30 Python Language Features and Tricks You May Not Know About

Posted on Mar 05, 2014 (2014-03-05T19:57:00-08:00) , last modified on May 19, 2014 (2014-05-19T10:28:00-07:00) - Permanent link

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 (#table-of-contents) is given at the end.

Update - April 9th, 2014

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.

Update - March 14th, 2014

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.

Update - March 8th, 2014

This article generated a lot of good discussion on Reddit (http://redd.it/1zv3q3 (http://redd.it/1zv3q3)), Hacker News (https://news.ycombinator.com/item?id=7365410 (https://news.ycombinator.com/item?id=7365410)), 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 (http://docs.python.org/2.7/library/functions.html#sum) 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: 'A
9'}
```

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 (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 ra
nge(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

Recommended Articles

A Study of Python's More Advanced Features Part I: Iterators, Generators, itertools

Programmer's Guide to Setting Up a Mac OS X Machine

Understanding Asynchronous IO With Python 3.4's Asyncio And Node.js

Interview Question: Grouping Word Anagrams (Facebook)

Understanding SAT by Implementing a Simple SAT Solver in Python

---