# 第 5 章
# 文件操作与管理

每个计算机程序都是用来解决特定问题的。较大规模的程序提供丰富的功能解决完整的计算问题。无论程序规模如何，每个程序都有统一的运算模式：输入数据、处理数据和输出数据。在程序运行时，数据保存在内存的变量里。内存中的数据在程序结束或关机后就会消失。如果想要在下次开机运行程序时还使用同样的数据，就需要把数据存储在不易失的存储介质中，如硬盘、光盘或 U 盘里。不易失存储介质上的数据保存在以存储路径命名的文件中。通过读/写文件，程序就可以在运行时保存数据了。

## 5.1 文件与文件操作

文件是常用的一种存储数据的载体，文件操作则是对数据进行管理的统称。程序经常需要访问文件和目录，读取文件信息或写入信息到文件，在 Python 语言中对文件的读/写是通过文件对象实现的。文件对象可以是实际的磁盘文件，也可以是其他存储或通信设备，如内存缓冲区、网络、键盘和控制台等。

### 5.1.1 文件的定义

在日常生活中，无论是 Word 还是 Excel 文件，人们经常与之打交道。然而，在计算机领域，文件则被定义为存储在辅助存储器上的一组数据序列，其可以包含任何数据内容。从概念上来说，文件是数据的抽象和集合。

### 5.1.2 文件的类型

文件包括两种类型：文本文件和二进制文件。

(1) 文本文件。一般由单一特定编码的字符组成，如 UTF-8、GB2312、ISO-8859-1、GBK 等编码格式，内容容易统一展示和阅读。文本文件存储常规字符串，由若干文本行组成，通常每行使用换行符"\n"结尾，其中字符串指的是记事本或其他文本编辑器能够正常显示、编辑并且能够被人类直接阅读和理解的字符串。

(2) 二进制文件。直接由比特 0 和比特 1 组成，文件内部数据的组织格式与文件用途有关。二进制是信息按照非字符但特定格式形成的文件，例如，jpg 格式的图片文件、mp4 格式的视频文件和 mp3 格式的音频文件等。二进制文件无法用记事本或其他普通文件编辑器直接进行编辑，通常也无法被人类直接阅读和理解，需要使用专门的软件进行解码后读

取、显示、修改和执行。

### 5.1.3 文件的操作与管理

在 Python 程序设计中，文件有多种操作和管理方式，文件操作主要包括对文件内容的读/写操作，这些操作是通过文件对象实现的，通过文件对象可以读写文本文件和二进制文件。文本文件可以处理各种语言所需的字符，只包含基本文本字符，不包括诸如字体、字号、颜色等修饰的信息。它可以在文本编辑器和浏览器中显示。即在任何情况下，文本文件都是可读的。使用其他编码方式的文件即二进制文件，如 Word 文档、PDF 文档、图像文件和可执行文件等。

**1. 文件的打开与关闭**

Python 对文本文件和二进制文件采用统一的操作步骤，即“打开—操作—关闭”，对文件首先需要将其打开，使得当前程序有权操作这个文件，打开不存在的文件可以创建文件。打开后的文件处于占用状态，此时，其他程序不能操作这个文件。接下来可以通过一组方法读取文件的内容或向文件写入内容。文件操作完成后需要将文件关闭，关闭释放对文件的控制使文件恢复存储状态，此时，别的程序才能够操作这个文件。

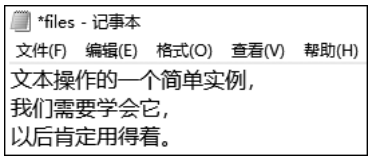


图 5-1 files.txt 的文本内容

在学习文件的打开和关闭操作之前，需要在计算机的任意盘上创建文件并储入些许文本内容。本书在此以 TXT 文本文档为例，编辑的文件名为 files.txt，文件的存储路径为 C:\Users\test2\Desktop\files.txt，编辑的文本内容如图 5-1 所示。

创建好文本文件后，可以通过 open()方法和 close()方法分别对文件进行打开和关闭操作。Python 通过 open()方法打开一个文件，并返回一个操作这个文件的变量，语法形式如下，其中可选参数＜打开模式＞的类型见表 5-1。

**表 5-1 open()方法的可选参数＜打开模式＞的类型**

| 打开模式 | 含　义 |
|---|---|
| "r" | 只读模式，如果文件不存在，返回异常 FileNotFoundError，默认值 |
| "w" | 覆盖写模式，文件不存在则创建，存在则完全覆盖源文件 |
| "x" | 创建写模式，文件不存在则创建，存在则返回异常 FileExistsError |
| "a" | 追加写模式，文件不存在则创建，存在则在源文件最后追加内容 |
| "b" | 二进制文件模式 |
| "t" | 文本文件模式，默认值 |
| "+" | 与 r/w/x/a 一同使用，在原功能基础上增加同时读写的功能 |

**注意**：打开模式使用字符串方式表示，根据字符串定义，单引号或者双引号均可。上述打开模式中，'r'、'w'、'x'、'a'可以和'b'、't'、'+'组合使用，形成既表达读写又表达文件模式的方式。

```
<文件对象名>=open(<文件路径及文件名>, <打开模式>,<编码格式>)
```

其中文件路径及文件名是不可省略的，其他参数都可以省略，省略时会使用默认值。

Python 用 close()方法关闭、释放文件的使用授权，语法形式如下：

```
<文件对象名>.close()
```

**【例 5.1】** 通过 open()和 close()方法打开和关闭文件 files.txt。

```
#L5.1 例 5.1
f =open(r"C:\Users\test2\Desktop\files.txt", "rt", encoding="UTF-8")
print(f.readlines())
f.close()
```

执行结果：

```
['文本操作的一个简单实例,\n', '我们需要学会它,\n', '以后肯定用得着。']
```

读者在实践中可能会发现当输入的文件路径为 C:\Users\Administrator\Desktop\files.txt 时，Python 解释器会反馈语法错误 SyntaxError。其实这是需要被提醒的一个非常重要的点，即在输入文件路径及文件名时，要么以 r"C:\Users\Administrator\Desktop\files.txt"的方式，要么以"C:/Users/Administrator/Desktop/files.txt"的方式输入，否则 Python 会反馈语法错误。

此外，一些用户在打开文件进行操作之后可能会忘记关闭文件，这样会造成文档信息丢失等不良后果，因此 Python 提供了一种名叫上下文管理器的模式来进行文件打开和关闭操作，以防信息丢失。

**【例 5.2】** 通过上下文管理器打开和关闭文件 files.txt。

```
#L5.2 例 5.2
with open(r"C:\Users\test2\Desktop\files.txt", "rt", encoding="UTF-8") as f:
    print(f.readlines())
```

执行结果：

```
['文本操作的一个简单实例,\n', '我们需要学会它,\n', '以后肯定用得着。']
```

编码格式指定文件的编码模式，一般可设定为 cp950(繁体中文 BIG5)或 UTF-8。默认的编码因操作系统不同而有所不同，如果是 Windows 简体中文系统，默认的编码是 cp936(GBK)，也就是记事本“存储为”界面中显示的 ANSI 编码。Windows 简体中文的记事本默认使用的是 ANSI 编码存储文本文件，因此在打开文件时不需要输入编码模式。但是如果在指定 encoding = 'cp936'的情况下去读取用 UTF-8 编码的文件，那么显示出来的数据内容有时会出现乱码。由于许多操作系统默认使用 UTF-8 编码，因此建议将文件保存为 UTF-8 编码，而不是 ANSI 形式。如果文件编码已改为 UTF-8，那么读取文件时要加入 encoding = 'UTF-8'编码格式，否则会出现错误。

