# 数据结构

## 解压序列

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。

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

## \*—解压可迭代对象

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。

*def* drop\_first\_last(*grades*):  
 first, \*middle, last = *grades  
 return* avg(middle)

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

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

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

递归算法

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

## deque—保留最后N 个元素

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。

*from* collections *import* deque  
*def* search(*lines*, *pattern*, *history*=5):  
 previous\_lines = deque(maxlen=*history*)  
 *for* li *in lines*:  
 *if pattern in* li:  
 *yield* li, previous\_lines  
 previous\_lines.append(li)  
*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)

## heapq—最大或最小的N个元素

*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]  
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'])

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

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。如果N 快接近集合大小了，那么使用排序操作会更好些

## heapq—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]  
*class* Item:  
 *def \_\_init\_\_*(self, *name*):  
 self.name = *name  
 def \_\_repr\_\_*(self):  
 *return* 'Item({!r})'.format(self.name)  
q = PriorityQueue()

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小优先级

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

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。

## defaultdict—字典中的键映射多个值

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

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

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

d.setdefault('a', []).append(1)

## 过滤元素

1. 列表推导

[n for n in mylist if n > 0]

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

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

1. 生成器表达式迭代

mylist = [1, 4, -5, 10, -7, 2, 3, -1]  
pos = (n *for* n *in* mylist *if* n > 0)  
*for* x *in* pos:  
 print(x)

1. filter

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)

filter() 函数创建了一个迭代器

1. 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

compress() 也是返回的一个迭代器。

*from* itertools *import* compress  
addresses = [  
'5412 N CLARK',  
'5148 N CLARK',  
'5800 E 58TH',  
'2122 N CLARK',  
'5645 N RAVENSWOOD',  
'1060 W ADDISON',  
'4801 N BROADWAY',]  
counts = [ 0, 3, 10, 4, 1, 7, 6]  
more5 = [n > 5 *for* n *in* counts]  
print(more5)  
results=list(compress(addresses, more5))  
print(results)

## OrderedDict—控制元素的顺序

*from* collections *import* OrderedDict  
*def* ordered\_dict():  
 d = OrderedDict()  
 d['foo'] = 1  
 d['bar'] = 2  
 *for* key *in* d:  
 print(key, d[key])

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。如果构建一个需要大量OrderedDict 实例的数据结构的时候，消耗大量内存

## zip—键和值反转

zip() 函数创建的是一个只能访问一次的迭代器

zip(prices.values(), prices.keys())

## min() 和max()

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

## Counter—元素计数

*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)

## itemgetter—关键字排序

rows = [   
{'fname': 'David', 'lname': 'Beazley', 'uid': 1002},  
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}]  
*from* operator *import* itemgetter  
rows\_by\_fname = sorted(rows, key=itemgetter('fname'))  
rows\_by\_uid = sorted(rows, key=itemgetter('uid','fname'))  
print(rows\_by\_fname)  
print(rows\_by\_uid)

## groupby—分组

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'}]  
*from* operator *import* itemgetter  
*from* itertools *import* groupby  
# Sort by the desired field first  
rows.sort(key=itemgetter('date'))  
print(rows)  
# Iterate in groups  
*for* date, items *in* groupby(rows, key=itemgetter('date')):  
 print(date)  
 *for* i *in* items:  
 print(' ', i)

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，最好使用defaultdict() 来构建一个多值字典

## 字典推导式

prices = {  
'ACME': 45.23,  
'AAPL': 612.78,  
'IBM': 205.55,  
'HPQ': 37.20,  
'FB': 10.75  
}  
p1 = {key: value *for* key, value *in* prices.items() *if* value > 200}  
tech\_names = {'AAPL', 'IBM', 'HPQ', 'MSFT'}  
p2 = {key: value *for* key, value *in* prices.items() *if* key *in* tech\_names}

## namedtuple—映射名称到序列元素

命名元组的一个主要用途是将代码从下标操作中解脱出来

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

*from* collections *import* namedtuple  
Subscriber = namedtuple('Subscriber', ['addr', 'joined'])  
sub = Subscriber('jonesy@example.com', '2012-10-19')  
addr, joined = sub  
print(addr,joined)

命名元组实例的\_replace()方法，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。

s = s.\_replace(shares=75)

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

如果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最佳选择。应该考虑定义一个包含\_slots\_ 方法的类

## 生成器表达式—转换并计算

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)  
print(min\_shares)

ChainMap—合并多个

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

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

如果出现重复键，那么第一次出现的映射值会被返回。

# 字符串和文本

## startswith—开头匹配

filename.startswith('.txt')

## endswith—结尾匹配

filename.endswith('.txt')

如果你想检查多种匹配可能，只需要将所有的匹配项放入到一个元组中去，然后传给startswith() 或者endswith() 方法

[name for name in filenames if name.endswith(('.c', '.h')) ]

if name.startswith(('http:', 'https:', 'ftp:')):

return urlopen(name).read()

这个方法中必须要输入一个元组作为参数。如果你恰巧有一个list 或者set 类型的选择项，要确保传递参数前先调用tuple() 将其转换为元组类型

## fnmatch、fnmatchcase—Shell 通配符匹配

*from* fnmatch *import* fnmatch, fnmatchcase  
fnmatch('foo.txt', '\*.txt')  
  
names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'foo.py']  
lists=[name *for* name *in* names *if* fnmatch(name, 'Dat\*.csv')]  
print(lists)

fnmatch() 函数使用底层操作系统的大小写敏感规则来匹配模式

## 匹配和搜索

find()

findall() 方法会搜索文本并以列表形式返回所有的匹配

如果你想以迭代方式返回匹配，可以使用finditer() 方法来代替

## IGNORECASE—忽略大小写的搜索和替换

*import* re  
text = 'UPPER PYTHON, lower python, Mixed Python'  
a=re.findall('python', text, flags=re.IGNORECASE)  
print(a)  
b=re.sub('python', 'snake', text, flags=re.IGNORECASE)  
print(b)

替换字符串并不会自动跟被匹配字符串的大小写保持一致

## unicodedata—Unicode 文本标准化

s1 = 'Spicy Jalape\u00f1o'  
s2 = 'Spicy Jalapen\u0303o'  
print(s1,'\n',s2)  
*import* unicodedata  
t1 = unicodedata.normalize('NFC', s1)  
t2 = unicodedata.normalize('NFC', s2)  
print(ascii(t1),'\n',ascii(t2))  
t3 = unicodedata.normalize('NFD', s1)  
t4 = unicodedata.normalize('NFD', s2)  
print(ascii(t3),'\n',ascii(t4))

normalize() 第一个参数指定字符串标准化的方式

## strip—删除

strip() 方法能用于删除开始或结尾的字符。lstrip() 和rstrip() 分别从左和从右执行删除操作。默认情况下，这些方法会去除空白字符，但是你也可以指定其他字符。

去除操作不会对字符串的中间的文本产生任何影响如果想处理中间的空格，那么需要求助其他技术。比如使用replace() 方法或者是用正则表达式替换

s = 'hello world'  
b=s.strip('d')  
print(b)

## translate—审查清理

translate消除整个区间上的字符或者去除变音符

*import* sys,unicodedata  
a='pýtĥöñ'  
cmb\_chrs = dict.fromkeys(c *for* c *in* range(sys.maxunicode) *if* unicodedata.combining(chr(c)))  
b = unicodedata.normalize('NFD', a)  
b=b.translate(cmb\_chrs)  
print(b)

## 字符串对齐

对于基本的字符串对齐操作，可以使用字符串的ljust() , rjust() 和center()方法，所有这些方法都能接受一个可选的填充字符

text.rjust(20,'=')

format() 同样可以用来对齐字符串。使用<，> 或者ˆ 字符后面紧跟一个指定的宽度。

format(text, '>20') #右对齐

format(text, '<20') #左对齐

format(text, '^20') #居中

format(text, '=>20s') #右对齐，其他以===补充

当格式化多个值的时候，这些格式代码也可以被用在format() 方法中。

'{:>10s} {:>10s}'.format('Hello', 'World')

format() 函数的一个好处是它不仅适用于字符串。它可以用来格式化任何值，使

得它非常的通用

## 合并拼接

如果想要合并的字符串是在一个序列或者iterable 中，那么最快的方式就是使用join() 方法

parts = ['Is', 'Chicago', 'Not', 'Chicago?']  
' '.join(parts)

