5.1 读写文本数据¶

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。

解决方案¶

使用带有 `rt` 模式的 `open()` 函数读取文本文件。如下所示：

```
# Read the entire file as a single string
with open('somefile.txt', 'rt') as f:
    data = f.read()

# Iterate over the lines of the file
with open('somefile.txt', 'rt') as f:
    for line in f:
        # process line
        ...
```

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。如下所示：

```
# Write chunks of text data
with open('somefile.txt', 'wt') as f:
    f.write(text1)
    f.write(text2)
    ...

# Redirected print statement
with open('somefile.txt', 'wt') as f:
    print(line1, file=f)
    print(line2, file=f)
    ...
```

如果是在已存在文件中添加内容，使用模式为 `at` 的 `open()` 函数。

文件的读写操作默认使用系统编码，可以通过调用 `sys.getdefaultencoding()` 来得到。 在大多数机器上面都是utf-8编码。如果你已经知道你要读写的文本是其他编码方式， 那么可以通过传递一个可选的 `encoding` 参数给open()函数。如下所示：

```
with open('somefile.txt', 'rt', encoding='latin-1') as f:
    ...
```

Python支持非常多的文本编码。几个常见的编码是ascii, latin-1, utf-8和utf-16。 在web应用程序中通常都使用的是UTF-8。ascii对应从U+0000到U+007F范围内的7位字符。 latin-1是字节0-255到U+0000至U+00FF范围内Unicode字符的直接映射。当读取一个未知编码的文本时使用latin-1编码永远不会产生解码错误。 使用latin-1编码读取一个文件的时候也许不能产生完全正确的文本解码数据， 但是它也能从中提取出足够多的有用数据。同时，如果你之后将数据回写回去，原先的数据还是会保留的。

讨论¶

读写文本文件一般来讲是比较简单的。但是也几点是需要注意的。 首先，在例子程序中的with语句给被使用到的文件创建了一个上下文环境， 但 `with` 控制块结束时，文件会自动关闭。你也可以不使用 `with` 语句，但是这时候你就必须记得手动关闭文件：

```
f = open('somefile.txt', 'rt')
data = f.read()
f.close()
```

另外一个问题是关于换行符的识别问题，在Unix和Windows中是不一样的(分别是 `\n` 和 `\r\n`)。 默认情况下，Python会以统一模式处理换行符。 这种模式下，在读取文本的时候，Python可以识别所有的普通换行符并将其转换为单个 `\n` 字

符。 类似的，在输出时会将换行符 `\n` 转换为系统默认的换行符。 如果你不希望这种默认的处理方式，可以给 `open()` 函数传入参数 `newline=''`，就像下面这样：

```
# Read with disabled newline translation
with open('somefile.txt', 'rt', newline='') as f:
    ...
```

为了说明两者之间的差异，下面我在Unix机器上面读取一个Windows上面的文本文件，里面的内容是 `hello world!\r\n`：

```
>>> # Newline translation enabled (the default)
>>> f = open('hello.txt', 'rt')
>>> f.read()
'hello world!\n'

>>> # Newline translation disabled
>>> g = open('hello.txt', 'rt', newline='')
>>> g.read()
'hello world!\r\n'
>>>
```

最后一个问题就是文本文件中可能出现的编码错误。 但你读取或者写入一个文本文件时，你可能会遇到一个编码或者解码错误。比如：

```
>>> f = open('sample.txt', 'rt', encoding='ascii')
>>> f.read()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/usr/local/lib/python3.3/encodings/ascii.py", line 26, in decode
        return codecs.ascii_decode(input, self.errors)[0]
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position
12: ordinal not in range(128)
>>>
```

如果出现这个错误，通常表示你读取文本时指定的编码不正确。 你最好仔细阅读说明并确认你的文件编码是正确的(比如使用UTF-8而不是Latin-1编码或其他)。 如果编码错误还是存在的话，你可以给 `open()` 函数传递一个可选的 `errors` 参数来处理这些错误。 下面是一些处理常见错误的方法：

```
>>> # Replace bad chars with Unicode U+fffd replacement char
>>> f = open('sample.txt', 'rt', encoding='ascii', errors='replace')
>>> f.read()
'Spicy Jalape?o!'
>>> # Ignore bad chars entirely
>>> g = open('sample.txt', 'rt', encoding='ascii', errors='ignore')
>>> g.read()
'Spicy Jalapeo!'
>>>
```

如果你经常使用 `errors` 参数来处理编码错误，可能会让你的生活变得很糟糕。 对于文本处理的首要原则是确保你总是使用的是正确编码。当模棱两可的时候，就使用默认的设置(通常都是UTF-8)。

5.10 内存映射的二进制文件¶

问题¶

你想内存映射一个二进制文件到一个可变字节数组中，目的可能是为了随机访问它的内容或者是原地做些修改。

解决方案¶

使用 `mmap` 模块来内存映射文件。 下面是一个工具函数，向你演示了如何打开一个文件并以一种便捷方式内存映射这个文件。

```
import os
import mmap

def memory_map(filename, access=mmap.ACCESS_WRITE):
    size = os.path.getsize(filename)
    fd = os.open(filename, os.O_RDWR)
    return mmap.mmap(fd, size, access=access)
```

为了使用这个函数，你需要有一个已创建并且内容不为空的文件。 下面是一个例子，教你怎样初始创建一个文件并将其内容扩充到指定大小：

```
>>> size = 1000000
>>> with open('data', 'wb') as f:
...     f.seek(size-1)
...     f.write(b'\x00')
...
>>>
```

下面是一个利用 `memory_map()` 函数类内存映射文件内容的例子：

```
>>> m = memory_map('data')
>>> len(m)
1000000
>>> m[0:10]
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> m[0]
0
>>> # Reassign a slice
>>> m[0:11] = b'Hello World'
>>> m.close()

>>> # Verify that changes were made
>>> with open('data', 'rb') as f:
... print(f.read(11))
...
b'Hello World'
>>>
```

`mmap()` 返回的 `mmap` 对象同样也可以作为一个上下文管理器来使用， 这时候底层的文件会被自动关闭。比如：

```
>>> with memory_map('data') as m:
...     print(len(m))
...     print(m[0:10])
...
1000000
b'Hello World'
>>> m.closed
True
>>>
```

默认情况下， `memeory_map()` 函数打开的文件同时支持读和写操作。 任何的修改内容都会复制回原来的文件中。 如果需要只读的访问模式，可以给参数 `access` 赋值为 `mmap.ACCESS_READ` 。比如：

```
m = memory_map(filename, mmap.ACCESS_READ)
```

如果你想在本地修改数据，但是又不想将修改写回到原始文件中，可以使用 `mmap.ACCESS_COPY`：

```
m = memory_map(filename, mmap.ACCESS_COPY)
```

讨论¶

为了随机访问文件的内容，使用 `mmap` 将文件映射到内存中是一个高效和优雅的方法。 例如，你无需打开一个文件并执行大量的 `seek()`， `read()`， `write()` 调用， 只需要简单的映射文件并使用切片操作访问数据即可。

一般来讲， `mmap()` 所暴露的内存看上去就是一个二进制数组对象。 但是，你可以使用一个内存视图来解析其中的数据。比如：

```
>>> m = memory_map('data')
>>> # Memoryview of unsigned integers
>>> v = memoryview(m).cast('I')
>>> v[0] = 7
>>> m[0:4]
b'\x07\x00\x00\x00'
>>> m[0:4] = b'\x07\x01\x00\x00'
>>> v[0]
263
>>>
```

需要强调的一点是，内存映射一个文件并不会导致整个文件被读取到内存中。 也就是说，文件并没有被复制到内存缓存或数组中。相反，操作系统仅仅为文件内容保留了一段虚拟内存。 当你访问文件的不同区域时，这些区域的内容才根据需要被读取并映射到内存区域中。 而那些从没被访问到的部分还是留在磁盘上。所有这些过程是透明的，在幕后完成！

如果多个Python解释器内存映射同一个文件，得到的 `mmap` 对象能够被用来在解释器直接交换数据。 也就是说，所有解释器都能同时读写数据，并且其中一个解释器所做的修改会自动呈现在其他解释器中。 很明显，这里需要考虑同步的问题。但是这种方法有时候可以用来在管道或套接字间传递数据。

这一小节中函数尽量写得很通用，同时适用于Unix和Windows平台。 要注意的是使用 `mmap()` 函数时会在底层有一些平台的差异性。 另外，还有一些选项可以用来创建匿名的内存映射区域。 如果你对这个感兴趣，确保你仔细研读了Python文档中 这方面的内容 。

5.11 文件路径名的操作¶

问题¶

你需要使用路径名来获取文件名，目录名，绝对路径等等。

解决方案¶

使用 `os.path` 模块中的函数来操作路径名。 下面是一个交互式例子来演示一些关键的特性：

```
>>> import os
>>> path = '/Users/beazley/Data/data.csv'

>>> # Get the last component of the path
>>> os.path.basename(path)
'data.csv'

>>> # Get the directory name
>>> os.path.dirname(path)
'/Users/beazley/Data'

>>> # Join path components together
>>> os.path.join('tmp', 'data', os.path.basename(path))
'tmp/data/data.csv'

>>> # Expand the user's home directory
>>> path = '~/Data/data.csv'
>>> os.path.expanduser(path)
'/Users/beazley/Data/data.csv'

>>> # Split the file extension
>>> os.path.splitext(path)
('~/Data/data', '.csv')
>>>
```

讨论¶

对于任何的文件名的操作，你都应该使用 `os.path` 模块，而不是使用标准字符串操作来构造自己的代码。 特别是为了可移植性考虑的时候更应如此， 因为 `os.path` 模块知道Unix和Windows系统之间的差异并且能够可靠地处理类似 `Data/data.csv` 和 `Data\data.csv` 这样的文件名。 其次，你真的不应该浪费时间去重复造轮子。通常最好是直接使用已经为你准备好的功能。

要注意的是 `os.path` 还有更多的功能在这里并没有列举出来。 可以查阅官方文档来获取更多与文件测试，符号链接等相关的函数说明。

5.12 测试文件是否存在¶

问题¶

你想测试一个文件或目录是否存在。

解决方案¶

使用 `os.path` 模块来测试一个文件或目录是否存在。比如：

```
>>> import os
>>> os.path.exists('/etc/passwd')
True
>>> os.path.exists('/tmp/spam')
False
>>>
```

你还能进一步测试这个文件时什么类型的。 在下面这些测试中，如果测试的文件不存在的时候，结果都会返回False：

```
>>> # Is a regular file
>>> os.path.isfile('/etc/passwd')
True

>>> # Is a directory
>>> os.path.isdir('/etc/passwd')
False

>>> # Is a symbolic link
>>> os.path.islink('/usr/local/bin/python3')
True

>>> # Get the file linked to
>>> os.path.realpath('/usr/local/bin/python3')
'/usr/local/bin/python3.3'
>>>
```

如果你还想获取元数据(比如文件大小或者是修改日期)，也可以使用 `os.path` 模块来解决：

```
>>> os.path.getsize('/etc/passwd')
3669
>>> os.path.getmtime('/etc/passwd')
1272478234.0
>>> import time
>>> time.ctime(os.path.getmtime('/etc/passwd'))
'Wed Apr 28 13:10:34 2010'
>>>
```

讨论¶

使用 `os.path` 来进行文件测试是很简单的。 在写这些脚本时，可能唯一需要注意的就是你需要考虑文件权限的问题，特别是在获取元数据时候。比如：

```
>>> os.path.getsize('/Users/guido/Desktop/foo.txt')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/usr/local/lib/python3.3/genericpath.py", line 49, in getsize
        return os.stat(filename).st_size
PermissionError: [Errno 13] Permission denied: '/Users/guido/Desktop/foo.txt'
>>>
```

5.13 获取文件夹中的文件列表¶

问题¶

你想获取文件系统中某个目录下的所有文件列表。

解决方案¶

使用 `os.listdir()` 函数来获取某个目录中的文件列表：

```
import os
names = os.listdir('somedir')
```

结果会返回目录中所有文件列表，包括所有文件，子目录，符号链接等等。 如果你需要通过某种方式过滤数据，可以考虑结合 `os.path` 库中的一些函数来使用列表推导。比如：

```
import os.path

# Get all regular files
names = [name for name in os.listdir('somedir')
        if os.path.isfile(os.path.join('somedir', name))]

# Get all dirs
dirnames = [name for name in os.listdir('somedir')
        if os.path.isdir(os.path.join('somedir', name))]
```

字符串的 `startswith()` 和 `endswith()` 方法对于过滤一个目录的内容也是很有用的。比如：

```
pyfiles = [name for name in os.listdir('somedir')
            if name.endswith('.py')]
```

对于文件名的匹配，你可能会考虑使用 `glob` 或 `fnmatch` 模块。比如：

```
import glob
pyfiles = glob.glob('somedir/*.py')

from fnmatch import fnmatch
pyfiles = [name for name in os.listdir('somedir')
            if fnmatch(name, '*.py')]
```

讨论¶

获取目录中的列表是很容易的，但是其返回结果只是目录中实体名列表而已。 如果你还想获取其他的元信息，比如文件大小，修改时间等等， 你或许还需要使用到 `os.path` 模块中的函数或着 `os.stat()` 函数来收集数据。比如：

```
# Example of getting a directory listing

import os
import os.path
import glob

pyfiles = glob.glob('*.py')

# Get file sizes and modification dates
name_sz_date = [(name, os.path.getsize(name), os.path.getmtime(name))
                for name in pyfiles]
for name, size, mtime in name_sz_date:
    print(name, size, mtime)

# Alternative: Get file metadata
file_metadata = [(name, os.stat(name)) for name in pyfiles]
for name, meta in file_metadata:
    print(name, meta.st_size, meta.st_mtime)
```

最后还有一点要注意的就是，有时候在处理文件名编码问题时候可能会出现一些问题。 通常来讲，函数 `os.listdir()` 返回的实体列表会根据系统默认的文件名编码来解码。 但是有时候也会碰到一些不能正常解码的文件名。 关于文件名的处理问题，在5.14和5.15小节有更详细的讲解。

5.14 忽略文件名编码¶

问题¶

你想使用原始文件名执行文件的I/O操作，也就是说文件名并没有经过系统默认编码去解码或编码过。

解决方案¶

默认情况下，所有的文件名都会根据 `sys.getfilesystemencoding()` 返回的文本编码来编码或解码。比如：

```
>>> sys.getfilesystemencoding()
'utf-8'
>>>
```

如果因为某种原因你想忽略这种编码，可以使用一个原始字节字符串来指定一个文件名即可。比如：

```
>>> # Wrte a file using a unicode filename
>>> with open('jalape\xf1o.txt', 'w') as f:
...     f.write('Spicy!')
...
6
>>> # Directory listing (decoded)
>>> import os
>>> os.listdir('.')
['jalapeño.txt']

>>> # Directory listing (raw)
>>> os.listdir(b'.') # Note: byte string
[b'jalapen\xcc\x83o.txt']

>>> # Open file with raw filename
>>> with open(b'jalapen\xcc\x83o.txt') as f:
...     print(f.read())
...
Spicy!
>>>
```

正如你所见，在最后两个操作中，当你给文件相关函数如 `open()` 和 `os.listdir()` 传递字节字符串时，文件名的处理方式会稍有不同。

讨论¶

通常来讲，你不需要担心文件名的编码和解码，普通的文件名操作应该就没问题了。 但是，有些操作系统允许用户通过偶然或恶意方式去创建名字不符合默认编码的文件。 这些文件名可能会神秘地中断那些需要处理大量文件的Python程序。

读取目录并通过原始未解码方式处理文件名可以有效的避免这样的问题， 尽管这样会带来一定的编程难度。

关于打印不可解码的文件名，请参考5.15小节。

5.15 打印不合法的文件名¶

问题¶

你的程序获取了一个目录中的文件名列表，但是当它试着去打印文件名的时候程序崩溃， 出现了 `UnicodeEncodeError` 异常和一条奇怪的消息——`surrogates not allowed`。

解决方案¶

当打印未知的文件名时，使用下面的方法可以避免这样的错误：

```
def bad_filename(filename):
    return repr(filename)[1:-1]

try:
    print(filename)
except UnicodeEncodeError:
    print(bad_filename(filename))
```

讨论¶

这一小节讨论的是在编写必须处理文件系统的程序时一个不太常见但又很棘手的问题。 默认情况下，Python假定所有文件名都已经根据 `sys.getfilesystemencoding()` 的值编码过了。 但是，有一些文件系统并没有强制要求这样做，因此允许创建文件名没有正确编码的文件。 这种情况不太常见，但是总会有些用户冒险这样做或者是无意之中这样做了(可能是在一个有缺陷的代码中给 `open()` 函数传递了一个不合规范的文件名)。

当执行类似 `os.listdir()` 这样的函数时，这些不合规范的文件名就会让Python陷入困境。 一方面，它不能仅仅只是丢弃这些不合格的名字。而另一方面，它又不能将这些文件名转换为正确的文本字符串。 Python对这个问题的解决方案是从文件名中获取未解码的字节值比如 `\xhh` 并将它映射成Unicode字符 `\udchh` 表示的所谓的”代理编码”。 下面一个例子演示了当一个不合格目录列表中含有一个文件名为bäd.txt(使用Latin-1而不是UTF-8编码)时的样子：

```
>>> import os
>>> files = os.listdir('.')
>>> files
['spam.py', 'b\udce4d.txt', 'foo.txt']
>>>
```

如果你有代码需要操作文件名或者将文件名传递给 `open()` 这样的函数，一切都能正常工作。 只有当你想要输出文件名时才会碰到些麻烦(比如打印输出到屏幕或日志文件等)。 特别的，当你想打印上面的文件名列表时，你的程序就会崩溃：

```
>>> for name in files:
...     print(name)
...
spam.py
Traceback (most recent call last):
    File "<stdin>", line 2, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\udce4' in
position 1: surrogates not allowed
>>>
```

程序崩溃的原因就是字符 `\udce4` 是一个非法的Unicode字符。 它其实是一个被称为代理字符对的双字符组合的后半部分。 由于缺少了前半部分，因此它是个非法的Unicode。 所以，唯一能成功输出的方法就是当遇到不合法文件名时采取相应的补救措施。 比如可以将上述代码修改如下：

```
>>> for name in files:
... try:
...     print(name)
... except UnicodeEncodeError:
...     print(bad_filename(name))
...
spam.py
```

```
b\udce4d.txt
foo.txt
>>>
```

在 bad_filename() 函数中怎样处置取决于你自己。 另外一个选择就是通过某种方式重新编码，示例如下：

```
def bad_filename(filename):
    temp = filename.encode(sys.getfilesystemencoding(), errors='surrogateescape')
    return temp.decode('latin-1')
```

译者注:

```
surrogateescape:
这种是Python在绝大部分面向OS的API中所使用的错误处理器，
它能以一种优雅的方式处理由操作系统提供的数据的编码问题。
在解码出错时会将出错字节存储到一个很少被使用到的Unicode编码范围内。
在编码时将那些隐藏值又还原回原先解码失败的字节序列。
它不仅对于OS API非常有用，也能很容易的处理其他情况下的编码错误。
```

使用这个版本产生的输出如下：

```
>>> for name in files:
...     try:
...         print(name)
...     except UnicodeEncodeError:
...         print(bad_filename(name))
...
spam.py
bäd.txt
foo.txt
>>>
```

这一小节主题可能会被大部分读者所忽略。但是如果你在编写依赖文件名和文件系统的关键任务程序时， 就必须得考虑到这个。否则你可能会在某个周末被叫到办公室去调试一些令人费解的错误。

5.16 增加或改变已打开文件的编码¶

问题¶

你想在不关闭一个已打开的文件前提下增加或改变它的Unicode编码。

解决方案¶

如果你想给一个以二进制模式打开的文件添加Unicode编码/解码方式， 可以使用 `io.TextIOWrapper()` 对象包装它。比如：

```
import urllib.request
import io

u = urllib.request.urlopen('http://www.python.org')
f = io.TextIOWrapper(u, encoding='utf-8')
text = f.read()
```

如果你想修改一个已经打开的文本模式的文件的编码方式，可以先使用 `detach()` 方法移除掉已存在的文本编码层，并使用新的编码方式代替。下面是一个在 `sys.stdout` 上修改编码方式的例子：

```
>>> import sys
>>> sys.stdout.encoding
'UTF-8'
>>> sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding='latin-1')
>>> sys.stdout.encoding
'latin-1'
>>>
```

这样做可能会中断你的终端，这里仅仅是为了演示而已。

讨论¶

I/O系统由一系列的层次构建而成。你可以试着运行下面这个操作一个文本文件的例子来查看这种层次：

```
>>> f = open('sample.txt','w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f.buffer
<_io.BufferedWriter name='sample.txt'>
>>> f.buffer.raw
<_io.FileIO name='sample.txt' mode='wb'>
>>>
```

在这个例子中，`io.TextIOWrapper` 是一个编码和解码Unicode的文本处理层， `io.BufferedWriter` 是一个处理二进制数据的带缓冲的I/O层， `io.FileIO` 是一个表示操作系统底层文件描述符的原始文件。 增加或改变文本编码会涉及增加或改变最上面的 `io.TextIOWrapper` 层。

一般来讲，像上面例子这样通过访问属性值来直接操作不同的层是很不安全的。 例如，如果你试着使用下面这样的技术改变编码看看会发生什么：

```
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f = io.TextIOWrapper(f.buffer, encoding='latin-1')
>>> f
<_io.TextIOWrapper name='sample.txt' encoding='latin-1'>
>>> f.write('Hello')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>>
```

结果出错了，因为f的原始值已经被破坏了并关闭了底层的文件。

`detach()` 方法会断开文件的最顶层并返回第二层，之后最顶层就没什么用了。例如：

```
>>> f = open('sample.txt', 'w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> b = f.detach()
>>> b
<_io.BufferedWriter name='sample.txt'>
>>> f.write('hello')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: underlying buffer has been detached
>>>
```

一旦断开最顶层后，你就可以给返回结果添加一个新的最顶层。比如：

```
>>> f = io.TextIOWrapper(b, encoding='latin-1')
>>> f
<_io.TextIOWrapper name='sample.txt' encoding='latin-1'>
>>>
```

尽管已经向你演示了改变编码的方法， 但是你还可以利用这种技术来改变文件行处理、错误机制以及文件处理的其他方面。例如：

```
>>> sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding='ascii',
...                               errors='xmlcharrefreplace')
>>> print('Jalape\u00f1o')
Jalape&#241;o
>>>
```

注意下最后输出中的非ASCII字符 `ñ` 是如何被 `&#241;` 取代的。

5.17 将字节写入文本文件¶

问题¶

你想在文本模式打开的文件中写入原始的字节数据。

解决方案¶

将字节数据直接写入文件的缓冲区即可，例如：

```
>>> import sys
>>> sys.stdout.write(b'Hello\n')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
>>> sys.stdout.buffer.write(b'Hello\n')
Hello
5
>>>
```

类似的，能够通过读取文本文件的 `buffer` 属性来读取二进制数据。

讨论¶

I/O系统以层级结构的形式构建而成。 文本文件是通过在一个拥有缓冲的二进制模式文件上增加一个Unicode编码/解码层来创建。 `buffer` 属性指向对应的底层文件。如果你直接访问它的话就会绕过文本编码/解码层。

本小节例子展示的 `sys.stdout` 可能看起来有点特殊。 默认情况下，`sys.stdout` 总是以文本模式打开的。 但是如果你在写一个需要打印二进制数据到标准输出的脚本的话，你可以使用上面演示的技术来绕过文本编码层。

5.18 将文件描述符包装成文件对象¶

问题¶

你有一个对应于操作系统上一个已打开的I/O通道(比如文件、管道、套接字等)的整型文件描述符， 你想将它包装成一个更高层的Python文件对象。

解决方案¶

一个文件描述符和一个打开的普通文件是不一样的。 文件描述符仅仅是一个由操作系统指定的整数，用来指代某个系统的I/O通道。 如果你碰巧有这么一个文件描述符，你可以通过使用 `open()` 函数来将其包装为一个Python的文件对象。 你仅仅只需要使用这个整数值的文件描述符作为第一个参数来代替文件名即可。例如：

```
# Open a low-level file descriptor
import os
fd = os.open('somefile.txt', os.O_WRONLY | os.O_CREAT)

# Turn into a proper file
f = open(fd, 'wt')
f.write('hello world\n')
f.close()
```

当高层的文件对象被关闭或者破坏的时候，底层的文件描述符也会被关闭。 如果这个并不是你想要的结果，你可以给 `open()` 函数传递一个可选的 `colsefd=False` 。比如：

```
# Create a file object, but don't close underlying fd when done
f = open(fd, 'wt', closefd=False)
...
```

讨论¶

在Unix系统中，这种包装文件描述符的技术可以很方便的将一个类文件接口作用于一个以不同方式打开的I/O通道上，如管道、套接字等。举例来讲，下面是一个操作管道的例子：

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_client(client_sock, addr):
    print('Got connection from', addr)

    # Make text-mode file wrappers for socket reading/writing
    client_in = open(client_sock.fileno(), 'rt', encoding='latin-1',
                closefd=False)

    client_out = open(client_sock.fileno(), 'wt', encoding='latin-1',
                closefd=False)

    # Echo lines back to the client using file I/O
    for line in client_in:
        client_out.write(line)
        client_out.flush()

    client_sock.close()

def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(1)
    while True:
        client, addr = sock.accept()
        echo_client(client, addr)
```

需要重点强调的一点是，上面的例子仅仅是为了演示内置的 `open()` 函数的一个特性，并且也只适用于基于Unix的系统。 如果你想将一个类文件接口作用在一个套接字并希望你的代码可以跨平台，请使用套接字对象的 `makefile()` 方

法。 但是如果不考虑可移植性的话，那上面的解决方案会比使用 `makefile()` 性能更好一点。

你也可以使用这种技术来构造一个别名，允许以不同于第一次打开文件的方式使用它。 例如，下面演示如何创建一个文件对象，它允许你输出二进制数据到标准输出(通常以文本模式打开)：

```
import sys
# Create a binary-mode file for stdout
bstdout = open(sys.stdout.fileno(), 'wb', closefd=False)
bstdout.write(b'Hello World\n')
bstdout.flush()
```

尽管可以将一个已存在的文件描述符包装成一个正常的文件对象， 但是要注意的是并不是所有的文件模式都被支持，并且某些类型的文件描述符可能会有副作用 (特别是涉及到错误处理、文件结尾条件等等的时候)。 在不同的操作系统上这种行为也是不一样，特别的，上面的例子都不能在非Unix系统上运行。 我说了这么多，意思就是让你充分测试自己的实现代码，确保它能按照期望工作。

5.19 创建临时文件和文件夹¶

问题¶

你需要在程序执行时创建一个临时文件或目录，并希望使用完之后可以自动销毁掉。

解决方案¶

tempfile 模块中有很多的函数可以完成这任务。 为了创建一个匿名的临时文件，可以使用 tempfile.TemporaryFile :

```
from tempfile import TemporaryFile

with TemporaryFile('w+t') as f:
    # Read/write to the file
    f.write('Hello World\n')
    f.write('Testing\n')

    # Seek back to beginning and read the data
    f.seek(0)
    data = f.read()

# Temporary file is destroyed
```

或者，如果你喜欢，你还可以像这样使用临时文件：

```
f = TemporaryFile('w+t')
# Use the temporary file
...
f.close()
# File is destroyed
```

TemporaryFile() 的第一个参数是文件模式，通常来讲文本模式使用 w+t ，二进制模式使用 w+b 。 这个模式同时支持读和写操作，在这里是很有用的，因为当你关闭文件去改变模式的时候，文件实际上已经不存在了。 TemporaryFile() 另外还支持跟内置的 open() 函数一样的参数。比如：

```
with TemporaryFile('w+t', encoding='utf-8', errors='ignore') as f:
    ...
```

在大多数Unix系统上，通过 TemporaryFile() 创建的文件都是匿名的，甚至连目录都没有。 如果你想打破这个限制，可以使用 NamedTemporaryFile() 来代替。比如：

```
from tempfile import NamedTemporaryFile

with NamedTemporaryFile('w+t') as f:
    print('filename is:', f.name)
    ...

# File automatically destroyed
```

这里，被打开文件的 f.name 属性包含了该临时文件的文件名。 当你需要将文件名传递给其他代码来打开这个文件的时候，这个就很有用了。 和 TemporaryFile() 一样，结果文件关闭时会被自动删除掉。 如果你不想这么做，可以传递一个关键字参数 delete=False 即可。比如：

```
with NamedTemporaryFile('w+t', delete=False) as f:
    print('filename is:', f.name)
    ...
```

为了创建一个临时目录，可以使用 tempfile.TemporaryDirectory() 。比如：

```
from tempfile import TemporaryDirectory

with TemporaryDirectory() as dirname:
    print('dirname is:', dirname)
```

```
    # Use the directory
    ...
# Directory and all contents destroyed
```

讨论¶

TemporaryFile()、NamedTemporaryFile() 和 TemporaryDirectory() 函数 应该是处理临时文件目录的最简单的方式了，因为它们会自动处理所有的创建和清理步骤。 在一个更低的级别，你可以使用 mkstemp() 和 mkdtemp() 来创建临时文件和目录。比如：

```
>>> import tempfile
>>> tempfile.mkstemp()
(3, '/var/folders/7W/7WZl5sfZEF0pljrEB1UMWE+++TI/-Tmp-/tmp7fefhv')
>>> tempfile.mkdtemp()
'/var/folders/7W/7WZl5sfZEF0pljrEB1UMWE+++TI/-Tmp-/tmp5wvcv6'
>>>
```

但是，这些函数并不会做进一步的管理了。 例如，函数 mkstemp() 仅仅就返回一个原始的OS文件描述符，你需要自己将它转换为一个真正的文件对象。 同样你还需要自己清理这些文件。

通常来讲，临时文件在系统默认的位置被创建，比如 /var/tmp 或类似的地方。 为了获取真实的位置，可以使用 tempfile.gettempdir() 函数。比如：

```
>>> tempfile.gettempdir()
'/var/folders/7W/7WZl5sfZEF0pljrEB1UMWE+++TI/-Tmp-'
>>>
```

所有和临时文件相关的函数都允许你通过使用关键字参数 prefix、suffix 和 dir 来自定义目录以及命名规则。比如：

```
>>> f = NamedTemporaryFile(prefix='mytemp', suffix='.txt', dir='/tmp')
>>> f.name
'/tmp/mytemp8ee899.txt'
>>>
```

最后还有一点，尽可能以最安全的方式使用 tempfile 模块来创建临时文件。 包括仅给当前用户授权访问以及在文件创建过程中采取措施避免竞态条件。 要注意的是不同的平台可能会不一样。因此你最好阅读 官方文档 来了解更多的细节。

5.2 打印输出至文件中¶

问题¶

你想将 `print()` 函数的输出重定向到一个文件中去。

解决方案¶

在 `print()` 函数中指定 `file` 关键字参数，像下面这样：

```
with open('d:/work/test.txt', 'wt') as f:
    print('Hello World!', file=f)
```

讨论¶

关于输出重定向到文件中就这些了。但是有一点要注意的就是文件必须是以文本模式打开。 如果文件是二进制模式的话，打印就会出错。

5.20 与串行端口的数据通信¶

问题¶

你想通过串行端口读写数据，典型场景就是和一些硬件设备打交道(比如一个机器人或传感器)。

解决方案¶

尽管你可以通过使用Python内置的I/O模块来完成这个任务，但对于串行通信最好的选择是使用 pySerial包 。 这个包的使用非常简单，先安装pySerial，使用类似下面这样的代码就能很容易的打开一个串行端口：

```
import serial
ser = serial.Serial('/dev/tty.usbmodem641', # Device name varies
                    baudrate=9600,
                    bytesize=8,
                    parity='N',
                    stopbits=1)
```

设备名对于不同的设备和操作系统是不一样的。 比如，在Windows系统上，你可以使用0, 1等表示的一个设备来打开通信端口”COM0”和”COM1”。 一旦端口打开，那就可以使用 `read()`， `readline()` 和 `write()` 函数读写数据了。例如：

```
ser.write(b'G1 X50 Y50\r\n')
resp = ser.readline()
```

大多数情况下，简单的串口通信从此变得十分简单。

讨论¶

尽管表面上看起来很简单，其实串口通信有时候也是挺麻烦的。 推荐你使用第三方包如 `pySerial` 的一个原因是它提供了对高级特性的支持 (比如超时，控制流，缓冲区刷新，握手协议等等)。举个例子，如果你想启用 `RTS-CTS` 握手协议， 你只需要给 `Serial()` 传递一个 `rtscts=True` 的参数即可。 其官方文档非常完善，因此我在这里极力推荐这个包。

时刻记住所有涉及到串口的I/O都是二进制模式的。因此，确保你的代码使用的是字节而不是文本 (或有时候执行文本的编码/解码操作)。 另外当你需要创建二进制编码的指令或数据包的时候， `struct` 模块也是非常有用的。

5.21 序列化Python对象¶

问题¶

你需要将一个Python对象序列化为一个字节流，以便将它保存到一个文件、存储到数据库或者通过网络传输它。

解决方案¶

对于序列化最普遍的做法就是使用 `pickle` 模块。为了将一个对象保存到一个文件中，可以这样做：

```
import pickle

data = ... # Some Python object
f = open('somefile', 'wb')
pickle.dump(data, f)
```

为了将一个对象转储为一个字符串，可以使用 `pickle.dumps()`：

```
s = pickle.dumps(data)
```

为了从字节流中恢复一个对象，使用 `pickle.load()` 或 `pickle.loads()` 函数。比如：

```
# Restore from a file
f = open('somefile', 'rb')
data = pickle.load(f)

# Restore from a string
data = pickle.loads(s)
```

讨论¶

对于大多数应用程序来讲，`dump()` 和 `load()` 函数的使用就是你有效使用 `pickle` 模块所需的全部了。 它可适用于绝大部分Python数据类型和用户自定义类的对象实例。 如果你碰到某个库可以让你在数据库中保存/恢复Python对象或者是通过网络传输对象的话， 那么很有可能这个库的底层就使用了 `pickle` 模块。

`pickle` 是一种Python特有的自描述的数据编码。 通过自描述，被序列化后的数据包含每个对象开始和结束以及它的类型信息。 因此，你无需担心对象记录的定义，它总是能工作。 举个例子，如果要处理多个对象，你可以这样做：

```
>>> import pickle
>>> f = open('somedata', 'wb')
>>> pickle.dump([1, 2, 3, 4], f)
>>> pickle.dump('hello', f)
>>> pickle.dump({'Apple', 'Pear', 'Banana'}, f)
>>> f.close()
>>> f = open('somedata', 'rb')
>>> pickle.load(f)
[1, 2, 3, 4]
>>> pickle.load(f)
'hello'
>>> pickle.load(f)
{'Apple', 'Pear', 'Banana'}
>>>
```

你还能序列化函数，类，还有接口，但是结果数据仅仅将它们的名称编码成对应的代码对象。例如：

```
>>> import math
>>> import pickle.
>>> pickle.dumps(math.cos)
b'\x80\x03cmath\ncos\nq\x00.'
>>>
```

当数据反序列化回来的时候，会先假定所有的源数据时可用的。 模块、类和函数会自动按需导入进来。对于Python数据被不同机器上的解析器所共享的应用程序而言， 数据的保存可能会有问题，因为所有的机器都必须访问同一个源代

码。

注

千万不要对不信任的数据使用pickle.load()。
pickle在加载时有一个副作用就是它会自动加载相应模块并构造实例对象。
但是某个坏人如果知道pickle的工作原理，
他就可以创建一个恶意的数据导致Python执行随意指定的系统命令。
因此，一定要保证pickle只在相互之间可以认证对方的解析器的内部使用。

有些类型的对象是不能被序列化的。这些通常是那些依赖外部系统状态的对象， 比如打开的文件，网络连接，线程，进程，栈帧等等。 用户自定义类可以通过提供 `__getstate__()` 和 `__setstate__()` 方法来绕过这些限制。 如果定义了这两个方法，`pickle.dump()` 就会调用 `__getstate__()` 获取序列化的对象。 类似的， `__setstate__()` 在反序列化时被调用。为了演示这个工作原理， 下面是一个在内部定义了一个线程但仍然可以序列化和反序列化的类：

```
# countdown.py
import time
import threading

class Countdown:
    def __init__(self, n):
        self.n = n
        self.thr = threading.Thread(target=self.run)
        self.thr.daemon = True
        self.thr.start()

    def run(self):
        while self.n > 0:
            print('T-minus', self.n)
            self.n -= 1
            time.sleep(5)

    def __getstate__(self):
        return self.n

    def __setstate__(self, n):
        self.__init__(n)
```

试着运行下面的序列化试验代码：

```
>>> import countdown
>>> c = countdown.Countdown(30)
>>> T-minus 30
T-minus 29
T-minus 28
...

>>> # After a few moments
>>> f = open('cstate.p', 'wb')
>>> import pickle
>>> pickle.dump(c, f)
>>> f.close()
```

然后退出Python解析器并重启后再试验下：

```
>>> f = open('cstate.p', 'rb')
>>> pickle.load(f)
countdown.Countdown object at 0x10069e2d0>
T-minus 19
T-minus 18
...
```

你可以看到线程又奇迹般的重生了，从你第一次序列化它的地方又恢复过来。

`pickle` 对于大型的数据结构比如使用 `array` 或 `numpy` 模块创建的二进制数组效率并不是一个高效的编码方式。 如果你需要移动大量的数组数据，你最好是先在一个文件中将其保存为数组数据块或使用更高级的标准编码方式如HDF5 (需要第三方库的支持)。

由于 `pickle` 是Python特有的并且附着在源码上，所有如果需要长期存储数据的时候不应该选用它。 例如，如果源码变动了，你所有的存储数据可能会被破坏并且变得不可读取。 坦白来讲，对于在数据库和存档文件中存储数据时，你最好使用更加标准的数据编码格式如XML，CSV或JSON。 这些编码格式更标准，可以被不同的语言支持，并且也能很好的适应源码变更。

最后一点要注意的是 `pickle` 有大量的配置选项和一些棘手的问题。 对于最常见的使用场景，你不需要去担心这个，但是如果你要在一个重要的程序中使用pickle去做序列化的话， 最好去查阅一下 官方文档 。

5.3 使用其他分隔符或行终止符打印¶

问题¶

你想使用 `print()` 函数输出数据，但是想改变默认的分隔符或者行尾符。

解决方案¶

可以使用在 `print()` 函数中使用 `sep` 和 `end` 关键字参数，以你想要的方式输出。比如：

```
>>> print('ACME', 50, 91.5)
ACME 50 91.5
>>> print('ACME', 50, 91.5, sep=',')
ACME,50,91.5
>>> print('ACME', 50, 91.5, sep=',', end='!!\n')
ACME,50,91.5!!
>>>
```

使用 `end` 参数也可以在输出中禁止换行。比如：

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>> for i in range(5):
...     print(i, end=' ')
...
0 1 2 3 4 >>>
```

讨论¶

当你想使用非空格分隔符来输出数据的时候，给 `print()` 函数传递一个 `sep` 参数是最简单的方案。 有时候你会看到一些程序员会使用 `str.join()` 来完成同样的事情。比如：

```
>>> print(','.join(('ACME','50','91.5')))
ACME,50,91.5
>>>
```

`str.join()` 的问题在于它仅仅适用于字符串。这意味着你通常需要执行另外一些转换才能让它正常工作。比如：

```
>>> row = ('ACME', 50, 91.5)
>>> print(','.join(row))
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: sequence item 1: expected str instance, int found
>>> print(','.join(str(x) for x in row))
ACME,50,91.5
>>>
```

你当然可以不用那么麻烦，只需要像下面这样写：

```
>>> print(*row, sep=',')
ACME,50,91.5
>>>
```

5.4 读写字节数据¶

问题¶

你想读写二进制文件，比如图片，声音文件等等。

解决方案¶

使用模式为 `rb` 或 `wb` 的 `open()` 函数来读取或写入二进制数据。比如：

```
# Read the entire file as a single byte string
with open('somefile.bin', 'rb') as f:
    data = f.read()

# Write binary data to a file
with open('somefile.bin', 'wb') as f:
    f.write(b'Hello World')
```

在读取二进制数据时，需要指明的是所有返回的数据都是字节字符串格式的，而不是文本字符串。 类似的，在写入的时候，必须保证参数是以字节形式对外暴露数据的对象(比如字节字符串，字节数组对象等)。

讨论¶

在读取二进制数据的时候，字节字符串和文本字符串的语义差异可能会导致一个潜在的陷阱。 特别需要注意的是，索引和迭代动作返回的是字节的值而不是字节字符串。比如：

```
>>> # Text string
>>> t = 'Hello World'
>>> t[0]
'H'
>>> for c in t:
...     print(c)
...
H
e
l
l
o
...
>>> # Byte string
>>> b = b'Hello World'
>>> b[0]
72
>>> for c in b:
...     print(c)
...
72
101
108
108
111
...
>>>
```

如果你想从二进制模式的文件中读取或写入文本数据，必须确保要进行解码和编码操作。比如：

```
with open('somefile.bin', 'rb') as f:
    data = f.read(16)
    text = data.decode('utf-8')

with open('somefile.bin', 'wb') as f:
    text = 'Hello World'
    f.write(text.encode('utf-8'))
```

二进制I/O还有一个鲜为人知的特性就是数组和C结构体类型能直接被写入，而不需要中间转换为自己对象。比如：

```
import array
nums = array.array('i', [1, 2, 3, 4])
with open('data.bin','wb') as f:
    f.write(nums)
```

这个适用于任何实现了被称之为"缓冲接口"的对象，这种对象会直接暴露其底层的内存缓冲区给能处理它的操作。 二进制数据的写入就是这类操作之一。

很多对象还允许通过使用文件对象的 `readinto()` 方法直接读取二进制数据到其底层的内存中去。比如：

```
>>> import array
>>> a = array.array('i', [0, 0, 0, 0, 0, 0, 0, 0])
>>> with open('data.bin', 'rb') as f:
...     f.readinto(a)
...
16
>>> a
array('i', [1, 2, 3, 4, 0, 0, 0, 0])
>>>
```

但是使用这种技术的时候需要格外小心，因为它通常具有平台相关性，并且可能会依赖字长和字节顺序(高位优先和低位优先)。 可以查看5.9小节中另外一个读取二进制数据到可修改缓冲区的例子。

5.5 文件不存在才能写入¶

问题¶

你想像一个文件中写入数据，但是前提必须是这个文件在文件系统上不存在。 也就是不允许覆盖已存在的文件内容。

解决方案¶

可以在 `open()` 函数中使用 `x` 模式来代替 `w` 模式的方法来解决这个问题。比如：

```
>>> with open('somefile', 'wt') as f:
...     f.write('Hello\n')
...
>>> with open('somefile', 'xt') as f:
...     f.write('Hello\n')
...
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'somefile'
>>>
```

如果文件是二进制的，使用 `xb` 来代替 `xt`

讨论¶

这一小节演示了在写文件时通常会遇到的一个问题的完美解决方案(不小心覆盖一个已存在的文件)。 一个替代方案是先测试这个文件是否存在，像下面这样：

```
>>> import os
>>> if not os.path.exists('somefile'):
...     with open('somefile', 'wt') as f:
...         f.write('Hello\n')
... else:
...     print('File already exists!')
...
File already exists!
>>>
```

显而易见，使用x文件模式更加简单。要注意的是x模式是一个Python3对 `open()` 函数特有的扩展。 在Python的旧版本或者是Python实现的底层C函数库中都是没有这个模式的。

5.6 字符串的I/O操作¶

问题¶

你想使用操作类文件对象的程序来操作文本或二进制字符串。

解决方案¶

使用 `io.StringIO()` 和 `io.BytesIO()` 类来创建类文件对象操作字符串数据。比如：

```
>>> s = io.StringIO()
>>> s.write('Hello World\n')
12
>>> print('This is a test', file=s)
15
>>> # Get all of the data written so far
>>> s.getvalue()
'Hello World\nThis is a test\n'
>>>

>>> # Wrap a file interface around an existing string
>>> s = io.StringIO('Hello\nWorld\n')
>>> s.read(4)
'Hell'
>>> s.read()
'o\nWorld\n'
>>>
```

`io.StringIO` 只能用于文本。如果你要操作二进制数据，要使用 `io.BytesIO` 类来代替。比如：

```
>>> s = io.BytesIO()
>>> s.write(b'binary data')
>>> s.getvalue()
b'binary data'
>>>
```

讨论¶

当你想模拟一个普通的文件的时候 `StringIO` 和 `BytesIO` 类是很有用的。 比如，在单元测试中，你可以使用 `StringIO` 来创建一个包含测试数据的类文件对象， 这个对象可以被传给某个参数为普通文件对象的函数。

