第6章 标准库与文件操作

学习目标

- 区分模块、库以及标准库
- 了解Python库的导入和使用方法
- 熟悉os、sys、time、random、re标准库的功能和使用
- 解释Python中常见的异常类型
- 了解Python中实现异常捕捉的不同方法及其适用情况
- 了解打开和关闭文件，以及读取和写入文件内容的方法

6.1.1 标准库——概念区分：模块、库与标准库

- 模块是一个包含已定义函数和变量的文件，后缀名为.py，可被别的程序引入以使用模块中定义的函数。
- 如创建一个名为fibo.py的斐波那契数列模块：

```
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result

if __name__ == '__main__':
    f = fib2(10)
    print(f)
```

6.1.1 标准库——概念区分：模块、库与标准库

- 在定义完该模块，将其放置到Python目录下Lib文件夹内或是其他Python解释器能够找到的位置，便可使用import语句导入该模块并使用它：

```
>>> import fibo
>>> fibo.fib(10)
0 1 1 2 3 5 8
```

- 如果模块的数量较多，可以利用库（Package）将这些模块组织起来。库的一个可能结构为：

```
package/
  __init__.py
  subpack1/
    __init__.py
    module_11.py
    module_12.py
  subpack2/
    __init__.py
    module_21.py
    module_22.py
```

6.1.1 标准库——概念区分：模块、库与标准库

- 如果需要使用库，例如将fibo模块组织在nums库里，调用fibo模块的fib函数可以使用如下语句：

```
>>> from nums import fibo
>>> fibo.fib(10)
0 1 1 2 3 5 8
```

- 库可以看做是另一类模块，只是这样的模块里含有多个子模块，为了方便起见，之后的所有内容都会将统称为模块。
- Python内置了一些标准的模块库（Standard Library），这些组件会根据不同的操作系统进行不同形式的配置。常见的标准库有string、re、datetime等。

6.1.2 安装第三方模块

- 第三方Python模块会被贮存在Python包索引（Python Package Index）中，其安装需要通过一些指令完成。
- 最简单的方法是在命令提示符中使用pip语句，以在Windows安装Numpy库为例，打开命令提示符后，输入以下语句：pip install numpy。安装完成后，便可在程序里导入Numpy库并在代码里使用它。

6.1.3 使用import语句导入模块

- 如果想使用一个模块，需要在Python文件里执行import语句。
- 如果import的语句比较长导致后续引用不方便，可以使用import...as...语法重新命名，再通过as后面名字来访问导入的模块：

```
>>> import random as rd
>>> print(rd.uniform(2,3))
2.2569802790968683
```

- 如果不想将整个模块导入而是模块的一部分，可以使用from...import...语句：

```
>>> from random import uniform
>>> print(uniform(2,3))
2.959050824759278
```

6.1.5 常用标准库之一：os

- 该模块是Python标准库中用于访问系统功能的模块，其中比较常用的变量或函数如下：

变量/函数	说明
os.environ	返回包含环境变量的字典
os.sep()	返回用于系统路径的分隔符
os.getcwd()	获取当前路径
os.listdir(path)	获取指定路径下的所有文件和目录名称，默认为当前路径
os.mkdir(path)	在指定路径创建一个目录
os.rmdir(path)	删除指定路径的目录，当且仅当目录存在且目录为空才能删除
os.path.exists(path)	判断文件或目录是否存在，返回True/False
os.path.isfile(path)	判断是否为文件，返回True/False
os.path.isdir(path)	判断是否为文件夹，返回True/False
os.path.join(path, name)	连接目录与文件名称或目录

6.1.5 常用标准库之一：os

- 获取当前程序所在路径：os.getcwd()

```
>>> import os
>>> current_dir = os.getcwd()
>>> print(current_dir)
C:\Users\xxx\Desktop
```

- 获取指定路径下的所有文件和目录名称：os.listdir()

```
>>> print(os.listdir(current_dir))
['test.py', 'test.txt', 'file']
```