如果仅仅只是合并少数几个字符串，使用加号(+)

当使用加号(+)去连接大量的字符串的时候是非常低效率的，因为加号连接会引起内存复制以及垃圾回收操作

利用生成器表达式转换数据为字符串的同时合并字符串

data = ['ACME', 50, 91.1]  
','.join(str(d) *for* d *in* data)

同样还得注意不必要的字符串连接操作

a = 'Is Chicago'  
b = 'Not Chicago?'  
print(a + ':' + b ) # Ugly  
print(':'.join([a, b])) # Still ugly  
print(a, b, sep=':') # Better

## textwrap—指定列宽格式化

*import* textwrap,os  
print(os.get\_terminal\_size().columns) #自动匹配终大小  
s = "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."  
print(textwrap.fill(s, 40))  
print(textwrap.fill(s, 40,initial\_indent=' '))  
print(textwrap.fill(s, 40,subsequent\_indent=' '))

fill() 方法接受一些其他可选参数来控制tab， 语句结尾等。

## escape—处理html 和xml

将HTML 或者XML 实体如&entity; 或&#code; 替换为对应的文本，转换文本中特定的字符(比如*<*, *>*, 或&)。

替换文本字符串中的‘<’ 或者‘>’ ，使用html.escape() 函数

*import* html  
s = 'Elements are written as "<tag>text</tag>".'  
print(html.escape(s))  
print(html.escape(s, quote=*False*))

如果是ASCII 文本，想将非ASCII 文本对应的编码实体嵌入进去，可以给某些I/O 函数传递参数errors='xmlcharrefreplace'

*import* html

s = 'Spicy Jalapeño'  
s=s.encode('ascii', errors='xmlcharrefreplace')  
print(s)

替换文本中的编码实体，先使用一个合适的HTML 或者XML 解析器。通常情况下，这些工具会自动替换这些编码值

s = 'Spicy &quot;Jalape&#241;o&quot.'  
*from* html.parser *import* HTMLParser  
p = HTMLParser()  
p=p.unescape(s)  
print(p)  
  
*from* xml.sax.saxutils *import* unescape  
p= 'The prompt is &gt;&gt;&gt;'  
p=unescape(p)  
print(p)

## 字节字符串

字节字符串同样也支持大部分和文本字符串一样的内置操作

data = b'Hello World'

data.replace(b'Hello', b'Hello Cruel')

也有一些需要注意的不同点。

首先，字节字符串的索引操作返回整数而不是单独字符。

第二点，字节字符串不会提供一个美观的字符串表示，也不能很好的打印出来，除

非它们先被解码为一个文本字符串。也不存在任何适用于字节字符串的格式化操作

data.decode('ascii')

如果格式化字节字符串，先使用标准的文本字符串，然后将其编码为字节字符串

'{:10s} {:10d} {:10.2f}'.format('ACME', 100, 490.1).encode('ascii')

使用字节字符串可能会改变一些操作的语义，特别是那些跟文件系统有关的操作

# 数字日期和时间

## round—四舍五入

round(value, ndigits) 函数

当一个值刚好在两个边界的中间的时候， round 函数返回离它最近的偶数。也就是说，对1.5 或者2.5 的舍入运算都会得到2。

传给round() 函数的ndigits 参数可以是负数，舍入运算会作用在十位、百位、千位等上面

## decimal—精确的浮点数

浮点数的一个普遍问题是它们并不能精确的表示十进制数。并且，即使是最简单的数学运算也会产生小的误差

如果想更加精确(并能容忍一定的性能损耗)，可以使用decimal 模块，主要用在涉及到金融的领域。

*from* decimal *import* Decimal  
a = Decimal('4.2')  
b = Decimal('2.1')  
print(a+b)

decimal 模块的一个主要特征是允许你控制计算的每一方面，包括数字位数和四舍

五入运算。为了这样做，你先得创建一个本地上下文并更改它的设置

*from* decimal *import* Decimal,localcontext  
a = Decimal('4.3')  
b = Decimal('2.1')  
print(a / b)  
*with* localcontext() *as* ctx:  
 ctx.prec = 4  
 print(a / b)

## format—格式化输出

将数字格式化后输出，并控制数字的位数、对齐、千位分隔符和其他的细节。

x = 1234.56789  
a=format(x, '0.2f')  
b=format(x, '>10.1f')  
c=format(x, ',')  
d=format(x, 'e')  
f=format(x, '0.2E')

同时适用于浮点数和decimal模块中的Decimal 数字对象

当指定数字的位数后，结果值会根据round() 函数同样的规则进行四舍五入后返回

# 迭代器与生成器

## next—手动遍历

为了手动的遍历可迭代对象，使用next() 函数并在代码中捕获StopIteration 异常

# 手动读取一个文件中的所有行：  
*def* manual\_iter():  
 *with* open('/etc/passwd') *as* f:  
 *try*:  
 *while True*:  
 line = next(f)  
 print(line, end='')  
 *except* StopIteration:  
 *pass*

StopIteration 用来指示迭代的结尾。然而，如果你手动使用上面演示的next() 函数的话，你还可以通过返回一个指定值来标记结尾，比如None

line = next(f, None)

## \_\_iter\_\_—代理迭代

构建一个自定义容器对象，里面包含有列表、元组或其他可迭代对象。在这个新容器对象上执行迭代操作。定义一个\_\_iter\_\_ () 方法，将迭代操作代理到容器内部的对象上去。

*class* Node:  
 *def \_\_init\_\_*(self, *value*):  
 self.\_value = *value* self.\_children = []  
 *def \_\_repr\_\_*(self):  
 *return* 'Node({!r})'.format(self.\_value)  
 *def* add\_child(self, *node*):  
 self.\_children.append(*node*)  
 *def \_\_iter\_\_*(self):  
 *return* iter(self.\_children)  
 # Example  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 root = Node(0)  
 child1 = Node(1)  
 child2 = Node(2)  
 root.add\_child(child1)  
 root.add\_child(child2)  
 *for* ch *in* root:  
 print(ch)

## yield—生成器

一个函数中需要有一个yield 语句即可将其转换为一个生成器。跟普通函数不同的是，生成器只能用于迭代操作。

一个生成器函数主要特征是它只会回应在迭代中使用到的next 操作。一旦生成器函数返回退出，迭代终止

*def* frange(*start*, *stop*, *increment*):  
 x = *start  
 while* x < *stop*:  
 *yield* x  
 x += *increment  
for* n *in* frange(0, 4, 0.5):  
 print(n)

## 迭代器协议

目前为止，在一个对象上实现迭代最简单的方式是使用一个生成器函数。

*class* Node:  
 *def \_\_init\_\_*(self, *value*):  
 self.\_value = *value* self.\_children = []  
 *def \_\_repr\_\_*(self):  
 *return* 'Node({!r})'.format(self.\_value)  
 *def* add\_child(self, *node*):  
 self.\_children.append(*node*)  
 *def \_\_iter\_\_*(self):  
 *return* iter(self.\_children)  
 *def* depth\_first(self):  
 *yield* self  
 *for* c *in* self:  
 *yield from* c.depth\_first()  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 root = Node(0)  
 child1 = Node(1)  
 child2 = Node(2)  
 root.add\_child(child1)  
 root.add\_child(child2)  
 child1.add\_child(Node(3))  
 child1.add\_child(Node(4))  
 child2.add\_child(Node(5))  
 *for* ch *in* root.depth\_first():  
 print(ch)

# Outputs Node(0), Node(1), Node(3), Node(4), Node(2), Node(5)

depth first() 方法简单直观。它首先返回自己本身并迭代每一个子节点并通过调用子节点的depth first() 方法(使用yield from 语句) 返回对应元素。

## reversed—反向迭代

反向迭代仅仅当对象的大小可预先确定或者对象实现了reversed () 的特殊方法时才能生效。如果两者都不符合，那你必须先将对象转换为一个列表才行, 如果可迭代对象元素很多的话，将其预先转换为一个列表要消耗大量的内存。

