1.1 解压序列赋值给多个变量¶

问题¶

现在有一个包含 N 个元素的元组或者是序列，怎样将它里面的值解压后同时赋值给 N 个变量？

解决方案¶

任何的序列（或者是可迭代对象）可以通过一个简单的赋值语句解压并赋值给多个变量。唯一的前提就是变量的数量必须跟序列元素的数量是一样的。

代码示例：

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> name, shares, price, date = data
>>> name
'ACME'
>>> date
(2012, 12, 21)
>>> name, shares, price, (year, mon, day) = data
>>> name
'ACME'
>>> year
2012
>>> mon
12
>>> day
21
>>>
```

如果变量个数和序列元素的个数不匹配，会产生一个异常。

代码示例：

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>>
```

讨论¶

实际上，这种解压赋值可以用在任何可迭代对象上面，而不仅仅是列表或者元组。包括字符串，文件对象，迭代器和生成器。

代码示例：

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
>>>
```

有时候，你可能只想解压一部分，丢弃其他的值。对于这种情况 Python 并没有提供特殊的语法。但是你可以使用任意变量名去占位，到时候丢掉这些变量就行了。

代码示例：

```
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>
```

你必须保证你选用的那些占位变量名在其他地方没被使用到。

1.2 解压可迭代对象赋值给多个变量¶

问题¶

如果一个可迭代对象的元素个数超过变量个数时，会抛出一个 `ValueError` 。 那么怎样才能从这个可迭代对象中解压出 N 个元素出来？

解决方案¶

Python 的星号表达式可以用来解决这个问题。比如，你在学习一门课程，在学期末的时候，你想统计下家庭作业的平均成绩，但是排除掉第一个和最后一个分数。如果只有四个分数，你可能就直接去简单的手动赋值，但如果有 24 个呢？这时候星号表达式就派上用场了：

```
def drop_first_last(grades):
    first, *middle, last = grades
    return avg(middle)
```

另外一种情况，假设你现在有一些用户的记录列表，每条记录包含一个名字、邮件，接着就是不确定数量的电话号码。你可以像下面这样分解这些记录：

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

值得注意的是上面解压出的 `phone_numbers` 变量永远都是列表类型，不管解压的电话号码数量是多少（包括 0 个），所以，任何使用到 `phone_numbers` 变量的代码就不需要做多余的类型检查去确认它是否是列表类型了。

星号表达式也能用在列表的开始部分。比如，你有一个公司前 8 个月销售数据的序列，但是你想看下最近一个月数据和前面 7 个月的平均值的对比。你可以这样做：

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

下面是在 Python 解释器中执行的结果：

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

讨论¶

扩展的迭代解压语法是专门为解压不确定个数或任意个数元素的可迭代对象而设计的。通常，这些可迭代对象的元素结构有确定的规则（比如第 1 个元素后面都是电话号码）。星号表达式让开发人员可以很容易的利用这些规则来解压出元素来。而不是通过一些比较复杂的手段去获取这些关联的元素值。

值得注意的是，星号表达式在迭代元素为可变长元组的序列时是很有用的。比如，下面是一个带有标签的元组序列：

```
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)
```

星号解压语法在字符串操作的时候也会很有用，比如字符串的分割。

代码示例：

```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

有时候，你想解压一些元素后丢弃它们，你不能简单就使用 `*` ，但是你可以使用一个普通的废弃名称，比如 `_` 或者 `ign` （ignore）。

代码示例：

```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_, (*_, year) = record
>>> name
'ACME'
>>> year
2012
>>>
```

在很多函数式语言中，星号解压语法跟列表处理有许多相似之处。比如，如果你有一个列表，你可以很容易的将它分割成前后两部分：

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

如果你够聪明的话，还能用这种分割语法去巧妙的实现递归算法。比如：

```
>>> def sum(items):
...     head, *tail = items
...     return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```

然后，由于语言层面的限制，递归并不是 Python 擅长的。因此，最后那个递归演示仅仅是个好奇的探索罢了，对这个不要太认真了。

1.3 保留最后 N 个元素¶

问题¶

在迭代操作或者其他操作的时候，怎样只保留最后有限几个元素的历史记录？

解决方案¶

保留有限历史记录正是 `collections.deque` 大显身手的时候。比如，下面的代码在多行上面做简单的文本匹配，并返回匹配所在行的最后N行：

```
from collections import deque


def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for line in lines:
        if pattern in line:
            yield line, previous_lines
        previous_lines.append(line)

# Example use on a file
if __name__ == '__main__':
    with open(r'../../Cookbook/somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end='')
            print(line, end='')
            print('-' * 20)
```

讨论¶

我们在写查询元素的代码时，通常会使用包含 `yield` 表达式的生成器函数，也就是我们上面示例代码中的那样。这样可以将搜索过程代码和使用搜索结果代码解耦。如果你还不清楚什么是生成器，请参看 4.3 节。

使用 `deque(maxlen=N)` 构造函数会新建一个固定大小的队列。当新的元素加入并且这个队列已满的时候，最老的元素会自动被移除掉。

代码示例：

```
>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
```

尽管你也可以手动在一个列表上实现这一的操作（比如增加、删除等等）。但是这里的队列方案会更加优雅并且运行得更快些。

更一般的， `deque` 类可以被用在任何你只需要一个简单队列数据结构的场合。 如果你不设置最大队列大小，那么就会得到一个无限大小队列。你可以在队列的两端执行添加和弹出元素的操作。

代码示例：

```
>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
>>> q.popleft()
4
```

在队列两端插入或删除元素时间复杂度都是 `O(1)` 。而在列表的开头插入或删除元素的时间复杂度为 `O(N)` 。

1.4 查找最大或最小的 N 个元素¶

问题¶

怎样从一个集合中获得最大或者最小的 N 个元素列表?

解决方案¶

heapq 模块有两个函数： `nlargest()` 和 `nsmallest()` 可以完美解决这个问题。

