内容简介

本书是面向编程零基础读者的 Python 入门教程，内容涵盖了 Python 的基础知识和初步应用。以比较轻快的风格，向零基础的学习者介绍了一门时下比较流行、并且用途比较广泛的编程语言。同时，其语法简洁而清晰，类库丰富而强大，非常适合于进行快速原型开发。另外，Python 可以运行在多种系统平台下，从而使得只需要编写一次代码，就可以多个系统平台下保持有同等的功能。

为了能够使广大读者既能够掌握 Python 语言的基础知识，又能够将 Python 语言用于某个特定的领域，本书将全面介绍和 Python 相关的这些内容。在学习完本书之后，相信读者能够很好地掌握 Python 语言，同时可以使用 Python 语言进行实际项目的开发。

本书以理论与实际相结合为原则，为每个知识点都设计了对应的示例，让 Python 的初学者能够既快速又深刻的理解这些知识点。同时在每章的最后设计了针对各章内容的作业题，能够让读者趁热打铁，以达到巩固所学知识的目的。

本书可配合尚学堂科技推出的《Python400 集大型视频课程》，作为参考教材。

前言

本书特色

学员需求：学习不枯燥、能够快速入门、实战操作、熟悉底层原理；企业要求：程序员既有实战技能可以快速上手，也内功扎实熟悉底层原理后劲十足。因此，针对以上需求我们北京尚学堂科技有限公司设计了本书。

本书的主要有以下几个特点：

1. 循序渐进，由浅入深

为了方便读者学习，本书首先让读者了解了 Python 的历史和特点，通过具体的例子逐渐把读者带入 Python 的世界，掌握 Python 语言的基本要点以及基础类库、常用库和工具的使用。

2. 技术全面，内容充实

本书在保证内容使用的前提下，详细介绍了 Python 语言的各个知识点。同时，本书所涉及的内容非常全面，无论从事什么行业的读者，都可以从本书中找到可应用 Python 与本行业的地方。

3. 代码完整，详解详尽

对于书中的每个知识点都有一段示例代码，并对代码的关键点进行了注释说明。每段代码的后面都有详细的分析，同时给出了代码运行后的结果。读者可以参考运行结果阅读源程序，可以加深对程序的理解。

本书如何学习

本书共分二十五章，为了方便大家的学习，我们对各章节做简要说明。

第一章：讲解了 Python 能做什么，Python 的特征和优势，Python 环境的搭建等内容。

第二章：讲解了 Python 的语法知识，深入讲解了 Python 的编码规则、变量和常量的声明及使用、数据类型、运算符和表达式。通过本章的学习，读者能掌握 Python 编码的一些规范以及一些基本概念。

第三章：讲解了 Python 中的控制语句、循环语句以及一些习惯用法，结合示例讲解了 Python 结构化编程的要点。

第四章：讲解了 Python 的内置数据结构（列表、元祖、字典、集合）。根据使用习惯分别介绍了这些内置数据结构的特点以及区别。

第五章：讲解了 Python 中函数的概念。重点介绍了 Python 的函数的定义、调用、传参、递归函数等内容。

第六章：讲解了面向对象程序设计，重点讲解如何实现面向对象的三大特性及设计模式。

第七章：讲解了 Python 中的模块、包的概念，重点讲解了模块的导入及使用。

第八章：讲解了 Python 对异常的处理、异常的捕获和抛出、自定义异常等内容。

第九章：讲解了 Python 对文件的基本操作，包括文件的创建、读写、删除、复制。重点讲解了 pickle 序列化、处理 JSON 格式的数据。

鸣谢

本书由北京尚学堂科技的教研部编制。北京尚学堂科技是一家从事人工智能、大数据、区块链、Java 技术、数据库、前端全栈、UI 设计等互联网技术培训的公司，其编写的教材和发布的教程影响了一代程序员。公司多次获得新浪、腾讯、网易、央视等年度最佳机构的荣誉。

本书在出版过程中，得到了清华大学出版社杨如林老师的大力支持，在此表示衷心感谢。另外，本书所有的编审、发行人员为本书的出版发行付出了辛勤的劳动，在此一并致以诚挚的谢意。

本书主编为高淇，主要参编人员有：高淇、张巧英等。我们以科学、严谨的态度，力求精益求精，但错误之处，在所难免，敬请广大读者批评指正，我们将不胜感激。

教研部出版组邮箱：book@sxt.cn　　高淇老师邮箱：gaoqi@sxt.cn

目录

案例目录

第一章 Python 入门

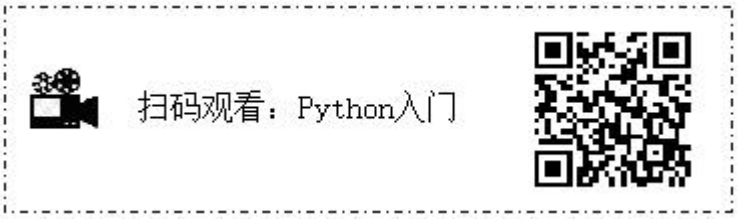
Python 是一种跨平台的，面向对象的程序设计语言。本章首先介绍一些 Python 的背景知识，包括 Python 简介、Python 的一些特点、Python 应用场景。其次一步步教读者 Python 运行环境的搭建。本章的主要目的是让读者对 Python 语言有一个整体的了解，然后再一点点深入学习 Python 语言，最后达到完全掌握 Python 语言的目的。

通过阅读本章，你可以：

- 了解 Python 语言的特点
- 了解 Python 语言应用场景
- 掌握如何搭建 Python 开发环境
- 掌握 PyCharm 的安装与配置
- 掌握 Python 程序的编写方法
- 了解图形化程序设计

1.1 Python 介绍

1.1.1 Python 简介



Python 是当今世界最流行的程序语言之一，它通俗易懂，可读性强，且拥有优秀的语法结构。Python 是一种解释型、面向对象的语言。由吉多·范罗苏姆（Guido van Rossum）（国内昵称：龟叔）于 1989 年发明，1991 年正式公布。官网：www.python.org，如图 1-1 所示。如果你是一个编程初学者，你会发现 Python 编程并没有那么枯燥，甚至可以体会到 Python 的优雅与简洁之美，希望本书将会是你 IT 生涯最佳的启蒙恩师。

Python 突出的简洁性、易读性和可扩展性，使得 Python 应用于科学研究的机构日益增多，这里也包括一些全球顶尖的大学也在采用 Python 教授程序设计课程。除此之外，Python 在数据科学、人工智能、云计算、图形处理与互联网应用等领域同样占尽了风头。

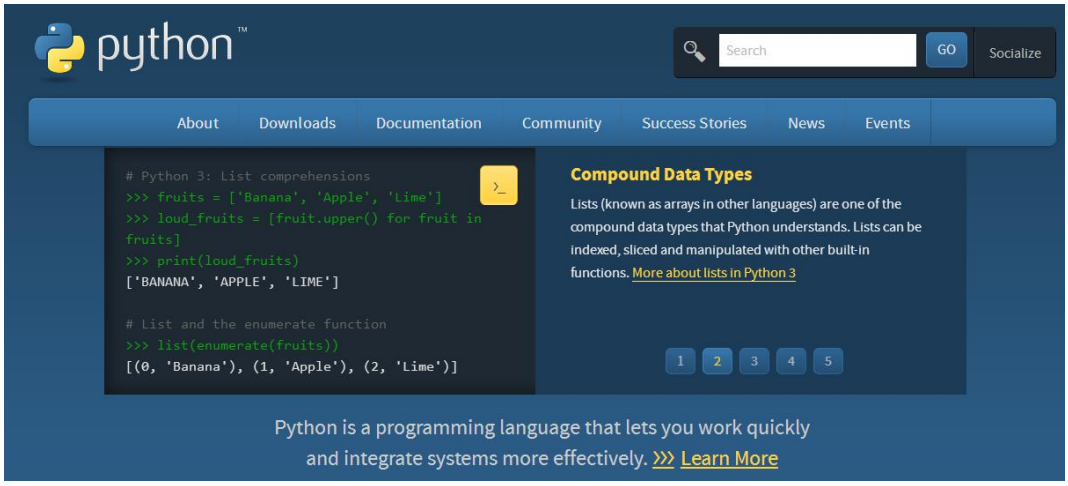


图 1-1 Python 官网

Python 单词是“大蟒蛇”的意思。但是龟叔不是喜欢蟒蛇才起这个名字，而是正在追剧：英国电视喜剧片《蒙提·派森的飞行马戏团》(Monty Python and the Flying Circus)。如图 1-2。

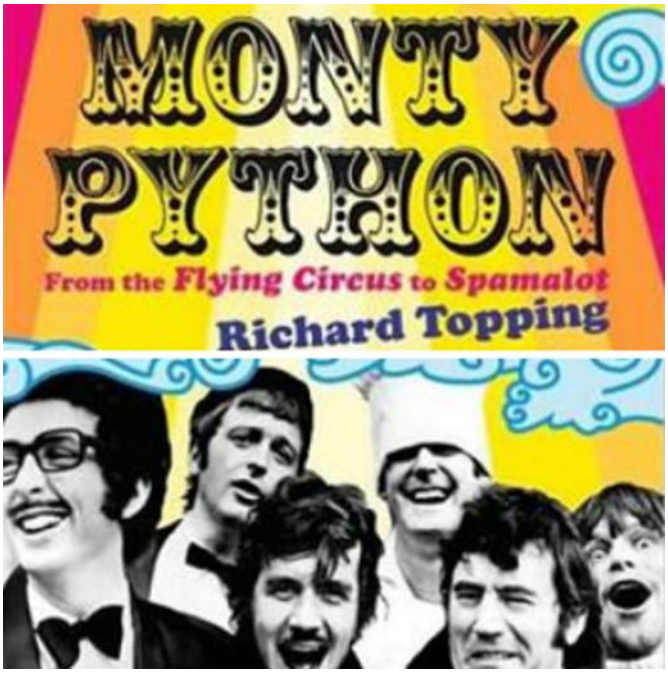


图 1-2 戏剧片剧照

使用 www.python.org 提供的 interactive shell 入门 Python，如图 1-3、1-4。

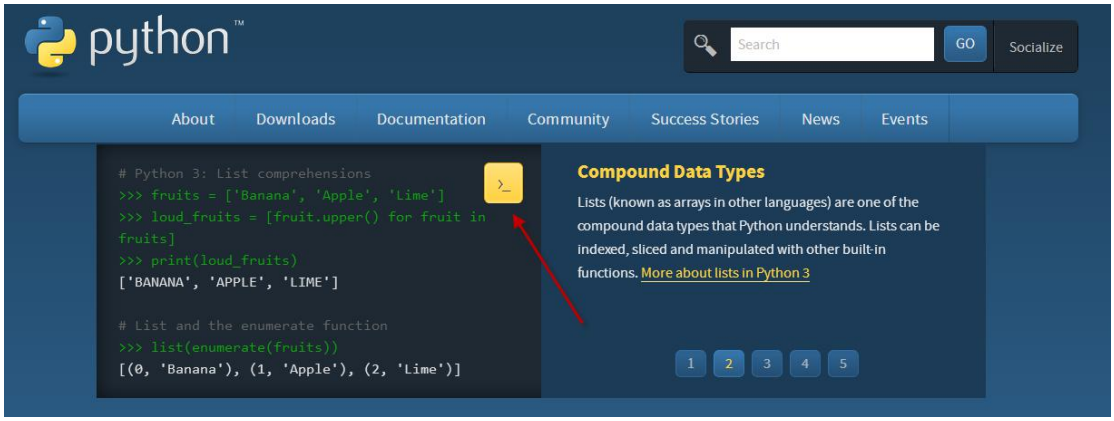


图 1-3 入门链接

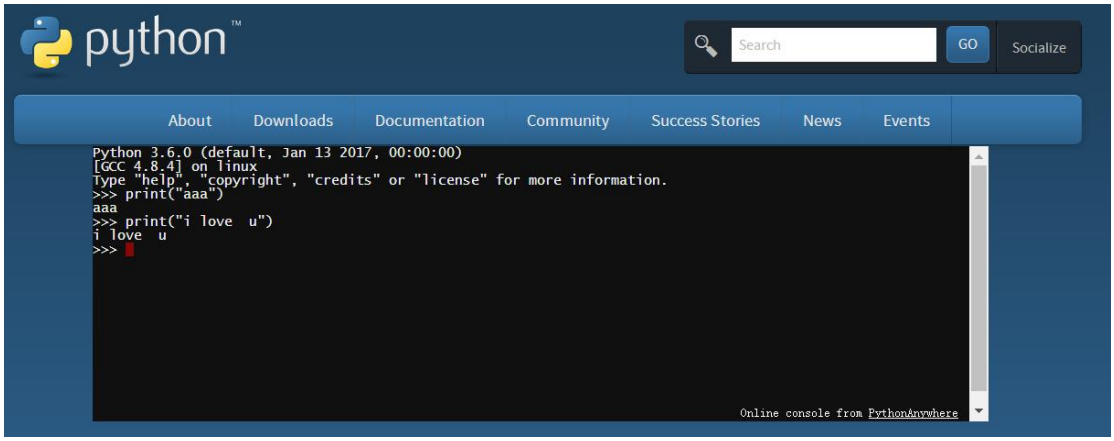


图 1-4 shell 窗口

1.1.2 Python 特点

- **可读性强**

  可读性远比听上去重要的多得多。一个程序会被反复的修改，可读性强意味着让你可以在更短时间内学习和记忆，直接提高生产率。

- **语法简洁**

  研究证明，程序员每天可编写的有效代码数是有限的。完成同样功能只用一半的代码，其实就是提高了一倍的生产率。

  Python 是由 C 语言开发，但是不再有 C 语言中指针等复杂数据类型，Python 的简洁性让开发难度和代码幅度大幅降低，开发任务大大简化。程序员再也不需要关注复杂的语法，而是关注任务本身。

- **入门级语言**

  只适合菜鸟？准确的说拉近了高手与初学者之间的距离。Python 简洁的语法结构，学习门槛低，编程极易上手，无论老鸟还是菜鸟都站在同一个起跑线上。

- **解释性与交互性**

  与典型的 Java 编译型语言相比，Python 属于解释型语言。一方面，Python 编写一条程序语句，即可解释执行返回一个结果。当程序出错时更容易跟踪与定位；另一方面：Python 这种交互式模式为人机互动提供了更广阔的可能空间。

- **优秀的模块化思维**

将代码组织为一个或若干模块，模块组织成为包、甚至库。试想当你编写程序的时候如果已经有针对科学计算、爬虫、数据分析、可视化、机器学习等模块或第三方库可以直接拿来使用，编程效率会极大的提高。

- **开源软件和丰富的第三方库**
  Python 是纯粹的开源语言， 源代码遵循 GPL 许可，这些特性使其更受大众欢迎，软件更容易移植到其他的平台，如 Mac、Linux 等，因此 Python 拥有丰富的第三方资源库是不足为奇的。
- **标准脚本语言**
  脚本程序是指只有需要被调用的时候才会被动态的解释执行。Python 允许混合使用 C、Java 与 Python 代码，通过增强扩展性来解决一些特殊的问题，例如 Python 程序中允许调用一段由 Java 编写的程序模块（库）。以上这些特点都充分体现了 Python 的可扩展性和作为脚本语言的动态灵活性。

1.1.3 Python 应用场景

1)Web 应用开发

Python 经常被用于 Web 开发。比如，通过 mod_wsgi 模块，Apache 可以运行用 Python 编写的 Web 程序。Python 定义了 WSGI 标准应用接口来协调 Http 服务器与基于 Python 的 Web 程序之间的通信。一些 Web 框架，如 Django，TurboGears，web2py，Zope 等，可以让程序员轻松地开发和管理复杂的 Web 程序。

2)操作系统管理、服务器运维的自动化脚本

在很多操作系统里，Python 是标准的系统组件。大多数 Linux 发行版以及 NetBSD、OpenBSD 和 MacOSX 都集成了 Python，可以在终端下直接运行 Python。有一些 Linux 发行版的安装器使用 Python 语言编写，比如 Ubuntu 的 Ubiquity 安装器，RedHatLinux 和 Fedora 的 Anaconda 安装器。GentooLinux 使用 Python 来编写它的 Portage 包管理系统。Python 标准库包含了多个调用操作系统功能的库。通过 pywin32 这个第三方软件包，Python 能够访问 Windows 的 COM 服务及其它 WindowsAPI。使用 IronPython，Python 程序能够直接调用.NetFramework。一般说来，Python 编写的系统管理脚本在可读性、性能、代码重用度、扩展性几方面都优于普通的 shell 脚本。

3)科学计算

NumPy，SciPy，Matplotlib 可以让 Python 程序员编写科学计算程序。

4)桌面软件

PyQt、PySide、wxPython、PyGTK 是 Python 快速开发桌面应用程序的利器。

5)网络编程

网络编程是 Python 学习的另一方向，网络编程在生活和开发中无处不在，哪里有通讯

就有网络，它可以称为是一切开发的“基石”。对于所有编程开发人员必须要知其然并知其所以然，所以网络部分将从协议、封包、解包等底层进行深入剖析。

6)游戏

很多游戏使用 C++编写图形显示等高性能模块，而使用 Python 或者 Lua 编写游戏的逻辑、服务器。相较于 Python，Lua 的功能更简单、体积更小；而 Python 则支持更多的特性和数据类型。

7)爬虫开发

在爬虫领域，Python 几乎是霸主地位，将网络一切数据作为资源，通过自动化程序进行有针对性的数据采集以及处理。从事该领域应学习爬虫策略、高性能异步 IO、分布式爬虫等，并针对 Scrapy 框架源码进行深入剖析，从而理解其原理并实现自定义爬虫框架。

8)云计算开发

Python 是从事云计算工作需要掌握的一门编程语言，目前很火的云计算框架 OpenStack 就是由 Python 开发的，如果想要深入学习并进行二次开发，就需要具备 Python 的技能。

9)人工智能

Python 在人工智能大范畴领域内的机器学习、神经网络、深度学习等方面都是主流的编程语言，得到广泛的支持和应用。

10)金融分析

金融分析包含金融知识和 Python 相关模块的学习，学习内容囊括 Numpy\Pandas\Scipy 数据分析模块等，以及常见金融分析策略如“双均线”、“周规则交易”、“羊驼策略”、“Dual Thrust 交易策略”等。

1.1.4 Python 版本和兼容问题解决方案

最初，Python 是 20 世纪 80 年代末 90 年代初，由荷兰国家数学和计算机科学研究所的 Guido van Rossum 设计出来的。Python 的发展少不了借鉴和吸收其它优秀语言的精华，包括 C、C++、Unix shell 等。

Python 有两大版本阵营，分别是 Python2.x 和 Python3.x。Python2 发布于 2000 年 10 月。最新版本是 2.7，已经停止更新，不会再有 2.8 以后了。预计 2020 年退出历史舞台。Python3 发布于 2008 年。Python3 有了较大的提升，不兼容 Python2。

表 1-1 Python 的主要版本

版本	发布日期	备注与说明
Python 2.0	2000/10/16	Python 语言框架基础形成
Python 2.4	2004/11/30	Web 框架 Django 诞生

版本	发布日期	备注与说明
Python 2.5	2006/09/19	
Python 2.6	2008/10/01	
Python 2.7	2010/07/03	2.x 系列将在 2020 年停止支持
Python 3.0	2008/12/03	Python3 不再考虑对 Python2.x 的向后兼容性
Python 3.1	2009/06/27	
Python 3.2	2011/02/20	
Python 3.3	2012/09/29	
Python 3.4	2014/03/16	
Python 3.5	2015/09/13	增加 async、await 等关键字，未来的主流
Python 3.6	2015/05/17	
Python 3.7	2017/09/19	
Python 3.8	2019/10/14	

注意：

- Python2.x 和 Python3.x 的语法规则有所不同，2.x 程序在 3.x 版本上无法运行。同时，3.x 解决了 2.x 存在的编码等问题。因 2.x 拥有大量的库和用户群，2010 年推出了 2.7 兼容版本，大量的 Python3 的特性被反向迁移到了 Python2.7，这也就是 Python2.7 为什么可以运行一些 Python3.x 库的原因。
- Python3 的很多新特性也被移植到了 Python2.7，作为过渡。如果程序可以在 2.7 运行，可以通过一个名为 2to3（Python 自带的一个脚本）的转换工具无缝迁移到 Python3。
- 2.建议大家学习从 Python3 开始，毕竟这才是未来。

1.2 Python 运行环境

不管用什么工具开发 Python 程序，都必须安装 Python 的运行环境。由于 Python 是跨平台的，所以在安装之前，先要确定在操作系统平台上安装，目前最常用的是 Windows、Linux 平台。由于目前使用 Windows 的人数最多，所以本书主要以 Windows 为主介绍 Python 运行环境的搭建与程序的开发。

1.2.1 Windows 平台下 Python 环境搭建

首先进入 Python 官网：www.python.org/downloads/ 下载首页，如图 1-5 所示。

扫码观看：Python环境搭建

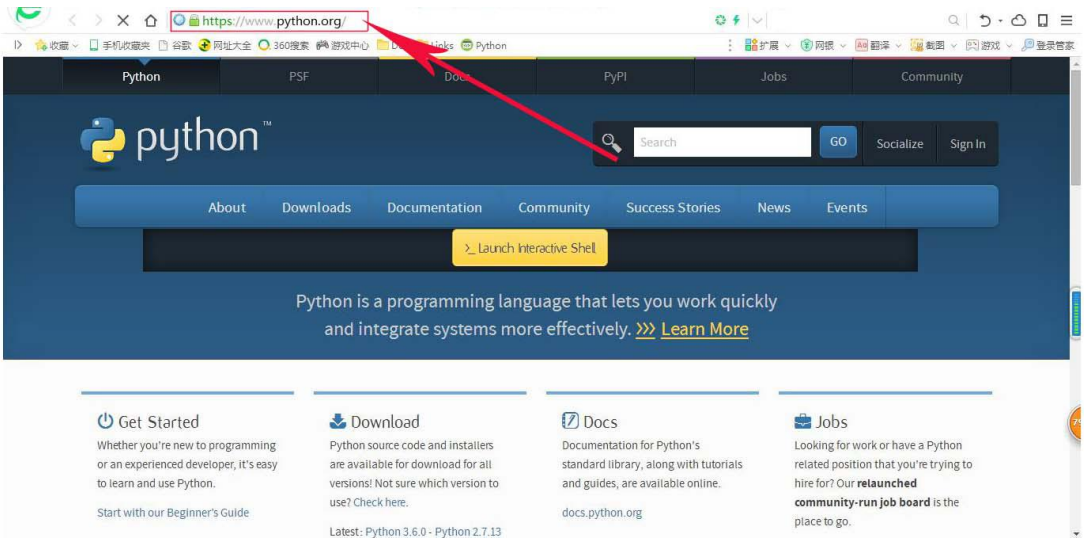


图 1-5 下载首页

进入下载页面，浏览器会根据不同的操作系统显示不同的 Python 安装包下载链接。如果读者使用的是 Windows 平台，会显示如图 1-6 所示的 Python 下载页面。

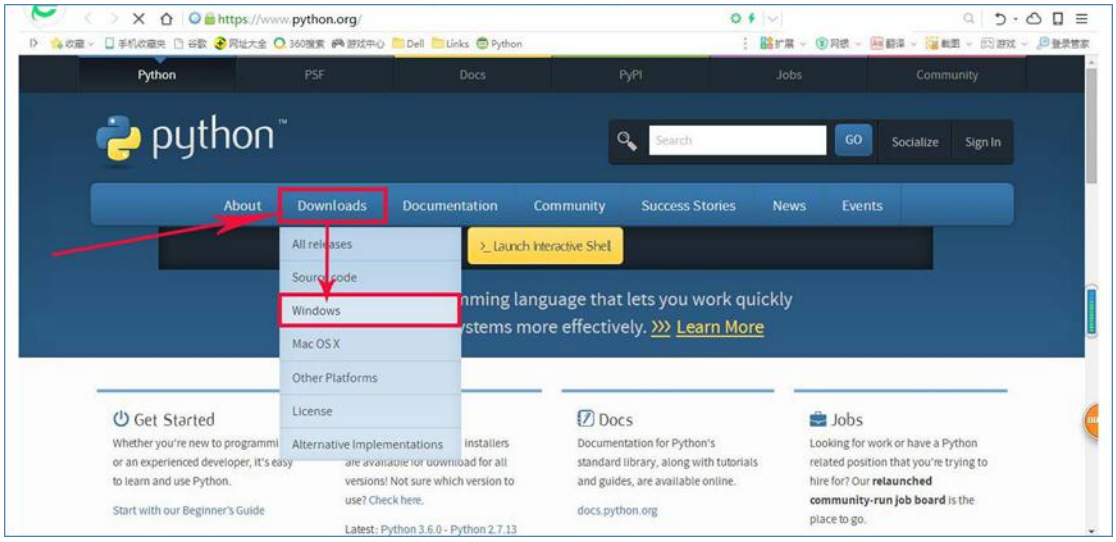


图 1-6 下载链接

进入 Windows 平台下载页面，选择对应操作系统位数的安装文件进行下载，如图 1-7 所示。

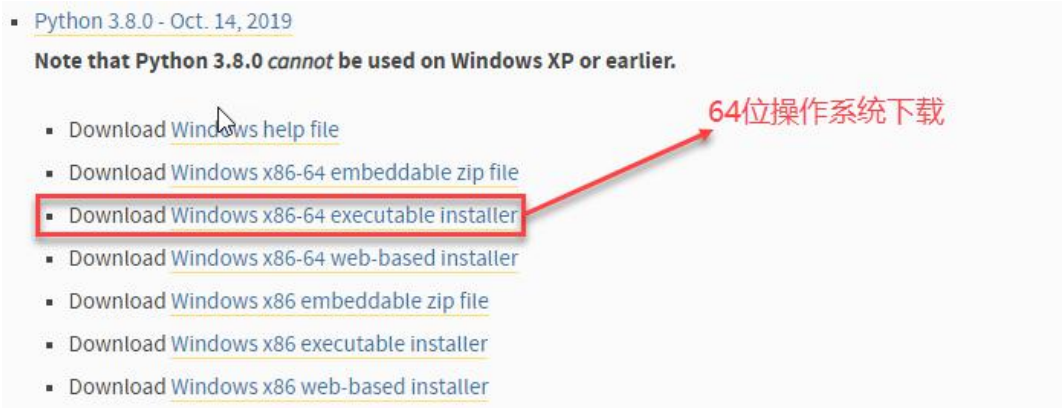


图 1-7 下载

现在主要介绍在 Windows 平台如何安装 Python 运行环境。首先运行下载的 exe 文件，会显示如图 1-8 所示的 Python 安装界面。建议选中界面下方的 Add Python 3.8 to PATH 复选框，这样安装程序就会自动将 Python 的路径加到 PATH 环境变量中。

在图 1-8 所示的界面中出现两个安装选项：Install Now 和 Customize installation，一般点击 Install Now 即可，单击该选项后，会开始安装 Python。图 1-9 所示是显示安装进度的界面。只需耐心安装完成即可。

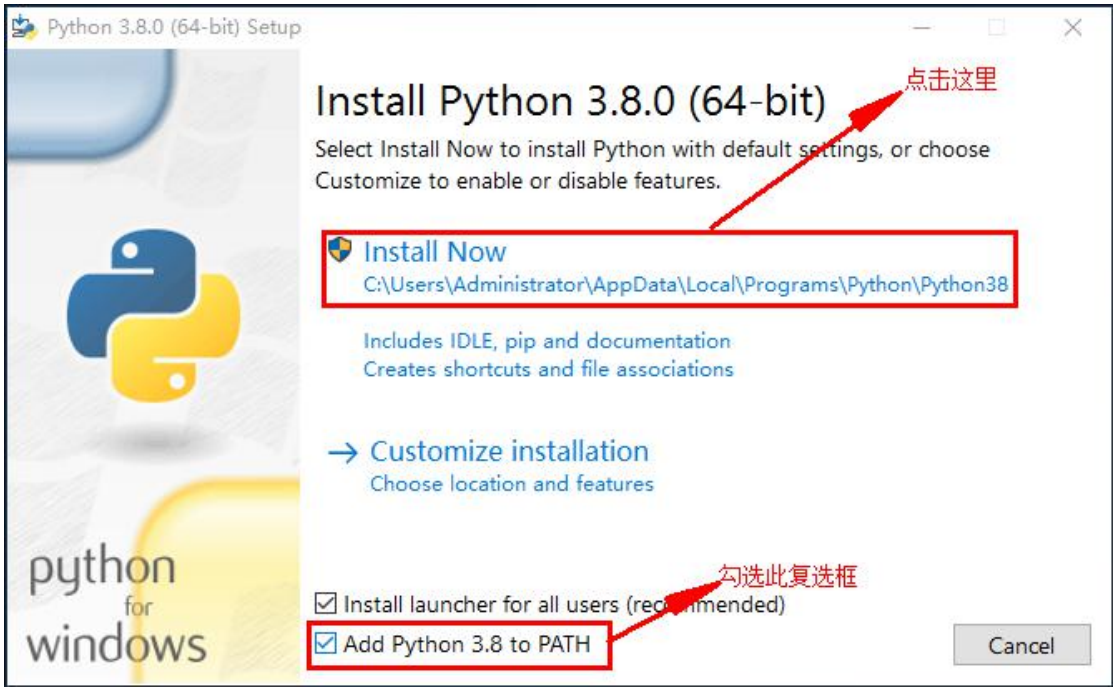


图 1-8 Windows 版 Python 安装程序初识界面

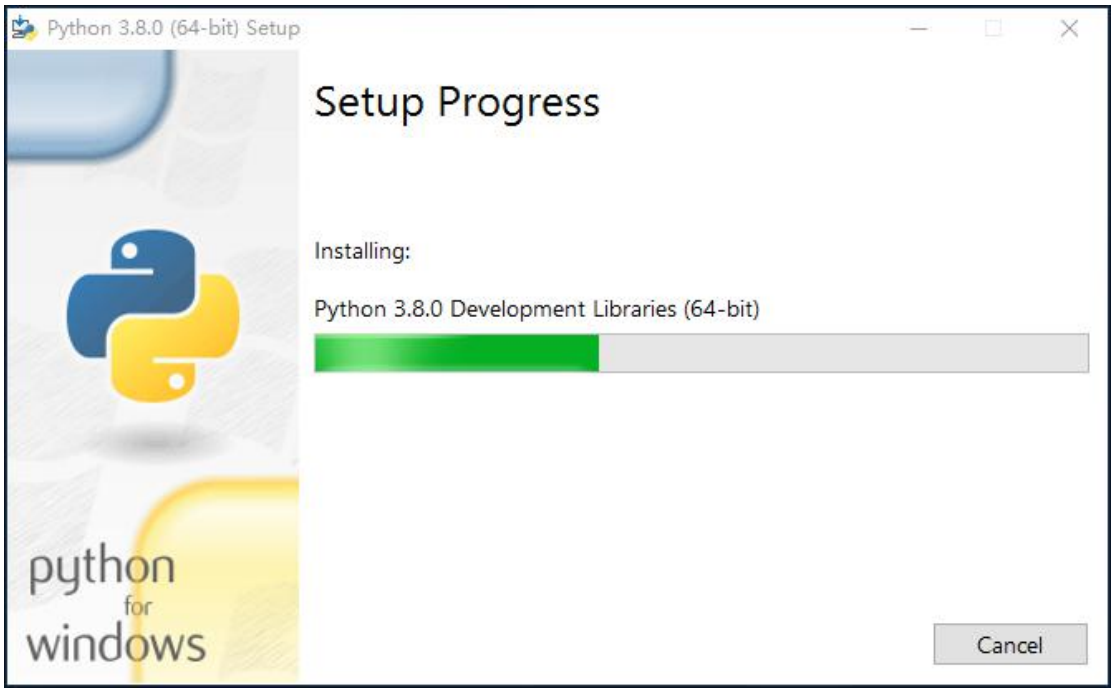


图 1-9 Python 安装进度

安装完后，会出现如图 1-10 所示的安装成功界面。

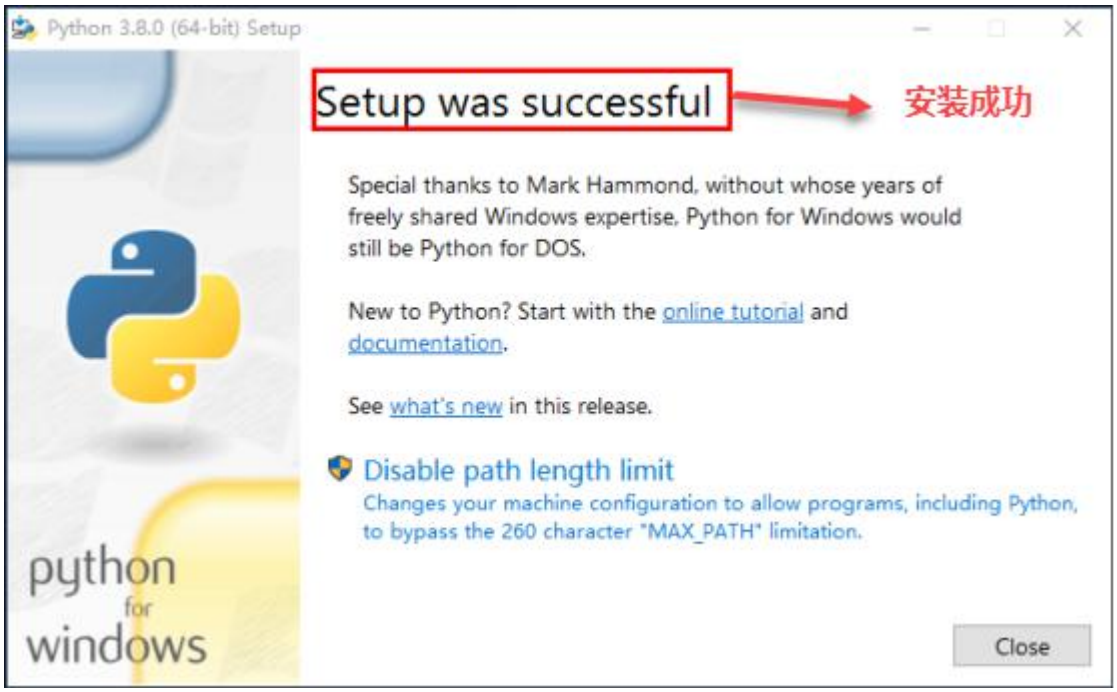


图 1-10 Python 安装成功界面

1.2.2 配置 PATH 环境变量

在安装完 Python 运行环境后，可以测试一下 Python 运行环境，如果在安装 Python 的过程中忘记勾选 Add Python 3.8 to PATH 复选框，那么默认情况下，Python 安装程序是不会将 Python 安装目录添加到 PATH 环境变量中的，这样一来，就无法在 Windows 命令行工具找那个的任何目录执行 python 命令，必须进入 Python 的安装目录才可以使用 python 命令。

为了更方便地执行 python 命令，建议将 python 安装目录添加到 PATH 环境变量中。在 Windows 平台配置 PATH 环境变量的步骤如下。

1)回到 Windows 的桌面，右击“计算机”，在弹出的快捷菜单中选择“属性”菜单项，会显示如图 1-11 所示的“系统”窗口。

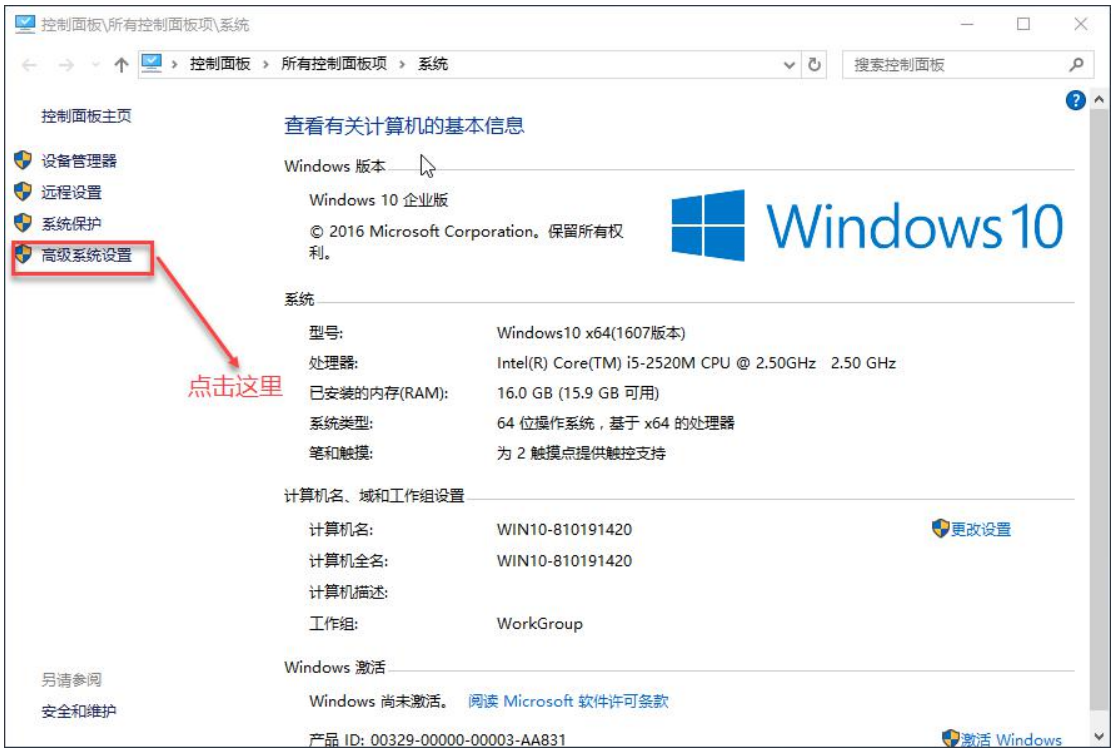


图 1-11 “系统”窗口

2)点击“系统”窗口左侧的“高级系统设置”，会弹出如图 1-11 所示的“系统属性的对话框。

图 1-12 “系统属性”窗口

3)点击“系统属性”对话框下方的“环境变量(N)…”按钮，会弹出如图 1-13 所示的“环境变量”对话框。

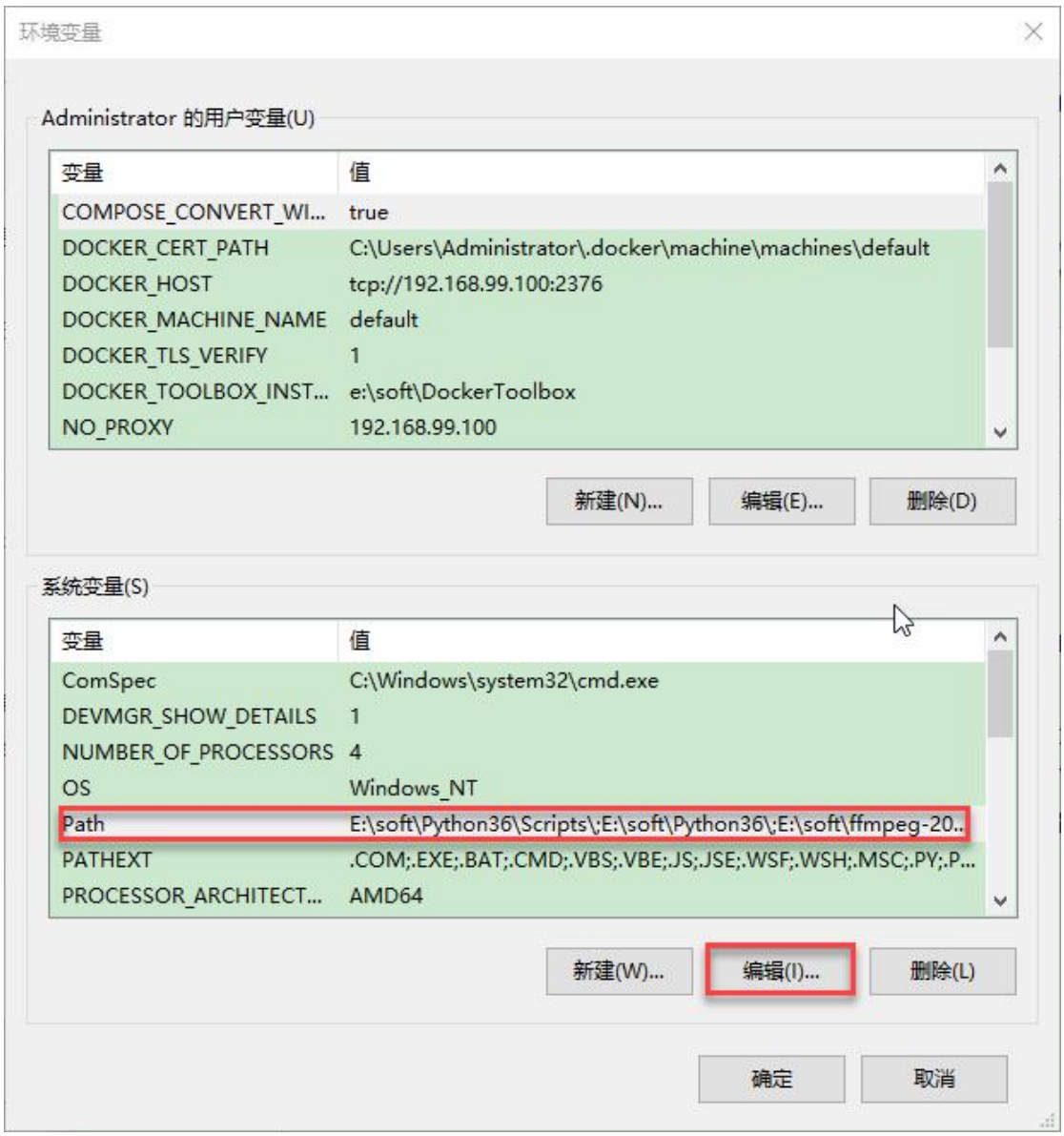


图 1-13 “环境变量”对话框

4)在“环境变量”对话框中有两个列表框，上面的列表框中是为 Windows 当前登录用户设置环境变量，在这里设置的环境变量只对当前登录用户有效。设置“系统变量”对所有登录用户都有效。本书设置“系统变量”中的 PATH 环境变量为例。在变量中找 PATH 环境变量，如果没有 PATH 环境变量，单击“新建(N)…”按钮新添加一个 PATH 环境变量，如果已经有一个 PATH 环境变量，双击 PATH，就会弹出如图 1-14 所示的“编辑”用户变量的对话框。

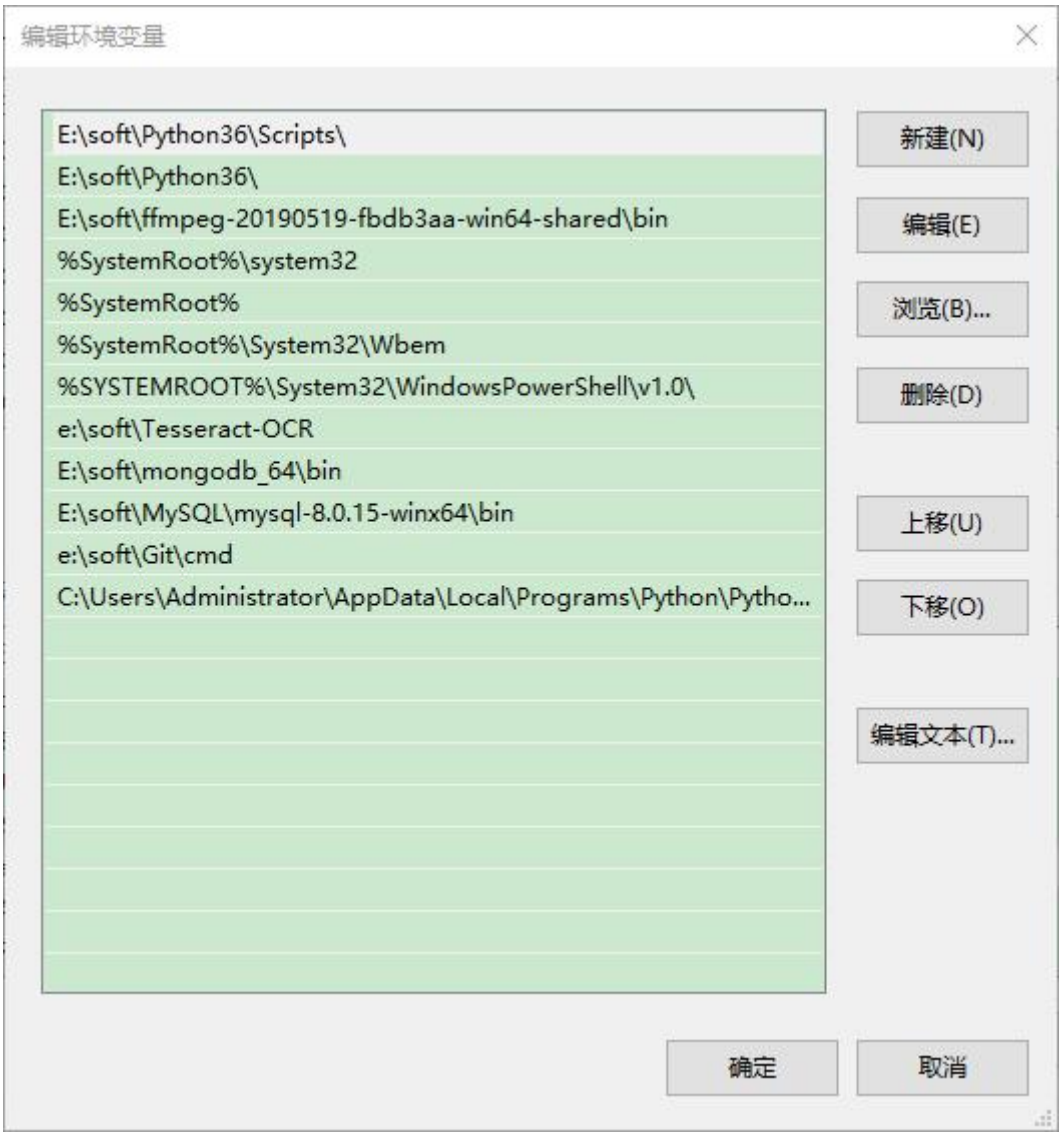


图 1-14 “编辑用户变量”对话框

5) 需要在“环境变量”中新增 Python 的安装目录(例如安装路径 C:\Users\Administrator\AppData\Local\Programs\Python\Python38），如图 1-15 所示

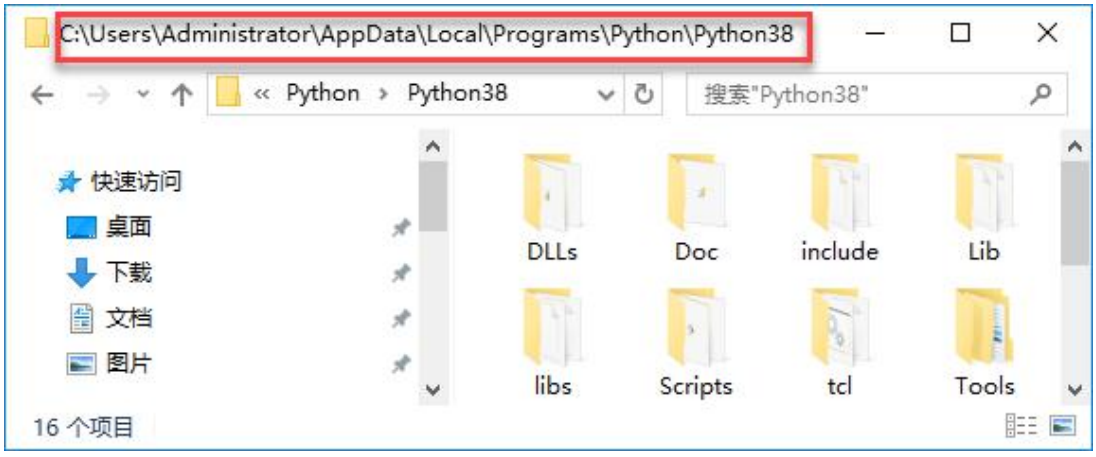


图 1-15 Python 安装目录

6)点击“新建(N)”按钮，文本框添加 Python 的安装目录如图 1-16 所示，添加完后点击确定。

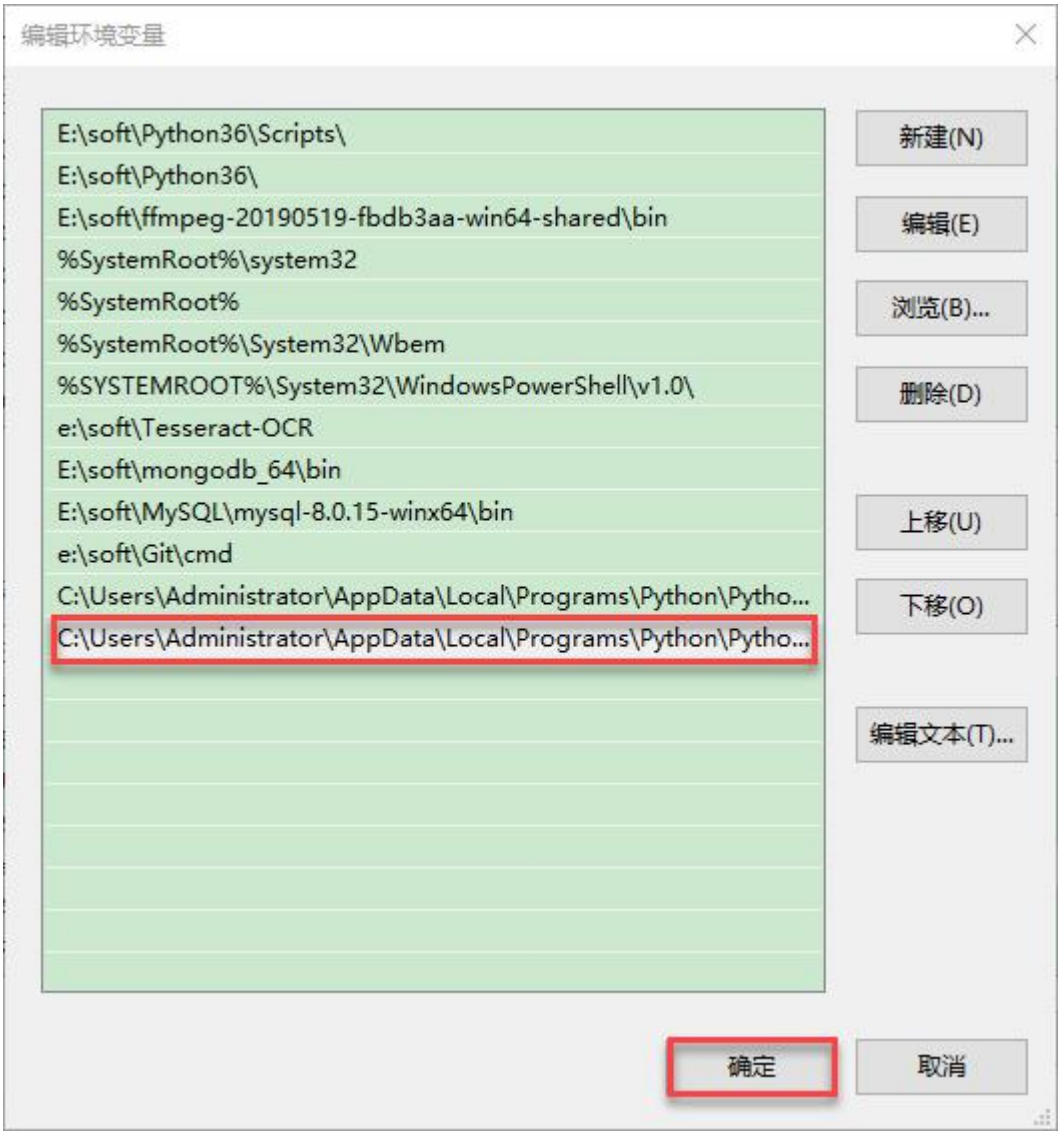


图 1-16 添加配置

6)打开 Windows 命令行工具，执行 python –-version 命令，如果输出如图 1-17 所示的内容，则说明 Python 已经安装成功了。

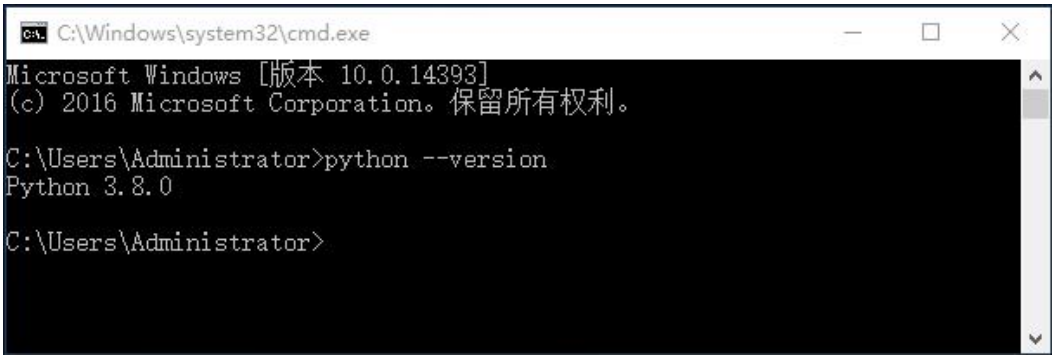


图 1-17 测试 Python 运行环境

1.2.3 安装 PyCharm

Python 的开发工具非常丰富，有许多强大的 IDE(Integrated Development Environment)工具，如 IDLE、Pycharm、wingIDE、Eclipse、IPython 等。这些工具不仅支持图形化操作，而且具备编辑、调试等功能。此外文本编辑器也可作为 Python 的开发环境，如 EditPlus、Vi 等。PyCharm 是 JetBRAINS 公司的开发的 Python IDE，功能强大，近期还发布了开源社区版本，非常适合于学习。

首先进入 https://www.jetbrains.com/pycharm/download/#section=windows 中 PyCharm 下载页面，如图 1-18 所示。

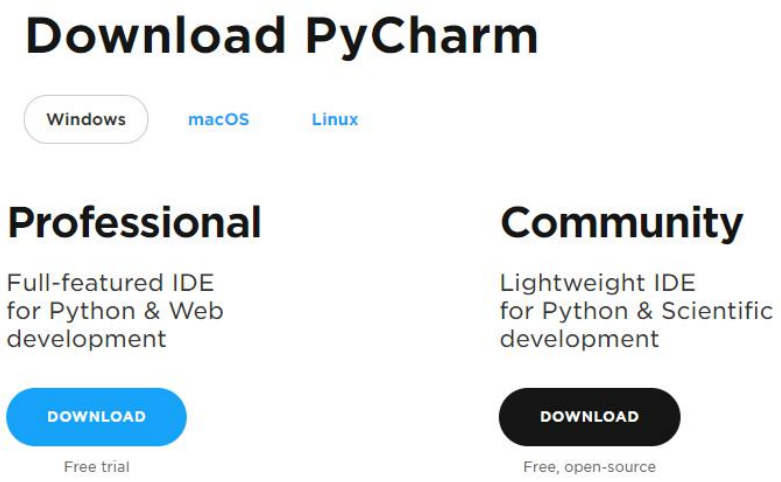


图 1-18 下载 PyCharm

根据下载路径找到下载好的 PyCharm 安装程序（pycharm-professional-2018.2.4.exe），具体步骤如下。

1) 双击安装程序 pycharm-professional-2018.2.4.exe，此时会弹出安装界面，如图 1-19 所示。点击“Next”按钮，进入选择安装路径界面如图 1-20 所示。

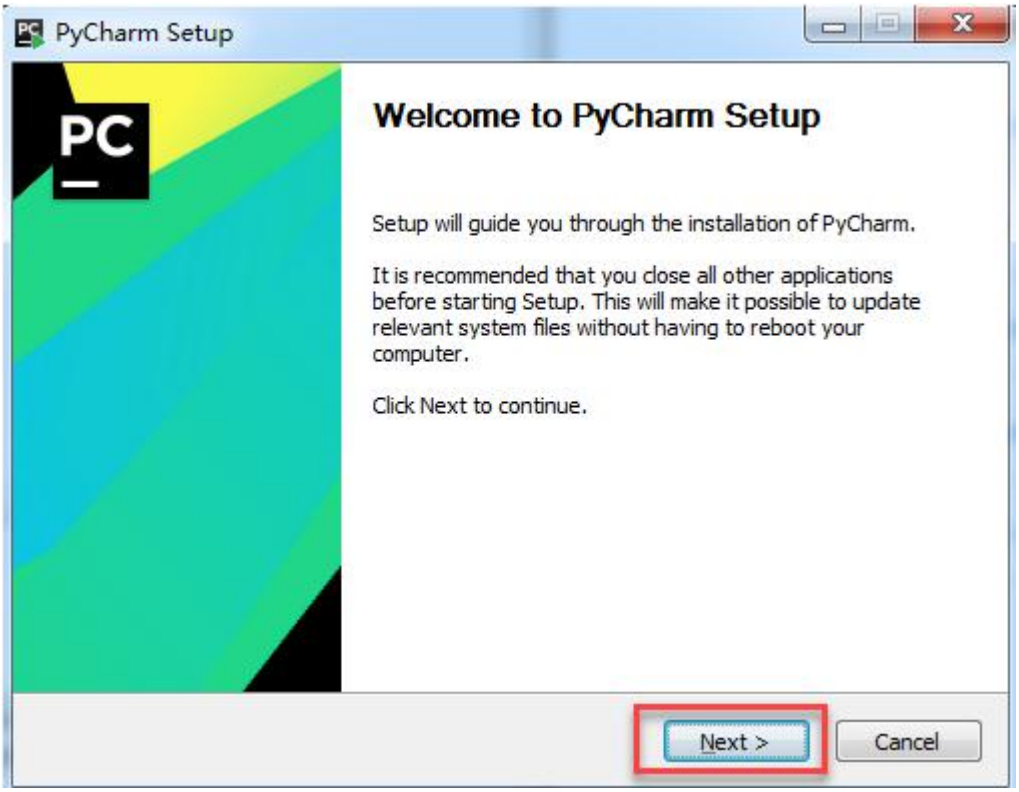


图 1-19 安装界面

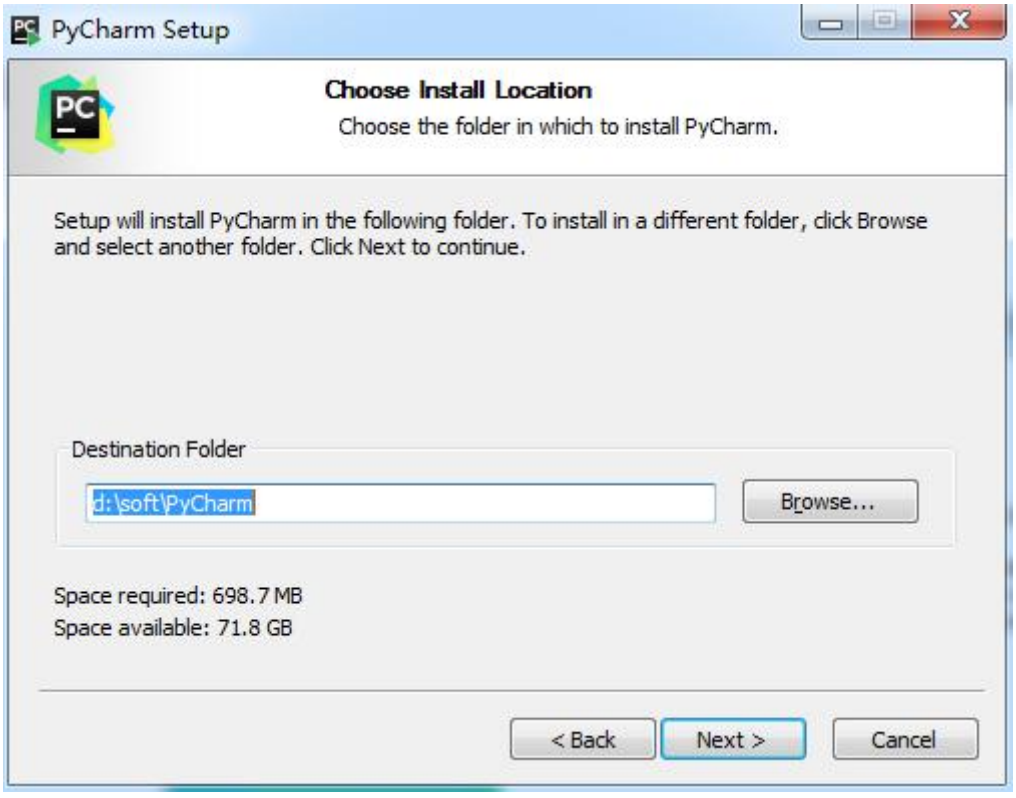


图 1-20 选择路径

2) 选择安装路径后（此路径可以根据用户自己的需求选择），然后单击“Next”按钮，如图 1-21 所示。

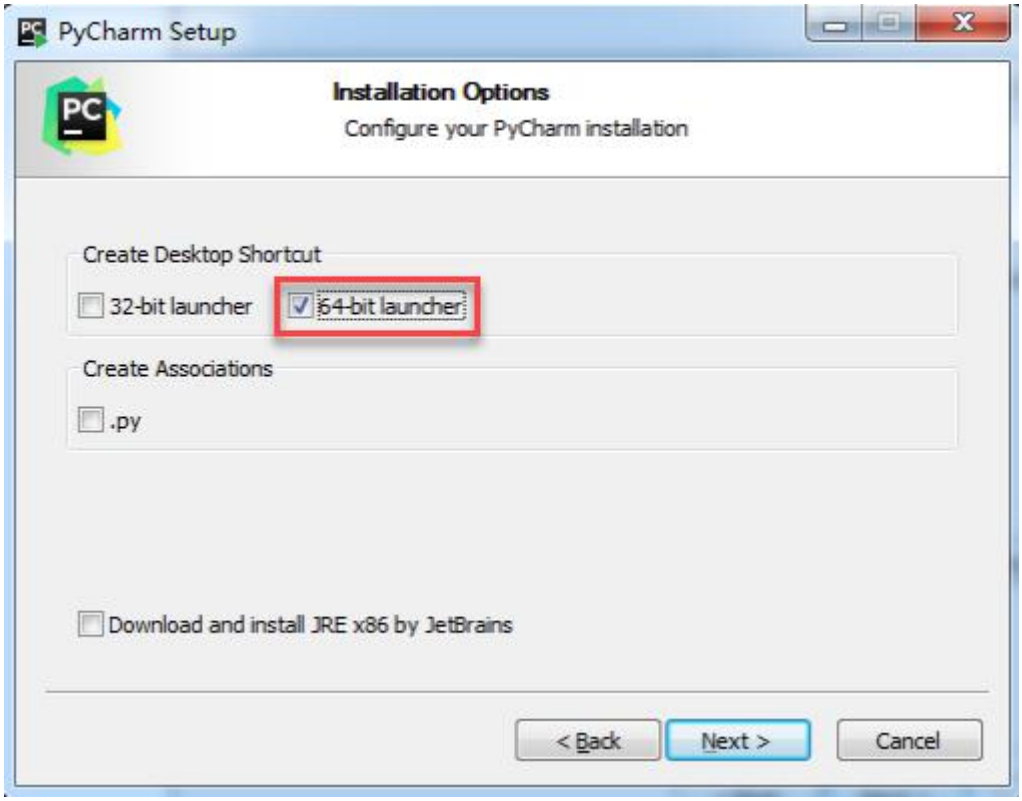


图 1-21 选择路径

3) 根据 win 系统是 64 还是 32 位，选择对应的位数操作系统，然后单击“Next”，按钮如图 1-22 所示。

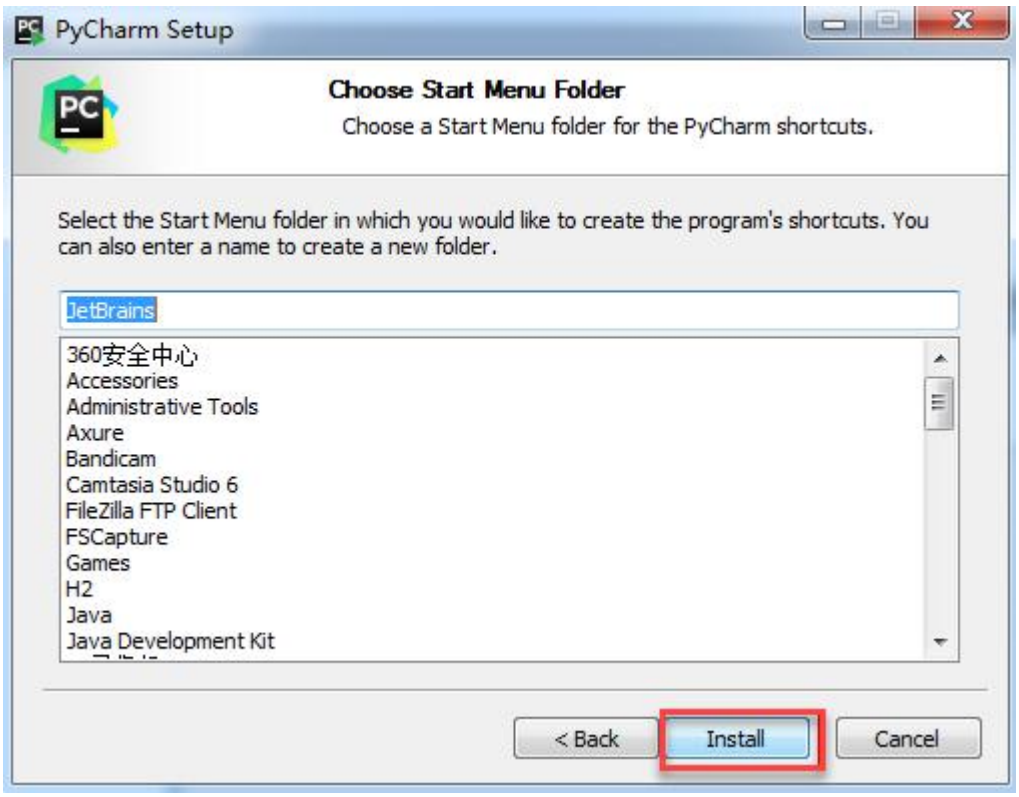


图 1-22 安装

4) 直接点击“Install”按钮进行安装，如图 1-23 所示是显示安装进度的界面。

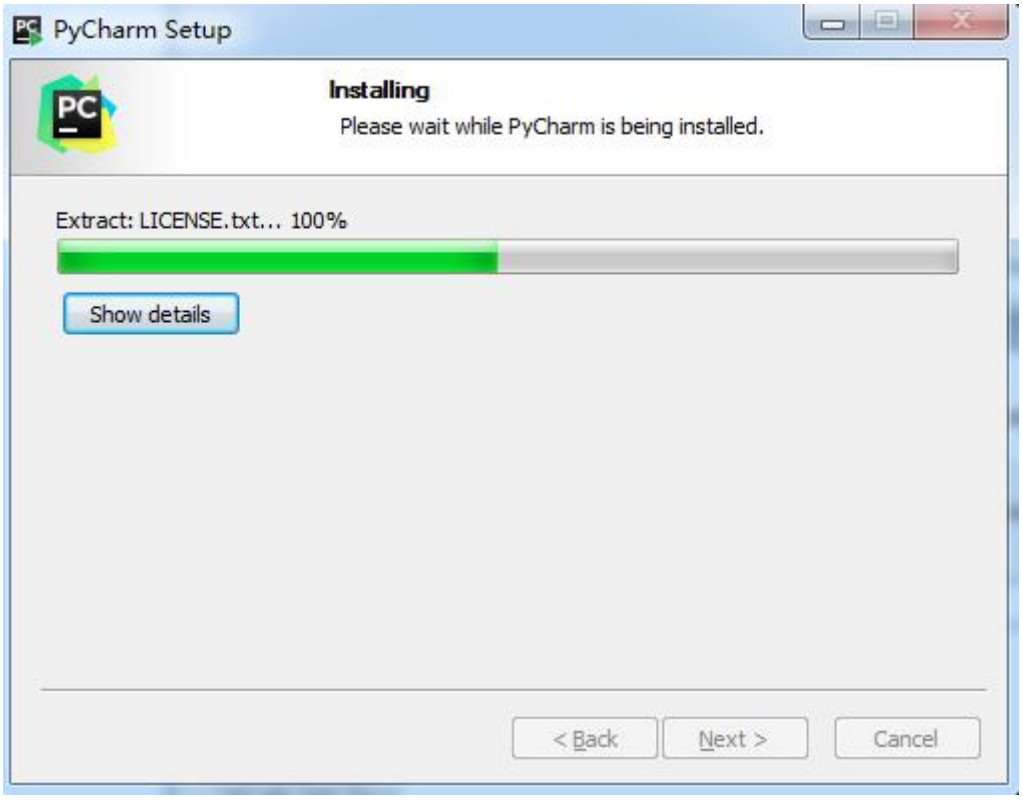


图 1-23 安装进度

5) 安装完后，会出现如图 1-24 所示的安装成功界面。

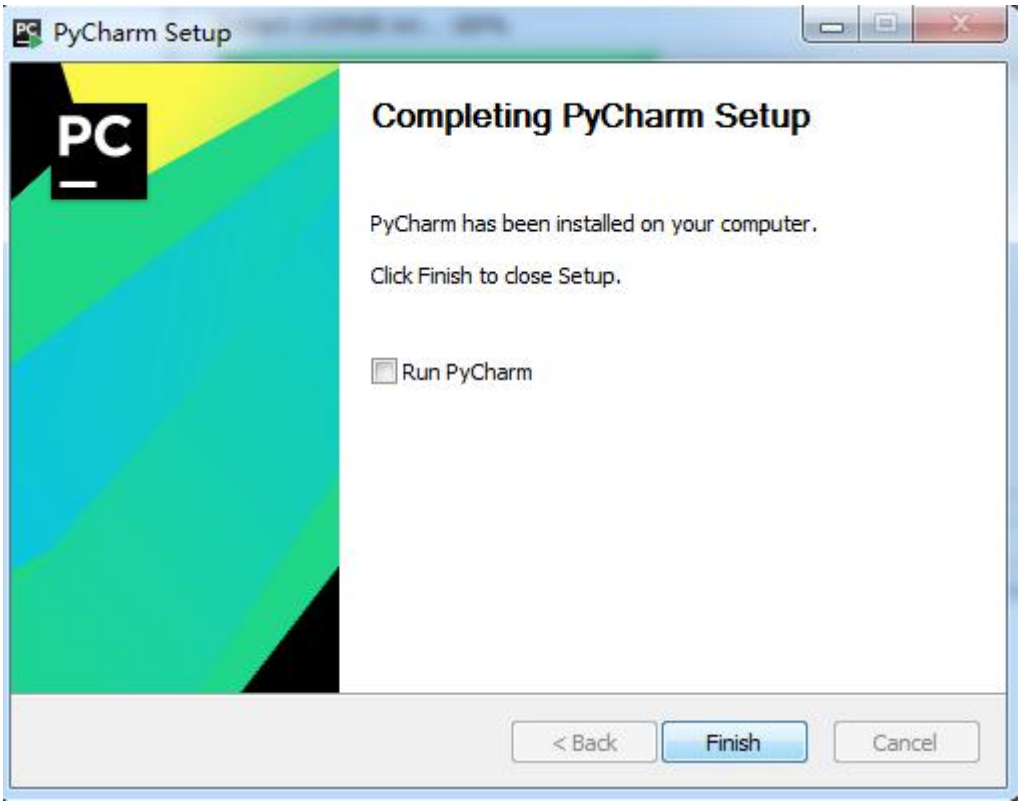


图 1-24 安装成功

1.2.4 配置 Pycharm

安装完成后，设置 Pycharm 的参数。仍然使用图形化界面对其进行配置，具体步骤如下。

1）勾选复选框 RunPyCharm 点击如图 1-24 中的“Finish”按钮，直接进入参数配置界面，如图 1-25 所示。

图 1-25 导入配置

2) 选择不导入配置，点击“ok”按钮。进入同意许可协议界面，将右侧滚动条拖到最下面，然后选择“accept”按钮，

图 1-26 许可协议

3) 不发送消息，选择“Don’t send”按钮，进入如图 1-27 选择试用界面，如果有激活码可以输入如图 1-28 所示。

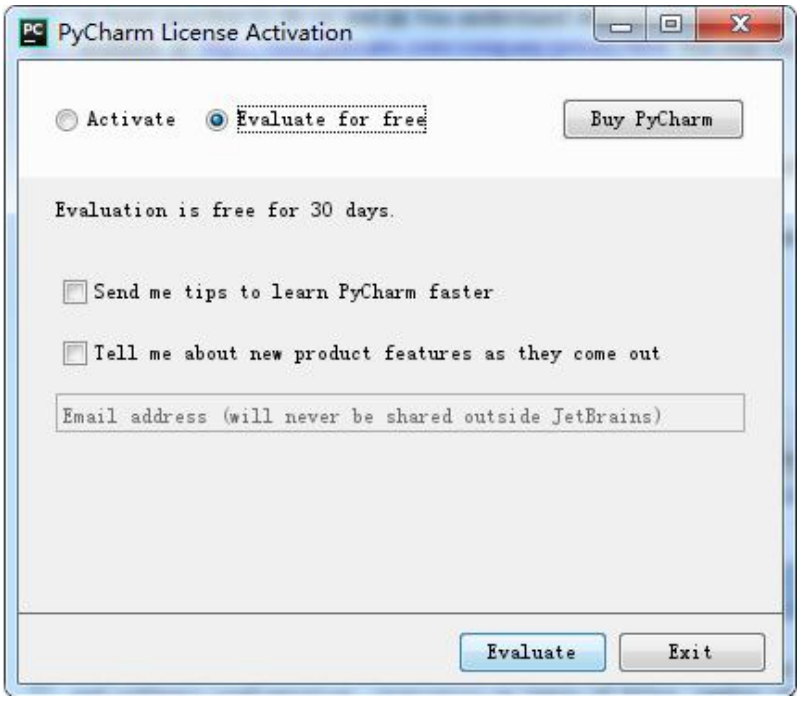


图 1-27 试用

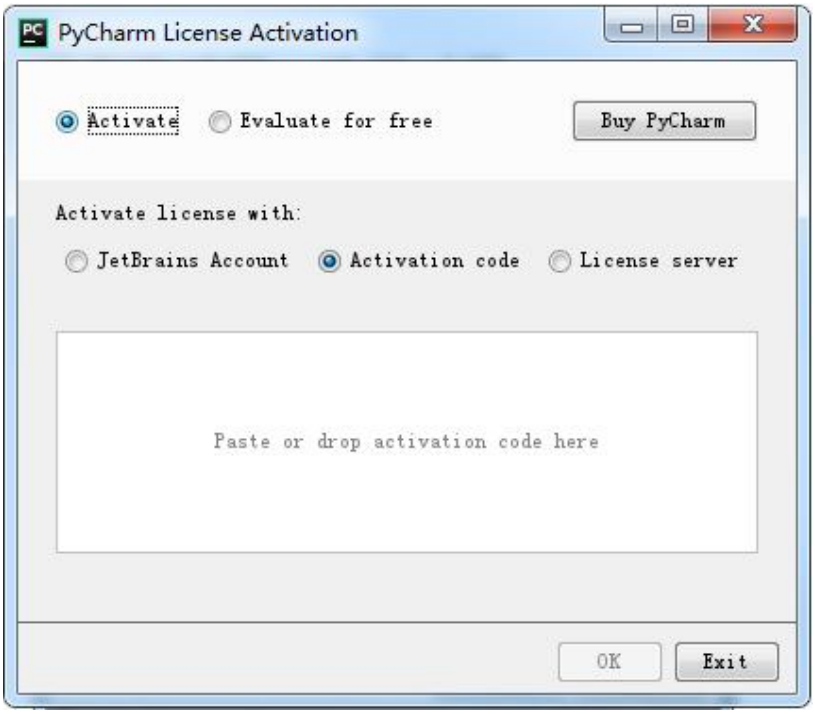


图 1-28 输入激活码

4) 选择试用或者输入激活码后进入选择 Dracula 风格或者 IntelliJ 风格。然后点击“Skip Remaining and Set Defaults”(根据自己喜好选择)，如图 1-29 所示。

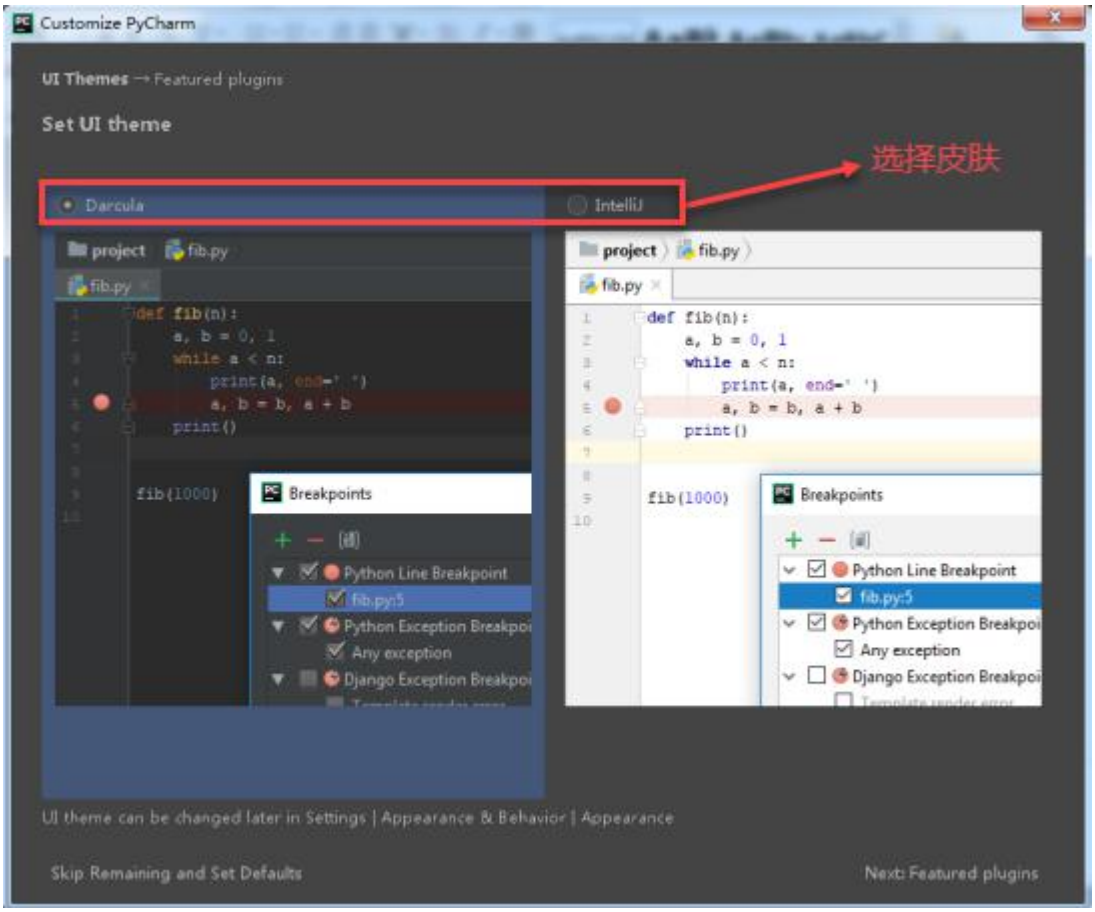


图 1-29 选择皮肤

1.2.5 创建项目初始化配置

安装配置完成后，创建项目。仍然使用图形化界面对其进行配置，具体步骤如下。

1) 选择：“Create New Project”，如图 1-30 所示

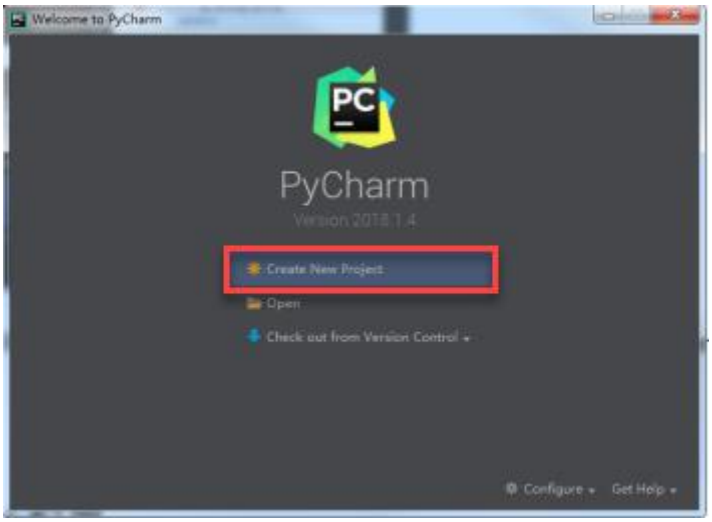


图 1-30 创建项目

2) 选择路径（尽量不要包含中文），文件名：mypro01 就是我们的项目名称，可以修改。其中 Project Interpreter 部分是选择新建项目所依赖的 python 库，第一个选项会在项目

中建立一个 venv（virtualenv）目录，这里存放一个虚拟的 python 环境。这里所有的类库依赖都可以直接脱离系统安装的 python 独立运行。 Existing Interpreter 关联已经存在的 python 解释器，如果不想在项目中出现 venv 这个虚拟解释器就可以选择本地安装的 python 环境。那么到底这两个该怎么去选择呢，这里建议选择 New Environment 可以在 Base Interpreter 选择系统中安装的 Python 解释器，这样做的好处如下：

- 1. python 项目可以独立部署
- 2. 防止一台服务器部署多个项目之间存在类库的版本依赖问题发生
- 3. 也可以充分发挥项目的灵活性

如图 1-31 所示

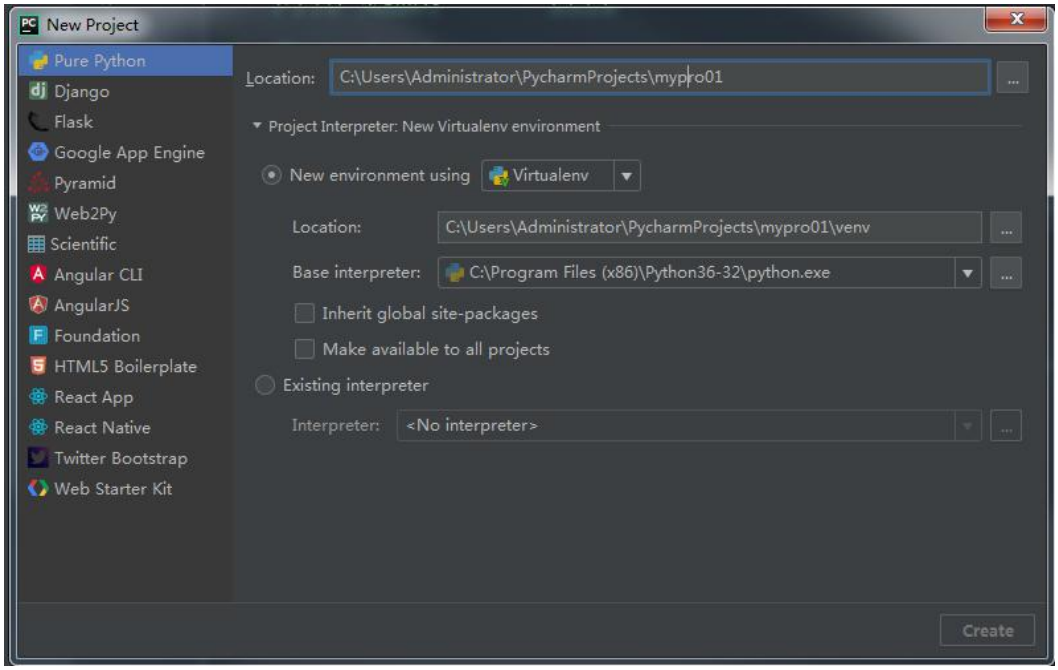


图 1-31 选择路径

3) 打开项目后，右键单击项目，创建 Python 文件“mypy01”，如图 1-32 所示

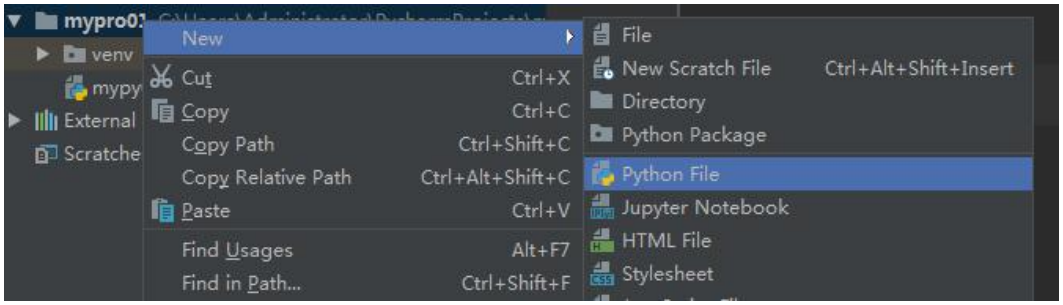


图 1-32 创建 Python 文件

4) 输入 print(“ssss”)，运行 py 文件，使用右键单击编辑区，选择“Run ‘mypy01’”即可。如图 1-33 所示。

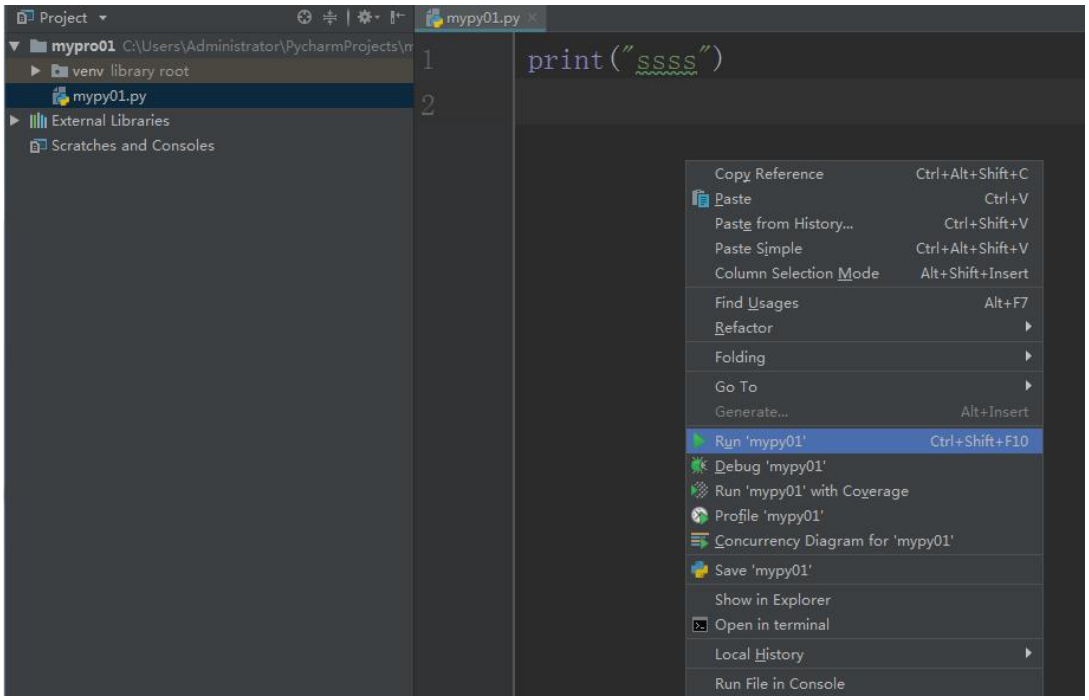


图 1-33 运行 Python 文件

1.2.6 学习图形化程序设计

为了让初学者更加容易接受编程，这里先从海龟画图开始讲解。这样，大家在不接触其他编程概念时，就能开始做出一些简单的效果。提高兴趣。

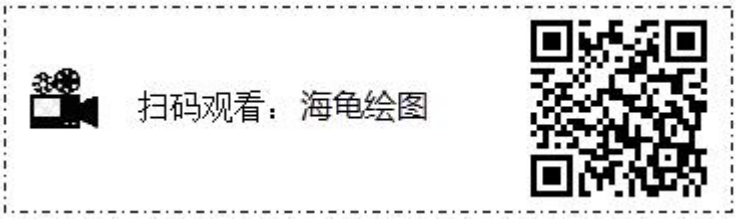


表 1-2 图形化语法介绍

语法	说明
import turtle	导入 turtle 模块
turtle.showturtle()	显示箭头
turtle.write("高淇")	写字符串
turtle.forward(300)	前进 300 像素
turtle.color("red")	画笔颜色改为 red
turtle.left(90)	箭头左转 90 度
turtle.forward(300)	前进 300 像素
turtle.goto(0,50)	去坐标（0,50）
turtle.goto(0,0)	去坐标（0,0）
turtle.penup()	抬笔。这样，路径就不会画出来
turtle.goto(0,300)	去坐标（0,300）

语法	说明
turtle.pendown()	下笔。这样，路径就会画出来
turtle.circle(100)	画圆

【示例 1-1】绘制奥运五环标记

```
import turtle

turtle.width(10)

turtle.color("blue")
turtle.circle(50)

turtle.color("black")
turtle.penup()
turtle.goto(120,0)
turtle.pendown()
turtle.circle(50)

turtle.color("red")
turtle.penup()
turtle.goto(240,0)
turtle.pendown()
turtle.circle(50)

turtle.color("yellow")
turtle.penup()
turtle.goto(60,-50)
turtle.pendown()
turtle.circle(50)

turtle.color("green")
turtle.penup()
turtle.goto(180,-50)
turtle.pendown()
turtle.circle(50)
```

执行结果如图 1-34 所示：

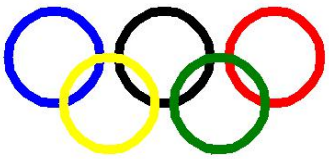


图 1-34 示例 1-1 执行结果

习题

一、选择题

1. 关于 Python 语言的特点，以下选项中描述错误的是（）

A.. Python 语言是非开源语言

B.. Python 语言是跨平台语言

C.. Python 语言是多模型语言

D.. Python 语言是脚本语言

2. 以下选项中说法不正确的是（）

A.. C 语言是静态语言，Python 语言是脚本语言

B.. 编译是将源代码转换成目标代码的过程

C.. 解释是将源代码逐条转换成目标代码同时逐条运行目标代码的过程

D.. 静态语言采用解释方式执行，脚本语言采用编译方式执行

3. IDLE 环境的退出命令是（）

A.. esc()　　B.. close()　C.. 回车键　D.. exit()

4. Python Web 开发方向的第三方库是（）

A. Django

B. scipy

C. pandas

D. requests

5. 崇尚优美、清晰、是一个优秀并广泛使用的语言，得到行内众多领域的认可，下列属于 Python 主要应用领域的是:（）

A. 系统运维

B. 科学计算、人工智能

C. 云计算

D. 金融量化

二、简答题

1. Python 的特点。
2. Python 的应用场景。
3. Windows 平台下 Python 环境的搭建过程。
4. PyCharm 的安装与配置。
5. 环境变量 path 的作用及配置。

三、编码题

1. 编写程序，完成奥运五环的绘图程序

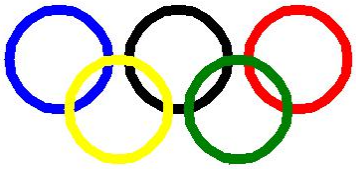


图 1-35 奥运五环

2. 使用海龟绘图，输出四个矩形：

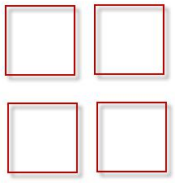


图 1-36 四个矩形

第二章 Python 基础语法

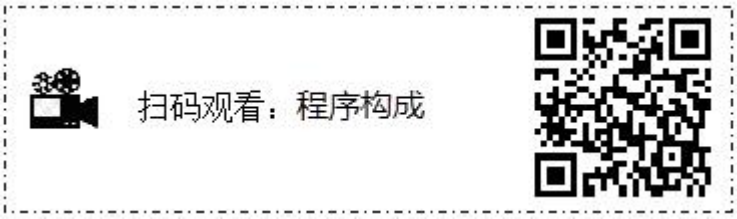
Python 的语法非常简练，因此用 Python 编写的程序可读性强、容易理解。本章将向读者介绍 Python 的基本语法、概念及其数据类型。Python 的语法与其他高级语言有很多不同，Python 使用了许多标记作为语法的一部分，例如空格、缩进、冒号等。还可以学习如何将数据存储到变量中，以及如何在程序中使用这些变量。

通过阅读本章，你可以：

- 了解如何声明变量
- 掌握如何在二、八、十和十六进制之间互相转换
- 掌握如何格式化数字
- 掌握当行注释和多行注释
- 了解单引号字符串和双引号字符串
- 掌握字符串的常用方法

2.1 Python 程序中的基本元素

2.1.1Python 程序的构成



尽管 Python 语言与 Java 语言一样，是一种面向对象语言，但 Python 语言相对 Java 语言来说，代码编写比较自由。Python 语言并不像 Java 语言那样需要定义一个 main 方法作为入口点。Python 程序的代码可以像批处理文件一样从上到下按顺序编写，这也是为什么 Python 语言适合运维的原因。

尽管 Python 语言的代码自由度更大，但这并不等于 Python 语言的代码可以随便编写，Python 语言的代码仍然需要一定的结构，例如，对于有一定复杂难度的程序，不可能只是用 Python 语言中内置的功能，还需要引用很多外部的 API，这些 API 在 Python 语言中称为模块。

因此，Python 程序由模块组成。一个模块对应 python 源文件，一般后缀名是：.py。模块由语句组成。运行 Python 程序时，按照模块中语句的顺序依次执行。语句是 Python 程序的构造单元，用于创建对象、变量赋值、调用函数、控制语句等。如图 2-1 所示：

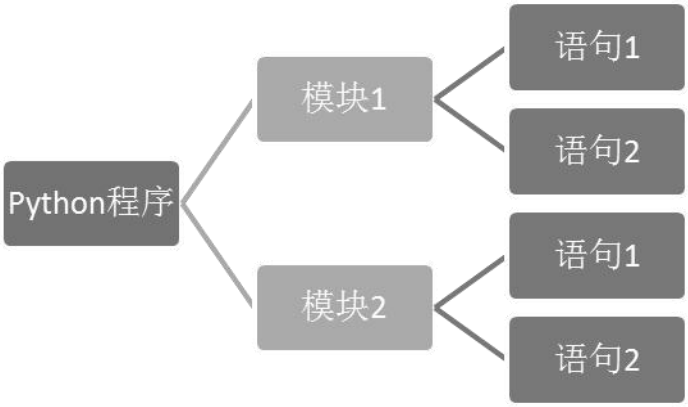
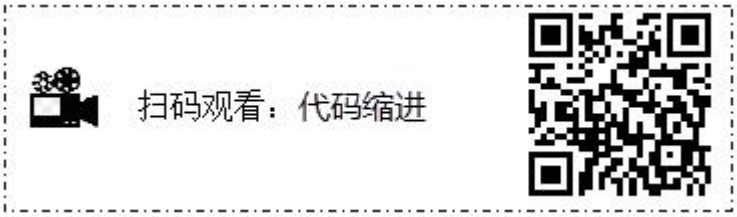


图 2-1 Python 程序组成

2.1.2 代码的组织和缩进



很多编程语言通过字符(例如：花括号{})、关键字(例如：begain/end)来划分代码块。同时，在配合代码的缩进增加可读性。“龟叔”设计 Python 语言时，直接通过缩进来组织代码块。“缩进”成为了 Python 语法强制的规定。

缩进时，几个空格都是允许的，但是数目必须统一。通常采用“四个空格”表示一个缩进。同时，也要避免将“tab 制表符”或者 tab 与空格混合的缩进风格。目前，常用的编辑器一般设置成：tab 制表符就是 4 个空格。

Python 与 C、Java 等高级语言不同，其代码块不使用大括号{}来控制语句、函数、类的作用域。Python 采用独特的缩进来控制模块作用域。缩进的方式有二种：

- Python 编辑器自动生成缩进：每当回车时，编辑器会自动完成缩进工作。
- 人为手工代码缩进：缩进的空白数量可变，但同一函数的空白数必须保持一致。

缩进的空白数量是可变的，但是所有代码块语句必须包含相同的缩进空白数量，且必须严格的执行这种规则。

【示例 2-1】代码测试缩进

```
if    False:
      print ("one")
      print ("two")
else:
print( "three") #  缩进没有对齐，在执行时会报错
   print ("four")
```

执行结果如图 2-2 所示:

图 2-2 示例 2-1 执行效果图

注意：

- 每个缩进层次使用单个制表符或两个空格或四个空格，互相之间不能混用。
- 缩进的空白数量可变，但同一函数的空白数必须保持一致。
- 代码块中必须使用相同数目的行首缩进空格数

2.1.3 注释

任何编程语言都有注释的功能。所谓注释，就是用一段文本描述代码的作用、代码的作者或是其他需要描述的东西。注释在程序编译时被忽略，也就是说，注释只在源代码中体现，编译生成的二进制文件中是没有注释的。

在 Python 语言中，注释分为单行注释和多行注释。单行注释用井号(#)开头，多行注释用三个引号(单引号或双引号)括起来。如果使用单行注释，井号后面的所有内容在编译程序时都会被忽略，如果使用多行注释，被引号括起来的内容在编译程序时都会被忽略。

【示例 2-2】单行注释和多行注释的使用

```
#我是单行注释
print('单行注释演示')
'''
我是多行注释
三个单引号实现多行注释
作者:
时间：
'''
print('三个单行引号实现多行注释')
"""
三个双引号实现多行注释
作者:
时间:
"""
```

```
print('双引号实现多行注释')
```

执行结果如图 2-3 所示：

图 2-3 示例 2-2 执行结果

2.1.4 使用\行连接符

一行程序长度是没有限制的，但是为了可读性更强，通常将一行比较长的程序分为多行。这时可以使用“\”行连接符，把它放在行结束的地方。Python 解释器仍然将它们解释为同一行。

【示例 2-3】使用“\”行连接符

```
s = 'I love ' \
    'py' \
    'thon'
print(s)
```

执行结果如图 2-4 所示：

图 2-4 示例 2-3 执行结果

2.1.5 标识符

标识符主要用于对变量、函数、类、模块等的命名。Python 语言有一套自己的命名规则，用户也可以借鉴 Java 语言的命名规则形成自己命名的规则。Python 语法中，标识符由数字、字母、下划线组成，所有标识符不能以数字开头，且 Python 标识符区分字母的大小写。以下划线开始的标识符有特殊意义，归类如下：

- _单下划线开头：不能直接访问类属性，需要通过接口进行访问，不能采用 from xxx

import *的方式。

- __双下划线开头：类的私有成员。
- __xxx__双下划线开头和结尾：特殊方法专用的标识。如_init_()代表构造函数。

标识符有如下特定的规则：

1) 区分大小写。如：sxt 和 SXT 是不同的
2) 第一个字符必须是字母、下划线。其后的字符是：字母、数字、下划线
3) 不能使用关键字。比如：if、or、while 等。
4) 以双下划线开头和结尾的名称通常有特殊含义，尽量避免这种写法。比如：__init__ 是类的构造函数。

开发中，我们通常约定俗称遵守如下表 2-1：

表 2-1 命名规则

类型	规则	例子
模块和包名	全小写字母，尽量简单。若多个单词之间用下划线	math, os, sys
函数名	全小写字母，多个单词之间用下划线隔开	phone, my_name
类名	首字母大写，采用驼峰原则。多个单词时，每个单词第一个字母大写，其余部分小写	MyPhone、MyClass、Phone
常量名	全大写字母，多个单词使用下划线隔开	SPEED、MAX_SPEED

Python 保留字不能作为常数或变量，以及其它任何标识符名称，且都是小写字母。

表 2-2 Python 保留字

and	def	exec	if	not	return
assert	del	finally	import	or	try
break	elif	for	in	pass	while
class	else	from	is	print	with
continue	except	global	lambda	raise	yield

【示例 2-4】合法的标识符

```
a=123
n_123='abc'
```

【示例 2-5】不合法的标识符

```
23a=123          #不能以数字开头
a#=123           #不能包含#这样的特殊符
```

```
and=123          #不能使用保留字
```

2.1.6 print 输出

1）普通的输出

print 的是打印，出版的意思。在 Python 中 print 代表输出的意思，比如说向控制台输出一句”Hello Python !”。

【示例 2-6】使用 print 打印”Hello Python !”

```
print("Hello Python")
```

执行结果如图 2-5 所示：

图 2-5 示例 2-6 执行结果

print()函数还可以打印多行文字，接下来就来使用 print 函数来打印一首<<静夜思>>。

【示例 2-7】使用 print 打印一首<<静夜思>>

```
#输出多行字符串
print('''
      静夜思
       李白
   床前明月光,
   疑是地上霜。
   举头望明月,
   低头思故乡。
''')
```

执行结果如图 2-6 所示：

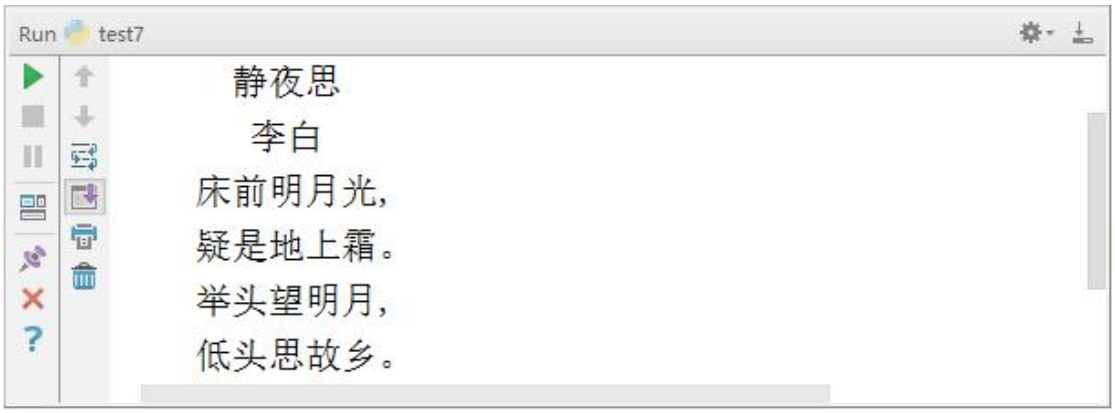


图 2-6 示例 2-7 执行结果

print()函数也可以接收多个字符串，使用”,”号隔开，这样就可以连成一串输出。

【示例 2-8】使用 print 打印多个字符串

```
#print 接收多个参数
print("The quick brown fox","jumps over","the lazy dog")
```

执行结果如图 2-7 所示：

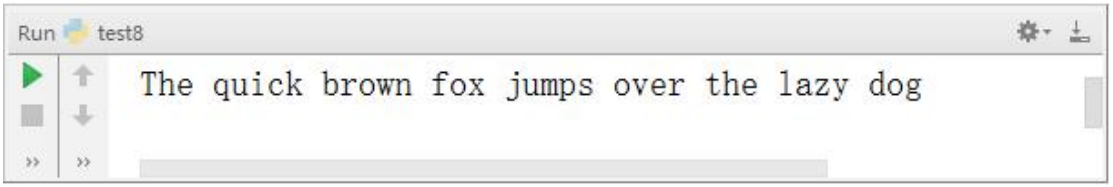


图 2-7 示例 2-8 执行结果

【示例 2-9】使用 print()对表达式进行美化

```
#print()对表达式进行美化
print("300+200",300+200)
```

执行结果如图 2-8 所示：

图 2-8 示例 2-9 执行结果

注意：

- 对于 100 + 200，Python 解释器自动计算出结果 300，但是'100 + 200 ='是字符串而非数学公式，Python 把它视为字符串

2）格式化输出

一个常见的问题是如何输出格式化的字符串。经常会输出类似'亲爱的 xxx 你好！你 xx

月的话费是 xx，余额是 xx 元'之类的字符串，而 xxx 的内容都是根据变量变化的，所以，需要一种简便的格式化字符串的方式。在 python 中,采用的格式化方式和 C 语言是一致的,都是使用%来实现的。

在字符串内部，%s 表示用字符串替换，%d 表示用整数替换，有几个%?占位符，后面就跟几个变量或者值，顺序要对应好。如果只有一个%?，括号可以省略。常见的占位符如表 2-3。

表 2-3 Python 占位符

占位符	类型	示例
%d	整数	a=10,b=5 , print(“%d-%d=%d”%(a,b,(a-b)))
%f	浮点数	a=3.5　print(“大白菜%0.1f 一斤”%a)
%s	字符串	name=”admin”;print(“我叫%s”%name)
%x	十六进制整数	a=20　print(“%x” %a)

【示例 2-10】使用%实现格式化输出

```
#格式输出
name="张三"
month=10
change=56.7
balance=23.3
print("亲爱的%s 您好，您%d 的话费是%f,余额为%f"%(name,month,change,balance))
```

执行结果如图 2-9 所示：

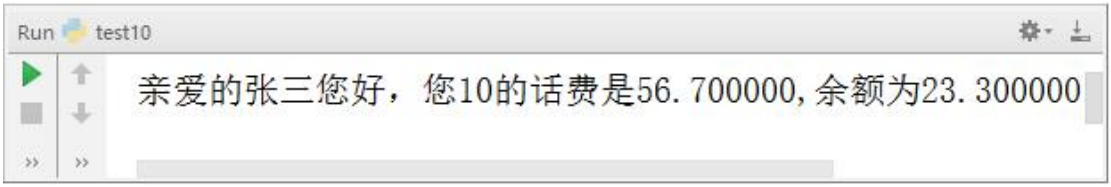


图 2-9 示例 2-10 执行结果

2.1.7 input 输入

要编写一个有实际价值的程序，就需要与用户交互。当然，与用户交互有很多方法。例如，GU1(图形用户接口)就是一种非常好的与用户交互的方式，不过先不讨论 GU1 的交互方式，本节会采用一种原始，但很有效的方式与用户交互，这就是命令行交互方式。也就是说，用户通过命令行方式输入数据，程序会读取这些数据，并做进一步的处理。

从命令行接收用户的输入数据，需要使用 input 函数。 input 函数接收一个字符串类型

的参数，作为输入的提示。 input 函数的返回值就是用户在命令行中录入的值。不管用户录入什么数据，input 函数都会以字符串形式返回。

如果要获取其他类型的值，如整数、浮点数，需要用相应的函数转换。例如，字符串转换为整数的函数是 int，字符串转换为浮点数的函数是 float。

【示例 2-11】使用 input 获取键盘输入

```
#input 输入
userName=input('请输入你的姓名:')
age=int(input('请输入你的年龄:'))
sal=float(input('请输入你的薪资:'))
print('姓名是%s,年龄是%d,薪资是%f'%(userName,age,sal))
```

执行结果如图 2-10 所示：

图 2-10 示例 2-11 执行结果

注意：

- 由于年龄和薪资都是数值,所以在获取用户输入值后,需要分别使用 int 和 float 函数将 Input 函数的返回值转换为整数和浮点数。

2.2 变量和简单的赋值语句

2.2.1 对象和引用

Python 中，一切皆对象。每个对象由：标识（identity）、类型（type）、value（值）组成。对象的本质就是：一个内存块，拥有特定的值，支持特定类型的相关操作。

1) 标识用于唯一标识对象，通常对应于对象在计算机内存中的地址。使用内置函数 id(obj)可返回对象 obj 的标识。
2) 类型用于表示对象存储的“数据”的类型。类型可以限制对象的取值范围以及可执行的操作。可以使用 type(obj)获得对象的所属类型。
3) 值表示对象所存储的数据的信息。使用 print(obj)可以直接打印出值。

在 Python 中，变量也成为：对象的引用。因为，变量存储的就是对象的地址。变量通过地址引用了“对象”。

变量位于：栈内存。

对象位于：堆内存。

·Python 是动态类型语言

变量不需要显式声明类型。根据变量引用的对象，Python 解释器自动确定数据类型。

·Python 是强类型语言

每个对象都有数据类型，只支持该类型支持的操作。

【示例 2-12】对象和引用

```
>>> a = 3
>>> a
3
>>> id(3)
1531372336
>>> type(3)
<class 'int'>
>>> b = "我爱你"
>>> id(b)
46806816
>>> type(b)
<class 'str'>
```

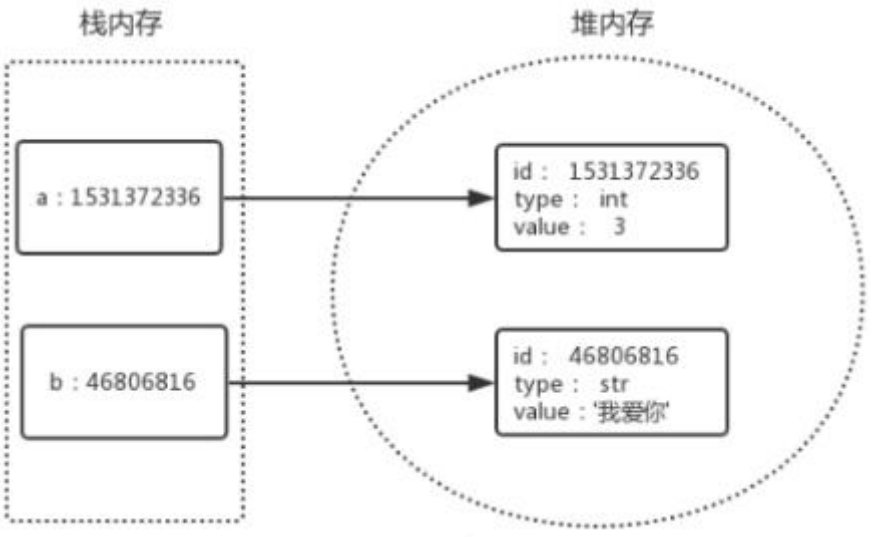
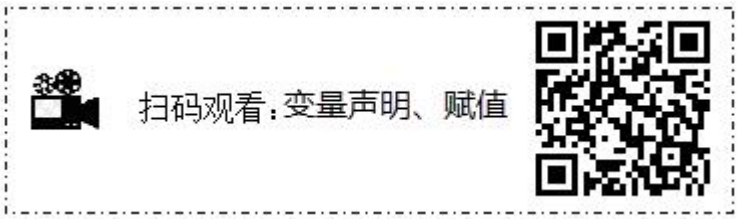


图 2-11 示例 2-12 内存示意图

2.2.2 变量的声明和赋值

变量 (variable) 是 Python 语言中非常重要的概念。变量的主要作用就是为 Python 中的某个值起个名字。类似于“张三”“李四”“王五”一样的人名，便于记忆。



变量的声明和赋值用于将一个变量绑定到一个对象上，语法结构如下：

```
变量名 = 变量值
```

最简单的表达式就是字面量。比如：a = 123 。 运行过程中，解释器先运行右边的表达式，生成一个代表表达式运算结果的对象，然后，将这个对象地址赋值给左边的变量。

在 Python 语言中，声明变量的同时需要为其赋值，毕竟不代表任何值的变量毫无意义，Python 中也不允许有这样的变量。

【示例 2-13】变量在使用前必须先被初始化（先被赋值）

```
#声明变量 my_name 没有赋值
my_name
print(my_name)#输出 my_name 报如下图异常
```

执行结果如图 2-12 所示：

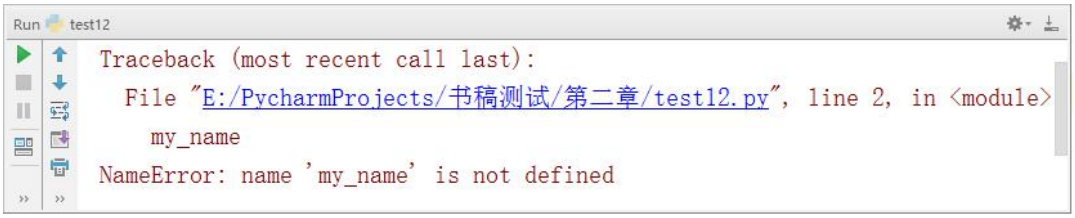


图 2-12 示例 2-13 执行结果

【示例 2-14】定义整数、字符串、浮点数的变量，并输出这些变量的值

```
#声明整数类型变量
a=36
#声明字符串类型变量
b='abc'
#声明浮点类型变量
c=55.41
#输出变量的值
```

```
print(a)
print(b)
print(c)
```

执行结果如图 2-13 所示：

图 2-13 示例 2-14 执行结果

变量存储在内存当中，当为变量赋值时，系统会在内存中开辟一个存储空间。变量中存放的数据都具有特定的数据类型，变量可以存储整数、小数、复数、字符等。如图 2-14 所示：

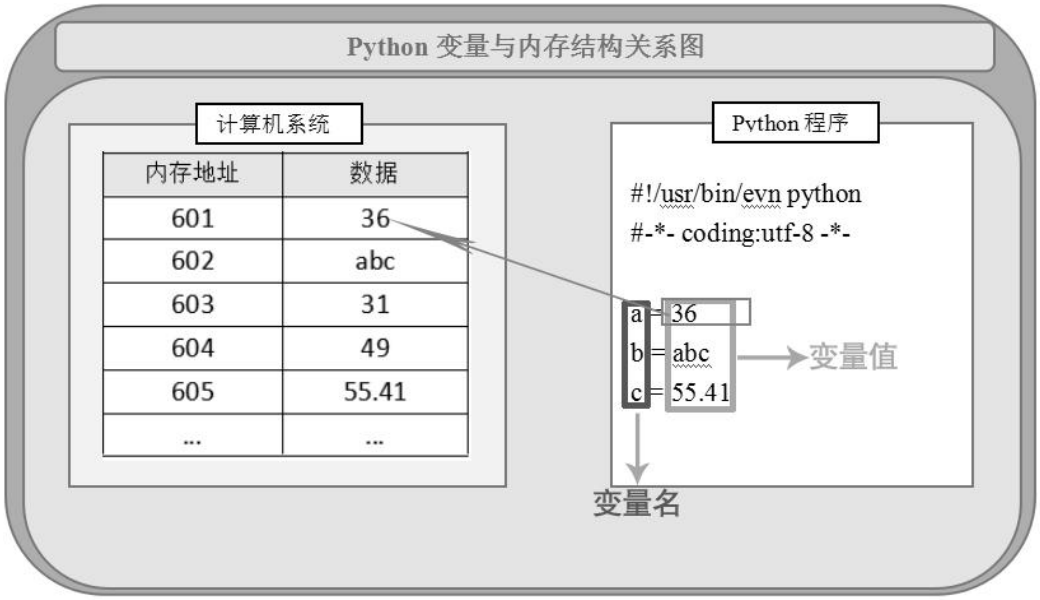


图 2-14 Python 变量与内存结构关系图

2.2.3 链式赋值

Python 变量赋值不需要类型声明，直接赋值即可，且 Python 支持多个变量赋值，多变量赋值语法格式。


```
变量名 1=变量名 2= 变量名 3...=变量值
```

【示例 2-15】链式赋值

```
#声明变量 a，b，c 并都赋值为 20
a = b = c = 20
print('a:',a)
print('b:',b)
print('c:',c)
```

执行结果如图 2-15 所示：

图 2-15 示例 2-15 执行结果

2.2.4 系列解包赋值

Python 变量赋值不需要类型声明，直接赋值即可，且 Python 支持系列数据赋值，及给对应相同个数的变量（个数必须保持一致）赋值。赋值语法格式如下所示：

```
变量名 1,变量名 2, 变量名 3...=变量值 1,变量值 2,变量 3....
```

【示例 2-16】使用系列解包赋值实现变量交换

```
#系列赋值，给多个变量赋值
a,b=10,20
print('a:',a)
print('b:',b)

print('交换变量后')
#实现变量交换
b,a=a,b
print('a:',a)
print('b:',b)
```

执行结果如图 2-16 所示：

图 2-16 示例 2-16 执行结果

2.2.5 删除变量

Python 中可以通过 del 语句删除不在使用的变量。语法格式如下所示：

```
del 变量名
```

【示例 2-17】使用 del 语句删除变量

```
#定义变量 x y
x=23
y='abc'
print('x:',x)
print('y:',y)

print('使用 del 删除变量后')
#使用 del 删除变量
del y
print('x:',x)
print('y:',y)
```

执行结果如图 2-17 所示：

图 2-17 示例 2-17 执行结果

2.3 数据类型

物以类聚，人以群分，数据亦是如此。数据类型是一个集合以及定义在这个集合上的一组操作。数值类型必然都是数字，字符串类型必然都是字符串。当然，还会有一些高级的数据类型，初学者可能不是很容易理解，但相信通过我们的讲解会顺利掌握。

Python 语言中的各种数据类型，如表 2-4 所示：

表 2-4　Python 数据类型

基本数据类型	Number	Integer（整型）	
		Long integer（长整型）	
		Double-precision floating（双精度浮点型）	
		Complex number（复数型）	
	Boolean	True 或 False（布尔型）	
	String	零个或多个字符组成的有限序列（字符串型）	Sequence 类型簇
高级数据类型	Tuple	内部元素不可修改（元组型）	Sequence 类型簇
	List	但内部元素可以修改（列表型）	Sequence 类型簇
	Set	内部元素相互之间无序的一组对象（集合型）	
	Dictionary	拥有键/值对的特殊列表，形式 key:value（字典型）	

注意：

- Sequence 类型簇：可以将字符串看作由字符组成的序列类型，包括 String、Tuple、List 三种类型

2.3.1 数字

数字是 Python 程序中最常见的元素。在 Python 语言中，数字分为整数和浮点数。支持基本的四则运算和一些其他的运算操作如表 2-5，并且可以利用一些函数在不同的进制之间进行转换，以及按一定的格式输出数字。

如果要计算两个数的除法，不管分子和分母是整数还是浮点数，使用除法运算符(/)的计算结果都是浮点数。例如，1/4 的计算结果是 0.25，8/2 的结果是 4.0。想要让 Python 解析器执行整除操作，可以使用整除运算符，也就是两个斜杠(//)。使用整除运算符后，8//2 的计算结果就是 4，1//4 的计算结果就是 0。

表 2-5　基本运算符

运算符	说明	示例	结果
+	加法	3+2	5
-	减法	30-5	25
*	乘法	3*6	18
/	浮点数除法	8/2	4.0
//	整数除法	7//2	3
%	模（取余）	7%4	3
**	幂	2**3	8

【示例 2-18】基本运算符的使用

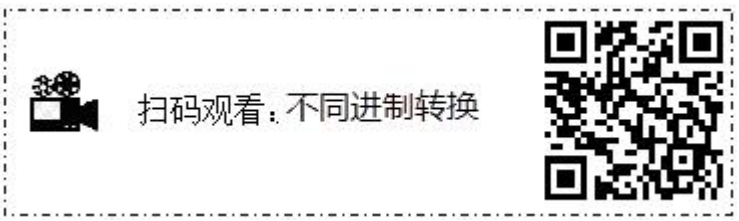
```
#基本运算符的使用
print(3+4)          #运算结果:7
print(129-789)      #运算结果:-660
print(5+20*5)       #运算结果:105
print(1/4)          #运算结果:0.25
print(8/2)          #运算结果:4.0
print(8//3)         #运算结果:2
print(8%3)           #运算结果:2
print(4**3)         #运算结果 64
```

执行结果如图 2-18 所示：

图 2-18 示例 2-18 执行结果

2.3.2 二进制、八进制和十六进制

Python 语言可以表示二进制、八进制和十六进制数。表示这三个进制的数，必须以 0 开头，然后分别跟着表示不同进制的字母。表示二



进制的字母是 b，表示八进制的字母是 o(这里是英文字母中小写的 o，不要和数字 0 搞混了)，表示十六进制的字母是 x。因此，二进制数的正确写法是 0b111000，八进制数的正确写法是 0o123456，十六进制数的正确写法是 0xF123EA。

前面一直使用的是十进制，Python 语言一共可以表示 4 种进制：二进制、八进制、十进制和十六进制。Python 语言提供了一些函数用于在这 4 种进制数之间进行转换。

如果是从其他进制转换到十进制，需要使用 int 函数。该函数有两个参数，含义如下：

(1)第一个参数为字符串类型，表示待转换的二进制、八进制或十六进制。参数值只需要指定带转换的数即可，不需要使用前缀，如二进制直接指定 101110，不需要指定 0b101110。

(2)第二个参数为数值类型，表示第 1 个参数值的进制，例如，如果要将二进制转换为十进制，第 2 个参数值就是 2。

(3)int 函数返回一个数值类型，表示转换后的十进制数。

下面的代码将二进制数 101101 转换为十进制数，并输出返回结果。

【示例 2-19】二进制数 101101 转换为十进制数

```
#二进制数 101101 转换为十进制数
print(int('101101',2))
```

执行结果如图 2-19 所示：

图 2-19 示例 2-19 执行结果

如果要将十进制转换到其他进制，需要分别使用 bin、oct、和 hex 函数。bin 函数用于将十进制的数转换为二进制数，oct 函数用于将十进制的数转换为八进制数，hex 函数将用于将十进制数转换为十六进制数。这三个函数都接收一个参数，就是待转换的十进制数。不过要注意，这三个函数的参数值也可以是二进制数、八进制数和十六进制数，也就是说，这三个函数可以在二进制、八进制、十进制和十六进制之间互转。

下面的代码将十进制数 12345 转换为十六进制数，并输出转换结果。

【示例 2-20】十进制数 12345 转换为十六进制数

```
#十进制数 12345 转换为十六进制数
print(hex(12345))
```

执行结果如图 2-20 所示：

图 2-20 示例 2-20 执行结果

下面的代码将 Python 语言中二进制、八进制、十进制和十六进制数之间的转换进行演示。

【示例 2-21】二进制、八进制、十进制和十六进制数之间的转换

```
#二进制、八进制、十进制和十六进制数之间的转换
print(0b110110)          #输出二进制数  输出结果:54
print(0o234)             #输出八进制  输出结果:156
print(0xF45)             #输出十六进制  输出结果:3909
print(bin(12345))        #十进制转换为二进制   输出结果:0b11000000111001
print(int('110110',2))    #二进制转换为十进制  输出结果:54
print(int('ABC',16))      #十六进制转换为十进制  输出结果:2748
print(hex(123456))        #十进制转换为十六进制  输出结果: 0x1e240
print(hex(0b11100001))   #二进制转换为十六进制  输出结果:0xe1
print(oct(123456))        #十进制转换为八进制   输出结果:0o361100
print(int('123456',8))      #八进制转十进制  输出结果：42798
```

执行结果如图 2-21 所示：

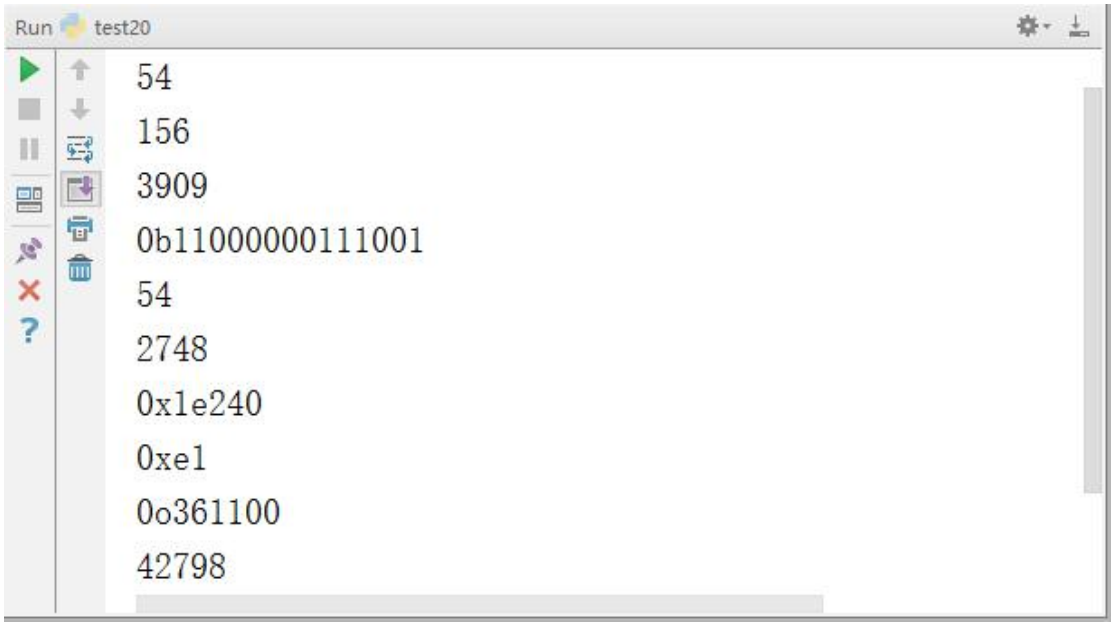


图 2-21 示例 2-21 执行结果

2.3.3 大整数

对于有符号 32 位整数来说，可表示的最大值是 2147483647(2^31-1)，可表示的最小值是-2147483647(-2^31)，如果超出这个范围，有符号 32 位整数就会溢出。不过在 Phon 语言中，可以处理非常大的整数，并不受位数限制，下面表达式的输出结果就超出了 32 位整数的范围。

下面的代码测试非常大的数，看看会不会溢出。

【示例 2-22】演示大整数是否会溢出

```
#测试非常大的数，看看会不会溢出
print(2**36)   #2 的 36 次幂
print(2**300) #2 的 300 次幂
```

执行结果如图 2-22 所示：

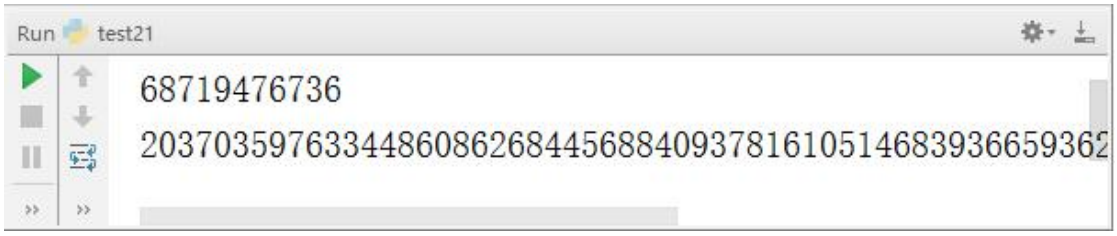


图 2-22 示例 2-22 执行结果

很显然，Python 语言仍然可以正确处理 2**600 的计算结果。因此，在 Python 语言中使用数字不需要担心溢出，因为 Python 语言可以处理非常大的数字，这也是为什么很多人使用 Python 语言进行科学计算和数据分析的主要原因。

2.3.4 浮点

Python 将带小数点的数字都称为浮点数。大多数编程语言都使用了这个术语，它指出了这样一个事实：小数点可出现在数字的任何位置。每种编程语言都须细心设计，以妥善地处理浮点数，确保不管小数点出现在什么位置，数字的行为都是正常的。

浮点数，称为 float。浮点数用形式的科学计数法表示。比如：3.14，表示成：314E-2 或者 314e-2。这些数字在内存中也是按照科学计数法存储。

运算符不仅可以对整数执行整除操作，也能对浮点数执行整除操作，在执行整除操作时，分子分母只要有一个浮点数，那么计算结果就是浮点数。例如，8.0//2 的计算结果就是 4.0. 除了四则运算符外，Python 还提供了两个特殊的运算符：%(取余运算符)和**（幂运算符）。取余运算符用于对整数和浮点数执行取余操作。例如，7%2 的计算结果是 1，而 7.0%2 的计算结果是 1.0。从这一点可以看出，%和//类似，只要是分子分母有一个是浮点数，计算结果就是浮点数。幂运算符用于计算一个数值的幂次方。例如 3**2 的计算结果是 9，4.3**2 的计算结果是 18.49。

下面的代码测试浮点数进行/(整除)、//(取整)、%(取余运算符)和**（幂运算符）。

【示例 2-23】演示整除、取整、取余、幂运算

```
#测试整除、取整、取余、幂运算
print(9.0/2)      #整除输出结果:4.5
print(9.0//2)     #取整输出结果:4.0
print(9.0%2)      #取余输出结果:1.0
print(4.3**2)     #幂次输出结果:18.49
```

执行结果如图 2-23 所示：

图 2-23 示例 2-23 执行结果

2.4 运算符

Python 的运算符号包括算术运算符、关系运算符和逻辑运算符。表达式是由数字或字符串和运算符组成的式子。表达式通常用于判断语句和循环语句的条件使用，表达式是学习控制语句编写的基础。

2.4.1 算术运算符

算术运算符包括四则运算符、求模运算符和求幂运算符。Python 中的算术运算符详细如表 2-4，示例 2-11。

2.4.2 关系运算符

关系运算符包括等于、不等于、大于、小于、大于等于、小于等于。详细如表 2-6。

表 2-6　关系运算符

运算符	描述	实例
==	等于->比较对象是否不相等	(a==b)返回 False
!=	不等于->比较两个对象是否相等	(a!=b)返回 True
>	大于->返回 x 是否大于 y	(a>b)返回 True
<	小于->返回 x 是否小于 y.所有比较运算符返回 1 表示真,返回 0 表示假.	(a<b)返回 False
>=	大于等于->返回 x 是否大于等于 y.	(a>=b)返回 False
<=	小于等于->返回 x 是否小于等于 y.	(a<=b)返回 True

【示例 2-24】演示比较运算符

```
#测试比较运算符
a,b,c=30,20,10
print("a 大于 b:",(a>b))          #输出结果：True
print("a 大于等于 b:",(a>=b))     #输出结果：True
print("b 小于等于 c：",(b<=c))    #输出结果:False
print("a 等于 c：",(a==c))        #输出结果：False
```

执行结果如图 2-24 所示：

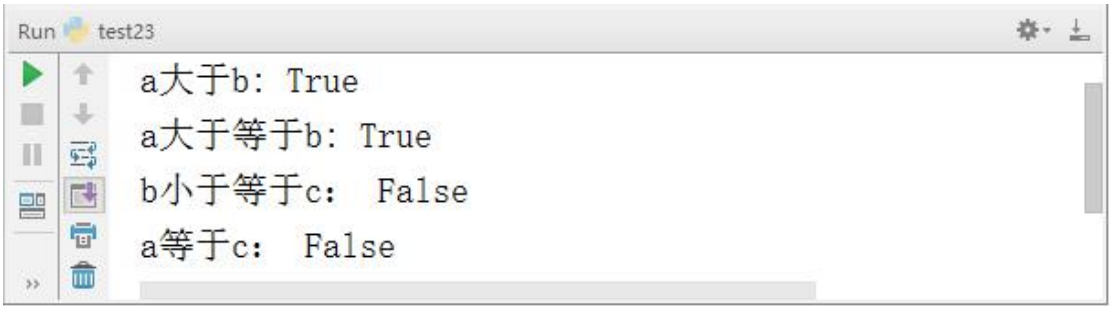


图 2-24 示例 2-24 执行结果

2.4.3 逻辑运算符

Python 语言中包括的逻辑运算符有 and(与)、or(或)和 not(非)，如表 2-7。

表 2-7　逻辑运算符

运算符	逻辑表达式	描述
and	x and y	布尔”与”->如果 x 为 True,y 也为 True,则 x and y 为 True,否则为 False
or	x or y	布尔”或”->如果 x 是非 0 的数,它返回 x 的值,否则返回 y 的值
not	not x	布尔”非”如果 x 为 True,则它返回 False,如果 x 为 False,则它返回 True

【示例 2-25】演示逻辑运算符

```
#测试逻辑运算符
a,b,c=10,20,30
print((a<b) and (b<c))     #and 并且   输出结果是 True
print((a>b) or (b>c))      #or 或者   输出结果是 False
print(not(b<c))            #not 非      输出结果是 False
```

执行结果如图 2-25 所示：

图 2-25 示例 2-25 执行结果

2.4.4 赋值运算符

表 2-8　赋值运算符

运算符	描述	实例
=	简单的赋值运算符	C =a+b 将 a+b 的运算结果赋值为 c
+=	加法赋值运算符	C += a 等效于 c=c+a
-=	减法赋值运算符	C -= a 等效于 c=c-a
*=	乘法赋值运算符	C *= a 等效于 c=c*a
/=	除法赋值运算符	C /= a 等效于 c=c/a
%=	取模赋值运算符	C %= a 等效于 c=c%a
**=	幂赋值运算符	C **= a 等效于 c=c**a
//=	取整除赋值运算符	C //= a 等效于 c=c//a

【示例 2-26】演示赋值运算符

```
#加法赋值运算符
a=3
b=4
a+=b
print('a:',a,"\tb:",b)
#乘法赋值运算符
a=3
a*=b
print('a:',a,"\tb:",b)
#除法赋值运算符
a=10
b=5
a/=b
print('a:',a,"\tb:",b)
#取模赋值运算符
```

```
a=9
b=4
a%=b
print('a:',a,"\tb:",b)
#测试幂赋值运算符
a=2
b=3
a**=b
print('a:',a,"\tb:",b)
#取整除赋值运算符
a=8
b=2
a//=b
print('a:',a,"\tb:",b)
```

执行结果如图 2-26 所示：

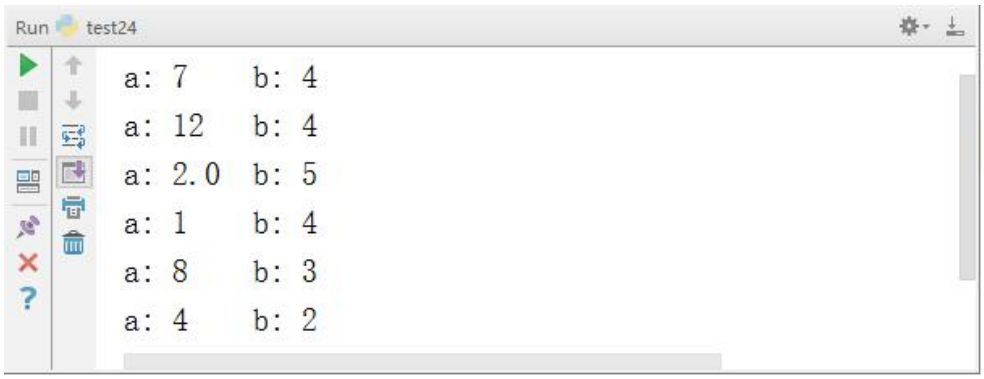


图 2-26 示例 2-26 执行结果

2.4.5 位运算符

按位运算符是把数字看作二进制来进行计算的。Python 中的按位运算法则如表 2-9 所示。

表 2-9 位运算符

运算符	描述	实例
&	按位与运算符:参与运算的两个值,如果两个相应位都为 1,则该位的结果为 1,否则 0	(a&b)输出结果 12,二进制解释:0000 1100
|	按位或运算符:只要对应的二个二进位有一个为 1 时,结果为就为 1.	(a|b)输出结果 61,二进制解释:0011 1101
^	按位异或运算符:当两对应的二进制位相异时,结果为 1	(a^b)输出结果 49,二进制解释:0011 0001
~	按位取反运算符:对数据的每个二进制取反,既把 1 变为 0,把 0 变为 1,~x 类似于-x-1	(~a)输出结果为-61,二进制解释:1100 0011,在一个有符号二进制数的补码形式.

<<	左移动运算符:运算数的每个二进位全部左移若干位,由"<<"右边的数据指定移动的位数,高位丢弃,低位补 0	a<<2 输出救过 240,二进制解释:1111 0000
>>	右移动运算符:把">>"左边运算数的每个二进制位全部右移若干位,">>"右边的数指定移动的位数	a>>2 输出结果 15,二进制解释:0000 1111

【示例 2-27】演示位运算符

```
a=6                  #0000 0110
b=3                  #0000 0011
c=a&b                #0000 0010
print('按位与结果 c 的值为',c) #2
c=a|b                #0000 0111
print('按位或结果 c 的值为',c) #7
c=a^b                #0000 0101
print('按位异或结果 c 的值为',c)#5
c=~a                  #类似于-a-1
print('按位取反结果 c 的值为',c)#-7
c=a<<2                        #00011000
print('左移 2 位结果 c 的值为',c) #24
c=a>>2                        #00000001
print('右移 2 位结果 c 的值为',c)   #1
```

执行结果如图 2-27 所示：

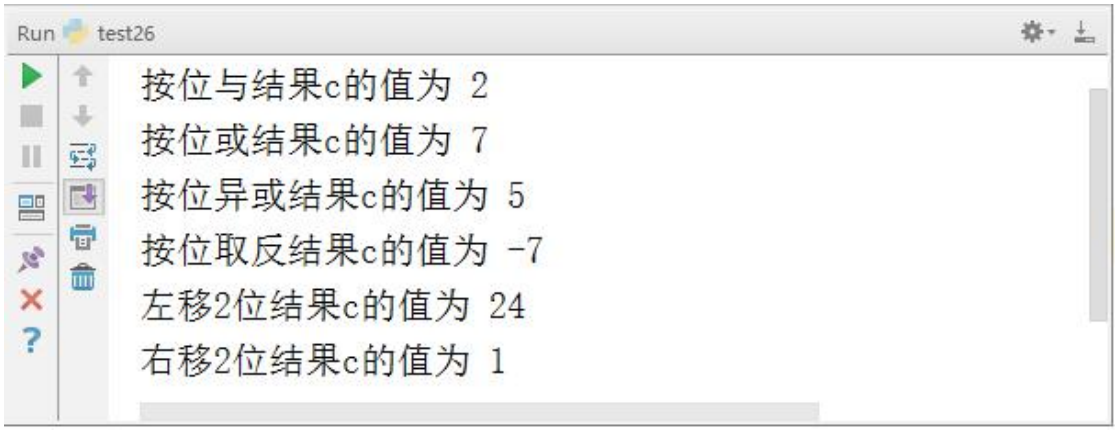


图 2-27 示例 2-27 执行结果

2.2.6 成员运算符

除了以上的一些运算符之外，Python 还支持成员运算符，测试实例中包含了一系列的成员，包括字符串、列表或元组。

表 2-10 成员运算符

运算符	描述	实例
in	如果在指定的序列中找到值返回 True，否则返回 False。	判断 x 是否在 y 序列中,如果 x 在 y 序列中返回 True 反之 False
not in	如果在指定的序列中没有找到值返回 True，否则返回 False。	判断 x 是否在 y 序列中,如果 x 在 y 序列中返回 False 反之 True

【示例 2-28】演示成员运算符

```
a=10
b=20
list=[1,2,3,4,5,20]
print('a 在列表中',(a in list)) #False
print('b 在列表中',(b in list)) #True
print('a 不在列表中',(a not in list)) #True
print('b 不在列表中',(b not in list)) #False
```

执行结果如图 2-28 所示：

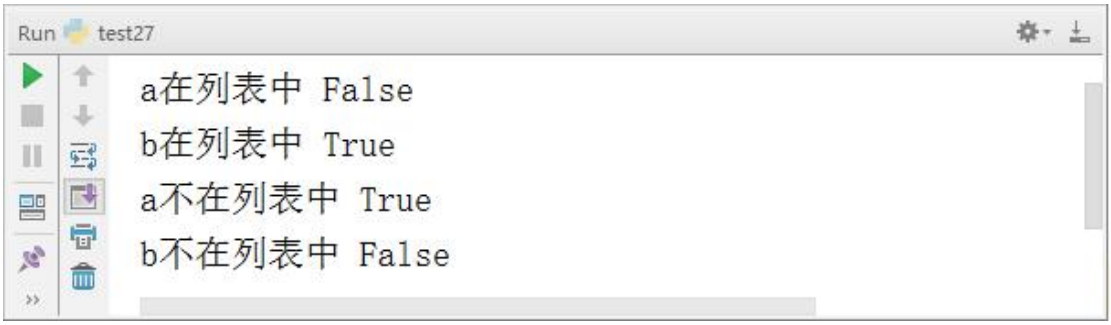


图 2-28 示例 2-28 执行结果

2.2.7 身份运算符

身份运算符用于比较两个对象的存储单元。

表 2-11 身份运算符

运算符	描述	实例
is	is 是判断两个标识符是不是引用自一个对象	x is y，类 id(x) == id(y)，如果引用的是同一个对象则返回 True，否则返回 False
is not	is not 是判断两个标识符是不是引用自不同对象	x is y，类 id(a) != id(b)，如果引用的不是同一个对象则返回 True，否则返回 False

【示例 2-29】演示身份运算符

```
#测试身份运算符
```

```
a=20
b=20
c=30
print("a 和 b 是同一个对象",a is b)     #执行结果:True
print("a 和 c 是同一个对象",a is c)     #执行结果 False
print("a 和 c 不是同一个对象",a is not c)     #执行结果 True
```

执行结果如图 2-29 所示：

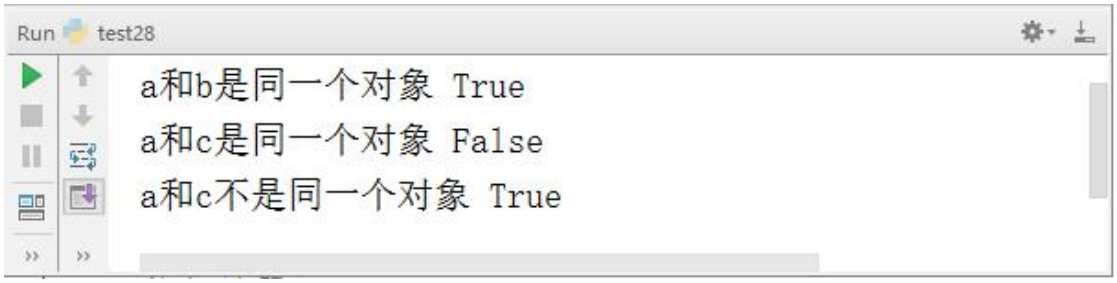


图 2-29 示例 2-29 执行结果

注意

- id()函数用于获取对象内存地址。
- is 用于判断两个变量引用对象是否为同一个， == 用于判断引用变量值是否相等。

2.2.8 运算符的优先级

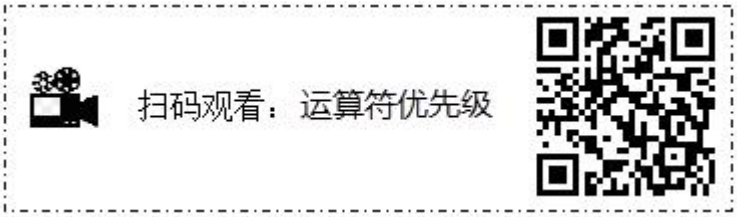


表 2-12 运算符的优先级列出了从最高到最低优先级的所有运算符。

表 2-12 运算符优先级

运算符	描述
**	指数(最高优先级)
~ + -	按位翻转,一元加号减号
* / % //	乘、除、取模和整除
+ -	加法减法
>> <<	右移、左移运算符
&	位,’and’
^ |	位运算符
<= >=	比较运算符
== !=	等于运算符
= %= /= //= -= += *= **=	赋值运算符
is is not	身份运算符
in not in	成员运算符
not or and	逻辑运算符

【示例 2-30】演示运算符优先级

```
#测试运算符优先级
a,b,c,d=20,10,15,5
e=(a+b)*c/d              #30*15/5
print('(a+b)*c/d 的执行结果:',e)
e=(a+b)*(c/d)          #30*(15/5)
print('(a+b)*(c/d)的执行结果：',e)
e=a+(b*c)/d              #20+150/5
print('a+(b*c)/d 的执行结果：',e)
```

执行结果如图 2-30 所示：

图 2-30 示例 2-30 执行结果

注意

- 建议使用小括号组织表达式，不需要记住表达式的运算优先级。

2.5 字符串

2.5.1 字符串创建

字符串是 Python 语言中另外一个重要数据类型，在 Python 语言中，字符串可以使用双引号（“”）或者单引号（‘’）将值括起来。

【示例 2-31】字符串创建

```
#创建字符串
s1='hello world'
s2="I love Python"
print(s1)
print(s2)
```

执行结果如图 2-31 所示：

图 2-31 示例 2-31 执行结果

使用三个单引号或者三个双引号括起来的文本会成为多行注释，其实如果将这样的字符串使用 print 函数输出，或赋给一个变量，将会成为一个长字符串。在长字符串中会保留原始的格式。

【示例 2-32】创建长字符串

```
#创建长字符串
s='''
  I
    Love
        Python
  '''
print(s)
```

执行结果如图 2-32 所示：

图 2-32 示例 2-32 执行结果

2.5.2 转义字符

使用“\+特殊字符”，实现某些难以用字符表示的效果。比如：换行等。常见的转义字符如表 2-13 所示。

表 2-13 转义字符

转义字符	描述
\(在行尾时)	续行符
\\	反斜杠符号
\’	单引号
\″	双引号
\b	退格(Backspace)
\n	换行
\t	横向制表符
\r	回车

【示例 2-33】转义字符

```
#测试转义字符
a = 'I\nLove\nPython' #\n 换行符
print(a)
b="aa\tbb\tcc"       #\t 制表符
print(b)
c='Let\'s go'       #单引号中同时包含单引号使用\'进行转义
print(c)
```

执行结果如图 2-33 所示：

图 2-33 示例 2-33 执行结果

2.5.3 字符串拼接

在输出字符串时，有时字符串会很长，在这种情况下，可以将字符串写成多个部分，然后拼接到一起。如果要连接字符串可以使用（+），也就是字符串的加法运算。还可以将多个字面字符串直接放到一起实现拼接。例如：'Hello"World'。

【示例 2-34】字符串拼接

```
#字符串拼接
a='Hello'
b='World'
c=a+b     #使用加号进行字符串拼接
print(c)
c='Hello''World'    #将多个字面字符串直接放到一起实现拼接
print(c)
```

执行结果如图 2-34 所示：

图 2-34 示例 2-34 执行结果

2.5.4 字符串复制

Python 中使用*可以实现字符串复制。

【示例 2-35】字符串复制

```
#字符串复制
a = 'Sxt'*3
print(a)
```

执行结果如图 2-35 所示：

图 2-35 示例 2-35 执行结果

2.5.5 不换行打印

前面调用 print 时，会自动打印一个换行符。有时，不想换行，不想自动添加换行符。可以通过参数 end = “任意字符串”。

【示例 2-36】使用 print 不换行打印

```
#print()不换行输出
print("sxt",end=' ')
print("sxt",end='##')
print("sxt")
```

执行结果如图 2-36 所示：

图 2-36 示例 2-36 执行结果

2.5.6 字符串的下标和切片

所有序列类型都可以进行某些特定的操作，这些操作包括：索引、切片以及检查某个元素是否属于序列的成员，除此之外，Python 还有计算序列长度、找出最大元素和最小元素的内建函数。

1) 下标即索引

字符串的本质就是字符序列，可以通过在字符串后面添加[]，在[]里面指定偏移量，可以提取该位置的单个字符。

正向搜索：

最左侧第一个字符，偏移量是 0，第二个偏移量是 1，以此类推。直到 len(str)-1 为止。

反向搜索：

最右侧第一个字符，偏移量是-1，倒数第二个偏移量是-2，以此类推，直到-len(str)为止。

【示例 2-37】字符串索引获取字符

```
s='Hello Python'
print('获取 s 的第 1 个字符:',s[0])      #获取 s 的第 1 个字符，运行结果是 H
print('获取 s 的第 5 个字符:',s[4])      #获取 s 的第 5 个字符，运行结果是 o
print('获取 s 的最后一个字符:',s[-1])   #获取 s 的最后一个字符,运行结果是 n
```

执行结果如图 2-37 所示：

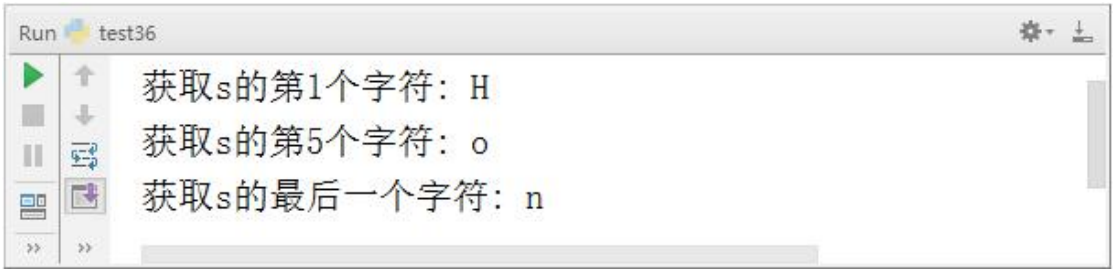


图 2-37 示例 2-37 执行结果

2）字符串的截取

字符串的截取是实际应用中经常使用的技术，被截取的部分称为“子串”。Python 可以通过切片获取子串。切片与使用索引来访问单个元素类似，语法格式如下，其中 string 表示需要取子串的源字符串变量。

```
string[start:end:step]
```

1） start、end、step 三个参数为正数时表示从 string 的第 start 个索引位置开始到第 end 个索引之间截取子串（包括 start 不包括 end），截取的步长是 step。典型操作如表 2-14 所示。

表 2-14 切片正数参数

操作和说明	示例	结果
[:] 提取整个字符串	“abcdef”[:]	“abcdef”
[start:]从 start 索引开始到结尾	“abcdef”[2:]	“cdef”
[:end]从头开始知道 end-1	“abcdef”[:2]	“ab”
[start:end]从 start 到 end-1	“abcdef”[2:4]	“cd”
[start:end:step]从 start 提取到 end-1，步长是 step	“abcdef”[1:5:2]	“bd”

【示例 2-38】字符串截取子串（参数为正数）

```
s='abcdef'
print(s[:])       #start 和 end 都省略，截取整个字符串   输出结果：abcdef
print(s[2:])      #end 省略，从 start 开始到结束         输出结果:cdef
print(s[:2])      #start 省略 从 0 开始到 end-1          输出结果:ab
print(s[2:4])     #从 start 开始到 end-1                 输出结果:cd
print(s[1:5:2])   #从 start 提取到 end-1，步长是 step    输出结果:bd
```

执行结果如图 2-38 所示：

图 2-38 示例 2-38 执行结果

2） start、end、step 三个参数为负数时表示从 string 的倒数第 start 个索引位置开始到倒数第 end 个索引之间截取子串（包括 start 不包括 end），截取的步长是 step，最后一个元素的索引为-1。典型操作如表 2-15 所示。

表 2-15 切片负数参数

示例	说明	结果
"abcdefghijklmnopqrstuvwxyz"[-3:]	倒数三个开始到结束	“xyz”
"abcdefghijklmnopqrstuvwxyz"[-8:-3]	倒数第八个到倒数第三个(包头不包尾)	'stuvw'
"abcdefghijklmnopqrstuvwxyz"[::-1]	步长为负，从右到左反向提取	'zyxwvutsrqponmlkjihgfedcba'

【示例 2-39】字符串截取子串（参数为负数）

```
s='abcdefghijklmnopqrstuvwxyz'
print(s[-3:])   #从倒数第三个开始到结束       输出结果 xyz
print(s[-8:-3])#从倒数第八个开始到倒数第四个  输出结果 stuvw
print(s[::-1]) #步长为-1，从右向左截取          输出 zyxwvutsrqponmlkjihgfedcba
```

执行结果如图 2-39 所示：

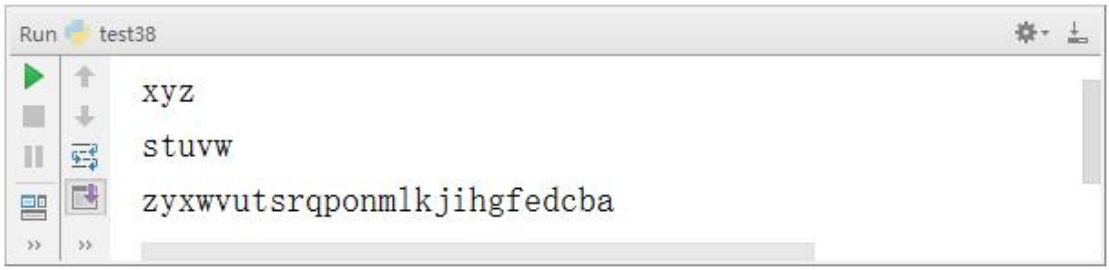
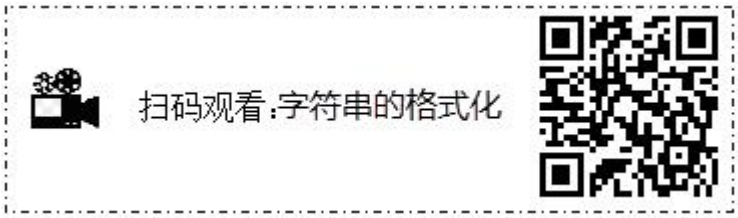


图 2-39 示例 2-39 执行结果

2.5.7 字符串的格式化

1）format 基本用法



Python2.6 开始，新增了一种格式化字符串的函数 str.format()，它增强了字符串格式化的功能。字符串格式化参数并不是用百分号(%)表示，而是用一对大括号({})，而且支持按顺序指定格式化参数和关键字格式化参数。例如，下面的代码通过 format 方法按顺序为格式化字符串指定了参数值。

```
print("{0} {1} {2}".format(11,22,33))              #运行结果:11  22  33
```

可以看到，上面的代码在字符串中指定了三对空的大括号，这代表三个格式化参数，不需要指定数据类型，可以向其传递 Python 语言支持的任何值。通过 format 方法传入三个值(11,22,33)，这三个值会按顺序替换格式化字符串中的三对空的大括号。

命名格式化参数是指在一对大括号中指定一个名称，然后调用 format 方法时也要指定这个名称。print("{a} {b} {c}".format(a=1,b=2,c=3)) #运行结果是:1 2 3

上面的代码在三对大括号中分别添加了“a”“b”“c”。通过 format 方法指定了这三个关键字参数的值。可以看到，并没有按顺序指定关键字参数的值。这也是使用关键字参数的好处。只要名字正确，format 参数的顺序可以任意指定。当然顺序方式和关键字参数方式可以混合使用，而且还可以指定顺序方式中格式化参数从 format 方法提取参数值的顺序，甚至可以取 format 方法参数值的一部分。

【示例 2-40】使用 format 实现字符串格式化

```
s="姓名:{0},年龄:{1},薪资:{2}"
print(s.format('张三',23,13000))
c = "名字是{name}，年龄是{age}"
print(c.format(age=19,name='李四'))
```

执行结果如图 2-40 所示：

图 2-40 示例 2-40 执行结果

2）format 填充对齐

format 方法还可以控制值的左、中、右对齐，“^”、“<”、“>”分别是居中、左对

齐、右对齐，后面带宽度。“:”号后面带填充的字符，只能是一个字符，不指定的话默认是用空格填充。

【示例 2-41】使用 format 实现填充对齐

```
print("{:*>8}".format("245"))
print("{:#^20}".format("Python"))
```

执行结果如图 2-41 所示：

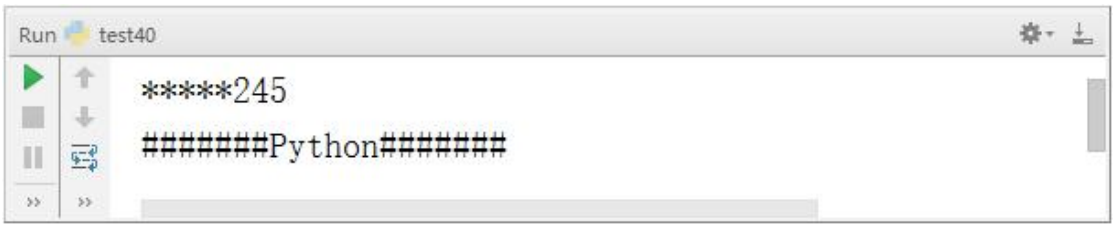


图 2-41 示例 2-41 执行结果

3） 数字格式化

format 方法还支持很多其他的控制符，例如，可以将整数按浮点数输出，详细如表 2-16。

表 2-16 数字格式化类型符

数字	格式	输出	描述
3.1415926	{:.2f}	3.14	保留小数点后两位
3.1415926	{:+.2f}	3.14	带符号保留小数点后两位
2.71828	{:.0f}	3	不带小数
5	{:0>2d}	05	数字补零（填充左边，宽度为 2）
5	{:x<4d}	5xxx	数字补 x（填充右边，宽度为 4）
10	{:x<4d}	10xx	数字补 x（填充右边，宽度为 4）
1000000	{:,}	1,000,000	以逗号分隔的数字格式
0.25	{:.2%}	25.00%	百分比格式
1000000000	{:.2e}	1.00E+09	指数记法
13	{:10d}	13	右对齐（默认，宽度为 10）
13	{:<10d}	13	左对齐（宽度为 10）
13	{:^10d}	13	中间对齐（宽度为 10）

【示例 2-42】使用 format 实现数字格式化

```
a = "我是{0}，我的存款有{1:.2f}"
print(a.format("张三",3888.234342))
```

执行结果如图 2-42 所示：

图 2-42 示例 2-42 执行结果

2.5.8 字符串的常用方法

1）find()

find 方法可以在一个较长的字符串中查找子串，它返回子串的第一个字符在大字符串中出现的位置。如果没有找到则返回-1。find 方法还可以指定开始和结束查找的位置。

【示例 2-43】字符串 find 方法的使用

```
a='Hello Python'
b=a.find('P')
print('字符 P 在字符串 Hello Python 中的位置{0}'.format(b))
b=a.find('Hello')
print('字符 P 在字符串 Hello Python 中的位置{0}'.format(b))
#指定开始查找子串位置
b=a.find('o',6)
print('从索引 6 开始查找字符 o 在字符串 Hello Python 中的位置{0}'.format(b))
#指定开始和结束查找子串位置
b=a.find('o',2,6)
print('从索引 2 开始到索引 6 查找字符 o 在字符串 Hello Python 中的位置{0}'.format(b))
```

执行结果如图 2-43 所示：

图 2-43 示例 2-43 执行结果

2）index()

跟 find()方法一样，区别是如果查找的子串不在原字符串中 index 会报异常，而 find 没有找到则返回-1。

【示例 2-44】字符串 index 方法的使用

```
a='Hello Python'
b=a.index('l')        #l 第一次出现的索引     输出结果:2
print(b)
b=a.find('s')         #find 方法未检索到返回-1     输出结果:-1
print(b)
b=a.index('s')        #index 方法未检索到抛出异常
print(b)
```

执行结果如图 2-44 所示：

图 2-44 示例 2-44 执行结果

3） count()

统计子串在原字符串中出现的次数。

【示例 2-45】字符串 count 方法的使用

```
a='Hello Python'
b=a.count('o')    #统计 o 字符在字符串中的个数     输出结果:2
print('统计字符 o 出现的次数：',b)
#指定开始索引统计
c=a.count('o',6,len(a))
print('从索引 6 开始统计字符 o 出现的次数：',c)
```

执行结果如图 2-45 所示：

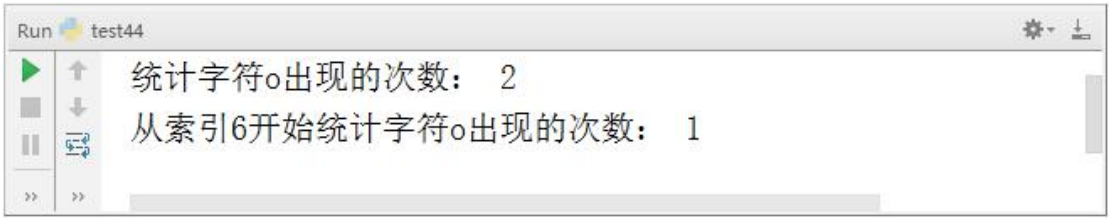


图 2-45 示例 2-45 执行结果

4） replace()

字符串是“不可改变”的，通过[]可以获取字符串指定位置的字符，但不能改变字符串。但是，确实有时候需要替换某些字符。这时，可以使用 replace 方法来实现。

【示例 2-46】字符串 replace 方法的使用

```
a='Hello Python'
b=a.replace('o','*')          #将 Hello Python 字符串中的 o 替换为* 默认全部替换
c=a.replace('o','*',1)          #将 Hello Python 字符串中的 o 替换为* 替换 1 次
print(a)
print(b)
print(c)
```

执行结果如图 2-46 所示：

图 2-46　示例 2-46 执行结果

从运行结果可以看到整个过程中，实际上是创建了新的字符串对象，并指向了变量 a，而不是修改了以前的字符串。

5） split()分割和 join()合并

split 方法通过分隔符将一个字符串拆分成一个序列。如果 split 方法不指定任何参数，那么 split 方法会把所有空格(空格符、制表符、换行符)作为分隔符。

join 方法用于连接序列中的元素，是 split 方法的逆方法。

【示例 2-47】字符串 split 方法和 join 方法的使用

```
#split 方法测试
a='This is my first Progame'
b=a.split(" ",2)
print(b)
a = "to be or not to be"
b=a.split()
print(b)
#join 方法测试
a = ['sxt','sxt100','sxt200']
```

```
print('*'.join(a))
```

执行结果如图 2-47 所示：

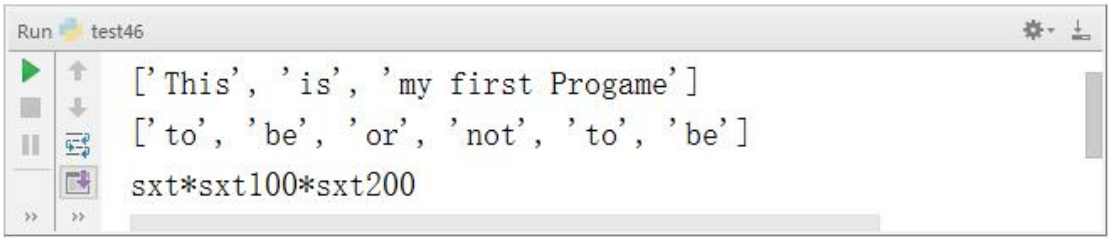


图 2-47 示例 2-47 执行结果

注意：

- 使用字符串拼接符+，会生成新的字符串对象，因此不推荐使用+来拼接字符串。推荐使用 join 函数，因为 join 函数在拼接字符串之前会计算所有字符串的长度，然后逐一拷贝，仅新建一次对象。

6） capitalize()、lower()、upper()、title()、swapcase()

lower 方法和 upper 方法分别用于将字符串的所有字母字符转换为小写和大写。capitalize 方法将字符串的第一个字符转换成大写。title 方法将字符串的每个单词首字母大写。swapcase 方法将字母大小写转换。

【示例 2-48】字符串 capitalize()、lower()、upper()、title()、swapcase()方法的使用

```
a='Hello python'
#将 Hello python 转换为大写
print('upper 全部大写:',a.upper())
#将 Hello python 转换为小写
print('lower 全部小写:',a.lower())
#将 Hello python 首字母大写
print('capitalize 首字母大写:',a.capitalize())
#将 Hello python  每个单词首字母转换为大写
print('title 每个单词首字母大写:',a.title())
#将 Hello python 字母大小写转换
print('swapcase 字母大小写转换:',a.swapcase())
```

执行结果如图 2-48 所示：

图 2-48 示例 2-48 执行结果

7） startswith()、endswith()

startwith 检索字符串是否以指定字符串开始，如果是则返回 True，否则返回 Fasle。endwith 检索字符串是否以指定字符串结尾，如果是则返回 True，否则返回 False。

【示例 2-49】字符串 startswith、endswith 方法的使用

```
a='hello python'
print('是否以字符 h 开头:',a.startswith('h'))
print('是否以字符 h 结尾:',a.endswith('h'))
```

执行结果如图 2-49 所示：

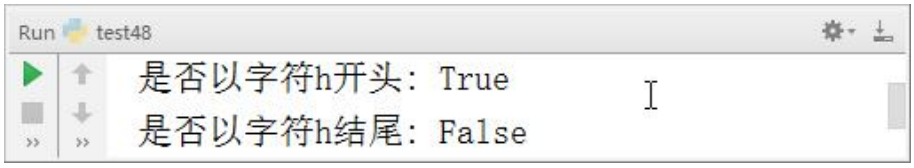


图 2-49 示例 2-49 执行结果

8） center()、ljust()、rjust()

center 方法用于将一个字符串在一定宽度的区域居中显示，并在字符串的两侧填充指定的字符，默认填充空格。前面将的 format 方法和居中符号(^)，其实完全可以用 format 方法代替 center 方法实现同样的效果，只是使用 center 方法更简单，更直接一些。

ljust()返回一个原字符串左对齐，并使用空格填充至长度 width 的新字符串,默认是空格填充。

rjust()返回一个原字符串右对齐，并使用空格填充至长度 width 的新字符串。

【示例 2-50】字符串 center、ljust、rjust 方法的使用

```
a='SXT'
print(a.center(10,"*"))
print(a.center(10))
print(a.ljust(10,"*"))
print(a.rjust(10,"*"))
```

执行结果如图 2-50 所示：

图 2-50 示例 2-50 执行结果

9） strip()、lstrip()、rstrip()

lstrip 方法删除原字符串左边的空白字符，rstrip 方法删除原字符串右边的空白字符，strip 删除原字符串两端的空白字符。

【示例 2-51】字符串 strip、lstrip、rstrip 方法的使用

```
a='      hellopython      '
print(a.strip())
print(a.lstrip())
print(a.rstrip())
```

执行结果如图 2-51 所示：

图 2-51 示例 2-51 执行结果

10） 其他方法

表 2-17 其他方法

方法名	说明
isalnum()	是否为字母或数字
isalpha()	检测字符串是否只由字母组成(含汉字)
isdigit()	检测字符串是否只由数字组成
isspace()	检测是否为空白符
isupper()	是否为大写字母
islower()	是否为小写字母

【示例 2-52】字符串其他方法的使用

```
#是否为字母或数字
print("sxt100".isalnum())
#检测字符串是否只由字母组成(含汉字)
print("sxt 尚学堂".isalpha())
#检测字符串是否只由数字组成
print("234.3".isdigit())
print("2343".isdigit())
#检测是否为空白符
print("sxt\t\n".isspace())
print("\t\n".isspace())
#是否为大写字母
print("A".isupper())
print('Ab'.isupper())
#是否为小写字母
print("a".islower())
print("aB".islower())
```

执行结果如图 2-52 所示：

图 2-52 示例 2-52 执行结果

2.5.9 可变字符串

在 Python 中，字符串属于不可变对象，不支持原地修改，如果需要修改其中的

值，智能创建新的字符串对象。但是，经常需要原地修改字符串，可以使用 io.StringIO 对象或 array 模块。

【示例 2-53】可变字符串

```
import io
s = "hello,bjsxt"
sio = io.StringIO(s)
print(sio)
print('原字符串：',sio.getvalue())
sio.seek(7)
sio.write("g")
print('修改后字符串：',sio.getvalue())
```

执行结果如图 2-53 所示：

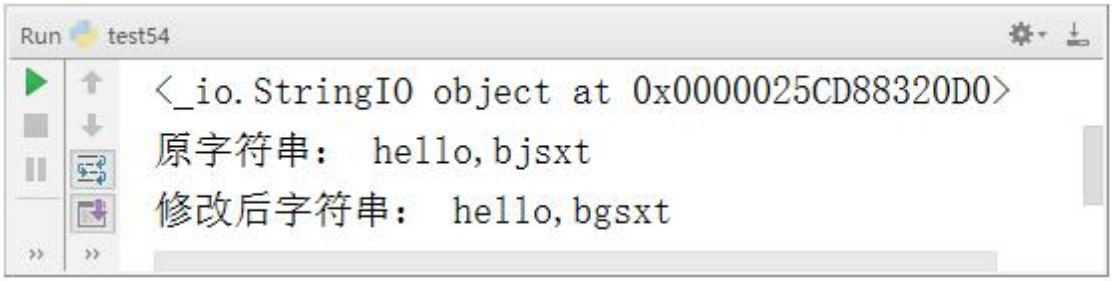
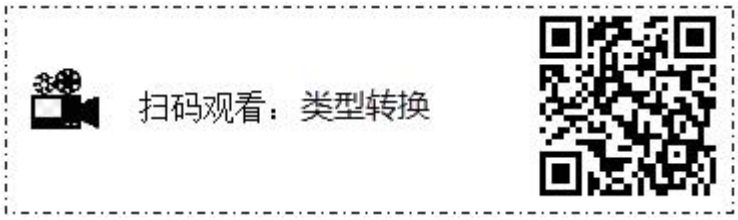


图 2-53 示例 2-53 执行结果

2.6 类型转换



与 C++、Java 等高级程序设计语言一样，Python 语言同样也支持数据类型转换。这里列举了常见的 Python 类型转换，如表 2-18 所示。

表 2-18 数据类型转换

类型转换	
int(x [,base])	将 x 转换为一个整数
long(x [,base])	将 x 转换为一个长整数
float(x)	将 x 转换到一个浮点数
complex(real[,imag])	创建一个复数
str(x)	将对象 x 转换为字符串
repr(x)	将对象 x 转换为表达式字符串

类型转换	
eval(str)	用来计算在字符串中的有效 Python 表达式,并返回一个对象
Complex(A)	将参数转换为复数型
tuple(s)	将序列 s 转换为一个元组
list(s)	将序列 s 转换为一个列表
set(s)	转换为可变集合
dict(d)	创建一个字典。d 必须是一个序列 (key,value)元组
frozenset(s)	转换为不可变集合
chr(x)	将一个整数转换为一个字符
unichr(x)	将一个整数转换为 Unicode 字符
ord(x)	将一个字符转换为它的整数值
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串

【示例 2-54】数据类型转换

```
#类型转换

#转换为 int
print('int()默认情况下为：', int())
print('str 字符型转换为 int：', int('010'))
print('float 浮点型转换为 int：', int(234.23))
#十进制数 10，对应的 2 进制，8 进制，10 进制，16 进制分别是：1010,12,10,0xa
print('int(\'0xa\', 16) = ', int('0xa', 16))
print('int(\'10\', 10) = ', int('10', 10))
print('int(\'12\', 8) = ', int('12', 8))
print('int(\'1010\', 2) = ', int('1010', 2))

#转换为 float
print('float()默认情况下为：', float())
print('str 字符型转换为 float：', float('123.01'))
print('int 浮点型转换为 float：', float(32))

#转换为 complex
print('创建一个复数(实部+虚部)：', complex(12, 43))
print('创建一个复数(实部+虚部)：', complex(12))
```

```
#转换为 str 字符串
print('str()默认情况下为：', str())
print('float 字符型转换为 str：', str(232.33))
print('int 浮点型转换为 str：', str(32))
lists = ['a', 'b', 'e', 'c', 'd', 'a']
print('列表 list 转换为 str:', ''.join(lists))

#转换为 list
strs = 'hongten'
print('序列 strs 转换为 list:', list(strs))

#转换为 tuple
print('列表 list 转换为 tuple:', tuple(lists))

#字符和整数之间的转换
print('整数转换为字符 chr:', chr(67))
print('字符 chr 转换为整数:', ord('C'))

print('整数转 16 进制数:', hex(12))
print('整数转 8 进制数:', oct(12))
```

执行结果如图 2-54 所示：

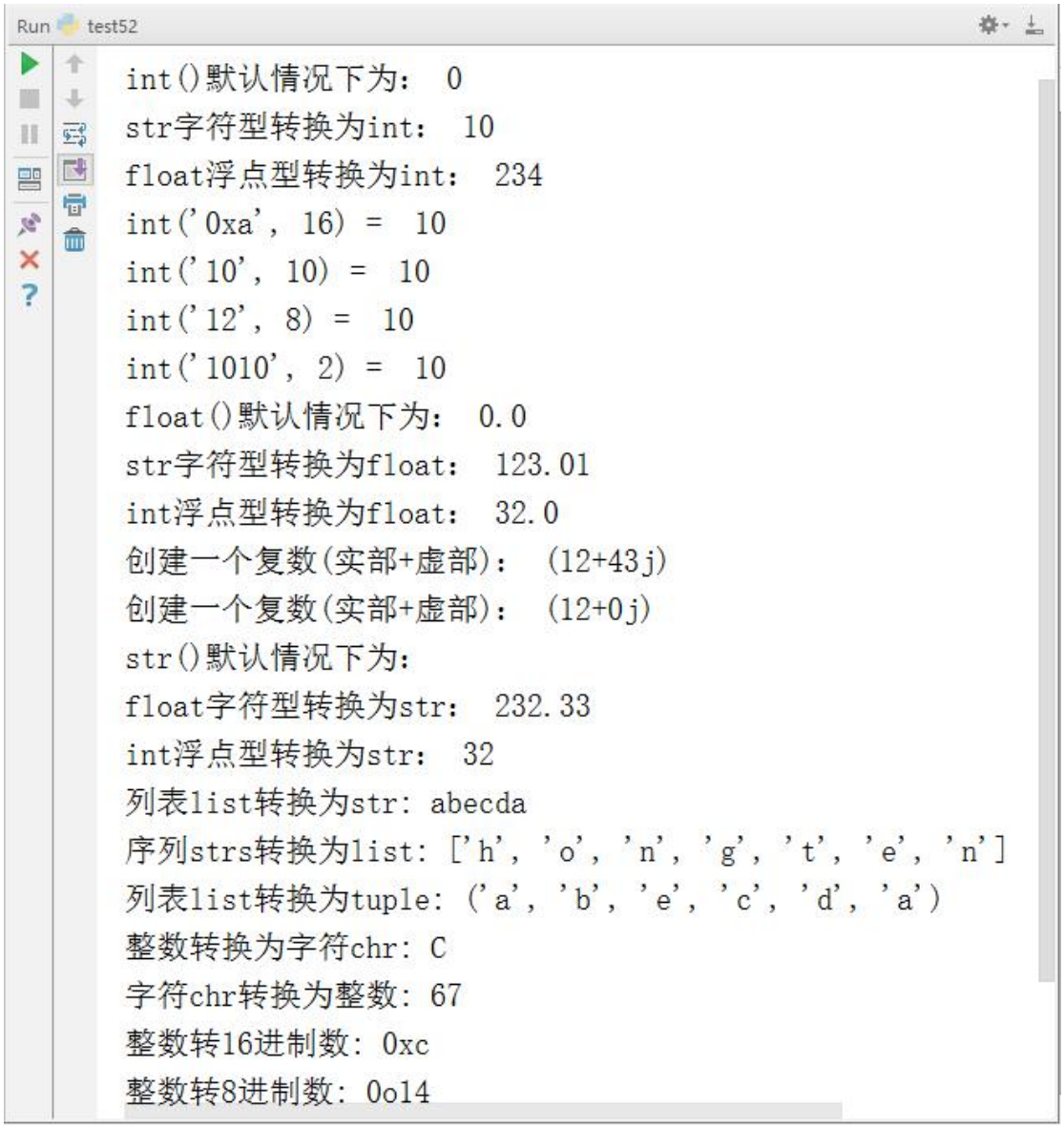


图 2-54 示例 2-54 执行结果

习题

一、选择题

1. 下列哪个语句在 Python 中是非法的？（ ）

A. x = y = z = 1

B. x = (y = z + 1)

C. x, y = y, x

D. x += y

2. 关于 Python 内存管理，下列说法错误的是（ ）

A. 变量不必事先声明

B. 变量无须先创建和赋值而直接使用

C. 变量无须指定类型

D. 可以使用 del 释放资源

3. 在 Python3 中执行如下语句后得到的结果是？ （）

```
>>> world="world"
>>> print "hello"+ world
```

A. helloworld

B. "hello"world

C. hello world

D. 语法错误

4. 下面哪个不是 Python 合法的标识符（ ）

A. int32 B. 40XL C. self D. name

5. Python 不支持的数据类型是 （ ）

A. char B. int C. float D. list

二、简答题

1. 变量赋值

   1)赋值语句 x,y,z=1,2,3 会在 x、y、z 的值分别是多少？

   2)执行在 z,x,y=y,z,x 后，x、y、z 的值分别是多少？

2. 如何判断一个变量是不是字符串？

3. is 和 == 的区别？

4. Python 有哪些数据类型？

5. 下列语句的执行结果是 False，分析为什么？

```
>>> from math import sqrt
>>> print(sqrt(3)*sqrt(3)==3)
False
```

三、编码题

1. 将下面的数值转成另外三种进制，并使用 print 函数输出转换结果。例如，如果数值是十进制，需要转换成二、八、十六进制；如果是十六进制，需要转换为二、八、十进制。

   1）123456

   2）0xA88A

   3)0b1010101010

2. 使用 python 表示数学式：$\frac{5+10x}{5}-\frac{13(y-1)(a+b)}{x}+9(\frac{5}{x}+\frac{12+x}{y})$。

3. 从控制台输入用户的月薪，进行运算计算出年薪。打印输出用户的年薪。

4. 使用字符串复制，用计算机打印出“爱你一百遍”，打印 100 次。

5. 将”to be or not to be”字符串倒序输出。

6. 将”sxtsxtsxtsxtsxt”字符串中所有的 s 输出。

第三章 流程控制语句

前面学习了 Python 语言的基础知识，这些 Python 代码都是从上到下顺序执行的，从第一条语句一直执行到最后一条语句，然后程序退出。但在实际应用中，程序需要选择执行和重复执行，“选择”和“重复执行”在编程语言中称为条件和循环，条件和循环有一个统一的名称，叫作“流程控制”。本章介绍 Python 中控制语句的编写。

通过阅读本章，你可以：

- 了解 Python 语言中的代码块
- 掌握 if 条件语句的用法
- 掌握嵌套代码块的用法
- 掌握 while 循环的使用方法
- 掌握 for 循环的使用方法
- 了解如何使用嵌套循环
- 掌握循环中的 else 子句的用法

3.1 布尔值和布尔变量

布尔（boolean）类型有两个值：True 和 False，可以将这两个值翻译成“真”和“假”。在 Python 语言中有一些特殊的布尔类型值为 False，例如 False、0、0.0、空值 None、空序列对象（空列表、空元祖、空集合、空字典、空字符串）、空 range 对象、空迭代对象，其他情况，均为 True。

【示例 3-1】使用 bool 函数测试特殊布尔类型

```
print('空字符串的布尔类型的值:',bool(""))
print('空列表布尔类型的值:',bool([]))
print('空元组布尔类型的值:',bool(()))
print('空字典布尔类型的值:',bool({}))
print('None 布尔类型的值:',bool(None))
print('0 布尔类型的值:',bool(0))
```

执行结果如图 3-1 所示：

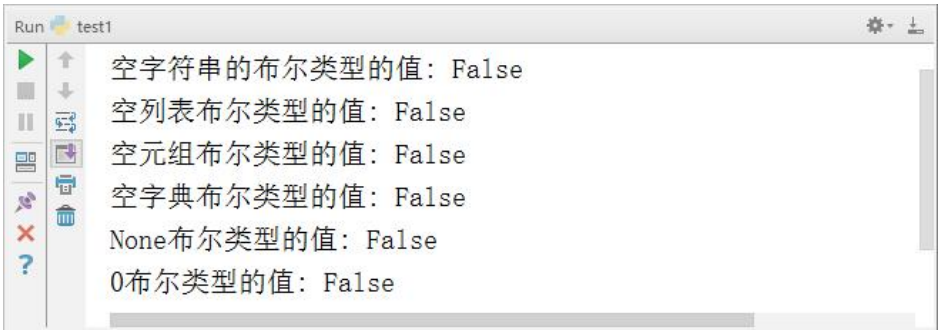


图 3-1 示例 3-1 执行结果

在 Python 语言底层，会将布尔值 True 看作 1，将布尔值 False 看作 0，尽管从表面上看，True 和 1、False 和 0 是完全不同的两个值，但实际上，它们是相同的。

【示例 3-2】测试 True 和 1、False 和 0 是相同的

```
print("True==1 的结果:",True==1)
print("False==0 的结果:",False==0)
print("True+False+20 的计算结果:",True+False+20)
```

执行结果如图 3-2 所示：

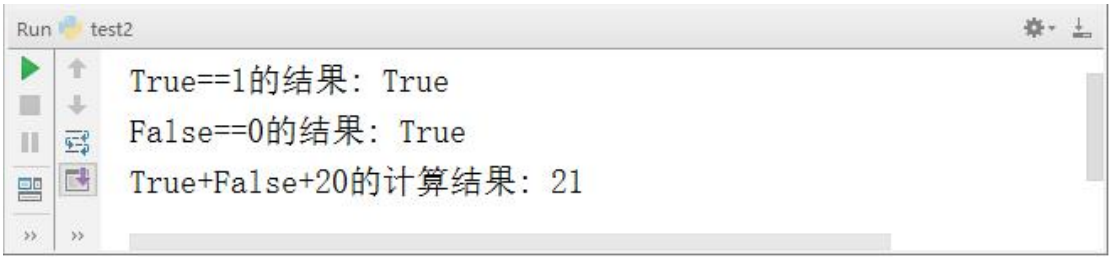


图 3-2 示例 3-2 执行结果

3.2 条件判断语句

完全按照设计好的流程执行程序，几乎是太完美了。然后现实生活更多的时候需要流程选择，因此流程控制至关重要。日常生活中大家都会有意识无意识的用到流程控制，用来判断自己下一步的行动计划。例如，今日行程是去新华书店买书还是去游泳馆游泳呢，需要作出选择。

Python 条件语句是通过一条或多条语句的执行结果(True 或 False)来执行代码块。选择结构通过判断条件是否成立，来决定执行哪个分支。选择结构有多种形式，分为：单分支如图 3-3、双分支如图 3-4、多分支如图 3-5。

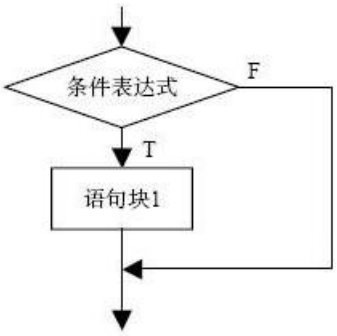


图 3-3 单分支

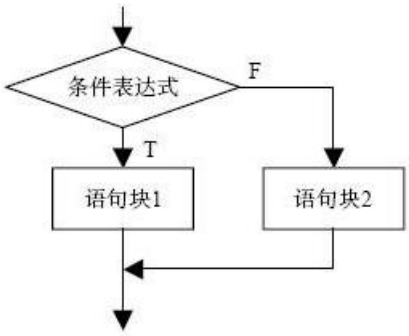


图 3-4 双分支

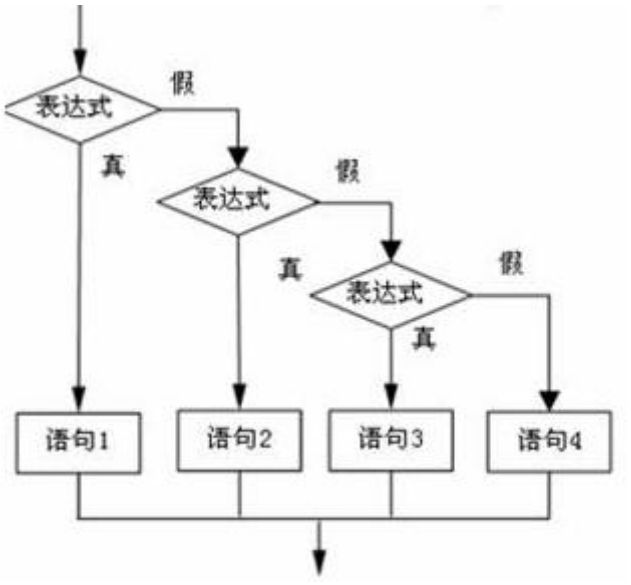


图 3-5 多分支

3.2.1 单分支

if 语句单分支结构的语法形式如下：

```
if 条件表达式:
    语句/语句块
```

其中：条件表达式：可以是逻辑表达式、关系表达式、算术表达式等等；语句/语句块：可以是一条语句，也可以是多条语句。多条语句，缩进必须对齐一致。

【示例 3-3】if 单分支结构

```
num = input("请输入一个整数：")
if int(num)<10:       #获取的 num 是字符串，所以使用 int()函数进行转换
    print('您输入的整数是:',num)
```

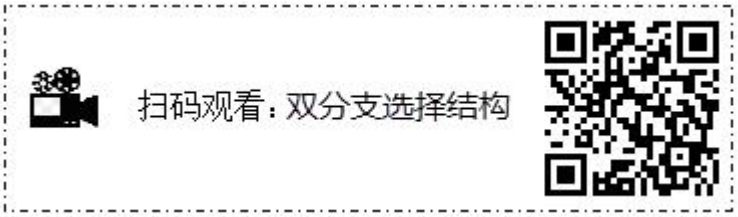
执行结果如图 3-6 所示：

图 3-6 示例 3-3 执行结果

注意：

- 条件表达式中，不能有赋值操作符“=” if 3<c and (c=20):将会包语法错误

3.2.2 双分支选择结构



双分支结构的语法格式如下：

```
if  条件表达式 :
    语句 1/语句块 1
else:
    语句 2/语句块 2
```

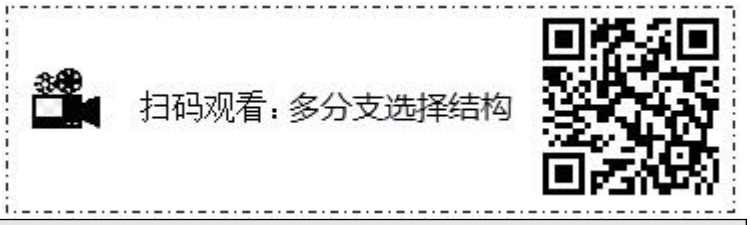
【示例 3-4】if-else 双分子结构

```
num = input("请输入一个整数：")
if int(num)<10:      #获取的 num 是字符串，所以使用 int()函数进行转换
    print('您输入的整数是:',num)
else:
    print('数字过大')
```

执行结果如图 3-7 所示：

图 3-7 示例 3-4 执行结果

3.2.3 多分支选择结构



多分支选择结构的语法格式如下：

```
if 条件表达式 1 :
    语句 1/语句块 1
elif 条件表达式 2:
    语句 2/语句块 2
.
.
.
```

```
elif 条件表达式 n :
    语句 n/语句块 n
[else:
    语句 n+1/语句块 n+1
]
```

注意：

- 计算机行业，描述语法格式时，使用中括号[ ]通常表示可选，非必选，及多分支结构中最后的 else 是可选的。
- 多分支结构，几个分支之间是有逻辑关系的，不能随意颠倒顺序。

【示例 3-5】if-elif 多分支结构

```
score = int(input("请输入分数"))
grade = ''
if score<60 :
    grade = "不及格"
elif  score<80 :
    grade = "及格"
elif  score<90 :
    grade = "良好"
elif  score<=100:
    grade = "优秀"
print("分数是{0},等级是{1}".format(score,grade))
```

执行结果如图 3-8 所示：

图 3-8 示例 3-5 执行结果

【示例 3-6】已知点的坐标(x,y)，判断其所在的象限

```
x = int(input("请输入 x 坐标"))
y = int(input("请输入 y 坐标"))
if(x==0 and y==0):print("原点")
```

```
elif(x==0):print("y 轴")
elif(y==0):print("x 轴")
elif(x>0 and y>0):print("第一象限")
elif(x<0 and y>0):print("第二象限")
elif(x<0 and y<0):print("第三象限")
else:
    print("第四象限")
```

执行结果如图 3-9 所示：

图 3-9 示例 3-6 执行结果

3.2.4 三元条件运算符

Python 提供了三元运算符，三元运算符是条件语句中比较简练的一种赋值方式，用来在某些简单双分支赋值情况。三元条件运算符语法格式如下。

```
A=Y  if  X  else Z
```

其中如果 X 为真，那么就执行 A=Y，如果 X 为假，就执行 A=Z。

【示例 3-7】三元条件运算符

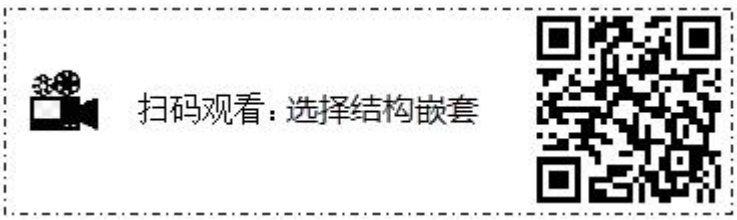
```
num=int(input('请输入一个整数:'))
print(num if num<10 else '整数过大')
```

执行结果如图 3-10 所示：

图 3-10 示例 3-7 执行结果

3.2.5 选择结构嵌套

条件语句可以进行嵌套，也就是说在一个条件代码块中，可以有另外一个条件代码块。嵌套代码块仍然



需要增加缩进。因此使用时一定要注意控制好不同级别代码块的缩进量，因为缩进量决定了代码的从属关系，语法格式如下：

```
if 表达式 1：
    语句块 1
    if 表达式 2：
        语句块 2
    else:
        语句块 3
else:
    if 表达式 4：
        语句块 4
```

【示例 3-8】选择结构嵌套----输入一个分数。分数在 0-100 之间。90 以上是 A，80 以上是 B，70 以上是 C，60 以上是 D，60 以下是 E。

```
score = int(input("请输入一个在 0-100 之间的数字："))
grade = ""
if score>100 or score<0:
    score = int(input("输入错误！请重新输入一个在 0-100 之间的数字："))
else:
    if score>=90:
        grade = "A"
    elif score>=80:
        grade = 'B'
    elif score>=70:
        grade = 'C'
    elif score>=60:
        grade = 'D'
    else:
        grade = 'E'
    print("分数为{0},等级为{1}".format(score,grade))
```

执行结果如图 3-11 所示：

图 3-11 示例 3-8 执行结果

3.3 循环语句

循环结构用来重复执行一条或多条语句。表达这样的逻辑：如果符合条件，则反复执行循环体里的语句。在每次执行完后都会判断一次条件是否为 True，如果为 True 则重复执行循环体里的语句。如图 3-12 所示：

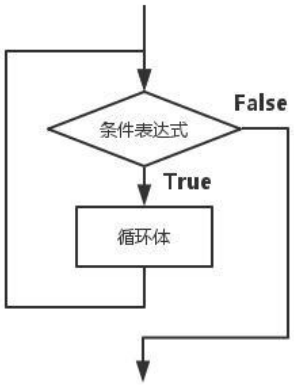


图 3-12 循环流程图

在 Python 语言中，有两类语句可以实现这个循环操作，这就是 while 循环和 for 循环。

3.3.1 while 循环

while 循环的语法格式如下：

```
while  条件表达式：
    循环体语句
```

【示例 3-9】利用 while 循环打印从 0-10 的数字。

```
num = 0
while num<=10:
    print(num)
    num += 1
```

执行结果如图 3-13 所示：

图 3-13 示例 3-9 执行结果

可以看到 while 关键字的后面是条件表达式，最后用冒号（:）结尾，这说明 while 循环也是一个代码块，因此，在 while 循环内部的语句需要使用缩进的写法。

在上面的代码中，首先在 while 循环的前面定义一个变量 num，初始值为 0。然后开始进入 while 循环。在第 1 次执行 while 循环中的语句时，会用 print 函数输出 num 变量的值，然后 num 变量的值加 1，最后 while 循环中的语句第一次执行完毕，然后会重新循环 while 后面的条件，这时 num 变量的值是 2，num<=10 的条件仍然满足，所以 while 循环将继续执行，直到 num=11 时，num<=10 不再满足，所以 while 执行结束，继续执行 while 后面的语句。

【示例 3-10】利用 while 循环，计算 1-100 之间数字的累加和；计算 1-100 之间偶数的累加和，计算 1-100 之间奇数的累加和。

```
num = 1
sum_all = 0              #1-100 所有数的累加和
sum_even = 0             #1-100 偶数的累加和
sum_odd = 0              #1-100 奇数的累加和
while num<=100:
    sum_all += num
    if num%2==0:sum_even += num
    else:sum_odd += num
    num += 1             #迭代，改变条件表达式，使循环趋于结束
print("1-100 所有数的累加和",sum_all)
print("1-100 偶数的累加和",sum_even)
```

```
print("1-100 奇数的累加和",sum_odd)
```

执行结果如图 3-14 所示：

图 3-14 示例 3-10 执行结果

3.3.2 for 循环

for 循环用于遍历一个集合，每次循环，会从集合中取得一个元素，并执行一次代码块，直到集合中所有的元素都获取，for 循环才结束。for 循环的语法格式如下：


```
for  变量   in  可迭代对象：
    循环体语句
```

Python 可以遍历的对象有序列(字符串、列表、元组)、字典、迭代器对象(iterator)、生成器函数文件对象。前面已经学习了字符串，通过循环来遍历字符串。

【示例 3-11】for 循环遍历字符串

```
#for 循环遍历字符串
for x  in  "sxt 尚学堂":
    print(x)
```

执行结果如图 3-15 所示：

图 3-15 示例 3-11 执行结果

for 循环通常与 range()函数一起使用，range()返回一个列表，for 遍历列表中的元素。range()函数的语法格式如下：

```
range(start, end    [,step])
```

其中生成的数值序列从 start 开始到 end 结束（不包含 end）。若没有填写 start，则默认从 0 开始。step 是可选的步长，默认为 1。如下是几种典型示例：

for i in range(10)　　产生序列：0 1 2 3 4 5 6 7 8 9

for i in range(3,10)　　产生序列：3 4 5 6 7 8 9

for i in range(3,10,2)　　产生序列：3　5　7　9

【示例 3-12】利用 for 循环，计算 1-100 之间数字的累加和；计算 1-100 之间偶数的累加和，计算 1-100 之间奇数的累加和。

```
sum_all = 0              #1-100 所有数的累加和
sum_even = 0             #1-100 偶数的累加和
sum_odd = 0              #1-100 奇数的累加和
for num in range(101):
    sum_all += num
    if num%2==0:sum_even += num
    else:sum_odd += num
print("1-100 累加总和{0},奇数和{1},偶数和{2}".format(sum_all,sum_odd,sum_even))
```

执行结果如图 3-16 所示：

图 3-16 示例 3-12 执行结果

3.3.3 循环嵌套

一个循环体内可以嵌入另一个循环，一般称为“嵌套循环”，或者“多重循环”。

【示例 3-13】循环嵌套----打印如下图案

```
0  0  0  0  0
1  1  1  1  1
2  2  2  2  2
3  3  3  3  3
```

4 4 4 4 4

```
for x in range(5):
    for y in range(5):
        print(x,end="\t")
    print()     #仅用于换行
```

执行结果如图 3-17 所示：

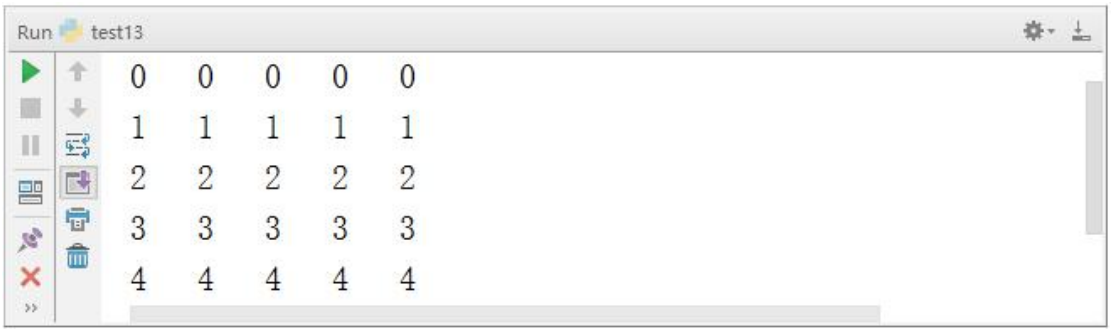


图 3-17 示例 3-13 执行结果

【示例 3-14】利用嵌套循环----打印九九乘法表

```
for m in range(1,10):
    for n in range(1,m+1):
        print("{0}*{1}={2}".format(m,n,(m*n)),end="\t")
    print()
```

执行结果如图 3-18 所示：

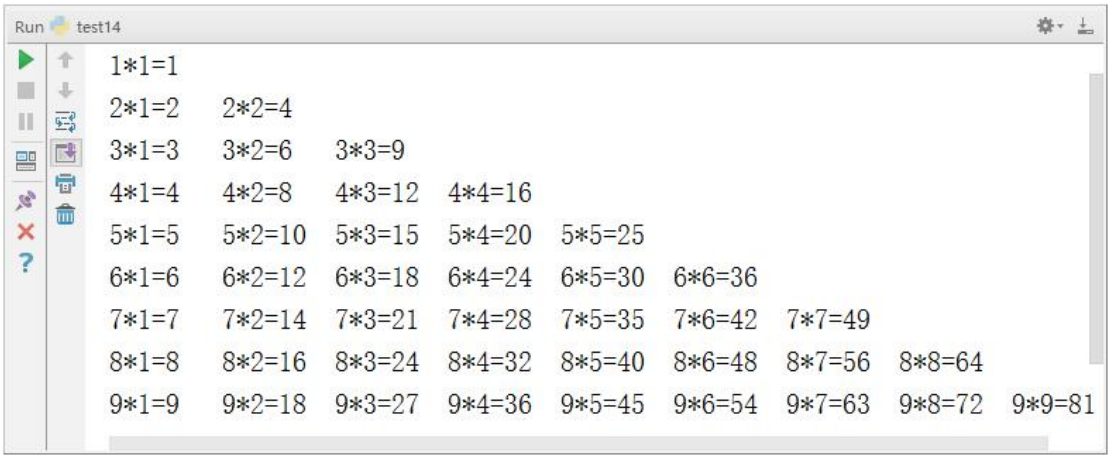


图 3-18 示例 3-14 执行结果

3.3.4 break

break 语句可用于 while 和 for 循环，用来结束整个循环。当有嵌套循环时，break 语句只

扫码观看：break

能跳出最近一层的循环。

【示例 3-15】使用 break 语句结束循环

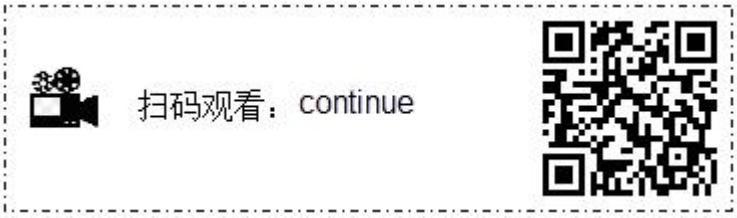
```
while True:
    a = input("请输入一个字符（输入 Q 或 q 结束）")
    if a.upper()=='Q':
        print("循环结束，退出")
        break
    else:
        print(a)
```

执行结果如图 3-19 所示：

图 3-19 示例 3-15 执行结果

在上面的代码中，while 循环的条件语句是 True，因此进入 while 循环后，每次都会接收控制台输入的值赋给变量 a，判断 a 的值是否是“q”或者“Q”。当输入的值是“q”或者“Q”时，执行 break 语句退出循环。

3.3.5 continue



与 break 语句对应的还有另一个 continue 语句，与 break 语句不同的是，continue 用于结束本次循环，继续下一次。多个循环嵌套时，continue 也是应用于最近的一层循环。而 break 语句用来彻底退出循环。

【示例 3-16】continue 语句

```
#for 循环来遍历 10 以内的奇数
for i in range(1,10):
    if i%2==0:
```

```
        continue
    else:
        print(i,end='\t')
```

执行结果如图 3-20 所示：

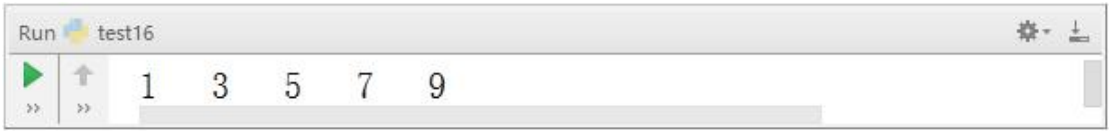
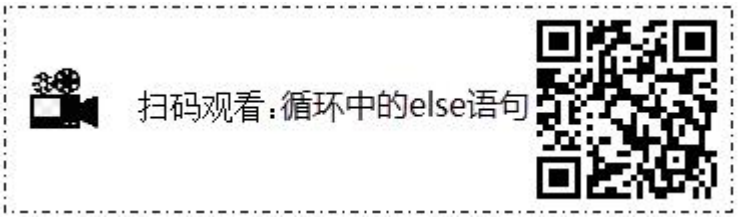


图 3-20 示例 3-16 执行结果

3.3.6 循环中的 else 语句



while、for 循环可以附带一个 else 语句（可选）。如果 for、while 语句没有被 break 语句结束，则会执行 else 子句，否则不执行。语法格式如下：

```
while  条件表达式：
    循环体
else:
    语句块

或者：
for  变量  in  可迭代对象：
    循环体
else:
    语句块
```

【示例 3-17】while 循环中的 else 语句

```
#while  else
num=0
while num<5:
    print(num,'的值小于 5')
    num+=1
else:
    print('循环正常结束，num 的值:',num)
```

执行结果如图 3-21 所示：

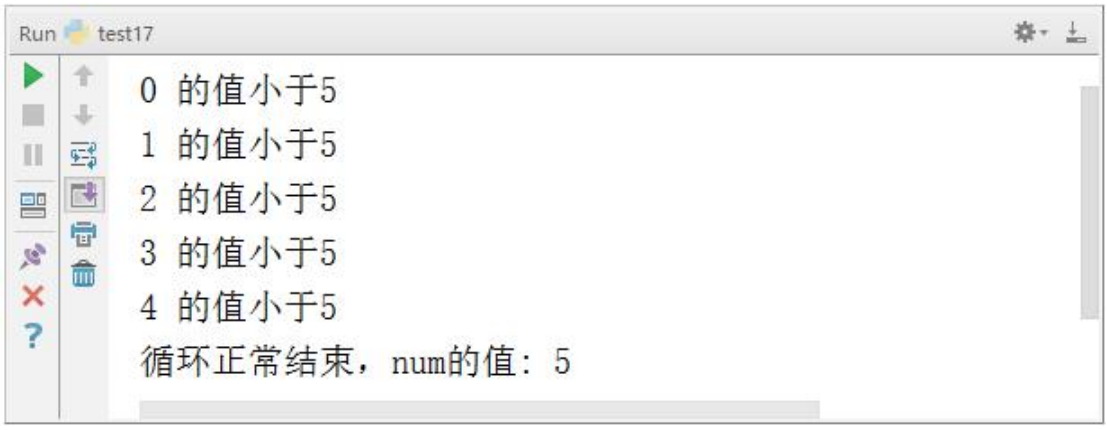


图 3-21 示例 3-17 执行结果

【示例 3-18】for 循环中的 else 语句

```
for num in range(1,10):
    if num==10:
        print(num)
        break
else:
    print('正常退出循环')
```

执行结果如图 3-22 所示：

图 3-22 示例 3-18 执行结果

习题

一、选择题

1. 下列 Python 语句正确的是 （ ）

A. min = x if x < y else y

B. max = x > y ? x : y

C. if (x > y) print x

D. while True : pass

2. 以下选项中，不是 Python 语言保留字的是（）

A. while B. continue C. goto D. for

3. 若 k 为整形， 下述 while 循环执行的次数为（）

```
k=1000
while k>1:
    print k
    k = k/2
```

A. 9 B. 10 C. 11 D. 1000

4. 设 s = ”Happy New Year”，则 s[3:8]的值为:（）

A. ‘ppy Ne’

B. ‘py Ne’

C. ‘ppy N’

D. ‘py New’

5. 以下叙述正确的是 （）

A. continue 语句的作用是结束整个循环的执行

B. 只能在循环体内使用 break 语句

C. 在循环体内使用 break 或 continue 语句的作用相同

D. 从多层循环嵌套中退出是，只能用使用 goto 语句 (C 语言是这样)

二、解答题

1. 条件判断语句有哪几种结构？
2. break 和 continue 语句的作用。
3. 在循环中 else 语句如何使用。
4. 下面程序的输出结果是：

```
x = True
y = False
z = False
if x or y and x:
```

```
    print('yes')
else:
    print('no')
```

三、编码题

1. 输入一个学生的成绩，将其转化成简单描述：不及格(小于 60)、及格(60-79)、良好(80-89)、优秀(90-100)。
2. 已知点的坐标(x,y)，判断其所在的象限。
3. 输入一个分数。分数在 0-100 之间。90 以上是 A,80 以上是 B，70 以上是 C，60 以上是 D。60 以下是 E。
4. 利用 while 循环，计算 1-100 之间数字的累加和；计算 1-100 之间偶数的累加和，计算 1-100 之间奇数的累加和。
5. 利用 for 循环，计算 1-100 之间数字的累加和；计算 1-100 之间偶数的累加和，计算 1-100 之间奇数的累加和。
6. 打印如下图案。

   0 0 0 0 0

   1 1 1 1 1

   2 2 2 2 2

   3 3 3 3 3

   4 4 4 4 4

7. 利用嵌套循环打印九九乘法表。

第四章 内建数据结构

本章引入一个新的概念：数据结构。数据结构是通过某种方式(例如对元素进行编号)组织在一起的数据元素的集合，这些数据元素可以是数字或者字符串，甚至可以是其他数据结构。在 Python 中，最基本的数据结构是序列，序列中的每个元素被分配一个序号，即元素的位置，也称为索引。第一个索引是 0，第二个索引则是 1，以此类推。

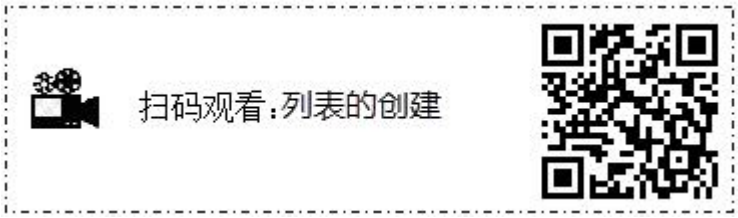
通过阅读本章，你可以：

- 了解元组、列表和字典的概念
- 元组、列表和字典的创建
- 列表、二维列表、表格数据的存储和读取
- 掌握列表、元组、字典的方法使用
- 了解元组的概念及与列表的区别
- 掌握如何将列表转换为元组
- 掌握如何用 dict 函数将序列转换为字典
- 掌握如何迭代字典和其他序列
- 掌握集合的创建和使用

4.1 列表

4.1.1 列表创建

1）基本语法[]创建



列表用于存储任意数目、任意类型的数据集合。在 Python 中，用方括号（[]）来表示列表，并用逗号来分隔其中的元素。语法格式如下：

```
list=[元素 1,元素 2,...]
```

【示例 4-1】创建整数类型的列表

```
a = [10,20,30,40]
```

其中，10,20,30,40 这些称为：列表 a 的元素。

同一个列表中不仅可以包含相同类型的元素，还可以包含不同类型的值。如示例 4-2 所示：

【示例 4-2】创建不同类型的列表

```
a = [10,20,'abc',True,3.14]
```

在上面的代码中，列表 a 中包含 5 个元素，这 5 个元素值是不同的数据类型，分别是整数（10,20）、字符串（abc）、布尔类型（True）、浮点类型（3.14）。

2）list()创建

使用 list()可以将任何可迭代的数据转化成列表。

【示例 4-3】list 创建列表

```
a = list()   #创建一个空的列表对象
b = list("gaoqi,sxt")
print(a)
print(b)
```

执行结果如图 4-1 所示：

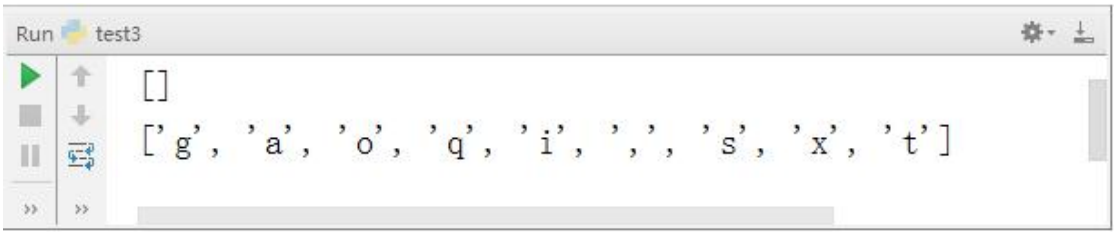


图 4-1 示例 4-3 执行结果

3）range()创建整数列表

range()可以帮助我们非常方便的创建整数列表，这在开发中及其有用。语法格式如下：

```
range([start,] end [,step])
```

其中 start 参数：可选，表示起始数字，默认是 0；end 参数：必选，表示结尾数字；step 参数：可选，表示步长，默认为 1。

Python3 中 range()返回的是一个 range 对象，而不是列表。需要通过 list()方法将其转换成列表对象。

【示例 4-4】range()创建整数列表

```
a=list(range(3,15,2))          #从 3 开始步长是 2 及从开始每次增加 2
b=list(range(15,3,-1))         #从 15 开始每次减 1
c=list(range(3,-10,-1))        #从 3 开始每次递减 1
print(a)
print(b)
print(c)
```

执行结果如图 4-2 所示：

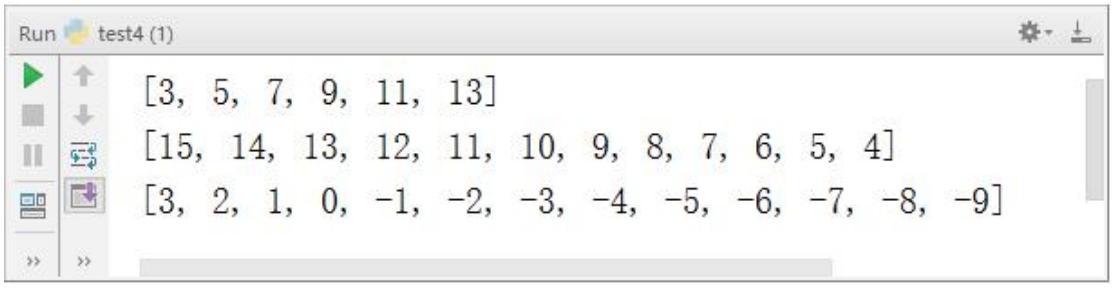


图 4-2 示例 4-4 执行结果

4）推导式生成列表

使用列表推导式可以非常方便的创建列表，在开发中经常使用。列表推导式生成列表对象，语法如下。

```
[表达式    for   item   in  可迭代对象 ]
```

【示例 4-5】推导式生成列表

```
a=[x for x in range(1,5)]
b=[x*2 for x in range(1,5)]
c=[x*2 for x in range(1,20) if x%5==0 ]
d=[x   for   x   in "abcdefg"]
print("a 列表的元素:",a)
print("b 列表的元素:",b)
print("c 列表的元素:",c)
print("d 列表的元素:",d)
```

执行结果如图 4-3 所示：

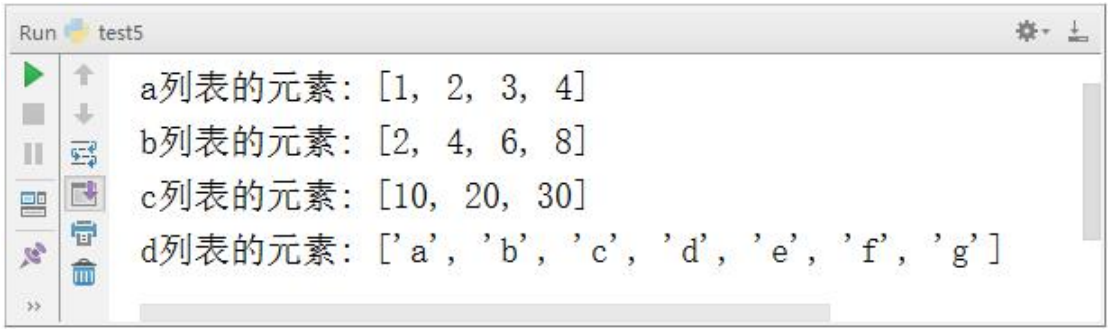
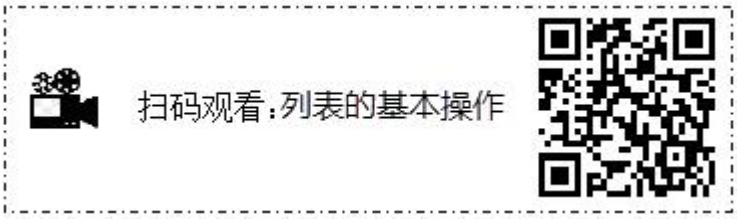


图 4-3 示例 4-5 执行结果

在上面的代码中，a 列表中的元素是通过遍历 range(1,5)生成的。b 列表中的元素是通过遍历 range(1,5)，对遍历的每个元素乘以 2 生成。c 列表中的元素是通过遍历 range(1,20)，对遍历的每个元素先判断是否能被 5 整除，如果能被 5 整除的元素再乘以 2 生成。d 列表中的元素是循环获取字符串”abcdefg”生成。

4.1.2 基本操作



列表中有一些自己的操作例如列表增加、列表修改、列表删除。当列表增加和删除元素时，列表会自动进行内存管理，大大减少了程序员的负担。但这个特点涉及列表元素的大量移动，效率较低。除非必要，一般只在列表的尾部添加元素或删除元素，这会大大提高列表的操作效率。

1） 列表的添加

在列表的末尾追加一个新对象，使用 append()方法。

【示例 4-6】使用 append()实现列表添加元素

```
#列表中添加元素
a=[10,20,30]
a.append(40)
print('增加元素后的列表：',a)
```

执行结果如图 4-4 所示：

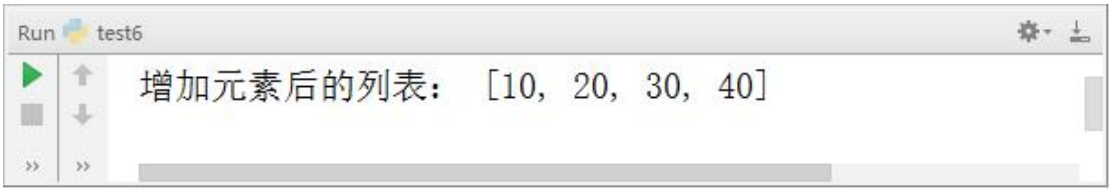


图 4-4 示例 4-6 执行结果

+运算符操作，并不是真正的尾部添加元素，而是创建新的列表对象；将原列表的元素和新列表的元素依次复制到新的列表对象中。这样，会涉及大量的复制操作，对于操作大量元素不建议使用。

【示例 4-7】使用+运算符实现列表添加元素

```
#列表使用+操作符相加
a=[10,20,30]
print('a 的地址：',id(a))
b=[40,50]
a=a+b
print('a 列表的元素：',a)
print('a 的地址：',id(a))
```

执行结果如图 4-5 所示：

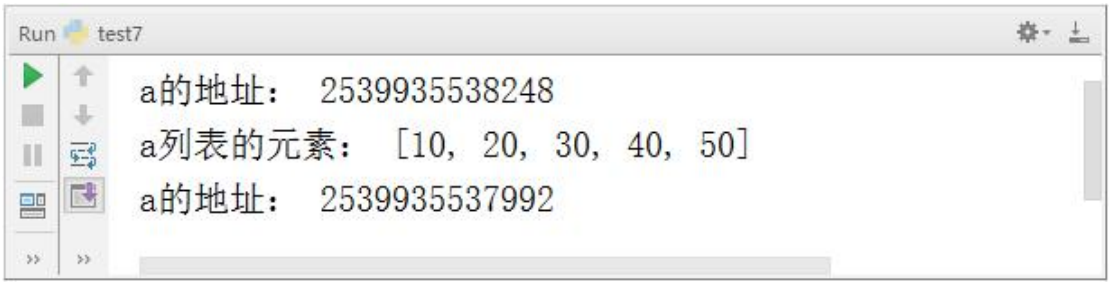


图 4-5 示例 4-7 执行结果

注意：

- 通过如上测试，发现变量 a 的地址发生了变化。也就是创建了新的列表对象。

extend()方法，将目标列表的所有元素添加到本列表的尾部，属于原地操作，不创建新的列表对象。

【示例 4-8】使用 extend()将目标列表的所有元素添加到本列表的尾部

```
#使用 extend()方法添加列表
a=[10,20,30]
print('a 的地址：',id(a))
b=[40,50]
a.extend(b)
print('a 列表的元素：',a)
print('a 的地址：',id(a))
```

执行结果如图 4-6 所示：

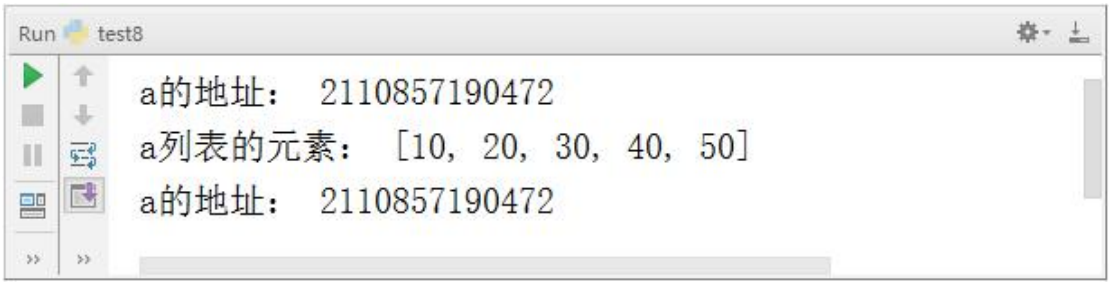


图 4-6 示例 4-8 执行结果

使用 insert()方法可以将指定的元素插入到列表对象的任意制定位置。这样会让插入位置后面所有的元素进行移动，会影响处理速度。涉及大量元素时，尽量避免使用。类似发生这种移动的函数还有：remove()、pop()、del()，它们在删除非尾部元素时也会发生操作位置后面元素的移动。

【示例 4-9】使用 insert()实现列表插入元素

```
#使用 insert 函数插入元素
a=[10,20,30]
a.insert(2,100)          #在列表 a 的索引 2 处插入元素 100
print(a)
```

执行结果如图 4-7 所示：

图 4-7 示例 4-9 执行结果

2） 列表的查找

a) 通过索引直接访问元素

序列中的所有元素都是有索引的，编号从 0 开始递增，最大到列表的长度减 1。序列中的所有元素都可以通过索引访问。

【示例 4-10】获取列表中第一个和第三个元素

```
a = [10,20,30,40,50,60,70,80,90]
print("列表中的第一个元素:",a[0])
print("列表中的第三个元素：",a[2])
```

执行结果如图 4-8 所示：

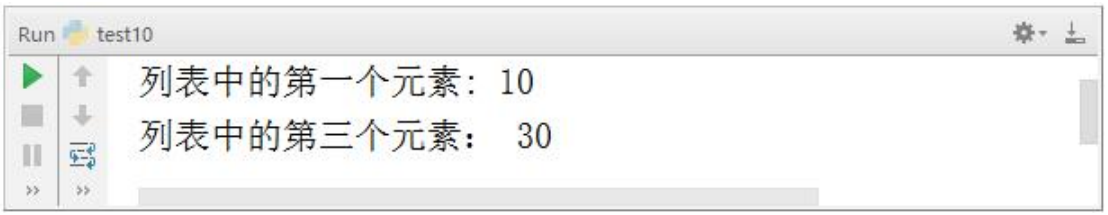


图 4-8 示例 4-10 执行结果

在上面的代码中，通过索引 0 和索引 2 分别获取列表中的第 1 个和第 3 个元素。

如果索引是 0 或正整数，那么 Python 语言会从列表的左侧第一个元素开始获取；如果索引是负数，那么 Python 语言会从列表右侧第一个元素开始获取。序列最后一个元素的索引是-1，倒数第二个元素的索引是-2，以此类推。

【示例 4-11】从右侧获取列表的值

```
a = [10,20,30,40,50,60,70,80,90]
```

```
print("列表中倒数第一个元素:",a[-1])
print("列表中倒数三个元素：",a[-3])
```

执行结果如图 4-9 所示：

图 4-9 示例 4-11 执行结果

当索引超过列表的索引范围，会抛出异常。

【示例 4-12】索引超过列表的索引范围，会抛出异常

```
a = [10,20,30,40,50,60,70,80,90]
#列表中有 9 个元素从左侧获取的索引范围：[0 到 8]
#列表中有 9 个元素从右侧获取的索引范围：[-1 到-9]
print(a[-9])
print(a[0])
print(a[9])          #超出范围
```

执行结果如图 4-10 所示：

图 4-10 示例 4-12 执行结果

b) index()获得指定元素在列表中首次出现的索引

index()可以获取指定元素首次出现的索引位置。语法是：index(value,[start,[end]])。其中，start 和 end 指定了搜索的范围。

【示例 4-13】使用 index()获得指定元素在列表中首次出现的索引

```
a = [10,20,30,40,50,20,30,20,30]
print(a.index(20))        #从列表中搜索第一个 20
print(a.index(20,3))      #从索引位置 3 开始往后搜索的第一个 20
```

```
print(a.index(30,5,7))   #从索引位置 5 到 7 这个区间，第一次出现 30 元素的位置
```

执行结果如图 4-11 所示：

图 4-11 示例 4-13 执行结果

3） 列表的修改

修改列表中的某一个元素，可以像使用数组一样对列表中的特定元素赋值，也就是使用一对中括号指定元素在列表中的索引，然后使用赋值运算符（=）进行赋值。

【示例 4-14】列表的修改

```
s=['admin','lili','john']
s[0]='管理员'
print('修改后列表的元素：',s)
```

执行结果如图 4-12 所示：

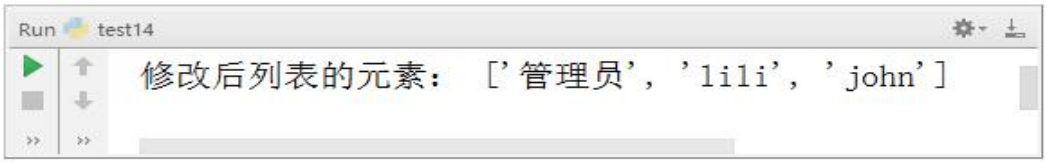


图 4-12 示例 4-14 执行结果

4） 列表的删除

a) del 删除

del 删除列表指定位置的元素。语法格式：del 元素。

【示例 4-15】del 删除元素

```
a=[1,2,3,4,5,6]
del a[2]
print("删除后列表的元素:",a)
```

执行结果如图 4-13 所示：

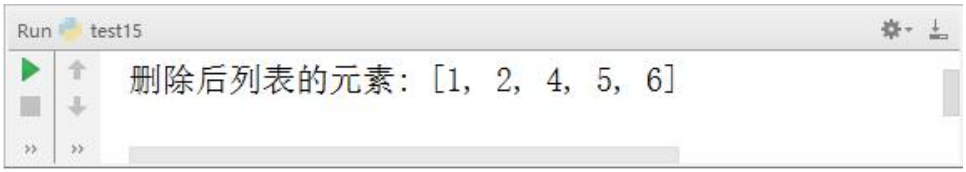


图 4-13 示例 4-15 执行结果

b) pop 删除

pop()删除并返回指定位置元素，如果未指定位置则默认操作列表最后一个元素。

【示例 4-16】pop 删除元素

```
a=[1,2,3,4,5,6]
b=a.pop()          #没有指定位置，默认的是最后一个元素
print(b)
print(a)
c=a.pop(2)         #删除下标为 2 的元素
print(c)
print(a)
```

执行结果如图 4-14 所示：

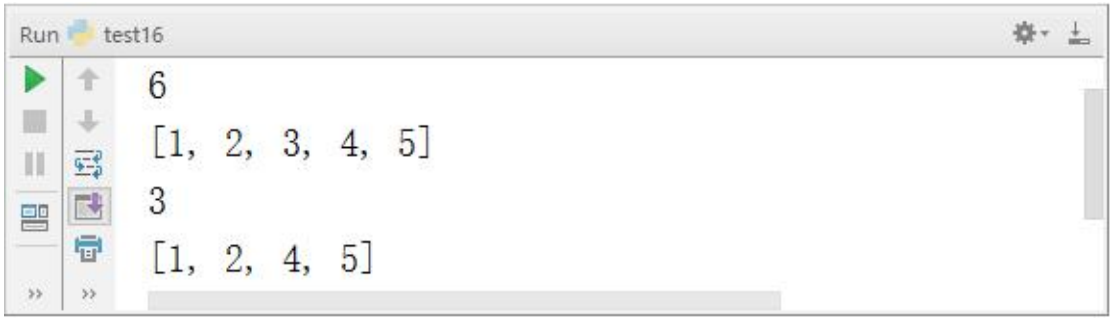


图 4-14 示例 4-16 执行结果

c) remove 删除

删除首次出现的指定元素，若不存在该元素抛出异常。

【示例 4-17】remove 删除元素

```
a = [10,20,30,40,50,20,30,20,30]
a.remove(20)
print(a)
a.remove(100)              #没有 100 这个元素，会抛出异常
```

执行结果如图 4-15 所示：

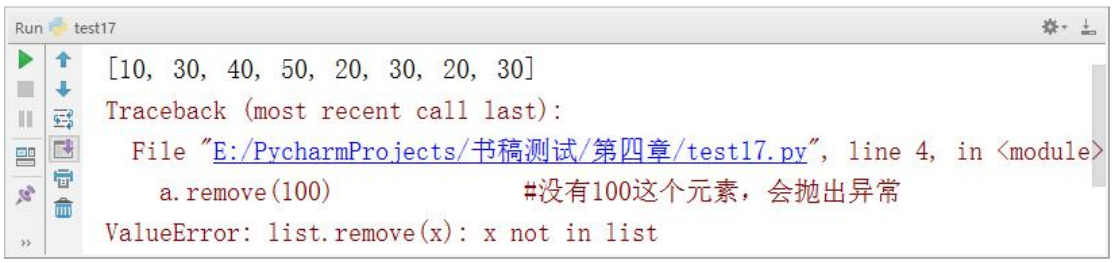


图 4-15 示例 4-17 执行结果

5） 列表的乘法

列表乘以一个数字 n 会生成新的列表，在新的列表中原来的列表将会被重复 n 次。

【示例 4-18】列表的乘法

```
a = ['sxt',100]
b = a*3
print(a)
print(b)
```

执行结果如图 4-16 所示：

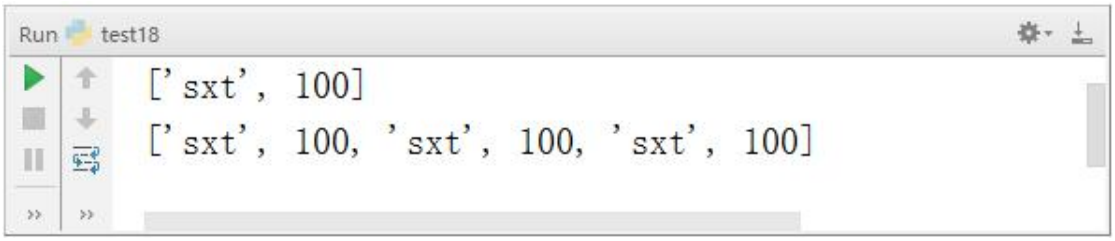


图 4-16 示例 4-18 执行结果

6） 列表的长度、最大值和最小值

len、max 和 min 这三个函数用于返回列表中元素的数量、列表中最大值、列表中最小值。使用 max 和 min 函数要注意一点，就是列表中的每个元素值必须是可比较的。否则会抛出异常。例如，如果列表中同事包含整数和字符串类型的元素，那么使用 max、min 函数就会抛出异常。

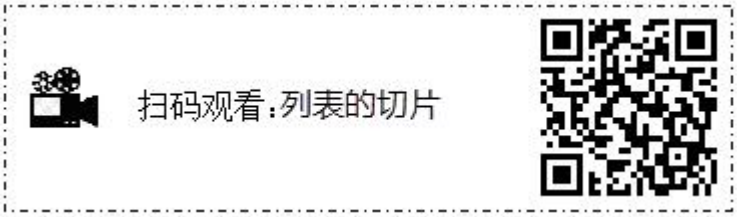
【示例 4-19】len()、max()、min()函数的使用

```
a=[10,20,30,40,50,60]
print(len(a))          #运行结果是 6
print(max(a))          #运行结果是 60
print(min(a))          #运行结果是 10
b=['a',30,'b',40]
print(max(b))          #字符串和数字不能比较，将抛出异常
```

执行结果如图 4-17 所示：

图 4-17 示例 4-19 执行结果

4.1.3 切片



切片操作是从列表 A 中获取一个子列表 B。列表 A 可以称为父列表。从 A 中获取 B，需要指定 B 在 A 中的开始索引和结束索引，因此，切片操作需要指定两个索引。对于列表的切片操作和字符串类似。切片是 Python 序列及其重要的操作，适用于列表、元组、字符串等等。切片的语法格式如下：

```
列表[起始偏移量 start：终止偏移量 end [:步长 step]]
```

典型操作(三个量为正数的情况)如表 4-1 所示：

表 4-1 典型操作(三个量为正数)

操作和说明	示例	结果
[:] 提取整个列表	[10,20,30][:]	[10,20,30]
[start:]从 start 索引开始到结尾	[10,20,30][1:]	[20,30]
[:end]从头开始知道 end-1	[10,20,30][:2]	[10,20]
[start:end]从 start 到 end-1	[10,20,30,40][1:3]	[20,30]
[start:end:step]从 start 提取到 end-1，步长是 step	[10,20,30,40,50,60,70][1:6:2]	[20, 40, 60]

【示例 4-20】列表的切片操作一

```
a=[10,20,30,40,50,60,70]
print(a[:])       #默认从列表的开始到列表结束 ，步长为 1
print(a[1:])      #从列表的索引 1 开始到结束，步长为 1
print(a[:2])      #从开始到索引 1，步长为 1
print(a[1:3])     #从列表索引 1 开始到索引 2，步长为 1
print(a[1:6:2]) #从列表索引 1 开始到索引 5，步长为 2
```

执行结果如图 4-18 所示：

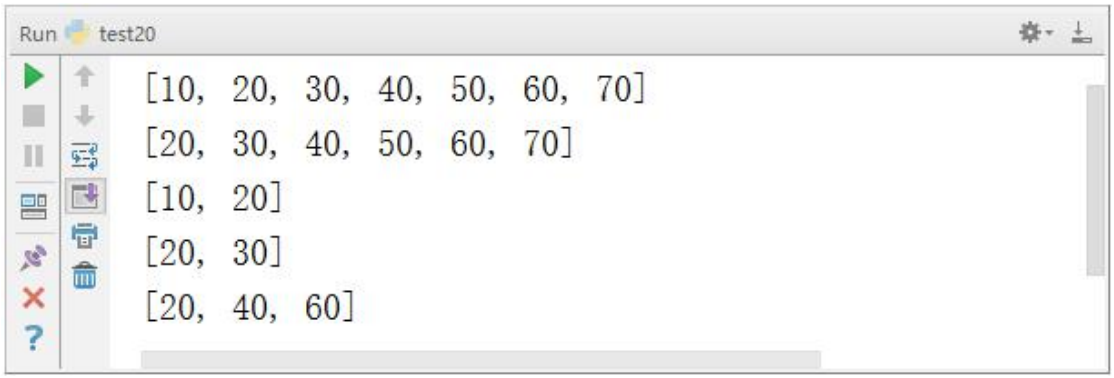


图 4-18 示例 4-20 执行结果

典型操作(三个量为负数的情况)如表 4-2 所示：

表 4-2 典型操作(三个量为负数)

示例	说明	结果
[10,20,30,40,50,60,70][-3:]	倒数三个	[50,60,70]
10,20,30,40,50,60,70][-5:-3]	倒数第五个到倒数第三个(包头不包尾)	[30,40]
[10,20,30,40,50,60,70][::-1]	步长为负，从右到左反向提取	[70, 60, 50, 40, 30, 20, 10]

【示例 4-21】列表的切片操作二

```
a=[10,20,30,40,50,60,70]
print(a[-3:])      #从列表的倒数第 3 个开始到列表结束
print(a[-5:-3])    #从倒数第 5 个开始到倒数第 4 个
print(a[::-1])     #从右到左反向提取
```

执行结果如图 4-19 所示：

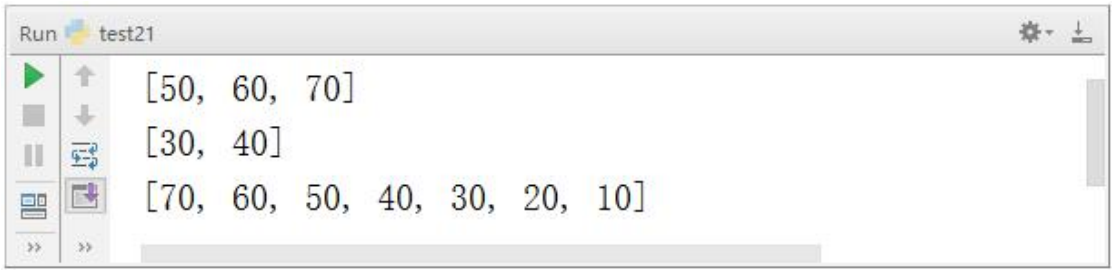


图 4-19 示例 4-21 执行结果

4.1.4 常用方法

列表提供了一些常用方法，如表 4-3 所示。

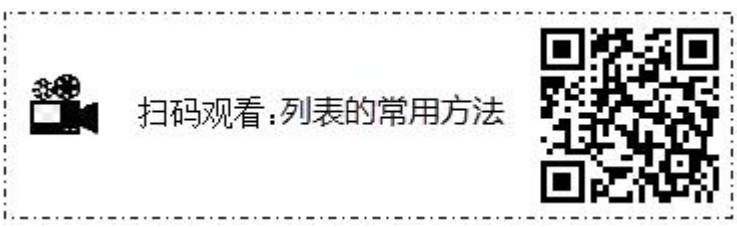


表 4-3 列表常用方法

方法	描述
copy()	用于赋值一个列表
count()	用于统计某个元素在列表中出现的次数
reverse()	用于将列表中的元素反向存放
sort()	用于对列表进行排序，调用该方法会改变原来的列表
sorted()	用于对列表排序，返回新列表，不对原列表做修改。
clear()	用于清空列表的内容

【示例 4-22】列表 copy()方法的使用

```
#copy()进行复制列表
aa=['a','b','c']
bb=aa.copy()
print(aa)
print(bb)
```

执行结果如图 4-20 所示：

图 4-20 示例 4-22 执行结果

【示例 4-23】列表 count()方法的使用

```
#count()统计元素出现的次数
a=[10,20,40,30,10,20,50,10]
print('元素 10 出现的次数：',a.count(10))
print('元素 20 出现的次数：',a.count(20))
print('元素 30 出现的次数：',a.count(30))
```

执行结果如图 4-21 所示：

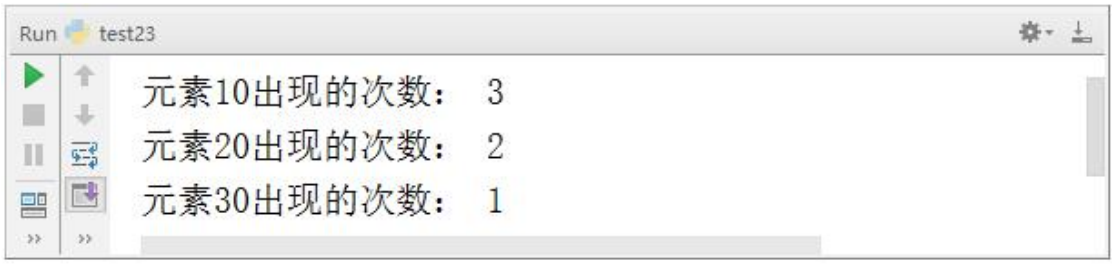


图 4-21 示例 4-23 执行结果

【示例 4-24】列表 reverse()方法的使用

```
#reverse()用于将列表中的元素反向存放
a=[1,2,3,4,5,6,7]
a.reverse()
print(a)
```

执行结果如图 4-22 所示：

图 4-22 示例 4-24 执行结果

【示例 4-25】列表 sort ()、sorted 方法的使用

```
#sort()用于对列表进行排序，调用该方法会改变原来的列表
a=[11,20,13,34,5,36,17]
a.sort()
print(a)
print('正序：',a)
a.sort(reverse=True)
print('逆序：',a)

#sorted 用于对列表进行排序，生成新列表，不改变原来的列表
print('-'*5,'sorted 排序','-'*5)
a=[11,20,13,34,5,36,17]
b=sorted(a)
print('a 列表：',a)                    #原来列表不会被修改
print('正序 b 列表：',b)
```

```
b=sorted(a,reverse=True)
print('逆序 b 列表：',b)
```

执行结果如图 4-23 所示：

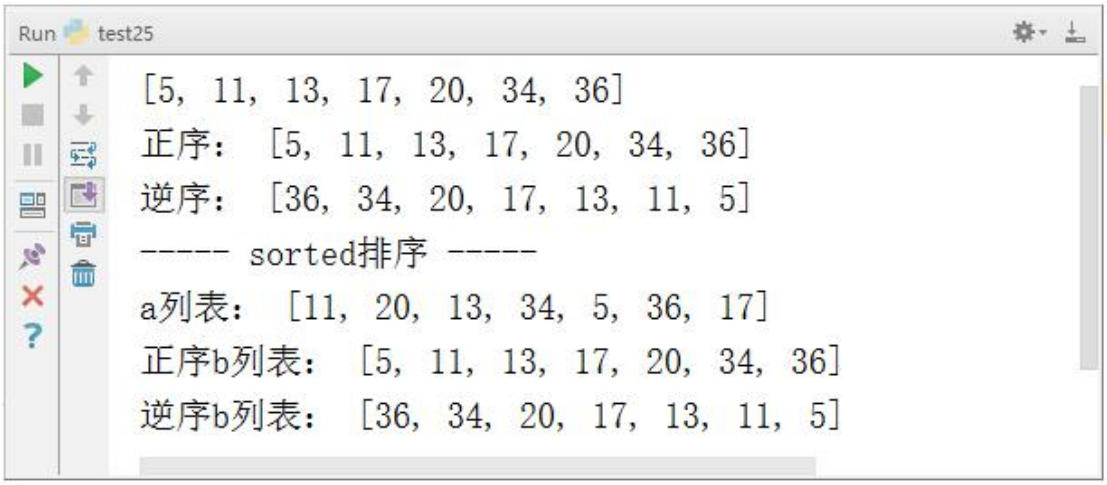


图 4-23 示例 4-25 执行结果

【示例 4-26】列表 clear()方法的使用

```
#clear()用于清空列表内容
a=[11,20,13,'a',True,'hello']
a.clear()
print(a)
```

执行结果如图 4-24 所示：

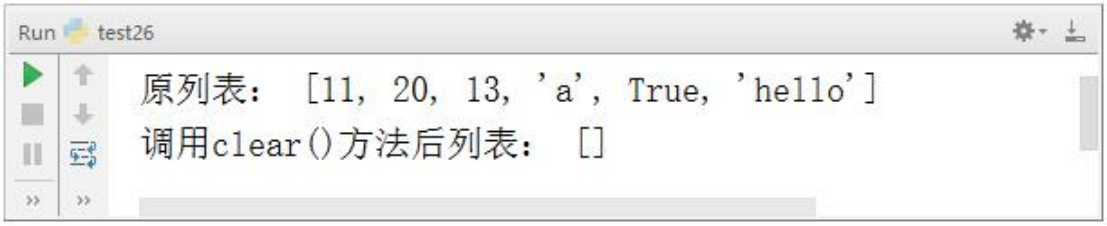


图 4-24 示例 4-26 执行结果

4.1.5 列表遍历

列表创建后，可以使用循环进行遍历获取列表中的每个元素。

【示例 4-27】for 循环遍历列表

```
#for 循环遍历列表
my_list = ["a","b","c","d"]
```

```
for value in my_list:
    print(value)
```

执行结果如图 4-25 所示：

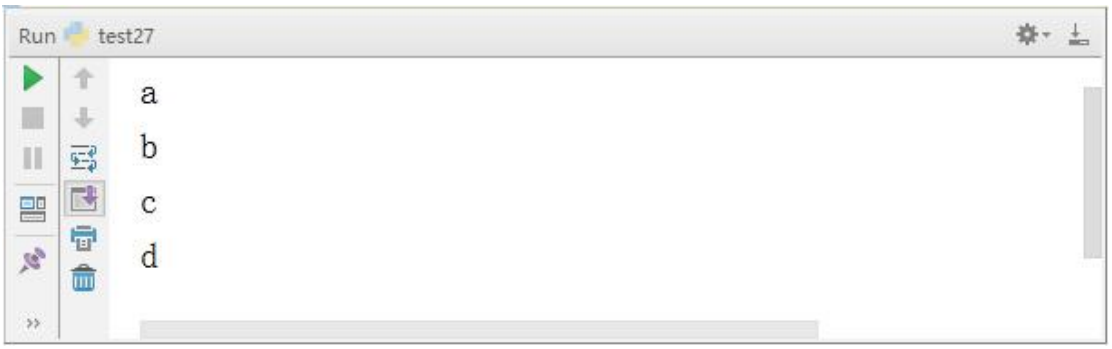


图 4-25 示例 4-27 执行结果

【示例 4-28】while 循环遍历列表

```
#while 循环遍历列表
my_list = ["a","b","c","d"]
#  定义变量
index = 0
#  定义变量  记录列表元素个数
l = len(my_list)
while index < l:
    #  通过下标索引获取元素
    value = my_list[index]
    print(value)
    index+=1
```

执行结果如图 4-26 所示：

图 4-26 示例 4-28 执行结果

4.1.6 enumerate() 函数

enumerate() 函数用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引

序列，同时列出数据和数据下标，一般用在 for 循环当中。enumerate() 方法的语法格式如下所示：

```
enumerate(sequence, [start=0])
```

其中 sequence 表示一个序列、迭代器或其他支持迭代对象；start 表示下标起始位置。

【示例 4-29】enumerate()函数的使用

```
seq = ['one', 'two', 'three']
for i, element in enumerate(seq):
    print(i, element)
```

执行结果如图 4-27 所示：

图 4-27 示例 4-29 执行结果

4.1.7 二维列表

一维列表可以存储一维、线性的数据。二维列表可以看成以列表为元素的列表，可以存储二维、表格的数据。二维列表的定义语法格式。


```
list=[ [元素 1,元素 2,元素 3...], [元素 1,元素 2,元素 3...]...]
```

表格数据模型是计算机世界最普遍的模型，可以这么说，大家在互联网上看到的所有数据本质上都是“表格”，无非是表格之间互相套用。如下表 4-4 是一张用户表。这样只需要再定义一个二维列表，存放用户表的数据。

表 4-4 用户表

姓名	年龄	薪资	城市
高小一	18	30000	北京
高小二	19	20000	上海
高小五	20	10000	深圳

【示例 4-30】二维列表存放用户数据

```
a = [
    ["高小一",18,30000,"北京"],
    ["高小二",19,20000,"上海"],
    ["高小一",20,10000,"深圳"],
    ]
for m in range(3):
    for n in range(4):
        print(a[m][n],end="\t")
    print() #打印完一行，换行
```

执行结果如图 4-28 所示：

图 4-28 示例 4-30 执行结果

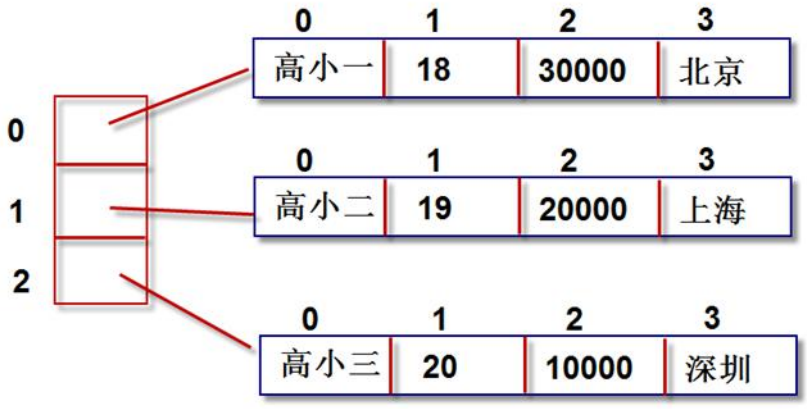


图 4-29 用户表内存结构图

4.2 元组（tuple）

元组和列表一样，也是一种序列。不同的是元组不能修改，也就是说，元组是只读的，不能对元组进行增加、删除、修改。定义元组非常简单，只需要用逗号(，)分隔值即可。

4.2.1 元组的创建

1）通过()创建元组，小括号可以省略。

【示例 4-31】通过()创建元组

```
#通过()创建元组
a=1,2,3,4,5,6,7          #创建一个元组  省略了括号
b=(1,2,3,4,5,6,7)        #创建一个元组
c=(42,)                  #创建一个只有一个元素值的元组
d=()                     #创建一个空的元组
print(a)
print(b)
print(c)
print(d)
```

执行结果如图 4-30 所示：

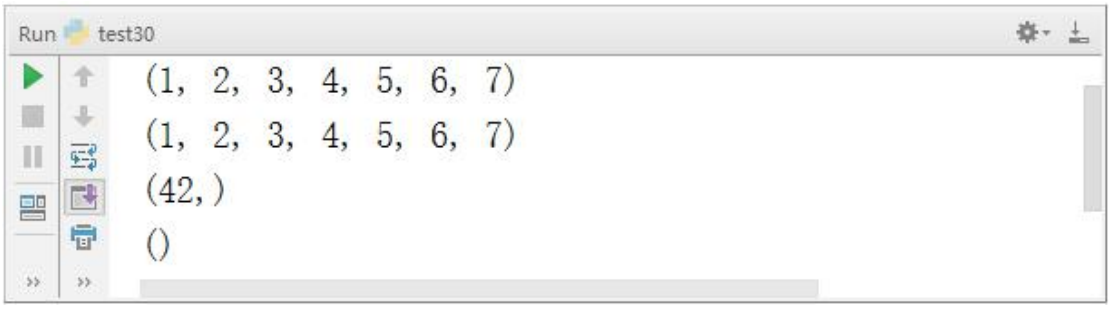


图 4-30 示例 4-31 执行结果

2）通过 tuple()创建元组

tuple 函数的功能与 list 函数基本上是一样的。以一个序列作为参数并把它转换为元祖。如果元素的参数就是元祖，那么改参数就会被原样返回。

【示例 4-32】使用 tuple()函数创建元组

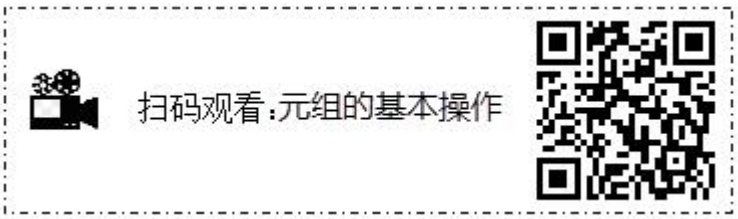
```
#使用 tuple()函数创建元组
a=tuple("abc")
b=tuple(range(3))
c=tuple((1,2,3,4,5))
print(a)
print(b)
```

```
print(c)
```

执行结果如图 4-31 所示：

图 4-31 示例 4-32 执行结果

4.2.2 元组的基本操作



【示例 4-33】元组的元素不能修改

```
a = (20,10,30,9,8)
a[3]=33                          #修改元组中的元素
```

执行结果如图 4-32 所示：

图 4-32 示例 4-33 执行结果

元组的元素访问和列表一样，只不过返回的仍然是元组对象。

【示例 4-34】元组的切片

```
a = (20,10,30,9,8)
print(a[1])
print(a[1:3])
print(a[:4])
```

执行结果如图 4-33 所示：

图 4-33 示例 4-34 执行结果

列表关于排序的方法 list.sort()是修改原列表对象，元组没有该方法。如果要对元组排序，只能使用内置函数 sorted(tupleObj)，并生成新的列表对象。

【示例 4-35】元组的排序

```
a = (20,10,30,9,8)
print(sorted(a))
```

执行结果如图 4-34 所示：

图 4-34 示例 4-35 执行结果

4.2.3 zip

zip(列表 1，列表 2，...)将多个列表对应位置的元素组合成为元组，并返回这个 zip 对象。

【示例 4-36】zip 合成元组

```
a = [10,20,30]
b = [40,50,60]
c = [70,80,90]
d = zip(a,b,c)
print(list(d))
```

执行结果如图 4-35 所示：

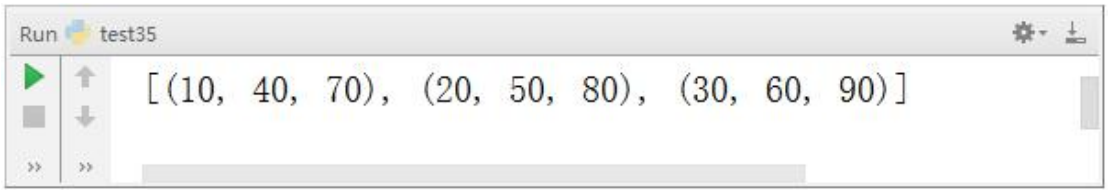
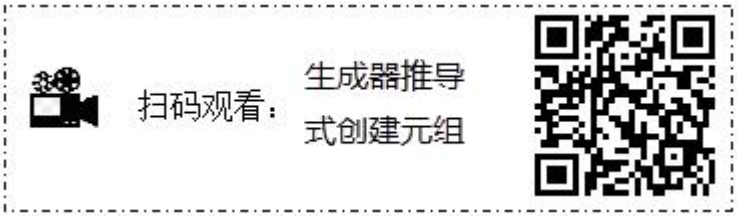


图 4-35 示例 4-36 执行结果

4.2.4 生成器推导式创建元组



从形式上看，生成器推导式与列表推导式类似，只是生成器推导式使用小括号。列表推导式直接生成列表对象，生成器推导式生成的不是列表也不是元组，而是一个生成器对象。

通过生成器对象，转化成列表或者元组。也可以使用生成器对象的__next__()方法进行遍历，或者直接作为迭代器对象来使用。不管什么方式使用，元素访问结束后，如果需要重新访问其中的元素，必须重新创建该生成器对象。

【示例 4-37】生成器的使用

```
#生成器的使用
s = (x*2 for x in range(5))
print(s)                    #生成器对象
print(tuple(s))             #tuple 函数转换为元组
print(tuple(s))             #只能访问一次元素。第二次就为空了。需要再生成一次
s = (x*2 for x in range(5))
print('next 方法获取元素：',s.__next__())
print('next 方法获取元素：',s.__next__())
print('next 方法获取元素：',s.__next__())
```

执行结果如图 4-36 所示：

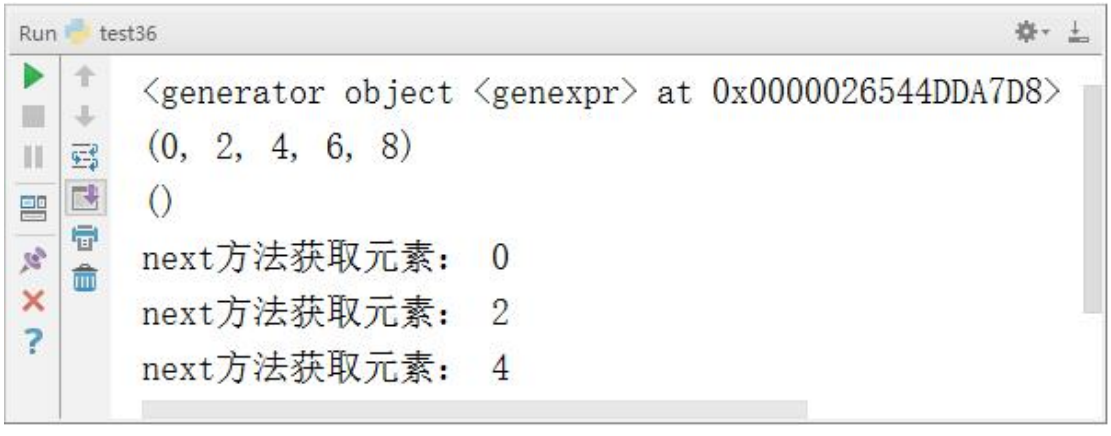


图 4-36 示例 4-37 执行结果

4.3 字典

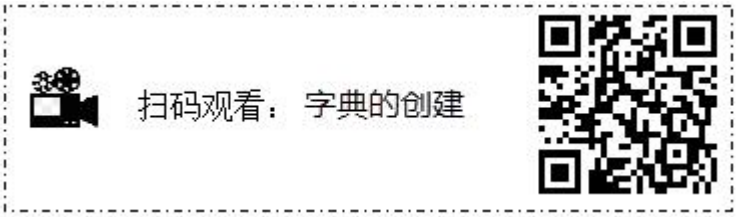
Python 内置了字典(dict)的支持，dict 全称 dictionary，在其他语言中也称为 map，使用

键-值（key-value）存储，具有极快的查找速度。字典是另一种可变容器模型，且可存储任意类型对象。字典的每个键值对(key=>value)用冒号(:)分割，每对之间用逗号(,)分割，整个字典包括在花括号({})中 ，语法格式如下：

```
d = {key1 : value1, key2 : value2 }
```

列表中通过“下标数字”找到对应的对象。字典中通过“键对象”找到对应的“值对象”。“键”是任意的不可变数据，比如：整数、浮点数、字符串、元组。但是：列表、字典、集合这些可变对象，不能作为“键”。并且“键”不可重复。“值”可以是任意的数据，并且可重复。

4.3.1 字典的创建



【示例 4-38】通过{}、dict()来创建字典对象

```
a = {'name':'gaoqi','age':18,'job':'programmer'}
b = dict(name='gaoqi',age=18,job='programmer')
c = dict([("name","gaoqi"),("age",18),("job",'programmer')])
d = {}     #空的字典对象
e = dict()     #空的字典对象
print('字典 a：',a)
print('字典 b：',b)
print('字典 c：',c)
print('字典 d：',d)
print('字典 e：',e)
```

执行结果如图 4-37 所示：

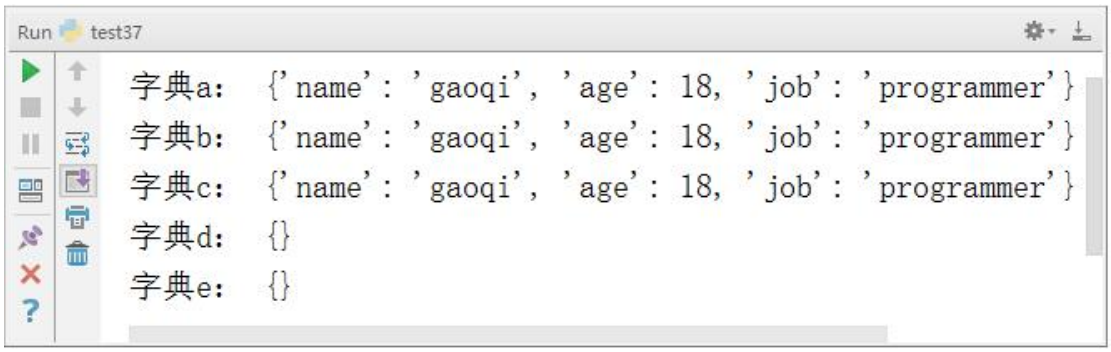


图 4-37 示例 4-38 执行结果

【示例 4-39】通过 zip()创建字典对象

```
k = ['name','age','job']
```

```
v = ['gaoqi',18,'techer']
a = dict(zip(k,v))
print('字典 a：',a)
```

执行结果如图 4-38 所示：

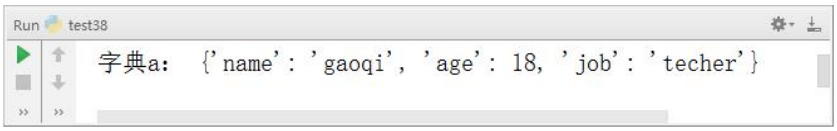


图 4-38 示例 4-39 执行结果

【示例 4-40】通过 fromkeys 创建值为空的字典

```
a = dict.fromkeys(['name','age','job'])
print('值为空的字典 a:',a)
```

执行结果如图 4-39 所示：

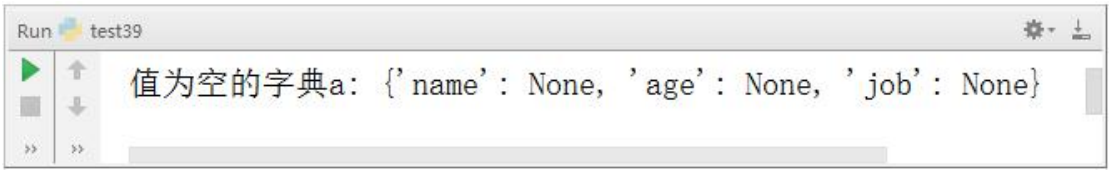
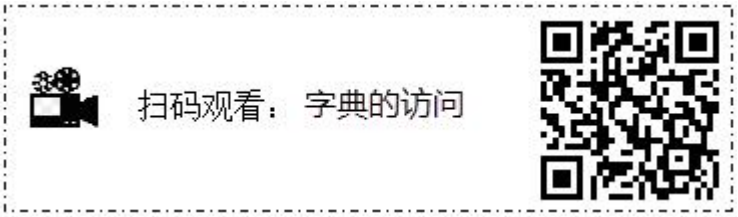


图 4-39 示例 4-40 执行结果

4.3.2 字典的访问



【示例 4-41】通过 [键] 获得“值”

```
a = {'name':'gaoqi','age':18,'job':'programmer'}
print('name:',a['name'])
print('age:',a['age'])
print('job:',a['job'])
```

执行结果如图 4-40 所示：

图 4-40 示例 4-41 执行结果

注意：

□ 若键不存在，则抛出异常

get 方法用于从字典中获取 key 对应的 value。优点是：指定键不存在，返回 None；也可以设定指定键不存在时默认返回的对象。推荐使用 get()获取“值对象”。

【示例 4-42】get 方法获取 key 对应的 value

```
a = {'name':'gaoqi','age':18,'job':'programmer'}
print('name:',a.get('name'))
print('age:',a.get('age'))
print('job:',a.get('job'))
print('sex:',a.get('sex'))
```

执行结果如图 4-41 所示：

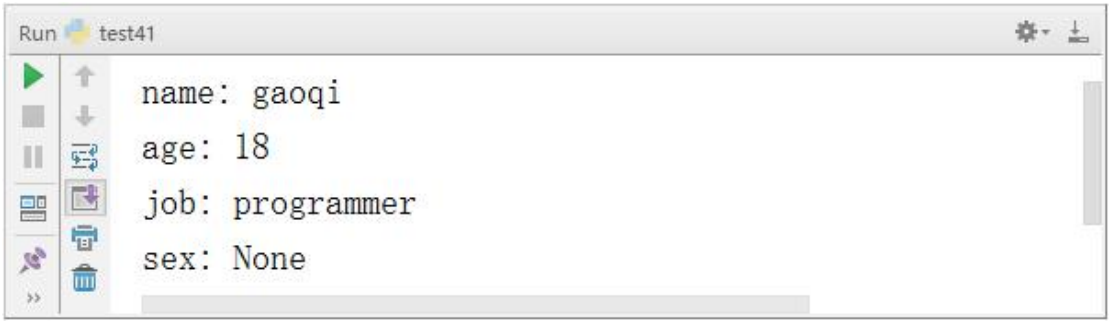
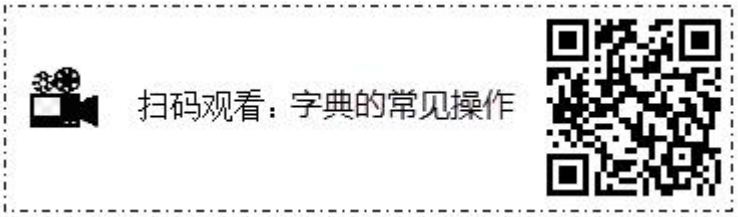


图 4-41 示例 4-42 执行结果

4.3.3 字典的常见操作

字典类似于列表，也可以进行增删改查等操作。就像其他内建类型一样，字典也有方法，这些方法非常有用。



给字典新增“键值对”。如果“键”已经存在，则覆盖旧的键值对；如果“键”不存在，则新增“键值对”。

【示例 4-43】字典添加、修改元素操作

```
a = {'name':'gaoqi','age':18,'job':'programmer'}
a['address']='北京'              #address 的键不存在，则新增
a['age']=28                      #age 的键存在，则进行修改
print(a)
```

执行结果如图 4-42 所示：

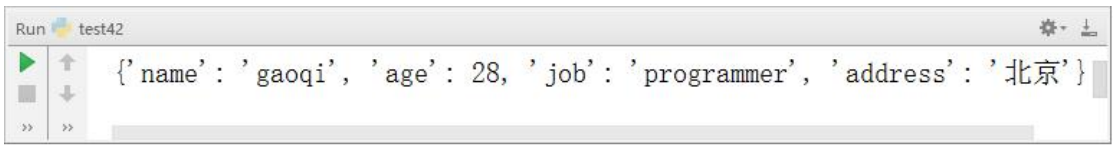


图 4-42 示例 4-43 执行结果

update 方法可以用一个字典中的元素更新另外一个字典。该方法接收一个参数。该参数表示用作更新数据的字典数据源。如果 key 有重复，则直接覆盖。

【示例 4-44】字典 update 方法的使用

```
a = {'name':'gaoqi','age':18,'job':'programmer'}
b = {'name':'gaoxixi','money':1000,'sex':'男的'}
a.update(b)
print(a)
```

执行结果如图 4-43 所示：

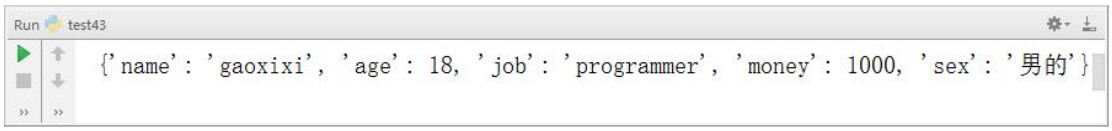


图 4-43 示例 4-42 执行结果

字典中元素的删除，可以使用 del()方法；或者 clear()删除所有键值对；pop()删除指定键值对，并返回对应的“值对象”。

【示例 4-45】字典使用 del()删除元素

```
a = {'name':'gaoqi','age':18,'job':'programmer'}
del(a['name'])            #del 删除元素
print(a)
b=a.pop('age')             #pop 删除元素，返回对应的值
print(b)
print(a)
a.clear()                  #清空字典元素
print(a)
```

执行结果如图 4-44 所示：

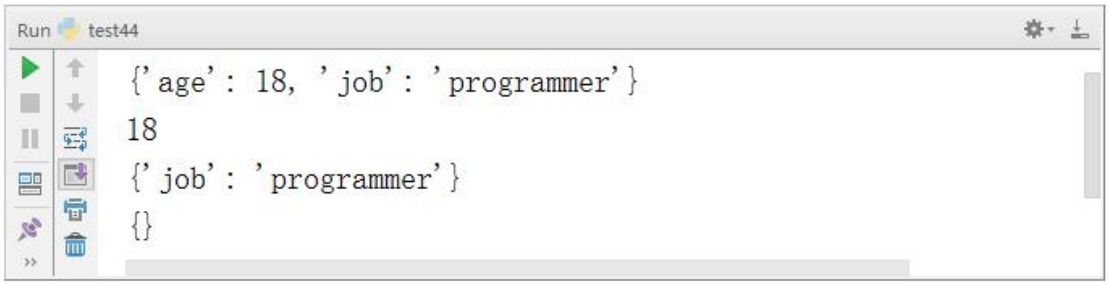


图 4-44 示例 4-45 执行结果

popitem()方法用于随机删除和返回该键值对。字典是“无序可变序列”，因此没有第一个元素、最后一个元素的概念；popitem 弹出随机的项，因为字典并没有"最后的元素"或者其他有关顺序的概念。若想一个接一个地移除并处理项，这个方法就非常有效（因为不用首先获取键的列表）。

【示例 4-46】字典使用 popitem()删除元素

```
a = {'name':'gaoqi','age':18,'job':'programmer'}
print(a.popitem())
print(a.popitem())
print(a.popitem())
```

执行结果如图 4-45 所示：

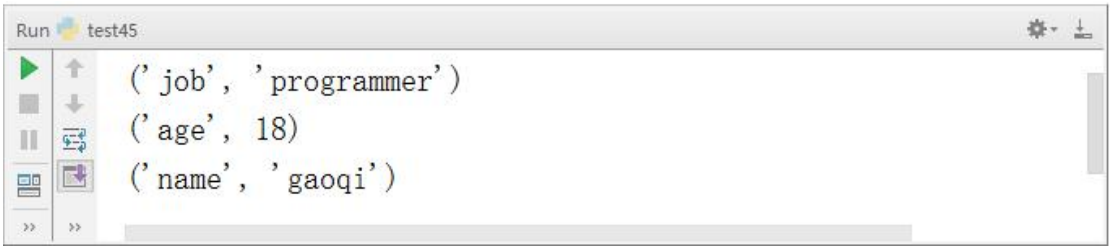


图 4-45 示例 4-46 执行结果

4.3.4 items 方法、keys 方法和 values 方法

items 方法用于返回字典中所有的 key-value 对。获得的每一个 key-value 对用一个元组表示。items 方法返回的值是一个被称为字典视图的特殊类型，可以被用于迭代（如使用在 for 循环中）。items 方法的返回值与字典使用了同样的值，也就是说，修改了字典或 items 方法的返回值，修改的结果就会反应在另一方法上。keys 方法用于返回字典中所有的 key，返回值类型与 items 方法类似，可以用于迭代。values 方法用于以迭代器形式返回字典中值的列表。

【示例 4-47】字典中 items()、keys()和 values()的使用

```
a = {'name':'gaoqi','age':18,'job':'programmer'}
print(a.items())      #字典中所有的 key-value 对
print(a.keys())       #字典中所有的 keys
print(a.values())     #字典中所有的 value
#通过遍历 key，根据 key 获取值
for key in a.keys():
    print(key,':',a.get(key))
```

执行结果如图 4-46 所示：

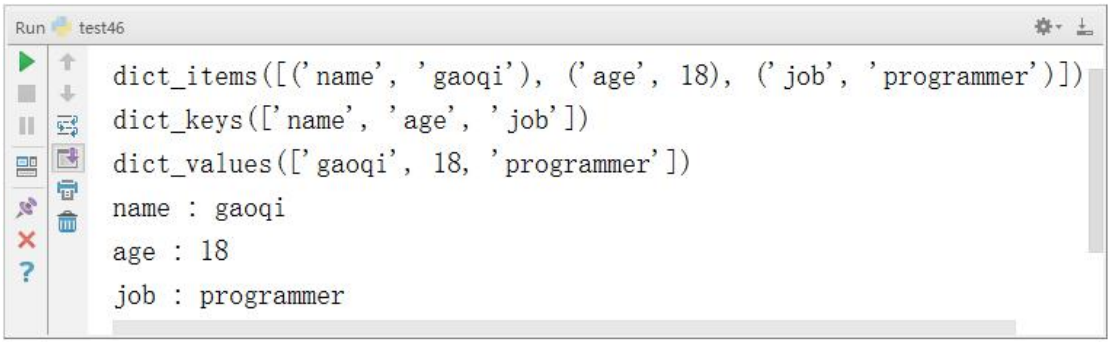


图 4-46 示例 4-47 执行结果

4.3.5 序列解包

序列解包可以用于元组、列表、字典。序列解包可以让我们方便的对多个变量赋值。

【示例 4-48】序列解包

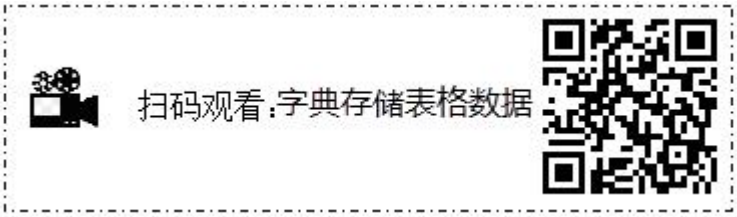
```
x,y,z=(20,30,10)
print('x:',x)
print('y:',y)
print('z:',z)
(a,b,c)=(9,8,10)
print('a:',a)
print('b:',b)
print('c:',c)
[a,b]=['hello','python']
print('a:',a)
print('b:',b)
```

执行结果如图 4-47 所示：

图 4-47 示例 4-48 执行结果

4.3.6 字典实现存储表格数据

之前使用列表实现了表 4-4 用户表，现在使用字典完成。



【示例 4-49】字典存储用户表数据

```
r1 = {"name":"高小一","age":18,"salary":30000,"city":"北京"}
r2 = {"name":"高小二","age":19,"salary":20000,"city":"上海"}
r3 = {"name":"高小五","age":20,"salary":10000,"city":"深圳"}

tb = [r1,r2,r3]
#获得第二行的人的薪资
print(tb[1].get("salary"))

#打印表中所有的的薪资
for i in range(len(tb)):      # i -->0,1,2
    print(tb[i].get("salary"))

#打印表的所有数据
for i in range(len(tb)):
    print(tb[i].get("name"),tb[i].get("age"),tb[i].get("salary"),tb[i].get("city"))
```

执行结果如图 4-48 所示：

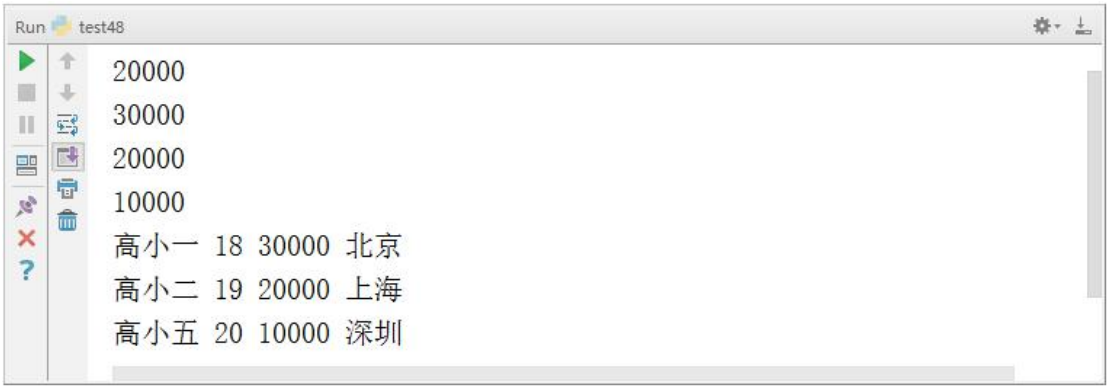


图 4-48 示例 4-49 执行结果

4.3.7 字典核心底层原理

字典对象的核心是散列表。散列表是一个稀疏数组（总是有空白元素的数组），数组的每个单元叫做 bucket。每个 bucket 有两部分：一个是键对象的引用，一个是值对象的引用。

由于，所有 bucket 结构和大小一致，可以通过偏移量来读取指定 bucket。

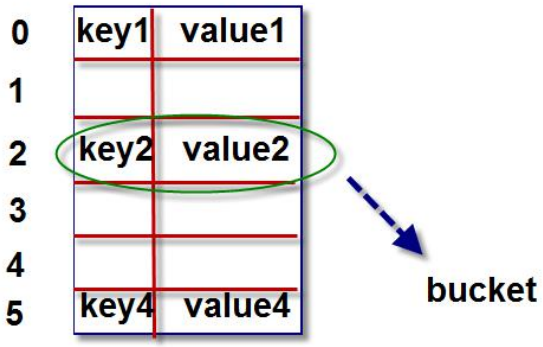


图 4-49 稀疏数组

1) 将一个键值对放进字典的底层过程

```
a = {}
a["name"]="gaoqi"
```

假设字典 a 对象创建完后，数组长度为 8。要把”name”=”gaoqi”这个键值对放到字典对象 a 中，首先第一步需要计算键”name”的散列值。Python 中可以通过 hash()来计算。

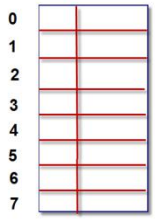


图 4-50 长度为 8 的稀疏数组

```
>>>bin(hash("name"))
'-0b10101111010011101101011001001O1'
```

由于数组长度为 8，可以拿计算出的散列值的最右边 3 位数字作为偏移量，即“101”，十进制是数字 5。查看偏移量 5，对应的 bucket 是否为空。如果为空，则将键值对放进去。如果不为空，则依次取右边 3 位作为偏移量，即“100”，十进制是数字 4。再查看偏移量为 4 的 bucket 是否为空。直到找到为空的 bucket 将键值对放进去。

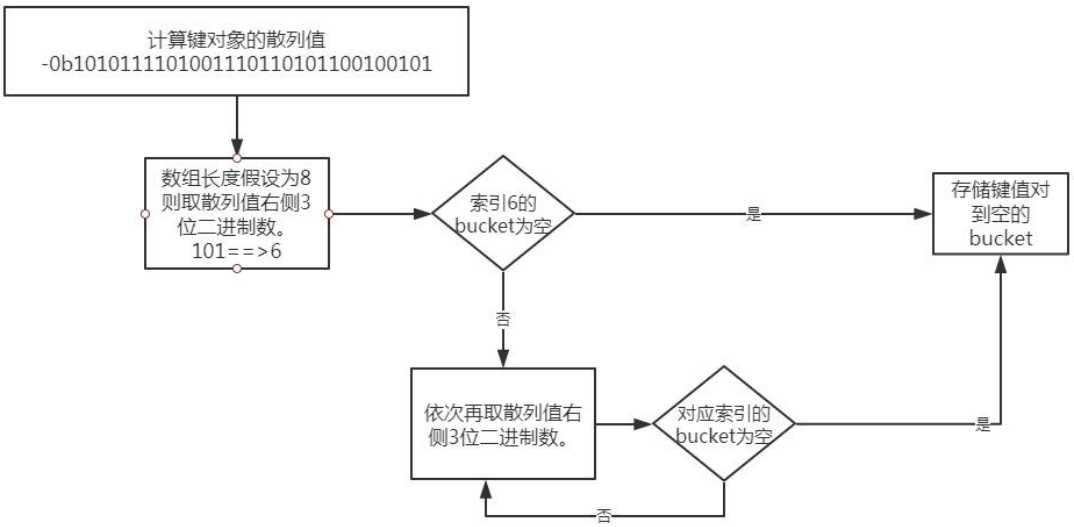


图 4-51 存储流程图

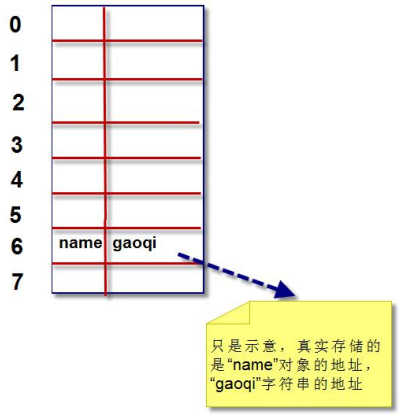


图 4-52 存储结果

python 会根据散列表的拥挤程度来看是否需要“扩容”，如果接近 2/3 时，数组就会扩

容。“扩容”指的是：创造更大的数组，将原有内容拷贝到新数组中。

2) 根据键查找“键值对”的底层过程

当调用 a.get(“name”)，就是根据键“name”查找到“键值对”，从而找到值对象“gaoqi”。首先仍然要计算“name”对象的散列值：

```
>>>bin(hash("name"))
'-0b10101111010011101101011001001O1'
```

和存储的底层流程算法一致，也是依次取散列值的不同位置的数字。假设数组长度为 8，可以拿计算出的散列值的最右边 3 位数字作为偏移量，即“101”，十进制是数字 5。查看偏移量 5，对应的 bucket 是否为空。如果为空，则返回 None。如果不为空，则将这个 bucket 的键对象计算对应散列值，和我们的散列值进行比较，如果相等。则将对应“值对象”返回。如果不相等，则再依次取其他几位数字，重新计算偏移量。依次取完后，仍然没有找到。则返回 None。

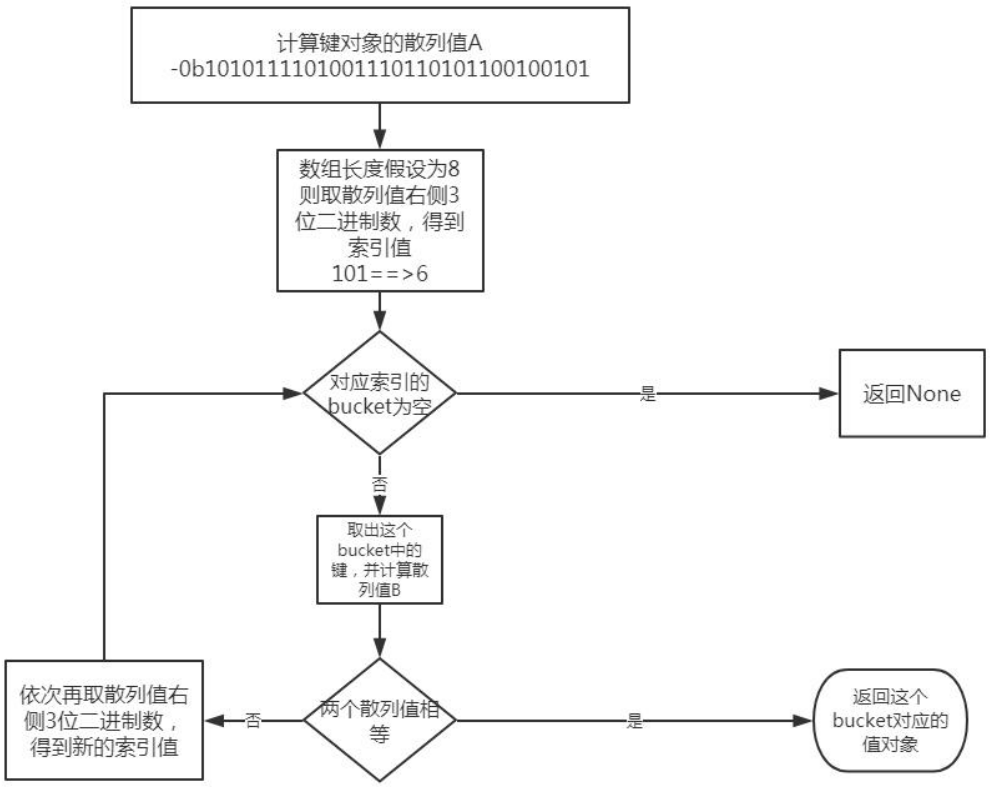


图 4-53 查找流程图

3) 用法总结：

1. 键必须可散列

(1) 数字、字符串、元组，都是可散列的。

(2) 自定义对象需要支持下面三点：

①支持 hash()函数

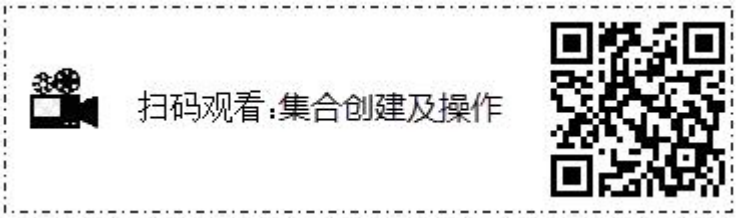
②支持通过__eq__()方法检测相等性。

③若 a==b 为真，则 hash(a)==hash(b)也为真。

2. 字典在内存中开销巨大，典型的空间换时间。
3. 键查询速度很快。
4. 往字典里面添加新建可能导致扩容，导致散列表中键的次序变化。因此，不要在遍历字典的同时进行字典的修改。

4.4 集合

集合是无序可变，元素不能重复。实际上，集合底层是字典实现，集合的所有元素都是字典中的“键对象”，因此是不能重复的且唯一的。



4.4.1 集合创建

【示例 4-50】 使用{}创建集合对象

```
a = {3,5,7}                    #集合存储整数
b={'hello','python'}           #集合存储字符串
c={3,4,True,'abc',56.4}        #集合存储不同类型的数据
print(a)
print(b)
print(c)
```

执行结果如图 4-54 所示：

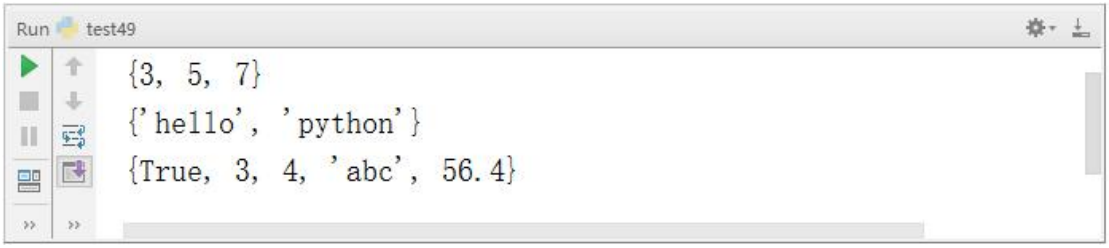


图 4-54 示例 4-50 执行结果

【示例 4-51】使用 set()，将列表、元组等可迭代对象转成集合。如果原来数据存在重复数据，则只保留一个。

```
a = ['a','b','c','b']
b=set(a)                       #将列表转换为集合
print(b)
```

```
c=(1,2,3,4,5)
d=set(c)                    #将元组转换为集合
print(d)
```

执行结果如图 4-55 所示：

图 4-55 示例 4-51 执行结果

4.4.2 集合添加

【示例 4-52】使用 add()实现集合添加元素

```
a = {3,5,7}
a.add('hello')          #add 方法添加元素
print(a)
```

执行结果如图 4-56 所示：

图 4-56 示例 4-52 执行结果

4.4.3 集合删除

【示例 4-53】使用 remove()实现集合删除指定元素、clear()清空整个集合

```
a = {10,20,30,40,50}
a.remove(20)                #删除集合中的元素
print(a)
a.clear()                   #将集合清空
print(a)
```

执行结果如图 4-57 所示：

图 4-57 示例 4-53 执行结果

4.4.4 集合其他操作

像数学中概念一样，Python 对集合也提供了并集、交集、差集等运算。

【示例 4-54】集合的并集、交集、差集运算

```
a = {1,3,'sxt'}
b = {'he','it','sxt'}
print('并集：',a|b)                    #并集
print('并集：',a.union(b))             #并集
print('交集：',a&b)                    #交集
print('交集：',a.intersection(b))      #交集
print('差集：',a-b)                    #差集
print('差集：',a.difference(b))        #差集
```

执行结果如图 4-58 所示：

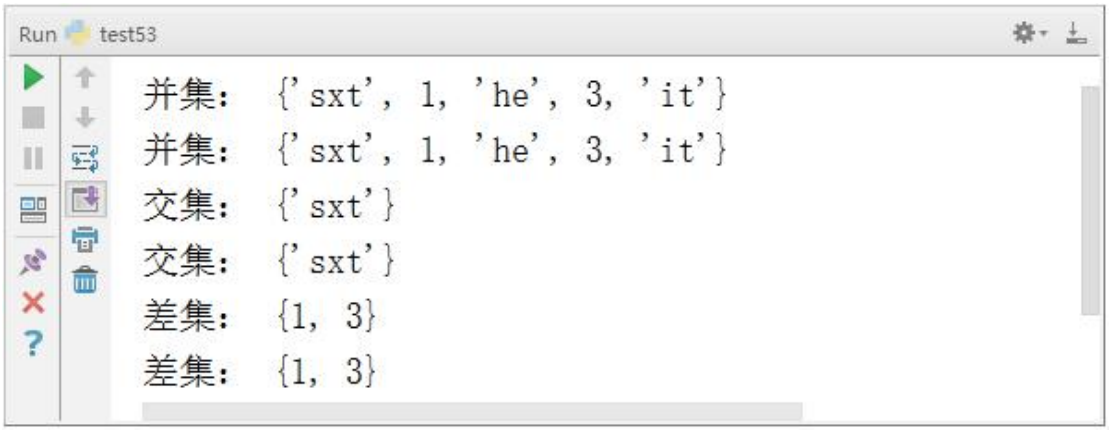


图 4-58 示例 4-54 执行结果

习题

一、选择题

1. 以下哪个对象不属于 Itarable 的( )

A. list　　B. tuple

C. dict　　D. float

2. 以下哪个类型不可以进行切片操作　( )

A. str　　B. list

C. tuple　D. dict

3. 以下哪个属于可变对象 ( )

A. 数值类型(int,float)

B.　list

C.　tuple

D. str

4. 以下表达式，正确定义了一个集合数据对象的是：（）

A. x = { 200, ’ flg’, 20.3}

B. x = (200, ’ flg’, 20.3)

C. x = [200, ’ flg’, 20.3]

D. x = {‘flg’ : 20.3}

5. 关于类型转换，说法错误的有(　)

A. int <-> float　　B. tuple <-> list

C. list<-> dict D. str <-> list

二、简答题

1. 元组和列表有哪些共同点？有哪些不同点？
2. 集合和字典有什么关系？
3. 判断 dict 有没有某 key 用的方法。
4. python 里面如何实现 tuple 和 list 的转换。
5. 如下代码执行结果是：

```
kvps = {'1':1,'2':2}
theCopy = kvps
kvps['1'] = 5
sum = kvps['1'] + theCopy['1']
print(sum)
```

三、编码题

1. 使用 range 生成序列：30,40,50,60,70,80。

2. 推导式生成列表： a = [x*2 for x in range(100) if x%9==0]，列表 a 的结果。

3. 使用二维列表存储表格信息，并画出简单的内存存储示意图：

姓名	年龄	薪资	城市
高小一	18	30000	北京
高小二	19	20000	上海
高小五	20	10000	深圳

4. 创建一个字典对象，包含如下信息：

   支出金额：300.15，支出日期：2018.10.18，支出人：高小七

5. 使用字典存储行数据，最后将整个表使用列表存储起来。

姓名	年龄	薪资	城市
高小一	18	30000	北京
高小二	19	20000	上海
高小五	20	10000	深圳

第五章 函数

本章为了便于维护和更好地实现模块化，大量的程序被分解为多个函数。Python 不仅简化了函数的定义过程，而且还引入了其他一些函数式编程中语言中的优秀特性。函数是可重用的程序代码块。函数的作用，不仅可以实现代码的复用，更能实现代码的一致性。一致性指的是，只要修改函数的代码，则所有调用该函数的地方都能得到体现。通过阅读本章，你可以：

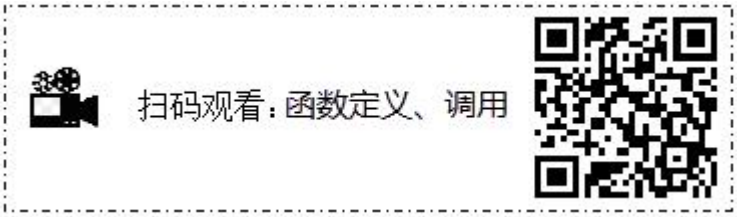
- 了解函数的简介
- 掌握如何创建一个函数
- 掌握函数参数的使用
- 掌握函数如何返回一个值
- lambda 表达式和匿名函数
- 了解函数中的作用域
- 了解递归函数

5.1 函数的基础

接触过 C、Java 等高级程序设计语言的读者对函数肯定不陌生。函数可重复使用，用来实现相关联功能的代码段。函数能提高程序的模块性、代码的重复利用率。Python 函数分为两类，即内建函数、自定义函数。例如，可以使用 print 函数输出计算结果，使用 input 函数接收用户的输入。除了系统内置的函数之外，程序员还可以根据需要编写自己的函数。

5.1.1 函数的定义

函数的定义非常简单，使用关键字 def 定义。函数在使用前必须定义，Python 函数的定义格式如下：




```
def  函数名 ([参数 1,参数 2...]) :
    代码
    [return 表达式]
```

其中函数名可以是字母、数字或下划线组成的字符串，但是不能以数字开头。函数的参数放在一对圆括号中，参数的个数可以有零个、一个或多个，参数之间用逗号隔开，这种参数称为形式参数。括号后面以冒号结束，冒号下面就是函数的主体。返回值不是必须的。很多函数可能没有返回值。如示例 5-1 所示：

【示例 5-1】定义函数实现自我介绍

```
def printInfo():
```

```
    print('我叫张三,今年 18 岁，是一名大学生')
```

5.1.2 函数的调用

函数创建成功后，接着看一下函数调用的问题。函数的调用采用函数名加一对圆括号的方式，圆括号中的参数是传递给函数的具体值。函数调用的语法格式如下：

```
函数名(实参 1,实参 2...)
```

注意：

- 即使函数不需要参数，调用函数时候，也要在函数名后使用一对空的圆括号

【示例 5-2】调用函数

```
def printInfo():
    print('我叫张三,今年 18 岁，是一名大学生')
printInfo()      #没有参数，后面也要加上()
```

执行结果如图 5-1 所示：

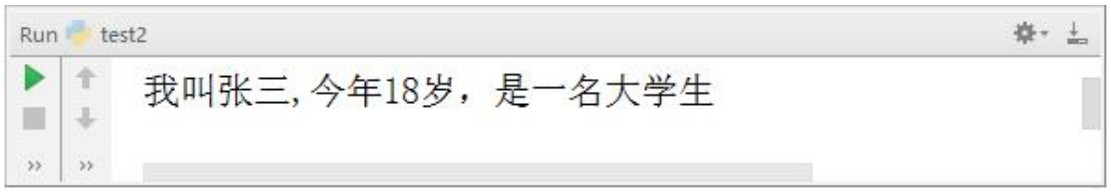


图 5-1 示例 5-2 执行结果

上面的程序虽然说达到了效果，但是不够灵活，如果想要打印李四的个人信息，就需要去修改源码，这样的话会麻烦，因此在定义的时候可以让函数接收数据，这样的话就使程序变得比较灵活，就能达到想要的效果，需要传递参数。

5.1.3 函数的形参和实参

在 def 语句中函数名后面圆括号中的参数称为形参，而调用函数时指定的参数称为实参。

【示例 5-3】定义带参数函数实现自我介绍

```
#定义函数
def printInfo(name,age):
    print('我叫{0},今年{1}岁，是一名大学生'.format(name,age))
#调用函数
printInfo('李四',19)
```

执行结果如图 5-2 所示：

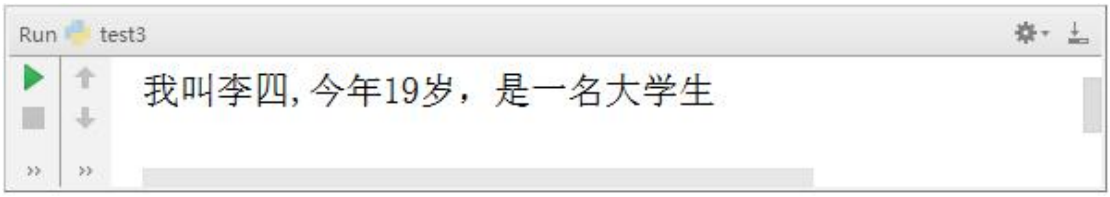
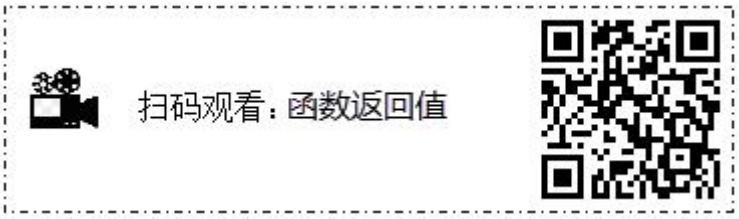


图 5-2 示例 5-3 执行结果

上面的 printInfo 函数中，在定义时写的 printInfo(name,age)。name 和 age 称为“形式参数”，简称“形参”。在调用函数 printInfo('李四',19)时，传递的'李四'和 19 称为“实际参数”，简称“实参”。

5.1.4 函数的返回值



函数的返回使用 return 语句，return 的后面可以是变量或者表达式。

【示例 5-4】使用 return 将两数之和返回

```
#定义函数计算两数之和
def add(a,b):
    return a+b
#调用函数
sum=add(20,30)
print('两数的和:',sum)
```

执行结果如图 5-3 所示：

图 5-3 示例 5-4 执行结果

【示例 5-5】通过函数参数传入斐波那切数列长度，使用 return 语句返回计算结果

```
#通过函数参数传入斐波那切数列长度，使用 return 语句返回计算结果
#定义函数
def fibs(n):        #n 代码斐波那切数列的长度
    #定义斐波那切数列的初识列表
    result=[0,1]
```

```
    #通过循环计算，将结果添加到列表中
    for x in range(n-2):
        result.append(result[-1]+result[-2])
    #将计算结果返回
    return result
#调用函数
re=fibs(10)
print('斐波那切数列:',re)
```

执行结果如图 5-4 所示：

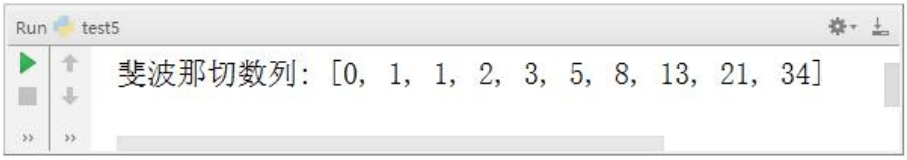
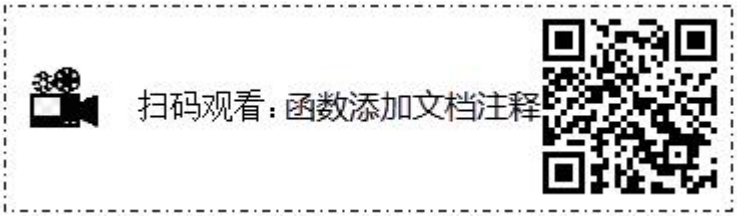


图 5-4 示例 5-5 执行结果

5.1.5 为函数添加文档注释



注释尽管在程序中不是必需的，但却是必要的。如果没有注释，那么程序就很难被别人读懂，甚至过段时间，自己都看不明白自己编写的程序。Python 语言支持单行注释和多行注释，其中单行注释使用“#”表示，多行注释使用三个单引号或三个双引号将注释内容括起来。对于函数来说，还可以使用另外一种注释：文档注释，也有人成为“函数的注释”。

文档注释，需要在函数头(包含 def 关键字的那一行)的下一行用三个单引号或者三个双引号来实现，中间可以加入多行文字进行说明。

【示例 5-6】测试文档注释

```
def print_star(n):
    '''根据传入的 n，打印多个星号'''
    print("*"*n)
#使用函数的属性"__doc__"获取文档注释
print(print_star.__doc__)
#直接使用 help 函数获取函数的文档注释
help(print_star)
```

执行结果如图 5-5 所示：

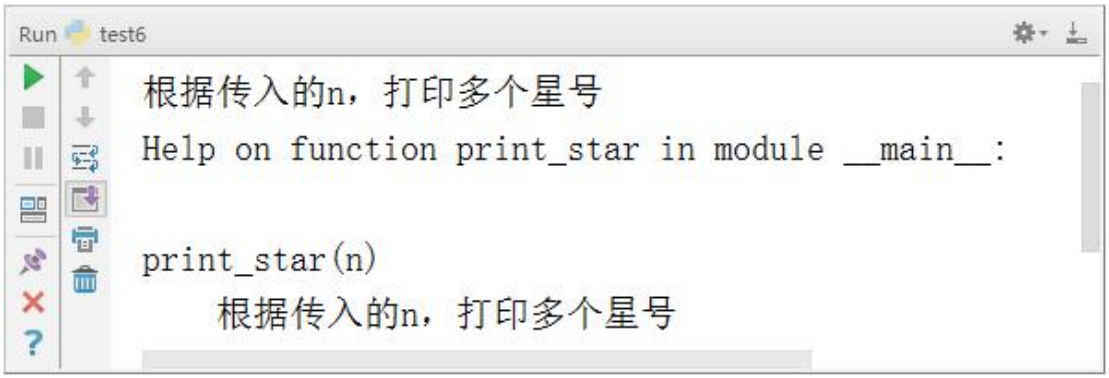
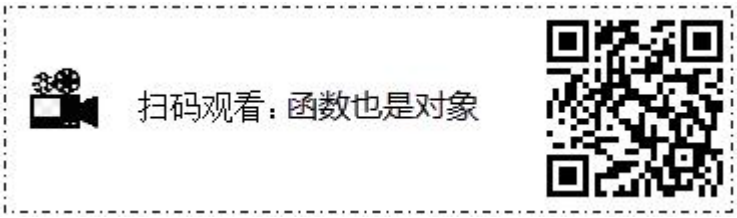


图 5-5 示例 5-6 执行结果

5.1.6 函数也是对象内存底层分析

Python 中，“一切都是对象”。实际上，执行 def 定义函数后，系统就创建了相应的函数对象。



【示例 5-7】函数也是对象内存底层分析

```
def print_star(n):
    print("*"*n)
print(print_star)
print('print_start 的 id：',id(print_star))
c = print_star
print('c 的 id：',id(c))
c(5)
```

执行结果如图 5-6 所示：

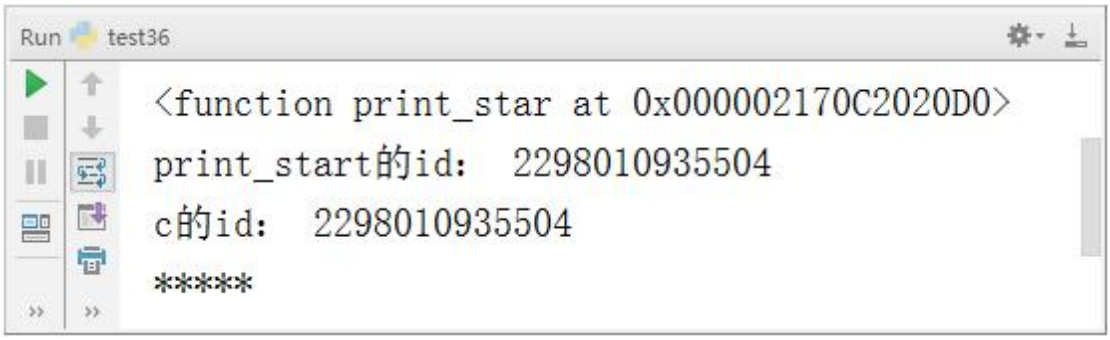


图 5-6 示例 5-7 执行结果

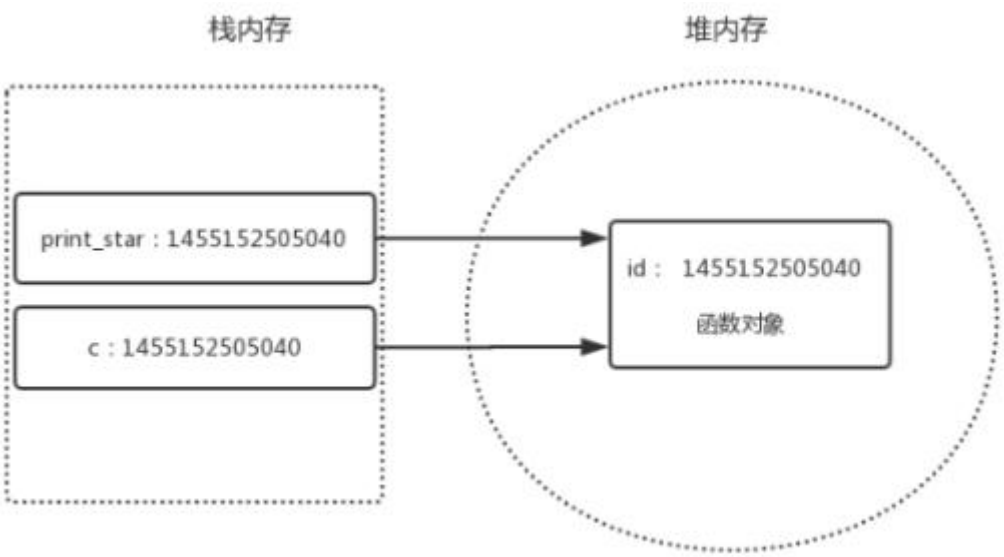
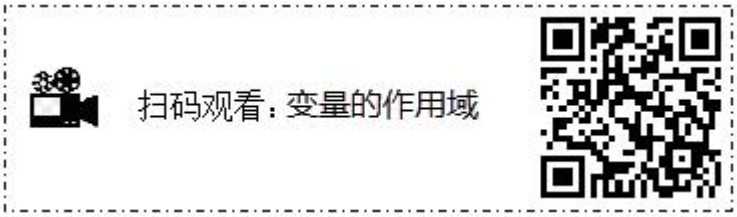


图 5-7 内存分析图

显然，可以看出变量 c 和 print_star 都是指向了同一个函数对象。因此，执行 c(3)和执行 print_star(3)的效果是完全一致的。 Python 中，圆括号意味着调用函数。在没有圆括号的情况下，Python 会把函数当做普通对象。

5.2 变量的作用域



一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里被赋值的。变量的作用域决定了在哪一部分程序可以访问那个特定的变量的名称。Python 语言中将不同作用范围的变量分为：局部变量、全局变量。

5.2.1 局部变量

局部变量是指变量声明在函数内部（包含形式参数），只有在特定的过程和函数中才可以访问的变量。

【示例 5-8】局部变量在函数内部访问

```
#测试局部变量
def fun1(x):
    y=100
    return x+y
#调用函数
print(fun1(200))
```

执行结果如图 5-8 所示：

图 5-8 示例 5-8 执行结果

【示例 5-9】局部变量在函数外部不能访问

```
#测试局部变量
def fun1(x):
    y=100
    return x+y
def fun2(x):
    return x+y
#调用函数
print(fun1(200))
print(fun2(200))
```

执行结果如图 5-9 所示：

图 5-9 示例 5-9 执行结果

由上面代码的运行结果得知：局部变量是声明在函数内部的变量，他的作用域在本函数内，出了该函数，其他的函数就访问不到，因此代码中的 fun2()方法会报错。

5.2.2 全局变量

由上面的示例可以看出局部变量不能被另一个函数访问，如果强制访问的话将会报错，为了解决这个变量共享的问题，在这里引进全局变量的概念，全局变量即在函数和类定义之外声明的变量。该变量供所有函数的调用，它的作用范围是整个程序。

【示例 5-10】测试全局变量

```
#定义全局变量
y=100
def fun1(x):
    return x+y
def fun2(x):
    return x+y
#调用函数
print(fun1(200))
print(fun2(200))
```

执行结果如图 5-10 所示：

图 5-10 示例 5-10 执行结果

5.2.3 global 关键字

全局变量的作用范围是所有的函数都可用使用此变量，函数内要改变全局变量的值，使用 global 关键字。

【示例 5-11】global 关键字

```
a = 100              #全局变量
def fun1():
    global a      #如果要在函数内改变全局变量的值，增加 global 关键字声明
    print(a)      #打印全局变量 a 的值
    a = 300       #修改全局变量的值
#调用函数
fun1()
print(a)
```

执行结果如图 5-11 所示：

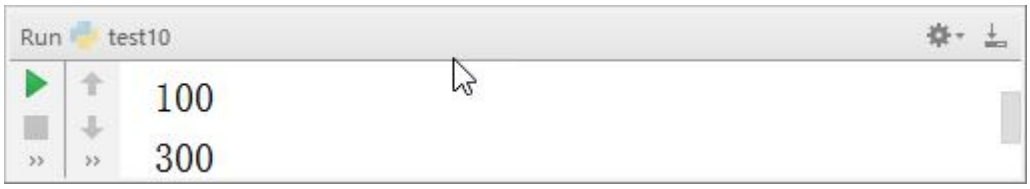


图 5-11 示例 5-11 执行结果

5.2.4 变量的就近原则

如果全局变量的名字和局部变量的名字相同，那么在函数调用变量的时候会采用 “就近原则”。

【示例 5-12】变量的就进原则

```
a = 100              #全局变量 a
def fun1():
    a=10             #局部变量 a
    print(a)     #打印变量 a 时候采用就近原则，则输出局部变量的 a
#调用函数
fun1()
```

执行结果如图 5-12 所示：

图 5-12 示例 5-12 执行结果

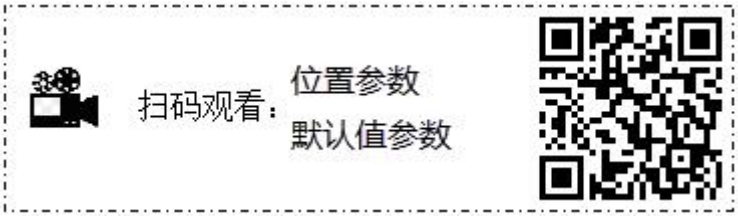
5.3 函数的参数

定义函数的时候，我们把参数的名字和位置确定下来，函数的接口定义就完成了。对于函数的调用者来说，只需要知道如何传递正确的参数，以及函数将返回什么样的值就够了，函数内部的复杂逻辑被封装起来，调用者无需了解。

Python 的函数定义非常简单，但灵活度却非常大。除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

5.3.1 位置参数

函数调用时，实参默认按位置顺序传递，需要个数和形参匹配。按位置传递的参数，



称为：“位置参数”。

【示例 5-13】位置参数

```
def fun1(a,b,c):
    print(a,b,c)
fun1(2,3,4)
fun1(3,4)          #报错，位置参数不匹配
```

执行结果如图 5-13 所示：

图 5-13 示例 5-13 执行结果

5.3.2 默认参数

在 Python 中，可以在声明函数时，预先为参数设置一个默认值。当调用函数时，如果某个参数具有默认值，则可以不向函数传递该参数，这时，函数将使用声明函数时为该参数设置的默认值。

声明一个参数具有默认值的语法如下：

```
def  函数名(参数=默认值):
    代码
```

【示例 5-14】默认参数

```
def fun1(a,b,c=10,d=20):
    print(a,b,c,d)
fun1(8,9)          #参数 c 和 d 使用默认值
fun1(8,9,19)       #参数 d 使用默认值
fun1(8,9,19,29)
```

执行结果如图 5-14 所示：

图 5-14 示例 5-14 执行结果

如果一个函数具有多个参数，并且这些参数都具有默认值，在调用函数的时候，可能仅想向最后一个参数传递值。示例如下：

【示例 5-15】向函数最后两个参数传递值测试一

```
def fun1(a=5,b=10,c=15,d=20):
    print(a,b,c,d)


fun1(,,19,29)        #只想向最后两个参数传递值
```

