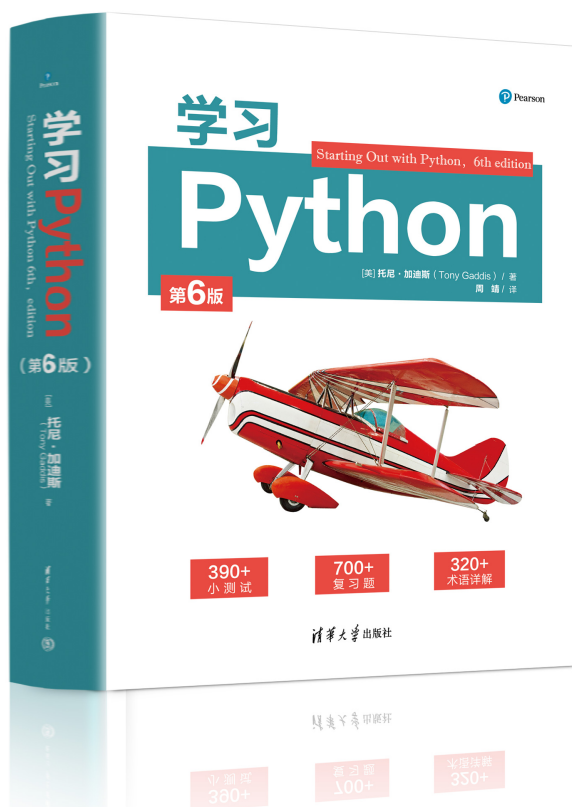


《学习 Python》第 6 版

(美) Tony Gaddis 著

周靖译 (<https://bookzhou.com>)



中文试读版 1-6 章和附录，翻译原稿，仅供参考，

更多精彩内容，请购买正版：[京东>>](#) [淘宝>>](#)

配套资源和试读下载：[ys168 网盘>>](#) [百度网盘>>](#)

[访问中文版博客，获取最新内容](#)

清华大学出版社

北京

学习 Python 中文版

(美) Tony Gaddis 著

本书简介：

本书深入浅出讨论了各种 Python 编程主题。利用从本书学到 Python 知识，你会对自己的编程技能充满信心，并掌握开发高质量程序背后的逻辑。全球知名教育作家 Tony Gaddis 采用一种易懂的、循序渐进的方法来介绍基本的编程概念。先从控制结构、函数和列表等概念开始，再深入讨论类。这有助于确保读者理解基本的编程概念，并知道如何解决现实中的问题。每一章都包括清晰美观的代码清单、真实世界的例子和大量练习。

第 6 版针对 Python 3.11 进行了全面更新，新增或改进了大量主题，包括：`with` 语句（第 6 章）、多重赋值（第 2 章）、单行 `if` 语句（第 3 章）、条件表达式（第 3 章）、海象操作符和赋值表达式（第 3 章和第 4 章）、`while` 循环作为计数控制循环（第 4 章）、单行 `while` 循环（第 4 章）、为循环使用 `break/continue/else`（第 4 章）、仅关键字参数（第 5 章）、仅位置参数（第 5 章）、默认实参（第 5 章）、为列表使用 `count` 和 `sum`（第 7 章）、在元组中存储可变对象（第 7 章）以及字典合并和更新操作符（第 9 章）。

作者简介


Tony Gaddis 从事计算机教学工作二十多年，具有丰富的教学经验，曾因教学上取得的巨大成就而获得美国 NISOD 优秀教师奖和北卡社区大学年度优秀教师奖。他是畅销教材系列“编程基础”(Starting Out with)的主创，该系列覆盖 C++、Java、C#、Python 和 Alice 等主流编程语言。

译者简介

周靖，微软最有价值专家 (MVP)，具有深厚的程序开发功底和良好的文学素养，其翻译风格严谨、准确、朴实、流畅，深受读者欢迎。他的代表译著有《Visual C#从入门到精通》系列、《C#本质论》系列、《CLR via C#》系列、《Windows 核心编程》、Walter Savitch 教授的《C++程序设计》系列和 Deitel 父子的《学习 C++ 20 中文版》等。

目录



译者序	29
前言	31
<i>首先讲控制结构，然后讲类.....</i>	<i>31</i>
<i>第6版的变化.....</i>	<i>31</i>
<i>各章内容概览.....</i>	<i>32</i>
第1章 计算机和编程概述.....	32
第2章：输入、处理和输出.....	32
第3章：判断结构和布尔逻辑.....	32
第4章：循环结构.....	32
第5章：函数	33
第6章：文件和异常.....	33
第7章：列表和元组.....	33
第8章：深入字符串.....	33
第9章：字典和集合.....	33
第10章：类和面向对象编程.....	34
第11章：继承	34
第12章：递归	34
第13章：GUI 编程	34
第14章：数据库编程.....	34
附录 A：安装 Python.....	34
附录 B：IDLE 简介	34
附录 C：ASCII 字符集	35
附录 D：预定义颜色名称	35

附录 E: import 语句.....	35
附录 F: 用 format()函数格式化输出.....	35
附录 G: 用 pip 工具安装模块.....	35
附录 H: 知识检查点答案.....	35
本书的组织方式.....	35
本书特色.....	36
补充材料.....	37
学生在线资源.....	37
中文版资源.....	37
教师资源.....	37
致谢.....	38
作者简介.....	40
第 1 章 计算机和编程概述.....	42
1.1 概述.....	42
1.2 硬件和软件.....	43
1.2.1 硬件.....	43
1.2.2 CPU.....	44
1.2.3 主存.....	45
1.2.4 辅助存储设备.....	46
1.2.5 输入设备.....	46
1.2.6 输出设备.....	47
1.2.7 软件.....	47
 检查点.....	48
1.3 计算机如何存储数据.....	48
1.3.1 存储数字.....	49

1.3.2 存储字符.....	52
1.3.3 高级数字存储.....	52
1.3.4 其他类型的数据.....	53
 检查点.....	53
1.4 程序如何工作.....	54
1.4.1 从机器语言到汇编语言.....	56
1.4.2 高级语言.....	57
1.4.3 关键字、操作符和语法：概述.....	58
1.4.4 编译器和解释器.....	59
 检查点.....	61
1.5 使用 Python	61
1.5.1 安装 Python	61
1.5.2 Python 解释器	62
1.5.3 IDLE 编程环境.....	64
复习题.....	65
第 2 章 输入、处理和输出	71
2.1 设计程序.....	71
2.1.1 程序开发周期.....	71
2.1.2 关于设计过程的更多说明.....	72
2.1.3 理解程序要执行的任务.....	72
2.1.4 确定执行任务必须采取的步骤.....	73
2.1.5 伪代码.....	73
2.1.6 流程图.....	74
 检查点.....	75

2.2 输入、处理和输出.....	75
2.3 用 <code>print</code> 函数显示输出.....	76
2.3.1 字符串和字符串字面值.....	77
 检查点.....	79
2.4 注释.....	79
2.5 变量.....	80
2.5.1 用赋值语句创建变量.....	80
2.5.2 多重赋值.....	83
2.5.3 变量命名规则.....	83
2.5.4 用 <code>print</code> 函数显示多项内容.....	85
2.5.5 变量重新赋值.....	85
2.5.6 数值数据类型和字面值.....	86
2.5.7 用 <code>str</code> 数据类型存储字符串.....	88
2.5.8 让变量引用不同数据类型.....	88
 检查点.....	89
2.6 从键盘读取输入.....	90
2.6.1 用 <code>input</code> 函数读取数字.....	91
 检查点.....	94
2.7 执行计算.....	94
2.7.1 浮点和整数除法.....	97
2.7.2 操作符优先级.....	97
2.7.3 用圆括号分组.....	99
2.7.4 求幂操作符.....	101
2.7.5 求余操作符.....	101

2.7.6 将数学公式转换为编程语句	102
2.7.7 混合类型的表达式和数据类型转换	104
2.7.8 将长语句拆分为多行	105
 检查点	106
2.8 字符串连接	107
隐式连接字符串字面值	108
 检查点	108
2.9 关于 <code>print</code> 函数的更多知识	108
2.9.1 阻止 <code>print</code> 函数的换行功能	109
2.9.2 指定分隔符	109
2.9.3 转义序列	110
 检查点	111
2.10 用 <code>f</code> 字符串格式化输出	111
2.10.1 占位符表达式	112
2.10.2 格式化值	113
2.10.3 浮点数四舍五入	113
2.10.4 插入逗号分隔符	114
2.10.5 将浮点数格式化为百分比	115
2.10.6 用科学计数法格式化	115
2.10.7 格式化整数	116
2.10.8 指定最小域宽	116
2.10.9 值的对齐	118
2.10.10 指示符的顺序	119
2.10.11 连接 <code>f</code> 字符串	120





 检查点	120
2.11 具名常量.....	121
 检查点	122
2.12 海龟图形概述.....	123
2.12.1 使用海龟图形来画线.....	123
2.12.2 海龟转向.....	124
2.12.3 使海龟朝向指定角度.....	127
2.12.4 获取海龟的当前朝向.....	127
2.12.5 画笔抬起和放下.....	128
2.12.6 画圆和画点.....	128
2.12.7 更改画笔大小.....	130
2.12.8 更改画笔颜色.....	130
2.12.9 更改背景颜色.....	130
2.12.10 重置屏幕.....	130
2.12.11 指定图形窗口的大小.....	131
2.12.12 获取海龟的当前位置.....	131
2.12.13 控制海龟动画的速度.....	131
2.12.14 隐藏海龟.....	132
2.12.15 将海龟移到指定位置.....	132
2.12.16 在图形窗口中显示文本.....	133
2.12.17 填充形状.....	134
2.12.18 从对话框获取输入.....	137
2.12.19 使用 <code>turtle.textinput</code> 命令获取字符串输入.....	139
2.12.20 使用 <code>turtle.done()</code> 使图形窗口保持打开	139

 检查点	147
复习题	148
编程练习	152
第 3 章 判断结构和布尔逻辑	157
3.1 if 语句	157
3.1.1 布尔表达式和关系操作符	161
3.1.2 >=和<=操作符	163
3.1.3 ==操作符	163
3.1.4 !=操作符	164
3.1.5 综合运用	164
3.1.6 单行 if 语句	168
 检查点	168
3.2 if-else 语句	168
if-else 语句的缩进	170
 检查点	172
3.3 比较字符串	172
其他字符串比较	173
 检查点	176
3.4 嵌套判断结构和 if-elif-else 语句	176
3.4.1 测试一系列条件	181
3.4.2 if-elif-else 语句	183
 检查点	185
3.5 逻辑操作符	185
3.5.1 and 操作符	186

3.5.2 or 操作符	187
3.5.3 短路求值.....	187
3.5.4 not 操作符	187
3.5.5 修订贷款资格判断程序.....	188
3.5.6 另一个贷款资格判断程序.....	190
3.5.7 用逻辑操作符检查数字范围.....	191
 检查点	191
3.6 布尔变量.....	192
 检查点	193
3.7 条件表达式.....	193
3.8 赋值表达式和海象操作符.....	195
海象操作符的优先级.....	197
3.9 海龟图形：判断海龟的状态.....	197
3.9.1 判断海龟位置.....	198
3.9.2 判断海龟朝向.....	198
3.9.3 判断笔是否放下.....	198
3.9.4 判断海龟是否可见.....	199
3.9.5 判断当前颜色.....	199
3.9.6 判断画笔大小.....	200
3.9.7 判断海龟的动画速度.....	200
 检查点	205
复习题.....	205
编程练习.....	208
第 4 章 重复结构.....	216


4.1 重复结构简介.....	216
条件控制和计数控制循环.....	217
 检查点.....	217
4.2 while 循环：条件控制循环.....	217
4.2.1 while 循环是预测试循环.....	221
4.2.2 无限循环.....	223
4.2.3 while 循环用作计数控制循环.....	224
4.2.4 单行 while 循环.....	227
 检查点.....	228
4.3 for 循环：计数控制循环.....	229
4.3.1 为 for 循环使用 range 函数.....	233
4.3.2 在循环内使用目标变量.....	234
4.3.3 让用户控制循环迭代.....	239
4.3.4 生成降序可迭代序列.....	241
 检查点.....	241
4.4 计算累加和.....	242
复合赋值操作符.....	244
 检查点.....	246
4.5 哨兵.....	246
 检查点.....	248
4.6 输入校验循环.....	248
在输入校验循环中使用海象操作符.....	253
 检查点.....	253
4.7 嵌套循环.....	254

4.8 为循环使用 <code>break</code> , <code>continue</code> 和 <code>else</code>	261
4.8.1 <code>break</code> 语句	262
4.8.2 <code>continue</code> 语句	263
4.8.3 在循环中使用 <code>else</code> 子句	264
4.9 海龟图形: 用循环来画图.....	266
复习题.....	271
编程练习.....	274
第 5 章 函数	280
5.1 函数简介.....	280
5.1.1 用函数将程序模块化的好处	281
5.1.2 <code>void</code> 函数和返回值的函数	282
 检查点	283
5.2 定义和调用 <code>void</code> 函数.....	283
5.2.1 函数名.....	283
5.2.2 定义和调用函数	283
5.2.3 调用函数.....	284
5.2.4 Python 的缩进	288
 检查点	289
5.3 使用函数来设计程序.....	289
5.3.1 使用了函数的程序的流程图	289
5.3.2 自上而下设计	291
5.3.3 层次结构图.....	291
5.3.4 暂停执行直到用户按 <code>Enter</code> 键	294
5.3.5 使用 <code>pass</code> 关键字.....	295

5.4 局部变量.....	295
作用域和局部变量.....	296
 检查点	298
5.5 向函数传递实参.....	298
5.5.1 形参变量的作用域.....	301
5.5.2 传递多个实参.....	302
5.5.3 修改形参.....	304
5.5.4 关键字参数.....	306
5.5.5 混合使用关键字实参和位置实参	308
5.5.6 仅关键字参数.....	308
5.5.7 仅位置参数.....	309
5.5.8 默认实参.....	310
 检查点	313
5.6 全局变量和全局常量.....	313
全局常量	315
 检查点	317
5.7 返回值的函数：生成随机数.....	317
5.7.1 标准库函数和 import 语句.....	317
5.7.2 生成随机数.....	318
5.7.3 从 f 字符串中调用函数.....	323
5.7.4 在交互模式下尝试使用随机数.....	323
5.7.5 randrange、random 和 uniform 函数.....	327
5.7.6 随机数种子.....	328
 检查点	329




5.8 自定义返回值的函数.....	329
5.8.1 更高效地利用 return 语句.....	332
5.8.2 使用返回值的函数.....	332
5.8.3 使用 IPO 图.....	334
5.8.4 返回字符串.....	339
5.8.5 返回布尔值.....	340
5.8.6 在校验代码中使用布尔函数.....	341
5.8.7 返回多个值.....	341
5.8.8 从函数返回 None.....	342
 检查点.....	344
5.9 math 模块.....	344
math.pi 和 math.e 值.....	346
 检查点.....	346
5.10 将函数存储到模块中.....	347
条件执行模块中的 main 函数.....	351
5.11 海龟图形：使用函数将代码模块化.....	353
将图形函数存储到模块中.....	357
复习题.....	360
编程练习.....	364
第 6 章 文件和异常.....	373
6.1 文件输入和输出简介.....	373
6.1.1 文件类型.....	375
6.1.2 文件访问方法.....	375
6.1.3 文件名和文件对象.....	375



6.1.4 打开文件.....	377
6.1.5 指定文件位置.....	378
6.1.6 向文件写入数据.....	378
6.1.7 从文件读取数据.....	380
6.1.8 将换行符连接到字符串.....	384
6.1.9 读取字符串并去掉换行符.....	385
6.1.10 向现有文件追加数据.....	387
6.1.11 写入和读取数值数据.....	387
 检查点.....	391
6.2 使用循环来处理文件.....	391
6.2.1 使用循环读取文件并检测文件尾.....	392
6.2.2 使用 for 循环读取行.....	395
 检查点.....	399
6.3 使用 with 语句打开文件.....	399
6.3.1 资源.....	399
6.3.2 with 语句.....	400
6.3.3 用 with 语句打开多个文件.....	402
6.3 处理记录.....	403
 检查点.....	415
6.4 异常.....	415
6.4.1 处理多个异常.....	422
6.4.2 用一个 except 子句捕获所有异常.....	423
6.4.3 显示异常的默认错误消息.....	424
6.4.4 else 子句.....	426

6.4.5 finally 子句.....	427
6.4.6 如果异常未被处理怎么办?	427
 检查点.....	428
复习题.....	428
编程练习.....	431
第 7 章 列表和元组.....	434
7.1 序列.....	434
7.2 列表简介.....	434
7.2.1 重复操作符.....	436
7.2.2 使用 for 循环遍历列表	437
7.2.3 索引.....	438
7.2.4 len 函数.....	439
7.2.5 使用 for 循环按索引来遍历列表	439
7.2.6 列表是可变的.....	440
7.2.7 连接列表.....	442
 检查点.....	442
7.3 列表切片.....	443
 检查点.....	446
7.4 使用 in 操作符查找列表项.....	446
 检查点.....	448
7.5 列表方法和有用的内置函数.....	448
7.5.1 append 方法	449
7.5.2 count 方法	450
7.5.3 index 方法.....	450

7.5.4 insert 方法	452
7.5.5 sort 方法	452
7.5.6 remove 方法	453
7.5.7 reverse 方法.....	454
7.5.8 del 语句.....	454
7.5.9 sum 函数.....	454
7.5.10 min 和 max 函数.....	455
 检查点	455
7.6 复制列表.....	456
7.7 处理列表.....	457
7.7.1 累加列表中的数值.....	459
7.7.2 计算列表中数值的平均值.....	460
7.7.3 将列表作为实参传给函数.....	461
7.7.4 从函数中返回列表.....	462
7.7.5 随机选择列表元素.....	466
7.7.6 处理列表和文件.....	466
7.8 列表推导式.....	470
在列表推导式中使用 if 子句.....	472
 检查点	472
7.9 二维列表.....	473
 检查点	477
7.10 元组.....	478
7.10.1 将不同的元组赋给变量.....	479
7.10.2 在元组中存储可变对象.....	480






7.10.3 有什么意义?	482
7.10.4 在列表和元组之间转换.....	482
 检查点.....	483
7.11 使用 matplotlib 包绘制列表数据.....	483
7.11.1 导入 pyplot 模块	483
7.11.2 绘制折线图.....	484
7.11.3 添加标题、轴标签和网格.....	486
7.11.4 自定义 X 轴和 Y 轴.....	487
7.11.5 在数据点上显示标记.....	491
7.11.6 绘制柱状图.....	494
7.11.7 自定义条柱宽度.....	495
7.11.8 更改条柱颜色.....	497
7.11.9 添加标题、坐标轴标签和自定义刻度线标签.....	497
7.11.10 绘制饼图.....	499
7.11.11 显示切片标签和标题.....	500
7.11.12 改变切片颜色.....	502
 检查点.....	502
复习题.....	503
编程练习.....	507
第 8 章 深入字符串.....	513
8.1 基本字符串操作.....	513
8.1.1 访问字符串中的单个字符.....	513
8.1.2 使用 for 循环遍历字符串.....	513
8.1.3 索引.....	516

8.1.4 IndexError 异常.....	517
8.1.5 len 函数.....	517
8.1.6 字符串连接.....	518
8.1.7 字符串是不可变的.....	519
 检查点.....	520
8.2 字符串切片.....	521
 检查点.....	525
8.3 测试、查找和操作字符串.....	525
8.3.1 使用 in 和 not in 测试字符串.....	525
8.3.2 字符串方法.....	526
8.3.3 字符串测试方法.....	526
8.3.4 修改方法.....	528
8.3.5 查找和替换.....	530
8.3.6 重复操作符.....	535
拆分字符串.....	536
 检查点.....	541
复习题.....	542
编程练习.....	545
第 9 章 字典和集合.....	551
9.1 字典.....	551
9.1.1 创建字典.....	551
9.1.2 从字典中检索值.....	552
9.1.3 使用 in 和 not in 操作符来判断键是否存在.....	553
9.1.4 向现有字典添加元素.....	553






9.1.5 删除元素.....	554
9.1.6 获取字典中的元素个数.....	555
9.1.7 在字典中混合不同的数据类型.....	555
9.1.8 创建空字典.....	557
9.1.9 使用 for 循环遍历字典.....	557
9.1.10 一些字典方法.....	558
9.1.11 字典合并和更新操作符.....	573
9.1.12 字典推导式.....	574
9.1.13 在字典推导式中使用 if 子句.....	576
 检查点.....	577
9.2 集合.....	578
9.2.1 创建集合.....	578
9.2.2 获取集合中的元素数量.....	579
9.2.3 添加和删除元素.....	580
9.2.4 使用 for 循环来遍历集合.....	582
9.2.5 使用 in 和 not in 操作符测试集合中的值.....	582
9.2.6 并集.....	583
9.2.7 交集.....	583
9.2.8 差集.....	584
9.2.9 对称差集.....	585
9.2.10 子集和超集.....	585
9.2.11 集合推导式.....	589
 检查点.....	589
9.3 对象序列化.....	591

 检查点	596
复习题	597
编程练习	602
第 10 章 类和面向对象编程	607
10.1 过程式编程和面向对象编程	607
10.1.1 对象的可重用性	609
10.1.2 日常生活中的对象	610
 检查点	611
10.2 类	611
10.2.1 类定义	613
10.2.2 隐藏属性	619
10.2.3 将类存储到模块中	622
10.2.4 BankAccount 类	624
10.2.5 __str__ 方法	626
 检查点	629
10.3 操作类的实例	629
10.3.1 取值和赋值方法	635
10.3.2 将对象作为参数传递	638
 检查点	651
10.4 类设计技术	652
10.4.1 统一建模语言(UML)	652
10.4.2 确定解决问题所需的类	654
10.4.3 这只是开始	669
 检查点	669

复习题.....	669
编程练习.....	672
第 11 章 继承.....	677
11.1 继承简介.....	677
11.1.1 泛化和特化.....	677
11.1.2 继承和“属于”关系	678
11.1.3 在 UML 图中表示继承	686
 检查点	691
11.2 多态性.....	691
isinstance 函数	694
 检查点	698
复习题.....	698
编程练习.....	700
第 12 章 递归.....	702
12.1 递归简介.....	702
12.2 用递归解决问题.....	706
12.2 使用递归来计算阶乘.....	707
直接递归和间接递归.....	710
 检查点	710
12.3 递归算法示例.....	710
12.3.1 用递归对列表元素的一个范围进行求和	710
12.3.2 斐波那契数列.....	711
12.3.3 寻找最大公约数.....	713
12.3.4 汉诺塔.....	714

12.3.5 递归与循环.....	717
复习题.....	718
编程练习.....	720
第 13 章 GUI 编程	722
13.1 图形用户界面.....	722
GUI 程序是由事件驱动的.....	723
 检查点	724
13.2 使用 tkinter 模块.....	724
 检查点	728
13.3 使用 Label 控件显示文本.....	729
13.3.1 为标签添加边框.....	732
13.3.2 填充.....	734
 检查点	740
13.4 使用 Frame 来组织控件.....	741
13.5 Button 控件和消息框.....	743
创建退出按钮.....	746
13.6 用 Entry 控件获取输入	747
13.7 将标签用作输出字段.....	750
 检查点	758
13.8 单选钮和复选框.....	759
13.8.1 单选钮.....	759
13.8.2 为 Radiobutton 指定回调函数.....	762
复选框	762
 检查点	765

13.9 <i>Listbox</i> 控件.....	765
13.9.1 指定列表框大小.....	767
13.9.2 使用循环来填充列表框.....	767
13.9.3 在列表框中选择项.....	768
13.9.4 获取选中的一项或多项.....	769
13.9.5 从列表框中删除项.....	771
13.9.6 当用户单击列表项时执行回调函数.....	772
13.9.10 为列表框添加滚动条.....	776
 检查点.....	785
13.10 使用 <i>Canvas</i> 控件绘制形状.....	785
13.10.1 <i>Canvas</i> 控件的屏幕坐标系.....	785
13.10.2 画线: <code>create_line</code> 方法.....	788
13.10.3 画矩形: <code>create_rectangle</code> 方法.....	791
13.10.4 画椭圆: <code>create_oval</code> 方法.....	793
13.10.5 画弧线: <code>create_arc</code> 弧形方法.....	796
13.10.6 画多边形: <code>create_polygon</code> 方法.....	801
13.10.7 绘制文本: <code>create_text</code> 方法.....	804
 检查点.....	809
复习题.....	809
编程练习.....	813
第 14 章 数据库编程.....	817
14.1 数据库管理系统.....	817
14.1.1 SQL.....	819
14.1.2 SQLite.....	819

 检查点	819
14.2 表、行和列	819
14.2.1 列数据类型	821
14.2.2 主键	822
14.2.3 标识列	822
14.2.4 允许空值	823
 检查点	823
14.3 使用 SQLite 打开和关闭数据库连接	824
14.3.1 指定数据库在磁盘上的位置	825
14.3.2 向 DBMS 传递 SQL 语句	826
 检查点	826
14.4 创建和删除表	827
14.4.1 创建表	827
14.4.2 创建多个表	829
14.4.3 仅在表不存在时创建表	830
14.4.4 删除表	831
 检查点	831
14.5 向表中添加数据	831
14.5.1 用一个 INSERT 语句插入多行	834
14.5.2 插入 NULL 数据	835
14.5.3 插入变量值	835
14.5.4 警惕 SQL 注入攻击	837
 检查点	838
14.6 使用 SQL SELECT 语句查询数据	839

14.6.1 示例数据库.....	839
14.6.2 SELECT 语句.....	840
14.6.3 选择表中的所有列.....	843
14.6.4 使用 WHERE 子句指定搜索条件.....	845
14.6.5 SQL 逻辑操作符：AND，OR 和 NOT	848
14.6.6 SELECT 语句中的字符串比较	848
14.6.7 使用 LIKE 操作符.....	849
14.6.8 对 SELECT 查询的结果排序	850
14.6.9 聚合函数.....	852
 检查点.....	854
14.7 更新和删除现有行.....	855
14.7.1 更新行.....	855
14.7.2 更新多列.....	858
14.7.3 确定更新的行数.....	858
14.7.4 使用 DELETE 语句删除行.....	859
14.7.5 确定删除的行数.....	861
 检查点.....	861
14.8 深入主键.....	861
14.8.1 SQLite 中的 RowID 列.....	862
14.8.2 SQLite 中的整数主键	862
14.8.3 非整数主键.....	863
14.8.4 复合键.....	863
 检查点.....	864
14.9 处理数据库异常.....	865

 检查点	867
14.10 CRUD 操作	867
14.11 关系数据	875
14.11.1 外键	878
14.11.2 实体关系图(ERD)	878
14.11.3 用 SQL 创建外键	879
14.11.4 在 SQLite 中强制外键约束	880
14.11.5 更新关系数据	883
14.11.6 删除关系数据	884
14.11.7 在 SELECT 语句中从多个表检索列	884
 检查点	892
复习题	892
编程练习	897
附录 A 安装 Python	901
下载 Python	901
为 Windows 安装 Python 3.x	901
附录 B IDLE 简介	903
启动 IDLE 并使用 Python Shell	903
在 IDLE 编辑器中写 Python 程序	904
语法彩色标注	905
自动缩进	906
保存程序	906
运行程序	907
其他资源	908

附录 C ASCII 字符集	909
附录 D 预定义颜色名称	910
附录 E import 语句	917
导入特定函数或类.....	917
通配符导入.....	918
使用别名.....	918
附录 F 用 format() 函数格式化输出	920
用科学计数法格式化.....	921
插入逗号分隔符.....	921
指定最小域宽.....	922
将浮点数格式化为百分比.....	924
格式化整数.....	924
附录 G 用 pip 工具安装模块	925
附录 H 知识检查点答案	926
第 1 章.....	926
第 2 章.....	927
第 3 章.....	930
第 4 章.....	932
第 5 章.....	934
第 6 章.....	936
第 7 章.....	937
第 8 章.....	940
第 9 章.....	941
第 10 章.....	943
第 11 章.....	944

第 12 章	944
第 13 章	945
第 14 章	947
术语表	950

译者序

Python 起源于 1989 年末。当时，CWI（阿姆斯特丹国家数学和计算机科学研究所）的研究人员 Guido van Rossum 需要一种高级脚本编程语言，为他的研究小组的 Amoeba 分布式操作系统执行管理任务。为了创建这种新语言，他从高级教学语言 ABC（All Basic Code）汲取了大量语法，并从系统编程语言 Modula-3 借鉴了错误处理机制。然而，ABC 的一个重大缺点是扩展性不足；语言不是开放式的，不利于改进或扩展。因此，Van Rossum 决定在新语言中，合成来自现有语言的许多元素，但要求必须能通过类和编程接口进行扩展。他将这种新语言命名为 Python（大蟒蛇）——来源于当时流行的 BBC 喜剧片集“Monty Python”（巨蟒剧团）。

自 1991 年初公开发售后，Python 开发人员和用户社区逐渐壮大，使其逐渐演变成一种成熟的、并获得了良好支持的编程语言。人们用 Python 开发了大量应用程序，从创建网上电子邮件程序，到控制水下自走车辆，以及配置操作系统和创建动画片，再到最近爆火的 AI 应用等等。

Python 是一种模块化的可扩展语言；它能随时集成新的“模块”（modules）——这是一种可重用的软件组件。任何 Python 开发人员都能写自己的新模块，对 Python 的功能进行扩充。Python 源代码和模块的一个重要集散地是官方的 PyPI（pypi.org）。

Python 经过了良好的设计，无论新手还是有经验的程序员才能快速学习和理解这种语言，并能轻松上手。和其他语言不同，Python 具有良好的移植和扩展能力。Python 的语法和设计促进了良好的编程实践，而且可以在不牺牲程序扩展性与维护性的同时，显著缩短开发时间。

自 2003 年翻译并出版了 Deitel 著名的《Python 编程宝典》（Python How to Program）一书后，虽然译者没有继续从而这个主题的翻译，但在工作中一直在使用这种方便、快捷的语言，而且亲身经历了它从 2.x 到 3.x 版本的迭代。时至今日，Python 已经取得了长足的发展，应用越来越广泛，其用户视区也越来越壮大。根据 TIOBE 的最新排行，Python 已经长时间占据编程语言排行榜的第一位，流行度达到 13.42%（2023 年 7 月）。这背后虽然有人工智能（AI）爆火的推动，但我们不要忘记，Python 之所以流行，还是跟它本身的特点有关。

作为一种通用语言，Python 可以用于各种应用程序，“简单易用”的特点也使得它成为用于自动化任务、构建网站或软件和分析数据的不错的选择。此外，易读、开源、跨平台、可扩展性、具有一个强大的标准库等特性，也使其在开发人员和工程师中很受欢迎。

今天，我很高兴为大家介绍《Starting Out with Python》一书的中文版。由全球知识教育作家 Tony Gaddis 编写的这本教科书是学习 Python 编程的绝佳入门之选。本书覆盖了从基础概念到实际应用的全套 Python 知识。无论你是初次接触编程，还是想要从其他编程语言过

渡到 Python，这本书都能帮助你轻松上手。书中以清晰而易懂的语言，系统地介绍了 Python 的核心概念、语法、数据类型、控制结构、面向对象编程、GUI 编程和数据库编程，为你建立一套完整的知识体系。

Tony Gaddis 在书中注重理论与实践的结合。通过丰富的示例、练习和项目，你有机会将所学的知识应用于实际问题的解决中。无论是编写小型脚本还是构建复杂的应用程序，你都将通过实际动手实践而深入理解 Python 编程的精髓。

此外，本书强调了编程思维和解决问题的能力。每一章都配有精心设计的练习，旨在锻炼你的逻辑思维和创新的能力。通过解决各种不同难度级别的编程挑战，你将逐步培养自己的编程思维，并在解决实际问题时游刃有余。

另外，本书的源代码大多进行了中文本地化，包括注释、程序中显示的文本等。除此之外，还对书中的一些 bug（有些是祖传的）进行了修正。请访问译者主页 <https://bookzhou.com> 下载中文版资源。

我衷心相信，《Starting Out with Python》中文版将成为你学习 Python 编程的得力伴侣。无论你的背景和经验如何，都能在这本书中找到合适的内容，逐步掌握 Python 编程的精髓。希望你在学习的过程中获得乐趣，掌握实用的技能，为未来的学习和职业发展打下坚实的基础。

祝愿你在《Starting Out with Python》的学习旅程中取得丰硕的成果！

——周靖，2023 年 8 月于北京

前言

欢迎阅读《学习 Python 中文版》第六版。本书使用 Python 语言来讲授编程概念和解决问题的技能，不要求学生之前有任何编程经验。通过易于理解的例子、伪代码、流程图和其他工具，学生将学会如何设计程序的逻辑，然后用 Python 实现这些程序。本书是编程入门课程或以 Python 为语言的编程逻辑和设计课程的理想选择。

和本书英文版“Starting Out With”系列的其他所有书籍一样，本书的特点是清晰、友好、易懂。另外，还提供了丰富的示例程序，它们都简洁而实用。一些例子比较短，强调了当前的特定编程主题。还有一些例子侧重于问题的解决，涉及到的东西会多一些。每章都提供了一个或多个称为“聚光灯”的案例研究，对一个具体问题进行分析，并展示如何解决这个问题。

首先讲控制结构，然后讲类

Python 是一种完全面向对象的编程语言，但学生不必理解面向对象的概念就可以开始用 Python 进行编程。本书首先介绍了数据存储、输入和输出、控制结构、函数、序列/列表、文件 I/O 以及创建标准库的类的对象等基础知识。然后，将学习如何编写类，探索继承和多态性的主题，并学习写递归函数。最后，将学习如何开发简单的、事件驱动的 GUI 应用程序。

第 6 版的变化

本书清晰的写作风格与上一版保持一致。然而，既然时间都来到 2023 年了，所以本书也做了不少补充和改进，下面进行了总结。

- 针对 Python 3.11 进行了全面更新：该版本使用了最高到 Python 3.11 的新语言特性。
- **with 语句**：第 6 章现在引入了 with 语句来作为打开文件的一种方式。全书增加了许多 with 语句与文件结合使用的例子。
- **多重赋值**：这一版在第 2 章引入了多重赋值的概念。
- **单行 if 语句**：第 3 章专门用一节解释了单行 if 语句。
- **条件表达式**：第 3 章现在介绍了条件表达式和三元操作符。
- **海象操作符和赋值表达式**：第 3 章新增了一节来讲述海象操作符和赋值表达式。第 3 章提供了在 if 语句中使用赋值表达式的例子，第 4 章提供了在 while 循环中使用赋值表达式的例子。
- **while 循环作为计数控制循环**：第 4 章新增了一节来讲述计数器变量和如何使用 while

语句编写计数控制循环。

- 单行 `while` 循环：第 4 章新增了一节来讲述单行 `while` 循环
- 为循环使用 `break`，`continue` 和 `else`：第 4 章新增了一节来讲述如何为循环使用 `break` 和 `continue`，以及如何为循环使用 `else` 子句，后者是 Python 独有的。
- 仅关键字参数：第 5 章现在讨论了仅关键字参数以及如何在函数中实现它们。
- 仅位置参数：第 5 章现在讨论了仅位置参数以及如何在函数中实现它们。
- 默认实参：第 5 章新增了一节来讲述函数的默认实参。
- 为列表使用 `count` 和 `sum`：第 7 章现在讨论了如何为列表使用 `count` 方法和 `sum` 函数。
- 在元组中存储可变对象：第 7 章新增了一节来讲述元组的“不可变”性，以及如何在元组中存储“可变”对象。
- 字典合并和更新操作符：第 9 章新增了一节来讨论字典的合并和更新操作符、

各章内容概览

第 1 章 计算机和编程概述

本章首先以具体和容易理解的方式来诠释计算机如何工作、数据如何存储和处理以及我们为什么要用高级语言来写程序。还介绍了 Python 语言、交互模式、脚本模式以及 Python 自带的 IDLE 开发环境。

第 2 章：输入、处理和输出

本章介绍了程序开发周期、变量、数据类型以及用顺序结构来写的简单程序。学生将学习如何编写简单的程序，从键盘读取输入，进行数学运算，并生成格式化的屏幕输出。还介绍了作为程序设计工具的伪代码和流程图。本章最后介绍了海龟图形库（选读）。

第 3 章：判断结构和布尔逻辑

在本章中，学生将学习关系操作符和布尔表达式，并了解如何用判断结构控制程序的执行流程。我们讨论了 `if`、`if-else` 和 `if-elif-else` 语句。还讨论了嵌套判断结构和逻辑操作符。本章包括一个选读的海龟图形小节，讨论了如何使用判断结构来测试海龟的状态。

第 4 章：循环结构

本章介绍了如何使用 `while` 和 `for` 循环结构。讨论了计数器、累加器和哨兵，还讨论了如何利用循环对输入进行校验。本章最后的选读小节讨论了如何在循环中利用海龟图形库来绘制图形。

第 5 章：函数

本章首先介绍了如何编写和调用不返回值的函数（void 函数^①）。解释了使用函数对程序进行模块化的好处，并讨论了自上而下设计方法。然后，讨论了如何向函数传递实参。讨论了常用的库函数，例如用于生成随机数的函数。在讨论了如何调用库函数和使用其返回值之后，我们将讨论如何编写和调用自定义函数。然后，将讨论如何使用模块来组织函数。选读小节讨论了如何使用函数对海龟图形代码进行模块化。

第 6 章：文件和异常

本章介绍了顺序文件输入和输出。学生将学习如何读写大型数据集，并将数据作为字段和记录来存储。本章最后讨论了异常，并展示了如何编写异常处理代码。

第 7 章：列表和元组

本章介绍了 Python 的“序列”概念，并探讨了两种常见的 Python 序列：列表和元组。学生将学习如何使用列表进行类似于数组的操作，例如在列表中存储对象、遍历列表、查找列表中的数据项以及对列表项进行求和/求平均值。本章讨论了列表推导式、切片以及许多列表方法。我们兼顾了一维和二维列表。本章还讨论了如何用 `matplotlib` 包提供的功能将列表数据绘制成图表。

第 8 章：深入字符串

本章更深入地讨论了字符串处理。讨论了字符串切片和遍历字符串中的单独字符的算法，并介绍了如何用几个内置函数和字符串方法来进行字符和文本处理。本章还提供了字符串标记化（tokenizing）和解析 CSV 文件的例子。

第 9 章：字典和集合

本章介绍了字典和集合数据结构。讨论了如何在字典中以“键值对”的形式存储数据、查找值、更改现有值、添加新的键值对、删除键值对以及如何写字典推导式。将学习如何在集合中将值作为具有唯一性的元素来存储，并执行常见的集合操作，例如并集（union）、交集（intersection）、差集（difference）和对称差集（symmetric difference）。还介绍了集合推导式。本章最后讨论了对象序列化（Python 称为“腌制”），并介绍了 Python `pickle` 模块。

^① 译注：和其他语言不同，Python 没有专门提供一个 void 关键字。

第 10 章：类和面向对象编程

本章比较了过程式和面向对象编程实践。讨论了类和对象的基本概念。讨论了属性、方法、封装和数据隐藏、`__init__` 方法（类似于构造函数）、取值方法（`accessor`）和赋值方法（`mutator`）。学生将学习如何用 UML 对类进行建模，以及如何在从特定的问题中找出合适的类。

第 11 章：继承

本章继续研究类，讨论继承和多态性。所涉及的主题包括超类（基类）、子类（派生类）、`__init__` 方法在继承中的作用、方法重写（`overriding`）和多态性。

第 12 章：递归

本章讨论了递归以及如何用它优雅地解决一些令人挠头的问题。展示了递归调用的一个可视化跟踪，并讨论了递归应用程序。介绍了如何用递归算法来完成许多任务，包括计算阶乘，寻找最大公约数（GCD），对列表中的一系列数值进行求和。另外，还介绍了经典的汉诺塔例子，这个问题特别适合用递归来求解。

第 13 章：GUI 编程

本章讨论了如何使用 Python 的 `tkinter` 模块来设计 GUI 应用程序。我们讨论了许多基本组件，例如标签、按钮、文本输入框（`entry fields`）、单选钮、复选框、列表框和对话框等。学生还将学习事件在 GUI 应用程序中是如何工作的，以及如何编写回调函数来处理事件。本章讨论了 `Canvas`（画布）组件，以及如何使用它来绘制直线、矩形、椭圆、弧线、多边形和文本。

第 14 章：数据库编程

本章介绍了数据库编程。首先讲述了数据库的基本概念，例如表、行和主键。然后讨论了如何在 Python 中使用 `SQLite` 连接数据库。我们讨论了 `SQL`，学生将学习如何执行查询，以及如何执行语句来查找行、添加新行、更新现有行和删除行。本章演示了执行 `CRUD`（创建、读取、更新和删除）操作的应用程序，最后还会讨论关系数据库。

附录 A：安装 Python

本附录解释了如何下载和安装最新的 Python 发行版。

附录 B：IDLE 简介

本附录概述了随同 Python 安装的 IDLE 集成开发环境。

附录 C：ASCII 字符集

本附录列出了完整 ASCII 字符集供参考。

附录 D：预定义颜色名称

本附录列出了可用于海龟图形库、matplotlib 和 tkinter 的预定义颜色名称。

附录 E：import 语句

本附录讨论了使用 import 语句的各种方法。例如，可以使用 import 语句来导入模块、类、函数或者为模块指定别名。

附录 F：用 format() 函数格式化输出

本附录讨论了 format() 函数的用法，介绍了如何使用它的格式说明符（format specifier）来控制数值的显示方式。

附录 G：用 pip 工具安装模块

本附录讨论了如何使用 pip 工具安装来自 Python Package Index（PyPI）的第三方模块。

附录 H：知识检查点答案

本附录给出了散布于全书的“知识检查点”的答案。

本书的组织方式

本书以循序渐进的方式讲授编程。每一章都涵盖了一组主要的主题，并随着学习的进行逐步建立起知识体系。尽管各章很容易按照现有的顺序讲授/自学，但在讲授/自学顺序上有一些灵活性。图 P-1 展示了各章的依赖性。每个方框都代表一章或多章。箭头从一章指向在它之前必须掌握的一章。

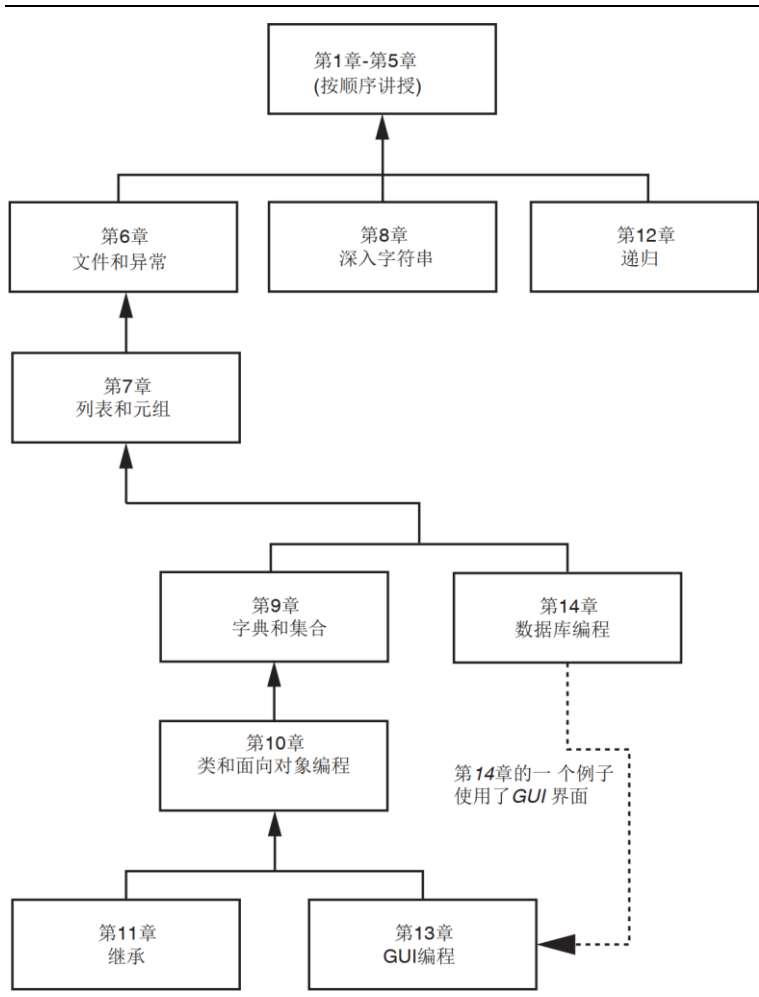








图 P-1 各章的依赖关系

本书特色

概念陈述	本书的每个主题小节都从一条概念陈述开始，概括了当前小节的主要内容。
示例程序	每一章都提供了大量或完整或不完整的示例程序，每个都是为了强调当前的主题而设计的。
 聚光灯	每一章都有一个或多个“聚光灯”案例研究，会详细地、逐步深入地分析问题，并告诉学生如何解决这些问题。
 视频讲解	专门为本书开发的“视频讲解”（VideoNote）网上视频，可在 https://media.pearsoncmg.com/ph/esm/ecs_gaddis_sowpython_6e/cw/

	观看。该图标散布于全书，提醒学生观看针对特定主题的视频。由于是英文版视频，所以本书的“视频讲解”保留了视频的英文名称，方便在上述页面上查找。
 注意	全书多处提供了“注意”。它们简要解释了与当前主题相关的有趣或经常被误解的要点。
 提示	“提示”告诉学生在处理不同编程问题的最佳技术。
 警告	“警告”提醒学生注意可能导致程序出错或数据丢失的编程技术或做法。
 检查点	“检查点”散布于每一章，旨在检验学生是否已经掌握了一个新主题。
复习题	每章都提供了一套全面而多样的复习题和练习，包括选择题、判断题、算法工作台和简答题。
编程练习	每一章都提供了大量编程练习，旨在巩固学生对当前所学主题的掌握。

补充材料

学生在线资源

出版商为本书提供了许多学生资源。以下资源可从 https://media.pearsoncmg.com/ph/esm/ecs_gaddis_sowpython_6e/cw/ 获取：

- 书中所有示例程序的源代码
- 英文版勘误（中文版资讯和勘误请访问 <https://bookzhou.com>）
- 本书配套的“视频讲解”（VideoNote）

中文版资源

本书的源代码大多进行了中文本地化，包括注释、程序中显示的文本等。除此之外，还对一些 bug 进行了修正。请访问 <https://bookzhou.com> 下载中文版资源。

教师资源

以下补充材料仅向符合资格的教师提供：

- 所有复习题的答案

-
- 所有编程练习的答案
 - 各章的 PowerPoint 幻灯片
 - 题库

请访问培生教育教师资源中心（www.pearsonhighered.com/irc）或联系当地培生教育代表，了解如何获取这些资料。

致谢

感谢以下老师担任本书的评审，感谢他们的洞察力、专业知识和周到的建议。

Desmond K. H. Chun <i>Chabot Community College</i>	Raymond Pettit <i>Abilene Christian University</i>
Sonya Dennis <i>Morehouse College</i>	Janet Renwick <i>University of Arkansas–Fort Smith</i>
Barbara Goldner <i>North Seattle Community College</i>	Haris Ribic <i>SUNY at Binghamton</i>
Paul Gruhn <i>Manchester Community College</i>	Ken Robol <i>Beaufort Community College</i>
Bob Husson <i>Craven Community College</i>	Eric Shaffer <i>University of Illinois at Urbana-Champaign</i>
Diane Innes <i>Sandhills Community College</i>	Tom Stokke <i>University of North Dakota</i>
Daniel Jinguji <i>North Seattle Community College</i>	Anita Sutton <i>Germanna Community College</i>
John Kinuthia <i>Nazareth College of Rochester</i>	Ann Ford Tyson <i>Florida State University</i>
Frank Liu <i>Sam Houston State University</i>	Karen Ughetta <i>Virginia Western Community College</i>
Gary Marrer <i>Glendale Community College</i>	Christopher Urban <i>SUNY Institute of Technology</i>
Keith Mehl <i>Chabot College</i>	Nanette Veilleux <i>Simmons College</i>
Shyamal Mitra <i>University of Texas at Austin</i>	Brent Wilson <i>George Fox University</i>
Vince Offenback <i>North Seattle Community College</i>	Linda F. Wilson <i>Texas Lutheran University</i>
Smiljana Petrovic <i>Iona College</i>	

感谢海伍德社区学院（Haywood Community College）的教职员工和管理部门，让我有机会教授我所喜爱的学科。还要感谢我的家人和朋友，感谢他们对我所有项目的无私支持。

本书能由培生出版是一个巨大的荣誉，我非常幸运地拥有 Tracy Johnson 作为我的编辑和内

容经理。她和她的同事 Holly Stark、Erin Sullivan、Krista Clark、Scott Disanno、Sandra Rodriguez、Bob Engelhardt、Abhijeet Gope、Adarsh Sushanth、Pallavi Pandit 和 Anu Sivakolundu 为本书的出版和推广做出了不懈的努力。感谢你们所有人！

作者简介

Tony Gaddis 在海伍德社区学院执教 20 多年，向各种各样的学生线上或线下教授计算机科学课程。他特别喜欢指导非计算机专业的学生以及那些最开始在编程概念上遇到困难的人。他是一位备受赞誉的教师，曾被评选为北卡罗来纳社区学院年度教师。

Tony 还是一位多产的作家。他的《...深入浅出》(Starting Out With) 系列教科书由 Pearson 出版，涵盖了广泛的编程语言和 CS1、CS2 课堂教学方法。Tony 的每本书都融入了他对教学的热情以及他向初学者解释困难概念的经验。



第 1 章 计算机和编程概述

1.1 概述

想象一下人们使用计算机的不同方式。在学校里，学生们使用计算机完成一些任务，例如写论文、搜索文章、发送电子邮件和上网课。在工作中，人们用计算机分析数据、做演示、进行商业交易、与客户和同事沟通、控制工厂的机器以及做其他许多事情。在家里，人们使用计算机完成各种任务，例如支付账单、网上购物、与朋友和家人聊天以及玩游戏。别忘了，手机、平板电脑、智能手机、汽车导航系统和许多其他设备本质上也是计算机。在我们的日常生活中，计算机几乎无所不在。

计算机可以执行如此广泛的任務，原因是它们可以被编程。这意味着计算机不是被设计成只做一种工作，而是做程序告诉它们要做的任何工作。**程序**（program）是一组指令，计算机遵循这些指令来执行任务。例如，图 1-1 展示了 Microsoft Word 和 PowerPoint 这两种常用程序的屏幕。

我们经常将程序称为**软件**（software）。软件对计算机来说必不可少，因为它控制着计算机的一切工作。所有用来让计算机发挥作用的软件都是由作为**程序员**（programmer）或**软件开发人员**（software developers）的人创建的。这些人经过训练之后，掌握了设计、创建和测试计算机程序所需的技能。计算机编程是一个令人兴奋和有价值的职业。今天，你会在商业、医学、政府、执法、农业、学术、娱乐和其他许多领域发现程序员的身影。

本书讲解了使用 Python 语言进行计算机编程的基础知识。Python 是初学者的一个很好的选择，因为它很容易学习，使用它可以快速完成程序的编写。Python 也是一种强大的语言，受到了专业软件开发人员的热烈欢迎。事实上，根据报道，Google、NASA、YouTube、各大游戏公司、纽约证券交易所和其他许多机构都在使用 Python。

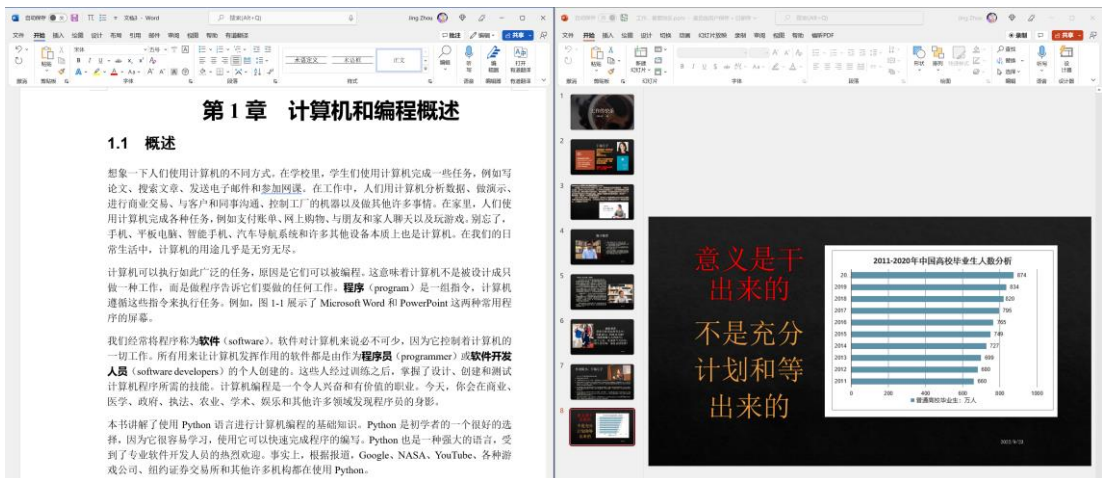


图 1-1 示例文字处理程序和演示程序

在开始探索编程的概念之前，需要先对计算机及其工作方式有一个基本的了解。本章将为你打下坚实的基础，在将来学习计算机科学的过程中，将不断地依赖这些知识。首先，我们将讨论计算机通常包含哪些物理组件。接着，将讨论计算机如何存储数据和执行程序。最后，本章简单介绍了用于写 Python 程序的软件。

1.2 硬件和软件

概念：构成计算机的物理设备称为计算机的硬件。在计算机上运行的程序称为软件。

1.2.1 硬件

硬件（Hardware）是指构成一台计算机的所有物理设备，或者称为**组件**（components）。计算机不是一个独立的设备，而是由多个设备构成的一个系统，这些设备要协同工作。如同交响乐团的各种不同的乐器，计算机中的每个设备都在发挥它自己的作用。

如果自己购买过计算机，那么可能会看到电脑配置单，上面列出了计算机的各种组件，包括处理器、内存、硬盘、显示器、显卡等等。除非自己很熟悉计算机，或者至少有一个朋友熟悉，否则从头开始理解这些组件（而且不被奸商坑）还是颇具挑战性的。如图 1-2 所示，一台典型的计算机由以下主要组件构成：

- CPU
- 主存
- 辅助存储设备
- 输入设备
- 输出设备

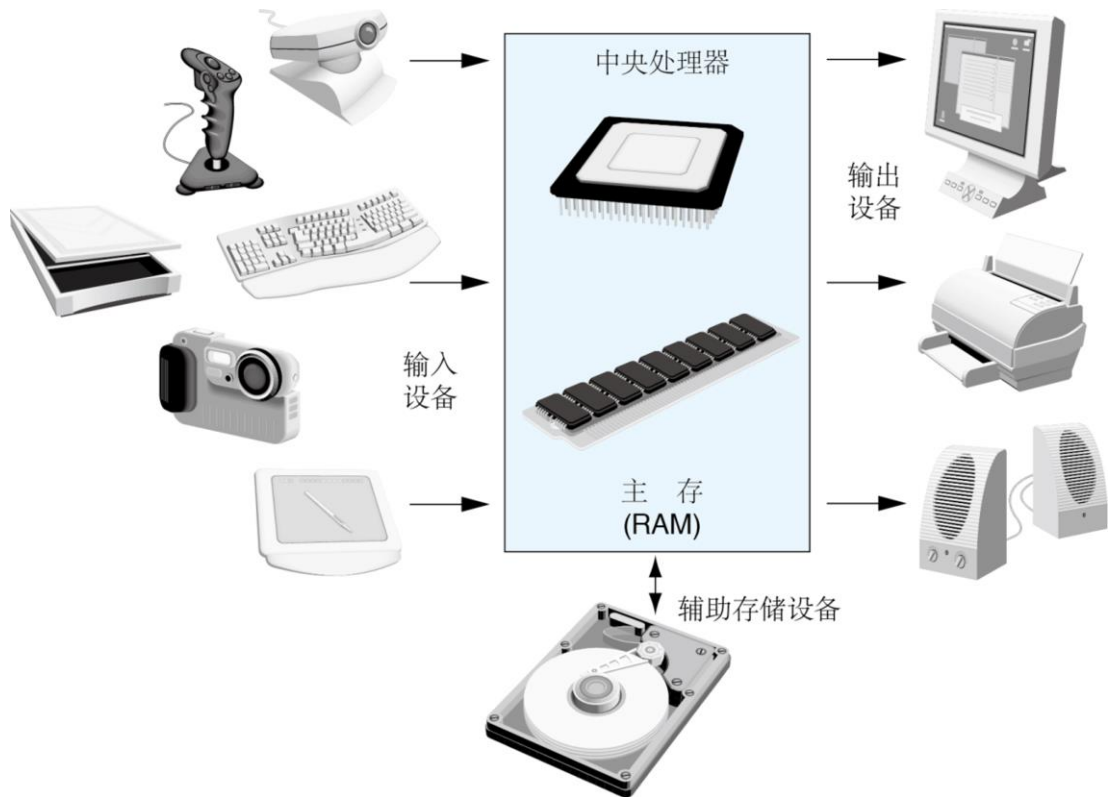


图 1-2 计算机系统的典型组件

下面让我们详细研究一下每个组件。

1.2.2 CPU

当计算机执行程序所要求的任务时，就说计算机正在**运行或执行程序**。**中央处理器**（Central Processing Unit, CPU）是负责实际运行程序的计算机组件。CPU 是最重要的一个组件，计算机没有它就不能运行程序。如果说软件是计算机的灵魂，那么 CPU 就是计算机的大脑。

在最早的计算机中，CPU 是由大量电子和机械零件（例如电子管和开关）组成的巨型设备。图 1-3 展示了这样的一个设备。照片中的两个女生操纵的是历史上赫赫有名的伊尼亚克（ENIAC）计算机。ENIAC 是历史上公认的第一台可编程电子计算机。它于 1945 年建造完成，用于为美国陆军的新型大炮研究执行弹道计算。这台机器基本上就是一个巨大的 CPU，它有 8 英尺高，100 英尺长，重达 30 吨。

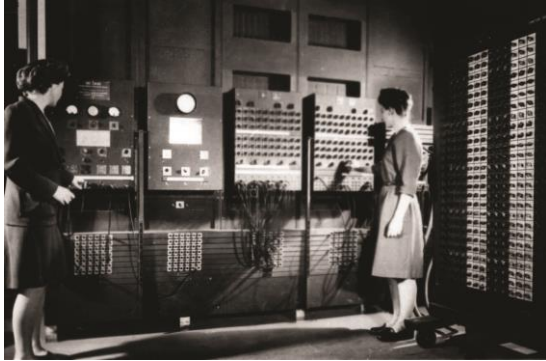


图 1-3 ENIAC 计算机（美国陆军照片）

如今的 CPU 已经变成了非常小的芯片，我们称为**微处理器**（microprocessor）。图 1-4 展示了一名实验室工作人员手持一枚新型微处理器的照片。除了比老式的电机式 CPU 小得多之外，微处理器的性能也要强劲得多。

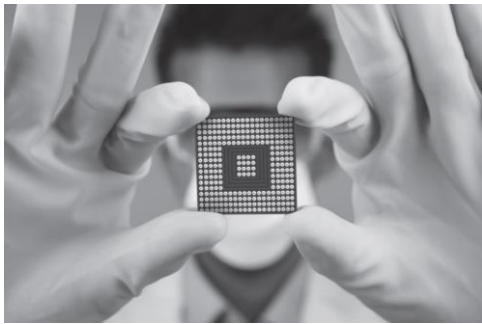


图 1-4 一名实验室工作人员手持一片微处理器

1.2.3 主存

可以将**主存**（Main memory）想象成计算机的工作区域。计算机将正在运行的程序和程序要处理的数据存储到这里。例如，假定用字处理程序为当前正在上的一门课写短文。在这个过程中，无论字处理程序还是那篇文章都存储在主存中。

主存一般简称为内存或 RAM。RAM 是**随机访问存储器**（Random-access memory）的简称。之所以叫这个名字，是因为 CPU 能快速访问存储在 RAM 中任意随机位置的数据。RAM 一般是**易失性**（volatile）的存储器，仅供程序运行时进行临时存储。计算机关机时，RAM 中的内容会消失。在计算机内部，RAM 是以**内存条**的形式提供的，如图 1-5 所示。

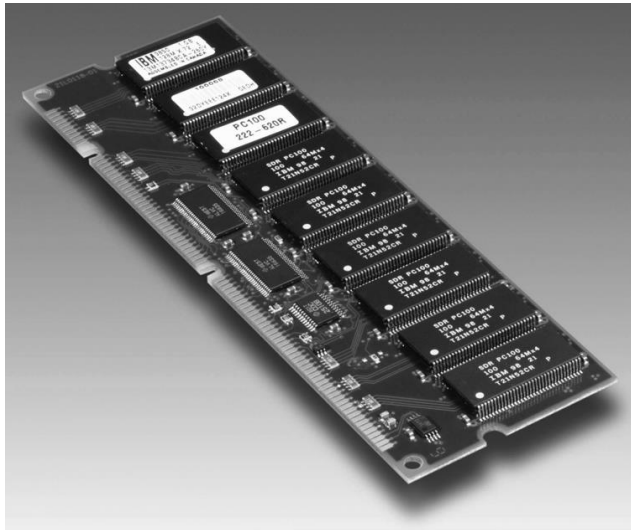


图 1-5 内存条

1.2.4 辅助存储设备

辅助存储（Secondary Storage）设备^①可以持久保存数据，即使在计算机关机之后也不会丢失。程序一般都保存到这种设备中，并在需要的时候加载到主存。重要数据——例如字处理文档、工资表数据以及库存记录——也用这种设备来保存。

最常见的辅助存储设备是硬盘驱动器。传统的**硬盘驱动器**（Hard Disk Drive, HDD）对数据进行磁性编码，并将编码后的数据记录到圆形碟片上，从而实现数据存储。目前，更受欢迎的固态硬盘。**固态硬盘**（Solid-State Drive, SSD）没有机械部件，存取速度比传统硬盘驱动器快得多。大多数计算机都配备了某种形式的辅助存储设备，要么是传统的硬盘驱动器，要么是固态驱动器。另外，还可以配备外置存储设备（例如 NAS），它们连接到计算机的某个通信端口。可以利用外置存储设备备份重要数据，或者将数据转移到另一台计算机。

除了外置存储设备，还有其他许多类型的设备可供备份和转移数据。例如，可以将 U 盘插入计算机的 USB 端口，系统会将其识别为一个磁盘驱动器。但是，这种驱动器实际并不包含一张“磁盘”。数据实际存储到一种特殊的、称为**闪存**（flash drive）的存储器中。U 盘的特点是便宜、可靠和小巧，很容易携带，可以很容易地放到口袋中或者串到钥匙环上。

1.2.5 输入设备

输入（input）是指计算机从人和其他设备那里收集的任何数据。收集数据并将其发送给计

^① 译注：或者称为“二级存储设备”。

算机的设备称为**输入设备**。常见的输入设备包括键盘、鼠标、游戏摇杆、扫描仪、麦克风和数码相机等。磁盘驱动器和 U 盘也被认为是输入设备，因为我们从中获取程序和数据，并加载到计算机的内存中。

1.2.6 输出设备

输出（output）是指由计算机生成，供人或其他设备使用的任何数据。它可能是一份销售报表、一个姓名清单或者一幅图像。数据发送到一个**输出设备**，后者对数据进行格式化，并把它们显示出来。常见的输出设备包括显示器和打印机等。硬盘驱动器和 U 盘也被认为是输出设备，因为系统要将数据发送到它们那里供长期存储。

1.2.7 软件

计算机要想发挥作用，软件是不可缺少的。从打开电源开关直到关机，计算机所做的一切都在软件的控制之下。软件一般分为两类：系统软件和应用软件。大多数计算机程序都能明显地划归于其中一类。下面让我们仔细看看每一类。

系统软件

控制和管理计算机基本操作的程序一般称为**系统软件**（system software）。系统软件通常包括以下类型的程序：

- **操作系统** 操作系统（operating system）是计算机最基本的一组程序的集合。操作系统控制计算机硬件的内部操作，管理连接到计算机的所有设备，允许将数据保存到存储设备和从存储设备取回，并允许其他程序在计算机上运行。笔记本和台式机的主流操作系统包括 Windows、macOS 和 Linux。移动设备的主流操作系统则包括安卓和 iOS。
- **实用程序** 实用程序（utility program）执行一项专门的任务，用于增强计算机的操作或者保护数据。实用程序的例子有防病毒程序，文件压缩程序和数据备份程序。
- **软件开发工具** 软件开发工具（software development tool）是程序员用来创建、修改和测试软件的程序。汇编器、编译器和解释器都属于这一类。

应用软件

用计算机处理日常任务的程序称为**应用软件**（application software）。我们平时大多数时间运行的都是这种程序。本章开头的图 1-1 展示了两个常用的应用软件的屏幕：字处理程序 Microsoft Word 和演示程序 PowerPoint。其他一些应用软件的例子包括电子表格程序、电子邮件程序、Web 浏览器和游戏等。

检查点

- 1.1 什么是程序？
- 1.2 什么是硬件？
- 1.3 列出计算机系统的 5 个主要组件。
- 1.4 实际运行程序的是计算机的什么组件？
- 1.5 在程序运行期间，计算机的什么组件是作为一个工作区域来存储程序及其数据？
- 1.6 计算机的什么组件负责长时间（即使在关机之后）存储数据？
- 1.7 计算机的什么组件负责从人或其他设备那里收集数据？
- 1.8 计算机的什么组件负责为人或其他设备格式化和显示数据？
- 1.9 哪一组基本程序在控制着计算机硬件的内部运作？
- 1.10 如何称呼执行一项专门任务的程序，例如防病毒程序、文件压缩程序或数据备份程序？
- 1.11 字处理程序、电子表格程序、电子邮件程序、Web 浏览器和游戏属于哪一类软件？

1.3 计算机如何存储数据

概念：计算机中存储的所有数据都转换成 0 和 1 的序列。

计算机的内存由许多小的存储单元构成，这些单元称为**字节**（byte）。一个字节足以容纳字母表中的一个字母或者一个小的数字。计算机要想做任何有意义的事情，就必须能存储大量字节。大多数计算机内存都能存储数十乃至数百亿字节。

每个字节都包含 8 个更小的存储单元，这些单元称为**位或比特**（bit）。在英语中，bit 一词来源于 binary digit，即**二进制位**。计算机科学家通常将位看成是一些小开关，它们可能处于开或关的状态。然而，位并不是真正的“开关”，至少不是传统意义上的那种。在大多数计算机系统中，可以将“位”理解成一种极其微小的电子零件，它既能容纳一个正电荷，也能容纳一个负电荷。计算机科学家认为正电荷是处于“开”位置的开关，认为负电荷是处于“关”位置的开关。图 1-6 展示了在计算机科学家眼中，一个字节的内存是什么样子的：它包含了多个小开关，每个都可以扳至开或关位置。

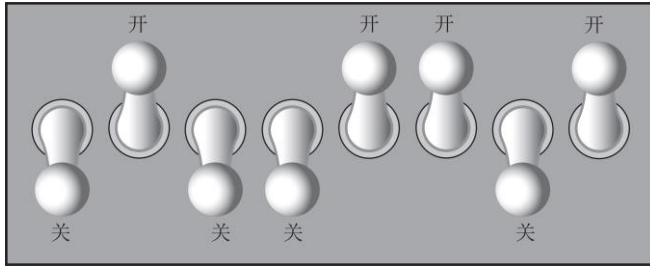
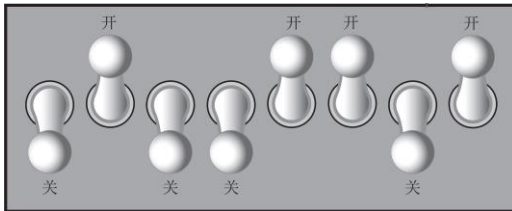
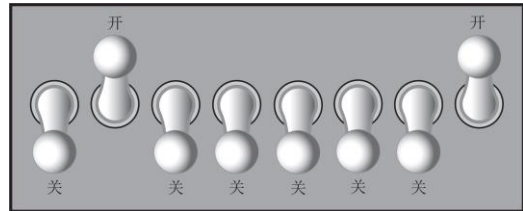


图 1-6 想象一个字节由 8 个开关组成

数据用一个字节来存储时，计算机将 8 个位设为开或关状态，以一种特定的模式来表示该数据。例如，图 1-7 左图的模式显示了如何将数字 77 存储到一个字节中，右图的模式则显示了如何将字母 A 存储到一个字节中。我们不久就会介绍具体如何确定这些模式。



用一个字节来存储数字 77



用一个字节来存储字母 A

图 1-7 数字 77 和字母 A 的位模式

1.3.1 存储数字

二进制位只能以一种非常有限的方式表示数字。取决于它是开还是关，它能表示两个不同的值之一。在计算机系统中，处于关闭位置的位表示数字 0，而处于打开位置的表示 1。这完美地对应于**二进制数字系统**（binary numbering system）。在二进制数字系统（简称二进制）中，所有数值都写成 0 和 1 的序列。以下是用二进制表示的一个数字：

10011101

二进制数字的每个位置都分配了一个值，这称为**位置值或位值**（position values）。如图 1-8 所示，从右向左对应的位置值分别是 2^0 ， 2^1 ， 2^2 ， 2^3 ……等等。图 1-9 展示了同一幅图将位置值（十进制）计算好之后的样子。从右向左，位置值分别是 1，2，4，8……等等。

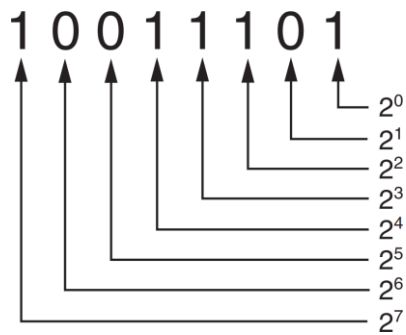


图 1-8 和二进制位对应的位置值是 2 的次方

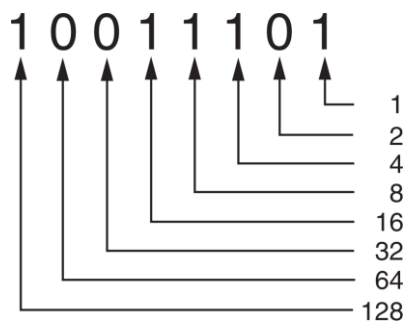


图 1-9 计算好的十进制位置值

为了确定一个二进制数字的值，只需将所有 1 的位置值加起来。例如，在二进制数字 10011101 中，所有 1 的位置值是 1，4，8，16 和 128，如图 1-10 所示。所有这些位置值之和是 157，因此二进制数字 10011101 的（十进制）值是 157。

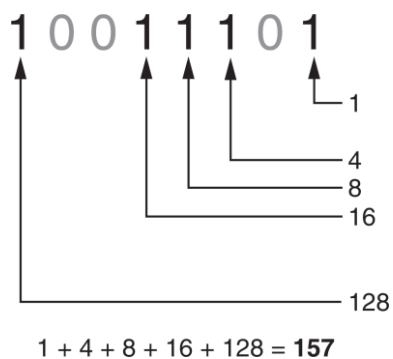


图 1-10 确定 10011101 的（十进制）值

图 1-11 展示了 157 在一个字节的内存中是如何存储的。每个 1 都用处于“开”位置的一个位来表示，而每个 0 都用处于“关”位置的一个位来表示。

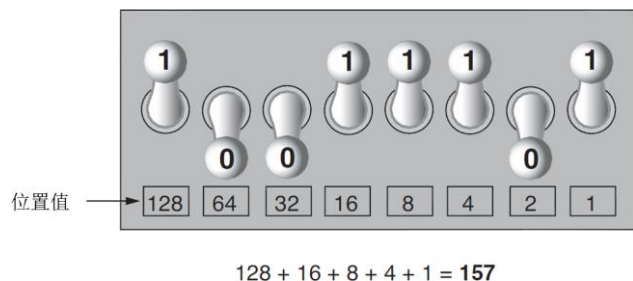


图 1-11 157 的位模式

如果一个字节的所有二进制位都设为 0（关），该字节的值为 0。如果一个字节中的所有二进制位都设为 1（开），该字节容纳的是它能表示的最大值。能用一个字节存储的最大值是 $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255$ 。之所以存在这个限制，是因为一个字节总共只有 8 位。

如果需要存储比 255 大的数字该怎么办？答案很简单：使用多个字节。例如，将两个字节合到一起就有 16 位。这 16 位的位置值是 $2^0, 2^1, 2^2, 2^3 \dots 2^{15}$ 。如图 1-12 所示，用两个字节能存储的最大值就是 65 535。如果需要存储比这还要大的数，使用更多的字节即可。

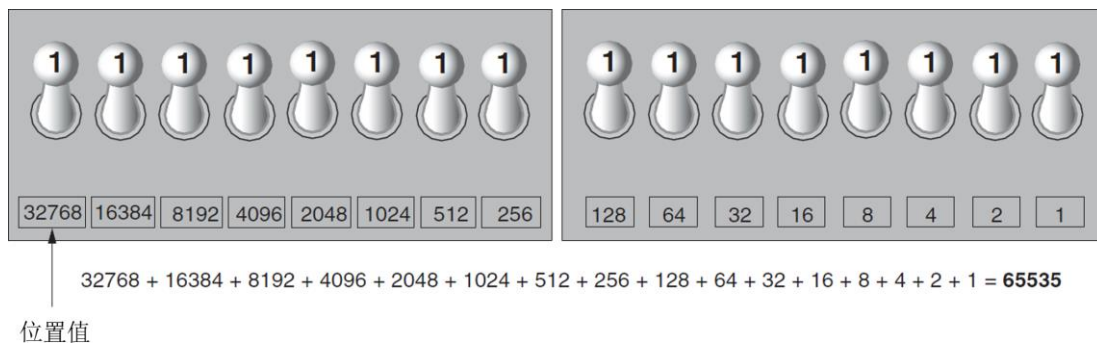


图 1-12 用两个字节存储一个较大的数

提示：如果觉得所有这些过于繁琐，请放松！编程时不需要亲自将数字转换成二进制。但是，知道计算机内部发生的事情，有助于你加深理解并成为一名更好的程序员。

1.3.2 存储字符

在计算机内存中，任何数据都必须以二进制形式存储。其中包括字符，例如字母和标点符号。将字符存储到内存时，它首先要转换成一个数值编码。然后，数值编码以二进制形式来存储。

多年来，人们开发了多种编码方案来表示计算机内存中的字符。其中最重要的是 ASCII，它是“美国信息交换标准代码”的简称，即 American Standard Code for Information Interchange。ASCII 包含 128 个数值编码，可以表示英语字母、各种标点符号以及其他字符。例如，大写字母 A 的 ASCII 编码是 65。用计算机键盘输入大写字母 A 时，实际会将数字 65 存储到内存中（当然是以二进制形式）。图 1-13 对此进行了演示。

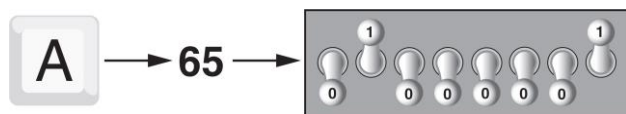


图 1-13 字母 A 作为数字 65 存储到内存中

 提示：ASCII 的发音是“askee”。

类似地，大写字母 B 的 ASCII 编码是 66，C 是 67……以此类推。附录 C 列出了所有 ASCII 编码及其所代表的字符。

ASCII 字符集是上个世纪 60 年代初开发的，得到了所有计算机制造商的采纳。但是，ASCII 也是极其有限的，因为它只定义了 128 个字符码。作为补救，人们又在 90 年代初开发了 Unicode 字符集。Unicode 是一套全面的编码方案，不仅兼容 ASCII，还能表示世界上大多数语言的全套文字。如今，Unicode 正在快速成为计算机工业使用的标准字符集。

1.3.3 高级数字存储

之前讨论了数字以及如何内存中存储它们。你或许会以为，二进制数字系统只能用来表示从 0 开始的正整数。确实，负数和实数（例如 3.14159265）不能用我们讨论的简单二进制编码技术来表示。

事实上，计算机确实能在内存中存储负数和实数，但为了做到这一点，要在使用二进制数字系统的同时采用某种编码方案。负数采用的编码方案称为“2 的补码”，实数采用的则是“浮点表示法”。你不需要知道这些编码方案具体是如何工作的，只需知道它们被用来将负数和实数转换成二进制形式。

1.3.4 其他类型的数据

计算机一般称为数字设备。可用**数字**（digital）一词来描述使用二进制值的任何东西。**数字数据**（digital data）是指以二进制存储的数据，而**数字设备**（digital device）是指处理二进制数据的任何设备。我们之前讨论了如何以二进制形式存储数字和字符，但计算机还能处理其他许多类型的数字数据。

以数码相机拍摄的照片为例。这种图像由名为**像素**（pixel）的彩色小点构成（像素是“图像元素”的简称）。如图 1-14 所示，图像中的每个像素都转换成代表颜色的一个数值编码。数值编码以二进制形式存储。

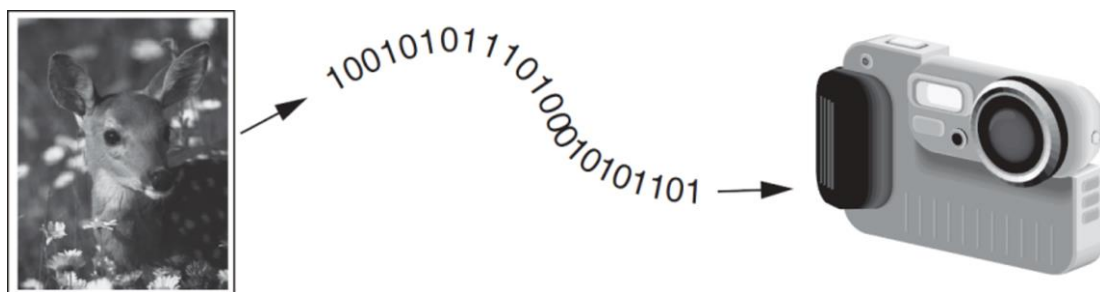


图 1-14 以二进制形式存储的数码照片

从网上或者音乐播放器播放的音乐也是数字的。一首数字歌曲分解成许多小的片段，这些片段称为**样本**（samples）。一首歌被分解成的样本数量越多（这个过程称为采样），播放时的保真度越高。例如，CD 音质的歌曲每秒要分解成 44000 个样本！

检查点

- 1.12 多大内存足以存储一个字母或者一个小的数字？
- 1.13 你将能设为“开”或“关”的小“开关”称为什么？
- 1.14 在什么数字系统中，所有数值都写成 0 和 1 的序列？
- 1.15 ASCII 的作用是什么？
- 1.16 什么编码方案是一套全面的编码方案，能表示世界上大多数语言的全套文字？
- 1.17 “数字数据”和“数字设备”是什么意思？

1.4 程序如何工作

概念：计算机 CPU 只理解以机器语言写成的指令。由于人们发现很难整个程序都用机器语言编写，所以发明了更高级的编程语言。

之前说过，CPU 是计算机最重要的组件，因为它是计算机中负责运行程序的那一部分。人们有时说 CPU 是计算机的“大脑”，有时还会说它很“聪明”。虽然经常都能看到这样或那样的比喻，但你应当理解的是，CPU 不是大脑，也并不聪明。CPU 是设计用来做特定事情的一种电子设备。具体地说，CPU 被设计用来执行以下操作：

- 从主存读取数据
- 两数相加
- 两相相减
- 两数相乘
- 两数相除
- 将数据从一个内存位置移到另一个位置
- 判断一个值是否等于另一个值
-

从这个列表可以看出，CPU 执行的是对数据的简单操作。但 CPU 自己做不了任何事情。必须告诉它要做什么，而那正是程序的作用。程序其实就是告诉 CPU 要做什么的指令清单。

程序中的每个指令都告诉 CPU 执行一项具体的操作。下面是可能在程序中出现的一个指令的例子：

10110000

对我们来说，这只是一系列的 0 和 1。但对 CPU 来说，它是执行一项操作的指令^①。之所以写成 0 和 1 的序列，是因为 CPU 只能理解用**机器语言**写的指令，而机器语言指令总是以二进制来写的。

CPU 能执行的每项操作都有一条对应的机器语言指令。例如，两数相加有一条指令，两数相减有另一条指令。CPU 能执行的全套指令称为 CPU 的**指令集**（instruction set）。



注意：当前有多家微处理器厂商都在制造 CPU，其中较著名的有 Intel、AMD、高通和

^① 本例是 Intel 微处理器的一条真实的指令，作用是将一个值移动到 CPU 中。

苹果等。检查你的计算机的配置，就知道它使用的是哪个厂商的 CPU。

每种品牌的微处理器通常有自己的一套特殊的指令集，其中的指令只有同品牌的微处理器才能理解。例如，Intel 微处理器就不能直接理解苹果微处理器的指令。

前面只展示了一个机器语言指令。然而，计算机要想做任何有意义的事情，执行的指令远不止一个。CPU 执行的每个操作在本质上是非常基本的（或者说，是最简单的）。所以，CPU 只有执行大量指令，才能真正完成有意义的任务。例如，为了计算存款账户今年的利息是多少，CPU 必须按照正确顺序执行大量指令。一个程序包含数千乃至数百万条机器语言指令是再正常不过的事情。

程序通常存储在像磁盘驱动器这样的辅助存储设备中。在计算机上安装程序时，程序通常从网上下载，或者从某个应用商店安装。

虽然程序能存储在像硬盘驱动器这样的辅助存储设备中，但 CPU 每次执行它时，都必须把它拷贝到主存（RAM）。例如，假定硬盘上有一个字处理程序。执行它需要用鼠标双击它的图标。这会导致程序从硬盘拷贝到 RAM。然后，CPU 执行 RAM 中的这个程序的拷贝。图 1-15 演示了这个过程。

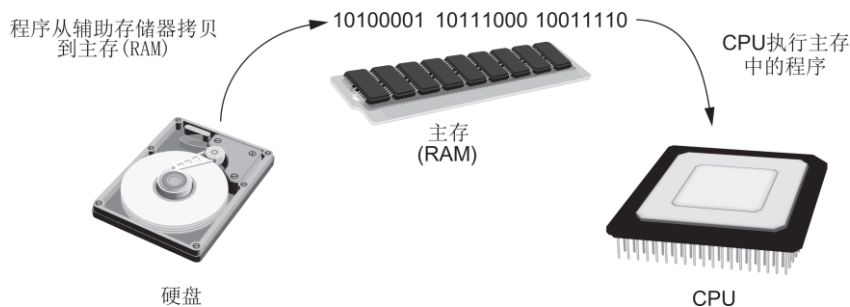


图 1-15 程序拷贝到 RAM 并执行

CPU 执行程序中的指令时，会经历一个“取回 - 解码 - 执行”（fetch-decode-execute）周期。程序中的每条指令都会重复一遍这个周期。这个周期由 3 个步骤构成：

- **取回/取指** 每个程序都包含大量机器语言指令。周期的第一步是取回（或称读取）内存中的下一条指令，并把它送入 CPU。这个“取回指令”的步骤也称为“取指”。
- **解码** 机器语言指令是二进制值，它指示 CPU 执行一个具体的操作。在这一步中，CPU 对刚才取回的指令进行解码，判断应该执行什么操作。
- **执行** 周期的最后一步是执行指定的操作。

图 1-16 演示了这些步骤。

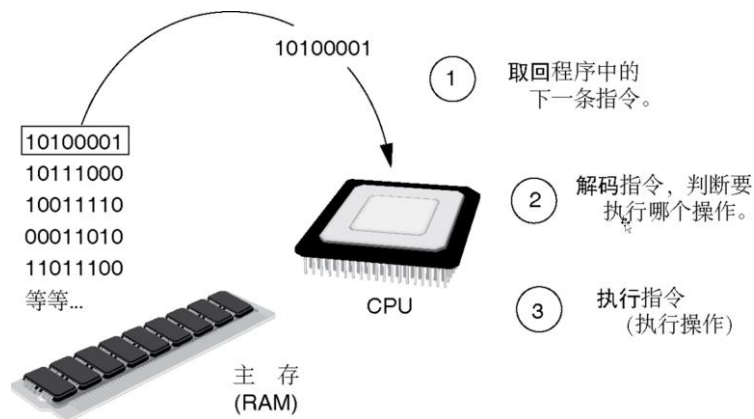



图 1-16 “取回 - 解码 - 执行”周期

1.4.1 从机器语言到汇编语言

计算机只能执行用机器语言写的程序。如前所述，一个程序可能有几千、几百万甚至更多的二进制指令，写这样的程序非常繁琐，而且会花非常多的时间。用机器语言编程还非常困难，因为一个 0 或 1 的位置有误，就会造成错误。

虽然计算机的 CPU 只能理解机器语言，但要求人们用机器语言写程序是不实际的。有鉴于此，在计算的早期岁月，人们创建了**汇编语言**（assembly language）来代替机器语言^①。汇编语言不是用二进制数字来写指令，而是使用称为**助记符**（mnemonics）的一些短字。例如，在汇编语言中，助记符 `add` 执行加法，`mul` 通常执行乘法，而 `mov` 将值移动到某个内存位置。用汇编语言写程序时，可以用简短的助记符来代替二进制数字。

 注意：汇编语言有许多版本。之前说过，每种品牌的 CPU 都有自己的机器语言指令集。每种品牌的 CPU 通常也有自己的汇编语言。

然而，汇编语言程序不能由 CPU 直接执行。CPU 只理解机器语言，所以要用称为**汇编器**（assembler）的特殊程序将汇编语言程序转换成机器语言程序。图 1-17 展示了这个过程。随后，汇编器创建的机器语言程序就可以由 CPU 执行了。

^① 人们创建的第一个汇编语言可能是 20 世纪 40 年代在剑桥大学为著名的 EDSAC 计算机开发的。

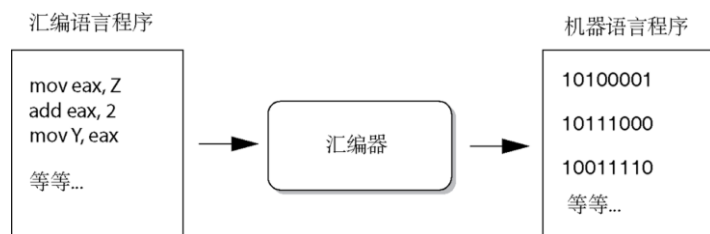


图 1-17 汇编器将汇编语言程序转换成机器语言程序

1.4.2 高级语言

虽然汇编语言使人们不再需要写二进制机器语言指令，但并不是说就不麻烦了。汇编语言基本上是机器语言的一种直接替代，而且和机器语言相似，也要求掌握很多关于 CPU 的知识。即便是最简单的程序，汇编语言也要求写大量指令。由于汇编语言在本质上与机器语言如此相似，所以人们认为它还是一种**低级语言**。

20 世纪 50 年代，新一代的**高级语言**出现了。高级语言允许创建强大和复杂的程序，同时不需要知道 CPU 是如何工作的，也不需要写大量低级指令。此外，大多数高级语言都使用易于理解的单词。例如在 COBOL 语言（50 年代创建的早期高级语言的一种）中，可以使用以下指令在屏幕上显示消息“Hello world”：

```
DISPLAY "Hello world"
```

作为一种现代的高级编程语言，本书使用的 Python 为了显示同样的消息只需执行以下指令：

```
print('Hello world')
```

但是，如果用汇编语言做同样的事情，则需要好几条指令，而且要深入掌握 CPU 如何与计算机的输出设备交互。从这个例子可以看出，高级语言允许程序员将精力集中在他们要用程序执行的任务上，而不必纠缠于 CPU 执行程序细节。

从 50 年代起，人们创建了数以千计的高级语言。表 1-1 列出了其中比较著名的。

表 1-1 编程语言

语言	说明
Ada	Ada 是 20 世纪 70 年代创建的，主要用于编写美国国防部使用的应用程序。Ada 这个名称是为了纪念阿达·奥古斯塔（Ada Lovelace）伯爵夫人，她是计算领域的一名著名的历史人物。
BASIC	BASIC 是“初学者通用符号指令代码”（Beginners All-purpose Symbolic

	Instruction Code) 的简称, 是 20 世纪 60 年代初问世的一种常规用途的语言, 初学者很容易上手。今天, BASIC 有许多不同的版本。
FORTRAN	FORTRAN 是“公式翻译语言”(FORmula TRANslator) 的简称, 是第一代高级编程语言。它设计于 20 世纪 50 年代, 用于执行复杂的数学计算。
COBOL	COBOL 是“面向商业的通用语言”(Common Business-Oriented Language) 的简称, 创建于 20 世纪 50 年代, 主要为商业应用程序而设计。
Pascal	Pascal 创建于 1970 年, 最早是作为一种教学语言而设计的。语言的名称是为了纪念数学家、物理学家以及哲学家布莱士·帕斯卡 (Blaise Pascal)。
C 和 C++	C 和 C++ (发音是“c plus plus”) 是贝尔实验室开发的两种功能强大的、常规用途的语言。C 语言创建于 1972 年。以 C 为基础的 C++ 创建于 1983 年。
C#	发音是“c sharp”。该语言由 Microsoft 于 2000 年左右创建, 用于开发基于 Microsoft .NET 平台的应用程序。
Java	Java 由 Sun Microsystems 于 20 世纪 90 年代初创建。用它开发的程序可以在单独一台计算机上运行, 也可以通过 Internet 从一台 Web 服务器上运行。
JavaScript	JavaScript 创建于 20 世纪 90 年代, 可以在网页中使用。虽然名字里面有个 Java, 但和 Java 没有关系。
Python	Python 是 20 世纪 90 年代初创建的一种常规用途的语言。在商业、学术和人工智能应用程序中比较流行。
Ruby	Ruby 是 20 世纪 90 年代创建的一种常规用途的语言。越来越多的人用它编写在 Web 服务器上运行的程序。
Rust	Rust 编程语言为高性能、内存安全和并发执行而设计。它于 2010 年由 Mozilla Research 发布。
Visual Basic	Visual Basic (简称 VB) 是 Microsoft 的一种编程语言和软件开发环境, 允许程序员快速创建基于 Windows 的应用程序。VB 在 20 世纪 90 年代初创建。

1.4.3 关键字、操作符和语法：概述

每种高级语言都有自己的一套预定义单词, 程序员必须用这些单词来写程序。这些预定义的单词称为**关键字** (keywords) 或**保留字** (reserved words)。每个关键字都具有特定含义, 不能用作其他用途。表 1-2 展示了本书使用的 Python 编程语言的所有关键字。

表 1-2 Python 关键字

and	continue	finally	is	raise
as	def	For	lambda	return
assert	del	From	None	True
async	elif	global	nonlocal	try
await	else	If	not	while
break	except	import	or	with
class	False	In	pass	yield

除了关键字，编程语言还通过一系列**操作符**（operator）^①对数据执行各种运算（操作）。例如，所有编程语言都提供了执行算术运算的各种数学操作符。C++和其他大多数语言一样，加号（+）执行两数相加。以下代码将 12 和 75 相加：

```
12 + 75
```

Python 语言还支持其他大量操作符，随着学习的深入，你会逐渐接触到它们。

除了关键字和操作符，每种语言还有自己的语法。**语法**（syntax）是一套必须严格遵守的编程规则，规定了如何在程序中使用关键字、操作符和各种标点符号。学习一种编程语言必须先掌握其语法。

用高级语言写程序时，你使用的单独的指令称为语句。**语句**（statements）可由关键字、操作符、标点符号以及其他允许的编程元素构成，它们按正确顺序排列以执行一个操作。

1.4.4 编译器和解释器

由于 CPU 只理解机器语言指令，所以用高级语言写的程序必须转换（翻译）成机器语言。取决于具体使用的是什么语言，程序员可以使用编译器或解释器来执行转换。

编译器（compiler）是一种特殊程序，能将使用高级语言写成的程序转换成一个单独的机器语言程序。然后，任何时候都可以直接执行机器语言程序。图 1-18 对此进行了演示。从图中可以看出，编译和执行是两个不同的过程。

Python 使用的则是**解释器**（interpreter），它既能转换也能执行高级语言程序中的指令。解释器每次读取程序中的一条指令，就会把它转换成机器语言指令，并马上执行。这个过程

^① 译注：operator 在本书统一为“操作符”而不是“运算符”。operand 则统一为“操作数”。

一直重复，直到处理完程序中的每一条指令。图 1-19 演示了这个过程。由于解释器是一边转换一边执行的，所以通常不会再创建单独的机器语言程序。

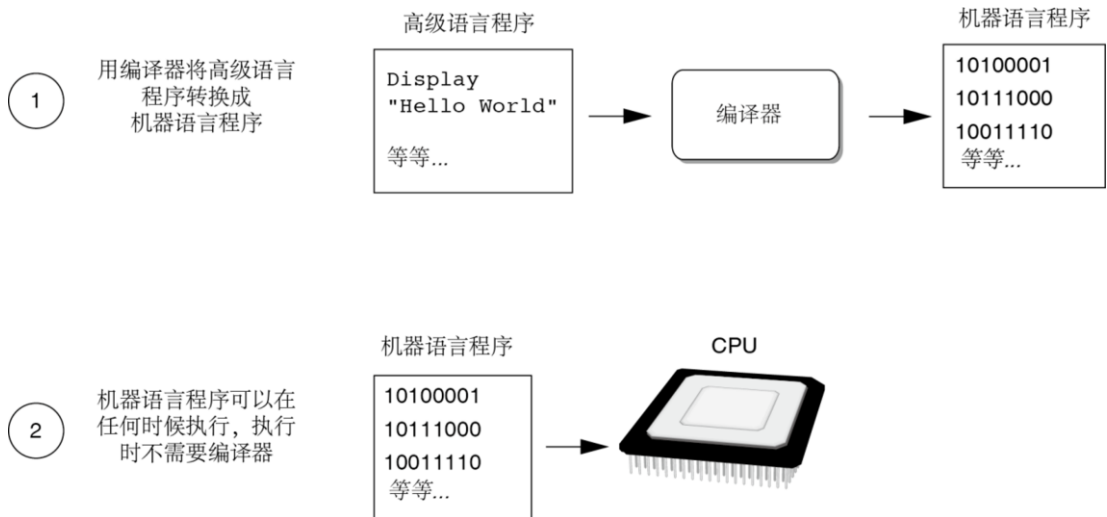


图 1-18 编译高级程序并执行它

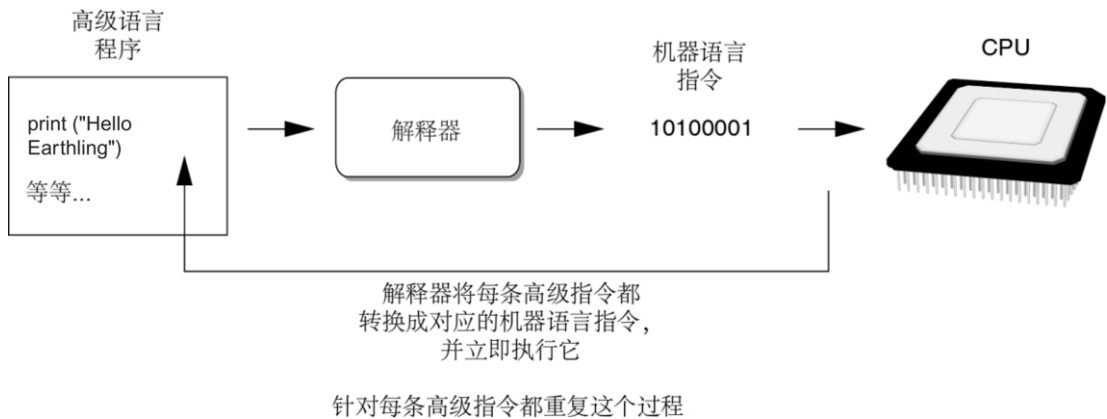


图 1-19 用解释器执行高级程序

在程序中，用高级语言写的一系列语句称为**源代码**（source code）或者简称为**代码**（code）。通常，程序员在文本编辑器中键入代码，并将代码存储到计算机磁盘上的文件中。接着，程序员用编译器将代码转换成机器语言程序，或者用解释器一边转换一边执行代码。但是，如果代码包含语法错误，转换就会失败。**语法错误**（syntax error）是指写程序时犯的错误，比如关键字拼写错误、遗漏标点符号或者不正确地使用了操作符等。在这个时候，编译器

或解释器会显示错误消息，指出程序中含有语法错误。程序员要纠正错误，并重新转换程序。



注意：人类语言也有语法规则。你还记得自己上第一堂英语课的情形吗？你要学习逗号、撇号、大写等等的使用规则。那时学习的就是这种语言的语法。

虽然在平时说和写的时候，即使稍微违反这些语法规则，别人一般也听得懂，但编译器和解释器无此本事。程序即使只有一处小小的语法错误，也不能成功编译或执行。

检查点

- 1.18 CPU 只理解用哪种语言写的指令？
- 1.19 CPU 每次执行程序时，必须把它拷贝到什么类型的内存中？
- 1.20 CPU 执行程序中的指令时，会经历一个什么周期？
- 1.21 什么是汇编语言？
- 1.22 什么类型的编程语言允许创建强大和复杂的应用程序，同时不必知道 CPU 是如何工作的？
- 1.23 每种语言都有一套在写程序时必须严格遵守的规则。这套规则叫什么？
- 1.24 一种程序能将高级语言程序转换成独立的机器语言程序，这种程序叫什么？
- 1.25 一种程序能一边转换一边执行高级语言程序中的指令，这种程序叫什么？
- 1.26 关键字拼写错误，遗漏了标点符号，或者不正确地使用了操作符，这些称为什么错误？

1.5 使用 Python

概念：Python 解释器可以运行作为文件保存的 Python 程序，也可以交互式执行通过键盘输入的 Python 语句。Python 自带一个名为 IDLE 的程序，它简化了程序的编写、执行和测试。

1.5.1 安装 Python

在你尝试本书的任何程序，或者自己编写任何程序之前，需要确保 Python 已经安装在计算机上并进行了正确的配置。如果是在学校的计算机实验室操作，这一步可能已经完成了。如果是使用自己的计算机，可以按照附录 A 的说明下载并安装 Python。

1.5.2 Python 解释器

之前说过，Python 是一种解释型语言。在计算机上安装 Python 语言时，安装的其中一项就是 Python 解释器。**Python 解释器**是一个可以读取 Python 语句并执行它们的程序。（有的时候，我们会把 Python 解释器简称为解释器）。

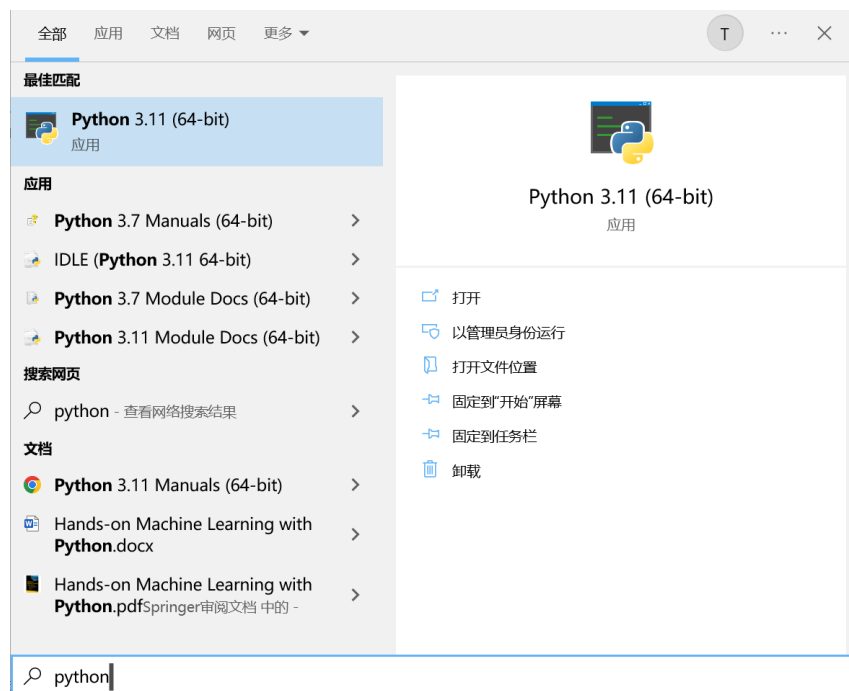
可以在两种模式下使用解释器：交互模式和脚本模式。在**交互模式**下，解释器等待你在键盘上输入 Python 语句。一旦输入一个语句，解释器就会执行它，然后等待你输入另一个语句。在**脚本模式**下，解释器读取包含 Python 语句的一个文件的内容。这样的文件称为 Python 程序或 Python 脚本。解释器在读取 Python 程序时执行其中的每个语句。

交互模式

在系统上安装并配置好 Python 后，可以进入操作系统的命令行界面，并输入以下命令：

```
python
```

在 Windows 平台上，还可以打开 Windows 搜索框并输入 python。在搜索结果中，会看到像“Python 3.11”这样的应用程序名称（“3.11”代表当前安装的 Python 版本）。在搜索结果中单击这一项，从而以交互模式启动 Python 解释器。





注意：Python 以交互模式运行时，它通常称为 Python shell（Python 外壳程序）。

Python 解释器以交互模式启动后，会在控制台窗口显示如下所示的消息：

```
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37)
[MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

>>>提示符表明解释器正在等待输入一个 Python 语句。现在来尝试一下。用 Python 能做的最简单的事情之一就是在屏幕上打印消息。例如，以下语句打印消息“Python 编程真有趣!”：

```
print('Python 编程真有趣!')
```

可以把它看成是发送给 Python 解释器的一个指令。如果严格照此输入该语句，屏幕上就会打印“Python 编程真有趣!”。下面是在解释器提示符下输入该语句的例子：

```
>>> print('Python 编程真有趣!') 
```

输入语句后按 Enter 键，Python 解释器就会执行语句，如下所示：

```
>>> print('Python 编程真有趣!') 
Python 编程真有趣!
>>>
```

消息显示完毕后，会再次出现>>>提示符，表明解释器正在等待继续输入下一个语句。下面来看另一个例子。在以下示例会话中，我们输入了两个语句：

```
>>> print('生存还是毁灭') 
生存还是毁灭
>>> print('这是一个值得考虑的问题') 
这是一个值得考虑的问题
>>>
```

如果在交互模式下输入的语句有误，解释器会显示一条错误消息。所以，在学习 Python 期间，交互模式是非常有用的。学习 Python 语言的新特性时，可以在交互模式下进行多次尝试，并即时从解释器获得反馈。

要在交互模式下退出 Python 解释器，在 Windows 计算机上，按 Ctrl-Z 组合键，然后按 Enter。在 Mac、Linux 或 UNIX 计算机上，则按 Ctrl-D。



注意：第 2 章将详细讨论语句。如果现在就想在交互模式中尝试它们，请务必严格照着例子输入。

编写 Python 程序并在脚本模式下运行

虽然交互模式在测试代码时很有用，但在交互模式中输入的语句不会作为一个程序保存下来，就是简单地执行并显示结果而已。如果希望将一组 Python 语句变成完整的程序，那么需要一个文件来保存这些语句。以后想执行该程序时，需要以脚本（script）模式使用 Python 解释器。

例如，假定要写一个 Python 程序来显示以下三行文本：

```
Nudge nudge
Wink wink
Know what I mean?
```

用一个简单的文本编辑器就可以写程序，例如所有 Windows 计算机自带的“记事本”程序。创建一个包含以下语句的文件：

```
print('Nudge nudge')
print('Wink wink')
print('Know what I mean?')
```



注意：可以使用一个字处理软件来创建 Python 程序，但务必将该程序另存为纯文本文件。否则，Python 解释器将无法读取它的内容。

保存 Python 程序文件时，最好使用.py 作为扩展名，代表这是一个 Python 程序。例如，可将上述语句保存到一个名为 test.py 的文件中。为了运行程序，在操作系统的命令行中切换到保存该文件的目录，然后执行以下命令：

```
python test.py
```

这样就会以脚本模式启动 Python 解释器并执行 test.py 中的语句。程序运行完毕后，Python 解释器会自动退出。

1.5.3 IDLE 编程环境



视频讲解： Using Interactive Mode in IDLE

前几节描述了如何在操作系统的命令行上以交互模式或脚本模式启动 Python 解释器。作为另一种选择，还可以使用某个集成开发环境，它整合了编写、执行和测试程序所需的全部工具。

Python 最近的版本都自带一个名为 IDLE 的程序，会在安装 Python 时自动安装。IDLE 的

全称是“Integrated DeveLopment Environment”，即“集成开发环境”。运行 IDLE 时^①，会出现如图 1-20 所示的窗口。注意 IDLE 窗口中出现的>>>提示符，它表明解释器当前在交互模式下运行。可以在这个提示符下输入 Python 语句，并在 IDLE 窗口中观察执行结果。

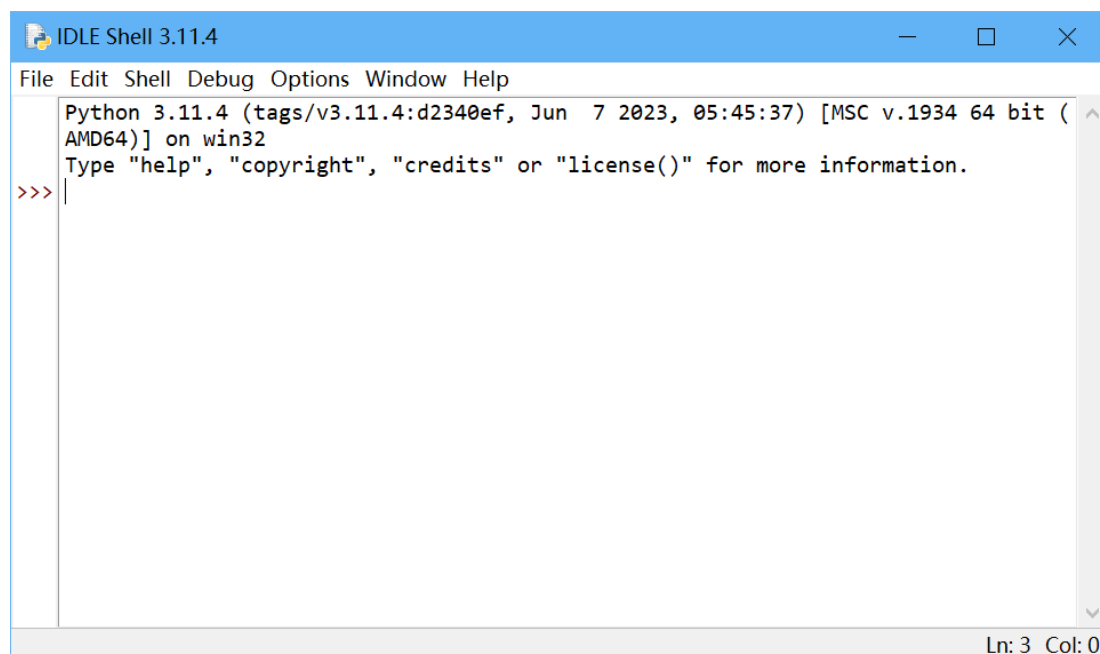



图 1-20 IDLE

IDLE 还内置了一个文本编辑器，是专门为了方便写 Python 程序而设计的。例如，IDLE 编辑器能对代码进行“彩色语法标注”，使关键字和程序的其他部分以不同的颜色显示，这显著增强了程序的可读性。可以在 IDLE 中编写程序代码，保存到磁盘，并执行最终的 Python 长夜难明。附录 B 简单介绍了 IDLE，指导你完成创建、保存和执行 Python 程序的过程。

 注意：虽然可以使用 Python 自带的 IDLE，但还有其他几个流行的 Python IDE 可供选择，其中包括 PyCharm 和 VS Code 等。在课堂上，你的老师可能会指定一种。

复习题

选择题

^① 译注：在 Windows 中打开搜索框，输入 IDLE 即可看到该程序。

-
1. _____是一套特殊的指令，计算机遵照这些指令来执行任务。
 - a. 编译器
 - b. 程序
 - c. 解释器
 - d. 编程语言
 2. 构成计算机的物理设备称为_____。
 - a. 硬件
 - b. 软件
 - c. 操作系统
 - d. 工具
 3. 计算机负责运行程序的组件称为_____。
 - a. RAM
 - b. 辅助存储
 - c. 主存
 - d. CPU
 4. 今天，CPU 是称为_____的小型芯片。
 - a. ENIAC
 - b. 微处理器
 - c. 内存条
 - d. 操作系统
 5. 计算机在_____中存储正在运行的程序以及程序处理的数据。
 - a. 辅助存储
 - b. CPU
 - c. 主存
 - d. 微处理器
 6. 有一种易失性的存储器，它只能在程序运行时进行临时存储。这种存储器称为_____。
 - a. RAM
 - b. 辅助存储
 - c. 磁盘驱动器
 - d. U 盘
 7. 有一种类型的存储设备能长时间保存数据——即使在计算机关机之后。这种存储设备称为_____。
 - a. RAM
 - b. 主存
 - c. 辅助存储
 - d. CPU 存储
 8. 从人或设备收集数据并将其发送给计算机的组件称为_____。
 - a. 输出设备
 - b. 输入设备
 - c. 辅助存储设备
 - d. 主存
 9. 显示器是一种_____设备。
 - a. 输出
 - b. 输入
 - c. 辅助存储
 - d. 主存
 10. 一个_____的空间足以存储一个字母或者一个小的数字。
 - a. 字节
 - b. 二进制位
 - c. 开关
 - d. 晶体管
 11. 一个字节由 8 个_____构成。
 - a. CPU
 - b. 指令
 - c. 变量
 - d. 二进制位
 12. 在_____数字系统中，所有数值都写成 0 和 1 的序列。
 - a. 十六进制
 - b. 二进制
 - c. 八进制
 - d. 十进制

13. 处于“关”状态的一个二进制位表示以下值：_____

- a. 1 b. -1 c. 0 d. "no"

14. 能表示英语字母、标点符号和其他字符的一套 128 个数值代码称为_____。

- a. 二进制数字系统 b. ASCII c. Unicode d. ENIAC

15. 可表示世界上大多数语言中的全套文字的编码方案是_____。

- a. 二进制数字系统 b. ASCII c. Unicode d. ENIAC

16. 负数用_____技术来编码。

- a. 2 的补码 b. 浮点 c. ASCII d. Unicode

17. 实数用_____技术来编码。

- a. 2 的补数 b. 浮点 c. ASCII d. Unicode

18. 构成数字图像的彩色小点称为_____。

- a. 二进制位 b. 字节 c. 颜色包 d. 像素

19. 查看机器语言程序，看到的是_____。

- a. Python 代码 b. 二进制数字流 c. 英语单词 d. 电路

20. CPU 在“取回 - 解码 - 执行”过程的哪个阶段决定自己要执行的操作？

- a. 取回 b. 解码 c. 执行 d. 解构

21. 计算机只能执行用_____写的程序。

- a. Java b. 汇编语言 c. 机器语言 d. Python

22. _____将汇编语言程序转换成机器语言程序。

- a. 汇编器 b. 编译器 c. 翻译器 d. 解释器

23. 构成高级编程语言的单词称为_____。

- a. 二进制指令 b. 助记符 c. 命令 d. 关键字

24. 写程序必须遵守的规则称为_____。

- a. 语法 b. 标点 c. 关键字 d. 操作符

25. _____程序将高级语言程序转换成独立的机器语言程序。

a. 汇编器 b. 编译器 c. 翻译器 d. 实用程序

判断题

1. 今天的 CPU 是由电子和机械组件（比如电子管和开关）组成的巨型设备。
2. 主存也称为 RAM。
3. 计算机内存中存储的任何数据都必须以二进制数形式存储。
4. 图像（比如用数码相机拍摄的照片）不能以二进制数形式存储。
5. 机器语言是 CPU 唯一能理解的语言。
6. 汇编语言被认为是高级语言。
7. 解释器是一种特殊的程序，能一边转换一边执行用高级语言写的程序中的指令。
8. 语法错误无碍程序的成功编译和执行。
9. Windows, Linux, Android, iOS 和 macOS 都是应用软件（application software）的例子。
10. 字处理程序、电子表格程序、电子邮件程序、Web 浏览器和游戏都是实用程序（utility program）的例子。

简答题

1. 为什么 CPU 是计算机最重要的组件？
2. 处于“开”状态的二进制位代表什么数字？处于“关”状态的位代表什么数字？
3. 处理二进制数据的设备叫做什么设备？
4. 构成高级编程语言的单词叫做什么？
5. 汇编语言中使用的短字叫做什么？
6. 编译器和解释器的区别是什么？
7. 什么类型的软件负责控制计算机硬件的内部运作？

练习题

1. 为了确定能与 Python 解释器进行交互，在你的计算机上尝试执行以下步骤：
 - 以交互模式启动 Python 解释器。
 - 在 >>> 提示符下，输入以下语句，然后按 Enter 键。

```
print('这是对 Python 解释器的测试。') Enter
```

- 按 Enter 键后，解释器将执行该语句。如果输入的内容正确，会话应该是这样的：

```
>>> print('这是对 Python 解释器的测试。') Enter  
这是对 Python 解释器的测试。  
>>>
```

- 如果显示一条错误消息，请重新输入语句，并确保严格照着例子输入。
- 退出 Python 解释器（在 Windows 中，先按 Ctrl-Z 再按 Enter。在其他系统中，直接按 Ctrl-D）。

2. 为了确保能与 IDLE 交互，在你的计算机上尝试执行以下步骤：

视频讲解：Performing Exercise 2

- 启动 IDLE。在 Windows 搜索框中输入 IDLE。单击搜索结果中出现的 IDLE 桌面应用。
- IDLE 启动后，应出现如图 1-20 所示的窗口。在 >>> 提示符下输入以下语句，然后按 Enter 键。

```
print('这是对 IDLE 的测试。。') Enter
```

- 按 Enter 键后，Python 解释器将执行该语句。如果输入的内容正确，会话应该是这样的：

```
>>> print('This is a test of IDLE.') Enter  
这是对 IDLE 的测试。  
>>>
```

- 如果显示一条错误消息，请重新输入语句，并确保完全照着例子输入。
- 退出 Python 解释器（在 Windows 中，先按 Ctrl-Z 再按 Enter。在其他系统中，直接按 Ctrl-D）。
- 单击菜单栏上的 File，再单击 Exit（或者按 Ctrl-Q）退出 IDLE。

3. 利用你在本章学到的关于二进制的知识，将下列十进制数字转换成二进制。

```
11  
65  
100  
255
```

4. 利用在本章学到的二进制数字系统的知识，将以下二进制数转换成十进制：

```
1101  
1000  
101011
```

5. 查询附录 C 的 ASCII 字符集，确定你的英文名字中的每个字母的代码。

6. 在网上研究一下 Python 编程语言的历史，并回答以下问题：

- 谁是 Python 的创始人？
- Python 是什么时候发明的？
- 在 Python 编程社区，人们亲切地将 Python 的创始人被称为“BDFL”。这是什么意思？

第 2 章 输入、处理和输出

2.1 设计程序

概念：程序在编写之前必须经过仔细的设计。在设计过程中，程序员使用伪代码和流程图等工具对程序进行建模。

2.1.1 程序开发周期

在第 1 章中，你了解到程序员通常使用 Python 等高级语言来创建程序。然而，创建程序并不单单是编写代码。为了创建一个能正确工作的程序，整个过程通常需要经历如图 2-1 所示的 5 个阶段。整个过程称为**程序开发周期**（program development cycle）。



图 2-1 程序开发周期

让我们仔细看看这个周期中的每个阶段。

- 1. 设计程序。**所有专业程序员都会告诉你，在上手写代码之前，应该先仔细设计好一个程序。程序员开始一个新项目时，不要直接开始编码。相反，应该先从创建程序的设计开始。有几种设计程序的方法，本节稍后会讨论一些可以用来设计 Python 程序的技术。
- 2. 编写代码。**设计好程序后，程序员开始用 Python 这样的高级语言编写代码。第 1 章讲过，每种语言都有自己的一套称为“语法”的规则，在编码时必须遵守。一种语言的语法规则规定了诸如关键字、操作符和标点符号的用法。如果程序员违反了这些规则中的任何一条，就会造成语法错误。
- 3. 纠正语法错误。**如果程序包含语法错误，即使只是一个简单的错误，例如拼写有误的关键词，编译器或解释器也会显示一条错误消息，告诉你出了什么错。几乎所有代码在第一次编写时都会出现这样或那样的语法错误，所以程序员通常会花一些时间来纠正这些错误。一旦所有语法错误和简单的打字错误都得到纠正，程序就可以编译并转换成机器语言程序（或由解释器执行，取决于具体使用的语言）。
- 4. 测试程序。**一旦代码处于可执行状态，就可以对其进行测试，以确定是否存在任何逻辑错误。**逻辑错误**是指不妨碍程序运行，但会导致它产生不正确结果的错误（数学错误是逻辑错误的一种常见来源）。

5. 纠正逻辑错误。如果程序产生不正确的结果，程序员需要对代码进行**调试 (debug)**。这意味着程序员发现并纠正程序中的逻辑错误。在这个过程中，程序员有时会发现必须修改程序的原始设计。在这种情况下，程序开发周期会重新开始，一直持续到程序没有错误为止。

2.1.2 关于设计过程的更多说明

程序的设计过程可以说是这个周期中最重要的部分。可以将一个程序的设计看成是它的基础。如果把房子建在一个糟糕的地基上，那么最终要做大量的工作来修复这个房子。一个程序的设计也应如此看待。程序设计得不好，最终要做大量的工作来修复这个程序！

设计一个程序的过程可以归纳为以下两个步骤：

1. 理解程序要执行的任务。
2. 确定执行该任务必须采取的步骤。

下面来仔细看看这些步骤中的每一个。

2.1.3 理解程序要执行的任务

在确定程序要执行的步骤之前，必须先理解程序要做什么。通常，专业程序员通过直接与客户合作获得这种理解。我们用**客户 (customer)**一词来描述要求你编写程序的个人、团体或组织。这可能是传统意义上的客户，即付钱给你写程序的人。它也可能是你的老板，或者是你公司内部某个部门的经理。不管是谁，客户将依靠你的程序来完成一项重要的任务。

为了理解一个程序要做的事情，程序员通常会和客户会面。在会面过程中，客户将描述程序应该执行的任务，而程序员将提出问题，以发现尽可能多的任务细节。后续会面通常是必不可少的，因为客户很少在初次见面时说清楚他们想要的一切，而程序员经常会想到其他一些具体的问题。

程序员研究从客户那里收集到的信息，并创建一个包含各种软件需求的清单。所谓**软件需求 (software requirement)**，其实就是程序必须执行的任务。一旦客户同意这个需求清单是完整的，程序员就可以进入下一个阶段。^①



提示：如果你选择成为一名专业的软件开发人员，“客户”是任何要求你写程序的

^① 译注：参见清华大学出版社 2023 年精译版的《软件需求》第三版和《软件需求精要》，更多地了解软件需求这一主题。注意，软件需求作为一个工程，主要应该由业务分析师 (BA) 来完成。但是，如果程序员具有 BA 的技能，而且项目的规模不大，那么确实可以像这里说的那样自己完成软件需求的工作。

人。然而，如果你是一名学生，“客户”就变成了你的老师！无论上的是什么编程课，都几乎可以肯定老师会布置一些编程问题让你完成。为了学业的成功，确保自己理解了老师的要求，并据此写代码。

2.1.4 确定执行任务必须采取的步骤

一旦理解了程序要执行的任务，就着手将任务分解成一系列步骤。你平时教别人完成一项任务时，也会采用类似的做法：将其分解成一系列容易照着做的步骤。例如，假定有人问你如何烧开水，你可以将这项任务分解成以下步骤：

1. 将适当数量的水倒入水壶中。
2. 将水壶放到灶上。
3. 开大火。
4. 观察水，如果看到冒大泡，就表明水开了。
5. 关火。

这是算法的一个例子。**算法**（algorithm）是一组明确定义的逻辑步骤，必须采取这些步骤来执行一项任务。注意算法中的步骤是按顺序排列的。第 1 步应先于第 2 步执行，以此类推。如果一个人完全遵照这些步骤，并以正确顺序操作，就应该能成功地烧出一壶开水。

程序员以类似的方式分解一个程序必须执行的任务。在创建好的算法中，应列出所有必须采取的逻辑步骤。例如，假设有人要求写程序来计算和显示一名时薪制员工的工资总额，以下是要采取的步骤：

1. 获取工作时数
2. 获取每小时工资
3. 将工作时数乘以每小时工资
4. 显示步骤 3 的计算结果

当然，这个算法还没有准备好在计算机上执行。上述清单中的步骤必须转换为代码。程序员通常使用两种工具来帮助自己完成这项工作：伪代码和流程图。让我们更详细地了解一下这两种工具。

2.1.5 伪代码

由于像单词拼写错误和遗漏标点符号等小疏忽会导致语法错误，所以程序员在写代码时必

须注意这样的小细节。为了减少这样的错误，可以在写实际的代码之前先写好伪代码。

伪代码的英语是 pseudocode（发音为"sue doe code"）。其中，“pseudo”一词的意思是“伪”（fake），所以 pseudocode 就是伪代码。它是一种非正式的语言，没有语法规则，目的也不是被编译或执行。相反，程序员使用伪代码来创建程序的模型（mock-up）。由于程序员在写伪代码时不必担心语法错误，所以可以将全部注意力集中在程序的设计上。一旦用伪代码创建了一个令人满意的设计，伪代码就能直接转化为实际的代码。下例展示了如何为之前讨论的工资计算程序编写伪代码。

```
输入工作时长
输入每小时工资
工作时长乘以每小时工资来计算总工资
显示总工资
```

伪代码中的每个语句都代表一个可以在 Python 中执行的操作。例如，Python 可以读取键盘输入，执行数学计算，并在屏幕上显示消息。

2.1.6 流程图

流程图是程序员用来设计程序的另一种工具。**流程图**（flowchart）描述了程序中所发生的步骤。图 2-2 展示了如何为工资计算程序创建一个流程图。

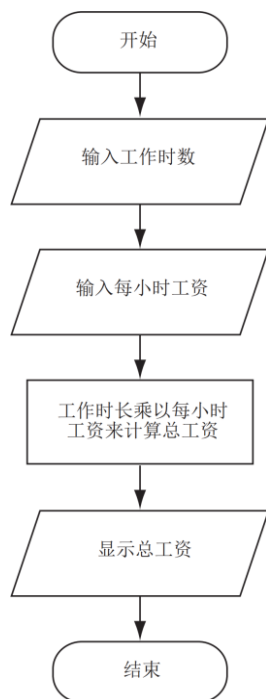


图 2-2 工资计算程序的流程图

注意，流程图中有三种类型的符号：椭圆、平行四边形和矩形。这些符号中的每一个都代表程序中的一个步骤，如下所示：

- 流程图顶部和底部的椭圆称为**终端符号**。终端符号“开始”标志着程序的起始点，终端符号“结束”标志着程序的结束点。
- 平行四边形是**输入符号**和**输出符号**，代表程序读取输入或显示输出的步骤。
- 矩形是**处理符号**，代表程序对数据进行某种处理的步骤，例如数学计算。

这些符号由箭头连接，代表程序的**流程**（flow）。为了按正确的顺序经历这些符号，需要从“开端”开始，沿着箭头一直走到“结束”。

检查点

2.1 谁是程序员的“客户”？

2.2 什么是软件需求？

2.3 什么是算法？

2.4 什么是伪代码？

2.5 什么是流程图？

2.6 以下符号在流程图中有何含义？

- 椭圆
- 平行四边形
- 矩形

2.2 输入、处理和输出

概念：输入是程序接收到的数据。当程序接收到数据时，它通常通过对其进行某种操作来处理它。操作的结果作为输出从程序中送出。

计算机程序通常执行以下三个步骤的操作：

1. 接收输入。
2. 对输入进行某种处理。
3. 生成输出。

输入是程序在运行时收到的任何数据。一种常见的输入形式是从键盘输入的数据。接收到输入后，通常要对其进行一些处理，如数学计算。然后，该过程的结果被作为输出从程序中送出。

图 2-3 展示了之前讨论的工资计算程序中的这三个步骤。工作时数和每小时工资作为输入提供。程序处理这些数据时，将工作时数乘以每小时工资。然后，计算结果作为输出显示在屏幕上。




图 2-3 工资计算程序的输入、处理和输出

本章将讨论用 Python 执行输入、处理和输出的基本方式。

2.3 用 print 函数显示输出

概念：使用 print 函数在 Python 程序中显示输出。

 视频讲解：Using the print function

函数（function）是一段预先写好的代码，用于执行一个操作。Python 有许多内置函数，可以执行各种操作。其中，最基本的内置函数或许就是 `print`，它在屏幕上显示输出。下面是一个执行 `print` 函数的语句：

```
print('Hello world')
```

在交互模式下输入这个语句并按下 Enter 键，就会显示消息“Hello world”，如下所示：

```
>>> print('Hello world') Enter
Hello world
>>>
```

当程序员执行一个函数时，他们说自己是在**调用**（call）该函数。为了调用 `print` 函数，需要输入 `print` 并后跟一对圆括号。在圆括号中，需要输入一个**实参**（argument），也就是想在屏幕上显示的数据。在上例中，我们传递的实参是 `'Hello world'`。注意，执行该语句时，引号并不会显示。引号在这里只是标记你希望显示的文本的开始与结束。

假设老师要求你写一个程序，在计算机屏幕上显示你的名字和地址。程序 2-1 展示了这种程序的一个例子，并列出了它运行时将产生的输出。（本书程序清单中的行号不是真实程序的一部分，仅供在讨论时引用）。

程序 2-1 (output.py)

```
1 print('Kate Austen')
2 print('123 Full Circle Drive')
3 print('Asheville, NC 28899')
```

程序输出

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

要注意的一个重点在于，该程序中的语句是按其出现的顺序执行的，从程序的顶部到底部。运行这个程序时，第一个语句将被执行，接着是第二个语句，然后是第三个语句，以此类推。

2.3.1 字符串和字符串面值

几乎所有程序都要处理某种类型的数据。例如，程序 2-1 使用了以下三个数据：

```
'Kate Austen'
'123 Full Circle Drive'
'Asheville, NC 28899'
```

这些数据都是字符序列。在编程术语中，作为数据使用的字符序列称为**字符串**（string）。当字符串出现在实际的程序代码中时，它被称为一个**字符串面值**（string literal）。在 Python 代码中，字符串面值必须用引号括起来。如前所述，引号的作用只是标记字符串数据的开始与结束。

在 Python 中，可以用一对单引号（' '）或一对双引号（" "）来包围字符串面值。程序 2-1 的字符串面值是用单引号括起来的，但程序也可以写成如程序 2-2 所示的样子。

程序 2-2 (double_quotes.py)

```
1 print("Kate Austen")
2 print("123 Full Circle Drive")
3 print("Asheville, NC 28899")
```

程序输出

Kate Austen
123 Full Circle Drive
Asheville, NC 28899

如果希望在字符串中包含单引号（或称撇号）字面值，可以换用双引号来包围字符串。例如，程序 2-3 打印了两个含有撇号的字符串。

程序 2-3 (apostrophe.py)

```
1 print("Don't fear!")
2 print("I'm here!")
```

程序输出

```
Don't fear!
I'm here!
```

类似地，用单引号包围字符串，就可以在字符串中包含作为字面值的双引号，如程序 2-4 所示。

程序 2-4 (display_quote.py)

```
1 print('Your assignment is to read "Hamlet" by tomorrow.')
```

程序输出

```
Your assignment is to read "Hamlet" by tomorrow.
```

Python 还允许使用三引号（"""或"""）来包围字符串字面值。在这种字符串中，单引号和双引号都可以作为字面值使用，如下例所示：

```
print("""I'm reading "Hamlet" tonight.""")
```

该语句会打印：

```
I'm reading "Hamlet" tonight.
```

还可以用三引号包围多行字符串，而这是单引号和双引号不支持的，如下例所示：

```
print("""One
Two
Three""")
```

该语句会打印：

```
One
Two
Three
```

检查点

2.7 写一个语句来显示你的名字。

2.8 写一个语句来显示以下文本：

```
Python's the best!
```

2.9 写一个语句来显示以下文本：

```
The cat said "meow."
```

2.4 注释

概念：注释对代码行或小节进行解释。注释是程序的一部分，但 Python 解释器会忽略它们。它们是为那些需要阅读源代码的人准备的。

注释（comment）是放置在程序不同部分的简短说明，用于解释程序的这些部分如何工作。虽然注释是程序的关键部分，但它们会被 Python 解释器忽略。只有需要阅读源代码的人才需要查看注释，计算机用不着。

在 Python 中，我们用#字符来开始注释。当 Python 解释器看到#字符时，它会忽略从该字符到行末的所有内容。例如，在程序 2-5 中，第 1 行和第 2 行是注释，简要地解释了该程序的作用。

2-5 (comment1.py)

```
1 # 这个程序显示一个人的
2 # 名字和地址
3 print('Kate Austen')
4 print('123 Full Circle Drive')
5 print('Asheville, NC 28899')
```

程序输出

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

程序员还经常在代码行末尾写注释，专门对当前代码行进行解释，这种注释称为**行末注释**（end-line comment）。程序 2-6 展示了一个例子。注意，每行末尾都用一条注释来解释该行的作用。

程序 2-6 (comment2.py)

```
1 print('Kate Austen') # 显示名字
2 print('123 Full Circle Drive') # 显示地址
3 print('Asheville, NC 28899') # 显示市、州和邮编
```

程序输出

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

新手不喜欢写注释，觉得写代码更有意思。但养成写注释的好习惯非常重要。未来修改或调试程序时，它们能节省大量时间。进行了良好注释的大型和复杂的程序更容易阅读和理解，修改或调试也更省事。

2.5 变量

概念：变量是内存中的具名存储位置。

程序经常要在计算机内存中存储并处理数据。以典型的网上购物为例：用户浏览网站，将想买的商品添加到购物车。商品添加到购物车时，与商品有关的数据会存储到内存中。然后，一旦单击“去购物车结算”，Web 服务器上运行的程序就会计算购物车中所有商品的价格、销售税、运费以及所有这些费用的总和。当程序执行这些计算时，会将结果存储到内存中。

程序是用变量将数据存储到内存中。**变量** (variable) 是内存中的具名存储位置。例如，计算销售税的程序可以使用名为 `tax` 的变量在内存中容纳那个值。而计算两个城市之间的距离的程序可以使用名为 `distance` 的变量。如果一个变量代表内存中的一个值，我们说该变量**引用** (reference) 那个值。

2.5.1 用赋值语句创建变量

我们用**赋值语句**创建变量并让它引用某个数据，下面是赋值语句的一个例子。

```
age = 25
```

执行该语句后，会创建名为 `age` 的一个变量，而且它会引用值 25。图 2-4 展示了这个概念。在图中，可想象值 25 存储在计算机内存的某个地方。从 `age` 指向值 25 的箭头表明变量 `age` 引用值 25。

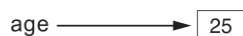


图 2-4 age 变量引用值 25

赋值语句的常规形式如下所示：

```
变量 = 表达式
```

其中，等号 (=) 称为**赋值操作符** (assignment operator)。在这种基本形式中，**变量**是变一个变量的名称，而**表达式**要么是一个值，要么是能获得一个求值结果的代码。赋值语句执行完毕后，位于等号左侧的变量将引用位于等号右侧的值。

下面来做个实验：请在交互模式下输入以下赋值语句：

```
>>> width = 10   
>>> length = 5   
>>>
```

第一个语句创建名为 **width** 的变量，并将值 **10** 赋给它。第二个语句创建名为 **length** 的变量，并将值 **5** 赋给它。接着，可以使用 **print** 函数显示这些变量引用的值，如下所示：

```
>>> print(width)   
10  
>>> print(length)   
5  
>>>
```

将变量作为实参传递给 **print** 函数时，千万不要用引号将变量名括起来。为了说明原因，请看以下交互会话的结果：

```
>>> print('width')   
width  
>>> print(width)   
10  
>>>
```

第一个语句将 **'width'** 作为实参传递给 **print** 函数，函数将打印字符串字面值 **width**，在第二个语句中，**width**（不带引号）作为实参传递给 **print** 函数，函数将打印变量 **width** 所引用的值 **10**。

在赋值语句中，被赋值的变量一定要出现在 **=** 操作符的左侧。在以下交互会话中，由于出现在 **=** 操作符左侧的不是变量，所以造成了错误。

```
>>> 25 = age   
SyntaxError: can't assign to literal  
>>>
```

程序 2-7 演示了变量的用法。第 2 行创建名为 **room** 的变量，并将值 **503** 赋给它。第 3 行和

第 4 行的语句显示一条消息。注意，第 4 行显示了由 `room` 变量引用的值。

程序 2-7 (`variable_demo.py`)

```
1 # 这个程序演示了变量的用法
2 room = 503
3 print('我的房号是')
4 print(room)
```

程序输出

```
我的房号是
503
```

程序 2-8 是使用了两个变量的例子。第 2 行创建名为 `top_speed` 的变量，并赋值 `160`。第 3 行创建名为 `distance` 的变量，并赋值 `300`。图 2-5 对此进行了演示。

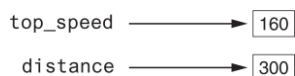



图 2-5 创建两个变量

程序 2-8 (`variable_demo2.py`)

```
1 # 创建两个变量: top_speed 和 distance.
2 top_speed = 160
3 distance = 300
4 # 显示变量引用的值
5 print('The top speed is')
6 print(top_speed)
7 print('The distance traveled is')
8 print(distance)
```

程序输出

```
The top speed is
160
The distance traveled is
300
```

 警告：变量必须先赋值再使用。对未赋值的变量进行某种操作（例如打印它）会导致一个错误。

一个简单的打字失误有时也会引起这种错误。下例的变量名称拼写有误：

```
temperature = 74.5 # 创建变量
```

```
print(tempereture) # 错误! 变量名拼错了
```

在上述代码中，赋值语句创建的变量是 `temperature`，但在 `print` 函数中拼错了变量名，所以引起错误。另外，变量名的字母大小写不一致也会引起错误。例如：

```
temperature= 74.5      # 创建变量
print(Temperature)     # 错误! 大小写不一致
```

在上述代码中，赋值语句创建的变量是 `temperature`（全小写）。而在 `print` 语句中，变量名首字母却被写成了大写 `T`。由于 Python 语言对字母的大小写是敏感的（即区分大小写），所以将引起一个错误。



注意：在内部，Python 变量的工作方式与其他大多数编程语言中的变量有所不同。在大多数编程语言中，变量是容纳值的内存位置。在这些语言中将值赋给变量时，该值被存储在变量的内存位置中。

而在 Python 中，变量是一个内存位置，它容纳了另一个内存位置的“地址”。将值赋给 Python 变量时，该值被存储到一个与变量分开的位置（即变量“指向”的位置）。变量保存的始终是实际存储了值的那个内存位置的地址。这就是为什么在 Python 中，我们不说变量“容纳”一个值，而是说变量“引用”一个值（或对象）。

2.5.2 多重赋值

Python 允许在一个语句中向多个变量赋值。这种语句称为**多重赋值语句**。如下例所示：

```
x, y, z = 0, 1, 2
```

在这个语句中，`=`操作符左侧显示了三个变量名称，分别以逗号分隔。`=`操作符右侧则有三个值，也以逗号分隔。该语句执行时，`=`操作符右侧的值会分别赋给左侧的变量。在本例中，`0` 赋给 `x`，`1` 赋给 `y`，而 `2` 赋给 `z`。

下面是另一个例子：

```
name, id = 'Trinidad', 847
```

这个语句执行后，`name` 变量被赋值 `'Trinidad'`，`id` 变量则被赋值 `847`。

2.5.3 变量命名规则

程序员在命名变量时要遵守以下规则：

- 不能使用 Python 关键字作为变量名（关键字列表请参见上一章的表 1-2）。
- 变量名不能包含空格。
- 第一个字符只能是 `a~z` 或 `A~Z` 的字母或下划线（`_`）。
- 在第一个字符之后，可以使用 `a~z` 或 `A~Z` 的任何字母、`0~9` 的任何数字或下划线的组

合。

- 字母严格区分大小写。这意味着 `ItemsOrdered` 和 `itemsordered` 是两个不同的变量名。

除了上述规则，还应该选择能清晰表达变量用途的变量名。例如，存储温度的变量可以命名为 `temperature`，存储汽车速度的变量可以命名为 `speed`。当然也可以随意给变量起名为 `x` 或 `b2`，但这样的名称无助于理解变量的用途。

由于变量名最好要反映变量的用途，所以程序员经常采用由多个单词组成的变量名，例如：

```
grosspay
payrate
hotdogssoldtoday
```

遗憾的是，这些名称不好读，因为不同的单词没有分开。由于不允许在变量名中使用空格，所以需要找到其他办法来分隔多个单词，使其更容易阅读。

一个办法是使用下划线字符来代替空格。例如，以下变量名比之前显示的更易读：

```
gross_pay
pay_rate
hot_dogs_sold_today
```

这种变量命名风格在 Python 程序员中很流行，也是本书准备采用的风格。然而，还有其他流行的风格，例如 `camelCase` 命名法。`camelCase` 命名法的规则是：

- 变量名以小写字母开始。
- 第二个单词和后续所有单词的首字母大写，如下所示：

```
grossPay
payRate
hotDogsSoldToday
```


 注意：由于出现在变量名中的大写字母很容易让人联想起骆驼的驼峰，所以这种命名法称为 `camelCase`。

表 2-1 举例说明了 Python 语言的哪些变量名合法，哪些非法。

表 2-1 示例变量名

变量名	合法还是非法？
<code>units_per_day</code>	合法

dayOfWeek	合法
3dGraph	非法，变量名不能以数字开头
June1997	合法
Mixture#3	非法，变量名只能使用字母、数字或下划线

2.5.4 用 print 函数显示多项内容

在之前的程序 2-7 中，我们在第 3 行和第 4 行使用了以下两个语句：

```
print('我的房号是')
print(room)
```

为了显示两个数据，我们调用了两次 print 函数。第 3 行显示字符串'我的房号是'，第 4 行显示变量 room 引用的值。

其实，这个程序可以简化，因为 Python 允许在一次 print 函数调用中显示多项内容。如程序 2-9 所示，用逗号来分隔多项内容即可。

程序 2-9 (variable_demo3.py)

```
1 # 这个程序演示了变量的用法
2 room = 503
3 print('我的房号是', room)
```

程序输出

```
我的房号是 503
```

第 3 行向 print 函数传递了两个实参。第一个实参是字符串'我的房号是'，第二个实参是 room 变量。当 print 函数执行时，它按照我们传递给函数的顺序显示两个实参的值。注意，print 函数会自动打印一个空格来分隔这两项。如果将多个实参传递给 print 函数，它们在屏幕上显示时会自动用一个空格隔开。

2.5.5 变量重新赋值

变量之所以称为“变量”，就是因为它能“变”——可以在程序运行期间引用不同的值。为一个变量赋值后，该变量将引用该值，直到为它赋一个不同的值。以程序 2-10 为例。第 3 行的语句创建一个名为 dollars 的变量，并为其赋值 2.75。这显示在图 2-6 的上半部分。然后，第 8 行的语句为变量 dollars 赋了一个不同的值，即 99.95。图 2-6 的下半部分显示了 dollars 变量的变化情况。旧值 2.75 仍然在计算机的内存中，只是无法再用，因为现在没有变量引用它。当内存中的一个值不再被任何变量引用时，Python 解释器会通过一个称

为**垃圾收集**（garbage collection）的过程自动将其从内存中删除。

程序 2-10 (variable_demo4.py)

```
1 # 这个程序演示了变量重新赋值。
2 # 为 dollars 变量赋一个值。
3 dollars = 2.75
4 print('I have', dollars, 'in my account.')
5
6 # 重新为 dollars 赋值,
7 # 使它引用不同的值。
8 dollars = 99.95
9 print('But now I have', dollars, 'in my account!')
```

程序输出

```
I have 2.75 in my account.
But now I have 99.95 in my account!
```

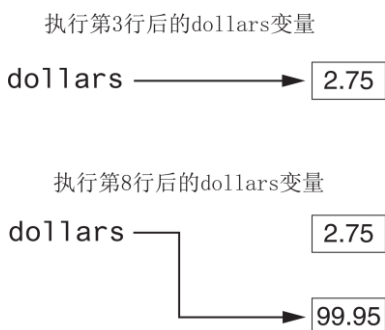


图 2-6 对程序 2-10 的变量重新赋值的说明

2.5.6 数值数据类型和字面值

第 1 章讨论了在计算机内存中存储数据的方式（1.3 节）。我们讲到，计算机使用不同的技术来存储实数（有小数部分的数字）和整数。这些类型的数字不仅存储方式不同，具体的操作方式也有所区别。

由于不同类型的数字以不同的方式存储和操作，所以 Python 使用**数据类型**对内存中的值进行归类。当一个整数被存储在内存中时，它被归类为 `int`，当一个实数被存储在内存中时，它被归类为 `float`。

现在来看看 Python 如何确定一个数字的数据类型。在之前展示的几个程序中，代码中都写入了一些数值数据。例如，在程序 2-9 的以下语句中，我们写入了数字 503：

```
room = 503
```

该语句导致值 503 存储到内存中，并让变量 `room` 引用它。程序 2-10 的以下语句写入了数字 2.75：

```
dollars = 2.75
```

该语句导致值 2.75 存储到内存中，并让变量 `dollars` 引用它。直接在程序代码中写入的数字称为**数值面值**（numeric literal）。当 Python 解释器读取代码中的一个数值面值时，会根据以下规则来确定其数据类型：

- 数值面值如果写成没有小数点的整数，那么它是 `int` 类型，例如 7、124 和 -9。
- 数值面值如果带有小数点，那么它是 `float` 类型，例如 1.5、3.14159 和 5.0。

因此，以下语句会将数字 503 作为一个 `int` 存储到内存中：

```
room = 503
```

以下语句会将数字 2.75 作为一个 `float` 存储到内存中：

```
dollars = 2.75
```

在内存中存储一个数据项时，必须明白它的数据类型是什么。稍后就会讲到，一些操作的行为会因数据类型而异，而且一些操作只能针对特定数据类型的值进行。


下面在交互模式下实验使用内置的 `type` 函数来判断一个值的数据类型：

```
>>> type(1) Enter  
<class 'int'>  
>>>
```

在这个例子中，值 1 作为实参传给 `type` 函数。下一行显示消息 `<class 'int'>`，表明该值是 `int` 类型。下面是另一个例子：

```
>>> type(1.0) Enter  
<class 'float'>  
>>>
```

在这个例子中，值 1.0 作为实参传给 `type` 函数。下一行显示消息 `<class 'float'>`，表明该值是 `float` 类型。

 **警告：**数值面值中不能包含货币符号、空格或逗号。例如，以下语句会出错：

```
value = $4,567.99 # 错误!
```

相反，上述语句必须写成：


```
value = 4567.99 # 正确
```

2.5.7 用 str 数据类型存储字符串

除了 `int` 和 `float` 数据类型，Python 还支持在内存中存储字符串的 `str` 数据类型。程序 2-11 展示了如何将字符串赋给变量。

程序 2-11 (string_variable.py)

```
1 # 创建变量来引用两个字符串
2 first_name = 'Kathryn'
3 last_name = 'Marino'
4
5 # 显示各个变量引用的值
6 print(first_name, last_name)
```

程序输出

```
Kathryn Marino
```

2.5.8 让变量引用不同数据类型

记住，Python 的所有变量都是指向内存中的一个数据块的指针。这个机制使程序员很容易存储和检索数据。在内部，Python 解释器会跟踪你所创建的变量名和它们指向的数据块。任何时候想要检索某个数据块，使用指向它的变量名即可。

Python 中的变量可以引用任何类型的数据项。将一种类型的数据项赋给一个变量后，可以重新将另一种类型的数据项赋给它。以下交互会话对此进行了演示（为方便引用，我们添加了行号）。

```
1 >>> x = 99 
2 >>> print(x) 
3 99
4 >>> x = 'Take me to your leader' 
5 >>> print(x) 
6 Take me to your leader
7 >>>
```

第 1 行的语句创建一个名为 `x` 的变量，并将 `int` 值 `99` 赋给它。图 2-7 展示了变量 `x` 是如何引用内存中的值 `99` 的。第 2 行的语句调用 `print` 函数，将 `x` 作为实参传递给它。`print` 函数的输出在第 3 行显示。然后，第 4 行的语句将一个字符串赋给 `x` 变量。这个语句执行后，`x` 变量就不再引用一个 `int`，而是引用 `'Take me to your leader'` 字符串。这在图 2-8 中进行了展示。第 5 行再次调用 `print` 函数，传递 `x` 作为实参。第 6 行显示了 `print` 函数的输出。



图 2-7 变量 x 目前引用一个整数

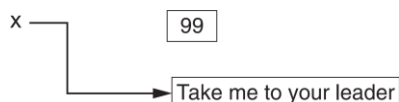


图 2-8 变量 x 现在引用一个字符串

检查点

2.10 变量是什么？

2.11 以下哪些变量名在 Python 中是非法的？为什么？

```
x
99bottles
july2009
theSalesFigureForFiscalYear
r&d
grade_report
```

2.12 变量名 Sales 和 sales 代表同一个变量吗？为什么？

2.13 以下赋值语句有效还是无效？如果无效，为什么？

```
72 = amount
```

2.14 以下代码的显示结果是什么？

```
val = 99
print('The value is', 'val')
```

2.15 请看以下赋值语句：

```
value1 = 99
value2 = 45.9
value3 = 7.0
value4 = 7
value5 = 'abc'
```

执行了这些语句后，每个变量所引用的值的数据类型是什么？


2.16 以下代码的显示结果是什么？

```
my_value = 99
my_value = 0
```

```
print(my_value)
```

2.6 从键盘读取输入

概念：程序经常需要读取用户的键盘输入。我们使用 Python 函数来实现这个功能。

 视频讲解：Reading Input from the Keyboard

你将来编写的大多数程序都需要读取输入并对其执行一些操作。本节将讨论一个基本的输入方式：读取从键盘输入的数据。当程序从键盘读取输入的数据时，通常会将这些数据存储在一个变量中，以便稍后在程序中使用。

本书使用 Python 内置的 `input` 函数从键盘读取输入。`input` 函数读取从键盘输入的数据，并将其以字符串的形式返回给程序。通常会在赋值语句中使用 `input` 函数，其常规形式如下所示：

```
变量 = input(提示)
```

采用这种形式，“提示”是在屏幕上显示的一个字符串，作用是提示用户输入一个值；“变量”则用于引用从键盘输入的数据。下例使用 `input` 函数从键盘读取数据：

```
name = input('你的名字是什么? ')
```

执行该语句时会发生下面这些事情：

- 在屏幕上显示字符串 '你的名字是什么?'。
- 程序暂停并等待用户输入并按 Enter 键。
- 按下 Enter 键后，输入的数据作为字符串返回，并赋给 `name` 变量。

以下交互会话对此进行了演示：

```
>>> name = input('你的名字是什么? ') 
你的名字是什么? Holly 
>>> print(name) 
Holly
>>>
```

输入第一个语句后，解释器显示提示消息 '你的名字是什么? '（注意问号后有个空格），并等待用户输入一些数据。用户输入 `Holly` 并按 Enter 键。结果是字符串 '`Holly`' 被赋给 `name` 变量。输入第二个语句后，解释器显示 `name` 变量所引用的值。

程序 2-12 使用 `input` 函数从键盘读取两个字符串作为输入。

程序 2-12 (string_input.py)

```
1 # 获取用户的名字。
2 first_name = input('输入你的名字: ')
3
4 # 获取用户的姓氏。
5 last_name = input('输入你的姓氏: ')
6
7 # 向用户打印一条欢迎消息。
8 print('你好,', last_name, first_name)
```

程序输出 (用户输入的内容加粗)

```
输入你的名字: 三丰 
输入你的姓氏: 张 
你好, 张 三丰
```

注意第 2 行作为提示使用的字符串:

```
'输入你的名字: '
```

冒号后有一个空格, 第 5 行的提示也是如此:

```
'输入你的姓氏: '
```

之所以在每个字符串后面添加一个空格, 是因为 `input` 函数不会在提示后自动加空格。当用户开始输入字符时, 它们感觉像是直接跟在提示消息后面。在提示字符串后面手动添加一个空格, 有助于在视觉上将提示消息与用户的输入分开。^①

2.6.1 用 `input` 函数读取数字

即使用户输入的是数值, `input` 函数也总是将输入作为字符串返回。例如, 如果调用 `input` 函数时输入 72, 并按下 Enter 键, 那么 `input` 函数返回的是字符串 '72'。在数学运算中直接使用这个值会出问题, 因为数学运算的操作数只能是数值, 不能是字符串。

幸好, Python 语言的内置函数可以帮你将字符串转换为数值。表 2-2 总结了其中的两个。

表 2-2 数据转换函数

函数	说明
<code>int(item)</code>	将一个实参传给 <code>int()</code> 函数, 它返回该实

^① 译注: 本书以后使用中文全角冒号时, 不会再添加这个空格来进行视觉分隔。

	参转换成 <code>int</code> 后的值。
<code>float(item)</code>	将一个实参传给 <code>float()</code> 函数，它返回该实参转换成 <code>float</code> 后的值。

例如，假定要写一个工资计算程序，并希望获得用户的工作时数。来看看以下代码：

```
string_value = input('你工作了多少小时? ')
hours = int(string_value)
```

第一个语句从用户处获得工作时数，并将该值作为字符串赋给 `string_value` 变量。第二个语句调用 `int()` 函数并传递 `string_value` 作为实参。随后，`string_value` 引用的值会被转换成 `int`，并赋给 `hours` 变量。

这个例子说明了 `int()` 函数是如何工作的，但它的效率很低，因为它创建了两个变量：一个用来保存从 `input` 函数返回的字符串，另一个用来保存从 `int()` 函数返回的整数。以下代码展示了一个更好的方法。它用一个语句完成了之前两个语句的所有工作，而且只创建了一个变量。

```
hours = int(input('你工作了多少小时? '))
```

该语句进行了嵌套函数调用。从 `input` 函数返回的值作为实参传递给 `int()` 函数。下面解释了它的工作方式：

- 调用 `input` 函数获取键盘输入的一个值。
- `input` 函数返回的值（一个字符串）作为实参传给 `int()` 函数。
- `int()` 函数返回的 `int` 值赋给 `hours` 变量。

执行了这个语句后，从键盘输入的值在转换成 `int` 后赋给 `hours` 变量。

下面来看另一个例子。假定要获取用户的每小时工资。以下语句提示用户输入值，将值转换成一个 `float`，再把它赋给 `pay_rate` 变量。

```
pay_rate = float(input('你的每小时工资是多少? '))
```

下面是它的工作方式：

- 调用 `input` 函数获取键盘输入的一个值。
- 将 `input` 函数返回的值（一个字符串）作为实参传给 `float()` 函数。
- 将 `float()` 函数返回的 `float` 值赋给 `pay_rate` 变量。

执行了这个语句后，从键盘输入的值在转换成 `float` 后赋给 `pay_rate` 变量。

程序 2-13 使用 `input` 函数从键盘读取一个字符串、一个 `int` 和一个 `float`。

程序 2-13 (input.py)

```
1 # 获取用户的名字、年龄和收入
2 name = input('你的名字是什么? ')
3 age = int(input('你的年龄多大? '))
4 income = float(input('你的收入有多少? '))
5
6 # 显示数据
7 print('这是你输入的数据:')
8 print('名字:', name)
9 print('年龄:', age)
10 print('收入:', income)
```

程序输出 (用户输入的内容加粗)

```
你的名字是什么? Chris 
你的年龄多大? 25 
你的收入有多少? 75000.0 
这是你输入的数据:
名字: Chris
年龄: 25
收入: 75000.0
```

下面是代码所做的事情:

- 第 2 行提示用户输入名字。输入的值作为一个字符串赋给 `name` 变量。
- 第 3 行提示用户输入年龄。输入的值被转换为一个 `int` 并赋给 `age` 变量。
- 第 4 行提示用户输入收入。输入的值被转换为一个 `float` 并赋给 `income` 变量。
- 第 7 行~第 10 行显示用户输入的值。

`int()` 和 `float()` 函数只有在被转换的数据项中包含有效数值时才起作用。如果传递的实参不能被转换为指定的数据类型, 就会发生一种称为**异常** (exception) 的错误。异常是指程序运行时发生的意外错误, 如果错误没有得到正确处理, 会导致程序停止运行。以下交互模式会话对此进行了演示。

```
>>> age = int(input('你的年龄多大? ')) 
你的年龄多大? xyz 
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    age = int(input('你的年龄多大? '))
ValueError: invalid literal for int() with base 10: 'xyz'
>>>
```



注意: 本节多次提到“用户”(user)。用户是一名假想的、任何使用程序并为其提供输入的人。有时也将其称为“终端用户”或“最终用户”(end user)。

检查点

2.17 某个程序要求输入客户的姓氏 (last name)。请写一个语句，提示用户输入该数据，并将输入的数据赋给一个变量。

2.18 某个程序要求输入本周销售额 (amount of sales)。请写一个语句，提示用户输入该数据，并将输入的数据赋给一个变量。

2.7 执行计算

概念：Python 提供了多个用于执行数学计算的操作符。

大多数现实世界的算法都要求执行某种计算。程序员利用**数学操作符**或**数学运算符** (math operators) 执行这些计算。表 2-3 总结了 Python 语言支持的数学操作符。

表 2-3 Python 数学操作(运算)符

符号	操作(运算)	说明
+	加	两数相加
-	减	一个数减去另一个数
*	乘	一个数乘以另一个数
/	除	一个数除以另一个数，结果作为浮点数返回
//	整数除法	一个数除以另一个数，结果作为整数返回
%	求余	一个数除以另一个数，返回余数
**	求幂	计算乘方

程序员可以利用表 2-3 的操作符创建数学表达式。**数学表达式** (math expression) 执行计算并返回一个值。下面展示了一个简单的数学表达式：

12 + 2

+操作符左右两侧的值称为**操作数** (operand)。它们是+操作符要加到一起的值。如果在交互模式中输入该表达式，会显示结果 14。

```
>>> 12 + 2 Enter
14
>>>
```

变量也可以在数学表达式中使用。例如，假设有两个名为 `hours` 和 `pay_rate` 的变量。以下数学表达式使用*操作符将 `hours` 变量引用的值乘以 `pay_rate` 变量引用的值。

```
hours * pay_rate
```

使用数学表达式来计算一个值时，通常希望把这个值保存在内存中，这样就可以在程序中再次使用。我们用一个赋值语句来做这件事。程序 2-14 展示了一个例子。

程序 2-14 (simple_math.py)

```
1 # 将一个值赋给 salary 变量
2 salary = 2500.0
3
4 # 将一个值赋给 bonus 变量
5 bonus = 1200.0
6
7 # 将 salary 和 bonus 相加来计算总工资，
8 # 并将结果赋给 pay。
9 pay = salary + bonus
10
11 # 显示 pay
12 print('你的工资是', pay)
```

程序输出

```
你的工资是 3700.0
```

第 2 行将 `2500.0` 赋给 `salary` 变量，第 5 行将 `1200.0` 赋给 `bonus` 变量。第 9 行将表达式 `salary + bonus` 的求值结果赋给 `pay` 变量。如程序输出所示，`pay` 变量现在容纳的值是 `3700.0`。

聚光灯：计算百分数



如果要写的程序使用了百分数，那么必须确保百分数的小数点在正确的位置，然后才能对其执行任何数学运算。由用户输入一个百分数时，这一点尤其重要。许多用户输入数字 50 表示 50%，输入 20 表示 20%，以此类推。在对这样的百分数进行任何计算之前，必须先

将其除以 100，使其小数点左移两位。

来看看写一个计算百分数的程序的步骤。假设一家零售企业计划进行一次全店大促销，所有商品打八折。我们被要求写一个程序来计算打折后的商品售价。以下是算法：

1. 获取商品的原价。
2. 计算原价的 20%。这就是折扣金额。
3. 从原价中减去折扣金额，这就是折后价。
4. 显示售价。

步骤 1 获取商品的原价。我们提示用户从键盘输入该数据，如以下语句所示。注意，用户输入的数值将被储存在一个名为 `original_price` 的变量中。

```
original_price = float(input("输入商品原价: "))
```

步骤 2 计算折扣金额。为此，我们用原价乘以 20%。以下语句执行这一计算，并将结果赋给 `discount` 变量：

```
discount = original_price * 0.2
```

步骤 3 从原价中减去折扣金额。以下语句进行这一计算，并将结果赋给 `sale_price` 变量：

```
sale_price = original_price - discount
```

最后，步骤 4 使用以下语句来显示折后售价：

```
print('折后价是', sale_price)
```

程序 2-15 展示了完整的程序和一个示例输出。

程序 2-15 (sale_price.py)

```
1  # 这个程序获取商品的原始价格，
2  # 并计算它打八折后的价格。
3
4  # 获取商品的原始价格
5  original_price = float(input("输入商品原价: "))
6
7  # 计算折扣金额
8  discount = original_price * 0.2
9
10 # 计算折后价
11 sale_price = original_price - discount
12
```

```
13 # 显示折后价
14 print('折后价是', sale_price)
```

程序输出（用户输入的内容加粗）

```
输入商品原价: 100.00 
折后价是 80.0
```

2.7.1 浮点和整数除法

注意，表 2-3 列出了 Python 的两种不同的除法操作符。其中，/操作符执行浮点除法，而//操作符执行整数除法。这两个操作符都是用一个数字除以另一个数字。区别在于，/操作符返回的是浮点数结果，而//操作符返回的是整数。以下交互模式对此进行了演示：

```
>>> 5 / 2 
2.5
>>>
```

在会话中，我们使用/操作符将 5 除以 2。和预期的一样，结果是 2.5。下面再使用//操作符执行整数除法：

```
>>> 5 // 2 
2
>>>
```

如你所见，返回的结果是 2。//操作符的工作方式向下取整，或者称为“地板除”：

- 如果结果为正数，就直接截去小数部分。
- 如果结果为负数，就取远离 0 的第一个整数。

以下交互会话演示了在结果为负时//操作符是如何工作的：

```
>>> -5 // 2 
-3
>>>
```

2.7.2 操作符优先级

可以在语句中使用包含多个操作符的复杂数学表达式。以下语句将 17、变量 x、21 以及变量 y 之和赋给变量 answer：

```
answer = 17 + x + 21 + y
```

但是，有的表达式并没有这么直观。来看看下面这个语句：

```
outcome = 12.0 + 6.0 / 3.0
```

最终赋给变量 outcome 的值是什么？6.0 既可能是加法操作符的操作数，也可能是除法操作符的操作数。根据除法是先运算还是后运算，变量 outcome 既可能被赋值为 14.0，也可

能被赋值为 `6.0`。幸好，这个结果是可以预测的，因为 Python 遵循的运算顺序与我们在数学课中学到的运算顺序完全一致。

首先，圆括号内的运算最先进行。然后，当两个操作符共享一个操作数时，优先级高的操作符先求值。数学操作符的优先级由高到低是：

1. 求幂操作符：`**`
2. 乘、除和求余：`*` / `/` / `%`
3. 加和减：`+` -

注意，乘法操作符 (`*`)、浮点除法操作符 (`/`)、整数除法操作符 (`//`) 以及求余操作符 (`%`) 的优先级相同，加法操作符 (`+`) 和减法操作符 (`-`) 的优先级相同。当优先级相同的两个操作符共享同一个操作数时，求值顺序是从左向右。

再来看看刚才的数学表达式：

```
outcome = 12.0 + 6.0 / 3.0
```

由于除法操作符的优先级高于加法操作符的优先级，所以最终赋给 `outcome` 的值是 `14.0`。换言之，除法运算先于加法运算执行。该表达式的运算顺序如图 2-9 所示。

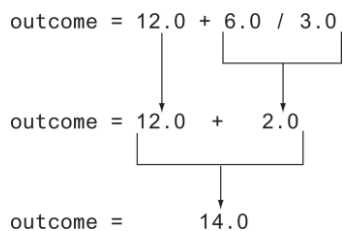


图 2-9 操作符优先级

表 2-4 展示了其他一些示例表达式及其求值结果。

表 2-4 示例表达式

表达式	值
<code>5 + 2 * 4</code>	13
<code>10 / 2 - 3</code>	2.0
<code>8 + 12 * 2 - 4</code>	28

$6 - 3 * 2 + 7 - 1$	6
---------------------	---



注意：从左向右规则有一个例外。当两个**操作符共享一个操作数时，两个操作符按从右向左的顺序求值。例如，表达式 $2**3**4$ 实际是这样求值的： $2**(3**4)$ 。不管从左向右，还是从右到左，这些都称为操作符的“结合性”。

2.7.3 用圆括号分组

数学表达式的各个部分可以用圆括号分组，以强制优先执行某些运算地。以下语句先求 a、b 之和，再将结果除以 4：

```
result = (a + b) / 4
```

如果不使用圆括号，b 会先除以 4，结果再和 a 相加。表 2-5 展示了用圆括号强制优先级的一些例子。

表 2-5 用圆括号强制优先级

表达式	值
$(5 + 2) * 4$	28
$10 / (5 - 3)$	5.0
$8 + 12 * (6 - 2)$	56
$(6 - 3) * (2 + 7) / 3$	9.0

聚光灯：计算平均值



求一组数字的平均值是一个简单的计算：先求所有数字之和，再用数字的个数去除这组数字之和。虽然计算简单，但在写一个求平均值的程序时，还是很容易出错的。例如，假设变量 a、b 和 c 分别存储了一个值，现在要计算它们的平均值。如果不细心的话，很容易写出如下所示的语句：

```
average = a + b + c / 3.0
```

能看出该语句中的错误吗？当它执行时，最先执行的是除法运算！c 的值先除以 3，结果

再与 $a + b$ 的结果相加。这并不是计算平均值的正确顺序。为了修复这个问题，需要像下面这样将 $a + b + c$ 用圆括号括起来：

$$\text{average} = (a + b + c) / 3.0$$

让我们一步一步体验计算平均值的程序应该如何编写。假设你是计算机专业某个班级的学生，参加了班级的三次考试，现在想写一个程序来显示三次考试的平均成绩。下面是算法：

1. 获得第一个成绩。
2. 获得第二个成绩。
3. 获得第三个成绩。
4. 将三个成绩相加并除以 3，计算平均值。
5. 显示平均成绩。

在步骤 1~步骤 3 中，程序提示输入三个考试成绩。我们把这些成绩存储在变量 `test1`，`test2` 和 `test3` 中。步骤 4 计算三个成绩的平均值。我们使用以下语句来进行这个计算，并将结果存储到 `average` 变量中。

$$\text{average} = (\text{test1} + \text{test2} + \text{test3}) / 3.0$$

最后，步骤 5 显示平均成绩。程序 2-16 展示了完整程序。

程序 2-16 (test_score_average.py)

```
1  # 获取三个考试成绩，并分别赋给
2  # test1, test2 和 test3 变量
3  test1 = float(input('输入第一个成绩: '))
4  test2 = float(input('输入第二个成绩: '))
5  test3 = float(input('输入第三个成绩: '))
6
7  # 计算三个成绩的平均值，
8  # 并将结果赋给 average 变量。
9  average = (test1 + test2 + test3) / 3.0
10
11 # 显示 average
12 print('平均成绩是', average)
```

程序输出 (用户输入的内容加粗)

```
输入第一个成绩: 90 
输入第二个成绩: 80 
输入第三个成绩: 100 
```

平均成绩是 90.0

2.7.4 求幂操作符

连写的两个星号（**）是求幂操作符，用于求一个数的乘方。例如，以下语句求 `length` 的 2 次方，结果（幂）赋给 `area` 变量：

```
area = length**2
```

以下交互会话展示了表达式 `4**2`、`5**3` 和 `2**10` 的求值结果：

```
>>> 4**2   
16  
>>> 5**3   
125  
>>> 2**10   
1024  
>>>
```

2.7.5 求余操作符

在 Python 中，`%` 符号是求余操作符（也称为取模操作符）。求余操作符执行除法运算，但返回余数而不是商。以下语句将余数 2 赋给 `leftover` 变量：

```
leftover = 17 % 3
```

由于 17 除以 3 的结果是商 5 余 2，所以上述语句将 2 赋给 `leftover`。求余操作符在某些情况下非常好用，包括转换时间/距离以及判断奇偶等。程序 2-17 要求用户输入一个秒数，然后将其转换成时、分和秒。例如，11 730 秒会被转换成 3 小时 15 分 30 秒。

程序 2-17 (`time_converter.py`)

```
1 # 获取秒数  
2 total_seconds = float(input('输入秒数: '))  
3  
4 # 计算小时数  
5 hours = total_seconds // 3600  
6  
7 # 计算余下的分钟数  
8 minutes = (total_seconds // 60) % 60  
9  
10 # 计算余下的秒数  
11 seconds = total_seconds % 60  
12  
13 # 显示结果  
14 print('时、分和秒分别是:')  
15 print('时:', hours)  
16 print('分:', minutes)  
17 print('秒:', seconds)
```

程序输出（用户输入的内容加粗）

```
输入秒数: 11730   
时、分和秒分别是:   
时: 3.0  
分: 15.0  
秒: 30.0
```

下面仔细研究一下代码：

- 第 2 行获取从键盘输入的秒数，将该值转换为 `float` 并赋给 `total_seconds` 变量。
- 第 5 行计算指定的秒数换算成小时。一小时有 3600 秒，所以这个语句将 `total_seconds` 除以 3600。注意，这里使用的是整数除法操作符 (`//`)，因为想要的是没有小数部分的小时数。
- 第 8 行计算剩余的分钟数。这个语句首先使用 `//` 操作符将 `total_seconds` 除以 60。这样就得到了总的分钟数。然后，它使用 `%` 操作符将总分钟数除以 60 并返回余数。结果就是剩余的分钟数。
- 第 11 行计算剩余的秒数。一分钟有 60 秒，所以这个语句使用 `%` 操作符，用 `total_seconds` 除以 60 并返回余数。结果就是剩余的秒数。
- 第 14 行~第 17 行显示时、分和秒。

2.7.6 将数学公式转换为编程语句

数学课会讲到，表达式 $2xy$ 被理解为 2 乘以 x 乘以 y 。在数学中，你并不总是使用操作符来表示乘法运算。但是，和其他编程语言一样，Python 的任何数学运算都要求一个操作符。表 2-6 展示了一些乘法代数表达式及其对应的编程表达式。

表 2-6 代数表达式

代数表达式	执行的运算	编程表达式
$6B$	6 乘 B	<code>6 * B</code>
$(3)(12)$	3 乘 12	<code>3 * 12</code>
$4xy$	4 乘 x 乘 y	<code>4 * x * y</code>

将某些代数表达式转换成编程表达式时，可能需要插入代数表达式中没有的圆括号。例如以下公式：

$$x = \frac{a + b}{c}$$

为了把它转换成编程语句， $a + b$ 必须放到一对圆括号中：

$$x = (a + b)/c$$

表 2-7 展示了其他一些代数表达式及其对应的 Python 表达式。

表 2-7 代数和编程表达式

代数表达式	Python 语句
$y = 3\frac{x}{2}$	<code>y = 3 * x / 2</code>
$z = 3bc + 4$	<code>z = 3 * b * c + 4</code>
$a = \frac{x + 2}{b - 1}$	<code>a = (x + 2) / (b - 1)</code>

聚光灯：将数学公式转换为编程语句



假设你想把一定金额的钱存入储蓄账户，并在未来 10 年内不动以生息。10 年后，你希望账户里有 10000 美元。你今天需要存入多少钱才能实现这一目标？计算公式如下所示：

$$P = \frac{F}{(1 + r)^n}$$

下面是对公式中各项的解释：

- P 是现值，即你今天需要存入的金额。
- F 是你希望的账户终值。（在本例中， F 是 10000 美元）。
- r 是年利率。
- n 是你打算让钱放在账户里的年数。

这个计算最好用一个计算机程序来实现，因为可以用不同的变量值进行实验。下面是算法：

1. 获取终值。
2. 获取年利率。
3. 获取存款年限。
4. 计算需要存入的金额。

5. 显示步骤 4 的计算结果。

步骤 1~步骤 3 提示用户输入特定的值。我们把希望的终值赋给 `future_value` 变量，把年利率赋给 `rate` 变量，把存款年限赋给 `year` 变量。

步骤 5 计算现值，也就是当前需要存入的金额。我们将之前展示的公式转换为以下语句，它将计算结果存储到 `present_value` 变量中。

```
present_value = future_value / (1.0 + rate)**years
```

步骤 5 显示 `present_value` 变量的值。程序 2-18 展示了完整程序。

程序 2-18 (`future_value.py`)

```
1  # 获取希望的终值(future value)
2  future_value = float(input('输入你希望的终值: '))
3
4  # 获取年利率
5  rate = float(input('输入年利率: '))
6
7  # 获取存款年限
8  years = int(input('输入存款年限: '))
9
10 # 计算当前需要的存款金额
11 present_value = future_value / (1.0 + rate)**years
12
13 # 显示需要的存款金额
14 print('你需要存入的金额是:', present_value)
```

程序输出 (用户输入的内容加粗)

```
输入你希望的终值: 10000.0 
输入年利率: 0.05 
输入存款年限: 10 
你需要存入的金额是: 6139.132535407592
```



注意：这个程序的输出并不专业，货币金额通常应四舍五入为两位小数。本章稍后会解释如何格式化数字来四舍五入为指定小数位数。

2.7.7 混合类型的表达式和数据类型转换

对两个操作数进行数学运算时，结果的数据类型取决于操作数的数据类型。Python 在对数学表达式求值时遵循以下规则：

- 对两个 `int` 进行运算时，结果是一个 `int`。
- 对两个 `float` 进行运算时，结果将是一个 `float`。

- 对一个 `int` 和一个 `float` 进行运算时，`int` 值被临时转换为 `float`，返回的结果是一个 `float`。（如果表达式使用了不同数据类型的操作数，那么这种表达式称为**混合类型的表达式**）。

前两种情况很容易理解：对 `int` 的运算生成 `int`，对 `float` 的运算生成 `float`。下面是第三种情况的一个例子，它使用了混合类型的表达式：

```
my_number = 5 * 2.0
```

该语句执行时，值 5 被转换成一个 `float` (`5.0`)，然后乘以 `2.0`。结果值 `10.0` 被赋给 `my_number`。

在上述语句中，从 `int` 到 `float` 的转换是隐式发生的。如果想要显式地转换，可以使用 `int()` 或 `float()` 函数。例如，可以使用 `int()` 函数将一个浮点值转换为整数，如以下代码所示：

```
fvalue = 2.6
ivalue = int(fvalue)
```

第一个语句将值 `2.6` 赋给 `fvalue` 变量。第二个语句将 `fvalue` 作为实参传给 `int()` 函数。`int()` 函数返回整数值 `2` 并赋给 `ivalue` 变量。执行这个语句后，`fvalue` 变量的值仍然是 `2.6`，但 `ivalue` 变量的值变成 `2`。

如上例所示，`int()` 函数通过截去一个浮点实参的小数部分来返回整数。下例对一个负数执行同样的操作：

```
fvalue = -2.9
ivalue = int(fvalue)
```

第二个语句从 `int()` 函数返回的值是 `-2`。执行这个语句后，`fvalue` 变量引用的值还是 `-2.9`，而 `ivalue` 变量引用值 `-2`。

还可以使用 `float()` 函数将一个 `int` 显式转换为 `float`，如下例所示：

```
ivalue = 2
fvalue = float(ivalue)
```

上述代码执行完毕后，`ivalue` 变量仍然引用整数值 `2`，而 `fvalue` 变量引用浮点值 `2.0`。

2.7.8 将长语句拆分为多行

在大多数编程语言中，语句都是一行一行写的。但是，如果一个语句太长，就不得不利用水平滚动条才能一睹全貌。另外，如果在纸上打印程序，同时其中某个语句太长，就会发生自动换行，造成代码难以阅读。

Python 允许程序员使用称为**续行符**（line continuation character）的反斜杠（`\`）字符将一个

语句拆分成若干行。只需在打算拆分语句的地方输入一个反斜杠字符，然后按 **Enter** 键即可。例如，以下执行数学运算的语句被拆分成两行：

```
result = var1 * 2 + var2 * 3 + \  
        var3 * 4 + var4 * 5
```

第一行末尾的续行符告诉解释器该语句在下一行延续。

Python 还允许程序员无须使用续行符就可以将圆括号内的语句片段拆分成若干行，如下例所示：

```
print("周一的销售额是", monday,  
      "周二的销售额是", tuesday,  
      "周三的销售额是", wednesday)
```

下面是另一个例子：

```
total = (value1 + value2 +  
        value3 + value4 +  
        value5 + value6)
```

检查点

2.19 请在下面“数值”这一栏填写相应表达式的值。

表达式	数值
$6 + 3 * 5$	_____
$12 / 2 - 4$	_____
$9 + 14 * 2 - 6$	_____
$(6 + 2) * 3$	_____
$14 / (11 - 4)$	_____
$9 + 12 * (8 - 3)$	_____

2.20 执行以下语句后，**result** 的值是什么？

```
result = 9 // 2
```

2.21 执行以下语句后，**result** 的值是什么？

```
result = 9 % 2
```

2.8 字符串连接

概念：字符串连接是指将一个字符串附加到另一个字符串的末尾。

对字符串进行的一个常见操作是**连接或拼接**（concatenation），也就是将一个字符串附加到另一个字符串的末尾。Python 使用+操作符来连接字符串。+操作符会生成一个新字符串，它合并了作为其操作数的两个字符串。以下交互会话展示了一个例子：

```
>>> message = 'Hello ' + 'world'   
>>> print(message)   
Hello world  
>>>
```

第一个语句合并'Hello '和'world'以生成字符串'Hello world'。然后，字符串'Hello world'被赋给 message 变量。第二个语句显示合并后的字符串。

程序 2-19 进一步演示了字符串连接。

程序 2-19 (concatenation.py)

```
1 # 这个程序演示了字符串连接(拼接)。  
2 first_name = input('输入你的名字: ')  
3 last_name = input('输入你的姓氏: ')  
4  
5 # Combine the names with a space between them.  
6 full_name = last_name + first_name  
7  
8 # 显示用户的全名  
9 print('你的全名是' + full_name)
```

程序输出（用户输入的内容加粗）

```
输入你的名字: 三丰  
输入你的姓氏: 张  
你的全名是张三丰
```

来仔细研究一下这个程序。第 2 行和第 3 行提示用户输入名字和姓氏。前者赋给 first_name 变量，后者赋给 last_name 变量。

第 6 行将字符串连接的结果赋给 full_name 变量。连接字符串的顺序是先姓氏后名字。在示例输出中，用户为名字输入“三丰”，为姓氏输入“张”。结果是字符串'张三丰'被赋给 full_name 变量。第 9 行的语句显示 full_name 变量的值。

如果一个字符串字面值很长，那么可以利用字符串连接技术对其进行拆分，使一个冗长的 `print` 函数调用能跨越多行。如下例所示：

```
print('输入每一天的'  
+ '销售额，然后按'  
+ 'Enter 键。')
```

该语句的输出结果如下所示：

```
输入每一天的销售额，然后按 Enter 键。
```

隐式连接字符串字面值

当两个或更多的字符串字面值写在一起，只用空格、制表符或换行符分隔时，Python 将隐式地把它们连接成一个字符串。例如，来看看下面这个交互会话：

```
>>> my_str = '一' '二' '三' Enter  
>>> print(my_str) Enter  
一二三
```

在第一行，字符串字面值 '一'、'二' 和 '三' 仅用一个空格分隔。结果是 Python 把它们连接成单个字符串 '一二三' 并赋给 `my_str` 变量。

经常利用字符串字面值的隐式连接将一个长字符串拆分为多行，如下例所示：

```
print('输入每一天的'  
      '销售额，然后按'  
      'Enter 键。')
```

该语句的输出结果如下所示：

```
输入每一天的销售额，然后按 Enter 键。
```

检查点

2.22 什么是字符串连接

2.23 执行以下语句后，`result` 的值是什么？

```
result = '1' + '2'
```

2.24 执行以下语句后，`result` 的值是什么？

```
result = 'h' 'e' 'l' 'l' 'o'
```

2.9 关于 `print` 函数的更多知识

到目前为止，我们只讨论了数据的基本显示方式。不过，你终究还是需要对着屏幕上显示的

数据进行更多的控制。本节将讨论 Python `print` 函数的更多细节，并说明如何以特定的方式格式化输出。

2.9.1 阻止 `print` 函数的换行功能

`print` 函数默认输出独立的一行。例如，以下三个语句将产生三行输出：

```
print('一')
print('二')
print('三')
```

每个语句都是先显示字符串，然后打印一个**换行符**（`newline character`）。换行符是看不见的，但打印它会将输出位置前进到下一行的起始位置。（也可以把换行符理解成一个让计算机从下一行开始输出的特殊命令。）

如果不希望 `print` 函数在打印完当前行的内容后换到新行，可以向函数传递特殊实参 `end = ''` 来取代默认的换行符。如下所示：

```
print('一', end='')
print('二', end='')
print('三')
```

注意，在本例使用的实参 `end=''` 中，两个引号之间没有空格。它告诉 `print` 函数在当前打印的字符串的末尾什么都不打印。将上述三个语句放到一个程序文件中，程序的输出是：

```
一二三
```

也可以在引号之间添加任意你想在当前字符串末尾打印的内容，例如一个空格。

2.9.2 指定分隔符

向 `print` 函数传递多个字符串实参时，这些字符串在显示时默认会以一个空格来分隔。以下交互会话展示了一个例子：

```
>>> print('一', '二', '三') Enter
一 二 三
>>>
```

如果不想要这个默认添加的空格，可以向 `print` 函数传递一个 `sep=''` 实参，如下所示：

```
>>> print('一', '二', '三', sep='') Enter
一二三
>>>
```

利用这个特殊的实参，还可以指定其他分隔符来代替空格，如下例所示：

```
>>> print('一', '二', '三', sep='*') Enter
一*二*三
```

```
>>>
```

本例向 `print` 函数传递的是 `sep='*'`，它指定在打印的各项之间用星号（*）分隔。下面是另一个例子：

```
>>> print('一', '二', '三', sep='~~~') Enter
一~~~二~~~三
>>>
```

2.9.3 转义序列

转义序列（escape sequence）是字符串字面值中以反斜杠（\）开始的特殊字符。打印包含转义序列的字符串字面值时，转义序列被视为在字符串中嵌入的一个特殊控制命令。

例如，`\n` 就是一个换行符转义序列。`\n` 在打印时不会显示，它的作用是将输出位置前进到下一行的起始位置。例如以下语句：

```
print('一\n二\n三')
```

执行该语句将输出：

```
一
二
三
```

表 2-8 列出了 Python 支持的部分转义序列。

表 2-8 Python 的部分转义序列

转义序列	效果
<code>\n</code>	使输出位置前进到下一行的起始位置
<code>\t</code>	使输出位置前进到下一个水平制表位（tab）
<code>\'</code>	打印一个单引号
<code>\"</code>	打印一个双引号
<code>\\</code>	打印一个反斜杠

转义序列 `\t` 使输出位置前进到下一个水平制表位（通常，每 8 个字符位置就是一个制表位），例如下面这些语句：

```
print('周一\t周二\t周三')
```

```
print('周四\t周五\t周六')
```

第一个语句先打印“周一”，然后前进到下一个水平制表位并打印“周二”，然后再次前进到下一个水平制表位并打印“周三”……

输出结果如下所示：

```
周一    周二    周三
周四    周五    周六
```

\'和\"转义序列分别用于显示单引号和双引号，以下语句进行了更直观的演示：

```
print("Your assignment is to read \"Hamlet\" by tomorrow.")
print('I\'m ready to begin.')
```

上述语句的输出是：

```
Your assignment is to read "Hamlet" by tomorrow.
I'm ready to begin.
```

\\转义序列用于显示一个反斜杠，如下所示：

```
print('文件路径是 C:\\temp\\data。')
```

上述语句的输出是：

```
文件路径是 C:\temp\data。
```

检查点

2.25 如何阻止 print 函数在末尾自动换行？

2.26 如何更改 print 函数在输出的多项内容之间自动插入的分隔符？

2.27 '\n' 转义序列输出的是什麼？

2.10 用 f 字符串格式化输出

概念：f 字符串是一种特殊类型的字符串字面值，允许以多种方式格式化输出。

f 字符串也称为**格式化字符串字面值**（formatted string literals），它允许以一种简便的方式格式化 print 函数的输出。f 字符串允许创建包含变量值的消息，还允许以多种方式格式化数字。

f 字符串是附加了字母 f 前缀并包含在引号中的一个字符串字面值，下面是一个非常简单

的例子：

```
f'Hello world'
```

它看起来就像一个普通的字符串字面值，只是附加了字母 `f` 前缀。为了显示一个 `f` 字符串，我们像对待普通字符串那样把它传给 `print` 函数：

```
>>> print(f'Hello world')   
Hello world
```

但是，`f` 字符串比普通字符串字面值强大得多。例如，`f` 字符串可以包含变量和其他表达式的占位符。以下交互会话展示了一个例子：

```
>>> name = '张三丰'   
>>> print(f'你好, {name}')   
你好, 张三丰
```

第一个语句将 `'张三丰'` 赋给 `name` 变量，第二个语句将一个 `f` 字符串传递给 `print` 函数。在 `f` 字符串中，`{name}` 是 `name` 变量的**占位符**（placeholder）。执行该语句时，占位符会被替换为 `name` 变量的值。最终，该语句会打印“你好，张三丰”。

下面是另一个例子：

```
>>> temperature = 25   
>>> print(f'当前温度是{temperature}度')   
当前温度是 25 度
```

第一个语句将值 `45` 赋给 `name` 变量，第二个语句将一个 `f` 字符串传递给 `print` 函数。在 `f` 字符串中，`{temperature}` 是 `temperature` 变量的占位符。执行该语句时，占位符会被替换为 `temperature` 变量的值。最终，该语句会打印“当前温度是 25 度”。

2.10.1 占位符表达式

在之前的 `f` 字符串例子中，我们使用占位符显示变量值。除了变量名，占位符中还可以包含任何有效的表达式。下面展示了一个例子：

```
>>> print(f'值是{10 + 2}。')   
值是 12。
```

在这个例子中，`{10 + 2}` 是占位符。执行该语句时，占位符会被替换为表达式 `10 + 2` 的求值结果。下面是另一个例子：

```
>>> val = 10   
>>> print(f'值是{val + 2}。')   
值是 12。
```

第一个语句将值 `10` 赋给 `val` 变量。第二个语句将包含占位符 `{val + 2}` 的一个 `f` 字符串传递给 `print` 函数。执行该语句时，占位符会被替换为表达式 `val + 2` 的求值结果。

上例的一个重点在于，`val` 变量的值没有发生任何变化。表达式 `val + 2` 直接返回值 `12`，它不会以任何方式修改 `val` 变量的值。

2.10.2 格式化值

可以为 `f` 字符串的占位符使用一个**格式说明符** (format specifier)，从而格式化占位符的值的输出。例如，可以通过格式说明符将数值四舍五入为指定的小数位数，可以使用逗号分隔符显示数字，还可以使值左、右或居中对齐。事实上，可以通过格式说明符来控制值的许多显示方式。

下面是在占位符中使用格式说明符的常规形式：

```
{占位符:格式说明符}
```

注意，采用这种常规形式，占位符和格式说明符之间要用一个冒号来分隔。后续几个小节描述了使用格式说明符的几种方式。

2.10.3 浮点数四舍五入

浮点数的默认显示方式并非总是令人感到满意。浮点数用 `print` 函数显示时，它最多可以有 17 位有效数字，如程序 2-20 的输出所示。

程序 2-20 (`f_string_no_formatting.py`)

```
1 # 这个程序演示了在不格式化的情况下，
2 # 一个浮点数是如何显示的。
3 amount_due = 5000.0
4 monthly_payment = amount_due / 12.0
5 print(f'月工资为{monthly_payment}。')
```

程序输出

```
月工资为 416.6666666666667。
```

由于这个程序显示的是货币金额，所以最好四舍五入为两位小数。我们使用以下格式说明符来实现这一点：

```
.2f
```

在格式说明符中，`.2` 是精度指示符，它指出数字应四舍五入为两位小数。字母 `f` 是类型指示符，代表浮点类型。在第 5 行，如果我们为 `f` 字符串添加这个格式说明符，如下所示：

```
{monthly_payment:.2f}
```

那么会造成 `monthly_payment`（月工资）变量的值在程序输出中显示为 `416.67`。程序 2-21 对此进行了演示。

程序 2-21 (f_string_rounding.py)

```
1 # 这个程序演示了如何
2 # 对浮点数进行四舍五入。
3 amount_due = 5000.0
4 monthly_payment = amount_due / 12.0
5 print(f'月工资为{monthly_payment:.2f}。')
```

程序输出

月工资为 416.67。

以下交互会话演示了如何四舍五入为 3 位小数：

```
>> pi = 3.1415926535 
>> print(f'{pi:.3f}') 
3.142
>>>
```

以下交互会话演示了如何将一个占位符表达式的求值结果四舍五入为 1 位小数：

```
>> a = 2 
>> b = 3 
>> print(f'{a / b:.1f}') 
0.7
>>>
```



注意：浮点类型指示符既可以写成小写的 `f`，也可以写成大写的 `F`。

2.10.4 插入逗号分隔符

一个较大的数字最好加上逗号（千位）分隔符以便阅读。可以使用格式说明符为数字自动添加这种分隔符，如以下交互会话所示：

```
>>> number = 1234567890.12345 
>>> print(f'{number:,}') 
1,234,567,890.12345
>>>
```

下例在对数字四舍五入的同时加上逗号分隔符：

```
>>> number = 1234567890.12345 
>>> print(f'{number:,.2f}') 
1,234,567,890.12
>>>
```

在格式说明符中，逗号必须放在精度指示符之前（左侧）。否则，代码执行时会报错。

程序 2-22 演示了如何利用逗号分隔符和两位精度将一个数字格式化为货币金额。

程序 2-22 (f_string_dollar_display.py)

```
1 # 这个程序演示了如何将浮点数
2 # 显示为货币金额。
3 monthly_pay = 5000.0
4 annual_pay = monthly_pay * 12
5 print(f'你的年工资为${annual_pay:,.2f}')
```

程序输出

```
你的年工资为$60,000.00
```

2.10.5 将浮点数格式化为百分比

可以不使用 `f` 作为类型指示符，而是使用 `%` 将浮点数格式化为百分比。`%` 会使数字乘以 `100`，并在后面显示一个 `%` 符号，如下例所示：

```
>>> discount = 0.5 
>>> print(f'{discount:%}') 
50.000000%
>>>
```

下例使输出的值不保留任何小数：

```
>>> discount = 0.5
>>> print(f'{discount:.0%}') 
50%
>>>
```

2.10.6 用科学计数法格式化

如果希望用科学记数法显示浮点数，可以用字母 `e` 或 `E` 代替 `f`，下面是一些例子：

```
>>> number = 12345.6789 
>>> print(f'{number:e}') 
1.234568e+04
>>> print(f'{number:.2e}') 
1.23e+04
>>>
```

第一个语句用科学记数法简单格式化数字。字母 `e` 之后的数字表示以 `10` 为底的指数。（如果在格式说明符中使用大写 `E`，结果也会用大写 `E` 来表示指数）。第二个语句额外指定了在小数点后保留两位精度。

2.10.7 格式化整数

之前所有的例子演示的都是如何对浮点数进行格式化。还可以使用 `f` 字符串格式化整数。和浮点数相比，整数的格式说明符有两个不同的地方需要注意：

- 使用 `d` 或 `D` 作为类型指示符，指定值要作为十进制整数来显示。
- 不能使用精度指示符。

下面展示了交互式解释器中的一些例子。以下会话打印 `123456`，不进行任何特殊的格式化：

```
>>> number = 123456   
>>> print(f'{number:d}')   
123456  
>>>
```

以下会话同样打印 `123456`，但添加了千位逗号分隔符：

```
>>> number = 123456   
>>> print(f'{number:,d}')   
123,456  
>>>
```

2.10.8 指定最小域宽

格式说明符还可以包含一个**最小域宽**（minimum field width）^①，即显示一个值时应占用的最小空格数。下例在 10 个字符宽的域（字段）中显示一个整数：

```
>>> number = 99   
>>> print(f'The number is {number:10}')   
The number is          99  
>>>
```

格式说明符中的 `10` 表示应在一个至少 10 个字符宽的域中显示当前要打印的值。在本例中，要显示的值比它所在的域更短。数字 `99` 在屏幕上只占了 2 个字符的空间，但它显示在一个 10 个字符宽的域中。在这种情况下，该数字在域中右对齐，如图 2.10 所示。



注意：如果一个值过大，超过了所在域的宽度，域会自动扩大以适应它。

请注意，在前面的例子中，域宽说明符要放在逗号分隔符之前（放到逗号左侧）。下例指

^① 译注：也称为最小字段宽度。

定了域宽和精度，但没有使用逗号（千位）分隔符：

The number is 99

↑

域宽为10字符

图 2-10 域宽

下例在 12 个字符宽的域中显示一个浮点数，把它四舍五入为两位小数，并添加逗号分隔符：

```
>>> number = 12345.6789 Enter
>>> print(f'The number is {number:12,.2f}') Enter
The number is 12,345.68
>>>
```

请注意，域宽指示符要放在逗号分隔符之前（在其左侧），否则在代码执行时会报错。下例指定了域宽和精度，但没有使用逗号分隔符：

```
>>> number = 12345.6789 Enter
>>> print(f'The number is {number:12.2f}') Enter
The number is 12345.68
>>>
```

通过指定域宽，我们可以打印在一列中对齐的数字。例如，程序 2-23 在宽度为 10 个字符的两列中显示变量值。

程序 2-23 (columns.py)

```
1 # 这个程序显示了
2 # 两列数字。
3 num1 = 127.899
4 num2 = 3465.148
5 num3 = 3.776
6 num4 = 264.821
7 num5 = 88.081
8 num6 = 799.999
9
10 # 每个数字都在 10 字符宽的域中
11 # 显示，并四舍五入为两位小数。
12 print(f'{num1:10.2f}{num2:10.2f}')
13 print(f'{num3:10.2f}{num4:10.2f}')
14 print(f'{num5:10.2f}{num6:10.2f}')
```

程序输出

```
127.90    3465.15
3.78      264.82
88.08     800.00
```

2.10.9 值的对齐

一个值在比它宽的域中显示时，它必须在域中右对齐、左对齐或居中对齐。如以下交互会话所示，数字默认右对齐：

```
>>> number = 22 
>>> print(f'The number is {number:10}') 
The number is      22
>>>
```

在这个例子中，数字 22 在 10 个字符宽的域中右对齐。相反，字符串默认左对齐，如下所示：

```
>>> name = 'Jay' 
>>> print(f'Hello {name:10}. Nice to meet you.') 
Hello Jay          . Nice to meet you.
>>>
```

在这个例子中，字符串 'Jay' 在 10 个空格宽的域中左对齐。如果想更改值的默认对齐方式，可以在格式说明符中使用表 2-9 总结的几个对齐指示符之一。

表 2-9 对齐指示符

对齐指示符	含义
<	左对齐
>	右对齐
^	居中对齐

假定 `number` 变量引用一个整数。以下 f 字符串使变量的值在 10 字符宽的域中左对齐：

```
f'{number:<10d}'
```

假定 `total` 变量引用一个浮点值。以下 f 字符串使变量的值在 20 字符宽的域中右对齐，并将值四舍五入为两位小数：

```
f'{total:>20.2f}'
```

程序 2-24 展示了字符串居中对齐的一个例子。程序显示 6 个字符串，全部在 20 字符宽的域中居中。

程序 2-24 (center_align.py)

```
1 # 这个程序演示了如何使字符串居中对齐
2 name1 = 'Gordon'
3 name2 = 'Smith'
4 name3 = 'Washington'
5 name4 = 'Alvarado'
6 name5 = 'Livingston'
7 name6 = 'Jones'
8
9 # 显示名字
10 print(f'***{name1:^20}***')
11 print(f'***{name2:^20}***')
12 print(f'***{name3:^20}***')
13 print(f'***{name4:^20}***')
14 print(f'***{name5:^20}***')
15 print(f'***{name6:^20}***')
```

程序输出

```
***      Gordon      ***
***      Smith      ***
***   Washington   ***
***    Alvarado    ***
***  Livingston  ***
***      Jones      ***
```

2.10.10 指示符的顺序

在格式说明符中使用多个指示符时，它们的顺序至关重要，如下所示：

[*对齐*][*宽度*][,][*精度*][*类型*]

如果将这些指示符弄错了顺序，就会发生错误。例如，假设 `number` 变量引用了一个浮点值。以下语句在 10 个字符宽的一个域中正确居中显示该变量的值，插入逗号（千位）分隔符，并将该值四舍五入为两位小数：

```
print(f'{number:^10,.2f}')
```

但是，由于指示符顺序不当，以下语句会造成一个错误：

```
print(f'{number:10^,.2f}') # 错误
```

2.10.11 连接 f 字符串

连接（拼接）两个或更多 f 字符串时，结果也是一个 f 字符串。以下交互会话展示了一个例子：

```
1 >>> name = 'Abbie Lloyd' 
2 >>> department = '销售' 
3 >>> position = '主管' 
4 >>> print(f'员工姓名: {name}, ' + 
5         f'部门: {department}, ' + 
6         f'职位: {position}')
7 员工姓名: Abbie Lloyd, 部门: 销售, 职位: 主管
8 >>>
```

第 4、5、6 行将三个 f 字符串连接起来，并将结果作为实参传递给 `print` 函数。注意，连接的每个字符串都有 `f` 前缀。如果连接的任何一个字符串字面值遗漏了 `f` 前缀，它会被视为普通字符串，而不是 f 字符串。例如，下例延续了上一个交互会话：

```
9 >>> print(f'员工姓名: {name}, ' + 
10         '部门: {department}, ' + 
11         '职位: {position}')
12 员工姓名: Abbie Lloyd, 部门: {department}, 职位: {position}
13 >>>
```

在这个例子中，第 10 行和第 11 行的字符串字面值没有 `f` 前缀，所以它们被视为普通字符串处理。换言之，其中的占位符不具有占位符的功能。相反，它们只是作为普通文本打印到屏幕上。

可以省略加号，使用 f 字符串的隐式连接法，如下所示：

```
print(f'姓名: {name}, ' 
      f'部门: {department}, ' 
      f'职位: {position}')
```

检查点

2.28 以下代码会显示什么？

```
name = 'Karlie'
print('Hello {name}')
```

2.29 以下代码会显示什么？

```
name = 'Karlie'
print(f'Hello {name}')
```

2.30 以下代码会显示什么？

```
value = 99
print(f'The value is {value + 1}')
```

2.31 以下代码会显示什么？

```
value = 65.4321
print(f'The value is {value:.2f}')
```

2.32 以下代码会显示什么？

```
value = 987654.129
print(f'The value is {value:,.2f}')
```

2.33 以下代码会显示什么？

```
value = 9876543210
print(f'The value is {value:,d}')
```

2.34 在以下语句中，格式说明符中的数字 **10** 有什么作用？

```
print(f'{name:10}')
```

2.35 在以下语句中，格式说明符中的数字 **15** 有什么作用？

```
print(f'{number:15,d}')
```

2.36 在以下语句中，格式说明符中的数字 **8** 有什么作用？

```
print(f'{number:8,.2f}')
```

2.37 在以下语句中，格式说明符中的字符 **<** 有什么作用？

```
print(f'{number:<12d}')
```

2.38 在以下语句中，格式说明符中的字符 **>** 有什么作用？

```
print(f'{number:>12d}')
```

2.39 在以下语句中，格式说明符中的字符 **^** 有什么作用？

```
print(f'{number:^12d}')
```

2.11 具名常量

概念：具名常量是代表特殊值（例如魔法数字）的一个名称。

假设你是一名在银行工作的程序员。在更改某个现有的贷款计算程序时，你看到了下面这行代码：

```
amount= balance * 0.069
```

由于该程序是别人写的，所以你不清楚数值 `0.069` 是什么意思。它看上去像是利率，但也可能是计算某种费用的参数。总之，仅凭这一行代码，你无法确定 `0.069` 的含义。这就是所谓的魔法数字。**魔法数字**或**幻数** (magic number)是指程序代码中含义不明的数字。

魔法数字会带来多种多样的问题。首先，如上例所示，以后看代码的人很难确定它的意义。其次，如果一个魔法数字在程序中多处使用，那么修改时全部都要修改，这会为程序员带来巨大的工作量。第三，在编写程序时，每次输入幻数都可能发生打字错误。例如，本来要输入的是 `0.069`，但可能不慎输入为 `.0069`。这个失误将导致很难发现的计算错误。

解决这些问题，我们可以使用具名常量来表示魔法数字。**具名常量** (named constant) 是代表特殊值的一个名称，这种值在程序执行期间不会发生更改。下例展示了如何在代码中声明具名常量：

```
INTEREST_RATE = 0.069
```

上述代码创建名为 `INTEREST_RATE` 的一个具名常量，并赋值 `0.069`。注意，具名常量中的字母全部大写。这是大多数语言的一个标准实践，它使具名常量很容易与普通变量区分。

具名常量的一个好处是使程序更容易理解。以下语句：

```
amount = balance * 0.069
```

可以更改为：

```
amount = balance * INTEREST_RATE
```

新上手的程序员看到第二个语句时，马上就能明白它的含义：金额等于账户余额乘以利率。

采用具名常量的另一个好处是易于对程序进行大面积的修改。假设程序中有几十处用到了利率。当利率变化时，只需在声明具名常量的语句中对初始值修改一次即可。例如，如果将利率提高到 7.2%，那么只需将声明语句更改为：

```
INTEREST_RATE = 0.072
```

然后，用到常量 `INTEREST_RATE` 的每个语句都将使用新值 `0.072`。

采用具名常量还有一个好处，即防范使用魔法数字时经常出现的打字错误。例如，在编码一个数学表达式时，如果无意中将 `0.069` 输入为 `.0069`，那么程序将用错误的数据进行计算。但是，如果在输入 `INTEREST_RATE` 时发生了打字错误，Python 解释器会显示一条错误消息，提示该变量名尚未定义。

检查点

2.40 请列举使用具名常量的三个好处。

2.41 写 Python 语句为百分之十的折扣率定义一个具名常量。

2.12 海龟图形概述

概念：海龟图形是学习基本程序设计概念的一种有趣且轻松的途径。Python 海龟图形系统模拟一只“海龟”，它会遵照命令来绘制一些简单的图形。

20 世纪 60 年代后期，麻省理工学院（MIT）的西摩·佩珀特（Seymour Papert）教授用一只机器“海龟”来教学生编程。这个机器海龟听命于一台计算机，学生可以在这台计算机上输入各种命令来指挥机器海龟移动。这个机器海龟还带有一支可以抬起或放下的笔，这样就可以通过编程控制它在纸上作图。Python 的**海龟图形**（turtle graphic）系统模拟了这个机器海龟。它在屏幕上显示一个小的光标（表示机器海龟）。你可以使用 Python 语句来控制海龟在屏幕上移动并绘制线段和图形。

 视频讲解：Introduction to Turtle Graphics

使用 Python 海龟图形系统的第一步是写以下语句：

```
import turtle
```

由于 Python 解释器没有内置海龟图形，所以上述语句不可或缺。由于海龟图形系统存储在一个名为 turtle 的模块文件中，所以需要 `import turtle` 语句将该文件加载到内存，这样 Python 解释器才能使用它。

如果要编写使用了海龟图形的 Python 程序，那么务必将上述 `import` 语句放在程序的最开头。如果只是想在交互模式下体验海龟图形的乐趣，那么只需像下面这样在 Python 外壳程序中执行以下语句：

```
>>> import turtle   
>>>
```

2.12.1 使用海龟图形来画线

Python 海龟的起始位置位于被当作画布的一个图形窗口的中心。可以输入命令 `turtle.showturtle()` 在窗口中显示这只海龟。以下交互会话导入 turtle 模块并显示海龟。

```
>>> import turtle   
>>> turtle.showturtle()
```

上述语句执行后，将出现如图 2-11 所示的一个图形窗口。这里解释一下，海龟的外形与现实世界的海龟外形没有任何相似之处，它就是一个箭头形状的光标（▶）。采用箭头形状很重要，因为箭头代表了海龟面对的方向。如果命令海龟前进，那么它会朝箭头所指的方向

向移动。现在来尝试一下。你可以使用 `turtle.forward(n)` 命令让海龟向前移动 *n* 个像素（用你需要的像素数代替 *n*）。例如，`turtle.forward(200)` 命令海龟前进 200 像素。下面是在 Python 外壳程序下的一个完整会话：

```
>>> import turtle   
>>> turtle.forward(200)   
>>>
```

图 2-12 展示了上述会话后的结果。注意，海龟在前进时画了一条线。

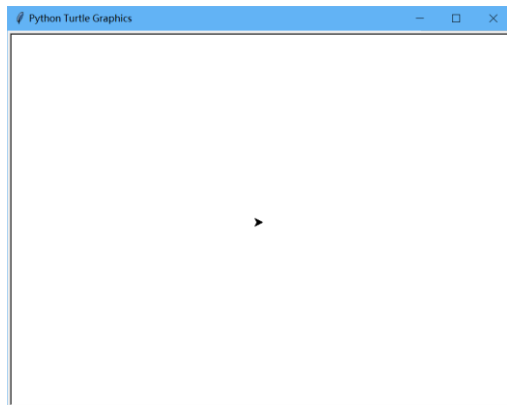


图 2-11 海龟图形窗口

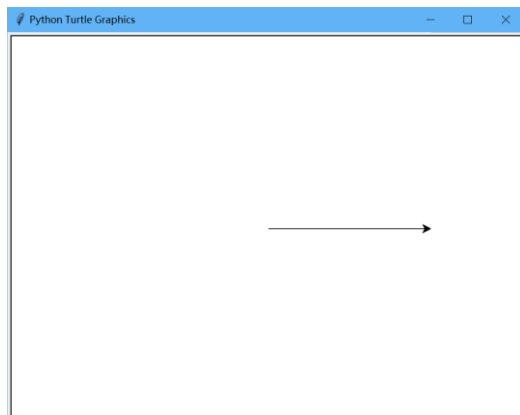


图 2-12 海龟前进 200 像素

2.12.2 海龟转向

当海龟第一次出现时，它的默认方向是 0 度（东），如图 2-13 所示。

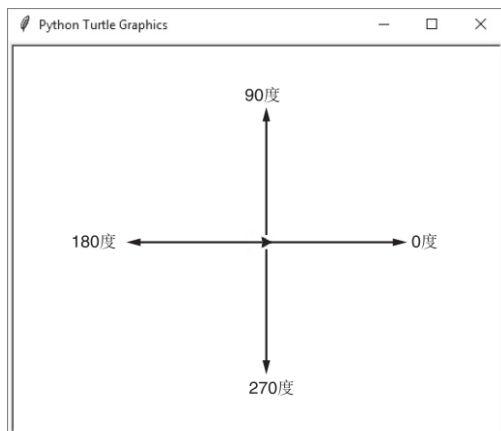


图 2.13 海龟的朝向

可以使用 `turtle.right(angle)` 或者 `turtle.left(angle)` 命令分别使海龟右转或左转 *angle* 度。以下示例会话使用了 `turtle.right(angle)` 命令：

```
>>> import turtle   
>>> turtle.forward(200)   
>>> turtle.right(90)   
>>> turtle.forward(200)   
>>>
```

在上述会话中，我们首先使海龟前进 200 像素，右转 90 度（海龟将朝下走），再前进 200 像素。输出结果如图 2-14 所示。

以下示例会话使用了 `turtle.left (angle)` 命令：

```
>>> import turtle   
>>> turtle.forward(100)   
>>> turtle.left(120)   
>>> turtle.forward(150)   
>>>
```

在上述会话中，我们首先使海龟前进 100 像素，左转 120 度（海龟将朝西北走），再前进 150 像素。输出结果如图 2-15 所示。

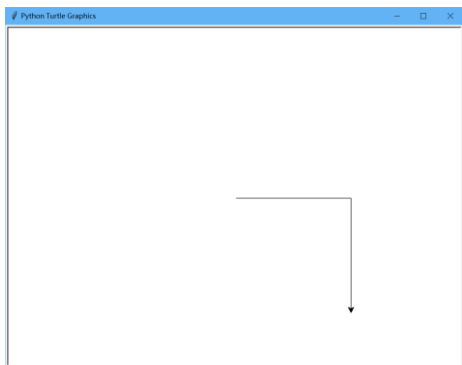


图 2-14 海龟右转

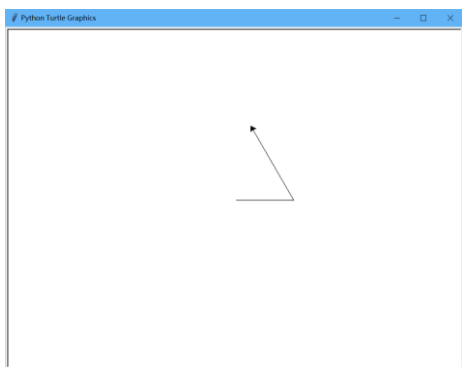


图 2-15 海龟左转

记住，`turtle.right` 和 `turtle.left` 命令会使海龟转过一个指定的角度。例如，假定海龟当前朝向 90 度（正北）。如果输入 `turtle.left(20)` 命令，那么海龟将左转 20 度。这意味着海龟将朝向 110 度。以下交互会话展示了另一个例子：

```
1 >>> import turtle 
2 >>> turtle.forward(50) 
3 >>> turtle.left(45) 
4 >>> turtle.forward(50) 
5 >>> turtle.left(45) 
6 >>> turtle.forward(50) 
7 >>> turtle.left(45) 
8 >>> turtle.forward(50) 
9 >>>
```

图 2-16 展示了上述会话的输出结果。会话开始时，海龟朝向 0 度。在第 3 行，海龟左转 45 度。在第 5 行，海龟又左转 45 度。在第 7 行，海龟再次左转 45 度。三次左转 45 度后，海龟最终朝向 135 度。

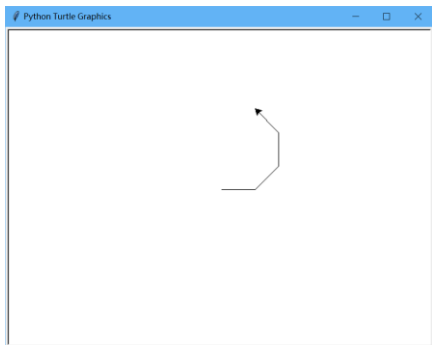


图 2-16 海龟连续左转 45 度

2.12.3 使海龟朝向指定角度

使用 `turtle.setheading(angle)` 命令，可以让海龟朝向一个指定的角度。`angle` 实参代表你希望的角度。以下交互会话对此进行了演示：

```
1 >>> import turtle   
2 >>> turtle.forward(50)   
3 >>> turtle.setheading(90)   
4 >>> turtle.forward(100)   
5 >>> turtle.setheading(180)   
6 >>> turtle.forward(50)   
7 >>> turtle.setheading(270)   
8 >>> turtle.forward(100)   
9 >>>
```

和之前一样，海龟最初的朝向是 0 度。第 3 行将它的朝向设为 90 度，第 5 行设为 180 度，第 7 行则设为 270 度。上述会话的输出结果如图 2-17 所示。

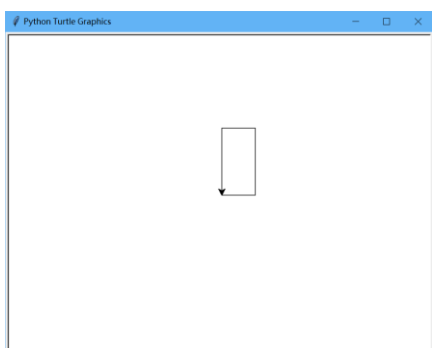


图 2-17 设置海龟的朝向

2.12.4 获取海龟的当前朝向

在交互会话中，可以使用 `turtle.heading()` 命令显示海龟的当前朝向，如下所示：


```
>>> import turtle Enter
>>> turtle.heading()Enter
0.0
>>> turtle.setheading(180) Enter
>>> turtle.heading()Enter
180.0
>>>
```

2.12.5 画笔抬起和放下

原始的机器龟是被放置在一张很大的纸上，并带有一支可以抬起和放下的画笔。当画笔被放下时，画笔与纸接触，随着机器龟的移动将绘制一条线段。当画笔被抬起时，画笔不再与纸接触，机器龟移动时不会绘制任何线段。

在 Python 中，我们将机器龟称为“海龟”。可以使用命令 `turtle.penup()` 来抬起画笔，使用命令 `turtle.pendown()` 来放下画笔。画笔抬起后，你可以随便移动海龟而不用担心它会绘制任何线段。画笔放下后，随着海龟的移动，它的身后会留下一条代表其移动轨迹的线段（画笔默认是放下的）。以下会话展示了一个例子，输出结果如图 2-18 所示。

```
>>> import turtle Enter
>>> turtle.forward(50) Enter
>>> turtle.penup()Enter
>>> turtle.forward(25) Enter
>>> turtle.pendown()Enter
>>> turtle.forward(50) Enter
>>> turtle.penup()Enter
>>> turtle.forward(25) Enter
>>> turtle.pendown()Enter
>>> turtle.forward(50) Enter
>>>
```

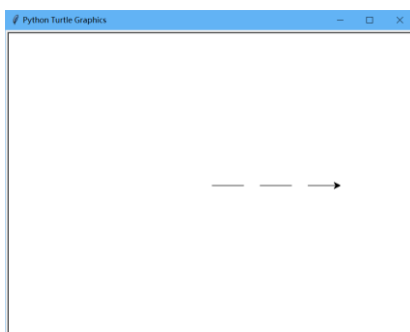


图 2-18 抬起和放下画笔

2.12.6 画圆和画点

可以使用 `turtle.circle(radius)` 命令使海龟画一个半径为 *radius* 像素的圆。例如，

`turtle.circle(100)`命令使海龟画一个半径为 100 像素的圆。以下交互会话展示了一个例子，输出结果如图 2-19 所示。

```
>>> import turtle   
>>> turtle.circle(100)   
>>>
```

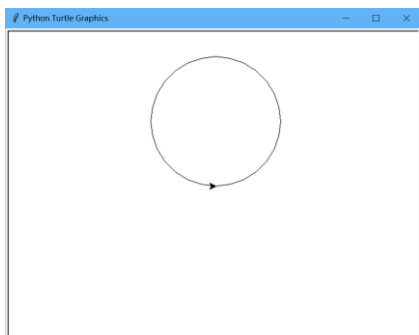


图 2-19 画圆

可以使用 `turtle.dot()`命令画一个简单的点。以下交互会话展示了一个例子，输出结果如图 2-20 所示。

```
>>> import turtle   
>>> turtle.dot()   
>>> turtle.forward(50)   
>>> turtle.dot()   
>>> turtle.forward(50)   
>>> turtle.dot()   
>>> turtle.forward(50)   
>>>
```

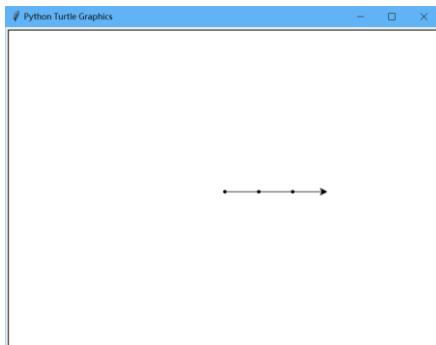


图 2-20 画点

2.12.7 更改画笔大小

可以使用 `turtle.pensize(width)` 命令更改画笔的宽度（以像素为单位）。*width* 实参是指定了画笔宽度的一个整数。例如，以下交互会话将笔宽设为 5 像素，然后画一个圆。

```
>>> import turtle   
>>> turtle.pensize(5)   
>>> turtle.circle(100)   
>>>
```

2.12.8 更改画笔颜色

可以使用 `turtle.pencolor(color)` 命令来更改画笔颜色。*color* 实参是作为字符串的一个颜色名称。例如，以下交互会话将画笔颜色设为红色（red），然后画一个圆。

```
>>> import turtle   
>>> turtle.pencolor('red')   
>>> turtle.circle(100)   
>>>
```

`turtle.pencolor` 命令支持大量预定义颜色名称，完整清单请参见附录 D。一些常用的颜色包括 'red', 'green', 'blue', 'yellow' 和 'cyan' 等。

2.12.9 更改背景颜色

可以使用 `turtle.bgcolor(color)` 命令来更改海龟图形窗口的背景颜色。*color* 实参是字符串形式的颜色名称。例如，以下交互会话将背景颜色更改为灰色（gray），将画笔颜色更改为红色（red），然后画一个圆。

```
>>> import turtle   
>>> turtle.bgcolor('gray')   
>>> turtle.pencolor('red')   
>>> turtle.circle(100)   
>>>
```

如前所述，有大量预定义颜色名称可用，附录 D 提供了详细清单。

2.12.10 重置屏幕

有三个命令可以用来重置海龟图形窗口：`turtle.reset()`，`turtle.clear()` 和 `turtle.clearscreen()`。下面总结了这些命令。

- `turtle.reset()` 命令擦除图形窗口中当前的所有绘画，将画笔颜色重置为黑色，并将海龟重置到图形窗口中心的原始位置。该命令不会重置图形窗口的背景颜色。
- `turtle.clear()` 命令仅擦除图形窗口中当前的所有绘画。它不会更改海龟的位置、画笔颜色和图形窗口的背景颜色。

- `turtle.clearscreen()` 命令擦除图形窗口中当前的所有绘画，将画笔颜色重置为黑色，将图形窗口的背景颜色重置为白色，并将海龟重置到图形窗口中心的原始位置。

2.12.11 指定图形窗口的大小

可以使用 `turtle.setup(width, height)` 命令来指定图形窗口的大小。`width` 和 `height` 实参分别是宽度和高度（以像素为单位）。例如，以下交互会话创建一个 640 像素宽、480 像素高的图形窗口。

```
>>> import turtle   
>>> turtle.setup(640, 480)   
>>>
```

2.12.12 获取海龟的当前位置

可以在交互会话中使用 `turtle.pos()` 命令显示海龟的当前位置，如下例所示：

```
>>> import turtle   
>>> turtle.goto(100, 150)   
>>> turtle.pos() (100.00, 150.00)   
>>>
```

还可以使用 `turtle.xcor()` 命令显示海龟的 X 坐标，用 `turtle.ycor()` 显示 Y 坐标，如下例所示：

```
>>> import turtle   
>>> turtle.goto(100, 150)   
>>> turtle.xcor()   
100  
>>> turtle.ycor()   
150  
>>>
```

2.12.13 控制海龟动画的速度

可以使用 `turtle.speed(speed)` 命令来更改海龟的移动速度。`speed` 实参是 0~10 的一个数字。如果设为 0，那么海龟将立即做出所有动作（动画被禁用）。例如，以下交互会话禁用海龟动画，然后画一个圆。结果是，这个圆会被立即画出。

```
>>> import turtle   
>>> turtle.speed(0)   
>>> turtle.circle(100)   
>>>
```

相反，如果指定了 1~10 的一个速度值，那么速度 1 最慢，10 最快。以下交互会话将动画速度设为 1（最慢），然后画一个圆。

```
>>> import turtle 
>>> turtle.speed(1) 
>>> turtle.circle(100) 
>>>
```

可以使用 `turtle.speed()` 命令获取当前动画速度（不提供 `speed` 实参），如下例所示：

```
>>> import turtle 
>>> turtle.speed() 
3
>>>
```

2.12.14 隐藏海龟

如果不想显示海龟，那么可以使用 `turtle.hideturtle()` 命令来隐藏它。这个命令不会改变绘图方式，它只是隐藏了海龟（箭头）图标。想再次显示海龟时，使用 `turtle.showturtle()` 命令即可。

2.12.15 将海龟移到指定位置

如图 2-21 所示，我们用**笛卡尔坐标系**（Cartesian coordinate system）来确定海龟图形窗口中每个像素的位置。每个像素都有一个 X 坐标和一个 Y 坐标。 X 坐标确定像素的水平位置， Y 坐标确定垂直位置。下面是需要注意的重点：

- 图形窗口中心像素位于 $(0, 0)$ ，这意味着它的 X 坐标为 0， Y 坐标为 0。
- 向窗口右侧移动， X 坐标值增大；向窗口左侧移动， X 坐标值减小。
- 向窗口顶部移动， Y 坐标值增大；向窗口底部移动， Y 坐标值减小。
- 位于中心点左侧的像素具有正的 X 坐标值，位于中心点右侧的像素具有负的 X 坐标值。
- 位于中心点上方的像素具有正的 Y 坐标值，位于中心点下方的像素具有负的 Y 坐标值。

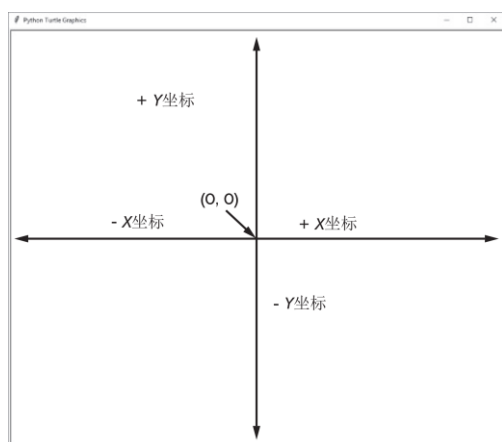


图 2-21 笛卡尔坐标系

可以使用 `turtle.goto(x, y)` 命令将海龟从当前位置移动到图形窗口中的一个特定位置。实参 `x` 和 `y` 是目标位置的坐标。如果画笔是放下的，海龟移动时会画一条线段。例如，以下交互会话的绘图结果如图 2-22 所示。

```
>>> import turtle   
>>> turtle.goto(0, 100)   
>>> turtle.goto(-100, 0)   
>>> turtle.goto(0, 0)   
>>>
```

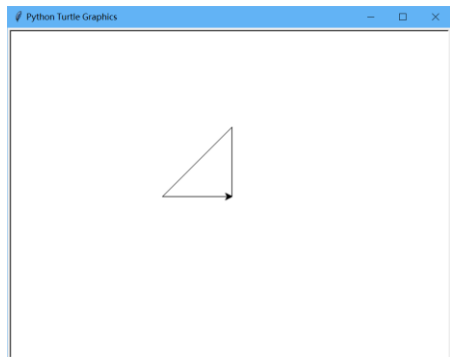


图 2-22 移动海龟

2.12.16 在图形窗口中显示文本

可以使用 `turtle.write(text)` 命令在图形窗口中显示文本。其中，`text` 是你想要显示的字符串。显示字符串时，它的第一个字符的左下角将被定位在海龟的 `X` 和 `Y` 坐标上。以下交互会话进行了演示，输出结果如图 2-23 所示。

```
>>> import turtle   
>>> turtle.write('你好!')   
>>>
```

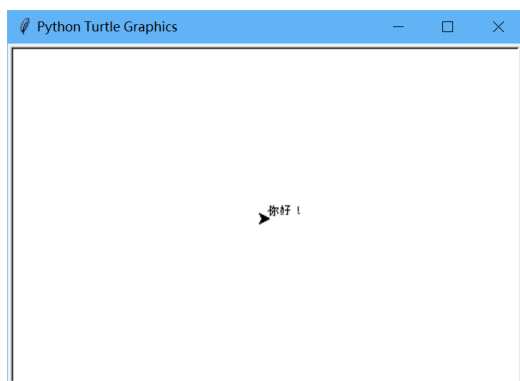


图 2-23 在图形窗口中显示文本

以下交互会话将海龟移至指定位置再显示文本，输出结果如图 2-24 所示。

```
>>> import turtle   
>>> turtle.setup(300, 300)   
>>> turtle.penup()   
>>> turtle.hideturtle()   
>>> turtle.goto(-120, 120)   
>>> turtle.write("左上方")   
>>> turtle.goto(70, -120)   
>>> turtle.write("右下方")   
>>>
```



图 2-24 在图形窗口的指定位置显示文本

2.12.17 填充形状

要为一个形状填充颜色，可以在绘制形状之前使用 `turtle.begin_fill()` 命令，并在形状绘制完成后使用 `turtle.end_fill()` 命令。执行 `turtle.end_fill()` 命令时，形状会被填充当前的填充颜色。以下交互会话对此进行了演示，输出结果如图 2-25 所示。

```
>>> import turtle   
>>> turtle.hideturtle()   
>>> turtle.begin_fill()   
>>> turtle.circle(100)   
>>> turtle.end_fill()   
>>>
```

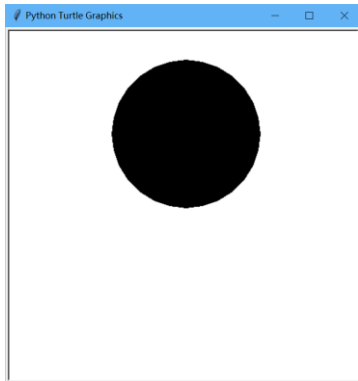


图 2-25 一个填充圆

图 2-25 的圆用黑色填充，这是默认填充颜色。可以使用 `turtle.fillcolor(color)` 命令来更改填充颜色。其中，`color` 实参是字符串形式的一个颜色名称。例如，以下交互会话将填充颜色更改为红色，然后画一个圆。

```
>>> import turtle   
>>> turtle.hideturtle()  
>>> turtle.fillcolor('red')   
>>> turtle.begin_fill()  
>>> turtle.circle(100)   
>>> turtle.end_fill()  
>>>
```

`turtle.fillcolor` 命令支持大量预定义颜色名称，完整清单请参见附录 D。其中一些常用的颜色包括 'red', 'green', 'blue', 'yellow' 和 'cyan' 等。

以下交互会话演示了如何画一个正方形，然后用蓝色填充，输出结果如图 2-26 所示。

```
>>> import turtle   
>>> turtle.hideturtle()  
>>> turtle.fillcolor('blue')   
>>> turtle.begin_fill()  
>>> turtle.forward(100)   
>>> turtle.left(90)   
>>> turtle.forward(100)   
>>> turtle.left(90)   
>>> turtle.forward(100)   
>>> turtle.left(90)   
>>> turtle.forward(100)   
>>> turtle.end_fill()  
>>>
```

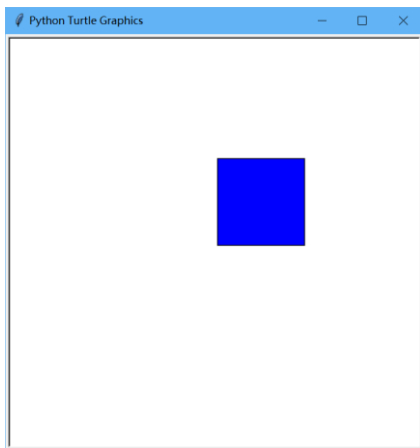



图 2-26 一个填充正方形

如果绘制的形状没有封闭，那么在填充时就好像你在起点和终点之间连了一条线。例如，以下交互会话只画了两条线。第一条从(0, 0)到(120, 120)，第二条从(120, 120)到(200, -100)。执行 `turtle.end_fill()` 命令时，所填充的形状就好像存在一条从(0, 0)到(200, -120)的线。图 2-27 展示了该会话的输出。

```
>>> import turtle   
>>> turtle.hideturtle()   
>>> turtle.begin_fill()   
>>> turtle.goto(120, 120)   
>>> turtle.goto(200, -100)   
>>> turtle.end_fill()   
>>>
```

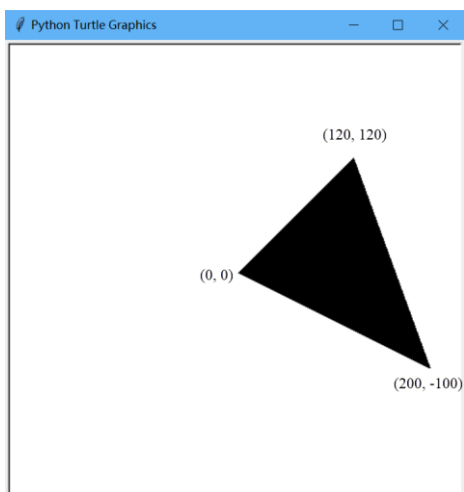


图 2-27 通过连接起点和终点来补完形状并填充

2.12.18 从对话框获取输入

可以使用 `turtle.numinput` 命令获取用户输入的一个数值，并将其赋给一个变量。`turtle.numinput` 命令显示一个称为**对话框**（dialog box）的小图形窗口。对话框包含一个供用户输入的区域，以及一个 OK（确定）按钮和一个 Cancel（取消）按钮。图 2-28 展示了一个例子。

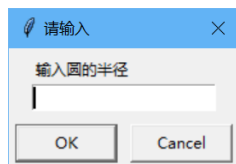


图 2-28 对话框

下面是使用 `turtle.numinput` 命令的语句的常规形式：

```
variable = turtle.numinput(title, prompt)
```

其中，`variable` 是一个变量的名称，用户输入的数值会赋给它。`title` 实参是在对话框标题栏（窗口顶部的栏）显示的一个字符串，`prompt` 实参是在对话框内部显示的一个字符串，用于指示用户输入一些数据。当用户单击 OK 按钮时，该命令会返回用户在对话框中输入的数值（以浮点数形式）并赋给 `variable`。

以下交互会话对此进行了演示，会话的输出如图 2-29 所示。

```
>>> import turtle 
>>> radius = turtle.numinput('请输入', '输入圆的半径') 
>>> turtle.circle(radius) 
>>>
```

交互会话的第二个语句会显示图 2-29 左半边的对话框。在示例会话中，用户在对话框中输入 `100`，并单击 OK 按钮。结果是 `turtle.numinput` 命令返回值 `100.0` 并赋给 `radius` 变量。会话的下一个语句执行 `turtle.circle` 命令，将 `radius` 变量作为实参传递，结果是绘制一个半径为 `100` 的圆（图 2-29 的右半边）。

如果用户单击 Cancel 按钮而不是 OK 按钮，`turtle.numinput` 命令会返回特殊值 `None`，表示用户没有输入。

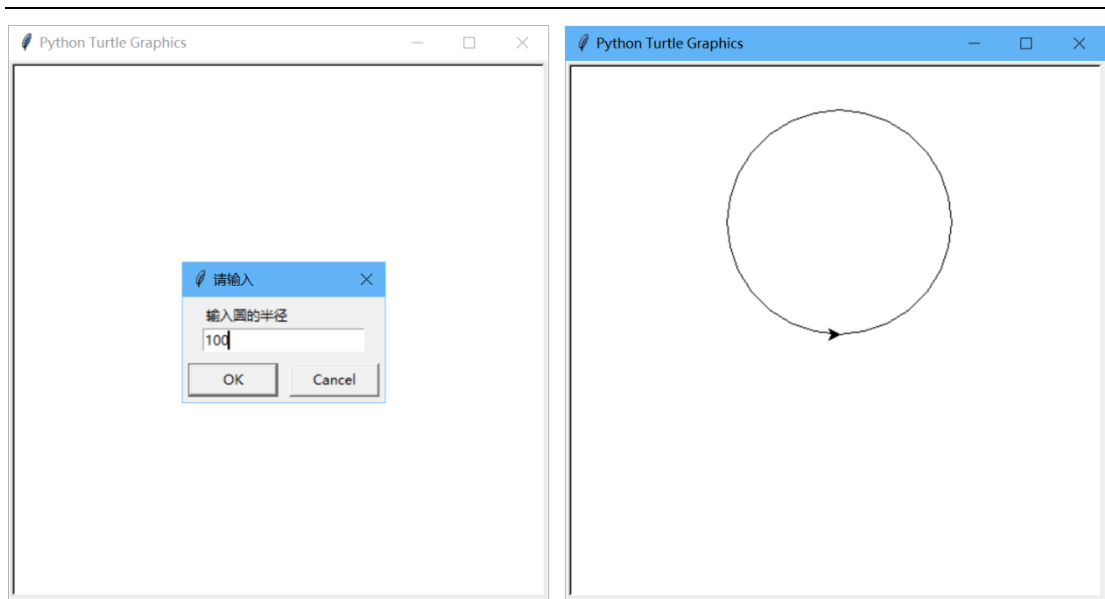


图 2-29 提示用户输入圆的半径

除了 *title* 和 *prompt* 参数，`turtle.numinput` 命令还能接收三个可选参数，下面是常规形式：

```
variable = turtle.numinput(title, prompt, default=x, minval=y, maxval=z)
```

- `default=x` 指定在输入框中显示的默认值 *x*，方便用户直接单击 **OK** 按钮并接受默认值。
- `minval=y` 指定允许用户输入的最小值。如果用户输入的数字小于 *y*，将显示一条错误消息，对话框仍将开启。
- `maxval=z` 指定允许用户输入的最大值，如果用户输入的数字大于 *z*，将显示一条错误消息，对话框仍将开启。

下面是一个使用了所有可选参数的语句的例子：

```
num = turtle.numinput('请输入', '输入范围在 1~10 之间的值',  
                      default=5, minval=1, maxval=10)
```

该语句指定默认值 5，最小值 1，最大值 10。图 2-30 展示了该语句所显示的对话框。如果用户输入一个小于 1 的值并单击 **OK**，系统将显示图 2-31 左半边的消息框。如果输入大于 10 的值并单击 **OK**，则将显示图 2-31 右半边的消息框。

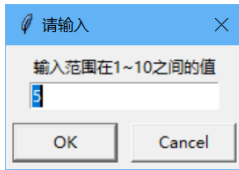


图 2-30 对话框显示了默认值

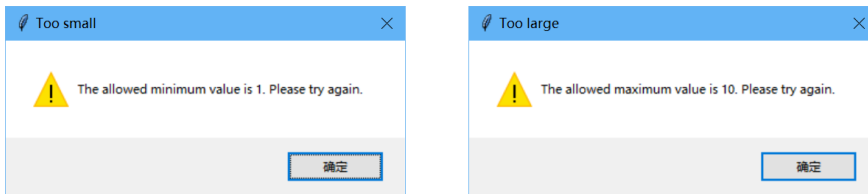


图 2-31 输入超出范围的值时显示的错误消息

2.12.19 使用 `turtle.textinput` 命令获取字符串输入

还可以使用 `turtle.textinput` 命令来获取用户输入的字符串。下面是使用该命令的语句的常规形式：

```
variable = turtle.textinput(title, prompt)
```

`turtle.textinput` 命令的工作方式和 `turtle.numinput` 命令相似，只是它将用户的输入作为字符串返回。以下交互会话对此进行了演示，第二个语句显示的对话框如图 2-32 所示。

```
>>> import turtle Enter
>>> name = turtle.textinput('请输入', '输入你的姓名') Enter
>>> print(name) Enter
张三丰
>>>
```

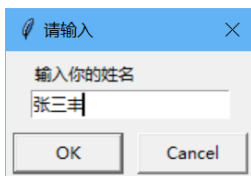


图 2-32 对话框提示输入一个姓名

2.12.20 使用 `turtle.done()` 使图形窗口保持打开

如果在 IDLE 以外的环境中运行一个 Python 海龟图形程序（例如，在命令行），那么可能会注意到程序一结束，图形窗口就会立即消失。为了防止窗口在程序结束后关闭，需要在海龟图形程序的最后加入一个 `turtle.done()` 语句。这会使图形窗口保持打开状态，以便

在程序执行完毕后观看其内容。要关闭该窗口，单击窗口的标准“关闭”按钮即可。

如果从 IDLE 运行程序，就没有必要在程序中添加 `turtle.done()` 语句。

聚光灯：猎户星座程序

猎户座（Orion）是夜空中最著名的星座之一。图 2-33 展示了该星座中几颗星星的大致位置。最上面的星星是猎户座的肩膀，中间一排三颗星是猎户座的腰带，最下面的两颗星是猎户座的膝盖。图 2-34 展示了这些星星的名称，图 2-35 展示了通常用来连接这些星星的线段。

我们将开发一个程序来显示如图 2-35 所示的星星、星星名称和星座连线。该程序将在一个宽 500 像素、高 600 像素的图形窗口中显示星座，用圆点代表不同的星星。我们先用一张坐标纸（如图 2-36 所示）画出这些点的位置并确定其坐标。

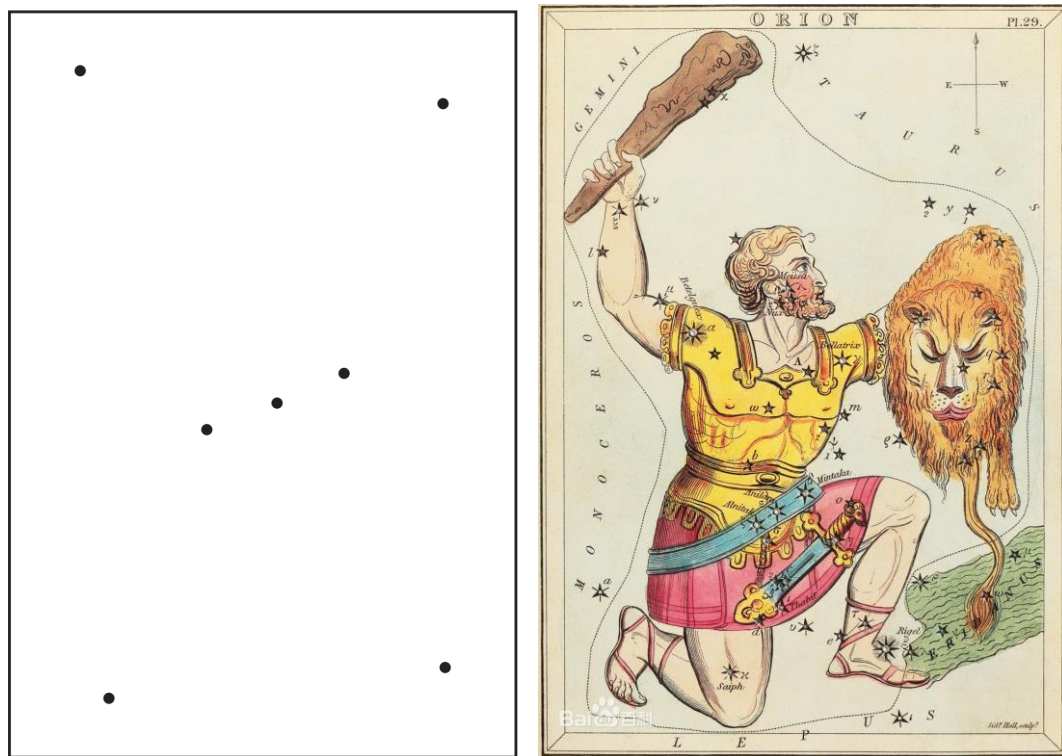


图 2-33 猎户座的星星

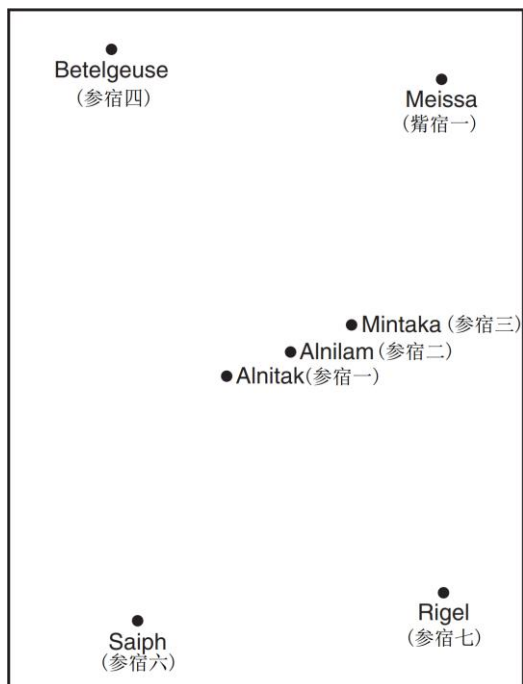


图 2-34 星星的名称

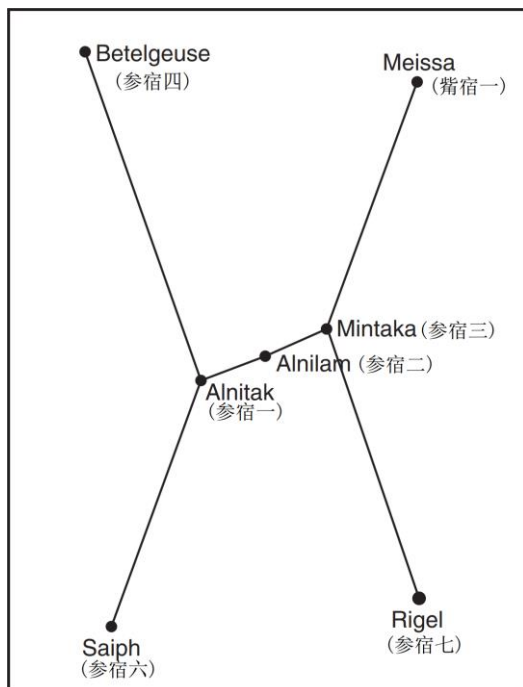


图 2-35 星座连线

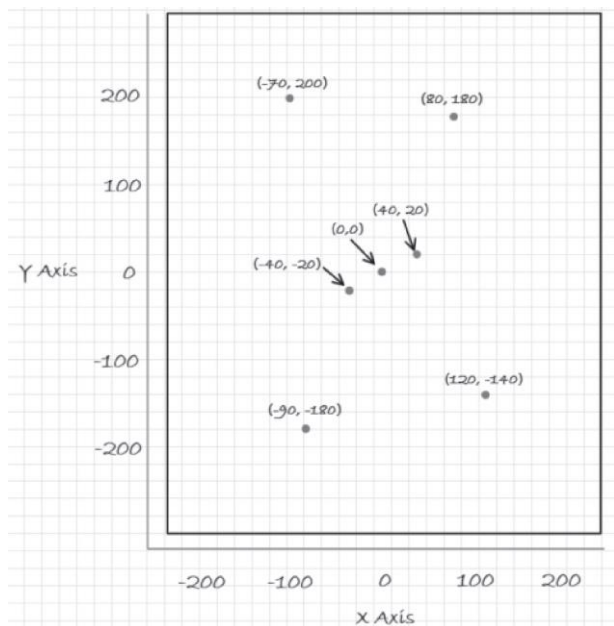


图 2-36 猎户座手绘

程序中需要多次用到图 2-36 所确定的坐标。显然，每次都手动输入每颗星星的正确坐标非常繁琐，而且容易出错。为了简化编码，我们将创建以下具名常量来表示每颗星星的坐标。

```
LEFT_SHOULDER_X = -70
LEFT_SHOULDER_Y = 200

RIGHT_SHOULDER_X = 80
RIGHT_SHOULDER_Y = 180

LEFT_BELTSTAR_X = -40
LEFT_BELTSTAR_Y = -20

MIDDLE_BELTSTAR_X = 0
MIDDLE_BELTSTAR_Y = 0

RIGHT_BELTSTAR_X = 40
RIGHT_BELTSTAR_Y = 20

LEFT_KNEE_X = -90
LEFT_KNEE_Y = -180

RIGHT_KNEE_X = 120
RIGHT_KNEE_Y = -140
```

现在，我们已经确定了星星的坐标，并创建了具名常量来表示这些坐标，接着开始为程序

的第一部分编写伪代码，即显示星星。

将图形窗口大小设为宽 500、高 600 像素。

```
在(LEFT_SHOULDER_X, LEFT_SHOULDER_Y)画一个点 # 左肩  
在(RIGHT_SHOULDER_X, RIGHT_SHOULDER_Y)画一个点 # 右肩  
在(LEFT_BELTSTAR_X, LEFT_BELTSTAR_Y)画一个点 # 腰带最左边的星星  
在(MIDDLE_BELTSTAR_X, MIDDLE_BELTSTAR_Y)画一个点 # 腰带中间的星星  
在(RIGHT_BELTSTAR_X, RIGHT_BELTSTAR_Y)画一个点 # 腰带最右边的星星  
在(LEFT_KNEE_X, LEFT_KNEE_Y)画一个点 # 左膝  
在(RIGHT_KNEE_X, RIGHT_KNEE_Y)画一个点 # 右膝
```

然后，我们显示每颗星星的名称，如图 2-37 的草图所示。下面是显示这些名称的伪代码。

```
在(LEFT_SHOULDER_X, LEFT_SHOULDER_Y)显示文本“Betelgeuse(参宿四)” # 左肩  
在(RIGHT_SHOULDER_X, RIGHT_SHOULDER_Y)显示文本“Meissa(觜宿一)” # 右肩  
在(LEFT_BELTSTAR_X, LEFT_BELTSTAR_Y)显示文本“Alnitak(参宿一)” # 腰带最左边的星星  
在(MIDDLE_BELTSTAR_X, MIDDLE_BELTSTAR_Y)显示文本“Alnilam(参宿二)” # 腰带中间的星星  
在(RIGHT_BELTSTAR_X, RIGHT_BELTSTAR_Y)显示文本“Mintaka(参宿三)” # 腰带最右边的星星  
在(LEFT_KNEE_X, LEFT_KNEE_Y)显示文本“Saiph(参宿六)” # 左膝  
在(RIGHT_KNEE_X, RIGHT_KNEE_Y)显示文本“Rigel(参宿七)” # 右膝
```

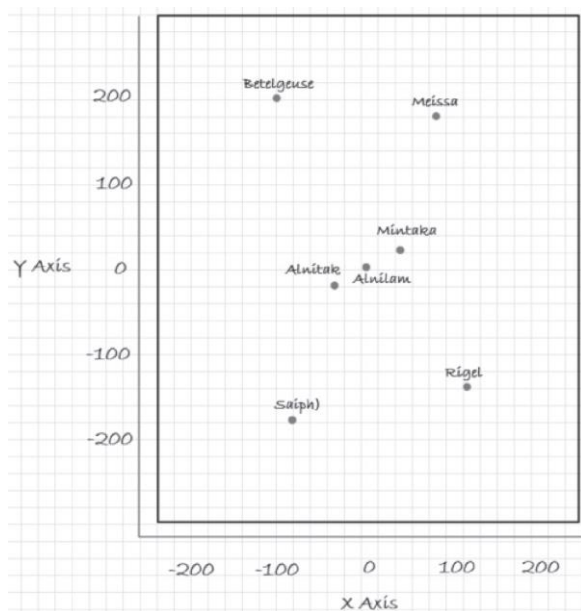


图 2-37 有星星名称的猎户座草图

接下来，我们将显示连接星星的线，如图 2-38 所示。下面是显示这些连线的伪代码。

```
# 左肩到腰带左边的星星  
从(LEFT_SHOULDER_X, LEFT_SHOULDER_Y)到(LEFT_BELTSTAR_X, LEFT_BELTSTAR_Y)画一条线  
  
# 右肩到腰带右边的星星
```

从(RIGHT_SHOULDER_X, RIGHT_SHOULDER_Y)到(RIGHT_BELTSTAR_X, RIGHT_BELTSTAR_Y)画一条线

腰带左边的星星到腰带中间的星星

从(LEFT_BELTSTAR_X, LEFT_BELTSTAR_Y)到(MIDDLE_BELTSTAR_X, MIDDLE_BELTSTAR_Y)画一条线

腰带中间的星星到腰带右边的星星

从(MIDDLE_BELTSTAR_X, MIDDLE_BELTSTAR_Y)到(RIGHT_BELTSTAR_X, RIGHT_BELTSTAR_Y)画一条线

腰带左边的星星到左膝

从(LEFT_BELTSTAR_X, LEFT_BELTSTAR_Y)到(LEFT_KNEE_X, LEFT_KNEE_Y)画一条线

腰带右边的星星到右膝

从(RIGHT_BELTSTAR_X, RIGHT_BELTSTAR_Y)到(RIGHT_KNEE_X, RIGHT_KNEE_Y)画一条线

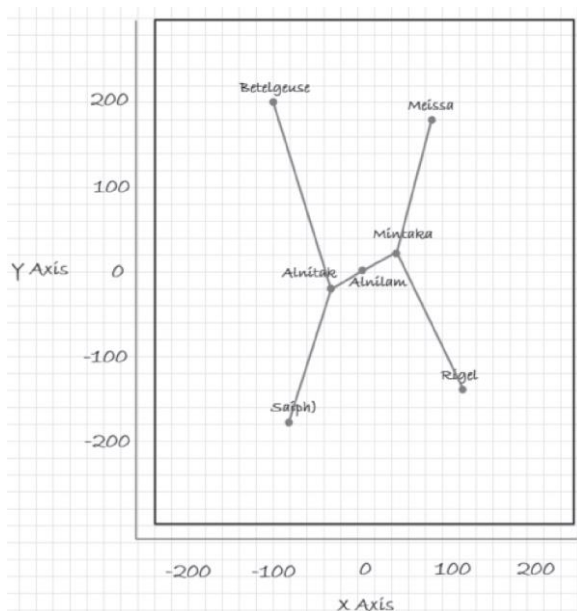


图 2-38 有星星名称和星座连线的猎户座草图

有了这些伪代码，我们就掌握了程序必须执行的逻辑步骤，接着可以开始写代码了。程序 2-25 展示了完整的程序。当程序运行时，它首先显示星星，然后显示星星的名称，最后显示星座连线。图 2-39 是该程序的输出。

程序 2-25 (orion.py)

```
1 # 这个程序绘制了猎户座的星星、  
2 # 星星的名称以及星座连线。  
3 import turtle  
4
```

```
5 # 设置窗口大小
6 turtle.setup(500, 600)
7
8 # 设置海龟
9 turtle.penup() # 笔抬起
10 turtle.hideturtle() #隐藏海龟
11
12 # 为星星的坐标创建具名常量
13 LEFT_SHOULDER_X = -70
14 LEFT_SHOULDER_Y = 200
15
16 RIGHT_SHOULDER_X = 80
17 RIGHT_SHOULDER_Y = 180
18
19 LEFT_BELTSTAR_X = -40
20 LEFT_BELTSTAR_Y = -20
21
22 MIDDLE_BELTSTAR_X = 0
23 MIDDLE_BELTSTAR_Y = 0
24
25 RIGHT_BELTSTAR_X = 40
26 RIGHT_BELTSTAR_Y = 20
27
28 LEFT_KNEE_X = -90
29 LEFT_KNEE_Y = -180
30
31 RIGHT_KNEE_X = 120
32 RIGHT_KNEE_Y = -140
33
34 # 绘制星星
35 turtle.goto(LEFT_SHOULDER_X, LEFT_SHOULDER_Y) # 左肩
36 turtle.dot()
37 turtle.goto(RIGHT_SHOULDER_X, RIGHT_SHOULDER_Y) # 右肩
38 turtle.dot()
39 turtle.goto(LEFT_BELTSTAR_X, LEFT_BELTSTAR_Y) # 腰带最左边的星星
40 turtle.dot()
41 turtle.goto(MIDDLE_BELTSTAR_X, MIDDLE_BELTSTAR_Y) # 腰带中间的星星
42 turtle.dot()
43 turtle.goto(RIGHT_BELTSTAR_X, RIGHT_BELTSTAR_Y) # 腰带最右边的星星
44 turtle.dot()
45 turtle.goto(LEFT_KNEE_X, LEFT_KNEE_Y) # 左膝
46 turtle.dot()
47 turtle.goto(RIGHT_KNEE_X, RIGHT_KNEE_Y) # 右膝
48 turtle.dot()
49
50 # 显示星星名称
51 turtle.goto(LEFT_SHOULDER_X, LEFT_SHOULDER_Y) # 左肩
52 turtle.write('Betegouse(参宿四)')
53 turtle.goto(RIGHT_SHOULDER_X, RIGHT_SHOULDER_Y) # 右肩
54 turtle.write('Meissa(觜宿一)')
```

```
55 turtle.goto(LEFT_BELTSTAR_X, LEFT_BELTSTAR_Y) # 腰带最左边的星星
56 turtle.write('Alnitak(参宿一)')
57 turtle.goto(MIDDLE_BELTSTAR_X, MIDDLE_BELTSTAR_Y) # 腰带中间的星星
58 turtle.write('Alnilam(参宿二)')
59 turtle.goto(RIGHT_BELTSTAR_X, RIGHT_BELTSTAR_Y) # 腰带最右边的星星
60 turtle.write('Mintaka(参宿三)')
61 turtle.goto(LEFT_KNEE_X, LEFT_KNEE_Y) # 左膝
62 turtle.write('Saiph(参宿六)')
63 turtle.goto(RIGHT_KNEE_X, RIGHT_KNEE_Y) # 右膝
64 turtle.write('Rigel(参宿七)')
65
66 # 从左肩到腰带左边的星星画一条线
67 turtle.goto(LEFT_SHOULDER_X, LEFT_SHOULDER_Y)
68 turtle.pendown()
69 turtle.goto(LEFT_BELTSTAR_X, LEFT_BELTSTAR_Y)
70 turtle.penup()
71
72 # 从右肩到腰带右边的星星画一条线
73 turtle.goto(RIGHT_SHOULDER_X, RIGHT_SHOULDER_Y)
74 turtle.pendown()
75 turtle.goto(RIGHT_BELTSTAR_X, RIGHT_BELTSTAR_Y)
76 turtle.penup()
77
78 # 从腰带左边的星星到腰带中间的星星画一条线
79 turtle.goto(LEFT_BELTSTAR_X, LEFT_BELTSTAR_Y)
80 turtle.pendown()
81 turtle.goto(MIDDLE_BELTSTAR_X, MIDDLE_BELTSTAR_Y)
82 turtle.penup()
83
84 # 从腰带中间的星星到腰带右边的星星画一条线
85 turtle.goto(MIDDLE_BELTSTAR_X, MIDDLE_BELTSTAR_Y)
86 turtle.pendown()
87 turtle.goto(RIGHT_BELTSTAR_X, RIGHT_BELTSTAR_Y)
88 turtle.penup()
89
90 # 从腰带左边的星星到左膝画一条线
91 turtle.goto(LEFT_BELTSTAR_X, LEFT_BELTSTAR_Y)
92 turtle.pendown()
93 turtle.goto(LEFT_KNEE_X, LEFT_KNEE_Y)
94 turtle.penup()
95
96 # 从腰带右边的星星到右膝画一条线
97 turtle.goto(RIGHT_BELTSTAR_X, RIGHT_BELTSTAR_Y)
98 turtle.pendown()
99 turtle.goto(RIGHT_KNEE_X, RIGHT_KNEE_Y)
100
101 # 保持窗口的打开状态(若使用 IDLE 则不必要)
102 turtle.done()
```

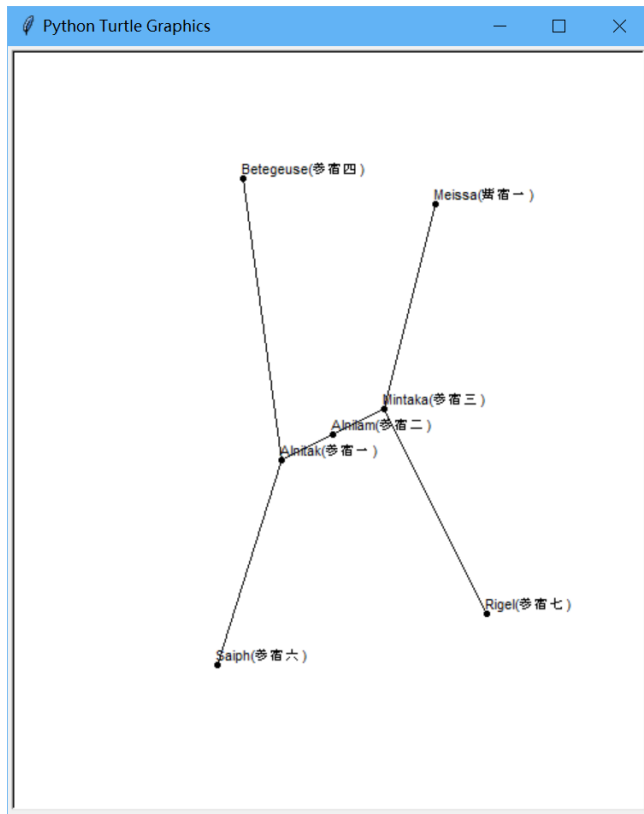


图 2-39 orion.py 程序的输出

检查点

- 2.42 海龟刚开始出现时，它默认朝向哪个方向？
- 2.43 如何使海龟前进？
- 2.44 如何使海龟右转 45 度？
- 2.45 如何在不画线的情况下将海龟移动到一个新位置？
- 2.46 用什么命令来显示海龟的当前朝向？
- 2.47 用什么命令来画一个半径为 100 像素的圆？
- 2.48 用什么命令来将海龟画笔的大小(宽度)更改为 8 像素？
- 2.49 用什么命令将海龟的画笔颜色更改为蓝色？
- 2.50 用什么命令将海龟图形窗口的背景颜色设为黑色？

-
- 2.51 用什么命令将海龟图形窗口的大小设置为宽 500 像素、高 200 像素？
- 2.52 用什么命令将海龟移动到位置(100, 50)？
- 2.53 用什么命令来显示海龟当前位置的坐标？
- 2.54 以下哪个命令造成更快的动画速度：`turtle.speed(1)`还是 `turtle.speed(10)`？
- 2.55 用什么命令来禁用海龟动画？
- 2.56 请描述如何绘制一个形状并填色。
- 2.57 如何在海龟图形窗口中显示文本？
- 2.58 写一个海龟图形语句来显示对话框，要求用户输入一个数字。在对话框的标题栏中显示“输入一个数值”，在对话框内部显示提示消息：“圆的半径是多大？”将用户在对话框中输入的数值赋给一个名为 `radius` 的变量。

复习题

选择题

- _____ 错误虽然不会阻止程序运行，但会造成不正确的结果。
a. 语法 b. 硬件 c. 逻辑 d. 致命
- _____ 是为了满足客户的要求，程序必须执行的一项任务。
a. 任务 b. 软件需求 c. 前提条件 d. 谓词/断言
- _____ 是为了执行任务而必须采取的一组明确定义的逻辑步骤。
a. 规范 b. 行动计划 c. 逻辑计划 d. 算法
- _____ 是一种非正式的语言，它没有语法规则，目的也不是被编译或执行
a. 假码 b. 伪码 c. Python d. 流程图
- _____ 图示了程序中采取的步骤。
a. 流程图 b. 步骤图 c. 代码图 d. 程序图
- _____ 是字符的序列。
a. 字符系列 b. 字符集 c. 字符串 d. 文本块

7. _____是引用了计算机内存中的一个值的名称。

- a. 变量 b. 寄存器 c. RAM d. 字节

8. _____是一名假想的、任何使用程序并为其提供输入的人。

- a. 设计者 b. 用户 c. 小白鼠 d. 测试主体

9. Python 中的字符串必须用_____括起来。

- a. 括号 b. 单引号 c. 双引号 d. 单引号或双引号

10. _____是放置在程序不同部分的简短说明，用于解释程序的这些部分如何工作。

- a. 注释 b. 参考资料 c. 教程 d. 外部文档

11. _____使变量引用计算机内存中的一个值。

- a. 变量声明 b. 赋值语句 c. 数学表达式 d. 字符串字面值

12. _____符号标记一条 Python 注释的开始。

- a. & b. * c. ** d. #

13. 以下哪个语句会造成程序出错? _____

- a. `x = 17` b. `17 = x` c. `x = 99999` d. `x = '17'`

14. 在表达式 `12 + 7` 中，加号左右两侧的值称为_____。

- a. 操作数 b. 操作符 c. 实参 d. 数学表达式

15. _____操作符执行整数除法。

- a. `//` b. `%` c. `**` d. `/`

16. _____操作符执行乘方运算。

- a. `%` b. `*` c. `**` d. `/`

17. _____操作符执行除法运算，但不返回商而是返回余。

- a. `%` b. `*` c. `**` d. `/`

18. 执行了语句 `price = 99.0` 后，`price` 变量将引用什么数据类型的一个值? _____

- a. `int` b. `float` c. `currency` d. `str`

19. 我们用内置函数_____读取键盘输入。

a. `input()` b. `get_input()` c. `read_input()` d. `keyboard()`

20. 我们用内置函数_____将 `int` 值转换成 `float` 值。

a. `int_to_float()` b. `float()` c. `convert()` d. `int()`

21. 魔法数字或幻数是指_____。

- a. 在数学上未定义的数字。
- b. 程序代码中含义不明的数字。
- c. 不能被 1 除的数字。
- d. 会导致计算机崩溃的数字。

22. _____是代表特殊值的名称，这种值在程序执行期间不会发生更改。

a. 具名字面值 b. 具名常量 c. 变量签名 d. 关键字

判断题

- 1. 程序员在编写伪代码时必须注意不要犯语法错误。
- 2. 在数学表达式中，乘法和除法发生在加法和减法之前。
- 3. 变量名称中可以有空格。
- 4. 在 Python 中，变量名的第一个字符不能是数字。
- 5. 如果打印一个未被赋值的变量，将显示数字 0。

简答题

- 1. 一个专业的程序员通常首先做什么来获得对问题的理解？
- 2. 什么是伪代码？
- 3. 计算机程序通常执行哪三个步骤的操作？
- 4. 如果数学表达式将一个 `float` 加到一个 `int` 上，那么结果数据类型是什么？
- 5. 浮点除法和整数除法的区别是什么？
- 6. 什么是魔法数字？这种数字会有什么问题？
- 7. 假设一个程序使用常量 `PI` 来表示值 `3.14159`，并在多个语句中使用该常量。在每个语句中使用具名常量而不是字面值 `3.14159` 有什么好处？

算法工作台

1. 写 Python 代码来提示用户输入他或她的身高，并将用户的输入赋给一个名为 `height` 的变量。
2. 写 Python 代码，提示用户输入他或她最喜欢的颜色，并将用户的输入赋给一个名为 `color` 的变量。
3. 写赋值语句来对变量 `a`、`b` 和 `c` 执行以下操作：
 - a. 将 2 加到 `a` 上，结果赋给 `b`
 - b. `b` 乘以 4，结果赋给 `a`
 - c. `a` 除以 3.14，结果赋给 `b`
 - d. `b` 减去 8，结果赋给 `a`
4. 假设变量 `result`、`w`、`x`、`y` 和 `z` 的值都是整数，并且 `w=5`、`x=4`、`y=8`、`z=2`。执行以下每个语句后，`result` 的值是什么？
 - a. `result = x + y`
 - b. `result = z * 2`
 - c. `result = y / x`
 - d. `result = y - z`
 - e. `result = w // z`
5. 写 Python 语句将 10 和 14 之和赋给变量 `total`。
6. 写 Python 语句从变量 `total` 中减去 `down_payment`，将结果赋给变量 `due`。
7. 写 Python 语句使变量 `subtotal` 乘以 0.15，将结果赋给变量 `total`。
8. 以下语句的显示结果是什么？

```
a = 5
b = 2
c = 3
result = a + b * c
print(result)
```
9. 以下语句的显示结果是什么？

```
num = 99
num = 5
print(num)
```

10. 假定变量 `sales` 引用了一个 `float` 值。写一个语句来显示四舍五入为两位小数的结果。

11. 假定已执行了以下语句：

```
number = 1234567.456
```

写一个 Python 语句来显示 `number` 变量引用的值，并格式化为 `1,234,567.5`。

12. 以下语句的显示结果是什么？

```
print('George', 'John', 'Paul', 'Ringo', sep='@')
```

13. 写一个海龟图形语句来画半径为 75 像素的圆。

14. 写海龟图形语句来画边长为 100 像素的一个正方形，并用蓝色填充。

15. 写海龟图形语句来画边长为 100 像素的一个正方形，再以正方形的中心为圆心画半径为 80 像素的一个圆。用红色填充圆，正方形不填充颜色。

编程练习

1. 个人资料

写一个程序来显示以下个人资料：

- 姓名
- 地址，包含省/州、市和 ZIP
- 电话号码
- 本科专业

2. 销售预测

 视频讲解：The Sales Prediction Problem

某公司的年利润通常为年销售额的 23%。编写一个程序，要求用户输入预计的年销售额，然后显示利润金额。

提示：用数值 0.23 代表 23%。

3. 土地计算

一英亩的土地相当于 43 560 平方英尺。编写一个程序，要求用户输入一块土地的总平方英

尺数，计算这块土地的英亩数。

提示：用输入的值除以 43 560 即可得到英亩数。

4. 购买总量

一个顾客在商店里购买了 5 件商品。编写一个程序，要求用户输入每件商品的价格，然后显示价格小计、销售税以及总额。假定销售税为 7%。

5. 行驶距离

假设没有事故或延误，汽车的行驶距离可以用以下公式计算：

$$\text{距离} = \text{速率} \times \text{时间}$$

一辆汽车以每小时 70 英里的速度行驶。写一个程序来显示以下数据：

- 汽车 6 小时将行驶多少英里
- 汽车 10 小时将行驶多少英里
- 汽车 15 小时将行驶多少英里

6. 销售税

写一个程序，要求用户输入购物金额。然后，程序应计算出州和郡县销售税。假设州销售税是 5%，郡县销售税是 2.5%。程序应显示购物金额、州销售税、郡县销售税、总销售税以及总额（总额是购物金额与各种销售税之和）。

提示：用数值 0.025 代表 2.5%，用 0.05 代表 5%。

7. 每加仑英里数

一辆汽车的每加仑英里数（MPG）可以用以下公式计算：

$$\text{MPG} = \text{行驶里程} \div \text{所耗汽油的加仑数}$$

写一个程序，要求用户提供行驶里程数和所耗汽油加仑数。它应计算出汽车的 MPG 并显示结果。

8. 小费、税金和总额

编写一个程序，计算在饭店吃一顿饭的应付总额。程序应要求用户输入餐费，然后计算 18% 小费和 7% 销售税的金额。显示所有这些金额之和。

9. 摄氏度到华氏度转换器

写一个程序，将摄氏温度（C）转换为华氏温度（F）。其公式如下：

$$F = \frac{9}{5}C + 32$$

该程序应要求用户输入摄氏温度，然后显示转换后的华氏温度。

10. 配方调节器

一个饼干食谱的配方是：

- 1.5 杯糖
- 1 杯黄油
- 2.75 杯面粉

该食谱用上述数量的原料能制作 48 块饼干。编写一个程序，询问用户想做多少块饼干，然后显示制作指定数量的饼干所需的每种原材料的杯数。

11. 狮虎比例

写一个程序，向用户询问当地动物园的大型猫科动物（Bit Cat）展区的狮子和老虎数量。程序应显示狮子和老虎的百分比。

提示：假设展区有 8 头狮子和 12 头老虎，那么共计 20 头大型猫科动物。其中，狮子的百分比可以计算为 $8 \div 20 = 0.4$ ，或 40%。老虎的百分比可以计算为 $12 \div 20 = 0.6$ ，或 60%。

12. 股票交易计划

上个月，乔购入了一些 Acme 软件公司的股票。以下是购入细节：

- 乔购入的股票数量是 2000 股
- 当乔购入股票时，他每股支付了 40.00 美元
- 乔向他的股票经纪人支付的佣金相当于他购买股票金额的 3%

两周后，乔卖出了这支股票。以下是卖出细节：

- 乔卖出的股票数量是 2000 股
- 他以每股 42.75 美元的价格卖出
- 他又向股票经纪人支付了一笔佣金，金额为他卖出股票金额的 3%

写一个程序来显示以下信息：

- 乔购入股票所支付的金额
- 乔购入股票时支付给经纪人的佣金金额
- 乔卖出股票的金额
- 乔卖出股票时支付给经纪人的佣金金额
- 显示乔在卖出股票和支付给经纪人（两次）后剩下的钱的金额。这个金额是正数，乔

赚了。数额是负数，乔就亏了。

13. 种植葡萄藤

一个葡萄园主想要种几排新的葡萄藤，她需要知道每一排能种多少葡萄藤。她已经确定，在测量好未来一排的长度后，就可以用以下公式来计算出这一排适合栽种的葡萄藤数量。注意，这一排的两端还要分别为葡萄架建造一个支柱。

$$V = \frac{R - 2E}{S}$$

下面是对该公式的解释：

- V 是这一排适合栽种的葡萄藤数量
- R 是这一排的长度，单位是英尺
- E 是支柱占用的空间量，单位是英尺
- S 是葡萄藤之间的空间量，单位是英尺

写一个程序，为葡萄园主完成计算。程序要求用户输入以下内容：

- 可栽种的一排的长度，单位是英尺。
- 一端的支柱占用的空间量，单位是英尺。
- 葡萄藤之间的空间量，单位是英尺。

输入数据后，程序应计算并显示适合在这一排栽种的葡萄藤的数量。

14. 复利

当银行账户支付复利时，它不仅为存入账户的本金支付利息，还为长期积累下来的利息支付利息。假定要把一些钱存入储蓄账户，并让该账户在一定年限内赚取复利。计算指定年限后的账户余额的公式是：

$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

下面是对这个公式的解释：

- A 是指定年限后账户中的资金数额
- P 是最初存入该账户的本金数额
- r 是年利率
- n 是每年计算复利的次数
- t 是指定的年限

写一个程序来帮助自己计算。程序要求用户输入以下内容：

- 最初存入该账户的本金数额

-
- 该账户的年利率
 - 每年计算复利的次数（例如，如果复利按月计算，则输入 12。如果按季度计算，则输入 4）
 - 为该账户留多少年来赚取利息

输入数据后，程序应计算并显示指定年限后账户中的资金数额。



注意：用户应将利率以百分点的形式输入。例如，2%应作为 2 来输入，而不是 0.02。程序要将输入除以 100，从而将小数点移动到正确位置。

15. 海龟图形的绘制

使用海龟图形库来写程序，复现图 2-40 的每个图形。

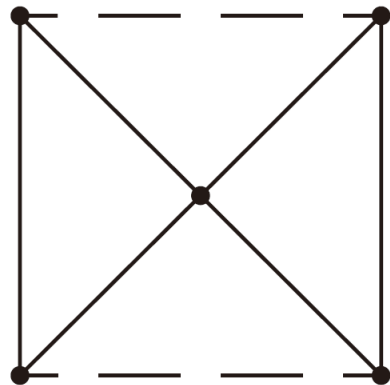
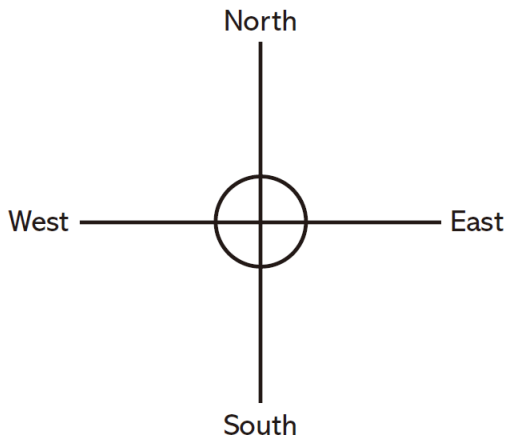
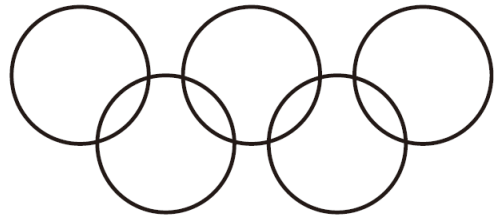
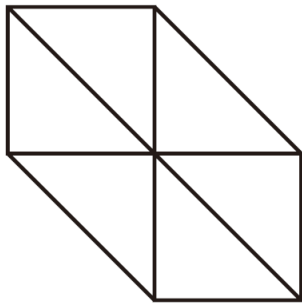
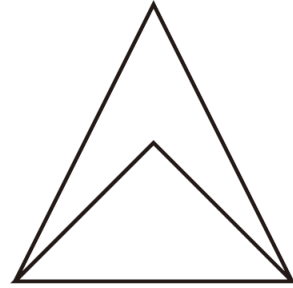
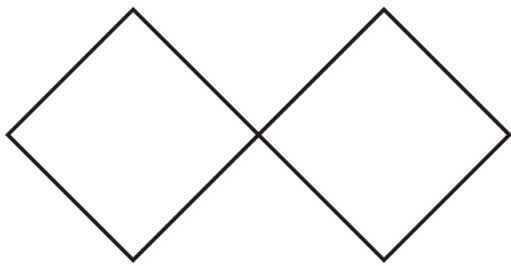


图 2-40 不同的图形

第 3 章 判断结构和布尔逻辑

3.1 if 语句

概念：if 语句创建了一个判断结构，使程序可以有多个执行路径。if 语句导致一个或多个语句仅在一个布尔表达式求值为真时才执行。

 视频讲解：The if Statement

控制结构是对语句的执行顺序进行控制的逻辑设计。到目前为止，本书只使用了最简单的控制结构类型：**顺序结构**。顺序结构按其出现顺序来执行一系列语句。例如，以下代码就是一个顺序结构，语句从头到尾顺序执行：

```
name = input('你的名字是什么? ')
age = int(input('你的年龄多大? '))
print('这是你输入的数据:')
print('名字:', name)
print('年龄:', age)
```

虽然顺序结构在编程中被大量使用，但它不能解决每一种类型的任务。这是因为有的问题根本无法通过顺序执行一系列步骤来解决。例如，考虑判断是否要为员工计算加班费的程序。如果员工的工时超过 40 小时，那么超过 40 小时的所有时间都要计算加班费，否则就跳过加班计算。像这样的程序需要一种不同类型的控制结构：只在特定情况下执行一组语句的结构。这可以通过一个**判断结构**来实现（判断结构也称为**选择结构**）。

在判断结构最简单的形式中，只有当某个条件成立时才执行一个特定的行动。该条件不成立，则不执行。图 3-1 的流程图展示了如何将一个日常生活中的判断逻辑绘制成判断结构。菱形符号代表一个真/假条件。条件为真，就选择一个分支路径，这导致一个行动被执行。条件为假，就选择正常路径，跳过该行动。

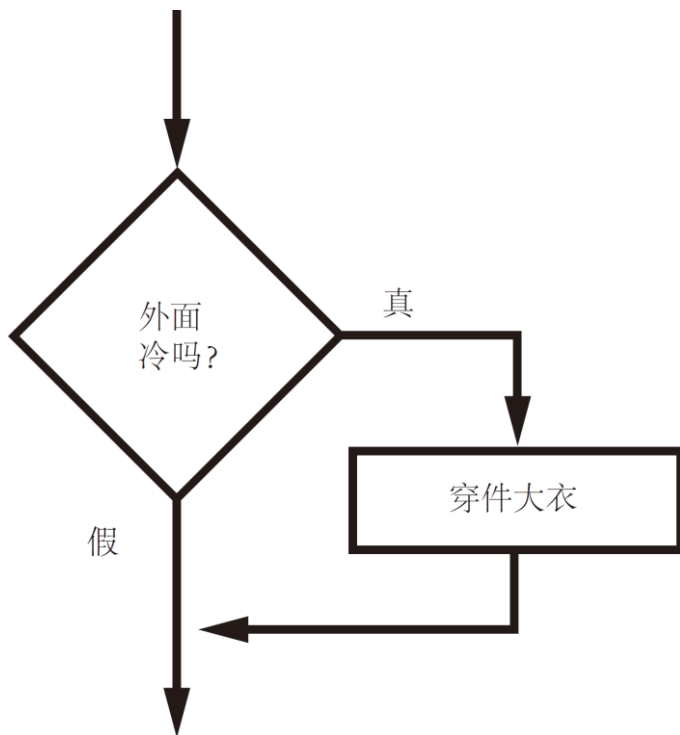


图 3-1 一个简单的判断结构

在流程图中，菱形符号表示某个必须测试的条件。在本例中，我们要判断条件“外面冷吗？”是真还是假。条件为真，就执行“穿件大衣”的行动；为假则跳过该行动。我们说这个行动“条件执行”，因为它只有在某个条件为真时才执行。

程序员将图 3-1 的判断结构类型称为**单分支判断结构**。这是因为它只提供一个分支执行路径。如果菱形符号中的条件为真，我们就选择该分支路径；否则退出整个结构。图 3-2 展示了一个更复杂的例子，在外面冷的时候会执行三个行动。但是，它仍然是单分支判断结构，因为还是只有一个分支执行路径。

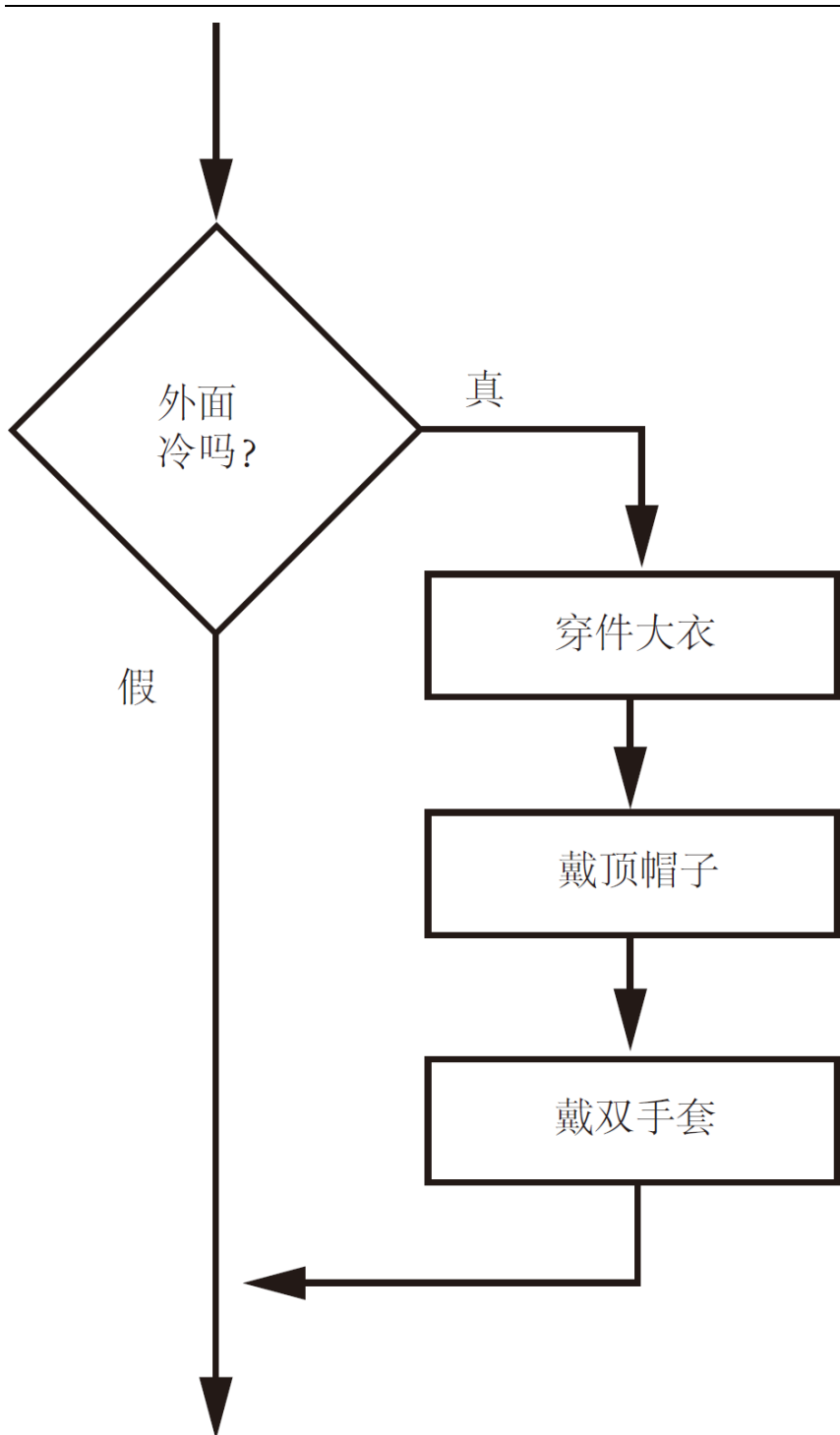


图 3-2 该判断结构在外面冷的时候执行三个行动

在 Python 中，我们使用 `if` 语句来写单分支判断结构。下面是 `if` 语句的常规格式。

```
if 条件:
    语句
    语句
    ...
```

为了简单起见，我们将第一行称为 `if` 子句。`if` 子句以单词 `if` 开始，后跟一个条件，这是一个求值为真 (`True`) 或假 (`False`) 的表达式。条件后面是一个冒号。从下一行起是一个语句块。所谓**语句块 (block)**，其实就是多个相关语句的一个分组。注意，在常规格式中，块中的所有语句都要缩进，因为 Python 解释器用它来区分块的开始和结束。

执行 `if` 语句时会测试条件。条件为真，就执行 `if` 子句后面的块中的语句。条件为假，则跳过该块中的语句。

3.1.1 布尔表达式和关系操作符

如前所述，`if` 语句测试一个表达式以判断它的真假。被 `if` 语句测试的表达式称为**布尔表达式 (Boolean expression)**，这是为了纪念英国数学家乔治·布尔 (George Boole) 而命名的。布尔在 19 世纪发明了将真和假的抽象概念应用于计算的一个数学分析系统。

`if` 语句测试的布尔表达式一般通过一个关系操作符来构建。关系操作符判断两个值是否存在特定关系。例如，大于操作符 (`>`) 判断一个值是否大于另一个值。相等性操作符 (`==`) 判断两个值是否相等。表 3-1 列出了 Python 支持的关系操作符。

表 3-1 关系操作符

操作符	含义
<code>></code>	大于
<code><</code>	小于
<code>>=</code>	大于等于
<code><=</code>	小于等于
<code>==</code>	等于
<code>!=</code>	不等于

下面是表达式的一个例子，它使用大于 (`>`) 操作符来比较两个变量，即 `length` 和 `width`：

```
length > width
```

这个表达式判断 `length` 引用的值是否大于 `width` 引用的值。如果是 `length` 大于 `width`，那么表达式的值为真；否则，表达式的值为假。以下表达式使用小于操作符来判断 `length` 是否小于 `width`：

```
length < width
```

表 3-2 列出了对变量 `x` 和 `y` 进行比较的几个布尔表达式的例子。

表 3-2 使用关系操作符的布尔表达式

表达式	含义
<code>x > y</code>	<code>x</code> 大于 <code>y</code> 吗?
<code>x < y</code>	<code>x</code> 小于 <code>y</code> 吗?
<code>x >= y</code>	<code>x</code> 大于等于 <code>y</code> 吗?
<code>x <= y</code>	<code>x</code> 小于等于 <code>y</code> 吗?
<code>x == y</code>	<code>x</code> 等于 <code>y</code> 吗?
<code>x != y</code>	<code>x</code> 不等于 <code>y</code> 吗?

可以在交互模式下使用 Python 解释器来实验这些操作符。在 `>>>` 提示符下输入一个布尔表达式，解释器会对该表达式进行求值，并将求值结果显示为 `True`（真）或 `False`（假）。以下交互会话对此进行了演示（为方便引用，我们添加了行号）。

```
1 >>> x = 1 
2 >>> y = 0 
3 >>> x > y 
4 True
5 >>> y > x 
6 False
7 >>>
```

第 1 行为变量 `x` 赋值 `1`，第 2 行为变量 `y` 赋值 `0`。第 3 行输入布尔表达式 `x > y`，第 4 行显示了该表达式的值（`True`）。然后，第 5 行输入布尔表达式 `y > x`，第 6 行显示了该表达式的值（`False`）。

以下交互会话演示了 `<` 操作符。

```
1 >>> x = 1 
2 >>> y = 0 
3 >>> y < x 
4 True
5 >>> x < y 
6 False
7 >>>
```

第 1 行为变量 x 赋值 1，第 2 行为变量 y 赋值 0。第 3 行输入布尔表达式 $y < x$ ，第 4 行显示了该表达式的值（True）。然后，第 5 行输入布尔表达式 $x < y$ ，第 6 行显示了该表达式的值（False）。

3.1.2 $>=$ 和 $<=$ 操作符

$>=$ 和 $<=$ 操作符测试一个以上的关系。其中， $>=$ 判断左侧的操作数是否大于或等于右侧的操作数， $<=$ 则判断左侧的操作数是否小于或等于右侧的操作数。

以下交互会话对此进行了演示。

```
1 >>> x = 1 
2 >>> y = 0 
3 >>> z = 1 
4 >>> x >= y 
5 True
6 >>> x >= z 
7 True
8 >>> x <= z 
9 True
10 >>> x <= y 
11 False
12 >>>
```

第 1 行~第 3 行为变量 x ， y 和 z 赋值。第 4 行输入布尔表达式 $x >= y$ ，求值结果为真。第 6 行输入布尔表达式 $x >= z$ ，同样为真。第 8 行输入布尔表达式 $x <= z$ ，同样为真。第 10 行输入布尔表达式 $x <= y$ ，求值结果为假。

3.1.3 $==$ 操作符

相等性操作符 $==$ 操作符判断左侧的操作数是否等于右侧的操作数。如果两个操作数所引用的值相同，那么表达式为真。假设 a 为 4，那么表达式 $a == 4$ 为真，表达式 $a == 2$ 则为假。

以下交互会话演示了 $==$ 操作符的用法。

```
1 >>> x = 1 
2 >>> y = 0 
3 >>> z = 1 
```

```
4 >>> x == y 
5 False
6 >>> x == z 
7 True
8 >>>
```



注意：相等性操作符是两个=符号连写。不要将其与赋值操作符=混淆，后者只有一个=符号。

3.1.4 !=操作符

!=操作符称为不等于操作符。它判断左侧的操作数是否不等于右侧的操作数，这正好与==操作符相反。假设 a 为 4，b 为 6，c 为 4，那么 a != b 和 b != c 都为真，因为 a 不等于 b，b 也不等于 c。但是 a != c 为假，因为 a 等于 c。

以下交互会话演示了 != 操作符的用法。

```
1 >>> x = 1 
2 >>> y = 0 
3 >>> z = 1 
4 >>> x != y 
5 True
6 >>> x != z 
7 False
8 >>>
```

3.1.5 综合运用

来看看下面这个 if 语句的例子：

```
if sales > 50000:
    bonus = 500.0
```

这个语句使用 > 操作符来判断 sales（销售额）是否大于 50000。如果表达式 sales > 50000 为真，那么变量 bonus（奖金）会被赋值为 500.0。然而，如果表达式为假，那么赋值语句会被跳过。图 3-3 展示了它的流程图。

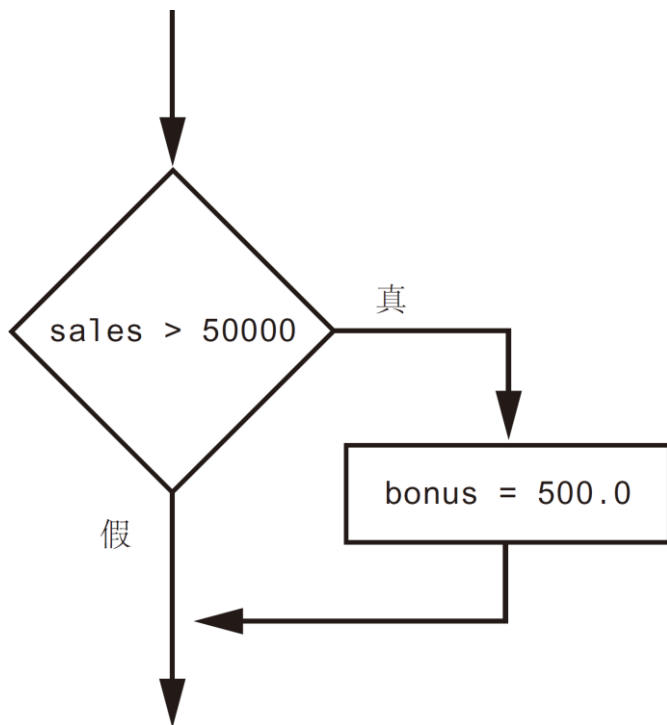


图 3-3 示例判断结构

下例条件执行包含三个语句的一个块。图 3-4 展示了流程图：

```
if sales > 50000:  
    bonus = 500.0  
    commission_rate = 0.12  
    print('你已完成销售配额!')
```

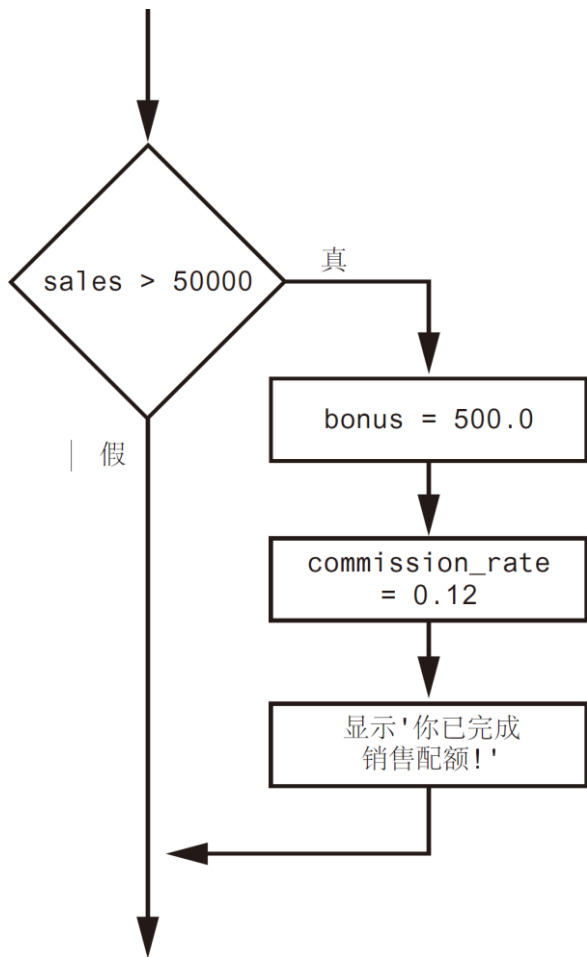


图 3-4 示例判断结构

以下代码使用==操作符来判断两个值是否相等。如果 `balance` 变量不等于 0，那么表达式 `balance==0` 为真，否则为假。

```
if balance == 0:  
    # 这里的语句只有  
    # 在 balance 为0  
    # 时才会执行
```

以下代码使用!=操作符来判断两个值是否不相等。如果 `choice` 变量不等于 5，那么表达式 `choice != 5` 为真，否则为假。

```
if choice != 5:  
    # 这里的语句只有  
    # 在 choice 不等于5  
    # 时才会执行
```

聚光灯：使用 if 语句



凯瑟琳是一门科学课的老师，她的学生需要参加三次考试。她想写一个程序，供学生计算他们的平均成绩。如果平均成绩超过 95 分，她希望程序能热情地祝贺学生。以下是伪代码算法：

```
获取第一次考试的成绩
获取第二次考试的成绩
获取第三次考试的成绩
计算平均成绩
显示平均成绩
如果平均成绩大于 95：
    祝贺用户
```

程序 3-1 展示了该程序的代码。

程序 3-1 (test_average.py)

```
1  # 这个程序获取三个考试成绩，并显示
2  # 平均成绩。如果平均成绩是一个高分，
3  # 那么向学生表示祝贺。
4
5  # HIGH_SCORE 变量存储了一个被认为
6  # 是高分的值。
7  HIGH_SCORE = 95
8
9  # 获取三次考试的成绩
10 test1 = int(input('输入考试 1 的成绩：'))
11 test2 = int(input('输入考试 2 的成绩：'))
12 test3 = int(input('输入考试 3 的成绩：'))
13
14 # 计算平均成绩
15 average = (test1 + test2 + test3) / 3
16
17 # 打印平均成绩
18 print(f'平均成绩为{average}。')
19
20 # 如果平均成绩为高分，
21 # 那么向学生表示祝贺。
22 if average >= HIGH_SCORE:
23     print('恭喜你，')
24     print('成绩非常好！')
```

程序输出（用户输入的内容加粗）

```
输入考试 1 的成绩: 93 
输入考试 2 的成绩: 99 
输入考试 3 的成绩: 96 
平均成绩为 96.0。
恭喜你!
成绩非常好!
```

3.1.6 单行 if 语句

如果在条件成立的情况只执行一个语句，那么 Python 允许将整个 if 语句写在一行上。下面是常规格式。

```
if 条件: 语句
```

以下交互会话对此进行了演示。

```
>> score = 90 
>> if score > 59: print('你通过了测验。') 
... 
你通过了测验。
>>>
```



提示：在 Python shell 中测试代码时，虽然整个 if 语句写在一行很方便，但在 Python 程序中这样做并没有什么好处。在程序中，条件执行的语句应该写在自己的缩进块中，这使 if 语句更容易阅读和调试。

检查点

- 3.1 什么是控制结构？
- 3.2 什么是判断结构？
- 3.3 什么是单分支判断结构？
- 3.4 什么是布尔表达式？
- 3.5 可以用关系操作符来测试值和值之间的哪些关系类型？
- 3.6 写一个 if 语句，如果 y 等于 20，就把 0 赋给 x。
- 3.7 写一个 if 语句，如果 sales（销售额）大于等于 10000，就将 0.2 赋给 commissionRate（佣金率或提成率）。

3.2 if-else 语句

概念：if-else 语句在条件为真时执行一个语句块，条件为假时执行另一个。

视频讲解: if-else 语句

上一节介绍了单分支判断结构（if 语句），它只有一个分支执行路径。现在来看看**双分支判断结构**，它有两个可能的执行路径，一个在条件为真时执行，另一个是在条件为假时执行。图 3-5 展示了双分支判断结构的流程图。

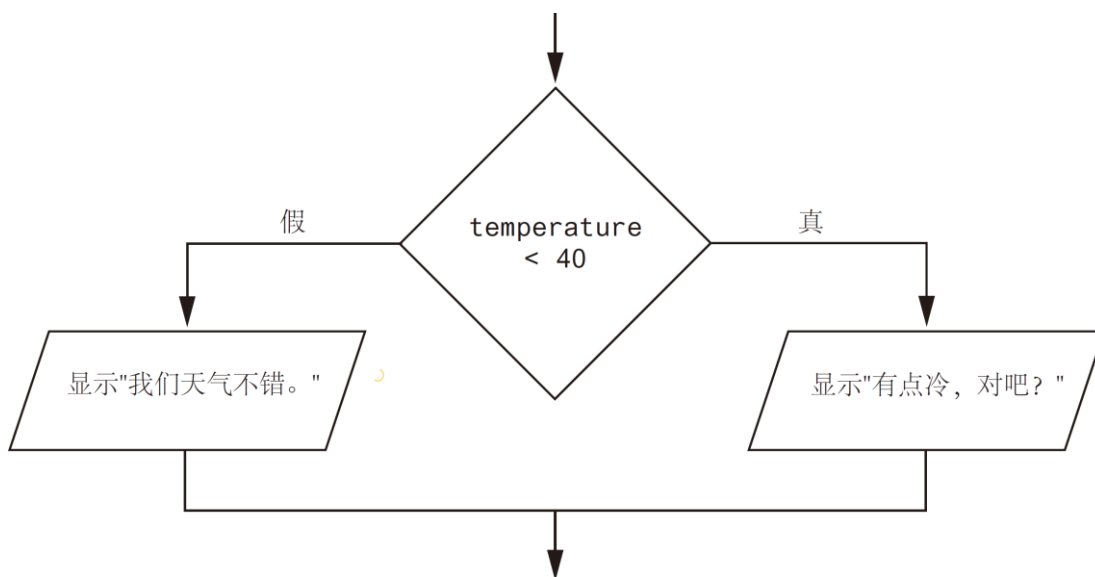


图 3-5 双分支判断结构

流程图中的判断结构测试条件 `temperature < 40`。如果条件为真，就显示“有点冷，对吧？”如果条件为假，则显示“天气不错。”

在代码中，我们用 `if-else` 语句来写双分支判断结构。下面是 `if-else` 语句的常规格式。

```
if 条件:
    语句
    语句
    ...
else:
    语句
    语句
    ...
```

执行这个语句时，首先对条件进行测试。条件为真，就执行 `if` 子句后的缩进语句块，然

后，程序的控制将跳转到 `if-else` 语句之后的语句。条件为假，则执行 `else` 子句后的缩进语句块，程序的控制随后同样跳转到 `if-else` 语句之后的语句。图 3-6 展示了这个过程。

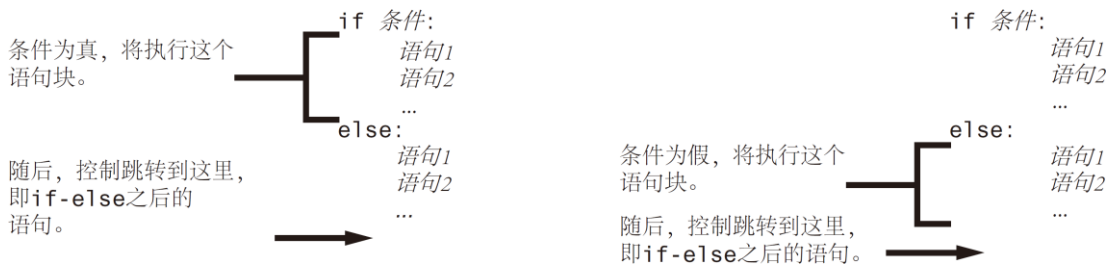


图 3-6 `if-else` 语句的条件执行

以下代码展示了 `if-else` 语句的一个例子，它对应的是图 3-5 的流程图。

```
if temperature < 40:
    print("有点冷, 对吧? ")
else:
    print("我们天气不错。")
```

if-else 语句的缩进

写 `if-else` 语句时要遵循以下缩进原则。

- 确保 `if` 子句和 `else` 子句对齐。
- `if` 子句和 `else` 子句后面都有一个语句块。确保块中语句的缩进是一致的。

图 3-7 对此进行了演示。

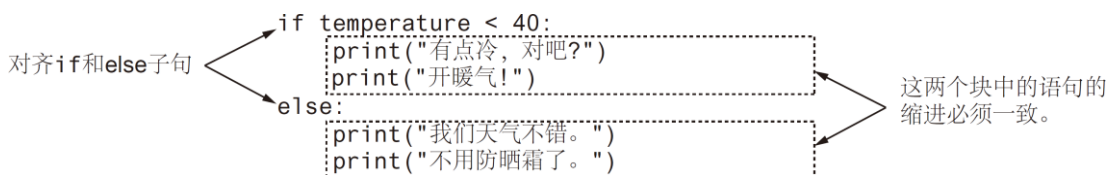


图 3-7 `if-else` 语句的缩进

聚光灯：使用 `if-else` 语句

克里斯拥有一家汽车维修店，有几名员工。如果任何员工在一周内工作超过 40 个小时，超过的部分要支付 1.5 倍基本时薪。克里斯要求你设计一个简单的工资程序来计算员工的工资总额，包括任何加班工资。你设计了以下算法。

获取工作时数。

获取每小时工资。

如果工作时数超过 40 小时：

计算并显示含加班费的总工资。

否则：

照常计算并显示总工资。

程序 3-2 展示了这个程序的代码。注意，第 3 行为具名常量 `BASE_HOURS` 赋值 40，这是员工一周内的基本工作时数，这段时间没有加班费。第 4 行为具名常量 `OT_MULTIPLIER` 赋值 1.5，这是加班费的时薪乘数，意味着员工的加班时薪要乘以 1.5。

程序 3-2 (auto_repair_payroll.py)

```
1  # 声明两个具名常量来表示基本工作时数和
2  # 加班期间的时薪乘数
3  BASE_HOURS = 40 # 每周基本工作时数
4  OT_MULTIPLIER = 1.5 # 加班时薪乘数
5
6  # 获取工作时数和基本时薪
7  hours = float(input('输入工作时数: '))
8  pay_rate = float(input('输入基本时薪: '))
9
10 # 计算并显示总工资
11 if hours > BASE_HOURS:
12     # 计算包含加班费的总工资
13     # 首先，获得加班时数
14     overtime_hours = hours - BASE_HOURS
15
16     # 然后计算包含加班期间的工资
17     overtime_pay = overtime_hours * pay_rate * OT_MULTIPLIER
18
19     # 最后计算总工资
20     gross_pay = BASE_HOURS * pay_rate + overtime_pay
21 else:
22     # 计算不含加班费的总工资
23     gross_pay = hours * pay_rate
24
25 # 显示总工资
26 print(f'总工资为${gross_pay:,.2f}。')
```

程序输出 (用户输入的内容加粗)

输入工作时数: **40**

输入基本时薪: **20**

总工资为**\$800.00**。

程序输出（用户输入的内容加粗）

```
输入工作时数: 50   
输入基本时薪: 20   
总工资为$1,100.00。
```

检查点

3.8 双分支判断结构是如何工作的？

3.9 在 Python 中用什么语句来写双分支判断结构？

3.10 写 if-else 语句时，在什么情况下会执行 else 子句之后的语句块？

3.3 比较字符串

概念：Python 允许比较字符串，这样就可以创建测试字符串值的判断结构。

之前的例子演示了如何在一个判断结构中比较数字。除此之外，还可以比较字符串，如以下代码所示。

```
name1 = 'Mary'  
name2 = 'Mark'  
if name1 == name2:  
    print('名字一样')  
else:  
    print('名字不一样')
```

==操作符对 name1 和 name2 进行比较，判断它们是否相等。由于字符串 'Mary' 和 'Mark' 不相等，所以 else 子句将显示 '名字不一样'。

再来看另一个例子。假设 month 变量引用了一个字符串，以下代码使用 != 操作符来判断 month 所引用的值是否不等于 '十月'：

```
if month != '十月':  
    print('还没有到国庆节!')
```

程序 3-3 用一个完整的程序来演示了如何比较两个字符串。程序提示用户输入一个密码，然后判断输入的字符串是否等于 'prospero'。

程序 3-3 (password.py)

```
1 # 这个程序比较两个字符串
```

```
2 # 从用户处获取一个密码
3 password = input('请输入密码: ')
4
5 # 判断密码是否
6 # 被正确输入
7 if password == 'prospero':
8     print('密码正确。')
9 else:
10    print('对不起, 密码错误。')
```

程序输出 (用户输入的内容加粗)

请输入密码: **ferdinand**
对不起, 密码错误。

程序输出 (用户输入的内容加粗)

请输入密码: **prospero**
密码正确。

程序输出 (用户输入的内容加粗)

请输入密码: **Prospero**
对不起, 密码错误。

字符串比较要区分大小写。例如, 字符串 'saturday' 和 'Saturday' 不相等, 因为第一个字符串中的 s 小写, 但在第二个字符串中大写。在程序 3-3 的最后一个示例会话中, 展示了当用户输入 Prospero 作为密码 (大写的 P) 时发生的情况。



提示: 第 8 章会讲解如何对字符串进行不区分大小写的比较。

其他字符串比较

除了判断字符串是否相等, 还可以判断一个字符串是否大于或小于另一个字符串。这是一个很有用的功能, 因为程序员经常需要设计程序, 按某种顺序对字符串进行排序。

第 1 章讲过, 计算机在内存中存储的并不是真正的字符, 例如 A, B, C, 等等。相反, 它们存储的是代表这些字符的数值编码。第 1 章提到, ASCII (美国信息交换标准代码) 是一个常用的字符编码系统。可以在附录 C 中查看完整的 ASCII 代码集, 但这里有一些关于它的事实。

- 大写字母 A~Z 对应的数字编码是 65~90。
- 小写字母 a~z 对应的数字编码是 97~122。
- 当数字 0~9 作为字符存储到内存时，它们由数字编码 48~57 表示。例如，字符串 'abc123' 在内存中存储时的编码是 97, 98, 99, 49, 50 和 51。
- 空格的编码是 32。

除了建立了一套数字编码来表示内存中的字符，ASCII 还为字符建立了一个顺序。字符 'A' 在字符 'B' 之前，'B' 在字符 'C' 之前，以此类推。

程序对字符进行比较时，它实际是在比较字符的编码。来看看下面这个 if 语句：

```
if 'a' < 'b':  
    print('字母 a 小于字母 b')
```

这段代码判断字符 'a' 的 ASCII 编码是否小于字符 'b' 的 ASCII 编码。表达式 'a' < 'b' 的求值结果为真，因为 'a' 的编码小于 'b' 的编码。所以，如果这是一个实际程序的一部分，它将显示 '字母 a 小于字母 b'。

再来看看包含一个以上字符的字符串通常是如何比较的。假设一个程序使用了字符串 'Mary' 和 'Mark'，如下所示：

```
name1 = 'Mary'  
name2 = 'Mark'
```

图 3-8 展示了字符串 'Mary' 和 'Mark' 中的各个字符在内存中是如何用 ASCII 码来存储的。

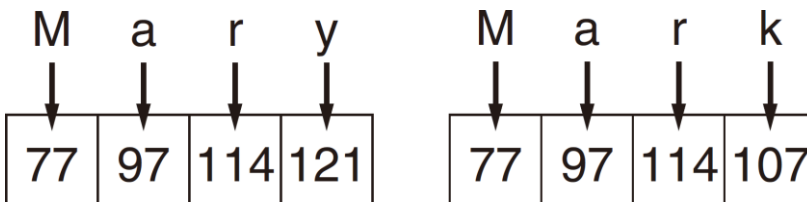


图 3-8 字符串 'Mary' 和 'Mark' 中的各个字符的编码

使用关系操作符来比较字符串时，会逐个比较字符串中的字符。如以下代码所示：

```
name1 = 'Mary'  
name2 = 'Mark'  
if name1 > name2:  
    print('Mary 大于 Mark')  
else:  
    print('Mary 不大于 Mark')
```

>操作符比较字符串 'Mary' 和 'Mark' 中的每个字符，从第一个（最左侧）字符开始。图 3-9

对此进行了展示。

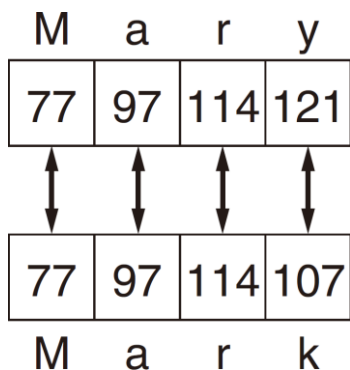


图 3-9 比较字符串中的每个字符

下面是比较过程。

1. 'Mary'中的'M'与'Mark'中的'M'比较。由于两个字符相同，所以比较下一个字符。
2. 'Mary'中的'a'与'Mark'中的'a'比较。由于两个字符相同，所以比较下一个字符。
3. 'Mary'中的'r'与'Mark'中的'r'比较。由于两个字符相同，所以比较下一个字符。
4. 'Mary'中的'y'与'Mark'中的'k'比较。由于两个字符不一样，所以这两个字符串不相等。由于字符'y'的ASCII码（121）大于'k'（107），所以字符串'Mary'大于字符串'Mark'。

如果一个字符串比另一个短，那么只比较对应的字符。如果所有对应的字符都相同，那么较短的字符串被认为小于较长的字符串。例如，假设比较字符串'High'和'Hi'。那么'Hi'被认为比'High'小，因为它更短。

程序 3-4 简单演示了如何用<操作符比较两个字符串。程序提示用户输入两个姓名，然后按从小到大的顺序显示这两个姓名。

程序 3-4 (sort_names.py)

```
1 # 这个程序用<操作符来比较字符串
2 # 从用户处获取两个姓名
3 name1 = input('输入一个姓名 (姓在前): ')
4 name2 = input('输入另一个姓名 (姓在前): ')
5
6 # 按从小到大的顺序显示两个姓名
7 print('姓名按从小到大的顺序排列: ')
8 if name1 < name2:
```

```
9     print(name1)
10    print(name2)
11    else:
12        print(name2)
13        print(name1)
```

程序输出（用户输入的内容加粗）

```
输入一个姓名（姓在前）: Jones, Richard 
输入另一个姓名（姓在前）: Costa, Joan 
姓名按从小到大的顺序排列:
Costa, Joan
Jones, Richard
```

检查点

3.11 以下代码会输出什么？

```
if 'z' < 'a':
    print('z 小于 a')
else:
    print('z 不小于 a')
```

3.12 以下代码会输出什么？

```
s1 = 'New York'
s2 = 'Boston'
if s1 > s2:
    print(s2)
    print(s1)
else:
    print(s1)
    print(s2)
```

3.4 嵌套判断结构和 if-elif-else 语句

概念：为了测试一个以上的条件，可以在一个判断结构中嵌套另一个判断结构。

第 3.1 节讲到，控制结构决定了一组语句的执行顺序。程序通常被设计成不同控制结构的组合。例如，图 3-10 的流程图展示了一个判断结构如何与两个顺序结构结合。

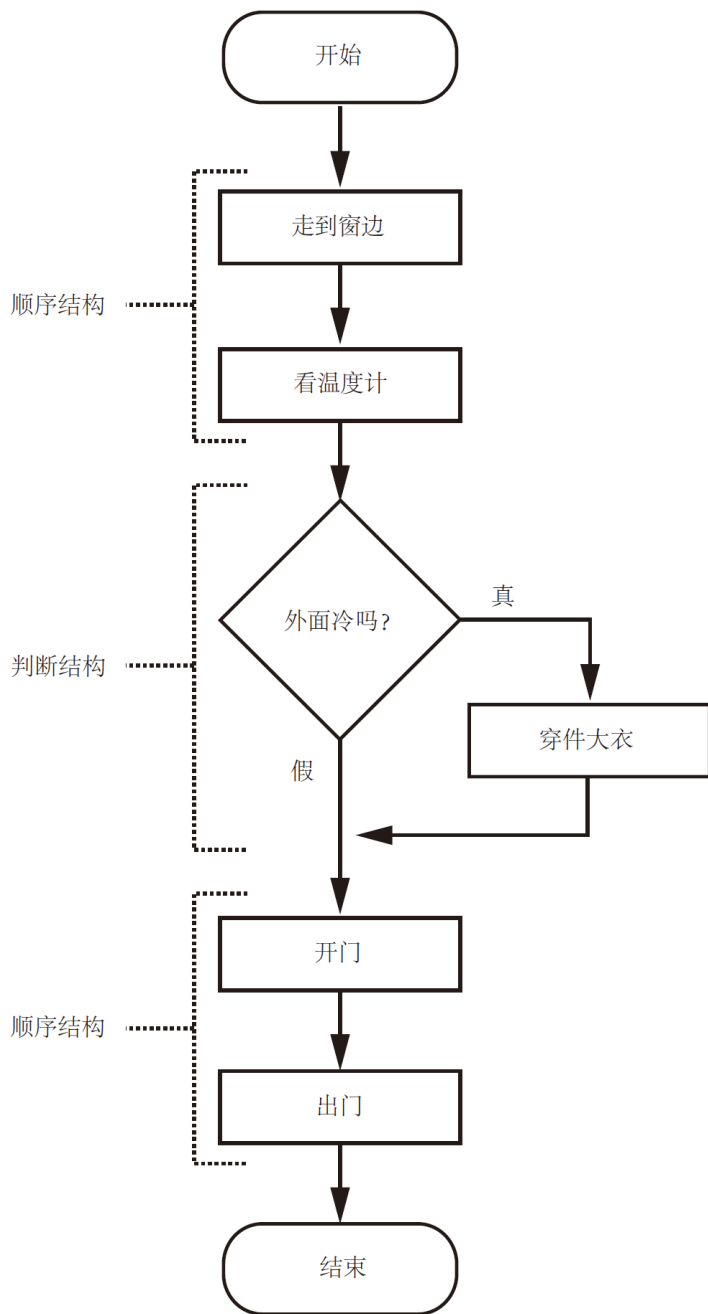


图 3-10 结合使用顺序结构和判断结构

该流程图以一个顺序结构开始。假设窗户那里有一个室外温度计，第一步是走到窗边，下一步是看温度计的读数。接着是一个判断结构，要测试的条件是室外冷不冷。如果条件为真，那么会执行穿件大衣的行动。接着是另一个顺序结构，开门并出门。

结构经常需要嵌套使用。以图 3-11 的部分流程图为例，它展示了在判断结构中嵌套一个顺序结构的情况。判断结构测试条件外面冷不冷的条件。如果条件为真，就执行顺序结构中的步骤。

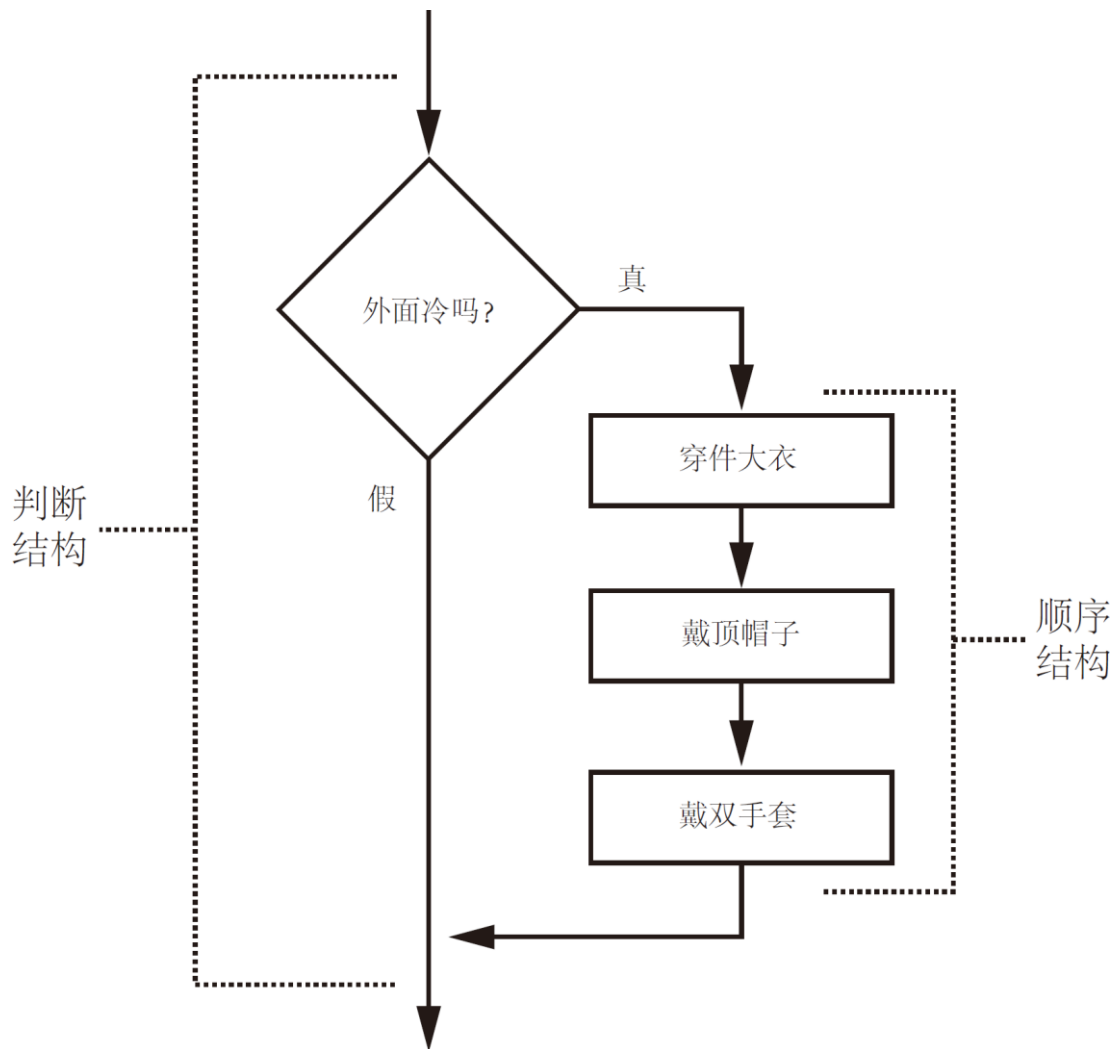


图 3-11 嵌套在判断结构中的顺序结构

还可以将判断结构嵌套在其他判断结构中。事实上，程序在测试一个以上的条件时经常需要这样做。例如，假定某个程序需要判断银行客户有没有贷款资格。批贷必须满足两个条件：(1) 客户年收入至少为 30000 美元，(2) 客户当前工作至少满两年。图 3-12 展示了算法流程图。假定已将客户的年薪赋给了 `salary` 变量，将客户当前工作年限赋给了 `years_on_job` 变量。

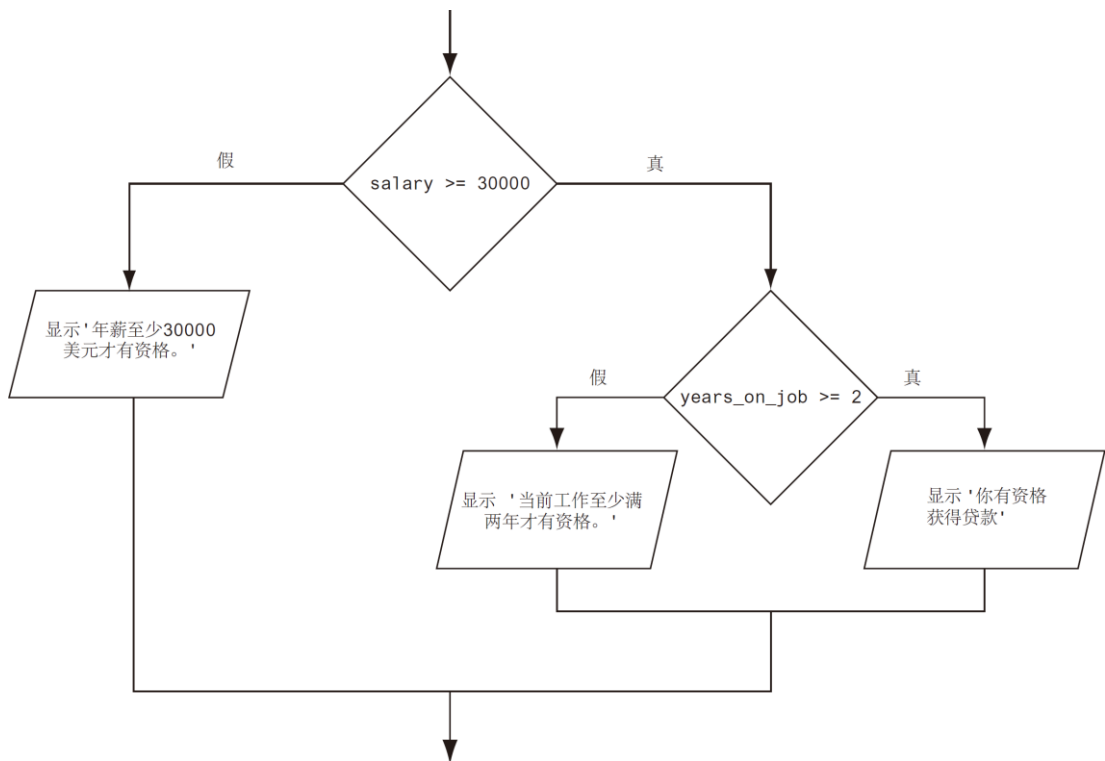


图 3-12 嵌套判断结构

过一遍执行流程，就会看到首先测试条件 `salary >= 30000`。如果该条件为假，就没有必要进行下一步测试，因为我们已经知道客户没有资格贷款。然而，如果该条件为真，那么需要测试第二个条件。这是通过一个嵌套判断结构来完成的，它测试的条件是 `years_on_job`。如果该条件为假，那么客户不符合批贷条件。程序 3-5 展示了完整程序。

程序 3-5 (loan_qualifier.py)

```

1  # 这个程序判断银行客户是否
2  # 有资格获得贷款。
3
4  MIN_SALARY = 30000.0 # 最低年薪
5  MIN_YEARS = 2 # 最低工作年限
6
7  # 获取客户的年薪
8  salary = float(input('请输入年薪: '))
9
10 # 获取当前工作年限
  
```

```
11 years_on_job = int(input( '请输入当前' +
12                          '工作年限: '))
13
14 # 判断客户是否符合贷款资格
15 if salary >= MIN_SALARY:
16     if years_on_job >= MIN_YEARS:
17         print('有资格贷款。')
18     else:
19         print(f'当前工作'
20               f'至少满{MIN_YEARS}年, '
21               f'才有资格贷款。')
22 else:
23     print(f'年薪至少为'
24           f'{MIN_SALARY:,.2f}美元, '
25           f'才有资格贷款。')
```

程序输出（用户输入的内容加粗）

```
请输入年薪: 35000 
请输入当前工作年限: 1 
当前工作至少满 2 年, 才有资格贷款。
```

程序输出（用户输入的内容加粗）

```
请输入年薪: 25000 
请输入当前工作年限: 5 
年薪至少为 30,000.00 美元, 才有资格贷款。
```

程序输出（用户输入的内容加粗）

```
请输入年薪: 35000 
请输入当前工作年限: 5 
有资格贷款。
```

注意从第 15 行开始的 if-else 语句。它首先测试条件 `salary >= MIN_SALARY`。只有该条件为真，才会继续执行从第 16 行开始的嵌套 if-else 语句。否则，程序将跳转到第 22 行，执行父 if-else 语句的 else 子句，并执行第 23 行~第 25 行的语句块。

嵌套判断结构中的缩进一定要正确。正确的缩进不仅仅是 Python 解释器所要求的，它还能使代码的读者（你和其他人）更容易看清楚结构的每一部分所执行的行动。写嵌套 if 语句时请遵循以下规则。

- 每个 else 子句都要和它匹配的 if 子句对齐。图 3-13 对此进行了演示。
- 确保每个块中的语句都一致地缩进。图 3-14 加了底纹的部分展示了判断结构中的嵌套块。注意每个块中的每个语句都有相同的缩进量。

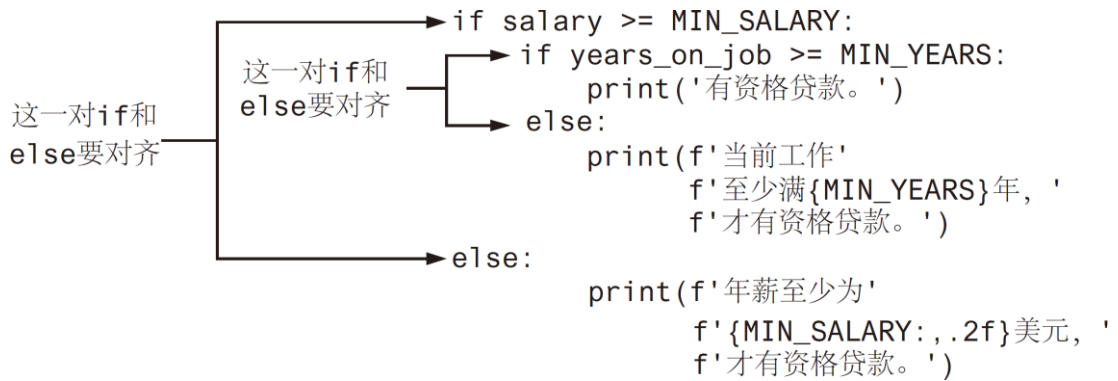


图 3-13 一对 if 和 else 子句要对齐

```

if salary >= MIN_SALARY:
    if years_on_job >= MIN_YEARS:
        print('有资格贷款。')
    else:
        print(f'当前工作'
              f'至少满{MIN_YEARS}年, '
              f'才有资格贷款。')
else:
    print(f'年薪至少为'
          f'{MIN_SALARY:,.2f}美元, '
          f'才有资格贷款。')

```

图 3-14 嵌套块

3.4.1 测试一系列条件

前面的例子展示了如何使用嵌套判断结构来测试一个以上的条件。我们经常需要测试一系列条件，并根据哪个条件为真来采取一个行动。为此，一个办法是在判断结构中嵌套多个其他判断结构。以下“聚光灯”小节展示了一个例子。

聚光灯：多个嵌套判断结构

苏亚雷斯博士教的是文学课，他的所有考试都使用以下成绩评定标准。

考试分数	成绩
90 或以上	A
80~89	B
70~79	C

60~69	D
低于 60	F

他请你写一个程序，让学生输入考试分数，然后显示和该分数对应的字母成绩。下面是程序所用的算法：

1. 要求用户输入一个考试分数（Score）。
2. 采用以下方式判断成绩（Grade）：

 如果分数大于等于 90 分，那么成绩为 A。

 否则，如果分数大于等于 80 分，那么成绩为 B。

 否则，如果分数大于或等于 70 分，那么成绩为 C。

 否则，如果分数大于或等于 60 分，那么成绩为 D。

 否则，成绩为 F。

为了判断成绩，需要用到多个嵌套判断结构，如图 3-15 所示。程序 3-6 展示了这个程序。嵌套判断结构在第 14 行~第 26 行。

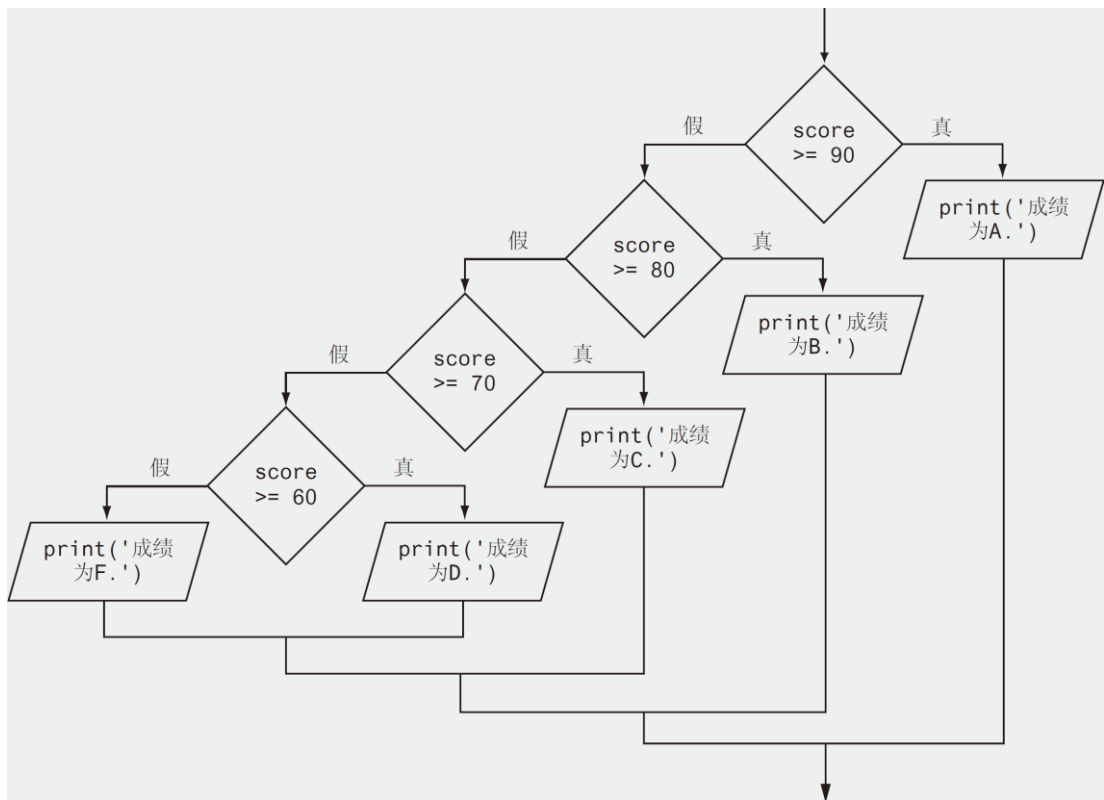


图 3-15 用嵌套判断结构来判断成绩

程序 3-6 (grader.py)

```
1  # 这个程序从用户处获取考试分数 (Score),
2  # 并显示相应的字母成绩 (Grade)。
3
4  # 代表成绩阈值的具名常量
5  A_score = 90
6  B_score = 80
7  C_score = 70
8  D_score = 60
9
10 # 从用户处获取一个考试分数
11 score = int(input('输入考试分数: '))
12
13 # 判断成绩
14 if score >= A_score:
15     print('成绩为 A。')
16 else:
17     if score >= B_score:
18         print('成绩为 B。')
19     else:
20         if score >= C_score:
21             print('成绩为 C。')
22         else:
23             if score >= D_score:
24                 print('成绩为 D。')
25             else:
26                 print('成绩为 F。')
```

程序输出 (用户输入的内容加粗)

```
输入考试分数: 78 
成绩为 C。
```

程序输出 (用户输入的内容加粗)

```
输入考试分数: 84 
成绩为 B。
```

3.4.2 if-elif-else 语句

虽然程序 3-6 很简单,但也能看出嵌套判断结构的逻辑相当复杂。Python 提供了一个特殊版本的判断结构,称为 if-elif-else 语句,它使这种类型的逻辑更容易编写。下面是它

的常规格式:

```
if 条件1:
    语句
    语句
    ...
elif 条件2:
    语句
    语句
    ...
```

根据需要插入更多的elif子句 ...

```
else:
    语句
    语句
    ...
```

执行 if-elif-else 语句时，会首先测试“条件 1”。如果“条件 1”为真，会执行紧随其后的语句块，直到遇到 elif 子句为止。随后，整个结构其余的部分都会被忽略。然而，如果“条件 1”为假，那么程序会跳到下一个 elif 子句并测试“条件 2”。如果“条件 2”为真，会执行紧随其后的语句块，直到遇到 elif 子句为止。随后，整个结构其余的部分都会被忽略。这个过程会一直持续，直到发现一个条件为真，或者不再有更多的 elif 子句。如果所有条件都不为真，那么会执行 else 子句之后的语句块。

下面是 if-elif-else 语句的一个例子。它改写了程序 3-6 的第 14 行~第 26 行的嵌套判断结构，但可读性变得更好了。

```
if score >= A_SCORE:
    print('成绩为 A。')
elif score >= B_SCORE:
    print('成绩为 B。')
elif score >= C_SCORE:
    print('成绩为 C。')
elif score >= D_SCORE:
    print('成绩为 D。')
else:
    print('成绩为 F。')
```

注意 if-elif-else 语句采用的对齐和缩进方式: if, elif 和 else 子句都对齐，条件执行的块则缩进。

if-elif-else 语句并非必须，因为它的逻辑完全可以用嵌套 if-else 语句来实现。但是，一长串嵌套 if-else 语句在调试代码时有两个特别不好的地方:

- 代码可能变得非常复杂，以至于难以理解。
- 由于需要缩进，一长串嵌套 if-else 语句会变得过于臃肿，在屏幕上显示时需要水平

滚动。另外，过长的语句在打印到纸上时往往会自动换行，使代码更加难以阅读。

`if-elif-else` 语句的逻辑通常比一长串嵌套 `if-else` 语句更容易理解。另外，由于 `if-elif-else` 语句中所有子句都对齐，所以行的长度可控。

检查点

3.13 将以下代码转换为 `if-elif-else` 语句。

```
if number == 1:
    print('One')
else:
    if number == 2:
        print('Two')
    else:
        if number == 3:
            print('Three')
        else:
            print('Unknown')
```

3.5 逻辑操作符

概念：逻辑 `and` 操作符和逻辑 `or` 操作符允许连接多个布尔表达式以创建一个复合表达式。逻辑 `not` 操作符可以反转布尔表达式的值。

Python 提供了一组**逻辑操作符**，可以用它们来创建复杂的布尔表达式。表 3-3 总结了这些操作符。

表 3-3 逻辑操作符

操作符	含义
<code>and</code>	<code>and</code> 操作符将两个布尔表达式连接成一个复合表达式。只有两个子表达式都为真，复合表达式才为真。
<code>or</code>	<code>or</code> 操作符将两个布尔表达式连接成一个复合表达式。任何子表达式为真，复合表达式就为真。
<code>not</code>	<code>not</code> 操作符是一元操作符，意味着它只作用于一个操作数。操作数必须是布尔表达式。 <code>not</code> 操作符的作用是对操作数的值进行取反。换言之，应用于一个为真的表达式，该操作符返回假；应用于一个为假的表达式，操作符则返回真。

表 3-4 展示了使用逻辑操作符的几个示例复合布尔表达式。

表 3-4 使用逻辑操作符的复合布尔表达式

表达式	含义
<code>x > y and a < b</code>	x 大于 y 而且 a 小于 b 吗?
<code>x == y or x == z</code>	x 等于 y 或者等于 z 吗?
<code>not (x > y)</code>	表达式 <code>x > y</code> 不为真吗?

3.5.1 and 操作符

`and` 操作符取两个布尔表达式作为操作数，从而创建一个复合布尔表达式，该表达式只有在两个子表达式都为真时才为真。下面是使用了 `and` 操作符的一个示例 `if` 语句。

```
if temperature < 20 and minutes > 12:  
    print('温度处于危险区域。')
```

在这个语句中，两个布尔表达式 `temperature < 20` 和 `minutes > 12` 被组合成一个复合表达式。只有当 `temperature` 小于 20 而且 `minutes` 大于 12 时，才会调用 `print` 函数。任何一个布尔子表达式为假，复合表达式就为假，不会打印那条消息。

表 3-5 展示了 `and` 操作符的**真值表**（truth table）。真值表列出的“表达式”涵盖了与 `and` 操作符连接的所有可能的真假值组合。该表同时显示了每个表达式的求值结果。

表 3-5 `and` 操作符的真值表

表达式	求值结果
<code>true and false</code>	<code>false</code>
<code>false and true</code>	<code>false</code>
<code>false and false</code>	<code>false</code>
<code>true and true</code>	<code>true</code>

如表所示，只有在 `and` 操作符两边都为真时，操作符才会返回真。

3.5.2 or 操作符

or 操作符取两个布尔表达式作为操作数，从而创建一个复合布尔表达式，该表达式在两个子表达式任何一个为真时就为真。下面是使用了 or 操作符的一个示例 if 语句。

```
if temperature < 20 or temperature > 100:  
    print('处于极端温度。')
```

只要 temperature 小于 20 或者大于 100，就会调用 print 函数。任何一个布尔子表达式为真，复合表达式就为真。表 3-6 是 or 操作符的真值表。

表 3-6 or 操作符的真值表

表达式	求值结果
true or false	true
false or true	true
false or false	false
true or true	true

可以看出，一个 or 表达式要想为真，or 操作符的任何一侧为真即可，另一侧为真还是为假并不重要。

3.5.3 短路求值

and 和 or 操作符都会执行**短路求值 (short-circuit evaluation)**。其原理是：如果 and 操作符左侧的表达式为假，那么右侧的表达式根本不需要检查。因为只要有一个子表达式为假，复合表达式必然为假，检查剩下的表达式只会浪费 CPU 时间。所以，当 and 操作符发现它左边的表达式是假的，它就会“短路”，不再求值右侧的表达式。

or 操作符的短路求值与相似：如果 or 操作符左侧的表达式为真，那么右侧的表达式根本不需要检查。因为只要有一个子表达式为真，复合表达式必然为真，检查剩下的表达式只会浪费 CPU 时间。

3.5.4 not 操作符

not 操作符是一元操作符，它只取一个布尔表达式作为操作数，并取反其逻辑值。换言之，如果表达式为真，那么 not 操作符返回假；表达式为假则返回真。下面是一个使用了 not 操作符的 if 语句：

```
if not(temperature > 100):
    print('低于最大温度。')
```

首先测试表达式 `temperature > 100`，结果是一个真或假的值。然后，向结果值应用 `not` 操作符。如果表达式 `temperature > 100` 为真，那么 `not` 操作符返回假。如果表达式 `temperature > 100` 为假，那么 `not` 操作符返回真。上述代码等同于问“温度是否不大于 100？”



注意：本例将表达式 `temperature > 100` 括起来了。这是为了清楚地表明，是在对表达式 `temperature > 100` 的求值结果应用 `not` 操作符，而不是对 `temperature` 变量。

表 3-7 是 `not` 操作符的真值表。

表 3-7 `not` 操作符的真值表

表达式	求值结果
<code>not true</code>	<code>false</code>
<code>not false</code>	<code>true</code>

3.5.5 修订贷款资格判断程序

某些时候，`and` 操作符可以用来简化嵌套判断结构。例如，程序 3-5 的贷款资格判断程序使用了以下嵌套 `if-else` 语句。

```
if salary >= MIN_SALARY:
    if years_on_job >= MIN_YEARS:
        print('有资格贷款。')
    else:
        print(f'当前工作'
              f'至少满{MIN_YEARS}年，'
              f'才有资格贷款。')
else:
    print(f'年薪至少为'
          f'{MIN_SALARY:,.2f}美元，'
          f'才有资格贷款。')
```

该判断结构的目的是判断一个人是否年薪至少为 30000 美元，而且当前工作至少满了两年。程序 3-7 对它进行了简化。

程序 3-7 (loan_qualifier2.py)

```
1 # 这个程序判断银行客户是否
2 # 有资格获得贷款。
3
4 MIN_SALARY = 30000.0 # 最低年薪
5 MIN_YEARS = 2 # 最低工作年限
6
7 # 获取客户的年薪
8 salary = float(input('请输入年薪: '))
9
10 # 获取当前工作年限
11 years_on_job = int(input('请输入当前' +
12                          '工作年限: '))
13
14 # 判断客户是否符合贷款资格
15 if salary >= MIN_SALARY and years_on_job >= MIN_YEARS:
16     print('有资格贷款。')
17 else:
18     print('没有资格贷款。')
```

程序输出 (用户输入的内容加粗)

```
请输入年薪: 35000 
请输入当前工作年限: 1 
没有资格贷款。
```


程序输出 (用户输入的内容加粗)

```
请输入年薪: 25000 
请输入当前工作年限: 5 
没有资格贷款。
```

程序输出 (用户输入的内容加粗)

```
请输入年薪: 35000 
请输入当前工作年限: 5 
有资格贷款。
```

第 15 行~第 18 行的 `if-else` 语句测试复合表达式 `salary >= MIN_SALARY and years_on_job >= MIN_YEARS`。两个子表达式都为真，复合表达式才为真，并显示有资格贷款的消息。任何一个子表达式为假，复合表达式就为假，并显示没有资格贷款的消息。

 **注意：**细心的读者会发现，程序 3-7 与程序 3-5 相似，但不完全一致。如果用户不符合贷款资格，程序 3-7 只显示消息“没有资格贷款”，而程序 3-5 会解释用户不符合资格的两个原因之一。

3.5.6 另一个贷款资格判断程序

假设银行的客户被一家对贷款对象要求不严格的竞争银行抢走。作为回应，银行决定改变其贷款要求。现在，客户只需满足之前的任何一个条件，而不是两个条件。程序 3-8 展示了新的贷款资格判断程序的代码。第 15 行由 `if-else` 语句测试的复合表达式现在使用了 `or` 操作符。

程序 3-8 (`loan_qualifier3.py`)

```
1  # 这个程序判断银行客户是否
2  # 有资格获得贷款。
3
4  MIN_SALARY = 30000.0 # 最低年薪
5  MIN_YEARS = 2 # 最低工作年限
6
7  # 获取客户的年薪
8  salary = float(input('请输入年薪: '))
9
10 # 获取当前工作年限
11 years_on_job = int(input('请输入当前' +
12                          '工作年限: '))
13
14 # 判断客户是否符合贷款资格
15 if salary >= MIN_SALARY or years_on_job >= MIN_YEARS:
16     print('有资格贷款。')
17 else:
18     print('没有资格贷款。')
```

程序输出 (用户输入的内容加粗)

```
请输入年薪: 35000 
请输入当前工作年限: 5 
有资格贷款。
```

程序输出 (用户输入的内容加粗)

```
请输入年薪: 25000 
请输入当前工作年限: 5 
有资格贷款。
```

程序输出 (用户输入的内容加粗)

```
请输入年薪: 12000 
请输入当前工作年限: 1 
没有资格贷款。
```

3.5.7 用逻辑操作符检查数字范围

有时需要设计算法来判断一个数值是否在特定范围内，此时最好使用 `and` 操作符。例如，以下 `if` 语句检查 `x` 的值，判断它是否在 20~40（含）的范围内：

```
if x >= 20 and x <= 40:  
    print('该值在可接受的范围内。')
```

该语句测试的复合布尔表达式只有在 `x` 大于或等于 20 而且小于或等于 40 时才为真。只有 `x` 值在 20~40 的范围内，这个复合表达式才为真。

判断一个数值是否在范围之外时，则最好使用 `or` 操作符。以下语句判断 `x` 是否在 20~40（含）的范围之外：

```
if x < 20 or x > 40:  
    print('该值超出可接受的范围。')
```

在测试数值范围时，很重要的一点是不要混淆逻辑操作符的逻辑。例如，以下代码中的复合布尔表达式永远不会为真：

```
# 这是个错误!  
if x < 20 and x > 40:  
    print('该值超出可接受的范围。')
```

显然，`x` 不可能既小于 20，又大于 40。

检查点

3.14 什么是复合布尔表达式？

3.15 以下真值表展示了由逻辑操作符连接的真值和假值的各种组合。请圈出 T 或 F 来表示当前组合的结果是真还是假。

逻辑表达式	求值结果 (圈出 T 或 F)
True and False	T F
True and True	T F
False and True	T F
False and False	T F
True or False	T F
True or True	T F

False or True	T F
False or False	T F
not True	T F
not False	T F

3.16 假设变量 a=2, b=4, c=6。为以下每个条件圈出 T 或 F，说明表达式的值是真还是假。

a == 4 or b > 2 T F

6 <= c and a > 3 T F

1 != b and c != 3 T F

a >= -1 or a <= b T F

not (a > 2) T F

3.17 解释 and 和 or 操作符的短路求值工作方式。

3.18 写一个 if 语句，如果 speed 所引用的值在 0~200（含）的范围内，那么显示消息“数字有效”。

3.19 写一个 if 语句，如果 speed 所引用的值不在 0~200（含）的范围内，那么显示消息“数字无效”。

3.6 布尔变量

概念：布尔变量可以引用两个值之一：True 或 False。布尔变量通常作为标志使用，表示是否存在特定条件。

本书到目前为止已经使用了 int、float 和 str（字符串）类型的变量。除了这些数据类型，Python 还支持 bool 数据类型。这种类型的变量只能引用两个可能的值之一：True 或 False，即真或假。下例展示了如何为 bool 变量赋值。

```
hungry = True
sleepy = False
```

布尔变量经常作为标志使用。**标志**（flag）是一种特殊的变量，用于表明程序中是否存在某个条件。标志变量设为 False，表示该条件不存在；设为 True，则表示条件存在。

例如，假定一个销售人员的配额是 50000 美元。假设 `sales` 是销售人员已完成的销售额，以下代码可以判断是否完成配额。

```
if sales >= 50000.0:
    sales_quota_met = True
else:
    sales_quota_met = False
```

在这段代码中，`sales_quota_met` 变量可以作为一个标志来表明是否已完成销售配额。在程序的后期，可以用以下方式测试该标志：

```
if sales_quota_met:
    print('你已完成销售配额!')
```

如果 `bool` 变量 `sales_quota_met` 为 `True`，那么上述代码将显示'你已完成销售配额!'。注意，这里不必使用 `==` 操作符来显式比较 `sales_quota_met` 变量和 `True`。上述代码等同于：

```
if sales_quota_met == True:
    print('你已完成销售配额!')
```

检查点

3.20 可以为 `bool` 变量赋什么值？

3.21 什么是标志（flag）变量？

3.7 条件表达式

概念：可以使用三元操作符来写对一个 `if-else` 语句进行简化的条件表达式。

我们经常需要写一个 `if-else` 语句，将两个可能的值之一赋给变量，如下例所示。

```
if score > 59:
    grade = "Pass"
else:
    grade = "Fail"
```

上述代码将两个可能的值中的一个赋给 `grade` 变量。如果 `score` 大于 59，那么将字符串 "Pass" 赋给 `grade`，否则就将 "Fail" 赋给 `grade`。

Python 允许使用**条件表达式**来简化这种形式的代码，简单的 `if-else` 语句可以用这个技术进行简化。条件表达式测试一个布尔表达式，结果为真就赋一个值，为假则赋另一个值。下面是条件表达式的常规格式：

```
值1 if 条件 else 值2
```

其中，“条件”是要测试的布尔表达式。如果条件为真，那么条件表达式返回“值 1”，否则返回“值 2”。下例展示了如何将之前的 if-else 语句改写为条件表达式：

```
grade = "Pass" if score > 59 else "Fail"
```

这行代码中发生了不少事情，下面来仔细看看。首先，重要的是要理解这行代码是一个赋值语句。我们之所以知道它是一个赋值语句，因为它是这样开始的：

```
grade =
```

这表明要向 `grade` 变量赋值。如图 3-16 所示，`=`操作符右侧是一个条件表达式。所以，这里是要将条件表达式的值赋给 `grade` 变量。

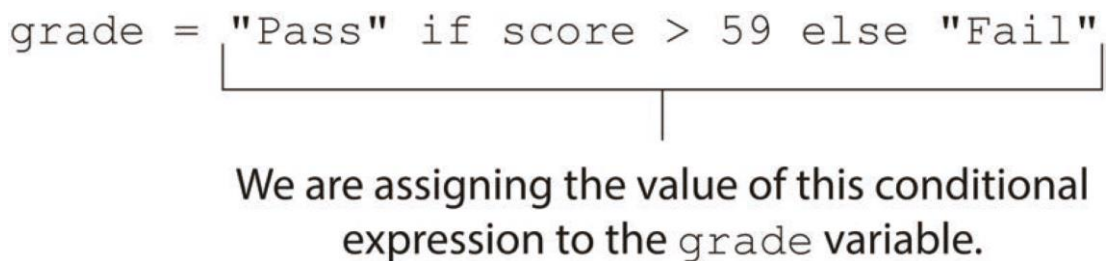


图 3-16 使用条件表达式

图中译文：

We are...: 将该条件表达式的值赋给 `grade` 变量

图 3-17 展示了条件表达式是如何工作的。首先测试布尔表达式 `score > 59`。结果为真，整个条件表达式将返回值 'Pass'；否则返回 'Fail'。

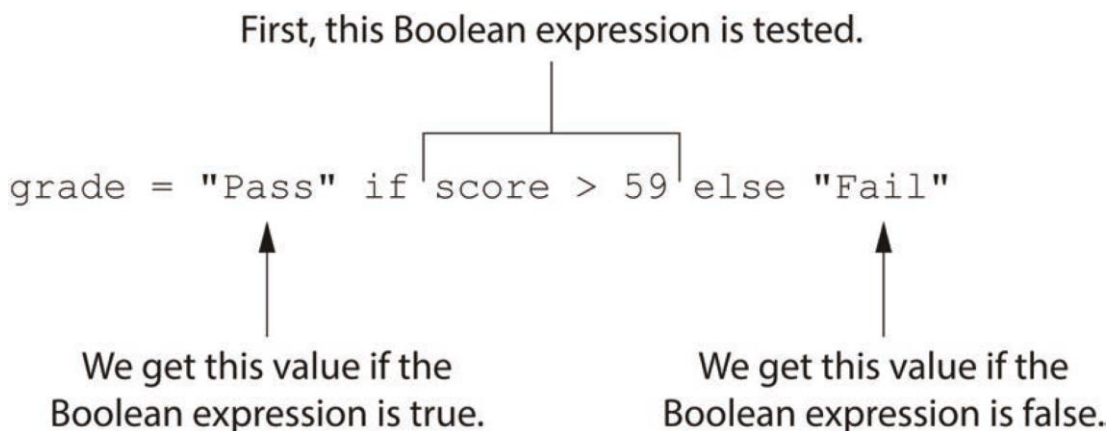


图 3-17 条件表达式的组成部分

图中译文：

First, this...: 首先测试这个布尔表达式

We get this value if the Boolean expression is true.: 布尔表达式为真，将得到这个值

We get this value if the Boolean expression is false.: 布尔表达式为假，将得到这个值

程序 3-9 展示了使用条件表达式的一个例子。用户被提示输入两个数字。第 4 行的语句使用一个条件表达式来判断哪个数字最大，并将该数字赋给 `max` 变量。

程序 3-9 (conditional_expression.py)

```
1 # 这个程序演示了条件表达式的用法
2 num1 = int(input('输入第一个数: '))
3 num2 = int(input('输入第二个数: '))
4 max = num1 if num1 > num2 else num2
5 print(f'较大的数是{max}。')
```

程序输出 (用户输入的内容加粗)

```
输入第一个数: 5 
输入第二个数: 10 
较大的数是 10。
```

3.8 赋值表达式和海象操作符

概念：可以用海象操作符 (`:=`) 来写赋值表达式，表达式将返回所赋的值。

你已经了解了赋值操作符 (`=`) 以及如何用它创建赋值语句，如下所示：

```
result = 27
```

赋值语句只做一件事：为变量赋值。上述赋值语句将值 27 赋给 `result` 变量。

除了 `=` 操作符，Python 还支持一个特殊的赋值操作符，称为**海象操作符**。海象操作符写成一个冒号并后跟一个等号，中间无空格，如下所示：

```
:=
```

之所以称为海象操作符，是因为偏过头看，它就像一张海象的脸。

可以使用海象操作符来创建**赋值表达式**，这种表达式能做两件事情：

-
- 向变量赋值
 - 返回赋给该变量的值

下面是使用了海象操作符的一个赋值表达式的例子：

```
result := 27
```

这个赋值表达式做了下面这些事情：

- 向 `result` 变量赋值 27
- 返回值 27

由于这个赋值表达式会返回一个值，所以它可以成为一个更大的语句的一部分。赋值表达式返回的值可由更大的语句所用。下面是一个例子：

```
print(result := 27)
```

上述语句将一个赋值表达式作为实参传给 `print` 函数。下面是该语句的工作方式：

- 向 `result` 变量赋值 27
- 赋值表达式返回值 27
- `print` 函数显示赋值表达式返回的值，即 27

图 3-18 对此进行了展示。

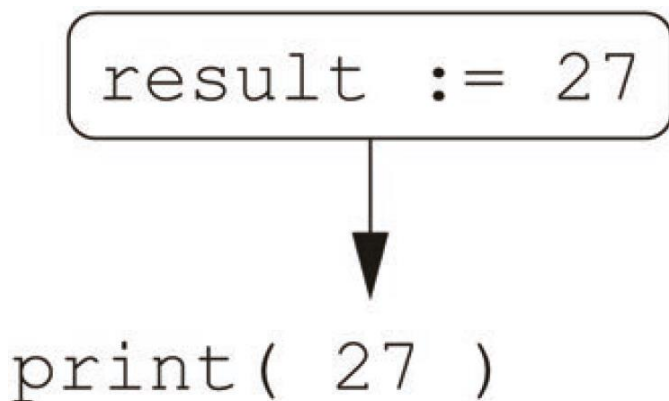


图 3-18 打印赋值表达式的值

赋值表达式可以用在很多地方，包括 `if` 语句所测试的布尔条件。下面是一个例子：

```
if (pay := hours * pay_rate) > 40:  
    print('你加班了。')
```

这个 `if` 语句的工作方式是：

- 将 `hours * pay_rate` 的值赋给 `pay`。
- 如果 `pay` 大于 40，那么显示消息“你加班了”。

上述代码等价于：

```
pay = hours * pay_rate
if pay > 40:
    print('你加班了。')
```



注意：赋值表达式并非完整语句，它必须作为一个更大语句的一部分来写。这正是为什么在单独一行中写这种表达式会报告语法错误的原因。

海象操作符的优先级

海象操作符在 Python 的所有操作符中具有最低的优先级。这意味着在一个较大的表达式中使用海象操作符时，如果同时存在其他操作符，那么海象操作符会最后求值。大多数时候，需要在赋值表达式两边加上括号，以确保海象操作符将正确的值赋给它的变量。

再来看看之前显示的 `if` 语句：

```
if (pay := hours * pay_rate) > 500:
    print('你加班了。')
```

注意，布尔条件同时使用了海象操作符、乘法操作符和大于操作符：

```
(pay := hours * pay_rate) > 500
```

还要注意，这里将赋值表达式括起来了，目的是确保将 `hours * pay_rate` 的值（工作时间和时薪的乘积）赋给 `pay` 变量。相反，如果不把赋值表达式括起来，即：

```
pay := hours * pay_rate > 500
```

那么会将表达式 `(hours * pay_rate > 500)` 的值赋给 `pay` 变量，这个值可能是 `True` 或 `False`。如果希望将实际的工资金额赋给 `pay` 变量，那么必须在赋值表达式两边加上括号。



提示：任何时候使用海象操作符创建赋值表达式时，最好都在赋值表达式两边加上括号。

3.9 海龟图形：判断海龟的状态

概念：海龟图形库提供了许多函数供判断海龟的状态并有条件地执行操作。

可以使用海龟图形库中的一些函数来了解有关海龟当前状态的大量信息。本节将讨论用于判断海龟位置、前进方向、笔是抬起还是放下、当前绘画颜色等的函数。

3.9.1 判断海龟位置

第 2 章讲过，可以使用 `turtle.xcor()` 和 `turtle.ycor()` 函数来获取海龟当前的 X 和 Y 坐标。以下代码使用 `if` 语句来判断海龟的 X 坐标是否大于 249，或者 Y 坐标是否大于 349。如果是，就将海龟重新定位到(0, 0)：

```
if turtle.xcor() > 249 or turtle.ycor() > 349:
    turtle.goto(0, 0)
```

3.9.2 判断海龟朝向

`turtle.heading()` 函数返回海龟的朝向。默认情况下，返回的朝向以度数为单位。以下交互会话对此进行了演示：

```
>>> import turtle 
>>> turtle.heading() 
0.0
>>>
```

以下代码片段使用 `if` 语句来判断海龟的朝向是否在 90 度~270 度之间。如果是，那么将海龟的朝向设为 180 度：

```
if turtle.heading() >= 90 and turtle.heading() <= 270:
    turtle.setheading(180)
```

3.9.3 判断笔是否放下

如果海龟画笔放下，那么 `turtle.isdown()` 函数返回 `True`，否则返回 `False`。以下交互会话对此进行了演示：

```
>>> turtle.isdown() 
True
>>>
```

以下代码使用 `if` 语句来判断海龟画笔是否放下。如果笔处于放下状态，那么代码会将其抬起：

```
if turtle.isdown():
    turtle.penup()
```

要判断笔是否抬起，可以连同 `turtle.isdown()` 函数使用 `not` 操作符，如以下代码所示：

```
if not(turtle.isdown()):
```

```
turtle.pendown()
```

3.9.4 判断海龟是否可见

如果海龟可见，那么 `turtle.isvisible()` 函数返回 `True`，否则返回 `False`，如以下交互会话所示：

```
>>> turtle.isvisible()   
True  
>>>
```

以下代码使用 `if` 语句来判断海龟是否可见。如果海龟可见，那么代码将其隐藏：

```
if turtle.isvisible():  
    turtle.hideturtle()
```

3.9.5 判断当前颜色

如果执行 `turtle.pencolor()` 函数而不传递任何参数，那么函数会将画笔的当前绘画颜色作为一个字符串返回，如以下交互会话所示：

```
>>> turtle.pencolor()   
'black'  
>>>
```

以下代码使用 `if` 语句来判断画笔当前的绘画颜色是否为红色。如果是，那么代码将其更改为蓝色：

```
if turtle.pencolor() == 'red':  
    turtle.pencolor('blue')
```

如果执行 `turtle.fillcolor()` 函数而不传递任何参数，那么函数会将画笔的当前填充颜色作为一个字符串返回，如以下交互会话所示：

```
>>> turtle.fillcolor()   
'black'  
>>>
```

以下代码使用 `if` 语句来判断画笔当前的填充颜色是否为蓝色。如果是，那么代码将其更改为白色：

```
if turtle.fillcolor() == 'blue':  
    turtle.fillcolor('white')
```

如果执行 `turtle.bgcolor()` 函数而不传递任何参数，那么函数会将海龟图形窗口的当前背景颜色作为一个字符串返回，如以下交互会话所示：

```
>>> turtle.bgcolor() 
'white'
>>>
```

以下代码使用 `if` 语句来判断当前背景颜色是否为白色。如果是，那么代码将其更改为灰色：

```
if turtle.bgcolor() == 'white':
    turtle.bgcolor('gray')
```

3.9.6 判断画笔大小

如果执行 `turtle.pensize()` 函数而不传递任何参数，那么函数将返回画笔的当前大小。如以下交互会话所示：

```
>>> turtle.pensize() 
1
>>>
```

以下代码使用 `if` 语句来判断画笔的当前大小是否小于 3。如果是，那么代码将其更改为 3：

```
if turtle.pensize() < 3:
    turtle.pensize(3)
```

3.9.7 判断海龟的动画速度

如果执行 `turtle.speed()` 函数而不传递任何参数，那么函数将返回海龟当前的动画速度。如以下交互会话所示：

```
>>> turtle.speed() 
3
>>>
```

第 2 章讲过，海龟的动画速度是 0~10 的一个值。如果速度为 0，那么动画会被禁用，海龟会立即做出所有动作。如果速度在 1~10 之间，那么 1 最慢，10 最快。

以下代码使用 `if` 语句来判断海龟的速率是否大于 0。如果是，那么代码将其设为 0：

```
if turtle.speed() > 0:
    turtle.speed(0)
```

以下代码展示了另一个例子。它使用 `if-elif-else` 语句来判断海龟的速度，并相应地设置画笔颜色。如果速度为 0，那么将画笔颜色设为红色。否则，如果速度大于 5，那么将画笔颜色设为蓝色。如果以上两个条件都不满足，那么将画笔颜色设为绿色。

```
if turtle.speed() == 0:
    turtle.pencolor('red')
elif turtle.speed() > 5:
```

```
turtle.pencolor('blue')
else:
    turtle.pencolor('green')
```

聚光灯：射击游戏



本节将研究一个使用海龟图形来玩一个简单的 Python 游戏。程序运行时会出现如图 3-19 所示的屏幕。右上方的的小方块是目标。游戏目标是将海龟发射出去，使其击中目标。可以通过在 Python shell 窗口中输入角度和力道值来完成此操作。是的，游戏确实有点简陋。但是，所有好的东西刚开始都是朴实无华的。程序会将海龟的朝向设为指定的角度，并在一个简单的公式中使用指定的力道值来计算海龟行进的距离。力道越大，海龟行进的距离就越远。如果海龟正好停在方块内，那么就显示已击中目标。

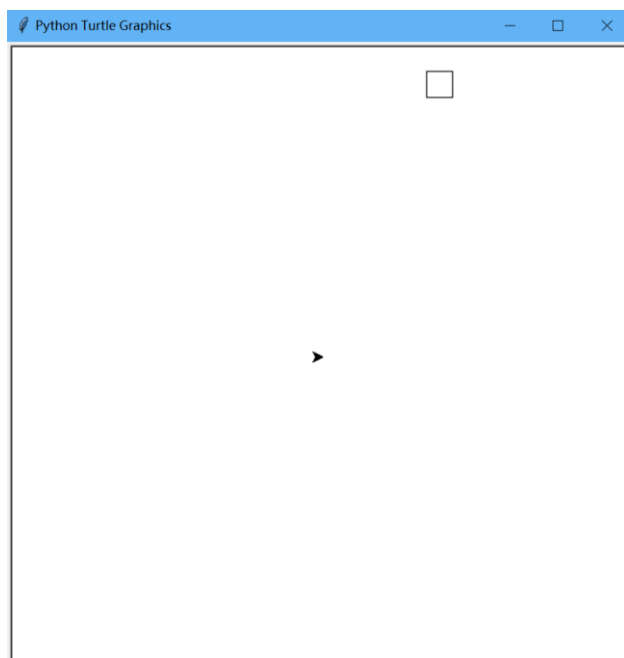


图 3-19 射击游戏

例如，图 3-20 展示了程序的一次会话，我们输入 45 作为角度，8 作为力道。如你所见，海龟没有命中目标。在图 3-21 中，我们再次运行程序，这次输入 67 作为角度，9.8 作为力道。这一次命中了目标。程序 3-10 展示了程序的完整代码。

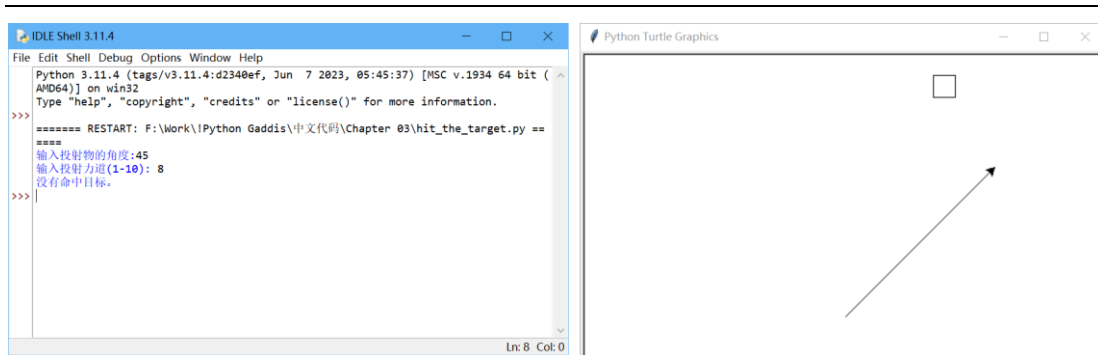


图 3-20 未命中目标

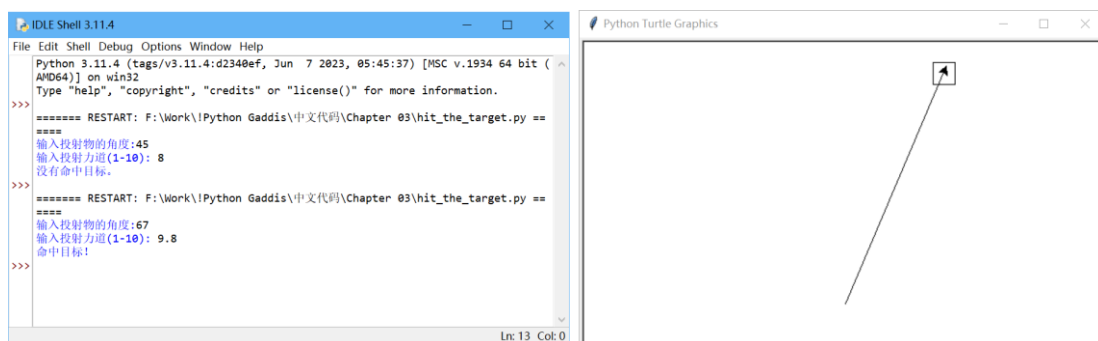


图 3-21 命中目标

程序 3-10 (hit_the_target.py)

```

1  # 射击游戏
2  import turtle
3
4  # 具名常量
5  SCREEN_WIDTH = 600 # 屏幕宽度
6  SCREEN_HEIGHT = 600 # 屏幕高度
7  TARGET_LLEFT_X = 100 # 目标左下 X
8  TARGET_LLEFT_Y = 250 # 目标左下 Y
9  TARGET_WIDTH = 25 # 目标的宽度
10 FORCE_FACTOR = 30 # 力道系数
11 PROJECTILE_SPEED = 1 # 投射物（海龟）的动画速度
12 NORTH = 90 # 北向角度
13 SOUTH = 270 # 南向角度
14 EAST = 0 # 东向角度
15 WEST = 180 # 西向角度
16
17 # 设置窗口

```

```

18 turtle.setup(SCREEN_WIDTH, SCREEN_HEIGHT)
19
20 # 绘制目标
21 turtle.hideturtle()
22 turtle.speed(0)
23 turtle.penup()
24 turtle.goto(TARGET_LLEFT_X, TARGET_LLEFT_Y)
25 turtle.pendown()
26 turtle.setheading(EAST)
27 turtle.forward(TARGET_WIDTH)
28 turtle.setheading(NORTH)
29 turtle.forward(TARGET_WIDTH)
30 turtle.setheading(WEST)
31 turtle.forward(TARGET_WIDTH)
32 turtle.setheading(SOUTH)
33 turtle.forward(TARGET_WIDTH)
34 turtle.penup()
35
36 # 海龟居中
37 turtle.goto(0, 0)
38 turtle.setheading(EAST)
39 turtle.showturtle()
40 turtle.speed(PROJECTILE_SPEED)
41
42 # 从用户处获取角度和力道
43 angle = float(input("输入投射物的角度:"))
44 force = float(input("输入投射力道(1-10): "))
45
46 # 计算距离
47 distance = force * FORCE_FACTOR
48
49 # 设置海龟朝向
50 turtle.setheading(angle)
51
52 # 绘制发射过程
53 turtle.pendown()
54 turtle.forward(distance)
55
56 # 命中目标了吗?
57 if (turtle.xcor() >= TARGET_LLEFT_X and
58     turtle.xcor() <= (TARGET_LLEFT_X + TARGET_WIDTH) and
59     turtle.ycor() >= TARGET_LLEFT_Y and
60     turtle.ycor() <= (TARGET_LLEFT_Y + TARGET_WIDTH)):
61     print('命中目标! ')
62 else:
63     print('没有命中目标。')

```

来仔细看看代码。第 5 行~第 15 行定义了一系列具名常量。

- 第 5 行和第 6 行定义了 `SCREEN_WIDTH` 和 `SCREEN_HEIGHT` 常量。将在第 14 行使用它

们将图形窗口的大小设为 600 像素宽×600 像素高。

- 第 7 行和第 8 行定义了 TARGET_LLEFT_X 和 TARGET_LLEFT_Y 常量。用于设定目标框左下角的(X,Y)坐标。
- 第 9 行定义了 TARGET_WIDTH 常量，代表目标框的宽度（和高度）。
- 第 10 行定义了 FORCE_FACTOR 常量，代表是一个力道系数，我们在公式中使用它来计算投射物发射后行进的距离。
- 第 11 行定义了 PROJECTILE_SPEED 常量，代表海龟（投射物）的动画速度。
- 第 12-15 行定义了 NORTH, SOUTH, EAST 和 WEST 常量，在绘制目标框时，将使用它们作为北、南、东、西方向的角度。

第 21 行~第 34 行绘制目标框：

- 第 21 行隐藏海龟，因为在绘制好目标框之前不需要看到它。
- 第 22 行将海龟的动画速度设为 0，这会禁用海龟动画。这样做是为了让目标框立即出现。
- 第 23 行抬起了海龟的画笔，这样将其从默认位置（窗口中心）移动到开始绘制目标框的位置时，它不会画一条线。
- 第 24 行将海龟移动到目标框左下角的位置。
- 第 25 行放下画笔，这样海龟会在移动它时画画。
- 第 26 行将海龟的朝向设为 0 度，使其面朝东方。
- 第 27 行将海龟向前移动 25 像素，绘制目标框的底部边线。
- 第 28 行将海龟的朝向设为 90 度，使其面朝北方。
- 第 29 行将海龟向前移动 25 像素，绘制目标框的右侧边线。
- 第 30 行将海龟的朝向设为 180 度，使其面朝西方。
- 第 31 行将海龟向前移动 25 像素，绘制目标框的顶部边线。
- 第 32 行将海龟的朝向设为 270 度，使其面朝南方。
- 第 33 行将海龟向前移动 25 像素，绘制目标框的左侧边线。
- 第 34 行抬起海龟的画笔，避免将其移回窗口中心时画一条线。

第 37 行~第 40 行将海龟移回窗口中心：

- 第 37 行将海龟移动到(0,0)。
- 第 38 行将海龟的朝向设为 0 度，使其朝东。
- 第 39 行显示海龟。
- 第 40 行将海龟的动画速度设为 1，该速度足够慢，足以在射击时看到投射物的移动。

第 43 行和第 44 行从用户处获取射击角度和力道：

- 第 43 行提示用户输入投射物的角度。输入的值将转换为浮点数并赋给 angle 变量。
- 第 44 行提示用户输入力道，范围为 1~10。输入的值将转换为浮点数并赋给 force 变量。我们将在第 47 行中使用力道值来计算投射物的行进距离。力道越大，行进得越

远。

第 47 行计算海龟的行进距离，并将该值赋给 `distance` 变量。距离的计算方法是将用户的力道乘以 `FORCE_FACTOR` 常量。之所以将 `FORCE_FACTOR` 常量设为 30，是因为从海龟到窗口边缘的距离是 300 像素（或者稍微多一点，具体取决于海龟的朝向）。如果用户输入 10 作为力道值，海龟将移动到屏幕边缘。

第 50 行将海龟的朝向设为用户在第 43 行输入的角度。第 53 行和第 54 行开始海龟绘图：

- 第 53 行放下画笔，以便在移动时画线。
- 第 54 行使海龟向前移动第 47 行计算好的距离。

最后要做的是判断海龟是否命中目标。如果海龟此时位于目标框内部，那么以下所有条件都为真：

- 海龟的 X 坐标将大于或等于 `TARGET_LLEFT_X`
- 海龟的 X 坐标将小于或等于 `TARGET_LLEFT_X + TARGET_WIDTH`
- 海龟的 Y 坐标将大于或等于 `TARGET_LLEFT_Y`
- 海龟的 Y 坐标将小于或等于 `TARGET_LLEFT_Y + TARGET_WIDTH`

第 57 行~第 63 行的 `if-else` 语句判断是否所有这些条件都为真。如果是，那么第 61 行显示消息“命中目标!”；否则第 63 行显示消息“没有命中目标。”

检查点

3.22 如何获取海龟的 X 和 Y 坐标？

3.23 如何判断画笔是否抬起？

3.24 如何获得海龟当前朝向？

3.25 如何判断海龟是否可见？

3.26 如何判断画笔颜色？如何判断当前填充颜色？如何判断海龟图形窗口的当前背景颜色？

3.27 如何判断画笔当前大小？

3.28 如何判断海龟当前动画速度？

复习题

选择题

1. _____ 结构仅在特定条件下才执行一组语句。

a. 顺序 b. 条件 c. 判断 d. 布尔

2. _____结构提供了单分支执行路径。

a. 顺序 b. 单分支判断 c. 单分支路径 d. 单执行判断

3. _____表达式的值为 True 或 False。

a. 二元 b. 判断 c. 无条件 d. 布尔

4. >, <和==是_____操作符。

a. 关系 b. 逻辑 c. 条件 d. 三元

5. _____结构测试一个条件, 条件为真选择一个路径, 为假则选择另一个路径。

a. if 语句 b. 单分支判断 c. 双分支判断 d. 顺序

6. 使用_____语句来写单分支判断结构。

a. test-jump b. if c. if-else d. if-call

7. 使用_____语句来写双分支判断结构。

a. test-jump b. if c. if-else d. if-call

8. and, or 和 not 是_____操作符。

a. 关系 b. 逻辑 c. 条件 d. 三元

9. 只有两个子表达式都为真, 使用_____操作符创建的复合布尔表达式才为真。

a. and b. or c. not d. both

10. 两个子表达式任何一个为真, 使用_____操作符创建的复合布尔表达式就为真。

a. and b. or c. not d. either

11. _____操作符获取一个布尔表达式作为操作数, 并取反其逻辑值。

a. and b. or c. not d. either

12. _____是一种布尔变量, 用于表明程序中是否存在某个条件。

a. 标志 b. 信号 c. 哨兵 d. 警报

判断题

1. 仅使用顺序结构就能编写任何程序。
2. 程序只能选用一种类型的控制结构。无法组合结构。
3. 单分支判断结构将测试一个条件，如果条件为真选择一个路径，条件为假则选择另一个。
4. 一个判断结构可以嵌套在另一个判断结构中。
5. 仅当两个子表达式都为真时，使用 `and` 操作符创建的复合布尔表达式才为真。

简答题

1. 解释“条件执行”的含义。
2. 需要测试一个条件，在条件为真时执行一组语句，条件为假则执行另一组语句。为此应该使用什么结构？
3. 简要描述 `and` 操作符的工作方式。
4. 简要描述 `or` 操作符的工作方式。
5. 为了判断一个数是否在某个范围内，最好使用哪种逻辑操作符？
5. 什么是“标志”（flag）？它是如何工作的？

算法工作台

1. 写一个 `if` 语句，在变量 `x` 大于 `100` 的情况下将 `20` 赋给变量 `y`，并将 `40` 赋给变量 `z`。
2. 写一个 `if` 语句，在变量 `a` 小于 `10` 的情况下将 `0` 赋给变量 `b`，并将 `1` 赋给变量 `c`。
3. 写一个 `if-else` 语句，在变量 `a` 小于 `10` 的情况下将 `0` 赋给变量 `b`；否则将 `99` 赋给变量 `b`。
4. 以下代码包含几个嵌套 `if-else` 语句。遗憾的是，它们的对齐和缩进不正确。请重写代码来正确对齐和缩进。

```
if score >= A_score:
    print('你的成绩为 A.')
```

```
else:
    if score >= B_score:
        print('你的成绩为 B.')
```

```
    else:
        if score >= C_score:
            print('你的成绩为 C.')
```

```
        else:
            if score >= D_score:
                print('你的成绩为 D.')
```

```
else:  
    print('你的成绩为 F.')
```

5. 写嵌套判断结构来执行以下操作：如果 `amount1` 大于 10 并且 `amount2` 小于 100，那么显示 `amount1` 和 `amount2` 中较大的那个。
6. 写一个 `if-else` 语句，如果 `speed` 变量的值在 24~56（含）之间，那么显示“速度正常”；超出范围则显示“速度异常”。
7. 写一个 `if-else` 语句来判断 `points` 变量是否在 9~51（含）的范围内。如果超出范围，那么显示“积分无效”；否则显示“积分有效”。
8. 写一个 `if` 语句，使用海龟图形库来判断海龟的朝向是否为 0 度~45 度（含）。如果是，就抬起画笔。
9. 写一个 `if` 语句，使用海龟图形库来判断画笔颜色是否为红色或蓝色。如果是，就将画笔大小设为 5 像素。
10. 写一个 `if` 语句，使用海龟图形库来判断海龟是否在一个矩形的内部。矩形左上角位于 (100, 100)，右下角位于 (200, 200)。如果海龟在矩形内，就隐藏海龟。

编程练习

1. 星期几

写一个程序，要求用户输入 1~7（含）的数字。程序应显示该数字对应星期几。其中，1 = 星期一，2 = 星期二，3 = 星期三，4 = 星期四，5 = 星期五，6 = 星期六，7 = 星期日。如果用户输入的数字超出 1~7 的范围，那么应显示错误消息。

2. 矩形面积

 视频讲解：The Areas of Rectangles Problem

矩形的面积是矩形长度乘以宽度。写一个程序，要求输入两个矩形的长度和宽度。程序应指出哪个矩形的面积更大，或者面积是否相同。

3. 年龄分类

写一个程序，要求输入一个人的年龄。程序应显示一条消息，指出这个人是婴儿、儿童、青少年还是成人。规则如下：

- 1 岁或以下是婴儿。
- 大于 1 岁但小于 13 岁是儿童。
- 年满 13 岁但小于 20 岁是青少年。

- 年满 20 岁就是成年人了。

4. 罗马数字

写一个程序，提示用户输入 1~10（含）的数字。程序应显示和这个数字对应的罗马数字。如果数字超出 1~10 的范围，那么程序应显示错误消息。下表列出了和 1~10 对应的罗马数字。

阿拉伯数字	罗马数字
1	I
2	II
3	III
4	IV
5	V
6	VI
7	VII
8	VIII
9	IX
10	X

5. 质量和重量

科学家以千克为单位测量物体的质量，以牛顿为单位测量物体的重量。给定物体的质量（单位：千克），可以使用以下公式来计算其重量（单位：牛顿）。

$$\text{重量} = \text{质量} \times 9.8$$

写一个程序，要求输入物体的质量，然后计算其重量。重量超过 500 牛顿，显示一条消息来指出该物体太重。重量小于 100 牛顿，则显示消息来指出物体太轻。

6. 神奇日期

1960 年 6 月 10 日是一个特别的日子，因为如果用以下格式书写，那么月份乘以天数等于年份。

6/10/60

写一个程序，要求输入月份（以数字形式）、天数和两位数的年份。然后，程序应判断月份乘以天数是否等于年份。如果是，那么应该显示一条消息，说明这是一个“神奇日期”。否则，它应该显示一条消息，说明这个日期一点都不神奇。

7. 混色器

红色、黄色和蓝色称为“原色”（primary color），因为它们不能通过混合其他颜色而获得。混合两种原色时，会得到一种“间色”（secondary color），如下所示：

- 红色和蓝色混合得到紫色
- 红色和黄色混合得到橙色
- 蓝色和黄色混合得到绿色

设计一个程序，提示输入要混合的两种原色的名称。如果用户输入“红色”、“蓝色”或“黄色”以外的任何内容，那么程序应显示一条错误消息。否则，程序应显示混合而成的间色名称。

8. 热狗野餐计算器

假设热狗每包有 10 个，热狗面包则每包有 8 个。写一个程序，计算一次野餐需要多少包热狗和多少包热狗面包，同时需要满足浪费最少这一条件。程序应询问野餐人数以及每人需要的热狗数量。程序应显示以下详细信息：

- 至少需要多少包热狗
- 至少需要多少包热狗面包
- 用剩的热狗数
- 用剩的热狗面包数

9. 轮盘颜色

玩（欧式）轮盘赌时，轮盘上的口袋从 0 到 36 编号。口袋颜色如下所示：

- 0 号袋为绿色。
- 1~10 号袋，奇数袋红色，偶数袋黑色。
- 11~18 号袋，奇数袋为色，偶数袋红色。
- 19~28 号袋，奇数袋红色，偶数袋黑色。
- 29~36 号袋，奇数袋黑色，偶数袋红色。

写一个程序，要求用户输入口袋号码并显示口袋是绿色、红色还是黑色。如果用户输入的数字超出 0~36（含）的范围，那么程序应显示错误消息。

10. 数硬币游戏

创建一个数硬币游戏，让用户输入用各种硬币如何凑出 1 美元。美国的硬币包括：Penny

(1分)、Nickel (5分)、Dime (1角)、Quarter (25分)。程序应提示用户分别输入 Penny, Nickel, Dime 和 Quarter 硬币的数量。如果输入的硬币总价值等于 1 美元, 就恭喜用户取得了胜利。否则, 程序应显示一条消息, 指示输入的金额是多于还是少于一美元。

11. 读书俱乐部积分

Serendipity Booksellers 有一个读书俱乐部, 根据每月购买的书籍数量向顾客奖励积分。积分奖励规则如下所示:

- 购买 0 本书, 获得 0 积分。
- 购买 2 本书, 获得 5 积分。
- 购买 4 本书, 获得 15 积分。
- 购买 6 本书, 获得 30 积分。
- 购买 8 本书或以上, 获得 60 积分。

写一个程序, 要求用户输入本月购买的书籍数量, 然后显示奖励的积分。

12. 软件销售

某软件公司以零售价 99 美元销售软件。批量购买的折扣规则如下表所示。

数量	折扣
10~19	10%
20~49	20%
50~99	30%
100 或更多	40%

写一个程序, 要求用户输入购买的软件数量。然后, 程序应显示折扣金额 (如果有) 以及折后总金额。

13. 运费

Fast Freight 快递公司按以下费率收取运费:

包裹重量	每磅运费 (单位: 美元)
2 磅或不足 2 磅	1.50
2 磅或以上, 但不超过 6 磅	3.00
6 磅或以上, 但不超过 10 磅	4.00

10 磅或以上	4.75
---------	------

写一个程序，要求用户输入包裹重量，然后显示运费。

14. 计算 BMI

编写一个计算并显示身体质量指数（Body Mass Index, BMI）的程序。经常用 BMI 判断一个人是否超重或体重过轻。一个人的 BMI 可以通过以下公式计算：

$$\text{BMI} = \text{体重（公斤）} / \text{身高（米）}^2$$

程序应该要求用户输入体重和身高，然后显示用户的 BMI。程序还应显示一条消息，指出体重是在理想范围内、过轻还是超重。如果 BMI 在 18.5~25 之间，那么这个人的体重被认为是理想的。BMI 小于 18.5，被认为体重过轻。BMI 大于 25，则被认为超重。

15. 时间计算器

写一个程序，要求输入秒数，并视情况执行以下操作。

- 一分钟有 60 秒。如果输入的秒数大于或等于 60，那么程序应将秒数转换为分钟和秒。
- 一小时有 3600 秒。如果用户输入的秒数大于或等于 3600，那么程序应将秒数转换为小时、分钟和秒。
- 一天有 86400 秒。如果用户输入的秒数大于或等于 86400，那么程序应将秒数转换为天、小时、分钟和秒。

16. 二月有多少天

二月通常有 28 天，但闰年的二月有 29 天。写一个程序，要求输入年份。然后显示当年二月有多少天。根据以下标准来判断闰年：

- 判断该年份是否能被 100 整除。如果能，那么当且仅当它同时能被 400 整除时，它才是闰年。例如，2000 年是闰年，但 2100 年不是。
- 如果年份不能被 100 整除，那么当且仅当它能被 4 整除时，它才是闰年。例如，2008 年是闰年，但 2009 年不是。

下面是程序的一次示例运行：

```
输入年份: 2008 
2008 年 2 月有 29 天。
```

17. Wi-Fi 诊断树

图 3-22 展示了对 Wi-Fi 连接问题进行故障诊断的一个简化流程图。基于该流程图来创建一个程序，引导用户完成解决 Wi-Fi 连接问题的步骤。以下是程序的一次示例运行：

重启电脑并尝试重连。

问题解决了吗？ 否

重启路由器并尝试重连。

问题解决了吗？ 是

注意，一旦问题得到解决，程序就会结束。下面是程序的另一次示例运行：

重启电脑并尝试重连。

问题解决了吗？ 否

重启路由器并尝试重连。

问题解决了吗？ 是

确定路由器和光猫之间的网线正常连接。

问题解决了吗？ 否

将路由器换到其他地方。

问题解决了吗？ 否

更换路由器。

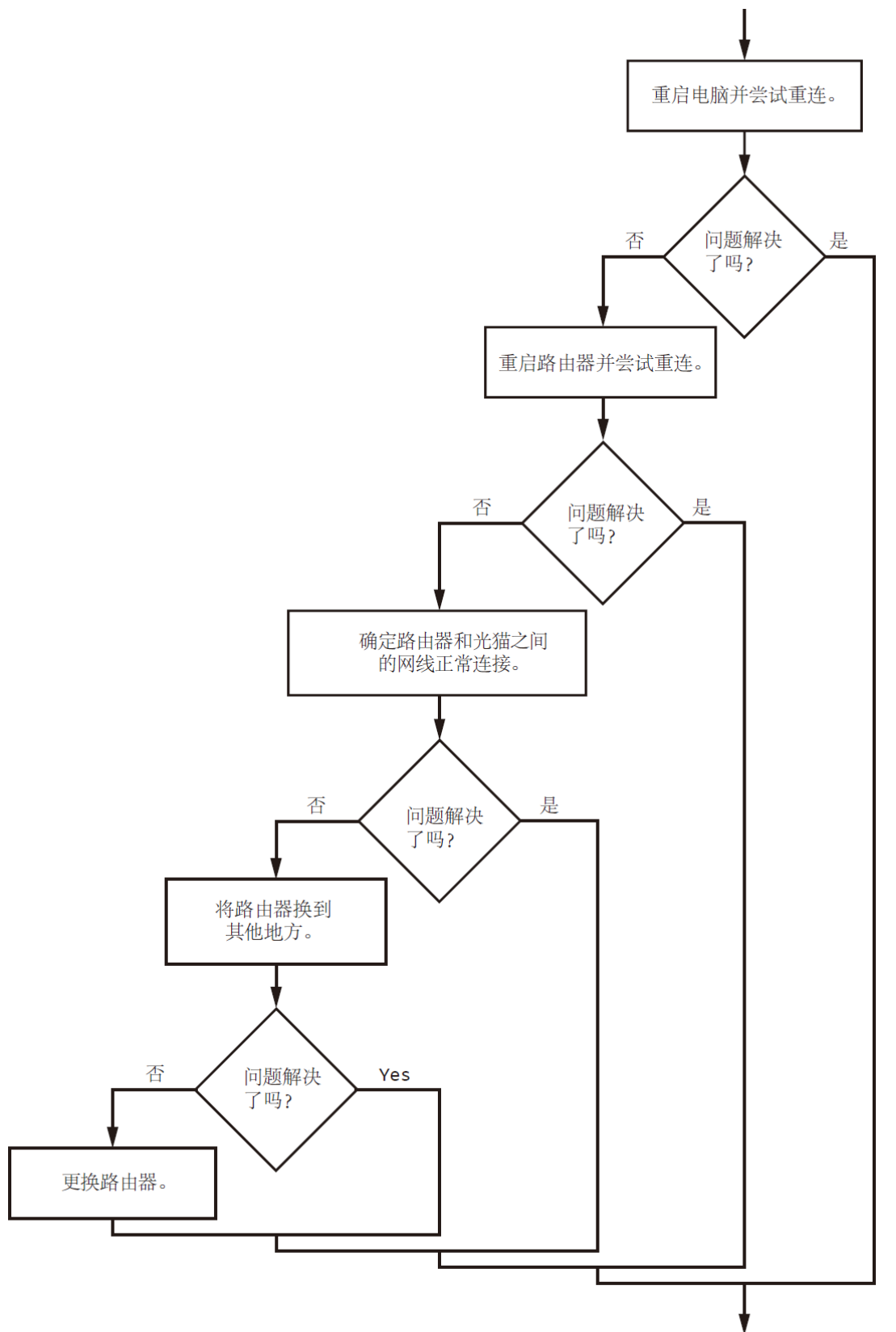


图 3-22 Wi-Fi 连接不良故障处理

17. 选择餐馆

一群朋友来参加你的高中聚会，你想带他们去当地餐馆吃饭，但不确定他们是否有忌口。可供选择的餐馆包括：^①

乔的美食汉堡店——素食者：否，纯素食者：否，无麸质：否
主街比萨公司——素食者：是，纯素食者：否，无麸质：是
角落咖啡馆——素食者：是，纯素食者：是，无麸质：是
老妈意大利餐厅——素食者：是，纯素食者：否，无麸质：否
厨师的厨房——素食者：是，纯素食者：是，无麸质：是

写一个程序，询问朋友中有没有素食者或纯素食者，或者要求无麸质，并列出现所有朋友的餐馆。

下面是程序的一次示例运行：

```
是否有人是素食者？ 否   
是否有人是纯素食者？ 否   
是否有人要求无麸质食物？ 是   
这是您的餐厅选择：  
    主街比萨公司  
    角落咖啡馆  
    厨师的厨房
```

下面是程序的另一次示例运行：

```
是否有人是素食者？ 是   
是否有人是纯素食者？ 是   
是否有人要求无麸质食物？ 是   
这是您的餐厅选择：  
    角落咖啡馆  
    厨师的厨房
```

18. 海龟图形：修改射击游戏

对程序 3-10 的 `hit_the_target.py` 游戏进行改进，当投射物未命中目标时，向用户显示提示，指出应该增大或减小角度和/或力道值。例如，程序应显示“尝试更大角度”和“使用较小的力道”等消息。

^① 译注：素食者（vegetarian）是一般意义上不吃肉的素食主义者。但这类素食主义者一般情况下还是会吃鸡蛋、蜂蜜、牛奶等由动物衍生出来的产品。纯素食者（vegan）就更加严格了，不但不吃肉，上面所提到的动物衍生产品他们也一概不碰。

第 4 章 重复结构

4.1 重复结构简介

概念：重复结构导致一个或一组语句重复执行。

程序员经常需要写重复执行同一个任务的代码。例如，假设要写一个程序来计算多个销售人员的 10%销售佣金（提成）。笨办法是先写代码来计算一个销售人员的佣金，然后为每个销售人员都复制粘贴该代码。下面展示了一个例子。

```
# 获取销售人员的销售额和佣金率
sales = float(input('输入销售额 '))
comm_rate = float(input('输入佣金率: '))
# 计算佣金
commission = sales * comm_rate
# 显示佣金
print(f'佣金为${commission:,.2f}')

# 获取另一个销售人员的销售额和佣金率
sales = float(input('输入销售额 '))
comm_rate = float(input('输入佣金率: '))
# 计算佣金
commission = sales * comm_rate
# 显示佣金
print(f'佣金为${commission:,.2f}')

# 获取另一个销售人员的销售额和佣金率
sales = float(input('输入销售额 '))
comm_rate = float(input('输入佣金率: '))
# 计算佣金
commission = sales * comm_rate
# 显示佣金
print(f'佣金为${commission:,.2f}')
```

上述代码会一直持续...

如你所见，这段代码是一个包含大量重复代码的长顺序结构。这是一个“笨”办法，它有几个缺点。

- 重复的代码使程序变得臃肿。
- 写这么多语句可能非常耗时，即使是复制粘贴。
- 如果重复的代码有需要更正或修改的地方，那么需要在多处进行更正或修改。

为了重复执行一个操作，更好方法是只写一次代码，然后将其放到在一种结构中，使计算机根据需要多次重复，而不是多次复制代码。这可以通过**重复结构**（通常称为**循环**）来完成。

条件控制和计数控制循环

本章将讨论两种主要的循环结构：条件控制循环和计数控制循环。条件控制循环使用真/假条件来控制重复次数。计数控制循环则重复特定次数。在 Python 中，可以使用 `while` 语句和 `for` 语句来写这些类型的循环。本章将对这两者进行演示。

检查点

- 4.1 什么是重复结构？
- 4.2 什么是条件控制循环？
- 4.3 什么是计数控制循环？

4.2 while 循环：条件控制循环

概念：只要条件为真，条件控制循环就会导致一个或一组语句重复执行。在 Python 中，可以使用 `while` 语句来写条件控制循环。

 视频讲解：The while Loop

`while` 循环因其工作方式而得名：当（`while`）条件为真时，就执行某些任务。这种循环由两部分构成：（1）求值结果为真或假的条件；以及（2）在条件为真时重复执行的一个或一组语句。图 4-1 展示了 `while` 循环的逻辑。

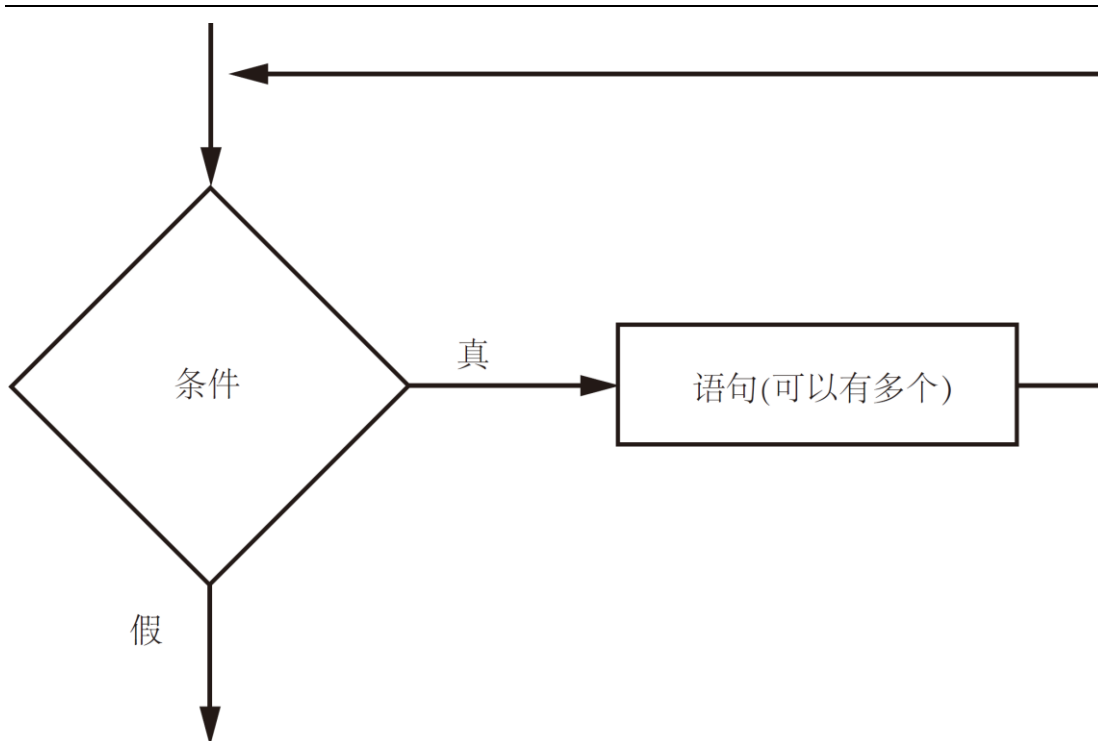


图 4-1 while 循环的逻辑

菱形代表要测试的条件。注意条件为真时发生的事情：执行一个或多个语句，完毕后回到菱形符号上方的位置。再次测试条件，如果为真，就重复该过程。如果条件为假，那么程序退出循环。在流程图中，只要看到一条流程线返回流程图之前的一部分，就表明存在一个循环。

Python 的 while 循环的常规格式如下所示：

```
while 条件:  
    语句  
    语句  
    ...
```

为简单起见，我们将第一行称为 **while 子句**。while 子句以关键字 **while** 开头，后跟求值为真或假的布尔条件。条件后是一个冒号。从下一行开始是语句块。第 3 章讲过，块（block）中的所有语句都必须一致地缩进。这种缩进是必须的，因为 Python 解释器根据它来判断块的开始和结束位置。

while 循环在执行时会先测试条件。条件为真，就执行 while 子句后面的块中的语句。执行完毕后开始下一次循环。条件为假，则直接退出循环，跳过整个块。程序 4-1 展示了如何使用 while 循环来重写本章开头描述的佣金计算程序。

程序 4-1 (commission.py)

```
1  # 该程序计算销售人员的佣金
2
3  # 创建一个变量来控制循环
4  keep_going = 'y'
5
6  # 计算一组佣金
7  while keep_going == 'y':
8      # 获取销售人员的销售额和佣金率
9      sales = float(input('输入销售额: '))
10     comm_rate = float(input('输入佣金率: '))
11
12     # 计算佣金
13     commission = sales * comm_rate
14
15     # 显示佣金
16     print(f'佣金为${commission:,.2f}')
17
18     # 判断用户是否想计算下一笔佣金
19     keep_going = input('要计算下一笔' +
20                        '佣金吗(是的话输入 y): ')
```

程序输出 (用户输入的内容加粗)

```
输入销售额: 10000.00 
输入佣金率: 0.10 
佣金为$1,000.00
要计算下一笔佣金吗(是的话输入 y): y 
输入销售额: 20000.00 
输入佣金率: 0.15 
佣金为$3,000.00
要计算下一笔佣金吗(是的话输入 y): y 
输入销售额: 12000.0 
输入佣金率: 0.10 
佣金为$1,200.00
要计算下一笔佣金吗(是的话输入 y): n 
```

第 4 行使用赋值语句创建一个名为 `keep_going` 的变量，并把它初始化为 `'y'`。这个初始化值很重要，稍后你就会明白为什么。

第 7 行开始 `while` 循环结构，它的第一行如下所示：

```
while keep_going == 'y':
```

注意测试的条件是 `keep_going=='y'`。如果结果为真，那么执行第 8 行~第 20 行的语句。

执行完毕后，从第 7 行开始下一次循环。继续测试表达式 `keep_going=='y'`，如果为真，那么再次执行第 8 行~第 20 行的语句。如此重复，直到第 7 行在测试表达式 `keep_going=='y'` 时发现结果为假。这时程序会退出循环。图 4-2 对此进行了演示。

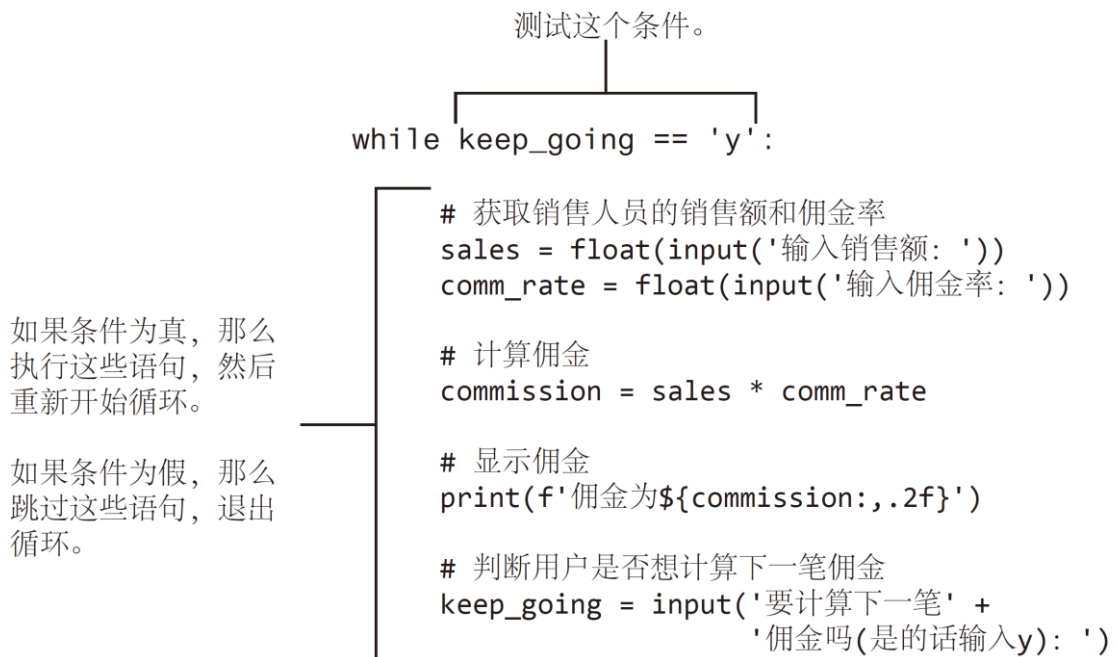


图 4-2 while 循环

要使循环停止，循环内部必须发生某事来使表达式 `keep_going == 'y'` 变成假。第 19 行~第 20 行解决的就是这个问题。该语句提示“要计算下一笔佣金吗(是的话输入 y)”。从键盘读取的值将赋给 `keep_going` 变量。如果用户输入 `y`（而且必须是小写的 `y`），那么当循环重新开始时，表达式 `keep_going == 'y'` 将为真。这导致循环体中的语句再次执行。但是，如果用户输入除小写 `y` 之外的其他任何内容，那么当循环重新开始时，表达式的求值结果将变成假，这导致程序退出循环。

理解了代码之后，再来看看示例程序输出。首先，用户输入 10000.00 作为销售额，输入 0.10 作为佣金率。然后，程序显示相应的佣金，即 1000 美元。接着，系统提示用户“要计算下一笔佣金吗(是的话输入 y)”。用户输入 `y`，开始下一次循环。在示例输出中，用户经历了该过程 3 次。循环体每次执行都称为一次**迭代**。在本例中，循环总共迭代了 3 次。

图 4-3 展示了程序 4-1 的流程图。流程图有一个用 `while` 循环写的重复结构。它测试条件 `keep_going=='y'`。条件为真，会执行一组语句，完毕后将返回条件测试上方的位置。

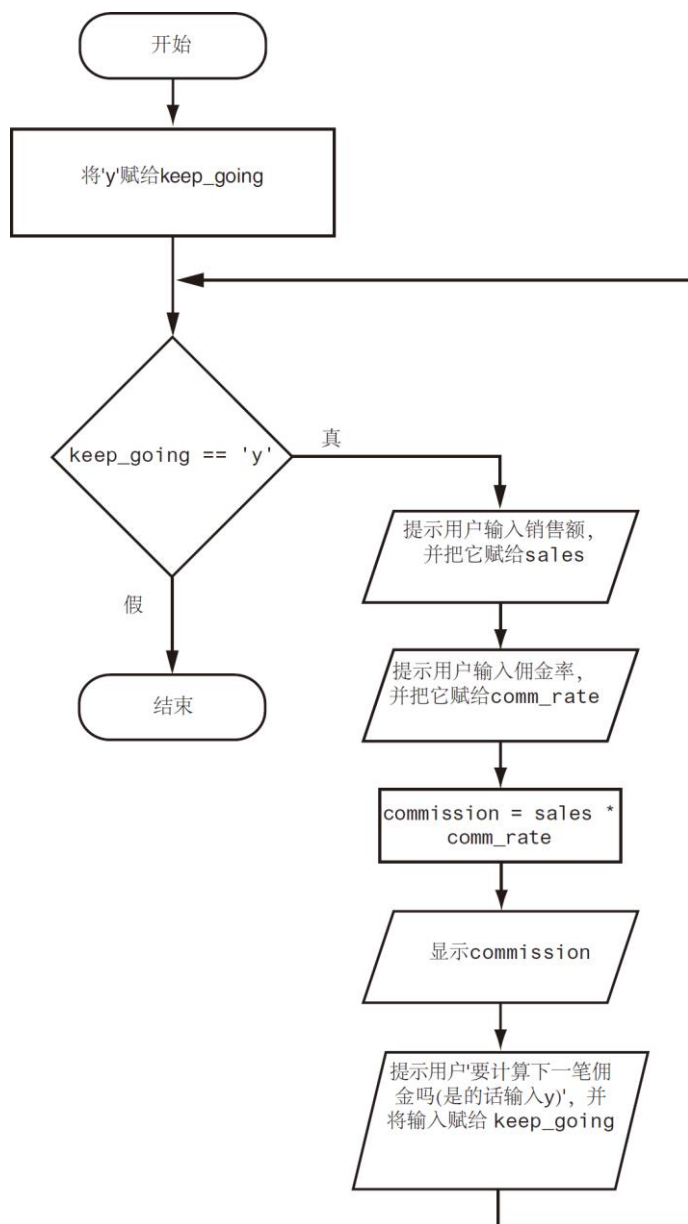


图 4-3 程序 4-1 的流程图

4.2.1 while 循环是预测试循环

while 循环是一种**预测试循环**，这意味着它在执行一次迭代之前会先测试其条件。由于测试在循环开始前完成，所以通常必须在循环之前执行一些步骤，以确保循环至少执行一次。例如，程序 4-1 的循环是这样开始的：

```
while keep_going == 'y':
```

仅当表达式 `keep_going=='y'` 为真时，循环才会执行迭代。这意味着（1）`keep_going` 变量必须存在，而且（2）它必须引用值 `'y'`。为了确保循环第一次执行时表达式为 `true`，我们在第 4 行事先将值 `'y'` 赋给 `keep_going` 变量，如下所示：

```
keep_going='y'
```

通过执行此步骤，我们知道条件 `keep_going=='y'` 在循环第一次执行时将真。这是 `while` 循环的一个重要特征：如果它的条件一开始就为假，那么它一次都不会执行。有的时候，这正是你想要的效果。下面“聚光灯”小节展示了一个例子。

聚光灯：用 `while` 循环设计程序



Chemical Labs 公司目前正在做的一个项目要求在桶中持续加热某种化学物质。技术人员必须每 15 分钟检查一次化学物质的温度。如果不超过 102.5 摄氏度，那么技术人员什么也不做。然而，如果高于 102.5 摄氏度，那么技术人员必须关闭桶的恒温器，等待 5 分钟，然后再再次检查温度。技术人员重复这些步骤，直到温度不超过 102.5 摄氏度。工程总监要求你写一个程序来指导技术人员完成此过程。

下面列出了算法。

1. 获取化学物质的温度。
2. 只要温度大于 102.5 摄氏度，就重复以下步骤：
 - a. 让技术人员关闭恒温器，等 5 分钟，然后再次检查温度。
 - b. 获取化学物质的温度。
3. 循环结束后，告诉技术人员温度可以接受，并在 15 分钟后再次检查。

检查这个算法，你意识到如果测试条件（温度大于 102.5）一开始就不成立，那么步骤 2(a) 和 2(b) 根本不会执行。`while` 循环在这种情况下能很好地工作，因为如果条件为假，它一次都不会执行。程序 4-2 展示了该程序的代码。

程序 4-2 (temperature.py)

```
1 # 该程序在检查化学物质温度
2 # 的过程中协助技术人员。
3
4 # 该具名常量代表
5 # 最大温度。
6 MAX_TEMP = 102.5
7
```

```

8 # 获取化学物质的温度
9 temperature = float(input("请输入化学物质的摄氏温度: "))
10
11 # 如有必要, 就指导用户
12 # 调节恒温器。
13 while temperature > MAX_TEMP:
14     print('温度太高。')
15     print('调低恒温器并等待')
16     print('5 分钟。重新测量, ')
17     print('并再次输入。')
18     temperature = float(input('请输入新的摄氏温度: '))
19
20 # 提醒用户在 15 分钟
21 # 后再次检查温度。
22 print('温度可以接受。')
23 print('15 分钟后再检查一次。')

```

程序输出（用户输入的内容加粗）

```

请输入化学物质的摄氏温度: 104.7 
温度太高。
调低恒温器并等待
5 分钟, 重新测量,
并再次输入。
请输入新的摄氏温度: 103.2 
温度太高。
调低恒温器并等待
5 分钟。重新测量,
并再次输入。
请输入新的摄氏温度: 102.1 
温度可以接受。
15 分钟后再检查一次。

```

程序输出（用户输入的内容加粗）

```

请输入化学物质的摄氏温度: 101.2 
温度可以接受。
15 分钟后再检查一次。

```

4.2.2 无限循环

除了极少数情况之外, 所有循环内部都必须包含一种终止循环的方法。这意味着循环内部必须有使测试条件最终变成假的机制。例如, 当表达式 `keep_going=='y'` 为假时, 程序 4-1 的循环就会停止。无法停止的循环称为**无限循环**。无限循环会一直重复, 除非程序被强行中断。当程序员忘记在循环内部编写导致测试条件为假的代码时, 就很容易发生无限循环。大多数时候都应避免写无限循环。

程序 4-3 演示了一个无限循环, 它修改了程序 4-1 的佣金计算程序, 删除了循环体内修改

`keep_going` 变量的代码。每次在第 6 行测试表达式 `keep_going=='y'` 时, `keep_going` 引用的都是字符串 'y'。因此, 会一直循环迭代下去。(退出该程序唯一的方法是按 Ctrl+C 来强行中断。)

程序 4-3 (infinite.py)

```
1 # 该程序演示了一个无限循环。
2 # 创建一个变量来控制循环
3 keep_going = 'y'
4
5 # 计算一组佣金
6 while keep_going == 'y':
7     # 获取销售人员的销售额和佣金率
8     sales = float(input('输入销售额: '))
9     comm_rate = float(input('输入佣金率: '))
10
11     # 计算佣金
12     commission = sales * comm_rate
13
14     # 显示佣金
15     print(f'佣金为${commission:,.2f}')
```

4.2.3 while 循环用作计数控制循环

本章开头说过, 计数控制循环会迭代特定的次数。虽然 `while` 循环本质上是一个条件控制循环, 但与一个计数器变量配合, 也完全可以作为一个计数控制循环来使用。**计数器变量**是每次循环迭代期都被赋予唯一值的变量。之所以称为计数器变量, 是因为通常用它们对循环迭代进行计数。

计数控制的 `while` 循环必须执行以下三个操作。

- **初始化:** 在循环开始之前, 计数器变量必须初始化为合适的起始值。
- **比较:** 循环必须将计数器变量与一个合适的结束值进行比较, 以决定循环是否应该继续下一次迭代。
- **更新:** 每次迭代期间, 循环必须使用新值更新计数器变量。

可以这样总结计数控制 `while` 循环的逻辑: 计数器变量有一个起始值和一个结束值。每次循环迭代时, 都会使用新值更新计数器变量。当计数器变量到达其结束值时, 循环停止。程序 4-4 是计数控制 `while` 循环的一个例子。

程序 4-4 (counter.py)

```
1 # 该程序演示计数控制的 while 循环
2
3 n = 0
4 while n < 5:
5     print(f'循环内 n 的值为{n}。')
6     n += 1
```

程序输出

```
循环内 n 的值为 0。
循环内 n 的值为 1。
循环内 n 的值为 2。
循环内 n 的值为 3。
循环内 n 的值为 4。
```

在这个程序中，变量 `n` 作为计数器使用。注意循环内部执行了以下操作。

- 计数器**初始化**发生在第 3 行。变量 `n` 初始化为值 `0`。
- 计数器**比较**发生在第 4 行。只要 `n` 小于 `5`，`while` 循环就会继续迭代。
- 计数器**更新**发生在第 6 行。每次循环迭代，`n` 都会递增 `1`。

换言之，`n` 从值 `0` 开始。每次循环迭代，`n` 都会递增 `1`。当 `n` 达到值 `5` 时，循环终止。

程序 4-5 是另一个例子，它有一个会迭代 10 次的计数控制 `while` 循环。

程序 4-5 (doubles.py)

```
1 # 该程序演示计数控制的 while 循环
2
3 number = 1;
4 while number <= 10:
5     print(f'{number}加{number}等于{number + number}')
6     number += 1
```

程序输出

```
1 加 1 等于 2
2 加 2 等于 4
3 加 3 等于 6
4 加 4 等于 8
5 加 5 等于 10
6 加 6 等于 12
7 加 7 等于 14
8 加 8 等于 16
9 加 9 等于 18
10 加 10 等于 20
```

在这个程序中，`number` 是计数器变量。注意循环内部执行了以下操作。

- 计数器的初始化发生在第 3 行。变量 `number` 被初始化为 1。
- 计数器的比较发生在第 4 行。只要 `number` 小于或等于 10，`while` 循环就会继续迭代。
- 计数器的更新发生在第 6 行。每次循环迭代，`number` 都会递增 1。

可以这样总结这个循环的逻辑：`number` 从值 0 开始。每次循环迭代，`number` 都会递增 1。当 `n` 达到值 11 时，循环终止。

程序 4-6 是另一个销售佣金计算程序，它使用一个计数控制的 `while` 循环为特定数量的销售人员计算佣金。

程序 4-6 (count_commission.py)

```
1  # 该程序计算销售人员的佣金
2
3  # 计数器变量
4  count = 1
5
6  # 获取销售人员数量
7  salespeople = int(input('输入销售人员数量: '))
8
9  # 计算每个销售人员的佣金
10 while count <= salespeople:
11     # 获取销售人员的销售额和佣金率
12     sales = float(input(f'输入销售人员{count}的销售额: '))
13     comm_rate = float(input('输入佣金率: '))
14
15     # 计算佣金
16     commission = sales * comm_rate
17
18     # 显示佣金
19     print(f'佣金为${commission:,.2f}')
20
21     # 更新计数器变量
22     count += 1
```

程序输出 (用户输入的内容加粗)

```
输入销售人员数量: 3 
输入销售人员 1 的销售额: 1000.00 
输入佣金率: 0.1 
佣金为$100.00
输入销售人员 2 的销售额: 2000.00 
输入佣金率: 0.15 
佣金为$300.00
输入销售人员 3 的销售额: 3000.00 
```

输入佣金率: 0.2
佣金为\$600.00

虽然计数器变量通常是递增的, 但完全可以递减。以程序 4-7 为例, 计数器变量 `count` 初始化为 10。每次循环迭代, `count` 都递减 1。当 `count` 到达值 0 时, 循环终止。

程序 4-7 (`count_down.py`)

```
1 # 这个程序显示了倒数过程
2
3 print('开始倒数。')
4
5 count = 10
6 while count > 0:
7     print(count)
8     count -= 1
9
10 print('发射!')
```

程序输出

```
10
9
8
7
6
5
4
3
2
1
发射!
```

4.2.4 单行 while 循环

如果 `while` 循环体只有一个语句, 那么 Python 允许在一行写下整个循环。常规格式如下所示:

```
while 条件: 语句
```

程序 4-8 展示了一个例子。在这个程序中, 整个 `while` 循环都在第 3 行。

程序 4-8 (`single_line_while.py`)

```
1 # 这个程序演示了单行 while 语句
2 n = 0
```

```
3 while n < 10: n += 1
4 print(f'循环终止后, n 为{n}。')
```

程序输出

循环终止后, n 为 10.

第 3 行的 `while` 循环等价于:

```
while n < 10:
    n += 1
```

写单行 `while` 循环时, 务必注意不要创建一个无限循环。循环体中的语句必须执行一个最终会造成条件变成假的行动。例如, 假定 `n` 为 0, 那么以下语句会创建一个无限循环:

```
while n < 10: print(n)
```

由于循环体内的语句不会改变 `n` 的值, 所以循环永不休止。



提示:

虽然在 Python shell 中将 `while` 循环写在一行上可以方便测试代码, 但在 Python 程序中使用这种方式并没有太多好处。在程序中, 将循环体写在它自己的缩进块中, 循环将更容易阅读和调试。

检查点

4.4 什么是循环迭代?

4.5 `while` 循环是在完成一次迭代之前还是之后测试它的条件?


4.6 以下程序会打印多少次 'Hello World'?

```
count = 10
while count < 1:
    print('Hello World')
```

4.7 什么是无限循环?

4.3 for 循环：计数控制循环

概念：计数控制循环将迭代特定的次数。在 Python 中，可以使用 `for` 语句来写计数控制的循环。

 视频讲解：The for Loop

本章开头说过，计数控制循环会迭代特定的次数。程序经常需要用到计数控制的循环。例如，假定某企业每周营业六天，你需要写程序来计算一周的总销售额。为此，需要一个恰好迭代 6 次的循环。每次循环迭代，都提示用户输入一天的销售额。

可以使用 `for` 语句来写计数控制循环。Python 的 `for` 语句设计用于处理一系列数据项。执行该语句时，它对序列中的每一项都会迭代一次。其常规格式如下所示：

```
for 变量 in [值1, 值2, ...]:  
    语句  
    语句  
    ...
```

我们将第一行称为 `for` 子句。在 `for` 子句中，“变量”是变量的名称。方括号中包含一系列值，每个值以逗号分隔。（在 Python 中，方括号中的以逗号分隔的数据项序列称为“列表”。第 7 章将更多地学习列表的知识。）从下一行开始，是每次循环迭代要执行的语句块。

`for` 语句按以下方式执行：将列表中的第一个值赋给“变量”，然后执行块中的语句。完毕后，将列表中的下一个值赋给“变量”。如此重复，将列表中的最后一个值赋给“变量”。程序 4-9 展示了一个简单的例子，它使用 `for` 循环来显示数字 1~5。

程序 4-9 (simple_loop1.py)

```
1 # 这个程序演示了使用数字列表  
2 # 的一个简单的 for 循环。  
3  
4 print('下面显示了数字 1~5。')  
5 for num in [1, 2, 3, 4, 5]:  
6     print(num)
```


程序输出

```
下面显示了数字 1~5。  
1  
2  
3  
4
```

`for` 循环第一次迭代时，`num` 变量被赋值 `1`，然后执行第 6 行的语句（显示值 `1`）。下一次循环迭代时，`num` 被赋值 `2`，并且执行第 6 行的语句（显示值 `2`）。如图 4-4 所示，此过程将重复进行，直到将列表中的最后一个值赋给 `num`。由于列表包含 5 个值，因此循环将迭代 5 次。


第1次迭代:

```
for num in [1, 2, 3, 4, 5]:  
    print(num)
```




第2次迭代:

```
for num in [1, 2, 3, 4, 5]:  
    print(num)
```




第3次迭代:

```
for num in [1, 2, 3, 4, 5]:  
    print(num)
```



第4次迭代:

```
for num in [1, 2, 3, 4, 5]:  
    print(num)
```



第5次迭代:

```
for num in [1, 2, 3, 4, 5]:  
    print(num)
```




图 4-4 for 循环

Python 程序员通常将 for 子句中使用的变量称为**目标变量**，因为它是每次循环迭代开始时

赋值的目标。

列表中的值不一定是连续数字的系列。例如，程序 4-10 使用 `for` 循环来显示奇数列表。列表中有 5 个数字，因此循环迭代 5 次。

程序 4-10 (simple_loop2.py)

```
1 # 这个程序演示了使用数字列表
2 # 的一个简单的 for 循环。
3
4 print('下面显示了 1 到 9 的奇数: ')
5 for num in [1, 3, 5, 7, 9]:
6     print(num)
```

程序输出

下面显示了 1 到 9 的奇数:

```
1
3
5
7
9
```

程序 4-11 是另一个例子。在这个程序中，`for` 循环遍历一个字符串列表。注意，列表（第 4 行）包含三个字符串：'Winken'、'Blinken'和'Nod'。结果，循环迭代了三次。

程序 4-11 (simple_loop3.py)

```
1 # 这个程序演示了使用字符串列表
2 # 的一个简单的 for 循环。
3
4 for name in ['张三丰', '张翠山', '张无忌']:
5     print(name)
```

程序输出

```
张三丰
张翠山
张无忌
```

4.3.1 为 for 循环使用 range 函数

Python 提供了一个名为 `range` 的内置函数，可以用它简化计数控制 `for` 循环的编写。`range` 函数创建一个**可迭代对象**。可迭代对象（iterable）是类似于列表的对象。它包含一系列值，可以使用循环之类的方式进行迭代。以下是使用了 `range` 函数的一个示例 `for` 循环。

```
for num in range(5):
    print(num)
```

注意，这里没有使用值列表，而是直接调用 `range` 函数，传递 5 作为实参。在这具语句中，`range` 函数将生成一个可迭代的整数序列，范围为 0~（但不包括）5。此代码的工作方式相当于：

```
for num in [0, 1, 2, 3, 4]:
    print(num)
```

如你所见，列表包含 5 个数字，因此循环将迭代 4 次。程序 4-12 使用 `range` 函数和 `for` 循环来显示 'Hello world' 五次。

程序 4-12 (simple_loop4.py)

```
1 # 这个程序演示了如何将 range 函数
2 # 用于 for 循环
3
4 # 打印一条消息五次
5 for x in range(5):
6     print('Hello world!')
```

程序输出

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

将一个实参传给 `range` 函数（如程序 4-12 所示），该实参将用作数字序列的结束限制。将两个实参传给 `range` 函数，那么第一个将用作序列的起始值，第二个则将用作结束限制。下面是一个例子。

```
for num in range(1, 5):
    print(num)
```

上述代码的结果如下所示。

```
1
2
3
4
```

在 `range` 函数生成数字序列中，每个连续的数字默认递增 `1`。但是，将第三个实参传给 `range` 函数，它就会作为步长值使用。序列中的每个连续的数字都将递增指定的步长值，而不是递增 `1`。下面是一个例子。

```
for num in range(1, 10, 2):
    print(num)
```

这个 `for` 循环向 `range` 函数传递了三个实参：

- 第一个实参 `1` 是序列的起始值。
- 第二个实参 `10` 是列表的结束限制。这意味着序列中的最后一个数字小于等于 `9`。
- 第三个实参 `2` 是步长值。这意味着序列中的每个连续的数字都会递增 `2`。

上述代码的结果如下所示。

```
1
3
5
7
9
```

4.3.2 在循环内使用目标变量

在 `for` 循环中，目标变量的作用是在循环迭代时引用数据项序列中的每一项。许多时候都需要在循环体内的计算或其他任务中使用目标变量。例如，假设需要编写一个程序，以如下所示的表格形式显示数字 `1~10` 以及每个数字的平方。

数字	平方
1	1
2	4
3	9
4	16
5	25
6	36
7	49

8	64
9	81
10	100

这可以通过编写一个遍历 1~10 的 for 循环来实现。第一次迭代时，目标变量被赋值 1，第二次迭代赋值 2，以此类推。由于目标变量在循环执行期间将引用值 1~10，所以可以在循环内的计算中使用它。程序 4-13 展示了完整程序。

程序 4-13 (squares.py)

```
1 # 这个程序使用循环，
2 # 以表格形式显示数字
3 # 1 到 10 及其平方。
4
5 # 打印表格的列标题
6 print('数字\t平方')
7 print('-----')
8
9 # 打印数字 1 到 10
10 # 及其平方。
11 for number in range(1, 11):
12     square = number**2
13     print(f'{number}\t{square}')
```

程序输出

```
数字    平方
-----
1        1
2        4
3        9
4       16
5       25
6       36
7       49
8       64
9       81
10      100
```

第 6 行显示了表格的列标题：

```
print('数字\t平方')
```

注意，在字符串面值中，“数字”和“平方”之间使用了一个\t 转义序列。第 2 章讲过，\t 转义序列代表制表符。它相当于按 Tab 键，使输出光标移动到下一个制表位。如示例输出所示，这会导致“数字”和“平方”之间出现空白间距。

从第 11 行开始的 for 循环使用 range 函数来生成包含数字 1~10 的序列。第一次迭代，number 将引用 1，第二次迭代，number 将引用 2，以此类推，直到最后引用 10。在循环内部，第 12 行的语句计算 number 的平方（第 2 章讲过，**是求幂操作符）并将结果赋给 square 变量。第 13 行的语句打印 number 引用的值，后跟一个制表符，然后打印 square 引用的值。由于使用了\t 转义序列添加了制表位，会导致每一行输出的数字在两列中对齐。

图 4-5 展示了这个程序的流程图。

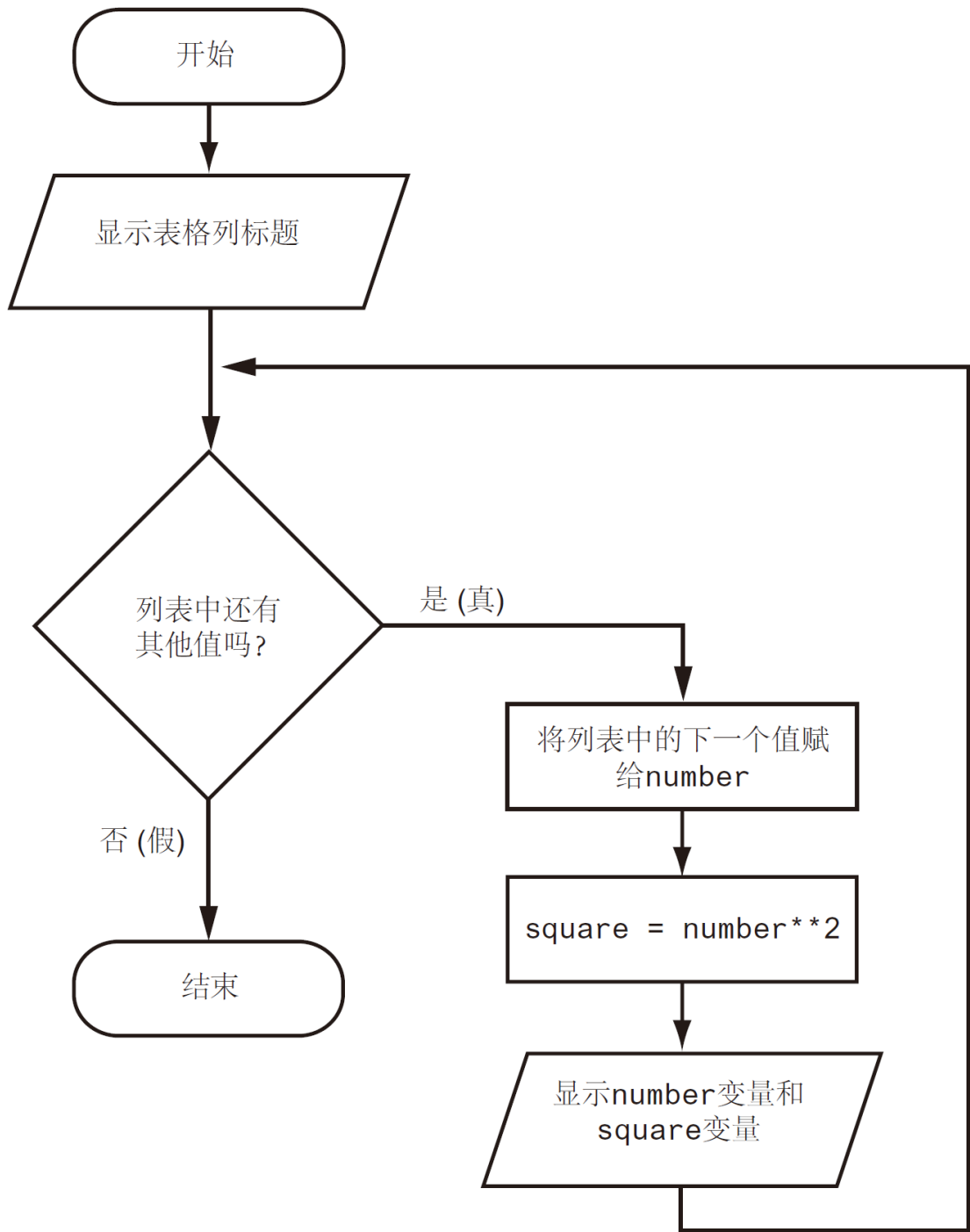


图 4-5 程序 4-8 的流程图

聚光灯：用 for 循环来设计一个计数控制循环



你的朋友阿曼达刚刚从她叔叔那里继承了一辆欧洲生产的跑车。阿曼达居住在美国，她担心自己会收到超速罚单，因为车速表显示的是公里每小时（KPH）。她要求你写一个程序，显示 KPH 速度和 MPH（英里每小时）速度的对应关系。KPH 到 MPH 的换算公式是：

$$MPH = KPH * 0.6214$$

在这个公式中，MPH 是英里/小时速度，KPH 是公里/小时速度。

程序应输出一个表格，显示 60 KPH~130 KPH 的速度（以 10 为增量），以及换算为 MPH 后的结果，如下所示。

KPH	MPH
60	37.3
70	43.5
80	49.7
...	
130	80.8

基于这个表格，你决定写一个 for 循环来遍历包含所有 KPH 值的一个序列，如下所示：

```
range(60, 131, 10)
```

序列中的第一个值是 60。注意，第三个实参指定 10 作为步长。这意味着序列中的数字将为 60, 70, 80 等。第二个参数指定 131 作为序列的结束限制，因此序列中的最后一个数字是 130。

在循环内，将使用目标变量来计算 MPH 速度。程序 4-14 展示了完整程序。

程序 4-14 (speed_converter.py)

```
# 这个程序将从 60 kph 到  
# 130 kph（以 10 kph 的增量）  
# 的速度换算为 mph。
```

```

START_SPEED = 60 #开始速度
END_SPEED = 131 #结束速度
INCREMENT = 10 #速度增量
CONVERSION_FACTOR = 0.6214 #速度系数

#打印表格的列标题
print('KPH\tMPH')
print('-----')

# 打印速度
for kph in range(START_SPEED, END_SPEED, INCREMENT):
    mph = kph * CONVERSION_FACTOR
    print(f'{kph}\t{mph:.1f}')

```

程序输出

KPH	MPH
60	37.3
70	43.5
80	49.7
90	55.9
100	62.1
110	68.4
120	74.6
130	80.8

4.3.3 让用户控制循环迭代

许多时候，程序员事先知道循环会迭代多少次。以程序 4-13 为例，它显示一个表格，其中列出了数字 1~10 及其平方。因此，程序员在写代码时就知道循环必须从值 1 遍历到值 10。

有的时候，程序员需要让用户控制循环迭代的次数。例如，如果希望程序 4-13 变得更通用，允许用户指定循环计算的最大值，那么该怎么办？程序 4-15 展示了具体如何做。

程序 4-15 (user_squares1.py)

```

1 # 这个程序使用循环来显示
2 # 数字及其平方的一个表格。
3
4 # 用户输入要打印平方结果的最大数字
5 print('该程序以表格形式显示(从 1 开始)')
6 print('的一个数字列表及其平方。')
7 end = int(input('最大打印哪个数字的平方? '))
8
9 #打印表格的列标题
10 print()

```

```
11 print('数字\t平方')
12 print('-----')
13
14 # 打印数字及其平方
15 for number in range(1, end + 1):
16     square = number**2
17     print(f'{number}\t{square}')
```

程序输出（用户输入的内容加粗）

程序以表格形式显示(从 1 开始)
的一个数字列表及其平方。
最大打印哪个数字的平方? **5**

```
数字    平方
-----
1        1
2        4
3        9
4       16
5       25
```

该程序要求用户输入要打印平方结果的最大数字。第 7 行将该值赋给 `end` 变量。然后，第 15 行将表达式 `end + 1` 用作 `range` 函数的第二个参数。注意，加 1 之后才是序列的“结束限制”（`end limit`），对序列的遍历会在这个限制之前结束，不会包括这个限制值。

程序 4-16 是一个允许用户同时指定序列起始值和结束限制的例子。

程序 4-16 (user_squares2.py)

```
1 # 这个程序使用循环来显示
2 # 数字及其平方的一个表格。
3
4 # 用户输入要打印平方结果的最小数字
5 print('该程序以表格形式显示')
6 print('一个数字列表及其平方。')
7 start = int(input('最小打印哪个数字的平方? '))
8
9 # 用户输入要打印平方结果的最大数字
10 end = int(input('最大打印哪个数字的平方? '))
11
12 #打印表格的列标题
13 print()
14 print('数字\t平方')
15 print('-----')
```

```
16
17 # 打印数字及其平方
18 for number in range(start, end + 1):
19     square = number**2
20     print(f'{number}\t{square}')
```

程序输出（用户输入的内容加粗）

该程序以表格形式显示
一个数字列表及其平方。

最小打印哪个数字的平方？ **5**
最大打印哪个数字的平方？ **10**

数字	平方
5	25
6	36
7	49
8	64
9	81
10	100

4.3.4 生成降序可迭代序列

之前的例子是用 `range` 函数生成从小到大的数字序列。但是，完全可以用它来生成从大到小的数字序列。例如：

```
range(10, 0, -1)
```

在这个函数调用中，起始值为 `10`，序列的结束限制为 `0`，步长值为 `-1`。该表达式将生成以下降序序列：

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

以下 `for` 循环降序打印数字 5 到 1。

```
for num in range(5, 0, -1):
    print(num)
```

检查点

4.8 重写以下代码，调用 `range` 函数而不是使用列表、

```
[0, 1, 2, 3, 4, 5]:
for x in [0, 1, 2, 3, 4, 5]:
    print('我爱编程!')
```

4.9 以下代码会显示什么？

```
for number in range(6):
```

```
print(number)
```

4.10 以下代码会显示什么？

```
for number in range(2, 6):  
    print(number)
```

4.11 以下代码会显示什么？

```
for number in range(0, 501, 100):  
    print(number)
```

4.12 以下代码会显示什么？

```
for number in range(10, 5, -1):  
    print(number)
```

4.4 计算累加和

概念：累加和（**running total**）是通过多次循环迭代来累积的一系列数字之和。用于保存累加和的变量称为累加器（**accumulator**）。

许多编程任务要求计算一系列数字之和。例如，假设要写程序来计算企业一周的总销售额。程序将读取每天的销售额作为输入，并计算这些数字的总和。

计算一系列数字总和的程序通常要使用两个元素：

- 读取序列中每个数字的循环
- 用于保存累加和的一个变量

用于累加数字之和的变量称为**累加器**。我们经常说循环会保留一个**累加和**，因为它会随着序列中每个数字的读取来累加其总和。图 4-6 展示了计算累加和的循环的常规逻辑。

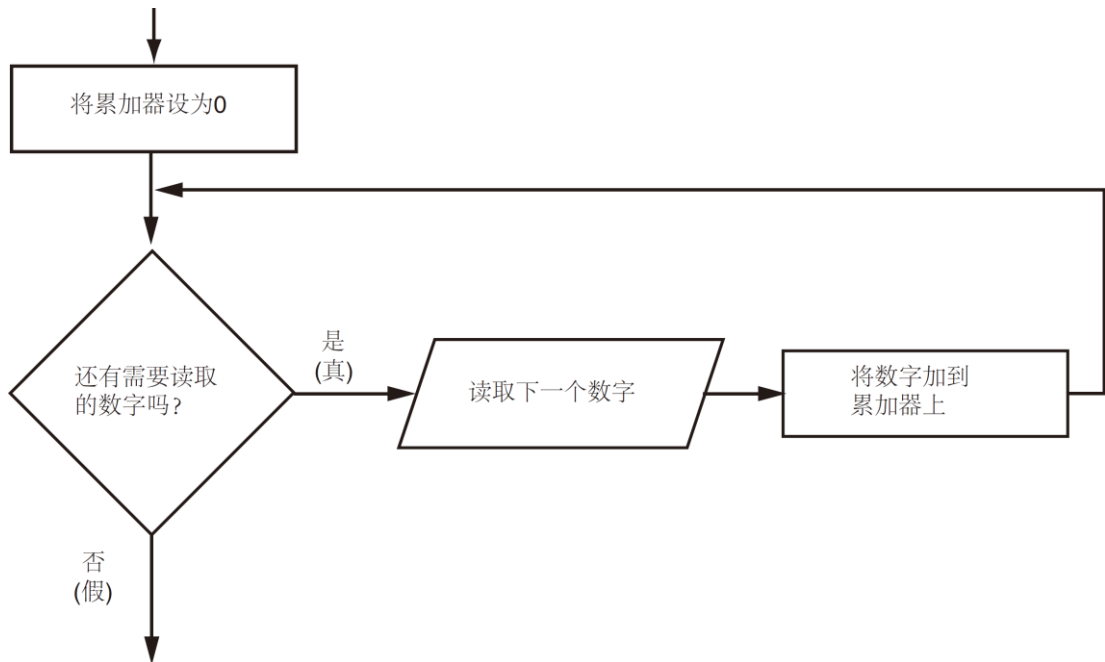


图 4-6 计算累加和的逻辑

当循环结束时，累加器将包含循环读取的所有数字之和。注意，流程图中的第一步是将累加器变量设为 0。这是很关键的一步。循环每次读取一个数字时，都会将其加到累加器变量上。如果累加器以 0 以外的其他任何值开始，那么循环结束时它包含的就不是正确的总和。

现在来看一个计算累加和的程序。程序 4-17 允许用户输入 5 个数字，并显示输入的所有数字之和。

程序 4-17 (sum_numbers.py)

```

1  # 这个程序计算用户输入的
2  # 一系列数字之和。
3
4  MAX = 5 # 最多能输入多少个数
5
6  # 初始化累加器变量
7  total = 0.0
8
9  # 程序说明
10 print('本程序计算你输入的', end='')
11 print(f'{MAX}个数字的总和。')
12
13 # 获取每个数字，并累加它们
  
```

```
14 for counter in range(MAX):
15     number = int(input('输入一个数字: '))
16     total = total + number
17
18 # 显示所有数字之和
19 print(f'总和为{total}。')
```

程序输出（用户输入的内容加粗）

本程序计算你输入的 5 个数字的总和。

输入一个数字: **1**

输入一个数字: **2**

输入一个数字: **3**

输入一个数字: **4**

输入一个数字: **5**

总和为 15.0。

第 7 行的赋值语句创建的 `total` 变量就是累加器，注意它初始化为 `0.0`。第 14 行~第 16 行的 `for` 循环从用户处获取数字并计算它们的总和。第 15 行提示用户输入数字，并将输入赋给 `number` 变量。然后，第 16 行的以下语句将 `number` 加到 `total` 上：

```
total = total + number
```

该语句将 `number` 的值加到 `total` 上。理解这个语句的工作方式非常重要。首先，解释器对 `=` 操作符右侧的表达式（`total + number`）进行求值。然后，返回的求值结果由 `=` 操作符赋给 `total` 变量。整个语句的作用就是计算 `number` 变量的值与 `total` 变量现有值之和，并将新值重新赋给 `total` 变量。整个循环结束后，`total` 变量存储的就是历次累加的总和。这个结果在第 19 行显示。

复合赋值操作符

我们经常需要写 `=` 操作符左侧的变量同时出现在右侧的赋值语句，例如：

```
x = x + 1
```

赋值操作符右侧计算 `x` 加 1，并将求值结果赋给 `x`，替换 `x` 之前引用的值。该语句的实现了 `x` 递增 1 的效果。程序 4-18 展示了这种语句的另一个例子：

```
total = total + number
```

该语句将 `total + number` 的求值结果赋给 `total`。如前所述，该语句的作用是将 `number` 加到 `total` 上。下面是另一个例子：

```
balance = balance - withdrawal
```

该语句将表达式 `balance - withdrawal` 赋给 `balance`，作用是从 `balance` 中减去

withdrawal。表 4-1 展示了以这种方式写的其他示例语句。

表 4-2 各种赋值语句（假设每个语句的 x 初始值都是 6）

语句	含义	执行完毕后的 x 值
<code>x = x + 4</code>	在 x 上加 4	10
<code>x = x - 3</code>	从 x 减 3	3
<code>x = x * 10</code>	x 乘以 10	60
<code>x = x / 2</code>	x 除以 2	3
<code>x = x % 4</code>	将 x / 4 的余数赋给 x	2

这些类型的操作在编程中很常见。为了简化编程，Python 提供了一组专门执行这些操作的特殊操作符，称为**复合赋值操作符**^①，如表 4-2 所示。

表 4-2 复合赋值操作符

操作符	示例用法	等价于
<code>+=</code>	<code>x += 5</code>	<code>x = x + 5</code>
<code>--</code>	<code>y -= 2</code>	<code>y = y - 2</code>
<code>*=</code>	<code>z *= 10</code>	<code>z = z * 10</code>
<code>/=</code>	<code>a /= b</code>	<code>a = a / b</code>
<code>%=</code>	<code>c %= 3</code>	<code>c = c % 3</code>
<code>//=</code>	<code>x //= 3</code>	<code>x = x // 3</code>
<code>**=</code>	<code>y **= 2</code>	<code>y = y**2</code>

^① 译注：Python 用 augmented assignment operators（增强赋值操作符）一词来称呼这种操作符。但一般都说成复合赋值操作符。

如你所见，复合赋值操作符不需要程序员输入变量名两次。例如，以下语句：

```
total = total + number
```

可以重写为：

```
total += number
```

类似地，以下语句：

```
balance = balance - withdrawal
```

可以重写为：

```
balance -= withdrawal
```

检查点

4.3 什么是累加器？

4.14 累加器是否需要初始化为任何特定值？为什么需要或者为什么不需要？

4.15 以下代码会显示什么？

```
total = 0
for count in range(1, 6):
    total = total + count
print(total)
```

4.16 以下代码会显示什么？

```
number1 = 10
number2 = 5
number1 = number1 + number2
print(number1)
print(number2)
```

4.17 使用复合赋值操作符来重写以下代码：

```
a) quantity = quantity + 1
b) days_left = days_left - 5
c) price = price * 10
d) price = price / 2
```

4.5 哨兵

概念：哨兵是标志值序列结尾一个特殊值。

假定设计一个程序，用循环来处理一长串值。设计程序时，并不知道序列中的值的数量。事实上，每次执行程序时序列中的值的数量都可能不同。为了设计这样一个循环，最佳的方法是什么？以下是本章已经讲过的一些技术，以及它们在处理一长串值时的缺点：

- 每次循环迭代结束，都询问用户是否还有另一个值需要处理。但是，如果值序列很长，那么在每次循环迭代结束时都问这个问题，可能造成用户的厌烦。
- 在程序开始时询问用户总共要处理多少个值。但是，这也可能给用户带来不便。如果要处理的值很多，而且用户不知道具体有多少个，那么还要麻烦用户先统计好。

使用循环来处理一长串值时，更好的技术是使用**哨兵**。哨兵是标记了值序列结尾的一个特殊值。一旦程序读取到哨兵值，就知道已经到达序列的末尾，因此循环终止。

例如，假设医生想要用一个程序来计算所有患者的平均体重。程序可这样工作：在循环中提示输入体重，如果没有更多的体重，就输入 0。当程序读入的体重为 0 时，就将其视为没有更多体重需要输入的信号。整个循环结束，程序显示平均体重。

哨兵值必须足够特殊，以免被误认为是序列中的常规值。在前面的例子中，医生输入 0 来表示结束体重序列。因为没有患者的体重会是 0，所以这是一个很好的哨兵值。

聚光灯：使用哨兵



某税务局使用以下公式来计算每年应收取的房产税：

$$\text{房产税} = \text{房产估值} \times 0.0065$$

税务局职员每天都会收到一份房产清单，而且必须计算清单上每处房产的税款。现在，你需要设计一个程序，使职员可以使用该程序来完成这些计算。

通过与职员面谈，你了解到每处房产都分配有一个土地编号（lot number），而且所有土地编号均为 1 或更大。你决定写使用数字 0 作为哨兵值的一个循环。每次循环迭代，程序都要求职员输入土地编号，或者输入 0 来结束。完整代码如程序 4-18 所示。

程序 4-18 (property_tax.py)

```
1 # 这个程序显示应收房产税
2
3 TAX_FACTOR = 0.0065 # 税率
4
5 # 获取第一个土地编号 (lot number)
6 print('输入土地编号，或输入 0 结束。')
```

```
7 lot = int(input('土地编号: '))
8
9 # 只要用户不输入 0,
10 # 就一直处理。1
11 while lot != 0:
12     # 获取房产估值
13     value = float(input('输入房产估值: '))
14
15     # 计算房产税
16     tax = value * TAX_FACTOR
17
18     # 显示税款
19     print(f'房产税: ${tax:,.2f}')
20
21     # 获取下一个土地编号
22     print('输入下一个土地编号, 或输入 0 结束。')
23     lot = int(input('土地编号: '))
```

程序输出 (用户输入的内容加粗)

```
输入土地编号, 或输入 0 结束。
土地编号: 100 
输入房产估值: 100000.0 
房产税: $650.00
输入下一个土地编号, 或输入 0 结束。
土地编号: 200 
输入房产估值: 5000.0 
房产税: $32.50
输入下一个土地编号, 或输入 0 结束。
土地编号: 0 
```

检查点

4.18 什么是哨兵?

4.19 为什么要选择一个足够特殊的值作为哨兵?

4.6 输入校验循环

概念: 输入校验对输入程序的数据进行检查, 以确保将合法的值用于实际的计算。可以用一个循环来进行输入校验, 只要输入变量引用了非法数据, 就一直迭代。

在程序员中流传甚广的一个谚语是“垃圾进, 垃圾出”(Garbage In, Garbage Out)。这句话有时缩写为 GIGO, 指的是计算机本身无法区分好数据和坏数据。如果用户提供错误的数
据作为输入, 那么程序会老老实实在地处理错误的输入, 并因此产生错误的输出。以程序 4-

19 的工资单程序为例，注意当用户提供错误的的数据作为输入时，示例程序输出中发生了什么。

程序 4-19 (gross_pay.py)

```
1 # 这个程序显示总工资
2 # 获取工作时数
3 hours = int(input('输入本周工作时数: '))
4
5 # 获取时薪
6 pay_rate = float(input('输入时薪: '))
7
8 # 计算总工资
9 gross_pay = hours * pay_rate
10
11 # 显示总工资。
12 print(f'总工资: ${gross_pay:,.2f}')
```

程序输出 (用户输入的内容加粗)

```
输入本周工作时数: 400 
输入时薪: 20 
总工资: $8,000.00
```

看出问题了吗？收到薪水的人会感到惊喜，因为在示例运行中，负责薪资的职员输入 **400** 作为工作时数。职员可能想输入的是 **40**，一周根本没有 400 小时。然而，计算机并没有意识到这一事实，程序会像处理好数据一样处理坏数据。你能想到其他会导致错误输出的输入吗？一个例子是为工作时数输入负数；另一个输入无效时薪。

有的时候，新闻报道中会出现一些有关计算机错误的故事，这些错误导致人们因小额购物而被收取数千美元的费用，或者获得他们无权获得的大额退税。然而，这些“计算机错误”很少是由计算机引起的；更常见的是因为输入了错误的的数据。

除非有正确的输入，否则程序输出的正确性无法保证。因此，在设计程序的时候，应确保不要接受错误的输入。向程序提供的输入应在使用前进行检查。如果输入无效，程序应丢弃它并提示用户输入正确的数据。这个过程称为**输入校验**。

图 4-7 展示了对输入数据进行校验的一个常用技术。它会读取输入并执行一个循环。如果输入数据有误，那么会执行循环体，显示一条错误消息，使用户知道输入无效，然后读取新的输入。只要输入错误，循环就会一直迭代。

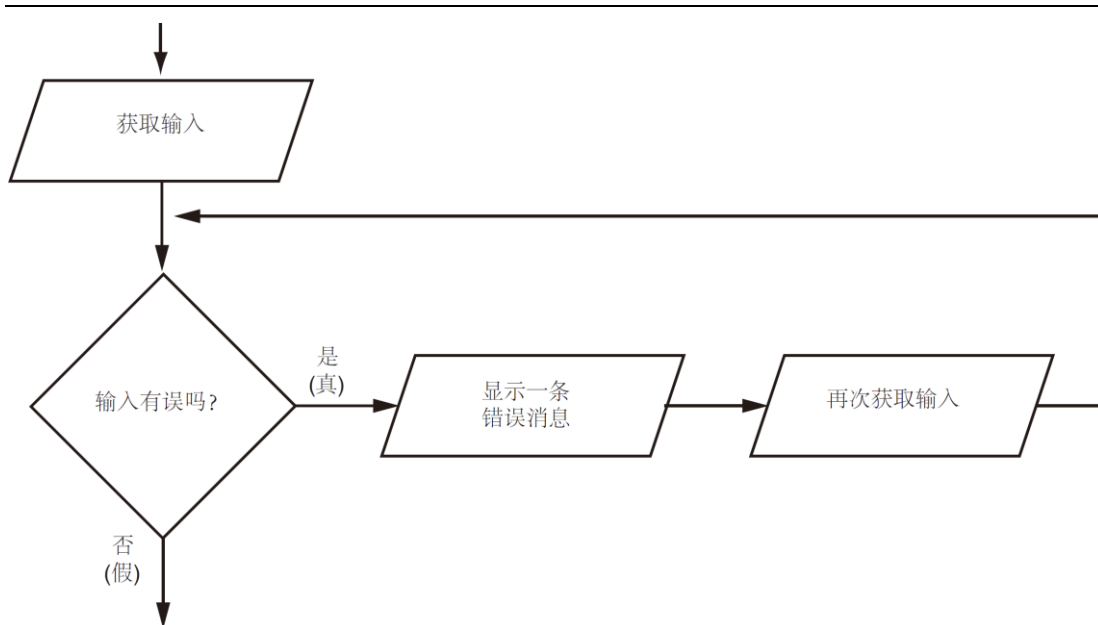


图 4-7 包含输入校验循环的逻辑


注意，图 4-7 的流程图在两个位置读取输入：进入循环之前，以及循环内部。第一个输入操作（在循环之前）称为**预读 (priming read)**，其目的是获取将由校验循环测试的第一个输入值。如果该值无效，那么将由循环接手来执行后续输入操作。除非提供有效的输入，否则循环不会终止。

例如，假设要设计一个读取考试分数的程序，并且希望确保用户不会输入小于 0 的值。以下代码展示了如何使用输入校验循环来拒绝任何小于 0 的输入值。

```
# 获取考试分数
score = int(input('输入考试分数: '))

# 确保它不小于 0
while score < 0:
    print('错误: 分数不能为负。')
    score = int(input('输入正确分数: '))
```

代码首先提示用户输入考试分数（预读），然后执行 `while` 循环。以前说过，`while` 循环是一种预测试循环，这意味着它在执行一次迭代之前会先测试条件表达式 `score < 0`。如果用户输入了有效的考试分数，那么该表达式将为假，不会执行迭代，并且循环结束。但是，如果考试分数无效，那么表达式将为真，将执行作为循环体的语句块，显示错误消息并提示用户输入正确的考试分数。该循环将一直迭代，直到用户输入有效的考试分数。

 注意：输入校验循环有时也称为错误陷阱或错误处理程序（`error handler`）。

上述代码只是拒绝负的考试分数。如果还想拒绝任何大于 100 的考试分数怎么办？可以修改输入校验循环来使用一具复合布尔表达式，如下所示。

```
# 获取考试分数
score = int(input('输入考试分数: '))

# 确保它不小于 0 或大于 100
while score < 0 or score >100:
    print('错误: 分数不能为负。')
    score = int(input('输入正确分数: '))
```

上述代码中的循环判断 `score` 是否小于 0 或大于 100。任何条件为真，就显示错误消息并提示用户输入正确的分数。

聚光灯：编写输入校验循环



萨曼莎拥有一家进口企业，她用以下公式计算产品的零售价：

$$\text{零售价} = \text{批发价} \times 2.5$$

她目前使用程序 4-15 来计算零售价。

程序 4-15 (retail_no_validation.py)

```
1  # 这个程序计算零售价
2  MARK_UP = 2.5 # 加价倍数
3  another = 'y' # 循环控制变量
4
5  # 处理一件或多件商品
6  while another == 'y' or another == 'Y':
7      # 获取商品批发价
8      wholesale = float(input("输入商品的批发价: "))
9
10     # 计算零售价
11     retail = wholesale * MARK_UP
12
13     # 显示零售价
14     print(f'零售价是: ${retail:,.2f}')
15
16     # 继续吗?
17     another = input('是否需要处理其他商品' +
18                     '(如果是, 请输入 y): ')
```

程序输出 (用户输入的内容加粗)

```
输入商品的批发价: 10 
零售价是: $25.00
是否需要处理其他商品(如果是, 请输入 y): y 
输入商品的批发价: 15.00 
零售价是: $37.50
是否需要处理其他商品(如果是, 请输入 y): y 
输入商品的批发价: 12.50 
零售价是: $31.25
是否需要处理其他商品(如果是, 请输入 y): n 
```

然而, 萨曼莎在使用该程序时遇到了问题。她销售的某些商品的批发价为 50 美分, 作为 0.50 输入。由于 0 键位于减号键旁边, 所以有时会不小心输入负数。她要求你修改程序, 不允许为批发价输入负数。

你决定添加一个输入校验循环, 该循环拒绝输入到 `wholesale` 变量中的任何负数。程序 4-21 展示了修改后的程序, 第 11 行~13 行是新的输入校验代码。

程序 4-21 (`retail_with_validation.py`)

```
1  # 这个程序计算零售价
2  MARK_UP = 2.5 # 加价倍数
3  another = 'y' # 循环控制变量
4
5  # 处理一件或多件商品
6  while another == 'y' or another == 'Y':
7      # 获取商品批发价
8      wholesale = float(input("输入商品的批发价: "))
9
10     # 校验批发价
11     while wholesale < 0:
12         print('错误: 价格不能为负。')
13         wholesale = float(input('输入正确的' +
14                                 '批发价: '))
15
16     # 计算零售价
17     retail = wholesale * MARK_UP
18
19     # 显示零售价
20     print(f'零售价是: ${retail:,.2f}')
21
22     # 继续吗?
23     another = input('是否需要处理其他商品' +
24                     '(如果是, 请输入 y): ')
```

程序输出 (用户输入的内容加粗)

```
输入商品的批发价: -.50 
```

错误：价格不能为负。

输入正确的批发价：0.50

零售价是：\$1.25

是否需要处理其他商品(如果是，请输入 y)：n

在输入校验循环中使用海象操作符

第 3 章介绍了如何用海象操作符 (`:=`) 来写赋值表达式。当时说过，赋值表达式将值赋给变量并返回该值。可以在一个更大的语句中使用由赋值表达式返回的值。

在输入校验循环中，可以使用赋值表达式将预读与输入校验循环结合起来。例如，假设有以下代码。

```
score = int(input('输入分数: '))
while score < 0:
    print('分数不能为负。')
    score = int(input('输入分数: '))
```

上述代码首先执行一次预读，提示用户输入分数。然后，只要 `score` 小于 0，就执行 `while` 循环。在校验循环内，系统通过第二个 `input` 语句提示用户输入分数。

使用海象操作符，可以将预读与 `while` 循环的布尔条件结合起来，从而使代码变得更简洁。

```
while (score := int(input('输入分数: '))) < 0:
    print('分数不能为负。')
```

`while` 循环的第一行使用以下赋值表达式从用户处获取分数：

```
(score:=int(input('输入分数: ')))
```

该海象表达式将返回赋给 `score` 变量的值。如果这个值小于 0，那么循环将迭代。注意，循环体内的第二个 `input` 语句也可以删除，因为循环每次测试其布尔条件时，都会提示用户输入分数。

检查点

- 4.20 “垃圾进，垃圾出”这句话是什么意思？
- 4.21 给出输入校验过程的一般性描述。
- 4.22 描述使用输入校验循环对数据进行校验时通常采取的步骤。
- 4.23 什么是“预读”（priming read）？它的目的是什么？
- 4.24 如果预读的输入有效，那么输入校验循环将迭代多少次？

4.7 嵌套循环

概念：一个循环位于另一个循环的内部，就称为嵌套循环。

一个循环位于另一个循环的内部，就称为**嵌套循环**。现在流行的石英钟就是嵌套循环的一个很好的例子。秒针、分针和时针都绕着钟面旋转。然而，分针每走完一圈 60 格（60 分钟），时针才跳 1 格（1 小时）。秒针每走完一圈 60 格（60 秒），分针才跳 1 格（1 分钟）。这意味着时针每跳 1 格，秒针都要跳 3600 格。下面是一个部分模拟了 24 小时格式的数字时钟的循环，它遍历并显示 0~59 秒的每一秒：

```
for seconds in range(60):
    print(seconds)
```

可以添加一个 `minutes` 变量来代表分钟，并将上述循环嵌套在另一个循环中，后者遍历 60 分钟的每一分钟：

```
for minutes in range(60):
    for seconds in range(60):
        print(minutes, ':', seconds)
```

为了使模拟的数字时钟变得完整，可以再添加一个变量和循环来遍历 24 小时的每一小时：

```
for hours in range(24):
    for minutes in range(60):
        for seconds in range(60):
            print(hours, ':', minutes, ':', seconds)
```

上述代码的输出是：

```
0 : 0 : 0
0 : 0 : 1
0 : 0 : 2
... (程序将遍历并显示 24 小时中的每一秒)
23:59:59
```

在上述三个循环构成的嵌套循环中，中间循环的每一次迭代，最内层循环都会迭代 60 次。最外层循环的每一次迭代，中间循环都会迭代 60 次。当最外层循环迭代了 24 次时，中间循环将迭代 1440 次，而最内层循环将迭代 86400 次！图 4-8 展示完整时钟模拟程序的流程图。

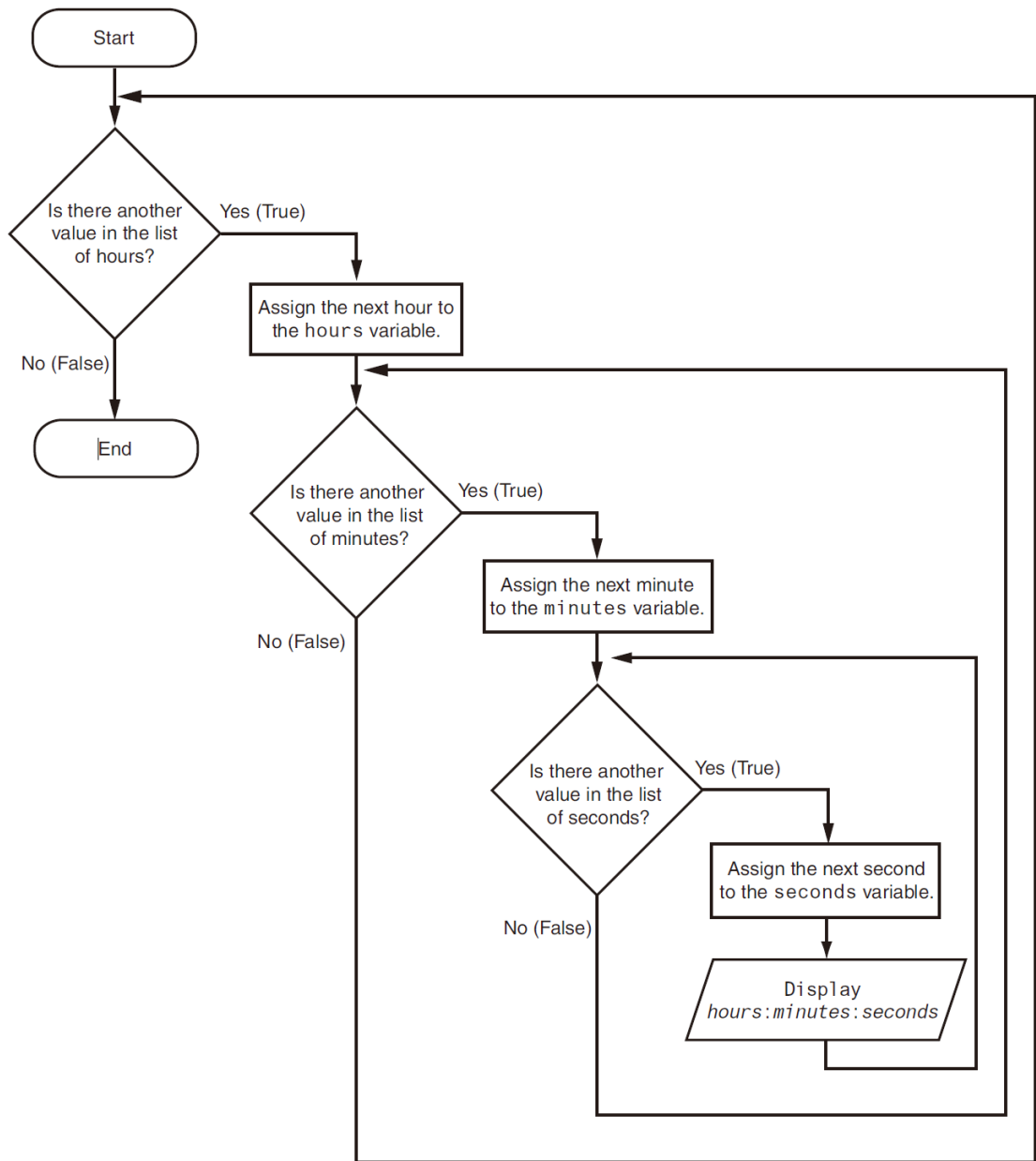


图 4-8 时钟模拟程序流程图

图中译文：

Start: 开始

End: 结束

Is there another value in the list of hours?: 小时列表还剩余其他值吗?

Yes (True): 是(真)

No (False): 否(假)

Assign the next hour to the hours variable.: 将下一个小时赋给 `hours` 变量

Is there another value in the list of minutes?: 分钟列表还剩余其他值吗?

Assign the next minute to the minutes variable.: 将下一个小时赋给 `minutes` 变量

Is there another value in the list of seconds?: 秒列表还剩余其他值吗?

Assign the next second to the seconds variable.: 将下一个小时赋给 `seconds` 变量

Display: 显示

模拟时钟例子说明了关于嵌套循环的几个重点:

- 外部循环每一次迭代, 内部循环都会经历其所有迭代。
- 内部循环比外部循环更快完成迭代。
- 要获得嵌套循环的迭代总数, 将所有循环的迭代次数相乘即可。

程序 4-22 展示了另一个例子。可以用它来计算每个学生的平均考试分数。第 5 行的语句询问学生的数量, 第 8 行的语句询问每个学生有多少个考试分数。从第 11 行开始的 `for` 循环为每个学生迭代一次。第 20 行~第 25 行嵌套的内部循环则为每个考试分数迭代一次。

程序 4-22 (`test_score_averages.py`)

```
1 # 这个程序计算平均考试分数。它要求用户指定
2 # 学生人数以及每个学生有多少个考试分数。
3
4 # 获取学生人数
5 num_students = int(input('有多少名学生? '))
6
7 #获取每个学生有多少个考试分数
8 num_test_scores = int(input('每个学生有多少个考试分数? '))
9
10 # 计算每个学生的平均分
11 for student in range(num_students):
12     # 初始化考试分数累加器
13     total = 0.0
14
```

```

15     #显示学生编号
16     print(f'学生编号{student + 1}')
17     print('-----')
18
19     # 获取学生的考试分数
20     for test_num in range(num_test_scores):
21         print(f'第{test_num + 1}门考试', end='')
22         score = float(input(': '))
23
24         # 累加分数
25         total += score
26
27     # 计算这名学生的平均考试分数
28     average = total / num_test_scores
29
30     # 显示平均分数
31     print(f'编号为{student + 1}的学生的平均分'
32           f'为: {average:.1f}')
33     print()

```

程序输出（用户输入的内容加粗）

```

有多少名学生? 3 
每个学生有多少个考试分数? 3 
学生编号 1
-----
第 1 门考试: 100 
第 2 门考试: 95 
第 3 门考试: 90 
编号为 1 的学生的平均分为: 95.0

学生编号 2
-----
第 1 门考试: 80 
第 2 门考试: 81 
第 3 门考试: 82 
编号为 2 的学生的平均分为: 81.0

学生编号 3
-----
第 1 门考试: 75 
第 2 门考试: 85 
第 3 门考试: 80 
编号为 3 的学生的平均分为: 80.0

```

聚光灯：使用嵌套循环打印图案

为了理解嵌套循环，一个有趣的方式是实验用它们在屏幕上打印图案。来看一个简单的例

子。假设要在屏幕上打印由星号构成的一个矩形。

```
*****
*****
*****
*****
*****
*****
*****
*****
```

将这个图案想象成由行和列构成，就可以看出它总共有 8 行和 6 列。以下代码显示一行共 6 个星号：

```
for col in range(6):
    print('*', end='')
```

在程序中或以交互模式运行上述代码，它将产生以下输出：

```
*****
```

为了完成整个图案，我们需要执行这个循环 8 次。可以将循环放到另一个迭代 8 次的循环中，如下所示：

```
1  for row in range(8):
2      for col in range(6):
3          print('*', end='')
4      print()
```

外部循环将迭代 8 次。每一次迭代，内部循环都会迭代 6 次。注意，第 4 行在打印完每一行后都调用了一次 `print()` 函数。必须这样做才能使屏幕光标换行。没有该语句，所有星号都会在屏幕上连成一长行。

可以轻松地写一个程序来提示用户输入行数和列数，并打印指定大小的矩形图案，如程序 4-23 所示。

程序 4-23 (rectangluar_pattern.py)

```
1  # 这个程序显示由星号
2  # 构成的矩形图案。
3  rows = int(input('多少行? '))
4  cols = int(input('多少列? '))
5
6  for r in range(rows):
7      for c in range(cols):
8          print('*', end='')
9      print()
```

程序输出（用户输入的内容加粗>）

```
多少行? 5 Enter
多少列? 10 Enter
*****
*****
*****
*****
*****
```

再来看看另一个例子。假设要在屏幕上打印由星号构成的一个三角形。

```
*
**
***
****
*****
*****
*****
*****
```

同样将这个图案想象成由行和列构成。总共 8 行，第一行有一列，第二行有两列，第三行有三列……第八行有八列。程序 4-24 展示了如何生成该图案。

程序 4-24 (triangle_pattern.py)

```
1 # 这个程序显示三角形图案
2 BASE_SIZE = 8
3
4 for r in range(BASE_SIZE):
5     for c in range(r + 1):
6         print('*', end='')
7     print()
```

程序输出

```
*
**
***
****
*****
*****
*****
*****
```

先来看看外层循环。第 4 行的表达式 `range(BASE_SIZE)` 生成包含以下整数序列的一个可迭代对象 (iterable):

0, 1, 2, 3, 4, 5, 6, 7

所以，当外层循环迭代时，变量 `r` 会被分别赋值为 0~7。第 5 行开始的内部循环的范围表达式为 `range(r + 1)`。下面描述了内层循环是如何执行的。

- 外层循环第一次迭代时，变量 `r` 被赋值 0。表达式 `range(r + 1)` 导致内层循环迭代一次，打印一个星号。
- 外层循环第二次迭代时，变量 `r` 被赋值 1。表达式 `range(r + 1)` 导致内层循环迭代两次，打印两个星号。
- 外层循环第三次迭代时，变量 `r` 被赋值 2。表达式 `range(r + 1)` 导致内层循环迭代三次，打印三个星号。以此类推。

再来看看另一个例子。假设要显示以下阶梯图案。

```
#
 #
  #
   #
    #
     #
```

图案共有 6 行，可以这样描述每一行的内容：一定数量的空格加一个#字符。下面描述了每一行。

- 第 1 行： 0 个空格加一个#字符
- 第 2 行： 1 个空格加一个#字符
- 第 3 行： 2 个空格加一个#字符
- 第 4 行： 3 个空格加一个#字符
- 第 5 行： 4 个空格加一个#字符
- 第 6 行： 5 个空格加一个#字符

可以用内外两层的一个嵌套循环来显示这个图案，下面是它的工作方式。

- 外层循环迭代 6 次。每次迭代都执行以下操作：
 - 内层循环显示正确数量的空格。
 - 然后显示一个#字符。

程序 4-25 展示了它的 Python 代码。

程序 4-25 (stair_step_pattern.py)

```
1 # 这个程序显示了阶梯图案
2 NUM_STEPS = 6
3
4 for r in range(NUM_STEPS):
5     for c in range(r):
6         print(' ', end='')
7         print('#')
```

程序输出

```
#
 #
  #
   #
    #
```

在第 4 行的表达式 `range(NUM_STEPS)` 生成包含以下整数序列的一个可迭代对象：


```
0, 1, 2, 3, 4, 5
```

所以，外层循环将迭代 6 次，并分别为变量 `r` 赋值 0~5。下面描述了内层循环是如何执行的。

- 外层循环第一次迭代时，变量 `r` 被赋值 0。内层循环 `for c in range(0):` 迭代零次。换言之，内层循环这一次不执行。
- 外层循环第二次迭代时，变量 `r` 被赋值 1。内层循环 `for c in range(1):` 迭代一次，打印一个空格。
- 外层循环第三次迭代时，变量 `r` 被赋值 2。内层循环 `for c in range(2):` 迭代两次，打印两个空格。依此类推。

4.8 为循环使用 `break`，`continue` 和 `else`

概念：`break` 语句造成循环提前终止。`continue` 语句造成循环停止当前迭代并开始下一次迭代。为 Python 循环使用的 `else` 子句仅在循环正常结束，而没有遇到 `break` 语句的前提下执行。

 **警告：** 使用 `break` 和 `continue` 语句时要非常小心。由于它们绕过了对循环迭代进行控制的正常条件判断，因此可能会使代码难以理解和调试。

4.8.1 break 语句

当循环中遇到 `break` 语句，会造成循环立即终止，如程序 4-26 所示。

程序 4-26 (`while_loop_with_break.py`)

```
1 # 这个程序演示在 while 循环中使用 break 语句
2 n = 0
3 while n < 100:
4     print(n)
5     if n == 5:
6         break
7     n += 1
8
9 print(f'循环终止，此时的 n 值为{n}。')
```

程序输出

```
0
1
2
3
4
5
循环终止，此时的 n 值为 5。
```

不注意看，你可能会以为循环将一直迭代，直到 `n` 达到值 `100`。但是，在 `n` 等于 `5` 时，就会执行第 6 行的 `break` 语句，导致循环提前终止。程序 4-27 展示了在 `for` 循环中使用 `break` 语句的一个例子。

程序 4-27 (`for_loop_with_break.py`)

```
1 # 这个程序演示了在 for 循环中使用 break 语句
2 for n in range(100):
3     print(n)
4     if n == 5:
5         break
6
7 print(f'循环终止，此时的 n 值为{n}。')
```

程序输出

```
0
```

```
1
2
3
4
5
循环终止，此时的 n 值为 5。
```

你可能以为这个程序中的 `for` 循环会迭代 100 次，因为在第 2 行中，`range` 函数生成了 0~99 的数字序列。但是，在 `n` 等于 5 时，就会执行第 5 行的 `break` 语句，导致循环提前终止。



注意：在嵌套循环中，`break` 语句仅终止它当前所在的循环。例如，如果 `break` 语句在内层循环中使用，那么它只会终止内层循环。外层循环（不管有多少个）将继续迭代。

4.8.2 continue 语句

`continue` 语句造成循环的当前迭代立即结束，并重新开始下一次迭代。遇到 `continue` 语句后，循环体中在它之后的所有语句都会被跳过，循环开始下一次迭代（如果有下一轮的话）。

程序 4-28 是在 `while` 循环中使用 `continue` 语句的一个例子。该程序打印 1~10 的整数，并跳过每个能被 3 整除的数字。

程序 4-28 (`while_loop_with_continue.py`)

```
1 # 这个程序演示了在 while 循环中使用 continue 语句
2 n = 0
3 while n < 10:
4     n += 1
5     if n % 3 == 0:
6         continue
7     print(n)
```

程序输出

```
1
2
4
5
7
8
```


程序 4-29 展示了如何用 `for` 循环来写相同的程序。

程序 4-29 (`for_loop_with_continue.py`)

```
1 # 这个程序演示了在 for 循环中使用 continue 语句
2 for n in range(1, 11):
3     if n % 3 == 0:
4         continue
5     print(n)
```

程序输出

```
1
2
4
5
7
8
10
```

程序 4-29 的 `for` 循环使用 `n` 作为目标变量来遍历 1~10 的数字序列。第 3 行判断 `n` 是否能被 3 整除。如果是，那么第 4 行的 `continue` 语句将结束循环的当前迭代，导致循环跳过第 5 行的语句，并开始下一次迭代。



注意：在嵌套循环中，`continue` 语句仅影响它当前所在的循环。例如，如果 `continue` 语句在内层循环使用，那么它只会导致内层循环结束当前迭代。

4.8.3 在循环中使用 `else` 子句

在 Python 中，`while` 循环和 `for` 循环有一个可选的 `else` 子句。以下是带有 `else` 子句的 `while` 循环的常规格式：

```
while 条件:
    语句
    语句
    ...
else:
    语句
    语句
    ...
```

`else` 子句后跟一个缩进的语句块。以下是带有 `else` 子句的 `for` 循环的常规格式：

```
for 变量 in [值1, 值2, ...]:
    语句
    语句
    ...
else:
    语句
    语句
    ...
```

仅在循环包含 `break` 语句时，循环的 `else` 子句才有用。这是因为 `else` 子句仅在循环正常结束，而没有遇到 `break` 语句的前提下执行。如果循环因为 `break` 语句而终止，那么 `else` 子句不会执行其语句块。程序 4-30 展示了一个例子。

程序 4-30 (for_else_break.py)

```
1  # 这个程序演示了一个带有 else 子句的循环。
2  # 这个例子会执行 break 语句。
3  for n in range(10):
4      if n == 5:
5          print('提前终止整个循环。')
6          break
7      print(n)
8  else:
9      print(f'循环完毕后, n 值为{n}。')
```

程序输出

```
0
1
2
3
4
提前终止整个循环。
```

第 3 行将目标变量 `n` 的值设为 `0~9`，要求 `for` 循环迭代 10 次。但是，`n` 等于 5 时会执行第 6 行的 `break` 语句，循环提前终止。从程序输出可以看出，提前终止的循环不会执行它的 `else` 子句。

而在程序 4-31，循环正常结束，所以会执行它的 `else` 子句。

程序 4-31 (for_else_no_break.py)

```
1 # 这个程序演示了一个带有 else 子句的循环。
2 # 这个例子永远遇不到 break 语句。
3 for n in range(3):
4     if n == 5:
5         print('提前终止整个循环。')
6         break
7     print(n)
8 else:
9     print(f'循环完毕后, n 值为{n}。')
```

程序输出

```
0
1
2
循环完毕后, n 值为 2。
```

它与程序 4-30 甚至一致，只是修改了第 3 行开始的 for 循环。在这个程序中，for 循环仅迭代 3 次，目标变量 n 被依次赋值 0，1 和 2。由于 n 从未被赋值 5，所以第 6 行的 break 语句永远不会执行。循环将正常完成所有迭代，并执行 else 子句。

4.9 海龟图形：用循环来画图

概念：可以使用循环来绘制从简单到复杂的各种图形。

配合海龟图形库来使用循环，可以绘制从简单到复杂的各种图形。例如，以下 for 循环迭代 4 次来绘制 100 像素边长的正方形：

```
for x in range(4):
    turtle.forward(100)
    turtle.right(90)
```

以下代码展示了另一个例子，for 循环迭代 8 次来绘制如图 4-9 所示的八边形。

```
for x in range(8):
    turtle.forward(100)
    turtle.right(45)
```

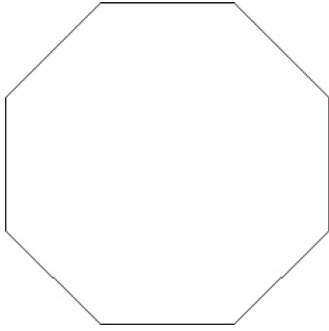


图 4-9 八边形

程序 4-32 展示了如何使用循环来绘制同心圆。程序输出如图 4-10 所示。

程序 4-32 (concentric_circles.py)

```
1  # 同心圆
2  import turtle
3
4  # 具名常量
5  NUM_CIRCLES = 20
6  STARTING_RADIUS = 20
7  OFFSET = 10
8  ANIMATION_SPEED = 0
9
10 # 设置海龟
11 turtle.speed(ANIMATION_SPEED)
12 turtle.hideturtle()
13
14 # 设置第一个圆的半径
15 radius = STARTING_RADIUS
16
17 # 绘制所有圆
18 for count in range(NUM_CIRCLES):
19     # 画圆
20     turtle.circle(radius)
21
22     # 获取下一个圆的坐标
23     x = turtle.xcor()
24     y = turtle.ycor() - OFFSET
25
26     # 计算下一个圆的半径
27     radius = radius + OFFSET
28
29     # 为下一个圆定位海龟
```

```
30 turtle.penup()
31 turtle.goto(x, y)
32 turtle.pendown()
```

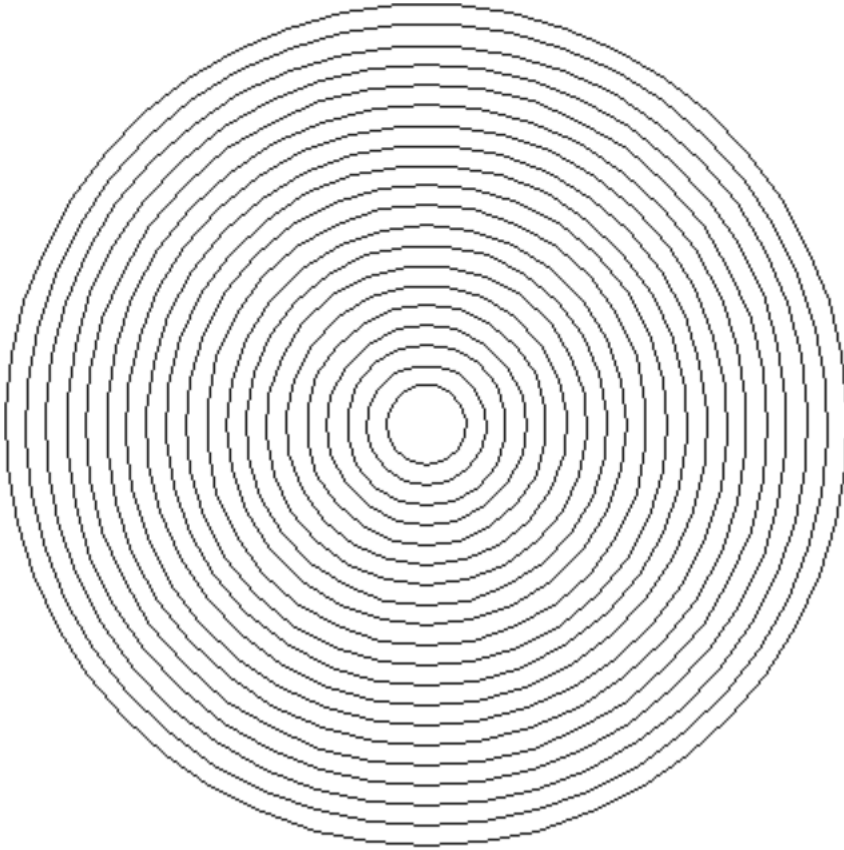


图 4-10 同心圆

重复绘制一个简单的形状，并且每次绘制时都让海龟稍微偏转不同的角度，可以创建许多有趣的图形。例如，图 4-11 的图形是通过在一个循环中绘制 36 个圆来创建的。每画一个圆，海龟就向左偏转 10 度。程序 4-33 展示了完整代码。

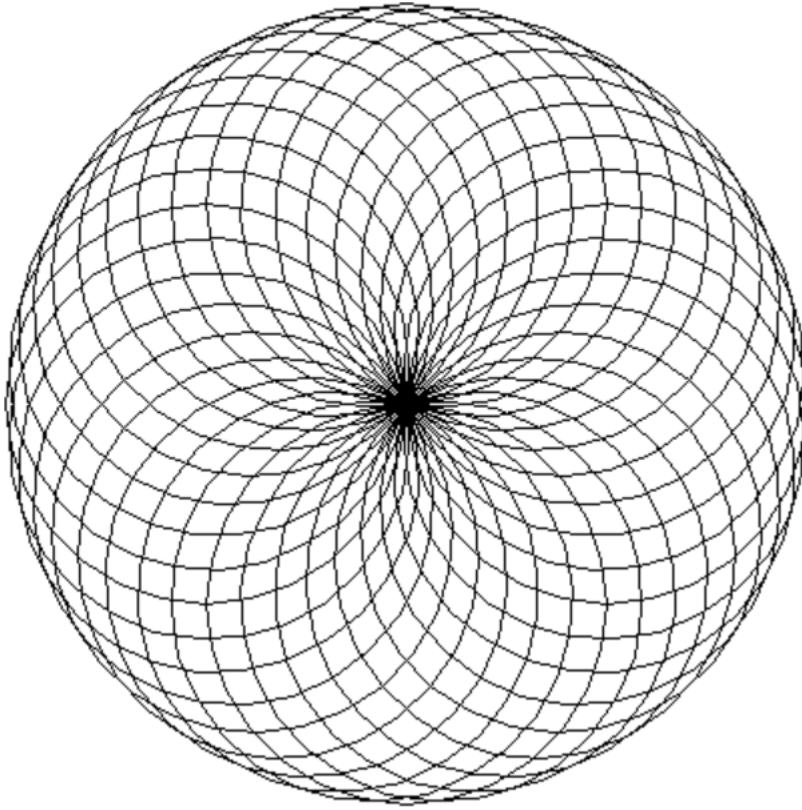


图 4-11 用圆来创建的复杂图形

程序 4-33 (spiral_circles.py)

```
1 # 这个程序使用重复的圆来画图
2 import turtle
3
4 # 具名常量
5 NUM_CIRCLES = 36 # 要绘制的圆的数量
6 RADIUS = 100 # 每个圆的半径
7 ANGLE = 10 # 偏转角度
8 ANIMATION_SPEED = 0 # 动画速度
9
10 # 设置动画速度
11 turtle.speed(ANIMATION_SPEED)
12
13 # 画 36 个圆，每画一个，
14 # 海龟就偏转 10 度。
15 for x in range(NUM_CIRCLES):
16     turtle.circle(RADIUS)
17     turtle.left(ANGLE)
```

程序 4-23 展示了另一个例子。它用一个循环来绘制 36 条直线，生成如图 4-12 所示的图形。

程序 4-23 (spiral_lines.py)

```
1  # 这个程序使用重复的线段来画图
2  import turtle
3
4  # 具名常量
5  START_X = -200      # 起始 X 坐标
6  START_Y = 0        # 起始 Y 坐标
7  NUM_LINES = 36     # 经绘制的线段数
8  LINE_LENGTH = 400  # 每条线的长度
9  ANGLE = 170        # 偏转角度
10 ANIMATION_SPEED = 0 # 动画速度
11
12 # 将海龟移至起始位置
13 turtle.hideturtle()
14 turtle.penup()
15 turtle.goto(START_X, START_Y)
16 turtle.pendown()
17
18 # 设置动画速度
19 turtle.speed(ANIMATION_SPEED)
20
21 # 画 36 条线，每画一条，
22 # 海龟就偏转 170 度。
23 for x in range(NUM_LINES):
24     turtle.forward(LINE_LENGTH)
25     turtle.left(ANGLE)
```

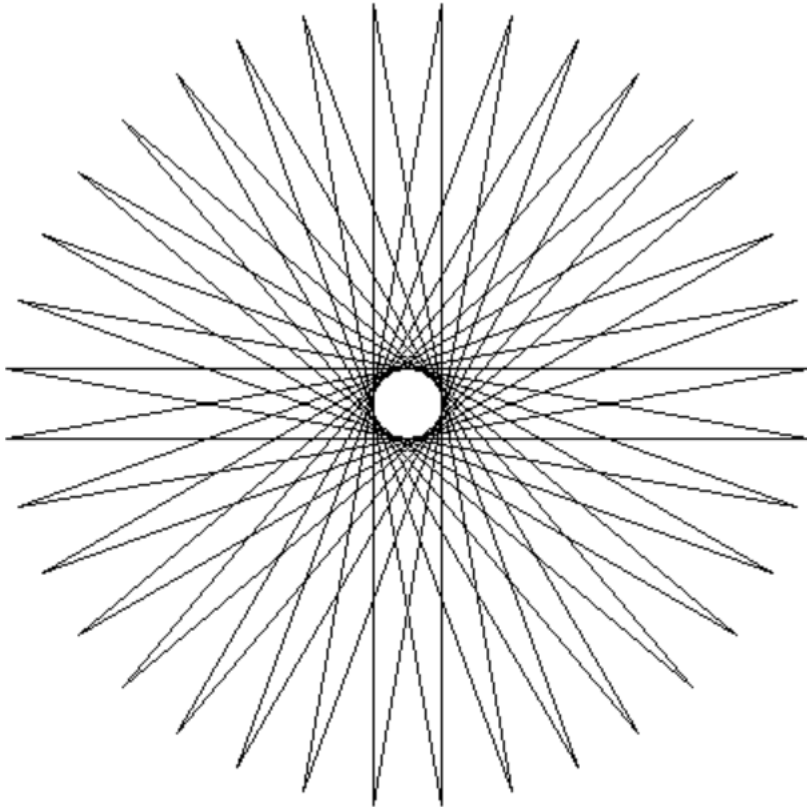


图 4-12 程序 4-34 创建的图形

复习题

选择题

1. _____控制的循环使用真/假条件来控制重复次数。
a. 布尔 b. 条件 c. 决策 d. 计数
2. _____控制的循环会重复特定的次数。
a. 布尔 b. 条件 c. 决策 d. 计数
3. 循环的每一次重复都称为一次_____。
a. 周期 b. 旋转 c. 绕圈 d. 迭代
4. `while` 循环是一种_____循环。

a. 预测试 b. 不测试 c. 预审 d. 后代

5. _____循环无法停止，只能一直重复，直到程序被强行中断。

a. 确定性 b. 长 c. 无限 d. 永恒

6. -=操作符是一种_____操作符

a. 关系型 b. 复合赋值 c. 复杂赋值 d. 反向赋值

7. _____变量用于容纳累加和。

a. 哨兵 b. 求和 c. 总和 d. 累加器

8. _____是一个特殊值，它指明值列表没有更多的值可供处理。该值不能和序列中的常规值混淆。

a. 哨兵 b. 标志 c. 信号 d. 累加器

9. GIGO 是_____的简称。

a. 好的输入，好的输出。 b. 垃圾进，垃圾出

c. GIGahertz Output d. GIGabyte Operation

10. 除非有正确的_____，否则程序输出的正确性无法保证。

a. 编译器 b. 编程语言 c. 输入 d. 调试器

11. 进入校验循环之前的那个输入操作称为_____。

a. 预校验读取 b. 原始读取 c. 初始化读取 d. 预读

12. 校验循环也称为_____。

a. 错误陷阱 b. 末日循环 c. 防错循环 d. 防卫循环

判断题

1. 条件控制循环总是重复特定次数。

2. while 循环是一个预测试循环。

3. 以下语句从 x 中减去 1: $x = x - 1$

4. 累加器变量不需要初始化。

5. 在嵌套循环中，外层循环每选一次，内层循环都要完成它的所有迭代。
6. 为了计算嵌套循环的总迭代次数，将所有循环的迭代次数相加即可。
7. 输入校验的过程是这样的：当用户输入无效数据时，程序应询问用户“你确定要输入该数据吗？”如果用户回答“是”，那么程序应该接受数据。

简答题

1. 什么是条件控制循环？
2. 什么是计数控制循环？
3. 什么是无限循环？写一段无限循环的代码。
4. 为什么正确初始化累加器变量至关重要？
5. 使用哨兵有什么好处？
6. 为什么必须谨慎选择用作哨兵的值？
7. “垃圾进，垃圾出”这句话是什么意思？
8. 给出输入校验过程的一般性的描述。

算法工作台

1. 写 `while` 循环，让用户输入一个数字。将这个数乘以 10，结果赋给名为 `product` 的变量。只要乘积小于 100，循环就继续迭代。
2. 写 `while` 循环，要求用户输入两个数字。两个数相加并显示总和。循环应询问用户是否希望再次执行该操作。如果是，循环就继续迭代，否则就终止。
3. 写 `for` 循环来显示以下一系列数字：

0, 10, 20, 30, 40, 50 . . . 1000

4. 写循环要求用户输入数字。循环应迭代 10 次，用一个变量来存储输入的数字的累加和。
5. 写循环来计算以下一系列数字的总和。

$$\frac{1}{30} + \frac{2}{29} + \frac{3}{28} + \dots + \frac{30}{1}$$

6. 使用复合赋值操作符重写以下语句。

-
- a. $x = x + 1$
 - b. $x = x * 2$
 - c. $x = x / 10$
 - d. $x = x - 100$

- 7. 写嵌套循环来显示 10 行字符#。每行显示 15 个#。
- 8. 写代码提示用户输入一个非零的正数，并对输入进行校验。
- 9. 写代码提示用户输入 1~100 的数字，并对输入进行校验。

编程练习

1. bug 收集者

bug 收集者连续 5 天每天都会收集 bug。写一个程序，记录 5 天内收集到的 bug 总数。循环应询问每天收集到的 bug 数量。循环完成后，程序应显示收集到的 bug 总数。

 视频讲解：Bug Collector Problem

2. 燃烧的卡路里

在一台跑步机上跑步，每分钟会燃烧 4.2 卡路里。写一个程序，使用循环来显示 10，15，20，25 和 30 分钟后会燃烧多少卡路里。

3. 预算分析

写一个程序，要求用户输入一个月的预算金额。然后，循环应提示用户输入该月的每项支出并保留累加和。循环完成后，程序应显示用户超出或低于预算多少金额。

4. 行驶距离

车辆的行驶距离可以用以下公式来计算：

$$\text{距离} = \text{速度} \times \text{时间}$$

例如，假定车辆以每小时 40 英里的速度行驶 3 小时，那么行驶距离为 120 英里。写程序来询问用户车速（单位：英里/小时）和行驶小时数。然后，使用循环来显示车辆在这段时间内，每过一小时所行驶的总距离。以下是示例输出。

```
车速是多少(MPH)? 40 
行驶了多少小时? 3 
小时      行驶距离
-----
1          40
2          80
```

5. 平均降雨量

写一个程序，使用嵌套循环来收集数据并计算几年内的月平均降雨量。程序首先询问年数。外层循环每年迭代一次。内层循环迭代 12 次，每月一次。内层循环的每次迭代都会询问用户该月的降雨量。所有迭代结束后，程序应显示在此期间的月数、总降雨量以及月平均降雨量。

6. 摄氏和华氏温度换算

写程序来显示一个表格，列出 0~20 的每一摄氏度及其对应的华氏温度。摄氏温度和华氏温度的换算公式如下所示。

$$F = \frac{9}{5}C + 32$$

其中， F 是华氏温度， C 是摄氏温度。程序必须使用循环来显示这个表格。

7. 工资计算

写程序来计算一个人在一段时间内的工资。假定第一天的工资是一分钱，第二天两分钱，并如此每天翻倍。程序应询问用户天数，并输出一个表格，列出每天的工资，最后显示这些天的总工资（以元为单位，而不要以分为单位）。

8. 数字求和

用循环来写一个程序，要求用户输入一系列正数。用户应输入一个负数来表示输入结束。程序应显示所有正数之和。

9. 海平面

假设目前海平面每年上升约 1.6 毫米，创建一个应用程序来显示海平面未来 25 年内每一年上升的毫米数。

10. 学费上涨

某大学全日制学生的学费为每学期 8000 美元。已宣布未来 5 年学费每年上涨 3%。用循环来写程序，显示未来 5 年内每一年预计的学期学费。

11. 计算阶乘

数学中用符号 $n!$ 来表示非负整数 n 的阶乘。 n 的阶乘是 $1 \sim n$ 的所有非负整数的乘积。例如：

$$7! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 = 5040$$

$$4! = 1 \times 2 \times 3 \times 4 = 24$$

写程序来提示输入一个非负整数，使用循环计算它的阶乘，最后显示结果。

12. 种群规模

写程序来预测生物体种群的大致规模。程序应提示输入生物体的起始数量、每日平均种群增长（百分比）以及生物体继续繁殖的天数。例如，假设用户输入以下值：

起始生物体数量：2
每日平均增长：30%
繁殖天数：10

程序应显示以下数据表：

天	近似种群规模（生物体数量）
1	2
2	2.6
3	3.38
4	4.394
5	5.7122
6	7.42586
7	9.653619
8	12.5497
9	16.31462
10	21.209

13. 使用嵌套循环写一个程序来绘制以下图案。

```
*****  
*****  
*****  
****  
***  
**  
*  
*
```

14. 使用嵌套循环写一个程序来绘制以下图案。

```
##  
# #
```

```
# #  
# #  
# #  
# #
```

15. 海龟图形：重复正方形

本章展示了如何用循环来绘制正方形。写一个海龟图形程序，使用嵌套循环来绘制 100 个正方形，以创建如图 4-13 所示的图形。

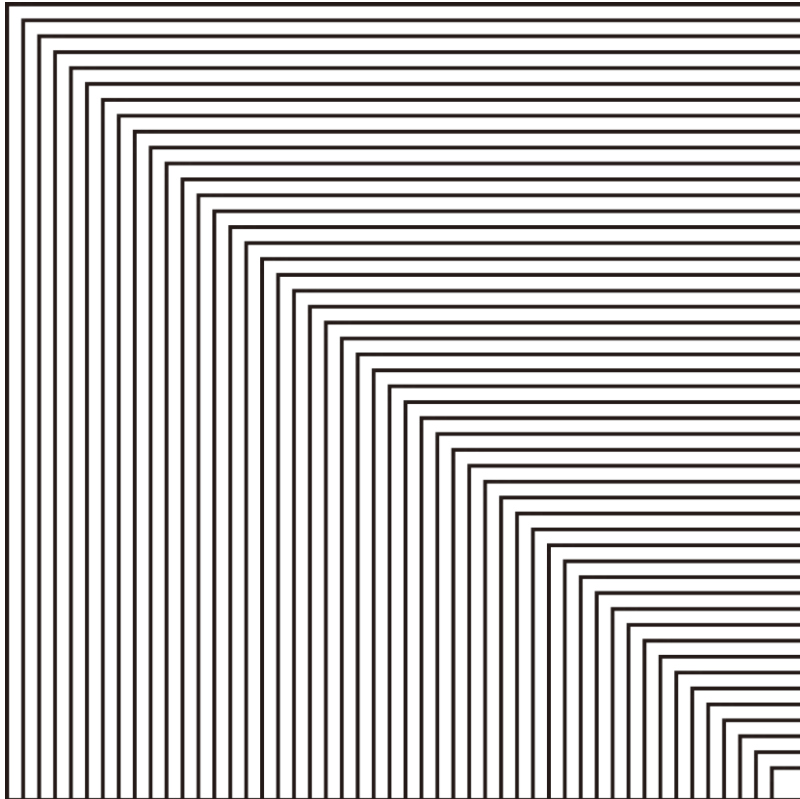


图 4-13 重复正方形

16. 海龟图形：星星图案

使用海龟图形库和循环来绘制如图 4-14 所示的图形。

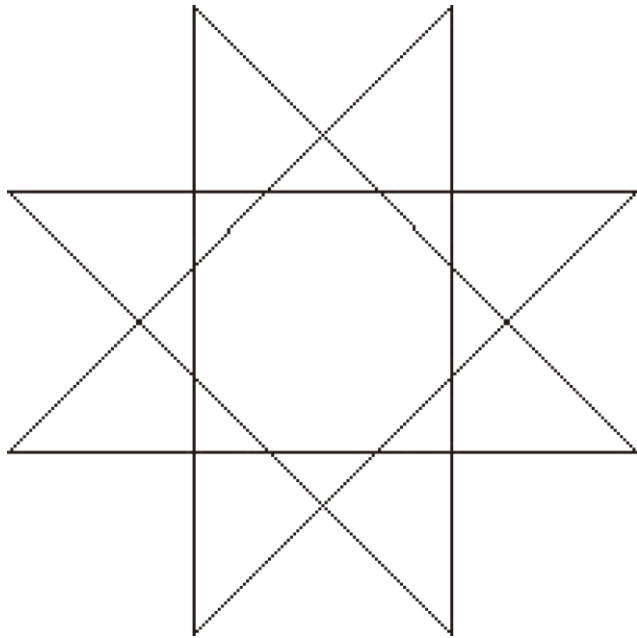


图 4-14 星星图案

17. 海龟图形：催眠图案

使用海龟图形库和循环来绘制如图 4-15 所示的图形。

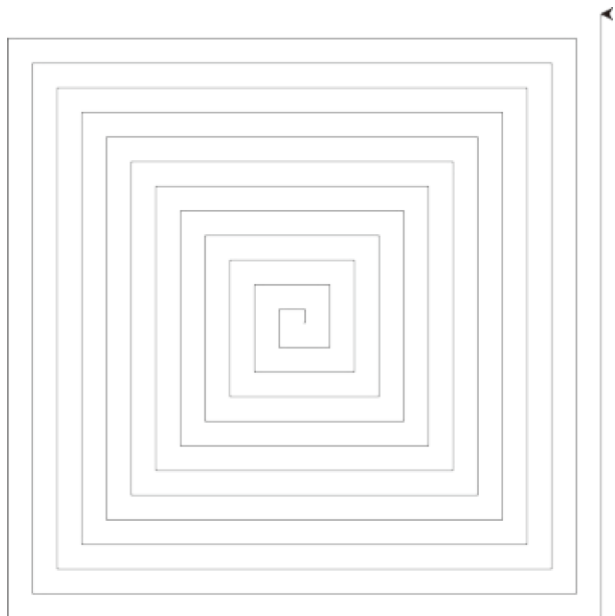


图 4-15 催眠图案

18. 海龟图形：STOP 标志

本章展示了如何用循环来绘制八边形。写一个海龟图形程序，使用循环来绘制一个八边形，并在其中心显示单词“STOP”。STOP 标志应位于图形窗口的中央。

第 5 章 函数

5.1 函数简介

概念：函数是程序中用于执行特定任务的一组语句。

第 2 章描述了计算员工工资的一个简单算法，即工作时数乘以时薪。然而，现实的工资算法所做的事情远不止于此。真正的工资计算任务将由多个子任务组成，例如：

- 获取员工时薪
- 获取工作时数
- 计算员工的税前工资（gross pay，毛收入）
- 计算加班工资
- 计算税和社保代扣
- 计算税后工资（net pay，净收入）
- 打印工资单

大多数程序执行的任务都足够大，可以分解为多个子任务。因此，程序员经常将程序分解为容易处理的小块，称为**函数**。函数是程序中用于执行特定任务的一组语句。不要将大型程序写成一长串语句。相反，把它编写为几个小函数，每个都执行任务的特定部分。然后，按照恰当的顺序来执行这些小函数，从而完成整个任务的执行。

这种编程方法有时称为**分而治之**，因为它将一个大型任务分解为几个容易执行的小任务。图 5-1 通过比较两个程序来说明了这一思路：一个程序使用长且复杂的语句序列来执行任务，另一个将任务分解为较小的任务，每个小任务都由单独的函数执行。

在程序中使用函数时，通常将程序中的每个任务都隔离在其自己的函数中。例如，一个真实的工资计算程序可能包含以下函数：

- 一个函数获取员工时薪
- 一个函数获取工作时数
- 一个函数计算税前工资
- 一个函数计算加班工资
- 一个函数计算税和社保代扣
- 一个函数计算税后工资
- 一个函数打印工资单

每个任务都用它自己的函数来写的程序称为**模块化程序**。

代码更简单

将程序分解成多个函数后，程序代码会变得更简单，更容易理解。多个小函数比一长串语句更容易阅读。

代码重用

函数还有利于减少程序中代码的重复。如果一个特定的操作要在程序中的多个地方执行，那么可以写一个函数来执行该操作，并在需要时调用该函数。函数的这一好处称为**代码重用**，因为只需写一次执行任务的代码，就可以在每次需要时重复使用。

更好的测试

将程序中的每个任务都包含到它们自己的函数中，测试和调试变得更简单了。程序员可以单独测试程序的每个函数，确定它是否正确执行了操作。这使我们更容易隔离和修复错误。

更快的开发

假设一个程序员或者一个程序员团队要开发多个程序，并发现每个程序都要执行几个通用的任务，例如询问用户名和密码、显示当前时间等。重复写这些任务的代码是没有意义的。相反，可以将通用任务写成函数，并将这些函数集成到每个需要的程序中。

更容易促进团队协作

函数还促进了团队协作。如果一个程序被开发为一组函数，每个函数都执行一个单独的任务，那么不同的程序员可以被分配编写不同函数的工作。

5.1.2 void 函数和返回值的函数

本章要学习编写两种类型的函数：**void 函数**^①和返回值的函数。调用 **void 函数** 时，它只是执行其中包含的语句，然后终止。而调用返回值的函数时，它会执行所包含的语句，然后返回一个值给调用它的语句。**input** 函数就是一个典型的返回值的函数。调用 **input** 函数后，它获取用户在键盘上键入的数据，并以字符串形式返回该数据。**int** 和 **float** 函数也是返回值的函数。向 **int** 函数传递一个实参，它返回将实参转换为整数后的值。类似地，向 **float** 函数传递一个实参，它返回实参转换为浮点数后的值。

^① 译注：**void 函数**就是不返回值的函数。和其他语言不同，Python 并没有专门提供一个 **void** 关键字。

void 函数是我们学习编写的第一种函数类型。

检查点

- 5.1 什么是函数？
- 5.2 什么是“分而治之”？
- 5.3 函数如何为代码的重用提供帮助？
- 5.4 函数如何使多个程序的开发更快？
- 5.5 函数如何使程序员团队更容易开发程序？

5.2 定义和调用 void 函数

概念：函数的代码称为函数定义。要执行函数，写调用该函数的语句即可。

5.2.1 函数名


在讨论创建和使用函数的过程之前，首先要了解一下函数名。正如要为程序中使用的变量命名一样，也要为函数命名。函数名应该有足够的描述性，使任何人在读你的代码时都能合理地猜到这个函数是做什么的。

Python 的函数命名规则和变量一样，如下所示。

- 不能使用 Python 关键字作为函数名（关键字列表请参见表 1-2）。
- 函数名不能包含空格。
- 第一个字符只能是 a~z 或 A~Z 的字母或下划线（_）。
- 在第一个字符之后，可以使用 a~z 或 A~Z 的任何字母、0~9 的任何数字或下划线的组合。
- 字母严格区分大小写。

由于函数执行的具体的行动，所以大多数程序员都喜欢在函数中使用动词。例如，计算税前工资的函数可以命名为 `calculate_gross_pay`。这个名称可以让任何阅读代码的人清楚地知道函数是在计算什么。计算（`calculate`）什么呢？计算的是工资总额（`gross_pay`）。其他一些好的函数名包括：`get_hours`，`get_pay_rate`，`calculate_overtime`，`print_check`，等等。每个函数名都描述了函数所做的事情。

5.2.2 定义和调用函数

 视频讲解：Defining and Calling a Function

为了创建函数，我们需要写它的**定义**。下面是 Python 函数定义的常规格式。

```
def function_name():  
    语句  
    语句  
    ...
```

第一行称为**函数头**。它标志着函数定义的开始。函数头以关键字 `def` 开头，后跟函数名，再后面是一对圆括号，最后是冒号。

下一行起是一个**语句块**，其中包含一组语句。每次执行函数，都会执行块中的这些语句。注意，块中的语句要一致地缩进。这种缩进是必须的，因为 Python 解释器使用它来区分块的开始和结束。

下面是一个示例函数。注意，它不是一个完整的程序。完整程序将在稍后展示。

```
def message():  
    print('我是亚瑟，')  
    print('不列颠的国王。')
```

上述代码定义了一个名为 `message` 的函数。函数中的块包含两个语句。执行该函数将导致这两个语句的执行。

5.2.3 调用函数

函数定义规定了函数要做什么，但它只是定义，并不会导致函数的实际执行。要执行函数，必须**调用**它。我们这样调用 `message` 函数：

```
message()
```

调用函数时，解释器会跳转到该函数并执行其语句块中的语句。当语句块结束时，解释器会跳回调用函数的位置，程序从这个地方恢复执行。当这种情况发生时，我们说函数**返回**。为了充分说明函数调用具体如何工作，下面来看看程序 5-1。

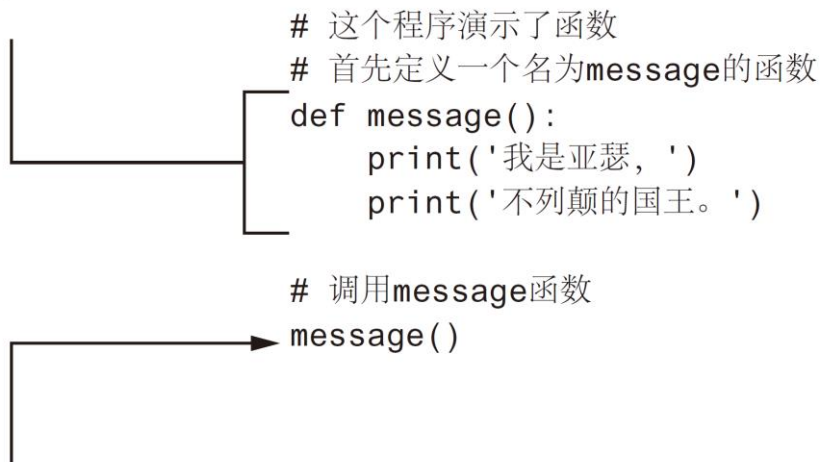
程序 5-1 (function_demo.py)

```
1  # 这个程序演示了函数  
2  # 首先定义一个名为 message 的函数  
3  def message():  
4      print('我是亚瑟，')  
5      print('不列颠的国王。')  
6  
7  # 调用 message 函数  
8  message()
```

来看看这个程序在运行时发生了什么。首先，解释器忽略第 1 行和第 2 行的注释。然后，

它读取第 3 行的 `def` 语句。这会在内存中创建一个名为 `message` 的函数，其中包含由第 4 行和第 5 行组成的语句块（记住，函数定义只是创建一个函数，不会导致函数的实际执行）。接着，解释器忽略第 7 行的注释，然后执行第 8 行的语句，这是一个**函数调用**。这导致 `message` 函数的实际执行，并打印两行输出。图 5-2 说明了该程序的各个部分。

这些语句导致 `message` 函数的创建



该语句调用 `message` 函数，造成它的实际执行

图 5-2 函数定义和函数调用

程序 5-1 只有一个函数，但完全可以在程序中定义多个函数。事实上，程序通常都有一个 `main` 函数，并在程序启动时调用它。`main` 函数根据需要调用程序中的其他函数。我们一般说 `main` 函数包含了程序的**主线逻辑**，即程序的整体逻辑。程序 5-2 定义并调用了两个函数：`main` 和 `message`。

程序 5-2 (`two_functions.py`)

```
1 # 这个程序有两个函数
2 # 首先定义 main 函数。
3 def main():
4     print('我有话给你。')
5     message()
6     print('再见! ')
7
8 # 接着定义 message 函数
9 def message():
```

```
10     print('我是亚瑟，')
11     print('不列颠的国王。')
12
13 # 调用 main 函数
14 main()
```

程序输出

```
我有话给你。
我是亚瑟，
不列颠的国王。
再见！
```

第 3 行~第 6 行是 `main` 函数的定义，第 9 行~第 11 行是 `message` 函数的定义。第 14 行调用 `main` 函数，如图 5-3 所示。

解释器跳转到 `main` 函数，开始执行其语句块中的语句

```
# 这个程序有两个函数
# 首先定义main函数。
def main():
    print('我有话给你。')
    message()
    print('再见! ')

# 接着定义message函数
def message():
    print('我是亚瑟，')
    print('不列颠的国王。')

# 调用main函数
main()
```


A diagram consisting of a vertical line on the left side of the code block. At the top of this line, an arrow points to the right, towards the `def main():` line. At the bottom of the line, an arrow points to the right, towards the `def message():` line. This indicates the execution flow from the `main()` call to the `main` function definition, and then to the `message` function definition.

图 5-3 调用 `main` 函数

在 `main` 函数中，第一个语句调用 `print` 函数（第 4 行），显示字符串 '我有话给你。' 然后，第 5 行的语句调用 `message` 函数。这导致解释器跳转到 `message` 函数，如图 5-4 所示。`message` 函数中的语句执行完毕后，解释器会返回 `main` 函数中的调用位置，继续执行紧随其后的语句。如图 5-5 所示，该语句显示字符串 '再见!'。

解释器跳转到message函数，开始执行其语句块中的语句

```
# 这个程序有两个函数
# 首先定义main函数。
def main():
    print('我有话给你。')
    message()
    print('再见! ')

# 接着定义message函数
def message():
    print('我是亚瑟, ')
    print('不列颠的国王。')

# 调用main函数
main()
```

图 5-4 调用 message 函数

message函数结束时，解释器会跳转回调用它的位置，从那个地方恢复执行

```
# 这个程序有两个函数
# 首先定义main函数。
def main():
    print('我有话给你。')
    message()
    print('再见! ')

# 接着定义message函数
def message():
    print('我是亚瑟, ')
    print('不列颠的国王。')

# 调用main函数
main()
```

图 5-5 message 函数返回

这样就到了 main 函数的结束位置，所以该函数返回，如图 5-6 所示。没有更多的语句需要执行，因此整个程序结束。



注意：当程序调用一个函数时，程序员通常说程序的控制转移到了该函数。意思其实就是现在由函数来接管程序的执行。

main函数结束时，解释器会跳转回调用它的位置。由于之后没有更多的语句，所以程序终止。

```
# 这个程序有两个函数
# 首先定义main函数。
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# 接着定义message函数
def message():
    print('我是亚瑟，')
    print('不列颠的国王。')

# 调用main函数
main()
```

图 5-6 main 函数返回

5.2.4 Python 的缩进

在 Python 中，语句块的每一行都必须缩进。如图 5-7 所示，在函数头之后缩进的最后一行就是整个块的最后一行。

缩进的最后一行是块中的最后一行

```
def greeting():
    print('早上好! ')
    print('今天将学习函数。')
```

这些语句不在块中

```
print('我会调用greeting函数。')
greeting()
```

图 5-7 块中的所有语句都缩进了

对语句块中代码进行缩进时，一定要确保每行开头的空格数相同，否则会出错。例如，以下函数定义会导致错误，因为各行缩进的空格数不同。

```
def my_function():
    print('现在来看看')
    print('某些完全不同的')
    print('东西。')
```

使用代码编辑器时有两种缩进方法：（1）在行首按 Tab 键插入制表符；（2）在行首按空格键插入空格。对块中的行进行缩进时，制表符和空格都可以使用，但不要同时都用，否则可能混淆 Python 解释器并导致错误。



提示：Python 程序员习惯使用 4 个空格来缩进块中的行。任何数量的空格都可以，只要一个块中的所有行的缩进量一致。



注意：块中的空行会被解释器忽略。

和其他大多数 Python 编辑器一样，IDLE 会自动缩进块中的行。在函数头末尾键入冒号时，之后键入的所有行都会自动缩进。键入块中的最后一行后，按退格（Backspace）键即可取消自动缩进。

检查点

- 5.6 函数定义包括哪两个部分？
- 5.7 “调用函数”是什么意思？
- 5.8 当函数执行时，到达块的末尾后会发生什么？
- 5.9 为什么必须缩进块中的语句？

5.3 使用函数来设计程序

概念：程序员通常使用自上而下设计技术，将一个算法分解为多个函数。

5.3.1 使用了函数的程序的流程图

第 2 章讨论了作为程序设计工具的流程图。如图 5-8 所示，在流程图中，**函数调用**表示成一个两边都有竖条的矩形，中间写上被调用的函数的名称。本例展示了对 `message` 函数的调用。

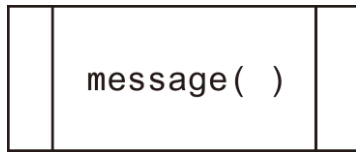


图 5-8 函数调用符号

程序员经常要为程序中的每个函数绘制单独的流程图。例如，图 5-9 展示了程序 5-2 的 `main` 函数和 `message` 函数的流程图。为函数绘制流程图时，注意在起始符号上通常标识函数名，在结束符号上则通常标识 `Return`（返回）。

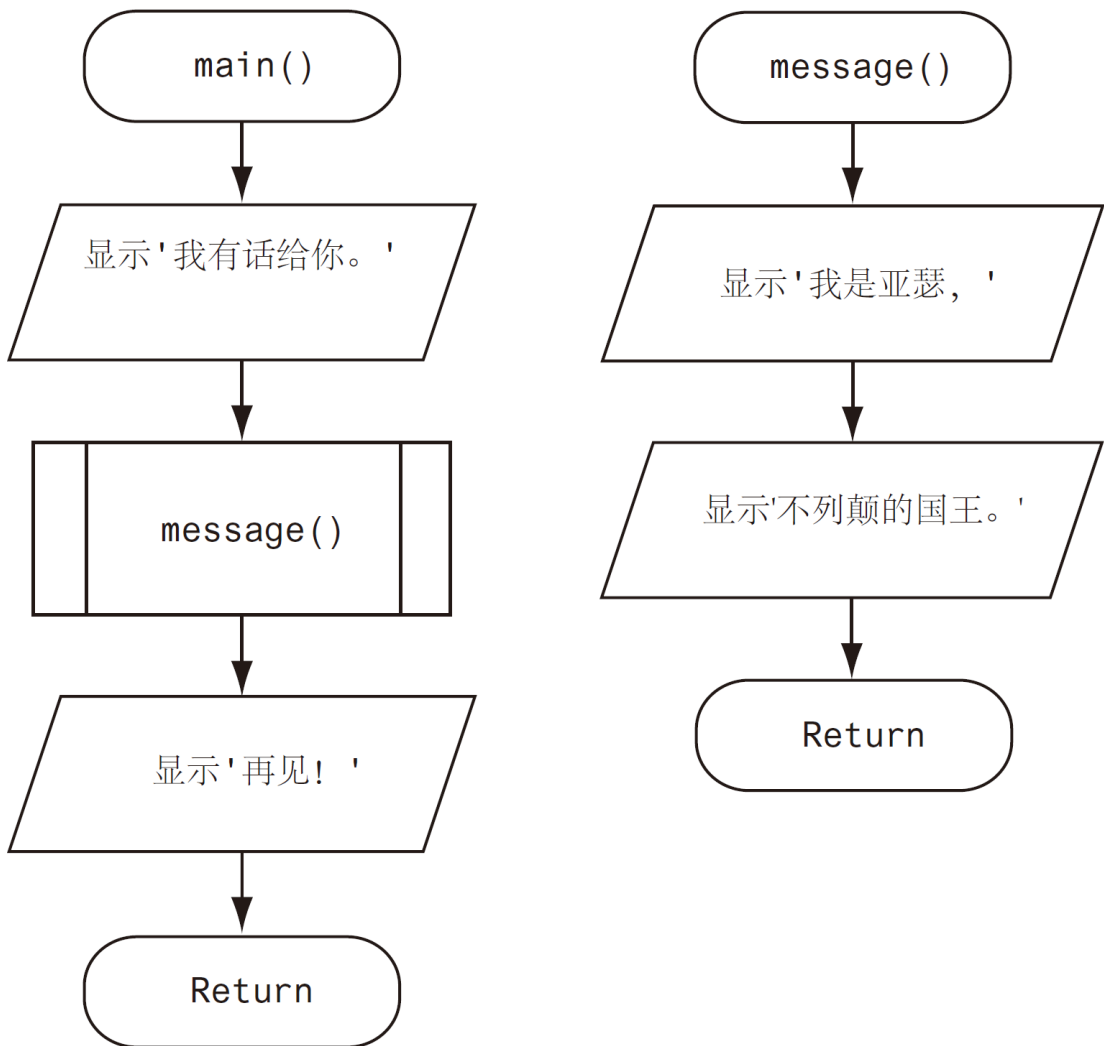


图 5-9 程序 5-2 的流程图

5.3.2 自上而下设计

之前讨论并演示了函数的工作方式。你看到了当函数被调用时，程序的控制如何转移到函数，以及当函数结束时，控制又如何返回函数的调用位置。理解函数的这些机制很重要。

与理解函数工作方式同等重要的是理解如何设计程序来使用函数。程序员通常使用一种称为**自上而下设计**的技术将算法分解成多个函数。下面是自上而下设计的过程。

- 程序要执行的总体任务被分解为一系列子任务。
- 针对每个子任务，判断是否能进一步分解为更多的子任务。重复这一步骤，直到无法分解为更多的子任务。
- 一旦确定了所有子任务，就把它们写成代码。

这个过程之所以称为自上而下设计，是因为程序员从必须执行的最顶层任务开始，然后将其分解为较低层次的子任务。

5.3.3 层次结构图

流程图是以图形方式描述函数内部逻辑流程的一种很好的工具，但它不能直观地表示函数之间的关系。程序员通常使用**层次结构图**来完成这个任务。层次结构图也称为**结构图**，它列出了一系列方框来代表程序中的每个函数。这些方框的连接方式说明了每个函数要调用其他哪些函数。图 5-10 展示了一个虚构的工资计算程序的层次结构图。

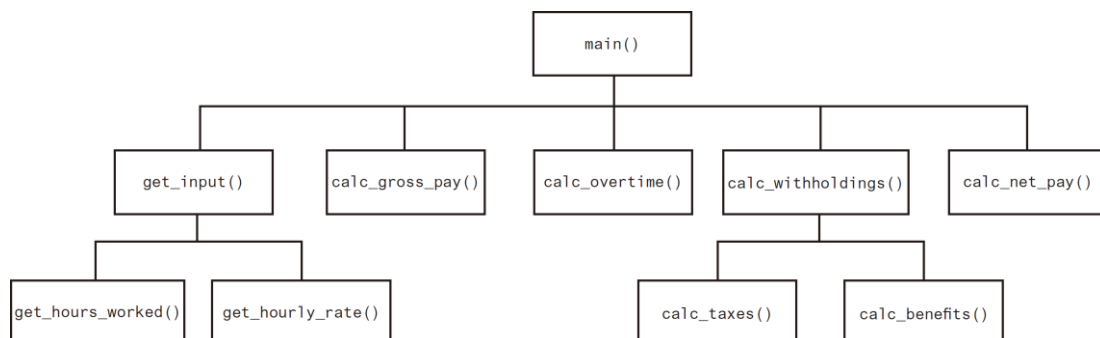


图 5-10 层次结构图

在这个层次结构图中，`main` 函数是最顶层的函数。`main` 函数调用了其他 5 个函数：`get_input`，`calc_gross_pay`，`calc_overtime`，`calc_withholdings` 和 `calc_net_pay`。`get_input` 函数调用了其他两个函数：`get_hours_worked` 和 `get_hourly_rate`。`calc_withholdings` 函数也调用了其他两个函数：`calc_taxes` 和 `calc_benefits`。

注意，层次结构图不显示函数内部执行的步骤。正是由于层次结构图不显示函数工作的任

何细节，所以不能取代流程图或伪代码。

聚光灯：定义和调用函数



Professional Appliance Service 公司提供家用电器维修服务。公司负责人要为公司所有技术服务人员配备一台小型手持电脑，在上面显示各种维修项目的步骤。为了演示这个过程，负责人要求你开发一个程序，为 Acme 干衣机的拆卸显示以下步骤说明：

步骤 1：拔下干衣机的电源插头，将机器移动到远离墙壁的位置。

步骤 2：拧下干衣机背面的六颗螺丝。

步骤 3：取下干衣机的背板。

步骤 4：将干衣机的顶部向上拉。

与负责人交谈后，你确定程序应一次显示一个步骤，并在显示每个步骤后要求用户按 Enter 键查看下一步。下面是该算法的伪代码：

显示初始消息，解释程序的作用。
要求用户按 Enter 键查看步骤 1。
显示步骤 1 的说明。
要求用户按 Enter 键查看下一步。
显示步骤 2 的说明。
要求用户按 Enter 键查看下一步。
显示步骤 3 的说明。
要求用户按 Enter 键查看下一步。
显示步骤 4 的说明。

该算法列出了程序需要执行的最高层级的任务，这成为程序的 main 函数的基础。图 5-11 展示了程序的层次结构图。

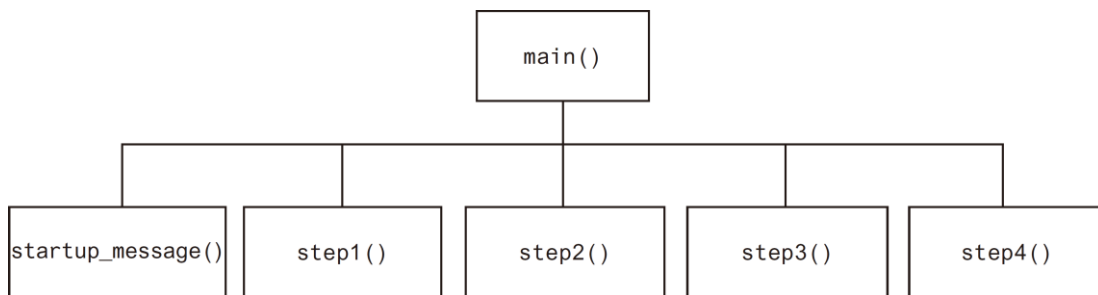


图 5-11 程序的层次结构图

从层次结构图可以看出，main 函数会调用其他几个函数，如下所示。

- `startup_message`: 显示初始消息，告诉技术人员程序的功能。
- `step1`: 显示步骤 1 的说明。
- `step2`: 显示步骤 2 的说明。
- `step3`: 显示步骤 3 的说明。
- `step4`: 显示步骤 4 的说明。

在调用这些函数的间隙，`main` 函数会指示用户按 `Enter` 键查看下一步指令。程序 5-3 展示了完整代码。

程序 5-3 (acme_dryer.py)

```
1  # 这个程序将逐步显示
2  # Acme 干衣机的拆卸说明。
3  # main 函数执行程序的主要逻辑
4  def main():
5      # 显示启动消息
6      startup_message()
7      input('按 Enter 键查看步骤 1。')
8      # 显示步骤 1
9      step1()
10     input('按 Enter 键查看步骤 2。')
11     # 显示步骤 2
12     step2()
13     input('按 Enter 键查看步骤 3。')
14     # 显示步骤 3
15     step3()
16     input('按 Enter 键查看步骤 4。')
17     # 显示步骤 4
18     step4()
19
20 # startup_message 函数在屏幕上
21 # 显示程序的初始消息。
22 def startup_message():
23     print('这个程序指导你')
24     print('拆卸 ACME 干衣机。')
25     print('总共有 4 个步骤。')
26     print()
27
28 # step1 函数显示了
29 # 步骤 1 的说明。
30 def step1():
31     print('步骤 1: 拔下干衣机的电源插头, ')
32     print('将机器移动到远离墙壁的位置。')
33     print()
34
35 # step2 函数显示了
```

```
36 # 步骤 2 的说明。
37 def step2():
38     print('步骤 2: 拧下干衣机背面')
39     print('的六颗螺丝。')
40     print()
41
42 # step3 函数显示了
43 # 步骤 3 的说明。
44 def step3():
45     print('步骤 3: 取下干衣机的')
46     print('背板。')
47     print()
48
49 # step4 函数显示了
50 # 步骤 4 的说明。
51 def step4():
52     print('步骤 4: 将干衣机的顶部')
53     print('向上拉。')
54
55 # 调用 main 函数来启动程序
56 main()
```

程序输出

这个程序指示你
拆卸 ACME 干衣机。
总共有 4 个步骤。

按 Enter 键查看步骤 1。

步骤 1: 拔下干衣机的电源插头,
将机器移动到远离墙壁的位置。

按 Enter 键查看步骤 2。

步骤 2: 拧下干衣机背面
的六颗螺丝。

按 Enter 键查看步骤 3。

步骤 3: 取下干衣机的
背板。

按 Enter 键查看步骤 4。

步骤 4: 将干衣机的顶部
向上拉。

5.3.4 暂停执行直到用户按 Enter 键

有的时候, 你希望程序暂停执行, 方便用户阅读屏幕上显示的消息。当用户准备好继续执行程序时, 可以按 Enter 键恢复程序的执行。在 Python 中, 可以使用 `input` 函数使程序暂停, 直到用户按下 Enter 键。程序 5-3 的第 7 行就是一个例子:

```
input('按 Enter 键查看步骤 1。')
```

该语句显示消息“按 Enter 键查看步骤 1。”并暂停直到用户按 Enter 键。程序的第 10 行、第 13 行和 16 行也使用了这个技术。

5.3.5 使用 pass 关键字

刚开始写程序代码的时候，你可能知道自己想要使用的函数名，但不确定这些函数中的代码的所有细节。在这种情况下，可以使用 `pass` 关键字来创建空函数。以后在知道了代码的细节后，可以回到空函数，用有意义的代码替换 `pass` 关键字。

例如，刚开始写程序 5-3 的代码时，可以为 `step1`，`step2`，`step3` 和 `step4` 函数写空的函数定义，如下所示。

```
def step1():
    pass
def step2():
    pass
def step3():
    pass
def step4():
    pass
```

Python 解释器会忽略 `pass` 关键字，所以上述代码实际创建了 4 个什么都不做的函数。



提示：`pass` 关键字可以在 Python 代码的任何地方作为占位符使用。例如，可以在 `if` 语句中使用它，如下所示。

```
if x > y:
    pass
else:
    pass
```

下面是使用了 `pass` 关键字的一个示例 `while` 循环。

```
while x < 100:
    pass
```

5.4 局部变量

概念：局部变量在函数内部创建，不能由函数外部的语句访问。不同函数可以有相同名称的局部变量，因为函数之间互相看不到对方的局部变量。

在函数内部为变量赋值，就创建了一个**局部变量**。局部变量属于创建它的函数，只有该函

数内部的语句才能访问该变量。“局部”的意思是变量只能在创建它的函数中“局部”地使用。

如果一个函数中的语句试图访问属于另一个函数的局部变量，就会发生错误，程序 5-4 展示了一个例子。

程序 5-4 (bad_local.py)

```
1  # main 函数定义
2  def main():
3      get_name()
4      print(f'你好, {name}。')    # 这会导致错误!
5
6  # get_name 函数的定义
7  def get_name():
8      name = input('输入你的姓名: ')
9
10 # 调用 main 函数
11 main()
```

该程序有两个函数：`main` 和 `get_name`。第 8 行将用户输入的值赋给 `name` 变量。由于该语句位于 `get_name` 函数内部，所以 `name` 变量是该函数的局部变量。这意味着 `name` 变量不能由 `get_name` 函数外部的语句访问。

`main` 函数在第 3 行调用 `get_name` 函数。然后，第 4 行的语句试图访问 `name` 变量。这会导致错误，因为 `name` 变量是 `get_name` 函数的局部变量，`main` 函数中的语句不能访问它。

作用域和局部变量

变量的**作用域**是程序中可以访问变量的那一部分。只有在变量作用域内的语句才能“看到”变量。局部变量的作用域是创建变量的那个函数。如程序 5-4 所示，函数外部的语句不能访问它。

此外，在局部变量被创建之前，函数内部的代码也不能访问它。在下例中，`print` 函数试图访问 `val` 变量，但该语句出现在 `val` 变量被创建之前，因而导致了错误。将赋值语句移到 `print` 语句之前的一行，即可解决这个错误。

```
def bad_function():
    print(f'值为{val}。')    #这会导致错误!
    val = 99
```

由于函数的局部变量对其他函数是隐藏的，所以其他函数可以有自己的同名局部变量。例

如，在程序 5-5 中，除了 `main` 函数外，这个程序还有另外两个函数：`texas` 和 `california`。这两个函数各有一个名为 `birds` 的局部变量。

程序 5-5 (birds.py)

```
1  # 这个程序演示了两个函数
2  # 可以拥有同名的局部变量。
3
4  def main():
5      # 调用 texas 函数
6      texas()
7      # 调用 california 函数
8      california()
9
10 # texas 函数的定义。它创建一个
11 # 名为 birds 的局部变量。
12 def texas():
13     birds = 5000
14     print(f'得克萨斯有{birds}只鸟。')
15
16 # california 函数的定义。它也创建一个
17 # 名为 birds 的局部变量。
18 def california():
19     birds = 8000
20     print(f'加利福尼亚有{birds}只鸟。')
21
22 # 调用 main 函数
23 main()
```

程序输出

```
得克萨斯有 5000 只鸟。
加利福尼亚有 8000 只鸟。
```

尽管程序有两个名为 `birds` 的变量，但由于分别在不同的函数中，所以每次只会看到其中一个，如图 5-12 所示。执行 `texas` 函数时，看到的是第 13 行创建的 `birds` 变量，而在执行 `california` 函数时，看到的是在第 19 行创建的 `birds` 变量。



图 5-12 每个函数都有自己的 `birds` 变量

检查点

- 5.10 什么是局部变量？如何限制对局部变量的访问？
- 5.11 什么是变量的作用域？
- 5.12 是否允许一个函数中的局部变量与另一个函数中的局部变量具有相同的名称？

5.5 向函数传递实参

概念：实参是在调用函数时向其传递的数据。形参是一个变量，它接收传入函数的实参。

 视频讲解：Passing Arguments to a Function

调用函数时，可能需要向其传递一个或多个数据。传给函数的数据称为**实参**。函数可以在计算或其他操作中使用传入的实参。

如果希望函数在被调用时接收实参，那么必须为函数定义一个或多个形参变量。**形参变量**通常简称为**形参**。^①作为一种特殊的变量，它会在函数被调用时接收传入的实参。以下函数定义了一个形参变量。

```
def show_double(number):
    result = number * 2
    print(result)
```

`show_double` 函数的作用是接收一个数字作为实参，并显示它乘以 2 的结果。注意函数头的圆括号中的 `number`，这就是形参变量的名称。调用函数时会向该变量赋值。程序 5-6 用一个完整的程序演示了这个函数。

程序 5-6 (pass_arg.py)

```
1  # 这个程序演示了向函数
2  # 传递实参的过程。
3
4  def main():
5      value = 5
6      show_double(value)
7
8  # show_double 函数接受一个实参，
9  # 并显示它乘以 2 的结果。
10 def show_double(number):
11     result = number * 2
12     print(result)
13
14 # 调用 main 函数
15 main()
```

程序输出

```
10
```

当这个程序运行时，`main` 函数在第 15 行被调用。在 `main` 函数内部，第 5 行创建一个名为 `value` 的局部变量，并为其赋值 5。然后，第 6 行调用 `show_double` 函数：

```
show_double(value)
```

^① 译注：一般仅在了为了区分时才会说形参或实参。大多数时候直接说“参数”即可。

注意圆括号中的 `value`。这意味着 `value` 将作为实参传入 `show_double` 函数，如图 5-13 所示。执行该语句会调用 `show_double` 函数。在函数内部，`number` 形参和 `value` 变量将引用同一个值，如图 5-14 所示。

```
def main():  
    value = 5  
    show_double(value)  
  
def show_double(number):  
    result = number * 2  
    print(result)
```

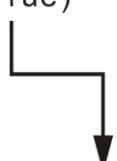


图 5-13 `value` 变量作为实参传递

```
def main():  
    value = 5  
    show_double(value)  
  
def show_double(number):  
    result = number * 2  
    print(result)
```

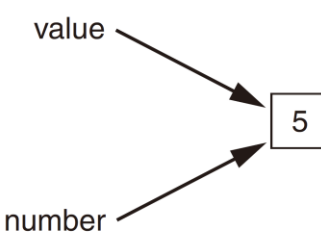


图 5-14 `value` 变量和 `number` 形参引用同一个值

来看一下 `show_double` 函数是如何工作的。首先要记住，会将作为实参传入的 `value` 的值赋给 `number` 形参变量。在这个程序中，`value` 的值为 5。

第 11 行将表达式 `number * 2` 的值赋给一个名为 `result` 的局部变量。由于 `number` 现在引用的值是 5，所以该语句会向 `result` 赋值 10。第 12 行显示 `result` 变量的值。

以下语句展示了如何调用 `show_double` 函数，并将一个字面值作为实参传递：

```
show_double(50)
```

该语句调用 `show_double` 函数，并将 50 赋给形参 `number`。因此，函数将打印 100。

5.5.1 形参变量的作用域

本章之前说过，变量的作用域是程序中可以访问该变量的那一部分。只有在变量作用域内的语句才能“看到”该变量。**形参变量的作用域**就是定义了该变量的那个函数的主体。函数内部的所有语句都能访问形参变量，但函数外部的语句不能访问。

聚光灯：向函数传递实参

你的朋友迈克尔经营着一家餐饮公司。他的菜谱中需要的一些液体配料是以杯为单位的。但是，在去杂货店购买这些配料时，发现它们以（液体）盎司为单位出售。他要求你写一个简单的程序，将杯换算为盎司。你设计了如下所示的算法。

1. 显示介绍性消息来解释程序的功能。
2. 获取杯数。
3. 将杯数换算为盎司数，并显示结果。

该算法列出了程序需要执行的最高层级的任务，它们构成了程序的 `main` 函数的基础。图 5-15 展示了程序的层次结构图。

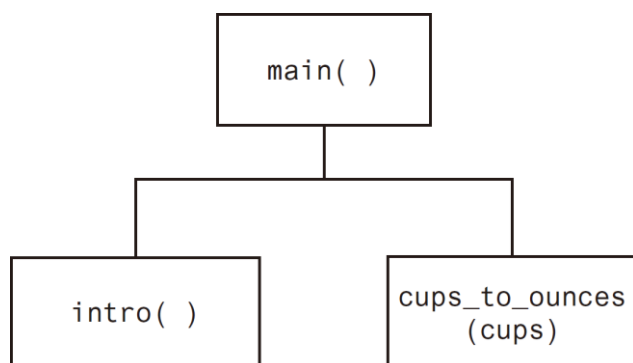


图 5-15 程序的层次结构图

如图所示，`main` 函数要调用另外两个函数，如下所示。

- `intro`：该函数在屏幕上显示一条消息来解释程序的作用。
- `cups_to_ounces`：该函数接受（液体）杯数作为实参，计算并显示等量的（液体）盎司数。

除了调用这些函数外，`main` 函数还要求用户输入杯数。该值将传给 `cups_to_ounces` 函数。程序代码如程序 5-7 所示。

程序 5-7 (cups_to_ounces.py)

```
1  # 这个程序将(液体)杯数换算为(液体)盎司数
2
3  def main():
4      # 显示介绍性消息
5      intro()
6      # 获取杯数
7      cups_needed = int(input('输入杯数: '))
8      # 将杯数换算为盎司数
9      cups_to_ounces(cups_needed)
10
11 # intro 函数显示介绍性消息
12 def intro():
13     print('这程序将以杯为度量单位')
14     print('的液体容量换算为盎司。')
15     print('所用的公式是: ')
16     print('    1 杯 = 8 盎司')
17     print()
18
19 # cups_to_ounces 函数接受杯数,
20 # 并显示相应的盎司数。
21 def cups_to_ounces(cups):
22     ounces = cups * 8
23     print(f'这相当于{ounces}盎司。')
24
25 # 调用 main 函数
26 main()
```

程序输出 (用户输入的内容加粗>)

这个程序将以杯为度量单位
的液体容量换算为盎司数。
所用的公式是:
1 杯 = 8 盎司

输入杯数: **4**
这相当于 32 盎司。

5.5.2 传递多个实参

我们经常需要写可以接受多个实参的函数。程序 5-8 展示了一个名为 `show_sum` 的函数，它接受两个实参。函数将两个实参相加并显示结果。

程序 5-8 (multiple_args.py)

```
1 # 这个程序定义了一个接受
2 # 两个实参的函数。
3
4 def main():
5     print('12 与 45 之和是')
6     show_sum(12, 45)
7
8 # show_sum 函数接受两个实参，
9 # 并显示两者之和。
10 def show_sum(num1, num2):
11     result = num1 + num2
12     print(result)
13
14 # 调用 main 函数
15 main()
```

程序输出

```
12 与 45 之和是
57
```

注意，`show_sum` 函数头的圆括号中出现了两个形参变量，即 `num1` 和 `num2`。我们通常把它称为**形参列表**或**参数列表**。另外还要注意，变量名之间以逗号分隔。

第 6 行的语句调用 `show_sum` 函数并传递两个实参：`12` 和 `45`。这些实参根据位置传递给函数中相应的形参变量。换言之，第一个实参传给第一个形参，第二个实参传给第二个形参。因此，该语句将 `12` 赋给 `num1`，将 `45` 赋给 `num2`，如图 5-16 所示。

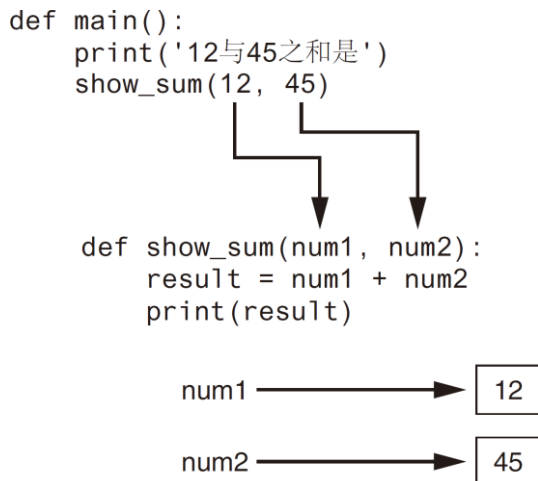


图 5-16 将两个实参传递给两个形参

假定调换一下函数调用中的实参顺序，如下所示：

```
show_sum(45, 12)
```

这会导致将 45 传给形参变量 num1，将 12 传给形参变量 num2。以下代码展示了另一个例子。这次将变量作为实参传递。

```
value1 = 2
value2 = 3
show_sum(value1, value2)
```

show_sum 函数执行时，形参 num1 的值为 2，num2 的值为 3。

程序 5-9 是另一个例子，它将两个字符串作为实参传递给一个函数。

程序 5-9 (string_args.py)

```
# 这个程序演示了将两个字符串
# 实参传递给函数。

def main():
    first_name = input('输入姓氏: ')
    last_name = input('输入名字: ')
    print('你的姓名反转后是: ')
    reverse_name(first_name, last_name)

def reverse_name(first, last):
    print(last, first)

# 调用 main 函数
main()
```

程序输出（用户输入的内容加粗）

```
输入姓氏: 张 
输入名字: 三丰 
你的姓名反转后是:
三丰 张
```

5.5.3 修改形参

将实参传入 Python 函数后，函数的形参变量将引用实参的值。但是，如果向形参变量重新赋值，那么不会对实参产生任何影响。程序 5-10 对此进行了演示。

程序 5-10 (change_me.py)

```

1  # 这个程序演示了更改了
2  # 形参变量的值后会发生什么。
3
4  def main():
5      value = 99
6      print(f'现在 value 是{value}。')
7      change_me(value)
8      print(f'返回 main 后, value 是{value}。')
9
10 def change_me(arg):
11     print('现在更改形参变量。')
12     arg = 0
13     print(f'现在形参变量的值是{arg}。')
14
15 # 调用 main 函数
16 main()

```

程序输出

```

现在 value 是 99。
现在更改形参变量。
现在形参变量的值是 0。
返回 main 后, value 是 99。

```

main 函数在第 5 行创建一个名为 value 的局部变量并赋值 99。第 6 行的语句显示'现在 value 是 99'，然后，第 7 行将 value 变量作为实参传递给 change_me 函数。这意味着在 change_me 函数中，形参 arg 也将引用值 99。如图 5-17 所示。

```

def main():
    value = 99
    print(f'现在value是{value}。')
    change_me(value)
    print(f'返回main后, value是{value}。')

def change_me(arg):
    print('现在更改形参变量。')
    arg = 0
    print(f'现在形参变量的值是{arg}。')

```

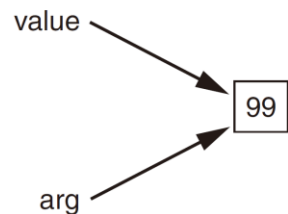


图 5-17 将 value 变量传递给 change_me 函数

在 change_me 函数内部，第 12 行将 0 赋给形参 arg。这个重新赋值更改了 arg 的值，但并不影响 main 中的 value 变量。如图 5-18 所示，这两个变量现在引用内存中不同的值。第 13 行的语句显示'现在形参变量的值是 0'，函数结束。

程序控制随后返回 `main` 函数，并执行第 8 行的语句。该语句显示‘返回 `main` 后，`value` 是 99’。这证明虽然形参变量 `arg` 在函数 `change_me` 中被更改，但不会影响实参变量（`main` 中的 `value` 变量）。

```
def main():  
    value = 99  
    print(f'现在value是{value}。')  
    change_me(value)  
    print(f'返回main后，value是{value}。')
```

```
def change_me(arg):  
    print('现在更改形参变量。')  
    arg = 0  
    print(f'现在形参变量的值是{arg}。')
```

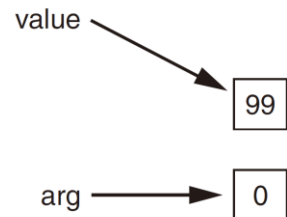


图 5-18 将 `value` 变量传递给 `change_me` 函数

在 Python 中，函数不能改变传递给它的实参的值。这通常称为**传值**^①，是函数与其他函数通信的一种方式。然而，这种通信只在一个方向上进行。换言之，调用函数可与被调用函数通信，但被调用函数不能使用实参与调用函数通信。本章稍后会解释如何写一个函数，通过返回值与调用它的那一部分的程序进行通信。

5.5.4 关键字参数

程序 5-8 和 5-9 演示了实参如何根据位置传给函数对应的形参变量。大多数编程语言都以这种方式匹配函数的实参和形参。但是，除了这种传统的传参方式，Python 语言还允许采用以下格式来写实参，指定每个实参应传递给哪个形参变量：

形参名 = 值

其中，“形参名”是形参变量的名称，“值”是传给该形参变量的值。采用这种语法写的实

^① 译注：作者在这里的说法不完全准确。大多数编程语言的“传值”都是指在内存中创建所传递的实参的一个“拷贝”。函数中的一切操作针对的都是这个拷贝，所以不会影响实参变量。但是，和这些语言不同，Python 的函数只支持“传对象引用”，不能选择传引用还是传值。这里所谓的“传对象引用”，其实是传值和传引用的一种结合。如果函数收到的是一个可变对象（例如字典或列表）的引用，那么可以修改对象的原始值，这相当于通过“传引用”来传递对象。如果函数收到的是一个不可变对象（例如数字、字符或元组）的引用，那么不能直接修改原始对象，这相当于通过“传值”来传递对象，本例只是展示了后面这种情况。

参称为**关键字实参**或**关键字参数**。

程序 5-11 演示了关键字实参。该程序使用一个名为 `show_interest` 的函数来显示一个银行账户在若干期间内获得的单利。函数的形参包括：`principal`（账户本金）、`rate`（每期利率）和 `period`（期数）。第 7 行在调用这个函数时，数据是作为关键字实参来传递的。

程序 5-11 (`keyword_args.py`)

```
1  # 这个程序演示了关键字参数
2
3  def main():
4      # 显示单利金额，使用 0.01 作为
5      # 每期利率，10 作为期数，
6      # 10000 作为本金。
7      show_interest(rate=0.01, periods=10, principal=10000.0)
8
9  # show_interest 函数显示在给定
10 # 了本金、每期利率和期数的前提
11 # 的单利金额。
12
13 def show_interest(principal, rate, periods):
14     interest = principal * rate * periods
15     print(f'单利金额为${interest:,.2f}。')
16
17 # 调用 main 函数
18 main()
```

程序输出

单利金额为\$1,000.00。

注意，第 7 行的关键字实参的顺序与第 13 行函数头中形参的顺序不一致。由于关键字实参具体指定了向哪个形参赋值，所以实参的顺序在函数调用中的位置并不重要。

程序 5-12 展示了另一个例子。它是程序 5-9 (`string_args.py`) 的一个变体，使用关键字实参来调用 `reverse_name` 函数。

程序 5-12 (`keyword_string_args.py`)

```
1  # 这个程序演示了将两个字符串
2  # 作为关键字实参传递给函数。
3
4  def main():
```

```
5     first_name = input('输入姓氏: ')
6     last_name = input('输入名字: ')
7     print('你的姓名反转后是: ')
8     reverse_name(last=last_name, first=first_name)
9
10    def reverse_name(first, last):
11        print(last, first)
12
13    # 调用 main 函数
14    main()
```

程序输出（用户输入的内容加粗）

```
输入姓氏: 张 
输入名字: 三丰 
你的姓名反转后是:
三丰 张
```

5.5.5 混合使用关键字实参和位置实参

在函数调用中，位置实参和关键字实参可以混用。但是，必须先写位置实参，然后才能写关键字实参，否则会报错。下例展示了如何同时使用位置实参和关键字实参来调用程序 5-10 中的 `show_interest` 函数：

```
show_interest(10000.0, rate=0.01, periods=10)
```

在这个语句中，第一个实参 `10000.0` 根据所在位置传给形参 `principal`。第二个和第三个实参则作为关键字实参传递。然而，以下函数调用会导致错误，因为在关键字实参后面出现了一个非关键字实参。

```
# 这会导致一个错误！
show_interest(1000.0, rate=0.01, 10)
```

5.5.6 仅关键字参数

如果要写一个函数定义，要求所有实参都必须作为关键字实参传递，那么可以使用 Python 的一种特殊的记号法：在函数参数列表的开头写一个星号，后跟一个逗号，如下所示。

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

注意，这个参数列表中的第一项是星号。这个星号不是参数，而是声明在它之后出现的所有参数都是**仅关键字参数**^①。仅关键字参数只接受关键字实参。在本例中，调用 `show_sum`

^① 译注：这里之所以不用“仅关键字形参”，是为了符合中文社区一般的说法。

函数时必须为 `a`, `b`, `c` 和 `d` 形参传递关键字实参。下面是一个示例函数调用。

```
show_sum(a=10, b=20, c=30, d=40)
```

在本例中，任何实参作为位置实参来传递都会报错。例如，以下语句会报错，因为 `a` 不是作为关键字实参来传递的。

```
show_sum(10, b=20, c=30, d=40) # 错误
```

星号不一定要放在参数列表的最开头。相反，它可以出现在参数列表的任何位置。但是，只有在星号之后的形参才是仅关键字参数。下面是一个例子。

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

在这个例子中，只有 `c` 和 `d` 才是仅关键字参数。

5.5.7 仅位置参数

上一节展示了如何在函数中声明仅关键字参数。除此之外，Python 还允许声明**仅位置参数**。调用函数时，仅位置参数将只接受位置实参。为了声明仅位置参数，需要在参数列表中插入一个正斜杠，它在之前的所有形参都是仅位置参数，如下例所示。

```
def show_sum(a, b, c, d, /):  
    print(a + b + c + d)
```

注意，参数列表的最后一项是正斜杠。出现在正斜杠之前的所有参数都是仅位置参数。在本例中，`a`, `b`, `c` 和 `d` 都是仅位置参数。下面是一个示例函数调用。

```
show_sum(10, 20, 30, 40)
```

根据位置，该函数调用将向形参 `a` 传递 `10`，向形参 `b` 传递 `20`，向形参 `c` 传递 `30`，向形参 `d` 传递 `40`。

将关键字实参传给仅位置参数会报错。例如，以下函数调用将导致错误，因为试图将一个关键字实参传递给形参 `d`。

```
show_sum(10, 20, 30, d=40) # 错误!
```

正斜杠可以出现在参数列表的任何位置，但只有在它之前的参数才会成为仅位置参数，如下例所示。

```
def show_sum(a, b, /, c, d):  
    print(a + b + c + d)
```

在这个例子中，只有 `a` 和 `b` 是仅位置参数。`c` 和 `d` 既可以接收位置参数，也可以接收关键字参数。但要记住，在 Python 中调用函数时，不能在关键字参数之后传递位置参数。所以，本例如果向 `c` 传递了关键字参数，那么也必须向 `d` 传递关键字参数。



注意：仅位置参数是自 Python 3.8 引入的。

5.5.8 默认实参

Python 允许为函数的形参提供默认实参。如果形参有默认实参，那么在调用函数时可以不显式地为其传递实参。下面是带有默认实参的一个示例函数定义。

```
def show_tax(price, tax_rate=0.07):
    tax = price * tax_rate
    print(f'税金为{tax}。')
```

这个函数定义为形参 `tax_rate` 提供了一个默认实参。注意，参数名后面有一个等号和一个值。等号后面的值就是默认实参。在本例中，形参 `tax_rate` 的默认值为 `0.07`。

调用 `show_tax` 函数时，必须为形参 `price` 传递一个实参。但是，由于形参 `tax_rate` 已经有了一个默认实参，所以可以忽略向它的赋值，如下例所示。

```
show_tax(100)
```

该语句调用 `show_tax` 函数，传递值 `100` 作为形参 `price` 的实参。由于没有为形参 `tax_rate` 显式传递一个实参，所以它的值默认为 `0.07`。如果希望形参 `tax_rate` 有一个不同的值，那么可以在调用函数时为其显式指定一个实参，如下所示。

```
show_tax(100, 0.08);
```

该语句调用 `show_tax` 函数，为形参 `price` 传递 `100`，为形参 `tax_rate` 传递 `0.08`（而不是使用默认的 `0.07`）。

在函数的参数列表中，必须先写不带默认实参的形参，然后才能写带默认实参的形参。例如，以下 `show_tax` 函数定义会报错。

```
# 这是一个无效的函数
def show_tax(price=10, tax_rate):
    tax = price * tax_rate
    print(f'税金为{tax}。')
```

在这个例子中，我们为形参 `price` 提供了默认实参，但是没有为形参 `tax_rate` 提供默认实参。由于 `tax_rate` 在 `price` 之后，所以也必须为它提供一个默认实参。

可以为函数的所有形参都提供默认实参，如下所示。

```
def show_tax(price=10, tax_rate=0.07):
    tax = price * tax_rate
    print(f'税金为{tax}。')
```

在这个函数定义中，形参 `price` 的默认实参是 `10`，形参 `tax_rate` 的默认实参是 `0.07`。由于函数的所有形参都有默认实参，因此可以直接调用该函数，而无需传递任何实参，如下

例所示。

```
show_tax()
```

执行该语句时，会向形参 `price` 传递 `10`，向 `tax_rate` 传递 `0.07`。

程序 5-13 使用一个函数在屏幕上显示星号。向函数传递的实参指定了星号的行列数。如果不传递实参，就使用函数的默认实参，即显示 1 行 10 列星号。

程序 5-13 (display_stars.py)

```
1  # 这个程序演示了默认实参
2  def main():
3      # 行列数使用默认实参（1 行 10 列）
4      display_stars()
5      print()
6
7      # 列数显式指定为 5，行数则使用默认实参（1 行）
8      display_stars(5)
9      print()
10
11     # 显式指定 7 列和 3 行
12     display_stars(7, 3)
13
14     # 该函数显示特定行列数的星号
15     def display_stars(cols=10, rows=1):
16         # 外层循环打印行，
17         # 内层循环打印列。
18         for row in range(rows):
19             for col in range(cols):
20                 print('*', end='')
21             print()
22
23     # 调用 main 函数
24     main()
```

程序输出

```
*****
*****
*****
*****
*****
```

下面总结了程序 5-13 的各个 `display_stars` 函数调用。

-
- 第 4 行的调用不传递任何实参，因此 `cols` 和 `rows` 形参都使用默认实参。
 - 第 8 行的调用传递了实参 5，该实参将传给 `cols` 形参，`rows` 形参则使用默认实参（1）。
 - 第 12 行的调用传递了实参 7 和 3。其中，实参 7 将传给 `cols` 形参，参数 3 则传给 `rows` 形参。

程序 5-13 在调用 `display_stars` 函数时使用的是位置实参。程序 5-14 演示了如何使用关键字实参来调用同一个函数。

程序 5-14 (`display_stars_keyword_args.py`)

```
1  # 这个程序演示了默认实参
2  # 和关键字实参。
3  def main():
4      # 显示 5 行 10 列星号
5      display_stars(rows=5, cols=10)
6      print()
7
8      # 列数显式指定为 7，行数则使用默认实参（1 行）
9      display_stars(cols=7)
10     print()
11
12     # 行数显式指定为 3，列数则使用默认实参（10 列）
13     display_stars(rows=3)
14
15 # 该函数显示特定行列数的星号
16 def display_stars(cols=10, rows=1):
17     # 外层循环打印行，
18     # 内层循环打印列。
19     for row in range(rows):
20         for col in range(cols):
21             print('*', end='')
22         print()
23
24 #调用 main 函数
25 main()
```

程序输出

```
*****
*****
*****
*****
*****

*****

*****
```

```
*****  
*****
```

下面总结了程序 5-14 的各个 `display_stars` 函数调用。

- 第 5 行的调用传递了关键字实参 `rows=5` 和 `cols=10`。这些值分别赋给 `cols` 和 `rows` 形参，不会使用它们的默认实参。
- 第 9 行的调用传递了关键字实参 `cols=7`。这个值代替了 `cols` 形参的默认实参，但 `rows` 形参使用了默认实参 1。
- 第 13 行的调用传递了关键字实参 `rows=3`。这个值代替了 `rows` 形参的默认实参，但 `cols` 形参使用了默认实参 10。

检查点

5.13 向被调用的函数传递的数据叫做什么？

5.14 在被调用的函数中接收数据的变量叫做什么？

5.15 什么是形参变量的作用域？

5.16 修改一个形参变量时，是否会影响传入函数的实参？

5.17 以下语句调用一个名为 `show_data` 的函数。哪个调用按位置传递实参，哪个调用传递关键字实参？

- `show_data(name='Kathryn', age=25)`
- `show_data('Kathryn', 25)`

5.6 全局变量和全局常量

概念：全局变量可由程序文件中的所有函数访问。

之前讲过，由一个函数中的赋值语句创建的变量是该函数的局部变量。换言之，只有这个函数内部的语句才能访问它。如果一个变量是由位于程序文件中的所有函数外部的赋值语句创建的，那么这个变量就称为全局变量。**全局变量**可由程序文件中的任何语句访问，包括任一函数中的语句。程序 5-15 展示了一个例子。

程序 5-15 (`global1.py`)

```
1 # 创建一个全局变量  
2 my_value = 10
```

```
3
4 # show_value 函数打印
5 # 全局变量的值。
6 def show_value():
7     print(my_value)
8
9 # 调用 show_value 函数
10 show_value()
```

程序输出

```
10
```

第 2 行的赋值语句创建一个名为 `my_value` 的变量。因其在任何函数的外部，所以是全局变量。执行 `show_value` 函数时，第 7 行的语句打印 `my_value` 所引用的值。

要在函数内部为全局变量赋值，需要采取一个额外的步骤，即用 `global` 关键字来声明一下全局变量，如程序 5-16 所示。

程序 5-16 (global2.py)

```
1 # 创建一个全局变量
2 number = 0
3
4 def main():
5     global number
6     number = int(input('输入一个数字: '))
7     show_number()
8
9 def show_number():
10    print(f'你输入的数字是{number}。')
11
12 # 调用 main 函数
13 main()
```

程序输出 (用户输入的内容加粗)

```
输入一个数字: 55 
你输入的数字是 55。
```

第 2 行的赋值语句创建名为 `number` 的全局变量。注意，在 `main` 函数中，第 5 行使用 `global` 关键字来声明 `number` 变量。该语句告诉 Python 解释器，`main` 函数打算向全局变量 `number` 赋值。这正是第 6 行发生的事情，用户输入的值被赋给 `number`。

大多数程序员都认为，应该限制全局变量的使用，或者根本不要使用全局变量，理由如下所示。

- 全局变量使调试变得困难。程序文件中的任何语句都可以更改全局变量的值。如果发现全局变量中存储了错误的值，那么必须追踪每一个访问全局变量的语句，以确定错误值的来源。在有数千行代码的一个程序中，这会令人抓狂。
- 使用全局变量的函数通常会依赖于这些变量。如果想在不同的程序中使用这种函数，那么很可能必须重新设计，使其不依赖于全局变量。
- 全局变量使程序难以理解。程序中的任何语句都可以修改全局变量。为了理解程序中使用了全局变量的任何部分，都必须理解程序中访问了该全局变量的其他所有部分。

大多数时候都应该创建局部变量，并将它们作为实参传递给需要访问它们的函数。

全局常量

虽然应该尽量避免使用全局变量，但完全可以在程序中使用全局常量。**全局常量**是一个全局名称，它引用了一个不可变的值。由于全局常量的值在程序执行过程中不能改变，所以不必担心使用全局变量时的许多潜在风险。

虽然 Python 语言不支持创建真正意义上的全局常量，但可以用全局变量来模拟。一个变量只要在函数中没有用 `global` 关键字声明成全局变量，就不能在这个函数中更改它的值。以下“聚光灯”小节演示了如何在 Python 中使用全局变量来模拟全局常量。

聚光灯：使用全局常量



玛丽莲在 Integrated Systems 公司工作，这家软件公司以福利而闻名。福利之一是向所有员工发放季度奖金。另一个福利是为每位员工提供的退休计划。公司代扣每位员工的税前工资和奖金的 5% 用于退休计划。Marilyn 希望写一个程序，计算公司一年来为员工退休账户代扣的金额。她希望程序能分别显示从税前工资和奖金代扣的金额。以下是这个程序的算法。

*获取员工的年度税前工资。
获取支付给员工的奖金金额。
计算并显示从税前工资代扣的退休金。
计算并显示从奖金代扣的退休金。*

程序 5-17 展示了完整代码。

程序 5-17 (retirement.py)

```
1 # 这是一个代表代扣率的
```

```
2 # 全局变量。
3 CONTRIBUTION_RATE = 0.05
4
5 def main():
6     gross_pay = float(input('输入税前工资金额: '))
7     bonus = float(input('输入奖金金额: '))
8     show_pay_contrib(gross_pay)
9     show_bonus_contrib(bonus)
10
11 # show_pay_contrib 函数接受年度
12 # 税前工资作为实参, 并显示要从中
13 # 代扣多少退休金。
14 def show_pay_contrib(gross):
15     contrib = gross * CONTRIBUTION_RATE
16     print(f'从税前工资中代扣的退休金: ${contrib:,.2f}。')
17
18 # show_bonus_contrib 函数接受
19 # 年度奖金作为实参, 并显示要
20 # 从中代扣多少退休金。
21 def show_bonus_contrib(bonus):
22     contrib = bonus * CONTRIBUTION_RATE
23     print(f'从奖金中代扣的退休金: ${contrib:,.2f}。')
24
25 # 调用 main 函数
26 main()
```

程序输出 (用户输入的内容加粗)

```
输入税前工资金额: 80000.00 
输入奖金金额: 20000.00 
从税前工资中代扣的退休金: $4,000.00。
从奖金中代扣的退休金: $1,000.00。
```

首先, 请注意第 3 行的声明:

```
CONTRIBUTION_RATE = 0.05
```

`CONTRIBUTION_RATE` 在本例中作为一个全局常量使用, 代表要将多大比例的员工收入存入退休账户。一般将常量名称写成全部大写字母的形式。这是为了提醒自己常量在程序中不能更改。

`CONTRIBUTION_RATE` 常量在第 15 行 (`show_pay_contrib` 函数) 和第 22 行 (`show_bonus_contrib` 函数) 的计算中使用。玛丽莲之所以决定用这个全局常量来表示 5% 的代扣率, 是出于两方面的考虑:

- 它使程序更易读。查看第 15 行和第 24 行的计算时, 所发现的事情会非常明显。
- 代扣率偶尔会发生变化。当这种情况发生时, 通过改变第 3 行的赋值语句, 可以很容

易地更新程序。

检查点

5.18 全局变量的作用域是什么？

5.19 给出最好不要在程序中使用全局变量的理由。

5.20 什么是全局常量？适合在程序中使用全局常量吗？

5.7 返回值的函数：生成随机数

概念：返回值的函数会将一个值返回给程序中调用它的那一部分。和其他大多数编程语言一样，Python 也提供了一个预先写好的函数库，用于执行各种常见任务。在这些库中，通常都包含一个生成随机数的函数。

本章第一部分讨论了 void 函数，即不返回值的函数。这种函数包含的语句用于执行一项特定的任务。需要函数执行特定的任务时，就调用函数。这会导致函数内部的语句执行。函数执行完毕后，程序的控制将返回到紧随函数调用之后的语句。

返回值的函数是一种特殊类型的函数。它和 void 函数相似的地方在于：

- 包含一组执行特定任务的语句
- 通过调用来执行函数

然而，当返回值的函数结束时，它会返回一个值给调用它的程序部分。从函数返回的值可以像其他值一样使用：可以赋给变量、在屏幕上显示、用于数学表达式（如果是数字）等。

5.7.1 标准库函数和 import 语句

和其他大多数编程语言一样，Python 带有一个已经写好的函数**标准库**。这些函数称为**库函数**，它们能执行许多常见任务，从而简化了程序员的工作。事实上，之前已经使用了几个 Python 库函数，包括 print, input 和 range 等。Python 还提供了其他许多库函数。尽管本书不会覆盖全部库函数，但会重点讨论执行一些基本操作的库函数。

Python 的一些库函数内置于 Python 解释器中。要在程序中使用某个内置函数，直接调用它即可。之前用过的 print, input, range 和其他一些函数就属于这种情况。然而，标准库中的许多函数都存储在称为**模块**的文件中。这些模块在安装 Python 时被复制到计算机上。模块的作用是对标准库函数进行组织。例如，执行数学运算的所有函数都存储在一个模块中，负责文件处理的所有函数存储在另一个模块中，等等。

为了调用存储在模块中的函数，必须在程序顶部写一个 `import` 语句。`import` 语句告诉解释器要导入什么模块。例如，`math` 是 Python 标准模块之一，其中包含各种处理浮点数的数学函数。如果程序要使用 `math` 模块中的任何函数，那么应该在程序顶部写以下 `import` 语句。

```
import math
```

该语句指示解释器将 `math` 模块的内容加载到内存，并使当前程序可以使用 `math` 模块中的所有函数。

由于看不到库函数内部是如何工作的，所以许多程序员都说它们是**黑盒**。我们平常使用黑盒一词来描述接收输入、使用输入来执行某些操作（不清楚细节）并生成输出的机制。图 5-19 说明了这一概念。



图 5-19 作为黑盒的库函数

为了理解返回值的函数是如何工作的，我们首先来看看生成随机数的标准库函数，以及用这种函数来写的一些有趣的程序。然后，你将学习如何写自己的返回值的函数，以及如何创建自己的模块。本章最后一节将回到库函数的主题，并介绍 Python 标准库的其他几个有用的函数。

5.7.2 生成随机数

许多编程任务都要用到随机数，下面列举了一些例子。

- 在游戏中会大量用到随机数。例如，掷骰子游戏用随机数来表示骰子的点数。从洗好的一副扑克牌中抽牌的程序用随机数来表示牌的点数。
- 随机数在模拟程序中非常有用。在某些模拟程序中，计算机必须随机决定一个人、动物、昆虫或其他生物的行为。可以用随机数构造公式，以决定程序中发生的各种行为和事件。
- 随机数在必须随机选择数据进行分析的统计程序中非常有用。
- 在计算机安全领域，经常要使用随机数加密敏感数据。

Python 提供了几个用于处理随机数的库函数，它们存储在标准库的 `random` 模块中。要使

用其中任何一个函数，首先都必须在程序顶部添加一个 `import` 语句。

```
import random
```

该语句指示解释器将 `random` 模块的内容加载到内存。随后，程序中的任何地方都能使用 `random` 模块中的函数。^①

我们讨论的第一个随机数生成函数是 `randint`。由于 `randint` 函数在 `random` 模块中，所以需要在程序中使用**点记号法**来引用它。采用点记号法，函数的全称是 `random.randint`。点号左边是模块名，右边是函数名。

以下语句展示了如何调用 `randint` 函数：

```
number = random.randint(1, 100)
```

在这个语句中，等号右侧的 `random.randint(1, 100)` 是对 `randint` 函数的一次调用。注意圆括号内有两个实参：`1` 和 `100`。它们告诉函数生成 `1~100` 的一个随机整数。图 5-20 对语句的这一部分进行了说明。

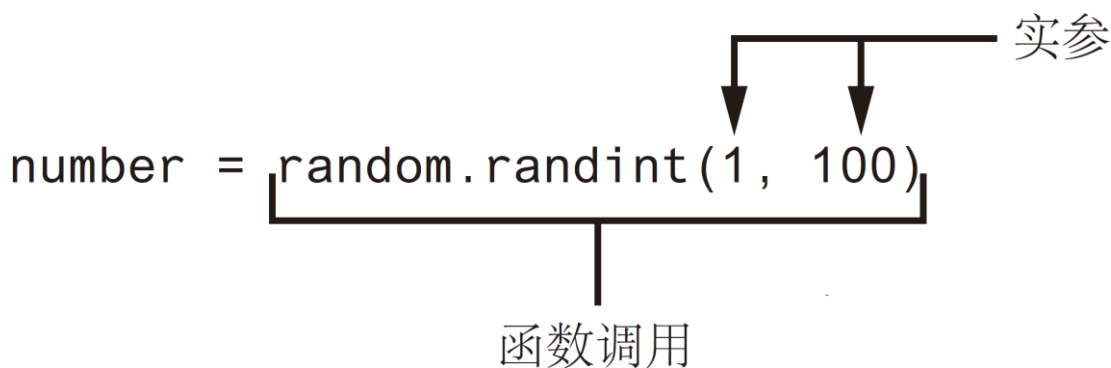
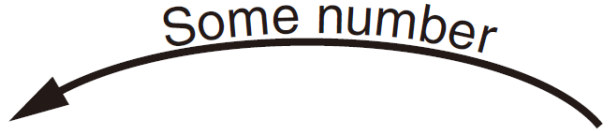


图 5-20 调用 `random` 模块中的一个函数

注意，`randint` 函数调用出现在 `=` 操作符右侧。调用这个函数时，它生成 `1~100` 范围内的一个随机数，然后返回该随机数。返回的数字会被赋给 `=` 操作符左侧的 `number` 变量，如图 5-21 所示。

^① 在 Python 中写 `import` 语句有几种方式，每种方式略有区别。但是，许多 Python 程序员都认为，导入模块的首选方式是本书所展示的方式。

Some number



```
number = random.randint(1, 100)
```

A random number in the range of
1 through 100 will be assigned to
the number variable.

图 5-21 random 函数返回一个值

图中译文：

Some number: 某个数字

A random number in the range of 1 through 100 will be assigned to the number variable.: 1~100 的一个随机数会被赋给 number 变量。

程序 5-18 展示了一个使用 randint 函数的完整程序。第 7 行的语句生成 1~10 的随机数，并将其赋给 number 变量。虽然示例输出生成的是数字 7，但这数字是任意的。事实上，每次运行这个程序，都可能生成 1~10 的任何数字。

程序 5-18 (random_numbers.py)

```
1  # 这个程序显示一个范围在 1 到 10
2  # 之间的随机数。
3  import random
4
5  def main():
6      # 获取随机数
7      number = random.randint(1, 10)
8
9      # 显示随机数
10     print(f'随机数为{number}。')
11
12 # 调用 main 函数
13 main()
```

程序输出

随机数为 7。

程序 5-19 展示了另一个例子，它使用了一个迭代 5 次的 `for` 循环。在循环内部，第 8 行的语句调用 `randint` 函数来生成 1~100 的随机数。

程序 5-19 (random_numbers2.py)

```
1  # 这个程序显示 5 个范围在 1 到 100
2  # 之间的随机数。
3  import random
4
5  def main():
6      for count in range(5):
7          # 获取随机数
8          number = random.randint(1, 100)
9
10         # 显示显示随机数
11         print(number)
12
13 # 调用 main 函数
14 main()
```

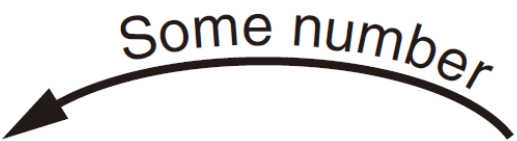
程序输出

```
89
7
16
41
12
```

程序 5-18 和程序 5-19 都调用了 `randint` 函数，并将它的返回值赋给 `number` 变量。如果只是想显示一个随机数，那么不需要将随机数赋给变量。可以直接将 `random` 函数的返回值发送给 `print` 函数，如下所示：

```
print(random.randint(1, 10))
```

执行这个语句会调用 `randint` 函数。该函数生成 1~10 的随机数，并将其返回给 `print` 函数。随后，`print` 函数将显示这个值。图 5-22 对此进行了说明。



```
print(random.randint(1, 10))
```

A random number in the range of
1 through 10 will be displayed.

图 5-22 显示随机数

图中译文：

Some number: 某个数字

A random number in the range of

1 through 10 will be displayed.: 显示 1~10 的某个数。

程序 5-20 展示了如何简化程序 5-19。它同样显示 5 个随机数，但是没有使用变量来保存这些随机数。在第 7 行，`randint` 函数的返回值被直接发送给 `print` 函数。

程序 5-20 (`random_numbers3.py`)

```
1 # 这个程序显示 5 个范围在 1 到 100
2 # 之间的随机数。
3 import random
4
5 def main():
6     for count in range(5):
7         print(random.randint(1, 100))
8
9 # 调用 main 函数
10 main()
```

程序输出

89
7
16
41
12

5.7.3 从 f 字符串中调用函数

函数调用可以作为 f 字符串的占位符，如下例所示。

```
print(f'number 为{random.randint(1, 100)}.')
```

该语句将显示如下所示的消息（其中的 58 为随机值）：

```
number 为 58。
```

对函数调用的结果进行格式化时，f 字符串特别有用。例如，以下语句在一个 10 字符宽的域内打印一个居中对齐的随机数。

```
print(f'{random.randint(0, 1000):^10d}')
```

5.7.4 在交互模式下尝试使用随机数

为了理解向 `randint` 函数传递不同实参时的结果，可以考虑在交互模式下进行实验。以下交互会话对此进行了演示（为方便引用，我们添加了行号）。

```
1 >>> import random   
2 >>> random.randint(1, 10)   
3 5  
4 >>> random.randint(1, 100)   
5 98  
6 >>> random.randint(100, 200)   
7 181  
8 >>>
```

来仔细看看交互会话中的每一行。

- 第 1 行的语句导入 `random` 模块（交互模式下也必须写恰当的 `import` 语句）。
- 第 2 行的语句调用 `randint` 函数，将 1 和 10 作为实参。结果，函数返回 1~10 的随机数。第 3 行显示了这个数字。
- 第 4 行的语句调用 `randint` 函数，将 1 和 100 作为实参。结果，函数返回 1~100 的随机数。第 5 行显示了这个数字。
- 第 6 行的语句调用 `randint` 函数，将 100 和 200 作为实参。结果，函数返回 100~200 的随机数。第 7 行显示了这个数字。

聚光灯：使用随机数



木村博士教授统计学入门课程，他要求你写一个程序，让他在课堂上模拟掷两粒骰子。程序应随机生成两个 1~6 的数字并显示出来。通过与木村博士面谈，你了解到他想用这个程序来模拟多次掷骰子的结果，每掷一次就询问是否继续。下面是程序的伪代码。

当 (while) 用户想要掷骰子时：
 显示 1~6 的随机数

显示另一个1~6的随机数
询问用户是否要再掷一次

为此，可以写一个 `while` 循环来模拟掷两粒骰子的过程，然后询问用户是否要再掷一次。只要用户回答“y”代表“是”，循环就会重复。程序 5-21 展示了完整代码。

程序 5-21 (dice.py)

```
1  # 这个程序模拟掷两粒骰子
2  import random
3
4  # 代表最小和最大随机数的常量
5  MIN = 1
6  MAX = 6
7
8  def main():
9      # 创建循环控制变量
10     again = 'y'
11
12     # 模拟掷骰子
13     while again == 'y' or again == 'Y':
14         print('正在掷骰...')
15         print('它们的点数为: ')
16         print(random.randint(MIN, MAX))
17         print(random.randint(MIN, MAX))
18
19         # 再掷一次吗?
20         again = input('再掷一次吗? (y = 是): ')
21
22 # 调用 main 函数
23 main()
```

程序输出 (用户输入的内容加粗)

```
正在掷骰...
它们的点数为:
3
1
再掷一次吗? (y = 是): y 
正在掷骰...
它们的点数为:
1
1
再掷一次吗? (y = 是): y 
正在掷骰...
它们的点数为:
5
6
```

再掷一次吗? (y = 是): n

因为 `randint` 函数返回一个整数值，所以在能接收一个整数值的地方都能调用该函数。在之前的例子中，我们展示了如何将函数的返回值赋给变量，以及如何将函数的返回值发送给 `print` 函数。为了加深这方面的理解，以下语句在数学表达式中使用了 `randint` 函数。

```
x = random.randint(1, 10) * 2
```

该语句生成 1~10 的随机数，将其乘以 2，结果是一个 2~20 的随机偶数。这个值会赋给 `x` 变量。还可以使用 `if` 语句测试来函数的返回值，后面的“聚光灯”小节对此进行了演示。

聚光灯：使用随机数来表示其他值



木村博士对你为他写的掷骰子模拟器非常满意，他要求再写一个程序来模拟抛十次硬币。每抛一次，程序都应随机显示“正面”或“背面”。

你决定通过随机生成一个 1~2 的整数来模拟抛硬币。你需要一个 `if` 语句，随机数为 1 就显示“正面”，否则显示“背面”。下面是伪代码：

重复 10 次：

 如果 (`if`) 在 1 到 2 范围内的随机数等于 1，那么：

 显示“正面”

 否则 (`else`):

 显示“背面”

由于事先知道程序需要模拟抛 10 次硬币，所以你决定使用一个 `for` 循环，如程序 5-22 所示。

程序 5-22 (coin_toss.py)

```
1  # 这个程序模拟抛 10 次硬币
2  import random
3
4  # 常量
5  HEADS = 1
6  TAILS = 2
7  TOSSES = 10
8
9  def main():
10     for toss in range(TOSSES):
11         # 模拟抛硬币
12         if random.randint(HEADS, TAILS) == HEADS:
13             print('正面')
14         else:
15             print('背面')
```

```
16
17 # 调用 main 函数
18 main()
```

程序输出

```
背面
背面
正面
背面
背面
背面
正面
正面
正面
背面
```

5.7.5 randrange、random 和 uniform 函数

标准库的 `random` 模块包含了许多处理随机数的函数。除了 `randint` 函数之外，`randrange`，`random` 和 `uniform` 函数也很有用（要使用这些函数中的任何一个，都需要在程序的顶部写 `import random`）。

如果还记得如何使用 `range` 函数（第 4 章），那么很容易理解 `randrange` 函数。`randrange` 函数的参数与 `range` 函数相同。不同之处在于，`randrange` 函数返回的不是一个数值列表。相反，它从一系列值中返回一个随机值。例如，以下语句向变量 `number` 随机赋值 0~9：

```
number = random.randrange(10)
```

本例传递的实参是 `10`，它代表数值序列的结束限制（`end limit`），即序列中的值必须在这个限制以下。函数将返回从 0~（但不包括）结束极限的一个随机数。以下语句同时指定了数值序列的起始值和结束限制：

```
number = random.randrange(5,10)
```

执行这个语句会将 5~9 的随机数赋给 `number`。以下语句同时指定了起始值、结束极限和步长：

```
number = random.randrange(0, 101, 10)
```

执行这个语句，`randrange` 函数会从以下值序列中随机返回一个值。

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

`randint` 和 `randrange` 函数返回的都是整数，`random` 函数返回的则是随机浮点数。`random` 函数不接受任何实参，调用时会返回一个 0.0~1.0（但不包括 1.0）的随机浮点数，如下例所示。

```
number = random.random()
```

`uniform` 函数也返回一个随机浮点数，但允许指定数值范围，如下例所示。

```
number=random.uniform(1.0, 10.0)
```

执行这个语句，`uniform` 函数将返回 `1.0~10.0` 的一个随机浮点数，并将其赋给 `number` 变量。

5.7.6 随机数种子

`random` 模块中的函数生成的并不是真正意义上的随机数。虽然经常说它们是随机数，但它们实际是通过公式计算出来的**伪随机数**。生成随机数的公式必须用一个称为**种子**的值来初始化。这个种子值用于计算要从序列中返回的下一个随机数。默认情况下，`random` 模块在导入时会从计算机的内部时钟获取系统时间，并将其用作种子值。系统时间其实是代表当前日期和时间的一个整数，精确到百分之一秒。

如果始终使用同一个种子值，随机数函数将始终生成相同的伪随机数序列。由于系统时间每百分之一秒都会发生变化，所以每次导入 `random` 模块时，都会生成不同的随机数序列。所以，基本上能模拟出“随机”效果。

在某些应用中，你可能希望始终生成相同的随机数序列。在这种情况下，可以调用 `random.seed` 函数来指定一个种子值，如下例所示。

```
random.seed(10)
```

在这这个例子中，`10` 被指定为种子值。如果程序每次运行时都调用 `random.seed` 函数，并将同一个值作为实参，那么它将始终生成相同的伪随机数序列。为了演示这一点，请看下面的交互会话。(为了便于参考，我们添加了行号)。

```
1  >>> import random 
2  >>> random.seed(10) 
3  >>> random.randint(1, 100) 
4  58
5  >>> random.randint(1, 100) 
6  43
7  >>> random.randint(1, 100) 
8  58
9  >>> random.randint(1, 100) 
10 21
11 >>>
```

第 1 行导入 `random` 模块。第 2 行调用 `random.seed` 函数，传递 `10` 作为种子值。第 3 行、第 5 行、第 7 行和第 9 行分别调用 `random.randint` 函数，获得 `1~100` 之间的伪随机数。本例生成的随机数是 `58`，`43`，`58` 和 `21`。下面，如果开始一个新的交互会话，并重复这些语句，那么会得到同一序列的伪随机数，如下所示。注意，不需要关闭 `IDLE Shell` 窗口，直

接重新输入即可。

```
1 >>> import random   
2 >>> random.seed(10)   
3 >>> random.randint(1, 100)   
4 58  
5 >>> random.randint(1, 100)   
6 43  
7 >>> random.randint(1, 100)   
8 58  
9 >>> random.randint(1, 100)   
10 21  
11 >>>
```

检查点

5.21 返回值的函数与 void 函数有什么不同？

5.22 什么是库函数？

5.23 为什么说库函数是“黑盒”？

5.24 以下语句会做什么？

```
x = random.randint(1, 100)
```

5.25 以下语句会做什么？

```
print(random.randint(1, 20))
```

5.26 以下语句会做什么？

```
print(random.randrange(10, 20))
```

5.27 以下语句会做什么？

```
print(random.random())
```

5.28 以下语句会做什么？


```
print(random.uniform(0.1, 0.5))
```

5.29 导入 random 模块时，它会使用什么作为随机数生成的种子值？

5.30 总是使用同一个种子值来生成随机数会发生什么？

5.8 自定义返回值的函数

概念：返回值的函数有一个 return 语句，将一个值返回给调用它的程序部分。

 视频讲解：Writing a Value-Returning Function

在写返回值的函数时，采用的方式与 void 函数相同，唯一的区别是必须有一个 return 语句。下面是 Python 中返回值的函数的常规格式：

```
def function_name():  
    语句  
    语句  
    ...  
    return 表达式
```

函数主体中必须有一个 return 语句，它采用以下形式：

```
return 表达式
```

关键字 return 后面的表达式的值会返回程序中调用该函数的那一部分。它可以是任何值、变量或者任何有一个值的表达式（例如数学表达式）。

下面是一个简单的返回值的函数：

```
def sum(num1, num2):  
    result = num 1 + num 2  
    return result
```

图 5-23 展示了该函数的各个组成部分。

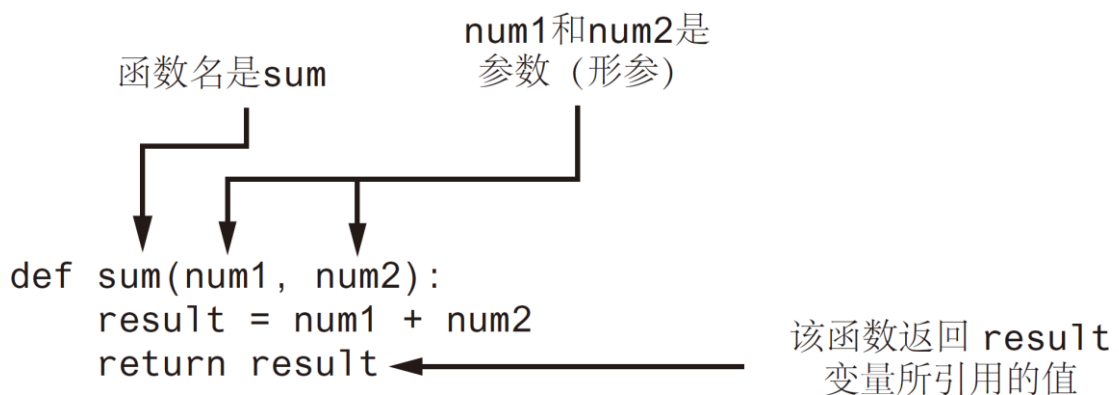


图 5-23 sum 函数的各个组成部分

这个函数的作用是接收两个整数值作为实参并返回两者之和。下面来仔细看看它是如何工作的。函数块（主体）中的第一个语句将 num1 + num2 的值赋给 result 变量，然后执行 return 语句来结束函数的执行，并将 result 变量引用的值返回给程序中调用函数的那一部分。程序 5-23 演示了这个函数的实际使用。

程序 5-23 (total_ages.py)

```
1  # 这个程序使用了函数的返回值
2
3  def main():
4      # 获取用户的年龄
5      first_age = int(input('你多少岁? '))
6
7      # 获取用户最好朋友的年龄
8      second_age = int(input("你的最好的朋友多少岁? "))
9
10     # 显示两个年龄相加的结果
11     total = sum(first_age, second_age)
12
13     # 显示求和结果
14     print(f'你们两个加起来有{total}岁。')
15
16     # sum 函数接收两个数值实参，
17     # 并返回两个实参之和。
18     def sum(num1, num2):
19         result = num1 + num2
20         return result
21
22     # 调用 main 函数
23     main()
```

程序输出 (用户输入的内容加粗)

```
你多少岁? 22 
你的最好的朋友多少岁? 24 
你们两个加起来有 46 岁。
```

在 `main` 函数中，程序从用户处获取两个值并将它们分别存储到变量 `first_age` 和 `second_age` 中。第 11 行的语句调用 `sum` 函数，将 `first_age` 和 `second_age` 作为实参传递。从 `sum` 函数返回的值被赋给 `total` 变量。在本例中，函数将返回 46。图 5-24 展示了向函数传递实参并从函数返回值的过程。

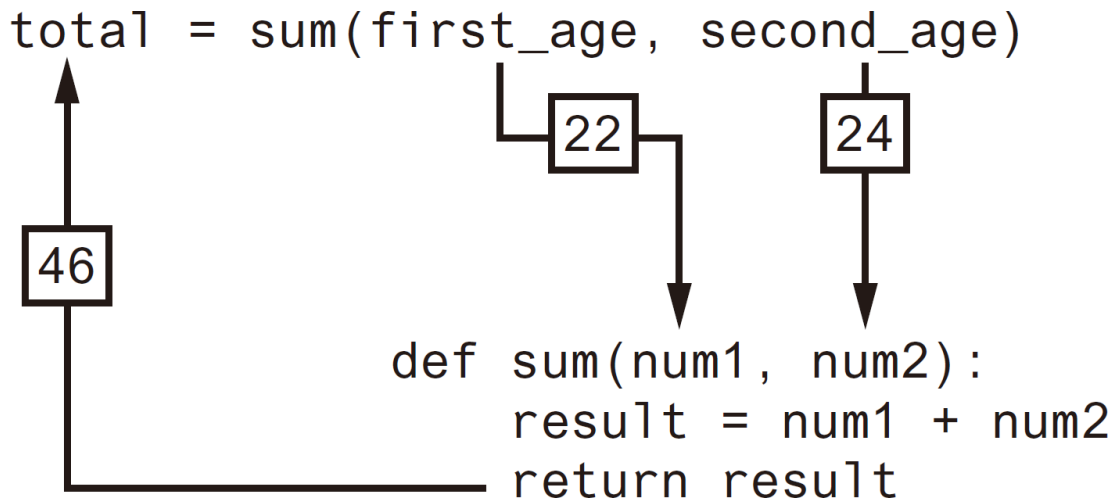


图 5-24 向 `sum` 函数传递实参并从中返回一个值

5.8.1 更高效地利用 `return` 语句

再来看一下程序 5-23 中的 `sum` 函数。

```
def sum(num1, num2):  
    result = num1 + num2  
    return result
```

这个函数发生了两件事情：（1）将表达式 `num1 + num2` 的值赋给 `result` 变量；（2）返回 `result` 变量的值。虽然函数完成了它的既定任务，但还可以简化。由于 `return` 语句能直接返回表达式的求值结果，所以完全可以去掉 `result` 变量，将两个步骤合并成一个，像下面这样重写函数。

```
def sum(num1, num2):  
    return num1 + num2
```

函数的这个版本没有用一个变量来存储 `num1 + num2` 的求值结果。相反，它利用了 `return` 语句能返回表达式求值结果这一事实。函数的这个版本和上一个版本做了相同的事情，但只用了一步。

5.8.2 使用返回值的函数

返回值的函数具有许多与 `void` 函数相同的好处。它们都简化了代码，减少了重复，提高了测试代码的能力，提高了开发速度，并方便团队协作。

由于返回值的函数会返回一个值，所以在特定情况下非常有用。例如，可以使用返回值的函数来提示用户输入，并返回用户输入的值。假定现在要设计一个程序来计算商品售价。为此，程序需要从用户处获得商品的正常价格。下面是为此目的定义的一个函数。

```
def get_regular_price():
    price = float(input("输入商品的正常价格: "))
    return price
```

然后就可以在程序的其他地方调用该函数，如下所示。

```
# 获取商品的正常价格
reg_price = get_regular_price()
```

执行这个语句时，会调用 `get_regular_price` 函数，后者从用户处获取一个值并返回该值。返回值会被赋给 `reg_price` 变量。

还可以利用函数来简化复杂的数学表达式。例如，计算商品售价似乎是一项简单的任务：计算折扣并从正常价格中减去折扣。但是，在程序中执行这种计算的语句并没有那么简单，如下例所示（假设 `DISCOUNT_PERCENTAGE` 是程序定义的一个全局常量，代表折扣比例）。

```
sale_price = reg_price - (reg_price * DISCOUNT_PERCENTAGE)
```

这个语句不能一下子就明白，因为它执行了多个步骤：计算折扣金额，从 `reg_price` 中减去这个金额，然后将结果赋给 `sale_price`。为了简化它，可以将数学表达式的一部分分离出来，并将其置于一个函数中。下面是一个名为 `discount` 的函数，它接收商品的正常价格作为实参，并返回折扣金额：

```
def discount(price):
    return price * DISCOUNT_PERCENTAGE
```

然后，可以在计算中调用该函数：

```
sale_price = reg_price - discount(reg_price)
```

这个语句比之前显示的语句更容易阅读，它更清楚地表明售价等于正常价格减折扣。程序 5-24 是使用上述函数来计算售价的一个完整程序。

程序 5-24 (sale_price.py)

```
1  # 这个程序计算一件
2  # 零售商品的售价。
3
4  # DISCOUNT_PERCENTAGE 是代表
5  # 折扣比例的一个全局常量。
6  DISCOUNT_PERCENTAGE = 0.20
7
8  # main 函数
9  def main():
10     # 获取商品的正常价格
11     reg_price = get_regular_price()
12
```

```
13     # 计算售价（折后价）
14     sale_price = reg_price - discount(reg_price)
15
16     # 显示售价
17     print(f'售价是${sale_price:,.2f}。')
18
19 # get_regular_price 函数提示
20 # 用户输入商品的正常价格，
21 # 并返回该值。
22 def get_regular_price():
23     price = float(input("输入商品的正常价格: "))
24     return price
25
26 # discount 函数接收商品的正常价格作为
27 # 实参，并返回折扣金额。折扣比例由
28 # DISCOUNT_PERCENTAGE 指定。
29 def discount(price):
30     return price * DISCOUNT_PERCENTAGE
31
32 # 调用 main 函数
33 main()
```

程序输出（用户输入的内容加粗）

```
输入商品的正常价格: 100.00 
售价是$80.00。
```

5.8.3 使用 IPO 图

IPO 图是一种简单而有效的工具，程序员有时会用它来设计和记录函数。IPO 是输入（Input）、处理（Processing）和输出（Output）的缩写，**IPO 图**描述了函数的输入、处理和输出。这些项通常以列的形式呈现。其中，输入列描述了作为实参传递给函数的数据，处理列描述了函数执行的处理，输出列则描述了函数返回的数据。例如，图 5-25 展示了程序 5-24 中的 `get_regular_price` 函数和 `discount` 函数的 IPO 图。

get_regular_price函数		
输入	处理	输出
无	提示用户输入一种商品的正常价格	商品的正常价格

discount函数		
输入	处理	输出
一种商品的正常价格	正常价格乘以代表折扣比例的全局变量DISCOUNT_PERCENTAGE, 得到商品的折扣金额	商品的折扣金额

图 5-25 get_regular_price 函数和 discount 函数的 IPO 图

注意，IPO 图只提供了关于函数输入、处理和输出的简要说明，而没有显示函数的具体步骤。但在许多情况下，IPO 图提供的信息足以用它来代替流程图。至于是使用 IPO 图、流程图还是两者兼而有之，通常取决于程序员的个人偏好。

聚光灯：使用函数来模块化程序

哈尔拥有一家名为 Make Your Own Music 的公司。公司主要销售吉他、鼓、班卓琴、合成器和其他许多种类的乐器。销售人员严格按照佣金制工作。月底，每个销售人员的佣金

(提成)率根据表 5-1 来计算。

表 5-1 销售佣金率

月销售额	佣金率
低于\$10000	10%
\$10000~14999	12%
\$15000~17999	14%
\$18000~21999	16%
\$22000 或更高	18%

例如，月销售额为 16000 美元的销售人员将获得 14%的佣金 (2240 美元)。月销售额为 18000 美元的销售人员将获得 16%的佣金 (2880 美元)。月销售额为 30000 美元的销售人员将获得 18%的佣金 (5400 美元)。

由于员工每月领取一次工资，公司允许每位员工每月最多预支 2000 美元。计算销售佣金时，从佣金中减去每位员工的预支工资。如果任何销售人员的佣金少于预支金额，他们必须向公司偿还差额。为了计算销售人员每个月的工资，哈尔使用了以下公式：

$$\text{工资} = \text{销售额} \times \text{佣金率} - \text{预支工资}$$

哈尔要求你写一个程序来完成工资计算。程序采用以下常规算法。

1. 获取销售人员的月销售额。
2. 获取预付工资额。
3. 根据月销售额来确定佣金率。
4. 使用前面的公式计算销售人员的工资。如果金额为负，表明销售人员必须偿还这个差额。

程序 5-25 展示了用几个函数来编写的代码。注意，这里没有一次性列出全部程序，而是先列出 main 函数，然后分别讨论它调用的每个函数。

程序 5-25 (commission_rate.py) — main 函数

```

1 # 这个程序计算 Make Your Own Music
2 # 公司销售人员的月薪
3 def main():
4     # 获取销售额
5     sales = get_sales()
6
7     # 获取预支工资额
8     advanced_pay = get_advanced_pay()
9
10    # 确定佣金率
11    comm_rate = determine_comm_rate(sales)
12
13    # 计算工资
14    pay = sales * comm_rate - advanced_pay
15
16    # 显示工资额
17    print(f'工资为${pay:,.2f}。')
18
19    # 判断工资是否为负
20    if pay < 0:
21        print('销售人员必须偿还')
22        print('公司。')
23

```

第 5 行调用 `get_sales` 函数，该函数从用户处获取销售额并返回该值。函数返回的值被赋给 `sales` 变量。第 8 行调用 `get_advanced_pay` 函数，该函数从用户处获取预支金额并返回该值。函数返回的值被赋给 `advanced_pay` 变量。

第 11 行调用 `determine_comm_rate` 函数，将销售额变量 `sales` 作为实参传递。该函数返回与这一档销售额对应的佣金率，结果被赋给 `comm_rate` 变量。第 14 行计算工资额，第 17 行显示该工资额。第 20 行~第 22 行的 `if` 语句判断工资是否为负，如果为负，那么会显示一条消息，说明销售人员必须向公司偿还这个差额。接下来是函数 `get_sales` 的定义。

程序 5-25 (`commission_rate.py`) — `get_sales` 函数

```

24 # get_sales 函数从用户处获取销售人员的
25 # 月销售额，并返回该金额。
26 def get_sales():
27     # 获取月销售额
28     monthly_sales = float(input('输入月销售额: '))
29
30     # 返回输入的金额
31     return monthly_sales
32

```

`get_sales` 函数的作用是提示用户输入销售人员的销售额，并返回该销售额。第 28 行提示用户输入销售额，并将用户的输入存储在变量 `monthly_sales` 中。第 31 行返回 `monthly_sales` 变量中的金额。接下来是函数 `get_advanced_pay` 的定义。

程序 5-25 (commission_rate.py) — get_advanced_pay 函数

```
33 # get_advanced_pay 函数获取
34 # 销售人员预支的工资金额,
35 # 并返回该金额。
36 def get_advanced_pay():
37     # 获取预支工资金额
38     print('输入预支工资金额, 如果')
39     print('没有预支工资则输入 0。')
40     advanced = float(input('预支工资: '))
41
42     # 返回输入的金额
43     return advanced
44
```

get_advanced_pay 函数的作用是提示用户输入销售人员的预支工资金额并返回该金额。第 38 行和第 39 行告诉用户输入预支工资的金额（没有预支则输入 0）。第 40 行获取用户输入的值并将其存储到 advanced 变量中。第 43 行返回变量中的金额。接下来是 determine_comm_rate 函数的定义。

程序 5-25 (commission_rate.py) — determine_comm_rate 函数

```
45 # determine_comm_rate 函数接收
46 # 销售额作为实参, 并返回相应的
47 # 佣金率。
48 def determine_comm_rate(sales):
49     # 确定佣金率
50     if sales < 10000.00:
51         rate = 0.10
52     elif sales >= 10000 and sales <= 14999.99:
53         rate = 0.12
54     elif sales >= 15000 and sales <= 17999.99:
55         rate = 0.14
56     elif sales >= 18000 and sales <= 21999.99:
57         rate = 0.16
58     else:
59         rate = 0.18
60
61     # 返回佣金率
62     return rate
63
64 # 调用 main 函数
65 main()
```

函数 determine_comm_rate 函数接收销售额变量 sales 作为实参, 并返回与这一档销售额对应的佣金率。第 50 行~第 59 行的 if-elif-else 语句测试 sales, 并相应地为局部变量

rate 赋值。第 62 行返回 rate 变量中的值。

程序输出（用户输入的内容加粗）

```
输入月销售额: 14650.00   
输入预支工资金额, 如果  
没有预支工资则输入 0。  
预支工资: 1000.00   
工资为$758.00。
```

程序输出（用户输入的内容加粗）

```
输入月销售额: 9000.00   
输入预支工资金额, 如果  
没有预支工资则输入 0。  
预支工资: 0   
工资为$900.00。
```

程序输出（用户输入的内容加粗）

```
输入月销售额: 12000.00   
输入预支工资金额, 如果  
没有预支工资则输入 0。  
预支工资: 2000.00   
工资为$-560.00。  
销售人员必须偿还  
公司。
```

5.8.4 返回字符串

之前展示的都是返回数字的函数。除此之外，函数也可以返回字符串。例如，以下函数提示用户输入姓名，然后返回用户输入的字符串：

```
def get_name():  
    # 获取用户姓名  
    name = input('输入你的姓名: ')  
    # 返回姓名  
    return name
```

函数也可以返回一个 f 字符串。在这种情况下，Python 解释器会先对 f 字符串中的占位符和格式说明符进行求值，并返回格式化好的结果。下面展示了一个例子。

```
def dollar_format(value):  
    return f'${value:,.2f}'
```

dollar_format 函数的作用是接受一个数值作为实参，并返回将这个数值格式化为美元金额的一个字符串。例如，将浮点值 89.578 传给函数，函数将返回字符串 '\$89.58'。

5.8.5 返回布尔值

Python 允许写返回 True 或 False 的**布尔函数**。可以使用布尔函数来测试一个条件，然后返回 True 或 False 来表示该条件是否成立。在决策和重复结构中测试的复杂条件可以用布尔函数来简化。

例如，假定程序要求用户输入一个数字，然后判断它是偶数还是奇数。以下代码展示了如何进行这个判断。

```
number = int(input('输入一个数: '))
if (number % 2) == 0:
    print('这个数是偶数。')
else:
    print('这个数是奇数。')
```

下面来仔细看看该 if-else 语句所测试的布尔表达式。

`(number % 2) == 0`

该表达式使用了第 2 章介绍的求余操作符%。它将两个数相除并返回余数。因此，这段代码的意思是：“如果 number 除以 2 的余数等于 0，那么显示一条消息，指出这个数是偶数，否则显示消息指出这个数是奇数”。

由于偶数除以 2 总是余 0，所以这个逻辑可行。然而，如果能以某种方式将代码改写为：“如果 number 是偶数，那么显示一条消息指出这个数是偶数，否则显示一条消息指出这个数是奇数”，那么代码会更容易理解。事实上，这完全能通过布尔函数来实现。为此，我们可以写一个名为 `is_even` 的布尔函数，它接收一个数字作为实参，数字为偶数就返回 True，否则返回 False。函数代码如下所示。

```
def is_even(number):
    # 判断 number 是否为偶数。如果是，
    # 就将 status 设为 True，否则将
    # status 设为 False。
    if (number % 2) == 0:
        status = True
    else:
        status = False
    # 返回 status 变量的值
    return status
```

然后，可以重写 if-else 语句，使其调用 `is_even` 函数来判断 number 是否为偶数：

```
number = int(input('输入一个数: '))
if is_even(number):
    print('这个数是偶数。')
else:
    print('这个数是奇数。')
```

这个逻辑不仅更容易理解，而且在程序任何需要测试数字奇偶性的地方，都可以调用该函数。

5.8.6 在校验代码中使用布尔函数

还可以使用布尔函数来简化复杂的输入校验代码。例如，假设程序提示用户输入一个产品型号，并且只能接受 100, 200 和 300 这三个值。可以设计如下所示的输入算法。

```
# 获取型号
model = int(input('请输入型号: '))
# 校验型号
while model != 100 and model != 200 and model != 300:
    print('有效型号是 100, 200 和 300。')
    model = int(input('请输入一个有效型号: '))
```

校验循环使用了一个长的复合布尔表达式，只要 `model` 不等于 100，不等于 200，也不等于 300，该循环就会进行迭代。尽管这种逻辑可以工作，但完全可以写一个布尔函数来测试 `model` 变量，并在循环条件中调用该函数来简化校验循环。例如，可以将 `model` 变量传给一个 `is_invalid` 函数。如果型号无效，那么函数返回 `True`，否则返回 `False`。然后，可以像下面这样重写校验循环。

```
# 获取型号
model = int(input('请输入型号: '))
# 校验型号
while is_invalid(model):
    print('有效型号是 100, 200 和 300。')
    model = int(input('请输入一个有效型号: '))
```

这使循环更易读。现在一眼就能看出，只要 `model` 无效，循环就会一直迭代。以下代码展示了如何编写 `is_invalid` 函数。它接受一个型号作为实参，如果实参不是 100, 200 或 300，函数就返回 `True` 表示型号无效；否则，函数返回 `False`。

```
def is_invalid(mod_num):
    if mod_num != 100 and mod_num != 200 and mod_num != 300:
        status = True
    else:
        status = False
    return status
```

5.8.7 返回多个值

到目前为止，我们看到的所有返回值的函数都只返回一个值。但在 `Python` 中，并不局限于只能返回一个值。可以在 `return` 语句后指定多个以逗号分隔的表达式，如以下常规格式所示。

```
return 表达式1, 表达式2, ...
```

下面给出了 `get_name` 函数的定义。该函数提示用户分别输入名字和姓氏。这两个值存储在两个局部变量中：`first` 和 `last`。`return` 语句一次性返回这两个变量。

```
def get_name():
    # 获取名字和姓氏
    first = input('输入名字: ')
    last = input('输入姓氏: ')
    # 返回名字和姓氏
    return first, last
```

在赋值语句中调用该函数时，需要在 `=` 操作符左侧使用两个变量，如下例所示。

```
first_name, last_name = get_name()
```

`return` 语句中列出的值按其出现顺序被赋给 `=` 操作符左侧的变量。上述语句执行后，第一个变量的值将被赋给 `first_name`，第二个变量的值则被赋给 `last_name`。注意，`=` 操作符左侧的变量个数必须与函数返回值的个数相匹配。否则会发生错误。

5.8.8 从函数返回 None

Python 有一个特殊的内置值 `None`，用来表示“没有值”。有的时候，我们需要从函数返回 `None` 来表示发生了某种错误。以下函数展示了一个例子。

`divide` 函数接收两个实参 `num1` 和 `num2`，并返回 `num1` 除以 `num2` 的结果。然而，如果 `num2` 等于 `0`，那么会发生错误，因为除以 `0` 是不允许的。为了防止程序崩溃，我们可以修改函数，在执行除法运算之前判断 `num2` 是否等于 `0`。如果 `num2` 等于 `0`，那么直接返回 `None`。下面是修改后的代码。

```
def divide(num1, num2):
    if num2 == 0:
        result = None
    else:
        result = num1 / num2
    return result
```

程序 5-26 演示了如何调用 `divide` 函数，并根据它的返回值来判断是否发生了错误。

程序 5-26 (`none_demo.py`)

```
1  # 这个程序演示了 None 关键字
2
3  def main():
4      # 从用户处获取两个数字
5      num1 = int(input('输入一个数: '))
6      num2 = int(input('输入另一个数: '))
7
```

```

8     # 调用 divide 函数
9     quotient = divide(num1, num2)
10
11    # 显示结果
12    if quotient is None:
13        print('不允许除以零。')
14    else:
15        print(f'{num1}除以{num2}的结果是{quotient}。')
16
17    # divide 函数计算 num1 除以 num2 的结果,
18    # 并返回这个结果。如果 num2 为 0,
19    # 那么函数会返回 None。
20    def divide(num1, num2):
21        if num2 == 0:
22            result = None
23        else:
24            result = num1 / num2
25        return result
26
27    # 调用 main 函数
28    main()

```

程序输出（用户输入的内容加粗）

```

输入一个数: 10 
输入另一个数: 0 
不允许除以零。

```

来仔细看看 `main` 函数。第 5 行~第 6 行从用户处获取两个数字。第 9 行调用 `divide` 函数，将两个数字作为实参传递。函数返回的值被赋值给 `quotient`（商）变量。第 12 行的 `if` 语句判断 `quotient` 是否等于 `None`。如果 `quotient` 等于 `None`，那么第 13 行显示消息“不允许除以零”，否则显示除法运算的结果。

注意，第 12 行的 `if` 语句使用的不是 `==` 操作符，而是 `is` 操作符，如下所示。

```
if quotient is None:
```

判断一个变量是否被设为 `None` 时，最好是使用 `is` 操作符而不是 `==` 操作符。在某些高级情况下（本书不涉及），`== None` 和 `is None` 这两种比较的结果是不一样的。因此，在比较一个变量是否为 `None` 时，原则上总是使用 `is` 操作符。

要判断变量是否不等于 `None`，则应该使用 `is not` 操作符，如下例所示。

```
if value is not None:
```

该语句判断 `value` 是否不等于 `None`。

检查点

5.31 函数中的 `return` 语句有什么作用？

5.32 对于以下函数定义：

```
def do_something(number):  
    return number * 2
```

- a. 函数的名称是什么？
- b. 作用是什么？
- c. 函数定义，以下语句将显示什么？

```
print(do_something(10))
```

5.33 什么是布尔函数？

5.9 math 模块

概念：Python 标准库的 `math` 模块包含许多用于数学计算的函数。

Python 标准库中的 `math` 模块包含许多用于执行数学运算的函数。表 5-2 列出了 `math` 模块中的一些函数。这些函数通常接受一个或多个值作为实参，使用它们执行数学运算，并返回结果。（表 5-2 列出的几乎所有函数都返回 `float` 值，只有 `ceil` 和 `floor` 返回 `int`。）例如，`sqrt` 函数接受一个实参并返回它的平方根。下例展示了它的用法。

```
result = math.sqrt(16)
```

该语句调用 `sqrt` 函数，将 `16` 作为实参。函数返回 `16` 的平方根，结果赋给 `result` 变量。程序 5-27 演示了 `sqrt` 函数。注意第 2 行中的 `import math` 语句。在任何使用 `math` 模块的程序中都要添加该语句。

程序 5-27 (square_root.py)

```
1  # 这个程序演示了 sqrt 函数  
2  import math  
3  
4  def main():  
5      # 获取一个数字  
6      number = float(input('输入一个数: '))
```

```

7
8     # 获取这个数字的平方根
9     square_root = math.sqrt(number)
10
11    # 显示平方根
12    print(f'{number}的平方根为{square_root}。')
13
14    # 调用 main 函数
15    main()

```

程序输出（用户输入的内容加粗）

输入一个数: **25.0**
25.0 的平方根为 **5.0**。

程序 5-28 展示了另一个使用 `math` 模块的例子。该程序使用 `hypot` 函数计算直角三角形的斜边长度。

程序 5-28 (hypotenuse.py)

```

1    # 这个程序计算直角三角形
2    # 斜边的长度。
3    import math
4
5    def main():
6        # 获取直角三角形两个直角边的长度
7        a = float(input('输入第一个直角边的长度: '))
8        b = float(input('输入第二个直角边的长度: '))
9
10       # 计算斜边的长度
11       c = math.hypot(a, b)
12
13       # 显示斜边的长度
14       print(f'斜边长度为{c}。')
15
16    # 调用 main 函数
17    main()

```

程序输出（用户输入的内容加粗）

输入第一个直角边的长度: **5.0**
输入第二个直角边的长度: **12.0**
斜边长度为 **13.0**。

表 5-2 `math` 模块中的部分函数

函数	描述
<code>acos(x)</code>	返回 x 的反余弦值（弧度）。
<code>asin(x)</code>	返回 x 的反正弦值（弧度）。
<code>atan(x)</code>	返回 x 的反正切值（弧度）。
<code>ceil(x)</code>	返回大于或等于 x 的最小整数。
<code>cos(x)</code>	返回 x 的余弦值（弧度）。
<code>degrees(x)</code>	假设 x 是以弧度表示的角度，该函数将其转换为度数。
<code>exp(x)</code>	返回 e^x ， e 是自然对数的底数。
<code>floor(x)</code>	返回小于或等于 x 的最大整数。
<code>hypot(x, y)</code>	返回从原点 $(0, 0)$ 到点 (x, y) 的斜边长度。
<code>log(x)</code>	返回 x 的自然对数。
<code>log10(x)</code>	返回 x 的以 10 为底的对数。
<code>radians(x)</code>	假设 x 是以度数表示的角度，该函数将其转换为弧度。
<code>sin(x)</code>	返回 x 的正弦值（弧度）。
<code>sqrt(x)</code>	返回 x 的平方根。
<code>tan(x)</code>	返回 x 的正切值（弧度）。

math.pi 和 math.e 值

`math` 模块还定义了两个数学常量：`pi` 和 `e`，它们分别是 π 和 e 的数学值。例如，以下计算圆面积的语句使用了 `pi`。注意，我们用点记号法来引用 `pi`。

```
area = math.pi * radius**2    # 面积 =  $\pi r^2$ 
```

检查点

5.34 如果程序要使用 `math` 模块中的函数，需要怎么写 `import` 语句？

5.35 写一个语句，使用 `math` 模块中的函数来求 `100` 的平方根并将结果赋给一个变量。

5.36 写一个语句，使用 `math` 模块中的函数将 45 度转换为弧度，并将结果赋给一个变量。

5.10 将函数存储到模块中

概念：模块是包含 Python 代码的一种文件。大型程序在分解为模块后，会更容易调试和维护。

随着程序变得越来越大和复杂，就越来越需要对代码进行高效的组织。通过之前的学习，你知道一个复杂的大型程序应该被分解为多个函数，每个函数都负责执行一个特定的任务。随着在程序中编写的函数越来越多，应该考虑将它们存储到模块中，对这些函数进行组织。

模块是包含 Python 代码的一种文件。将程序分解为模块时，每个模块都应包含执行相关任务的函数。例如，为了写一个会计系统，可以考虑将所有应收账款函数存储到它们自己的模块中，将所有应付账款函数存储到它们自己的模块中，将所有工资单函数存储到它们自己的模块中。这种编程方法称为**模块化**，它使程序更容易理解、测试和维护。

模块化还方便了代码在多个程序中的重用。如果写了一组在多个程序中都需要用到的函数，那么可以将这些函数集中到一个模块中。然后，任何程序如果需要调用其中一个函数，那么导入该模块即可。

下面来看一个简单的例子。假设需要写一个程序来计算以下数据：

- 圆的面积
- 圆的周长
- 矩形的面积
- 矩形的周长

这个程序显然需要执行两类计算：与圆相关的计算和与矩形相关的计算。可以在一个模块中写所有与圆相关的函数，在另一个模块中写所有与矩形相关的函数。程序 5-29 展示了 `circle` 模块，其中包含两个函数定义：`area`（返回圆的面积）和 `circumference`（返回圆的周长）。

程序 5-29 (`circle.py`)

```
1 # circle 模块中的函数执行
2 # 与圆相关的计算。
3 import math
4
5 # area 函数接收圆的半径作为实参，
6 # 返回圆的面积。
```

```
7 def area(radius):
8     return math.pi * radius**2
9
10 # circumference 函数接收圆的半径
11 #作为实参，返回圆的周长。
12 def circumference(radius):
13     return 2 * math.pi * radius
```

程序 5-30 展示了 `rectangle` 模块，其中包含两个函数定义：`area`（返回矩形的面积）和 `perimeter`（返回矩形的周长）。

程序 5-30 (rectangle.py)

```
1 # rectangle 模块中的函数
2 # 执行与矩形相关的计算。
3
4 # area 积函数接受一个矩形的宽度和长度
5 # 作为实参，并返回矩形的面积。
6 def area(width, length):
7     return width * length
8
9 # perimeter 函数接受一个矩形
10 # 的宽度和长度作为实参，
11 # 并返回矩形的周长。
12 def perimeter(width, length):
13     return 2 * (width + length)
```

注意这两个文件只包含函数定义，不包含用函数的调用代码。调用是由导入这些模块的程序来进行的。

在继续之前，请注意以下几点关于模块名称的问题。

- 模块的文件名应以 `.py` 结尾。不以 `.py` 结尾的模块文件名无法导入其他程序。
- 模块名称不能与 `Python` 关键字相同。例如，将模块命名为 `for` 会发生错误。

要在程序中使用这些模块，可以用 `import` 语句导入。下例导入 `circle` 模块：

```
import circle
```

当 `Python` 解释器读取这条语句时，它会先在与当前程序相同的文件夹中查找 `circle.py` 文件。如果在当前文件夹中没有找到指定的模块，`Python` 解释器会在系统中其他各种预定义位置查找。找到这个文件后，会把它加载到内存。如果没有找到，那么会报告一个错误。

一旦模块被导入，就可以调用它的函数。假设 `radius` 是代表圆的半径的一个变量，下例

演示了如何调用 `circle` 模块中的 `area` 和 `circumference` 函数。

```
my_area = circle.area(radius) # 计算圆的面积
my_circum = circle.circumference(radius) # 计算圆的周长
```

程序 5-31 是使用了上述两个模块的一个完整的程序。

程序 5-31 (geometry.py)

```
1  # 这个程序允许用户从菜单中选择执行
2  # 各种几何计算。程序导入了 circle 和
3  # rectangle 模块。
4  import circle
5  import rectangle
6
7  # 用于菜单选项的常量
8  AREA_CIRCLE_CHOICE = 1
9  CIRCUMFERENCE_CHOICE = 2
10 AREA_RECTANGLE_CHOICE = 3
11 PERIMETER_RECTANGLE_CHOICE = 4
12 QUIT_CHOICE = 5
13
14 # main 函数
15 def main():
16     # choice 变量控制循环并
17     # 容纳用户的菜单选择。
18     choice = 0
19
20     while choice != QUIT_CHOICE:
21         # 显示菜单
22         display_menu()
23
24         # 获取用户的选择
25         choice = int(input('输入你的选择: '))
26
27         # 执行选择的行动
28         if choice == AREA_CIRCLE_CHOICE:
29             radius = float(input("输入圆的半径: "))
30             print('圆的面积为', circle.area(radius))
31         elif choice == CIRCUMFERENCE_CHOICE:
32             radius = float(input("输入圆的半径: "))
33             print('圆的周长为',
34                   circle.circumference(radius))
35         elif choice == AREA_RECTANGLE_CHOICE:
36             width = float(input("输入矩形的宽度: "))
37             length = float(input("输入矩形的长度: "))
38             print('矩形的面积为', rectangle.area(width, length))
39         elif choice == PERIMETER_RECTANGLE_CHOICE:
```

```
40         width = float(input("输入矩形的宽度: "))
41         length = float(input("输入矩形的长度: "))
42         print('矩形的周长为',
43               rectangle.perimeter(width, length))
44     elif choice == QUIT_CHOICE:
45         print('退出程序...')
46     else:
47         print('错误: 无效的选择。')
48
49 # display_menu 函数显示一个菜单
50 def display_menu():
51     print('      菜单')
52     print('1) 圆的面积')
53     print('2) 圆的周长')
54     print('3) 矩形的面积')
55     print('4) 矩形的周长')
56     print('5) 退出')
57
58 # 调用 main 函数
59 main()
```

程序输出 (用户输入的内容加粗)

```
      菜单
1) 圆的面积
2) 圆的周长
3) 矩形的面积
4) 矩形的周长
5) 退出
输入你的选择: 1 
输入圆的半径: 10 
圆的面积为 314.1592653589793
      菜单
1) 圆的面积
2) 圆的周长
3) 矩形的面积
4) 矩形的周长
5) 退出
输入你的选择: 2 
输入圆的半径: 10 
圆的周长为 62.83185307179586
      菜单
1) 圆的面积
2) 圆的周长
3) 矩形的面积
4) 矩形的周长
5) 退出
输入你的选择: 3 
输入矩形的宽度: 5 
输入矩形的长度: 10 
```

```
矩形的面积为 50.0
    菜单
1) 圆的面积
2) 圆的周长
3) 矩形的面积
4) 矩形的周长
5) 退出
输入你的选择: 4 
输入矩形的宽度: 5 
输入矩形的长度: 10 
矩形的周长为 30.0
    菜单
1) 圆的面积
2) 圆的周长
3) 矩形的面积
4) 矩形的周长
5) 退出
输入你的选择: 5 
退出程序...
```

条件执行模块中的 main 函数

导入一个模块时，Python 解释器会执行模块中的语句，就像模块是一个独立的程序一样。例如，在导入程序 5-29 的 `circle.py` 模块时，会发生下面这些事情：

- 导入 `math` 模块。
- 定义一个名为 `area` 的函数。
- 定义一个名为 `circumference` 的函数。

在导入程序 5-30 的 `rectangle.py` 模块时，会发生下面这些事情：

- 定义一个名为 `area` 的函数。
- 定义一个名为 `perimeter` 的函数。

当程序员在创建模块时，一般并不打算将这些模块作为独立的程序运行。模块的目的就是在其他程序中导入。因此，大多数模块只定义诸如函数之类的东西。

但是，完全可以创建一个 Python 模块，它既可以作为独立的程序运行，也可以导入到其他程序中。例如，假设程序 A 定义了几个有用的函数，而你想在程序 B 中使用这些函数。为此，可以将程序 A 导入到程序 B 中。但是，你并不希望在导入程序 A 时时执行它的 `main` 函数。换言之，你希望它只是定义函数，而不希望它执行任何函数。为了达到这个目的，需要在程序 A 中写代码来判断当前文件是如何使用的。是作为一个独立的程序运行？还是被导入到另一个程序中？答案将决定是否应该执行程序 A 中的 `main` 函数。

幸好，Python 提供了进行这个判断的机制。当 Python 解释器处理源代码文件时，它会创建一个名为 `__name__` 的特殊变量（变量名以两个下划线字符开始，以两个下划线字符结束）。

如果文件作为模块导入，那么变量 `__name__` 会被设为模块名称。否则，如果文件作为一个独立的程序执行，那么 `__name__` 变量会被设为字符串 `'__main__'`。可以根据变量的值来决定是否应该执行 `main` 函数。如果 `__name__` 变量的值等于 `'__main__'`，那么应该执行 `main` 函数，因为文件是作为独立程序来执行的。否则不应执行 `main` 函数，因为文件是作为模块导入的。

程序 5-32 (`rectangle2.py`) 展示了一个例子。。

程序 5-32 (`rectangle2.py`)

```
1  # area 函数接受一个矩形的宽度和长度
2  # 作为实参，并返回矩形的面积。
3  def area(width, length):
4      return width * length
5
6  # perimeter 函数接受一个
7  # 矩形的宽度和长度作为
8  # 实参，并返回矩形的周长。
9  def perimeter(width, length):
10     return 2 * (width + length)
11
12 # main 函数用于测试本程序文件定义的其他函数
13 def main():
14     width = float(input("输入矩形的宽度: "))
15     length = float(input("输入矩形的长度: "))
16     print('矩形的面积为', area(width, length))
17     print('矩形的周长为', perimeter(width, length))
18
19 # 只有当文件作为独立程序运行时，
20 # 才会调用 main 函数
21 if __name__ == '__main__':
22     main()
```

`rectangle2.py` 程序定义了 `area`、`perimeter` 和 `main` 这三个函数。然后，第 21 行的 `if` 语句测试 `__name__` 变量的值。如果变量值等于 `'__main__'`，那么第 22 行的语句会调用 `main` 函数。否则，如果 `__name__` 变量被设为其他任何值，`main` 函数都不会执行。

任何 Python 源代码文件只要包含了一个 `main` 函数，程序 5-32 展示的技术都是一个很好的实践。它确保在导入该文件时，它会作为一个模块使用，而在直接执行该文件时，它会作为一个独立的程序。从现在起，本书在展示一个使用 `main` 函数的例子时，都会采用这个技术。

5.11 海龟图形：使用函数将代码模块化

概念：可以将常用的海龟图形操作写成函数，并在需要时调用。

使用海龟来绘图通常需要执行多个步骤。例如，假设要绘制一个 100 像素宽的正方形，填充颜色为蓝色，那么需要执行以下步骤。

```
turtle.fillcolor('blue')
turtle.begin_fill()
for count in range(4):
    turtle.forward(100)
    turtle.left(90)
turtle.end_fill()
```

写这 6 行代码表面上并不费事，但如果要在屏幕的不同位置绘制大量蓝色方格呢？突然之间，你会发现需要重复大量类似的代码。在这种情况下，可以写一个函数在指定位置绘制正方形，并在需要的时候调用该函数来简化程序（并节省大量时间）。

程序 5-33 演示了这样的函数。第 14 行~第 23 行定义了 `square` 函数，它的参数如下所示。

- `x` 和 `y`：这是正方形左下角的(X, Y)坐标。
- `width`：正方形的边长，单位为像素。
- `color`：代表填充颜色的一个字符串。

我们在 `main` 函数中调用了三次 `square` 函数。

- 第 5 行绘制第一个正方形，将其左下角定位在(100, 0)。正方形边长为 50 像素，填充颜色为红色。
- 第 6 行绘制第二个正方形，将其左下角定位在(-150, -100)。正方形边长为 200 像素，填充颜色为蓝色。
- 第 7 行绘制第三个正方形，将其左下角定位在(-200, 150)。正方形边长为 75 像素，填充颜色为绿色。

程序绘制了如图 5-26 所示的三个正方形。

程序 5-33 (draw_squares.py)

```
1 import turtle
2
3 def main():
4     turtle.hideturtle()
```

```
5     square(100, 0, 50, 'red')
6     square(-150, -100, 200, 'blue')
7     square(-200, 150, 75, 'green')
8
9     # square 函数绘制正方形。
10    # x 和 y 参数是左下角坐标,
11    # width 参数是边长。
12    # color 参数是代表填充颜色的字符串。
13
14    def square(x, y, width, color):
15        turtle.penup()          # 抬起笔
16        turtle.goto(x, y)       # 移至指定位置
17        turtle.fillcolor(color) # 设置填充颜色
18        turtle.pendown()        # 放下笔
19        turtle.begin_fill()     # 开始填充
20        for count in range(4):  # 画正方形
21            turtle.forward(width)
22            turtle.left(90)
23        turtle.end_fill()       # 结束填充
24
25    # 调用 main 函数
26    if __name__ == '__main__':
27        main()
```

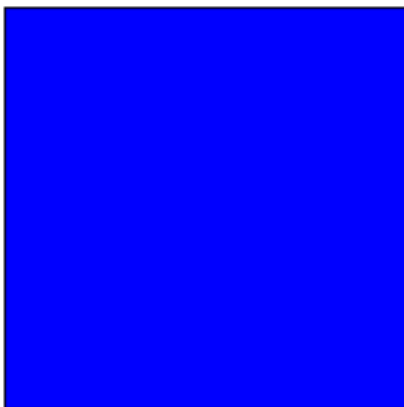


图 5-26 程序 5-33 的输出结果

程序 5-34 展示了另一个例子，它使用函数将画圆的代码模块化。第 14 行~第 21 行定义了 `circle` 函数，它的参数如下所示。

- `x` 和 `y`: 这是圆心的(X, Y)坐标。
- `radius`: 圆的半径，单位为像素。
- `color`: 代表填充颜色的一个字符串。

我们在 `main` 函数中调用了三次 `circle` 函数。

- 第 5 行绘制第一个圆，圆心在($0, 0$)，圆的半径为 100 像素，填充颜色为红色。
- 第 6 行绘制第二个圆，圆心在($-150, -75$)，圆的半径为 50 像素，填充颜色为蓝色。
- 第 7 行绘制第三个圆，圆心在($-200, 150$)，圆的半径为 75 像素，填充颜色为绿色。

程序绘制了如图 5-27 所示的三个圆。

程序 5-34 (`draw_circles.py`)

```
1  import turtle
2
3  def main():
4      turtle.hideturtle()
5      circle(0, 0, 100, 'red')
6      circle(-150, -75, 50, 'blue')
7      circle(-200, 150, 75, 'green')
8
9  # circle 函数画圆。
10 # x 和 y 参数是圆心，
11 # radius 参数是半径。
12 # color 参数是代表填充颜色的字符串。
13
14 def circle(x, y, radius, color):
15     turtle.penup()           # 抬起笔
16     turtle.goto(x, y - radius) # 移至指定位置
17     turtle.fillcolor(color)  # 设置填充颜色
18     turtle.pendown()        # 放下笔
19     turtle.begin_fill()      # 开始填充
20     turtle.circle(radius)    # 画圆
21     turtle.end_fill()        # 结束填充
22
23 # 调用 main 函数
24 if __name__ == '__main__':
25     main()
```

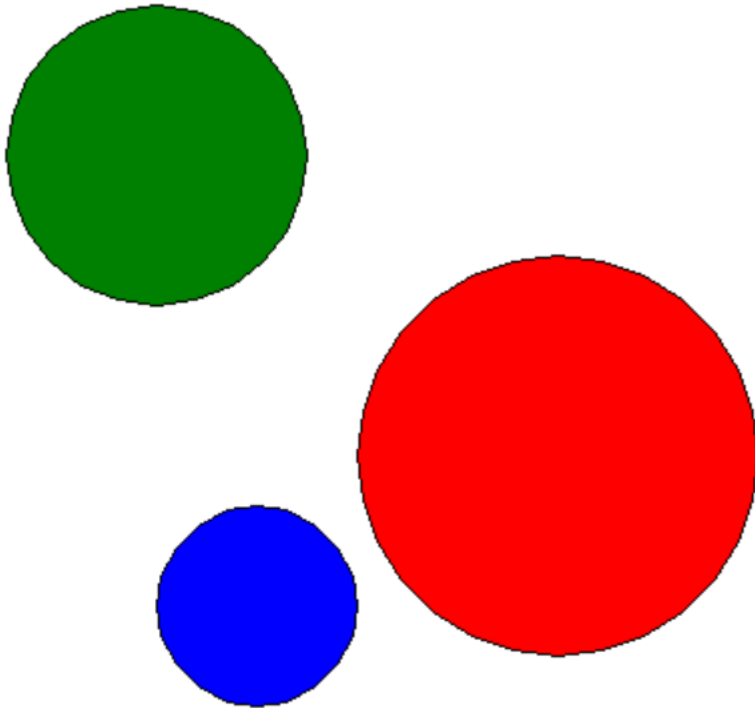


图 5-27 程序 5-34 的输出结果

程序 5-35 展示了另一个例子，它使用函数将用于画线的代码模块化。第 20 行~第 25 行定义了 `line` 函数，它的参数如下所示。

- `startX` 和 `startY`: 线段起点的 (X, Y) 坐标。
- `endX` 和 `endY`: 线段终点的 (X, Y) 坐标。
- `color`: 代表线段颜色的一个字符串。

我们在 `main` 函数中调用了三次 `line` 函数来画一个三角形。

- 第 13 行从三角形上顶点 $(0, 100)$ 到左顶点 $(-100, -100)$ 画一条红色的线。
- 第 14 行从三角形上顶点 $(0, 100)$ 到右顶点 $(100, 100)$ 画一条蓝色的线。
- 第 15 行从三角形左顶点 $(-100, -100)$ 到右顶点 $(100, 100)$ 画一条绿色的线。

程序绘制的三角形如图 5-28 所示。

程序 5-35 (`draw_lines.py`)

```

1  import turtle
2
3  # 代表三角形顶点的具名常量
4  TOP_X = 0
5  TOP_Y = 100
6  BASE_LEFT_X = -100
7  BASE_LEFT_Y = -100
8  BASE_RIGHT_X = 100
9  BASE_RIGHT_Y = -100
10
11 def main():
12     turtle.hideturtle()
13     line(TOP_X, TOP_Y, BASE_LEFT_X, BASE_LEFT_Y, 'red')
14     line(TOP_X, TOP_Y, BASE_RIGHT_X, BASE_RIGHT_Y, 'blue')
15     line(BASE_LEFT_X, BASE_LEFT_Y, BASE_RIGHT_X, BASE_RIGHT_Y, 'green')
16
17 # line 函数绘制一条从(startX, startY)到(endX, endY)的线,
18 # color 参数是线的颜色。
19
20 def line(startX, startY, endX, endY, color):
21     turtle.penup()          # 抬起笔
22     turtle.goto(startX, startY) # 移至起点
23     turtle.pendown()        # 放下笔
24     turtle.pencolor(color)   # 设置笔的颜色
25     turtle.goto(endX, endY)  # 画一条线
26
27 # 调用 main 函数
28 if __name__ == '__main__':
29     main()

```

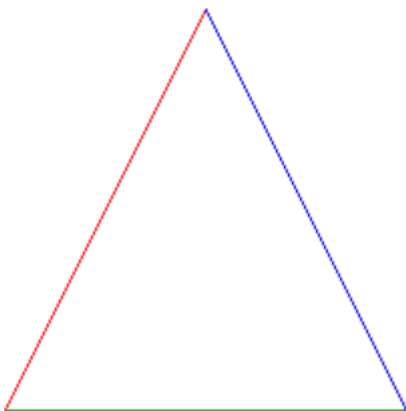


图 5-28 程序 5-35 的输出

将图形函数存储到模块中

随着你写的海龟图形函数越来越多，应该考虑将它们存储到一个模块中，以便在任何需要使用它们的程序中导入。例如，程序 5-36 展示了一个名为 `my_graphics.py` 的模块，它包含

了前面介绍的 `square`, `circle` 和 `line` 函数。程序 5-37 则展示了如何导入该模块并调用其中的函数。图 5-29 是程序的输出。

程序 5-36 (my_graphics.py)

```
1  # 海龟图形函数
2  import turtle
3
4  # square 函数绘制正方形。
5  # x 和 y 参数是左下角坐标,
6  # width 参数是边长。
7  # color 参数是代表填充颜色的字符串。
8
9  def square(x, y, width, color):
10     turtle.penup()          # 抬起笔
11     turtle.goto(x, y)      # 移至指定位置
12     turtle.fillcolor(color) # 设置填充颜色
13     turtle.pendown()      # 放下笔
14     turtle.begin_fill()   # 开始填充
15     for count in range(4): # 画正方形
16         turtle.forward(width)
17         turtle.left(90)
18     turtle.end_fill()     # 结束填充
19
20 # circle 函数画圆。
21 # x 和 y 参数是圆心,
22 # radius 参数是半径。
23 # color 参数是代表填充颜色的字符串。
24
25 def circle(x, y, radius, color):
26     turtle.penup()          # 抬起笔
27     turtle.goto(x, y - radius) # 移至指定位置
28     turtle.fillcolor(color) # 设置填充颜色
29     turtle.pendown()      # 放下笔
30     turtle.begin_fill()   # 开始填充
31     turtle.circle(radius)  # 画圆
32     turtle.end_fill()     # 结束填充
33
34 # line 函数绘制一条从(startX, startY)到(endX, endY)的线,
35 # color 参数是线的颜色。
36
37 def line(startX, startY, endX, endY, color):
38     turtle.penup()          # 抬起笔
39     turtle.goto(startX, startY) # 移至起点
40     turtle.pendown()      # 放下笔
41     turtle.pencolor(color)  # 设置笔的颜色
42     turtle.goto(endX, endY)  # 画一条线
```

程序 5-37 (graphics_mod_demo.py)

```
1  import turtle
2  import my_graphics
3
4  # 具名常量
5  X1 = 0
6  Y1 = 100
7  X2 = -100
8  Y2 = -100
9  X3 = 100
10 Y3 = -100
11 RADIUS = 50
12
13 def main():
14     turtle.hideturtle()
15
16     # 画一个正方形
17     my_graphics.square(X2, Y2, (X3 - X2), 'gray')
18
19     # 画一些圆
20     my_graphics.circle(X1, Y1, RADIUS, 'blue')
21     my_graphics.circle(X2, Y2, RADIUS, 'red')
22     my_graphics.circle(X3, Y3, RADIUS, 'green')
23
24     # 画一些线
25     my_graphics.line(X1, Y1, X2, Y2, 'black')
26     my_graphics.line(X1, Y1, X3, Y3, 'black')
27     my_graphics.line(X2, Y2, X3, Y3, 'black')
28
29 # 调用 main 函数
30 if __name__ == '__main__':
31     main()
```

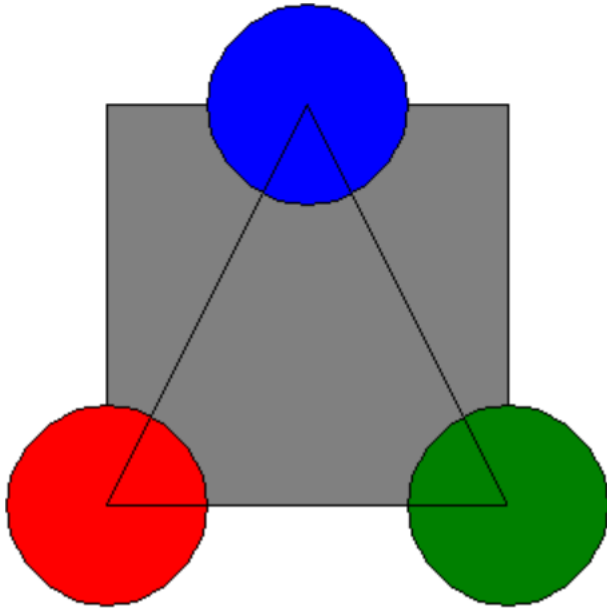



图 5-29 程序 5-37 的输出结果

复习题

选择题

1. 我们在程序中使用_____来组织用于执行一项特定任务的一组语句。
a. 块 b. 参数 c. 函数 d. 表达式
2. 有助于在程序中减少代码重复的一种设计技术称为_____，函数能为此提供帮助。
a. 代码重用 b. 分而治之 c. 调试 d. 团队合作
3. 函数定义的第一行称为_____。
a. 函数主体 b. 函数指引 c. 初始化器 d. 函数头
4. 我们通过_____一个函数来执行它。
a. 定义 b. 调用 c. 导入 d. 导出
5. 程序员将算法分解为多个函数的设计技术称为_____。
a. 自上而下设计 b. 代码简化 c. 代码重构 d. 子任务分级
6. _____是一种直观表示程序中函数之间关系的图。

a. 流程图 b. 函数关系图 c. 符号图 d. 层次结构图

7. _____关键字会被 Python 解释器忽略，可以把它用作以后才编写的代码的占位符。

a. placeholder b. pass c. pause d. skip

8. _____是在函数内部创建的变量。

a. 全局变量 b. 局部变量

c. 隐藏变量 d. 以上都不对，不能在函数内部创建变量。

9. _____是指程序中可以访问一个变量的部分。

a. 声明空间 b. 可见区域 c. 范围 d. 模式

10. _____是发送到函数中的实际数据。

a. 实参 b. 形参 c. 数据头 d. 数据包

11. _____是一个特殊变量，负责在函数被调用时接收数据。

a. 实参 b. 形参 c. 数据头 d. 数据包

12. _____变量在程序文件中对每个函数都可见。

a. 局部 b. 全局 c. 引用 d. 参数

13. 程序中应尽量避免使用_____变量。

a. 局部 b. 全局 c. 引用 d. 参数

14. _____是编程语言预先帮我们写好的函数。

a. 标准函数 b. 库函数 c. 自定义函数 d. 自助函数

15. 标准库函数_____返回指定范围内的一个随机整数。

a. random b. randint c. random_integer d. uniform

16. 标准库函数_____返回 0.0~1.0（但不包括 1.0）范围内的一个随机浮点数。

a. random b. randint c. random_integer d. uniform

17. 标准库函数_____返回指定范围内的一个随机浮点数。

a. random b. randint c. random_integer d. uniform

18. _____语句结束函数执行，并向程序调用函数的那个部分返回一个值。

a. end b. send c. exit d. return

19. _____ 是一种设计工具，用于描述函数的输入、处理和输出。

a. 层次结构图 b. IPO 图 c. 数据报图 d. 数据处理图

20. _____ 函数只返回 True 或 False。

a. 二元 b. 真假 c. 布尔 d. 逻辑

21. _____ 是 math 模块中的一个函数。

a. derivative b. factor c. sqrt d. differentiate

判断题

1. “分而治之”的意思是一个团队中的所有程序员应该分工合作。
2. 函数方便了团队中的程序员的协作。
3. 函数名应尽可能简短。
4. 调用函数和定义函数的意思是一样的。
5. 流程图显示程序中函数之间的层次关系。
6. 层次结构图不显示函数内部执行的具体步骤。
7. 一个函数中的语句可以访问另一个函数中的局部变量。
8. 在 Python 中，不能写接受多个实参的函数。
9. 在 Python 中，可以在调用函数时指定将哪个实参传给哪个形参。
10. 一个函数调用不能同时存在关键字参数和非关键字实参。
11. Python 解释器已经内置了一些库函数，不需要显式导入。
12. 在程序中不需要使用 import 语句就能使用 random 模块中的函数。
13. 为了简化复杂的数学表达式，可以分解表达式的一部分，并将其放到函数中。
14. Python 中的函数可以返回多个值。
15. IPO 图只提供对函数输入、处理和输出的简要说明，不会显示函数中执行的具体步骤。

简答题

1. 函数如何为程序中的代码重用提供帮助？
2. 说出并描述函数定义的两个组成部分。
3. 在函数执行过程中，函数块（函数主体）结束时会发生什么？
4. 什么是局部变量？什么语句能访问局部变量？
5. 什么是局部变量的作用域？
6. 为什么全局变量使程序难以调试？
7. 假设要从以下序列中随机选择一个数：0, 5, 10, 15, 20, 25, 30，你会使用什么库函数？
8. 返回值的函数中必须有什么语句？
9. IPO 图上列出了哪三样东西？
10. 什么是布尔函数？
11. 将大型程序分解为多个模块有什么好处？

算法工作台

1. 写一个名为 `times_ten` 的函数。该函数应接受一个实参并显示它乘以 10 的结果。
2. 给定以下函数头，写一个调用该函数的语句，将 12 作为实参传递。

```
def show_value(quantity):
```

3. 给定以下函数头：

```
def my_function(a, b, c):
```

再给定以下 `my_function` 调用：

```
my_function(3, 2, 1)
```

执行这个调用时，会为 `a` 赋什么值？为 `b` 赋什么值？为 `c` 赋什么值？

4. 以下程序将显示什么？

```
def main():
```

```
x = 1
y = 3.4
print(x, y)
change_us(x, y)
print(x, y)
def change_us(a, b):
    a = 0
    b = 0
    print(a, b)
main()
```

5. 给定以下函数定义：

```
def my_function(a, b, c):
    d = (a + c) / b
    print(d)
```

a. 写一个语句来调用该函数，并使用关键字参数（关键字实参）将 2 传给 a，将 4 传给 b，将 6 传给 c。

b. 像上面这样调用函数后，会显示什么值？

6. 写语句来生成 1~100 的一个随机数，并将其赋给名为 rand 的变量。

7. 以下语句调用一个名为 half 的函数，该函数返回实参值的一半（假设 number 变量引用一个 float 值）。请为函数编写代码。

```
result = half(number)
```

8. 某个程序包含以下函数定义：

```
def cube(num):
    return num * num * num
```

写一个语句，将值 4 传给该函数，并将返回值赋给 result 变量。

9. 写一个名为 times_ten 的函数，它接受一个数字作为实参。函数被调用时，会返回实参乘以 10 的结果。

10. 写一个名为 get_first_name 的函数，要求用户输入姓名并返回该姓名。

编程练习

1. 公里换算

 视频讲解：The Kilometer Converter Problem

写一个程序，要求用户输入以公里为单位的距离，然后使用一个函数将其换算为英里。换算公式是：

$$\text{英里} = \text{公里} \times 0.6214$$

2. 重构销售税程序

第 2 章的编程练习 6 要求写一个销售税程序，计算并显示购买商品时郡县和州的销售税。如果已经写好了该程序，请重新设计它，将不同的子任务放在单独的函数中。如果还没有写过该程序，请现在使用函数来写。

3. 多大保额合适？

许多财务专家建议，业主应该为自己的房子或建筑物投保，保额至少为结构重建成本的 80%。请写一个程序，要求用户输入房子或建筑物的重建成本，然后显示最低应该投保的金额。

4. 用车成本

写一个程序，要求用户输入每月的用车成本。这些成本包括：车贷、保险、油费、机油、轮胎和保养。然后，程序应显示这些项目的月总成本和年总成本。

5. 房产税

美国一个郡（县）的房产税根据房产评估价值来征收房产税，评估价值为房产实际价值的 60%。例如，如果一英亩土地的实际价值为 10000 美元，那么它的评估价值为 6000 美元。在这种情况下，房产税为评估价值每 100 美元 72 美分。评估价值为 6000 美元的一英亩土地的税额为 43.20 美元。请写一个程序，询问房产的实际价值，并显示评估价值和房产税。

6. 体育场座位

体育场有三种座位。A 类座位 20 美元，B 类座位 15 美元，C 类座位 10 美元。请写一个程序，询问每类座位售出了多少张票，然后显示总销售金额。

7. 油漆工作估算器

一家油漆公司确定每 112 平方英尺的墙面需要 1 加仑的油漆和 8 小时的工时。该公司的工时费为每小时 35.00 美元。请写一个程序，要求用户输入需要油漆的墙面面积（单位：平方英尺）和每加仑油漆的价格。程序应显示以下数据：

- 需要多少加仑油漆
- 所需工时
- 油漆费用
- 工时费

-
- 油漆工作的总费用

8. 月销售税

某零售企业必须提交月度销售税报告，列出当月销售总额以及州和郡县收取的销售税额。其中，州销售税率为 5%，郡县销售税率为 2.5%。请写一个程序，要求用户输入月销售总额。程序基于该数据计算并显示以下内容：

- 郡县销售税额
- 州销售税额
- 销售税总额（郡县和州相加）

9. 英尺换算为英寸

 视频讲解：The Feet To Inches Problem

1 英尺等于 12 英寸。写一个名为 `feet_to_inches` 的函数，接收一个英尺数作为实参，并返回相应的英寸数。在一个程序中调用该函数，提示用户输入英尺数，然后显示相应的英寸数。

10. 数学测验

写一个进行简单数学测验的程序。程序应显示要做加法的两个随机数，例如：

```
247
+129
```

程序应允许学生输入答案。如果答案正确，那么显示祝贺消息。如果不正确，那么显示正确答案是多少。

11. 判断较大的值

写一个名为 `max` 的函数来接收两个整数值作为实参，并返回两者中较大的那个。例如，如果向函数传递 7 和 12，那么函数应返回 12。在一个程序中提示用户输入两个整数值，并调用该函数。程序应显示两个值中较大的那个。

12. 下落距离

物体在重力作用下下落时，可以使用以下公式计算物体在特定时间内下落的距离。

$$d = \frac{1}{2} gt^2$$

其中， d 为距离（米）， g 为重力加速度（9.8）， t 为物体下落时间（秒）。

写一个名为 `falling_distance` 的函数，接收物体下落时间（秒）作为实参。函数应返回物体在该时间内下落的距离（米）。写一个程序，在循环中调用该函数，将 1~10 的每个值

作为实参传递，并显示返回值。

13. 计算动能

在物理学中，我们说运动中的物体具有动能。可以使用以下公式计算运动物体的动能。

$$KE = \frac{1}{2} mv^2$$

其中， KE 为动能， m 为物体的质量（千克）， v 为物体的速度（米/秒）。

写一个名为 `kinetic_energy` 的函数，接收物体的质量（千克）和速度（米/秒）作为实参。函数应返回物体的动能。写一个程序，要求用户输入质量和速度值，然后调用 `kinetic_energy` 函数来计算并显示物体的动能。

14. 考试平均分和字母成绩

写程序要求用户输入 5 个考试分数。程序应显示和每个分数对应的字母成绩和平均考试分数。在程序中写以下函数。

- `calc_average`: 该函数接受 5 个考试分数作为实参，并返回这些分数的平均值。
- `determine_grade`: 该函数接受一个考试分数作为实参，并根据以下字母成绩表返回对应的字母成绩。

分数	字母成绩
90~100	A
80~89	B
70~79	C
60~69	D
60 以下	F

15. 奇/偶计数器

本章展示了判断数字是偶数还是奇数的一个算法。写一个程序来生成 100 个随机数，并统计其中有多少个偶数，有多少个奇数。

16. 质数

质数（prime number）是指只能被自身和 1 整除的数。例如，数字 5 是质数，因为它只能被 1 和 5 整除。然而，数字 6 不是质数，因为它可以被 1, 2, 3 和 6 整除。

写一个名为 `is_prime` 的布尔函数，它接受一个整数作为实参，并在实参是质数时返回 `True`，否则返回 `False`。在一个程序中使用该函数，提示用户输入一个数字，然后显示一条消息来指出这个数是不是质数。



提示：以前说过，`%`操作符返回两个数相除的余数。在表达式 `num1 % num2` 中，如果 `num1` 能被 `num2` 整除，那么`%`操作符返回 `0`。

17. 质数列表

本练习假定你已经在编程练习 16 中实现了 `is_prime` 函数。再写一个程序，显示 1~100 的所有质数。程序应该有一个调用 `is_prime` 函数的循环。

18. 未来值

假设在一个按月计算复利的储蓄账户中存入一定金额的钱，并且想计算在特定月数之后会有多少金额。计算公式如下所示：

$$F = P \times (1 + i)^t$$

在这个公式中：

- F 是指定月数后的账户未来值
- P 是账户现值
- i 是月利率
- t 是月数

写一个程序，提示用户输入账户的现值、月利率以及存款期限（月数）。程序应将这些值传给一个函数，函数返回账户在指定月数后的未来值。程序应显示账户的未来值。

19. 猜随机数游戏

写程序来生成 1~100 的一个随机数，并要求用户猜测这个数。如果用户猜的数比随机数大，程序应显示“太大了，请再试一次。”如果猜的数比随机数小，程序应显示“太小了，请再试一次。”如果用户猜中了这个数字，那么祝贺用户并生成一个新的随机数，重新开始游戏。

*可选改进：*对游戏进行改进，以统计用户猜测的次数。用户正确猜到随机数后，程序应显示总共猜了多少次。

20. 剪刀石头布

写一个程序，让用户与计算机玩“剪刀石头布”游戏。下面是程序运行的过程。

1.当程序开始时，生成 1~3 的一个随机数。如果数字为 1，则计算机选择石头。如果数字

为 2，则计算机选择布。如果数字是 3，则计算机选择剪刀。（先不要显示计算机的选择。）

2. 用户在键盘上输入“石头”、“布”或“剪刀”。

3. 显示计算机的选择。

4. 根据以下规则选出获胜者。

- 如果一个玩家选择石头，另一个玩家选择剪刀，那么石头获胜（石头砸坏剪刀）。
- 如果一个玩家选择剪刀，另一个玩家选择布，那么剪刀获胜（剪刀剪烂布）。
- 如果一个玩家选择布，另一个玩家选择石头，那么布获胜（布包住石头）。
- 如果两个玩家选择相同，那么必须再次进行游戏以决胜负。

21. 海龟图形：三角形函数

写一个名为 `triangle` 的函数，使用海龟图形库来绘制三角形。函数的参数应包括三角形顶点的 X 和 Y 坐标，以及三角形的填充颜色。在一个程序中演示该函数。

22. 海龟图形：模块化的雪人

写一个使用海龟图形来显示雪人的程序，如图 5-30 所示。除了 `main` 函数外，程序还应包含以下函数。

- `drawBase`: 绘制雪人的底座，即底部的大雪球。
- `drawMidSection`: 绘制中间较小的雪球。
- `drawArms`: 绘制雪人的手臂。
- `drawHead`: 绘制雪人的头，包括眼睛、嘴巴和其他你想要的面部特征。
- `drawHat`: 绘制雪人的帽子。

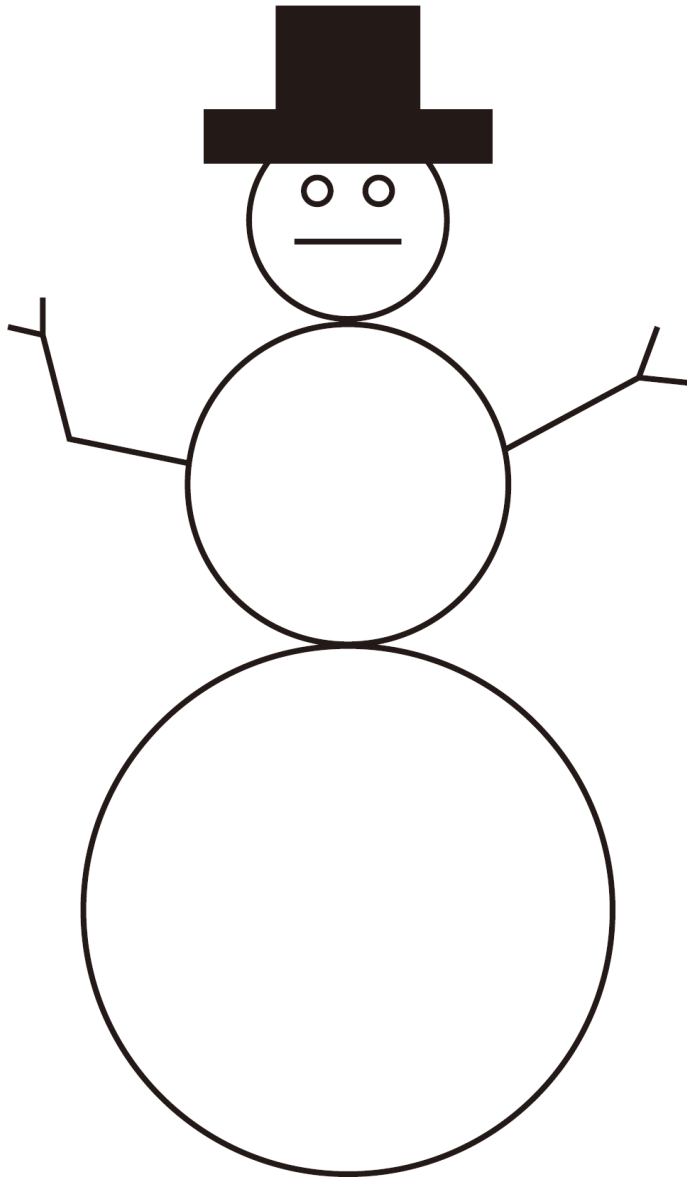


图 5-30 示例雪人

23. 海龟图形：矩形图案

在程序中写一个 `drawPattern` 函数，使用海龟图形库绘制如图 5-31 所示的矩形图案。`drawPattern` 函数应接受两个实参：一个指定图案的宽度，另一个指定高度。图 5-31 的例子展示了当宽度和高度相同时图案的外观。程序运行时，应询问用户图案的宽度和高度，然后，将这些值作为实参传递给 `drawPattern` 函数。

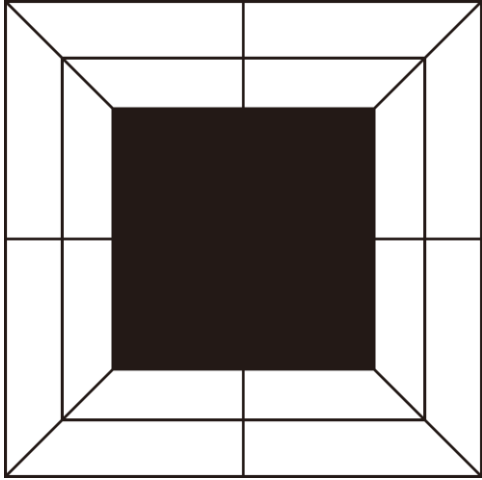


图 5-31 矩形图案

24. 海龟图形：棋盘

写一个海龟图形程序，使用本章编写的 `square` 函数和一个（或多个）循环来绘制如图 5-32 所示的棋盘图案。

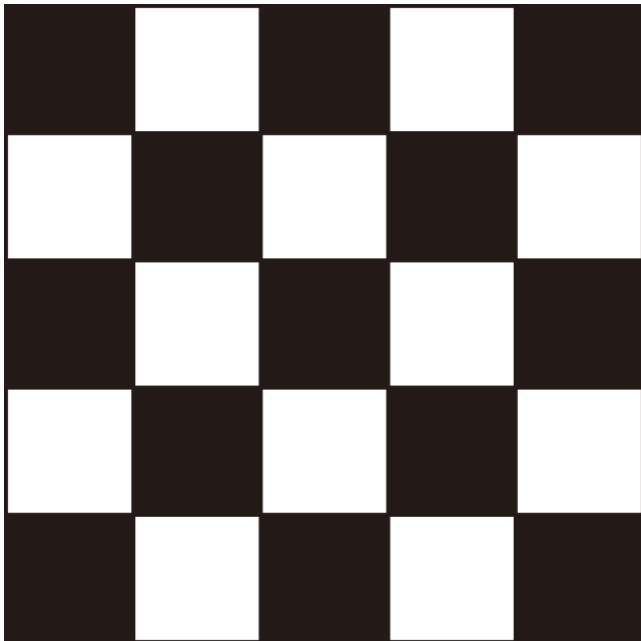


图 5-32 棋盘图案

25. 海龟图形：城市天际线

写一个海龟图形程序来绘制如图 5-33 所示的城市天际线。程序的总体任务是绘制夜空中的
一些城市建筑的轮廓。写执行以下任务的函数，对程序进行模块化：

- 绘制建筑物轮廓。
- 在建筑物上绘制一些窗户。
- 使用随机出现的点作为星星（确保星星出现在天空中，而不是在建筑物上）。



图 5-33 城市天际线

第 6 章 文件和异常

6.1 文件输入和输出简介

概念：当程序需要保存数据供以后使用时，它将数据写入文件。将来可以从文件中读取数据。

之前写的所有程序都要求在每次运行程序时重新输入数据，因为一旦程序停止运行，存储在 RAM 中的数据（由变量引用）就会消失。如果程序要保留数据供下一次使用，就必须使用一种对数据进行**持久化**的方法。为此，我们将数据保存到文件中，后者通常存储在计算机磁盘上。一旦数据保存到文件，程序停止运行后，数据仍将保留在文件中。存储在文件中的数据可以在将来任何时候检索和使用。

我们日常使用的大多数商业软件都允许将数据存储到文件中。下面列举了几个例子。

- **字处理软件**。字处理软件用于撰写信件、备忘录、报告和其他文档。文档保存到文件中，以便将来编辑和打印。
- **图像编辑软件**。图像编辑软件用于绘制和编辑图像，例如用数码相机拍摄的照片。用图像编辑软件创建或编辑的图像保存在文件中。
- **电子表格**。电子表格程序用于处理数字数据。可以在电子表格的单元格中插入数字和数学公式。电子表格保存到文件中，供将来使用。
- **游戏**。许多游戏将数据保存在文件中。例如，有的游戏将玩家名字及其分数保存在一个文件中。这些游戏通常按照得分从高到低的顺序排列玩家名字。有的游戏还允许将当前游戏状态保存在文件中，这样就可以随时退出游戏，将来从退出的位置继续游玩，而不必每次都重新开始。
- **Web 浏览器**。访问网页时，浏览器有时会在你的计算机上存储一个称为 cookie 的小文件。cookie 通常包含有关浏览会话的信息，例如购物车的内容。

日常工作中使用的程序广泛依赖于文件。例如，工资单程序将员工数据保存在文件中，库存程序将公司产品数据保存在文件中，会计系统将公司财务数据保存在文件中，等等。

程序员一般将在文件中保存数据的过程称为向文件“写入数据”。当数据被写入文件时，它从 RAM 中的变量复制到文件中，如图 6-1 所示。我们常用**输出文件**一词来描述向其写入数据的文件。之所以称为输出文件，是因为要将程序的输出存储到其中。

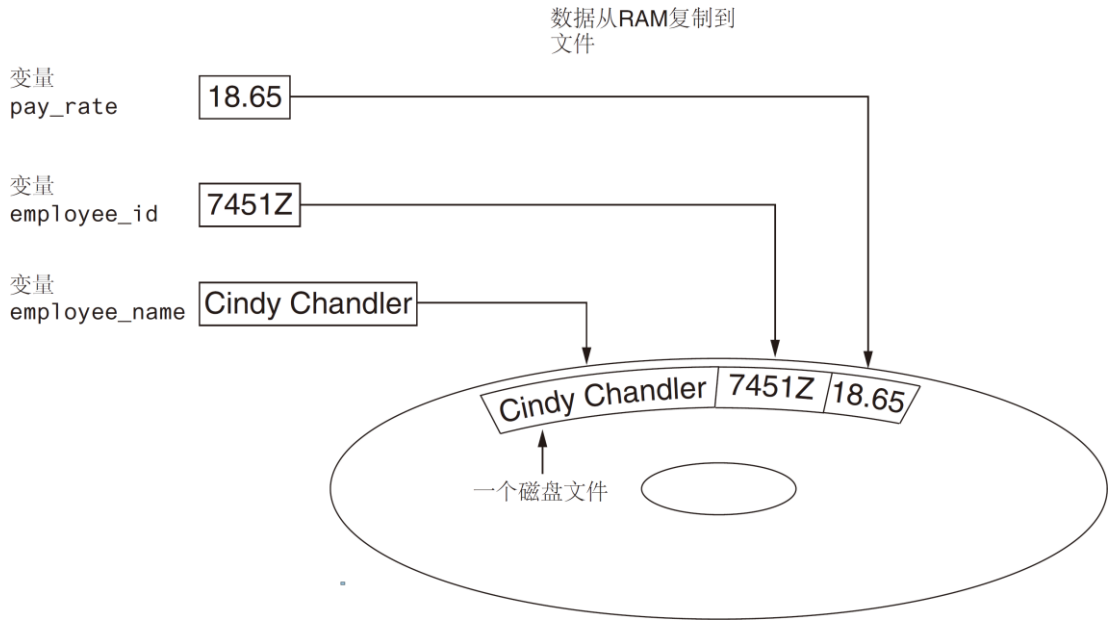


图 6-1 将数据写入文件

从文件获取数据的过程则称为从文件中“读取数据”。从文件中读取数据时，数据从文件复制到 RAM，并被变量引用，如图 6-2 所示。**输入文件**一词用于描述从中读取数据的文件。之所以称为输入文件，是因为程序从文件获取输入。

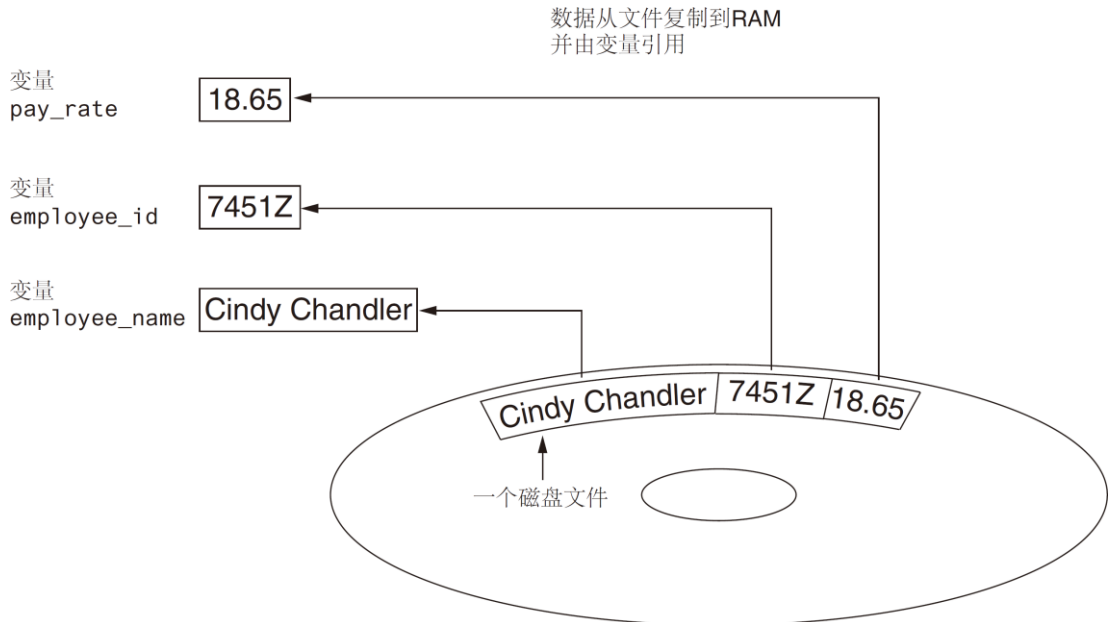


图 6-2 从文件读取数据

本章将讨论如何向文件写入数据和从文件读取数据。程序要想使用一个文件，必须采取三个步骤。

- **打开文件。** 在文件与程序之间建立连接。打开输出文件通常会在磁盘上创建文件，并允许程序向其写入数据。打开输入文件则允许程序从现有文件中读取数据。
- **处理文件。** 在这一步中，数据被写入文件（如果是输出文件）或者从文件中读取（如果是输入文件）。
- **关闭文件。** 当程序使用完文件后，必须关闭文件。关闭文件可以断开文件与程序的连接。

6.1.1 文件类型

一般来说有两种类型的文件：文本文件和二进制文件。**文本文件**包含使用 ASCII 或 Unicode 等方案编码为文本的数据。即使文件中包含数字，这些数字也是作为一系列字符存储在文件中的。因此，这种文件可在记事本等文本编辑器中打开并查看。**二进制文件**包含未转换为文本的数据。存储在二进制文件中的数据仅供程序读取。因此，不能用文本编辑器来查看二进制文件的内容。即使能查看，看到的也只是一堆“乱码”。

虽然 Python 同时支持文本文件和二进制文件，但本书只会涉及文本文件。因此，可以使用任意编辑器来检查程序所创建的文件。

6.1.2 文件访问方法

大多数编程语言提供两种不同的方法来访问存储在文件中的数据：顺序访问和直接访问。使用**顺序访问文件**，必须从文件头一直访问到文件尾。如果想读取存储在文件末尾的数据，那么必须先读取它前面的所有数据，而不能直接跳转到目标数据。这类似于老式磁带播放机。如果想听盒带上的最后一首歌，那么必须快进播放它之前的所有歌，或者慢慢地听。没办法直接跳到一首特定的歌。

使用**直接访问文件**（也称为**随机访问文件**），则可以直接跳转到文件中的任何数据，而无需读取之前的数据。这类似于 CD 或 MP3 播放器的工作方式。可以直接跳转到想听的任何歌曲。

本书将使用顺序访问文件。顺序访问文件很容易使用，可以用它来了解基本的文件操作。

6.1.3 文件名和文件对象

大多数计算机用户习惯于用文件名来标识文件。例如，使用字处理软件创建文档并保存为文件时，必须指定文件名。使用 Windows 文件资源管理器等工具查看磁盘内容时，会看到文件名列表。图 6-3 展示了名为猫.jpg，笔记.txt 和简历.docx 的三个文件在 Windows 中的

显示。



图 6-3 Windows 中的三个文件

每个操作系统都有自己的文件命名规则。许多系统支持使用文件扩展名，即出现在文件名末尾，最后一个句点后的短字符序列。例如，图 6-3 的文件扩展名为 .jpg、.txt 和 .docx。扩展名通常表示文件中存储的数据类型。例如，.jpg 扩展名通常表示文件包含根据 JPEG 图像标准压缩的图像，.txt 扩展名通常表示文件包含纯文本，而 .docx 扩展名（以及 .doc）通常表示该文件包含 Microsoft Word 文档。

为了处理计算机磁盘上的文件，程序必须先在内​​存中创建一个文件对象。**文件对象**是与一个特定文件关联的对象，它为程序提供了处理该文件的方法。在程序中，我们用一个变量来引用文件对象，并通过该变量来执行对文件的任何操作。图 6-4 说明了这个概念。

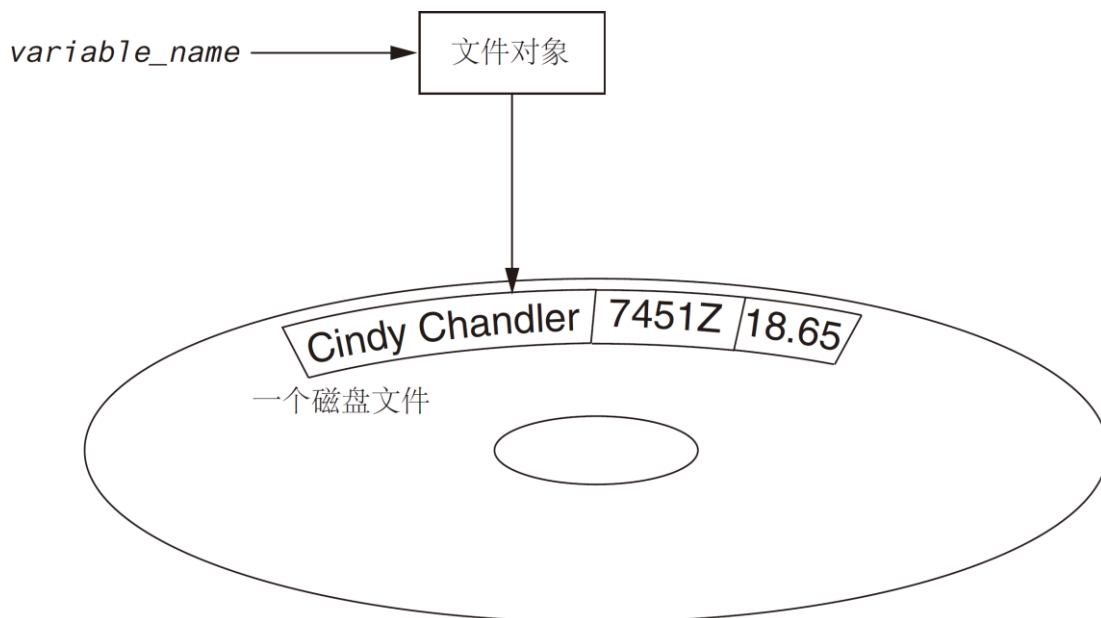


图 6-4 变量名引用与文件关联的文件对象

6.1.4 打开文件

在 Python 中，我们使用 `open` 函数打开文件。`open` 函数创建一个文件对象，并将它与磁盘上的文件关联。下面是 `open` 函数使用的常规格式。

```
file_variable = open(filename, mode)
```

在这个常规格式中：

- `file_variable` 是引用文件对象的变量的名称。
- `filename` 是指定了文件名的字符串。
- `mode` 是代表文件打开模式（读、写等）的一个字符串。表 6-1 列示了可以用来指定模式的三个字符串（还有其他更复杂的模式）。表 6-1 中的是本书将使用的模式。

表 6-1 部分 Python 文件模式

模式	说明
'r'	以只读模式打开文件。文件不能修改或写入。
'w'	以写入模式打开文件。如果文件已存在，将擦除其内容。如果文件不存在，就新建一个。
'a'	以追加模式打开文件。所有写入文件的数据都将追加到文件末尾。文件不存在就新建一个。

例如，假设 `customers.txt` 文件包含客户数据，我们希望打开该文件进行读取，下面展示了如何调用 `open` 函数。


```
customer_file = open('customers.txt', 'r')
```

执行该语句后，名为 `customers.txt` 的文件将被打开，`customer_file` 变量将引用一个文件对象，之后可以用它从文件中读取数据。

假设要创建一个名为 `sales.txt` 的文件并向其中写入数据，下面展示了如何调用 `open` 函数。

```
sales_file = open('sales.txt', 'w')
```

执行该语句后，将创建名为 `sales.txt` 的文件，`sales_file` 变量将引用一个文件对象，之后可以用它向文件写入数据。

 警告：记住，使用 'w' 模式会在磁盘上创建文件。如果指定的文件已经存在，那么文件

现有的内容会被删除。

6.1.5 指定文件位置

将不包含路径的一个文件名作为实参传递给 `open` 函数时，Python 解释器假定文件的位置与程序位置相同。例如，假设一个程序位于 Windows 计算机上的以下文件夹中。

```
C:\Users\Zhou Jing\Documents\Python 中文版代码
```

当程序运行并执行以下语句时，会在同一文件夹下创建文件 `test.txt`。

```
test_file = open('test.txt', 'w')
```

要打开位于不同位置的文件，可以在传递给 `open` 函数的实参中指定路径和文件名。以字符串形式指定路径时（特别是在 Windows 计算机上），请确保在字符串前加上字母 `r`，如下例所示。

```
test_file = open(r'C:\Users\Zhou Jing\temp\test.txt', 'w')
```

该语句在文件夹 `C:\Users\Zhou Jing\temp` 中创建文件 `test.txt`。`r` 前缀表示字符串是**原始字符串**。这使 Python 解释器将反斜杠字符视为字面意义的反斜杠。如果没有 `r` 前缀，解释器就会认为反斜杠字符是转义序列的一部分，从而引起错误。

6.1.6 向文件写入数据

本书之前已经使用了几个 Python 库函数，甚至编写了自己的函数。现在，我们将向你介绍另一种类型的函数，即方法。**方法**是属于一个对象，并使用该对象来执行某些操作的函数。打开文件后，可以使用文件对象提供的方法来对文件执行操作。

例如，文件对象有一个名为 `write` 的方法，用于向文件写入数据。下面是该方法的常规调用格式。

```
file_variable.write(string)
```

其中，`file_variable` 是引用了文件对象的变量，`string` 是要向文件写入的字符串。文件的打开模式必须支持写入（`'w'`或`'a'`模式），否则会出错。

假设 `customer_file` 引用了一个文件对象，而且该文件是以 `'w'` 模式打开的。下例将字符串 `'Charles Pace'` 写入文件：

```
customer_file.write('Charles Pace')
```

下面的代码展示了另一个例子：

```
name = 'Charles Pace'  
customer_file.write(name)
```

第二个语句将 `name` 变量引用的值写入与 `customer_file` 关联的文件中。在本例中，将向文件中写入字符串 `'Charles Pace'`。虽然这些例子展示的是将字符串写入文件，但也可以写入数值。

程序结束对文件的处理后，应该显式地关闭文件。关闭文件可以断开程序与文件的连接。在某些系统中，未关闭的输出文件可能导致数据丢失。之所以会发生这种情况，是因为向文件写入的数据首先会写入一个**缓冲区**，这是内存中的一个小的“保留区”。缓冲区满时，系统将缓冲区的内容正式写入文件。这种技术提高了系统的性能，因为将数据写入内存比写入磁盘更快。关闭输出文件的过程会强制将缓冲区中任何未保存的数据写入文件。

在 Python 中，我们使用文件对象的 `close` 方法来关闭文件。例如，以下语句关闭与 `customer_file` 关联的文件。

```
customer_file.close()
```

程序 6-1 展示了一个完整的 Python 程序，它打开一个输出文件，向其中写入数据，然后关闭它。

程序 6-1 (file_write.py)

```
1  # 这个程序向文件写入
2  # 三行数据。
3  def main():
4      # 打开一个名为 philosophers.txt 的文件
5      outfile = open('philosophers.txt', 'w')
6
7      # 向文件中写入三位哲学家
8      # 的名字。
9      outfile.write('John Locke\n')
10     outfile.write(' David Hume\n')
11     outfile.write('Edmund Burke\n')
12
13     # 关闭文件
14     outfile.close()
15
16 # 调用 main 函数
17 if __name__ == '__main__':
18     main()
```

第 5 行使用 `'w'` 模式打开 `philosophers.txt` 文件。这导致创建文件并打开以准备写入。该语句还会在内存中创建一个文件对象，并将该对象赋给 `outfile` 变量。

第 9 行~第 11 行的语句向文件写入三个字符串。第 9 行写入字符串 `'John Locke\n'`，第 10 行写入字符串 `'David Hume\n'`，第 11 行写入字符串 `'Edmund Burke\n'`。第 14 行关闭文件。程序运行后，如图 6-5 所示的三个数据项会被写入 `philosophers.txt` 文件。

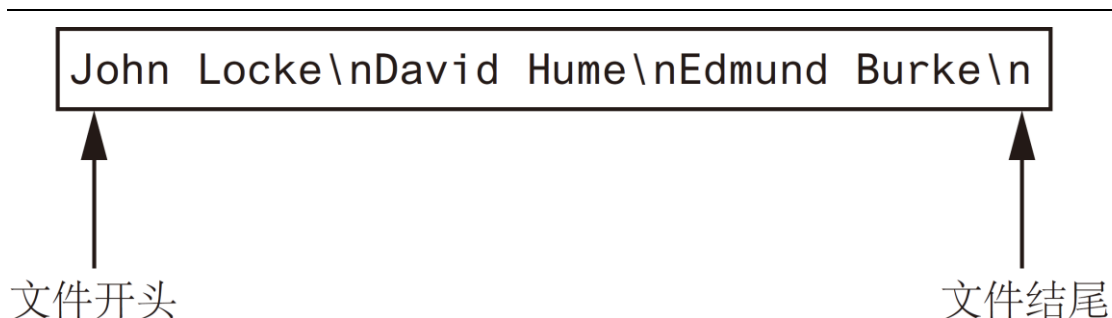


图 6-5 philosophers.txt 文件的内容

注意，每个写入文件的字符串都以\n 结尾，你应该记得这是换行符的转义序列。\\n 不仅分隔了文件中的条目，而且在文本编辑器中查看时，还会使每个条目都显示在单独的一行中。例如，图 6-6 展示了在“记事本”程序中显示的 philosophers.txt 文件。

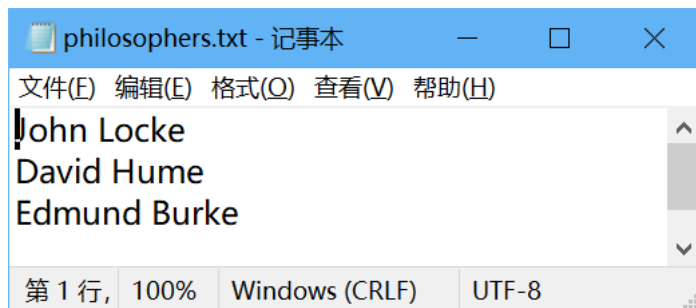


图 6-6 在“记事本”中查看 philosophers.txt 的内容

6.1.7 从文件读取数据

如果文件已经打开供读取（使用'r'模式），那么可以使用文件对象的 read 方法将其全部内容读入内存。调用 read 方法时，它以字符串形式返回文件内容。例如，程序 6-2 展示了如何使用 read 方法读取之前创建的 philosophers.txt 文件的内容。

程序 6-2 (file_read.py)

```
1 # 这个程序读取并显示 philosophers.txt
2 # 文件的内容。
3 def main():
4     # 打开一个名为 philosophers.txt 的文件
5     infile = open('philosophers.txt', 'r')
6
7     # 读取文件内容
8     file_contents = infile.read()
```

```

9
10     # 关闭文件
11     infile.close()
12
13     # 打印读入内存
14     # 的数据。
15     print(file_contents)
16
17 # 调用 main 函数
18 if __name__ == '__main__':
19     main()

```

程序输出

```

John Locke
David Hume
Edmund Burke

```

第 5 行的语句使用 'r' 模式打开 `philosophers.txt` 文件进行读取。它还创建了一个文件对象，并将该对象赋给 `infile` 变量。第 8 行调用 `infile.read` 方法读取文件内容。文件内容以字符串的形式读入内存，并赋给 `file_contents` 变量，如图 6-7 所示。第 15 行的语句打印该变量引用的字符串。

`file_contents` → `John Locke\nDavid Hume\nEdmund Burke\n`

图 6-7 `file_contents` 变量引用了从文件中读取的字符串

尽管 `read` 方法允许用一个语句轻松读取文件的全部内容，但许多程序都需要一次读取并处理文件中存储的一个数据项。例如，假设文件中包含一系列销售金额，需要写一个程序来计算它们的总金额，那么可以每次从文件中读取一个销售金额，然后将它加到一个累加器上。

在 Python 中，可以使用 `readline` 方法从文件中读取一行（一行是指以 `\n` 结尾的一个字符串）。该方法将整行作为一个字符串返回，基保包括 `\n`。程序 6-3 展示了如何使用 `readline` 方法逐行读取 `philosophers.txt` 文件的内容。

程序 6-3 (`line_read.py`)

```

1     # 这个程序逐行读取 philosophers.txt
2     # 文件的内容。
3     def main():
4         # 打开一个名为 philosophers.txt 的文件
5         infile = open('philosophers.txt', 'r')
6

```

```
7     # 从文件中读取三行
8     line1 = infile.readline()
9     line2 = infile.readline()
10    line3 = infile.readline()
11
12    # 关闭文件
13    infile.close()
14
15    # 打印读入内存
16    # 的数据。
17    print(line1)
18    print(line2)
19    print(line3)
20
21    # 调用 main 函数
22    if __name__ == '__main__':
23        main()
```

程序输出

John Locke

David Hume

Edmund Burke

在查看代码之前，请注意输出结果中的每一行后面都多了一个空行。这是因为从文件中读取的每个数据项都以换行符（\n）结束。稍后你将学习如何删除换行符。

第 5 行的语句使用 'r' 模式打开 `philosophers.txt` 文件进行读取。它还创建了一个文件对象，并将该对象赋给 `infile` 变量。打开一个文件进行读取时，会在内部为该文件维护一个特殊的值，称为**读取位置**。文件的读取位置标志着将从文件中读取的下一个数据项的位置。初始读取位置被设为文件的起始位置。执行第 5 行的语句后，`philosophers.txt` 文件的读取位置如图 6-8 所示，这是初始读取位置。

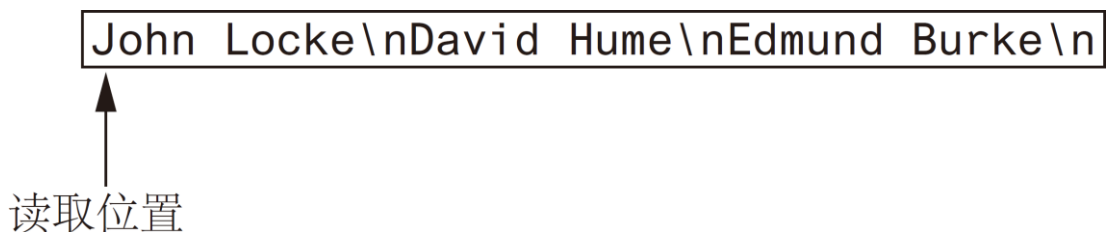


图 6-8 初始读取位置

第 8 行语句调用 `infile.readline` 方法从文件中读取第一行。以字符串形式返回的行被赋

给 `line1` 变量。这个语句执行后，`line1` 变量将被赋值为字符串 `'John Locke/n'`。此外，文件的读取位置将前进到文件中的下一行，如图 6-9 所示。

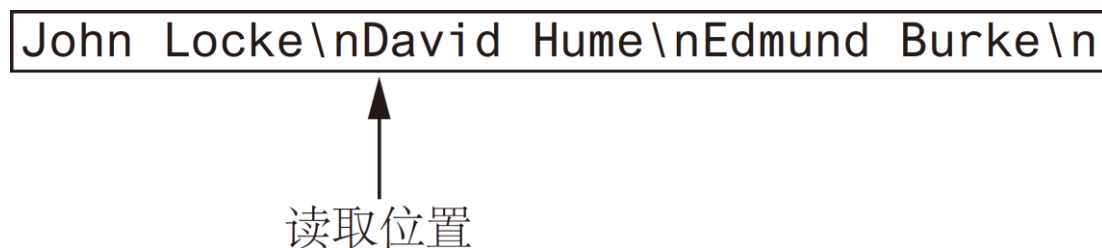


图 6-9 读取位置前进到下一行

然后，第 9 行的语句从文件中读取下一行，并将其赋给 `line2` 变量。在这个语句执行之后，`line2` 变量将引用字符串 `'David Hume/n'`。文件的读取位置将前进到文件的下一行，如图 6-10 所示。

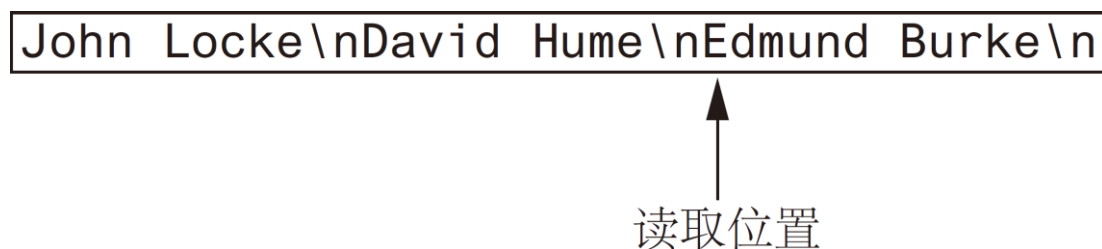


图 6-10 读取位置前进到下一行

然后，第 10 行的语句从文件中读取下一行，并将其赋给 `line3` 变量。这个语句执行之后，`line3` 变量将引用字符串 `'Edmund Burke/n'`。文件的读取位置将前进到文件末尾，如图 6-11 所示。图 6-12 展示了在执行了这些语句之后，`line1`，`line2` 和 `line3` 变量以及它们引用的字符串。

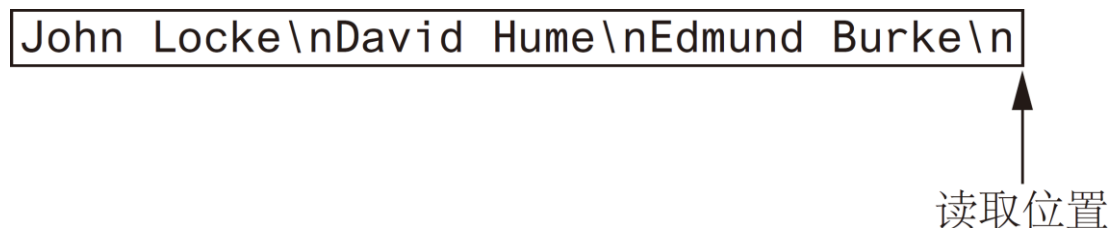


图 6-11 读取位置前进到文件末尾

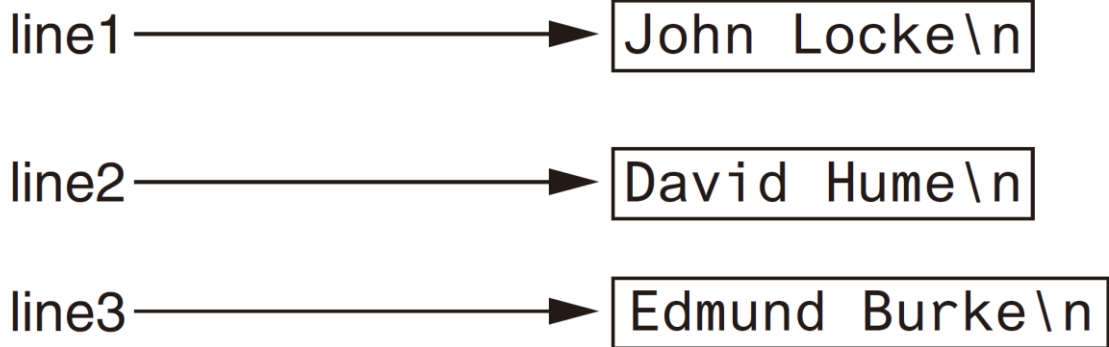



图 6-12 line1, line2 和 line3 变量引用的字符串

第 13 行关闭文件。第 17 行~第 19 行显示 line1, line2 和 line3 变量的内容。

 注意：如果文件的最后一行不以\n 结束，那么 readline 方法将返回不带\n 的一行。

6.1.8 将换行符连接到字符串

程序 6-1 向文件中写入了三个字符串字面值，每个字符串字面值都以转义序列\n 结束。但在大多数情况下，向文件中写入的数据项并不是字符串字面值，而是内存中由变量引用的值。例如，如果程序提示用户输入数据，并将数据写入文件，那么就属于这种情况。

当程序将用户输入的数据写入文件时，通常需要在写入之前先连接好一个\n 转义序列。这样可以确保每个数据项都被写入文件中单独的一行。程序 6-4 演示了具体如何做。

程序 6-4 (write_names.py)

```
1 # 这个程序从用户处获取三个名字，
2 # 并将它们写入文件。
3
4 def main():
5     # 获取三个名字
6     print('输入三个朋友的名字。')
7     name1 = input('朋友#1: ')
8     name2 = input('朋友#2: ')
9     name3 = input('朋友#3: ')
10
11     # 打开 friends.txt 文件
12     myfile = open('friends.txt', 'w')
13
14     # 将名字写入文件
15     myfile.write(name1 + '\n')
```

```

16     myfile.write(name2 + '\n')
17     myfile.write(name3 + '\n')
18
19     # 关闭文件
20     myfile.close()
21     print('名字已经写入 friends.txt 文件。')
22
23     # 调用 main 函数
24     if __name__ == '__main__':
25         main()

```

程序输出（用户输入的内容加粗）

```

输入三个朋友的名字。
朋友#1: Joe 
朋友#2: Rose 
朋友#3: Geri 
名字已经写入 friends.txt 文件。

```

第 7 行~第 9 行提示用户输入三个名字，这些名字被赋给变量 `name1`，`name2` 和 `name3`。第 12 行打开一个名为 `friends.txt` 的文件。然后，第 15 行~第 17 行向文件写入用户输入的名字，每个名字都连接了一个 `'\n'`。因此，当写入文件时，每个名字都会加上转义序列 `\n`。如图 6-13 所示，文件中已经存储了用户在示例运行中输入的名字。

```

Joe\nRose\nGeri\n

```

图 6-13 `friends.txt` 文件



提示：程序 6-4 的第 15 行~17 行很容易用 `f` 字符串来写，如下所示。

```

myfile.write(f'{name1}\n')
myfile.write(f'{name2}\n')
myfile.write(f'{name3}\n')

```

6.1.9 读取字符串并去掉换行符

有的时候，从 `readline` 方法返回的字符串末尾出现的 `\n` 会引起一些复杂的问题。例如，你是否注意到在程序 6-3 的示例输出中，输出的每一行后面都打印了一个空行？这是因为第 17 行~第 19 行打印的每个字符串都以 `\n` 转义序列结束。打印这种字符串时，`\n` 会导致额外的空行。

在文件中，`\n`有一个必要的作用，即分隔文件中存储的数据项。但在许多情况下，当字符串从文件中读出后，我们希望将它的`\n`去掉。Python中的每个字符串都有一个名为`rstrip`的方法，它可以去掉字符串末尾的特定字符。之所以命名为`rstrip`，是因为它从字符串的右侧删除字符。以下示例代码展示了如何使用`rstrip`方法。

```
name = 'Joanne Manchester\n'  
name = name.rstrip('\n')
```

第一个语句将字符串'`Joanne Manchester\n`'赋给`name`变量。注意，字符串以`\n`转义序列结束。第二个语句调用了`name.rstrip('\n')`方法。该方法返回一个末尾没有`\n`的`name`字符串的拷贝。该字符串会被重新赋给`name`变量。结果是`name`所引用的字符串现在已经去掉了尾部的`\n`。

程序 6-5 是另一个读取并显示 `philosophers.txt` 文件内容的程序。程序先用 `rstrip` 方法去除从文件中读取的字符串末尾的`\n`，然后才在屏幕上显示。因此，额外的空行不会出现。

程序 6-5 (`strip_newline.py`)

```
1  # 这个程序逐行读取 philosophers.txt  
2  # 文件的内容。  
3  def main():  
4      # 打开一个名为 philosophers.txt 的文件  
5      infile = open('philosophers.txt', 'r')  
6  
7      # 从文件中读取三行  
8      line1 = infile.readline()  
9      line2 = infile.readline()  
10     line3 = infile.readline()  
11  
12     # 去除每个字符串的\n 字符  
13     line1 = line1.rstrip('\n')  
14     line2 = line2.rstrip('\n')  
15     line3 = line3.rstrip('\n')  
16  
17     # 关闭文件  
18     infile.close()  
19  
20     # 打印读入内存的数据  
21     print(line1)  
22     print(line2)  
23     print(line3)  
24  
25     # 调用 main 函数  
26     if __name__ == '__main__':  
27         main()
```

程序输出

```
John Locke
David Hume
Edmund Burke
```

6.1.10 向现有文件追加数据

以 'w' 模式打开一个输出文件时，如果磁盘上已经存在同名文件，那么现有文件会被删除，并新建一个同名的空文件。有的时候，我们希望保留现有文件，并将新数据追加到当前内容上。向文件“追加”数据意味着将新数据写到文件中现有数据的末尾。

在 Python 中，可以使用 'a' 模式以追加（append）模式打开一个输出文件，这意味着：

- 如果文件已经存在，那么它的内容不会被删除。如果文件不存在，它将被创建。
- 向文件写入数据时，数据会写入文件当前内容的末尾。

例如，假定文件 `friends.txt` 包含以下名字，每个名字单独一行。

```
Joe
Rose
Geri
```

以下代码打开文件，并为现有内容附加额外的数据。

```
myfile = open('friends.txt', 'a')
myfile.write('Matt\n')
myfile.write('Chris\n')
myfile.write('Suze\n')
myfile.close()
```

程序运行后，`friends.txt` 文件将包含以下数据。

```
Joe
Rose
Geri
Matt
Chris
Suze
```

6.1.11 写入和读取数值数据

字符串可以直接用 `write` 方法写入文件。但是，数字在写入之前必须先转换为字符串。在 Python 中，可以利用一个名为 `str` 的内置函数将数值转换为字符串。例如，假设变量 `num` 被赋值为 99，那么表达式 `str(num)` 将返回字符串 '99'。

程序 6-6 展示了如何使用 `str` 函数将数字转换为字符串，并将转换后的字符串写入文件。

程序 6-6 (write_numbers.py)

```
1  # 这个程序演示了将数字写入
2  # 文本文件之前，如何先将其
3  # 转换为字符串。
4
5  def main():
6      # 打开一个文件来写入
7      outfile = open('numbers.txt', 'w')
8
9      # 从用户处获取三个数字
10     num1 = int(input('输入第一个数: '))
11     num2 = int(input('输入第二个数: '))
12     num3 = int(input('输入第三个数: '))
13
14     # 将数字写入文件
15     outfile.write(str(num1) + '\n')
16     outfile.write(str(num2) + '\n')
17     outfile.write(str(num3) + '\n')
18
19     # 关闭文件
20     outfile.close()
21     print('数据已写入 numbers.txt 文件。')
22
23 # 调用 main 函数
24 if __name__ == '__main__':
25     main()
```

程序输出 (用户输入的内容加粗)

```
输入第一个数: 22 
输入第二个数: 14 
输入第三个数: -99 
数据已写入 numbers.txt 文件。
```

第 7 行打开 `numbers.txt` 文件。然后，第 10 行~第 12 行提示用户输入三个数字，并将其赋给变量 `num1`、`num2` 和 `num3`。

来仔细看看第 15 行的语句，它将 `num1` 引用的值写入文件。

```
outfile.write(str(num1) + '\n')
```

表达式 `str(num1) + '\n'` 将 `num1` 引用的值转换为字符串，并将转义序列 `\n` 连接到字符串上。在程序的示例运行中，用户输入 `22` 作为第一个数字，因此这个表达式生成了字符串 `'22\n'`。结果，字符串 `'22\n'` 被写入文件。

第 16 行和第 17 行执行类似的操作，将 `num2` 和 `num3` 引用的值写入文件。这些语句执行后，文件中将包含如图 6-14 所示的值。图 6-15 展示了在“记事本”中查看的文件。

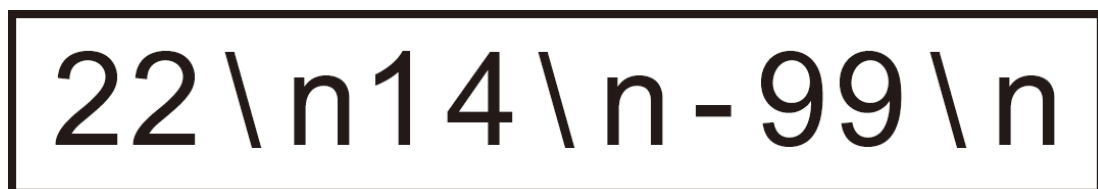


图 6-14 `numbers.txt` 文件的内容

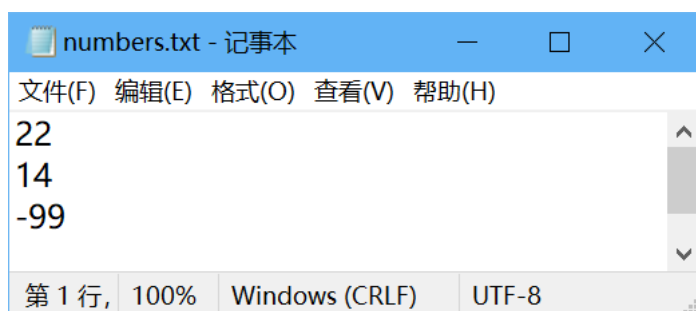


图 6-15 在“记事本”中查看的 `numbers.txt` 文件

从文本文件中读取数字时，总是以字符串的形式读取。例如，假设一个程序使用以下代码从程序 6-6 创建的 `numbers.txt` 文件中读取第一行。

```
1 infile = open('numbers.txt', 'r')
2 value = infile.readline()
3 infile.close()
```

第 2 行的语句使用 `readline` 方法从文件中读取一行。这个语句执行后，`value` 变量将引用字符串 `'22/n'`。如果打算用 `value` 变量执行数学运算，那么可能会出问题，因为不能在字符串上进行数学运算。在这种情况下，必须将字符串转换为数值类型。

第 2 章讲过，Python 提供了内置函数 `int` 将字符串转换为整数，内置函数 `float` 将字符串转换为浮点数。例如，可以像下面这样修改前面的代码。

```
1 infile = open('numbers.txt', 'r')
2 string_input = infile.readline()
3 value = int(string_input)
4 infile.close()
```

第 2 行的语句从文件中读取一行并将其赋给 `string_input` 变量。结果，`string_input` 将引用字符串 `'22/n'`。然后，第 3 行的语句使用 `int` 函数将 `string_input` 转换为整数，并将结果赋给 `value`。这个语句执行后，`value` 变量将引用整数 22。记住，`int` 和 `float` 函数都

会忽略作为实参传递的字符串末尾的\n。

上述代码演示了使用 `readline` 方法从文件中读取字符串，然后使用 `int` 函数将字符串转换为整数的步骤。但在许多情况下，代码还可以简化。更好的方法是在一个语句中完成从文件中读取字符串并执行转换的操作，如下所示。

```
1 infile = open('numbers.txt', 'r')
2 value = int(infile.readline())
3 infile.close()
```

注意，第 2 行对 `readline` 方法的调用被用作 `int` 函数的实参。代码的工作方式是：调用 `readline` 方法，返回一个字符串。字符串被传给 `int` 函数，`int` 函数将其转换为整数。结果被赋给 `value` 变量。

程序 6-7 是一个更完整的演示。它读取 `numbers.txt` 文件的内容，将其转换为整数并计算它们的和。

程序 6-7 (`read_numbers.py`)

```
1 # 这个程序演示了从文件读取
2 # 的数字在用于数学运算之前，
3 # 如何先从字符串转换为数字。
4
5 def main():
6     # 打开一个文件来读取
7     infile = open('numbers.txt', 'r')
8
9     # 从文件中读取三个数字
10    num1 = int(infile.readline())
11    num2 = int(infile.readline())
12    num3 = int(infile.readline())
13
14    # 关闭文件
15    infile.close()
16
17    # 三个数字相加
18    total = num1 + num2 + num3
19
20    # 显示每个数字和它们的总和
21    print(f'数字分别为: {num1}, {num2}, {num3}')
22    print(f'它们的总和为: {total}')
23
24 # 调用 main 函数
25 if __name__ == '__main__':
26     main()
```

程序输出

数字分别为：22, 14, -99
它们的总和为：-63

检查点

- 6.1 什么是输出文件？
- 6.2 什么是输入文件？
- 6.3 程序为了使用文件必须采取哪三个步骤？
- 6.4 通常有哪两种类型的文件？这两种文件有什么区别？
- 6.5 有哪两种文件访问类型？两种类型有什么区别？
- 6.6 在写执行文件操作的程序时，代码中要用到哪两个与文件相关的名称？
- 6.7 如果一个文件已经存在，尝试将其作为输出文件打开（使用'w'模式）时会发生什么？
- 6.8 打开文件的目的是什么？
- 6.9 关闭文件的目的是什么？
- 6.10 什么是文件的读取位置？打开输入文件时，初始读取位置在哪里？
- 6.11 如果要向文件中写入数据，但又不想删除文件中已有的内容，那么应该以何种模式打开文件？向这样的文件写入数据时，数据在文件的什么位置写入？

6.2 使用循环来处理文件

概念：文件中通常存储着大量数据，程序通常使用循环来处理文件中的数据。

视频讲解：Using Loops to Process Files

虽然有的程序只用文件存储少量数据，但大多数文件都需要存储大量数据。当程序使用文件来写入或读取大量数据时，通常会使用循环。例如，程序 6-8 从用户处获取多天的销售额，并将这些销售额写入一个名为 `sales.txt` 的文件。用户指定需要输入销售数据的天数。在程序示例运行中，用户输入了 5 天的销售金额。图 6-16 展示了 `sales.txt` 文件的内容，其中包含用户在示例运行中输入的数据。

程序 6-8 (write_sales.py)

```
1 # 这个程序提示用户输入销售金额，
2 # 并将这些金额写入 sales.txt 文件。
3
4 def main():
5     # 获取要输入数据的天数
6     num_days = int(input('你想输入多少天' +
7                          '的销售额? '))
8
9     # 打开一个名为 sales.txt 的新文件
10    sales_file = open('sales.txt', 'w')
11
12    # 获取每天的销售额，
13    # 并将其写入文件。
14    for count in range(1, num_days + 1):
15        # 获取一天的销售额
16        sales = float(input(
17            f'输入第{count}天的销售额: '))
18
19        # 将销售额写入文件
20        sales_file.write(f'{sales}\n')
21
22    # 关闭文件
23    sales_file.close()
24    print('数据已写入 sales.txt 文件。')
25
26 # 调用 main 函数
27 if __name__ == '__main__':
28     main()
```

程序输出 (用户输入的内容加粗)

```
你想输入多少天的销售额? 5 
输入第 1 天的销售额: 1000.0 
输入第 2 天的销售额: 2000.0 
输入第 3 天的销售额: 3000.0 
输入第 4 天的销售额: 4000.0 
输入第 5 天的销售额: 5000.0 
数据已写入 sales.txt 文件。
```

```
1000.0\n2000.0\n3000.0\n4000.0\n5000.0\n
```

图 6-16 sales.txt 文件的内容

6.2.1 使用循环读取文件并检测文件尾


程序经常需要在不知道文件中存储了多少数据项数的情况下读取文件内容。例如，程序 6-8 创建的 sales.txt 文件可以存储任意数量的数据项，因为程序会先询问用户要输入多少天

的销售额。如果用户输入 5，那么程序将获取 5 个销售额并写入文件。如果用户输入 100，那么程序将获取 100 个销售额并写入文件。

如果要写一个程序来处理文件中的所有数据项，但事先不知道具体有多少个，这就会造成一个问题。例如，假设需要写一个程序，读取 `sales.txt` 文件中的所有金额并计算其总额。虽然可以使用一个循环来读取文件中的数据项，但是需要一种方法来知道何时到达文件尾。

在 Python 中，当 `readline` 方法试图读取超过文件末尾的内容时，它会返回一个空字符串（''）。因此，我们可以写一个判断何时到达文件尾的 `while` 循环。下面是用伪代码来写的常规算法。图 6-17 展示了该算法的流程图。

```
打开文件
使用 readline 读取文件的第一行
当 (while) 从 readline 返回的值不是空字符串时:
    处理刚才从文件中读取的数据
    使用 readline 从文件中读取下一行。
关闭文件
```

 **注意：**在这个算法中，我们在进入 `while` 循环之前调用了一次 `readline` 方法。这个调用的目的是获取文件中的第一行，以便循环对其进行测试。这个初始的读取操作称为“预读”。

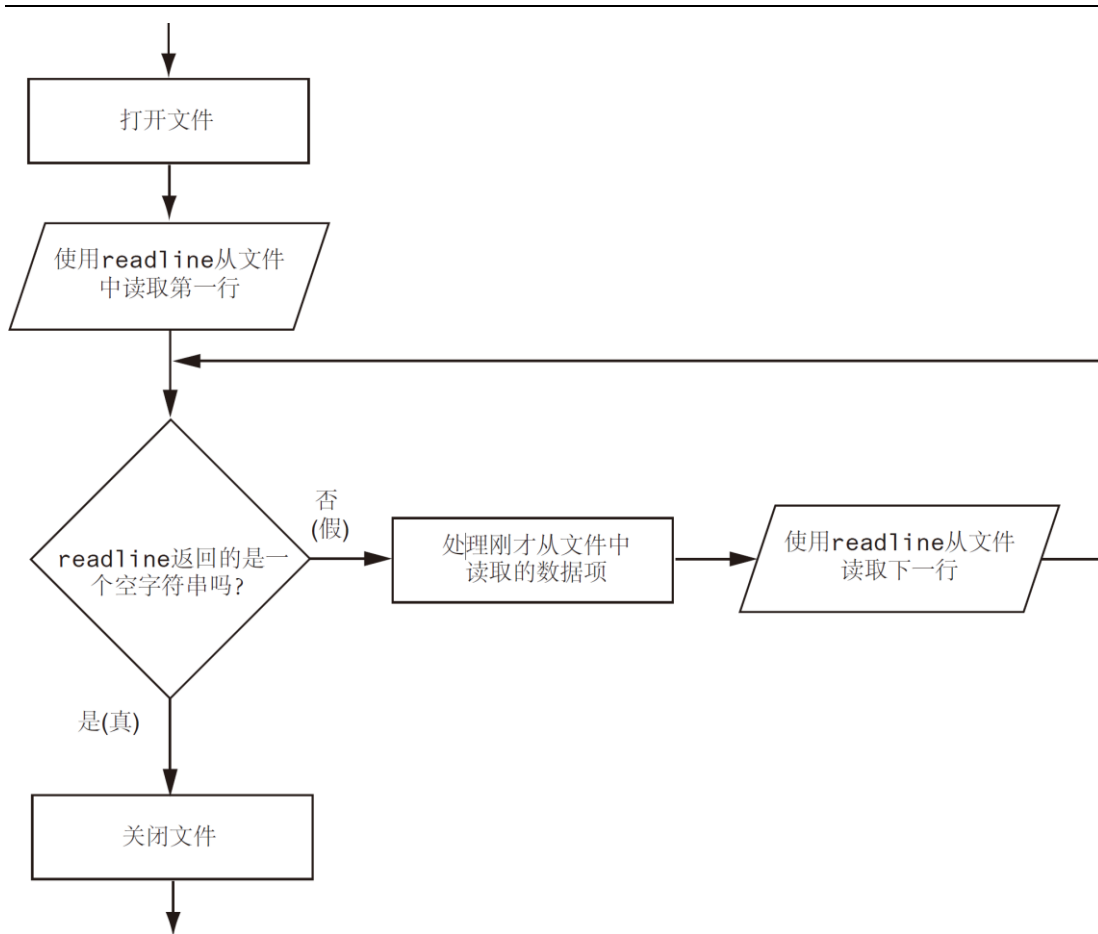


图 6-17 检测文件尾的常规逻辑

程序 6-9 演示了如何在代码中实现这个过程。该程序读取并显示 sales.txt 文件中的所有值。

程序 6-9 (read_sales.py)

```
1 # 这个程序读取 sales.txt 文件
2 # 中的所有值。
3
4 def main():
5     # 打开 sales.txt 文件以读取
6     sales_file = open('sales.txt', 'r')
7
8     # 从文件中读取第一行，但先不要
```

```

9     # 转换成数字。我们仍然需要测试
10    # 它是不是空字符串。
11    line = sales_file.readline()
12
13    # 只要 readline 返回的不是空字符串，
14    # 就继续处理。
15    while line != '':
16        # 将 line 转换为 float
17        amount = float(line)
18
19        # 格式化并显示金额
20        print(f'{amount:.2f}')
21
22        # 读取下一行
23        line = sales_file.readline()
24
25    # 关闭文件
26    sales_file.close()
27
28    # 调用 main 函数
29    if __name__ == '__main__':
30        main()

```

程序输出

```

1000.00
2000.00
3000.00
4000.00
5000.00

```

6.2.2 使用 for 循环读取行

在前面的例子中，你看到了当到达文件尾时，`readline` 方法会返回空字符串。大多数编程语言提供了类似的检测**文件尾**（End of File, EOF）的技术。如果计划学习 Python 之外的编程语言，那么了解如何构造这种逻辑是很重要的。

Python 语言还允许写一个 `for` 循环来自动读取文件中的行，而不需要测试任何特殊条件来判断是否到达文件尾。这个循环不需要“预读”操作。在到达文件尾时，它会自动停止。如果只想一个接一个地读取文件中的行，那么这种技术比写一个明确测试文件尾的 `while` 循环更简单、更优雅。下面是这种循环的常规格式。

```

for variable in file_object:
    语句
    语句
    ...

```

其中, *variable* 是变量名, *file_object* 是引用了文件对象的变量。循环将遍历文件中的每一行。在循环的第一次迭代中, *variable* 将引用文件中的第一行 (作为字符串), 在循环的第二次迭代中, *variable* 将引用第二行, 以此类推。

程序 6-10 对此进行了演示, 它读取并显示 sales.txt 文件中的所有数据项。

程序 6-10 (read_sales2.py)

```
1 # 这个程序使用 for 循环来读取
2 # sales.txt 文件中的所有值。
3
4 def main():
5     # 打开 sales.txt 文件以读取
6     sales_file = open('sales.txt', 'r')
7
8     # 从文件中读取所有行
9     for line in sales_file:
10        # 将 line 转换为 float
11        amount = float(line)
12        # 格式化并显示金额
13        print(f'{amount:.2f}')
14
15    # 关闭文件
16    sales_file.close()
17
18 # 调用 main 函数
19 if __name__ == '__main__':
20    main()
```

程序输出

```
1000.00
2000.00
3000.00
4000.00
5000.00
```

聚光灯：文件处理



凯文是一名自由视频制作人, 为当地企业制作电视广告。制作广告时, 他通常会拍摄几个短片。之后, 他将这些短片组合在一起, 制作成最终的广告片。他要求你写以下两个程序。

- 一个程序允许他输入项目中每个短片的运行时间 (以秒为单位)。这些运行时间被保存到一个文件中。
- 另一个程序读取文件内容, 显示每个短片的运行时间, 最后显示所有短片的总运行时

间。

下面是第一个程序的常规算法（伪代码）。

获取项目中视频的数量。
打开输出文件。
对于（for）项目中的每个视频：
 获取视频的运行时间。
 将运行时间写入文件。
关闭文件。

程序 6-11 展示了第一个程序的代码。

程序 6-11 (save_running_times.py)

```
1  # 该程序将视频运行时间序列保存到
2  # video_times.txt 文件中。
3
4  def main():
5      # 获取项目中的短片数量
6      num_videos = int(input('项目中有多少个短片? '))
7
8      # 打开用于容纳运行时间的文件
9      video_file = open('video_times.txt', 'w')
10
11     # 获取每个短片的运行时间，
12     # 并写入文件。
13     print('输入每个短片的运行时间。')
14     for count in range(1, num_videos + 1):
15         run_time = float(input(f'视频#{count}: '))
16         video_file.write(f'{run_time}\n')
17
18     # Close the file.
19     video_file.close()
20     print('时间已写入 video_times.txt 文件。')
21
22 # 调用 main 函数
23 if __name__ == '__main__':
24     main()
```

程序输出（用户输入的内容加粗）

项目中有多少个短片? **6**
输入每个短片的运行时间。
视频#1: **24.5**
视频#2: **12.2**

视频#3: 14.6

视频#4: 20.4

视频#5: 22.5

视频#6: 19.3

时间已写入 video_times.txt 文件。

下面是第二个程序的常规算法。

将一个累加器初始化为 0。

将一个计数器变量初始化为 0。

打开输入文件。

对于 (for) 文件中的每一行:

将该行转换为浮点数 (这是一个短片的运行时间)。

计数器变量递增 1 (以便知道已经处理了多少视频)。

将运行时间加到累加器上。

关闭文件。

显示累加器中的总运行时间。

程序 6-12 展示了第二个程序的代码。

程序 6-12 (read_running_times.py)

```
1  # 这个程序读取 video_times.txt 文件中的值,
2  # 并计算它们的总和 (总的视频时间)
3
4  def main():
5      # 打开 video_times.txt 文件进行读取
6      video_file = open('video_times.txt', 'r')
7
8      # 将一个累加器初始化为 0.0
9      total = 0.0
10
11     # 初始化一个变量来跟踪视频计数
12     count = 0
13
14     print('下面是每个短片的运行时间: ')
15
16     # 从文件中读取值, 计算它们的总和
17     for line in video_file:
18         # 将一行的内容转换为 float
19         run_time = float(line)
20
21         # count 变量递增 1
22         count += 1
23
24         # 显示这一行存储的时间
```

```
25         print(f'视频#{count}: {run_time}')
26
27         # 将时间加到 total 上
28         total += run_time
29
30     # 关闭文件
31     video_file.close()
32
33     # 显示的视频总的运行时间
34     print(f'总运行时间为{total}秒。')
35
36 # 调用 main 函数
37 if __name__ == '__main__':
38     main()
```

程序输出

下面是每个短片的运行时间：

视频#1: 24.5

视频#2: 12.2

视频#3: 14.6

视频#4: 20.4

视频#5: 22.5

视频#6: 19.3

总运行时间为 113.5 秒。

检查点

6.12 写一个简短的程序，使用 `for` 循环将数字 1~10 写入一个文件。

6.13 当 `readline` 方法返回空字符串时意味着什么？

6.14 假设存在 `data.txt` 文件，并且包含几行文本。使用 `while` 循环写一个简短的程序，显示文件中的每一行。

6.15 修改为检查点 6.14 写的程序，用 `for` 循环代替 `while` 循环。

6.3 使用 `with` 语句打开文件

概念：程序必须关闭所有已打开的文件。如果使用 `with` 语句打开文件，那么当程序使用完文件后，文件会自动关闭。

6.3.1 资源

资源是程序使用的外部对象或数据源。本章一直在使用文件，它是资源的一种。除文件外，还有其他许多类型的资源，例如数据库和网络连接等。

当程序使用资源时，通常会经历以下过程：

1. 打开资源
2. 使用资源
3. 关闭资源

当程序不再使用资源时，关闭资源非常重要。例如，程序在打开文件后，必须确保在程序结束前关闭文件。如果程序没有关闭它使用的资源，我们说程序存在**资源泄漏**的情况，这可能导致各种错误。

6.3.2 with 语句

幸好，Python 提供了一个 `with` 语句，可以用它打开像文件这样的资源，并在程序使用完后自动关闭它们。下面是用于打开文件的 `with` 语句的常规格式。

```
with open(filename, mode) as file_variable:
    语句
    语句
    ...
```

其中，

- `filename` 是代表文件名的字符串。
- `mode` 是代表文件打开模式的字符串，例如 `'r'`、`'w'` 或 `'a'`。
- `file_variable` 是引用了文件对象的变量名。

接着是一个语句块，我们将其称为 `with suite`。这个 `suite` 是由一条或多个与文件相关的语句构成的语句块。当 `with suite` 中的语句执行完毕后，文件将自动关闭。因此，不需要显式调用 `close` 方法来关闭文件。

程序 6-13 展示了一个例子。该程序在第 3 行使用 `with` 语句打开一个名为 `sequence.txt` 的文件。在 `with` 语句中，第 4 行~第 5 行的循环将数字 `0~9` 写入文件。一旦循环结束，`with suite` 中就没有其他代码了，所以文件自动关闭。用“记事本”等编辑器打开 `sequence.txt` 文件，会看到如图 6-18 所示的内容。

程序 6-13 (`write_with.py`)

```
1 # 这个程序将 0~9 的数字写入 sequence.txt
2 def main():
3     with open('sequence.txt', 'w') as outfile:
4         for number in range(10):
5             outfile.write(f'{number}\n')
```

```
6     print('数据已写入 sequence.txt 文件。')
7
8     # 调用 main 函数
9     if __name__ == '__main__':
10        main()
```

程序输出

数据已写入 sequence.txt 文件。

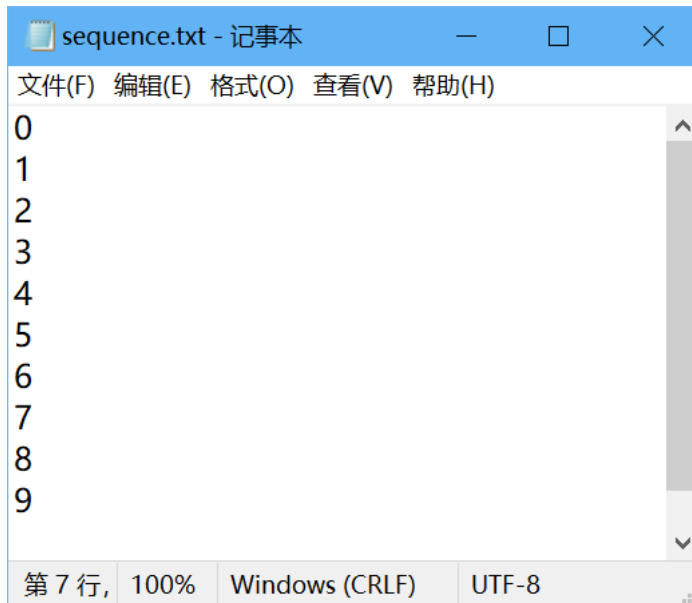


图 6-18 sequence.txt 的内容

代码清单 6-14 展示了如何使用 `with` 语句打开文件并从中读取数据。这个程序读取由代码清单 6-13 的程序创建的 `sequence.txt` 文件。

程序 6-14 (read_with.py)

```
1     # 这个程序读取 sequence.txt 文件的内容
2     def main():
3         with open('sequence.txt', 'r') as infile:
4             line = infile.read()
5             while line != '':
6                 print(f'{line}')
7                 line = infile.read()
8
9     # 高射 main 函数
```

```
10 if __name__ == '__main__':
11     main()
```

程序输出

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

6.3.3 用 with 语句打开多个文件

可以用一个 `with` 语句打开多个文件。为此，只需写多个以逗号分隔的 `open` 语句。下面是打开两个文件时的常规格式。

```
with open(filename1, mode1) as file_var1, open(filename2, mode2) as file_var2:
    语句
    语句
    ...
```

其中，

- `filename1` 是代表第一个文件名的字符串。
- `mode1` 是代表第一个文件的打开模式的字符串，例如 `'r'`、`'w'` 或 `'a'`。
- `file_var1` 是引用了第一个文件对象的变量名。
- `filename2` 是代表第二个文件名的字符串。
- `mode2` 是代表第二个文件的打开模式的字符串，例如 `'r'`、`'w'` 或 `'a'`。
- `file_var2` 是引用了第二个文件对象的变量名。

程序 6-15 展示了一个例子，它的作用是将 `sequence.txt` 文件中的内容复制到 `copy.txt` 文件中。首先，第 3 行的 `with` 语句打开这两个文件。执行该语句后，`infile` 将引用与 `sequence.txt` 关联的文件对象，`outfile` 将引用与 `copy.txt` 关联的文件对象。在 `with` 语句中（第 4 行~第 7 行），程序读取输入文件的所有行。每读取一行，就将该行写入输出文件。当这个 `with suite` 完成后，两个文件会自动关闭。

程序 6-15 (multiple_with.py)

```
# 该程序创建 sequence.txt 文件的拷贝
```

```
def main():
    with open('sequence.txt', 'r') as infile, open('copy.txt', 'w') as outfile:
        line = infile.read()
        while line != '':
            outfile.write(f'{line}')
            line = infile.read()
        print('已完成文件的复制。')

# 调用 main 函数
if __name__ == '__main__':
    main()
```

程序输出

已完成文件的复制。



注意：以后学习写更复杂的代码时，会遇到不能用 `with` 语句打开文件的情况。对于这种情况，需要确保在完成文件的处理后将文件关闭。因此，必须知道如何在使用或不使用 `with` 语句的情况下打开文件。本章已经展示了这两种方法的例子。

6.3 处理记录

概念：存储在文件中的数据通常以记录的形式组织。记录是对一个数据项（`item`）进行了描述的完整数据集，字段是记录中的单个数据。

向文件中写入的数据通常以记录和字段的形式组织。其中，**记录**是对一个数据项进行描述的完整数据集，**字段**是记录中的单个数据。例如，假设要在文件中存储员工数据。该文件将为每个员工包含一条记录。每条记录是多个字段的集合，例如姓名、员工 ID 和部门。如图 6-19 所示。

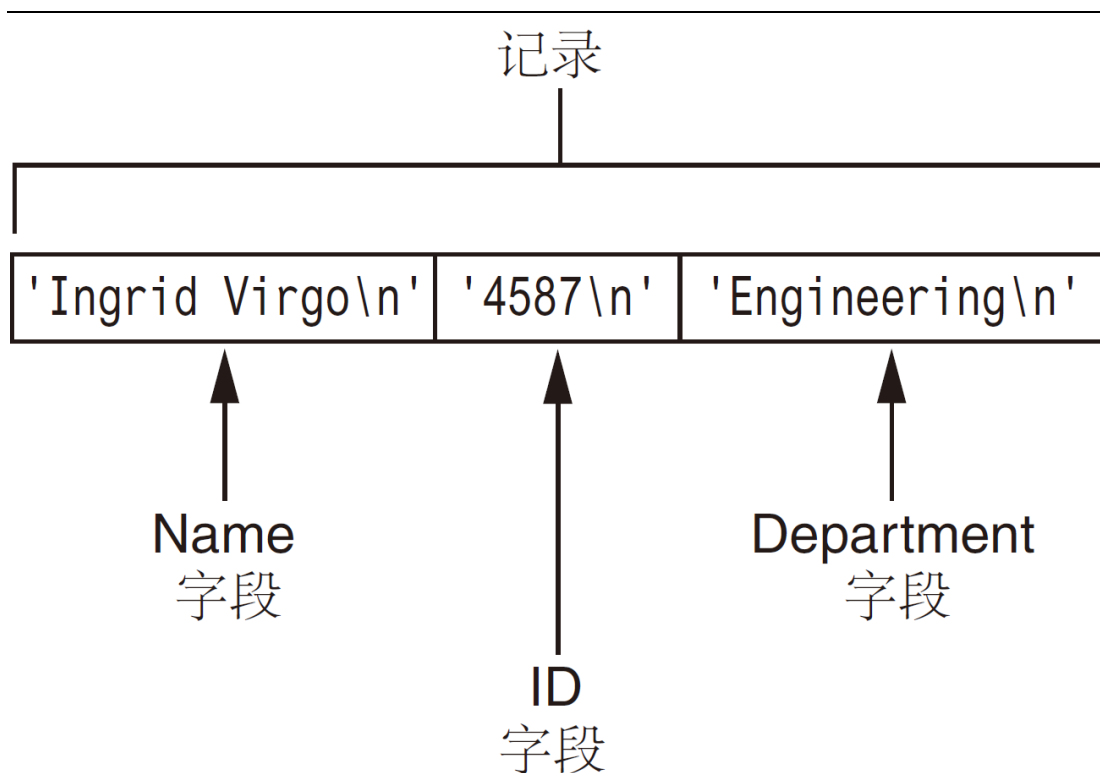


图 6-19 记录中的字段

每次向顺序访问文件写入记录时，都会一个接一个地写入构成记录的字段。例如，图 6-20 展示了包含三条员工记录的文件。每条记录都由员工姓名、ID 和部门构成。



图 6-20 文件中的记录

程序 6-16 是将员工记录写入文件的一个简单的例子。

程序 6-16 (save_emp_records.py)

```

1  # 这个程序从用户处获取员工数据，并将其作为记录
2  # 存储到 employee.txt 文件中。
3
4  def main():

```

```

5     # 获取要创建的员工记录数
6     num_emps = int(input('你想创建多少条' +
7                           '员工记录? '))
8
9     # 打开文件以便写入
10    with open('employees.txt', 'w') as emp_file:
11        # 获取每个员工的数据并写入文件
12        for count in range(1, num_emps + 1):
13            # 获取一名员工的数据
14            print(f'输入员工#{count}的数据。')
15            name = input('姓名: ')
16            id_num = input('员工 ID: ')
17            dept = input('部门: ')
18
19            # 将数据作为一条记录写入文件
20            emp_file.write(f'{name}\n')
21            emp_file.write(f'{id_num}\n')
22            emp_file.write(f'{dept}\n')
23
24            # 显示空行
25            print()
26
27        print('员工记录已写入 employees.txt 文件。')
28
29    # 调用 main 函数
30    if __name__ == '__main__':
31        main()

```

程序输出（用户输入的内容加粗）

想创建多少条员工记录? **3**

输入员工#1 的数据。

姓名: **Ingrid Virgo**

员工 ID: **4587**

部门: **Engineering**

输入员工#2 的数据。

姓名: **Julia Rich**

员工 ID: **4588**

部门: **Research**

输入员工#3 的数据。

姓名: **Greg Young**

员工 ID: **4589**

部门: **Marketing**

员工记录已写入 employees.txt 文件。

第 6 行~第 7 行的语句提示用户输入想要创建的员工记录数量。在循环内部，第 15 行~第 17 行获取员工的姓名、ID 和部门。第 20 行~第 22 行将这三项合并作为一条员工记录写入文件。针对每条员工记录都迭代要一次。

从顺序访问文件中读取一条记录时，我们一个接一个地读取每个字段的数据，直到读取完整的记录。程序 6-17 演示了如何读取 `employee.txt` 文件中的员工记录。

程序 6-17 (`read_emp_records.py`)

```
1 # 这个程序显示了 employees.txt
2 # 文件中的记录。
3
4 def main():
5     # 打开 employees.txt 文件。
6     with open('employees.txt', 'r') as emp_file:
7         # 读取文件的第一行，
8         # 它是第一条记录的姓名字段。
9         name = emp_file.readline()
10
11        # 只要读取到一个字段，循环就继续
12        while name != '':
13            # 读取员工 ID 字段
14            id_num = emp_file.readline()
15
16            # 读取部门字段
17            dept = emp_file.readline()
18
19            # 从字段中去除换行符
20            name = name.rstrip('\n')
21            id_num = id_num.rstrip('\n')
22            dept = dept.rstrip('\n')
23
24            # 显示记录
25            print(f'姓名: {name}')
26            print(f'ID: {id_num}')
27            print(f'部门: {dept}')
28            print()
29
30            # 读取下一条记录的姓名字段
31            name = emp_file.readline()
32
33 # 调用 main 函数
34 if __name__ == '__main__':
35     main()
```

程序输出

姓名: Ingrid Virgo
ID: 4587
部门: Engineering

姓名: Julia Rich
ID: 4588
部门: Research

姓名: Greg Young
ID: 4589
部门: Marketing

程序在第 6 行打开文件，然后在第 10 行读取第一条记录的第一个字段。这将是第一个员工的姓名。第 12 行的 `while` 循环测试该值是不是空字符串。如果不是，那么循环迭代。在循环内部，程序读取记录的第二个和第三个字段（员工 ID 和部门）并显示。然后，第 31 行读取下一条记录的第一个字段（下一个员工的姓名）。循环重新开始，这个过程会一直持续到没有记录可读取为止。

在文件中存储记录的程序通常需要比简单的记录读写更多的功能。以下“聚光灯”小节将研究向文件中添加记录、搜索文件中的特定记录、修改记录和删除记录的算法。

聚光灯：添加和显示记录

Midnight Coffee Roasters 公司从世界各地进口咖啡生豆并烘焙成各种精品咖啡。公司负责人朱莉要求你写一系列用于管理库存的程序。在与她交谈后，你确定需要一个文件来保存库存记录。每条记录用两个字段来保存以下数据。

- 描述：包含咖啡名称的字符串
- 库存量：以浮点数表示的库存量；单位：磅。

第一项工作是写一个程序向文件中添加记录。程序 6-18 展示了代码。注意，输出文件是以追加模式打开的。每次执行程序，新的记录都会被添加到文件的现有内容之后。

程序 6-18 (add_coffee_record.py)

```
1 # 这个程序将咖啡库存记录添加
2 # 到 coffee.txt 文件中。
3
4 def main():
5     # 创建循环控制变量
6     another = 'y'
7
8     # 以追加模式打开 coffee.txt 文件
9     with open('coffee.txt', 'a') as coffee_file:
```

```

10     # 向文件添加记录
11     while another == 'y' or another == 'Y':
12         # 获取咖啡记录数据
13         print('输入以下咖啡数据。')
14         descr = input('描述: ')
15         qty = int(input('库存量(以磅为单位): '))
16
17         # 将数据追加到文件中
18         coffee_file.write(f'{descr}\n')
19         coffee_file.write(f'{qty}\n')
20
21         # 判断用户是否要向文件
22         # 添加另一条记录
23         print('要添加另一条记录吗? ')
24         another = input('Y = 是, 其他任何输入 = 否: ')
25
26     print('数据已追加到 coffee.txt 文件。')
27
28 # 调用 main 函数
29 if __name__ == '__main__':
30     main()

```

程序输出 (用户输入的内容加粗)

```

输入以下咖啡数据。
描述: Brazilian Dark Roast 
库存量(以磅为单位): 18 
要添加另一条记录吗?
Y = 是, 其他任何输入 = 否: y 
输入以下咖啡数据。
描述: Sumatra Medium Roast 
库存量(以磅为单位): 25 
要添加另一条记录吗?
Y = 是, 其他任何输入 = 否: n 
数据已追加到 coffee.txt 文件。

```

下一项工作是写一个程序来显示库存文件中的所有记录。程序 6-19 展示了代码。

程序 6-19 (show_coffee_records.py)

```

1 # 这个程序将显示 coffee.txt 文件中的记录。
2
3 def main():
4     # 打开 coffee.txt 文件
5     with open('coffee.txt', 'r') as coffee_file:
6         # 读取第一条记录的描述字段

```

```

7         descr = coffee_file.readline()
8
9         # 读取文件其余内容
10        while descr != '':
11            # 读取库存量字段
12            qty = float(coffee_file.readline())
13
14            # 从描述中去除\n 字符
15            descr = descr.rstrip('\n')
16
17            # 显示记录
18            print(f'Description: {descr}')
19            print(f'Quantity: {qty}')
20
21            # 读取下一个描述
22            descr = coffee_file.readline()
23
24    # 调用 main 函数
25    if __name__ == '__main__':
26        main()

```

程序输出

```

描述: Brazilian Dark Roast
库存量: 18.0
描述: Sumatra Medium Roast
库存量: 25.0

```

聚光灯：搜索记录



朱莉一直在使用你之前为她编写的两个程序，并且已经在 `coffee.txt` 文件中存储了不少记录，她希望你再写一个程序来搜索记录。她希望能输入一个描述，然后查看所有符合描述的记录。程序 6-20 展示了该程序的代码。

程序 6-20 (search_coffee_records.py)

```

1    # 这个程序允许用户在 coffee.txt 文件中
2    # 搜索与描述匹配的记录。
3
4    def main():
5        # 创建一个 bool 变量作为标志
6        found = False
7
8        # 获取要搜索的值
9        search = input('输入要查找的咖啡描述: ')
10

```

```

11 # 打开 coffee.txt 文件
12 with open('coffee.txt', 'r') as coffee_file:
13     # 读取第一条记录的描述字段
14     descr = coffee_file.readline()
15
16     # 读取文件其余内容
17     while descr != '':
18         # 读取库存量字段
19         qty = float(coffee_file.readline())
20
21         # 从描述中去除\n 字符
22         descr = descr.rstrip('\n')
23
24         # 判断该记录是否与
25         # 搜索值匹配
26         if descr == search:
27             # 显示记录
28             print(f'描述: {descr}')
29             print(f'库存量: {qty}')
30             print()
31             # 将 found 标志设为 True
32             found = True
33
34         # 读取下一个描述
35         descr = coffee_file.readline()
36
37     # 如果在文件中没有发现搜索值,
38     # 就显示一条消息。
39     if not found:
40         print('在文件中没有找到符合描述的商品。')
41
42 # 调用 main 函数
43 if __name__ == '__main__':
44     main()

```

程序输出（用户输入的内容加粗）

输入要查找的咖啡描述: **Sumatra Medium Roast**

描述: Sumatra Medium Roast

库存量: 25.0

程序输出（用户输入的内容加粗）

输入要查找的咖啡描述: **Mexican Altura**

在文件中没有找到符合描述的商品。

聚光灯：修改记录

朱莉对你写的程序非常满意。下个任务是写程序来修改现有记录中的“库存量”字段，以

便在咖啡售出或者为现有类型的咖啡补货时更新库存。

要修改顺序文件中的记录，必须创建一个临时文件。将原始文件中的所有记录都复制到临时文件中，但在找到要修改的记录时，不是将其旧的内容写入临时文件。相反，将新的修改值写入临时文件。然后，继续将原始文件中的剩余记录复制到临时文件。

最后一步是用临时文件取代原始文件。具体做法是删除原始文件，并将临时文件重命名为原始文件的名称。以下是该程序的常规算法。

```
打开原始文件作为输入，创建临时文件作为输出。
获取要修改的记录的描述字段和库存量的新值。
从原始文件读取第一个描述字段。
当 (while) 描述字段不为空时：
    读取库存量字段。
    如果 (if) 该记录的描述字段与输入的描述相符：
        将新数据写入临时文件。
    否则 (else):
        将现有记录写入临时文件。
    读取下一个描述字段。
关闭原始文件和临时文件。
删除原始文件。
将临时文件重命名为原始文件的名称。
```

注意，算法最后要求删除原始文件，并重命名临时文件。Python 标准库的 `os` 模块提供了一个名为 `remove` 的函数来删除磁盘上的文件。只需将文件名作为实参传递给函数。下例展示了如何删除 `coffee.txt` 文件。

```
remove('coffee.txt')
```

`os` 模块还提供了一个名为 `rename` 的函数来重命名文件。下例将 `temp.txt` 文件重命名为 `coffee.txt`。

```
rename('temp.txt', 'coffee.txt')
```

程序 6-21 展示了完整代码。

程序 6-21 (modify_coffee_records.py)

```
1 # 这个程序允许用户修改 coffee.txt 文件中
2 # 一条记录的“库存量”字段。
3
4 import os # remove 和 rename 函数所在的模块
5
6 def main():
7     # 创建一个 bool 变量作为标志
```

```
8     found = False
9
10    # 获取要搜索的值和新的库存量
11    search = input('输入要查找的咖啡描述: ')
12    new_qty = int(input('输入新的库存量: '))
13
14    # 打开原始 coffee.txt 文件和一个临时文件
15    with open('coffee.txt', 'r') as coffee_file, open('temp.txt', 'w') as temp_file:
16        # 读取第一条记录的描述字段
17        descr = coffee_file.readline()
18
19        # 读取文件其余内容
20        while descr != '':
21            # 读取库存量字段
22            qty = float(coffee_file.readline())
23
24            # 从描述中去除\n字符
25            descr = descr.rstrip('\n')
26
27            # 要么将此记录原样写入临时文件,
28            # 要么创建新记录 (如果此记录是
29            # 要修改的记录)。
30            if descr == search:
31                # 将修改后的记录写入临时文件
32                temp_file.write(f'{descr}\n')
33                temp_file.write(f'{new_qty}\n')
34
35                # 将 found 标志设为 True
36                found = True
37            else:
38                # 将原始记录写入临时文件
39                temp_file.write(f'{descr}\n')
40                temp_file.write(f'{qty}\n')
41
42            # 读取下一个描述
43            descr = coffee_file.readline()
44
45    # 删除原始 coffee.txt 文件
46    os.remove('coffee.txt')
47
48    # 重命名临时文件
49    os.rename('temp.txt', 'coffee.txt')
50
51    # 如果在文件中没有发现搜索值,
52    # 就显示一条消息。
53    if found:
54        print('文件已更新。')
55    else:
56        print('在文件中没有找到符合描述的商品。')
57
```

```
58 # 调用 main 函数
59 if __name__ == '__main__':
60     main()
```

程序输出（用户输入的内容加粗）

输入要查找的咖啡描述: **Brazilian Dark Roast**

输入新的库存量: **10**

文件已更新。



注意：在处理顺序访问文件时，每次修改文件中的一个数据项，都需要复制整个文件。可以想象，这种方法的效率是很低的，尤其是当文件较大时。处理大量数据更合适的方法是使用数据库。我们将在第 14 章介绍数据库。

聚光灯：删除记录



最后一项任务是写一个程序使朱莉能用它来删除 `coffee.txt` 文件中的记录。与修改记录的过程一样，从顺序访问文件中删除记录需要创建一个临时文件。将除了要删除的记录之外的其他所有记录都从原始文件复制到临时文件。最后，用临时文件取代原始文件，即删除原始文件并将临时文件重命名为原始文件的名称。以下是该程序的常规算法。

打开原始文件作为输入，创建临时文件作为输出。
获取要删除的记录的描述。
从原始文件读取第一条记录的描述字段。
当 (`while`) 描述字段不为空时：
 读取库存量字段。
 如果 (`if`) 该记录的描述字段与输入的描述**不相符**：
 将记录写入临时文件。
 读取下一个描述字段。
关闭原始文件和临时文件。
删除原始文件。
将临时文件重命名为原始文件的名称。

程序 6-22 展示了完整代码。

程序 6-22 (`delete_coffee_record.py`)

```
1 # 这个程序允许用户删除 coffee.txt
2 # 文件中的一条记录。
3
4 import os # remove 和 rename 函数所在的模块
```

```
5
6 def main():
7     # 创建一个 bool 变量作为标志
8     found = False
9
10    # 获取要删除的咖啡商品记录
11    search = input('要删除哪种咖啡? ')
12
13    # 打开原始 coffee.txt 文件和一个临时文件
14    with open('coffee.txt', 'r') as coffee_file, open('temp.txt', 'w') as temp_file:
15        # 读取第一条记录的描述字段
16        descr = coffee_file.readline()
17
18        # 读取文件其余内容
19        while descr != '':
20            # 读取库存量字段
21            qty = float(coffee_file.readline())
22
23            # 从描述中去除\n 字符
24            descr = descr.rstrip('\n')
25
26            # 如果这不是要删除的记录,
27            # 那么将其写入临时文件。
28            if descr != search:
29                # 将记录写入临时文件
30                temp_file.write(f'{descr}\n')
31                temp_file.write(f'{qty}\n')
32            else:
33                # 将 found 标志设为 True
34                found = True
35
36            # 读取下一个描述
37            descr = coffee_file.readline()
38
39    # 删除原始 coffee.txt 文件
40    os.remove('coffee.txt')
41
42    # 重命名临时文件
43    os.rename('temp.txt', 'coffee.txt')
44
45    # 如果在文件中没有发现搜索值,
46    # 就显示一条消息。
47    if found:
48        print('文件已更新。')
49    else:
50        print('在文件中没有找到符合描述的商品。')
51
52    # 调用 main 函数
53    if __name__ == '__main__':
54        main()
```

程序输出（用户输入的内容加粗）

要删除哪种咖啡？ **Brazilian Dark Roast**
文件已更新。



注意：在处理顺序访问文件时，每次删除文件中的一个数据项，都必须复制整个文件。如前所述，这种方法效率很低，尤其是在文件较大的时候。还有其他更先进的技术。特别是在处理直接访问文件的时候，效率会高得多。本书不涉及这些高级技术，但可能会在以后的课程中学习它们。

检查点

6.16 什么是记录？什么是字段？

6.17 为了修改顺序访问文件中的记录，程序应该如何使用临时文件？

6.18 为了删除顺序访问文件中的记录，程序应该如何使用临时文件？

6.4 异常

概念：异常是程序运行时发生的错误，它导致程序突然停止。可以使用 `try/except` 语句来得体地处理异常。

异常是程序运行时发生的错误。大多数情况下，异常会导致程序突然停止。程序 6-23 展示了一个例子。该程序从用户那里获取两个数，然后用第一个数除以第二个数。然而，在程序的示例运行中，由于用户输入 0 作为第二个数，所以发生了异常。（除以 0 会导致异常，因为在数学上这是不可能的）。

程序 6-23 (division.py)

```
1 # 这个程序用一个数除以另一个数
2
3 def main():
4     # 获取两个数字
5     num1 = int(input('输入第一个数: '))
6     num2 = int(input('输入第二个数: '))
7
8     # 计算 num1 除以 num2 并显示结果
9     result = num1 / num2
10    print(f'{num1}除以{num2}等于{result}。')
```

```
11
12 # 调用 main 函数
13 if __name__ == '__main__':
14     main()
```

程序输出（用户输入的内容加粗）

```
输入第一个数: 10 
输入第二个数: 0 
Traceback (most recent call last):
  File "C:\Python\中文代码\Chapter 06\division.py", line 14, in <module>
    main()
  File "C:\Python\中文代码\Chapter 06\division.py", line 9, in main
    result = num1 / num2
ZeroDivisionError: division by zero
```

在示例运行中显示的跟踪（Traceback）消息指出了导致异常的行号。错误消息最后一行显示了所引发的异常的名称（ZeroDivisionError），并简要描述了引发异常的错误（除以零）。

写程序时细心一点，就能防止许多异常的发生。例如，程序 6-24 展示了如何通过一个简单的 if 语句来防止“除以 0”异常。程序不允许发生这种异常，它会测试 num2 的值，如果值为 0，那么会显示错误消息，不会执行除法运算

程序 6-24 (division2.py)

```
1 # 这个程序用一个数除以另一个数
2
3 def main():
4     # 获取两个数字
5     num1 = int(input('输入第一个数: '))
6     num2 = int(input('输入第二个数: '))
7
8     # 如果 num2 不为零，就计算 num1 除以 num2，
9     # 并显示结果。
10    if num2 != 0:
11        result = num1 / num2
12        print(f'{num1}除以{num2}等于{result}。')
13    else:
14        print('不能被零除。')
15
16 # 调用 main 函数
17 if __name__ == '__main__':
18     main()
```

程序输出（用户输入的内容加粗）

输入第一个数: 10
输入第二个数: 0
不能被零除。

但是，无论如何小心地写程序，有些异常都是无法避免的。例如，程序 6-25 计算工资总额。它提示用户输入工作时数和时薪。将这两个数字相乘，得到用户的总工资，并在屏幕上显示。

程序 6-25 (gross_pay1.py)

```
1  # 这个程序计算总工资
2
3  def main():
4      # 获取工时
5      hours = int(input('输入工时: '))
6
7      # 获取时薪
8      pay_rate = float(input('输入时薪: '))
9
10     # 计算总工资
11     gross_pay = hours * pay_rate
12
13     # 显示总工资
14     print(f'总工资: ${gross_pay:,.2f}')
15
16 # 调用 main 函数
17 if __name__ == '__main__':
18     main()
```

程序输出 (用户输入的内容加粗)

```
输入工时: 四十 
Traceback (most recent call last):
  File "C:\Python\中文代码\Chapter 06\gross_pay1.py", line 18, in <module>
    main()
  File "C:\Python\中文代码\Chapter 06\gross_pay1.py", line 5, in main
    hours = int(input('输入工时: '))
ValueError: invalid literal for int() with base 10: '四十'
```

来看看这一次示例程序运行。当提示输入工作时数时，由于用户输入的是字符串'四十'而不是数字 40，所以发生了异常。由于字符串'四十'不能转换为整数，所以函数 `int()` 在第 5 行发生了异常，程序停止运行。仔细观察跟踪消息的最后一行，会发现异常名称是 `ValueError`，对它的描述是：`int()` 收到无效的十进制字面值：'四十'。

和其他大多数现代编程语言一样，Python 允许写代码来响应异常，防止程序突然崩溃。这样的代码称为**异常处理程序 (exception handler)**，^①用 `try/except` 语句编写。`try/except` 语句有几种写法，以下常规格式展示了最简单的一种。

```
try:
    语句
    语句
    ...
except 异常名称:
    语句
    语句
    ...
```

首先是关键字 `try`，然后是冒号。接着是一个语句块，我们将其称为 `try suite`。`try suite` 是一个或多个可能引发异常的语句。

`try suite` 后是一个 `except` 子句。`except` 子句以关键字 `except` 开头，后跟异常名称，最后以冒号结束。从下一行开始是一个语句块，我们将其称为**处理程序 (handler)**。

当 `try/except` 语句执行时，`try suite` 中的语句会开始执行。下面描述了接下来发生的事情：

- 如果 `try suite` 中的语句引发了由 `except` 子句中的*异常名称*指定的一个异常，那么会执行紧跟在 `except` 子句后面的处理程序。然后，程序继续执行紧跟在 `try/except` 语句后面的语句。
- 如果 `try suite` 中的语句引发了一个异常，但该异常和 `except` 子句中的*异常名称*不匹配，那么程序将停止运行并显示一条错误跟踪消息。
- 如果 `try suite` 中执行的语句没有引发异常，那么语句中的任何 `except` 子句和处理程序都会被跳过，程序将继续执行紧跟在 `try/except` 语句后面的语句。

程序 6-26 展示了如何编写 `try/except` 语句来得体地响应 `ValueError` 异常。

程序 6-26 (`gross_pay2.py`)

```
1  # 这个程序计算总工资
2
3  def main():
4      try:
```

^① 译注：本书只是按照约定俗成的译法，将 `exception handler` 翻译成“异常处理程序”，但它并不是一个真正意义上的“程序”。在 Python 中，请把它理解成“异常处理语句块”。

```
5         # 获取工时
6         hours = int(input('输入工时: '))
7
8         # 获取时薪
9         pay_rate = float(input('输入时薪: '))
10
11        # 计算总工资
12        gross_pay = hours * pay_rate
13
14        # 显示总工资
15        print(f'总工资: ${gross_pay:,.2f}')
16    except ValueError:
17        print('错误: 工时和时薪必须')
18        print('是有效的数字。')
19
20    # 调用 main 函数
21    if __name__ == '__main__':
22        main()
```

程序输出（用户输入的内容加粗）

```
输入工时: 四十 Enter
错误: 工时和时薪必须
是有效的数字。
```

来看看在示例运行中发生了什么。第 6 行的语句提示用户输入工时，用户输入字符串 '四十'。由于字符串 '四十' 不能转换为整数，所以 `int()` 函数引发了 `ValueError` 异常。在这种情况下，程序会立即跳出 `try suite`，并跳转到第 16 行的 `except ValueError` 子句，开始执行从第 17 行开始的处理程序。如图 6-21 所示。

```

# 这个程序计算总工资

def main():
    try:
        # 获取工时
        hours = int(input('输入工时: '))

        # 获取时薪
        pay_rate = float(input('输入时薪: '))

        # 计算总工资
        gross_pay = hours * pay_rate

        # 显示总工资
        print(f'总工资: ${gross_pay:,.2f}')
    except ValueError:
        print('错误: 工时和时薪必须')
        print('是有效的数字。')

# 调用main函数
if __name__ == '__main__':
    main()

```

如果该语句引发一个 ValueError 异常...

那么程序会跳转到 except ValueError 子句，并执行它的处理程序。

图 6-21 异常处理

程序 6-27 展示了另一个例子。这个程序没有使用异常处理。它从用户处获取文件名，然后显示指定文件中的内容。只要用户输入的是现有文件的名称，程序就能正常工作。但是，如果用户指定的文件不存在，就会发生异常，如示例运行所示。

程序 6-27 (display_file.py)

```

1  # 这个程序显示一个
2  # 文件的内容。
3
4  def main():
5      # 获取文件名
6      filename = input('输入文件名: ')
7
8      # 打开文件
9      with open(filename, 'r') as infile:
10         # 读取文件内容
11         contents = infile.read()
12
13         # 显示文件内容
14         print(contents)
15

```

```
16 # 调用 main 函数
17 if __name__ == '__main__':
18     main()
```

程序输出（用户输入的内容加粗）

```
输入文件名: badfile.txt 
Traceback (most recent call last):
  File "C:\Python\中文代码\Chapter 06\display_file.py", line 18, in <module>
    main()
  File "C:\Python\中文代码\Chapter 06\display_file.py", line 9, in main
    with open(filename, 'r') as infile:
FileNotFoundError: [Errno 2] No such file or directory: 'badfile.txt'
```

第 9 行的语句在调用 `open` 函数时发生了异常。在错误跟踪消息中，注意异常的名称是 `FileNotFoundError`。当程序试图打开一个不存在的文件时，就会引发该异常。可以在跟踪消息中看到错误的原因是：不存在该文件或目录：`'badfile.txt'`。

程序 6-28 展示了如何通过一个 `try/except` 语句来修改程序 6-27，以得体地响应 `FileNotFoundError` 异常。在示例运行中，同样假设文件 `badfile.txt` 不存在。

程序 6-28 (display_file2.py)

```
1 # 这个程序显示一个
2 # 文件的内容。
3
4 def main():
5     # 获取文件名
6     filename = input('输入文件名: ')
7
8     try:
9         # 打开文件
10        with open(filename, 'r') as infile:
11            # 读取文件内容
12            contents = infile.read()
13
14            # 显示文件内容
15            print(contents)
16    except FileNotFoundError:
17        print(f'文件{filename}不存在。')
18
19 # 调用 main 函数
20 if __name__ == '__main__':
21     main()
```

程序输出（用户输入的内容加粗）

```
输入文件名: badfile.txt Enter  
文件 bad_file.txt 不存在。
```

来看看在示例运行中发生了什么。执行第 6 行时，用户输入了 `badfile.txt`，它被赋给 `filename` 变量。在 `try suite` 中，第 10 行尝试打开文件 `badfile.txt`。由于该文件不存在，所以语句引发了 `FileNotFoundError` 异常。发生这种情况时，程序会退出 `try suite`，跳过第 11 行~第 15 行。由于第 16 行的 `except` 子句指定了 `FileNotFoundError` 异常，所以程序跳转到第 17 行，打印文件不存在的消息。

6.4.1 处理多个异常

`try suite` 中的代码经常会抛出不止一种类型的异常。在这种情况下，可以为每种异常编写一个 `except` 子句。例如，程序 6-29 读取一个名为 `sales_data.txt` 文件的内容。文件中的每一行都包含一个月的销售额，文件有多行。下面是文件的内容：

```
24987.62  
26978.97  
32589.45  
31978.47  
22781.76  
29871.44
```

程序 6-29 从文件中读取所有数字，并将它们加到一个累加器变量上。

程序 6-29 (`sales_report1.py`)

```
1  # 这个程序显示 sales_data.txt 文件  
2  # 中的所有金额之和。  
3  
4  def main():  
5      # 初始化累加器  
6      total = 0.0  
7  
8      try:  
9          # 打开 sales_data.txt 文件  
10         with open('sales_data.txt', 'r') as infile:  
11             # 从文件中读取数值并累加  
12             for line in infile:  
13                 amount = float(line)  
14                 total += amount  
15  
16         # 打印总计
```

```

17         print(f'{total:,.2f}')
18
19     except FileNotFoundError:
20         print('尝试读取文件时发生一个错误。')
21
22     except ValueError:
23         print('文件中存在非数值数据。')
24
25     except:
26         print('发生一个错误。')
27
28 # 调用 main 函数
29 if __name__ == '__main__':
30     main()

```

try suite 包含了可能引发不同类型异常的代码。

- 如果不存在 sales_data.txt 文件，第 10 行的语句可能引发 FileNotFoundError 异常。
- 如果 line 变量引用的字符串不能转换为浮点数（例如字母字符串），那么第 13 行中的 float 函数可能引发 ValueError 异常。

注意，try/except 语句中有三个 except 子句。

- 第 19 行的 except 子句指定了 FileNotFoundError 异常。如果发生 FileNotFoundError 异常时，那么会执行第 20 行的处理程序。
- 第 22 行的 except 子句指定了 ValueError 异常。如果发生 ValueError 异常，那么会执行第 23 行的处理程序。
- 第 25 行的 except 子句没有列出特定的异常。如果发生其他 except 子句没有特地处理的异常，那么会执行第 26 行的常规异常处理程序。

在 try suite 中发生异常后，Python 解释器会从上向下检查 try/except 语句中的每个 except 子句。如果发现一个 except 子句所指定的异常类型与发生的异常类型相匹配，它就分支到这个 except 子句。如果没有任何 except 子句指定了与发生的异常类型匹配的类型，那么解释器将分支到第 25 行的 except 子句。

6.4.2 用一个 except 子句捕获所有异常

前面的例子展示了如何在 try/except 语句中单独处理多种类型的异常。有的时候，你可能想写一个 try/except 语句来简单地捕获在 try suite 中发生的所有异常，并且不管异常的类型如何，都以相同的方式响应。在 try/except 语句中，可以写一个不指定异常类型的 except 子句来达到这个目的。程序 6-30 展示了一个例子。

程序 6-30 (sales_report2.py)

```
1 # 这个程序显示 sales_data.txt 文件
2 # 中的所有金额之和。
3
4 def main():
5     # 初始化累加器
6     total = 0.0
7
8     try:
9         # 打开 sales_data.txt 文件
10        with open('sales_data.txt', 'r') as infile:
11            # 从文件中读取数值并累加
12            for line in infile:
13                amount = float(line)
14                total += amount
15
16            # 打印总计
17            print(f'{total:,.2f}')
18        except:
19            print('发生一个错误。')
20
21 # 调用 main 函数
22 if __name__ == '__main__':
23     main()
```

注意，这个程序中的 `try/except` 语句只有一个 `except` 子句（第 18 行），而且该 `except` 子句没有指定任何具体的异常类型。因此，在 `try suite`（第 9 行~第 14 行）中发生的任何异常都会导致程序分支到第 18 行，并执行第 19 行的语句。

6.4.3 显示异常的默认错误消息

抛出异常时，内存中会创建一个**异常对象**。异常对象通常包含该异常的默认错误消息。事实上，当异常未得到处理时，跟踪消息最后显示的就是它。在写 `except` 子句时，可以选择将异常对象赋给一个变量，如下所示。

```
except ValueError as err:
```

这个 `except` 子句捕获 `ValueError` 异常。`except` 子句后面的表达式指出要将异常对象赋给 `err` 变量。`err` 这个名称本身并没有什么特别之处。这只是我们为本例选择的名称，完全可以使用自己希望的任何名称。然后，在异常处理程序中，可以将 `err` 变量传递给 `print` 函数，以显示 Python 为这种类型的错误准备的默认错误消息。程序 6-31 展示了一个例子。

程序 6-31 (gross_pay3.py)

```
1 # 这个程序计算总工资
2
3 def main():
```

```

4     try:
5         # 获取工时
6         hours = int(input('输入工时: '))
7
8         # 获取时薪
9         pay_rate = float(input('输入时薪: '))
10
11        # 计算总工资
12        gross_pay = hours * pay_rate
13
14        # 显示总工资
15        print(f'总工资: ${gross_pay:,.2f}')
16    except ValueError as err:
17        print(err)
18
19    # 调用 main 函数
20    if __name__ == '__main__':
21        main()

```

程序输出（用户输入的内容加粗）

```

输入工时: 四十 Enter
invalid literal for int() with base 10: '四十'

```

在 try suite（第 5 行~第 15 行）中发生 ValueError 异常时，程序会分支到第 16 行的 except 子句。第 16 行的表达式 ValueError as err 将异常对象赋给一个名为 err 的变量。第 17 行的语句将 err 变量传递给 print 函数，print 函数将显示该异常的默认错误消息。

为了用一个 except 子句来捕获 try suite 中发生的所有异常，可以将 Exception 指定为要捕获的异常类型。程序 6-32 展示了一个例子。

程序 6-32 (sales_report3.py)

```

1    # 这个程序显示 sales_data.txt 文件
2    # 中的所有金额之和。
3
4    def main():
5        # 初始化累加器
6        total = 0.0
7
8        try:
9            # 打开 sales_data.txt 文件
10           with open('sales_data.txt', 'r') as infile:
11               # 从文件中读取数值并累加
12               for line in infile:
13                   amount = float(line)
14                   total += amount

```

```
15
16     # 打印总计
17     print(f'{total:,.2f}')
18 except Exception as err:
19     print(err)
20
21 # 调用 main 函数
22 if __name__ == '__main__':
23     main()
```

6.4.4 else 子句

try/except 语句支持一个可选的 else 子句，它出现在所有 except 子句之后。下面是带有 else 子句的 try/except 语句的常规格式。

```
try:
    语句
    语句
    ...
except 异常名称:
    语句
    语句
    ...
else:
    语句
    语句
    ...
```

else 子句之后的那个语句块称为 else suite。只有在没有发生异常的情况下，else suite 中的语句才会在 try suite 中的语句之后执行。如果发生异常，那么会跳过 else suite。程序 6-33 展示了一个例子。

程序 6-33 (sales_report4.py)

```
1 # 这个程序显示 sales_data.txt 文件
2 # 中的所有金额之和。
3
4 def main():
5     # 初始化累加器
6     total = 0.0
7
8     try:
9         # 打开 sales_data.txt 文件
10        with open('sales_data.txt', 'r') as infile:
11            # 从文件中读取数值并累加
12            for line in infile:
13                amount = float(line)
```

```

14             total += amount
15
16     except Exception as err:
17         print(err)
18     else:
19         # 打印总计
20         print(f'{total:,.2f}')
21
22 # 调用 main 函数
23 if __name__ == '__main__':
24     main()

```

在程序 6-33 中，第 20 行的语句只有在 `try suite` 中的语句（第 9 行~第 14 行）没有引发异常的前提下才会执行。

6.4.5 finally 子句

`try/except` 语句支持一个可选的 `finally` 子句，它必须出现在所有 `except` 子句之后。下面是带有 `finally` 子句的 `try/except` 语句的常规格式。

```

try:
    语句
    语句
    ...
except 异常名称:
    语句
    语句
    ...
finally:
    语句
    语句
    ...

```

`finally` 子句之后的语句块称为 `finally suite`。`finally suite` 中的语句总是在 `try suite` 和任何 `except` 处理程序执行完毕后执行。无论是否发生异常，最后都会执行 `finally suite` 中的语句。`finally suite` 的目的是执行一些清理操作，例如关闭文件或其他资源。`finally suite` 中的代码始终都会执行，即使 `try suite` 引发了异常。

6.4.6 如果异常未被处理怎么办？

除非异常得到处理，否则它会导致程序停止。有两种可能会导致异常未被处理。第一种可能是 `try/except` 语句没有包含指定了正确异常类型的 `except` 子句。第二种可能是异常在 `try suite` 的外部引发。无论哪种情况，异常都会导致程序停止。

本节的示例程序演示了 `ZeroDivisionError`，`FileNotFoundError` 和 `ValueError` 等异常。Python 程序中还可能发生其他许多类型的异常。在设计 `try/except` 语句时，为了了解需要处理的异常，一个办法是查阅 Python 文档。它详细描述了每种可能的异常，以及可能导致

它们发生的原因。

另一个了解程序中可能发生的异常的办法是多做测试。可以运行程序，并故意执行一些可能导致错误的操作。观察显示的跟踪消息，可以看到所引发的异常的名称。然后，可以写相应的 `except` 子句来处理这些异常。

检查点

- 6.19 简单地说明什么是异常。
- 6.20 如果发生异常，而程序没有用 `try/except` 语句来处理它，那么会发生什么？
- 6.21 当程序试图打开一个不存在的文件时，会引发什么类型的异常？
- 6.22 当程序使用 `float` 函数将非数字字符串转换为数字时，会引发哪种类型的异常？

复习题

选择题

1. 向其写入数据的文件称为_____。
a. 输入文件 b. 输出文件 c. 顺序访问文件 d. 二进制文件
2. 从中读取数据的文件称为_____。
a. 输入文件 b. 输出文件 c. 顺序访问文件 d. 二进制文件
3. 在程序能够使用文件之前，文件必须先_____。
a. 格式化 b. 加密 c. 关闭 d. 打开
4. 程序用完一个文件后，应该_____。
a. 擦除文件 b. 打开文件 c. 关闭文件 d. 加密文件
5. _____的内容可以在“记事本”等编辑器中正常查看。
a. 文本文件 b. 二进制文件 c. 英文文件 d. 人类可读文件
6. _____包含未转换为文本的数据。
a. 文本文件 b. 二进制文件 c. Unicode 文件 d. 符号文件
7. 在处理_____文件时，必须从文件头到文件尾来访问其数据。

- a. 有序访问 b. 二进制访问 c. 直接存取 d. 顺序访问
8. 处理_____文件时，可以直接跳到文件中的任意数据，无需事先读取在它前面的数据。
- a. 有序访问 b. 二进制访问 c. 直接存取 d. 顺序访问
9. _____是内存中的一个小的“保留区”，许多系统在将数据实际写入文件之前都会先写入到这里。
- a. 缓冲区 b. 变量 c. 虚拟文件 d. 临时文件
10. _____标志着将从文件中读取的下一个数据项的位置。
- a. 输入位置 b. 分隔符 c. 指针 d. 读取位置
11. 以_____模式打开文件时，数据将写入文件现有内容的末尾。
- a. 输出 b. 追加 c. 备份 d. 只读
12. _____是记录中的单个数据。
- a. 字段 b. 变量 c. 分隔符 d. 子记录
13. 当一个异常发生时，我们说该异常被_____。
- a. 生成 b. 引发 c. 捕捉 d. 杀死
14. _____是一段用于得体地响应异常的代码。
- a. 异常生成器 b. 异常处理程序 c. 异常操作器 d. 异常监视器
15. 在 Python 中，我们写_____语句来响应异常。
- a. run/handle b. try/except c. try/handle d. attempt/except

判断题

1. 在处理顺序访问文件时，可以直接跳转到文件中的任何数据，而无需读取其前面的数据。
2. 使用 'w' 模式打开磁盘上的现有文件时，文件中的内容会被删除。
3. 只有输入文件才需要打开文件。输出文件在写入数据时自动打开。
4. 打开输入文件时，其读取位置初始化为文件中的第一个数据项。
5. 以追加模式打开一个现有的文件时，文件中的内容会被删除。
6. 未处理的异常会被 Python 解释器忽略，并且程序将继续执行。

-
7. `try/except` 语句中可以有多多个 `except` 子句。
 8. `try/except` 语句中的 `else suite` 只有在 `try suite` 中的语句引发异常时才会执行。
 9. `try/except` 语句中的 `finally` 子句只有在 `try suite` 中的语句没有引发异常时才会执行。

简答题

1. 描述程序使用文件时必须采取的三个步骤。
2. 程序使用完文件后为什么要关闭文件？
3. 什么是文件的读取位置？当文件首次打开进行读取时，读取位置在哪里？
4. 以追加模式打开一个现有文件，文件中现有的内容会发生什么变化？
5. 如果一个文件不存在，而程序试图以追加模式打开它，那么会发生什么？

算法工作台

1. 写程序来打开一个名为 `my_name.txt` 的输出文件，将你的姓名写入该文件，然后关闭文件。
2. 写程序来打开由问题 1 的程序创建的 `my_name.txt` 文件，从文件中读取你的姓名，在屏幕上显示，然后关闭文件。
3. 写代码来打开一个名为 `number_list.txt` 的输出文件，利用循环将数字 1~100 写入文件，然后关闭文件。
4. 写代码来打开由问题 3 的代码创建的 `number_list.txt` 文件，从文件中读取并显示所有数字，然后关闭文件。
5. 修改问题 4 编写的代码，计算并显示从文件中读取的所有数字之和。
6. 写代码来打开一个名为 `number_list.txt` 的输出文件。但是，如果该文件已经存在，那么不要清除文件内容。
7. 磁盘上存在一个名为 `students.txt` 的文件。该文件包含多条记录，每条记录包含两个字段：(1)学生姓名；(2)学生期末考试分数。写代码来删除包含学生姓名“John Perz”的记录。
8. 磁盘上存在一个名为 `students.txt` 的文件。该文件包含多条记录，每条记录包含两个字段：(1)学生姓名；(2)学生期末考试分数。写代码将 Julie Milan 的分数改为 100 分。

9. 以下代码会显示什么？


```
try:
    x = float('abc123')
    print('The conversion is complete.')
except IOError:
    print('This code caused an IOError.')
except ValueError:
    print('This code caused a ValueError.')
print('The end.')
```

10. 以下代码会显示什么？

```
try:
    x = float('abc123')
    print(x)
except IOError:
    print('This code caused an IOError.')
except ZeroDivisionError:
    print('This code caused a ZeroDivisionError.')
except:
    print('An error happened.')
print('The end.')
```

编程练习

1. 文件显示

 视频讲解：File Display

假设 numbers.txt 文件包含一系列整数。写一个程序来显示文件中的所有数字。

2. 文件头显示

写一个程序向用户询问文件名。程序应仅显示文件内容的前五行。如果少于 5 行，那么显示文件的全部内容。

3. 行号

写一个程序向用户询问文件名。程序应显示文件的内容，每行前面加上行号，后跟一个冒号。行号应从 1 开始。

4. 姓名计数器

假设 names.txt 文件包含一系列字符串形式的姓名。写程序来显示文件中存储的姓名数量。提示：打开文件并读取其中存储的每个字符串。使用变量来记录读取的个数。

5. 数字求和

假设 `names.txt` 文件包含一系列整数。写程序来读取文件中存储的所有数字并计算它们的总和。

6. 求平均数

假设 `names.txt` 文件包含一系列整数。写程序来读取文件中存储的所有数字并计算它们的平均数。

7. 随机数文件写入器

写程序将一系列随机数写入文件。每个随机数的范围为 1~500。程序应允许用户指定文件中将包含多少个随机数。

8. 随机数文件读取器

本练习假定你已经完成了编程练习 7。写另一个程序，从文件中读取随机数，显示随机数，然后显示以下数据：

- 所有数字之和
- 从文件中读取的随机数的个数

9. 异常处理

修改为练习 6 编的程序，使其能进行以下异常处理。

- 处理打开文件时可能发生的 `FileNotFoundError` 异常。
- 将从文件中读取的数据项被转换成数字时，它应该处理任何 `ValueError` 异常。

10. 高尔夫分数

Springfork 业余高尔夫俱乐部每个周末都举办比赛。俱乐部主席要求你写两个程序：

1. 一个程序从键盘获取每个玩家的姓名和高尔夫分数，将它们作为记录保存到 `golf.txt` 文件中。每条记录都有两个字段，一个代表玩家姓名，另一个代表分数。
2. 另一个程序从 `golf.txt` 文件中读取记录并显示。

11. 个人网页生成器

写程序询问用户的姓名，然后要求用户输入一个描述自己的句子。下面是程序的一个示例屏幕。

输入姓名：张三丰

描述一下你自己：我是武当派的始祖祖师，自创了太极拳和太极剑。

在用户完成输入后，程序将创建一个简单的 HTML 文件，其中包含输入的内容。下面是一个例子。

```
<html>
<head>
</head>
<body>
  <center>
    <h1>张三丰</h1>
  </center>
  <hr />
  我是武当派的始祖祖师，自创了太极拳和太极剑。
  <hr />
</body>
</html>
```

为了测试这个程序，请随意输入你虚构的任何信息。

12. 平均步数

个人健身追踪器是一种可穿戴装备，用于追踪你的锻炼活动、卡路里燃烧量、心率、睡眠模式等。大多数此类设备跟踪的一项常见锻炼活动是每天所走的步数。

在你下载的本书配套资源中，Chapter 06 文件夹下有一个名为 `steps.txt` 的文件，其中包含一个人一年中每天所走的步数。该文件共有 365 行，每行包含一天所走的步数。第一行是 1 月 1 日的步数，第二行是 1 月 2 日的步数，以此类推。写一个程序来读取该文件，然后显示每个月的平均步数。注意，数据来自非闰年，因此二月有 28 天。

附录 A 安装 Python

下载 Python

运行本书的程序需要安装 Python 3.11 或更高版本。可以从 www.python.org/downloads 下载最新版本的 Python，本附录介绍了如何为 Windows 安装 Python。也可以在 Mac、Linux 和其他一些平台上安装 Python。Python 下载页面提供了针对这些系统的 Python 下载链接：www.python.org/downloads。



提示：有两个 Python “家族” 可供下载：Python 3.x 和 Python 2.x。本书的程序要求 Python 3.x。

为 Windows 安装 Python 3.x

访问 Python 下载页面（www.python.org/downloads）来下载最新的 Python 3.x 版本。图 A-1 展示了本书写作时的下载页面内容。如图所示，Python 3.11.4 是当时的最新版本。

Python version	Maintenance status	First released	End of support	Release schedule
3.12	prerelease	2023-10-02 (planned)	2028-10	PEP 693
3.11	bugfix	2022-10-24	2027-10	PEP 664
3.10	security	2021-10-04	2026-10	PEP 619
3.9	security	2020-10-05	2025-10	PEP 596
3.8	security	2019-10-14	2024-10	PEP 569

图 A-1 下载最新版本的 Python

执行下载回来的 Python 安装程序。图 A-2 展示了 Python 3.10.7 的安装程序界面。强烈建议

勾选底部的两个选项：**Install launcher for all users**（为所有用户安装启动程序）和 **Add Python 3.x to PATH**（将 Python 3.x 添加到 PATH 环境变量）。完成后，单击 **Install Now**（立即安装）。

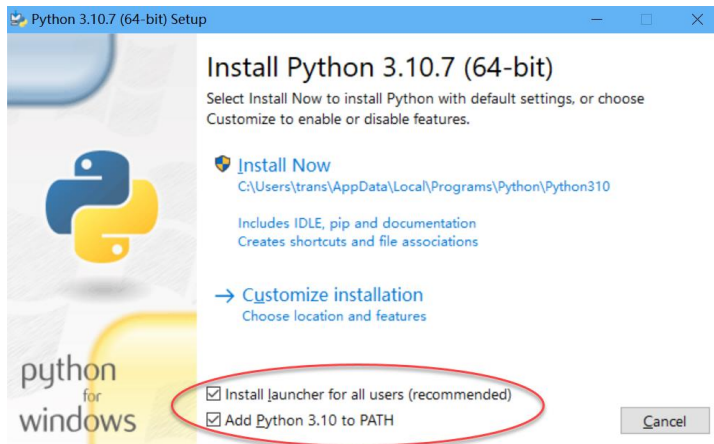


图 A.2 Python 安装程序

如果 Windows 提示“你要允许此应用对你的设备进行更改吗？”，请单击“是”以继续。安装完成后，会看到消息“Setup was successful”（已成功安装）。单击“Close”（关闭）退出安装程序。

附录 B IDLE 简介

 视频讲解：Introduction to IDLE

IDLE 是一个集成开发环境，它将多个开发工具整合到一个程序中，这些工具包括：

- 一个在交互模式下运行的 Python Shell（外壳程序）。可以在 Shell 的提示符下输入 Python 语句并立即执行。还可以运行完整的 Python 程序。
- 一个文本编辑器，支持 Python 关键字和程序其他部分的彩色编码。
- 一个“check module”（检查模块）工具，能在不运行程序的前提下检查 Python 程序的语法错误。
- 允许在一个或多个文件中查找文本的搜索工具。
- 文本格式化工具，帮助你在 Python 程序中保持缩进的一致性。
- 调试器，允许单步执行 Python 程序，并观察变量值在每个语句执行时的变化。
- 其他一些供开发者使用的高级工具。

IDLE 软件是和 Python 捆绑的。安装 Python 解释器时，IDLE 也会自动安装。本附录提供了 IDLE 的快速上手指南，并描述了创建、保存和执行 Python 程序的基本步骤。

启动 IDLE 并使用 Python Shell

在系统上安装了 Python 之后，一个 Python 程序组会出现在“开始”菜单的程序列表中。单击其中的 IDLE (Python 3.x 64-bit)来启动 IDLE，随后会出现如图 B-1 所示的 Python Shell 窗口。在这个窗口中，Python 解释器以交互模式运行，窗口顶部是一个菜单栏，可以通过它来访问 IDLE 的所有工具。

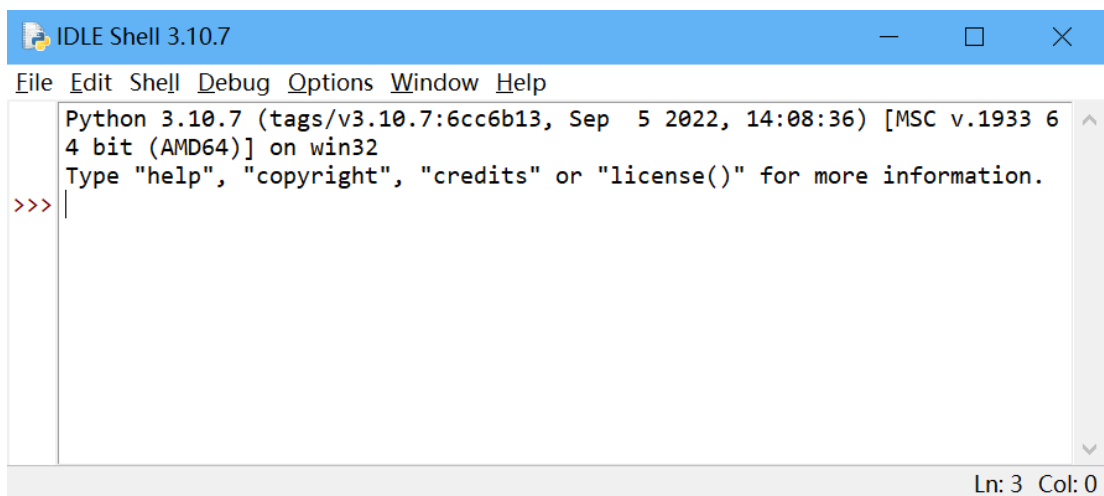
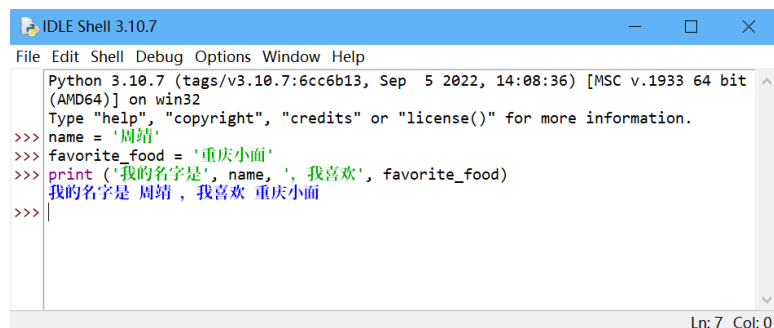


图 B-1 IDLE Shell 容器

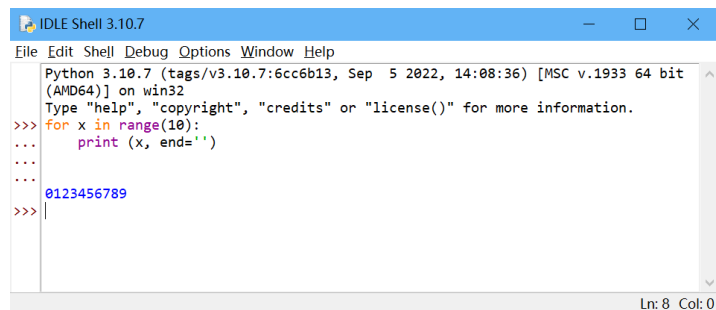
>>>提示符表示解释器正在等待你输入一个 Python 语句。在>>>提示符下输入一个语句并按下 Enter 键后，该语句会立即执行。例如，图 B-2 展示了在输入并执行了三个语句之后的 Python Shell 窗口。



```
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep 5 2022, 14:08:36) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> name = '周靖'
>>> favorite_food = '重庆小面'
>>> print('我的名字是', name, '，我喜欢', favorite_food)
我的名字是 周靖 ，我喜欢 重庆小面
>>>
```

图 B-2 用 Python 解释器执行多个语句

如果输入一个多行语句（例如 if 语句或循环）的开头，随后每一行都会自动缩进。在空行上按 Enter 键表示多行语句结束，并使解释器执行它。图 B-3 展示了输入并执行一个 for 循环后的 Python Shell 窗口。



```
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep 5 2022, 14:08:36) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> for x in range(10):
...     print(x, end='')
...
...
0123456789
>>>
```

图 B-3 用 Python 解释器执行一个多行语句

在 IDLE 编辑器中写 Python 程序

为了在 IDLE 中写新的 Python 程序，需要打开一个新的编辑窗口。如图 B-4 所示，单击菜单栏上的 File，然后单击下拉菜单中的 New File（或者按 Ctrl+N）。这将打开一个如图 B-5 所示的文本编辑窗口。

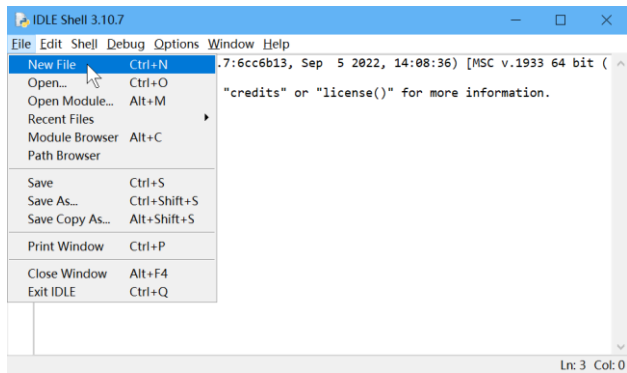


图 B-4 File 菜单

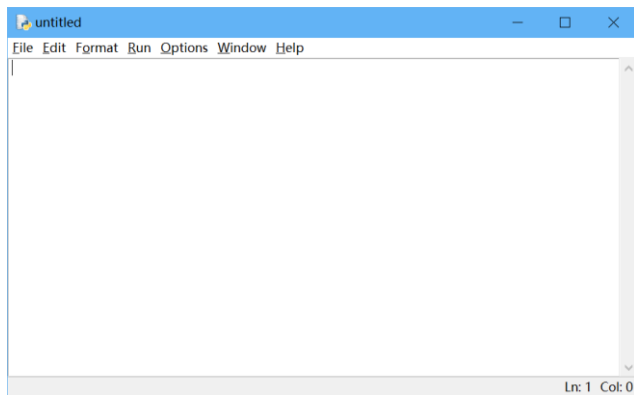


图 B-5 新建的文本编辑器窗口

要打开现有的程序，单击菜单栏上的 **File**，再单击 **Open**。找到目标文件后单击“打开”，即可在文本编辑器中打开它。

语法彩色标注

在编辑器窗口和 Python Shell 窗口中输入的代码会以彩色标注：

- Python 关键字显示为橙色。
- 注释显示为红色。
- 字符串字面值显示为绿色。
- 已定义名称（例如函数名和类名）显示为蓝色。
- 内置函数显示为紫色。



提示：可以单击菜单栏上的 Options，再单击 Configure IDLE 来更改 IDLE 的代码彩色标注。选择对话框顶部的 Highlights 标签，可以为 Python 程序的每个元素指定颜色。

自动缩进

IDLE 编辑器的一些特色可以帮助你 Python 程序中保持一致的缩进。也许这些特色中最有用的就是自动缩进。输入一个以冒号结尾的行时，例如 `if` 子句、循环的第一行或者一个函数头，然后按 Enter 键，编辑器会自动缩进接下来的输入行。例如，假设正在输入图 B-6 的代码。在标有 ① 的行末按下 Enter 键后，编辑器会自动缩进接下来的输入行。然后，在标有 ② 的行末按下 Enter 键后，编辑器会再次缩进。在缩进行的开头按 Backspace 键可以取消一级缩进。

```
File Edit Format Run Options Window Help
def main():
    # 打印一条消息5遍
    for x in range(5):
        print('Hello world!')
# 调用main函数
main()
```

图 B-6 造成自动缩进的代码行

默认情况下，IDLE 每缩进一级就缩进 4 个空格。可以单击菜单栏上的 Options，再单击 Configure IDLE 来更改空格数量。在对话框顶部单击 Windows 标签，然后利用 Indent spaces 控件指定你希望的缩进空格数。然而，4 个空格是标准的 Python 缩进宽度，建议保持默认设置不变。

保存程序

在编辑器窗口中，可从 File 菜单中选择以下选项来保存当前程序：

- Save（保存）
- Save As（另存为）
- Save Copy As（副本另存为）

Save 和 Save As 的作用和其他任何 Windows 应用程序中的操作一致。Save Copy As 的工作方式与 Save As 相似，只是编辑器窗口会保留原始程序。

运行程序

在编辑器输入一个程序后，可以按 F5 执行它，也可以像图 B-7 那样单击菜单栏上的 Run，再单击 Run Module。如果程序自上次修改后尚未存盘，会出现如图 B-8 所示的对话框。单击“确定”来保存程序。程序运行时，会在 IDLE 的 Python Shell 窗口中显示输出，如图 B-9 所示。

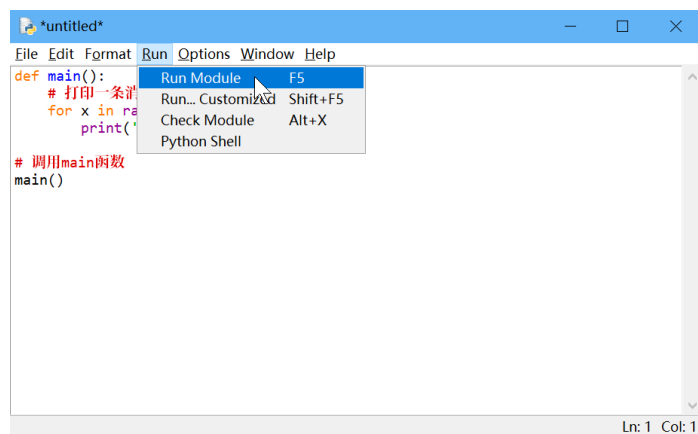


图 B-7 编辑器窗口的 Run 菜单

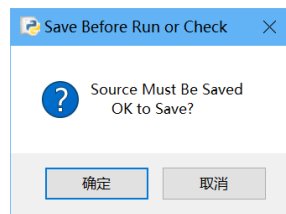


图 B-8 在这个对话框中确认保存

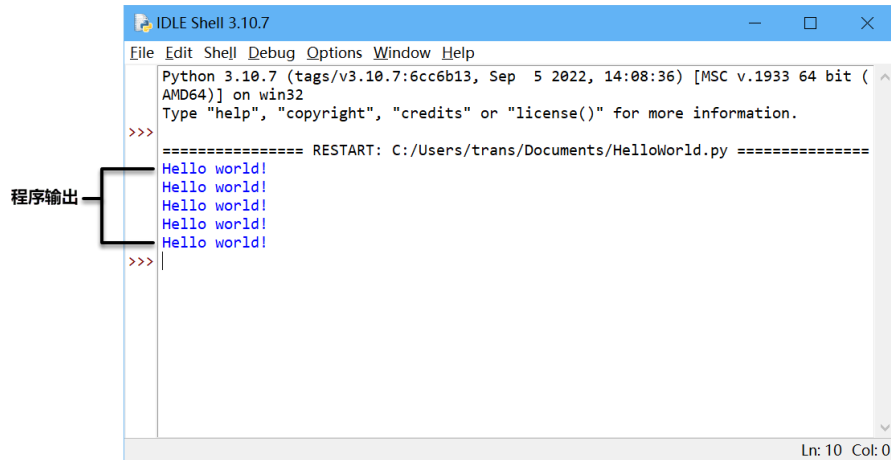


图 B-9 输出在 Python Shell 窗口中显示

如果程序含有语法错误，运行程序时会出现如图 B-10 所示的对话框。单击“确定”按钮后，编辑器会突出显示错误在代码中的位置。如果想在运行程序的前提下检查程序的语法，可以单击菜单栏上的 Run，再单击 Check Module。随后会报告发现的任何语法错误。

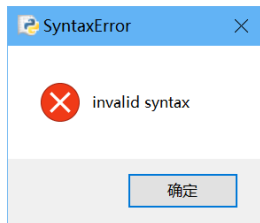


图 B-10 在对话框中报告的语法错误

其他资源

本附录概述了如何使用 IDLE 创建、保存和执行程序。IDLE 还提供了其他许多更高级的功能，详情请参见官方 IDLE 文档：www.python.org/idle。

附录 C ASCII 字符集

下表列出了 ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）字符集，它与前 127 个 Unicode 字符码完全一致（这组字符码也被称为 Unicode 的基本拉丁字母子集）。“代码”列显示了字符码，“字符”栏显示了相应的字符。例如，代码 65 代表字母 A。注意，前 31 个代码和代码 127 代表不可打印的控制字符。

代码	字符	代码	字符	代码	字符	代码	字符	代码	字符
0	NUL	26	SUB	52	4	78	N	104	h
1	SOH	27	Escape	53	5	79	O	105	i
2	STX	28	FS	54	6	80	P	106	j
3	ETX	29	GS	55	7	81	Q	107	k
4	EOT	30	RS	56	8	82	R	108	l
5	ENQ	31	US	57	9	83	S	109	m
6	ACK	32	(空格)	58	:	84	T	110	n
7	BEL	33	!	59	;	85	U	111	o
8	Backspace	34	"	60	<	86	V	112	p
9	HTab	35	#	61	=	87	W	113	q
10	换行	36	\$	62	>	88	X	114	r
11	VTab	37	%	63	?	89	Y	115	s
12	换页	38	&	64	@	90	Z	116	t
13	CR	39	'	65	A	91	[117	u
14	SO	40	(66	B	92	\	118	v
15	SI	41)	67	C	93]	119	w
16	DLE	42	*	68	D	94	^	120	x
17	DC1	43	+	69	E	95	_	121	y
18	DC2	44	,	70	F	96	`	122	z
19	DC3	45	-	71	G	97	a	123	{
20	DC4	46	.	72	H	98	b	124	
21	NAK	47	/	73	I	99	c	125	}
22	SYN	48	0	74	J	100	d	126	~
23	ETB	49	1	75	K	101	e	127	DEL
24	CAN	50	2	76	L	102	f		
25	EM	51	3	77	M	103	g		

附录 D 预定义颜色名称

下表展示了预定义的颜色名称，可用于海龟图形库、matplotlib 和 tkinter。

'snow'	'ghost white'	'white smoke'
'gainsboro'	'floral white'	'old lace'
'linen'	'antique white'	'papaya whip'
'blanched almond'	'bisque'	'peach puff'
'navajo white'	'lemon chiffon'	'mint cream'
'azure'	'alice blue'	'lavender'
'lavender blush'	'misty rose'	'dark slate gray'
'dim gray'	'slate gray'	'light slate gray'
'gray'	'light grey'	'midnight blue'
'navy'	'cornflower blue'	'dark slate blue'
'slate blue'	'medium slate blue'	'light slate blue'
'medium blue'	'royal blue'	'blue'
'dodger blue'	'deep sky blue'	'sky blue'
'light sky blue'	'steel blue'	'light steel blue'
'light blue'	'powder blue'	'pale turquoise'
'dark turquoise'	'medium turquoise'	'turquoise'
'cyan'	'light cyan'	'cadet blue'
'medium aquamarine'	'aquamarine'	'dark green'
'dark olive green'	'dark sea green'	'sea green'
'medium sea green'	'light sea green'	'pale green'
'spring green'	'lawn green'	'medium spring green'
'green yellow'	'lime green'	'yellow green'

'forest green'	'olive drab'	'dark khaki'
'khaki'	'pale goldenrod'	'light goldenrod yellow'
'light yellow'	'yellow'	'gold'
'light goldenrod'	'goldenrod'	'dark goldenrod'
'rosy brown'	'indian red'	'saddle brown'
'sandy brown'	'dark salmon'	'salmon'
'light salmon'	'orange'	'dark orange'
'coral'	'light coral'	'tomato'
'orange red'	'red'	'hot pink'
'deep pink'	'pink'	'light pink'
'pale violet red'	'maroon'	'medium violet red'
'violet red'	'medium orchid'	'dark orchid'
'dark violet'	'blue violet'	'purple'
'medium purple'	'thistle'	'snow2'
'snow3'	'snow4'	'seashell2'
'seashell3'	'seashell4'	'AntiqueWhite1'
'AntiqueWhite2'	'AntiqueWhite3'	'AntiqueWhite4'
'bisque2'	'bisque3'	'bisque4'
'PeachPuff2'	'PeachPuff3'	'PeachPuff4'
'NavajoWhite2'	'NavajoWhite3'	'NavajoWhite4'
'LemonChiffon2'	'LemonChiffon3'	'LemonChiffon4'
'cornsilk2'	'cornsilk3'	'cornsilk4'
'ivory2'	'ivory3'	'ivory4'
'honeydew2'	'honeydew3'	'honeydew4'

'LavenderBlush2'	'LavenderBlush3'	'LavenderBlush4'
'MistyRose2'	'MistyRose3'	'MistyRose4'
'azure2'	'azure3'	'azure4'
'SlateBlue1'	'SlateBlue2'	'SlateBlue3'
'SlateBlue4'	'RoyalBlue1'	'RoyalBlue2'
'RoyalBlue3'	'RoyalBlue4'	'blue2'
'blue4'	'DodgerBlue2'	'DodgerBlue3'
'DodgerBlue4'	'SteelBlue1'	'SteelBlue2'
'SteelBlue3'	'SteelBlue4'	'DeepSkyBlue2'
'DeepSkyBlue3'	'DeepSkyBlue4'	'SkyBlue1'
'SkyBlue2'	'SkyBlue3'	'SkyBlue4'
'LightSkyBlue1'	'LightSkyBlue2'	'LightSkyBlue3'
'LightSkyBlue4'	'SlateGray1'	'SlateGray2'
'SlateGray3'	'SlateGray4'	'LightSteelBlue1'
'LightSteelBlue2'	'LightSteelBlue3'	'LightSteelBlue4'
'LightBlue1'	'LightBlue2'	'LightBlue3'
'LightBlue4'	'LightCyan2'	'LightCyan3'
'LightCyan4'	'PaleTurquoise1'	'PaleTurquoise2'
'PaleTurquoise3'	'PaleTurquoise4'	'CadetBlue1'
'CadetBlue2'	'CadetBlue3'	'CadetBlue4'
'turquoise1'	'turquoise2'	'turquoise3'
'turquoise4'	'cyan2'	'cyan3'
'cyan4'	'DarkSlateGray1'	'DarkSlateGray2'
'DarkSlateGray3'	'DarkSlateGray4'	'aquamarine2'

'aquamarine4'	'DarkSeaGreen1'	'DarkSeaGreen2'
'DarkSeaGreen3'	'DarkSeaGreen4'	'SeaGreen1'
'SeaGreen2'	'SeaGreen3'	'PaleGreen1'
'PaleGreen2'	'PaleGreen3'	'PaleGreen4'
'SpringGreen2'	'SpringGreen3'	'SpringGreen4'
'green2'	'green3'	'green4'
'chartreuse2'	'chartreuse3'	'chartreuse4'
'OliveDrab1'	'OliveDrab2'	'OliveDrab4'
'DarkOliveGreen1'	'DarkOliveGreen2'	'DarkOliveGreen3'
'DarkOliveGreen4'	'khaki1'	'khaki2'
'khaki3'	'khaki4'	'LightGoldenrod1'
'LightGoldenrod2'	'LightGoldenrod3'	'LightGoldenrod4'
'LightYellow2'	'LightYellow3'	'LightYellow4'
'yellow2'	'yellow3'	'yellow4'
'gold2'	'gold3'	'gold4'
'goldenrod1'	'goldenrod2'	'goldenrod3'
'goldenrod4'	'DarkGoldenrod1'	'DarkGoldenrod2'
'DarkGoldenrod3'	'DarkGoldenrod4'	'RosyBrown1'
'RosyBrown2'	'RosyBrown3'	'RosyBrown4'
'IndianRed1'	'IndianRed2'	'IndianRed3'
'IndianRed4'	'sienna1'	'sienna2'
'sienna3'	'sienna4'	'burlywood1'
'burlywood2'	'burlywood3'	'burlywood4'
'wheat1'	'wheat2'	'wheat3'

'wheat4'	'tan1'	'tan2'
'tan4'	'chocolate1'	'chocolate2'
'chocolate3'	'firebrick1'	'firebrick2'
'firebrick3'	'firebrick4'	'brown1'
'brown2'	'brown3'	'brown4'
'salmon1'	'salmon2'	'salmon3'
'salmon4'	'LightSalmon2'	'LightSalmon3'
'LightSalmon4'	'orange2'	'orange3'
'orange4'	'DarkOrange1'	'DarkOrange2'
'DarkOrange3'	'DarkOrange4'	'coral1'
'coral2'	'coral3'	'coral4'
'tomato2'	'tomato3'	'tomato4'
'OrangeRed2'	'OrangeRed3'	'OrangeRed4'
'red2'	'red3'	'red4'
'DeepPink2'	'DeepPink3'	'DeepPink4'
'HotPink1'	'HotPink2'	'HotPink3'
'HotPink4'	'pink1'	'pink2'
'pink3'	'pink4'	'LightPink1'
'LightPink2'	'LightPink3'	'LightPink4'
'PaleVioletRed1'	'PaleVioletRed2'	'PaleVioletRed3'
'PaleVioletRed4'	'maroon1'	'maroon2'
'maroon3'	'maroon4'	'VioletRed1'
'VioletRed2'	'VioletRed3'	'VioletRed4'
'magenta2'	'magenta3'	'magenta4'

'orchid1'	'orchid2'	'orchid3'
'orchid4'	'plum1'	'plum2'
'plum3'	'plum4'	'MediumOrchid1'
'MediumOrchid2'	'MediumOrchid3'	'MediumOrchid4'
'DarkOrchid1'	'DarkOrchid2'	'DarkOrchid3'
'DarkOrchid4'	'purple1'	'purple2'
'purple3'	'purple4'	'MediumPurple1'
'MediumPurple2'	'MediumPurple3'	'MediumPurple4'
'thistle1'	'thistle2'	'thistle3'
'thistle4'	'gray1'	'gray2'
'gray3'	'gray4'	'gray5'
'gray6'	'gray7'	'gray8'
'gray9'	'gray10'	'gray11'
'gray12'	'gray13'	'gray14'
'gray15'	'gray16'	'gray17'
'gray18'	'gray19'	'gray20'
'gray21'	'gray22'	'gray23'
'gray24'	'gray25'	'gray26'
'gray27'	'gray28'	'gray29'
'gray30'	'gray31'	'gray32'
'gray33'	'gray34'	'gray35'
'gray36'	'gray37'	'gray38'
'gray39'	'gray40'	'gray42'
'gray43'	'gray44'	'gray45'

'gray46'	'gray47'	'gray48'
'gray49'	'gray50'	'gray51'
'gray52'	'gray53'	'gray54'
'gray55'	'gray56'	'gray57'
'gray58'	'gray59'	'gray60'
'gray61'	'gray62'	'gray63'
'gray64'	'gray65'	'gray66'
'gray67'	'gray68'	'gray69'
'gray70'	'gray71'	'gray72'
'gray73'	'gray74'	'gray75'
'gray76'	'gray77'	'gray78'
'gray79'	'gray80'	'gray81'
'gray82'	'gray83'	'gray84'
'gray85'	'gray86'	'gray87'
'gray88'	'gray89'	'gray90'
'gray91'	'gray92'	'gray93'
'gray94'	'gray95'	'gray97'
'gray98'	'gray99'	

附录 E import 语句

模块（module）是包含函数和/或类的 Python 源代码文件。Python 标准库的许多函数都存储在模块中。例如，`math` 模块包含各种数学函数，而 `random` 模块包含处理随机数的函数。

为了使用存储在一个模块中的函数和/或类，你必须先导入该模块。为了导入模块，需要在程序顶部写一个 `import` 语句。以下 `import` 语句导入 `math` 模块：

```
import math
```

该语句指示 Python 解释器将 `math` 模块的内容加载到内存，使程序能使用存储在 `math` 模块中的函数和/或类。为了使用模块中的任何项，必须使用该项的 *限定名称*。这意味着必须在该项的名称前加上模块名称，并后跟一个点号。例如，`math` 模块包含一个求平方根的 `sqrt` 函数，为了调用 `sqrt` 函数，需要使用 `math.sqrt` 这个限定名称。以下交互会话展示了一个例子：

```
>>> import math   
>>> x = math.sqrt(25)   
>>> print(x)   
5.0  
>>>
```

导入特定函数或类

之前的 `import` 语句将一个模块的全部内容加载到内存。有的时候，你只想从模块中导入一个特定的函数或类。在这种情况下，可以在使用 `import` 语句时添加 `from` 关键字，如下所示：

```
from math import sqrt
```

该语句只从 `math` 模块中导入 `sqrt` 函数。像这样写，还可以直接调用 `sqrt` 函数而不必在函数名称前附加模块名称前缀。以下交互会话展示了一个例子：

```
>>> from math import sqrt   
>>> x = sqrt(25)   
>>> print(x)   
5.0  
>>>
```

使用这种形式的 `import` 语句，还可以指定多个要导入的项的名称，不同项以逗号分隔。例如，以下会话中的 `import` 语句只从 `math` 模块中导入 `sqrt` 函数和 `radians` 函数：

```
>>> from math import sqrt, radians   
>>> x = sqrt(25)   
>>> a = radians(180) 
```

```
>>> print(x) 
5.0
>>> print(a) 
3.141592653589793
>>>
```

通配符导入

通配符 `import` 语句加载模块的全部内容，如下例所示：

```
from math import *
```

该语句和 `import math` 语句的区别在于，通配符 `import` 语句不要求使用模块名称来限定。例如，以下交互会话使用了一个通配符 `import` 语句：

```
>>> from math import* 
>>> x = sqrt(25) 
>>> a = radians(180) 
>>>
```

以下交互会话使用的则是普通的 `import` 语句：

```
>>> import math 
>>> x = math.sqrt(25) 
>>> a = math.radians(180) 
>>>
```

通常应避免使用通配符 `import` 语句，因为在导入多个模块时，它可能造成名称冲突。如果程序导入的两个模块包含同名的函数或类，就会发生名称冲突。但是，如果用模块名称来限定函数和/或类，就不会发生名称冲突。

使用别名

可以使用 `as` 关键字为导入的模块使用别名，如下例所示：

```
import math as mt
```

该语句将 `math` 模块加载到内存，并为该模块分配别名 `mt`。要使用模块中的任何项，可以在该项的名称前加上别名，后跟一个点号。例如，为了调用 `sqrt` 函数，现在可以使用 `mt.sqrt` 这个名称，如下所示：

```
>>> import math as mt 
>>> x = mt.sqrt(25) 
>>> a = mt.radians(180) 
>>> print(x) 
5.0
```

```
>>> print(a) 3.141592653589793
>>>
```

也可以为导入的特定函数或类分配别名。以下语句从 `math` 模块中导入 `sqrt` 函数，并为该函数分配别名 `square_root`：

```
from math import sqrt as square_root
```

使用这个 `import` 语句后，以后就可以用 `square_root` 这个名称来调用 `sqrt` 函数，如下例所示：

```
>>> from math import sqrt as square_root 
>>> x = square_root(25) 
>>> print(x) 
5.0
>>>
```

在以下交互会话中，我们从 `math` 模块导入两个函数，并分别分配了别名。`sqrt` 函数作为 `square_root` 导入，而 `tan` 函数作为 `tangent` 导入：

```
>>> from math import sqrt as square_root, tan as tangent 
>>> x = square_root(25) 
>>> y = tangent(45) 
>>> print(x) 
5.0
>>> print(y) 
1.6197751905438615
```

附录 F 用 format() 函数格式化输出



注意：本书正文使用 f 字符串作为首选的输出格式化方法。f 字符串是自 Python 3.6 引入的。如果使用的是更早的 Python 版本，那么可以考虑使用本附录描述的 format() 函数。

程序 F-1 (no_formatting.py)

```
1 # 这个程序演示了在不格式化的情况下，
2 # 一个浮点数是如何显示的。
3 amount_due = 5000.0
4 monthly_payment = amount_due / 12.0
5 print('每月付款金额为', monthly_payment)
```

程序输出

```
每月付款金额为 416.6666666666667
```

对于数字（特别是浮点数）的默认显示方式，你不一定会感到满意。print 函数在打印一个浮点数时，它最多可以有 17 位有效数字，如程序 F-1 的输出所示。

由于这个程序显示的是货币金额，所以最好四舍五入为两位小数。幸好，可以通过 Python 内置的 format 函数来达到这个目的，该函数甚至可以做更多的事情。

调用内置 format 函数时，要向函数传递两个实参：一个数值和一个格式说明符。**格式说明符**（format specifier）是包含特殊字符的一个字符串，它指定了如何对数值进行格式化，如下例所示：

```
format(12345.6789, '.2f')
```

第一个实参是浮点数 12345.6789，是我们想要格式化的数字。第二个实参是字符串 '.2f'，这就是格式说明符。下面解释了它的含义：

- .2 指定精度，表示要将数字四舍五入到小数点后两位。
- f 指定要格式化的数字的数据类型是浮点数。

format 函数返回包含格式化好的数字的一个字符串。以下交互会话展示了如何在 print 函数中使用 format 函数来显示一个格式化的数字：

```
>>> print(format(12345.6789, '.2f')) Enter
12345.68
```

```
>>>
```

注意，数字被四舍五入为两位小数。下例将同一个数字四舍五入为一位小数：

```
>>> print(format(12345.6789, '.1f'))   
12345.7  
>>>
```

下面是另一个例子：

```
>>> print('The number is', format(1.234567, '.2f'))   
The number is 1.23  
>>>
```

程序 F-2 展示了如何修改程序 F-1，用这个技术来格式化它的输出。

程序 F-2 (formatting.py)

```
1 # 这个程序演示了如何  
2 # 对浮点数进行格式化。  
3 amount_due = 5000.0  
4 monthly_payment = amount_due / 12  
5 print('每月付款金额为',  
6       format(monthly_payment, '.2f'))
```

程序输出

```
每月付款金额为 416.67
```

用科学计数法格式化

如果希望用科学记数法显示浮点数，可以用字母 e 或 E 代替 f，下面是一些例子：

```
>>> print(format(12345.6789, 'e'))   
1.234568e+04  
>>> print(format(12345.6789, '.2e'))   
1.23e+04  
>>>
```

第一个语句用科学记数法简单格式化数字。字母 e 之后的数字表示以 10 为底的指数。（如果在格式说明符中使用大写 E，结果也会用大写 E 来表示指数）。第二个语句额外指定了在小数点后保留两位精度。

插入逗号分隔符

如果想用逗号（千位）分隔符对数字进行格式化，可以在格式说明符中插入一个逗号，如下所示：

```
>>> print(format(12345.6789, ',.2f')) Enter
12,345.68
>>>
```

下例格式化一个更大的数：

```
>>> print(format(123456789.456, ',.2f')) Enter
123,456,789.46
>>>
```

注意，在格式说明符中，逗号要放在精度指定符之前（在其左侧）。下例只指定了千位逗号分隔符，而不指定精度：

```
>>> print(format(12345.6789, ',f')) Enter
12,345.678900
>>>
```

程序 F-3 演示了如何利用逗号分隔符和两位小数精度将一个较大的数字格式化为货币金额。

程序 F-3 (dollar_display.py)

```
1 # 这个程序演示了如何将浮点数
2 # 显示为货币金额。
3 monthly_pay = 5000.0
4 annual_pay = monthly_pay * 12
5 print('你的年度付款金额是$',
6       format(annual_pay, ',.2f'),
7       sep='')
```

程序输出

```
你的年度付款金额是$60,000.00
```

注意，第 7 行向 `print` 函数传递了实参 `sep=''`。它指定在显示的各项之间不添加空格。如果不传递这个实参，`$` 符号后会默认打印一个空格。

指定最小域宽

格式说明符还可以包含一个**最小域宽**（minimum field width）^①，即显示一个值时应占用的最小空格数。下例在 12 个字符宽的域（字段）中打印一个数字：

```
>>> print('The number is', format(12345.6789, '12,.2f')) Enter
The number is    12,345.68
```

^① 译注：也称为最小字段宽度。


```
>>>
```

格式说明符中的 12 表示应在一个至少 12 个字符宽的域中显示当前要打印的值。在本例中，要显示的值比它所在的域更短。数字 12,345.68 在屏幕上只占了 9 个字符的宽度，但它显示在一个 12 个字符宽的域中。在这种情况下，该数字在域中右对齐。如果一个值过大，超过了所在域的宽度，域会自动扩大以适应它。

请注意，在前面的例子中，域宽指示符要放在逗号分隔符之前（在其左侧）。下例指定了域宽和精度，但没有使用逗号（千位）分隔符：

```
>>> print('The number is', format(12345.6789, '12.2f'))   
The number is 12345.68  
>>>
```

通过指定域宽，我们可以打印在一列中对齐的数字。例如，在程序 F-4 中，每个变量的值都在 7 个字符宽度的域中显示。

程序 F-4 (columns.py)

```
1 # 这个程序在一列中打印  
2 # 浮点数，其小数点在垂直  
3 # 方向对齐。  
4 num1 = 127.899  
5 num2 = 3465.148  
6 num3 = 3.776  
7 num4 = 264.821  
8 num5 = 88.081  
9 num6 = 799.999  
10  
11 # 在 7 个字符宽的域内显示  
12 # 每个数字，并保留 2 位小数。  
13 print(format(num1, '7.2f'))  
14 print(format(num2, '7.2f'))  
15 print(format(num3, '7.2f'))  
16 print(format(num4, '7.2f'))  
17 print(format(num5, '7.2f'))  
18 print(format(num6, '7.2f'))
```

程序输出

```
127.90  
3465.15  
3.78  
264.82  
88.08  
800.00
```

将浮点数格式化为百分比

可以使用%将浮点数格式化为百分比，而不是使用 f 作为类型指示符。%会使数字乘以 100，并在后面显示一个%符号，如下例所示：

```
>>> print(format(0.5, '%'))   
50.000000%  
>>>
```

下例使输出的值不保留任何小数：

```
>>> print(format(0.5, '.0%'))   
50%  
>>>
```

格式化整数

之前所有的例子演示的都是如何对浮点数进行格式化。还可以使用 format 函数格式化整数。和浮点数相比，整数的格式说明符有两个不同之处需要注意：

- 使用 d 作为类型指示符。
- 不可指定精度。

下面展示了交互式解释器中的一些例子。以下会话打印 123456，不进行任何特殊的格式化：

```
>>> print(format(123456, 'd'))   
123456  
>>>
```

以下会话同样打印 123456，但添加了千位逗号分隔符：

```
>>> print(format(123456, ',d'))   
123,456  
>>>
```

以下会话在 10 字符宽的域中打印 123456：

```
>>> print(format(123456, '10d'))   
123456  
>>>
```

以下会话在 10 字符宽的域中打印以逗号作为千位分隔符的 123456：

```
>>> print(format(123456, '10,d'))   
123,456  
>>>
```

附录 G 用 pip 工具安装模块

Python 标准库提供了可供程序执行基本操作以及许多高级任务的类和函数。然而，有的操作是标准库无法执行的。需要做一些超出标准库范围的事情时，你有两个选择：自己写代码，或者使用别人已经写好的代码。

幸好，目前已经有成千上万的 Python 模块，它们由独立的程序员编写，提供了标准 Python 库不具备的功能。这些模块称为**第三方模块**。官方网站 pypi.org 罗列了大量第三方模块，PyPI 是 Python Package Index 的简称。

在 PyPI 上提供的模块以“包”的形式组织。所谓“包”（package），其实就是一个或多个相关模块的组合。下载和安装一个包最简单的方法是使用 pip 工具。pip 工具是一个实用程序，是标准 Python 安装的一部分，自 Python 3.4 开始引入。要用 pip 工具在 Windows 系统上安装一个包，必须打开一个命令提示符窗口，然后按以下格式输入命令：

```
pip install package_name
```

package_name 是想要下载和安装的包的名称。如果使用的是 Mac 或 Linux 系统，那么要换成 pip3 命令而不是 pip 命令。另外，需要有超级用户权限才能在 Mac 或 Linux 系统上执行 pip3 命令，所以可能需要在命令前加上 sudo，如下所示：

```
sudo pip3 install package_name
```

一旦输入该命令，pip 工具将开始下载并安装包^①。根据包的大小，可能需要几分钟的时间来完成安装过程。一旦这个过程完成，通常可以通过启动 IDLE 并输入以下命令来验证包是否被正确安装：

```
>>> import package_name
```

其中 *package_name* 是你安装的包的名称。如果没有出现错误消息，就可以认为包已成功安装。

^① 译注：要卸载一个已安装的包，将 install 换成 uninstall 即可。

附录 H 知识检查点答案

第 1 章

1.1 程序是一组指令，计算机遵循这些指令来执行任务。

1.2 硬件是构成一台计算机的所有物理设备（或称组件）。

1.3 中央处理器（CPU）、主存、辅助存储设备、输入设备和输出设备。

1.4 CPU

1.5 主存

1.6 辅助存储设备

1.7 输入设备

1.8 输出设备

1.9 操作系统

1.10 实用程序

1.11 应用软件

1.12 一个字节

1.13 一个二进制位（比特）

1.14 二进制数字系统

1.15 它是一种编码方案，使用一组 128 个数字代码来表示英文字母、各种标点符号和其他字符。这些数字代码被用来在计算机内存中存储字符。(ASCII 是美国信息交换标准代码的缩写。)

1.16 Unicode

1.17 数字数据是以二进制存储的数据，而数字设备是任何处理二进制数据的设备。

1.18 机器语言

1.19 主存，或称 RAM

1.20 取回 - 解码 - 执行（fetch-decode-execute）周期

1.21 机器语言的替代品。汇编语言不是用二进制数字表示指令，而是使用称为“助记符”的短字。

1.22 高级语言

1.23 语法

1.24 编译器

1.25 解释器

1.26 语法错误

第 2 章

2.1 你为之写程序的任何人、团队或组织。

2.2 程序必须执行的一项任务。

2.3 一组明确定义的逻辑步骤，必须采取这些步骤来执行一项任务。

2.4 一种非正式语言，没有语法规则，目的也不是被编译或执行。相反，程序员使用伪代码来创建程序的模型。

2.5 描述程序步骤的一张示意图。

2.6 椭圆代表终端。平行四边形代表输入或输出。矩形代表处理（例如数学计算）。

2.7 `print('Jimmy Smith')`

2.8 `print("Python's the best!")`

2.9 `print('The cat said "meow"')`

2.10 变量是引用了计算机内存中的值的一个名称，或者说是一个指针。

2.11 `99bottles` 是非法的变量名，因为它以一个数字开头。`r&d` 也是非法的变量名，因为变量名不允许包含 `&`。

2.12 不是同一个变量，变量名要区分大小写。

2.13 该赋值语句无效，因为接收赋值的变量（`amount`）必须在赋值操作符 `=` 的左侧。

2.14 `The value is val`

2.15 `value1` 将引用一个 `int`，`value2` 将引用一个 `float`，`value3` 将引用一个 `float`，

value4 将引用一个 int，而 values 将引用一个 str（字符串）。

2.16 0

2.17 last_name = input("输入客户的姓氏：")

2.18 sales= float(input('输入本周销售额:'))

2.19 填好的表格如所示：

表达式	数值
6 + 3 * 5	21
12 / 2 - 4	2
9 + 14 * 2 - 6	31
(6 + 2) * 3	24
14 / (11 - 4)	2
9 + 12 * (8 - 3)	69

2.20 4

2.21 1

2.22 字符串连接（拼接）是指将一个字符串附加到另一个字符串末尾。

2.23 '12'

2.24 'hello'

2.25 如果不希望 print 函数在打印完当前行的内容后自动换行，可以向函数传递特殊实参 end = '' 来取代默认的换行符。

2.26 可以向 print 函数传递实参 sep=来指定你希望的分隔符。

2.27 一个换行符

2.28 Hello {name}

2.29 Hello Karlie

2.30 The value is 100

2.31 The value is 65.43

2.32 The value is 987,654.13

2.33 The value is 9,876,543,210

2.34 是域宽指示符，指定值在最小 10 个字符宽的域中显示。

2.35 是域宽指示符，指定值在最小 15 个字符宽的域中显示。

2.36 是域宽指示符，指定值在最小 8 字符宽的域中显示。

2.37 是对齐指示符，指定值左对齐。

2.38 是对齐指示符，指定值右对齐。

2.39 是对齐指示符，指定值居中对齐。

2.40 (1)具名常量增强了程序的可读性；(2)方便大面积修改程序；(3)防范使用“魔法数字”时常见的打字错误。

2.41 `DISCOUNT_PERCENTAGE = 0.1`

2.42 0 度。

2.43 使用 `turtle.forward` 命令。

2.44 使用 `turtle.right(45)`命令。

2.45 先用 `turtle.penup()`命令将笔抬起。

2.46 `turtle.heading()`

2.47 `turtle.circle(100)`

2.48 `turtle.pensize(8)`

2.49 `turtle.pencolor('blue')`

2.50 `turtle.bgcolor('black')`

2.51 `turtle.setup(500, 200)`

2.52 `turtle.goto(100, 50)`

2.53 `turtle.pos()`

2.54 `turtle.speed(10)`

2.55 `turtle.speed(0)`

2.56 要在形状中填色，要在绘制形状前使用 `turtle.begin_fill()`命令，形状绘制好之后使用 `turtle.end_fill()`命令开始填充。执行 `turtle.end_fill()`命令时，会用当前填充颜色来填充形状、

2.57 使用 `turtle.write()` 命令。

2.58 `radius = turtle.numinput('输入一个数值', '圆的半径是多大?')`

第 3 章

3.1 一种逻辑设计，用于控制一组语句的执行顺序。

3.2 一种程序结构，仅在满足特定条件时才执行一组语句。

3.3 一种判断结构，只提供了一个分支执行路径。测试的条件为真，程序就会选择分支路径。

3.4 求值为真或假的一种表达式。

3.5 可以判断一个值是否大于、小于、大于等于、小于等于、等于或者不等于另一个值。

3.6

```
if y == 20:  
    x = 0
```

3.7

```
if sales >= 10000:  
    commissionRate = 0.2
```

3.8 双分支判断结构有两个可能的执行路径；一个在条件为真时执行，一个在条件为假时执行。

3.9 if-else

3.10 条件为假时。

3.11 z 不小于 a

3.12

```
Boston  
New York
```

3.13

```
if number == 1:  
    print('One')  
elif number == 2:  
    print('Two')  
elif number == 3:
```



```
    print('Three')
else:
    print('Unknown')
```

3.14 由逻辑操作符连接两个布尔子表达式而成的一个表达式。

3.15

```
F
T
F
F
T
T
T
F
F
T
```

3.16

```
T
F
T
T
T
```

3.17 **and** 操作符：操作符左侧的表达式为 **false**，就立即“短路”，不求值右侧的表达式。**or** 操作符：操作符左侧的表达式为 **true**，就立即“短路”，不求值右侧的表达式。

3.18

```
if speed >= 0 and speed <= 200:
    print('数字有效')
```

3.19

```
if speed < 0 or speed > 200:
    print('数字无效')
```

3.20 True 或 False

3.21 表明程序中是否存在某个条件的变量。

3.22 使用 `turtle.xcor()`和 `turtle.ycor()`函数。

3.23 将 **not** 操作符应用于 `turtle.isdown()`函数，如下所示：

```
if turtle.isdown():
    语句
```

3.24 使用 `turtle.heading()` 函数

3.25 使用 `turtle.isvisible()` 函数。

3.26 使用 `turtle.pencolor()` 函数判断画笔颜色。使用 `turtle.fillcolor()` 函数判断当前填充颜色。使用 `turtle.bgcolor()` 函数判断海龟图形窗口的当前背景颜色。

3.27 使用 `turtle.pensize()` 函数。

3.28 使用 `turtle.speed()` 函数。

第 4 章

4.1 导致重复执行一个或一组语句的结构。

4.2 使用真/假条件来控制重复次数的循环。

4.3 重复特定次数的循环。

4.4 执行一遍循环体内的语句，就称为一次“迭代”。

4.5 之前。

4.6 一次都不会。条件 `count < 0` 一开始就为假。

4.7 没有使循环条件变成假的机制，循环一直停不下来。会一直重复，程序只能强行中断。

4.8

```
for x in range(6):  
    print('我爱编程!')
```

4.9

```
0  
1  
2  
3  
4  
5
```

4.10

```
2  
3  
4  
5
```

4.11

```
0
100
200
300
400
500
```

4.12

```
10
9
8
7
6
```

4.13 对一系列数字进行累加的一个变量。

4.14 是的，它需要初始化为值 0。这是由于值通过循环将多个值加到一个累加器变量上。如果累加器一开始不为 0，那么在循环结束时，它包含的就不是正确的总和。

4.15 15

4.16

```
15
5
```

4.17

```
a) quantity += 1
b) days_left -= 5
c) price *= 10
d) price /= 2
```

4.18 哨兵是标志值序列结尾一个特殊值。

4.19 只有足够特殊的哨兵值才不会和列表中的常规值发生混淆。

4.20 这意味着如果将错误的输入（垃圾）作为程序的输入，那么程序也将产生错误的输出（垃圾）作为输出。

4.21 提供给程序的输入应在使用之前进行检查。如果输入无效，那么应将其丢弃并提示用户输入正确的数据。

4.22 首先读取输入，然后执行一个预测试循环。如果输入数据无效，那么执行循环体。循环体内显示错误消息，使用户知道输入无效，并重新读取输入。只要输入无效，循环就会一直迭代。

4.23 “预读”是指在输入校验循环之前进行的一次输入操作。预读的目的是获取第一个输入值。

4.24 一次都不会。

第 5 章

5.1 函数是程序中用于执行特定任务的一组语句。

5.2 将一个大型任务分解为几个容易执行的小任务。

5.3 如果一个特定的操作要在程序中的多个地方执行，那么可以写一个函数来执行该操作，并在需要时调用该函数。

5.4 将不同程序都需要的通用任务写成函数，并在需要的地方调用。

5.5 将程序分解为一系列函数，每个函数都执行一个单独的任务，并为不同的程序员分配写一个函数的任务。

5.6 函数定义的两部分是函数头和语句块（简称“块”或“代码块”）。函数头代表函数定义的开始。块是函数主体，是调用函数时会执行的一组语句。

5.7 调用函数意味着执行函数（中的代码）。

5.8 到达函数末尾时，计算机返回函数的调用位置，并从这里恢复程序的执行。

5.9 因为 Python 解释器根据缩进来判断一个块的起始和结束位置。

5.10 局部变量是在函数内部声明的变量。它属于声明它的函数，只有同一函数中的语句才能访问它。

5.11 可以访问变量的那一部分程序。

5.12 是的，允许。

5.13 实参。

5.14 形参。

5.15 形参变量的作用域是声明它的整个函数

5.16 不会。

5.17

a. 传递关键字实参

b. 按位置传递

5.18 整个程序文件。

5.19 有三个理由：

- 全局变量使调试变得困难。程序文件中的任何语句都可以更改全局变量的值。如果发现全局变量中存储了错误的值，那么必须追踪每一个访问全局变量的语句，以确定错误值的来源。在有数千行代码的一个程序中，这会令人抓狂。
- 使用全局变量的函数通常会依赖于这些变量。如果想在不同的程序中使用这种函数，那么很可能必须重新设计，使其不依赖于全局变量。
- 全局变量使程序难以理解。程序中的任何语句都可以修改全局变量。为了理解程序中使用了全局变量的任何部分，都必须理解程序中访问了该全局变量的其他所有部分。

5.20 全局常量是程序中每个函数都可以访问的名称。和全局变量不同，全局常量适合在程序中使用。程序执行期间，它的值不会改变，所以不必担心会被修改。

5.21 区别在于，返回值函数会将一个值返回给调用它的程序部分。void 函数则不会返回值。

5.22 用于执行一些常见任务的由语言预定义的函数。

5.23 “黑盒”描述的是接受输入、使用输入来执行某些操作（不清楚细节）并生成输出的任何机制。库函数符合这一描述。

5.24 为变量 x 赋值 1~100 的随机整数。

5.25 打印 1~20 的随机整数。

5.26 打印 10~19 的随机整数。

5.27 打印 0.0~1.0（不含 1.0）的随机浮点数。

5.28 打印 0.1~0.5 的随机浮点数。

5.29 使用从计算机内部时钟获取的系统时间。

5.30 如果始终使用同一个种子值，那么随机数函数将始终生成相同的伪随机数序列。

5.31 将一个值返回给调用它的那个程序部分。

5.32

a) do_something

b) 返回实参值乘以 2 的结果。

c) 20

5.33 返回 True 或 False 的函数。

5.34 `import math`

5.35 `square_root = math.sqrt(100)`

5.36 `angle = math.radians(45)`

第 6 章

6.1 程序向其写入数据的文件。之所以称为输出文件，是因为程序向其发送输出。

6.2 程序从中读取数据的文件。之所以称为输入文件，是因为程序从中接收输入。

6.3 (1) 打开文件。 (2) 处理文件。 (3) 关闭文件。

6.4 文本文件和二进制文件。文本文件包含使用 ASCII 等方案编码为文本的数据。即使文件中包含数字，这些数字也以一系列字符的形式存储在文件中。因此，可以使用“记事本”等文本编辑器打开和查看文件。二进制文件则包含未转换为文本的数据。因此，无法使用文本编辑器查看二进制文件的内容，打开只会显示一些“乱码”。

6.5 顺序访问和直接访问。使用顺序访问文件时，必须从文件开头到文件末尾依次访问数据。而在使用直接访问文件时，可以直接跳转到文件中的任何数据，而不需要读取它之前的数据。

6.6 文件在磁盘上的名称和引用文件对象的变量的名称。

6.7 文件中现有的内容会被删除。

6.8 打开文件会创建文件与程序之间的连接。它还会创建文件与文件对象之间的关联。

6.9 关闭文件会断开程序与文件之间的连接。

6.10 文件的读取位置标记着将从文件的什么位置读取下一个数据项。打开输入文件时，其读取位置最初会被设为文件中第一个数据项开头的位置。

6.11 以追加模式（'a'）打开文件。在追加模式下写入数据时，数据会被写入文件现有内容的末尾。

6.12

```
outfile = open('numbers.txt', 'w')
for num in range(1, 11):
```

```
        outfile.write(str(num) + '\n')
outfile.close()
```

6.13 `readline` 方法如果返回空字符串（''），那么意味着它试图超出文件末尾进行读取。

6.14

```
infile = open('data.txt', 'r')
line = infile.readline()
while line != '':
    print(line)
    line = infile.readline()
infile.close()
```

6.15

```
infile = open('data.txt', 'r')
for line in infile:
    print(line)
infile.close()
```

6.16 记录是对一个数据项进行描述的完整数据集，字段则是记录内的单个数据项。

6.17 将所有原始文件的记录复制到临时文件，但在到达要修改的记录时，不要将它旧的内容写入临时文件。相反，将新的修改值写入临时文件。然后，继续将剩余记录从原始文件复制到临时文件。最后用临时文件取代原始文件，即删除原始文件并将临时文件重命名为原始文件的名称。

6.18 将原始文件中除了要删除的记录之外的其他所有记录复制到临时文件。然后，用临时文件取代原始文件，即删除原始文件并将临时文件重命名为原始文件的名称。

6.19 异常是程序运行时发生的错误。大多数时候，异常会导致程序突然停止。

6.20 程序停止。

6.21 `FileNotFoundError`

6.22 `ValueError`

第 7 章

7.1 [1, 2, 99, 4, 5]

7.2 [0, 1, 2]

7.3 [10, 10, 10, 10, 10]

7.4

```
1  
3  
5  
7  
9
```

7.5 4

7.6 使用语言内置的 `len` 函数。

7.7

```
[1, 2, 3]  
[10, 20, 30]  
[1, 2, 3, 10, 20, 30]
```

7.8

```
[1, 2, 3]  
[10, 20, 30, 1, 2, 3]
```

7.9 [2, 3]

7.10 [2, 3]

7.11 [1]

7.12 [1, 2, 3, 4, 5]

7.13 [3, 4, 5]

7.14

```
Jasmine's family:  
['Jim', 'Jill', 'John', 'Jasmine']
```

7.15 `remove` 方法用于搜索并移除包含特定值的元素。`del` 语句用于删除指定索引位置的元素，不管该元素的值是什么。

7.16 可以使用 Python 内置的 `min` 和 `max` 函数。

7.17 应该使用语句 `b`，即 `names.append('Wendy')`。由于此时元素 0 还不存在，所以使用语句 `a` 来访问索引位置 0 处的元素会导致出错。

7.18

a) `index` 方法在列表中查找指定元素，并返回包含该元素的第一个元素的索引。

b) `insert` 方法将一个元素插入列表的指定索引位置。

c) `sort` 方法将列表中的元素按升序排序。

d) `reverse` 方法反转列表中所有元素的顺序。

7.19 结果表达式是：`x`。迭代表达式是：`for x in my_list`。

7.20 `[2, 24, 4, 40, 6, 30, 8]`

7.21 `[12, 20, 15]`

7.22 列表包含 4 行 2 列。

7.23 `mylist = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]`

7.24

```
for r in range(4):
    for c in range(2):
        print(numbers[r][c])
```

7.25 元组和列表的主要区别在于元组是不可变的。这意味着元组一旦创建就无法更改。

7.26 下面列举了三个原因：

- 处理元组比处理列表快，因此当处理大量数据且该数据不会发生改变时，元组是一个不错的选择。
- 元组是安全的。由于不允许更改元组的内容，所以存储在其中的数据可以保证不会被程序中的任何代码（无论是意外还是其他方式）修改。
- Python 中的一些操作要求使用元组。

7.27 `my_tuple = tuple(my_list)`

7.28 `my_list = list(my_tuple)`

7.29 必须传递两个列表：一个包含数据点的 X 坐标，另一个包含它们的 Y 坐标。

7.30 折线图

7.31 使用 `xlabel` 和 `ylabel` 函数。

7.32 调用 `xlim` 和 `ylim` 函数，传递 `xmin`，`xmax`，`ymin` 和 `ymax` 关键字参数。

7.33 调用 `xticks` 和 `yticks` 函数。向这些函数传递两个参数。第一个参数是刻度线位置的列表，第二个参数是在指定位置显示的标签列表。

7.34 两个列表：一个包含每个条柱的 X 坐标，另一个包含每个条柱在 Y 轴上的高度。

7.35 条柱分别为红色、蓝色、红色和蓝色。

7.36 向该函数传递一个值列表作为实参。`pie` 函数将计算列表中值的总和，然后将该总和作为整体值。然后，列表中的每个元素将成为饼图中的一个切片（扇区）。每个切片的大小表示该元素值占整体值的百分比。

第 8 章

8.1

```
for letter in name:  
    print(letter)
```

8.2 0

8.3 9

8.4 如果尝试使用超出特定字符串范围的索引，将会引发 `IndexError` 异常。

8.5 使用内置的 `len` 函数。

8.6 第二个语句试图向字符串中的一个字符赋值。然而，字符串是不可变的，所以表达式 `animal[0]` 不能出现在赋值操作符的左侧。

8.9 abc

8.10 abcdefg

8.11

```
if 'd' in mystring:  
    print('是的，其中有"d"。')
```

8.12 `little = big.upper()`

8.13

```
if ch.isdigit():  
    print('Digit')  
else:  
    print('No digit')
```

8.14 a A

8.15

```
again = input('你想要重复' + '程序还是退出? (R/Q)')  
while again.upper() != 'R' and again.upper() != 'Q':  
    again = input('你想要重复' + '程序还是退出? (R/Q)')
```

8.16 \$

8.17

```
for letter in mystring:
    if letter.isupper():
        count += 1
```

8.18 `my_list = days.split()`

8.19 `my_list = values.split('$')`

第 9 章

9.1 键和值。

9.2 键。

9.3 字符串 'start' 是键，整数 1472 是值。

9.4 它在 `employee` 字典中存储键值对 'id' : 54321。

9.5 显示 'ccc'。

9.6 可以使用 `in` 操作符来测试是否存在特定的键，进而判断是否存在特定的键值对。

9.7 删除键为 654 的那个元素（整个键值对）。

9.8 显示 3。

9.9

```
1
2
3
```

9.10 `pop()` 方法接收一个键作为参数，返回与该键关联的值，然后从字典中删除该键值对。
`popitem()` 方法返回字典中最后添加的那个键值对（以元组形式），然后从字典中删除该键值对。

9.11 `items()` 方法以元组序列（字典视图）的形式返回字典的所有键及其关联的值。

9.12 `keys()` 方法以元组序列（字典视图）的形式返回字典的所有键。

9.13 `values()` 方法以元组序列（字典视图）的形式返回字典的所有值。

9.14 `result = {item:len(item) for item in names}`

9.15 `phonebook_copy = {k:v for k,v in phonebook.items() if v.startswith('919')}`

9.16 集合元素是无序的。

9.17 不允许。

9.18 可以调用内置的 `set` 函数。

9.19 集合中的元素将是 'j'、'u'、'p'、'i'、't'、'e' 和 'r'。注意，本题包括后续各题的集合元素均无固定顺序。

9.20 集合中的元素将是 25。

9.21 集合中的元素将是 'w'、' '、'x'、'y' 和 'z'。

9.22 集合中的元素将是 1、2、3 和 4。

9.23 集合中的元素将是 'www'、'xxx'、'yyy' 和 'zzz'。

9.24 将集合作为实参传递给 `len` 函数。

9.25 集合中的元素将是 10、9、8、1、2 和 3。

9.26 集合中的元素将是 10、9、8、'a'、'b' 和 'c'。

9.27 如果要删除的元素不在集合中，`remove` 方法会引发 `KeyError` 异常，`discard` 方法则不会。

9.28 可以使用 `in` 操作符来判断元素是否在集合中。

9.29 `{10, 20, 30, 100, 200, 300}`

9.30 `{3, 4}`

9.31 `{1, 2}`

9.32 `{5, 6}`

9.33 `{'a', 'd'}`

9.34 `set2` 是 `set1` 的子集，而 `set1` 是 `set2` 的超集。

9.35 对象序列化是指将对象转换为字节流，以便保存到文件中供将来取回并恢复。

9.36 使用 `'wb'` 以二进制写入模式打开。

9.37 使用 `'rb'` 以二进制读取模式打开。

9.38 pickle 模块。

9.39 pickle.dump

9.40 pickle.load

第 10 章

10.1 对象是同时包含数据和方法的一个软件实体。

10.2 封装是指将数据和代码合并到一个对象中。

10.3 当一个对象的数据属性对外部代码隐藏，而且只有对象的方法才能访问数据属性时，可以防范数据属性被意外破坏。此外，外部代码不需要知道对象数据的格式或内部结构。

10.4 公共方法可直接由对象外部的实体访问。私有方法则不能，它们被设计成在内部访问。

10.5 蓝图代表的是类。

10.6 饼干是对象。

10.7 它的作用是初始化对象的数据属性，会在对象创建后立即执行。

10.8 类的方法在执行时，必须知道它应该操作哪个对象的数据属性。这就是 `self` 参数的作用。在调用方法时，Python 会自动使 `self` 参数引用方法应该操作的那个特定的对象。

10.9 通过在属性名称前加两个下划线来私有化属性。

10.10 它返回对象（状态）的一个字符串表示。

10.11 将对象传递给内置的 `str` 方法即可自动调用。

10.12 实例属性是指从属于类的特定实例的属性。

10.13 10 个。

10.14 返回值的方法称为取值方法。存储值或以其他方式更改数据属性的方法称为赋值方法。

10.15 在这三个部分中，顶部写上类名，中间列出类的字段，底部则列出类的方法。

10.16 问题域是在现实世界中，与当前要解决的问题相关的对象、参与方以及主要事件的集合。

10.17 如果你充分理解了要解决的问题的性质，那么可以自己撰写问题域描述。如果对问题的性质不够理解，可以请专家为你撰写这个描述。

10.18 首先，识别问题域描述中出现的名词、代词和名词短语。然后，完善这个列表以消除重复项、和当前问题无关的项、本质上是对象而非类的项以及可以用变量来存储的简单的值。

10.19 类需要知道的事情和需要执行的行动。

10.20 就当前问题来说，类需要知道什么？类需要做什么？

10.21 不一定，取决于问题的复杂性。

第 11 章

11.1 超类是泛化类，子类是特化类。

11.2 当一个对象是另一个对象的特化版本时，它们之间就存在“属于”关系。特化类的对象是泛化类对象的一个特殊的版本。

11.3 子类继承了超类的所有属性和方法。

11.4 Bird 是超类，Canary 是子类。

11.5

我是蔬菜。
我是土豆。

第 12 章

12.1 递归算法需要执行多次方法/函数调用。每次调用都需要系统执行一些准备行动，包括为参数和局部变量分配内存，并存储每个函数调用的返回位置等。所有这些行动和相关的资源统称为开销。而在使用循环来实现的迭代算法中，这种开销是不需要的。

12.2 可以在不继续递归的情况下解决问题的情况。

12.3 需要递归来解决问题的情况。

12.4 当递归到达基本情况时。

12.5 在直接递归中，递归方法调用自身。而在间接递归中，方法 A 调用方法 B，方法 B 又调用方法 A。

第 13 章

13.1 计算机和操作系统中用户与之交互的部分。

13.2 命令行界面通常显示一个提示符，用户键入一个命令，然后执行该命令。

13.3 程序。

13.4 对所生的事件（例如用户单击按钮）进行响应的程序。

13.5

a) Label – 标签控件，用于显示文本或图像标签。

b) Entry - 允许用户从键盘输入一行文本的区域。

c) Button - 单击时，可以执行某个行动的按钮。

d) Frame – 可以容纳其他控件的容器。

13.6 创建 tkinter 模块的 Tk 类的一个实例。

13.7 该函数会像无限循环一样运行，直到关闭主窗口为止。

13.8 pack 方法将控件安排在适当的位置，并在显示主窗口时使控件可见。

13.9 两个控件上下挨在一起。

13.10 side='left'

13.11

```
self.label = tkinter.Label(self.main_window,  
                             text='你好，世界',  
                             borderwidth=3,  
                             relief='raised')
```

13.12 self.label1.pack(ipadx=10, ipady=20)

13.13 self.label1.pack(padx=10, pady=20)

13.14 self.label1.pack(padx=10, pady=20, ipadx=10, ipady=10)

13.15 可以使用 Entry 控件的 get 方法来获取用户在控件中键入的数据。

13.16 字符串类型。

13.17 tkinter

13.18 任何存储在 `StringVar` 对象中的值都将自动显示在 `Label` 控件中。

13.19 使用单选钮。

13.20 使用复选框。

13.21 创建一组 `Radiobutton` 对象时，把它们与同一个 `IntVar` 对象关联。然后，为每个 `Radiobutton` 分配唯一的整数值。当选中一个 `Radiobutton` 控件时，它会将其唯一的整数值存储到 `IntVar` 对象中。

13.22 为每个 `Checkbutton` 都关联不同的 `IntVar` 对象。一个 `Checkbutton` 被选中时，它关联的 `IntVar` 对象将保存值 1。取消选中该 `Checkbutton` 时，它关联的 `IntVar` 对象将保存值 0。

13.23

```
self.listbox.insert(0, '一月')
self.listbox.insert(1, '二月')
self.listbox.insert(2, '三月')
```

13.24 默认高度为 10 行，默认宽度为 20 字符。

13.25 `self.listbox = tkinter.Listbox(self.main_window, height=20, width=30)`

13.26 'Peter' 存储在索引 0 处，'Paul' 存储在索引 1 处，'Mary' 存储在索引 2 处。

13.27 返回一个元组，其中包含 `Listbox` 中当前选中项的索引。

13.28 可以调用 `Listbox` 控件的 `delete` 方法，传递想要删除的列表项的索引，从而删除指定的项。

13.29 (0, 0)

13.30 (639, 479)

13.31 在 `Canvas` 控件中，点(0,0)位于窗口的左上角。在海龟图形中，点(0,0)位于窗口的中心。此外，在 `Canvas` 控件中，*Y*坐标在向下移动时增加。在海龟图形中，*Y*坐标在向下移动时减小。

13.32

a) `create_oval`

b) `create_rectangle`

c) `create_rectangle`

d) `create_polygon`

e) `create_oval`

f) `create_arc`

第 14 章

14.1 数据库管理系统 (DBMS) 是一种进行了专门设计, 能以有组织且高效的方式存储、检索和处理大量数据的软件。

14.2 需要存储和处理大量数据时, 传统文件 (如文本文件) 是不实用的。许多企业需要处理数百万个数据项。而如果用传统文件来包含如此多的数据, 简单的操作 (例如搜索、插入和删除) 就会变得繁琐且低效。

14.3 开发者只需要知道如何与 DBMS 交互。DBMS 负责实际的数据读取、写入和搜索。

14.4 一种和数据库管理系统协同工作的标准语言。

14.5 Python 应用程序将 SQL 语句构造为字符串, 然后利用库的一个方法将这些字符串传递给 DBMS。

14.6 `import sqlite3`

14.7

a) 数据库: 数据库容纳了以表、行和列的形式来组织的数据。

b) 表: 表容纳相关数据的集合。在表中存储的数据用行和列来组织。

c) 行: 行容纳关于一个数据项的完整数据集。行中存储的数据被划分为列,

d) 列: 列包含一个数据项中的单个数据。

14.8 1. c 2. d 3. a 4. e 5. b

14.9 主键用于对每一行进行唯一性的标识。

14.10 标识列包含由 DBMS 自动生成的唯一值, 通常用作主键。

14.11 游标是一个对象, 提供了访问和操作数据库中的数据的能力。

14.12 在连接数据库之后。

14.13 对数据库进行更改时, 这些更改实际并未保存到数据库中, 除非最终提交它们。

14.14 调用 SQLite 模块的 `connect` 函数。

14.15 会创建数据库文件。

14.16 使用一个 `Cursor`（游标）类型的对象。

14.17 调用 `Connection` 对象的 `cursor` 方法。

14.18 调用 `Connection` 对象的 `commit` 方法。

14.19 调用 `Connection` 对象的 `close` 方法。

14.20 调用 `Cursor` 对象的 `execute` 方法。

14.21

```
CREATE TABLE Book (Publisher TEXT, Author TEXT,
                    Pages INTEGER, Isbn TEXT)
```

14.22 `DROP TABLE Book`

14.23

```
INSERT INTO Inventory (ItemID, ItemName, Price)
VALUES (10, "Table Saw", 199.99)
```

14.24

```
INSERT INTO Inventory (ItemName, Price)
VALUES ("Chisel", 8.99)
```

14.25

```
cur.execute(''INSERT INTO Inventory (ItemName, Price)
            VALUES (?, ?)'',
            (name_input, price_input))
```

14.26

a) `Account`

b) `Id`

14.27

a) `SELECT * FROM Inventory`

b) `SELECT ProductName FROM Inventory`

c) `SELECT ProductName, QtyOnHand FROM Inventory`

d) `SELECT ProductName FROM Inventory WHERE Cost < 17.0`

e) `SELECT * FROM Inventory WHERE ProductName LIKE "%ZZ"`

14.28 用于判断一个列是否包含指定的字符模式。

14.29 %符号是通配符，代表任意零个或多个字符的序列。

14.30 使用 `ORDER BY` 子句。

14.31 `fetchall` 方法将 `SELECT` 查询的结果作为元组的列表返回。如果查询只返回一个值，可以使用 `fetchone` 方法返回包含一个元素的元组，该元素位于索引 `0` 处。

14.32

```
UPDATE Products
SET RetailPrice = 4.99
WHERE Description LIKE "%Chips"
```

14.33 `DELETE FROM Products WHERE UnitCost > 4.0`

14.34 不可以。

14.35 不可以。

14.36 `INTEGER`

14.37 `100`

14.38 如果在表中创建一个列作为 `INTEGER PRIMARY KEY`，那么该列将成为 `RowID` 列的别名。

14.39 复合键是通过组合两个或更多现有的列来创建的键。

14.40 `sqlite3.Error`

14.41 因为重复的数据会浪费存储空间，并可能导致数据库中出现不一致和冲突的信息。

14.42 外键是一个表中引用了另一个表的主键的列。

14.43 一对多关系意味着一个表中的行可能由另一个表中的多行引用。

14.44 多对一关系意味着一个表中的多行可能引用了另一个表中的一行。

术语表

“开始”终端符号 (start terminal) : 在流程图中标记程序起始点的符号，是一个椭圆。

ASCII : 参见“美国信息交换标准代码”。

cookie : 存储了 Web 浏览会话信息的一种小文件。

CPU : 参见“中央处理单元”。

CRUD : 创建 (Create)、读取 (Read)、更新 (Update) 和删除 (Delete) 的首字母缩写。这是数据库应用程序执行的 4 种基本操作。

CSV : 参见“以逗号分隔的值”

else suite : 作为 try 语句一部分的语句块，相当于 else 子句的主体。该语句块在 try 语句块执行后执行，但前提是 try 语句块没有引发异常。

Entry 组件 : 允许用户输入文本的一种 GUI 控件

finally suite : 作为 try 语句一部分的语句块，出现在 finally 子句之后。该语句块在 try 语句块执行完毕以及任何异常处理程序执行完毕后执行。

f 字符串 (f-string) : 包含特殊代码的一种字符串，可以插入变量值并对其进行格式化。

IPO 图 (IPO chart) : 对函数的输入、处理和输出进行描述的一种图。

Python shell : 以交互模式运行的 Python 解释器。

Python 程序 (Python program) : 包含 Python 语句的文件，也称为“Python 脚本”。

Python 脚本 (Python script) : 参见“Python 程序”。

Python 解释器 (Python interpreter) : 能读取并执行 Python 语句的程序。

RAM : 参见“随机访问存储器”

SQL 注入 (SQL injection) : 一种安全攻击，用户将精心制作的 SQL 代码作为输入，导致该代码被插入（注入）程序的 SQL 代码中，从而对数据库执行恶意操作。

try suite : try 语句中可能引发异常的代码块。

Unicode : 一套全面的字符编码方案，与 ASCII 兼容，可以表示世界上大多数语言的字符。

U 盘 (USB drives) : 便宜、可靠和小巧的一种辅助存储设备，通过 USB 端口与计算机连接。

void 函数 (void function) : 只执行其中包含的语句，不向程序中调用它的位置返回一个值的函数。

with suite : 作为 with 语句的一部分并与资源一起工作的代码块。with suite 中的语句执行完毕后，资源将自动关闭。

按钮 (button) : 用户可以单击以执行某个操作的 GUI 组件。

保留字 (reserved words) : 参见“关键字”。

编译器 (compiler) : 一种特殊的程序，能将高级语言程序转换成一个单独的机器语言程序。

变量 (variable) : 计算机内存中的一个具名的存储位置。

标记 (token) : 从字符串中提取的单独的数据项。

标记化 (tokenizing) : 将字符串分解为多个标记 (token) 的过程。

标签 (label) : 显示了一段文本描述的 GUI 组件。

标识列 (identity column) : 包含数据库管理系统生成的唯一值的列。

标志 (flag) : 一个布尔值或变量，当程序中存在某个条件时发出信号。

标准库 (standard library) : 编程语言事先写好的函数的集合。

不区分大小写的比较 (case-insensitive comparison) : 在比较字符串时，不考虑字符串中字母的大小写。

布尔表达式 (Boolean expression) : 可以求值为真 (True) 或假 (False) 的表达式。

布尔函数 (Boolean function) : 返回 True 或 False 的函数。

步长值 (step value) : 在 for 循环中，计数器变量的增量。

参数化查询 (parameterized query) : 使用占位符在查询中插入变量值的 SQL 查询。

操作符 (operators) : 对数据执行特定操作的符号或单词。

操作数 (operand) : 操作符所操作的值或数据。

操作系统 (operating system) : 计算机最基本的一组程序的集合，是控制计算机硬件内部操作的基本程序，管理连接到计算机的所有设备，允许将数据保存到存储设备或从存储设备中取回数据，并允许其他程序在计算机上运行。

层次结构图 (hierarchy chart) : 显示函数之间关系的一种图，也称为“结构图”。

差集 (difference) : 两个集合之差，包含在一个集合中有，但在另一个集合中没有的元素。

超集 (superset): 包含子集的集合。

超类 (superclass): 继承关系中较一般 (常规) 的类, 也称为 “基类”。

程序 (program): 计算机执行任务时遵循的一组指令

程序开发周期 (program development cycle): 设计、编写、纠正、测试和调试软件的过程。

程序员 (programmer): 经过训练之后, 掌握了设计、创建和测试计算机程序所需技能的人员。也称为 “软件开发人员”。

初始化方法 (initializer method): 创建对象时自动调用的方法。它通常用一些值来初始化对象的各种属性。在 Python 中通常是指 `__init__` 方法。

处理符号 (processing symbol): 流程图的三种符号之一, 该矩形符号代表程序对数据进行某种处理的步骤, 例如数学计算。

传值 (pass by value): 若实参传值, 会将实参的副本传给形参变量。

错误处理程序 (error handler): 对错误进行响应的代码。

错误陷阱 (error trap): 同 “错误处理程序”。

代码 (code): 程序员用一种编程语言写的语句。

代码重用 (code reuse): 执行某个任务的代码只需写一次, 以后每次需要时重复使用。

单分支决策结构 (single alternative decision structure): 只提供了一个分支执行路径的决策结构。

单选钮 (radio button): 可以处于选中或取消选中状态的一种 GUI 组件。多个单选钮通常成组使用, 让用户从中选择一个选项 (而且只能选择一个)。

低级语言 (low-level language): 与机器语言性质接近的一种编程语言。

递归函数 (recursive function): 调用自身的函数。

递归情况 (recursive case): 在递归求解的问题中, 问题必须通过递归来缩小的情况。

递归深度 (depth of recursion): 递归函数调用自身的次数。

调试 (debug): 发现并纠正程序中错误的过程。

调用 (call): 执行一个函数或方法, 并可选择传递特定的参数值。

调用函数 (calling a function): 参见 “调用”。

迭代 (iteration): 循环体的一次执行。

顶点 (vertex): 两条线相连的点。

读取位置 (read position): 标志了要从文件中读取的下一个数据项的位置。

短路求值 (short-circuit evaluation): 一旦可以确定复合布尔表达式的值，就停止对该表达式的求值。

对称差集 (symmetric difference): 两个集合的对称差集包含不会在两个集合中同时出现的元素。

对话框 (dialog box): 一种 GUI 组件，用于显示信息或收集用户输入。

对齐指示符 (alignment designator): 在 f 字符串中表示域内的值如何对齐的一个符号。

对象 (object): 同时包含了数据和过程的一种软件实体。

对象可重用性 (object reusability): 出于多种目的而重复使用对象的一种能力。

多对一关系 (many-to-one relationship): 一个表中的多行可以引用另一个表中的一行。

多态性 (polymorphism): 对象具有不同形式的功能。

多行字符串 (multiline string): 在程序代码中跨越多行的字符串。

多重赋值语句 (multiple assignment statement): 为多个变量赋值的一个赋值语句。

二的补码 (two's complement): 一种将负数存储为二进制值的技术。

二进制数字系统 (binary numbering system): 一种数字系统，其中所有数值都写成 0 和 1 的序列。

二进制位 (binary digit): 二进制位（也称为 bit）是内存存储的基本单位，可以处于“开”或“关”状态，只能为值 0 或 1。

二进制文件 (binary file): 包含未转换为文本的数据的文件。通常是特殊的程序数据，在文本编辑器中会显示成“乱码”。

二维列表 (two-dimensional list): 包含其他列表作为其元素的一种列表。

返回值的函数 (value-returning function): 将值返回给程序中调用它的那一部分的函数。

方法 (method): 从属于一个对象并在该对象上执行操作的函数。注意，本书没有刻意区分方法和函数。

分而治之 (divide and conquer): 将一个大任务分解为几个容易执行的小任务的过程。

分隔符 (delimiter): 对字符串中的标记 (token) 进行分隔的字符。

封装 (encapsulation): 将数据和代码合并到一个对象中。

浮点表示法 (floating-point notation): 一种将实数存储为二进制值的技术。

辅助存储 (secondary storage): 一种即使在计算机断电的情况下也能长时间保存数据的存储设备。也称为“二级存储”。

复合赋值操作符 (augmented assignment operator): 与数学运算符相结合的一些赋值操作符。例如，+= 将先执行加法运算，再执行赋值操作。

复合键 (composite key): 由两列或多列组合而成的键。

复选框 (check box): 以小方框形式显示的 GUI 组件，可以选中或取消选中。

赋值表达式 (assignment expression): 为变量赋值并返回所赋值的表达式。

赋值操作符 (assignment operator): 赋值操作符 (=) 将其右侧的值赋给左侧的变量。

赋值方法 (mutator method): 在属性中存储值或以其他方式更改属性值的方法。

赋值语句 (assignment statement): 向变量赋值的语句。和其他语言不同，Python 允许用赋值语句来创建一个之前不存在的变量。

高级语言 (high-level languages): 一种编程语言，使用了比低级语言指令更容易理解的单词，允许在不知道 CPU 具体如何工作的情况下创建程序。

格式说明符 (format specifier): f 字符串中的特殊代码，指定以特定方式格式化数值。

根据位置传递 (passed by position): 第一个实参传给第一个形参，第二个实参传给第二个形参，以此类推。

跟踪 (traceback): 指出程序中发生错误的位置的消息。也称为“回溯”。

公共方法 (public method): 类的一种方法，可由类外部的代码访问。

固态硬盘 (solid-state drive): 不像传统硬盘那样有机械部件的新一代辅助存储设备，速度非常快。

关键字 (keyword): 在编程语言中具有特定含义的词，不能用于任何其他目的。也称为“保留字”。

关键字参数 (keyword argument): 一种函数调用语法，通过显式提供参数名称来指定要将实参传递给哪个形参。也称为“关键字实参”。

关系操作符 (relational operator): 比较两个值并确定它们之间是否存在特定关系的操作符。

过程 (procedure): 执行特定任务的函数。

海龟图形 (turtle graphics): 模拟机器“海龟”的一种 Python 图形系统，可在移动过程中绘制线和形状。

函数 (function): 程序中为重复执行特定任务而编写的一组语句。

函数定义 (function definition): 对函数进行定义的代码。

函数头 (function header): 函数定义的第一行，包含函数名和参数等。

黑盒 (black box): 任何接受输入、执行某些操作（在外部不清楚细节）并产生输出的机制。

画布 (Canvas): 一种显示为空白矩形区域的 GUI 组件。可以在画布上绘制各种二维图形。

缓冲区 (buffer): 内存中的一个小的“保留区”。

幻数 (magic number): 程序代码中出现的含义不明的数字。

换行符 (newline character): 标志一行文本结束的不可见字符。

换行符转义序列 (newline escape character): 标记文件中新行开始的一个转义序列，即\n。

回调函数 (callback function): 发生特定事件时自动执行的函数。

汇编器 (assembler): 一种特殊的程序，用于将汇编语言程序转换为机器语言程序。

汇编语言 (assembly language): 一种低级编程语言，使用称为助记符的短字来代替二进制指令。

混合类型的表达式 (mixed-type expression): 使用了不同数据类型的操作数的表达式。

机器语言 (machine language): 计算机 CPU 可以理解并执行的指令。

基本情况 (base case): 在递归求解的问题中，无需继续递归即可求解的情况。

基类 (base case): 参见“超类”。

集成开发环境 (Integrated Development Environment , IDE): 集成了文本编辑器、编译器、解释器、调试器等编程工具的一种专用软件，例如 Visual Studio。在 Python 中特指它自带的 IDLE 程序。

集合 (set): 存储唯一的、无序的元素集合的对象。

集合推导式 (set comprehension): 读取一个集合的元素并创建另一个集合的简化表达式。

计数控制循环 (count-controlled loop): 重复特定次数的循环。

计数器变量 (counter variable): 对循环迭代次数进行计数的变量。

记录 (record): 对一个数据项进行描述的完整数据集。

键 (key): 一个字典元素中对该元素进行标识的那一部分，另一部分是“值”。

键值对 (key-value pairs): 由键和值两部分组成的数据元素。我们通过查找数据元素的键来检索相应的值。

交互模式 (interactive mode): 用键盘输入一个 Python 语句，就由 Python 解释器执行一个语句的模式。

交集 (intersection): 两个集合的交集只包含两个集合中都有的元素。

脚本模式 (script mode): Python 解释器读取并执行包含 Python 语句的文件内容的一种模式。

结构化查询语言 (Structured Query Language , SQL): 数据库管理系统使用的一种标准语言。

结构图 (structure chart): 显示函数之间关系的一种图。

结束终端 (End terminal): 结束终端

截断 (truncate): 截去数据的一部分，例如数字的小数部分。

解释器 (interpreter): 转换（翻译）并执行高级语言程序指令的程序。

仅关键字参数 (keyword-only parameter): 只接受关键字参数的参数。

仅位置参数 (positional-only parameter): 只接受位置实参（根据位置传递的参数）。

精度指示符 (precision designator): f 字符串中的特殊代码，表示浮点数四舍五入后的小数位。数。

局部变量 (local variable): 在函数内部创建的变量。当前函数外部的语句不能访问该变量。

具名常量 (named constant): 代表特殊值的一个名称，这种值在程序执行期间不会发生更改。

聚合函数 (aggregate function): 对数据库表中的一组值执行特定计算的 SQL 函数。

决策结构 (decision structure): 一种控制结构，只有符合特定条件才执行一组语句。

开销 (overhead): 为函数调用所做的准备操作，例如为参数和局部变量分配内存，以及存储返回地址。

可变 (mutable): 特指一样东西可以修改，而非固定不变（不可变）。

可迭代对象 (iterable): 包含一系列值的对象，可通过循环来遍历这些值。

客户 (customer): 要求你编写程序的个人、团体或组织。

控件 (widget): 在 GUI 程序中显示的图形元素，用户可与之交互或查看其内容。

控制结构 (control structure): 控制一组语句执行顺序的逻辑设计。

库函数 (library functions): 编程语言内置或由模块导入的一系列函数的集合。

块 (block): 参见“语句块”。

框架 (frame): 可以容纳其他 GUI 组件的一种容器。

垃圾回收 (garbage collection): 自动删除内存中不再被变量引用的数据的过程。

类 (class): 为特定类型的对象规定了属性和方法的代码，相当于这些对象的“蓝图”。

类的职责 (responsibilities): 类需要了解的事情和类需要执行的行动。

类定义 (class definition): 定义了类的方法和数据属性的一组语句。

类型指示符 (type designator): f 字符串中的特殊代码，用于指示所显示的值的类型。

累加和 (running total): 累加器变量中不断增加的数字之和。

累加器 (accumulator): 在循环迭代过程中累加一系列值的变量。

连接 (concatenation): 将一个字符串附加（连接）到另一个字符串的末尾，也称为“拼接”。

列 (column): 一个数据项（数据表中的一行或者说一条记录）中包含的单个数据。

列表 (list): 可以包含多个数据项的一种对象。

列表框 (Listbox): 显示数据列表并允许用户从列表中选择一项或多项的 GUI 组件。

列表推导式 (list comprehension): 根据一个列表的内容生成另一个列表的表达式。

流程图 (flow chart): 以图形方式描述程序运行步骤的一种图。

逻辑操作符 (logical operator) : 连接两个布尔表达式以创建复合布尔表达式的操作符。

逻辑错误 (logic error) : 程序中的一种错误, 虽然不会造成程序意外中止, 但会导致错误的结果。

美国信息交换标准代码 (American Standard Code for Information Interchange , ASCII) : 一组 128 个数值编码, 用于表示英文字母、各种标点符号和其他字符。

面向对象编程 (object-oriented programming) : 以创建对象为中心的一种编程范式, 对象是同时包含数据和过程的软件实体。

命令行界面 (command line interface) : 显示提示, 并执行用户键入的命令的一种界面。

模块 (module) : 可以导入其他 Python 程序的一种 Python 源代码文件

模块化 (modularization) : 在编写程序时, 每项任务都由单独的函数执行。

模块化程序 (modularized program) : 每个任务都由单独的函数来执行的程序。

目标变量 (target variable) : 在 for 循环的每次迭代中作为赋值目标的变量。

内部填充 (internal padding) : 围绕 GUI 组件内边缘的空白空间。

内存条 (memory sticks) : 计算机的 RAM。

派生类 (derived class) : 参见“子类”。

屏幕坐标系 (screen coordinate system) : 用于指定屏幕上的特定像素位置的坐标系。

嵌套列表 (nested list) : 列表元素是其他列表的一种列表。

切片 (slice) : 序列中元素的一个 span。也称为“分片”。

求余操作符 (modulus operator) : 返回除法运算的余数的操作符, 也称为取模操作符。

取值方法 (accessor method) : 从类的属性中获取值但不更改该值的方法。

取指-解码-执行 (fetch-decode-execute cycle) : 中央处理器执行程序中的每条指令的三步过程。

全局变量 (global variable) : 在程序中所有函数的外部定义的变量。全局变量的作用域是整个程序。全局变量的特点是可变。

全局常量 (global constant) : 程序中每个函数都可以使用的具名常量, 特点是不可变。

软件 (software) : 计算机程序。

软件开发工具 (software development tools): 程序员用来创建、修改和测试软件的程序。

软件开发人员 (software developer): 参见“程序员”。

软件需求 (software requirement): 为满足客户需求，程序必须执行的功能。

闪存 (flash drives): U 盘等存储设备使用的一种特殊存储器。

哨兵 (sentinel): 标志值序列结尾一个特殊值。

实参 (argument): 调用函数时向其实际传递的数据。

实例 (instance): 基于类来创建的对象。

实例属性 (instance attribute): 一个类的特定实例的属性。

实用程序 (utility program): 执行专门任务的一种程序，用于增强计算机的操作或者保护数据。

事件处理程序 (event handler): 在发生特定事件时执行的方法/函数。

事件对象 (event object): 包含事件信息的对象。

输出 (output): 计算机为人或其他设备/程序生成的数据。

输出符号 (output symbol): 流程图的三种符号之一，用平行四边形代表程序生成输出的步骤。也作为输入符号使用。

输出设备 (output device): 格式化和呈现输出的设备。

输出文件 (output file): 程序向其中写入数据的一种文件。

输入 (input): 计算机从人和其他设备处收集的任何数据。

输入符号 (input symbol): 流程图的三种符号之一，用平行四边形代表程序获取输入的步骤。也作为输出符号使用。

输入设备 (input device): 收集输入并将其发送给计算机的一种组件，例如键盘和鼠标。

输入文件 (input file): 程序从中数据的一种文件。

输入校验 (input validation): 确认输入有效性的过程。

数据库 (database): 由表、行和列组成并由数据库管理系统维护的数据集合。

数据库管理系统 (Database Management System , DBMS): 专门用于有组织地、高效地存储、检索和处理大量数据的软件。

数据类型 (data type): 变量将引用的数据类型。

数据属性 (data attribute): 存储在对象中的变量。

数据隐藏 (data hiding): 一个对象对其外部代码隐藏数据的能力。

数学表达式 (math expression): 执行数学计算并给出结果的代码。

数学操作符 (math operator): 对操作数执行运算的符号，也称为“数学运算符”。

数值字面值 (numeric literal): 直接在程序代码中写入的数字。

数字 (digital): 任何以二进制数字表示的数据。

数字设备 (digital device): 处理二进制数据的设备。

数字数据 (digital data): 以二进制格式存储的数据。

双分支判断结构 (dual alternative decision structure): 一种判断结构，有两种可能的执行路径，如果条件为真，则执行其中一个路径。如果条件为假，则执行另一个。

顺序访问文件 (sequential access file): 包含了数据的一种文件，必须从文件开始到文件结束按顺序访问。

私有方法 (private method): 只有类中的代码才能访问的类方法。

私有数据属性 (private data attribute): 只有类中的代码才能访问的类属性。

算法 (algorithm): 执行任务时必须采取的一组明确定义的逻辑步骤。

随机访问存储器 (Random-Access Memory , RAM): 一种允许从任意位置快速检索数据的内存。

随机访问文件 (random access file): 允许程序直接跳转到文件中特定位置的文件。

索引 (index): 指定元素在列表中位置的一个整数。注意，Python 没有专门的“数组”概念。

提示 (prompt): 告诉（或要求）用户输入特定值的一条消息。

填充 (padding): 出现在组件周围的空白空间。

条件表达式 (conditional expression): 对 if-else 语句进行简化一种语法结构。

条件控制循环 (condition-controlled loop): 使用真/假条件控制重复次数的循环。

条件执行 (conditionally executed): 仅在特定条件为真时才执行一个或一组语句。

通配符 (wildcard character): 用于替换字符串中一个或多个字符的特殊字符。

图像元素 (picture element): 参见“像素”。

图形用户界面 (Graphical User Interface , GUI): 允许用户通过屏幕上的图形元素与操作系统和程序进行交互的界面。

退出按钮 (Quit button): 在 GUI 中设计的单击时关闭程序的一种按钮。

外部填充 (external padding): 在 GUI 组件外边缘留出的空白空间。

外键 (foreign key): 一个表中引用了另一个表的主键的列。

微处理器 (microprocessor): 一般是指计算机的 CPU 芯片。

伪代码 (pseudocode): 一种非正式语言，没有语法规则，不能直接编译或执行。

伪随机数 (pseudorandom number): 看似随机但实际上可以用同一个种子值来复现的一组“随机”数。

位 (bit): 参见“二进制位”。

文本文件 (text file): 文本文件包含使用 ASCII 或 Unicode 等方案编码为文本的数据。

文件对象 (file object): 内存中与特定文件关联的一个对象。程序使用文件对象来处理文件。

文件扩展名 (filename extensions): 文件名末尾最后一个句点后的短字符序列，说明了文件的类型。

问题域 (problem domain): 现实世界中与当前问题相关的对象、参与方以及主要事件的集合。

无限循环 (infinite loop): 无法停止的循环。

系统软件 (system software): 控制和管理计算机基本操作的程序。

限定列名 (qualified column name): 同时标识了列以及列所属的表，格式是“表名.列表”。

像素 (pixel): “像素元素”的简称，代有图像中的一个彩色小点，是计算机图形的基本组成元素。

消息框 (message box): 一种 GUI 组件，显示提示消息，并提供一个“确定”按钮供关闭对话框。

行 (row): 存储在数据库表中的关于单个数据项的整套数据。

行末注释 (end-line comment): 出现在一行代码末尾的注释。

形参变量 (parameter variable): 在函数头中声明并接收传给函数的实参的一种特殊变量。本书若非必要，一般不区分形参和实参。所以说成“参数变量”也可以。

形参列表 (parameter list): 函数头中出现的两个或多个形参变量声明。本书若非必要，一般不区分形参和实参。所以说成“参数列表”也可以。

序列 (sequence): 一种容纳多个数据项的对象，这些数据项一个接一个地存储。

序列化 (serialize): 将对象转换为可保存到文件中的字节流。在 Python 中称为“腌制”。

序列结构 (sequence structure): 一组按出现顺序执行的语句。

选择结构 (selection structure): 一种控制结构，只有符合特定条件才执行一组语句。

循环 (loop): 根据需要多次重复一个或一组语句的控制结构。

循环体 (body of loop): 由一个循环重复执行的语句。

腌制 (pickle): 将对象序列化为可以保存到文件中的字节流。“腌制”是 Python 特有的说法，其实就是序列化的意思。反序列化在 Python 称为 unpickle。

样本 (sample): 一小段数字音频数据。

一对多关系 (one-to-many relationship): 表中的一行可能由另一个表中的多行引用。

伊尼亚克计算机 (ENIAC): 世界上第一台可编程电子计算机。

以逗号分隔的值 (Comma Separated Values , CSV): 各个数据项以逗号分隔的一种文件格式。

异常 (exception): 程序运行过程中出现的必须加以处理的错误。

异常处理程序 (exception handler): 响应异常并防止程序意外中止的代码。

异常对象 (exception object): 异常发生时在内存中创建的包含异常信息的对象。

易失性存储器 (volatile memory): 程序运行时用于临时存储代码和数据的一种存储器。

应用软件 (application software): 使计算机在日常工作中发挥作用的程序。

映射 (mappings): “键值对”的另一个称呼。

硬件 (hardware): 计算机的物理设备。

硬盘驱动器 (hard disk drive): 一种通过磁性编码将数据存储于圆形碟片上的存储设备。

用户 (user) : 使用程序并为其提供输入并接收输出的人。

用户界面 (User Interface , UI) : 用户与系统进行交互的那一部分。

优先级 (precedence) : 当表达式中包含多个操作符时，根据操作符的优先级来分先后应用它们。

由菜单驱动的程序 (menu-driven program) : 显示选项清单供用户选择的一种程序。

由事件驱动的程序 (event-driven program) : 一种等待事件发生并对事件作出响应的程序。

语法 (syntax) : 编写程序时必须严格遵守的一系列规则。

语法错误 (syntax error) : 程序员所犯的一种错误，例如关键字拼写错误、标点符号缺失或者操作符使用不当。

语句 (statement) : 由关键字、操作符、标点符号和其他允许的编程元素构成的代码单元，以适当的顺序排列以执行特定的操作。

语句块 (statements block) : 多个相关语句的一个分组，简称为“块”。

预测试循环 (pretest loop) : 在每次迭代之前都先测试其条件表达式的循环。

预读 (priming read) : 获取将由校验循环测试的第一个输入值。如果该值无效，那么将由循环接手来执行后续的输入操作。除非提供有效的输入，否则循环不会终止。

域宽说明符 (field width designator) : f字符串中说明字段最小宽度的一个指示符。

元素 (element) : 列表或元组中的单个数据项。

元组 (tuple) : 和列表非常相似的一种序列。元组和列表的区别在于元组不可变。

原始字符串 (raw string) : 一种特殊格式的字符串，其中的反斜杠字符会作为字面意义的反斜杠字符读取，而不会解释成转义序列。

源代码 (source code) : 程序员用编程语言编写的语句。

约束 (constraint) : 数据库管理系统强制执行的一种规则，用于限制列中可以存储的数据类型。

运行 (running) : 计算机执行程序指令的过程。

执行 (executing) : 计算机根据提供给它的指令来执行任务的过程。

直接递归 (direct recursion) : 递归函数直接调用自身的情况。

直接访问文件 (direct access file) : 允许程序直接跳转到文件中特定位置的文件。

值 (value): 字典元素中保存数据并与键关联的那一部分。我们通过键来检索值。

指令集 (instruction set): CPU 能执行的一整套指令的集合。

中央处理单元 (central processing unit): 计算机中实际执行程序的硬件设备，相当于计算机的“大脑”。

终端符号 (terminal symbols): 指定流程图开始和结束的椭圆形符号。

种子值 (seed value): 一个特殊的值，用于初始化计算伪随机数的公式。

重复操作符 (repetition operator): 重复操作符生成一个数据序列的多个副本，并将它们连接到一起。在 Python 中，它既可用于字符串，也可用于列表。

重复结构 (repetition structure): 一种控制结构，可根据需要多次重复一个或一组语句。

重写 (override): 子类方法与超类方法同名时，我们说子类方法“重写”或“覆盖”了超类方法。

主存 (main memory): 计算机内存，存储了正在运行的程序以及程序要处理的数据。

主调函数 (calling function): 一个函数内部调用了另一个函数时，前者称为主调函数。

主键 (primary key): 对数据库表中的每一行进行唯一标识的列。

主线逻辑 (mainline logic): 程序的整体逻辑。

助记符 (mnemonics): 汇编语言中表示机器语言指令的一些短字，例如 add。

注释 (comment): 程序代码中的文本解释，供阅读代码的人使用。

转义序列 (escape sequences): 出现在字符串面值中的以反斜杠(\)为前缀的特殊字符。转义序列被视为嵌入字符串中的特殊控制命令。

状态 (state): 对象属性在任何特定时刻的内容。

追加模式 (append mode): 一种文件打开模式，写入文件的所有数据将追加到文件当前内容的末尾。

资源 (resource): 程序使用的外部对象或数据源。

资源泄漏 (resource leak): 当程序未能关闭所有打开的资源时可能发生的情况。

子串 (substring): 从较大字符串中取出的字符串，即较大字符串的一个“切片”。

子集 (subset): 也属于另一个集合的一组值。

子类 (subclass): 继承关系中较具体 (特化) 的类, 也称为 “ 派生类 ”

自动递增 (autoincremented): 由数据库管理系统自动生成并存储在标识列中的整数值。该值比当前存储在标识列中的最大值大 1。

自上而下设计 (top-down design): 将算法分解成多个函数的设计过程。

字典 (dictionary): 存储键值对集合的对象。

字典推导式 (dictionary comprehension): 读取输入元素的序列, 并依据特定条件来生成一个字典的表达式。

字段 (field): 一条记录中的单个数据。

字符串 (string): 字符序列。

字符串字面值 (string literal): 直接出现在程序代码中的字符串, 要用引号括起来。

字节 (bytes): 内存单位, 其大小足以容纳一个字母或者小的数字。

组件 (component): 作为计算机系统一部分的物理设备。

最终用户 (end user): 也称为 “ 终端用户 ”。参见 “ 用户 ”。

作用域 (scope): 程序中可以访问某个变量或常量的那一部分称为该变量或常量的 “ 作用域 ”。