第3章

Python 编程基础

本章先向读者介绍 Python 语言的发展史和安装方法，然后介绍 Python 语言的基本语法、结构控制语句和常用数据结构，最后介绍函数定义、类的使用和模块调用的方法。

3.1 引　　言

Python 的第一个版本发行于 1991 年，经过 30 多年的发展，它已经渗透到计算机科学与技术、统计分析、逆向工程与软件分析、电子取证、图形图像处理、人工智能等专业和领域。目前，Python 已成为国内外很多大学的计算机专业或非计算机专业的程序设计入门教学语言。自 2020 年以来，Python 一直稳居 TIOBE 编程语言排行榜前三位。Python 语言发展势头如此迅猛，原因在于 Python 是一门免费、开源的跨平台高级动态编程语言，支持命令式编程、函数式编程，完全支持面向对象程序设计，语法简洁清晰，并且拥有大量功能强大的标准库和扩展库，以及众多狂热的支持者，可以帮助各领域的科研人员、策划师，甚至管理人员快速实现和验证自己的思路与创意。Python 用户可以把主要精力放在业务逻辑的设计与实现上，而不用过多考虑语言本身的细节，开发效率非常高，其精妙之处令人击节叹赏。Python 编程模式非常符合人类的思维方式和习惯。与 C 语言系列和 Java 语言等相比，Python 更加容易入门，对于没有任何编程经验的初学者来说，简洁而强大的 Python 就是最理想的选择。

3.2 Python 安装与运行

Python 目前有两个版本，Python 2 和 Python 3。其中 Python 2.7 为 Python 2 的最后一个版本。Python 3 与 Python 2 不完全兼容。Python 3.x 的设计理念更加合理、高效和人性化，全面普及和应用是必然的。越来越多的扩展库也以非常快的速度推出了与最新 Python 版本相适应的版本。对于初学者来说，建议大家选择 Python 3 系列的最新版本。

选定 Python 版本后，即可安装 Python 开发环境。Python 的开发环境非常丰富，可以根据自己的使用习惯进行选择。除了 Python 官方网站提供的 IDLE 开发环境，还有如 PyCharm、wingIDE、PythonWin、Eclipse＋PyDev、Eric 等。另外，为了方便使用 Python，Anaconda、Python(x,y)、zwPython 等安装包集成了大量常

用的 Python 扩展库，大幅节约了用户配置 Python 开发环境的时间。以原生 Python 为例，有两种基本模式：脚本模式和交互模式。其中交互模式有利于快速方便地运行单行代码或代码块，因为它总是能立即给出结果。图 3-1 为 Python 原生 IDLE 启动后的界面，启动后默认进入交互模式。交互模式下用户需在“>>>”符号后面输入代码，当一段代码编写完毕之后按 Enter 键即时获取该段代码的运行结果。如果用户想要在完成整体的代码编写工作之后再进行程序的运行，可以选择脚本模式，如图 3-2 所示。脚本模式将 Python 程序代码以文件形式(.py 为扩展名)保存，保存之后使用命令行"Python 脚本文件名.py"来运行该代码。

图 3-1 Python IDLE 交互模式界面

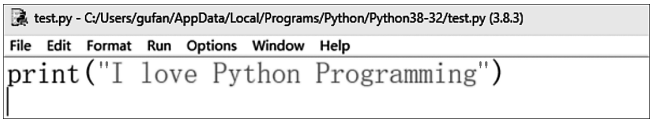

图 3-2 Python IDLE 脚本模式界面

以下例子是利用 Python 实现一个猜数字游戏的代码。游戏规则如下：系统随机产生一个 1～10 的整数，玩家可以进行最多 5 次猜测；每次猜测完成后，系统会返回正确、偏大和偏小三种结果之一，玩家可以根据系统的提示，对下一次的猜测进行适当调整；若结果正确，则游戏结束，否则进入下一轮猜测。具体代码及相关注释(注释指向阅读代码的人而非计算机解释代码含义的注解，在 Python 中以字符“#”开头到行末的文字就是注释)如下。

```
from random import randint
def guess(start=1, end=10, maxTimes=5): #变量 maxTimes 指定最大猜测次数,默认值为 5
    #随机生成一个 start 到 end 范围内的整数
    value = randint(start, end)
    for i in range(maxTimes):
        prompt = '开始猜:' if i==0 else '再猜一次:'
        #使用异常处理结构 try-except,防止输入不是数字的情况
        try:
            x = int(input(prompt))
            assert start<=x<=end
        except:
            print('不要浪费机会啊,必须输入{0}到{1}之间的数字'.format(start, end))
        else:
            #猜对了
```

```
            if x == value:
                print('太牛了,居然猜对了! ')
                break
            elif x > value:
                print('真遗憾,猜的太大啦。')
            else:
                print('有点小哦,加油。')
        else:
            #次数用完还没猜对,游戏结束,提示正确答案
            print('失败,游戏结束!!! ')
            print('本次游戏正确的数是: ', value)

if __name__ == '__main__':
    guess(10, 100) #在主函数中调用 guess()函数开始游戏
```

以上猜数游戏代码涉及的知识点包括 Python 模块的引用、for 循环语句、分支 if...else 结构、异常处理结构 try...except 等,后续将对这些知识点进行详细介绍。

近些年,随着人工智能技术的迅猛发展,出现了一系列自动或半自动的编程工具。例如,OpenAI 公司开发的 ChatGPT 模型及由微软、OpenAI、GitHub 三家联合打造的 AI 编程辅助工具 Copilot。只要用自然语言向 ChatGPT 描述想要实现的功能,它就能写出所有的代码。Copilot 是一款基于人工智能的代码辅助工具,能够根据用户输入的上下文和语言,提供代码片段和建议。使用 Copilot 可以大幅提高编程效率。

以下是利用清华大学开发的 ChatGLM2-6B,根据用户输入生成的一段代码。

用户输入:“帮我用 Python 写一个冒泡排序的代码。”

以下是 ChatGLM2-6B 模型输出的文本。

当然可以,以下是一个使用 Python 实现的冒泡排序算法。

```
```python
def bubble_sort(arr):
    n = len(arr)
    #遍历所有数组元素
    for i in range(n):
        #最后 i 个元素已经排好序,不需要再比较
        for j in range(n - i - 1):
            #如果当前元素比下一个元素大,则交换它们的位置
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```
```

此刻我们还没有介绍 Python 的具体语法,仅对上述 ChatGLM2-6B 生成代码的大致意思进行粗略介绍。在该代码中,我们首先获取数组的长度 n,然后使用两层循环来遍历所有的数组元素。在内部循环中,我们跳过已经排好序的 i 个元素,这样可以确保我们只在未排序的元素之间进行比较和交换。

对于每一行中的元素,我们比较当前元素和下一个元素的大小,如果当前元素比下一个元素大,则交换它们的位置。这样就可以让最大的元素逐渐“冒泡”到数组的末尾,从而实现冒泡排序。

最后，我们将排好序的数组元素返回给调用者。

下面是一段使用 ChatGLM2-6B 生成的代码进行冒泡排序操作的示例（同样由 ChatGLM2-6B 工具生成）。

```
```python
my_list = [5, 2, 9, 1, 5, 6]        #待排序数值列表
sorted_list = bubble_sort(my_list)
print(sorted_list)                  #打印排序结果为[1, 2, 5, 5, 6, 9]
```
```

在这个示例中，我们使用一个名为 my_list 的列表，其中包含一些数字。然后我们使用 bubble_sort()函数来将这些数字排序并返回一个排好序的新列表。我们将排好序的新列表打印出来，这样我们就可以看到原始列表是如何被排序的。

可以看到，该工具自动生成的冒泡排序函数代码可以正常使用。同时，为了帮助使用者理解，生成代码中还包含了对冒泡排序原理的阐述、代码的注释和解释，以及调用示例。其中代码部分以“```python”开始，以“```”结束。

3.3 Python 基本元素

对象是 Python 中最基本的概念之一，在 Python 中一切都是对象，除了整数、实数、复数、字符串、列表、元组、字典、集合外，还有 zip、map、enumerate、filter 等对象，函数和类也是对象。每个对象都有一个类型，它规定了程序可以对该类型对象进行哪些操作。类型分为标量和非标量。标量对象为不可分割的单个元素，非标量对象如字符串，通常包含多个可分解的元素。

以下例子利用 Python 实现求解一元二次方程的根。分两种情况处理，delta＞＝0 和 delta＜0。当 delta＜0 时，输出两个共轭的复数根。代码中包含了部分算数运算符的使用和函数的调用。

```
import math
a = float(input("请输入 a 的值: "))          #用户输入方程系数 a
b = float(input("请输入 b 的值: "))          #用户输入方程系数 b
c = float(input("请输入 c 的值: "))          #用户输入方程系数 c
delta = b**2-4*a*c                          #利用参数 a、b、c 计算 delta
if delta>=0:
    root = math.sqrt(delta)                 #调用求根函数
    x1 = (-b+root)/(2*a)
    x2 = (-b-root)/(2*a)
else:
    root = math.sqrt(-delta)
    x1 = complex(-b/(2*a),root/(2*a))       #complex 表示复数，2 个参数分别为实部
                                            #和虚部
    x2 = complex(-b/(2*a),-root/(2*a))
print("x1=",x1,"\t","x2=",x2)               #打印结果
```

3.3.1　数值类型

在 Python 中，内置的数值类型有整型、浮点型、布尔型、复数。

(1) 整型(int)：用来表示整数，如 100、－10 或 27 等。

(2) 浮点型(float)：即小数，如 3.7、－19.19 或 20.0。也可以用科学记数法表示，如 3.9e3 即 3.9×10^3，等同于 3900.0。

(3) 布尔型(bool)：用来表示布尔值，即“真”或“假”，在 Python 中分别用 True 和 False 来表示。

(4) 复数(complex)：用来表示复数，如 3＋4j，3 为实部，4 为虚部。

Python 支持任意大的数字，具体可以大到什么程度受内存大小的限制。由于精度的问题，对于浮点数运算可能会有一定的误差，应尽量避免在浮点数之间直接进行相等性测试，而是应该以两者之差的绝对值是否足够小作为两个浮点数是否相等的依据。在数字的算术运算表达式求值时会进行隐式的类型转换，如果存在复数则都变成复数，如果没有复数但是有浮点数就都变成浮点数，如果都是整数则不进行类型转换。

3.3.2　运算符与表达式

对象和运算符可以构成表达式，表达式运算后会得到一个值，称为表达式的值。运算符包括算术运算符和逻辑运算符，具体说明如下。

(1) 加法 a＋b：如果 a 和 b 都是 int 型，结果为 int 型；只要其中有一个类型为 float 型，结果即为 float 型。

(2) 减法 a－b：同加法类似，如果 a 和 b 都是 int 型，结果为 int 型；只要其中有一个类型为 float 型，结果即为 float 型。

(3) 乘法 a * b：表示 a 与 b 的乘积，类型转换规则同加减法。

(4) 取整除，a//b：如 8//2 的值为 4，2.1//2 值为 1.0，即取整除只取整数商，去掉小数部分，类型转换规则同加减法。

(5) 除法 a/b：结果为 float 型，如 10/4 结果为 2.5，10/2 结果为 5.0。

(6) 取模 a％b：表示 a 除以 b 的余数，即数学的模运算。如 10％2 结果为 0，11％2.0 结果为 1.0。

(7) 幂 a**b：表示 a 的 b 次方。

(8) 比较运算符：包含小于＜、大于＞、小于或等于＜＝、大于或等于＞＝、等于＝＝、不等于!＝，结果为布尔型。

(9) 逻辑运算符 and、or、not：分别表示与、或、非三种逻辑运算，在功能上可以与电路的连接方式做个简单类比：or 运算符类似于并联电路，只要有一个开关是通的那么灯就是亮的；and 运算符类似于串联电路，必须所有开关都是通的灯才会亮；not 运算符类似于短路电路，如果开关通了那么灯就灭了。就“表达式 1 and 表达式 2”而言，如果“表达式 1”的值为 False 或其他等价值时，不论“表达式 2”的值是什么，整个表达式的值都是 False，丝毫不受“表达式 2”的影响，因此“表达式 2”不会被计算。在设计包含多个条件的条件表达式时，如果能够大概预测不同条件失败的概率，并将多个条件根据 and 和 or 运算符的短路求值特性来组织顺序，可以提高程序运行效率。

3.3.3 常量与变量

所谓常量，一般是指不需要改变也不能改变的字面值，如一个数字 3，又如一个列表[1，2，3]，都是常量。与常量相反，变量的值是可以变化的，这一点在 Python 中更是体现得淋漓尽致。在 Python 中，不需要事先声明变量名及其类型，直接赋值即可创建任意类型的对象变量。不仅变量的值是可以变化的，变量的类型也是随时可以发生改变的。例如，下面第一条语句创建了整型变量 x，并赋值为 3。

```
>>> x=3
>>> type(x)              #内置函数 type()用来查看变量类型，返回结果为 <class 'int'>
>>> x='hello python'     #该语句将字符串'hello python'赋值给变量 x，x 的类型变为字符串
```

注意：此处（以及本书后续内容）每行代码开头的“>>>”并不属于代码，而是 Python 原生编辑器 IDLE 交互模式界面的一部分，如图 3-1 所示。

Python 采用基于值的内存管理模式。赋值语句的执行过程是：首先把等号右侧表达式的值计算出来，然后在内存中寻找一个位置把值存放进去，最后创建变量并指向这个内存地址。Python 中的变量并不直接存储值，而是存储值的内存地址或引用，这也是变量类型随时可以改变的原因。

```
>>> a='Hello Python'
>>> b=a
>>> a=123
>>> print(b)
```

变量 a 一开始指向字符串'Hello Python'，b＝a 创建了变量 b，变量 b 也指向了 a 指向的字符串'Hello Python'，最后 a＝123，使变量 a 重新指向了整数 123，所以最后输出的变量 b 是'Hello Python'。

此外，Python 还允许多重赋值。

```
>>> a,b,c=1,2,3        #将 1,2,3 分别赋值给 a,b,c 三个变量
```

在 Python 中定义变量名时，需要注意以下问题：变量名必须以字母或下画线开头，但以下画线开头的变量在 Python 中有特殊含义；变量名中不能有空格或标点符号；变量命名要避开 Python 关键字，不建议使用内置模块名或函数名命名；变量大小写敏感。

3.3.4 字符串与输入输出

在 Python 中，只有字符串类型的常量和变量，单个字符也是字符串，通常使用单引号、双引号、三引号来定义字符串，其中三引号可以将复杂的字符串进行赋值。Python 中三引号允许一个字符串跨多行，字符串中可以包含换行符、制表符及其他特殊字符。Python 3.x 全面支持中文，中文和英文字母都作为一个字符对待，甚至可以使用中文作为变量名。除了支持使用加号运算符连接字符串以外，Python 字符串还提供了大量的方法支持查找、替换、排版等操作。很多内置函数和标准库对象也都支持对字符串的操作，这里仅简单介绍一下字符串对象的创建和连接。

```
>>> x='hello'
>>> type(x)              #查看 x 的类型,返回值为 <class 'str'>
>>> x * 3                #将字符串复制操作,结果为 'hellohellohello'
>>> x+'5'                #字符串拼接操作,结果为 'hello5'
```

Python 中字符串类型用 str 表示。上面代码中,运算符＋和 * 被重载了,运算符重载是指根据所关联的操作数的不同,表现出不同的运算功能。很显然,针对字符串,运算符＋为拼接操作,运算符 * 为复制式拼接。

字符串变量一旦创建,不可修改,但字符串仍有一些常见操作。例如,len()函数,可用来获取字符串的长度,如 len('abcd')结果为 4;也可以对字符串做索引操作,从左到右,索引起始值从 0 开始,当索引值为负数时,表示从右往左进行索引,－1 表示最后一个元素,示例如下。

```
>>> x='abcd'
>>> x[-1]               #结果为 'd'
>>> x[1]                #结果为 'b'
```

此外,字符串截取也是一个字符串的常用操作。假设一个字符串为 s,s[start:end],得到一个从索引 start 开始,到 end－1 处字符构成的字符串。其中 start 若省略,表示从索引 0 开始;end 若省略,表示 end 值为 len(s),即字符串 s 的长度。

```
>>> x='abcde'
>>> x[0:2]              #结果为 'ab'
>>> x[1:]               #结果为'bcde'
>>> x[:5]               #结果为'abcde'
```

Python 转义字符,当需要在字符中使用特殊字符时,Python 用反斜杠“\”转义字符,如表 3-1 所示。

表 3-1　转义字符表

转义字符	描　　述	转义字符	描　　述
\(在行尾时)	续行符	\n	换行
\\	反斜杠符号	\v	纵向制表符
\'	单引号	\t	横向制表符
\"	双引号	\r	回车
\a	响铃	\f	换页
\b	退格(Backspace)	\oyy	八进制数,y 代表 0～7 的字符,例如:\012 代表换行
\e	转义	\xyy	十六进制数,以\x 开头,yy 代表的字符,例如:\x0a 代表换行
\000	空	\other	其他的字符以普通格式输出

Python 中有两个常用的函数,input()和 print(),input()函数接收一个标准输入数据,返回字符串类型数据。print()函数用于打印输出。

```
>>> a=input()
5
>>> a
'5'
>>> a=input("please input a:")
please input a:5
>>> a
'5'
>>>print(1)
1
>>> print("Hello World")
Hello World
>>> a = 1
>>> b = 'hello'
>>> print(a,b)
1 hello
```

Python 支持格式化字符串的输出，通常利用%来实现。%的主要作用是将数据转换为指定的输出格式，可将数字、字符传递到字符串里所在的位置，传递的时候按照顺序传递。

```
>>> name="zhang san"
>>> age = 19
>>> print("my name is %s, age is %d"%(name,age))
my name is zhang san, age is 19
```

以上代码将 name 变量以字符串的格式显示，同时将 age 变量以整数的格式显示。常用字符格式化符号如表 3-2 所示。

表 3-2 常用字符格式化符号

符号	描　述	符号	描　述
%c	格式化字符及其 ASCII 码	%f	格式化浮点数字，可指定小数点后的精度
%s	格式化字符串	%e	用科学记数法格式化浮点数
%d	格式化整数	%E	作用同%e，用科学记数法格式化浮点数
%u	格式化无符号整型	%g	%f 和%e 的简写
%o	格式化无符号八进制数	%G	%F 和 %E 的简写
%x	格式化无符号十六进制数	%p	用十六进制数格式化变量的地址
%X	格式化无符号十六进制数(大写)		

此外，Python 代码中可以增加注释内容以增强代码的可读性。如果是单行注释，可以使用在行首添加＃的方式。若是多行注释，可以使用三引号的方式。注释示例如下。

```
#这是一行单行注释内容
'''以下为多行注释
注释 1
注释 2
'''
```

3.4　Python 序列

序列是一种数据存储方式，用来存储一系列的数据。Python 3 常用的序列对象包含字符串、列表、元组、字典、集合。

3.4.1　列表

列表(list)是一个有序的对象集合，列表中每个元素都有一个索引值。列表中的元素均放在方括号内，元素之间用逗号分隔。元素自左向右排列，左边为头，右边为尾。列表中可以包含不同类型的元素。

```
>>> a=[1,2,3,4,5]
>>> b=[1,'a',2,'b',3]
```

与字符串类似，列表可以通过索引来访问列表中的某个元素。索引从左至右递增。若索引为负数，从右边开始递减。例如，索引－1 指向倒数第一个元素，索引 0 指向正数第一个元素。与字符串不同的是，列表是可修改的数据类型，列表的元素可增加、删除和修改。

```
>>> list1=[1,2,3,4,5,6]
>>> list1[2]=10                #将第三个元素修改为 10
>>> list1
[1, 2, 10, 4, 5, 6]
```

另外，列表元素增加的方法主要有如下几种：追加 append()、插入 insert()、扩展 extend()。

```
>>> list1.append(22)           #列表末尾添加一个元素
>>> list1
[1, 2, 10, 4, 5, 6, 22]
>>> list1.insert(2,20)         #指定位置插入一个元素
>>> list1
[1, 2, 20, 10, 4, 5, 6, 22]
>>> list1.extend([5,6,7])      #列表末尾扩展多个元素
>>> list1
[1, 2, 20, 10, 4, 5, 6, 22, 5, 6, 7]
```

列表元素删除的方法主要有两种，方法 pop()用于从列表中删除指定索引位置的元素，如没有参数，默认删除最后一个元素，并返回这一元素。

```
>>> list1.pop(2)               #删除索引为 2 的元素
20
>>> list1.pop()
7
```

方法 remove()用于删除从左至右第一个指定值的元素。如果删除元素不存在，则报错。该函数不返回任何值。

```
>>> list1
[1, 2, 10, 4, 5, 6, 22, 5, 6]
>>> list1.remove(2)            #删除元素 2
```

```
>>> list1
[1, 10, 4, 5, 6, 22, 5, 6]
>>> list1.remove(33)          #因为 list1 中没有 33 这个元素,所以会报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

此外,列表还有一些常用操作如表 3-3 所示。

表 3-3 列表常用操作

方 法	作 用	方 法	作 用
list.count(obj)	统计 obj 在列表中出现的次数	len(list)	返回列表 list 的元素个数
list.index(obj)	从列表中找出 obj 第一个匹配项的索引位置	max(list)	返回列表 list 中的最大值
list.reverse()	反向列表中元素,原地操作	min(list)	返回列表 list 中的最小值
list.sort(reverse=False)	对原列表进行排序,原地操作。reverse = False 为默认值,表示按照升序排列	list(seq)	将可迭代对象 seq 转换为列表

以下代码利用列表实现对输入的年、月、日进行格式化打印。

```
months = [ 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']  #利用列表 months 保存每个月份的名字
endings = ['st', 'nd', 'rd'] + 17 * ['th'] + ['st', 'nd', 'rd'] + 7 * ['th'] + ['st']
                                                  #创建日期的后缀列表
year = input('Year: ')
month = input('Month (1-12): ')
day = input('Day (1-31): ')

month_number = int(month)
day_number = int(day)
month_name = months[month_number-1]               #基于索引的列表元素访问
ordinal = day + endings[day_number-1]
print(month_name + ' ' + ordinal + ', ' + year)
```

程序运行后输入:

```
Year: 1995
Month (1-12): 12
Day (1-31): 2
```

输出为:

```
December 2nd, 1995
```

3.4.2 元组

Python 的元组 tuple 与列表类似,不同之处在于元组的元素不能修改。元组使用圆括号“()”表示,圆括号也可以省略。元组创建很简单,只需要在括号中添加元素,元素之间使

用逗号隔开。

```
>>> tup1 = ('physics', 'chemistry', 1997, 2000)
>>> tup2 = (1, 2, 3, 4, 5 )
>>> tup3 = "a", "b", "c", "d"      #任意无符号的对象,以逗号隔开,默认为元组
>>> tup=()                         #创建空列表
>>> tup=(1,)                       #元组中只包含一个元素时,需要在元素后面添加逗号
```

与字符串和列表一样,元组之间可以使用+号和*号进行运算。这就意味着它们可以组合和复制,运算后会生成一个新的元组。元组支持索引和截取操作,也支持 Python 内置函数,如 max()、min()、len()、tuple()。但是元组不支持修改、添加、删除操作。

```
>>> a=(1,2,3)
>>> b=3,4,5
>>> len(a)                         #返回值为 3
>>> max(b)                         #返回值为 5
>>> c=[1,2,3,5]
>>> tuple(c)                       #将一个列表转换为元组
(1, 2, 3, 5)
>>> c=(1,3,4,5,6)
>>> c[2:]                          #切片
(4, 5, 6)
>>> c+(7,8)                        #拼接
(1, 3, 4, 5, 6, 7, 8)
>>> c*2                            #复制
(1, 3, 4, 5, 6, 1, 3, 4, 5, 6)
```

3.4.3　字典与集合

字典是 Python 中唯一的内置映射类型,其中的值不按顺序排列,而是存储在键下。键可能是数、字符串或元组。字典的每个键值对(key:value)用冒号分隔,每个键值对之间用逗号分隔,整个字典包括在花括号中,格式为 d={key1: value1, key2: value2, key3: value3}。

```
>>> emptyDict = {}           #创建一个空字典
>>> tinydict = {'Name': 'Zhangsan', 'Age': 7, 'Class': 'First'}
>>> print ("tinydict['Name']: ", tinydict['Name'])
tinydict['Name']: Zhangsan
```

字典修改操作如下。

```
>>> tinydict['Age'] = 8    #更新键为'Age'的值为 8,若键'Age'不存在,则添加一个键值对
>>> del tinydict['Name']   #删除键为'Name'的元素
```

字典值可以是任何的 Python 对象,既可以是标准的对象,也可以是用户定义的,但键不行。字典中不允许同一个键出现两次。创建时如果同一个键被赋值两次,后一个值会被记住。键必须不可变,因此可以用数字、字符串或元组充当,而用列表就不行。字典常用操作如表 3-4 所示。

表 3-4 字典常用操作

方　法	作　用	方　法	作　用
>>>tinydict={'Name': 'Runoob', 'Age': 7, 'Class': 'First'}	创建一个字典	>>>tinydict.keys() dict_keys(['Name', 'Age', 'Class'])	返回所有关键字列表
>>>tinydict.get('Name') 'Runoob'	返回指定键的值，如果键不在字典中返回 default 设置的默认值	>>>tinydict.items() dict_items([('Name', 'Runoob'), ('Age', 7), ('Class', 'First')])	返回字典元素列表
>>>'age' in tinydict False	如果键‘age’在字典 dict 里返回 true，否则返回 false	>>>tinydict.clear()	删除字典内所有元素
>>>tinydict.values() dict_values(['Runoob', 7, 'First'])	返回所有值列表		

集合(set)是一个无序的不重复元素序列。可以使用花括号“{}”或者 set()函数创建集合，需要注意的是创建一个空集合必须用 set()而不是“{}”，因为“{}”是用来创建一个空字典的。

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                #集合会自动将重复的元素删除
{'orange', 'banana', 'apple', 'pear'}
```

下面展示两个集合间的运算。

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                        #集合 a 中包含而集合 b 中不包含的元素
{'r', 'd', 'b'}
>>> a | b                        #集合 a 或 b 中包含的所有元素
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                        #集合 a 和 b 中都包含了的元素
{'a', 'c'}
>>> a ^ b                        #不同时包含于 a 和 b 的元素
{'r', 'd', 'b', 'm', 'z', 'l'}
```

以下代码利用字典实现中文单词的分词和统计功能。其中中文分词使用了 jieba 库中的函数来实现。

```
import jieba                      #导入 jieba 中文分词库
def getWords():                   #定义分词函数
    txt = open('三国演义.txt', 'r', encoding = 'utf-8').read()  #读入 TXT 文本文件
    words = jieba.lcut(txt)       #利用 jieba 函数进行分词
    counts = {}                   #定义空字典,用来统计词频
    for word in words:
        if len(word) == 1:        #不统计长度为 1 的词
```

```
            continue
        else:
            counts[word] = counts.get(word, 0) + 1          #统计 word 出现的次数
    word_list = list(counts.items())                        #将字典转换为列表
    word_list.sort(key = lambda x : x[1], reverse = True)   #按照词频由高到低排序
    return word_list
word_list = getWords()
for i in range(30):
    word, count = word_list[i]
    print('{0:^10}{1:{3}^10}{2:^15}'.format(i+1, word, count,chr(12288)))
#格式化输出
```

3.5　控制语句

有了合适的数据类型和数据结构后，还要依赖于选择和循环结构来实现特定的业务逻辑。一个完整的选择结构或循环结构可以看作是一个大的“语句”，从这个角度来讲，程序中的多条“语句”是顺序执行的。

3.5.1　条件表达式

在选择结构和循环结构中，都要根据条件表达式的值来确定下一步的执行流程。条件表达式的值只要不是 False、0（或 0.0、0j 等）、空值 None、空列表、空元组、空集合、空字典、空字符串、空 range 对象或其他空迭代对象，Python 解释器均认为与 True 等价。从这个意义上来讲，所有的 Python 合法表达式都可以作为条件表达式，包括含有函数调用的表达式。

在条件表达式中，经常会用到关系运算符和逻辑运算符。

```
>>> 1>4>6              #等价于 1>4 and 4>6,结果为 False
>>> 3 and 5            #与运算为真时,and 后面的值为最终结果,因此结果为 5
```

3.5.2　分支结构

常见的选择结构有单分支选择结构、双分支选择结构、多分支选择结构及嵌套的分支结构，也可以构造跳转表来实现类似的逻辑。

单分支选择结构语法如图 3-3 所示，其中表达式后面的冒号“:”是不可缺少的，表示一个语句块的开始，并且语句块必须做相应的缩进，一般是以 4 个空格为缩进单位。

```
if 表达式:
    语句块
```

当表达式的值为 True 或其他与 True 等价的值时，表示条件满足，语句块被执行，否则该语句块不被执行，而是继续执行后面的代码。

下面的代码演示了单分支选择结构的用法。

```
a, b = 10, 5
if a>b:
a,b=b,a                 #交换两个变量的值
print (a,b)             #结果为: 5 10
```

在 Python 中，代码的缩进非常重要，缩进是体现代码逻辑关系的重要方式，同一个代码块必须保证相同的缩进量。

双分支选择结构的语法如下。

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

当表达式的值为 True 或其他等价值时，执行语句块 1，否则执行语句块 2。语句块 1 或语句块 2 总有一个会执行，然后执行后面的代码(如果有)，如图 3-4 所示。

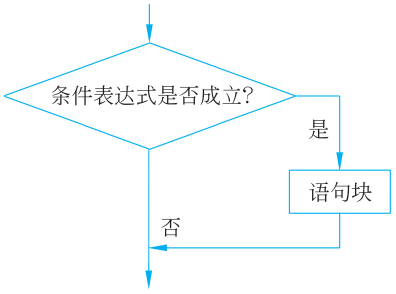

图 3-3 单分支结构

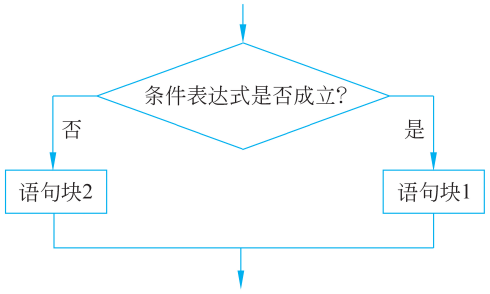

图 3-4 双分支结构

当出现多个并列的条件判断时，可以采用多分支选择结构，语法如下。

```
if 表达式 1:
    语句块 1
elif 表达式 2:
    语句块 2
elif 表达式 3:
    语句块 3
...
else:
    语句块 n
```

多分支语句示例如下，该代码实现狗与人的对应年龄转换。

```
age = int(input("请输入你家狗狗的年龄: "))
print("")
if age <= 0:
    print("你是在逗我吧!")
elif age == 1:
    print("相当于 14 岁的人。")
elif age == 2:
    print("相当于 22 岁的人。")
elif age > 2:
    human = 22 + (age -2) * 5
print("对应人类年龄: ", human)
```

if 嵌套，在嵌套 if 语句中，可以把 if-elif-else 结构放在另外一个 if-elif-else 结构中。以下代码判断一个数字是否可以被 2 和 3 整除，并打印相应的结论。

```
num=int(input("输入一个数字: "))
if num%2==0:
    if num%3==0:
        print("你输入的数字可以整除 2 和 3")
    else:
        print("你输入的数字可以整除 2,但不能整除 3")
else:
    if num%3==0:
        print("你输入的数字可以整除 3,但不能整除 2")
    else:
        print("你输入的数字不能整除 2 和 3")
```

3.5.3　循环结构

Python 主要有 for 循环和 while 循环两种形式的循环结构，多个循环可以嵌套使用，并且还可以和选择结构嵌套使用来实现复杂的业务逻辑。while 循环一般用于循环次数难以提前确定的情况，当然也可以用于循环次数确定的情况；for 循环一般用于循环次数可以提前确定的情况，尤其适用于枚举、遍历序列或迭代对象中元素的场合。

```
while 条件表达式:
    循环体
for 取值 in 可循环对象:
    循环体
```

其中可循环对象包含序列、range 对象等。下面的代码用来输出 1～100 之间能被 7 整除但不能同时被 5 整除的所有整数，利用 range 产生 1～100 的整数。

```
for i in range(1,101):
    if i%7==0 and i%5! =0:
        print(i)
```

其中 range([start,] end [,step]) 返回 range 对象，其中包含左闭右开区间[start,end)内以 step 为步长的整数。

```
>>> for i in range(5):
    print(i,end=' ')
0 1 2 3 4
```

下面的代码演示了带有 else 子句的循环结构，该代码用来计算 1＋2＋3＋…＋99＋100 的结果。

```
s=0
for i in range(1,101):          #不包括 101
    s+=i
else:
    print (s)
```

for-else 表示如下逻辑，for 中的语句和普通的没有区别，else 中的语句会在循环正常执行完(即 for 不是通过 break 跳出而中断的)的情况下执行，while-else 同理。

break 与 continue 语句在 while 循环和 for 循环中都可以使用，并且一般常与选择结构

或异常处理结构结合使用。一旦 break 语句被执行，将使 break 语句所属层次的循环提前结束；continue 语句的作用是提前结束本次循环，忽略 continue 之后的所有语句，提前进入下一次循环。

