4.1 手动遍历迭代器¶

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。

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('/etc/passwd') as f:
        try:
            while True:
                line = next(f)
                print(line, end='')
        except StopIteration:
            pass
```

通常来讲， `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` 。 下面是示例：

```
with open('/etc/passwd') as f:
    while True:
        line = next(f, None)
        if line is None:
            break
        print(line, end='')
```

讨论¶

大多数情况下，我们会使用 `for` 循环语句用来遍历一个可迭代对象。 但是，偶尔也需要对迭代做更加精确的控制，这时候了解底层迭代机制就显得尤为重要了。

下面的交互示例向我们演示了迭代期间所发生的基本细节：

```
>>> items = [1, 2, 3]
>>> # Get the iterator
>>> it = iter(items) # Invokes items.__iter__()
>>> # Run the iterator
>>> next(it) # Invokes it.__next__()
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
StopIteration
>>>
```

本章接下来几小节会更深入的讲解迭代相关技术，前提是你先要理解基本的迭代协议机制。 所以确保你已经把这章的内容牢牢记在心中。

4.10 序列上索引值迭代¶

问题¶

你想在迭代一个序列的同时跟踪正在被处理的元素索引。

解决方案¶

内置的 `enumerate()` 函数可以很好的解决这个问题：

```
>>> my_list = ['a', 'b', 'c']
>>> for idx, val in enumerate(my_list):
...     print(idx, val)
...
0 a
1 b
2 c
```

为了按传统行号输出(行号从1开始)，你可以传递一个开始参数：

```
>>> my_list = ['a', 'b', 'c']
>>> for idx, val in enumerate(my_list, 1):
...     print(idx, val)
...
1 a
2 b
3 c
```

这种情况在你遍历文件时想在错误消息中使用行号定位时候非常有用：

```
def parse_data(filename):
    with open(filename, 'rt') as f:
        for lineno, line in enumerate(f, 1):
            fields = line.split()
            try:
                count = int(fields[1])
                ...
            except ValueError as e:
                print('Line {}: Parse error: {}'.format(lineno, e))
```

`enumerate()` 对于跟踪某些值在列表中出现的位置是很有用的。 所以，如果你想将一个文件中出现的单词映射到它出现的行号上去，可以很容易的利用 `enumerate()` 来完成：

```
word_summary = defaultdict(list)

with open('myfile.txt', 'r') as f:
    lines = f.readlines()

for idx, line in enumerate(lines):
    # Create a list of words in current line
    words = [w.strip().lower() for w in line.split()]
    for word in words:
        word_summary[word].append(idx)
```

如果你处理完文件后打印 `word_summary` ，会发现它是一个字典(准确来讲是一个 `defaultdict`)， 对于每个单词有一个 `key` ，每个 `key` 对应的值是一个由这个单词出现的行号组成的列表。 如果某个单词在一行中出现过两次，那么这个行号也会出现两次， 同时也可以作为文本的一个简单统计。

讨论¶

当你想额外定义一个计数变量的时候，使用 `enumerate()` 函数会更加简单。你可能会像下面这样写代码：

```
lineno = 1
for line in f:
```

```
    # Process line
    ...
    lineno += 1
```

但是如果使用 `enumerate()` 函数来代替就显得更加优雅了：

```
for lineno, line in enumerate(f):
    # Process line
    ...
```

`enumerate()` 函数返回的是一个 `enumerate` 对象实例， 它是一个迭代器，返回连续的包含一个计数和一个值的元组，元组中的值通过在传入序列上调用 `next()` 返回。

还有一点可能并不很重要，但是也值得注意， 有时候当你在一个已经解压后的元组序列上使用 `enumerate()` 函数时很容易调入陷阱。 你得像下面正确的方式这样写：

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

# Correct!
for n, (x, y) in enumerate(data):
    ...
# Error!
for n, x, y in enumerate(data):
    ...
```

4.11 同时迭代多个序列¶

问题¶

你想同时迭代多个序列，每次分别从一个序列中取一个元素。

解决方案¶

为了同时迭代多个序列，使用 `zip()` 函数。比如：

```
>>> xpts = [1, 5, 4, 2, 10, 7]
>>> ypts = [101, 78, 37, 15, 62, 99]
>>> for x, y in zip(xpts, ypts):
...     print(x,y)
...
1 101
5 78
4 37
2 15
10 62
7 99
>>>
```

`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。

```
>>> a = [1, 2, 3]
>>> b = ['w', 'x', 'y', 'z']
>>> for i in zip(a,b):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
>>>
```

如果这个不是你想要的效果，那么还可以使用 `itertools.zip_longest()` 函数来代替。比如：

```
>>> from itertools import zip_longest
>>> for i in zip_longest(a,b):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
(None, 'z')

>>> for i in zip_longest(a, b, fillvalue=0):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
(0, 'z')
>>>
```

讨论¶

当你想成对处理数据的时候 `zip()` 函数是很有用的。 比如，假设你头列表和一个值列表，就像下面这样：

```
headers = ['name', 'shares', 'price']
values = ['ACME', 100, 490.1]
```

使用zip()可以让你将它们打包并生成一个字典：

```
s = dict(zip(headers,values))
```

或者你也可以像下面这样产生输出：

```
for name, val in zip(headers, values):
    print(name, '=', val)
```

虽然不常见，但是 `zip()` 可以接受多于两个的序列的参数。 这时候所生成的结果元组中元素个数跟输入序列个数一样。比如;

```
>>> a = [1, 2, 3]
>>> b = [10, 11, 12]
>>> c = ['x','y','z']
>>> for i in zip(a, b, c):
...     print(i)
...
(1, 10, 'x')
(2, 11, 'y')
(3, 12, 'z')
>>>
```

最后强调一点就是， `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。比如：

```
>>> zip(a, b)
<zip object at 0x1007001b8>
>>> list(zip(a, b))
[(1, 10), (2, 11), (3, 12)]
>>>
```

4.12 不同集合上元素的迭代¶

问题¶

你想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，你希望代码在不失可读性的情况下避免写重复的循环。

解决方案¶

`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，有效的屏蔽掉在多个容器中迭代细节。 为了演示清楚，考虑下面这个例子：

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

使用 `chain()` 的一个常见场景是当你想对不同的集合中所有元素执行某些操作的时候。比如：

```
# Various working sets of items
active_items = set()
inactive_items = set()

# Iterate over all items
for item in chain(active_items, inactive_items):
    # Process item
```

这种解决方案要比像下面这样使用两个单独的循环更加优雅，

```
for item in active_items:
    # Process item
    ...

for item in inactive_items:
    # Process item
    ...
```

讨论¶

`itertools.chain()` 接受一个或多个可迭代对象作为输入参数。 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。 这种方式要比先将序列合并再迭代要高效的多。比如：

```
# Inefficent
for x in a + b:
    ...

# Better
for x in chain(a, b):
    ...
```

第一种方案中， `a + b` 操作会创建一个全新的序列并要求a和b的类型一致。 `chian()` 不会有这一步，所以如果输入序列非常大的时候会很省内存。 并且当可迭代对象类型不一样的时候 `chain()` 同样可以很好的工作。

4.13 创建数据处理管道¶

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。

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：

```
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):
        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:
```

```
        yield from it

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
```

现在你可以很容易的将这些函数连起来创建一个处理管道。 比如，为了查找包含单词python的所有日志行，你可以这样做：

```
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?i)python', lines)
for line in pylines:
    print(line)
```

如果将来的时候你想扩展管道，你甚至可以在生成器表达式中包装数据。 比如，下面这个版本计算出传输的字节数并计算其总和。

```
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?i)python', lines)
bytecolumn = (line.rsplit(None,1)[1] for line in pylines)
bytes = (int(x) for x in bytecolumn if x != '-')
print('Total', sum(bytes))
```

讨论¶

以管道方式处理数据可以用来解决各类其他问题，包括解析，读取实时数据，定时轮询等。

为了理解上述代码，重点是要明白 `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。 在例子最后部分， `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。

这种方式一个非常好的特点是每个生成器函数很小并且都是独立的。这样的话就很容易编写和维护它们了。 很多时候，这些函数如果比较通用的话可以在其他场景重复使用。 并且最终将这些组件组合起来的代码看上去非常简单，也很容易理解。

使用这种方式的内存效率也不得不提。上述代码即便是在一个超大型文件目录中也能工作的很好。 事实上，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。

在调用 `gen_concatenate()` 函数的时候你可能会有些不太明白。 这个函数的目的是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作为参数传入。 在上面这个例子中，你可能会写类似这样的语句 `lines = itertools.chain(*files)` ， 这将导致 `gen_opener()` 生成器被提前全部消费掉。 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 `chain()` 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。 关于这个我们在4.14小节会有更进一步的描述。

最后还有一点需要注意的是，管道方式并不是万能的。 有时候你想立即处理所有数据。 然而，即便是这种情况，使用生成器管道也可以将这类问题从逻辑上变为工作流的处理方式。

David Beazley 在他的 Generator Tricks for Systems Programmers 教程中对于这种技术有非常深入的讲解。可以参考这个教程获取更多的信息。

4.14 展开嵌套的序列¶

问题¶

你想将一个多层嵌套的序列展开成一个单层列表

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：

```
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)
```

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。 这样的话字符串数组就能最终返回我们所期望的结果了。比如：

```
>>> items = ['Dave', 'Paula', ['Thomas', 'Lewis']]
>>> for x in flatten(items):
...     print(x)
...
Dave
Paula
Thomas
Lewis
>>>
```

讨论¶

语句 `yield from` 在你想在生成器中调用其他生成器作为子例程的时候非常有用。 如果你不使用它的话，那么就必须写额外的 `for` 循环了。比如：

```
def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            for i in flatten(x):
                yield i
        else:
            yield x
```

尽管只改了一点点，但是 `yield from` 语句看上去感觉更好，并且也使得代码更简洁清爽。

之前提到的对于字符串和字节的额外检查是为了防止将它们再展开成单个字符。 如果还有其他你不想展开的类型，修改参数 `ignore_types` 即可。

最后要注意的一点是， `yield from` 在涉及到基于协程和生成器的并发编程中扮演着更加重要的角色。 可以参考12.12小节查看另外一个例子。

4.15 顺序迭代合并后的排序迭代对象¶

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。

解决方案¶

`heapq.merge()` 函数可以帮你解决这个问题。比如：

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

讨论¶

`heapq.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。 比如，下面是一个例子来演示如何合并两个排序文件：

```
with open('sorted_file_1', 'rt') as file1, \
    open('sorted_file_2', 'rt') as file2, \
    open('merged_file', 'wt') as outf:

    for line in heapq.merge(file1, file2):
        outf.write(line)
```

有一点要强调的是 `heapq.merge()` 需要所有输入序列必须是排过序的。 特别的，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。

4.16 迭代器代替while无限循环¶

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？

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)
```

如果你怀疑它到底能不能正常工作，可以试验下一个简单的例子。比如：

```
>>> import sys
>>> f = open('/etc/passwd')
>>> for chunk in iter(lambda: f.read(10), ''):
...     n = sys.stdout.write(chunk)
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_uucp:*:4:4:Unix to Unix Copy Protocol:/var/spool/uucp:/usr/sbin/uucico
...
>>>
```

讨论¶

`iter` 函数一个鲜为人知的特性是它接受一个可选的 `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。

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。 举例来讲，如果你想从套接字或文件中以数据块的方式读取数据，通常你得要不断重复的执行 `read()` 或 `recv()`， 并在后面紧跟一个文件结尾测试来决定是否终止。这节中的方案使用一个简单的 `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。

4.2 代理迭代¶

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。

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：

```
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)
    # Outputs Node(1), Node(2)
    for ch in root:
        print(ch)
```

在上面代码中， `__iter__()` 方法只是简单的将迭代请求传递给内部的 `_children` 属性。

讨论¶

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。

这里的 `iter()` 函数的使用简化了代码， `iter(s)` 只是简单的通过调用 `s.__iter__()` 方法来返回对应的迭代器对象，就跟 `len(s)` 会调用 `s.__len__()` 原理是一样的。

4.3 使用生成器创建新的迭代模式¶

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。

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
```

为了使用这个函数， 你可以用for循环迭代它或者使用其他接受一个可迭代对象的函数(比如 `sum()` , `list()` 等)。示例如下：

```
>>> for n in frange(0, 4, 0.5):
...     print(n)
...
0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
>>> list(frange(0, 1, 0.125))
[0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
>>>
```

讨论¶

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。 下面是一个实验，向你展示这样的函数底层工作机制：

```
>>> def countdown(n):
...     print('Starting to count from', n)
...     while n > 0:
...         yield n
...         n -= 1
...     print('Done!')
...

>>> # Create the generator, notice no output appears
>>> 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。我们在迭代中通常使用的for语句会自动处理这些细节，所以你无需担心。

4.4 实现迭代器协议¶

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。

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：

```
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()

# Example
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)
```

在这段代码中，`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。

讨论¶

Python的迭代协议要求一个 `__iter__()` 方法返回一个特殊的迭代器对象， 这个迭代器对象实现了 `__next__()` 方法并通过 `StopIteration` 异常标识迭代的完成。 但是，实现这些通常会比较繁琐。 下面我们演示下这种方式，如何使用一个关联迭代器类重新实现 `depth_first()` 方法：

```
class Node2:
    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):
        return DepthFirstIterator(self)


class DepthFirstIterator(object):
    '''
    Depth-first traversal
    '''

    def __init__(self, start_node):
        self._node = start_node
        self._children_iter = None
        self._child_iter = None

    def __iter__(self):
        return self

    def __next__(self):
        # Return myself if just started; create an iterator for children
        if self._children_iter is None:
            self._children_iter = iter(self._node)
            return self._node
        # If processing a child, return its next item
        elif self._child_iter:
            try:
                nextchild = next(self._child_iter)
                return nextchild
            except StopIteration:
                self._child_iter = None
                return next(self)
        # Advance to the next child and start its iteration
        else:
            self._child_iter = next(self._children_iter).depth_first()
            return next(self)
```

`DepthFirstIterator` 类和上面使用生成器的版本工作原理类似， 但是它写起来很繁琐，因为迭代器必须在迭代处理过程中维护大量的状态信息。 坦白来讲，没人愿意写这么晦涩的代码。将你的迭代器定义为一个生成器后一切迎刃而解。

4.5 反向迭代¶

问题¶

你想反方向迭代一个序列

解决方案¶

使用内置的 `reversed()` 函数，比如：

```
>>> a = [1, 2, 3, 4]
>>> for x in reversed(a):
...     print(x)
...
4
3
2
1
```

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：

```
# Print a file backwards
f = open('somefile')
for line in reversed(list(f)):
    print(line, end='')
```

要注意的是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。

讨论¶

很多程序员并不知道可以通过在自定义类上实现 `__reversed__()` 方法来实现反向迭代。比如：

```
class Countdown:
    def __init__(self, start):
        self.start = start

    # Forward iterator
    def __iter__(self):
        n = self.start
        while n > 0:
            yield n
            n -= 1

    # Reverse iterator
    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。

4.6 带有外部状态的生成器函数¶

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。

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：

```
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()
```

为了使用这个类，你可以将它当做是一个普通的生成器函数。 然而，由于可以创建一个实例对象，于是你可以访问内部属性值， 比如 `history` 属性或者是 `clear()` 方法。代码示例如下：

```
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='')
```

讨论¶

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你的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)， 可能会导致你的代码异常的复杂。 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__()` 方法中定义你的生成器不会改变你任何的算法逻辑。 由于它是类的一部分，所以允许你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，那么你得先调用 `iter()` 函数。比如：

```
>>> f = open('somefile.txt')
>>> lines = linehistory(f)
>>> next(lines)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'linehistory' object is not an iterator

>>> # Call iter() first, then start iterating
>>> it = iter(lines)
>>> next(it)
'hello world\n'
>>> next(it)
'this is a test\n'
>>>
```

4.7 迭代器切片¶

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。

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：

```
>>> def count(n):
...     while True:
...         yield n
...         n += 1
...
>>> c = count(0)
>>> c[10:20]
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not subscriptable

>>> # Now using islice()
>>> import itertools
>>> for x in itertools.islice(c, 10, 20):
...     print(x)
...
10
11
12
13
14
15
16
17
18
19
>>>
```

讨论¶

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。

这里要着重强调的一点是 `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。 所以如果你需要之后再次访问这个迭代器的话，那你就得先将它里面的数据放入一个列表中。

4.8 跳过可迭代对象的开始部分¶

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。

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Flase之前的所有元素，然后返回后面所有元素。

为了演示，假定你在读取一个开始部分是几行注释的源文件。比如：

```
>>> with open('/etc/passwd') as f:
...     for line in f:
...         print(line, end='')
...
##
# User Database
#
# Note that this file is consulted directly only when the system is running
# in single-user mode. At other times, this information is provided by
# Open Directory.
...
##
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
>>>
```

如果你想跳过开始部分的注释行的话，可以这样做：

```
>>> from itertools import dropwhile
>>> with open('/etc/passwd') as f:
...     for line in dropwhile(lambda line: not line.startswith('#'), f):
...         print(line, end='')
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
>>>
```

这个例子是基于根据某个测试函数跳过开始的元素。 如果你已经明确知道了要跳过的元素的序号的话，那么可以使用 itertools.islice() 来代替。比如：

```
>>> from itertools import islice
>>> items = ['a', 'b', 'c', 1, 4, 10, 15]
>>> for x in islice(items, 3, None):
...     print(x)
...
4
10
15
>>>
```

在这个例子中， islice() 函数最后那个 None 参数指定了你要跳过前面3个元素，获取第4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， (这个跟切片的相反操作 [3:] 和 [:3] 原理是一样的)。

讨论¶

函数 dropwhile() 和 islice() 其实就是两个帮助函数，为的就是避免写出下面这种冗余代码：

```
with open('/etc/passwd') as f:
    # Skip over initial comments
    while True:
        line = next(f, '')
        if not line.startswith('#'):
            break

    # Process remaining lines
    while line:
        # Replace with useful processing
        print(line, end='')
        line = next(f, None)
```

跳过一个可迭代对象的开始部分跟通常的过滤是不同的。 比如，上述代码的第一个部分可能会这样重写：

```
with open('/etc/passwd') as f:
    lines = (line for line in f if not line.startswith('#'))
    for line in lines:
        print(line, end='')
```

这样写确实可以跳过开始部分的注释行，但是同样也会跳过文件中其他所有的注释行。 换句话讲，我们的解决方案是仅仅跳过开始部分满足测试条件的行，在那以后，所有的元素不再进行测试和过滤了。

最后需要着重强调的一点是，本节的方案适用于所有可迭代对象，包括那些事先不能确定大小的， 比如生成器，文件及其类似的对象。

4.9 排列组合的迭代¶

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

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：

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

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：

```
>>> for p in permutations(items, 2):
...     print(p)
...
('a', 'b')
('a', 'c')
('b', 'a')
('b', 'c')
('c', 'a')
('c', 'b')
>>>
```

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

```
>>> from itertools import combinations
>>> for c in combinations(items, 3):
...     print(c)
...
('a', 'b', 'c')

>>> for c in combinations(items, 2):
...     print(c)
...
('a', 'b')
('a', 'c')
('b', 'c')

>>> for c in combinations(items, 1):
...     print(c)
...
('a',)
('b',)
('c',)
>>>
```

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)。 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')
>>>
```

讨论¶

这一小节我们向你展示的仅仅是 itertools 模块的一部分功能。 尽管你也可以自己手动实现排列组合算法，但是这样做得要花点脑力。 当我们碰到看上去有些复杂的迭代问题时，最好可以先去看看itertools模块。 如果这个问题很普遍，那么很有可能会在里面找到解决方案！

第四章：迭代器与生成器¶

迭代是Python最强大的功能之一。初看起来，你可能会简单的认为迭代只不过是处理序列中元素的一种方法。 然而，绝非仅仅就是如此，还有很多你可能不知道的， 比如创建你自己的迭代器对象，在itertools模块中使用有用的迭代模式，构造生成器函数等等。 这一章目的就是向你展示跟迭代有关的各种常见问题。

Contents: