



高等学校计算机科学与技术教材

Python 程序设计教程

COMPUTER Science and Technology

□ 江 红 余青松 编著

- 原理与技术的完美结合
- 教学与科研的最新成果
- 语言精练，实例丰富
- 可操作性强，实用性突出



清华大学出版社 ● 北京交通大学出版社

- ☐ VB.NET 程序设计实验指导与习题测试
- ☐ VB.NET 程序设计
- ☐ 面向对象程序设计 (C# 实现)
- ☐ 高等计算机网络与安全
- ☐ 电子支付与网络银行
- ☐ SQL Server 2005 实用教程
- ☐ Java 程序设计与应用
- ☐ ASP .NET Web 应用程序设计教程
- ☐ 操作系统实验教程
- ☐ Web 程序设计
- ☐ Java 精解案例教程
- ☐ 计算机硬件维修
- ☐ 软件工程基础
- ☐ 计算机系统组装与维护
- ☐ 数据结构实例教程
- ☐ 程序设计导论——Java 编程
- ☐ 网络工程与组网技术
- ☐ Visual Basic 程序设计实验与习题指导
- ☐ 数据库应用系统开发实例
- ☐ 多媒体技术基础
- ☐ ASP 动态网页设计教程
- ☐ Access 2010 数据库应用教程
- ☐ Delphi 数据库编程
- ☐ Java EE 编程技术 (第2版)
- ☐ SQL Server 2005 数据库原理及应用教程
- ☐ 基于 Web 的远程监控系统
- ☐ ANSYS 辅助分析应用基础教程 (第2版)
- ☐ Microsoft Project 2003 项目管理与应用: 上机指导
- ☐ Visual C++ 程序设计与实践: 实验与指导

- ☐ Visual C# 应用程序设计
- ☐ Java 程序设计与网络编程
- ☐ PHP 精解案例教程
- ☐ 计算机网络安全教程 (修订本)
- ☐ Microsoft Project 2003 项目管理与应用
- ☐ 软件工程
- ☐ 网站全程设计技术 (修订本)
- ☐ ASP 精解案例教程 (修订本)
- ☐ 计算机网络与应用教程
- ☐ 计算机网络管理——Windows 2000 管理基础
- ☐ 嵌入式系统的设计与开发
- ☐ 多媒体课件设计理论与实践
- ☐ 数据库系统及应用基础
- ☐ 计算机组成原理
- ☐ Linux 操作系统分析教程
- ☐ 数据结构基础教程
- ☐ 微机原理与接口技术
- ☐ Visual Basic 高级图形应用程序设计
- ☐ 多媒体计算机外部设备
- ☐ Visual FoxPro 系统开发教程
- ☐ 计算机网络技术与应用
- ☐ 计算机网络基础教程 (修订本)
- ☐ C 语言程序设计
- ☐ C# 程序设计教程
- ☐ 计算机维修教程
- ☒ Python 程序设计教程

责任编辑: 谭文芳
封面美编: 乔 楚

ISBN 978-7-5121-2043-3



9 787512 120433 >

定价: 45.00 元

高等学校计算机科学与技术教材

Python 程序设计教程

江 红 余青松 编著

清华大学出版社
北京交通大学出版社
· 北京 ·

内 容 简 介

本书基于 Windows 7 和 Python 3.3.2 构建 Python 开发平台, 阐述 Python 语言的基础知识, 以及使用 Python 语言的实际开发应用实例。本书集教材、练习册、上机指导于一体, 具体内容包括: Python 概述、Python 语言基础、程序流程控制、数值类型、系列(元组、列表和字符串)、字典和集合类型、文件和流 I/O、函数与函数编程、类和对象、模块和包、迭代器和生成器、数据结构和算法、日期和时间处理、正则表达式、多线程编程、图形用户界面应用程序、数据库访问、网络编程和通信、系统管理。

本书作者结合多年的程序设计、开发及授课经验, 由浅入深、循序渐进地介绍 Python 程序设计语言, 让读者能够较为系统全面地掌握程序设计的理论和应用。

本书可作为高等学校各专业的计算机程序设计教材, 同时也可作为广大程序设计开发者、爱好者的自学参考书。

本书封面贴有清华大学出版社防伪标签, 无标签者不得销售。

版权所有, 侵权必究。侵权举报电话: 010-62782989 13501256678 13801310933

图书在版编目(CIP)数据

Python 程序设计教程 / 江红, 余青松编著. —北京: 北京交通大学出版社; 清华大学出版社, 2014. 9

(高等学校计算机科学与技术教材)

ISBN 978-7-5121-2043-3

I. ①P… II. ①江… ②余… III. ①软件工具-程序设计-高等学校-教材
IV. ①TP311.56

中国版本图书馆 CIP 数据核字 (2014) 第 188247 号

责任编辑: 谭文芳

出版发行: 清华大学出版社 邮编: 100084 电话: 010-62776969

北京交通大学出版社 邮编: 100044 电话: 010-51686414

印刷者: 北京交大印刷厂

经 销: 全国新华书店

开 本: 185×260 印张: 23.75 字数: 729 千字

版 次: 2014 年 9 月第 1 版 2014 年 9 月第 1 次印刷

书 号: ISBN 978-7-5121-2043-3/TP·793

印 数: 1~2 000 册 定价: 45.00 元

本书如有质量问题, 请向北京交通大学出版社质监局反映。对您的意见和批评, 我们表示欢迎和感谢。

投诉电话: 010-51686043, 51686008; 传真: 010-62225406; E-mail: press@bjtu.edu.cn。

前 言

程序设计是大专院校计算机、电子信息、工商管理等相关专业的必修课程。Python 语言是一种解释型、面向对象的计算机程序设计语言，广泛用于计算机程序设计教学语言、系统管理脚本语言、科学计算等，特别适用于快速的应用程序开发。Python 编程语言广受开发者的喜爱，并被列入 LAMP（Linux、Apache、MySQL 以及 Python/Perl/PHP）。

本教程集“教材、练习册、上机指导”于一体，基于 Windows 7 和 Python 3.3.2 构建 Python 开发平台，阐述 Python 语言的基础知识，以及使用 Python 语言的实际开发应用实例，具体内容包括：Python 概述、Python 语言基础、程序流程控制、数值类型、系列（元组、列表和字符串）、字典和集合类型、文件和流 I/O、函数与函数编程、类和对象、模块和包、迭代器和生成器、数据结构和算法、日期和时间处理、正则表达式、多线程编程、图形用户界面应用程序、数据库访问、网络编程和通信、系统管理。

本教程涉及的各章节所有的源程序代码和相关素材，以及供教师参考的教学电子文稿均可以通过北京交通大学出版社网站 <http://www.bjtup.com.cn> 下载，也可以通过电子邮件 hjiang@cc.ecnu.edu.cn 直接与作者联系。

本教程由华东师范大学江红和余青松共同编写。衷心感谢本书的编辑谭文芳老师，敬佩她的睿智和敬业。由于时间和编者学识有限，书中不足之处在所难免，敬请诸位同行、专家和读者指正。

编 者

2014 年 7 月

目 录

第 1 章 Python 概述	1
1.1 Python 语言概述	1
1.1.1 Python 语言简介	1
1.1.2 Python 语言的特点	1
1.1.3 Python 语言的应用范围	2
1.2 Python 语言的版本和开发环境	2
1.2.1 Python 语言的版本	2
1.2.2 Python 语言的实现	3
1.2.3 Python 语言的集成开发环境	3
1.3 下载和安装 Python	3
1.3.1 下载 Python	3
1.3.2 安装 Python	4
1.4 使用 Python 解释器解释执行 Python 程序	5
1.4.1 运行 Python 解释器	5
1.4.2 运行 Python 集成开发环境	6
1.5 使用文本编辑器和命令行编写和执行 Python 源文件程序	7
1.5.1 编写 Hello world 程序	7
1.5.2 Hello world 程序 (hello.py) 源代码分析	8
1.5.3 运行 Python 源代码程序	8
1.6 使用集成开发环境 IDLE 编写和执行 Python 源文件程序	9
1.6.1 使用 IDLE 编写 Hello world 程序	10
1.6.2 使用 IDLE 编辑程序	10
1.7 复习题	11
1.8 上机实践	12
第 2 章 Python 语言基础	13
2.1 语句	13
2.1.1 Python 语句	13
2.1.2 Python 语句的书写规则	13
2.1.3 复合语句及其缩进书写规则	14
2.1.4 注释语句	14
2.1.5 空语句 pass	15
2.2 表达式	15
2.2.1 表达式的组成	15

2.2.2	表达式的书写规则	15
2.3	运算符	15
2.3.1	运算符概述	15
2.3.2	Python 运算符及其优先级	16
2.4	标识符及其命名规则	17
2.4.1	标识符	17
2.4.2	保留关键字	17
2.4.3	Python 预定义标识符	18
2.4.4	命名规则	18
2.5	对象与引用	18
2.5.1	对象的类型 (type) 和标识 (id)	18
2.5.2	对象引用	19
2.5.3	对象比较 (==) 和类型判别 (is)	19
2.5.4	不可变对象 (immutable) 和可变对象 (mutable)	20
2.6	变量和赋值	20
2.6.1	变量和数据类型	20
2.6.2	变量的声明和赋值	21
2.6.3	链式赋值语句	21
2.6.4	复合赋值语句	21
2.6.5	删除变量 (del)	22
2.6.6	系列解包赋值	22
2.7	数据类型	23
2.7.1	NoneType, NotImplementedType 和 Ellipsis	23
2.7.2	数值数据类型	23
2.7.3	序列数据类型	24
2.7.4	集合数据类型	25
2.7.5	字典数据类型	25
2.7.6	其他数据类型	25
2.8	类的声明和对象的创建与调用	26
2.8.1	类的声明	26
2.8.2	对象的创建和调用	26
2.9	函数	27
2.9.1	函数的声明和调用	27
2.9.2	内置函数	28
2.9.3	模块函数和 import 语句	28
2.9.4	输入和输出函数	29
2.9.5	运行时提示输入密码	29
2.10	模块和包	30
2.11	Python 文档注释	30

2.11.1 文档字符串	30
2.11.2 文档注释规范	31
2.12 复习题	32
2.13 上机实践	35
第3章 程序流程控制	36
3.1 bool 数据类型和相关运算符	36
3.1.1 bool 类型	36
3.1.2 关系和测试运算符	36
3.1.3 逻辑运算符	37
3.2 顺序结构	38
3.3 选择结构	39
3.3.1 分支结构的形式	39
3.3.2 单分支结构	39
3.3.3 双分支结构	40
3.3.4 多分支结构	41
3.3.5 if 语句的嵌套	42
3.3.6 选择结构综合举例	43
3.4 循环结构	44
3.4.1 可迭代对象 (iterable)	44
3.4.2 for 循环	45
3.4.3 range 对象	45
3.4.4 while 循环	46
3.4.5 循环的嵌套	47
3.4.6 break 语句	48
3.4.7 continue 语句	49
3.4.8 else 子句	50
3.5 异常处理	51
3.5.1 错误和异常	51
3.5.2 异常处理概述	51
3.5.3 内置的异常类	52
3.5.4 自定义异常类	53
3.5.5 引发异常	53
3.5.6 捕获处理异常 try...except...finally	54
3.6 断言处理	56
3.6.1 断言处理概述	56
3.6.2 assert 语句和 AssertionError 类	57
3.6.3 启用/禁用断言	57
3.7 复习题	58
3.8 上机实践	62

第 4 章 数值类型	64
4.1 int 类型（任意精度整数）	64
4.1.1 整型常量	64
4.1.2 int 对象	65
4.1.3 数制转换函数	65
4.1.4 整数的运算	65
4.2 float 类型（有限精度浮点数）	65
4.2.1 浮点类型常量	66
4.2.2 float 对象	66
4.2.3 浮点数的运算	66
4.3 Decimal 类型（高精度浮点数）	67
4.3.1 浮点数运算误差	67
4.3.2 Decimal 对象	67
4.3.3 Context 对象	68
4.3.4 高精度浮点数的运算	71
4.4 Fraction 类型（分数）	71
4.4.1 Fraction 对象	71
4.4.2 分数的运算	72
4.5 complex 类型（复数）	72
4.5.1 复数类型常量	72
4.5.2 创建 complex 对象	72
4.5.3 complex 类型的方法	73
4.5.4 复数的运算	73
4.6 算术运算符和位运算符	73
4.6.1 算术运算符	73
4.6.2 位运算符	74
4.6.3 混合运算和数值类型转换	74
4.6.4 内置标准数学函数	75
4.7 math 模块和数学函数	75
4.8 cmath 模块和复数数学函数	78
4.9 random 模块和随机函数	79
4.9.1 种子和随机状态	80
4.9.2 随机整数	80
4.9.3 随机系列	81
4.9.4 随机实值分布	82
4.10 相关模块	82
4.10.1 标准库中的相关模块	82
4.10.2 数值运算模块 NumPy	83
4.10.3 科学计算模块 SciPy	83

4.11	复习题	83
4.12	上机实践	84
第5章	系列：元组、列表和字符串	89
5.1	系列的基本操作	89
5.1.1	系列的索引访问操作	89
5.1.2	系列的切片操作	90
5.1.3	系列的连接和重复操作	91
5.1.4	系列的成员关系操作	91
5.1.5	系列的比较运算操作	92
5.1.6	系列的排序操作	92
5.1.7	系列长度、最大值、最小值、求和	93
5.1.8	内置函数 all() 和 any()	93
5.1.9	系列拆封	93
5.2	元组	94
5.2.1	元组的定义	94
5.2.2	元组的基本操作	95
5.3	列表	95
5.3.1	列表的定义	95
5.3.2	列表的基本操作	95
5.3.3	list 对象的方法	96
5.3.4	列表解析表达式	96
5.3.5	列表作为队列和栈	97
5.4	字符串	97
5.4.1	Unicode 和字符常量	97
5.4.2	字符串常量	98
5.4.3	创建 str 对象	99
5.4.4	字符串的基本操作	99
5.4.5	str 对象的方法	99
5.4.6	字符串编码	102
5.4.7	字符串格式化	103
5.5	字节系列	105
5.5.1	bytes 常量	105
5.5.2	创建 bytes 对象	106
5.5.3	创建 bytearray 对象	106
5.5.4	bytes 和 bytearray 的基本操作	107
5.6	复习题	107
5.7	上机实践	110
第6章	字典和集合类型	112
6.1	字典	112

6.1.1	对象的 hash 值	112
6.1.2	字典的定义	112
6.1.3	字典的访问操作	113
6.1.4	字典的视图对象	113
6.1.5	判断字典键是否存在	113
6.1.6	字典对象的长度和比较	114
6.1.7	字典对象的方法	114
6.2	集合	115
6.2.1	集合的定义	115
6.2.2	判断集合元素是否存在	115
6.2.3	集合的运算：并集、交集、差集和对称差集	116
6.2.4	集合的比较运算：相等、子集和超集	116
6.2.5	集合的长度、最大值、最小值、元素和	117
6.2.6	可变集合的方法	117
6.3	复习题	118
6.4	上机实践	119
第7章	文件和流 I/O	121
7.1	文件和文件对象	121
7.1.1	文件和流概述	121
7.1.2	文件对象和 open 函数	121
7.1.3	with 语句和上下文管理协议	122
7.2	文本文件的读取和写入	122
7.2.1	文本文件的写入	122
7.2.2	文本文件的读取	123
7.2.3	文本文件的编码	124
7.3	二进制文件的读取和写入	125
7.3.1	二进制文件的写入	125
7.3.2	二进制文件的读取	125
7.4	随机文件访问	126
7.5	CSV 文件格式的读取和写入	127
7.5.1	csv.reader 对象和 csv 文件的读取	127
7.5.2	csv.writer 对象和 csv 文件的写入	128
7.5.3	csv.DictReader 对象和 csv 文件的读取	128
7.5.4	csv.DictWriter 对象和 csv 文件的写入	129
7.5.5	csv 文件格式化参数和 Dialect 对象	130
7.6	os 模块和文件访问	131
7.6.1	文件描述符	131
7.6.2	使用 os 模块提供的函数访问文件	131
7.7	对象序列化	132

7.7.1 对象序列化	132
7.7.2 pickle 模块和对象序列化	132
7.8 输入重定向和管道	133
7.8.1 FileInput 对象	133
7.8.2 fileinput 模块的函数	134
7.8.3 输入重定向	135
7.9 复习题	136
7.10 上机实践	136
第8章 函数与函数编程	137
8.1 函数的声明和调用	137
8.2 参数的传递	138
8.2.1 形式参数和实际参数	138
8.2.2 可选参数	138
8.2.3 位置参数和命名参数	139
8.2.4 可变参数 (VarArgs)	140
8.2.5 强制命名参数 (Keyword - only)	140
8.3 函数的返回值	141
8.4 变量的作用域	142
8.4.1 全局变量	142
8.4.2 局部变量	142
8.4.3 类成员变量	142
8.4.4 全局语句 global	142
8.4.5 非局部语句 nonlocal	143
8.5 函数装饰器	144
8.5.1 装饰器的声明和使用	144
8.5.2 @functools.wraps 装饰器	146
8.5.3 functools.update_wrapper 函数	146
8.6 递归函数	147
8.7 作为对象的函数	148
8.8 Lambda 表达式和匿名函数	148
8.9 operator 模块和运算符函数	149
8.10 eval、exec 和 compile 函数	149
8.10.1 运行上下文的局部变量和全局变量	149
8.10.2 eval 函数和动态表达式的求值	150
8.10.3 exec 函数和动态语句的执行	150
8.10.4 compile 函数和动态语句的执行	150
8.11 functools 模块和函数工具	150
8.11.1 partial 对象	151
8.11.2 reduce 函数	151

8.11.3	@functools.lru_cache 装饰器	151
8.12	复习题	152
8.13	上机实践	153
第9章	类和对象	155
9.1	面向对象概念	155
9.1.1	对象的定义	155
9.1.2	封装	155
9.1.3	继承	156
9.1.4	多态性	156
9.2	类和对象	156
9.2.1	类的声明	157
9.2.2	对象的创建和使用	157
9.3	属性	157
9.3.1	实例属性和类属性	158
9.3.2	私有属性和公有属性	160
9.3.3	@property 装饰器	160
9.3.4	特殊属性 (Special Attributes)	161
9.3.5	自定义属性 (Custom Attributes)	162
9.4	方法	163
9.4.1	方法的声明和调用	163
9.4.2	__init__方法(构造函数)和__new__方法	164
9.4.3	__del__方法(析构函数)	164
9.4.4	实例方法、类方法 (@classmethod) 与静态方法 (@staticmethod)	165
9.4.5	私有方法与公有方法	166
9.4.6	方法重载	167
9.4.7	运算符重载	168
9.4.8	@functools.total_ordering 装饰器	169
9.4.9	特殊方法(Special Method)	170
9.4.10	__call__方法和可调用对象(callable)	170
9.5	继承	171
9.5.1	派生类	171
9.5.2	类成员的继承和重写	172
9.6	对象的引用、浅拷贝和深拷贝	173
9.6.1	对象的引用	173
9.6.2	对象的浅拷贝	173
9.6.3	对象的深拷贝	174
9.7	复习题	174
9.8	上机实践	175
第10章	模块和包	177

10.1	模块的导入和使用	177
10.1.1	import 语句	177
10.1.2	from...import 语句	177
10.1.3	__import__() 内置函数	178
10.1.4	模块搜索路径 sys.path	178
10.1.5	dir() 内置函数	178
10.2	模块的定义	179
10.2.1	创建模块	179
10.2.2	模块的名称	180
10.2.3	按字节编译的 .pyc 文件	181
10.3	包的定义	181
10.3.1	包的概念	181
10.3.2	创建包	182
10.3.3	包的导入和使用	182
10.4	命令行参数	182
10.4.1	sys.argv 与命令行参数	182
10.4.2	argparse 和命令行选项参数解析	183
10.5	终止程序运行时返回消息	185
10.6	复习题	185
10.7	上机实践	186
第 11 章	迭代器和生成器	187
11.1	迭代和可迭代对象	187
11.1.1	可迭代对象、迭代器和可迭代协议	187
11.1.2	可迭代对象的迭代: iter 和 next 函数	188
11.1.3	可迭代对象的迭代: for 语句	188
11.2	自定义可迭代对象和迭代器	189
11.3	生成器函数	189
11.4	反向迭代: reversed 迭代器	190
11.5	生成器表达式	191
11.6	Python 内置的可迭代对象	191
11.6.1	range 可迭代对象	191
11.6.2	map 迭代器和 itertools.starmap 迭代器	191
11.6.3	filter 迭代器和 itertools.filterfalse 迭代器	192
11.6.4	zip 迭代器和 zip_longest 迭代器	192
11.6.5	enumerate 迭代器	193
11.7	itertools 模块和迭代器函数	193
11.7.1	无穷系列迭代器	193
11.7.2	累计迭代器 accumulate	194
11.7.3	级联迭代器 chain	194

11.7.4	选择压缩迭代器 compress	194
11.7.5	截取迭代器 dropwhile 和 takewhile	194
11.7.6	切片迭代器 islice	195
11.7.7	迭代器 groupby	195
11.7.8	返回多个迭代器 tee	195
11.7.9	组合迭代器 combinations、combinations_with_replacement	195
11.7.10	排列迭代器 permutations	196
11.7.11	笛卡儿积迭代器 product	196
11.8	复习题	196
11.9	上机实践	198
第 12 章	数据结构和算法	200
12.1	ABC 模块	200
12.1.1	collections.abc 模块和抽象基类	200
12.1.2	collections.abc 模块中的抽象基类	200
12.2	collections 模块和容器类型	202
12.2.1	ChainMap 对象	202
12.2.2	Counter 对象	203
12.2.3	deque 对象	203
12.2.4	defaultdict 对象	205
12.2.5	OrderedDict 对象	205
12.2.6	namedtuple 对象	206
12.2.7	UserDict、UserList 和 UserString 对象	207
12.3	array.array 对象	207
12.3.1	array 的定义	207
12.3.2	array 对象的基本操作	208
12.3.3	array 对象的方法	208
12.4	heapq 模块和堆队列算法	209
12.4.1	堆 (Heap) 的基本概念	209
12.4.2	堆的基本操作	210
12.4.3	heapq 模块中基于 heap 的通用函数	210
12.5	bisect 模块和二分排序算法	211
12.6	复习题	211
12.7	上机实践	213
第 13 章	日期和时间处理	214
13.1	日期和时间处理概述	214
13.1.1	相关模块	214
13.1.2	相关术语	214
13.2	time 模块	215
13.2.1	struct_time 对象	215

13.2.2	time 模块中的常用函数	215
13.2.3	日期格式化字符串	216
13.3	datetime 模块	217
13.3.1	date 对象	218
13.3.2	tzinfo 对象和 timezone 对象	219
13.3.3	time 对象	220
13.3.4	datetime 对象	220
13.3.5	timedelta 对象	222
13.3.6	日期和时间的运算与比较	223
13.4	calendar 模块	223
13.4.1	Calendar 对象	223
13.4.2	TextCalendar 对象和 LocaleTextCalendar 对象	224
13.4.3	HTMLCalendar 对象和 LocaleHTMLCalendar 对象	225
13.4.4	calendar 模块的属性和常用函数	226
13.5	复习题	227
13.6	上机实践	228
第 14 章	正则表达式	230
14.1	正则表达式语言	230
14.1.1	正则表达式语言概述	230
14.1.2	正则表达式引擎	230
14.1.3	普通字符和转义字符	231
14.1.4	字符类和预定义字符类	231
14.1.5	边界匹配符	232
14.1.6	重复限定符	232
14.1.7	匹配算法：贪婪和懒惰	233
14.1.8	选择分支	234
14.1.9	分组和向后引用	234
14.1.10	正则表达式的匹配模式	236
14.1.11	常用正则表达式	236
14.2	正则表达式模块 re	236
14.2.1	匹配和搜索函数	236
14.2.2	匹配选项	237
14.2.3	正则表达式对象	237
14.2.4	匹配对象	238
14.2.5	匹配和替换	238
14.2.6	分割字符串	238
14.3	正则表达式应用举例	239
14.3.1	字符串包含验证	239
14.3.2	字符串查找和拆分	239

14.3.3	字符串替换和清除	240
14.3.4	字符串查找和截取	240
14.4	复习题	241
14.5	上机实践	242
第 15 章	多线程编程	243
15.1	线程处理概述	243
15.1.1	进程和线程	243
15.1.2	线程的优缺点	243
15.2	创建和启动多线程	244
15.2.1	使用 start_new_thread 函数创建线程	244
15.2.2	使用 Thread 对象创建线程	244
15.2.3	自定义派生于 Thread 的对象	245
15.2.4	线程加入 join()	246
15.2.5	用户线程和 daemon 线程	247
15.3	线程同步	248
15.3.1	线程同步处理	248
15.3.2	基于原语锁 (Lock/RLock 对象) 的简单同步	248
15.3.3	基于条件变量 (Condition 对象) 的同步和通信	250
15.4	复习题	253
15.5	上机实践	253
第 16 章	图形用户界面应用程序	254
16.1	图形用户界面概述	254
16.1.1	tkinter	254
16.1.2	其他 GUI 库简介	254
16.2	tkinter 概述	255
16.2.1	tkinter 模块	255
16.2.2	图形用户界面构成	255
16.2.3	框架和 GUI 应用程序类	256
16.2.4	tkinter 主窗口	257
16.3	几何布局管理器	257
16.3.1	pack 几何布局管理器	258
16.3.2	grid 几何布局管理器	259
16.3.3	place 几何布局管理器	260
16.4	事件处理	261
16.4.1	事件类型	261
16.4.2	事件绑定	262
16.4.3	事件处理函数	263
16.5	组件概述	264
16.5.1	创建组件和设置属性	264

16.5.2	坐标系	265
16.5.3	大小单位 (width 和 height)	265
16.5.4	颜色 (background、foreground 等)	265
16.5.5	字体 (font)	266
16.5.6	停靠位置 (anchor)	266
16.5.7	光标 (cursor)	267
16.5.8	显示文本 (text、textvariable、wraplength 和 justify)	267
16.5.9	位图 (bitmap)	267
16.5.10	图像 (image)	268
16.5.11	同时显示文本和位图/图像 (compound)	268
16.5.12	3D 显示样式 (relief 和 overrelief)	269
16.5.13	设置组件的边框 (borderwidth)	269
16.5.14	设置组件的填充 (padx 和 pady)	269
16.5.15	组件的状态 (state)	269
16.5.16	启用和禁用输入法	269
16.5.17	设置字符下划线位置 (underline)	269
16.5.18	焦点 (highlightbackground、highlightcolor 等)	270
16.5.19	组件的数据绑定 (textvariable、variable 和 listvariable)	270
16.5.20	组件 id 和名称	270
16.5.21	组件的通用方法	270
16.6	常用组件	271
16.6.1	标签 Label	271
16.6.2	标签框架 LabelFrame	271
16.6.3	按钮 Button	272
16.6.4	消息 Message	273
16.6.5	单行文本框 Entry	274
16.6.6	多行文本框 Text	275
16.6.7	单选按钮 Radiobutton	277
16.6.8	复选框 Checkbutton	278
16.6.9	列表框 Listbox	280
16.6.10	选择项 OptionMenu	282
16.6.11	移动滑块 Scale	283
16.6.12	顶层窗口 Toplevel	285
16.6.13	ttk 子模块控件	286
16.7	对话框	286
16.7.1	通用消息对话框	286
16.7.2	文件对话框	288
16.7.3	颜色选择对话框	289
16.7.4	通用对话框应用举例	290

16.7.5 简单对话框	292
16.8 菜单和工具栏	293
16.8.1 菜单组件 Menu	293
16.8.2 创建主菜单	294
16.8.3 创建上下文菜单	295
16.8.4 菜单应用举例	296
16.9 图形绘制	299
16.9.1 Canvas 组件	299
16.9.2 Canvas 上的对象	299
16.9.3 绘制矩形	301
16.9.4 绘制选项	302
16.9.5 绘制椭圆	305
16.9.6 绘制圆弧	305
16.9.7 绘制线条	306
16.9.8 绘制多边形	307
16.9.9 绘制位图	307
16.9.10 绘制图像	308
16.9.11 绘制字符串	308
16.9.12 绘制组件	309
16.10 复习题	310
16.11 上机实践	311
第 17 章 数据库访问	313
17.1 数据库基础	313
17.1.1 数据库概念	313
17.1.2 关系数据库	313
17.2 Python 数据库访问模块	314
17.2.1 通用数据库访问模块	314
17.2.2 专用数据库访问模块	315
17.2.3 SQLite 数据库和 sqlite3 模块	315
17.3 使用 sqlite3 模块连接和操作 SQLite 数据库	316
17.3.1 访问数据库的典型步骤	316
17.3.2 创建数据库和表	318
17.3.3 数据库表的插入、更新和删除操作	318
17.3.4 数据库表的查询操作	319
17.4 复习题	319
17.5 上机实践	319
第 18 章 网络编程和通信	320
18.1 网络编程的基本概念	320
18.1.1 网络基础知识	320

18.1.2	TCP/IP 协议简介	321
18.1.3	IP 地址和域名	322
18.1.4	统一资源定位器 URL	322
18.2	基于 socket 模块的网络编程	323
18.2.1	socket 概述	323
18.2.2	创建 socket 对象	324
18.2.3	将服务器端 socket 绑定到指定地址上	325
18.2.4	服务器端 socket 开始侦听	325
18.2.5	连接和接收连接	326
18.2.6	发送和接收数据	326
18.2.7	简单 TCP 程序: Echo Server	326
18.2.8	简单 UDP 程序: Echo Server	327
18.2.9	UDP 程序: Quote Server	329
18.3	基于 urllib 模块的网络编程	330
18.3.1	打开和读取 URL 网络资源	330
18.3.2	创建 Request 对象	330
18.4	基于 http 模块的网络编程	331
18.5	基于 ftplib 模块的网络编程	331
18.5.1	创建 FTP 对象	331
18.5.2	创建 FTP_TLS 对象	333
18.6	基于 poplib 和 smtplib 模块的网络编程	333
18.6.1	使用 poplib 接收邮件	333
18.6.2	使用 smtplib 发送邮件	334
18.7	复习题	335
18.8	上机实践	336
第 19 章	系统管理	337
19.1	目录、文件和磁盘的基本操作	337
19.1.1	创建目录	337
19.1.2	临时目录和文件的创建	337
19.1.3	切换当前工作目录	338
19.1.4	目录内容列表	338
19.1.5	文件通配符和 glob.glob 函数	338
19.1.6	遍历目录和 os.walk 函数	338
19.1.7	判断文件/目录是否存在	339
19.1.8	测试文件类型	339
19.1.9	文件的日期及其大小	339
19.1.10	文件和目录的删除	340
19.1.11	文件和目录复制、重命名和移动	340
19.1.12	磁盘的基本操作	340

19.2	执行操作系统命令和运行其他程序	341
19.2.1	os.system 函数	341
19.2.2	os.popen 函数	341
19.2.3	subprocess 模块	341
19.3	获取终端的大小	342
19.4	文件压缩和解压缩	343
19.4.1	shutil 模块支持的压缩和解压缩格式	343
19.4.2	make_archive() 和文件压缩	343
19.4.3	unpack_archive() 函数和文件解压缩	343
19.5	configparser 模块和配置文件	344
19.5.1	INI 文件及 INI 文件格式	344
19.5.2	ConfigParser 对象和 INI 文件操作	344
19.6	复习题	346
19.7	上机实践	347
附录 A	参考答案	348
	参考文献	358

第 1 章 Python 概述

Python 语言是一种解释型、面向对象的计算机程序设计语言。Python 语言广泛用于计算机程序设计教学语言、系统管理编程脚本语言、科学计算等，特别适用于快速的应用程序开发。

本章要点：

- ◆ Python 语言及其特点；
 - ◆ Python 语言版本和开发环境；
 - ◆ 下载和安装 Python；
 - ◆ 交互式解释执行 Python 程序；
 - ◆ 编写、编辑和执行 Python 程序。
-

1.1 Python 语言概述

1.1.1 Python 语言简介

Python（英音/ 'paiθən/，美音/ 'paiθɑ: n/）语言是一种解释型、面向对象的编程语言。由吉多·范罗苏姆（Guido van Rossum）于 1989 年底发明，被广泛应用于处理系统管理任务和科学计算。

Python 是一个开源语言，拥有大量的库，可以高效地开发各种应用程序。

1.1.2 Python 语言的特点

Python 语言具有下列特点。

- ① 简单。Python 是一种解释型的编程语言，遵循“优雅”、“明确”、“简单”的设计哲学，语法简单，易学、易读、易维护。
- ② 高级。Python 属于高级语言，无需考虑底层细节（如内存分配和释放等）。Python 还包括了内置的高级数据结构（例如：list 和 dict）。
- ③ 面向对象。Python 既支持面向过程的编程，也支持面向对象的编程。Python 支持继承、重载，有易于源代码的复用性。
- ④ 可扩展性。Python 提供了丰富的 API 和工具，以便程序员能够轻松地使用 C、C++ 语言来编写扩充模块。
- ⑤ 免费和开源。Python 是 FLOSS（自由/开放源码软件）之一，允许自由地发布此软件的拷贝、阅读和修改其源代码，或将其一部分用于新的自由软件中。

⑥ 可移植性。基于其开源本质，Python 已经被移植到许多平台上，包括：Linux/Unix、Windows、Macintosh 等。用户程序编写的 Python 程序，如果未使用依赖于系统的特性，无需修改就可以在任何支持 Python 的平台上运行。

⑦ 丰富的库。Python 语言提供了功能丰富的标准库，包括正则表达式、文档生成、单元测试、数据库、GUI（图形用户界面）等。还有许多其他高质量的库，如 Python 图像库等。

⑧ 可嵌入性。可以将 Python 嵌入到 C、C++ 程序中，从而为 C、C++ 程序提供脚本功能。

1.1.3 Python 语言的应用范围

Python 具有广泛的应用范围，常用的应用场景如下。

① 操作系统管理。Python 作为一种解释型的脚本语言，特别适合于编写操作系统管理脚本。Python 编写的系统管理脚本在可读性、性能、源代码重用度、扩展性等方面都优于普通的 shell 脚本。

② 科学计算。Python 程序员可以使用 NumPy、SciPy、Matplotlib 等模块编写科学计算程序。众多开源的科学计算软件包均提供了 Python 的调用接口，如著名的计算机视觉库 OpenCV、三维可视化库 VTK、医学图像处理库 ITK 等。

③ Web 应用。Python 经常被用于 Web 开发。例如，通过 mod_wsgi 模块，Apache 可以运行用 Python 编写的 Web 程序。

④ 图形用户界面（GUI）开发。Python 支持 GUI 开发，使用 Tkinter、wxPython 或者 PyQt 库，可以开发跨平台的桌面软件。

⑤ 其他。例如游戏开发，很多游戏使用 C++ 编写图形显示等高性能模块，而使用 Python 编写游戏的逻辑。

1.2 Python 语言的版本和开发环境

1.2.1 Python 语言的版本

Python 目前包含两个主要版本：Python 2 和 Python 3。

Python 2 于 2000 年 10 月发布。目前的最新版本为 Python 2.7。Python 2 实现完整的垃圾回收，并且支持 Unicode。目前存在大量使用 Python 2 开发的程序和库。

Python 3 于 2008 年 12 月发布。相对于 Python 的早期版本，Python 3 是一个较大的升级。Python 3 在设计时，为了不带入过多的累赘，没有考虑向下兼容。

例如，Python 3 中不支持 `print`，而使用新增的 `print()` 函数：

```
print('abc') # Python 3 正确, Python 2 错误
print 'abc'  # Python 3 错误, Python 2 正确
```

因此，许多针对早期 Python 版本设计的程序都无法在 Python 3 上正常运行。使用 Python 3，一般也不能直接调用 Python 2 开发的库，而必须使用相应的 Python 3 版本的库。

注：Python 3 的很多新特性后来也被移植到 Python 2.6/2.7。作为一个过渡版本，Python

2.6/2.7 基本使用 Python 2.x 的语法和库，也允许使用部分 Python 3 的语法与函数。如果程序可以在 Python 2.6/2.7 上正常运行，则可以通过一个名为 2to3 的转换工具（Python 自带的实用脚本）无缝迁移到 Python 3。

1.2.2 Python 语言的实现

Python 2 和 Python 3 规定相应版本 Python 的语法规则。实现 Python 语法的解释程序就是 Python 的解释器。

Python 解释器用于解释和执行 Python 语句和程序。常用的 Python 实现如下。

① CPython。使用 C 语言实现的 Python，即原始的 Python 实现。这是最常用的 Python 版本，也称之为 ClassicPython。通常 Python 就是指 CPython，需要区别的时候才使用 CPython。

② Jython。使用 Java 语言实现的 Python，原名 JPython。Jython 可以直接调用 Java 的类库，适用于 Java 平台的开发。

③ IronPython。面向 .NET 的 Python 实现。IronPython 能够直接调用 .NET 平台的类，适用于 .NET 平台的开发。

④ PyPy。使用 Python 语言实现的 Python。

1.2.3 Python 语言的集成开发环境

Python 是一门跨平台的脚本语言，在不同平台上提供了众多的集成开发环境（IDE），可以提高编程效率。常用的集成开发环境如下。

① IDLE。Python 内置的集成开发工具。

② PythonWin。适用于 Windows 环境的 Python 集成开发工具。

③ Eclipse + Pydev 插件。在通用集成开发环境 Eclipse 上安装 Pydev 插件，可以实现 Python 集成开发环境，方便调试程序。

④ Visual Studio + Python Tools for Visual Studio。在 Visual Studio 基础上安装 Python Tools for Visual Studio，可以使用功能完善的 Visual Studio 开发 Python 程序。

1.3 下载和安装 Python

1.3.1 下载 Python

Python 支持多平台，不同平台的安装和配置大致相同。本书基于 Windows 7 和 Python 3.3.2 构建 Python 开发平台。

【例 1-1】 下载 Python 安装程序。

操作步骤：

(1) 打开 Python 官网。打开浏览器（如 IE），在地址栏中输入：<http://Python.org/download>。按 Enter 键，打开 Python 官网，如图 1-1 所示。

(2) 下载 Python 安装程序。单击图 1-1 中所示的超链接，以下载最新的版本 Python 3.3.2 Windows x86 MSI Installer: python-3.3.2.msi (19.764M)。

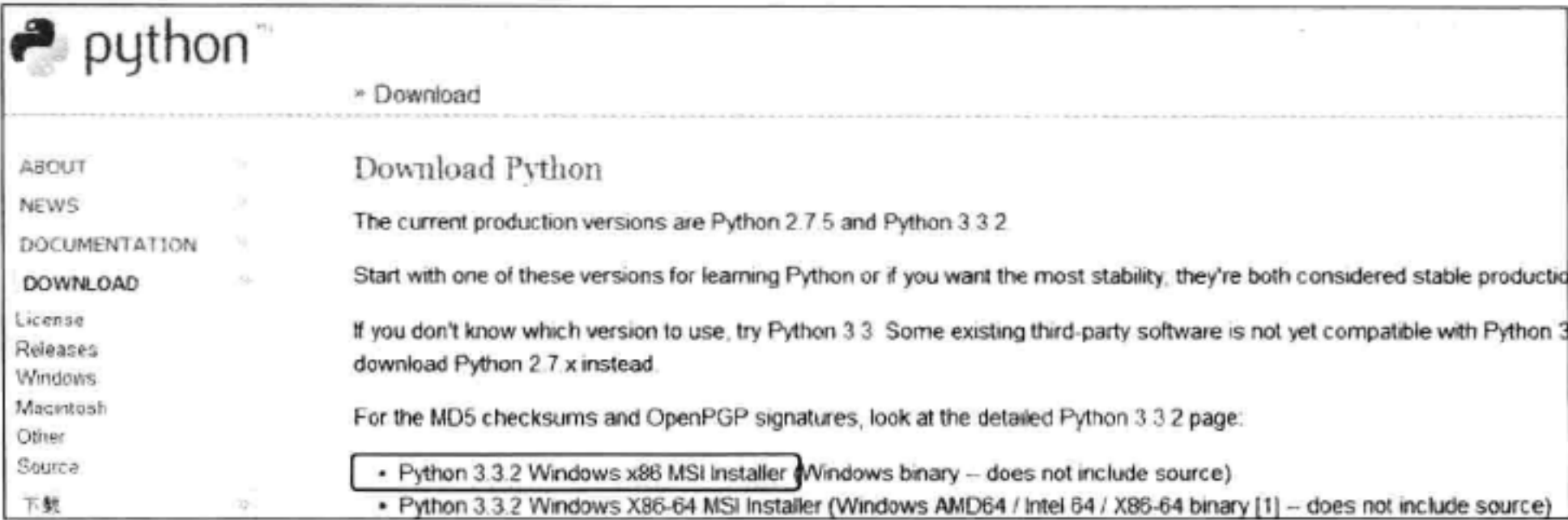


图 1-1 Python 官网

1.3.2 安装 Python

Python 的安装过程与其他的 Windows 安装程序类似。

【例 1-2】安装 Python 应用程序。

操作步骤：

- (1) 运行 Python 安装程序。双击下载的 Windows 格式安装文件：python - 3.3.2. msi，打开安装程序向导。
- (2) 设置安装选项。根据安装向导，安装 Python。默认安装路径为 C:\Python33\。在定制 Python 对话框中，单击 Add python. exe to Path，选择 Will be installed on local hard drive，以添加安装路径 C:\Python33\到系统路径，如图 1-2 所示。

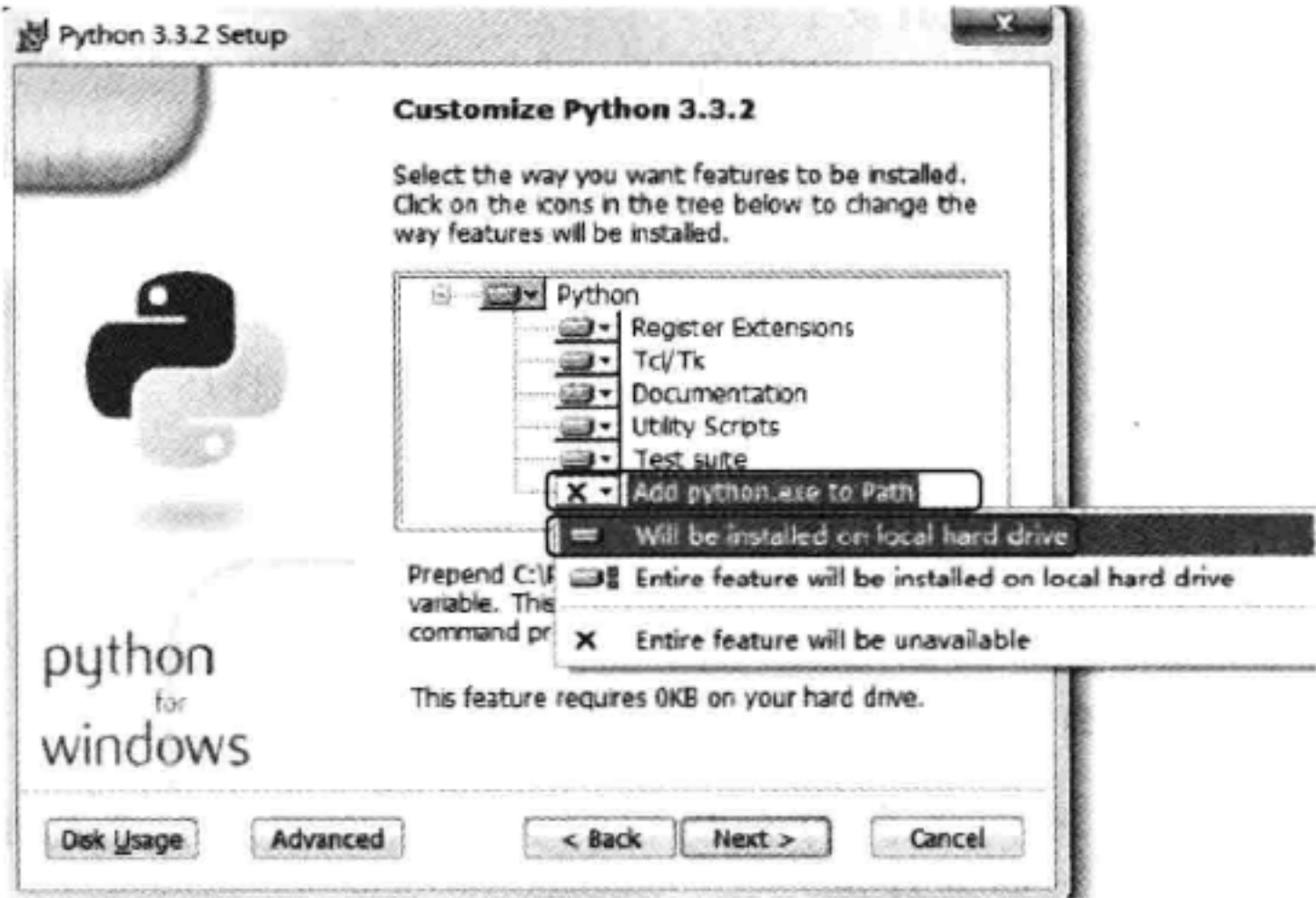


图 1-2 设置 Python 安装选项

重新启动 Windows 系统后，系统环境变量 path 中将包括 C:\Python33\。

注意，添加 C:\Python33\到系统路径后，在命令行窗口运行 python. exe 时，系统会自动寻找到 C:\Python33\python. exe。

- (3) 完成安装。完成安装，重新启动 Windows 系统。

1.4 使用 Python 解释器解释执行 Python 程序

1.4.1 运行 Python 解释器

Python 的默认安装路径为 C:\Python33\, 该目录下包括 Python 解释器 python.exe, 以及 Python 库目录和其他文件。

可以使用命令行, 也可以通过 Windows 开始菜单运行 python.exe。

【例 1-3】 运行 Python 解释器。

执行 Windows 菜单命令【开始】|【所有程序】| Python 3.3 | Python (command line), 打开 Python 解释器命令行窗口, 如图 1-3 所示。

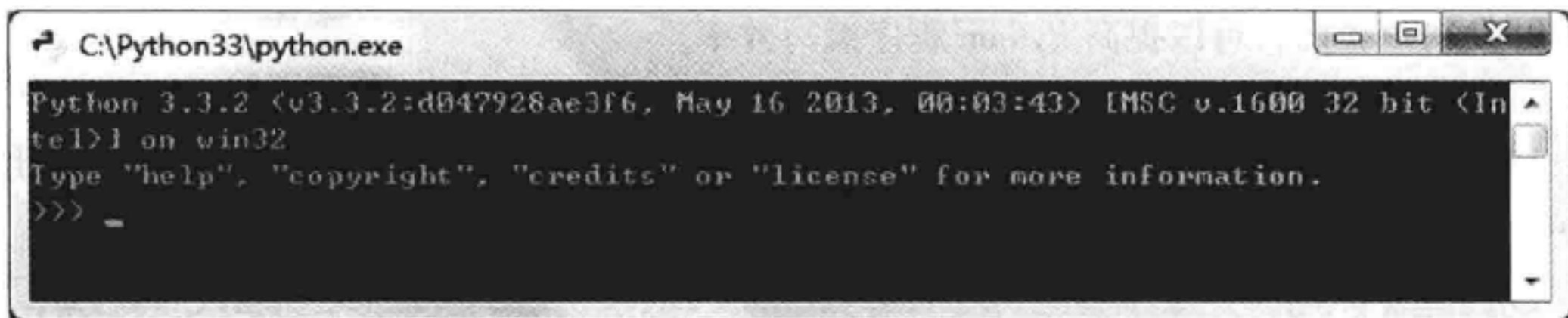


图 1-3 Python 解释器命令行窗口

【例 1-4】 输出 Hello world。

Python 解释器的提示符为: >>>。在提示符下输入语句, Python 解释器将解释执行, 并输出结果。例如, 输入: print('Hello, world!'), 则 Python 解释器将调用 print 函数, 打印输出字符串 Hello, world!, 如图 1-4 所示。

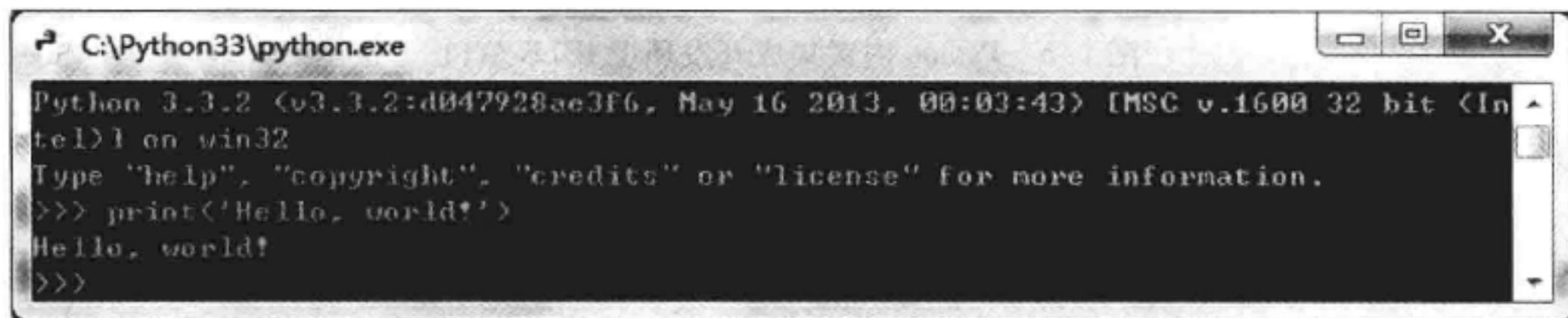


图 1-4 Python 解释器输出 Hello world

【例 1-5】 使用 Python 解释器进行数学运算。

在 Python 解释器的提示符下, 可以输入数学公式, Python 解释器将解析执行, 实现计算器的功能。例如: 11+22+33+44+55, 结果为 165; (1+0.05)¹⁰, 结果为 1.628894626777442。如图 1-5 所示。



图 1-5 使用 Python 解释器进行数学运算

【例 1-6】 使用解释器环境中特殊变量：_。

Python 解释器环境中，存在一个特殊变量：_，用于表示上一次运算的结果。例如：

```
>>> 11 + 22    #输出:33
>>> _          #输出:33
>>> _ + 33     #输出:66
```

【例 1-7】 关闭 Python 解释器。

通过 Ctrl + Z 及 Enter 键；或输入 quit() 命令；或直接关闭命令行窗口，关闭 Python 解释器。

1.4.2 运行 Python 集成开发环境

Python 内置集成开发环境 IDLE。相对于 Python 解释器命令行，集成开发环境 IDLE 提供图形开发用户界面，可以提高 Python 程序编写效率。

【例 1-8】 运行 Python 内置集成开发环境 IDLE。

执行 Windows 菜单命令【开始】|【所有程序】| Python 3.3 | IDLE (Python GUI)，打开 Python 内置集成开发环境 IDLE 窗口，如图 1-6 所示。

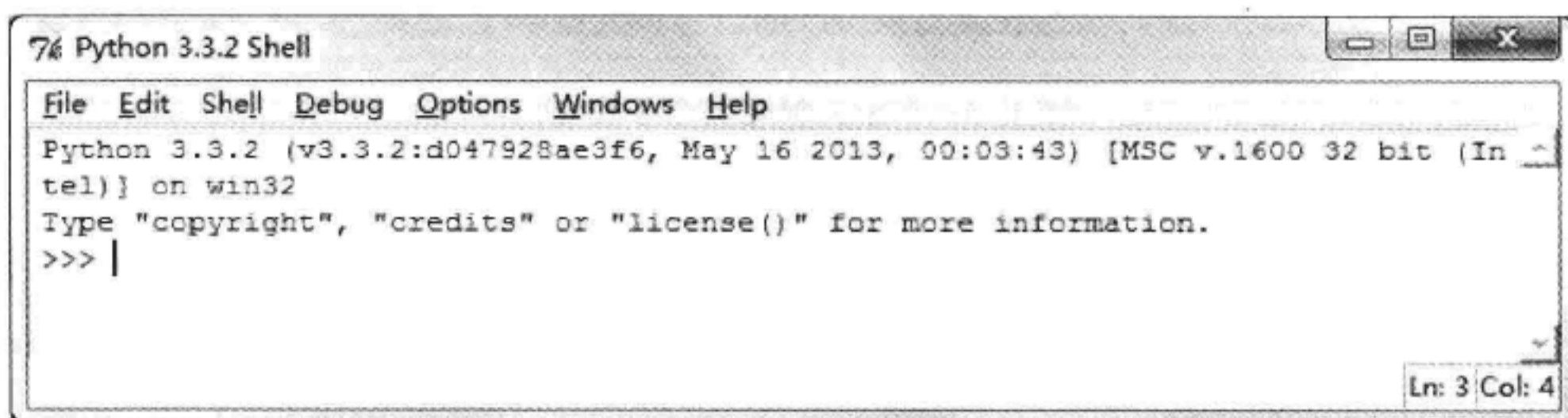


图 1-6 Python 内置集成开发环境 IDLE 窗口

【例 1-9】 使用集成开发环境 IDLE 解释执行 Python 语句。

在 Python 集成开发环境 IDLE 中输入 print('Good!' * 5)，则打印输出字符串 Good! Good! Good! Good!。注：'Good!' * 5 的结果为 5 个 'Good!' 的拼接，如图 1-7 所示。

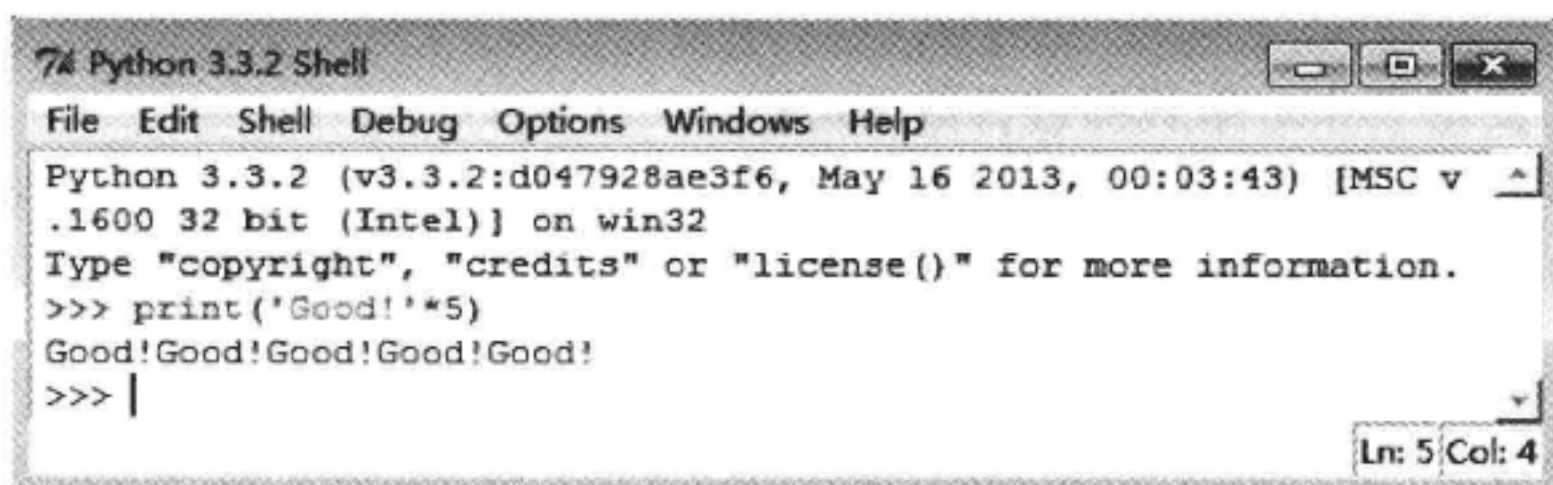


图 1-7 使用 IDLE 解释执行 Python 语句

【例 1-10】 使用 IDLE 执行多行代码。

复杂的 Python 语句包含多行代码。例如循环语句（打印 0 到 9 范围的数字，分隔符为空格）：

```
for x in range(10):  
    print(x, end='')
```

在 Python 解释器的提示符下，输入“for x in range(10):”后（注，冒号代表复合语句），按 Enter 键，Python 解释器在下一行自动缩进，等待输入；输入 print(x, end='') 后，按 Enter 键，Python 解释器在下一行等待输入（注：for 循环语句块可以包含多条语句）。直接按 Enter 键（本例中 for 循环语句块只包含一条语句），结束 for 循环语句，Python 解释器解释执行并输出结果，如图 1-8 所示。

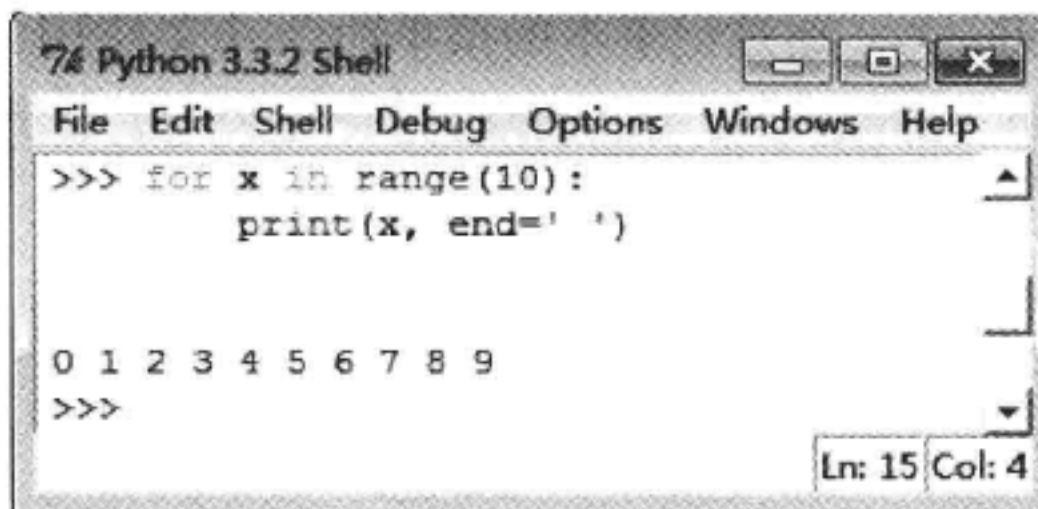


图 1-8 使用 Python 解释器执行多行代码

【例 1-11】关闭 IDLE。

通过 Ctrl + Z 及 Enter 键；或输入 quit()；或直接关闭 IDLE 窗口，均可关闭 Python 解释器。

1.5 使用文本编辑器和命令行编写和执行 Python 源文件程序

Python 解释器命令行采用交互方式执行 Python 语句，其优点是方便直接。但在交互式环境下，需要逐条输入语句，且执行的语句没有保存到文件中，因而不能重复执行，故不适合于复杂规模的程序设计。

可以把 Python 程序编写成一个文本文件，其后缀通常为 .py，然后，通过 Python 解释器编译执行。

1.5.1 编写 Hello world 程序

使用文本编辑软件（如 Windows 记事本 Notepad.exe），在 C:\Python\chapter01 目录下，创建程序文件 hello.py。

【例 1-12】使用文本编辑器（记事本）编写 Hello world 程序。

操作步骤：

（1）运行 Windows 记事本程序。执行 Windows 菜单命令【开始】|【所有程序】|【附件】|【记事本】，打开【记事本】文本编辑器。

（2）输入程序源代码。在记事本中，输入程序源代码，如图 1-9 所示。

（3）创建用于保存源文件的目录。打开资源管理器，在 C 盘中创建目录 Python；然后在 C:\Python 下创建目录 chapter01。注：本书正文源代码保存在 C:\Python 中的各章节子目

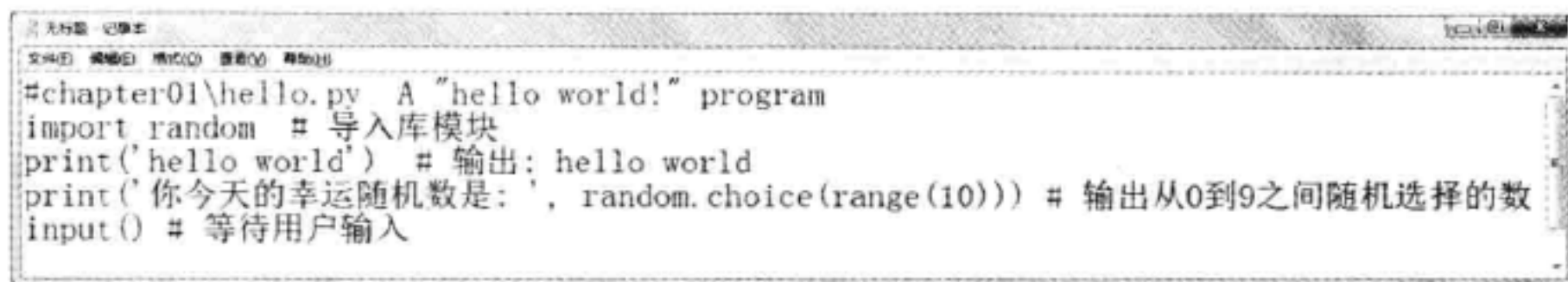


图 1-9 使用文本编辑器（记事本）编写 Hello world 程序

录下，例如，第 1 章的源代码保存在 C:\Python\chapter01 中，以此类推。

(4) 文件另存为：hello.py。通过记事本的菜单命令【文件】|【另存为】，将源程序文件 hello.py 保存到 C:\Python\chapter01 中。注意，【保存类型】选择【所有文件 (*.*)】，【编码】选择 UTF-8，如图 1-10 所示。



图 1-10 保存源程序文件到 C:\Python\chapter01\hello.py

1.5.2 Hello world 程序（hello.py）源代码分析

第 1 行为注释。Python 注释以符号#开始，到行尾结束。

第 2 行导入库模块 random。Python 可以导入和使用功能丰富的标准库或扩展库。

第 3 行调用内置库的函数 print，输出：hello world。

第 4 行使用 random 库中的 choice 函数，在 0~9 范围中随机选择一个数并输出。

第 5 行调用内置库的函数 input。用户按 Enter 键，程序结束运行。

1.5.3 运行 Python 源代码程序

在 Windows【命令提示符】窗口中，通过输入命令行命令：C:\Python33\python.exe C:\Python\chapter01\hello.py，直接调用 Python 解释器，执行程序 hello.py，并输出结果。

也可以在 Windows【命令提示符】窗口中，通过输入命令行命令：C:\Python\chapter01

hello.py，间接调用 Python 解释器，执行程序 hello.py，并输出结果。

注：安装 Python 后，Windows 关联后缀为 .py 的文件的默认打开程序为：Python Launcher for Windows (Console)。

【例 1-13】使用 Windows 命令提示符窗口运行 hello.py。

操作步骤：

(1) 打开【命令提示符】窗口。执行 Windows 菜单命令【开始】|【所有程序】|【附件】|【命令提示符】，打开【命令提示符】窗口，如图 1-11 所示。

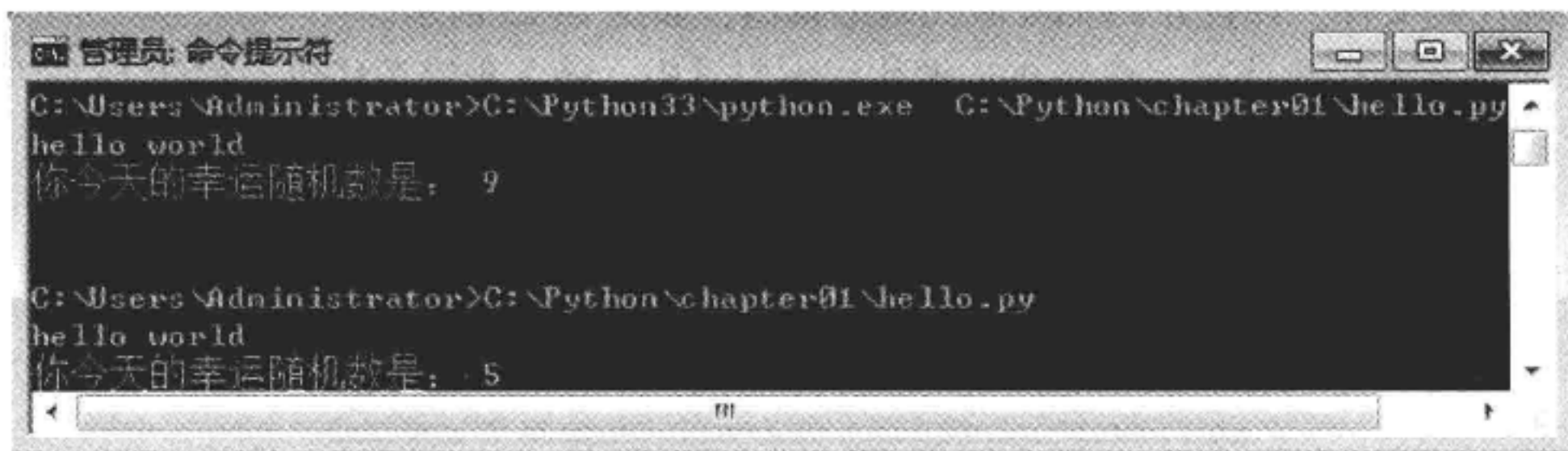


图 1-11 打开 Windows【命令提示符】窗口

(2) 直接调用 Python 解释器，执行程序 hello.py。输入命令行：

C:\Python33\python.exe C:\Python\chapter01\hello.py

按 Enter 键执行程序；然后再按 Enter 键结束程序执行。

(3) 间接调用 Python 解释器，执行程序 hello.py。输入命令行：C:\Python\chapter01\hello.py，按 Enter 键执行程序；然后再按 Enter 键结束程序执行。

【例 1-14】使用资源管理器运行 hello.py。

在资源管理器中，双击 C:\Python\chapter01 目录下的 hello.py 文件，Windows 自动调用其默认打开程序 Python Launcher for Windows (Console)，解释执行 hello.py 源程序，如图 1-12 所示。



图 1-12 使用资源管理器运行 hello.py

注：hello.py 文件最后包含一个语句 input()，用于等待用户输入，按 Enter 键后，程序结束运行，并关闭窗口。如果不包含该语句，则双击 hello.py，程序运行后，会自动关闭 Windows 命令行窗口，从而无法观察到程序运行的结果。

1.6 使用集成开发环境 IDLE 编写和执行 Python 源文件程序

集成开发环境 IDLE 提供了编写和执行 Python 源文件程序的图形界面，可以提高 Python

程序编写效率。

1.6.1 使用 IDLE 编写 Hello world 程序

【例 1-15】使用 IDLE 编写 Hello world 程序。

操作步骤：

- (1) 运行 Python 内置集成开发环境 IDLE（参照例 1-8）。
- (2) 新建源代码文件。执行 IDLE 菜单命令 File | New Window（快捷键 Ctrl + N），新建 Python 源代码文件，并打开 Python 源代码编辑器，如图 1-13 所示。
- (3) 输入程序源代码。在 Python 源代码编辑器中，输入程序源代码，如图 1-13 所示。

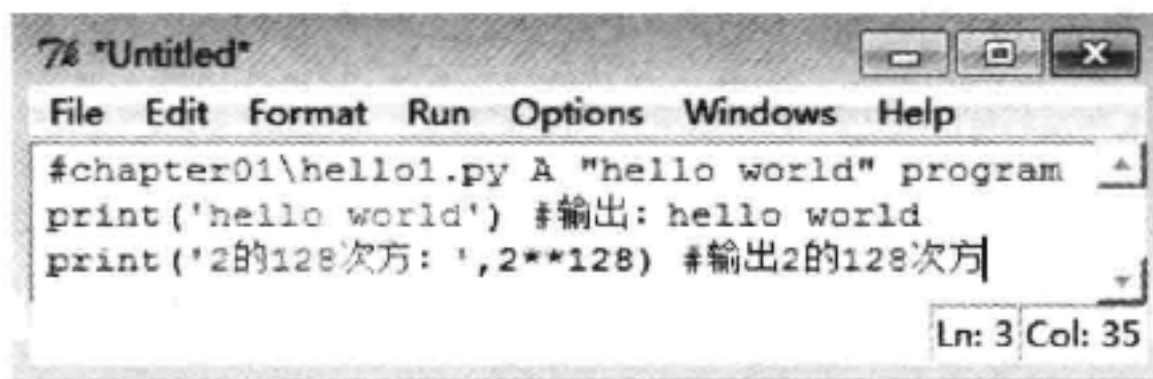


图 1-13 IDLE 源代码编辑器

- (4) 文件保存为 hello1.py。执行 IDLE 菜单命令 File | Save（快捷键 Ctrl + S），保存文件到位置 C:\Python\chapter01；文件名为 hello1.py。
- (5) 运行程序 hello1.py。执行 IDLE 菜单命令 Run | Run Module（快捷键 F5），打开 Python 3.3.2 Shell，输出运行结果，如图 1-14 所示。

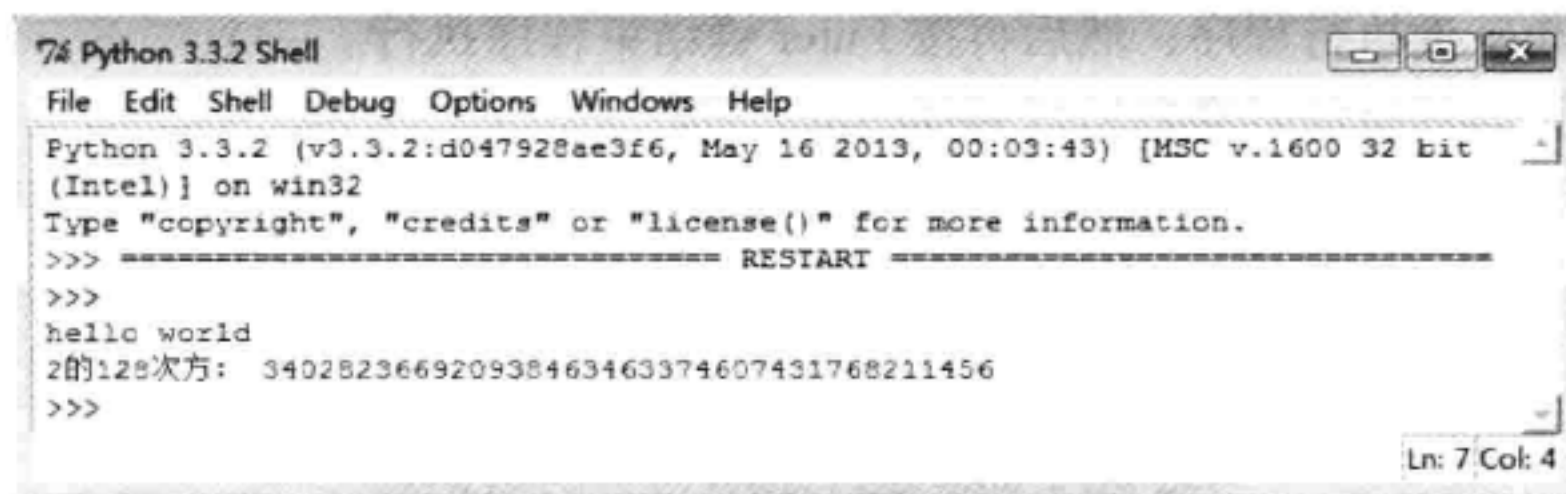


图 1-14 在 IDLE 环境中运行源代码程序

1.6.2 使用 IDLE 编辑程序

【例 1-16】使用 IDLE 编辑 hello.py 程序。

操作步骤：

- (1) 运行 Python 内置集成开发环境 IDLE。
- (2) 打开程序 hello.py。通过快捷键 Ctrl + O，在随后出现的【打开】窗口中，选择 C:\Python\chapter01\下的 hello.py，单击【打开】按钮，打开文件。
- (3) 编辑文件。在 Python 源代码编辑器中，编辑修改程序源代码，将输出 hello world 改为输出“Good Luck!”，如图 1-15 所示。
- (4) 保存文件 hello.py。通过快捷键 Ctrl + S，保存文件。

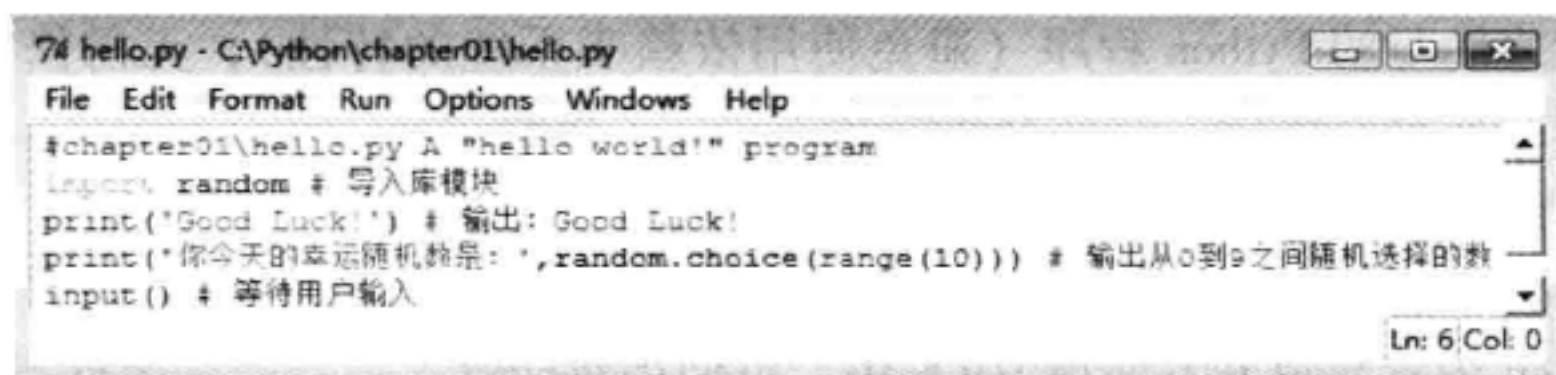


图 1-15 编辑 hello.py 程序

(5) 运行程序 hello.py。通过快捷键 F5，输出运行结果。

【例 1-17】 使用 IDLE 编辑 hello1.py 程序。

操作步骤：

(1) 打开程序 hello1.py。在资源管理器中，右击 C:\Python\chapter01 目录下的 hello1.py 文件，执行快捷菜单命令 Edit with IDLE，在 IDLE 中打开 hello1.py。

(2) 编辑文件。在 Python 源代码编辑器中，编辑修改程序源代码，利用 math.pow 函数求 2 的 128 次方，如图 1-16 所示。

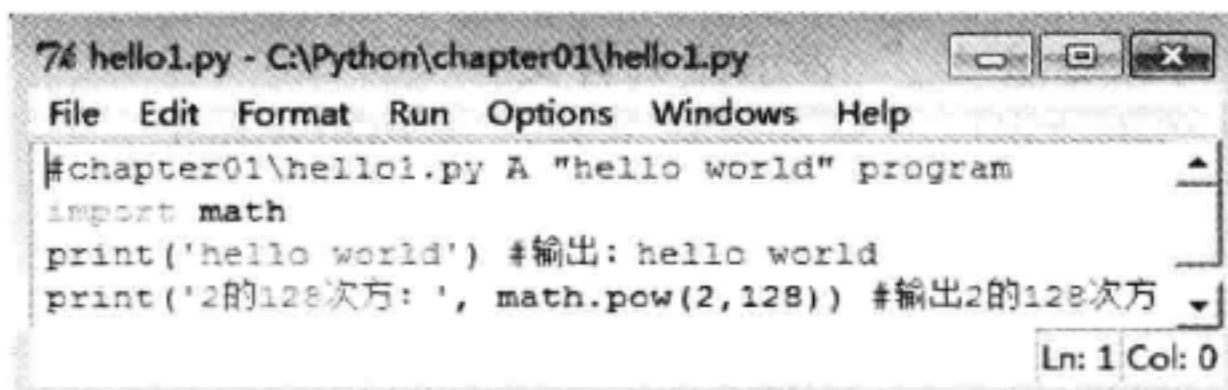


图 1-16 编辑 hello1.py 程序

(3) 保存文件 hello1.py。通过快捷键 Ctrl + S 保存文件。

(4) 运行程序 hello1.py。通过快捷键 F5 输出运行结果。注意，运行结果将以浮点数形式表示。

1.7 复习题

一、单选题

- Python 语言属于_____。
A. 机器语言 B. 汇编语言 C. 高级语言 D. 以上都不是
- 下列选项中，不属于 Python 特点的是_____。
A. 面向对象 B. 运行效率高 C. 可移植性 D. 免费和开源
- 下列选项中，_____实现是最常用的 Python 版本，也称之为 ClassicPython。
A. CPython B. Jython C. IronPython D. PyPy
- Python 内置的集成开发工具是_____。
A. PythonWin B. Pydev C. IDE D. IDLE

二、填空题

- Python 语言是一种解释型、面向_____的计算机程序设计语言。

2. 用户程序编写的 Python 程序（避免使用依赖于系统的特性），无需修改就可以在任何支持 Python 的平台上运行，这是 Python 的_____特性。
3. Python 的官方网址为_____。
4. 要关闭 Python 解释器，可使用函数_____或快捷键_____。
5. 在 Python 内置集成开发环境 IDLE 中，可使用快捷键_____，运行当前打开的源代码程序。

三、思考题

1. 简述 Python 语言的主要特点。
2. 简述 Python 语言的应用范围。
3. 简述 Python 2 和 Python 3 的主要区别。
4. Python 语言包括哪些实现？
5. Python 语言主要包括哪些集成开发环境？
6. 简述下载和安装 Python 的主要步骤。
7. 什么是 Python 解释器？如何使用其交互式测试 Python 代码？
8. Python 语言的特殊变量“_”表示什么含义？
9. 什么是 Python 源代码程序？如何运行 Python 源代码程序？
10. 如何使用 Python 内置集成开发环境 IDLE 编写和运行程序？
11. Python 是如何进行内存管理的？

1.8 上机实践

完成本章例 1-1 ~ 例 1-17，熟悉 Python 编辑、开发和运行环境。

第 2 章 Python 语言基础

语句是 Python 程序基本构成元素。语句通常包含表达式，而表达式由操作数和运算符构成。Python 语言可以定义函数和类。

本章要点：

- ◆ 语句、表达式和运算符；
 - ◆ 标识符及其命名规则；
 - ◆ 对象与引用；
 - ◆ 变量和赋值；
 - ◆ 数据类型；
 - ◆ 类和对象；
 - ◆ 函数、模块和包；
 - ◆ 文档注释。
-

2.1 语句

2.1.1 Python 语句

语句是 Python 程序的过程构造块，用于定义函数、定义类、创建对象、变量赋值、调用函数、控制分支、创建循环等。例如：

```
>>> aString = "张三"    #变量赋值
>>> print(aString)      #调用函数。输出:张三
```

Python 语句分为简单语句和复合语句。

简单语句包括：表达式语句、赋值语句、assert 语句、pass 空语句、del 语句、return 语句、yield 语句、raise 语句、break 语句、continue 语句、import 语句、global 语句、nonlocal 语句等。

复合语句包括：if 语句、while 语句、for 语句、try 语句、with 语句、函数定义、类定义等。

Python 语句涉及许多程序构造要素，将在本书后续章节陆续阐述。

2.1.2 Python 语句的书写规则

Python 语句书写规则如下。

- ① 使用换行符分隔，一般情况下，一行一条语句。

② 从第一列开始，前面不能有任何空格，否则会产生语法错误。注：注释语句可以从任意位置开始；复合语句构造体必须缩进。例如：

```
>>> #正确
>>> print(" abc") #输出:SyntaxError:unexpected indent
```

③ 反斜杠（\）用于一个代码跨越多行的情况。如果语句太长，可以使用续行符（\）。但三引号定义的字符串（"""..."""）、元组（（...））、列表（[...]）、字典（{...}），可以放在多行，而不用使用续行符（\），因为它们可清晰地表示定义的开始和结束。例如：

```
>>> print("如果语句太长,可以使用续行符(\),\
续行内容。")
```

④ 分号（;）用于在一行书写多条语句。例如：

```
>>> a=0;b=0;c=0
>>> s="abc";print(s) #输出:abc
```

2.1.3 复合语句及其缩进书写规则

由多行代码组成的语句称为复合语句。复合语句（条件语句、循环语句、函数定义和类定义，例如 if、for、while、def、class 等）由头部语句（header line）和构造体语句块（suites）组成。构造体语句块由一条或多条语句组成。复合语句和构造体语句块的缩进书写规则如下。

① 头部语句由相应的关键字（如：if）开始，构造体语句块则为下一行开始的一行或多行缩进代码。例如：

```
>>> sum = 0
>>> for i in range(1,11):
    sum = sum + i
    print(i,end=' ')
1 2 3 4 5 6 7 8 9 10
>>> print(sum) #输出:55
```

② 通常缩进是相对头部语句缩进 4 个空格，也可以是任意空格，但同一构造体代码块的多条语句缩进的空格数必须一致对齐。不缩进，或缩进不一致，将导致编译错误。注：Python 强制缩进，以保证源代码的规范性和可读性。另外，Python 不建议使用制表符缩进，因为制表符在不同系统产生的缩进效果可能不一致。

③ 如果条件语句、循环语句、函数定义和类定义比较短，可以放在同一行。例如：

```
>>> for i in range(1,11);print(i,end=' ')
```

2.1.4 注释语句

Python 注释语句以符号“#”开始，到行末结束。Python 注释语句可以出现在任何位置。Python 解释器将忽略所有的注释语句，注释语句不会影响程序的执行结果。良好的注释可以帮助程序的阅读和理解。例如：

```
>>> #A "hello world!" program
>>> #注释可以在任意位置,以#开始,到行末结束
>>> print("hello world") # 输出:hello world
```

Python 模块、类和函数可以定义规范的注释信息，以生成帮助文档。这将在后续章节阐述。

2.1.5 空语句 pass

要表示一个空的代码块，可以使用 pass 语句。例如：

```
>>> def do_nothing():  
    pass
```

2.2 表达式

2.2.1 表达式的组成

表达式是可以计算的代码片段，由操作数和运算符构成。操作数、运算符和圆括号按一定规则组成表达式。表达式通过运算后产生运算结果，运算结果的类型由操作数和运算符共同决定。

运算符指示对操作数适用什么样的运算。运算符的示例包括 +、-、*、/ 等。操作数包括文本常量（没有名称的常数值，如 1、"abc"）、变量（如 i = 123）、类的成员变量/函数（如 math.pi；math.sin(x)）等，也可以包含子表达式（如 (2 ** 10)）。

表达式既可以非常简单，也可以非常复杂。当表达式包含多个运算符时，运算符的优先级控制各运算符的计算顺序（参见 2.3.2 节）。例如，表达式 $x + y * z$ 按 $x + (y * z)$ 计算，因为 * 运算符的优先级高于 + 运算符。再如：

```
>>> import math  
>>> a = 2; b = 10  
>>> a + b           #输出:12  
>>> math.pi         #输出:3.141592653589793  
>>> math.sin(math.pi/2) #输出:1.0
```

2.2.2 表达式的书写规则

Python 表达式遵循下列书写规则。

- ① 表达式从左到右在同一个基准上书写。例如，数学公式 $a^2 + b^2$ 应该写为：a ** 2 + b ** 2。
- ② 乘号不能省略。例如，数学公式 ab （ a 乘以 b ）应写为：a * b。
- ③ 括号必须成对出现，而且只能使用圆括号；圆括号可以嵌套使用。

例如，数学表达式 $\frac{1}{2} \sin[a(x+1) + b]$ 写成 Python 表达式为：

```
math.sin(a * (x + 1) + b) / 2
```

2.3 运算符

2.3.1 运算符概述

Python 运算符用于在表达式中对一个或多个操作数进行计算并返回结果值。接受一个操作数的运算符被称作一元运算符，如正负号运算符 + 或 -。接受两个操作数的运算符被称作

二元运算符，如算术运算符 +、-、*、/等。

如果一个表达式中包含多个运算符，计算顺序则取决于运算符的结合顺序和优先级。

优先级高的运算符优先计算，例如，1 + 2 * 3 中，* 的优先级比 + 高，故先计算 2 * 3。同一优先级的运算符按结合顺序依次计算，例如 +、- 或者 *、/ 为同一优先级左结合的运算符，故 1 + 2 - 3 等同于 (1 + 2) - 3；2 * 4 / 2 等同于 (2 * 4) / 2。注：赋值运算符 = 为右结合运算符，故 a = b = c 等同于 a = (b = c)。可以使用()强制改变运算顺序。例如：

```
>>> 11 + 22 * 3      #输出:77
>>> (11 + 22) * 3    #输出:99
```

2.3.2 Python 运算符及其优先级

Python 语言定义了许多运算符，按优先顺序排列，如表 2-1 所示。本书后续章节也将陆续阐述。通过运算符重载（overload）（具体可参见 6.8 节）可以为用户自定义的类型定义新的运算符。

表 2-1 Python 运算符优先级

运 算 符	描 述
lambda	Lambda 表达式
or	布尔“或”
and	布尔“与”
not x	布尔“非”
in, not in	成员测试
is, is not	同一性测试
<, <=, >, >=, !=, ==	比较
	按位或
^	按位异或
&	按位与
<<, >>	移位
+, -	加法与减法
*, /, %, //	乘法、除法、取余、整数除法
+ x, - x	正负号
~ x	按位翻转
**	指数/幂
x. attribute	属性参考
x[index]	下标
x[index ; index]	寻址段
f(arguments, ...)	函数调用
(experection, ...)	绑定或元组显示
[expression, ...]	列表显示
{ key : datum, ... }	字典显示
'expression, ...'	字符串转换

2.4 标识符及其命名规则

2.4.1 标识符

标识符是变量、函数、类、模块和其他对象的名称。标识符的第一个字符必须是字母、下划线（“_”），其后的字符可以是字母、下划线或数字。一些特殊的名称，如 if、for 等，作为 Python 语言的保留关键字，不能作为标识符。

例如：a_int、a_float、str1、_strname、func1 为正确的变量名；而 99var、It'sOK、for（关键字）为错误的变量名。

注意：

- （1）Python 标识符区分大小写。例如，ABC 和 abc 视为不同的名称。
- （2）以双下划线开始和结束的名称通常具有特殊的含义。例如，__init__为类的构造函数。一般应避免使用。
- （3）避免使用 Python 预定义标识符名作为自定义标识符名。例如：NotImplemented、Ellipsis、int、float、list、str、tuple 等。

2.4.2 保留关键字

关键字即预定义保留标识符。关键字有特殊的语法含义。各关键字的使用，将在后续章节陆续阐述。关键字不能在程序中用作标识符，否则会产生编译错误。Python 3 的关键字如表 2-2 所示。

表 2-2 Python 3 的关键字

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

【例 2-1】使用 Python 帮助系统查看关键字。

- （1）运行 Python 内置集成开发环境 IDLE。
- （2）进入帮助系统。命令如下：
 >>> help()
（3）查看 Python 关键字列表，命令如下：
 help > keywords
（4）查看关键字 if 的帮助信息，命令如下：
 help > if
（5）退出帮助系统，命令如下：

help > quit

2.4.3 Python 预定义标识符

Python 语言包含许多预定义内置类、异常、函数等，如 float、ArithmeticError、print 等。用户应该避免使用 Python 预定义标识符名作为自定义标识符名。

使用 Python 的内置函数 dir(__builtins__)，查看所有内置的异常名、函数名等。

使用 <http://www.logilab.org/project/pylint> 上提供的 pylint 工具，可以检测 Python 源代码是否存在潜在的问题。

【例 2-2】查看 Python 内置内容列表。

(1) 运行 Python 内置集成开发环境 IDLE。

(2) 查看 Python 内置内容列表，命令如下：

```
>>> dir(__builtins__)
```

(3) 查看 float 的信息，命令如下：

```
>>> float
```

(4) 查看内置类 float 的帮助信息，命令如下：

```
>>> help(float)
```

2.4.4 命名规则

Python 语言一般遵循的命名规则如表 2-3 所示。

表 2-3 Python 命名规则

类 型	命 名 规 则	举 例
模块/包名	全小写字母，简单有意义，如果需要可以使用下划线	math、sys
函数名	全小写字母，可以使用下划线增加可阅读性	foo()，my_func()
变量名	全小写字母，可以使用下划线增加可阅读性	age、my_var
类名	采用 PascalCase 命名规则，即多个单词组成名称，每个单词除第一个字母大写外，其余的字母均小写	MyClass
常量名	全大写字母，可以使用下划线增加可阅读性	LEFT、TAX_RATE

2.5 对象与引用

Python 3 中，一切皆为对象。对象是某个类（类型）的实例，对象由唯一的 id 标识。对象可通过标识符来引用，对象引用即指向具体对象实例的标识符。对象也称为“变量”。

2.5.1 对象的类型（type）和标识（id）

通过内置的 type() 函数，可以判断一个对象的类型。通过内置的 id() 函数，可以获取一个对象有唯一的 id 标识（CPython 的实现为内存存放位置）。例如：

```
>>> type(25)          #输出: <class 'int' >
>>> id(25)            #输出:505911248
```

```
>>> type('123')    #输出: <class 'str'>
>>> id('123')      #输出: 16882368
```

Python 3 中，函数和类等也是对象，也具有相应的类型和 id。例如：

```
>>> type(abs)       #输出: <class 'builtin_function_or_method'>
>>> id(abs)         #输出: 11225368
>>> type(range)     #输出: <class 'type'>
>>> id(range)       #输出: 505673456
```

2.5.2 对象引用

通过标识符和赋值运算符（=），可以指定某个变量指向某个对象，即引用该对象。多个变量可以引用同一个对象；一个变量也可以改变指向其他的对象。例如：

```
>>> type(123)       #输出: <class 'int'>
>>> id(123)         #输出: 505912816
>>> a = 123
>>> id(a)           #输出: 505912816
>>> b = 123
>>> id(b)           #输出: 505912816
>>> c = a
>>> id(c)           #输出: 505912816
>>> id('abc')       #输出: 11898048
>>> a = 'abc'
>>> id(a)           #输出: 11898048
```

注：123 为类 int 的对象实例，其 id 为 505912816；a = 123，即变量 a 指向（引用）对象实例 123，故其 id 也为 505912816；b = 123，即变量 b 也指向（引用）对象实例 123，故其 id 亦为 505912816；c = a，变量 c 和 a 一样，指向（引用）对象实例 123，其 id 也同为 505912816。a = 'abc'，变量 a 指向（引用）对象实例 'abc'，其 id 为 11898048。

Python 3 中，作为对象的函数和类等也可以通过变量引用。但这样的引用一般意义不大，建议直接使用函数/类，以提高程序的可读性。例如：

```
>>> x = abs
>>> x(-123)         #输出: 123
>>> y = str
>>> id(y)           #输出: 505760232
>>> y.format('{0:.2f}', 123) #输出: '123.00'
```

2.5.3 对象比较（==）和类型判别（is）

通过 == 运算符可以判断两个变量指向的对象的值是否相同；通过 is 运算符可以判断两个变量是否指向同一对象。例如：

```
>>> x = 'abc'
>>> y = x
>>> z = 'abcd'
>>> x == y          #输出: True
```



```
>>> x is y      #输出:True
>>> x == z      #输出:False
>>> x is z      #输出:False
```

2.5.4 不可变对象 (immutable) 和可变对象 (mutable)

Python 3 对象可分为不可变对象 (immutable) 和可变对象 (mutable)。不可变对象一旦创建, 就不能被修改; 可变对象的内容可以被修改。

Python 大部分对象都是不可变对象, 例如: int、str、complex 等。变量是指向某个对象的引用, 多个变量可以指向同一个对象。给变量重新赋值, 并不改变原始对象的值, 只是创建一个新对象, 并指向它。例如:

```
>>> a = 18      #变量 a 指向 int 对象 18
>>> id(a)       #输出:505911136。表示 a 指向的 int 对象 18 的 id
>>> a = 25      #变量 a 指向 int 对象 25
>>> id(a)       #输出:505911248。表示 a 指向的 int 对象 25 的 id
>>> b = 25      #变量 b 指向 int 对象 25
>>> id(b)       #输出:505911248。表示 b 指向的 int 对象 25 的 id
>>> id(25)      #输出:505911248。表示 int 对象 25 的 id
```

对象本身值可以改变的对象称为可变对象 (如: list、dict 等)。例如:

```
>>> x = y = [1,2,3]  #变量 x 和 y 指向 list 对象[1,2,3]
>>> id(x)            #输出:16536960。表示变量 x 指向的 list 对象[1,2,3]的 id
>>> id(y)            #输出:16536960。表示变量 y 指向的 list 对象[1,2,3]的 id
>>> x.append(4)       #变量 x 指向的 list 对象[1,2,3]附加一个元素
>>> x                #输出:[1,2,3,4]。表示变量 x 指向的 list 对象
>>> id(x)            #输出:16536960。变量 x 指向的 list 对象[1,2,3,4]的 id 未改变
>>> x is y           #输出:True。表示变量 x 和 y 指向同一个 list 对象[1,2,3,4]
>>> x == y           #输出:True。表示变量 x 和 y 指向的 list 对象值相等
>>> z = [1,2,3,4]    #变量 z 指向的 list 对象[1,2,3,4]
>>> id(z)            #输出:16542456。表示变量 z 指向的 list 对象[1,2,3,4]的 id
>>> x is z           #输出:False。表示变量 x 和 z 指向不同的 list 对象[1,2,3,4]
>>> x == z           #输出:True。表示变量 x 和 z 指向的 list 对象值相等
```

2.6 变量和赋值

2.6.1 变量和数据类型

计算机程序处理的数据必须放入内存, Python 所有的数据都是对象, 每个对象都是某个类的实例, 即数据对象具有数据类型。

指向对象的引用即变量。Python 属于动态类型定义语言, 即变量不需要显式声明数据类型。根据变量的赋值, Python 解释器自动确定其数据类型。

事实上, 变量仅仅用于指向某个类型对象, 所以变量可以不限定类型, 即它可以指向任何类型的对象。例如:

```
>>>a=1      #1 为 int 类型,变量 a 指向 int 类型的对象 1
>>>b="11"    #"11" 为 str 类型,变量 b 指向 str 类型的对象"11"
```

Python 也是一种强类型语言，即每个变量指向的对象均属于某个数据类型。例如：

```
>>>a=1      #a 为 int 型
>>>b="11"    #b 为 str 型
>>>a+b       #int 型和 str 型不能直接相加,即 str 型不能自动转换为 int 型
Traceback(most recent call last):
  File "<pyshell#2>",line 1,in <module>
    a+b
TypeError:unsupported operand type(s) for + : 'int'and 'str'
>>>b=11      #赋值语句,b 为 int 型
>>>a+b       #输出:12
```

2.6.2 变量的声明和赋值

变量的声明和赋值格式如下：

变量名 = 要赋的值

Python 变量被访问之前必须被初始化，即赋值；否则会报错。例如：

```
>>>x=0;y=0;z=0
>>>str1="abc"
>>>aFloat    #NameError:name 'aFloat'is not defined
```

2.6.3 链式赋值语句

链式赋值（chained assignment）的语句形式如下：

变量 1 = 变量 2 = 表达式

等价于：

变量 2 = 表达式
变量 1 = 变量 2

链式赋值用于为多个变量赋值同一个值。例如：

```
>>>x=y=123
>>>x      #输出:123
>>>y      #输出:123
```

2.6.4 复合赋值语句

复合赋值运算符不仅可以简化程序代码，使程序精炼，还可以提高程序的效率。Python 中的复合赋值运算符如表 2-4 所示。运算符的含义请参见 4.6 节。

表 2-4 复合赋值运算符

运 算 符	含 义	举 例	等 效 于
+=	加法赋值 字符串拼接	sum += item aStr += " Foo"	sum = sum + item aStr = aStr + " Foo"
-=	减法赋值	count -= 1	count = count - 1

续表

运 算 符	含 义	举 例	等 效 于
<code>*</code>	乘法赋值	<code>x *= y + 5</code>	<code>x = x * (y + 5)</code>
<code>/</code>	除法赋值	<code>x /= y - z</code>	<code>x = x / (y - z)</code>
<code>//</code>	整除赋值	<code>x //= y - z</code>	<code>x = x // (y - z)</code>
<code>%</code>	取模赋值	<code>x %= 2</code>	<code>x = x % 2</code>
<code>**</code>	幂运算赋值	<code>x **= 2</code>	<code>x = x ** 2</code>
<code><<</code>	左移赋值	<code>x <<= y</code>	<code>x = x << y</code>
<code>>></code>	右移赋值	<code>x >>= y</code>	<code>x = x >> y</code>
<code>&</code>	按位与赋值	<code>x &= y</code>	<code>x = x & y</code>
<code> </code>	按位或赋值	<code>x = y</code>	<code>x = x y</code>
<code>^</code>	按位异或赋值	<code>x ^= y</code>	<code>x = x ^ y</code>

2.6.5 删除变量 (del)

可以使用 del 语句删除不再使用的变量。例如：

```
>>> x = 1
>>> del x
>>> x      #NameError:name 'x' is not defined
```

2.6.6 系列解包赋值

Python 支持组合数据类型（元组、列表、字典等），例如：

```
>>> tuple1 = 1,2,3
>>> tuple2 = (11,22,33)
>>> list1 = [111,222,333]
>>> type(tuple1);type(tuple2);type(list1)
<class 'tuple'>
<class 'tuple'>
<class 'list'>
```

上例中定义了元组 tuple1、tuple2 和列表 list1。有关组合数据类型，请参见第 6 章。

Python 支持把组合数据类型解包为对应相同个数的变量。例如：

```
>>> a,b = 1,2
>>> a      #输出:1
>>> b      #输出:2
>>> a,b = b,a
>>> a      #输出:2
>>> b      #输出:1
```

注意：

(1) 变量的个数必须与元组或列表元素个数一致，否则会产生错误，例如：`x, y = list1`，由于 list1 包含 3 个元素，故会产生错误。

(2) 使用 “a,b = b,a” 的方式, 可以优雅地实现两个变量值的交换。

2.7 数据类型

Python 语言中的每个对象都属于某个数据类型。Python 的数据类型包括内置的数据类型、模块中定义的数据类型和用户自定义的类型 (参见第 10 章)。数据对象可以使用运算符、内置函数、系统函数和对象所属类定义的方法进行运算操作。本节简单阐述内置的数据类型, 详细阐述请参见后续章节。

2.7.1 NoneType, NotImplementedType 和 Ellipsis

Python 包含三种特殊的数据类型: None, NotImplemented 和 Ellipsis。

1. NoneType

NoneType 数据类型包含唯一值 None, 主要用于表示空值, 如没有返回值的函数的结果。例如:

```
>>> None
>>> print(None)           #输出:None
>>> type(None), id(None)  #输出:( <class 'NoneType' >, 505672132)
```

2. NotImplementedType

NotImplementedType 数据类型包含唯一值 NotImplemented。数值运算和比较运算时, 如果对象不支持, 则可能返回该值。例如:

```
>>> NotImplemented      #输出:NotImplemented
>>> type(NotImplemented), id(NotImplemented)
(<class 'NotImplementedType' >, 505672340)
```

3. EllipsisType

EllipsisType 数据类型包含唯一值 Ellipsis, 表示省略字符串符号: ...。例如:

```
>>> Ellipsis             #输出:Ellipsis
>>> type(Ellipsis), id(Ellipsis)  #输出:( <class 'ellipsis' >, 505677708)
```

2.7.2 数值数据类型

Python 包括四种内置的数值类型。

- (1) 整数类型 (int), 用于表示整数。例如: 123、1024、-982。
- (2) 布尔类型 (bool), 用于表示布尔逻辑值。例如: True、False。
- (3) 浮点类型 (float), 用于表示实数。例如: 3.14、-1.23、1.1E10、-3e-4。
- (4) 复数类型 (complex), 用于表示复数。例如: 3+4j、-2-4j、1.2+3.4j。

数值可以使用运算符 (四则运算 +、-、*、/ 以及幂运算 ** 等)、内置函数 (abs、round 等)、math/cmath 模块中的数学函数、int/float/complex/bool 类的方法。

【例 2-3】数值数据类型示例 (profit.py): 计算复利。

```
nb = float(input("请输入本金:"))      #输入本金并转换为浮点数
```

```
nr = float(input("请输入年利率:"))    #输入年利率并转换为浮点数
ny = int(input("请输入年份:"))         #输入年份并转换为整数
amount = nb * (1 + nr/100) ** ny
print("本金利率和为:", amount)
```

运行结果如下:

```
请输入本金:1000
请输入年利率:6
请输入年份:10
本金利率和为: 1790.8476965428547
```

2.7.3 序列数据类型

序列数据类型表示若干有序数据。Python 序列数据类型分为:不可变序列数据类型和可变序列数据类型。

不可变序列数据类型包括以下三种。

- ☞ 字符串 (str), 表示 Unicode 字符序列。例如:"hello"。
- ☞ 元组类型 (tuple), 表示任意类型数据的序列。例如:(1, 2, 3), (1,"2")。
- ☞ 字节序列 (bytes), 表示的字节(8位)序列数据。例如:b'abc'。

可变序列数据类型包括以下两种。

- ☞ 列表类型 (list), 表示可修改的任意类型数据的序列。例如:[1,"two"]。
- ☞ 字节数组 (bytearray), 表示可修改的字节(8位)数组。

Python 包括字符串类型 (str), 用于实现文本处理。Python 字符串可以用以下四种方式定义。

- ☞ 单引号 (' ')。例如:'单引号字符串中可包含"双引号"字符'。
- ☞ 双引号 (" ")。例如:"双引号字符串中可包含'单引号'字符"。
- ☞ 三单引号 (""" """)。例如: """\t 字符串以制表符开始, 以换行符结束\n"""。
- ☞ 三双引号 (""" """)。例如: """三引号字符串中可以跨行,
第二行内容."""。

字符串处理可以使用运算符 (+、*、in)、内置函数 (如 len、max、min 等) 和 str 类的方法 (如 replace、split 等)。

通过字符串格式化, 可以输出特定格式的字符串。Python 字符串格式化包括:

```
字符串 1.format(值1,值2,...)
str.format(格式字符串1,值1,值2,...)
format(值,格式字符串)
格式字符串%(值1,值2,...) #兼容 Python 2 的格式,不建议使用
```

例如:

```
>>>"学生人数|0|,平均成绩|1|".format(15,81.2)
'学生人数 15,平均成绩 81.2'
>>>str.format("学生人数|0|,平均成绩|1:2.2f|",15,81.2)
'学生人数 15,平均成绩 81.20'
>>>format(81.2,"0.5f")          #输出:'81.20000'
>>>"学生人数%4d,平均成绩%2.1f"%(15,81)  #输出:'学生人数 15,平均成绩 81.0'
```

【例 2-4】 字符串示例 (string.py): 格式化输出字符串堆积的三角形。

```
print("1".center(20))          #1 行 20 个字符,居中对齐
print(format("121","~20"))      #1 行 20 个字符,居中对齐
print(format("12321","~20"))    #1 行 20 个字符,居中对齐
print("1".rjust(20,"*"))        #1 行 20 个字符,右对齐,加*号
print(format("121","*>20"))     #1 行 20 个字符,右对齐,加*号
print(format("12321","*>20"))   #1 行 20 个字符,右对齐,加*号
```

运行结果如下:

```
      1
     121
    12321
*****1
*****121
*****12321
```

2.7.4 集合数据类型

集合数据类型表示若干数据的集合,数据项目没有顺序,且不重复。Python 集合数据类型包括以下两种。

✎ 集 (set), 可变对象。例如: {1,2,3}。

✎ 不可变集 (frozenset), 不可变对象。例如:

```
>>> frozenset('abc')          #输出:frozenset({'b','c','a'})
```

2.7.5 字典数据类型

用于表示键-值对的字典。Python 的内置字典数据类型为 dict。例如: {1:"one",2:"two"}。

2.7.6 其他数据类型

Python 中一切对象都有一个数据类型,模块、类、对象、函数都属于某种数据类型。这部分涉及 Python 语言本身的构造。本节进行简单阐述。

1. 可调用数据类型

Python 中一切可调用的函数、方法等,都属于可调用数据类型,包括:自定义函数、对象方法、生成器函数、内置函数、类和对象。

Python 为自定义函数创建一个可调用对象。可调用对象包含特殊系统属性: __doc__ (文档字符串)、__name__ (名称)、__qualname__、__module__、__defaults__、__code__、__globals__、__dict__、__closure__、__annotations__、__kwdefaults__。另外还可设置并访问其任意属性。注: 这些特殊系统属性名称前后为双下划线。例如:

```
>>> def fl():                #定义函数 fl
    pass
>>> type(fl),id(fl)         #获取 fl 的类型和 id:( <class 'function' >,17342944)
>>> fl.__name__              #获取 fl 的属性:'fl'
>>> fl.attr1 = 123           #设置(增加)fl 的属性
```



```
>>> fl.attr1          #获取 fl 的属性:123
```

2. 模块

模块类型是一个容器，是可以使用 `import` 语句加载的对象。模块对象包含特殊属性：`__doc__`（文档字符串）、`__name__`（模块名称）、`__file__`（文件名）、`__path__`（路径名）。例如：

```
>>> import math
>>> type(math)        #输出: <class 'module'>
```

3. 类

定义类会生成一个 `type` 类型的对象，包括特殊属性：`__doc__`（文档字符串）、`__name__`（类名称）、`__bases__`（基类）、`__dict__`（方法和变量字典）、`__module__`（模块名称）、`__abstractmethods__`（抽象方法）。例如：

```
>>> class Class1():
    pass
>>> type(Class1)        #输出: <class 'type'>
>>> Class1.__bases__    #输出: (<class 'object'>,)
```

4. 对象实例

创建一个类的对象实例，其类型就是该类，包括特殊属性：`__class__`（对象实例的类）、`__dict__`（保存实例数据的字典）。例如：

```
>>> c = complex(1,2)
>>> type(c)            #输出: <class 'complex'>
>>> c.__class__         #输出: <class 'complex'>
```

5. 解释器内置类型

Python 解释器包含内置类型。例如：代码对象（Code Objects）、框架对象（Frame Objects）、跟踪对象（Traceback Objects）、切片对象（Slice Objects）、静态方法对象（Static Method Objects）、类方法对象（Class Method Objects）。

2.8 类的声明和对象的创建与调用

2.8.1 类的声明

类和对象是面向对象编程的两个主要方面。类创建一个新类型，而对象是类的具体实例。类的声明格式如下：

```
class 类名:
    类体
```

2.8.2 对象的创建和调用

对象的创建和调用格式如下：

```
anObject = 类名(参数列表)
```

`anObject`. 对象方法 或 `anObject`. 对象属性

注意:

- (1) 类使用关键字 `class` 声明, 类名为有效的标识符。
- (2) 类体中可以定义属于类的属性、方法等。
- (3) 创建对象后, 可访问其属性、调用其方法。

【例 2-5】类和对象示例 (Person.py): 定义类 `Person`, 创建其对象, 并调用对象方法。

```
class Person:           #定义类 Person
    def sayHello(self):  #定义类 Person 的函数 sayHi
        print('Hello,how are you? ')
p = Person()            #创建对象
p.sayHello()            #调用对象的方法
```

运行结果如下:

```
Hello,how are you?
```

2.9 函数

Python 语言包括许多内置的函数, 如 `print`、`range` 等, 用户也可以自定义函数。函数是可以重复调用的代码块。使用函数, 可以有效地组织代码, 提高代码的重用率。

本节简要介绍函数的定义和调用, 有关函数的展开阐述, 请参见第 8 章。

2.9.1 函数的声明和调用

函数的声明格式如下:

```
def 函数名([形参列表]):
    函数体
```

函数的调用格式如下:

```
函数名([实参列表])
```

注意:

- (1) 函数使用关键字 `def` 声明, 函数名为有效的标识符, 形参列表为函数的参数。
- (2) 声明函数时, 可声明函数的参数, 即形式参数, 简称形参; 形参在函数定义的圆括号对内指定, 用逗号分隔。调用函数时, 需要提供函数需要的参数的值, 即实际参数, 简称实参。
- (3) 函数可以使用 `return` 返回值。无返回值的函数相当于其他编程语言中的过程。

【例 2-6】声明和调用函数示例 (sayHello.py)。

```
def sayHello():         #定义函数 sayHello
    print('Hello World! ') #函数体
    print('To be or not to be,this is a question! ') #函数体
sayHello()              #调用函数 sayHello
```

运行结果如下:

```
Hello World!
```

```
To be or not to be,this is a question!
```

【例 2-7】 声明和调用函数 `getValue(b,r,n)`，根据本金 b 、年利率 r 、年数 n ，计算最终收益 v 。提示： $v=b(1+r)^n$ 。

```
def getValue(b,r,n):          #定义函数 getValue
    v=b*((1+r)**n)
    return v
total=getValue(1000,0.05,5)   #调用函数 getValue
print(total)                  #打印结果
```

运行结果如下：
1276.2815625000003

2.9.2 内置函数

Python 语言包含若干常用的内置函数，可直接使用。Python 内置函数如表 2-5 所示。

表 2-5 内置函数

<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	
<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>	

2.9.3 模块函数和 import 语句

通过 `import` 语句，可以导入模块 `module`，然后使用 `module.function (arguments)` 的形式调用模块中的函数。例如：

```
>>> import math
>>> math.sin(2)      #输出:0.9092974268256817
```

也可以通过“`from...import...`”形式直接导入包中的常量、函数和类。或通过“`from...import *`”导入包中的所有元素。例如：

```
>>> from math import sin
>>> sin(2)           #输出:0.9092974268256817
```


2.9.4 输入和输出函数

使用 Python 内置的输入函数 `input` 和输出函数 `print`，可以使程序与用户进行交互。

`input` 函数的格式为：

```
input([prompt])
```

`input` 函数提示用户输入，并返回用户从控制台输入的内容（字符串）。

`print` 函数的格式为：

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

`print` 函数用于打印一行内容，即多个以分隔符（`sep`，默认为空格）的值（`value`，...，以逗号分隔的值），到指定文件流（`file`，默认为控制台 `sys.stdout`）。`flush` 指定是否强制写入到流。例如：

```
>>> print(1,2,3)           #输出:1 2 3
>>> print(1,2,3,sep=',')   #输出:1,2,3
```

【例 2-8】使用 Python 内置的输入和输出函数实现用户交互（`ioTest.py`）。

```
import datetime
sName = input("请输入您的姓名:")
birthyear = int(input("请输入您的出生年份:")) #把输入值通过 int 转换为整型
age = datetime.date.today().year - birthyear
print("您好! {0}。您{1}岁.".format(sName,age))
```

运行结果如下：

```
请输入您的姓名:张三
请输入您的出生年份:1990
您好! 张三。您 23 岁。
```

2.9.5 运行时提示输入密码

如果在程序运行时，需要提示用户输入密码，则可以使用模块 `getpass`，以保证用户输入的密码在控制台中不回显。`getpass` 包含以下两个函数：

```
getpass.getpass(prompt='Password:', stream=None) #提示用户输入密码并返回
getpass.getuser()                               #获取当前登录用户名
```

如果系统不支持不回显，则 `getpass` 将导致异常 `getpass.GetPassWarning`。

【例 2-9】运行时提示输入密码（`getpass1.py`）。

```
import getpass
def checkuser(user,passwd):
    if user == 'jianghong' and passwd == 'password': #实际运用中,需要与数据库中的账户信息比较
        return True
    else:
        return False
if __name__ == '__main__':
    #提示输入用户名和密码
    user = input('用户名:')
```

```
passwd = getpass.getpass('密码:')
if checkuser( user,passwd):
    print('登录成功')
else:
    print('登录失败')
```

运行结果如下：

```
用户名:jianghong
密码:
登录成功
```

2.10 模块和包

Python 语言中，包含 Python 代码的源文件（通常包含用户自定义的变量、函数和类）称为模块，其扩展名为：`.py`。功能相近的模块可以组织成包，包是模块的层次性组织结构。有关模块和包的展开阐述，请参见第 10 章。

Python 标准库和第三方库中提供了大量的模块，通过 `import` 语句，可以导入模块，并使用其定义的功能。导入和使用模块功能的基本形式如下：

```
import 模块名      #导入模块
模块名.函数名      #使用包含模块的全限定名称调用模块中的函数
模块名.变量名      #使用包含模块的全限定名称访问模块中的变量
```

【例 2-10】 模块和包示例（`module1.py`）：求解一元二次方程 $x^2 + 5x + 6 = 0$ 。

```
import math
a = 1;b = 5;c = 6
x1 = (-b + math.sqrt(b * b - 4 * a * c))/(2 * a)
x2 = (-b - math.sqrt(b * b - 4 * a * c))/(2 * a)
print('方程 x * x + 5 * x + 6 = 0 的解为:',x1,x2)
```

运行结果如下：

```
方程 x * x + 5 * x + 6 = 0 的解为: -2.0 -3.0
```

2.11 Python 文档注释

2.11.1 文档字符串

程序源代码中，可以在特定的地方添加描述性文字，以说明包、模块、函数、类、类方法的相关信息。

在函数的第一个逻辑行的字符串称为函数的文档字符串。函数的文档字符串用于提供有关函数的帮助信息。

文档字符串一般遵循下列惯例：文档字符串是一个多行字符串；首行以大写字母开始，句号结尾；第 2 行是空行；从第 3 行开始是详细的描述。

可以使用以下三种方法抽取函数的文档字符串帮助信息：

- (1) 使用内置函数: `help(函数名)`;
- (2) 使用函数的特殊属性: `函数名.__doc__`;
- (3) 第三方自动化工具也可以抽取文档字符串信息, 以形成帮助文档。例如:

```
>>> help(abs)
Help on built-in function abs in module builtins:
abs(...)
    abs(number) -> number
    Return the absolute value of the argument.
>>> print(abs.__doc__)
abs(number) -> number
Return the absolute value of the argument.
```

同样, 在包的 `__init__.py` 中的注释, 称为包的文档字符串; 在文件头部注释, 称为模块的文档字符串; 在 `class` 声明后第一个逻辑行的注释, 称为类文档字符串。

【例 2-11】 文档字符串示例 (`doc.py`)。

```
"""doc 模块说明文档"""          #模块注释
def d2b(i):
    """函数 d2b 的说明文档"""    #模块注释
    print(bin(i))
class Doc:                        #定义类 Person
    """类 Doc 的说明文档"""
    def sayHello(self):           #定义类 Person 的函数 sayHi
        """方法 sayHello 的说明文档"""
        print('hi')
```

运行过程和结果如下:

```
C:\Python\chapter02 > python
>>> import doc
>>> doc.__doc__
'doc 模块说明文档'
>>> doc.d2b.__doc__
'函数 d2b 的说明文档'
>>> doc.Doc.__doc__
'类 Doc 的说明文档'
>>> doc.Doc.sayHello.__doc__
'方法 sayHello 的说明文档'
```

2.11.2 文档注释规范

在正式项目开发中, 则应该使用标准的注释语法, 在源代码中, 为包、模块、函数、类、类方法添加规范的注释信息, 以阐述其使用方法, 并适当描述设计思路和算法。

符合标准规范的文档注释, 可以通过文档化工具, 自动生成文档。Python 常用的文档化工具包括 `epydoc` 和 `DoxyGen`。其中, `epydoc` 是纯 Python 实现, 因而与 Python 的结合非常自然, 稳定可扩展。`epydoc` 注释采用标签格式:

@标签:内容

epydoc 规范包含的常用标签如下。

✎ 文献信息: @ author:(作者)、@ license:(版权)、@ contact:(联系)。

✎ 状态信息: @ version:(版本如 \$ Id \$)、@ todo[ver]:(改进,可以指定针对的版本)。

✎ py 模块信息: @ var v:(模块变量 v 说明)、@ type v:(模块变量类型 v 说明)。

✎ py 函数信息: @ param p:(参数 p 说明)、@ type p:(参数 p 类型说明)、@ return:(返回值说明)、@ rtype:(返回值类型说明)。

✎ 提醒信息: @ note:(注解)、@ attention:(注意)、@ bug:(问题)、@ warning:(警告)。

✎ py 关联信息: @ see:(参考资料)。

✎ 特殊标签: U{ text < url > }:(URL)、L{ text < object > } :交叉引用。

【例 2-12】 epydoc 文档注释规范示例 (epydoc. py)。

```
'''
@ version: $Id $
@ author: U{ username < mailto:email_address > }
@ see: 资料
'''

import sys,os,other
class Templates( object ):
    '''
    类说明
    '''
    def __init__( self,param1 ):
        '''
        @ param param1 :注释内容。
        @ type param1 :介绍。
        @ return :返回简介。
        @ rtype v: 返回类型简介。
        '''
        self.param1 = param1
```

2.12 复习题

一、单选题

1. 在 Python 中,合法的标识符是_____。
A. _ B. 3C C. it's D. str
2. Python 表达式中,可以使用_____控制运算的优先顺序。
A. 圆括号() B. 方括号[] C. 花括号{} D. 尖括号<>
3. 下列 Python 语句中,非法的是_____。
A. x = y = 1 B. x = (y = 1) C. x,y = y,x D. x = 1;y = 1
4. 下列数据类型中,Python 不支持的是_____。

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

5. 以下 Python 注释代码, 不正确的是_____。

- A. # Python 注释代码
- B. #Python 注释代码 1 #Python 注释代码 2
- C. """Python 文档注释"""
- D. //Python 注释代码

6. 数学关系式 $2 < x \leq 10$ 表示成正确的 Python 表达式为_____。

- A. $2 < x \leq 10$
- B. $2 < x \text{ and } x \leq 10$
- C. $2 < x \ \&\& \ x \leq 10$
- D. $x > 2 \text{ or } x \leq 10$

7. 在 Python 中, 正确的赋值语句为_____。

- A. $x + y = 10$
- B. $x = 2y$
- C. $x = y = 30$
- D. $3y = x + 1$

8. 为了给整型变量 x、y、z 赋初值 10, 下面正确的 Python 赋值语句是_____。

- A. $xyz = 10$
- B. $x = 10 \ y = 10 \ z = 10$
- C. $x = y = z = 10$
- D. $x = 10, y = 10, z = 10$

9. 为了给整型变量 x、y、z 赋初值 5, 下面正确的 Python 赋值语句是_____。

- A. $x = 5; y = 5; z = 5$
- B. $xyz = 5$
- C. $x, y, z = 5$
- D. $x = 5, y = 5, z = 5$

10. 已知 $x = 2; y = 3$, 复合赋值语句 $x *= y + 5$ 执行后, x 变量中的值是_____。

- A. 11
- B. 16
- C. 13
- D. 26

11. 整型变量 x 中存放了一个两位数, 要将这个两位数的个位数字和十位数字交换位置, 例如, 13 变成 31, 正确的 Python 表达式是_____。

- A. $(x \% 10) * 10 + x // 10$
- B. $(x \% 10) // 10 + x // 10$
- C. $(x / 10) \% 10 + x // 10$
- D. $(x \% 10) * 10 + x \% 10$

12. 与数学表达式 $\frac{cd}{2ab}$ 对应的 Python 表达式中, 不正确的是_____。

- A. $c * d / (2 * a * b)$
- B. $c / 2 * d / a / b$
- C. $c * d / 2 * a * b$
- D. $c * d / 2 / a / b$

二、填空题

1. Python 语句分为_____和复合语句。

2. Python 使用_____划分语句块。

3. Python 中, 如果语句太长, 可以使用_____作为续行符。

4. Python 中, 在一行书写两条语句时, 语句间使用_____作为分隔符。

5. Python 使用符号_____标示注释。

6. 在 Python 中, 要表示一个空的代码块, 可以使用空语句_____。

7. 在 Python 解释器中, 使用函数_____可以进入帮助系统。

8. 在 Python 解释器的帮助系统中, 使用_____, 可以查看关键字列表。

9. 计算 $2^{32} - 1$ 的 Python 表达式可书写为_____。

10. 使用 Python 的内置函数_____可以查看所有内置的异常名、函数名等。

11. Python 语句 `print(1,2,3,sep='-',end='!')` 的结果是_____。

12. Python 表达式 $12/4 - 2 + 5 * 8/4 \% 5/2$ 的值为_____。
13. Python 大部分对象均为不可变对象, 例如: _____等。_____等则为可变对象。
14. Python 的内置字典数据类型为_____。
15. Python 提供了两个对象身份比较运算符_____和_____来测试两个变量是否指向同一个对象; 通过内置函数_____来测试对象的类型; 通过_____运算符判断两个变量指向的对象的值是否相同。
16. Python 语句 `a,b = 3,4;a,b = b,a;print(a,b)` 的结果是_____。

