## 数据结构和算法

### 1.1 将序列分解为单独的变量

#### 1.1.1 问题

把一个包含N个元素的元组或序列，将它分解为N个独立的变量。

#### 1.1.2 解决方案

任何序列（或可迭代对象）都可以通过一个简单的赋值操作来分解为单独的变量。

如果元素的数目不匹配，将得到一个错误提示。

#### 1.1.3 讨论

只要对象恰好是可迭代的，那么就可以执行分解操作。

s = *"Hello"*

a, b, c, d, e = s

print a,b,c,d,e

当做分解操作时，有时候想要丢弃默写特定的值，通常可以选用一个用不到的变量名，以此作为要丢弃的值的名称，例如\_,但要确保变量名没有在其他地方用过。

\_, shares, price,\_ = [1,2,3,4]

### 1.2 从任意长度的可迭代对象中分解元素

#### 1.2.1 问题

需要从某个可迭代对象中分解N个元素，但是这个可迭代对象的长度可能超过N。

#### 1.2.2 解决方案

python的“\*表达式”可以用来解决这个问题（python3.x版本）。

def **avg**(seq):

return sum(seq)/len(seq)

def **drop\_first\_last**(grades):

first, \*middle, last = grades

return avg(middle)

print drop\_first\_last([1,2,3,4])

#### 1.2.3 讨论

### 1.3 保存最后N各元素

#### 1.3.1 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？

#### 1.3.2 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 li in lines:

if pattern in li:

yield li, previous\_lines

previous\_lines.append(li)

# Example use on a file

if \_\_name\_\_ == *'\_\_main\_\_'*:

with open(*r'somefile.txt'*) as f:

for line, prevlines in search(f, *'python'*, 5):

#print "line=%s prevlines=%s" %(line, prevlines)

for pline in prevlines:

print *"pline:"*, pline,

print *"line:"*, line,

print(*'-'* \* 20)

#### 1.3.3 讨论

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。

>>> 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.appendleft(5)

>>> q

deque([5, 2, 3], maxlen=3)

>>> q.pop()

3

>>> q

deque([5, 2], maxlen=3)

>>> q.popleft()

5

>>> q

deque([2], maxlen=3)

>>>

### 1.4 找到最大或最小的N个元素

#### 1.4.1 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？

#### 1.4.2 解决方案

heapq（堆数据结构模块）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]

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'*])

print cheap

print expensive

### 1.5 实现一个优先级队列

#### 1.5.1 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

#### 1.5.2 解决方案

## 字符串和文本

### 2.1 使用多个界定符分割字符串

#### 2.1.1问题

你需要将一个字符串分割为多个字段，但是分隔符(还有周围的空格)并不是固定的。

#### 2.1.2 解决方案

string 对象的 split() 方法只适应于非常简单的字符串分割情形， 它并不允许有多个分隔符或者是分隔符周围不确定的空格。 当你需要更加灵活的切割字符串的时候，最好使用 **re.split()**方法：

当你使用 re.split() 函数时候，需要特别注意的是正则表达式中是否包含**一个括号捕获分组**。 如果使用了捕获分组，那么被匹配的文本也将出现在结果列表中。比如，观察一下这段代码运行后的结果：

如果你不想保留分割字符串到结果列表中去，但仍然需要使用到括号来分组正则表达式的话， 确保你的分组是非捕获分组，形如**(?:...)**。比如：

import re

line = *'asdf fjdk; afed, fjek,asdf, foo'*

sp = re.split(*r'[;,\s]\s\*'*, line)

print sp #['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']

fileds = re.split(*r'(;|,|\s)\s\*'*, line)

print fileds #['asdf', ' ', 'fjdk', ';', 'afed', ',', 'fjek', ',', 'asdf', ',', 'foo']

nocatch =re.split(*r'(?:,|;|\s)\s\*'*, line)

print nocatch #['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']

### 2.2 字符串开头或结尾匹配

#### 2.2.1 问题

你需要通过指定的文本模式去检查字符串的开头或者结尾，比如文件名后缀，URL Scheme 等等。

#### 2.2.2 解决方案

检查字符串开头或结尾的一个简单方法是使用 **str.startswith()**或者是**str.endswith()**方法。比如：

filename = *'spam.txt'*

print filename.startswith(*'file:'*) #False

print filename.endswith(*'.txt'*) #True

url = *'http://www.python.org'*

print url.startswith(*'http:'*) #True

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：

import os

filenames = os.listdir(*'.'*)

print [name for name in filenames if name.endswith((*'.py'*, *'pyc'*))]

奇怪的是，startswith和endswith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。

### 2.3 用 Shell 通配符匹配字符串

#### 2.3.1 问题

你想使用 Unix Shell 中常用的通配符(比如 \*.py , Dat[0-9]\*.csv 等)去匹配文本字符串。

#### 2.3.2 解决方案

fnmatch 模块提供了两个函数—— fnmatch() 和 fnmatchcase()，可以用来实现这样的匹配。用法如下：

>>> from fnmatch import fnmatch, fnmatchcase

>>> fnmatch('foo.txt', '\*.txt')

True

>>> fnmatch('foo.txt', '?oo.txt')

True

>>> fnmatch('Dat45.csv', 'Dat[0-9]\*')

True

>>> names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'foo.py']

>>> [name for name in names if fnmatch(name, 'Dat\*.csv')]

['Dat1.csv', 'Dat2.csv']

fnmatch() 函数使用底层操作系统的大小写敏感规则(不同的系统是不一样的)来匹配模式。比如：

>>> # On OS X (Mac)

>>> fnmatch('foo.txt', '\*.TXT')

False

>>> # On Windows

>>> fnmatch('foo.txt', '\*.TXT')

True

>>>

如果你对这个区别很在意，可以使用 fnmatchcase() 来代替。它完全使用你的模式大小写匹配。比如：

>>> fnmatchcase('foo.txt', '\*.TXT')

False

>>>

这两个函数通常会被忽略的一个特性是在处理非文件名的字符串时候它们也是很有用的。 比如，假设你有一个街道地址的列表数据：

addresses = [

'5412 N CLARK ST',

'1060 W ADDISON ST',

'1039 W GRANVILLE AVE',

'2122 N CLARK ST',

'4802 N BROADWAY',

]

你可以像这样写列表推导：

>>> from fnmatch import fnmatchcase

>>> [addr for addr in addresses if fnmatchcase(addr, '\* ST')]

['5412 N CLARK ST', '1060 W ADDISON ST', '2122 N CLARK ST']

>>> [addr for addr in addresses if fnmatchcase(addr, '54[0-9][0-9] \*CLARK\*')]