假设你要找出小于 100 的最大平方数(可以写成某个整数的平方的数)，则可以选择从 100 开始向下查找。找到一个平方数后，无须再迭代，因此直接跳出循环。

```
from math import sqrt
for n in range(99, 0, -1):
    root = sqrt(n)
    if root == int(root):
        print(n)
        break
```

3.6 函　　数

在软件开发过程中，经常有很多操作是完全相同或是非常相似的，仅仅是要处理的数据不同而已，因此我们经常需要在不同的代码位置多次执行相似甚至完全相同的代码块。很显然，从软件设计和代码复用的角度来讲，直接将代码块复制到多个相应的位置，然后进行简单修改绝对不是一个好主意。虽然这样可以使多份复制的代码彼此独立地进行修改，但这样不仅增加了代码量，也增加了代码阅读、理解和维护的难度，为代码测试和纠错带来很大的困难。一旦被复制的代码块将来被发现存在问题而需要修改，必须对所有的复制代码做同样的修改，这在实际中是很难完成的一项任务。更糟糕的情况是，由于代码量的大幅增加，导致代码之间的关系更加复杂，很可能在修补旧漏洞的同时又引入新漏洞，维护成本大幅增加。

将可能需要反复执行的代码封装为函数，然后在需要该功能的地方调用封装好的函数，不仅可以实现代码的复用，更重要的是可以保证代码的一致性，只需要修改该函数的代码，则所有调用位置均得到体现。同时，把大任务拆分成多个函数也是分治法的经典应用，复杂问题简单化，使软件开发像搭积木一样简单。

3.6.1 函数的定义与调用

在 Python 中使用 def 关键字来定义函数，然后是一个空格和函数名称，接下来是一对括号，在括号内是形式参数列表，如果有多个参数则使用逗号分隔开，括号之后是一个冒号和换行，最后是注释和函数体代码。定义函数时在语法上需要注意的问题主要有：①函数形参不需要声明其类型，也不需要指定函数的返回值类型；②即使该函数不需要接收任何参数，也必须保留一对空的括号；③括号后面的冒号必不可少；④函数体相对于 def 关键字必须保持一定的空格缩进。

```
def 函数名(参数列表):
    '''注释'''
    函数体
    return
```

以下代码定义并调用了一个简单的 add_two()函数。该函数包含 2 个输入参数 a 和 b。

```
#定义函数
def add_two( a, b ):          #a、b 为形参
   #计算加法结果
   print (a+b)
   return
#调用函数
add_two(10,20)                #10、20 为实参
add_two(400,200)
```

在 Python 中，定义函数时也不需要声明函数的返回值类型，而是使用 return 语句结束函数执行的同时返回任意类型的值，函数返回值类型与 return 语句返回表达式的类型一致。不论 return 语句出现在函数的什么位置，一旦得到执行将直接结束函数的执行。如果函数没有 return 语句、有 return 语句但是没有执行到或者执行了不返回任何值的 return 语句，解释器都会认为该函数以 return None 结束，即返回空值。

3.6.2　参数的传递方式

函数定义时括号内是使用逗号分隔开的形参列表(parameters)，函数可以有多个参数，也可以没有参数，但定义和调用时括号必须有，表示这是一个函数并且不接收参数。调用函数时向其传递实参(arguments)，根据不同的参数类型，将实参的值或引用传递给形参。

参数传递有如下几种基本方法。

(1) 位置参数(positional arguments)是比较常用的形式，调用函数时实参和形参的顺序必须严格一致，并且实参和形参的数量必须相同。

```
>>>def demo(a,b,c): #所有形参都是位置参数
       print(a,b,c)
>>>demo(3,4,5)       #结果为 3 4 5
>>>demo(1,2,3,4)     #实参与形参的数量必须相同，若数量不匹配则会报如下错误
                     #TypeError: demo() takes 3 positional arguments but 4 were given
```

(2) 默认值参数。在定义函数时，Python 支持默认值参数，在定义函数时可以为形参设置默认值。在调用带有默认值参数的函数时，可以不用为设置了默认值的形参进行传值，此时函数将会直接使用函数定义时设置的默认值，当然也可以通过显式赋值来替换其默认值。也就是说，在调用函数时是否为默认值参数传递实参是可选的，具有较大的灵活性，在一定程度上类似于函数重载的功能，同时还能在为函数增加新的参数和功能时，通过为新参数设置默认值来保证向后兼容，而不影响老用户的使用。需要注意的是，在定义带有默认值参数的函数时，任何一个默认值参数右边都不能再出现没有默认值的普通位置参数，否则会提示语法错误。带有默认值参数的函数定义语法如下。

```
def 函数名(形参名=默认值):
    函数体

def demo( a=1,b=2 ):      #默认值参数
   print (a+b)
   return

demo()                    #函数调用，使用默认值，结果为 3
demo(3)                   #函数调用，a=3，b 为默认值，结果为 5
```

(3) 关键参数主要指调用函数时的参数传递方式,与函数定义无关。通过关键参数可以按照参数名字传递值,明确指定哪个值传递给哪个参数,实参顺序可以和形参顺序不一致,但不影响参数值的传递结果,这避免了用户需要牢记参数位置和顺序的麻烦,使函数的调用和参数传递更加灵活方便。

```
>>>def demo (a,b,c=5):
        print (a,b,c)
>>>demo(3,7)                    #按位置传递参数,结果为 3 7 5
>>> demo(c=8,a=9,b=0)           #关键参数,结果为 9 0 8
```

3.6.3 变量的作用域

变量起作用的代码范围称为变量的作用域,不同作用域内的同名变量之间互不影响,就像不同文件夹中的同名文件之间互不影响一样。在函数外部和在函数内部定义的变量,其作用域是不同的,在函数内部定义的变量一般为局部变量,在函数外部定义的变量为全局变量。不管是局部变量还是全局变量,其作用域都是从定义的位置开始的,在此之前无法访问。

在函数内定义的局部变量只在该函数内可见,当函数运行结束后,在其内部定义的所有局部变量将被自动删除而不可访问。在函数内部使用关键字 global 定义的全局变量,当函数结束以后仍然存在并且可以访问。如果在函数内部修改一个定义在函数外部的变量值,必须使用关键字 global 明确声明,否则会自动创建新的局部变量。在函数内部通过关键字 global 来声明或定义全局变量,分为以下两种情况。

(1) 一个变量已在函数外定义,如果在函数内需要修改这个变量的值,并将修改的结果反映到函数之外,可以在函数内用关键字 global 明确声明要使用已定义的同名全局变量。

(2) 在函数内部直接使用关键字 global 将一个变量声明为全局变量,如果在函数外部没有定义该全局变量,在调用这个函数后,会创建新的全局变量。或者也可以这么理解:①在函数内如果只引用某个变量的值而没有为其赋新值,该变量为(隐式的)全局变量;②如果在函数内某条代码有为变量赋值的操作,该变量就被认为是(隐式的)局部变量,除非在函数内赋值操作之前显式地用关键字 global 进行了声明。

下面的代码演示了局部变量和全局变量的用法。

```
>>>def demo():
        global x              #声明或创建全局变量,必须在使用 x 之前执行
        x=3                   #修改全局变量的值
        y=4                   #局部变量
        print(x,y)
>>>x=5                       #在函数外部定义了全局变量 x
>>>demo()                    #本次调用修改了全局变量 x 的值
3 4
>>>x
3
>>>y                         #局部变量在函数运行结束后自动删除,不再存在
NameError: name 'y' is not defined
>>>del x                     #删除了全局变量
```

```
>>> x
NameError: name 'x' is not defined
>>>demo()          #本次调用创建了全局变量
3 4
>>>x
3
```

3.7　模块的使用

Python 默认安装仅包含基本或核心模块，启动时也仅加载基本模块，在需要时再显式地导入和加载标准库和第三方扩展库，这样可以减小程序运行的压力，并且具有很强的可扩展性。从“木桶原理”的角度来看，这样的设计与安全配置时遵循的“最小权限”原则是一致的，也有助于提高系统的安全性。模块导入的语法如下。

```
import 模块名 [as 别名]
```

用这种方式导入后，使用时需要在对象之前加上模块名作为前缀，必须以“模块名.对象名”的形式进行访问。如果模块名很长，可以为导入的模块设置一个别名，然后用“别名.对象名”的方式来使用其中的对象。

```
>>>import math           #导入标准库 math
>>>print(math.sqrt(4))
>>>import math as th     #导入标准库 math,并设置别名为 th
>>>print(th.sqrt(10))
```

运行结果：

```
2.0
3.1622776601683795
```

也可以用如下语法使用模块。

```
from 模块名 import 对象名 [as 别名]
```

用这种方式仅导入明确指定的对象，并且可以为导入的对象确定一个别名。这种导入方式可以减少查询次数、提高访问速度，同时也可以减少开发者需要输入的代码量，因为不需要使用模块名作为前缀。

```
from math import sqrt              #只导入模块中的指定对象
print(sqrt(16))
from math import sqrt as f         #给导入的对象起个别名
print(f(25))

>>> import math
>>> dir(math)                      #返回参数的属性、方法列表
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos',
'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

```
>>> help(math.sqrt)                    #查看 math.sqrt 函数用法，输出以下内容
Help on built-in function sqrt in module math:
sqrt(x, /)
Return the square root of x.
```

除了可以在开发环境或命令提示符环境中直接运行，任何 Python 程序文件都可以作为模块导入并使用其中的对象，这也是实现代码复用的重要形式。通过 Python 程序的__name__属性可以识别程序的使用方式，每个 Python 脚本在运行时都会有一个__name__属性，如果脚本作为模块被导入，则其__name__属性的值被自动设置为模块名；如果脚本作为程序直接运行，则其__name__属性值被自动设置为字符串'__main__'。例如，假设程序 hello.py 中的代码如下。

```
def main():                           #def 是用来定义函数的 Python 关键字
    if __name__=='__main__':          #选择结构，识别当前的运行方式
        print('This program is running directly.')
    elif __name__  == 'hello':        #冒号、换行、缩进表示一个语句块的开始
        print('This program is used as a module.')
main()                                #调用上面定义的函数
python hello.py                       #直接从命令行运行以上函数
This program is running directly.
```

从 Python 中以模块导入：

```
>>> import hello
This program is used as a module.
```

3.8 面向对象基础

面向对象程序设计(object oriented programming，OOP)的思想主要是针对大型软件设计提出的，它使软件设计更加灵活，能很好地支持代码复用和设计复用，并且代码具有更好的可读性和可扩展性，因此大幅降低了软件开发的难度。面向对象程序设计的一个关键观念是将数据及对数据的操作封装在一起，组成一个相互依存、不可分割的整体(对象)，不同对象之间通过消息机制来通信或同步。对于相同类型的对象(instance)进行分类、抽象后，得出共同的特征形成了类(class)，面向对象程序设计的关键，就是如何合理地定义这些类并组织多个类之间的关系。

Python 是面向对象的解释型高级动态编程语言，完全支持面向对象的基本功能，如封装、继承、多态以及对基类方法的覆盖或重写。创建类时，用变量形式表示对象特征的成员称为数据成员(attribute)，用函数形式表示对象行为的成员称为成员方法(method)。数据成员和成员方法统称为类的成员。需要注意的是，Python 中对象的概念很广泛，Python 中的一切内容都可以称为对象，函数是对象，类也是对象。

3.8.1 类的定义与实例化

Python 使用 class 关键字来定义类，class 关键字之后是一个空格，接下来是类的名字，