三、思考题

1. Python 语句的主要作用是什么? Python 主要包含哪些语句?
2. Python 中 `pass` 语句的作用是什么?
3. Python 中有哪几种注释方式?
4. Python 是如何进行类型转换的?
5. Python 语句的主要书写规则是什么?
6. Python 表达式遵循哪些主要的书写规则?
7. Python 包括哪 4 种内置的数值类型?
8. Python 包括哪些不可变序列数据类型? 哪些可变序列数据类型?
9. Python 字符串有哪 4 种定义方式?
10. Python 文档字符串一般遵循哪些惯例?
11. Python 可以使用哪些方法抽取函数的文档字符串帮助信息?
12. 假设有 `a = 10`, 写出下面表达式运算后 `a` 的值:
(1) `a += a` (2) `a -= 2` (3) `a *= 2 + 3`
(4) `a /= 2 + 3` (5) `a %= a - a%4` (6) `a //= a - 3`
13. 请使用各种方法判断字符变量 `c` 是否为字母字符 (不区分大小写字母)。
14. 请使用各种方法判断字符变量 `c` 是否为数字字符。
15. 请使用各种方法判断字符变量 `c` 是否为大写字母。
16. 请使用各种方法判断字符变量 `c` 是否为小写字母。
17. 当运行测试输入 6789 时, 写出下面 Python 程序的执行结果。

```
num = int(input("请输入一个整数:"))
while(num != 0):
    print(num%10,end=' ')
    num = num//10
```

18. 下列 Python 语句的输出结果是_____。

```
def f():pass
print(type(f()))
```

19. 下列 Python 语句的输出结果是_____。

```
x = y = [1,2];x.append(3)
print(x is y,x == y,end=' ')
z = [1,2,3]
```



```
print(x is z, x == z, y == z)
```

20. 下列 Python 语句的输出结果是_____。

```
print("数量|0|,单价|1|".format(100,285.6))
```

```
print(str.format("数量|0|,单价|1:3.2f|",100,285.6))
```

```
print("数量%4d,单价%3.3f"%(100,285.6))
```

21. 下列 Python 语句的输出结果是_____。

```
print("1".rjust(20," "))
```

```
print(format("121",">20"))
```

```
print(format("12321",">20"))
```

2.13 上机实践

1. 编写程序，输入本金、年利率和年份，计算复利（结果保留两位小数）。运行效果参见图 2-1。

2. 编写程序，格式化输出杨辉三角。杨辉三角即二项式定理的系数表，各元素满足如下条件：第一列及对角线上的元素均为 1；其余每个元素等于它上一行同一列元素与上一行前一列元素之和。运行效果参见图 2-2。

```
请输入本金：2000
请输入年利率：5.6
请输入年份：5
本金利率和为：2626.33
```

图 2-1 计算复利运行效果

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
```

图 2-2 杨辉三角运行效果

3. 编写程序，声明函数 `getValue(b,r,n)`，根据本金 b 、年利率 r 、年数 n ，计算最终收益 $v = b(1+r)^n$ 。编写测试代码，提示输入本金、年利率和年数，显示最终收益（保留两位小数）。

4. 编写程序，求解一元二次方程 $x^2 - 10x + 16 = 0$ 。运行效果参见图 2-3。

```
方程x* x -10*x + 16 = 0 的解为： 8.0 2.0
```

图 2-3 求解一元二次方程运行效果

5. 编写程序，提示输入姓名和出生年份，输出姓名和年龄。运行效果参见图 2-4。

```
请输入您的姓名：Mary
请输入您的出生年份：1996
您好！Mary。您18岁。
```

图 2-4 姓名和年龄运行效果

第3章 程序流程控制

Python 程序中语句执行的顺序包括四种基本控制结构：顺序结构、选择结构、循环结构和异常处理逻辑结构。

本章要点：

- ◆ bool 数据类型及相关运算；
- ◆ 顺序结构；
- ◆ 选择结构：if 语句；
- ◆ 循环结构：for 语句、while 语句；
- ◆ Python 异常处理机制；
- ◆ Python 断言处理。

3.1 bool 数据类型和相关运算符

3.1.1 bool 类型

Python 的 bool 数据类型用于逻辑运算。bool 数据类型包含两个值：True（真）或 False（假）。选择语句中的条件表达式，最后评价为 bool 值：True（真）或 False（假）。Python 评价方法为：如果表达式的结果为数值类型（0）、空字符串（""）、空元组（（））、空列表（[]）、空字典（{}），则其 bool 值为 False（假）；否则其 bool 值为 True（真），如 123、“abc”、（1,2）均为 True。

3.1.2 关系和测试运算符

关系和测试运算符是二元运算符。关系运算符用于将两个操作数的大小进行比较。若关系成立，则比较的结果为 True，否则为 False。原则上，关系比较运算符应该是两个相同类型的对象之间的比较。例如：123 >= 23。理论上，不同类型的对象也允许进行比较，但结果往往无意义。例如：123 > “ABC”。Python 语言的关系和测试运算符如表 3-1 所示。

表 3-1 关系和类型测试运算符

运 算 符	表 达 式	含 义	实 例	结 果
==	x == y	x 等于 y	"ABCDEF" == "ABCD"	False
!=	x != y	x 不等于 y	"ABCD" != "abcd"	True

续表

运 算 符	表 达 式	含 义	实 例	结 果
>	x > y	x 大于 y	" ABC " > " ABD "	False
>=	x >= y	x 大于等于 y	123 >= 23	True
<	x < y	x 小于 y	" ABC " < " 上海 "	True
<=	x <= y	x 小于等于 y	" 123 " <= " 23 "	True
is	x is y	x 和 y 是同一个对象	x = y = 1 ; x is y x = 1 ; y = 2 ; x is y	True False
is not	x is not y	x 和 y 不是同一个对象	x = 1 ; y = 2 ; x is not y	True
in	x in y	x 是 y 的成员 (y 是容器, 如元组)	1 in (1, 2, 3) " A " in " ABCDEF "	True True
not in	x not in y	x 不是 y 的成员 (y 是容器, 如元组)	1 not in (1, 2, 3)	False

注意：

- (1) 关系运算符的优先级相同。
- (2) 对于两个预定义的数值类型，关系运算符按照操作数的数值大小进行比较。
- (3) 对于字符串类型，关系运算符比较字符串的值，即按字符的 ASCII 码值从左到右一一比较：首先比较两个字符串的第一个字符，其 ASCII 码值大的字符串大，若第一个字符相等，则继续比较第二个字符，依此类推，直至出现不同的字符为止。
- (4) 对象的比较运算（==、!=、>、>=、<、<=）对应于对象特殊方法的实现：__eq__()、__ne__()、__gt__()、__ge__()、__lt__()、__le__()。

3.1.3 逻辑运算符

逻辑运算符，即布尔运算符。用于检测两个以上条件的情况，即多个 bool 值的逻辑运算，其结果为 bool 类型值。

逻辑运算符除逻辑非（not）是一元运算符，其余均为二元运算符，用于将操作数进行逻辑运算，结果为 True 或 False。表 3-2 按优先级从高到低的顺序列出了 Python 中的逻辑运算符。

表 3-2 逻辑运算符

运 算 符	含 义	说 明	优 先 级	实 例	结 果
not	逻辑非	当操作数为 False 时返回 True；当操作数为 True 时返回 False	1	not True not False	False True
and	逻辑与	两个操作数均为 True 时，结果才为 True，否则为 False	2	True and True True and False False and True False and False	True False False False
or	逻辑或	两个操作数中有一个为 True 时，结果即为 True，否则为 False	3	True or True True or False False or True False or False	True True True False

注意：

(1) Python 的任意表达式都可以评价为布尔逻辑值，故均可以参与逻辑运算。例如：

```
>>> not 0      #输出:True
>>> not 'a'    #输出:False
```

(2) $C = A \text{ or } B$ 。如果 A 不为 0 或不为空或为 True，则返回 A；否则返回 B。仅在必要时才计算第二个操作数，即如果 A 不为 0 或不为空或为 True，则不用计算 B。即“短路”计算。例如：

```
>>> 1 or 2      #输出:1
>>> 0 or 2      #输出:2
>>> False or True  #输出:True
>>> True or False  #输出:True
```

(3) $C = A \text{ and } B$ 。如果 A 为 0 或为空或为 False，则返回 A；否则返回 B。仅在必要时才计算第二个操作数，即如果 A 为 0 或为空或为 False，则不用计算 B。即“短路”计算。例如：

```
>>> 1 and 2     #输出:2
>>> 0 and 2     #输出:0
>>> False and 2  #输出:False
>>> True and 2   #输出:2
```

这种写法，常用于不确定 A 是否为空值时，把 B 作为候补来赋值给 C。

3.2 顺序结构

程序中语句执行的基本顺序按各语句出现位置的先后次序执行，称为顺序结构，参见图 3-1。先执行语句块 1，再执行语句块 2，最后执行语句块 3。三者是顺序执行关系。

【例 3-1】顺序结构示例 (area.py)：已知三角形三条边的边长（为简单起见，假设这三条边可以构成三角形），求三角形的面积。提示：三角形面积 = $\sqrt{p(p-a)(p-b)(p-c)}$ ，其中，a、b、c 是三角形三边的边长，p 是三角形周长的一半。

```
import math
a = float(input("请输入三角形的边长 a:"))
b = float(input("请输入三角形的边长 b:"))
c = float(input("请输入三角形的边长 c:"))
p = (a + b + c)/2      #三角形周长的一半
area = math.sqrt(p * (p - a) * (p - b) * (p - c));#三角形面积
print(str.format("三角形三边分别为:a = {0} ,b = {1} ,c = {2} ",a,b,c))
print(str.format("三角形的面积 = {0} ",area))
```

运行结果如下：

```
请输入三角形的边长 a:3
请输入三角形的边长 b:4
```

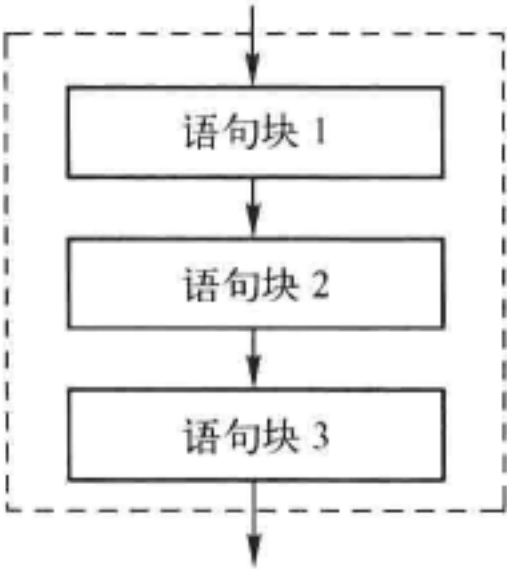


图 3-1 顺序结构示意图

请输入三角形的边长 c:5
三角形三边分别为:a = 3.0,b = 4.0,c = 5.0
三角形的面积 = 6.0

3.3 选择结构

选择结构可以根据条件来控制代码的执行分支，选择结构也叫分支结构。Python 使用 if 语句来实现分支结构。

3.3.1 分支结构的形式

分支结构包含多种形式：单分支、双分支和多分支，流程如图 3-2 所示。

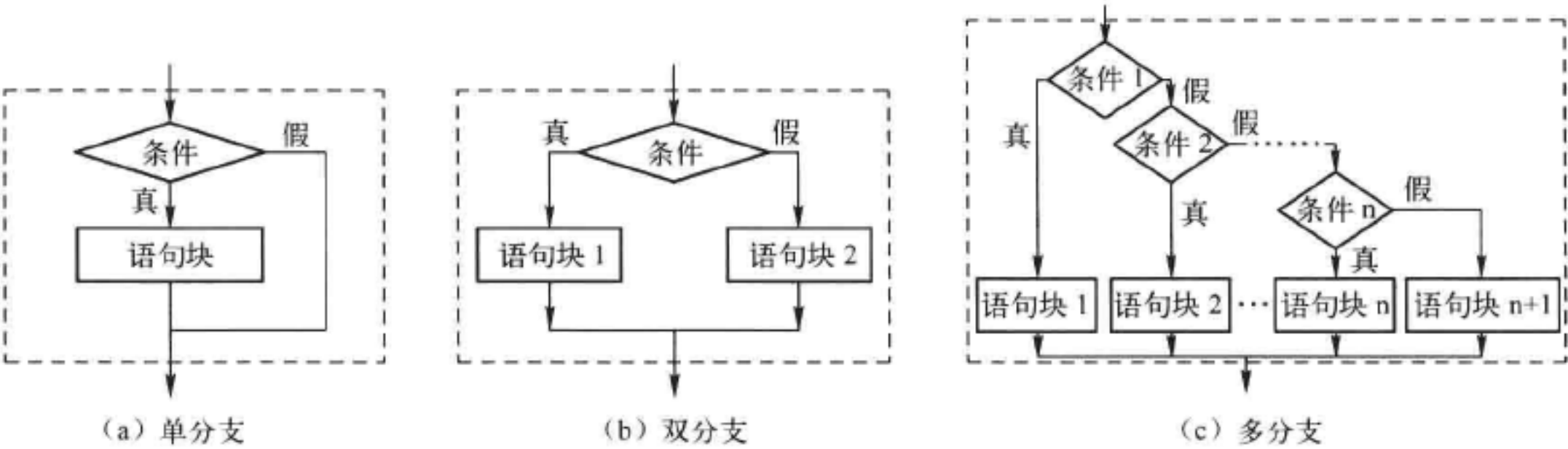


图 3-2 if 语句的选择结构

3.3.2 单分支结构

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：

- (1) 条件表达式可以是关系表达式、逻辑表达式、算术表达式等。
- (2) 语句/语句块可以是单个语句，也可以是多个语句。多个语句的缩进必须对齐一致。

当条件表达式的值为真（True）时，执行 if 后的语句（块），否则不做任何操作，控制将转到 if 语句的结束点。其流程如图 3-2（a）所示。

【例 3-2】单分支结构示例（if_2desc.py）：输入两个数 a 和 b，比较两者大小，使得 a 大于 b。

```
a = int(input("请输入第 1 个整数:"))  
b = int(input("请输入第 2 个整数:"))  
print(str.format("输入值:{0},{1}",a,b))  
if( a < b):  
    t = a  
    a = b
```

```
b = t
print(str.format("降序值:{0},{1}",a,b))
```

运行结果如下：

```
请输入第 1 个整数:23
请输入第 2 个整数:34
输入值:23,34
降序值:34,23
```

3.3.3 双分支结构

if 语句双分支结构的语法形式如下：

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

当条件表达式的值为真（True）时，执行 if 后的语句（块）1，否则执行 else 后的语句（块）2，其流程如图 3-2（b）所示。

Python 提供了下列条件表达式来实现等价于其他语言的三元条件运算符（（条件）？语句 1：语句 2）的功能：

条件为真时的值 if(条件表达式) else 条件为真时的值

例如，如果 $x \geq 0$ ，则 $y = x$ ，否则 $y = 0$ ，可以记述为：

```
y = x if(x >= 0) else 0
```

【例 3-3】计算分段函数：

$$y = \begin{cases} \sin x + 2\sqrt{x + e^4} - (x + 1)^3 & x \geq 0 \\ \ln(-5x) - \frac{|x^2 - 8x|}{7x} + e & x < 0 \end{cases}$$

此分段函数有以下几种实现方式，请读者自行编程测试。

（1）利用单分支结构实现。

一句单分支语句：

```
y = math.sin(x) + 2 * math.sqrt(x + math.exp(4)) - math.pow(x + 1,3)
if(x < 0):
    y = math.log(-5 * x) - math.fabs(x * x - 8 * x)/(7 * x) + math.e
```

或两句单分支语句：

```
if(x >= 0):
    y = math.sin(x) + 2 * math.sqrt(x + math.exp(4)) - math.pow(x + 1,3)
if(x < 0):
    y = math.log(-5 * x) - math.fabs(x * x - 8 * x)/(7 * x) + math.e
```

（2）利用双分支结构实现。

```
if(x >= 0):
    y = math.sin(x) + 2 * math.sqrt(x + math.exp(4)) - math.pow(x + 1,3)
else:
```



```
y = math.log(-5 * x) - math.fabs(x * x - 8 * x) / (7 * x) + math.e
```

(3) 利用条件运算语句实现。

```
y = (math.sin(x) + 2 * math.sqrt(x + math.exp(4)) - math.pow(x + 1, 3)) if (x >= 0) else \
(math.log(-5 * x) - math.fabs(x * x - 8 * x) / (7 * x) + math.e)
```

3.3.4 多分支结构

if 语句多分支结构的语法形式如下：

```
if( 条件表达式 1) :
    语句/语句块 1
elif( 条件表达式 2) :
    语句/语句块 2
...
elif( 条件表达式 n) :
    语句/语句块 n
[ else:
    语句/语句块 n + 1; ]
```

该语句的作用是根据不同条件表达式的值确定执行哪个语句（块），其流程如图 3-2（c）所示。

【例 3-4】已知某课程的百分制分数 mark，将其转换为五级制（优、良、中、及格和不及格）的评定等级 grade。评定条件如下：

$$\text{成绩等级} = \begin{cases} \text{优} & \text{mark} \geq 90 \\ \text{良} & 80 \leq \text{mark} < 90 \\ \text{中} & 70 \leq \text{mark} < 80 \\ \text{及格} & 60 \leq \text{mark} < 70 \\ \text{不及格} & \text{mark} < 60 \end{cases}$$

根据评定条件，有以下三种不同的方法实现。

方法一：

```
mark = int(input("请输入分数:"))
if (mark >= 90): grade = "优"
elif (mark >= 80): grade = "良"
elif (mark >= 70): grade = "中"
elif (mark >= 60): grade = "及格"
else: grade = "不及格"
```

方法二：

```
if (mark >= 90): grade = "优"
elif (mark >= 80 and mark < 90): grade = "良"
elif (mark >= 70 and mark < 80): grade = "中"
elif (mark >= 60 and mark < 70): grade = "及格"
else: grade = "不及格"
```

方法三：

```
if (mark >= 60): grade = "及格"
```

```
elif( mark >= 70 ):grade = "中"
elif( mark >= 80 ):grade = "良"
elif( mark >= 90 ):grade = "优"
else:grade = "不及格"
```

其中，方法一中使用关系运算符“>=”，按分数从大到小依次比较；方法二使用关系运算符和逻辑运算符表达完整的条件，即使语句顺序不按比较的分数从大到小依次书写，也可以得到正确的等级评定结果；方法三使用关系运算符“>=”，但按分数从小到大依次比较。

上述三种方法中，方法一、方法二正确，而且方法一最简捷明了，方法二虽然正确，但是存在冗余条件。方法三虽然语法没有错误，但是判断结果错误：根据 mark 分数所得等级评定结果只有“及格”和“不及格”两种，请读者根据程序流程自行分析原因。

【例 3-5】 已知坐标点(x,y)，判断其所在的象限 (if_coordinate.py)。

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

运行结果如下：

```
请输入 x 坐标:1
请输入 y 坐标:2
位于第一象限
```

3.3.5 if 语句的嵌套

在 if 语句中又包含一个或多个 if 语句的结构称为 if 语句的嵌套。一般形式如下：

```
if( 条件表达式 1 ):
    if( 条件表达式 11 ):
        语句 1
    [ else:
        语句 2 ]
[ else:
    if( 条件表达式 21 ):
        语句 3
    [ else:
        语句 4 ] ]
```

} 内嵌 if

} 内嵌 if

【例 3-6】 计算分段函数：

$$y = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

此分段函数有以下几种实现方式，请判断哪些是正确的？并自行编程测试正确的实现方式。

方法一（多分支结构）：

```
if(x > 0): y = 1
elif(x == 0): y = 0
else: y = -1
```

方法二（if 语句嵌套结构）：

```
if(x >= 0):
    if(x > 0): y = 1
    else: y = 0
else: y = -1
```

方法三：

```
y = 1
if(x != 0):
    if(x < 0): y = -1
else: y = 0
```

方法四：

```
y = 1
if(x != 0):
    if(x < 0): y = -1
else: y = 0
```

请读者画出每种方法相应的流程图，并进行分析测试。其中，方法1、2和3是正确的，而方法4是错误的。

3.3.6 选择结构综合举例

【例3-7】输入三个数，按从大到小的顺序排序（if_3desc.py）。

先用 a 和 b 比较，使得 $a > b$ ；然后比较 a 和 c ，使得 $a > c$ ，此时 a 最大；最后 b 和 c 比较，使得 $b > c$ 。

```
a = int(input("请输入整数 a:"))
b = int(input("请输入整数 b:"))
c = int(input("请输入整数 c:"))
if(a < b): t = a; a = b; b = t
if(a < c): t = a; a = c; c = t
if(b < c): t = b; b = c; c = t
print("排序结果(降序):", a, b, c)
```

运行结果如下：

```
请输入整数 a:3
请输入整数 b:2
请输入整数 c:5
排序结果(降序): 5 3 2
```

【例3-8】编程（leapyear.py）判断某一年是否为闰年。判断闰年的条件是：年份能被4整除但不能被100整除，或者能被400整除，其判断流程参见图3-3所示。

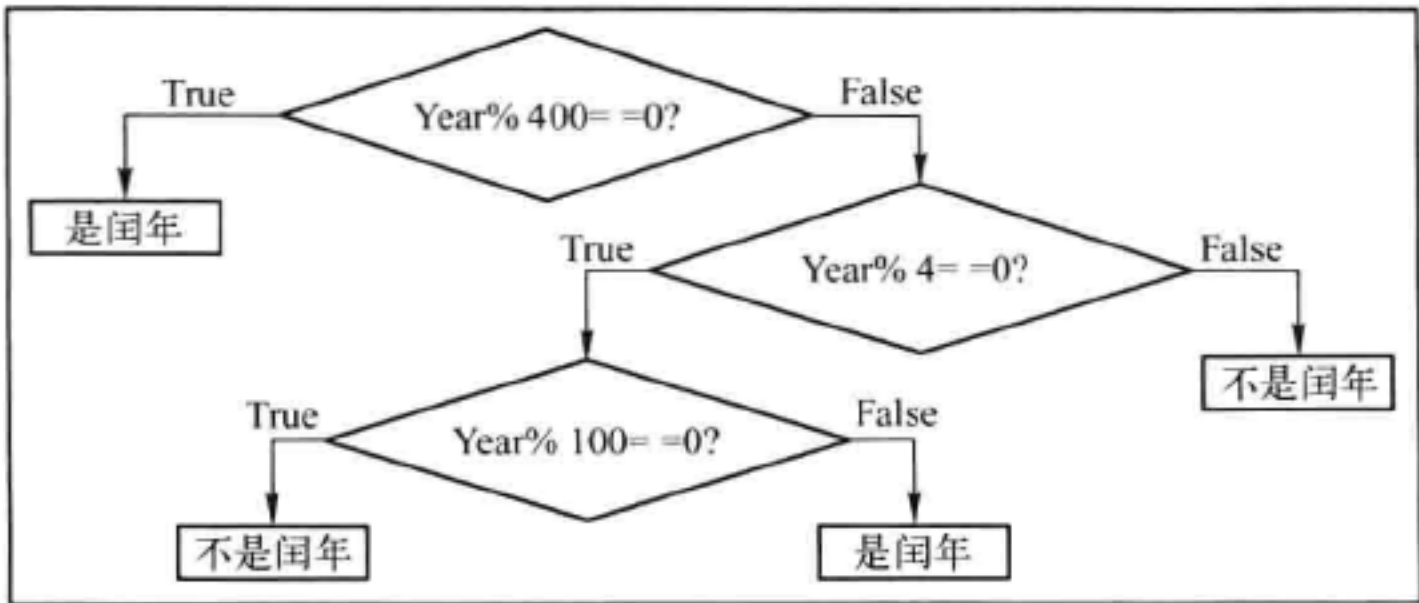


图 3-3 闰年的判断条件

方法一：使用一个逻辑表达式包含所有的闰年条件。相关语句如下：

```
if ( ( y%4 == 0 and y%100 != 0 ) or y%400 == 0 ) :
    print( " 是闰年" )
else:print( " 不是闰年" )
```

方法二：使用嵌套的 if 语句。相关语句如下：

```
if( y%400 == 0 ):print( " 是闰年" )
else:
    if( y%4 == 0 ) :
        if( y%100 == 0 ):print( " 不是闰年" )
        else:print( " 是闰年" )
    else:print( " 不是闰年" )
```

方法三：使用 if...elif 语句。相关语句如下：

```
if( y%400 == 0 ):print( " 是闰年" )
elif( y%4 != 0 ):print( " 不是闰年" )
elif( y%100 == 0 ):print( " 不是闰年" )
else:print( " 是闰年" )
```

方法四：使用 calendar 模块的 isleap 函数来判断闰年。相关语句如下：

```
if( calendar.isleap(y) ):print( " 是闰年" )
else:print( " 不是闰年" )
```

3.4 循环结构

循环结构用来重复执行一条或多条语句。使用循环结构，可以减少源程序重复书写的工作量。许多算法需要使用到循环结构。Python 使用 for 语句和 while 语句来实现循环结构。

3.4.1 可迭代对象（iterable）

可迭代对象一次返回一个元素，因而适用于循环。可迭代对象包括以下几种：

- ☞ 系列（sequence），如字符串（str）、列表（list）、元组（tuple）；
- ☞ 字典（dict）；
- ☞ 文件对象；
- ☞ 迭代器对象（iterator）；

生成器函数（generator）。

迭代器是一个对象，表示可迭代的数据集合，包括方法 `__iter__()` 和 `__next__()`，可实现迭代功能。请参见第 11 章。

生成器是一个函数，使用 `yield` 语句，每次产生一个值，也可用于循环迭代。请参见第 11 章。

3.4.2 for 循环

`for` 语句用于遍历可迭代对象集合中的元素，并对集合中的每个元素执行一次相关的嵌入语句。当集合中的所有元素完成迭代后，控制传递给 `for` 之后的下一个语句。`for` 语句的格式如下：

```
for 变量 in 对象集合:
    循环体语句/语句块
```

例如：

```
>>> for i in (1,2,3):
        print(i,i**2,i**3)
1 1 1
2 4 8
3 9 27
```

3.4.3 range 对象

Python 3 内置对象 `range` 是一个迭代器对象，迭代时产生指定范围的数字序列。其格式为：

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

`range` 返回的数值系列从 `start` 开始，到 `stop` 结束（不包含 `stop`）。如果指定了可选的步长 `step`，则序列按步长增长。例如：

```
>>> for i in range(1,11):print(i,end=' ')    #输出:1 2 3 4 5 6 7 8 9 10
>>> for i in range(1,11,3):print(i,end=' ')  #输出:1 4 7 10
```

注意：Python 2 中 `range` 的类型为函数，是生成器；Python 3 中 `range` 的类型为类，是一个迭代器。

【例 3-9】 利用 `for` 循环求 1 ~ 100 中所有奇数的和以及偶数的和（`for_sum1_100.py`）。

```
sum_odd = 0;sum_even = 0
for i in range(1,101):
    if i%2 !=0:    #奇数
        sum_odd += i
    else:          #偶数
        sum_even += i
print("1 ~ 100 中所有奇数的和:",sum_odd)
print("1 ~ 100 中所有偶数的和:",sum_even)
```

运行结果如下：

```
1 ~ 100 中所有奇数的和:2500
1 ~ 100 中所有偶数的和:2550
```

【例 3-10】显示 Fibonacci 数列 (for_fibonacci.py): 1, 1, 2, 3, 5, 8, ... 的前 20 项。

即

$$\begin{cases} F_1 = 1 & n = 1 \\ F_2 = 1 & n = 2 \\ F_n = F_{n-1} + F_{n-2} & n \geq 3 \end{cases}$$

要求每行显示 4 项。运行效果如图 3-4 所示。

1	1	2	3
5	8	13	21
34	55	89	144
233	377	610	987
1597	2584	4181	6765

图 3-4 显示 Fibonacci 数列

相关语句如下：

```
f1 = 1; f2 = 1
for i in range(1, 11):
    print( str.format( "{0:6} {1:6} ", f1, f2 ), end = " ") #每次输出 2 个数, 每个占 6 位
    if i%2 == 0: print() #显示 4 项后换行
    f1 += f2; f2 += f1
```

3.4.4 while 循环

与 for 循环一样, while 也是一个预测试的循环, 但是 while 在循环开始前, 并不知道重复执行循环语句序列的次数。while 语句按不同条件执行循环语句 (块) 零次或多次。while 循环语句的格式为:

while(条件表达式):
 循环体语句/语句块

while 循环的执行流程如图 3-5 所示。

说明:

(1) while 循环语句的执行过程如下。

① 计算条件表达式。

② 如果条件表达式结果为 True, 控制将转到循环语句 (块), 即进入循环体。当到达循环语句序列的结束点时, 转 (1), 即控制转到 while 语句的开始, 继续循环。

③ 如果条件表达式结果为 False, 退出 while 循环, 即控制转到 while 循环语句的后继语句。

(2) 条件表达式是每次进入循环之前进行判断的条件, 可以为关系表达式或逻辑表达式, 其运算结果为 True (真) 或 False (假)。条件表达式中必须包含控制循环的变量。

(3) 循环语句序列可以是一条语句, 也可以是多条语句。

(4) 循环语句序列中至少应包含改变循环条件的语句, 以使循环趋于结束, 避免“死循环”。

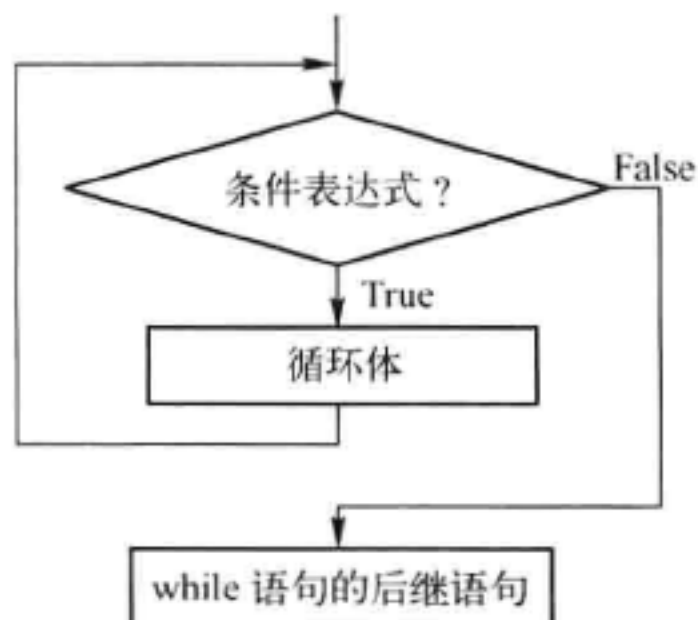


图 3-5 while 循环的执行流程

【例 3-11】利用 while 循环求 $\sum_{i=1}^{100} i$, 以及 1 ~ 100 中所有奇数的和、偶数的和 (while_sum.py)。


```
i = 1;sum_all = 0;sum_odd = 0;sum_even = 0
while(i <= 100):
    sum_all += i
    if(i%2 == 0):          #偶数
        sum_even += i
    else:                  #奇数
        sum_odd += i
    i += 1
print("和 = %d、奇数和 = %d、偶数和 = %d" % (sum_all,sum_odd,sum_even))
```

运行结果如下：

和 = 5050、奇数和 = 2500、偶数和 = 2550

【例 3-12】 用如下近似公式求自然对数的底数 e 的值，直到最后一项的绝对值小于 10^{-6} 为止（while_e.py）。

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}$$

```
i = 1;e = 1;t = 1
while(1/t >= pow(10, -6)):
    t *= i
    e += 1/t
    i += 1
print("e = ",e)
```

运行结果如下：

e = 2.7182818011463845

3.4.5 循环的嵌套

在一个循环体内又包含另一个完整的循环结构，成为循环的嵌套。这种语句结构称为多重循环结构。内层循环中还可以包含新的循环，形成多层循环结构。

在多层循环结构中，两种循环语句（for 循环、while 循环）可以相互嵌套。多重循环的循环次数等于每一重循环次数的乘积。

【例 3-13】 利用嵌套循环打印运行效果如图 3-6 所示的九九乘法表（nest_for.py）。

1*1=1	1*2=2	1*3=3	1*4=4	1*5=5	1*6=6	1*7=7	1*8=8	1*9=9
2*1=2	2*2=4	2*3=6	2*4=8	2*5=10	2*6=12	2*7=14	2*8=16	2*9=18
3*1=3	3*2=6	3*3=9	3*4=12	3*5=15	3*6=18	3*7=21	3*8=24	3*9=27
4*1=4	4*2=8	4*3=12	4*4=16	4*5=20	4*6=24	4*7=28	4*8=32	4*9=36
5*1=5	5*2=10	5*3=15	5*4=20	5*5=25	5*6=30	5*7=35	5*8=40	5*9=45
6*1=6	6*2=12	6*3=18	6*4=24	6*5=30	6*6=36	6*7=42	6*8=48	6*9=54
7*1=7	7*2=14	7*3=21	7*4=28	7*5=35	7*6=42	7*7=49	7*8=56	7*9=63
8*1=8	8*2=16	8*3=24	8*4=32	8*5=40	8*6=48	8*7=56	8*8=64	8*9=72
9*1=9	9*2=18	9*3=27	9*4=36	9*5=45	9*6=54	9*7=63	9*8=72	9*9=81

图 3-6 九九乘法表运行效果图

```
for i in range(1,10):
    s = ""
```

```
for j in range(1,10):
    s += str.format(" {0:1} * {1:1} = {2: <2} ",i,j,i*j)
print(s)
```

思考：请修改程序，分别打印如图 3-7（a）和图 3-7（b）所示的九九乘法表。

```
1*1=1
2*1=2 2*2=4
3*1=3 3*2=6 3*3=9
4*1=4 4*2=8 4*3=12 4*4=16
5*1=5 5*2=10 5*3=15 5*4=20 5*5=25
6*1=6 6*2=12 6*3=18 6*4=24 6*5=30 6*6=36
7*1=7 7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49
8*1=8 8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64
9*1=9 9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
```

（a）下三角

```
1*1=1 1*2=2 1*3=3 1*4=4 1*5=5 1*6=6 1*7=7 1*8=8 1*9=9
2*2=4 2*3=6 2*4=8 2*5=10 2*6=12 2*7=14 2*8=16 2*9=18
3*3=9 3*4=12 3*5=15 3*6=18 3*7=21 3*8=24 3*9=27
4*4=16 4*5=20 4*6=24 4*7=28 4*8=32 4*9=36
5*5=25 5*6=30 5*7=35 5*8=40 5*9=45
6*6=36 6*7=42 6*8=48 6*9=54
7*7=49 7*8=56 7*9=63
8*8=64 8*9=72
9*9=81
```

（b）上三角

图 3-7 九九乘法表

3.4.6 break 语句

break 语句在用于退出 for、while 循环，即提前结束循环，接着执行循环语句的后继语句。注意：当多个 for、while 语句彼此嵌套时，break 语句只应用于最里层的语句，即 break 语句只能跳出最近的一层循环。

【例 3-14】使用 break 语句终止循环（break.py）。

```
while True:
    s = input('请输入字符串(按 Q 或者 q 结束):')
    if s.upper() == 'Q':
        break
    print('字符串的长度为:',len(s))
```

运行结果如下：

```
请输入字符串(按 Q 或者 q 结束):Hello,World!
字符串的长度为:13
请输入字符串(按 Q 或者 q 结束):您好!
字符串的长度为:3
请输入字符串(按 Q 或者 q 结束):q
```

【例 3-15】编程（prime1.py 和 prime2.py）判断所输入的任意一个正整数是否为素数。

所谓素数（或称质数），是指除了 1 和该数本身，不能被任何整数整除的正整数。判断一个正整数 m 是否为素数，只要判断 m 可否被 $2 \sim \sqrt{m}$ 之中的任何一个整数整除，如果 m 不能被此范围中任何一个整数整除， m 即为素数，否则 m 为合数。

方法一（利用 for 循环和 break 语句）：

```
import math
m = int(input("请输入一个整数(>1):"))
k = int(math.sqrt(m))
for i in range(2,k+2):
    if m%i == 0:
        break #可以整除,肯定不是素数,结束循环
if i == k+1 :print(m,"是素数!")
else:print(m,"是合数!")
```

方法二（利用 while 循环和 bool 变量）：

```
import math
m = int(input("请输入一个整数(>1):"))
k = int(math.sqrt(m))
flag = True          #先假设所输整数为素数
i = 2
while(i <= k and flag == True):
    if(m%i == 0): flag = False #可以整除,肯定不是素数,结束循环
    else: i += 1
if(flag == True): print(m, "是素数!")
else: print(m, "是合数!")
```

3.4.7 continue 语句

continue 语句类似于 break 语句，也必须在 for、while 循环中使用。但它结束本次循环，即跳过循环体内自 continue 下面尚未执行的语句，返回到循环的起始处，并根据循环条件判断是否执行下一次循环。

continue 语句与 break 语句的区别在于：continue 语句仅结束本次循环，并返回到循环的起始处，循环条件满足的话就开始执行下一次循环；而 break 语句则是结束循环，跳转到循环的后继语句执行。

与 break 语句相类似，当多个 for、while 语句彼此嵌套时，continue 语句只应用于最里层的语句。

【例 3-16】 使用 continue 语句跳过循环（continue_score.py）。要求输入若干学生成绩（按 Q 或 q 结束），如果成绩小于 0，则重新输入。统计学生人数和平均成绩。

```
num = 0; scores = 0;          #初始化学生人数和成绩和
while True:
    s = input('请输入学生成绩(按 Q 或 q 结束):')
    if s.upper() == 'Q':
        break
    if float(s) < 0:           #成绩必须 >= 0
        continue
    num += 1                  #统计学生人数
    scores += float(s) #成绩和
print('学生人数为:{0}, 平均成绩为:{1}'.format(num, scores/num))
```

运行结果如下：

```
请输入学生成绩(按 Q 或 q 结束):65
请输入学生成绩(按 Q 或 q 结束):87
请输入学生成绩(按 Q 或 q 结束):-40
请输入学生成绩(按 Q 或 q 结束):q
学生人数为:2, 平均成绩为:76.0
```

【例 3-17】 显示 100 ~ 200 之间不能被 3 整除的数（continue_div3.py）。要求一行显示 10 个数。程序运行结果如图 3-8 所示。

100~200之间不能被3整除的数为:									
100	101	103	104	106	107	109	110	112	113
115	116	118	119	121	122	124	125	127	128
130	131	133	134	136	137	139	140	142	143
145	146	148	149	151	152	154	155	157	158
160	161	163	164	166	167	169	170	172	173
175	176	178	179	181	182	184	185	187	188
190	191	193	194	196	197	199	200		

图 3-8 例 3-17 运行结果

```
j=0          #控制一行显示的数值个数
print('100 ~ 200 之间不能被 3 整除的数为:')
for i in range(100,200 + 1):
    if(i%3 == 0):continue #跳过被 3 整除的数
    print( str. format( " {0: <5} ",i),end = " ")
    j += 1
    if(j%10 == 0):print( ) #一行显示 10 个数后换行
```

3.4.8 else 子句

for、while 语句可以附带一个 else 子句（可选）。如果 for、while 语句没有被 break 语句中止，则会执行 else 子句，否则不执行。其语法如下：

```
for 变量 in 对象集合:
    循环体语句(块)1
else:
    语句(块)2
```

或者：

```
while( 条件表达式):
    循环体语句(块)1
else:
    语句(块)2
```

【例 3-18】 使用 for 语句的 else 子句（for_else.py）。

```
hobbies = ""
for i in range(1,3 + 1):
    s = input('请输入爱好之一(最多三个,按 Q 或 q 结束):')
    if s.upper() == 'Q':
        break
    hobbies += s + ' '
else:
    print('您输入了三个爱好。')
print('您的爱好为:',hobbies)
```

运行结果如下：

```
>>>
请输入爱好之一(最多三个,按 Q 或 q 结束):旅游
请输入爱好之一(最多三个,按 Q 或 q 结束):音乐
请输入爱好之一(最多三个,按 Q 或 q 结束):运动
```

```
您输入了三个爱好。
您的爱好为： 旅游 音乐 运动
>>>
请输入爱好之一(最多三个,按 Q 或 q 结束):音乐
请输入爱好之一(最多三个,按 Q 或 q 结束):q
您的爱好为： 音乐
```

3.5 异常处理

3.5.1 错误和异常

Python 程序中可能出现的错误可以分为下列几种类型。

1. 编译（解析器）错误

即各种语法错误。对于编译错误，Python 解释器会直接抛出异常。可根据输出的异常信息修改程序代码。例如：

```
>>> if 1 > 2          #语法错误:SyntaxError
>>> Print('abc')      #拼写错误,引用了不存在的函数名称:name 'Print'is not defined
>>> x                  #引用了不存在的变量名称:name 'x'is not defined
```

2. 运行时错误

如打开不存在的文件、零除溢出等。对于运行时错误，Python 解释器也会抛出异常，代码中可以通过 try...except 语句捕获并处理（参见 3.5.2 节）。如果程序中没有 try...except，则 Python 解释器直接打印出异常信息。例如：

```
>>> f = open('abc.txt') #打开不存在的文件:FileNotFoundError
>>> 1/0                  #零除溢出:ZeroDivisionError
>>> 1 + 'abc'            #不同类型不能相加:TypeError
```

3. 逻辑错误

程序运行本身不报错，但结果不正确。对于逻辑错误，Python 解释器无能为力，需要读者根据结果来调试判断。例如，计算一元二次方程 $ax^2 + bx + c = 0$ 的两个根： $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ 。方程 $x^2 + 2x + 1 = 0$ 正确的解为： $x_1 = x_2 = -1$ 。

```
>>> a=1;b=2;c=1
>>> x1 = -b + math.sqrt(b*b-4*a*c)/2*a    #公式有误,故结果不正确
>>> x2 = -b - math.sqrt(b*b-4*a*c)/2*a    #公式有误,故结果不正确
>>> print(x1,x2)                          #输出:-2.0 -2.0
```

3.5.2 异常处理概述

Python 语言采用结构化的异常处理机制。在程序运行过程中，如果产生错误，则抛出异常；通过 try 语句来定义代码块，以运行可能抛出异常的代码；通过 except 语句，可以捕获特定的异常并执行相应的处理；通过 finally 语句，可以保证即使产生异常（处理失败），也

可以在事后清理资源等。例如，读取文件内容的伪代码一般如下：

```
def readile():
    打开文件           #可能产生错误:文件不存在
    读取文件内容       #可能产生错误:无读取权限
    关闭文件
```

使用 Python 的结构化异常处理机制，其伪代码一般如下：

```
def read_file():
    try:
        打开文件           #可能产生错误:文件不存在
        读取文件内容       #可能产生错误:无读取权限
        关闭文件
    except FileNotFoundError:  #捕获异常:无法打开文件
        #异常处理逻辑
    except PermissionError:  #捕获异常:无读取权限
        #异常处理逻辑
```

从上面伪代码可以看出，异常处理机制可以把错误处理和正常代码逻辑分开，从而可以更加高效地实现错误处理，增加程序的可维护性。

异常处理机制已经成为许多现代程序设计语言处理错误的标准模式。

3.5.3 内置的异常类

在程序运行过程中，如果出现错误，Python 解释器会创建一个异常对象，并抛出给系统运行时。即程序终止正常执行流程，转而执行异常处理流程。

在某种特殊条件下，代码中也可以创建一个异常对象，并通过 raise 语句，抛出给系统运行时。

异常对象是异常类的对象实例。Python 异常类均派生于 BaseException，Python 内置的异常类的层次结构如图 3-9 所示。

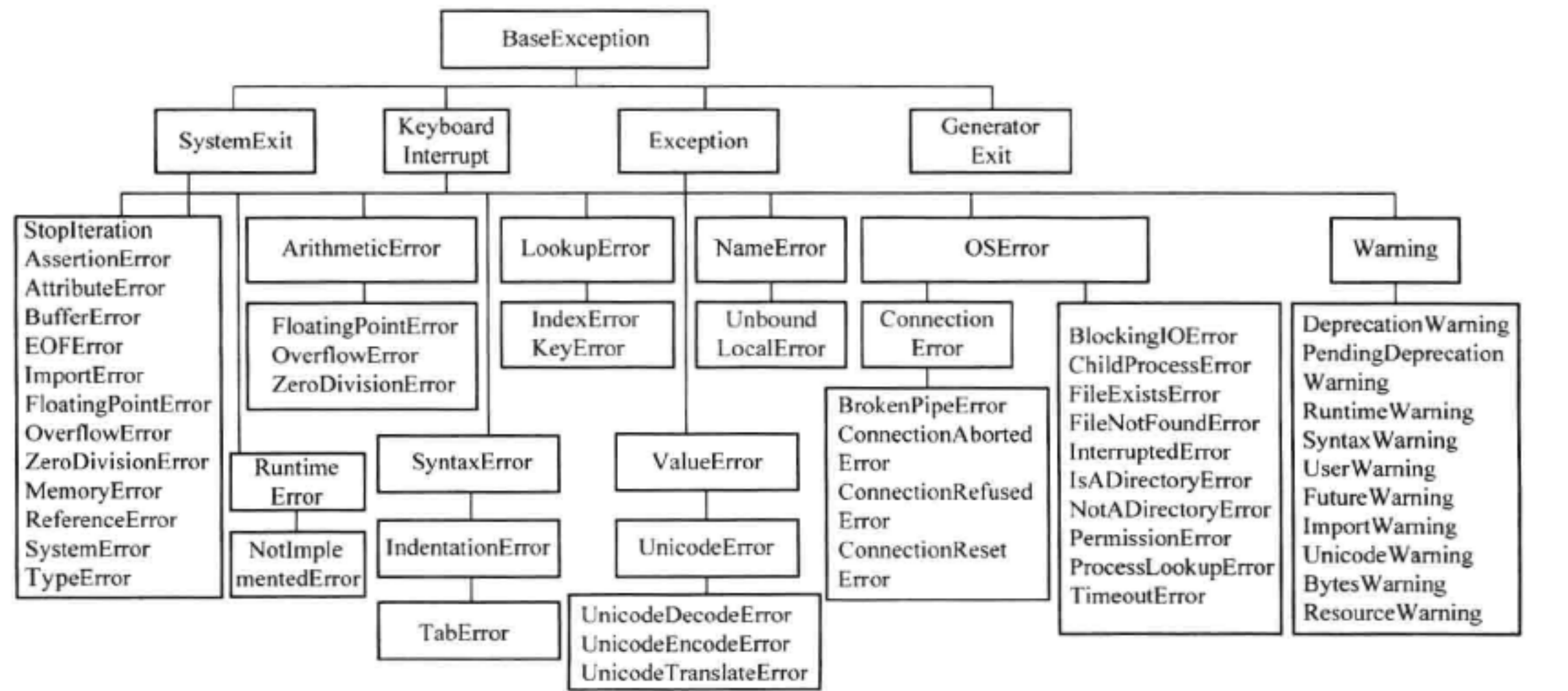


图 3-9 Python 异常类的层次结构

3.5.4 自定义异常类

Python 库中提供了许多异常。在应用程序开发过程中,有时候需要定义特定于应用程序的异常类,表示应用程序的一些错误类型。

自定义异常类一般继承于 `Exception` 或其子类。自定义异常类的命名规则一般以 `Error` 或 `Exception` 为后缀。

【例 3-19】 创建自定义异常 (`NumberError.py`), 处理应用程序中出现负数参数的异常 (例如, 学生成绩处理类, 不能容许成绩为负数)。

```
class NumberError(Exception): #自定义异常类,继承于 Exception
    def __init__(self, data):
        Exception.__init__(self, data)
        self.data = data
    def __str__(self):          #重载__str__方法
        return self.data + ':非法数值(<0)'
if __name__ == '__main__':
    raise NumberError('-123')
```

3.5.5 引发异常

1. Python 虚拟机抛出异常

大部分由程序错误而产生的错误和异常,一般由 Python 虚拟机自动抛出。例如:

```
>>> 1/0          # ZeroDivisionError:division by zero
```

2. 程序代码抛出异常

在方法体内,如果判断某种应用程序错误,则可创建相应的异常类的对象,并通过 `raise` 语句抛出。

【例 3-20】 抛出异常示例 (`raise_exception.py`)。如果传入的参数 (学生成绩) 小于 0, 则处理抛出异常 `NumberError`。

```
from NumberError import *
def average(data):
    sum = 0
    for i in data:
        if i < 0:raise NumberError(str(i))
        sum += i
    return sum/len(data)
if __name__ == '__main__':
    data1 = (44,78,90,80,55)
    print('平均值=',average(data1))
    data2 = (44,78,90,-80,55)
    print('平均值=',average(data2))
```

运行结果如下:

```
平均值 = 69.4
```

```
Traceback (most recent call last):
  File "C:/Python/chapter03/raise.py", line 18, in <module>
    print('平均值 =', average(data2))
  File "C:/Python/chapter03/raise.py", line 11, in average
    if i < 0: raise NumberError(str(i))
NumberError: -80:非法数值(<0)
```

3.5.6 捕获处理异常 try...except...finally

1. Python 异常捕获机制

当程序中的某个方法抛出异常后，Python 虚拟机通过调用堆栈查找相应的异常捕获程序。如果找到匹配的异常捕获程序（即调用堆栈中某函数使用 try...except 语句捕获处理），则执行相应的处理程序（try...except 语句中匹配的 except 语句块）。如果堆栈中没有匹配的异常捕获程序，则 Python 虚拟机捕获处理异常。

2. Python 虚拟机捕获处理异常

如果堆栈中没有匹配的异常捕获程序，则该异常最后会传递给 Python 虚拟机，Python 虚拟机通用异常处理程序在控制台打印出异常的错误信息和调用堆栈，并中止程序的执行。

【例 3-21】 Python 虚拟机捕获处理异常示例（pvmexcept.py）。

```
i1 = 1
i2 = 0
print(i1/i2)
```

运行结果如下：

```
Traceback (most recent call last):
  File "C:/Python/chapter03/pvmexcept.py", line 4, in <module>
    print(i1/i2)
ZeroDivisionError: division by zero
```

3. 使用 try...except...else...finally 语句捕获处理异常

Python 语言采用结构化的异常处理机制。在程序运行过程中，如果产生错误，则抛出异常；try 语句定义代码块，运行可能抛出异常的代码；except 语句捕获特定的异常并执行相应的处理；else 语句执行无异常时的处理；finally 语句保证即使产生异常（处理失败），也可以在事后清理资源等。try...except...else...finally 语句的一般格式如下：

```
try:
    可能产生异常的语句
except Exception1:                #捕获异常 Exception1
    发生异常时执行的语句
except (Exception2, Exception3):  #捕获异常 Exception2、Exception3
    发生异常时执行的语句
except Exception4 as e:           #捕获异常 Exception4,其实例为 e
    发生异常时执行的语句
except:                            #捕获其他所有异常
    发生异常时执行的语句
```

```
else:                                #无异常
    无异常时执行的语句
finally:                             #不管发生异常与否,保证执行
    不管发生异常与否,保证执行的语句
```

try 语句有以下三种可能的形式。

- ① try...except...[else...]语句：一个 try 块后接一个或多个 except 块，可选 else 块。
- ② try...finally 语句：一个 try 块后接一个 finally 块。
- ③ try...except...[else...] finally 语句：一个 try 块后接一个或多个 except 块，可选 else 块，后面再跟一个 finally 块。

except 块可以捕获并处理特定的异常类型（此类型称为“异常筛选器”），具有不同异常筛选器的多个 except 块可以串联在一起。系统自动由上至下匹配引发的异常：如果匹配（引发的异常为“异常筛选器”的类型或子类型），则执行该 except 块中的异常处理代码；否则继续匹配下一个 except 块。故需要将带有最具体的（即派生程度最高的）异常类的 except 块放在最前面。

finally 块始终在执行完 try 和 except 块之后执行，而与是否引发异常或者是否找到与异常类型匹配的 except 块无关。finally 块用于清理在 try 块中执行的操作，如释放其占有的资源（如文件流、数据库连接和图形句柄），而不用等待由运行库中的垃圾回收器来完成对象。

使用 try...except...else...finally 语句，还可以重新引发异常，即处理部分异常，然后使用 raise 语句重新引发异常，以便调用堆栈中的其他异常处理程序捕获并处理。

【例 3-22】 try...except...else...finally 示例（try_except.py）。

```
from NumberError import *
from raise_exception import *
if __name__ == '__main__':
    try:
        data2 = (44,78,90,-80,55)
        print('平均值 =',average(data2))
    except NumberError as e:
        print(e)
```

【例 3-23】 异常类位置顺序示例（try_except2.py）：派生程度高的异常类 NumberError 放置在派生程度低的 Exception 后面，导致程序永远无法捕获。

```
from NumberError import *
from raise_exception import *
if __name__ == '__main__':
    try:
        data2 = (44,78,90,-80,55)
        print('平均值 =',average(data2))
    except Exception:
        print('发生异常')
    except NumberError:
        print('数值不能为负')
```


【例 3-24】 使用 finally 语句保证执行代码示例 (try_finally.py)。将输入的字符串写入到文本中，直至按 Q 键结束。如果按 Ctrl + C 键中断程序运行，也保证打开的文件正常关闭。

```
try:
    f = open('mytext.txt','w') #打开要写入的文件
    while True:
        s = input('请输入字符串(按 Q 键结束):')
        if s.upper() == 'Q':break
        f.write(s + '\n')
except KeyboardInterrupt:
    print('程序中断! (Ctrl + C 键)')
finally:
    f.close()
```

【例 3-25】 重新引发异常示例 (try_except_raise.py)：使用 exception 语句捕获处理异常，随后重新引发该异常，以便对该异常进行后续处理。

```
from NumberError import *
from raise_exception import *
if __name__ == '__main__':
    try:
        data2 = (44,78,90,-80,55)
        print('平均值 =',average(data2))
    except NumberError as e:
        print('数值不能为负')
        raise e
```

3.6 断言处理

3.6.1 断言处理概述

编写程序时，在调试阶段往往需要判断代码执行过程变量的值等信息（例如：对象是否为空，数值是否为 0 等）。

虽然可以使用 print() 函数打印输出结果，也可以通过断点跟踪调试查看变量，但使用断言更加灵活高效。断言一般用于下列情况。

- ① 前置条件断言：代码执行之前必须具备的特性。
- ② 后置条件断言：代码执行之后必须具备的特性。
- ③ 前后不变断言：代码执行前后不能变化的特性。

断言的主要功能是帮助程序员调试程序，以保证程序运行的正确性；断言一般在开发调试阶段使用。即调试模式时断言有效，优化模式运行时，自动忽略断言。

相对的，异常处理则是通过错误的抛出和捕获机制，以保证程序的健壮性；异常处理贯穿于程序开发运行的各个阶段。

3.6.2 assert 语句和 AssertionError 类

使用关键字 `assert` 可以声明断言。断言声明包括下列两种形式：

```
assert <布尔表达式>                #简单形式
assert <布尔表达式>, <字符串表达式> #带参数的形式
```

其中 <布尔表达式> 的结果为一个布尔值 (True 或 False)，<字符串表达式> 是断言失败时输出的失败消息。在调试模式，如果 <布尔表达式> 为假，则抛出 `AssertionError` 对象实例。

Python 解释器有两种运行模式：通常为调试模式，内置只读变量 `__debug__` 为 True。使用选项 `-O` 运行时（即 `python.exe -O`）为优化模式，此时内置只读变量 `__debug__` 为 False。故两种形式的 `assert` 语句相当于：

```
if __debug__:
    if not testexpression: raise AssertionError
if __debug__:
    if not testexpression: raise AssertionError( data)
```

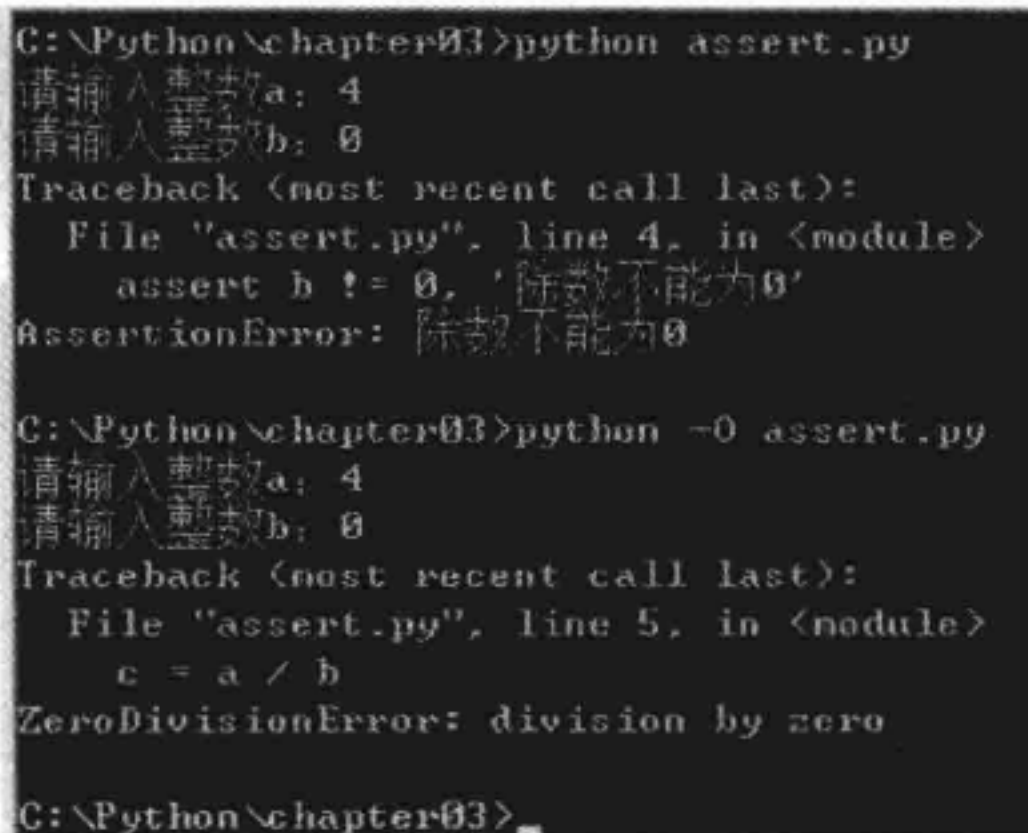
【例 3-26】断言示例 (assert.py)。

```
a = int(input("请输入整数 a:"))
b = int(input("请输入整数 b:"))
assert b != 0, '除数不能为 0'
c = a/b
print(a, '/', b, '=', c)
```

3.6.3 启用/禁用断言

通常 python 运行在调试模式，程序中的断言语句可以帮助程序调错。正式运行时，使用运行选项 `-O`，以优化模式运行来禁用断言，从而提高程序效率。

【例 3-27】启用/禁用断言选项示例：分别在两种模式下运行例 3-26。运行效果如图 3-10 所示。



```
C:\Python\chapter03>python assert.py
请输入整数a: 4
请输入整数b: 0
Traceback (most recent call last):
  File "assert.py", line 4, in <module>
    assert b != 0, '除数不能为0'
AssertionError: 除数不能为0

C:\Python\chapter03>python -O assert.py
请输入整数a: 4
请输入整数b: 0
Traceback (most recent call last):
  File "assert.py", line 5, in <module>
    c = a / b
ZeroDivisionError: division by zero

C:\Python\chapter03>
```

图 3-10 启用/禁用断言运行效果

3.7 复习题

一、单选题

1. 下面 Python 循环体执行的次数与其他不同的是_____。

- | | |
|---|--|
| A. <code>i = 0</code>
<code>while(i <= 10):</code>
<code>print(i)</code>
<code>i = i + 1</code> | B. <code>i = 10</code>
<code>while(i > 0):</code>
<code>print(i)</code>
<code>i = i - 1</code> |
| C. <code>for i in range(10):</code>
<code>print(i)</code> | D. <code>for i in range(10, 0, -1):</code>
<code>print(i)</code> |

2. 执行下列 Python 语句将产生的结果是_____。

```
x = 2; y = 2.0
if(x == y): print("Equal")
else: print("Not Equal")
```

- | | |
|----------|--------------|
| A. Equal | B. Not Equal |
| C. 编译错误 | D. 运行时错误 |

3. 执行下列 Python 语句将产生的结果是_____。

```
i = 1
if(i): print(True)
else: print(False)
```

- | | |
|-------------|------------|
| A. 输出 1 | B. 输出 True |
| C. 输出 False | D. 编译错误 |

4. 用 if 语句表示如下分段函数 $f(x)$ ，下面不正确的程序是_____。

$$f(x) = \begin{cases} 2x + 1 & x \geq 1 \\ 3x / (x - 1) & x < 1 \end{cases}$$

- | | |
|--|--|
| A. <code>if(x >= 1): f = 2 * x + 1</code>
<code>f = 3 * x / (x - 1)</code> | B. <code>if(x >= 1): f = 2 * x + 1</code>
<code>if(x < 1): f = 3 * x / (x - 1)</code> |
| C. <code>f = 3 * x / (x - 1)</code>
<code>if(x >= 1): f = 2 * x + 1</code> | D. <code>if(x < 1): f = 3 * x / (x - 1)</code>
<code>else: f = 2 * x + 1</code> |

5. 下面 if 语句统计满足“性别 (gender) 为男、职称 (duty) 为副教授、年龄 (age) 小于 40 岁”条件的人数，正确的语句为_____。

- | |
|---|
| A. <code>if(gender == "男" or age < 40 and duty == "副教授"): n += 1</code> |
| B. <code>if(gender == "男" and age < 40 and duty == "副教授"): n += 1</code> |
| C. <code>if(gender == "男" and age < 40 or duty == "副教授"): n += 1</code> |
| D. <code>if(gender == "男" or age < 40 or duty == "副教授"): n += 1</code> |

6. 下面程序段求两个数 x 和 y 中的大数，_____是不正确的。

- | | |
|---|---|
| A. <code>maxNum = x if x > y else y</code> | B. <code>maxNum = math.max(x, y)</code> |
|---|---|

C. `if(x > y):maxNum = x`
`else: maxNum = y`

D. `if(y >= x):maxNum = y`
`maxNum = x`

7. 下面 if 语句统计“成绩 (mark) 优秀的男生以及不及格的男生”的人数, 正确的语句为_____。

A. `if(gender == "男" and mark < 60 or mark >= 90):n += 1`

B. `if(gender == "男" and mark < 60 and mark >= 90):n += 1`

C. `if(gender == "男" and(mark < 60 or mark >= 90)):n += 1`

D. `if(gender == "男" or mark < 60 or mark >= 90):n += 1`

8. 用 if 语句表示如下分段函数:

$$y = \begin{cases} x^2 - 2x + 3 & x < 1 \\ \sqrt{x-1} & x \geq 1 \end{cases}$$

下面不正确的程序段为_____。

A. `if(x < 1):y = x * x - 2 * x + 3`
`else: y = math. sqrt(x - 1)`

B. `if(x < 1):y = x * x - 2 * x + 3`
`y = math. sqrt(x - 1)`

C. `y = x * x - 2 * x + 3`

D. `if(x < 1):y = x * x - 2 * x + 3`

`if(x >= 1):y = math. sqrt(x - 1)`

`if(x >= 1):y = math. sqrt(x - 1)`

9. 以下 for 语句结构中, _____不能完成 1 ~ 10 的累加功能。

A. `for i in range(10,0):sum += i`

B. `for i in range(1,11):sum += i`

C. `for i in range(10,0, -1):sum += i`

D. `for i in(10,9,8,7,6,5,4,3,2,1):sum += i`

10. 以下关于异常处理 try 语句块的说法, 不正确的是_____。

A. finally 语句中的代码段始终要保证被执行

B. 一个 try 块后接一个或多个 except 块

C. 一个 try 块后接一个或多个 finally 块

D. try 块必须与 except 或 finally 块一起使用

二、填空题

1. Python 语句 `x = True; y = False; z = False; print(x or y and z)` 的运行结果是_____。

2. Python 语句 `x = 0; y = True; print(x >= y and 'A' < 'B')` 的运行结果是_____。

3. 在直角坐标系中, x 、 y 是坐标系中任意点的位置, 用 x 和 y 表示第一象限或第二象限的 Python 表达式为_____。

4. 判断整数 i 能否同时被 3 和 5 整除的 Python 表达式为_____。

5. Python 四种内置的数值类型为: _____。

6. Python 内置的序列数据类型包括: _____。

7. Python 无穷循环 `while True:` 的循环体中可用_____语句退出循环。

8. 已知 `a = 3; b = 5; c = 6; d = True`, 则表达式 `not d or a >= 0 and a + c > b + 3` 的值是_____。

9. Python 表达式 `16 - 2 * 5 > 7 * 8 / 2 or "XYZ" != "xyz" and not(10 - 6 > 18 / 2)` 的值为_____。

- _____。
10. 执行下列 Python 语句将产生的结果是_____。
- ```
m = True;n = False;p = True
b1 = m | n^p;b2 = n | m^p
print(b1,b2)
```
11. 循环语句 for i in range( -3,21,4) 的循环次数为\_\_\_\_\_。
12. 要使语句 for i in range(\_\_\_\_, -4, -2) 循环执行 15 次, 则循环变量 i 的初值应当为\_\_\_\_\_。
13. 执行下列 Python 语句后的输出结果是\_\_\_\_\_, 循环执行了\_\_\_\_\_次。

```
i = -1;
while(i < 0):i * = i
print(i)
```

### 三、思考题

1. Python 中 type(1) 的含义是什么?
2. 简述 Python 中 try...except...finally 中各语句的用法和作用。
3. 说明以下三个 if 语句的区别:

(1) if(i > 0):

if(j > 0):n = 1

else:n = 2

(2) if(i > 0):

if(j > 0):n = 1

else:n = 2

(3) if(i > 0):n = 1

else:

if(j > 0):n = 2

4. 下列 Python 语句的运行结果为\_\_\_\_\_。

```
x = False;y = True;z = False
if x or y and z:print("yes")
else:print("no")
```

5. 下列 Python 语句的运行结果为\_\_\_\_\_。

```
x = True;y = False;z = True;
if not x or y:print(1)
elif not x or not y and z:print(2)
elif not x or y or not y and x:print(3)
else:print(4)
```

6. 下列 Python 语句的运行结果为\_\_\_\_\_。

```
for i in range(3):print(i,end='')
for i in range(2,5):print(i,end='')
```

7. 阅读下面的 Python 程序, 请问程序的功能是什么?

```
import math;n = 0
```

```

for m in range(101,201,2):
 k = int(math. sqrt(m))
 for i in range(2,k+2):
 if m%i == 0:break
 if i == k + 1:
 if n%10 == 0:print()
 print('%d'%m,end=' ')
 n += 1

```

8. 阅读下面的 Python 程序, 请问输出结果是什么?

```

n = int(input(" 请输入图形的行数:"))
for i in range(0,n):
 for j in range(0,10-i):print(" ",end=' ')
 for j in range(0,2*i+1):print(" * ",end=' ')
 print("\n")

```

9. 阅读下面的 Python 程序, 请问输出结果是什么? 程序的功能是什么?

```

from math import *
print(" 三位数中的所有的水仙花数为:")
for i in range(100,1000):
 n1 = i//100;n2 = (i%100)//10;n3 = i%10
 if(pow(n1,3) + pow(n2,3) + pow(n3,3) == i):print(i,end=' ')

```

10. 阅读下面的 Python 程序, 请问输出结果是什么? 程序的功能是什么?

```

print(" 1 ~ 1000 之间所有的完数有,其因子为:")
for n in range(1,1001):
 sum = 0;j = 0;factors = []
 for i in range(1,n):
 if(n%i == 0):
 factors. append(i);sum += i
 if(sum == n):print(" {0} : {1} ". format(n,factors))

```

11. 阅读下面的 Python 程序, 请问输出结果是什么? 程序的功能是什么?

```

m = int(input(" 请输入整数 m:")); n = int(input(" 请输入整数 n:"))
while(m != n):
 if(m > n):m = m - n
 else:n = n - m
print(m)

```

12. 阅读下面的 Python 程序, 请问输出结果是什么?

```

print("T",end=' ') if not 0 else print('F',end=' ')
print("T",end=' ') if 6 else print('F',end=' ')
print("T",end=' ') if "" else print('F',end=' ')
print("T",end=' ') if "abc" else print('F',end=' ')
print("T",end=' ') if() else print('F',end=' ')
print("T",end=' ') if(1,2) else print('F',end=' ')
print("T",end=' ') if [] else print('F',end=' ')

```



```
print("T",end='') if [1,2] else print('F',end='')
```

```
print("T",end='') if {} else print('F',end='')
```

```
print("T",end='') if {1,2} else print('F',end='')
```

13. 阅读下面的 Python 程序，请问输出结果是什么？

```
print(1 or 2,0 or 2,False or True,True or False,False or 2,sep='')
```

```
print(1 and 2,0 and 2,False and 2,True and 2,False and True,sep='')
```

## 3.8 上机实践

1. 编写程序，计算  $1 + 2 + 3 + \dots + 100$ 。

2. 编写程序，计算  $10 + 9 + 8 + \dots + 1$ 。

3. 编写程序，计算  $1 + 3 + 5 + 7 + \dots + 99$ 。

4. 编写程序，计算  $2 + 4 + 6 + 8 + \dots + 100$ 。

5. 编写程序，输出 2000 ~ 3000 年之间的所有闰年。请思考，有哪几种实现方法？

6. 编写程序，计算  $S_n = 1 - 3 + 5 - 7 + 9 - 11 + \dots$ 。

7. 编写程序，计算  $S_n = 1 + 1/2 + 1/3 + \dots$ 。

8. 编写程序，打印九九乘法表。

9. 编写程序，输入三角形三条边，先判断是否可以构成三角形，如果可以，则求三角形的周长和面积，否则报错。运行效果如图 3-11 所示（结果均保留一位小数）。

```
请输入三角形的边A: 3
请输入三角形的边B: 4
请输入三角形的边C: 5
三角形三边分别为: a=3.0, b=4.0, c=5.0
三角形的周长 = 12.0, 面积 = 6.0
```

图 3-11 三角形周长和面积运行效果

提示：

(1) 3 个数可以构成三角形必须满足如下条件：每条边长均大于 0，并且任意两边之和大于第三边。

(2) 已知三角形的三条边，则三角形的面积  $= \sqrt{h * (h - a) * (h - b) * (h - c)}$ ，其中， $h$  为三角形周长的一半。

10. 编写程序，输入  $x$ ，根据如下公式，计算分段函数  $y$  的值。分别利用“一句单分支语句”、“两句单分支语句”、“双分支结构”及“条件运算语句”四种方法实现。运行效果如图 3-12 所示。

```
请输入x: 5
方法一: x = 5.0, y = 6.990927699183115
方法二: x = 5.0, y = 6.990927699183115
方法三: x = 5.0, y = 6.990927699183115
方法四: x = 5.0, y = 6.990927699183115
```

图 3-12 分段函数运行效果

$$y = \begin{cases} \frac{x^2 - 3x}{x + 1} + 2\pi + \sin x & x \geq 0 \\ \ln(-5x) + 6\sqrt{|x|} + e^4 - (x + 1)^3 & x < 0 \end{cases}$$

11. 编写程序, 输入一元二次方程的三个系数  $a$ 、 $b$  和  $c$ , 求  $ax^2 + bx + c = 0$  方程的解。运行效果如图 3-13 所示。

|           |              |                 |
|-----------|--------------|-----------------|
| 请输入系数a: 0 | 请输入系数a: 0    | 请输入系数a: 1       |
| 请输入系数b: 0 | 请输入系数b: 1    | 请输入系数b: -2      |
| 请输入系数c: 6 | 请输入系数c: 2    | 请输入系数c: 1       |
| 此方程无解!    | 此方程的解为: -2.0 | 此方程有两个相等实根: 1.0 |

(a) 无解

(b) 一个实根

(c) 两个相等实根

|                        |
|------------------------|
| 请输入系数a: 1              |
| 请输入系数b: -1             |
| 请输入系数c: -6             |
| 此方程有两个不等实根: 3.0 和 -2.0 |

(d) 两个不等实根

|                                 |
|---------------------------------|
| 请输入系数a: 1                       |
| 请输入系数b: -1                      |
| 请输入系数c: 0.5                     |
| 此方程有两个不等实根: 0.5+0.5i 和 0.5-0.5i |

(e) 两个共轭复根

图 3-13 求解一元二次方程

提示: 方程  $ax^2 + bx + c = 0$  的解有以下几种情况。

(1)  $a = 0$ ,  $b = 0$ , 无解。

(2)  $a = 0$ ,  $b \neq 0$ , 有一个实根:  $x = -\frac{c}{b}$ 。

(3)  $b^2 - 4ac = 0$ , 有两个相等实根:  $x_1 = x_2 = -\frac{b}{2a}$ 。

(4)  $b^2 - 4ac > 0$ , 有两个不等实根:  $x_1 = -\frac{b}{2a} + \frac{\sqrt{b^2 - 4ac}}{2a}$ ,  $x_2 = -\frac{b}{2a} - \frac{\sqrt{b^2 - 4ac}}{2a}$ 。

(5)  $b^2 - 4ac < 0$ , 有两个共轭复根:  $x_1 = -\frac{b}{2a} + \frac{\sqrt{4ac - b^2}}{2a}j$ ,  $x_2 = -\frac{b}{2a} - \frac{\sqrt{4ac - b^2}}{2a}j$ 。

12. 编写程序, 输入整数  $n$  ( $n \geq 0$ ), 分别利用 for 循环和 while 循环求  $n!$ 。运行效果如图 3-14 所示。

提示:

(1)  $n! = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1$ 。例如,  $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ 。特别地,  $0! = 1$ 。

(2) 一般地, 累乘的初值为 1, 而累加的初值为 0。

(3) 如果输入的是负整数, 则继续提示输入非负整数, 直至  $n \geq 0$ 。

13. 编写程序, 产生两个 0 ~ 100 之间 (包含 0 和 100) 的随机数  $a$  和  $b$ , 求这两个整数的最大公约数和最小公倍数。运行效果如图 3-15 所示。

|                   |
|-------------------|
| 请输入非负整数n: -5      |
| 请输入非负整数n: 5       |
| for循环: 5! = 120   |
| while循环: 5! = 120 |

图 3-14 阶乘运行效果

|                        |
|------------------------|
| 整数1 = 88, 整数2 = 16     |
| 最大公约数 = 8, 最小公倍数 = 176 |

图 3-15 最大公约数和最小公倍数运行效果

# 第4章 数值类型

Python 提供了丰富的数据类型和库函数，用于程序设计中的数值处理。

本章要点：

- ◆ int 类型（任意精度整数）；
- ◆ float 类型（有限精度浮点数）；
- ◆ Decimal 类型（高精度浮点数）；
- ◆ Fraction 类型（分数）；
- ◆ complex 类型（复数）；
- ◆ 算术运算符和位运算符；
- ◆ math 模块和数学函数；
- ◆ cmath 模块和复数数学函数；
- ◆ random 模块和随机函数。

## 4.1 int 类型（任意精度整数）

整数类型（int）是表示整数的数据类型。与其他计算机语言有精度限制不同，Python 的整数位数可以为任意长度位数（只受限于计算机内存）。整型对象是不可变对象。

### 4.1.1 整型常量

数字字符串（前面可以带负号 -）即整型常量。Python 解释器自动创建 int 型对象实例。

数字字符串通常解释为十进制（基数为 10）数制。可以用前缀 0x（或 0X，数字 0 加小写或大写字母 x）将整数强制为十六进制（基数为 16）；或使用前缀 0o（或 0O，数字 0 加小写或大写字母 o）将整数强制为八进制（基数为 8）；或使用前缀 0b（或 0B，数字 0 加小写或大写字母 b）将整数强制为二进制（基数为 2）。跟在前缀后面的数字必须适合于数制，如表 4-1 所示。

表 4-1 整型常量

| 数 制           | 前 缀     | 基 本 数 码                | 示 例                               |
|---------------|---------|------------------------|-----------------------------------|
| 十进制（以 10 为基）  |         | 0 ~ 9                  | 0, 1, 2, 7, 999, -12（负数）, +12（正数） |
| 十六进制（以 16 为基） | 0x 或 0X | 0 ~ 9 和 A ~ F（或 a ~ f） | 0x0, 0X1, 0x2, 0X7, 0x3e7         |
| 八进制（以 8 为基）   | 0o 或 0O | 0 ~ 7                  | 0o0, 0O1, 0o2, 0O7, 0o1747        |
| 二进制（以 2 为基）   | 0b 或 0B | 0 ~ 1                  | 0b0, 0B1, 0b10, 0B111, 0b1100011  |



## 4.1.2 int 对象

### 1. 创建 int 对象

可创建 int 类型的对象实例，其基本形式为：

```
int(x = 0) #创建 int 对象(十进制)
int(x, base = 10) #创建 int 对象,指定进制为 base(2 到 36 之间)
```

通过创建 int 对象，可以把数值或任何符合格式的字符串或其他对象转换为 int 对象。

注意，如果对象 x 不能转换为整型，则导致 TypeError；如果对象 x 转换失败，则导致 ValueError。例如：

```
>>> int(), int(123), int('456 '), int(1.23) #结果:(0, 123, 456, 1)
>>> int('FF ', 16), int('100 ', 2) #结果:(255, 4)
>>> int('abc ') #ValueError: invalid literal for int() with base 10: 'abc '
>>> int(100, 2) #TypeError: can 't convert non - string with explicit base
```

### 2. int 对象的方法

int 对象 i 包含的主要方法如下。

i. bit\_length(): 返回 i 的二进制位数，不包括符号。

例如：

```
>>> i = -10
>>> bin(i) #结果:'-0b1010 '
>>> i.bit_length(), int.bit_length(i) #结果:(4, 4)
```

## 4.1.3 数制转换函数

Python 包含下列内置函数，用于不同数制之间的转换，如表 4-2 所示。

表 4-2 数制转换函数

| 函 数          | 说 明          | 示 例                        |
|--------------|--------------|----------------------------|
| bin( number) | 数值转换为二进制字符串  | bin( 100) #结果:'0b1100100 ' |
| hex( number) | 数值转换为十六进制字符串 | hex( 100) #结果:'0x64 '      |
| oct( number) | 数值转换为八进制字符串  | oct( 100) #结果:'0o144 '     |

## 4.1.4 整数的运算

整数支持算术运算符（参见 4.6.1 节）、位运算符（参见 4.6.2 节）、内置函数（参见 4.6.4 节）、math 模块中的数学运算函数（参见 4.7 节）。例如：

```
>>> 234 * 456 #结果:106704
>>> import math
>>> math. pow(2, 10) #结果:1024.0
```

## 4.2 float 类型（有限精度浮点数）

浮点类型（float）是表示实数的数据类型。与其他计算机语言的双精度（double）和单

精度对应。Python 浮点类型的精度与系统相关。

4.2.1 浮点类型常量

浮点类型常量可以为带小数点的数字字符串，或用科学计数器表示的数字字符串（前面可以带负号-），即整型常量。Python 解释器自动创建 float 型对象实例。

浮点类型常量的举例如表 4-3 所示。

表 4-3 浮点类型常量举例

| 举 例                       | 说 明                                                    |
|---------------------------|--------------------------------------------------------|
| 1.23, -24.5, 1.0, 0.2     | 带小数点的数字字符串                                             |
| 1., .2                    | 小数点的前后 0 可以省略                                          |
| 3.14e-10, 4E210, 4.0e+210 | 科学计数法（e 或 E 表示底数 10），如 3.14e-10=3.14*10 <sup>-10</sup> |

4.2.2 float 对象

1. 创建 float 对象

可创建 float 类型的对象实例，其基本形式为：

float(x)

通过创建 float 对象，可以把数值或任何符合格式的字符串转换为 float 对象。

注意，如果对象 x 不能转换为 float 对象，将导致 TypeError；如果对象 x 转换失败，将导致 ValueError。特殊字符串'Infinity'、'-Infinity'、'NaN'，分别用于表示正无穷大、负无穷大、非数值。例如：

```
>>> float(123), float('3.14') #结果:(123.0, 3.14)
>>> float('Infinity'), float('-Infinity'), float('NaN') #结果:(inf, -inf, nan)
>>> float('45abc') #ValueError: could not convert string to float: '45abc'
```

2. float 对象的方法

float 对象包含的主要方法如表 4-4 所示。

表 4-4 float 对象的主要方法

| 方 法                | 说 明           | 示 例                                                                                            |
|--------------------|---------------|------------------------------------------------------------------------------------------------|
| as_integer_ratio() | 转换为分数         | 1.25.as_integer_ratio() #结果:(5, 4)<br>float.as_integer_ratio(1.25) #结果:(5, 4)                  |
| hex()              | 转换为十六进制字符串    | 12.3.hex() #结果:'0x1.899999999999ap+3'<br>float.hex(12.3) #结果:'0x1.899999999999ap+3'            |
| fromhex(string)类方法 | 十六进制字符串转换为浮点数 | float.fromhex('0xFF') #结果:255.0<br>#格式:[sign][ '0x ' ] integer [ '.'fraction ] [ 'p'exponent ] |
| is_integer()       | 判断是否为 int 类型  | 3.14.is_integer() #结果:False<br>float.is_integer(2.0) #结果:True                                  |

4.2.3 浮点数的运算

浮点数支持算术运算符（参见 4.6.1 节），math 模块中包含了用于浮点数运算的函数

(参见 4.7 节)。注意，浮点数运算会产生误差（参见 4.3.1 节）。例如：

```
>>> 2.34 * 4.56 #结果:10.670399999999999
>>> import math
>>> math.pow(1.05, 5) #结果:1.2762815625000004
```

## 4.3 Decimal 类型（高精度浮点数）

### 4.3.1 浮点数运算误差

float 数据类型的精度有限。例如：

```
>>> 1.1 + 2.2 #结果:3.3000000000000003
>>> (1.1 + 2.2) == 3.3 #结果:False
```

(1.1 + 2.2) 在 Python 的内部表示为 3.3000000000000003，这是由于计算机 CPU 的浮点运算单元采用 IEEE 754 格式产生的误差。

### 4.3.2 Decimal 对象

#### 1. 创建 Decimal 对象

如果需要高精度的小数运算，可以导入 decimal 模块。该模块提供了两种基本数据类型：Decimal（高精度浮点数）和 Context（精度和舍入参数），以控制运算精度、有效位数和四舍五入操作。

可以使用 Python 模块 decimal 中的 Decimal 类型，创建 Decimal 类型的对象实例，其基本形式为：

```
from decimal import Decimal #从模块 decimal 导入 Decimal
Decimal(value="0", context=None) #创建 Decimal 对象
```

其中，value 可以为整型、字符串、元组（（符号，数字，指数），例如：(1,(2,3,4,5),-3)、(0,(1,2),2))、Decimal 对象。通过创建 Decimal 对象，可以把数值或任何符合格式的字符串转换为 Decimal 对象。特殊字符串 'Infinity'、'-Infinity' 以及 'NaN'，则用于表示正无穷大、负无穷大以及非数值。context 对象用于控制高精度浮点数运算。例如：

```
>>> from decimal import Decimal
>>> a = Decimal('1.1'); b = Decimal('2.2'); c = Decimal('3.3')
>>> a, b, c #结果:(Decimal('1.1'), Decimal('2.2'), Decimal('3.3'))
>>> Decimal((1, (2, 3, 4, 5), -3)), Decimal((0, (1, 2), 2))
(Decimal('-2.345'), Decimal('1.2E+3'))
>>> Decimal('Infinity'), Decimal('-Infinity'), Decimal('NaN'), Decimal('sNaN')
(Decimal('Infinity'), Decimal('-Infinity'), Decimal('NaN'), Decimal('sNaN'))
>>> (a + b) == c #结果:True
```

比较：

```
>>> from decimal import Decimal
>>> x = Decimal(1.1); y = Decimal(2.1); z = Decimal(3.3)
>>> (x + y) == z #结果:False
```



```
>>> x #结果:Decimal('1.100000000000000088817841970012523233890533447265625')
>>> y #结果:Decimal('2.200000000000000017763568394002504646778106689453125')
>>> z #结果:Decimal('3.29999999999999982236431605997495353221893310546875')
```

说明：Decimal(1.1)，其中浮点数 1.1 会先转换为 IEEE 854 – 1987 标准格式，故等价于 Decimal('1.100000000000000088817841970012523233890533447265625')。所以 (x + y) 不等于 z。

2. Decimal 对象的方法

Decimal 对象包含的主要方法如表 4-5 所示。

表 4-5 Decimal 对象的主要方法

| 方 法                                 | 说 明                        | 示 例                                                                                                                                                                                     |
|-------------------------------------|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| sqrt()<br>exp()<br>log10()<br>..... | 取平方根<br>求 e 的幂<br>求 10 的对数 | >>> Decimal('2').sqrt()<br>Decimal('1.414213562373095048801688724')<br>>>> Decimal('1').exp()<br>Decimal('2.718281828459045235360287471')<br>>>> Decimal('10').log10() #结果:Decimal('1') |
| quantize()                          | 数值舍入                       | >>> Decimal('1.41421356').quantize(Decimal('1.000'))<br>Decimal('1.414')                                                                                                                |

Decimal 对象包含的其他主要方法与 math 模块中的同名数学函数相对应，参见 4.7 节。

4.3.3 Context 对象

1. Context 对象概述

可以通过 Context 对象设置高精度浮点数运算涉及的各种控制参数：

```
Context (prec = None, rounding = None, Emin = None, Emax = None, capitals = None, clamp =
None, flags = None, traps = None)
```

其中，prec 为精度；rounding 为四舍五入方式；Emin/Emax 为指数范围；capitals 为科学计数法中 e 的大小写（1：大写；0 小写）；clamp 指定指数范围的精确度；flags 为环境初始状态信号列表，通常不用指定；traps 为信号列表，计算过程中发生某些事件（如除 0）时产生 Python 异常。

2. 获取 Context 对象

通常不直接创建 Context 对象，而是通过 decimal 模块中的函数 getcontext() 获取当前活动的 Context 对象。可以修改 Context 对象（如 ctx）的属性（prec、rounding、Emin、Emax、capitals、clamp 和 traps）；ctx.copy() 返回其副本。例如：

```
>>> from decimal import *
>>> ctx = getcontext() #获取当前活动的 Context 对象
>>> ctx
Context(prec = 28, rounding = ROUND_HALF_EVEN, Emin = - 999999, Emax = 999999, capitals = 1,
clamp = 0, flags = [], traps = [InvalidOperation, DivisionByZero, Overflow])
>>> ctx.prec = 5
>>> Decimal('1.1') / Decimal('2.3') #结果:Decimal('0.47826')
```

也可在 with 语句中, 通过 localcontext (ctx = None), 创建当前环境管理器, 用于局部临时改变计算控制参数。注意, 如果没有传入 ctx, 则复制当前活动的 Context 对象。例如:

```
>>> from decimal import *
>>> d = Decimal('1.1'); e = Decimal('2.3')
>>> print(d/e) #结果:0.4782608695652173913043478261
>>> with localcontext() as ctx:
 ctx.prec = 5
 print(d/e)
0.47826
```

### 3. 设置 Context 对象

decimal 模块预定义了三个 Context 对象: BasicContext、ExtendedContext 和 DefaultContext。例如:

```
>>> from decimal import *
>>> DefaultContext
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1,
clamp=0, flags=[], traps=[InvalidOperation, DivisionByZero, Overflow])
>>> BasicContext
Context(prec=9, rounding=ROUND_HALF_UP, Emin=-999999, Emax=999999, capitals=1, clamp=0,
flags=[], traps=[Clamped, InvalidOperation, DivisionByZero, Overflow, Underflow])
>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1,
clamp=0, flags=[], traps=[])
```

通过 decimal 模块中的函数 setcontext(), 可以设置当前活动的 Context 对象, 以切换计算环境。例如:

```
>>> from decimal import *
>>> Decimal('1.1') / Decimal('2.3') #结果:Decimal('0.4782608695652173913043478261')
>>> setcontext(ExtendedContext)
>>> Decimal('1.1') / Decimal('2.3') #结果:Decimal('0.478260870')
```

### 4. 舍入方式控制

通过 Context 对象的 rounding 属性可以设置高精浮点数运算的舍入方式。decimal 模块中定义了相关常量, 如表 4-6 所示。假设表中示例基于:

```
>>> from decimal import *
>>> ctx = getcontext()
>>> ctx.prec = 2
>>> Decimal('1.52') #结果:Decimal('1.52')
>>> Decimal('1.52') + 0 #结果:Decimal('1.5')
```

注: 精度和舍入只是在计算的结果中有效。故 Decimal('1.52') 的精度为 3, Decimal('1.52') + 0 的精度为 2。

表 4-6 Context 对象的 rounding 属性取值常量

| 常 量             | 含 义                                         | 举 例                                                                                                                          | 结 果                                                                    |
|-----------------|---------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| ROUND_UP        | 远离零舍入                                       | ctx.rounding = ROUND_UP<br>Decimal('1.58') + 0<br>Decimal('-2.36') + 0                                                       | Decimal('1.6')<br>Decimal('-2.4')                                      |
| ROUND_DOWN      | 向零舍入                                        | ctx.rounding = ROUND_DOWN<br>Decimal('1.58') + 0<br>Decimal('-2.36') + 0                                                     | Decimal('1.5')<br>Decimal('-2.3')                                      |
| ROUND_CEILING   | 向正无穷大舍入                                     | ctx.rounding = ROUND_CEILING<br>Decimal('1.52') + 0<br>Decimal('-2.36') + 0                                                  | Decimal('1.6')<br>Decimal('-2.3')                                      |
| ROUND_FLOOR     | 向负无穷大舍入                                     | ctx.rounding = ROUND_FLOOR<br>Decimal('1.52') + 0<br>Decimal('-2.36') + 0                                                    | Decimal('1.5')<br>Decimal('-2.4')                                      |
| ROUND_HALF_UP   | 四舍五入                                        | ctx.rounding = ROUND_HALF_UP<br>Decimal('1.55') + 0<br>Decimal('-2.35') + 0                                                  | Decimal('1.6')<br>Decimal('-2.4')                                      |
| ROUND_HALF_DOWN | 向最接近的数字舍入，如果与两个相邻数字的距离相等，则为 ROUND_DOWN 舍入模式 | ctx.rounding = ROUND_HALF_DOWN<br>Decimal('1.55') + 0<br>Decimal('-2.35') + 0                                                | Decimal('1.5')<br>Decimal('-2.3')                                      |
| ROUND_HALF_EVEN | 向最接近的数字舍入，如果与两个相邻数字的距离相等，则向相邻的偶数舍入          | ctx.rounding = ROUND_HALF_EVEN<br>Decimal('1.55') + 0<br>Decimal('1.65') + 0<br>Decimal('-2.35') + 0<br>Decimal('-2.45') + 0 | Decimal('1.6')<br>Decimal('1.6')<br>Decimal('-2.4')<br>Decimal('-2.4') |

5. 信号列表

信号列表表示计算过程中可能发生的数学异常。信号列表如表 4-7 所示。

表 4-7 Context 模块信号列表

| Context 信号列表     | 含 义                                                                |
|------------------|--------------------------------------------------------------------|
| Clamped          | 对指数进行调整以适合允许范围                                                     |
| DecimalException | 高精度浮点数异常                                                           |
| DivisionByZero   | 非无限数值除 0。例如：Decimal('1') / Decimal('0')                            |
| Inexact          | 舍入导致结果不精确。例如：Decimal('1') / Decimal('3')                           |
| InvalidOperation | 执行了非法操作。例如：Decimal('abc')                                          |
| Overflow         | 舍入导致溢出（指数 > Emax）。例如：Decimal('999999999') * * Decimal('999999999') |
| Rounded          | 舍入操作。例如：Decimal('1') / Decimal('3')                                |
| Subnormal        | 舍入操作前指数 < Emin                                                     |
| Underflow        | 数值下溢（指数 < Emin）                                                    |
| FloatOperation   | 运算中包含 float 数。例如：Decimal(1) ; Decimal('1') + 1.2                   |



计算结果产生的信号会设置给当前活动的 Context 对象的 flags 属性，通过检查该属性，可以判断计算的状态。通过 decimal 模块的 clear\_flags() 函数可清除 flags。例如：

```
>>> from decimal import *
>>> ctx = getcontext()
>>> Decimal('1') / Decimal('3') #结果:Decimal('0.3333333333333333333333333333')
>>> ctx.flags[Inexact] #结果:True
>>> ctx.clear_flags()
>>> ctx.flags[Inexact] #结果:False
```

当前活动的 Context 对象的 traps 属性中列举的信号，计算过程中发生该事件时产生 Python 异常；否则不产生异常。通过 decimal 模块的 clear\_traps() 函数可清除 traps。例如：

```
>>> from decimal import *
>>> setcontext(ExtendedContext)
>>> Decimal('1') / Decimal('0') #结果:Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = True
>>> Decimal('1') / Decimal('0') #报错 decimal.DivisionByZero: [<class 'decimal.DivisionByZero'>]
```

### 4.3.4 高精度浮点数的运算

高精度浮点数（Decimal）支持算术运算符、内置函数、math 模块中数学运算函数。另外，Decimal 以及 Context 类型的方法也可以用于高精度浮点数的运算。

## 4.4 Fraction 类型（分数）

Python 的模块 fractions 包含支持分数的类型和函数。分数对象是不可变对象。

### 4.4.1 Fraction 对象

#### 1. 创建 Fraction 对象

使用 Python 模块 fractions 模块中的 Fraction 类型，可以创建 Fraction 类型的对象实例，其基本形式为：

|                                             |                                   |
|---------------------------------------------|-----------------------------------|
| <b>from fractions import Fraction</b>       | <b>#从模块 fractions 导入 Fraction</b> |
| <b>Fraction(numerator=0, denominator=1)</b> | <b>#指定分子和分母</b>                   |
| <b>Fraction(other_fraction)</b>             | <b>#指定其他分数</b>                    |
| <b>Fraction(float)</b>                      | <b>#指定浮点数</b>                     |
| <b>Fraction(decimal)</b>                    | <b>#指定 Decimal 对象</b>             |
| <b>Fraction(string)</b>                     | <b>#指定数值字符串</b>                   |

例如：

```
>>> from decimal import *
>>> from fractions import *
>>> Fraction(4, 6) #结果:Fraction(2, 3)
>>> Fraction(1.25) #结果:Fraction(5, 4)
```

```
>>> Fraction(Decimal('1.33')) #结果:Fraction(133, 100)
>>> Fraction('3/12') #结果:Fraction(1, 4)
```

2. Fraction 对象的方法

Fraction 对象包含的主要方法如表 4-8 所示。

表 4-8 Fraction 对象的主要方法

| 方 法                                                 | 说 明                 | 示 例                                                                                                                            |
|-----------------------------------------------------|---------------------|--------------------------------------------------------------------------------------------------------------------------------|
| from_float(flt)                                     | float 浮点数转换为分数      | >>> Fraction.from_float(0.3) #会产生误差<br>Fraction(5404319552844595, 18014398509481984)                                           |
| from_decimal                                        | Decimal 对象转换为分数     | >>> Fraction.from_decimal(Decimal("0.3"))<br>Fraction(3, 10)                                                                   |
| limit_denominator<br>(max_denominator<br>= 1000000) | 返回近似分数,最大分母为指<br>定值 | >>> Fraction(0.3)<br>Fraction(5404319552844595, 18014398509481984)<br>>>> Fraction(0.3).limit_denominator()<br>Fraction(3, 10) |
| gcd(a, b)                                           | 计算整数 a 和 b 的最大公约数   | >>> gcd(16, 56) #结果:8                                                                                                          |

4.4.2 分数的运算

分数支持算术运算符，math 模块中包含了用于浮点数运算的函数。例如：

```
>>> Fraction('1/2') * Fraction('1/3') #结果:Fraction(1, 6)
>>> Fraction('1/2') + Fraction('1/3') #结果:Fraction(5, 6)
>>> Fraction('1/2') / Fraction('1/3') #结果:Fraction(3, 2)
>>> import math
>>> math.pow(Fraction('1/2'), Fraction('1/3')) #结果:0.7937005259840998
```

4.5 complex 类型（复数）

4.5.1 复数类型常量

当数值字符串中包含虚部（j 或 J）时，即复数常量。Python 解释器自动创建 complex 型对象实例。例如：

```
>>> 0j #结果:0j
>>> 2j #结果:2j
>>> 1 + 2j #结果:(1 + 2j)
>>> j #NameError: name 'j' is not defined
```

4.5.2 创建 complex 对象

可创建 complex 类型的对象实例，其基本形式为：

```
complex(real[, imag]) #创建 complex 对象(虚部可选)
```

例如：

```
>>> c = complex(4, 5)
```

### 4.5.3 complex 类型的方法

complex 对象包含的属性和方法如表 4-9 所示。

表 4-9 complex 对象的属性和方法

| 属性/方法        | 说 明   | 示 例                                    |
|--------------|-------|----------------------------------------|
| real         | 复数的实部 | >>> (1 + 2j). real      #结果:1.0        |
| imag         | 复数的实部 | >>> (1 + 2j). imag      #结果:2.0        |
| .conjugate() | 共轭复数  | >>> (1 + 2j). conjugate() #结果:(1 - 2j) |

复数在 Python 内部使用正交笛卡儿坐标表示，故： $z == z.\text{real} + z.\text{imag} * 1j$ 。

### 4.5.4 复数的运算

复数支持算术运算符、内置函数、cmath 模块中用于复数运算的数学函数。例如：

```
>>> a = 1 + 2j
>>> b = 1.1 + 2.2j
>>> c = complex(4, 5)
>>> a + b + c #结果:(6.1+9.2j)
>>> 1j * 1j #结果:(-1+0j)
>>> sqrt(c) #NameError: name 'sqrt' is not defined
>>> import cmath
>>> cmath.sqrt(c) #结果:(2.280693341665298 + 1.096157889501519j)
```

## 4.6 算术运算符和位运算符

### 4.6.1 算术运算符

Python 提供了丰富的算术运算符，用于提供包括四则运算的各种算术运算。表 4-10 以优先级为顺序列出了 Python 算术运算符。假设表中  $n$  为整型变量，取值为 8。

表 4-10 算术运算符

| 运 算 符 | 含 义  | 说 明               | 优 先 级 | 实 例       | 结 果  |
|-------|------|-------------------|-------|-----------|------|
| +     | 一元 + | 操作数的值             | 1     | + n       | 8    |
| -     | 一元 - | 操作数的反数            | 1     | - n       | -8   |
| *     | 乘法   | 操作数的积             | 2     | n * n * 2 | 128  |
| /     | 除法   | 第二个操作数除第一个操作数     | 2     | 10/n      | 1.25 |
| //    | 整数除法 | 两个整数相除，结果为整数      | 2     | 10//n     | 1    |
| %     | 模数   | 第二个操作数除第一个操作数后的余数 | 2     | 10% n     | 2    |
| +     | 加法   | 两个操作数之和           | 3     | 10 + n    | 18   |
| -     | 减法   | 从第一个操作数中减去第二个操作数  | 3     | n - 10    | -2   |



### 4.6.2 位运算符

位运算符用于按二进制位进行逻辑运算，操作数必须为整数。Python 位运算符如表 4-11 所示。

表 4-11 位运算符

| 运 算 符 | 用 法        | 含 义            | 优 先 级 | 实 例             | 结 果     |
|-------|------------|----------------|-------|-----------------|---------|
| ~     | ~ op       | 按位求补           | 1     | ~ 0x1           | - 0x2   |
| <<    | op1 << op2 | 将 op1 左移 op2 位 | 2     | 0xf0 << 4       | 0xf00   |
| >>    | op1 >> op2 | 将 op1 右移 op2 位 | 2     | 0xf0 >> 4       | 0xf     |
| &     | op1 & op2  | 按位逻辑与          | 3     | 0xff00 & 0xf0f0 | 0xf000  |
| ^     | op1 ^ op2  | 按位逻辑异或         | 4     | 0xff00 ^ 0xf0f0 | 0xff0   |
|       | op1   op2  | 按位逻辑或          | 5     | 0xff00   0xf0f0 | 0xffff0 |

【例 4-1】 位运算符示例（op\_bit.py）。

```
print(" ~0x1 结果为:", hex(~0x1))
print("0b11110000 <<4 结果为:", bin(0b111110000 <<4))
print("0b11110000 >>4 结果为:", bin(0b111110000 >>4))
print("0b1111111100000000 & 0b1111000011110000 结果为:", bin(0b1111111100000000 &
0b1111000011110000))
print("0b1111111100000000 | 0b1111000011110000 结果为:", bin(0b1111111100000000 |
0b1111000011110000))
print("0b1111111100000000 ^ 0b1111000011110000 结果为:", bin(0b1111111100000000 ^
0b1111000011110000))
```

运行结果如下：

```
~0x1 结果为: -0x2
0b11110000 << 4 结果为: 0b1111100000000
0b11110000 >>4 结果为: 0b11111
0b1111111100000000 & 0b1111000011110000 结果为: 0b1111000000000000
0b1111111100000000 | 0b1111000011110000 结果为: 0b11111111111110000
0b1111111100000000 ^ 0b1111000011110000 结果为:0b1111111110000
```

### 4.6.3 混合运算和数值类型转换

#### 1. 混合运算和隐式转换

int、float 和 complex 对象可以混合运算。如果表达式中包含 complex 对象，则其他对象自动转换（隐式转换）为 complex 对象，结果为 complex 对象；如果表达式中包含 float 对象，则其他对象自动转换（隐式转换）为 float 对象，结果为 float 对象。例如：

```
>>> f = 2 + 3.3
>>> f, type(f) #结果:(5.3, <class 'float'>)
>>> c = f - (1 + 2j)
>>> c, type(c) #结果:((4.3 - 2j), <class 'complex'>)
```

2. 显式转换（强制转换）

“显式转换”又称为“强制转换”，使用 `target - type (value)` 将表达式强制转换为所需的数据类型。如果未定义相应的转换运算符，则强制转换会失败。显式转换实际上使用目标类型的构造函数创建其对象。例如：

```
>>> int(12.3) #结果:12
>>> float(1+2j) #TypeError: can 't convert complex to float
```

显式数值转换可能导致精度损失，也可能引发异常（如溢出异常 `OverflowError`）。例如：

```
>>> i = 9999 ** 9999
>>> float(i) #OverflowError: long int too large to convert to float
```

4.6.4 内置标准数学函数

Python 包含若干用于数学运算的内置函数，如表 4-12 所示。

表 4-12 用于数学运算的内置函数

| 函 数                                      | 含 义                                                                       | 实 例                                                            | 结 果                     |
|------------------------------------------|---------------------------------------------------------------------------|----------------------------------------------------------------|-------------------------|
| <code>abs(x)</code>                      | 数值 x 的绝对值。如果 x 为复数,则返回 x 的模                                               | <code>abs(-1.2)</code><br><code>abs(1-2j)</code>               | 1.2<br>2.23606797749979 |
| <code>divmod(a,b)</code>                 | 返回 a 除以 b 的商和余数                                                           | <code>divmod(5,3)</code>                                       | (1, 2)                  |
| <code>pow(x, y[, z])</code>              | 返回 x 的 y 次幂( <code>x * * y</code> )。如果指定 z,则为: <code>pow(x, y) % z</code> | <code>pow(2,10)</code><br><code>pow(2,10,10)</code>            | 1024<br>4               |
| <code>round ( number [, ndigits])</code> | 四舍五入取整。如果指定 ndigits,则保留 ndigits 小数                                        | <code>round(3.14159)</code><br><code>round(3.14159,4)</code>   | 3<br>3.1416             |
| <code>sum ( iterable [, start])</code>   | 求和                                                                        | <code>sum((1, 2, 3))</code><br><code>sum((1, 2, 3), 44)</code> | 6<br>50                 |

4.7 math 模块和数学函数

Python 的标准模块 `math` 中，提供了许多常用的数学函数，包括三角函数、对数函数和其他通用数学函数。`math` 模块中的函数不支持复数，复数函数位于 `cmath` 模块中（参见 4.8 节）。`math` 模块包含的常量和函数如表 4-13 所示。其中的三角函数以弧度为单位。假设表中示例基于“`from math import *`”。

表 4-13 (1) math 的常量和函数（一）：常量

| 名 称             | 说 明     | 示 例             | 结 果               |
|-----------------|---------|-----------------|-------------------|
| <code>e</code>  | 数学常量 e  | <code>e</code>  | 2.718281828459045 |
| <code>pi</code> | 数学常量 pi | <code>pi</code> | 3.141592653589793 |

表 4-13 (2) math 的常量和函数（二）：数值运算和表示

| 名 称                         | 说 明               | 示 例                                              | 结 果     |
|-----------------------------|-------------------|--------------------------------------------------|---------|
| <code>ceil(x)</code>        | 返回 $\geq x$ 的最小整数 | <code>ceil(1.2)</code> , <code>ceil(-1.6)</code> | (2, -1) |
| <code>copysign(x, y)</code> | 返回符号为 y 的 x 的值    | <code>copysign(1.0, -0.0)</code>                 | -1.0    |

续表

| 名 称                         | 说 明                                               | 示 例                                                                                                                                                                  | 结 果                              |
|-----------------------------|---------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------|
| <code>fabs(x)</code>        | 返回 $x$ 的绝对值                                       | <code>fabs(-1.2)</code>                                                                                                                                              | 1.2                              |
| <code>factorial(x)</code>   | 返回正整数 $x$ 的阶乘                                     | <code>factorial(10)</code>                                                                                                                                           | 3628800                          |
| <code>floor(x)</code>       | 返回 $\leq x$ 的最大整数                                 | <code>floor(1.8)</code> , <code>floor(-2.1)</code>                                                                                                                   | (1, -3)                          |
| <code>fmod(x, y)</code>     | 返回 $x \% y$                                       | <code>fmod(5, 3)</code>                                                                                                                                              | 2.0                              |
| <code>frexp(x)</code>       | 返回 $(m, e)$ , 使得:<br>$x == m * 2 ** e$            | <code>frexp(1024)</code>                                                                                                                                             | (0.5, 11)                        |
| <code>fsum(iterable)</code> | 返回序列之和, 浮点数的计<br>算结果比 <code>sum</code> 更精确        | <code>fsum([.1, .1, .1, .1, .1, .1, .1,<br/>.1, .1, .1])</code><br><code>sum([.1, .1, .1, .1, .1, .1, .1,<br/>.1, .1, .1])</code>                                    | 1.0<br><br>0.9999999999999999    |
| <code>isfinite(x)</code>    | 判断 $x$ 是否为有限值                                     | <code>isfinite(float('Infinity'))</code><br><code>isfinite(float('-Infinity'))</code><br><code>isfinite(float('NaN'))</code><br><code>isfinite(float('12.3'))</code> | False<br>False<br>False<br>True  |
| <code>isinf(x)</code>       | 判断 $x$ 是否为无穷大                                     | <code>isinf(float('Infinity'))</code><br><code>isinf(12.3)</code>                                                                                                    | True<br>False                    |
| <code>isnan(x)</code>       | 判断 $x$ 是否为非数值                                     | <code>isnan(float('NaN'))</code><br><code>isnan(12.3)</code>                                                                                                         | True<br>False                    |
| <code>ldexp(x, i)</code>    | 返回 $x * (2 ** i)$<br>是 <code>frexp(x)</code> 的反函数 | <code>ldexp(2, 10)</code>                                                                                                                                            | 2048.0                           |
| <code>modf(x)</code>        | 返回 $x$ 的小数和整数部分, 结<br>果为元组                        | <code>modf(-12.3)</code>                                                                                                                                             | (-0.30000000000000007,<br>-12.0) |
| <code>trunc(x)</code>       | 将 $x$ 截为最接近 0 的整数                                 | <code>trunc(1.2)</code><br><code>trunc(-2.8)</code>                                                                                                                  | 1<br>-2                          |

表 4-13 (3) math 的常量和函数 (三): 幂和对数运算

| 名 称                                              | 说 明                                             | 示 例                                                    | 结 果                                              |
|--------------------------------------------------|-------------------------------------------------|--------------------------------------------------------|--------------------------------------------------|
| <code>exp(x)</code>                              | 返回 $e ** x$                                     | <code>exp(5)</code>                                    | 148.4131591025766                                |
| <code>expm1(x)</code>                            | 返回 $e ** x - 1$<br>比 <code>exp(x) - 1</code> 精确 | <code>expm1(1e-5)</code><br><code>exp(1e-5) - 1</code> | 1.0000050000166667e-05<br>1.0000050000069649e-05 |
| <code>log(x)</code><br><code>log(x, base)</code> | 返回 $\log_e x$<br>返回 $\log_{base} x$             | <code>log(e)</code><br><code>log(e, 2)</code>          | 1.0<br>1.4426950408889634                        |
| <code>log1p(x)</code>                            | 返回 $\log(1 + x)$                                | <code>log1p(1)</code><br><code>log(2)</code>           | 0.6931471805599453<br>0.6931471805599453         |
| <code>log2(x)</code>                             | 返回 $\log_2 x$                                   | <code>log2(e)</code>                                   | 1.4426950408889634                               |
| <code>log10(x)</code>                            | 返回 $\log_{10} x$                                | <code>log10(100)</code>                                | 2.0                                              |
| <code>pow(x, y)</code>                           | 返回 $x^y$ , 即 $x ** y$                           | <code>pow(2, 8)</code>                                 | 256.0                                            |

表 4-13 (4) math 的常量和函数 (四): 三角函数

| 名 称                  | 说 明         | 示 例                  | 结 果                |
|----------------------|-------------|----------------------|--------------------|
| <code>acos(x)</code> | 返回 $x$ 的反余弦 | <code>acos(1)</code> | 0.0                |
| <code>asin(x)</code> | 返回 $x$ 的正弦  | <code>asin(1)</code> | 1.5707963267948966 |
| <code>atan(x)</code> | 返回 $x$ 的正切  | <code>atan(1)</code> | 0.7853981633974483 |



续表

| 名 称         | 说 明                             | 示 例         | 结 果                |
|-------------|---------------------------------|-------------|--------------------|
| atan2(y, x) | 返回 x 的 atan(y/x)                | atan2(1, 2) | 0.4636476090008061 |
| cos(x)      | 返回 x 的余弦                        | cos(2 * pi) | 1.0                |
| hypot(x, y) | 返回 sqrt(x * x + y * y), 即欧几里德距离 | hypot(3, 4) | 5.0                |
| sin(x)      | 返回 x 的正弦                        | sin(pi/2)   | 1.0                |
| tan(x)      | 返回 x 的正切                        | tan(pi/4)   | 0.9999999999999999 |

表 4-13 (5) math 的常量和函数 (五): 双曲线函数

| 名 称      | 说 明          | 示 例        | 结 果                 |
|----------|--------------|------------|---------------------|
| acosh(x) | 返回 x 的双曲线反余弦 | acosh(1)   | 0.0                 |
| asinh(x) | 返回 x 的双曲线反正弦 | asinh(1)   | 0.8813735870195429  |
| atanh(x) | 返回 x 的双曲线反正切 | atanh(0.1) | 0.1003353477310756  |
| cosh(x)  | 返回 x 的双曲线余弦  | cosh(1)    | 1.5430806348152437  |
| sinh(x)  | 返回 x 的双曲线正弦  | sinh(0.1)  | 0.10016675001984403 |
| tanh(x)  | 返回 x 的双曲线正切  | tanh(0.1)  | 0.09966799462495582 |

表 4-13 (6) math 的常量和函数 (六): 角度弧度转换函数

| 名 称        | 说 明          | 示 例         | 结 果                |
|------------|--------------|-------------|--------------------|
| degrees(x) | 将 x 从弧度转换为角度 | degrees(pi) | 180.0              |
| radians(x) | 将 x 从角度转换为弧度 | radians(90) | 1.5707963267948966 |

表 4-13 (7) math 的常量和函数 (七): 其他函数

| 名 称       | 说 明                 | 示 例                          | 结 果                     |
|-----------|---------------------|------------------------------|-------------------------|
| erf(x)    | 返回 x 的误差函数          | (1.0 + erf(1/sqrt(2.0)))/2.0 | 0.841344746068543       |
| erfc(x)   | 返回 1.0 - erf(x)     | erfc(1)                      | 0.157299207050285       |
| gamma(x)  | 返回 x 的伽玛函数          | gamma(50)                    | 6.082818640342675e + 62 |
| lgamma(x) | 返回 x 的伽玛函数的绝对值的自然对数 | lgamma(50)                   | 144.5657439463449       |

说明:

如果 x 不是 float 数据类型, 则 ceil(x)、floor(x)、trunc(x) 等同于 x 的对象方法 x. \_\_ceil\_\_(), x. \_\_floor\_\_(), x. \_\_trunc\_\_()。

【例 4-2】数学函数的使用示例 (math\_test.py)。输入三角形的三条边长, 求面积、周长、某边长所对应的高、最长边长、最短边长。

```
import math
#三角形三边 a、b、c, 必须满足:a>0 and b>0 and c>0 and a+b>c and a+c>b and b+c>a
a = int(input("请输入边长 a:"))
b = int(input("请输入边长 b:"))
c = int(input("请输入边长 c:"))
if (a>0 and b>0 and c>0 and a+b>c and a+c>b and b+c>a):
```

```
p = (a + b + c) / 2 #周长的一半
area = math.sqrt(p * (p - a) * (p - b) * (p - c)) #面积
perimeter = a + b + c #周长
height_a = 2 * area / a #边长 a 所对应的高
max_side = max(a, b, c) #最长边长
min_side = min(a, b, c) #最短边长
print("三角形的三条边为:{0}、{1}和{2}".format(a, b, c))
print("三角形的面积为:{0:.2f}".format(area))
print("三角形的周长为:{0:.2f}".format(perimeter))
print("边长 A 对应的高为:{0:.2f}".format(height_a))
print("三角形的最长的边为:{0:.2f}".format(max_side))
print("三角形的最短的边为:{0:.2f}".format(min_side))
else:
 print("三条边:{0}、{1}和{2},不能构成三角形".format(a, b, c))
```

运行结果如下：

```
请输入边长 a:3
请输入边长 b:4
请输入边长 c:5
三角形的三条边为:3、4 和 5
三角形的面积为:6.00
三角形的周长为:12.00
边长 A 对应的高为:4.00
三角形的最长的边为:5.00
三角形的最短的边为:3.00
```

## 4.8 cmath 模块和复数数学函数

Python 的标准模块 `cmath` 中，提供了许多用于复数运算的函数。`cmath` 模块包含的常量和函数如表 4-14 所示。假设表中示例基于“`from cmath import *`”。

注：`cmath` 模块中的函数的参数可以为整数、浮点数、复数，或其他可以自动转换为复数或浮点数的对象，即具有 `__complex__()` 或 `__float__()` 方法的对象。

表 4-14 (1) cmath 的常量和函数（一）：常量

| 名 称 | 说 明     | 示 例 | 结 果                |
|-----|---------|-----|--------------------|
| e   | 数学常量 e  | e   | 2. 718281828459045 |
| pi  | 数学常量 pi | pi  | 3. 141592653589793 |

表 4-14 (2) cmath 的常量和函数（二）：转换函数（笛卡儿坐标和极坐标）

| 名 称      | 说 明                                        | 示 例                                    | 结 果                |
|----------|--------------------------------------------|----------------------------------------|--------------------|
| phase(x) | 返回 <code>math.atan2(x.imag, x.real)</code> | <code>phase(complex(-1.0, 0.0))</code> | 3. 141592653589793 |

续表

| 名 称          | 说 明                                                                 | 示 例           | 结 果                                        |
|--------------|---------------------------------------------------------------------|---------------|--------------------------------------------|
| polar(x)     | 返回 (abs(x), phase(x)), 即 (r, phi)                                   | polar(3+4j)   | (5.0, 0.9272952180016122)                  |
| rect(r, phi) | 返回(r, phi)对应的复数, 即 $r * (\cos(\text{phi}) + \sin(\text{phi}) * 1j)$ | rect(1, pi/4) | (0.7071067811865476 + 0.7071067811865475j) |

表 4-14 (3) cmath 的常量和函数 (三): 幂和对数运算

| 名 称      | 说 明             | 示 例         | 结 果                                         |
|----------|-----------------|-------------|---------------------------------------------|
| exp(x)   | 返回 $e^{**x}$    | exp(3+4j)   | (-13.128783081462158 - 15.200784463067954j) |
| log10(x) | 返回 $\log_{10}x$ | log10(3+4j) | (0.6989700043360187 + 0.4027191962733731j)  |
| sqrt(x)  | 返回 x 的平方根       | sqrt(3+4j)  | (2+1j)                                      |

表 4-14 (4) cmath 的常量和函数 (四): 三角函数

| 名 称     | 说 明       | 示 例        | 结 果                                         |
|---------|-----------|------------|---------------------------------------------|
| acos(x) | 返回 x 的反余弦 | acos(3+4j) | (0.9368124611557198 - 2.305509031243477j)   |
| asin(x) | 返回 x 的正弦  | asin(3+4j) | (0.6339838656391766 + 2.305509031243477j)   |
| atan(x) | 返回 x 的反正切 | atan(3+4j) | (1.4483069952314644 + 0.15899719167999918j) |
| cos(x)  | 返回 x 的余弦  | cos(2*pi)  | (1+0j)                                      |
| sin(x)  | 返回 x 的正弦  | sin(pi/2)  | (1+0j)                                      |
| tan(x)  | 返回 x 的正切  | tan(pi/4)  | (0.9999999999999999 + 0j)                   |

表 4-14 (5) cmath 的常量和函数 (五): 双曲线函数

| 名 称      | 说 明          | 示 例        | 结 果                        |
|----------|--------------|------------|----------------------------|
| acosh(x) | 返回 x 的双曲线反余弦 | acosh(1)   | 0j                         |
| asinh(x) | 返回 x 的双曲线反正弦 | asinh(1)   | (0.8813735870195429 + 0j)  |
| atanh(x) | 返回 x 的双曲线反正切 | atanh(0.1) | (0.10033534773107558 + 0j) |
| cosh(x)  | 返回 x 的双曲线余弦  | cosh(1)    | (1.5430806348152437 + 0j)  |
| sinh(x)  | 返回 x 的双曲线正弦  | sinh(0.1)  | (0.10016675001984403 + 0j) |
| tanh(x)  | 返回 x 的双曲线正切  | tanh(0.1)  | (0.09966799462495582 + 0j) |

表 4-14 (6) cmath 的常量和函数 (六): 判别函数

| 名 称         | 说 明                         | 示 例            | 结 果   |
|-------------|-----------------------------|----------------|-------|
| isfinite(x) | 判断 x.real 和 x.imag 是否都为有限值  | isfinite(3+4j) | True  |
| isinf(x)    | 判断 x.real 和 x.imag 是否其一为无穷大 | isinf(3+4j)    | False |
| isnan(x)    | 判断 x.real 和 x.imag 是否其一为非数值 | isnan(3+4j)    | False |

4.9 random 模块和随机函数

random 模块包含各种伪随机数生成函数，以及各种根据概率分布生成随机数的函数。



该模块中大部分函数都基于 random() 函数，该函数使用 Mersenne Twister 生成器在 [0.0, 1.0) 范围内生成一致分布的随机值。

### 4.9.1 种子和随机状态

使用 random 模块函数 seed() 可以设置伪随机数生成器的种子。其基本形式为：

```
random.seed(a=None, version=2)
```

其中 a 为种子。没有指定 a 时，使用系统时间。如果 a 为整数，直接使用。当 a 不为整数且 version = 2 时，则 a 转换为整数；否则使用 a 的哈希值（参见 6.1.1 节）。

同一种子，每次运行产生的随机数相同（故称之为伪随机数）。例如：

```
>>> import random
>>> random.seed(1) #设置种子为 1
>>> for i in range(5): print(random.randint(1,5),end=',') #结果:2,5,1,3,1,
>>> for i in range(5): print(random.randint(1,5),end=',') #结果:4,4,4,4,2,
>>> random.seed(1) #重新设置种子为 1,结果重复
>>> for i in range(5): print(random.randint(1,5),end=',') #结果:2,5,1,3,1,
```

伪随机数生成器运行时存在一个内部状态，可以保存或获取该状态：

```
random.getstate() #获取当前伪随机数生成器状态
random.setstate(state) #设置伪随机数生成器状态
```

例如：

```
>>> random.seed(1)
>>> for i in range(5): print(random.randint(1,5),end=',') #结果:2,5,1,3,1,
>>> s = random.getstate() #获取当前伪随机数生成器状态
>>> for i in range(5): print(random.randint(1,5),end=',') #结果:4,4,4,4,2,
>>> random.setstate(s) #设置伪随机数生成器状态
>>> for i in range(5): print(random.randint(1,5),end=',') #结果:4,4,4,4,2,
```

### 4.9.2 随机整数

random 模块中用于生成随机整数的函数如表 4-15 所示。假设表中示例基于“from random import \*”。

表 4-15 Random 模块中随机整数函数

| 名 称                            | 说 明                                         | 示 例                                                  | 结 果（随机）              |
|--------------------------------|---------------------------------------------|------------------------------------------------------|----------------------|
| randrange(stop)                | 返回随机整数 N,N 属于序列[0, stop)                    | for i in range(10):<br>print(randrange(10),end=',')  | 8,6,7,4,0,9,1,5,4,9, |
| randrange(start, stop[, step]) | 返回随机整数 N,N 属于序列[start, stop, step)          | for i in range(10):<br>print(randrange(1,5),end=',') | 3,4,1,2,3,4,4,4,1,1, |
| randint(a, b)                  | 返回随机整数 N,使得 a <= N <= b,即 randrange(a, b+1) | for i in range(10):<br>print(randint(1,5),end=',')   | 1,2,5,5,1,1,1,2,3,4, |
| getrandbits(k)                 | 返回随机整数 N,使得 N 的位(bit)长为 k                   | for i in range(10):<br>print(getrandbits(2),end=',') | 0,0,1,2,1,3,1,1,1,1, |

【例 4-3】猜数游戏（random\_randint.py）：随机产生一个 1 ~ 100 以内的整数，请用户猜测具体是哪个数。

```
import random
num1 = random.randint(1,100) #随机产生一个1~100 以内的整数
while True:
 guess = int(input('请猜一个整数(1~100):'))
 if guess == num1:
 print('您猜对了!')
 break
 else:
 if guess > num1: print('猜错了。提示:小一点。')
 else: print('猜错了。提示:大一点。')
```

运行结果（每次生成不同的随机数）如下：

```
请猜一个整数(1~100):50
猜错了。提示:大一点。
请猜一个整数(1~100):75
猜错了。提示:小一点。
请猜一个整数(1~100):60
您猜对了!
```

4.9.3 随机系列

random 模块中用于从序列中随机抽取数据的函数如表 4-16 所示。假设表中示例基于：

```
>>> from random import *
>>> seq = ('a','e','i','o','u'); seq1 = [1, 2, 3, 4, 5]
```

表 4-16 Random 模块中随机系列函数

| 名 称                   | 说 明                                | 示 例                                              | 结果（随机）          |
|-----------------------|------------------------------------|--------------------------------------------------|-----------------|
| choice(seq)           | 从非空的序列 seq 中随机返回一个元素               | for i in range(5):<br>print(choice(seq),end=',') | e,e,e,a,e,      |
| sample(population, k) | 从非空的序列 population 随机抽取 k 个元素,返回其列表 | sample(seq,3)                                    | ['i','u','a']   |
| shuffle(x[, random])  | 混排列表。可选的 random 为随机函数,默认为 random() | shuffle(seq1)                                    | [2, 1, 5, 3, 4] |

【例 4-4】混排示例（random\_shuffle.py）：随机生成一副扑克牌。

```
import random
#一幅牌:Club(梅花)、Diamond(方块)、Heart(红桃)、Spade(黑桃)、2-10,J,Q,K,A
cards = ['2C','3C','4C','5C','6C','7C','8C','9C','10C','JC','QC','KC','AC',
 '2D','3D','4D','5D','6D','7D','8D','9D','10D','JD','QD','KD','AD',
 '2H','3H','4H','5H','6H','7H','8H','9H','10H','JH','QH','KH','AH',
 '2S','3S','4S','5S','6S','7S','8S','9S','10S','JS','QS','KS','AS']
random.shuffle(cards) #混排,洗牌
deck1 = [];deck2 = [];deck3 = [];deck4 = [] #初始化四手牌
for i in range(13): #发牌
 deck1.append(cards.pop())
```

```
deck2.append(cards.pop())
deck3.append(cards.pop())
deck4.append(cards.pop())
print(deck1);print(deck2);print(deck3);print(deck4)
```

运行结果（随机）如下：

```
['KC','3H','9H','AH','KH','4H','QC','6S','2H','KD','8C','3D','6H']
['9S','JC','7H','AS','5D','JS','3S','KS','AC','JD','JH','10H','9D']
['7D','4C','7C','8D','QD','2S','6D','2C','4S','QH','QS','10S','5H']
['4D','5C','3C','6C','8H','7S','9C','5S','8S','10D','2D','AD','10C']
```

4.9.4 随机实值分布

random 模块中用于生成随机实值分布的函数如表 4-17 所示。有些函数涉及概率分布的数学知识，请读者参考相关书籍。假设表中示例基于“from random import \*”。

表 4-17 Random 模块中随机实值分布函数

| 名 称                          | 说 明                                           | 示 例                                                      |
|------------------------------|-----------------------------------------------|----------------------------------------------------------|
| random( )                    | 返回范围在[0,1)之间的随机数                              | random( )                                                |
| uniform( a, b)               | 返回[ a,b]或[ b,a]之间的随机数                         | uniform( 1,10)<br>uniform( 10,1)                         |
| triangular(low, high, mode)  | 返回遵循三角分布的随机数。默认 [ low,high] 为[0,1],mode 为其中间值 | triangular( )<br>triangular(1,10)<br>triangular( 1,10,9) |
| betavariate( alpha, beta)    | 返回遵循 Beta 分布的随机数                              | betavariate(1,1)                                         |
| expovariate( lambd)          | 返回遵循指数分布的随机数                                  | expovariate(1)                                           |
| gammavariate( alpha, beta)   | 返回遵循 Gamma 分布的随机数                             | gammavariate (1,1)                                       |
| gauss( mu, sigma)            | 返回遵循 Gauss( 高斯)分布的随机数                         | gauss(1,0.1)                                             |
| lognormvariate( mu, sigma)   | 返回遵循对数正态分布的随机数                                | lognormvariate( 1,0.1)                                   |
| normalvariate( mu, sigma)    | 返回遵循正态分布的随机数                                  | gauss(1,0.1)                                             |
| vonmisesvariate( mu, kappa)  | 返回遵循 von Mises 分布的随机数                         | vonmisesvariate(1,0.1)                                   |
| paretovariate( alpha)        | 返回遵循 Pareto 分布的随机数                            | paretovariate(1)                                         |
| weibullvariate( alpha, beta) | 返回遵循 Weibull 分布的随机数                           | weibullvariate(1,1)                                      |
| gammavariate( alpha, beta)   | 返回遵循 Gamma 分布的随机数                             | gammavariate (1,1)                                       |

4.10 相关模块

4.10.1 标准库中的相关模块

Python 标准库中包括下列数值处理相关模块。  
numbers 模块：数值抽象类，包含类 Complex、Real、Rational、Integral。  
math 模块：数学函数。



cmath 模块：复数运算数学函数。

decimal 模块：高精度数值运算。

fractions 模块：分数运算模块。

random 模块：随机数模块。

## 4.10.2 数值运算模块 NumPy

Python 扩展模块 NumPy 提供了更高效的数值处理功能。NumPy 模块主要提供数组和矩阵处理功能，还包括高级功能，如傅里叶变换等。NumPy 的官网地址为：<http://www.scipy.org>。

## 4.10.3 科学计算模块 SciPy

Python 扩展模块 SciPy 提供了用于科学计算的功能。SciPy 模块包括统计、优化、整合、线性代数、傅里叶变换、信号和图像处理、常微分方程求解器等功能。SciPy 的官网地址为：<http://www.scipy.org>。

## 4.11 复习题

### 一、单选题

- Python 语句 `print(type(1J))` 的输出结果是\_\_\_\_\_。  
A. `<class 'complex'>`                      B. `<class 'int'>`  
C. `<class 'float'>`                      D. `<class 'dict'>`
- Python 语句 `print(type(1/2))` 的输出结果是\_\_\_\_\_。  
A. `<class 'int'>`                      B. `<class 'number'>`  
C. `<class 'float'>`                      D. `<class 'double'>`
- Python 语句 `print(type(1//2))` 的输出结果是\_\_\_\_\_。  
A. `<class 'int'>`                      B. `<class 'number'>`  
C. `<class 'float'>`                      D. `<class 'double'>`
- Python 语句 `a = 121 + 1.21; print(type(a))` 的输出结果是\_\_\_\_\_。  
A. `<class 'int'>`                      B. `<class 'float'>`  
C. `<class 'double'>`                      D. `<class 'long'>`
- Python 语句 `print(0xA + 0xB)` 的输出结果是\_\_\_\_\_。  
A. `0xA + 0xB`                      B. `A + B`  
C. `0xA0xB`                      D. `21`
- Python 语句 `x = 'car'; y = 2; print(x + y)` 的输出结果是\_\_\_\_\_。  
A. 语法错                      B. `2`  
C. `'car2'`                      D. `'carcar'`
- Python 表达式 `sqrt(4) * sqrt(9)` 的值为\_\_\_\_\_。  
A. `36.0`                      B. `1296.0`  
C. `13.0`                      D. `6.0`

8. 关于 Python 中的复数,下列说法错误的是\_\_\_\_\_。
- A. 表示复数的语法是 `real + image j`      B. 实部和虚部都是浮点数  
C. 虚部必须后缀 `j`,且必须是小写      D. 方法 `conjugate` 返回复数的共轭复数

## 二、填空题

- Python 表达式 `4.5/2` 的值为\_\_\_\_\_。
- Python 表达式 `4.5//2` 的值为\_\_\_\_\_。
- Python 表达式 `4.5%2` 的值为\_\_\_\_\_。
- Python 语句 `print(round(3.84,0),floor(15.5))` 的输出结果是\_\_\_\_\_。
- Python 语句 `print(int('20',16),int('101',2))` 的输出结果是\_\_\_\_\_。
- Python 语句 `print(hex(16),bin(10))` 的输出结果是\_\_\_\_\_。
- Python 语句 `print(2.5.as_integer_ratio())` 的输出结果是\_\_\_\_\_。
- Python 语句 `print(float.as_integer_ratio(1.5))` 的输出结果是\_\_\_\_\_。
- Python 语句 `print(gcd(12,16),divmod(7,3))` 的输出结果是\_\_\_\_\_。
- Python 语句 `print((2-3j).conjugate()*complex(2,3))` 的输出结果是\_\_\_\_\_。
- Python 语句 `print(sum((1,2,3)),sum((1,2,3),10))` 的输出结果是\_\_\_\_\_。
- Python 语句 `print(abs(-3.2),abs(1-2j))` 的输出结果是\_\_\_\_\_。
- Python 的标准随机数生成器模块是\_\_\_\_\_。
- 数学表达式  $\sin 15^\circ + \frac{e^x - 5x}{\sqrt{x^2 + 1}} - \ln(3x)$  的 Python 表达式为: \_\_\_\_\_。

- 数学表达式  $\frac{\frac{1}{2}cd}{\frac{c+d}{a+b} - \frac{c-d}{c+d}} - \frac{4\pi}{a+b}$  的 Python 表达式为: \_\_\_\_\_。

## 三、思考题

1. 阅读下面 Python 程序。请问输出结果是什么?

```
from decimal import *
ctx = getcontext(); ctx.prec = 2; print(Decimal('1.78'))
print(Decimal('1.78') + 0); ctx.rounding = ROUND_UP
print(Decimal('1.65') + 0); print(Decimal('1.62') + 0); print(Decimal('-1.45') + 0)
print(Decimal('-1.42') + 0); ctx.rounding = ROUND_HALF_UP
print(Decimal('1.65') + 0); print(Decimal('1.62') + 0); print(Decimal('-1.45') + 0)
ctx.rounding = ROUND_HALF_DOWN; print(Decimal('1.65') + 0); print(Decimal('-1.45') + 0)
```

2. 阅读下面 Python 语句。请问输出结果是什么? 程序的功能是什么?

```
import random
a = random.randint(100,999) #随机产生一个3位整数
b = (a % 10) * 100 + (a // 10 % 10) * 10 + a // 100
print("原数 =", a, ", 变换后 =", b)
```

## 4.12 上机实践

1. 输入直角三角形的两个直角边,求三角形的周长和面积,以及两个锐角的度数。结

果均保留一位小数。运行效果如图 4-1 所示。

|                    |                     |
|--------------------|---------------------|
| 请输入直角三角形的直角边A(>0): | 4                   |
| 请输入直角三角形的直角边B(>0): | 4                   |
| 直角三角形三边分别为:        | a=4.0, b=4.0, c=5.7 |
| 三角形的周长 =           | 13.7, 面积 = 8.0      |
| 三角形两个锐角的度数分别为:     | 45.0 和 45.0         |

图 4-1 直角三角形运行效果

提示:

(1) `math.asin` 函数返回正弦值为指定数字的弧度; `math.acos` 函数返回余弦值为指定数字的弧度。

(2) 将弧度转换为角度的公式为:  $\text{角度} = \frac{\text{弧度} * 180}{\pi}$ 。

2. 编程产生三个 0 ~ 100 之间 (包含 0 和 100) 的随机数 a、b 和 c, 按从小到大的顺序排序。运行效果如图 4-2 所示 (其中, a、b 和 c 的值随机生成)。

|            |                  |
|------------|------------------|
| 原始值:       | a=97, b=89, c=99 |
| (方法一) 升序值: | a=89, b=97, c=99 |
| (方法二) 升序值: | a=89, b=97, c=99 |

图 4-2 三个数比较大小运行效果

提示:

(1) 方法一: 先 a 和 b 比较, 使得  $a < b$ ; 然后 a 和 c 比较, 使得  $a < c$ , 此时 a 最小; 最后 b 和 c 比较, 使得  $b < c$ 。

(2) 方法二: 利用 `max` 函数和 `min` 函数求 a、b、c 三个数中最大数、最小数, 而三个数之和减去最大数和最小数就是中间数。

3. 编程计算有固定工资收入的党员每月所缴纳的党费。月工资收入 400 元及以下者, 交纳月工资总额的 0.5%; 月工资收入 401 元到 600 元者, 交纳月工资总额的 1%; 月工资收入在 601 元到 800 元者, 交纳月工资总额的 1.5%; 月工资收入在 801 元到 1500 元者 (税后), 交纳月工资收入的 2%; 月工资收入在 1500 元以上 (税后) 者, 交纳月工资收入的 3%。运行效果如图 4-3 所示。

$$\text{党费 } f = \begin{cases} 0.5\% * \text{salary} & \text{salary} \leq 400 \\ 1\% * \text{salary} & 401 \leq \text{salary} \leq 600 \\ 1.5\% * \text{salary} & 601 \leq \text{salary} \leq 800 \\ 2\% * \text{salary} & 801 \leq \text{salary} \leq 1500 \\ 3\% * \text{salary} & \text{salary} > 1500 \end{cases}$$

|                         |
|-------------------------|
| 请输入有固定工资收入的党员的月工资: 1200 |
| 月工资 = 1200, 交纳党费 = 24.0 |

图 4-3 党费交纳运行效果

4. 编程实现模拟袖珍计算器, 要求输入两个操作数和一个操作符 (+、-、\*、/、%), 根据操作符输出运算结果。特别注意 “/” 和 “%” 运算符的零除异常问题。运行效果如图 4-4 所示。



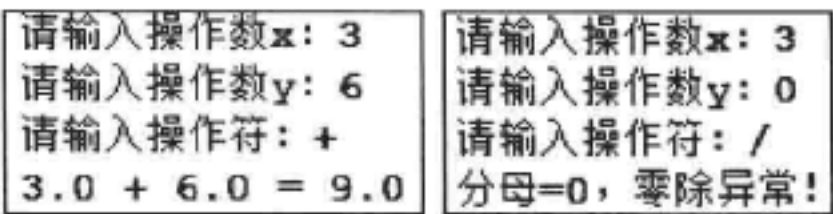


图 4-4 袖珍计算器运行效果

5. 输入三角形的三条边  $a$ 、 $b$ 、 $c$ ，判断此三边是否可以构成三角形。若能，进一步判断三角形的性质：等边、等腰、直角或其他三角形。本题的判断准则参见表 4-18。运行效果如图 4-5 所示。

表 4-18 各类三角形的判断准则

| 形 状   | 满 足 条 件                                                        |
|-------|----------------------------------------------------------------|
| 三角形   | 任意两边之和大于第三边                                                    |
| 等边三角形 | 三边均相等的三角形                                                      |
| 等腰三角形 | 只有两边相等的三角形                                                     |
| 直角三角形 | 勾股定理：斜边 <sup>2</sup> = 直角边 1 <sup>2</sup> + 直角边 2 <sup>2</sup> |

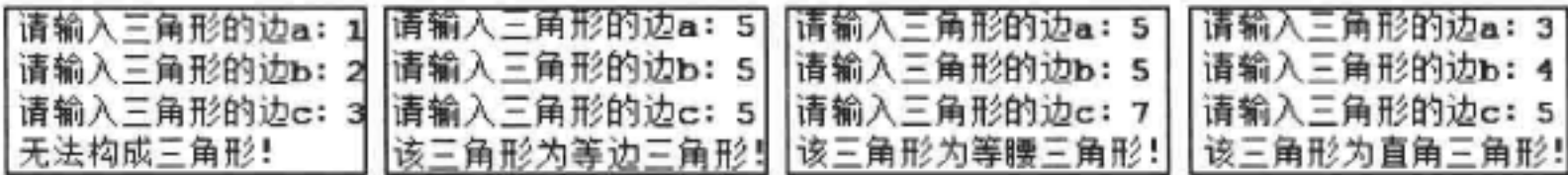


图 4-5 判断三角形运行效果

6. 编程实现鸡兔同笼问题。已知在同一个笼子里总共有  $h$  只鸡和兔，鸡和兔的总脚数为  $f$  只，其中， $h$  和  $f$  由用户输入，求鸡和兔各有多少只？要求使用两种方法：一是求解方程，二是利用循环进行枚举测试。运行效果如图 4-6 所示。

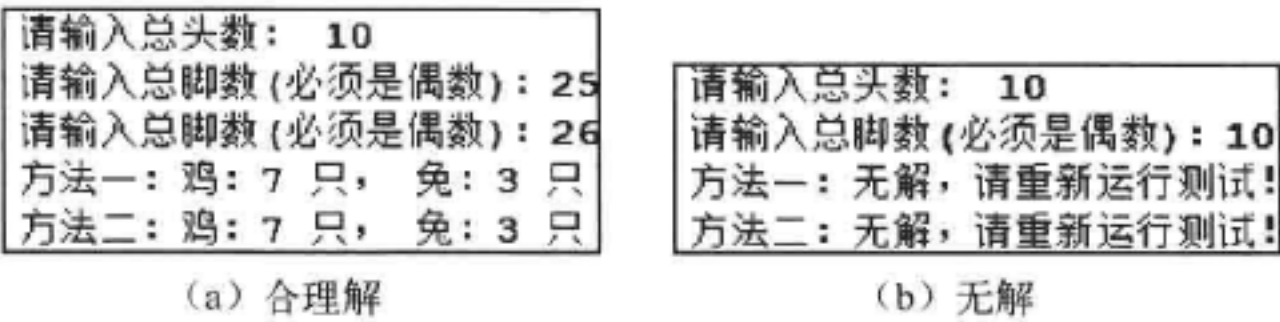


图 4-6 鸡兔同笼运行效果

提示：

- (1) 已知鸡和兔的总头数为  $h$ ，总脚数为  $f$ 。假设鸡有  $c$  只，兔有  $r$  只。
- (2) 方法一：求解方程法。由公式：

$$\begin{cases} c + r = h \\ 2c + 4r = f \end{cases}$$

解得：

$$\begin{cases} r = \frac{f}{2} - h \\ c = h - r \end{cases}$$

由公式推得，鸡和兔的总脚数  $f$  必须是偶数，并且鸡和兔的只数必须是非负整数。

- (3) 方法二：利用循环进行枚举测试。鸡的只数  $c$  取值范围为： $0 \sim h$ ，兔的数量  $r$  为

$(h-c)$ ，如果满足条件  $(2c+4r==f)$ ，则求得解。

7. 输入任意实数  $x$ ，计算  $e^x$  的近似值，直到最后一项的绝对值小于  $10^{-6}$  为止。运行效果如图 4-7 所示。

$$e^x \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

```
请输入x: 2
Pow(e,x) = 7.3890560703259105
```

图 4-7  $e^x$  运行效果

8. 输入任意实数  $a(a \geq 0)$ ，用迭代法求  $x = \sqrt{a}$ ，要求计算的相对偏差小于  $10^{-6}$ 。运行效果如图 4-8 所示。求平方根的迭代公式为：

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right)$$

```
2 的算术平方根= 1.4142135623746899
3 的算术平方根= 1.7320508075688772
```

图 4-8 平方根运行效果

9. 我国汉代有位大将，名叫韩信。他每次集合部队，只要求部下先后按 1~3、1~5、1~7 报数，然后再报告一下各队每次报数的余数，他就知道到了多少人。他的这种巧妙算法，人们称为鬼谷算，也叫隔墙算，或称为韩信点兵，外国人还称它为“中国余数定理”。即：有一个数，用 3 除余 2，用 5 除余 3，用 7 除余 2，请问 0~1000 中这样的数有哪些？运行效果如图 4-9 所示。

```
0~1000中用3除余2，用5除余3，用7除余2的数有：
23 128 233 338 443 548 653 758 863 968
```

图 4-9 韩信点兵运行效果

10. 一球从 100 米的高度自由落下，每次落地后反弹回原高度的一半，再落下。求小球在第 10 次落地时，共经过多少米？第 10 次反弹多高？运行效果如图 4-10 所示。

```
小球在第10次落地时，共经过199.80米
第10次反弹0.20米
```

图 4-10 自由落体运行效果

11. 猴子吃桃问题。猴子第一天摘下若干个桃子，当天吃掉一半多一个；第二天接着吃了剩下的桃子的一半多一个；以后每天都吃了前一天剩下的桃子的一半多一个。到第 8 天想吃桃时，发现只剩一个桃子了。请问猴子第一天共摘了多少个桃子？提示：这是一个递推问题。假设第  $n$  天的桃子数为  $P_n$ ，前一天（第  $n-1$ ）天的桃子数为  $P_{n-1}$ ，则： $P_n = \frac{1}{2}P_{n-1} - 1$ ，也即： $P_{n-1} = 2(P_n + 1)$ 。现在已知第 8 天（ $n=8$ ）的桃子数  $P_8 = 1$ ，根据公式得第 7 天的桃子数  $P_7 = 4$ ，依次类推，可以求得第 1 天  $P_1$  的桃子

数。运行效果如图 4-11 所示。

|            |     |
|------------|-----|
| 第 8 天桃子数为： | 1   |
| 第 7 天桃子数为： | 4   |
| 第 6 天桃子数为： | 10  |
| 第 5 天桃子数为： | 22  |
| 第 4 天桃子数为： | 46  |
| 第 3 天桃子数为： | 94  |
| 第 2 天桃子数为： | 190 |
| 第 1 天桃子数为： | 382 |

图 4-11 猴子吃桃运行效果

12. 计算  $S_n = 1 + 11 + 111 + 1111 + \cdots + 1111\cdots111$  (最后一项是  $n$  个 1)。提示 (第 1 项  $T_1 = 1$ ; 第 2 项  $T_2 = T_1 * 10 + 1$ ;  $\cdots$ ; 第  $n$  项  $= T_{(n-1)} * 10 + 1$ )。  $n$  是一个由随机数产生的 1 ~ 10 (包括 1 和 10) 中的一个正整数。



# 第 5 章 系列：元组、列表和字符串

序列类型（bytes、bytearray、list、str 和 tuple）是 Python 内置的组合数据类型，可以实现复杂数据的处理。

本章要点：

- ◆ 系列的定义和基本操作；
- ◆ 元组的定义和基本操作；
- ◆ 列表的定义和基本操作；
- ◆ 字符串的定义和基本操作；
- ◆ 字节系列的定义和基本操作。

## 5.1 系列的基本操作

序列（sequence）表示可通过索引下标访问的可迭代（iterable）对象。系列对象定义了特殊方法\_\_len\_\_(), 故可使用内置函数 len()测试其长度。len(obj)调用 obj 的方法\_\_len\_\_()。

Python 内置的序列数据类型包括：元组（tuple）、列表（list）、字符串（str）和字节数据（bytes 和 bytearray）。

### 5.1.1 系列的索引访问操作

序列对象定义了一个特殊方法\_\_getitem\_\_(), 故可通过整数下标访问系列 s 的元素。

```
s[i] #访问系列 s 在索引 i 处的元素
如果 i 越界，则导致 IndexError；如果 i 不是整数，则导致 TypeError。例如：
>>> s = 'abc '
>>> s['a '] #TypeError: string indices must be integers
>>> s[3] #IndexError: string index out of range
```

s[i] 调用系列对象 s 的方法\_\_getitem\_\_(self, key)，返回其位于下标 i 处的元素。系列 s 的索引下标示意图如图 5-1 所示。

|         |       |          |       |       |
|---------|-------|----------|-------|-------|
| s[-5]   | s[-4] | s[-3]    | s[-2] | s[-1] |
| 'bonus' | -228  | 'purple' | '100' | 19.84 |
| s[0]    | s[1]  | s[2]     | s[3]  | s[4]  |

图 5-1 系列 s 的索引下标示意图

**【例 5-1】** 系列的索引访问示例。

|                  |                               |                       |                   |
|------------------|-------------------------------|-----------------------|-------------------|
| >>> s = 'abcdef' | >>> t = ('a','e','i','o','u') | >>> lst = [1,2,3,4,5] | >>> b = b'ABCDEF' |
| >>> s[0]         | >>> t[0]                      | >>> lst[0]            | >>> b[0]          |
| 'a'              | 'a'                           | 1                     | 65                |
| >>> s[2]         | >>> t[1]                      | >>> lst               | >>> b[1]          |
| 'c'              | 'e'                           | [1, 2, 3, 4, 5]       | 66                |
| >>> s[-1]        | >>> t[-1]                     | >>> lst[2] = 'a'      | >>> b[-1]         |
| 'f'              | 'u'                           | >>> lst[-2] = 'b'     | 70                |
| >>> s[-3]        | >>> t[-5]                     | >>> lst               | >>> b[-2]         |
| 'd'              | 'a'                           | [1, 2, 'a', 'b', 5]   | 69                |

## 5.1.2 系列的切片操作

通过切片 (slice) 操作, 可以截取系列 s 的一部分。切片操作的基本形式为:

`s[i:j]` 或者 `s[i:j:k]`

其中, i 为开始下标 (包含 `s[i]`), j 为结束下标 (不包含 `s[j]`), k 为步长。如果省略 i, 则从下标 0 开始; 如果省略 j, 则直到结束为止; 如果省略 k, 则步长为 1。

**注:** 下标也可以为负数。如果截取范围内没有数据, 则返回空元组; 如果超过下标范围, 不报错。

**【例 5-2】** 系列的切片操作示例。

|                  |                               |                       |                   |
|------------------|-------------------------------|-----------------------|-------------------|
| >>> s = 'abcdef' | >>> t = ('a','e','i','o','u') | >>> lst = [1,2,3,4,5] | >>> b = b'ABCDEF' |
| >>> s[1:3]       | >>> t[-2:-1]                  | >>> lst[:2]           | >>> b[2:2]        |
| 'bc'             | ('o',)                        | [1, 2]                | b''               |
| >>> s[3:10]      | >>> t[-2:]                    | >>> lst[:1] = []      | >>> b[0:1]        |
| 'def'            | ('o', 'u')                    | >>> lst               | b'A'              |
| >>> s[8:2]       | >>> t[-99:-5]                 | [2, 3, 4, 5]          | >>> b[1:2]        |
| ''               | ()                            | >>> lst[:2]           | b'B'              |
| >>> s[:]         | >>> t[-99:-3]                 | [2, 3]                | >>> b[2:2]        |
| 'abcdef'         | ('a', 'e')                    | >>> lst[:2] = 'a'     | b''               |
| >>> s[:2]        | >>> t[::]                     | >>> lst[1:] = 'b'     | >>> b[-1:]        |
| 'ab'             | ('a', 'e', 'i', 'o', 'u')     | >>> lst               | b'F'              |
| >>> s[:2]        | >>> t[1:-1]                   | ['a', 'b']            | >>> b[-2:-1]      |
| 'ace'            | ('e', 'i', 'o')               | >>> del lst[:1]       | b'E'              |
| >>> s[::-1]      | >>> t[1:2]                    | >>> lst               | >>> b[0:len(b)]   |
| 'fedcba'         | ('e', 'o')                    | [5]                   | b'ABCDEF'         |

对于复杂的数据, 切片操作 `s[i:j]` 以及 `s[i:j:k]` 的可读性比较差。使用内置的 slice 对象, 可以保存用于切片的下标信息:

`slice(start, stop, step)`

其中, start、stop 和 step 分别对应于开始下标、结束下标 (不包含) 和步长。

slice 对象 slice1 包括下列属性和方法。

slice1.start、slice1.step、slice1.stop: 属性。

slice1.indices(len): 方法, 根据 start、stop、step 和系列长度, 返回元组 (start, stop, step)。

例如：

```
>>> s = 'abcdef'; slice1 = slice(1, len(s), 2)
>>> slice1 #slice(1, 6, 2)
>>> s[slice1] #'bdf'
>>> slice1.start, slice1.stop, slice1.step #(1, 6, 2)
>>> slice1.indices(100) #(1, 6, 2)
>>> slice1.indices(4) #根据长度,自动调整 stop 的值:(1, 4, 2)
```

### 5.1.3 系列的连接和重复操作

通过连接操作符 `+`，可以连接两个系列（`s1` 和 `s2`），形成一个新的系列对象；通过重复操作符 `*`，可以重复一个系列 `n` 次（`n` 为正整数）。系列连接和重复操作的基本形式为：

`s1 + s2` 或者 `s * n` 或者 `n * s`

连接操作符 `+` 和重复操作符 `*` 也支持复合赋值运算，即：`+=` 和 `*=`。

【例 5-3】系列的连接和重复操作示例。

|                                      |                                          |                                            |                                       |
|--------------------------------------|------------------------------------------|--------------------------------------------|---------------------------------------|
| <code>&gt;&gt;&gt; s1 = 'abc'</code> | <code>&gt;&gt;&gt; t1 = (1,2)</code>     | <code>&gt;&gt;&gt; lst1 = [1,2]</code>     | <code>&gt;&gt;&gt; b1 = b'ABC'</code> |
| <code>&gt;&gt;&gt; s2 = 'xyz'</code> | <code>&gt;&gt;&gt; t2 = ('a','b')</code> | <code>&gt;&gt;&gt; lst2 = ['a','b']</code> | <code>&gt;&gt;&gt; b2 = b'XYZ'</code> |
| <code>&gt;&gt;&gt; s1 + s2</code>    | <code>&gt;&gt;&gt; t1 + t2</code>        | <code>&gt;&gt;&gt; lst1 + lst2</code>      | <code>&gt;&gt;&gt; b1 + b2</code>     |
| <code>'abcxyz'</code>                | <code>(1, 2, 'a', 'b')</code>            | <code>[1, 2, 'a', 'b']</code>              | <code>b'ABCXYZ'</code>                |
| <code>&gt;&gt;&gt; s1 * 3</code>     | <code>&gt;&gt;&gt; t1 * 2</code>         | <code>&gt;&gt;&gt; 2 * lst2</code>         | <code>&gt;&gt;&gt; b1 * 3</code>      |
| <code>'abcabcabc'</code>             | <code>(1, 2, 1, 2)</code>                | <code>['a', 'b', 'a', 'b']</code>          | <code>b'ABCABCABC'</code>             |
| <code>&gt;&gt;&gt; s1 += s2</code>   | <code>&gt;&gt;&gt; t1 += t2</code>       | <code>&gt;&gt;&gt; lst1 += lst2</code>     | <code>&gt;&gt;&gt; b1 += b2</code>    |
| <code>&gt;&gt;&gt; s1</code>         | <code>&gt;&gt;&gt; t1</code>             | <code>&gt;&gt;&gt; lst1</code>             | <code>&gt;&gt;&gt; b1</code>          |
| <code>'abcxyz'</code>                | <code>(1, 2, 'a', 'b')</code>            | <code>[1, 2, 'a', 'b']</code>              | <code>b'ABCXYZ'</code>                |
| <code>&gt;&gt;&gt; s2 *= 2</code>    | <code>&gt;&gt;&gt; t2 *= 2</code>        | <code>&gt;&gt;&gt; lst2 *= 2</code>        | <code>&gt;&gt;&gt; b2 *= 2</code>     |
| <code>&gt;&gt;&gt; s2</code>         | <code>&gt;&gt;&gt; t2</code>             | <code>&gt;&gt;&gt; lst2</code>             | <code>&gt;&gt;&gt; b2</code>          |
| <code>'xyzxyz'</code>                | <code>('a', 'b', 'a', 'b')</code>        | <code>['a', 'b', 'a', 'b']</code>          | <code>b'XYZXYZ'</code>                |

### 5.1.4 系列的成员关系操作

可以通过下列方式之一判断一个元素 `x` 是否存在于系列 `s` 中：

|                                   |                                                                             |
|-----------------------------------|-----------------------------------------------------------------------------|
| <code>x in s</code>               | #如果为 <code>True</code> ,则表示存在                                               |
| <code>x not in s</code>           | #如果为 <code>True</code> ,则表示不存在                                              |
| <code>s.count(x)</code>           | #返回 <code>x</code> 在 <code>s</code> (指定范围 <code>[start,end)</code> ) 中出现的次数 |
| <code>s.index(x[, i[, j]])</code> | #返回 <code>x</code> 在 <code>s</code> (指定范围 <code>[i, j)</code> ) 中第一次出现的下标   |

其中，指定范围 `[i, j)`，从下标 `i`（包括，默认为 0）开始，到下标 `j` 结束（不包括，默认为 `len(s)`）。

对于 `s.index(value, [start, [stop]])` 方法，如果找不到时，则导致 `ValueError`。例如：

```
>>> 'To be or not to be, this is a question'.index('123') #ValueError: substring not found
```

【例 5-4】系列中元素的存在性判断示例。



|                               |                         |                       |                       |
|-------------------------------|-------------------------|-----------------------|-----------------------|
| >>> s = 'Good, better, best!' | >>> t = ('r', 'g', 'b') | >>> lst = [1,2,3,2,1] | >>> b = b'Oh, Jesus!' |
| >>> 'o' in s                  | >>> 'r' in t            | >>> 1 in lst          | >>> b'O' in b         |
| True                          | True                    | True                  | True                  |
| >>> 'g' not in s              | >>> 'y' not in t        | >>> 2 not in lst      | >>> b'o' not in b     |
| True                          | True                    | False                 | True                  |
| >>> s.count('e')              | >>> t.count('r')        | >>> lst.count(1)      | >>> b.count(b's')     |
| 3                             | 1                       | 2                     | 2                     |
| >>> s.index('e', 10)          | >>> t.index('g')        | >>> lst.index(3)      | >>> b.index(b's')     |
| 10                            | 1                       | 2                     | 6                     |

### 5.1.5 系列的比较运算操作

两个系列支持比较运算符（<、<=、==、!=、>=、>），字符串比较运算按顺序逐个元素进行比较。

【例 5-5】系列的比较运算示例。

|                 |                  |                        |                  |
|-----------------|------------------|------------------------|------------------|
| >>> s1 = 'abc'  | >>> t1 = (1,2)   | >>> s1 = ['a','b']     | >>> b1 = b'abc'  |
| >>> s2 = 'abc'  | >>> t2 = (1,2)   | >>> s2 = ['a','b']     | >>> b2 = b'abc'  |
| >>> s3 = 'abcd' | >>> t3 = (1,2,3) | >>> s3 = ['a','b','c'] | >>> b3 = b'abcd' |
| >>> s4 = 'cba'  | >>> t4 = (2,1)   | >>> s4 = ['c','b','a'] | >>> b4 = b'ABCD' |
| >>> s1 > s4     | >>> t1 < t4      | >>> s1 < s2            | >>> b1 < b2      |
| False           | True             | False                  | False            |
| >>> s2 <= s3    | >>> t1 <= t2     | >>> s1 <= s2           | >>> b1 <= b2     |
| True            | True             | True                   | True             |
| >>> s1 == s2    | >>> t1 == t3     | >>> s1 == s2           | >>> b1 == b2     |
| True            | False            | True                   | True             |
| >>> s1 != s3    | >>> t1 != t2     | >>> s1 != s3           | >>> b1 >= b3     |
| True            | False            | True                   | False            |
| >>> 'a' > 'A'   | >>> t1 >= t3     | >>> s1 >= s3           | >>> b3 != b4     |
| True            | False            | False                  | True             |
| >>> 'a' >= ''   | >>> t4 > t3      | >>> s4 > s3            | >>> b4 > b3      |
| True            | True             | True                   | False            |

### 5.1.6 系列的排序操作

通过内置函数 sorted()，可以返回序列的排序列表。通过类 reversed 构造函数，可以返回系列的反序的迭代器。

**sorted(iterable, key=None, reverse=False)** #返回系列的排序列表

其中，key 是用于计算比较键值的函数（带 1 个参数），例如：key = str.lower；如果 reverse = True，则反向排序。

【例 5-6】系列的排序操作示例。

|                  |                                |                                 |
|------------------|--------------------------------|---------------------------------|
| >>> s1 = 'axd'   | >>> sorted(s2)                 | >>> s3 = 'abAC'                 |
| >>> sorted(s1)   | [1, 2, 4]                      | >>> sorted(s3, key = str.lower) |
| ['a', 'd', 'x']  | >>> sorted(s2, reverse = True) | ['a', 'A', 'b', 'C']            |
| >>> s2 = (1,4,2) | [4, 2, 1]                      |                                 |

### 5.1.7 系列长度、最大值、最小值、求和

通过内置函数 `len()`、`max()`、`min()`，可以获取系列的长度、元素最大值、元素最小值。内置函数 `sum()` 可获取列表或元组各元素之和；如果有非数值元素，则导致 `TypeError`；对于字符串（`str`）和字节数据（`bytes`），也将导致 `TypeError`。例如：

```
>>> t1 = (1,2,3,4)
>>> sum(t1) #输出:10
>>> t2 = (1,'a',2)
>>> sum(t2) #TypeError: unsupported operand type(s) for + : 'int' and 'str'
>>> s = '1234'
>>> sum(s) #TypeError: unsupported operand type(s) for + : 'int' and 'str'
```

【例 5-7】系列的长度、最大值、最小值操作示例。

|                   |                  |                       |                 |
|-------------------|------------------|-----------------------|-----------------|
| >>> s = 'abcdefg' | >>> t = (10,2,3) | >>> lst = [1,2,9,5,4] | >>> b = b'ABCD' |
| >>> len(s)        | >>> len(t)       | >>> len(lst)          | >>> len(b)      |
| 7                 | 3                | 5                     | 4               |
| >>> max(s)        | >>> max(t)       | >>> max(lst)          | >>> max(b)      |
| 'g'               | 10               | 9                     | 68              |
| >>> min(s)        | >>> min(t)       | >>> min(lst)          | >>> min(b)      |
| 'a'               | 2                | 1                     | 65              |
| >>> s2 = ''       | t2 = ()          | lst2 = []             | >>> b2 = b''    |
| >>> len(s2)       | >>> len(t2)      | >>> len(lst2)         | >>> len(b2)     |
| 0                 | 0                | 0                     | 0               |

### 5.1.8 内置函数 `all()` 和 `any()`

通过内置函数 `all()` 和 `any()`，可以判断序列的元素是否全部和部分为 `True`。

**`all(iterable)`** #如果序列的所有值都为 `True`，返回 `True`；否则，返回 `False`

**`any(iterable)`** #如果序列的任意值为 `True`，返回 `True`；否则，返回 `False`

例如：

|                    |                    |
|--------------------|--------------------|
| >>> any((1, 2, 0)) | >>> all([1, 2, 0]) |
| True               | False              |

### 5.1.9 系列拆封

#### 1. 变量个数和系列长度相等

使用赋值语句，可以将系列值拆封，赋值给多个变量：

变量 1，变量 2，…，变量 n = 系列或可迭代对象

变量个数和系列的元素个数不一致时，将导致 `ValueError`。例如：

```
>>> a, b = (1, 2)
>>> a, b # (1, 2)
>>> a, b, c = (1, 2) #ValueError: need more than 2 values to unpack
>>> data = (1001, '张三', (80, 79, 92))
```

```
>>> sid, name, scores = data
>>> scores #(80, 79, 92)
>>> sid, name, (chinese, math, english) = data
>>> math #79
```

## 2. 变量个数和系列长度不等

如果系列长度未知，可使用 \* 元组变量，将多个值作为元组赋值给元组变量。一个赋值语句中，\* 变量只允许出现一次，否则导致 SyntaxError。例如：

```
>>> first, * middles, last = range(10)
>>> middles #[1, 2, 3, 4, 5, 6, 7, 8]
>>> first, second, third, * lasts = range(10)
>>> lasts #[3, 4, 5, 6, 7, 8, 9]
>>> * firsts, last3, last2, last1 = range(10)
>>> firsts #[0, 1, 2, 3, 4, 5, 6]
>>> first, * middles, last = sorted([70, 85, 89, 88, 86, 95, 89]) #去掉最高和最低分,求平均值
>>> sum(middles)/len(middles) #87.4
```

## 3. 使用临时变量\_

如果只需要部分数据，系列其他位置可以使用临时变量\_。例如：

```
>>> _, b, _ = (1, 2, 3)
>>> b #2
>>> record = ('Zhangsan ', 'szhang@ abc. com ', '021 - 62232333 ', '13912349876 ')
>>> name, _, * phones = record
>>> phones #['021 - 62232333 ', '13912349876 ']
```

# 5.2 元组

元组 (tuple) 是一组有序系列，包含 0 个或多个对象引用。元组和列表十分类似，但元组是不可变的对象，即不能修改、添加或删除元组中的项目，但可以访问元组中的项目。

## 5.2.1 元组的定义

元组采用圆括号中用逗号分隔的项目定义。圆括号可以省略。其基本形式如下：

**x1, [x2, ..., xn]**

或者

**(x1, [x2, ..., xn])**

其中，x1, x2, ..., xn 为任意对象。注意：如果元组中只有一个项目时，后面的逗号不能省略，这是因为 Python 解释器把 (x1) 解释为 x1，例如 (1) 解释为整数 1，(1,) 解释为元组。

元组也可以通过创建 tuple 对象来创建。其基本形式为：

**tuple()**                    #创建一个空列表

**tuple(iterable)**          #创建一个列表，包含的项目为可枚举对象 iterable 中的元素



【例 5-8】创建元组对象示例。

|             |         |                 |                     |                    |
|-------------|---------|-----------------|---------------------|--------------------|
| >>> 1,2,3   | >>> 1,  | >>> 'a','b','c' | >>> tuple()         | >>> tuple('abc')   |
| (1, 2, 3)   | (1,)    | ('a', 'b', 'c') | ()                  | ('a', 'b', 'c')    |
| >>> () #空元组 | >>> (1) | >>> 'a',        | >>> tuple(range(3)) | >>> tuple([1,2,3]) |
| ()          | 1       | ('a',)          | (0, 1, 2)           | (1, 2, 3)          |

## 5.2.2 元组的基本操作

元组支持系列的基本操作，包括索引访问、切片操作、连接操作、重复操作、成员关系操作、比较运算操作，以及求元组长度、最大值、最小值等。

【例 5-9】元组的基本操作示例。

|                            |             |                                               |
|----------------------------|-------------|-----------------------------------------------|
| >>> t1 = (1,2,1)           | >>> max(t1) | >>> sum(t2)                                   |
| >>> t2 = ('a','x','y','b') | 2           | Traceback (most recent call last):            |
| >>> len(t1)                | >>> min(t2) | File "<pyshell#30>", line 1, in <module>      |
| 3                          | 'a'         | sum(t2)                                       |
| >>> len(t2)                | >>> sum(t1) | TypeError: unsupported operand type(s) for +: |
| 4                          | 4           | 'int' and 'str'                               |

## 5.3 列表

列表 (list) 是一组有序项目的数据结构。创建一个列表后，可以访问、修改、添加或删除列表中的项目，即列表是可变的数据类型。Python 没有数组，可以使用列表代替。

### 5.3.1 列表的定义

列表采用方括号中用逗号分隔的项目定义。其基本形式如下：

[x1, [x2, ..., xn]]

列表也可以通过创建 list 对象来创建。其基本形式如下：

list() #创建一个空列表

list(iterable) #创建一个列表,包含的项目为可枚举对象 iterable 中的元素

【例 5-10】创建列表对象。

|         |             |                    |                   |
|---------|-------------|--------------------|-------------------|
| >>> []  | >>> [1,2,3] | >>> list((1,2,3))  | >>> list('abc')   |
| []      | [1, 2, 3]   | [1, 2, 3]          | ['a', 'b', 'c']   |
| >>> [1] | >>> list()  | >>> list(range(3)) | >>> list([1,2,3]) |
| [1]     | []          | [0, 1, 2]          | [1, 2, 3]         |

### 5.3.2 列表的基本操作

列表支持系列的基本操作，包括索引访问、切片操作、连接操作、重复操作、成员关系操作、比较运算操作，以及求列表长度、最大值、最小值等。

列表是可变对象，故可以改变列表对象中元素的值，也可通过 del 删除某元素。

s[下标] = x #设置列表元素,x 为任意对象

del s[下标] #删除列表元素

列表是可变对象，故可改变其切片的值，也可通过 del 删除切片。

```
s[i:j] = x #设置列表内容,x 为任意对象,也可以是元组、列表
dd s[i:j] #移去列表一系列元素,等同于 s[i:j] = []
s[i:j] = [] #移去列表一系列元素
```

【例 5-11】列表的基本操作示例。

```
>>> s = [1,2,3,4,5,6]
>>> s[1] = 'a'
>>> s
[1, 'a', 3, 4, 5, 6]
>>> s[2] = []
>>> s
[1, 'a', [], 4, 5, 6]
>>> del s[3]
>>> s
[1, 'a', [], 5, 6]
>>> s[:2]
[1, 'a']
>>> s[2:3] = []
>>> s
[1, 'a', 5, 6]
>>> s[:1] = []
>>> s
['a', 5, 6]
>>> s[:2] = 'b'
>>> s
['b', 6]
>>> del s[:1]
>>> s
[6]
```

5.3.3 list 对象的方法

列表是可变对象，其包含的主要方法如表 5-1 所示。假设表中的示例基于 s = [1,3,2]。

表 5-1 列表对象的主要方法

| 方 法            | 说 明                                             | 示 例                                                                                |
|----------------|-------------------------------------------------|------------------------------------------------------------------------------------|
| s.append(x)    | 把对象 x 追加到列表 s 尾部                                | s.append('a') #s = [1, 3, 2, 'a']<br>s.append([1,2]) #s = [1, 3, 2, 'a', [1, 2]]   |
| s.clear()      | 删除所有元素。相当于 del s[:]                             | s.clear() #s = []                                                                  |
| s.copy()       | 拷贝列表                                            | s1 = s.copy() #s = s1 = [1,2,3]<br>id(s),id(s1) #(43280824, 42613096)              |
| s.extend(t)    | 把序列 t 附加到 s 尾部                                  | s.extend([4]) #s = [1, 3, 2, 4]<br>s.extend('ab') #s = [1, 3, 2, 4, 'a', 'b']      |
| s.insert(i, x) | 在下标 i 位置插入对象 x                                  | s.insert(1,4) #s = [1, 4, 3, 2]<br>s.insert(8,5) #s = [1, 4, 3, 2, 5]              |
| s.pop([i])     | 返回并移除下标 i 位置对象,省略 i 时为最后对象。若超出下标,将导致 IndexError | s.pop()#输出 2。s = [1, 3]<br>s.pop(0)#输出 1。s = [3]                                   |
| s.remove(x)    | 移除列表中第一个出现的 x。若对象不存在,将导致 ValueError             | s.remove(1)#s = [3, 2]<br>s.remove('a') #ValueError: list.remove(x): x not in list |
| s.reverse()    | 列表反转                                            | s.reverse()#s = [2, 3, 1]                                                          |
| s.sort()       | 列表排序                                            | s.sort() #s = [1, 2, 3]                                                            |

5.3.4 列表解析表达式

使用列表解析，可以简单高效地处理一个可迭代对象，并生成结果列表。列表解析表达式的形式如下：

```
[expr for i1 in 序列 1... for iN in 序列 N] #迭代序列里所有内容,并计算生成列表
[expr for i1 in 序列 1... for iN in 序列 N if cond_expr] #按条件迭代,并计算生成列表
```

表达式 expr 使用每次迭代内容 i<sub>1</sub>...i<sub>N</sub>，计算生成一个列表。如果指定了条件表达式 cond\_expr，则只有满足条件的元素参与迭代。例如：

```
>>> [i * * 2 for i in range(10)] #输出:[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> [i for i in range(10) if i%2 == 0] #输出:[0, 2, 4, 6, 8]
>>> [(x, y, x * y) for x in range(1, 4) for y in range(1, 4) if x >= y]
[(1, 1, 1), (2, 1, 2), (2, 2, 4), (3, 1, 3), (3, 2, 6), (3, 3, 9)]
```

### 5.3.5 列表作为队列和栈

队列（Queue）是先进先出的系列（First In First Out, FIFO），即最先添加的元素，是最先弹出的元素；栈（Stack）是后进先出的队列（Last In First Out, LIFO），即最后添加（push）的元素，是最先弹出（pop）的元素。

向列表最后位置添加元素和从最后位置移除元素非常方便和高效，故使用 list，可以快速高效地实现栈。list.append() 方法对应于入栈操作（push），list.pop() 对应于出栈操作（pop）。

列表可以实现队列（Queue），但并不适合。因为从列表的头部移除一个元素，列表中的所有元素都需要移动位置，所以效率不高。可以使用 collections 模块中的 deque 对象。例如：

```
>>> stack = list()
>>> stack.append('Apple'); stack.append('Banana'); stack.append('Blueberry')
>>> stack.pop() #输出:'Blueberry'
>>> stack.pop() #输出:'Banana'
>>> stack.pop() #输出:'Apple'
>>> stack.pop() #IndexError: pop from empty list
```

## 5.4 字符串

字符串（str）是一个有序的字符集合，即字符序列。Python 内置数据类型 str，用于字符串处理。Python 中没有独立的字符数据类型，字符即长度为 1 的字符串。

### 5.4.1 Unicode 和字符常量

#### 1. 字符常量

使用单引号（例如'a'）或双引号（例如"张"）括起来的字符，是字符常量，Python 解释器自动创建 str 型对象实例。例如：

```
>>> 'A', "张" #结果:('A', '张')
```

#### 2. Unicode 码

Python 3 字符默认为 16 位 Unicode 编码，ASCII 码是 Unicode 编码的子集。例如：字符'A'的 ASCII 码为 65，对应的八进制为 101，对应的十六进制为 41。

使用内置函数 ord() 可以把字符转换为对应的 Unicode 码；使用内置函数 chr() 可以把十进制数转换为对应的字符。例如：

```
>>> ord('张'), chr(24352) #结果:(24352, '张')
```

#### 3. 转义字符

特殊符号（不可打印字符），可以使用转义序列表示。转义序列以反斜杠开始，紧跟一



个字母，如“\n”（新行）和“\t”（制表符）。如果字符串中希望包含反斜杠，则它前面必须还有另一个反斜杠。

使用转义字符后跟 Unicode 编码也可以表示字符。例如：

```
>>> 'A', chr(65), '\101', '\x41' #结果:('A', 'A', 'A', 'A')
```

Python 转义字符如表 5-2 所示。

表 5-2 特殊符号的转义序列

| 转义序列 | 字 符      | 转义序列 | 字 符                 |
|------|----------|------|---------------------|
| \'   | 单引号      | \n   | 换行 (LF)             |
| \"   | 双引号      | \r   | 回车 (CR)             |
| \\   | 反斜杠      | \t   | 水平制表符 (HT)          |
| \a   | 响铃 (BEL) | \v   | 垂直制表符 (VT)          |
| \b   | 退格 (BS)  | \ooo | 八进制 Unicode 码对应的字符  |
| \f   | 换页 (FF)  | \xhh | 十六进制 Unicode 码对应的字符 |

5.4.2 字符串常量

1. 字符串

使用单引号或双引号括起来的内容，是字符串常量，Python 解释器自动创建 str 型对象实例。Python 字符串可以用以下 4 种方式定义。

- (1) 单引号 ( ' ' )：包含在单引号中的字符串，其中可以包含双引号。
- (2) 双引号 ( " " )：包含在双引号中的字符串，其中可以包含单引号。
- (3) 三单引号 ( ' ' ' ' )：包含在三单引号中的字符串，可以跨行。
- (4) 三双引号 ( " " " " )：包含在三双引号中的字符串，可以跨行。

注：两个紧邻的字符串，如果中间只有空格分隔，则自动拼接为一个字符串。例如：

```
>>> 'Blue ' 'Sky ' #结果:'BlueSky '
```

【例 5-12】字符串常量示例。

|                                                                                                                                                                                                       |                                                                                                                                                                      |                                                                                                                   |                                                                                                                                            |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>&gt;&gt;&gt; 'abc ' 'abc ' &gt;&gt;&gt; 'abc\x\" "abc 'x '" &gt;&gt;&gt; 'abc"x"' 'abc"x"' &gt;&gt;&gt; x = 'c:\Python33 ' &gt;&gt;&gt; x 'c:\Python33 ' &gt;&gt;&gt; print(x) c:\Python33</pre> | <pre>&gt;&gt;&gt; "xyz" 'xyz ' &gt;&gt;&gt; "x\tyz" 'x\tyz ' &gt;&gt;&gt; print("x\tyz") x yz &gt;&gt;&gt; print("x'y'z") x'y'z &gt;&gt;&gt; print("x\ny") x y</pre> | <pre>&gt;&gt;&gt; s1 = "'a \tb \tc \td"' &gt;&gt;&gt; s1 'a\n\tb\n\tc\n\td ' &gt;&gt;&gt; print(s1) a b c d</pre> | <pre>&gt;&gt;&gt; s2 = """ She said, " Yes!" """ &gt;&gt;&gt; s2 '\nShe said;\n" Yes!"\n ' &gt;&gt;&gt; print(s2)  She said: " Yes!"</pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|

## 2. 原始字符串

使用 `r` 或 `R` 的字符串称为原始字符串，其中包含的任何字符都不进行转义。

```
>>> print(r'c:\n\t') #输出:c:\n\t
```

## 3. Unicode 字符串

使用 `u` 或 `U` 的字符串称为 Unicode 字符串。Python 3 默认为 Unicode 字符串。

```
>>> u'abc' #输出:'abc'
```

### 5.4.3 创建 str 对象

创建 `str` 类型的对象实例的基本形式如下：

```
str(object='') #创建 str 对象,默认为空字符串
```

通过创建 `str` 对象，可以把任意对象转换为 `str` 对象，返回 `object.__str__()`，如果对象没有定义 `__str__()`，则返回 `repr(object)`。

【例 5-13】创建 `str` 对象示例。

|                               |                                  |                                    |                                           |
|-------------------------------|----------------------------------|------------------------------------|-------------------------------------------|
| <pre>&gt;&gt;&gt; str()</pre> | <pre>&gt;&gt;&gt; str(123)</pre> | <pre>&gt;&gt;&gt; str('abc')</pre> | <pre>&gt;&gt;&gt; str(complex(1,2))</pre> |
| <pre>"</pre>                  | <pre>'123'</pre>                 | <pre>'abc'</pre>                   | <pre>'(1+2j)'</pre>                       |

### 5.4.4 字符串的基本操作

字符串支持系列的基本操作，包括索引访问、切片操作、连接操作、重复操作、成员关系操作、比较运算操作，以及求字符串长度、最大值、最小值等。

通过 `len(s)`，可以获取字符串 `s` 的长度；如果其长度为 0，则为空字符串。

【例 5-14】字符串的基本操作示例。

|                                       |                                    |                                    |                                 |
|---------------------------------------|------------------------------------|------------------------------------|---------------------------------|
| <pre>&gt;&gt;&gt; s1 = 'abcxyz'</pre> | <pre>&gt;&gt;&gt; s1[3:]</pre>     | <pre>&gt;&gt;&gt; s1 &gt; s2</pre> | <pre>'123123123'</pre>          |
| <pre>&gt;&gt;&gt; len(s1)</pre>       | <pre>'xyz'</pre>                   | <pre>True</pre>                    | <pre>&gt;&gt;&gt; max(s1)</pre> |
| <pre>6</pre>                          | <pre>&gt;&gt;&gt; s2 = '123'</pre> | <pre>&gt;&gt;&gt; 3 * s2</pre>     | <pre>'z'</pre>                  |

### 5.4.5 str 对象的方法

使用 `str` 对象提供的方法，可以实现常用的字符串处理功能。`str` 对象是不可变对象，故调用方法返回的字符串是新创建的对象。`str` 对象的方法有两种调用方式。例如：

```
>>> s = 'abc'
>>> s.upper() #字符串对象 s 的方法。结果:'ABC'
>>> str.upper(s) #str 类方法,字符串 s 作为参数。结果:'ABC'
```

#### 1. 字符串类型判断

`str.isalnum()`：是否全为字母或数字。

`str.isalpha()`：是否全字母。

`str.isdecimal()`：是否只包含十进制数字字符。

`str.isdigit()`：是否全数字（0~9）。

`str.isidentifier()`：是否是合法标识。

`str.islower()`：是否全小写。

`str.isupper()`：是否全大写。

str.isnumeric(): 是否只包含数字字符。

str.isprintable(): 是否只包含可打印字。

str.isspace(): 是否只包含空白字。

str.istitle(): 是否为标题, 即各单词首字母大写。

**【例 5-15】** 字符串类型判断示例。

|                           |                  |                    |                  |
|---------------------------|------------------|--------------------|------------------|
| >>> s1 = 'yellow ribbon ' | >>> s1.islower() | >>> s4.isalnum()   | >>> s1.isdigit() |
| >>> s2 = 'Pacal Case '    | True             | True               | False            |
| >>> s3 = '123 '           | >>> s2.isupper() | >>> s3.isnumeric() | >>> s2.istitle() |
| >>> s4 = 'iPhone6 '       | False            | True               | True             |

## 2. 大小写转换

str.capitalize(): 转换为首字母大写, 其余小写。

str.lower(): 转换为小写。

str.upper(): 转换为大写。

str.swapcase(): 大小写互换。

str.title(): 转换为各单词首字母大写。

str.casefold(): 转换为大小写无关字符串比较的格式字符串。

**【例 5-16】** 字符串大小写转换示例。

|                        |                     |                   |                   |
|------------------------|---------------------|-------------------|-------------------|
| >>> s1 = 'red car '    | >>> s1.capitalize() | >>> s3.upper()    | >>> s1.title()    |
| >>> s2 = 'Pacal Case ' | 'Red car '          | 'PYTHON33 '       | 'Red Car '        |
| >>> s3 = 'python33 '   | >>> s2.lower()      | >>> s2.swapcase() | >>> s4.casefold() |
| >>> s4 = 'iPhone6 '    | 'pacal case '       | 'pACAL cASE '     | 'iphone6 '        |

## 3. 填充、空白和对齐

str.strip([chars]): 去两边空格, 也可指定要去除的字符列表。

str.lstrip([chars]): 去左边空格, 也可指定要去除的字符列表。

str.rstrip([chars]): 去右边空格, 也可指定要去除的字符列表。

str.zfill(width): 左填充, 使用 0 填充到 width 长度。

str.center(width[, fillchar]): 两边填充, 使用填充字符 fillchar (默认空格) 填充到 width 长度。

str.ljust(width[, fillchar]): 左填充, 使用填充字符 fillchar (默认空格) 填充到 width 长度。

str.rjust(width[, fillchar]): 右填充, 使用填充字符 fillchar (默认空格) 填充到 width 长度。

str.expandtabs([tabsize]): 将字符串中的制表符 (tab) 扩展为若干个空格, tabsize 默认为 8。

**【例 5-17】** 字符串填充、空白和对齐示例。

|                  |                 |                      |                      |
|------------------|-----------------|----------------------|----------------------|
| >>> s1 = '123 '  | >>> s2.strip()  | >>> s1.zfill(5)      | >>> s1.ljust(5)      |
| >>> s2 = ' 123 ' | '123 '          | '00123 '             | '123 '               |
| >>> len(s2)      | >>> s2.lstrip() | >>> s1.center(5, '') | >>> s1.rjust(5, '0') |
| 6                | '123 '          | '123 '               | '00123 '             |

## 4. 测试、查找和替换

str.startswith(prefix[, start[, end]]): 是否以 prefix 开头。



`str.endswith(suffix[, start[, end]])`: 是否以 `suffix` 结尾。

`str.count(sub[, start[, end]])`: 返回指定字符串出现的次数。

`str.index(sub[, start[, end]])`: 搜索指定字符串, 返回下标, 无则导致 `ValueError`。

`str.rindex(sub[, start[, end]])`: 从右边开始搜索指定字符串, 返回下标, 无则导致 `ValueError`。

`str.find(sub[, start[, end]])`: 搜索指定字符串, 返回下标, 没有则返回 `-1`。

`str.rfind(sub[, start[, end]])`: 从右边开始搜索指定字符串, 返回下标, 没有则返回 `-1`。

`str.replace(old, new[, count])`: 替换 `old` 为 `new`, 可选 `count` 为替换次数。

其中, 可选指定范围 `[start, end)`, 从下标 `start` (包括 `start`, 默认为 `0`) 开始, 到下标 `end` 结束 (不包括 `end`, 默认为 `len(s)`)。

**【例 5-18】** 字符串测试、查找和替换示例。

|                                        |                                       |                                            |                                            |
|----------------------------------------|---------------------------------------|--------------------------------------------|--------------------------------------------|
| <code>&gt;&gt;&gt; s1 = '123 '</code>  | <code>&gt;&gt;&gt; s2.strip()</code>  | <code>&gt;&gt;&gt; s1.zfill(5)</code>      | <code>&gt;&gt;&gt; s1.ljust(5)</code>      |
| <code>&gt;&gt;&gt; s2 = ' 123 '</code> | <code>'123 '</code>                   | <code>'00123 '</code>                      | <code>'123 '</code>                        |
| <code>&gt;&gt;&gt; len(s2)</code>      | <code>&gt;&gt;&gt; s2.lstrip()</code> | <code>&gt;&gt;&gt; s1.center(5, '')</code> | <code>&gt;&gt;&gt; s1.rjust(5, '0')</code> |
| <code>6</code>                         | <code>'123 '</code>                   | <code>'123 '</code>                        | <code>'00123 '</code>                      |

## 5. 拆分和组合

`str.split(sep = None, maxsplit = -1)`: 按指定字符 (默认为空格) 分割字符串, 返回列表。 `maxsplit` 为最大分割次数, 默认 `-1`, 无限制。

`str.rsplit(sep = None, maxsplit = -1)`: 从右侧按指定字符分割字符串, 返回列表。

`str.partition(sep)`: 根据分隔符 `sep` 分割字符串为两部分, 返回元组 (`left`, `sep`, `right`)。

`str.rpartition(sep)`: 根据分隔符 `sep` 从右侧分割字符串为两部分, 返回元组 (`left`, `sep`, `right`)。

`str.splitlines([keepends])`: 按行分割字符串, 返回列表。

`str.join(iterable)`: 组合 `iterable` 中的各元素成字符串, 若包含非字符串元素, 则导致 `TypeError`。

**【例 5-19】** 字符串拆分和组合示例。

|                                                 |                                                |                                               |                                          |
|-------------------------------------------------|------------------------------------------------|-----------------------------------------------|------------------------------------------|
| <code>&gt;&gt;&gt; s1 = 'one,two,three '</code> | <code>&gt;&gt;&gt; s1.partition(',')</code>    | <code>&gt;&gt;&gt; s2.splitlines()</code>     | <code>&gt;&gt;&gt; s4 = ':'</code>       |
| <code>&gt;&gt;&gt; s1.split(',')</code>         | <code>('one', ',', 'two,three')</code>         | <code>['abc', '123', 'xyz']</code>            | <code>&gt;&gt;&gt; s4.join(s3)</code>    |
| <code>['one', 'two', 'three']</code>            | <code>&gt;&gt;&gt; s1.rpartition(',')</code>   | <code>&gt;&gt;&gt; s2.splitlines(True)</code> | <code>'a:b:c'</code>                     |
| <code>&gt;&gt;&gt; s1.rsplit(',', 1)</code>     | <code>('one,two', ',', 'three')</code>         | <code>['abc\n', '123\n', 'xyz']</code>        | <code>&gt;&gt;&gt; s4.join('123')</code> |
| <code>['one,two', 'three']</code>               | <code>&gt;&gt;&gt; s2 = 'abc\n123\nxyz'</code> | <code>&gt;&gt;&gt; s3 = ('a','b','c')</code>  | <code>'1:2:3'</code>                     |

## 6. 翻译和转换

`static str.maketrans(x[, y[, z]])`: 创建用于 `translate` 的转换表。

`str.translate(map)`: 根据 `map` 转换。

**【例 5-20】** 字符串翻译和转换示例。

|                                                                        |                                                                        |
|------------------------------------------------------------------------|------------------------------------------------------------------------|
| <code>&gt;&gt;&gt; table1 = str.maketrans('1234567', '一二三四五六日')</code> | <code>&gt;&gt;&gt; weeks = {'1': 'M 一', '2': 'T 二', '3': 'W 三',</code> |
| <code>&gt;&gt;&gt; s1 = '1 3 4 9'</code>                               | <code>'4': 'T 四', '5': 'F 五', '6': 'S 六', '7': 'S 日'}</code>           |
| <code>&gt;&gt;&gt; s1.translate(table1)</code>                         | <code>&gt;&gt;&gt; table2 = str.maketrans(weeks)</code>                |
| <code>'一 三 四 9'</code>                                                 | <code>&gt;&gt;&gt; s1.translate(table2)</code>                         |
|                                                                        | <code>'M 一 W 三 T 四 9'</code>                                           |

**【例 5-21】** 字符串的使用示例 (str\_count.py): 输入任意字符串, 统计其中元音字母 ('a'、'e'、'i'、'o'、'u', 不区分大小写) 出现的次数和频率。

```
s1 = input('请输入字符串:') #The quick brown fox jumps over the lazy dog '
s2 = s1.upper() #转换为大写
countall = len(s1)
counta = s2.count('A');counte = s2.count('E');counti = s2.count('I')
counto = s2.count('O');countu = s2.count('U')
print('所有字母的总数为:', countall)
print('元音字母出现的次数和频率分别为:')
print('A: {0} \t {1:2.2f} %'.format(counta, counta/countall * 100))
print('E: {0} \t {1:2.2f} %'.format(counte, counte/countall * 100))
print('I: {0} \t {1:2.2f} %'.format(counti, counti/countall * 100))
print('O: {0} \t {1:2.2f} %'.format(counto, counto/countall * 100))
print('U: {0} \t {1:2.2f} %'.format(countu, countu/countall * 100))
```

运行结果如下:

请输入字符串:The quick brown fox jumps over the lazy dog

所有字母的总数为: 43

元音字母出现的次数和频率分别为:

A:1     2.33%

E:3     6.98%

I:1     2.33%

O:4     9.30%

U:2     4.65%

## 5.4.6 字符串编码

默认情况下, Python 字符串采用 utf-8 编码。创建字符串时, 也可以指定其编码方式:

**str(object = b'', encoding = 'utf-8', errors = 'strict')** #按指定编码, 根据字节码对象创建 str 对象

其中, object 为字节码对象 (bytes 或 bytearray); encoding 为编码; errors 为错误控制。该构造函数的结果, 等同于 bytes 对象 b 的对象方法:

**b.decode(encoding, errors)** #把字节码对象 b 解码为对应编码的字符串

对应的, 也可以把字符串对象 s 编码为字节码对象:

**s.encode(encoding = "utf-8", errors = "strict")** #把字符串对象 s 编码为字节码对象

**【例 5-22】** 字符串编码和解码示例。

|                                                                                                                                                                                                       |                                                                                                                                                                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>&gt;&gt;&gt; s1 = 'Sample!例子!' &gt;&gt;&gt; b1 = s1.encode(encoding = 'cp936') &gt;&gt;&gt; b1 b'Sample!\xc0\xfd\xd7\xd3\xa3\xa1' &gt;&gt;&gt; b1.decode(encoding = 'cp936') 'Sample! 例子!'</pre> | <pre>&gt;&gt;&gt; b1.decode()#默认解码 utf-8, 出错 Traceback (most recent call last):   File "&lt;pyshell#56&gt;", line 1, in &lt;module&gt;     b1.decode() UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc0 in position 7; invalid start byte</pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 5.4.7 字符串格式化

### 1. %元算符形式

Python 支持类似于 C 语言的 printf 格式化输出。采用如下形式：

格式字符串 `%(值1, 值2, ...)` #兼容 Python 2 的格式, 不建议使用

格式化字符串与 C 语言的 printf 格式化字符串基本相同。格式字符串由固定文本和格式说明符混合组成。格式说明符的语法如下：

`%[(key)][flags][width][.precision][Length]type`

其中，key（可选）为映射键（适用于映射的格式化，例如 `'%(lang)s'`）；flags（可选）为修改输出格式的字符集；width（可选）为最小宽度，如果为 `*`，则使用下一个参数值；precision（可选）为精度，如果为 `*`，则使用下一个参数值；Length 为修饰符（`h`、`l` 或 `L`，可选），Python 忽略该字符；type 为格式化类型字符。例如：

```
>>> '结果:%f'% 88 #输出:'结果:88.000000'
>>> '姓名:%s, 年龄:%d, 体重:%3.2f'% ('张三', 20, 53)
'姓名:张三, 年龄:20, 体重:53.00'
>>> '%(lang)s has %(num)03d quote types. '% ('lang': 'Python', 'num': 2)
'Python has 002 quote types. '
>>> '%0*. *f'% (10, 5, 88) #输出:'0088.00000'
```

格式字符串的标志符（flags）如下。

- ☞ `'0'`：数值类型格式化结果左边用零填充。
- ☞ `'-'`：结果左对齐。
- ☞ `' '`：对于正值，结果中将包括一个前导空格。
- ☞ `'+'`：数值结果总是包括一个符号（`'+'`或`'-'`）。
- ☞ `'#'`：使用另一种转换方式。

格式化类型字符（type）如下。

- ☞ `%d` 或 `%i`：有符号整数（十进制）。
- ☞ `%o`：有符号整数（八进制）。
- ☞ `%u`：同 `%d`，已过时。
- ☞ `%x`：有符号整数（十六进制，小写字符），标志符为 `'#'` 时，输出前缀 `'0x'`。
- ☞ `%X`：有符号整数（十六进制，大写字符），标志符为 `'#'` 时，输出前缀 `'0X'`。
- ☞ `%e`：浮点数字（科学计数法，小写 `e`），标志符为 `'#'` 时，总是带小数点。
- ☞ `%E`：浮点数字（科学计数法，大写 `E`），标志符为 `'#'` 时，总是带小数点。
- ☞ `%f` 或 `%F`：浮点数字（用小数点符号），标志符为 `'#'` 时，总是带小数点。
- ☞ `%g`：浮点数字（根据值的大小采用 `%e` 或 `%f`），标志符为 `'#'` 时，总是带小数点，保留后面 0。
- ☞ `%G`：浮点数字（根据值的大小采用 `%E` 或 `%F`），标志符为 `'#'` 时，总是带小数点，保留后面 0。



- ☞ %c: 字符及其 ASCII 码。
- ☞ %r: 字符串, 使用转换函数 `repr()`, 标志符为 '#' 且指定 precision 时, 截取 precision 个字符。
- ☞ %s: 字符串, 使用转换函数 `str()`, 标志符为 '#' 且指定 precision 时, 截取 precision 个字符。
- ☞ %a: 字符串, 使用转换函数 `ascii()`, 标志符为 '#' 且指定 precision 时, 截取 precision 个字符。
- ☞ %%: 百分号标记。

## 2. format 内置函数

format 内置函数的基本形式如下:

```
format(value) #等同于 str(value)
format(value, format_spec) #等同于 type(value).__format__(format_spec)
```

格式化说明符 (format\_spec) 的基本格式如下:

```
[[fill]align][sign][#][0][width][,][.precision][type]
```

其中, fill (可选) 为填充字符, 可以为除 {} 外的任何字符; align 为对齐方式, 包括 "<" (左对齐)、">" (右对齐)、"=" (填充位于符号和数字之间, 如 '+000000120')、"^" (居中对齐); sign (可选) 为符号字符, 包括: "+" (正数)、"-" (负数)、" " (正数带空格, 负数带 -); '#' (可选) 使用另一种转换方式; '0' (可选) 数值类型格式化结果左边用零填; width (可选) 是最小宽度; precision (可选) 是精度; type 是格式化类型字符。格式化类型字符 (type) 如下。

- ☞ b: 二进制数。
- ☞ c: 字符, 整数转换为对应的 unicode。
- ☞ d: 十进制数。
- ☞ o: 八进制数。
- ☞ x: 十六进制数, 小写字符, 标志符为 '#' 时, 输出前缀 '0x'。
- ☞ X: 十六进制数, 大写字符, 标志符为 '#' 时, 输出前缀 '0X'。
- ☞ e: 浮点数字 (科学计数法, 小写 e), 标志符为 '#' 时, 总是带小数点。
- ☞ E: 浮点数字 (科学计数法, 大写 E), 标志符为 '#' 时, 总是带小数点。
- ☞ f 或 F: 浮点数字 (用小数点符号), 标志符为 '#' 时, 总是带小数点。
- ☞ g: 浮点数字 (根据值的大小采用 e 或 f), 标志符为 '#' 时, 总是带小数点, 保留后面 0。
- ☞ G: 浮点数字 (根据值的大小采用 E 或 F), 标志符为 '#' 时, 总是带小数点, 保留后面 0。
- ☞ n: 数值, 使用本地千位分隔符。
- ☞ s: 字符串, 使用转换函数 `str()`, 标志符为 '#' 且指定 precision 时, 截取 precision 个字符。
- ☞ %: 百分比。

例如：

```
>>> format(81.2, "0.5f") #输出:'81.20000 '
>>> format(81.2, "%") #输出:'8120.000000% '
```

### 3. 字符串的 format 方法

字符串 format 方法的基本形式如下：

```
str.format(格式字符串,值1,值2,...) #类方法
格式字符串.format(值1,值2,...) #对象方法
格式字符串.format_map(mapping)
```

格式字符串由固定文本和格式说明符混合组成。格式说明符的语法如下：

```
{[索引和键]:format_spec}
```

其中，可选的索引对应于要格式化参数值的位置，可选的键对应于要格式化的映射的键；格式化说明符（format\_spec）同 format 内置函数。例如：

```
>>> "int:{0:d};hex:{0:x};oct:{0:o};bin:{0:b}".format(100)
'int: 100;hex: 64;oct: 144;bin: 1100100 '
>>> "int:{0:d};hex:{0:#x};oct:{0:#o};bin:{0:#b}".format(100)
'int: 100;hex: 0x64;oct: 0o144;bin: 0b1100100 '
>>> '{2},{1},{0}'.format('a','b','c') #输出:'c, b, a '
>>> str.format_map({'name:s'},{'age:d'},{'weight:3.2f'},{'name':'Mary','age':20,'weight':49})
'Mary,20,49.00 '
```

## 5.5 字节系列

字节系列（bytes 和 bytearray）是由 8 位字节数据组成的系列数据类型，即  $0 \leq x < 256$  的整数系列。Python 内置的字节系列数据类型包括：bytes（不可变对象）、bytearray（可变对象）和 memoryview。

### 5.5.1 bytes 常量

使用字母 b 加单引号或双引号括起来的内容，是 bytes 常量。Python 解释器自动创建 bytes 型对象实例。bytes 常量与字符串定义方式类似。

- ① 单引号（b''）：包含在单引号中的字符串，其中可以包含双引号。
- ② 双引号（b" "）：包含在双引号中的字符串，其中可以包含单引号。
- ③ 三单引号（b''' '''）：包含在三单引号中的字符串，可以跨行。
- ④ 三双引号（b""" """）：包含在三双引号中的字符串，可以跨行。

注：引号中只能包含 ASCII 码字符，否则导致 SyntaxError。例如：

```
>>> b'张' #SyntaxError: bytes can only contain ASCII literal characters
```

【例 5-23】 bytes 常量示例。

|                        |                     |                     |                  |
|------------------------|---------------------|---------------------|------------------|
| >>> b'abc'             | >>> b"xyz"          | >>> s1 = "a"        | >>> s2 = b" "    |
| b'abc'                 | b'xyz'              | \tb                 | She said:        |
| >>> b'abc\x\"          | >>> b"x\tyz"        | \tc                 | "Yes!"           |
| b"abc'x'"              | b'x\tyz'            | \td"                | " "              |
| >>> b'abc"x'"          | >>> print(b"x\tyz") | >>> s1 = b"a"       | >>> s2           |
| b'abc"x'"              | b'x\tyz'            | \tb                 | b'\nShe          |
| >>> x = b'c:\Python33' | >>> print(b"x'y'z") | \tc                 | said:\n"Yes!"\n' |
| >>> x                  | b"x'y'z"            | \td"                | >>> print(s2)    |
| b'c:\Python33'         | >>> print(b"x\ny")  | >>> s1              | b'\nShe          |
|                        | b'x\ny'             | b'a\n\tb\n\tc\n\td' | said:\n"Yes!"\n' |

## 5.5.2 创建 bytes 对象

创建 bytes 类型的对象实例的基本形式如下:

|                                             |                                 |
|---------------------------------------------|---------------------------------|
| <b>bytes()</b>                              | #创建空 bytes 对象                   |
| <b>bytes(n)</b>                             | #创建长度为 n(整数)的 bytes 对象,各字节为 0   |
| <b>bytes(iterable)</b>                      | #创建 bytes 对象,使用 iterable 中的字节整数 |
| <b>bytes(object)</b>                        | #创建 bytes 对象,拷贝 object 字节数据     |
| <b>bytes([source[, encoding[, errors]])</b> | #创建 bytes 对象                    |

如果 iterable 中包含非  $0 \leq x < 256$  的整数,则导致 ValueError。

【例 5-24】创建 bytes 对象示例。

|              |                          |                                            |
|--------------|--------------------------|--------------------------------------------|
| >>> bytes()  | >>> bytes((1,2,3))       | >>> bytes((123, 456))                      |
| b"           | b'\x01\x02\x03'          | Traceback (most recent call last):         |
| >>> bytes(2) | >>> bytes('abc','utf-8') | File "<pyshell#95>", line 1, in <module>   |
| b'\x00\x00'  | b'abc'                   | bytes((123, 456))                          |
|              |                          | ValueError: bytes must be in range(0, 256) |

## 5.5.3 创建 bytearray 对象

创建 bytearray 类型的对象实例的基本形式如下:

|                                                 |                                     |
|-------------------------------------------------|-------------------------------------|
| <b>bytearray()</b>                              | #创建空 bytearray 对象                   |
| <b>bytearray(n)</b>                             | #创建长度为 n(整数)的 bytearray 对象,各字节为 0   |
| <b>bytearray(iterable)</b>                      | #创建 bytearray 对象,使用 iterable 中的字节整数 |
| <b>bytearray(object)</b>                        | #创建 bytearray 对象,拷贝 object 字节数据     |
| <b>bytearray([source[, encoding[, errors]])</b> | #创建 bytearray 对象                    |

如果 iterable 中包含非  $0 \leq x < 256$  的整数,则导致 ValueError。

【例 5-25】创建 bytearray 对象示例。

|                        |                              |                                           |
|------------------------|------------------------------|-------------------------------------------|
| >>> bytearray()        | >>> bytearray((1,2,3))       | >>> bytearray((123,456))                  |
| bytearray(b"           | bytearray(b'\x01\x02\x03')   | Traceback (most recent call last):        |
| >>> bytearray(2)       | >>> bytearray('abc','utf-8') | File "<pyshell#102>", line 1, in <module> |
| bytearray(b'\x00\x00') | bytearray(b'abc')            | bytearray((123,456))                      |
|                        |                              | ValueError: byte must be in range(0, 256) |





- A. 4                      B. 5                      C. 6                      D. 7
5. Python 语句 `nums = set([1,2,2,3,3,3,4]); print(len(nums))` 的输出结果是\_\_\_\_\_。
- A. 1                      B. 2                      C. 4                      D. 7
6. Python 语句 `s = 'hello'; print(s[1:3])` 的运行结果是\_\_\_\_\_。
- A. hel                      B. he                      C. ell                      D. el
7. 关于 Python 字符串, 下列说法错误的是\_\_\_\_\_。
- A. 字符即长度为 1 的字符串
- B. 字符串以 \0 标志字符串的结束
- C. 既可以用单引号, 也可以用双引号创建字符串
- D. 在三引号字符串中可以包含换行回车等特殊字符
8. Python 语句 `s1 = [4,5,6]; s2 = s1; s1[1] = 0; print(s2)` 的运行结果是\_\_\_\_\_。
- A. [4,5,6]                      B. [0,5,6]                      C. [4,0,6]                      D. 以上都不对
9. Python 语句 `d = {1:'a', 2:'b', 3:'c'}; print(len(d))` 的运行结果是\_\_\_\_\_。
- A. 0                      B. 1                      C. 3                      D. 6
10. Python 语句 `a = [1,2,3,None,(),[],]; print(len(a))` 的运行结果是\_\_\_\_\_。
- A. 语法错                      B. 4                      C. 5                      D. 6
11. Python 语句 `print('\x48\x41!')` 的运行结果是\_\_\_\_\_。
- A. '\x48\x41!'                      B. 4841!                      C. 4841                      D. HA!
12. Python 语句 `s = {'a',1,'b',2}; print(s['b'])` 的运行结果是\_\_\_\_\_。
- A. 语法错                      B. 'b'                      C. 1                      D. 2
13. Python 语句 `print(chr(65))` 的运行结果是\_\_\_\_\_。
- A. 65                      B. 6                      C. 5                      D. A
14. Python 语句 `print(r"\nGood")` 的运行结果是\_\_\_\_\_。
- A. 新行和字符串 Good                      B. r"\nGood"
- C. \nGood                      D. 字符 r、新行和字符串 Good

## 二、填空题

1. Python 序列类型包括: \_\_\_\_\_。
2. Python 语句 `s1 = 'red hat'; print(str.upper(s1))` 的结果是\_\_\_\_\_, `str.swapcase(s1)` 的结果是\_\_\_\_\_, `s1.title()` 的结果是\_\_\_\_\_, `s1.replace('hat','cat')` 的结果是\_\_\_\_\_。
3. Python 语句 `fruits = ['apple','banana','pear']; print(fruits[-1][-1])` 的结果是\_\_\_\_\_。
4. Python 语句 `fruits = ['apple','banana','pear']; print(fruits.index('apple'))` 的结果是\_\_\_\_\_。
5. Python 语句 `fruits = ['apple','banana','pear']; print('Apple' in fruits)` 的结果是\_\_\_\_\_。
6. Python 语句 `print(sum(range(10)))` 的结果是\_\_\_\_\_。

7. Python 语句 `print('%d%%d'%(3/2, 3%2))` 的结果是\_\_\_\_\_。
8. Python 语句 `print(chr(ord('B')))` 的运行结果是\_\_\_\_\_。
9. Python 语句 `print("hello" 'world')` 的运行结果是\_\_\_\_\_。
10. Python 语句 `print("hello" 'world')` 的运行结果是\_\_\_\_\_。
11. Python 语句 `s = [1, 2, 3, 4]; s.append([5, 6]); print(len(s))` 的运行结果是\_\_\_\_\_。
12. Python 语句 `s1 = [1, 2, 3, 4]; s2 = [5, 6, 7]; print(len(s1 + s2))` 的运行结果是\_\_\_\_\_。
13. Python 语句 `print(tuple(range(2)), list(range(2)))` 的运行结果是\_\_\_\_\_。
14. Python 语句 `print(tuple([1, 2, 3]), list([1, 2, 3]))` 的运行结果是\_\_\_\_\_。
15. Python 列表解析表达式 `[i for i in range(5) if i%2!=0]` 和 `[i**2 for i in range(3)]` 的值分别为\_\_\_\_\_。
16. Python 语句 `first, *middles, last = range(6)` 执行后, `middles` 的值为\_\_\_\_\_; `first, second, third, *lasts = range(6)` 执行后, `lasts` 的值为\_\_\_\_\_; `*firsts, last3, last2, last1 = range(6)` 执行后, `firsts` 的值为\_\_\_\_\_; `first, *middles, last = sorted([86, 85, 99, 88, 60, 95, 96])` 执行后, `sum(middles)/len(middles)` 的值为\_\_\_\_\_。
17. 在 Python 中, 设有 `s = ('a', 'b', 'c', 'd', 'e')`, 则 `s[2]` 值为\_\_\_\_\_; `s[2:4]` 值为\_\_\_\_\_; `s[:3]` 值为\_\_\_\_\_; `s[3:]` 值为\_\_\_\_\_; `s[1:2]` 值为\_\_\_\_\_; `s[-2]` 值为\_\_\_\_\_; `s[:: -1]` 值为\_\_\_\_\_; `s[-2: -1]` 值为\_\_\_\_\_; `s[-2:]` 值为\_\_\_\_\_; `s[-99: -5]` 值为\_\_\_\_\_; `s[-99: -3]` 值为\_\_\_\_\_; `s[::]` 值为\_\_\_\_\_; `s[1: -1]` 值为\_\_\_\_\_。
18. 在 Python 中, 设有 `s = [1, 2, 3, 4, 5, 6]`, 则 `max(s)` 值为\_\_\_\_\_; `min(s)` 值为\_\_\_\_\_; 语句序列 `"s[:1] = []; s[:2] = 'a'; s[2:] = 'b'; s[2:3] = ['x', 'y']; del s[:1]"` 执行后, `s` 值为\_\_\_\_\_。
19. 在 Python 中, 设有 `s = ['a', 'b']`, 则语句序列 `"s.append([1, 2]); s.extend('34'); s.extend([5, 6]); s.insert(1, 7); s.insert(10, 8); s.pop(); s.remove('b'); s[3:] = []; s.reverse()"` 执行后, `s` 值为\_\_\_\_\_。
20. 在 Python 中, 设有 `s = 'abc'`, 则 `s.zfill(7)`、`s.center(7, '')`、`s.ljust(7)`、`s.rjust(7, '0')` 的结果分别为\_\_\_\_\_。
21. 在 Python 中, 设有 `s = 'a,b,c'`、`s2 = ('x', 'y', 'z')` 以及 `s3 = ':'`, 则 `s.split(',')`、`s.rsplit(',', 1)`、`s.partition(',')`、`s.rpartition(',')`、`s3.join('abc')`、`s3.join(s2)` 的结果分别为\_\_\_\_\_。

### 三、思考题

1. Python 中如何实现 tuple 和 list 的转换?
2. 阅读下面 Python 语句。请问输出结果是什么?

```
n = int(input("请输入图形的行数:"))
for i in range(n, 0, -1):
```



```

 print(" ".rjust(20 - i), end = " ")
 for j in range(2 * i - 1): print(" * ", end = " ")
 print("\n")
 for i in range(1, n):
 print(" ".rjust(19 - i), end = " ")
 for j in range(2 * i + 1): print(" * ", end = " ")
 print("\n")

```

3. 阅读下面 Python 语句。请问输出结果是什么？

```

n = int(input("请输入上(或下)三角的行数: "))
for i in range(0, n):
 print(" ".rjust(19 - i), end = " ")
 for j in range(2 * i + 1): print(" * ", end = " ")
 print("\n")
for i in range(n - 1, 0, -1):
 print(" ".rjust(20 - i), end = " ")
 for j in range(2 * i - 1): print(" * ", end = " ")
 print("\n")

```

4. 阅读下面 Python 语句。请问输出结果是什么？

```

daysOfWeek = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
print("DAYS: %s, MONTHS %s" % (daysOfWeek, months))

```

5. 阅读下面 Python 语句。请问输出结果是什么？

```

fruits = ['pear', 'apple', 'kiwi', 'avocado', 'orange']
print("\n".join(fruits))

```

6. 阅读下面 Python 语句。请问输出结果是什么？

```

name = "happy birthday"
print("%s" % name[6:11])
name1 = name.replace(name[6], 'B')
print(name1)

```

7. 阅读下面 Python 语句。请问输出结果是什么？

```

names1 = ['Amy', 'Bob', 'Charlie', 'Daling']
names2 = names1; names3 = names1[:]
names2[0] = 'Alice'; names3[1] = 'Ben'
sum = 0
for ls in (names1, names2, names3):
 if ls[0] == 'Alice': sum += 1
 if ls[1] == 'Ben': sum += 2
print(sum)

```

## 5.7 上机实践

1. 统计输入的字符串中英文字母、数字、空格和其他字符出现的次数。本题使用表 5-3

所示的字符串对象的方法、str 类方法确定字符/字符串是否为字母、数字、空格等。运行效果如图 5-2 所示。

表 5-3 字符串对象的方法/str 类方法

| 方 法     | 功 能                 |
|---------|---------------------|
| isalpha | 判断字符/字符串是否全为字母      |
| isdigit | 判断字符/字符串是否全为数字（0~9） |
| isspace | 判断字符/字符串是否只包含空白字    |

```
请输入字符串：This is a test. 123 45678~ End?
所有字母的总数为： 33
英文字母出现的次数： 13
数字出现的次数： 8
空格出现的次数： 9
其他字符出现的次数： 3
```

图 5-2 统计字符运行效果

2. 统计输入的字符串中单词的个数，单词之间用空格分隔。运行效果如图 5-3 所示。

```
请输入字符串：The quick brown fox jumps over the lazy dog.
其中的单词总数有： 9
```

图 5-3 统计单词运行效果

- 3. 编写程序，实现删除一个 list 里面的重复元素。
- 4. 编写程序，求列表 s = [9,7,8,3,2,1,55,6] 中的元素个数、最大值、最小值、元素之和、平均值。请思考，有哪几种实现方法？提示：可以分别利用 for 循环、while 循环、直接访问列表元素（for i in s...）、间接访问列表元素（for i in range(0,len(s))...）、正序访问（i = 0;while i < len(s)...）、反序访问（i = len(s) - 1;while i >= 0...）以及 while True; ... break 等各种方法。
- 5. 编写程序，将列表 s = [9,7,8,3,2,1,5,6] 中的偶数变成它的平方，奇数保持不变。
- 6. 编写程序，输入字符串，为其每个字符的 ASCII 码形成列表并输出，运行效果如图 5-4 所示。

```
请输入一个字符串：ABCDE123
[65, 66, 67, 68, 69, 49, 50, 51]
```

图 5-4 ASCII 码列表运行效果

## 第 6 章 字典和集合类型

Python 包括内置的高级组合数据类型：字典类型（dict）和集合类型（set 和 frozenset），用于实现复杂的数据结构。

本章要点：

---

- ◆ 字典；
  - ◆ 集合。
- 

### 6.1 字典

字典是一组键 - 值对的数据结构。每个键对应于一个值。在字典中，键不能重复。根据键可以查询到值。

#### 6.1.1 对象的 hash 值

字典是键和值的映射关系。字典的键必须是可 hash 的对象，即实现了 `__hash__()` 的对象。一个对象的 hash 值也可以使用内置函数 `hash()` 获得。例如：

```
>>> hash(100) #结果:100
>>> hash(1.23) #结果:579820504
>>> hash('abc') #结果:1223355671
```

不可变对象（bool、int、float、complex、str、tuple、frozenset 等），是可 hash 对象；而可变对象通常是不可 hash 对象，因为不可变对象的内容可以改变，因而无法通过 `hash()` 函数获取其 hash 值。

字典的键只能使用不可变的对象，但字典的值可以使用不可变或可变的对象。一般而言，应该使用简单的对象作为键。

#### 6.1.2 字典的定义

字典通过花括号中用逗号分隔的项目（键 - 值。键 - 值对使用冒号分隔）定义。其基本形式如下：

```
{键 1:值 1, [键 2:值 2, ..., 键 n:值 n]}
```

键必须为可 hash 对象，因此不可变对象（bool、int、float、complex、str、tuple、frozenset 等）可以作为键；值则可以为任意对象。字典中的键是唯一的，不能重复。

字典也可以通过创建 dict 对象来创建。其基本形式如下：

```
dict() #创建一个空字典
```



```
dict(* * kwargs) #使用关键字参数,创建一个新的字典。此方法最紧凑
dict(mapping) #从一个字典对象创建一个新的字典
dict(iterable) #使用序列,创建一个新的字典
```

【例 6-1】 创建字典对象示例。

```
>>> {}
{}
>>> {'a': 'apple', 'b': 'boy'}
{'a': 'apple', 'b': 'boy'}
>>> dict()
{}
>>> dict({1: 'food', 2: 'drink'})
{1: 'food', 2: 'drink'}
>>> dict([('id', '1001'), ('name', 'Jenny')])
{'name': 'Jenny', 'id': '1001'}
>>> dict(baidu = 'baidu.com', google = 'google.com')
{'baidu': 'baidu.com', 'google': 'google.com'}
```

### 6.1.3 字典的访问操作

字典 d 可以通过键 key 来访问,其基本形式如下:

```
d[key] #返回键为 key 的 value;如果 key 不存在,则导致 KeyError
d[key] = value #设置 d[key] 的值为 value;如果 key 不存在,则添加键/值对
del d[key] #删除字典元素;如果 key 不存在,则导致 KeyError
```

【例 6-2】 字典的访问示例。

```
>>> d = {1: 'food', 2: 'drink'}
>>> d
{1: 'food', 2: 'drink'}
>>> d[1]
'food'
>>> d[3] = 'fruit'
>>> d
{1: 'food', 2: 'drink', 3: 'fruit'}
>>> del d[2]
>>> d
{1: 'food', 3: 'fruit'}
>>> d[2]
Traceback (most recent call last):
 File "<pyshell#100>", line 1, in <module>
 d[2]
KeyError: 2
```

### 6.1.4 字典的视图对象

字典 d 支持下列视图对象,通过它们,可以动态访问字典的数据。

d.keys(): 返回字典 d 的键 key 的列表。

d.values(): 返回字典 d 的值 value 的列表。

d.items(): 返回字典 d 的 (key, value) 对的列表。

【例 6-3】 字典的视图对象示例。

```
>>> d = {1: 'food', 2: 'drink', 3: 'fruit'}
>>> d.keys()
dict_keys([1, 2, 3])
>>> for k in d.keys():
 print(k, end = " ")
1 2 3
>>> d.values()
dict_values(['food', 'drink', 'fruit'])
>>> for v in d.values():
 print(v, end = " ")
food drink fruit
>>> d.items()
dict_items([(1, 'food'), (2, 'drink'), (3, 'fruit')])
>>> for item in d.items():
 print(item, end = " ")
(1, 'food') (2, 'drink') (3, 'fruit')
```

### 6.1.5 判断字典键是否存在

可以通过下列方式之一判断键 key 是否存在于字典 d 中:

key in d            #如果为 True,则表示存在  
key not in d        #如果为 True,则表示不存在

【例 6-4】 判断字典键是否存在示例。

```
>>> d = dict(a = 'apple ', b = 'boy ', c = 'cat ', d = 'dog ')
>>> d
{'d': 'dog ', 'a': 'apple ', 'c': 'cat ', 'b': 'boy '}
>>> 'a' in d
True
>>> 'e' not in d
True
```

6.1.6 字典对象的长度和比较

通过内置函数 len(), 可以获取字典的长度 (元素个数)。虽然字典对象也支持内置函数 max()、min()、sum(), 以计算字典 Key, 但没有太大意义。另外, 字典对象也支持比较运算符 (<、<=、==、!=、>=、>), 但只有 ==、!= 有意义。

【例 6-5】 字典对象的长度和比较示例。

```
>>> d1 = {1:'food ', 2:'drink '}
>>> d2 = {1:'food ', 2:'drink ', 3:'fruit '}
>>> d3 = {1:'food ', 2:'drink ', 3:'fruit '}
>>> len(d1)
2
>>> d1 == d2
False
>>> d2 != d3
False
```

6.1.7 字典对象的方法

字典是可变对象, 其包含的主要方法如表 6-1 所示。假设表中的示例基于 d = {1: 'food ', 2: 'drink ', 3: 'fruit '}。

表 6-1 字典对象的主要方法

| 方 法                 | 说 明                                      | 示 例                                                                                                             |
|---------------------|------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| d. clear()          | 删除所有元素                                   | >>> d. clear();d    #结果:{}                                                                                      |
| d. copy()           | 浅拷贝字典                                    | >>> d1 = d. copy(); id(d), id(d1)<br>(44845896, 45751352)                                                       |
| d. get(k)           | 返回键 k 对应的值, 如果 key 不存在, 返回 None          | >>> d. get(1), d. get(5)<br>( 'food ', None)                                                                    |
| d. get(k, v)        | 返回键 k 对应的值, 如果 key 不存在, 返回 v             | >>> d. get(1, '无'), d. get(5, '无')<br>( 'food ', '无')                                                           |
| d. pop(k)           | 如果键 k 存在, 返回其值, 并删除该项目; 否则导致 KeyError    | >>> d. pop(1), d<br>( 'food ', {2: 'drink ', 3: 'fruit '})                                                      |
| d. pop(k, v)        | 如果键 k 存在, 返回其值, 并删除该项目; 否则返回 v           | >>> d. pop(5, '无'), d<br>( '无', {1: 'food ', 2: 'drink ', 3: 'fruit '})                                         |
| d. setdefault(k, v) | 如果键 k 存在, 返回其值; 否则添加项目 k = v, v 默认为 None | >>> d. setdefault(1)   #结果:'food '<br>>>> d. setdefault(4);d<br>{1: 'food ', 2: 'drink ', 3: 'fruit ', 4: None} |
| d. update([ other]) | 使用字典或键值对, 更新或添加项目到字典 d                   | >>> d1 = {1: '食物', 4: '书籍'}<br>>>> d. update( d1 );d<br>{1: '食物', 2: 'drink ', 3: 'fruit ', 4: '书籍'}            |

## 6.2 集合

集合数据类型是没有顺序的简单对象的聚集，且集合中元素不重复。Python 集合数据类型包括可变集合对象（set）和不可变集合对象（frozenset）。

### 6.2.1 集合的定义

可变集合（set）通过花括号中用逗号分隔的项目定义。其基本形式如下：

```
{x1, [x2, ..., xn]}
```

其中， $x_1, x_2, \dots, x_n$  为任意可 hash 对象。集合中的元素不可重复，且无序，其存储依据对象的 hash 码。hash 码是根据对象的值计算出来的一个唯一值。一个对象如果定义了特殊方法 `__hash__()`，则该对象为可 hash 对象。所有内置不可变对象（bool、int、float、complex、str、tuple、frozenset 等），都是可 hash 对象；所以内置可变对象（list、dict、set），都是非 hash 对象（因为可变对象的值可以变化，故无法计算一个唯一的 hash 值）。集合中可以包含内置不可变对象，不能包含内置可变对象。

**注意：**{} 表示空的 dict，因为 dict 也使用花括号定义。空集为 `set()`。

可变集合也可以通过创建 set 对象来创建，不可变集合通过创建 frozenset 对象来创建。其基本形式如下：

|                                  |                                                    |
|----------------------------------|----------------------------------------------------|
| <code>set()</code>               | #创建一个空的可变集合                                        |
| <code>set(iterable)</code>       | #创建一个可变集合, 包含的项目为可枚举对象 <code>iterable</code> 中的元素  |
| <code>frozenset()</code>         | #创建一个空的不可变集合                                       |
| <code>frozenset(iterable)</code> | #创建一个不可变集合, 包含的项目为可枚举对象 <code>iterable</code> 中的元素 |

【例 6-6】创建集合对象示例。

|                                        |                                        |                                           |
|----------------------------------------|----------------------------------------|-------------------------------------------|
| <code>&gt;&gt;&gt; {1,2,1}</code>      | <code>&gt;&gt;&gt; set()</code>        | <code>&gt;&gt;&gt; {'a',[1,2]}</code>     |
| <code>{1, 2}</code>                    | <code>set()</code>                     | Traceback (most recent call last):        |
| <code>&gt;&gt;&gt; {1,'a',True}</code> | <code>&gt;&gt;&gt; frozenset()</code>  | File "<pyshell#125>", line 1, in <module> |
| <code>{True, 'a'}</code>               | <code>frozenset()</code>               | <code>{'a',[1,2]}</code>                  |
| <code>&gt;&gt;&gt; {1.2, True}</code>  | <code>&gt;&gt;&gt; set('Hello')</code> | TypeError: unhashable type: 'list'        |
| <code>{True, 1.2}</code>               | <code>{'o','e','l','H'}</code>         |                                           |

### 6.2.2 判断集合元素是否存在

可以通过下列方式之一判断一个元素 `x` 是否在集合 `s` 中存在：

|                         |                   |
|-------------------------|-------------------|
| <code>x in s</code>     | #如果为 True, 则表示存在  |
| <code>x not in s</code> | #如果为 True, 则表示不存在 |

【例 6-7】集合中元素的判断示例。

|                                            |                                        |
|--------------------------------------------|----------------------------------------|
| <code>&gt;&gt;&gt; s = set('Hello')</code> | <code>&gt;&gt;&gt; 'h' in s</code>     |
| <code>&gt;&gt;&gt; s</code>                | False                                  |
| <code>{'o','e','l','H'}</code>             | <code>&gt;&gt;&gt; 'o' not in s</code> |
|                                            | False                                  |



6.2.3 集合的运算：并集、交集、差集和对称差集

集合支持表 6-2 所示的集合运算。

表 6-2 集合运算

| 运 算 符                    | 说 明                                                                  |
|--------------------------|----------------------------------------------------------------------|
| $s1 \mid s2 \mid \cdots$ | 返回 $s1$ 、 $s2$ 、 $\cdots$ 的并集： $s1 \cup s2 \cup \cdots$              |
| $s1 \& s2 \& \cdots$     | 返回 $s1$ 、 $s2$ 、 $\cdots$ 的交集： $s1 \cap s2 \cap \cdots$              |
| $s1 - s2 - \cdots$       | 返回 $s1$ 、 $s2$ 、 $\cdots$ 的差集，也记作 $s1 \setminus s2 \setminus \cdots$ |
| $s1 \wedge s2$           | 返回 $s1$ 、 $s2$ 的对称差集： $s1 \triangle s2$                              |

集合的对象方法如表 6-3 所示。

表 6-3 集合的对象方法

| 方 法                                      | 说 明                                                                   |
|------------------------------------------|-----------------------------------------------------------------------|
| <code>s1.isdisjoint(s2)</code>           | 如果集合 $s1$ 和 $s2$ 没有共同元素，返回 <code>True</code> ；否则返回 <code>False</code> |
| <code>s1.issubset(s2)</code>             | 如果集合 $s1$ 是 $s2$ 的子集，返回 <code>True</code> ；否则返回 <code>False</code>    |
| <code>s1.issuperset(s2)</code>           | 如果集合 $s1$ 是 $s2$ 的超集，返回 <code>True</code> ；否则返回 <code>False</code>    |
| <code>s1.union(s2, ...)</code>           | 返回 $s1$ 、 $s2$ 、 $\cdots$ 的并集： $s1 \cup s2 \cup \cdots$               |
| <code>s1.intersection(s2, ...)</code>    | 返回 $s1$ 、 $s2$ 、 $\cdots$ 的交集： $s1 \cap s2 \cap \cdots$               |
| <code>s1.difference(s2, ...)</code>      | 返回 $s1$ 、 $s2$ 、 $\cdots$ 的差集： $s1 - s2 - \cdots$                     |
| <code>s1.symmetric_difference(s2)</code> | 返回 $s1$ 和 $s2$ 的对称差集： $s1 \triangle s2$                               |

【例 6-8】集合的运算示例。

```
>>> s1 = {1,2,3}
>>> s2 = {2,3,4}
>>> s1 | s2
{1, 2, 3, 4}
>>> s1 & s2
{2, 3}
>>> s1 - s2
{1}
>>> s1 ^ s2
{1, 4}
>>> s1.union(s2)
{1, 2, 3, 4}
>>> s1.intersection(s2)
{2, 3}
>>> s1.difference(s2)
{1}
>>> s1.symmetric_difference(s2)
{1, 4}
```

6.2.4 集合的比较运算：相等、子集和超集

集合支持表 6-4 所示的比较运算。

表 6-4 集合的比较运算

| 运 算 符      | 说 明                  | 运 算 符        | 说 明              |
|------------|----------------------|--------------|------------------|
| $s1 == s2$ | $s1$ 和 $s2$ 的元素相同    | $s1 \leq s2$ | $s1$ 是 $s2$ 的子集  |
| $s1 != s2$ | $s1$ 和 $s2$ 的元素不完全相同 | $s1 \geq s2$ | $s1$ 是 $s2$ 的超集  |
| $s1 < s2$  | $s1$ 是 $s2$ 的纯子集     | $s1 > s2$    | $s1$ 是 $s2$ 的纯超集 |

集合对象的比较方法包括 `s1.isdisjoint(s2)`、`s1.issubset(s2)` 和 `s1.issuperset(s2)`，参见表 6-3。

【例 6-9】 集合比较运算示例。

|                  |              |              |                       |
|------------------|--------------|--------------|-----------------------|
| >>> s1 = {1,2,3} | >>> s1 != s4 | >>> s3 < s4  | >>> s1.isdisjoint(s2) |
| >>> s2 = {3,2,1} | True         | False        | False                 |
| >>> s3 = {1,2}   | >>> s3 <= s1 | >>> s3 > s4  | >>> s3.issubset(s1)   |
| >>> s4 = {7,9}   | True         | False        | True                  |
| >>> s1 == s2     | >>> s2 > s3  | >>> s1 >= s2 | >>> s2.issuperset(s3) |
| True             | True         | True         | True                  |

6.2.5 集合的长度、最大值、最小值、元素和

通过内置函数 len()、max()、min()、sum()，可以获取集合的长度、元素最大值、元素最小值、元素之和。如果有非数值元素，则求和将导致 TypeError。

【例 6-10】 集合的长度、最大值、最小值、元素和示例。

|                      |             |                                                               |
|----------------------|-------------|---------------------------------------------------------------|
| >>> s1 = {1,3,5,7,9} | >>> max(s1) | >>> sum(s2)                                                   |
| >>> s2 = {'1',       | 9           | Traceback (most recent call last):                            |
| '2','3'}             | >>> min(s2) | File "<pyshell#172>", line 1, in <module>                     |
| >>> len(s1)          | '1'         | sum(s2)                                                       |
| 5                    |             | TypeError: unsupported operand type(s) for +: 'int' and 'str' |

6.2.6 可变集合的方法

set 集合是可变对象，包含的主要方法如表 6-5 所示。假设表中的示例基于 “s1 = {1,2,3}; s2 = {2,3,4}”。

表 6-5 可变集合对象的主要方法

| 方 法                                               | 说 明                                 | 示 例                                                 |
|---------------------------------------------------|-------------------------------------|-----------------------------------------------------|
| s1.update(s2,...)<br>s1  = s2   ...               | 并集<br>s1 = s1 ∪ s2 ∪ ...            | >>> s1.update(s2)<br>#s1 = {1, 2, 3, 4}             |
| s1.intersection_update(s2, ...)<br>s1 &= s2 & ... | 交集<br>s1 = s1 ∩ s2 ∩ ...            | >>> s1.intersection_update(s2)<br># s1 = {2, 3}     |
| s1.difference_update(s2, ...)<br>s1 -= s2 - ...   | 差集<br>s1 = s1 - s2 - ...            | >>> s1.difference_update(s2)<br># s1 = {1}          |
| s1.symmetric_difference_update(s2)<br>s1 ^= s2    | 对称差集<br>s1 = s1 Δ s2                | s1.symmetric_difference_update(s2)<br># s1 = {1, 4} |
| s.add(x)                                          | 把对象 x 添加到集合 s                       | >>> s1.add('a') # s1 = {1, 2, 3, 'a'}               |
| s.remove(x)                                       | 从集合 s 中移除对象 x。若不存在，则导致 KeyError     | >>> s1.remove(1) #s1 = {2, 3}                       |
| s.discard(x)                                      | 从集合 s 中移除对象 x（如果存在的话）               | >>> s1.discard(3) # s1 = {1, 2}                     |
| s.pop()                                           | 从集合 s 随机弹出一个元素，如果 s 为空，则导致 KeyError | >>> s1.pop() #输出:1。s1 = {2, 3}                      |
| s.clear()                                         | 清空集合 s                              | >>> s1.clear() # s1 = set()                         |





```
list1 = { }; list1[1] = 1; list1['1'] = 3; list1[1] += 2; sum = 0
for k in list1: sum += list1[k]
print(sum)
```

2. 阅读下面 Python 语句。请问输出结果是什么？

```
d = {1:'a', 2:'b', 3:'c'};
del d[1]; d[1] = 'x'; del d[2]; print(d)
```

3. 阅读下面 Python 语句。请问输出结果是什么？

```
item_counter = {}
def addone(item):
 if item in item_counter: item_counter[item] += 1
 else: item_counter[item] = 1
addone('Apple'); addone('Pear'); addone('apple')
addone('Apple'); addone('kiwi'); addone('apple')
print(item_counter)
```

4. 阅读下面 Python 语句。请问输出结果是什么？

```
numbers = {}; numbers[(1,2,3)] = 1;
numbers[(2,1)] = 2; numbers[(1,2)] = 3; sum = 0
for k in numbers: sum += numbers[k]
print(len(numbers),',',sum,',',numbers)
```

5. 阅读下面 Python 语句。请问输出结果是什么？

```
d1 = {'a':1, 'b':2}; d2 = d1; d1['a'] = 6
sum = d1['a'] + d2['a']
print(sum)
```

6. 阅读下面 Python 语句。请问输出结果是什么？

```
d1 = {'a':1, 'b':2}; d2 = dict(d1); d1['a'] = 6
sum = d1['a'] + d2['a']
print(sum)
```

## 6.4 上机实践

1. 创建由'Monday'~'Sunday'七个值组成的字典，输出键列表、值列表及键值列表。运行效果参见图 6-1 所示。

|            |             |            |             |            |            |            |
|------------|-------------|------------|-------------|------------|------------|------------|
| 1          | 2           | 3          | 4           | 5          | 6          | 7          |
| Mon        | Tues        | Wed        | Thur        | Fri        | Sat        | Sun        |
| (1, 'Mon') | (2, 'Tues') | (3, 'Wed') | (4, 'Thur') | (5, 'Fri') | (6, 'Sat') | (7, 'Sun') |

图 6-1 字典运行效果

2. 随机生成 10 个 0（含）~10（含）的整数，分别组成集合 A 和集合 B，输出 A 和 B 的内容、长度、最大值、最小值以及它们的并集、交集和差集。运行效果参见图 6-2 所示。

集合的内容、长度、最大值、最小值分别为：  
{0, 8, 10, 5, 7} 5 10 0  
{9, 2, 10, 5, 6} 5 10 2  
A和B的并集、交集和差集分别为：  
{0, 2, 5, 6, 7, 8, 9, 10} {10, 5} {0, 8, 7}

图 6-2 集合运行效果

## 第 7 章 文件和流 I/O

应用程序往往需要从磁盘文件中读取数据，或者把数据存储到磁盘文件中，以持久地保存应用程序的数据。

### 本章要点：

---

- ◆ 文件和文件对象；
  - ◆ 文本文件的读取和写入；
  - ◆ 二进制文件的读取和写入；
  - ◆ 随机文件访问；
  - ◆ os 模块和文件访问；
  - ◆ 对象序列化。
- 

## 7.1 文件和文件对象

### 7.1.1 文件和流概述

文件可以看作是数据的集合，一般保存在磁盘或其他存储介质上。文件 I/O（数据的输入/输出）通过流（Stream）来实现；流提供一种向后备存储写入字节和从后备存储读取字节的方式。后备存储包括各种存储媒介，如磁盘、磁带、内存和网络等，对应于文件流、磁带流、内存流和网络流等。流有 5 种基本的操作：打开、读取、写入、改变当前位置和关闭。

### 7.1.2 文件对象和 open 函数

内置函数 `open()` 用于打开或创建文件对象，其语法格式如下：

```
f = open(file, mode = 'r', buffering = -1, encoding = None)
```

其中，`file` 是要打开或创建的文件名，如果不在当前路径，需指出具体路径；`mode` 是打开文件的模式；`buffering` 表示是否使用缓存（默认为 `-1`，表示使用系统默认的缓冲区大小）；`encoding` 是文件的编码。`open()` 函数返回一个文件对象 `f`。

使用 `open()` 函数时，可以指定打开文件的模式 `mode` 为：`'r'`（只读）、`'w'`（写入，写入前删除旧内容）、`'x'`（创建新文件，如果文件存在，则导致 `FileExistsError`）、`'a'`（追加）、`'b'`（二进制文件）、`'t'`（文本文件，默认值）、`'+'`（更新，读写）。

`open()` 函数默认打开模式为 `'r'`（即 `'rt'`），即文本读取模式。

文件操作容易产生异常，而且最后需要关闭打开的文件。故一般使用 `try...except...final-`



ly 语句，在 try 语句块中执行文件相关操作，使用 except 捕获可能发生的异常，在 finally 语句块中确保关闭打开的文件。

```
try:
 f = open(file, mode) #打开文件
 #操作打开的文件
except:
 #捕获异常
 #发生异常时执行的操作
finally:
 f.close() #关闭打开的文件
```

### 7.1.3 with 语句和上下文管理协议

使用 try...except...finally，可以确保在 try 语句块中获得的资源（如打开的文件），在 finally 语句块中释放。

为了简化操作，Python 语言与资源相关的对象可以实现上下文管理协议。实现上下文管理协议的对象包含两个特殊方法。

contextmanager.\_\_enter\_\_(): 进入上下文管理，一般返回资源对象。

contextmanager.\_\_exit\_\_(exc\_type, exc\_val, exc\_tb): 离开上下文管理器，一般关闭打开的资源。

其中 exc\_type、exc\_val 和 exc\_tb 分别为异常类型、值和跟踪信息。

实现上下文管理协议的对象可以使用 with 语句：

```
with context [as var]
 操作语句
```

with 语句定义了一个上下文。执行 with 语句时，首先调用上下文对象 context 的 \_\_enter\_\_(), 其返回值赋值给 var；离开 with 语句块时，最后调用 context 的 \_\_exit\_\_(), 确保释放资源。

文件对象支持使用 with 语句，确保打开的文件自动关闭：

```
with open(file, mode) as f:
 #操作打开的文件
```

## 7.2 文本文件的读取和写入

使用 open() 函数打开或创建一个文件时，其默认的打开模式为只读文本文件。文本文件用于储存文本字符串，默认编码为 Unicode。

### 7.2.1 文本文件的写入

文本文件的写入一般包括三个步骤：打开文件、写入数据和关闭文件。

#### 1. 创建或打开文件对象

通过内置函数 open() 可创建或打开文件对象，可指定覆盖模式（文件存在时）、编码和缓存大小。例如：

```
fl = open('data1.txt', 'w') #创建或打开 data1.txt
```

```
f2 = open('data2.txt', 'x') #创建文件 data1.txt,若 data2.txt 已存在,则导致 FileExistsError
f3 = open('data1.txt', 'a') #创建或打开 data1.txt,附加模式
```

## 2. 写入字符串到文本文件

打开文件后,可以使用其实例方法 `write()`/`writelines()`, 写入字符串到文本文件。可使用实例方法 `flush` 强制把缓冲的数据更新到文件中。

`f.write(s)`: 把字符串 `s` 写入到文件 `f`。

`f.writelines(lines)`: 依次把列表 `lines` 中的各字符串写入到文件 `f`。

`f.flush()`: 把缓冲的数据更新到文件中。

实例方法 `write()`/`writelines()` 不会添加换行符, 可以通过添加 `\n` 实现换行。例如:

```
f.write('123\n') #写入字符串
f.write('abc\n') #写入字符串
f.writelines(['456\n', 'def\n']) #写入字符串
```

## 3. 关闭文件

写入文件完成后, 应该使用 `close()` 方法关闭流, 以释放资源, 并把缓冲的数据更新到文件中。

**f.close()** #关闭文件

可以使用异常处理的 `finally` 子句, 以保证即使发生异常时, 也会关闭打开的文件。

```
f = open('data1.txt', 'w') #打开文件
try:
 #文件处理操作
finally:
 f.close() #关闭文件
```

通常, 文件操作一般采用 `with` 语句, 以保证系统自动关闭打开的流。

```
with open('data1.txt', 'w') as f:
 #文件处理操作
```

**【例 7-1】** 文本文件的写入示例 (`textwrite.py`)。

```
with open(r'c:\python\data1.txt', 'w') as f:
 f.write('123\n') #写入字符串
 f.write('abc\n') #写入字符串
 f.writelines(['456\n', 'def\n']) #写入字符串
```

## 7.2.2 文本文件的读取

文本文件的读取一般包括三个步骤: 打开文件、读取数据和关闭文件。

### 1. 打开文件对象

通过内置函数 `open()` 可打开文件对象。可指定编码和缓存大小。例如:

```
fl = open('data1.txt', 'r') #打开 data1.txt,若文件不存在,则导致 FileNotFoundError
```

### 2. 从打开的文本文件中读取字符数据

打开文件后, 可以使用下列实例方法读取字符数据。

`f.read()`: 从 `f` 中读取剩余内容直至文件结尾, 返回一个字符串。

`f.read(n)`: 从 `f` 中读取至多 `n` 个字符, 返回一个字符串; 如果 `n` 为负数或 `None`, 读取直至文件结尾。

`f.readline()`: 从 `f` 中读取 1 行内容, 返回一个字符串。

`f.readlines()`: 从 `f` 中读取剩余多行内容, 返回一个列表。

例如:

```
>>> f1 = open(r'c:\python\data1.txt', 'r') #打开文件
>>> f1.readline() #读入 1 行内容:'123\n'
>>> f1.readlines() #读入剩下多行内容:['abc\n', '456\n', 'def\n']
```

另外, 文件可以直接迭代。文本文件按行迭代。例如:

```
>>> f1 = open(r'c:\python\data1.txt', 'r')
>>> for s in f1:
 print(s, end = ")
123
abc
456
def
```

### 3. 关闭文件

可以使用 `close()` 方法关闭流, 以释放资源。通常采用 `with` 语句, 以保证系统自动关闭打开的流。

**【例 7-2】** 文本文件的读取示例 (`textread.py`)。

```
with open(r'c:\python\data1.txt', 'r') as f:
 for s in f.readlines():
 print(s, end = ")
```

运行结果如下:

```
123
abc
456
def
```

## 7.2.3 文本文件的编码

文本文件用于存储编码的字符串, 使用 `open()` 函数打开文本文件时, 可以指定所使用的编码。

```
open(file, mode = 'r', buffering = -1, encoding = None, errors = None, newline = None, closefd =
True, opener = None)
```

`encoding` 默认为 `None`, 即不指定。默认的编码与平台有关, 其值为:

```
>>> import locale
>>> locale.getpreferredencoding() #'cp936'
```

Python 内置的编码包括: `utf-8`、`utf8`、`latin-1`、`latin1`、`iso-8859-1`、`mbcs` (仅 Windows 系统)、`ascii`、`utf-16`、`utf-32` 等。例如:

```
>>> f = open("1.txt", mode = "w", encoding = "utf-8")
```

## 7.3 二进制文件的读取和写入

使用 `open()` 函数打开或创建一个文件时，可以指定打开模式为 `'b'`，以打开二进制文件。文本文件用于存储编码的字符串，二进制文件直接存储字节码，广泛用于存储各种程序数据，如图像文件、音频/视频文件等。

### 7.3.1 二进制文件的写入

二进制文件的写入一般包括三个步骤：打开文件、写入数据和关闭文件。

#### 1. 创建或打开文件对象

通过内置函数 `open()`，指定打开模式 `'b'`，可创建或打开二进制文件对象。可指定覆盖模式（文件存在时）和缓存大小。例如：

```
f1 = open('data1.dat', 'wb') #创建或打开 data1.dat
f2 = open('data2.dat', 'xb') #创建文件 data1.dat,若 data2.txt 已存在,则导致 FileExistsError
f3 = open('data1.dat', 'ab') #创建或打开 data1.dat,附加模式
```

#### 2. 写入字节数据到二进制文件

打开文件后，可以使用其实例方法 `write()`，写入字节数据（`bytes` 或 `bytearray`）到二进制文件。可使用实例方法 `flush()` 强制把缓冲的数据更新到文件中。

`f.write(b)`：将字节数据 `b` 写入到二进制文件 `f`，返回实际写入的字节数。

`f.flush()`：将缓冲的数据更新到文件中。

例如：

```
f1.write(b'123') #写入字节数据
f1.write(b'abc') #写入字节数据
```

#### 3. 关闭文件

可以使用 `close()` 方法关闭流，以释放资源。通常采用 `with` 语句，以保证系统自动关闭打开的流。

**【例 7-3】** 二进制文件的写入示例（`binarywrite.py`）。

```
with open(r'c:\python\data1.dat', 'wb') as f:
 f.write(b'123') #写入字节数据
 f.write(b'abc') #写入字节数据
```

### 7.3.2 二进制文件的读取

二进制文本文件的读取一般包括三个步骤：打开文件、读取数据和关闭文件。

#### 1. 打开文件对象

通过内置函数 `open()`，指定打开模式 `'b'`，可打开二进制文件对象。可指定编码和缓存大小。例如：

```
f1 = open('data1.dat', 'rb') #打开 data1.dat,若文件不存在,则导致 FileNotFoundError
```



## 2. 从打开的文本文件中读取字符数据

打开文件后，可以使用下列实例方法读取字符数据。

`f.read()`：从 `f` 中读取剩余内容直至文件结尾，返回一个 `bytes` 对象。

`f.read(n)`：从 `f` 中读取至多 `n` 个字节，返回一个 `bytes` 对象；如果 `n` 为负数或 `None`，读取直至文件结尾。

`f.readinto(b)`：从 `f` 中读取至多 `len(b)` 个字节到 `bytes` 对象 `b`。

例如：

```
fl.read(1) #读取 1 个字节,结果:b 'l'
fl.read() #读取剩余字节,结果:b '23abc'
```

## 3. 关闭文件

可以使用 `close()` 方法关闭流，以释放资源。通常采用 `with` 语句，以保证系统自动关闭打开的流。

【例 7-4】文本文件的读取示例（`binaryread.py`）。

```
with open(r'c:\python\data1.dat', 'rb') as f:
 b = f.read()
 print(b)
```

运行结果如下：

```
b '123abc'
```

# 7.4 随机文件访问

文件的写入和读取一般从当前位置开始（打开文件时位置为 0），直至文件结尾（EOF），即按顺序访问。文件对象支持 `seek()` 方法，`seek()` 通过字节偏移量将读取/写入位置移动到文件中的任意位置，从而实现文件的随机访问。

`seek(offset, whence = os.SEEK_SET)`

其中，`offset` 为移动的字节偏移量，`whence` 为相对参考点（文件开始、当前位置、结尾，分别对应于 `os.SEEK_SET`、`os.SEEK_CUR`、`os.SEEK_END`，或 0、1、2）。

随机文件访问一般针对二进制文件，因为其存储内容为字节码。文本文件也可以使用 `seek()` 方法，但多字节的偏移量不容易控制，有时候会导致无意义。

## 1. 创建或打开随机文件

随机文件一般同时提供读写操作，即使用内置函数 `open()`，指定打开模式 '+'。例如：

```
f1 = open('data1.dat', 'w+b') #创建或打开 data1.dat
f2 = open('data2.dat', 'x+b') #创建文件 data1.dat;若 data2.txt 已存在,则导致 FileExistsError
f3 = open('data1.dat', 'a+b') #创建或打开 data1.dat,附加模式
f4 = open('data1.dat', 'wb+') #创建或打开 data1.dat,同'w+b'
```

## 2. 定位

打开文件后，可以使用其实例方法 `seek()` 进行定位，即将该文件的当前位置设置为给定值。例如：

```
fl.seek(0) #定位到开始位置
```

### 3. 写入/读取数据

打开文件，并定位文件位置后，可以使用其实例方法 `write()`/`read()`，写入或读取字节数据。例如：

```
fl.seek(0, os.SEEK_END) #定位到结束位置
fl.write(b'hello') #写入字节数据
fl.seek(3) #定位到结束位置
fl.read(3) #读取3个字节,结果:b'abc'
```

### 4. 关闭文件

可以使用 `close()` 方法关闭流，以释放资源。通常采用 `with` 语句，以保证系统自动关闭打开的流。

## 7.5 CSV 文件格式的读取和写入

csv 是逗号分隔符文本格式，常用于 Excel 和数据库的数据导入和导出。Python 标准库的模块 `csv` 提供了读取和写入 csv 格式文件的对象。

本节基于以下 `scores.csv` 文件，其内容为：

```
学号,姓名,性别,班级,语文,数学,英语
101511,宋颐园,男,一班,72,85,82
101513,王二丫,女,一班,75,82,51
101531,董再永,男,三班,55,74,79
101521,陈香燕,女,二班,80,86,68
101535,周一萍,女,三班,72,76,72
```

### 7.5.1 csv.reader 对象和 csv 文件的读取

`csv.reader` 对象用于从 csv 文件读取数据（格式为列表对象）：

```
csv.reader(csvfile, dialect='excel', **fmtparams) #构造函数
```

其中，`csvfile` 是文件对象或 `list` 对象；`dialect` 用于指定 csv 的格式模式，不同程序输出的 csv 格式有细微差别；`fmtparams` 用于指定特定格式，以覆盖 `dialect` 中的格式。

`csv.reader` 对象是可迭代对象。`reader` 对象包含以下属性。

`csvreader.dialect`：返回其 `dialect`。

`csvreader.line_num`：返回读入的行数。

**【例 7-5】** 使用 `reader` 对象读取 csv 文件（`csv_reader1.py`）。

```
import csv
def readcsv1(csvfilepath):
 with open(csvfilepath, newline="") as f: #打开文件
 f_csv = csv.reader(f) #创建 csv.reader 对象
 headers = next(f_csv) #标题
 print(headers) #打印标题(列表)
```

```

 for row in f_csv: #循环打印各行(列表)
 print(row)
 if __name__ == '__main__':
 readcsv1(r'scores.csv')

```

运行结果如下：

```

['学号', '姓名', '性别', '班级', '语文', '数学', '英语']
['101511', '宋颐园', '男', '一班', '72', '85', '82']
['101513', '王二丫', '女', '一班', '75', '82', '51']
...(略)

```

## 7.5.2 csv.writer 对象和 csv 文件的写入

csv.writer 对象用于把列表对象数据写入到 csv 文件：

```
csv.writer(csvfile, dialect='excel', **fmtparams) #构造函数
```

其中，csvfile 是任何支持 write() 方法的对象，通常为文件对象；dialect 和 fmtparams 与 csv.reader 对象构造函数中的参数意义相同。

csv.writer 对象支持下列方法和属性。

csvwriter.writerow (row)：方法，写入一行数据。

csvwriter.writerows (rows)：方法，写入多行数据。

csvreader.dialect：只读属性，返回其 dialect。

**【例 7-6】** 使用 writer 对象写入 csv 文件 (csv\_writer1.py)。

```

import csv
def writecsv1(csvfilepath):
 headers = ['学号', '姓名', '性别', '班级', '语文', '数学', '英语']
 rows = [('101511', '宋颐园', '男', '一班', '72', '85', '82'),
 ('101513', '王二丫', '女', '一班', '75', '82', '51')]
 with open(csvfilepath, 'w', newline=") as f: #打开文件
 f_csv = csv.writer(f) #创建 csv.writer 对象
 f_csv.writerow(headers) #写入 1 行(标题)
 f_csv.writerows(rows) #写入多行(数据)
if __name__ == '__main__':
 writecsv1(r'scores1.csv')

```

## 7.5.3 csv.DictReader 对象和 csv 文件的读取

使用 csv.reader 对象从 csv 文件读取数据，结果为列表对象 row，需要通过索引 row[i] 访问。如果希望通过 csv 文件的首行标题字段名访问，则可以使用 csv.DictReader 对象的构造函数，以返回 map：

```
csv.DictReader(csvfile, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwds)
```

其中，csvfile 是文件对象或 list 对象；fieldnames 用于指定字段名，如果没有指定，则第 1 行为字段名；restkey 和 restval 是可选的，用于指定字段名和数据个数不一致时所对应的字



段名或数据值。其他参数同 reader 对象。

除了支持 csv.reader 对象的方法和属性，DictReader 还包含下面的属性：

**csvreader.fieldnames** #返回标题字段名

【例 7-7】使用 DictReader 对象读取 csv 文件（csv\_reader2.py）。

```
import csv
def readcsv2(csvfilepath):
 with open(csvfilepath, newline = "") as f: #打开文件
 f_csv = csv.DictReader(f) #创建 csv.DictReader 对象
 headers = next(f_csv) #标题
 print(headers) #打印标题(列表)
 for row in f_csv: #循环打印各行(列表)
 print(row)
if __name__ == '__main__':
 readcsv2(r'scores.csv')
```

运行结果如下：

```
['学号', '姓名', '性别', '班级', '语文', '数学', '英语']
['101511', '宋颐园', '男', '一班', '72', '85', '82']
['101513', '王二丫', '女', '一班', '75', '82', '51']
...(略)
```

#### 7.5.4 csv.DictWriter 对象和 csv 文件的写入

如果需要写入 map 数据到 csv，则可以使用 csv.DictWriter 对象的构造函数：

**csv.DictWriter(csvfile, fieldnames, restval = "", extrasaction = 'raise', dialect = 'excel', \* args, \* kwds)**

其中，csvfile 是文件对象或 list 对象；fieldnames 用于指定字段名；可选的 restval 用于指定默认数据。extrasaction 是可选的，用于指定多余字段时的操作；其他参数同 writer 对象。

除了支持 csv.writer 对象的方法和属性，DictWriter 还包含下面的方法。

DictWriter.writeheader()：写入标题字段名（构造函数中的参数）。

【例 7-8】使用 DictWriter 对象写入 csv 文件（csv\_writer2.py）。

```
import csv
def writecsv2(csvfilepath):
 headers = ['学号', '姓名', '语文', '数学', '英语']
 rows = [{'学号': '101511', '姓名': '宋颐园', '语文': '72', '数学': '85', '英语': '82'},
 {'学号': '101513', '姓名': '王二丫', '语文': '75', '数学': '82', '英语': '51'}]
 with open(csvfilepath, 'w', newline = "") as f: #打开文件
 f_csv = csv.DictWriter(f, headers) #创建 csv.DictWriter 对象
 f_csv.writeheader() #写入标题
 f_csv.writerows(rows) #写入多行(数据)
if __name__ == '__main__':
 writecsv2(r'scores2.csv')
```



### 7.5.5 csv 文件格式化参数和 Dialect 对象

#### 1. csv 文件格式化参数

创建 reader/writer 对象时，可以指定 csv 文件格式化命名参数。

delimiter: 项目分隔符，默认为','。

doublequote: 如果为 True (默认值)，字符串中的双引号使用"" 表示；如果为 False，使用转义字符 escapechar: 指定的字符。

escapechar: 转义字符，默认为 None。

lineterminator: 用于 writer 的换行符，默认为'\r\n'。

quotechar: 用于限定包含特殊字符的字段的符号，默认为'"'

quoting: 用于指定使用双引号的规则，可以为 csv 模块中的常量: QUOTE\_ALL (全部)、QUOTE\_MINIMAL (仅特殊字符字段)、QUOTE\_NONNUMERIC (非数字字段)、QUOTE\_NONE (全部不)。

skipinitialspace: 如果为 True，省略分隔符前面的空格；默认值为 False。

strict: 如果为 True，读入错误格式 CSV 行时将导致 csv. Error；默认值为 False。

【例 7-9】 csv 文件格式化参数示例 (csv\_writer3.py)。

```
import csv
def writecsv3(csvfilepath):
 headers = ['学号', '姓名', '性别', '班级', '语文', '数学', '英语']
 rows = [('101511', '宋颐园', '男', '一班', '72', '85', '82'),
 ('101513', '王二丫', '女', '一班', '75', '82', '51')]
 with open(csvfilepath, 'w', newline='') as f:
 f_csv = csv.writer(f, delimiter=';', quoting=csv.QUOTE_ALL) #指定格式化参数
 f_csv.writerow(headers) #写入一行(标题)
 f_csv.writerows(rows) #写入多行(数据)
if __name__ == '__main__':
 writecsv3(r'scores3.csv')
```

#### 2. Dialect 对象

若干格式化参数可以组织成 Dialect 对象，Dialect 对象包含对应于命名格式化参数的属性。可以创建 Dialect 或其派生类的对象，然后传递给 reader 或 writer 的构造函数。

可以使用下列 csv 模块的函数，创建 Dialect 对象。

csv.register\_dialect(name[, dialect], \*\*fmtparams): 使用命名参数，注册一个名称。

csv.unregister\_dialect(name): 取消注册的名称。

csv.get\_dialect(name): 获取注册名称的 Dialect 对象，无注册时 csv. Error。

csv.list\_dialects(): 所有注册 Dialect 对象的列表。

例如:

```
>>> csv.list_dialects() #['excel', 'unix', 'excel-tab']
```

另外，可以使用 csv 模块的函数，获取和设置字段长度限制:

```
csv.field_size_limit([new_limit])
```

【例 7-10】 Dialect 对象示例 (csv\_writer4.py)。

```
import csv
def writecsv4(csvfilepath):
 csv.register_dialect('mydialect', delimiter=';', quoting=csv.QUOTE_NONE)
 headers = ['学号', '姓名', '性别', '班级', '语文', '数学', '英语']
 rows = [('101511', '宋颐园', '男', '一班', '72', '85', '82'),
 ('101513', '王二丫', '女', '一班', '75', '82', '51')]
 with open(csvfilepath, 'w', newline='') as f:
 f_csv = csv.writer(f, 'mydialect') #指定格式化参数
 f_csv.writerow(headers) #写入1行(标题)
 f_csv.writerows(rows) #写入多行(数据)
if __name__ == '__main__':
 writecsv4(r'scores4.csv')
```

## 7.6 os 模块和文件访问

### 7.6.1 文件描述符

操作系统中，进程所打开的文件一般通过文件描述符（一个简单的整数）来标识。操作系统通过文件描述符进行文件访问操作。

例如，标准输入、标准输出、标准错误，其对应的文件描述符分别为 0、1、2。

注：在 UNIX/Linux 系统中，套接字（Socket）和管道（Pipe）也使用文件描述符来标识。

### 7.6.2 使用 os 模块提供的函数访问文件

os 模块提供了使用文件描述符来访问文件的相关函数，它们属于底层文件访问，提供更高级的文件操作功能。一般建议直接使用 Python 内置的文件对象进行文件访问。

#### 1. 创建或打开文件

通过 os 模块中的函数 open()，可以创建或打开文件，返回文件描述符。

**os.open(file, flags, mode=0o777)** #打开文件,返回文件描述符

其中，file 为文件路径；flags 为打开标志（如只读：os.O\_RDONLY）；mode 为打开模式。

os 模块中定义了下列标志位常量：os.O\_RDONLY、os.O\_WRONLY、os.O\_RDWR、os.O\_APPEND、os.O\_CREAT、os.O\_EXCL、os.O\_TRUNC；以及特定于操作系统的其他常量，如 Windows 平台上的 os.O\_BINARY（二进制文件）。例如：

fd = os.open('data2.dat', (os.O\_RDWR | os.O\_CREAT | os.O\_BINARY)) #创建二进制文件

#### 2. 定位

打开文件后，可以使用模块函数 os.lseek() 进行定位。

**os.lseek(fd, pos, how)**

其中，fd 为文件描述符；pos 为移动的字节偏移量，how 为相对参考点（文件开始、当前位置、结尾，分别对应于 os.SEEK\_SET、os.SEEK\_CUR、os.SEEK\_END，或 0、1、2）。

例如：

```
os.lseek(fd,0) #/定位到开始位置
os.lseek(fd,0, SEEK_END) #/定位到结束位置
```

### 3. 写入/读取数据

打开文件，并定位文件位置后，可以使用模块函数 `os.write()` 或 `os.read()`，写入或读取字节数据；可使用模块函数 `os.flush()` 强制把缓冲的数据更新到文件中。

`os.write(fd, str)`：将字节字符串 `str` 写入到文件 `fd`，返回实际写入的字节数。

`os.read(fd, n)`：从 `fd` 中读取至多 `n` 个字节，返回一个 `bytestring` 对象。

`os.fsync(fd)`：将缓冲的数据更新到文件中。

例如：

```
os.lseek(fd,0, os.SEEK_END) #定位到结束位置
os.write(fd,b'hello') #写入字节数据
os.lseek(fd,0, os.SEEK_SET) #定位到开始位置
os.read(fd,3) #b'hel' #读取3个字节,结果:b'hel'
```

### 4. 关闭文件

可以使用 `close()` 方法关闭流，以释放资源。

```
os.close(fd) #关闭文件 fd
```

## 7.7 对象序列化

### 7.7.1 对象序列化

程序运行时，其数据对象创建在内存中。如果需要持久保存到磁盘，或通过网络传递给其他机器，则需要通过对象序列化机制。

对象序列化也称为串行化，将对象转换为数据形式，并转储到磁盘文件或通过网络实现跨平台传输。反过来，从磁盘数据文件或接收到的数据形式恢复以得到相应对象的过程，则称为反序列化。

多个对象可以串行化转储到一个磁盘文件，用户不必关心数据的格式。对象序列化机制广泛用于各种分布式并行处理系统。

使用 `pickle/cPickle` 模块中提供的函数，可以实现 Python 对象的序列化。`cPickle` 使用 C 语言实现，故其运行效率更高。另外，`marshal` 模块也提供了类似的函数，但 `pickle` 模块更具有普遍意义。

### 7.7.2 pickle 模块和对象序列化

`pickle` 模块实现了 Python 数据对象的序列化和反序列化。

`pickle.dump(obj, file, protocol=None)`：将对象 `obj` 保存到文件 `file` 中去。

`pickle.load(file)`：从 `file` 中读取并重构 1 个对象。

其中，`protocol` 为序列化使用的协议版本：0（默认值，ASCII 协议，所序列化的对象使用可打印的 ASCII 码表示。）、1（二进制协议）、2（二进制协议，2.3 版本）和 3（二进制



协议, 3.0 版本)。

【例 7-11】对象序列化示例 (pickledump.py)。

```
import pickle
with open(r'c:\python\dataObj1.dat', 'wb') as f:
 s1 = 'Hello! '
 c1 = 1 + 2j
 t1 = (1, 2, 3)
 d1 = dict(name = 'Mary', age = 19)
 pickle.dump(s1, f)
 pickle.dump(c1, f)
 pickle.dump(t1, f)
 pickle.dump(d1, f)
```

【例 7-12】对象反序列化示例 (pickleload.py)。

```
import pickle
with open(r'c:\python\dataObj1.dat', 'rb') as f:
 o1 = pickle.load(f)
 o2 = pickle.load(f)
 o3 = pickle.load(f)
 o4 = pickle.load(f)
 print(type(o1), str(o1))
 print(type(o2), str(o2))
 print(type(o3), str(o3))
 print(type(o4), str(o4))
```

运行结果如下:

```
< class 'str' > Hello!
< class 'complex' > (1 + 2j)
< class 'tuple' > (1, 2, 3)
< class 'dict' > {'name': 'Mary', 'age': 19}
```

## 7.8 输入重定向和管道

fileinput 模块提供处理用于循环处理输入、输入重定向、管道或一个或多个文本文件的函数和辅助对象。

### 7.8.1 FileInput 对象

fileinput 模块的 FileInput 对象用于循环处理多个文件、重定向输入或管道。该对象用于循环遍历, 支持上下文管理协议:

```
fileinput.FileInput(files = None, inplace = False, backup = "", bufsize = 0, mode = 'r', openhook = None)
```

其中, files 是文件路径的元组; mode 是文件打开模式, 默认为文本读取模式。例如:

```
with FileInput(files = ('spam.txt', 'eggs.txt')) as f:
 for line in f:
 process(line)
```



使用 `for...in` 可以循环遍历文件列表中各文件的行。当前读取的文件、行等全局信息可以使用 `FileInput` 对象 `f` 的方法获取。

**f.filename()**: 返回当前读取文件的文件名, 读取第 1 个文件前为 `None`。

**f.fileno()**: 返回当前读取文件的文件名, 无法打开文件时为 `-1`。

**f.lineno()**: 返回累计读取的行数。

**f.filelineno()**: 返回当前文件读取的行数。

**f.isfirstline()**: 判断是否问当前读取文件的首行。

**f.isstdin()**: 判断最后读入的行是否为从标准输入 `stdin` 读入。

**f.nextfile()**: 关闭当前读取文件, 读取下一文件。

**f.close()**: 关闭所有文件。

注: `fileinput` 模块中包含与上述 `FileInput` 对象 `f` 的方法同名的模块函数, 实现相同的功能。

## 7.8.2 fileinput 模块的函数

通常, 使用 `fileinput` 模块的 `input` 函数, 可以返回 `FileInput` 对象:

```
fileinput.input(files = None, inplace = False, backup = '', bufsize = 0, mode = 'r', openhook = None)
```

其中, `files` 是文件路径的元组。例如:

```
with fileinput.input(files = ('spam.txt', 'eggs.txt')) as f:
 for line in f:
 process(line)
```

如果输入的文件为压缩文件, 或者非 `utf8` 编码文件, 则可以使用 `fileinput` 模块的 `hook_compressed` 函数或 `hook_encoded`, 传递给 `FileInput` 的构造函数 `openhook` 参数。

`fileinput.hook_compressed(filename, mode)`: `openhook` 函数, 用于压缩文件。

`fileinput.hook_encoded(encoding)`: `openhook` 函数, 用于编码文件。

压缩文件支持 `'.gz'` 和 `'.bz2'`, 根据后缀自动使用模块 `gzip` 或 `bz2`。例如:

```
fi1 = fileinput.FileInput(openhook = fileinput.hook_compressed)
fi2 = fileinput.FileInput(openhook = fileinput.hook_encoded("iso-8859-1"))
```

**【例 7-13】** `fileinput` 示例 (`fileinput_1.py`)。

```
import fileinput, glob
def main():
 txtfiles = glob.glob(r'c:\python33*.txt')
 with fileinput.input(files = txtfiles) as f:
 for line in f:
 print(f.filename(), f.lineno(), line, end = "\n")
if __name__ == '__main__':
 main()
```

运行结果如下:

```
c:\python33\LICENSE.txt 1 A. HISTORY OF THE SOFTWARE
c:\python33\LICENSE.txt 2 =====
...(略)
```

### 7.8.3 输入重定向

UNIX/Linux 和 Windows 均支持输入重定向。

**【例 7-14】** 使用 fileinput 实现输入重定向 (fileinput\_2.py)。

```
import fileinput
def main():
 with fileinput.input() as f:
 for line in f:
 print(f.filename(), f.lineno(), line, end="")
if __name__ == '__main__':
 main()
```

#### 1. 从文件输入

从文件输入的执行过程和运行结果如下：

```
C:\Python\chapter07 > fileinput_2.py fileinput_1.py fileinput_2.py
filein1.py 1 #chapter20\fileinput_1.py
filein1.py 2 import fileinput, glob
filein1.py 3 txtfiles = glob.glob(r'c:\python33*.txt')
filein1.py 4 with fileinput.input(files = txtfiles) as f:
filein1.py 5 for line in f:
filein1.py 6 print(f.filename(), f.lineno(), line, end="")
filein2.py 7 #chapter20\fileinput_2.py
filein2.py 8 import fileinput
filein2.py 9 with fileinput.input() as f:
filein2.py 10 for line in f:
filein2.py 11 print(f.filename(), f.lineno(), line, end="")
```

#### 2. 从管道输入

从管道输入的执行过程和运行结果如下：

```
C:\Python\chapter07 > dir | fileinput_2.py
<stdin> 1 驱动器 C 中的卷是 Windows8_OS
<stdin> 2 卷的序列号是 0CB1 - ED8B
<stdin> 3
<stdin> 4 C:\Python\chapter07 的目录
...(略)
```

#### 3. 输入重定向

输入重定向的执行过程和运行结果如下：

```
C:\Python\chapter07 > fileinput_2.py < fileinput_2.py
<stdin> 1 #chapter07\fileinput_1.py
<stdin> 2 import fileinput, glob
<stdin> 3 txtfiles = glob.glob(r'c:\python33*.txt')
<stdin> 4 with fileinput.input(files = txtfiles) as f:
```

```
< stdin > 5 for line in f:
< stdin > 6 print(f.filename(), f.lineno(), line, end = ")
```

## 7.9 复习题

### 一、填空题

1. Python 可使用函数\_\_\_\_\_打开文件。
2. 文件操作可以使用\_\_\_\_\_方法关闭流，以释放资源。通常采用\_\_\_\_\_语句，以保证系统自动关闭打开的流。
3. 打开随机文件后，可以使用实例方法\_\_\_\_\_进行定位。
4. \_\_\_\_\_模块提供处理用于循环处理输入、输入重定向、管道或一个或多个文本文件的函数和辅助对象。
5. 可以使用\_\_\_\_\_模块中提供的函数，实现 Python 对象的系列化。

### 二、思考题

1. 流有哪五种基本的操作？
2. 使用 open() 函数时，指定打开文件的模式 mode 有哪几种？其默认打开模式是什么？
3. 文本文件的读取和写入基本步骤是什么？
4. 二进制文件的打开模式是什么？简述其读取和写入的基本步骤。
5. 如何随机访问文件？其打开模式是什么？
6. 如何实现 Python 对象的系列化和反系列化？
7. 如何使用 os 模块提供的函数读取和写入文件？

## 7.10 上机实践

1. 参照例 7-1 编写文本文件的写入程序。
2. 参照例 7-2 编写文本文件的读取程序。
3. 参照例 7-3 编写二进制文件的写入程序。
4. 参照例 7-4 编写文本文件的读取程序。
5. 参照例 7-5 编写使用 reader 对象读取 csv 文件的程序。
6. 参照例 7-6 编写使用 writer 对象写入 csv 文件的程序。
7. 参照例 7-7 编写使用 DictReader 对象读取 csv 文件的程序。
8. 参照例 7-8 编写使用 DictWriter 对象写入 csv 文件的程序。
9. 参照例 7-9 编写 csv 文件格式化参数的程序。
10. 参照例 7-10 编写 Dialect 对象示例程序。
11. 参照例 7-11 编写对象系列化示例程序。
12. 参照例 7-12 编写对象反系列化示例程序。
13. 参照例 7-13 编写 fileinput 示例程序。
14. 参照例 7-14 编写使用 fileinput 实现输入重定向的程序。

## 第 8 章 函数与函数编程

函数是可重用的程序代码段。Python 包括常用的内置函数，如 `len()`、`sum()` 等。Python 模块和程序中也可以自定义函数。使用函数，可以提高编程效率。

本章要点：

- 
- ◆ 函数的声明和调用；
  - ◆ 参数和返回值；
  - ◆ 变量的作用域；
  - ◆ 函数装饰器；
  - ◆ 递归函数；
  - ◆ 作为对象的函数；
  - ◆ Lamda 表达式和匿名函数；
  - ◆ `operator` 模块和运算符函数；
  - ◆ `eval`、`exec` 和 `compile` 函数；
  - ◆ `functools` 模块和函数工具。
- 

### 8.1 函数的声明和调用

函数的声明格式如下：

```
def 函数名([形参列表]):
 函数体
```

函数的调用格式如下：

```
函数名([实参列表])
```

函数名是一种标识符，命名规则为全小写字母，可以使用下划线增加可读性，例如：`my_func`；形参列表（用圆括号括起来，并用逗号隔开，可能为空），表示传递给该函数的值或变量引用；函数体是函数执行的代码块。

函数可以返回值，也可以不返回。如果函数体中包含 `return` 语句，则返回值；否则不返回，即返回值为空（`None`）。

函数调用时，根据需要，可指定实际传入的参数值：

```
函数名([实参列表]);
```

【例 8-1】函数的声明和调用示例（`my_add.py`）。

```
def my_add(a,b): #声明函数,返回值
 return(a + b)
```



```
def my_print(msg): #声明函数,无返回值
 print(msg)
c = my_add(11,22) #调用函数
my_print(c) #调用函数
```

运行结果如下:

33

## 8.2 参数的传递

### 8.2.1 形式参数和实际参数

函数的声明可以包含一个形参列表,而函数调用时则通过传递实参列表,以允许函数体中的代码引用这些参数变量。

声明函数时声明的参数,即形式参数,简称形参;调用函数时,提供函数需要的参数的值,即实际参数,简称实参。

实际参数值默认按位置顺序依次传递给形式参数。如果参数个数不对,会产生错误。

**【例 8-2】**形式参数和实际参数示例(my\_max1.py)。

```
def my_max1(a,b):
 if a > b:print(a,'>',b)
 elif a == b:print(a,'=',b)
 else:print(a,'<',b)
my_max1(1,2)
x = 11;y = 8
my_max1(x,y)
my_max1(1)
```

运行结果如下:

1 < 2

11 > 8

Traceback (most recent call last):

File "C:/Python/chapter08/my\_max1.py", line 9, in <module>

my\_max1(1)

TypeError:my\_max1() missing 1 required positional argument:'b'

### 8.2.2 可选参数

在声明函数时,如果希望函数的一些参数是可选的,可以在声明函数时为这些参数指定默认值。调用该函数时,如果没有传入对应的实参值,则函数使用声明时指定的默认参数值。例如:

```
>>> def babble(words,times = 1): #声明函数 babble,第 2 个参数指定了默认值
 print(words * times)
>>> babble('Hello') #调用函数 babble,只指定了一个参数传给 words,times 使用默认值 1
Hello
```

```
>>> babble('Tiger ',3) #调用函数 babble,传值'Tiger '给 words,3 给 times
Tiger Tiger Tiger
```

**注意：**必须先声明没有默认值的形参，然后声明有默认值的形参。这是因为函数调用时，默认是按位置传递实际参数值的。例如：

```
>>> def my_func(a,b=5):
 pass
>>> def my_func1(a=5,b): #默认值的形参位置不正确
 pass
SyntaxError: non - default argument follows default argument
```

**【例 8-3】** 可选参数示例（my\_sum1.py）。

```
def my_sum1(mid_score,end_score,mid_rate=0.4):
 score = mid_score * mid_rate + end_score * (1 - mid_rate)
 print(format(score,'.2f'))
my_sum1(88,79) #
my_sum1(88,79,0.5) #
```

运行结果如下：

```
82.60
83.50
```

### 8.2.3 位置参数和命名参数

函数调用时，实参默认按位置顺序传递形参。按位置传递的参数称为位置参数。

函数调用时，也通过名称（关键字）指定传入的参数，例如：`my_max1(a=1,b=2)`；`my_max1(b=2,a=1)`。

按名称指定传入的参数称为命名参数，也称为关键字参数。使用关键字参数具有三个优点：参数按名称意义明确；传递的参数与顺序无关；如果有多个可选参数，可选择指定某个参数值。

在带星号的参数后面声明的参数强制为命名参数，如果这些参数没有默认值，且调用时必须使用命名参数赋值，否则会引发错误。

如果不需要带星的参数，只需要强制命名参数，则可以简单地使用一个星号，如 `def total(initial=5,*vegetables)`。

**【例 8-4】** 命名参数示例（my\_sum2.py）。三种调用方式等价。

```
def my_sum2(mid_score,end_score,mid_rate=0.4):
 score = mid_score * mid_rate + end_score * (1 - mid_rate)
 print(format(score,'.2f'))
my_sum2(88,79)
my_sum2(mid_score=88,end_score=79)
my_sum2(end_score=79,mid_score=88)
```

运行结果如下：

```
82.60
82.60
```

82. 60

### 8.2.4 可变参数 (VarArgs)

在声明函数时，通过带星的参数，如 `* param1`，允许向函数传递可变数量的实参。调用函数时，从那一点后所有的参数被收集为一个元组。

在声明函数时，也通过带双星的参数，如 `** param2`，允许向函数传递可变数量的实参。调用函数时，从那一点后所有的参数被收集为一个字典。

带星或双星的参数必须位于形参列表的最后位置。

**【例 8-5】** 可变参数示例 1 (`my_sumVarArgs1.py`)。

```
def my_sum3(a,b,*c):
 total = a + b
 for n in c:
 total = total + n
 return total
print(my_sum3(1,2))
print(my_sum3(1,2,3,4,5))
print(my_sum3(1,2,3,4,5,6,7))
```

运行结果如下：

```
3
15
28
```

**【例 8-6】** 可变参数示例 2 (`my_sumVarArgs2.py`)。

```
def my_sum4(a,b,*c,**d):
 total = a + b
 for n in c:
 total = total + n
 for key in d:
 total = total + d[key]
 return total
print(my_sum4(1,2))
print(my_sum4(1,2,3,4,5))
print(my_sum4(1,2,3,4,5,male=6,female=7))
```

运行结果如下：

```
3
15
28
```

### 8.2.5 强制命名参数 (Keyword – only)

在带星号的参数后面申明参数会导致强制命名参数 (Keyword – only)。调用时必须显式使用命名参数传递值，因为按位置传递的参数默认收集为一个元组，传递给前面带星号的可

变参数。

如果不需要带星的可变参数，只想使用强制命名参数，可以简单地使用一个星号。例如：

```
def my_func(*,a,b,c)
```

**【例 8-7】** 强制命名参数示例（my\_sum5.py）。

```
def my_sum(*,mid_score,end_score,mid_rate=0.4):
 score = mid_score * mid_rate + end_score * (1 - mid_rate)
 print(format(score, '. 2f'))
my_sum(mid_score = 88 ,end_score = 79)
my_sum(end_score = 79 ,mid_score = 88)
my_sum(88,79)
```

运行结果如下：

82.60

82.60

Traceback (most recent call last):

File "C:/Python/chapter08/my\_sum5.py", line 6, in <module>

my\_sum5( 88,79 )

TypeError: my\_sum5() takes 0 positional arguments but 2 were given

## 8.3 函数的返回值

函数可以返回值，即在函数体中使用 return 语句从函数返回一个值，并跳出函数。如果需要返回多个值，则可以返回一个元组。例如：

```
>>> def f1():return 1,2,3
>>> a,b,c = f1()
>>> a,b,c #结果:(1,2,3)
```

**【例 8-8】** 函数的返回值示例（my\_maxReturn.py）。

求若干数中最大值的方法一般如下：

- ① 将最大值的初值设为一个比较小的数，或者取第一个数为最大值的初值；
- ② 利用循环，将每个数与最大值比较，若此数大于最大值，则将此数设置为最大值。

```
def my_max(a,b,*c): #求最大值
 max_value = a
 if max_value < b:
 max_value = b
 for n in c:
 if max_value < n:
 max_value = n
 return max_value
print(my_max(1,2))
print(my_max(1,7,11,2,5))
```



运行结果如下：

```
2
11
```

## 8.4 变量的作用域

变量声明的位置不同，其可被访问的范围也不同。变量的可被访问范围称为变量的作用域。变量按其作用域大致可以分为：全局变量、局部变量和类型成员变量。

### 8.4.1 全局变量

在一个源代码文件中，在函数和类定义之外声明的变量称之为全局变量。全局变量的作用域为其定义的模块，从定义的位置起，直到文件结束位置。

通过 `import` 语句导入模块，也可通过全限定名称“**模块名. 变量名**”访问。或者通过 `from...import` 语句导入模块中的变量并访问。

### 8.4.2 局部变量

在函数体中声明的变量（包括函数参数），称为局部变量。其有效范围（作用域）为函数体。语句块中声明的变量，也是局部变量。例如 `if`、`for`、`while` 等语句中定义的变量，其有效范围（作用域）为控制结构块内。

### 8.4.3 类成员变量

类成员变量是在类中声明的变量，包括静态变量和实例变量。其有效范围（作用域）为类定义体内。

在外部，通过创建类的对象实例，然后通过“**对象. 实例变量**”访问类的实例变量。或者通过“**类. 静态变量**”访问类的静态变量。

### 8.4.4 全局语句 `global`

在函数体中，如果要为定义在函数外的全局变量赋值，可以使用 `global` 语句，表明变量是在外面定义的全局变量。例如：

```
>>> g = 123
>>> def fl():
 global g
 g = 345; print(g)
>>> g; fl() #123 345
>>> g #345
```

`global` 语句可指定多个全局变量，如 `global x, y, z`。一般应该尽量避免这样使用全局变量，因为全局变量会导致程序的可读性差。

**【例 8-9】** 全局语句 `global` 示例（`global.py`）。

```
pi = 3.141592653589793 #全局变量
e = 2.718281828459045 #全局变量
def my_func():
 global pi #全局变量,与前面的全局变量 pi 指向相同的对象
 pi = 3.14
 print('global pi = ',pi)
 e = 2.718 #局部变量,与前面的全局变量 e 指向不同的对象
 print('local e = ',e)
print('module pi = ',pi)
print('module e = ',e)
my_func()
print('module pi = ',pi)
print('module e = ',e)
```

运行结果如下:

```
module pi = 3.141592653589793
module e = 2.718281828459045
global pi = 3.14
local e = 2.718
module pi = 3.14
module e = 2.718281828459045
```

### 8.4.5 非局部语句 nonlocal

在函数体中,可以定义嵌套函数,在嵌套函数中,如果要为定义在上级函数体的局部变量赋值,可以使用 nonlocal 语句,表明变量不是所在块的局部变量,而是在上级函数体中定义的局部变量。nonlocal 语句可指定多个非局部变量,如 nonlocal x, y, z。

【例 8-10】非局部语句 nonlocal 示例 (nonlocal.py)。

```
def outer_func():
 tax_rate = 0.17 #局部变量示例
 print('outerfunc tax rate = ',tax_rate)
 def inner_func():
 nonlocal tax_rate
 tax_rate = 0.05
 print('inner func tax rate = ',tax_rate)
 inner_func()
 print('outer func tax rate = ',tax_rate)
outer_func() #测试代码
```

运行结果如下:

```
outer func tax rate = 0.17
inner func tax rate = 0.05
outer func tax rate = 0.05
```

## 8.5 函数装饰器

### 8.5.1 装饰器的声明和使用

Python 函数装饰器（Decorators）使用下列形式装饰一个函数：

```
@装饰器 1
def 函数 1:
 函数体
```

上述定义相当于：

```
函数 1 = 装饰器 1(函数 1)
```

装饰器实际上就是一个函数，一个用来包装函数的函数。装饰器返回一个修改之后的函数对象，且具有相同的函数签名。

装饰器是一种设计模式，其作用是为已经存在的函数添加额外的功能，插入日志、性能测试、事务处理等。

一个函数定义可以使用多个装饰器，结果与装饰器的位置顺序有关。例如；

```
@ foo
@ spam
def bar():pass
```

等同于：

```
def bar():pass
bar = foo(spam(bar))
```

Python 包含内置的装饰器，例如：staticmethod、classmethod 和 property（参见 9.3.3 节）。用户也可以自定义装饰器。

**【例 8-11】** 函数装饰器示例 1（decorator1.py）。

```
def foo(f):
 """foo fuction Docstring"""
 def wrapper(*x, **y):
 """wrapper fuction Docstring"""
 print('调用函数:', f.__name__)
 return f(*x, **y)
 return wrapper

@ foo
def bar(x):
 """bar fuction Docstring"""
 return x * * 2

#测试代码
if __name__ == '__main__':
 print(bar(2))
 print(bar.__name__)
 print(bar.__doc__)
```

运行结果如下：

```
调用函数: bar
4
wrapper
wrapper fuction Docstring
```

【例 8-12】 函数装饰器示例 2 (decorator2.py)。

```
import time,functools
def timeit(func):
 def wrapper(*s):
 start = time.perf_counter()
 func(*s)
 end = time.perf_counter()
 print('运行时间:',end - start)
 return wrapper
@timeit
def my_sum(n):
 sum = 0
 for i in range(n):sum += i
 print(sum)
#测试代码
if __name__ == '__main__':
 my_sum(100000)
```

运行结果如下：

```
4999950000
运行时间:0.026586049825071918
```

【例 8-13】 函数装饰器示例 3 (decorator3.py)。

```
def makebold(fn):
 def wrappper(*s):
 return "" + fn(*s) + ""
 return wrappper
def makeitalic(fn):
 def wrappper(*s):
 return "<i>" + fn(*s) + "</i>"
 return wrappper
@makebold
@makeitalic
def htmltags(str1):
 return str1
#测试代码
print(htmltags('Hello'))
```

运行结果如下：

```
 <i> Hello </i> s
```



## 8.5.2 @functools.wraps 装饰器

函数包含若干特殊属性，如`__name__`、`__doc__`等。函数被装饰后，其特殊属性将变成相应装饰器函数的特殊属性内容，故使用反射时，可能会导致意外的结果。例 8-11 中，`print(bar.__name__)`的结果为 `wrapper`，而不是 `bar`。

使用 `functools` 模块的 `wraps` 装饰器用于解决上述问题，使用 `wraps` 装饰器装饰函数 `foo`，则 `foo` 装饰的函数的特殊属性保留。

`@functools.wraps(wrapped, assigned = WRAPPER_ASSIGNMENTS, updated = WRAPPER_UPDATES)`

其中，`assigned` 为元组，用于指定使用的 `wrapped` 函数的特殊属性，默认为 `functools` 模块常量；`updated` 为元组，用于指定更新的 `wrapped` 函数的特殊属性。

```
>>> import functools
>>> functools.WRAPPER_ASSIGNMENTS
('__module__', '__name__', '__qualname__', '__doc__', '__annotations__')
>>> functools.WRAPPER_UPDATES
('__dict__',)
```

【例 8-14】 `wraps` 装饰器示例 4 (`decorator4.py`)。

```
import time,functools
def foo(f):
 """foo fuction Docstring"""
 @functools.wraps(f)
 def wrapper(*x,**y):
 """wrapper fuction Docstring"""
 print('调用函数:',f.__name__)
 return f(*x,**y)
 return wrapper
@foo
def bar(x):
 """bar fuction Docstring"""
 return x**2
#测试代码
if __name__ == '__main__':
 print(bar(2))
 print(bar.__name__)
 print(bar.__doc__)
```

运行结果如下：

```
调用函数: bar
4
bar
bar fuction Docstring
```

## 8.5.3 functools.update\_wrapper 函数

使用 `functools` 模块的 `update_wrapper` 函数，也可以指定 `wrapper` 函数使用指定的特殊

属性。

```
functools.update_wrapper(wrapper, wrapped, assigned = WRAPPER_ASSIGNMENTS, updated = WRAPPER_UPDATES)
```

其中，assigned 和 updated 的含义参见 8.5.2 节。

【例 8-15】 update\_wrapper 函数示例 5（update\_wrapper.py）。

```
import time,functools
def foo(f):
 """foo fuction Docstring"""
 def wrapper(*x,**y):
 """wrapper fuction Docstring"""
 print('调用函数:',f.__name__)
 return f(*x,**y)
 return functools.update_wrapper(wrapper,f)
@foo
def bar(x):
 """bar fuction Docstring"""
 return x**2
#测试代码
if __name__ == '__main__':
 print(bar(2))
 print(bar.__name__)
 print(bar.__doc__)
 print(bar.__name__)
 print(bar.__doc__)
```

运行结果如下：

```
调用函数: bar
4
bar
bar fuction Docstring
bar
bar fuction Docstring
```

## 8.6 递归函数

递归函数也称自调用函数，可在函数体内部直接或间接地自己调用自己，即函数的嵌套调用是函数本身。递归函数不能无限递归，否则会耗尽内存。一般在递归函数中，需要设置终止条件。sys 模块中，函数 getrecursionlimit 和 setrecursionlimit 用于获取和设置最大递归次数。例如：

```
>>> import sys
>>> sys.getrecursionlimit() #获取最大递归次数(1000)
>>> sys.setrecursionlimit(100) #设置最大递归次数为 100
```

递归函数常用来实现数值计算的方法。例如，非负整数的阶乘定义为：

$n! = 1 * 2 * 3 * \cdots * n$ , 当  $n=0$  时,  $n! = 1$

故可以使用递归函数来实现：

```
factorial(n) = 1 #当 n == 0 时
```

```
factorial(n) = n * factorial(n - 1) #当 n > 0 时
```

**【例 8-16】** 使用递归函数实现阶乘（factorial.py）。

```
def factorial(num):
 if num == 0:
 return 1
 else:
 return num * factorial(num - 1)

#测试代码
for i in range(10): print(factorial(i), end = '')
```

运行结果如下：

```
1 1 2 6 24 120 720 5040 40320 362880
```

## 8.7 作为对象的函数

Python 语言中，函数也是对象，故函数对象可以赋值给变量。例如：

```
>>> f = abs
>>> type(f) # < class 'builtin_function_or_method' >
>>> f(-123) #123
```

函数对象也可以作为参数传递给函数，还可以作为函数的返回值。例如：

```
>>> def compute(f,s): #f 为函数对象,s 为系列对象
 return f(s)
>>> compute(min,(1,5,3,2)) #1
>>> compute(max,(1,5,3,2)) #5
```

## 8.8 Lambda 表达式和匿名函数

Lambda 是一种简便的、在同一行中定义函数的方法。Lambda 实际上生成一个函数对象，即匿名函数。例如：

```
>>> f = lambda x,y:x + y
>>> type(f) # < class 'function' >
>>> f(12,34) #46
```

Lambda 表达式的基本格式为：

**lambda arg1, arg2...: <expression>**

其中 arg1, arg2...为函数的参数，<expression> 为函数的语句，其结果为函数的返回值。

例如，语句“lambda x,y:x + y”，生成一个函数对象，函数参数为“x, y”，返回值为 x + y。

匿名函数广泛用于需要函数对象作为参数，函数比较简单且只使用一次的场合。例如，

map() (参见 11.6 节) 的第一个参数为函数对象, 可使用匿名对象:

```
>>> i = map(lambda x:x*3,(1,2,3))
>>> for t in i:print(t,end=' ') #3 6 9
```

## 8.9 operator 模块和运算符函数

operator 模块提供了一系列的函数, 这些函数封装了运算符的功能。例如:

```
>>> import operator
>>> dir(operator)
['_abs_', '_add_', '_and_', '_concat_', '_contains_', '_delitem_', '_doc_', '_eq_', '_floordiv_', '_ge_', '_getitem_', '_gt_', '_iadd_', '_iand_', '_iconcat_', '_ifloordiv_', '_ilshift_', '_imod_', '_imul_', '_index_', '_inv_', '_invert_', '_ior_', '_ipow_', '_irshift_', '_isub_', '_itruediv_', '_ixor_', '_le_', '_loader_', '_lshift_', '_lt_', '_mod_', '_mul_', '_name_', '_neg_', '_neg_', '_not_', '_or_', '_package_', '_pos_', '_pow_', '_rshift_', '_setitem_', '_sub_', '_truediv_', '_xor_', '_compare_digest', 'abs', 'add', 'and_', 'attrgetter', 'concat', 'contains', 'countOf', 'delitem', 'eq', 'floordiv', 'ge', 'getitem', 'gt', 'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift', 'imod', 'imul', 'index', 'indexOf', 'inv', 'invert', 'ior', 'ipow', 'irshift', 'is_', 'is_not', 'isub', 'itemgetter', 'itruediv', 'ixor', 'le', 'lshift', 'lt', 'methodcaller', 'mod', 'mul', 'ne', 'neg', 'not_', 'or_', 'pos', 'pow', 'rshift', 'setitem', 'sub', 'truediv', 'truth', 'xor']
>>> operator.add(12,34) #等同于操作符+, 即 12 + 34 = 46
```

把运算符封装为函数对象, 可以作为参数传递给需要函数对象作为参数的场合, 实现灵活编程的需求。例如:

```
>>> import operator
>>> func = lambda f,x,y:f(x,y)
>>> func(operator.add,12,34) #46
```

有关 operator 模块提供的函数与运算符的对应关系, 请参见 Python 帮助文档。

## 8.10 eval、exec 和 compile 函数

### 8.10.1 运行上下文的局部变量和全局变量

程序运行过程中, 在上下文中会生成各种局部变量和全局变量。使用内置函数 globals() 和 locals(), 可以返回它们的字典映射表。

无参内置函数 vars(), 等同于 locals(); 有参内置函数 vars(object) 返回指定对象的字典列表, 等同于 object.\_\_dict\_\_。例如:

```
>>> globals()
{'__builtins__': <module 'builtins' (built-in)>, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__name__': '__main__', '__doc__': None}
>>> locals()
{'__builtins__': <module 'builtins' (built-in)>, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__name__': '__main__', '__doc__': None} ... (略)
```



```
>>> vars(str)
mappingproxy({'rstrip': <method 'rstrip' of 'str' objects>, ...}(略))
```

### 8.10.2 eval 函数和动态表达式的求值

使用内置的 eval 函数，可以对动态表达式进行求值：

```
eval(expression, globals = None, locals = None)
```

其中，expression 是动态表达式的字符串；globals 和 locals 是求值时使用的上下文环境的全局变量和局部变量，如果不指定，则使用当前运行上下文。例如：

```
>>> x = 2
>>> str_func = input("请输入表达式:")
请输入表达式:x * * 2 + 2 * x + 1
>>> eval(str_func) #9
```

eval 函数的功能是将字符串生成语句执行，如果字符串包含不安全的语句，则存在注入安全隐患。例如删除文件的语句。

### 8.10.3 exec 函数和动态语句的执行

使用内置的 exec 函数，可以执行动态语句：

```
exec(str[, globals[, locals]])
```

其中，str 是动态语句的字符串；globals 和 locals 是使用的上下文环境的全局和局部变量，如果不指定，则使用当前运行上下文。

通常，eval 用于动态表达式求值，返回一个值；exec 用于动态语句的执行，不返回值。同样，exec 也存在注入安全隐患。例如：

```
>>> exec("for i in range(10):print(i,end=' ')") #0 1 2 3 4 5 6 7 8 9
```

### 8.10.4 compile 函数和动态语句的执行

使用内置的 compile 函数，可以编译代码为代码对象：

```
compile(source, filename, mode) #返回代码对象
```

其中，source 为代码语句的字符串；如果是多行语句，则每一行的结尾必须有换行符 \n。filename 为包含代码的文件。mode 为编译模式，可以为：'exec'（用于语句系列执行）、'eval'（用于表达式求值）和'single'（用于单个交互语句）。

编译后的代码对象可以通过 eval 函数或 exec 函数执行，因为编译为代码对象，所以可以提高效率。例如：

```
>>> co = compile("for i in range(10):print(i,end=' ')", '', 'exec')
>>> exec(co) #0 1 2 3 4 5 6 7 8 9
```

## 8.11 functools 模块和函数工具

标准库 functools 提供了若干关于函数的函数，这些工具函数以函数作为参数，返回修改后的函数。functools 提供了 Haskell 和 Standard ML 中的函数式程序设计工具。

@functools.wraps 装饰器请参见 8.5.2 节；functools.update\_wrapper 函数请参见 8.5.3 节；@functools.total\_ordering 装饰器请参见 9.4.8 节。

### 8.11.1 partial 对象

重新绑定函数的可选参数，生成一个可调用（callable）的 partial 对象：

**functools.partial(func, \*args, \*\*keywords)**

其中，func 是函数，args 是其位置参数，keywords 为关键字参数。

partial 主要用于设置预先已知的参数，从而减少调用时传递参数的个数。例如：

```
>>> import functools, math
>>> pow2 = functools.partial(math.pow, 2) #封装 pow(x,y[,z]),指定参数 x=2
>>> for i in range(11): #输出 2 的 0~10 次方
 print(format(pow2(i), '0.0f'), end = ' ')
1 2 4 8 16 32 64 128 256 512 1024
```

### 8.11.2 reduce 函数

reduce 使用指定的带两个参数的函数 func，对一个数据集合（可迭代对象中）的所有数据进行下列操作：使用第 1、2 个数据作为参数用 func 函数运算，得到的结果再与第 3 个数据作为参数用 func() 函数运算，以此类推，最后得到一个结果。

**functools.reduce(func, iterable[, initializer])**

其中，func 为函数名，iterable 为可迭代对象，可选的 initializer 为初始值。例如：

```
>>> import functools, operator
>>> functools.reduce(operator.add, [1, 2, 3, 4, 5]) # (((1+2)+3)+4)+5 = 15
15
>>> functools.reduce(operator.add, [1, 2, 3, 4, 5], 10) # 10 + (((1+2)+3)+4)+5 = 25
25
```

### 8.11.3 @functools.lru\_cache 装饰器

使用 functools 模块的 lru\_cache 装饰器，可以缓存最多 maxsize 个此函数的调用结果，从而提高程序执行的效率。特别适合于耗时的函数或 I/O 函数。

**@functools.lru\_cache(maxsize=128, typed=False)**

其中，maxsize 为最多缓存的次数，如果为 None，则无限制，设置为 2^n 时，性能最佳；如果 typed=True，则不同参数类型的调用将分别缓存，例如 f(3) 和 f(3.0)。

**【例 8-17】** lru\_cache 装饰器示例（lru\_cache\_fibs.py）。

```
import functools
@functools.lru_cache(maxsize=None)
def fib(n):
 if n < 2: return n
 return fib(n-1) + fib(n-2)
#测试代码
if __name__ == '__main__':
```

```
for i in range(16):
 print(fib(i),end='')
print('\n',fib.cache_info())
```

运行结果如下:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
CacheInfo(hits = 28, misses = 16, maxsize = None, cursize = 16)
```

## 8.12 复习题

### 一、单选题

1. Python 语句 `print ( type ( lambda: None ) )` 的输出结果是\_\_\_\_\_。  
A. `< class 'NoneType' >`                      B. `< class 'tuple' >`  
C. `< class 'type' >`                              D. `< class 'function' >`
2. Python 语句 `f1 = lambda x:x * 2;f2 = lambda x:x * * 2;print( f1( f2(2) ) )` 的运行结果是\_\_\_\_\_。  
A. 2                                      B. 4                                      C. 6                                      D. 8
3. Python 中, 若 `def f1 ( p, * * p2 ): print ( type ( p2 ) )`, 则 `f1 ( 1, a = 2 )` 的运行结果是\_\_\_\_\_。  
A. `< class 'int' >`              B. `< class 'str' >`      C. `< class 'dict' >`      D. `< class 'list' >`
4. Python 中, 若 `def f1 ( a, b, c ): print ( a + b )`, 则 `nums = ( 1, 2, 3 ); f1 ( * nums )` 的运行结果是\_\_\_\_\_。  
A. 语法错                      B. 6                                      C. 3                                      D. 1

### 二、填空题

1. Python 表达式 `eval( "5/2 + 5 % 2 + 5//2" )` 的结果是\_\_\_\_\_。
2. 使用 Python 关键字\_\_\_\_\_, 可以在一个函数中设置一个全局的变量。
3. 变量按其作用域大致可以分为: \_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。
4. Python 的 sys 模块中, 函数\_\_\_\_\_和\_\_\_\_\_分别用于获取和设置最大递归次数。

### 三、思考题

1. Python 如何定义一个函数?
2. 什么是 Lambda 函数?
3. 什么是递归函数? 在递归函数使用过程中, 为什么需要设置终止条件?
4. 下列 Python 语句的输出结果是\_\_\_\_\_。  

```
d = lambda p:p * 2;t = lambda p:p * 3
x = 2;x = d(x);x = t(x);x = d(x);print(x)
```
5. 下列 Python 语句的输出结果是\_\_\_\_\_。  

```
i = map(lambda x:x * * 2,(1,2,3))
for t in i:print(t,end = '')
```
6. 下列 Python 语句的运行结果为\_\_\_\_\_。

```
def fl():
 "simple function"
 pass
print(fl.__doc__)
```

7. 下列 Python 语句的输出结果是\_\_\_\_\_。

```
counter = 1; num = 0
def TestVariable():
 global counter
 for i in (1,2,3): counter += 1
 num = 10
TestVariable(); print(counter, num)
```

8. 下面 Python 程序的功能是什么？

```
def f(a,b):
 if b == 0: print(a)
 else: f(b,a%b)
print(f(9,6))
```

9. 下列 Python 语句的输出结果是\_\_\_\_\_。

```
def aFunction():
 "The quick brown fox"
 return 1
print(aFunction.__doc__[4:9])
```

10. 下列 Python 语句的输出结果是\_\_\_\_\_。

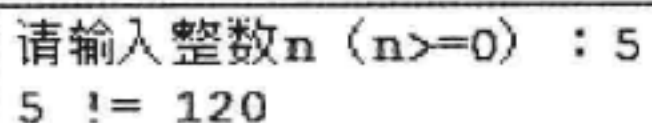
```
def judge(param1, * param2):
 print(type(param2))
 print(param2)
judge(1,2,3,4,5)
```

11. 下列 Python 语句的输出结果是\_\_\_\_\_。

```
def judge(param1, ** param2):
 print(type(param2))
 print(param2)
judge(1, a=2, b=3, c=4, d=5)
```

## 8.13 上机实践

1. 编写程序，定义一个求阶乘的函数 `fact(n)`，并编写测试代码，要求输入整数 `n` ( $n \geq 0$ )。运行效果参见图 8-1 所示。



```
请输入整数n (n>=0) : 5
5 != 120
```

图 8-1 阶乘运行效果

2. 编写程序，定义一个求 Fibonacci 数列的函数 `fib(n)`，并编写测试代码，输出前 20 项



(每项宽度 5 个字符位置，右对齐)，每行输出 10 个。运行效果参见图 8-2 所示。

|    |     |     |     |     |     |      |      |      |      |
|----|-----|-----|-----|-----|-----|------|------|------|------|
| 1  | 1   | 2   | 3   | 5   | 8   | 13   | 21   | 34   | 55   |
| 89 | 144 | 233 | 377 | 610 | 987 | 1597 | 2584 | 4181 | 6765 |

图 8-2 Fibonacci 运行效果

3. 编写程序，利用可变参数定义一个求任意个数数值的最小值的函数 min\_n(a,b,\*c)，并编写测试代码。

4. 编写程序，利用元组作为函数的返回值，求序列类型中的最大值、最小值和元素个数，并编写测试代码，假设测试数据分别为 s1 = [9,7,8,3,2,1,55,6]、s2 = ["apple","pear","melon","kiwi"] 和 s3 = "TheQuickBrownFox"。提示：函数形参为序列类型，返回值是形如 “（最大值，最小值，元素个数）” 的元组。运行效果参见图 8-3 所示。

```
list= [9, 7, 8, 3, 2, 1, 55, 6]
最大值= 55 ,最小值= 1 ,元素个数= 8
list= ['apple', 'pear', 'melon', 'kiwi']
最大值= pear ,最小值= apple ,元素个数= 4
list= TheQuickBrownFox
最大值= x ,最小值= B ,元素个数= 16
```

图 8-3 元组作为函数返回值运行效果

# 第9章 类和对象

类是一种数据结构，可以包含数据成员和函数成员。程序中可以定义类，并创建和使用其对象实例。

## 本章要点：

---

- ◆ 面向对象的基本概念；
  - ◆ 类的声明、对象的创建和使用；
  - ◆ 类的成员定义；
  - ◆ 类的继承；
  - ◆ 对象的引用、浅拷贝和深拷贝。
- 

## 9.1 面向对象概念

面向对象的程序设计具有三个基本特征：封装、继承和多态，可以大大增加程序的可靠性、代码的可重用性和程序的可维护性，从而提高程序开发效率。

### 9.1.1 对象的定义

所谓的对象（Object），从概念层面讲，就是某种事物的抽象（功能）。抽象原则包括数据抽象和过程抽象两个方面：数据抽象就是定义对象的属性；过程抽象就是定义对象的操作。

面向对象的程序设计强调把数据（属性）和操作（服务）结合为一个不可分的系统单位（即对象），对象的外部只需要知道它做什么，而不必知道它如何做。

从规格层面讲，对象是一系列可以被其他对象使用的公共接口（对象交互）。从语言实现层面来看，对象封装了数据和代码（数据和程序）。

### 9.1.2 封装

封装（Encapsulation）是面向对象的主要特性。所谓封装，也就是把客观事物抽象并封装成对象，即将数据成员、属性、方法和事件等集合在一个整体内。通过访问控制，还可以隐藏内部成员，只允许可信的对象访问或操作自己的部分数据或方法。

封装保证了对象的独立性，可以防止外部程序破坏对象的内部数据，同时便于程序的维护和修改。

9.1.3 继承

继承（Inheritance）是面向对象的程序设计中代码重用的主要方法。继承是允许使用现有类的功能，并在无需重新改写原来的类的情况下，对这些功能进行扩展。继承可以避免代码复制和相关的代码维护等问题。

继承的过程，就是从一般到特殊的过程。被继承的类称为“基类（Base Class）”、“父类”或“超类（Super Class）”，通过继承创建的新类称为“子类（Subclass）”或“派生类（Derived Class）”。例如，长方形是一个四边形。因此，Rectangle（长方形）类继承于Quadrilateral（四边形）类。Quadrilateral 是基类，Rectangle 是派生类。也就是说，长方形是一种特殊的四边形，但反过来说四边形是长方形是不正确的，因为四边形还可能是梯形、平行四边形或其他类型的四边形。表 9-1 列出了基类和派生类的几个简单例子。

表 9-1 继承示例

| 基 类           | 派 生 类                                    |
|---------------|------------------------------------------|
| Quadrilateral | Trapezoid、Parallelogram、Rectangle、Square |
| Shape         | Rectangle、Triangle、Circle                |
| Degree        | Doctor、Master、Bachelor                   |

9.1.4 多态性

派生类具有基类的所有非私有数据和行为，以及新类自己定义的所有其他数据或行为，即子类具有两个有效类型：子类的类型及其继承的基类的类型。对象可以表示多个类型的能力称为多态性（Polymorphism）。

多态性允许每个对象以自己的方式去响应共同的消息，从而允许用户以更明确的方式建立通用软件，提高软件开发的可维护性。

例如，假设设计了一个绘图软件，所有的图形（Square、Circle 等）都继承于基类 Shape，每种图形有自己特定的绘制方法（draw）的实现。如果要显示画面的所有图形，则可以创建一个基类 Shape 的集合，其元素分别指向各子类对象，然后循环调用父类类型对象的绘制方法（draw），实际绘制根据当前赋值给它的子对象调用各自的绘制方法（draw），这就是多态性。如果要扩展软件的功能，例如增加图形 Ellipse，则只需要增加新的子类，并实现其绘制方法（draw）即可。

Python 是一种动态语言，变量或参数无法也无需确定其类型。程序运行过程中，根据实际的对象类型确定变量的类型。严格意义上讲，Python 不支持多态性，即动态确定变量指向的对象类型。

9.2 类和对象

类（class）是 Python 语言的核心，Python 3 的一切类型都是类，包括内置的 int、str 等。类是一个数据结构，类定义数据类型的数据（属性）和行为（方法）。对象是类的具体



实体，也可称为类的实例（Instance）。

类与对象的关系类似于车型设计和具体的车。车型设计（类）描述了该车型所应该具备的属性和功能，但车型设计并不是具体的车，不能发动和驾驶车型设计。相应型号的车（对象），则是根据车型设计制造出的车（类的实例），它们都具备该车型设计所描述的属性和功能，可以发动和驾驶。

### 9.2.1 类的声明

类使用关键字 `class` 声明。类的声明格式如下：

**class** 类名：

类体

其中，类名为有效的标识符，命名规则一般为多个单词组成的名称，每个单词除第一个字母大写外，其余的字母均小写；类体由缩进的语句块组成。

定义在类体内的元素都是类的成员。类的主要成员包括两种类型，即描述状态的数据成员（属性）和描述操作的函数成员（方法）。

【例 9-1】定义类 `Person1`（`Person1.py`）。

```
class Person1: #定义类 Person1
 pass #类体为空语句
p1 = Person1() #创建和使用对象
print(p1)
```

### 9.2.2 对象的创建和使用

类是抽象的，要使用类定义的功能，就必须实例化类，即创建类的对象。创建对象后，可以使用“.”运算符来调用其成员。

**注意：**创建类的对象、创建类的实例、实例化类等说法是等价的，都说明以类为模板生成了一个对象的操作。

对象的创建和调用格式如下：

**anObject = 类名(参数列表)**

**anObject.对象函数 或 anObject.对象属性**

例如：

```
>>> c1 = complex(1,2) #创建类 complex 的实例对象
>>> c1.conjugate() #c1.conjugate()调用对象 c1 的 conjugate()方法,返回其共轭值:(1-2j)
>>> c1.real #c1.real 引用对象 c1 的实部:1.0
```

## 9.3 属性

类的数据成员是在类中定义的成员变量，用来存储描述类的特征的值，称为属性。属性可以被该类中定义的方法访问，也可以通过类或类的实例进行访问。而在函数体或代码块中定义的局部变量，则只能在其定义的范围内进行访问。

属性实际上是在类中的变量。Python 变量不需要声明，可直接使用。建议在类定义的开



始位置初始化类属性，或者在构造函数（\_\_init\_\_）中初始化实例属性。

### 9.3.1 实例属性和类属性

#### 1. 实例属性

通过 self. 变量名定义的属性，称为实例属性，也称为实例变量。类的每个实例都包含了该类的实例变量一个单独副本，实例变量属于特定的实例。实例变量在类的内部通过 self 访问，在外部通过对象实例访问。

实例属性一般在\_\_init\_\_方法中通过如下形式初始化：

**self. 实例变量名 = 初始值**

然后，在其他实例函数中，通过 self 访问：

**self. 实例变量名 = 值**            **#写入**

**self. 实例变量名**            **#读取**

或者，创建对象实例后，通过对象实例访问：

**obj1 = 类名()**            **#创建对象实例**

**obj1. 实例变量名 = 值**        **#写入**

**obj1. 实例变量名**            **#读取**

**【例 9-2】** 定义类 Person2（Person2.py）。定义成员变量（域）。

```
class Person2: #定义类 Person2
 def __init__(self, name, age): #__init__方法
 self.name = name #初始化 self.name, 即成员变量 name(域)
 self.age = age #初始化 self.age, 即成员变量 age(域)
 def say_hi(self): #定义类 Person2 的函数 sayHi
 print('您好, 我叫', self.name) #在实例方法中通过 self.name 读取成员变量 name(域)

p1 = Person2('张三', 25) #创建对象
p1. say_hi() #调用对象的方法
print(p1. age) #通过 p1. age(obj1. 变量名) 读取成员变量 age(域)
```

运行结果如下：

```
您好, 我叫 张三
25
```

#### 2. 类属性

Python 也允许声明属于类本身的变量，即类属性，也称为类变量、静态属性。类属性属于整个类，不是特定实例的一部分，而是所有实例之间共享一个副本。

类属性一般在类体中通过如下形式初始化：

**类变量名 = 初始值**

然后，在其类定义的方法中或外部代码中，通过类名访问：

**类名. 类变量名 = 值**    **#写入**

**类名. 类变量名**        **#读取**

**【例 9-3】** 定义类 Person3（Person3.py）。定义类域和类方法。

```
class Person3:
```

```

count = 0 #定义类域 count,表示计数
def __init__(self, name, age): #构造函数
 self. name = name
 self. age = age
 Person3. count + = 1 #创建一个实例时,计数加 1
def __del__(self): #析构函数
 Person3. count - = 1 #销毁一个实例时,计数减 1
def say_hi (self):
 print('您好,我叫', self. name)
def get_count(): #创建类方法
 print('总计数为:', Person3. count)
print('总计数为:', Person3. count) #类名访问
p31 = Person3('张三', 25) #创建对象
p31. say_hi() #调用对象的方法
Person3. get_count() #通过类名访问
p32 = Person3('李四', 28) #创建对象
p32. say_hi() #调用对象的方法
Person3. get_count() #通过类名访问
del p31 #删除对象 p31
Person3. get_count() #通过类名访问
del p32 #删除对象 p32
Person3. get_count() #通过类名访问

```

运行结果如下:

```

总计数为: 0
您好,我叫 张三
总计数为: 1
您好,我叫 李四
总计数为: 2
总计数为: 1
总计数为: 0

```

类属性如果通过“obj. 属性名”来访问,则属于该实例的实例属性。例如:

```

>>> class A: #定义类
 name = 'aaa'
>>> a = A() #创建对象实例
>>> b = A() #创建对象实例
>>> a. name, b. name #('aaa', 'aaa')
>>> A. name = 'AAA'
>>> a. name, b. name #('AAA', 'AAA')
>>> b. name = 'bbb'
>>> a. name, b. name #('AAA', 'bbb')

```

虽然类属性可以使用对象实例来访问,但这样容易造成困惑。所以建议不要这样使用,

而是应该使用标准的访问方式：

类名. 类变量名

### 9.3.2 私有属性和公有属性

Python 类的成员没有访问控制限制，这与其他面向对象的语言不同。

通常，约定两个下划线开头，但是不以两个下划线结束的属性是私有的（Private），其他为公共的（Public）。不能直接访问私有属性，但可以在方法中访问。例如：

```
>>> class A:
 __name = 'class A' #私有类属性
 def get_name():
 print(A.__name) #在类方法中访问私有类属性
>>> A.get_name() #class A
>>> A.__name #不能直接访问私有类属性
#AttributeError: type object 'A' has no attribute '__name'
```

### 9.3.3 @property 装饰器

面向对象编程的封装性原则要求不直接访问类中的数据成员。Python 中可以通过定义私有属性，然后定义相应的访问该私有属性的函数，并使用@ property 装饰器装饰这些函数。程序可以把函数“当做”属性访问，从而提供更加友好访问方式。

【例 9-4】property 装饰器示例 1（property1.py）。

```
class Person11:
 def __init__(self, name):
 self.__name = name
 @property
 def name(self):
 """I'm the 'x' property. """
 return self.__name
#测试代码
if __name__ == '__main__':
 p = Person11('王五')
 print(p.name)
```

运行结果如下：

王五

@property 装饰器默认提供一个只读属性，如果需要，可以使用对应的 getter、setter 和 deleter 装饰器实现其他访问器函数。

【例 9-5】property 装饰器示例 2（property2.py）。

```
class Person12:
 def __init__(self, name):
 self.__name = name
```

```

@ property
def name(self):
 """ I 'm the 'x 'property. """
 return self. __name

@ name. setter
def name(self, value):
 self. __name = value

@ name. deleter
def name(self):
 del self. __name

#测试代码
if __name__ == '__main__':
 p = Person12('姚六')
 p. name = '王依依'
 print(p. name)

```

运行结果如下：

王依依

property 的调用格式为：

**property( fget = None, fset = None, fdel = None, doc = None)**

其中，fget 为 get 访问器；fset 为 set 访问器；fdel 为 del 访问器。

**【例 9-6】** property 装饰器示例 3（property3. py）。

```

class Person13:
 def __init__(self, name):
 self. __name = name
 def getname(self):
 return self. __name
 def setname(self, value):
 self. __name = value
 def delname(self):
 del self. __name
 name = property(getname, setname, delname, "I 'm the 'name 'property. ")

#测试代码
if __name__ == '__main__':
 p = Person13('爱丽丝'); print(p. name)
 p. name = '罗伯特'; print(p. name)

```

运行结果如下：

爱丽丝

罗伯特

### 9.3.4 特殊属性（Special Attributes）

Python 对象中包含许多以双下划线开始和结束的方法，称为特殊属性。常用的特殊属性



如表 9-2 所示。假设示例基于 i = 123。

表 9-2 Python 特殊属性

| 特殊方法                    | 含 义          | 示 例                                                                                                                |
|-------------------------|--------------|--------------------------------------------------------------------------------------------------------------------|
| object. __dict__        | 对象的属性字典      | >>> int. __dict__<br>mappingproxy({'__sizeof__': <method '__sizeof__' of 'int' objects> ... (略)                    |
| instance. __class__     | 对象所属的类       | >>> i. __class__ # <class 'int'><br>>>> int. __class__ # <class 'type'>                                            |
| class. __bases__        | 类的基类元组       | >>> int. __bases__ #( <class 'object'> , )                                                                         |
| class. __base__         | 类的基类         | >>> int. __base__ # <class 'object'>                                                                               |
| class. __name__         | 类的名称         | >>> int. __name__ #'int '                                                                                          |
| class. __qualname__     | 类的限定名称       | >>> int. __qualname__ #'int '                                                                                      |
| class. __mro__          | 方法查找顺序, 基类元组 | >>> int. __mro__ #( <class 'int'> , <class 'object'> )                                                             |
| class. mro()            | 同上, 可被子类重写   | >>> int. mro() #[ <class 'int'> , <class 'object'> ]                                                               |
| class. __subclasses__() | 子类列表         | >>> int. __subclasses__()<br>[ <class 'bool'> , <class 'inspect. _ParameterKind'> , <class 'subprocess. Handle'> ] |

9.3.5 自定义属性 (Custom Attributes)

Python 中, 可以赋予一个对象自定义的属性, 即类定义中不存在的属性。对象通过特殊属性\_\_dict\_\_存储自定义属性。例如:

```
>>> class C1:
 pass
>>> o = C1()
>>> o.name = 'custom name'
>>> o.name #'custom name'
>>> o.__dict__ #{'name': 'custom name'}
```

通过重载\_\_getattr\_\_和\_\_setattr\_\_, 可以拦截对成员的访问, 从而自定义属性的行为。\_\_getattr\_\_只有在访问不存在的成员时才会被调用, \_\_getattribute\_\_拦截所有(包括不存在的成员)的获取操作。在\_\_getattribute\_\_中不要使用"return self.\_\_dict\_\_[name]"来返回结果, 因为在访问"self.\_\_dict\_\_"时同样会被\_\_getattribute\_\_拦截, 从而造成无限递归形成死循环。

- \_\_getattr\_\_(self, name): 获取属性, 比\_\_getattribute\_\_优先调用。
- \_\_getattribute\_\_(self, name): 获取属性。
- \_\_setattr\_\_(self, name, value): 设置属性。
- \_\_delattr\_\_(self, name): 删除属性。

【例 9-7】自定义属性示例 (custom\_attribute.py)。

```
class CustomAttribute(object):
 def __init__(self):
 pass
 def __getattribute__(self, name):
```

```
 return str.upper(object.__getattr__(self, name))
 def __setattr__(self, name, value):
 object.__setattr__(self, name, str.strip(value))
#测试代码
if __name__ == '__main__':
 o = CustomAttribute()
 o.firstname = ' mary '
 print(o.firstname)
```

运行结果如下：

MARY

## 9.4 方法

### 9.4.1 方法的声明和调用

方法是与类相关的函数。Python 也允许定义与类不相关的普通全局函数（参见第8章）。类方法的定义与普通的函数一致，唯一的区分是类的方法定义在类体中，且其第一个形式参数必须为对象本身，通常为 `self`。方法的声明格式如下：

```
def 方法名(self, [形参列表]):
 函数体
```

方法的调用格式如下：

```
对象. 方法名([实参列表])
```

值得注意的是，虽然类方法的第一个参数为 `self`，但调用时，用户不需要也不能给该参数传值。事实上，Python 自动把对象实例传递给该参数。

例如，假设声明了一个类 `MyClass` 和类方法 `my_func(self, p1, p2)`，则：

```
obj1 = MyClass() #创建 MyClass 的对象实例 obj1
obj1.my_func(p1, p2) #调用对象 obj1 的方法
```

调用对象 `obj1` 的方法 `obj1.my_func(p1, p2)`，Python 自动转换为：`obj1.my_func(obj1, p1, p2)`，即自动把对象实例 `obj1` 传值给 `self` 参数。

**注：**Python 中的 `self` 等价于 C++ 中的 `self` 指针和 Java、C# 中的 `this` 关键字。虽然没有限制第一个参数名必须为 `self`，但建议读者遵循惯例，这样便于阅读和理解，且集成开发环境（IDE）也会提供相应的支持。

**【例 9-8】** 定义类 `Person4`（`PersonMethod.py`），创建其对象，并调用对象函数。

```
class Person4: #定义类 Person
 def say_hi(self, name): #定义方法 say_hi
 self.name = name #把参数 name 赋值给 self.name, 即成员变量 name(域)
 print('您好, 我叫', self.name)
p4 = Person4() #创建对象
p4.say_hi('Alice') #调用对象的方法
```

运行结果如下：

您好,我叫 Alice

### 9.4.2 \_\_init\_\_方法(构造函数)和\_\_new\_\_方法

Python 类体中,可以定义特殊的方法:\_\_new\_\_和\_\_init\_\_方法。

\_\_new\_\_是一个类方法,创建对象时调用,返回当前对象的一个实例,一般无需重载该方法。

\_\_init\_\_方法即构造函数(构造方法),用于执行类的实例的初始化工作。创建完对象后调用,初始化当前对象的实例,无返回值。

**【例 9-9】** \_\_init\_\_方法示例 1 (PersonInit.py)。定义类 Person5,并调用对象的方法。

```
class Person5: #定义类 Person5
 def __init__(self,name): #__init__方法
 self.name = name #把参数 name 赋值给 self.name,即成员变量 name(域)
 def say_hi(self): #定义类 Person 的函数 sayHi
 print('您好,我叫',self.name)
p5 = Person5('Helen') #创建对象
p5.say_hi() #调用对象的方法
```

运行结果如下：

您好,我叫 Helen

**【例 9-10】** \_\_init\_\_方法示例 2 (PointInit.py)。定义类 Point,表示平面坐标点。

```
class Point:
 def __init__(self,x=0,y=0): #构造函数
 self.x = x
 self.y = y
p1 = Point() #创建对象
print("p1({0},{1})".format(p1.x,p1.y))
p1 = Point(5,5) #创建对象
print("p1({0},{1})".format(p1.x,p1.y))
```

运行结果如下：

p1(0,0)  
p1(5,5)

### 9.4.3 \_\_del\_\_方法(析构函数)

Python 类体中,可以定义一个特殊的方法: \_\_del\_\_方法。

\_\_del\_\_方法即析构函数(析构方法),用于实现销毁类的实例所需的操作,如释放对象占用的非托管资源(如打开的文件、网络连接等)。

默认情况下,当对象不再被使用时,\_\_del\_\_方法运行,由于 Python 解释器实现自动垃圾回收,即无法保证这个方法究竟在什么时候运行。

通过 del 语句,可强制销毁一个对象实例,从而保证调用对象实例的\_\_del\_\_方法。



### 9.4.4 实例方法、类方法 (@classmethod) 与静态方法 (@staticmethod)

#### 1. 实例方法

一般情况下，类方法的第一个参数一般为 self，这种方法称为实例方法。实例方法对类的某个给定的实例进行操作，可以通过 self 显式地访问该实例。实例方法的声明格式如下：

```
def 方法名(self,[形参列表]):
 函数体
```

实例方法通过对象实例调用：

```
对象.方法名([实参列表])
```

例如：

```
p = Person3() #创建对象
p.say_hi() #调用对象的方法
```

#### 2. 类方法 (@classmethod)

Python 也允许声明属于类本身的方法，即类方法。类方法不对特定实例进行操作，在类方法中访问对象实例属性会导致错误。类方法通过装饰器 @staticmethod 来定义，第一个形式参数必须为类对象本身，通常为 cls。类方法的声明格式如下：

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

类方法一般通过类名来访问，也可通过对象实例来调用。其调用格式如下：

```
类名.类方法名([实参列表])
```

值得注意的是，虽然类方法的第一个参数为 cls，但调用时，用户不需要也不能给该参数传值。事实上，Python 自动把类对象传递给该参数。类对象与类的实例对象不同，在 Python 中，类本身也是对象。调用子类继承父类的类方法时，传入 cls 是子类对象，而非父类对象。

#### 3. 静态方法 (@staticmethod)

Python 也允许声明属于与类的对象实例无关的方法，称为静态方法。静态方法不对特定实例进行操作，在静态方法中访问对象实例会导致错误。静态方法通过装饰器 @staticmethod 来定义，其声明格式如下：

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

静态方法一般通过类名来访问，也可通过对象实例来调用。其调用格式如下：

```
类名.类方法名([实参列表])
```

【例 9-11】类方法示例 (TemperatureConverter.py)。摄氏温度与华氏温度之间的相互转换。

```
class TemperatureConverter:
 @classmethod
 def c2f(cls,t_c): #摄氏温度到华氏温度的转换
```



```

 t_c = float(t_c)
 t_f = (t_c * 9/5) + 32
 return t_f

 @classmethod
 def f2c(cls, t_f): #华氏温度到摄氏温度的转换
 t_f = float(t_f)
 t_c = (t_f - 32) * 5 / 9
 return t_c

#测试代码
if __name__ == '__main__':
 print("1. 从摄氏温度到华氏温度.")
 print("2. 从华氏温度到摄氏温度.")
 choice = int(input("请选择转换方向:"))
 if choice == 1:
 t_c = float(input("请输入摄氏温度:"))
 t_f = TemperatureConverter.c2f(t_c)
 print("华氏温度为: {0:.2f}".format(t_f))
 elif choice == 2:
 t_f = float(input("请输入华氏温度:"))
 t_c = TemperatureConverter.f2c(t_f)
 print("摄氏温度为: {0:.2f}".format(t_c))
 else:
 print("无此选项,只能选择1或2!")

```

运行结果如图 9-1 所示。

```

1. 从摄氏温度到华氏温度.
2. 从华氏温度到摄氏温度.
请选择转换方向: 1
请输入摄氏温度: 30
华氏温度为: 86.00

```

(a) 从摄氏到华氏

```

1. 从摄氏温度到华氏温度.
2. 从华氏温度到摄氏温度.
请选择转换方向: 2
请输入华氏温度: 70
摄氏温度为: 21.11

```

(b) 从华氏到摄氏

图 9-1 类方法示例运行结果

### 9.4.5 私有方法与公有方法

与私有属性类似, Python 约定两个下划线开头, 但不以两个下划线结束的方法是私有的 (Private), 其他为公共的 (Public)。以双下划线开始和结束的方法是 Python 的专有特殊方法。不能直接访问私有方法, 但在其他方法中访问。

**【例 9-12】** 私有方法示例 (BookPrivate.py)。

```

class Book: #定义类 Book
 def __init__(self, name, author, price):
 self.name = name
 self.author = author
 self.price = price

```

```

def __check_name(self): #定义私有方法
 if self.name == '':return False
 else:return True
def get_name(self):
 if self.__check_name():print(self.name,self.author) #调用私有方法
 else:print('No value')
b = Book('Python 程序设计教程','江红',1.0) #创建对象
b.get_name() #调用对象的方法
b.__check_name() #直接调用私有方法,非法

```

运行结果如下:

```

Python 程序设计教程 江红
Traceback (most recent call last):
 File "C:\Python\chapter09\BookPrivate.py", line 15, in <module>
 b.__check_name() #直接调用私有方法,非法
AttributeError: 'Book' object has no attribute '__check_name'

```

#### 9.4.6 方法重载

在其他程序设计语言中,方法可以重载,即可以定义多个重名的方法,只要保证方法签名是唯一的。方法签名包括三个部分:方法名、参数数量和参数类型。

但 Python 本身是动态语言,方法的参数没有声明类型(调用传值时确定参数的类型),参数的数量由可选参数和可变参数来控制(参见 8.2 节)。故 Python 对象方法不需要重载,定义一个方法即可实现多种调用,从而实现相当于其他程序设计语言的重载功能。

**【例 9-13】**方法重载示例 1 (Person21Overload.py)。

```

class Person21: #定义类 Person21
 def say_hi(self,name=None): #定义方法 say_hi
 self.name = name #把参数 name 赋值给 self.name,即成员变量 name(域)
 if name == None:print('您好!')
 else:print('您好,我叫',self.name)
p21 = Person21() #创建对象
p21.say_hi() #调用对象的方法,无参数
p21.say_hi('威尔逊') #调用对象的方法,带参数

```

运行结果如下:

```

您好!
您好,我叫威尔逊

```

在 Python 类体中,可以定义多个重名的方法,虽然不会报错,但只有最后一个方法有效。所以建议不要定义重名的方法。

**【例 9-14】**方法重载示例 2 (Person22Overload.py)。

```

class Person22: #定义类 Person22
 def say_hi(self,name): #定义方法 say_hi
 print('您好,我叫',self.name)
 def say_hi(self,name,age): #定义方法 say_hi

```

```
print('hi,{0},年龄:{1}'.format(name,age))
p22 = Person22() #创建对象
p22.say_hi('Lisa',22) #调用对象的方法
#p22.say_hi('Bob') #TypeError:say_hi() missing 1 required positional argument:'age'
```

运行结果如下：

```
hi,Lisa,年龄:22
```

9.4.7 运算符重载

Python 的运算符实际上是通过调用对象的特殊方法实现的。例如：

```
>>> x = 12;y = 23
>>> x + y #等价于调用 x.__add__(y),结果:35
>>> x.__add__(y) #35
```

在 Python 类体中，通过重写各运算符对应的特殊方法，即可实现运算符的重载。Python 运算符与对应的特殊方法如表 9-3 所示。

表 9-3 运算符与对应的特殊方法

| 运 算 符                | 特 殊 方 法                                                                                      | 含 义           |
|----------------------|----------------------------------------------------------------------------------------------|---------------|
| <, <=, ==, >, >=, != | __lt__, __le__, __eq__, __ge__, __gt__, __ne__                                               | 比较运算符         |
| , ^, &               | __or__, __ror__, __xor__, __rxor__, __and__, __rand__                                        | 按位或、异或、与      |
| =, ^ =, & =          | __ior__, __ixor__, __iand__                                                                  | 按位复合赋值运算      |
| <<, >>               | __lshift__, __rlshift__, __rshift__, __rrshift__                                             | 移位运算          |
| <<=, >>=             | __ilshift__, __irlshift__, __irshift__, __irrshift__                                         | 移位复合赋值运算      |
| +, -                 | __add__, __radd__, __sub__, __rsub__                                                         | 加法与减法         |
| + =, - =             | __iadd__, __isub__                                                                           | 加减复合赋值运算      |
| *, /, %, //          | __mul__, __rmul__, __truediv__, __rtruediv__, __mod__, __rmod__, __floordiv__, __rfloordiv__ | 乘法、除法、取余、整数除法 |
| * =, / =, % =, // =  | __imul__, __idiv__, __itruediv__, __imod__, __ifloordiv__                                    | 乘除复合赋值运算      |
| + x, - x             | __pos__, __neg__                                                                             | 正负号           |
| ~ x                  | __invert__                                                                                   | 按位翻转          |
| ** , ** =            | __pow__, __rpow__, __ipow__                                                                  | 指数运算          |

【例 9-15】运算符重载示例（OpOverload.py）。

```
class MyList: #定义类 MyList
 def __init__(self, * args):
 self.__mylist = [] #初始化私有属性,空列表
 for arg in args:
 self.__mylist.append(arg)
 def __add__(self,n): #重载运算符" + ",每个元素增加 n
 for i in range(0,len(self.__mylist)):
 self.__mylist[i] += n
 def __sub__(self,n): #重载运算符" - ",每个元素减少 n
```

```

 for i in range(0, len(self.__mylist)):
 self.__mylist[i] -= n
 def __mul__(self, n): #重载运算符" * ", 每个元素 * n
 for i in range(0, len(self.__mylist)):
 self.__mylist[i] *= n
 def __truediv__(self, n): #重载运算符"/", 每个元素/n
 for i in range(0, len(self.__mylist)):
 self.__mylist[i] /= n
 def __len__(self): #对应于内置函数 len()
 return len(self.__mylist)
 def __repr__(self): #对应于内置函数 str(), 显示列表
 str1 = ''
 for i in range(0, len(self.__mylist)):
 str1 += str(self.__mylist[i]) + ' '
 return str1
#测试代码
m = MyList(1,2,3,4,5) #创建对象
m + 2; print(repr(m)) #每个元素加 2
m - 1; print(repr(m)) #每个元素减 1
m * 4; print(repr(m)) #每个元素乘 4
m / 2; print(repr(m)) #每个元素除 2
print(len(m)) #列表长度

```

运行结果如下：

```

3 4 5 6 7
2 3 4 5 6
8 12 16 20 24
4.0 6.0 8.0 10.0 12.0
5

```

#### 9.4.8 @functools.total\_ordering 装饰器

支持大小比较的对象需要实现特殊方法：\_\_eq\_\_、\_\_lt\_\_、\_\_le\_\_、\_\_ge\_\_、\_\_gt\_\_。使用functools模块的total\_ordering装饰器装饰类，则只需要实现\_\_eq\_\_，以及\_\_lt\_\_、\_\_le\_\_、\_\_ge\_\_、\_\_gt\_\_中的任意一个。total\_ordering装饰器实现其他比较运算，以简化代码量。

【例9-16】total\_ordering装饰器函数示例（total\_ordering\_student.py）。

```

import functools
@functools.total_ordering
class Student:
 def __init__(self, firstname, lastname): #姓和名
 self.firstname = firstname
 self.lastname = lastname
 def __eq__(self, other): #判断姓名是否一致
 return ((self.lastname.lower(), self.firstname.lower()) ==

```



```

 (other. lastname. lower(),other. firstname. lower()))
def __lt__(self,other):
 #self 姓名 < other 姓名
 return ((self. lastname. lower(),self. firstname. lower()) <
 (other. lastname. lower(),other. firstname. lower()))

#测试代码
if __name__ == '__main__':
 s1 = Student('Mary ', 'Clinton ')
 s2 = Student('Mary ', 'Clinton ')
 s3 = Student('Charlie ', 'Clinton ')
 print(s1 == s2)
 print(s1 > s3)
```

运行结果如下：

```

True
True
```

9.4.9 特殊方法 ( Special Method)

Python 对象中包含许多以双下划线开始和结束的方法，称之为特殊方法。特殊方法通常在针对对象的某种操作时自动调用。例如，创建对象实例时自动调用其\_\_init\_\_方法，a < b 时，自动调用对象 a 的\_\_lt\_\_方法。特殊方法如表 9-4 所示。

表 9-4 Python 特殊方法

| 特殊方法                           | 含 义               |
|--------------------------------|-------------------|
| __lt__、__add__等                | 对应运算符 <、+ 等       |
| __init__、__del__               | 创建或销毁对象时调用        |
| __len__                        | 对应于内置函数 len( )    |
| __setitem__、__getitem__        | 按索引赋值、取值          |
| __repr__( self)                | 对应于内置函数 repr( )   |
| __str__( self)                 | 对应于内置函数 str( )    |
| __bytes__( self)               | 对应于内置函数 bytes( )  |
| __format__( self, format_spec) | 对应于内置函数 format( ) |
| __bool__( self)                | 对应于内置函数 bool( )   |
| __hash__( self)                | 对应于内置函数 hash( )   |
| __dir__( self)                 | 对应于内置函数 dir( )    |

9.4.10 \_\_call\_\_方法和可调用对象 ( callabe)

Python 类体中可以定义一个特殊的方法：\_\_call\_\_方法。定义了\_\_call\_\_方法的对象称为可调用对象 ( Callabe)，即该对象可以像函数一样被调用。

【例 9-17】可调用对象示例 ( CallabeObj. py)。

```

class GDistance:
 #类:自由落体距离
 def __init__(self,g):
 #构造函数
```

```

 self.g = g
 def __call__(self,t):
 return (self.g * t * * 2)/2
#测试代码
if __name__ == '__main__':
 e_gdist = GDistance(9.8)#地球上的重力加速度
 for t in range(11): #0 ~ 10 秒的下落距离
 print(format(e_gdist(t) ,"0. 2f"),end = '') #调用可调用对象 e_gdist

```

运行结果如下：

```
0. 00 4. 90 19. 60 44. 10 78. 40 122. 50 176. 40 240. 10 313. 60 396. 90 490. 00
```

## 9.5 继承

### 9.5.1 派生类

Python 支持多重继承，即一个派生类可以继承多个基类。派生类的声明格式如下：

```
class 派生类名(基类1,[基类2,...]):
 类体
```

其中，派生类名后为所有基类的名称元组。如果在类定义中没有指定基类，则默认其基类为 object。object 是所有对象的根基类，定义了公用方法的默认实现，如\_\_new\_\_()。例如：

```
class Foo:pass
```

等同于：

```
class Foo(object):pass
```

多个类的继承可以形成层次关系，通过类的方法 mro() 或类的属性\_\_mro\_\_可以输出其继承的层次关系。例如：

```

>>> class A:pass
>>> class B(A):pass
>>> class C(B):pass
>>> class D(A):pass
>>> class E(B,D):pass
>>> D.mro()
[<class '__main__. D'> , <class '__main__. A'> , <class 'object'>]
>>> E.__mro__
(<class '__main__. E'> , <class '__main__. B'> , <class '__main__. D'> , <class '__main__. A'> ,
 <class 'object'>)

```

声明派生类时，必须在其构造函数中调用基类的构造函数。调用格式如下：

```
基类名.__init__(self, 参数列表)
```

【例 9-18】派生类示例（DerivedClass.py）。创建基类 Person，包含 2 个数据成员 name 和 age；创建派生类 Student，包含 1 个数据成员 stu\_id。

```

class Person: #基类
 def __init__(self,name,age): #构造函数

```

```

 self.name = name #姓名
 self.age = age #年龄
 def say_hi(self):
 print('您好,我叫{0},{1}岁'.format(self.name,self.age))
class Student(Person): #派生类
 def __init__(self,name,age,stu_id): #构造函数
 Person.__init__(self,name,age)
 self.stu_id = stu_id #学号
 def say_hi(self):
 Person.say_hi(self)
 print('我是学生,我的学号为:',self.stu_id)
p1 = Person('张王一',33) #创建对象
p1.say_hi()
s1 = Student('李姚二',20,'2013101001') #创建对象
s1.say_hi()

```

运行结果如下:

您好,我叫张王一,33 岁

您好,我叫李姚二,20 岁

我是学生,我的学号为: 2013101001

## 9.5.2 类成员的继承和重写

通过继承,派生类继承基类中除构造方法之外的所有成员。如果在派生类中重新定义从基类继承的方法,则派生类中定义的方法覆盖从基类中继承的方法。

**【例 9-19】** 类成员的继承和重写示例 (SubClass.py)。

```

class Dimension: #定义类 Dimensions
 def __init__(self,x,y): #构造函数
 self.x = x #x 坐标
 self.y = y #y 坐标
 def area(self): #覆盖基类的方法 area()
 pass
class Circle(Dimension): #定义类 Circle
 def __init__(self,r): #构造函数
 Dimension.__init__(self,r,0)
 def area(self): #圆面积
 return 3.14 * self.x * self.x
class Rectangle(Dimension): #定义类 Circle
 def __init__(self,w,h): #构造函数
 Dimension.__init__(self,w,h)
 def area(self): #覆盖基类的方法 area()
 return self.x * self.y #矩形面积
d1 = Circle(2.0) #创建对象:圆
d2 = Rectangle(2.0,4.0) #创建对象:矩形

```

```
print(d1.area(),d2.area()) #打印面积
```

运行结果如下：

```
12.56 8.0
```

上例中，派生类 Circle 和 Rectangle 继承了基类的成员变量 x 和 y，重写了继承的方法 area()。

## 9.6 对象的引用、浅拷贝和深拷贝

### 9.6.1 对象的引用

对象的赋值实际上是对象引用，创建一个对象并把它赋值给一个变量时，该变量是指向该对象的引用，其 id() 返回值保持一致。

例如，若银行卡采用列表[户主名,[卡种别,金额]]表示，则：

```
>>> acc10 = ['Charlie',['credit',0.0]] #创建列表对象(信用卡账户),变量 acc10 代表主卡
>>> acc11 = acc10 #变量 acc11 代表副卡,指向 acc10(主卡)的对象
>>> id(acc10),id(acc11) #二者 id 相同:(46594168,46594168)
```

### 9.6.2 对象的浅拷贝

对象的赋值引用同一个对象，即不拷贝对象。如果要拷贝对象，可使用下列方法之一。

- ① 切片操作。例如：acc11[:]
- ② 对象实例化。例如：list(acc11)。
- ③ copy 模块的 copy 函数。例如：copy.copy(acc1)。

例如：

```
>>> import copy
>>> acc1 = ['Charlie',['credit',0.0]]
>>> acc2 = acc1[:] #使用切片方式拷贝对象
>>> acc3 = list(acc1) #使用对象实例化方法拷贝对象
>>> acc4 = copy.copy(acc1) #使用 copy.copy 函数拷贝对象
>>> id(acc1),id(acc2),id(acc3),id(acc4) #拷贝对象 id 各不相同
(46594008,6806144,47411888,47411968)
>>> acc2[0] = 'Mary' #acc2 的第 1 个元素赋值,即户主为'Mary'
>>> acc2[1][1] = -99.9 #acc2 的第 2 个元素的第 2 个元素赋值,即消费金额 99.9
>>> acc1,acc2 #注意,acc2 消费金额改变 99.9,acc1 也随之改变
(['Charlie',['credit',-99.9]],['Mary',['credit',-99.9]])
>>> id(acc1[1]),id(acc2[1]) # acc1[1] 和 acc2[1] 指向同一个对象
(6806144,6806144)
```

例中，acc2[1][1] 赋值 -99.9，acc1[1][1] 也一同改变，因为二者指向同一个对象。Python 拷贝一般是浅拷贝，即拷贝对象时，对象中包含的子对象并不拷贝，而是引用同一个子对象。如果要递归拷贝对象中包含的子对象，请参见 9.6.3 节。



### 9.6.3 对象的深拷贝

如果要递归拷贝对象中包含的子对象，可使用 copy 模块的 deepcopy 函数。例如：

```
>>> import copy
>>> acc1 = ['Charlie', ['credit', 0.0]]
>>> acc5 = copy.deepcopy(acc1) #使用 copy.copy 函数深拷贝对象
>>> acc5[0] = 'Clinton' #acc5 的第 1 个元素赋值,即户主为'Clinton'
>>> acc5[1][1] = -19.9 #acc5 的第 2 个元素的第 2 个元素赋值,即消费金额 19.9
>>> acc1, acc5 #(['Charlie', ['credit', 0.0]], ['Clinton', ['credit', -19.9]])
>>> id(acc1), id(acc5), id(acc1[1]), id(acc5[1])
(47412008, 47398864, 6806144, 47412488)
```

## 9.7 复习题

### 一、填空题

1. 面向对象的程序设计具有三个基本特征：\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。
2. Python 语句 `x = '123'; print(isinstance(x, int))` 的运行结果为\_\_\_\_\_。
3. 创建对象后，可以使用\_\_\_\_\_运算符来调用其成员。
4. Python 类体中，\_\_\_\_\_是一个类方法，创建对象时调用，返回当前对象的一个实例，一般无需重载该方法。\_\_\_\_\_方法即构造函数（构造方法），用于执行类的实例的初始化工作。对象创建后调用，初始化当前对象的实例，无返回值。\_\_\_\_\_方法即析构函数，用于实现销毁类的实例所需的操作，如释放对象占用的非托管资源。

### 二、思考题

1. Python 如何拷贝一个对象？
2. Python 提供哪些特殊属性？如何表示这些特殊属性？各自的含义是什么？
3. 下列 Python 语句的运行结果为\_\_\_\_\_。

```
class parent:
 def __init__(self, param):
 self.v1 = param
class child(parent):
 def __init__(self, param):
 parent.__init__(self, param)
 self.v2 = param
obj = child(100); print("%d %d" % (obj.v1, obj.v2))
```

4. 下列 Python 语句的运行结果为\_\_\_\_\_。

```
class Account:
 def __init__(self, id):
 self.id = id; id = 888
acc = Account(100); print(acc.id)
```

5. 下列 Python 语句的运行结果为\_\_\_\_\_。

```
class account:
 def __init__(self, id, balance):
 self.id = id; self.balance = balance
 def deposit(self, amount): self.balance += amount
 def withdraw(self, amount): self.balance -= amount
acc1 = account('1234', 100); acc1.deposit(500)
acc1.withdraw(200); print(acc1.balance)
```

6. 下列 Python 语句的运行结果为\_\_\_\_\_。

```
class A:
 def __init__(self, a, b, c): self.x = a + b + c
a = A(6, 2, 3); b = getattr(a, 'x'); setattr(a, 'x', b + 1); print(a.x)
```

7. 阅读下面 Python 语句。请问输出结果是什么？

```
d1 = {'a': [1, 2], 'b': 2}; d2 = d1.copy(); d1['a'][0] = 6
sum = d1['a'][0] + d2['a'][0]; print(sum)
```

8. 阅读下面 Python 语句。请问输出结果是什么？

```
from copy import *
d1 = {'a': [1, 2], 'b': 2}; d2 = deepcopy(d1); d1['a'][0] = 6
sum = d1['a'][0] + d2['a'][0]; print(sum)
```

9. 下列 Python 语句的运行结果为\_\_\_\_\_。

```
list1 = [1, 2, 3]; list2 = [3, 4, 5]; dict1 = {'1': list1, '2': list2}; dict2 = dict1.copy()
dict1['1'][0] = 15; print(dict1['1'][0] + dict2['1'][0])
```

10. 下列 Python 语句的运行结果为\_\_\_\_\_。

```
import copy
list1 = [1, 2, 3]; list2 = [3, 4, 5]; dict1 = {'1': list1, '2': list2}
dict2 = copy.deepcopy(dict1); dict1['1'][0] = 15
print(dict1['1'][0] + dict2['1'][0])
```

11. 下列 Python 语句的运行结果为\_\_\_\_\_。

```
class Person:
 def __init__(self, id): self.id = id
mary = Person(123); mary.__dict__['age'] = 18
mary.__dict__['gender'] = 'female'; print(mary.age + len(mary.__dict__))
```

## 9.8 上机实践

1. 参照例 9-1 编写定义类 Person1 的程序。
2. 参照例 9-2 编写定义类 Person2 的程序，定义成员变量（域）name 和 age，以及 say\_hi 成员函数。
3. 参照例 9-3 编写定义类 Person3 的程序，定义类域 count（计数）和类方法 get\_count，返回总计数。
4. 参照例 9-4 编写 property 装饰器示例程序 1。
5. 参照例 9-5 编写 property 装饰器示例程序 2。

6. 参照例 9-6 编写 property 装饰器示例程序 3。
7. 参照例 9-7 编写自定义属性示例程序。
8. 参照例 9-8 编写定义类 Person4 的程序，创建其对象，并调用对象函数。
9. 参照例 9-9 编写\_\_init\_\_方法的示例程序。定义类 Person5，并调用对象的方法。
10. 参照例 9-10 编写\_\_init\_\_方法的示例程序。定义类 Point，表示平面坐标点。
11. 参照例 9-11 编写类方法的示例程序。实现摄氏温度与华氏温度之间的相互转换。
12. 参照例 9-12 编写私有方法的示例程序。
13. 参照例 9-13 编写方法重载的示例程序 1。
14. 参照例 9-14 编写方法重载的示例程序 2。
15. 参照例 9-15 编写运算符重载的示例程序。
16. 参照例 9-16 编写 total\_ordering 装饰器函数的示例程序。
17. 参照例 9-17 编写可调用对象的示例程序。
18. 参照例 9-18 编写派生类的示例程序。创建基类 Person，包含 2 个数据成员 name 和 age；创建派生类 Student，包含 1 个数据成员 stu\_id。
19. 参照例 9.19 编写类成员的继承和重写示例程序。
20. 编写程序，创建类 MyMath，计算圆的周长、面积和球的体积，并编写测试代码，结果均保留 2 位小数。运行效果参见图 9-2 所示。

```
请输入半径： 5
圆的周长 = 31.42
圆的面积 = 78.54
球的体积 = 523.60
```

图 9-2 圆周长面积和球体积运行效果

21. 编写程序，创建类 TemperatureCelsius，包含成员变量（域）degree（表示摄氏温度）和实例方法 ToFahrenheit（将摄氏温度转换为华氏温度），并编写测试代码。运行效果参见图 9-3 所示。

```
请输入摄氏温度： 15
摄氏温度 = 15.0，华氏温度 = 59.0
```

图 9-3 摄氏华氏温度运行效果

# 第 10 章 模块和包

模块对应于 Python 源代码文件。Python 模块中可以定义变量、函数和类。多个功能相似的模块（源文件）可以组织成一个包（文件夹）。通过导入其他模块，可以使用该模块中定义的变量、函数和类，从而重用功能。Python 包含了数量众多的模块，可以实现不同功能和应用。

## 本章要点：

- ◆ 模块的导入和使用；
- ◆ 模块的定义；
- ◆ 包的导入和使用。

## 10.1 模块的导入和使用

Python 包含了数量众多的模块，通过 import 语句，可以导入模块，并使用其定义的功能。

### 10.1.1 import 语句

使用 import 语句可以导入模块。其基本形式如下：

```
import 模块名 #导入模块
import 模块 1,模块 2,...,模块 n #导入多个模块
```

其中模块名是要导入的模块的名称。注：模块名区分大小写。

导入模块后，可以使用全限定名称访问模块中定义的成员：

```
模块名.函数名/变量名 #使用包含模块的全限定名称调用模块中的成员
```

例如：

```
>>> import math
>>> math.pi #3.141592653589793
>>> math.trunc(1.23) #1
```

### 10.1.2 from...import 语句

Python 使用 from...import 语句直接导入模块中的成员。其基本形式如下：

```
from 模块名 import 成员名 #导入模块中的具体成员
成员名 #直接调用
```

如果希望同时导入一个模块中的多个成员，可以采用下列形式：



**from 模块名 import 成员名 1,成员名 2,...,成员名 n**

如果希望同时导入一个模块中的所有成员，则可以采用下列形式：

**from 模块名 import \***

例如：

```
>>> from math import pi, sin
>>> x = sin(2 * pi)
>>> print(str. format('{0:. 2f}', x)) #-0.00
```

注：虽然 from...import 语句可以简化代码，但建议读者避免使用，因为这样可能导致名称冲突（例如，导入多个模块时，多个模块中可能存在同一个名称的函数），且导致程序的可读性差（例如，导入多个模块时，无法准确确定某个名称的函数具体属于哪一个模块）。

### 10.1.3 \_\_import\_\_( ) 内置函数

使用内置函数\_\_import\_\_( )也可导入模块：

**\_\_m = \_\_import\_\_( name)    #导入模块 name 到\_\_m**

内置函数\_\_import\_\_( )具有更大的灵活性，例如要导入的模块 name 可以是计算的结果字符串，但一般不直接使用。事实上，import 语句在内部调用该函数。例如：

```
>>> s = 'os' + '.' + 'path'
>>> __m = __import__(s)
>>> __m. curdir # '.'
```

### 10.1.4 模块搜索路径 sys. path

sys 模块的 sys. path 属性返回一个路径列表。使用 import 语句导入模块时，系统自动从该列表的路径中搜索模块，如果没有找到，则程序报错。

```
>>> import sys
>>> sys. path
['', 'C:\\Python33\\Lib\\idlelib', 'C:\\Windows\\system32\\python33. zip', 'C:\\Python33\\DLLs', 'C:\\Python33\\lib', 'C:\\Python33', 'C:\\Python33\\lib\\site-packages']
```

其中第一个"，表示当前目录；最后一个'C:\\Python33\\lib\\site-packages'用于扩展模块。建议用户自定义模块放置在这两个位置。

程序中也直接修改 sys. path 列表，以添加模块搜索路径。但这种修改只是临时的，即只适用于包含该代码的程序。例如：

```
>>> import sys
>>> sys. path. append('c:\\python\\works')
```

### 10.1.5 dir( ) 内置函数

模块中定义的成员，包括变量、函数和类，可以通过内置的函数 dir( ) 查询，也可以通过 help( ) 函数查询其帮助信息。dir( ) 函数的基本形式如下。

dir( )：不带参数，列举当前模块的所有成员。

dir( 模块名)：列举指定模块的所有成员。

dir( 类/对象)：列举指定类的所有成员。注：Python 所有的成员都是对象。

列举的成员中包含系统定义的特殊意义的成员（\_\_xxx\_\_形式）。例如：

```
>>> dir() #列举当前模块的所有成员
['__builtins__', '__doc__', '__loader__', '__name__', '__package__']
>>> a = 10 #增加变量 a
>>> dir() #列举当前模块的所有成员, 包含 a
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', 'a']
>>> del a #删除变量 a
>>> dir() #列举当前模块的所有成员, 不包含 a
['__builtins__', '__doc__', '__loader__', '__name__', '__package__']
>>> import math
>>> dir(math) #列举 math 模块的所有成员
['__doc__', '__loader__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> dir(10) #列举 10(int 对象)的所有成员
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
>>> dir(str) #列举类的所有成员
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__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', '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']
```

## 10.2 模块的定义

### 10.2.1 创建模块

Python 模块对应于包含 Python 代码的源文件（其扩展名为 .py），在文件中可以定义变量、函数和类。

在模块中，除了可以定义变量、函数和类之外，还可包含一般的语句，称为主块。当运行该模块，或导入该模块时，主块语句将依次执行。

一般而言，独立运行的源代码中主要包含主块，以实现相应的功能。作为库的模块，主要包含供调用的变量、函数和类，还可以包含用于测试的主块代码。

值得注意的是，主块代码语句只在模块第一次被导入时被执行，重复导入时，不会多次导入多次执行。

**【例 10-1】** 创建模块 my\_math1.py，在模块中定义了算术四则运算。

```
PI = 3.14 #定义变量
def add(x,y): #定义函数
 return x + y #加
def sub(x,y):
 return x - y #减
def mul(x,y):
 return x * y #乘
def div(x,y):
 return x / y #除
#测试代码
if __name__ == '__main__': #如果独立运行时,则运行测试代码
 print('123 + 456 = ', add(123,456))
 print('123 - 456 = ', sub(123,456))
 print('123 * 456 = ', mul(123,456))
 print('123 / 456 = ', div(123,456))
```

运行结果如下：

```
123 + 456 = 579
123 - 456 = -333
123 * 456 = 56088
123 / 456 = 0.26973684210526316
```

## 10.2.2 模块的名称

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。例如：

```
>>> import os
>>> os.chdir(r'c:\python\chapter10')
>>> import my_math1
>>> my_math1.__name__ #'my_math1'
```

特别地，当一个模块被用户单独运行时，其\_\_name\_\_的值为 '\_\_main\_\_'，所以可以把模块源代码文件的测试代码写在相应的测试判断中，以保证只有单独运行时，才会运行测试代码。

**【例 10-2】** 创建模块 my\_math2.py。测试代码只有独立运行时才执行。

```
PI = 3.14
def add(x,y):
 return x + y
def sub(x,y):
 return x - y
```

```
def mul(x,y):
 return x * y
def div(x,y):
 return x / y
#测试代码
def main():
 print('123 + 456 = ',add(123,456))
 print('123 - 456 = ',sub(123,456))
 print('123 * 456 = ',mul(123,456))
 print('123 / 456 = ',div(123,456))
if __name__ == '__main__':#如果独立运行时,则运行测试代码
 main()
```

运行结果如下:

```
123 + 456 = 579
123 - 456 = -333
123 * 456 = 56088
123 / 456 = 0.26973684210526316
```

### 10.2.3 按字节编译的 .pyc 文件

导入模块时,python 解释器为加快程序的启动速度,会在与模块文件同一目录下生成 .pyc 文件。

.pyc 文件是经过编译后的字节码,这样下次导入时,如果模块源代码 .py 文件没有修改(通过比较两者的时间戳),则直接导入 .pyc 文件,从而提高程序效率。

按字节编译的 .pyc 文件是在导入模块时,python 解释器自动完成。无需程序员手动编译。

## 10.3 包的定义

### 10.3.1 包的概念

在大型项目中,往往需要创建许多模块,这些功能相似的模块可以使用包组成层次组织结构。

Python 模块是 .py 文件,而包则是文件夹。只要文件夹中包含一个特殊的文件: \_\_init\_\_.py,则 Python 解释器将该文件夹作为包,其中的模块文件(.py 文件)则属于包中的模块。

特殊文件 \_\_init\_\_.py 可以为空,也可以包含属于包的代码,当导入包或该包中的模块时,执行 \_\_init\_\_.py。

包可以包含子包,没有层次限制。

例如,若 C:\Python\chapter10\目录中包含下列目录:

```
package1
 __init__.py
```



```
subPackage1
 __init__.py
 module11.py
 module12.py
 module13.py
subPackage2
 __init__.py
 module21.py
 module22.py
```

则 package1 是顶级包，包含子包 subPackage1 和 subPackage2；包 subPackage1 包含模块 module11、module12 和 module13；包 subPackage2 包含模块 module21 和 module22。

### 10.3.2 创建包

包和模块组成的层次组织结构，对应于文件夹和模块文件。

创建包，首先需要在指定目录中创建对应包名的目录，然后在该目录下创建一个特殊文件 `__init__.py`，最后在该目录下创建模块文件。

### 10.3.3 包的导入和使用

使用 `import` 语句导入包中的模块时，需要指定对应的包名。其基本形式如下：

```
import [包名 1. [包名 2. ...]]. 模块名 #导入包中模块
```

其中包名是模块的上层组织包的名称。注意：包名和模块名区分大小写。

导入包中模块后，可以使用全限定名称访问包中模块定义的成员：

```
[包名 1. [包名 2. ...]]. 模块名. 函数名 #使用全限定名称调用模块中的成员
```

例如：

```
>>> import os.path
>>> os.path.curdir #'.'
>>> os.path.exists(r'c:\python33')#True
```

也可使用 `from ... import` 语句直接导入包中模块的成员。其基本形式如下：

```
from [包名 1. [包名 2. ...]]. 模块名 import 成员名 #导入模块中的具体成员
```

如果希望同时导入一个包中的所有模块，则可以采用下列形式：

```
from 包名 import *
```

同一个包/子包的模块，可以直接导入相同包/子包的模块，而不需要指定包名。这是因为同一个包/子包的模块位于同一个目录。例如，如果包 subPackage2 中包含模块 module21 和 module22，则在模块 module22 中，可通过 `import module21` 直接导入 module21。

## 10.4 命令行参数

### 10.4.1 sys.argv 与命令行参数

在操作系统命令行运行程序时，可以指定若干命令行参数。例如：

```
c:\test.py Para1 Para2
```

在程序中, 可以通过列表 `sys.argv` 访问命令行参数。`argv[0]` 为 Python 脚本名, 如 `c:\test.py`; `argv[1]` 为第 1 个参数名, 如 `Para1`; `argv[2]` 为第 2 个参数名, 如 `Para2`; 以此类推。

【例 10-3】输出命令行参数示例 (`sys_argv.py`)。运行结果如图 10-1 所示。

```
C:\Python\chapter10>sys_argv.py Para1 Para2
sys.argv[0]=C:\Python\chapter10\sys_argv.py
sys.argv[1]=Para1
sys.argv[2]=Para2
```

图 10-1 输出命令行参数

```
import sys
def main(argv):
 for i in range(len(sys.argv)):
 print('sys.argv[{0}] = {1}'.format(i, sys.argv[i]))
if __name__ == '__main__':
 main(sys.argv)
```

## 10.4.2 argparse 和命令行选项参数解析

程序可以根据传入的命令行选项参数执行不同的功能。Python 程序本身也具有若干命令行选项, 如图 10-2 所示。

```
C:\>python -h
usage: python [option] ... [-c cmd | -m mod | file | -l [arg] ...]
Options and arguments (and corresponding environment variables):
-h : issue warnings about str(bytes_instance), str(bytearray_instance)
 and comparing bytes/bytearray with str. (-bb: issue errors)
-B : don't write .pyc[od] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d : debug output from parser; also PYTHONDEBUG=x
-E : ignore PYTHON* environment variables (such as PYTHONPATH)
-h : print this help message and exit (also --help)
```

图 10-2 Python 命令行选项

Python 程序使用 `argparse` 模块实现命令行选项和参数的解析和处理。使用 `argparse` 模块解析命令行选项和参数的步骤为:

### (1) 创建参数解析器 `ArgumentParser` 对象

```
ArgumentParser(prog=None, usage=None, description=None, epilog=None, parents=[],
 formatter_class=argparse.HelpFormatter, prefix_chars='-', fromfile_prefix_chars=None, argument_default=None, conflict_handler='error', add_help=True)
```

其中, `prog` 为程序的名称, 默认为 `sys.argv[0]`; `usage` 为程序的使用方法, 默认根据参数自动生成; `description` 为程序的描述 (在参数之前输出); `epilog` 为程序的描述 (在参数之后输出); `parents` 为父 `ArgumentParser` 对象列表, 如果指定, 则共用父 `ArgumentParser` 对象的参数; `formatter_class` 为格式化帮助信息的对象; `prefix_chars` 为选项的前缀, 默认为 `'-'`; `fromfile_prefix_chars` 为读入额外选项参数的文件; `argument_default` 为参数的默认值; `conflict_handler`: 冲突时的处理函数, 通常不需要; `add_help`: 是否自动添加 `-h/-help` 选项, 默认为 `True`。例如:

```
parser = argparse.ArgumentParser()
```

## (2) 添加参数

使用 `ArgumentParser` 对象 `parser` 的方法 `add_argument` 添加参数：

```
add_argument(name or flags...[, action][, nargs][, const][, default][, type][, choices][, required][, help][, metavar][, dest])
```

各参数的意义如下。

✎ `name` 或 `flags`：指定该参数为名称参数，还是选项参数列表。例如：`foo` 或者 `-f`，`--foo`。

✎ `action`：针对该参数的动作。

✎ `nargs`：对应 `action` 消耗的命令行参数的个数。

✎ `const`：用于某些 `action` 和 `nargs` 的常量值。

✎ `default`：命令行参数没有指定该参数时的默认值。

✎ `type`：命令行参数的类型。

✎ `choices`：参数允许的值的列表。

✎ `required`：命令行选项是否必须。

✎ `help`：参数的简单描述信息。

✎ `metavar`：在使用方法 `usage` 中，该参数的名称。

✎ `dest`：`parse_args()` 方法返回对象中，针对该参数添加的属性。

例如：

```
parser.add_argument(nargs = ' * ', metavar = 'filename ', dest = 'filenames ')
```

`argparse` 是一个完整的参数处理库。参数可以根据 `add_argument()` 的 `action` 选项触发不同动作。支持的 `action` 有存储参数（单个，或作为列表的一部分）；存储常量的值（对布尔开关 `true/false` 有特殊处理）。默认动作是存储参数值。支持 `type`（指定存储类型）和 `dest`（指定存储变量）等参数。

## (3) 解析参数

使用 `ArgumentParser` 对象 `parser` 的方法 `parse_args()` 解析参数：

```
parse_args(args = None, namespace = None)
```

解析参数指定参数，默认为 `None` 时使用 `sys.argv[1:]`，即除程序外的所有命令行参数，返回 1 个对象，所有参数以属性的形式存在。

**【例 10-4】** `argparse` 和命令行选项示例（`argparse1.py`）。执行过程和运行结果如图 10-3 所示。

```
import argparse
def main():
 parser = argparse.ArgumentParser()
 parser.add_argument('-o', dest = 'outfile', help = 'output file')
 parser.add_argument(dest = 'filenames', metavar = 'filename', nargs = ' * ')
 args = parser.parse_args()
 print(args. filenames)
 print(args. outfile)
if __name__ == '__main__':
```



```
main()

C:\Python\chapter10>argparse1.py
[]
None

C:\Python\chapter10>argparse1.py -h
usage: argparse1.py [-h] [-o OUTFILE] [filename [filename ...]]

positional arguments:
 filename

optional arguments:
 -h, --help show this help message and exit
 -o OUTFILE output file

C:\Python\chapter10>argparse1.py -o result.txt f1.txt f2.txt
['f1.txt', 'f2.txt']
result.txt
```

图 10-3 argparse 和命令行选项

## 10.5 终止程序运行时返回消息

如果在某种情况下终止程序，且需要返回特定的消息，可以使用语句：

```
raise SystemExit('返回的消息')
```

**【例 10-5】** 终止程序运行并返回消息示例（sys\_exit.py）。运行结果如图 10-4 所示。

```
import sys
def main(argv):
 if len(sys.argv) != 3:
 sys.stderr.write('参数个数必须为 2')
 raise SystemExit(1)
 print('参数个数正确')
if __name__ == '__main__':
 main(sys.argv)
```

```
C:\Python\chapter10>sys_exit.py
参数个数必须为 2
C:\Python\chapter10>sys_exit.py 1 2
参数个数正确
```

图 10-4 终止程序运行并返回消息

## 10.6 复习题

### 一、填空题

1. Python 包含了数量众多的模块，通过\_\_\_\_\_语句，可以导入模块，并使用其定义的功能。
2. Python 中假设有模块 m，如果希望同时导入 m 中的所有成员，则可以采用\_\_\_\_\_的导入形式。



- 3. Python 中使用内置函数\_\_\_\_\_也可导入模块。
- 4. Python 中 sys 模块的\_\_\_\_\_属性返回一个路径列表。
- 5. Python 中每个模块都有一个名称，通过特殊变量\_\_\_\_\_可以获取模块的名称。特别地，当一个模块被用户单独运行时，模块名称为\_\_\_\_\_。
- 6. 在 Python 程序中，可以通过列表\_\_\_\_\_访问命令行参数。\_\_\_\_\_为 Python 脚本名，\_\_\_\_\_为第 1 个参数名，\_\_\_\_\_为第 2 个参数名。
- 7. Python 程序使用\_\_\_\_\_模块实现命令行选项和参数的解析和处理。
- 8. Python 模块中定义的所有成员，包括变量、函数和类，可以通过内置的函数\_\_\_\_\_查询，也可以通过\_\_\_\_\_函数查询其帮助信息。

二、思考题

- 1. 什么是模块？模块是如何导入解释器的？分别有哪几种方法？
- 2. Python 包和模块是什么关系？包和模块组成的层次组织结构分别对应于什么？
- 3. Python 中创建包的基本步骤和内容是什么？
- 4. Python 中使用 argparse 模块解析命令行选项和参数的主要步骤是什么？
- 5. Python 中如何终止程序运行并返回消息？

10.7 上机实践

- 1. 编写程序，创建一个实现 +、-、\*、/和 \* \*（幂）运算的模块 my\_math1.py，并编写测试代码。运行效果参见图 10-5 所示。
- 2. 编写程序，创建一个求圆的面积和球体体积的模块 area\_volume.py，并编写只有独立运行时才执行的测试代码，要求输入半径，输出结果保留两位小数。运行效果参见图 10-6 所示。

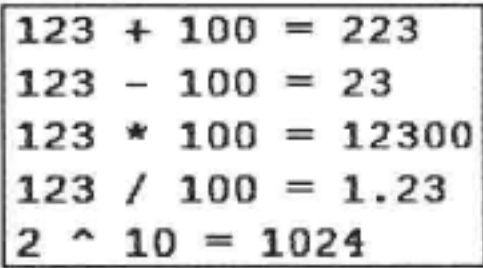


图 10-5 运算运行效果

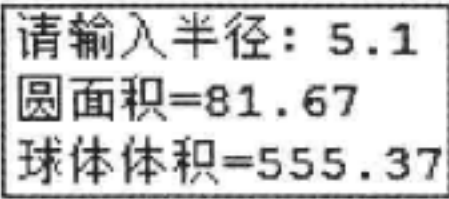


图 10-6 面积体积运行效果

- 3. 编写程序，创建输出命令行参数个数及各参数内容的模块 sys\_argvs.py，并编写测试代码。运行效果参见图 10-7 所示。

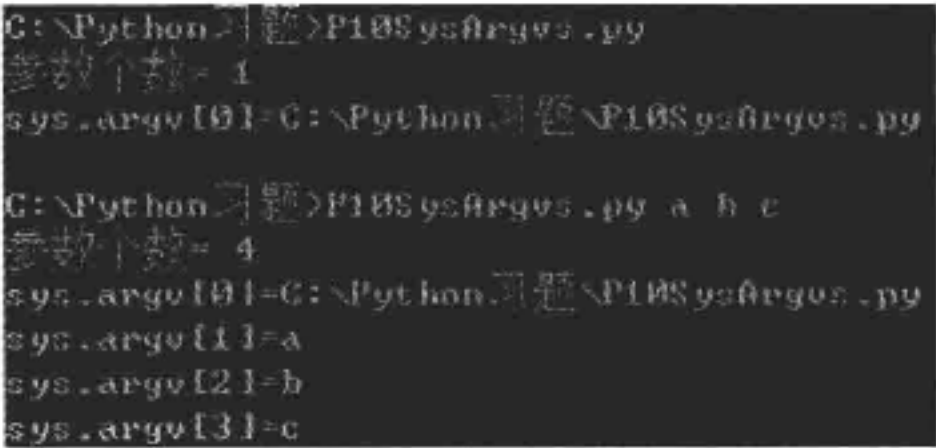


图 10-7 命令行参数运行效果

# 第 11 章 迭代器和生成器

可循环迭代的对象称为可迭代对象，迭代器和生成器函数是可迭代对象，Python 提供了定义迭代器和生成器的协议和方法。

## 本章要点：

- ◆ 可迭代对象、迭代器和可迭代协议；
- ◆ 自定义可迭代对象和迭代器；
- ◆ 生成器函数；
- ◆ 生成器表达式；
- ◆ Python 内置的可迭代对象；
- ◆ itertools 模块和迭代器函数。

## 11.1 迭代和可迭代对象

### 11.1.1 可迭代对象、迭代器和可迭代协议

#### 1. 可迭代对象

在 Python 中，实现了 `__iter__()` 的对象是可迭代对象（Iterable）。`collections.abc` 模块中定义了抽象基类 `Iterable`（参见 12.1 节）。

使用内置函数 `iter(obj)`，可以调用可迭代对象 `obj` 的 `__iter__()` 方法，以返回一个迭代器（iterator）。系列对象都是可迭代对象（参见第 5 章），生成器函数和生成器表达式也是可迭代对象。例如：

```
>>> import collections.abc
>>> isinstance((1,2,3),collections.abc.Iterable) #True
>>> isinstance('python33',collections.abc.Iterable) #True
>>> isinstance(123,collections.abc.Iterable) #False
```

#### 2. 迭代器

实现了 `__next__()` 的对象是迭代器，可使用内置函数 `next()`，调用迭代器的 `__next__()` 方法，依次返回下一个项目值；如果没有新项目时，则导致 `StopIteration`。

`collections.abc` 模块中定义了抽象基类 `Iterator`。使用迭代器可以实现对象的迭代循环，迭代器让程序更加通用、优雅、高效，更加 Python 化。例如，对于大量项目的迭代，使用列表会占用更多的内存，而使用迭代器可以避免之。例如：

```
>>> import collections.abc
>>> il = (i * 2 for i in range(10))
>>> isinstance(il, collections.abc.Iterator) #True
```

### 3. 迭代器协议

迭代器对象必须实现两个方法：`__iter__()`和`__next__()`，二者合称为迭代器协议。`__iter__()`用于返回对象本身，以方便 `for` 语句进行迭代，`__next__()`用于返回下一元素。例如：

```
>>> il = (i * 2 for i in range(10))
>>> help(il)
...(略)
| __iter__(...)
| x.__iter__() <==> iter(x)
| __next__(...)
| x.__next__() <==> next(x)
...(略)
```

## 11.1.2 可迭代对象的迭代：iter 和 next 函数

使用内置函数 `iter(iterable)`，可以返回可迭代对象 `iterable` 的迭代器；使用内置函数 `next()`，可依次返回迭代器对象的下一个项目值；如果没有新项目，则导致 `StopIteration`。例如：

```
>>> t = (1,2) #tuple
>>> i = iter(t) #通过内置函数 iter 获得 iterator
>>> next(i) #通过内置函数 next 获得下一个项目:1
>>> next(i) #2
>>> next(i) #没有新项目时,则导致 StopIteration
```

使用 `while` 循环，也可以循环迭代可迭代对象。

**【例 11-1】** 使用 `while` 循环迭代可迭代对象（`while.py`）。

```
t = (1,2,3,4,5,6,7,8,9,0) #tuple
fetch = iter(t) #获取迭代器
while True:
 try:i = next(fetch)
 except StopIteration:break
 print(i,end = '')
```

运行结果如下：

```
1 2 3 4 5 6 7 8 9 0
```

## 11.1.3 可迭代对象的迭代：for 语句

通常使用 `for` 语句实现可迭代对象的迭代。Python 的 `for` 循环实现了自动迭代可迭代对象的功能。例如：

```
>>> il = (1,2,3,4,5,6,7,8,9,0) #tuple
>>> for i in il:print(i,end = '') #1 2 3 4 5 6 7 8 9 0
>>> i2 = [1,2,3,4,5,6,7,8,9,0] #list
```

```
>>> for i in i2:print(i,end='') #1 2 3 4 5 6 7 8 9 0
>>> i3 = 'python33' #str
>>> for i in i3:print(i,end='') #p y t h o n 3 3
>>> i4 = range(10) #可迭代对象
>>> for i in i4:print(i,end='') #0 1 2 3 4 5 6 7 8 9
```

## 11.2 自定义可迭代对象和迭代器

声明一个类，定义\_\_iter\_\_方法和\_\_next\_\_()。创建该类的对象，即是可迭代对象，也是迭代器。

**【例11-2】**定义类 Fib，实现 Fibonacci 数列（fibonacciIterNext.py）。Fib 对象定义了\_\_iter\_\_方法和\_\_next\_\_()，所以是可迭代对象，也是迭代器。

```
class Fib:
 def __init__(self):
 self.a,self.b=0,1 #前两项值
 def __next__(self):
 self.a,self.b=self.b,self.a+self.b
 return self.a #f(n)=f(n-1)+f(n-2)
 def __iter__(self):
 return self
#测试代码
fibs=Fib()
for f in fibs:
 if f<1000:print(f,end=',')
 else:break
```

运行结果如下：

```
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

## 11.3 生成器函数

在函数定义中，如果使用 yield 语句代替 return 返回一个值，则定义了一个生成器函数（generator）。

生成器函数使用 yield 语句返回一个值，然后保存当前函数整个执行状态，等待下一次调用。生成器函数是一个迭代器，是可迭代对象，支持迭代。例如：

```
>>> def gentripls(n):
 for i in range(n):
 yield i*3
>>> f=gentripls(10)
>>> f # <generator object gentripls at 0x02ED2E40 >
>>> i=iter(f) #通过内置函数 iter 获得 iterator
>>> next(i) #通过内置函数 next 获得下一个项目:0
```



```
>>> next(i) #通过内置函数 next 获得下一个项目:3
>>> for t in f:print(t,end='') #6 9 12 15 18 21 24 27
```

**【例 11-3】** 利用生成器函数创建 Fibonacci 数列 (fibonacciYield.py)。

```
def fib():
 a,b=0,1 #前两项值
 while 1:
 a,b=b,a+b
 yield a #f(n)=f(n-1)+f(n-2)

#测试代码
fibs=fib()
for f in fibs:
 if f<1000:print(f,end=',')
 else:break
```

运行结果如下:

```
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

## 11.4 反向迭代: reversed 迭代器

使用内置函数 reversed(), 可以实现一个系列的反向系列。如果一个可迭代对象实现了 \_\_reversed\_\_() 方法, 则可使用 reversed() 函数获得其反向可迭代对象。

只有长度有限的系列或实现了 \_\_reversed\_\_() 方法的可迭代对象, 才可以使用内置函数 reversed()。例如:

```
>>> reversed([1,2,3,4,5]) # <list_reverseiterator object at 0x02AEE7D0>
>>> for i in reversed([1,2,3,4,5]):print(i,end='') #5 4 3 2 1
```

**【例 11-4】** 可反向迭代的迭代器示例 (reversedCountdown.py)。

```
class Countdown:
 def __init__(self,start):
 self.start=start

 #正向迭代
 def __iter__(self):
 n=self.start
 while n>0:
 yield n
 n-=1

 #反向迭代
 def __reversed__(self):
 n=1
 while n<=self.start:
 yield n
 n+=1
```

#测试代码



```
>>> list(map(abs, (1, -2, 3))) #[1,2,3]
```

```
>>> import operator
```

```
>>> list(map(operator. add, (1,2,3), (1,2,3))) #[2,4,6]
```

如果函数的参数为元组，则需要使用 `itertools.starmap` 迭代器：

```
itertools.starmap(function, iterable) #构造函数
```

例如：

```
>>> import itertools
```

```
>>> list(itertools.starmap(pow, [(2,5), (3,2), (10,3)])) #[32,9,1000]
```

### 11.6.3 filter 迭代器和 itertools.filterfalse 迭代器

Python 3 中的 `filter` 是可迭代对象，可以节省内存空间。

```
filter(function, iterable) #构造函数
```

`filter` 使用指定函数处理可迭代对象的每个元素，函数返回 `bool` 类型的值。若结果为 `True`，则返回该元素。如果 `function` 为 `None`，则返回元素为 `True` 的元素。例如：

```
>>> filter # < class 'filter ' >
```

```
>>> list(filter(lambda x:x>0, (-1,2, -3,0,5))) #[2,5]
```

```
>>> list(filter(None, (1,2,3,0,5))) #[1,2,3,5]
```

如果需要返回结果为 `False` 的元素，则需要使用 `itertools.filterfalse` 迭代器：

```
filterfalse(predicate, iterable) #构造函数
```

`filterfalse` 根据条件函数 `predicate` 处理可迭代对象的每个元素，若结果为 `True`，则丢弃；否则返回该元素。例如：

```
>>> import itertools
```

```
>>> list(itertools.filterfalse(lambda x:x%2, range(10))) #[0,2,4,6,8]
```

### 11.6.4 zip 迭代器和 zip\_longest 迭代器

Python 3 中的 `zip` 是可迭代对象，可以节省内存空间。

```
zip(* iterables) #构造函数
```

`zip` 拼接多个可迭代对象 `iter1`、`iter2`... 的元素，返回新的可迭代对象，其元素为各系列 `iter1`、`iter2`... 对象元素组成的元组。如果各系列 `iter1`、`iter2`... 的长度不一致，则截断至最小系列长度。例如：

```
>>> zip # < class 'zip ' >
```

```
>>> zip((1,2,3), 'abc', range(3)) # < zip object at 0x02A6D210 >
```

```
>>> list(zip((1,2,3), 'abc', range(3))) #[(1,'a',0), (2,'b',1), (3,'c',2)]
```

```
>>> list(zip('abc', range(10))) #[('a',0), ('b',1), ('c',2)]
```

多个可迭代对象的元素个数不一致时，如果需要取最大的长度，则需要使用 `itertools.zip_longest` 迭代器：

```
zip_longest(* iterables, fillvalue = None)
```

其中，`fillvalue` 是填充值，默认为 `None`。例如：

```
>>> import itertools
```

```
>>> list(itertools.zip_longest('ABCD', 'xy', fillvalue = '-')) #[('A','x'), ('B','y'), ('C','-'), ('D','-')]
```

### 11.6.5 enumerate 迭代器

Python 3 中的 `enumerate` 是可选迭代对象，可以节省内存空间。

**`enumerate(iterable, start = 0)`      #构造函数**

`enumerate` 用于枚举可选迭代对象 `iterable` 中的元素，返回元素为元组（计数，元素）的可选迭代对象。计数从 `start` 开始（默认为 0）。例如：

```
>>> enumerate # < class 'enumerate' >
>>> list(enumerate('ABCD', start = 10001))
[(10001, 'A'), (10002, 'B'), (10003, 'C'), (10004, 'D')]
```

【例 11-5】 `enumerate` 迭代器示例（`enumerate_lineno.py`）。打印文本文件的行号和内容。

```
def printfilewithlineno(path):
 with open(path, 'r', encoding = 'utf8') as f:
 lines = f.readlines()
 for idx, line in enumerate(lines):
 print(idx, line)
#测试代码
if __name__ == '__main__':
 thisfile = __file__
 printfilewithlineno(thisfile)
```

运行结果如下：

```
0 def printfilewithlineno(path):
1 with open(path, 'r') as f:
... (略)
```

## 11.7 itertools 模块和迭代器函数

`itertools` 模块包含各种迭代器。这些迭代器非常高效，且内存消耗小。`itertools` 模块中的迭代器既可以单独使用，也可以组合使用。

### 11.7.1 无穷系列迭代器

`itertools` 模块包含 3 个无穷系列的迭代器：

**`count(start = 0, step = 1)`      #从 `start` 开始，步长为 `step` 的无穷系列**  
**`cycle(iterable)`                  #可选迭代对象 `iterable` 元素的无限重复**  
**`repeat(object[, times])`        #重复对象 `object` 无数次（若指定 `times`，则重复 `times` 次）**

例如：

```
>>> from itertools import *
>>> list(zip(count(1), 'abcde')) #[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
>>> list(zip(range(10), cycle('abc')))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'a'), (4, 'b'), (5, 'c'), (6, 'a'), (7, 'b'), (8, 'c'), (9, 'a')]
>>> list(repeat('God', 5)) #['God', 'God', 'God', 'God', 'God']
```



### 11.7.2 累计迭代器 accumulate

itertools 模块的 accumulate 迭代器用于返回累计和：

```
accumulate(iterable[, func])
```

其中，若可迭代对象 iterable 的元素 p0、p1、p2、… 为数值，则结果为：p0，p0 + p1，p0 + p1 + p2，…。如果指定了带两个参数的 func，则 func 代替默认加法运算。例如：

```
>>> import itertools
>>> list(accumulate((1,2,3,4,5))) #结果元素为 1、1+2、1+2+3、…，即：[1,3,6,10,15]
>>> list(accumulate((1,2,3,4,5), operator.mul)) #使用乘法作为运算，结果：[1,2,6,24,120]
```

### 11.7.3 级联迭代器 chain

itertools 模块的 chain 迭代器用于返回级联元素：

```
chain(* iterables) #构造函数
```

连接所有的可迭代对象 iterables1、iterables2、…，即连接多个可迭代对象的元素，作为一个系列。例如：

```
>>> import itertools
>>> list(itertools.chain((1,2,3), 'abc', range(5))) # [1,2,3,'a','b','c',0,1,2,3,4]
chain 的类工厂函数 chain.from_iterable(iterable)，也可用于连接多个系列。例如：
```

```
>>> list(itertools.chain.from_iterable(['ABC', 'DEF'])) # ['A','B','C','D','E','F']
```

### 11.7.4 选择压缩迭代器 compress

itertools 模块的 compress 迭代器用于返回可迭代对象的部分元素：

```
compress(data, selectors) #构造函数
```

根据选择器 selectors 的元素（True/False），返回元素为 True 对应的 data 系列中的元素。当 data 系列或 selectors 终止时，停止判断。例如：

```
>>> import itertools
>>> list(itertools.compress('ABCDEF', [1,0,1,0,1,1])) # ['A','C','E','F']
```

### 11.7.5 截取迭代器 dropwhile 和 takewhile

itertools 模块的 dropwhile 和 takewhile 迭代器用于返回可迭代对象的部分元素：

```
dropwhile(predicate, iterable) #构造函数
```

```
takewhile(predicate, iterable) #构造函数
```

dropwhile 根据条件函数 predicate 处理可迭代对象的每个元素，丢弃 iterable 的元素，直至条件函数的结果为 True；takewhile 则根据条件函数 predicate 处理可迭代对象的每个元素，返回 iterable 的元素，直至条件函数的结果为 False。例如：

```
>>> import itertools
>>> list(itertools.dropwhile(lambda x: x < 5, [1,4,6,4,1])) # [6,4,1]
>>> list(itertools.takewhile(lambda x: x < 5, [1,4,6,4,1])) # [1,4]
```

### 11.7.6 切片迭代器 islice

itertools 模块的 islice 迭代器用于返回可迭代对象的切片：

**islice( iterable, stop)**                      #构造函数

**islice( iterable, start, stop[, step])**      #构造函数

系列支持切片操作，同样，可迭代对象可使用 islice 实现切片功能。islice 返回可迭代对象 iterable 的切片，从索引位置 start（第 1 个元素为 0）开始，到 stop（不包括）结束，步长为 step（默认为 1）。如果 stop 为 None，则直至结束。例如：

```
>>> import itertools
>>> list(itertools.islice('ABCDEFGH', 2)) #['A', 'B']
>>> list(itertools.islice('ABCDEFGH', 2, 4)) #['C', 'D']
>>> list(itertools.islice('ABCDEFGH', 2, None)) #['C', 'D', 'E', 'F', 'G']
>>> list(itertools.islice('ABCDEFGH', 0, None, 2)) #['A', 'C', 'E', 'G']
```

### 11.7.7 迭代器 groupby

itertools 模块的 groupby 迭代器用于返回可迭代对象的分组：

**groupby( iterable, key = None)**      #构造函数

其中，iterable 为待分组的可迭代对象；可选的 key 为用于计算键值的函数，默认为 None，即键值为元素本身值。groupby 返回的结果为迭代器，其元素为 (key, group)，其中 key 是分组的键值，group 为 iterable 中具有相同 key 值的元素的集合的子迭代器。

```
>>> import itertools
>>> data = [1, -2, 0, 0, -1, 2, 1, -1, 2, 0, 0]
>>> data1 = sorted(data, key = abs)
>>> for k, g in itertools.groupby(data1, key = abs):
>>> print(k, list(g))
0 [0, 0, 0, 0]
1 [1, -1, 1, -1]
2 [-2, 2, 2]
```

### 11.7.8 返回多个迭代器 tee

itertools 模块的 tee 迭代器用于返回多个可迭代对象：

**tee( iterable, n = 2)**      #构造函数

返回可迭代对象 iterable 的 n 个（默认为 2）迭代器。例如：

```
>>> import itertools
>>> for i in itertools.tee(range(10), 3): print(list(i))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### 11.7.9 组合迭代器 combinations、combinations\_with\_replacement

itertools 模块的 combinations（元素不重复）和 combinations\_with\_replacement（元素可重

复) 迭代器用于系列的组合:

```
combinations(iterable, r) #构造函数
combinations_with_replacement(iterable, r) #构造函数
```

返回可迭代对象 `iterable` 的元素的组合, 组合长度为 `r`。例如:

```
>>> import itertools
>>> list(itertools.combinations([1,2,3],2)) #[(1,2),(1,3),(2,3)]
>>> list(itertools.combinations([1,2,3,4],2)) #[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
>>> list(itertools.combinations([1,2,3,4],3)) #[(1,2,3),(1,2,4),(1,3,4),(2,3,4)]
>>> list(itertools.combinations_with_replacement([1,2,3],2))
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

### 11.7.10 排列迭代器 `permutations`

`itertools` 模块的 `permutations` 迭代器用于系列的排列:

```
permutations(iterable, r = None) #构造函数
```

返回可迭代对象 `iterable` 的元素的排列, 组合长度为 `r` (默认为系列长度)。例如:

```
>>> import itertools
>>> list(itertools.permutations([1,2,3],2)) #[(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)]
>>> list(itertools.permutations([1,2,3]))
[(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)]
```

### 11.7.11 笛卡儿积迭代器 `product`

`itertools` 模块的 `product` 迭代器用于系列的笛卡儿积:

```
product(* iterables, repeat = 1) #构造函数
```

返回可迭代对象 `iterables1`、`iterables2`、... 的元素的笛卡儿积, `repeat` 为可迭代对象的重复次数 (默认为 1)。例如:

```
>>> import itertools
>>> list(itertools.product([1,2], 'abc')) #[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c')]
>>> list(itertools.product([1,2], repeat = 3))
[(1,1,1),(1,1,2),(1,2,1),(1,2,2),(2,1,1),(2,1,2),(2,2,1),(2,2,2)]
```

## 11.8 复习题

### 一、填空题

1. 使用 Python 内置函数\_\_\_\_\_, 可以调用可迭代对象 `obj` 的\_\_\_\_\_方法, 以返回一个迭代器。
2. Python 迭代器对象必须实现两个方法: \_\_\_\_\_和\_\_\_\_\_, 二者合称为迭代器协议。其中, 前者用于返回对象本身, 以方便 `for` 语句进行迭代, 后者则用于返回下一元素。
3. Python 中使用内置函数\_\_\_\_\_, 可以实现一个系列的反向系列。
4. Python 语句 `for j in (i ** 2 for i in range(10) if i % 3 == 0): print(j, end = '')` 的结果是\_\_\_\_\_。



5. Python 语句 `for i in reversed((1,2,3,4)):print(i,end=',')` 的结果是\_\_\_\_\_。
6. Python 语句 `list(range(0,10,5));list(map(abs,(-1,2,-3)))` 的结果是\_\_\_\_\_。
7. Python 语句 `list(map(operator.add,(1,2),(3,4)))` 的结果是\_\_\_\_\_。
8. Python 语句 `list(itertools.starmap(pow,[(2,6),(3,3),(10,2)]))` 的结果是\_\_\_\_\_。
9. Python 语句 `list(filter(lambda x:x>0,(1,-2,-5,0,6)))` 的结果是\_\_\_\_\_。
10. Python 语句 `list(filter(None,('a',5,3.2,0,(),[],{1,2})))` 的结果是\_\_\_\_\_。
11. Python 语句 `list(itertools.filterfalse(lambda x:x%4,range(10)))` 的结果是\_\_\_\_\_。
12. Python 语句 `list(zip((4,5),'xy',range(3)))` 的结果是\_\_\_\_\_。
13. Python 语句 `list(itertools.zip_longest('AB','abc',fillvalue='*'))` 的结果是\_\_\_\_\_。
14. Python 语句 `list(enumerate('ab',start=101))` 的结果是\_\_\_\_\_。
15. Python 语句 `list(zip('abc',count(1)))` 的结果是\_\_\_\_\_。
16. Python 语句 `list(zip(cycle('ab'),range(5)))` 的结果是\_\_\_\_\_。
17. Python 语句 `list(repeat('Tiger',3))` 的结果是\_\_\_\_\_。
18. Python 语句 `list(accumulate((1,2,3)))` 的结果是\_\_\_\_\_。
19. Python 语句 `list(accumulate((1,2,3),operator.mul))` 的结果是\_\_\_\_\_。
20. Python 语句 `list(itertools.chain((1,2),'xy',range(3)))` 的结果是\_\_\_\_\_。
21. Python 语句 `list(itertools.chain.from_iterable(['ab','123']))` 的结果是\_\_\_\_\_。
22. Python 语句 `list(itertools.compress('xyz',[1,0,1]))` 的结果是\_\_\_\_\_。
23. Python 语句 `list(itertools.dropwhile(lambda x:x<4,[1,2,6,3,7]))` 的结果是\_\_\_\_\_。
24. Python 语句 `list(itertools.takewhile(lambda x:x<4,[1,2,6,3,7]))` 的结果是\_\_\_\_\_。
25. Python 语句 `for i in itertools.tee(range(3),2):print(list(i),end='')` 的结果是\_\_\_\_\_。

## 二、思考题

1. Python 中如何反序迭代一个序列?
2. 简述 Python 中 `range` 可迭代对象的用法。
3. Python 中如何使用列表解析处理可迭代对象,并生成结果列表?
4. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
values = [2,3,1,4]
def my_transformation(num):return num ** 2
for i in map(my_transformation,values):print(i)
```

5. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
values = [1,2,3,2]
def transform(num):return(num ** 2)
for i in map(transform,values):print(i,end='')
```

6. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
values = [1,2,1,2,3];nums = set(values)
```



```
def existcheck(num):
 if num in nums: return True
 else: return False
for i in filter(existcheck, values): print(i, end = '')
```

7. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
fruits1 = ['pear', 'apple', 'kiwi', 'avocado', 'orange']
fruits2 = [fruit.upper() for fruit in fruits1]
print(fruits2[2][0])
```

8. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
def gendoubles(n):
 for i in range(n): yield i * 2
f = gendoubles(5); i = iter(f); next(i); next(i)
for t in f: print(t, end = '')
```

9. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
import itertools; s = 'abcde'
print(list(itertools.islice(s, 2))) ; print(list(itertools.islice(s, 2, 4)))
print(list(itertools.islice(s, 2, None))) ; print(list(itertools.islice(s, 0, None, 2)))
```

10. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
import itertools; data = [-1, 0, -2, 0, 1, 2, -1, 1, 2, 0]
data1 = sorted(data, key = abs)
for k, d in itertools.groupby(data1, key = abs): print(k, list(d))
```

11. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
import itertools
print(list(itertools.combinations(('a', 'b', 'c', 'd'), 2)))
print(list(itertools.combinations(('a', 'b', 'c', 'd'), 3)))
print(list(itertools.combinations_with_replacement(('a', 'b', 'c'), 2)))
```

12. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
import itertools; s = 'ABC'
print(list(itertools.permutations(s, 2)))
print(list(itertools.permutations(s)))
```

13. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
import itertools
print(list(itertools.product('ab', [1, 2, 3])))
print(list(itertools.product([1, 2], repeat = 3)))
```

## 11.9 上机实践

1. 编写程序，使用 while 循环以及内置函数 iter() 和 next() 迭代 1 ~ 10 中的偶数。

2. 编写程序，定义类 Fib，实现 Fibonacci 数列。为 Fib 对象定义 \_\_iter\_\_ 方法和 \_\_next\_\_()。

编写测试代码，显示 Fibonacci 数列：1，1，2，3，5，8，…。当 Fibonacci 值大于 10000 时停止显示。要求每行显示 5 项，每项占 5 个字符的位置，右对齐。运行效果参见图 11-1 所示。

|     |      |      |      |      |
|-----|------|------|------|------|
| 1   | 1    | 2    | 3    | 5    |
| 8   | 13   | 21   | 34   | 55   |
| 89  | 144  | 233  | 377  | 610  |
| 987 | 1597 | 2584 | 4181 | 6765 |

图 11-1 Fibonacci 数列运行效果 (1)

3. 编写程序，利用生成器函数创建 Fibonacci 数列。编写测试代码，显示 Fibonacci 数列：1, 1, 2, 3, 5, 8, ...的前 20 项。要求每行显示 10 项，每项占 5 个字符的位置，右对齐。运行效果参见图 11-2 所示。

|    |     |     |     |     |     |      |      |      |      |
|----|-----|-----|-----|-----|-----|------|------|------|------|
| 1  | 1   | 2   | 3   | 5   | 8   | 13   | 21   | 34   | 55   |
| 89 | 144 | 233 | 377 | 610 | 987 | 1597 | 2584 | 4181 | 6765 |

图 11-2 Fibonacci 数列运行效果 (2)

4. 编写程序，实现可反向迭代的迭代器，分别正向迭代和反向迭代 1 ~ 10 之间的奇数。运行效果参见图 11-3 所示。

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 9 | 7 | 5 | 3 | 1 | 1 | 3 | 5 | 7 | 9 |
|---|---|---|---|---|---|---|---|---|---|

图 11-3 可反向迭代的迭代器运行效果

5. 参照例 11-5 编写 enumerate 迭代器示例程序，打印文本文件的行号和内容。

# 第 12 章 数据结构和算法

Python 的标准库模块提供了若干对象和函数，用于实现各种通用数据结构和算法。

## 本章要点：

- ◆ collections.abc 模块和抽象基类；
- ◆ collections 模块和容器类型；
- ◆ array.array 对象；
- ◆ heapq 模块和堆队列算法；
- ◆ bisect 模块和二分排序算法。

## 12.1 ABC 模块

### 12.1.1 collections.abc 模块和抽象基类

collections.abc 模块包含若干 ABCs（Abstract Base Classes，抽象基类）。抽象基类用于定义接口，继承抽象基类的派生类实现这些接口功能。通过测试一个对象是否属于某个抽象基类，可以判断该对象是否支持对应抽象基类定义的接口功能。

例如，抽象基类 collections.abc.Sized 定义了方法\_\_len\_\_，即支持内置函数 len()：

```
size = None
if isinstance(myvar, collections.abc.Sized):
 size = len(myvar)
```

### 12.1.2 collections.abc 模块中的抽象基类

collections.abc 模块包含的抽象基类如表 12-1 所示。抽象类的派生类需要实现对应的抽象方法；抽象类的派生类自动拥有抽象类包含的继承方法（mixins），不需要重新实现，方便派生类的开发。

表 12-1 collections.abc 模块包含的抽象基类

| 抽象基类      | 父类       | 抽象方法         | 继承方法（mixins） |
|-----------|----------|--------------|--------------|
| Container |          | __contains__ |              |
| Hashable  |          | __hash__     |              |
| Iterable  |          | __iter__     |              |
| Iterator  | Iterable | __next__     | __iter__     |
| Sized     |          | __len__      |              |

续表

| 抽象基类            | 父 类                        | 抽象方法                                                                                                                          | 继承方法 (mixins)                                                                                                                                                                                                                                          |
|-----------------|----------------------------|-------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Callable        |                            | <code>__call__</code>                                                                                                         |                                                                                                                                                                                                                                                        |
| Sequence        | Sized, Iterable, Container | <code>__getitem__</code> , <code>__len__</code>                                                                               | <code>__contains__</code> , <code>__iter__</code> , <code>__reversed__</code> , <code>index</code> , <code>count</code>                                                                                                                                |
| MutableSequence | Sequence                   | <code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__len__</code> , <code>insert</code>   | 继承序列的所有方法, 以及 <code>append</code> , <code>reverse</code> , <code>extend</code> , <code>pop</code> , <code>remove</code> , <code>__iadd__</code>                                                                                                        |
| Set             | Sized, Iterable, Container | <code>__contains__</code> , <code>__iter__</code> , <code>__len__</code>                                                      | <code>__le__</code> , <code>__lt__</code> , <code>__eq__</code> , <code>__ne__</code> , <code>__gt__</code> , <code>__ge__</code> , <code>__and__</code> , <code>__or__</code> , <code>__sub__</code> , <code>__xor__</code> , <code>isdisjoint</code> |
| MutableSet      | Set                        | <code>__contains__</code> , <code>__iter__</code> , <code>__len__</code> , <code>add</code> , <code>discard</code>            | 继承集合的所有方法, 以及 <code>clear</code> , <code>pop</code> , <code>remove</code> , <code>__ior__</code> , <code>__iand__</code> , <code>__ixor__</code> , <code>__isub__</code>                                                                               |
| Mapping         | Sized, Iterable, Container | <code>__getitem__</code> , <code>__iter__</code> , <code>__len__</code>                                                       | <code>__contains__</code> , <code>keys</code> , <code>items</code> , <code>values</code> , <code>get</code> , <code>__eq__</code> , <code>__ne__</code>                                                                                                |
| MutableMapping  | Mapping                    | <code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__iter__</code> , <code>__len__</code> | 继承映射的所有方法, 以及 <code>pop</code> , <code>popitem</code> , <code>clear</code> , <code>update</code> , <code>setdefault</code>                                                                                                                             |
| MappingView     | Sized                      |                                                                                                                               | <code>__len__</code>                                                                                                                                                                                                                                   |
| ItemsView       | MappingView, Set           |                                                                                                                               | <code>__contains__</code> , <code>__iter__</code>                                                                                                                                                                                                      |
| KeysView        | MappingView, Set           |                                                                                                                               | <code>__contains__</code> , <code>__iter__</code>                                                                                                                                                                                                      |
| ValuesView      | MappingView                |                                                                                                                               | <code>__contains__</code> , <code>__iter__</code>                                                                                                                                                                                                      |

【例 12-1】自定义抽象基类的派生类示例 (ListBasedSet.py): 自定义派生于抽象基类 collections.abc.Set 的类 ListBasedSet 侧重于空间效率, 且不要求元素为可 Hash 的元素。

```
import collections.abc
class ListBasedSet(collections.abc.Set):
 def __init__(self, iterable): #生成集合
 self.elements = lst = []
 for value in iterable:
 if value not in lst:
 lst.append(value)
 def __iter__(self): #返回集合元素
 return iter(self.elements)
 def __contains__(self, value): #判断元素是否属于集合
 return value in self.elements
 def __len__(self): #集合元素个数
 return len(self.elements)
s1 = ListBasedSet('abcde')
s2 = ListBasedSet('defghi')
overlap = s1 & s2 #自动支持__and__()方法
for i in overlap: #显示集合交集
 print(i)
```

运行结果如下:

d  
e



## 12.2 collections 模块和容器类型

`collections` 模块和容器类型包含若干实用的容器类型对象，用于实现常用的数据结构。

### 12.2.1 ChainMap 对象

`collections.ChainMap` 对象用于连接多个 `map`：

**ChainMap(\*maps)**

`ChainMap` 对象内部包含多个 `map` 的列表，`maps` 属性返回这个列表。第 1 个 `map` 为子 `map`，其他 `map` 为父 `map`。查询时，首先查询第 1 个 `map`，如果没有查到，则依次查询其他 `map`。`ChainMap` 对象只允许更新第 1 个 `map`。

`ChainMap` 对象 `cmap` 除了支持字典映射的属性和方法外，还包含下列属性和方法。

`cmap.maps`：属性。返回 `cmap` 对象内部包含的 `map` 的列表。

`cmap.parents`：属性。返回包含其父 `map` 的新的 `ChainMap` 对象，即 `ChainMap(*d.maps[1:])`。

`cmap.new_child()`：方法。返回新的 `ChainMap` 对象，即 `ChainMap({}, *d.maps)`。

例如：

```
>>> from collections import *
>>> m1 = {'a':1,'b':2}; m2 = {'a':2,'x':3,'y':4}; m = ChainMap(m1,m2)
>>> m.maps #[{'a':1,'b':2},{ 'x':3,'a':2,'y':4}]
>>> m.parents #ChainMap({'x':3,'a':2,'y':4})
>>> m.new_child() #ChainMap({},{'a':1,'b':2},{ 'x':3,'a':2,'y':4})
>>> m['a'] #查询键'a'的值:1
>>> m['x'] #查询键'x'的值:3
>>> m['a'] = 99 #更新键'a'的值为 99
>>> m['x'] = 10 #更新键'x'的值,因为父 map 不能更新,故实际上在子 map 中插入键/值对
>>> m #ChainMap({'a':99,'x':10,'b':2},{ 'x':3,'a':2,'y':4})
```

**【例 12-2】**`ChainMap` 对象示例 (`ChainMap.py`)。程序中参数值可以为默认值 `map1`、环境变量 `map2`、命令行参数 `map3`，优先顺序为 `map3 > map2 > map1`。使用 `ChainMap (map3, map2, map1)`，可以保证优先使用命令行指定的参数。

```
import os, argparse
from collections import *
defaults = {'color': 'red', 'user': 'guest'}
parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k:v for k,v in vars(namespace).items() if v}
combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

运行结果如下：

```
red
guest
```

## 12.2.2 Counter 对象

`collections.Counter` 对象（计数器）用于统计各元素的计数，结果为 `map`：

**`Counter([iterable – or – mapping])`**

可选参数为系列或字典 `map`。例如：

```
>>> from collections import *
>>> c1 = Counter() #创建空的 Counter 对象
>>> c2 = Counter('banana') #基于系列创建 Counter 对象
>>> c3 = Counter({'red':4,'blue':2}) #基于字典映射创建 Counter 对象
>>> c4 = Counter(cats=4,dogs=8) #基于命名参数创建 Counter 对象
```

`Counter` 对象支持字典映射的属性和方法，但查询时，如果键不存在，不会报错，而是返回值 0。例如：

```
>>> c = Counter({'red':4,'blue':2})
>>> c['green'] #0
```

`Counter` 对象 `c` 还包含下列属性和方法。

`c.elements()`：返回元素列表，各元素重复的次数为其计数。

`c.most_common([n])`：返回计数值最大的 `n` 个元素的元组（元素，计数）列表。

`c.subtract([iterable – or – mapping])`：元素的计数值减去系列或字典中对应元素的计数。

例如：

```
>>> c = Counter({'r':3,'g':2,'b':1,'y':4,'w':3})
>>> c.elements() # < itertools.chain object at 0x02F31FB0 >
>>> list(c.elements()) #['y','y','y','y','r','r','r','g','g','b','w','w','w']
>>> c.most_common(2) # [('y',4),('r',3)]
>>> c.subtract('red');c #Counter({'y':4,'w':3,'r':2,'g':2,'b':1,'d':-1,'e':-1})
```

【例 12-3】`Counter` 对象示例（`counter.py`）：统计文本文件中单词频率，输出最高频率的 5 个单词。

```
import collections,re
path = r'c:\python33\README.txt'
with open(path) as f:
 words = re.findall(r'\w+',f.read()).lower() #读取文本内容,转化为小写
 c = collections.Counter(words) #统计各单词的计数
 print(c.most_common(5)) #最高计数的 5 个单词
```

运行结果如下：

```
[('the',43),('python',34),('to',25),('you',22),('a',22)]
```

## 12.2.3 deque 对象

`collections.deque`（双端队列）支持从任意一端增加和删除元素。`deque` 是线程安全的、

内存高效的队列，它被设计为从两端追加和弹出都非常快。

`deque([iterable[, maxlen]])` #构造函数

其中，可选的 `iterable` 为初始元素，`maxlen` 用于指定队列长度（默认无限制）。

`deque` 对象 `dq` 支持下列方法。

`dq.append(x)`：在右端添加元素 `x`。

`dq.appendleft(x)`：在左端添加元素 `x`。

`dq.pop()`：从右端弹出元素。队列中无元素，则导致 `IndexError`。

`dq.popleft()`：从左端弹出元素。队列中无元素，则导致 `IndexError`。

`dq.extend(iterable)`：在右端添加系列 `iterable` 中的元素。

`dq.extendleft(iterable)`：在左端添加系列 `iterable` 中的元素。

`dq.remove(value)`：移除第一个找到的 `x`。未找到，则导致 `IndexError`。

`dq.count(x)`：返回元素 `x` 在队列中出现的个数。

`dq.clear()`：删除所有元素，即清空队列。

`dq.reverse()`：反转队列中所有元素。

`dq.rotate(n)`：如果 `n > 0`，所有元素向右移动 `n` 个位置（循环）；否则向左。

## 1. deque 作为栈（LIFO）

`deque` 对象方法 `append()` 用于入栈操作；`pop()` 对应于出栈操作。例如：

```
>>> from collections import *
>>> dq = deque()
>>> dq.append(1);dq.append(2);dq.append(3)
>>> dq.pop();dq.pop();dq.pop() #3 2 1
```

## 2. deque 作为队列（FIFO）

`deque` 对象方法 `append()` 用于进队操作；`popleft()` 对应于出队操作。例如：

```
>>> from collections import *
>>> dq = deque()
>>> dq.append(1);dq.append(2);dq.append(3)
>>> dq.popleft();dq.popleft();dq.popleft() #1 2 3
```

【例 12-4】 `deque` 对象示例 1（`deque_tail.py`）：读取文件，返回文件的最后 `n` 行。相当于 UNIX 的 `tail` 命令。

```
import collections
def tail(filename, n = 10):
 'Return the last n lines of a file'
 with open(filename) as f:
 return collections.deque(f, n)
if __name__ == '__main__':
 path = r'deque_tail.py'
 dq = tail(path, n = 2) #最后两行
 print(dq.popleft())
 print(dq.popleft())
```

运行结果如下：

```
print(dq.popleft())
print(dq.popleft())
```

【例 12-5】 deque 对象示例 2 (deque\_moving\_average.py): 移动平均值的计算。

```
import collections, itertools
def moving_average(iterable, n=3):
 # moving_average([40,30,50,46,39,44]) --> 40.0 42.0 45.0 43.0
 # http://en.wikipedia.org/wiki/Moving_average
 it = iter(iterable)
 d = collections.deque(itertools.islice(it, n-1))
 d.appendleft(0)
 s = sum(d)
 for elem in it:
 s += elem - d.popleft()
 d.append(elem)
 yield s / n
if __name__ == '__main__':
 data = [40,30,50,46,39,44]
 list1 = list(moving_average(data, n=3))
 print(list1)
```

运行结果如下:

```
[40.0,42.0,45.0,43.0]
```

## 12.2.4 defaultdict 对象

`collections.defaultdict(function_factory)` 用于构建类似 `dict` 的对象。与 `dict` 的区别是, 创建 `defaultdict` 对象时, 可以使用构造函数参数 `function_factory`, 指定其键/值对中值的类型。

**`defaultdict([default_factory[,...]])` #构造函数**

其中, 可选参数 `function_factory` 为字典键/值对中值的类型; 其他可选参数同 `dict` 构造函数。

`defaultdict` 实现了 `__missing__(key)`, 即键不存在时, 返回值的类型 (`function_factory`) 对应的默认值, 例如数值为 0、字符串为 ''、list 为 [] 等。例如:

|                                                                                                                                                                                                                                  |                                                                                                                                                                                                                                              |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>&gt;&gt;&gt; s = [('r',1),('g',2),('b',3)] &gt;&gt;&gt; dd = defaultdict(int,s) &gt;&gt;&gt; dd defaultdict(&lt;class 'int'&gt;, {'g':2,'r':1,'b':3}) &gt;&gt;&gt; dd['b']    #3 &gt;&gt;&gt; dd['w']    #不存在时,返回默认值0</pre> | <pre>&gt;&gt;&gt; s1 = [('r',1),('g',2),('b',3),('r',4),('b',5)] &gt;&gt;&gt; dd1 = defaultdict(list) &gt;&gt;&gt; for k,v in s1: &gt;&gt;&gt;     dd1[k].append(v) &gt;&gt;&gt; list(dd1.items()) [('g',[2]),('r',[1,4]),('b',[3,5])]</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 12.2.5 OrderedDict 对象

`collections.OrderedDict` 是 `dict` 的子类, 能够记录字典元素插入的顺序。其构造函数为:



**collections.OrderedDict([items])** #构造函数

OrderedDict 对象的元素保持插入的顺序, 更新键的值时, 不改变顺序; 删除项, 然后再插入与删除项相同的键/值对时, 则置于末尾。

除了继承 dict 的方法, OrderedDict 对象包括两个方法:

popitem(last = True): 弹出最后 1 个元素 (即 LIFO), 如果 last = False, 则弹出第 1 个元素 (即 FIFO)。

move\_to\_end(key, last = True): 移动键 key 到最后, 如果 last = False, 则移动到最前面。

注意, 两个 OrderedDict 对象的相等比较运算 (==) 与元素的位置顺序有关。OrderedDict 对象支持反向迭代, 使用 reversed() 反转列表中元素。例如:

```
>>> from collections import *
>>> d = {'banana':3, 'apple':4, 'pear':1, 'orange':2}
>>> d.items() #dict_items([('banana',3), ('pear',1), ('orange',2), ('apple',4)])
>>> sorted(d.items()) #[('apple',4), ('banana',3), ('orange',2), ('pear',1)]
>>> od = OrderedDict(sorted(d.items()))
>>> od #OrderedDict([('apple',4), ('banana',3), ('orange',2), ('pear',1)])
>>> od.popitem() #('pear',1)
```

## 12.2.6 namedtuple 对象

元组 (Tuple) 是常用的数据类型, 但只能通过索引访问其元素, 如果 tuple 中的元素很多时, 操作起来就比较麻烦, 且容易出错。

使用 namedtuple 的构造函数, 可以定义一个 tuple 的子类, 命名元组。namedtuple 对象既可以使用元素名称访问其元素, 也可使用索引访问。

**namedtuple(typename, field\_names, verbose = False, rename = False)** #构造函数

其中, typename 是返回 tuple 的子类的类名; field\_names 是命名元组元素的名称, 必须为合法的标识符; verbose 为 True 时, 创建命名元组后会打印类定义信息; rename 为 True 时, field\_names 中如果包含保留关键字, 则自动命名为 \_1、\_2、... 等。

创建的命名元组的类可通过其 \_source 和 \_fields 返回其代码和字段属性。

somenamedtuple.\_source: 类的代码。

somenamedtuple.\_fields: 类的字段属性。

例如:

```
>>> from collections import *
>>> p = namedtuple('Point', ['x', 'y'])
>>> print(p._source) #打印类的代码
>>> print(p._fields) #打印类的字段属性:('x','y')
>>> p.x = 11; p.y = 22
>>> p.x + p.y #使用元素名称访问命名元组的元素:33
```

namedtuple 创建的类继承于 tuple, 包含三个额外的方法。

somenamedtuple.\_make(iterable): 从指定系列 iterable 构建命名元组对象。

somenamedtuple.\_asdict(): 把命名元组对象转换为 OrderedDict 对象。

somenamedtuple.\_replace(kwargs): 创建新的命名元组对象, 替换指定字段。

例如：

```
>>> t = [3,4]
>>> p1 = p._make(t);p1 #Point(x=3,y=4)
>>> p1._asdict() #OrderedDict([('x',3),('y',4)])
>>> p1._replace(x=30) #Point(x=30,y=4)
```

【例 12-6】namedtuple 对象示例 (namedtuple.py)：读取 csv 格式的文件内容（姓名、年龄、职称、系别和工资），其文件内容为：

```
Mary 45 Professor Chinese 8000
Tom 30 Lecturer Math 5000
Clinton 50 Professor Computer 9000
```

```
from collections import *
import csv
EmployeeRecord = namedtuple('EmployeeRecord','name,age,title,department,paygrade')
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv"))):
 print(emp.name, emp.title)
```

运行结果如下：

```
Mary Professor
Tom Lecturer
Clinton Professor
```

## 12.2.7 UserDict、UserList 和 UserString 对象

collections.UserDict、UserList 和 UserString 分别是 dict、list 和 str 的子类，一般用于创建其派生类。

```
UserDict([initialdata]) #构造函数
UserList([list]) #构造函数
UserString([sequence]) #构造函数
```

虽然可以直接基于 dict、list 和 str 创建派生类，但 UserDict、UserList 和 UserString 包括一个属性成员：data，用于存放内容，因而可以更方便实现。例如：

```
>>> us = UserString('abc')
>>> us.data #'abc'
```

## 12.3 array.array 对象

array 模块包含一个 array 对象，用于实现其他编程语言中的数组数据结构。array 对象是包含相同基本数据类型的列表，其操作与 list 对象基本一致，区别是创建 array 对象时，必须指定其元素类型 typecode，且其元素只能为该类型，否则导致 TypeError。

### 12.3.1 array 的定义

array 模块的 array 对象构造函数为：

```
array(typecode[,initializer])
```

其中，typecode 为 array 对象中数据元素的类型；initializer 为初始化数据系列或可迭代对象，其元素类型必须与 typecode 一致。

array 模块的 array.typecodes 包含 array 对象可用的 typecode，如表 12-2 所示。

表 12-2 array 对象可用的 typecode

| 类型代码 | C 类型               | Python 类型         | 最小字节长度 |
|------|--------------------|-------------------|--------|
| 'b'  | signed char        | int               | 1      |
| 'B'  | unsigned char      | int               | 1      |
| 'u'  | Py_UNICODE         | Unicode character | 2      |
| 'h'  | signed short       | int               | 2      |
| 'H'  | unsigned short     | int               | 2      |
| 'i'  | signed int         | int               | 4      |
| 'I'  | unsigned int       | int               | 4      |
| 'l'  | signed long        | int               | 4      |
| 'L'  | unsigned long      | int               | 4      |
| 'q'  | signed long long   | int               | 8      |
| 'Q'  | unsigned long long | int               | 8      |
| 'f'  | float              | float             | 4      |
| 'd'  | double             | float             | 8      |

例如：

```
>>> import array
>>> array.array('b',(1,2,3,4,5)) #array('b',[1,2,3,4,5])
```

12.3.2 array 对象的基本操作

array 对象支持包括索引访问、切片操作、连接操作、重复操作、成员关系操作、比较运算操作，以及求长度、最大值、最小值等的基本操作。

和 list 对象类似，array 是可变对象，故可以改变其元素的值，也可通过 del 删除某元素；可改变其切片的值，也可通过 del 删除切片。例如：

```
>>> import array
>>> arr1 = array.array('b',(1,2,3,4,5))
>>> arr1[1] #2
>>> arr1[1] = 22;arr1 #array('b',[1,22,3,4,5])
>>> arr1[2:] #array('b',[3,4,5])
>>> del arr1[2:];arr1 #array('b',[1,22])
```

12.3.3 array 对象的方法

array 是可变对象，其包含的主要属性和方法如表 12-3 所示。假设表中的示例基于 a = array.array('b',(1,2,3))。



表 12-3 array 对象的主要属性和方法

| 属性和方法              | 说 明                     | 示 例                                                                     |
|--------------------|-------------------------|-------------------------------------------------------------------------|
| a.typecode         | a 的类型                   | >>> a.typecode #结果:'b'                                                  |
| a.itemsize         | a 的元素字节长度               | >>> a.itemsize #结果:1                                                    |
| a.append(x)        | 把 x 追加到 a 尾部            | >>> a.append(1) #a 为:array('b',[1,2,3,1])                               |
| a.count(x)         | 返回 x 的重复次数              | >>> a.count(1) #结果:2                                                    |
| a.extend(iterable) | 把 iterable 附加到 a 尾部     | >>> a.extend((4,5))<br>#a 为:array('b',[1,2,3,1,4,5])                    |
| a.frombytes(s)     | 把字节系列 s 附加到 a 尾部        | >>> a.frombytes(b'al')<br>#a 为:array('b',[1,2,3,1,4,5,97,49])           |
| a.fromfile(f,n)    | 从文件 f 中读取 n 个字节附加到 a 尾部 |                                                                         |
| a.fromlist(list)   | 把 list 中的元素加到 a 尾部      | >>> a.fromlist([10,11])<br>#a 为:array('b',[1,2,3,1,4,5,97,49,10,11])    |
| a.index(x)         | 返回 x 的索引(第一次出现)         | >>> a.index(1) #结果:0                                                    |
| a.insert(i,x)      | 在下标 i 位置插入 x            | >>> a.insert(0,0)<br>#a 为:array('b',[0,1,2,3,1,4,5,97,49,10,11])        |
| a.pop([i])         | 弹出下标 i 的元素,默认 -1,即最后元素  | >>> a.pop() #结果:11                                                      |
| a.remove(x)        | 删除第一次出现的 x              | >>> a.remove(1)<br>#a 为:array('b',[0,2,3,1,4,5,97,49,10])               |
| a.reverse()        | 反转数组                    | >>> a.reverse()<br>#a 为:array('b',[10,49,97,5,4,1,3,2,0])               |
| a.tobytes()        | 转换为字节系列 bytes           | >>> a = array.array('b',(1,2,3))<br>>>> a.tobytes() #结果:b'\x01\x02\x03' |
| a.tofile(f)        | 写入到文件 f                 |                                                                         |
| a.tolist()         | 转换为 list                | >>> a.tolist() #结果:[1,2,3]                                              |

12.4 heapq 模块和堆队列算法

12.4.1 堆 (Heap) 的基本概念

堆 (Heap) 是一个树形数据结构, 其中子结点与父结点是一种有序关系。二叉堆 (Binary Heap) 可以使用以如下方式组织的列表或数组表示, 即元素 N 的子元素位于  $2 * N + 1$  和  $2 * N + 2$  (索引从 0 开始)。这种布局允许原地重新组织堆, 从而不必在增加或删除元素时分配大量内存。

最大堆 (max - heap) 确保父结点大于或等于其两个子结点。最小堆 (min - heap) 要求父结点小于或等于其子结点。

Python 的 heapq 模块实现了一个最小堆。即对于任意索引 k, 满足  $a[k] \leq a[2 * k + 1]$  并且  $a[k] \leq a[2 * k + 2]$ , 如图 12-1 所示。

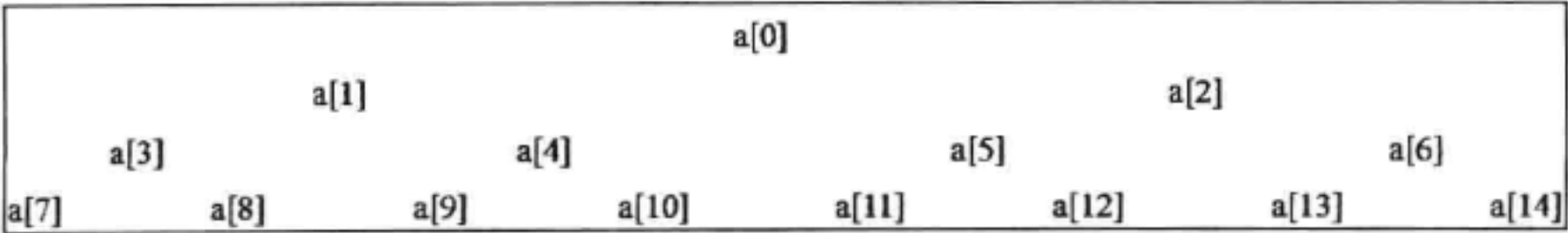


图 12-1 最小堆示意图



## 12.4.2 堆的基本操作

Python 堆通常使用列表存储其数据，即满足堆的结构的 list 对象就是 heap。创建堆的方法如下。

`heap = []`：使用空列表作为堆。

`heapq.heapify(alist)`：将列表转换为堆，该函数可以自动完成数据的重新排序。

创建堆后，可以使用 `heapq` 模块中的函数对其进行操作：

`heapq.heappush(heap, item)`      #把 item 推入堆 heap

`heapq.heappop(heap)`              #从堆 heap 中弹出最小值

`heapq.heappushpop(heap, item)` #推入 item 并弹出最小值,比两步操作效率更高

`heapq.heapreplace(heap, item)` #弹出最小值并推入 item,比两步操作效率更高

堆操作自动完成其数据的重新排序，以保证数据满足堆的结构。例如：

```
>>> heap1 = []
>>> heappush(heap1,3);heappush(heap1,2);heappush(heap1,4);heappush(heap1,1)
>>> heappop(heap1),heappop(heap1),heappop(heap1),heappop(heap1) #(1,2,3,4)
```

【例 12-7】基于 heap 的排序 (`heapsort.py`)。把数据推入到 heap，然后逐一弹出。

```
from heapq import *
def heapsort(iterable):
 h = []
 for value in iterable:
 heappush(h, value)
 return [heappop(h) for i in range(len(h))]
#测试代码
if __name__ == '__main__':
 data = [1,3,5,7,9,2,4,6,8,0]
 print(heapsort(data))
```

运行结果如下：

```
[0,1,2,3,4,5,6,7,8,9]
```

## 12.4.3 heapq 模块中基于 heap 的通用函数

`heapq` 模块中包含若干基于 heap 的通用函数：

`heapq.merge(*iterables)`              #合并多个可迭代对象,返回包含其排序后元素的可迭代对象

`heapq.nlargest(n, iterable, key=None)`      #返回可迭代对象中的最大 n 个值

`heapq.nsmallest(n, iterable, key=None)`      #返回可迭代对象中的最小 n 个值

其中，`iterable` 为可迭代对象，`key` 为比较函数。`merge()` 函数常用于多个可迭代对象的合并与排序，`nlargest()` 和 `nsmallest()` 函数常用于获取最大和最小数据部分。例如：

```
>>> import heapq
>>> for i in heapq.merge([1,3,5,8],[2,4,7],[0,6,9]):
 print(i, end = ' ')
#0 1 2 3 4 5 6 7 8 9
>>> heapq.nlargest(3,[1,3,5,7,9,2,4,6,8,0])
#[9,8,7]
>>> heapq.nsmallest(3,[1,3,5,7,9,2,4,6,8,0])
#[0,1,2]
```

## 12.5 bisect 模块和二分排序算法

bisect 模块提供了基于二分排序算法的函数，用于维护有序列表的顺序：

```
bisect.bisect_left(a, x, lo=0, hi=len(a)) #查找有序列表 a 中 x 的插入位置索引
bisect.bisect_right(a, x, lo=0, hi=len(a))
bisect.bisect(a, x, lo=0, hi=len(a))
bisect.insort_left(a, x, lo=0, hi=len(a)) #在有序列表 a 中插入 x, 保持 a 的有序性
bisect.insort_right(a, x, lo=0, hi=len(a))
bisect.insort(a, x, lo=0, hi=len(a))
```

其中，a 为有序列表；x 为要插入的值；[lo, hi) 是插入的查找范围，默认为整个列表。如果 x 已经在列表 a 中存在，则 bisect\_left 返回既存 x 之前的位置（左边），insort\_left 在既存的 x 之前（左边）插入；bisect\_right 和 bisect 返回既存 x 之后的位置（右边），insort\_right 和 insort 在既存的 x 之后（右边）插入。

insort\_left 等价于 `a.insert(bisect.bisect_left(a, x, lo, hi), x)`；insort\_right 和 insort 等价于 `a.insert(bisect.bisect_right(a, x, lo, hi), x)`。例如：

```
>>> import bisect
>>> list1 = [1, 3, 5, 7, 9]; bisect.bisect(list1, 2) #1
>>> bisect.insort(list1, 2); list1 #[1, 2, 3, 5, 7, 9]
>>> bisect.bisect(list1, 2) #2
>>> bisect.insort(list1, 2); list1 #[1, 2, 2, 3, 5, 7, 9]
```

【例 12-8】基于 bisect 的成绩等级 (bisect\_grade.py)。

```
import bisect
def grade(score, breakpoints = [60, 70, 80, 90], grades = 'FDCBA'):
 i = bisect.bisect(breakpoints, score)
 return grades[i]
#测试代码
if __name__ == '__main__':
 data = [33, 65, 77, 70, 89, 90, 100]
 for s in data: print(s, grade(s), end = ',')
```

运行结果如下：

```
33 F, 65 D, 77 C, 70 C, 89 B, 90 A, 100 A,
```

## 12.6 复习题

1. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
from collections import *
m1 = {1: 'a', 2: 'b'}; m2 = {2: 'a', 3: 'x', 4: 'y'}; m = ChainMap(m1, m2)
print(m.maps, m.parents, m.new_child())
print(m[1], m[3]); m[1] = 'A'; m[3] = 'X'; print(m)
```

2. 下列 Python 语句的执行结果是\_\_\_\_\_。

```

from collections import *
c1 = Counter(); print(c1); c2 = Counter('banana'); print(c2)
c3 = Counter({'R':4, 'B':2}); print(c3)
c4 = Counter(birds=2, cats=4, dogs=8); print(c4)
print(c4['flowers'], c4['cats']); print(list(c3.elements()))
print(c4.most_common(2)); c3.subtract('RGB'); print(c3)

```

3. 下列 Python 语句的执行结果是\_\_\_\_\_。

```

from collections import *
dq = deque(); dq.append('a'); dq.append(2); dq.append('c'); data = iter(dq)
while True:
 try: i = next(data)
 except StopIteration: break
 print(i, end = '')
print(dq.pop(), dq.pop(), dq.pop())

```

4. 下列 Python 语句的执行结果是\_\_\_\_\_。

```

from collections import *
dq = deque(); dq.append('a'); dq.append(2); dq.append('c')
print(dq.popleft(), dq.popleft(), dq.popleft())

```

5. 下列 Python 语句的执行结果是\_\_\_\_\_。

```

from collections import defaultdict
s = [('r',3), ('g',2), ('b',1)]; dd = defaultdict(int, s); print(dd['b'], dd['w'])
s1 = [('r',3), ('g',2), ('b',1), ('r',5), ('b',4)]; dd1 = defaultdict(list)
for k,v in s1: dd1[k].append(v)
print(list(dd1.items()))

```

6. 下列 Python 语句的执行结果是\_\_\_\_\_。

```

from collections import *
d = {'red':3, 'green':4, 'blue':1}; print(d.items(), sorted(d.items()))
od = OrderedDict(sorted(d.items())); print(od.popitem(), od.popitem(False))

```

7. 下列 Python 语句的执行结果是\_\_\_\_\_。

```

from collections import *
p = namedtuple('Point', ['x', 'y']); p.x = 1; p.y = 2; print(p._fields, p.x, p.y)
t = [10, 20]; p1 = p._make(t); print(p1._asdict())
print(p1._replace(x=100), p1.x, p1.y)

```

8. 下列 Python 语句的执行结果是\_\_\_\_\_。

```

import array; arr1 = array.array('i', (1,2,3,4,5))
arr1[1] = 22; print(arr1, arr1[2:], type(arr1[1]))
del arr1[2:]; print(arr1, arr1.typecode, arr1.itemsize)

```

9. 下列 Python 语句的执行结果是\_\_\_\_\_。

```

import array; a = array.array('b', (3,2)); a.append(3); a.extend((3,5))
print(a, a.count(3)); a.frombytes(b'A1'); a.fromlist([8,9])
print(a, a.index(3)); a.insert(0,1); a.pop()

```

```
a.remove(2);a.reverse();print(a.tolist())
```

10. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
from heapq import *;heap1 = []
heappush(heap1,3);heappush(heap1,2);heappush(heap1,4);heappush(heap1,1)
for i in range(len(heap1)):print(heappop(heap1),end='')
```

11. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
import heapq;h = heapq.merge([6,9],[1,2,8],[4,5])
for i in h:print(i,end='')
print(heapq.nlargest(2,[4,3,7,5,9,2,1,6,8,0]))
print(heapq.nsmallest(2,[4,3,7,5,9,2,1,6,8,0]))
```

12. 下列 Python 语句的执行结果是\_\_\_\_\_。

```
import bisect;list1 = [2,4,6];print(bisect.bisect(list1,3))
bisect.insort(list1,3);print(list1)
print(bisect.bisect(list1,3));bisect.insort(list1,3);print(list1)
```

## 12.7 上机实践

1. 参照例 12-1 实现自定义抽象基类的派生类示例程序。
2. 参照例 12-2 实现 ChainMap 对象示例程序。
3. 参照例 12-3 实现 Counter 对象示例程序，统计指定文本文件中的单词频率，输出最高频率的 5 个单词。
4. 参照例 12-4 实现 deque 对象示例程序，读取指定文件的内容，返回文件的最后 3 行。
5. 参照例 12-5 实现 deque 对象示例程序，计算[95,90,65,86,99,84]移动平均值。
6. 参照例 12-6 实现 namedtuple 对象示例程序，读取成绩文件 scores.csv 的内容（学员 ID、语文、数学、外语和信息，内容如图 12-2（a）所示），显示学员 ID 和平均成绩。运行效果如图 12-2（b）所示。

|         |    |    |    |    |
|---------|----|----|----|----|
| 2014111 | 97 | 92 | 81 | 60 |
| 2014112 | 75 | 84 | 91 | 39 |
| 2014113 | 88 | 94 | 65 | 91 |
| 2014114 | 97 | 89 | 85 | 82 |
| 2014115 | 35 | 72 | 91 | 70 |
| 2014116 | 99 | 86 | 90 | 94 |

（a）文件 scores.csv 的内容

| 学号      | 平均成绩  |
|---------|-------|
| 2014111 | 82.5  |
| 2014112 | 72.25 |
| 2014113 | 84.5  |
| 2014114 | 88.25 |
| 2014115 | 67.0  |
| 2014116 | 92.25 |

（b）显示学员 ID 和平均成绩

图 12-2 学生信息运行效果

7. 参照例 12-7 实现基于 heap 的排序程序。把 1 ~ 10 中的偶数推入到 heap，然后逐一弹出。
8. 参照例 12-8 实现基于 bisect 的成绩等级。测试数据为[99,65,87,50,90,100]，成绩等级为 A、B、C、D 和 F。



# 第 13 章 日期和时间处理

Python 提供了丰富的数据类型和库函数，以用于数值和日期处理。

## 本章要点：

- ◆ 日期和时间处理概述；
- ◆ time 模块和时间处理函数；
- ◆ datetime 模块和日期时间处理；
- ◆ calendar 模块和日历处理。

## 13.1 日期和时间处理概述

### 13.1.1 相关模块

datetime 模块包含各种用于日期和时间处理的类。calendar 模块包含用于用于处理日历的函数和类；time 模块包含用于处理时间的函数。

### 13.1.2 相关术语

#### 1. epoch（新纪元）

系统规定的时间起始点。UNIX 系统是 1970/1/1 0:0:0 开始。日期和时间在内部表示为从 epoch 开始的秒数。time 模块中的函数使用对应 C 语言函数库中的函数，故只能处理 1970/1/1 至 2038/12/31 之间的日期和时间。

使用 time 模块的 gettime 函数可以获取当前系统的起始点。例如：

```
>>> import time; time.gmtime(0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=3, tm_yday=1, tm_isdst=0)
```

#### 2. UTC

UTC 是协调世界时（Coordinated Universal Time）的英文缩写，是一种兼顾理论与应用的时标。旧称 GMT（Greenwich Mean Time，格林威治时间）。

#### 3. DST

DST（Daylight Saving Time）即夏令时。不同地域可能规定不同的夏令时，C 语言函数库使用表格对应这些规定。time 模块中的 daylight 属性用于判定是否使用夏令时。

```
>>> time.daylight #0
```

## 13.2 time 模块

### 13.2.1 struct\_time 对象

time 模块的函数 `gmtime()`、`localtime()` 和 `strptime()`，返回 `struct_time` 对象；`asctime()`、`mktime()` 和 `strftime()` 均使用 `struct_time` 对象作为参数。`struct_time` 对象是一个命名元组 (named tuple)，包含表 13-1 所示的 9 个整数类型字段属性。

表 13-1 struct\_time 对象包含的字段属性

| 索 引 | 属 性       | 含义和取值范围                      |
|-----|-----------|------------------------------|
| 0   | tm_year   | 年。(例如：1993)                  |
| 1   | tm_mon    | 月。取值范围[1,12]                 |
| 2   | tm_mday   | 日。取值范围[1,31]                 |
| 3   | tm_hour   | 时。取值范围[0,23]                 |
| 4   | tm_min    | 分。取值范围[0,59]                 |
| 5   | tm_sec    | 秒。取值范围[0,61]                 |
| 6   | tm_wday   | 星期。取值范围[0,6]。Monday 为 0      |
| 7   | tm_yday   | 当前的天数。取值范围[1,366]            |
| 8   | tm_isdst  | 是否为 DST。0、1 或 -1。-1 时，系统自动确定 |
| N/A | tm_zone   | 时区的缩写                        |
| N/A | tm_gmtoff | 相对于 UTC 的时差（单位为秒）            |

```
>>> st = time.gmtime();st
time.struct_time(tm_year=2013,tm_mon=9,tm_mday=25,tm_hour=7,tm_min=25,tm_sec=55,tm_wday=2,tm_yday=268,tm_isdst=0)
>>> st.n_fields #9
>>> st[0] #2013
>>> st.tm_year #2013
```

### 13.2.2 time 模块中的常用函数

time 模块中包含的主要函数如表 13-2 所示。假设表中示例基于 `import time`。

表 13-2 time 模块中包含的主要函数/属性

| 函 数           | 说 明                  | 示 例                                               |
|---------------|----------------------|---------------------------------------------------|
| time.timezone | 时区。相对于 UTC 的时差（单位为秒） | >>> time.timezone<br>-28800                       |
| time.tzname   | 时区名                  | >>> time.tzname<br>(‘ÖD’ú±ê×¼Ê±¼ä’, ‘ÖD’úÏÄÁiÊ±’) |
| time.time()   | 获取当前时间（单位为秒）         | >>> time.time()<br>1380094669.10948               |

续表

| 函 数                                 | 说 明                                                     | 示 例                                                 |
|-------------------------------------|---------------------------------------------------------|-----------------------------------------------------|
| time.ctime([secs])                  | 把秒数转换为日期时间字符串。默认当前时间。<br>等同于 asctime (localtime (secs)) | >>> time.ctime()<br>'Wed Sep 25 16: 02: 39 2013'    |
| time.gmtime([secs])                 | 把秒数转换为 struct_time 对象 (UTC)。默认当前时间                      | >>> time.gmtime()                                   |
| time.localtime([secs])              | 把秒数转换为 struct_time 对象 (本地)。默认当前时间                       | >>> time.localtime()                                |
| time.strptime ( string [, format] ) | 把字符串转换为 struct_time 对象 (本地)                             | 参见 13. 2. 3 节                                       |
| time.asctime([t])                   | 把 struct_time 对象转换为日期时间字符串。默认当前时间                       | >>> time.asctime()<br>'Wed Sep 25 16: 02: 23 2013'  |
| time.mktime(t)                      | 把 struct_time 对象转换为本地时间 (秒)                             | >>> time.mktime (time.localtime())<br>1380096644. 0 |
| time.strftime (format[,t])          | 把 struct_time 对象转换为字符串。默认当前时间                           | 参见 13. 2. 3 节                                       |
| time.sleep (secs)                   | 当前线程睡眠指定时间 (单位为秒)                                       | >>> time.sleep(10)                                  |
| time.process_time()                 | 返回当前进程处理器运行时间。一般使用两处的差值计算程序花费的时间                        | >>> time.process_time()<br>0. 2028013               |
| time.perf_counter()                 | 返回性能计数器。一般使用两处的差值计算程序花费的时间                              | >>> time.perf_counter()<br>9. 621530676486614e - 07 |
| time.monotonic()                    | 返回单向时钟。一般使用两处的差值计算程序花费的时间                               | >>> time.monotonic()<br>191429. 809                 |

【例 13-1】 测量程序运行时间 (time\_pmrunning. py)。

```
import time
def test():
 sum = 0
 for i in range(0,9999999):
 sum += i
 return sum
if __name__ == '__main__':
 t1 = time.monotonic() #单向时钟
 print(test())
 t2 = time.monotonic() #单向时钟
 print('运行时间:',t2 - t1)
```

运行结果如下：

49999985000001  
运行时间：1. 0139999999955762

13. 2. 3 日期格式化字符串

time 模块中的 strptime() 将字符串解析为 struct\_time 对象；strftime() 将 struct\_time 对象

格式化为字符串。

```
time.strptime(string[,format])
time.strftime(format[,t])
```

其中，string 为日期字符串；t 为 struct\_time 对象；format 为日期格式化字符串。常见的日期格式化字符串如表 13-3 所示。

表 13-3 日期格式化字符串

| 符 号 | 说 明                | 符 号 | 说 明                                        |
|-----|--------------------|-----|--------------------------------------------|
| % a | 本地化星期名（略称）         | % S | 秒。[00,61]                                  |
| % A | 本地化星期名（全称）         | % U | 年中的第几个星期（Sunday 作为星期的第 1 天）。[00,53]        |
| % b | 本地化月份名（略称）         | % w | 星期。[0(Sunday),6]                           |
| % B | 本地化月份名（全称）         | % W | 年中的第几个星期（Monday 作为星期的第 1 天）。[00,53]        |
| % c | 本地化的日期和时间表示        | % x | 本地化的日期表示                                   |
| % d | 天。[01,31]          | % X | 本地化的时间表示                                   |
| % H | 小时（24 小时制）。[00,23] | % y | 年（2 位数）。[00,99]                            |
| % I | 小时（12 小时制）。[01,12] | % Y | 年（4 位数）                                    |
| % j | 年中的第几天。[001,366]   | % % | 字符 %'                                      |
| % m | 月。[01,12]          | % Z | 时区名                                        |
| % M | 分。[00,59]          | % z | 时区时差（+ HHMM 或 - HHMM）。<br>[-23:59, +23:59] |
| % p | AM/PM 的本地化表示       |     |                                            |

例如：

```
>>> from time import *
>>> strftime("% c", localtime()) #01/05/14 18:50:40'
>>> strftime("% Y 年% m 月% d 日(% A) % H 时% M 分% S 秒", localtime())
'2014 年 01 月 05 日(Sunday) 18 时 50 分 51 秒'
>>>.strptime("30 Nov 00", "% d % b % y")
time.struct_time(tm_year = 2000, tm_mon = 11, tm_mday = 30, tm_hour = 0, tm_min = 0, tm_sec = 0, tm_wday = 3, tm_yday = 335, tm_isdst = -1)
```

13.3 datetime 模块

datetime 模块包括两个常量：datetime.MINYEAR 和 datetime.MAXYEAR，表示最小年份和最大年份，分别为 1 和 9999。

datetime 模块包含用于表示日期的 date 对象、表示时间的 time 对象和表示日期时间的 datetime 对象。timedelta 对象表示日期或时间之间的差值，可用于日期或时间的运算。tzinfo 对象和 timezone 对象表示时区信息，创建 time 和 datetime 对象时，可指定时区信息。datetime 模块的类的继承关系如图 13-1 所示。

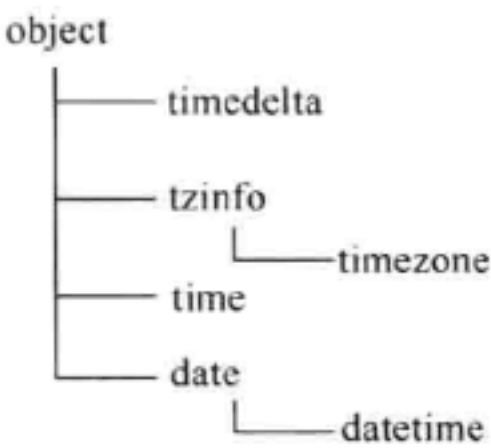


图 13-1 datetime 模块的类的继承关系



13.3.1 date 对象

date 对象表示由年、月、日组成的简单日期。

1. 创建 date 对象

date 对象的构造函数为：

```
date(year, month, day) #指定年月日,构造 date 对象
```

其中，MINYEAR <= year <= MAXYEAR，1 <= month <= 12，1 <= day <= 该月的最大天数。

2. 通过 date 类工厂函数获取 date 对象

date.today()：通过类方法，获得 date 对象（当天）。

date.fromtimestamp(timestamp)：获得 date 对象（当前时间戳）。

date.fromordinal(ordinal)：获得 date 对象（指定从 1/1/1 开始的天数）。

例如：

```
>>> from datetime import *;import time
>>> date(2013,9,26) #datetime.date(2013,9,26)
>>> date.fromtimestamp(time.time()) #datetime.date(2013,9,26)
>>> date.fromordinal(32) #datetime.date(1,2,1)
```

3. date 对象的方法

date 对象包含的主要方法如表 13-4 所示。假设表中示例基于：

```
>>> import datetime;d = datetime.date(2013,9,26)
```

表 13-4 date 的主要方法和属性

| 方法和属性                       | 说 明                           | 示 例                                                   |
|-----------------------------|-------------------------------|-------------------------------------------------------|
| date.min                    | 类属性。返回最小、最大日期                 | >>> datetime.date.min<br>datetime.date(1,1,1)         |
| date.max                    | 类属性。返回最大日期                    | >>> datetime.date.max<br>datetime.date(9999,12,31)    |
| date.resolution             | 类属性。日期精度，即 2 个日期之间的最小差（1 天）   | >>> datetime.date.resolution<br>datetime.timedelta(1) |
| d.year、d.month、d.day        | 属性。返回年、月、日                    | >>> d.year, d.month, d.day<br>(2013,9,26)             |
| d.replace(year, month, day) | 方法。替换年、月、日                    | >>> d.replace(month=10)<br>datetime.date(2013,10,26)  |
| d.timetuple()               | 方法。返回 struct_time 对象          | >>> d.timetuple()                                     |
| d.toordinal()               | 方法。返回从 1/1/1 开始的天数            | >>> d.toordinal()<br>735137                           |
| d.weekday()                 | 方法。返回星期。Monday 为 0，Sunday 为 6 | >>> d.weekday()<br>3                                  |
| d.isoweekday()              | 方法。返回星期。Monday 为 1，Sunday 为 7 | >>> d.isoweekday()<br>4                               |

| 续表                                         |                                        |                                             |
|--------------------------------------------|----------------------------------------|---------------------------------------------|
| 方法和属性                                      | 说 明                                    | 示 例                                         |
| d.isocalendar()                            | 方法。返回日历元组：(year, week number, weekday) | >>> d.isocalendar()<br>(2013,39,4)          |
| d.isoformat()<br>d.__str__()               | 方法。返回 ISO 8601 格式字符串                   | >>> d.isoformat()<br>'2013-09-26'           |
| d.ctime()                                  | 方法。换为日期时间字符串（参照 time.ctime()）          | >>> d.ctime()<br>'Thu Sep 26 00:00:00 2013' |
| d.strftime(format)<br>d.__format__(format) | 方法。转换为格式化为字符串                          | >>> d.strftime("%Y/%m/%d")<br>'2013/09/26'  |

13.3.2 tzinfo 对象和 timezone 对象

tzinfo 对象表示时区信息。tzinfo 是一个抽象类，不能直接创建其对象实例。可以声明其派生类，并创建对象实例。datetime 模块 timezone 是 tzinfo 的派生类，timezone 对象用于表示相对于 UTC 固定偏差的时区。

1. tzinfo 的类方法

tzinfo 类包括下列方法，声明其派生类时可能需要实现这些方法。

- tzinfo.utcoffset(dt)：返回本地 dt 与 UTC 的偏差（分钟）。
- tzinfo.dst(dt)：返回 dt 夏令时矫正（分钟）。
- tzinfo.tzname(dt)：返回 dt 对应时区的名称。
- tzinfo.fromutc(dt)：UTC 转换为本地日期时间。

2. timezone 对象

timezone 对象的构造函数为：

```
timezone(offset[,name])
```

其中，offset 是 timedelta 对象，取值范围为：-timedelta（hours = 24）和 timedelta（hours = 24），只能为分钟的整数倍；可选的 name 为时区名称，默认为‘UTCsHH:MM’形式。

timezone 对象 tz 包含下列方法。

- tz.utcoffset(dt)：返回 tz 的 offset，与 dt 无关。
- tz.tzname(dt)：返回 tz 的 name，与 dt 无关。
- tz.dst(dt)：返回 None。
- tz.fromutc(dt)：UTC 转换为本地日期时间，即 dt + offset。
- timezone.utc：类属性，0 时区。等同于 timezone(timedelta(0))。

例如：

```
>>> from datetime import *
>>> tz = timezone(timedelta(hours = 2)); dt = datetime(2013,9,26,9,31,45,tzinfo = tz)
>>> tz.utcoffset(dt) #datetime.timedelta(0,7200)
>>> tz.tzname(dt) #UTC + 02:00'
>>> tz.fromutc(dt)
datetime.datetime(2013,9,26,11,31,45,tzinfo = datetime.timezone(datetime.timedelta(0,7200)))
```

13.3.3 time 对象

time 对象表示由时、分、秒和微秒组成的简单时间。

1. 创建 time 对象

time 对象的构造函数为：

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

其中， $0 \leq \text{hour} < 24$ ， $0 \leq \text{minute} < 60$ ， $0 \leq \text{second} < 60$ ， $0 \leq \text{microsecond} < 1000000$ ，tzinfo 为时区信息。

2. time 对象的方法

time 对象包含的主要方法和属性如表 13-5 所示。假设表中示例基于：

```
>>> import datetime ; t = datetime. time(9 , 31 , 45)
```

表 13-5 time 的主要方法和属性

| 方法和属性                                                                     | 说 明                    | 示 例                                                                |
|---------------------------------------------------------------------------|------------------------|--------------------------------------------------------------------|
| time. min                                                                 | 类属性。返回最小时间             | >>> datetime. time. min<br>datetime. time( 0 , 0 )                 |
| time. max                                                                 | 类属性。返回最大时间             | >>> datetime. time. max<br>datetime. time( 23 , 59 , 59 , 999999 ) |
| time. resolution                                                          | 类属性。返回时间精度（1 秒）        | >>> datetime. time. resolution<br>datetime. timedelta( 0 , 0 , 1 ) |
| t. hour、t. minute、t. second                                               | 属性。返回时、分、秒             | >>> t. hour , t. minute , t. second<br>( 9 , 31 , 45 )             |
| t. microsecond                                                            | 属性。返回微秒。范围为：0 ~ 999999 | >>> t. microsecond<br>0                                            |
| t. tzinfo                                                                 | 属性。返回时区信息              | >>> t. tzinfo                                                      |
| t. replace( [ hour[ , minute[ , second[ , microsecond[ , tzinfo ] ] ] ] ) | 方法。替换时、分、秒、微秒、时区信息     | >>> t. replace ( hour = 10 )<br>datetime. time( 10 , 31 , 45 )     |
| t. isoformat( )<br>t. __str__( )                                          | 方法。返回 ISO 8601 格式字符串   | >>> t. isoformat( )<br>'09 : 31 : 45 '                             |
| t. strftime( format )<br>t. __format__( format )                          | 方法。转换为格式化为字符串          | >>> t. strftime( " % H 时 % M 分 % S 秒 " )<br>'09 时 31 分 45 秒 '      |

13.3.4 datetime 对象

datetime 对象表示日期和时间，包含了 date 对象和 time 对象的信息。

1. 创建 datetime 对象

datetime 对象的构造函数为：

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

2. 通过 datetime 类工厂函数获取

datetime. today( )：获得 datetime 对象（本地当前时间）。

datetime. now( tz = None )：获得 datetime 对象（当前时间，可指定时区）。



`datetime.utcnow()`：获得 `datetime` 对象（UTC 当前时间）。

`datetime.fromtimestamp(timestamp, tz = None)`：获得 `datetime` 对象（指定时间戳，可指定时区）。

`datetime.utcfromtimestamp(timestamp)`：获得 `datetime` 对象（指定 UTC 时间戳）。

`datetime.fromordinal(ordinal)`：获得 `datetime` 对象（指定从 1/1/1 开始的天数）。

`datetime.combine(date, time)`：获得 `datetime` 对象（指定 `date` 和 `time` 对象）。

`datetime.strptime(date_string, format)`：获得 `datetime` 对象（指定格式化字符串）。

例如：

```
>>> from datetime import * ;import time
>>> datetime.today() #datetime.datetime(2013,9,26,10,14,29,44656)
>>> datetime.now() #datetime.datetime(2013,9,26,10,14,37,89117)
>>> datetime.utcnow() #datetime.datetime(2013,9,26,2,14,53,640064)
>>> datetime.fromtimestamp(time.time())
datetime.datetime(2013,9,26,10,15,42,676867)
>>> datetime.utcfromtimestamp(time.time())
datetime.datetime(2013,9,26,2,16,21,673099)
>>> datetime.fromordinal(2) #datetime.datetime(1,1,2,0,0)
```

3. `datetime` 对象的方法

`datetime` 对象包含的主要方法和属性如表 13-6 所示。假设表中示例基于：

```
>>> import datetime;dt = datetime.datetime(2013,9,26,9,31,45)
```

表 13-6 `datetime` 的主要方法和属性

| 方法和属性                                                                                                                                                                      | 说 明                                                  | 示 例                                                                                                  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <code>datetime.min</code>                                                                                                                                                  | 类属性。返回最小日期                                           | <pre>&gt;&gt;&gt; datetime.datetime.min datetime.datetime(1,1,1,0,0)</pre>                           |
| <code>datetime.max</code>                                                                                                                                                  | 类属性。返回最大日期                                           | <pre>&gt;&gt;&gt; datetime.datetime.max datetime.datetime(9999,12,31,23,59,59,999999)</pre>          |
| <code>datetime.resolution</code>                                                                                                                                           | 类属性。日期精度，即 2 个日期之间的最小差（1 天）                          | <pre>&gt;&gt;&gt; datetime.datetime.resolution datetime.timedelta(0,0,1)</pre>                       |
| <code>dt.year</code> 、 <code>dt.month</code> 、 <code>dt.day</code><br><code>dt.hour</code> 、 <code>dt.minute</code> 、 <code>dt.second</code> 、 <code>dt.microsecond</code> | 属性。返回年、月、日、时、分、秒、微秒                                  | <pre>&gt;&gt;&gt; dt.year, dt.month, dt.day, dt.hour, dt.minute, dt.second (2013,9,26,9,31,45)</pre> |
| <code>dt.date()</code>                                                                                                                                                     | 方法。返回 <code>date</code> 对象                           | <pre>&gt;&gt;&gt; dt.date() datetime.date(2013,9,26)</pre>                                           |
| <code>dt.time()</code><br><code>dt.timetz()</code>                                                                                                                         | 方法。返回不带和带 <code>tzinfo</code> 的 <code>time</code> 对象 | <pre>&gt;&gt;&gt; dt.time() datetime.time(9,31,45)</pre>                                             |
| <code>dt.replace([year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]])</code>                                                                        | 方法。替换年、月、日、时、分、秒、微秒、时区信息                             | <pre>&gt;&gt;&gt; dt.replace(month=10) datetime.datetime(2013,10,26,9,31,45)</pre>                   |
| <code>dt.timetuple()</code><br><code>dt.utctimetuple()</code>                                                                                                              | 方法。返回 <code>struct_time</code> 对象                    | <pre>&gt;&gt;&gt; dt.timetuple()</pre>                                                               |



续表

| 方法和属性                                             | 说 明                                                                | 示 例                                                        |
|---------------------------------------------------|--------------------------------------------------------------------|------------------------------------------------------------|
| dt. toordinal( )                                  | 方法。返回从 1/1/1 开始的天数                                                 | >>> dt. toordinal( )<br>735137                             |
| dt. timestamp( )                                  | 方法。返回时间戳                                                           | >>> dt. timestamp( )<br>1380159105.0                       |
| dt. weekday( )                                    | 方法。返回星期。Monday 为 0，Sunday 为 6                                      | >>> dt. weekday( )<br>3                                    |
| dt. isoweekday( )                                 | 方法。返回星期。Monday 为 1，Sunday 为 7                                      | >>> dt. isoweekday( )<br>4                                 |
| dt. isocalendar( )                                | 方法。返回日历元组： ( year, week number, weekday)                           | >>> dt. isocalendar( )<br>(2013,39,4)                      |
| dt. isoformat ( sep ='T')<br>dt. __str__( )       | 方法。返回 ISO 8601 格式字符串。<br>d. __str__( ) 等同于： dt. isoformat( sep =") | >>> dt. isoformat ( sep ='、')<br>'2013 - 09 - 26、09：31：45' |
| dt. ctime( )                                      | 方法。换为日期时间字符串（参<br>照 time. ctime( )）                                | >>> dt. ctime( )<br>'Thu Sep 26 09：31：45 2013'             |
| dt. strftime ( format)<br>dt. __format__( format) | 方法。转换为格式化为字符串                                                      | >>> dt. strftime( " % Y/% m/% d" )<br>'2013/09/26'         |

4. 日期的字符串格式化和日期字符串的解析

date、time 和 datetime 对象的 strftime( ) 方法将 struct\_time 对象格式化为字符串；datetime 类方法 strptime( ) 将字符串解析为 datetime 对象。

dt. strftime ( format)：datetime 对象格式化为日期时间字符串。

datetime. strptime ( date\_string, format)：日期时间字符串的解析为 datetime 对象。

strftime( ) 方法大致等价于 time. strftime ( fmt, d. timetuple( ))。datetime. strptime ( date\_string, format) 等价于 datetime( \*( time. strptime( date\_string, format)[0:6] ) )。

其中，format 为日期格式化字符串。例如：

```
>>> from datetime import * ;dt = datetime(2013,9,26,9,31,45)
>>> dt. strftime(" % Y 年% m 月% d 日 (% A) % H 时% M 分% S 秒")
'2013 年 09 月 26 日 (Thursday) 09 时 31 分 45 秒'
```

13.3.5 timedelta 对象

timedelta 对象表示两个日期或时间之间的差值。

1. 创建 timedelta 对象

timedelta 对象的构造函数为：

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

其中，0 <= microseconds < 1000000，0 <= seconds < 3600 \* 24，- 999999999 <= days <= 999999999。1 millisecond 转换为 1000microseconds，1 minute 转换为 60seconds，1 hour 转换为 3600seconds，1 week 转换为 7days。

timedelta 对象 td 具有下列属性：td. days 获取天数，td. seconds 获取秒数，td. microseconds 获取毫秒数。

2. timedelta 的运算

timedelta 支持的运算与比较如表 13-7 所示。假设表中示例基于：

```
>>> from datetime import *
>>> td1 = timedelta(hours = 2) #时间差:2 小时
>>> td2 = timedelta(hours = 3) #时间差:3 小时
```

表 13-7 timedelta 的运算与比较

| 运算与比较           | 说 明         | 示 例                                                        |
|-----------------|-------------|------------------------------------------------------------|
| 时间差 2 + 时间差 1   | 时间差相加       | >>> td2 + td1<br>datetime.timedelta (0, 18000) #相差 18000 秒 |
| 时间差 2 - 时间差 1   | 时间差相减       | >>> td2 - td1<br>datetime.timedelta (0, 3600) #相差 3600 秒   |
| 时间差 * 整数        | 返回时间差       | >>> td1 * 10<br>datetime.timedelta (0, 72000) #相差 72000 秒  |
| <、<=、>、>=、==、!= | 比较<br>返回布尔值 | >>> td1 < td2<br>True                                      |

13.3.6 日期和时间的运算与比较

date 和 datetime 对象支持的运算与比较如表 13-8 所示。假设表中示例基于：

```
>>> from datetime import *
>>> dt1 = datetime(2013,3,1);dt2 = datetime(2013,12,1)
>>> td = timedelta(days = 200) #时间差:200 天
```

表 13-8 date/datetime 的运算与比较

| 运算和比较           | 说 明                 | 示 例                                                 |
|-----------------|---------------------|-----------------------------------------------------|
| 日期 2 - 日期 1     | 返回日期之差，timedelta 对象 | >>> dt2 - dt1<br>datetime.timedelta (275) #相差 275 天 |
| 日期 + 时间差        | 返回日期                | >>> dt1 + td<br>datetime.datetime(2013,9,17,0,0)    |
| 日期 - 日期差        | 返回日期                | >>> dt2 - td<br>datetime.datetime(2013,5,15,0,0)    |
| <、<=、>、>=、==、!= | 日期大小比较<br>返回布尔值     | >>> dt1 > dt2<br>False                              |

13.4 calendar 模块

calendar 模块包含若干用于处理日历的类和函数。

13.4.1 Calendar 对象

Calendar 对象用于表示日历。对于 Calendar 对象，星期的第一天默认为 0（Monday），最后一天为 6（Sunday）。

1. 创建 Calendar 对象

Calendar 对象的构造函数为：

`Calendar( firstweekday = 0 )`

其中, `firstweekday` 为星期的第一天, 默认为 0 (Monday)。

## 2. Calendar 对象的方法

Calendar 对象的主要方法如下。

`iterweekdays()`: 返回星期的数字迭代器。

`itermonthdates(year, month)`: 返回指定年月的日期 (date 对象), 包括前后的日期 (完整的星期)。

`itermonthdays2(year, month)`: 返回指定年月的日期 (日期格式为元组 (day, week))。

`itermonthdays(year, month)`: 返回指定年月的日期 (数字)。

`monthdatescalendar(year, month)`: 返回指定年月的星期列表, 列表元素为 7 个 date 对象。

`monthdays2calendar(year, month)`: 返回指定年月的星期列表, 列表元素为 7 个元组 (day, week)。

`monthdayscalendar(year, month)`: 返回指定年月的星期列表, 列表元素为 7 个数字。

`yeardatescalendar(year, width = 3)`: 返回指定年的日历数据。

`yeardays2calendar(year, width = 3)`: 返回指定年的日历数据。

`yeardayscalendar(year, width = 3)`: 返回指定年的日历数据。

例如:

```
>>> import calendar; cal = calendar.Calendar()
>>> list(cal.iterweekdays()) #[0,1,2,3,4,5,6]
>>> list(cal.itermonthdates(2013,9))
[datetime.date(2013,8,26),datetime.date(2013,8,27),datetime.date(2013,8,28),datetime.date(2013,8,29),datetime.date(2013,8,30),datetime.date(2013,8,31),datetime.date(2013,9,1),datetime.date(2013,9,2),...(略)]
>>> list(cal.itermonthdays2(2013,9))
[(0,0),(0,1),(0,2),(0,3),(0,4),(0,5),(1,6),(2,0),(3,1),(4,2),(5,3),...(略)]
>>> list(cal.itermonthdays(2013,9))
[0,0,0,0,0,0,1,2,3,4,5,6,7,8,9,...(略)]
>>> list(cal.monthdayscalendar(2013,9))
[[0,0,0,0,0,0,1],[2,3,4,5,6,7,8],[9,10,11,12,13,14,15],[16,17,18,19,20,21,22],[23,24,25,26,27,28,29],[30,0,0,0,0,0,0]]
>>> list(cal.yeardayscalendar(2013,9))
[[[0,1,2,3,4,5,6],[7,8,9,10,11,12,13],[14,15,16,17,18,19,20],...(略)]
```

## 13.4.2 TextCalendar 对象和 LocaleTextCalendar 对象

TextCalendar 对象用于生成文本格式的日历, 其派生类 LocaleTextCalendar 用于生成本地区域的文本格式日历。

### 1. 创建 TextCalendar 对象和 LocaleTextCalendar 对象

TextCalendar 对象的构造函数为:



TextCalendar( firstweekday = 0)

LocaleTextCalendar( firstweekday = 0, locale = None)

其中，firstweekday 为星期的第一天，默认为 0（Monday）；可选参数 locale 为区域信息。

2. TextCalendar 对象的方法

TextCalendar 对象包含的主要方法如下。

formatmonth( theyear, themonth, w = 0, l = 0)：返回指定年月的格式化字符串。

prmonth( theyear, themonth, w = 0, l = 0)：打印指定年月的格式化日历。

formatyear( theyear, w = 2, l = 1, c = 6, m = 3)：返回指定年的格式化字符串。

pryear( theyear, w = 2, l = 1, c = 6, m = 3)：打印指定年的格式化日历。

其中，theyear 为指定年；themoth 为指定月；可选的 w、l、c、m 为日期列的宽度、每个星期的行数、月份之间的空格数、每行的月份数（默认横向打印 3 个月）。例如：

```
>>> import calendar; textcal = calendar.TextCalendar()
>>> textcal.formatmonth(2014,9)
' September 2014\nMo Tu We Th Fr Sa Su\n 1 2 3 4 5 6 7\n 8 9 10 11 12 13 14\n15 16 17 18 19 20 21\n22 23 24 25 26 27 28\n29 30\n'
>>> textcal.prmonth(2014,9)
```

| September 2014 |    |    |    |    |    |    |
|----------------|----|----|----|----|----|----|
| Mo             | Tu | We | Th | Fr | Sa | Su |
| 1              | 2  | 3  | 4  | 5  | 6  | 7  |
| 8              | 9  | 10 | 11 | 12 | 13 | 14 |
| 15             | 16 | 17 | 18 | 19 | 20 | 21 |
| 22             | 23 | 24 | 25 | 26 | 27 | 28 |
| 29             | 30 |    |    |    |    |    |

```
>>> textcal.pryear(2014)
```

| 2014    |    |    |    |    |    |    |          |    |    |    |    |    |    |       |    |    |    |    |    |    |
|---------|----|----|----|----|----|----|----------|----|----|----|----|----|----|-------|----|----|----|----|----|----|
| January |    |    |    |    |    |    | February |    |    |    |    |    |    | March |    |    |    |    |    |    |
| Mo      | Tu | We | Th | Fr | Sa | Su | Mo       | Tu | We | Th | Fr | Sa | Su | Mo    | Tu | We | Th | Fr | Sa | Su |
|         |    |    |    |    |    |    |          |    |    |    |    |    |    |       |    |    |    |    |    |    |
|         |    |    |    |    |    |    |          |    |    |    |    |    |    |       |    |    |    |    |    |    |
| 6       | 7  | 8  | 9  | 10 | 11 | 12 | 3        | 4  | 5  | 6  | 7  | 8  | 9  | 3     | 4  | 5  | 6  | 7  | 8  | 9  |
| 13      | 14 | 15 | 16 | 17 | 18 | 19 | 10       | 11 | 12 | 13 | 14 | 15 | 16 | 10    | 11 | 12 | 13 | 14 | 15 | 16 |
| 20      | 21 | 22 | 23 | 24 | 25 | 26 | 17       | 18 | 19 | 20 | 21 | 22 | 23 | 17    | 18 | 19 | 20 | 21 | 22 | 23 |
| 27      | 28 | 29 | 30 | 31 |    |    | 24       | 25 | 26 | 27 | 28 |    |    | 24    | 25 | 26 | 27 | 28 | 29 | 30 |
|         |    |    |    |    |    |    |          |    |    |    |    |    |    |       |    |    |    |    |    | 31 |

…(略)

再如：

```
>>> import locale,calendar; loc = locale.getlocale() #获取当前系统的 locale
>>> localtextcal = calendar.LocaleTextCalendar(locale = loc)
>>> localtextcal.prmonth(2014,9,w = 5)
```

| 九月 2014 |    |    |    |    |    |    |
|---------|----|----|----|----|----|----|
| 周一      | 周二 | 周三 | 周四 | 周五 | 周六 | 周日 |
| 1       | 2  | 3  | 4  | 5  | 6  | 7  |
| 8       | 9  | 10 | 11 | 12 | 13 | 14 |
| 15      | 16 | 17 | 18 | 19 | 20 | 21 |
| 22      | 23 | 24 | 25 | 26 | 27 | 28 |
| 29      | 30 |    |    |    |    |    |

13.4.3 HTMLCalendar 对象和 LocaleHTMLCalendar 对象

HTMLCalendar 对象用于生成 HTML 格式的日历。



## 1. 创建 HTMLCalendar 对象和 LocaleHTMLCalendar 对象

HTMLCalendar 对象的构造函数为：

```
HTMLCalendar(firstweekday = 0)
```

其中，firstweekday 为星期的第一天，默认为 0（Monday）。

## 2. HTMLCalendar 对象的方法

HTMLCalendar 对象包含的主要方法如下。

```
formatmonth(theyear, themonth, withyear = True) #返回星期的数字迭代器
```

```
formatyear(theyear, width = 3)
```

```
formatyearpage(theyear, width = 3, css = 'calendar. css', encoding = None)
```

其中，theyear 为指定年；themoth 为指定月；可选参数 withyear 指定标题是否带年份（默认为 True）；可选参数 width 指定每行的月份数（默认横向打印 3 个月）；可选参数 css 指定样式表；可选参数 encoding 指定编码。例如：

```
>>> import calendar; htmlcal = calendar. HTMLCalendar()
>>> htmlcal. formatmonth(2013, 9)
' < table border = "0" cellpadding = "0" cellspacing = "0" class = "month" > \n < tr > < th colspan = "7"
class = "month" > September 2013 < /th > < /tr > \n < tr > < th class = "mon" > Mon < /th > ... (略)
```

## 13.4.4 calendar 模块的属性和常用函数

### 1. calendar 模块的属性

calendar 模块包括下列属性。

calendar. day\_name：星期名一览。

calendar. day\_abbr：星期缩略名一览。

calendar. month\_name：月份名一览。

calendar. month\_abbr：月份缩略名一览。

calendar. MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY 以及 SUNDAY：对应星期。

例如：

```
>>> import calendar
>>> calendar. day_name # < calendar. _localized_day object at 0x03040B70 >
>>> list(calendar. day_name)
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
>>> list(calendar. day_abbr) # ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
>>> list(calendar. month_name)
[' ', 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October',
'November', 'December']
>>> list(calendar. month_abbr)
[' ', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
>>> calendar. MONDAY #0
```

## 2. calendar 模块的常用函数

calendar 模块包括下列常用函数。

calendar.setfirstweekday(weekday): 设置星期的第一天。

calendar.firstweekday(): 返回星期的第一天。

calendar.isleap(year): 判断指定年是否为闰年。

calendar.leapdays(y1,y2): 返回指定年范围[y1,y2)中的闰年数目。

calendar.weekday(year, month, day): 返回指定年月日的星期, 0 为 Monday, 6 为 Sunday。

calendar.weekheader(n): 返回日历的头部(包括星期名), n 为各星期名的宽度。

calendar.monthrange(year, month): 返回指定年月的元组(第 1 天的星期, 天数)。

calendar.monthcalendar(year, month): 返回指定年月的日历列表。

calendar.prmnth(theyear, themonth, w=0, l=0): 打印指定年月的格式化日历。

calendar.month(theyear, themonth, w=0, l=0): 返回指定年月的格式化字符串。

calendar.prcal(year, w=0, l=0, c=6, m=3): 打印指定年的格式化日历。

calendar.calendar(year, w=2, l=1, c=6, m=3): 返回指定年的格式化字符串。

calendar.timegm(tuple): 将 struct\_time 对象转换时间戳, 是 time.gmtime([secs]) 的反函数。

例如:

```
>>> import calendar, time; calendar.setfirstweekday(calendar.SUNDAY)
>>> calendar.firstweekday() #6
>>> calendar.monthrange(2013, 9) #(6, 30)
>>> calendar.monthcalendar(2013, 9)
[[1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14], [15, 16, 17, 18, 19, 20, 21], [22, 23, 24, 25, 26, 27, 28],
 [29, 30, 0, 0, 0, 0, 0]]
>>> calendar.prmnth(2013, 9)
 September 2013
Su Mo Tu We Th Fr Sa
 1 2 3 4 5 6 7
 8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
>>> calendar.isleap(2004) #True
>>> calendar.timegm(time.gmtime(100)) #100
```

## 13.5 复习题

### 一、填空题

1. Python 中的\_\_\_\_\_模块包含各种用于日期和时间处理的类。\_\_\_\_\_模块包含用于处理日历的函数和类; \_\_\_\_\_模块包含用于处理时间的函数。

2. datetime 模块包含用于表示日期的\_\_\_\_\_对象、表示时间的\_\_\_\_\_对象和表示日期时间的\_\_\_\_\_对象, 表示日期或时间之间差值的\_\_\_\_\_对象, 表示时区信息的\_\_\_\_\_

对象和\_\_\_\_\_对象。

3. 使用 time 模块的\_\_\_\_\_函数可以获取当前系统的起始点。
4. time 模块中的\_\_\_\_\_属性用于判定是否使用夏令时。
5. time 模块中的\_\_\_\_\_函数将字符串解析为 struct\_time 对象; \_\_\_\_\_函数将 struct\_time 对象格式化为字符串。
6. datetime 模块包括两个常量: \_\_\_\_\_和\_\_\_\_\_, 表示最小年份和最大年份, 分别为\_\_\_\_\_和\_\_\_\_\_。
7. date、time 和 datetime 对象的\_\_\_\_\_方法将 struct\_time 对象格式化为字符串; datetime 类方法\_\_\_\_\_将字符串解析为 datetime 对象。
8. timedelta 对象 td 的属性\_\_\_\_\_获取天数, \_\_\_\_\_获取秒数, \_\_\_\_\_获取毫秒数。
9. Python 中 calendar.isleap(2000) 的结果为\_\_\_\_\_。

## 二、思考题

1. 下列 Python 语句的执行结果是: \_\_\_\_\_。

```
from datetime import *;import time,datetime
print(date. min,date. max,date. fromordinal(32))
d = date(2014,10,1);print(d. year,d. month,d. day);d. replace(month = 12)
print(d. toordinal(),d. weekday(),d. ctime(),d. strftime("%Y/%m/%d(%a)"))
```

2. 下列 Python 语句的执行结果是: \_\_\_\_\_。

```
import datetime;t = datetime. time(19,30,45,196)
print(datetime. time. min,datetime. time. max)
print(t. hour,t. minute,t. second,t. microsecond)
t. replace(hour = 23);print(t. strftime("%H 时%M 分%S 秒"))
```

3. 下列 Python 语句的执行结果是: \_\_\_\_\_。

```
import datetime;dt = datetime. datetime(2014,5,1,9,35,46)
print(datetime. datetime. min,datetime. datetime. max)
print(dt. year,dt. month,dt. day,dt. hour,dt. minute,dt. second)
print(dt. date(),dt. time(),dt. strftime("%Y/%m/%d(%A),%H 时%M 分%S 秒"))
```

4. 下列 Python 语句的执行结果是: \_\_\_\_\_。

```
from datetime import *;td1 = timedelta(minutes = 10)
td2 = timedelta(minutes = 15);print(td1 + td2,(td2 - td1). seconds,td1 * 10,td1 < td2)
```

5. 下列 Python 语句的执行结果是: \_\_\_\_\_。

```
from datetime import *;dt1 = date(2014,6,1);dt2 = date(2014,5,1)
td = timedelta(days = 10);print((dt1 - dt2). days,dt1 + td,dt1 - td,dt1 > dt2)
```

## 13.6 上机实践

1. 编写程序,打印 2014 年 1 月份~12 月份的日历,以及 2014 年 12 月份的本地区域文本格式的日历。运行效果如图 13-2 所示。其中,1 月份~12 月份的日历只截取了部分信息。



| 2014    |    |    |    |    |    |    |  |          |    |    |    |    |    |    |  |       |    |    |    |    |    |    |  |
|---------|----|----|----|----|----|----|--|----------|----|----|----|----|----|----|--|-------|----|----|----|----|----|----|--|
| January |    |    |    |    |    |    |  | February |    |    |    |    |    |    |  | March |    |    |    |    |    |    |  |
| Mo      | Tu | We | Th | Fr | Sa | Su |  | Mo       | Tu | We | Th | Fr | Sa | Su |  | Mo    | Tu | We | Th | Fr | Sa | Su |  |
|         |    | 1  | 2  | 3  | 4  | 5  |  |          |    |    |    |    | 1  | 2  |  |       |    |    |    |    | 1  | 2  |  |
| 6       | 7  | 8  | 9  | 10 | 11 | 12 |  | 3        | 4  | 5  | 6  | 7  | 8  | 9  |  | 3     | 4  | 5  | 6  | 7  | 8  | 9  |  |
| 13      | 14 | 15 | 16 | 17 | 18 | 19 |  | 10       | 11 | 12 | 13 | 14 | 15 | 16 |  | 10    | 11 | 12 | 13 | 14 | 15 | 16 |  |
| 20      | 21 | 22 | 23 | 24 | 25 | 26 |  | 17       | 18 | 19 | 20 | 21 | 22 | 23 |  | 17    | 18 | 19 | 20 | 21 | 22 | 23 |  |
| 27      | 28 | 29 | 30 | 31 |    |    |  | 24       | 25 | 26 | 27 | 28 |    |    |  | 24    | 25 | 26 | 27 | 28 | 29 | 30 |  |
|         |    |    |    |    |    |    |  |          |    |    |    |    |    |    |  | 31    |    |    |    |    |    |    |  |

| 十二月 2014 |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|
| 周一       | 周二 | 周三 | 周四 | 周五 | 周六 | 周日 |
| 1        | 2  | 3  | 4  | 5  | 6  | 7  |
| 8        | 9  | 10 | 11 | 12 | 13 | 14 |
| 15       | 16 | 17 | 18 | 19 | 20 | 21 |
| 22       | 23 | 24 | 25 | 26 | 27 | 28 |
| 29       | 30 | 31 |    |    |    |    |

图 13-2 2014 年日历运行效果

2. 编写程序，定义一个返回指定年月的天数的函数 `ndays(y,m)`，并编写测试代码。运行效果如图 13-3 所示。提示：可使用 `calendar` 模块的 `isleap` 函数来判断闰年。

```
请输入年份 (>=1)，否则为1：2000
请输入月份 (1~12)，否则<1为1、>12为12：2
29
```

图 13-3 返回指定年月的天数运行效果

3. 编写程序，定义一个返回从公元 1 年 1 月 1 日（含）到  $y$  年  $m$  月  $d$  日（含）的天数的函数 `caldays(y,m,d)`，并编写测试代码。运行效果如图 13-4 所示。提示：计算从公元 1 年 1 月 1 日到  $y$  年  $m$  月  $d$  日的天数，可以分为三部分计算。（1）计算从公元 1 年到  $y-1$  年的天数，每年是 365 天或 366 天（闰年）。（2）对于第  $y$  年，先计算  $1\sim m-1$  月整月的天数。可利用上一题返回指定年月的天数的函数 `ndays(y,m)`。（3）最后加上零头（第  $m$  月的  $d$  天）。

```
请输入年份 (>=1)，否则为1：2014
请输入月份 (1~12)，否则<1为1、>12为12：10
请输入日期 (1~31)，否则<1为1、>ndays(y,m)为ndays(y,m)：1
从1年1月1日到 2014 年 10 月 1 日共 735507 天
```

图 13-4 返回总天数的运行效果

4. 编写程序，定义一个返回指定年月日的星期的函数 `weekday(y,m,d)`，结果为：星期一、…、星期日，并编写测试代码。运行效果如图 13-5 所示。提示：要求出指定年月日是星期几，只需调用第 3 题返回从公元 1 年 1 月 1 日到  $y$  年  $m$  月  $d$  日的天数的函数 `caldays(y,m,d)`，再除以 7 的余数即为星期几，余 0 就是星期日。

```
请输入年份 (>=1)，否则为1：2014
请输入月份 (1~12)，否则<1为1、>12为12：10
请输入日期 (1~31)，否则<1为1、>ndays(y,m)为ndays(y,m)：1
2014 年 10 月 1 日是 星期三
```

图 13-5 返回星期的运行效果



# 第 14 章 正则表达式

正则表达式提供了功能强大、灵活而又高效的方法来处理文本：快速分析大量文本以找到特定的字符模式；提取、编辑、替换或删除文本子字符串；将提取的字符串添加到集合以生成报告。正则表达式广泛用于各种字符串处理应用程序，如 HTML 处理、日志文件分析和 HTTP 标头分析等。

## 本章要点：

- ◆ 正则表达式语言；
- ◆ 正则表达式模块；
- ◆ 正则表达式应用。

## 14.1 正则表达式语言

### 14.1.1 正则表达式语言概述

在文本字符串处理时，常常需要查找符合某些复杂规则（也称为模式）的字符串。正则表达式语言就是用于描述这些规则（模式）的语言。使用正则表达式，可以匹配和查找字符串，并对其进行相应的修改处理。

正则表达式是由普通字符（如字符 a 到 z）及特殊字符（称为元字符）组成的文字模式，元字符包括：`. ^ $ * + ? { } [ ] \ | ( )`。例如：

|        |                                                            |
|--------|------------------------------------------------------------|
| "Go"   | #匹配字符串"God Good"中的"Go"                                     |
| "G. d" | #匹配字符串"God Good"中的"God", <code>.</code> 为元字符,匹配除行终止符外的任何字符 |
| "d \$" | #匹配字符串"God Good"中的最后一个"d", <code>\$</code> 为元字符,匹配结尾       |

正则表达式的模式可以包含普通字符（包括转义字符）、字符类和预定义字符类、边界匹配符、重复限定符、选择分支、分组和引用等。

### 14.1.2 正则表达式引擎

正则表达式引擎是一种可以处理正则表达式的软件。流行的计算机语言都包含支持正则表达式处理的类库。Python 的模块 `re` 实现了正则表达式处理的功能。导入 `re` 模块后，使用 `findall`、`search` 函数可以进行匹配（参见 14.2 节）。

|                                          |                                                          |
|------------------------------------------|----------------------------------------------------------|
| <code>re.findall(pattern, string)</code> | : 返回匹配结果列表。                                              |
| <code>re.search(pattern, string)</code>  | : 如果匹配，返回 <code>Match</code> 对象，否则返回 <code>None</code> 。 |

例如：

```
>>> import re #导入模块 re
>>> re.findall('d','godness') #['d']
```

### 14.1.3 普通字符和转义字符

最基本的正则表达式由单个或多个普通字符组成，用以匹配字符串中对应的单个或多个普通字符。普通字符包括 ASCII 字符、Unicode 字符和转义字符（参见 5.4.1 节）。

另外，正则表达式中的元字符（`^$*+?{}[]\|()`），包含特殊含义，如果要作为普通字符使用，则需要转义。例如：`\$`。

```
>>> re.findall("fo","the quick brown fox jumps for food") #['fo','fo','fo']
>>> re.findall("1+1=2","1+1=2") #元字符+重复一次或多次,即匹配11=2。结果:[]
>>> re.findall("1+1=2","11=2") #元字符+重复一次或多次,即匹配11=2。结果:['11=2']
>>> re.findall("1\\+1=2","1+1=2") #转义元字符+,即匹配1+1=2。结果:['1+1=2']
>>> re.findall("(note)","please (note)") #()在正则表达式中为分组。结果:['note']
>>> re.findall("\\(note\\)","please (note)") #\\(匹配圆括号。结果:['(note)']
```

**注：**正则表达式中包含特殊字符，例如：`\\b` 表示单词边界；而字符串中的转义字符 `\\b` 表示退格字符。因此在正则表达式中，这些与标准转义字符重复的特殊符号必须使用两个反斜线字符（`\\`），或者使用原始字符串 `r""` 或 `r"`。例如：

```
>>> re.findall("\\bon\\b","only on air") #\\b 匹配退格符。结果:[]
>>> re.findall("\\\\bon\\\\b","only on air") #\\\\b 匹配单词边界。结果:['on']
>>> re.findall(r"\\bon\\b","only on air") #使用原始字符串,\\b 匹配单词边界。结果:['on']
```

### 14.1.4 字符类和预定义字符类

#### 1. 字符类

字符类是由一对方括号 `[]` 括起来的字符集合，正则表达式引擎匹配字符集中的任意一个字符。字符类的定义方式包括以下几种。

`[xyz]`：枚举字符集，匹配括号中的任意字符。例如，`t[aeio]n` 匹配 `tan`、`ten`、`tin`、`ton`。

`[^xyz]`：否定枚举字符集，匹配不在此括号中的任意字符。例如：`[^aeiou]`。

`[a-z]`：指定范围的字符，匹配指定范围的任意匹配。例如：`[0-9]`。

`[^m-z]`：指定范围以外的字符，匹配指定范围以外的任意匹配。例如：`[^0-9]`

例如：

```
>>> re.findall("fo[xr]","the quick brown fox jumps for food") #['fox','for']
```

#### 2. 预定义字符类

使用正则表达式时，常常用到一些特定的字符类，例如数字字母。正则表达式语言包含若干预定义字符类，这些预定义字符集通常使用缩写形式，例如：`\\d` 等价于 `[0-9]`。常用的预定义字符类参见表 14-1。

表 14-1 常用的预定义字符类

| 预定义字符 | 说 明                    |
|-------|------------------------|
| .     | 除行终止符外的任何字符            |
| \d    | 数字。等价于[0-9]            |
| \D    | 非数字。等价于[^\d]           |
| \s    | 空白字符。等价于[\t\n\r\f\v]   |
| \S    | 非空白字符。等价于[^\t\n\r\f\v] |
| \w    | 单词字符。等价于[a-zA-Z0-9_]   |
| \W    | 非单词字符。等价于[^\w]         |

14.1.5 边界匹配符

字符串匹配往往涉及从某个位置开始匹配，例如行的开头或结尾、单词边界等。边界匹配符用于匹配字符串的位置，参见表 14-2。

表 14-2 边界匹配符

| 边界匹配符 | 含 义            | 说 明                                                                                                                 |
|-------|----------------|---------------------------------------------------------------------------------------------------------------------|
| ^     | 行开头            | (1) "^a" 匹配"abc" 中的"a", "^a" 不匹配"abc" 中的"b";<br>(2) "^\s*" 匹配" abc " 中的左边空格                                         |
| \$    | 行结尾            | (1) "c \$" 匹配"abc" 中的"c", "b \$" 不匹配"abc" 中的"b";<br>(2) "^123 \$" 匹配"123" 中的"123";<br>(3) "\s* \$" 匹配" abc " 中的右边空格 |
| \b    | 单词边界           | r'\bfoo\b' 匹配'foo'、'foo.'、'(foo)'、'bar foo baz'，但不匹配'foobar'或'foo3'                                                 |
| \B    | 非单词边界          | r'py \B' 匹配'python'、'py3'、'py2'，但不匹配'happy'、'sleepy.'、'py!'                                                         |
| \A    | 字符串开头          |                                                                                                                     |
| \Z    | 字符串结尾（除最后行终止符） |                                                                                                                     |

14.1.6 重复限定符

使用重复限定符，可以指定重复的次数。例如：中华人民共和国邮政编码由 6 位数字组成，使用重复限定符“\d{6}”，表示数字字母重复 6 次。重复限定符参见表 14-3。

表 14-3 重复限定符

| 重复限定符  | 说 明                                                                                    |
|--------|----------------------------------------------------------------------------------------|
| X?     | X 重复 0 次或 1 次，等价于 X{0,1}。例如,"colou? r" 可以匹配"color" 或者"colour"                          |
| X*     | X 重复 0 次或多次，等价于 X{0,}。例如,"zo*" 可以匹配"z"、"zo"、"zoo" 等                                    |
| X+     | X 重复 1 次或多次，等价于 X{1,}。例如,"zo+" 可以匹配"zo" 和"zoo"，但不匹配"z"                                 |
| X{n}   | X 重复 n 次。例如:\b[0-9]{3}，匹配 000~999。例如,"o{2}" 不能与"Bob" 中的"o" 匹配，但是可以与"food" 中的两个"o" 匹配   |
| X{n,}  | 至少重复 n 次。例如,"o{2,}" 不匹配"Bob" 中的"o"，但是匹配"fooooo" 中所有的 o。"o{1,}" 等价于"o+"。"o{0,}" 等价于"o*" |
| X{n,m} | 重复 n 到 m 次。例如,"o{1,3}" 匹配"fooooo" 中前三个 o。"o{0,1}" 等价于"o?"                              |



14.1.7 匹配算法：贪婪和懒惰

1. 贪婪性匹配算法

默认情况下，Python 正则表达式的匹配算法采用贪婪性算法。例如：

```
>>> re.findall("<.+>","<book><title>Python</title><author>Jiang<author></book>")
['<book><title>Python</title><author>Jiang<author></book>']
```

例中，正则表达式“<.+>”，以<开始，紧跟 1 个或多个字符，以>结束，即 XML 的开始或结束标签。但结果并不是“<book>”，这是因为 Python 正则表示匹配算法针对重复限定符，默认采用贪婪性匹配算法。

贪婪性匹配算法是指重复限定符会导致正则表达式引擎试图尽可能多的重复前导字符，只有当这种重复会引起整个正则表达式匹配失败的情况下，引擎会进行回溯。

上例中，正则表达式“<.+>”，第一个字符<为普通字符，匹配字符串的第一个<;“.”匹配 1 个或多个字符（换行符除外），即匹配字符 b 并一直匹配其余的字符，直至换行符，匹配失败（. 不匹配换行符）。于是引擎开始对下一个>进行匹配，引擎会试图将“>”与换行符进行匹配，结果失败了。于是引擎进行回溯，直到“<.+”匹配“<book><title>Python</title><author>Jiang<author></book>”，则>与>匹配。于是引擎找到了一个匹配“<book><title>Python</title><author>Jiang<author></book>”。

2. 懒惰性匹配算法

贪婪性算法返回了一个最左边的最长的匹配。如果在重复限定符后面加后缀“?”，则正则表达式引擎使用懒惰性匹配算法，如表 14-4 所示。

表 14-4 懒惰性匹配算法

| 符 号    | 说 明                 |
|--------|---------------------|
| *?     | 重复任意次，但尽可能少重复       |
| +?     | 重复 1 次或更多次，但尽可能少重复  |
| ??     | 重复 0 次或 1 次，但尽可能少重复 |
| {n,m}? | 重复 n 到 m 次，但尽可能少重复  |
| {n,}?? | 重复 n 次以上，但尽可能少重复    |

例如：

```
>>> re.findall("<.+?>","<book><title>Python</title><author>Jiang<author></book>")
['<book>','<title>','</title>','<author>','<author>','</book>']
```