- 连接目录与文件名称或目录：os.path.join()

```
>>> new_path = os.path.join(current_dir, 'file')
>>> print(new_path)

C:\Users\xxx\Desktop\file
```

6.1.6 常用标准库之二：sys

- 该模块提供对解释器使用或维护的一些变量访问，以及与Python解释器强烈交互的函数和变量，其中比较常用的变量或函数如下：

变量/函数	说明
argv	命令行参数，包括脚本名称
exit([arg])	退出当前的程序，可选参数为给定的返回值或者错误信息
modules	映射模块名字到载入模块的字典
path	查找模块所在目录的目录名列表
platform	类似sunos5或者win32的平台标识符
stdin	标准输入流
stdout	标准输出流
stderr	标准错误流

6.1.6 常用标准库之二：sys

- 获得脚本参数：sys.argv
  - 该变量是一个包含程序外部输入参数的列表，列表的第一个元素为程序名称，其他元素为用户输入的参数：

```
import sys

a = sys.argv
print(sys.argv[0])

if(len(sys.argv)>1):
    print(sys.argv[1:])
else:
    print('sorry, there is no argument')
```

  - 它的效果应将程序保存后从外部运行并给出参数，如命令提示符下执行以下命令：

    C:\ Users\xxx\Desktop >python test.py 20 21 22

  - 得到结果：

```
test.py
['20', '21', '22']
```

6.1.6 常用标准库之二：sys

- 处理标准输出：sys.stdout
  - sys.stdout.write()和print()都是与输出相关的函数。实际上print()内部也调用了sys.stdout，但是print()可以输出任意类型，sys.stdout.write()只能输出字符串类型；print()默认最后换行，但是sys.stdout.write()默认不换行。
  - 使用print()函数的输出：

```
>>> for i in range(3):
        print('Hello World')


Hello World
Hello World
Hello World
```

  - 改用sys.stdout.write()：

```
>>> import sys
>>> for i in range(3):
        sys.stdout.write('Hello World')

Hello WorldHello WorldHello World
```

6.1.6 常用标准库之二：sys

- 输出错误信息：sys.stderr
  - 当程序崩溃并打印出调试信息的时候，信息前往stderr管道。我们也可以通过该函数自定义错误信息：

```
import sys

def test(x):
    if(x==0):
        sys.stderr.write("x can't be zero")
    else:
        print(4/x)

test(0)
```

```
x can't be zero
```

6.1.7 常用标准库之三：time

- 该模块能够实现获得当前时间、操作时间和日期、从字符串中读取时间以及格式化时间为字符串等功能。
- 日期可以用实数或者是包含有9个整数的元组：

索引	字段	值
0	年	比如2000等
1	月	范围1~12
2	日	范围1~31
3	时	范围0~23
4	分	范围0~59
5	秒	范围0~61
6	周	当周一为0时，范围0~6
7	儒历日	范围1~366
8	夏令时	0、1或-1

6.1.7 常用标准库之三：time

- 该模块中比较常用的函数如下：

函数	说明
asctime([tuple])	将时间元组转换为字符串，默认为当前时间
localtime([secs])	将秒数转换为日期元组，以本地时间为准
mktime(tuple)	将时间元组转换为本地时间
sleep(secs)	休眠（不做任何事情）secs秒
strptime(string[,format])	将字符串解析为时间元组
time()	当前时间（新纪元开始后的秒数，以UTC为准）

6.1.7 常用标准库之三：time

- 将时间元组转换为字符串：asctime()

```
>>> import time
>>> print(time.asctime((2020, 1, 18, 12, 59, 59, 59, 21, 0)))
Thu Jan 18 12:59:59 2020
```

- 将秒数转换为日期元组：localtime()
  - 该函数作用是格式化时间戳为本地的时间，如果不传入任何参数，则返回的是当前时间，输出日期元组是一个struct_time对象：

```
>>> print(time.localtime())
time.struct_time(tm_year=2021, tm_mon=8, tm_mday=31, tm_hour=0, tm_min=22, tm_sec=3
6, tm_wday=1, tm_yday=243, tm_isdst=0)
```

- 将时间元组转换为本地时间：mktime()
  - 与localtime()相反，该接收struct_time对象或者9位整数元组作为参数，返回用秒数来表示时间的浮点数：

```
>>> print(time.mktime((2020, 1, 18, 23, 12, 22, 1, 49, 0)))
1579360342.0
```

6.1.7 常用标准库之三：time

- 将字符串解析为时间元组：strptime()
  - 在将字符串解析为时间元组时，需指定第二个参数说明字符串表示的日期格式：

```
>>> print(time.strptime('30 Jan 20', '%d %b %y'))
time.struct_time(tm_year=2020, tm_mon=1, tm_mday=30, tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=3, tm_yday=30, tm_isdst=-1)
```

  - Python中的部分时间日期的格式化符号如下：

符号	说明	符号	说明
%y	两位数的年份表示（00-99）	%B	本地完整的月份名称
%Y	四位数的年份表示（000-9999）	%c	本地相应的日期表示和时间表示
%m	月份（01-12）	%j	年内的一天（001-366）
%d	月内中的一天（0-31）	%p	本地A.M.或P.M.的等价符
%H	24小时制小时数（0-23）	%U	一年中的星期数（00-53）星期天为星期的开始
%I	12小时制小时数（01-12）	%w	星期（0-6），星期天为0
%M	分钟数（00-59）	%W	一年中的星期数（00-53）星期一为星期的开始
%S	秒（00-59）	%x	本地相应的日期表示

6.1.8 常用标准库之四：random

- 该模块包括返回随机数的函数，可用于模拟或者用于任何产生随机输出的程序，其中比较常用的函数如下：

函数	说明
random()	返回0≤n＜1之间的随机实数n
getrandbits(n)	以长整型形式返回n个随机位
uniform(a,b)	返回随机实数n，其中a≤n＜b
randrange([start],stop,[step])	返回range(start.stop,step)中的随机数
choice(seq)	从序列seq中返回随意元素
shuffle(seq[,random])	原地指定序列seq
sample(seq,n)	从序列seq中选择n个随机且独立的元素

6.1.8 常用标准库之四：random

- 返回0≤n＜1之间的随机实数：random()
  - 该函数是最基本的随机函数之一，它的结果服从[0,1)均匀分布：

```
>>> import random
>>> print(random.random())
0.8378224646486986
```

- 返回随机实数：uniform()
  - 该函数需要提供两个数值参数a和b，它的结果服从[a, b)的均匀分布：

```
>>> print(random.uniform(2,3))
2.957222688879452
```

6.1.9 常用标准库之五：re

- 正则表达式（regex）是由一些特殊符号和字符组成的字符串，这些字符和特殊符号描述了文本的模式，正则表达式能按照模式匹配一系列有相似特征的字符串。
- Python通过标准库的re模块来支持正则表达式。
- 正则表达式常用的特殊符号和字符

表示方法	说明	正则表达式样例
literal	匹配字符串的值	foo
re1|re2	匹配正则表达式re1或re2	foo|bar
.	匹配任何字符（换行符除外）	b.b
^	匹配字符串的开始	^Dear
\$	匹配字符串的结尾	/bin/*sh$
*	匹配前面出现的正则表达式零次或多次	[A-Za-z0-9]*
+	匹配前面出现的正则表达式一次或多次	[a-z]+\.com
?	匹配前面出现的正则表达式零次或一次	goo?