执行结果如图 5-15 所示：

图 5-15 示例 5-15 执行结果

从上面代码可以看出，在 Python 中，传递参数是按照声明函数时定义的参数的顺序依次传递的。如果在调用函数时使用且仅使用“，”表示向函数的最后一个参数传递值的话，则会引发错误。即默认值参数必须放到位置参数后面。如果需要向指定的参数传递值，则可以使用以下方式重新定义一下函数。

【示例 5-16】向函数最后两个参数传递值测试二

```
def fun1(a=None,b=None,c=None,d=None):
    if a==None:
        a=5
    if b==None:
        b=10
    if c==None:
```

```
        c=15
    if d==None:
        d=20
    print(a,b,c,d)
fun1(None,None,19,29)        #a 和 b 使用默认值，给最后两个参数 c 和 d 传递值
```

执行结果如图 5-16 所示：

图 5-16 示例 5-16 执行结果

5.3.3 关键字参数

在 Python 中，参数传递不仅仅可以按照声明函数时参数的顺序进行传递，还可以按照形参的名称传递参数，称为“命名参数”，也称“关键字参数”。

使用按形参的名称传递参数的方式调用函数时，要在调用函数名后的圆括号里为函数的所有参数赋值，赋值的顺序可以不必按照函数声明时的参数顺序。

【示例 5-17】关键字参数的使用

```
def fun1(a,b,c):
    print(a,b,c)


fun1(8,9,19)              #位置参数
fun1(c=10,a=20,b=30)   #命名参数，参数顺序可以和声明时的顺序不一致
```

执行结果如图 5-17 所示：

图 5-17 示例 5-17 执行结果

5.3.4 可变参数

在 Python 中，函数可以具有任意个参数，而不必在声明函数时对所有参数进行

定义。声明函数时，如果在参数名前加上一个星号“*”，则表示将多个参数收集到一个“元组”对象中；如果在参数名前加上两个星号“**”，则表示将多个参数收集到一个“字典”对象中。

【示例 5-18】可变参数的使用

```
def fun1(a,b,*c):            #函数参数前添加一个"*",可变参数以元组形式体现
    print(a,b,c)
fun1(8,9,19,20)

def fun2(a,b,**c):          #函数参数前添加两个"**",可变参数以字典形式体现
    print(a,b,c)
fun2(8,9,name='gaoqi',age=18)

def   fun3(a,b,*c,**d):
    print(a,b,c,d)

fun3(8,9,20,30,name='gaoqi',age=18)
```

执行结果如图 5-18 所示：

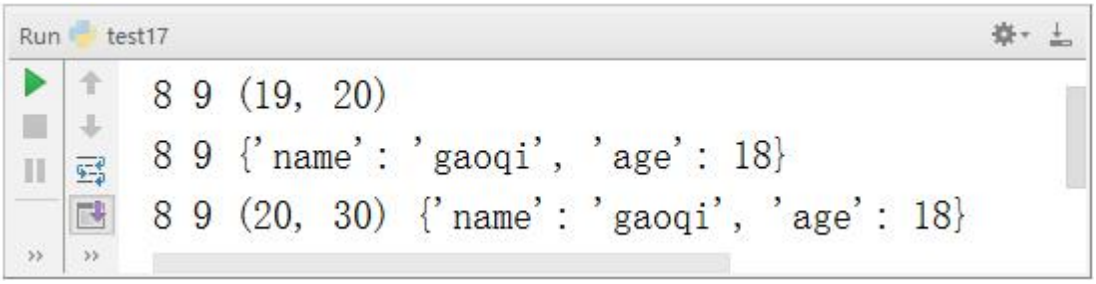


图 5-18 示例 5-18 执行结果

5.3.5 强制命名参数

使用可变参数需要考虑形参位置的问题。如果在函数中，既有普通参数，也有可变参数，通常可变参数会放在最后。如上面示例。其实可变参数也可以放在函数参数的中间或最前面，只是在调用函数时，可变参数后面的普通参数要使用关键字参数形式传递参数。

【示例 5-19】强制命名参数的使用一

```
def fun1(*a,b,c):
    print(a,b,c)
#f1(2,3,4)     #会报错。由于 a 是可变参数，将 2,3,4 全部收集。造成 b 和 c 没有赋值。
```

```
def fun2(a,*b,c):
    print(a,b,c)
fun1(2,b=3,c=4)   #可变参数放在最前面，普通参数使用关键字参数形式传递
fun2('hello',2,3,4,5,c='end')     #可变参数放在中间，普通参数使用关键字参数形式传递
```

执行结果如图 5-19 所示：

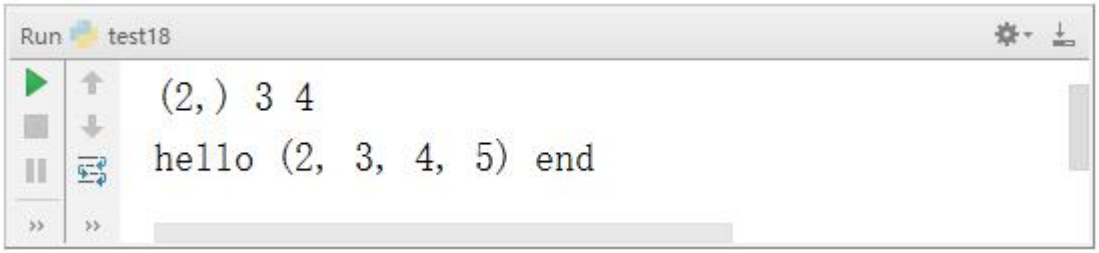


图 5-19 示例 5-19 执行结果

如果可变参数在函数参数的中间位置，而且在为可变参数后面的普通参数传值时也不想使用关键字参数，那么就必须为这些普通参数指定默认值。

【示例 5-20】强制命名参数的使用二

```
#可变参数放在中间位置，调用函数时候不想使用关键字参数形式传递参数，则普通参数必须有默认值
def fun1(a,*b,c=10,d=20):
    print(a,b,c,d)
fun1('hello','Python','I','Love','You')
```

执行结果如图 5-20 所示：

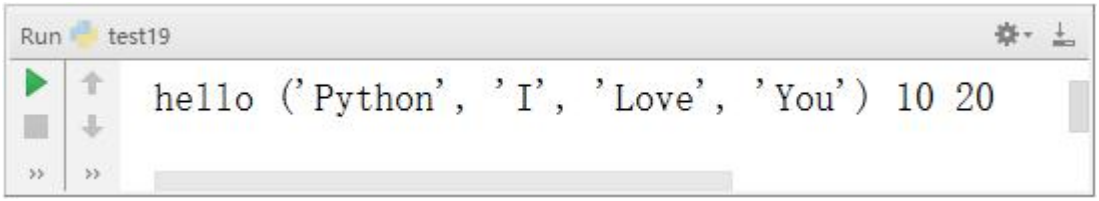


图 5-20 示例 5-20 执行结果

如果可变参数在函数参数的中间位置，并且没有为可变参数后面的普通形式参数指定默认值，在调用时也未使用关键字参数指定这些形参的值，那么调用函数时会抛出如图 5-18 所示的异常。

【示例 5-21】强制命名参数的使用三

```
def fun1(a,*b,c,d):
    print(a,b,c,d)
```

```
fun1('hello','Python','I','Love','You')
```

执行结果如图 5-21 所示：

图 5-21 示例 5-21 执行结果

5.4 参数的传递

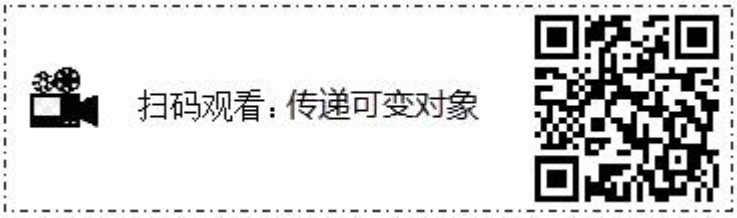
函数的参数传递本质上就是：从实参到形参的赋值操作。 Python 中“一切皆对象”，所有的赋值操作都是“引用的赋值”。所以，Python 中参数的传递都是“引用传递”，不是“值传递”。具体操作时分为两类：

1） 对“可变对象”进行“写操作”，直接作用于原对象本身。

2） 对“不可变对象”进行“写操作”，会产生一个新的“对象空间”，并用新的值填充这块空间。

5.4.1 传递可变对象的引用

传递参数是可变对象(例如：列表、字典、自定义的其他可变对象等)，实际传递的还是对象的引用。在函数体中不创建新的对象拷贝，而是可以直接修改所传递的对象。



【示例 5-22】参数传递：传递可变对象的引用

```
b = [10,20]
def fun2(m):
    print("m:",id(m))          #b 和 m 是同一个对象
    m.append(30)      #由于 m 是可变对象，不创建对象拷贝，直接修改这个对象
fun2(b)
print("b:",id(b))
print(b)
```

执行结果如图 5-22 所示：

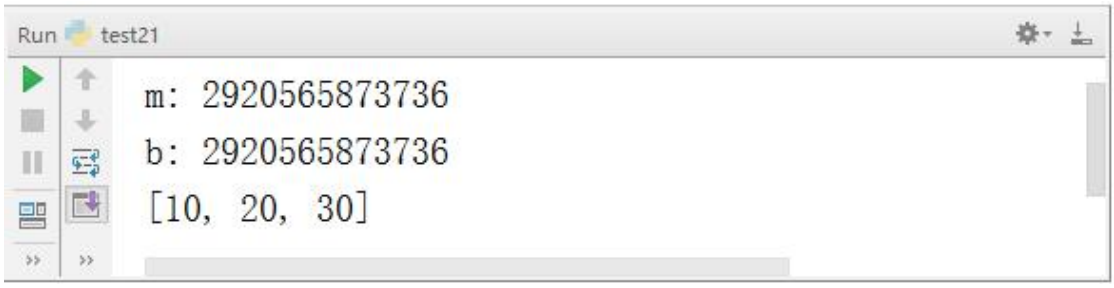
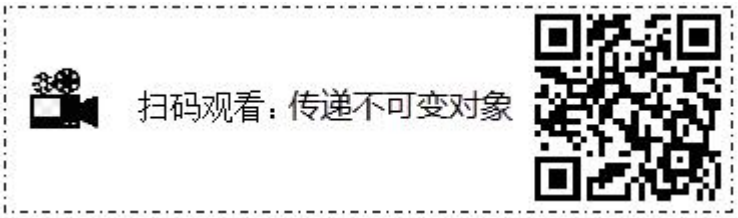


图 5-22 示例 5-22 执行结果

5.4.2 传递不可变对象的引用

传递参数是不可变对象（例如：int、float、字符串、元组、布尔值），实际传递的还是对象的引用。在”赋值操作”时，由于不可变对象无法修改，系统会新创建一个对象。



【示例 5-23】参数传递：传递不可变对象的引用

```
a = 100
def fun1(n):
    print("n 的 id:",id(n))          #传递进来的是 a 对象的地址
    n = n+200                        #由于 a 是不可变对象，因此创建新的对象 n
    print("n 的 id:",id(n))      #n 已经变成了新的对象
    print('n 的值:',n)
#调用函数
fun1(a)
print("a 的 id:",id(a))
print('a 的值',a)
```

执行结果如图 5-23 所示：

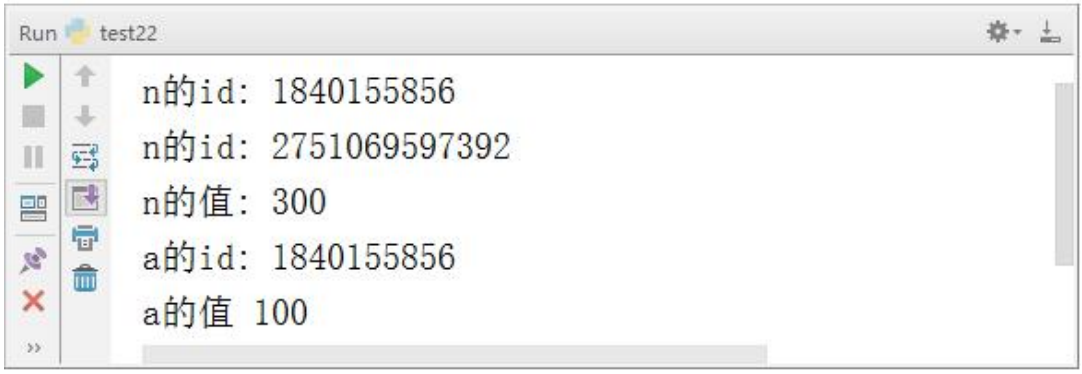
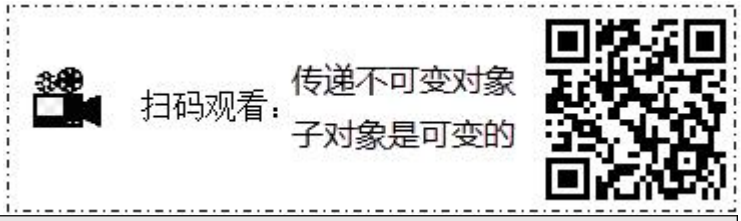


图 5-23 示例 5-23 执行结果

5.4.3 传递不可变对象包含的子对象是可变的情况

【示例 5-24】参数传递：参数是不可变对象包含的子对象是可变




```
a = (10,20,[5,6])
print("a 的 id:",id(a))
def test01(m):
    print("m 的 id:",id(m))
    m[2][0] = 888
    print('m 元素的值:',m)
    print("m 的 id:",id(m))
#调用函数
test01(a)
print('a 列表的值:',a)
```

执行结果如图 5-24 所示：

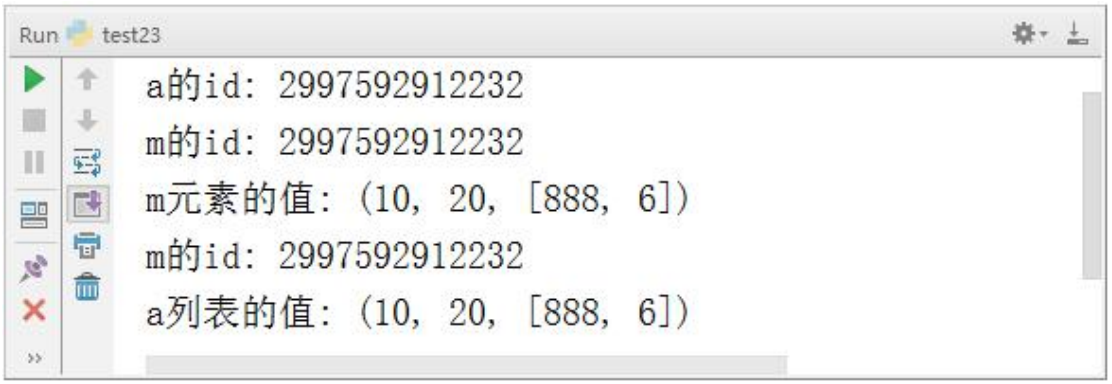
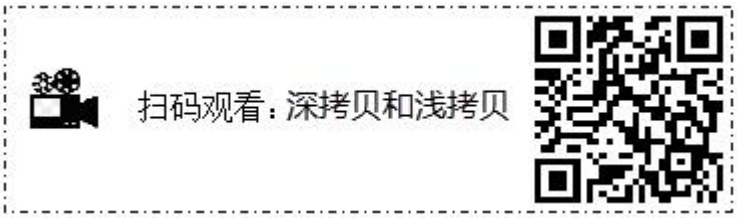


图 5-24 示例 5-24 执行结果

上面的代码可以看出，传递不可变对象时。不可变对象里面包含的子对象是可变的。则方法内修改了这个可变对象，原对象也发生了变化。

5.4.4 浅拷贝和深拷贝



Python 中提供了两个内置函数 copy()、deepcopy()来实现浅拷贝、深拷贝。其中浅拷贝指不拷贝子对象的内容，只是拷贝子对象的引用。而深拷贝会连子对象的内容也全部拷贝一份，对子对象的修改不会影响源对象。实现浅

拷贝和深拷贝需要使用 copy 模块，因此先导入 copy 模块。

【示例 5-25】浅拷贝

```
import copy
def testCopy():
    '''测试浅拷贝'''
    a = [10, 20, [5, 6]]
    b = copy.copy(a)

    print("a", a)
    print("b", b)
    b.append(30)
    b[2].append(7)
    print("浅拷贝.....")
    print("a", a)
    print("b", b)
testCopy()
```

执行结果如图 5-25 所示：

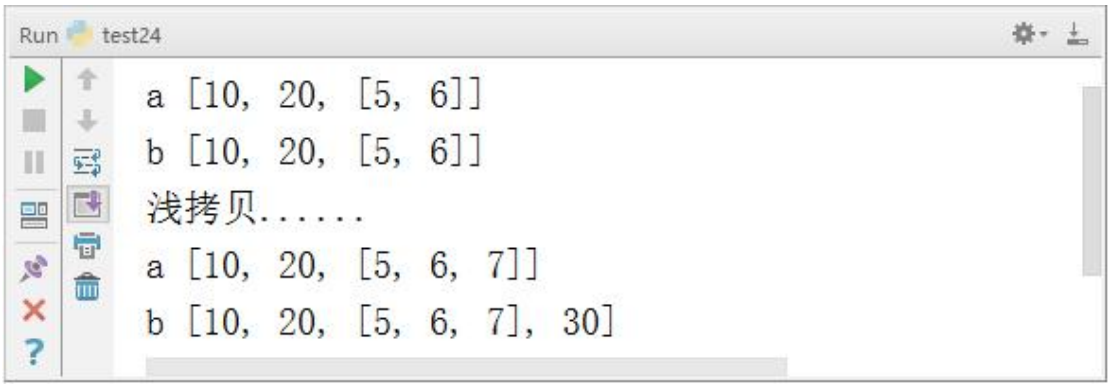


图 5-25 示例 5-25 执行结果

上面的代码可以看出，b = copy.copy(a)：浅拷贝后，a 和 b 是一个独立的对象，但他们的子对象还是指向同一个对象；也就是说，不管是修改 a 列表还是 b 列表中的子对象元素，a 列表和 b 列表子对象元素都会同时改变。所以在 b[2]中添加元素，a 列表元素也会发生改变。

【示例 5-26】深拷贝

```
import copy
def testDeepCopy():
```

```
    '''测试深拷贝'''
    a = [10, 20, [5, 6]]
    b = copy.deepcopy(a)

    print("a", a)
    print("b", b)
    b.append(30)
    b[2].append(7)
    print("深拷贝.....")
    print("a", a)
    print("b", b)
testDeepCopy()
```

执行结果如图 5-26 所示：

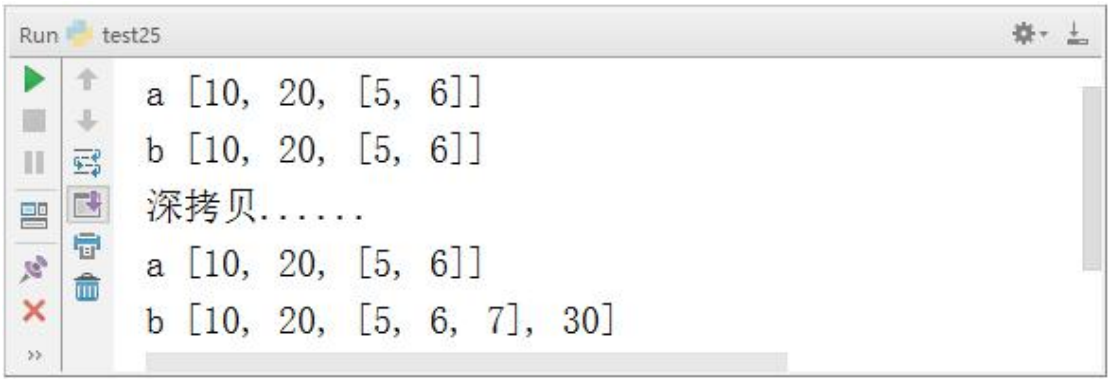


图 5-26 示例 5-26 执行结果

上面代码可以看出，b = copy.deepcopy(a): 深度拷贝后， a 和 b 完全拷贝了父对象及其子对象，两者是完全独立的。所以在 b[2]中添加元素，a 列表元素不会发生改变。

5.5 eval()函数

5.5.1eval() 函数实现 list、dict、tuple 与 str 之间的转化

【示例 5-27】eval()实现字符串转换成列表


扫码观看：eval()函数


```
a = "[[1,2], [3,4], [5,6], [7,8], [9,0]]"
print(type(a))              #查看 a 的类型
b = eval(a)                 #将字符串转换为列表
print(b)
```

执行结果如图 5-27 所示：

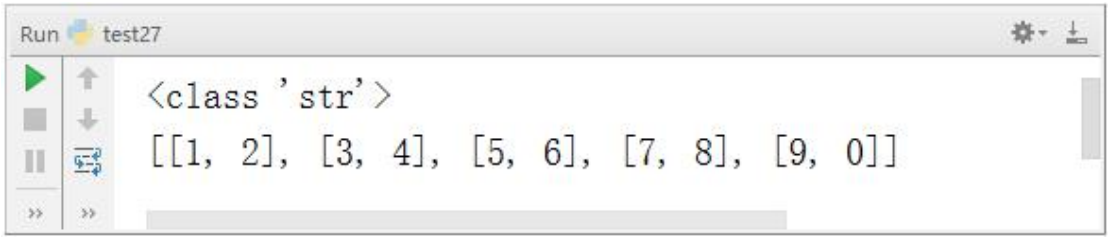


图 5-27 示例 5-27 执行结果

【示例 5-28】eval()实现字符串转换成字典

```
# 字符串转换成字典
a = "{1: 'a', 2: 'b'}"
print(type(a))
b = eval(a)
print(type(b))
print(b)
```

执行结果如图 5-28 所示：

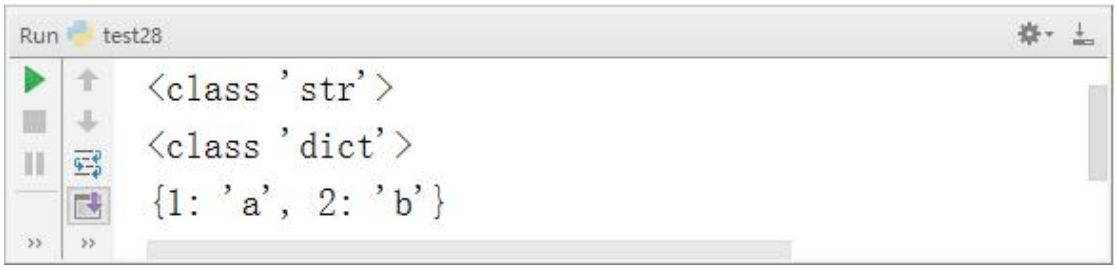


图 5-28 示例 5-28 执行结果

【示例 5-29】eval()实现字符串转换成元组

```
# 字符串转换成元组
a = "([1,2], [3,4], [5,6], [7,8], (9,0))"
print(type(a))
b=eval(a)
print(type(b))
print(b)
```

执行结果如图 5-29 所示：

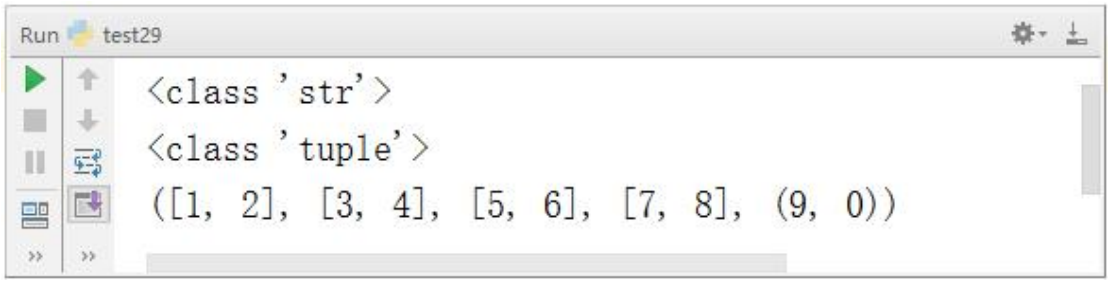


图 5-29 示例 5-29 执行结果

5.5.2eval() 函数用来执行一个字符串表达式，并返回表达式的值

eval() 函数用来执行一个字符串表达式，并返回表达式的值。语法格式如下：

```
eval(expression[, globals[, locals]])
```

其中 expression 是一个 Python 表达式，globals 是可选，如果被提供必须是 dictionary 字典对象，locals 也是可选的，如果被提供，可以是任何映射对象。

【示例 5-30】eval()函数执行一个字符串表达式

```
s=eval('2 + 2')      #将 string 变成算术表达式来执行
print(s)


a = 10
b = 20
c = eval("a+b")
print(c)


dict1 = dict(a=100,b=200)
d = eval("a+b",dict1)
print(d)
```

执行结果如图 5-30 所示：

图 5-30 示例 5-30 执行结果

eval 函数，eval 去除引号后会检查到它是不是可计算的，如果可计算会将计算的结果打印出来，如果不可计算直接返回结果。

【示例 5-31】eval()函数表达式不可计算直接返回结果

```
s='["a","b","c"]'
print(eval(s))
```

执行结果如图 5-31 所示：

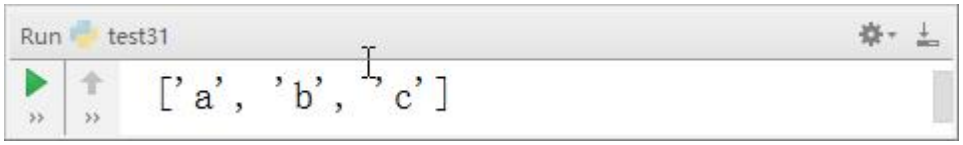


图 5-31 示例 5-31 执行结果

5.6 递归

所谓递归，就是在函数内部调用自身。在执行过程中，Python 解析器会利用栈处理递归函数返回的数据。所以递归函数的一个必要条件是要有终止条件，否则栈就会溢出。通过递归可以实现很多经典的算法，如阶乘、斐波那切数列等。

【示例 5-32】使用递归函数计算阶乘(factorial)

```
def factorial(n):
    if n==1:
        return 1
    return n*factorial(n-1)

#计算 1 到 5 的阶乘
for i in range(1,6):
    print(i,'!=',factorial(i))
```

执行结果如图 5-32 所示：

图 5-32 示例 5-32 执行结果

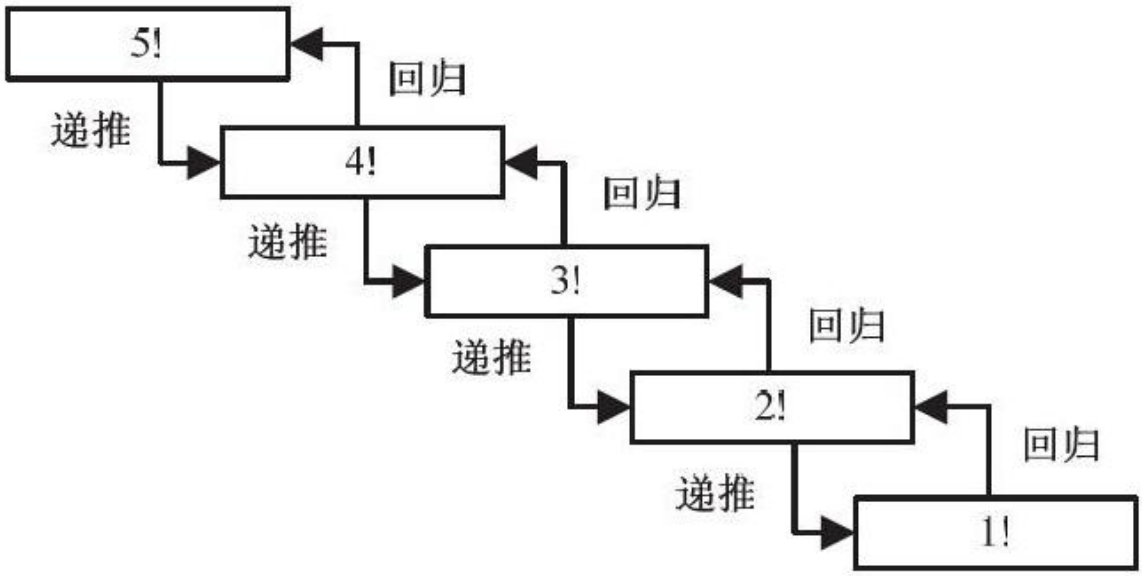


图 5-33 使用递归计算阶乘的过程

5.7 嵌套函数(内部函数)

函数的嵌套是指在函数的内部调用其他函数。C、C++只允许在函数体内部嵌套，而 Python 不仅仅支持函数体内嵌套，还支持函数定义的嵌套。

【示例 5-33】嵌套函数定义

```
def   fun1():
      print('fun1 running...')
      def fun2():                          #在函数 fun1 中定义函数 fun2
            print('fun2 running...')
      fun2()                               #调用函数 fun2
fun1()
```

执行结果如图 5-34 所示：

图 5-34 示例 5-33 执行结果

上面代码中可以看出，fun2()就是定义在 fun1 函数内部的函数。fun2()的定义和调用都在 fun1()函数内部。

【示例 5-34】嵌套函数：内部函数引用外部函数的变量

```
def  func():
    a=10
    b=20
    x=30
    y=40
    def sum():
        return a+b
    def sub():
        return y-x
    return sum()*sub()
#调用外部函数
print(func())
```

执行结果如图 5-35 所示：

图 5-35 示例 5-34 执行结果

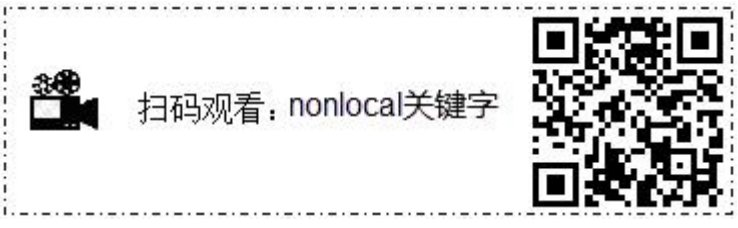
在上面的代码中可以看出，在内部函数 sum()和 sub()中没有任何参数，直接引用外部函数的变量。

注意：

- 函数的嵌套的层数不宜过多。否则，容易造成代码的可读性差、不易维护等问题。

5.8 nonlocal 关键字

内部函数可以引用外部



函数的变量，如果内部函数想改变外部函数变量的值，使用 nonlocal 关键字。

【示例 5-35】nonlocal 关键字

```
#测试 nonlocal、global 关键字的用法
a = 100          #全局变量
def outer():
    b = 10       #外部函数的变量
    def inner():
        nonlocal  b              #声明外部函数的局部变量
        print("inner b:",b)
        b = 20                   #对外部函数的变量重新赋值
        global a                 #声明全局变量
        a = 1000                 #对全局变量重新赋值
    inner()
    print("outer b:",b)
outer()
print("a：",a)
```

执行结果如图 5-36 所示：

图 5-36 示例 5-35 执行结果

习题

一、选择题

1. 语句 eval('2+4/5')执行后的输出结果是（）。

A．2.8

B．2

C．2+4/5

D．'2+4/5'

2. 下列属于 math 库中的数学函数的是（）。

A. time　B.　round()　C.　sqrt()　D.　random

3. 下面代码实现的功能描述的是

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)
num =eval(input(“请输入一个整数：”))
print(fact(abs(int(num))))
```

A. 接受用户输入的整数 n，判断 n 是否是素数并输出结论

B. 接受用户输入的整数 n，判断 n 是否是完数并输出结论

C. 接受用户输入的整数 n，判断 n 是否是水仙花数

D. 接受用户输入的整数 n，输出 n 的阶乘值

4. 下面代码的输出结果是：（）

```
ls = ["F",'f']
def fun(a):
    ls.append(a)
    return
fun('C.')
print(ls)
```

A. [‘F’, ‘f’]

B. [‘C’]

C. 出错

D. [‘F’, ‘f’, ‘C’]

5. 运行以下程序，当从键盘上输入{1:”清华大学”,2:”北京大学”}，运行结果的是：

```
x =eval(input())
```

```
print(type(x))
```

A. <class ‘int’>

B. <class ‘list’>

C. 出错

D. <class ‘dict’>

二、简答题

1. python 中定义函数时如何书写可变参数和默认参数。
2. *args 和**kwargs 在什么情况下会使用到？请给出使用**kwargs 的事例。
3. 递归的定义和优缺点。
4. 简述局部变量和全局变量的作用域。
5. 简述什么是深拷贝和浅拷贝。

三、编码题

1. 定义一个函数实现反响输出一个整数。比如：输入 3245，输出 5432。
2. 编写一个函数，计算下面的数列：
3. $$m(n)=\frac{1}{2}+\frac{2}{3}+...+\frac{n}{n+1}$$
4. 输入三角形三个顶点的坐标，若有效则计算三角形的面积；如坐标无效，则给出提示。
5. 输入一个毫秒数，将该数字换算成小时数，分钟数、秒数。
6. 使用海龟绘图。输入多个点，将这些点都两两相连。

第六章 面向对象编程

面向对象的程序设计提供了一种新的思维方式，软件设计的焦点不再是程序的逻辑流程，而是软件或程序中的对象以及对象之间的关系。使用面向对象的思想进行程序的设计，能够更好地设计软件架构，维护软件模块，并易于框架和组件的重用。

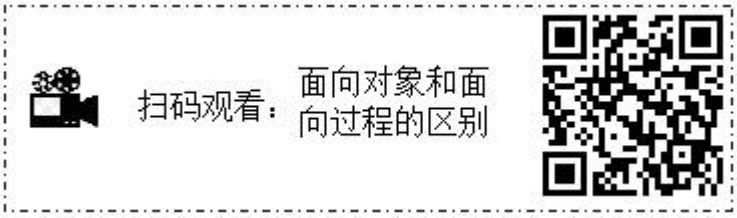
Python 是一种面向对象的编程语言，不过，Python 与 C++一样，还支持面向过程的程序设计。在 Python 中完全可以使用函数、模块等方式来完成工作。

通过阅读本章，你可以：

- 了解什么是对象、类及区别
- 了解面向过程和面向对象思想
- 掌握创建类的方法
- 掌握如何为类添加私有方法
- 掌握如何继承一个或者多个类(多继承)
- 掌握工厂模式和单例模式的实现

6.1 面向过程和面向对象思想

6.1.1 面向过程和面向对象的区别



面向过程和面向对象都是对软件分析、设计和开发的一种思想，它指导着人们以不同的方式去分析、设计和开发软件。早期先有面向过程思想，随着软件规模的扩大，问题复杂性的提高，面向过程的弊端越来越明显的显示出来，出现了面向对象思想并成为目前主流的方式。两者都贯穿于软件分析、设计和开发各个阶段，对应面向对象就分别称为面向对象分析（OOA）、面向对象设计（OOD）和面向对象编程（OOP）。C 语言是一种典型的面向过程语言，Java 是一种典型的面向对象语言。

面向过程思想思考问题时，首先思考“怎么按步骤实现？”并将步骤对应成方法，一步一步，最终完成。 这个适合简单任务，不需要过多协作的情况下。比如，如何开车？很容易就列出实现步骤：

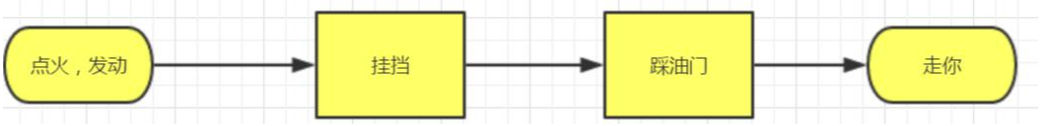


图 6-1 开车步骤

面向过程适合简单、不需要协作的事务，重点关注如何执行。

但是当我们思考比较复杂的设计任务时，比如“如何造车？”，就会发现列出 1234 这样的步骤，是不可能的。那是因为，造车太复杂，需要很多协作才能完成。此时面向对象思想

就应运而生了。

面向对象(Oriented-Object)思想更契合人的思维模式。首先思考的是“怎么设计这个事物？” 比如思考造车，就会先思考“车怎么设计？”，而不是“怎么按步骤造车的问题”。这就是思维方式的转变。

比如，用面向对象思想思考“如何设计车”：

图 6-2 车的结构

图 6-3 车的组成部件

天然的就会从“车由什么组成”开始思考。发现，车由如下对象组成：

为了便于协作，找轮胎厂完成制造轮胎的步骤，发动机厂完成制造发动机的步骤；这样，发现大家可以同时进行车的制造，最终进行组装，大大提高了效率。但是，具体到轮胎厂的一个流水线操作，仍然是有步骤的，还是离不开执行者、离不开面向过程思维！

因此，面向对象可以帮助我们从宏观上把握、从整体上分析整个系统。 但是，具体到实现部分的微观操作（就是一个个方法），仍然需要面向过程的思路去处理。

千万不要把面向过程和面向对象对立起来。他们是相辅相成的。面向对象离不开面向过程！

- **面向对象和面向过程思想的总结**
  - 都是解决问题的思维方式，都是代码组织的方式。

- 面向过程是一种“执行者思维”，解决简单问题可以使用面向过程。
- 面向对象是一种“设计者思维”，解决复杂、需要协作的问题可以使用面向对象。
- 面向对象离不开面向过程：
  - 宏观上：通过面向对象进行整体设计。
  - 微观上：执行和处理数据，仍然是面向过程。

6.1.2 面向对象是“设计者思维”

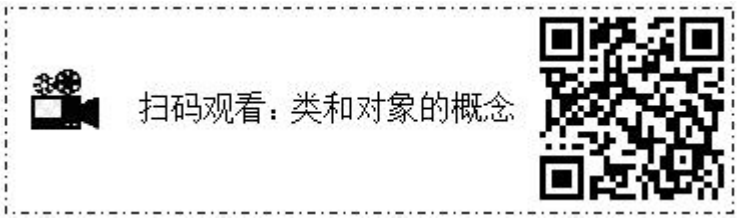
面向对象是一种“设计者思维”。设计时，先从问题中找名词，然后确立这些名词哪些可以作为类，再根据问题需求确定类中的属性和方法，确定类之间的关系。

例如设计一款企业管理软件，需要进行面向对象分析（OOA：Object-Oriented Analysis），写一首诗、一篇文章、一篇小说也需要进行面向对象分析。

因此，面向对象这种思维是任何人都需要学习、任何人都需要掌握的。

6.2 类和对象

6.2.1 类和对象的概念



我们人认识世界，其实就是面向对象的（此对象可不是男女谈对象的彼对象呀）。比如现在让大家认识一下“天使”这个新事物，天使大家没见过吧，怎么样认识呢？最好的办法就是，给你们面前摆 4 个天使，带翅膀的美女，让大家看，看完以后，即使我不说，大家下一次是不是就都认识天使了。

图 6-4 认识天使

但是，看完 10 个天使后，总要总结一下，什么样的东东才算天使？天使是无数的，总有没见过的！所以必须总结抽象，便于认识未知事物！总结的过程就是抽象的过程。小时候，学自然数时怎么定义的？像 1，2，3，4…这样的数就叫做自然数。 通过抽象，发现天使有如下特征：

1) 带翅膀（带翅膀不一定是天使，还可能是鸟人）
2) 女孩（天使掉下来脸着地，也是天使！）
3) 善良

4)　头上有光环

那么通过这 4 个具体的天使，我们进行抽象，抽象出了天使的特征，也可以归纳一个天使类。 因此类就是一些对象的抽象，隐藏了对象内部复杂的结构和实现。对象可以看成该类的一个具体实例。类是用于描述同一类型的对象的一个抽象概念，类中定义了这一类对象所应具有的共同的属性、方法。

6.2.2 类和对象的区别

类和对象是面向对象中的两个重要概念，初学者经常把类和对象混为一谈。类是对客观世界中事物的抽象，而对象是类实例化后的实体。实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。还可以把对象比作一个“饼干”，类就是制造这个饼干的“模具”。如图 6-5、6-6 所示。

图 6-5 饼干

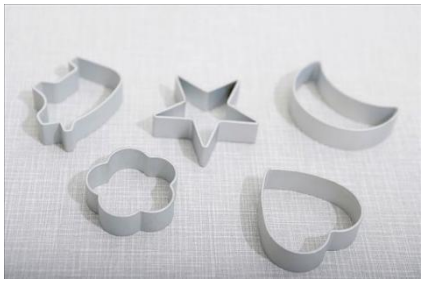


图 6-6 饼干模具

对象是具体的事物，是类的实例化结果。每个对象的属性值可能不同，但所有由同一类实例化得来的对象都拥有共同的属性和方法。在程序中，由类实例化生成对象，然后使用对象的方法进行操作，从而来完成任务。一个类可以实例化生成多个对象。

6.2.3 类的定义

Python 使用 class 关键字定义一个类，类名的首字母一般要大写。当程序员需要创建的类型不能用简单类型来表示时，则需要定义类，然后利用定义的类创建对象。类把需要使用的变量和方法组合在一起，这种方法称为封装。定义类的语法格式如下：

```
class　类名(父类名)：
    类体
```

其中圆括号中的父类名就是要继承的类。如果没有写默认继承的类是 object，关于继承将在后面详细讲解。此处定义一个 Student 类，示例如下：

【示例 6-1】类的定义

```
#定义 Student 类
```

```
class Student:
    pass
```

上面的代码中，pass 为空语句。就是表示什么都不做，只是作为一个占位符存在。没有任何的其他信息，就跟我们拿到一张张图纸，但是纸上没有任何信息，这是一个空类，没有任何实际意义。所以需要定义类的具体信息。对于一个类来说，一般有三种常见的成员：属性、方法、构造器。这三种成员都可以定义零个或多个。

【示例 6-2】一个典型的学生类

```
#定义 Student 类
class Student:
    def __init__(self,name,score):      #构造方法第一个参数必须为 self
        self.name = name                 #实例属性
        self.score = score
    def say_score(self):                 #实例方法
        print(self.name,'的分数是：',self.score)
s1 = Student('张三',80)                  #s1 是实例对象，自动调用__init__()方法
s1.say_score()
```

执行结果如图 6-7 所示：

图 6-7 示例 6-2 执行结果

6.2.4 创建对象

创建对象的过程称为实例化。创建对象，需要定义构造方法__init__()。构造方法用于执行“实例对象的初始化工作”，即对象创建后，初始化当前对象的相关属性，无返回值。

__init__()的要点如下：

- 构造方法的名称是固定的，必须为：__init__()。
- 第一个参数固定，必须为：self。 self 指的就是刚刚创建好的实例对象。
- 构造函数通常用来初始化实例对象的实例属性，如示例 6-2 中的构造方法就是初始

化实例属性：name 和 score。

- 通过“类名(参数列表)”来调用构造函数。调用后，将创建好的对象返回给相应的变量。 例如：s1 = Student('张三', 80)
- 如果不定义__init__方法，系统会提供一个默认的__init__方法。如果定义了带参的__init__方法，系统不创建默认的__init__方法。
- 如果定义了__init__方法，在创建实例的时候，就不能传入空的参数了，必须传入与__init__方法匹配的参数，但 self 不需要传，Python 解释器自己会把实例变量传进去。

【示例 6-3】创建对象

```
#定义 Student 类
class Student:
    def __init__(self,name,score):      #构造方法第一个参数必须为 self
        self.name = name                 #实例属性
        self.score = score
#创建对象 s1，调用__init__()方法，参数匹配
s1 = Student('张三',80)
print(s1)
s2 =Student()    #参数不匹配，报错
```

执行结果如图 6-8 所示：

图 6-8 示例 6-3 执行结果

从上面的代码中可以看到实际上生成了一个变量名就是类名“Student”的对象。通过赋值给新变量 s1，说明，确实创建了“类对象”。

注意：

Python 中的 self 相当于 C++中的 self 指针，JAVA 和 C#中的 this 关键字。Python 中，self 必须为构造函数的第一个参数，名字可以任意修改。但一般遵守惯例，都叫做 self。

6.3 属性和方法

每一个类都具有自己的属性和方法。属性和方法是面向对象程序设计所独有的概念。属性是类所封装的数据，而方法则是类对数据进行的操作。

6.3.1 实例属性

实例属性是从属于实例对象的属性，也称为“实例变量”。实例属性一般定义在__init__()方法中，语法格式如下：


```
self.实例属性名 = 初始值
```

在本类的其他实例方法中，可以通过 self 进行访问：

```
self.实例属性名
```

创建实例对象后，通过实例对象访问：

```
obj01 = 类名()          #创建对象，调用__init__()初始化属性
obj01.实例属性名 = 值      #可以给已有属性赋值，也可以新加属性
```

【示例 6-4】实例属性的定义及访问

```
#定义 Student 类
class Student:
    def __init__(self,name,score):      #构造方法第一个参数必须为 self
        self.name = name                #实例属性
        self.score = score
    def say_score(self):                #实例方法
        #在实例方法中访问实例属性
        print(self.name,'的分数是：',self.score)
s1 = Student('张三',80)                 #s1 是实例对象，自动调用__init__()方法
s1.say_score()                          #调用实例方法
#给已有实例属性重新赋值
s1.name='李四'
s1.score=98
print(s1.name,'的分数是:',s1.score)
```

执行结果如图 6-9 所示：

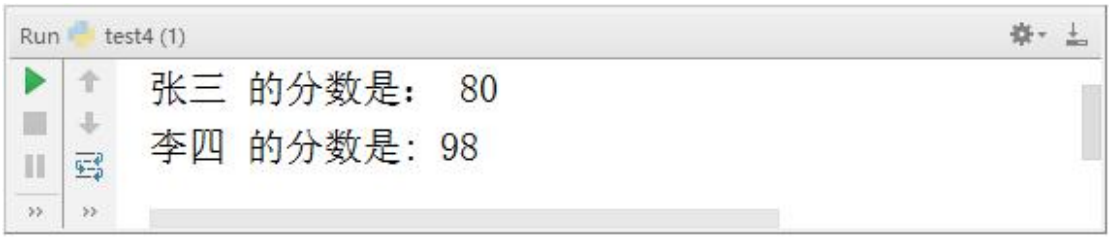


图 6-9 示例 6-4 执行结果

6.3.2 实例方法


扫码观看：实例方法

实例方法是从属于实例对象的方法。实例方法的定义语法格式如下：

```
def  方法名(self [, 形参列表])：
    方法体
```

方法的调用格式如下：

```
对象.方法名([实参列表])
```

【示例 6-5】实例方法的定义及访问

```
class Student:
    def __init__(self,name,score):
        self.name = name              #实例属性
        self.score = score
    def say_score(self):              #实例方法
        print('{0}的分数是{1}'.format(self.name,self.score))
s1 = Student('张三',80)               #s1 是实例对象，自动调用__init__()方法
#调用实例方法
s1.say_score()
s2 = Student('李四',89)               #s2 是实例对象，自动调用__init__()方法
#调用实例方法
s2.say_score()
```

执行结果如图 6-10 所示：

图 6-10 示例 6-5 执行结果

注意：

- 定义实例方法时，第一个参数必须为 self。和前面一样，self 指当前的实例对象。
- 调用实例方法时，不需要也不能给 self 传参。self 由解释器自动传参。

函数和方法的区别：

1) 都是用来完成一个功能的语句块，本质一样。

2) 方法调用时，通过对象来调用。方法从属于特定实例对象，普通函数没有这个特点。

3) 直观上看，方法定义时需要传递 self，函数不需要。

实例方法调用本质：

实例对象调用实例方法时候本质是类进行调用该方法，将实例对象作为参数传进去。因为定义实例方法的代码就在类中，被所有实例对象共享。

图 6-11 实例方法调用本质

6.3.3 类属性

直接在 class 中定义的属性就是类属性，实际上就是类内部的变量。定义语法格式如下：


```
class 类名：
    类变量名= 初始值
```

在类中或者类的外面，可以通过：“类名.类变量名”来读写。

【示例 6-6】类属性的使用

```
class Student:
    company = "尚学堂"   # 类属性
    count = 0   # 类属性

    def __init__(self, name, score):
        self.name = name   # 实例属性
        self.score = score
        Student.count = Student.count + 1

    def say_score(self):   # 实例方法
        print("我的公司是：", Student.company)
```

```
            print(self.name, '的分数是：', self.score)


s1 = Student('高淇', 80)   # s1 是实例对象，自动调用__init__()方法
s1.say_score()
print('一共创建{0}个 Student 对象'.format(Student.count))
```

执行结果如图 6-12 所示：

图 6-12 示例 6-6 执行结果

通过内存分析上述示例中实例对象和类对象创建过程，如图 6-13 所示：

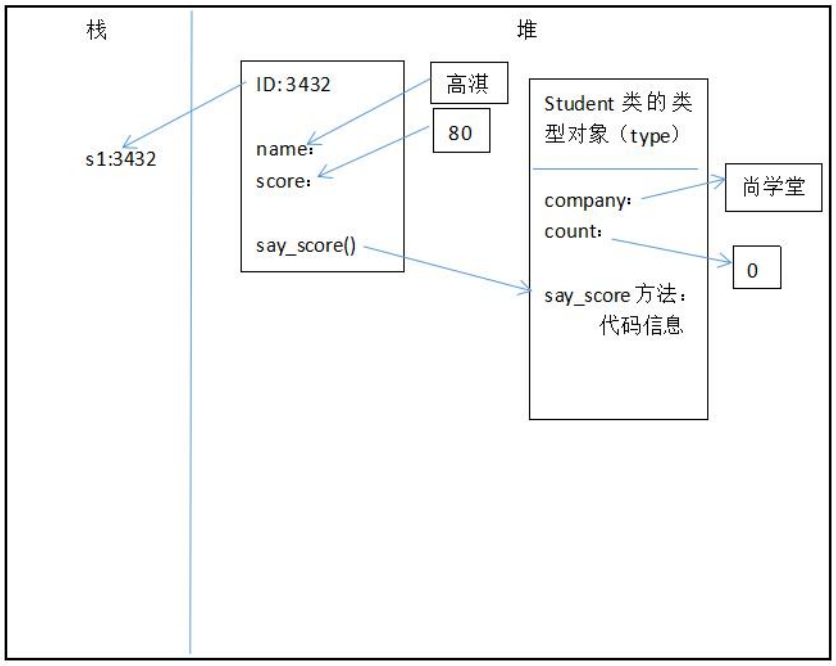


图 6-13 实例对象和类对象创建过程

6.3.4 类方法

类方法是从属于“类对象”的方法。类方法通过装饰器@classmethod 来定义，语法格式如下：

```
@classmethod
def 类方法名(cls [，形参列表]) ：
    函数体
```

其中类方法定义的要点如下：

- @classmethod 必须位于方法上面一行。
- 第一个 cls 必须有，cls 指的就是“类对象”本身。
- 调用类方法格式：“类名.类方法名(参数列表)”。 参数列表中，不需要也不能给 cls 传值。
- 类方法中访问实例属性和实例方法会导致错误。
- 子类继承父类方法时，传入 cls 是子类对象，而非父类对象。

【示例 6-7】类方法的使用

```
class Student:
    company = "SXT"        #类属性
    @classmethod
    def printCompany(cls):
        print(cls.company)
Student.printCompany()
```

执行结果如图 6-14 所示：

图 6-14 示例 6-7 执行结果

6.3.5 静态方法

Python 中允许定义与“类对象”无关的方法，称为“静态方法”。“静态方法”和在模块中定义普通函数没有区别，只不过“静态方法”放到了“类的名字空间里面”，需要通过“类调用”。静态方法通过装饰器@staticmethod 来定义，语法格式如下：

```
@staticmethod
def 静态方法名([形参列表]) ：
    函数体
```

【示例 6-8】静态方法的使用

```
class Student:
```

```
    company = "SXT"    # 类属性
    @staticmethod
    def add(a, b):    # 静态方法
        print("{0}+{1}={2}".format(a,b,(a+b)))
        return a+b
Student.add(20,30)
```

执行结果如图 6-15 所示：

图 6-15 示例 6-8 执行结果

6.3.6 内置方法

Python 类定义了一些专用的方法，这些专用的方法丰富了程序设计的功能，用于不同的应用场合。如表 6-1 所示：

表 6-1 常用内置方法

内置方法	说明
__init__(self,...)	初始化对象，在创建新对象时调用
__del__(self)	释放对象，在对象被删除之前调用
__new__(cls,*args,**kwd)	实例的生成操作
__str__(self)	在使用 print 语句时被调用
__getitem__(self,key)	获取序列的索引 key 对应的值，等价于 seq[key]
__len__(self)	在调用内联函数 len()时被调用
内置方法	**说明**
__cmp__(stc,dst)	比较两个对象 src 和 dst
__getattr__(s,name)	获取属性的值
__setattr__(s,name,value)	设置属性的值
__delattr__(s,name)	删除 name 属性
__getattribute__()	__getattribute__()功能与__getattr__()类似
__gt__(self,other)	判断 self 对象是否大于 other 对象
__lt__(slef,other)	判断 self 对象是否小于 other 对象
__ge__(slef,other)	判断 self 对象是否大于或者等于 other 对象

__le__(slef,other)	判断 self 对象是否小于或者等于 other 对象
__eq__(slef,other)	判断 self 对象是否等于 other 对象
__call__(self,*args)	把实例对象作为函数调用

(1)__del__方法(析构函数)

析构函数用于释放对象占用的资源。Python 提供了析构函数__del__()。析构函数可以显示释放对象占用的资源，是另一种释放资源的方法。析构函数也是可选的。如果程序中不提供析构函数，Python 会在后台提供默认的析构函数。由于 Python 中定义了__del__()的实例将无法被 Python 循环垃圾收集器（gc）收集，所以建议只有在需要时才定义__del__()。下面演示 Python 的函数__del__()的使用。

【示例 6-9】函数__del__()的使用

```
#析构函数
class Person:
    def _del_(self):
        print("销毁对象：{0}".format(self))
p1 = Person()
p2 = Person()
del p2
print("程序结束")
```

执行结果如图 6-16 所示：

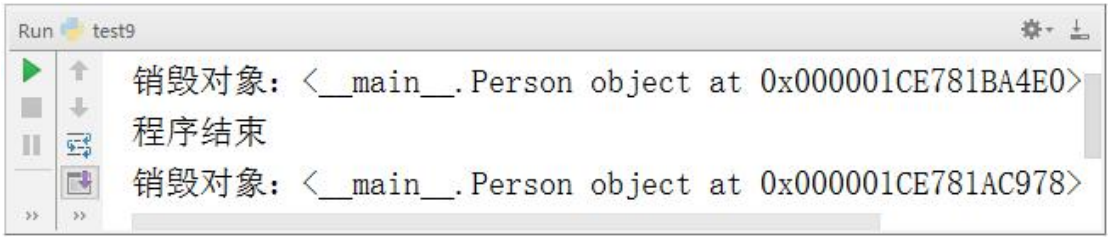


图 6-16 示例 6-9 执行结果

(2) __str__()

在 python 中，使用 python 输出对象变量，默认情况下，输出变量的引用对象是有哪一个类创建的，以及该对象在内存中的十六进制的地址。如果希望使用 print 输出对象的描述信息，且能够打印自定义内容，可以使用 str 内置方法。

【示例 6-10】函数__str__()的使用

```
class Student:
    def _init_(self,name,score):
        self.name = name              #实例属性
```

```
            self.score = score
    def __str__(self):
        return '{0}的分数是{1}'.format(self.name,self.score)   #返回一个字符串
s1 = Student('张三',80)                  #s1 是实例对象，自动调用__init__()方法
print(s1) #输出 s1 对象时候，调用__str__()方法
```

执行结果如图 6-17 所示：

图 6-17 示例 6-10 执行结果

(3) __call__()

定义了__call__方法的对象，称为“可调用对象”，即该对象可以像函数一样被调用。

【示例 6-11】函数__call__()的使用

```
#测试__call__，可调用对象
class SalaryAccount:
    '''工资计算类'''
    def __call__(self, salary):
        yearSalary = salary*12
        daySalary = salary//30
        hourSalary = daySalary//8
        return
dict(monthSalary=salary,yearSalary=yearSalary,daySalary=daySalary,hourSalary=hourSalary
)
s = SalaryAccount()
print(s(5000))          #可以像调用函数一样调用对象的__call__方法
```

执行结果如图 6-18 所示：

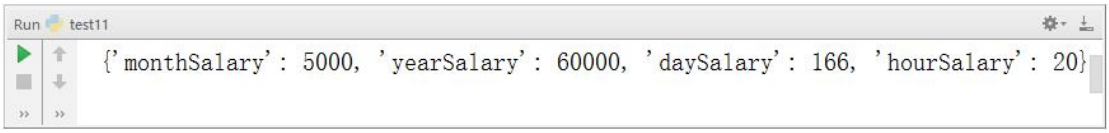
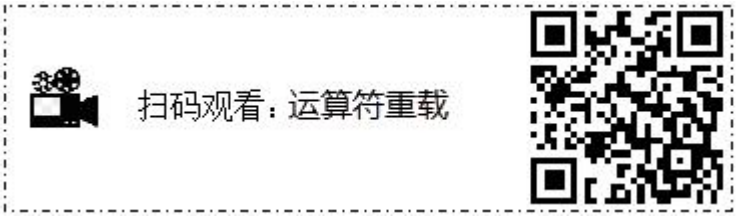


图 6-18 示例 6-11 执行结果

6.3.7 运算符重载

由于在 Python 中，运



算符都有其对应的函数。因此在 Python 中，运算符重载不需要像在 C++中那样使用 operator 关键字。在类中，运算符对应类中的一些专有方法。因此运算符的重载实际上是对运算符对应的专有方法的重载。部分运算符和类的专有方法名对照如表 6-2 所示：

表 6-2 运算符与类专有方法对照表

运算符	特殊方法	说明
+	__add__	加法
-	__sub__	减法
<,<=,==	__lt__,__le__，__eq__	比较运算符
>,>=,!=	__gt__,__ge__,__ne__	
|,^,&	__or__,__xor__,__and__	或、异或、与
<<,>>	__lshift__,__rshift__	左移、右移
*,/,%,//	__mul__,__truediv__,__mod__,__floordiv__	乘、浮点除、模运算（取余）、整数除
**	__pow__	指数运算

【示例 6-12】函数__add__ ()的使用

```
a = 20
b = 30
c = a+b
d = a.__add__(b)
print("c=",c)
print("d=",d)
```

执行结果如图 6-19 所示：

图 6-19 示例 6-12 执行结果

【示例 6-13】运算符的重载

```
#测试运算符的重载
class Person:
    def __init__(self,name):
        self.name = name
```

```
    def __add__(self, other):
        if isinstance(other,Person):
            return "{0}--{1}".format(self.name,other.name)
        else:
            return "不是同类对象，不能相加"
    def __mul__(self, other):
        if isinstance(other,int):
            return    self.name*other
        else:
            return "不是同类对象，不能相乘"
p1 = Person("高淇")
p2 = Person("高希希")
x = p1 + p2       #调用 __add__()方法
print(x)
print(p1*3)       #调用__mul__()方法
```

执行结果如图 6-20 所示：

图 6-20 示例 6-13 执行结果

6.3.8 私有属性和私有方法

Python 对于类的成员没有严格的访问控制限制，这与其他面向对象语言有区别。关于私有属性和私有方法，有如下要点：

1. 通常约定，两个下划线开头的属性是私有的(private)。其他为公共的(public)。
2. 类内部可以访问私有属性(方法)。
3. 类外部不能直接访问私有属性(方法)。
4. 类外部可以通过“_类名__私有属性(方法)名”访问私有属性(方法)。

【示例 6-14】私有属性和私有方法使用

```
#测试私有属性、私有方法
```

```
class Employee:
    __company = "百战程序员"          #私有类属性.
    def __init__(self,name,age):
        self.name = name
        self.__age = age              #私有实例属性
    def say_company(self):
        #类内部可以直接访问私有属性
        print("我的公司是：",Employee.__company)
        print(self.name,"的年龄是：",self.__age)
        self.__work()                 #类内部可以直接访问私有方法
    def __work(self):                 #私有实例方法
        print("工作！好好工作！")
p1 = Employee("高淇",32)
print(p1.name)
p1.say_company()
print(p1._Employee__age)   #通过这种方式可以直接访问到私有属性
# print(p1.__age)                     #直接访问私有属性，报错
# p1.__work()                         #直接访问私有方法，报错
```

执行结果如图 6-21 所示：

图 6-21 示例 6-14 执行结果

6.3.9 方法没有重载

Python 中，方法的的参数没有声明类型（调用时确定参数的类型），参数的数量也可以由可变参数控制。因此，Python 中是没有方法的重载的。定义一个方法即可有多种调用方式，相当于实现了其他语言中的方法的重载。

如果在类体中定义了多个重名的方法，只有最后一个方法有效。

【示例 6-15】多个重名方法的测试

```
#Python 中没有方法的重载。定义多个同名方法，只有最后一个有效
class Person:
    def say_hi(self):
        print("hello")
    def say_hi(self,name,age):
        print("姓名:{0},年龄:{1}".format(name,age))
p1 = Person()
p1.say_hi("张三",23)
p1.say_hi()          #不带参，报错
```

执行结果如图 6-22 所示：

图 6-22 示例 6-15 执行结果

6.4 获取对象信息

6.4.1 type()函数

在 python 中一切类型皆对象，如果想知道一个对象是什么类型可以使用 type()函数。

【示例 6-16】type()函数使用

```
print(type(123))
print(type('123'))
print(type(None))
```

执行结果如图 6-23 所示：

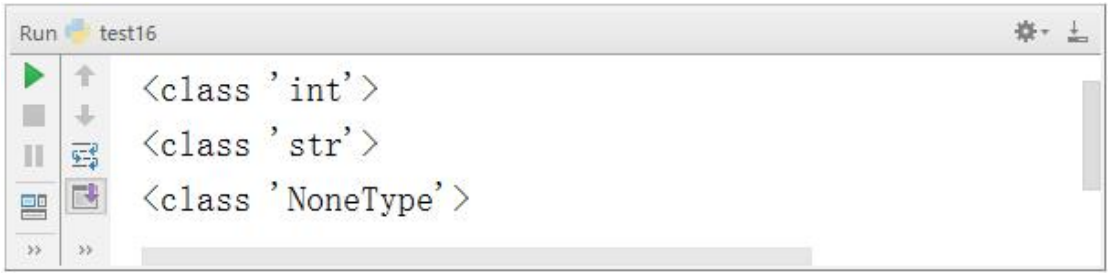


图 6-23 示例 6-16 执行结果

6.4.2 isinstance()函数

isinstance() 函数来判断一个对象是否是一个已知的类型，类似 type()。isinstance() 方法的语法如下：

```
isinstance(object, classinfo)
```

其中第一个参数（object）为实例对象，第二个参数（type）可以是直接或间接类名、基本类型或者由它们组成的元组。其返回值为布尔型（True or Flase）。

若对象的类型与参数二的类型相同则返回 True。若参数二为一个元组，则若对象类型与元组中类型名之一相同即返回 True。

【示例 6-17】isinstance 函数的使用

```
a=5
print(isinstance(a,int))        #True
print(isinstance(a,str))        #False
print(isinstance(a,(str,int,list)))    #是元组中的一个返回 True
```

执行结果如图 6-24 所示：

图 6-24 示例 6-17 执行结果

注意：

- type() 不会认为子类是一种父类类型，不考虑继承关系。isinstance() 会认为子类是一种父类类型，考虑继承关系。如果要判断两个类型是否相同推荐使用 isinstance()。

【示例 6-18】type() 与 isinstance()区别

```
class A:
    pass
class B(A):#继承 A
    pass
print(isinstance(A(), A))       #执行结果 True
print(type(A()) == A)           #执行结果 True
```

```
print(isinstance(B(), A))        #执行结果 True
print(type(B()) == A )           #执行结果 False
```

执行结果如图 6-25 所示：

图 6-25 示例 6-18 执行结果

6.4.3 dir()函数

dir() 函数不带参数时，返回当前范围内的变量、方法和定义的类型列表；带参数时，返回参数的属性、方法列表。如果参数包含方法__dir__()，该方法将被调用。如果参数不包含__dir__()，该方法将最大限度地收集参数信息。

【示例 6-19】dir()函数的使用

```
print(dir('abc'))
```

执行结果如图 6-26 所示：

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

图 6-26 示例 6-19 执行结果

6.5 @property 装饰器

在绑定属性时，如果直接把属性暴露出去，虽然写起来很简单，但是，没办法检查参

数，导致随便可以修改成绩。示例代码如下：

```
class Student:
    pass
s=Student()
s.score=99
```

这显然不合逻辑。为了限制 score 的范围，可以通过一个 set_score()方法来设置成绩，再通过一个 get_score()来获取成绩，这样，在 set_score()方法里，就可以检查参数，示例如下：

【示例 6-20】get 和 set 方法

```
class Student(object):
    def get_score(self):
        return self.__score
    def set_score(self, value):
        if not isinstance(value, int):
            print('成绩必须是整数')
        else:
            if value < 0 or value > 100:
                print('成绩必须在 0-100 之间')
            else:
                self.__score = value
s = Student()
s.set_score(60)
print('获取的成绩是:',s.get_score())
s.set_score(1000)
s.set_score('abc')
```

执行结果如图 27 所示：

图 6-27 示例 6-20 执行结果

但是，上面的调用方法又略显复杂，没有直接用属性这么直接简单。在 Python 中可以使用@property 可以将一个方法的调用方式变成“属性调用”。示例代码如下：

【示例 6-21】@property 装饰器的使用

```
class Student(object):
    @property
    def score(self):
        return self.__score
    @score.setter
    def score(self, value):
        if not isinstance(value, int):
            print('成绩必须是整数')
        else:
            if value < 0 or value > 100:
                print('成绩必须在 0-100 之间')
            else:
                self.__score = value
s = Student()
s.score=60          #实际转化为 s.set_score(60)
print('获取的成绩是:',s.score)     #实际转化为 s.get_score()
s.score=1000
s.score='abc'
```

执行结果如图 6-28 所示：

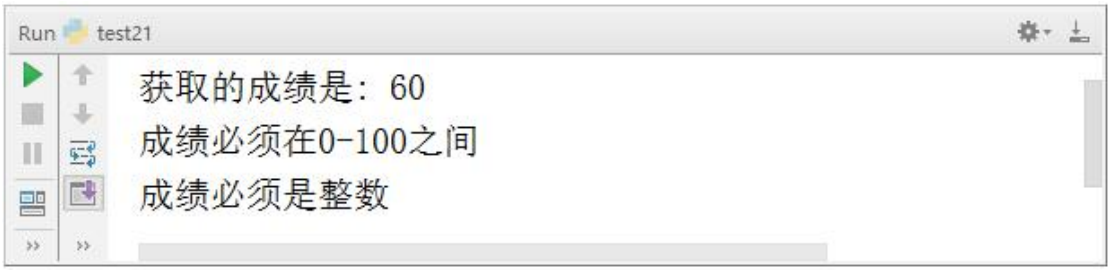


图 6-28 示例 6-21 执行结果

通过使用@property，可以对实例属性不直接暴露，如果想定义只读属性，则只定义 getter 方法，不定义 setter 方法就是一个只读属性，示例代码如下：

【示例 6-22】定义只读属性

```
import datetime
class Student(object):
    @property
    def birth(self):
```

```
        return self.__birth
    @birth.setter
    def birth(self, value):
        self.__birth = value
    @property
    def age(self):
        #获取当前年份
        now_year=datetime.datetime.now().strftime('%Y')
        return int(now_year) - int(self.__birth)
s=Student()
s.birth='1987'                  #设置出生年份
print('出生日期:',s.birth)      #获取出生年份
print('年龄:',s.age)            #获取年龄
```

执行结果如图 6-29 所示：

图 6-29 示例 6-22 执行结果

上面代码中的 birth 是可读写属性，而 age 就是一个只读属性，因为 age 是根据 birth 和当前时间获取而来。

6.6 面向对象三大特征

Python 是面向对象的语言，也支持面向对象编程的三大特性：继承、封装（隐藏）多态。

(1) 封装（隐藏）

隐藏对象的属性和实现细节，只对外提供必要的方法。相当于将“细节封装起来”，只对外暴露“相关调用方法”。

通过前面学习的“私有属性、私有方法”的方式，实现“封装”。Python 追求简洁的语法，没有严格的语法级别的“访问控制符”，更多的是依靠程序员自觉实现。

(2) 继承

继承可以让子类具有父类的特性，提高了代码的重用性。

从设计上是一种增量进化，原有父类设计不变的情况下，可以增加新的功能，或者改进

已有的算法。

(3) 多态

多态是指同一个方法调用由于对象不同会产生不同的行为。生活中这样的例子比比皆是：同样是休息方法，人不同休息方法不同。张三休息是睡觉，李四休息是玩游戏，程序员休息是“敲几行代码”。

6.7 类的继承

6.7.1 继承的实现

与其他面向对象的编程语言一样，Python 也支持类的继承。所谓类的继承，就是一个类(子类)从另外一个类(父类)获得了所有的成员。父类的成员可以在子类中使用。

继承是面向对象程序设计的重要特征，也是实现“代码复用”的重要手段。如果一个新类继承自一个设计好的类，就直接具备了已有类的特征，就大大降低了工作难度。已有的类，称为“父类或者基类”，新的类，称为“子类或者派生类”。现实世界中的继承无处不在。如图 6-30 所示：

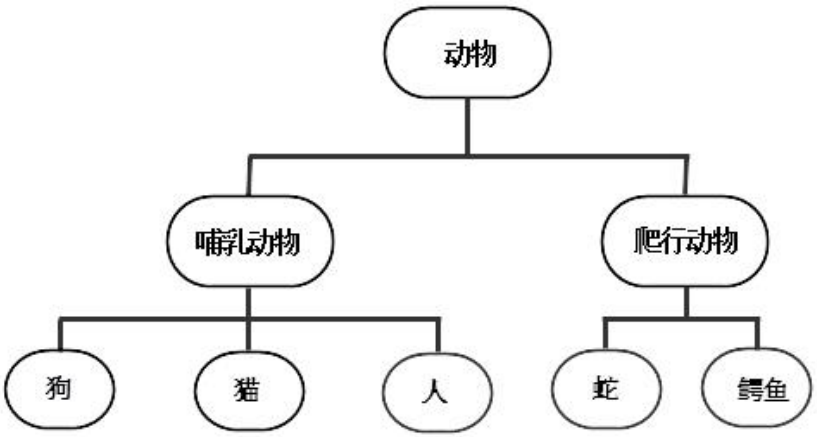


图 6-30 现实世界中的继承

上图中，哺乳动物继承了动物。意味着，动物的特性，哺乳动物都有；在编程中，如果新定义一个 Student 类，发现已经有 Person 类包含了需要的属性和方法，那么 Student 类只需要继承 Person 类即可拥有 Person 类的属性和方法。继承的语法格式如下：

```
class  子类类名(父类)：
    类体
```

如果在类定义中没有指定父类，则默认父类是 object 类。也就是说，object 是所有类的父类。

【示例 6-23】继承实现

```
class Person:
    def sing(self):
        print('人在唱歌')
class Student(Person):
    def dance(self):
        print('学生在跳舞')
s1 = Student()
s1.sing()                #调用父类的方法
s1.dance()
```

执行结果如图 6-31 所示：

图 6-31 示例 6-23 执行结果

如果在子类中需要父类的构造方法就需要显示的调用父类的构造方法，调用格式如下：

```
父类名.__init__(self, 参数列表)
```

【示例 6-24】子类中显示调用父类的构造方法

```
class Person:
    def __init__(self,name,age):
        self.name = name
        self.__age = age
    def say_age(self):
        print(self.name,"的年龄是：",self.__age)
class Student(Person):
    def __init__(self,name,age,score):
        self.score = score
        #子类并不会自动调用父类的__init__()构造方法，需要显式的调用
        Person.__init__(self,name,age)
s1 = Student("张三",15,85)
s1.say_age()
```

执行结果如图 6-32 所示：

图 6-32 示例 6-24 执行结果

6.7.2 方法的重写

子类继承了父类除构造方法之外的所有成员，如果父类中方法的功能不能满足子类的需求，子类可以重写父类的方法。示例如下：

【示例 6-25】方法的重写

```
class Person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def say_age(self):
        print("年龄：",self.age)
    def say_name(self):
        print("姓名：",self.name)
class Student(Person):
    def __init__(self,name,age,score):
        self.score = score
        Person.__init__(self,name,age)   #构造函数中包含调用父类构造函数
    def say_score(self):
        print("分数是：",self.score)
    #重写父类的方法
    def say_name(self):
        print("报告老师，我是",self.name)
s1 = Student("张三",15,85)
s1.say_score()
s1.say_name()
s1.say_age()
```

执行结果如图 6-33 所示：

图 6-33 示例 6-25 执行结果

6.7.3 检测继承关系

在很多场景中，需要知道一个类 A 是否是从另外一个类 B 继承，这种校验主要是为了调用 B 类中的成员(方法和属性)。如果 B 是 A 的父类，那么创建 A 类的实例肯定会拥有 B 类所有的成员，关键是要判断 B 是否是 A 的父类可以使用 issubclass 函数。语法格式如下：

```
issubclass(sub,sup)
```

其中，第 1 个参数是子类、第 2 个参数是父类。如果第 1 个参数指定的类与第 2 个参数指定的类确实是继承关系，那么该函数返回 True，否则返回 False。

【示例 6-26】issubclass()函数的使用

```
class Person(object):
    pass
class Child(Person):                    # Child 继承 Person
    pass
class GrandSon(Child):
    pass
print(issubclass(Child,Person))         # True
print(issubclass(GrandSon,Person))      # True
```

执行结果如图 6-34 所示：

图 6-34 示例 6-26 执行结果

6.7.4 多继承

Python 支持多重继承，一个子类可以有多个“直接

父类”。这样，就具备了“多个父类”的特点。要想为某一个类指定多个父类，需要在类名后面的圆括号中设置。多个父类名之间用逗号（,）分隔。语法格式如下：

```
class  子类类名(父类 1,父类 2,父类 3...)：
    类体
```

【示例 6-27】多继承的使用

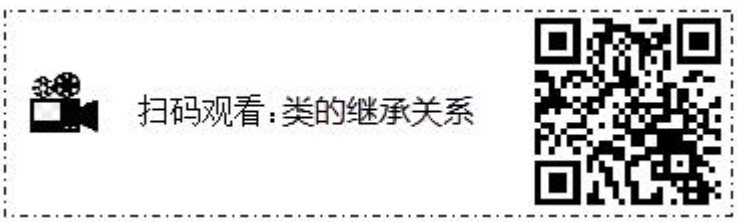
```
#多重继承
class A:
    def aa(self):
        print("aa")
class B:
    def bb(self):
        print("bb")
class C(B,A):
    def cc(self):
        print("cc")
c = C()
c.cc()
c.bb()
c.aa()
```

执行结果如图 6-35 所示：

图 6-35 示例 6-27 执行结果

6.7.5 查看类的继承层次结构

通过类的方法 mro()或者类的属性__mro__可以输出这个类的继承层次结构。



【示例 6-28】类的继承层次结构

```
class A:
    pass
class B(A):
    pass
class C(B):
    pass
print(C.mro())
```

执行结果如图 6-36 所示：

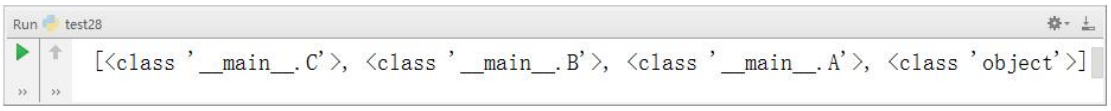


图 6-36 示例 6-28 执行结果

6.7.6 MRO()

MRO（Method Resolution Order）指方法解析顺序。Python 支持多继承，如果父类中有相同名字的方法，在子类没有指定父类名时，解释器将“从左向右”按顺序搜索。可以通过 mro()方法获得“类的层次结构”，方法解析顺序也是按照这个“类的层次结构”寻找的。

【示例 6-29】MRO()方法的使用

```
#多重继承
class A:
    def aa(self):
        print("aa")
    def say(self):
        print("say AAA!")
class B:
    def bb(self):
        print("bb")
    def say(self):
        print("say BBB!")
class C(B,A):
    def cc(self):
        print("cc")
c = C()
```

```
print(C.mro())          #打印类的层次结构
#解释器寻找方法是“从左到右”的方式寻找，此时会执行 B 类中的 say()
c.say()
```

执行结果如图 6-46 所示：

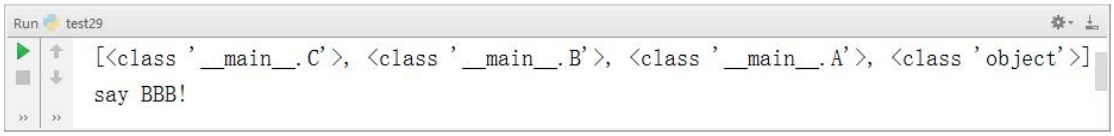


图 6-37 示例 6-29 执行结果

6.7.7 super()

在子类中，如果想要获得父类的方法时，可以通过super()。

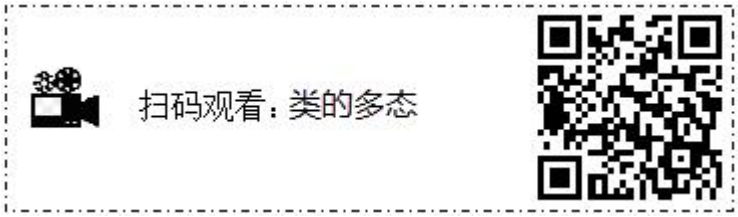
【示例 6-30】super()的使用

```
#super()
class A:
    def say(self):
        print("say AAA")
class B(A):
    def say(self):
        A.say(self)       #调用父类的 say 方法
        super().say()     #通过 super()调用父类的方法
        print("say BBB")
b = B()
b.say()
```

执行结果如图 6-38 所示：

图 6-38 示例 6-30 执行结果

6.8 类的多态



多态指的是同一个方法调用，由于对象不同可能会有不同的行为。现实生活中，同一个方法，具体实现会完全不同。 比如：同样是调用人的“休息”方法，张三是睡觉，李四是旅游，高淇老师是敲代码，数学教授是做数学题； 同样是调用人“吃饭”的方法，中国人用筷子吃饭，英国人用刀叉吃饭，印度人用手吃饭。

多态的要点：

1. 多态是方法的多态，不是属性的多态（多态与属性无关）。

2. 多态的存的必要条件：继承，方法重写。

【示例 6-31】多态的使用

```
#多态
class Animal:
    def shout(self):
        print("动物叫了一声")
class Dog(Animal):
    def shout(self):
        print("小狗，汪汪汪")
class Cat(Animal):
    def shout(self):
        print("小猫，喵喵喵")
def animalShout(a):
    if isinstance(a,Animal):
        a.shout()       #传入的对象不同，shout 方法对应的实际行为也不同。
animalShout(Dog())
animalShout(Cat())
```

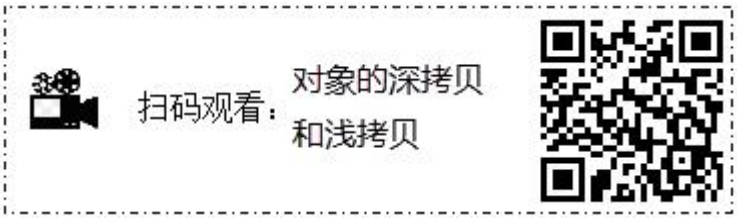
执行结果如图 6-39 所示：

图 6-39 示例 6-31 执行结果

上面的代码展示了多态最为多见的一种用法，即父类引用做方法的形参，实参可以是任

意的子类对象，可以通过不同的子类对象实现不同的行为方式。

6.9 对象的深拷贝和浅拷贝



Python 拷贝一般都是浅拷贝。拷贝时，对象包含的子对象内容不拷贝。因此，源对象和拷贝对象会引用同一个子对象。如果想实现深拷贝可以使用 copy 模块的 deepcopy 函数，递归拷贝对象中包含的子对象。源对象和拷贝对象所有的子对象也不同。

【示例 6-32】对象的深拷贝和浅拷贝

```
#测试对象的引用赋值、浅拷贝、深拷贝
import copy
class MobilePhone:
    def __init__(self,cpu,screen):
        self.cpu = cpu
        self.screen = screen
class CPU:
    def calculate(self):
        print("CPU 对象:",self)
class Screen:
    def show(self):
        print("屏幕对象：",self)
c = CPU()
s = Screen()
m = MobilePhone(c,s)
print('------浅拷贝-------')
m2 = copy.copy(m)     #m2 是新拷贝的另一个手机对象
m.cpu.calculate()
m2.cpu.calculate()     #m2 和 m 拥有了一样的 cpu 对象和 screen 对象
m.screen.show()
m2.screen.show()
print('------深拷贝-------')
m3 = copy.deepcopy(m)
m.cpu.calculate()
```

```
m3.cpu.calculate()          #m3 和 m 拥有不一样的 cpu 对象和 screen 对象
m.screen.show()
m3.screen.show()
```

执行结果如图 6-40 所示：

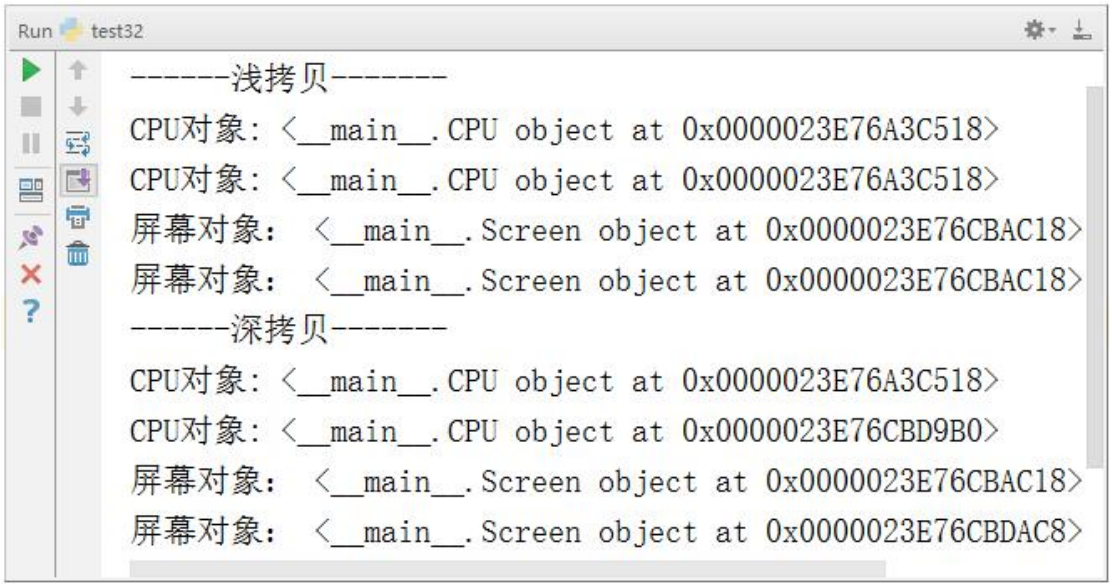


图 6-40 示例 6-32 执行结果

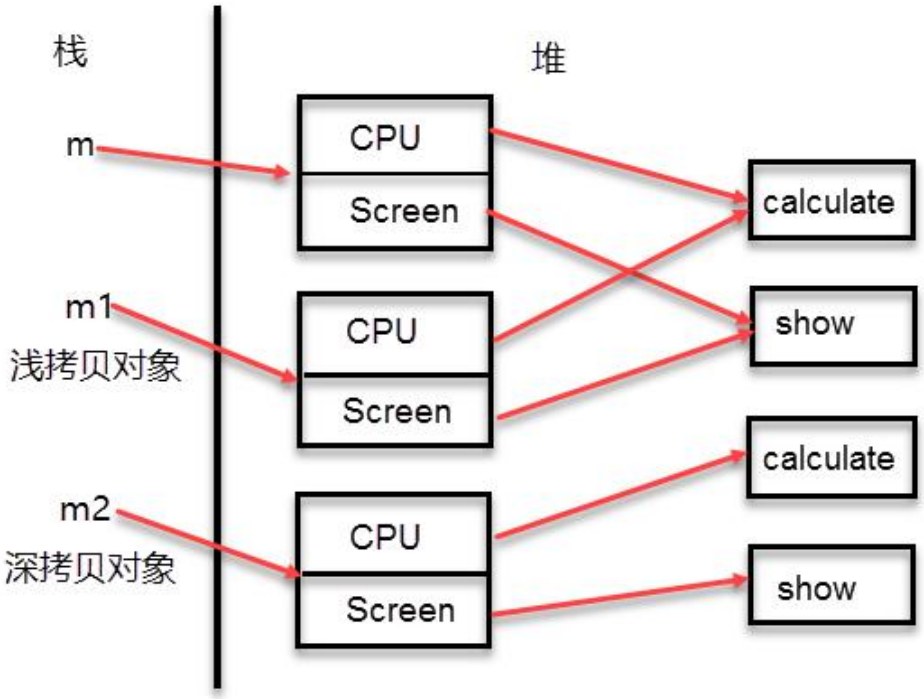
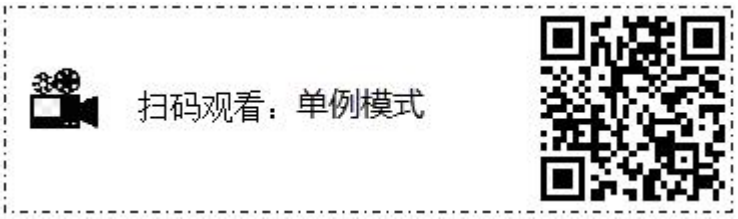


图 6-41 示例 6-32 内存分析图

6.10 设计模式

6.10.1 单例模式



单例模式（Singleton Pattern）是一种常用的软件设计模式，该模式的主要目的是确保某一个类只有一个实例存在。并且提供一个访问该实例的全局访问点。

单例模式只生成一个实例对象，减少了对系统资源的开销。当一个对象的产生需要比较多的资源，如读取配置文件、产生其他依赖对象时，可以产生一个“单例对象”，然后永久驻留内存中，从而极大的降低开销。

__new__()方法在__init__()方法之前被调用，用于创建实例对象。利用这个方法和类属性的特性可以实现设计模式中的单例模式，示例如下：

【示例 6-33】单例模式

```
#单例模式
class MySingleton:
    __instance = None
    def __new__(cls, *args, **kwargs):     #重写__new__()方法
        if cls.__instance == None:
            cls.__instance = object.__new__(cls)
        return cls.__instance
    def __init__(self,name):
        self.name = name
a = MySingleton("aa")
print(a)
b = MySingleton("bb")
print(b)
c = MySingleton("cc")
print(c)
```

执行结果如图 6-42 所示：

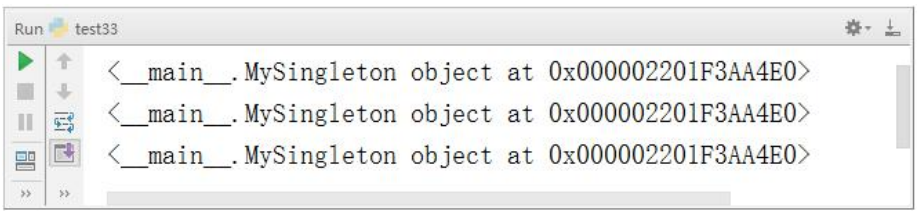


图 6-42 示例 6-33 执行结果

在上面的代码中，定义了私有的类属性__instance，初始值为 None。__instance 的值是第 1 次创建的实例，以后的实例化操作都将共享这个属性。因此，就达到了创建唯一实例的目的。

6.10.2 工厂模式

所谓工厂模式就是通过专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

工厂模式实现了创建者和调用者的分离，使用专门的工厂类将选择实现类、创建对象进行统一的管理和控制。

假设需要创建奥迪车、比亚迪车等各种各样的车，如果没有“工厂”来生产它们，就要在代码中自己进行实例化。代码如下：

```
class Benz:
    pass
class BMW:
    pass
class BYD:
    pass
#创建实例
benz=Benz()
bm=BMW()
byd=BYD()
```

但现实中，可能会面对很多汽车产品，而且每个产品的构造参数还不一样，这样在创建实例时会遇到麻烦。这时就可以构造一个“简单工厂”把所有汽车实例化的过程封装在里面。

【示例 6-34】工厂模式

```
#工厂模式
class CarFactory:
    def createCar(self,brand):
        if brand == "奔驰":
            return Benz()
        elif brand == "宝马":
            return BMW()
        elif brand == '比亚迪':
            return BYD()
```

```
            else:
                return "未知品牌，无法创建"
class Benz:
    def run(self):
        print('奔驰在马路上...')
class BMW:
    def run(self):
        print('宝马在马路上...')
class BYD:
    def run(self):
        print('比亚迪在马路上...')
factory = CarFactory()
c1 = factory.createCar("奔驰")
c2 = factory.createCar("宝马")
c1.run()
c2.run()
```

执行结果如图 6-43 所示：

图 6-43 示例 6-34 执行结果

在上面的代码中，有了 CarFactory 类后，就可以通过向固定的接口传入参数获得想要的对象实例，从而创建了奥迪车、比亚迪车、宝马车。

习题

一、选择题

1. 以下哪个不属于面向对象的特征( )

A. 封装 B. 继承

C. 多态 D. 复合

2. 实现以下哪个方法可以让对象像函数一样被调用 ( )

A. str() B. iter()

C. call() D. next()

3. 构造方法的作用是（）。

A. 一般成员方法

B. 类的初始化

C. 对象的初始化

D. 对象的建立

4. Python 类中包含一个特殊的变量（），它表示当前对象自身，可以访问类的成员。

A. self B. me C.this D.与类同名

5. Python 中用于释放类占用资源的方法是（）。

A. __init__ B. __del__ C. _del D. delete

二、简答题

1. 面向对象三大特性，各有什么用处，说说你的理解。面向过程和面向对象的区别。
2. 类和对象的关系。
3. 简述实例属性与类属性的区别。
4. 简述实例方法与类方法，静态方法的区别。
5. 多重继承的执行顺序，请解答以下输出结果是什么？并解释。

三、编码题

1. 定义发动机类 Motor、底盘类 Chassis、座椅类 Seat，车辆外壳类 Shell，并使用组合关系定义汽车类。具体需求如下：

（1）定义汽车的 run()方法，里面需要调用 Motor 类的 work()方法，

（2）run()方法也需要调用座椅类 Seat 的 work()方法，

（3）run()方法也需要调用底盘类 Chassis 的 work()方法。

2. 使用工厂模式、单例模式实现如下需求：

(1) 电脑工厂类 ComputerFactory 用于生产电脑 Computer。工厂类使用单例模式，也就是说只能有一个工厂对象。

(2) 工厂类中可以生产各种品牌的电脑：联想、华硕、神舟

(3) 各种品牌的电脑使用继承实现：

(4) 父类是 Computer 类，定义了 calculate 方法

(5) 各品牌电脑类需要重写父类的 calculate 方法

3. 定义一个 Employee 雇员类，要求如下：

(1) 属性有：id、name、salary

(2) 运算符重载+：实现两个对象相加时，默认返回他们的薪水和

(3) 构造方法要求：输入 name、salary，不输入 id。id 采用自增的方式，从 1000 开始自增，第一个新增对象是 1001，第二个新增对象是 1002。

(4) 根据 salary 属性，使用@property 设置属性的 get 和 set 方法。set 方法要求输入：1000-50000 范围的数字。

第七章 模块

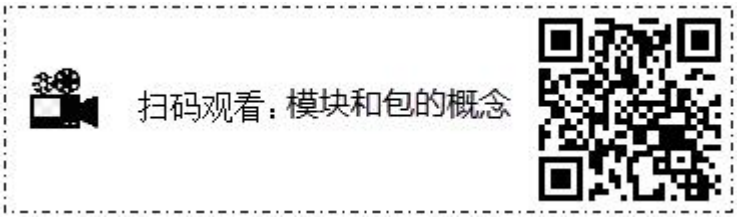
在软件开发过程中，随着程序规模越来越大，将程序代码全部放在一个文件里代码就会越来越长，不容易维护。为了编写可维护的代码，需要把很多函数分组与归类，分别放到不同的文件里。每个文件包含的代码就相对较少，当今主流的程序语言基本都是采用这样的代码组织方式。

通过阅读本章，你可以：

- 了解什么是模块
- 了解模块有什么好处
- 掌握模块的使用
- 掌握自定义模块的定义及使用
- 掌握包的使用
- 掌握自定义模块跨项目使用
- 掌握常用第三方库的安装及使用

7.1 模块化(module)程序设计理念

7.1.1 模块和包概念的进化史



“量变引起质变”是哲学中一个重要的理论。量变为什么会引起质变呢？本质上理解，随着数量的增加，管理方式会发生本质的变化；旧的管理方式完全不适合，必须采用新的管理方式。

程序越来越复杂，语句多了，怎么管理？很自然的，我们会将实现同一个功能的语句封装到函数中，统一管理和调用，于是函数诞生了。

程序更加复杂，函数和变量多了，怎么管理？同样的思路，“物以类聚”，我们将同一类型对象的“数据和行为”，也就是“变量和函数”，放到一起统一管理和调用，于是“类和对象”诞生了。

程序继续复杂，函数和类更加多了，怎么办？好，我们将实现类似功能的函数和类统统放到一个模块中，于是“模块”诞生了。

程序还要复杂，模块多了，怎么办？ 于是，我们将实现类似功能的模块放到一起，于是“包”就诞生了。

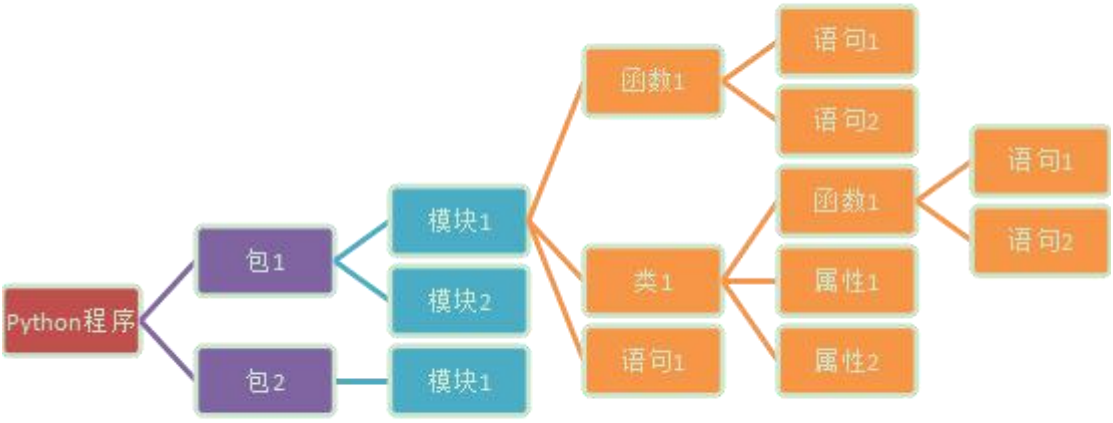


图 7-1　程序结构图

总结：

- Python 程序由模块组成。一个模块对应 python 源文件，一般后缀名是：.py。
- 模块由语句组成。运行 Python 程序时，按照模块中语句的顺序依次执行。
- 语句是 Python 程序的构造单元，用于创建对象、变量赋值、调用函数、控制语句等。

7.1.2 为什么需要模块化编程

模块(module)对应于 Python 源代码文件(.py 文件)。模块中可以定义变量、函数、类、普通语句。 这样，可以将一个 Python 程序分解成多个模块，便于后期的重复应用。

模块化编程（Modular　Programming）将一个任务分解成多个模块。每个模块就像一个积木一样，便于后期的反复使用、反复搭建。

图 7-2　搭积木

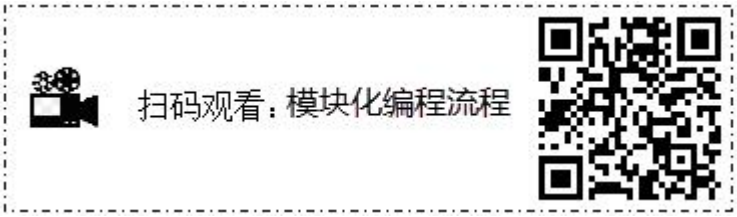
模块化编程有如下几个重要优势：

- 合理控制模块规模：按照逻辑性将代码尽可能的划分为多个子模块，方便客户端程序的调用。便于将一个任务分解成多个模块，实现团队协同开发，完成大规模程序。
- 提高代码可维护性：从代码的视角分析，变量越多，代码维护性越大，甚至不可能完成维护任务。尽可能将变量作用范围缩小，控制在一个局部范围之内。无论如何，这样都会提高模块的可维护性成本。
- 代码可重用性：模块设计的优势之一便是代码的可重用性。充分的利用系统标准库或者自己编写的第三方库来为程序服务。已经实现的功能无需再次编写，只需要在模块中引入需要的其他功能模块即可。

7.1.3 模块化编程流程

模块化编程的一般流程：

1) 设计 API，进行功能描述。
2) 编码实现 API 中描述的功能。
3) 在模块中编写测试代码，并消除全局代码。
4) 使用私有函数实现不被外部客户端调用的模块函数。



1) 模块的 API 和功能描述要点

API（Application Programming Interface 应用程序编程接口），这种接口适用于标准模块，也适用于自定义编程模块。API 没有规定具体的代码实现，只是提供访问模块的规范。API 的设计应当包括多少信息，尽管没有完全一致的规范，但基本都涵盖为模块调用的客户端提供必要的信息。例如 API 包含的函数名称、参数类型、返回值类型等。每个 Python 标准模块和我们自己编写的模块都属于 API 的一种实现方式。如图 7-3 所示，展示了 API 与模块、客户端调用的常见关系。

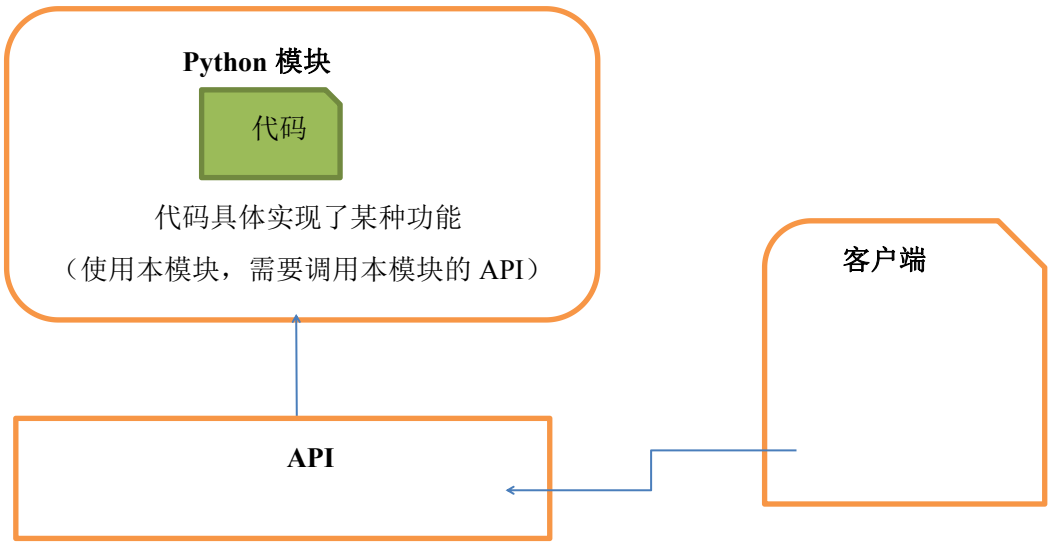


图 7-3 API 模块关系图

模块化编程中，首先设计的就是模块的 API（即要实现的功能描述），然后开始编码实现 API 中描述的功能。最后，在其他模块中导入本模块进行调用。

想查看模块的 API，可以通过如下方式：

（1）help(模块名)。

【示例 7-1】导入 math 模块，并通过 help()查看 math 模块的 API

```
import math
help(math)
```

（2）python 的 api 文档

进入 python 的安装目录下的 docs 子目录，可以看到 python 的 api 文档。双击打开 chm 文档，即可通过索引输入“math”查询到对应的 API 内容，如图 7-4 所示：

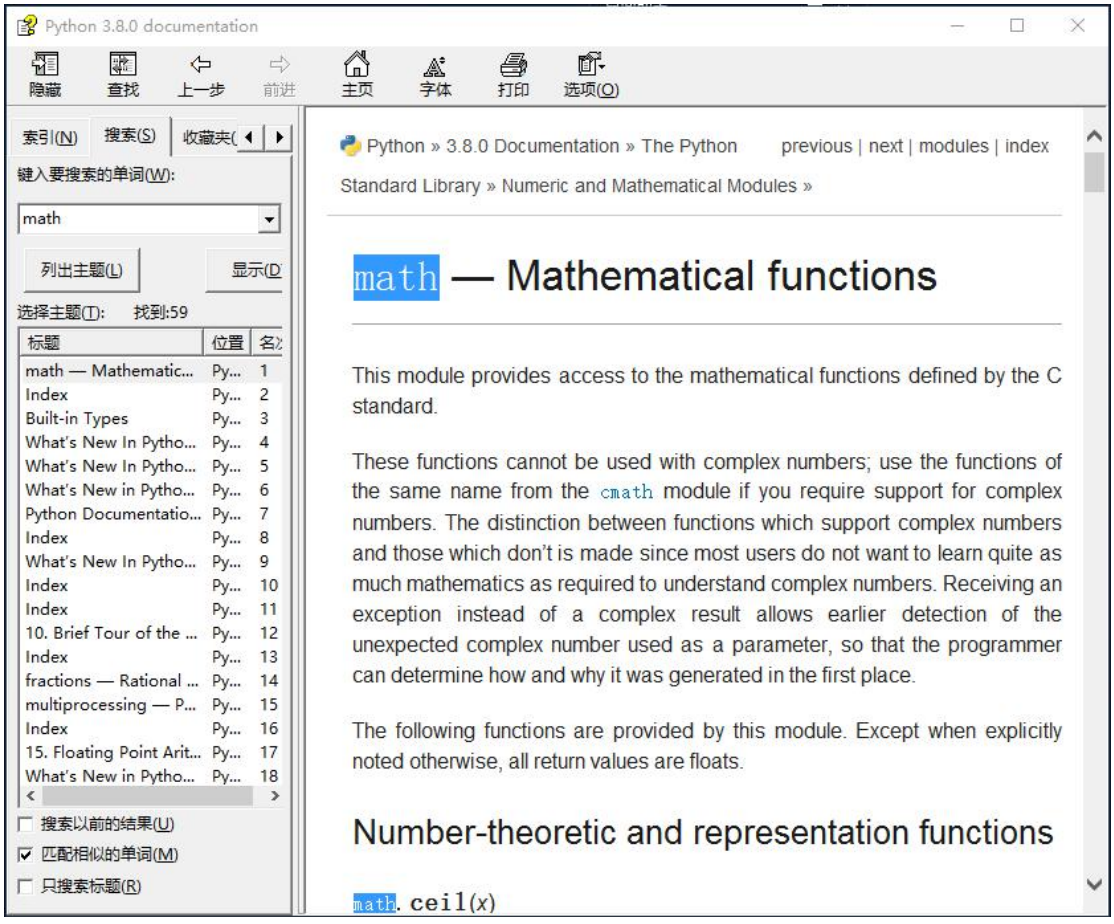


图 7-4　python API 文档

【示例 7-2】设计计算薪水模块的 API

```
"""本模块用于计算公司员工的薪资 """
```

```
company = "北京尚学堂"
def yearSalary(monthSalary):
    """根据传入的月薪，计算出年薪"""
    pass
def daySalary(monthSalary):
    """根据传入的月薪，计算出每天的薪资"""
    pass
```

从上面示例可以看到，此模块只有功能描述和规范，需要编码人员按照要求实现编码。

【示例 7-3】通过__doc__可以获得模块的文档字符串的内容

```
import salary
print(salary.__doc__)
print(salary.yearSalary.__doc__)
```

执行结果如图 7-5 所示：

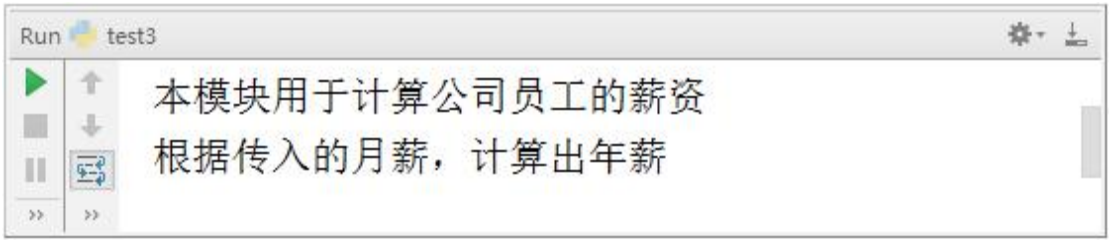


图 7-5 示例 7-3 执行结果

2) 模块的创建和测试代码

每个模块都有一个名称，通过特殊变量__name__可以获取模块的名称。在正常情况下，模块名字对应源文件名。仅有一个例外，就是当一个模块被作为程序入口时（主程序、交互式提示符下），它的__name__的值为“__main__”。可以根据这个特点，将模块源代码文件中的测试代码进行独立的处理。

【示例 7-4】通过__name__==“__main__”独立处理模块的测试代码

```
"""
本模块用于计算公司员工的薪资
"""
company = "北京尚学堂"
def yearSalary(monthSalary):
    """根据传入的月薪，计算出年薪"""
    return monthSalary*12
def daySalary(monthSalary):
```

```
    """根据传入的月薪，计算出每天的薪资"""
    return monthSalary/22.5          #国家规定每个月的平均工作日是 22.5
if __name__ =="__main__":            #测试代码
    print(yearSalary(3000))
    print(daySalary(3000))
```

执行结果如图 7-6 所示：

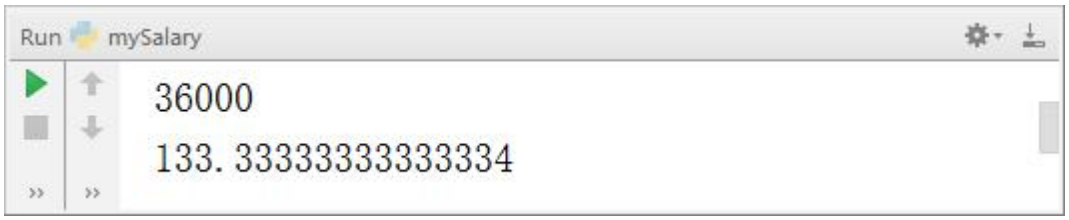


图 7-6 示例 7-4 执行结果

3) 模块文档字符串和 API 设计

可以在模块的第一行增加一个文档字符串，用于描述模块的相关功能。然后，通过__doc__可以获得文档字符串的内容。

【示例 7-5】导入模块读取文档字符串

```
import mySalary
print(mySalary.__doc__)
print(mySalary.yearSalary.__doc__)
```

执行结果如图 7-7 所示：

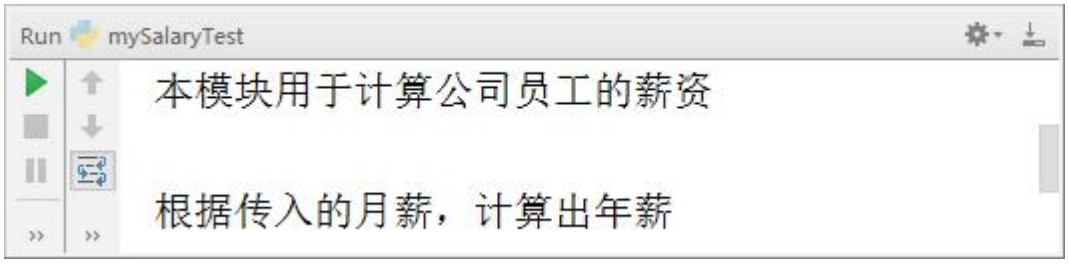
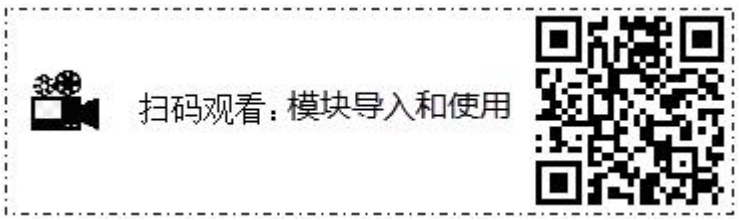


图 7-7 示例 7-5 执行结果

7.2 模块的导入和使用

Python 模块(Module)是一个 Python 文件，以 .py 作为后缀结尾的文件，通常包含了 Python 对象定义和 Python 语句。模块内的 Python 代码通常是按照一定逻辑组织起来的，把相关的代码分配到一个模块里能让你的代



码更好用，更易懂，更有可维护性。模块内可以定义函数、类、变量。当然，模块里也能包含可执行的代码。

导入模块的方式主要有四种，分别是：

- import 模块名
- from 模块名 import 函数名
- from 模块名 import *

7.2.1 import 语句

如果希望在一个 Python 源码模块中调用另外一个 Python 源码模块，只需要在这个 Python 源码中引入 import 语句，语法格式如下：

```
import module_name1[, module_name 2[,... module_name X]
```

引用模块中的函数，语法格式如下：

```
module_name.fuanction_name
```

Python 本身就内置了很多非常有用的模块，只要安装完毕，这些模块就可以立刻使用。以内建的 math 模块和 math 模块中的 sqrt 函数为例。

【示例 7-6】import 语句的使用

```
#导入 math 模块及调用 math 模块中的 sqrt 函数
import math
a=100
print(math.sqrt(a))
```

执行结果如图 7-8 所示：

图 7-8 示例 7-6 执行结果

7.2.2 from...import 语句

如果在 Python 程序中大量使用模块中的某些函数，那么每次在调用函数时候都要加上“模块名”显得有些麻烦，所以在这种情况下，可以使用 from…import..语句导入模块中函数。该语句的语法结构如下：

```
from  module_name  import  function_name
```

使用 from …import 语句导入 math 模块，调用 math 模块中的 sqrt 函数，示例如下：

【示例 7-7】from...import 语句的使用

```
#导入 math 模块中的 sqrt 函数
from math import sqrt
print(sqrt(100))
```

执行结果如图 7-9 所示：

图 7-9 示例 7-7 执行结果

7.2.3 from...import *语句

使用上面导入模块的方式只能使用指定的函数，如果想调用导入模块中的所有函数，可以将 function_name 替换成星号（*），语法格式如下：

```
from  module_name  import  *
```

【示例 7-8】from...import *语句的使用

```
from math import *
print(sqrt(100))
print(abs(-123))
print(ceil(10.2))
print(floor(10.2))
```

执行结果如图 7-10 所示：

图 7-10 示例 7-8 执行结果

注意：

- import 和 from…import…语句可以写在 Python 代码中的任何位置，但一定要在引用相应模块函数之前执行 import 或 from…import…语句，否则调用函数时会抛出异常。
- import 语句导入的是模块，from...import 导入的是模块中的一个函数或一个类。

7.2.4 __import__()动态导入

import 语句本质上就调用内置函数__import__()，可以通过__import__()实现动态导入。给__import__()传入不同的参数，就能导入不同的模块。

【示例 7-9】__import__()动态导入

```
s = 'math'
m = __import__(s)  #导入后生成的模块对象的引用给变量 m
print(m.pi)
```

执行结果如图 7-11 所示：

图 7-11 示例 7-9 执行结果

注意：

- 一般不建议使用__import__()导入，其行为在 python2 和 python3 中有差异，会导致意外错误，如果需要动态导入可以使用 importlib 模块。

【示例 7-10】importlib 模块的使用

```
import importlib
a = importlib.import_module('math')
print(a.pi)
```

执行结果如图 7-12 所示：

图 7-12 示例 7-10 执行结果

7.2.5 模块加载问题

当导入一个模块的时候，模块中的代码就会被执行。如果再次导入这个模块，则模块中的代码不会被再次执行。Python 之所以这样设计，因为导入模块更多的是需要定义模块中变量、函数、对象等。这些并不需要反复定义和执行。“只导入一次 import-only-once”就成了优化。

【示例 7-11】模块加载测试

（1）创建一个模块 test9

```
print('test_module 模块被加载。。。')
```

（2）创建测试模块 test_test9

```
import test9
import test9
```

执行结果如图 7-13 所示：

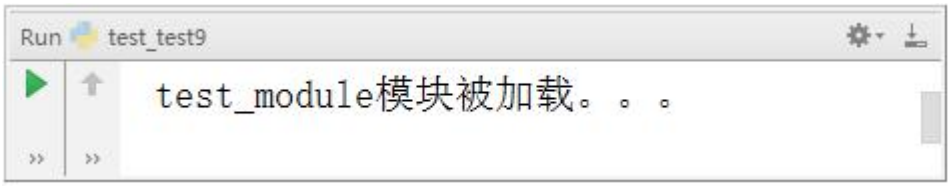


图 7-13 示例 7-11 执行结果

从上诉示例可以看到，导入两次模块 test9，但只加载了一次。因此一个模块无论导入多少次，这个模块在整个解释器进程内有且仅有一个实例对象。

如果有时候需要重新加载一个模块，这时候可以使用：importlib.reload()方法。

【示例 7-12】importlib.reload()重新加载模块

```
import test9
import test9
print('*'*20)
import importlib
importlib.reload(test9)
```

执行结果如图 7-14 所示：

图 7-14 示例 7-12 执行结果

7.3 自定义模块

7.3.1 创建模块

在 Python 中，每个 Python 文件都可以作为一个模块，模块的名字就是文件的名字。在实际开发中，为了满足某些功能的需求，需要自己建立一些模块来实现项目需求，接下来看

看自定义模块。

【示例 7-13】创建一个名为 myModule.py 的文件，并定义 hello 方法

```
#定义 hello 方法
def    hello():
       print('hello module')
```

在同目录下的另一个.py 文件中输入：

```
#导入自定义的模块
from mymodule import hello
#调用模块中的方法
hello()
```

执行结果如图 7-15 所示：

图 7-15 示例 7-13 执行结果

7.3.2 查看模块

如果要查看一个模块中的所有成员，可以使用 dir()函数，示例如下：

【示例 7-14】查看模块

```
#导入自定义模块
import myModule
#查看模块
print(dir(myModule))
```

执行结果如图 7-16 所示：

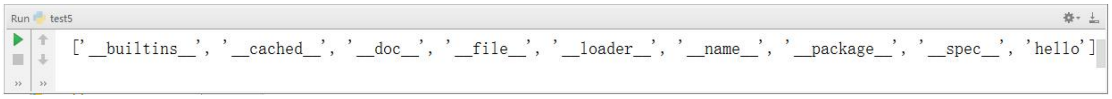


图 7-16 示例 7-14 执行结果

7.3.3 内置属性 __name__

模块有一些内置属性，用于完成特定的任务，如__name__、__doc__。每个模块都有一个名称，例如，__name__用于判断当前模块是否是程序的入口，如果当前程序正在被使用，__name__的值为“__main__”。通常给每个模块都添加一个条件语句，用于单独测试该模

块的功能。

【示例 7-15】__name__属性的使用创建一个模块 myModule_name

```
def add(a,b):
    return a+b
if __name__=='__main__':
    print('myModule 主程序正在运行')
else:
    print('myModule_name 正在被另一个程序调用')
```

执行 myModule_name 模块结果如图 7-17 所示：

图 7-17 示例 7-15 执行结果

创建另一个模块 myModule_name_test，代码如下:

```
from myModule_name import add
print(add(4,5))
```

执行 myModule_name_test 模块结果如图 7-18 所示：

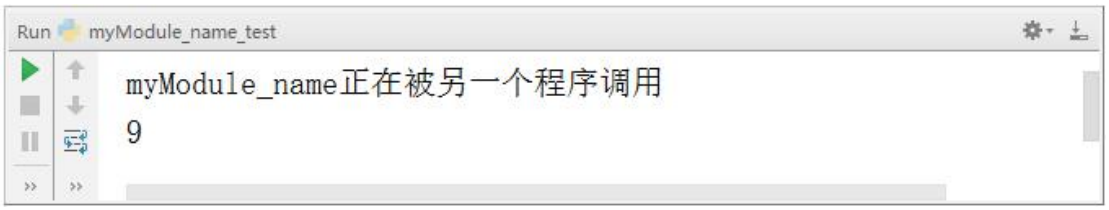
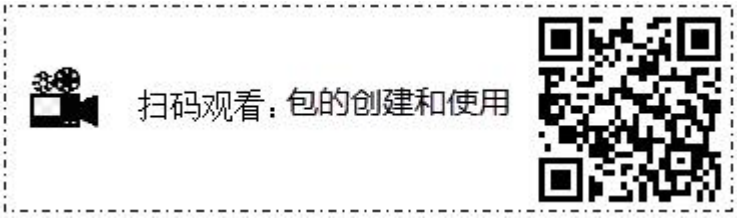


图 7-18 示例 7-15 执行结果

__name__ == '__main__'是一种常见的判断模块是被导入还是直接运行的方法。很多情况下，模块是需要被导入使用的。直接运行模块是为了进行测试。因此，使用这种方法进行判定即可保证即使在正式发布后测试代码也不需要删除。

7.4 包

7.4.1 包的概念和结构



当一个项目中有很多个模块时，需要再进行组织。将功能类似的模块放到一起，形成了“包”。本质上“包”就是一个必须有__init__.py 的文件夹。典型结构如图 7-19 所示：

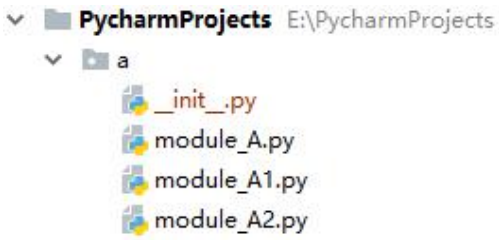


图 7-19 包的典型结构图

包下面可以包含“模块(module)”，也可以再包含“子包(subpackage)”。就像文件夹下面可以有文件，也可以有子文件夹一样。

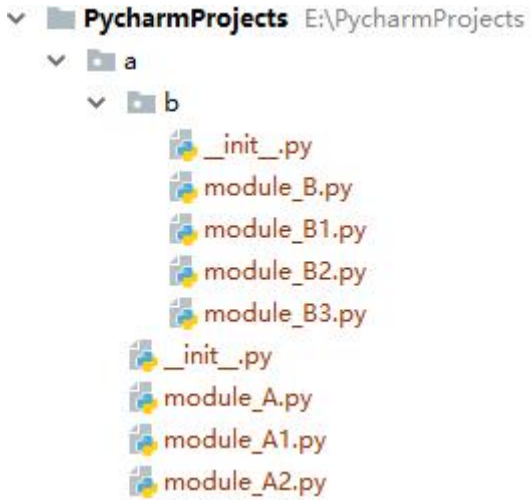


图 7-20 包中包含子包结构图

从图 7-20 中可以看到 a 是上层的包，下面有一个子包：b。可以看到每个包里面都有__init__.py 文件。

7.4.2 包的使用

Python 包和 Java 包的作用是相同的，都是为了实现程序的重用，把实现一个常用功能的代码组合到一个包中，调用包提供的服务从而实现重用。包的使用与模块非常类似。可以使用 import 导入也可以使用 from...import 来导入，比如访问图 7-20 中的 module_B 模块中的函数 fun_B()，访问方式如下：

（1）使用 import 导入

```
import  a.b.module_B
```

此种方式的导入，在访问时候，必须加完整包名称来访问，如：a.b.module_B.fun_B()。

（2）使用 from...import 导入

```
from a.b import module_B
```

此种方式的导入，访问时不需要加那么冗长的前缀，直接可以使用模块名。 如：module_B.fun_B()。

（3）使用 from...import 直接导入函数

```
from a.b.module_B  import  fun_B()
```

此种方式的导入，访问时候直接使用函数名。如：fun_B()。

注意：

- 当使用“from package import item”这种形式的时候，对应的 item 既可以是包里面的子模块（子包）也可以是包里面定义的其他名称，比如函数、类或变量。
- import item1,item2 这种语法，item 必须是包或者模块，不能是其他。

【示例 7-16】定义一个包 pack，在 pack 包下定义 myMath 模块

```
def add(a,b):
    return a+b
def sub(a,b):
    return a-b
def mul(a,b):
    return a*b
def div(a,b):
    return a/b
```

创建 packTest 模块，在模块中导入 pack 包下的 myMath 模块，示例代码如下：

```
#调用模块中的方法
if __name__=="__main__":
    a=10
    b=20
    # 1.使用 import 导入
    import pack.myMath
    print('import 导入，访问 mymath 模块中的方法：')
    print('加法运算:',pack.myMath.add(a,b))
    print('减法运算:',pack.myMath.sub(a,b))
    #2.from ...import 导入
    from pack import myMath
    print('from ...import 导入，访问 mymath 模块中的方法：')
    print('乘法运算:',myMath.mul(a,b))
    print('除法运算:',myMath.div(a,b))
    #3.from ...import 导入  直接导入函数
    from pack.myMath import add,sub,mul,div
    print('from ...import 导入  直接导入函数')
    print('加法运算:',add(a,b))
```

```
    print('减法运算:',sub(a,b))
    print('乘法运算:',mul(a,b))
    print('除法运算:',div(a,b))
```

执行结果如图 7-21 所示：

图 7-21 示例 7-16 执行结果

7.4.3 包__init__的使用

一个包是一个带有特殊文件 __init__.py 的目录。__init__.py 文件定义了包的属性和方法。其实它可以什么也不定义；可以只是一个空文件，但是必须存在。如果 __init__.py 不存在，这个目录就仅仅是一个目录，而不是一个包。

当你将一个包作为模块导入，实际上导入了它的__init__.py 文件。且在调用包中模块时候，首先会执行__init__.py 文件。示例如下：

【示例 7-17】定义一个包 pack2，在 pack2 包下的__init__.py 中添加如下代码：

```
print('执行了包 pack2 中的_init_.py 文件')
```

创建 pack2Test 模块，在模块中导入 pack2 包，代码如下：

```
import pack2
```

执行 pack2Test 模块结果如图 7-22 所示：

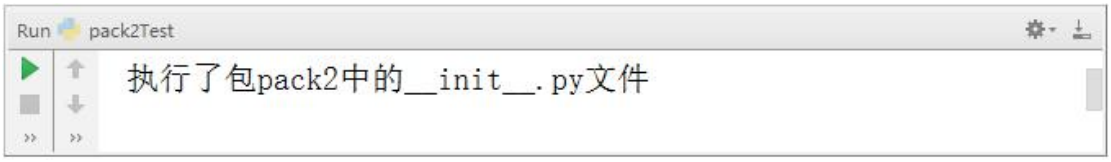


图 7-22 示例 7-17 执行结果

从上面的代码可以看出，包中的__init__.py 具有初始化包的作用，控制包的导入行为，可以在 __init__.py 里把包 pack3 中模块预先导入。示例如下：

【示例 7-18】定义一个包 pack3，在 pack3 包下定义 myMath 模块

```
def add(a,b):
    return a+b
def sub(a,b):
    return a-b
def mul(a,b):
    return a*b
def div(a,b):
    return a/b
```

在 pack3 包下__init__.py 中添加代码如下：

```
#导入 myMath 模块
print('pack3 中的__init__.py 文件执行')
#导入当前模块 myMath 中的所有方法(.代表当前)
from .myMath import *
```

创建 pack3Test 模块，代码如下：

```
#导入包下所有模块
from pack3 import *
#调用模块中的方法
print('加法运算：',add(20,10))
print('减法运算：',sub(20,10))
print('乘法运算：',mul(20,10))
print('除法运算：',div(20,10))
```

执行 pack3Test 模块结果如图 7-23 所示：

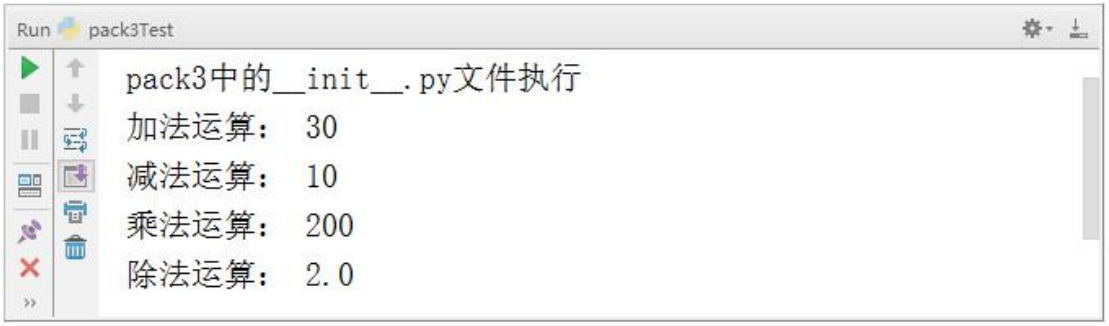
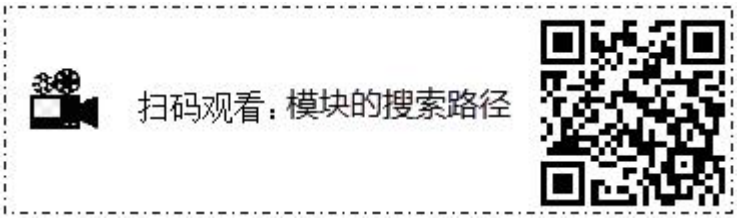


图 7-23 示例 7-18 执行结果

7.4.4 sys.path 和模块搜索路径



当导入某个模块文件时候，Python 解释器去哪里找这个文件？只有找到这个文件才能读取、装载运行该模块文件。它一般按照如下路径搜索模块文件。

（1）内置模块。

（2）当前目录。

（3）程序的主目录。

（4）pythonpath 目录（如果已经设置了 pythonpath 环境变量）。

（5）标准链接库目录。

（6）第三方库目录（site-packages 目录）。

（7）.pth 文件的内容（如果存在）。

（8）sys.path.append()临时添加的目录。

当任何一个 python 程序启动的时候，就将上面的搜索路径（除内置模块以外的路径）进行收集。放到 sys 模块的 path 属性中。

【示例 7-19】使用 sys.path 查看和临时修改搜索路径

```
import sys
sys.path.append('d:/')
print(sys.path)
```

执行结果如图 7-24 所示：

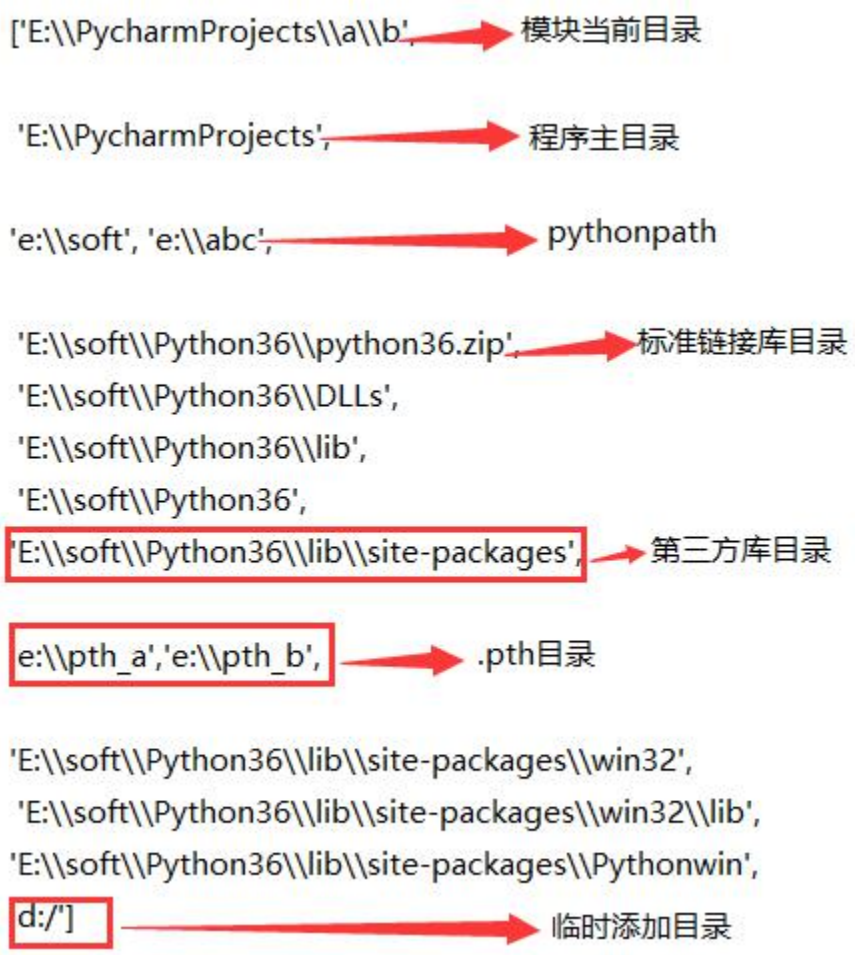


图 7-24 示例 7-19 执行结果

(1) pythonpath 环境变量的设置

pythonpath 环境变量设置步骤如下：

1）打开 Windows 的桌面，右击“计算机”，在弹出的快捷菜单中选择“属性”菜单项，会显示如图 7-25 所示的“系统”窗口。

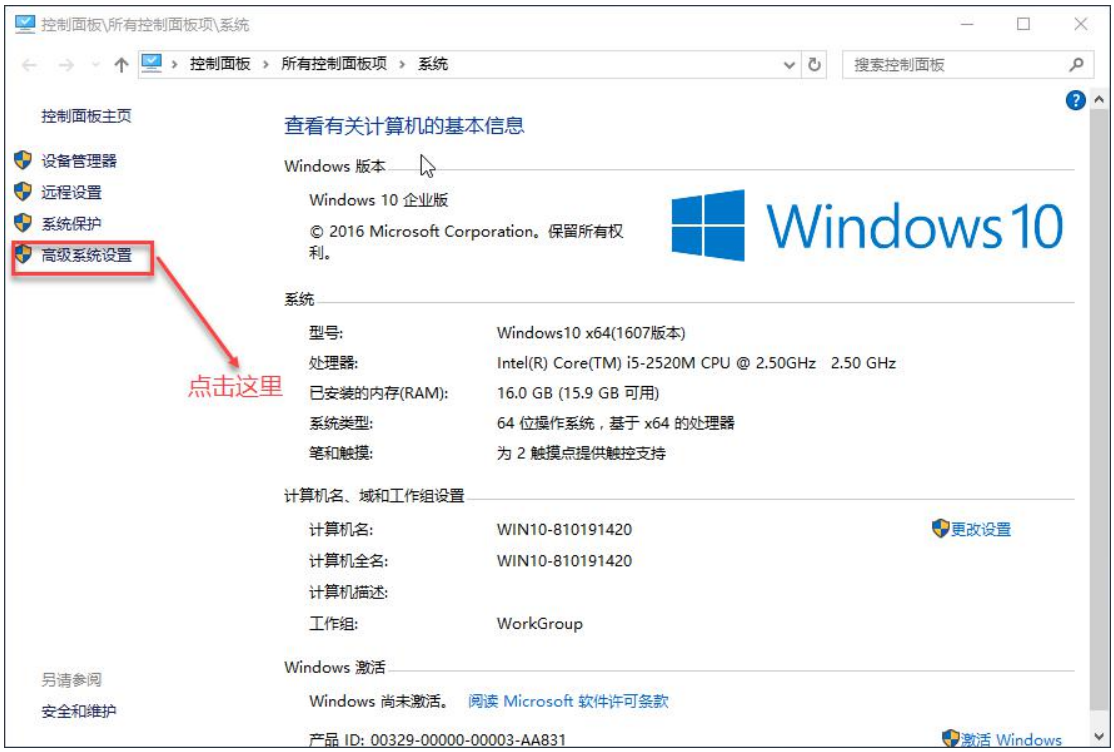


图 7-25 “系统”窗口

2)点击“系统”窗口左侧的“高级系统设置”，会弹出如图 7-26 所示的“系统属性的对话框。

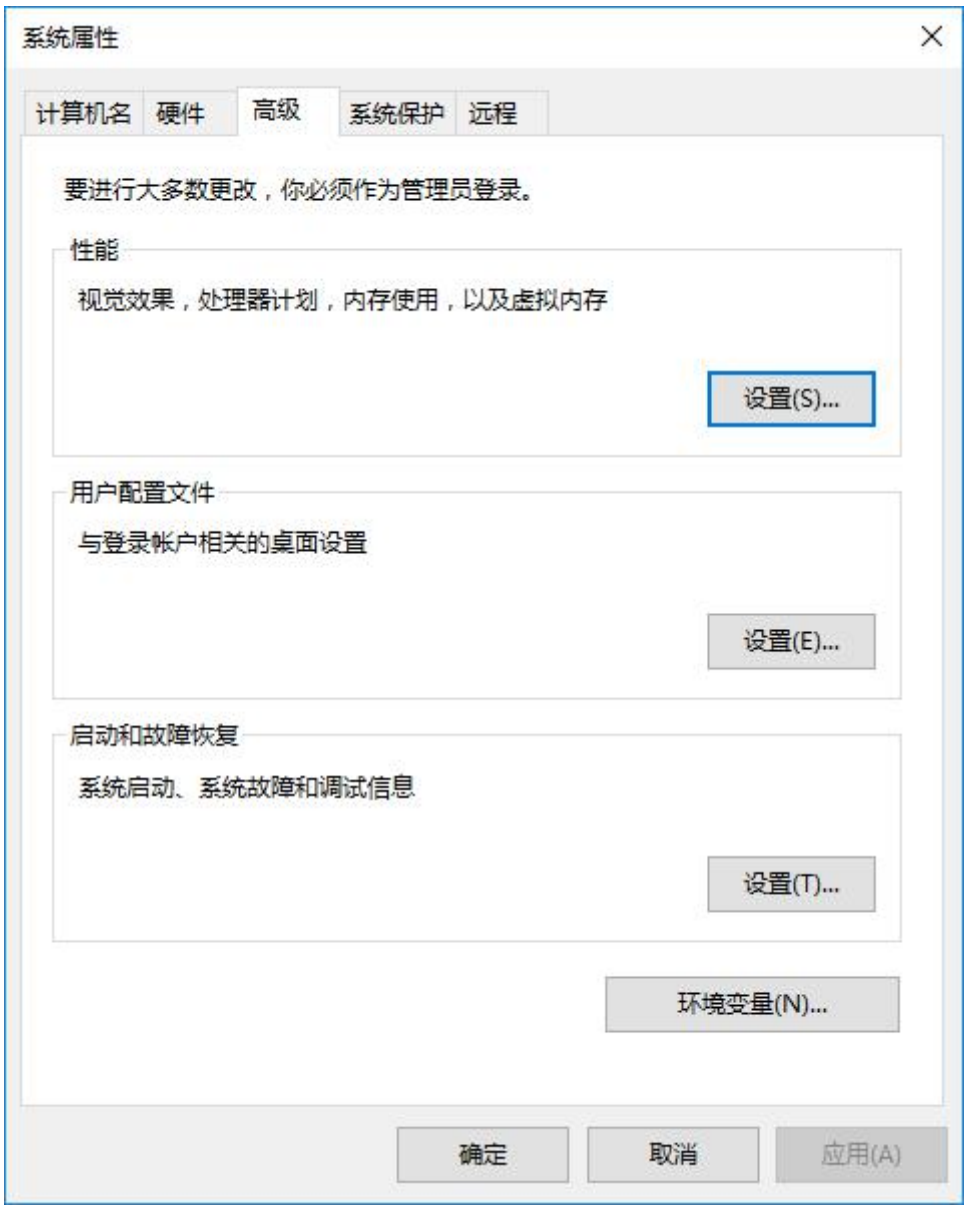


图 7-26 “系统属性”窗口

3)点击“系统属性”对话框下方的“环境变量(N)…”按钮，会弹出如图 7-27 所示的“环境变量”对话框。

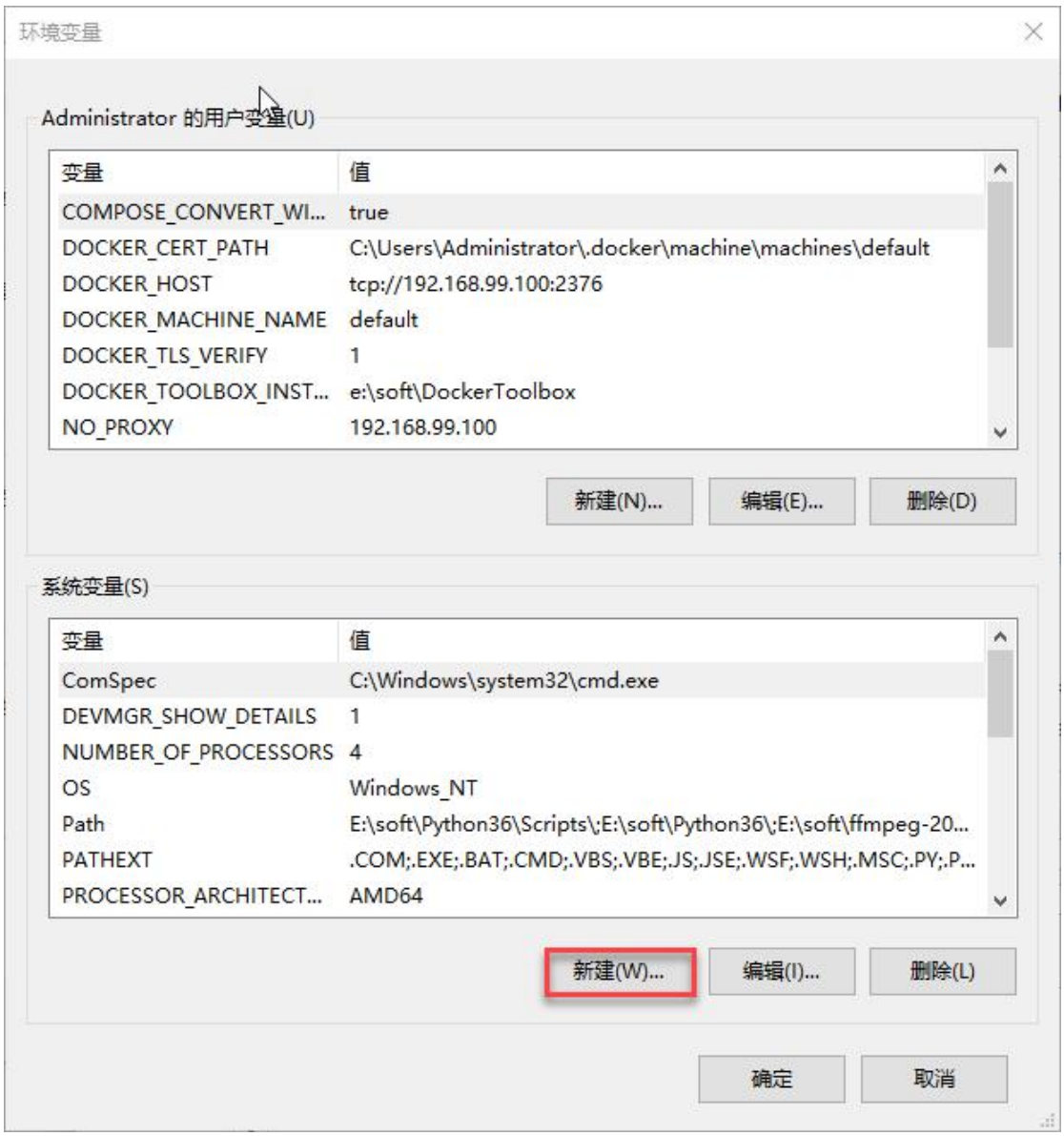


图 7-27 “环境变量”对话框

4)在“环境变量”对话框中有两个列表框，单击“新建(W)…”按钮就会弹出如图 7-28 所示的“新建系统变量”的对话框。填写变量名(N)和变量值(V)，点击“确定”按钮完成配置。

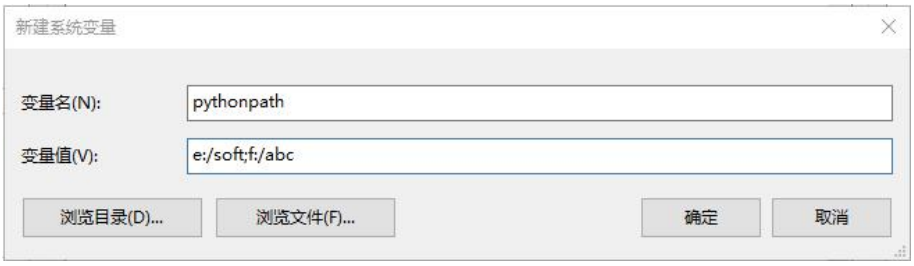


图 7-28 “编辑用户变量”对话框

(2) .pth 文件写法

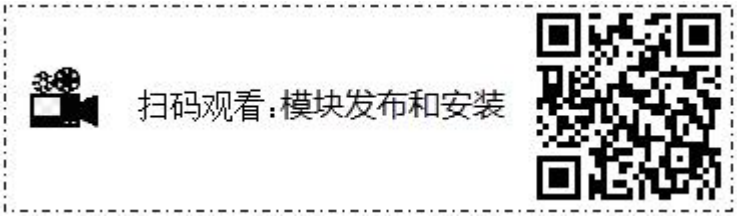
在 site-packages 目录下添加.pth 文件，并在文件中添加内容：

```
e:\pth_a
e:\pth_b
```

注意：

- 需要确保 e:\pth_a 和 e:\pth_b 对应目录真实存在。
- 在 windows 系统中建立.pth 文件，由于没有文件名不能直接创建，需要输入 .pth. 才能正常建立.pth 文件。

7.5 模块的发布和安装



前面引入了包的概念，将多个模块放在一起打包，这样做不仅避免了模块名冲突的问题，同时也解决了调用繁琐的问题，但是自定义的模块只能在当前项目目录下导入使用，脱离了该项目就会报错，为了解决这个问题，我们要学习模块的发布和安装，即完成某个模块的开发后，可以将他对外开放，其他开发者也可以以“第三方扩展库”的方式使用我们的模块。

7.5.1 模块的发布

模块发布的步骤如下所述：

（1） 为模块文件创建如图 7-29 目录结构的文件夹。

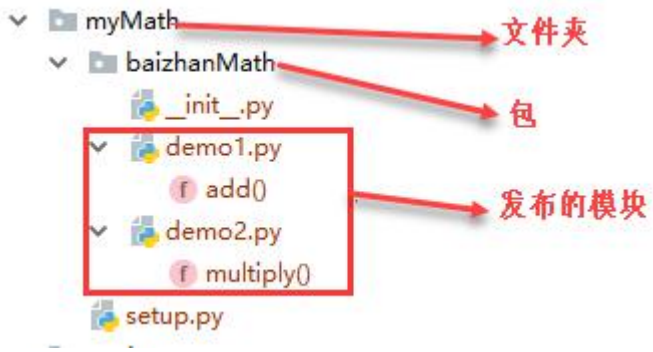


图 7-29 发布模块目录结构图

（2） 配置 setup.py 文件内容如下：

```
from distutils.core import setup
setup(name='baizhanMath',
version='1.0',
description='测试自定义模块的发布与安装',
author='gaoqi', # 作者
author_email='gaoqi110@163.com',
```

```
py_modules=['baizhanMath.demo1','baizhanMath.demo2'])
```

（3） 打开 dos 命令窗口，进入此文件夹目录，执行如下命令进行发布，如图 7-30 所示：

```
python setup.py sdist
```

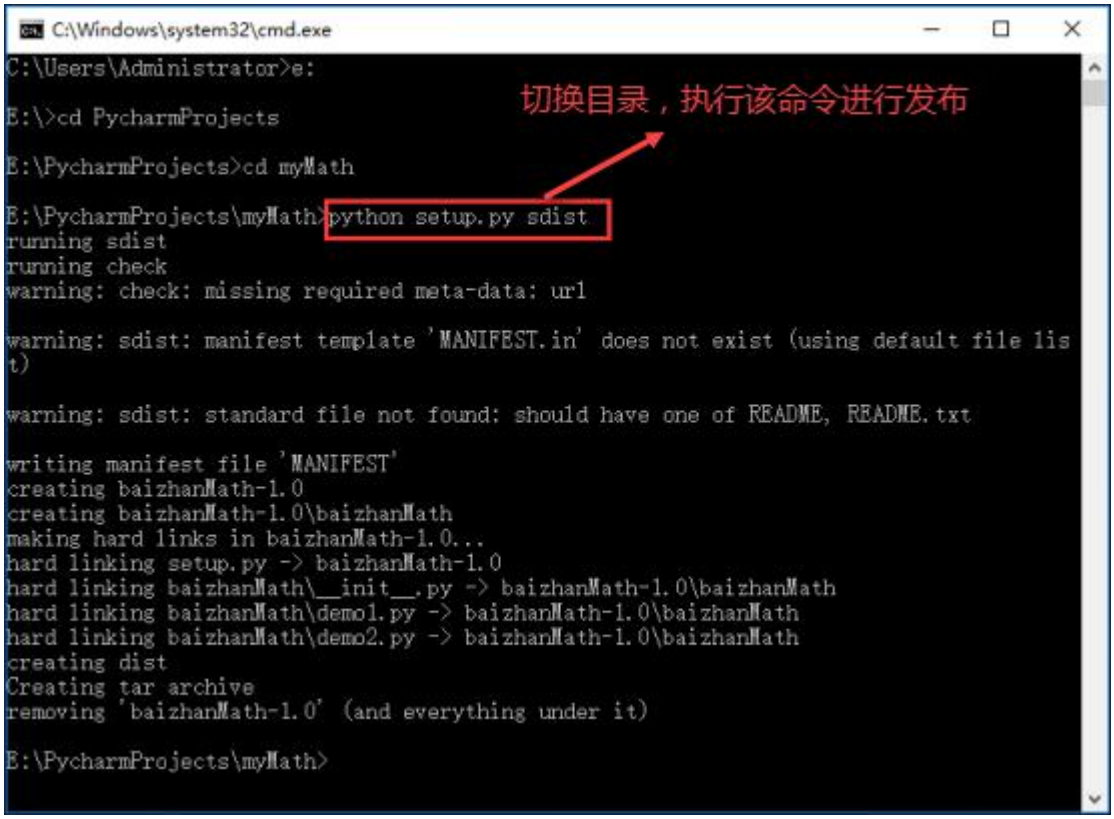


图 7-30 发布模块

（4） 此时文件夹的目录结构变为如图 7-31 所示：

图 7-31 发布后目录结构图

（5） 以上步骤完成了发布，其中在 dist 目录中 baizhanMath-1.0.tar.gz 这个压缩包就是要安装的的自定义模块压缩包。

7.5.2 模块的安装和使用

为了将自己封装的模块能够像系统模块一样，在任何地方都可以被调用，这时候就需要将 baizhanMath-1.0.tar.gz 这个安装包安装到 python 解释器运行的环境中，模块的安装步骤如下所述：

（1） 打开 dos 命令窗口，进入 setup.py 文件所在目录，执行如下命令进行安装。如图 7-32 所示：

```
python setup.py install
```

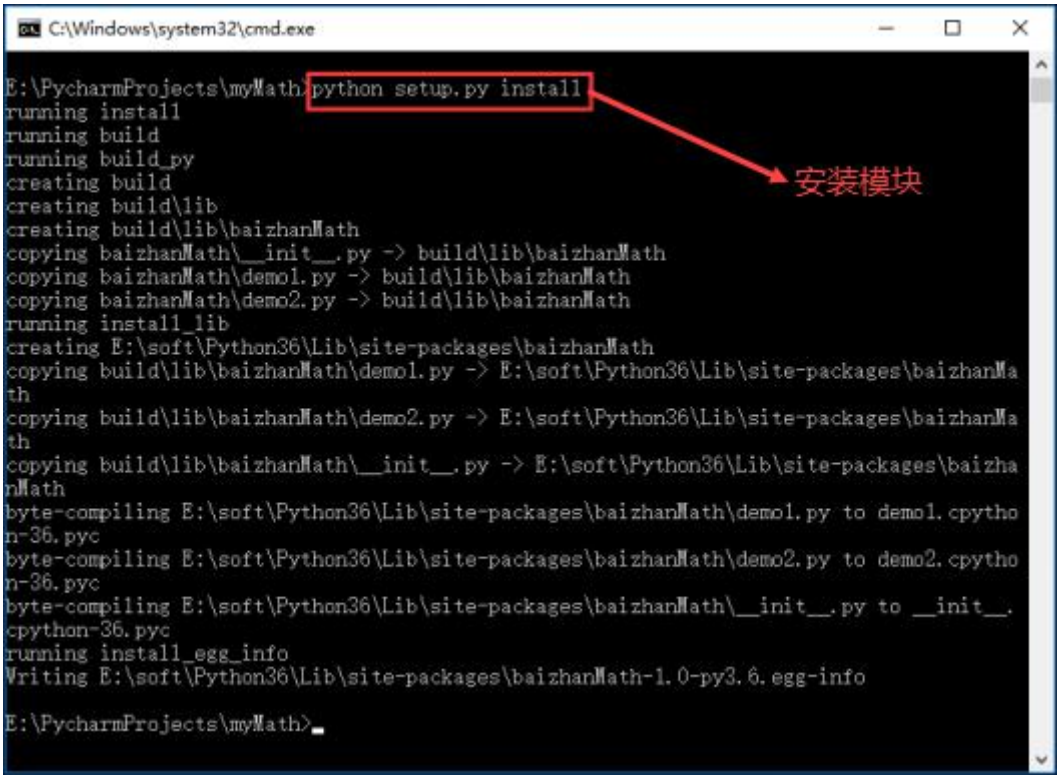


图 7-32 模块的安装

安装成功后，进入 python 环境安装目录 Lib/site-packages 目录（第三方库都安装在这里，python 解释器执行时候会搜索该路径），可以找到刚刚安装的 baizhanMath 模块，如图 7-33 所示：

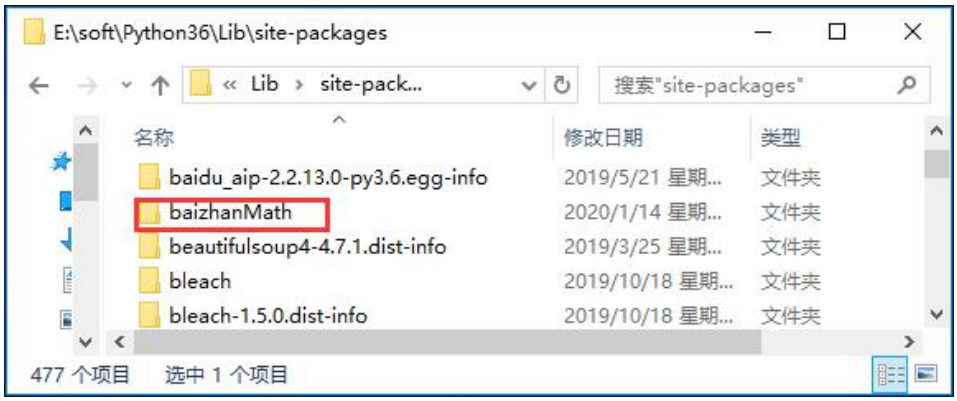


图 7-33 安装成功

【示例 7-20】创建 baizhanMathTest.py

```
from baizhanMath import demo1,demo2
demo1.add()
demo2.multiply()
```

执行结果如 7-34 所示：

图 7-34 示例 7-18 执行结果

7.5.3 上传模块到 PyPI 网站

将自己开发好的模块上传到 PyPI 网站上，成为公共的资源。可以让全球用户自由使用。上传步骤如下所诉：

（1）注册 PyPI 网站

进入 https://pypi.org/网站如图 7-35 所示，点击 Register 按钮，进入注册页面如图 7-36 所示：

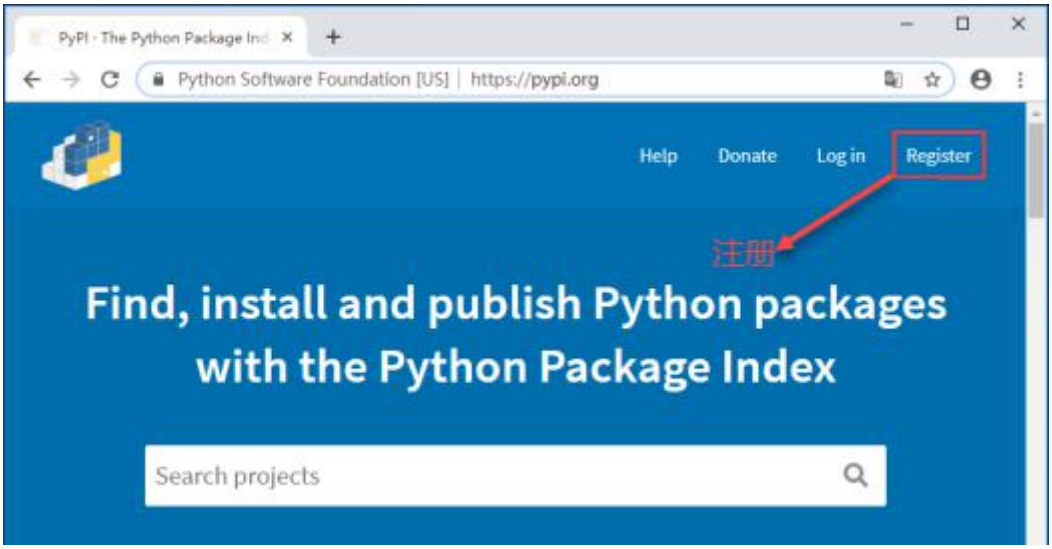


图 7-35 PyPI 网站

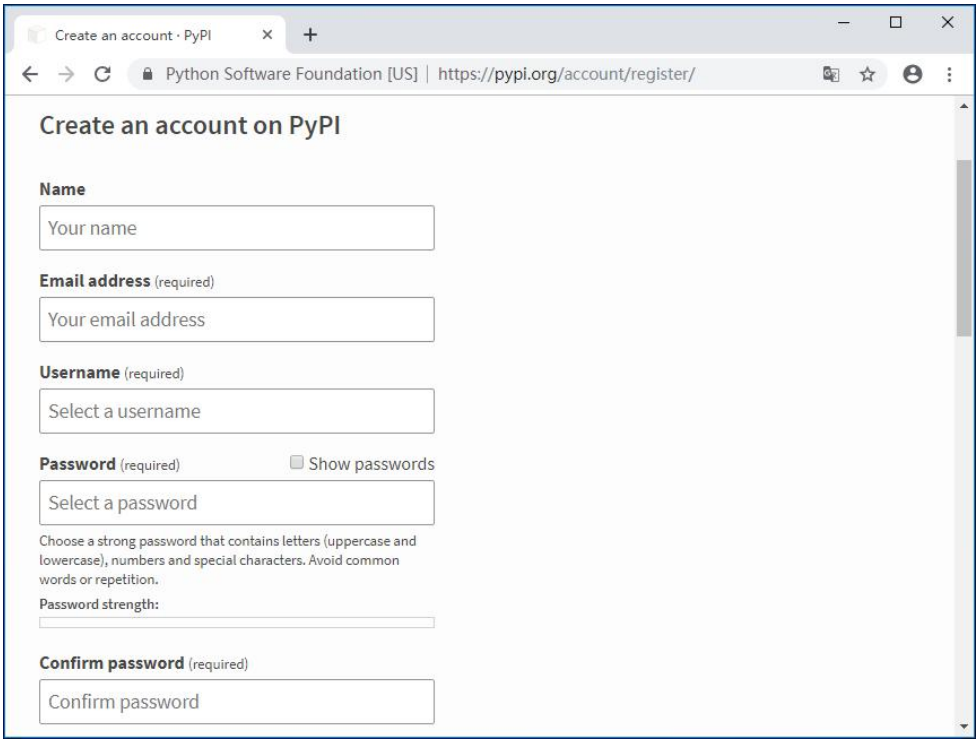


图 7-36 注册页面

注意：

- 注册完成后，会发生一封邮件到您的邮箱，必须点击验证后才可以下面的操作

（2）在用户家目录（C:\Users\Administrator），创建用户信息文件.pypirc，内容如下：

```
[distutils]
index-servers=pypi
[pypi]
repository =https://upload.pypi.org/legacy/
```

```
username =账户名
password =密码
```

（3）上传并远程发布

打开 dos 命令窗口，进入 setup.py 文件所在目录，执行如下命令进行上传并发布。

```
python setup.py sdist upload
```

（4）登录 PyPI 网站，如果您的模块上传成功，那么登录后右侧导航栏可以看到管理入口。

注意：

- 在 Windows 下创建不包含文件名的文件会失败，因此创建用户信息文件时候应写成“.pypirc.”，前后两个点都要。
- 上传 PyPI 网站的模块名要唯一，否则上传失败，名称已由其他用户注册。
- 上传成功后，登录 PyPI 网站并不是马上就可以看到上传模块，需要时间。

7.6 库(Library)

Python 中库是借用其他编程语言的概念，没有特别具体的定义。模块和包侧重于代码组织，有明确的定义。

一般情况，库强调的是功能性，而不是代码组织。通常将某个功能的“模块的集合”，称为库。

7.6.1 标准库(Standard Library)

Python 拥有一个强大的标准库。Python 语言的核心只包含数字、字符串、列表、字典、文件等常见类型和函数，而由 Python 标准库提供了系统管理、网络通信、文本处理、数据库接口、图形系统、XML 处理等额外的功能。

Python 标准库的主要功能有：

（1）文本处理，包含文本格式化、正则表达式匹配、文本差异计算与合并、Unicode 支持，二进制数据处理等功能。

（2）文件处理，包含文件操作、创建临时文件、文件压缩与归档、操作配置文件等功能。

（3）操作系统功能，包含线程与进程支持、IO 复用、日期与时间处理、调用系统函数、

日志（logging）等功能。

（4）网络通信，包含网络套接字，SSL 加密通信、异步网络通信等功能。

（5）网络协议，支持 HTTP，FTP，SMTP，POP，IMAP，NNTP，XMLRPC 等多种网络协议，并提供了编写网络服务器的框架。

（6）W3C 格式支持，包含 HTML，SGML，XML 的处理。

（7）其它功能，包括国际化支持、数学运算、HASH、Tkinter 等。

目前学过的有：random、math、time、file、os、sys 等模块。可以通过 random 模块实现随机数处理、math 模块实现数学相关的运算、time 模块实现时间的处理、file 模块实现对文件的操作、OS 模块实现和操作系统的交互、sys 模块实现和解释器的交互。

7.6.2 第三方扩展库

强大的标准库奠定了 python 发展的基石，丰富和不断扩展的第三方库是 python 壮大的保证。可以进入 PyPI 官网 https://pypi.org 如图 7-37 所示：

图 7-37 PyPI 官网

可以看到发布的第三方库达到了十多万种，众多的开发者为 Python 贡献了自己的力量。

表 7-1 第三方扩展库

分类	库名称	说明
环境管理	P	非常简单的交互式 python 版本管理工具
	Pyenv	简单的 Python 版本管理工具
	Vex	可以在虚拟环境中执行命令
	Virtualenv virtualenvwrapper	创建独立 Python 环境的工具
包管理	pip	Python 包和依赖关系管理工具
	pip-tools	保证 Python 包依赖关系更新的一组工具
	Pipenv	Python 官方推荐的新一代包管理工具
	Poetry	可完全取代 setup.py 的包管理工具
包仓库	warehouse	下一代 PyPI
	Devpi	PyPI 服务和打包/测试/分发工具
分发 (打包为可执行文件以	PyInstaller	将 Python 程序转成独立的执行文件（跨平台）
	Nuitka	将脚本、模块、包编译成可执行文件或扩展模块

分类	库名称	说明
便分发)	py2app	将 Python 脚本变为独立软件包（Mac OS X）
	py2exe	将 Python 脚本变为独立软件包（Windows）
	pynsist	一个用来创建 Windows 安装程序的工具，可以在安装程序中打包 Python 本身
构建工具 (将源码编译成软件)	Buildout	构建系统，从多个组件来创建，组装和部署应用
	BitBake	针对嵌入式 Linux 的类似 make 的构建工具
	Fabricate	对任何语言自动找到依赖关系的构建工具
交互式 Python 解析器	IPython	功能丰富的工具，非常有效的使用交互式 Python
	bpython	界面丰富的 Python 解析器
	Ptpython	高级交互式 Python 解析器，构建于 python-prompt-toolkit 之上
文件管理	Aiofiles	基于 asyncio，提供文件异步操作
	Imghdr	(Python 标准库）检测图片类型
	Mimetypes	(Python 标准库）将文件名映射为 MIME 类型
	path.py	对 os.path 进行封装的模块
	Pathlib	(Python3.4+ 标准库）跨平台的、面向对象的路径操作库
	Unipath	用面向对象的方式操作文件和目录
	Watchdog	管理文件系统事件的 API 和 shell 工具
日期和时间	Arrow	更好的 Python 日期时间操作类库
	Chronyk	解析手写格式的时间和日期
	Dateutil	Python datetime 模块的扩展
	PyTime	一个简单易用的 Python 模块，用于通过字符串来操作日期/时间
	when.py	提供用户友好的函数来帮助用户进行常用的日期和时间操作
文本处理	chardet	字符编码检测器，兼容 Python2 和 Python3
	Difflib	(Python 标准库)帮助我们进行差异化比较
	Fuzzywuzzy	模糊字符串匹配
	Levenshtein	快速计算编辑距离以及字符串的相似度
	Pypinyin	汉字拼音转换工具 Python 版
	Shortuuid	一个生成器库，用以生成简洁的，明白的，URL 安全的 UUID
	simplejson	Python 的 JSON 编码、解码器
	Unidecode	Unicode 文本的 ASCII 转换形式
	Xpinyin	一个用于把汉字转换为拼音的库
	Pygment	通用语法高亮工具
	Phonenumbers	解析，格式化，储存，验证电话号码
	Sqlparse	一个无验证的 SQL 解析器
特殊文本格式处理	Tablib	一个用来处理中表格数据的模块
	Pyexcel	用来读写，操作 Excel 文件的库
	python-docx	读取，查询以及修改 word 文件
	PDFMiner	一个用于从 PDF 文档中抽取信息的工具
	Python-Markdown	纯 Python 实现的 Markdown 解析器

分类	库名称	说明
	Csvkit	用于转换和操作 CSV 的工具
自然语言处理	NLTK	一个先进的平台，用以构建处理人类语言数据的 Python 程序
	Jieba	中文分词工具
	langid.py	独立的语言识别系统
	SnowNLP	一个用来处理中文文本的库
	Thulac	清华大学自然语言处理与社会人文计算实验室研制推出的一套中文词法分析工具包
下载器	you-get	一个 YouTube/Youku/Niconico 视频下载器
图像处理	pillow	最常用的图像处理库
	imgSeek	一个使用视觉相似性搜索一组图片集合的项目
	face_recognition	简单易用的 python 人脸识别
	python-qrcode	一个纯 Python 实现的二维码生成器
OCR	Pyocr	Tesseract 和 Cuneiform 的一个封装(wrapper)
	pytesseract	Google Tesseract OCR 的另一个封装(wrapper)
音频处理	Audiolazy	Python 的数字信号处理包
	Dejavu	音频指纹提取和识别
	id3reader	一个用来读取 MP3 元数据的 Python 模块
	TimeSide	开源 web 音频处理框架
	Tinytag	一个用来读取 MP3, OGG, FLAC 以及 Wave 文件音乐元数据的库
	Mingus	一个高级音乐理论和曲谱包，支持 MIDI 文件和回放功能
视频和 GIF 处理	Moviepy	一个用来进行基于脚本的视频编辑模块，适用于多种格式，包括动图 GIFs
	scikit-video	SciPy 视频处理常用程序
地理位置	GeoDjango	世界级地理图形 web 框架
	GeoIP	MaxMind GeoIP Legacy 数据库的 Python API
	Geopy	Python 地址编码工具箱
HTTP	requests	人性化的 HTTP 请求库
	httplib2	全面的 HTTP 客户端库
	urllib3	一个具有线程安全连接池，支持文件 post，清晰友好的 HTTP 库
Python 实现的数据库	pickleDB	一个简单，轻量级键值储存数据库
	PipelineDB	流式 SQL 数据库
	TinyDB	一个微型的，面向文档型数据库
web 框架	Django	Python 界最流行的 web 框架
	Flask	一个 Python 微型框架
	Tornado	一个 web 框架和异步网络库
CMS 内容管理系统	odoo-cms	一个开源的，企业级 CMS，基于 odoo
	djedi-cms	一个轻量级但却非常强大的 Django CMS ，考虑到了插件，内联编辑以及性能
	Opps	一个为杂志，报纸网站以及大流量门户网站设计的 CMS 平台，基于 Django
电子商务和支付系统	django-oscar	一个用于 Django 的开源的电子商务框架

分类	库名称	说明
	django-shop	一个基于 Django 的店铺系统
	Shoop	一个基于 Django 的开源电子商务平台
	Alipay	Python 支付宝 API
	Merchant	一个可以接收来自多种支付平台支付的 Django 应用
游戏开发	Cocos2d	用来开发 2D 游戏
	Panda3D	由迪士尼开发的 3D 游戏引擎，并由卡内基梅陇娱乐技术中心负责维护。使用 C++ 编写，针对 Python 进行了完全的封装
	Pygame	Pygame 是一组 Python 模块，用来编写游戏
	RenPy	一个视觉小说（visual novel）引擎
计算机视觉库	OpenCV	开源计算机视觉库
	Pyocr	Tesseract 和 Cuneiform 的包装库
	SimpleCV	一个用来创建计算机视觉应用的开源框架
机器学习 人工智能	TensorFlow	谷歌开源的最受欢迎的深度学习框架
	keras	以 tensorflow/theano/CNTK 为后端的深度学习封装库，快速上手神经网络
	Hebel	GPU 加速的深度学习库
	Pytorch	一个具有张量和动态神经网络，并有强大 GPU 加速能力的深度学习框架
	scikit-learn	基于 SciPy 构建的机器学习 Python 模块
	NuPIC	智能计算 Numenta 平台
科学计算和数据分析	NumPy	使用 Python 进行科学计算的基础包
	Pandas	提供高性能，易用的数据结构和数据分析工具
	SciPy	用于数学，科学和工程的开源软件构成的生态系统
	PyMC	马尔科夫链蒙特卡洛采样工具
代码分析和调试	code2flow	把你的 Python 和 JavaScript 代码转换为流程图
	Pycallgraph	这个库可以把你的 Python 应用的流程(调用图)进行可视化
	Pylint	一个完全可定制的源码分析器
	autopep8	自动格式化 Python 代码，以使其符合 PEP8 规范
	Wdb	一个奇异的 web 调试器，通过 WebSockets 工作
	Lineprofiler	逐行性能分析
	Memory Profiler	监控 Python 代码的内存使用
图形用户界面	Pyglet	一个 Python 的跨平台窗口及多媒体库
	PyQt	跨平台用户界面框架 Qt 的 Python 绑定，支持 Qt v4 和 Qt v5
	Tkinter	Tkinter 是 Python GUI 的一个事实标准库
	wxPython	wxPython 是 wxWidgets C++ 类库和 Python 语言混合的产物
网络爬虫和 HTML 分析	Scrapy	一个快速高级的屏幕爬取及网页采集框架
	Cola	一个分布式爬虫框架

分类	库名称	说明
	Grab	站点爬取框架
	Pyspider	一个强大的爬虫系统
	html2text	将 HTML 转换为 Markdown 格式文本
	python-goose	HTML 内容/文章提取器
硬件编程	Ino	操作 Arduino 的命令行工具
	Pyro	Python 机器人编程库
	PyUserInput	跨平台的，控制鼠标和键盘的模块
	Pingo	Pingo 为类似 Raspberry Pi，pcDuino， Intel Galileo 等设备提供统一的 API

7.6.3 PyPI 网站和 PIP 模块管理工具

PyPI(Python Package Index)是 python 官方的第三方库的仓库，所有人都可以下载第三方库或上传自己开发的库到 PyPI。PyPI 推荐使用 pip 包管理器来下载第三方库。

pip 是一个现代的，通用的 Python 包管理工具。提供了对 Python 包的查找、下载、安装、卸载的功能。pip 可正常工作在 Windows、Mac OS、Unix/Linux 等操作系统上，但是需要至少 2.6+和 3.2+的 CPython 或 PyPy 的支持。python 2.7.9 和 3.4 以后的版本已经内置累 pip 程序，所以不需要安装。

7.6.4 安装第三方扩展库

第三方库有数十万种之多，以 pillow 库为例讲解第三方扩展库的安装。pillow 是 Python 平台事实上的图像处理标准库，本节以安装 pillow 为例，给大家介绍第三方库的两种常用的安装方法。

（1）命令行下安装

安装第三方 pillow 图像库，在命令行提示符下输入如下命令：

```
pip  install  pillow
```

安装完，输入 pip show pillow， 进行确认。如图 7-38 所示：

```
C:\Users\Administrator>pip show pillow
Name: Pillow
Version: 5.3.0
Summary: Python Imaging Library (Fork)
Home-page: http://python-pillow.org
Author: Alex Clark (Fork Author)
Author-email: aclark@aclark.net
License: Standard PIL License
Location: c:\program files (x86)\python36-32\lib\site-packages
Requires:
Required-by:
```

图 7-38 确认 pillow 库安装成功

（2）Pycharm 中直接安装到项目中

在 Pycharm 中安装，首先点击菜单： file-->setting-->Project 本项目名-->Project Interpreter。出现如图 7-39 的界面。

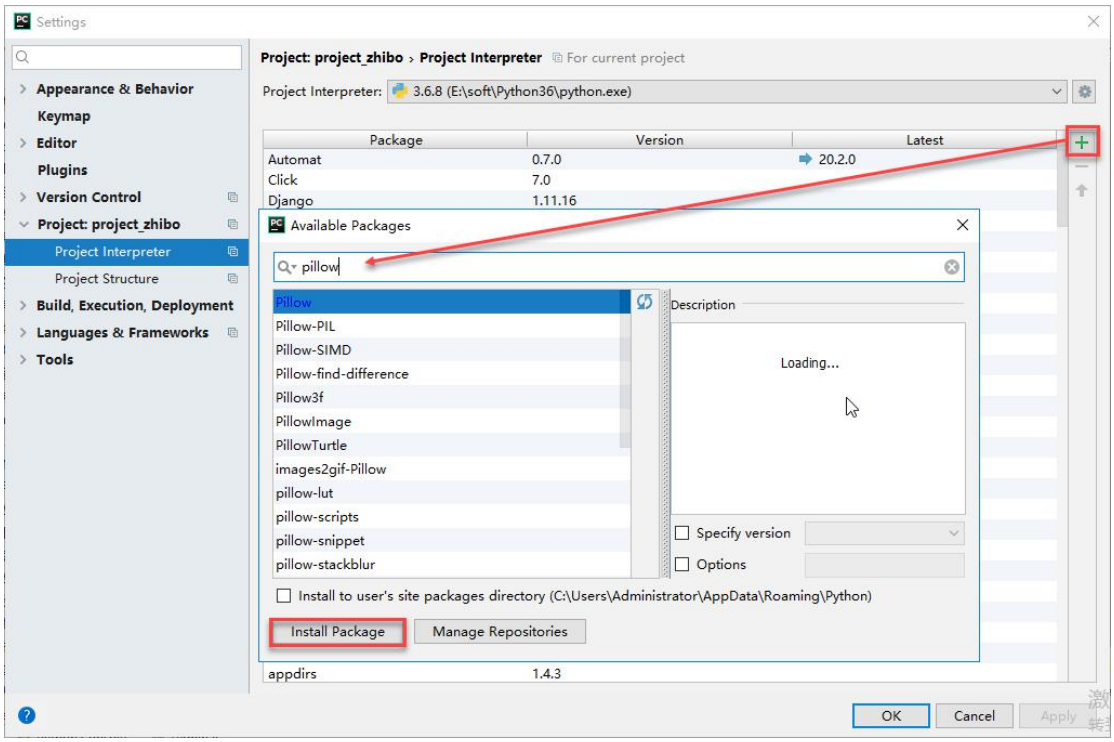


图 7-39 Pycharm 中安装 pillow 库

然后点击右上角“+”，输入要安装的第三方库“pillow”，再点击按钮“Install Package”，等待安装即可，几秒种后，即提示安装成功。如图 7-40 所示：

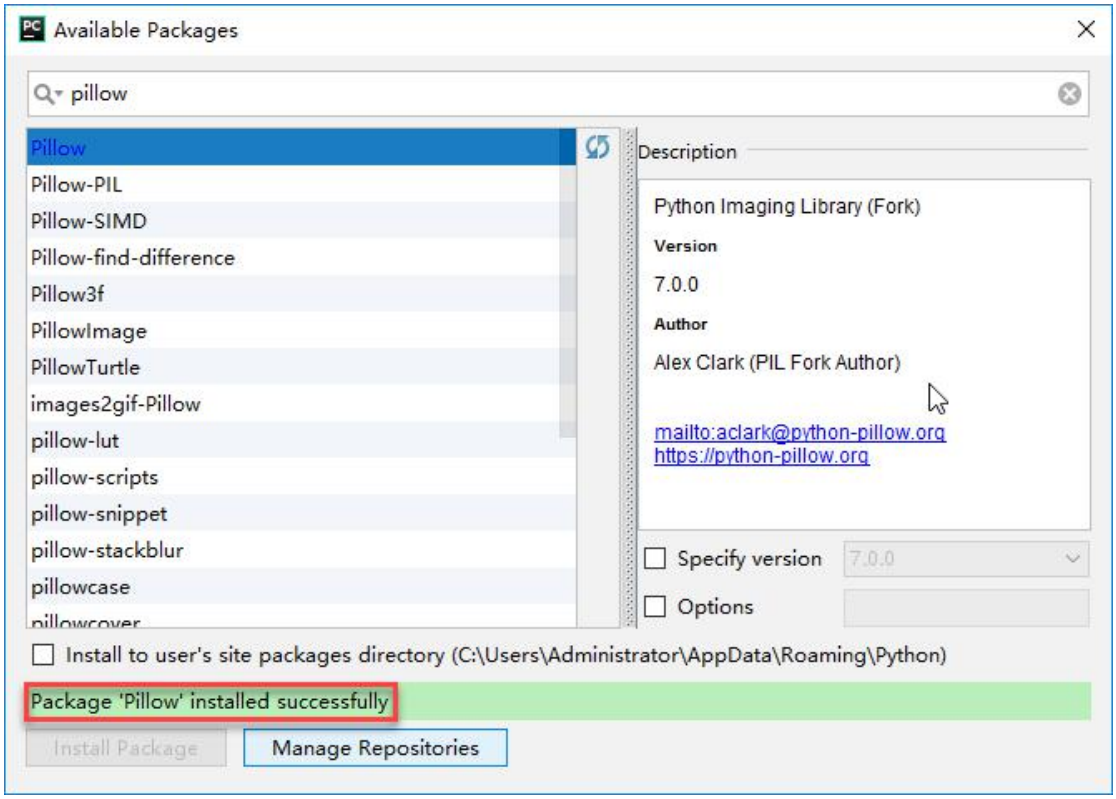


图 7-40 安装 pillow 库成功

习题

一、选择题

1. 执行如下代码：

```
import time
print(time.time())
```

以下选项中描述错误的是（）

A. time 库是 Python 的标准库

B. 可使用 time.ctime()，显示为更可读的形式

C. time.sleep(5) 推迟调用线程的运行，单位为毫秒

D. 输出自 1970 年 1 月 1 日 00:00:00 AM 以来的秒数

2. 执行后可以查看 Python 的版本的是（）

A. import sys print(sys.Version)

B. import system print(system.version)

C. import system print(system.Version)

D. import sys print(sys.version)

3. 以下选项中不是 Python 数据分析的第三方库的是（）

A. numpy B. scipy C. pandasD. requests

4. 关于 time 库的描述，以下选项中错误的是（）

A. time 库提供获取系统时间并格式化输出功能

B. time.sleep(s)的作用是休眠 s 秒

C. time.perf_counter()返回一个固定的时间计数值

D. time 库是 Python 中处理时间的标准库

5. 关于 random 库，以下选项中描述错误的是（）

A. 设定相同种子，每次调用随机函数生成的随机数相同

B. 通过 from random import *可以引入 random 随机库

C. 通过 import random 可以引入 random 随机库

D. 生成随机数之前必须要指定随机数种子

二、简答题

1. 简述导入模块的方法。
2. 解释 Python 脚本程序“__name__”变量及其作用。
3. 包中__init__的使用。
4. 简述 Python 解释器搜索模块位置的顺序。
5. 简述如何安装第三方库。

三、编码题

1. 使用 time 模块，根据年月日，返回星期几。

2. 使用 time 模块，根据年月日，算出活了多少天。
3. 写一个 6 位随机验证码程序（使用 random 模块)，要求验证码中至少包含一个数字、一个小写字母、一个大写字母。
4. 编程一个自定义模块，完成对该模块的发布和安装。

第八章 错误和异常

在编写程序的过程中，程序员通常希望识别正常执行的代码和执行异常的代码。这种异常可能是程序的错误，也可以是不希望发生的事情。为了能够处理这些异常，可以在所有可能发生这种地方使用条件语句进行判断。但这么做既效率低，也不灵活，而且还无法保证条件语句覆盖所有可能的异常。为了更好地解决这个问题，Python 语言提供了非常强大的异常处理机制。通过这种异常处理机制，可以直接处理所有发生的异常，也可以选择忽略这些异常。

通过阅读本章，你可以：

- 了解异常、错误的概念
- 掌握如何使用 try- except 语句捕捉异常
- 掌握 else 子句的使用方法
- 学会自定义异常
- 学会 raise 语句抛出异常
- 学会 try 语句中的 finally 子句的使用
- 掌握 Pycharm 开发环境中断点调试

8.1 错误

程序员在编写程序的时候，错误往往是难以避免的，可能是因为语法用错了，也可能是拼写错了，当然还可能有其他莫名其妙的错误，比如冒号写成了全角的了等。总之，编程中有相当一部分就是要不停地修正错误。

Python 中的常见错误之一是语法错误（syntax errors），也是常见的错误。示例如下：

【示例 8-1】错误演示

```
#语法错误
for i in range(10)
    print(i)
```

执行结果如图 8-1 所示：

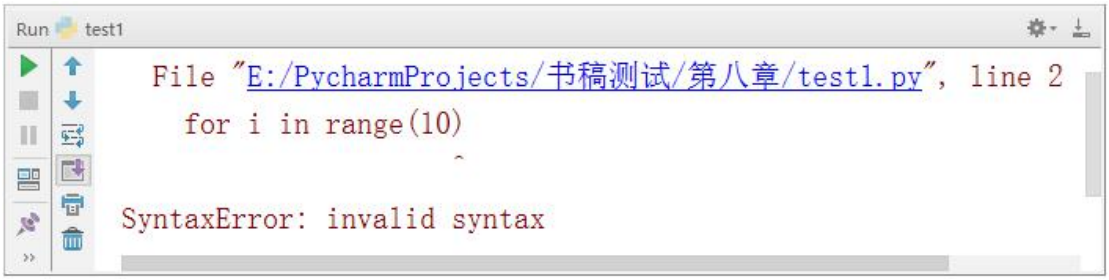


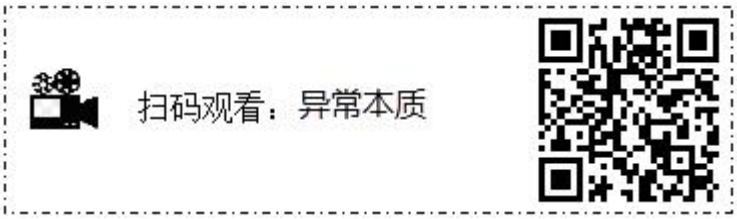
图 8-1 示例 8-1 执行结果

上面代码因为缺少冒号”:”(英文半角)，导致解释器无法解释，于是报错。这个报错行为是由 Python 的语法分析器完成的，并且检测到了错误所在文件和行号，还可以向上箭头“^”标识错误位置，最后一行显示错误类型。

常见的错误之二是在没有语法错误时，会出现逻辑错误。逻辑错误可能会由于不完整或者不合法的输入导致，也可能是无法生成、计算等。或者是其他逻辑错误。

8.2 异常

当 Python 检测到一个错误时，解释器就无法继续执行下去，于是抛出相应的信息，这些信息称之为异常。



让程序抛出异常的情况很多，但可以分为两大类：系统自己抛出的异常和主动抛出的异常。如果由于执行了某些代码(如分母为 0 的除法)，系统会抛出异常，这种异常是系统自动抛出的(由 Python 解释器抛出)。还有一种异常，是由于执行 raise 语句抛出的异常，这种异常属于主动抛出的异常。

python 中，引进了很多用来描述和处理异常的类，称为异常类。异常类定义中包含了该类异常的信息和对异常进行处理的方法。下面较为完整的展示了 python 中内建异常类的继承层次，如图 8-2 所示：

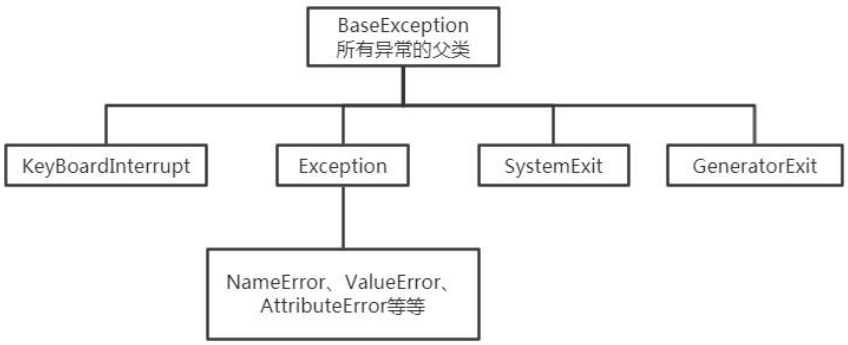


图 8-2 内建异常类的继承层次

8.3 解决异常问题的态度

学习完异常相关知识点，只是开始对异常有些认识，不意味着你会调试任何异常；调试异常，需要大量的经验作为基础。因此，大家不要在此停留，继续往后学习。碰到每个异常，都要花心思去解决而不要动不动张口问人。通过自己的努力无法解决，再去找老师同学帮助解决。

解决每一个遇到的异常，建议大家遵循如下三点：

（1）不慌张，细看信息，定位错误。看清楚报的错误信息，并定位发生错误的地方。

（2）百度并查看十个相关帖子。将异常类信息进行百度，至少查看十个以上的相关帖子。

（3）以上两步仍然无法解决，找老师和同学协助解决。

8.4 异常解决的关键：定位

当发生异常时，解释器会报相关的错误信息，并会在控制台打印出相关错误信息。只需按照从上到下的顺序即可追溯（Trackback）异常发生的过程，最终定位引起错误的那一行代码。

【示例 8-2】追溯异常发生的过程

```
def a():
    print("run in a() start! ")
    num = 1/0
    print("run in a() end! ")
def b():
    print("run in b() start!")
    a()
    print("run in b() end! ")

def c():
    print("run in c() start!")
    b()
    print("run in c() end! ")
print("step1")
c()
print("step2")
```

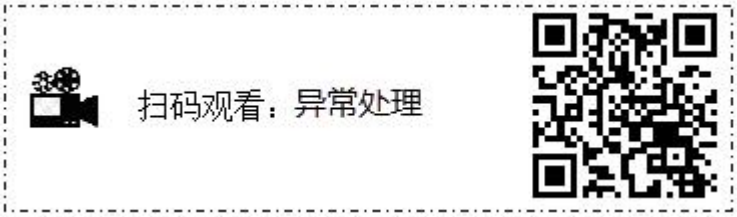
执行结果如图 8-3 所示：

图 8-3 示例 8-2 执行结果

从上述示例执行结果可以看到，该示例抛出了不止一个异常。按照从上到下的顺序进行追溯（Trackback）异常发生的过程，最终定位发现引发异常的是第三行代码。

8.5 异常处理

在程序运行过程中，常见的异常有除零、下标越界等。在 Python 中，当这些异常产生时，可以捕获这些异常，并编写相关的处理语句，使程序不会意外中断。



8.5.1try…except 语句捕获异常

异常处理功能实现了处理程序运行时出现的任何意外的处理方法。最常用的捕捉异常方式为使用 try/except 语句。try/except 语句当中，首先是用 try 语句检测 try 语句块中的错误，然后让 except 语句捕获异常信息并处理。语法格式如下：

```
try:
    <语句>
except <异常参数>:
    <语句>          #如果在 try 部份引发了异常，except 可被定义多次
```

【示例 8-3】计算两数的商(不使用 try…exception 语句)

```
a=int(input('请输入分子：'))
b=int(input('请输入分母：'))
print('a/b 的结果:',a/b)
```

```
print('程序运行结束')
```

执行结果如图 8-4 所示：

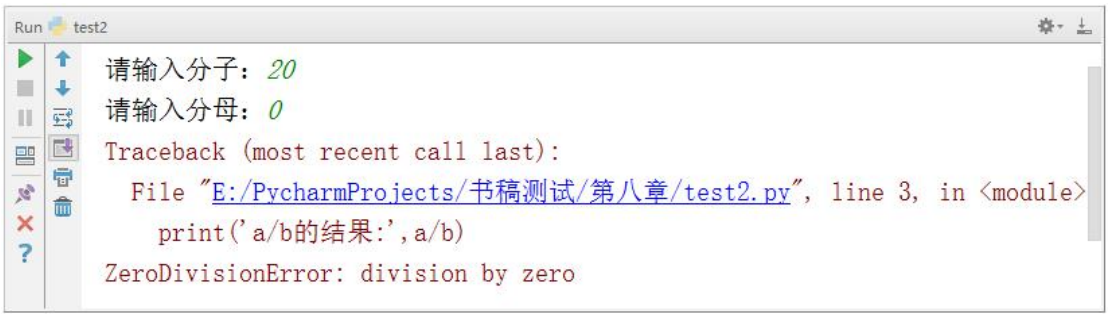


图 8-4 示例 8-3 执行结果

执行上面的代码，分子输入任意的数，分母输入 0，会抛出如图 8-2 所示的异常，从而导致程序中断，后面的代码就不会被执行。

由于输入是不可控的，所以当采集用户输入的数据时，应该使用 try…except 语句对相关代码进行异常捕获，示例如下：

【示例 8-4】try…except 的使用

```
a=int(input('请输入分子：'))
b=int(input('请输入分母：'))
try:
    print('a/b 的结果:',a/b)
except:
    print('分母不能为 0')
print('程序运行结束')
```

执行结果如图 8-5 所示：

图 8-5 示例 8-4 执行结果

执行上面的代码，分子输入 20，分母输入 0，程序出现异常，执行 except 进行捕获异常，程序不会被中断，后面的代码被执行。

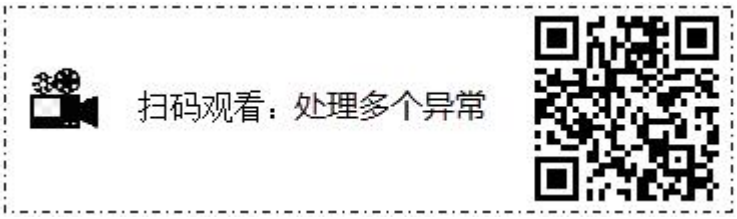
try…except 语句需注意：

- try…except 语句是一个代码块，所以 try 和 except 后面都要加上冒号（:）。
- try 和 except 之间是正常执行的语句，如果 try 中的代码不发生异常，这时 except 部分

的代码是不会执行的。如果 try 中的代码发生了异常，那么异常后面的代码都不会被执行了，会最直接跳到 except 子句去执行 except 代码块中的代码。

- 如果 except 关键字后面没有指定任何异常类，那么 except 部分可以捕获任何的异常。

8.5.2 处理多个异常



处理多个异常并不是因为同时报出多个异常，程序在运行中只要遇到一个异常就会捕获，所以，每次捕获到的异常一定是一个。所谓处理多个异常的意思是可以容许捕获不同的异常，由不同的 except 子句处理。语法格式如下：

```
try:
    <语句>
except   异常类 1:
    <语句>
except   异常类 2:
    <语句>
    ...
except    异常类 n:
except:          #捕获其他未指定的异常
```

【示例 8-5】处理多个异常

```
a=input('请输入分子：')
b=input('请输入分母：')
try:
    print('a/b 的结果:',int(a)/int(b))
except ZeroDivisionError:
    print('分母不能为 0')
except ValueError:
    print('输入的值必须是数字')
except:                          #捕获其他未指定的异常
    print('****其他异常****')
print('程序运行结束')
```

运行上面的程序，如果输入的内容不是数字，执行结果如图 8-6 所示：

图 8-6 示例 8-5 执行结果

运行上面的程序，如果输入的分母是 0，执行结果如图 8-7 所示：

图 8-7 示例 8-5 执行结果

从上面的代码可以看出，如果有多个异常 except，try 里面遇到一个异常，就转到相应的 except 子句。

8.5.3 用同一个代码块处理多个异常

处理多个异常时，除了用多个 except 之外，还可以在一个 except 后面放多个异常参数，语法格式如下：

```
try:
    ...
except(异常类 1,异常类 2，异常类 3,...,异常类 n):
    ...
```

表 8-1 描述了一些最重要的内置异常。

表 8-1 内置异常类

异常名称	说明
ArithmeticError	所有数值计算错误的基类
AssertionError	断言语句失败
AttributeError	对象没有这个属性
BaseException	所有异常的基类
DeprecationWarning	关于被弃用的特征的警告
EnvironmentError	操作系统错误的基类
EOFError	没有内建输入,到达 EOF 标记

异常名称	说明
Exception	常规错误的基类
FloatingPointError	浮点计算错误
FutureWarning	关于构造将来语义会有改变的警告
GeneratorExit	生成器(generator)发生异常来通知退出
ImportError	导入模块/对象失败
IndentationError	缩进错误
IndexError	序列中没有此索引(index)
IOError	输入/输出操作失败
KeyboardInterrupt	用户中断执行(通常是输入^C)
KeyError	映射中没有这个键
LookupError	无效数据查询的基类
MemoryError	内存溢出错误(对于 Python 解释器不是致命的)
NameError	未声明/初始化对象 (没有属性)
NotImplementedError	尚未实现的方法
OSError	操作系统错误
OverflowError	数值运算超出最大限制
OverflowWarning	旧的关于自动提升为长整型(long)的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
ReferenceError	弱引用(Weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
StandardError	所有的内建标准异常的基类
StopIteration	迭代器没有更多的值
SyntaxError	Python 语法错误
SyntaxWarning	可疑的语法的警告
SystemError	一般的解释器系统错误
SystemExit	解释器请求退出
TabError	Tab 和空格混用
TypeError	对类型无效的操作
UnboundLocalError	访问未初始化的本地变量
UnicodeDecodeError	Unicode 解码时的错误

UnicodeEncodeError	Unicode 编码时错误
UnicodeError	Unicode 相关的错误
UnicodeTranslateError	Unicode 转换时错误
UserWarning	用户代码生成的警告
ValueError	传入无效的参数
Warning	警告的基类
WindowsError	系统调用失败
ZeroDivisionError	除(或取模)零 (所有数据类型)

【示例 8-6】一个 except 后面放多个异常参数的使用

```
a=input('请输入分子：')
b=input('请输入分母：')
try:
    print('a/b 的结果:',int(a)/int(b))
except (ZeroDivisionError,ValueError):
    print('请输入正确的值')
except:                          #捕获其他未指定的异常
    print('****其他异常****')
print('程序运行结束')
```

执行结果如图 8-8 所示：

图 8-8 示例 8-6 执行结果

注意：

- except 后面如果是多个参数，一定要用圆括号括起来。

8.5.4 捕获对象

在上面的示例中使用 except 子句捕获了多个异常，但都是根据异常类输出自定义的异常信息。如果希望输出默认错误提示信息，需要为异常类指定一个变量，也可以称为异常对

象。也是给这些异常对象一个名字而已。为异常对象指定名字需要用 as 关键字。语法格式如下：

```
try:
    ...
except  异常类  as   e:
except(异常类 1,异常类 2，异常类 3,...,异常类 n)   as e:
    ...
```

如果使用 print 函数输出 e，会将通过构造方法参数传给异常对象的异常信息输出到控制台。

【示例 8-7】异常对象的使用

```
a=input('请输入分子：')
b=input('请输入分母：')
try:
    print('a/b 的结果:',int(a)/int(b))
except ZeroDivisionError as e:
    #输出相应的异常信息
    print(e)
except ValueError as e:
    #输出相应的异常信息
    print(e)
except:                          #捕获其他未指定的异常
    print('****其他异常****')
print('程序运行结束')
```

运行上面的程序，如果输入的内容不是数字，执行结果如图 8-9 所示：

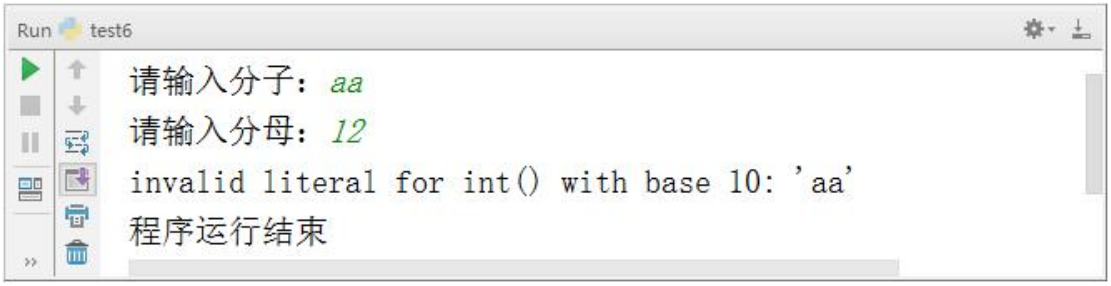
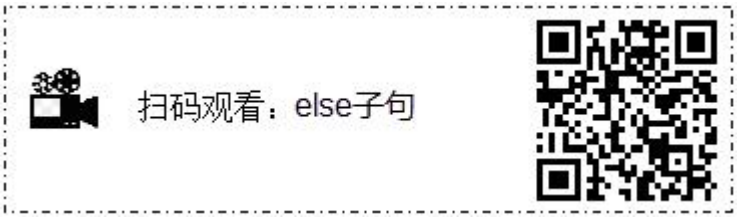


图 8-9 示例 8-7 执行结果

运行上面的程序，如果输入的分母是 0，执行结果如图 8-10 所示：

图 8-10 示例 8-7 执行结果

8.5.5 else 子句



与循环语句类似，try…except 语句也有 else 语句。与 except 子句正好相反，except 子句中的代码会在 try 和 except 之间的代码抛出异常时执行，而 else 子句会在 try 和 except 之间的代码正常执行后才执行。可以利用 else 子句的这个特性控制循环体的执行，如果没有任何异常抛出，那么循环体就结束，否则一直处于循环状态。语法格式如下：

```
try:
    ...
except:
    #抛出异常时执行这段代码
    ...
else:
    #正常执行后执行这段代码
    ...
```

【示例 8-8】else 子句的使用（没有异常）

```
try:
    print('I am try')
except:
    print('I am except')
else:
    print('I am else')
```

执行结果如图 8-11 所示：

图 8-11 示例 8-8 执行结果

【示例 8-9】else 子句的使用（有异常）

```
try:
    result=1/0
    print('I am try')
except:
    print('I am except')
else:
    print('I am else')
```

执行结果如图 8-12 所示：

图 8-12 示例 8-9 执行结果

【示例 8-10】else 子句的使用

通过 while 循环控制输入，并计算两数的商。如果输入的有误，就重新输入，直到输入正确并得到结果，然后程序结束。

```
while True:
    try:
        a=input('请输入分子：')
        b=input('请输入分母：')
        result=int(a)/int(b)
    except Exception as e:
        print(e)
        print('****重新输入****')
    else:
        print('程序结束')
        break;
```

执行结果如图 8-13 所示：

图 8-13 示例 8-10 执行结果

8.5.6 finally 子句

捕获异常语句的最后一个子句是 finally。将所有需要最后收尾的代码放到 finally 子句中。不管是正常执行，还是抛出异常，最后都会执行 finally 子句中代码，所以应该在 finally 子句中放置关闭资源的代码，如关闭文件、关闭数据连接等。finally 子句语法格式如下：

```
try:
    <语句>
except <异常参数>:
    <语句>          #如果在 try 部份引发了异常，except 可被定义多次
else:
    <语句>          #如果没有异常发生就执行 else 块，else 为可选块
finally：
    <语句>          #finally 是可选块，无论是否发生异常，如果有 finally 的话都执行
```

【示例 8-11】finally 子句的使用一

```
try:
    a=input('请输入分子：')
    b=input('请输入分母：')
```

```
    print('a/b 的结果:',int(a)/int(b))
except Exception as e:
    print(e)
finally:
    print('删除对象')
    del a,b
```

如果输入分母为 0 时，执行结果如图 8-14 所示：

图 8-14 示例 8-11 执行结果

如果输入都是数字时，执行结果如图 8-15 所示：

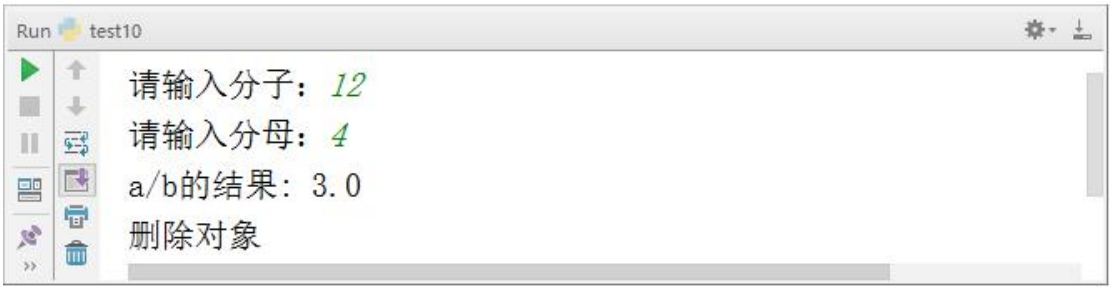


图 8-15 示例 8-11 执行结果

从上面的代码可以看到，尽管无论代码中是否抛出异常，finally 子句中的代码都会执行。

如果使用 return 语句退出函数，那么会首先执行 finally 子句中的代码，才会退出函数。因此并不用担心 finally 子句中的代码不会被执行，只要为 try 语句加上了 finally 子句，并且程序执行流程进入 try 语句，finally 子句中的代码是一定会执行的。

【示例 8-12】finally 子句的使用二

```
def funTest():
    try:
        a=input('请输入分子：')
        b=input('请输入分母：')
        result=int(a)/int(b)
        return result        #使用 result 返回结果
    except Exception as e:
        print(e)
```

```
    finally:
        print('删除对象')
        del a,b
#调用函数
result=funTest()
print('返回结果：',result)
```

执行结果如图 8-16 所示：

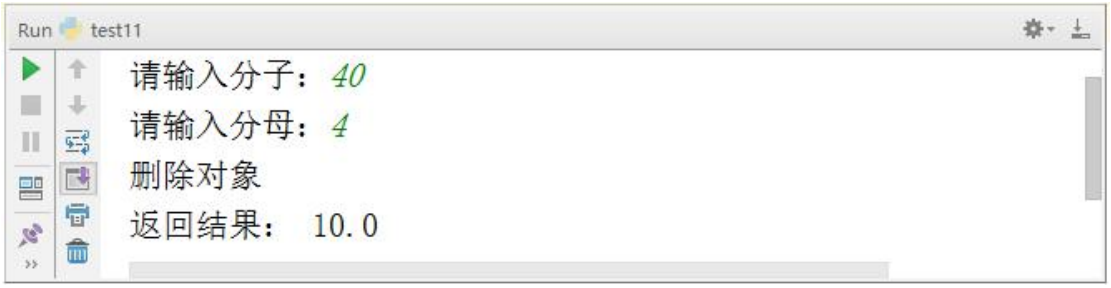
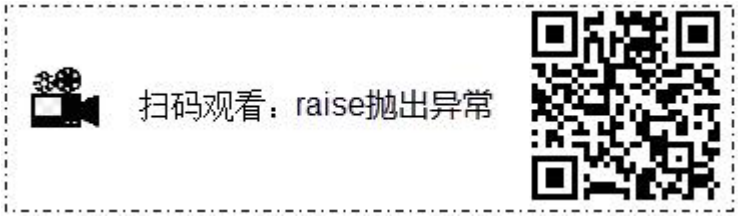


图 8-16 示例 8-12 执行结果

从上面的代码可以看到，在 funTest 函数中通过 return 语句退出函数时，会首先执行 finally 子句中的代码，然后再退出函数。

8.6 抛出异常

8.6.1 raise 语句抛出异常



在 Python 中异常可以是系统自动抛出的，也可以由程序员使用 raise 语句手动抛出。raise 语句可以使用一个类（必须是 Exception 或者 Exception 类的子类）或异常对象抛出异常。如果使用类，系统会自动创建类的实例。语法格式如下：

```
raise [Exception [, args [, traceback]]]
```

语句中 Exception 是异常的类型（例如，NameError）参数标准异常中任一种，args 是自己提供的异常参数。最后一个参数是可选的（在实践中很少使用），如果存在，是跟踪异常对象。

【示例 8-13】raise 语句的使用

```
#输出等级的方法，如果等级<1 则抛出异常
def funTest(level):
    if level<1:
        raise Exception('等级不合法',level)
```

```
    print('您输入的等级是：',level)


#调用函数,应该该函数可能有异常，所以加 try..except
try:
    funTest(1)
except Exception as e:
    print(e)
```

调用函数参数传递 1，执行结果如图 8-17 所示：

图 8-17 示例 8-13 执行结果

调用函数参数传递 0，执行结果如图 8-18 所示：

图 8-18 示例 8-13 执行结果

8.6.2 assert 语句

在 Python 中，使用 assert 语句同样可以引发异常。但与 raise 语句不同，assert 语句是在条件测试为假时，才引发异常。assert 语句的语法格式如下：

```
assert condition
```

如果 condition 为 false，那么 raise 一个 AssertionError。

【示例 8-14】assert 语句的使用一

```
assert 1==2
```

执行结果如图 8-19 所示：

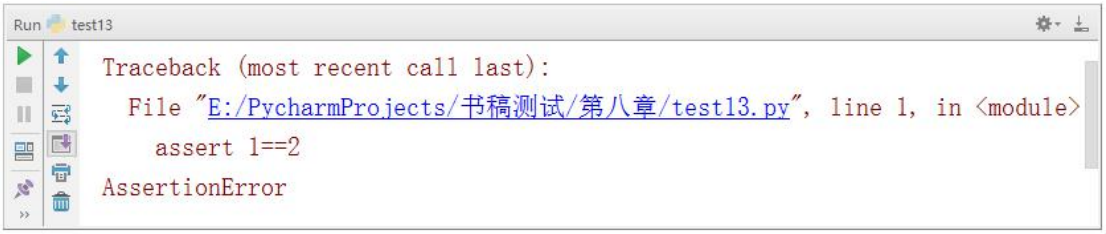


图 8-19 示例 8-14 执行结果

assert 的应用情景与其翻译的意思“断言”一样，即当程序运行到某个节点的时候，就断定某个变量的值必然是什么，或者对象必然拥有某个属性等，简单说就是断定什么东西必

然是什么，如果不是，就会抛出错误。

【示例 8-15】assert 语句的使用二

```
class Account:
    def __init__(self):
        self.balance=0
    def deposit(self,amount):
        assert amount>0
        self.balance+=amount
if __name__=='__main__':
    a=Account()
    a.deposit(-10)
```

执行结果如图 8-20 所示：

图 8-20 示例 8-15 执行结果

从上面代码可以看出，deposit()方法的参数 amount 值必须大于零的，如果不满足条件就会报错。

8.6.3 自定义异常

在 Python 中，可以通过继承 Exception 类来创建自己的异常称为自定义异常。异常类和其他的类并没有区别。最简单的自定义异常类就是一个空的 Exception 类的子类。语法格式如下：

```
class MyException(Exception):
    pass
```

下面通过继承 Exception 类来生成一个 SexException 类。如果给 Student 类的属性 sex 赋值时候，如果不是“男”或者“女”就抛出自定义的 SexException。示例如下：

【示例 8-16】自定义异常

```
#自定义异常
class SexException(Exception):
```

```
    def __init__(self,msg):
        self.msg=msg
class Student:
    def __init__(self,name):
        self.name=name
        # self.__sex=sex
    @property
    def sex(self):
        return self.__sex
    @sex.setter
    def sex(self,sex):
        if sex=='男' or sex=='女':
            self.__sex=sex
        else:
            raise SexException('性别的参数值不正确，只能是男或者女')
    def __str__(self):
        return '姓名：{0},性别：{1}'.format(self.name,self.__sex)
s1=Student('张三')
s1.sex='abc'      #给 s1 的属性 sex 赋值，
print(s1)
```

执行结果如图 8-21 所示：

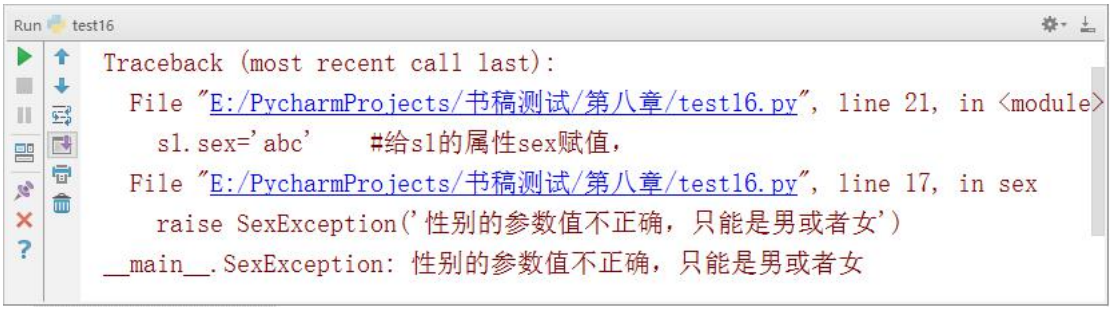


图 8-21 示例 8-16 执行结果

从上面的代码中可以看出，对 sex 属性赋值时候有可能有异常，需要进行捕获异常。

【示例 8-17】捕获自定义异常

```
#自定义异常
class SexException(Exception):
    def __init__(self,msg):
        self.msg=msg
```

```
class Student:
    def __init__(self,name):
        self.name=name
        # self.__sex=sex
    @property
    def sex(self):
        return self.__sex
    @sex.setter
    def sex(self,sex):
        if sex=='男' or sex=='女':
            self.__sex=sex
        else:
            raise SexException('性别的参数值不正确，只能是男或者女')
    def __str__(self):
        return '姓名：{0},性别：{1}'.format(self.name,self.__sex)
s1=Student('张三')
try:
    s1.sex='abc'      #给 s1 的属性 sex 赋值，
    print(s1)
except Exception as e:
    print(e)
```

执行结果如图 8-22 所示：

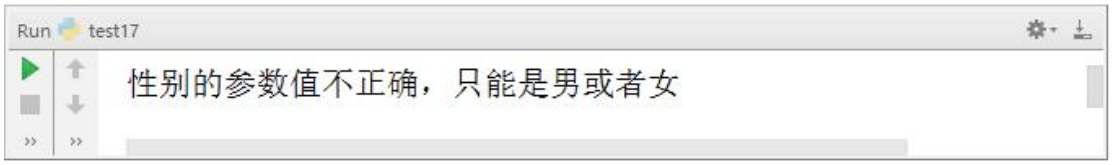


图 8-22 示例 8-17 执行结果

8.7 Pycharm 开发环境调试

Pycharm 进行调试的核心是设置断点。程序执行到断点时，暂时挂起，停止执行。就像看视频按下停止一样，可以详细的观看停止处的每一个细节。

8.7.1 断点

程序运行到此处，暂时挂起，停止执行。可以详细在此时观察程序的运行情况，方便做出进一步的判断。

设置断点：

（1）在行号后面单击即可增加断点。

```
4    def aa():
5        print("run in aa")
6        num1 = 3
7        num2 = num1*4
```

图 8-23 增加断点

（2）在断点上再单击即可取消断点

8.7.2 调试视图

进入调试视图的方式如下：

（1）单击工具栏上的按钮，如图 8-24 所示：

图 8-24 工具栏

（2）右键单击编辑区，点击：debug “模块名”。

（3）快捷键：shift+F9。

进入调试视图后，布局如图 8-25 所示：

图 8-25 调试视图

左侧为“浏览帧”：

调试器列出断点处，当前线程正在运行的方法，每个方法对应一个“栈帧”。最上面的是当前断点所处的方法。

变量值观察区：

调试器列出了断点处所在方法相关的变量值。我们可以通过它，查看变量的值的变化。也可以通过，增加要观察的变量。

8.7.3 调试操作

在调试视图中，通过点击操作按钮进行调试，如图 8-26 所示：

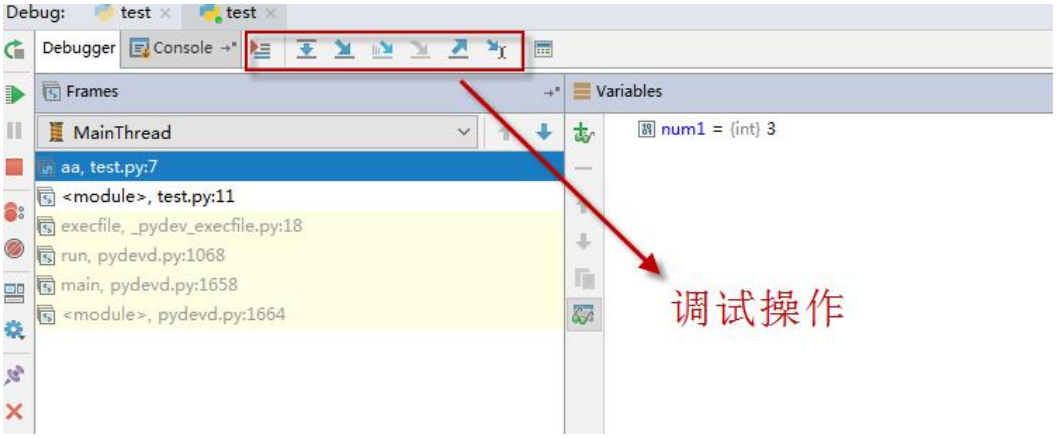


图 8-26 调试操作

表 8-2 调试按钮含义

中文名称	英文名称	图标和快捷键	说明
显示当前所有断点	show Execution Point	Alt+F10	
单步调试：遇到函数跳过	step over	F8	若当前执行的是一个函数，则会把这个函数当做整体一步执行完。不会进入这个函数内部
单步调试：遇到函数进入	step into	F7	若当前执行的是一个函数，则会进入这个函数内部
跳出函数	step out	Shift+F8	当单步执行到子函数内时，用 step out 就可以执行完子函数余下部分，并返回到上一层函数
执行的光标处	run to cursor	Alt+F9	一直执行，到光标处停止，用在循环内部时，点击一次就执行一个循环

习题

一、选择题

1. 关于程序的异常处理，以下选项中描述错误的是（）

A. 程序异常发生经过妥善处理可以继续执行

B. 异常语句可以与 else 和 finally 保留字配合使用

C. 编程语言中的异常和错误是完全相同的概念

D. Python 通过 try、except 等保留字提供异常处理功能

2. 以下 Python 语言关键字在异常处理结构中用来捕获特定类型异常的是：

A. for

B. lambda

C. in

D. Expect

3. 以下关于异常处理的描述，正确的是（）

A. try 语句中有 except 子句就不能有 finally 子句

B. Python 中，可以用异常处理捕获程序中的所有错误

C. 引发一个不存在索引的列表元素会引发 NameError 错误

D. Python 中允许利用 raise 语句由程序主动引发异常

4. 用户输入整数的时候不合规导致程序出错，为了不让程序异常中断，需要用到的语句是（）

A. if 语句

B. eval 语句

C. 循环语句

D. try-except 语句

5. 以下关于异常处理的描述，错误的选项是（）

A. Python 通过 try、except 等保留字提供异常处理功能

B. ZeroDivisionError 是一个变量未命名错误

C. NameError 是一种异常类型

D. 异常语句可以与 else 和 finally 语句配合使用

二、解答题

1. 错误和异常的区别。
2. 常见异常有哪些。
3. Python 异常处理中关键字 try、except、finally 分别代表什么含义。
4. 简述关键字 raise 的使用。
5. Pycharm 开发环境断点调试过程。

三、编码题

1. 编写函数 devide(x, y)，x 为被除数，y 为除数。要求考虑如下几点：
   1) 被零除时，输出"division by zero! "；
   2) 类型不一致时，强制转换为整数再调用本函数；
   3) 若没有上述异常则输出计算结果。
2. 自己定义一个异常类，继承 Exception 类, 捕获下面的过程：判断输入的字符串长度是否小于 5， 如果小于 5，比如输入长度为 3 则抛出自定义异常“当前输入长度为 3，输入的长度至少为 5”，大于 5 输出“输出该字符串”。
3. 电脑随机生成 1~100 随机数，用户输入一个数字，电脑提示用户大或者小，猜错，继续提示；猜对，则程序终止。
4. 编写一个计算减法的方法，当第一个数小于第二个数时，抛出“被减数不能小于减数"的异常。

第九章 文件和流

IO 在计算机中指 Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由 CPU 这个超快的计算核心来执行，涉及到数据交换。与用户交互的部分只是通过 input 和 print 函数在控制台输入/输出，但真正有价值的应用需要程序将结果进行保存，还会读取外部数据作为数据源，这就是本章要介绍的文件和流的相关内容。

通过阅读本章，你可以：

- 了解打开文件的方法
- 掌握读文件的方法
- 掌握写文件的方法
- StringIO 和 BytesIO 的使用
- 掌握 JSON 格式数据处理
- 掌握对 CSV 文件读写操作

9.1 文本文件和二进制文件

按文件中数据组织形式，把文件分为文本文件和二进制文件两大类。

（1）文本文件

文本文件存储的是普通“字符”文本，python 默认为 unicode 字符集（两个字节表示一个字符，最多可以表示：65536 个），可以使用记事本程序打开。但是，像 word 软件编辑的文档不是文本文件。

（2）二进制文件

二进制文件把数据内容用“字节”进行存储，无法用记事本打开。必须使用专用的软件解码。常见的有：MP4 视频文件、MP3 音频文件、JPG 图片、doc 文档等等。

9.2 文件操作相关模块

Python 标准库中，如表 9-1 所示是文件操作相关的模块。

表 9-1 文件操作相关模块

名称	说明
io 模块	文件流的输入和输出操作 input output
os 模块	基本操作系统功能，包括文件操作
glob 模块	查找符合特定规则的文件路径名
fnmatch 模块	使用模式来匹配文件路径名
fileinput 模块	处理多个输入文件
filecmp 模块	用于文件的比较
cvs 模块	用于 csv 文件处理
pickle 和 cPickle	用于序列化和反序列化

名称	说明
xml 包	用于 XML 数据处理
bz2、gzip、zipfile、zlib、tarfile	用于处理压缩和解压缩文件（分别对应不同的算法）

9.3 打开文件

Python open() 方法用于打开一个文件，并返回文件对象，文件对象常用的函数如表 9-2 所示：

表 9-2 文件对象常用的函数

方法	描述
file.close()	关闭文件。关闭后文件不能再进行读写操作。
file.flush()	刷新文件内部缓冲，直接把内部缓冲区的数据立刻写入文件, 而不是被动的等待输出缓冲区写入。
file.fileno()	返回一个整型的文件描述符(file descriptor FD 整型), 可以用在如 os 模块的 read 方法等一些底层操作上
file.isatty()	如果文件连接到一个终端设备返回 True，否则返回 False。
file.next()	返回文件下一行。
file.read([size])	从文件读取指定的字节数，如果未给定或为负则读取所有。
file.readline([size])	读取整行，包括 "\n" 字符。
file.readlines([sizehint])	读取所有行并返回列表，若给定 sizeint>0，返回总和大约为 sizeint 字节的行, 实际读取值可能比 sizhint 较大, 因为需要填充缓冲区。
file.seek(offset[,whence])	设置文件当前位置
file.tell()	返回文件当前位置。
file.truncate([size])	截取文件，截取的字节通过 size 指定，默认为当前文件位置。
file.write(str)	将字符串写入文件，没有返回值。
file.writelines(sequence)	向文件写入一个序列字符串列表，如果需要换行则要自己加入每行的换行符。

在对文件进行处理过程都需要使用到 open()这个函数。语法格式如下：

```
open(file[,mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None])
```

其中第一个参数是必须的，指定要打开的文件名（可以是相对路径，也可以是绝对路径）。示例如下：

【示例 9-1】打开文件

```
#打开文件
f=open('testFile.txt')
f=open('E:/testFile.txt')
```

如果要打开的文件不存在，则抛出如图 9-1 所示的 FileNotFoundError 异常。

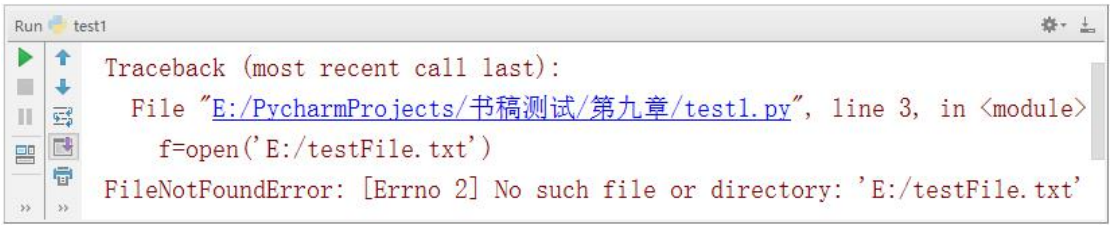


图 9-1 FileNotFoundError 异常

第二个参数用于指定文件模式(用一个字符串表示)。这里的文件模式是操作文件的方式，如只读、写入、追加等。表 9-3 描述了 Python3 支持的常用文件模式。

表 9-3 常用文件模式

文件模式	描述
‘r’	读模式（默认值）
‘w’	写模式
‘a’	追加模式
‘b’	二进制模式（可添加到其他模式中使用）
‘x’	排他的写模式（只能自己写）
‘t’	文本模式（默认值，可添加到其他模式中使用）
‘+’	读写模式（必须与其他文件模式一起使用）

写模式和追加模式的区别是如果文件存在，写模式会覆盖原来的文件，而追加模式会在原文件内容的基础上添加新的内容。

在表 9-3 最后一项‘+’文件模式，表示读写模式，必须与其他文件模式一起使用，如‘r+’、‘w+’、‘a+’。这三个组合文件模式都可以对文件进行读写操作，它们之间的区别如下：

□ r+：文件读写，如果文件不存在，会抛出异常；如果文件存在，会从当前位置开始写入新内容，通过 seek 函数可以改变当前的位置，也就是文件指针。

□ w+：文件可读写，如果文件不存在，会创建一个新文件；如果文件存在，会清空整个文件，并写入新内容。

□ a+：文件可读写，如果文件不存在，会创建一个新文件；如果文件存在，会将要写入的内容添加到原文件的最后，也就是说，使用‘a+’模式打开文件，文件指针会直接跳到文件的尾部，如果要使用 read 方法读取文件内容，需要使用 seek 方法改变文件指针，如果调用 seek(0)会直接将文件指针移动到文件开始的位置。

【示例 9-2】打开文件（指定模式）

```
f=open('testFile.txt','w') #以写模式打开文件
f=open('E:/testFile.txt','a') #以追加模式打开文件
```

9.4 文件读写

9.4.1 读文件

如果文件打开成功，接下来，调用 read()方法可以一次读取文件的全部内容，Python 把内容读到内存。调用 close()方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的。

【示例 9-3】使用 read()方法读文件

```
#打开文件
f=open('testFile.txt','r')
#读取文件内容
str=f.read()
print(str)
#关闭文件
f.close()
```

执行结果如图 9-2 所示：

图 9-2 示例 9-3 执行结果

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

【示例 9-4】read()方法读文件（添加 try…finally 捕获异常）

```
try:
    #打开文件
    f=open('testFile.txt','r')
```

```
    #读取文件内容
    str=f.read()
    print(str)
    #关闭文件
finally:
    if f:
        f.close()
```

每次都这么写实在太繁琐，所以，Python 引入了 with 语句来自动调用 close()方法。

【示例 9-5】read()方法读文件（使用 with 语句实现）

```
with open('testFile.txt','r') as f:
    #读取文件内容
    print(f.read())
```

调用 read()会一次性读取文件的全部内容，如果文件有 10G，内存就爆了，所以，要保险起见，可以反复调用 read(size)方法，其中 size 是一个整数，表示从文件指针指定的位置开始读取 size 个字节。

【示例 9-6】使用 read(size)方法读文件

```
try:
    #打开文件
    f=open('testFile.txt','r')
    #读取文件内容
    print(f.read(5))    #读取到 hello
    print(f.read(1))    #读取到空格
    print(f.read(6))    #读取到 python
    #关闭文件
finally:
    if f:
        f.close()
```

执行结果如图 9-3 所示：

图 9-3 示例 9-6 执行结果

9.4.2 二进制文件

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

【示例 9-7】读图片

```
with open('pen.png','rb') as f:
    print(f.read())
```

执行结果如图 9-4 所示：

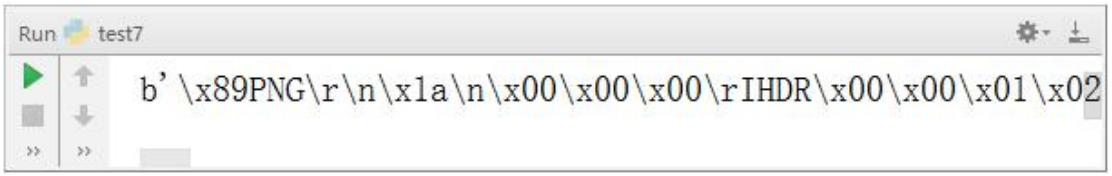


图 9-4 示例 9-7 执行结果

9.4.3 写文件

写文件和读文件是一样的，唯一区别是调用 open()函数时，传入的文件模式是'w'或者'wb'表示写文本文件或写二进制文件。向文件写入内容使用 write(string)方法，该方法返回写入文件的字节数。

【示例 9-8】write(string)写文件

```
#写文件
f=open('testWrite.txt','w')
f.write('hello python')
f.close()
```

执行结果如图 9-5 所示：

图 9-5 示例 9-8 执行结果

当写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用 close()方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用 close()的后果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，使用 with 语句比较好。

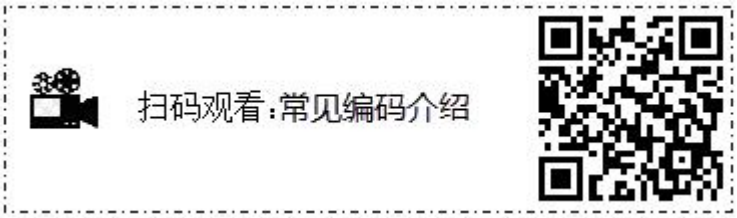
【示例 9-9】write(string)写文件(使用 with 语句实现)

```
with open('testWrite2.txt','w') as f:
    f.write('hello python use with')
```

执行结果如图 9-6 所示：

图 9-6 示例 9-9 执行结果

9.4.4 常用编码介绍



在操作文本文件时，经常会操作中文，这时候就经常会碰到乱码问题。为了让大家有能力解决中文乱码问题，这里简单介绍一下各种编码之间的关系。

常用编码之间的关系如图 9-7 所示：

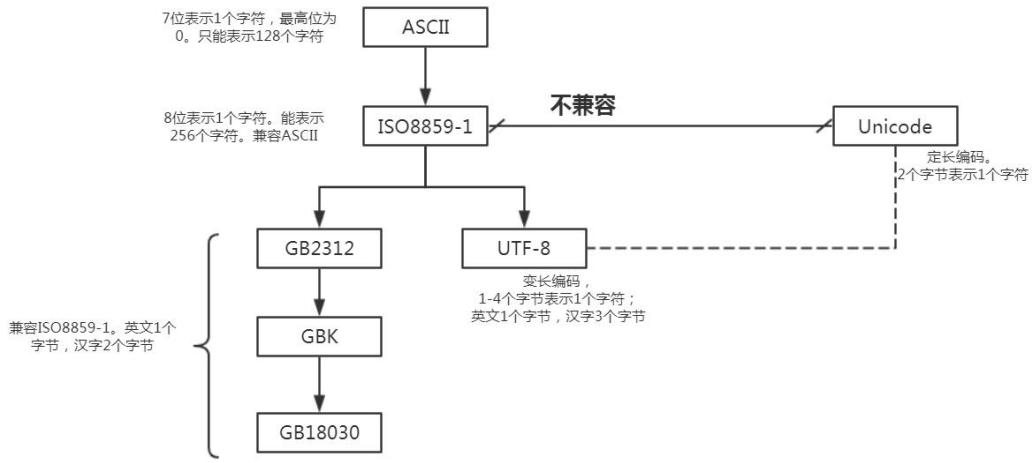


图 9-7 常用编码

（1）ASCII

全称为 American Standard Code for Information Interchange，美国信息交换标准代码，这是世界上最早最通用的单字节编码系统，主要用来显示现代英语及其他西欧语言。

ASCII 码用 7 位表示，只能表示 128 个字符。只定义了 27=128 个字符，用 7bit 即可完全编码，而一字节 8bit 的容量是 256，所以一字节 ASCII 的编码最高位总是 0。

0～31 表示控制字符如回车、退格、删除等；32～126 表示打印字符即可以通过键盘输入并且能显示出来的字符； 其中 48～57 为 0 到 9 十个阿拉伯数字，65～90 为 26 个大写英文字母，97～122 号为 26 个小写英文字母，其余为一些标点符号、运算符号等，具体可以参考 ASCII 标准表（大家自行百度，不在此赘述）。

（2）ISO8859-1

ISO-8859-1 又称 Latin-1，是一个 8 位单字节字符集，它把 ASCII 的最高位也利用起来，并兼容了 ASCII，新增的空间是 128，但它并没有完全用完。

在 ASCII 编码之上又增加了西欧语言、希腊语、泰语、阿拉伯语、希伯来语对应的文字符号，它是向下兼容 ASCII 编码。

（3）GB2312

GB2312 全称为信息交换用汉字编码字符集，是中国于 1980 年发布，主要用于计算机系统中的汉字处理。GB2312 主要收录了 6763 个汉字、682 个符号。

GB2312 覆盖了汉字的大部分使用率，但不能处理像古汉语等特殊的罕用字，所以后来出现了像 GBK、GB18030 这种编码。

GB2312 完全兼容 ISO8859-1。

（4）GBK

全称为 Chinese Internal Code Specification，即汉字内码扩展规范，于 1995 年制定。它主要是扩展了 GB2312，在它的基础上又加了更多的汉字，它一共收录了 21003 个汉字。

（5）GB18030

现在最新的内码字集于 2000 年发布，并于 2001 年强制执行，包含了中国大部分少数民族的语言字符，收录汉字数超过 70000 余个。

它主要采用单字节、双字节、四字节对字符编码，它是向下兼容 GB2312 和 GBK 的，虽然是我国的强制使用标准，但在实际生产中很少用到，用得最多的反而是 GBK 和 GB2312。

（6）Unicode

Unicode 编码设计成了固定两个字节，所有的字符都用 16 位(2^16=65536)表示，包括之前只占 8 位的英文字符等，所以会造成空间的浪费，UNICODE 在很长的一段时间内都没有得到推广应用。

Unicode 完全重新设计，不兼容 iso8859-1，也不兼容任何其他编码。

（7）UTF-8

对于英文字母，unicode 也需要两个字节来表示。所以 unicode 不便于传输和存储。因此而产生了 UTF 编码，UTF-8 全称是（8-bit Unicode Transformation Format）。

UTF 编码兼容 iso8859-1 编码，同时也可以用来表示所有语言的字符，不过，UTF 编码是不定长编码，每一个字符的长度从 1-4 个字节不等。其中，英文字母都是用一个字节表示，而汉字使用三个字节。

注意：

- 一般项目都会使用 UTF-8。unicode 中虽然汉字是两个字节，UTF-8 中汉字是 3 个字节。但是互联网中一个网页也包含了大量的英文字母，这些英文字母只占用 1 个字节，整体占用空间，UTF-8 仍然由于 Unicode。

9.4.5 中文乱码问题

windows 操作系统默认的编码是 GBK，Linux 操作系统默认的编码是 UTF-8。当用 open() 时，调用的是操作系统打开的文件，默认的编码是 GBK。

【示例 9-10】中文字符文件，乱码出现测试

```
#测试写入中文
f = open(r"b.txt","w")
f.write("尚学堂\n 百战程序员\n")
f.close()
```

执行结果如图 9-8 所示：

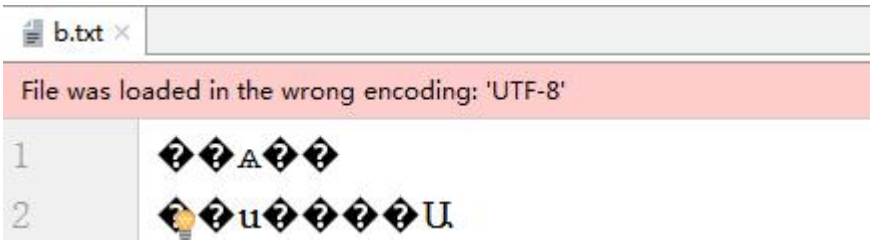


图 9-8 示例 9-10 执行结果

在文件编辑区单击右键，选择 FileEncoding，选择 GBK 即可如图 9-9 所示：

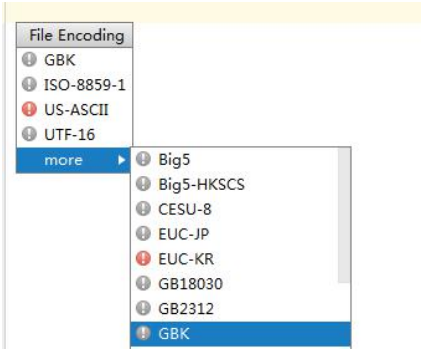


图 9-9 修改编码

再选择 Reload，文件即显示正常。

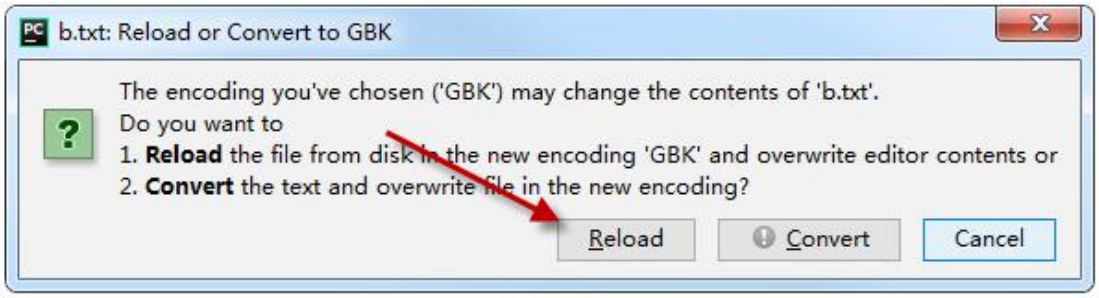


图 9-10 选择 Reload

【示例 9-11】指定文件编码解决中文乱码问题

```
#测试写入中文
f = open(r"b.txt","w",encoding="utf-8")
f.write("尚学堂\n 百战程序员\n")
f.close()
```

执行结果如图 9-11 所示：

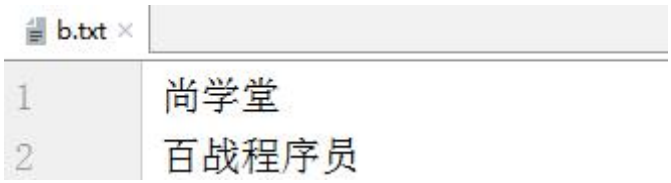


图 9-11 示例 9-11 执行结果

9.4.6 读行和写行

读写一整行是文本文件最常用的操作。尽管可以使用 read 和 write 方法加上行结束符来读写文件中的整行，但比较麻烦。因此，如果要读写一行或多行文本，建议使用 readline 方法、readlines 方法和 writelines 方法。注意，并没有 writeline 方法，写一行文本需要直接使用 write 方法。

readline 方法用于从文件指针当前位置读取一整行文本。也就是说，遇到行结束符停止

读取文本，但读取的内容包括了行结束符。readlines 方法从文件指针当前的位置读取后面的所有数据，并将这些数据按行结束符分隔后，放到列表中返回。writelines 方法需要通过参数指定一个字符串类型的列表，该方法会将列表中的每一个元素值写入文件。换行需要制定换行符 \n。

另外，调用 seek(n)方法，可以重新设置文件指针，也就是改变文件指针的当前位置。使用 write 方法向文件写入内容后，需要调用 seek(0)才能读取刚才写入的内容。

【示例 9-12】使用 writelines()将列表写入文件，readlines()读文件

```
import os
list=['java\n','c\n','c++\n','python\n','javascript\n']
try:
    f=open('testWrite3.txt','w+')
    f.writelines(list)
    #将文件指针移动到文件开始位置
    f.seek(0)
    #读取文件内容
    list=f.readlines()
    for line in list:
        print(line.strip()) #将末尾的'\n'删掉
finally:
    f.close()
```

执行结果如图 9-12 所示：

图 9-12 示例 9-12 执行结果

9.5 StringIO 与 BytesIO

9.5.1 StringIO

很多时候，数据读写不一定是文件，也可以在内存中读写。StringIO 就是在内存中读写 str。要把 str 写入 StringIO，需要先创建一个 StringIO，然后，像文件一样写入即可。

【示例 9-13】StringIO 写 str

```
#StringIO 的使用
from io import StringIO
f=StringIO()
f.write('hello')
f.write(' ')
f.write('python')
print(f.getvalue())
```

执行结果如图 9-13 所示：

图 9-13 示例 9-13 执行结果

getvalue()方法用于获得写入后的 str。要读取 StringIO，可以用一个 str 初始化 StringIO，然后，像读文件一样读取。

【示例 9-14】StringIO 读 str

```
from io import StringIO
f=StringIO('hello\npython')
while True:
    s=f.readline()
    if s=='':
        break
    print(s.strip())
```

执行结果如图 9-14 所示：

图 9-14 示例 9-14 执行结果

9.5.2 BytesIO

StringIO 操作的只能是 str，如果要操作二进制数据，就需要使用 BytesIO。BytesIO 实

现了在内存中读写 bytes，创建一个 BytesIO，然后写入 bytes。

【示例 9-15】BytesIO 写入 bytes

```
#BytesIO 写入
from io import BytesIO
f=BytesIO()
f.write('我喜欢学习 python'.encode('utf-8'))
print(f.getvalue())
```

执行结果如图 9-15 所示：

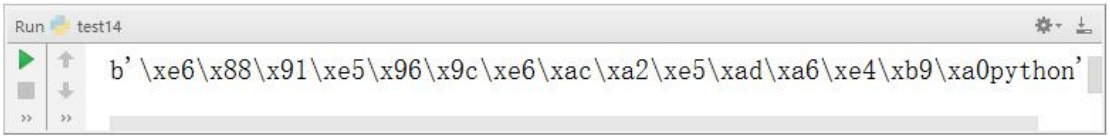


图 9-15 示例 9-15 执行结果

需要注意的是，写入的不是 str，而是经过 UTF-8 编码的 bytes。和 StringIO 类似，可以用一个 bytes 初始化 BytesIO，然后读取 bytes。

【示例 9-16】BytesIO 读取 bytes

```
#BytesIO 读取
from io import BytesIO
f=BytesIO(b'\xe6\x88\x91\xe5\x96\x9c\xe6\xac\xa2\xe5\xad\xa6\xe4\xb9\xa0python')
print(f.read().decode('utf-8'))
```

执行结果如图 9-16 所示：

图 9-16 示例 9-16 执行结果

9.6 操作文件和目录

os 模块以及子模块 path 中包含了大量操作文件和目录的函数。

9.6.1 os 模块-调用操作系统命令

os.system 可以直接调用系统的命令。

【示例 9-17】os.system 调用 windows 系统的记事本程序

```
import os
os.system("notepad.exe")
```

【示例 9-18】os.system 调用 windows 系统中 ping 命令

```
import os
os.system("ping www.baidu.com")
```

执行结果：

```
正在 Ping www.a.shifen.com [111.206.223.206] 具有 32 字节的数据:
来自 111.206.223.206 的回复: 字节=32 时间=9ms TTL=56
来自 111.206.223.206 的回复: 字节=32 时间=7ms TTL=56
来自 111.206.223.206 的回复: 字节=32 时间=6ms TTL=56
来自 111.206.223.206 的回复: 字节=32 时间=9ms TTL=56

111.206.223.206 的 Ping 统计信息:
    数据包: 已发送 = 4，已接收 = 4，丢失 = 0 (0% 丢失)，
往返行程的估计时间(以毫秒为单位):
    最短 = 6ms，最长 = 9ms，平均 = 7ms
```

os.startfile：直接调用可执行文件。

【示例 9-19】运行安装好的微信

```
import os
os.startfile(r"E:\soft\weixin\Tencent\WeChat\WeChat.exe")
```

9.6.2 获取与改变工作目录

getcwd 函数用于获取当前的工作目录，如果指定文件或目录，则在不指定任何路径的情况下，系统会认为该文件或者目录在当前的工作目录中。通过 chdir 函数可以改变当前的工作目录。

【示例 9-20】获取与改变工作目录

```
import os
#获取当前的工作目录，并输出该目录
print('当前工作目录：',os.getcwd())
#获取 path 指定的文件夹包含的文件或文件夹名称列表
print(os.listdir(os.getcwd()))
#改变当前工作目录
os.chdir('../')
print('改变后的工作目录：',os.getcwd())
print(os.listdir(os.getcwd()))
```

执行结果如图 9-17 所示：

图 9-17 示例 9-20 执行结果

9.6.3 文件与目录的操作

在 os 模块中提供了一些操作目录和文件的函数。这些函数的功能描述如下所述：

□ mkdir(dirname,permissions)：创建目录，dirname 表示目录名，如果 dirname 指定的目录名存在，则抛出 OSError 异常。permission 表示目录的权限。在 Linux 或 Mac OS X 上可以设置目录读（r）、写（w）和执行（x）权限。permission 参数值一般是一个八进制的数值。如 0o777 表示最高的权限，7 表示同事具有 rwx 权限。关于 Linux/Mac OS X 权限的细节，请参阅相关的文档，这里不再详细讲解。

□ makedirs(dirname,permissions,exist_ok)：与 mkdir 函数类似，用于创建目录，但 dirname 指定的目录可以是多级的，而且上一级不存在，也会创建上一级的目录。例如，dirname 参数的值是 a/b/c。也就是说需要在当前目录建立三级目录。如果使用 mkdir 函数，并且 a 或者 b 不存在，那么程序会直接抛出 OSError 错误，但使用 makedirs 函数创建目录，会连同 a 和 b 一起创建。makedirs 函数最后一个参数的值为 False，当目录存在时，会抛出 OSError 异常，否则即使目录存在，也不会抛出异常。

□ rmdir(dirname)：删除 dirname 参数指定的目录，如果目录不为空，则抛出一个 OSError 异常。

□ removedirs(dirname)：删除 dirname 参数指定的目录，dirname 参数可以指定多级目录。如 a/b/c，该函数会同时删除 a、b 和 c 目录。不过如果某一级目录不为空，那么该目录以及所有的父目录都不会被删除。例如，如果在 b 目录中除了有一个 c 子目录外，还有一个 test.txt 文件，那么在删除 a/b/c 目录时，只会删除 b 目录中的 c 子目录，a 和 b 目录都不会被删除。

□ remove(filename)：删除 filename 参数指定的文件。

□ rename(src,dst)：将 src 参数指定的文件(目录)名改成 dst 参数指定的文件名。

□ renames(src,dst)：与 rename 函数的功能类似，只是 src 和 dst 可以是多级目录（最后一级可以是文件名）。该函数将每一级的目录都修改为对应的目录名，如可以将 a/b/c 修改为 aa/bb/cc。该函数将 a 改成 aa，b 改成 bb，c 改成 cc。

【示例 9-21】mkdir 函数的使用

```
import os
#使用 mkdir 创建目录
#判断当前目录下是否存在 newfile 目录
if not os.path.exists('newfile'):
    os.mkdir('newfile')
#mkdir 函数不能创建多级目录，会抛出异常
os.mkdir('x/y/z')
```

mkdir 函数不能创建多级目录，会抛出异常，执行结果如图 9-18 所示：

图 9-18 示例 9-21 执行结果

【示例 9-22】mkdirs 函数的使用

```
import os
#判断当前目录下是否存在 newfile2 目录
if not os.path.exists('newfile2'):
    os.makedirs('newfile2')
#创建多级目录应该使用 makedirs 函数
os.makedirs('x/y/z')
```

【示例 9-23】rmdir 函数的使用

```
import os
#rmdir 函数删除目录，如果目录不为空，则抛出 OSError 异常
try:
    os.rmdir('newfile2')
except OSError as e:
    print(e)
```

因为 newfile2 目录不为空，会抛出异常，执行结果如图 9-19 所示：

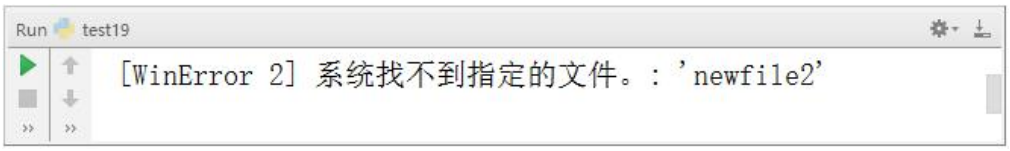


图 9-19 示例 9-23 执行结果

【示例 9-24】removedirs 函数的使用

```
import os
#使用 removedirs 函数删除多级目录
os.removedirs('x/y/z')
```

【示例 9-25】rename、renames、remove 函数的使用

```
import os
#rename 函数的使用
if not os.path.exists('newfile3'):
    os.makedirs('newfile3')
    #将 newfile3 目录重命名为 newfile4
    os.rename('newfile3','newfile4')
if  os.path.exists('testFile.txt'):
    os.rename('testFile.txt','testFileNew.txt')
#renames 函数的使用
if not os.path.exists('a/b/c'):
    os.makedirs('a/b/c')
    os.renames('a/b/c','aa/bb/cc')
if os.path.exists('newfile/sss.txt'):
    os.rename('newfile/sss.txt','newfile/aaa.txt')
```

```
    #使用 remove 函数删除目录下的文件
    os.remove('newfile/aaa.txt')
```

9.7 pickles 序列化

python 程序运行中得到了一些字符串，列表，字典等数据，想要长久的保存下来，方便以后使用，而不是简单的放入内存中关机断电就丢失数据。python 模块中 pickle 模块可以将对象转换为一种可以传输或存储的格式。

pickle 是 python 语言的一个标准模块，安装 python 后已包含 pickle 库，不需要单独再安装。

pickle 模块实现了基本的数据序列化和反序列化。通过 pickle 模块的序列化操作能够将程序中运行的对象信息保存到文件中去，永久存储；通过 pickle 模块的反序列化操作，能够从文件中创建上一次程序保存的对象。常用函数如表 9-4 所示：

表 9-4 pickle 模块中常用的函数

方法	描述
pickle.dump(obj, file, [,protocol])	将要持久化的数据“对象”，保存到“文件”中
pickle.load(file)	从“文件”中读取字符串，将他们反序列化转换为 python 的数据对象
pickle.dumps(obj[, protocol])	将 obj 对象序列化为 string 形式，而不是存入文件中
pickle.loads(string)	从 string 中读出序列化前的 obj 对象

【示例 9-26】对象序列化并写入文件，从文件反序列化获取对象

```
import pickle
a = {" name ": "Tom", "age": "40"}
#将字典序列化并写入文件
with open('text.txt', 'wb') as file:
    pickle.dump(a, file)
#从文件中反序列化出对象
with open('text.txt','rb') as file2:
     b = pickle.load(file2)
print(type(b))
```

```
print(b)
```

执行结果如图 9-20 所示：

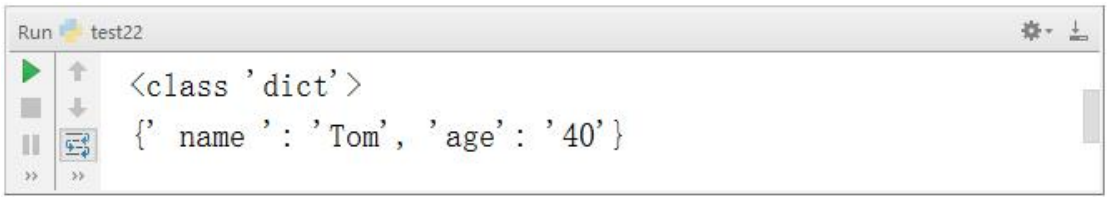


图 9-20 示例 9-26 执行结果

【示例 9-27】将 obj 对象序列化为 string，从 string 中读出序列化前的 obj 对象

```
import pickle
info = [1, 2, 3,'hello','python']
print('原始数据：',info)
#将对象序列化为 string
data1 = pickle.dumps(info)
#从 string 中读出序列化前的对象
data2 = pickle.loads(data1)
print("序列化：%r" % data1)
print("反序列化：  %r" % data2)
```

执行结果如图 9-21 所示：

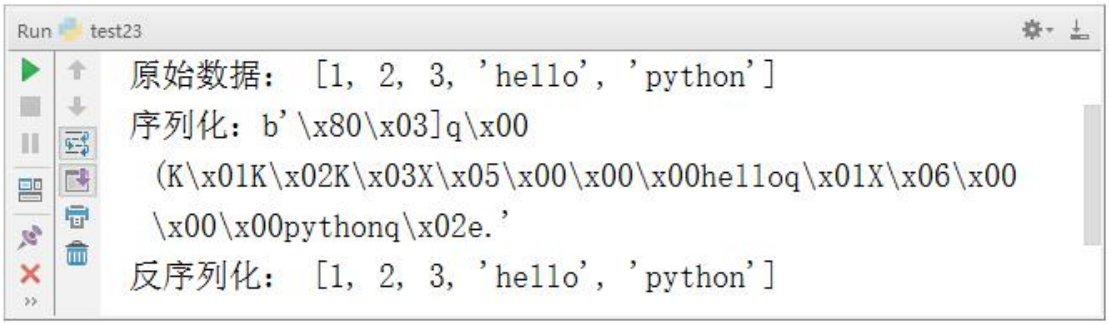


图 9-21 示例 9-27 执行结果

9.8 处理 JSON 格式的数据

Pickle 的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于 Python，并且可能不同版本的 Python 彼此都不兼容，因此，如果要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如 XML，但更好的方法是序列化为 JSON，因为 JSON 表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON 不仅是标准格式，并且比 XML 更快，而且可以直接在 Web 页面中读取，非常方便。

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它基于 ECMAScript

的一个子集。 JSON 采用完全独立于语言的文本格式，但是也使用了类似于 C 语言家族的习惯(包括 C、C++、Java、JavaScript、Perl、Python 等)。这些特性使 JSON 成为理想的数据交换语言。易于人阅读和编写，同时也易于机器解析和生成(一般用于提升网络传输速率)。

JSON 格式的数据可以保存数组和对象，JSON 数组用一对中括号将数据括起来，JSON 对象用一对大括号将数据括起来。下面就是一个典型的 JSON 格式的字符串。在这个 JSON 格式字符串中定义了一个有三个元素的数组，每一个元素的类型都是一个对象。对象的 key 和 value 之间要用冒号（:）分隔，key-value 对之间用逗号（,）分隔。注意，key 和字符串类型的值要用双引号括起来，不能使用单引号。

```
[
    {"name":"zs","age":23,"sex":"man"},
    {"name":"ls","age":22,"sex":"woman"},
    {"name":"ww","age":21,"sex":"man"}
]
```

9.8.1JSON 字符串与字典互相转换

将字典转换为 JSON 字符串需要使用 json 模块的 dumps 函数，该函数需要将字典通过参数传入，然后返回与字典对应的 JSON 字符串。将 JSON 字符串转换为字典可以使用下面两种方法。

（1） 使用 json 模块的 loads 函数，该函数通过参数传入 JSON 字符串，然后返回与该 JSON 字符串对应的字典。

（2） 使用 eval 函数将 JSON 格式字符串当作普通的 Python 代码执行，eval 函数会直接返回与 JSON 格式字符串对应的字典。

【示例 9-28】JSON 字符串与字典互相转换

```
import json
d={"name":"zs","age":23,"sex":"man"}
#将字典转换为 json 字符串
jsonString=json.dumps(d)
#输出 jsonString 变量的类型
print(type(jsonString))
#输出 JSON 字符串
print(jsonString)
print('使用 loads 函数将 JSON 字符串转换为字典')
#使用 loads 函数将 JSON 字符串转换为字典
d=json.loads(jsonString)
```

```
print(type(d))
#输出字典
print(d)
print('eval 函数使用 eval 函数将 JSON 字符串转换为字典')
#使用 eval 函数将 JSON 字符串转换为字典
dd=eval(jsonString)
print(type(dd))
print(dd)
```

执行结果如图 9-22 所示：

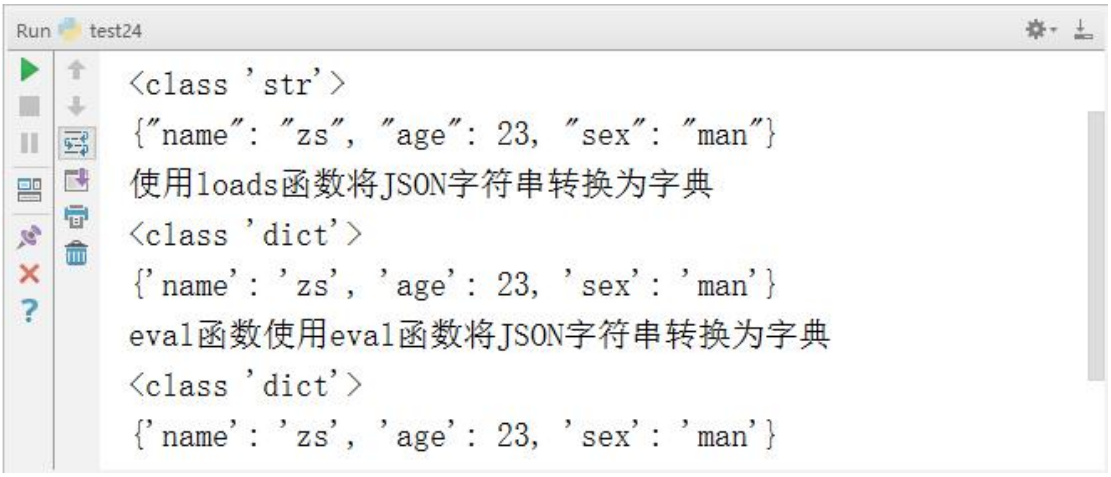


图 9-22 示例 9-28 执行结果

尽管 eval 函数与 loads 函数都可以将 JSON 字符串转换为字典，但建议使用 loads 函数进行转换，因为 eval 函数可以执行任何 Python 代码，如果 JSON 字符串中包含了有害的 Python 代码，执行 JSON 字符串可能会带来风险。

9.8.2 将类实例转换为 JSON 字符串

dumps 函数不仅可以将字典转换为 JSON 字符串，还可以将类实例转换为 JSON 字符串。dumps 函数需要通过 default 关键字参数指定一个回调函数，在转换的过程中，dumps 函数会向这个回调函数传入类实例(通过 dumps 函数第 1 个参数传入)，而回调函数的任务是将传入的对象转换为字典，然后 dumps 函数再将由回调函数返回的字典转换为 JSON 字符串。也就是说，dumps 函数的本质还是将字典转换为 JSON 字符串，只是如果将类实例也转换为 JSON 字符串，需要先将类实例转换为字典，然后再将字典转换为 JSON 字符串，而将类实例转换为字典的任务就是通过 default 关键字参数指定的回调函数完成的。

【示例 9-29】类实例转换为 JSON 字符串

```
import json
```

```
class Student(object):
    def __init__(self,name,age,score):
        self.name=name
        self.age=age
        self.score=score
#定义类实例转换为字典的方法
def student_dict(stu):
    return {"name":stu.name,"age":stu.age,"score":stu.score}
#创建实例对象
stu=Student('张三',23,98)
#将 Student 类的实例转换为 JSON 字符串，ensure_ascii 关键字参数的值设置为 False，
#可以让返回的 JSON 字符串正常显示中文
jsonStr=json.dumps(stu,default=student_dict,ensure_ascii=False)
print(jsonStr)
```

执行结果如图 9-23 所示：

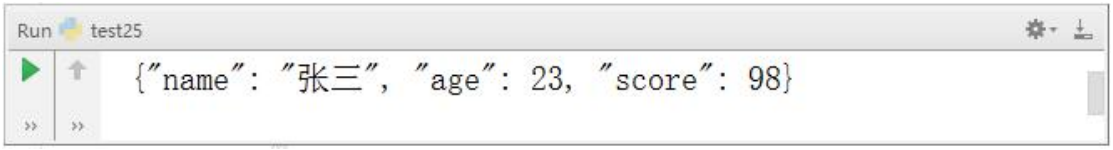


图 9-23 示例 9-29 执行结果

9.8.3 将 JSON 字符串转换为类实例

loads 函数不仅可以将 JSON 字符串转换为字典，还可以将 JSON 字符串转换为类实例。转换原理是通过 loads 函数的 object_hook 关键字参数指定一个类或一个回调函数。具体处理方式如下：

（1） 指定类：loads 函数会自动创建指定类的实例，并将由 JSON 字符串转换成的字典通过类的构造方法传入类实例，也就是说，指定的类必须有一个可以接收字典的构造方法。

（2） 指定回调函数：loads 函数会调用回调函数返回类实例，并将由 JSON 字符串转换成的字典传入回调函数，也就是说，回调函数也必须有一个参数可以接收字典。

从前面的描述可以看出，不管指定的是类还是回调函数，都会由 loads 函数传入由 JSON 字符串转换成的字典，也就是说，loads 函数将 JSON 字符串转换为类实例的本质是先将 JSON 字符串转换为字典，然后再将字典转换为对象。区别是指定类时，创建类实例的任务由 1oads 函数完成，而指定回函数时，创建类实例的任务需要在回调函数中完成，前者更方便，后者更灵活。

【示例 9-30】JSON 字符串转换为类实例

```
import json
class Student(object):
    def __init__(self,name,age,score):
        self.name=name
        self.age=age
        self.score=score
#定义类实例转换为字典的方法
def student_dict(stu):
    return {"name":stu.name,"age":stu.age,"score":stu.score}
#创建实例对象
stu=Student('张三',23,98)
#将 Student 类的实例转换为 JSON 字符串，ensure_ascii 关键字参数的值设置为 False，
#可以让返回的 JSON 字符串正常显示中文
jsonStr=json.dumps(stu,default=student_dict,ensure_ascii=False)
print(jsonStr)
```

执行结果如图 9-24 所示：

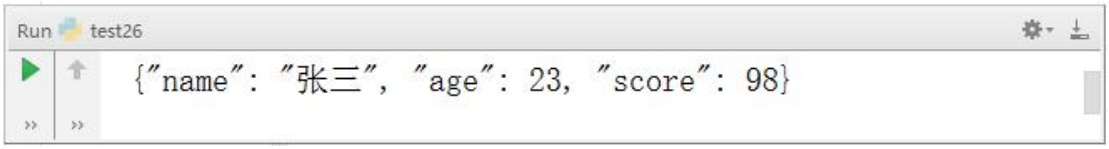


图 9-24 示例 9-30 执行结果

9.8.4 类实例列表与 JSON 字符串互相转换

前面讲的类实例和 JSON 字符串直接地互相转换，转换的只是单个对象，如果 JSON 字符串是一个类实例数组或一个类实例的列表，也可以互相转换。

【示例 9-31】类实例列表与 JSON 字符串互相转换

```
import json
class Student:
    def __init__(self,name,age,score):
        self.name=name
        self.age=age
        self.score=score
    def __str__(self):
        return '姓名是：{0},年龄是：{1},成绩是:{2}'.format(self.name,self.age,self.score)
```

```
#定义类实例转换为字典的方法
def student_dict(stu):
    return {"name":stu.name,"age":stu.age,"score":stu.score}
#定义将字典转换为实例对象的函数
def jsonToStudent(dic):
    return Student(dic['name'],dic['age'],dic['score'])
#创建实例类列表
stu1=Student("张三",23,98)
stu2=Student("李四",20,99)
stu3=Student("王五",22,88)
stuList=[stu1,stu2,stu3]
#将 Student 对象列表转换为 JSON 字符串
stuString=json.dumps(stuList,default=student_dict,ensure_ascii=False)
#将 JSON 字符串转换为 Student 对象列表
stuList=json.loads(stuString,object_hook=jsonToStudent)
for stu in stuList:
    print(stu)
```

执行结果如图 9-25 所示：

图 9-25 示例 9-31 执行结果

9.9 CSV 文件操作

9.9.1 csv.reader 对象从 csv 文件读取

【示例 9-32】csv.reader 对象从 csv 文件读取

```
import csv
with    open(r"d:\a.csv") as a:
    a_csv = csv.reader(a)            #创建 csv 对象,它是一个包含所有数据的列表，每一行为一个元素
```

```
    headers = next(a_csv)            #获得列表对象，包含标题行的信息
    print(headers)
    for row in a_csv:                #循环打印各行内容
        print(row)
```

执行结果如图 9-26 所示：：

图 9-26 示例 9-32 执行结果

9.9.2 csv.writer 对象写入

【示例 9-33】csv.writer 对象写入

```
import csv
headers = ["工号","姓名","年龄","地址","月薪"]
rows = [("1001","高淇",18,"西三旗 1 号院","50000"),("1002","高八",19,"西三旗 1 号院","30000")]

with    open(r"d:\b.csv","w",newline='') as b:
    b_csv = csv.writer(b)            #创建 csv 对象
    b_csv.writerow(headers)          #写入一行（标题）
    b_csv.writerows(rows)            #写入多行（数据）
```

执行结果在 D 盘生成 b.csv 文件，文件内容如图 9-27 所示：

	工号	姓名	年龄	地址	月薪
2	1001	高淇	18	西三旗1号	50000
3	1002	高八	19	西三旗1号	30000

图 9-27 示例 9-33 执行结果

9.10 shutil 模块(拷贝和压缩)

shutil 模块是 python 标准库中提供的，主要用来做文件和文件夹的拷贝、移动、删除等；还可以做文件和文件夹的压缩、解压缩操作。

os 模块提供了对目录或文件的一般操作。shutil 模块作为补充，提供了移动、复制、压缩、解压等操作，这些 os 模块都没有提供。

【示例 9-34】shutil 实现文件的拷贝

```
import shutil
#copy 文件内容
shutil.copyfile("1.txt","1_copy.txt")
```

【示例 9-35】shutil 实现递归拷贝文件夹内容

```
import shutil
#"音乐"文件夹不存在才能用。
shutil.copytree("电影/学习","音乐",ignore=shutil.ignore_patterns("*.html","*.htm"))
```

上述示例将文件夹“电影/学习”下面的内容拷贝到文件夹“音乐”下。拷贝时忽略所有的 html 和 htm 文件。运行结果如图 9-28 所示：

图 9-28 示例 9-35 执行结果

【示例 9-36】shutil 实现将文件夹所有内容压缩

```
import shutil
import zipfile
#将"电影/学习"文件夹下所有内容压缩到"音乐 2"文件夹下生成 movie.zip
#shutil.make_archive("音乐 2/movie","zip","电影/学习")


#压缩:将指定的多个文件压缩到一个 zip 文件
#z = zipfile.ZipFile("a.zip","w")
#z.write("1.txt")
#z.write("2.txt")
#z.close()
```

【示例 9-37】shutil 实现将压缩包解压缩到指定文件夹

```
import shutil
import zipfile
#解压缩：
z2 = zipfile.ZipFile("a.zip","r")
z2.extractall("d:/")    #设置解压的地址
z2.close()
```

习题

一、选择题

1. 关于 Python 对文件的处理，以下选项中描述错误的是（）

A. Python 通过解释器内置的 open() 函数打开一个文件

B. 当文件以文本方式打开时，读写按照字节流方式

C. 文件使用结束后要用 close() 方法关闭，释放文件的使用授权

D. Python 能够以文本和二进制两种方式处理文件

2. 以下选项中，对文件的描述错误的是

A. 文件中可以包含任何数据内容

B. 文本文件和二进制文件都是文件

C. 文本文件不能用二进制文件方式读入

D. 文件是一个存储在辅助存储器上的数据序列

3. 以下选项中，不是 Python 对文件的读操作方法的是（）

A. readline　B. readall　C. readtext　D. read

4. 关于 Python 文件处理，以下选项中描述错误的是（）

A. Python 能处理 JPG 图像文件

B. Python 不可以处理 PDF 文件

C. Python 能处理 CSV 文件

D. Python 能处理 Excel 文件

5. 以下选项中，不是 Python 对文件的打开模式的是

A. ‘w’　B. ‘+’ C. ‘c’ D. ‘r’

二、简答题

1. 描述 Python 文件打开模式。
2. 请描述 unicode，utf-8,gbk 等编码之间的关系。
3. 如何使用 python 删除一个文件。
4. 简述 read. readline. readlines 的区别。
5. 简述文件中的 mkdir 跟 mkdirs，有什么区别?

三、编码题

1. 编写一个 Python 程序，把一个文件中的内容拷贝到另一个文件中。
2. 编写一个 Python 程序，使用 pickle 模块对 data= [1,'a',2,'b',3,'c']数据进行序列化和反序列化。
3. 编写一个 Python 程序，举例实现 JSON 字符串与字典互相转换。
4. 编写一个 Python 程序，使用 shutil 模块完成对文件的拷贝和压缩。

附录 1：Python3.8 新特性

目前 Python 3.8 是 Python 语言的最新版本，它适合用于编写脚本、自动化以及机器学习和 Web 开发等各种任务。现在 Python 3.8 已经进入官方的 beta 阶段，这个版本带来了许多语法改变、内存共享、更有效的序列化和反序列化、改进的字典和更多新功能。

Python 3.8 还引入了许多性能改进。总的来说，我们即将拥有一个更快、更精确、更一致和更现代的 Python。下面是 Python 3.8 的新功能和最重要的改变。

1）赋值表达式

Python 3.8 最明显的变化就是赋值表达式，即:=操作符。赋值表达式可以将一个值赋给一个变量，即使变量不存在也可以。它可以用在表达式中，无需作为单独的语句出现。

【示例 1-1】赋值表达式

```
#原来实现，需要两个语句
flag = False
print(flag)

#Python3.8 中，可以使用 flag 运算符将上面两个语句合并为一句
print(flag := False)
```

执行结果如图 1-1 所示:

图 1-1 示例 1-1 执行效果图

从上述示例可以看到，赋值表达式可以把 True 分配给 flag，并直接 print 这个值。一定要有（:= ），不然表达式也是无法正常执行的，有了新的赋值表达式符号，不仅在构造上更简便，有时也可以更清楚的传达代码意图。

【示例 1-2】在 while 循环中，使用赋值表达式（:= ）

```
#原来实现方式
inputs = list()
current = input("请输入: ")
while current != "quit":
    inputs.append(current)
    current = input("请输入: ")
print('原来实现方式执行结果：',inputs)
```

```
#使用赋值表达式，进一步简化这段循环：
inputs = list()
while (current := input('请输入：')) != 'quit':
    inputs.append(current)
print('使用赋值表达式执行结果：',inputs)
```

执行结果如图 1-2 所示:

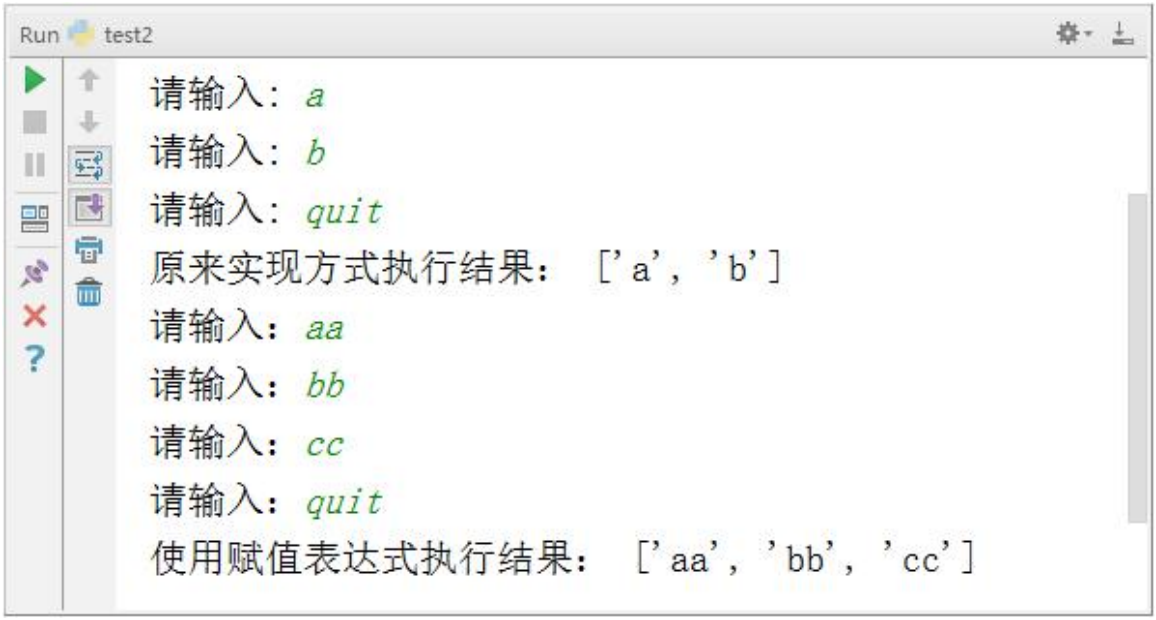


图 1-2 示例 1-2 执行效果图

从上述示例可以看到，使用赋值表达式，可以再进一步简化代码，但是代码可读性确变差了，所以，大家要使用赋值表达式的方法还需要结合自身进行判断。

注意：

□ PEP572 中描述了复制表达式的所有细节，大家可以深入阅读。

2） 仅通过位置指定的参数

仅通过位置指定的参数是函数定义中的一个新语法，可以让程序员强迫某个参数只能通过位置来指定。这样可以解决 Python 函数定义中哪个参数是位置参数、哪个参数是关键字参数的模糊性。

仅通过位置指定的参数可以用于如下情况：某个函数接受任意关键字参数，但也能接受一个或多个未知参数。Python 的内置函数通常都是这种情况，所以允许程序员这样做，能增强 Python 语言的一致性。

【示例 1-3】仅通过位置指定的参数

```
def pow(x, y, z=None, /):
    r = x**y
    if z is not None:
        r %= z
    return r
print('调用 pow(2,3)执行结果：',pow(2,3))
print('调用 pow(2,3,4)执行结果：',pow(2,3,4))
print('调用 pow(2,3,z=4)执行结果：',pow(2,3,z=4))
```

执行结果如图 1-3 所示:

图 1-3 示例 1-3 执行效果图

符号 / 分隔了位置参数和关键字参数。在上述示例中，所有参数都是未知参数。在以前版本的 Python 中，z 会被认为是关键字参数。但采用上述函数调用 pow(2, 3)和 pow(2, 3, 4)都是正确的调用方式，而 pow(2, 2, z=4)是不正确的。

3） 支持 f 字符串调试

在 Python 3.6 版本中增加了 f-strings，可以使用 f 前缀更方便地格式化字符串，同时还能进行计算，示例如下：

【示例 1-4】使用 f 前缀格式化字符串，并进行计算

```
a = 20
print(f'a 的值:{a}')
print(f'a 的值:{a+1}')
```

执行结果如图 1-4 所示:

图 1-4 示例 1-4 执行效果图

在 3.8 中只需要增加一个 = 符号，即可在同一个表达式内进行输出文本和值或变量的计算，而且效率更高。

【示例 1-5】在 3.8 中前缀格式化字符串 f 的使用

```
print(f'{a=}')
print(f'{a+1=}')
```

执行结果如图 1-5 所示:

图 1-5 示例 1-5 执行效果图

4） 多进程共享内存

在旧版本的 Python 中，进程间共享数据只能通过写入文件、通过网络套接字发送，或采用 Python 的 pickle 模块进行序列化等方式。

在 Python 3.8 中，multiprocessing 模块提供了 SharedMemory 类，可以在不同的 Python 进城之间创建共享的内存区域。

共享内存提供了进程间传递数据的更快的方式，从而使得 Python 的多处理器和多内核编程更有效率。

共享内存片段可以作为单纯的字节区域来分配，也可以作为不可修改的类似于列表的对象来分配，其中能保存数字类型、字符串、字节对象、None 对象等一小部分 Python 对象。

【示例 1-6】多进程共享内存

```
from multiprocessing import Process
from multiprocessing import shared_memory
share_nums = shared_memory.ShareableList(range(5))
def do_work1(nums):
    for i in range(5):
        nums[i] += 10
    print('do_work1 nums = %s'% nums)

def do_work2(nums):
    print('do_work2 nums = %s'% nums)

if __name__ == '__main__':
    p1 = Process(target=do_work1, args=(share_nums, ))
```

```
    p1.start()
    p1.join()
    p2 = Process(target=do_work2, args=(share_nums, ))
    p2.start()
```

执行结果如图 1-6 所示:

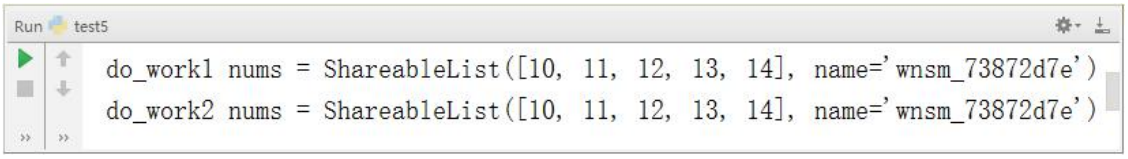


图 1-6 示例 1-6 执行效果图

从上述示例可以看到，do_work1 与 do_work2 虽然运行在两个进程中，但都可以访问和修改同一个 ShareableList 对象。

5） Asyncio 异步交互模式

asyncio.run() 已经从暂定状态晋级为稳定 API。 此函数可被用于执行一个 coroutine（协程对象）并返回结果，同时自动管理事件循环。

【示例 1-7】asyncio.run()执行一个协程对象，并返回结果，同时自动管理事件循环

```
import asyncio
async def main():
    await asyncio.sleep(0)
    return 42
asyncio.run(main())
```

【示例 1-8】Asyncio 异步交互，原来实现

```
import asyncio
async def main():
    await asyncio.sleep(0)
    return 42
loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
try:
    loop.run_until_complete(main())
finally:
    asyncio.set_event_loop(None)
    loop.close()
```

6） 新增和改进的数学和统计功能

Python 3.8 对现有的标准库软件包和模块进行了许多改进。标准库中的数学有了一些新功能。math.prod（）与内置 sum（）类似，但对于乘法乘积。

【示例 1-9】math.prod()的使用

```
import math
result = math.prod((2,3,4,5))
print('math.prod((2,3,4,5))执行结果：',result)
```

执行结果如图 1-7 所示:

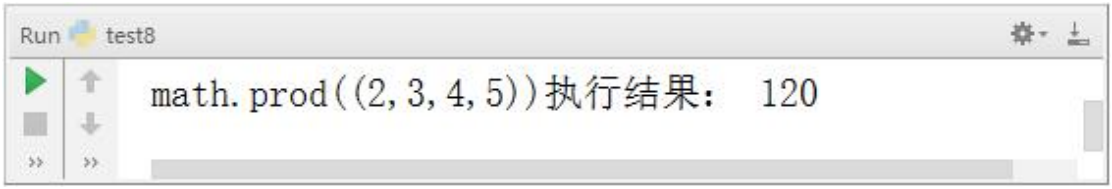


图 1-7 示例 1-9 执行效果图

使用 isqrt（）来找到平方根的整数部分。

【示例 1-10】math.isqrt()的使用

```
import math
result = math.isqrt(25)
print('math.isqrt(25)执行结果：',result)
result = math.isqrt(15)
print('math.isqrt(15)执行结果：',result)
```

执行结果如图 1-8 所示:

图 1-8 示例 1-10 执行效果图

25 的平方根是 5。你可以看到 isqrt（）返回整数结果，而 math.sqrt（）始终返回浮点数。15 的平方根约等于 3.9。isqrt（）将答案截断为一个整数。

表 1-1 统计模块新增功能

函数名称	说明
statistics.fmean（）	计算浮点数的平均值
statistics.geometric_mean（）	计算浮点数的几何平均值
statistics.multimode（）	查找序列中最频繁出现的值
statistics.quantiles（）	计算用于将数据等概率分为 n 个连续区间的切点

【示例 1-11】统计模块新增功能的使用

```
import statistics
data = [11, 2, 33, 14, 2, 26, 33, 38,9]
print('浮点数的平均值：',statistics.fmean(data))
print('浮点数的几何平均值:',statistics.geometric_mean(data))
print('查找序列中最频繁出现的值:',statistics.multimode(data))
print('数据等概率分为 n 个连续区间的切点:',statistics.quantiles(data, n=4))
```

执行结果如图 1-9 所示:

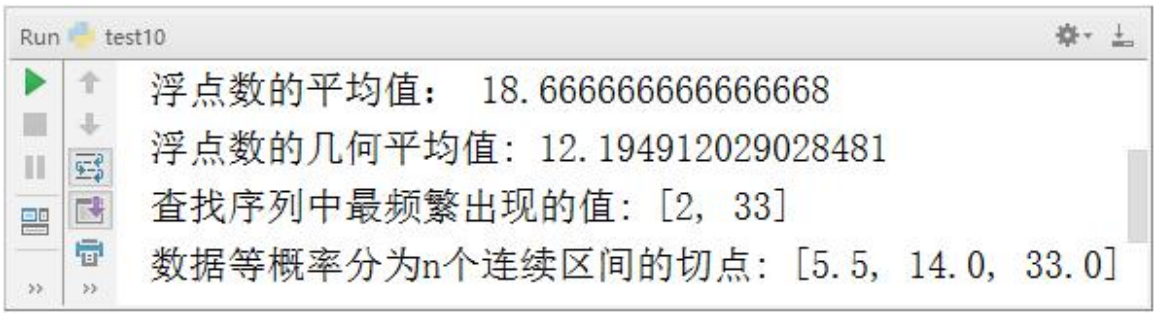


图 1-9 示例 1-11 执行效果图

7） 新增危险语法警告功能

Python 有一个 SyntaxWarning 功能，可以警告不是 SyntaxError 的可疑语法。Python 3.8 添加了一些新功能，可以在编码和调试过程中为你提供帮助。

is 和==之间的区别可能会造成混淆。后者用于检查是否有相等的值，而只有在对象相同时才为 true。Python 3.8 将在应该使用==而不是 is 时发出警告。

【示例 1-12】危险语法警告功能

```
version = "3.8"
print(version == "3.8")
print(version is "3.8")
```

执行结果如图 1-10 所示:

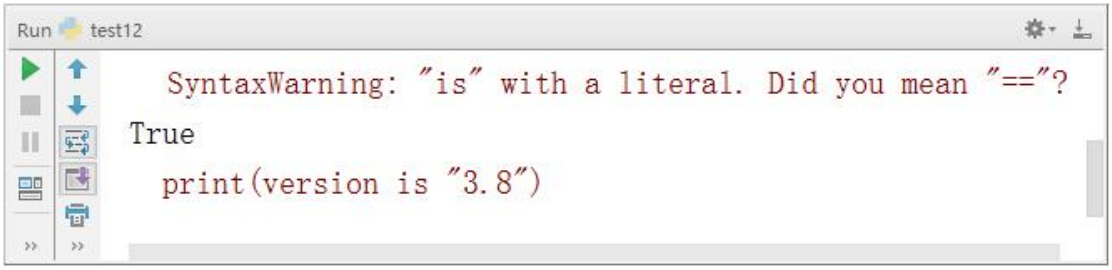


图 1-10 示例 1-12 执行效果图

写长列表时，尤其是垂直格式化时，很容易漏掉逗号。当忘记元组列表中的逗号时会发出让你不解的不可调用元组错误消息。Python 3.8 不仅会发出警告，还会指出实际问题。

【示例 1-13】危险语法警告功能

```
a = [
  (1, 3)
  (2, 4)
]
print(a)
```

执行结果如图 1-11 所示:

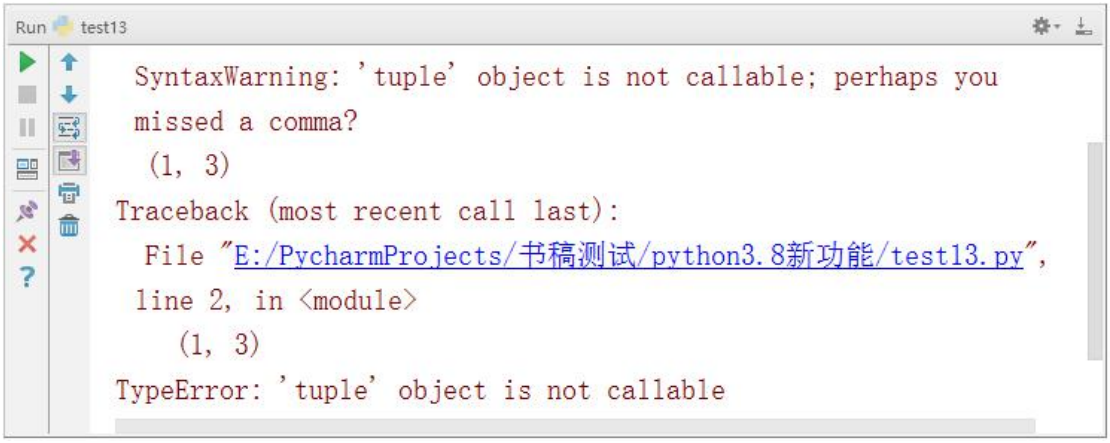


图 1-11 示例 1-13 执行效果图

该警告正确地将丢失的逗号标识为真正的罪魁祸首。

8） 优化

Python 3.8 进行了一些优化，有的让代码运行得更快，有的优化减少了内存占用。示例如下：

【示例 1-14】Python3.7 环境

```
import collections
from timeit import timeit
import sys
Person = collections.namedtuple("Person", "name twitter")
raymond = Person("Raymond", "@raymondh")
# Python 3.7
print('Python3.7 环境下执行时间：')
print(timeit("raymond.twitter", globals=globals()))
print('Python3.7 环境下所占内存：')
print(sys.getsizeof(list(range(20191014))))
```

执行结果如图 1-12 所示:

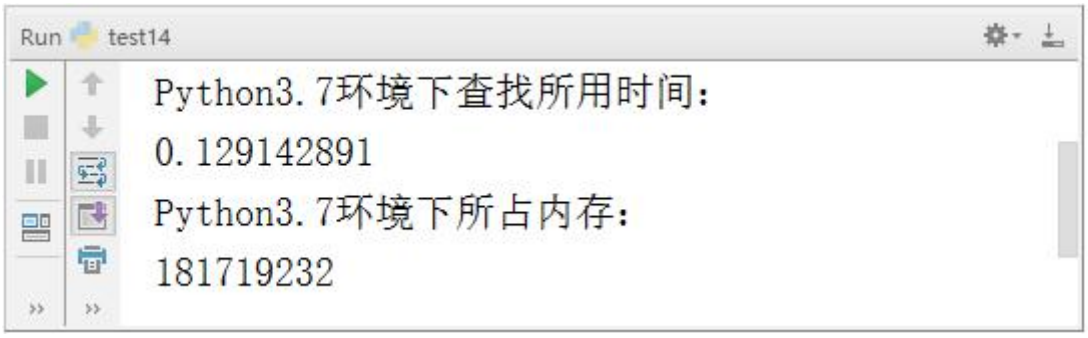


图 1-12 示例 1-14 执行效果图

【示例 1-15】Python3.8 环境

```
import collections
from timeit import timeit
import sys
Person = collections.namedtuple("Person", "name twitter")
raymond = Person("Raymond", "@raymondh")
# Python 3.8
print('Python3.8 环境下执行时间：')
print(timeit("raymond.twitter", globals=globals()) )
print('Python3.8 环境下所占内存：')
print(sys.getsizeof(list(range(20191014))))
```

执行结果如图 1-13 所示:

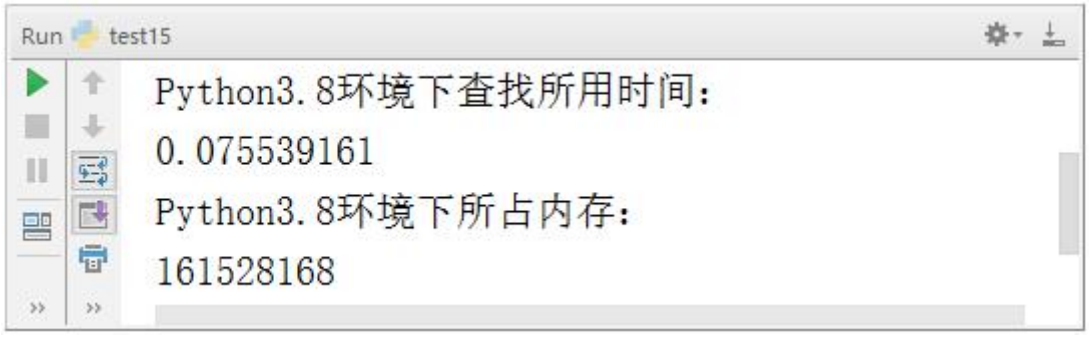


图 1-13 示例 1-15 执行效果图

从示例 14 和示例 15 可以看到，与 Python 3.7 相比，在 Python 3.8 中查找命名元组中的字段要快得多，而且对已知长度的可迭代对象初始化列表时，可以节省一些空间。

附录 2：Python400 集教学视频目录

Python400 集教学视频介绍

《Python400 集》由高淇、张巧英老师历经两年多时间录制而成。整个教学视频以较轻快的风格，从零基础开始讲解。不仅讲解 Python 的基础知识和初步应用，同时注重对底层原理的讲解（内存分析、数据结构、丰富强大的类库）。让大家学习完本视频之后，很好地掌握 Python 语言，同时可以使用 Python 语言进行实际项目的开发。

整套视频以理论与实际相结合，为每个知识点都设计了对应的示例，让 Python 的初学者能够既快速又深刻的理解这些知识点。并且已经被北京大学教授推荐为学习 Python 必看视频。

全套视频分为四季：

- 第一季 【基础篇】Python 基础 （1～115 集）
- 第二季 【提高篇】Python 深入和扩展（116～248 集）
- 第三季 【扩展篇】网络编程、多线程、扩展库（249～278 集）
- 第四季 【高手篇】算法、函数式编程、基于 TensorFlow 人工智能实战（279～402 集）

Python400 集教学视频内容如下：

第一季 1.1 Python 入门

顺序	名称	时长
课时 1	Python 介绍_特性_版本问题_应用范围	17:00
课时 2	Python 下载_安装_配置_第一个 Python 程序	8:04
课时 3	环境介绍_交互模式的使用_IDLE 介绍和使用	8:59
课时 4	IDLE 开发环境的使用_建立 Python 源文件	7:18
课时 5	Python 程序格式_缩进_行注释_段注释	8:18
课时 6	简单错误如何处理_守破离学习法_程序员修炼手册	11:53
课时 7	海归绘图_坐标系问题_画笔各种方法	8:26
课时 8	海龟绘图_画出奥运五环	12:58

第一季 1.2 内置数据类型

顺序	名称	时长
课时 9	程序的构成	7：25
课时 10	对象的基本组成和内存示意图	13:57
课时 11	引用的本质_栈内存和堆内存_内存示意图	4:01
课时 12	标识符_帮助系统的简单实用_命名规则	14:03
课时 13	变量声明_初始化_删除变量_垃圾回收机制	4:25

顺序	名称	时长
课时 14	链式赋值_系列解包赋值_常量	5:13
课时 15	内置数据类型_基本算术运算符	5:55
课时 16	整数_不同进制_其他类型转换成整数	9:59
课时 17	浮点数_自动转换_强制转换_增强赋值运算符	8:39
课时 18	时间表示_unix 时间点_毫秒_微妙_time 模块	8:39
课时 19	多点坐标_绘出折线图_计算两点距离	8:03
课时 20	布尔值_比较运算符_逻辑运算符_短路问题	7:07
课时 21	同一运算符_整数缓存问题	9:56
课时 22	字符串_unicode 字符集_三种创建字符串方式 len()	11:38
课时 23	字符串_转义字符_字符串拼接_字符串复制_input()获得键盘输入	10:17
课时 24	字符串_str()提取字符_replace()替换_内存分析	10:41
课时 25	字符串_切片 slice 操作_逆序	9:49
课时 26	字符串_split()分割_join()合并_join 效率测试	16:08
课时 27	字符串_驻留机制_内存分析_字符串同一判断_值相等判断	7:00
课时 28	字符串_常用查找方法_去除首位信息_大小写转换_排版	12:21
课时 29	字符串_format 格式化_数字格式化操作	13:51
课时 30	可变字符串_io.StringIO	3:34
课时 31	运算符总结_位操作符_优先级问题	12:14

第一季 1.3 序列

顺序	名称	时长
课时 32	列表_特点_内存分析	9:34
课时 33	创建列表的 4 种方式_推导式创建列表	12:03
课时 34	列表_元素的 5 种添加方式_效率问题	10:40
课时 35	列表_元素删除的三种方式_删除本质是数组元素拷贝	10:27
课时 36	列表_元素的访问_元素出现次数统计_成员资格判断	5:43
课时 37	列表_切片 slice 操作	9:45
课时 38	列表_排序_revered 逆序_max_min_sum	7:33
课时 39	列表_二维列表_表格数据的存储和读取	11:53
课时 40	元组_特点_创建的两种方式_tuple()要点	8:25
课时 41	元组_元素访问_计数方法_切片操作_成员资格判断_zip()	5:34
课时 42	元组_生成器推导式创建元组_总结	7:59
课时 43	字典_特点_4 种创建方式_普通_dict_zip_formkeys	10:46

顺序	名称	时长
课时 44	字典_元素的访问_键的访问_值的访问_键值对的访问	5:10
课时 45	字典_元素的添加_修改_删除	6:04
课时 46	字典_序列解包用于列表元组的字典	3:33
课时 47	字典_复杂表格数据存储_列表和字典综合嵌套	10:24
课时 48	字典_核心底层原理_内存分析_存储键值对过程	11:23
课时 49	字典_核心底层原理_内存分析_查找值对象过程	6:22
课时 50	集合_特点_创建和删除_交集并集差集运算	5:05

第一季 1.4 流程控制语句

顺序	名称	时长
课时 51	Pycharm 开发环境的下载安装配置_项目管理	14:02
课时 52	单分支选择结构_条件表达式详解	15:39
课时 53	双分子选择结构_三元运算符的使用详解	5:17
课时 54	多分支选择结构	9:17
课时 55	选择结构的嵌套	14:08
课时 56	while 循环结构_死循环处理	10:38
课时 57	for 循环结构_遍历各种迭代对象_range 对象	13:00
课时 58	嵌套循环	6:12
课时 59	嵌套循环练习_九九乘法表_打印表格数据	8:38
课时 60	break 语句	6:05
课时 61	continue 语句	5:09
课时 62	else 语句	3:56
课时 63	循环代码优化技巧	10:44
课时 64	zip()并行迭代	5:00
课时 65	推导式创建序列_列表推导式_字典推导式_集合推导式_生成器推导式	5:00
课时 66	综合练习_绘制不同颜色的多个同心圆_绘制棋盘	15:07

第一季 1.5 函数和内存分析

顺序	名称	时长
课时 67	函数的基本概念_内存分析_函数的分类_定义和调用	14:00
课时 68	形参和实参_文档字符串_函数注释	11:12
课时 69	返回值详解	8:57
课时 70	函数也是对象_内存分析	7:35

顺序	名称	时长
课时 71	变量的作用域_全局变量_局部变量_栈帧内存分析讲解	14:05
课时 72	局部变量和全局变量_效率测试	5:15
课时 73	参数的传递_传递可变对象_内存分析	8:38
课时 74	参数的传递_传递不可变对象_内存分析	5:05
课时 75	浅拷贝和深拷贝_内存分析	15:23
课时 76	参数的传递_不可变对象包含可变对象子对象_内存分析	10:47
课时 77	参数的类型_位置参数_默认值参数_命名参数	8:36
课时 78	参数的类型_可变参数_强制命名参数	4:17
课时 79	lambda 表达式和匿名函数	10:30
课时 80	eval()函数用法	6:05
课时 81	递归函数_函数调用内存分析_栈帧的创建	21:38
课时 82	递归函数_阶乘计算案例	8:23
课时 83	嵌套函数_内部函数_数据隐藏	11:22
课时 84	nonlocal_global	5:41
课时 85	LEGB 规则	6:07

第一季 1.6 面向对象和内存分析

顺序	名称	时长
课时 86	面向对象和面向过程的区别_执行者思维_设计者思维	14:50
课时 87	对象的进化故事	8:26
课时 88	类的定义_类和对象的关系	15:49
课时 89	构造函数__init__	7:39
课时 90	实例属性_内存分析	9:21
课时 91	实例方法_内存分析方法调用过程_dir()_isinstance()	13:21
课时 92	类和对象	6:38
课时 93	类属性_内存分析创建类和对象的底层	9:26
课时 94	类方法_静态方法_内存分析图示	10:13
课时 95	__del__()析构方法和垃圾回收机制	7:34
课时 96	__call__()方法和可调用对象	7:56
课时 97	方法没有重载_方法的动态性	10:24
课时 98	私有属性	7:00
课时 99	私有方法	5:12
课时 100	@property 装饰器_get 和 set 方法	15:23

顺序	名称	时长
课时 101	面向对象的三大特征说明（封装、继承、多态）	7:22
课时 102	继承	17:13
课时 103	方法的重写	5:03
课时 104	object 根类_dir()	4:54
课时 105	重写__str__()方法	3:25
课时 106	多重继承	4:03
课时 107	mro()	2:46
课时 108	super()获得父类的定义	5:05
课时 109	多态	7:33
课时 110	特殊方法和运算符重载	9:53
课时 111	特殊属性	6:14
课时 112	对象的浅拷贝和深拷贝_内存分析	12:34
课时 113	组合	8:38
课时 114	设计模式_工厂模式实现	9:23
课时 115	设计模式_单例模式实现	13:00

第二季 2.1 文件处理

顺序	名称	时长
课时 116	file 文件操作_操作系统底层关系_写入文件	18:21
课时 117	编码知识_中文乱码问题解决	18:53
课时 118	关闭流要点 1_try 异常管理	11:51
课时 119	关闭流要点 2_with 上下文管理_现场还原	4:47
课时 120	文本文件的读取	8:10
课时 121	enumerate()函数和推导式生成列表_操作每行增加行号	12:48
课时 122	二进制文件的读写_图片文件拷贝	4:02
课时 123	文件对象常用方法和属性总结_seek()任意位置操作	9:25
课时 124	使用 pickple 实现序列化和反序列化_神经元记忆移植	14:18
课时 125	CSV 文件的读取_写入	12:49
课时 126	os 模块_调用操作系统可执行文件_控制台乱码问题	8:25
课时 127	os 模块_获得文件信息_创建文件夹_递归创建	22:05
课时 128	os.path 模块_常用方法	18:24
课时 129	os 模块_使用 walk 遍历	9:05
课时 130	shutil 模块_文件和目录拷贝	8:03

顺序	名称	时长
课时 131	shutil 和 zipfile 模块_压缩和解压缩	5:56
课时 132	递归算法原理_阶乘计算	14:04
课时 133	递归算法原理_目录树结构展示	9:07

第二季 2.2 异常

顺序	名称	时长
课时 134	异常本质_调试核心理念	26:25
课时 135	try_except 基本结构	16:20
课时 136	try 多个 except 结构	8:32
课时 137	else 结构	3:54
课时 138	finally 结构	9:03
课时 139	常见异常汇总和说明	8:38
课时 140	with 上下文管理	4:39
课时 141	trackback 模块的使用_异常写入日志文件	6:11
课时 142	自定义异常类_raise 抛出异常	10:15
课时 143	Pycharm 的调试模块	22:16

第二季 2.3 模块

顺序	名称	时长
课时 144	模块化编程理念_什么是模块_哲学思想	11:59
课时 145	模块化编程的流程_设计和实现分离	19:26
课时 146	模块导入_import 和 from_import 语句详解和区别	11:46
课时 147	import 加载底层原理_importlib 模块	7:38
课时 148	包的概念和创建包和导入包	12:10
课时 149	包的本质和 init 文件_批量导入_包内引用	8:29
课时 150	sys.path 和模块搜索路径详解	12:07
课时 151	模块的本地发布_模块的安装	9:11
课时 152	PyPI 官网_远程上传和管理模块_PIP 方式安装模块	10:24

第二季 2.4 GUI 编程

顺序	名称	时长
课时 153	GUI 编程和 tinkter 介绍_第一个 GUI 程序	19:18
课时 154	PEP8 编码规范_窗口大小和位置	7：06

顺序	名称	时长
课时 155	GUI 编程整体描述_常用组件汇总	7:39
课时 156	GUI 程序的经典面向对象写法	22:41
课时 157	Label 组件_tkinter 中图像正确显示全局变量写法	16:49
课时 158	options 选项详解_底层源码分析和阅读_可变参数和运算符重载复习	15:31
课时 159	Button_anchor 位置控制	10:33
课时 160	Entry_StringVar_登录界面设计和功能实现	18:51
课时 161	Text 多行文本框详解_复杂 tag 标记	12:40
课时 162	Radiobutton_Checkbutton 详解	8:17
课时 163	Canvas 画布组件	8:09
课时 164	Grid 布局管理器详解	7:43
课时 165	计算器软件界面的设计	17:27
课时 166	Pack 布局管理器_钢琴软件界面设计	6:38
课时 167	Place 管理器_绝对位置和相对位置	6:28
课时 168	扑克游戏界面设计_增加事件操作	14:54
课时 169	事件机制和消息循环原理_鼠标事件_键盘事件_event 对象	18:32
课时 170	lambda 表达式_事件传参应用	7:41
课时 171	三种事件绑定方式总结	4:23
课时 172	optionmenu 选项菜单_scale 滑块	7:34
课时 173	颜色框_文件选择框_读取文件内容	9:01
课时 174	简单对话框_通用消息框_ttk 子模块问题	5:59
课时 175	主菜单_上下文菜单	7:44
课时 176	【记事本项目 01】_打开和保存修改文件的实现	11:27
课时 177	【记事本项目 02】_新建文件_背景色改变_快捷键功能	13:02
课时 178	【记事本项目 03】python 项目打包成 exe 可执行文件	4:04
课时 179	【画图项目 01】_界面实现	13:35
课时 180	【画图项目 02】_绘制直线_拖动删除上一个图形	12:55
课时 181	【画图项目 03】_箭头直线_矩形绘制	5:36
课时 182	【画图项目 04】_画笔和橡皮檫实现	7:48
课时 183	【画图项目 05】_清屏_颜色框_快捷键处理	8:33

第二季 2.5 坦克大战

顺序	名称	时长
课时 184	Pygame 模块的安装	7:25

顺序	名称	时长
课时 185	面向对象分析项目需求	6:42
课时 186	坦克大战项目框架搭建	6:31
课时 187	加载主窗口	12:05
课时 188	坦克大战之事件处理	10:24
课时 189	左上角文件的绘制	16:58
课时 190	加载我方坦克	15:16
课时 191	我方坦克切换方向_移动	7:00
课时 192	我方坦克移动优化	8:30
课时 193	我方坦克优化 2	13:11
课时 194	加载敌方坦克	15:31
课时 195	敌方坦克随机移动	11:15
课时 196	完善子弹类	12:00
课时 197	我方坦克发射子弹	7:24
课时 198	子弹移动	10:35
课时 199	子弹消亡及数量控制	8:51
课时 200	敌方坦克发射子弹	12:07
课时 201	我方子弹与敌方坦克的碰撞	12:47
课时 202	实现爆炸效果	13:13
课时 203	我方坦克的消亡	10:42
课时 204	我方坦克无限重生	7:18
课时 205	加载墙壁	12:40
课时 206	子弹不能穿墙	8:09
课时 207	坦克不能穿墙	10:14
课时 208	敌我双方坦克发生碰撞	10:12
课时 209	音效处理	9:43

第二季 2.6 数据库编程输出

顺序	名称	时长
课时 210	操作 SQLite3 创建表	10:43
课时 211	操作 SQLite3 数据库插入数据	11:22
课时 212	操作 SQLite3 数据库查询数据	8:05
课时 213	操作 SQLite3 数据库修改_删除数据	8:40
课时 214	MySQL 数据库的下载	4:25

顺序	名称	时长
课时 215	MySQL 数据库的安装	5:54
课时 216	PyMySQL 模块的安装	5:18
课时 217	操作 MySQL 数据库创建表	8:36
课时 218	操作 MySQL 数据库插入数据	10:57
课时 219	操作 MySQL 数据库查询数据	9:31
课时 220	操作 MySQL 数据库修改_删除数据	8:00

第二季 2.7 numpy

顺序	名称	时长
课时 221	numpy 是什么及 numpy 模块的安装	8:34
课时 222	array 进行创建数组	8:39
课时 223	arange 创建数组	7:34
课时 224	随机数创建数组 1	11:44
课时 225	随机数创建数组 2	10:30
课时 226	ndarray 对象的属性	14:01
课时 227	其他方式创建数组	15:37
课时 228	一维数组的切片索引	8:35
课时 229	二维数组的切片和索引	18:59
课时 230	数组的复制	8:28
课时 231	修改数组的维度	12:14
课时 232	数组的拼接	15:32
课时 233	数组的分割	18:55
课时 234	数组的转置	10:54
课时 235	函数 1	19:19
课时 236	函数 2	11:11

第二季 2.8 matplotlib

顺序	名称	时长
课时 237	基本绘制图形	12:26
课时 238	设置样式	5:44
课时 239	绘制曲线	9:05
课时 240	subplot 的使用	8:03
课时 241	绘制散点图	19:28

顺序	名称	时长
课时 242	绘制不同样式不同颜色的线条	10:44
课时 243	绘制柱状图	9:17
课时 244	bar 及 barh 函数的使用	14:50
课时 245	柱状图使用实例	16:23
课时 246	绘制饼状图	10:22
课时 247	绘制直方图	8:56
课时 248	绘制等高线图和三维图	10:57

第三季 3.1 并发编程

顺序	名称	时长
课时 249	多任务的概念	8:04
课时 250	创建子进程并调用	14:24
课时 251	join 方法的使用	9:34
课时 252	属性的使用_多任务的创建	14:16
课时 253	使用继承方式创建进程	8:23
课时 254	进程池的使用	14:06
课时 255	多个进程之间数据不共享	6:40
课时 256	队列常用方法使用	10:57
课时 257	进程之间通信	9:41
课时 258	进程和线程的区别	6:01
课时 259	thread 创建线程	11:18
课时 260	threading 模块创建线程	18:13
课时 261	线程之间共享全局变量	17:46
课时 262	互斥锁	9:06
课时 263	线程同步的使用	10:08
课时 264	生产者消费者模块	12:51
课时 265	ThreadLocal 的使用	9:03

第三季 3.2 网络编程

顺序	名称	时长
课时 266	IP 地址_端口	10:43
课时 267	网络通信协议	6:20
课时 268	TCP 协议_UDP 协议	7:09

顺序	名称	时长
课时 269	UDP 发送数据	8:40
课时 270	UDP 接收数据	13:27
课时 271	UDP 使用多线程实现聊天	11:26
课时 272	TFTP 文件下载器过程及格式介绍	23:02
课时 273	TFTP 下载器客户端实现	18:55
课时 274	TCP 通信	9:15
课时 275	TCP 服务器端接收 数据	7:27
课时 276	TCP 模拟 QQ	12:21
课时 277	TCP 多线程完成聊天	23:51
课时 278	TCP 多线程聊天优化	8:12

第四季 4.1 算法

顺序	名称	时长
课时 279	算法的概念	26:02
课时 280	第二次获取值	8:02
课时 281	时间复杂度	13:27
课时 282	最坏时间复杂度_常见时间复杂度与大小关系	10:12
课时 283	空间复杂度	7:06
课时 284	排序算法的稳定性	3:57
课时 285	冒泡排序	28:00
课时 286	选择排序	12:55
课时 287	选择排序_时间复杂度_稳定性	5:32
课时 288	插入排序	13:35
课时 289	选择排序_时间复杂度	6:06
课时 290	快速排序的思想	8:18
课时 291	快速排序实现	11:35
课时 292	快速排序_时间复杂度	9:49
课时 293	归并排序思想	10:28
课时 294	归并排序实现	14:30
课时 295	归并排序_时间复杂度	6:34
课时 296	顺序查找法	5:43
课时 297	二分法查找	14:29

第四季 4.2 数据结构

顺序	名称	时长
课时 298	数据结构的引入	12:15
课时 299	顺序表	11:26
课时 300	测试 list 列表中 insert 和 append 的执行速度	9:05
课时 301	链表的引入	8:20
课时 302	单链表及节点的定义	14:14
课时 303	单链表_是否为空_计算长度方法的实现	16:44
课时 304	单链表_查找_遍历方法的实现	6:58
课时 305	单链表_头部_尾部添加节点	20:14
课时 306	单链表指定位置插入元素	15:24
课时 307	单链表删除节点	14:39
课时 308	链表与顺序表的对比	6:18
课时 309	双向链表节点定义	6:57
课时 310	双向链表添加节点	12:40
课时 311	双向链表指定位置插入节点	7:29
课时 312	双向链表其它操作	13:49
课时 313	栈的实现	10:38
课时 314	队列的实现	7:17
课时 315	树的概念	21:05
课时 316	二叉树的概念	6:59
课时 317	二叉树节点定义_添加节点	14:26
课时 318	广度优先遍历	9:09
课时 319	深度优先遍历	18:58

第四季 4.3 函数式编程和高阶函数

顺序	名称	时长
课时 320	高阶函数概念	13:42
课时 321	高阶函数 map 的使用	16:05
课时 322	高阶函数 reduce 的使用	12:38
课时 323	高阶函数 filter 的使用	11:55
课时 324	高阶函数 sorted 的使用	12:43
课时 325	匿名函数	14:55
课时 326	闭包定义及使用	10:43

顺序	名称	时长
课时 327	使用闭包求两点之间的距离	9:43
课时 328	闭包的特殊用途	16:54
课时 329	装饰器的基本使用	13:29
课时 330	多个装饰器的使用	9:32
课时 331	带参数的装饰器	11:51
课时 332	通用装饰器	7:04
课时 333	偏函数	8:09

第四季 4.4 正则表达式

顺序	名称	时长
课时 334	正则表达式的概念	8:18
课时 335	match 方法的使用	10:03
课时 336	常用字符的使用	12:20
课时 337	重复数量限定符	15:23
课时 338	重复数量限定符的使用	16:21
课时 339	原生字符串	5:48
课时 340	边界字符的使用	9:17
课时 341	search 方法的使用	12:29
课时 342	择一匹配符和列表的使用差异	9:44
课时 343	正则表达式分组的使用	17:01
课时 344	其他函数的使用	18：12
课时 345	贪婪模式和非贪婪模式	6:44

第四季 4.5 pillow 图像处理

顺序	名称	时长
课时 346	Image 打开显示图片	12:10
课时 347	Image 完成图像混合	8:19
课时 348	图像的缩放_复制_剪切_粘贴	12:23
课时 349	图像的旋转_分离合并	10:48
课时 350	图像滤镜	10:22
课时 351	图片合成	11:26
课时 352	调整图像色彩_亮度	17:26
课时 353	ImageDraw 绘图二维图像	15:36

顺序	名称	时长
课时 354	ImageFont 的使用	12:09
课时 355	绘制十字	6:51
课时 356	绘制验证码	15:29
课时 357	绘制九宫格	11:35
课时 358	将图片中黄色修改为红色	9:56
课时 359	读取图片实例	21:28

第四季 4.6 人脸识别

顺序	名称	时长
课时 360	加载图片	10:51
课时 361	将图片灰度转换	6:40
课时 362	修改图片尺寸	8:18
课时 363	绘制矩形_圆	9:38
课时 364	人脸检测	13:00
课时 365	检测多张人脸	14:37
课时 366	检测视频中人脸	8:49
课时 367	训练数据	25:06
课时 368	人脸识别	10:13

第四季 4.7 语音识别

顺序	名称	时长
课时 369	原生字符串	5:54
课时 370	边界字符的使用	9:13
课时 371	search 方法的使用	8:10

第四季 4.8 协程_异步 IO

顺序	名称	时长
课时 372	协程的概念	12:03
课时 373	yeild 的使用	14:45
课时 374	yield 实现生产者消费者	9:21
课时 375	同步和异步的概念	10:18
课时 376	定义协议	9:03
课时 377	创建任务 task	8:14

顺序	名称	时长
课时 378	绑定回调	7:57
课时 379	阻塞和 await	7:46
课时 380	asyncio 实现并发	9:51
课时 381	协程嵌套	15:25
课时 382	协程停止	16:04

第四季 4.4 正则表达式

顺序	名称	时长
课时 383	神经网络原理	12:52
课时 384	激活函数	7:45
课时 385	TensorFlow 简介	10:46
课时 386	查看默认图	8:36
课时 387	自定义图	6:48
课时 388	使用 TensorBoard 将图可视化	8:36
课时 389	会话 Session	11:37
课时 390	feed_dict 的使用	10:59
课时 391	张量的创建	14:59
课时 392	修改张量形状	16:45
课时 393	矩阵运算	13:38
课时 394	变量的定义	9:11
课时 395	线性回归案例实现	24:12
课时 396	增加变量显示	10:37
课时 397	增加命名空间	6:58
课时 398	保存读取模型	9:23
课时 399	加载 mnist 数据集	22:10
课时 400	手写数字识别	18:13
课时 401	手写数字识别_增加变量显示_命名空间	11:36
课时 402	手写数字识别_保存读取模型	8:45