**第五部分 I\O**

##### 文件

读写文件是最常见的IO操作。Python内置了读写文件的函数，用法和C是兼容的。

读写文件前，先必须了解一下，在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

###### 读文件

要以读文件的模式打开一个文件对象，使用Python内置的open()函数，传入文件名和标示符：

f = open('/Users/michael/test.txt', 'r')

标示符'r'表示读，这样，就成功地打开了一个文件。

如果文件不存在，open()函数就会抛出一个IOError的错误，并且给出错误码和详细的信息告诉文件不存在：

f=open('/Users/michael/notfound.txt', 'r')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

FileNotFoundError: [Errno 2] No such file or directory: '/Users/michael/notfound.txt'

如果文件打开成功，接下来，调用read()方法可以一次读取文件的全部内容，Python把内容读到内存，用一个str对象表示：

f.read()

最后一步是调用close()方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

f.close()

由于文件读写时都有可能产生IOError，一旦出错，后面的f.close()就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，可以使用try ... finally来实现：

try:

f = open('/path/to/file', 'r')

print(f.read())

finally:

if f:

f.close()

但是每次都这么写实在太繁琐，所以，Python引入了with语句来自动帮调用close()方法：

with open('/path/to/file', 'r') as f:

print(f.read())

这和前面的try ... finally是一样的，但是代码更佳简洁，并且不必调用f.close()方法。

调用read()会一次性读取文件的全部内容，如果文件有10G，内存就爆了，所以，要保险起见，可以反复调用read(size)方法，每次最多读取size个字节的内容。另外，调用readline()可以每次读取一行内容，调用readlines()一次读取所有内容并按行返回list。因此，要根据需要决定怎么调用。

如果文件很小，read()一次性读取最方便；如果不能确定文件大小，反复调用read(size)比较保险；如果是配置文件，调用readlines()最方便：

for line in f.readlines():

print(line.strip()) # 把末尾的'\n'删掉

###### 读取二进制文件

前面讲的默认都是读取文本文件，并且是UTF-8编码的文本文件。要读取二进制文件，比如图片、视频等等，用'rb'模式打开文件即可：

f = open('/Users/michael/test.jpg', 'rb')

f.read()

###### 写文件

写文件和读文件是一样的，唯一区别是调用open()函数时，传入标识符'w'或者'wb'表示写文本文件或写二进制文件：

f = open('/Users/michael/test.txt', 'w')

f.write('Hello, world!')

###### with

可以反复调用write()来写入文件，但是务必要调用f.close()来关闭文件。当写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用close()方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用close()的后果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，还是用with语句来得保险：

with open('/Users/michael/test.txt', 'w') as f:

f.write('Hello, world!')

以'w'模式写入文件时，如果文件已存在，会直接覆盖（相当于删掉后新写入一个文件）。如果希望追加到文件末尾怎么办？可以传入'a'以追加（append）模式写入。

##### 环境变量、目录

###### Sys

如果要操作文件、目录，可以在命令行下面输入操作系统提供的各种命令来完成。比如dir、cp等命令。

如果要在Python程序中执行这些目录和文件的操作怎么办？其实操作系统提供的命令只是简单地调用了操作系统提供的接口函数，Python内置的os模块也可以直接调用操作系统提供的接口函数。

打开Python交互式命令行，来看看如何使用os模块的基本功能：

import os

os.name # 操作系统类型

如果是posix，说明系统是Linux、Unix或Mac OS X，如果是nt，就是Windows系统。

要获取详细的系统信息，可以调用uname()函数：

os.uname()

注意uname()函数在Windows上不提供，也就是说，os模块的某些函数是跟操作系统相关的。

###### 环境变量

在操作系统中定义的环境变量，全部保存在os.environ这个变量中，可以直接查看：

os.environ

要获取某个环境变量的值，可以调用os.environ.get('key')：

os.environ.get('PATH')

###### 操作文件和目录

操作文件和目录的函数一部分放在os模块中，一部分放在os.path模块中，这一点要注意一下。查看、创建和删除目录可以这么调用：

# 查看当前目录的绝对路径:

os.path.abspath('.')

'/Users/michael'

# 在某个目录下创建一个新目录，首先把新目录的完整路径表示出来:

os.path.join('/Users/michael', 'testdir')

'/Users/michael/testdir'

# 然后创建一个目录:

os.mkdir('/Users/michael/testdir')

# 删掉一个目录:

os.rmdir('/Users/michael/testdir')

把两个路径合成一个时，不要直接拼字符串，而要通过os.path.join()函数，这样可以正确处理不同操作系统的路径分隔符。在windows下，os.path.join()返回这样的字符串：

同样的道理，要拆分路径时，也不要直接去拆字符串，而要通过os.path.split()函数，这样可以把一个路径拆分为两部分，后一部分总是最后级别的目录或文件名：

os.path.split('c:\Users\michael\testdir\file.txt')

('/Users/michael/testdir', 'file.txt')

os.path.splitext()可以直接让得到文件扩展名，很多时候非常方便：

os.path.splitext(' c:\Users\michael\testdir\file.txt ')

这些合并、拆分路径的函数并不要求目录和文件要真实存在，它们只对字符串进行操作。

文件操作使用下面的函数。假定当前目录下有一个test.txt文件：

# 对文件重命名:

os.rename('test.txt', 'test.py')

# 删掉文件:

os.remove('test.py')

但是复制文件的函数居然在os模块中不存在！原因是复制文件并非由操作系统提供的系统调用。理论上讲，通过上一节的读写文件可以完成文件复制，只不过要多写很多代码。

最后看看如何利用Python的特性来过滤文件。比如要列出当前目录下的所有目录，只需要一行代码：

[x for x in os.listdir('.') if os.path.isdir(x)]

要列出所有的.py文件，也只需一行代码：

[x for x in os.listdir('.') if os.path.isfile(x) and os.path.splitext(x)[1]=='.py']

##### 序列化

###### 序列化字典

在程序运行的过程中，所有的变量都是在内存中，比如，定义一个dict：

d = dict(name='Bob', age=20, score=88)

可以随时修改变量，比如把name改成'Bill'，但是一旦程序结束，变量所占用的内存就被操作系统全部回收。如果没有把修改后的'Bill'存储到磁盘上，下次重新运行程序，变量又被初始化为'Bob'。

把变量从内存中变成可存储或传输的过程称之为序列化。

JSON是轻量级的，而且可以直接在Web页面中读取，非常方便。

JSON表示的对象就是标准的JavaScript语言的对象，JSON和Python内置的数据类型对应如下：

|  |  |
| --- | --- |
| JSON类型 | Python类型 |
| {} | dict |
| [] | list |
| "string" | str |
| 1234.56 | int或float |
| true/false | True/False |
| null | None |

Python内置的json模块提供了非常完善的Python对象到JSON格式的转换。先看看如何把Python对象变成一个JSON：

import json

d = dict(name='Bob', age=20, score=88)

json.dumps(d)

dumps()方法返回一个str，内容就是标准的JSON。类似的，dump()方法可以直接把JSON写入一个file-like Object。

要把JSON反序列化为Python对象，用loads()或者对应的load()方法，前者把JSON的字符串反序列化，后者从file-like Object中读取字符串并反序列化：

json\_str = '{"age": 20, "score": 88, "name": "Bob"}'

json.loads(json\_str)

由于JSON标准规定JSON编码是UTF-8，所以总是能正确地在Python的str与JSON的字符串之间转换。

###### JSON进阶

Python的dict对象可以直接序列化为JSON的{}，不过，很多时候，更喜欢用class表示对象，比如定义Student类，然后序列化：

import json

class Student(object):

def \_\_init\_\_(self, name, age, score):

self.name = name

self.age = age

self.score = score

s = Student('Bob', 20, 88)

print(json.dumps(s))

运行代码，毫不留情地得到一个TypeError：

Traceback (most recent call last):

...

TypeError: <\_\_main\_\_.Student object at 0x10603cc50> is not JSON serializable

错误的原因是Student对象不是一个可序列化为JSON的对象。

dumps的可选参数就是来定制JSON序列化。前面的代码之所以无法把Student类实例序列化为JSON，是因为默认情况下，dumps()方法不知道如何将Student实例变为一个JSON的{}对象。

可选参数default就是把任意一个对象变成一个可序列为JSON的对象，只需要为Student专门写一个转换函数，再把函数传进去即可：

def student2dict(std):

return {

'name': std.name,

'age': std.age,

'score': std.score

}

这样，Student实例首先被student2dict()函数转换成dict，然后再被顺利序列化为JSON：

print(json.dumps(s, default=student2dict))

不过，下次如果遇到一个Teacher类的实例，照样无法序列化为JSON。可以偷个懒，把任意class的实例变为dict：

print(json.dumps(s, default=lambda obj: obj.\_\_dict\_\_))

因为通常class的实例都有一个\_\_dict\_\_属性，它就是一个dict，用来存储实例变量。也有少数例外，比如定义了\_\_slots\_\_的class。

同样的道理，如果要把JSON反序列化为一个Student对象实例，loads()方法首先转换出一个dict对象，然后，传入的object\_hook函数负责把dict转换为Student实例：

def dict2student(d):

return Student(d['name'], d['age'], d['score'])

运行结果如下：

json\_str = '{"age": 20, "score": 88, "name": "Bob"}'

print(json.loads(json\_str, object\_hook=dict2student))

打印出的是反序列化的Student实例对象。