Chapter 1

数据结构和算法

1.1 将序列分解为单独的变量

任何序列（或可迭代的对象）都可以通过一个简单的赋值操作来分解为单独的变量。唯一的要求是变量的总数和结构要与序列相吻合。只要对象恰好是可迭代的，那么就可以执行分解操作。这包括字符串、文件、迭代器以及生成器。当做分解操作时，有时候可能想丢弃某些特定的值。Python 并没有提供特殊的语法来实现这一点，但是通常可以选一个用不到的变量名，以此来作为要丢弃的值的名称。

1.2 从任意长度的可迭代对象中分解元素

Python 的“*表达式”可以用来解决这个问题。由*修饰的变量也可以位于列表的第一个位置。这类可迭代对象中会有一些已知的组件或模式（例如，元素 1 之后的所有内容都是电话号码），利用*表达式分解可迭代对象使得开发者能够轻松利用这些模式，而不必在可迭代对象中做复杂花哨的操作才能得到相关的元素。

当和某些特定的字符串处理操作相结合，比如做拆分（splitting）操作时，这种*式的语法所支持的分解操作也非常有用。

1.3 保存最后 N 个元素

保存有限的历史记录可算是 collections.deque 的完美应用场景了。

更普遍的是，当需要一个简单的队列结构时，deque 可祝你一臂之力。如果不指定队列的大小，也就得到了一个无界限的队列，可以在两端执行添加和弹出操作。

从队列两端添加或弹出元素的复杂度都是 $O(1)$。这和列表不同，当从列表的头部插入或移除元素时，列表的复杂度为 $O(N)$。

1.4 找到最大或最小的 N 个元素

heapq 模块中有两个函数—nlargest()和 nsmallest()。堆最重要的特性就是 heap[0]总是最小那个的元素。此外，接下来的元素可依次通过 heapq.heappop()方法轻松找到。该方法会将第一个元素（最小的）弹出，然后以第二小的元素取而代之（这个操作的复杂度是 $O(logN)$，N 代表堆的大小）。

当所要找的元素数量相对较小时，函数 nlargest()和 nsmallest()才是最适用的。如果只是简单地想找到最小或最大的元素（$N = 1$ 时），那么用 min()和 max()会更加快。同样，如果 N 和集合本身的大小差不多大，通常更快的方法是先对集合排序，然后做切片操作（例如，使用 sorted(items)[:N]或者 sorted(items)[-N:]）。应该要注意的是，nlargest()和 nsmallest()的实际实现会根据使用它们的方式而有所不同，可能会相应作出一些优化措施（比如，当 N 的大小同输入大小很接近时，就会采用排序的方法）。

1.5 实现优先级队列

1.6 在字典中将键映射到多个值上

为了能方便地创建这样的字典，可以利用 collections 模块中的 defaultdict 类。defaultdict的一个特点就是它会自动初始化第一个值，这样只需关注添加元素即可。

关于 defaultdict，需要注意的一个地方是，它会自动创建字典表项以待稍后的访问（即使这些表项当前在字典中还没有找到）。如果不想要这个功能，可以在普通的字典上调用 setdefault()方法来取代。

1.7 让字典保持有序

要控制字典中元素的顺序，可以使用 collections 模块中的 OrderedDict 类。当对字典做迭代时，它会严格按照元素初始添加的顺序进行。当想构建一个映射结构以便稍后对其做序列化或编码成另一种格式时，OrderedDict 就显得特别有用。请注意 OrderedDict 的大小是普通字典的 2 倍多，这是由于它额外创建的链表所致。(**读者注：Python 后续的版本字典默认是有序的，这个保留下来是为了兼容性问题**)

1.8 与字典有关的计算问题

如果尝试在字典上执行常见的数据操作，将会发现它们只会处理键，而不是值。

1.9 在两个字典中寻找相同点

要找出两个字典中的相同之处，只需通过 keys()或者 items()方法执行常见的集合操作即可。这些类型的操作也可用来修改或过滤掉字典中的内容。例如，假设想创建一个新的字典，其中会去掉某些键。

字典就是一系列键和值之间的映射集合。字典的 keys()方法会返回 keys-view 对象，其中暴露了所有的键。关于字典的键有一个很少有人知道的特性，那就是它们也支持常见的集合操作，比如求并集、交集和差集。因此，如果需要对字典的键做常见的集合操作，那么就能直接使用 keys-view 对象而不必先将它们转化为集合。

字典的 items()方法返回由(key,value)对组成的 items-view 对象。这个对象支持类似的集合操作，可用来完成找出两个字典间有哪些键值对有相同之处的操作。

尽管类似，但字典的 values()方法并不支持集合操作。部分原因是因为在字典中键和值是不同的，从值的角度来看并不能保证所有的值都是唯一的。单这一条原因就使得某些特定的集合操作是有问题的。但是，如果必须执行这样的操作，还是可以先将值转化为集合来实现。

