
0
>>> m = collections.defaultdict(str)
>>> m['a']
''
>>> m['b'] += 'a'
>>> m['b']
'a'
>>> m = collections.defaultdict(lambda: '[default value]')
>>> m['a']
'[default value]'
>>> m['b']
'[default value]'
```

1.30 Using default dictionaries to represent simple trees

```
>>> import json
>>> tree = lambda: collections.defaultdict(tree)
>>> root = tree()
>>> root['menu']['id'] = 'file'
>>> root['menu']['value'] = 'File'
>>> root['menu']['menuitems']['new']['value'] = 'New'
>>> root['menu']['menuitems']['new']['onclick'] = 'new();'
>>> root['menu']['menuitems']['open']['value'] = 'Open'
>>> root['menu']['menuitems']['open']['onclick'] = 'open();'
>>> root['menu']['menuitems']['close']['value'] = 'Close'
>>> root['menu']['menuitems']['close']['onclick'] = 'close();'
>>> print json.dumps(root, sort_keys=True, indent=4, separators=(',', ': '))
{
    "menu": {
        "id": "file",
        "menuitems": {
            "close": {
                "onclick": "close();",
                "value": "Close"
            },
            "new": {
                "onclick": "new();",
                "value": "New"
            },
            "open": {
                "onclick": "open();",
                "value": "Open"
            }
        },
        "value": "File"
    }
}
```

(See https://gist.github.com/hrldcpr/2012250 for more on this.)

1.31 Mapping objects to unique counting numbers (`collections.defaultdict`)

```
>>> import itertools, collections
>>> value_to_numeric_map = collections.defaultdict(itertools.count().next)
>>> value_to_numeric_map['a']
0
>>> value_to_numeric_map['b']
1
>>> value_to_numeric_map['c']
2
>>> value_to_numeric_map['a']
0
>>> value_to_numeric_map['b']
1
```

1.32 Largest and smallest elements (`heapq.nlargest` and `heapq.nsmallest`)

```
>>> a = [random.randint(0, 100) for __ in xrange(100)]
>>> heapq.nsmallest(5, a)
[3, 3, 5, 6, 8]
>>> heapq.nlargest(5, a)
[100, 100, 99, 98, 98]
```

1.33 Cartesian products (`itertools.product`)

```
>>> for p in itertools.product([1, 2, 3], [4, 5]):
(1, 4)
(1, 5)
(2, 4)
(2, 5)
(3, 4)
(3, 5)
>>> for p in itertools.product([0, 1], repeat=4):
...     print ''.join(str(x) for x in p)
...
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

1.34 Combinations and combinations with replacement (`itertools.combinations` and `itertools.combinations_with_replacement`)

```
>>> for c in itertools.combinations([1, 2, 3, 4, 5], 3):
...     print ''.join(str(x) for x in c)
...
123
124
125
134
135
145
234
235
245
345
>>> for c in itertools.combinations_with_replacement([1, 2, 3], 2):
...     print ''.join(str(x) for x in c)
...
11
12
13
22
23
33
```

1.35 Permutations (`itertools.permutations`)

```
>>> for p in itertools.permutations([1, 2, 3, 4]):
...     print ''.join(str(x) for x in p)
...
1234
1243
1324
1342
1423
1432
2134
2143
2314
2341
2413
2431
3124
3142
3214
3241
3412
3421
4123
4132
4213
4231
4312
4321
```

1.36 Chaining iterables (`itertools.chain`)

```
>>> a = [1, 2, 3, 4]
>>> for p in itertools.chain(itertools.combinations(a, 2), itertools.combinations(a, 3)):
...     print p
...
(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)
(1, 2, 3)
(1, 2, 4)
(1, 3, 4)
(2, 3, 4)
>>> for subset in itertools.chain.from_iterable(itertools.combinations(a, n) for n in range(len(a) + 1))
...     print subset
...
()
(1,)
(2,)
(3,)
(4,)
(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)
(1, 2, 3)
(1, 2, 4)
(1, 3, 4)
(2, 3, 4)
(1, 2, 3, 4)
```