需要注意的是， `StringIO` 和 `BytesIO` 实例并没有正确的整数类型的文件描述符。 因此，它们不能在那些需要使用真实的系统级文件如文件，管道或者是套接字的程序中使用。

5.7 读写压缩文件¶

问题¶

你想读写一个gzip或bz2格式的压缩文件。

解决方案¶

gzip 和 bz2 模块可以很容易的处理这些文件。 两个模块都为 open() 函数提供了另外的实现来解决这个问题。 比如，为了以文本形式读取压缩文件，可以这样做：

```
# gzip compression
import gzip
with gzip.open('somefile.gz', 'rt') as f:
    text = f.read()

# bz2 compression
import bz2
with bz2.open('somefile.bz2', 'rt') as f:
    text = f.read()
```

类似的，为了写入压缩数据，可以这样做：

```
# gzip compression
import gzip
with gzip.open('somefile.gz', 'wt') as f:
    f.write(text)

# bz2 compression
import bz2
with bz2.open('somefile.bz2', 'wt') as f:
    f.write(text)
```

如上，所有的I/O操作都使用文本模式并执行Unicode的编码/解码。 类似的，如果你想操作二进制数据，使用 rb 或者 wb 文件模式即可。

讨论¶

大部分情况下读写压缩数据都是很简单的。但是要注意的是选择一个正确的文件模式是非常重要的。 如果你不指定模式，那么默认的就是二进制模式，如果这时候程序想要接受的是文本数据，那么就会出错。 gzip.open() 和 bz2.open() 接受跟内置的 open() 函数一样的参数， 包括 encoding，errors，newline 等等。

当写入压缩数据时，可以使用 compresslevel 这个可选的关键字参数来指定一个压缩级别。比如：

```
with gzip.open('somefile.gz', 'wt', compresslevel=5) as f:
    f.write(text)
```

默认的等级是9，也是最高的压缩等级。等级越低性能越好，但是数据压缩程度也越低。

最后一点， gzip.open() 和 bz2.open() 还有一个很少被知道的特性， 它们可以作用在一个已存在并以二进制模式打开的文件上。比如，下面代码是可行的：

```
import gzip
f = open('somefile.gz', 'rb')
with gzip.open(f, 'rt') as g:
    text = g.read()
```

这样就允许 gzip 和 bz2 模块可以工作在许多类文件对象上，比如套接字，管道和内存中文件等。

5.8 固定大小记录的文件迭代¶

问题¶

你想在一个固定长度记录或者数据块的集合上迭代，而不是在一个文件中一行一行的迭代。

解决方案¶

通过下面这个小技巧使用 `iter` 和 `functools.partial()` 函数：

```
from functools import partial

RECORD_SIZE = 32

with open('somefile.data', 'rb') as f:
    records = iter(partial(f.read, RECORD_SIZE), b'')
    for r in records:
        ...
```

这个例子中的 `records` 对象是一个可迭代对象，它会不断的产生固定大小的数据块，直到文件末尾。 要注意的是如果总记录大小不是块大小的整数倍的话，最后一个返回元素的字节数会比期望值少。

讨论¶

`iter()` 函数有一个鲜为人知的特性就是，如果你给它传递一个可调用对象和一个标记值，它会创建一个迭代器。 这个迭代器会一直调用传入的可调用对象直到它返回标记值为止，这时候迭代终止。

在例子中， `functools.partial` 用来创建一个每次被调用时从文件中读取固定数目字节的可调用对象。 标记值 `b''` 就是当到达文件结尾时的返回值。

最后再提一点，上面的例子中的文件时以二进制模式打开的。 如果是读取固定大小的记录，这通常是最普遍的情况。而对于文本文件，一行一行的读取(默认的迭代行为)更普遍点。

5.9 读取二进制数据到可变缓冲区中¶

问题¶

你想直接读取二进制数据到一个可变缓冲区中，而不需要做任何的中间复制操作。 或者你想原地修改数据并将它写回到一个文件中去。

解决方案¶

为了读取数据到一个可变数组中，使用文件对象的 `readinto()` 方法。比如：

```
import os.path

def read_into_buffer(filename):
    buf = bytearray(os.path.getsize(filename))
    with open(filename, 'rb') as f:
        f.readinto(buf)
    return buf
```

下面是一个演示这个函数使用方法的例子：

```
>>> # Write a sample file
>>> with open('sample.bin', 'wb') as f:
...     f.write(b'Hello World')
...
>>> buf = read_into_buffer('sample.bin')
>>> buf
bytearray(b'Hello World')
>>> buf[0:5] = b'Hello'
>>> buf
bytearray(b'Hello World')
>>> with open('newsample.bin', 'wb') as f:
...     f.write(buf)
...
11
>>>
```

讨论¶

文件对象的 `readinto()` 方法能被用来为预先分配内存的数组填充数据，甚至包括由 `array` 模块或 `numpy` 库创建的数组。 和普通 `read()` 方法不同的是， `readinto()` 填充已存在的缓冲区而不是为新对象重新分配内存再返回它们。 因此，你可以使用它来避免大量的内存分配操作。 比如，如果你读取一个由相同大小的记录组成的二进制文件时，你可以像下面这样写：

```
record_size = 32 # Size of each record (adjust value)

buf = bytearray(record_size)
with open('somefile', 'rb') as f:
    while True:
        n = f.readinto(buf)
        if n < record_size:
            break
        # Use the contents of buf
        ...
```

另外有一个有趣特性就是 `memoryview` ， 它可以通过零复制的方式对已存在的缓冲区执行切片操作，甚至还能修改它的内容。比如：

```
>>> buf
bytearray(b'Hello World')
>>> m1 = memoryview(buf)
>>> m2 = m1[-5:]
>>> m2
<memory at 0x100681390>
```

```
>>> m2[:] = b'WORLD'
>>> buf
bytearray(b'Hello WORLD')
>>>
```

使用 `f.readinto()` 时需要注意的是，你必须检查它的返回值，也就是实际读取的字节数。

如果字节数小于缓冲区大小，表明数据被截断或者被破坏了(比如你期望每次读取指定数量的字节)。

最后，留心观察其他函数库和模块中和 `into` 相关的函数(比如 `recv_into()` ， `pack_into()` 等)。 Python的很多其他部分已经能支持直接的I/O或数据访问操作，这些操作可被用来填充或修改数组和缓冲区内容。

关于解析二进制结构和 `memoryviews` 使用方法的更高级例子，请参考6.12小节。

第五章：文件与IO¶

所有程序都要处理输入和输出。 这一章将涵盖处理不同类型的文件，包括文本和二进制文件，文件编码和其他相关的内容。 对文件名和目录的操作也会涉及到。

Contents: