

## data structures & algorithms

### 1. Unpacking a Sequence into Separate Variables

```
In [0]: p = (4, 5)
        x, y = p
```

```
In [6]: print(x, y, sep="\n")

4
5
```

```
In [0]: data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
        name, shares, price, date = data
```

```
In [7]: print(name, date, sep="\n")

ACME
(2012, 12, 21)
```

```
In [0]: name, shares, price, (year, mon, day) = data
```

```
In [9]: print(name, year, mon, day, sep="\n")

ACME
2012
12
21
```

```
In [0]: def drop_first_last(grades):
        first, *middle, last = grades
        return avg(middle)
```

```
In [11]: record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
        name, email, *phone_numbers = record

        print(name, email, phone_numbers, sep="\n")

Dave
dave@example.com
['773-555-1212', '847-555-1212']
```

```
In [12]: records = [
        ('foo', 1, 2),
        ('bar', 'hello'),
        ('foo', 3, 4),
        ]

        def do_foo(x, y):
            print('foo', x, y)

        def do_bar(s):
            print('bar', s)

        for tag, *args in records:
            if tag == 'foo':
                do_foo(*args)
            elif tag == 'bar':
                do_bar(*args)

foo 1 2
bar hello
foo 3 4
```

```
In [0]: items = [1, 10, 7, 4, 5, 9]
        head, *tail = items
```

```
In [14]: print(head, tail, sep="\n")

1
[10, 7, 4, 5, 9]
```

```
In [15]: def sum(items):
```

```

    head, *tail = items
    return head + sum(tail) if tail else head

sum(items)

```

Out[15]: 36

In [0]:

## 1. collections.deque

Using deque(maxlen=N) creates a fixed-sized queue. When new items are added and the queue is full, the oldest item is automatically removed. For example:

In [0]: **from collections import deque**

```

q = deque(maxlen=3)
q.append(1)
q.append(2)
q.append(3)

```

In [17]: q

Out[17]: deque([1, 2, 3])

In [0]: q.append(4)

In [19]: q

Out[19]: deque([2, 3, 4])

In [20]: q.append(5)  
q

Out[20]: deque([3, 4, 5])

In [22]: *#operations*  
q = deque()  
q.append(1)  
q.append(2)  
q.append(3)  
q

Out[22]: deque([1, 2, 3])

In [23]: q.appendleft(4)  
q

Out[23]: deque([4, 1, 2, 3])

In [24]: q.pop()  
q

Out[24]: deque([4, 1, 2])

In [25]: q.popleft()  
q

Out[25]: deque([1, 2])

## 1. heapq module

In [26]: **import heapq**

```

nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]

```

```

print(heapq.nlargest(3, nums))
print(heapq.nsmallest(3, nums))

```

```

[42, 37, 23]
[-4, 1, 2]

```

In [0]: portfolio = [  
 {'name': 'IBM', 'shares': 100, 'price': 91.1},

```

{'name': 'AAPL', 'shares': 50, 'price': 543.22},
{'name': 'FB', 'shares': 200, 'price': 21.09},
{'name': 'HPQ', 'shares': 35, 'price': 31.75},
{'name': 'YHOO', 'shares': 45, 'price': 16.35},
{'name': 'ACME', 'shares': 75, 'price': 115.65}
]

cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])

```

```

In [28]: print(cheap)
         print(expensive)

[{'name': 'YHOO', 'shares': 45, 'price': 16.35}, {'name': 'FB', 'shares': 200, 'price': 21.09}, {'name': 'HPQ', 'shares': 35, 'price': 31.75}]
[{'name': 'AAPL', 'shares': 50, 'price': 543.22}, {'name': 'ACME', 'shares': 75, 'price': 115.65}, {'name': 'IBM', 'shares': 100, 'price': 91.1}]

```

```

In [29]: nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]

```

```

import heapq

heap = list(nums)
heapq.heapify(heap)
heap

```

```

Out[29]: [-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]

```

```

In [30]: """heapq.heappop() method, which pops off the first item and
         replaces it with the next smallest item (an operation that requires O(Log N)
         operations where N is the size of the heap)."""
         heapq.heappop(heap)

```

```

Out[30]: -4

```

```

In [31]: heapq.heappop(heap)

```

```

Out[31]: 1

```

**Mapping Keys to Multiple Values in a Dictionary** defaultdict in the collections module. A feature of defaultdict is that it automatically initializes the first value so you can simply focus on adding items.

```

In [2]: from collections import defaultdict

```

```

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
d

```

```

Out[2]: defaultdict(list, {'a': [1, 2], 'b': [4]})

```

```

In [4]: d['a'][0]

```

```

Out[4]: 1

```

```

In [6]: d = defaultdict(set)
         d['a'].add(1)
         d['a'].add(2)
         d['a'].add(2)
         d['b'].add(4)
         d

```

```

Out[6]: defaultdict(set, {'a': {1, 2}, 'b': {4}})

```

```

In [0]:

```

**OrderedDict** from the collections module. It exactly preserves the original insertion order of data when iterating.

```

In [7]: from collections import OrderedDict

```

```

d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
d['spam'] = 3

```

```
d['grok'] = 4

# Outputs "foo 1", "bar 2", "spam 3", "grok 4"
for key in d:
    print(key, d[key])
```

```
foo 1
bar 2
spam 3
grok 4
```

In [10]: `import json`

```
json.dumps(d)
```

Out[10]: `'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'`

In [0]:

### Determining the Most Frequently Occurring Items in a Sequence `collections.Counter`

```
In [12]: words = [
    'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
    'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',
    'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',
    'my', 'eyes', "you're", 'under'
]

from collections import Counter
word_counts = Counter(words)
top_three = word_counts.most_common(3)
print(top_three)
# Outputs [('eyes', 8), ('the', 5), ('look', 4)]

[('eyes', 8), ('the', 5), ('look', 4)]
```

```
In [13]: morewords = ['why', 'are', 'you', 'not', 'looking', 'in', 'my', 'eyes']
a = Counter(words)
b = Counter(morewords)
a+b
```

```
Out[13]: Counter({'are': 1,
    'around': 2,
    "don't": 1,
    'eyes': 9,
    'in': 1,
    'into': 3,
    'look': 4,
    'looking': 1,
    'my': 4,
    'not': 2,
    'the': 5,
    'under': 1,
    'why': 1,
    'you': 1,
    "you're": 1})
```

In [0]:

### Grouping Records Together Based on a Field `itertools.groupby()`

```
In [0]: rows = [
    {'address': '5412 N CLARK', 'date': '07/01/2012'},
    {'address': '5148 N CLARK', 'date': '07/04/2012'},
    {'address': '5800 E 58TH', 'date': '07/02/2012'},
    {'address': '2122 N CLARK', 'date': '07/03/2012'},
    {'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'},
    {'address': '1060 W ADDISON', 'date': '07/02/2012'},
    {'address': '4801 N BROADWAY', 'date': '07/01/2012'},
    {'address': '1039 W GRANVILLE', 'date': '07/04/2012'},
]
```

```
In [15]: from operator import itemgetter
from itertools import groupby

# Sort by the desired field first
```

```
rows.sort(key=itemgetter('date'))

# Iterate in groups
for date, items in groupby(rows, key=itemgetter('date')):
    print(date)
    for i in items:
        print('    ', i)

07/01/2012
{'address': '5412 N CLARK', 'date': '07/01/2012'}
{'address': '4801 N BROADWAY', 'date': '07/01/2012'}
07/02/2012
{'address': '5800 E 58TH', 'date': '07/02/2012'}
{'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'}
{'address': '1060 W ADDISON', 'date': '07/02/2012'}
07/03/2012
{'address': '2122 N CLARK', 'date': '07/03/2012'}
07/04/2012
{'address': '5148 N CLARK', 'date': '07/04/2012'}
{'address': '1039 W GRANVILLE', 'date': '07/04/2012'}
```

```
In [16]: addresses = [
    '5412 N CLARK',
    '5148 N CLARK',
    '5800 E 58TH',
    '2122 N CLARK',
    '5645 N RAVENSWOOD',
    '1060 W ADDISON',
    '4801 N BROADWAY',
    '1039 W GRANVILLE',
    ]

counts = [ 0, 3, 10, 4, 1, 7, 6, 1]
from itertools import compress
more5 = [n > 5 for n in counts]
more5  # [False, False, True, False, False, True, True, False]

list(compress(addresses, more5))
```

```
Out[16]: ['5800 E 58TH', '1060 W ADDISON', '4801 N BROADWAY']
```

```
In [0]:
```

`collections.namedtuple()` provides these benefits, while adding minimal overhead over using a normal tuple object. `collections.namedtuple()` is actually a factory method that returns a subclass of the standard Python tuple type.

```
In [18]: from collections import namedtuple
Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
sub = Subscriber('jonesy@example.com', '2012-10-19')
sub  # Subscriber(addr='jonesy@example.com', joined='2012-10-19')
```

```
Out[18]: Subscriber(addr='jonesy@example.com', joined='2012-10-19')
```

```
In [0]:
```

### Combining Multiple Mappings into a Single Mapping

```
In [0]: a = {'x': 1, 'z': 3 }
        b = {'y': 2, 'z': 4 }
```

```
In [20]: from collections import ChainMap
c = ChainMap(a,b)
print(c['x'])    # Outputs 1 (from a)
print(c['y'])    # Outputs 2 (from b)
print(c['z'])    # Outputs 3 (from a)
```

```
1
2
3
```

```
In [21]: values = ChainMap()
values['x'] = 1

# Add a new mapping
values = values.new_child()
values['x'] = 2
```

```
# Add a new mapping
values = values.new_child()
values['x'] = 3
values
ChainMap({'x': 3}, {'x': 2}, {'x': 1})
values['x']  # 3

# Discard last mapping
values = values.parents
values['x']  # 2

# Discard last mapping
values = values.parents
values['x']  # 1
```