# 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。