Python 文件输入/输出

文件不单单指磁盘上的普通文件，也指代任何抽象层面上的文件。例如：通过URL打开一个Web页面“文件”，Unix系统下进程间通讯也是通过抽象的进程“文件”进行的。由于使用了统一的接口，从而统一了各种抽象类型及非抽象类型文件的操作方式。

所有程序都要处理输入和输出，需要在非易失性介质上做持久化存储和检索数据，所以文件操作的重要性无需多言。

读写文本数据

Python模仿Unix系统来进行文件操作，读写一个文件之前需要打开它：

```
file_obj = open(file_name, mode, buffering=-1, encoding=None, errors=None, ...)
```

- `file_obj`是`open()`返回的可迭代的**文件对象**，可以用`for`循环遍历；
- `file_name`是要打开文件的文件名，非当前目录下的文件要指明路径；
- `mode`指明文件类型和操作的字符串（文件读取模式）；
- `buffering`用来设定缓冲模式，默认值为`-1`，即使用系统默认缓冲模式；
- `encoding`用来指定文件编码、解码的编码类型，只针对文本文件有效，默认选择所在系统使用的编码方式；
- `errors`用来指定编码错误的处理方式。

`mode`第一个字母表明对文件的操作：

- `r`表示读模式。
- `w`表示写模式。如果文件不存在则新建文件，如果文件存在则重写新内容。
- `x`表示在文件不存在是新建并写入文件。
- `a`表示如果文件存在，在文件末尾追加写入内容。

`mode`第二个字母是文件类型：

- `t`（可省略）代表文本类型；
- `b`表示二进制文件。

一些`mode`参数的详细说明：

```
# 文件对象访问模式
r    只读方式打开              rU    读方式打开，提供通用换行符支持
w    以写方式打开              a     以追加模式打开
r+   以读写模式打开            w+    以读写模式打开
rb   以二进制读模式打开        wb    以二进制写模式打开
ab   以二进制追加模式打开      rb+   以二进制读写模式打开
wb+  以二进制读写模式打开      ab+   以二进制读写模式打开

# 以r模式打开时，文件必须存在，否则引发错误
# 以w模式打开文件时，如果文件存在则清空，不存在则创建
# 以a模式打开时，如果文件存在，则从EOF位置追加，否则创建新文件
```

`buffering`参数模式：

```
0         不缓冲
1         只缓冲一行
value>1   缓冲区大小为value
value<0   使用系统默认缓冲机制
```

使用带有`rt`模式的`open()`函数读取文本文件。

```
# 将整个文件当作一个字符串读取
with open('somefile.txt', 'rt') as f:
    data = f.read()


# 迭代读取文件的每一行
with open('somefile.txt', 'rt') as f:
    for line in f:
        # process line
        pass
```

这里的`with`语句是Python中的上下文管理器（context manager）类型，形式为`with expression as variable`，其主要作用包括：保存、重置各种全局状态，锁住或解锁资源，关闭打开的文件等。

你也可以不使用`with`语句，但这时候你必须记得手动关闭文件：

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

上面的例子也可以用`try...finally...`实现，它们的效果是相同的（上下文管理器封装、简化了错误捕捉的过程）。

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

除了文件对象之外，可以自己创建上下文管理器，只要定义了`__enter__()`和`__exit__()`方法就成为了上下文管理器类型。`with`语句的执行过程如下：

1. 执行`with`后的语句获取上下文管理器，例如`open('somefile.txt', 'rt')`就是返回一个文件对象；
2. 加载`__exit__()`方法备用；
3. 执行`__enter__()`，该方法的返回值将传递给`as`后的变量（如果有的话）；
4. 执行`with`语法块的子句；
5. 执行`__exit__()`方法，如果`with`语法块子句中出现异常，将会传递`type, value, traceback`给`__exit__()`，否则将默认为`None`；如果`__exit__()`方法返回`False`，将会抛出异常给外层处理；如果返回`True`，则忽略异常。

关于上文管理器的更多内容可参考Python官方文档。

为了写入一个文本文件，使用带有`wt`模式的`open()`函数，如果之前文件有内容则清除并覆盖掉。

```
# 写入文本数据块
with open('somefile.txt', 'wt') as f:
    f.write(text1)
    f.write(text2)
    pass

# 重定向打印语句
with open('somefile.txt', 'wt') as f:
    print(line1, file=f)
    print(line2, file=f)
    pass
```

如果是在已存在文件中添加内容，使用模式`at`。

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。在大多数机器上面默认都是utf-8编码。如果你已经知道要读写的文本是其他编码方式，那么可通过传递一个可选的`encoding`参数给`open()`函数。

```
with open('somefile.txt', 'rt', encoding='gb18030') as f:
    pass
```