6.1.9 常用标准库之五：re

- 正则表达式常用的特殊符号和字符（续）：

表示方法	说明	正则表达式样例
{N}	匹配前面出现的正则表达式N	[0-9]{3}
{M,N}	匹配重复出现M次到N次的正则表达式	[0-9]{5,9}
[…]	匹配字符组里出现的任意一个字符	[aeiou]
[..x-y..]	匹配从字符x到y的任意一个字符	[0-9],[A-Za-z]
[^…]	不匹配此字符集中出现的任何一个字符，包括某一范围的字符	[^aeiou],[^A-z]
\d	匹配任何数字，和[0-9]一样（\D是\d的反义：任何非数符字）	data\d+.txt
\w	匹配任何数字字母字符，和[A-Za-z0-9]相同（\W是\w的反义）	[A-Za-z]\w+
\s	匹配任何空白符，和[\n\t\r\v\f]相同，（\S是\s的反义）	of\sthe
\b	匹配单词边界（\B是\b的反义）	\bThe\b
\nn	匹配已保存的子组（请参考上面的正则表达式符号：（…））	price:\16
\c	逐一匹配特殊字符c（取消它的特殊含义，按字面匹配）	\., \\, *
\A(\Z)	匹配字符串的起始（结束）	\ADear

6.1.9 常用标准库之五：re

- 管道符号‘|’表示“或”操作，即从多个模式中选择其一的操作，它用于分割不同的正则表达式。
- re模块中比较常用的函数或方法如下：

函数/方法	描述
compile(pattern, flags=0)	对正则表达式pattern进行编译，flags是可选标识符，并返回一个regex对象
group(num=0)	返回全部匹配对象(或指定编号是num的子组)
groups()	返回一个包含全部匹配的子组的元组（如果没有成功，就返回一个空元组）
match(pattern,string, flags=0)	尝试用正则表达式模式pattern匹配字符串string，flags是可选标识符，如果匹配成果，则返回一个匹配对象；否则返回None
search(pattern, string, flags=0)	在字符串string中搜索正则表达式模式pattern的第一次出现，flags是可选标识符，如果匹配成功，则返回一个匹配对象；否则返回None
findall(pattern, string, flags)	在字符串string中搜索正则表达式模式pattern的所有（非重复）出现；返回一个匹配对象的列表
finditer(pattern[, string, flags])	和findall()相同，但返回的是迭代器
split(pattern, string, max=0)	根据正则表达式pattern中的分隔符把字符string分割为一个列表，返回成功匹配的列表，最多分割max次（默认分割所有匹配的地方）
sub(pattern, string, max=0)	把字符串string中所有匹配pattern的地方替换成repl

6.1.9 常用标准库之五：re

- 预编译：compile()
  - 在模式匹配发生之前，正则表达式模式必须先被编译成regex对象。
  - 虽然正则表达式的预编译不是必须的，且re模块的大部分函数或方法也自带编译操作，但预编译可以提高运行效率。
  - re.compile()函数就能实现预编译功能：

```
>>> import re
>>> texts = ['Hello World', 'Come Here', 'I Love You']
>>> regex = re.compile('(\w*o\w*)')
```

  - 在预编译之后，可以使用正则表达式对象的函数或方法：

```
>>> for text in texts:
        m = regex.search(text)
        print(m.group())


Hello
Come
Love
```

6.1.9 常用标准库之五：re

- 匹配对象和group()、groups()方法
  - 在处理正则表达式时，除regex对象外，还有另一种对象类型：匹配对象(match objects)。
  - 这些对象是在match()或search()被成功调用之后所返回的结果。匹配对象主要有两个方法：group()和groups()。
  - group()方法能返回所有匹配对象，或是根据要求返回某个特定子组。groups()方法返回一个包含唯一或所有子组的元组。如果正则表达式中没有子组的话，groups()将返回一个空元组，而group()仍会返回全部匹配对象：

```
>>> for text in texts:
        m = regex.search(text)
        print(m.group())
```

```
Hello
Come
Love
```

```
>>> for text in texts:
        m = regex.search(text)
        print(m.groups())
```

```
('Hello',)
('Come',)
('Love',)
```

6.1.9 常用标准库之五：re

- 起始位置匹配字符串：match()
  - match()从字符串的起始位置开始对模式进行匹配，如果匹配成功，就返回一个匹配对象，如果匹配失败，就返回None。
  - 匹配对象的group()方法可以用来显示那个成功的匹配（注意None是NoneType Object，没有group()属性）：

```
>>> texts = ['Hello World', 'Come Here', 'I Love You']
>>> for text in texts:
        m = re.match('(\w*o\w*)', text)
        print(m)
        print(m.group())

<re.Match object; span=(0, 5), match='Hello'>
Hello
<re.Match object; span=(0, 4), match='Come'>
Come
None
Traceback (most recent call last):
  File "<pyshell#12>", line 4, in <module>
    print(m.group())
AttributeError: 'NoneType' object has no attribute 'group'
```

6.1.9 常用标准库之五：re

- 任意位置匹配字符串：search()
  - search()与match()一样都可以匹配字符串模式，不同之处在于search会检查参数字符串任意位置的地方给定正则表达式的匹配情况。如果搜索到成功的匹配，会返回一个匹配对象，否则返回None。
- 匹配所有字符串：findall()
  - findall()用于非重叠地搜索字符串中正则表达式模式出现的情况。
  - findall()和search()相似之处在于二者都执行字符串搜索；与match()、search()不同之处在于findall()总返回一个列表：

```
>>> texts = ['Hello World', 'Come Here', 'I Love You']
>>> for text in texts:
        m = re.findall('(\w*o\w*)', text)
        print(m)


['Hello', 'World']
['Come']
['Love', 'You']
```

6.1.9 常用标准库之五：re

- 搜索和替换：sub()和subn()
  - sub()和subn()用于完成搜索和替换的功能，两者都能将字符串中正则表达式匹配的部分进行某种形式的替换。
  - subn()函数仅返回返回替换后的结果：

```
>>> m = re.sub(r'(\d{1,2})/(\d{1,2})/(\d{2}|\d{4})', r'\2/\1/\3', '1/18/2020')
>>> print(m)
18/1/2020
```

  - 而subn()函数还会返回一个替换的总数，替换的字符串和该总数一起作为一个元组返回：

```
>>> n = re.subn(r'(\d{1,2})/(\d{1,2})/(\d{2}|\d{4})', r'\2/\1/\3', '1/18/2020')
>>> print(n)
('18/1/2020', 1)
```

6.1.9 常用标准库之五：re

- 字符串分割：split()
  - re.split()与字符串的split()方法相似，前者根据正则表达式模式分割字符串，后者根据固定的字符串分割。
  - 如果分隔符没有使用由特殊符号表示的正则表达式来匹配多个模式，那么re.split()和string.split()的执行过程是一样的，但是使用string.split()方法指定分隔符，只能指定一个，而使用re.split()方法可以通过管道符号'|'指定多个分隔符：

```
>>> m = re.split('&|\*', 'str1*str2&str3')
>>> print(m)
['str1', 'str2', 'str3']
```

6.2 异常

- 当程序在执行的过程中遇到异常情况时，需要引发异常，并告知异常原因。如果异常未被处理或捕捉，程序就会用回溯（Traceback，一种错误信息）终止执行：

```
>>> print(10/0)
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

- Python用异常对象（exception object）来表示异常情况。这些情况可以被引发，并且可以用很多种方法进行捕捉，使得程序可以捉住错误并且对其进行处理，而不是让整个程序失败。

6.2 异常

- Python中常见异常类型如下：

类名	说明
Exception	所有异常的基类
AttributeError	特性引用或赋值失败时引发
IOError	试图打开不存在文件（包括其他情况）时引发
IndexError	在使用序列中不存在的索引时引发
KeyError	在使用映射中不存在的键时引发
NameError	在找不到名字（变量）时引发
SyntaxError	在代码为错误形式时引发
TypeError	在内置操作或者函数应用于错误类型的对象时引发
ValueError	在内置操作或者函数应用于正确类型的对象，但是该对象使用不合适的值时引发
ZeroDivisionError	在除法或者模除操作的第二个参数为0时引发

6.2.1 捕捉异常：try/except语句

- 捕捉异常的一个基本结构是try/except语句，该语句的执行步骤为：
  - 执行try子句，如果没有异常发生，则忽略except子句；
  - 如果发生异常，则try子句中发生错误的语句及余下部分会被忽略，若异常类型和except后的异常类型名称相同，则执行except语句（except后若没有跟异常类型名称默认为全捕捉，即捕捉所有异常）：

```
>>> def test(x):
        try:
                print("Let's begin")
                a = 1/x
                b = x+a
                print('a: {} b: {}'.format(a, b))
        except ZeroDivisionError:
                print("x can't be zero")


>>> test(0)
Let's begin
x can't be zero
```

6.2.1 捕捉异常：try/except语句

- 通过多个except语句可以捕捉不同特定异常，但最多只能执行一个分支：

```
>>> def test(x):
        try:
                print("Let's begin")
                a = 1/x
                b = x+a
                print('a: {} b: {}'.format(a, b))
        except ZeroDivisionError:
                print("x can't be zero")
        except TypeError:
                print("x should be a number")


>>> test('x')
Let's begin
x should be a number
```

6.2.1 捕捉异常： try/except语句

- 如果需要在一个except子句内捕捉多种错误，则需要将多种异常类型名称构成元组：

```
>>> def test(x):
        try:
                print("Let's begin")
                a = 1/x
                b = x+a
                print('a: {} b: {}'.format(a, b))
        except (ZeroDivisionError, TypeError):
                print('wrong')
        except:
                print('Unexpected error')


>>> test('x')
Let's begin
wrong
```

6.2.2 捕捉异常： try/except...else语句

- 可以给try/except语句后加个else子句，else子句只会在try子句没有异常的情况下执行：

```
>>> def test(x):
        try:
                print("Let's begin")
                a = 1/x
                b = x+a
                print('a: {} b: {}'.format(a, b))
        except (ZeroDivisionError, TypeError):
                print('wrong')
        except:
                print('Unexpected error')
        else:
                c = b**x
                print('c: {}'.format(c))


>>> test(2)
Let's begin
a: 0.5 b: 2.5
c: 6.25
```

6.2.3 捕捉异常：try/finally语句

- finally子句可以用来在可能的异常后进行清理，它和try子句联合使用，不管try子句中是否发生异常，finally子句肯定会被执行。
- finally子句和except子句以及else不冲突，但finally子句必须放在最后：

```
>>> def test(x):
        try:
                print("Let's begin")
                a = 1/x
                b = x+a
                print('a: {} b: {}'.format(a, b))
        except (ZeroDivisionError, TypeError):
                print('wrong')
        except:
                print('Unexpected error')
        else:
                c = b**x
                print('c: {}'.format(c))
        finally:
                print('End')
```

```
>>> test('x')
Let's begin
wrong
End
```

6.3 文件与流

- 目前为止介绍的内容都是内部数据的处理，与外部的交互只是通过input()和print()以及sys模块的部分函数。
- 实际上，在很多数据处理过程中，需要与外部数据交互。因此需要对文件和流的相关操作有所了解。
- 本节将介绍一些常用函数和对象，以便在程序执行过程中读取、处理外部数据。

6.3.1 打开和关闭文件

- open()函数可以用来打开文件。
- open()有文件路径、模式（mode）和缓冲（buffering）三个参数，其中文件路径是必不可少的参数，模式和缓冲参数都是可选的。
- 通过调用open()可以返回一个文件对象：

```
>>> f = open(r'c:\test.txt')
```

6.3.1 打开和关闭文件

- 指定模式
  - 如果open()只带一个文件名参数，那么可以获得能读取文件内容的文件对象。
  - 如果要向文件内写入内容，则必须提供一个模式参数来显式声明：

值	说明
'r'	读模式
'w'	写模式
'a'	追加模式
'b'	二进制模式（可添加到其他模式中使用）
'+'	读/写模式（可添加到其他模式中使用）

  - ‘+’参数可以添加到其他任何模式中，指明读和写都是允许的。
  - ‘b’模式可以改变处理文件的方法，该模式通常用于非文本文件，如图像和音频。

6.3.1 打开和关闭文件

- 指定缓冲
  - open()的第3个参数控制着文件的缓冲。一般来说，根据参数buffering的大小可以设置以下三种缓冲方式：
    - 无缓冲，即buffering = 0（或者是False），I/O（输人/输出）就是无缓冲的（所有的读写操作都直接针对硬盘）。
    - 行缓冲，即buffering = 1（或者是True），I/O就是有缓冲的（意味着Python使用内存来代替硬盘，让程序更快，只有使用flush或者close时才会更新硬盘上的数据）。
    - 全缓冲，也即buffering = n（n>1），n的数字代表缓冲区的大小（单位是字节），
    - 比较特殊的是-1（或者是任何负数），其代表使用默认的缓冲区大小。对于二进制文件模式时，采用固定块内存缓冲区方式；对于交互的文本文件，采用一行缓冲区的方式。其它文本文件使用跟二进制一样的方式。

6.3.1 打开和关闭文件

- 关闭文件
  - 在完成对一个文件对象的操作后应使用close()方法进行关闭。虽然一个文件对象在退出程序后（也可能在退出前）会自动关闭，但主动关闭文件可以避免在某些操作系统或设置中进行无用的修改，这样做也会避免用完系统中所打开文件的配额。
  - 如果对文件进行了写入操作，Python可能会出于效率而将写入的临时储存在某处（缓存）而没有真正写入文件，如果程序因某些原因崩溃，那么数据就不会被成功写入到文件中，因此为了安全起见，建议在使用完文件后关闭。
  - 注意如果使用'w'模式打开文件，在不对其进行任何操作的情况下关闭文件，原文件的内容将被删除。

6.3.2 读取文件内容

- 文件（或流）最重要的能力是提供或接受数据。如果有一个类文件对象f，那么就可以用f以字符串形式读取数据。
- 读取类文件对象的常用方法有三种：f.read()、f.readline()、f.readlines()。

- 使用read()方法读取
  - read()有一个参数n，如果设置了参数n，就会读取n个字符。使用不带参数的read()一次性读取整个文件：

```
f = open('./test.txt')
print(f.read())
f.close()
```

6.3.2 读取文件内容

- 使用readline()方法读取
  - readline()方法可以按行读取文件，每调用一次该方法，则会向下读取一行：

```
f = open('./test.txt')
while(1):
    line = f.readline()
    if(not(line)):
        break
    else:
        print('line:', line)
f.close()
```

- 使用readlines()方法读取所有内容成一个列表
  - 使用readlines()方法将所有内容读取成一个列表，列表中每个元素是一行的内容（第0个元素是第1行）：

```
f = open('./test.txt')
line = f.readlines()
print(line)
f.close()
```

6.3.3 写入文件内容

- 文件（或流）还有一个重要的功能是接受数据。如果有一个名为f的类文件对象，那么就可以用f.write()方法或f.writelines()方法写入文件。
- write()的参数必须为string类型，而writelines()不仅可以接受string类型，而且可以接受序列类型。在完成文件操作后使用close()方法关闭文件：

```
f = open('./test1.txt', 'w')
f.write('Hello World')
f.close()

f = open('./test1.txt', 'r')
print(f.read())
f.close()
```

```
Hello World
```

```
f = open('./test2.txt', 'w')
f.writelines(['Hello World','Come Here', 'I Love You'])
f.close()

f = open('./test2.txt', 'r')
print(f.read())
f.close()
```

```
Hello WorldCome HereI Love You
```

- f.writelines()方法将序列直接转换为字符串进行写入操作，如果要使用换行、空格等进行区分，则需要自行加入相应的符号。