6.1 读写CSV数据¶

问题¶

你想读写一个CSV格式的文件。

解决方案¶

对于大多数的CSV格式的数据读写问题，都可以使用 `csv` 库。 例如：假设你在一个名叫stocks.csv文件中有一些股票市场数据，就像这样：

```
Symbol,Price,Date,Time,Change,Volume
"AA",39.48,"6/11/2007","9:36am",-0.18,181800
"AIG",71.38,"6/11/2007","9:36am",-0.15,195500
"AXP",62.58,"6/11/2007","9:36am",-0.46,935000
"BA",98.31,"6/11/2007","9:36am",+0.12,104800
"C",53.08,"6/11/2007","9:36am",-0.25,360900
"CAT",78.29,"6/11/2007","9:36am",-0.23,225400
```

下面向你展示如何将这些数据读取为一个元组的序列：

```
import csv
with open('stocks.csv') as f:
    f_csv = csv.reader(f)
    headers = next(f_csv)
    for row in f_csv:
        # Process row
        ...
```

在上面的代码中， `row` 会是一个列表。因此，为了访问某个字段，你需要使用下标，如 `row[0]` 访问Symbol， `row[4]` 访问Change。

由于这种下标访问通常会引起混淆，你可以考虑使用命名元组。例如：

```
from collections import namedtuple
with open('stock.csv') as f:
    f_csv = csv.reader(f)
    headings = next(f_csv)
    Row = namedtuple('Row', headings)
    for r in f_csv:
        row = Row(*r)
        # Process row
        ...
```

它允许你使用列名如 `row.Symbol` 和 `row.Change` 代替下标访问。 需要注意的是这个只有在列名是合法的Python标识符的时候才生效。如果不是的话， 你可能需要修改下原始的列名(如将非标识符字符替换成下划线之类的)。

另外一个选择就是将数据读取到一个字典序列中去。可以这样做：

```
import csv
with open('stocks.csv') as f:
    f_csv = csv.DictReader(f)
    for row in f_csv:
        # process row
        ...
```

在这个版本中，你可以使用列名去访问每一行的数据了。比如，`row['Symbol']` 或者 `row['Change']`

为了写入CSV数据，你仍然可以使用csv模块，不过这时候先创建一个 `writer` 对象。例如:

```
headers = ['Symbol','Price','Date','Time','Change','Volume']
rows = [('AA', 39.48, '6/11/2007', '9:36am', -0.18, 181800),
         ('AIG', 71.38, '6/11/2007', '9:36am', -0.15, 195500),
         ('AXP', 62.58, '6/11/2007', '9:36am', -0.46, 935000),
```

```
       ]

with open('stocks.csv','w') as f:
    f_csv = csv.writer(f)
    f_csv.writerow(headers)
    f_csv.writerows(rows)
```

如果你有一个字典序列的数据，可以像这样做：

```
headers = ['Symbol', 'Price', 'Date', 'Time', 'Change', 'Volume']
rows = [{'Symbol':'AA', 'Price':39.48, 'Date':'6/11/2007',
        'Time':'9:36am', 'Change':-0.18, 'Volume':181800},
        {'Symbol':'AIG', 'Price': 71.38, 'Date':'6/11/2007',
        'Time':'9:36am', 'Change':-0.15, 'Volume': 195500},
        {'Symbol':'AXP', 'Price': 62.58, 'Date':'6/11/2007',
        'Time':'9:36am', 'Change':-0.46, 'Volume': 935000},
        ]

with open('stocks.csv','w') as f:
    f_csv = csv.DictWriter(f, headers)
    f_csv.writeheader()
    f_csv.writerows(rows)
```

讨论¶

你应该总是优先选择csv模块分割或解析CSV数据。例如，你可能会像编写类似下面这样的代码：

```
with open('stocks.csv') as f:
for line in f:
    row = line.split(',')
    # process row
    ...
```

使用这种方式的一个缺点就是你仍然需要去处理一些棘手的细节问题。 比如，如果某些字段值被引号包围，你不得不去除这些引号。 另外，如果一个被引号包围的字段碰巧含有一个逗号，那么程序就会因为产生一个错误大小的行而出错。

默认情况下，`csv` 库可识别Microsoft Excel所使用的CSV编码规则。 这或许也是最常见的形式，并且也会给你带来最好的兼容性。 然而，如果你查看csv的文档，就会发现有很多种方法将它应用到其他编码格式上(如修改分割字符等)。 例如，如果你想读取以tab分割的数据，可以这样做：

```
# Example of reading tab-separated values
with open('stock.tsv') as f:
    f_tsv = csv.reader(f, delimiter='\t')
    for row in f_tsv:
        # Process row
        ...
```

如果你正在读取CSV数据并将它们转换为命名元组，需要注意对列名进行合法性认证。 例如，一个CSV格式文件有一个包含非法标识符的列头行，类似下面这样：

```
Street Address,Num-Premises,Latitude,Longitude 5412 N CLARK,10,41.980262,-87.668452
```

这样最终会导致在创建一个命名元组时产生一个 `ValueError` 异常而失败。 为了解决这问题，你可能不得不先去修正列标题。 例如，可以像下面这样在非法标识符上使用一个正则表达式替换：

```
import re
with open('stock.csv') as f:
    f_csv = csv.reader(f)
    headers = [ re.sub('[^a-zA-Z_]', '_', h) for h in next(f_csv) ]
    Row = namedtuple('Row', headers)
    for r in f_csv:
        row = Row(*r)
        # Process row
        ...
```

还有重要的一点需要强调的是，csv产生的数据都是字符串类型的，它不会做任何其他类型的转换。 如果你需要做这样

的类型转换，你必须自己手动去实现。 下面是一个在CSV数据上执行其他类型转换的例子：

```
col_types = [str, float, str, str, float, int]
with open('stocks.csv') as f:
    f_csv = csv.reader(f)
    headers = next(f_csv)
    for row in f_csv:
        # Apply conversions to the row items
        row = tuple(convert(value) for convert, value in zip(col_types, row))
        ...
```

另外，下面是一个转换字典中特定字段的例子：

```
print('Reading as dicts with type conversion')
field_types = [ ('Price', float),
                ('Change', float),
                ('Volume', int) ]

with open('stocks.csv') as f:
    for row in csv.DictReader(f):
        row.update((key, conversion(row[key]))
                for key, conversion in field_types)
        print(row)
```

通常来讲，你可能并不想过多去考虑这些转换问题。 在实际情况中，CSV文件都或多或少有些缺失的数据，被破坏的数据以及其它一些让转换失败的问题。 因此，除非你的数据确实有保障是准确无误的，否则你必须考虑这些问题(你可能需要增加合适的错误处理机制)。

最后，如果你读取CSV数据的目的是做数据分析和统计的话， 你可能需要看一看 `Pandas` 包。`Pandas` 包含了一个非常方便的函数叫 `pandas.read_csv()`， 它可以加载CSV数据到一个 `DataFrame` 对象中去。 然后利用这个对象你就可以生成各种形式的统计、过滤数据以及执行其他高级操作了。 在6.13小节中会有这样一个例子。

6.10 编码解码Base64数据¶

问题¶

你需要使用Base64格式解码或编码二进制数据。

解决方案¶

`base64` 模块中有两个函数 `b64encode()` and `b64decode()` 可以帮你解决这个问题。例如;

```
>>> # Some byte data
>>> s = b'hello'
>>> import base64

>>> # Encode as Base64
>>> a = base64.b64encode(s)
>>> a
b'aGVsbG8='

>>> # Decode from Base64
>>> base64.b64decode(a)
b'hello'
>>>
```

讨论¶

Base64编码仅仅用于面向字节的数据比如字节字符串和字节数组。 此外，编码处理的输出结果总是一个字节字符串。 如果你想混合使用Base64编码的数据和Unicode文本，你必须添加一个额外的解码步骤。例如：

```
>>> a = base64.b64encode(s).decode('ascii')
>>> a
'aGVsbG8='
>>>
```

当解码Base64的时候，字节字符串和Unicode文本都可以作为参数。 但是，Unicode字符串只能包含ASCII字符。

6.11 读写二进制数组数据¶

问题¶

你想读写一个二进制数组的结构化数据到Python元组中。

解决方案¶

可以使用 `struct` 模块处理二进制数据。 下面是一段示例代码将一个Python元组列表写入一个二进制文件，并使用 `struct` 将每个元组编码为一个结构体。

```
from struct import Struct
def write_records(records, format, f):
    '''
    Write a sequence of tuples to a binary file of structures.
    '''
    record_struct = Struct(format)
    for r in records:
        f.write(record_struct.pack(*r))

# Example
if __name__ == '__main__':
    records = [ (1, 2.3, 4.5),
                (6, 7.8, 9.0),
                (12, 13.4, 56.7) ]
    with open('data.b', 'wb') as f:
        write_records(records, '<idd', f)
```