*class* Countdown:  
 *def \_\_init\_\_*(self, *start*):  
 self.start = *start* # Forward iterator  
 *def \_\_iter\_\_*(self):  
 n = self.start  
 *while* n > 0:  
 *yield* n  
 n -= 1  
 *def \_\_reversed\_\_*(self):  
 n = 1  
 *while* n <= self.start:  
 *yield* n  
 n += 1  
*for* rr *in* reversed(Countdown(30)):  
 print(rr)  
*for* rr *in* Countdown(30):  
 print(rr)

定义一个反向迭代器可以使得代码非常的高效，因为它不再需要将数据填充到一个

列表中然后再去反向迭代这个列表。

## 带有外部状态的生成器

如果想让生成器暴露外部状态 .，可以简单的将它实现为一个类，然后把生成器函数放到\_\_iter\_\_() 方法中

为了使用这个类，可以将它当做是一个普通的生成器函数。然而，由于可以创建一个例对象,于是可以访问内部属性值，比如 history 属性或者是 clear() 方法。

*from* collections *import* deque  
*class* linehistory:  
 *def \_\_init\_\_*(self, *lines*, *histlen*=3):  
 self.lines = *lines* # 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。  
 self.history = deque(maxlen=*histlen*)  
 *def \_\_iter\_\_*(self):  
 #enumerate() 函数用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，  
 # 同时列出数据和数据下标，一般用在 for 循环当中  
 *for* lineno, line *in* enumerate(self.lines, 1):  
 self.history.append((lineno, line))  
 *yield* line  
 *def* clear(self):  
 self.history.clear()  
  
*with* open('./somefile.txt') *as* f:  
 lines = linehistory(f)  
 *for* line *in* lines:  
 *if* 'python' *in* line:  
 *for* lineno, hline *in* lines.history:  
 print('{}:{}'.format(lineno, hline), end='')

关于生成器，很容易掉进函数无所不能的陷阱。如果生成器函数需要跟程序其他部分打交道的话(比如暴露属性值，允许通过方法调用来控制等等)，可能会导致代码异常的复杂。如果是这种情况的话，可以考虑使用上面介绍的定义类的方式。在\_\_iter\_\_ () 方法中定义生成器不会改变任何算法逻辑。由于它是类的一部分，所以允许定义各种属性和方法来供用户使用。

一个需要注意的小地方是，如果你在迭代操作时不使用for 循环语句，那么得先调用iter() 函数。

*with* open('./somefile.txt') *as* f:  
 lines = linehistory(f)  
 it = iter(lines)  
 print(next(it))  
 print(next(it))

## islice—迭代器切片

函数itertools.islice() 适用于在迭代器和生成器上做切片操作

islice(iterable, [start, ] stop [, step])

迭代器和生成器不能使用标准的切片操作，因为它们的长度事先我们并不知道(并

且也没有实现索引)。函数islice() 返回一个可以生成指定元素的迭代器，它通过遍

历并丢弃直到切片开始索引位置的所有元素。然后才开始一个个的返回元素，并直到

切片结束索引位置。

*def* count(*n*):  
 *while True*:  
 *yield n  
 n* += 1  
*import* itertools  
c = count(0)  
#返回序列c的从10开始到20结束的步长为1的元素的迭代器  
*for* x *in* itertools.islice(c, 10, 20):  
 print(x)

islice() 会消耗掉传入的迭代器中的数据。必须考虑到迭代器是不可逆的这个事实。所以如果需要之后再次访问这个迭代器的话，就得先将它里面的数据放入一个列表中。

## dropwhile—跳过可迭代对象的开始部分

dropwhile传递一个函数对象和一个可迭代对象，返回一个迭代器对象，丢弃原有序列中直到函数返回True 之前的所有元素，然后返回后面所有元素。

*from* itertools *import* dropwhile  
*with* open('/etc/passwd') *as* f:  
 *for* line *in* dropwhile(*lambda* line: line.startswith('#'), f):  
 print(line, end='')

如果明确知道了要跳过的元素的个数的话，那么可以使用itertools.islice() 来代替

for x in islice(items, 3, None):

None 参数指定了你要获取从第3 个到最后的所有元素，如果None 和3 的位置对调，意思就是仅仅获取前三个元素恰恰相反，(个跟切片的相反操作[3:] 和[:3] 原理是一样的)

## 排列组合的迭代

1. itertools.permutations()：接受一个集合并产生一个元组序列， 每个元组

由集合中所有元素的一个可能排列组成。也就是说通过打乱集合中元素排列顺序生成

一个元组，如果你想得到指定长度的所有排列，你可以传递一个可选的长度参数。

items = ['a', 'b', 'c']  
*from* itertools *import* permutations  
*for* p *in* permutations(items):  
 print(p)  
*for* p *in* permutations(items, 2):  
 print(p)

1. itertools.combinations()：得到输入集合中元素的所有的组合

对于combinations() 来讲，元素的顺序已经不重要了。也就是说，组合('a','b') 跟('b', 'a') 其实是一样的(最终只会输出其中一个)。在计算组合的时候， 一旦元素被选取就会从候选中剔除掉(比如如果元素’a’ 已经被选取了， 那么接下来就不会再考虑它了)

items = ['a', 'b', 'c']  
*from* itertools *import* combinations  
*for* c *in* combinations(items, 2):  
 print(c)

1. itertools.combinations with replacement() 允许同一个元素被选择多次

items = ['a', 'b', 'c']  
*from* itertools *import* combinations\_with\_replacement  
*for* c *in* combinations\_with\_replacement(items, 3):  
 print(c)

## enumerate—序列上索引值迭代

迭代一个序列的同时跟踪正在被处理的元素索引

my\_list = ['a', 'b', 'c']  
*for* idx, val *in* enumerate(my\_list):  
 print(idx, val)

为了按传统行号输出(行号从1 开始)，可以传递一个开始参数

enumerate(my\_list, 1)

这种情况在遍历文件时想在错误消息中使用行号定位时候非常有用

## zip—同时迭代多个序列

zip(a, b) 会生成一个可返回元组 (x, y) 的迭代器，其中 x 来自 a，y 来自 b。一旦其中某个序列到底结尾，迭代宣告结束。因此迭代长度跟参数中最短序列长度一致。zip() 可以接受多于两个的序列的参数

xpts = [1, 5, 4, 2, 10, 7]  
ypts = [101, 78, 37, 15, 62]  
*for* x, y *in* zip(xpts, ypts):  
 print(x,y)

itertools.zip longest()

*from* itertools *import* zip\_longest  
*for* i *in* zip\_longest(xpts,ypts,fillvalue=0):  
 print(i)

zip() 会创建一个迭代器来作为结果返回。如果你需要将结对的值存储在列表中，要使用 list() 函数

### 多列表生成字典

dict(zip(headers,values))

xpts = [1, 5, 4, 2, 10, 7]  
ypts = [101, 78, 37, 15, 62]  
print(dict(zip(xpts,ypts)))  
*for* x,y *in* zip(xpts,ypts):  
 print(x,'==>',y)

## chain不同集合上元素的迭代

在多个对象执行相同的操作，但是这些对象在不同的容器中，希望代码在不失可读性的情况下避免写重复的循环。

itertools.chain() 方法可以用来简化这个任务。它接受一个可迭代对象列表作为输入，并返回一个迭代器，itertools.chain() 接受一个或多个可迭代对象最为输入参数。然后创建一个迭代 器，依次连续的返回每个可迭代对象中的元素。

*from* itertools *import* chain  
a = [1, 2, 3, 4]  
b = ['x', 'y', 'z']  
*for* x *in* chain(a, b):  
 print(x)