**2. 文件的读写**

除了文件的打开与关闭操作，在编写程序的过程中，经常还需要对文件进行读写。

通常来说，与文件打开方式相同，文件读写方式也会根据文本文件或二进制文件的不同而有所不同。常用的关于读取文件的方法及含义如表 5-2 所示。

**表 5-2 常用的文件读取的方法及含义**

| 方 法 | 含 义 |
| --- | --- |
| 文件对象名.read(size=－1) | 从文件中读入整个文件内容。参数可选，如果给出，读入前 size 长度的字符串或字节流 |
| 文件对象名.readline(size = －1) | 从文件中读入一行内容。参数可选，如果给出，读入该行前 size 长度的字符串或字节流 |
| 文件对象名.readlines(hint=－1) | 从文件中读入所有行，以每行为元素形成一个列表。参数可选，如果给出，读入 hint 行 |
| 文件对象名.seek(offset) | 改变当前文件操作指针的位置。offset 的值：0 代表文件开头；2 代表文件结尾 |

当文件中存储的内容不多时，可以使用 read()方法一次性将文本内容读取到文件对象中，该方法也是 Python 程序设计中常用于一次性读入文本内容的方法。

**【例 5.3】** 通过 read()方法读取文件 files.txt 中存储的内容。

```
#L5.3 例 5.3
with open(r"C:\Users\test2\Desktop\files.txt", "rt", encoding="UTF-8") as f:
    contents =f.read()
print(contents)
```

执行结果：

```
文本操作的一个简单实例，
我们需要学会它，
以后肯定用得着。
```

当文件中存储的内容过大时，通常采用 readline()方法对文件中存储的内容进行读取操作。readline()方法主要用于从文件中按行读取存储的内容，值得注意的是，当读取的内容超过文本中存储的行数时，该函数读取到的内容为空。

**【例 5.4】** 通过 readline()方法按行读取文件 files.txt 中存储的内容。

```
#L5.4 例 5.4
with open(r"C:\Users\test2\Desktop\files.txt", "rt", encoding="UTF-8") as f:
    contents1 =f.readline()
    contents2 =f.readline()
    contents3 =f.readline()
    contents4 =f.readline()
print(contents1)
print(contents2)
print(contents3)
print(contents4)
```

执行结果：

```
文本操作的一个简单实例,

我们需要学会它,

以后肯定用得着。
```

readlines()方法和 read()方法相似，也是用于一次性将文本内容读入文件对象的方法，不过 readlines()方法读入的文本内容是以列表的形式存储在文件对象中的，而 read()方法则是直接存储在文件对象中。

**【例 5.5】** 通过 readlines()方法读取文件 files.txt 中存储的内容。

```
#L5.5 例 5.5
with open(r"C:\Users\test2\Desktop\files.txt", "rt", encoding="UTF-8") as f:
    contents =f.readlines()
print(contents)
```

执行结果：

```
['文本操作的一个简单实例,\n', '我们需要学会它,\n', '以后肯定用得着。']
```

在 seek()方法的使用中引入了文件读取指针的概念，在文件打开后，对文件的读写有一个读取指针，当从文件中读入内容后，读取指针将向前进，再次读取的内容将从指针的新位置开始。比如当使用完 read()方法对文件进行读取之后，读取指针将指向文件的末尾，如果再次使用其他读取函数对文件进行读取的话由于读取指针无法读取到任何内容，因此返回的结果将为空。此时，seek()方法便可以发挥其作用，移动读取指针，令读取指针指向文件的其他位置。

**【例 5.6】** 通过 seek()方法移动文件读取指针。

```
#L5.6 例 5.6
with open(r"C:\Users\test2\Desktop\files.txt", "rt", encoding="UTF-8") as f:
    contents =f.readlines()
                          #通过 readlines()方法读取文件 files.txt 中存储的全部内容
    contents1 =f.readlines()
            #再次调用 readlines()方法读取文件 files.txt 中存储的内容,将返回一个空列表
    print("第一次读取的内容: ", contents)          #将读取的全部内容打印到屏幕上
    print("再次读取的内容: ", contents1)           #将返回的空列表打印到屏幕上
    f.seek(0)                                     #将文件指针移动到文件开头
    contents1 =f.readlines()    #调用 readlines()方法读取文件 files.txt 中存储的内容
    print("读取的内容: ", contents1)               #将读取的文件内容打印到屏幕上
```

执行结果：

```
第一次读取的内容:['文本操作的一个简单实例,\n', '我们需要学会它,\n', '以后肯定用得着。']
再次读取的内容:[]
读取的内容:['文本操作的一个简单实例,\n', '我们需要学会它,\n', '以后肯定用得着。']
```

除了以上 4 个常用的关于文件读取的方法之外,从文本文件按行读取内容并进行处理也是文件读取的一个常用操作,由于文本文件可以看作由行组成的对象,因此可以使用循环遍历来对文件进行读取和处理操作。

**【例 5.7】** 通过循环读取文件 files.txt 中存储的内容。

```
#L5.7 例 5.7
with open(r"C:\Users\test2\Desktop\files.txt", "rt", encoding="UTF-8") as f:
    for line in f:          #使用 for 循环读取文件 files.txt 中存储的内容
        print(line)         #将 for 循环按行读取到的内容打印到屏幕上,也可进行其他的处理
操作
```

执行结果:

```
文本操作的一个简单实例,

我们需要学会它,

以后肯定用得着。
```

接下来将介绍关于文件写的方法,常用的文件写的方法及其含义如表 5-3 所示。

**表 5-3 常用的文件写的方法及含义**

| 方　法 | 含　义 |
|---|---|
| 文件对象名.write(str) | 向文件写入一个字符串或字节流 |
| 文件对象名.writelines(lists) | 将一个字符串列表写入文件 |

write()方法用于向指定文件中写入字符串,每次写入后,都会记录一个写入指针。该方法可以反复调用,将在写入指针后分批写入内容,直至文件被关闭。

**【例 5.8】** 通过 write()方法向指定文件中写入内容。

```
#L5.8 例 5.8
with open(r"D:\example.txt", "w", encoding="UTF-8") as f:
    f.write("读者,您好!\n")          #向指定文件 example.txt 中写入字符串"读者,您好!"
    f.write("这是文件写的一个示例程序。")
                    #向指定文件 example.txt 中追加写入字符串"这是文件写的一个示例程序。"
```

以上语句运行后将在 D 盘目录下生成一个文件 example.txt,并写入如图 5-2 所示的内容。当使用 write()方法时,要显式地使用换行符"\n"对写入的文本进行换行,如果不进行换行的话,每次写入的字符串会被连接起来。

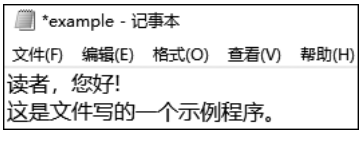


图 5-2 使用 write()方法写入的内容

writelines()方法用于直接将列表类型的各元素连

接起来写入文件。

【例 5.9】　通过 writelines()方法向指定文件中写入内容。

```
#L5.9 例 5.9
lists =["1, 2, 3\n", "读者,您好!\n", "这是文件写的另一个示例程序。\n"]
with open(r"D:\example1.txt", "w", encoding="UTF-8") as f:
    f.writelines(lists)          #向文件 example1.txt 中写入 lists 列表
```

以上语句运行后将在 D 盘目录下生成一个文件 example1.txt,并写入如图 5-3 所示的内容。

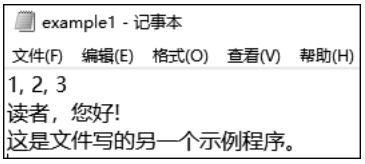


图 5-3　使用 writelines()方法写入的内容

## 5.2　os 模块的使用

OS,操作系统(operating system)的缩写形式。很明显,os 模块可以直接对操作系统进行操作,因此,os 模块是 Python 语言中相当重要的一个模块。使用 os 模块,可以访问操作系统提供的功能。此外,通过使用 os 模块中提供的接口,还可以实现跨平台访问。本节将介绍 os 模块的系统操作、对目录的增删改查操作及 path 模块中基本方法的使用。

### 5.2.1　os 模块的系统操作

在 Python 语言中,os 模块用于系统操作的基本方法有 4 个,这 4 个基本方法及含义如表 5-4 所示。

表 5-4　os 模块用于系统操作的基本方法及含义

| 方　法 | 含　义 |
| --- | --- |
| os.sep | 用于系统路径的分隔符,其中 Windows 系统的分隔符为“\”或“\\”;Linux 类系统如 Ubuntu 的分隔符是“/” |
| os.name | 指示当前使用的工作平台,比如对于 Windows 用户,它是“nt”;对于 Linux 或 Unix 用户,它是“posix” |
| os.getenv(环境变量名称) | 读取环境变量 |
| os. getcwd() | 获取当前的工作路径 |

【例 5.10】　通过 os 模块的 4 个基本方法查看当前操作系统的相关信息。

```
#L5.10 例 5.10
import os                        #导入 os 模块
print(os.sep)                    #打印当前系统的分隔符
```

```
print(os.name)              #打印当前正在使用的工作平台信息
print(os.getenv("path"))    #打印出系统当前配置的环境变量(每个人的配置可能有所不同)
print(os.getcwd())          #打印当前的工作路径
```

执行结果：

```
\
nt
H:\Anaconda3\envs\tensorflow; H:\Anaconda3\envs\tensorflow\Library\usr\bin; H:\
Anaconda3\envs\tensorflow\Library\bin
G:\PythonProjects\figure_examples
```

### 5.2.2 对目录和文件的管理

对目录的增删改查操作是 Python 程序设计中必不可少的一部分，在 os 模块中用于对目录进行增删改查操作的基本方法及含义如表 5-5 所示。

**表 5-5 os 模块中用于对目录进行增删改查操作的基本方法及含义**

| 方 法 | 含 义 |
|---|---|
| os.listdir() | 返回指定目录下的所有目录和文件名 |
| os.mkdir(dirname) | 创建一个目录文件。若目录已经存在，则创建目录失败 |
| os.rmdir(dirname) | 删除一个空目录。若目录中有文件则无法删除 |
| os.makedirs(dirname) | 可以生成多层递归目录。若目录全部存在，则创建目录失败 |
| os.removedirs(dirname) | 可以删除多层递归的空目录，若目录中有文件则无法删除 |
| os.remove(filename) | 删除 filename 参数指定的文件 |
| os.chdir() | 改变当前目录，到指定目录中 |
| os.rename() | 重命名目录名或者文件名。若重命名后的文件名已存在，则重命名失败 |

由于表 5-5 中列出的方法的使用流程基本相同，因此仅以 listdir()和 mkdir()方法为例，剩余的方法请自己实践完成。

**【例 5.11】** 通过 os 模块中的 listdir()和 mkdir()方法实现对目录的操作。

```
#L5.11 例 5.11
import os                       #导入 os 模块
print(os.listdir(r"G:\\"))      #打印出 G 盘下的所有目录和文件名
os.mkdir(r"G:\examples")        #在 G 盘下创建 examples 目录
```

以上语句运行后将在屏幕上打印出 G 盘下的所有目录和文件名，并在 G 盘下创建了 examples 目录文件。

**【例 5.12】** 通过 os 模块中的 remove()方法实现对文件的删除操作。

```
#L5.12 例 5.12
import os                       #导入 os 模块
```

```
file ="testfile.txt"
if os.path.exists(file):          #检查当前路径下 testfile.txt 是否存在
    os.remove(file)               #如果文件存在则删除该文件
else:
    print(file+"文件未找到!")
```

【例 5.13】 通过 os 模块中的 system()方法管理目录。

```
#L5.13 例 5.13
import os                                                    #导入 os 模块
my_path =os.path.dirname(__file__)                           #取得当前路径
os.system("cls")                                             #清除屏幕
os.system("mkdir testdir")                                   #创建 testdir 目录
os.system("copy testfile.txt testdir\copytestfile.txt")#复制文件
file =my_path +"\testdir\copytestfile.txt"
os.system("notepad" +file)                          #以记事本方式打开 copytestfile.txt
```

### 5.2.3　path 模块中基本方法的使用

path,汉语翻译为路径,顾名思义,path 模块与目录或文件路径的操作相关。在 path 模块中用于对文件或目录路径进行操作的基本方法及含义如表 5-6 所示。

**表 5-6　path 模块中用于对文件或路径进行操作的基本方法及含义**

| 方　法 | 含　义 |
|---|---|
| os.path.exists(path) | 判断文件或者目录是否存在,存在则返回 True,否则返回 False |
| os.path.isfile(path) | 判断是否为文件,是文件则返回 True,否则返回 False |
| os.path.isdir(path) | 判断是否为目录,是目录则返回 True,否则返回 False |
| os.path.basename(path) | 返回文件名 |
| os.path.dirname(path) | 返回文件路径 |
| os.path.getsize(name) | 获得文件大小 |
| os.path.abspath(name) | 获得文件或目录的绝对路径 |
| os.path.join(path, name) | 连接路径与文件/目录 |

【例 5.14】 通过 path 模块中的方法实现对目录或文件的操作。

```
#L5.14 例 5.14
import os                                    #导入 os 模块
path =r"G:\\"                                #定义 path 路径
os.mkdir(r"G:\examples")
              #在 G 盘下创建 examples 目录,如果 G 盘下存在 examples 目录,则先删除再运行
print(os.path.exists(r"G:\examples"))   #判断 G 盘下是否存在 examples 目录
print(os.path.isfile (r"G:\examples"))  #判断 G 盘下的 examples 是否为文件
print(os.path.isdir (r"G:\examples"))   #判断 G 盘下的 examples 是否为目录
```

```
print(os.path.basename (r"G:\examples"))      #打印 G 盘下的 examples 的文件名
print(os.path.dirname (r"G:\examples"))       #打印 G 盘下的 examples 的父路径
print(os.path.getsize (r"G:\examples"))       #打印 G 盘下的 examples 目录的大小
print(os.path.abspath (r"G:\examples"))       #打印 G 盘下的 examples 目录的绝对路径
new_path =os.path.join(path, "examples")   #连接 path 路径和 examples 目录,从而形成新的路径
print("新的路径为: ",new_path)
```

执行结果：

```
True
False
True
examples
G:\
0
G:\examples
新的路径为: G:\\examples
```

## 5.3 数据的处理

数据的存储和处理是 Python 程序设计中必不可少的一个模块,本节将介绍数据的组织维度以及一维数据(如列表)、二维数据(如表格数据)和多维数据(如嵌套字典)的存储与处理。

### 5.3.1 数据的组织维度

数据在被计算机处理前,都需要一定的组织形式来表明数据之间的逻辑和关系,而不同的组织形式也就形成了不同的“数据的组织维度”。根据数据组织形式的不同,可以将数据的维度分为一维数据、二维数据和多维数据。

### 5.3.2 一维数据的存储与处理

一维数据(如 C/C++ 语言中的一维数组)是最简单的数据组织形式,由于采用线性关系的方式进行组织,所以在 Python 语言中主要采用 list 列表的形式表示。

一维数据的文件存储方式有多种,通常采用拼接逗号作为分隔元素的方式对其进行存储。使用逗号分隔方式存储的文件叫作 CSV(comma-separated values,逗号分隔值)文件,在 CSV 文件中,各元素间被逗号分隔,形成一行数据。在对 CSV 文件中存储的一维数据进行处理时,需要以 CSV 格式读入和输出。

常见的一维数据如姓名列表：

```
data =["张三", "李四", "王五"]
```

**【例 5.15】** 使用 CSV 格式对一维数据进行读写处理。

```
#L5.15 例 5.15
data =["张三","李四","王五"]          #定义一维数据
```