['5412 N CLARK ST']

>>>

讨论

fnmatch() 函数匹配能力介于简单的字符串方法和强大的正则表达式之间。 如果在数据处理操作中只需要简单的通配符就能完成的时候，这通常是一个比较合理的方案。

### 2.4 字符串匹配和搜索

#### 2.4.1 问题

你想匹配或者搜索特定模式的文本

#### 2.4.2 解决方案

如果你想匹配的是字面字符串，那么你通常只需要调用基本字符串方法就行， 比如**str.find(), str.endswith() , str.startswith()** 或者类似的方法：

>>> text = 'yeah, but no, but yeah, but no, but yeah'

>>> # Exact match

>>> text == 'yeah'

False

>>> # Match at start or end

>>> text.startswith('yeah')

True

>>> text.endswith('no')

False

>>> # Search for the location of the first occurrence

>>> text.find('no')

10

>>>

对于复杂的匹配需要使用正则表达式和 re 模块。 为了解释正则表达式的基本原理，假设你想匹配数字格式的日期字符串比如 11/27/2012 ，你可以这样做：

import re

text = *'Today is 11/27/2012. PyCon starts 3/13/2013.'*

text1 = *"11/27/2012"*

text2 = *'Nov 27, 2012'*

pattern1 = re.compile(*r'\d{1,2}/\d{1,2}/\d{1,4}'*)

pattern2 = re.compile(*r'[a-zA-Z]+\s\d{1,2},\s\d{1,4}'*)

m1 = pattern1.match(text1)

m2 = pattern2.match(text2)

print m1.group()

print m2.group()

print pattern1.findall(text)

输出：

11/27/2012

Nov 27, 2012

['11/27/2012', '3/13/2013']

在定义正则式的时候，通常会利用括号去捕获分组。比如：

>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')

>>>

>>>

>>> m = datepat.match('11/27/2012')

>>> m.groups()

('11', '27', '2012')

>>> m.group(1)

'11'

>>> f = datepat.findall('11/27/2012 18/08/2017')

>>> f

[('11', '27', '2012'), ('18', '08', '2017')]

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，比如：

>>> m = datepat.finditer('11/27/2012 18/08/2017')

>>> m

<callable-iterator object at 0x0000000002E9C9E8>

>>>

>>> for m in datepat.finditer('11/27/2012 18/08/2017'):

print m

print m.groups()

<\_sre.SRE\_Match object at 0x0000000002DEF290>

('11', '27', '2012')

<\_sre.SRE\_Match object at 0x0000000002DEF1F8>

('18', '08', '2017')

>>>

### 2.5 字符串搜索和替换

#### 2.5.1 问题

你想在字符串中搜索和匹配指定的文本模式

#### 2.5.2 解决方案

对于简单的字面模式，直接使用 **str.repalce()**方法即可

>>> text = 'yeah, but no, but yeah, but no, but yeah'

>>> text.replace('yeah', 'yep')

'yep, but no, but yep, but no, but yep'

>>>

对于复杂的模式，请使用 re 模块中的**sub()** 函数。 为了说明这个，假设你想将形式为 11/27/201 的日期字符串改成 2012-11-27 。示例如下：

>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'

>>> import re

>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)

'Today is 2012-11-27. PyCon starts 2013-3-13.'

>>>

sub() 函数中的第一个参数是被匹配的模式，第二个参数是替换模式。**反斜杠数字比如 \3 指向前面模式的捕获组号。**

如果你打算用相同的模式做多次替换，考虑先编译它来提升性能。比如：

>>> import re

>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')

>>> datepat.sub(r'\3-\1-\2', text)

'Today is 2012-11-27. PyCon starts 2013-3-13.'

>>>

对于更加复杂的替换，可以传递一个替换回调函数来代替，比如：

>>> from calendar import month\_abbr

>>> def change\_date(m):

... mon\_name = month\_abbr[int(m.group(1))]

... return '{} {} {}'.format(m.group(2), mon\_name, m.group(3))

...

>>> datepat.sub(change\_date, text)

'Today is 27 Nov 2012. PyCon starts 13 Mar 2013.'

>>>

一个替换回调函数的参数是一个 match 对象，也就是 match() 或者 find() 返回的对象。 使用 group() 方法来提取特定的匹配部分。回调函数最后返回替换字符串。

如果除了替换后的结果外，你还想知道有多少替换发生了，可以使用 **re.subn()** 来代替。比如：

>>> newtext, n = datepat.subn(r'**\3-\1-\2**', text)

>>> newtext

'Today is 2012-11-27. PyCon starts 2013-3-13.'

>>> n

2

>>>

### 2.6 字符串忽略大小写的搜索替换

#### 2.6.1 问题

你需要以忽略大小写的方式搜索与替换文本字符串

#### 2.6.2 解决方案

为了在文本操作时忽略大小写，你需要在使用 re 模块的时候给这些操作提供 **re.IGNORECASE**标志参数。比如：

>>> text = 'UPPER PYTHON, lower python, Mixed Python'

>>> re.findall('python', text, flags=re.IGNORECASE)

['PYTHON', 'python', 'Python']

>>> re.sub('python', 'snake', text, flags=re.IGNORECASE)

'UPPER snake, lower snake, Mixed snake'

>>>

### 2.7 最短匹配模式

#### 2.7.1 问题

你正在试着用正则表达式匹配某个文本模式，但是它找到的是模式的最长可能匹配。 而你想修改它变成查找最短的可能匹配：

#### 2.7.2 解决方案

这个问题一般出现在需要匹配一对分隔符之间的文本的时候(比如引号包含的字符串)。 为了说明清楚，考虑如下的例子：

>>> str\_pat = re.compile(r'\"(.\*)\"')

>>> text1 = 'Computer says "no."'

>>> str\_pat.findall(text1)

['no.']

>>> text2 = 'Computer says "no." Phone says "yes."'

>>> str\_pat.findall(text2)

['no." Phone says "yes.']

>>>

在这个例子中，模式 r'\"(.\*)\"' 的意图是匹配被双引号包含的文本。 但是在正则表达式中\*操作符是贪婪的，因此匹配操作会查找最长的可能匹配。 于是在第二个例子中搜索 text2 的时候返回结果并不是我们想要的。

为了修正这个问题，可以**在模式中的\*操作符后面加上?修饰符**，就像这样：

>>> str\_pat = re.compile(r'\"(.\*?)\"')

>>> str\_pat.findall(text2)

['no.', 'yes.']

>>>

这样就使得匹配变成非贪婪模式，从而得到最短的匹配，也就是我们想要的结果。

 在一个模式字符串中，点(.)匹配除了换行外的任何字符。 然而，如果你将点(.)号放在开始与结束符(比如引号)之间的时候，那么匹配操作会查找符合模式的最长可能匹配。 这样通常会导致很多中间的被开始与结束符包含的文本被忽略掉，并最终被包含在匹配结果字符串中返回。 通过在 \* 或者 + 这样的操作符后面添加一个 ? 可以强制匹配算法改成寻找最短的可能匹配。

### 2.8 多行匹配模式

#### 2.8.1 问题

你正在试着使用正则表达式去匹配一大块的文本，而你需要跨越多行去匹配。

#### 2.8.2 解决方案

这个问题很典型的出现在当你用点(.)去匹配任意字符的时候，忘记了点(.)不能匹配换行符的事实。 比如，假设你想试着去匹配 C 语言分割的注释：

>>> comment = re.compile(r'/\\*(.\*?)\\*/')

>>> text1 = '/\* this is a comment \*/'

>>> text2 = '''/\* this is a

... multiline comment \*/

... '''

>>>

>>> comment.findall(text1)

[' this is a comment ']

>>> comment.findall(text2)

[]

>>>

为了修正这个问题，你可以修改模式字符串，增加对换行的支持。比如：

>>> comment = re.compile(r'/\\*((?:.|\n)\*?)\\*/')

>>> comment.findall(text2)

[' this is a\n multiline comment ']

>>>

在这个模式中，**(?:.|\n)**指定了一个非捕获组 (也就是它定义了一个仅仅用来做匹配，而不能通过单独捕获或者编号的组)。

re.compile() 函数接受一个标志参数叫 re.DOTALL ，在这里非常有用。 它可以让正则表达式中的点(.)匹配包括换行符在内的任意字符。比如：

>>> comment = re.compile(r'/\\*(.\*?)\\*/', re.DOTALL)

>>> comment.findall(text2)

[' this is a\n multiline comment ']

### 2.9 删除字符串中不需要的字符

#### 2.9.1 问题

你想去掉文本字符串开头，结尾或者中间不想要的字符，比如空白。

#### 2.9.2解决方案

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。比如：

>>> # Whitespace stripping

>>> s = ' hello world \n'

>>> s.strip()

'hello world'

>>> s.lstrip()

'hello world \n'

>>> s.rstrip()

' hello world'

>>>

>>> # Character stripping

>>> t = '-----hello====='

>>> t.lstrip('-')

'hello====='

>>> t.strip('-=')

'hello'

### 字符串对齐

#### 2.5.1问题

你想通过某种对齐方式来格式化字符串

#### 2.5.2解决方案

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。比如：

>>> text = 'Hello World'

>>> text.ljust(20)

'Hello World '

>>> text.rjust(20)

' Hello World'

>>> text.center(20)

' Hello World '

>>>

所有这些方法都能接受一个可选的填充字符。比如：

>>> text.rjust(20,'=')

'=========Hello World'

>>> text.center(20,'\*')

'\*\*\*\*Hello World\*\*\*\*\*'

>>>

## 类与对象

### 3.1 改变对象的字符串显示

#### 3.1.1问题

你想改变对象实例的打印或显示输出，让它们更具可读性。

#### 3.1.2解决方案

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

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)

\_\_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，跟我们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。比如：

>>> p = Pair(3, 4)

>>> p

Pair(3, 4) # \_\_repr\_\_() output

>>> print(p)

(3, 4) # \_\_str\_\_() output

我们在这里还演示了在格式化的时候怎样使用不同的字符串表现形式。 特别来讲，!r 格式化代码指明输出使用\_\_repr\_\_() 来代替默认的\_\_str\_\_() 。 你可以用前面的类来试着测试下：

>>> p = Pair(3, 4)

>>> print('p is {0!r}'.format(p))

p is Pair(3, 4)

>>> print('p is {0}'.format(p))

p is (3, 4)

>>>

如果 \_\_str\_\_() 没有被定义，那么就会使用 \_\_repr\_\_() 来代替输出。

### 3.2 自定义字符串的格式化

#### 3.2.1问题

你想通过 format()函数和字符串方法使得一个对象能支持自定义的格式化。

#### 3.2.2解决方案

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。例如：

\_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)

>>> d = Date(2012, 12, 21)

>>> format(d)

'2012-12-21'

>>> format(d, 'mdy')

'12/21/2012'

>>> 'The date is {:ymd}'.format(d) # ymd作为参数传给 \_\_format\_\_函数

'The date is 2012-12-21'

>>> 'The date is {:mdy}'.format(d) # mdy作为参数传给 \_\_format\_\_函数

'The date is 12/21/2012'

>>>

### 3.3 让对象支持上下文管理协议

#### 3.3.1问题

你想让你的对象支持上下文管理协议(with 语句)。

#### 3.3.2解决方案

为了让一个对象兼容 with语句，你需要实现 \_\_enter\_\_() 和 \_\_exit\_\_() 方法。 例如，考虑如下的一个类，它能为我们创建一个网络连接：

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.sock = None

def \_\_enter\_\_(self):

if self.sock is not None:

raise RuntimeError('Already connected')

self.sock = socket(self.family, self.type)

self.sock.connect(self.address)

return self.sock

def \_\_exit\_\_(self, exc\_ty, exc\_val, tb):

self.sock.close()

self.sock = None

这个类的关键特点在于它表示了一个网络连接，但是初始化的时候并不会做任何事情(比如它并没有建立一个连接)。 连接的建立和关闭是使用 with语句自动完成的，例如：

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''))

# conn.\_\_exit\_\_() executes: connection closed

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。

### 3.4 创建大量对象时节省内存方法

#### 3.4.1问题

你的程序要创建大量(可能上百万)的对象，导致占用很大的内存。

#### 3.4.2解决方案

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。比如：

class Date:

\_\_slots\_\_ = ['year', 'month', 'day']

def \_\_init\_\_(self, year, month, day):

self.year = year

self.month = month

self.day = day

当你定义 \_\_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我们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。

 Python 的很多特性都依赖于普通的基于字典的实现。 另外，定义了 slots 后的类不再支持一些普通类特性了，比如多继承。 大多数情况下，你应该只在那些经常被使用到的用作数据结构的类上定义 slots (比如在程序中需要创建某个类的几百万个实例对象)。

关于 \_\_slots\_\_ 的一个常见误区是它可以作为一个封装工具来防止用户给实例增加新的属性。 尽管使用 slots 可以达到这样的目的，但是这个并不是它的初衷。 \_\_slots\_\_ 更多的是用来作为一个内存优化工具。

3.5

## 4 函数

### 4.1 可接受任意数量参数的函数

#### 4.1.1问题

你想构造一个可接受任意数量参数的函数。

#### 4.2.2解决方案

为了能让一个函数接受任意数量的位置参数，可以使用一个\*参数。例如：

def **avg**(first, \*args):

return (first + sum(args)) / (1 + len(args))

print avg(1, 2) # 注意：python2中结果为1

print avg(2, 3.0, 4.0)

为了接受任意数量的关键字参数，使用一个以\*\*开头的参数。比如：

import cgi

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 = cgi.escape(value))

return element

# Example

# Creates '<item size="large" quantity="6">Albatross</item>'

make\_element(*'item'*, *'Albatross'*, size=*'large'*, quantity=6)

# Creates '<p>&lt;spam&gt;</p>'

make\_element(*'p'*, *'<spam>'*)

如果你还希望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\*\*。比如：

def **anyargs**(\*args, \*\*kwargs):

print(args) # A tuple

print(kwargs) # A dict

使用这个函数时，所有位置参数会被放到 args 元组中，所有关键字参数会被放到字典 kwargs 中。

### 4.2 只接受关键字参数的函数（python3.x才有）

#### 4.2.1问题

你希望函数的某些参数强制使用关键字参数传递

#### 4.2.2解决方案

将强制关键字参数放到某个参数或者当个后面就能达到这种效果。比如：

def **recv**(maxsize, \*, block):

*'Receives a message'*

pass

recv(1024, True) # TypeError

recv(1024, block=True) # Ok

利用这种技术，我们还能在接受任意多个位置参数的函数中指定关键字参数。比如：

def **mininum**(clip=None, \*values):

m = min(values)

if clip is not None:

m = clip if clip > m else m

return m

#minimum(1, 5, 2, -5, 10) # Returns -5

mininum(1, 5, 2, -5, 10, clip=0) # Returns 0

很多情况下，使用强制关键字参数会比使用位置参数表意更加清晰，程序也更加具有可读性。**Python3.x中才可用，python2.x不支持**。

### 4.3 给函数参数增加元信息（Python3.x才有）

#### 4.3.1问题

你写好了一个函数，然后想为这个函数的参数增加一些额外的信息，这样的话其他使用者就能清楚的知道这个函数应该怎么使用。

#### 4.3.2解决方案

使用函数参数注解是一个很好的办法，它能提示程序员应该怎样正确使用这个函数。 例如，下面有一个被注解了的函数：

def add(x:int, y:int) -> int:

return x + y

python 解释器不会对这些注解添加任何的语义。它们不会被类型检查，运行时跟没有加注解之前的效果也没有任何差距。 然而，对于那些阅读源码的人来讲就很有帮助啦。第三方工具和框架可能会对这些注解添加语义。同时它们也会出现在文档中。

>>> help(add)

Help on function add in module \_\_main\_\_:

add(x: int, y: int) -> int

>>>

尽管你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。

讨论

函数注解只存储在函数的 \_\_annotations\_\_ 属性中。例如：

>>> add.\_\_annotations\_\_

{'y': <class 'int'>, 'return': <class 'int'>, 'x': <class 'int'>}

尽管注解的使用方法可能有很多种，但是它们的主要用途还是文档。 因为 python 并没有类型声明，通常来讲仅仅通过阅读源码很难知道应该传递什么样的参数给这个函数。 这时候使用注解就能给程序员更多的提示，让他们可以争取的使用函数。

### 4.4 返回多个值的函数

#### 4.4.1问题

你希望构造一个可以返回多个值的函数

#### 4.4.2解决方案

为了能返回多个值，函数直接 return 一个元组就行了。例如：

def **myfun**():

return 1, 2, 3

a, b, c = myfun()

### 4.5定义有默认参数的函数

#### 4.5.1问题

你想定义一个函数或者方法，它的一个或多个参数是可选的并且有一个默认值。

#### 4.5.2解决方案

定义一个有可选参数的函数是非常简单的，直接在函数定义中给参数指定一个默认值，并放到参数列表最后就行了。

如果默认参数是一个可修改的容器比如一个列表、集合或者字典，可以使用 None 作为默认值，就像下面这样：

def **spam1**(a, b=None):

if b is None:

b = []

如果你并不想提供一个默认值，而是想仅仅测试下某个默认参数是不是有传递进来，可以像下面这样写：

\_no\_value = object()

def **spam2**(a, b=\_no\_value):

if b is \_no\_value:

print(*'No b value supplied'*)

spam2(1) # No b value supplied

spam2(1, 2) # b = 2

spam2(1, None) # b = None

**在函数定义的时候就已经确定了它的默认值了。**

**其次，默认参数的值应该是不可变的对象，比如 None、True、False、数字或字符串。**

**千万不要像下面这样写代码：No**

def **spam4**(a, b=[]):

print b

return b

x = spam4(1)

print x # []

x.append(99) # [99]

x.append(*'Yow！'*) # [99, 'Yow!']

spam4(1) # Modified list gets returned!

在测试 None 值时使用 is 操作符是很重要的，也是这种方案的关键点。 有时候大家会犯下下面这样的错误：

def spam(a, b=None):

if not b: # NO! Use 'b is None' instead

b = []

...

这么写的问题在于尽管 None 值确实是被当成 False， 但是还有其他的对象(比如长度为0的字符串、列表、元组、字典等)都会被当做 False。 因此，上面的代码会误将一些其他输入也当成是没有输入。比如：

>>> spam(1) # OK

>>> x = []

>>> spam(1, x) # Silent error. x value overwritten by default

>>> spam(1, 0) # Silent error. 0 ignored

>>> spam(1, '') # Silent error. '' ignored

>>>

最后一个问题比较微妙，那就是一个函数需要测试某个可选参数是否被使用者传递进来。 这时候需要小心的是你不能用某个默认值比如 None、 0或者 False值来测试用户提供的值(因为这些值都是合法的值，是可能被用户传递进来的)。 因此，你需要其他的解决方案了。

为了解决这个问题，你可以创建一个独一无二的私有对象实例，就像上面的 \_no\_value 变量那样。 在函数里面，你可以通过检查被传递参数值跟这个实例是否一样来判断。 这里的思路是用户不可能去传递这个 \_no\_value 实例作为输入。 因此，这里通过检查这个值就能确定某个参数是否被传递进来了。

### 4.6定义匿名或内联函数

#### 4.6.1 问题

你想为 sort()操作创建一个很短的回调函数，但又不想用def去写一个单行函数， 而是希望通过某个快捷方式以内联方式来创建这个函数。

#### 4.6.2解决方案

当一些函数很简单，仅仅只是计算一个表达式的值的时候，就可以使用 lambda 表达式来代替了。比如：

names = [

*'David Beazley'*, *'Brian Jones'*,

*'Raymond Hettinger'*, *'Ned Batchelder'*

]

# ['Ned Batchelder', 'David Beazley', 'Raymond Hettinger', 'Brian Jones']

print sorted(names, key = lambda name: name.split()[-1].lower())

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。

### 4.7 匿名函数捕获变量值

#### 4.7.1问题

你用 lambda 定义了一个匿名函数，并想在定义时捕获到某些变量的值。

#### 4.7.2解决方案

先看下下面代码的效果：

>>> x = 10

>>> a = lambda y: x + y

>>> x = 20

>>> b = lambda y: x + y

>>>

结果：

>>> a(10)

30

>>> b(10)

30

>>>

这其中的奥妙在于 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，这跟函数的默认值参数定义是不同的。 因此，在调用这个 lambda 表达式的时候，x 的值是执行时的值。例如：

>>> x = 15

>>> a(10)

25

>>> x = 3

>>> a(10)

13

>>>

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，就像下面这样：

>>> x = 10

>>> a = lambda y, x=x: x + y

>>> x = 20

>>> b = lambda y, x=x: x + y

>>> a(10)

20

>>> b(10)

30

>>>

**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：**

funcs = [lambda x: x + n for n in range(5)]

# n为自由值，没有在函数定义的时候绑定，运行是时n的值为迭代的最后一个值结果为 4 4 4 4 4

for f in funcs:

print f(0)

funcs = [lambda x, n=n: x + n for n in range(5)]

# 结果为0 1 2 3 4

for f in funcs:

print f(0)

### 4.8 减少可调用对象的参数个数

#### 4.8.1 问题

你有一个被其他 python 代码使用的 callable 对象，可能是一个回调函数或者是一个处理器， 但是它的参数太多了，导致调用时出错。

#### 4.8.2解决方案

如果需要减少某个函数的参数个数，你可以使用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。 为了演示清楚，假设你有下面这样的函数：

from functools import partial

def **spam**(a, b, c, d):

print a, b, c, d

s1 = partial(spam, 1)

s1(2, 3, 4) # 1 2 3 4

s2 = partial(spam, 1, d=42)

s2(2, 3) # 1 2 3 42

partial() 固定某些参数并返回一个新的 callable 对象。这个新的 callable 接受未赋值的参数， 然后跟之前已经赋值过的参数合并起来，最后将所有参数传递给原始函数。

import math

def **distance**(p1, p2):

x1, y1 = p1

x2, y2 = p2

return math.hypot(x2-x1, y2-y1)

points = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

pt = (4, 3)

points.sort(key = partial(distance, pt))

# [(3, 4), (1, 2), (5, 6), (7, 8)]

print points

### 4.9将单方法的类转换为函数

#### 4.9.1问题

你有一个除 \_\_init\_\_()方法外只定义了一个方法的类。为了简化代码，你想将它转换成一个函数。

#### 4.9.2解决方案

大多数情况下，可以使用闭包来将单个方法的类转换成函数。 举个例子，下面示例中的类允许使用者根据某个模板方案来获取到 URL 链接地址。

from urllib.request import urlopen

class UrlTemplate:

def \_\_init\_\_(self, template):

self.template = template

def open(self, \*\*kwargs):

return urlopen(self.template.format\_map(kwargs))

# Example use. Download stock data from yahoo

yahoo = UrlTemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f={fields}')

for line in yahoo.open(names='IBM,AAPL,FB', fields='sl1c1v'):

print(line.decode('utf-8'))

这个类可以被一个更简单的函数来代替：

def urltemplate(template):

def opener(\*\*kwargs):

return urlopen(template.format\_map(kwargs))

return opener

# Example use

yahoo = urltemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f={fields}')

for line in yahoo(names='IBM,AAPL,FB', fields='sl1c1v'):

print(line.decode('utf-8'))

任何时候只要你碰到需要给某个函数增加额外的状态信息的问题，都可以考虑使用闭包。 相比将你的函数转换成一个类而言，闭包通常是一种更加简洁和优雅的方案。

### 4.10带额外状态信息的回调函数

#### 4.10.1问题

你的代码中需要依赖到回调函数的使用(比如事件处理器、等待后台任务完成后的回调等)， 并且你还需要让回调函数拥有额外的状态值，以便在它的内部使用到。

#### 4.10.2解决方案

这一小节主要讨论的是那些出现在很多函数库和框架中的回调函数的使用——特别是跟异步处理有关的。 为了演示与测试，我们先定义如下一个需要调用回调函数的函数：

## 迭代器与生成器

### 5.1 手动遍历迭代器

#### 5.1.1 问题

你想遍历一个可迭代对象中的所有元素，但是却不想使用 for 循环。

#### 5.1.2解决方案

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。 比如，下面的例子手动读取一个文件中的所有行：

def **manual\_iter**():

with open(*'1\_1.py'*) as f:

try:

while True:

line = next(f)

print line

except StopIteration:

pass

通常来讲， **StopIteration** 用来指示迭代的结尾。

### 5.2 代理迭代

#### 5.2.1问题

你构建了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。 你想直接在你的这个新容器对象上执行迭代操作。

#### 5.2.2解决方案

实际上你只需要定义一个 \_\_iter\_\_() 方法，将迭代操作代理到容器内部的对象上去。比如：

# -\*- coding:utf-8 -\*-

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

 \_\_iter\_\_()方法只是简单的将迭代请求传递给内部的 \_children属性。

Python 的迭代器协议需要\_\_iter\_\_()方法返回一个实现了 \_\_next\_\_()方法的迭代器对象。 如果你只是迭代遍历其他容器的内容，你无须担心底层是怎样实现的。你所要做的只是传递迭代请求既可。

### 5.3 使用生成器创建新的迭代模式

#### 5.3.1 问题

你想实现一个自定义迭代模式，跟普通的内置函数比如 range(), reversed()不一样。

#### 5.3.2解决方案

如果你想实现一种新的迭代模式，使用一个生成器函数来定义它。 下面是一个生产某个范围内浮点数的生成器：

def **frange**(start, stop, increment):

x = start

while x < stop:

yield x

x += increment

for n in frange(0, 4, 0.5):

print n

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。

def **frange**(start, stop, increment):

x = start

while x < stop:

yield x

x += increment

for n in frange(0, 4, 0.5):

print n

def **countdown**(n):

print *"Starting to count from"*, n

while n > 0:

yield n

n -= 1

print *'Done'*

c = countdown(3)

>>> c

<generator object countdown at 0x1006a0af0>

>>> # Run to first yield and emit a value

>>> next(c)

Starting to count from 3

3

>>> # Run to the next yield

>>> next(c)

2

>>> # Run to next yield

>>> next(c)

1

>>> # Run to next yield (iteration stops)

>>> next(c)

Done!

Traceback (most recent call last):

File *"<stdin>"*, line 1, in <module>

StopIteration

>>>

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。

### 5.4 实现迭代协议

#### 5.4.1 问题

你想构建一个能支持迭代操作的自定义对象，并希望找到一个能实现迭代协议的简单方法。

#### 5.4.2解决方案

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。 在4.2小节中，使用 Node 类来表示树形数据结构。你可能想实现一个以深度优先方式遍历树形节点的生成器。 下面是代码示例：

# -\*- coding:utf-8 -\*-

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*:

for item in c.depth\_first():

yield item

# Example

if \_\_name\_\_ == *"\_\_main\_\_"*:

root = Node(0)

child1 = Node(1)

child2 = Node(2)

root.add\_child(child1)

root.add\_child(child2)

subchild1 = Node(3)

subchild2 = Node(4)

child1.add\_child(subchild1)

child1.add\_child(Node(5))

child2.add\_child(subchild2)

subchild1.add\_child(Node(6))

subchild2.add\_child(Node(7))

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。

### 5.5 反向迭代

#### 5.5.1 问题

你想反方向迭代一个序列

#### 5.5.2解决方案

使用内置的 reversed()函数，比如：

a = [1, 2, 3, 4]

# 4 3 2 1

for x in reversed(a):

print x

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，比如：

class **CountDown**:

def **\_\_init\_\_**(*self*, start):

*self*.start = start

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 CountDown(30):

print rr

for rr in reversed(CountDown(30)):

print rr

### 5.6 带有外部状态的生成器函数

#### 5.6.1 问题

你想定义一个生成器函数，但是它会调用某个你想暴露给用户使用的外部状态值。

#### 5.6.2 解决方案

如果你想让你的生成器暴露外部状态给用户， 别忘了你可以简单的将它实现为一个类，然后把生成器函数放到 \_\_iter\_\_() 方法中过去。比如：

# -\*- coding:utf-8 -\*-

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(*'5\_1.py'*) as f:

lines = LineHistory(f)

for line in lines:

if *'print'* in line:

for lineno, hline in lines.history:

print *'{}:{}'*.format(lineno, hline)

### 5.7 迭代器切片

#### 5.7.1问题

你想得到一个由迭代器生成的切片对象，但是标准切片操作并不能做到。

#### 5.7.2解决方案

**函数 itertools.islice()正好适用于在迭代器和生成器上做切片操作**。比如：

# -\*- coding:utf-8 -\*-

import itertools

def **count**(n):

while True:

yield n

n += 1

c = count(0)

for x in itertools.islice(c, 10, 20):

print x

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。

### 5.8 跳过可迭代对象

#### 5.8.1 问题

你想遍历一个可迭代对象，但是它开始的某些元素你并不感兴趣，想跳过它们。

#### 5.8.2解决方案

itertools 模块中有一些函数可以完成这个任务。 首先介绍的是 itertools.dropwhile()函数。使用时，你给它传递一个函数对象和一个可迭代对象。 它会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

# -\*- coding:utf-8 -\*-

from itertools import dropwhile

with open(*'5\_4.py'*) as f:

for line in dropwhile(lambda line: line.startswith(*'#'*), f):

print line

### 5.9 排列组合的迭代

#### 5.9.1 问题

你想迭代遍历一个集合中元素的所有可能的排列或组合

#### 5.9.2解决方案

itertools 模块提供了三个函数来解决这类问题。 其中一个是 itertools.permutations()， 它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，比如：

from itertools import permutations

items = [*'a'*, *'b'*, *'c'*]

# Result:

# ('a', 'b', 'c')

# ('a', 'c', 'b')

# ('b', 'a', 'c')

# ('b', 'c', 'a')

# ('c', 'a', 'b')

# ('c', 'b', 'a')

for p in permutations(items):

print p

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。就像这样：

# Result:

# ('a', 'b')

# ('a', 'c')

# ('b', 'a')

# ('b', 'c')

# ('c', 'a')

# ('c', 'b')

for p in permutations(items, 2):

print p

用**itertools.combinations()**可得到输入集合中元素的所有的组合。比如：

# Result:

# ('a', 'b', 'c')

for c in combinations(items, 3):

print c

# Result:

# ('a', 'b')

# ('a', 'c')

# ('b', 'c')

for c in combinations(items, 2):

print c

 而函数 itertools.combinations\_with\_replacement()允许同一个元素被选择多次，比如：

>>> for c in combinations\_with\_replacement(items, 3):

... print(c)

...

('a', 'a', 'a')

('a', 'a', 'b')

('a', 'a', 'c')

('a', 'b', 'b')

('a', 'b', 'c')

('a', 'c', 'c')

('b', 'b', 'b')

('b', 'b', 'c')

('b', 'c', 'c')

('c', 'c', 'c')

>>>

### 5.10 创建数据处理管道

#### 5.10.1问题

你想以数据管道(类似 Unix 管道)的方式迭代处理数据。 比如，你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。

#### 5.10.2解决方案

生成器函数是一个实现管道机制的好办法。 为了演示，假定你要处理一个非常大的日志文件目录：

foo/

access-log-012007.gz

access-log-022007.gz

access-log-032007.gz

...

access-log-012008

bar/

access-log-092007.bz2

...

access-log-022008

假设每个日志文件包含这样的数据：

124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200 71

210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 11875

210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 404 369

61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 304 -

为了处理这些文件，你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。就像这样：

# -\*- coding:utf-8 -\*-

import os

import fnmatch

import gzip

import bz2

import re

def **gen\_find**(filepat, top):

*'''*

*Find all filenames in a directory tree that match a shell wildcard pattern*

*'''*

for path, dirlist, filelist in os.walk(top):

# fnmatch检查文件是否匹配模式：fnmatch.fnmatch(filename,pattern),例如fnmatch.fnmatch('tlie.py','\*.py')

# filter过滤或筛选符合模式的列表：fnmatch.filter(names, pat)

for name in fnmatch.filter(filelist, filepat):

yield os.path.join(path, name)

def **gen\_opener**(filenames):

*'''*

*Open a sequence of filenames one at a time producing a file object.*

*The file is closed immediately when proceeding to the next iteration.*

*'''*

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):

*'''*

*Chain a sequence of iterators together into a single sequence.*

*'''*

for it in iterators:

for item in it:

yield item

def **gen\_grep**(pattern, lines):

*'''*

*Look for a regex pattern in a sequence of lines*

*'''*

pat = re.compile(pattern)

for line in lines:

if pat.search(line):

yield line

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。

### 5.11 展开嵌套的序列

#### 5.11.1 问题

你想将一个多层嵌套的序列展开成一个单层列表

#### 5.11.2解决方案

可以写一个包含 yield from 语句的递归生成器来轻松解决这个问题。比如：

# -\*- coding:utf-8 -\*-

from collections import Iterable

def **flatten**(items, ignnore\_types=(str, bytes)):

for x in items:

if isinstance(x, Iterable) and not isinstance(x, ignnore\_types):

# Python3.x写为:yield from flatten(x)

for y in flatten(x):

yield y

else:

yield x

items = [1, 2, [3, 4, [5, 6], 7], 8]

for x in flatten(items):

print x

# [1, 2, 3, 4, 5, 6, 7, 8]

print list(flatten(items))

在上面代码中， 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。

额外的参数 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。

### 5.12 顺序迭代合并后的排序迭代对象

#### 5.12.1 问题

你有一系列排序序列，想将它们合并后得到一个排序序列并在上面迭代遍历。

#### 5.12.2解决方案

heapq.merge() 函数可以帮你解决这个问题。比如：

# -\*- coding:utf-8 -\*-

import heapq

a = [1, 4, 7, 10]

b = [2, 4, 6, 11]

# [1, 2, 4, 4, 6, 7, 10, 11]

print list(heapq.merge(a, b))

 heapq.merge()需要所有输入序列必须是排过序的。

### 5.13 迭代器代替 while 无限循环

#### 5.13.1问题

你在代码中使用while循环来迭代处理数据，因为它需要调用某个函数或者和一般迭代模式不同的测试条件。 能不能用迭代器来重写这个循环呢？

#### 5.13.2解决方案

一个常见的 IO 操作程序可能会想下面这样：

CHUNKSIZE = 8192

def reader(s):

while True:

data = s.recv(CHUNKSIZE)

if data == b'':

break

process\_data(data)

这种代码通常可以使用 iter()来代替，如下所示：

def reader2(s):

for chunk in iter(lambda: s.recv(CHUNKSIZE), b''):

pass

# process\_data(data)

## 第 6章 文件与IO

### 6.1 读写文本数据

#### 6.1.1问题

你需要读写各种不同编码的文本数据，比如 ASCII，UTF-8 或 UTF-16 编码等。

#### 6.1.2解决方案

使用带有**rt**模式的 open()函数读取文本文件。

类似的，为了写入一个文本文件，使用带有 **wt**模式的 open() 函数， 如果之前文件内容存在则清除并覆盖掉。

如果是在已存在文件中添加内容，使用模式为**at**的open()函数。

# -\*- coding:utf-8 -\*-

#w,r,wt,rt都是python里面文件操作的模式。

#

#w是写模式，r是读模式。

#

#t是windows平台特有的所谓text mode(文本模式）,区别在于会自动识别windows平台的换行符。

#类Unix平台的换行符是\n，而windows平台用的是\r\n两个ASCII字符来表示换行，python内部采用的是\n来表示换行符。

#rt模式下，python在读取文本时会自动把\r\n转换成\n.

#wt模式下，Python写文件时会用\r\n来表示换行。

with open(*'somefile.txt'*, *'wt'*) as f:

text1 = *"""你好！*

*好久不见*

*你还好吗？*

*"""*

text2 = *"""是啊！好久不见！*

*你呢？*

*"""*

f.write(text1)

f.write(text2)

with open(*'somefile.txt'*, *'rt'*) as f:

data = f.read()

print data

with open(*'somefile.txt'*, *'at'*) as f:

f.write(*'Hello \r\n'*)

## 模块与包

模块与包是任何大型程序的核心，就连 Python 安装程序本身也是一个包。

### 7.1 构建一个模块的层级包

#### 7.1.1问题

你想将你的代码组织成由很多分层模块构成的包。

#### 7.1.2解决方案

封装成包是很简单的。在文件系统上组织你的代码，并确保每个目录都定义了一个init.py 文件。 例如：

graphics/

\_\_init\_\_.py

primitive/

\_\_init\_\_.py

line.py

fill.py

text.py

formats/

\_\_init\_\_.py

png.py

jpg.py

一旦你做到了这一点，你应该能够执行各种 import 语句，如下：

import graphics.primitive.line

from graphics.primitive import line

import graphics.formats.jpg as jpg

**文件init.py 的目的是要包含不同运行级别的包的可选的初始化代码。** 举个例子，如果你执行了语句 import graphics， 文件 graphics/init.py 将被导入,建立 graphics 命名空间的内容。

绝大部分时候让init.py 空着就好。但是有些情况下可能包含代码。 举个例子，init.py 能够用来自动加载子模块:

# graphics/formats/\_\_init\_\_.py

from . import jpg

from . import png

像这样一个文件,用户可以仅仅通过 import grahpics.formats 来代替 import graphics.formats.jpg 以及 import graphics.formats.png。

### 7.2 控制模块被全部导入的内容

#### 7.2.1问题

当使用’from module import \*‘ 语句时，希望对从模块或包导出的符号进行精确控制。

#### 7.2.2解决方案

在你的模块中定义一个变量 all 来明确地列出需要导出的内容。

举个例子:

# somemodule.py

def spam():

pass

def grok():

pass

blah = 42

# Only export 'spam' and 'grok'

\_\_all\_\_ = ['spam', 'grok']

尽管强烈反对使用 ‘from module import \*‘, 但是在定义了大量变量名的模块中频繁使用。 如果你不做任何事, 这样的导入将会导入所有不以下划线开头的。 另一方面,如果定义了 all , 那么只有被列举出的东西会被导出。