## 数据处理管道

以数据管道 (类似 Unix 管道) 的方式迭代处理数据。比如有个大量的数据需要处理，但是不能将它们一次性放入内存中

可以定义一个由多个执行特定任务独立任务的简单生成器函数组成的容器。

*import* os,fnmatch,gzip,bz2,re  
  
#查找所有符合条件的文件  
*def* gen\_find(*filepat*, *top*):  
 *for* path, dirlist, filelist *in* os.walk(*top*):  
 *for* name *in* fnmatch.filter(filelist, *filepat*):  
 *yield* os.path.join(path,name)  
*def* gen\_opener(*filenames*):  
 *for* filename *in filenames*:  
 *if* filename.endswith('.gz'):  
 f = gzip.open(filename, 'rt')  
 *elif* filename.endswith('.bz2'): f = bz2.open(filename, 'rt')  
 *else*:  
 f = open(filename, 'rt')  
 *yield* f  
 f.close()  
*def* gen\_concatenate(*iterators*):  
 *for* it *in iterators*:  
 *yield from* it  
*def* gen\_grep(*pattern*, *lines*):  
 pat = re.compile(*pattern*)  
 *for* line *in lines*:  
 *if* pat.search(line):  
 *yield* line  
  
lognames = gen\_find('access-log\*', 'www')  
files = gen\_opener(lognames)  
lines = gen\_concatenate(files)  
pylines = gen\_grep('(?i)python', lines)  
#查找包含单词 python 的所有日志行  
*for* line *in* pylines:  
 print(line)  
#计算出传输的字节数并计算其总和  
bytecolumn = (line.rsplit(*None*,1)[1] *for* line *in* pylines)  
bytes = (int(x) *for* x *in* bytecolumn *if* x != '-')  
print('Total', sum(bytes))

yield 语句作为数据的生产者而 for 循环语句 作为数据的消费者。当这些生成器被连在一起后，每个 yield 会将一个单独的数据元 素传递给迭代处理管道的下一阶段。sum() 函数是最终的程序驱动者，每次从生成器管道中提取出一个元素

由于使用了迭代方式处理，代码运行过程中只需要很小很小 的内存。

gen concatenate() 函数将输入序列拼接成一个很长的行序列。 itertools.chain() 函数同样有类似的功能，但是它需要将所有可迭代对象最为参数传入。

类似语句 lines = itertools.chain(\*files) ，使得 gen opener() 生成器能被全部消费掉。但由于 gen opener() 生成器每次生成一个打开过的文件，等到下一个迭代步 骤时文件就关闭了，因此 china() 在这里不能这样使用。上面的方案可以避免这种情况。

gen\_concatenate() 函数中出现过 yield from 语句，它将 yield 操作代理到父生成器上去。语句 yield from it 简单的返回生成器 it 所产生的所有值。

## yield from—展开嵌套的序列

包含yield from 语句的递归生成器可将一个多层嵌套的序列展开成一个单层列表

*from* collections *import* Iterable  
*def* flatten(*items*, *ignore\_types*=(str, bytes)):  
 *for* x *in items*:  
 *if* isinstance(x, Iterable) *and not* isinstance(x, *ignore\_types*):  
 *yield from* flatten(x)  
 *else*:  
 *yield* x  
items = [1, 2, [3, 4, [5, 6], 7], 8]  
# Produces 1 2 3 4 5 6 7 8  
*for* x *in* flatten(items):  
 print(x)

isinstance(x, Iterable) 检查某个元素是否是可迭代的。如果是的话， yield from 就会返回所有子例程的值。最终返回结果就是一个没有嵌套的简单序列了

参数ignore types 和检测语句isinstance(x, ignore types) 用来将字符串和字节排除在可迭代对象外，防止将它们再展开成单个的字符。

## merge—顺序迭代合并后的排序迭代对象

eapq.merge 可迭代特性意味着它不会立马读取所有序列。这就意味着你可以在非常长的序列中使用它，而不会有太大的开销。

它并不会预先读取所有数据到堆栈中或者预先排序，也不会对输入做任何的排序检测。

它仅仅是检查所有序列的开始部分并返回最小的那个，这个过程一直会持续直到所有

输入序列中的元素都被遍历完。

*import* heapq  
a = [1, 4, 7, 10]  
b = [2, 5, 6, 11]  
*for* c *in* heapq.merge(a, b):  
 print(c)

## iter—无限循环

iter 函数接受一个可选的callable 对象和一个标记(结尾) 值作为输入参数。当以这种方式使用的时候，它会创建一个迭代器，这个迭代器会不断调用callable 对象直到返回值和标记值相等为止。

*import* sys  
f = open('/etc/passwd')  
*for* chunk *in* iter(*lambda*: f.read(10), ''):  
 n = sys.stdout.write(chunk)

这种特殊的方法对于一些特定的会被重复调用的函数很有效果，比如涉及到I/O调用的函数。如果想从套接字或文件中以数据块的方式读取数据，通常要不断重复的执行read() 或recv() ，并在后面紧跟一个文件结尾测试来决定是否终止。使用一个简单的iter() 调用就可以将两者结合起来了。其中lambda 函数参数是为了创建一个无参的callable 对象，并为recv 或read() 方法提供了size 参数。

# 文件与IO

## 读写文本

文件的读写操作默认使用系统编码，可以通过调用 sys.getdefaultencoding() 来得到。在大多数机器上面都是 utf-8 编码。如果已经知道要读写的文本是其他编码方式，可以通过传递一个可选的 encoding 参数给 open() 函数。

with open('somefile.txt', 'rt', encoding='latin-1') as f:

latin-1 是字节 0-255 到 U+0000 至 U+00FF 范围内 Unicode 字符的直接映射。当读取一个未知编码的文本时使用 latin-1 编码永远不会产生解码错误。使用 latin-1 编码读取一个文件的时候也许不能产生完全正确的文本解码数据，但是它也能从中提取出足够多的有用数据。同时，如果之后将数据回写回去，原先的数据还是会保留的。

with 语句给被使用到的文件创建了一个上下文环境，with 控制块结束时， 文件会自动关闭。不使用 with 语句，就必须记得手动关闭文件

f = open('somefile.txt', 'rt')  
data = f.read()  
f.close()  
  
*with* open('somefile.txt', 'rt') *as* f:   
 data1 = f.read()  
# Iterate over the lines of the file  
*with* open('somefile.txt', 'rt') *as* f:   
 *for* line *in* f:  
 *pass*

## print—使用其他分隔符或行终止符打印

print() 函数输出数据，但是想改变默认的分隔符或者行尾符 使用 sep 和 end 关键字参数

print('ACME', 50, 91.5, sep=',', end='!!\n')

end 参数也可以在输出中禁止换行

print(i,end=' ')

## 读写字节数据

使用模式为 rb 或 wb 的 open() 函数来读取或写入二进制数据

在读取二进制数据时，需要指明的是所有返回的数据都是字节字符串格式的，而不是文本字符串。类似的，在写入的时候，必须保证参数是以字节形式对外暴露数据的对象 (比如字节字符串，字节数组对象等)。

# Read the entire file as a single byte string  
*with* open('somefile.bin', 'rb') *as* f:  
 data = f.read()  
# Write binary data to a file  
*with* open('somefile.bin', 'wb') *as* f:  
 f.write(b'Hello World')

想从二进制模式的文件中读取或写入文本数据，必须确保要进行解码和编码操作。

*with* open('somefile.bin', 'rb') *as* f:  
 data1 = f.read(16)  
 text = data.decode('utf-8')  
  
*with* open('somefile.bin', 'wb') *as* f:  
 text1 = 'Hello World'  
 f.write(text.encode('utf-8'))

数组和 C 结构体类型能直接被写入，而 不需要中间转换为自己对象。

*import* array  
nums = array.array('i', [1, 2, 3, 4])  
*with* open('data.bin','wb') *as* f:  
 f.write(nums)

很多对象还允许通过使用文件对象的 readinto() 方法直接读取二进制数据到其底层的内存

*import* array  
a = array.array('i', [0, 0, 0, 0, 0, 0, 0, 0])  
*with* open('data.bin', 'rb') *as* f:  
 f.readinto(a)

## 文件不存在才能写

open() 函数中使用 x 模式来代替 w 模式的方法来解决这个问题

*with* open('somefile', 'xt') *as* f:  
 f.write('Hello\n')

## 读写压缩文件

*import* gzip  
*with* gzip.open('somefile.gz', 'rt') *as* f:  
 text = f.read()  
   
*import* bz2  
*with* bz2.open('somefile.bz2', 'wt') *as* f:  
 f.write(text)

选择一个正确的文件模 式是非常重要的。如果不指定模式，默认的就是二进制模式，如果这时候程序想要接受的是文本数据就会出错。

gzip.open() 和 bz2.open() 接受跟内置的 open() 函数一样的参数，包括 encoding，errors，newline 等等。

当写入压缩数据时，可以使用 compresslevel 这个可选的关键字参数来指定一个 压缩级别，默认的等级是 9，也是最高的压缩等级。等级越低性能越好，但是数据压缩程度也 越低。

with gzip.open('somefile.gz', 'wt', compresslevel=5) as f:

## pySerial—与串行端口的数据通信

通过串行端口读写数据，典型场景就是和一些硬件设备打交道

*import* serial  
ser = serial.Serial('/dev/tty.usbmodem641', # Device name varies  
baudrate=9600, bytesize=8, parity='N', stopbits=1)  
ser.write(b'G1 X50 Y50\r\n')  
resp = ser.readline()

设备名对于不同的设备和操作系统是不一样的。比如，在 Windows 系统上，可以使用 0, 1 等表示的一个设备来打开通信端口”COM0” 和”COM1”。一旦端口打开，可以使用 read()，readline() 和 write() 函数读写数据

pySerial 的一个原因是它提供了对高级特性的支持 (比如超时，控制流，缓冲 区刷新，握手协议等等)。

时刻记住所有涉及到串口的 I/O 都是二进制模式的。因此，确保代码使用的是字节而不是文本 (或有时候执行文本的编码/解码操作)。另外当需要创建二进制编码的指令或数据包的时候，struct 模块也是非常有用的。

## pickle—序列化

*import* pickle  
data = ... # Some Python object   
f = open('somefile', 'wb')   
pickle.dump(data, f)

将一个对象转储为一个字符串，可以使用 pickle.dumps()

从字节流中恢复一个对象，使用 picle.load() 或 pickle.loads() 函数

pickle.dump(data)  
  
f = open('somefile', 'rb') data = pickle.load(f)  
data = pickle.loads(s)

dump() 和 load() 函数的使用就是你有效使用 pickle 模块所需的全部了。它可适用于绝大部分 Python 数据类型和用户自定义类的对象实例。如果碰到某个库可以让你在数据库中保存/恢复 Python 对象或者是通过网络传输对象的话，那么很有可能这个库的底层就使用了 pickle 模块。

pickle 是一种 Python 特有的自描述的数据编码。通过自描述，被序列化后的数据包含每个对象开始和结束以及它的类型信息。因此，无需担心对象记录的定义，它总是能工作。

# 数据编码和处理

## 编码和解码十六进制

将一个十六进制字符串解码成一个字节字符串或者将一个字节字符串编码成一个十六进制字符

s = b'hello'  
# Encode as hex  
*import* binascii  
h = binascii.b2a\_hex(s)  
# Decode back to bytes   
h=binascii.a2b\_hex(h)

类似的功能同样可以在 base64 模块中找到

*import* base64  
h = base64.b16encode(s)   
base64.b16decode(h)

函数 base64.b16decode() 和 base64.b16encode() 只能 操作大写形式的十六进制字母，而 binascii 模块中的函数大小写都能处理

编码函数所产生的输出总是一个字节字符串。如果想强制以 Unicode 形式输出，需要增加一个额外的界面步骤

print(h.decode('ascii'))

解码十六进制数时，函数 b16decode() 和 a2b hex() 可以接受字节或 unicode 字 符串。但是，unicode 字符串必须仅仅只包含 ASCII 编码的十六进制数。

# 函数

## \*—接受任意数量参数

1. 让一个函数接受任意数量的位置参数，可以使用一个 \* 参数。

*def* avg(*first*, *\*rest*):  
 print(*rest*)  
 print ((*first* + sum(*rest*)) / (1 + len(*rest*)))  
avg(1, 2) # 1.5  
avg(1, 2, 3, 4) # 2.5

rest 是由所有其他位置参数组成的元组

1. 接受任意数量的关键字参数，使用一个以 \*\* 开头的参数。

*import* html  
*def* make\_element(*name*, *value*, *\*\*attrs*):  
 keyvals = [' %s="%s"' % item *for* item *in attrs*.items()]  
 attr\_str = ''.join(keyvals)  
 element = '<{name}{attrs}>{value}</{name}>'.format(  
 name=*name*,  
 attrs=attr\_str,  
 value=html.escape(*value*))  
 print(element)   
make\_element('item', 'Albatross', size='large', quantity=6)  
make\_element('p', '<spam>')

某个函数能同时接受任意数量的位置参数和关键字参数，可以同时使 用 \* 和 \*\*

一个 \* 参数只能出现在函数定义中最后一个位置参数后面，而\*\*参数只能出现在最后一个参数。在 \* 参数后面仍然可以定义其他参数，即7.2-强制关键字。

## 强制关键字参数

将强制关键字参数放到某个 \* 参数或者当个 \* 后面

*def* recv(maxsize, \*, block):   
 *pass*recv(1024, *True*) # TypeError  
recv(1024, block=*True*) # Ok

还能在接受任意多个位置参数的函数中指定关键字参数

*def* mininum(*\*values*, *clip*=*None*):  
 m = min(*values*)  
 *if clip is not None*:  
 m = *clip if clip* > m *else* m  
 *return* m  
  
mininum(1, 5, 2, -5, 10) # Returns -5  
mininum(1, 5, 2, -5, 10, clip=0) # Returns 0

强制关键字参数在一些更高级场合同样也很有用。例如，它们可以被用来在使用 \*args 和 \*\*kwargs 参数作为输入的函数中插入参数

## 函数参数增加元信息

可以使用任意类型的对象给函数添加注解 (例如数字，字符串，对象实例等 等)，不过通常来讲使用类或着字符串会比较好点

*def* add(*x*:int, *y*:int) -> int:  
 *return x* + *y*

## 匿名或内联函数

当一些函数很简单的时候，就可以使用 lambda表达式来代替

尽管 lambda 表达式允许你定义简单函数，但是它的使用是有限制的。你只能指定 单个表达式，它的值就是最后的返回值。也就是说不能包含其他的语言特性了，包括 多个语句、条件表达式、迭代以及异常处理等等。

## 匿名函数捕获变量值

lambda 表达式中的 x 是一个自由变量，在运行时绑定值，而不 是定义时就绑定， 在调用lambda 表 达式的时候，x 的值是执行时的值。

x = 10  
a = *lambda* y: x + y  
x = 20  
b = *lambda* y: x + y  
print(a(10),b(10))

如果你想让某个匿名函数在定义时就捕获到值，可以将那个参数值定义成默认参数即可

x = 10  
a = *lambda* y,x=x: x + y  
x = 20  
b = *lambda* y,x=x: x + y  
print(a(10),b(10))

通过使用函数默认值参数形式，lambda 函数在定义时就能绑定到值。

## 减少可调用对象的参数个数

减少某个函数的参数个数，可以使用 functools.partial()

partial() 函数允许你给一个或多个参数设置固定的值，减少接下来被调用时的参数个数。

*from* functools *import* partial  
*def* spam(*a*, *b*, *c*, *d*):  
 print(*a*, *b*, *c*, *d*)  
  
s1 = partial(spam, 1)  
s2 = partial(spam, d=42)  
s3 = partial(spam, 1, 2, d=42)

partial() 通常被用来微调其他库函数所使用的回调函数的参数。

## nonlocal—访问闭包中定义的变量

通常来讲，闭包的内部变量对于外界来讲是完全隐藏的。但是，你可以通过编写访问函数并将其作为函数属性绑定到闭包

nonlocal 声明可以让我们编写函数来修改内部变量的值。其次，函数属性允许我们用一种很简单的方式将访问方法绑定到闭包函数上，

*def* sample():  
 n = 0  
 # Closure function  
 *def* func():  
 print('n=', n)  
 # Accessor methods for n  
 *def* get\_n():  
 *return* n  
 *def* set\_n(*value*):  
 *nonlocal* n  
 n = *value* # Attach as function attributes  
 func.get\_n = get\_n  
 func.set\_n = set\_n  
 *return* func  
  
f = sample()  
f.set\_n(10)

还可以进一步的扩展，让闭包模拟类的实例。你要做的仅仅是复制上面的内部函数到一个字典实例中并返回它即可

*import* sys  
*class* ClosureInstance:  
 *def \_\_init\_\_*(self, *locals*=*None*):  
 *if locals is None*:  
 locals = sys.\_getframe(1).f\_locals  
 # Update instance dictionary with callables  
 self.\_\_dict\_\_.update((key,value) *for* key, value *in locals*.items() *if* callable(value) )  
 # Redirect special methods  
 *def \_\_len\_\_*(self):  
 *return* self.\_\_dict\_\_['\_\_len\_\_']()  
 # Example use  
*def* Stack():  
 items = []  
 *def* push(*item*):  
 items.append(*item*)  
 *def* pop():  
 *return* items.pop()  
 *def* \_\_len\_\_():  
 *return* len(items)  
 *return* ClosureInstance()

闭包的方案运行起来要快，大部分原因是因为对实例变量的简化访问，闭包更快是因为不会涉及到额外的self 变量。

# 类与对象

## 改变对象的字符串显示

改变一个实例的字符串表示，可重新定义它的\_\_str\_\_ () 和\_\_repr\_\_ () 方法。

\_\_repr\_\_ () 方法返回一个实例的代码表示形式，通常用来重新构造这个实例。内置的\_\_repr\_\_ () 函数返回这个字符串，跟使用交互式解释器显示的值是一样的。

\_\_str\_\_ () 方法将实例转换为一个字符串，使用str() 或print() 函数会输出这个字符串

自定义repr () 和str () 通常是很好的习惯，因为它能简化调试和实例输出。

*class* Pair:  
 *def \_\_init\_\_*(self, *x*, *y*):  
 self.x = *x* self.y = *y  
 def \_\_repr\_\_*(self):  
 *return* 'Pair({0.x!r}, {0.y!r})'.format(self)  
 *def \_\_str\_\_*(self):  
 *return* '({0.x!s}, {0.y!s})'.format(self)

格式化代码{0.x}对应的是第1 个参数的x 属性。因此， 0 实际上指的就是self 本身

作为这种实现的一个替代，你也可以使用% 操作符

*def \_\_repr\_\_*(self):  
 *return* 'Pair(%r, %r)' % (self.x, self.y)

## 自定义字符串的格式化

为了自定义字符串的格式化，我们需要在类上面定义format () 方法。

\_formats = {  
 'ymd' : '{d.year}-{d.month}-{d.day}',  
 'mdy' : '{d.month}/{d.day}/{d.year}',  
 'dmy' : '{d.day}/{d.month}/{d.year}'  
 }  
*class* Date:  
 *def \_\_init\_\_*(self, *year*, *month*, *day*):  
 self.year = *year* self.month = *month* self.day = *day  
 def \_\_format\_\_*(self, *code*):  
 *if code* == '':  
 code = 'ymd'  
 fmt = \_formats[*code*]  
 *return* fmt.format(d=self)

## with—支持上下文管理协议

*from* socket *import* socket, AF\_INET, SOCK\_STREAM  
*class* LazyConnection:  
 *def \_\_init\_\_*(self, *address*, *family*=AF\_INET, *type*=SOCK\_STREAM):  
 self.address = *address* self.family = *family* self.type = *type* self.connections = []  
 *def \_\_enter\_\_*(self):  
 sock = socket(self.family, self.type)  
 sock.connect(self.address)  
 self.connections.append(sock)  
 *return* sock  
 *def \_\_exit\_\_*(self, *exc\_ty*, *exc\_val*, *tb*):  
 self.connections.pop().close()  
  
*from* functools *import* partial  
conn = LazyConnection(('www.python.org', 80))  
# Connection closed  
*with* conn *as* s:  
# conn.\_\_enter\_\_() executes: connection open  
 s.send(b'GET /index.html HTTP/1.0\r\n')  
 s.send(b'Host: www.python.org\r\n')  
 s.send(b'\r\n')  
 resp = b''.join(iter(partial(s.recv, 8192), b''))

编写上下文管理器的主要原理是你的代码会放到with 语句块中执行。当出现with语句的时候，对象的enter () 方法被触发，它返回的值(如果有的话) 会被赋值给as 声明的变量。然后，with 语句块里面的代码开始执行。最后， exit () 方法被触发进行清理工作

不管with 代码块中发生什么，上面的控制流都会执行完，就算代码块中发生了异常也是一样的。事实上， exit () 方法的第三个参数包含了异常类型、异常值和追溯信息(如果有的话)。exit () 方法能自己决定怎样利用这个异常信息，或者忽略它并返回一个None 值。如果exit () 返回True ，那么异常会被清空，就好像什么都没发生一样， with 语句后面的程序继续在正常执行

## \_\_slots\_\_—创建大量对象时节省内存

对于主要是用来当成简单的数据结构的类而言，你可以通过给类添加slots 属性来极大的减少实例所占的内存。

*class* Date:  
\_\_slots\_\_ = ['year', 'month', 'day']  
*def \_\_init\_\_*(*self*, *year*, *month*, *day*):  
 *self*.year = *year  
 self*.month = *month  
 self*.day = *day*

定义\_\_slots\_\_ 后，Python 就会为实例使用一种更加紧凑的内部表示。实例通过一个很小的固定大小的数组来构建，而不是为每个实例定义一个字典，这跟元组或列表很类似。在\_\_slots\_\_中列出的属性名在内部被映射到这个数组的指定小标上。

使用\_\_slots\_\_一个不好的地方就是不能再给实例添加新的属性了，只能使用在\_\_slots\_\_中定义的那些属性名。

## 类中封装属性名

封装类的实例上面的“私有”数据

Python不依赖语言特性去封装数据，而是通过遵循一定的属性和方法命名规约来达到这个效果。

1. 任何以单下划线开头的名字都应该是内部实现。
2. 使用双下划线开始会导致访问名称变成其他形式，这种属性通过继承是无法被覆盖的

*class* A:  
 *def \_\_init\_\_*(self):  
 self.\_internal = 0 # An internal attribute  
 self.public = 1 # A public attribute  
 *def* public\_method(self):  
 *pass  
 def* \_internal\_method(self):  
 *pass  
class* B:  
 *def \_\_init\_\_*(self):  
 self.\_\_private = 0  
 *def* \_\_private\_method(self):  
 *pass  
 def* public\_method(self):  
 *pass* self.\_\_private\_method()

类B 中，私有属性会被分别重命名为\_B\_\_private 和\_B\_\_private method

重命名的目的是什么，答案就是继承——这种属性通过继承是无法被覆盖的

*class* C(B):  
 *def \_\_init\_\_*(self):  
 super().\_\_init\_\_()  
 self.\_\_private = 1 # Does not override B.\_\_private  
 # Does not override B.\_\_private\_method()  
 *def* \_\_private\_method(self):  
 *pass*

## 创建可管理的属性

自定义某个属性的一种简单方法是将它定义为一个property

*class* Person:  
 *def \_\_init\_\_*(self, *first\_name*):  
 self.first\_name = *first\_name* # Getter function  
 @property  
 *def* first\_name(self):  
 *return* self.\_first\_name  
 # Setter function  
 @first\_name.setter  
 *def* first\_name(self, *value*):  
 *if not* isinstance(*value*, str):  
 *raise* TypeError('Expected a string')  
 self.\_first\_name = *value* # Deleter function (optional)  
 @first\_name.deleter  
 *def* first\_name(self):  
 *raise* AttributeError("Can't delete attribute")

上述代码中有三个相关联的方法，这三个方法的名字都必须一样。第一个方法是一个getter 函数，它使得first name 成为一个属性。其他两个方法给first\_name 属性添加了setter 和deleter 函数。需要强调的是只有在first\_name 属性被创建后，后面的两个装饰器@first name.setter 和@first name.deleter 才能被定义。

property 的一个关键特征是它看上去跟普通的attribute 没什么两样，但是访问它的时候会自动触发getter 、setter 和deleter 方法。

在实现一个property 的时候，底层数据(如果有的话) 仍然需要存储在某个地方。因此，在get 和set 方法中，会看到对\_first\_name 属性的操作，这也是实际数据保存的地方。

另外， \_\_init\_\_ () 方法中设置了self. first\_name 而不是self. \_first\_name 。创建一个property 的目的就是在设置attribute 的时候进行检查。因此，你可能想在初始化的时候也进行这种类型检查。通过设置self.first\_name ，自动调用setter 方法，这个方法里面会进行参数的检查，否则就是直接访问self. \_first\_name 了。