1.10 从序列中移除重复项且保持元素间顺序不变

1.11 对切片命名

一般来说，内置的 slice()函数会创建一个切片对象，可以用在任何允许进行切片操作的地方。

如果有一个 slice 对象的实例 s，可以分别通过 s.start、s.stop 以及 s.step 属性来得到关于该对象的信息。

1.12 找出序列中出现次数最多的元素

collections 模块中的 Counter 类正是为此类问题所设计的。它甚至有一个非常方便的`most_common()`方法可以直接告诉我们答案。

可以给 Counter 对象提供任何可哈希的对象序列作为输入。在底层实现中，Counter 是一个字典，在元素和它们出现的次数间做了映射。

如果想手动增加计数，只需简单地自增即可，另一种方式是使用 update()方法。关于 Counter 对象有一个不为人知的特性，那就是它们可以轻松地同各种数学运算操作结合起来使用。

1.13 通过公共键对字典列表排序

利用 operator 模块中的 itemgetter 函数对这类结构进行排序是非常简单的。itemgetter()函数还可以接受多个键。

内建的 sorted()函数，该函数接受一个关键字参数 key。这个参数应该代表一个可调用对象（callable），该对象从 rows 中接受一个单独的元素作为输入并返回一个用来做排序依据的值。itemgetter()函数创建的就是这样一个可调用对象。

有时候会用 lambda 表达式来取代 itemgetter()的功能。但是用 itemgetter()通常会运行得更快一些。因此如果需要考虑性能问题的话，应该使用 itemgetter()。

1.14 对不支持原生比较操作的对象排序

内建的 sorted()函数可接受一个用来传递可调用对象（callable）的参数 key，而该可调用对象会返回待排序对象中的某些值，sorted 则利用这些值来比较对象。另一种方式是使用 operator.attrgetter()。

要使用 lambda 表达式还是 attrgetter()或许只是一种个人喜好。但是通常来说，attrgetter()要更快一些，而且具有允许同时提取多个字段值的能力。这和针对字典的 operator.itemgetter()的使用很类似（参见 通过公共键对字典列表排序）。

1.15 根据字段将记录分组

itertools.groupby()函数在对数据进行分组时特别有用。

函数 groupby()通过扫描序列找出拥有相同值（或是由参数 key 指定的函数所返回的值）的序列项，并将它们分组。groupby()创建了一个迭代器，而在每次迭代时都会返回一个值（value）和一个子迭代器（sub_iterator），这个子迭代器可以产生所有在该分组内具有该值的项。groupby()只能检查连续的项，不首先排序的话，将无法按所想的方式来对记录分组。

如果只是简单地根据日期将数据分组到一起，放进一个大的数据结构中以允许进行随机访问，那么利用 defaultdict()构建一个一键多值字典（multidict，见 在字典中将键映射到多个值上）可能会更好。我们并不需要先对记录做排序。因此，如果不考虑内存方面的因素，这种方式会比先排序再用 groupby()迭代要来的更快。

1.16 筛选序列中的元素

要筛选序列中的数据，通常最简单的方法是使用列表推导式（list comprehension）。使用列表推导式的一个潜在缺点是如果原始输入非常大的话，这么做可能会产生一个庞大的结果。如果这是你需要考虑的问题，那么可以使用生成器表达式通过迭代的方式产生筛选的结果。

有时候筛选的标准没法简单地表示在列表推导式或生成器表达式中。于此，可以将处理筛选逻辑的代码放到单独的函数中，然后使用内建的 filter()函数处理。filter()创建了一个迭代器，因此如果我们想要的是列表形式的结果，请确保加上了 list()。

列表推导式和生成器表达式通常是用来筛选数据的最简单和最直接的方式。此外，它们也具有同时对数据做转换的能力。关于筛选数据，有一种情况是用新值替换掉不满足标准的值，而不是丢弃它们。

另一个值得一提的筛选工具是 itertools.compress()，它接受一个可迭代对象以及一个布尔选择器序列作为输入。输出时，它会给出所有在相应的布尔选择器中为 True 的可迭代对象元素。如果想把对一个序列的筛选结果施加到另一个相关的序列上时，这就会非常有用。关键在于首先创建一个布尔序列，用来表示哪个元素可满足我们的条件。然后compress()函数挑选出满足布尔值为 True 的相应元素。

同 filter()函数一样，正常情况下 compress()会返回一个迭代器。因此，如果需要的话，得使用 list()将结果转为列表。

1.17 从字典中提取子集

利用字典推导式（dictionary comprehension）可轻松解决。

1.18 将名称映射到序列的元素中

相比普通的元组，collections.namedtuple()（命名元组）只增加了极小的开销就提供了这些便利。尽管 namedtuple 的实例看起来就像一个普通的类实例，但它的实例与普通的元组是可互换的，而且支持所有普通元组所支持的操作，例如索引（indexing）和分解（unpacking）。

命名元组的主要作用在于将代码同它所控制的元素位置间解耦。所以，如果从数据库调用中得到一个大型的元组列表，而且通过元素的位置来访问数据，那么假如在表单中新增了一列数据，那么代码就会崩溃。但如果首先将返回的元组转型为命名元组，就不会出现问题。

如果需要修改任何属性，可以通过使用 namedtuple 实例的`_replace()`方法来实现。该方法会创建一个全新的命名元组，并对相应的值做替换。`_replace()`方法有一个微妙的用途，那就是它可以作为一种简便的方法填充具有可选或缺失字段的命名元组。要做到这点，首先创建一个包含默认值的“原型”元组，然后使用`_replace()`方法创建一个新的实例，把相应的值替换掉。

最后，也是相当重要的是，应该要注意如果我们的目标是定义一个高效的数据结构，而且将来会修改各种实例属性，那么使用 namedtuple 并不是最佳选择。相反，可以考虑定义一个使用`__slots__`属性的类。

1.19 同时对数据做转换和换算

有一种非常优雅的方式能将数据换算和转换结合在一起—在函数参数中使用生成器表达式。这种解决方案展示了当把生成器表达式作为函数的单独参数时在语法上的一些微妙之处（即，不必重复使用括号）。比起首先创建一个临时的列表，使用生成器做参数通常是更为高效和优雅的方式。

1.20 将多个映射合并为单个映射

我们有多个字典或映射，想在逻辑上将它们合并为一个单独的映射结构，以此执行某些特定的操作，比如查找值或检查键是否存在。一种简单的方法是利用 collections 模块中的 ChainMap 类来解决这个问题。

ChainMap 可接受多个映射然后在逻辑上使它们表现为一个单独的映射结构。但是，这些映射在字面上并不会合并在一起。相反，ChainMap 只是简单地维护一个记录底层映射关系的列表，然后重定义常见的字典操作来扫描这个列表。如果有重复的键，那么这里会采用第一个映射中所对应的值。

修改映射的操作总是会作用在列出的第一个映射结构上。

作为 ChainMap 的替代方案，我们可能会考虑利用字典的 update()方法将多个字典合并在一起。这么做行得通，但这需要单独构建一个完整的字典对象（或者修改其中现有的一个字典，这就破坏了原始数据）。此外，如果其中任何一个原始字典做了修改，这个改变都不会反应到合并后的字典中。而 ChainMap 使用的就是原始的字典，因此它不会产生这种令人不悦的行为。

Chapter 2

迭代器和生成器

如果一个函数定义中包含 yield 表达式，那么这个函数就不再是一个普通函数，而是一个生成器函数。yield 语句类似 return 会返回一个值但它会记住这个返回的位置，下次 next() 迭代就从这个位置下一行继续执行。生成器函数并不是生成器，它运行返回后的结果才是生成器。

2.1 手动访问迭代器中的元素

我们需要处理某个可迭代对象中的元素，但是基于某种原因不能也不想使用 for 循环。要手动访问可迭代对象中的元素，可以使用 next()函数，然后自己编写代码来捕获 StopIteration 异常。一般来说，StopIteration 异常是用来通知我们迭代结束的。但是，如果是手动使用 next()，也可以命令它返回一个结束值，比如说 None。

2.2 委托迭代

Python 的迭代协议要求`__iter__()`返回一个特殊的迭代器对象，由该对象实现的`__next__()`方法来完成实际的迭代。如果要做的只是迭代另一个容器中的内容，我们不必担心底层细节是如何工作的，所要做的就是转发迭代请求。

2.3 用生成器创建新的迭代模式

如果想实现一种新的迭代模式，可使用生成器函数来定义。函数中只要出现了 yield 语句就会将其转变成一个生成器。与普通函数不同，生成器只会在响应迭代操作时才运行。

2.4 实现迭代协议

Python 的迭代协议要求`__iter__()`返回一个特殊的迭代器对象，该对象必须实现`__next__()`方法，并使用 StopIteration 异常来通知迭代的完成。

2.5 反向迭代

可以使用内建的 reversed()函数实现反向迭代。反向迭代只有在待处理的对象拥有可确定的大小，或者对象实现了`__reversed__()`特殊方法时，才能奏效。如果这两个条件都无法满足，则必须首先将这个对象转换为列表。许多程序员都没有意识到如果他们实现了`__reversed__()`方法，那么

就可以在自定义的类上实现反向迭代。定义一个反向迭代器可使代码变得更加高效，因为这样就无需先把数据放到列表中，然后再反向去迭代列表了。

2.6 定义带有额外状态的生成器函数

如果想让生成器将状态暴露给用户，别忘了可以轻易地将其实现为一个类，然后把生成器函数的代码放到`__iter__()`方法中即可。

2.7 对迭代器做切片操作

我们想对由迭代器产生的数据做切片处理，但是普通的切片操作符在这里不管用。要对迭代器和生成器做切片操作，itertools.islice()函数是完美的选择。

2.8

2.9

2.10

2.11

2.12

2.13

2.14

2.15

2.16