1.37 Grouping rows by a given key (`itertools.groupby`)

```
>>> from operator import itemgetter
>>> import itertools
>>> with open('contactlenses.csv', 'r') as infile:
...     data = [line.strip().split(',') for line in infile]
...
>>> data = data[1:]
>>> def print_data(rows):
...     print '\n'.join('\t'.join('{: <16}'.format(s) for s in row) for row in rows)
...

>>> print_data(data)
young               myope                   no                      reduced                 none
young               myope                   no                      normal                  soft
young               myope                   yes                     reduced                 none
young               myope                   yes                     normal                  hard
young               hypermetrope            no                      reduced                 none
young               hypermetrope            no                      normal                  soft
young               hypermetrope            yes                     reduced                 none
young               hypermetrope            yes                     normal                  hard
pre-presbyopic      myope                   no                      reduced                 none
```

```
pre-presbyopic      myope                   no                      normal                  soft
pre-presbyopic      myope                   yes                     reduced                 none
pre-presbyopic      myope                   yes                     normal                  hard
pre-presbyopic      hypermetrope            no                      reduced                 none
pre-presbyopic      hypermetrope            no                      normal                  soft
pre-presbyopic      hypermetrope            yes                     reduced                 none
pre-presbyopic      hypermetrope            yes                     normal                  none
presbyopic          myope                   no                      reduced                 none
presbyopic          myope                   no                      normal                  none
presbyopic          myope                   yes                     reduced                 none
presbyopic          myope                   yes                     normal                  hard
presbyopic          hypermetrope            no                      reduced                 none
presbyopic          hypermetrope            no                      normal                  soft
presbyopic          hypermetrope            yes                     reduced                 none
presbyopic          hypermetrope            yes                     normal                  none

>>> data.sort(key=itemgetter(-1))
>>> for value, group in itertools.groupby(data, lambda r: r[-1]):
...     print '-----------'
...     print 'Group: ' + value
...     print_data(group)
...
-----------
Group: hard
young               myope                   yes                     normal                  hard
young               hypermetrope            yes                     normal                  hard
pre-presbyopic      myope                   yes                     normal                  hard
presbyopic          myope                   yes                     normal                  hard
-----------
Group: none
young               myope                   no                      reduced                 none
young               myope                   yes                     reduced                 none
young               hypermetrope            no                      reduced                 none
young               hypermetrope            yes                     reduced                 none
pre-presbyopic      myope                   no                      reduced                 none
pre-presbyopic      myope                   yes                     reduced                 none
pre-presbyopic      hypermetrope            no                      reduced                 none
pre-presbyopic      hypermetrope            yes                     reduced                 none
pre-presbyopic      hypermetrope            yes                     normal                  none
presbyopic          myope                   no                      reduced                 none
presbyopic          myope                   no                      normal                  none
presbyopic          myope                   yes                     reduced                 none
presbyopic          hypermetrope            no                      reduced                 none
presbyopic          hypermetrope            yes                     reduced                 none
presbyopic          hypermetrope            yes                     normal                  none
-----------
Group: soft
young               myope                   no                      normal                  soft
young               hypermetrope            no                      normal                  soft
pre-presbyopic      myope                   no                      normal                  soft
pre-presbyopic      hypermetrope            no                      normal                  soft
presbyopic          hypermetrope            no                      normal                  soft
```

1.38 Start a static HTTP server in any directory

```
[10:26] $ python -m SimpleHTTPServer 5000
Serving HTTP on 0.0.0.0 port 5000 ...
```

1.39 Learn the Zen of Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.40 Use C-Style Braces Instead of Indentation to Denote Scopes

```
>>> from __future__ import braces
```

2 Table of contents

List of language features and tricks in this article:

Comments

63 Comments **Math U Coding** 1 **Login**

Recommend 36 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Brendan Rodrigues • 7 months ago

Great post. I'm working on a Python tricks video tutorial. Would you be interested in creating it?

I would love to share the idea with you. You can email me at brendanr@packtpub.com

25 ^ | v • Reply • Share ›

Jeremy Smith • 5 years ago

Thanks for this. I also liked 1.24, which I hadn't seen before. By the way, here is a good trick that I have used on a couple of occasions: https://gist.github.com/hrl...

25 ^ | v • Reply • Share ›

Chuntao Lu ➦ Jeremy Smith • 5 years ago

That's one slick trick

1 ^ | v • Reply • Share ›

chanlvh • 5 years ago

Shouldn't the ToC be at the beginning of the post?

20 ^ | v • Reply • Share ›

Robb Jones ➦ chanlvh • 3 years ago

Nah, it's right here by the comments where we need it :-D

^ | v • Reply • Share ›

Wei-Ting Kuo • 5 years ago

For 1.12, we can use
zip(*[iter(a)]*2)

7 ^ | v • Reply • Share ›

Guest ➦ Wei-Ting Kuo • 5 years ago

or alternatively using dictionary comprehension: {a[k]:k for k in a}

1 ^ | v • Reply • Share ›

thefourtheye ➦ Guest • 5 years ago

This is how it should have been.

^ | v • Reply • Share ›

Abram Clark • 5 years ago

I also liked 1.24. Totally ingenious :). Also, in 1.30, I think the file reading is just a tidbit cleaner using with, eliminating the need to close:

```
with open('contactlenses.csv') as infile:
    data = [line.strip().split(',') for line in infile.readlines()]
```

4 ^ | v • Reply • Share ›

Paul Scott • 5 years ago

Your unpacking example doesn't demonstrate that you can equally unpack nested tuples:

```
>>> for i, (key, (left, right)) in enumerate({'a': (0, 1), 'b': (1, 2), 'c': (2, 3)}.items()):
...     print i, key, left, right
...
0 a 0 1
1 c 2 3
2 b 1 2
```

3 ^ | v • Reply • Share ›

hughdbrown • 5 years ago

And you can use itertools.chain to flatten lists (an alternative to the solution you listed):

```
>>> from itertools import chain
>>> a = [list(range(4)), list(range(1, 5)), list(range(10,20))]
>>> a = [list(range(4)), list(range(1, 5)), list(range(10,20))]
>>> a
[[0, 1, 2, 3], [1, 2, 3, 4], [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]]
>>> list(chain(*a))
[0, 1, 2, 3, 1, 2, 3, 4, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

3 ^ | v • Reply • Share ›

HamstaGangsta ➦ hughdbrown • 5 years ago

list() in line 2 is unnecessary. range() returns list by default. This works as well:

```
>>> a = [range(4), range(1, 5), range(10,20)]
```

^ | v • Reply • Share ›

Hugh Brown ➦ HamstaGangsta • 5 years ago

You are right but not for the reason you cite. In python 3, range() does not return a list. It is a range object.

```
>>> from itertools import chain
>>> a = [range(4), range(1, 5), range(10, 20)]
>>> a
[range(0, 4), range(1, 5), range(10, 20)]
```

However, in the scope of the itertools.chain() call, it is handled correctly.

2 ^ | v • Reply • Share ›

HamstaGangsta ➦ Hugh Brown • 5 years ago

Thanks for pointing it out :) (I'm not familiar with Python 3)

^ | v • Reply • Share ›

Jeff Jenkins • 5 years ago

You might want to change the OrderedDict example so that it shows that the keys are in order but not sorted. That's a common misconception people have

3 ^ | v • Reply • Share ›

Sebastian Silva • 4 years ago

I think this one didn't work... :o)

```
>>> from __future__ import braces
File "<stdin>", line 1
SyntaxError: not a chance
```

2 ^ | v • Reply • Share ›

Theron • 5 years ago

For #25, you don't even need to use heapq, you can achieve the same with slices.

sorted(a)[:5]

and:

sorted(a)[-5:]

2 ^ | v • Reply • Share ›

hughdbrown ➦ Theron • 5 years ago

You can construct a heap on O(log n) time. Sorting is O(n log n). If you want all of the elements in sorted order, sort them. If you need a few, use a heap. It's a trade-off, but in the simple case, the heap uses less resources.

resources.

Reply • Share ›

EdmondsCarp ➦ hughdbrown • 5 years ago

Constructing a heap takes O(n) time. (It has to - how can you even insert n elements into a list in anything better than O(n)?)

Reply • Share ›

hughdbrown ➦ EdmondsCarp • 5 years ago

Ack. Sorry. It takes O(n) time. Retrieving all elements would be O(n log n), but in the reduced case, you retrieve fewer so you can do it faster. And the trade-off is that as you approach retrieving every element from the heap, you approach the time of sorting so you may as well do it that way in the first place.

Reply • Share ›

Robert King ➦ hughdbrown • 5 years ago

heapq is generally O(N) + k * log(n) where k is the number of elements you want. So as k approaches n, it approaches n*log(n). Sorting in python has a very quick implementation especially if your data is already somewhat sorted. Hence usually I use sort() unless N is much bigger than K.

Reply • Share ›

Mikhail Golubev • 5 years ago

Hi, Sahand. First of all, very nice article!

It seems that I found a couple of places for some improvements.

In 1.13 where is no need for call to iter(), because each slice produces brand new list. So it could be just zip(*[xs[i:] for i in range(k)]).

In 1.35, it's often recommended to use functions from operator module instead of trivial lambdas, in particular as keys in sort()/sorted()/max()/min() etc. and function from itertoos, e.g. itemgetter(-1) instead of lambda r: r[-1].

Regarding 1.15. It's may be not so neat as the last lambda, but I've used the following generic purpose flatten for arbitrarily iterables a couple of times: https://gist.github.com/eas...

BTW, functools module has some great goods as well, e.g. I recently stumbled upon its awesome lru_cache() decorator, and before I had to write such memoizing decorators by myself for DP algorithms. Take a look!

1 • Reply • Share ›

sahands Mod ➦ Mikhail Golubev • 5 years ago

Great suggestions. Thank you. Regarding 1.13, I originally had a different solution using iterators in mind, which I changed to what you saw using slices instead because it was simpler, and forgot to take out the iters. But looking back at it, it really isn't a great idea to create n copies of the list for this task, and plus, slices won't work with general iterators. So I changed it back to the slightly more complicated but better version that I originally had in mind. It's updated now. Take a look and let me know what you think! (Is there an easier way to achieve what the 'advance' function does?)

I also changed the lambda in the last example to itemgetter instead. I often forget about the 'operator' namespace and just write my own lambdas so that's a good reminder.

And I agree with functools having some very useful decorators in it. I'll see if I get around to adding a few more items. Or feel free to fork and add them!

Reply • Share ›

Mikhail Golubev ➦ sahands • 5 years ago

Interesting! But, can't you use itertools.islice if you want to avoid copying of the original list, like

```
zip(*(itertools.islice(xs, k, None) for k in range(n)))
```

Also, regarding 1.28, if you're looking for interesting ways of representing tree structure, there is a cool pattern called
Bunch.

1 • Reply • Share ›

sahands Mod Mikhail Golubev • 5 years ago

Yes! islice is exactly what I was looking for. Replaced my "advance" function with islice instead. Thank you.

• Reply • Share ›

meequz . • 5 years ago

1.14 is just sum(a, []). Examples:

>>> a = [[1, 2], [3, 4], [5, 6]]
>>> sum(a, [])
[1, 2, 3, 4, 5, 6]

>>> a = [[[1, 2], [3, 4]], [5, 6]]
>>> sum(a, [])
[[1, 2], [3, 4], 5, 6]

1 • Reply • Share ›

asmeurer meequz . • 5 years ago

Unfortunately, as I once learned the hard way, sum(a, []) is very inefficient (it creates a new list at each pass). If your list of lists is quite large, this can make a huge difference. Consider:

In [7]: a = [[1, 2, 3] for i in range(100000)]

In [8]: %timeit sum(a, [])

1 loops, best of 3: 49.8 s per loop

In [9]: %timeit list(itertools.chain.from_iterable(a))

100 loops, best of 3: 9.09 ms per loop

(make note of the units on those numbers). This is similar to the common idiom that you should never use sum(a, '') or += to concatenate strings in Python, but rather always use ''.join(a).

2 • Reply • Share ›

asmeurer asmeurer • 5 years ago

Although for lists, += is also quite fast (just as fast as itertools.chain):

In [10]: %%timeit
....: b = []
....: for i in a:
....: b += i
....:

100 loops, best of 3: 9.05 ms per loop

1 • Reply • Share ›

asmeurer • 5 years ago

One thing that could be done to make this post more awesome would be to make it Python 3 compatible. After all, in Python 3, things like zip and range are iterables, which make them much better suited for the kinds of manipulations presented here.

1 • Reply • Share ›

asmeurer • 5 years ago

I think the real takeaway from this is that zip is awesome. I think it, along with zip_longest, can be used to do almost everything else from itertools. My favorite is using it to only get n items from an infinite iterable:

for i, _ in zip(i, range(n)):
...

1 • Reply • Share ›

Florian Hartl • 5 years ago

Thanks for the code snippets and some tricks I didn't know till now! Great content! Btw: In the last example, the "readlines()" is not needed --> "for line in infile" works just fine and according to the python doc is more memory efficient as well as faster.

1 • Reply • Share ›

Mark • 5 years ago

Nice tricks! In 1.14, you can replace type(x) is list with isinstance(x,list).

1 • Reply • Share ›

Robert King • 5 years ago

For 1.12 - Grouping Adjacent list items using zip, one other technique is as follows:
it = iter(list)
zip(it, it)

1 • Reply • Share ›

coolRR • 5 years ago

I'd change 1.12 into:

{value: key for key, value in d.items()}

(Or `iteritems` if you're on Python 2.x.)

1 • Reply • Share ›

LE • 5 years ago

Great summary, you have a small issue in your usage of namedtuple though; the class that inherits from the namedtuple should probably not have the same name since that would be paradoxical: a class cannot be its own superclass.

1 • Reply • Share ›

hughdbrown • 5 years ago

You can also use a slice expression to index into a list as an l-value, as in this implementation of the sieve of Eratosthenes:

```
def eras(n):
    sieve = [0, 0] + list(range(2, n + 1))
    sqn = int(round(n ** 0.5))
    it = (i for i in xrange(2, sqn + 1) if sieve[i])
    for i in it:
        sieve[i * i::i] = [0] * (n // i - i + 1)
    return filter(None, sieve)
```

return filter(None, sieve)

1 ^ | v • Reply • Share ›

hughdbrown • 5 years ago

And then there is the recursive defaultdict of defaultdicts:

```
>>> import collections
>>> def tree():
... return collections.defaultdict(tree)
...
>>> a = tree()
>>> a["a"]
defaultdict(<function tree at 0x7ff5dbb89410>, {})
>>> a["b"]
defaultdict(<function tree at 0x7ff5dbb89410>, {})
>>> a["b"]["c"]
defaultdict(<function tree at 0x7ff5dbb89410>, {})
```

1 ^ | v • Reply • Share ›

Dominic May • 5 years ago

1.13 can be replaced with `itertools.chain.from_iterable`, and the second half of 1.29 where you use splat can also use `itertools.chain.from_iterable`

1 ^ | v • Reply • Share ›

Florian Neagu • 5 years ago

Nice list, realized going through it that i didn't know a couple.

1 ^ | v • Reply • Share ›

Paddy3118 • 5 years ago

1.10 naming slices was new for me. I had known that slices was there, but your example use case illuminated.
1.24 On mapping objects to unique integers was a nice recipe that I have solved in other ways before. I might swap to your method but it may be less maintainable.

In short. Thanks for a great post!

1 ^ | v • Reply • Share ›

Joe s. • 20 days ago

Thank you sir! There are tons of "tips and tricks" sites but they mostly contain the same 20 tips. You cover those with more depth plus tips not seen in other sites. This is the go to site for python tips!

^ | v • Reply • Share ›

Kishore Venkatesh Kumar • 9 months ago

U have book Like this?
If u have a book, I buy a book

^ | v • Reply • Share ›

James Moon • a year ago

C-Style Braces is painful =))

^ | v • Reply • Share ›

Denys Contant • a year ago

wonderful article! I constantly return back to it.
I particularly like the last example in 1.17 to flatten a list of undefined depth with the list comprehension and recursion.

My suggestion would be to do it with a formal function def as recommended by PEP 8

def flatten(master_list):
if type(master_list) is list:
return [item for sublist in master_list for item in flatten(sublist)]
else:
return [master_list]

https://www.python.org/dev/...
"Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier.

Yes:

def f(x): return 2*x
No:

f = lambda x: 2*x
The first form means that the name of the resulting function object is specifically 'f' instead of the generic '<lambda>'. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e. that it can be embedded inside a larger expression)
"

• Reply • Share ›

Tcll5850 • a year ago
nice, there's only 2 things I didn't know about...

the first, I'm somewhat new to py3, but have picked up on quite a bit of the hidden stuff, the most recent being `function.__closures__`.
anyways, I didn't know about `start, *through, end = List`

the second, I'd ripped apart future in py2 and had seen all it's feature types, but never really played around with anything other than `print_function`...
so yeah, I didn't know `braces` was for C-like scopes...

recently though, id stopped using `from __future__ ...` in favor of simply injecting from `sys.modules['__future__']`, since this isn't strict about being on top, giving me abilities to do some real namespace magic...

also, collections.py and some of itertools are red flags for me, so I generally resort to more raw methods...

• Reply • Share ›

Voilin • 2 years ago
Thank you! It is really helpful))
• Reply • Share ›

nadrimajstor • 2 years ago
Not sure if it fits the format but could someone please explain how does `1.31 Mapping objects to unique counting numbers ` work?
• Reply • Share ›

Pham Tu • 4 years ago
Many thanks for you. Great post :)
• Reply • Share ›

Chris Dimoff • 4 years ago

Sahand is the man.

Reply • Share ›

Alexei Martchenko • 5 years ago

Nice post

Reply • Share ›

Load more comments

Subscribe Add Disqus to your siteAdd DisqusAdd Disqus' Privacy PolicyPrivacy PolicyPrivacy

Recommended Articles

- A Study of Python's More Advanced Features Part I: Iterators, Generators, itertools
- The Infinite In Haskell and Python
- Programmer's Guide to Setting Up a Mac OS X Machine
- From An Iterator of Iterators to Cantor's Paradise: A Deep Dive Into An Interview Question
- Understanding Asynchronous IO With Python's Asyncio And Node.js