有很多种方法来读取这个文件并返回一个元组列表。 首先，如果你打算以块的形式增量读取文件，你可以这样做：

```
from struct import Struct

def read_records(format, f):
    record_struct = Struct(format)
    chunks = iter(lambda: f.read(record_struct.size), b'')
    return (record_struct.unpack(chunk) for chunk in chunks)

# Example
if __name__ == '__main__':
    with open('data.b','rb') as f:
        for rec in read_records('<idd', f):
            # Process rec
            ...
```

如果你想将整个文件一次性读取到一个字节字符串中，然后在分片解析。那么你可以这样做：

```
from struct import Struct

def unpack_records(format, data):
    record_struct = Struct(format)
    return (record_struct.unpack_from(data, offset)
            for offset in range(0, len(data), record_struct.size))

# Example
if __name__ == '__main__':
    with open('data.b', 'rb') as f:
        data = f.read()
    for rec in unpack_records('<idd', data):
        # Process rec
        ...
```

两种情况下的结果都是一个可返回用来创建该文件的原始元组的可迭代对象。

讨论¶

对于需要编码和解码二进制数据的程序而言，通常会使用 `struct` 模块。 为了声明一个新的结构体，只需要像这样创建一个 `Struct` 实例即可：

```
# Little endian 32-bit integer, two double precision floats
record_struct = Struct('<idd')
```

结构体通常会使用一些结构码值i, d, f等 [参考 Python文档]。 这些代码分别代表某个特定的二进制数据类型如32位整数，64位浮点数，32位浮点数等。 第一个字符 < 指定了字节顺序。在这个例子中，它表示"低位在前"。 更改这个字符为 > 表示高位在前，或者是 ! 表示网络字节顺序。

产生的 `Struct` 实例有很多属性和方法用来操作相应类型的结构。 `size` 属性包含了结构的字节数，这在I/O操作时非常有用。 `pack()` 和 `unpack()` 方法被用来打包和解包数据。比如：

```
>>> from struct import Struct
>>> record_struct = Struct('<idd')
>>> record_struct.size
20
>>> record_struct.pack(1, 2.0, 3.0)
b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x08@'
>>> record_struct.unpack(_)
(1, 2.0, 3.0)
>>>
```

有时候你还会看到 `pack()` 和 `unpack()` 操作以模块级别函数被调用，类似下面这样：

```
>>> import struct
>>> struct.pack('<idd', 1, 2.0, 3.0)
b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x08@'
>>> struct.unpack('<idd', _)
(1, 2.0, 3.0)
>>>
```

这样可以工作，但是感觉没有实例方法那么优雅，特别是在你代码中同样的结构出现在多个地方的时候。 通过创建一个 `Struct` 实例，格式代码只会指定一次并且所有的操作被集中处理。 这样一来代码维护就变得更加简单了(因为你只需要改变一处代码即可)。

读取二进制结构的代码要用到一些非常有趣而优美的编程技巧。 在函数 `read_records` 中，`iter()` 被用来创建一个返回固定大小数据块的迭代器，参考5.8小节。 这个迭代器会不断的调用一个用户提供的可调用对象(比如 `lambda: f.read(record_struct.size)`)， 直到它返回一个特殊的值(如b'')，这时候迭代停止。例如：

```
>>> f = open('data.b', 'rb')
>>> chunks = iter(lambda: f.read(20), b'')
>>> chunks
<callable_iterator object at 0x10069e6d0>
>>> for chk in chunks:
... print(chk)
...
b'\x01\x00\x00\x00ffffff\x02@\x00\x00\x00\x00\x00\x00\x12@'
b'\x06\x00\x00\x00333333\x1f@\x00\x00\x00\x00\x00\x00"@'
b'\x0c\x00\x00\x00\xcd\xcc\xcc\xcc\xcc\xcc*@\x9a\x99\x99\x99\x99YL@'
>>>
```

如你所见，创建一个可迭代对象的一个原因是它能允许使用一个生成器推导来创建记录。 如果你不使用这种技术，那么代码可能会像下面这样：

```
def read_records(format, f):
    record_struct = Struct(format)
    while True:
        chk = f.read(record_struct.size)
        if chk == b'':
            break
        yield record_struct.unpack(chk)
```

在函数 `unpack_records()` 中使用了另外一种方法 `unpack_from()` 。 `unpack_from()` 对于从一个大型二进制数组中提取二进制数据非常有用， 因为它不会产生任何的临时对象或者进行内存复制操作。 你只需要给它一个字节字符串(或数组)和一个字节偏移量，它会从那个位置开始直接解包数据。

如果你使用 `unpack()` 来代替 `unpack_from()`， 你需要修改代码来构造大量的小的切片以及进行偏移量的计算。比如：

```
def unpack_records(format, data):
    record_struct = Struct(format)
    return (record_struct.unpack(data[offset:offset + record_struct.size])
            for offset in range(0, len(data), record_struct.size))
```

这种方案除了代码看上去很复杂外，还得做很多额外的工作，因为它执行了大量的偏移量计算， 复制数据以及构造小的切片对象。 如果你准备从读取到的一个大型字节字符串中解包大量的结构体的话，`unpack_from()` 会表现的更出色。

在解包的时候，`collections` 模块中的命名元组对象或许是你想要用到的。 它可以让你给返回元组设置属性名称。例如：

```
from collections import namedtuple

Record = namedtuple('Record', ['kind','x','y'])

with open('data.p', 'rb') as f:
    records = (Record(*r) for r in read_records('<idd', f))

for r in records:
    print(r.kind, r.x, r.y)
```

如果你的程序需要处理大量的二进制数据，你最好使用 `numpy` 模块。 例如，你可以将一个二进制数据读取到一个结构化数组中而不是一个元组列表中。就像下面这样：

```
>>> import numpy as np
>>> f = open('data.b', 'rb')
>>> records = np.fromfile(f, dtype='<i,<d,<d')
>>> records
array([(1, 2.3, 4.5), (6, 7.8, 9.0), (12, 13.4, 56.7)],
dtype=[('f0', '<i4'), ('f1', '<f8'), ('f2', '<f8')])
>>> records[0]
(1, 2.3, 4.5)
>>> records[1]
(6, 7.8, 9.0)
>>>
```

最后提一点，如果你需要从已知的文件格式(如图片格式，图形文件，HDF5等)中读取二进制数据时， 先检查看看Python是不是已经提供了现存的模块。因为不到万不得已没有必要去重复造轮子。

6.12 读取嵌套和可变长二进制数据¶

问题¶

你需要读取包含嵌套或者可变长记录集合的复杂二进制格式的数据。这些数据可能包含图片、视频、电子地图文件等。

解决方案¶

`struct` 模块可被用来编码/解码几乎所有类型的二进制的数据结构。为了解释清楚这种数据，假设你用下面的Python数据结构 来表示一个组成一系列多边形的点的集合：

```
polys = [
    [ (1.0, 2.5), (3.5, 4.0), (2.5, 1.5) ],
    [ (7.0, 1.2), (5.1, 3.0), (0.5, 7.5), (0.8, 9.0) ],
    [ (3.4, 6.3), (1.2, 0.5), (4.6, 9.2) ],
]
```

现在假设这个数据被编码到一个以下列头部开始的二进制文件中去了：

```
+------+--------+-------------------------------------+
|Byte  | Type   |  Description                        |
+======+========+=====================================+
|0     | int    |  文件代码（0x1234，小端）           |
+------+--------+-------------------------------------+
|4     | double |  x 的最小值（小端）                 |
+------+--------+-------------------------------------+
|12    | double |  y 的最小值（小端）                 |
+------+--------+-------------------------------------+
|20    | double |  x 的最大值（小端）                 |
+------+--------+-------------------------------------+
|28    | double |  y 的最大值（小端）                 |
+------+--------+-------------------------------------+
|36    | int    |  三角形数量（小端）                 |
+------+--------+-------------------------------------+
```

紧跟着头部是一系列的多边形记录，编码格式如下：

```
+------+--------+-------------------------------------------+
|Byte  | Type   |  Description                              |
+======+========+===========================================+
|0     | int    |  记录长度（N字节）                        |
+------+--------+-------------------------------------------+
|4-N   | Points |  (X,Y) 坐标，以浮点数表示                 |
+------+--------+-------------------------------------------+
```

为了写这样的文件，你可以使用如下的Python代码：

```
import struct
import itertools

def write_polys(filename, polys):
    # Determine bounding box
    flattened = list(itertools.chain(*polys))
    min_x = min(x for x, y in flattened)
    max_x = max(x for x, y in flattened)
    min_y = min(y for x, y in flattened)
    max_y = max(y for x, y in flattened)
    with open(filename, 'wb') as f:
        f.write(struct.pack('<iddddi', 0x1234,
                            min_x, min_y,
                            max_x, max_y,
                            len(polys)))
        for poly in polys:
            size = len(poly) * struct.calcsize('<dd')
```

```
            f.write(struct.pack('<i', size + 4))
            for pt in poly:
                f.write(struct.pack('<dd', *pt))
```

将数据读取回来的时候，可以利用函数 `struct.unpack()`，代码很相似，基本就是上面写操作的逆序。如下：

```
def read_polys(filename):
    with open(filename, 'rb') as f:
        # Read the header
        header = f.read(40)
        file_code, min_x, min_y, max_x, max_y, num_polys = \
            struct.unpack('<iddddi', header)
        polys = []
        for n in range(num_polys):
            pbytes, = struct.unpack('<i', f.read(4))
            poly = []
            for m in range(pbytes // 16):
                pt = struct.unpack('<dd', f.read(16))
                poly.append(pt)
            polys.append(poly)
    return polys
```

尽管这个代码可以工作，但是里面混杂了很多读取、解包数据结构和其他细节的代码。如果用这样的代码来处理真实的数据文件， 那未免也太繁杂了点。因此很显然应该有另一种解决方法可以简化这些步骤，让程序员只关注自最重要的事情。

在本小节接下来的部分，我会逐步演示一个更加优秀的解析字节数据的方案。 目标是可以给程序员提供一个高级的文件格式化方法，并简化读取和解包数据的细节。但是我要先提醒你， 本小节接下来的部分代码应该是整本书中最复杂最高级的例子，使用了大量的面向对象编程和元编程技术。 一定要仔细的阅读我们的讨论部分，另外也要参考下其他章节内容。

首先，当读取字节数据的时候，通常在文件开始部分会包含文件头和其他的数据结构。 尽管struct模块可以解包这些数据到一个元组中去，另外一种表示这种信息的方式就是使用一个类。 就像下面这样：

```
import struct

class StructField:
    '''
    Descriptor representing a simple structure field
    '''
    def __init__(self, format, offset):
        self.format = format
        self.offset = offset
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            r = struct.unpack_from(self.format, instance._buffer, self.offset)
            return r[0] if len(r) == 1 else r

class Structure:
    def __init__(self, bytedata):
        self._buffer = memoryview(bytedata)
```

这里我们使用了一个描述器来表示每个结构字段，每个描述器包含一个结构兼容格式的代码以及一个字节偏移量， 存储在内部的内存缓冲中。在 `__get__()` 方法中，`struct.unpack_from()` 函数被用来从缓冲中解包一个值，省去了额外的分片或复制操作步骤。

`Structure` 类就是一个基础类，接受字节数据并存储在内部的内存缓冲中，并被 `StructField` 描述器使用。 这里使用了 `memoryview()`，我们会在后面详细讲解它是用来干嘛的。

使用这个代码，你现在就能定义一个高层次的结构对象来表示上面表格信息所期望的文件格式。例如：

```
class PolyHeader(Structure):
    file_code = StructField('<i', 0)
    min_x = StructField('<d', 4)
    min_y = StructField('<d', 12)
```

```
    max_x = StructField('<d', 20)
    max_y = StructField('<d', 28)
    num_polys = StructField('<i', 36)
```

下面的例子利用这个类来读取之前我们写入的多边形数据的头部数据：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader(f.read(40))
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

这个很有趣，不过这种方式还是有一些烦人的地方。首先，尽管你获得了一个类接口的便利， 但是这个代码还是有点臃肿，还需要使用者指定很多底层的细节(比如重复使用 `StructField` ，指定偏移量等)。 另外，返回的结果类同样确实一些便利的方法来计算结构的总数。

任何时候只要你遇到了像这样冗余的类定义，你应该考虑下使用类装饰器或元类。 元类有一个特性就是它能够被用来填充许多低层的实现细节，从而释放使用者的负担。 下面我来举个例子，使用元类稍微改造下我们的 `Structure` 类：

```
class StructureMeta(type):
    '''
    Metaclass that automatically creates StructField descriptors
    '''
    def __init__(self, clsname, bases, clsdict):
        fields = getattr(self, '_fields_', [])
        byte_order = ''
        offset = 0
        for format, fieldname in fields:
            if format.startswith(('<','>','!','@')):
                byte_order = format[0]
                format = format[1:]
            format = byte_order + format
            setattr(self, fieldname, StructField(format, offset))
            offset += struct.calcsize(format)
        setattr(self, 'struct_size', offset)

class Structure(metaclass=StructureMeta):
    def __init__(self, bytedata):
        self._buffer = bytedata

    @classmethod
    def from_file(cls, f):
        return cls(f.read(cls.struct_size))
```

使用新的 `Structure` 类，你可以像下面这样定义一个结构：

```
class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        ('d', 'min_x'),
        ('d', 'min_y'),
        ('d', 'max_x'),
        ('d', 'max_y'),
        ('i', 'num_polys')
    ]
```

正如你所见，这样写就简单多了。我们添加的类方法 `from_file()` 让我们在不需要知道任何数据的大小和结构的情况下就能轻松的从文件中读取数据。比如：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

一旦你开始使用了元类，你就可以让它变得更加智能。例如，假设你还想支持嵌套的字节结构， 下面是对前面元类的一个小的改进，提供了一个新的辅助描述器来达到想要的效果：

```
class NestedStruct:
    '''
    Descriptor representing a nested structure
    '''
    def __init__(self, name, struct_type, offset):
        self.name = name
        self.struct_type = struct_type
        self.offset = offset

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            data = instance._buffer[self.offset:
                                    self.offset+self.struct_type.struct_size]
            result = self.struct_type(data)
            # Save resulting structure back on instance to avoid
            # further recomputation of this step
            setattr(instance, self.name, result)
            return result

class StructureMeta(type):
    '''
    Metaclass that automatically creates StructField descriptors
    '''
    def __init__(self, clsname, bases, clsdict):
        fields = getattr(self, '_fields_', [])
        byte_order = ''
        offset = 0
        for format, fieldname in fields:
            if isinstance(format, StructureMeta):
                setattr(self, fieldname,
                        NestedStruct(fieldname, format, offset))
                offset += format.struct_size
            else:
                if format.startswith(('<','>','!','@')):
                    byte_order = format[0]
                    format = format[1:]
                format = byte_order + format
                setattr(self, fieldname, StructField(format, offset))
                offset += struct.calcsize(format)
        setattr(self, 'struct_size', offset)
```

在这段代码中，`NestedStruct` 描述器被用来叠加另外一个定义在某个内存区域上的结构。 它通过将原始内存缓冲进行切片操作后实例化给定的结构类型。由于底层的内存缓冲区是通过一个内存视图初始化的， 所以这种切片操作不会引发任何的额外的内存复制。相反，它仅仅就是之前的内存的一个叠加而已。 另外，为了防止重复实例化，通过使用和8.10小节同样的技术，描述器保存了该实例中的内部结构对象。

使用这个新的修正版，你就可以像下面这样编写：

```
class Point(Structure):
    _fields_ = [
        ('<d', 'x'),
        ('d', 'y')
    ]

class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        (Point, 'min'), # nested struct
        (Point, 'max'), # nested struct
        ('i', 'num_polys')
    ]
```

令人惊讶的是，它也能按照预期的正常工作，我们实际操作下：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min # Nested structure
<__main__.Point object at 0x1006a48d0>
>>> phead.min.x
0.5
>>> phead.min.y
0.5
>>> phead.max.x
7.0
>>> phead.max.y
9.2
>>> phead.num_polys
3
>>>
```

到目前为止，一个处理定长记录的框架已经写好了。但是如果组件记录是变长的呢？ 比如，多边形文件包含变长的部分。

一种方案是写一个类来表示字节数据，同时写一个工具函数来通过多少方式解析内容。跟6.11小节的代码很类似：

```
class SizedRecord:
    def __init__(self, bytedata):
        self._buffer = memoryview(bytedata)

    @classmethod
    def from_file(cls, f, size_fmt, includes_size=True):
        sz_nbytes = struct.calcsize(size_fmt)
        sz_bytes = f.read(sz_nbytes)
        sz, = struct.unpack(size_fmt, sz_bytes)
        buf = f.read(sz - includes_size * sz_nbytes)
        return cls(buf)

    def iter_as(self, code):
        if isinstance(code, str):
            s = struct.Struct(code)
            for off in range(0, len(self._buffer), s.size):
                yield s.unpack_from(self._buffer, off)
        elif isinstance(code, StructureMeta):
            size = code.struct_size
            for off in range(0, len(self._buffer), size):
                data = self._buffer[off:off+size]
                yield code(data)
```

类方法 `SizedRecord.from_file()` 是一个工具，用来从一个文件中读取带大小前缀的数据块， 这也是很多文件格式常用的方式。作为输入，它接受一个包含大小编码的结构格式编码，并且也是自己形式。 可选的 `includes_size` 参数指定了字节数是否包含头部大小。 下面是一个例子教你怎样使用从多边形文件中读取单独的多边形数据：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.num_polys
```

```
3
>>> polydata = [ SizedRecord.from_file(f, '<i')
...              for n in range(phead.num_polys) ]
>>> polydata
[<__main__.SizedRecord object at 0x1006a4d50>,
<__main__.SizedRecord object at 0x1006a4f50>,
<__main__.SizedRecord object at 0x10070da90>]
>>>
```

可以看出，`SizedRecord` 实例的内容还没有被解析出来。 可以使用 `iter_as()` 方法来达到目的，这个方法接受一个结构格式化编码或者是 `Structure` 类作为输入。 这样子可以很灵活的去解析数据，例如：

```
>>> for n, poly in enumerate(polydata):
...     print('Polygon', n)
...     for p in poly.iter_as('<dd'):
...         print(p)
...
Polygon 0
(1.0, 2.5)
(3.5, 4.0)
(2.5, 1.5)
Polygon 1
(7.0, 1.2)
(5.1, 3.0)
(0.5, 7.5)
(0.8, 9.0)
Polygon 2
(3.4, 6.3)
(1.2, 0.5)
(4.6, 9.2)
>>>

>>> for n, poly in enumerate(polydata):
...     print('Polygon', n)
...     for p in poly.iter_as(Point):
...         print(p.x, p.y)
...
Polygon 0
1.0 2.5
3.5 4.0
2.5 1.5
Polygon 1
7.0 1.2
5.1 3.0
0.5 7.5
0.8 9.0
Polygon 2
3.4 6.3
1.2 0.5
4.6 9.2
>>>
```

将所有这些结合起来，下面是一个 `read_polys()` 函数的另外一个修正版：

```
class Point(Structure):
    _fields_ = [
        ('<d', 'x'),
        ('d', 'y')
    ]

class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        (Point, 'min'),
        (Point, 'max'),
        ('i', 'num_polys')
    ]

def read_polys(filename):
    polys = []
```

```
    with open(filename, 'rb') as f:
        phead = PolyHeader.from_file(f)
        for n in range(phead.num_polys):
            rec = SizedRecord.from_file(f, '<i')
            poly = [ (p.x, p.y) for p in rec.iter_as(Point) ]
            polys.append(poly)
    return polys
```

讨论¶

这一节向你展示了许多高级的编程技术，包括描述器，延迟计算，元类，类变量和内存视图。 然而，它们都为了同一个特定的目标服务。

上面的实现的一个主要特征是它是基于懒解包的思想。当一个 `Structure` 实例被创建时， `__init__()` 仅仅只是创建一个字节数据的内存视图，没有做其他任何事。 特别的，这时候并没有任何的解包或者其他与结构相关的操作发生。 这样做的一个动机是你可能仅仅只对一个字节记录的某一小部分感兴趣。我们只需要解包你需要访问的部分，而不是整个文件。

为了实现懒解包和打包，需要使用 `StructField` 描述器类。 用户在 `_fields_` 中列出来的每个属性都会被转化成一个 `StructField` 描述器， 它将相关结构格式码和偏移值保存到存储缓存中。元类 `StructureMeta` 在多个结构类被定义时自动创建了这些描述器。 我们使用元类的一个主要原因是它使得用户非常方便的通过一个高层描述就能指定结构格式，而无需考虑低层的细节问题。

`StructureMeta` 的一个很微妙的地方就是它会固定字节数据顺序。 也就是说，如果任意的属性指定了一个字节顺序(< 表示低位优先 或者 >表示高位优先)， 那后面所有字段的顺序都以这个顺序为准。这么做可以帮助避免额外输入，但是在定义的中间我们仍然可能切换顺序的。 比如，你可能有一些比较复杂的结构，就像下面这样：

```
class ShapeFile(Structure):
    _fields_ = [ ('>i', 'file_code'), # Big endian
        ('20s', 'unused'),
        ('i', 'file_length'),
        ('<i', 'version'), # Little endian
        ('i', 'shape_type'),
        ('d', 'min_x'),
        ('d', 'min_y'),
        ('d', 'max_x'),
        ('d', 'max_y'),
        ('d', 'min_z'),
        ('d', 'max_z'),
        ('d', 'min_m'),
        ('d', 'max_m') ]
```

之前我们提到过，`memoryview()` 的使用可以帮助我们避免内存的复制。 当结构存在嵌套的时候，`memoryviews` 可以叠加同一内存区域上定义的机构的不同部分。 这个特性比较微妙，但是它关注的是内存视图与普通字节数组的切片操作行为。 如果你在一个字节字符串或字节数组上执行切片操作，你通常会得到一个数据的拷贝。 而内存视图切片不是这样的，它仅仅是在已存在的内存上面叠加而已。因此，这种方式更加高效。

还有很多相关的章节可以帮助我们扩展这里讨论的方案。 参考8.13小节使用描述器构建一个类型系统。 8.10小节有更多关于延迟计算属性值的讨论，并且跟NestedStruct描述器的实现也有关。 9.19小节有一个使用元类来初始化类成员的例子，和 `StructureMeta` 类非常相似。 Python的 `ctypes` 源码同样也很有趣，它提供了对定义数据结构、数据结构嵌套这些相似功能的支持。

6.13 数据的累加与统计操作¶

问题¶

你需要处理一个很大的数据集并需要计算数据总和或其他统计量。

解决方案¶

对于任何涉及到统计、时间序列以及其他相关技术的数据分析问题，都可以考虑使用 Pandas库 。

为了让你先体验下，下面是一个使用Pandas来分析芝加哥城市的 老鼠和啮齿类动物数据库 的例子。 在我写这篇文章的时候，这个数据库是一个拥有大概74,000行数据的CSV文件。

```
>>> import pandas

>>> # Read a CSV file, skipping last line
>>> rats = pandas.read_csv('rats.csv', skip_footer=1)
>>> rats
<class 'pandas.core.frame.DataFrame'>
Int64Index: 74055 entries, 0 to 74054
Data columns:
Creation Date 74055 non-null values
Status 74055 non-null values
Completion Date 72154 non-null values
Service Request Number 74055 non-null values
Type of Service Request 74055 non-null values
Number of Premises Baited 65804 non-null values
Number of Premises with Garbage 65600 non-null values
Number of Premises with Rats 65752 non-null values
Current Activity 66041 non-null values
Most Recent Action 66023 non-null values
Street Address 74055 non-null values
ZIP Code 73584 non-null values
X Coordinate 74043 non-null values
Y Coordinate 74043 non-null values
Ward 74044 non-null values
Police District 74044 non-null values
Community Area 74044 non-null values
Latitude 74043 non-null values
Longitude 74043 non-null values
Location 74043 non-null values
dtypes: float64(11), object(9)

>>> # Investigate range of values for a certain field
>>> rats['Current Activity'].unique()
array([nan, Dispatch Crew, Request Sanitation Inspector], dtype=object)
>>> # Filter the data
>>> crew_dispatched = rats[rats['Current Activity'] == 'Dispatch Crew']
>>> len(crew_dispatched)
65676
>>>

>>> # Find 10 most rat-infested ZIP codes in Chicago
>>> crew_dispatched['ZIP Code'].value_counts()[:10]
60647 3837
60618 3530
60614 3284
60629 3251
60636 2801
60657 2465
60641 2238
60609 2206
60651 2152
60632 2071
>>>
```

```
>>> # Group by completion date
>>> dates = crew_dispatched.groupby('Completion Date')
<pandas.core.groupby.DataFrameGroupBy object at 0x10d0a2a10>
>>> len(dates)
472
>>>

>>> # Determine counts on each day
>>> date_counts = dates.size()
>>> date_counts[0:10]
Completion Date
01/03/2011 4
01/03/2012 125
01/04/2011 54
01/04/2012 38
01/05/2011 78
01/05/2012 100
01/06/2011 100
01/06/2012 58
01/07/2011 1
01/09/2012 12
>>>

>>> # Sort the counts
>>> date_counts.sort()
>>> date_counts[-10:]
Completion Date
10/12/2012 313
10/21/2011 314
09/20/2011 316
10/26/2011 319
02/22/2011 325
10/26/2012 333
03/17/2011 336
10/13/2011 378
10/14/2011 391
10/07/2011 457
>>>
```

嗯，看样子2011年10月7日对老鼠们来说是个很忙碌的日子啊！^_^

讨论¶

Pandas是一个拥有很多特性的大型函数库，我在这里不可能介绍完。 但是只要你需要去分析大型数据集合、对数据分组、计算各种统计量或其他类似任务的话，这个函数库真的值得你去看一看。

6.2 读写JSON数据¶

问题¶

你想读写JSON(JavaScript Object Notation)编码格式的数据。

解决方案¶

`json` 模块提供了一种很简单的方式来编码和解码JSON数据。 其中两个主要的函数是 `json.dumps()` 和 `json.loads()` ， 要比其他序列化函数库如pickle的接口少得多。 下面演示如何将一个Python数据结构转换为JSON：

```
import json

data = {
    'name' : 'ACME',
    'shares' : 100,
    'price' : 542.23
}

json_str = json.dumps(data)
```

下面演示如何将一个JSON编码的字符串转换回一个Python数据结构：

```
data = json.loads(json_str)
```

如果你要处理的是文件而不是字符串，你可以使用 `json.dump()` 和 `json.load()` 来编码和解码JSON数据。例如：

```
# Writing JSON data
with open('data.json', 'w') as f:
    json.dump(data, f)

# Reading data back
with open('data.json', 'r') as f:
    data = json.load(f)
```

讨论¶

JSON编码支持的基本数据类型为 `None` ， `bool` ， `int` ， `float` 和 `str` ， 以及包含这些类型数据的lists，tuples和dictionaries。 对于dictionaries，keys需要是字符串类型(字典中任何非字符串类型的key在编码时会先转换为字符串)。 为了遵循JSON规范，你应该只编码Python的lists和dictionaries。 而且，在web应用程序中，顶层对象被编码为一个字典是一个标准做法。

JSON编码的格式对于Python语法而已几乎是完全一样的，除了一些小的差异之外。 比如，True会被映射为true，False被映射为false，而None会被映射为null。 下面是一个例子，演示了编码后的字符串效果：

```
>>> json.dumps(False)
'false'
>>> d = {'a': True,
...      'b': 'Hello',
...      'c': None}
>>> json.dumps(d)
'{"b": "Hello", "c": null, "a": true}'
>>>
```

如果你试着去检查JSON解码后的数据，你通常很难通过简单的打印来确定它的结构， 特别是当数据的嵌套结构层次很深或者包含大量的字段时。 为了解决这个问题，可以考虑使用pprint模块的 `pprint()` 函数来代替普通的 `print()` 函数。 它会按照key的字母顺序并以一种更加美观的方式输出。 下面是一个演示如何漂亮的打印输出Twitter上搜索结果的例子：

```
>>> from urllib.request import urlopen
>>> import json
>>> u = urlopen('http://search.twitter.com/search.json?q=python&rpp=5')
```

```
>>> resp = json.loads(u.read().decode('utf-8'))
>>> from pprint import pprint
>>> pprint(resp)
{'completed_in': 0.074,
'max_id': 264043230692245504,
'max_id_str': '264043230692245504',
'next_page': '?page=2&max_id=264043230692245504&q=python&rpp=5',
'page': 1,
'query': 'python',
'refresh_url': '?since_id=264043230692245504&q=python',
'results': [{'created_at': 'Thu, 01 Nov 2012 16:36:26 +0000',
            'from_user': ...
            },
            {'created_at': 'Thu, 01 Nov 2012 16:36:14 +0000',
            'from_user': ...
            },
            {'created_at': 'Thu, 01 Nov 2012 16:36:13 +0000',
            'from_user': ...
            },
            {'created_at': 'Thu, 01 Nov 2012 16:36:07 +0000',
            'from_user': ...
            }
            {'created_at': 'Thu, 01 Nov 2012 16:36:04 +0000',
            'from_user': ...
            }],
'results_per_page': 5,
'since_id': 0,
'since_id_str': '0'}
>>>
```

一般来讲，JSON解码会根据提供的数据创建dicts或lists。 如果你想要创建其他类型的对象，可以给 `json.loads()` 传递object_pairs_hook或object_hook参数。 例如，下面是演示如何解码JSON数据并在一个OrderedDict中保留其顺序的例子：

```
>>> s = '{"name": "ACME", "shares": 50, "price": 490.1}'
>>> from collections import OrderedDict
>>> data = json.loads(s, object_pairs_hook=OrderedDict)
>>> data
OrderedDict([('name', 'ACME'), ('shares', 50), ('price', 490.1)])
>>>
```

下面是如何将一个JSON字典转换为一个Python对象例子：

```
>>> class JSONObject:
...     def __init__(self, d):
...         self.__dict__ = d
...
>>>
>>> data = json.loads(s, object_hook=JSONObject)
>>> data.name
'ACME'
>>> data.shares
50
>>> data.price
490.1
>>>
```

最后一个例子中，JSON解码后的字典作为一个单个参数传递给 `__init__()` 。 然后，你就可以随心所欲的使用它了，比如作为一个实例字典来直接使用它。

在编码JSON的时候，还有一些选项很有用。 如果你想获得漂亮的格式化字符串后输出，可以使用 `json.dumps()` 的indent参数。 它会使得输出和pprint()函数效果类似。比如：

```
>>> print(json.dumps(data))
{"price": 542.23, "name": "ACME", "shares": 100}
>>> print(json.dumps(data, indent=4))
{
    "price": 542.23,
    "name": "ACME",
    "shares": 100
```

```
}
>>>
```

对象实例通常并不是JSON可序列化的。例如：

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> p = Point(2, 3)
>>> json.dumps(p)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/usr/local/lib/python3.3/json/__init__.py", line 226, in dumps
        return _default_encoder.encode(obj)
    File "/usr/local/lib/python3.3/json/encoder.py", line 187, in encode
        chunks = self.iterencode(o, _one_shot=True)
    File "/usr/local/lib/python3.3/json/encoder.py", line 245, in iterencode
        return _iterencode(o, 0)
    File "/usr/local/lib/python3.3/json/encoder.py", line 169, in default
        raise TypeError(repr(o) + " is not JSON serializable")
TypeError: <__main__.Point object at 0x1006f2650> is not JSON serializable
>>>
```

如果你想序列化对象实例，你可以提供一个函数，它的输入是一个实例，返回一个可序列化的字典。例如：

```
def serialize_instance(obj):
    d = { '__classname__' : type(obj).__name__ }
    d.update(vars(obj))
    return d
```

如果你想反过来获取这个实例，可以这样做：

```
# Dictionary mapping names to known classes
classes = {
    'Point' : Point
}

def unserialize_object(d):
    clsname = d.pop('__classname__', None)
    if clsname:
        cls = classes[clsname]
        obj = cls.__new__(cls) # Make instance without calling __init__
        for key, value in d.items():
            setattr(obj, key, value)
        return obj
    else:
        return d
```

下面是如何使用这些函数的例子：

```
>>> p = Point(2,3)
>>> s = json.dumps(p, default=serialize_instance)
>>> s
'{"__classname__": "Point", "y": 3, "x": 2}'
>>> a = json.loads(s, object_hook=unserialize_object)
>>> a
<__main__.Point object at 0x1017577d0>
>>> a.x
2
>>> a.y
3
>>>
```

`json` 模块还有很多其他选项来控制更低级别的数字、特殊值如NaN等的解析。 可以参考官方文档获取更多细节。

6.3 解析简单的XML数据¶

问题¶

你想从一个简单的XML文档中提取数据。

解决方案¶

可以使用 `xml.etree.ElementTree` 模块从简单的XML文档中提取数据。 为了演示，假设你想解析Planet Python上的RSS源。下面是相应的代码：

```
from urllib.request import urlopen
from xml.etree.ElementTree import parse

# Download the RSS feed and parse it
u = urlopen('http://planet.python.org/rss20.xml')
doc = parse(u)

# Extract and output tags of interest
for item in doc.iterfind('channel/item'):
    title = item.findtext('title')
    date = item.findtext('pubDate')
    link = item.findtext('link')

    print(title)
    print(date)
    print(link)
    print()
```

运行上面的代码，输出结果类似这样：

```
Steve Holden: Python for Data Analysis
Mon, 19 Nov 2012 02:13:51 +0000
http://holdenweb.blogspot.com/2012/11/python-for-data-analysis.html

Vasudev Ram: The Python Data model (for v2 and v3)
Sun, 18 Nov 2012 22:06:47 +0000
http://jugad2.blogspot.com/2012/11/the-python-data-model.html

Python Diary: Been playing around with Object Databases
Sun, 18 Nov 2012 20:40:29 +0000
http://www.pythondiary.com/blog/Nov.18,2012/been-...-object-databases.html

Vasudev Ram: Wakari, Scientific Python in the cloud
Sun, 18 Nov 2012 20:19:41 +0000
http://jugad2.blogspot.com/2012/11/wakari-scientific-python-in-cloud.html

Jesse Jiryu Davis: Toro: synchronization primitives for Tornado coroutines
Sun, 18 Nov 2012 20:17:49 +0000
http://feedproxy.google.com/~r/EmptysquarePython/~3/_DOZT2Kd0hQ/
```

很显然，如果你想做进一步的处理，你需要替换 `print()` 语句来完成其他有趣的事。

讨论¶

在很多应用程序中处理XML编码格式的数据是很常见的。 不仅因为XML在Internet上面已经被广泛应用于数据交换，同时它也是一种存储应用程序数据的常用格式(比如字处理，音乐库等)。 接下来的讨论会先假定读者已经对XML基础比较熟悉了。

在很多情况下，当使用XML来仅仅存储数据的时候，对应的文档结构非常紧凑并且直观。 例如，上面例子中的RSS订阅源类似于下面的格式：

```
<?xml version="1.0"?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
```

```
    <channel>
        <title>Planet Python</title>
        <link>http://planet.python.org/</link>
        <language>en</language>
        <description>Planet Python - http://planet.python.org/</description>
        <item>
            <title>Steve Holden: Python for Data Analysis</title>
            <guid>http://holdenweb.blogspot.com/...-data-analysis.html</guid>
            <link>http://holdenweb.blogspot.com/...-data-analysis.html</link>
            <description>...</description>
            <pubDate>Mon, 19 Nov 2012 02:13:51 +0000</pubDate>
        </item>
        <item>
            <title>Vasudev Ram: The Python Data model (for v2 and v3)</title>
            <guid>http://jugad2.blogspot.com/...-data-model.html</guid>
            <link>http://jugad2.blogspot.com/...-data-model.html</link>
            <description>...</description>
            <pubDate>Sun, 18 Nov 2012 22:06:47 +0000</pubDate>
        </item>
        <item>
            <title>Python Diary: Been playing around with Object Databases</title>
            <guid>http://www.pythondiary.com/...-object-databases.html</guid>
            <link>http://www.pythondiary.com/...-object-databases.html</link>
            <description>...</description>
            <pubDate>Sun, 18 Nov 2012 20:40:29 +0000</pubDate>
        </item>
        ...
    </channel>
</rss>
```

`xml.etree.ElementTree.parse()` 函数解析整个XML文档并将其转换成一个文档对象。 然后，你就能使用 `find()`、`iterfind()` 和 `findtext()` 等方法来搜索特定的XML元素了。 这些函数的参数就是某个指定的标签名，例如 `channel/item` 或 `title` 。

每次指定某个标签时，你需要遍历整个文档结构。每次搜索操作会从一个起始元素开始进行。 同样，每次操作所指定的标签名也是起始元素的相对路径。 例如，执行 `doc.iterfind('channel/item')` 来搜索所有在 `channel` 元素下面的 `item` 元素。 `doc` 代表文档的最顶层(也就是第一级的 `rss` 元素)。 然后接下来的调用 `item.findtext()` 会从已找到的 `item` 元素位置开始搜索。

`ElementTree` 模块中的每个元素有一些重要的属性和方法，在解析的时候非常有用。 `tag` 属性包含了标签的名字，`text` 属性包含了内部的文本，而 `get()` 方法能获取属性值。例如：

```
>>> doc
<xml.etree.ElementTree.ElementTree object at 0x101339510>
>>> e = doc.find('channel/title')
>>> e
<Element 'title' at 0x10135b310>
>>> e.tag
'title'
>>> e.text
'Planet Python'
>>> e.get('some_attribute')
>>>
```

有一点要强调的是 `xml.etree.ElementTree` 并不是XML解析的唯一方法。 对于更高级的应用程序，你需要考虑使用 `lxml` 。 它使用了和ElementTree同样的编程接口，因此上面的例子同样也适用于lxml。 你只需要将刚开始的import语句换成 `from lxml.etree import parse` 就行了。 `lxml` 完全遵循XML标准，并且速度也非常快，同时还支持验证，XSLT，和XPath等特性。

6.4 增量式解析大型XML文件¶

问题¶

你想使用尽可能少的内存从一个超大的XML文档中提取数据。

解决方案¶

任何时候只要你遇到增量式的数据处理时，第一时间就应该想到迭代器和生成器。 下面是一个很简单的函数，只使用很少的内存就能增量式的处理一个大型XML文件：

```
from xml.etree.ElementTree import iterparse

def parse_and_remove(filename, path):
    path_parts = path.split('/')
    doc = iterparse(filename, ('start', 'end'))
    # Skip the root element
    next(doc)

    tag_stack = []
    elem_stack = []
    for event, elem in doc:
        if event == 'start':
            tag_stack.append(elem.tag)
            elem_stack.append(elem)
        elif event == 'end':
            if tag_stack == path_parts:
                yield elem
                elem_stack[-2].remove(elem)
            try:
                tag_stack.pop()
                elem_stack.pop()
            except IndexError:
                pass
```

为了测试这个函数，你需要先有一个大型的XML文件。 通常你可以在政府网站或公共数据网站上找到这样的文件。 例如，你可以下载XML格式的芝加哥城市道路坑洼数据库。 在写这本书的时候，下载文件已经包含超过100,000行数据，编码格式类似于下面这样：

```
<response>
    <row>
        <row ...>
            <creation_date>2012-11-18T00:00:00</creation_date>
            <status>Completed</status>
            <completion_date>2012-11-18T00:00:00</completion_date>
            <service_request_number>12-01906549</service_request_number>
            <type_of_service_request>Pot Hole in Street</type_of_service_request>
            <current_activity>Final Outcome</current_activity>
            <most_recent_action>CDOT Street Cut ... Outcome</most_recent_action>
            <street_address>4714 S TALMAN AVE</street_address>
            <zip>60632</zip>
            <x_coordinate>1159494.68618856</x_coordinate>
            <y_coordinate>1873313.83503384</y_coordinate>
            <ward>14</ward>
            <police_district>9</police_district>
            <community_area>58</community_area>
            <latitude>41.808090232127896</latitude>
            <longitude>-87.69053684711305</longitude>
            <location latitude="41.808090232127896"
            longitude="-87.69053684711305" />
        </row>
        <row ...>
            <creation_date>2012-11-18T00:00:00</creation_date>
            <status>Completed</status>
            <completion_date>2012-11-18T00:00:00</completion_date>
            <service_request_number>12-01906695</service_request_number>
```

```
            <type_of_service_request>Pot Hole in Street</type_of_service_request>
            <current_activity>Final Outcome</current_activity>
            <most_recent_action>CDOT Street Cut ... Outcome</most_recent_action>
            <street_address>3510 W NORTH AVE</street_address>
            <zip>60647</zip>
            <x_coordinate>1152732.14127696</x_coordinate>
            <y_coordinate>1910409.38979075</y_coordinate>
            <ward>26</ward>
            <police_district>14</police_district>
            <community_area>23</community_area>
            <latitude>41.91002084292946</latitude>
            <longitude>-87.71435952353961</longitude>
            <location latitude="41.91002084292946"
            longitude="-87.71435952353961" />
        </row>
    </row>
</response>
```

假设你想写一个脚本来按照坑洼报告数量排列邮编号码。你可以像这样做：

```
from xml.etree.ElementTree import parse
from collections import Counter

potholes_by_zip = Counter()

doc = parse('potholes.xml')
for pothole in doc.iterfind('row/row'):
    potholes_by_zip[pothole.findtext('zip')] += 1
for zipcode, num in potholes_by_zip.most_common():
    print(zipcode, num)
```

这个脚本唯一的问题是它会先将整个XML文件加载到内存中然后解析。 在我的机器上，为了运行这个程序需要用到450MB左右的内存空间。 如果使用如下代码，程序只需要修改一点点：

```
from collections import Counter

potholes_by_zip = Counter()

data = parse_and_remove('potholes.xml', 'row/row')
for pothole in data:
    potholes_by_zip[pothole.findtext('zip')] += 1
for zipcode, num in potholes_by_zip.most_common():
    print(zipcode, num)
```

结果是：这个版本的代码运行时只需要7MB的内存–大大节约了内存资源。

讨论¶

这一节的技术会依赖 `ElementTree` 模块中的两个核心功能。 第一，`iterparse()` 方法允许对XML文档进行增量操作。 使用时，你需要提供文件名和一个包含下面一种或多种类型的事件列表： `start`，`end`，`start-ns` 和 `end-ns` 。 由 `iterparse()` 创建的迭代器会产生形如 `(event, elem)` 的元组， 其中 `event` 是上述事件列表中的某一个，而 `elem` 是相应的XML元素。例如：

```
>>> data = iterparse('potholes.xml',('start','end'))
>>> next(data)
('start', <Element 'response' at 0x100771d60>)
>>> next(data)
('start', <Element 'row' at 0x100771e68>)
>>> next(data)
('start', <Element 'row' at 0x100771fc8>)
>>> next(data)
('start', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('end', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('start', <Element 'status' at 0x1006a7f18>)
>>> next(data)
('end', <Element 'status' at 0x1006a7f18>)
```

```
>>>
```

start 事件在某个元素第一次被创建并且还没有被插入其他数据(如子元素)时被创建。 而 end 事件在某个元素已经完成时被创建。 尽管没有在例子中演示， start-ns 和 end-ns 事件被用来处理XML文档命名空间的声明。

这本节例子中， start 和 end 事件被用来管理元素和标签栈。 栈代表了文档被解析时的层次结构， 还被用来判断某个元素是否匹配传给函数 parse_and_remove() 的路径。 如果匹配，就利用 yield 语句向调用者返回这个元素。

在 yield 之后的下面这个语句才是使得程序占用极少内存的ElementTree的核心特性：

```
elem_stack[-2].remove(elem)
```

这个语句使得之前由 yield 产生的元素从它的父节点中删除掉。 假设已经没有其它的地方引用这个元素了，那么这个元素就被销毁并回收内存。

对节点的迭代式解析和删除的最终效果就是一个在文档上高效的增量式清扫过程。 文档树结构从始自终没被完整的创建过。尽管如此，还是能通过上述简单的方式来处理这个XML数据。

这种方案的主要缺陷就是它的运行性能了。 我自己测试的结果是，读取整个文档到内存中的版本的运行速度差不多是增量式处理版本的两倍快。 但是它却使用了超过后者60倍的内存。 因此，如果你更关心内存使用量的话，那么增量式的版本完胜。

6.5 将字典转换为XML¶

问题¶

你想使用一个Python字典存储数据，并将它转换成XML格式。

解决方案¶

尽管 `xml.etree.ElementTree` 库通常用来做解析工作，其实它也可以创建XML文档。 例如，考虑如下这个函数：

```
from xml.etree.ElementTree import Element

def dict_to_xml(tag, d):
'''
Turn a simple dict of key/value pairs into XML
'''
elem = Element(tag)
for key, val in d.items():
    child = Element(key)
    child.text = str(val)
    elem.append(child)
return elem
```

下面是一个使用例子：

```
>>> s = { 'name': 'GOOG', 'shares': 100, 'price':490.1 }
>>> e = dict_to_xml('stock', s)
>>> e
<Element 'stock' at 0x1004b64c8>
>>>
```

转换结果是一个 `Element` 实例。对于I/O操作，使用 `xml.etree.ElementTree` 中的 `tostring()` 函数很容易就能将它转换成一个字节字符串。例如：

```
>>> from xml.etree.ElementTree import tostring
>>> tostring(e)
b'<stock><price>490.1</price><shares>100</shares><name>GOOG</name></stock>'
>>>
```

如果你想给某个元素添加属性值，可以使用 `set()` 方法：

```
>>> e.set('_id','1234')
>>> tostring(e)
b'<stock _id="1234"><price>490.1</price><shares>100</shares><name>GOOG</name>
</stock>'
>>>
```

如果你还想保持元素的顺序，可以考虑构造一个 `OrderedDict` 来代替一个普通的字典。请参考1.7小节。

讨论¶

当创建XML的时候，你被限制只能构造字符串类型的值。例如：

```
def dict_to_xml_str(tag, d):
    '''
    Turn a simple dict of key/value pairs into XML
    '''
    parts = ['<{}>'.format(tag)]
    for key, val in d.items():
        parts.append('<{0}>{1}</{0}>'.format(key,val))
    parts.append('</{}>'.format(tag))
    return ''.join(parts)
```

问题是如果你手动的去构造的时候可能会碰到一些麻烦。例如，当字典的值中包含一些特殊字符的时候会怎样呢？

```
>>> d = { 'name' : '<spam>' }

>>> # String creation
>>> dict_to_xml_str('item',d)
'<item><name><spam></name></item>'

>>> # Proper XML creation
>>> e = dict_to_xml('item',d)
>>> tostring(e)
b'<item><name>&lt;spam&gt;</name></item>'
>>>
```

注意到程序的后面那个例子中，字符 '<' 和 '>' 被替换成了 `<` 和 `>`

下面仅供参考，如果你需要手动去转换这些字符， 可以使用 `xml.sax.saxutils` 中的 `escape()` 和 `unescape()` 函数。例如：

```
>>> from xml.sax.saxutils import escape, unescape
>>> escape('<spam>')
'&lt;spam&gt;'
>>> unescape(_)
'<spam>'
>>>
```

除了能创建正确的输出外，还有另外一个原因推荐你创建 `Element` 实例而不是字符串， 那就是使用字符串组合构造一个更大的文档并不是那么容易。 而 `Element` 实例可以不用考虑解析XML文本的情况下通过多种方式被处理。 也就是说，你可以在一个高级数据结构上完成你所有的操作，并在最后以字符串的形式将其输出。

6.6 解析和修改XML¶

问题¶

你想读取一个XML文档，对它最一些修改，然后将结果写回XML文档。

解决方案¶

使用 `xml.etree.ElementTree` 模块可以很容易的处理这些任务。 第一步是以通常的方式来解析这个文档。例如，假设你有一个名为 `pred.xml` 的文档，类似下面这样：

```
<?xml version="1.0"?>
<stop>
    <id>14791</id>
    <nm>Clark &amp; Balmoral</nm>
    <sri>
        <rt>22</rt>
        <d>North Bound</d>
        <dd>North Bound</dd>
    </sri>
    <cr>22</cr>
    <pre>
        <pt>5 MIN</pt>
        <fd>Howard</fd>
        <v>1378</v>
        <rn>22</rn>
    </pre>
    <pre>
        <pt>15 MIN</pt>
        <fd>Howard</fd>
        <v>1867</v>
        <rn>22</rn>
    </pre>
</stop>
```

下面是一个利用 `ElementTree` 来读取这个文档并对它做一些修改的例子：

```
>>> from xml.etree.ElementTree import parse, Element
>>> doc = parse('pred.xml')
>>> root = doc.getroot()
>>> root
<Element 'stop' at 0x100770cb0>

>>> # Remove a few elements
>>> root.remove(root.find('sri'))
>>> root.remove(root.find('cr'))
>>> # Insert a new element after <nm>...</nm>
>>> root.getchildren().index(root.find('nm'))
1
>>> e = Element('spam')
>>> e.text = 'This is a test'
>>> root.insert(2, e)

>>> # Write back to a file
>>> doc.write('newpred.xml', xml_declaration=True)
>>>
```

处理结果是一个像下面这样新的XML文件：

```
<?xml version='1.0' encoding='us-ascii'?>
<stop>
    <id>14791</id>
    <nm>Clark &amp; Balmoral</nm>
    <spam>This is a test</spam>
    <pre>
        <pt>5 MIN</pt>
```

```
        <fd>Howard</fd>
        <v>1378</v>
        <rn>22</rn>
    </pre>
    <pre>
        <pt>15 MIN</pt>
        <fd>Howard</fd>
        <v>1867</v>
        <rn>22</rn>
    </pre>
</stop>
```

讨论¶

修改一个XML文档结构是很容易的，但是你必须牢记的是所有的修改都是针对父节点元素， 将它作为一个列表来处理。例如，如果你删除某个元素，通过调用父节点的 `remove()` 方法从它的直接父节点中删除。 如果你插入或增加新的元素，你同样使用父节点元素的 `insert()` 和 `append()` 方法。 还能对元素使用索引和切片操作，比如 `element[i]` 或 `element[i:j]`

如果你需要创建新的元素，可以使用本节方案中演示的 `Element` 类。我们在6.5小节已经详细讨论过了。

6.7 利用命名空间解析XML文档¶

问题¶

你想解析某个XML文档，文档中使用了XML命名空间。

解决方案¶

考虑下面这个使用了命名空间的文档：

```
<?xml version="1.0" encoding="utf-8"?>
<top>
    <author>David Beazley</author>
    <content>
        <html xmlns="http://www.w3.org/1999/xhtml">
            <head>
                <title>Hello World</title>
            </head>
            <body>
                <h1>Hello World!</h1>
            </body>
        </html>
    </content>
</top>
```

如果你解析这个文档并执行普通的查询，你会发现这个并不是那么容易，因为所有步骤都变得相当的繁琐。

```
>>> # Some queries that work
>>> doc.findtext('author')
'David Beazley'
>>> doc.find('content')
<Element 'content' at 0x100776ec0>
>>> # A query involving a namespace (doesn't work)
>>> doc.find('content/html')
>>> # Works if fully qualified
>>> doc.find('content/{http://www.w3.org/1999/xhtml}html')
<Element '{http://www.w3.org/1999/xhtml}html' at 0x1007767e0>
>>> # Doesn't work
>>> doc.findtext('content/{http://www.w3.org/1999/xhtml}html/head/title')
>>> # Fully qualified
>>> doc.findtext('content/{http://www.w3.org/1999/xhtml}html/'
... '{http://www.w3.org/1999/xhtml}head/{http://www.w3.org/1999/xhtml}title')
'Hello World'
>>>
```

你可以通过将命名空间处理逻辑包装为一个工具类来简化这个过程：

```
class XMLNamespaces:
    def __init__(self, **kwargs):
        self.namespaces = {}
        for name, uri in kwargs.items():
            self.register(name, uri)
    def register(self, name, uri):
        self.namespaces[name] = '{'+uri+'}'
    def __call__(self, path):
        return path.format_map(self.namespaces)
```

通过下面的方式使用这个类：

```
>>> ns = XMLNamespaces(html='http://www.w3.org/1999/xhtml')
>>> doc.find(ns('content/{html}html'))
<Element '{http://www.w3.org/1999/xhtml}html' at 0x1007767e0>
>>> doc.findtext(ns('content/{html}html/{html}head/{html}title'))
'Hello World'
>>>
```

讨论¶

解析含有命名空间的XML文档会比较繁琐。 上面的 `XMLNamespaces` 仅仅是允许你使用缩略名代替完整的URI将其变得稍微简洁一点。

很不幸的是，在基本的 `ElementTree` 解析中没有任何途径获取命名空间的信息。 但是，如果你使用 `iterparse()` 函数的话就可以获取更多关于命名空间处理范围的信息。例如：

```
>>> from xml.etree.ElementTree import iterparse
>>> for evt, elem in iterparse('ns2.xml', ('end', 'start-ns', 'end-ns')):
... print(evt, elem)
...
end <Element 'author' at 0x10110de10>
start-ns ('', 'http://www.w3.org/1999/xhtml')
end <Element '{http://www.w3.org/1999/xhtml}title' at 0x1011131b0>
end <Element '{http://www.w3.org/1999/xhtml}head' at 0x1011130a8>
end <Element '{http://www.w3.org/1999/xhtml}h1' at 0x101113310>
end <Element '{http://www.w3.org/1999/xhtml}body' at 0x101113260>
end <Element '{http://www.w3.org/1999/xhtml}html' at 0x10110df70>
end-ns None
end <Element 'content' at 0x10110de68>
end <Element 'top' at 0x10110dd60>
>>> elem # This is the topmost element
<Element 'top' at 0x10110dd60>
>>>
```

最后一点，如果你要处理的XML文本除了要使用到其他高级XML特性外，还要使用到命名空间， 建议你最好是使用 `lxml` 函数库来代替 `ElementTree` 。 例如，`lxml` 对利用DTD验证文档、更好的XPath支持和一些其他高级XML特性等都提供了更好的支持。 这一小节其实只是教你如何让XML解析稍微简单一点。

6.8 与关系型数据库的交互¶

问题¶

你想在关系型数据库中查询、增加或删除记录。

解决方案¶

Python中表示多行数据的标准方式是一个由元组构成的序列。例如：

```
stocks = [
    ('GOOG', 100, 490.1),
    ('AAPL', 50, 545.75),
    ('FB', 150, 7.45),
    ('HPQ', 75, 33.2),
]
```

依据PEP249，通过这种形式提供数据， 可以很容易的使用Python标准数据库API和关系型数据库进行交互。 所有数据库上的操作都通过SQL查询语句来完成。每一行输入输出数据用一个元组来表示。

为了演示说明，你可以使用Python标准库中的 `sqlite3` 模块。 如果你使用的是一个不同的数据库(比如MySql、Postgresql或者ODBC)， 还得安装相应的第三方模块来提供支持。 不过相应的编程接口几乎都是一样的，除了一点点细微差别外。

第一步是连接到数据库。通常你要执行 `connect()` 函数， 给它提供一些数据库名、主机、用户名、密码和其他必要的一些参数。例如：

```
>>> import sqlite3
>>> db = sqlite3.connect('database.db')
>>>
```

为了处理数据，下一步你需要创建一个游标。 一旦你有了游标，那么你就可以执行SQL查询语句了。比如：

```
>>> c = db.cursor()
>>> c.execute('create table portfolio (symbol text, shares integer, price real)')
<sqlite3.Cursor object at 0x10067a730>
>>> db.commit()
>>>
```

为了向数据库表中插入多条记录，使用类似下面这样的语句：

```
>>> c.executemany('insert into portfolio values (?,?,?)', stocks)
<sqlite3.Cursor object at 0x10067a730>
>>> db.commit()
>>>
```

为了执行某个查询，使用像下面这样的语句：

```
>>> for row in db.execute('select * from portfolio'):
...     print(row)
...
('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
('FB', 150, 7.45)
('HPQ', 75, 33.2)
>>>
```

如果你想接受用户输入作为参数来执行查询操作，必须确保你使用下面这样的占位符``?``来进行引用参数：

```
>>> min_price = 100
>>> for row in db.execute('select * from portfolio where price >= ?',
                          (min_price,)):
...     print(row)
...
```

```
('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
>>>
```

讨论¶

在比较低的级别上和数据库交互是非常简单的。 你只需提供SQL语句并调用相应的模块就可以更新或提取数据了。 虽说如此，还是有一些比较棘手的细节问题需要你逐个列出去解决。

一个难点是数据库中的数据和Python类型直接的映射。 对于日期类型，通常可以使用 `datetime` 模块中的 `datetime` 实例， 或者可能是 `time` 模块中的系统时间戳。 对于数字类型，特别是使用到小数的金融数据，可以用 `decimal` 模块中的 `Decimal` 实例来表示。 不幸的是，对于不同的数据库而言具体映射规则是不一样的，你必须参考相应的文档。

另外一个更加复杂的问题就是SQL语句字符串的构造。 你千万不要使用Python字符串格式化操作符(如%)或者 `.format()` 方法来创建这样的字符串。 如果传递给这些格式化操作符的值来自于用户的输入，那么你的程序就很有可能遭受SQL注入攻击(参考 http://xkcd.com/327)。 查询语句中的通配符 `?` 指示后台数据库使用它自己的字符串替换机制，这样更加的安全。

不幸的是，不同的数据库后台对于通配符的使用是不一样的。大部分模块使用 `?` 或 `%s` ， 还有其他一些使用了不同的符号，比如:0或:1来指示参数。 同样的，你还是得去参考你使用的数据库模块相应的文档。 一个数据库模块的 `paramstyle` 属性包含了参数引用风格的信息。

对于简单的数据库数据的读写问题，使用数据库API通常非常简单。 如果你要处理更加复杂的问题，建议你使用更加高级的接口，比如一个对象关系映射ORM所提供的接口。 类似 `SQLAlchemy` 这样的库允许你使用Python类来表示一个数据库表， 并且能在隐藏底层SQL的情况下实现各种数据库的操作。

6.9 编码和解码十六进制数¶

问题¶

你想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串。

解决方案¶

如果你只是简单的解码或编码一个十六进制的原始字符串，可以使用 `binascii` 模块。例如：

```
>>> # Initial byte string
>>> s = b'hello'
>>> # Encode as hex
>>> import binascii
>>> h = binascii.b2a_hex(s)
>>> h
b'68656c6c6f'
>>> # Decode back to bytes
>>> binascii.a2b_hex(h)
b'hello'
>>>
```

类似的功能同样可以在 `base64` 模块中找到。例如：

```
>>> import base64
>>> h = base64.b16encode(s)
>>> h
b'68656C6C6F'
>>> base64.b16decode(h)
b'hello'
>>>
```

讨论¶

大部分情况下，通过使用上述的函数来转换十六进制是很简单的。 上面两种技术的主要不同在于大小写的处理。 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。

还有一点需要注意的是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，你需要增加一个额外的界面步骤。例如：

```
>>> h = base64.b16encode(s)
>>> print(h)
b'68656C6C6F'
>>> print(h.decode('ascii'))
68656C6C6F
>>>
```

在解码十六进制数时，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。

第六章：数据编码和处理¶

这一章主要讨论使用Python处理各种不同方式编码的数据，比如CSV文件，JSON，XML和二进制包装记录。 和数据结构那一章不同的是，这章不会讨论特殊的算法问题，而是关注于怎样获取和存储这些格式的数据。

Contents: