# 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* self.history = deque(maxlen=*histlen*)  
 *def \_\_iter\_\_*(self):  
 *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='')