另外一个需要注意的问题是关于换行符的识别问题。在不同的系统平台上，换行符是不同的，例如在Unix下是`\n`，而Windows中是`\r\n`。

Python为了解决这个问题，使用了通用换行符支持（Universal NEWLINE Support）。默认情况下，Python会以统一模式处理换行符。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。通过这种方式，Python屏蔽了不同平台下的换行符差异。

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=''`：

```
# 读取时停用换行符转换
with open('somefile.txt', 'rt', newline='') as f:
    pass
```

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'：

```
# 开启换行符转换（默认）
f = open('hello.txt', 'rt')
f.read()
```

```
# 停用换行符转换
g = open('hello.txt', 'rt', newline='')
g.read()
```

这里需要提醒大家，跨平台开发会遇到一些不可避免的问题，不同平台下的**换行符差异**及**路径分隔符差异**等等就是一个特例。Python的`os`模块提供了一些属性值以便于跨平台应用的开发：

- `os.linesep` （当前系统下，下同）用于在文件中分隔行的字符串
- `os.sep` 用于分隔文件路径名的字符串
- `os.pathsep` 用于分隔文件路径的字符串
- `os.curdir` 当前工作目录的字符串名称

- `os.pardir` 当前工作目录父目录的字符串名称

最后一个问题就是文本文件中可能出现的编码错误。

```
f = open('rain.txt', 'rt', encoding='ascii')
f.read()
```

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，你需要确认和指定正确的文件编码。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。

```
# 使用Unicode U+fffd 替换错误字符
f = open('rain.txt', 'rt', encoding='ascii', errors='replace')
f.read()
```

```
# 忽略错误字符
g = open('rain.txt', 'rt', encoding='ascii', errors='ignore')
g.read()
```

对于文本处理的首要原则是确保你的总是使用正确的文件编码。当模棱两可时，就使用默认的设置（通常都是UTF-8）。

打印输出至文件中

一般我们使用文件对象的`write()`方法写入文本文件。它没有增加空格或者换行符。

```
f = open('allo.txt', 'wt')
print(f.write('Hello, world!'))  # 返回写入文件的字节数
f.close()
```

另外，我们可以指定`print()`函数的`file`关键字参数，将其输出重定向到一个文件中。

```
with open('allo.txt', 'wt') as f:
    print('Hello, world!', file=f)
```

这里需要注意的是文件必须以文本模式打开。`print()`默认会在每个参数后面添加空格，在每行结束处添加换行。

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

```
print(2017, 4, 11)
print(2017, 4, 11, sep='/')
print(2017, 4, 11, sep='/', end=' <--\n')
```

读写二进制数据

使用模式为`rb`或`wb`的`open()`函数来读取或写入二进制数据。

```
# 将整个文件当作一个字节字符串读取
with open('somefile.bin', 'rb') as f:
    data = f.read()

# 将二进制数据写入文件
with open('somefile.bin', 'wb') as f:
    f.write(b'Hello World')
```

在读取二进制数据时，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。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）。

另外，在读取二进制数据的时候，索引和迭代动作返回的是字节的值而不是字节字符串。

```
# 文本字符串
t = 'Hello World'
print(t[0])

for c in t:
    print(c)


# 字节字符串
b = b'Hello World'
print(b[0])
for c in b:
    print(c)
```

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。

```
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'))
```

文件不存在才能写入

当文件不存在时才能写入，不允许覆盖已存在的文件内容。

在`open()`函数中使用`x`模式来代替`w`模式来进行处理：

```
with open('somefile', 'wt') as f:
    f.write('Hello\n')

with open('somefile', 'xt') as f:
    f.write('Hello\n')
```

如果文件是二进制的，使用`xb`来代替`xt`。

一个替代方案是先测试这个文件是否存在：

```
import os

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

使用`read()`、`readline()`和`readlines()`读文本文件

使用不带参数的`read()`函数一次读入文件的所有内容。可以设置最大的读入字符数限制`read()`函数一次返回的大小，如果没有给定size或者size为负数，文件将被读取直至末尾：

```
chunk = 5
content = ''
fobj = open('allo.txt', 'rt')
while 1:
    fragment = fobj.read(chunk)
    if not fragment:
        break
    content += fragment

fobj.close()
len(content)
```

读到文件结尾之后，再次调用read()会返回空字符串（''）。这个函数不推荐使用。

使用`readline()`每次读入文件的一行，如果指定了`size`，每次读取文件中`size`个字节，返回一个字符串。如果没有给定`size`或者`size`为负数则返回一行（包括行结束符）。

```
content = ''
fobj = open('argv.py', 'rt')
while 1:
    line = fobj.readline()
    if not line:
        break
    content += line

fobj.close()
len(content)
```

对于文本文件，空行也有1字符长度（换行符`\n`），当文件读取结束后，`readline()`会返回空字符串。

使用迭代器读取文件，每次返回一行。

```
content = ''
fobj = open('argv.py', 'rt')
for line in fobj:
    content += line

fobj.close()
len(content)
```

采用迭代方式读取超大型文件或者网络流文件的好处是显而易见的，避免了一次性将大型文件读入内存所带来的负担（有时候甚至是不可能的，例如网络流文件）。对于小型文件还是一次性读入好些，可以尽快释放文件资源。

函数`readlines()`读取剩余的所有的行（文件指针不一定在开始位置！）并将其以一个字符串列表形式返回。此方法一次性读取文件所有的内容至内存中，适用于小型文件。

```
fobj = open('argv.py', 'rt')
lines = fobj.readlines()
fobj.close()
print(len(lines))
for line in lines:
    print(line, end='')
```

与`readlines()`相反，可以使用`file_obj.writelines()`来写入文件，它接收一个字符串列表作为参数并将其写入到文件中，每个字符串的行结束符不会被自动写入，通常适用于每个字符串末尾都包含换行符的字符串列表。

跟踪文件读写位置（移动文件指针）

- `tell()`返回距离文件开始处的字节偏移量，类似于C语言中的`ftell`。
- `seek(offset[, whence])`跳转到文件其他字节偏移量的位置，同样返回当前的偏移量。其类似于C语言中的`fseek`，`offset`为偏移量，`whence`代表相对位置，是一个可选参数。
    - 如果`whence`等于0（默认为0），从开头偏移`offset`个字节；
    - 如果`whence`等于1，从当前位置处偏移`offset`个字节；
    - 如果`whence`等于2，距离最后结尾处偏移`offset`个字节。

```
bdata = bytes(range(0, 256))
fobj = open('bfile', 'wb')
fobj.write(bdata)
fobj.close

fobj = open('bfile', 'rb')
print(fobj.tell())
print(fobj.seek(255))
print(fobj.seek(-5, 1))  # # 从当前位置后退5个字节
bdata = fobj.read()
print(len(bdata))
print(bdata)
print(bdata[0])
fobj.close()
```

一些其它方法

- `file_obj.fileno()` 返回打开文件的文件描述符，这是一个整形，可以用于os模块的一些底层操作
- `file_obj.flush()` 把输出缓冲区内的数据立即写入文件，调用`close`时会自动调用这个方法。
- `file_obj.isatty()` 当文件是一个tty设备时，返回`True`。tty是字符型终端设备，例如老式的打印机以及操作系统中的终端（Terminal）程序。
- `file_obj.next()` 返回文件的下一行，类似于`readline`方法，没有其它行时引发`StopIteration`异常。

数据编码和处理

读写CSV数据

使用Python自带的`csv`库来读写CSV格式的文件。假设有一个保存股票市场数据的文件`stocks.csv`：

```
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
```

可以通过下面的代码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
        pass
```

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

这里可以使用命名元组避免引起混淆：

```
from collections import namedtuple

with open('stocks.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
        pass
```

现在，你可以使用列名如`row.Rymbol`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，你可能需要修改原始的列名(如将非标识符字符替换成下划线之类的)。例如，一个CSV格式文件有一个包含非法标识符的头行，类似下面这样：

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

可以使用正则表达式替换非法标识符：

```
import re

with open('stocks.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
        pass
```

另外，可以选择将数据读取到一个字典中去：

```
import csv

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

现在，你可以使用列名去访问每一行的数据。比如`row['Symbol']`和`row['Change']`。

为了写入CSV数据，需要创建一个`csv.writer`对象。

```
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)
```

如果是字典数据的话，可以这样：

```
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.writerow(rows)
```

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转换，你必须自己手动去实现。

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)
```

读写JSON数据

Python的`json`模块提供了非常简单的方式来编码和解码JSON(JavaScript Object Notation)数据。其中两个主要的函数是`json.dumps()`和`json.loads()`。

下面的代码将Python数据结构转换为JSON：

```
import json

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

json_str = json.dumps(data)
```

下面的代码将一个JSON编码的字符串转换回一个Python数据结构：

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

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

```
# 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)
```

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。

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`。

```
print(json.dumps(False))

d = {'a': True,
     'b': 'Hello',
     'c': None}

print(json.dumps(d))
```

解析简单的XML数据

可以使用`xml.etree.ElementTree`模块从简单的XML文档中提取数据。下面的例子展示了如何解析Planet Python上的RSS源。

```
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()
```

`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：

```
print(doc)

e = doc.find('channel/title')
print(e)

print(e.text)

print(e.get('some_attribute'))
```

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。

序列化数据

序列化（serializing）即存储数据结构到一个文件中。Python提供了`pickle`模块以特殊的二进制格式保存和恢复数据对象。

```
import pickle
import datetime

now1 = datetime.datetime.utcnow()
pickled = pickle.dumps(now1)
now2 = pickle.loads(pickled)
print(now1)
print(now2)
```

使用`pickle.dump()`序列化数据到文件，而函数`pickle.load()`用作反序列化。