**Some Python Language Features and Tricks**

**Useful Python tricks.**

Each trick or language feature is demonstrated only through examples

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 then start a discussion.

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

**01: 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, b, c, d

(1, 2, 3, 4)

**02: Swapping Variables**

>>> a, b = 1, 2

>>> a, b = b, a

>>> a, b

(2, 1)

**03: Extended unpacking (Python 3 only)**

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

>>> a, b, c

(1, [2, 3, 4], 5)

**04: Negative indexing**

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

>>> a[-1], a[-3]

(10, 8)

**05: 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]

**06: List slices with negative indexing**

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

>>> a[-4:-2]

[7, 8]

**07: List slices with step (a[start:end:step])**

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

>>> a[::2], a[::3], a[2:8:2]

([0, 2, 4, 6, 8, 10], [0, 3, 6, 9], [2, 4, 6])

**08: List slices with negative step**

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

>>> a[::-1], a[::-2]

([10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0], [10, 8, 6, 4, 2, 0])

**09: List slice assignment**

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

>>> a[2:3] = [0, 0]

>>> a[1:1] = [8, 9]

>>> a[1:-1] = []

>>> a

[1, 5]

**10: Naming slices (slice(start, end, step)**

>>> a = [0, 1, 2, 3, 4, 5]

>>> LASTTHREE = slice(-3, None)

>>> LASTTHREE, a[LASTTHREE]

(slice(-3, None, None), [3, 4, 5])

**11: Iterating over index and value pairs (enumerate)**

>>> a = ['Hello', 'world', '!']

>>> for i, x in enumerate(a):

>>> print('{}: {}'.format(i, x))

0: Hello

1: world

2: !

**12: Iterating over dictionary key and value pairs**

>>> m = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

>>> # Note: Python 2 Use `iteritems`

>>> for k, v in m.items():

>>> print('{}: {}'.format(k, v))

d: 4

b: 2

c: 3

a: 1

**13: Zipping and unzipping lists and iterables**

>>> a = [ 1, 2, 3 ]

>>> b = ['a', 'b', 'c']

>>> z = zip(a, b)

list(zip(\*z))

[(1, 2, 3), ('a', 'b', 'c')]

**14: Grouping adjacent list items using zip**

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

>>> # Using iterators

>>> group\_adjacent = lambda a, k: list(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: list(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,)]

**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 list(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)]

**16: Inverting a dictionary using zip**

>>> m = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

>>> mi = dict(zip(m.values(), m.keys()))

>>> mi

{1: 'a', 2: 'b', 3: 'c', 4: 'd'}

**17: Flattening lists:**

Note: According to Python's documentation sum and chain.from\_iterable is the preferred method.

**Singly Nested List**

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

>>> a

[[1, 2], [3, 4], [5, 6]]

**Using Chain**

>>> from itertools import chain

>>> list(chain.from\_iterable(a))

[1, 2, 3, 4, 5, 6]

**Using sum**

>>> sum(a, [])

[1, 2, 3, 4, 5, 6]

**Using List Comprehension**

>>> [x for l in a for x in l]

[1, 2, 3, 4, 5, 6]

**Now a Doubly Nested List**

>>> a = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]

**List Comprehension**

>>> [x for l1 in a for l2 in l1 for x in l2]

[1, 2, 3, 4, 5, 6, 7, 8]

**Random Nested List**

>>> a = [1, 2, [3, 4], [[5, 6], [7, 8]]]

**Generic Solution**

>>> def flatten(nested):

>>> return (

>>> [x for l in nested for x in flatten(l)]

>>> if isinstance(nested, list) else

>>> [nested]

>>> )

>>> flatten(a)

[1, 2, 3, 4, 5, 6, 7, 8]

**1.18: Generator expressions**

>>> g = (x \*\* 2 for x in range(10))

>>> g

<generator object <genexpr> at 0x7f1c30bbc5a0>

Doesn't do anythin unless you iterate over it

>>> for f in g:

>>> print(f)

0

1

4

9

16

25

36

49

64

81

Or deplete it using list

>>> list(x \*\* 2 for x in range(10))

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Just think of them as lazy list comprehensions

>>> [x \*\* 2 for x in range(10)]

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Except generators get depleted

>>> for more\_f in g:

>>> print(more\_f)

Generator expressions yield using next()

>>> g = (x \*\* 2 for x in range(10))

>>> next(g)

0

>>> next(g)

1

>>> next(g)

4

**1.19: Dictionary comprehensions**

>>> m = {x: x \*\* 2 for x in range(5)}

>>> m

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

{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'}

**20: Inverting a dictionary using a dictionary comprehension**

>>> m = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

>>> {v: k for k, v in m.items()}

{1: 'a', 2: 'b', 3: 'c', 4: 'd'}

**21: namedtuple**

>>> from collections import namedtuple

>>> Point = namedtuple('Point', ['x', 'y'])

>>> point = Point(x=1.0, y=2.0)

>>> point.x, point.y

(1.0, 2.0)

**22: Inheriting from namedtuple**

>>> PointBaseClass = namedtuple('PointBaseClass', ['x', 'y'])

>>> class Point(PointBaseClass):

>>> \_\_slots\_\_ = ()

>>> def \_\_add\_\_(self, other):

>>> return Point((self.x + other.x), (self.y + other.y))

>>> x = Point(1.0, 2.0)

>>> y = Point(2.0, 3.0)

>>> x + y

Point(x=3.0, y=5.0)

**23: set operations**

**Set Constructor**

>>> A = set()

>>> B = set()

**Set Literal Notation**

>>> A = {1, 2, 3, 4}

>>> B = {3, 4, 5, 6, 7}

**Set Converted from List**

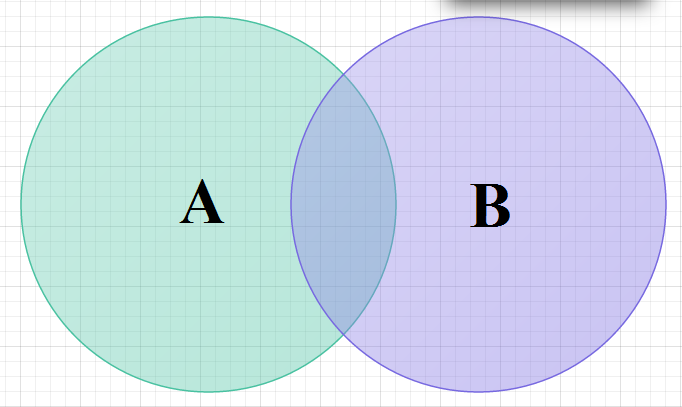
>>> A\_ = set([1, 2, 3, 4])

>>> B\_ = set([3, 4, 5, 6, 7])

>>> A == A\_ and B == B\_

True

**Union Operator**



>>> A = {1, 2, 3, 4}

>>> B = {3, 4, 5, 6, 7}

>>> A | B

{1, 2, 3, 4, 5, 6, 7}

**Intersection Operator**

>>> A = {1, 2, 3, 4}

>>> B = {3, 4, 5, 6, 7}

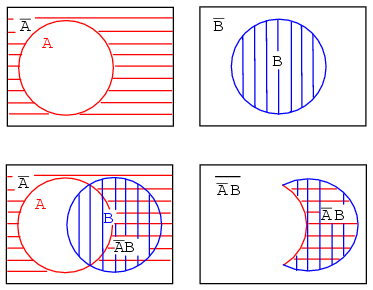
>>> A & B

**Difference Operator**

>>> A = {1, 2, 3, 4}

>>> B = {3, 4, 5, 6, 7}

>>> A - B

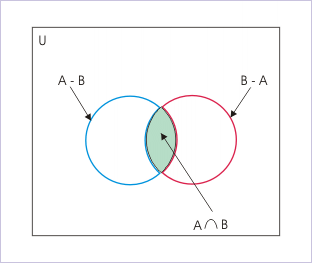


**Symmetric Difference Operator**

>>> A ^ B

**Putting it Together**

>>> (A ^ B) == ((A - B) | (B - A))



**24: Multisets and multiset operations (collections.Counter)**

>>> from collections import Counter

>>> # Construct a Counter from a List

>>> A = Counter([1, 2, 3])

>>> B = Counter([2, 2, 3])

>>> # Union Operator

>>> A | B

>>> # Intersection Operator

>>> A & B

>>> # Addition Operator

>>> A + B

>>> # Subtraction Operator

>>> A - B, B - A

**30: Simple Dictionary Trees (**[**Autovivification**](http://en.wikipedia.org/wiki/Autovivification)**)**

>>> import json

>>> from collections import defaultdict

>>> # Recursive Definition

>>> Tree = lambda: defaultdict(Tree)

>>> root = Tree()

>>> # Instantiation

>>> 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=2, 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.)

>>> # Using `\_\_missing\_\_`

>>> class Tree(dict):

>>> def \_\_missing\_\_(self, key):

>>> value = self[key] = type(self)()

>>> return value

>>> 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=2, separators=(',', ': ')))

{

"menu": {

"id": "file",

"menuitems": {

"close": {

"onclick": "close();",

"value": "Close"

},

"new": {

"onclick": "new();",

"value": "New"

},

"open": {

"onclick": "open();",

"value": "Open"

}

},

"value": "File"

}

}

**Scientific Python Tricks**

>>> import numpy as np

>>> ## Learning to avoid unnecessary array copies

>>> def id(x):

>>> "Returns identifies the memory block address of the array"

>>> return x.\_\_array\_interface\_\_['data'][0]

These two arrays of zeros occupy different blocks of memory

>>> a = np.zeros(10)

>>> b = np.zeros(10)

>>> id(a) != id(b)

But the spacing between memory is fixed width.

>>> id(a[3:]) - id(a[2:]) == id(a[2:]) - id(a[1:])

**In-place and implicit copy operations**

Memory is copied implicitly

>>> a = np.arange(10)

>>> b = a \* 2

>>> id(a) == id(b)

Memory is manipulated in place and a much cheaper operation

>>> a = np.arange(10)

>>> b = a

>>> a \*= 2

>>> id(b) == id(a)

True

**Broadcasting**

Broadcasting rules allow you to make computations on arrays with different but compatible shapes. In other words, you don't always need to reshape or tile your arrays to make their shapes match. The following example illustrates two ways of doing an outer product between two vectors: the first method involves array tiling, the second one involves broadcasting. The last method is significantly faster.

>>> n = 1000

>>> a = np.arange(n)

>>> ac = a[:, np.newaxis]

>>> ar = a[np.newaxis, :]

>>> %timeit np.tile(ac, (1, n)) \* np.tile(ar, (n, 1))

10 loops, best of 3: 20.8 ms per loop

>>> %timeit ar \* ac

100 loops, best of 3: 2.33 ms per loop

**Slicing**

Array views refer to the original data buffer of an array, but with different offsets, shapes and strides. They only permit strided selections (i.e. with linearly spaced indices). NumPy also offers specific functions to make arbitrary selections along one axis.

Fancy indexing is the most general selection method, but it is also the slowest as we will see in this recipe.

Faster alternatives should be chosen when possible.

**ETC..**