```
import heapq
nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums)) # Prints [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Prints [-4, 1, 2]
```

两个函数都能接受一个关键字参数，用于更复杂的数据结构中：

```
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'])
```

译者注：上面代码在对每个元素进行对比的时候，会以 `price` 的值进行比较。

讨论¶

如果你想在一个集合中查找最小或最大的 N 个元素，并且 N 小于集合元素数量，那么这些函数提供了很好的性能。 因为在底层实现里面，首先会先将集合数据进行堆排序后放入一个列表中：

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq
>>> heap = list(nums)
>>> heapq.heapify(heap)
>>> heap
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
>>>
```

堆数据结构最重要的特征是 `heap[0]` 永远是最小的元素。并且剩余的元素可以很容易的通过调用 `heapq.heappop()` 方法得到，该方法会先将第一个元素弹出来。然后用下一个最小的元素来取代被弹出元素（这种操作时间复杂度仅仅是 O(log N)，N 是堆大小）。 比如，如果想要查找最小的 3 个元素，你可以这样做：

```
>>> heapq.heappop(heap)
-4
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
2
```

当要查找的元素个数相对比较小的时候，函数 `nlargest()` 和 `nsmallest()` 是很合适的。 如果你仅仅想查找唯一的最小或最大（N=1）的元素的话，那么使用 `min()` 和 `max()` 函数会更快些。 类似的，如果 N 的大小和集合大小接近的时候，通常先排序这个集合然后再使用切片操作会更快点 （ `sorted(items)[:N]` 或者是 `sorted(items)[-N:]` ）。 需要在正确场合使用函数 `nlargest()` 和 `nsmallest()` 才能发挥它们的优势 （如果 N 快接近集合大小了，那么使用排序操作会更好些）。

尽管你没有必要一定使用这里的方法，但是堆数据结构的实现是一个很有趣并且值得你深入学习的东西， 基本上只要是数据结构和算法书籍里面都会有提及到。 heapq 模块的官方文档里面也详细的介绍了堆数据结构底层的实现细节。

1.5 实现一个优先级队列¶

问题¶

怎样实现一个按优先级排序的队列？ 并且在这个队列上面每次 pop 操作总是返回优先级最高的那个元素

解决方案¶

下面的类利用 heapq 模块实现了一个简单的优先级队列：

```
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]
```

下面是它的使用方式：

```
>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>
```

仔细观察可以发现，第一个 `pop()` 操作返回优先级最高的元素。 另外注意到如果两个有着相同优先级的元素（ `foo` 和 `grok` ），pop 操作按照它们被插入到队列的顺序返回的。

讨论¶

这一小节我们主要关注 heapq 模块的使用。函数 `heapq.heappush()` 和 `heapq.heappop()` 分别在队列 `_queue` 上插入和删除第一个元素， 并且队列 `_queue` 保证第一个元素拥有最高优先级（ 1.4 节已经讨论过这个问题）。 `heappop()` 函数总是返回”最小的”的元素，这就是保证队列pop操作返回正确元素的关键。 另外，由于 push 和 pop 操作时间复杂度为 O(log N)，其中 N 是堆的大小，因此就算是 N 很大的时候它们运行速度也依旧很快。

在上面代码中，队列包含了一个 `(-priority, index, item)` 的元组。 优先级为负数的目的是使得元素按照优先级从高到低排序。 这个跟普通的按优先级从低到高排序的堆排序恰巧相反。

`index` 变量的作用是保证同等优先级元素的正确排序。 通过保存一个不断增加的 `index` 下标变量，可以确保元素按照它们插入的顺序排序。 而且， `index` 变量也在相同优先级元素比较的时候起到重要作用。

为了阐明这些，先假定 `Item` 实例是不支持排序的：

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

如果你使用元组 `(priority, item)` ，只要两个元素的优先级不同就能比较。 但是如果两个元素优先级一样的话，那么比较操作就会跟之前一样出错：

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

通过引入另外的 `index` 变量组成三元组 `(priority, index, item)` ，就能很好的避免上面的错误， 因为不可能有两个元素有相同的 `index` 值。Python 在做元组比较时候，如果前面的比较已经可以确定结果了， 后面的比较操作就不会发生了：

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
True
>>>
```

如果你想在多个线程中使用同一个队列，那么你需要增加适当的锁和信号量机制， 可以查看 12.3 小节的例子演示是怎样做的。

heapq 模块的官方文档有更详细的例子程序以及对于堆理论及其实现的详细说明。

1.6 字典中的键映射多个值¶

问题¶

怎样实现一个键对应多个值的字典（也叫 multidict）？

解决方案¶

一个字典就是一个键对应一个单值的映射。如果你想要一个键映射多个值，那么你就需要将这多个值放到另外的容器中，比如列表或者集合里面。比如，你可以像下面这样构造这样的字典：

```
d = {
    'a' : [1, 2, 3],
    'b' : [4, 5]
}
e = {
    'a' : {1, 2, 3},
    'b' : {4, 5}
}
```

选择使用列表还是集合取决于你的实际需求。如果你想保持元素的插入顺序就应该使用列表， 如果想去掉重复元素就使用集合（并且不关心元素的顺序问题）。

你可以很方便的使用 collections 模块中的 `defaultdict` 来构造这样的字典。 `defaultdict` 的一个特征是它会自动初始化每个 key 刚开始对应的值，所以你只需要关注添加元素操作了。比如：