懒惰性匹配是指重复限定符会导致正则表达式引擎试图尽可能少的重复前导字符，只有当这种重复会引起整个正则表达式匹配失败的情况下，引擎会进行回溯。

例中，正则表达式“<.+>”，第一个字符<为普通字符，匹配字符串的第一个<;“.”匹配 1 个或多个字符（换行符除外），即匹配字符 b；然后试图匹配>，匹配字符 o 失败，于是引擎回溯，“<.+”匹配“<bo”；然后试图匹配>，匹配字符 o 失败，于是引擎回溯，“<.+”匹配“<boo”；然后试图匹配>，匹配字符 k 失败，于是引擎回溯，“<.+”匹配“<book”，>匹配>。于是引擎找到了一个匹配“<book>”。



## 14.1.8 选择分支

正则表达式中“|”表示选择。用于选择符匹配多个可能的正则表达式中的一个。例如：“red | green | blue”。

正则表达式中，选择符“|”的优先级最低。如果需要，可以使用圆括号来限制选择符的作用范围。例如：“\b(red | green | blue)\b >>”。

例如，中国电话号码组成形式一般为“区号 - 电话号码”，区号为3位或4位数字，电话号码为6位或8位数字，故其正则表达式为：`(0\d{2}|10\d{3})-(\d{8}|\d{6})`。例如：

```
>>> re.findall(r"((0\d{2}|10\d{3})-(\d{8}|\d{6}))","电话号码 021-62232333")
[('021-62232333','021','62232333')]
```

如果正则表达式包含组，则 `re.findall` 返回组的列表。

## 14.1.9 分组和向后引用

### 1. 分组

重复限定符重复前导字符，如果需要重复多个字符，则需要把正则表达式的一部分放在圆括号（）内，形成分组。然后对整个组使用诸如重复操作符的正则操作。

例如 IP 地址的一般形式为“ddd.ddd.ddd.ddd”，ddd. 重复了三次，可以使用分组：`(\d{1,3}\.){3}\d{1,3}`。再如：

```
>>> re.findall("((\d{1,3}\.){3}\d{1,3})", "IP 地址 192.168.0.1") # [('192.168.0.1','0. ')]
```

### 2. 分组缓存和引用

当用“（）”定义了一个正则表达式组后，正则引擎则会把被匹配的组按照顺序编号，存入缓存。对被匹配的组可以进行向后引用：，\1 引用第一个匹配的组，\2 引用第二个组，以此类推。而\0则引用整个被匹配的正则表达式本身。

分组引用一般用于对称的模式，例如 HTML 的开始和结束标签。例如网页中包含开始标签和结束标签及中间文本，`<h1>News</h1>`，可以使用正则表达式：

```
<([\a-zA-Z][a-zA-Z0-9]*)[>]*>.*?</\1>
```

首先，<匹配第一个字符<；然后[a-zA-Z]匹配h，[a-zA-Z0-9]\*将会匹配0到多次字母数字，后面紧接着0到多个非>的字符。最后正则表达式的>将会匹配“<B>”的>。接下来正则引擎将对结束标签之前的字符进行惰性匹配，直到遇到一个“</”符号。然后正则表达式中的“\1”表示对前面匹配的组“([a-zA-Z][a-zA-Z0-9]\*)”进行引用，引擎缓存的内容为h1，所以需要被匹配的结尾标签为“</h1>”。

```
>>> re.search(r"<([\a-zA-Z][a-zA-Z0-9]*)[>]*>.*?</\1>",r"abc<h1>News</h1>xyz")
<_sre.SRE_Match object at 0x02EE30A0>
```

说明：

- (1) 可以多次引用组。例如：“([a-b])x\1x\1”匹配“axaxa”、“bxbxb”。
- (2) 如果引用的组没有有效的匹配，则引用到的内容为空。
- (3) 引用不能用于其自身。例如：“([a-c]\1)”是错误的。同样“\0”表示正则表

- 达式匹配本身，故只能用于替换操作中。
- (4) 引用不能用于字符集内部。例如：“(a)[\1b]”中的“\1”，被解释为八进制转码。
- (5) 向后引用会降低引擎的速度，因为它需要存储匹配的组。
- (6) 如果不需要向后引用，即对某个组不存储，可以使用组前缀“?:”。例如：“Get(?:Value)”。“(”后面紧跟的“?:”告诉引擎对于组(Value)不存储。
- (7) 当对组使用重复操作符时，缓存里引用内容会被不断刷新，只保留最后匹配的内容。例如：“([abc]+)=\1”将匹配“cab=cab”，但是“([abc])+=\1”不匹配“cab=cab”。因为([abc])第一次匹配c时，“\1”代表“c”；然后([abc])会继续匹配a和b。最后“\1”代表“b”，所以它会匹配“cab=b”。

3. 分组命名和引用

在 Python 中，对组可以进行命名并引用：

```
(?P<name>group) #组命名
(?P=name) #引用命名组
```

例如：

```
>>> m = re.search(r"(?P<Area>\d+) - (?P<No>\d+)", "电话号码:021 - 62232333")
>>> m.groupdict() #{'Area': '021', 'No': '62232333'}
```

4. 分组的扩展语法

分组支持扩展语法如表 14-5 所示。

表 14-5 分组扩展语法

| 符 号                                | 说 明                                                                                                                                         |
|------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| (?aiLmsux)                         | 应用于该组的匹配标志选项，参见第 14.2.2 节                                                                                                                   |
| (?:...)                            | 不存储组                                                                                                                                        |
| (?P<name>...)                      | 命名组                                                                                                                                         |
| (?P=name)                          | 引用命名组                                                                                                                                       |
| (?#...)                            | 注解。例如：“(?#comment) be”匹配 beg 中的 be                                                                                                          |
| (?=...)                            | 后置条件。例如：“be(=ing)”匹配 being 中的 be，但不匹配 been 中的 be                                                                                            |
| (?!...)                            | 后置非条件。例如：“be(!ing)”不匹配 being 中的 be，但匹配 been 中的 be                                                                                           |
| (?<=...)                           | 前置条件。例如：“(?<=pre)fix”匹配 prefix 中的 fix，但不匹配 suffix 中的 fix                                                                                    |
| (?<!...)                           | 前置非条件。例如：“(?<!pre)fix”不匹配 prefix 中的 fix，但匹配 suffix 中的 fix                                                                                   |
| (?(id/name)yes-pattern no-pattern) | 条件判别。如果组 id/name 存在，则 yes-pattern，否则 no-pattern。例如：(<)?(\w+@\w+(?:\.\w+)+)(?(1)>  \$)匹配 <user@host.com> 和 user@host.com，但不匹配 <user@host.com |

14.1.10 正则表达式的匹配模式

正则表达式引擎都支持不同的匹配模式，也称为匹配选项。例如，`/i` 使正则表达式对大小写不敏感；`/m` 开启多行模式，即 “`^`” 和 “`$`” 匹配新行符的前面和后面的位置。匹配模式可以通过分组扩展语法支持：（`? aiLmsux`）。也可通过匹配选项支持，参见 14.2.2 节。

14.1.11 常用正则表达式

常用的正则表达式如表 14-6 所示。

表 14-6 常用的正则表达式

| 用 途                | 正则表达式                                                           |
|--------------------|-----------------------------------------------------------------|
| Internet 电子邮件地址    | <code>^\w+([ -+.']\w+)*@\w+([ -.]\w+)*\.\w+([ -.]\w+)*\$</code> |
| 中华人民共和国电话号码        | <code>^(\(\d{3}\) \d{3}-)?\d{8}\$</code>                        |
| 中华人民共和国邮政编码        | <code>^\d{6}\$</code>                                           |
| Internet URL       | <code>^https?://\w+((?!\.)[^\.]+)?((?:/\.)+)*\$</code>          |
| 中华人民共和国身份证号码（ID 号） | <code>^\d{17}[\dX] \d{15}\$</code>                              |

14.2 正则表达式模块 re

14.2.1 匹配和搜索函数

`re` 模块提供了若干用于匹配和搜索的函数。

`re.match (pattern, string, flags = 0)`：若匹配，返回 `Match` 对象，否则返回 `None`。

`re.search (pattern, string, flags = 0)`：若匹配，返回 `Match` 对象，否则返回 `None`。

`re.findall (pattern, string, flags = 0)`：返回匹配结果列表；若 `pattern` 中包含组，返回组的列表。

`re.finditer (pattern, string, flags = 0)`：返回所有匹配结果 `Match` 对象迭代器。

其中，`pattern` 为匹配模式，`string` 为要匹配的字符串，`flags` 为匹配选项。

`match` 函数从字符串头部开始匹配，而 `search` 在字符串任何位置匹配。

```
>>> re.match("to", "To be, \nor not to be") #无匹配
>>> re.search("^to", "To be, \nor not to be") #无匹配
>>> re.search("to", "To be, \nor not to be") #匹配
<_sre.SRE_Match object at 0x02F02528 >
>>> re.findall("be", "To be, \nor not to be") #返回结果列表:['be','be']
>>> re.finditer("be", "To be, \nor not to be") #返回结果迭代器
< callable_iterator object at 0x02ED44D0 >
>>> for i in re.finditer("be", "To be, \nor not to be"): print(i, end = " ")
<_sre.SRE_Match object at 0x02ED8870 > <_sre.SRE_Match object at 0x02F02528 >
```



14.2.2 匹配选项

re 模块中包括表 14-7 所示的匹配选项。

表 14-7 re 模块的匹配选项

| 匹配选项                 | 说 明                                                                                                                                                                  |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| re. A、re. ASCII      | \w、\W、\b、\B、\d、\D、\s 以及\S，只匹配 ASCII 字符                                                                                                                               |
| re. I、re. IGNORECASE | 忽略大小写。例如：<br>>>> re. match("to","To be,\nor not to be") #无匹配<br>>>> re. match("to","To be,\nor not to be",re. I) #匹配<br><_sre. SRE_Match object at 0x02ED8870 >      |
| re. L、re. LOCALE     | \w、\W、\b、\B、\s 以及\S，与本地字符集有关                                                                                                                                         |
| re. M、re. MULTILINE  | 多行匹配模式。例如：<br>>>> re. search("^or","To be,\nor not to be") #无匹配<br>>>> re. search("^or","To be,\nor not to be",re. M) #匹配<br><_sre. SRE_Match object at 0x02ED8870 > |
| re. S、re. DOTALL     | 匹配所有字符，包括换行符                                                                                                                                                         |
| re. X、re. VERBOSE    | 忽略空格并可使用#注释，提高可读性。例如：<br>b = re.compile(r"\d+\.\d*") 等价于<br>a = re.compile(r"""\d +      #整数部分<br>\.      #小数点<br>\d *    #小数部分""", re. X)                           |
| re. DEBUG            | 输出调试信息                                                                                                                                                               |

14.2.3 正则表达式对象

使用 re.compile 函数，可以将正则表达式编译为正则表达式对象 regex，然后使用其对象方法处理字符串。

- regex = re.compile(pattern, flags = 0)：编译模式，生成正则表达式对象。
- regex.search(string[, pos[, endpos]])：若匹配，返回 Match 对象，否则返回 None。
- regex.match(string[, pos[, endpos]])：若匹配，返回 Match 对象，否则返回 None。
- regex.findall(string[, pos[, endpos]])：返回匹配结果列表；若 pattern 中包含组，返回组的列表。
- regex.finditer(string[, pos[, endpos]])：返回所有匹配结果 Match 对象迭代器。

其中，pattern 为匹配模式；string 为要匹配的字符串；flags 为匹配选项。pos 和 endpos 为搜索范围，从 pos 到 endpos - 1。

正则表达式对象方法 search、match、findall 和 finditer 与 re 模块中的对应函数基本一致。例如：

```
>>> regex1 = re.compile('to')
>>> regex2 = re.compile('^to')
>>> regex1.match("To be,\nor not to be") #无匹配
>>> regex2.search("To be,\nor not to be") #无匹配
>>> regex1.search("To be,\nor not to be") #匹配
<_sre. SRE_Match object at 0x02ED8870 >
```



```
>>> regex1.findall("To be,\nor not to be") #返回结果列表:['to']
>>> regex1.finditer("To be,\nor not to be") #返回结果迭代器
< callable_iterator object at 0x02ED4870 >
```

## 14.2.4 匹配对象

使用 re 中的函数 search 和 match，或正则表达式对象方法 search 和 match，返回的结果为匹配对象（Match object）。使用匹配对象的方法，可以进行匹配结果处理。

m.group([group1, ...]): 返回匹配的 1 个或多个组。  
m.groups(default = None): 返回匹配的所有子组，结果为元组。  
m.groupdict(default = None): 返回匹配的所有命名组，结果为字典。  
m.start([group]): 返回匹配的组的开始位置。  
m.end([group]): 返回匹配的组的结束位置。  
m.span([group]): 返回匹配的组的位置范围：(m.start(group), m.end(group))。

例如：

```
>>> m = re.search("be", "To be,\nor not to be")
>>> m.group() # 'be'
>>> m.span() # (3,5)
```

再如：

```
>>> m = re.search(r"(?P<Area>\d+) - (?P<No>\d+)", "电话号码:021 - 62232333")
>>> m.group(), m.group(0), m.group(1), m.group(2)
('021 - 62232333', '021 - 62232333', '021', '62232333')
>>> m.groups() # ('021', '62232333')
>>> m.groupdict() # {'Area': '021', 'No': '62232333'}
```

## 14.2.5 匹配和替换

re 中的函数 sub 和 subn，或正则表达式对象方法 sub 和 subn，使用正则表达式匹配字符串，用指定内容替换结果，并返回替换后的字符串。

re.sub(pattern, repl, string, count = 0, flags = 0): 返回替换后的字符串。

re.subn(pattern, repl, string, count = 0, flags = 0): 返回元组：（替换后的字符串，替换次数）。

regex.sub(repl, string, count = 0): 同 re.sub。

regex.subn(repl, string, count = 0): 同 re.subn。

其中，pattern 为匹配模式，string 为要匹配和替换的字符串，repl 为要替换的内容，count 为替换的最大次数，flags 为匹配选项。例如：

```
>>> re.sub('bad', 'good', 'It tastes bad. ') # 'It tastes good. '
```

例如，当编辑文字时，很容易就会输入重复单词，例如“thethe”。使用正则表达式 \b(\w+)\s+\1\b 可以检测到这些重复单词。要删除第二个单词，只要简单地利用替换功能替换掉“\1”就可以了。

## 14.2.6 分割字符串

re 中的函数 split，或正则表达式对象方法 split，使用正则表达式匹配字符串（匹配分隔

符)，并分割字符串，返回分割后的字符串列表。

`re.split(pattern, string, maxsplit=0, flags=0)`：返回分割后的字符串列表。

`regex.split(string, maxsplit=0)`：同 `re.split`。

其中，`pattern` 为匹配模式，`string` 为要匹配和分割的字符串，`maxsplit` 为分割的最大次数，`flags` 为匹配选项。例如：

```
>>> re.split('\W+', 'Good, better, best! ') # \W + 1 个以上非单词字符: ['Good', 'better', 'best', '']
>>> re.split('\W+', 'Good, better, best! ', 1) # 分割 1 次: ['Good', 'better, best! ']
>>> re.split('(\W+)', 'Good, better, best! ') # 使用分组时, 同时返回分割字符
['Good', ',', 'better', ',', 'best', '!', '']
>>> re.split('\d', '1a2b3c4d') # 分割符为数字字符: ['', 'a', 'b', 'c', 'd']
```

## 14.3 正则表达式应用举例

### 14.3.1 字符串包含验证

通过 `re` 模块的匹配和搜索，或者正则表达式对象的匹配和搜索，可以验证一个字符串是否与指定模式匹配，即实现字符串包含验证。

**【例 14-1】** 验证一个字符串是否为有效的电子邮件格式（`emailaddress_check.py`）。

```
import os, re
def check_email(strEmail):
 regex_email = re.compile(r'^[\w\.-]+@([\w\.-]+\.)+[\w\.-]+.$')
 result = True if regex_email.match(strEmail) else False
 return result
#测试代码
if __name__ == '__main__':
 str1 = "hjiang@yahoo.com" #有效的电子邮箱
 str2 = "hjiang.yahoo.com" #无效的电子邮箱
 print(str1, '是有效的电子邮件格式吗?', check_email(str1))
 print(str2, '是有效的电子邮件格式吗?', check_email(str2))
```

运行结果如下：

```
hjiang@yahoo.com 是有效的电子邮件格式吗? True
hjiang.yahoo.com 是有效的电子邮件格式吗? False
```

### 14.3.2 字符串查找和拆分

使用 `re` 中的函数 `split`，或正则表达式对象方法 `split` 可以分割字符串；也可以通过 `re` 中的函数 `findall`，或正则表达式对象方法 `findall` 分割字符串，因为如果匹配中包含组，`findall` 返回组的列表。

**【例 14-2】** 根据给定正则表达式的匹配拆分字符串示例（`tasklist.py`）。例子中使用了 `os.popen` 执行操作系统命令并返回管道，可参见 19.2.2 节。

```
import os, re
```

```
def tasklist():
 #tasklist /nh
 #System Idle Process 0 Services 0 20 K
 regex_task = re.compile(r'([\w.] + (?:[\w.] +) *) \s\s + (\d +) \w + \s\s + \d + \s\s + ([\d,] + K)')
 with os.popen('tasklist /nh','r') as f:
 for line in f:
 print(regex_task.findall(line.strip()))
if __name__ == '__main__':
 tasklist()
```

运行结果如下：

```
[]
[('System Idle Process','0','20 K')]
[('System','4','1,452 K')]
[('smss.exe','336','1,112 K')]
...(略)
```

### 14.3.3 字符串替换和清除

使用 re 中的函数 sub 或正则表达式对象方法 sub，可以实现字符串替换。例如把 html 文件的所有大写 HTML 标记替换成小写标记。

**【例 14-3】** 从输入字符串中清除 HTML 标记 (html\_txt.py)。

```
import re
def html_txt(htmlwithtag):
 regex_href = re.compile(r'<. +? >')
 return regex_href.sub("",htmlwithtag) #替换为空",并返回替换结果
#测试代码
if __name__ == '__main__':
 htmltext = r' Welcome to Python world! '
 print(html_txt(htmltext))
```

运行结果如下：

```
Welcome to Python world!
```

### 14.3.4 字符串查找和截取

通过正则表达式对象的 findall() / finditer() 方法，可以查找与该模式匹配的结果列表。例如，从网页中查找所有的 URL。

**【例 14-4】** 从指定的网页中查找所有的超链接地址 (url\_extract.py)。例中使用 urllib.request.urlopen 打开网页，请参见 18.3 节。

```
import re,urllib.request
def url_extract(homepage):
 regex_href = re.compile(r'href = "(. + ?)"')
```

```
f = urllib.request.urlopen(homepage)
webcontents = f.read().decode()
matches = regex_href.finditer(webcontents)
for m in matches:
 print(m.group(1))
#测试代码
if __name__ == '__main__':
 www = r'http://www.baidu.com'
 url_extract(www)
```

运行结果如下：

```
http://www.baidu.com/gaoji/preferences.html
https://passport.baidu.com/v2/?login&tpl=mn&u=http%3A%2F%2Fwww.baidu.com%2F
...(略)
```

## 14.4 复习题

### 一、填空题

1. Python 语句 `re.match('back','text.back')` 的执行结果是\_\_\_\_\_。
2. Python 语句 `re.findall("to","Tom likes to swim too")` 的执行结果是\_\_\_\_\_。
3. Python 语句 `re.findall("bo[xy]","The boy is sitting on the box")` 的执行结果是\_\_\_\_\_。
4. 中华人民共和国邮政编码由 6 位数字组成，使用重复限定符\_\_\_\_\_，表示数字字母重复 6 次。
5. 正则表达式引擎均支持不同的匹配模式，也称为匹配选项，其中，\_\_\_\_\_使正则表达式对大小写不敏感，\_\_\_\_\_开启多行模式。

6. Python 语句 `re.sub('hard','easy','Python is hard to learn.')` 的执行结果是\_\_\_\_\_。
7. Python 语句 `re.split('\W+', 'go, went, gone')` 的执行结果是\_\_\_\_\_。
8. Python 语句 `re.split('\d', 'a1b2c3')` 的执行结果是\_\_\_\_\_。

### 二、思考题

1. 用 Python 匹配 HTML 标签时，`<.*>` 和 `<.*?>` 有什么区别？
2. 请问 Python 中的 `search()` 和 `match()` 有什么区别？
3. 如何使用 Python 查询和替换一个文本字符串？
4. 正则表达式包括哪些元字符？
5. 下列 Python 语句的输出结果是\_\_\_\_\_。

```
import re;sum=0;pattern='boy'
if re.match(pattern,'boy and girl'):sum+=1
if re.match(pattern,'girl and boy'):sum+=2
if re.search(pattern,'boy and girl'):sum+=3
if re.search(pattern,'girl and boy'):sum+=4
print(sum)
```

6. 下列 Python 语句的输出结果是\_\_\_\_\_。



```
import re;re.match("to","Tom likes to swim too")
re.search("to","Tom likes to swim too");re.findall("to","Tom likes to swim too")
```

7. 下列 Python 语句的输出结果是\_\_\_\_\_。

```
import re;m=re.search("to","Tom likes to swim too")
print(m.group(),m.span())
```

# 14.5 上机实践

1. 编写程序，分别输入三个字符串，依次验证其是否为有效的中华人民共和国电话号码、中华人民共和国邮政编码和网站网址格式，运行效果如图 14-1 所示。

提示：

- (1) 中华人民共和国电话号码（电话号码必须是 8 位号码，如果有区号，区号必须 3 位）的正则表达式为：`^(\\(\\d{3}\\)|\\d{3}-)?\\d{8}$`。
- (2) 中华人民共和国邮政编码（必须 6 位数字）的正则表达式为：`^\\d{6}$`。
- (3) 网站网址（Internet URL）的正则表达式为：`^https?:\\/\\w+(?:\\.[^\\.]*)+(?:\\/.*+)?*`。

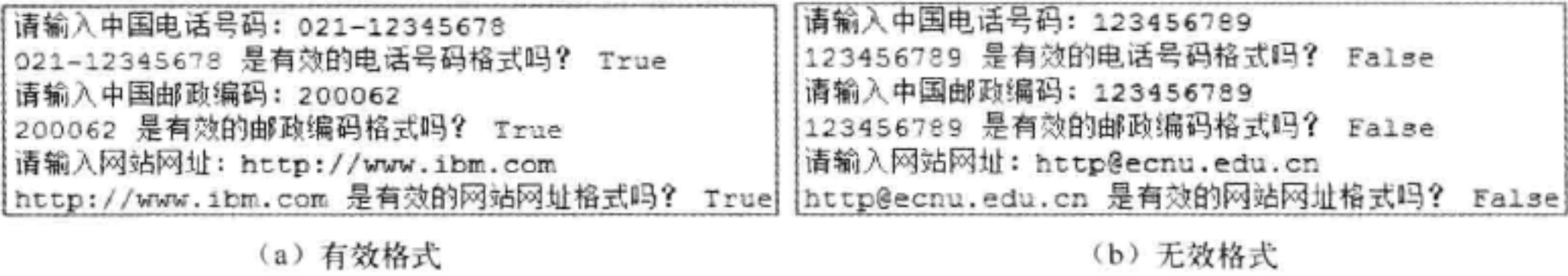


图 14-1 验证有效格式运行效果

- 2. 参照例 14-2 编写根据给定正则表达式的匹配拆分字符串程序。使用 `os.popen` 执行操作系统命令并返回管道。
- 3. 参照例 14-3 编写从输入字符串中清除 HTML 标记程序。
- 4. 参照例 14-4 编写从指定的网页中查找所有的超链接地址程序。使用 `urllib.request.urlopen` 打开网页。

## 第 15 章 多线程编程

线程能够执行并发处理，即同时执行多个操作。例如，使用线程处理来同时监视用户输入，并执行后台任务，以及处理并发输入流。

### 本章要点：

---

- ◆ 线程处理的基本概念；
  - ◆ 创建和启动线程；
  - ◆ 线程同步处理。
- 

### 15.1 线程处理概述

#### 15.1.1 进程和线程

进程是操作系统中正在执行的不同应用程序的一个实例，操作系统把不同的进程分离开来。每一个进程都有自己的地址空间，一般情况下，包括文本区域、数据区域和堆栈。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

线程是进程中的一个实体，是被操作系统独立调度和分派处理器时间的基本单位。线程一般由线程 ID、当前指令指针、寄存器集合和堆栈，即一组用于调度的该线程上下文的结构；线程与同属一个进程的其他线程共享进程所拥有的全部资源。线程又称为轻量级进程（Light Weight Process, LWP）。

支持抢先多任务处理的操作系统可以实现多个进程中的多个线程同时执行的效果：在需要处理器时间的线程之间分割可用处理器时间，并轮流为每个线程分配处理器时间片（时间片的长度取决于操作系统和处理器数目），由于每个时间片都很小，因此多个线程看起来似乎在同时执行。

#### 15.1.2 线程的优缺点

线程使程序能够执行并发处理，因而特别适合需要同时执行多个操作的场合。例如，使用一个线程来执行复杂的后台计算任务，使用另一个线程来监视用户输入，以提高系统的用户响应性能；使用高优先级线程管理时间关键的任务，使用低优先级线程执行其他任务，以区分具有不同优先级的任务；为服务器应用程序创建包含多个线程的线程池，以及处理并发的客户端请求。

多线程处理可解决用户响应性能和多任务的问题，但同时引入了资源共享和同步等问

题。例如：过多的线程将占用大量的资源和处理器调度时间，从而影响运行性能；为了避免对共享资源的访问冲突，必须对共享资源进行同步或控制处理，因而有可能导致死锁；使用多线程控制代码执行非常复杂，并可能产生许多错误。

## 15.2 创建和启动多线程

### 15.2.1 使用 `start_new_thread` 函数创建线程

`_thread` 模块包含创建线程的函数：

`_thread.start_new_thread(function, args[, kwargs])` #创建一个新线程并返回其标识符

其中，`function` 是线程运行的函数；`args` 是函数的参数，必须为元组；可选的 `kwargs` 是命名参数字典。

`start_new_thread` 创建一个线程并运行指定函数，当函数返回时，线程自动结束。也可以通过 `_thread.exit()` 结束线程。

【例 15-1】使用 `_thread` 模块的 `start_new_thread` 函数创建线程（`startnewthread.py`）。

```
import _thread, time, random
def timer(interval):
 for i in range(5):
 time.sleep(random.choice(range(interval))) #随机睡眠 0 - interval 秒
 thread_id = _thread.get_ident() #获取当前线程标识符
 print('Thread: {0} Time: {1} \n'.format(thread_id, time.ctime()))
def test(): #使用 start_new_thread() 函数创建 2 个线程
 _thread.start_new_thread(timer, (5,)) #创建线程
 _thread.start_new_thread(timer, (5,)) #创建线程
if __name__ == '__main__':
 test()
```

运行结果如下：

```
Thread:16716 Time:Thu Sep 5 16:08:10 2013
Thread:16172 Time:Thu Sep 5 16:08:10 2013
Thread:16172 Time:Thu Sep 5 16:08:12 2013
Thread:16716 Time:Thu Sep 5 16:08:14 2013
Thread:16716 Time:Thu Sep 5 16:08:14 2013
Thread:16172 Time:Thu Sep 5 16:08:15 2013
Thread:16716 Time:Thu Sep 5 16:08:15 2013
Thread:16172 Time:Thu Sep 5 16:08:16 2013
Thread:16716 Time:Thu Sep 5 16:08:19 2013
Thread:16172 Time:Thu Sep 5 16:08:19 2013
```

### 15.2.2 使用 `Thread` 对象创建线程

`threading` 模块封装了 `_thread` 模块，并提供更多功能。虽然可以使用 `_thread` 模块中的 `start_new_thread()` 函数创建线程，但一般建议使用 `threading` 模块。



通过创建 Thread 的对象，可创建线程：

**Thread( target = None, name = None, args = (), kwargs = {} )** #构造方法

其中，target 是线程运行的函数；name 是线程的名称；args 和 kwargs 是传递给 target 的参数元组和命名参数字典。

通过调用 Thread 对象的 start 方法，可启动线程。

t.start()：启动线程。

t.is\_alive()：判断线程是否活动。

t.name：属性，线程名。对应于老版本的方法 getname() 和 setname()。

t.id：返回线程标识符。

threading 模块包含以下若干实用函数。

threading.get\_ident()：返回当前线程的标识符。

threading.current\_thread()：返回当前线程。

threading.active\_count()：返回活动的线程数目。

threading.enumerate()：返回活动线程的列表。

**【例 15-2】** 直接使用 Thread 对象创建和启动新线程（Thread.py）。

```
import threading,time,random
def timer(interval):
 for i in range(5):
 time.sleep(random.choice(range(interval))) #随机睡眠 0 - interval 秒
 thread_id = threading.get_ident() #获取当前线程标识符
 print('Thread: {0} Time: {1} \n'.format(thread_id,time.ctime()))
def test():#Use thread.start_new_thread() to create 2 new threads
 t1 = threading.Thread(target = timer,args = (5,)) #创建线程
 t2 = threading.Thread(target = timer,args = (5,)) #创建线程
 t1.start();t2.start() #启动线程
if __name__ == '__main__':
 test()
```

### 15.2.3 自定义派生于 Thread 的对象

通过声明 Thread 的派生类，并重写对象的 run 方法，然后创建其对象实例，可创建线程。通过对象的 start 方法，可启动线程，并自动执行对象的 run 方法。

**【例 15-3】** 通过声明 Thread 派生类，以创建和启动新线程（MyThread.py）。

```
import threading,time,random
class MyThread(threading.Thread): #继承 threading.Thread
 def __init__(self,interval): #构造函数
 threading.Thread.__init__(self) #调用父类构造函数
 self.interval = interval #对象属性
 def run(self): #定义 run 方法
 for i in range(5):
 time.sleep(random.choice(range(self.interval))) #随机睡眠 0 - interval 秒
 thread_id = threading.get_ident() #获取当前线程标识符
```



```

 print('Thread:{0} Time:{1}\n'.format(thread_id,time.ctime()))
if __name__ == '__main__':
 t1 = MyThread(5) #创建对象
 t2 = MyThread(5) #创建对象
 t1.start(); t2.start() #启动线程

```

### 15.2.4 线程加入 join()

所谓线程加入 (t.join()), 即让包含代码的线程 (tc, 即当前线程) “加入” 到另外一个线程 (t) 的尾部。在线程 (t) 执行完毕之前, 线程 (tc) 不能执行。

**join(timeout=None)**

其中, timeout 是超时参数, 单位为秒。如果指定了超时, 则线程 t 执行完毕或超时都可能使当前线程继续, 此时可通过 t.is\_alive() 来判断线程是否终止。

注意, 线程不能加入自己, 否则导致 RuntimeError, 因为这样将导致死锁。线程也不能加入未启动的线程, 否则导致 RuntimeError。

**【例 15-4】** 线程 join 示例 (join.py)。

```

import threading,time,random
class MyThread(threading.Thread): #继承 threading.Thread
 def __init__(self): #构造函数
 threading.Thread.__init__(self) #调用父类构造函数
 def run(self): #定义 run 方法
 for i in range(5):
 time.sleep(1) #睡眠 1 秒
 t = threading.current_thread() #获取当前线程
 print('{0} at {1}\n'.format(t.name,time.ctime())) #打印线程名、当前时间
 print('线程 t1 结束')
def test():
 t1 = MyThread() #创建线程对象
 t1.name = 't1' #设置线程名称
 t1.start() #启动线程
 print('主线程开始等待线程(t1)2s');t1.join(2)
 print('主线程等待线程(t1)2s 结束')
 print('主线程开始等待线程结束');t1.join()
 print('主线程结束')
if __name__ == '__main__':
 test()

```

运行结果如下:

```

主线程开始等待线程(t1)2s
t1 at Tue Sep 10 13:57:52 2013
t1 at Tue Sep 10 13:57:53 2013
主线程等待线程(t1)2s 结束
主线程开始等待线程结束

```

```
t1 at Tue Sep 10 13:57:54 2013
t1 at Tue Sep 10 13:57:55 2013
t1 at Tue Sep 10 13:57:56 2013
线程 t1 结束
主线程结束
```

15.2.5 用户线程和 daemon 线程

线程可分为用户线程和 daemon 线程。

用户线程（非 daemon 线程）是通常意义的线程，应用程序运行即为主线程，在主线程中可以创建和启动新线程，默认为用户线程。只有当所有的非 daemon 的用户线程（包括主线程）结束后，应用程序终止。

如果在主线程中创建新线程时，设置其对象属性 daemon 为 True，则该线程为 daemon 线程。

**t.daemon** #属性。对应于老版本的方法 isDaemon() 和 setDaemon()

默认为 False，即非 daemon 线程；如果设置为 True，则为 daemon 线程。必须在线程启动之前调用，否则导致 RuntimeError，即已启动的线程不能改变其 daemon 属性。

daemon 线程，又称守护线程，其优先级是最低的，一般为其他的线程提供服务。通常，daemon 线程体是一个无限循环。如果所有的非 daemon 线程都结束了，则 daemon 线程自动就会终止。

【例 15-5】 用户线程和 Daemon 线程示例（daemon.py）。运行结果如图 15-1 所示。

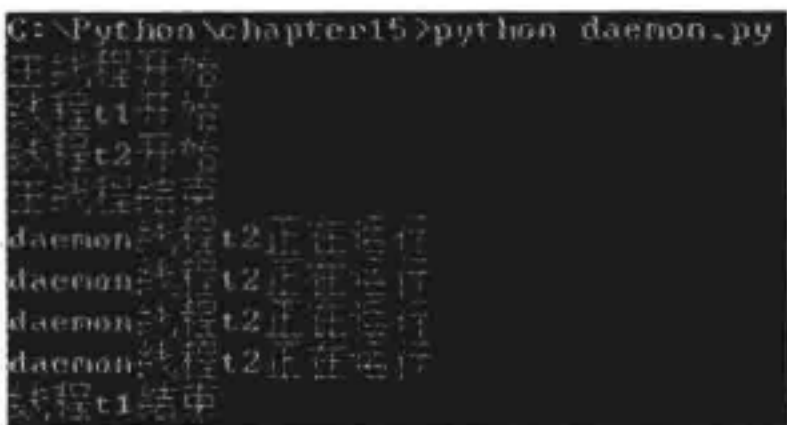


图 15-1 用户线程和 Daemon 线程运行结果

```
import threading,time
class MyThread(threading. Thread):
 def __init__(self,interval):
 threading. Thread. __init__(self)
 self. interval = interval
 def run(self):
 t = threading. current_thread()
 print('线程' + t. name + '开始')
 time. sleep(self. interval)
 print('线程' + t. name + '结束')
class MyThreadDaemon(threading. Thread):
 def __init__(self,interval):
 threading. Thread. __init__(self)
```

```
 self.interval = interval #对象属性
 def run(self): #定义 run 方法
 t = threading.current_thread() #获取当前线程
 print('线程' + t.name + '开始')
 while True:
 time.sleep(self.interval) #延迟 self.interval 秒
 print('daemon 线程' + t.name + '正在运行')
 print('线程' + t.name + '结束')
def test():
 print('主线程开始')
 t1 = MyThread(5) #创建线程对象
 t2 = MyThreadDaemon(1) #创建线程对象
 t1.name = 't1'; t2.name = 't2' #设置线程名称
 t2.daemon = True #设置为 daemon
 t1.start() #启动线程
 t2.start()
 print('主线程结束')
if __name__ == '__main__':
 test()
```

## 15.3 线程同步

### 15.3.1 线程同步处理

当多个线程调用单个对象的属性和方法时，一个线程可能会中断另一个线程正在执行的任务，使该对象处于一种无效状态，因此必须针对这些调用进行同步处理。

### 15.3.2 基于原语锁（Lock/RLock 对象）的简单同步

原语锁是同步原语，使用 threading 模块的 Lock 对象锁可以实现线程的简单同步。threading 模块的 RLock 是可重入的同步锁。

Lock 对象锁有两个状态：locked 和 unlocked（初始状态）。

线程可以使用 Lock 对象锁的 acquire() 方法获得锁，则锁进入 locked 状态。每次只有一个线程可以获得锁。如果当另一个线程试图使用 acquire() 方法获得 locked 状态的锁时，就会被系统变为 blocked 状态，直到那个拥有锁的线程调用锁的 release() 方法来释放锁，这样锁就会进入 unlocked 状态。blocked 状态的线程就会收到一个通知，并有权利获得锁。如果多个线程处于 blocked 状态，所有线程都会先解除 blocked 状态，然后系统选择一个线程来获得锁（具体是哪个线程获得锁与实现有关），其他的线程继续沉默（blocked）。

一般要实现线程同步的关键代码放置在 acquire() 和 release() 方法之间，即：

```
import threading
lock = threading.Lock()
lock.acquire()
```

...

`lock.release()`

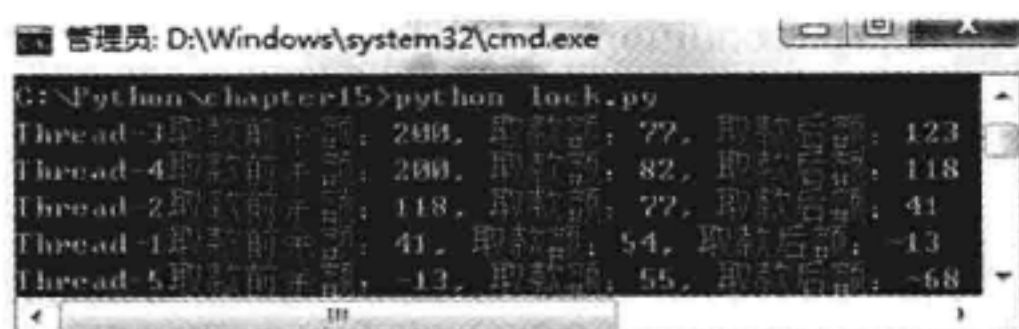
Lock 对象支持 with 语句:

`with some_lock:``# do something...`

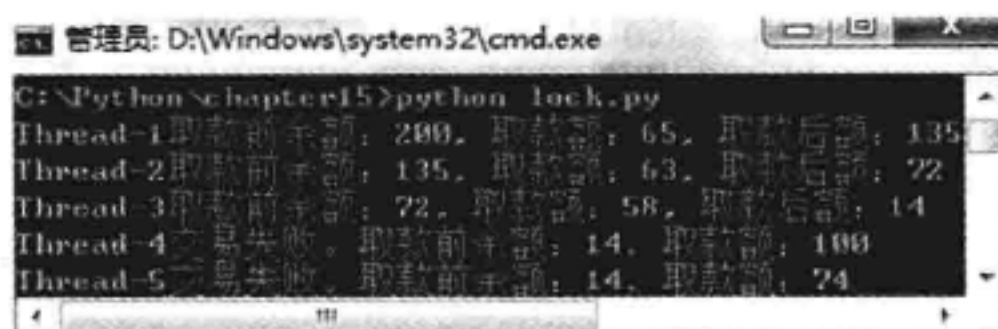
等价于:

`some_lock.acquire()``try:``# do something...``finally:``some_lock.release()`

【例 15-6】使用 lock 语句同步代码块示例 (lock.py)。创建工作线程，模拟银行现金账户取款。多个线程同时执行取款操作时，如果不使用同步处理 (图 15-2 (a))，会造成账户余额混乱；尝试使用同步锁对象 Lock，以保证多个线程同时执行取款操作时，银行现金账户取款的有效和一致 (图 15-2 (b))。



(a) 不使用同步锁，交易混乱



(b) 使用同步锁，交易正常

图 15-2 同步锁运行效果

```
import threading,time,random
class Account(threading.Thread):
 lock = threading.Lock()
 def __init__(self,amount):
 threading.Thread.__init__(self)
 Account.amount = amount
 def run(self):
 self.withdraw()
 def withdraw(self):
 Account.lock.acquire()
 t = threading.current_thread()
 a = random.choice(range(50,101))
 if Account.amount < a:
 print('{0} 交易失败。取款前余额:{1},取款额:{2}'.format(t.name,Account.amount,a))
 Account.lock.release()
 return 0
 time.sleep(random.choice(range(5)))
 prev = Account.amount
 Account.amount -= a
```

#继承 threading.Thread  
#创建锁  
#构造函数  
#调用父类构造函数  
#账户金额  
#定义 run 方法  
#取款  
#获取锁。注释不使用同步处理  
#拒绝交易  
#随机睡眠[0-5)秒  
#取款



```

 print ('|0| 取款前余额: {1}, 取款额: {2}, 取款后额: {3}'.format (t.name, prev, a,
Account.amount))
 Account.lock.release() #释放锁。注释不使用同步处理
def test():
 for i in range(5): #创建 10 个线程对象并启动
 Account(200).start()
if __name__ == '__main__':
 test()

```

### 15.3.3 基于条件变量（Condition 对象）的同步和通信

使用 Lock 对象 lock 同步代码块时，假设线程 A 获得同步锁 lock，在执行同步代码时，需要等待某资源，而该资源由线程 B 提供资源。此时，线程 B 无法获取线程 A 占用的同步锁 lock，故无法提供线程 A 需要的资源，因而会导致死锁。

为了解决死锁的问题，可以使用基于条件变量（Condition）的线程间通信的通信机制：wait()、notify() 和 notifyAll()。

条件变量（Condition）对象关联一个锁 lock。创建 Condition 对象时，可以传入一个参数 lock，如果没有传入该参数，则自动创建一个。

```
cv = threading.Condition(lock=None)
```

Condition 对象 cv 的 acquire() 和 release() 调用其关联的锁 lock，cv 还支持 with 语句：

```
with cv:
 同步操作
```

Condition 对象 cv 还包括对象方法 wait()、notify() 和 notifyAll()。wait()/wait(timeout) 释放锁 lock，并阻塞当前线程允许，直至其他线程使用 notify() 和 notifyAll() 唤醒后，重新获得锁 lock。notify() 唤醒一个等待线程，notifyAll() 唤醒所有等待线程。

使用 Condition 对象 cv 进行线程通信避免死锁的原理可以简单概述为：线程 A 获得同步锁 lock，在执行同步代码时，如果需要等待线程 B 占用的某资源，则可以调用 wait()/wait(毫秒数) 方法，阻塞当前线程的运行，并释放其占用的同步锁 lock；然后调用 notify() 方法，通知等待同步锁 lock 的线程 B。线程 B 获得同步锁 lock 后，执行操作，释放 A 所需要的资源，然后释放同步锁，并调用 notify() 方法，通知等待同步锁 lock 的线程 A。线程 A 获得同步锁 lock，继续执行。

典型的生产者 - 消费者模型，可使用线程间通信，保证生产者线程生产一件，消费者线程消费一件，二者保持同步。其基本代码片段为：

```

#生产者代码片段
with cv:
 while not an_item_is_available():
 cv.wait()
 get_an_available_item()
#消费者代码片段
with cv:
 make_an_item_available()

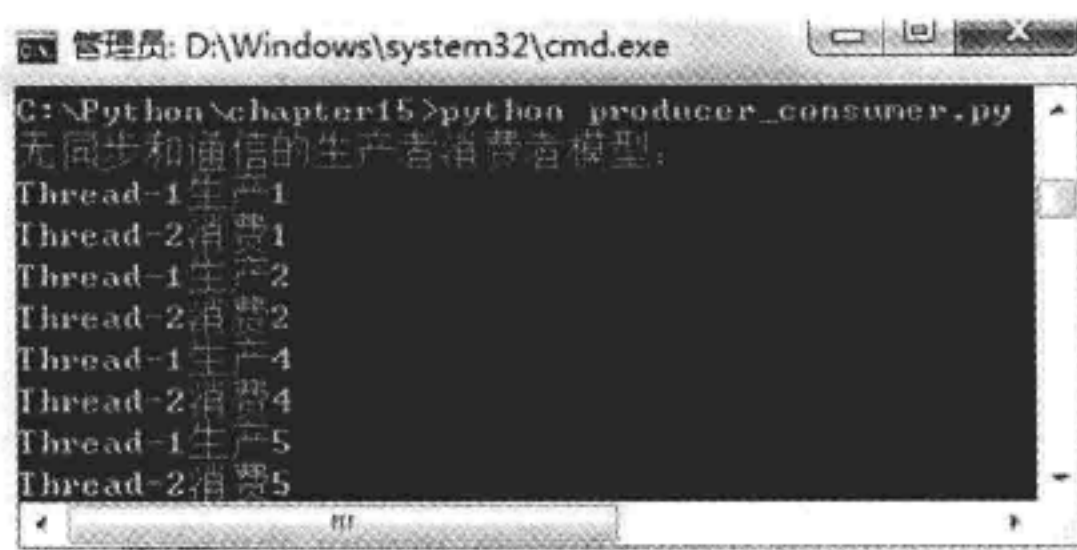
```

cv.notify()

【例15-7】线程间通信示例 (producer\_consumer.py)。生产者-消费者模型，使用线程间通信，生产者生产一件、消费者消费一件，二者保持同步 (见图15-3 (a))。未使用线程同步 (把最后1行代码改为test2())，则结果无法预料 (见图15-3 (b))。



(a) 使用同步锁，交易正常



(b) 不使用同步锁，交易混乱

图15-3 线程间通信运行效果

```
import threading,time,random
class Container1():
 def __init__(self):
 self.contents = 0
 self.available = False
 self.cv = threading.Condition()
 def put(self,value):
 with self.cv:
 if self.available:
 self.cv.wait()
 self.contents = value
 t = threading.current_thread()
 print('{0} 生产{1}'.format(t.name,self.contents))
 self.available = True
 self.cv.notify()
 def get(self):
 with self.cv:
 if not self.available:
 self.cv.wait()
 t = threading.current_thread()
 print('{0} 消费{1}'.format(t.name,self.contents))
 self.available = False
 self.cv.notify()
class Container2():
 def __init__(self):
 self.contents = 0
 self.available = False
 def put(self,value):
 #基于同步和通信
 #构造函数
 #容器内容
 #容器内容
 #条件变量
 #生产函数
 #使用条件变量同步
 #如果已经生产,则等待
 #等待
 #生产,设置内容
 #设置容器状态:已生产
 #通知等待的消费者
 #消费函数
 #使用条件变量同步
 #如果已经生产,则等待
 #等待
 #设置容器状态:未生产
 #通知等待的生产者
 #无同步和通信
 #构造函数
 #容器内容
 #容器内容
 #生产函数
```

```

 if self. available; #如果已经生产
 pass
 else;
 self. contents = value #生产,设置内容
 t = threading. current_thread()
 print(' {0} 生产 {1}'. format(t. name, self. contents))
 self. available = True #设置容器状态:已生产
def get(self); #消费函数
 if not self. available; #如果已经生产,则等待
 pass
 else;
 t = threading. current_thread()
 print(' {0} 消费 {1}'. format(t. name, self. contents))
 self. available = False #设置容器状态:未生产
class Producer(threading. Thread); #生产者类
 def __init__(self, container); #构造函数
 threading. Thread. __init__(self) #调用父类构造函数
 self. container = container #容器
 def run(self); #定义 run 方法
 for i in range(1,6);
 time. sleep(random. choice(range(5)))#随机睡眠[0-5)秒
 self. container. put(i) #生产
class Consumer(threading. Thread); #消费者类
 def __init__(self, container); #构造函数
 threading. Thread. __init__(self) #调用父类构造函数
 self. container = container #容器
 def run(self); #定义 run 方法
 for i in range(1,6);
 time. sleep(random. choice(range(5)))#随机睡眠[0-5)秒
 self. container. get() #消费
def test1();
 print(' 基本同步和通信的生产者消费者模型:')
 container = Container1() #创建容器
 Producer(container). start() #创建消费者线程并启动
 Consumer(container). start() #创建消费者线程并启动
def test2();
 print(' 无同步和通信的生产者消费者模型:')
 container = Container2() #创建容器
 Producer (container). start() #创建消费者线程并启动
 Consumer(container). start() #创建消费者线程并启动
if __name__ == '__main__':
 test1()

```

## 15.4 复习题

1. Python 可使用\_\_\_\_\_创建一个线程并运行指定函数，当函数返回时，线程自动结束。也可以通过\_\_\_\_\_结束线程。
2. Python 可通过声明 Thread 的派生类，并重写对象的\_\_\_\_\_方法，然后创建其对象实例，创建线程。通过对象的\_\_\_\_\_方法，可启动线程，并自动执行对象的 run 方法。
3. 线程可分为\_\_\_\_\_和\_\_\_\_\_。
4. \_\_\_\_\_，又称守护线程，其优先级是最低的，一般为其他的线程提供服务。
5. Lock 对象锁有两个状态：\_\_\_\_\_和\_\_\_\_\_。

## 15.5 上机实践

1. 参照例 15-1 编写使用\_thread 模块的 start\_new\_thread 函数创建线程的程序。
2. 参照例 15-2 编写直接使用 Thread 对象创建和启动新线程的程序。
3. 参照例 15-3 编写通过声明 Thread 派生类创建和启动新线程的程序。
4. 参照例 15-4 编写线程 join 示例程序。
5. 参照例 15-5 编写用户线程和 Daemon 线程示例程序。
6. 参照例 15-6 编写使用 lock 语句同步代码块的程序。创建工作线程，模拟银行现金账户取款。多个线程同时执行取款操作时，如果不使用同步处理，会造成账户余额混乱；尝试使用同步锁对象 Lock，以保证多个线程同时执行取款操作时，银行现金账户取款的有效和一致。
7. 参照例 15-7 编写线程间通信的程序。生产者 - 消费者模型，使用线程间通信，生产者生产一件、消费者消费一件，二者保持同步。未使用线程同步，则结果无法预料的程序。



## 第 16 章 图形用户界面应用程序

相对于字符界面的控制台应用程序，基于图形化用户界面（Graphic User Interface, GUI）的应用程序可提供丰富的用户交互界面，从而实现各种复杂功能的应用程序。

### 本章要点：

---

- ◆ 图形用户界面概述；
  - ◆ tkinter 概述；
  - ◆ 几何布局管理器；
  - ◆ tkinter 事件处理；
  - ◆ 常用 tkinter 组件；
  - ◆ 对话框；
  - ◆ 菜单和工具栏；
  - ◆ 图形绘制。
- 

## 16.1 图形用户界面概述

### 16.1.1 tkinter

tkinter（Tk interface，tk 接口），是 Tk 图形用户界面工具包标准的 Python 接口。tkinter 是 Python 的标准 GUI 库，支持跨平台的图形用户界面应用程序开发，包括 Windows、Linux、Unix 和 Macintosh 操作系统。

tkinter 的特点是简单实用。tkinter 是 Python 语言的标准库之一，Python 自带的 IDLE 就是采用它开发的。tkinter 开发的图形界面，其显示风格是本地化的。

tkinter 适用于小型图形界面应用程序的快速开发。本书主要基于 tkinter，阐述图形用户界面应用程序开发的主要流程。

### 16.1.2 其他 GUI 库简介

#### 1. pyGtk

Gtk 是 LINUX 下 Gnome 的核心 GUI 开发库，功能齐全。pyGtk 模块是 Gnome 图形用户界面工具包标准的 Python 接口。glade 界面设计器支持快速开发 pyGtk 图形界面用户程序。

#### 2. PyQT

Qt 是一种开源的 GUI 库，Qt 的类库大约有 300 多个，函数大约有 5700 多个。Qt 适合于

大型应用程序开发。PyQT 模块是 Qt 图形用户界面工具包标准的 Python 接口。Qt Designer 界面设计器支持快速开发 PyQT 图形界面用户程序。

### 3. wxPython

WxWidgets 是比较流行的 GUI 跨平台开发技术，适合于大型应用程序开发。wxPython 模块是 WxWidgets 图形用户界面工具包标准的 Python 接口，其功能强于 tkinter，设计的框架类似于 MFC（Microsoft Foundation Classes，微软基础类）。Boa Constructor 支持快速开发 wxPython 图形界面用户程序。

### 4. Jython

Jython 是 Python 的 Java 实现，故可以访问 Java 类库，使用 Java 的 Swing 技术构建图形用户界面程序。

### 5. IronPython

IronPython 是 Python 的 .NET 实现，故可以访问 .NET 类库，使用 .NET 类库技术构建图形用户界面程序。

## 16.2 tkinter 概述

### 16.2.1 tkinter 模块

tkinter 由若干的模块组成：`_tkinter`、`tkinter` 和 `tkinter.constants` 等。

`_tkinter` 是二进制扩展模块，提供了对 Tk 的低级接口，应用级程序员不会直接使用。`_tkinter` 通常是一个共享库（或 DLL），但是在一些情况下也可被 Python 解释器静态链接。

`tkinter` 是主要使用的模块，导入 `tkinter` 时，会自动导入 `tkinter.constants`。`tkinter.constants` 模块定义了许多常量。导入 `tkinter` 模块一般采用以下方式。

`import tkinter`：导入 `tkinter` 模块。

`import tkinter as tk`：导入 `tkinter` 模块为 `tk`。

`from tkinter import *`：导入 `tkinter` 模块的所有内容。

### 16.2.2 图形用户界面构成

图形用户界面程序一般包含一个顶层窗口（也称根窗口、主窗口）。通过类 `Tk` 的无参构造函数，可以创建应用程序主窗口。

```
from tkinter import * #导入 tkinter 模块的所有内容
root = Tk() #创建 1 个 Tk 根窗口组件 root
```

应用程序主窗口中，可以包含各种可视化组件，如文本框（`Label`）、按钮（`Button`）等。通过对应组件类的构造函数，可以创建其实例并设置其属性。例如：

```
btnSayHi = Button(root) #创建 1 个按钮组件 btnSayHi, 作为 root 的子组件
btnSayHi["text"] = "Hello" #设置 btnSayHi 的 text 属性
```

可以调用组件的 `pack/grid/place` 方法，通过几何布局管理器（Geometry Manager），调整其显示位置和大小。例如：

btnSayHi. pack()            #调用组件的 pack 方法,调整其显示位置和大小

图形用户界面通常基于事件处理机制:用户操作(如单击按钮)会引发事件,如果绑定了事件处理程序,则会调用相应的事件处理程序。例如:

```
def sayHi(e): #定义事件处理程序
 messagebox.showinfo("Message","Hello,world!") #弹出消息框
btnSayHi. bind("< Button - 1 >",sayHi) #绑定事件处理程序,鼠标左键
root. mainloop() #调用组件的 mainloop 方法,进入事件循环
```

**【例 16-1】** 创建图形用户界面 Hello world 程序 (Hello1. py)。运行结果如图 16-1 所示。

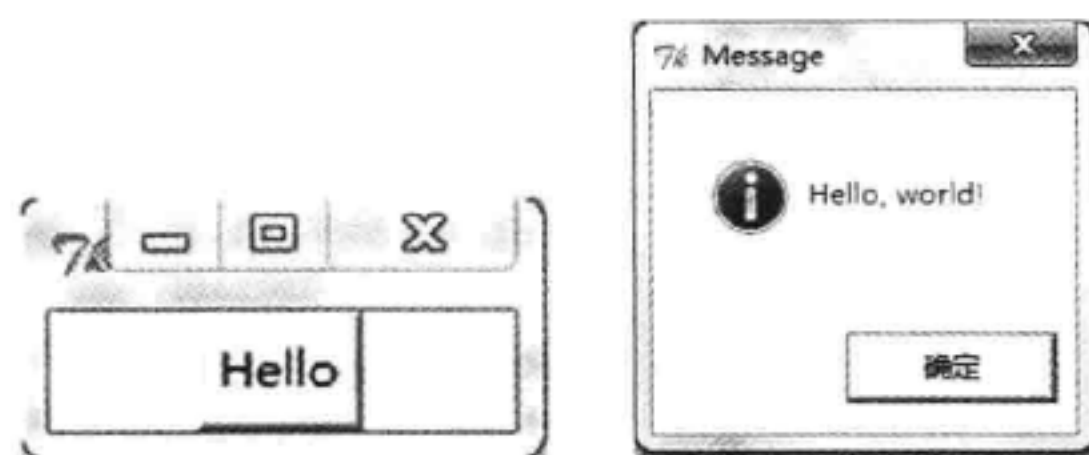


图 16-1 Hello world 窗体应用程序

|                                               |                                   |
|-----------------------------------------------|-----------------------------------|
| from tkinter import *                         | #导入 tkinter 模块所有内容                |
| root = Tk()                                   | #创建 1 个 Tk 根窗口组件 root             |
| btnSayHi = Button(root)                       | #创建 1 个按钮组件 btnSayHi,作为 root 的子组件 |
| btnSayHi["text"] = "Hello"                    | #设置 btnSayHi 的 text 属性            |
| btnSayHi. pack()                              | #调用组件的 pack 方法,调整其显示位置和大小         |
| def sayHi(e):                                 | #定义事件处理程序                         |
| messagebox.showinfo("Message","Hello,world!") | #弹出消息框                            |
| btnSayHi. bind("< Button - 1 >",sayHi)        | #绑定事件处理程序,鼠标左键                    |
| root. mainloop()                              | #调用组件的 mainloop 方法,进入事件循环         |

## 16.2.3 框架和 GUI 应用程序类

### 1. 框架 (Frame)

框架 (Frame) 是 tkinter 组件之一,表示屏幕上的一块矩形区域。框架一般作为容器使用,框架中可以包含其他组件,从而实现复杂界面的布局窗体。其构造函数为:

**f = Frame(parent, option, ...)**

其中, parent 为父组件, option 为选项,通过如下命令可列举其选项:

```
>>> f = Frame(); f. keys()
['bd','borderwidth','class','relief','background','bg','colormap','container','cursor','height','highlight-
background','highlightcolor','highlightthickness','padx','pady','takefocus','visual','width']
```

### 2. GUI 应用程序类

在开发正规和复杂的 GUI 应用程序时,一般创建一个继承于 tkinter Frame 的类 Application,在其构造函数中,调用创建其子组件的方法 createWidgets。

通过创建 Application 的对象实例,可以运行 GUI 应用程序。

**【例 16-2】** 创建 GUI 应用程序类 (Hello2. py), 实现 Hello world 程序。



|                                                                                |                           |
|--------------------------------------------------------------------------------|---------------------------|
| <code>import tkinter as tk</code>                                              | #导入 tkinter 模块            |
| <code>class Application(tk.Frame):</code>                                      | #定义 GUI 应用程序类,派生于 Frame 类 |
| <code>def __init__(self, master=None):</code>                                  | #构造函数, master 为父窗口        |
| <code>tk.Frame.__init__(self, master)</code>                                   | #调用父类的构造函数                |
| <code>self.pack()</code>                                                       | #调用组件的 pack 方法,调整其显示位置和大小 |
| <code>self.createWidgets()</code>                                              | #调用对象方法,创建子组件             |
| <code>def createWidgets(self):</code>                                          | #对象方法:创建子组件               |
| <code>self.btnSayHi = tk.Button(self)</code>                                   | #创建按钮组件 btnSayHi          |
| <code>self.btnSayHi["text"] = "Hello"</code>                                   | #设置显示文本属性                 |
| <code>self.btnSayHi["command"] = self.sayHi</code>                             | #设置命令属性,绑定事件处理程序          |
| <code>self.btnSayHi.pack()</code>                                              | #调用组件的 pack 方法,调整其显示位置和大小 |
| #创建按钮组件 btnQuit,其显示文本为"Quit",命令事件处理程序为 root.destroy                            |                           |
| <code>self.btnQuit = tk.Button(self, text="Quit", command=root.destroy)</code> |                           |
| <code>self.btnQuit.pack()</code>                                               | #调用组件的 pack 方法,调整其显示位置和大小 |
| <code>def sayHi(self):</code>                                                  | #定义事件处理程序                 |
| <code>tk.messagebox.showinfo("Message", "Hello, world!")</code>                | #弹出消息框                    |
| <code>root = tk.Tk()</code>                                                    | #创建 1 个 Tk 根窗口组件 root     |
| <code>app = Application(master=root)</code>                                    | #创建 Application 的对象实例     |
| <code>app.mainloop()</code>                                                    | #调用组件的 mainloop 方法,进入事件循环 |

## 16.2.4 tkinter 主窗口

### 1. 主窗口属性

通过类 Tk 的无参构造函数,可以创建应用程序主窗口。通过其对象方法 title,可以设置窗口标题;通过字典键,可以设置其他属性。通过如下命令可列举字典键:

```
>>> root = Tk(); root.keys()
['bd', 'borderwidth', 'class', 'menu', 'relief', 'screen', 'use', 'background', 'bg', 'colormap', 'container', 'cursor', 'height', 'highlightbackground', 'highlightcolor', 'highlightthickness', 'padx', 'pady', 'takefocus', 'visual', 'width']
```

例如:

```
>>> root = Tk() #创建 1 个 Tk 根窗口组件 root
>>> root.title('示例') #设置窗口标题
>>> root['width'] = 300; root['height'] = 50 #设置窗口宽度和高度
```

### 2. 主窗口大小和位置

通过 geometry 函数,可以设置主窗口的大小和位置,例如:

```
>>> root.geometry('200x50+0+0') #窗口大小 200 * 50,位于屏幕右上角
```

其中,参数的形式为 'wxh ± x ± y'。w 为宽度;h 为高度;+x 为主窗口左边离屏幕左边的距离,-x 为主窗口右边离屏幕右边的距离;+y 为主窗口上边离屏幕上边的距离,-y 为主窗口下边离屏幕下边的距离。

## 16.3 几何布局管理器

tkinter 几何布局管理器用于组织和管理在父组件中子配件的布局方式。tkinter 提供了 3



种不同的几何布局管理类：pack、grid 和 place。

16.3.1 pack 几何布局管理器

pack 几何布局管理器采用块的方式组织组件。pack 根据组件创建生成的顺序将子组件添加到父组件中，通过设置选项，可以控制子组件的位置等。采用 pack 的代码量最少，故在快速生成界面设计中广泛采用。

调用子组件的方法 pack，则该子组件在其父组件中采用 pack 布局：

```
pack(option = value,...)
```

pack 方法提供如表 16-1 所示的若干选项。

表 16-1 pack 方法提供的选项

| 选 项          | 意 义                  | 取值范围及说明                                                   |
|--------------|----------------------|-----------------------------------------------------------|
| side         | 停靠在父组件的哪一边上          | 'top'(默认值), 'bottom', 'left', 'right'                     |
| anchor       | 停靠对齐方式。对应于东南西北中以及四个角 | 'n', 's', 'w', 'e', 'nw', 'sw', 'se', 'ne', 'center'(默认值) |
| fill         | 填充空间                 | 'x', 'y', 'both', 'none'                                  |
| expand       | 扩展空间                 | 0 或 1                                                     |
| ipadx, ipady | 组件内部在 x/y 方向上填充的空间大小 | 单位为 c (厘米)、m (毫米)、i (英寸)、p (打印机的点)                        |
| padx, pady   | 组件外部在 x/y 方向上填充的空间大小 | 同上                                                        |

【例 16-3】 pack 几何布局示例（pack.py）。运行效果如图 16-2 所示。

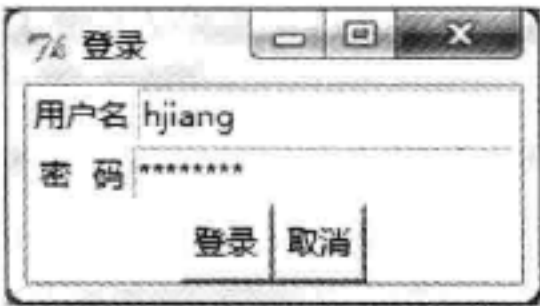


图 16-2 pack 几何布局示例

```
from tkinter import * #导入 tkinter 模块所有内容
root = Tk();root.title(" 登录") #窗口标题
f1 = Frame(root);f1.pack() #界面分为上下 3 个 Frame,f1 放置第 1 行标签和文本框
f2 = Frame(root);f2.pack() #f2 放置第 2 行标签和文本框
f3 = Frame(root);f3.pack() #f3 放置第 3 行 2 个按钮
Label(f1,text="用户名").pack(side=LEFT) #标签放置在 f1 中,左停靠
Entry(f1).pack(side=LEFT) #单行文本框放置在 f1 中,左停靠
Label(f2,text="密 码").pack(side=LEFT) #标签放置在 f2 中,左停靠
Entry(f2,show=" * ").pack(side=LEFT) #单行文本框放置在 f2 中,左停靠
Button(f3,text="取消").pack(side=RIGHT) #按钮放置在 f3 中,右停靠
Button(f3,text="登录").pack(side=RIGHT) #按钮放置在 f3 中,右停靠
root.mainloop()
```

16.3.2 grid 几何布局管理器

grid 几何布局管理采用表格结构组织组件。子组件的位置由行/列确定的单元格决定，子组件可以跨越多行/列。每一列中，列宽由这一列中最宽的单元格确定。采用 grid 布局，适合于表格形式的布局，可以实现复杂的界面，因而广泛采用。

调用子组件的方法 grid，则该子组件在其父组件中采用 grid 布局：

```
grid(option = value, ...)
```

grid 方法提供如表 16-2 所示的若干选项。

表 16-2 grid 方法提供的选项

| 选 项          | 意 义                             | 取值范围及说明                                                                            |
|--------------|---------------------------------|------------------------------------------------------------------------------------|
| column       | 单元格列号                           | 从 0 开始的正整数                                                                         |
| columnspan   | 列跨度                             | 正整数                                                                                |
| row          | 单元格行号                           | 从 0 开始的正整数                                                                         |
| rowspan      | 行跨度                             | 正整数                                                                                |
| ipadx, ipady | 组件内部在 x/y 方向上填充的空间大小            | 单位为 c（厘米）、m（毫米）、i（英寸）、p（打印机的点）                                                     |
| padx, pady   | 组件外部在 x/y 方向上填充的空间大小            | 同上                                                                                 |
| sticky       | 组件紧贴所在单元格的某一边角<br>对应于东南西北中以及四个角 | 'n', 's', 'w', 'e', 'nw', 'sw', 'se', 'ne', 'center'（默认值）。注：可紧贴多个边角。例如：tk.N + tk.S |

【例 16-4】 grid 几何布局示例 1（grid1.py）。运行效果如图 16-3 所示。

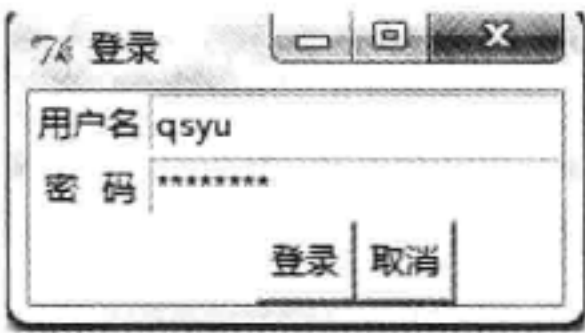


图 16-3 grid 几何布局示例 1

```
from tkinter import * #导入 tkinter 模块所有内容
root = Tk();root.title(" 登录") #窗口标题
Label(root,text = "用户名").grid(row = 0,column = 0) #用户名标签放置第 0 行第 0 列
Entry(root).grid(row = 0,column = 1,columnspan = 2) #用户名文本框放置第 0 行第 1 列,跨 2 列
Label(root,text = "密 码").grid(row = 1,column = 0) #密码标签放置第 1 行第 0 列
Entry(root,show = " * ").grid(row = 1,column = 1,columnspan = 2) #密码文本框放置第 1 行第 1 列,跨
 2 列
Button(root,text = " 登录").grid(row = 3,column = 1,sticky = E) #登录按钮右侧贴紧
Button(root,text = " 取消").grid(row = 3,column = 2,sticky = W) #取消按钮左侧贴紧
root.mainloop()
```

【例 16-5】 grid 几何布局示例 2（grid2.py）。运行效果如图 16-4 所示。

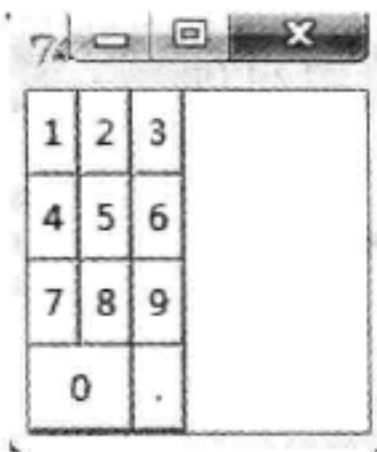


图 16-4 grid 几何布局示例 2

```
from tkinter import * #导入 tkinter 模块所有内容
root = Tk()
Button(root, text = "1").grid(row = 0, column = 0) #按钮 1 放置于 0 行 0 列
Button(root, text = "2").grid(row = 0, column = 1) #按钮 2 放置于 0 行 1 列
Button(root, text = "3").grid(row = 0, column = 2) #按钮 3 放置于 0 行 2 列
Button(root, text = "4").grid(row = 1, column = 0) #按钮 4 放置于 1 行 0 列
Button(root, text = "5").grid(row = 1, column = 1) #按钮 5 放置于 1 行 1 列
Button(root, text = "6").grid(row = 1, column = 2) #按钮 6 放置于 1 行 2 列
Button(root, text = "7").grid(row = 2, column = 0) #按钮 7 放置于 2 行 0 列
Button(root, text = "8").grid(row = 2, column = 1) #按钮 8 放置于 2 行 1 列
Button(root, text = "9").grid(row = 2, column = 2) #按钮 9 放置于 2 行 2 列
Button(root, text = "0").grid(row = 3, column = 0, columnspan = 2, sticky = E + W) #跨 2 列,左右贴紧
Button(root, text = ".").grid(row = 3, column = 2, sticky = E + W) #左右贴紧
root.mainloop()
```

16.3.3 place 几何布局管理器

place 几何布局管理允许指定组件的大小与位置。place 的优点是可以精确控制组件的位置，不足之处是改变窗口大小时，子组件不能随之灵活改变大小。

调用子组件的方法 place，则该子组件在其父组件中采用 place 布局：

```
place(option,...)
```

place 方法提供如表 16-3 所示的若干选项，可以直接给选项赋值或以字典变量加以修改。

表 16-3 place 方法提供的选项

| 选 项                 | 意 义                | 取值范围及说明                                                    |
|---------------------|--------------------|------------------------------------------------------------|
| x, y                | 绝对坐标               | 从 0 开始的正整数                                                 |
| relx, rely          | 相对坐标               | 正整数                                                        |
| width, height       | 宽和高的绝对值            |                                                            |
| relwidth, relheight | 宽和高的相对值            |                                                            |
| anchor              | 对齐方式。对应于东南西北中以及四个角 | 'n', 's', 'w', 'e', 'nw', 'sw', 'se', 'ne', 'center' (默认值) |

【例 16-6】 place 几何布局示例（place.py）。运行效果如图 16-5 所示。

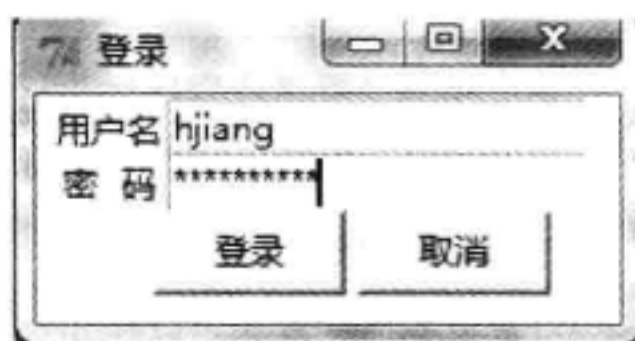


图 16-5 place 几何布局示例

```
from tkinter import * #导入 tkinter 模块所有内容
root = Tk();root.title("登录") #窗口标题
root['width'] = 200;root['height'] = 80 #窗口宽度、高度
Label(root,text="用户名",width=6).place(x=1,y=1) #绝对坐标(1,1)
Entry(root,width=20).place(x=45,y=1) #绝对坐标(45,1)
Label(root,text="密 码",width=6).place(x=1,y=20) #绝对坐标(1,20)
Entry(root,width=20,show=" * ").place(x=45,y=20) #绝对坐标(45,20)
Button(root,text="登录",width=8).place(x=40,y=40) #绝对坐标(40,40)
Button(root,text="取消",width=8).place(x=110,y=40) #绝对坐标(110,40)
root.mainloop()
```

## 16.4 事件处理

### 16.4.1 事件类型

#### 1. 事件系列

用户通过鼠标和键盘与图形用户界面交互时，会触发事件。tkinter 事件采用放置于尖括号（<>）内的字符串表示，称为事件系列（Event sequences）。其通用格式为：

<[modifier -]...type[ -detail]>

其中，可选的 modifier 用于组合键定义，如同时按下 Ctrl 键；type 表示通用类型，如键盘按键（KeyPress）；可选的 detail 用于具体信息，如按键 A。

```
<Control - Shift - Alt - KeyPress - A> : #同时按下 Ctrl、Shift、Alt 和 A 四个键
<KeyPress - A> : #按下键盘上的 A 键
<Button - 1> : #单击鼠标左键
<Double - Button - 1> : #双击鼠标左键
```

也可以使用短格式表示事件，例如：' < 1 >' 等同于 ' < Button - 1 >'；' x' 等同于 ' < KeyPress - x >'。

#### 2. 事件类型

tkinter 事件如表 16-4 所示。



表 16-4 tkinter 事件

| 类 别  | Type | 名 称           | 说 明                                                          |
|------|------|---------------|--------------------------------------------------------------|
| 键盘事件 | 2    | KeyPress      | 按下键盘时触发，可以在 detail 部分指定是哪个键                                  |
|      | 3    | KeyRelease    | 释放键盘时触发，可以在 detail 部分指定是哪个键                                  |
| 鼠标事件 | 4    | Button        | 按下鼠标时触发，可以在 detail 部分指定是哪个键                                  |
|      | 5    | ButtonRelease | 释放鼠标时触发，可以在 detail 部分指定是哪个键                                  |
|      | 6    | Motion        | 点中组件的同时托拽组件移动时触发                                             |
|      | 7    | Enter         | 当鼠标指针移进某组件时，该组件触发                                            |
|      | 8    | Leave         | 当鼠标指针移出某组件时，该组件触发                                            |
|      | 38   | MouseWheel    | 当鼠标滚轮滚动时触发                                                   |
| 窗体事件 | 15   | Visibility    | 当组件变为可视状态时触发                                                 |
|      | 18   | Unmap         | 当组件由显示状态变为隐藏状态时触发                                            |
|      | 19   | Map           | 当组件由隐藏状态变为显示状态时触发                                            |
|      | 12   | Expose        | 当组件从被其他组件遮盖状态中暴露出来时触发                                        |
|      | 9    | FocusIn       | 组件获得焦点时触发                                                    |
|      | 10   | FocusOut      | 组件失去焦点时触发                                                    |
|      | 22   | Configure     | 当改变组件大小时触发。例如托拽窗体边缘                                          |
|      | 36   | Activate      | 与组件选项中的 state 项有关，表示组件由不可用转为可用。例如按钮由 disabled（灰色）转为 enabled  |
|      | 37   | Deactivate    | 与组件选项中的 state 项有关，表示组件由可用转为不可用。例如按钮由 enabled 转为 disabled（灰色） |
|      | 17   | Destroy       | 当组件被销毁时触发                                                    |

3. 组合键修饰符

tkinter 组合键修饰符如表 16-5 所示。

表 16-5 tkinter 组合键修饰符

| 修 饰 符   | 说 明                                         |
|---------|---------------------------------------------|
| Shift   | 当 Shift 键按下                                 |
| Control | Ctrl 键按下                                    |
| Alt     | 当 Alt 键按下                                   |
| Any     | 任何按键按下，如 < Any - KeyPress >                 |
| Double  | 两个事件在短时间内发生，如双击鼠标左键 < Double - Button - 1 > |
| Triple  | 类似于 Double，三个事件短时间内发生                       |
| Lock    | 当 Caps Lock 键按下                             |

16.4.2 事件绑定

1. 创建组件对象实例时指定

创建组件对象实例时，可通过其命名参数 command 指定事件处理函数。

2. 实例绑定

调用组件对象实例方法 `bind`，可为指定组件实例绑定事件：

```
w. bind("<event>", eventhandler, add = "")
```

其中，`<event>` 为事件；`eventhandler` 为事件处理函数；可选参数 `add` 默认为`""`，表示事件处理函数替代其他绑定，如果为`'+'`，则加入事件处理队列。

例如，绑定组件对象，使得 `Canvas` 组件实例 `canvas1` 可以处理鼠标左键事件：

```
>>> canvas1 = Canvas(); canvas1. bind("<Button-1>", drawline)
```

3. 类绑定

调用组件对象实例方法 `bind_class` 函数，可以为特定组件类绑定事件：

```
w. bind_class("Widget", "<event>", eventhandler, add = "")
```

其中，`Widget` 为组件类；`<event>` 为事件；`eventhandler` 为事件处理函数。

例如，绑定组件类，使得所有 `Canvas` 组件实例都可以处理鼠标左键事件：

```
>>> canvas1 = Canvas(); canvas1. bind_class("Canvas", "<Button-1>", drawline)
```

4. 程序界面绑定

调用组件对象实例方法 `bind_all` 函数，可以为所有组件类绑定事件：

```
w. bind_all("<event>", eventhandler, add = "")
```

其中，`<event>` 为事件；`eventhandler` 为事件处理函数。

例如，将 `PrintScreen` 键与程序中的所有组件对象绑定，使得整个程序界面都能处理打印屏幕的键盘事件：

```
>>> canvas1 = Canvas(); canvas1. bind_all("<Key-Print>", printscreen)
```

16.4.3 事件处理函数

1. 定义事件函数和事件方法

事件处理可以定义为函数，也可以定义为对象的方法。两者都带一个参数：`event`。触发事件调用事件处理函数时，将传递 `Event` 对象实例。

```
def handlerName(event) :
 函数体
def handlerName(self, event) :
 方法体
```

2. Event 事件对象参数属性

通过传递的 `Event` 事件对象的属性，可以获取各种相关参数。`Event` 事件对象主要参数属性如表 16-6 所示。

表 16-6 Event 事件对象的主要参数属性

| 参 数                                         | 意 义             |
|---------------------------------------------|-----------------|
| <code>.x</code> 、 <code>.y</code>           | 鼠标相对于组件对象左上角的坐标 |
| <code>.x_root</code> 、 <code>.y_root</code> | 鼠标相对于屏幕左上角的坐标   |

续表

| 参 数          | 意 义                                                                                                                                                              |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| . state      | 辅助键的状态信息<br>0x0001：Shift 键<br>0x0002：大写锁定键<br>0x0004：Ctrl 键<br>0x0008：左 Alt 键<br>0x0010：NumLock 键<br>0x0080：右 Alt 键<br>0x0100：鼠标左键<br>0x0200：鼠标中键<br>0x0400：鼠标右键 |
| . keysym     | 命名按键                                                                                                                                                             |
| . keysym_num | 按键序号                                                                                                                                                             |
| . keycode    | 键码                                                                                                                                                               |
| . time       | 时间                                                                                                                                                               |
| . type       | 类型                                                                                                                                                               |
| . widget     | 组件                                                                                                                                                               |

3. 按键详细信息

按键和释放键（KeyPress，KeyRelease）时，Event 事件涉及不同的按键详细信息：  
. keysym、. keysym\_num、. keycode。对应关系如表 16-7 所示。

表 16-7 Event 事件的按键详细信息

| . keysym  | . keycode | . keysym_num  | Key          |
|-----------|-----------|---------------|--------------|
| F1 ~ F11  | 67 ~ 77   | 65470 ~ 65480 | 功能键 F1 ~ F11 |
| F12       | 96        | 65481         | 功能键 F12      |
| Print     | 111       | 65377         | 打印屏幕键        |
| Alt_L     | 64        | 65513         | 左 Alt 键      |
| Alt_R     | 113       | 65514         | 右 Alt 键      |
| BackSpace | 22        | 65288         | BackSpace    |
| Cancel    | 110       | 65387         | Pause Break  |
| Escape    | 9         | 65307         | Esc 键        |

16.5 组件概述

16.5.1 创建组件和设置属性

1. 创建组件对象实例

tkinter 提供了若干组件类，如表示按钮的 Button、表示标签的 Label 等。通过组件类的构造函数，可以创建其对象实例。例如：

```
>>> from tkinter import *
>>> root = Tk() #创建 1 个 Tk 根窗口组件 root
>>> button1 = Button(root, text = "确定") #创建 Button 对象实例,设置标题
>>> button1. pack() #调用组件的 pack 方法,调整其显示位置和大小
>>> Label(root, text = 'ABC'). pack() #创建 Label 对象并显示
```

运行效果如图 16-6 所示。



图 16-6 创建组件对象实例

## 2. 设置组件属性

可以设置组件的属性,如大小、字体、颜色、显示文本等。各组件类包含相应的选项。可以通过下列方式之一设置组件属性。

`button1 = Button(text = "OK")`: 组件构造函数的命名参数。

`button1. config(text = "OK")`: 组件对象的 `config` 方法的命名参数。

`button1["text"] = "OK"`: 组件对象的字典属性。

## 16.5.2 坐标系

每个组件都有一个坐标系。坐标原点  $(0, 0)$  位于组件的左上角,  $x$  轴的方向向右,  $y$  轴的方向向下, 计量单位为像素。组件的坐标系如图 16-7 (a) 所示。请注意与传统坐标系 (如图 16-7 (b) 所示) 的区别。

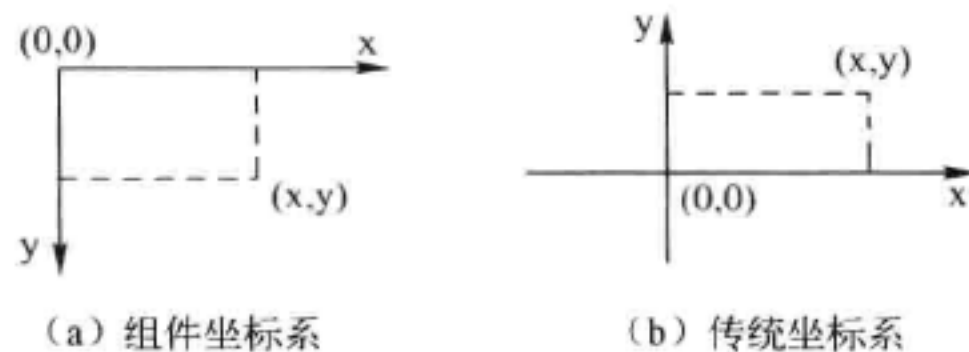


图 16-7 坐标系

## 16.5.3 大小单位 (width 和 height)

默认情况下, 组件的大小由其内容决定。通过组件的 `width` 和 `height` 选项, 可以设置组件的宽度和高度。例如:

```
Label(root, text = 'red', width = 10, height = 2, bg = 'red'). pack() #红色标签文本内容为 red'
Label(root, text = 'yellow', width = 10, height = 2, bg = 'yellow'). pack() #黄色标签文本内容为 yellow'
```

## 16.5.4 颜色 (background、foreground 等)

通过组件的选项, 可以设置其颜色配置。颜色选项包括以下几种。

`background` 或 `bg`、`foreground` 或 `fg`: 背景色、前景色。



disabledforeground: 禁用时前景色。

highlightbackground: 无焦点高亮颜色。

highlightcolor: 有焦点高亮颜色。

highlightthickness: 焦点高亮的宽度。

activebackground、activeforeground: 鼠标经过时背景色、前景色。

颜色可以使用下列四种方式表示。

'#rgb': 4 位颜色值, 例如: '#fff' 表示白色

'#rrggbb': 8 位颜色值, 例如: '#000000' 表示黑色

'#rrrgggbbb': 12 位颜色值, 例如: '#fff000000' 表示红色

'colorname': 颜色名称, 常用的包括 'white', 'black', 'red', 'green', 'blue', 'cyan', 'yellow', 'magenta'。颜色名称与本地安装有关, 且大小写无关。

例如:

```
>>> Label(root, bg = 'blue', fg = 'WHITE', text = 'Hello, tkinter! '). pack() # Hello, tkinter!
```

### 16.5.5 字体 (font)

通过组件的 font 选项, 可以设置其显示的文本的字体。

#### 1. 通过元组表示字体

通过 3 个元素的元组, 可以表示字体:

```
("font family", size, modifiers)
```

其中, "font family" 为字体; size 为大小, 正数单位为 point, 负数单位为 pixel; modifiers 为修饰符, 包括 bold、italic、underline 和 overstrike, 分别为粗体、斜体、下划线和删除线。例如:

```
>>> Label(root, text = '您好!', font = ('宋体', 9, 'italic')). pack() #9 - point 斜体宋体标签: 您好!
```

#### 2. 通过 Font 对象表示字体

导入 tkFont 模块后, 可创建 Font 对象, 并设置其各种属性选项。

```
font = tkFont. Font(option, ...)
```

option 选项包括: family = 字体; size = 大小; weight = 'bold' 或 'normal', 'bold' 为粗体; slant = 'italic' 或 'roman', 'italic' 为斜体; underline = 1 或 0, 1 为加下划线; overstrike = 1 或 0, 1 为加删除线。例如:

```
>>> import tkinter. font
```

```
>>> song9 = tkinter. font. Font(family = '宋体', size = 9, weight = 'bold', slant = 'italic') #9 - point 粗斜体宋体
```

```
>>> Label(root, text = '您好!', font = song9). pack() #您好!
```

通过 tkFont. families() 函数, 可以返回所有可用的字体。

```
>>> import tkinter, tkinter. font; root = tkinter. Tk(); tkinter. font. families()
```

### 16.5.6 停靠位置 (anchor)

通过组件的 anchor 选项, 可以设置内容停靠位置。anchor 取值包括: 'e'、'w'、'n'、's'、'center' (默认值)、'nw'、'sw'、'se'、'ne'; 也可使用 tkinter 模块中的常量: E、W、N、S、CENTER、NW、SW、SE、NE, 分别对应于如图 16-8 所示的地理方位东西南北中和四角。

例如：

```
>>> Label(root, text = '左上角', width = 15, height = 2, anchor = 'nw', bg = 'yellow'). pack()
```

运行效果如图 16-9 所示。

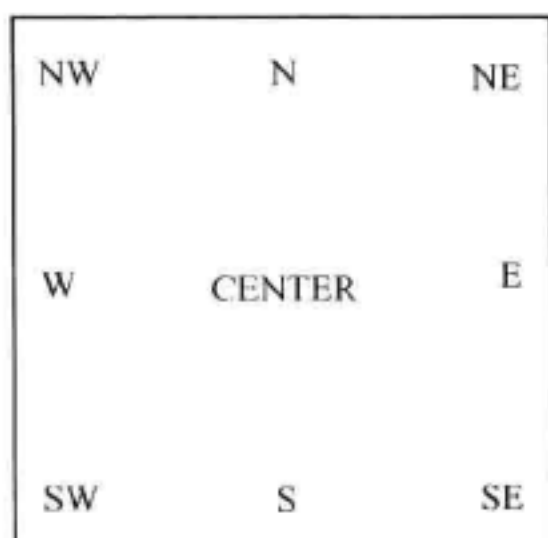


图 16-8 地理方位

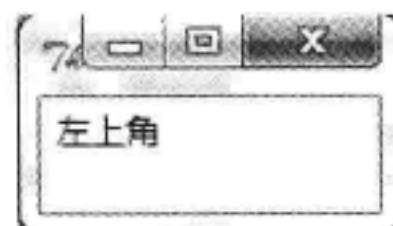


图 16-9 停靠位置

### 16.5.7 光标 (cursor)

通过组件的 `cursor` 选项，可以设置鼠标经过组件时的光标形状。tkinter 内置的光标包括 `arrow`、`question_arrow`、`watch` 等。例如：

```
>>> Button(root, text = "Click me!", cursor = "watch"). pack() #设置按钮光标为"watch":Click me!
```

### 16.5.8 显示文本 (text、textvariable、wraplength 和 justify)

通过组件的 `text` 选项，可以设置其显示的内容，通过 `wraplength` 选项，可指定多少单位后开始换行，即显示多行；通过 `justify` 选项，可指定多行的对齐方式。`justify` 取 tkinter 模块中的常量：`LEFT`，左对齐；`CENTER`，居中对齐（默认值）；`RIGHT`，右对齐。例如：

```
>>> Label(root, text = '文本标签'). pack() #设置 text 选项
```

```
>>> Label(root, text = '多行文本!', wraplength = 30, justify = RIGHT). pack()
```

运行效果如图 16-10 所示。

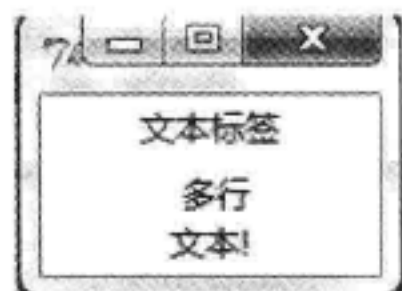


图 16-10 显示文本

### 16.5.9 位图 (bitmap)

通过组件的 `bitmap` 选项，可以设置其显示的位图。内置位图包括：`'error'`、`'gray75'`、`'gray50'`、`'gray25'`、`'gray12'`、`'hourglass'`、`'info'`、`'questhead'`、`'question'` 和 `warning'`，分别对应图 16-11 所示的图标。自定义位图为 `.xpm` 格式的文件。



图 16-11 组件的位图图标

例如：

```
>>> Label(root, bitmap='error'). pack()
>>> Label(root, bitmap=r'@ C:\Python\images\xbm\face. xbm'). pack()
```

运行效果如图 16-12 所示。



图 16-12 显示位图

### 16.5.10 图像 (image)

通过组件的 image 选项，可以设置其显示的图像。BitmapImage 和 PhotoImage 对象提供对位图和图像的支持，更复杂的图像处理，建议采用 Python 图像库 (PIL)。

```
tkinter. BitmapImage(file = f[, background = b][, foreground = c])
```

```
tkinter. PhotoImage(file = f)
```

其中，file 为文件名，BitmapImage 为 .xbm 格式的文件，PhotoImage 为 .gif、.pgm 和 .ppm 格式的文件；background 和 foreground 分别为背景色和前景色，默认情况下，foreground = 1 (黑色)，background = 0 (透明)。例如：

```
>>> logo = BitmapImage(file = r'C:\Python\images\xbm\face. xbm')
>>> Label(root, image = logo). pack()
>>> logo = PhotoImage(file = r'C:\Python\images\gif\earth. gif')
>>> Label(root, image = logo). pack()
```

运行效果如图 16-13 所示。

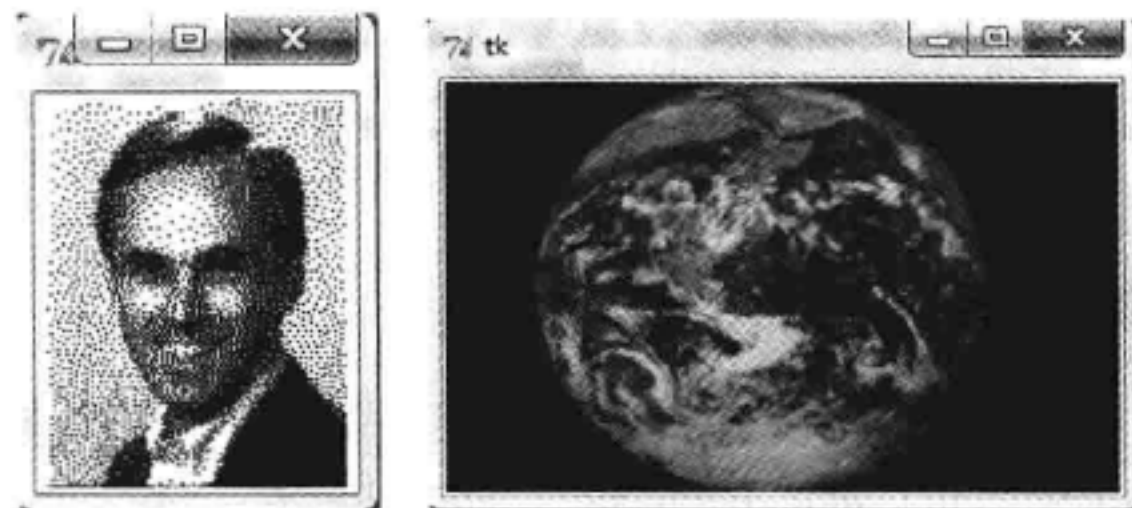



图 16-13 显示图像

### 16.5.11 同时显示文本和位图/图像 (compound)

通过组件的 compound 选项，可以设置其同时显示文本和位图/图像。compound 选项默



认值为 None，即当指定 bitmap/image 时，文本（text）将被覆盖，只显示位图/图像。compound 选项包括：left，图像居左；right，图像居右；top，图像居上；bottom，图像居下；center，文字覆盖在图像上。例如：

```
>>> Label(root, text='Error', compound='left', bitmap='error'). pack() #  Error
```



### 16.5.12 3D 显示样式（relief 和 overrelief）

通过组件的 relief 选项，可以设置其 3D 显示样式。通过 overrelief 选项，可以设置其鼠标经过时的 3D 显示样式。其可用值包括 tkinter 模块常量：FLAT、RAISED、SUNKEN、GROOVE 和 RIDGE，如图 16-14 所示。





图 16-14 3D 显示样式

例如：

```
>>> Button(root, text='取消', relief=GROOVE). pack() # 
>>> Button(root, text='取消', overrelief=SUNKEN). pack() # 
```


### 16.5.13 设置组件的边框（borderwidth）

通过组件的 borderwidth 或 bd 选项，可以设置其边框宽度。例如：

```
>>> Button(root, text='取消', bd=6). pack(side=LEFT) # 
>>> Button(root, text='取消', bd=1). pack(side=LEFT) # 
```


### 16.5.14 设置组件的填充（padx 和 pady）

通过组件的 padx 和 pady 选项，可以设置其显示内容与边框之间的填充宽度和高度。例如：

```
>>> Button(root, text='取消', padx=20, pady=4). pack() # 
```

### 16.5.15 组件的状态（state）

通过组件的 state 选项，可以设置其启用或禁用状态。state 取 tkinter 模块中的常量：DISABLED，禁用；NORMAL，启用（正常）、ACTIVE，鼠标经过时的值。例如：

```
>>> Button(root, text='取消', state=DISABLED). pack() # 
```

### 16.5.16 启用和禁用输入法

```
void enableInputMethods(boolean b) #启用(或禁用)输入法
```

例如：

```
textfield1. enableInputMethods(true); #启用文本框的输入法
```

### 16.5.17 设置字符下划线位置（underline）

通过组件的 underline 选项，可以设置组件显示文本第几个字符加下划线。默认为 -1，不加下划线；如果为 1，则第 2 个字符加下划线。例如：



```
>>> Button(root, text = 'Save As', underline = 5). pack() # Save As
```

### 16.5.18 焦点 (highlightbackground、highlightcolor 等)

通过组件的焦点选项，可以设置其焦点配置。焦点选项包括：

|                            |                       |
|----------------------------|-----------------------|
| <b>highlightbackground</b> | #无焦点高亮颜色              |
| <b>highlightcolor</b>      | #有焦点高亮颜色              |
| <b>highlightthickness</b>  | #焦点高亮的宽度              |
| <b>takefocus</b>           | #默认为 1;如果设置为 0,则不接受焦点 |

### 16.5.19 组件的数据绑定 (textvariable、variable 和 listvariable)

通过组件的 textvariable 选项，可以绑定 StringVar 对象到组件，改变 StringVar 对象的值，使其自动显示在组件上。通过 StringVar 对象的 set() 和 get() 方法，可以设置或获取其值。例如：

```
>>> w = Button(root, text = "确定") #创建 Button 组件对象,显示文本为"确定"
>>> w. pack() # 确定
>>> v = StringVar() #创建 StringVar 对象
>>> w['textvariable'] = v #绑定到组件
>>> v. set('OK') #设置 StringVar 对象的值,组件文本自动更新 # OK
>>> v. get() #获取 StringVar 对象的值 #'OK'
```

通过 Radiobutton、Checkbutton、Scale 等组件的 variable 选项、Listbox 组件的 listvariable 选项，可以绑定 StringVar、IntVar 或 DoubleVar 对象到组件的选择值。

### 16.5.20 组件 id 和名称

tkinter 的窗口 (组件) 都有一个 id，通过 w. wininfo\_id() 可以获取：

```
>>> b = Button(root, text = "OK"); b. wininfo_id() #9898886
```

tkinter 使用窗口路径名称来命名窗口 (组件)。主窗口 (根) 为 .；子窗口 (组件) 为 . n，为数字字符串；子窗口 (组件) 的子窗口 (组件) 为 . p. n；以此类推。str(w) 返回组件 w 的名称。例如：

```
>>> str(Button(root, text = "OK")) #' .49959152'
```

使用组件的下列函数，可以处理窗口名称。

w. wininfo\_name()：返回 w 的名称。

w. wininfo\_parent()：返回 w 的父窗口的名称。

w. wininfo\_pathname(id)：返回 id 的窗口名称。

例如：

```
>>> b1 = Button(root, text = "OK")
>>> b1. wininfo_name(), b1. wininfo_parent(), b1. wininfo_pathname(b1. wininfo_id())
('49821200', '.', '.', '49821200')
```

### 16.5.21 组件的通用方法

tkinter 的组件具有许多通用方法，主要包括以下几种。

w. keys(): 返回组件的选项。  
 w. destroy(): 销毁组件。  
 w. clipboard\_clear(): 清空剪贴板。  
 w. clipboard\_append(text): 附加内容到剪贴板。  
 w. selection\_get(): 返回选择内容。  
 w. selection\_get(selection='CLIPBOARD'): 返回剪贴板内容。

## 16.6 常用组件

### 16.6.1 标签 Label

Label (标签) 主要用于显示文本信息。Label 既可显示文本, 也可显示图像。

**w = tkinter.Label( parent, option = value, ... )**

其中, parent 为父组件; option 为选项, 通过如下命令可列举其选项:

```
>>> name = Label(); name.keys()
['activebackground', 'activeforeground', 'anchor', 'background', 'bd', 'bg', 'bitmap', 'borderwidth', 'compound', 'cursor', 'disabledforeground', 'fg', 'font', 'foreground', 'height', 'highlightbackground', 'highlightcolor', 'highlightthickness', 'image', 'justify', 'padx', 'pady', 'relief', 'state', 'takefocus', 'text', 'textvariable', 'underline', 'width', 'wraplength']
```

创建和设置 Label 组件可参考 16.5 节。例如:

```
>>> w = Label(root, text = "姓名") #创建 Label 组件对象, 显示文本为"姓名"
>>> w.config(width = 20, bg = 'black', fg = 'white') #设置宽度、背景色、前景色
>>> w['anchor'] = E #设置停靠方式为右对齐
>>> w.pack() #
```

### 16.6.2 标签框架 LabelFrame

LabelFrame (标签框架) 是一个带标签的矩形框架, 主要用于包含若干组件。

**w = tkinter.LabelFrame( parent, option = value, ... )**

其中, parent 为父组件; option 为选项, 通过如下命令可列举其选项:

```
>>> labelf = LabelFrame(); labelf.keys()
['bd', 'borderwidth', 'class', 'fg', 'font', 'foreground', 'labelanchor', 'labelwidget', 'relief', 'text', 'background', 'bg', 'colormap', 'container', 'cursor', 'height', 'highlightbackground', 'highlightcolor', 'highlightthickness', 'padx', 'pady', 'takefocus', 'visual', 'width']
```

特定于 LabelFrame 的属性如下。

labelanchor: 指定框架的标签显示的方位, 取值同 anchor, 参见 16.5 节。

labelwidget: 指定组件对象, 来代替通常的文本标签。

创建和设置 LabelFrame 组件可参考 16.5 节。例如:

```
>>> lf = LabelFrame(root, text = "组 1") #创建 LabelFrame 组件对象
>>> lf.pack()
>>> Button(lf, text = "确定"). pack(side = LEFT)
```

```
>>> Button(lf, text = "取消"). pack(side = LEFT)
```

运行效果如图 16-15 所示。

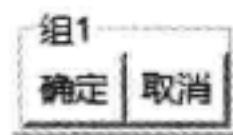


图 16-15 标签框架

### 16.6.3 按钮 Button

Button 用于执行用户的单击操作。如果焦点位于某个 Button，使用鼠标或空格键单击该按钮时，会产生 command 事件。

```
w = tkinter.Button(parent, option = value, ...)
```

其中，parent 为父组件，option 为选项，通过如下命令可列举其选项：

```
>>> b = Button(); b.keys()
['activebackground', 'activeforeground', 'anchor', 'background', 'bd', 'bg', 'bitmap', 'borderwidth', 'command', 'compound', 'cursor', 'default', 'disabledforeground', 'fg', 'font', 'foreground', 'height', 'highlightbackground', 'highlightcolor', 'highlightthickness', 'image', 'justify', 'overrelief', 'padx', 'pady', 'relief', 'repeatdelay', 'repeatinterval', 'state', 'takefocus', 'text', 'textvariable', 'underline', 'width', 'wraplength']
```

特定于 Button 的属性包括以下几种。

command：指定 Button 的回调函数。

repeatinterval 和 repeatdelay：重复间隔和时延，可设置长按按钮时产生重复事件。

创建和设置 Button 组件可参考 16.5 节。例如：

```
>>> w = Button(root, text = "确定") #创建 Button 组件对象,显示文本为"确定"
>>> w.config(state = DISABLED) #设置 Label 组件的状态为禁用
>>> w['width'] = 20 #设置宽度
>>> w.pack() # 
```

Button 包括下列对象方法。

flash()：按钮在 active color 和 normal color 之间闪烁几次，disabled 状态无效。

invoke()：调用按钮的 command 指定的回调函数，disabled 状态无效。

【例 16-7】Label 和 Button 示例 (PictureViewer.py)：简易图片浏览器。运行效果如图 16-16 所示。

```
import tkinter as tk, os #导入 tkinter 模块
class Application(tk.Frame): #定义 GUI 应用程序类,派生于 Frame 类
 def __init__(self, master = None): #构造函数, master 为父窗口
 self.files = os.listdir(r'c:\python\images\gif') #获取图像文件名列表
 self.index = 0 #图片索引,初始显示第 1 张图片
 self.img = tk.PhotoImage(file = r'c:\python\images\gif' + '\\' + self.files[self.index])
 tk.Frame.__init__(self, master) #调用父类的构造函数
 self.pack() #调用组件的 pack 方法,调整其显示位置和大小
 self.createWidgets() #调用对象方法,创建子组件
 def createWidgets(self): #对象方法:创建子组件
```



```

self. lblImage = tk. Label(self, width = 300, height = 300) #创建 Label 组件,显示图片
self. lblImage['image'] = self. img #显示第 1 张图片
self. lblImage. pack() #调用组件的 pack 方法,调整其显示位置和大小
self. f = tk. Frame() #创建窗口框架
self. f. pack()
self. btnPrev = tk. Button(self. f, text = '上一张', command = self. prev) #创建按钮组件
self. btnPrev. pack(side = tk. LEFT)
self. btnNext = tk. Button(self. f, text = '下一张', command = self. next) #创建按钮组件
self. btnNext. pack(side = tk. LEFT)

def prev(self): #定义事件处理程序
 self. showfile(- 1) #显示上一张图片
def next(self): #定义事件处理程序
 self. showfile(1) #显示下一张图片
def showfile(self, n):
 self. index += n
 if self. index < 0: self. index = len(self. files) - 1 #循环显示最后 1 张
 if self. index > len(self. files) - 1: self. index = 0 #循环显示第 1 张
 self. img = tk. PhotoImage(file = r'c:\python\images\gif' + '\\' + self. files[self. index])
 self. lblImage['image'] = self. img

root = tk. Tk() #创建 1 个 Tk 根窗口组件 root
root. title('简易图片浏览器') #设置窗口标题
app = Application(master = root) #创建 Application 的对象实例
app. mainloop() #调用组件的 mainloop 方法,进入事件循环

```

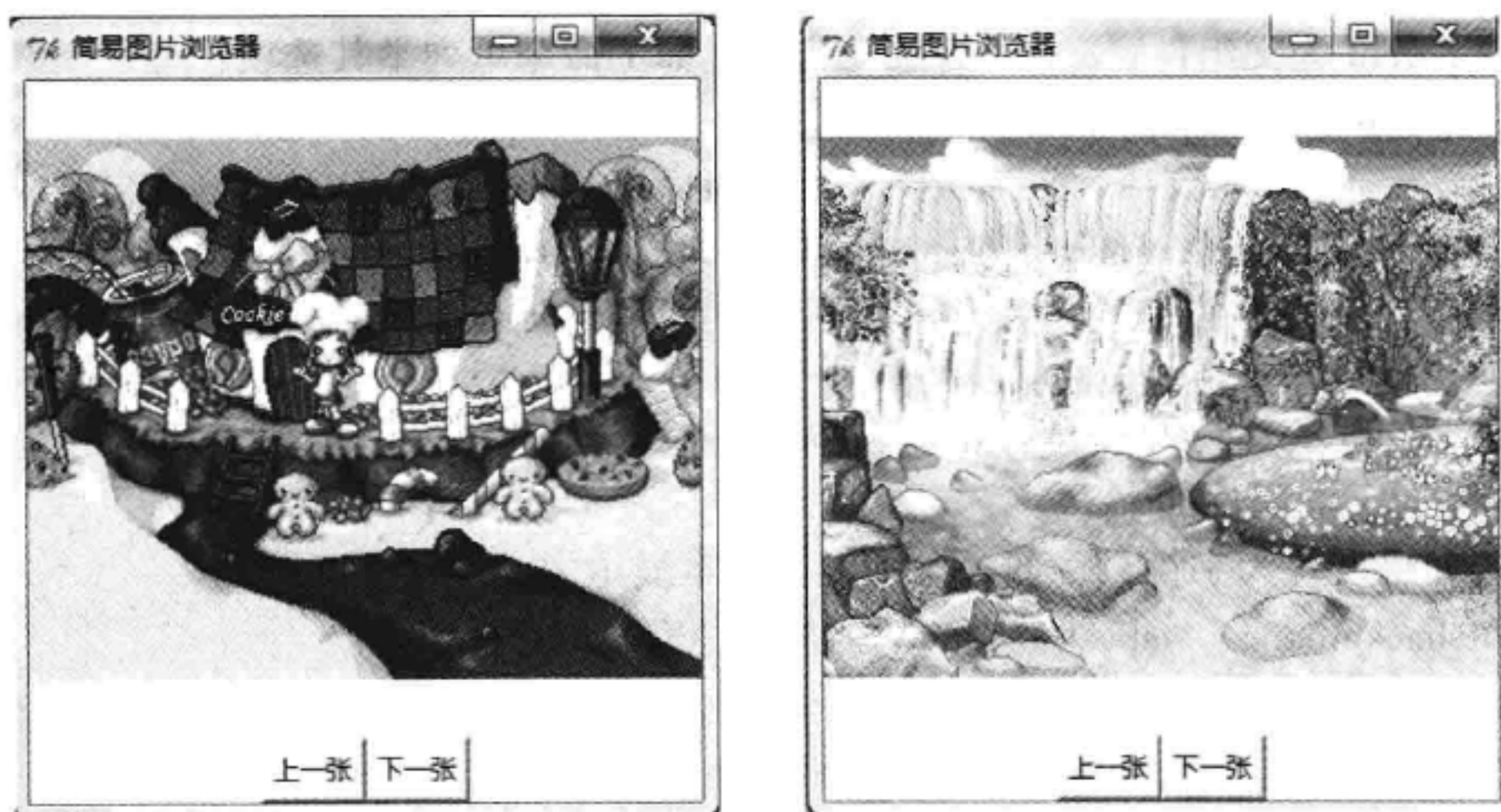


图 16-16 简易图片浏览器运行效果

### 16.6.4 消息 Message

Message (消息) 和 Label 一样,也是用来显示文本信息,但主要用于显示多行文本信息。



**w = tkinter. Message( parent, option = value, ...)**

其中, parent 为父组件; option 为选项, 通过如下命令可列举其选项:

```
>>> m = Message(); m.keys()
['anchor', 'aspect', 'background', 'bd', 'bg', 'borderwidth', 'cursor', 'fg', 'font', 'foreground', 'highlightbackground',
'highlightcolor', 'highlightthickness', 'justify', 'padx', 'pady', 'relief', 'takefocus', 'text', 'textvariable', 'width']
```

特定于 Message 的属性如下。

**aspect:** 指定宽高比例。默认为 150, 即宽是高的 150 %

例如:

```
>>> w = Message(root, bg = 'black', fg = 'white') #创建 Message 组件对象
>>> w.config(text = "内容显示在一个宽高比为 150% 的消息框中") #设置显示文本
>>> w['anchor'] = W #设置停靠方式为左对齐
>>> w.pack()
```

运行效果如图 16-17 所示。

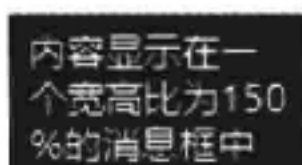


图 16-17 消息

## 16.6.5 单行文本框 Entry

Entry (单行文本框) 主要用于显示和编辑文本。

**w = tkinter. Entry( parent, option, ...)**

其中, parent 为父组件; option 为选项, 通过如下命令可列举其选项:

```
>>> e = Entry(); e.keys()
['background', 'bd', 'bg', 'borderwidth', 'cursor', 'disabledbackground', 'disabledforeground', 'exportselection',
'fg', 'font', 'foreground', 'highlightbackground', 'highlightcolor', 'highlightthickness', 'insertbackground', 'insert-
borderwidth', 'insertofftime', 'insertontime', 'insertwidth', 'invalidcommand', 'invcmd', 'justify', 'readonlyback-
ground', 'relief', 'selectbackground', 'selectborderwidth', 'selectforeground', 'show', 'state', 'takefocus', 'textvari-
able', 'validate', 'validatecommand', 'vcmd', 'width', 'xscrollcommand']
```

特定于 Entry 的属性如下。

**show:** 如果设置为字符, 例如 \*, 则输入文本显示为 \*, 用于密码框。

**readonlybackground:** state 为 'readonly' 时的背景色。

**exportselection:** 默认为 1, 选择内容自动导出到剪贴板; 设置为 0 时, 不导出。

**insertbackground:** 插入光标的颜色, 默认为 'black'。

**insertwidth** 和 **insertborderwidth:** 插入光标的宽度和边框宽度。

**insertofftime:** 插入光标闪烁熄灭时间。默认为 300ms, 设置为 0 时, 不闪烁。

**insertontime:** 插入光标闪烁点亮时间。默认为 600ms。

**selectbackground** 和 **selectforeground:** 选中文本的背景色和前景色。

**selectborderwidth:** 选中文本的边框。

**validate** 和 **validatecommand:** 用于输入内容检验。

xscrollcommand: 用于显示滚动条。

例如:

```
>>> v = StringVar() #创建 StringVar 对象
>>> w1 = Entry(root, textvariable = v) #创建 Entry 组件对象
>>> w1.pack() #显示单行文本框 | | | | |
>>> w1.get() #输入 abcd 后, 获取组件的内容: |abcd|
>>> v.set('1234') #设置 StringVar 对象的值, 组件文本自动更新: |1234|
```

Entry 包括下列对象方法。

w.get(): 返回文本框中的内容。

其他方法涉及文本选择和滚动条处理, 本书不展开阐述, 具体请参见帮助文档。

### 16.6.6 多行文本框 Text

Text (多行文本框) 主要用于显示和编辑多行文本。

**w = tkinter.Text(parent, option, ...)**

其中, parent 为父组件; option 为选项, 通过如下命令可列举其选项:

```
>>> t = Text(); t.keys()
['autoseparators', 'background', 'bd', 'bg', 'blockcursor', 'borderwidth', 'cursor', 'endline', 'exportselection',
'fg', 'font', 'foreground', 'height', 'highlightbackground', 'highlightcolor', 'highlightthickness', 'inactiveselect-
background', 'insertbackground', 'insertborderwidth', 'insertofftime', 'insertontime', 'insertwidth', 'maxundo',
'padx', 'pady', 'relief', 'selectbackground', 'selectborderwidth', 'selectforeground', 'setgrid', 'spacing1', 'spac-
ing2', 'spacing3', 'startline', 'state', 'tabs', 'tabstyle', 'takefocus', 'undo', 'width', 'wrap', 'xscrollcommand',
'yscrollcommand']
```

Text 组件支持剪贴板操作 (使用 Ctrl + C、Ctrl + V 等), 并提供文本的获取、删除、插入、替换等功能。

w.get(index1, index2 = None): 返回指定范围的文本, 没有指定 index2 时, 返回 1 个字符。

w.delete(index1, index2 = None): 删除指定范围的文本。

w.insert(index, text): 在 index 位置插入文本 text。

w.replace(self, index1, index2, chars): 替换指定范围的文本。

其中, index 格式为 line.column, line 从 1 开始, column 从 0 开始, 如 1.0 表示第 1 行第 1 列。index 也可以为位置标签 (对应于 line.column 的标记), tkinter 模块定义了若干特殊的位置标签, 其中包括: INSERT (当前插入光标位置)、END (最后一个字符之后)、SEL\_FIRST (选择开始位置)、SEL\_LAST (选择结束位置之后)。

多行文本框往往需要结合滚动条组件 Scrollbar 使用, 限于篇幅, 本书没有展开。模块 tkinter 的子模块 scrolledtext 中包含了滚动条组件。

Text 组件还支持其他高级编辑功能, 包括撤销、重做等。限于篇幅, 本书没有展开, 请读者参考 tkinter 手册。

例如:

```
>>> w = Text(root, width = 20, height = 5)
>>> w.pack()
```

```
>>> w.insert(1.0,'生,还是死,这是一个问题!\n')
>>> w.get(1.0) #'生'
>>> w.get(1.0,END) #'生,还是死,这是一个问题!\n'
```

运行效果如图 16-18 所示。

【例 16-8】 Entry、Text 示例 (register.py): 用户注册。运行效果如图 16-19 所示。

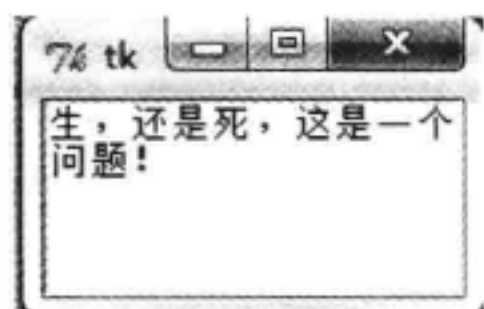


图 16-18 多行文本框



图 16-19 Entry 和 Text 的运行效果

```
import tkinter as tk
class Application(tk.Frame):
 def __init__(self, master=None):
 tk.Frame.__init__(self, master)
 self.grid()
 self.createWidgets()
 def createWidgets(self):
 self.lblEmail = tk.Label(self, text='用户名')
 self.lblPass1 = tk.Label(self, text='密码')
 self.lblPass2 = tk.Label(self, text='确认密码')
 self.lblDesc = tk.Label(self, text='自我简介')
 self.lblEmail.grid(row=0, column=0, sticky=tk.E)
 self.lblPass1.grid(row=1, column=0, sticky=tk.E)
 self.lblPass2.grid(row=2, column=0, sticky=tk.E)
 self.lblDesc.grid(row=3, column=0, sticky=tk.NE)
 self.entryEmail = tk.Entry(self)
 self.entryPass1 = tk.Entry(self, show='*')
 self.entryPass2 = tk.Entry(self, show='*')
 self.textDesc = tk.Text(self, width=20, height=5)
 self.entryEmail.grid(row=0, column=1, columnspan=2)
 self.entryPass1.grid(row=1, column=1, columnspan=2)
 self.entryPass2.grid(row=2, column=1, columnspan=2)
 self.textDesc.grid(row=3, column=1, columnspan=2)
 self.btnOk = tk.Button(self, text='注册', command=self.funcOK)
 self.btnCancel = tk.Button(self, text='取消', command=root.destroy)
 self.btnOk.grid(row=4, column=1, sticky=tk.E)
 self.btnCancel.grid(row=4, column=2, sticky=tk.W)
 def funcOK(self):
 str1='欢迎注册:\n'
```

```
#导入 tkinter 模块
#定义 GUI 应用程序类,派生于 Frame 类
#构造函数, master 为父窗口
#调用父类的构造函数
#调用组件的 pack 方法,调整其显示位置和大小
#调用对象方法,创建子组件
#对象方法:创建子组件
#创建 Label 组件 - 用户名
#创建 Label 组件 - 密码
#创建 Label 组件 - 确认密码
#创建 Label 组件 - 自我简介
#Email 标签放置 0 行 0 列
#密码标签放置 1 行 0 列
#确认密码标签放置 2 行 0 列
#自我简介标签放置 3 行 0 列
#创建 Entry 组件
#密码显示为 *
#确认密码显示为 *
#创建 Text 组件
#用户名文本框放置 0 行 1 列
#密码文本框放置 1 行 1 列
#确认密码文本框放置 2 行 1 列
#自我简介文本框放置 3 行 1 列
#创建按钮组件
#注册按钮放置 4 行 1 列
#创建按钮组件
#取消按钮放置 4 行 2 列
#定义注册事件处理程序
```



```

 str1 += "您的账户为:" + self.entryEmail.get() + '\n'
 str1 += "您的特长为:\n" + self.textDesc.get(0.0,tk.END)
 tk.messagebox.showinfo("注册",str1) #弹出消息框
root = tk.Tk() #创建 1 个 Tk 根窗口组件 root
root.title('新用户注册') #设置窗口标题
app = Application(master = root) #创建 Application 的对象实例
app.mainloop() #调用组件的 mainloop 方法,进入事件循环

```

## 16.6.7 单选按钮 Radiobutton

Radiobutton（单选按钮）控件用于选择同一组单选按钮中的一个单选按钮（不能同时选定多个）。Radiobutton 可显示文本，也可显示图像：

**w = tkinter.Radiobutton( parent, option = value, ... )**

其中，parent 为父组件；option 为选项，通过如下命令可列举其选项：

```

>>> rb = Radiobutton() ; rb.keys()
['activebackground', 'activeforeground', 'anchor', 'background', 'bd', 'bg', 'bitmap', 'borderwidth', 'command',
'compound', 'cursor', 'disabledforeground', 'fg', 'font', 'foreground', 'height', 'highlightbackground', 'highlightcolor',
'highlightthickness', 'image', 'indicatoron', 'justify', 'offrelief', 'overrelief', 'padx', 'pady', 'relief',
'selectcolor', 'selectimage', 'state', 'takefocus', 'text', 'textvariable', 'tristateimage', 'tristatevalue', 'underline',
'value', 'variable', 'width', 'wraplength']

```

特定于 Radiobutton 的属性包括以下几种。

**command**：指定 Radiobutton 的回调函数（改变状态时调用）。

**indicatoron**：默认为 1，选择状态为句点；如果设置为 0，则选择状态为按下按钮。

**offrelief**：指定未选时的 3D 样式和值。默认为 tkinter.RAISED。

**selectcolor** 和 **selectimage**：选择时的颜色（默认为 red'）和图像（可指定）。

**value**：指定选择时值。默认为 1。

**variable**：绑定 IntVar 或 StringVar 对象到的选择值。一组单选按钮绑定到同一个对象。绑定值与 value 值相同时，选择该按钮，其他按钮不选中。

创建和设置 Radiobutton 组件可参考 16.5 节。例如：

```

>>> from tkinter import *
>>> root = Tk()
>>> v = StringVar() ; v.set('M') #创建 StringVar 对象,并设置初始值
>>> w1 = Radiobutton(root, text = "男", value = 'M', variable = v)
>>> w2 = Radiobutton(root, text = "女", value = 'F', variable = v)
>>> w1.pack(side = LEFT) # 男
>>> w2.pack(side = LEFT) # 女
>>> v.get() #选择女后,获取其值:'F'

```

Radiobutton 包括下列对象方法。

**flash()**：按钮在 active color 和 normal color 之间闪烁几次，disabled 状态无效。

**invoke()**：调用按钮的 command 指定的回调函数，disabled 状态无效。

**deselect()**：取消选择。

**select()**：选择。



toggle(): 切换选择状态。

## 16.6.8 复选框 Checkbutton

Checkbutton (复选框) 控件用于选择一项或多项选项 (可以同时选定多个)。Checkbutton 可显示文本, 也可显示图像:

```
w = tkinter.Checkbutton(parent, option = value, ...)
```

其中, parent 为父组件; option 为选项, 通过如下命令可列举其选项:

```
>>> cb = Checkbutton(); cb.keys()
['activebackground', 'activeforeground', 'anchor', 'background', 'bd', 'bg', 'bitmap', 'borderwidth', 'command',
'compound', 'cursor', 'disabledforeground', 'fg', 'font', 'foreground', 'height', 'highlightbackground', 'highlightcolor',
'highlightthickness', 'image', 'indicatoron', 'justify', 'offrelief', 'offvalue', 'onvalue', 'overrelief', 'padx',
'pady', 'relief', 'selectcolor', 'selectimage', 'state', 'takefocus', 'text', 'textvariable', 'tristateimage',
'tristatevalue', 'underline', 'variable', 'width', 'wraplength']
```

特定于 Checkbutton 的属性包括以下几种。

command: 指定 Checkbutton 的回调函数 (改变状态时调用)。

indicatoron: 默认为 1, 选择状态为打勾; 设置为 0, 则选择状态为按下按钮。

selectcolor 和 selectimage: 选择时的颜色 (默认为 'red') 和图像 (可指定)。

offrelief: 指定未选时的 3D 样式和值。默认为 tkinter.RAISED。

offvalue 和 onvalue: 指定未选中和选中时值。默认分别为 0 和 1。

variable: 绑定 IntVar 对象到的选择状态值 offvalue 和 onvalue。

创建和设置 Checkbutton 组件可参考 16.5 节。例如:

```
>>> v = StringVar() #创建 StringVar 对象
>>> v.set('yes') #设置默认值为'yes', 对应选择状态
>>> w = Checkbutton(root, text = "音乐", variable = v, onvalue = 'yes', offvalue = 'no')
>>> w.pack() # ☒ 音乐
>>> v.get() #用户去选后, 获取其值为'no': ☐ 音乐
```

Checkbutton 包括下列对象方法。

flash(): 按钮在 active color 和 normal color 之间闪烁几次, disabled 状态无效。

invoke(): 调用按钮的 command 指定的回调函数, disabled 状态无效。

deselect(): 取消选择。

select(): 选择。

**【例 16-9】** Radiobutton 和 Checkbox 示例 (Questionnaire.py): 实现 Questionnaire 调查个人信息。运行效果如图 16-20 所示。

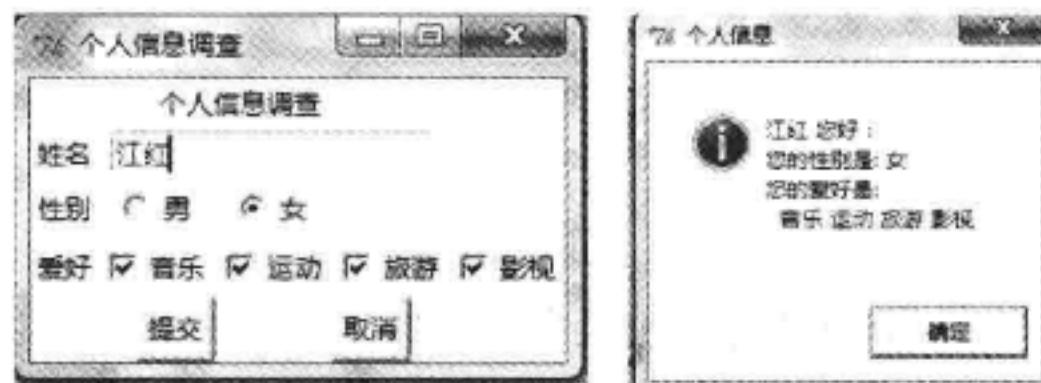


图 16-20 Radiobutton 和 Checkbox 运行效果

```

import tkinter as tk #导入 tkinter 模块
class Application(tk.Frame): #定义 GUI 应用程序类,派生于 Frame 类
 def __init__(self, master=None): #构造函数, master 为父窗口
 tk.Frame.__init__(self, master) #调用父类的构造函数
 self.grid() #调用组件的 pack 方法,调整其显示位置和大小
 self.createWidgets() #调用对象方法,创建子组件
 def createWidgets(self): #对象方法:创建子组件
 self.lblTitle = tk.Label(self, text='个人信息调查') #个人信息调查标签
 self.lblName = tk.Label(self, text='姓名') #姓名标签
 self.lblSex = tk.Label(self, text='性别') #性别标签
 self.lblHobby = tk.Label(self, text='爱好') #爱好标签
 self.lblTitle.grid(row=0, column=0, columnspan=4) #个人信息标签置于 0 行 0 列跨 4 列
 self.lblName.grid(row=1, column=0) #姓名标签置于 1 行 0 列
 self.lblSex.grid(row=2, column=0) #性别标签置于 2 行 0 列
 self.lblHobby.grid(row=3, column=0) #爱好标签置于 3 行 0 列
 #文本框
 self.entryName = tk.Entry(self) #创建 Entry 组件,姓名
 self.entryName.grid(row=1, column=1, columnspan=3) #姓名文本框置于 1 行 1 列
 #单选按钮
 self.vSex = tk.StringVar() #创建 StringVar 对象,性别
 self.vSex.set('M') #设置初始值:男性
 self.radioSexM = tk.Radiobutton(self, text="男", value='M', variable=self.vSex) #单选按钮
 self.radioSexF = tk.Radiobutton(self, text="女", value='F', variable=self.vSex)
 self.radioSexM.grid(row=2, column=1) #男性单选按钮置于 2 行 1 列
 self.radioSexF.grid(row=2, column=2) #女性单选按钮置于 2 行 2 列
 #复选框
 self.vHobbyMusic = tk.IntVar() #创建 IntVar 对象:爱好音乐
 self.vHobbySports = tk.IntVar() #创建 IntVar 对象:爱好运动
 self.vHobbyTravel = tk.IntVar() #创建 IntVar 对象:爱好旅游
 self.vHobbyMovie = tk.IntVar() #创建 IntVar 对象:爱好影视
 self.checkboxMusic = tk.Checkbutton(self, text="音乐", variable=self.vHobbyMusic) #音乐
 self.checkboxSports = tk.Checkbutton(self, text="运动", variable=self.vHobbySports) #运动
 self.checkboxTravel = tk.Checkbutton(self, text="旅游", variable=self.vHobbyTravel) #旅游
 self.checkboxMovie = tk.Checkbutton(self, text="影视", variable=self.vHobbyMovie) #影视
 self.checkboxMusic.grid(row=3, column=1) #音乐复选框置于 3 行 1 列
 self.checkboxSports.grid(row=3, column=2) #运动复选框置于 3 行 2 列
 self.checkboxTravel.grid(row=3, column=3) #旅游复选框置于 3 行 3 列
 self.checkboxMovie.grid(row=3, column=4) #影视复选框置于 3 行 4 列
 #按钮
 self.btnOk = tk.Button(self, text='提交', command=self.funcOK) #创建提交按钮组件
 self.btnOk.grid(row=4, column=1, sticky=tk.E) #提交按钮置于 4 行 1 列
 self.btnCancel = tk.Button(self, text='取消', command=root.destroy) #创建取消按钮组件
 self.btnCancel.grid(row=4, column=3, sticky=tk.W) #取消按钮置于 4 行 3 列

```

```

def funcOK(self): #定义提交事件处理程序
 strSex = '男' if (self.vSex.get() == 'M') else '女'
 strMusic = self.checkboxMusic['text'] if (self.vHobbyMusic.get() == 1) else "
 strSports = self.checkboxSports['text'] if (self.vHobbySports.get() == 1) else "
 strTravel = self.checkboxTravel['text'] if (self.vHobbyTravel.get() == 1) else "
 strMovie = self.checkboxMovie['text'] if (self.vHobbyMovie.get() == 1) else "
 str1 = self.entryName.get() + '您好:\n'
 str1 += "您的性别是:" + strSex + '\n'
 str1 += '您的爱好是:\n ' + strMusic + " + strSports + " + strTravel + " + strMovie
 tk.messagebox.showinfo("个人信息",str1) #弹出消息框
root = tk.Tk() #创建 1 个 Tk 根窗口组件 root
root.title('个人信息调查') #设置窗口标题
app = Application(master = root) #创建 Application 的对象实例
app.mainloop() #调用组件的 mainloop 方法,进入事件循环

```

## 16.6.9 列表框 Listbox

Listbox（列表框）用于显示对象列表，并且允许用户选择一个或多个项。

**w = tkinter. Listbox( parent, option = value, ...)**

其中，parent 为父组件；option 为选项，通过如下命令可列举其选项：

```

>>> lb = Listbox() ; lb.keys()
['activestyle', 'background', 'bd', 'bg', 'borderwidth', 'cursor', 'disabledforeground', 'exportselection', 'fg',
'font', 'foreground', 'height', 'highlightbackground', 'highlightcolor', 'highlightthickness', 'relief', 'selectback-
ground', 'selectborderwidth', 'selectforeground', 'selectmode', 'setgrid', 'state', 'takefocus', 'width', 'xscroll-
command', 'yscrollcommand', 'listvariable']

```

特定于 Listbox 的属性包括以下几种。

**activestyle**：指定选择项目的样式，取值为 'underline'、'dotbox'、'none'。

**selectmode**：指定选择项目模式，取值为 tk.BROWSE，单选，可鼠标拖动；tk.SINGLE，单选；tk.MULTIPLE，多选；tk.EXTENDED，多选，可鼠标拖选。

**listvariable**：绑定 StringVar 对象到的列表框。

Listbox 包括下列方法，用于处理其项目。

**w.insert( index, \* elements)**：在 index 位置插入 1 个或多个项目。

**w.delete( first, last = None)**：删除指定范围的项目，不指定 last 时，删除 1 个项目。

**w.selection\_set( first, last = None)**：选择指定范围的项目。

**w.selection\_clear( first, last = None)**：取消选择指定范围的项目。

**w.size()**：返回列表框中项目的个数。

**w.curselection()**：返回当前选择项目的索引，结果为元组。

**w.get( first, last = None)**：返回指定位置范围的项目，没有指定 last 时，返回 1 个项目。

**w.selection\_includes( index)**：判断一个项是否被选中，使用索引。

**w.index(i)**：显示位置 i 的项目。

其中，index 为位置索引，从 0 开始，表示第 1 个项目。tkinter 模块定义了若干特殊的位置索引，其中包括：ACTIVE（当前选中位置）、END（最后一个位置之后）等。



Listbox 组件还支持其他高级功能,包括滚动条等。限于篇幅,本书没有展开,具体请读者参考 tkinter 手册。

例如:

```
>>> from tkinter import *
>>> root = Tk()
>>> v = StringVar()
>>> v.set(('linux','windows','unix'))
>>> lb = Listbox(root,selectmode = EXTENDED,listvariable = v)
>>> lb.pack()
>>> for item in ['python','tkinter','widget']:lb.insert(END,item)
>>> lb.curselection() #选择项目的索引位置:('2','3')
>>> for i in lb.curselection():print(lb.get(i),end = "#输出选择项目:unix python
```

运行效果如图 16-21 所示。

**【例 16-10】** Listbox 示例 (Listbox.py): 实现列表选择功能。运行效果如图 16-22 所示。



图 16-21 列表框

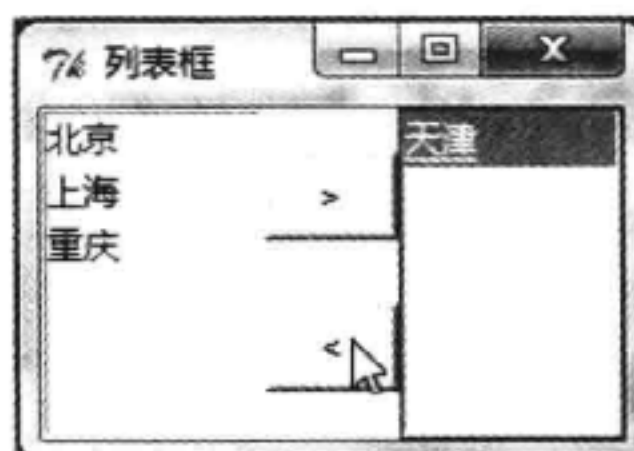
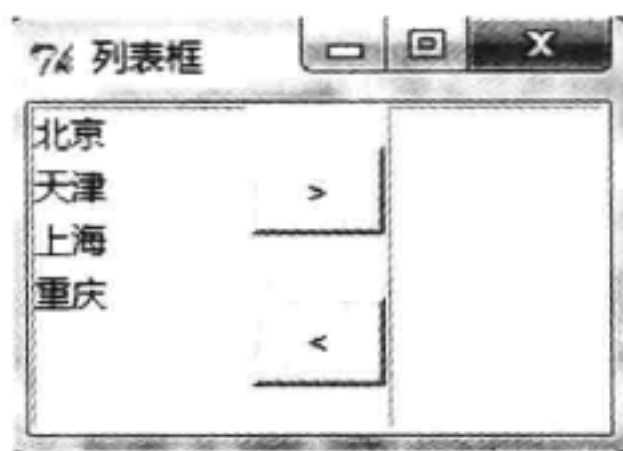


图 16-22 列表选择运行效果

|                                                                           |                           |
|---------------------------------------------------------------------------|---------------------------|
| import tkinter as tk                                                      | #导入 tkinter 模块            |
| class Application(tk.Frame):                                              | #定义 GUI 应用程序类,派生于 Frame 类 |
| def __init__(self, master = None):                                        | #构造函数, master 为父窗口        |
| tk.Frame.__init__(self, master)                                           | #调用父类的构造函数                |
| self.grid()                                                               | #调用组件的 pack 方法,调整其显示位置和大小 |
| self.createWidgets()                                                      | #调用对象方法,创建子组件             |
| def createWidgets(self):                                                  | #对象方法:创建子组件               |
| self.listboxLeft = tk.Listbox(self, width = 10, height = 6)               | #创建 Listbox 组件            |
| self.listboxLeft.insert(0,'北京','天津','上海','重庆')                            | #插入数据                     |
| self.listboxLeft.grid(row = 0, column = 0, rowspan = 5)                   | #置于 0 行 0 列跨 5 行          |
| self.listboxRight = tk.Listbox(self, width = 10, height = 6)              | #创建 Listbox 组件            |
| self.listboxRight.grid(row = 0, column = 2, rowspan = 5)                  | #0 行 2 列跨 5 行             |
| #按钮                                                                       |                           |
| self.btnToRight = tk.Button(self, text = '>', command = self.funcToRight) | #创建按钮组件                   |
| self.btnToRight.grid(row = 1, column = 1)                                 | #置于 1 行 1 列               |
| self.btnToLeft = tk.Button(self, text = '<', command = self.funcToLeft)   | #创建按钮组件                   |
| self.btnToLeft.grid(row = 3, column = 1)                                  | #置于 3 行 1 列               |



```

def funcToRight(self): #定义事件处理程序:在右边列表框显示左边列表框选中的内容
 for item in self.listBoxLeft.curselection(): #选中的内容
 self.listBoxRight.insert(tk.END,self.listBoxLeft.get(item)) #插入到右边列表框
 for item in self.listBoxLeft.curselection():
 self.listBoxLeft.delete(item) #从左边列表框——删除选中的内容
def funcToLeft(self): #定义事件处理程序:在左边列表框显示右边
 列表框选中的内容
 for item in self.listBoxRight.curselection(): #选中的内容
 self.listBoxLeft.insert(tk.END,self.listBoxRight.get(item)) #插入左边列表框
 for item in self.listBoxRight.curselection():
 self.listBoxRight.delete(item) #从右边列表框——删除选中的内容
root = tk.Tk() #创建 1 个 Tk 根窗口组件 root
root.title('列表框') #设置窗口标题
app = Application(master = root) #创建 Application 的对象实例
app.mainloop() #调用组件的 mainloop 方法,进入事件循环

```

### 16.6.10 选择项 OptionMenu

OptionMenu（选择项）允许用户选择一个项的列表框（在用户请求时显示）。用户单击下拉按钮可显示列表框，选择的内容会显示在顶部文本框中。

**w = tk.OptionMenu(parent, variable, choice1, choice2, ...)**

其中，parent 为父组件；variable 是选择项绑定的 StringVar 对象；choice1、choice2 等是选择项。例如：

```

>>> root = Tk()
>>> v = StringVar(root); v.set('Python')
>>> om = OptionMenu(root, v, 'Python', 'Perl', 'JavaScript', 'VBScript')
>>> om['width'] = 10; om['anchor'] = W; om.pack()
>>> om.keys() #OptionMenu 的选项
['activebackground', 'activeforeground', 'anchor', 'background', 'bd', 'bg', 'bitmap', 'borderwidth',
'cursor', 'direction', 'disabledforeground', 'fg', 'font', 'foreground', 'height', 'highlightbackground',
'highlightcolor', 'highlightthickness', 'image', 'indicatoron', 'justify', 'menu', 'padx', 'pady', 'relief',
'compound', 'state', 'takefocus', 'text', 'textvariable', 'underline', 'width', 'wraplength']

```

运行效果如图 16-23 所示。

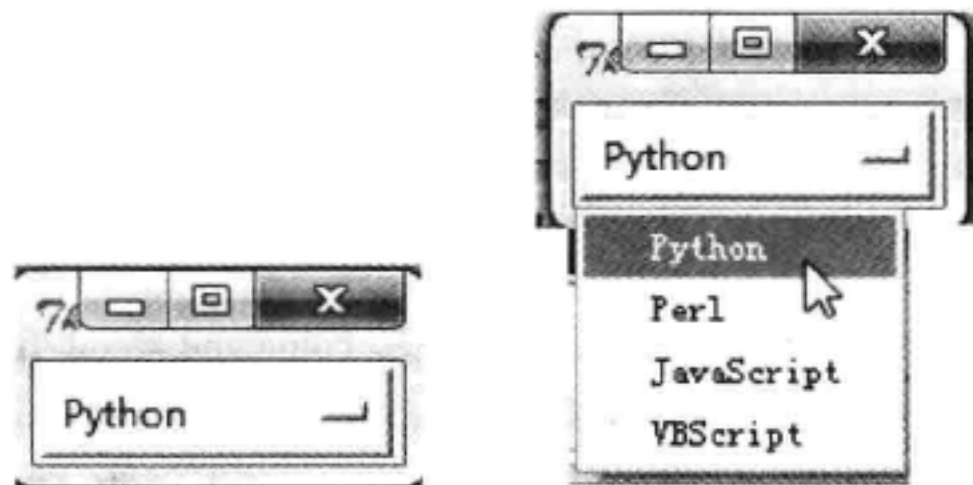


图 16-23 选择项

【例 16-11】OptionMenu 示例 (OptionMenu.py)。从组合框中选择字体大小, 然后单击“改变字体”按钮, 改变标签文本的字体大小。运行效果如图 16-24 所示。

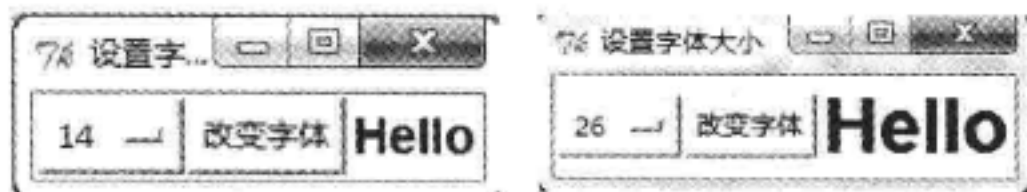


图 16-24 OptionMenu 运行效果

```
import tkinter as tk #导入 tkinter 模块
class Application(tk.Frame): #定义 GUI 应用程序类,派生于 Frame 类
 def __init__(self, master = None): #构造函数, master 为父窗口
 tk.Frame.__init__(self, master) #调用父类的构造函数
 self.grid() #调用组件的 pack 方法,调整其显示位置和大小
 self.createWidgets() #调用对象方法,创建子组件
 def createWidgets(self): #对象方法:创建子组件
 #创建 Scale 组件
 optionList = range(10,61,4)
 self.vFont = tk.StringVar()
 self.vFont.set(14) #设置初始值
 self.optionMenuFont = tk.OptionMenu(self, self.vFont, *optionList)
 self.optionMenuFont.pack(side = tk.LEFT)
 self.buttonFont = tk.Button(self, text = '改变字体', command = self.changeFont)
 self.buttonFont.pack(side = tk.LEFT)
 self.lblTitle = tk.Label(self, text = 'Hello', font = ('Helvetica', 14, 'bold')) #创建 Label 组件
 self.lblTitle.pack(side = tk.LEFT)
 def changeFont(self): #定义事件处理程序:改变字体
 fontNew = ('Helvetica', self.vFont.get(), 'bold')
 self.lblTitle.config(font = fontNew)

root = tk.Tk() #创建 1 个 Tk 根窗口组件 root
root.title('设置字体大小') #设置窗口标题
root['width'] = 400; root['height'] = 50 #设置窗口宽、高
app = Application(master = root) #创建 Application 的对象实例
app.mainloop() #调用组件的 mainloop 方法,进入事件循环
```

### 16.6.11 移动滑块 Scale

Scale (移动滑块) 控件用于在有界区间内, 通过移动滑块来选择值。

**w = tkinter.Scale( parent, option = value, ... )**

其中, parent 为父组件; option 为选项, 通过如下命令可列举其选项:

```
>>> scale = Scale(); scale.keys()
['activebackground', 'background', 'bigincrement', 'bd', 'bg', 'borderwidth', 'command', 'cursor',
'digits', 'fg', 'font', 'foreground', 'from', 'highlightbackground', 'highlightcolor', 'highlightthickness', 'la-
bel', 'length', 'orient', 'relief', 'repeatdelay', 'repeatinterval', 'resolution', 'showvalue', 'sliderlength',
'sliderrelief', 'state', 'takefocus', 'tickinterval', 'to', 'troughcolor', 'variable', 'width']
```

特定于 Scale 的属性包括以下几种。

command: 指定回调函数, 带 1 个参数, 滑块的新值。

digits: 显示滑块值的位数。

from\_ 和 to: 指定滑块的最小值和最大值, 默认为 0.0 和 100.0。

label: 设置 Scale 的标签属性。

length 和 width: 长和宽。

orient: 指定滑块的方向, 取值为 HORIZONTAL (默认值) 或 VERTICAL。

repeatinterval 和 repeatdelay: 重复间隔和时延, 可设置长按按钮时产生重复事件。

resolution: 指定滑块的步长精度, 即步距。

showvalue: 指定是否显示滑块当前值文本。

sliderlength: 设置滑块的长度, 默认为 30 个像素。

sliderrelief: 设置滑块的 3D 样式。

tickinterval: 指定滑块刻度间隔。

variable: 绑定 IntVar、DoubleVar、StringVar 对象到滑块。

Scale 组件对象包括下列方法, 用于处理设置或获取滑块当前值。

- w.get()           #返回滑块当前值
- w.set(value)      #设置滑块当前值

【例 16-12】 Scale 示例 (Scale.py)。移动滑块, 改变字体大小。运行效果如图 16-25 所示。



图 16-25 Scale 运行效果

```
import tkinter as tk #导入 tkinter 模块
class Application(tk.Frame): #定义 GUI 应用程序类,派生于 Frame 类
 def __init__(self, master = None): #构造函数, master 为父窗口
 tk.Frame.__init__(self, master) #调用父类的构造函数
 self.grid() #调用组件的 pack 方法,调整其显示位置和大小
 self.createWidgets() #调用对象方法,创建子组件
 def createWidgets(self): #对象方法:创建子组件
 #创建 Scale 组件
 self.scaleFont = tk.Scale(self, from_ = 10, to = 60, length = 400,
 orient = tk.HORIZONTAL, command = self.changeFont)
 self.scaleFont.set(20) #设置初始值
 self.scaleFont.pack()
 self.lblTitle = tk.Label(self, text = 'Hello', font = ('Helvetica', 20, 'bold')) #创建 Label 组件
 self.lblTitle.pack()
 def changeFont(self, value): #定义事件处理程序:改变字体
 fontNew = ('Helvetica', self.scaleFont.get(), 'bold')
 self.lblTitle.config(font = fontNew)
```



|                                                       |                            |
|-------------------------------------------------------|----------------------------|
| <code>root = tk.Tk()</code>                           | #创建 1 个 Tk 根窗口组件 root      |
| <code>root.title('设置字体大小')</code>                     | #设置窗口标题                    |
| <code>root['width'] = 400; root['height'] = 50</code> |                            |
| <code>app = Application(master = root)</code>         | #创建 Application 的对象实例      |
| <code>app.mainloop()</code>                           | #调用组件的 mainloop 方法, 进入事件循环 |

### 16.6.12 顶层窗口 Toplevel

Toplevel（顶层窗口）是直接由窗口管理器管理的窗口，顶层窗口独立于其他窗口，可以创建任意数量的顶层窗口。

`w = tkinter.Toplevel(master = None, option, ...)`

其中，master 为父组件，默认为无；option 为选项，通过如下命令可列举其选项。

```
>>> tl = Toplevel(); tl.keys()
['bd', 'borderwidth', 'class', 'menu', 'relief', 'screen', 'use', 'background', 'bg', 'colormap', 'container', 'cursor', 'height', 'highlightbackground', 'highlightcolor', 'highlightthickness', 'padx', 'pady', 'takefocus', 'visual', 'width']
```

Toplevel 组件对象包括下列方法。

w. `aspect(nmin, dmin, nmax, dmax)`：设置窗口的宽长比范围为  $[n_{\min}/d_{\min}, n_{\max}/d_{\max}]$ 。

w. `iconify()`、w. `deiconify()`：最小化窗口、取消窗口最小化。

w. `lift(aboveThis = None)`：前移窗口。

w. `lower(belowThis = None)`：后移窗口。

w. `maxsize(width = None, height = None)`：设置窗口的最大值。

w. `minsize(width = None, height = None)`：设置窗口的最小值。

w. `resizable(width = None, height = None)`：设置宽、长为可调整大小。

w. `state(newstate = None)`：窗口状态，取值为 'normal'（正常）、'iconic'（图标化）、'withdrawn'（隐藏）。

w. `title(text = None)`：窗口标题。

w. `transient(parent = None)`：设置为 parent 的 transient 窗口。

w. `withdraw()`：隐藏窗口。

【例 16-13】使用 Toplevel，实现自定义关于对话框（MyDialog.py）。运行效果如图 16-26 所示。

|                                                                                      |                 |
|--------------------------------------------------------------------------------------|-----------------|
| <code>import tkinter as tk</code>                                                    | #导入 tkinter 模块  |
| <code>class MyDialog:</code>                                                         | #自定义对话框         |
| <code>def __init__(self, master):</code>                                             |                 |
| <code>self.top = tk.Toplevel(master)</code>                                          | #生成 Toplevel 组件 |
| <code>self.label1 = tk.Label(self.top, text = '版权所有')</code>                         | #创建标签组件         |
| <code>self.label1.pack()</code>                                                      |                 |
| <code>self.label2 = tk.Label(self.top, text = 'V 1.0.0')</code>                      | #创建标签组件         |
| <code>self.label2.pack()</code>                                                      |                 |
| <code>self.buttonOK = tk.Button(self.top, text = 'OK', command = self.funcOk)</code> | #创建按钮           |
| <code>self.buttonOK.pack()</code>                                                    |                 |



```

def funcOk(self) :
 self. top. destroy() #销毁对话框
class Application(tk. Frame) : #定义 GUI 应用程序类,派生于 Frame 类
 def __init__(self, master = None) : #构造函数, master 为父窗口
 tk. Frame. __init__(self, master) #调用父类的构造函数
 self. pack() #调用组件的 pack 方法,调整其显示位置和大小
 self. createWidgets() #调用对象方法,创建子组件
 def createWidgets(self) : #对象方法:创建子组件
 self. btnAbout = tk. Button(self, text = " About" , command = self. funcAbout)
 self. btnAbout. pack() #调用组件的 pack 方法,调整其显示位置和大小
 def funcAbout(self) : #定义事件处理程序
 d = MyDialog(self) #创建对话框
root = tk. Tk() #创建 1 个 Tk 根窗口组件 root
root['width '] = 400; root['height '] = 50 #设置窗口宽、高
app = Application(master = root) #创建 Application 的对象实例
app. mainloop()

```

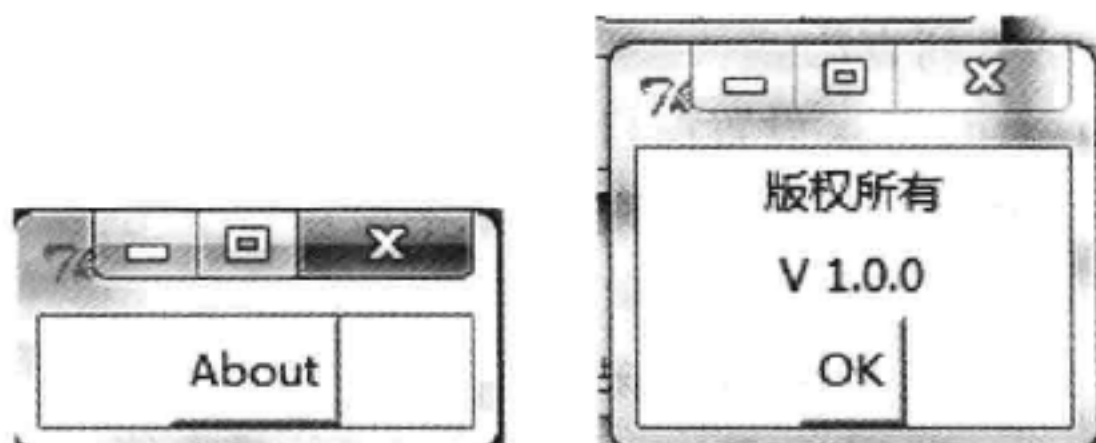


图 16-26 自定义关于对话框的运行效果

### 16.6.13 ttk 子模块控件

tkinter 模块包括子模块 ttk, ttk 包含了 tkinter 缺少的基本控件: Combobox、Progressbar、Notebook、Treeview 等,使 tkinter 更实用。ttk 还支持控件呈现操作系统本地化风格,在 Windows 下像 Windows,在 Mac osx 下像 Mac,在 LINUX 下像 LINUX。

限于篇幅,本书没有展开阐述,具体请读者参考 tkinter 参考手册。

## 16.7 对话框

对话框用于与用户交互和检索信息。tkinter 模块中的子模块 messagebox、filedialog、colorchooser、simpledialog,包括一些通用的预定义对话框;用户也可以通过继承 TopLevel 创建自定义对话框。

### 16.7.1 通用消息对话框

模块 tkinter 的子模块 messagebox 包含以下若干用于打开消息对话框的函数。

askokcancel( title = None, message = None, \* \* options ): OK/Cancel 对话框。

askquestion( title = None, message = None, \* \* options ): Yes/No 问题对话框。

`askretrycancel(title = None, message = None, * * options)`: Retry/Cancel 对话框。

`askyesno(title = None, message = None, * * options)`: Yes/No 是/否对话框。

`showerror(title = None, message = None, * * options)`: 错误消息对话框。

`showinfo(title = None, message = None, * * options)`: 信息消息对话框。

`showwarning(title = None, message = None, * * options)`: 警告消息对话框。

其中, `title` 是弹出对话框窗口的标题; `message` 是对话框中显示的内容, 使用转义字符“\n”可多行显示。命名参数 `options` 指定以下各种选项。

`default = C`: 默认按钮。取值为模块常量 `CANCEL`、`IGNORE`、`OK`、`NO`、`RETRY`、`YES`。默认为 `CANCEL` 按钮。

`icon = I`: 图标。取值为模块常量 `ERROR`、`INFO`、`QUESTION`、`WARNING`。

`parent = W`: 父窗口。默认为根窗口。

`askokcancel`、`askretrycancel` 和 `askyesno` 返回 `bool` 值: 选择 `OK` 或 `Yes` 按钮时返回 `True`, 选择 `No` 或 `Cancel` 时返回 `False`。`askquestion` 返回字符串: 选择 `Yes` 时返回 `u'yes'`, 选择 `No` 时返回 `u'no'`。例如:

```
>>> from tkinter.messagebox import *
>>> askokcancel(title='askokcancel',message='是否放弃修改的内容?')
>>> askquestion(title='askquestion',message='是否放弃修改的内容?')
>>> askyesno(title='askyesno',message='是否放弃修改的内容?')
>>> askretrycancel(title='askretrycancel',message='系统忙,是否重试?')
>>> showerror(title='showerror',message='无法连接!')
>>> showinfo(title='showinfo',message='连接成功!')
>>> showwarning(title='showwarning',message='磁盘碎片过多!')
```

运行结果分别如图 16-27 所示。

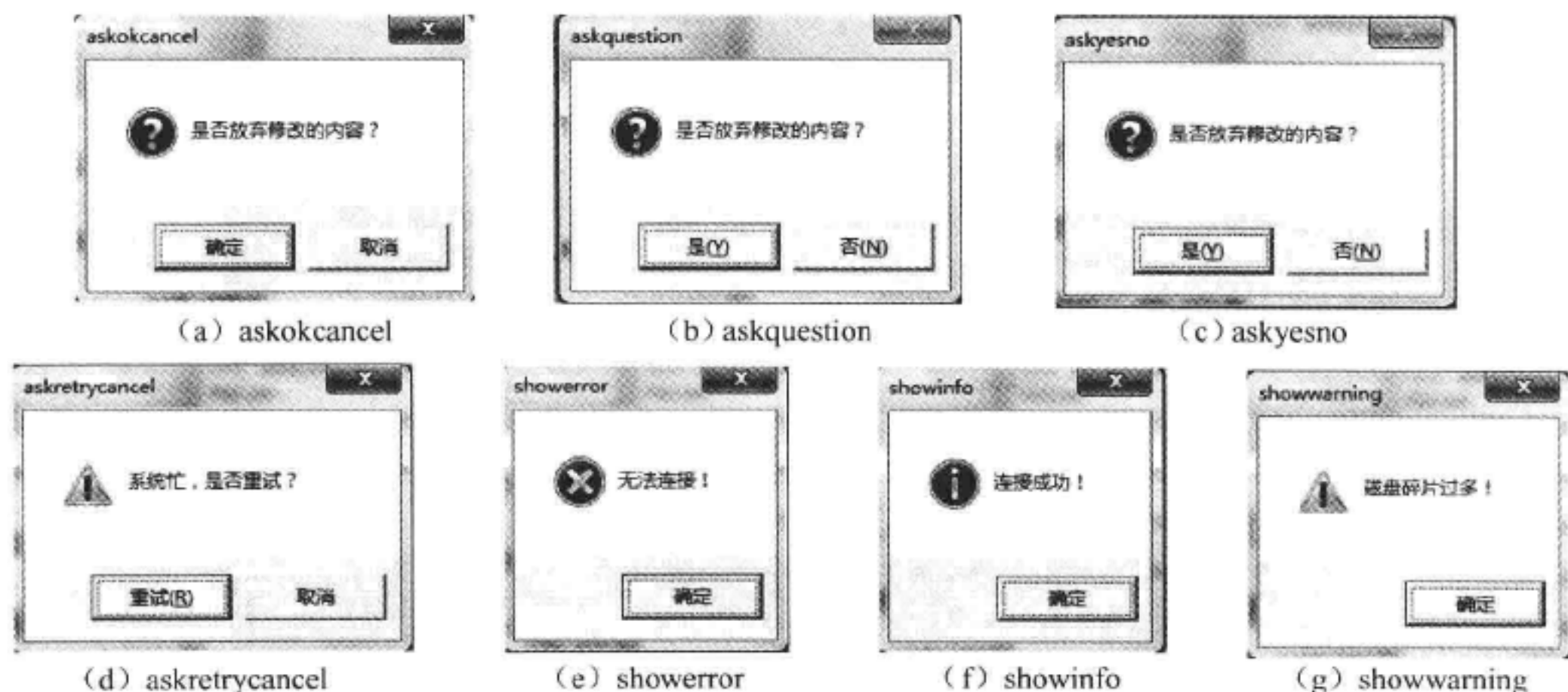


图 16-27 通用消息对话框

## 16.7.2 文件对话框

模块 `tkinter` 的子模块 `filedialog` 包含若干用于打开文件对话框的函数。

`askdirectory( ** options )`: 打开目录对话框, 返回目录名。

`askopenfile( ** options )`: 打开文件对话框, 返回打开的文件对象。

`askopenfile( ** options )`: 打开文件对话框, 返回打开的文件对象列表。

`askopenfilename( ** options )`: 打开文件对话框, 返回打开的文件名。

`askopenfilenames( ** options )`: 打开文件对话框, 返回打开的文件名列表。

`asksaveasfile( mode = 'w', ** options )`: 打开保存对话框, 返回保存的文件对象。

`asksaveasfilename( mode = 'w', ** options )`: 打开保存对话框, 返回保存的文件名。

使用命名参数 `options` 指定以下各种选项。

`defaultextension = s`: 默认后缀 `.xxx`。用户没有输入后缀自动添加。

`filetypes = [ (label1, pattern1), (label2, pattern2), ... ]`: 文件过滤器。

`initialdir = D`: 初始目录。

`initialfile = F`: 初始文件。

`parent = W`: 父窗口。默认为根窗口。

`title = T`: 窗口标题。

例如:

```
>>> from tkinter.filedialog import *
```

```
>>> askopenfilename(title = 'askopenfilename', filetypes = [('Python 源文件', '.py')])
```

运行结果如图 16-28 所示。

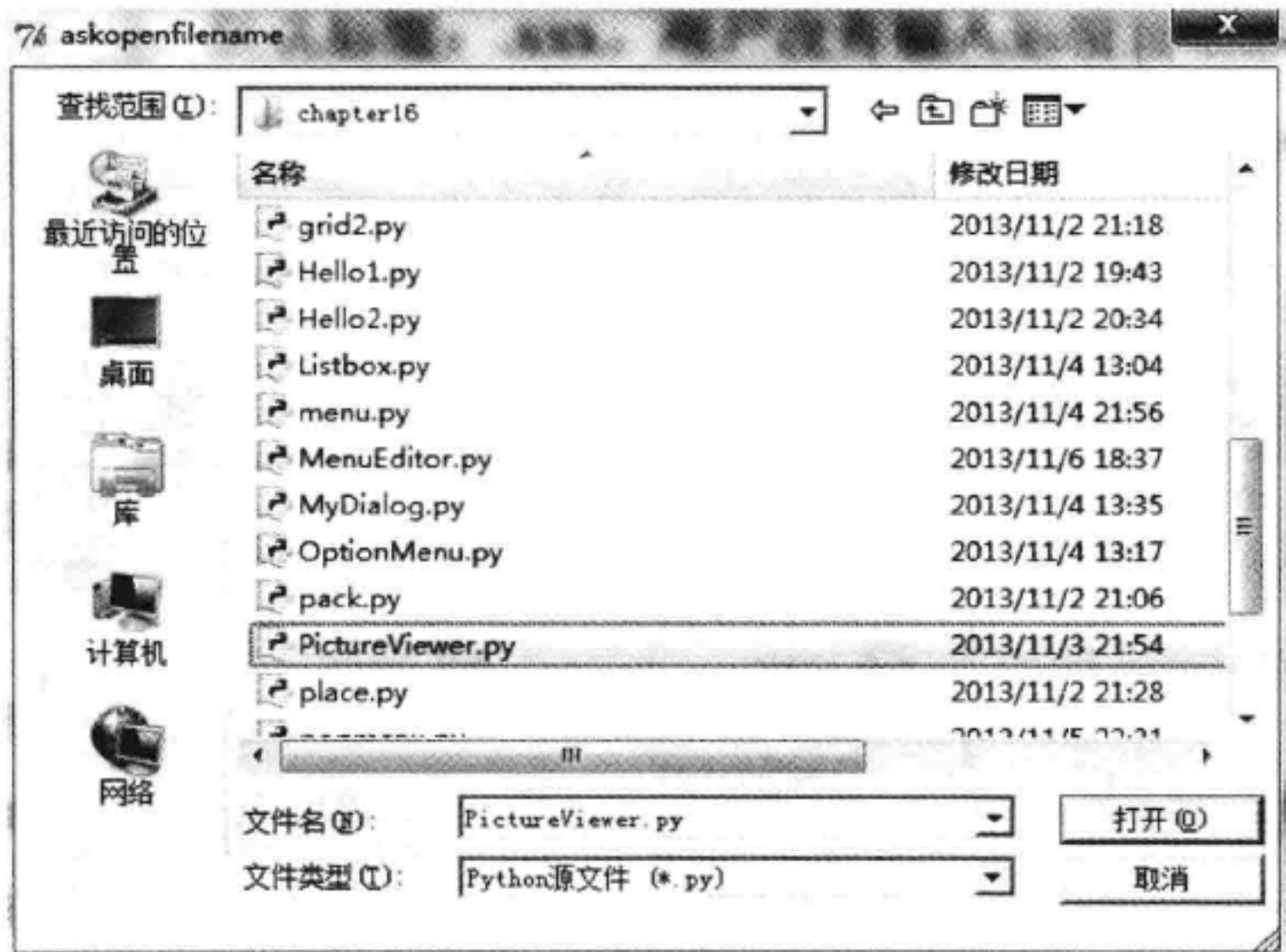


图 16-28 文件对话框



### 16.7.3 颜色选择对话框

模块 `tkinter` 的子模块 `colorchooser` 包含用于打开颜色选择对话框的函数：

`askcolor( color = None, * * options)` #打开颜色选择对话框

其中，`color` 为初始颜色；命名参数 `options` 指定以下各种选项。

`parent = W`：父窗口。默认为根窗口。

`title = T`：窗口标题。

`askcolor()` 返回 `((R, G, B), color)`。例如：

```
>>> from tkinter.colorchooser import *
>>> askcolor(color = 'red', title = 'askcolor') #((0.0,0.0,255.99609375), '#0000ff')
```

运行效果如图 16-29 所示。

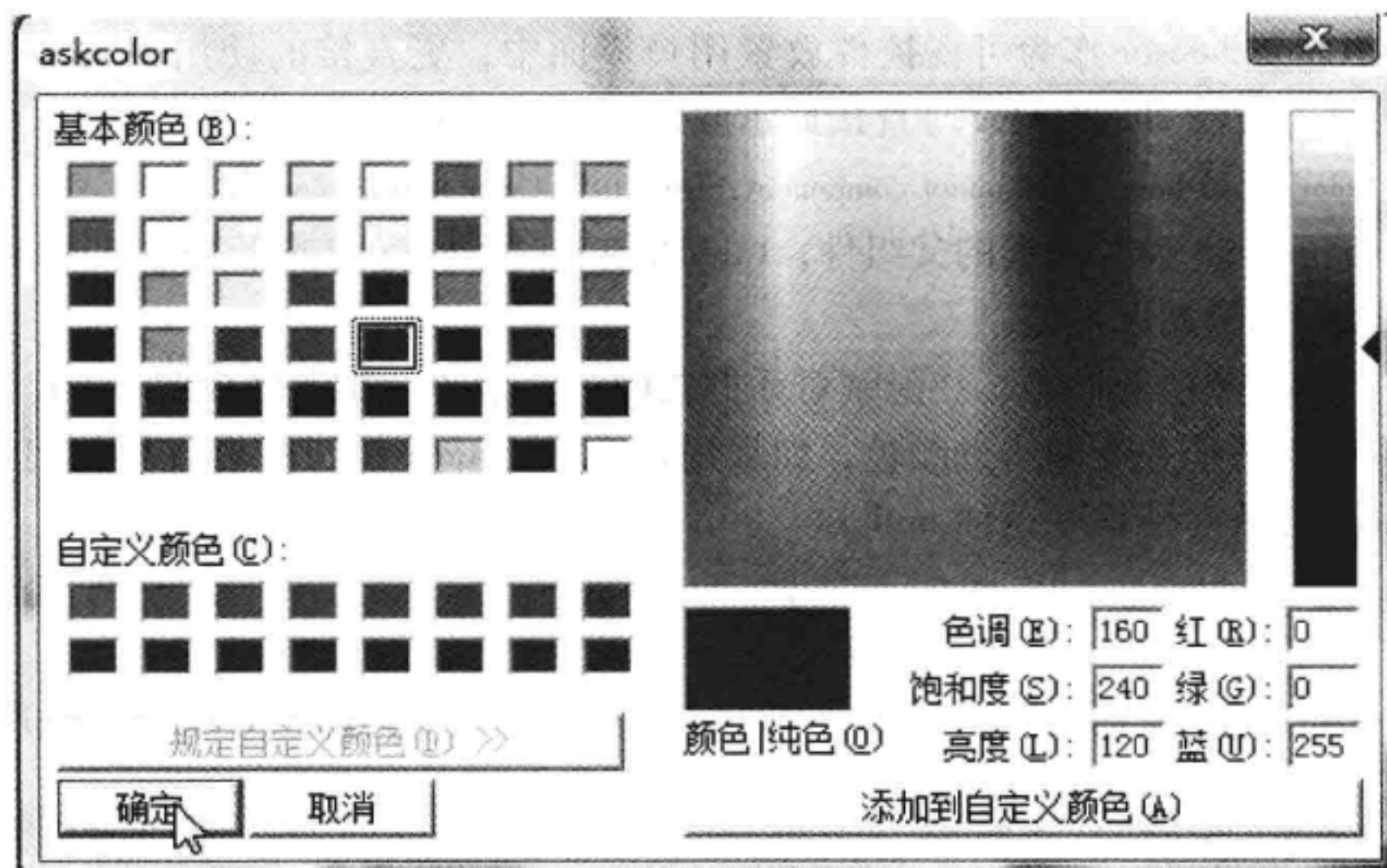


图 16-29 颜色选择对话框

`JColorChooser` 提供一个用于允许用户操作和选择颜色的控制器窗格。`JColorChooser` 可以作为控件放置在任何自定义的界面当中，也可以作为单独的对话框使用。

`JColorChooser` 使用 `ColorSelectionModel` 数据模型来管理当前颜色选择状态。

#### 1. 构造方法

`JColorChooser()`：创建初始颜色为白色的颜色选取器窗格。

`JColorChooser( Color initialColor)`：创建具有指定初始颜色的颜色选取器窗格。

#### 2. 设置和获取颜色

`void setColor( Color color)`：设置当前颜色。

`void setColor( int r, int g, int b)`：设置当前颜色。

`void setColor( int c)`：设置当前颜色（低 8 位为 Blue 值，接着 8 位为 Green 值，高 8 位为 Red 值）。



Color getColor(): 获取颜色选取器的当前颜色值。

### 3. 注册事件监听器

使用 JColorChooser 对象的 getSelectionModel() 可以获得其 ColorSelectionModel 数据模型对象, 使用 ColorSelectionModel 数据模型对象的 addChangeListener() 方法可以添加颜色更改事件监听器。

ColorSelectionModel getSelectionModel(): 返回处理颜色选择的数据模型。

void addChangeListener (ChangeListener listener): 在模型中添加颜色更改事件监听器。

例如:

```
tcc. getSelectionModel(). addChangeListener(this);
public void stateChanged(ChangeEvent e) { Color newColor = tcc. getColor(); #颜色处理 }
```

### 4. 作为对话框显示颜色选择窗格

除了将 JColorChooser 作为可视控件放置用户界面中, 更直接的使用方法是使用 JColorChooser 类的静态方法 showDialog() 直接显示颜色选择对话框。

```
static Color showDialog(Component component, String title, Color initialColor)
```

其中, component 为对话框的父组件; title 为包含对话框标题的字符串; initialColor 为显示颜色选取器时的初始颜色值。

显示有模式的颜色选取器, 在隐藏对话框之前一直阻塞。如果用户按下 OK 按钮, 则此方法隐藏/释放对话框并返回所选颜色; 如果用户按下 Cancel 按钮或者直接关闭对话框, 则此方法将隐藏/释放对话框并返回 null。

## 16.7.4 通用对话框应用举例

【例 16-14】通用对话框应用示例 (DialogEditor. py)。运行效果如图 16-30 所示。

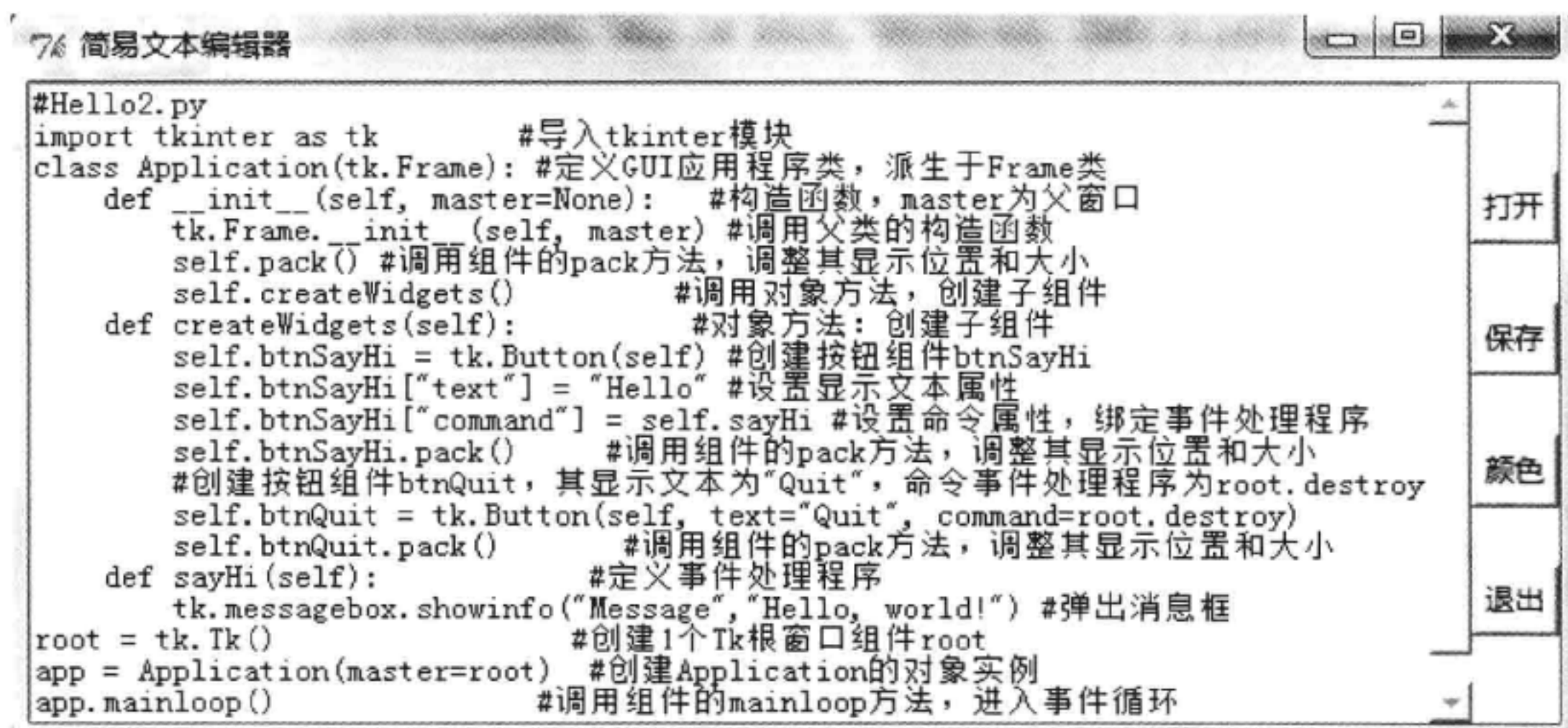


图 16-30 通用对话框的运行效果

```

import tkinter as tk #导入 tkinter 模块
import tkinter.scrolledtext as tst

class Application(tk.Frame): #定义 GUI 应用程序类,派生于 Frame 类
 def __init__(self, master=None): #构造函数, master 为父窗口
 tk.Frame.__init__(self, master) #调用父类的构造函数
 self.grid() #调用组件的 pack 方法,调整其显示位置和大小
 self.createWidgets() #调用对象方法,创建子组件
 def createWidgets(self): #对象方法:创建子组件
 self.textEdit = tst.ScrolledText(self, width=80, height=20) #创建 Text 组件
 self.textEdit.grid(row=0, column=0, rowspan=6) #文本框置于 0 行 0 列
 self.btnOpen = tk.Button(self, text='打开', command=self.funcOpen) #创建按钮组件
 self.btnOpen.grid(row=1, column=1) #打开按钮置于 1 行 1 列
 self.btnSave = tk.Button(self, text='保存', command=self.funcSave)
 #创建按钮组件
 self.btnSave.grid(row=2, column=1) #保存按钮置于 2 行 1 列
 self.btnColor = tk.Button(self, text='颜色', command=self.funcColor)
 #创建按钮组件
 self.btnColor.grid(row=3, column=1) #颜色按钮置于 3 行 1 列
 self.btnQuit = tk.Button(self, text='退出', command=self.funcQuit)
 #创建按钮组件
 self.btnQuit.grid(row=4, column=1) #退出按钮置于 4 行 1 列
 def funcOpen(self): #定义事件处理程序:打开文件
 self.textEdit.delete(1.0, tk.END) #清空 Text 组件的内容
 fname = tk.filedialog.askopenfilename(filetypes=[('Python 源文件', '.py')])
 with open(fname, 'r', encoding='utf-8') as f1: #打开文件
 str1 = f1.read() #读入文件内容
 self.textEdit.insert(0.0, str1) #插入内容到 Text 组件
 def funcSave(self): #定义事件处理程序:保存文件
 str1 = self.textEdit.get(1.0, tk.END)
 fname = tk.filedialog.asksaveasfilename(filetypes=[('Python 源文件', '.py')])
 with open(fname, 'w', encoding='utf-8') as f1: #打开文件
 f1.write(str1)
 def funcColor(self): #定义事件处理程序:设置颜色
 t, c = tk.colorchooser.askcolor(title='askcolor')
 self.textEdit.config(bg=c)
 def funcQuit(self): #定义事件处理程序:退出程序
 root.destroy() #退出程序

root = tk.Tk() #创建 1 个 Tk 根窗口组件 root
root.title('简易文本编辑器') #设置窗口标题
app = Application(master=root) #创建 Application 的对象实例
app.mainloop() #调用组件的 mainloop 方法,进入事件循环

```

16.7.5 简单对话框

模块 `tkinter` 的子模块 `simpdialog` 中，包含用于打开输入对话框的函数。  
`askfloat(title, prompt, * * kw)`：打开输入对话框，输入并返回浮点数。  
`askinteger(title, prompt, * * kw)`：打开输入对话框，输入并返回整数。  
`askstring(title, prompt, * * kw)`：打开输入对话框，输入并返回字符串。  
其中，`title` 为窗口标题，`prompt` 为提示文本信息；命名参数 `kw` 指定各种选项，包括：  
`initialvalue`（初始值）、`minvalue`（最小值）和 `maxvalue`（最大值）。例如：

```
>>> root = Tk()
>>> from tkinter.simpdialog import *
>>> askinteger(title='请输入',prompt='请输入整数:',initialvalue=100) #133
>>> askfloat(title='请输入',prompt='请输入实数:') #33.3
>>> askstring(title='请输入',prompt='请输入字符串:') #'hello,world'
```

测试运行效果如图 16-31 所示。

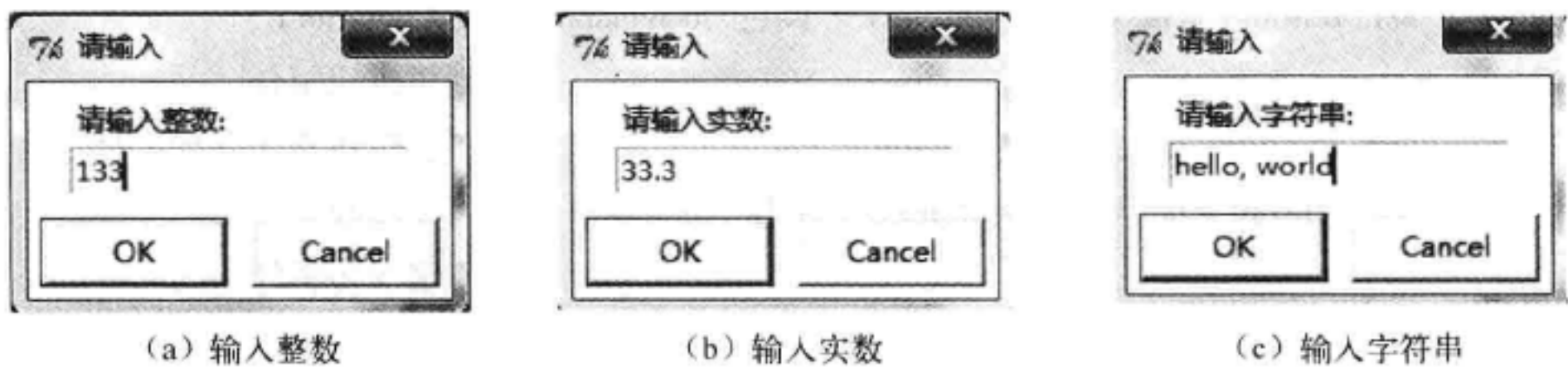


图 16-31 简单对话框 (1)

`tkinter` 的子模块 `simpdialog` 包含 `SimpleDialog` 组件，其构造函数为：  
`SimpleDialog ( master, text = '' , buttons = [ ] , default = None , cancel = None , title = None , class _ = None )`  
其中，`master` 是父窗口；`buttons` 为要显示的按钮列表；`default` 为默认选中的按钮；`title` 为窗口标题。例如：

```
>>> root = Tk()
>>> dlg = SimpleDialog(root , text = '继续?' , buttons = ['Yes ' , 'No ' , 'cancel '] , default = 0)
```

运行效果如图 16-32 所示。

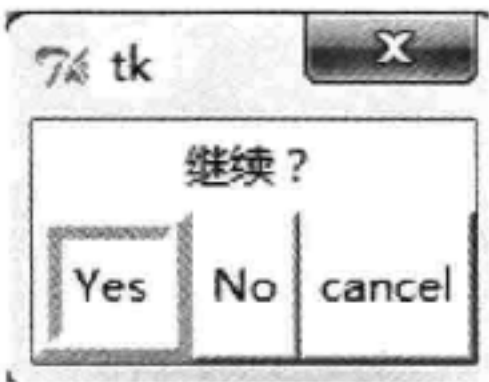


图 16-32 简单对话框 (2)



## 16.8 菜单和工具栏

图形用户界面应用程序通常提供菜单，菜单包括各种按照主题分组的基本命令。图形用户界面应用程序包括三种类型的菜单。

主菜单：提供窗体的菜单系统。通过单击可下拉出子菜单，选择命令可执行相关的操作。常用的主菜单通常包括：文件、编辑、视图、帮助等。

上下文菜单（也称为快捷菜单）：通过鼠标右击某对象而弹出的菜单，一般为与该对象相关的常用菜单命令。例如：剪切、复制、粘贴等。

工具栏：提供窗体的工具栏。通过单击工具栏上的图标，可以执行相关的操作。

### 16.8.1 菜单组件 Menu

Menu（菜单）用于创建菜单。

**w = tk.Menu(mb, option, ...)**

其中，mb 为父组件；option 为选项，通过如下命令可列举其选项：

```
>>> m = Menu(); m.keys()
['activebackground', 'activeborderwidth', 'activeforeground', 'background', 'bd', 'bg', 'borderwidth',
'cursor', 'disabledforeground', 'fg', 'font', 'foreground', 'postcommand', 'relief', 'selectcolor',
'takefocus', 'tearoff', 'tearoffcommand', 'title', 'type']
```

特定于 Menu 的属性包括以下几种。

postcommand：指定选择菜单时要执行的命令。

tearoff：菜单是否可以拆分。

tearoffcommand：菜单拆分时要执行的命令。

title：可以拆分菜单的标题。

Menu 组件对象包括下列方法。

add(kind, coption, ...): 追加菜单项。kind 取值为：'cascade'（层叠菜单项）、'checkboxbutton'（复选框菜单项）、'command'（命令菜单项）、'radiobutton'（单选按钮选项）和 'separator'（分隔符）。

add\_cascade(coption, ...): 追加层叠菜单项。

add\_checkbox(coption, ...): 追加复选框菜单项。

add\_command(coption, ...): 追加命令菜单项。

add\_radiobutton(coption, ...): 追加单选按钮选项。

add\_separator(): 追加分隔符。

delete(index1, index2 = None): 删除指定范围的菜单项。

entrycget(index, coption): 获取指定菜单项的选项属性。

entryconfigure(index, coption, ...): 设置指定菜单项的选项属性。

index(i): 获取索引 i 的位置。

insert\_cascade(index, coption, ...): 插入层叠菜单项。

insert\_checkbox(index, coption, ...): 插入复选框菜单项。



insert\_command(index, coption, ...): 插入命令菜单项。  
 insert\_radiobutton(index, coption, ...): 插入单选按钮选项。  
 insert\_separator(index): 插入分隔符。  
 invoke(index): 调用索引位置的菜单命令。  
 post(x, y): 在坐标 (x, y) 处显示菜单。  
 type(index): 返回索引 index 处的菜单项的类型。  
 yposition(n): 返回第 n 个菜单选项的偏移值。

其中, index 为位置索引, 第 1 个项目从 0 开始。coption 包括: accelerator、activebackground、activeforeground、background、bitmap、columnbreak、columnbreak、command、compound、font、foreground、hidemargin、image、label、menu、offvalue、onvalue、selectcolor、selectimage、state、underline、value、variable。

## 16.8.2 创建主菜单

主菜单一般提供窗体的菜单系统。通过单击可下拉出子菜单, 选择命令以执行相关的操作。常用的主菜单通常包括: 文件、编辑、视图和帮助等。创建主菜单一般遵循下列步骤。

(1) 创建主菜单栏。例如:

```
menubar = tk.Menu(root) #创建主菜单栏 menubar
```

(2) 创建菜单, 并添加菜单到步骤 (1) 创建的菜单栏。例如:

```
menufile = tk.Menu(menubar) #创建菜单 filemenu
#把菜单 filemenu 作为层叠菜单添加到主菜单栏 menubar
menubar.add_cascade(label='File', menu=menufile)
```

(3) 添加菜单项到步骤 (2) 创建的菜单。例如:

```
menufile.add_command(label='Open') #在菜单 filemenu 中添加菜单项 Open
menufile.add_command(label='Save') #在菜单 filemenu 中添加菜单项 Save
menufile.add_command(label='Print', accelerator='^P', command=f_print) #添加菜单项 Print
menufile.add_separator() #添加分隔符
menufile.add_command(label='Exit') #添加菜单项 Exit
```

(4) 将菜单栏添加到根窗体。例如:

```
root['menu'] = menubar #附加主菜单到根窗口
```

**【例 16-15】** 主菜单示例 (menu.py)。运行效果如图 16-33 所示。



图 16-33 菜单的运行效果

```

import tkinter as tk #导入 tkinter 模块
def f_print():
 tk.messagebox.showinfo('信息','打印功能')
root = tk.Tk() #创建 1 个 Tk 根窗口组件 root
#创建主菜单栏
menubar = tk.Menu(root) #创建主菜单栏 menubar
#创建子菜单
menufile = tk.Menu(menubar) #创建菜单 menufile
menuedit = tk.Menu(menubar,tearoff=0)
menuhelp = tk.Menu(menubar,tearoff=0)
menuTest = tk.Menu(menubar)
menubar.add_cascade(label='File',menu=menufile) #menufile 作为层叠菜单添加到主菜单栏
menubar.add_cascade(label="Edit",menu=menuedit)
menubar.add_cascade(label="Help",menu=menuhelp)
menubar.add_cascade(label="菜单 2",menu=menuTest)
#添加菜单项
menufile.add_command(label='Open') #菜单 menufile 中添加菜单项 Open
menufile.add_command(label='Save') #添加菜单项 Save
menufile.add_command(label='Print',accelerator='^P',command=f_print) #添加菜单项 Print
menufile.add_separator() #添加分隔符
menufile.add_command(label='Exit') #添加菜单项 Exit
menuedit.add_command(label="Cut") #在菜单 menuedit 中添加菜单项 Cut
menuedit.add_command(label="Copy") #添加菜单项 Copy
menuedit.add_command(label="Paste") #添加菜单项 Paste
menuhelp.add_command(label="About") #菜单 menuhelp 中添加菜单项 About
menuTest.add_command(label="菜单项 1") #菜单 menuTest 中添加菜单项 1
menuTest.add_command(label="菜单项 2") #添加菜单项 2
menuTest.add_separator() #添加分隔符
menuTest.add_checkbutton(label="复选框菜单项 1") #添加复选框菜单项 1
menuTest.add_checkbutton(label="复选框菜单项 2") #添加复选框菜单项 2
menuTest.add_separator() #添加分隔符
menuTest.add_radiobutton(label="单选按钮菜单项 1") #添加单选按钮菜单项 1
menuTest.add_radiobutton(label="单选按钮菜单项 2") #添加单选按钮菜单项 2
menuTest.add_separator() #添加分隔符
menusub = tk.Menu(menuTest) #创建子菜单
menuTest.add_cascade(label="子菜单",menu=menusub) #menusub 作为层叠菜单添加到菜单
menusub.add_command(label="子菜单项 1") #添加子菜单项 1
menusub.add_command(label="子菜单项 2") #添加子菜单项 2
#附加主菜单到根窗口
root['menu'] = menubar
root.mainloop() #调用组件的 mainloop 方法,进入事件循环

```

### 16.8.3 创建上下文菜单

上下文菜单（也称为快捷菜单）是通过鼠标右击某对象而弹出的菜单，一般为与该对

象相关的常用菜单命令。例如：剪切、复制、粘贴等。创建上下文菜单一般遵循下列步骤。

(1) 创建菜单（与创建主菜单相同）。例如：

```
menubar = tk.Menu(root)
```

```
menubar.add_command(label="Font")
```

(2) 绑定鼠标右击事件，并在事件处理函数中弹出菜单。例如：

```
def popup(event):
```

#事件处理函数

```
 menubar.post(event.x_root,event.y_root)
```

#在鼠标右键位置显示菜单

```
root.bind('<Button-3>',popup)
```

#绑定事件

【例 16-16】上下文菜单示例（popmenu.py）。运行效果如图 16-34 所示。

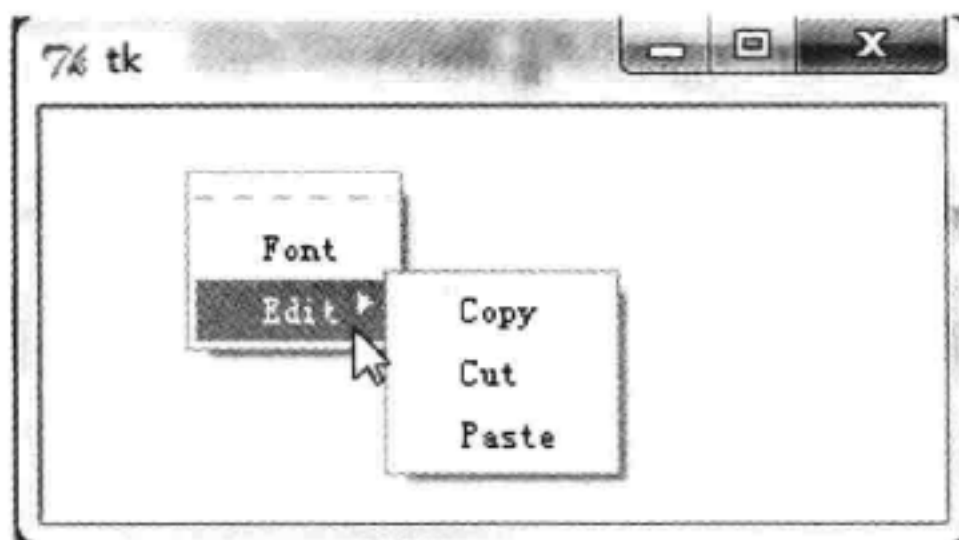


图 16-34 上下文菜单运行效果

```
import tkinter as tk
```

#导入 tkinter 模块

```
def popup(event):
```

#事件处理函数

```
 menubar.post(event.x_root,event.y_root)
```

#鼠标右键位置显示菜单

```
root = tk.Tk()
```

#创建 1 个 Tk 根窗口组件 root

#创建菜单

```
menubar = tk.Menu(root)
```

#创建菜单

```
menubar.add_command(label="Font")
```

```
menuedit = tk.Menu(menubar,tearoff=0)
```

```
menubar.add_cascade(label="Edit",menu=menuedit)
```

```
menuedit.add_command(label="Copy")
```

```
menuedit.add_command(label="Cut")
```

```
menuedit.add_command(label="Paste")
```

#创建界面

```
textEdit = tk.Text(root,width=40,height=10)
```

#创建 Text 组件

```
textEdit.pack()
```

```
root.bind('<Button-3>',popup)
```

#绑定事件

#附加主菜单到根窗口

```
root.mainloop()
```

#调用组件的 mainloop 方法,进入事件循环

## 16.8.4 菜单应用举例

【例 16-17】简单文本编辑器示例（MenuEditor.py）。运行效果如图 16-35 所示。





图 16-35 简单文本编辑器的运行效果

```

import tkinter as tk #导入 tkinter 模块
import tkinter.scrolledtext as tst

class Application(tk.Frame):
 #定义 GUI 应用程序类,派生于 Frame 类
 def __init__(self, master=None):
 #构造函数, master 为父窗口
 tk.Frame.__init__(self, master)
 #调用父类的构造函数
 self.grid()
 #调用组件的 grid 方法,调整其显示位置和大小
 self.createWidgets()
 #调用对象方法,创建子组件
 self.createMenu()
 #调用对象方法,创建菜单
 root['menu'] = self.menubar
 #附加主菜单到根窗口
 root.bind('<Button-3', self.f_popup)
 #绑定事件

 def createWidgets(self):
 #对象方法:创建子组件
 self.textEdit = tst.ScrolledText(self, width=80, height=20)
 #创建 Text 组件
 self.textEdit.grid(row=0, column=0, rowspan=6)
 #Text 组件置于 0 行 0 列跨 6 行

 def createMenu(self):
 #对象方法:创建菜单
 self.menubar = tk.Menu(root)
 #创建主菜单栏 menubar
 #创建子菜单
 self.menufile = tk.Menu(self.menubar)
 #创建菜单 menufile
 self.menuedit = tk.Menu(self.menubar, tearoff=0)
 #创建菜单 menuedit
 self.menuhelp = tk.Menu(self.menubar, tearoff=0)
 #创建菜单 menuhelp
 self.menubar.add_cascade(label='File', menu=self.menufile)
 self.menubar.add_cascade(label="Edit", menu=self.menuedit)
 self.menubar.add_cascade(label="Help", menu=self.menuhelp)
 #添加菜单项
 self.menufile.add_command(label='New', command=self.f_new)
 #File - New
 self.menufile.add_command(label='Open', command=self.f_open)
 #File - Open
 self.menufile.add_command(label='Save', accelerator='^A', command=self.f_save)
 #File - Save

```



```

self.menufile.add_separator() #分隔符
self.menufile.add_command(label='Exit',command=root.destroy) #File - Exit
self.menueit.add_command(label="Cut",command=self.f_cut) #Edit - Cut
self.menueit.add_command(label="Copy",command=self.f_copy) #Edit - Copy
self.menueit.add_command(label="Paste",command=self.f_paste) #Edit - Paste
self.menuhelp.add_command(label="About",command=self.f_about)
#Help - About

def f_new(self): #定义事件处理程序:File - New
 self.textEdit.delete(1.0,tk.END) #清空 Text 组件的内容
def f_open(self): #定义事件处理程序:File - Open
 self.textEdit.delete(1.0,tk.END) #清空 Text 组件的内容
 fname = tk.filedialog.askopenfilename(filetypes=[('Python 源文件','.py')])
 with open(fname,'r',encoding='utf-8') as fl: #打开文件
 str1 = fl.read() #读入文件内容
 self.textEdit.insert(0.0,str1) #插入内容到 Text 组件
def f_save(self): #定义事件处理程序:File - Save
 str1 = self.textEdit.get(1.0,tk.END) #获取 Text 组件中全部内容
 fname = tk.filedialog.asksaveasfilename(filetypes=[('Python 源文件','.py')])
 with open(fname,'w',encoding='utf-8') as fl: #打开文件
 fl.write(str1) #将 Text 组件中全部内容写入文件
def f_about(self): #定义事件处理程序:Help - About
 tk.messagebox.showinfo('关于','版本 V 1.0.1')
def f_cut(self): #定义事件处理程序>Edit - Cut
 try:
 str1 = self.textEdit.get(tk.SEL_FIRST,tk.SEL_LAST) #获取选择的内容
 self.textEdit.clipboard_clear() #清空剪贴板
 self.textEdit.clipboard_append(str1) #附加到剪贴板
 self.textEdit.delete(tk.SEL_FIRST,tk.SEL_LAST) #删除选择的内容
 except: pass
def f_copy(self): #定义事件处理程序>Edit - Copy
 try:
 str1 = self.textEdit.get(tk.SEL_FIRST,tk.SEL_LAST) #获取选择的内容
 self.textEdit.clipboard_clear() #清空剪贴板
 self.textEdit.clipboard_append(str1) #附加到剪贴板
 except: pass
def f_paste(self): #定义事件处理程序>Edit - Paste
 str1 = self.textEdit.selection_get(selection='CLIPBOARD') #获取剪贴板内容
 try: #使用剪贴板内容替换所选内容,否则插入剪贴板内容
 self.textEdit.replace(tk.SEL_FIRST,tk.SEL_LAST,str1)
 except:
 self.textEdit.insert(tk.INSERT,str1) #插入内容到当前位置
def f_popup(self,event): #事件处理函数
 self.menueit.post(event.x_root,event.y_root) #在鼠标右键位置显示菜单

```

```
root = tk.Tk() #创建 1 个 Tk 根窗口组件 root
root.title('简易文本编辑器') #设置窗口标题
app = Application(master=root) #创建 Application 的对象实例
app.mainloop() #调用组件的 mainloop 方法,进入事件循环
```

## 16.9 图形绘制

### 16.9.1 Canvas 组件

Canvas（画布）是一个长方形的区域，用于图形绘制或复杂的图形界面布局。可以在画布上绘制图形、文字，放置各种组件和框架。

```
w = tk.Canvas(parent, option = value, ...)
```

其中，master 为父组件，默认无；option 为选项，通过如下命令可列举其选项：

```
>>> c = Canvas(); c.keys()
['background', 'bd', 'bg', 'borderwidth', 'closeenough', 'confine', 'cursor', 'height', 'highlightback-
ground', 'highlightcolor', 'highlightthickness', 'insertbackground', 'insertborderwidth', 'insertofftime', 'in-
sertontime', 'insertwidth', 'offset', 'relief', 'scrollregion', 'selectbackground', 'selectborderwidth', 'select-
foreground', 'state', 'takefocus', 'width', 'xscrollcommand', 'xscrollincrement', 'yscrollcommand',
'yscrollincrement']
```

例如：

```
>>> from tkinter import *; root = Tk()
>>> c = Canvas(root, bg='white'); c.pack(); root.mainloop()
```

### 16.9.2 Canvas 上的对象

#### 1. 绘制对象

Canvas 组件对象包括下列方法（绘制函数），用于绘制各种图形对象。

create\_arc()：绘制圆弧。

create\_bitmap()：绘制位图。

create\_image()：绘制位图图像。

create\_line()：绘制线。

create\_oval()：绘制椭圆。

create\_polygon()：绘制多边形。

create\_rectangle()：绘制矩形。

create\_text()：绘制文本。

create\_window()：绘制子窗口。

#### 2. 绘制对象的标识 id

Canvas 上每个绘制对象都有一个标识 id（整数），使用绘制函数创建绘制对象时，返回其对象标识 id。例如：

```
>>> from tkinter import *
```

```
>>> root = Tk()
>>> c = Canvas(root, bg = 'white', width = 130, height = 70); c. pack()
>>> c. create_rectangle(10, 10, 100, 50) #1
```

运行结果如图 16-36 所示。

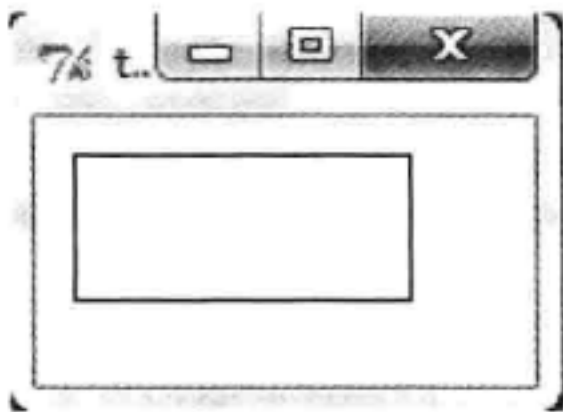


图 16-36 Canvas 上绘制矩形

### 3. 绘制对象的标签 tag

绘制对象可以使用标签 tag（字符串）标识。一个绘制对象可以对应 0 或 n 个 tag；一个 tag 可以对应 0 或 n 个绘制对象。绘制对象时可以指定其标签列表，例如：

```
>>> c. create_rectangle(10, 10, 100, 50, tags = 'rect')
>>> c. create_rectangle(10, 10, 100, 50, fill = 'blue', tags = ('rect', 'blue'))
```

运行结果如图 16-37 所示。

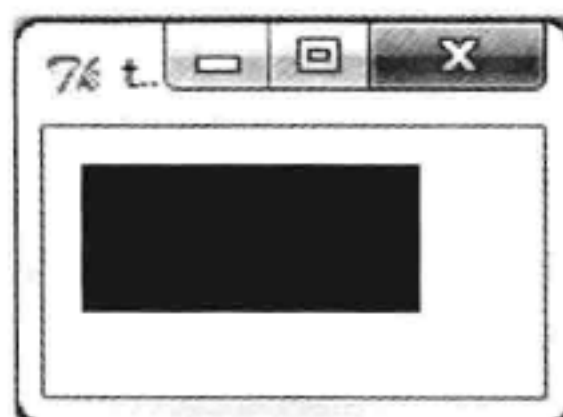


图 16-37 Canvas 上绘制填充矩形

也可使用 Canvas 组件的对象方法添加或删除标签。

addtag\_all( newTag )：为 Canvas 上的所有绘制对象添加标签 newTag。

addtag\_above( newTag, tagOrId )：为上面的对象添加标签。

addtag\_below( newTag, tagOrId )：为下面的对象添加标签。

addtag\_closest( newTag, x, y, halo = None, start = None )：为最近对象添加标签。

addtag\_enclosed( newTag, x1, y1, x2, y2 )：为区域包含的对象添加标签。

addtag\_overlapping( newTag, x1, y1, x2, y2 )：为与区域重叠的对象添加标签。

addtag\_withtag( newTag, tagOrId )：为指定对象添加标签。

dtag( tagOrId, tagToDelete )：删除指定对象的指定标签。

其中，tagOrId 可以为 id（整数）或 tag（字符串）。

### 4. 绘制对象列表

Canvas 上的所有绘制对象组成绘制对象列表。在同一个区域绘制图形，新绘制的图像对象在绘制对象列表中位于其他绘制对象之上。显示时，上面的绘制对象会覆盖下面的绘制的重叠部分。相关的对象方法如下。



`tag_lower(tagOrId,belowThis)`: 将指定对象移动到下面。

`tag_raise(tagOrId,aboveThis)`: 将指定对象移动到上面。

## 5. 查找绘制对象

Canvas 组件包含以下若干查找绘制对象的对象方法。

`find_all()`: 查找所有对象。

`find_below(tagOrId)`: 查找下面的对象。

`find_closest(x,y,halo=None,start=None)`: 查找最近的对象。

`find_enclosed(x1,y1,x2,y2)`: 查找区域包含的对象。

`find_overlapping(x1,y1,x2,y2)`: 查找区域重叠的对象。

`find_withtag(tagOrId)`: 查找指定标签的对象。

其中, `tagOrId` 可以为 `id` (整数) 或 `tag` (字符串)。

## 6. 删除对象

Canvas 组件对象包括以下方法, 用于删除已绘制的图形对象。

`delete(tagOrId)`: 删除指定标签或 `id` 的绘制对象。

**【例 16-18】** 查找和删除绘制对象 (`canvas_delete.py`)。运行结果如图 16-38 所示。

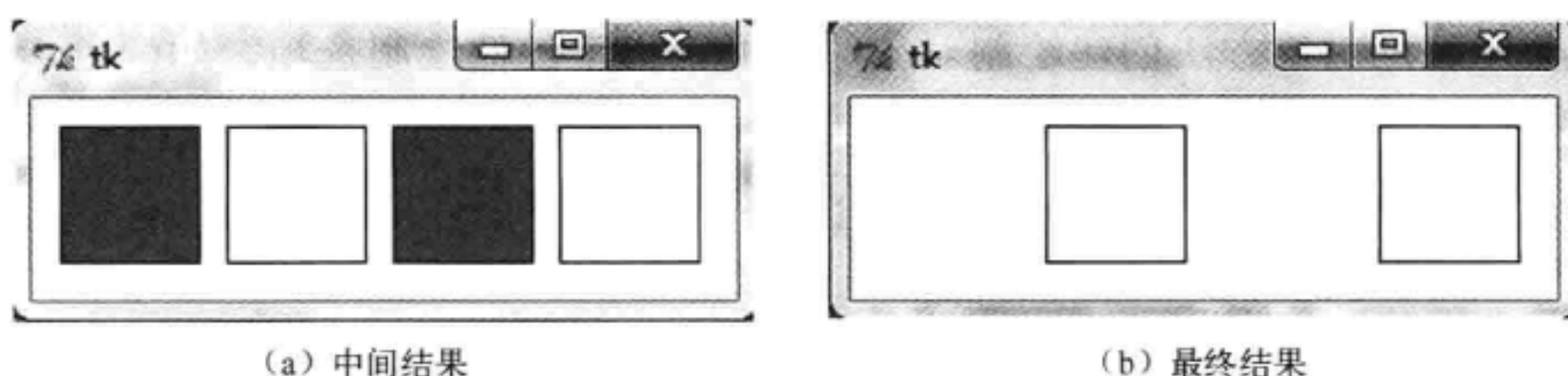


图 16-38 查找和删除绘制对象

```
from tkinter import *
root = Tk()
c = Canvas(root,bg='white',width=250,height=70);c.pack() #创建并显示 Canvas 组件
c.create_rectangle(10,10,60,60,fill='red',tags='red') #红色填充矩形
c.create_rectangle(70,10,120,60) # (无填充) 矩形框
c.create_rectangle(130,10,180,60,fill='red',tags='red') #红色填充矩形
c.create_rectangle(190,10,240,60) # (无填充) 矩形框
for i in c.find_withtag('red'): #查找带标签'red'的绘制对象
 c.delete(i) #删除绘制对象
```

### 16.9.3 绘制矩形

在 Canvas 对象 `c` 上绘制线条 (直线或折线) 的方法为:

`id = c.create_rectangle(x0,y0,x1,y1,option,...)`

其中,  $(x0,y0)$  是左上角的坐标;  $(x1,y1)$  是右下角的坐标; `option` 为选项, 包括: `activedash`、`activefill`、`activeoutline`、`activeoutlinestipple`、`activestipple`、`activewidth`、`dash`、`dashoffset`、`disableddash`、`disabledfill`、`disabledoutline`、`disabledoutlinestipple`、`disabledstipple`、



disabledwidth、fill、offset、outline、outlineoffset、outlinestipple、state、stipple、tags、width。

【例 16-19】矩形绘制示例（create\_rectangle.py）。运行结果如图 16-39 所示。

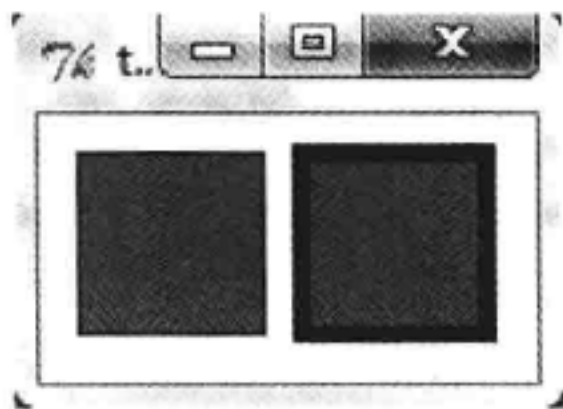


图 16-39 绘制矩形

```
from tkinter import *
root = Tk()
c = Canvas(root, bg = 'white', width = 130, height = 70); c.pack() #创建并显示 Canvas
c.create_rectangle(10,10,60,60,fill = 'red') #红色填充矩形
c.create_rectangle(70,10,120,60,fill = 'red',outline = 'blue',width = 5) #蓝色填充矩形,边框宽度 5
```

## 16.9.4 绘制选项

使用 Canvas 的对象方法创建绘制对象时，可以指定各种绘制选项，如颜色、边框等。也可以使用 Canvas 对象的方法 itemconfigure 设置：

itemconfigure (tagOrId, option, ...)：设置选项。

itemcget (tagOrId, option)：获取选项。

### 1. 填充颜色

绘制选项 fill、activefill 和 disabledfill，分别用于设置绘制图形的填充颜色、活动（鼠标经过）时的填充颜色和禁用时的填充颜色。例如：

```
>>> c.create_rectangle(10,10,110,110,fill = 'red') #红色填充矩形
```

### 2. 边框颜色和宽度

绘制选项 outline、activeoutline 和 disabledoutline，分别用于设置绘制图形的边框颜色、活动（鼠标经过）时的边框颜色和禁用时的边框颜色。绘制选项 width 用于设置边框的宽度。例如：

```
>>> c.create_rectangle(10,10,110,110,outline = 'blue',width = 5) #边框蓝色、宽度 5 的矩形
```

### 3. 虚线样式

绘制选项 dash 用于设置绘制图形边框的虚线样式，dashoffset 用于设置虚线的间隔。activedash 和 disableddash 用于设置活动禁用时的虚线样式。dash 的值为元组，元组指定交叉的实线像素和空白像素。dashoffset 则指定从元组的哪个元素开始。

【例 16-20】虚线样式示例（create\_option\_dash.py）。运行效果如图 16-40 所示。

```
from tkinter import *
root = Tk()
c = Canvas(root, bg = 'white', width = 250, height = 70); c.pack() #创建并显示 Canvas
c.create_rectangle(10,10,60,60,dash = (4,2))#虚线为 4 个像素点紧跟 2 个像素空格
```

```
c. create_rectangle(70,10,120,60,dash=(4,))#虚线为 4 个像素点紧跟 4 个像素空格
#虚线为 4 个像素点跟 4 个像素空格跟 2 个像素点跟 2 个像素空格
c. create_rectangle(130,10,180,60,dash=(4,2,2,4))#按指定虚线样式绘制矩形框
#虚线为 2 个像素点跟 2 个像素空格跟 4 个像素点跟 4 个像素空格
c. create_rectangle(190,10,240,60,dash=(4,2,2,4),dashoffset=2) #按指定虚线样式绘制矩形框
```

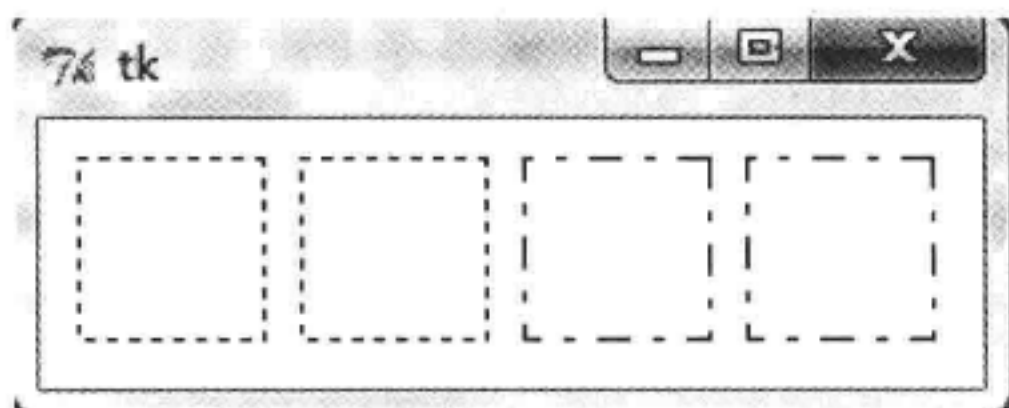


图 16-40 虚线样式

#### 4. 填充画刷

绘制选项 `stipple` 用于设置填充画刷（位图对象），`outlinestipple` 用于设置边框画刷（位图对象）。内置的位图对象参见 16.5.9 节。`offset` 和 `outlineoffset` 用于设置画刷的对齐方式，取值为：'x, y'、'#x, y'、tk.NE、tk.SE、tk.SW、tk.NW、tk.N、tk.E、tk.S、tk.W 和 tk.CENTER。对应的 `activestipple`、`activeoutlinestipple`、`disabledstipple` 和 `disabledoutlinestipple` 分别用于设置活动禁用时的画刷。

【例 16-21】填充画刷示例（`create_option_stipple.py`）。运行效果如图 16-41 所示。

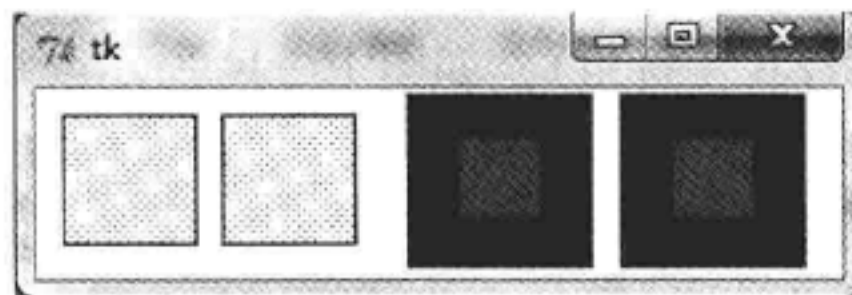


图 16-41 填充画刷

```
from tkinter import *
root = Tk()
c = Canvas(root,bg='white',width=300,height=70);c.pack() #创建并显示 Canvas
#按指定填充画刷绘制各类红色填充矩形
c.create_rectangle(10,10,60,60,fill='red',stipple='gray12')
c.create_rectangle(70,10,120,60,fill='red',stipple='gray12',offset=NE)
c.create_rectangle(150,10,200,60,fill='red',outline='blue',width=20,outlinestipple='error')
c.create_rectangle(230,10,280,60,fill='red',outline='blue',width=20,outlinestipple='error',
offset=NE)
```

#### 5. 箭头和箭头样式

绘制选项 `arrow` 用于设置线条是否带箭头（默认不带箭头），取值为：tk.FIRST（开始端带箭头）、tk.LAST（结束端带箭头）、tk.BOTH（两端带箭头）。

绘制选项 `arrowshape` 用于设置线条箭头样式，`arrowshape` 取值为元组：(d1, d2, d3)，如图 16-42 所示，默认值为 (8, 10, 3)。

【例 16-22】箭头和箭头样式示例（create\_option\_arrow.py）。运行效果如图 16-43 所示。

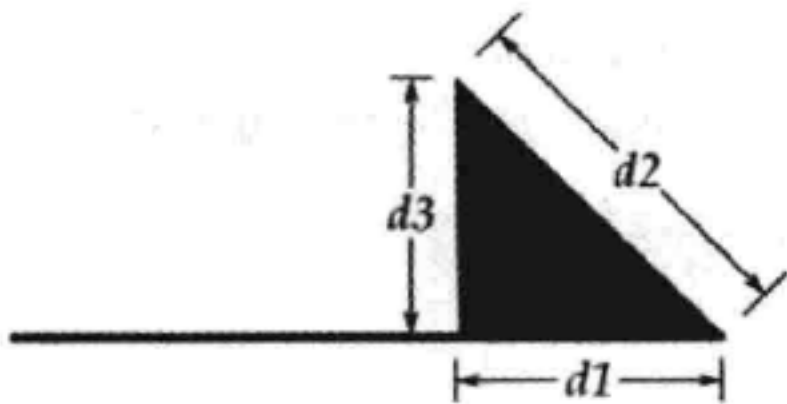


图 16-42 arrowshape 取值

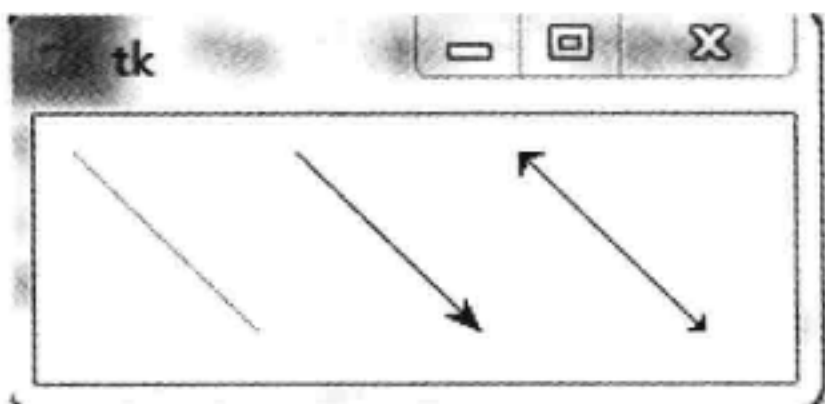


图 16-43 箭头和箭头样式

```
from tkinter import *
root = Tk()
c = Canvas(root, bg = 'white', width = 200, height = 70); c.pack()
c.create_line(10,10,60,60,fill = 'red')
c.create_line(70,10,120,60,arrow = LAST)
c.create_line(130,10,180,60,arrow = BOTH,arrowshape = (3,5,4))
```

#创建并显示 Canvas  
#绘制红线  
#绘制结束端带箭头的线  
#绘制两端带箭头的线

6. 顶点连接样式和光滑度

cap style 用于设置线条端点的样式，取值为：tk.BUTT（方形，边在顶点上）、tk.PROJECTING（方形，边在顶点延伸 1/2 宽度处）和 tk.ROUND（圆形），如图 16-44 所示。

绘制选项 join style 用于设置顶点 2 个线条连接方式，取值为：tk.ROUND（圆角）、tk.BEVEL（斜面角）、tk.MITER（斜角），如图 16-44 所示。

绘制选项 smooth 为 0（False）时，顶点连接处为直线；为 1（True）时，顶点连接处为曲线，且可使用 splinesteps（默认值 12）指定曲线包括线段。

【例 16-23】顶点连接样式示例（create\_option\_style.py）。运行效果如图 16-45 所示。

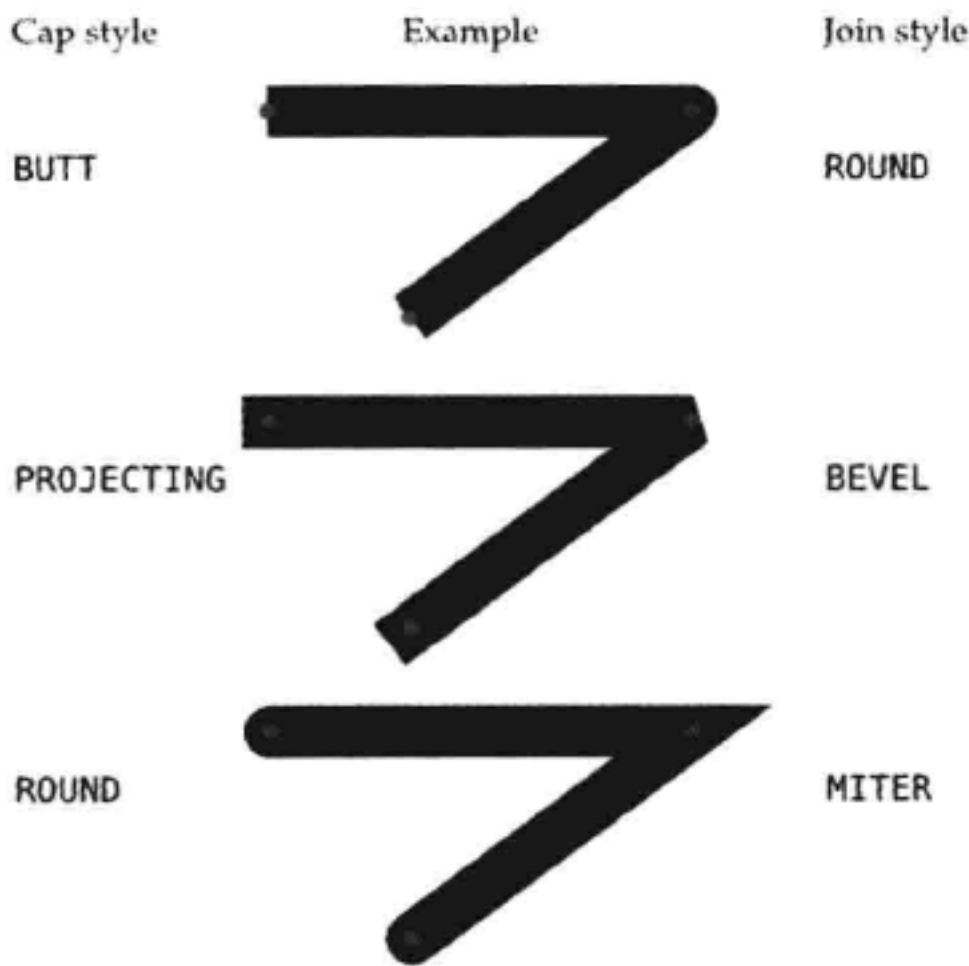


图 16-44 cap style 和 join style 样式



图 16-45 顶点连接样式



```

from tkinter import *
root = Tk()
c = Canvas(root, bg = 'white', width = 250, height = 70); c. pack() #创建并显示 Canvas
#按顶点连接样式画线
c. create_line(10,10,60,10,10,60,60,60, width = 10, joinstyle = ROUND)
c. create_line(70,10,120,10,70,60,120,60, width = 10, joinstyle = BEVEL)
c. create_line(130,10,180,10,130,60,180,60, width = 10, joinstyle = MITER)
c. create_line(190,10,240,10,190,60,240,60, width = 10, smooth = 1, splinesteps = 30)

```

### 16.9.5 绘制椭圆

在 Canvas 对象 c 上绘制椭圆的对象方法为：

```
id = c. create_oval(x0,y0,x1,y1,option,...)
```

其中，(x0, y0) 是左上角的坐标；(x1, y1) 是右下角的坐标；option 为选项，包括 activedash、activefill、activeoutline、activeoutlinestipple、activestipple、activewidth、dash、dashoffset、disableddash、disabledfill、disabledoutline、disabledoutlinestipple、disabledstipple、disabledwidth、fill、offset、outline、outlineoffset、stipple、outlinestipple、state、tags、width。

【例 16-24】椭圆绘制示例 (create\_oval.py)。运行效果如图 16-46 所示。

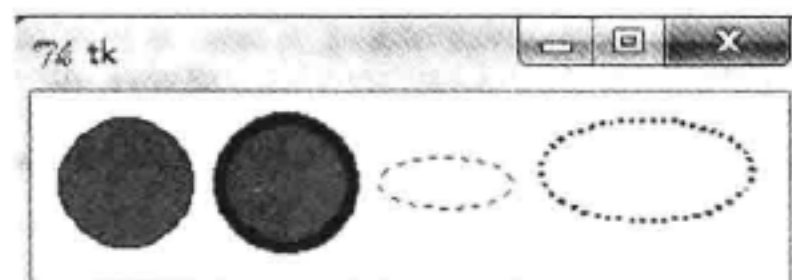


图 16-46 绘制椭圆

```

from tkinter import *
root = Tk()

c = Canvas(root, bg = 'white', width = 280, height = 70); c. pack() #创建并显示 Canvas
c. create_oval(10,10,60,60, fill = 'red') #红色填充圆
c. create_oval(70,10,120,60, fill = 'red', outline = 'blue', width = 5) #红色填充边框蓝色宽度 5 的圆
c. create_oval(130,25,180,45, dash = (4,)) #虚线椭圆
c. create_oval(190,10,270,50, dash = (4,), width = 2) #宽度为 2 的虚线椭圆

```

### 16.9.6 绘制圆弧

在 Canvas 对象 c 上绘制圆弧的对象方法为：

```
id = c. create_arc(x0,y0,x1,y1,option,...)
```

其中，(x0, y0) 是左上角的坐标；(x1, y1) 是右下角的坐标；option 为选项，包括 activedash、activefill、activeoutline、activeoutlinestipple、activestipple、activewidth、dash、dashoffset、disableddash、disabledfill、disabledoutline、disabledoutlinestipple、disabledstipple、disabledwidth、extent、fill、offset、outline、outlineoffset、outlinestipple、start、state、stipple、style、tags、width。其中，选项 start（开始角度，默认为 0）和 extent（圆弧角度，从 start 开始逆时针，默认为 90）决定圆弧的角度范围；如果省略，则绘制从 0 开始 360 度的圆弧，即椭圆。style 用于设置圆弧的样式，取值为：tk.PIESLICE（默认值）、tk.CHORD 和 tk.ARC。对应的形状样式如图 16-47 所示。

【例 16-25】圆弧绘制示例 (create\_arc.py)。运行效果如图 16-48 所示。



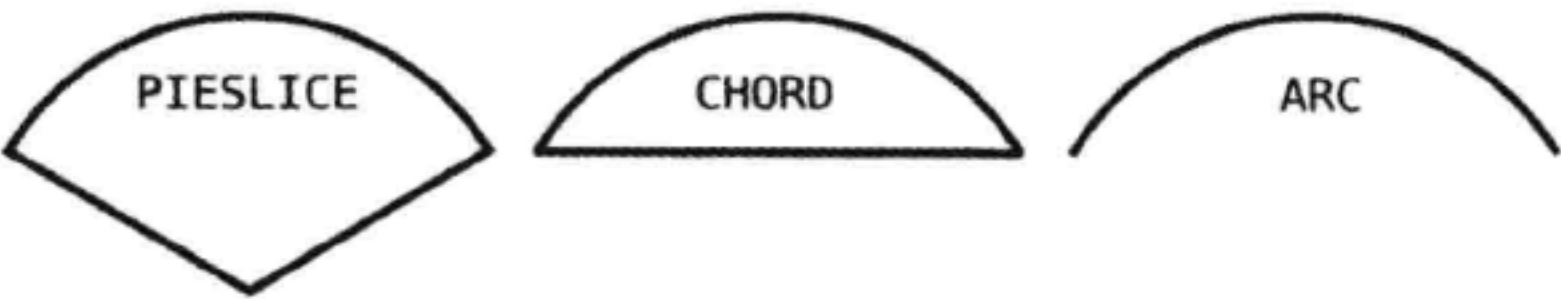


图 16-47 圆弧形状样式

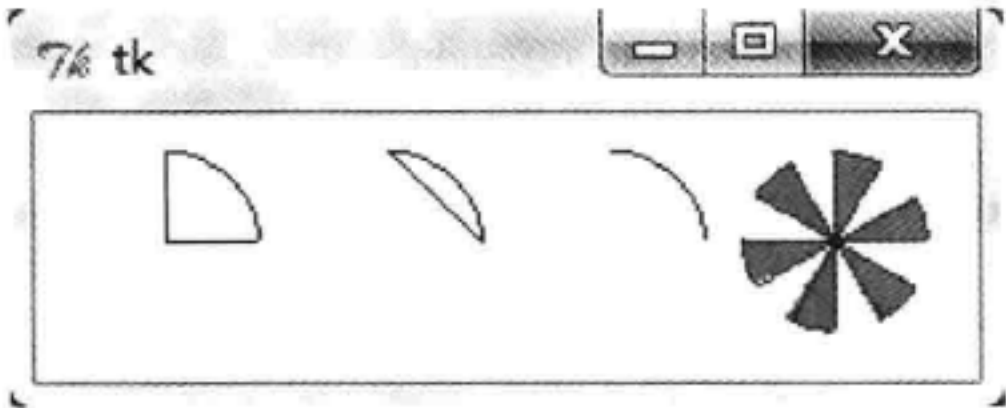


图 16-48 绘制圆弧

```
from tkinter import *
root = Tk()
c = Canvas(root,bg = 'white',width = 250,height = 70);c.pack() #创建并显示 Canvas
c.create_arc(10,10,60,60,style = PIESLICE) #绘制 PIESLICE 样式圆弧
c.create_arc(70,10,120,60,style = CHORD) #绘制 CHORD 样式圆弧
c.create_arc(130,10,180,60,style = ARC) #绘制 ARC 样式圆弧
for i in range(0,360,60): #绘制菊花瓣图形
 c.create_arc(190,10,240,60,fill = 'red',outline = 'blue',start = i,extent = 30)
```

16.9.7 绘制线条

在 Canvas 对象 c 上绘制线条（直线或折线）的对象方法为：

```
id = c.create_line(x0,y0,x1,y1,...,xn,yn,option,...)
```

其中，(x0,y0),(x1,y1),... (xn,yn)是各点的坐标；option 为选项，包括 activedash、activefill、activestipple、activewidth、arrow、arrowshape、capstyle、dash、dashoffset、disabled-dash、disabledfill、disabledstipple、disabledwidth、fill、joinstyle、offset、smooth、splinesteps、state、stipple、tags、width。

【例 16-26】 线条绘制示例（create\_line.py）。运行效果如图 16-49 所示。

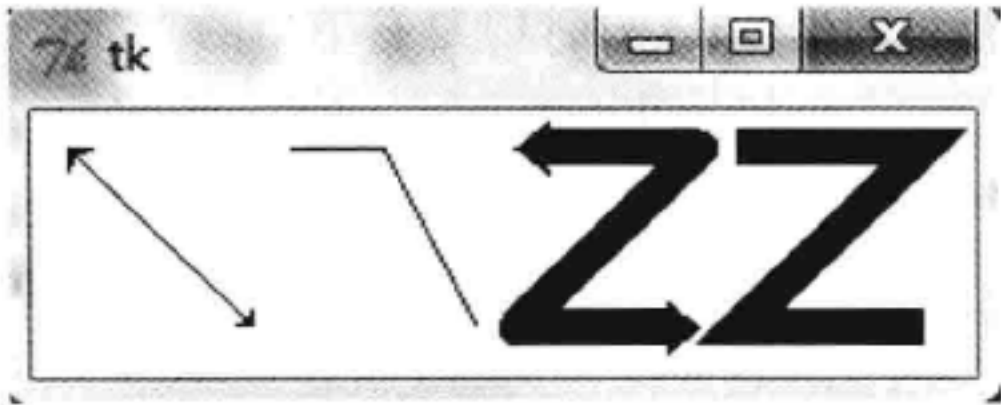


图 16-49 绘制线条

```
from tkinter import *
root = Tk()
```

```

c = Canvas(root,bg = 'white',width = 250,height = 70);c. pack() #创建并显示 Canvas
c. create_line(10,10,60,60,arrow = BOTH,arrowshape = (3,5,4)) #双向箭头
c. create_line(70,10,95,10,120,60) #折线
c. create_line(130,10,180,10,130,60,180,60,width = 10,arrow = BOTH) #Z 字型双向箭头
c. create_line(190,10,240,10,190,60,240,60,width = 10,joinstyle = MITER)#Z 字型

```

## 16.9.8 绘制多边形

在 Canvas 对象 c 上绘制多边形的对象方法为：

```
id = c. create_polygon(x0,y0,x1,y1,...,option,...)
```

其中,  $(x_0,y_0), (x_1,y_1), \dots, (x_n,y_n)$  是各顶点的坐标; option 为选项, 包括 activedash、activefill、activeoutline、activeoutlinestipple、activestipple、activewidth、dash、dashoffset、disableddash、disabledfill、disabledoutline、disabledoutlinestipple、disabledstipple、disabledwidth、fill、joinstyle、offset、outline、outlineoffset、outlinestipple、smooth、splinesteps、state、stipple、tags、width。

【例 16-27】多边形绘制示例 (create\_polygon.py)。运行效果如图 16-50 所示。

```

from tkinter import *
root = Tk()
c = Canvas(root,bg = 'white',width = 250,height = 70);c. pack() #创建并显示 Canvas
c. create_polygon(35,10,10,60,60,60,fill = 'white',outline = 'black') #黑边等腰三角形
c. create_polygon(70,10,120,10,120,60,fill = 'white',outline = 'black') #黑边直角三角形
c. create_polygon(130,10,180,10,180,60,130,60) #黑色填充的正方形
c. create_polygon(190,10,240,10,190,60,240,60,fill = 'white',outline = 'black') #对顶三角形

```

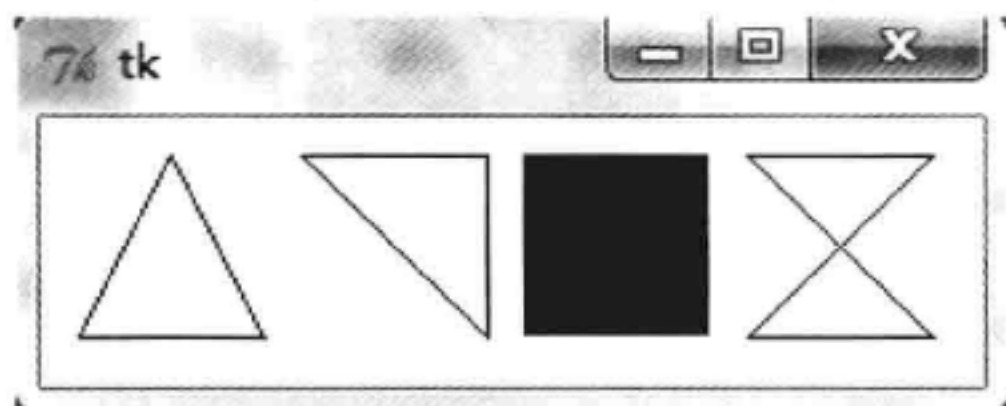


图 16-50 绘制多边形

## 16.9.9 绘制位图

在 Canvas 对象 c 上绘制位图 (bitmap) 的对象方法为：

```
id = c. create_bitmap(x,y,option,...)
```

其中,  $(x,y)$  是位图放置的中心坐标; option 为选项, 包括 activebackground、activebitmap、activeforeground、anchor、background、bitmap、disabledbackground、disabledbitmap、disabledforeground、foreground、state、tags。其中, bitmap、activebitmap 和 disabledbitmap 用于指定正常、活动和禁用状态显示的位图。

【例 16-28】位图绘制示例 (create\_bitmap.py)。运行效果如图 16-51 所示。

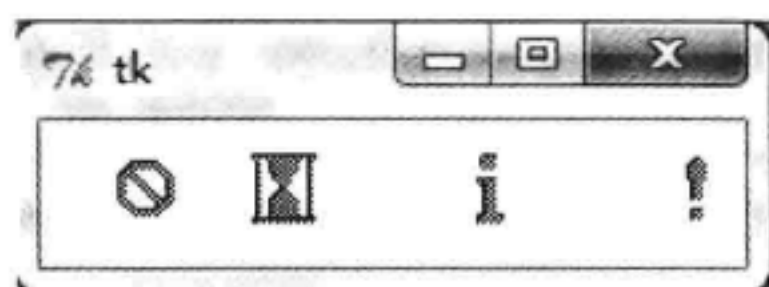


图 16-51 绘制位图

```

from tkinter import *
root = Tk()
c = Canvas(root, bg = 'white', width = 200, height = 40); c.pack() #创建并显示 Canvas
c.create_bitmap(30,20,bitmap = 'error') #error 图标
c.create_bitmap(70,20,bitmap = 'hourglass') #沙漏图标
c.create_bitmap(130,20,bitmap = 'info') #信息图标
c.create_bitmap(190,20,bitmap = 'warning') #警告图标

```

### 16.9.10 绘制图像

在 Canvas 对象 c 上绘制图像的对象方法为：

```
id = c.create_image(x,y,option,...)
```

其中，(x,y) 是图像放置的中心坐标；option 为选项，包括 activeimage、anchor、disabledimage、image、state、tags。其中，image、activeimage 和 disabledimage 用于指定正常、活动和禁用状态显示的图像。

【例 16-29】图像绘制示例 (create\_image.py)。运行效果如图 16-52 所示。



图 16-52 绘制图像

```

from tkinter import *
root = Tk()
c = Canvas(root, bg = 'white', width = 100, height = 140); c.pack() #创建并显示 Canvas
logo = BitmapImage(file = r'C:\Python\images\xbm\face.xbm') #图像文件
c.create_image(50,70,image = logo) #绘制图像

```

### 16.9.11 绘制字符串

在 Canvas 对象 c 上绘制字符串的对象方法为：

```
id = c.create_text(x,y,option,...)
```

其中，(x,y) 是字符串放置的中心坐标；option 为选项，包括 activefill、activestipple、anchor、disabledfill、disabledstipple、fill、font、justify、offset、state、stipple、tags、text、width。其中，text 用于指定要绘制的字符串；font 用于指定字体；justify 用于指定对齐方式。

【例 16-30】字符串和图形绘制 (sin.py)：绘制给定函数  $y = \sin(x)$  的图形。运行效果如图 16-53 所示。



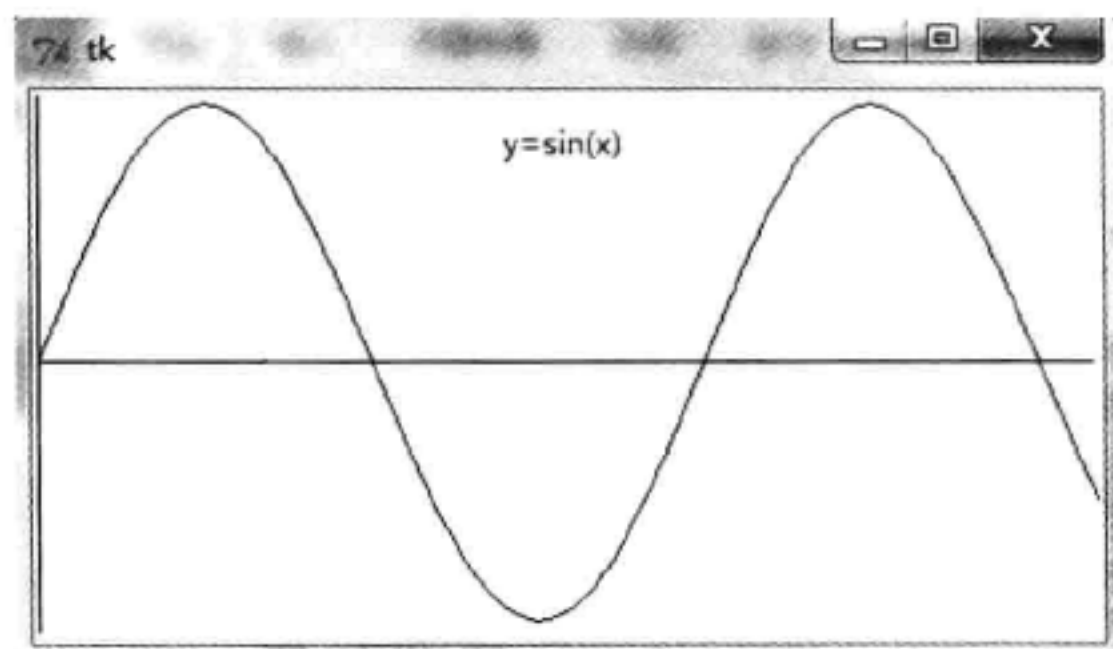


图 16-53 绘制函数

```

from tkinter import *
import math
WIDTH = 400; HEIGHT = 210 #画布宽度、高度
ORIGIN_X = 2; ORIGIN_Y = HEIGHT/2 #原点 X、原点 Y:窗体左边中心
SCALE_X = 40; SCALE_Y = 100 #X 轴、Y 轴的缩放倍数
END_ARC = 360 * 2 #画多长
ox = 0; oy = 0; x = 0; y = 0 #坐标初始值
arc = 0 #弧度
root = Tk()
c = Canvas(root, bg = 'white', width = WIDTH, height = HEIGHT); c.pack() #创建并显示 Canvas
c.create_text(200, 20, text = 'y = sin(x)') #绘制文字
c.create_line(0, ORIGIN_Y, WIDTH, ORIGIN_Y) #绘制 x 纵轴
c.create_line(ORIGIN_X, 0, ORIGIN_X, HEIGHT) #绘制 y 横轴
for i in range(0, END_ARC, 10): #绘制线
 arc = math. pi * i * 2/360
 x = ORIGIN_X + arc * SCALE_X
 y = ORIGIN_Y - math. sin(arc) * SCALE_Y
 c.create_line(ox, oy, x, y)
 ox = x; oy = y

```

### 16.9.12 绘制组件

在 Canvas 对象 c 上创建组件的对象方法为:

**id = c.create\_window( x, y, option, ... )**

其中, (x, y) 是组件放置的中心坐标; option 为选项, 包括 anchor、height、state、tags、width、window。其中, window 用于指定要绘制的组件对象。

【例 16-31】 组件绘制示例 (create\_window.py)。运行效果如图 16-54 所示。

```

from tkinter import *
def fl():

```

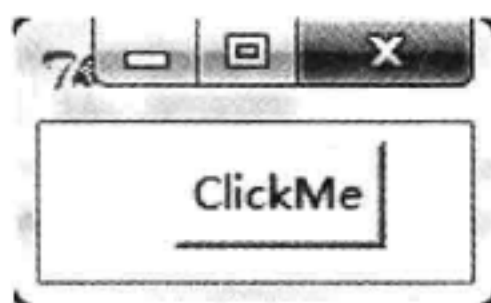


图 16-54 绘制组件

```
print('You click me! ')\nroot = Tk()\nc = Canvas(root, bg = 'white', width = 100, height = 40); c.pack() #创建并显示 Canvas\nbutton1 = Button(c, text = 'ClickMe', command = f1) #ClickMe 按钮\nc.create_window(30, 20, window = button1, anchor = W) #创建组件\nroot.mainloop()
```

## 16.10 复习题

### 一、填空题

1. Python 的标准 GUI 库 tkinter 由若干的模块组成：\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_等。
2. Python 图形用户界面程序一般包含一个顶层窗口，也称\_\_\_\_\_或\_\_\_\_\_。
3. tkinter 提供了三种不同的几何布局管理类：\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_，用于组织和管理在父组件中子配件的布局方式。
4. 通过组件的\_\_\_\_\_和\_\_\_\_\_选项，可以设置组件的宽度和高度。
5. 通过组件的\_\_\_\_\_选项，可以设置其显示的文本的字体。
6. 通过组件的\_\_\_\_\_选项，可以设置内容停靠位置。
7. 通过组件的\_\_\_\_\_选项，可以设置鼠标经过组件时的光标形状。
8. 通过组件的\_\_\_\_\_选项，可以设置其显示的内容。通过\_\_\_\_\_选项，可指定多少单位后开始换行，即显示多行；通过\_\_\_\_\_选项，可指定多行的对齐方式。
9. 通过组件的\_\_\_\_\_选项，可以设置其显示的位图。自定义位图为\_\_\_\_\_格式的文件。
10. 通过组件的\_\_\_\_\_选项，可以设置其显示的图像。
11. 通过组件的\_\_\_\_\_选项，可以设置其同时显示文本和位图/图像。
12. 通过组件的\_\_\_\_\_选项，可以设置其 3D 显示样式。通过\_\_\_\_\_选项，可以设置其鼠标经过时的 3D 显示样式。
13. 通过组件的\_\_\_\_\_或\_\_\_\_\_选项，可以设置其边框宽度。
14. 通过组件的\_\_\_\_\_和\_\_\_\_\_选项，可以设置其显示内容与边框之间的填充宽度和高度。
15. 通过组件的\_\_\_\_\_选项，可以设置其启用或禁用状态。
16. 通过组件的\_\_\_\_\_选项，可以设置组件显示文本第几个字符加下划线。
17. 通过组件的\_\_\_\_\_选项，可以绑定 StringVar 对象到组件。
18. \_\_\_\_\_控件用于选择同一组单选按钮中的一个单选按钮（不能同时选定多个），可显示文本，也可显示图像。
19. \_\_\_\_\_控件用于选择一项或多项选项（可以同时选定多个），可显示文本，也可显示图像。
20. \_\_\_\_\_用于显示对象列表，并且允许用户选择一个或多个项。
21. \_\_\_\_\_允许用户选择一个项的列表框（在用户请求时显示）。用户单击下拉按钮可显示列表框，选择的内容会显示在顶部文本框中。

22. \_\_\_\_\_控件用于在有界区间内, 通过移动滑块来选择值。
23. tkinter 模块中的子模块\_\_\_\_\_, \_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_, 包括通用的预定义对话框; 用户也可以通过继承 TopLevel 创建自定义对话框。
24. tkinter 模块中的子模块\_\_\_\_\_用于实现通用消息对话框的功能。
25. tkinter 模块中的子模块\_\_\_\_\_用于实现文件对话框的功能。
26. tkinter 模块中的子模块\_\_\_\_\_用于实现颜色选择对话框的功能。
27. tkinter 模块中的子模块\_\_\_\_\_用于实现输入对话框的功能。
28. \_\_\_\_\_是一个长方形的区域, 用于图形绘制或复杂的图形界面布局。可以在其上绘制图形、文字, 放置各种组件和框架。

## 二、思考题

1. Python 中有哪几种导入 tkinter 模块的方法?
2. Python 图形用户界面应用程序包括哪三种类型的菜单?
3. Python 中创建主菜单一般遵循哪些步骤?
4. Python 中创建上下文菜单一般遵循哪些步骤?

## 16.11 上机实践

1. 参照例 16-1 创建图形用户界面 Hello world 程序。
2. 参照例 16-2 创建 GUI 应用程序类, 实现 Hello world 程序。
3. 参照例 16-3 创建 pack 几何布局程序。
4. 参照例 16-4 创建 grid 几何布局程序, 实现用户登录界面。
5. 参照例 16-5 创建 grid 几何布局程序, 实现按钮布局界面。
6. 参照例 16-6 创建 place 几何布局程序, 实现用户登录界面。
7. 参照例 16-7 利用 Label 和 Button 组件, 创建简易图片浏览器程序。
8. 参照例 16-8 利用 Entry 和 Text 组件, 创建用户注册程序。
9. 参照例 16-9 利用 Radiobutton 和 Checkbox, 创建 Questionnaire 调查个人信息程序。
10. 参照例 16-10 创建列表选择功能程序。
11. 参照例 16-11 创建 OptionMenu 选择项程序, 从组合框中选择字体大小, 然后单击“改变字体”按钮, 改变标签文本的字体大小。
12. 参照例 16-12 创建 Scale 程序, 通过移动滑块, 改变字体大小。
13. 参照例 16-13 利用 Toplevel, 创建自定义关于对话框程序。
14. 参照例 16-14 创建通用对话框应用示例程序。
15. 参照例 16-15 创建主菜单示例程序。
16. 参照例 16-16 创建上下文菜单示例程序。
17. 参照例 16-17 创建简单文本编辑器程序。
18. 参照例 16-18 利用 Canvas 组件, 创建查找和删除绘制对象的程序。
19. 参照例 16-19 利用 Canvas 组件, 创建绘制矩形的程序。
20. 参照例 16-20 利用 Canvas 组件, 创建绘制各样式虚线的程序。
21. 参照例 16-21 利用 Canvas 组件, 创建填充画刷程序。



22. 参照例 16-22 利用 Canvas 组件, 创建绘制各样式箭头的程序。
23. 参照例 16-23 利用 Canvas 组件, 创建绘制各样式顶点连接的程序。
24. 参照例 16-24 利用 Canvas 组件, 创建绘制圆弧程序。
25. 参照例 16-25 利用 Canvas 组件, 创建绘制椭圆程序。
26. 参照例 16-26 利用 Canvas 组件, 创建绘制绘制线条 (直线或折线) 程序。
27. 参照例 16-27 利用 Canvas 组件, 创建绘制多边形程序。
28. 参照例 16-28 利用 Canvas 组件, 创建绘制位图程序。
29. 参照例 16-29 利用 Canvas 组件, 创建绘制图像程序。
30. 参照例 16-30 利用 Canvas 组件, 创建绘制字符串和图形的程序, 绘制函数  $y = \cos(x)$  的图形, 运行效果如图 16-55 所示。

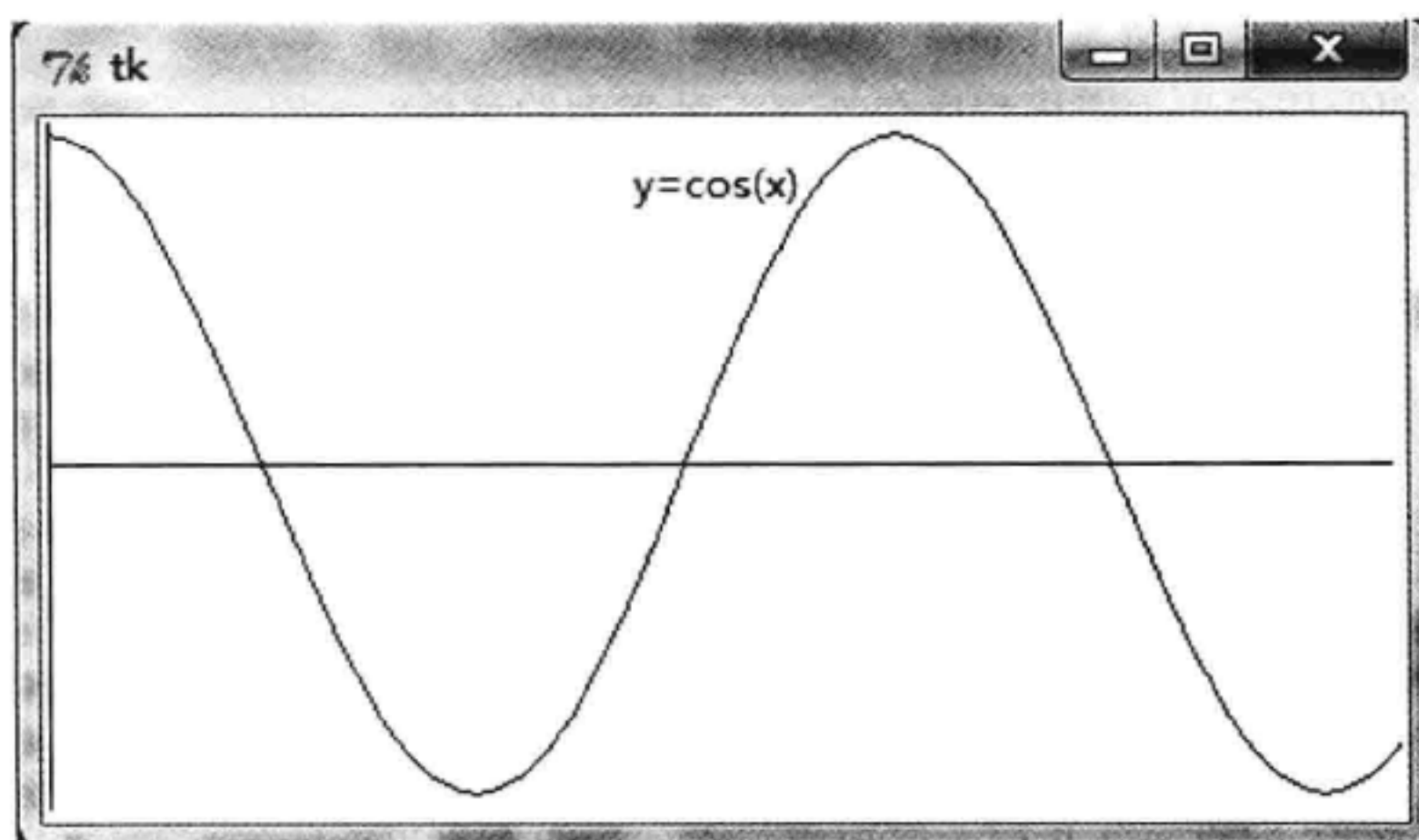


图 16-55 函数  $y = \cos(x)$  运行效果

31. 参照例 16-31 利用 Canvas 组件, 创建绘制按钮组件的程序。

# 第 17 章 数据库访问

应用程序往往使用数据库来存储大量的数据。Python 提供了对大多数数据库的支持。使用 Python 中相应的模块，可以连接到数据库，进行查询、插入、更新和删除等操作。

## 本章要点：

- ◆ 数据库基础；
- ◆ Python 数据库访问模块简介；
- ◆ 使用 sqlite 访问数据库。

## 17.1 数据库基础

### 17.1.1 数据库概念

数据库就是存储数据的仓库，即存储在计算机系统中结构化的、可共享的相关数据的集合。数据库中的数据按一定的数据模型组织、描述和存储，可以最大限度地减少数据的冗余度。

数据库管理系统（Database Management System, DBMS）是用于管理数据的计算机软件。数据库管理系统使用户能够方便地定义数据、操作数据以及维护数据。其主要功能包括：

（1）数据定义功能。使用数据定义语言（Data Definition Language, DDL），可以生成和维护各种数据对象的定义。

（2）数据操作功能。使用数据操作语言（Data Manipulation Language, DML），可以对数据库进行查询、插入、删除和修改等基本操作。

（3）数据库的管理和维护。数据库的安全性、完整性、并发性、备份和恢复等功能。

目前流行的数据库管理系统产品可以分为两类：

（1）适合于企业用户的网络版 DBMS：如 Oracle、Microsoft SQL Server、IBM DB2 等。

（2）适合于个人用户的桌面 DBMS：如 Microsoft Access 等。

数据库系统（Database System, DBS）是指在计算机系统中引入数据库后组成的系统。数据库系统一般包括：计算机硬件、操作系统、DBMS、开发工具、应用系统、数据库管理员（Database Administrator, DBA）、用户。

### 17.1.2 关系数据库

常用的数据库模型包括：层次模型（Hierarchical Model）、网状模型（Network Model）、关系模型（Relational Model）和面向对象的数据模型（Object Oriented Model）。

关系模型具有完备的数学基础，简单灵活，易学易用，已经成为数据库的标准。目前流

行的 DBMS 都是基于关系模型的关系数据库管理系统。

关系模型把世界看作是由实体（Entity）和联系（Relationship）构成的。实体是指现实世界中具有一定特征或属性并与其他实体有联系的对象，在关系模型中实体通常是以表的形式来表现。表的一行描述实体的一个实例，表的每一列描述实体的一个特征或属性。

联系是指实体之间的对应关系，通过联系就可以用一个实体的信息来查找另一个实体的信息。联系可以分为以下三种。

① 一对一：如一个部门只能有一个经理，而一个经理只能在一个部门任职，部门和经理为一对一的联系。

② 一对多：如一个部门有多名员工，而一名员工只能在一个部门工作，部门和员工为一对多的联系。

③ 多对多：如一名学生可以选修多门课程，而一门课程可以有多名选修的学生，学生和课程是多对多的联系。

关系数据库中，常见的数据库对象包括表、视图、触发器、存储过程等。

数据库中的表由行（Row）和列（Column）组成。列由同类的信息组成，又称为字段；列的标题称为字段名。行是指包括了若干列信息项的一行数据，也称为记录。一个数据库表由一条或多条记录组成，没有记录的表称为空表。

每个数据表中通常都有一个主关键字（Primary Key），用于唯一确定一条记录，例如图 17-1 中“学号”字段即为 Exam 数据表的主关键字。

列标题（字段名）

|       | 学号    | 姓名 | 班级 | 语文  | 数学  | 英语 | 计算机 |
|-------|-------|----|----|-----|-----|----|-----|
| 行（记录） | 03101 | 张咏 | 1  | 87  | 97  | 89 | 90  |
|       | 03102 | 刘炎 | 1  | 100 | 90  | 95 | 96  |
|       | 03103 | 王政 | 1  | 78  | 85  | 70 | 55  |
|       | 03104 | 李石 | 1  | 20  | 56  | 38 | 50  |
|       | 03105 | 姚亮 | 1  | 97  | 90  | 95 | 92  |
|       | 03201 | 张晶 | 2  | 50  | 45  | 67 | 89  |
|       | 03202 | 姜玲 | 2  | 90  | 98  | 97 | 93  |
|       | 03203 | 汪茗 | 2  | 98  | 100 | 96 | 97  |
|       | 03204 | 赵骅 | 2  | 44  | 56  | 46 | 58  |
|       | 03205 | 桑恬 | 2  | 76  | 70  | 86 | 80  |
|       | 03206 | 陆锋 | 2  | 87  | 88  | 85 | 90  |

图 17-1 数据表的列（字段）

## 17.2 Python 数据库访问模块

### 17.2.1 通用数据库访问模块

#### 1. ODBC

ODBC（Open Database Connectivity，开放数据库互连）提供了一种标准的应用程序编程



接口（Application Programming Interface，API）方法来访问数据库管理系统。

在 Windows 平台上，常用的数据库产品都实现了其各自的 ODBC 驱动程序，包括 Oracle、DB2、Microsoft SQL Server、Access 等数据库。因为通过 ODBC，可以实现通用的数据库访问。Python 提供了通过 ODBC 访问数据的模块。包括：

- ✎ ODBC Interface：随 PythonWin 附带发行的模块。
- ✎ pyodbc：开源的 Python ODBC 接口，完整实现了 DB - API 2.0 接口。
- ✎ mxODBC：流行的 mx 系列工具包中的一部分（非商业开发需付费），实现了绝大部分 DB - API 2.0 接口。

2. JDBC

JDBC（Java Data Base Connectivity，Java 数据库连接）是基于 Java 的面向对象的应用编程接口，描述了一套访问关系数据库的 Java 类库标准。

Jython 2.1 以后的发行版中，包括通过 JDBC 访问数据的模块 zxJDBC，建立在底层的 JDBC 接口之上，支持 DB - API 2.0 接口。

17.2.2 专用数据库访问模块

Python 针对各种流行的数据库，提供了各种专用的数据库访问模块，参见表 17-1 所示。

表 17-1 专用数据库访问模块

| 数 据 库      | Python 模块      | 网 址                                                                                                     |
|------------|----------------|---------------------------------------------------------------------------------------------------------|
| MySQL      | mysql - python | <a href="http://sourceforge.net/projects/mysql-python">http://sourceforge.net/projects/mysql-python</a> |
| PostgreSQL | PyGreSQL       | <a href="http://www.pygresql.org/">http://www.pygresql.org/</a>                                         |
| Oracle     | DCOracle2      | <a href="http://www.zope.org/Members/matt/dco2">http://www.zope.org/Members/matt/dco2</a>               |
| IBM DB2    | pydb2          | <a href="http://sourceforge.net/projects/pydb2">http://sourceforge.net/projects/pydb2</a>               |
| SQL Server | pymssql        | <a href="http://pymssql.sourceforge.net/">http://pymssql.sourceforge.net/</a>                           |

17.2.3 SQLite 数据库和 sqlite3 模块

1. SQLite 数据库

SQLite 是一款开源的轻型的数据库，占用资源非常低，广泛用于各种嵌入式设备中。SQLite 支持各种主流的操作系统，包括 Windows，Linux，UNIX 等，并与许多程序语言紧密结合，包括 Python。

SQLite 是遵守 ACID Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）、Durability（持久性）的关系数据库管理系统，实现了多数的 SQL - 92 标准，包括事务、触发器和多数的复杂查询。

SQLite 不进行类型检查，例如，可以把字符串插入到整数列中。该特点特别适于与无类型的脚本语言（如 Python）一起使用。

SQLite 的整个数据库，包括数据库定义、表、索引和数据本身等，都存储在一个单一的文件中，其事务处理通过锁定整个数据文件而完成。

SQLite 引擎不是程序与之通信的独立进程，而是在编程语言内直接调用 API 来实现，即 SQLite 是应用程序的组成部分。故具有内存消耗低、延迟时间短、整体结构简单等优点。

SQLite 目前的版本是 3，其官方网址为：<http://www.sqlite.org>。

## 2. SQLite 支持的数据类型

SQLite 支持的数据类型包括 NULL、INTEGER、REAL、TEXT 和 BLOB，分别对应 Python 的数据类型 None、int、float、str 和 bytes。

可以使用适配器（adapters），以存储更多的 Python 类型到 SQLite 数据库；也可使用转换器（converters），把 SQLite 数据类型转换为 Python 的数据类型。

## 3. sqlite3 模块

Python 标准模块 sqlite3 使用 C 语言实现，提供访问和操作数据库 SQLite 的各种功能。sqlite3 模块主要包括下列常量、函数和对象：

sqlite3.version：常量，版本号。

sqlite3.connect(database)：函数，连接到数据库，返回 Connect 对象。

sqlite3.Connection：数据库连接对象。

sqlite3.Cursor：游标对象。

sqlite3.Row：行对象。

# 17.3 使用 sqlite3 模块连接和操作 SQLite 数据库

## 17.3.1 访问数据库的典型步骤

Python 的数据库模块具有统一的接口标准，数据库操作遵循一致的模式。使用 sqlite3 模块操作数据的典型步骤为：

### 1. 导入相应的数据库模块

Python 标准库中带有 sqlite3 模块，可直接导入：

```
import sqlite3
```

### 2. 建立数据库连接，返回 Connection 对象

使用数据库模块的 connect 函数建立数据库连接，返回连接对象 con：

```
con = sqlite3.connect(connectstring) #连接到数据库,返回 sqlite3.Connection 对象
```

其中，connectstring 是连接字符串。对于不同的数据库连接对象，其连接字符串的格式各不相同。sqlite 的连接字符串为数据库的文件名，如 c:\example.db。如果指定连接字符串为:memory:，则可创建一个内存数据库。例如：

```
>>> import sqlite3
```

```
>>> con = sqlite3.connect("c:\db1.db")
```

如果 c:\db1.db 存在，则打开数据库；否则创建并打开数据库 c:\db1.db。

创建数据库连接对象（Connection 对象）后，可设置其属性。例如：

```
>>> con.isolation_level = None #设置事务隔离级别,默认为自动提交
```

```
>>> con.row_factory = sqlite3.Row #设置连接对象使用的行工厂对象
```

### 3. 创建游标对象 cur

调用 con.cursor() 创建游标对象 cur：

```
cur = con.cursor() #创建游标对象
```

#### 4. 使用 Cursor 对象的 execute 执行 SQL 命令返回结果

调用 cur.execute, executemany, executescript 方法查询数据库。

cur.execute(sql): 执行 SQL 语句。

cur.execute(sql, parameters): 执行带参数的 SQL 语句。

cur.executemany(sql, seq\_of\_parameters): 根据参数执行多次 SQL 语句。

cur.executescript(sql\_script): 执行 SQL 脚本。

一般地, 建议直接使用 Connection 对象的 execute, executemany, executescript 方法。事实上, 它们是 Cursor 对象对应方法的快捷方式, 系统创建一个临时 Cursor 对象, 然后调用对应的方法, 并返回 Cursor 对象。

con.execute(sql): 执行 SQL 语句, 返回结果。

con.execute(sql, parameters): 执行带参数的 SQL 语句, 返回结果。

con.executemany(sql, seq\_of\_parameters): 根据参数执行多次 SQL 语句, 返回结果。

con.executescript(sql\_script): 执行 SQL 脚本。

例如:

```
>>> con.execute("create table if not exists t1(id primary key,name)")
```

将创建 1 个包含 2 个字段 id 和 name 的表 t1。

SQL 语句字符串中可以使用占位符? 表示参数, 传递的参数使用元组; 或者使用命名参数, 传递参数则使用字典。例如:

```
>>> con.execute("insert into t1(id,name) values (?,?)",('001','北京'))
```

```
>>> con.execute("insert into t1(id,name) values (:id,:name)",{'id':'027','name':'武汉'})
```

#### 5. 获取游标的查询结果集

调用 cur.fetchall, cur.fetchone, cur.fetchmany 返回查询结果:

cur.fetchone(): 返回结果集的下一行(Row 对象); 无数据时, 返回 None。

cur.fetchall(): 返回结果集的剩余行(Row 对象列表); 无数据时, 返回空 list。

cur.fetchmany(size): 返回结果集的多行(Row 对象列表); 无数据时, 返回空 list。

cur.rowcount: 返回影响的行数、结果集的行数。

Row 对象 r 为 1 行查询结果系列, 支持下列访问:

r[i]: 按索引访问, 返回第 i 列的数据。

r[colname]: 按列名称访问, 返回 colname 列的数据。

len(r): 返回列数。

tuple(r): 把数据转换为元组。

r.keys(): 返回列名称的列表。

例如:

```
>>> cur = con.cursor()
```

#创建游标对象

```
>>> cur.execute("select * from t1")
```

#执行 SQL 查询

```
>>> r = cur.fetchone()
```

#获取 1 行 Row 对象结果

```
>>> r.keys()
```

#返回列名称的列表

```
>>> r[0], r['name']
```



也可直接使用循环输出结果。例如：

```
>>> for row in con.execute("select * from t1"):
 print(row[0],row[1])
```

## 6. 数据库的提交和回滚

根据数据库事务隔离级别的不同，可以提交或回滚：

conn.commit()：提交。

conn.rollback()：回滚。

## 7. 关闭 Cursor 对象和 Connection 对象

最后，需要关闭打开的 Cursor 对象和 Connection 对象：

cur.close()：关闭 Cursor 对象。

con.close()：关闭 Connection 对象。

### 17.3.2 创建数据库和表

使用 sqlite3.connect("数据库文件名") 可创建或打开 SQLite 数据库，并返回连接对象 con；使用 con.execute("create table ...")，可创建表。

【例 17-1】创建数据库和表（DBCreate.py）。创建数据库 sales，并在其中创建表 region，表中包含两个列：id 和 name，其中 id 为主码（primary key）。

```
import sqlite3
#创建 SQLite 数据库:c:\Python\chapter17\sales.db
con = sqlite3.connect(r"c:\Python\chapter17\sales.db")
#创建表:regions,包含 2 个列,id(主码)和 name
con.execute("create table region(id primary key,name)")
```

### 17.3.3 数据库表的插入、更新和删除操作

在数据库表中插入、更新和删除记录的一般步骤为：（1）建立数据库连接；（2）根据 SQL Insert、Update、Delete 语句，使用 con.execute(sql) 执行数据库记录插入、更新、删除操作，并根据返回的值判断操作结果。（3）提交操作；（4）关闭数据库。

【例 17-2】数据库表记录的插入、更新和删除操作示例（DBUpdate.py）。

```
import sqlite3
regions = [("021","上海"), ('022',"天津"), ("023","重庆"), ("024","沈阳")]
#打开 SQLite 数据库:c:\Python\chapter17\sales.db
con = sqlite3.connect(r"c:\Python\chapter17\sales.db")
#插入 1 行数据
con.execute("insert into region(id,name) values ('020','广东')")
con.execute("insert into region(id,name) values (?,?)",('001','北京'))
#插入多行数据
con.executemany("insert into region(id,name) values (?,?)",regions)
#修改 1 行数据
con.execute("update region set name = ? where id = ?",('广州','020'))
#删除 1 行数据
```

```
n = con. execute("delete from region where id = ?", ("024",))
print('删除了', n. rowcount, '行记录')
con. commit() #提交
con. close() #关闭数据库
```

### 17.3.4 数据库表的查询操作

查询数据库的一般步骤为：（1）建立数据库连接；（2）根据 SQL Select 语句，使用 `con. execute (sql)` 执行数据库查询操作，返回游标对象 `cur`；（3）循环输出结果。

【例 17-3】查询数据表中的记录信息（DBquery.py）。

```
import sqlite3
#打开 SQLite 数据库:c:\Python\chapter17\sales. db
con = sqlite3. connect(r"c:\Python\chapter17\sales. db")
#查询数据库表
cur = con. execute("select id, name from region")
for row in cur: #循环输出结果
 print(row)
```

运行结果如下：

```
('020 ', '广州')
('001 ', '北京')
('021 ', '上海')
('022 ', '天津')
('023 ', '重庆')
```

## 17.4 复习题

1. 数据库管理系统主要包括哪些功能？
2. 目前有哪些流行的数据库管理系统产品？
3. 常用的数据库模型是什么？目前流行的 DBMS 主要基于哪一类数据库模型？
4. Python 提供哪几类数据库访问模块？
5. SQLite 支持哪几类数据类型？分别对应于 Python 的哪些数据类型？
6. sqlite3 模块主要包括哪些常量、函数和对象？
7. 使用 sqlite3 模块操作数据的典型步骤是什么？
8. Python 在数据库表中插入、更新和删除记录的一般步骤是什么？
9. Python 在数据库表中查询记录的一般步骤是什么？

## 17.5 上机实践

1. 参照例 17-1 编写创建数据库和表的程序，创建数据库 `sales`，并在其中创建表 `region`，表中包含两个列：`id` 和 `name`，其中 `id` 为主码。
2. 参照例 17-2 编写数据库表记录的插入、更新和删除操作的程序。
3. 参照例 17-3 编写查询数据表中记录信息的程序。

# 第 18 章 网络编程和通信

Python 提供了用于网络编程和通信的各种模块，可以使用 socket 模块进行基于套接字的底层网络编程，也可以使用 urllib、http、ftplib、poplib、smtpplib 等模块针对特定网络协议编程，还可以使用扩展库进行网络编程。

## 本章要点：

- ◆ 网络编程的基本概念；
- ◆ 基于 Socket 网络编程；
- ◆ 基于 urllib 的网络编程；
- ◆ 基于 http 的网络编程；
- ◆ 基于 ftplib 的网络编程；
- ◆ 基于 poplib 和 smtpplib 的网络编程。

## 18.1 网络编程的基本概念

### 18.1.1 网络基础知识

计算机网络是由传输介质连接在一起的一系列设备（网络结点）组成。一个结点可以是一台计算机、打印机或是任何能够发送或接收由网络上其他结点产生数据的设备。

两台计算机之间要进行通信，必须采用相同的信息交换规则。在计算机网络中，用于规定信息的格式，以及如何发送和接收信息的一套规则、标准或约定称为网络协议（Network Protocol）。目前使用最广泛的网络协议是 Internet 上所使用的 TCP/IP 协议。

网络编程就是通过网络协议与其他计算机进行通信。网络编程涉及主机定位和数据传输。在 TCP/IP 协议中，IP 层负责网络主机定位、数据传输路由；TCP 层则提供面向应用的数据传输机制。

目前较为流行的网络编程模型是客户 - 服务器（Client/Server，C/S）结构。如图 18-1 所示。

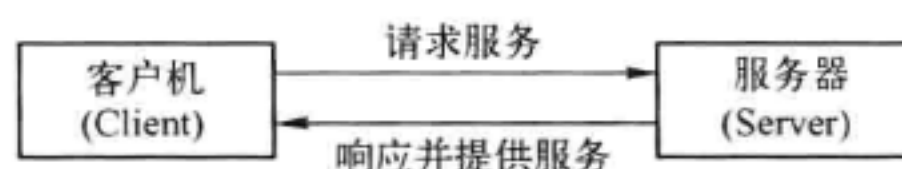


图 18-1 客户机/服务器（C/S）结构



事实上，C/S 模型体现的是一种网络数据访问的实现方式。请求服务的一方为客户机；响应请求并提供服务的一方为服务器。所以一台主机有可能同时为客户机角色和服务器角色。

18.1.2 TCP/IP 协议简介

TCP/IP 协议，即传输控制协议/互连网协议（Transmission Control Protocol/Internet Protocol，TCP/IP），是一种网际互联通信协议，其目的在于通过它实现网际间各种异构网络和异种计算机的互联通信。众多的网络产品厂家都支持 TCP/IP 协议，TCP/IP 已成为一个事实上的工业标准。TCP/IP 协议模型把 TCP/IP 协议族分成四个层次，如图 18-2 所示。

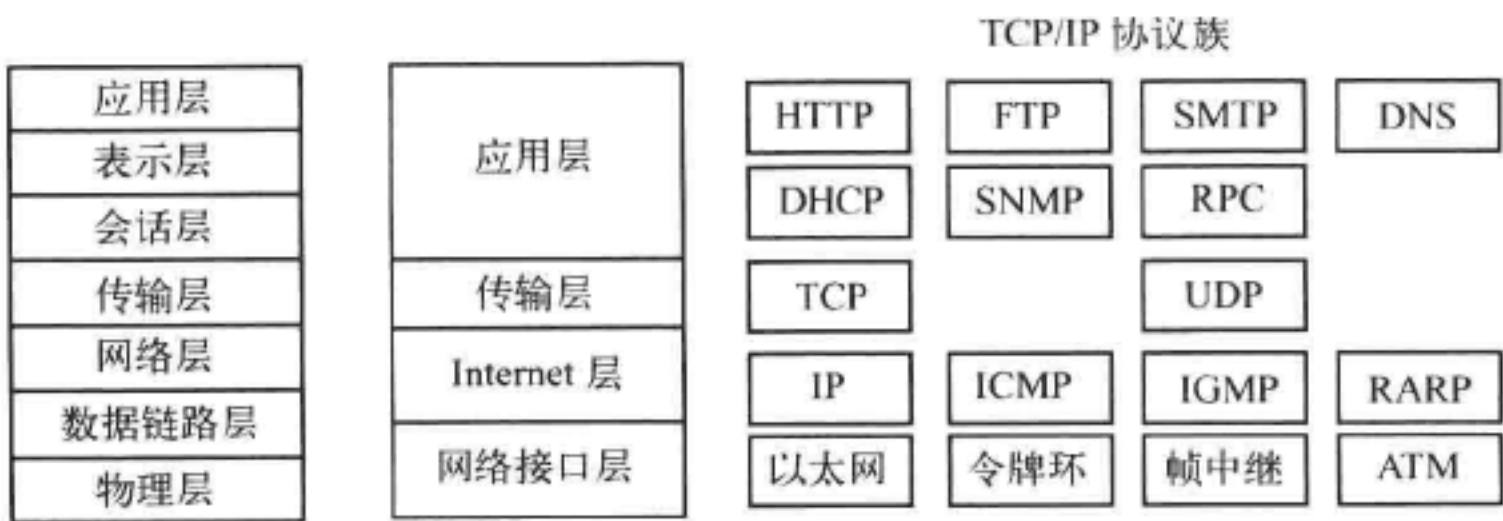


图 18-2 TCP/IP 四层参考模型

1. 网络接口层

网络接口层（又称网络访问层）对应于 OSI 模型的数据链路层和物理层，负责向网络媒体发送 TCP/IP 数据包并从网络媒体接收 TCP/IP 数据包。从理论上讲，该层不是 TCP/IP 协议的组成部分，但它是 TCP/IP 协议的基础，是各种网络与 TCP/IP 协议的接口。

2. Internet 层

Internet 层对应于 OSI 模型的网络层，负责相同或不同网络中计算机之间的通信，主要处理数据报和路由。IP 是一个可路由的协议，可以为数据包进行寻址、路由、分段和重组。IP 协议在 TCP/IP 协议组中处于核心地位。

3. 传输层

传输层对应于 OSI 模型的传输层，为应用层提供端到端的会话和数据报通信服务，主要功能是数据格式化、数据确认和丢失重传等。该层主要包括以下两种协议。

(1) TCP 协议

TCP 协议定义了两台计算机之间进行可靠传输时交换的数据和确认信息的格式，以及计算机为了确保数据的正确到达而采取的措施。该协议是面向连接的，可提供可靠的、按序传送数据的服务。TCP 协议采用的最基本的可靠性技术包括三个方面：确认与超时重传、流量控制和拥塞控制。

TCP 协议的优点是可靠通信服务，缺点是建立连接需要额外的网络开销。

(2) UDP 协议

UDP 协议（User Datagram Protocol，用户数据报协议）也是建立在 IP 协议之上，同 IP 协议一样提供无连接数据报传输。UDP 本身并不提供可靠性服务，相对 IP 协议，它唯一增

加的能力是提供协议端口，以保证进程通信。与 TCP 不同，UDP 提供一对一或一对多的、无连接的不可靠通信服务。

UDP 协议的优点是简单高效，缺点是通信服务有可能不可靠。

#### 4. 应用层

应用层是 TCP/IP 协议的最高层，对应于 OSI 模型的应用层、表示层和会话层。应用层允许应用程序访问其他层的服务，它定义了应用程序用来交换数据的协议。应用层包含大量的协议，而且随着网络技术和应用的发展，不断会产生许多新的应用层协议。

### 18.1.3 IP 地址和域名

#### 1. IP 地址

在 Internet 中，网络中的两台主机进行通信时，其传送的数据包里必须包含附加信息的地址信息（即发送数据的计算机的地址和接收数据的计算机的地址），以保证通信主机间的正确路由。

Internet 采用一种全局通用的地址格式，为网络中的每一台主机都分配一个唯一的地址，称为 IP 地址。

Internet 中使用的 IPV4 版本的 TCP/IP 协议标准，规定 IP 地址由 32 位二进制数码组成。IP 地址在计算机中一般采用 32 位二进制位表示，如 IP 地址 10101100 00010000 00000000 00000001；为了人工阅读方便，一般采用以点分十进制表示方法，即 32 位二进制数码组成的 IP 地址，每 8 位为一组，共分为 4 组，中间用“.”隔开。例如，IP 地址 10101100 00010000 00000000 00000001，用点分十进制表示法，可记为 172.16.0.1。

以 127 开头的 IP 地址（如 127.0.0.1）为本机回送地址（Loopback Address），主要用于网络软件测试及本地机进程间通信，无论什么程序，一旦使用回送地址发送数据，协议软件立即返回之，不进行任何网络传输。

#### 2. 域名系统

Internet 上计算机之间的 TCP/IP 通信是通过 IP 地址来进行的，Internet 上的计算机都应有一个唯一的 IP 地址。但是 IP 地址是基于数字来标识的，如 64.233.189.104，可记忆性差，十分不友好，所以人们使用比较友好的计算机域名，如 www.google.com。

Internet 使用域名系统（Domain Name System, DNS）来管理计算机域名与 IP 地址的对应关系。用户先在域名系统中注册域名及与其对应的 IP 地址。

当需要使用域名进行通信时，DNS 客户机通过查询 DNS 服务器将此域名解析为相对应的 IP 地址信息，然后通过 IP 地址进行通信。

### 18.1.4 统一资源定位器 URL

IP 地址用来标识 Internet 上的主机，而位于 Internet 主机上的资源（如各种文档、图像等）则通过统一资源定位器来标识。

URL（Uniform Resource Locator，统一资源定位器）是专为标识 Internet 网上资源位置而设的一种编址方式。通过 URL 可以访问 Internet 上的各种网络资源，比如最常见的 WWW、FTP 站点。浏览器通过解析给定的 URL，可以在网络上查找相应的文件或其他资源。URL



一般由以下几个部分组成：

传输协议：//主机 IP 地址（或域名地址）[: 端口号] /资源所在路径和文件名

其中，传输协议是指访问该资源所使用的访问协议；主机 IP 地址（或域名地址）是指资源所在的 Internet 主机；端口号是指主机上提供资源的服务的 TCP/IP 端口，如 http 使用 WWW 服务（默认端口为 80），ftp 表示 FTP 服务（默认端口为 21）；路径是指资源所在路径和文件名。例如：

`http://www.baidu.com/`

`http://home.yahoo.com:80/index.html`

`http://www.gamelan.com:80/Gamelan/network.html#BOTTOM`

`http://user:passwd@www.google.com/pages/index.html?key1=data1&key2=data2#faq`

注：TCP/IP 系统中的端口号是一个 16 位的数字，它的范围是 0 ~ 65 535。

## 18.2 基于 socket 模块的网络编程

### 18.2.1 socket 概述

#### 1. 套接字

套接字是网络中两个应用程序之间通信的端点。网络上的两个程序通过一个双向的通信连接实现数据的交换，这个双向链路的一端就是一个 socket。

基于 TCP/IP 通信协议的 socket，是由一个 IP 地址和一个端口号唯一确定。

TCP/IP 协议的传输层包含两个传输协议：面向连接的 TCP 和非面向连接的 UDP。TCP 广泛用于各种可靠的传输，例如 HTTP、FTP、SMTP 等都使用 TCP 传输协议；UDP 不保证可靠传输，但其传输更简单高效，故适合于实时交互性应用，如音频、视频会议等。TCP 和 UDP 的程序架构各不相同。UDP 使用数据报传输数据。

#### 2. TCP 通信程序设计

基于 socket 的面向连接的 TCP 网络程序的 C/S 架构如图 18-3 所示。

基于套接字的 TCP Server 的网络编程一般包括以下基本步骤：

- (1) 创建 socket 对象。
- (2) 将 socket 绑定到指定地址上。
- (3) 准备好套接字，以便接收连接请求。
- (4) 通过 socket 对象方法 `accept`，等待客户请求连接。
- (5) 服务器和客户机通过 `send` 和 `recv` 方法通信（传输数据）。
- (6) 传输结束，调用 socket 的 `close` 方法以关闭连接。

其中，第（5）步是实现程序功能的关键步骤，其他步骤在各种程序中基本相同。

基于套接字的 TCP Client 的网络编程一般包括以下基本步骤：

- (1) 创建 socket 对象。
- (2) 通过 socket 对象方法 `connect` 连接服务器。
- (3) 客户机和服务器通过 `send` 和 `recv` 方法通信（传输数据）。



(4) 传输结束，调用 socket 的 close 方法以关闭连接。  
其中，第 (3) 步是实现程序功能的关键步骤，其他步骤在各种程序中基本相同。

3. UDP 通信程序设计

基于 socket 的面向非连接的 UDP 网络程序的 C/S 架构如图 18-4 所示。

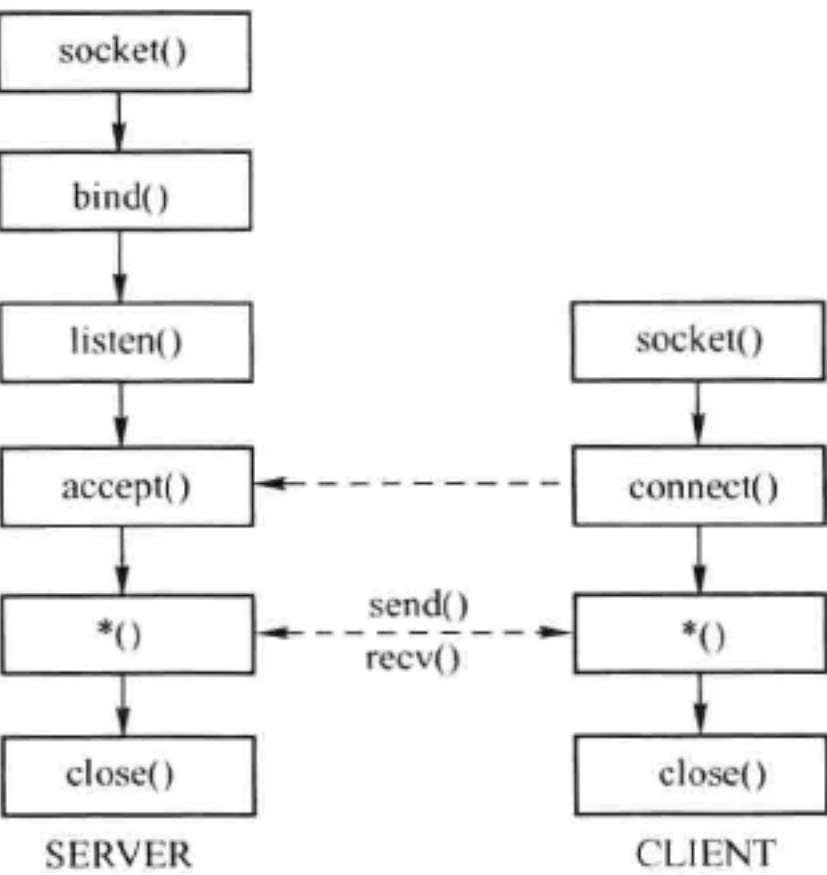


图 18-3 基于 socket 的 TCP 程序架构

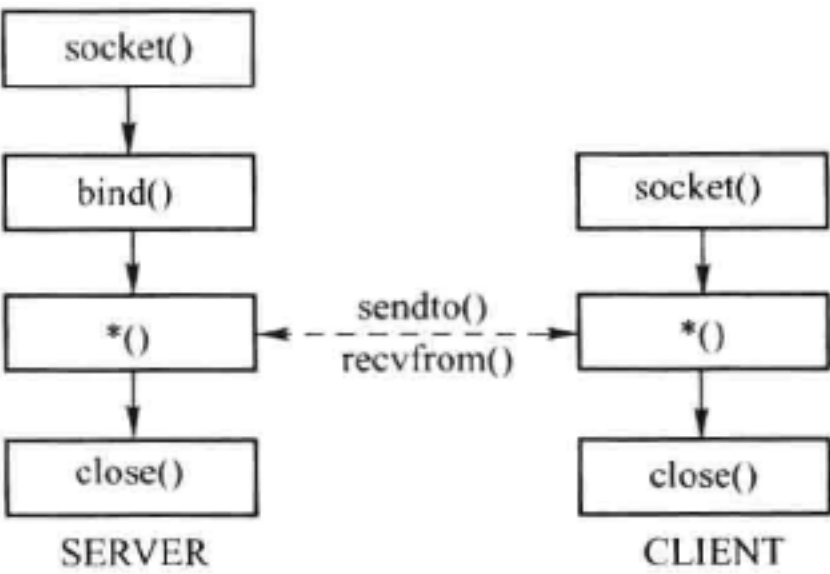


图 18-4 基于 socket 的 UDP 程序架构

基于套接字的 UDP Server 的网络编程一般包括以下基本步骤：

- (1) 创建 socket 对象。
- (2) 将 socket 绑定到指定地址上。
- (3) 服务器和客户机通过 send 和 recv 方法通信（传输数据）。
- (4) 传输结束，调用 socket 的 close 方法以关闭连接。

其中，第 (3) 步是实现程序功能的关键步骤，其他步骤在各种程序中基本相同。

基于套接字的 UDP Client 的网络编程一般包括以下基本步骤：

- (1) 创建 socket 对象。
- (2) 客户机和服务器通过 send 和 recv 方法通信（传输数据）。
- (3) 传输结束，调用 socket 的 close 方法以关闭连接。

其中，第 (2) 步是实现程序功能的关键步骤，其他步骤在各种程序中基本相同。

18.2.2 创建 socket 对象

可以使用 socket 对象的构造函数创建一个 socket 对象：

```
socket(family = 2, type = 1, proto = 0, fileno = None)
```

各参数的意义如下。

- family：地址系列。默认为 AF\_INET (2, socket 模块中的常量)，对应于 IPV4；AF\_UNIX，对应于 Unix 的进程间通信；AF\_INET6，对应于 IPV6。
- type：socket 类型。默认为 SOCK\_STREAM，对应于 TCP 流套接字；SOCK\_DGRAM，对应于 UDP 数据报套接字；SOCK\_RAW，对应于 raw 套接字。

例如：

```
>>> import socket
>>> s1 = socket.socket() #创建用于 TCP 通信的套接字
>>> s2 = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #创建用于 TCP 通信的套接字
>>> s3 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) #创建用于 UDP 通信的套接字
```

### 18.2.3 将服务器端 socket 绑定到指定地址上

#### 1. 主机名和 IP 地址

socket 模块包含若干函数，用于获取主机名和 IP 地址等信息：

socket.gethostname(): 返回主机名。

socket.gethostbyname(hostname): 返回主机名的 IP 地址。

socket.gethostbyname\_ex(hostname): 返回扩展信息元组：(hostname, aliaslist, ipaddrlist)。

getfqdn([name]): 返回全限定名称。

gethostbyaddr(ip\_address): 返回 IP 地址的主机信息元组：(hostname, aliaslist, ipaddrlist)。

getservbyname(servicename[, protocolname]): 返回服务所使用的端口号。

例如：

```
>>> socket.gethostname() #返回本机的主机名:'PC201307031137'
>>> socket.gethostbyname('www.baidu.com') # '115.239.210.26'
>>> socket.gethostbyname_ex('www.baidu.com')
('www.a.shifen.com', ['www.baidu.com'], ['115.239.210.26', '115.239.210.27'])
>>> socket.getservbyname('http', 'tcp') #80
```

#### 2. 绑定 socket 对象到 IP 地址

创建服务器端 socket 对象后，必须把对象绑定到某个 IP 地址，然后客户机才可以与之连接。可以使用对象方法 bind 将 socket 绑定到指定 IP 地址上：

**sock.bind(address)**

其中，address 是要绑定的 IP 地址，对应 IPV4 的地址为一个元组：

(主机名或 IP 地址, 端口号)

例如：

```
>>> sock = socket.socket()
>>> sock.bind(('localhost', 8000)) #绑定到本机 localhost 端口号 8000
>>> sock1 = socket.socket()
>>> sock1.bind((socket.gethostname(), 8001)) #绑定到本机端口号 8001
>>> sock2 = socket.socket()
>>> sock2.bind(('127.0.0.1', 8002)) #绑定到本机 127.0.0.1 端口号 8002
```

### 18.2.4 服务器端 socket 开始侦听

创建服务器端 socket 对象并绑定到 IP 地址后，可以使用对象方法 listen 和 accept 进行侦听和接收连接：

**sock.listen(backlog)**

其中，backlog 是最多连接数，至少为 1，接到连接请求后，这些请求必须排队，如果队列已满，则拒绝请求。例如：

```
>>> sock = socket.socket()
>>> sock.bind(('localhost',8000)) #绑定到本机 localhost 端口号 8000
>>> sock.listen(5) #开始侦听,连接队列长度为 5
```

### 18.2.5 连接和接收连接

客户机端 socket 对象通过 connect 方法尝试建立到服务器端 socket 对象的连接:

```
client_sock.connect(address) #client_sock 连接到绑定到 address 的服务器端 socket 对象
```

其中, address 是要连接的服务器端 socket 对象的绑定的 IP 地址, 对应 IPV4 的地址为一个元组。

服务器端 socket 对象通过 accept 方法, 进入 waiting (阻塞) 状态。接收到来自客户请求连接时, accept 方法建立连接并返回服务器。accept 方法返回一个含有 2 个元素的元组: (clientsocket, address), 其中, clientsocket 是新建的 socket 对象, 服务器通过它与客户通信; address 为对应的 IP 地址:

```
clientsocket,address = server_sock.accept()
```

### 18.2.6 发送和接收数据

对于面向连接的 TCP 通信程序, 客户机和服务器建立连接后, 通过 socket 对象的 send 和 recv 方法分别发送和接收数据。

send(bytes): 发送数据 bytes, 返回实际发送的字节数。

sendall(bytes): 发送数据 bytes, 持续发送; 成功返回 None, 否则出错。

recv(bufsize): 接收数据, 返回接收到的数据: bytes 对象。

其中, bytes 为字节系列, bufsize 为一次接收的数据的最大字节数。

对于非面向连接的 UDP 通信程序, 客户机和服务器不需要预先建立连接, 直接通过 socket 对象的 sendto 指定发送目标地址参数, recvfrom 方法返回接收的数据以及发送源地址。

sendto(bytes, address): 发送数据 bytes 到地址 address, 返回实际发送的字节数。

recvfrom(bufsize[, flags]): 接收数据, 返回元组: (bytes, address)。

其中, bytes 为字节系列, address 是发送的目标地址, bufsize 为一次接收的数据的最大字节数。

### 18.2.7 简单 TCP 程序: Echo Server

基于 TCP 的 Echo Server 包括服务器、客户机两个部分: 服务端应用程序和客户机应用程序。服务端应用程序创建一个 socket 并绑定到某个 IP 地址: 端口号上, 然后侦听 listen, 并使用阻塞方法 accept 以等待客户机连接请求; 客户机创建一个 socket, 并建立到服务器的连接; 客户机循环接收用户数据并发送数据到服务器, 服务器接收数据后回送 (Echo) 给客户机。客户机输入空数据时, 关闭 socket 并终止运行; 服务器接收到空数据时, 关闭 socket 并终止运行。运行效果如图 18-5 所示。

注: 读者可以在单机上同时运行服务端应用程序和客户机应用程序。但建议在不同的机器上运行服务端应用程序和客户机应用程序。如果服务端应用程序在其他机器上运行, 请把





图 18-5 简单 TCP 程序: Echo Server

代码中创建 socket 对象时的服务器地址“127.0.0.1”修改为对应服务器的机器地址。

**【例 18-1】**简单 TCP 程序 (ChatServer.py): Echo Server。服务端应用程序 ChatServer。

|                                                                               |                           |
|-------------------------------------------------------------------------------|---------------------------|
| <code>import socket</code>                                                    | #导入 socket 模块             |
| <code>serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)</code> | #创建服务器 socket             |
| <code>serversocket.bind(('127.0.0.1', 8000))</code>                           | #绑定到 IP 地址和端口号            |
| <code>serversocket.listen(1)</code>                                           | #开始侦听, 队列长度为 1            |
| <code>clientsocket, clientaddress = serversocket.accept()</code>              | #使用阻塞方法 accept 以等待客户机连接请求 |
| <code>print('Connection from ', clientaddress)</code>                         | #接收客户机请求后输出客户机的信息         |
| <code>while 1: #循环以接收和回送客户机数据</code>                                          |                           |
| <code>data = clientsocket.recv(1024)</code>                                   | #接收数据                     |
| <code>if not data: break</code>                                               | #接收到空数据时, 终止循环            |
| <code>print('Received from client: ', repr(data))</code>                      | #输出接收到的数据, repr 函数转换为字符串  |
| <code>print('Echo: ', repr(data))</code>                                      | #输出发送到客户机数据的信息            |
| <code>clientsocket.send(data)</code>                                          | #回送数据到客户机                 |
| <code>clientsocket.close()</code>                                             | #关闭客户机 socket             |
| <code>serversocket.close</code>                                               | #关闭服务器 socket             |

**【例 18-2】**简单 TCP 程序 (ChatClient.py): Echo Server。客户机应用程序 ChatClient。

|                                                                               |                           |
|-------------------------------------------------------------------------------|---------------------------|
| <code>import socket</code>                                                    | #导入 socket 模块             |
| <code>clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)</code> | #创建客户机 socket             |
| <code>clientsocket.connect(('127.0.0.1', 8000))</code>                        | #连接到服务器                   |
| <code>while 1: #循环以接收用户输入, 并发送到服务器, 接收服务器的回送数据</code>                         |                           |
| <code>data = input('&gt;')</code>                                             | #接收用户输入数据                 |
| <code>clientsocket.send(data.encode())</code>                                 | #把数据转换为 bytes 对象, 并发送到服务器 |
| <code>if not data: break</code>                                               | #如果数据为空, 终止循环             |
| <code>newdata = clientsocket.recv(1024)</code>                                | #接收服务器的回送数据               |
| <code>print('Received from server: ', repr(newdata))</code>                   | #输出接收到数据                  |
| <code>clientsocket.close()</code>                                             | #关闭客户机 socket             |

## 18.2.8 简单 UDP 程序: Echo Server

基于 UDP 的网络程序是无连接的, 服务器和客户端不需要实现建立连接, 发送数据时

直接指定地址参数，接收数据时，同时返回地址。通信双方地位平等，传输无法保证对方能够接收到数据报。

基于 UDP 的 Echo Server 包括服务器和客户机两个部分：服务端应用程序和客户机应用程序。服务端应用程序创建一个 socket 并绑定到某个 IP 地址：端口号上，然后循环使用 `recvfrom` 接收数据（返回数据和客户机地址），并使用 `sendto` 回送数据到客户机地址；客户机创建一个 socket，然后循环使用 `sendto` 发送用户输入的数据到服务器，并接收服务器回送的数据。客户机输入空数据时，关闭 socket 并终止运行；服务器接收到空数据时，关闭 socket 并终止运行。运行效果如图 18-6 所示。



图 18-6 简单 UDP 程序：Echo Server

注：读者可以在单机上同时运行服务端应用程序和客户机应用程序。但建议在不同的机器上运行服务端应用程序和客户机应用程序。如果服务端应用程序在其他机器上运行，请把代码中创建 socket 对象时的服务器地址“127.0.0.1”修改为对应服务器的机器地址。

【例 18-3】简单 UDP 程序：Echo Server。服务端应用程序 ChatServerUDP。

```
#ChatServerUDP.py
import socket #导入 socket 模块
serversocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) #创建服务器 socket
serversocket.bind(('127.0.0.1', 8000)) #绑定到 IP 地址和端口号
while 1: #循环以接收和回送客户机数据
 data, address = serversocket.recvfrom(1024) #接收数据, 返回数据和客户机地址
 if not data: break; #接收到空数据时, 终止循环
 print('Received from client: ', address, repr(data)) #输出接收到的数据, repr 函数转换为字符串
 print('Echo: ', repr(data)) #输出发送到客户机数据的信息
 serversocket.sendto(data, address) #发送数据到客户机
serversocket.close() #关闭服务器 socket
```

【例 18-4】简单 UDP 程序：Echo Server。客户机应用程序 ChatClientUDP。

```
#ChatClientUDP.py
import socket #导入 socket 模块
clientsocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) #创建客户机 socket
while 1: #循环以接收用户输入, 并发送到服务器, 接收服务器的回送数据
 data = input('> ') #接收用户输入数据
 clientsocket.sendto(data.encode(), ('127.0.0.1', 8000)) #把数据转换为 bytes 对象, 并发送
```



```

#如果数据为空,终止循环
if not data: break
newdata = clientsocket.recvfrom(1024) #接收服务器的回送数据
print('Received from server:', repr(newdata)) #输出接收到数据
clientsocket.close() #关闭客户机 socket

```

## 18.2.9 UDP 程序: Quote Server

Quote Server 实现 Quote of the day (每日名言) 功能: 客户机发送一个数据报到 Quote 服务器 (相当于请求); 服务器使用接收来自客户机的数据报 (请求); 服务器从格言列表中读取一句名言, 并作为数据报发送给客户机; 客户机接收 Quote 服务器的数据报 (包含一句名言), 并显示该名言。运行结果如图 18-7 所示。

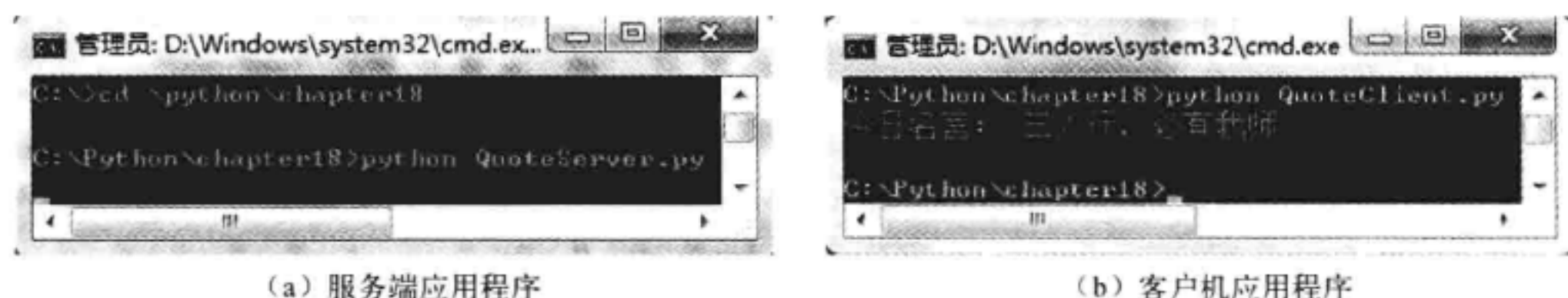


图 18-7 UDP 程序: Quote Server

注: 读者可以在单机上同时运行服务端应用程序和客户机应用程序。但建议在不同的机器上运行服务端应用程序和客户机应用程序。

**【例 18-5】UDP 程序 (QuoteServer.py):** 实现 Quote of the day (每日名言) 功能。服务器应用程序 QuoteServer。

```

import socket, random #导入 socket 和 random 模块
quotes = ['不妄求,则心安,不妄做,则身安','多门之室生风,多言之人生祸','人之心胸,多欲则窄,寡欲则宽','三人行,必有我师','滴水穿石,磨杵成针','是非天天有,不听自然无','积德为产业,强胜于美宅良田']
serversocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) #创建服务器 socket
serversocket.bind(('127.0.0.1', 8002)) #绑定到 IP 地址和端口号
while 1: #循环以接收和回送客户机数据
 data, address = serversocket.recvfrom(1024) #接收数据,返回数据和客户机地址
 quote = random.choice(quotes) #从 Quotes 列表中随机选择一个项目
 serversocket.sendto(quote.encode(), address) #把数据转换为 bytes 对象,并发送数据到客户机
serversocket.close() #关闭服务器 socket

```

**【例 18-6】UDP 程序 (QuoteClient.py):** 实现 Quote of the day (每日名言) 功能。客户机应用程序 QuoteClient。

```

import socket #导入 socket 模块
clientsocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) #创建客户机 socket
clientsocket.sendto(b'hello', ('127.0.0.1', 8002)) #把数据转换为 bytes 对象,并发送到服务器
newdata, address = clientsocket.recvfrom(1024) #接收服务器的回送数据
print('今日名言:', newdata.decode()) #接收到数据解码为字符串,并输出

```



```
clientsocket.close()
```

```
#关闭客户机 socket
```

## 18.3 基于 urllib 模块的网络编程

urllib 模块包含 4 个子模块：urllib.request（打开和读取 URL），urllib.parse（解析 URL），urllib.error（urllib.request 引发的异常），urllib.robotparser（解析 robots.txt 文件）。

### 18.3.1 打开和读取 URL 网络资源

使用 urllib.request 模块中的 urlopen() 函数，可以打开 URL：

```
urllib.request.urlopen(url, data = None) #打开指定的 url
```

其中，url 可以为字符串或 Request 对象，可选参数 data 是向服务器传送的数据。对于 HTTP/HTTPS 协议，urlopen 返回 response 对象（file-like 的对象），可以从中读取和输出内容（参见第 7 章）。例如：

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.baidu.com') #打开 URL 资源
>>> print(f.read(200)) #读取 200 个字节,返回 bytes 对象并输出
b'<!DOCTYPE html><!--STATUS OK--><html><head><meta http-equiv="content-type" content="text/html; charset=utf-8"><title>\xe7\x99\xbe\xe5\xba\xa6\xe4\xb8\x80\xe4\xb8\x8b\xef\xbc\x8c\xe4\xbd\xa0\xe5\xb0\xb1\xe7\x9f\xa5\xe9\x81\x93</title><style>html, body { height:100% | html | overflow-y:auto '
>>> f = urllib.request.urlopen('http://www.baidu.com') #重新打开 URL 资源
>>> print(f.read(200).decode()) #读取 200 个字节,返回 bytes 对象,转换为字符串并输出
>>> with urllib.request.urlopen('http://www.baidu.com/') as f: #重新打开 URL 资源
 print(f.read(200).decode('utf-8')) #读取返回 bytes 对象转换为字符串并输出
<!DOCTYPE html><!--STATUS OK--><html><head><meta http-equiv="content-type" content="text/html; charset=utf-8"><title>百度一下,你就知道</title><style>html, body { height:100% | html | overflow-y:auto
```

### 18.3.2 创建 Request 对象

urllib.request 模块中 Request 对象的构造函数为：

```
urllib.request.Request(url, data = None, headers = {}, origin_req_host = None, unverifiable = False, method = None)
```

其中，url 为字符串；可选参数 data（需要编码为 utf-8）是向服务器传送的数据；header 是字典，为传递的 header 数据。

Request 对象 request 包含下列主要属性和方法。

request.full\_url：Request 对象 request 的 URL。

request.host：主机和端口号。

request.data：向服务器传送的数据。

request.method：请求方法 GET 和 POST。

request.add\_data(data)：添加向服务器传送的数据。

request.add\_header(key, val)：添加向服务器传送的 header。

**【例 18-7】** Request 对象示例 (request.py)。

```
import urllib.request #导入 urllib.request 模块
def getURLInfo(url,data,headers):
 req = urllib.request.Request(url,data,headers) #创建 Request 对象
 print('Full url:',req.full_url) #URL
 print('Host:',req.host) #主机和端口号
 print('Data:',req.data) #向服务器传送的数据
 print('Method:',req.method) #请求方法
#测试代码
if __name__ == '__main__':
 url = 'http://www.baidu.com/s'
 values = {'wd':'python'}
 data = urllib.parse.urlencode(values)
 data = data.encode(encoding='UTF8')
 headers = {'User-Agent':'Mozilla/4.0 (compatible;MSIE 5.5;Windows NT)'}
 getURLInfo(url,data,headers)
```

运行结果如下：

```
Full url: http://www.baidu.com/s
Host: www.baidu.com
Data: b'wd=python'
Method: None
```

## 18.4 基于 http 模块的网络编程

http 模块包含 4 个子模块：http.client（低级别的 HTTP 协议客户端，高级别的 URL 打开则使用 urllib.request）；http.server（基于 socketserver 的 HTTP 服务器类）；http.cookies（使用 cookies 实现状态管理的工具）；http.cookiejar（提供 cookies 的持久性）。

一般不直接使用 http.client 模块访问 HTTP 服务器，建议使用 urllib.request 模块。事实上，urllib.request 模块使用 http.client 模块处理包含 http 和 https 的 URL。

## 18.5 基于 ftplib 模块的网络编程

ftplib 模块包含 FTP 对象，实现了 FTP 客户端协议，用于访问 FTP 服务器。使用 ftplib 模块，可以编写批量处理 FTP 服务器内容的程序。

### 18.5.1 创建 FTP 对象

ftplib 模块中 FTP 对象的构造函数为：

```
ftplib.FTP(host='',user='',passwd='',acct='',timeout=None,source_address=None)
```

其中，host 为 FTP 服务器；user、passwd 和 acct 为用于登录的用户名、密码和账户（账户信息大部分 FTP 服务器不支持）；timeout 为超时时间；source\_address 为元组（host, port）。创

建 FTP 对象时, 如果指定了 host, 则自动调用对象方法 connect (host) 连接到 FTP 服务器; 如果指定了 user, 则自动调用对象方法 login (user, passwd, acct) 登录到 FTP 服务器。

FTP 对象 ftp 包含下列主要属性和方法 (通常对应于 FTP 命令)。

ftp.set\_debuglevel(level): 设置调试级别, 0 (默认值, 无调试信息)、1 (基本调试信息)、2 以上 (详细调试信息)。

ftp.connect(host='', port=0, timeout=None, source\_address=None): 连接到 FTP 服务器。

ftp.getwelcome(): 返回欢迎信息。

ftp.login(user='anonymous', passwd='', acct=''): 登录到 FTP 服务器。

ftp.abort(): 终止传输。

ftp.retrbinary(cmd, callback, blocksize=8192, rest=None): 下载文件 (二进制传输模式)。

ftp.retrlines(cmd, callback=None): 下载文件 (文本传输模式)。

ftp.storbinary(cmd, file, blocksize=8192, callback=None, rest=None): 上传文件 (二进制传输模式)。

ftp.storlines(cmd, file, callback=None): 上传文件 (文本传输模式)。

ftp.set\_pasv(boolean): 设置传输模式, True 为被动模式; False 为主动模式。

ftp.cwd(pathname): 切换当前目录。

ftp.mkd(pathname): 创建目录。

ftp.pwd(): 打印当前目录。

ftp.rmd(dirname): 删除目录。

ftp.mlsd(path="", facts=[]): 列目录, 取代旧版本的 dir 方法。

ftp.rename(fromname, toname): 文件重命名。

ftp.delete(filename): 删除文件。

ftp.size(filename): 获取文件大小。

ftp.quit(): 退出 (polite way)。

ftp.close(): 关闭。注: 退出或关闭 FTP 对象, 不能继续操作 FTP 对象。

例如:

```
>>> from ftplib import FTP
>>> ftp = FTP("ftp1.at.proftpd.org")
>>> ftp.login() #'230 Anonymous access granted,restrictions apply '
>>> ftp.dir()
-rw-rw-r-- 1 ftp ftp 451 Jul 1 2005 README.MIRRORS
drwxrwxr-x 3 ftp ftp 4096 Jul 1 2005 devel
drwxrwxr-x 3 ftp ftp 4096 Dec 2 2010 distrib
drwxrwxr-x 4 ftp ftp 4096 Jul 1 2005 historic
>>> ftp.cwd('devel') #'250 CWD command successful '
>>> ftp.dir()
drwxrwxr-x 2 ftp ftp 4096 Sep 14 00:05 source
```



## 18.5.2 创建 FTP\_TLS 对象

ftplib 模块中 FTP\_TLS 对象继承于 FTP 对象。FTP\_TLS 对象的构造函数为：

```
ftplib.FTP_TLS(host='', user='', passwd='', acct='', keyfile=None, certfile=None, context=None, timeout=None, source_address=None)
```

其中，keyfile 和 certfile 为证书文件，context 为 ssl.SSLContext。其他参数意义同 FTP 构造函数。

FTP 对象 ftps 增加了下列主要属性和方法。

ftps.ssl\_version：SSL 版本（默认为 TLSv1）。

ftps.auth()：设置加密控制连接。

ftps.ccc()：取消控制通道，回到明文传输。

ftps.prot\_p()：设置为加密传输。

ftps.prot\_c()：设置为明文传输。

例如：

```
>>> from ftplib import FTP_TLS
>>> ftps = FTP_TLS('ftp. python. org')
>>> ftps.login() #匿名登录安全控制通道
>>> ftps.prot_p() #安全数据连接(加密传输)
>>> ftps.retrlines('LIST') #罗列目录清单
total 9
drwxr-xr-x 8 root wheel 1024 Jan 3 1994 .
drwxr-xr-x 8 root wheel 1024 Jan 3 1994 ..
drwxr-xr-x 2 root wheel 1024 Jan 3 1994 bin
drwxr-xr-x 2 root wheel 1024 Jan 3 1994 etc
d-wxrw-r-x 2 ftp wheel 1024 Sep 5 13:43 incoming
drwxr-xr-x 2 root wheel 1024 Nov 17 1993 lib
drwxr-xr-x 6 1094 wheel 1024 Sep 13 19:07 pub
drwxr-xr-x 3 root wheel 1024 Jan 3 1994 usr
-rw-r--r-- 1 root root 312 Aug 1 1994 welcome.msg
'226 Transfer complete.'
>>> ftps.quit() #退出
```

## 18.6 基于 poplib 和 smtplib 模块的网络编程

poplib 模块提供了对 POP3 协议的支持，smtplib 模块提供了对 SMTP 协议的支持。使用 poplib 和 smtplib，可以实现接收和发送邮件的功能。

### 18.6.1 使用 poplib 接收邮件

poplib 模块中 POP3 对象用于连接到 POP3 服务器，其构造函数为：

```
poplib.POP3(host, port=POP3_PORT[, timeout])
```

其中, host 和 port 为 POP3 服务器及其端口号, timeout 为超时时间。

POP3 对象 pop3 包含下列主要属性和方法。

pop3.set\_debuglevel(level): 设置调试级别, 0 (默认值, 无调试信息)、1 (基本调试信息)、2 以上 (详细调试信息)。

pop3.user(username): 发送 user 命令, 响应需要密码。

pop3.user(username): 发送密码, 返回邮件数和邮箱大小。锁定邮箱直至调用 quit()。

pop3.getwelcome(): 返回欢迎信息。

pop3.stat(): 返回邮箱状态, 结果为元组: (message count, mailbox size)。

pop3.list([which]): 返回邮件列表。

pop3.retr(which): 接收邮件, 并设置其状态为已读。

pop3.dele(which): 设置邮件删除标记, 调用 quit() 时删除邮件。

pop3.rset(): 清除邮件删除标记。

pop3.noop(): 空操作, 用于保持连接状态。

pop3.quit(): 注销退出, 释放邮箱锁定, 释放连接。

pop3.top(which, howmuch): 接收邮件部分内容。

pop3.uidl(which = None): 返回邮件摘要列表。

**【例 18-8】** POP3 示例 (pop3.py)。

```
import getpass, poplib
host = 'YourPop3Host' #POP3 服务器的主机名或 IP 地址,运行时需修改为对应的值
port = 110 #POP3 服务器的端口号,默认为 110,运行时需修改为对应的值
pop3 = poplib.POP3(host, port = port) #创建 POP3 对象
pop3.user(getpass.getuser()) #用户名
pop3.pass_(getpass.getpass()) #密码
numMessages = len(pop3.list()[1]) #邮件数
for i in range(numMessages): #接收邮件
 for j in pop3.retr(i+1)[1]:
 print(j)
```

## 18.6.2 使用 smtplib 发送邮件

poplib 模块中 SMTP 对象用于连接到 SMTP/ESMTP 服务器, 其构造函数为:

**smtplib.SMTP(host='', port=0, local\_hostname=None[, timeout], source\_address=None)**

其中, host 和 port 为 SMTP/ESMTP 服务器及其端口号; local\_hostname 为本地主机; timeout 为超时时间。如果指定了 host 和 port, 则自动调用对象方法 connect() 连接到 SMTP/ESMTP 服务器

SMTP 对象 smtp 包含下列主要属性和方法。

smtp.set\_debuglevel(level): 设置调试级别, 0 (默认值, 无调试信息)、1 (基本调试信息)、2 以上 (详细调试信息)。

smtp.docmd(cmd, args=''): 发送命令到服务器。

smtp.connect(host='localhost', port=0): 连接到服务器。

smtp.login(user, password): 登录到服务器。

`smtp.sendmail(from_addr, to_addrs, msg, mail_options = [], rcpt_options = [])`: 发送邮件。

`smtp.quit()`: 注销退出, 释放邮箱锁定, 释放连接。

**【例 18-9】SMTP 示例 (smtp.py)。**

```
import smtplib
def prompt(prompt):
 return input(prompt).strip()
fromaddr = prompt("From: ")
toaddrs = prompt("To: ").split()
print("输入信息, ^D (Unix) or ^Z (Windows) 结束输入:")
#添加 From: 和 To: 头信息
msg = ("From: %s\r\nTo: %s\r\n\r\n" % (fromaddr, ",".join(toaddrs)))
while True:
 try:
 line = input()
 except EOFError:
 break
 if not line:
 break
 msg = msg + line
print("信息长度为:", len(msg))
server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

## 18.7 复习题

### 一、填空题

1. TCP/IP 协议模型把 TCP/IP 协议族分成四个层次: \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。
2. Internet 采用一种全局通用的地址格式, 为网络中的每一台主机都分配一个唯一的地址, 称为\_\_\_\_\_。
3. Internet 使用\_\_\_\_\_来管理计算机域名与 IP 地址的对应关系。
4. IP 地址用来标识 Internet 上的主机, 而位于 Internet 主机上的资源 (如各种文档、图像等) 则通过\_\_\_\_\_来标识。
5. TCP/IP 协议的传输层包含两个传输协议: 面向连接的\_\_\_\_\_和非面向连接的\_\_\_\_\_。
6. 创建服务器端 socket 对象并绑定到 IP 地址后, 可以使用\_\_\_\_\_和\_\_\_\_\_对象方法进行侦听和接收连接。



7. 客户机端 socket 对象通过\_\_\_\_\_方法尝试建立到服务器端 socket 对象的连接。

## 二、思考题

1. 如何用 Python 接收和发送邮件?
2. 基于套接字的 TCP Server 的网络编程一般包括哪些基本步骤?
3. 基于套接字的 TCP Client 的网络编程一般包括哪些基本步骤?
4. 基于套接字的 UDP Server 的网络编程一般包括哪些基本步骤?
5. 基于套接字的 UDP Client 的网络编程一般包括哪些基本步骤?
6. 对于面向连接的 TCP 通信程序, 客户机和服务器建立连接后, 如何发送和接收数据?
7. 对于非面向连接的 UDP 通信程序, 客户机和服务器间如何发送和接收数据?
8. urllib 模块包含哪 4 个子模块, 分别实现什么功能?
9. http 模块包含哪 4 个子模块, 分别实现什么功能?
10. 如何实现基于 ftplib 的网络编程?
11. 如何使用 poplib 模块和 smtplib 模块实现邮件接收和发送功能?

## 18.8 上机实践

1. 参照例 18-1 和例 18-2, 编写 Echo Server 简单 TCP 程序, 包括服务端应用程序和客户机应用程序。服务端应用程序创建一个 socket 并绑定到某个 IP 地址: 端口号上, 然后侦听 listen, 并使用阻塞方法 accept 以等待客户机连接请求; 客户机创建一个 socket, 并建立到服务器的连接; 客户机循环接收用户数据并发送数据到服务器, 服务器接收数据后回送给客户机。客户机输入空数据时, 关闭 socket 并终止运行; 服务器接收到空数据时, 关闭 socket 并终止运行。

2. 参照例 18-3 和例 18-4, 编写 Echo Server 简单 UDP 程序, 包括服务端应用程序和客户机应用程序。服务端应用程序创建一个 socket 并绑定到某个 IP 地址: 端口号上, 然后循环使用 recvfrom 接收数据 (返回数据和客户机地址), 并使用 sendto 回送数据到客户机地址; 客户机创建一个 socket, 然后循环使用 sendto 发送用户输入的数据到服务器, 并接收服务器回送的数据。客户机输入空数据时, 关闭 socket 并终止运行; 服务器接收到空数据时, 关闭 socket 并终止运行。

3. 参照例 18-5 和例 18-6, 编写 Quote Server UDP 程序, 实现 Quote of the day 功能: 客户机发送一个数据报到 Quote 服务器; 服务器使用接收来自客户机的数据报; 服务器从格言列表中读取一句名言, 并作为数据报发送给客户机; 客户机接收 Quote 服务器的数据报 (包含一句名言), 并显示该名言。

4. 参照例 18-7, 编写 Request 对象示例程序。
5. 参照例 18-8, 编写 POP3 示例程序。
6. 参照例 18-9, 编写 SMTP 示例程序。

# 第 19 章 系 统 管 理

Python 是一种非常适合系统管理员的脚本编写语言。使用 Python，可以实现各种复杂的系统管理工作。Python 标准库中包括下列系统管理相关模块。

- os 模块：与操作系统相关的函数。
- os.path：与路径相关的函数
- glob 模块：文件通配符操作。
- tempfile 模块：创建临时目录和文件。
- shutil 模块：与目录和文件操作相关的函数。
- subprocess 模块：用于执行其他程序。

## 本章要点：

- 
- ◆ 目录、文件和磁盘的基本操作；
  - ◆ 执行操作系统命令和运行其他程序；
  - ◆ 系统管理相关模块。
- 

## 19.1 目录、文件和磁盘的基本操作

### 19.1.1 创建目录

使用 os 模块中的 mkdir 函数，可以创建目录。

os.mkdir (path, mode = 0o777)：创建目录 path。

os.makedirs (path, mode = 0o777)：创建目录 path，以及所有的 path 中包含的上级目录。

其中，path 为指定目录。如果 path 已存在，则导致 FileExistsError。例如：

```
>>> import os
>>> os.makedirs(r 'c:\python\chapter19\temp\dir1 ')
>>> os.mkdir(r 'c:\python\chapter19\dir2 ')
```

### 19.1.2 临时目录和文件的创建

使用 tempfile 模块中的函数，可以创建临时目录和文件。

tempfile.mkdtemp(suffix = '', prefix = 'tmp ', dir = None)：创建并返回临时目录。

tempfile.mkstemp(suffix = '', prefix = 'tmp ', dir = None, text = False)：创建并返回临时文件。

tempfile.TemporaryDirectory(suffix = '', prefix = 'tmp ', dir = None)：调用 mkdtemp，创建临时目录。

`tempfile.TemporaryFile(mode='w+b', buffering=None, encoding=None, newline=None, suffix='', prefix='tmp', dir=None)`: 调用 `mkstemp`, 创建临时文件。

`tempfile.tempdir`: 设置临时目录对应的路径。

`tempfile.gettempdir()`: 获取临时目录。

临时目录和文件只是在程序运行时有效, 当文件关闭时, 系统自动删除。例如:

```
>>> import tempfile
>>> tempfile.gettempdir() # 'c:\\users\\robert\\appdata\\local\\temp'
>>> tempfile.mkstemp() # (3, 'c:\\users\\robert\\appdata\\local\\temp\\tmp5a')
>>> tempfile.mkdtemp() # 'c:\\users\\robert\\appdata\\local\\temp\\tmp4h5k'
```

### 19.1.3 切换当前工作目录

使用 `os` 模块中的 `chdir` 函数, 可以切换当前工作目录:

`os.chdir(path)` # 切换当前工作目录为 `path`

其中, `path` 为指定文件。如果找不到 `path`, 则导致 `FileNotFoundError`。例如:

```
>>> os.chdir(r'c:\python')
```

### 19.1.4 目录内容列表

使用 `os` 模块中的 `listdir` 函数, 可以显示一个目录中的文件/子目录列表:

`os.listdir(path='.')` # 返回指定目录 `path` 中的所有文件/子目录的列表

其中, `path` 为指定目录, 默认为当前目录: `'.'`。 `os.getcwd` 也表示当前目录。例如:

```
>>> os.listdir(r'c:\python33')
['DLLs', 'Doc', 'include', 'Lib', 'libs', 'LICENSE.txt', 'NEWS.txt', 'python.exe', 'pythonw.exe',
 'README.txt', 'tcl', 'Tools', 'w9xopen.exe']
```

### 19.1.5 文件通配符和 `glob.glob` 函数

使用 `glob` 模块中的 `glob` 函数, 可以获取满足指定模式的文件/目录列表:

`glob.glob(pathname)` # 返回满足指定模式 `pathname` 的文件/目录的列表

其中, `pathname` 为目录/文件模式, 可以包含通配符 `*` (0 或多个字符) 和 `?` (1 个字符)。例如:

```
>>> import glob
>>> glob.glob('*.py')
['binaryread.py', 'binarywrite.py', 'textread.py', 'textwrite.py']
```

### 19.1.6 遍历目录和 `os.walk` 函数

使用 `os` 模块的 `walk` 函数, 可以遍历指定的目录结构:

`os.walk(top, topdown=True, onerror=None, followlinks=False)` # 返回目录结构的迭代器

其中, `top` 为起始目录; `topdown` 若为 `False`, 则从下往上遍历。对于目录中结构中的每一个目录, 生成一个元组: `(dirpath, dirnames, filenames)`, `dirpath` 为目录, `dirnames` 为其中包含的子目录列表, `filenames` 为其中包含的文件列表。

使用 `os` 模块的 `join` 函数, 可以将目录名和文件名连接成全限定路径:



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

【例 19-1】输出指定目录的目录结构（oswalk.py）。

```
import re,os,os.path
def ls_py(top):
 for (dirname,subdirs,files) in os.walk(top):
 print('[' + dirname + ']')
 for fname in files:
 print(os.path.join(dirname,fname))
#测试代码
if __name__ == '__main__':
 path1 = r 'c:\python33 '
 ls_py(path1)
```

运行结果如下：

```
[c:\python33]
c:\python33\data1.txt
c:\python33\earth.gif
...(略)
```

### 19.1.7 判断文件/目录是否存在

使用 os.path 模块函数 exists，可以判断文件/目录是否存在：

```
os.path.exists(路径名)
```

例如：

```
>>> import os.path
>>> os.path.exists(r 'c:\abc ')
```

### 19.1.8 测试文件类型

文件名、目录名和链接名都是用一个字符串作为其标识符。使用 os.path 模块函数，可以判断其类型：

os.path.isfile (path)：路径 path 是否为文件类型。  
os.path.isdir (path)：路径 path 是否为目录类型。  
os.path.islink (path)：路径 path 是否为链接类型。  
os.path.ismount (path)：路径 path 是否为装载点类型。  
os.path.isabs (path)：路径 path 是否为绝对路径。

例如：

```
>>> os.path.isdir(r 'c:\python33 ') #True
```

### 19.1.9 文件的日期及其大小

使用 os.path 模块函数，可以获取文件和其他目录的属性：

os.path.getatime (path)：返回上次访问时间。  
os.path.getmtime (path)：返回上次修改时间。  
os.path.getctime (path)：返回创建时间。

`os.path.getsize(path)`: 返回指定路径 `path` 的大小。

其中, `path` 为指定文件目录路径, 默认为当前目录: `'.'`。例如:

```
>>> import os, path, time
>>> os.path.getctime(r'c:\python33\README.txt') #结果为秒:1368629348.0
>>> time.strftime('%c',time.gmtime(os.path.getctime(r'c:\python33\README.txt'))))
'05/15/13 14:49:08'
```

## 19.1.10 文件和目录的删除

### 1. 删除文件

使用 `os` 模块中的 `remove` 函数, 可以删除指定文件:

```
os.remove(path) #删除指定文件 path
```

其中, `path` 为指定文件。如果找不到 `path`, 则导致 `FileNotFoundError`; 如果 `path` 为目录, 则导致 `PermissionError`。例如:

```
>>> os.remove(r'c:\python\temp\1.txt')
```

### 2. 删除目录

使用 `os` 模块中的 `rmdir` 函数, 可以删除指定目录:

```
os.rmdir(path) #删除指定目录 path
```

其中, `path` 为指定目录。如果找不到 `path`, 则导致 `FileNotFoundError`; 如果目录不为空, 则导致 `OSError`。例如:

```
>>> os.rmdir(r'c:\python\temp')
```

使用 `shutil` 模块中的 `rmtree` 函数, 可以删除指定目录及目录下的所有内容。

```
shutil.rmtree(path) #删除指定目录 path
```

## 19.1.11 文件和目录复制、重命名和移动

### 1. 复制文件

使用 `shutil` 模块中的下列函数, 可以复制文件和目录。

`shutil.copy(src, dst)`: 拷贝文件 `src` 到 `dst`, 如果 `dst` 为目录, 则拷贝到 `dst` 目录下。

`shutil.copy2(src, dst)`: 拷贝文件 `src` 到 `dst`, 如果 `dst` 为目录, 则拷贝到 `dst` 目录下。

`shutil.copytree(src, dst, symlinks=False, ignore=None)`: 拷贝目录树 `src` 到 `dst`。

`shutil.move(src, dst)`: 将文件/目录 `src` 移动到 `dst`。

其中, `src` 为源路径; `dst` 为目标路径。`copy` 除了拷贝文件内容外, 还拷贝文件许可权限; `copy2` 则拷贝所有元数据, 包括创建时间和修改时间。例如:

```
>>> import shutil
>>> shutil.copytree(r'c:\python\chapter19\temp',r'c:\python\chapter19\temp1')
```

拷贝目录树时, 可以指定忽略的文件。通常使用 `shutil.ignore_patterns(*patterns)` 返回的函数对象。例如:

```
>>> shutil.copytree(r'c:\python',r'c:\python1',ignore=shutil.ignore_patterns('*~','*.pyc'))
```

## 19.1.12 磁盘的基本操作

使用 `shutil` 模块中的 `disk_usage` 函数, 可以获取磁盘空间的使用情况:

**shutil.disk\_usage(path)** #返回指定 path 上的磁盘的空间使用情况:(总数,已用,可用)

例如:

```
>>> import shutil;shutil.disk_usage(r'c:')
usage(total = 107370213376,used = 18693632000,free = 88676581376)
```

## 19.2 执行操作系统命令和运行其他程序

### 19.2.1 os.system 函数

使用 os 模块中的 system 函数,可以在 Python 程序中执行操作系统的命令和脚本,或运行其他程序:

**os.system(command)** #执行操作系统命令,返回命令执行结果的返回代码

例如:

```
>>> os.system('dir') #执行操作系统命令
>>> os.system('notepad.exe') #执行程序
```

### 19.2.2 os.popen 函数

使用 os 模块中的 popen 函数,可以在 Python 程序中执行操作系统的命令和脚本:

**os.popen(...)** #执行操作系统命令,返回打开的管道(相当于文件)

例如:

```
>>> os.popen(r'dir c:\python33') # < os._wrap_close object at 0x030E8470 >
>>> list(os.popen(r'dir c:\python33')) #['驱动器 C 中的卷是 Windows8_OS\n',...(略)]
```

### 19.2.3 subprocess 模块

subprocess 模块提供若干函数和对象,用于创建子进程、运行外部程序、连接到其输入/输出/错误管道、获取其返回值。subprocess 模块用于取代 os.system 和 os.popen 函数,提供更高级的功能。

subprocess 模块函数 call()/check\_call()/check\_output()用于执行外部程序。call()返回 returncode; 如果 returncode 不为 0, 则 check\_call()引发 CalledProcessError; check\_output()返回程序运行结果。

```
call(args, *, stdin = None, stdout = None, stderr = None, shell = False, timeout = None)
check_call(args, *, stdin = None, stdout = None, stderr = None, shell = False, timeout = None)
check_output(args, *, stdin = None, stderr = None, shell = False, universal_newlines = False, timeout = None)
```

其中, args 是外部程序及其参数列表。若 shell 设定为 True, 则执行操作系统命令。

例如:

```
>>> import subprocess
>>> subprocess.call(['notepad.exe',r'c:\python33\readme.txt']) #记事本中打开指定文件
>>> subprocess.check_output(r'python -h',shell=True)
b"usage: python [option]... [-c cmd | -m mod | file | -] [arg]
```



...(略)

如果需要与所创建的子进程进行高级通信,例如,传递输入参数,可以使用 subprocess 模块的 Popen 对象构造函数:

```
Popen(args, bufsize = -1, executable = None, stdin = None, stdout = None, stderr = None, preexec_fn
= None, close_fds = True, shell = False, cwd = None, env = None, universal_newlines = False, startupin-
fo = None, creationflags = 0, restore_signals = True, start_new_session = False, pass_fds = ())
```

Popen 对象包含下列方法和属性。

poll(): 检查子进程是否终止。

wait(timeout = None): 等待子进程终止。

communicate(input = None, timeout = None): 发送数据给子进程。

send\_signal(signal): 发送信号给子进程。

terminate(): 终止子进程。

kill(): 强行终止子进程。

stdin/stdout/stderr: 子进程的输入/输出/错误文件对象(构造函数 stdin/stdout/stderr 为 subprocess.PIPE 时)。

pid: 子进程的进程 id(当 shell = True, 即执行操作系统命令时, 为 Shell 的 pid)。

returncode: 子进程的返回值。

例如:

```
>>> import subprocess
>>> p = subprocess.Popen(['dir'], shell = True, stdout = subprocess.PIPE, stdin = subprocess.PIPE)
>>> stdoutdata, stderrdata = p.communicate()
>>> stdoutdata #b '\xc7\xfd\xbb\xaf\xc6\xf7...' (略)
```

## 19.3 获取终端的大小

通过 os 或 shutil 模块的 get\_terminal\_size 函数, 可以获取终端的大小, 以方便输出内容的格式化操作。

**os.get\_terminal\_size(fd = STDOUT\_FILENO):** 获取并返回终端大小。

**shutil.get\_terminal\_size(fallback = (80, 24)):** 获取并返回终端大小。

两者返回结果 os.terminal\_size 对象为元组的子类, 包含 (columns, lines), 及窗口的列数和行数。通常建议使用高级别的函数 shutil.get\_terminal\_size。

**【例 19-2】** 获取终端的大小示例 (get\_term\_size.py)。

```
import os,shutil
def get_term_size_test():
 sz = shutil.get_terminal_size()
 print('窗口大小:',sz)
 for i in range(sz.lines):
 print('*'*sz.columns)
if __name__ == '__main__':
 get_term_size_test()
```

运行过程和结果如下：

```
c:\Python\chapter19 > get_term_size.py
窗口大小: os. terminal_size(columns = 80 , lines = 25)

...(略)
```

## 19.4 文件压缩和解压缩

Python 支持常用压缩格式（.tar、.tgz 和 .zip）文件的压缩和解压缩功能。

使用 shutil 模块的 make\_archive 和 unpack\_archive 等函数，可以实现文件的压缩和解压缩功能。shutil 模块实现高级别的操作，依赖于 zipfile 和 tarfile 模块。

### 19.4.1 shutil 模块支持的压缩和解压缩格式

shutil 模块的 get\_archive\_formats 和 get\_unpack\_formats 函数返回支持的压缩和解压缩格式：

```
>>> import shutil
>>> shutil.get_archive_formats()
[('bztar', 'bzip2 'ed tar - file'), ('gztar', 'gzip 'ed tar - file'), ('tar', 'uncompressed tar file'), ('zip', 'ZIP file')]
>>> shutil.get_unpack_formats()
[('bztar', ['.bz2'], 'bzip2 'ed tar - file'), ('gztar', ['.tar.gz', '.tgz'], 'gzip 'ed tar - file'), ('tar', ['.tar'], 'uncompressed tar file'), ('zip', ['.zip'], 'ZIP file')]
```

额外的压缩和解压缩格式可以使用 shutil 模块的注册和取消注册功能：

```
shutil.register_archive_format(name, function, extra_args = None, description = '')
shutil.unregister_archive_format(name)
shutil.register_unpack_format(name, extensions, function, extra_args = None, description = '')
shutil.unregister_unpack_format(name)
```

### 19.4.2 make\_archive() 和文件压缩

make\_archive() 函数的基本格式如下：

```
make_archive(base_name, format, root_dir = None, base_dir = None, verbose = 0, dry_run = 0, owner = None, group = None, logger = None)
```

其中，base\_name 是目标文件的路径，不包括文件后缀；format 是文件格式，'zip'、'tar'、'bztar' 或 'gztar'；root\_dir 是压缩文件的根目录，默认为当前目录；base\_dir 是压缩的起始目录，默认为当前目录。例如：

```
>>> shutil.make_archive('py33', 'zip', root_dir = r'c:\python33', base_dir = r'c:\python')
'c:\python\py33.zip'
```

### 19.4.3 unpack\_archive() 函数和文件解压缩

unpack\_archive() 函数的基本格式如下：

```
unpack_archive(filename, extract_dir=None, format=None)
```

其中, file\_name 是压缩文件的名称; extract\_dir 是解压缩到的目录, 默认为当前目录; format 是压缩文件的格式, 如果没有指定, 则使用 file\_name 的扩展名。例如:

```
>>> import shutil, os
>>> shutil.unpack_archive(r'c:\python\py33.zip', extract_dir=r'c:\python\temp')
>>> os.listdir(r'c:\python\temp') #['python']
```

## 19.5 configparser 模块和配置文件

configparser 模块用于读取和写入配置文件。

### 19.5.1 INI 文件及 INI 文件格式

INI 文件即 Initialization File (初始化文件), 也称为配置文件, 其后缀一般为 .ini 或 .cfg。INI 文件是文本格式的文件, 通常位于应用程序的配置文件文件夹中, 用于保存应用程序的各种配置信息。

应用程序启动时, 会根据 INI 中的参数来重新初始化应用程序的配置; 系统关闭之前, 会将应用程序当前所需的全部配置都保存到 INI 文件中。

INI 文件的内容由节 (Section)、键 (Option) 和值 (Value) 组成。键和值对以 = 或 : 关联。注解以 # 号或 ; 号开始, 直到该行结尾均为注解。其基本格式如下:

```
;注解行
[Section1 Name]
Option11 = Value11
Option 12 = Value12
...
#注解内容
[Section2 Name]
Option 21 : Value21
Option22 : Value22
...
```

例如, 示例 config.ini 的内容为:

```
;config.ini file
[SystemInfo]
port = 8080
[GameInfo]
level = 1
scores = 0
```

### 19.5.2 ConfigParser 对象和 INI 文件操作

configparser 模块的 ConfigParser 函数用于读取和写入 INI 文件:

```
configparser.ConfigParser (defaults = None, dict_type = collections.OrderedDict, allow_no_value =
```



```
False, delimiters = ('=', ':'), comment_prefixes = ('#', ';'), inline_comment_prefixes = None, strict = True, empty_lines_in_values = True, default_section = configparser.DEFAULTSECT, interpolation = BasicInterpolation())
```

创建 ConfigParser 对象参数众多, 对象方法 defaults() 返回其默认值。

ConfigParser 对象的主要方法如下。

get(section, option, \*, raw = False, vars = None[, fallback]): 返回指定键的值。

getint(section, option, \*, raw = False, vars = None[, fallback]): 返回指定键的值(整型)。

getfloat(section, option, \*, raw = False, vars = None[, fallback]): 返回指定键的值(浮点数)。

getboolean(section, option, \*, raw = False, vars = None[, fallback]): 返回指定键的值(布尔值)。

sections(): 返回所有 section 的列表, 不包括 default section。

items(section, raw = False, vars = None): 返回所有项目的列表。

options(section): 返回所有键的列表。

has\_section(section): 判断是否存在 section。

add\_section(section): 添加 section, 若已经存在, 则导致 DuplicateSectionError。

remove\_section(section): 删除 section。

set(section, option, value): 设置键的值。若 section 不存在, 则导致 NoSectionError。

remove\_option(section, option): 删除键。

read(filename, encoding = None): 从指定文件名读取并解析 INI 配置。

read\_file(f, source = None): 从指定文件对象 f 读取并解析 INI 配置。

read\_string(string, source = '<string>'): 从指定字符串读取并解析 INI 配置。

read\_dict(dictionary, source = '<dict>'): 从指定字典读取并解析 INI 配置。

write(fileobject, space\_around\_delimiters = True): 写入到文件对象。

**【例 19-3】** 读取和写入 INI 文件示例 (configparser.py)。

```
import configparser

def ini_create(): #创建 INI 文件
 config = configparser.ConfigParser()
 config['SystemInfo'] = {'port': '8080'}
 config['GameInfo'] = {'level': 1, 'scores': 0}
 with open('example.ini', 'w') as configfile:
 config.write(configfile)

def ini_read_write(): #读取和设置 INI 文件
 config = configparser.ConfigParser()
 config.read('example.ini')
 config['SystemInfo']['port'] = '8088'
 config.set('GameInfo', 'scores', '1000')
 for item in config.items('GameInfo'): print(item)
 with open('example.ini', 'w') as configfile:
```

```
 config.write(configfile)
if __name__ == '__main__':
 ini_create() #创建 INI 文件
 ini_read_write() #读取和设置 INI 文件
```

运行结果如下：

```
('scores','1000')
('level','1')
```

## 19.6 复习题

### 一、填空题

1. 使用 os 模块中的\_\_\_\_\_函数，可以创建目录。
2. 使用\_\_\_\_\_模块中的相关函数，可以创建临时目录和文件。
3. 使用 os 模块中的\_\_\_\_\_函数，可以切换当前工作目录。
4. 使用 os 模块中的\_\_\_\_\_函数，可以显示一个目录中的文件/子目录列表。
5. 使用 glob 模块中的\_\_\_\_\_函数，可以获取满足指定模式的文件/目录列表。
6. 使用 os 模块的\_\_\_\_\_函数，可以遍历指定的目录结构。
7. 使用 os 模块的\_\_\_\_\_函数，可以将目录名和文件名连接成全限定路径。
8. 使用 os.path 模块的\_\_\_\_\_函数，可以判断文件/目录是否存在。
9. 使用 os.path 模块的\_\_\_\_\_函数，可以判断路径 path 是否为文件类型。
10. 使用 os.path 模块的\_\_\_\_\_函数，可以判断路径 path 是否为目录类型。
11. 使用 os.path 模块的\_\_\_\_\_函数，可以判断路径 path 是否为绝对路径。
12. 使用 os.path 模块的\_\_\_\_\_函数，可以判断路径 path 是否为链接类型。
13. 使用 os.path 模块的\_\_\_\_\_函数，可以获取指定文件和目录的上次访问时间。
14. 使用 os.path 模块的\_\_\_\_\_函数，可以获取指定文件和目录的上次修改时间。
15. 使用 os.path 模块的\_\_\_\_\_函数，可以获取指定文件和目录的创建时间。
16. 使用 os.path 模块的\_\_\_\_\_函数，可以获取指定路径 path 的大小。
17. 使用 os 模块中的\_\_\_\_\_函数，可以删除指定文件。
18. 使用 os 模块中的\_\_\_\_\_函数，可以删除指定目录。
19. 使用 shutil 模块中的\_\_\_\_\_函数，可以删除指定目录及目录下的所有内容。
20. 使用 shutil 模块中的\_\_\_\_\_函数，可以复制目录树。
21. 使用 shutil 模块中的\_\_\_\_\_函数，可以移动文件/目录。
22. 使用 shutil 模块中的\_\_\_\_\_函数，可以复制文件/目录，并且除了复制文件内容外，还复制文件许可权限。
23. 使用 shutil 模块中的\_\_\_\_\_函数，可以复制所有元数据，包括创建时间和修改时间。
24. 使用 shutil 模块中的\_\_\_\_\_函数，可以获取磁盘空间的使用情况。
25. 使用 os 模块中的\_\_\_\_\_函数，可以在 Python 程序中执行操作系统的命令和脚本，或运行其他程序。

26. 使用 os 模块中的\_\_\_\_\_函数, 可以在 Python 程序中执行操作系统的命令和脚本。
27. 通过 os 或 shutil 模块的\_\_\_\_\_函数, 可以获取终端的大小, 以方便输出内容的格式化操作。
28. 使用 shutil 模块的\_\_\_\_\_和\_\_\_\_\_等函数, 可以实现文件的压缩和解压缩功能。
29. shutil 模块的\_\_\_\_\_和\_\_\_\_\_函数返回支持的压缩和解压缩格式。
30. INI 文件即 Initialization File (初始化文件), 也称之为配置文件, 其后缀一般为\_\_\_\_\_或\_\_\_\_\_。
31. INI 文件的内容由\_\_\_\_\_, \_\_\_\_\_和\_\_\_\_\_组成。键和值对以\_\_\_\_\_或\_\_\_\_\_关联。注解以\_\_\_\_\_或\_\_\_\_\_开始, 直到该行结尾均为注解。
32. configparser 模块的\_\_\_\_\_函数用于读取和写入 INI 文件。

## 二、思考题

1. Python 标准库中包括哪些系统管理相关模块?
2. 如何用 Python 删除、复制、重命名文件和目录?
3. Python 如何实现文件的压缩和解压缩功能?
4. Python 如何实现配置文件的读取和写入功能?

## 19.7 上机实践

1. 参照例 19-1, 编写输出指定目录的目录结构的程序。
2. 参照例 19-2, 编写获取终端大小示例程序。
3. 参照例 19-3, 编写读取和写入 INI 文件示例程序。



# 附录 A 参 考 答 案

## 第 1 章 Python 语言概述

### 一、单选题

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| C | B | A | D |

### 二、填空题

1. 对象     2. 可移植性     3. `http://Python.org/`     4. `quit()`, `Ctrl + Z`     5. `F5`

## 第 2 章 Python 语言基础

### 一、单选题

|   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| A | A | B | A | D | B | C | C | A | B  | A  | C  |

### 二、填空题

1. 简单语句     2. 缩进对齐     3. `\`     4. `;`     5. `#`     6. `pass`     7. `help()`  
8. `keywords`     9. `2 * 32 - 1`     10. `dir(__builtins__)`     11. `1 - 2 - 3!`  
12. `1.0`     13. 整数类型 `int`、字符串 `str`、`complex`、元组 `tuple`、字节序列 `bytes`。列表 `list`、字典 `dict`、集合 `set`、字节数组 `bytearray`     14. `dict`     15. `is` 和 `is not`、`type()`、`==`  
16. `4 3`

### 三、思考题

12. `20 8 50 2.0 2 1`  
13. `c.isalpha()` 或者 `c.lower() <= 'z' and c.lower() >= 'a'`,  
      `c.upper() <= 'Z' and c.upper() >= 'A'`, `c <= 'Z' and c >= 'A' or c <= 'z'`  
      and `c >= 'a'`  
14. `c.isdigit()` 或者 `c <= '9' and c >= '0'`  
15. `c.isupper()` 或者 `c <= 'Z' and c >= 'A'`  
16. `c.islower()` 或者 `c <= 'z' and c >= 'a'`  
17. `9 8 7 6`

18. <type 'NoneType' >
19. True True False True True。说明：

x = y = [1,2]

#变量 x 和 y 指向 list 对象[1,2]

x.append(3)

#变量 x 指向的 list 对象[1,2]附加一个元素

x is y

#输出:True。表示变量 x 和 y 指向同一个 list 对象[1,2,3]

x == y

#输出:True。表示变量 x 和 y 指向的 list 对象值相等

z = [1,2,3]

#变量 z 指向的 list 对象[1,2,3]

x is z

#输出:False。表示变量 x 和 z 指向不同的 list 对象[1,2,3]

x == z

#输出:True。表示变量 x 和 z 指向的 list 对象值相等

y == z

#输出:True。表示变量 x 和 z 指向的 list 对象值相等

20. 数量 100，单价 285.6；数量 100，单价 285.60；数量 100，单价 285.600

21. 格式化输出数字三角形（右对齐）。

1

121

12321
- 第 3 章 程序流程控制
- 一、单选题
- |   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| A | A | B | A | B | D | C | B | A | C  |
- 二、填空题
1. True 2. False 3. x > 0 and y > 0 or x < 0 and y > 0

4. i % 3 == 0 and i % 5 == 0

5. 整数类型（int），布尔类型（bool）、浮点类型（float）、复数类型（complex）

6. 元组（tuple）、列表（list）、字符串（str）和字节数据（bytes 和 bytearray）

7. break 8. True 9. True 10. True False

11. 6 12. 25 或者 26 13. 1、1
- 三、思考题
3. (1) if (i > 0):

if (j > 0): n = 1

else: n = 2

相当于： $\begin{cases} i > 0, j > 0 & n = 1 \\ i > 0, j \leq 0 & n = 2 \end{cases}$ ，流程图参见图 3-1（a）所示。

(2) if (i > 0):

if (j > 0): n = 1

else: n = 2

相当于:  $\begin{cases} i > 0, j > 0 & n = 1 \\ i \leq 0 & n = 2 \end{cases}$ , 流程图参见图 3-1 (b) 所示。

(3) if (i > 0): n = 1  
else:  
    if (j > 0): n = 2

相当于:  $\begin{cases} i > 0 & n = 1 \\ i \leq 0, j > 0 & n = 2 \end{cases}$ , 流程图参见图 3-1 (c) 所示。

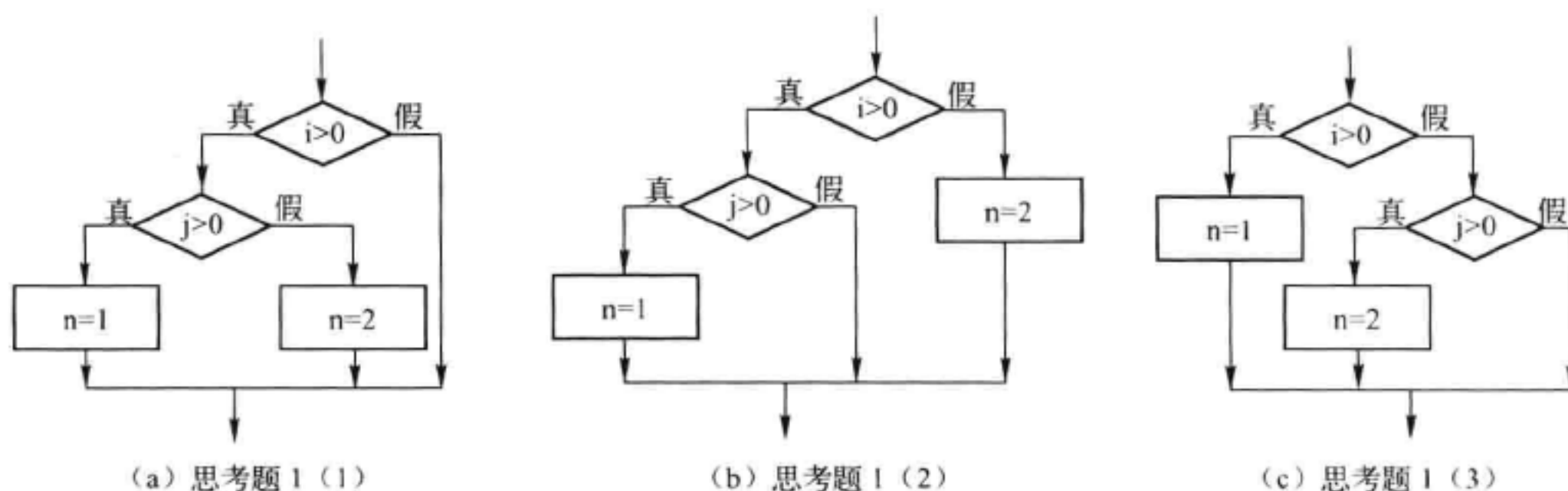


图 3-1 思考题 1 流程图

4. no                      5. 2                      6. 0 1 2 2 3 4

7. 打印 100 ~ 200 间的全部素数, 每行输出 10 个。

8. 利用循环语句显示有规律的图形 (用 \* 构成的上三角), 要求输入显示的行数。运行效果如图 3-4 所示。

9. 本题显示三位数中的所有水仙花数, 运行效果如图 4-5 所示。所谓“水仙花数”是指一个三位数, 其各位数字立方和等于该数字本身。例如, 153 是水仙花数, 因为  $153 = 1^3 + 5^3 + 3^3$ 。

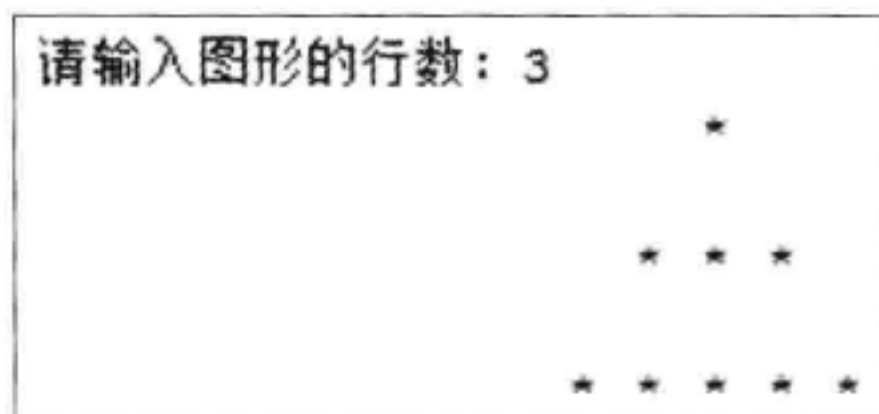


图 3-4 用 \* 构成的上三角运行效果

三位数中的所有水仙花数为:  
153 370 371 407

图 3-5 水仙花数运行效果

10. 本题找出 1000 以内的所有完数, 运行效果如图 3-6 所示。一个数如果恰好等于它的因子之和, 这个数就称为“完数”。例如, 6 的因子为 1、2、3, 而  $6 = 1 + 2 + 3$ , 因此 6 就是“完数”。

1~1000之间所有的完数有, 其因子为:  
6: [1, 2, 3]  
28: [1, 2, 4, 7, 14]  
496: [1, 2, 4, 8, 16, 31, 62, 124, 248]

图 3-6 1000 以内的所有完数运行效果



11. 求任意两个整数的最大公约数。

12. T T F T F T F T F T。说明：如果 Python 表达式的结果为数值类型 (0)、空字符串 ("")、空元组(())、空列表([])、空字典({})，则其 bool 值为 False (假)；否则其 bool 值为 True (真)。例如：123、"abc"、(1, 2) 均为 True。

13. 1 2 True True 2、2 0 False 2 False。说明：(1)  $C = A \text{ or } B$ 。如果 A 不为 0 或不为空或为 True，则返回 A；否则返回 B。仅在必要时才计算第二个操作数，即如果 A 不为 0 或不为空或为 True，则不用计算 B。(2)  $C = A \text{ and } B$ 。如果 A 为 0 或为空或为 False，则返回 A；否则返回 B。仅在必要时才计算第二个操作数，即如果 A 为 0 或为空或为 False，则不用计算 B。

## 第 4 章 数值类型

### 一、单选题

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| A | C | A | B | D | A | D | C |

### 二、填空题

2. 25
2. 0
- 0.5
4. 0 15
- 32 5
- 0x10 0b1010
- (5, 2)
- (3, 2)
- 4 (2, 1)
- (-5 + 12j)
- 6 16
3. 2 2. 23606797749979
- random
- $\text{math. sin}(15 * \text{math. pi}/180) + (\text{math. pow}(\text{math. e}, x) - 5 * x) / \text{math. sqrt}(x * x + 1) - \text{math. log}(3 * x)$
- $((c * d) / 2 / (c + d) - 4 * \text{math. pi} / (c - d)) / (a + b)$

### 三、思考题

1. 78、1. 8、1. 7、1. 7、-1. 5、-1. 5、1. 7、1. 6、-1. 5、1. 6、-1. 4
- 随机产生一个 3 位正整数，然后逆序输出。

## 第 5 章 系列：元组、列表和字符串

### 一、单选题

|   |   |   |   |   |   |   |   |   |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| D | A | C | C | C | D | B | C | C | D  | D  | A  | D  | C  |

### 二、填空题

- 字符串 (str)、列表 (list)、元组 (tuple) 和字节系列 (bytes 和 bytearray)
- RED HAT、'RED HAT'、'Red Hat'、'red cat'
- r
- 0
- False
- 45
- 1%1
- B
- helloworld
- helloworld
- 5
- 7
- (0, 1) [0, 1]

```
14. (1, 2, 3) [1, 2, 3] 15. [1, 3] [0, 1, 4] 16. [1, 2, 3, 4] [3, 4, 5]
[0, 1, 2] 90.0
17. 'c'; ('c', 'd'); ('a', 'b', 'c'); ('d', 'e'); ('b', 'd'); 'd'; ('e', 'd',
 'c', 'b', 'a'); ('d',); ('d', 'e'); (); ('a', 'b'); ('a', 'b', 'c', 'd', 'e');
 ('b', 'c', 'd')
18. 6、1、[4, 'x', 'y'] 19. [[1,2], 7, 'a']
20. '0000abc', 'abc', 'abc', '0000abc'
21. ['a','b','c'], ['a,b','c'], ('a',,,'b,c'), ('a,b',,,'c'), 'a:b:c', 'x:y:z'
```

三、思考题

- 2. 利用循环语句显示有规律的图形，要求输入上（或下）三角的行数。运行效果如图 5-1 所示。
- 3. 利用循环语句显示有规律的图形，要求输入上（或下）三角的行数，运行效果如图 5-2 所示。



图 5-1 沙漏型三角形运行效果

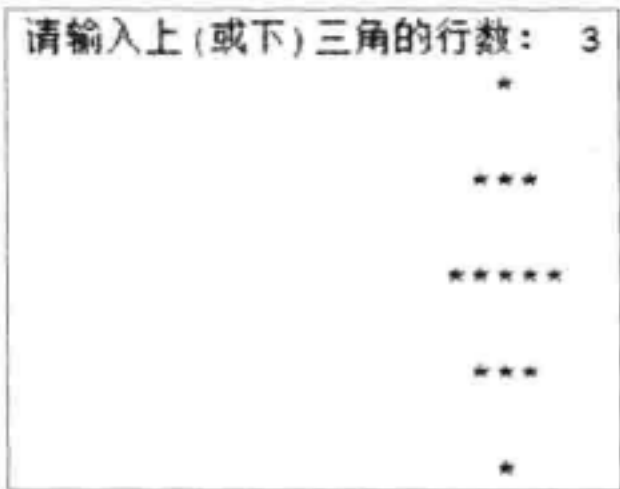


图 5-2 菱形三角形运行效果

- 4. 先输出星期，然后输出月份。即：DAYS: ['Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday'], MONTHS ['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']
- 5. 分行输出 fruits 列表中各元素的值，等价于输出 “pear\napple\nkiwi\navocado\norange\n”。
- 6. birth、happy Birthday
- 7. 4。说明：语句 names2 = names1 使得 names1 和 names2 指向（引用）相同的对象实例，而语句 names3 = names1[:] 表示 names3 是复制（创建）的新实例，其内容与 names1（也即 names2）的内容相同。

第 6 章 字典和集合类型

一、单选题

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| B | D | A | C | C |

## 二、填空题

1. 0    2. 2    3. {1, 2, 3}    4. 15    5. 食品  
 6. g    7. {1,2,3,5}, {2,3}, {1}, {1,5}, {1,2,3,'x'}, {2,3}, {1,2}, set()

## 三、思考题

1. 6    2. {1: 'x', 3: 'c'}    3. {'Pear': 1, 'Apple': 2, 'kiwi': 1, 'apple': 2}  
 4. 3    6    {(1,2): 3, (2,1): 2, (1,2,3): 1}  
 5. 12。语句 d2 = d1 使得 d1 和 d2 指向 (引用) 相同的对象实例。  
 6. 7。本例中的 d2 = dict(d1) 相当于 d2 = d1.copy()。

## 第 7 章 文件和流 IO

## 一、填空题

1. open    2. close、with    3. seek    4. fileinput  
 5. pickle/cPickle/marshal

## 第 8 章 函数和函数编程

## 一、单选题

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| D | D | C | C |

## 二、填空题

1. 5.5    2. global    3. 全局变量、局部变量和类型成员变量  
 4. getrecursionlimit 和 setrecursionlimit

## 三、思考题

4. 24    5. 1 4 9    6. simple function    7. 4 0    8. 求两个数的最大公约数  
 9. quick  
 10. <class 'tuple'>、(2, 3, 4, 5)。在声明函数时, 通过带星的参数, 如 \* param2, 允许向函数传递可变数量的实参。调用函数时, 从那一点后所有的参数被收集为一个元组。  
 11. <class 'dict'>、{'d': 5, 'a': 2, 'c': 4, 'b': 3}。在声明函数时, 通过带双星的参数, 如 \*\* param2, 允许向函数传递可变数量的实参。调用函数时, 从那一点后所有的参数被收集为一个字典。带星或双星的参数必须位于形参列表的最后位置。

## 第 9 章 类和对象

## 一、填空题

1. 封装、继承和多态    2. False    3. .    4. \_\_new\_\_, \_\_init\_\_, \_\_del\_\_



## 二、思考题

3. 100 100      4. 100      5. 400      6. 12      7. 12  
8. 7      9. 30      10. 16

11. 21。“object.\_\_dict\_\_”返回对象的属性字典,本例为{'id': 123, 'age': 18, 'gender': 'female'}。

# 第 10 章 模块和包

## 一、填空题

1. import      2. from m import \*      3. \_\_import\_\_()      4. sys.path  
5. \_\_name\_\_, \_\_main\_\_      6. sys.argv, argv[0], argv[1], argv[2]  
7. argparse      8. dir(), help()

# 第 11 章 迭代器和生成器

## 一、填空题

1. iter()、\_\_iter\_\_()      2. \_\_iter\_\_()和\_\_next\_\_()      3. reversed()  
4. 0 9 36 81  
5. 4, 3, 2, 1,      6. [0,5], [1,2,3]      7. [4,6]      8. [64,27,100]      9. [1,6]  
10. ['a',5,3.2,{1,2}]      11. [0,4,8]      12. [(4,'x',0),(5,'y',1)]  
13. [('A','a'),('B','b'),('\*','c')]      14. [(101,'a'),(102,'b')]      15. [('a',1),  
( 'b',2),('c',3)]  
16. [('a',0),('b',1),('a',2),('b',3),('a',4)]      17. ['Tiger','Tiger','Tiger']  
18. [1,3,6]      19. [1,2,6]      20. [1,2,'x','y',0,1,2]      21. ['a','b','1','2','3']  
22. ['x','z']      23. [6,3,7]      24. [1,2]      25. [0,1,2] [0,1,2]

## 二、思考题

4. 4、9、1、16      5. 1 4 9 4      6. 1 2 1 2 3      7. K  
8. 4 6 8      9. ['a','b'], ['c','d'], ['c','d','e'], ['a','c','e']  
10. 0 [0,0,0], 1 [-1,1,-1,1], 2 [-2,2,2]  
11. [('a','b'),('a','c'),('a','d'),('b','c'),('b','d'),('c','d')], [('a','b','c'),  
( 'a','b','d'),('a','c','d'),('b','c','d')], [('a','a'),('a','b'),('a','c'),('b','b'),('b','c'),  
( 'c','c')]  
12. [ ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')], [ ('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C', 'A'), ('C', 'A', 'B'), ('C', 'B', 'A')]  
13. [('a',1),('a',2),('a',3),('b',1),('b',2),('b',3)], [(1,1,1),(1,1,2),(1,2,1),(1,2,2),(2,1,1),(2,1,2),(2,2,1),(2,2,2)]

## 第 12 章 数据结构和算法

### 一、思考题

1. `[{1: 'a', 2: 'b'}, {2: 'a', 3: 'x', 4: 'y'}]` `ChainMap({2: 'a', 3: 'x', 4: 'y'})` `ChainMap({}, {1: 'a', 2: 'b'}, {2: 'a', 3: 'x', 4: 'y'})`, `a x`, `ChainMap({1: 'A', 2: 'b', 3: 'X'}, {2: 'a', 3: 'x', 4: 'y'})`
2. `Counter()`, `Counter({'a': 3, 'n': 2, 'b': 1})`, `Counter({'R': 4, 'B': 2})`, `Counter({'dogs': 8, 'cats': 4, 'birds': 2})`, `0 4`, `['R', 'R', 'R', 'R', 'B', 'B']`, `[('dogs', 8), ('cats', 4)]`, `Counter({'R': 3, 'B': 1, 'G': -1})`
3. `a 2 c c 2 a`    4. `a 2 c`    5. `1 0`, `[('r', [3, 5]), ('b', [1, 4]), ('g', [2])]`
6. `dict_items([('red', 3), ('green', 4), ('blue', 1)])` `[('blue', 1), ('green', 4), ('red', 3)]`, `('red', 3)` `('blue', 1)`
7. `('x', 'y')` `1 2`, `OrderedDict([('x', 10), ('y', 20)])`, `Point(x=100, y=20)` `1 2`
8. `array('i', [1, 22, 3, 4, 5])` `array('i', [3, 4, 5])` `<class 'int'>`, `array('i', [1, 22])` `i 4`
9. `array('b', [3, 2, 3, 3, 5])` `3`, `array('b', [3, 2, 3, 3, 5, 65, 49, 8, 9])` `0`, `[8, 49, 65, 5, 3, 3, 3, 1]`
10. `1 2 3 4`    11. `1 2 4 5 6 8 9`, `[9, 8]`, `[0, 1]`    12. `1`, `[2, 3, 4, 6]`, `2`, `[2, 3, 3, 4, 6]`

## 第 13 章 日期和时间处理

### 一、填空题

1. `datetime`, `calendar`, `time`    2. `date`, `time`, `datetime`, `timedelta`, `tzinfo`, `timezone`
3. `getime`    4. `daylight`    5. `strptime()`, `strftime()`
6. `datetime.MINYEAR` 和 `datetime.MAXYEAR`, `1` 和 `9999`
7. `strftime()`, `strptime()`    8. `td.days`, `td.seconds`, `td.microseconds`
9. `True`

### 二、思考题

1. `0001-01-01 9999-12-31 0001-02-01`, `2014 10 1`, `735507 2 Wed Oct 1 00:00:00 2014 2014/10/01 (Wed)`
2. `00:00:00 23:59:59.999999`, `19 30 45 196`, `19 时 30 分 45 秒`
3. `0001-01-01 00:00:00 9999-12-31 23:59:59.999999`, `2014 5 1 9 35 46`, `2014-05-01 09:35:46 2014/05/01 (Thursday)`, `09 时 35 分 46 秒`
4. `0:25:00 300 1:40:00` `True`
5. `31 2014-06-11 2014-05-22` `True`

## 第 14 章 正则表达式

### 一、填空题

1. 空
2. ['to', 'to']
3. ['boy', 'box']
4. \d{6}
5. /i、/m
6. 'Python is easy to learn.'
7. ['go', 'went', 'gone']
8. ['a', 'b', 'c', '']

### 二、思考题

5. 8。注意，match 函数从字符串头部开始匹配，而 search 函数在字符串任何位置匹配。
6. None、<\_sre.SRE\_Match object at 0x02DDF640>、['to', 'to']
7. to (10, 12)

## 第 15 章 多线程编程

### 一、填空题

1. start\_new\_thread, \_thread.exit()
2. run, start
3. 用户线程和 daemon 线程
4. daemon 线程
5. locked 和 unlocked (初始状态)

## 第 16 章 图形用户界面应用程序

### 一、填空题

1. \_tkinter, tkinter 和 tkinter.constants
2. 根窗口或主窗口
3. pack, grid 和 place
4. width 和 height
5. font
6. anchor
7. cursor
8. text, wraplength, justify
9. bitmap, . xbm
10. image
11. compound
12. relief, overrelief
13. borderwidth 或 bd
14. padx 和 pady
15. state
16. underline
17. textvariable
18. Radiobutton (单选按钮)
19. Checkbutton (复选框)
20. Listbox (列表框)
21. OptionMenu (选择项)
22. Scale (移动滑块)
23. messagebox, filedialog, colorchooser 和 simpledialog
24. messagebox
25. filedialog
26. colorchooser
27. simpledialog
28. Canvas (画布)

## 第 18 章 网络编程和通信

### 一、填空题

1. 网络接口层、Internet 层、传输层和应用层
2. IP 地址
3. 域名系统 (Domain Name System, DNS)



4. 统一资源定位器 (Uniform Resource Locator, URL)  
5. TCP、UDP                      6. listen 和 accept                      7. connect

## 第 19 章 系 统 管 理

### 一、填空题

- |                                   |                                                   |                       |
|-----------------------------------|---------------------------------------------------|-----------------------|
| 1. mkdir                          | 2. tempfile                                       | 3. chdir              |
| 4. listdir                        | 5. glob                                           | 6. walk               |
| 7. join                           | 8. exists                                         | 9. isfile             |
| 10. isdir                         | 11. isabs                                         | 12. islink            |
| 13. getatime                      | 14. getmtime                                      | 15. getctime          |
| 16. getsize                       | 17. remove                                        | 18. rmdir             |
| 19. rmtree                        | 20. copytree                                      | 21. move              |
| 22. copy                          | 23. copy2                                         | 24. disk_usage        |
| 25. system                        | 26. popen                                         | 27. get_terminal_size |
| 28. make_archive 和 unpack_archive | 29. get_archive_formats 和 get_unpack_formats      |                       |
| 30. .ini 或 .cfg                   | 31. 节 (Section)、键 (Option) 和值 (Value)、= 或:、#号或; 号 |                       |
| 32. ConfigParser                  |                                                   |                       |

## 参 考 文 献

- [1] Python Software Foundation. Python v3.3.2 documentation. <http://Python.org/>.
- [2] BEAZLEY D, JONES B K. PythonCookbook. 3ed. O'REILLY, 2013.
- [3] 江红, 余青松. C#程序设计教程. 2 版. 北京: 清华大学出版社, 2014.
- [4] 余青松, 江红. C#程序设计实验指导与习题测试. 2 版. 北京: 清华大学出版社, 2014.
- [5] CHU W J. Core Python Applications Programming. 3rd ed. Prentice Hall, 2012.
- [6] HELLMANN D. ThePython StandardLibrary byExample. Addison – Wesley, 2011.
- [7] <http://Python.org/>.