如果你将 all 定义成一个空列表, 没有东西将被导出。 如果 all 包含未定义的名字, 在导入时引起 AttributeError。

### 7.3 使用相对路径名导入包中子模块

#### 7.3.1问题

将代码组织成包,想用 import 语句从另一个包名没有硬编码过的包的中导入子模块。

#### 7.3.2解决方案

使用包的相对导入，使一个的模块导入同一个包的另一个模块 举个例子，假设在你的文件系统上有 mypackage 包，组织如下：

mypackage/

\_\_init\_\_.py

A/

\_\_init\_\_.py

spam.py

grok.py

B/

\_\_init\_\_.py

bar.py

如果模块 mypackage.A.spam 要导入同目录下的模块 grok，它应该包括的 import 语句如下：

# mypackage/A/spam.py

from . import grok

如果模块 mypackage.A.spam 要导入不同目录下的模块 B.bar，它应该使用的 import 语句如下：

# mypackage/A/spam.py

from ..B import bar

两个 import 语句都没包含顶层包名，而是使用了 spam.py 的相对路径。

import 语句的 . 和 ..看起来很滑稽, 但它指定目录名.为当前目录，..B 为目录../B。这种语法只适用于 import。 举个例子：

from . import grok # OK

import .grok # ERROR

尽管使用相对导入看起来像是浏览文件系统，但是不能到定义包的目录之外。也就是说，使用点的这种模式从不是包的目录中导入将会引发错误。

最后，相对导入只适用于在合适的包中的模块。尤其是在顶层的脚本的简单模块中，它们将不起作用。如果包的部分被作为脚本直接执行，那它们将不起作用 例如：

% python3 mypackage/A/spam.py # Relative imports fail

另一方面，如果你使用 Python 的-m 选项来执行先前的脚本，相对导入将会正确运行。 例如：

% python3 -m mypackage.A.spam # Relative imports work

**但是 存在相对导入语句的模块，不能直接运行，否则会有异常：**

|  |  |
| --- | --- |
| **1** | **ValueError: Attempted relative import in non-package** |

**这是什么原因呢？我们需要先来了解下导入模块时的一些规则：**

**在没有明确指定包结构的情况下，Python 是根据 \_\_name\_\_ 来决定一个模块在包中的结构的，如果是 \_\_main\_\_ 则它本身是顶层模块，没有包结构，如果是A.B.C 结构，那么顶层模块是 A。基本上遵循这样的原则：**

* **如果是绝对导入，一个模块只能导入自身的子模块或和它的顶层模块同级别的模块及其子模块**
* **如果是相对导入，一个模块必须有包结构且只能导入它的顶层模块内部的模块**

如果一个模块被直接运行，则它自己为顶层模块，不存在层次结构，所以找不到其他的相对路径。

### 7.4将模块分割成多个文件

#### 7.4.1问题

你想将一个模块分割成多个文件。但是你不想将分离的文件统一成一个逻辑模块时使已有的代码遭到破坏。

#### 7.4.2解决方案

程序模块可以通过变成包来分割成多个独立的文件。考虑下下面简单的模块：

# mymodule.py

class A:

def spam(self):

print('A.spam')

class B(A):

def bar(self):

print('B.bar')

假设你想 mymodule.py 分为两个文件，每个定义的一个类。要做到这一点，首先用 mymodule 目录来替换文件 mymodule.py。 这这个目录下，创建以下文件：

mymodule/

\_\_init\_\_.py

a.py

b.py

在 a.py 文件中插入以下代码：

# a.py

class A:

def spam(self):

print('A.spam')

在 b.py 文件中插入以下代码：

# b.py

from .a import A

class B(A):

def bar(self):

print('B.bar')

最后，在 ****init****.py 中，将2个文件粘合在一起：

# \_\_init\_\_.py

from .a import A

from .b import B

如果按照这些步骤，所产生的包 MyModule 将作为一个单一的逻辑模块：

>>> import mymodule

>>> a = mymodule.A()

>>> a.spam()

A.spam

>>> b = mymodule.B()

>>> b.bar()

B.bar

>>>

### 讨论

在这个章节中的主要问题是一个设计问题，不管你是否希望用户使用很多小模块或只是一个模块。举个例子，在一个大型的代码库中，你可以将这一切都分割成独立的文件，让用户使用大量的 import 语句，就像这样：

from mymodule.a import A

from mymodule.b import B

...

这样能工作，但这让用户承受更多的负担，用户要知道不同的部分位于何处。通常情况下，将这些统一起来，使用一条 import 将更加容易，就像这样：

from mymodule import A, B

对后者而言，让 mymodule 成为一个大的源文件是最常见的。但是，这一章节展示了如何合并多个文件合并成一个单一的逻辑命名空间。 这样做的关键是创建一个包目录，使用 ****init****.py 文件来将每部分粘合在一起。

当一个模块被分割，你需要特别注意交叉引用的文件名。举个例子，在这一章节中，B类需要访问A类作为基类。用包的相对导入 from .a import A 来获取。

整个章节都使用包的相对导入来避免将顶层模块名硬编码到源代码中。这使得重命名模块或者将它移动到别的位置更容易。（见10.3小节）

作为这一章节的延伸，将介绍延迟导入。如图所示，****init****.py 文件一次导入所有必需的组件的。但是对于一个很大的模块，可能你只想组件在需要时被加载。 要做到这一点，****init****.py 有细微的变化：

# \_\_init\_\_.py

def A():

from .a import A

return A()

def B():

from .b import B

return B()

在这个版本中，类 A 和类 B 被替换为在第一次访问时加载所需的类的函数。对于用户，这看起来不会有太大的不同。 例如：

>>> import mymodule

>>> a = mymodule.A()

>>> a.spam()

A.spam

>>>

延迟加载的主要缺点是继承和类型检查可能会中断。你可能会稍微改变你的代码，例如:

if isinstance(x, mymodule.A): # Error

...

if isinstance(x, mymodule.a.A): # Ok

...

延迟加载的真实例子, 见标准库 multiprocessing/****init****.py 的源码.

### 7.5 运行目录或压缩文件

#### 7.5.1问题

您有已经一个复杂的脚本到涉及多个文件的应用程序。你想有一些简单的方法让用户运行程序。

#### 7.5.2解决方案

如果你的应用程序已经有多个文件，你可以把你的应用程序放进它自己的目录并添加一个****main****.py 文件。 举个例子，你可以像这样创建目录：

myapplication/

spam.py

bar.py

grok.py

\_\_main\_\_.py

如果****main****.py 存在，你可以简单地在顶级目录运行 Python 解释器：

bash % python3 myapplication

解释器将执行****main****.py 文件作为主程序。

如果你将你的代码打包成 zip 文件，这种技术同样也适用，举个例子：

bash % ls

spam.py bar.py grok.py \_\_main\_\_.py

bash % zip -r myapp.zip \*.py

bash % python3 myapp.zip

... output from \_\_main\_\_.py ...

### 讨论

创建一个目录或 zip 文件并添加****main****.py 文件来将一个更大的 Python 应用打包是可行的。这和作为标准库被安装到 Python 库的代码包是有一点区别的。相反，这只是让别人执行的代码包。

由于目录和 zip 文件与正常文件有一点不同，你可能还需要增加一个 shell 脚本，使执行更加容易。例如，如果代码文件名为 myapp.zip，你可以创建这样一个顶级脚本：

#!/usr/bin/env python3 /usr/local/bin/myapp.zip