```
from collections import defaultdict

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

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

需要注意的是， `defaultdict` 会自动为将要访问的键（就算目前字典中并不存在这样的键）创建映射实体。 如果你并不需要这样的特性，你可以在一个普通的字典上使用 `setdefault()` 方法来代替。比如：

```
d = {} # A regular dictionary
d.setdefault('a', []).append(1)
d.setdefault('a', []).append(2)
```

```
d.setdefault('b', []).append(4)
```

但是很多程序员觉得 `setdefault()` 用起来有点别扭。因为每次调用都得创建一个新的初始值的实例（例子程序中的空列表 `[]`）。

讨论¶

一般来讲，创建一个多值映射字典是很简单的，但是，如果你选择自己实现的话，那么对于值的初始化可能会有点麻烦，你可能会像下面这样来实现：

```
d = {}
for key, value in pairs:
    if key not in d:
        d[key] = []
    d[key].append(value)
```

如果使用 `defaultdict` 的话代码就更加简洁了：

```
d = defaultdict(list)
for key, value in pairs:
    d[key].append(value)
```

这一小节所讨论的问题跟数据处理中的记录归类问题有大的关联，可以参考 1.15 小节的例子。

1.7 字典排序¶

问题¶

你想创建一个字典，并且在迭代或序列化这个字典的时候能够控制元素的顺序。

解决方案¶

为了能控制一个字典中元素的顺序，你可以使用 `collections` 模块中的 `OrderedDict` 类。在迭代操作的时候它会保持元素被插入时的顺序，示例如下：

```
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])
```

当你想要构建一个将来需要序列化或编码成其他格式的映射的时候，`OrderedDict` 是非常有用的。比如，你想精确控制以 JSON 编码后字段的顺序，你可以先使用 `OrderedDict` 来构建这样的数据：

```
>>> import json
>>> json.dumps(d)
'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'
>>>
```

讨论¶

`OrderedDict` 内部维护着一个根据键插入顺序排序的双向链表。每次当一个新的元素插入进来的时候，它会被放到链表的尾部。对于一个已经存在的键的重复赋值不会改变键的顺序。

需要注意的是，一个 `OrderedDict` 的大小是一个普通字典的两倍，因为它内部维护着另外一个链表。所以如果你要构建一个需要大量 `OrderedDict` 实例的数据结构的时候（比如读取 100,000 行 CSV 数据到一个 `OrderedDict` 列表中去），那么你就得仔细权衡一下是否使用 `OrderedDict` 带来的好处要大过额外内存消耗的影响。

1.8 字典的运算¶

问题¶

怎样在数据字典中执行一些计算操作（比如求最小值、最大值、排序等等）？

解决方案¶

考虑下面的股票名和价格映射字典：

```
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
```

为了对字典值执行计算操作，通常需要使用 `zip()` 函数先将键和值反转过来。比如，下面是查找最小和最大股票价格和股票值的代码：

```
min_price = min(zip(prices.values(), prices.keys()))
# min_price is (10.75, 'FB')
max_price = max(zip(prices.values(), prices.keys()))
# max_price is (612.78, 'AAPL')
```

类似的，可以使用 `zip()` 和 `sorted()` 函数来排列字典数据：

```
prices_sorted = sorted(zip(prices.values(), prices.keys()))
# prices_sorted is [(10.75, 'FB'), (37.2, 'HPQ'),
#                   (45.23, 'ACME'), (205.55, 'IBM'),
#                   (612.78, 'AAPL')]
```

执行这些计算的时候，需要注意的是 `zip()` 函数创建的是一个只能访问一次的迭代器。比如，下面的代码就会产生错误：

```
prices_and_names = zip(prices.values(), prices.keys())
print(min(prices_and_names)) # OK
print(max(prices_and_names)) # ValueError: max() arg is an empty sequence
```

讨论¶

如果你在一个字典上执行普通的数学运算，你会发现它们仅仅作用于键，而不是值。比如：

```
min(prices) # Returns 'AAPL'
max(prices) # Returns 'IBM'
```

这个结果并不是你想要的，因为你想要在字典的值集合上执行这些计算。或许你会尝试着使用字典的 `values()` 方法来解决这个问题：

```
min(prices.values()) # Returns 10.75
max(prices.values()) # Returns 612.78
```

不幸的是，通常这个结果同样也不是你想要的，你可能还想要知道对应的键的信息（比如那种股票价格是最低的？）。

你可以在 `min()` 和 `max()` 函数中提供 `key` 函数参数来获取最小值或最大值对应的键的信息，比如：

```
min(prices, key=lambda k: prices[k]) # Returns 'FB'
max(prices, key=lambda k: prices[k]) # Returns 'AAPL'
```

但是，如果还想要得到最小值，你又得执行一次查找操作。比如：

```
min_value = prices[min(prices, key=lambda k: prices[k])]
```

前面的 `zip()` 函数方案通过将字典"反转"为 (值，键) 元组序列来解决了上述问题。当比较两个元组的时候，值会先进行比较，然后才是键。这样的话你就能通过一条简单的语句就能很轻松的实现在字典上的求最值和排序操作了。

需要注意的是在计算操作中使用到了 (值，键) 对。当多个实体拥有相同的值的时候，键会决定返回结果。比如，在执行 `min()` 和 `max()` 操作的时候，如果恰巧最小或最大值有重复的，那么拥有最小或最大键的实体会返回：

```
>>> prices = { 'AAA' : 45.23, 'ZZZ': 45.23 }
>>> min(zip(prices.values(), prices.keys()))
(45.23, 'AAA')
>>> max(zip(prices.values(), prices.keys()))
(45.23, 'ZZZ')
>>>
```

1.9 查找两字典的相同点¶

问题¶

怎样在两个字典中寻找相同点（比如相同的键、相同的值等等）？

解决方案¶

考虑下面两个字典：

```
a = {
    'x' : 1,
    'y' : 2,
    'z' : 3
}

b = {
    'w' : 10,
    'x' : 11,
    'y' : 2
}
```

