

# 第六章 面向对象编程基础

在前面的章节中，我们解决问题的思想是先把一个问题分解为几个步骤，然后逐一实现每个步骤，如果某个步骤比较复杂，还需要把这个步骤分解成许多子步骤，直到问题得以解决为止。我们把这种自顶向下、逐步求精、分而治之的编程过程叫做面向过程编程。面向过程编程关注每个过程的具体实现。随着计算机技术的发展，软件越来越复杂，面向过程编程的方法已经难以设计出大型软件。20 世纪 60 年代，人们提出了面向对象编程(Object Oriented Programming, 即 OOP) 的思想，比面向过程编程具有更强的灵活性和扩展性，可以使软件设计更加灵活，并且能更好地进行代码复用。本章我们介绍面向对象编程技术的基础知识。

## 6.1 面向对象编程概述

面向对象编程的思想是这样的：把要解决的问题分解成很多对象，编程人员主要关注在什么条件下对象做什么事情，而不关注对象做事情的具体过程。想象这样的场景：屏幕下方每隔 1 秒就会出现大小、颜色各异的多个气球，每个气球缓缓上升，最后飘出屏幕。该怎样模拟这个场景呢？

面向对象编程的思想：把气球当成对象，气球的大小、颜色、运动由气球自己决定。设计一个循环，每隔 1 秒执行 1 次，在每次循环中产生几个新气球即可。

面向过程编程的思想：首先设计好控制气球大小、颜色、运动的各个函数。设计一个循环，每隔 1 秒执行 1 次，在每次循环中去调用这些函数。

面向对象编程的基础是对象，每个对象都有属于自己的数据（属性）和操作这些数据的函数（方法）。在设计软件时，首先要仔细分析每个对象都有哪些属性以及哪些方法，构造出对象的“模板”，然后再根据这个“模板”生成具体的对象。

学习面向对象编程，需要理解两个基本概念：类、对象，灵活运用三大特点：封装、继承、多态。

### (1) 类

类就是“模板”，是用来生成具体对象的“模型”。比如工厂生产玩具的模具就是类。类是对现实生活中一类具有共同特征的事物的抽象，是一种自定义数据类型，每个类都包含相应的数据（属性）和操作数据的函数（方法）。编写类是面向对象编程的前期主要工作。

### (2) 对象

对象是根据类创建的一个个实体，比如工厂根据模具生产出来的具体的玩具。

### (3) 封装

封装是面向对象编程的核心思想，是指将对象的属性和方法绑定到一起封装起来的过程。可以选择性地隐藏属性和隐藏实现细节。这就是封装的思想。

### (4) 继承

继承是指类与类之间的关系，如果一个类(A)除了具有另一个类(B)的全部功能外，还有自己的特殊功能，这时类 A 就可以继承于类 B，从而减少代码的书写，提高代码复用性。

### (5) 多态

多态是指子类和父类具有相同的行为名称，但这种行为在子类和父类中表现的实际效果却不相同。比如父亲有“跑”的行为，儿子也有“跑”的行为，但父亲跑得更快一些，儿子跑得慢一些。具体实现方法是在子类中重写父类的方法。

## 6.2 类的创建与使用

### 6.2.1 类的创建

在 Python 中，定义类的基本语法如下：

```
class ClassName():  
    class_suite #类体
```

ClassName: 类的名字, Python 建议类名采用“大驼峰式命名法”(即每个英语单词的首字母大写)。class\_suite:

类体，主要是由属性、方法等语句组成。

例子，定义一个 Dog 类，代码如下

```
01 class Dog():
02     def run(self):
03         print("Dog is running")
```

## 6.2.2 创建类实例

上面定义了一个 Dog 类，但仅仅是一个“模具”，有了这个“模具”，我们就可以创建很多“狗”，根据“模具”创建“狗”，称为创建类的实例（对象），语法如下：

```
object = ClassName ()
```

ClassName: 类名，object: 根据类创建的实例对象。比如根据前面的 Dog 类创建一个实例对象：

```
dog = Dog ()      #创建 Dog 类实例
dog.run()         # 调用 dog 的 run 方法
```

运行效果如图 6.1.1。



图 6.1.1

在上面定义 Dog 类的代码中，run(self) 方法中的 self 是指实例本身，Python 解释器会自动把实例对象本身传入，无需显示传入，如 dog.run(dog)，将会引发“TypeError”异常。

## 6.2.3 \_\_init\_\_() 方法

Python 中，当实例对象被创建或销毁时，会默认调用一些特殊方法。其中 \_\_init\_\_() 方法就是当创建一个实例对象时，将会自动调用的方法。通常情况，我们会把一些需要对对象进行初始化的操作放在这个方法里面。现在改写上面的 Dog 类，当创建对象时，完成对颜色、体重、身高进行初始化的功能。

```
01 class Dog():
02     def __init__(self,color,weight,height):
03         self.color = color
04         self.weight = weight
05         self.height = height
06 dog = Dog('black' ,30,40)
07 print(dog.color,dog.weight,dog.height)
```

输出: black 30 40

03、04、05 行代码的功能是创建实例属性并初始化。

**说明:**

- (1) \_\_init\_\_() 方法中必须要有 1 个表示实例对象本身的参数，习惯取名为 self;
- (2) \_\_init\_\_() 方法中除了第 1 个参数表示实例对象本身的参数外，还可以自定义其它参数，参数之间用逗号

号“,” 隔开。

## 6.2.4 类成员创建与访问

类的成员主要由类方法、实例方法、类属性和实例属性、静态方法等组成。下面先给出一个例子说明这些成员在形式上的区别：

```
01 class Student():
02     sum = 0                #类属性
03     def __init__(self,name): #实例方法
04         self.name = name   #实例属性
05         sum +=1
06     def study(self):       #实例方法
07         print(self.name +" is learning.....")
08     @classmethod           #类方法装饰器
09     def get_total(cls):    #类方法
10         print("学生总数: %d" %(sum))
11     @staticmethod         #静态方法装饰器
12     def print_sum():      #静态方法
13         print(Student.sum)
```

通过前面章节的学习，已经熟悉了变量和函数的概念，在类体中，我们把与变量类似的对象称为**属性**，与函数类似的对象称为**方法**。凡是前面有“self.”标识的属性叫做**实例属性**，没有这种标识的叫做**类属性**，把参数中有“self”的方法叫做实例方法，把用@classmethod 装饰器装饰的方法叫做**类方法**，把用@staticmethod 装饰器装饰的方法叫做**静态方法**。

本节只介绍实例方法和属性的创建与访问。

### (1) 实例方法的创建和访问

创建实例方法的语法格式如下：

```
def functionname(self,parameterlist):
    block
```

functionname: 方法名，一般使用小写字母开头。self: 必要参数，表示类的实例对象，名称可以是任何合法的Python 标识符，使用self 只是习惯而已。parameterlist: 其它参数，参数之间用逗号“,” 隔开。block: 方法中的语句块。

访问实例方法的语法格式如下：

实例名. functionname(parameterlist)。

### (2) 属性的创建和访问

根据属性定义时的位置，属性分为类属性和实例属性。

**类属性**：定义在类中，并且在方法体外的属性叫做类属性。类属性主要用于在类的实例对象之间共享值。访问类属性可以通过“类名.类属性”或“实例名.类属性”访问。

**实例属性**：定义在方法体中的属性叫做实例属性。实例属性属于实例对象所有。访问实例属性只能通过“实例名.实例属性”访问。

例子：创建一个学生类，能记录通过该类创建的男生与女生的个数。

```
01 class Student():
02     male = 0
03     female = 0
```

```

04 def __init__(self, name, sex):
05     self.name = name
06     self.sex = sex
07     if sex == '男':
08         Student.male +=1
09     else:
10         Student.female +=1
11 zhang = Student('张帅', '男')
12 wang = Student('王兰', '女')
13 print(Student.male, Student.female)           #通过类名访问类属性
14 print(zhang.male, wang.female)               #通过实例名访问类属性
15 print(zhang.name)                             #访问实例属性

```

输出:

```

1 1
1 1
张霞

```

说明: 通过“实例名.类属性 = 值”的形式并不能修改类属性, 而是增加同名的实例属性。

**知识拓展:** 创建实例对象时, Python 解释器会根据类的定义, 为每个实例划分一块内存空间用于保存实例属性和一些特殊的属性, 类属性、实例方法、类方法并不复制到这块空间中。

## 6.3 数据封装与访问限制

### 6.3.1 数据封装

封装是面向对象编程的核心思想, 封装的主要目的是把属性和方法绑定到一起, 以接口的形式提供给使用者, 使用者不必了解接口内部是怎么实现的, 只需调用接口就可以获得希望的结果。

例子: 打印名片。

```

01 class Card():
02     def __init__(self, name, sex, tel, address):
03         self.name = name
04         self.sex = sex
05         self.tel =tel
06         self.address = address
07     def print_card(self):           #print_card()就是对外的接口
08         print('='*30)
09         print('姓名: ', self.name, ''*10, '性别: ', self.sex)
10         print('联系电话: ', self.tel)
11         print('地址: ', self.address)
12         print('='*30)
13 p = Card('张三', '男', 13888888888, '贵州省贵阳市云岩区飞山街')
14 p.print_card()

```

输出:

```

=====
姓名: 张三 性别: 男
联系电话: 13888888888
地址: 贵州省贵阳市云岩区飞山街
=====

```

解析: 面向对象编程的基本单元是对象, 用户只需关注对象能做什么, 不需要考虑这些功能是怎么实现的, 直

接调用即可。在本例中我们生产了 1 个对象 p，接下来只需关注 p 能做什么就行了，而不考虑怎样把 p 中的数据提取出来，以及如何打印成名片。如果把定义类的代码看成 1 条语句的话，这段代码就只有 3 条语句，定义类、创建实例对象、调用对象的接口。这就是面向对象编程的思想。根据经验，在分析面向对象编程时，考虑定义类是一个生产者，使用类是另一个人（用户，消费者）。

但是，我们总得要在某个地方编写打印名片的具体过程，由于名片所需要的数据都来自实例本身，因此，可以把这些具体的操作过程写到类里面，对外提供 print\_card() 方法，供用户直接调用，这样用户就不需要知道（关注）print\_card() 方法内部是如何实现的，从而达到了隐藏内部的复杂逻辑的目的，这就是封装，这里，print\_card() 方法也称接口，API（应用程序接口）。电脑主机上的接口也是这个意思。

为什么使用 Python 编程简单，原因就是它丰富的库已经为我们写好了很多类，里面封装了很多接口，我们只需调用这些接口就能完成几乎所有功能。

### 6.3.2 访问限制

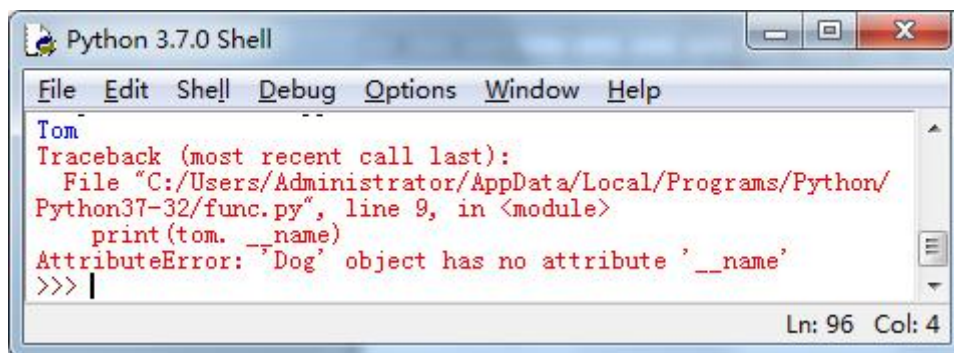
在 Python 中，可以通过类似“实例名.实例属性”的形式直接修改数据，有时需要禁止这种访问方式来保证数据的完整性和有效性，Python 提供的解决方案是在属性和方法名前面加下划线来限制访问权限。

- (1) 单下划线，表示只运行类本身和子类访问，但不能使用 from 模块 import \* 语句导入；
- (2) 首尾双下划线，表示特殊方法；
- (3) 只有双下划线开头，只允许定义该方法的类本身进行访问，也不能通过实例对象来访问。

例子：实践双下划线限制访问权限。

```
01 class Dog():
02     def __init__(self, name):
03         self.__name = name           #加双下划线，限制访问
04     def get_name(self):
05         return self.__name          #在类中访问
06 tom = Dog("Tom")
07 print(tom.get_name())
08 print(tom.__name)                  #通过实例对象访问
```

运行结果如下图：



```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Tom
Traceback (most recent call last):
  File "C:/Users/Administrator/AppData/Local/Programs/Python/Python37-32/func.py", line 9, in <module>
    print(tom.__name)
AttributeError: 'Dog' object has no attribute '__name'
>>> |
Ln: 96 Col: 4
```

可以看出，在类中是能够访问的，但不能通过实例对象访问，从而达到了保护的目的。

**知识拓展：**（1）加双下划线后，Python 内部做了什么事情？其实 Python 只是在这些属性和方法名前增加了“\_类名”而已，因此，即使加了双下划线，也可以通过“实例名.\_类名\_XXX”方式访问，比如本例中就可以使用 tom.\_Dog\_\_name 访问。（2）对属性的进一步控制可以通过@property 装饰器实现，请参考其它资料。

## 6.4 继承和多态

### 6.4.1 继承



我们在前面定义了一个 Dog 类，狗有很多种类，假设现在需要一个哈士奇 (Husky) 类，是不是要重新定义一个新类呢？哈士奇具有狗的一般特征，同时又有自己的特征。与自然界中的遗传类似，由此我们可以想到让哈士奇类继承于狗类。

继承的语法格式如下：

```
class ClassName(baseclasslist):  
    class_suite #类体
```

ClassName: 类的名字。baseclasslist : 要继承的基类列表，类名之间用逗号 (,) 隔开，如果不指定，默认为所有类的根: object。class\_suite: 类体，主要是由属性、方法等语句组成。

在继承关系中，把被继承的类叫做父类或基类，新的类叫子类或派生类。下面写出 Husky 类继承于 Dog 类的例子。

```
01 class Dog():  
02     def run(self):  
03         print("Dog is running")  
04 class Husky(Dog):  
05     pass  
06 hsk= Husky( )  
07 hsk.run()
```

输出: Dog is running

说明 Husky 类继承了 Dog 类的 run 方法。下面我们编写一个案例来探索下面这些问题：

- (1) 什么样的属性和方法可以被继承？
- (2) 当多个父类中有相同的属性和方法时，是继承哪一个父类的？
- (3) 怎样重写父类方法？

```
01 class Animal(object): #定义 Animal 类  
02     def __init__(self):  
03         self.ears = 2  
04     def eat(self):  
05         print("Animal is eating .....")  
06 class Dog(Animal): #定义 Dog 类，继承 Animal 类  
07     def __init__(self):  
08         super().__init__()  
09         self._name = 'dog'  
10         self.__color = 'white' #定义私有属性  
11         self.leg = 4  
12     def run(self):  
13         print("Dog is running")  
14     def eat(self):  
15         print('Dog is eating .....')  
16 class Shape(object): #定义 Shape 类  
17     def __init__(self):
```

```

18         self.height = 40
19 class Husky(Dog, Shape):           #定义 Husky, 同时继承 Dog 类, Shape 类
20     def __init__(self):
21         Dog.__init__(self)
22         Shape.__init__(self)
23 hsk= Husky()                       #创建 Husky 的实例对象
24 hsk.eat()                           #调用 hsk 的 eat() 方法
25 print(dir(hsk))                     # 查看 hsk 的所有属性和方法

```

**输出结果:**

```

['_Dog_color', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_name', '_ears', '_eat',
'height', 'leg', 'run']
Dog is eating .....

```

**总结如下:**

假设子类的定义格式为: class 子类(父类 1, 父类 2, ……):

- (1) 以双下划线开头的实例属性是私有属性, 不能被继承, 如案例中的 `__color` 属性;
- (2) 如果一个子类继承多个父类, 子类的实例对象搜索方法和属性的顺序是: 子类本身、父类 1、父类 2, ……、父类 1 的父类、父类 2 的父类……, 直到找到为止, 也即广度优先搜索;
- (3) 如果子类中没有重写 `__init__()` 方法, 则子类会默认调用父类 1 的 `__init__()` 方法;
- (4) 调用父类 `__init__()` 方法有两种格式: ① `super().__init__()`, 这种形式只调用父类 1 的 `__init__()` 方法。② 父类名. `__init__(self)`;
- (5) 子类与父类有相同的方法, 但功能不同时, 重写父类同名方法即可;
- (6) 使用继承的好处之一是可以代码复用、提高效率。

**知识拓展:** 如果要判断对象是什么数据类型, 可以用 `type` 和 `isinstance` 函数。

## 6.4.2 多态

多态是指同一个对象在不同的情况下, 有不同的状态。由于 Python 的变量不需声明类型, 所以从严格意义上说 Python 并不支持多态, 但是可以模拟多态。

## 本章小结

通过本章的学习, 我们了解了面向对象编程技术中类的概念和使用方法。详细介绍了封装、继承的相关内容, 但是, 这些只是面向对象编程的入门知识, 这些知识的具体运用体现在第八章的“模拟牧场救援游戏”中。当然, 并不是所有程序都需要面向对象, 如果面向过程编程能够轻松实现的, 就不必使用类。

## 练习题

### 一、填空

1. 面向对象程序设计的三大特性是\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_。
2. 如果不允许外部访问类的内部数据, 可以给内部属性添加\_\_\_\_\_个下划线。
3. Python3 中默认继承\_\_\_\_\_类, 它是所有类的基类(父类)。
4. 在定义类时, 实例方法的第一个参数习惯性写成\_\_\_\_\_, 而类方法的第一个参数习惯性写成\_\_\_\_\_。
5. 属性分为类属性和实例属性, 实例对象\_\_\_\_\_访问类属性。

### 二、程序设计

1. 请你定义一个学生 (Student) 类, 给定姓名 (name)、年龄 (age) 等私有属性 (可通过 get 和 set 方法进行访问), 并设定至少一个方法 (如 playBasketball 等), 并实例化一个学生实例。
2. 猜数字游戏: 一个类 ClsA 有一个成员变量 num, 设定一个初值 (比如 64)。定义一个类 ClsB, 对 ClsA 的成员变量 num 进行猜。如果输入的数字大了则提示大了, 小了则提示小了, 等于则提示 “猜测成功”。
3. 请定义一个汽车工具 (Vehicle) 的类, 其中属性有速度 (speed)、体积 (volume)、车牌编号 (number)、颜色 (color) 等。方法有移动 (move()), 设置速度 (setSpeed(int speed)), 体积 (setSize(int size)) 加速 (speedUp()), 减速 (speedDown()) 等。  
实例化一个 Vehicle, 并使用它的加速、减速的方法。
4. 在第 3 题基础上, 定义一个新类 Car 继承 Vehicle, 重写加速 (speedUp()) 和 (speedDown()) 功能, 使加速和减速为 Vehicle 的 2 倍, 并重新实例化一个 Car 实例。根据物理学知识, 在初始速度下加速 5 秒后匀速运行 1 分钟, 能够运行多少米。



# 第七章 文件及目录操作

在前面，我们编写的程序运行结束后数据就会丢失，有时我们希望将程序的运行结果保存到文件中，也希望处理一些磁盘中已经存在的文件。为满足这种需求，就需要掌握文件及目录的相关操作，本章将介绍 Python 中如何进行文件及目录的操作。

## 7.1 文件操作

对文件的主要操作有：创建文件、打开文件、读取文件内容、向文件中写入内容、关闭文件等。

### 7.1.1 创建和打开文件

Python 提供了内建函数 `open()`，可用于创建和打开文件。基本语法格式如下：

```
file = open(filename[, mode][, buffering][, encoding])
```

`filename`: 文件名称。`mode`: 打开模式，可选参数。`buffering`: 对文件读写的缓存模式，可选参数，值为 0、1 或大于 1 的整数，0 表示不缓存，1 表示缓存，其它值表示缓冲区的大小，默认为缓存模式，`encoding`: 编码方式，`file`: 文件对象。

例子：

```
>>>file = open(r'd:\123.txt') #字符串前面的 r 是使字符串中的\不转义
```

以只读模式打开 `d:\123.txt` 文件并创建文件对象，用变量 `file` 指向该文件对象。对文件的操作，就使用文件对象提供的方法即可。下面介绍 `mode` 的取值。

**mode 参数的可能值**

值	意义
r	以只读方式打开文件，文件的指针将会放在文件的开头。这是默认模式
r+	打开一个文件用于读写。文件指针将会放在文件的开头
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头，一般用于非文本文件，如声音文件。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
w	以只写方式打开文件，文件原有内容会被删除。
w+	打开一个文件用于读写，文件原有内容会被删除。
wb	以二进制格式打开一个文件只用于写入。
wb+	以二进制格式打开一个文件用于读写。
a	打开一个文件用于追加。
a+	打开一个文件用于读写
ab	以二进制格式打开一个文件用于追加。
ab+	以二进制格式打开一个文件用于追加

规律：模式是由 `r`、`w`、`a` 后面跟 `+`、`b`、`b+` 组成的。模式中凡是包含 `r` 的文件必须存在；模式中凡是包含 `w` 的如果文件存在则覆盖，不存在就创建新文件；模式中凡是包含 `a` 的如果文件存在则追加，不存在就创建新文件；模式中凡是包含 `b` 的表示以二进制格式打开，不包含 `b` 的是以文本文件方式打开，模式中凡是包含 `+` 的表示用于读写。

拓展阅读：文本文件与二进制文件的区别：它们的区别就是编码方式不同，文本文件是基于字符编码，二进制文件是基于值编码。

## 7.1.2 关闭文件

当一个文件被打开后，对文件操作的结果将会放到文件缓冲区中，比如增加了新的内容，如果操作完成后不关闭，这些增加的内容就不会被写入文件中，从而造成不必要的破坏，因此要及时关闭文件。关闭文件的语法为：`file.close()`，`file` 表示文件对象。

另一方面，如果在打开文件时或对文件的操作过程中遇到了错误，则不能使用 `file.close()` 来关闭文件。为了避免这类问题的发生，Python 提供了 `with` 语句来保证不论异常是否发生，`with` 语句执行完毕后文件都能关闭。`with` 语句的基本语法格式如下：

`with` 表达式 as 对象：

语句块

例子：

```
with open(r'd:\123.txt', 'w') as file
```

```
    file.write('人生苦短，我学 Python')
```

无需使用 `file.close()` 也可以关闭文件。建议使用这种方式打开文件进行操作。

## 7.1.3 读取文件

当文件被以读方式打开后，就可以对文件进行读操作了，可以读取指定长度的字符，读取一行或所有行。

文件对象提供的常用读写方法

方法名称	语法	描述
<code>readline</code>	<code>file.readline([size])</code>	操作文本文件时 <code>size</code> 为字符，操作二进制文件时 <code>size</code> 为字节，读取 <code>size</code> 个数据，省略 <code>size</code> 或 <code>size</code> 大于一行的数据时，读取一行，返回的是一个字符串对象
<code>read</code>	<code>file.read([size])</code>	操作文本文件时 <code>size</code> 为字符，操作二进制文件时 <code>size</code> 为字节，读取 <code>size</code> 个数据，省略 <code>size</code> 时，读取全部数据。返回的是一个字符串对象
<code>readlines</code>	<code>file.readlines([size])</code>	操作文本文件时 <code>size</code> 为字符，操作二进制文件时 <code>size</code> 为字节，读取大约 <code>size</code> 个数据，省略 <code>size</code> 时，读取全部数据，以字符串列表返回
<code>write</code>	<code>file.write(str)</code>	向文件中写入指定字符串
<code>writelines</code>	<code>file.writelines(seq)</code>	将字符串序列迭代后写入文件
<code>tell</code>	<code>file.tell()</code>	以字节为单位，返回文件的当前位置，即文件指针当前位置
<code>seek</code>	<code>file.seek(offset[, whence])</code>	文件指针移动 <code>offset</code> 字节。 <code>whence</code> 为 0 时，从文件头开始，为 1 时从当前位置开始，为 2 时从文件末尾开始。对于文本文件， <code>whence</code> 只能为 0。
<code>close</code>	<code>file.close()</code>	关闭文件
<code>flush</code>	<code>file.flush()</code>	将缓冲区中的数据立刻写入文件，同时清空缓冲区

truncate	file.truncate([size])	从文件开头开始截断文件。
next	Python3 不支持 file.next()	Python3 用 next(file) 获取下一项

例子：构造内容类似下面二行数据的文本文件（文件名 test.txt，与当前 py 文件放在同一目录下）。读入文件内容，并计算所有数值之和。

```
1,2,3,4,5,6,7,8,9
1,8,7,3,4,2,1,2,3,7,8
```

程序代码如下：

```
01 # *_ coding:utf-8 *_
02 sum = 0
03 with open(r'test.txt','r') as file:
04     while True:
05         line = file.readline()
06         if line == '':
07             break
08         for i in line.split(','):
09             sum += int(i)
10 print(sum)
```

通过这个例子，我们学会了一种构造测试数据的方法，即将测试数据放在外部文件中，免去了每次都需要逐个输入数据的麻烦。如果需要构造成千上万个测试数据，你可以用随机函数生成数据。

如果你尝试读取一个二进制文件，你会看到类似于“\x00\x00\x00\x00U^TALB\x00”的信息，没有任何意义。解析二进制文件时必须知道文件的数据结构，了解每个字节代表的意义，才能正确解析，通常二进制文件都是由对应的工具软件读取。

### 7.1.4 写入文件

向文件写入数据，使用文件对象的 write() 方法。语法格式如下：

```
file.write(str)
```

str: 要写入的字符串，file: 打开的文件对象

例 1: 从键盘输入一行数据，然后保存到文件中，假设文件名为 save\_test.txt.

程序代码如下：

```
01 # *_ coding:utf-8 *_
02 with open(r'd:\save_test.txt','w') as file:
03     string = input("请输入你要保存的数据：")
04     file.write(string)
05     file.flush() #将缓存区数据写入文件
06 print("数据已保存！")
```

说明：使用文件对象的 writelines() 方法，可以将字符串序列迭代后写入文件。如果希望以追加方式写入字符串要以 a 模式打开文件。

如果需要保存敏感信息（如用户名、登录密码），最好不要使用文本文件保存。

## 7.1.5 动手实践

**字符串替换：**将文本文件中的原字符串替换为目标字符串。

输入格式：文件名 原字符串 目标字符串

算法分析：问题可以划分为三个步骤：获取用户输入、以读方式打开文件，读取所有内容到内存、以写方式打开文件，替换后写入文件。

程序实现代码如下：

```
01 # *_ coding:utf-8 *_
02 opera_word = input("请输入文件名、原字符串、目标字符串：").strip().split()
03 filename = opera_word[0]
04 source_word = opera_word[1]
05 target_word = opera_word[2]
06 with open(filename,'r') as file:
07     old_lines = file.readlines()
08 with open(filename,'w') as file:
09     for line in old_lines:
10         new_line = line.replace(source_word,target_word)
11         file.write(new_line)
```

完成过程的思路较简单，但是，需要两次打开文件才能完成，操作耗时。下面给出只需打开一次的解决方法：

```
01 # *_ coding:utf-8 *_
02 opera_word = input("请输入文件名、原字符串、目标字符串：").strip().split()
03 filename = opera_word[0]
04 source_word = opera_word[1]
05 target_word = opera_word[2]
06 with open(filename,'r+') as file:
07     old_lines = file.readlines()
08     file.seek(0)
09     file.truncate()
10     for line in old_lines:
11         new_line = line.replace(source_word,target_word)
12         file.write(new_line)
```

修改之后的代码主要是 `file.seek(0)`、`file.truncate()`，功能是删除所有内容。这两种方式的共同点是先读取所有内容，然后从文件开始处写入内容。更好的方法是使用 `fileinput` 模块实现，请同学们自行查找相关资源学习。

## 7.2 目录操作

目录也叫文件夹，通过目录可以对文件进行分类存放，方便查找文件。Python 没有提供内置函数用于对目录进行操作，通常是使用内置的 `os` 模块实现。

对目录的主要操作有：创建目录、删除目录、遍历目录。

### 7.2.1 路径

计算机中把标识一个文件或目录的位置的字符串叫做路径。路径类似于我们的家庭住址。如 Windows 平台下的 `c:\system32\driver`，linux 平台下的 `/usr/local/src`。路径包括相对路径和绝对路径两种。**说明：Windows 平台下路径也可以使用 `c:/system32/driver`，为了统一，后面的介绍中都使用 “/”，“/” 理解为 “里面”。**

#### 1. 相对路径

首先说明什么是当前工作目录，当前工作目录是指当前文件所在的目录。当前工作目录用点 “.” 表示。相对路径是指从当前工作目录开始的路径，如在当前工作目录有一个名为 “test.py” 文件，则其相对路径表示为：

“./test.py”。如在当前工作目录下有一个目录“script”，在目录“script”下有一个名为“test.py”文件，则其相对路径表示为：“./script/test.py”。

## 2. 绝对路径

绝对路径是指，从根开始的路径。Windows 平台下的根是盘符，linux 平台下的根是“/”。

## 3. 路径的获取与拼接

os 是 Python 的内置模块，是一个用于访问操作系统功能的模块。在使用前需要使用 import os 语句导入。下面列出 os 模块提供的常用方法。

方法名称	语法	描述
name	os.name	判断当前正在使用的平台，Windows 返回'nt'；Linux 返回'posix'，Mac OS 返回'Unix'
getcwd	os.getcwd()	得到当前工作的目录
listdir	os.listdir(path)	返回指定目录下所有的文件名和目录名
remove	os.remove(filename)	删除指定文件
rename	os.rename(src, dst)	重命名目录或文件
rmdir	os.rmdir(dirname)	删除指定目录
mkdir	os.mkdir(dirname)	创建目录
makedirs	os.makedirs(path1/path2/...)	创建多级目录
removedirs	os.removedirs(path1/path2/...)	删除多级目录
chdir	os.chdir(path)	把 path 设为当前工作目录
walk	os.walk(top)	遍历目录树
startfile	os.startfile(path[, operation])	使用关联的应用程序打开 path 文件
abspath	os.path.abspath(path)	获取绝对路径
exists	os.path.exists(path)	判断文件或目录是否存在
join	os.path.join(path, name)	拼接目录与目录或文件名
splitext	os.path.splitext(filename)	分离文件名与扩展名
basename	os.path.basename(path)	从路径中提取文件名
dirname	os.path.dirname(path)	从路径中提取不包括文件名的路径
isdir	os.path.isdir(path)	判断是否为真实存在的路径
isfile	os.path.isfile(path)	判断是否为真实存在的文件

实践：请在 IDLE 的交互模式下运行如下代码：

```
>>>import os
>>>os.name
>>>os.getcwd()
>>>os.listdir()
>>>os.linesep
```

```
>>>os. sep
```

```
>>>os. path
```

如果需要把多个路径拼接起来组成一个新路径，使用 `os.path.join()` 方式实现，基本语法格式如下：

```
os.path.join(path1, path2, ...)
```

`path1, path2, ...` 表示要拼接的路径，如果这些路径中存在绝对路径，则以最后一个绝对路径开始拼接，如果都是相对路径则拼接得到的也是一个相对路径。

例子：

```
>>>os.path.join('/aa/bb/', 'd:') #d:/
```

```
>>>os.path.join('./aa/bb/cc', './dd/ee') # ./aa/bb/cc/./dd/ee
```

说明：由于拼接的路径中可能有绝对路径，所以不要使用字符串拼接。

## 7.2.2 创建目录

`os` 模块提供了两个创建目录的方法 `mkdir()` 和 `makedirs()`，分别用于创建一级目录和多级目录。

### 1. 创建一级目录

创建一级目录是指一次只能创建一级目录，基本语法格式如下：

```
os.mkdir(path, mode=0o777)
```

`path`：目录名称，可以是相当路径也可以绝对路径。

`mode`：访问权限，在非 `unix` 系统上将被忽略。

例子：

```
os.mkdir('./ipeg') #将在当前目录下创建名称为 jpeg 的目录。
```

说明：（1）如果指定的路径包含多级目录，只创建最后一级目录，如果上级目录不存在，则会引发 `FileNotFoundError` 异常。（2）如果要创建的目录（或同名的文件）已经存在，则会引发 `FileExistsError` 异常。

### 2. 创建多级目录

创建多级目录是指一次可以创建多级目录，基本语法格式如下：

```
os.makedirs(path, mode=0o777)
```

`path`：目录名称，可以是相对路径也可以绝对路径。

`mode`：访问权限，在非 `unix` 系统上将被忽略。

例子：

```
os.makedirs('c:/wwwroot/root/doc') #将创建 c:/wwwroot/root/doc
```

如果要创建的目录（或同名的文件）已经存在，则会引发 `FileExistsError` 异常。为了屏蔽该异常出现，在创建目录前先判断目录是否已经存在。实现代码如下：

```
import os
path = 'c:/path'
if not os.path.exists(path):
    os.makedirs(path)
```

## 7.2.3 删除目录

使用 `os` 模块的 `rmdir()` 方法删除空目录，基本语法格式如下：

```
os.rmdir(oath)
```

`path`：要删除的目录，可以是相对路径也可以是绝对路径。

例子：

```
os.rmdir('c:/wwwroot/root/doc') #将删除 doc 目录
```

如果要删除的目录不存在会引发 `FileNotFoundError` 异常，如果不是空目录会引发 `OSError` 异常。可以使用内

置模块 `shutil` 的 `rmtree()` 方法删除非空目录。

## 7.2.4 删除文件

使用 `os` 模块的 `remove()` 方法删除文件，基本语法格式如下：

```
os.remove(path)
```

`path`: 要删除的文件路径，可以是相对路径也可以是绝对路径。

例子：

```
os.remove('./test.py')    #将删除当前目录下的 test.py 文件
```

如果要删除的文件不存在会引发 `FileNotFoundError` 异常。为了屏蔽该异常出现，在删除文件前先判断文件是否存在。在同一路径下，可能会存在文件名与目录名相同的情况，为了区别是目录还是文件，使用 `os` 模块的 `isdir()` 和 `isfile()` 判断。具体参见后面的动手实践。

## 7.2.5 重命名文件和目录

使用 `os` 模块的 `rename()` 方法重命名文件或目录，基本语法格式如下：

```
os.rename(src, dst)
```

`src`: 需要重命名的文件或目录，`dst`: 重命名后的文件或目录。

如果要重命名的文件或目录不存在会引发 `FileNotFoundError` 异常，如果目标文件名或目录名与现有的文件名或目录名同名会引发 `FileExistsError` 异常，为了屏蔽异常出现，在操作前先判断原文件或目录及目标文件或目录是否存在。

例子：

```
01 import os
02 srcpath = 'c:/wwwroot/a'
03 dstpath = 'c:/wwwroot/b'
04 if os.path.exists(srcpath) and not os.path.exists(dstpath):
05     os.rename(srcpath, dstpath)
```

## 7.2.6 遍历目录

使用 `os` 模块的 `walk()` 方法可以遍历目录，基本语法格式如下：

```
os.walk(top[, topdown])
```

`top`: 要遍历的根目录，可以是相对路径也可以是绝对路径。

`topdown`: 可选参数，用于指定遍历顺序，`True` 表示先遍历父目录，再遍历子目录，`False` 表示先遍历子目录，再遍历父目录。

返回值：一个包含 3 个元素 (`path`, `dirs`, `files`) 的元组生成器对象。`path`: 当前路径；`dirs`: 当前路径下子目录列表；`files`: 当前路径下的文件列表。

★生成器是指仅保存生成算法而不实际生成元素的对象。

例子：

```

01 # -*- coding:utf-8 -*-
02 import os
03 path = "c:\java" #请根据实际情况修改
04 for root, dirs, files in os.walk(path):
05     for dir in dirs:
06         print("目录: {}".format(os.path.join(root, dir)))
07     for file in files:
08         print("文件: {}".format(os.path.join(root, file)))

```

### 可能的输出:

```

目录: c:\java\bin
目录: c:\java\lib
文件: c:\java\COPYRIGHT
文件: c:\java\LICENSE
文件: c:\java\README.txt
文件: c:\java\release
文件: c:\java\THIRDPARTYLICENSEREADME-JAVAFX.txt
文件: c:\java\THIRDPARTYLICENSEREADME.txt

```

## 7.2.7 获取文件基本信息

计算机上的每个文件都包含了一些基本信息，如创建时间、最后修改时间、最后访问时间、文件大小等信息，在 Python 中，通过 os 模块的 stat() 方法可以获取文件的这些基本信息，基本语法格式如下：

```
os.stat(path)
```

path: 要获取基本信息的文件路径，可以是相对路径也可以是绝对路径。

返回值是一个 stat\_result 对象，使用“对象.属性”获取具体信息。

**stat\_result 对象的属性**

属性	说明	属性	说明
st_mode	保护模式	st_gid	组 ID
st_ino	索引号	st_size	文件大小，单位为字节
st_dev	设备名	st_atime	最后一次访问时间
st_nlink	硬链接数	st_mtime	最后一次修改时间
st_uid	用户 ID	st_ctime	文件的创建时间

例子：查看当前目录下 test.py 的基本信息，实现代码如下：

```

01 # -*- coding:utf-8 -*-
02 import time
03 import os
04 def formattime(utime):
05     return time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(utime))

```



```

06 file_info = os.stat('./test.py')
07 print('保护模式: ', file_info.st_mode)
08 print('索引号: ', file_info.st_ino)
09 print('设备名: ', file_info.st_dev)
10 print('硬链接数: ', file_info.st_nlink)
11 print('用户 ID: ', file_info.st_uid)
12 print('组 ID: ', file_info.st_gid)
13 print('文件大小: ', file_info.st_size)
14 print('最后一次访问时间: ', formattime(file_info.st_atime))
15 print('最后一次修改时间: ', formattime(file_info.st_mtime))
16 print('文件的创建时间: ', formattime(file_info.st_ctime))

```

#### 输出结果:

```

保护模式: 16895
索引号: 1970324837462416
设备名: 2226649119
硬链接数: 1
用户 ID: 0
组 ID: 0
文件大小: 1674
最后一次访问时间: 2018-09-23 09:43:08
最后一次修改时间: 2018-09-23 09:43:08
文件的创建时间: 2018-09-23 09:43:08
说明: formattime 函数是将 Unix 时间转换为我们熟悉的日期时间格式。

```

## 7.2.8 动手实践

**删除非空目录:** 仅使用 `os` 模块删除一个非空目录。假设需要删除的目录名为 `test`，其下可能有若干文件及目录，每个目录下可能还有若干文件及目录。

算法分析: 由于 `os.rmdir()` 只能删除空目录，所有在删除目录之前必须先删除里面的所有子目录和文件，子目录里面可能还有子目录和文件，……，适合使用递归实现。

程序代码如下:

```

01 # -*- coding:utf-8 -*-
02 import os
03 def deldir(path):
04     f_list = os.listdir(path)
05     if len(f_list) > 0:                                     #判断是否为空目录
06         for item in f_list:
07             if os.path.isfile(os.path.join(path, item)):

```

```

08         os.remove(os.path.join(path, item))           #是文件，直接删除
09         elif os.path.isdir(os.path.join(path, item)):
10             deldir(os.path.join(path, item))           #是目录，递归删除
11     os.rmdir(path)                                     #删除目录
12 path = "./test"                                       #请根据实际修改
13 deldir(path)

```

## 7.3 shutil 模块简介

shutil 是 Python 提供的一个内置模块，主要用于文件的移动、复制、打包、压缩、解压等。本节我们仅列出 shutil 模块的常用方法。

shutil 模块的常用方法

方法名称	语法	简述
copyfileobj	shutil.copyfile(src, dst [, length])	复制指定长度的文件内容
copyfile	shutil.copyfile(src, dst)	复制文件内容
copy	shutil.copy(src, dst)	复制文件的内容及权限
copy2	shutil.copy2(src, dst)	复制文件的内容以及文件的所有状态信息
copymode	shutil.copymode (src, dst)	仅复制权限，不更改文件内容，组和用户
copytree	shutil.copytree(src, dst)	递归复制文件内容及状态信息
move	shutil.move(src, dst)	将 src 文件或文件夹移动到 des 中
rmtree	shutil.rmtree(path)	递归删除目录
make_archive	shutil.make_archive(name, format, root_dir)	压缩打包
unpack_archive	shutil.unpack_archive(name)	解压文件
disk_usage	shutil.disk_usage('.')	获取磁盘使用空间

在使用 shutil 模块前，需要使用 import shutil 导入。

## 本章小结

本章介绍了使用内置函数 open()、内置模块 os 对文件及目录的一些基本操作。开发任何一个系统都必然要与文件和目录打交道，熟练掌握这些操作是设计程序的基本功。需要注意的是 os 模块还有很多方法我们并未列出。在实际开发中有时还需要引入其它模块才能更高效地完成工作，比如 shutil 模块。

在 IDLE 的交互模式下使用 help (模块名) 可以查看模块的所有信息。

## 练习题

**对文件和目录的操作有风险，请首先构造用于实验的文件和目录，再完成下面各题。**

1. 请把某个目录下的扩展名为 txt 的文件中追加一行‘被我找到了’，如果没有找到文件输出“没有找到文件”，否则输出“已追加文件 X 个”。

输出格式：没有找到文件

已追加文件 X 个

2. 统计某个目录下文件数和子目录数（含子目录）。

输出格式：目录数： 5

文件数： 2

3. 查找某个目录下是否存在某个文件名（含子目录）。如果存在输出路径，否则输出“未找到”。

输出格式：未找到

c:/www/ww/

4. 按时间顺序输出某个目录下的子目录名和文件名的部分信息。

输出格式：

文件名	修改日期	大小
f	2018-07-02	
1.txt	2018-07-01	234KB