*class* Person:  
 *def \_\_init\_\_*(self, *first\_name*):  
 self.set\_first\_name(*first\_name*)  
 # Getter function  
 *def* get\_first\_name(self):  
 *return* self.\_first\_name  
 # Setter function  
 *def* set\_first\_name(self, *value*):  
 *if not* isinstance(*value*, str):  
 *raise* TypeError('Expected a string')  
 self.\_first\_name = *value* # Deleter function (optional)  
 *def* del\_first\_name(self):  
 *raise* AttributeError("Can't delete attribute")  
 # Make a property from existing get/set methods  
 name = property(get\_first\_name, set\_first\_name, del\_first\_name)

在已存在的get 和set 方法基础上定义property

一个property 属性其实就是一系列相关绑定方法的集合。以后想给普通attribute 访问添加额外的处理逻辑的时候，你可以将它变成一个property 而无需改变原来的代码。因为访问

attribute 的代码还是保持原样。

Properties 还是一种定义动态计算attribute 的方法。这种类型的attributes 并不会被实际的存储，而是在需要的时候计算出来

## super—调用父类方法

super() 函数的一个常见用法是在init () 方法中确保父类被正确的初始化了：

*class* A:  
 *def \_\_init\_\_*(self):  
 self.x = 0  
*class* B(A):  
 *def \_\_init\_\_*(self):  
 super().\_\_init\_\_()  
 self.y = 1

super() 的另外一个常见用法出现在覆盖Python 特殊方法的代码中，比如：

*class* Proxy:  
 *def \_\_init\_\_*(self, *obj*):  
 self.\_obj = *obj* # Delegate attribute lookup to internal obj  
 *def \_\_getattr\_\_*(self, *name*):  
 *return* getattr(self.\_obj, *name*)  
 # Delegate attribute assignment  
 *def \_\_setattr\_\_*(self, *name*, *value*):  
 *if name*.startswith('\_'):  
 super().\_\_setattr\_\_(*name*, *value*) # Call original \_\_setattr\_\_  
 *else*:  
 setattr(self.\_obj, *name*, *value*)

\_\_setattr\_\_ () 的实现包含一个名字检查。如果某个属性名以下划线( \_ )开头，就通过super() 调用原始的\_\_setattr\_\_ () ，否则的话就委派给内部的代理对象self. \_obj 去处理。就算没有显式的指明某个类的父类， super() 仍然可以有效的工作。

super() 有个令人吃惊的地方是它并不一定去查找某个类在MRO 中下一个直接父类，甚至可以在一个没有直接父类的类中使用它。

## 子类中扩展property

扩展定义在父类中的property 的功能

*class* Person:  
 *def \_\_init\_\_*(self, *name*):  
 self.name = *name* # Getter function  
 @property  
 *def* name(self):  
 *return* self.\_name  
 # Setter function  
 @name.setter  
 *def* name(self, *value*):  
 *if not* isinstance(*value*, str):  
 *raise* TypeError('Expected a string')  
 self.\_name = *value* # Deleter function  
 @name.deleter  
 *def* name(self):  
 *raise* AttributeError("Can't delete attribute")  
  
*class* SubPerson(Person):  
 @property  
 *def* name(self):  
 print('Getting name')  
 *return* super().name  
 @name.setter  
 *def* name(self, *value*):  
 print('Setting name to', *value*)  
 super(SubPerson, SubPerson).name.\_\_set\_\_(self, *value*)  
 @name.deleter  
 *def* name(self):  
 print('Deleting name')  
 super(SubPerson, SubPerson).name.\_\_delete\_\_(self)

所有的property 方法都被重新定义。在每一个方法中，使用了super() 来调用父类的实现

为了委托给之前定义的setter 方法，需要将控制权传递给之前定义的name 属性的\_\_set\_\_ () 方法. 获取这个方法的唯一途径是使用类变量而不是实例变量来访问它。这也是使用super(SubPerson, SubPerson) 的原因。

1.仅仅只想扩展property 的某一个方法

*class* SubPerson(Person):  
 @Person.name.getter  
 *def* name(self):  
 print('Getting name')  
 *return* super().name

2.只想修改setter 方法

*class* SubPerson(Person):  
 @Person.name.setter  
 *def* name(self, *value*):  
 print('Setting name to', *value*)  
 super(SubPerson, SubPerson).name.\_\_set\_\_(self, *value*)

property 之前已经定义过的方法会被复制过来，而函数被替换。

在子类中扩展一个property 可能会引起很多不易察觉的问题，因为一个property其实是getter、setter 和deleter 方法的集合，而不是单个方法。因此，扩展一个property 的时候你需要先确定是否要重新定义所有的方法还是说只修改其中某一个。

## 创建新的类或实例属性

创建一个全新的实例属性，可以通过一个描述器类的形式来定义它的功能。为了使用一个描述器，需将这个描述器的实例作为类属性放到一个类的定义中

它只能在类级别被定义，而不能为每个实例单独定义

*class* Integer:  
 *def \_\_init\_\_*(self, *name*):  
 self.name = *name  
 def \_\_get\_\_*(self, *instance*, *cls*):  
 *if instance is None*:  
 *return* self  
 *else*:  
 *return instance*.\_\_dict\_\_[self.name]  
 *def \_\_set\_\_*(self, *instance*, *value*):  
 *if not* isinstance(*value*, int):  
 *raise* TypeError('Expected an int')  
 *instance*.\_\_dict\_\_[self.name] = *value  
 def \_\_delete\_\_*(self, *instance*):  
 *del instance*.\_\_dict\_\_[self.name]

一个描述器就是一个实现了三个核心的属性访问操作(get, set, delete) 的类，分别

为\_\_get \_\_ () 、\_\_set\_\_ () 和\_\_delete\_\_ () 这三个特殊的方法。这些方法接受一个实例

作为输入，之后相应的操作实例底层的字典。

*class* Point:  
 x = Integer('x')  
 y = Integer('y')  
*def \_\_init\_\_*(*self*, *x*, *y*):  
 *self*.x = *x  
 self*.y = *y*

所有对描述器属性(如x 或y) 的访问会被\_\_get\_\_ () 、\_\_set\_\_ ()和\_\_delete\_\_ () 方法捕获到。

描述器可实现大部分Python 类特性中的底层魔法， 包括@classmethod 、@staticmethod 、@property ，甚至是\_\_slots\_\_ 特性。

最后要指出的一点是，如果只是想简单的自定义某个类的单个属性访问的话就不用去写描述器了。这种情况下使用property 技术会更加容易。当程序中有很多重复代码的时候描述器就很有用了(比如想在代码的很多地方使用描述器提供的功能或者将它作为一个函数库特性)

## 延迟计算属性

将一个只读属性定义成一个property，并且只在访问的时候才会计算结果。但是一旦被访问后，结果值被缓存起来，不用每次都去计算

定义一个延迟属性的一种高效方法是通过使用一个描述器类

*class* lazyproperty:  
 *def \_\_init\_\_*(self, *func*):  
 self.func = *func  
 def \_\_get\_\_*(self, *instance*, *cls*):  
 *if instance is None*:  
 *return* self  
 *else*:  
 value = self.func(*instance*)  
 setattr(*instance*, self.func.\_\_name\_\_, value)  
 *return* value  
  
*import* math  
*class* Circle:  
 *def \_\_init\_\_*(self, *radius*):  
 self.radius = *radius* @lazyproperty  
 *def* area(self):  
 print('Computing area')  
 *return* math.pi \* self.radius \*\* 2  
 @lazyproperty  
 *def* perimeter(self):  
 print('Computing perimeter')  
 *return* 2 \* math.pi \* self.radius

很多时候，构造一个延迟计算属性的主要目的是为了提升性能。

## 简化数据结构的初始化

可以在一个基类中写一个公用的init () 函数，然后使你的类继承自这个基类

如果还想支持关键字参数，可以将关键字参数设置为实例属性

*class* Structure2:  
 \_fields = []  
 *def \_\_init\_\_*(self, *\*args*, *\*\*kwargs*):  
 *if* len(*args*) > len(self.\_fields):  
 *raise* TypeError('Expected {} arguments'.format(len(self.\_fields)))  
 # Set all of the positional arguments  
 *for* name, value *in* zip(self.\_fields, *args*):  
 setattr(self, name, value)  
 # Set the remaining keyword arguments  
 *for* name *in* self.\_fields[len(*args*):]:  
 setattr(self, name, *kwargs*.pop(name))  
 # Check for any remaining unknown arguments  
 *if kwargs*:  
 *raise* TypeError('Invalid argument(s): {}'.format(','.join(*kwargs*)))  
# Example use  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 *class* Stock(Structure2):  
 \_fields = ['name', 'shares', 'price']  
 s1 = Stock('ACME', 50, 91.1)  
 s2 = Stock('ACME', 50, price=91.1)  
 s3 = Stock('ACME', shares=50, price=91.1)  
# s3 = Stock('ACME', shares=50, price=91.1, aa=1)

## abc—定义接口或者抽象基类

定义一个接口或抽象类，并且通过执行类型检查来确保子类实现了某些特定方法

使用abc 模块可以很轻松的定义抽象基类，抽象类的一个特点是它不能直接被实例化

*from* abc *import* ABCMeta, abstractmethod  
*class* IStream(metaclass=ABCMeta):  
 @abstractmethod  
 *def* read(self, *maxbytes*=-1):  
 *pass* @abstractmethod  
 *def* write(self, *data*):  
 *pass*

抽象基类的一个主要用途是在代码中检查某些类是否为特定类型，实现了特定接口

1. 抽象类的目的就是让别的类继承它并实现特定的抽象方法

*class* SocketStream(IStream):  
 *def* read(self, *maxbytes*=-1):  
 *pass  
 def* write(self, *data*):  
 *pass*

1. 还可以通过注册方式来让某个类实现抽象基类

*import* io  
# Register the built-in I/O classes as supporting our interface  
IStream.register(io.IOBase)  
# Open a normal file and type check  
f = open('foo.txt')  
isinstance(f, IStream) # Returns True

@abstractmethod 还能注解静态方法、类方法和properties 。你只需保证这个注解紧靠在函数定义前即可

*class* A(metaclass=ABCMeta):  
 @property  
 @abstractmethod  
 *def* name(self):  
 *pass* @name.setter  
 @abstractmethod  
 *def* name(self, *value*):  
 *pass* @classmethod  
 @abstractmethod  
 *def* method1(cls):  
 *pass* @staticmethod  
 @abstractmethod  
 *def* method2():  
 *pass*

## 数据模型的类型约束

定义某些在属性赋值上面有限制的数据结构，需要在对某些实例属性赋值时进行检查。所以自定义属性赋值函数，这种情况下最好使用描述器。

# Base class. Uses a descriptor to set a value  
*class* Descriptor:  
 *def \_\_init\_\_*(self, *name*=*None*, *\*\*opts*):  
 self.name = *name  
 for* key, value *in opts*.items():  
 setattr(self, key, value)  
 *def \_\_set\_\_*(self, *instance*, *value*):  
 *instance*.\_\_dict\_\_[self.name] = *value*# Descriptor for enforcing types  
*class* Typed(Descriptor):  
 expected\_type = type(*None*)  
 *def \_\_set\_\_*(self, *instance*, *value*):  
 *if not* isinstance(*value*, self.expected\_type):  
 *raise* TypeError('expected ' + str(self.expected\_type))  
 super().\_\_set\_\_(*instance*, *value*)  
  
# Descriptor for enforcing values  
*class* Unsigned(Descriptor):  
 *def \_\_set\_\_*(self, *instance*, *value*):  
 *if value* < 0:  
 *raise* ValueError('Expected >= 0')  
 super().\_\_set\_\_(*instance*, *value*)  
  
*class* MaxSized(Descriptor):  
 *def \_\_init\_\_*(self, *name*=*None*, *\*\*opts*):  
 *if* 'size' *not in opts*:  
 *raise* TypeError('missing size option')  
 super().\_\_init\_\_(*name*, \*\**opts*)  
  
 *def \_\_set\_\_*(self, *instance*, *value*):  
 *if* len(*value*) >= self.size:  
 *raise* ValueError('size must be < ' + str(self.size))  
 super().\_\_set\_\_(*instance*, *value*)

定义各种不同的数据类型

*class* Integer(Typed):  
 expected\_type = int  
  
*class* UnsignedInteger(Integer, Unsigned):  
 *pass  
  
class* Float(Typed):  
 expected\_type = float  
  
*class* UnsignedFloat(Float, Unsigned):  
 *pass  
  
class* String(Typed):  
 expected\_type = str  
  
*class* SizedString(String, MaxSized):  
 *pass*

还有一些技术可以简化上面的代码，其中一种是使用类装饰器，另外一种方式是使用元类

所有方法中，类装饰器方案应该是最灵活和最高明的。

1. 首先，它并不依赖任何其他新的技术，比如元类。
2. 其次，装饰器可以很容易的添加或删除。
3. 最后，装饰器还能作为混入类的替代技术来实现同样的效果;

# Decorator for applying type checking  
*def* Typed(*expected\_type*, *cls*=*None*):  
 *if cls is None*:  
 *return lambda cls*: Typed(*expected\_type*, *cls*)  
 super\_set = *cls*.\_\_set\_\_  
 *def \_\_set\_\_*(*self*, *instance*, *value*):  
 *if not* isinstance(*value*, expected\_type):  
 *raise* TypeError('expected ' + str(expected\_type))  
 super\_set(*self*, *instance*, *value*)  
 *cls*.\_\_set\_\_ = \_\_set\_\_  
 *return cls*# Decorator for unsigned values  
*def* Unsigned(*cls*):  
 super\_set = *cls*.\_\_set\_\_  
 *def \_\_set\_\_*(*self*, *instance*, *value*):  
 *if value* < 0:  
 *raise* ValueError('Expected >= 0')  
 super\_set(*self*, *instance*, *value*)  
 *cls*.\_\_set\_\_ = \_\_set\_\_  
 *return cls*# Decorator for allowing sized values  
*def* MaxSized(*cls*):  
 super\_init = *cls*.\_\_init\_\_  
 *def \_\_init\_\_*(*self*, *name*=*None*, *\*\*opts*):  
 *if* 'size' *not in opts*:  
 *raise* TypeError('missing size option')  
 super\_init(*self*, *name*, \*\**opts*)  
 *cls*.\_\_init\_\_ = \_\_init\_\_  
 super\_set = *cls*.\_\_set\_\_  
  
 *def \_\_set\_\_*(*self*, *instance*, *value*):  
 *if* len(*value*) >= *self*.size:  
 *raise* ValueError('size must be < ' + str(*self*.size))  
 super\_set(*self*, *instance*, *value*)  
 *cls*.\_\_set\_\_ = \_\_set\_\_  
 *return cls*# Specialized descriptors  
@Typed(int)  
*class* Integer(Descriptor):  
 *pass*@Unsigned  
*class* UnsignedInteger(Integer):  
 *pass*@Typed(float)  
*class* Float(Descriptor):  
 *pass*@Unsigned  
*class* UnsignedFloat(Float):  
 *pass*@Typed(str)  
*class* String(Descriptor):  
 *pass*@MaxSized  
*class* SizedString(String):  
 *pass*

这种方式定义的类跟之前的效果一样，而且执行速度会更快。设置一个简单的类型属性的值，装饰器方式要比之前的混入类的方式几乎快100%

## 自定义容器

collections 定义了很多抽象基类，当你想自定义容器类的时候它们会非常有用。

比如你想让你的类支持迭代，那就让你的类继承collections.Iterable 即可，不过你需要实现collections.Iterable 所有的抽象方法，否则会报错。你可以先试着去实例化一个对象，在错误提示中可以找到需要实现哪些方法

*import* collections  
*class* A(collections.Iterable):  
 *pass*

使用collections 中的抽象基类可以确保你自定义的容器实现了所有必要的方法。并且还能简化类型检查。

collections 中很多抽象类会为一些常见容器操作提供默认的实现， 这样一来你只需要实现那些你最感兴趣的方法即可。

## 属性的代理访问

代理是一种编程模式，它将某个操作转移给另外一个对象来实现。