为了寻找两个字典的相同点，可以简单的在两字典的 `keys()` 或者 `items()` 方法返回结果上执行集合操作。比如：

```
# Find keys in common
a.keys() & b.keys() # { 'x', 'y' }
# Find keys in a that are not in b
a.keys() - b.keys() # { 'z' }
# Find (key,value) pairs in common
a.items() & b.items() # { ('y', 2) }
```

这些操作也可以用于修改或者过滤字典元素。比如，假如你想以现有字典构造一个排除几个指定键的新字典。下面利用字典推导来实现这样的需求：

```
# Make a new dictionary with certain keys removed
c = {key:a[key] for key in a.keys() - {'z', 'w'}}
# c is {'x': 1, 'y': 2}
```

讨论¶

一个字典就是一个键集合与值集合的映射关系。字典的 `keys()` 方法返回一个展现键集合的键视图对象。键视图的一个很少被了解的特性就是它们也支持集合操作，比如集合并、交、差运算。所以，如果你想对集合的键执行一些普通的集合操作，可以直接使用键视图对象而不用先将它们转换成一个 `set`。

字典的 `items()` 方法返回一个包含 (键，值) 对的元素视图对象。这个对象同样也支持集合操作，并且可以被用来查找两个字典有哪些相同的键值对。

尽管字典的 `values()` 方法也是类似，但是它并不支持这里介绍的集合操作。某种程度上是因为值视图不能保证所有的值互不相同，这样会导致某些集合操作会出现问题。不过，如果你硬要在值上面执行这些集合操作的话，你可以先将值集合转换成 `set`，然后再执行集合运算就行了。

1.10 删除序列相同元素并保持顺序¶

问题¶

怎样在一个序列上面保持元素顺序的同时消除重复的值？

解决方案¶

如果序列上的值都是 `hashable` 类型，那么可以很简单的利用集合或者生成器来解决这个问题。比如：

```
def dedupe(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)
```

下面是使用上述函数的例子：

```
>>> a = [1, 5, 2, 1, 9, 1, 5, 10]
>>> list(dedupe(a))
[1, 5, 2, 9, 10]
>>>
```

这个方法仅仅在序列中元素为 `hashable` 的时候才管用。如果你想消除元素不可哈希（比如 `dict` 类型）的序列中重复元素的话，你需要将上述代码稍微改变一下，就像这样：

```
def dedupe(items, key=None):
    seen = set()
    for item in items:
        val = item if key is None else key(item)
        if val not in seen:
            yield item
            seen.add(val)
```

这里的`key`参数指定了一个函数，将序列元素转换成 `hashable` 类型。下面是它的用法示例：

```
>>> a = [ {'x':1, 'y':2}, {'x':1, 'y':3}, {'x':1, 'y':2}, {'x':2, 'y':4}]
>>> list(dedupe(a, key=lambda d: (d['x'],d['y'])))
[{'x': 1, 'y': 2}, {'x': 1, 'y': 3}, {'x': 2, 'y': 4}]
>>> list(dedupe(a, key=lambda d: d['x']))
[{'x': 1, 'y': 2}, {'x': 2, 'y': 4}]
>>>
```

如果你想基于单个字段、属性或者某个更大的数据结构来消除重复元素，第二种方案同样可以胜任。

讨论¶

如果你仅仅就是想消除重复元素，通常可以简单的构造一个集合。比如：

```
>>> a
[1, 5, 2, 1, 9, 1, 5, 10]
>>> set(a)
{1, 2, 10, 5, 9}
>>>
```

然而，这种方法不能维护元素的顺序，生成的结果中的元素位置被打乱。而上面的方法可以避免这种情况。

在本节中我们使用了生成器函数让我们的函数更加通用。不仅仅是局限于列表处理。比如，如果如果你想读取一个文件，消除重复行，你可以很容易像这样做：

```
with open(somefile,'r') as f:
for line in dedupe(f):
    ...
```

上述key函数参数模仿了 `sorted()`, `min()` 和 `max()` 等内置函数的相似功能。 可以参考 1.8 和 1.13 小节了解更多。

1.11 命名切片¶

问题¶

你的程序已经出现一大堆已无法直视的硬编码切片下标，然后你想清理下代码。

解决方案¶

假定你有一段代码要从一个记录字符串中几个固定位置提取出特定的数据字段（比如文件或类似格式）：

```
######    0123456789012345678901234567890123456789012345678901234567890'
record = '.....100 .....513.25 ..........'
cost = int(record[20:23]) * float(record[31:37])
```

与其那样写，为什么不想这样命名切片呢：

```
SHARES = slice(20, 23)
PRICE = slice(31, 37)
cost = int(record[SHARES]) * float(record[PRICE])
```

第二种版本中，你避免了大量无法理解的硬编码下标，使得你的代码更加清晰可读了。

讨论¶

一般来讲，代码中如果出现大量的硬编码下标值会使得可读性和可维护性大大降低。比如，如果你同过来看看一年前你写的代码，你会摸着脑袋想那时候自己到底想干嘛啊。 这里的解决方案是一个很简单的方法让你更加清晰的表达代码到底要做什么。

内置的 `slice()` 函数创建了一个切片对象，可以被用在任何切片允许使用的地方。比如：

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10,11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

如果你有一个切片对象a，你可以分别调用它的 `a.start`, `a.stop`, `a.step` 属性来获取更多的信息。比如：

```
>>> a = slice(5, 50, 2)
>>> a.start
5
>>> a.stop
50
>>> a.step
2
>>>
```

另外，你还能通过调用切片的 `indices(size)` 方法将它映射到一个确定大小的序列上，这个方法返回一个三元组 `(start, stop, step)`，所有值都会被合适的缩小以满足边界限制，从而使用的时候避免出现 `IndexError` 异常。比如：

```
>>> s = 'HelloWorld'
>>> a.indices(len(s))
(5, 10, 2)
>>> for i in range(*a.indices(len(s))):
...     print(s[i])
...
W
r
d
>>>
```

1.12 序列中出现次数最多的元素¶

问题¶

怎样找出一个序列中出现次数最多的元素呢？

解决方案¶

`collections.Counter` 类就是专门为这类问题而设计的，它甚至有一个有用的 `most_common()` 方法直接给了你答案。

为了演示，先假设你有一个单词列表并且想找出哪个单词出现频率最高。你可以这样做：

```
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)
# 出现频率最高的3个单词
top_three = word_counts.most_common(3)
print(top_three)
# Outputs [('eyes', 8), ('the', 5), ('look', 4)]
```

讨论¶

作为输入，`Counter` 对象可以接受任意的由可哈希（`hashable`）元素构成的序列对象。在底层实现上，一个 `Counter` 对象就是一个字典，将元素映射到它出现的次数上。比如：

```
>>> word_counts['not']
1
>>> word_counts['eyes']
8
>>>
```

如果你想手动增加计数，可以简单的用加法：

```
>>> morewords = ['why','are','you','not','looking','in','my','eyes']
>>> for word in morewords:
...     word_counts[word] += 1
...
>>> word_counts['eyes']
9
>>>
```

或者你可以使用 `update()` 方法：

```
>>> word_counts.update(morewords)
>>>
```

`Counter` 实例一个鲜为人知的特性是它们可以很容易的跟数学运算操作相结合。比如：

```
>>> a = Counter(words)
>>> b = Counter(morewords)
>>> a
Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2,
 "you're": 1, "don't": 1, 'under': 1, 'not': 1})
>>> b
Counter({'eyes': 1, 'looking': 1, 'are': 1, 'in': 1, 'not': 1, 'you': 1,
'my': 1, 'why': 1})
>>> # Combine counts
>>> c = a + b
>>> c
Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'not': 2,
 "you're": 2, 'you': 1, "don't": 1, 'in': 1, 'why': 1,
 'looking': 1, 'are': 1, 'under': 1, 'around': 2})
>>> # Subtract counts
>>> d = a - b
>>> d
Counter({'eyes': 7, 'the': 5, 'look': 4, 'into': 3, 'my': 2, 'around': 2,
"you're": 1, "don't": 1, 'under': 1})
>>>
```

毫无疑问，`Counter` 对象在几乎所有需要制表或者计数数据的场合是非常有用的工具。在解决这类问题的时候你应该优先选择它，而不是手动的利用字典去实现。

1.13 通过某个关键字排序一个字典列表¶

问题¶

你有一个字典列表，你想根据某个或某几个字典字段来排序这个列表。

解决方案¶

通过使用 `operator` 模块的 `itemgetter` 函数，可以非常容易的排序这样的数据结构。假设你从数据库中检索出来网站会员信息列表，并且以下列的数据结构返回：

```
rows = [
    {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
    {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
    {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
    {'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
]
```

根据任意的字典字段来排序输入结果行是很容易实现的，代码示例：

```
from operator import itemgetter
rows_by_fname = sorted(rows, key=itemgetter('fname'))
rows_by_uid = sorted(rows, key=itemgetter('uid'))
print(rows_by_fname)
print(rows_by_uid)
```

代码的输出如下：

```
[{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'}]
[{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'}]
```

`itemgetter()` 函数也支持多个 keys，比如下面的代码

```
rows_by_lfname = sorted(rows, key=itemgetter('lname','fname'))
print(rows_by_lfname)
```

会产生如下的输出：

```
[{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'}]
```

讨论¶

在上面例子中，`rows` 被传递给接受一个关键字参数的 `sorted()` 内置函数。这个参数是 `callable` 类型，并且从 `rows` 中接受一个单一元素，然后返回被用来排序的值。 `itemgetter()` 函数就是负责创建这个 `callable` 对象的。

`operator.itemgetter()` 函数有一个被 `rows` 中的记录用来查找值的索引参数。可以是一个字典键名称，一个整形值或者任何能够传入一个对象的 `__getitem__()` 方法的值。 如果你传入多个索引参数给 `itemgetter()`，它生成的 `callable` 对象会返回一个包含所有元素值的元组， 并且 `sorted()` 函数会根据这个元组中元素顺序去排序。 但你想要同时在几个字段上面进行排序（比如通过姓和名来排序，也就是例子中的那样）的时候这种方法是很有用的。

`itemgetter()` 有时候也可以用 `lambda` 表达式代替，比如：

```
rows_by_fname = sorted(rows, key=lambda r: r['fname'])
rows_by_lfname = sorted(rows, key=lambda r: (r['lname'],r['fname']))
```

这种方案也不错。但是，使用 `itemgetter()` 方式会运行的稍微快点。因此，如果你对性能要求比较高的话就使用 `itemgetter()` 方式。

最后，不要忘了这节中展示的技术也同样适用于 `min()` 和 `max()` 等函数。比如：

```
>>> min(rows, key=itemgetter('uid'))
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}
>>> max(rows, key=itemgetter('uid'))
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
>>>
```

1.14 排序不支持原生比较的对象¶

问题¶

你想排序类型相同的对象，但是他们不支持原生的比较操作。

解决方案¶

内置的 `sorted()` 函数有一个关键字参数 `key` ，可以传入一个 `callable` 对象给它， 这个 `callable` 对象对每个传入的对象返回一个值，这个值会被 `sorted` 用来排序这些对象。 比如，如果你在应用程序里面有一个 `User` 实例序列，并且你希望通过他们的 `user_id` 属性进行排序， 你可以提供一个以 `User` 实例作为输入并输出对应 `user_id` 值的 `callable` 对象。比如：

```
class User:
    def __init__(self, user_id):
        self.user_id = user_id

    def __repr__(self):
        return 'User({})'.format(self.user_id)


def sort_notcompare():
    users = [User(23), User(3), User(99)]
    print(users)
    print(sorted(users, key=lambda u: u.user_id))
```

另外一种方式是使用 `operator.attrgetter()` 来代替 lambda 函数：

```
>>> from operator import attrgetter
>>> sorted(users, key=attrgetter('user_id'))
[User(3), User(23), User(99)]
>>>
```

讨论¶

选择使用 lambda 函数或者是 `attrgetter()` 可能取决于个人喜好。 但是， `attrgetter()` 函数通常会运行的快点，并且还能同时允许多个字段进行比较。 这个跟 `operator.itemgetter()` 函数作用于字典类型很类似（参考1.13小节）。 例如，如果 `User` 实例还有一个 `first_name` 和 `last_name` 属性，那么可以向下面这样排序：

```
by_name = sorted(users, key=attrgetter('last_name', 'first_name'))
```

同样需要注意的是，这一小节用到的技术同样适用于像 `min()` 和 `max()` 之类的函数。比如：

```
>>> min(users, key=attrgetter('user_id'))
User(3)
>>> max(users, key=attrgetter('user_id'))
User(99)
>>>
```

1.15 通过某个字段将记录分组¶

问题¶

你有一个字典或者实例的序列，然后你想根据某个特定的字段比如 `date` 来分组迭代访问。

解决方案¶

`itertools.groupby()` 函数对于这样的数据分组操作非常实用。 为了演示，假设你已经有了下列的字典列表：

```
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'},
]
```

现在假设你想在按 `date` 分组后的数据块上进行迭代。为了这样做，你首先需要按照指定的字段(这里就是 `date`)排序， 然后调用 `itertools.groupby()` 函数：

```
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)
```

运行结果：

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

讨论¶

`groupby()` 函数扫描整个序列并且查找连续相同值（或者根据指定 `key` 函数返回值相同）的元素序列。 在每次迭代的时候，它会返回一个值和一个迭代器对象， 这个迭代器对象可以生成元素值全部等于上面那个值的组中所有对象。

一个非常重要的准备步骤是要根据指定的字段将数据排序。 因为 `groupby()` 仅仅检查连续的元素，如果事先并没有排序完成的话，分组函数将得不到想要的结果。

如果你仅仅只是想根据 `date` 字段将数据分组到一个大的数据结构中去，并且允许随机访问， 那么你最好使用 `defaultdict()` 来构建一个多值字典，关于多值字典已经在 1.6 小节有过详细的介绍。比如：

```
from collections import defaultdict
rows_by_date = defaultdict(list)
for row in rows:
    rows_by_date[row['date']].append(row)
```

这样的话你可以很轻松的就能对每个指定日期访问对应的记录：

```
>>> for r in rows_by_date['07/01/2012']:
... print(r)
...
{'date': '07/01/2012', 'address': '5412 N CLARK'}
{'date': '07/01/2012', 'address': '4801 N BROADWAY'}
>>>
```

在上面这个例子中，我们没有必要先将记录排序。因此，如果对内存占用不是很关心， 这种方式会比先排序然后再通过 `groupby()` 函数迭代的方式运行得快一些。

1.16 过滤序列元素¶

问题¶

你有一个数据序列，想利用一些规则从中提取出需要的值或者是缩短序列

解决方案¶

最简单的过滤序列元素的方法就是使用列表推导。比如：

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> [n for n in mylist if n > 0]
[1, 4, 10, 2, 3]
>>> [n for n in mylist if n < 0]
[-5, -7, -1]
>>>
```

使用列表推导的一个潜在缺陷就是如果输入非常大的时候会产生一个非常大的结果集，占用大量内存。 如果你对内存比较敏感，那么你可以使用生成器表达式迭代产生过滤的元素。比如：

```
>>> pos = (n for n in mylist if n > 0)
>>> pos
<generator object <genexpr> at 0x1006a0eb0>
>>> for x in pos:
... print(x)
...
1
4
10
2
3
>>>
```

有时候，过滤规则比较复杂，不能简单的在列表推导或者生成器表达式中表达出来。比如，假设过滤的时候需要处理一些异常或者其他复杂情况。这时候你可以将过滤代码放到一个函数中， 然后使用内建的 `filter()` 函数。示例如下：

```
values = ['1', '2', '-3', '-', '4', 'N/A', '5']
def is_int(val):
    try:
        x = int(val)
        return True
    except ValueError:
        return False
ivals = list(filter(is_int, values))
print(ivals)
# Outputs ['1', '2', '-3', '4', '5']
```

`filter()` 函数创建了一个迭代器，因此如果你想得到一个列表的话，就得像示例那样使用 `list()` 去转换。

讨论¶

列表推导和生成器表达式通常情况下是过滤数据最简单的方式。 其实它们还能在过滤的时候转换数据。比如：

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> import math
>>> [math.sqrt(n) for n in mylist if n > 0]
[1.0, 2.0, 3.1622776601683795, 1.4142135623730951, 1.7320508075688772]
>>>
```

过滤操作的一个变种就是将不符合条件的值用新的值代替，而不是丢弃它们。比如，在一列数据中你可能不仅想找到正数，而且还想将不是正数的数替换成指定的数。 通过将过滤条件放到条件表达式中去，可以很容易的解决这个问题，就像这样：

```
>>> clip_neg = [n if n > 0 else 0 for n in mylist]
>>> clip_neg
[1, 4, 0, 10, 0, 2, 3, 0]
>>> clip_pos = [n if n < 0 else 0 for n in mylist]
>>> clip_pos
[0, 0, -5, 0, -7, 0, 0, -1]
>>>
```

另外一个值得关注的过滤工具就是 `itertools.compress()` 。 它以一个 `iterable` 对象和一个相对应的 `Boolean` 选择器序列作为输入参数， 然后输出 `iterable` 对象中对应选择器为 `True` 的元素。 当你需要用另外一个相关联的序列来过滤某个序列的时候，这个函数是非常有用的， 比如，假如现在你有下面两列数据：

```
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]
```

现在你想将那些对应 `count` 值大于5的地址全部输出，那么你可以这样做：

```
>>> from itertools import compress
>>> more5 = [n > 5 for n in counts]
>>> more5
[False, False, True, False, False, True, True, False]
>>> list(compress(addresses, more5))
['5800 E 58TH', '1060 W ADDISON', '4801 N BROADWAY']
>>>
```

这里的关键点在于先创建一个 `Boolean` 序列，指示哪些元素符合条件。 然后 `compress()` 函数根据这个序列去选择输出对应位置为 `True` 的元素。

和 `filter()` 函数类似， `compress()` 也是返回的一个迭代器。因此，如果你需要得到一个列表， 那么你需要使用 `list()` 来将结果转换为列表类型。

1.17 从字典中提取子集¶

问题¶

你想构造一个字典，它是另外一个字典的子集。

解决方案¶

最简单的方式是使用字典推导。比如：

```
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
# Make a dictionary of all prices over 200
p1 = {key: value for key, value in prices.items() if value > 200}
# Make a dictionary of tech stocks
tech_names = {'AAPL', 'IBM', 'HPQ', 'MSFT'}
p2 = {key: value for key, value in prices.items() if key in tech_names}
```

讨论¶

大多数情况下字典推导能做到的，通过创建一个元组序列然后把它传给 `dict()` 函数也能实现。比如：

```
p1 = dict((key, value) for key, value in prices.items() if value > 200)
```

但是，字典推导方式表意更清晰，并且实际上也会运行的更快些 （在这个例子中，实际测试几乎比 `dcit()` 函数方式快整整一倍）。

有时候完成同一件事会有多种方式，比如，第二个例子程序也可以像这样重写：

```
# Make a dictionary of tech stocks
tech_names = {  'AAPL',  'IBM',  'HPQ',  'MSFT' }
p2 = { key:prices[key] for key in prices.keys() & tech_names }
```

但是，运行时间测试结果显示这种方案大概比第一种方案慢 1.6 倍， 如果对程序运行性能要求比较高的话，需要花点时间去做计时测试。关于更多计时和性能测试，可以参考 14.13 小节。

1.18 映射名称到序列元素¶

问题¶

你有一段通过下标访问列表或者元组中元素的代码，但是这样有时候会使得你的代码难以阅读，于是你想通过名称来访问元素。

解决方案¶

`collections.namedtuple()` 函数通过使用一个普通的元组对象来帮你解决这个问题。 这个函数实际上是一个返回 Python 中标准元组类型子类的一个工厂方法。 你需要传递一个类型名和你需要的字段给它，然后它就会返回一个类。你可以初始化这个类，为你定义的字段传递值等。 代码示例：

```
>>> 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')
>>> sub.addr
'jonesy@example.com'
>>> sub.joined
'2012-10-19'
>>>
```

尽管 `namedtuple` 的实例看起来像一个普通的类实例，但是它跟元组类型是可交换的，支持所有的普通元组操作，比如索引和解压。 比如：

```
>>> len(sub)
2
>>> addr, joined = sub
>>> addr
'jonesy@example.com'
>>> joined
'2012-10-19'
>>>
```

命名元组的一个主要用途是将你的代码从下标操作中解脱出来。 因此，如果你从数据库调用中返回了一个很大的元组列表，通过下标去操作其中的元素， 当你在表中添加了新的列的时候你的代码可能就会出错了。但是如果你使用了命名元组，那么就不会有这样的顾虑。

为了说明清楚，下面是使用普通元组的代码：

```
def compute_cost(records):
    total = 0.0
    for rec in records:
        total += rec[1] * rec[2]
    return total
```

下标操作通常会让代码表意不清晰，并且非常依赖记录的结构， 下面是使用命名元组的版本：

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price'])
def compute_cost(records):
    total = 0.0
    for rec in records:
        s = Stock(*rec)
        total += s.shares * s.price
    return total
```

讨论¶

命名元组另一个用途就是作为字典的替代，因为字典存储需要更多的内存空间， 如果你需要构建一个非常大的包含字典的数据结构，那么使用命名元组会更加高效， 但是需要注意的是，不像字典那样，一个命名元组是不可更改的，比如：

```
>>> s = Stock('ACME', 100, 123.45)
>>> s
Stock(name='ACME', shares=100, price=123.45)
>>> s.shares = 75
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

如果你真的需要改变属性的值，那么可以使用命名元组实例的 `_replace()` 方法，它会创建一个全新的命名元组并将对应的字段用新的值取代。比如：

```
>>> s = s._replace(shares=75)
>>> s
Stock(name='ACME', shares=75, price=123.45)
>>>
```

`_replace()` 方法还有一个很有用的特性就是当你的命名元组拥有可选或者缺失字段时候， 它是一个非常方便的填充数据的方法。 你可以先创建一个包含缺省值的原型元组，然后使用 `_replace()` 方法创建新的值被更新过的实例。比如：

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price', 'date', 'time'])

# Create a prototype instance
stock_prototype = Stock('', 0, 0.0, None, None)

# Function to convert a dictionary to a Stock
def dict_to_stock(s):
    return stock_prototype._replace(**s)
```

下面是它的使用方法：

```
>>> a = {'name': 'ACME', 'shares': 100, 'price': 123.45}
>>> dict_to_stock(a)
Stock(name='ACME', shares=100, price=123.45, date=None, time=None)
>>> b = {'name': 'ACME', 'shares': 100, 'price': 123.45, 'date': '12/17/2012'}
>>> dict_to_stock(b)
Stock(name='ACME', shares=100, price=123.45, date='12/17/2012', time=None)
>>>
```

最后要说的是，如果你的目标是定义一个需要更新很多实例属性的高效数据结构，那么命名元组并不是你的最佳选择， 这时候你应该考虑定义一个包含 `__slots__` 方法的类（参考8.4小节）。

1.19 转换并同时计算数据¶

问题¶

你需要在数据序列上执行聚集函数（比如 `sum()`, `min()`, `max()`）， 但是首先你需要先转换或者过滤数据

解决方案¶

一个非常优雅的方式去结合数据计算与转换就是使用一个生成器表达式参数。 比如，如果你想计算平方和，可以像下面这样做：

```
nums = [1, 2, 3, 4, 5]
s = sum(x * x for x in nums)
```

下面是更多的例子：

```
# Determine if any .py files exist in a directory
import os
files = os.listdir('dirname')
if any(name.endswith('.py') for name in files):
    print('There be python!')
else:
    print('Sorry, no python.')
# Output a tuple as CSV
s = ('ACME', 50, 123.45)
print(','.join(str(x) for x in s))
# Data reduction across fields of a data structure
portfolio = [
    {'name':'GOOG', 'shares': 50},
    {'name':'YHOO', 'shares': 75},
    {'name':'AOL', 'shares': 20},
    {'name':'SCOX', 'shares': 65}
]
min_shares = min(s['shares'] for s in portfolio)
```

讨论¶

上面的示例向你演示了当生成器表达式作为一个单独参数传递给函数时候的巧妙语法（你并不需要多加一个括号）。 比如，下面这些语句是等效的：

```
s = sum((x * x for x in nums)) # 显示的传递一个生成器表达式对象
s = sum(x * x for x in nums) # 更加优美的实现方式，省略了括号
```

使用一个生成器表达式作为参数会比先创建一个临时列表更加高效和优雅。 比如，如果你不使用生成器表达式的话，你可能会考虑使用下面的实现方式：

```
nums = [1, 2, 3, 4, 5]
s = sum([x * x for x in nums])
```

这种方式同样可以达到想要的效果，但是它会多一个步骤，先创建一个额外的列表， 对于小型列表可能没什么关系，但是如果元素数量非常大的时候， 它会创建一个巨大的仅仅被使用一次就被丢弃的临时数据结构，而生成器方案会以迭代的方式转换数据，因此更省内存。

在使用一些聚集函数比如 `min()` 和 `max()` 的时候你可能更加倾向于使用生成器版本， 它们接受的一个 `key` 关键字参数或许对你很有帮助，比如，在上面的证券例子中，你可能会考虑下面的实现版本：

```
# Original: Returns 20
min_shares = min(s['shares'] for s in portfolio)
# Alternative: Returns {'name': 'AOL', 'shares': 20}
min_shares = min(portfolio, key=lambda s: s['shares'])
```

1.20 合并多个字典或映射¶

问题¶

现在有多个字典或者映射，你想将它们从逻辑上合并为一个单一的映射后执行某些操作，比如查找值或者检查某些键是否存在。

解决方案¶

假如你有如下两个字典：

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

现在假设你必须在两个字典中执行查找操作（比如先从 `a` 中找，如果找不到再在 `b` 中找）。 一个非常简单的解决方案就是使用 `collections` 模块中的 `ChainMap` 类。比如：

```
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)
```

讨论¶

一个 `ChainMap` 接受多个字典并将它们在逻辑上变为一个字典。 然后，这些字典并不是真的合并在一起了， `ChainMap` 类只是在内部创建了一个容纳这些字典的列表 并重新定义了一些常见的字典操作来遍历这个列表。大部分字典操作都是可以正常使用的，比如：

```
>>> len(c)
3
>>> list(c.keys())
['x', 'y', 'z']
>>> list(c.values())
[1, 2, 3]
>>>
```

如果出现重复键，那么第一次出现的映射值会被返回。 因此，例子程序中的 `c['z']` 总是会返回字典 `a` 中对应的值，而不是 `b` 中对应的值。

对于字典的更新或删除操作总是影响的是列表中第一个字典。比如：

```
>>> c['x'] = 10
>>> c['w'] = 40
>>> del c['x']
>>> a
{'w': 40, 'z': 10}
>>> del c['y']
Traceback (most recent call last):
...
KeyError: "Key not found in the first mapping: 'y'"
>>>
```

ChainMap 对于编程语言中的作用范围变量（比如 globals , locals 等）是非常有用的。事实上，有一些方法可以使它变得简单：

```
>>> 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
>>> values
ChainMap({'x': 1})
>>>
```

作为 ChainMap 的替代，你可能会考虑使用 update() 方法将两个字典合并。比如：

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = dict(b)
>>> merged.update(a)
>>> merged['x']
1
>>> merged['y']
2
>>> merged['z']
3
>>>
```

这样也能行得通。但是它需要你创建一个完全不同的字典对象（或者是破坏现有字典结构）。同时，如果原字典做了更新，这种改变不会反应到新的合并字典中去。比如：

```
>>> a['x'] = 13
>>> merged['x']
1
```

ChainMap 使用原来的字典，它自己不创建新的字典。所以它并不会产生上面所说的结果。比如：

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = ChainMap(a, b)
>>> merged['x']
1
>>> a['x'] = 42
>>> merged['x'] # Notice change to merged dicts
42
>>>
```