详解100个趣味编程算法实例，专为Python初学者量身打造
培养编程兴趣，拓宽编程思维，提高编程能力和算法设计能力

趣学Python算法100例

刘河飞 闫凯峰◎编著

机械工业出版社
China Machine Press

趣学Python算法100例

刘河飞　闫凯峰　编著

ISBN：978-7-111-66598-4

本书纸版由机械工业出版社于2020年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

前言

当前，随着人工智能的发展，Python成为深受程序员欢迎的编程语言。它是一门面向对象的解释型动态编程语言，其语法灵活，容易上手。通过对Python语言的学习，读者可以采用编程的方式解决实际生活中的许多问题。

本书以通俗易懂的语言详尽地介绍了用Python语言编写的100个算法实例。这些实例大体上按照“问题描述→问题分析→算法设计→确定程序框架→程序编码实现→运行结果→问题拓展”的流程进行讲解，每个实例又根据实际需要有所取舍。这些实例兼顾了趣味性、实用性和可操作性，而且大多是围绕一些经典算法问题展开的。

相信通过学习和演练本书中的实例，读者可以极大地提高编程兴趣，拓宽编程思维，提高编程能力和算法设计能力，体会程序设计的乐趣，最终解决生活和工作中的相关问题。

本书特色

1．实例详解

本书用通俗易懂的语言详细介绍Python编程的100个常见算法实例。在介绍实例如何实现的同时将程序开发的基本原理、方法和技术融入其中，并对涉及的Python模块做了详细的扩展讲解。

2．趣味性强

本书选取的实例都是趣味性较强的例子，可以极大地提高读者的编程兴趣，让读者能充分感受到学习Python编程的乐趣和魅力。

3．代码详尽

本书所有实例代码完整，注释详尽，流程图规范，而且均通过了测试，可以正常运行，便于读者自己动手编写并验证每一个实例程序。

4．讲解透彻

本书内容按照不同类型的趣味问题进行分类，力求将每一类问题都能讲解透彻，并总结出解决同类问题的一般规律，以便读者在遇到类似问题时可以快速解决。

5．注重基础

本书在注重实例趣味性的基础上还加强了Python语言的语法知识讲解，对解决问题时涉及的重要知识点进行详尽说明，并提供相关的方法及操作示例。

6．拓展训练

本书中的很多实例都提供了拓展训练，旨在帮助读者拓展编程思维，从而在碰到实际问题时能举一反三、融会贯通，有思路去解决。

本书内容

第1章为趣味算法入门，通过一些经典算法的介绍，带领读者走进计算机算法的世界，让读者学会使用Python语言实现一个算法。

第2章为趣味数学问题，从与生活相关的一些小例子中抽象出数学公式，再用Python语言将这些模型化的数学问题表达出来，并得出问题的求解答案。

第3章为各种趣味整数，对各类整数问题进行详细讲解，让读者体会到数学之美。

第4章为趣味分数，讲述各类与分数相关的趣味问题，并带领读者掌握相关算法。

第5章为趣味素数，介绍判别素数的方法及几种特殊素数的验证方法，让读者做到学以致用。

第6章为趣味逻辑推理，提供几个有趣的小故事，引导读者进行分析判断，并使用Python语言来实现，以及表达逻辑推理的过程，从而求解出最终答案。

第7章为趣味游戏，使用Python语言编写几个小游戏，通过趣味小游戏带领读者学习编程，从而激发读者的学习兴趣，培养读者的逻辑思维。

第8章为趣味数组，讲解Python语言中列表（数组）的使用方法及相关的编程技巧。

第9章为趣味函数递归，深入阐述Python语言中递归的概念，将递归融入各个问题的讲解中，让读者理解递归的思想，学会使用递归思想来解决实际问题。

第10章为定理与猜想，使用Python语言对常用的一些定理和猜想进行验证。

第11章为趣味图形，演示如何使用Python语言绘制出一些简单而又常用的图形，帮助读者掌握使用Python绘图的技巧，同时介绍具有绘图功能的相关Python模块。

第12章为其他趣味问题，介绍一些综合性较强的编程问题，以提高读者的编程动手能力。

本书配套资源

本书涉及的所有实例源代码文件及拓展程序文件需要读者自行下载。请在华章官网www.hzbook.com 上搜索到本书，然后单击

“资料下载”按钮，即可在本书页面上找到“配书资源”下载链接。

本书读者对象

- Python编程初学者；
- Python编程爱好者；
- 程序设计爱好者；
- 算法设计爱好者；
- 高校理工科专业的学生；
- 培训机构的学员。

售后支持

由于作者水平所限，加之写作时间有限，书中可能还存有一些疏漏和不当之处，敬请各位读者指正。阅读本书时若有疑问，请发E-mail到hzbook2017@163.com 。

刘河飞

2020年8月

第1章　趣味算法入门

算法是解决特定问题的方法，是程序设计的基础和灵魂。作为一个算法，应具备5个特性，即有穷性、确定性、可行性、输入和输出。计算机算法可分为两大类，即数值计算算法和非数值计算算法。数值计算算法的目的是求解数值，例如求方程的根；非数值计算算法主要用于处理事务领域的问题，如排序、查找等。本章旨在通过介绍一些典型算法，引领读者进入计算机算法的世界，了解算法设计，学会用Python语言来实现算法。

1.1　抓交通肇事犯

1．问题描述

一辆卡车违反交通规则，撞人后逃跑。现场有三人目击该事件，但都没有记住车号，只记下了车号的一些特征。甲说：牌照的前两位数字是相同的；乙说：牌照的后两位数字是相同的，但与前两位不同；丙是数学家，他说：4位的车号刚好是一个整数的平方。请根据以上线索求出车号。

2．问题分析

按照题目的要求造出一个前两位数相同、后两位数相同且相互间又不同的4位整数，然后判断该整数是否是另一个整数的平方。即求一个4位数$a_1\ a_2\ a_3\ a_4$，满足如下条件：

$$\begin{cases} a_1 = a_2 & 1 \leqslant a_1 \leqslant 9, 0 \leqslant a_2 \leqslant 9 \\ a_3 = a_4 & a_3 \geqslant 0, a_4 \leqslant 9 \\ a_1 \neq a_3 & \\ 1000 \times a_1 + 100 \times a_2 + 10a_3 + a_4 = x^2 & x \in \mathrm{Z} \end{cases}$$

3．算法设计

本题目是数值计算问题，求解不定方程。对于这种求解不定方程组的问题，一般采用穷举循环，首先设计双层循环穷举出所有由前两位数和后两位数组成的4位数车牌，然后在最内层穷举出所有平方后值为4位数并且小于车牌号的数，判断该数是否与车牌相等，若相等，则打印车牌。

4．确定程序框架

程序流程图如图1.1所示。

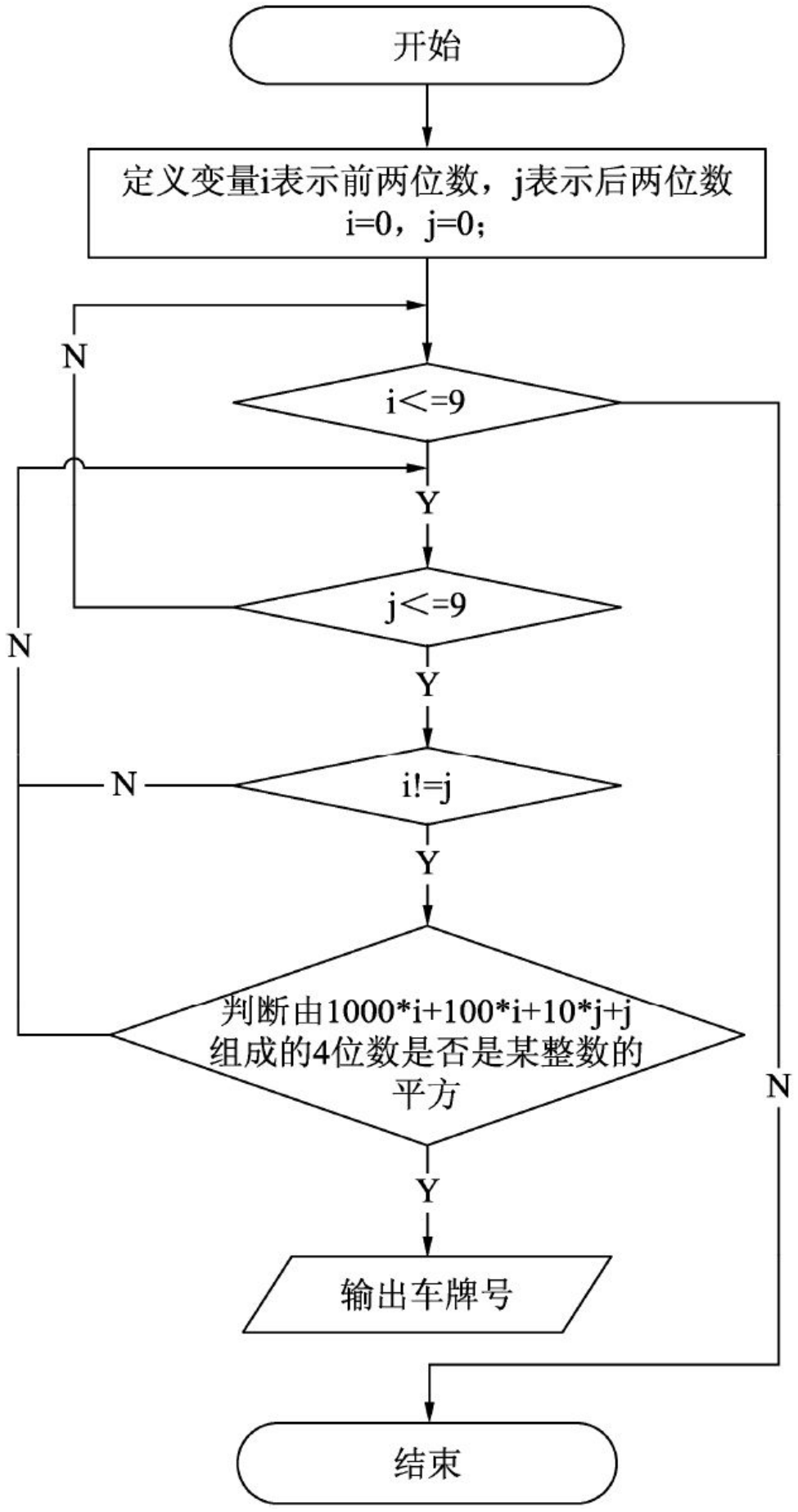

开始
定义变量i表示前两位数，j表示后两位数
i=0，j=0；
i<=9
N
Y
j<=9
N
Y
i!=j
N
Y
判断由1000*i+100*i+10*j+j
组成的4位数是否是某整数的
平方
N
Y
输出车牌号
结束

图1.1　程序流程图

根据流程，构建程序框架如下：

```
if __name__=="__main__":
    # i代表前两位车牌号数字，j代表后两位车牌号的数字，k代表车牌号
    for i in range(10):
        for j in range(10):          # 穷举前两位和后两位车牌数字
            # 判断前两位和后两位数字是否相同
            if i != j:
                # 组成4位车牌号码
                k = 1000 * i + 100 * i + 10 * j + j
                # 判断k是否是某个数的平方，是就输出
```

5. 判断车牌k是否为某个整数的平方

再次利用循环来实现，循环变量temp求平方后和车牌号k比较，相等则找到车牌号，优化算法，temp的初值应该从31开始，因为小于30的数的平方小于4位数。故该层循环为最内层循环，对每一个车牌号均作如此操作。

```
for temp in range(31, 100):
    if temp * temp == k:
        print("车牌号为: ", k)
```

6. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 抓交通肇事犯

if __name__=="__main__":
    # i代表前两位车牌号数字，j代表后两位车牌号的数字，k代表车牌号
    for i in range(10):
        for j in range(10):                                    # 穷举前两位和后两位车
牌数字
            # 判断前两位和后两位数字是否相同
            if i != j:
                # 组成4位车牌号码
                k = 1000 * i + 100 * i + 10 * j + j
                # 判断k是否是某个数的平方，是就输出
                for temp in range(31, 100):
                    if temp * temp == k:
                        print("车牌号为: ", k)
```

7. 运行结果

在PyCharm下运行程序，结果如图1.2所示。

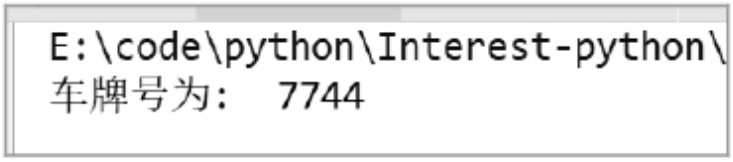


图1.2　运行结果

8. 问题拓展

针对上述程序，如果已经找到相应的车牌号，请读者考虑循环是否还需要继续呢？答案是肯定的，因为算法在设计穷举循环的时候，并没有在找到车牌的时候就退出循环，而是继续穷举其他i、j的情况。我们可以改进算法，设置一个“标识变量”，该变量初值为0，一旦找到车牌号，则改变该标识变量的值为1，每次循环判断一下标识变量的值，如果值为1，则退出所有循环，这样能有效地减少循环次数。改进后的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 抓交通肇事犯

if __name__ == "__main__":
    # i代表前两位车牌号的数字，j代表后两位车牌号的数字，k代表车牌号
    flog = 0                                                    # 循环标识变量，为1时退出所有循环
    for i in range(10):
        if flog:
            break
        for j in range(10):                 # 穷举前两位和后两位车牌数字
            if flog:
                break
            # 判断前两位和后两位数字是否相同
            if i != j:
                # 组成4位车牌号码
                k = 1000 * i + 100 * i + 10 * j + j
                # 判断k是否是某个数的平方，是就输出
                for temp in range(31, 100):
                    if temp * temp == k:
                        print("车牌号为: ", k)
                        flog = 1
                        break
```

1.2　兔子产子

1．问题描述

有一对兔子，从出生后的第3个月起每个月都生一对兔子。小兔子长到第3个月后每个月又生一对兔子，假设所有的兔子都不死，问30个月内每个月的兔子总对数为多少？

2．问题分析

兔子产子问题是一个有趣的古典数学问题，我们画一张表来找一下兔子数的规律，如表1.1所示。

表1.1　兔子数量参照表

月　数	小兔子对数	中兔子对数	老兔子对数	兔子总对数
1	1	0	0	1

（续）

月　数	小兔子对数	中兔子对数	老兔子对数	兔子总对数
2	0	1	0	1
3	1	0	1	2
4	1	1	1	3
5	2	1	2	5
6	3	2	3	8
7	5	3	5	13

说明：　不满1个月的兔子为小兔子，满1个月不满2个月的为中兔子，满3个月以上的为老兔子。

可以看出，每个月的兔子总数依次为1，1，2，3，5，8，13…这就是Fibonacci数列。总结数列规律即为从前两个月的兔子对数可以推出第3个月的兔子对数。

3．算法设计

本题目是典型的迭代循环，即是一个不断用新值取代变量的旧值，然后由变量旧值递推出变量新值的过程。这种迭代与这些因素有关：初值、迭代公式和迭代次数。经过问题分析，算法可以描述为

$$\begin{cases} fib_{n-1} = fib_{n-1} = 1 (n < 3) & \text{初值} \\ fib_n = fib_{n-1} + fib_{n-2} (n \geqslant 3) & \text{迭代公式} \end{cases}$$

用Python语言来描述迭代公式即为fib=fib1+fib2，其中fib为当前新求出的兔子对数，fib1为前一个月的兔子对数，fib2为前两个月的兔子对数，然后为下一次迭代做准备， 

，进行如下的赋值fib2=fib1，fib1=fib，要注意赋值的次序；迭代次数由循环变量控制，为所求的月数。

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 兔子产子问题

if __name__=="__main__":
    fib1 = 1
    fib2 = 1
    i = 3
    # 输出第一个月和第二个月的兔子对数
    print("%6d       %6d" %(fib1, fib2), end="       ")
    while i <= 30:
        fib = fib1 + fib2                                   # 迭代求出当前月份的兔子对数
        print("%6d" %fib, end="       ")     # 输出当前月份的兔子对数
        if i % 4 == 0:
            print()                                                 # 每行输出4个
        fib2 = fib1                                                 # 为下一次迭
代做准备，求出新的fib2
        fib1 = fib                                                  # 求出新的
fib1
        i += 1
```

5. 运行结果

在PyCharm下运行程序，结果如图1.3所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe
     1          1          2          3
     5          8         13         21
    34         55         89        144
   233        377        610        987
  1597       2584       4181       6765
 10946      17711      28657      46368
 75025     121393     196418     317811
514229     832040
```

图1.3　运行结果

6. 问题拓展

这个程序虽然是正确的，但可以进行改进。目前用3个变量来求下一个月的兔子对数，其实可以在循环体中一次求出下两个月的兔子对数，这样就可以只用两个变量来实现。这里将fib1+fib2的结果不放在fib中而是放在fib1中，此时fib1不再代表前一个月的兔子对数，而是代表最新一个月的兔子对数，再执行fib2=fib1+fib2，由于此时fib1中已经是第3个月的兔子对数了，故fib2中就是第4个月的兔子对数。可以看出，此时fib1和fib2均为最近两个月的兔子对数，循环推出下两个月的兔子对数。改进后的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 兔子产子问题

if __name__=="__main__":
    fib1 = 1
    fib2 = 1
    i = 1
    while i <= 15:  #每次求两个，因此循环变量循环到15
        print("%8d    %8d" %(fib1, fib2), end="      ")
        if i % 2 == 0:
            print()
        fib1 = fib1 + fib2                                  # 最新一个月的兔子对数
        fib2 = fib1 + fib2                                  # 第4个月的兔子对数
        i += 1
```

1.3　牛顿迭代法求方程根

1．问题描述

编写用牛顿迭代法求方程根的函数。方程为$ax^3+bx^2+cx+d=0$，系数a、b、c、d由主函数输入，求x在1附近的一个实根。求出根后，由主函数输出。

牛顿迭代法的公式：$x=x_0-\frac{f(x_0)}{f'(x_0)}$，设迭代到$|x-x_0|\leqslant 10^{-5}$时结束。

2．问题分析

牛顿迭代法是取x_0之后，在这个基础上找到比x_0更接近的方程根，一步一步迭代，从而找到更接近方程根的近似根。

设r是f(x)=0的根，选取x_0作为r的初始近似值，过点$(x_0,f(x_0))$做曲线y=f(x)的切线L，L的方程为$y=f(x_0)+f'(x_0)(x-x_0)$，求出L与x轴交点的横坐标$x_1=x_0-f(x_0)/f'(x_0)$，称x_1为r的一次近似值；过点$(x_1,f(x_2))$做曲线y=f(x)的切线，并求出该切线与x轴交点的横坐标$x_2=x_1-f(x_1)/f'(x_1)$，称x_2为r的二次近似值；重复以上过程，得到r的近似值x_n。上述过程即为牛顿迭代法的求解过程。

3．算法设计

程序流程分析：

1）在1附近找任一实数作为x_0的初值，我们取1.5，即$x_0=1.5$。

2）用初值x_0代入方程中计算此时的$f(x_0)$及$f'(x_0)$；程序中用变量f描述方程的值，用fd描述方程求导之后的值。

3）计算增量h=f/fd。

4）计算下一个x，x=x_0 -h。

5）用新产生的x替换原来的x_0 ，为下一次迭代做好准备。

6）若|x-x_0 |>=1e-5，则转到步骤（3）继续执行，否则转到步骤（7）。

7）所求x就是方程ax^3 +bx^2 +cx+d=0的根，将其输出。

4．确定程序框架

该程序的主体结构如下：

```
if __name__ == '__main__':
    # 输入方程的系数
    # 用牛顿迭代法求方程的根
    # 输出所求方程的根
```

程序流程图如图1.4所示。

5．迭代法求方程根

编写程序时要注意的一点是判定|x-x_0 |>=1e-5，许多初学者认为判定条件应该是|x-x_0 |<1e-5，从牛顿迭代法的原理可以看出，迭代的实质就是越来越接近方程根的精确值，最初给x_0 所赋初值与根的精确值是相差很多了，正是因为这个我们才需要不断地进行迭代，也就是程序中循环体的功能。在经过一番迭代之后所求得的值之间的差别也越来越小，直到求得的某两个值的差的绝对值在某个范围之内时便可结束迭代。若我们把判定条件改为|x-x_0 |<1e-5，则第一次的判断结果必为假，这样我们就不能进入循环体再次执行。希望初学者对于本类题目条件的判定要多加注意。

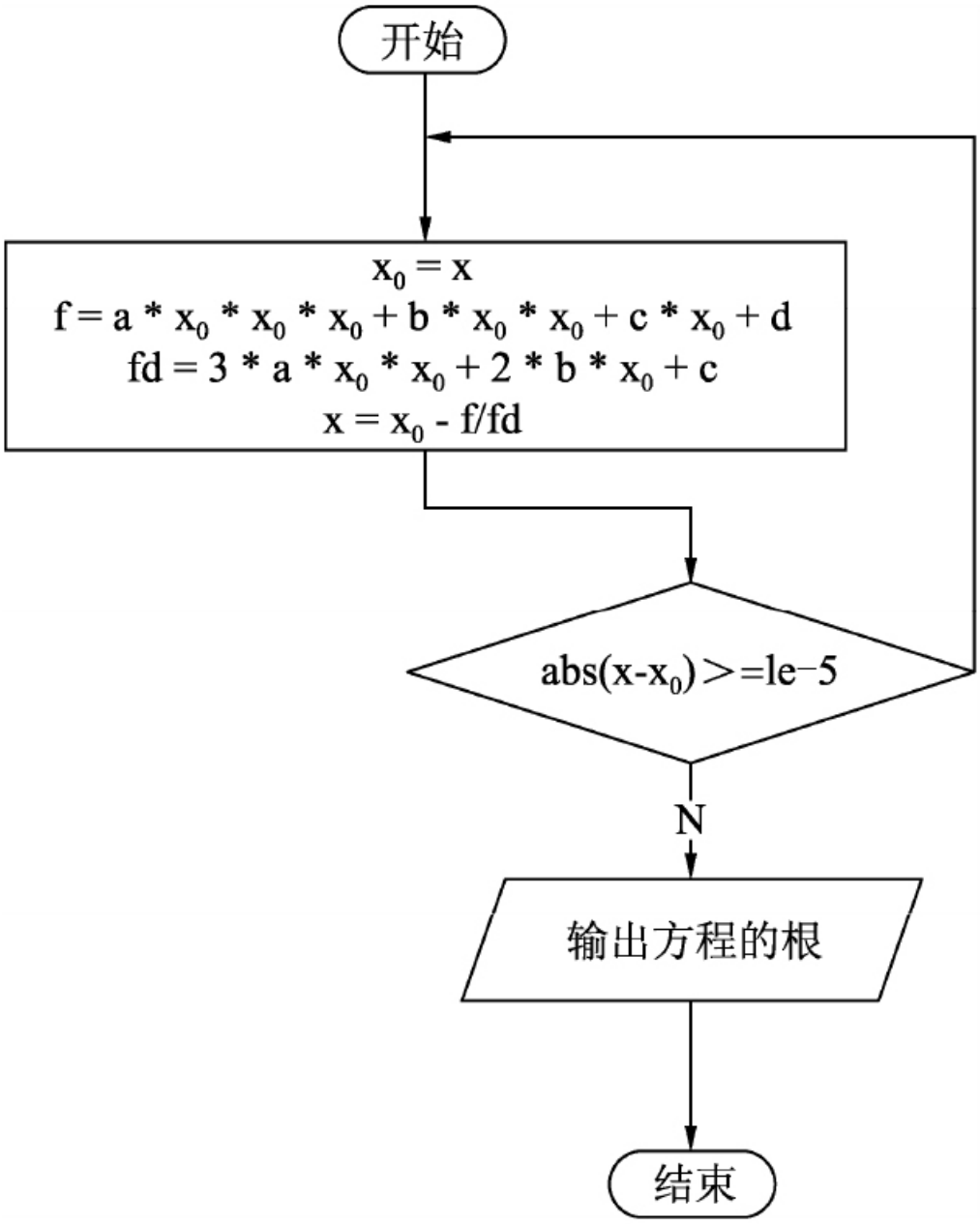


图1.4　程序流程图

定义solution()函数求方程的根。solution()函数的代码如下：

```
# 函数功能是用牛顿迭代法求方程的根
def solution(a, b, c, d):
    x = 1.5
    x0 = x                                                    # 用所求得的x的值代替
x0原来的值
    # f用来描述方程的值，fd用来描述方程求导之后的值
    f = a * x0 * x0 * x0 + b * x0 * x0 + c * x0 + d
    fd = 3 * a * x0 * x0 + 2 * b * x0 + c
    h = f / fd
    x = x0 - h                                                # 求得更接近方程根的x的值
```

```
    while abs(x - x0) >= 1e-5:
        x0 = x
        f = a * x0 * x0 * x0 + b * x0 * x0 + c * x0 + d
        fd = 3 * a * x0 * x0 + 2 * b * x0 + c
        h = f / fd
        x = x0 - h                                  # 求得更接近方程根的x的值

    return x
```

6. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 牛顿迭代法求方程根


# 函数功能是用牛顿迭代法求方程的根
def solution(a, b, c, d):
    x = 1.5
    x0 = x                                                  # 用所求得的
x值代替x0原来的值
    # f用来描述方程的值，fd用来描述方程求导之后的值
    f = a * x0 * x0 * x0 + b * x0 * x0 + c * x0 + d
    fd = 3 * a * x0 * x0 + 2 * b * x0 + c
    h = f / fd
    x = x0 - h                                              # 求得更接近方程根的x
值
    while abs(x - x0) >= 1e-5:
        x0 = x
        f = a * x0 * x0 * x0 + b * x0 * x0 + c * x0 + d
        fd = 3 * a * x0 * x0 + 2 * b * x0 + c
        h = f / fd
        x = x0 - h                                          # 求得更接近方程根的x
值

    return x


if __name__ == '__main__':
    print("请输入方程的系数：")
    # a,b,c,d代表所求方程的系数
    a, b, c, d = map(float, input().split())
    print("方程的参数为：" , a, b, c, d)
    # x用来记录求得的方程根
    x = solution(a, b, c, d)
    print("所求方程的根为x=%.6f"% x)
```

7. 运行结果

在PyCharm下运行程序，结果如图1.5所示。

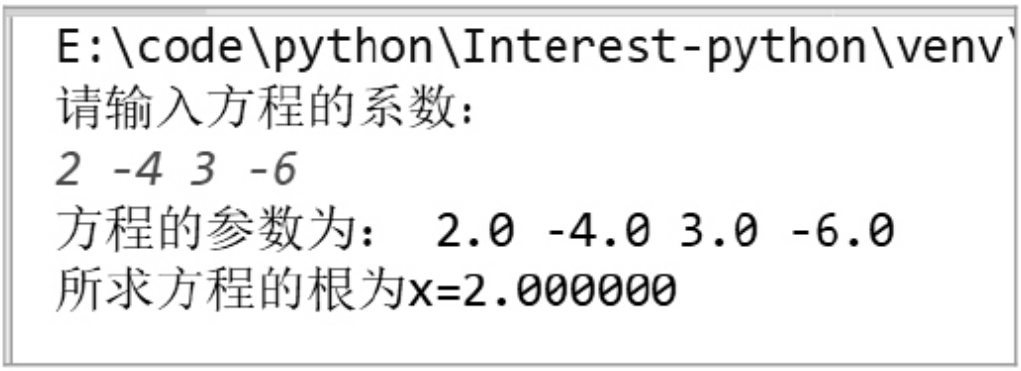

E:\code\python\Interest-python\venv
请输入方程的系数：
2 -4 3 -6
方程的参数为： 2.0 -4.0 3.0 -6.0
所求方程的根为x=2.000000

图1.5　运行结果

1.4　百钱百鸡

1．问题描述

中国古代数学家张丘建在他的《算经》中提出了一个著名的“百钱百鸡问题”：一只公鸡值五钱，一只母鸡值三钱，三只小鸡值一钱，现在要用百钱买百鸡，请问公鸡、母鸡、小鸡各多少只？

2．问题分析

用百钱如果只买公鸡，最多可以买20只，但题目要求买100只，由此可知，所买公鸡的数量肯定在0～20之间。同理，母鸡的数量在0～33之间。在此不妨把公鸡、母鸡和小鸡的数量分别设为cock、hen、chicken，则cock+hen+chicken=100，因此百钱买百鸡问题就转化成解不定方程组

$$\begin{cases} cock + hen + chicken = 100 \\ 5 \times cock + 3 \times hen + \dfrac{chicken}{3} = 100 \end{cases}$$

的问题了。

3．算法设计

对于不定方程组，我们可以利用穷举循环的方法来解决，也就是通过对未知数可变范围的穷举，验证方程在什么情况下成立，从而得到相应的解。因公鸡的取值范围是0～20，可用循环语句“for cock in range(0,21);”实现。钱的数量是固定的，要买的鸡的数量也是固定的，所以母鸡数量是受到公鸡数量限制的。同理，小鸡数量受到公鸡和母鸡数量的限制，因此我们可以利用三层循环的嵌

套来解决，第一层循环控制公鸡的数量，第二层控制母鸡的数量，最内层控制小鸡的数量。

4. 知识点补充

结构化程序设计包括三种基本结构：顺序结构、选择结构（分支结构）和循环结构（重复结构），利用这三种基本结构可以解决很多复杂问题。

• 顺序结构：一种简单的程序设计，按照程序中语句的顺序依次执行，每条语句都能被执行且只执行一次。

• 选择结构：包括简单选择和多分支选择结构，可根据条件，判断应该选择哪一条分支来执行相应的语句序列。简单选择结构采用简单或一般的if语句即可解决，对于复杂的选择结构可以使用嵌套的if…elif…else语句实现。

• 循环结构：可根据给定条件，判断是否需要重复执行某一相同程序段。

下面介绍Python语言的循环结构。

（1）while循环

while循环语法格式如下：

```
while 判断条件:
    执行语句
```

其中，判断条件可以是任何表达式，所有非空、非零的值都为True，当判断条件为False时，循环结束；执行语句可以是单条语句，也可以是一个语句块。

（2）for循环

for循环语法格式如下：

```
for iterating_var in sequence:
    statements(s),
```

其中，sequence是任意序列，如列表（数组）、字典或元组等，也可以通过range()函数产生一个整数列表，以完成计数循环；iterating_var是序列中需要遍历的元素；statements是待执行的语句块。

range()函数使用格式如下：

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

其中，start为可选参数，表示起始数；stop为终止数，如果range()函数只有一个参数n，则将产生一个0～(n-1)的整数列表，也就是循环n-1次；step为可选参数，表示循环步长，不写时，默认步长为1。

需要注意的是，while循环结构首先对while条件进行判断，当条件为True时，执行条件语句块；当执行完语句块时，再次判断while条件是否为True，若仍然为True，则继续执行语句块，直到条件为False时结束循环。

while循环结构的流程图如图1.6所示。

（3）for循环结构

for循环结构首先对for语句的条件判断，游标（序列都会有一个游标，游标一般从第0个位置开始）指向序列的第0个位置，也就是序列的第一个元素，判断序列sequence中是否有元素。如果有，就将这个元素赋值给iterating_var，然后执行循环体语句块，执行完成后，游标下移一位，再次判断该位置是否有元素；如果有，继续将该元素赋值给iterating_var，继续执行循环语句块；游标再往下移一位，一直循环下去，直到下一个位置没有元素时结束循环。

for循环结构的流程图如图1.7所示。

在循环语句中，循环体可以是由一个或多个语句构成的，当其中某个语句是循环语句时，即一个循环体中完整地包含了另外一个循环，就形成了循环嵌套结构，我们称之为多重循环，并且把这个循环语句称为外层循环语句，而把循环体中的循环语句称为内层循环语句。在理解多重循环语句时，只要把内层循环语句看作是外层循环语句的循环体的一部分就可以了。在程序执行时，外层循环语句与内层循环语句的关系，有点像钟表的时针与分针的关系，分针走了60格，时针才走1格。对于多重循环来说，只有内层循环语句执行到判断条件为假时，才返回到其上层循环语句继续执行。

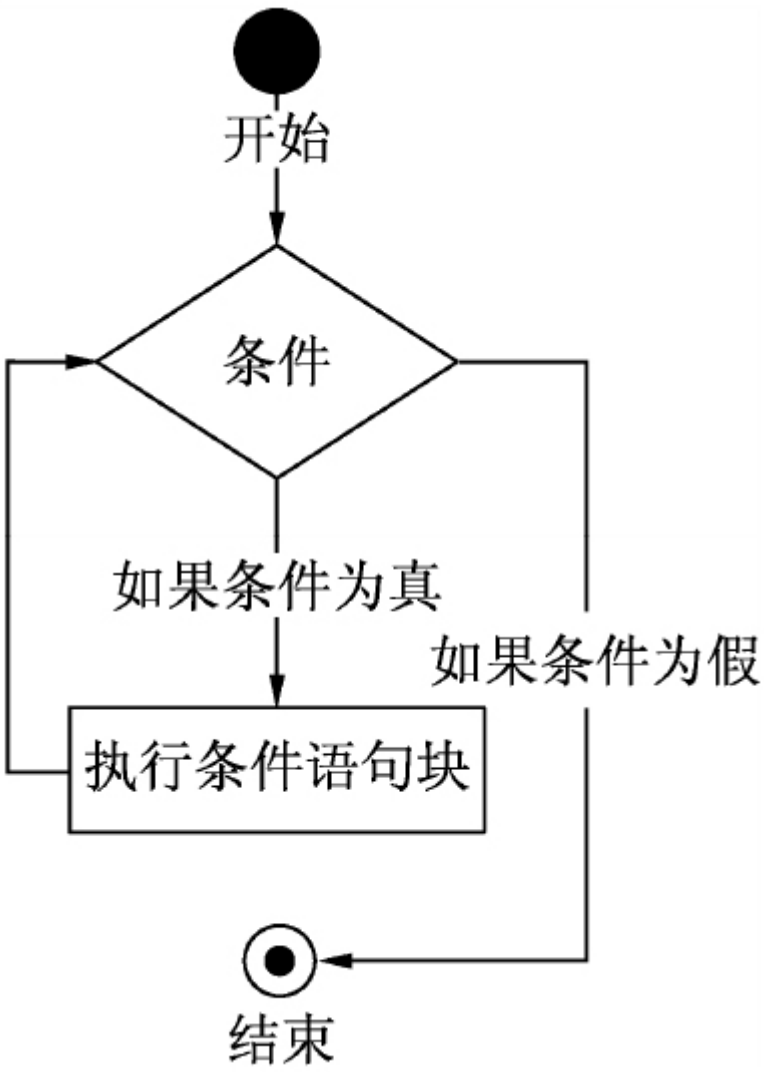


图1.6　while循环流程图

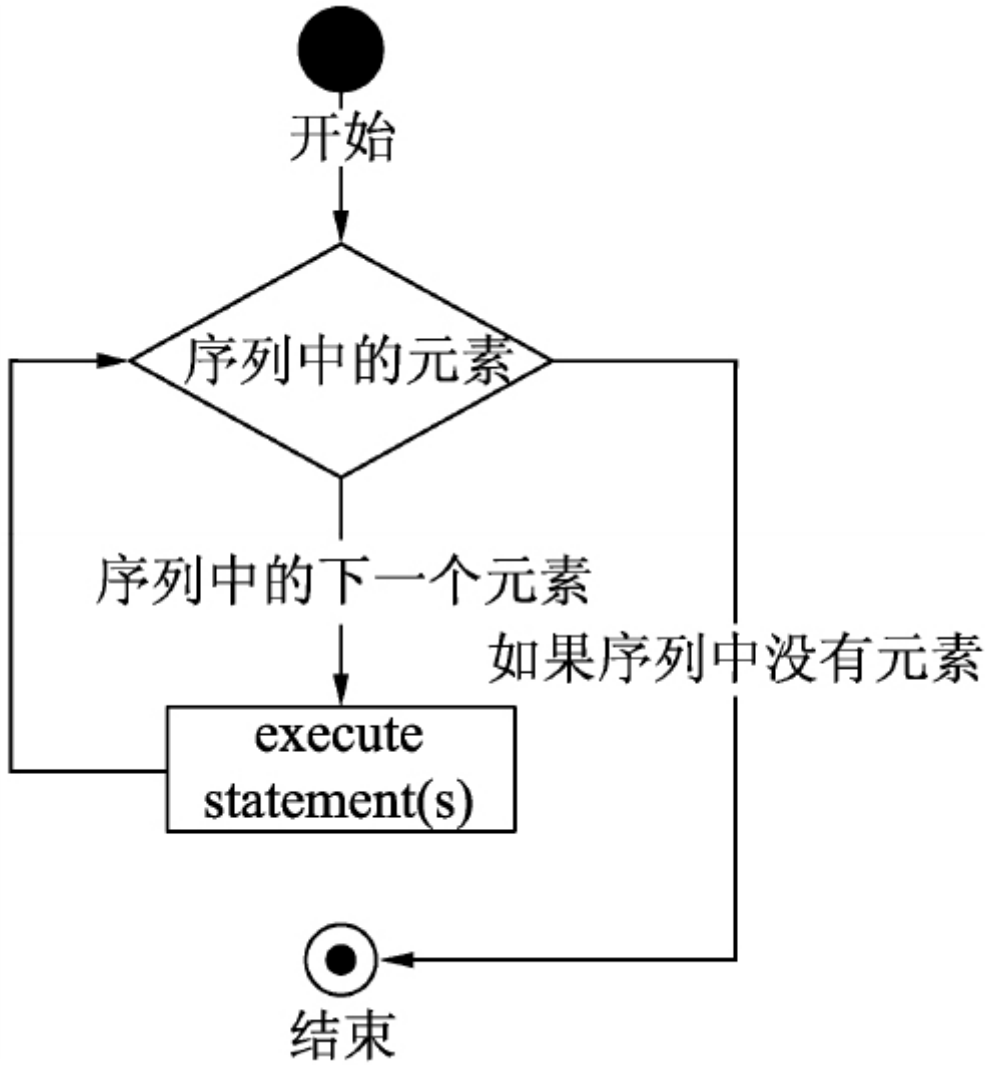


图1.7　for循环流程图

5．确定程序框架

在设计循环时首先要考虑循环的三要素，即循环变量的初值、循环的控制条件和使循环趋于结束的循环变量值的改变。

针对本题来说，每层循环的初值是0（即买的100只鸡中，可能没有公鸡，也可能没有母鸡或小鸡）；循环的控制条件是公鸡、母鸡和小鸡用百钱最多能够买到的数量［根据上面分析可知：公鸡最多20只，母鸡最多33只，小鸡最多100只（虽然百钱最多可以买到300只小鸡，但题目要求只买100只）］；穷举循环的特点就是把所

有情况都考虑到，因此每层循环执行一次，对应循环变量的值就要加1。

程序流程图如图1.8所示。

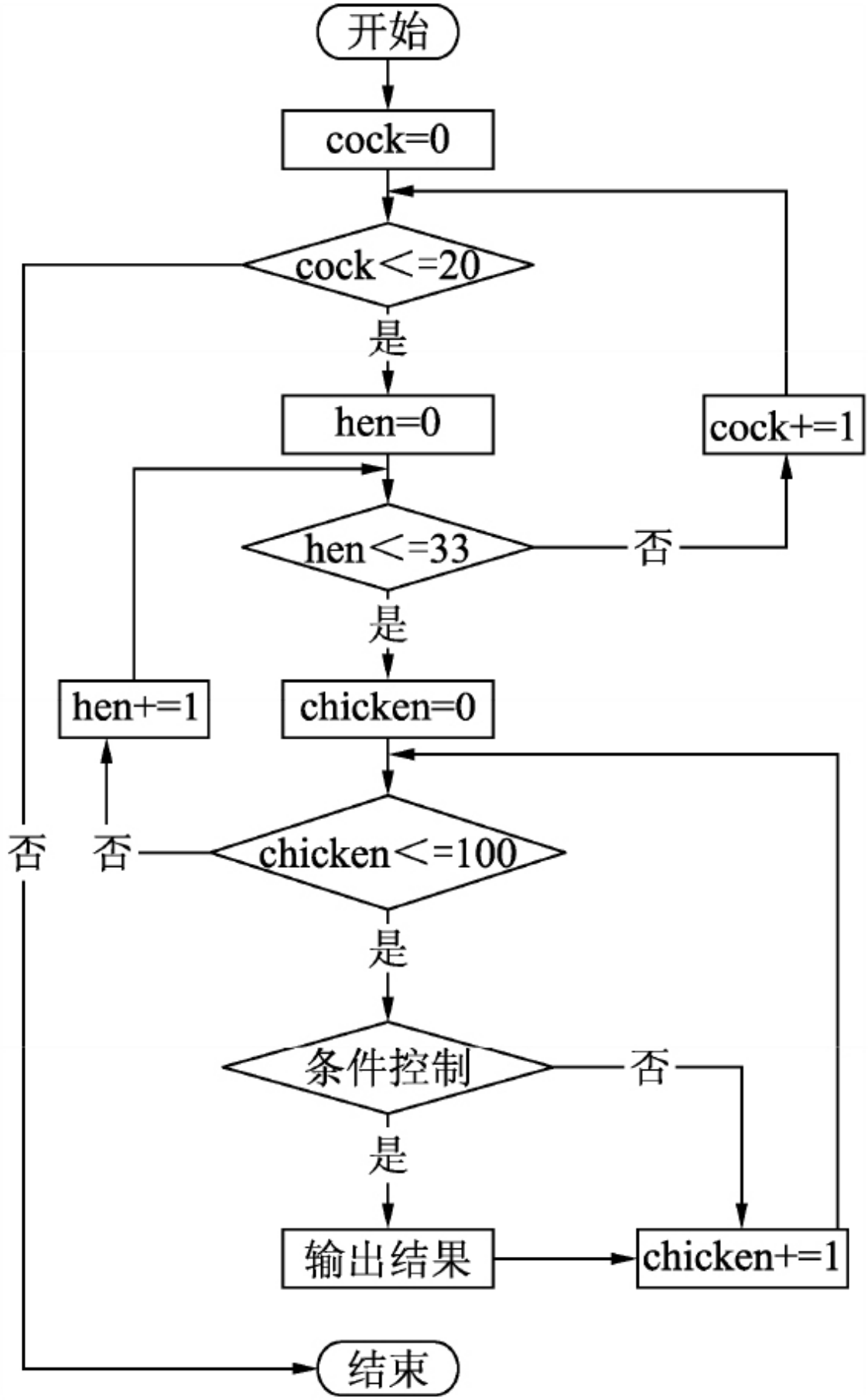

开始
cock=0
cock<=20
是
hen=0
cock+=1
hen<=33
否
是
hen+=1
chicken=0
否
否
chicken<=100
是
条件控制
否
是
输出结果
chicken+=1
结束

图1.8　程序流程图

根据流程图，构建程序框架如下：

```
cock = 0
while cock <= 20:
    # 内层循环控制母鸡数量取值范围为0～33
    hen = 0
    while hen <= 33:
        #内层循环控制小鸡数量取值范围为0～100
        chicken = 0
        while chicken <= 100:
            #条件控制
            print("cock=%2d,hen=%2d,chicken=%2d\n" %(cock,hen,chicken))
            chicken += 1
        hen += 1
    cock += 1
```

6．确定公鸡、母鸡和小鸡数量

根据这三层循环我们可以得到很多种方案，在这些方案中有些是不符合cock+hen+chicken=100并且5×cock+3×hen+chicken/3=100这两个条件的。因此结果输出之前我们要把合理的方案筛选出来，即如果结果满足cock+hen+chicken=100和5×cock+3×hen+chicken/3=100，则输出。很明显，控制条件即为语句if(5×cock+3×hen+chicken/3.0==100)and(cock+hen+chicken==100)。

注意：　在Python语言中，使用and关键字表示“逻辑与”；使用or关键字表示“逻辑或”；使用not关键字表示“逻辑非”。另外，运算符“/”表示除，x除以y，结果为带小数点的浮点数；而运算符“//”表示取整除，返回商的整数部分。

7．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 百钱百鸡问题
```

```
if __name__=="__main__":
    # cock表示公鸡数量，hen表示母鸡数量，chicken表示小鸡数量，总共100只
    # 外层循环控制公鸡数量取值范围为0～20
    cock = 0
    while cock <= 20:
        # 内层循环控制母鸡数量取值范围为0～33
        hen = 0
        while hen <= 33:
            #内层循环控制小鸡数量取值范围为0～100
            chicken = 0
            while chicken <= 100:
                # 条件控制
                if (5 * cock + 3 * hen + chicken / 3.0 ==100) and (cock + hen
+ chicken ==100):
                    print("cock=%2d,hen=%2d,chicken=%2d\n" %
(cock,hen,chicken))
                chicken += 1
            hen += 1
        cock += 1
```

注意： Python语言的缩进格式。

8. 运行结果

在PyCharm下运行程序，结果如图1.9所示。

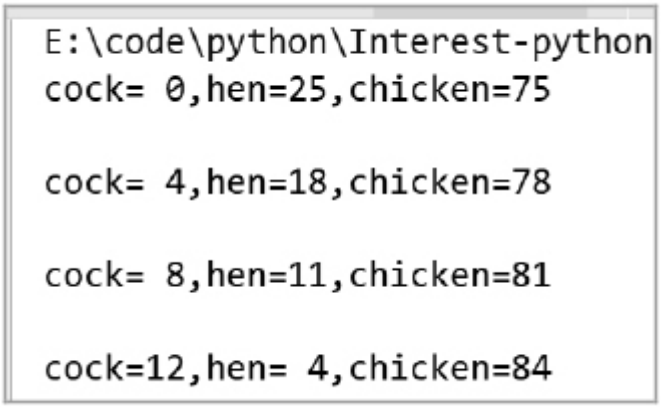


图1.9 运行结果

9. 问题拓展

以上算法需要穷举尝试21×34×101=72 114次，算法的效率显然太低了。对于这类求解不定方程的问题，各层循环的控制变量直接与方程的未知数相关，并且采用对未知数的取值范围穷举和组合的方法得到全部的解。对于本题来说，公鸡的数量确定后，小鸡的数量就固定为100-cock-hen，无须再进行穷举了，此时约束条件只

有一个，即5×cock+3×hen+chicken/3=100，这样我们利用两重循环即可求解本题，代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 百钱百鸡问题

if __name__=="__main__":
    # 外层循环控制公鸡数量取整范围为0～20
    cock = 0
    while cock <= 20:
        # 内层循环控制母鸡数量取值范围为0～30
        hen = 0
        while hen <= 33:
            # 小鸡的数量
            chicken = 100 - cock - hen
            if 5 * cock + 3 * hen + chicken / 3.0 == 100:
                print("cock=%2d,hen=%2d,chicken=%2d\n" %(cock, hen, chicken))
            hen+=1
        cock+=1
```

此算法只需穷举21×34=714次，实现时约束条件又限定chicken能被3整除时才会判断“5×cock+3×hen+chicken/3.0=100”，这样便省去了chicken不能整除3时的算术计算和条件判断，进一步提高了算法的效率。

1.5　借书方案知多少

1．问题描述

小明有5本新书，要借给A、B、C三位小朋友，若每人每次只能借1本，则可以有多少种不同的借法？

2．问题分析

本题属于数学中常见的排列组合问题，即求从5个数中取3个不同数的排列组合的总数。我们可以将5本书进行1～5编号，A、B、C三个人每次都可以从5本书中任选1本，即每人都有5种选择，由于1本书不可能同时借给一个以上的人，因此只要这三个人所选书的编号不同，则即为一次有效的借阅方法。

3．算法设计

对于每个人所选书号，我们可以采用穷举循环来实现，即从每个人可选书号（1、2、3、4、5）的范围内进行穷举，从而得到可行的结果。对于第一个人的选择，可以用循环将其列出，即for a in range(1,6)。同理，对于第二个人、第三个人可以用同样的方法。由于一本书只能借给一个人，故第二个人的选择会受到第一个人的限制，最后一个人的选择会受到第二个人的限制，即后面的选择都是在前面选择的前提下进行的，所以可采用循环的嵌套来解决问题。

利用循环解决问题的时候，找到循环的三要素，即循环变量的初值、循环的控制条件和使循环趋于结束的循环变量值的改变是进行编程的关键。读者可参照1.4节的例子来找一下本题中所对应的循环三要素。本题的输出结果有一个条件限制：三个人所选书号各不相同。这在输出语句前只要用一个if语句“if a != b and a != c and c != b”进行判断即可。

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 借书方案知多少

if __name__=="__main__":
    #A、B、C三位小朋友，5本书，每人每次只能借一本
    #用a、b、c分别表示三人所选图书的编号
    i = 0                                                                    # i
表示有效借阅次数
    print("A,B,C三人所选书号分别为：")
    #用来控制A借阅图书的编号
    for a in range(1, 6):
        #用来控制B借阅图书的编号
        for b in range(1, 6):
            #用来控制C借阅图书的编号
            for c in range(1, 6):
                if a != b and a != c and c != b:
                    print("A:%2d  B:%2d  C:%2d    " %(a, b, c) ,  end='')
                    i += 1
                    if i % 4 == 0:
                        print()                                  #换行
    print("共有%d种有效借阅方法" %i)
```

5. 运行结果

在PyCharm下运行程序，结果如图1.10所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/code/python/Interest
A,B,C三人所选书号分别为：
A: 1  B: 2  C: 3    A: 1  B: 2  C: 4    A: 1  B: 2  C: 5    A: 1  B: 3  C: 2
A: 1  B: 3  C: 4    A: 1  B: 3  C: 5    A: 1  B: 4  C: 2    A: 1  B: 4  C: 3
A: 1  B: 4  C: 5    A: 1  B: 5  C: 2    A: 1  B: 5  C: 3    A: 1  B: 5  C: 4
A: 2  B: 1  C: 3    A: 2  B: 1  C: 4    A: 2  B: 1  C: 5    A: 2  B: 3  C: 1
A: 2  B: 3  C: 4    A: 2  B: 3  C: 5    A: 2  B: 4  C: 1    A: 2  B: 4  C: 3
A: 2  B: 4  C: 5    A: 2  B: 5  C: 1    A: 2  B: 5  C: 3    A: 2  B: 5  C: 4
A: 3  B: 1  C: 2    A: 3  B: 1  C: 4    A: 3  B: 1  C: 5    A: 3  B: 2  C: 1
A: 3  B: 2  C: 4    A: 3  B: 2  C: 5    A: 3  B: 4  C: 1    A: 3  B: 4  C: 2
A: 3  B: 4  C: 5    A: 3  B: 5  C: 1    A: 3  B: 5  C: 2    A: 3  B: 5  C: 4
A: 4  B: 1  C: 2    A: 4  B: 1  C: 3    A: 4  B: 1  C: 5    A: 4  B: 2  C: 1
A: 4  B: 2  C: 3    A: 4  B: 2  C: 5    A: 4  B: 3  C: 1    A: 4  B: 3  C: 2
A: 4  B: 3  C: 5    A: 4  B: 5  C: 1    A: 4  B: 5  C: 2    A: 4  B: 5  C: 3
A: 5  B: 1  C: 2    A: 5  B: 1  C: 3    A: 5  B: 1  C: 4    A: 5  B: 2  C: 1
A: 5  B: 2  C: 3    A: 5  B: 2  C: 4    A: 5  B: 3  C: 1    A: 5  B: 3  C: 2
A: 5  B: 3  C: 4    A: 5  B: 4  C: 1    A: 5  B: 4  C: 2    A: 5  B: 4  C: 3
共有60种有效借阅方法
```

图1.10　运行结果

6. 问题拓展

如果前两个人所选书号相同，那么无论第三个人所选书号与前两人相同与否都是无效的借阅方法。因此在执行第三个循环之前可先判定前两人的编号是否相同，进而提高程序效率。实现代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 借书方案知多少

if __name__=="__main__":
    # A、B、C三位小朋友，5本书，每人每次只能借一本
    # 用a、b、c分别表示三人所选图书的编号
    i = 0                                                                    # i
表示有效借阅次数
    print("A,B,C三人所选书号分别为：")
    a = 1
    while a <= 5:
        b = 1
        while b <= 5:
            c = 1
            while c <= 5 and a != b:
                if a != c and b != c:                # 控制有效借阅组合
                    print("A:%2d  B:%2d  C:%2d    " % (a, b, c), end='')
                    i += 1
                    if i % 4 == 0:
                        print()                                  # 换行
                c += 1
            b += 1
        a += 1

    print("共有%d种有效借阅方法" % i)
```

对原程序稍做修改之后，在长度上虽没有改进，仍有三层循环，但是在程序的执行效率上有了很大的提高。对于原程序中的第三层循环来说，不管a和b的取值是否相同，循环都要重复进行5次；而修改后的程序在进入循环体之前首先判断a和b的取值，如果两者取值相同，则内层循环无须重复执行5次便可结束。本题的数据较小，在处理数据很大的问题时使用该方法效率的提高会更加明显。

1.6　打鱼还是晒网

1．问题描述

中国有句俗语叫“三天打鱼两天晒网”。某人从1990年1月1日起便开始“三天打鱼两天晒网”，问这个人在以后的某一天中是“打鱼”还是“晒网”。

2．问题分析

根据题意可以将解题过程分为以下三步：

1）计算从1990年1月1日开始至指定日期共有多少天。

2）由于“打鱼”和“晒网”的周期为5天，所以将计算出的天数用5去除。

3）根据余数判断他是在“打鱼”还是在“晒网”。

若余数为1，2，3，则他是在“打鱼”，否则是在“晒网”。

3．算法设计

本题目使用的算法为数值计算算法，要利用循环求出指定日期距1990年1月1日的天数，并考虑到循环过程中的闰年情况，闰年二月为29天，平年二月为28天。判断闰年的方法可以用伪语句描述如下：

如果能被4整除并且不能被100整除或者能被400整除，则该年是闰年，否则不是闰年。

提示：　在Python语言中判断能否整除可以使用求余运算符“%”。

4．确定程序框架

程序流程图如图1.11所示。

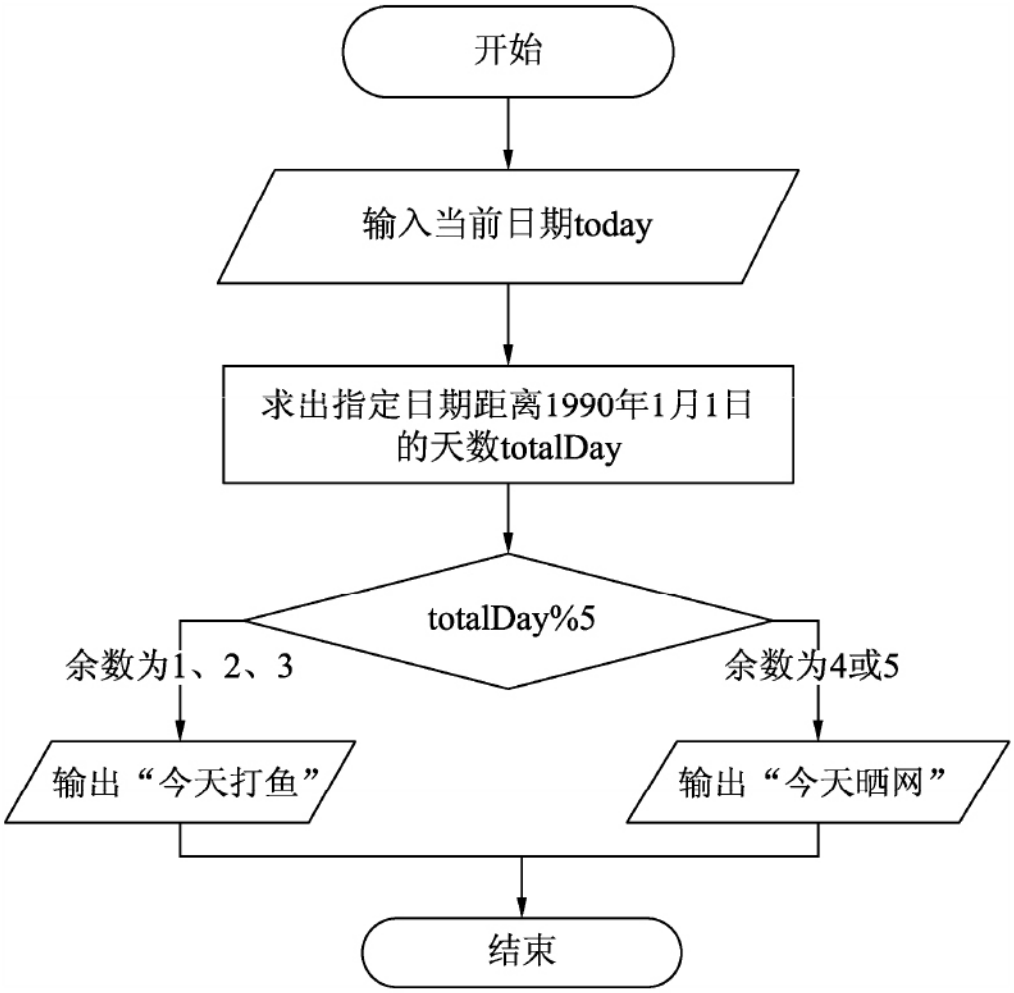


图1.11　程序流程图

根据流程，构建程序框架如下：

```
if __name__ == '__main__':
    print("please input 指定日期 包括年,月,日 如:1999 1 31")
    year, month, day = [int(i) for i in input().split()]
    # 定义一个日期字典
    today = {'year': year, 'month': month, 'day': day}

    totalDay = countDay(today)                # 求出指定日期距离1990年1月1日的天数
    print("%d年%d月%d日与1990年1月1日相差 %d 天" % (year, month, day,
totalDay))

    # 天数 % 5，判断输出打鱼还是晒网
    result = totalDay % 5
    if result > 0 and result < 4:
        print("今天打鱼")
    else:
        print("今天晒网")
```

5. 求出指定日期距离1990年1月1日的天数

这里为整个算法的核心部分，经过分析可以得到指定日期距离1990年1月1日的天数totalDay=1990年至指定年的前一年共有多少天+指定年中到指定日期的天数。由于每月天数不同，可以设置一个月份数组int perMonth[13]存放每月的天数。程序利用年份作为循环变量，要判断指定年份之前的每一年是否为闰年，若为闰年则执行totalDay=totalDay+366，否则执行totalDay=totalDay+365；对于指定年份，也要判定是否为闰年，然后根据月份数，将每月的天数累加到totalDay中。

perMonth数组的初始化设置如表1.2所示。该设置含有13个元素，perMonth[0]元素并不使用，原因在于这种设置可以使数组下标和月份对应，便于编程设置循环变量，数组中二月天数初始设置为28，如果当前年份为闰年，则需要执行perMonth[2]+=1操作。

表1.2　perMonth数组初始化

perMonth[0]	perMonth[1] 一月天数	perMonth[2] 二月天数	perMonth[3] 三月天数	perMonth[4] 四月天数	perMonth[5] 五月天数	perMonth[6] 六月天数
0	31	28	31	30	31	30
perMonth[7] 七月天数	perMonth[8] 八月天数	perMonth[9] 九月天数	perMonth[10] 十月天数	perMonth[11] 十一月天数	perMonth[12] 十二月天数	
31	31	30	31	30	31	

提炼功能模块，我们设计一个函数countDay(currentDay)来实现求总天数的功能，设计一个函数runYear(year)来判断是否为闰年。判断是否为闰年的函数runYear(year)的实现如下：

```
# 判断是否为闰年。是，则返回1，否，则返回0
def runYear(year):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):  # 是闰年
        return 1
    else:
        return 0
```

求总天数的函数countDay(currentDay)的实现如下：

```
# 计算指定日期距离1990年1月1日的天数
def countDay(currentDay):
    # 每月天数数组
```

```
    perMonth = [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30]
    totalDay = 0
    year = 1990
    while year < currentDay['year']:      # 求出指定日期之前的每一年的天数之和
        if runYear(year) == 1:                          # 判断是否为闰年
            totalDay = totalDay + 366
        else:
            totalDay = totalDay + 365

        year += 1

    # 如果为闰年，则二月份为29天
    if runYear(currentDay['year']) == 1:
        perMonth[2] += 1

    i = 0
    while i < currentDay['month']:                # 将本年内的天数累加到totalDay中
        totalDay += perMonth[i]
        i += 1

    totalDay += currentDay['day']            # 将本月内的天数累加到totalDay中

    return totalDay
```

---

6. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 打鱼还是晒网

# 判断是否为闰年。是，则返回1；否，则返回0
def runYear(year):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):  # 是闰年
        return 1
    else:
        return 0


# 计算指定日期距离1990年1月1日的天数
def countDay(currentDay):
    # 每月天数数组
    perMonth = [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30]
    totalDay = 0
    year = 1990
    while year < currentDay['year']:      # 求出指定日期之前的每一年的天数之和
        if runYear(year) == 1:                          # 判断是否为闰年
            totalDay = totalDay + 366
        else:
            totalDay = totalDay + 365

        year += 1

    # 如果为闰年，则二月份为29天
    if runYear(currentDay['year']) == 1:
        perMonth[2] += 1

    i = 0
    while i < currentDay['month']:                # 将本年内的天数累加到totalDay中
        totalDay += perMonth[i]
        i += 1

    totalDay += currentDay['day']            # 将本月内的天数累加到totalDay中
```

```
    return totalDay


if __name__ == '__main__':
    while True:
        print("please input 指定日期 包括年,月,日 如:1999 1 31")
        year, month, day = [int(i) for i in input().split()]
        # 定义一个日期字典
        today = {'year': year, 'month': month, 'day': day}

        totalDay = countDay(today)           # 求出指定日期距离1990年1月1日的天数
        print("%d年%d月%d日与1990年1月1日相差 %d 天" % (year, month, day,
totalDay))

        # 天数 % 5，判断输出打鱼还是晒网
        result = totalDay % 5
        if result > 0 and result < 4:
            print("今天打鱼")
        else:
            print("今天晒网")
```

7. 运行结果

在PyCharm下运行程序，结果如图1.12所示。

```
E:\code\python\Interest-python\venv\Scripts\python
please input 指定日期 包括年,月,日 如:1999 1 31
2010 1 31
2010年1月31日与1990年1月1日相差 7336 天
今天打鱼
please input 指定日期 包括年,月,日 如:1999 1 31
2010 10 25
2010年10月25日与1990年1月1日相差 7603 天
今天打鱼
please input 指定日期 包括年,月,日 如:1999 1 31
2012 10 25
2012年10月25日与1990年1月1日相差 8334 天
今天晒网
```

图1.12　运行结果

8. 补充知识点

（1）函数

函数是指用于实现某个功能的一系列语句的组合。在定义函数时，需要指定函数的名称及所需参数，并编写一系列程序语句（函数体），之后可以使用这个函数名称来直接调用它。

在Python中定义了许多内置函数，可以直接使用，在后续的章节中将会讲到。同时Python也支持自定义函数，可以根据自己的需要，自己定义一个函数来实现某个功能。自定义函数语法如下：

```
def 函数名(参数列表):
    函数体
```

规则如下：

• Python使用关键字def定义函数，也就是函数代码块以def关键字开头，后面跟上函数标识符名称和圆括号“()”。

• 在圆括号“()”中定义函数所需的参数，换句话说就是所有传入的参数和变量都必须放到圆括号中。

• 函数内容以冒号开始，注意函数体语句的缩进格式。

• 可以使用return关键字来向函数调用方返回一个结果值，“return[表达式]”用于结束函数，不带表达式的return语句相当于返回None。

• 函数名的定义规则：必须以字母或下划线开头，如main函数(__name__=="__main__")，不能以数字开头。同时不能把Python自带的关键字定义为函数的名称。

（2）字典

在代码中，我们定义了一个日期字典，如下：

```
today = {'year': year, 'month': month, 'day': day}
```

字典是Python语言中的一种数据结构，以{}包含数据集，定义语法如下：

```
字典名 = {key1: value1, key2: value2}
```

字典内的元素是由键/值（key/value）对组成的，键和值之间使用冒号（:）分割，每个键/值对之间使用逗号（,）分割，每个键key必须是唯一的，值可以不唯一。字典的值可以取任何数据类型，例如字符串、数字或元组，而键必须是不可变的。

字典是无序的，通常情况下，我们是使用字典的键来访问其成员的，它是采用哈希原理实现的。一个空字典不包含任何元素，仅使用一个大括号定义，如{}。

下面介绍字典的常用操作方法。

• dict.clear()：清空字典，也就是删除字典中的所有元素。

• dict.copy()：复制字典，返回一个具有相同键/值对的新字典，是浅复制。

• dict.fromkeys(seq[,value])：创建一个新字典，以序列seq中的元素作为字典的键，value为字典所有键对应的初始值。

• dict.get(key,default=None)：获取键key的value值，如果值不存在，就返回默认值。

• dict.items()：以列表的形式返回可遍历的（键/值）元组数组。

• dict.keys()：以列表的形式，返回字典中的所有键。

• dict.pop(key)：删除键key。

• dict.update(dict2)：更新字典元素，用于把字典dict2的键/值对更新到dict中。

- dict.values()：以列表的形式返回字典中的所有值。

1.7　最佳存款方案

1．问题描述

假设银行一年整存零取的月息为0.63%。现在某人手中有一笔钱，他打算在今后5年中的每年年底取出1000元，到第5年时刚好取完，请算出他存钱时应存入多少。

2．问题分析

根据题意，可以从第5年向前推算。已知“在今后5年中的每年年底取出1000元，这样到第5年的时候刚好可以取完”，因此，第5年年底会取出1000元，则可以计算出第5年年初在银行中所存的钱数为：

第5年年初存款数=1000/（1+12×0.0063）

据此推算出第4年、第3年直至第1年年初的银行存款数如下：

第4年年初存款数=（第5年年初存款数+1000）/（1+12×0.0063）

第3年年初存款数=（第4年年初存款数+1000）/（1+12×0.0063）

第2年年初存款数=（第3年年初存款数+1000）/（1+12×0.0063）

第1年年初存款数=（第2年年初存款数+1000）/（1+12×0.0063）

将推导过程用表格表示出来，如表1.3所示。

表1.3　年初存款参照表

年 初 存 款	公　　式
第5年年初存款	1000/（1+12×0.0063）
第4年年初存款	（1000+第5年年初存款）/（1+12×0.0063）
第3年年初存款	（1000+第4年年初存款）/（1+12×0.0063）
第2年年初存款	（1000+第3年年初存款）/（1+12×0.0063）
第1年年初存款	（1000+第2年年初存款）/（1+12×0.0063）

3. 算法设计

根据上述分析，从第5年年初开始向前递推就可求出这个人应该在银行中存钱的钱数。因此可以使用for循环语句，循环4次，每次循环都在上一次的基础上加上1000，再除以（1+12×0.0063）。

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 最佳存款方案

if __name__=="__main__":
    i = 0
    money = 0.0
    while i < 5:
        money = (money + 1000)/(1 + 0.0063 * 12)
        i += 1
    print("应该存入钱数为：%0.2f" %money)                                    # 结果保留两位小
数
```

5. 运行结果

在PyCharm下运行程序，结果如图1.13所示。由于在程序中控制了输出结果的小数位数为两位，因此最后的计算结果为4039.44。

图1.13　运行结果

1.8　冒泡排序

1．问题描述

对N个整数（数据由键盘输入）进行升序排列。

2．问题分析

对于N个类型相同的数，我们可利用数组进行存储。冒泡排序是利用两个相邻元素之间进行比较交换的过程将一个无序表变成有序表。

冒泡排序的思想是：首先，从表头开始往后扫描数组，在扫描过程中逐对比较相邻两个元素的大小。若相邻两个元素中，前面的元素大于后面的元素，则将它们互换，称之为消去了一个逆序。在扫描过程中，不断地将两相邻元素中的大者往后移动，最后就将数组中的最大者换到了表的最后，这正是数组中最大元素应有的位置。然后，在剩下的数组元素中（n-1个元素）重复上面的过程，将次小元素放到倒数第2个位置。不断重复上述过程，直到剩下的数组元素为0为止，此时的数组就变为了有序。假设数组元素的个数为n，在最坏的情况下需要的比较总次数为：((n-1)+(n-2)+…+2+1)=n(n-1)/2。

3．算法设计

冒泡排序的过程我们可以用示意图简单地表示，如图1.14所示。从整个排序过程中寻找规律，可知n个元素只需比较n-1次即可。假设一个数组中有7个元素，现在对这7个元素进行排序，只需比较6次即可得到所要的有序序列。示意图中最后加粗的数字即为经过一次交换位置固定的数字。

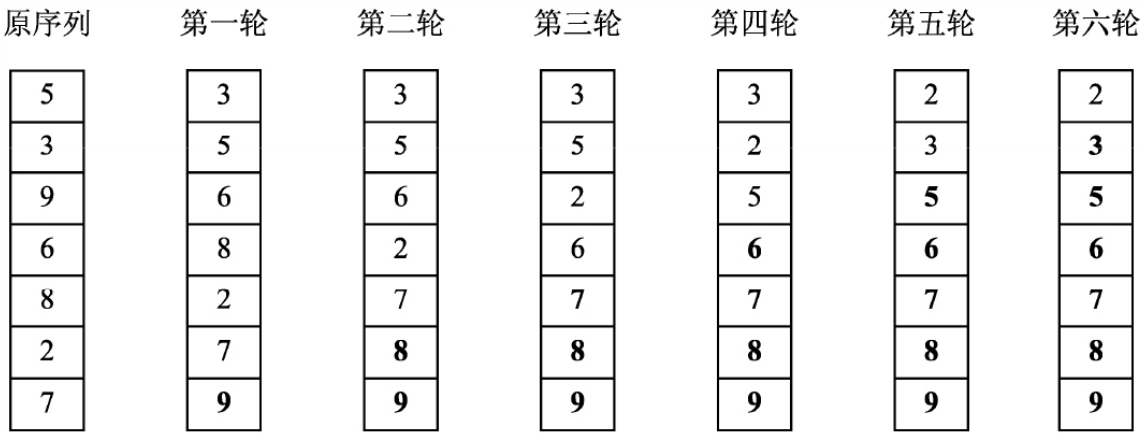


图1.14　冒泡排序示意

数组名用a表示，数组下标用j表示，则数组中相邻两个元素的下标可表示为a[j]、a[j+1]或a[j-1]、a[j]。在利用数组解决问题时需要注意数组下标不要越界，假如定义一个整形数组int a[n]，则数组元素下标的取值范围是0～n-1，下标小于0或者大于n-1都视为下标越界。如果相邻元素采用a[j]、a[j+1]表示，则下标取值范围是0～n-2，如果采用a[j-1]、a[j]表示，下标取值范围则是1～n-1，所以读者在进行编程时一定要注数组下标越界的问题。

数组元素互换也是经常遇到的一类题型，一般这种情况下我们需要借助一个中间变量才可以完成，对于许多初学者来说经常犯的一个错误是对两个元素直接相互赋值，而不借助中间变量。我们先来看一个生活中的例子，在蓝墨水瓶中装有蓝墨水，红墨水瓶中装有红墨水，现在我们要把蓝墨水放到红墨水瓶中，红墨水放到蓝墨水瓶中。做法是先找一个白色空瓶（作用相当于程序中的中间变量），首先将蓝墨水倒入白色空瓶（t=a[i]或t=a[i+1]），接着将红墨水倒入蓝墨水瓶（a[i]=a[i+1]或a[i+1]=a[i]），最后将白瓶中的蓝墨水倒入红墨水瓶（a[i+1]=t或a[i]=t），经过这三步就完成了红墨水与蓝墨水的互换。如果不借助白色空瓶，直接把蓝墨水倒入红墨水瓶或把红墨水倒入蓝墨水瓶，这样必将破坏原来所存储的内容。

第一次的交换过程可以用简单的程序段进行表示，代码如下：

```
for j in range(0, n-1):
if a[j] > a[j+1]:
        t = a[j]                                    # 使用变量t暂存
        a[j] = a[j+1]
        a[j+1] = t
```

第二次的交换过程（最后一个元素经过第一轮比较已经确定，不需要再次进行比较）可表示为：

```
for j in range(0, n-2):
    if a[j] > a[j+1]:
        t = a[j]                                    # 使用变量t暂存
        a[j] = a[j + 1]
        a[j + 1] = t
```

第三次的交换过程（最后两个元素已经确定，不需要再次进行比较）可表示为：

```
for j in range(0, n-3):
    if a[j] > a[j+1]:
        t = a[j]                                    # 使用变量t暂存
        a[j] = a[j + 1]
        a[j + 1] = t
```

由上面的程序段可以发现，第一次比较的判定条件为j<n-1，第二次为j<n-2，第三次为j<n-3，以此类推，第i次的循环判定条件必为j<n-i。在编程过程中我们可以用两层循环来控制，第一层循环控制交换的轮数，第二层循环控制每轮需要交换的次数。

4. 完整的程序

程序流程图如图1.15所示。

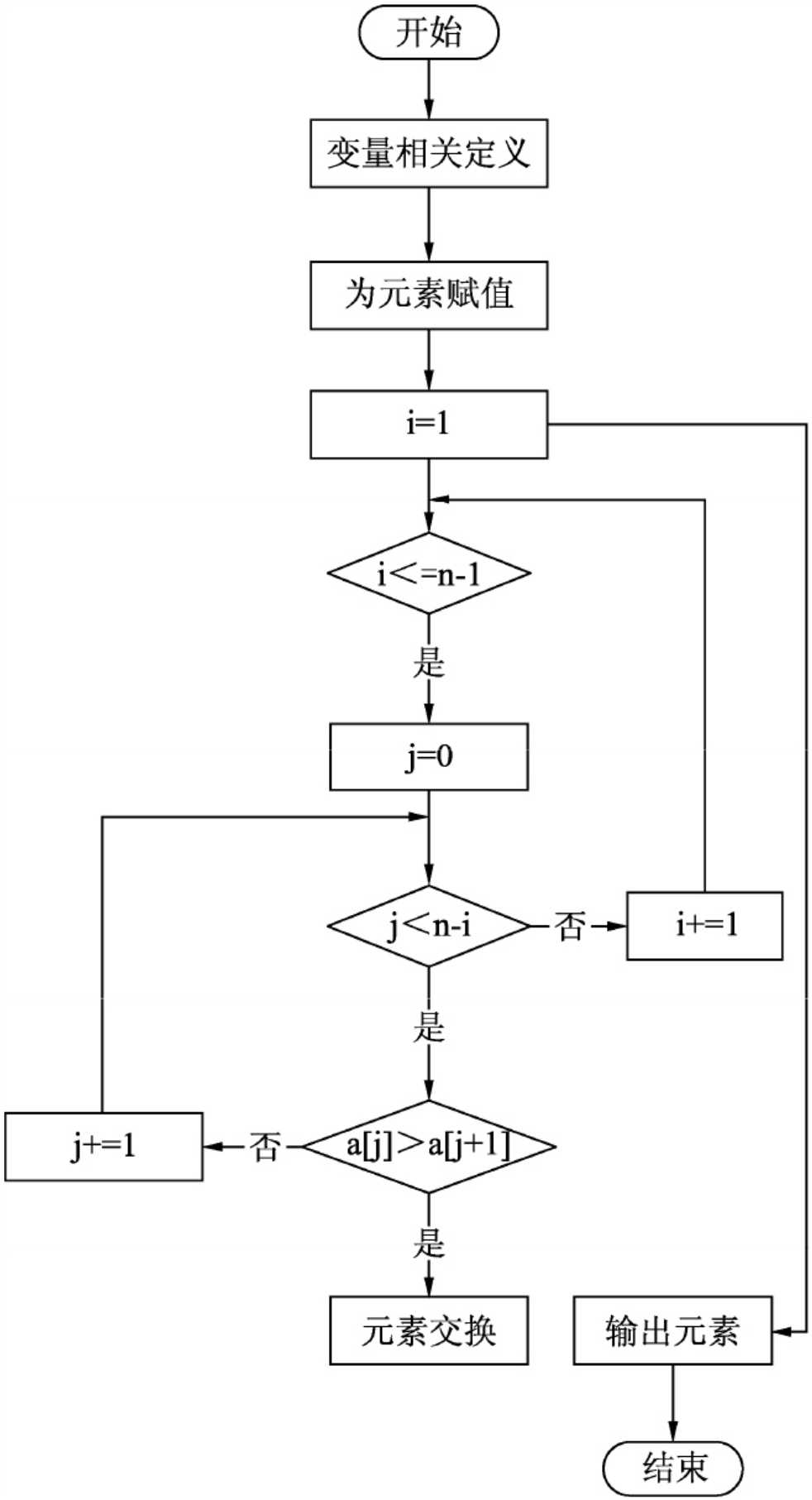

开始
变量相关定义
为元素赋值
i=1
i<=n-1
是
j=0
j<n-i
否
i+=1
是
a[j]>a[j+1]
否
j+=1
是
元素交换
输出元素
结束

图1.15　程序流程图

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 冒泡排序

def bubbleSort(a):
    # 首先获取列表list的总长度，为之后循环比较做准备
    n = len(a)
    # 进行 n-1 次比较，控制比较的轮数
    i = 1
    while i <= n-1:
        # 控制每轮比较的次数
        j = 0
        while j < n-i:
            # 交换
            if a[j] > a[j+1]:
                t = a[j]
                a[j] = a[j+1]
                a[j+1] = t
            j += 1
        i += 1
    # 打印每一轮交换后的列表
    for a1 in a:
        print(a1, end=" ")


if __name__=="__main__":
    print("请为列表元素赋初值，列表末尾不能有空格：")
    x = input()
    a = x.split(" ")                                    # 以空格方式分割每个元
素
    for i in range(0, len(a)):                  # 输入多个值
        a[i] = int(a[i])
    print("你输入的列表元素为：\n", a)
    print("经过交换后的数组元素为：")
    bubbleSort(a)
```

5. 运行结果

在PyCharm下运行程序，屏幕上提示“请为数组元素赋初值，列表末尾不能有空格：”，输入两组不同的初值，运行结果如图1.16所示。

```
E:\code\python\Interest-python\venv\
请为列表元素赋初值，列表末尾不能有空格：
5 3 9 6 8 2 7
你输入的列表元素为：
 [5, 3, 9, 6, 8, 2, 7]
经过交换后的数组元素为：
2 3 5 6 7 8 9
Process finished with exit code 0
```

a）运行结果 1

```
E:\code\python\Interest-python\venv\Scripts\python.exe E
请为列表元素赋初值，列表末尾不能有空格：
8 7 9 5 3 10 11 20 6 11 15 21 18 22 23
你输入的列表元素为：
 [8, 7, 9, 5, 3, 10, 11, 20, 6, 11, 15, 21, 18, 22, 23]
经过交换后的数组元素为：
3 5 6 7 8 9 10 11 11 15 18 20 21 22 23
Process finished with exit code 0
```

b）运行结果 2

图1.16　运行结果

6. 问题拓展

常用的排序方法除了上述的冒泡法外，还有选择排序、插入排序、快速排序、堆排序等，下面简单介绍选择排序。

选择排序的思想是：扫描整个线性表，第一轮比较拿数组中的第一个元素与其他元素进行比较，遇到比第一个元素小的元素则进行交换，再拿着交换之后的第一个元素接着从上次比较的位置与后面的元素进行比较，直到扫描到线性表的最后，从中选出最小的元素，将它交换到表的最前面（这是它应有的位置）；第二轮比较的时候从第二个元素开始，依次与第三个、第四个直到最后一个进行比较，在比较过程中有比第二个元素小的元素则进行交换，接着与后面的元素比较；对剩下的子表采用同样的方法，直到子表为空。在最坏的情况下，需要比较n(n-1) / 2次。

完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 选择排序

def selectionSort(a):
    # 求出列表的长度
    n = len(a)

    for i in range(0, n-1):
        for j in range(i+1, n):
            #交换
            if a[j] < a[i]:
                t = a[i]
                a[i] = a[j]
                a[j] = t
    for i in a:
        print(i, end=" ")


if __name__=="__main__":
    print("请为列表元素赋初值，列表末尾不能有空格：")
    x = input()
    a = x.split(" ")                                    # 以空格方式分割每个元
素
    for i in range(0, len(a)):                  # 输入多个值
        a[i] = int(a[i])
    print("你输入的列表元素为：\n", a)
    print("经过交换后的数组元素为：")
    selectionSort(a)
```

不同排序法的效率是不同的，不同需求情况下可选择不同的方法。其他几种排序方法的原理有兴趣的读者可参阅数据结构的相关内容。

1.9　折半查找

1．问题描述

N个有序整数数列已放在一维数组中，利用二分查找法查找整数m在数组中的位置。若找到，则输出其下标值；反之，则输出“Not be found!”。

2．问题分析

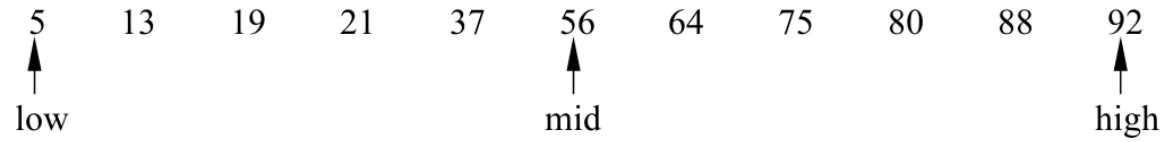
二分查找法（也叫折半查找）其本质是分治算法的一种，所谓分治算法指的是分而治之，即将较大规模的问题分解成几个较小规模的问题，这些子问题互相独立且与原问题相同，通过对较小规模问题的求解达到对整个问题的求解。当我们将问题分解成两个较小问题求解时的分治方法称为二分法。需要注意的是，二分查找法只适用于有序序列。

二分查找法的基本算法是：每次查找前先确定数组中待查的范围。假设指针low和high（low<high）分别指示待查范围的下界和上界，指针mid指示待查范围的中间位置，即mid=（low+high）/2，把m与中间位置（mid）上元素的值进行比较。如果m的值大于中间位置上元素的值，则下一次的查找范围放在中间位置之后的元素中；反之，下一次的查找范围放在中间位置之前的元素中。直到low>high，查找结束。

3．算法设计

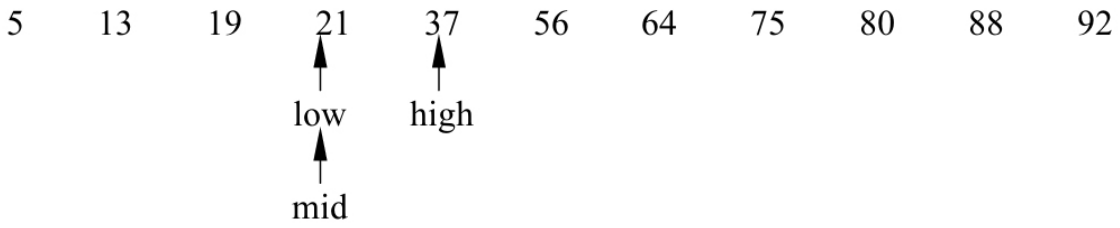
N个有序数因存放在数组中，根据数组下标的取值范围可知指针low和high的初值分别为0和N-1。除了3个指针变量low、high、mid之外，还需要一个变量假设为k来记录下标，利用变量k的值来判断整数m是否在所给出的数组中。下面我们用示意图来表示二分查找的过程。

假设一维数组中存储的有序数列为：5 13 19 21 37 56 64 75 80 88 92，要查找的整数m为21。根据二分查找方法可知指针low和high最初分别指向元素5和92，由mid=（low+high）/2知，指针mid指向元素56。示意如下：



变量m所代表的整数21与指针mid所指的元素56进行比较，21小于56，根据二分查找算法可知，查找范围现在缩小到指针mid所指元素的前面，即从5到37的范围。指针high原来指向下标为N-1的元素，现在指向下标为mid-1的元素，接着重新计算指针mid所指元素的下标。示意如下：

再次进行比较，21大于19，现在比较范围再次转移到mid所指元素的后面，low元素所指元素下标由0变为mid+1。示意如下：



当前mid所指元素的值为21，与要查找的整数值相同，故查找成功，所查元素在表中的序号等于指针mid的值。

查找不成功的过程请读者自己完成。

4. 完整的程序

程序流程图如图1.17所示。

完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 折半查找

if __name__ == "__main__":
    a = [-3, 4, 7, 9, 13, 45, 67, 89, 100, 180]
    low = 0                                                    # 数组上界
    high = len(a) - 1                                          # 数组下界
    k = -1                                                     #
记录下标
    print("a数组中的数据如下：")
    for i in a:
        print(i, end=" ")                                      # 输出数组中原数据序列
    print()
    m = int(input("Enter m = : "))                             # 变量m为要查找的整数
    while low <= high:                                         # 继续查找的控制条件
        mid = (low + high) // 2                                # 变量mid为数组序列的中间位置
        if m < a[mid]:
            high = mid - 1
        else:
            if m > a[mid]:
                low = mid + 1
            else:
                k = mid
                break                                          # 一旦找到所要查找的元
素便跳出循环
    if k ≥ 0:
        print("m = %d, index = %d" %(m, k))
    else:
        print("Not be found!")
```

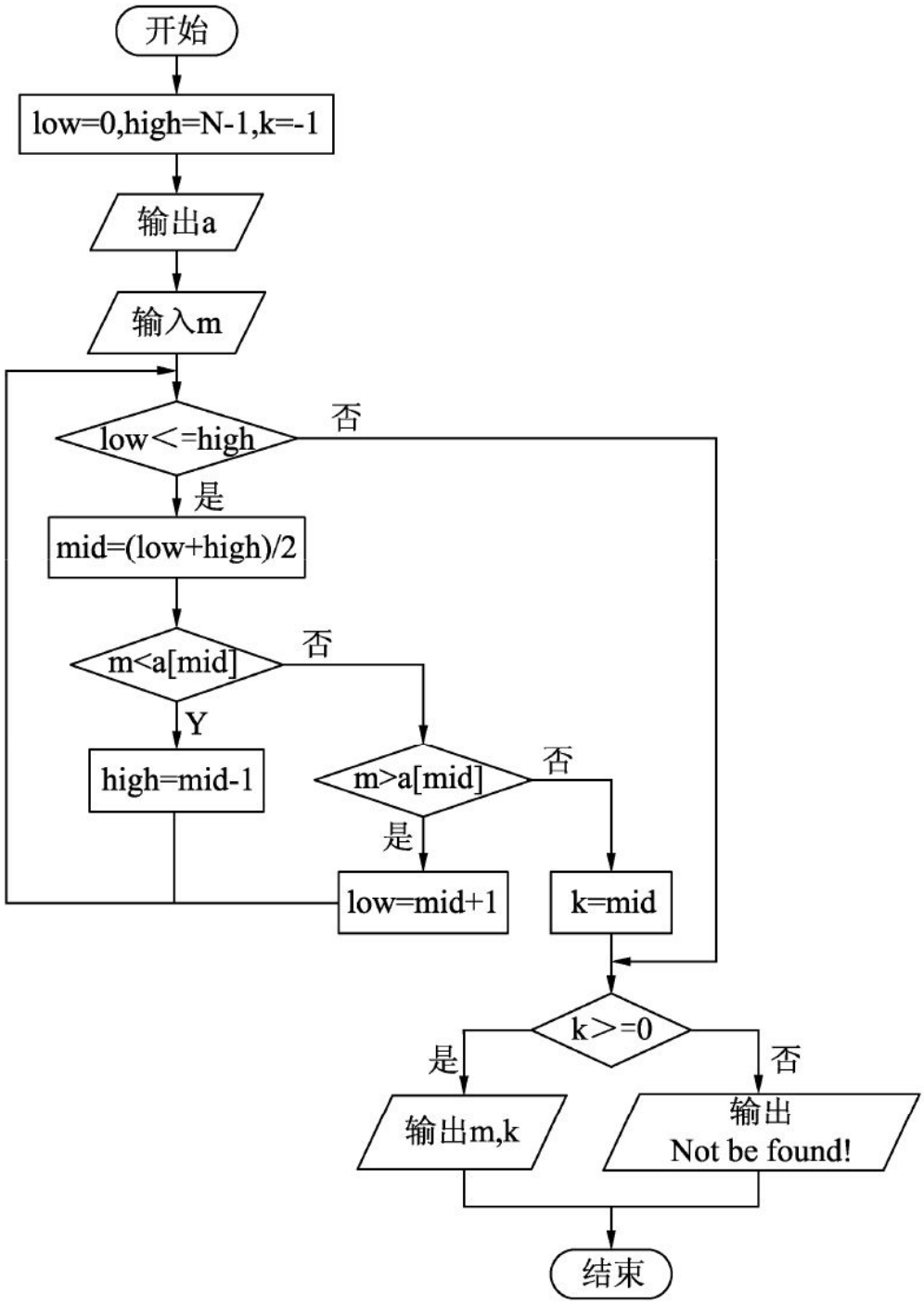


图1.17　程序流程图

在上述程序中循环结束可以有两种情况，一种是在循环的判定条件low≤high不成立的情况下跳出循环，此时可知查找不成功。在查找不成功的情况下if m>a[mid]语句的else语句块是不执行的，所以变量k的值不变仍为初值-1。第二种结束循环的情况是由于执行了break语句而跳出循环，在此情况下，变量k的值由-1变成了一个大

于等于0的数，即指针mid所指元素的下标值。所以在最后用选择结构来判定k的值，从而确定整个查找过程是否成功。

补充知识点：

（1）continue语句

continue语句的格式为：

```
continue
```

continue语句用于Python的循环语句中，作为循环体的一部分。在程序执行时，一旦遇到了continue语句，则立即结束本次循环，即跳过循环体中continue后面尚未执行的语句，接着进行是否继续循环的条件判定。

（2）break语句

break语句的格式如下：

```
break
```

break语句用于while循环和for循环中，用来终止循环语句，即使循环条件中没有False条件或序列还没有遍历完，也会停止执行循环语句。

如果使用嵌套循环，break语句就会停止执行最深层的循环，并开始执行下一行代码。

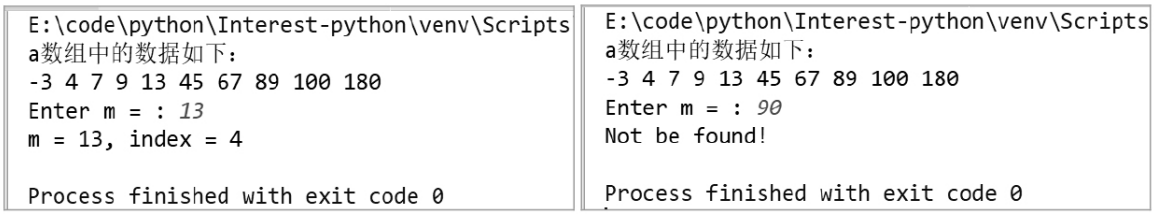
continue语句和break语句的区别如下：

• continue只是结束本次循环，不再执行循环体中continue后面的其余语句，并不是终止当前循环。

• 当遇到break语句时，无论执行什么条件，都跳出这个循环。

5. 运行结果

在PyCharm下运行程序，结果如图1.18所示。



a）运行结果 1　　b）运行结果 2

图1.18　运行结果

6. 问题拓展

在一个给定的数据结构中查找某个指定的元素，通常根据不同的数据结构应采用不同的查找方法。对于一个有序数列，除了采用二分查找法之外还可以采用顺序查找的方法。

顺序查找一般是指在线性表中查找指定的元素，其基本方法是：从线性表的第一个元素开始，依次将线性表中的元素与被查元素进行比较，若相等则表示找到，即查找成功；若线性表中所有的元素都与被查元素进行了比较但都不相等，则表示线性表中没有要找的元素，即查找失败。

在长度为n的线性表中查找指定元素，最好的情况是比较1次就成功，最坏的情况是比较n次，平均要比较(1+2+3+…+n)/n=(1+n)/2次。尽管顺序查找的效率低，但有些情况只能采用顺序查找的方法，如对于一个无序表进行查找。

完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 折半查找

if __name__ == "__main__":
    a = [-3, 4, 7, 9, 13, 45, 67, 89, 100, 180]
    k = -1                                                    #
记录下标
    print("a数组中的数据如下：")
    for i in a:
```

```
        print(i, end=" ")                                   # 输出数组中原数据序列
    print()
    m = int(input("Enter m = : "))          # 变量m为要查找的整数
    i = 0
    while i < len(a):
        if m == a[i]:
            k = i
            break                                                   # 一旦找到所
要查找的元素便跳出循环
        i += 1
    if k >= 0:
        print("m = %d, index = %d" %(m, k))
    else:
        print("Not be found!")
```

对数据进行排序及查找的方法较多，理解每一种方法的思想是编程的第一步，不同方法之间的效率是不同的，在没有特别要求的情况下读者可根据自己掌握的情况进行编程。但对于一些有一定适用范围的方法，读者一定要牢记。

1.10　数制转换

1．问题描述

给定一个M进制的数x，实现对x向任意一个非M进制的数的转换。

2．问题分析

掌握不同数制间的转换关系是解决本题的关键，这里所说的数制一般包括二进制、八进制、十六进制及十进制。除了不同的数制之外，还有几个必须要了解的概念：

• 基数：在一种数制中，只能使用一组固定的数字来表示数的大小。具体使用多少个不同的数字来表示一个数值的大小，就称为该计数制的基数（Base）。如十进制的基数为10、二进制的基数为2等。

• 权：又称为位权或权值，即每一个数位都有一个固定的基值与之相对应。如十进制的个位对应的权值为1（10^0 ），十位对应的权值为10（10^1 ），百位对应的权值为100（10^2 ），对于一个M进制的数来说，小数点左边各位上对应的权值从左到右分别为基数的0次方、基数的1次方、基数的2次方等，对于小数点右边各位上对应的权值从右到左分别为基数的-1次方、基数的-2次方等。

二进制、八进制、十六进制向十进制转换的方法为：按权展开相加。

十进制转换成二进制、八进制、十六进制的方法为：整数部分除以基数取余数（取余的方向为从后向前），小数部分乘以基数取整数（取整的方向为从前向后）。

二进制、八进制、十六进制相互转换的方法为：先转换成十进制再转换成其他进制；或者按照其对应关系进行转换（3位二进制数

对应一位八进制数，4位二进制数对应一位十六进制数）。本题按照前一种转换方式进行编程。

3. 算法设计

十六进制是由0～F这一组固定的数字来表示的，所以采用字符数组进行存储。在进行输入、输出时，数组元素都是以字符的形式存在的，但是在进行数制转换时数组元素又以数字的形式存在，程序中我们用两个自定义函数char_to_number()及number_to_char()来实现字符与其对应数字之间的转换。

在执行程序时我们希望可以输入多组数据来验证程序的正确性，以前的程序都是多次运行，输入不同的数据来实现。我们对程序稍做改进，使只运行一次程序但可以输入多组数据进行验证。解决这个问题只需要加一层循环，如果循环条件为真则继续输入数据，否则退出。循环条件为真即表达式的值不为0，这样我们可以利用一个变量假设为flag，利用语句“while flag:循环体”来进行控制，当flag的值为1时我们可以接着输入，若为0则结束循环。

4. 确定程序框架

程序主体框架如下：

```
if __name__ == '__main__':
    MAXCHAR = 101                                   # 允许的最大字符串长度
    flag = 1                                                # 存储是否退出程序的标
志
    while flag:                                     # 利用输入的flag值控制循环是否
结束
        # 将原数转换成十进制数
        # 求出转换成目标数制后字符数组的长度
        # 逆序打印字符数组
        print("继续请输入1,否则输入0：")
flag = int(input())
```

5. 字符与数字进行转换

将输入或存储的字符转换为对应的数字，可以分两类进行考虑：第一类是介于’0’到’9’之间的字符，转换成相应的数字0～9时，可利用其ASCII码之间的对应关系。字符’0’的ASCII码为

48，'1'的ASCII码为49，'1'-'0'=1（在0～127之间字符型与整型是可以通用的）得到的差即为字符ch对应的数字。第二类是介于'A'到'Z'之间的字符，字符'A'对应的数字为10，'B'对应的数字为11，其他字母以此类推。

值得高兴的是，Python为我们提供了以下函数，用于实现数字、字母和字符之间的转换。

- int(x)：将其他类型转换成数字。
- ord(x)：将字符转换成对应的Unicode码。
- str(x)：将其他类型转换成字符串。
- chr(x)：将十进制数字转换成对应的字符。

将字符转换成数字，代码如下：

```
# 将字符转换成数字
def char_to_num(ch):
    if ch >= '0' and ch <= '9':
        return int(ch)                                  # 将数字字符转换成数字
    else:
        return ord(ch)                                  # 将字母字符转换成数字

# 将数字转换为字符
def num_to_char(num):
    if num >= 0 and num <= 9:
        return str(num)                         # 将0～9之间的数字转换成字符
    else:
        return chr(num)                         # 将大于10的数字转换成字符
```

6. 其他数制转换成十进制

其他数制转换成十进制，采用按权展开相加的方法，所以需要定义一个变量来存储相加之后的和，假设变量为decimal_num。因数组元素类型为字符型，所以首先需要调用char_to_num(temp[i])函数将元素类型转化成数值型然后参与运算。

定义source_to_decimal()函数来完成数值转换，该函数代码如下：

```
# 其他数制转换为十进制
def source_to_decimal(temp, source):
    decimal_num = 0                                    # 存储展开之后的和

    for i in range(len(temp)):       # 累加
        decimal_num=(decimal_num * source)+char_to_num(temp[i])

    return decimal_num
```

7. 十进制转换成其他数制

十进制转换成其他数制，采用除以基数取余的方法。以十进制转换成八进制为例，首先用当前的十进制数除以要转换成的数制的基数8，得到的余数存放在数组元素decimal中，为了使余数的类型由数值型转换成字符型，需调用num_to_char(decimal_num%object)函数，将相除之后的十进制数再次赋值给存储原数据的变量decimal_num；然后用得到的新十进制数再去除基数，将余数转换成字符型存入decimal数组中；一直重复上述过程，直到原来的十进制数为0。

定义函数decimal_to_object()来实现上述功能，其代码如下：

```
# 十进制转换为其他数制
def decimal_to_object(decimal_num, object):
    decimal = []
    while decimal_num:
        # 求出余数并转换为字符
        decimal.append(num_to_char(decimal_num % object))
        decimal_num //= object                                  # 用十进制数除以基数

    return decimal
```

由余数组成的新数制数与余数的顺序是相反的，所以在输出新数的时候我们采用的是逆序输出的方式，定义output()函数用于完成新数的输出，代码如下：

```
# 修改余数数制
def output(decimal):
    f = len(decimal)-1
    while f >= 0:
        print(decimal[f], end='')
        f -= 1
    print()
```

8. 完整的程序

完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 数制转换

# 将字符转换成数字
def char_to_num(ch):
    if ch >= '0' and ch <= '9':
        return int(ch)                                  # 将数字字符转换成数字
    else:
        return ord(ch)                                  # 将字母字符转换成数字


# 将数字转换为字符
def num_to_char(num):
    if num >= 0 and num <= 9:
        return str(num)                         # 将0～9之间的数字转换成字符
    else:
        return chr(num)                         # 将大于10的数字转换成字符


# 其他数制转换为十进制
def source_to_decimal(temp, source):
    decimal_num = 0                                     # 存储展开之后的和

    for i in range(len(temp)):      # 累加
        decimal_num=(decimal_num * source)+char_to_num(temp[i])

    return decimal_num


# 十进制转换为其他数制
def decimal_to_object(decimal_num, object):
    decimal = []
    while decimal_num:
        # 求出余数并转换为字符
        decimal.append(num_to_char(decimal_num % object))
        decimal_num //= object              # 用十进制数除以基数

    return decimal


# 修改余数数制
def output(decimal):
    f = len(decimal)-1
    while f >= 0:
        print(decimal[f], end='')
        f -= 1
    print()

if __name__ == '__main__':
    MAXCHAR = 101                                       # 允许的最大字符串长度
    flag = 1                                                    # 存储是否退出程序的标
志
    while flag:                                         # 利用输入的flag值控制循环是否
结束
        print("转换前的数是：", end='')
        temp = input()
        print("转换前的数制是：", end='')
        source = int(input())
        print("转换后的数制是：", end='')
        object = int(input())
        print("转换后的数是：", end=’’)
        decimal_num = source_to_decimal(temp, source)
        decimal = decimal_to_object(decimal_num, object)
        output(decimal)
```

```
    print("继续请输入1,否则输入0：")
    flag = int(input())
```

9. 运行结果

在PyCharm下运行程序，结果如图1.19所示。

```
E:\code\python\Interest-python\venv\Scripts
转换前的数是：15
转换前的数制是：16
转换后的数制是：10
转换后的数是：21
继续请输入1,否则输入0:
1
转换前的数是：10
转换前的数制是：8
转换后的数制是：16
转换后的数是：8
继续请输入1,否则输入0:
1
转换前的数是：85
转换前的数制是：10
转换后的数制是：8
转换后的数是：125
继续请输入1,否则输入0:
1
转换前的数是：20
转换前的数制是：10
转换后的数制是：2
转换后的数是：10100
继续请输入1,否则输入0:
0

Process finished with exit code 0
```

图1.19　运行结果

第2章　趣味数学问题

本章通过与生活相关的一些小例子引出其中包含的数学问题。这些问题有的可以抽象出数学公式，有的可以转换成不定方程，还有的需要简单推导得出通式。使用Python语言将这些模型化的数学问题表达出来，就可以获得正确结果。

趣味数学问题的求解常常需要使用Python语言中的分支结构和循环结构。本章中的每个问题都给出了详细的流程图，对有些易混淆的地方读者可以参照流程图来理清思路，把握程序的走向。相信通过学习本章内容，读者对生活中的数学问题能更好地理解和掌握。

2.1　三色球

1．问题描述

一个口袋中放有12个球，已知其中3个是红的，3个是白的，6个是黑的，现从中任取8个，问共有多少种可能的颜色搭配？

2．问题分析

根据问题描述可设任取的8个球中红球为m个，白球为n个，则黑球为8-m-n个。又已知12个球中有3个红球、3个白球、6个黑球，因此，m的取值范围为[0,3]，n的取值范围为[0,3]，黑球的个数小于等于6，即8-m-n≤6。

3．算法设计

由上述分析可知，红、白、黑三种颜色球的个数的取值范围已经确定了，现在要求的是所有可能的颜色搭配情况，因此可以使用循环结构检测m、n范围内的所有可能取值，再代入8-m-n≤6中进行验证，能够满足条件8-m-n≤6的那些m、n和8-m-n的组合即为问题的解。

4．确定程序框架

程序流程图如图2.1所示。

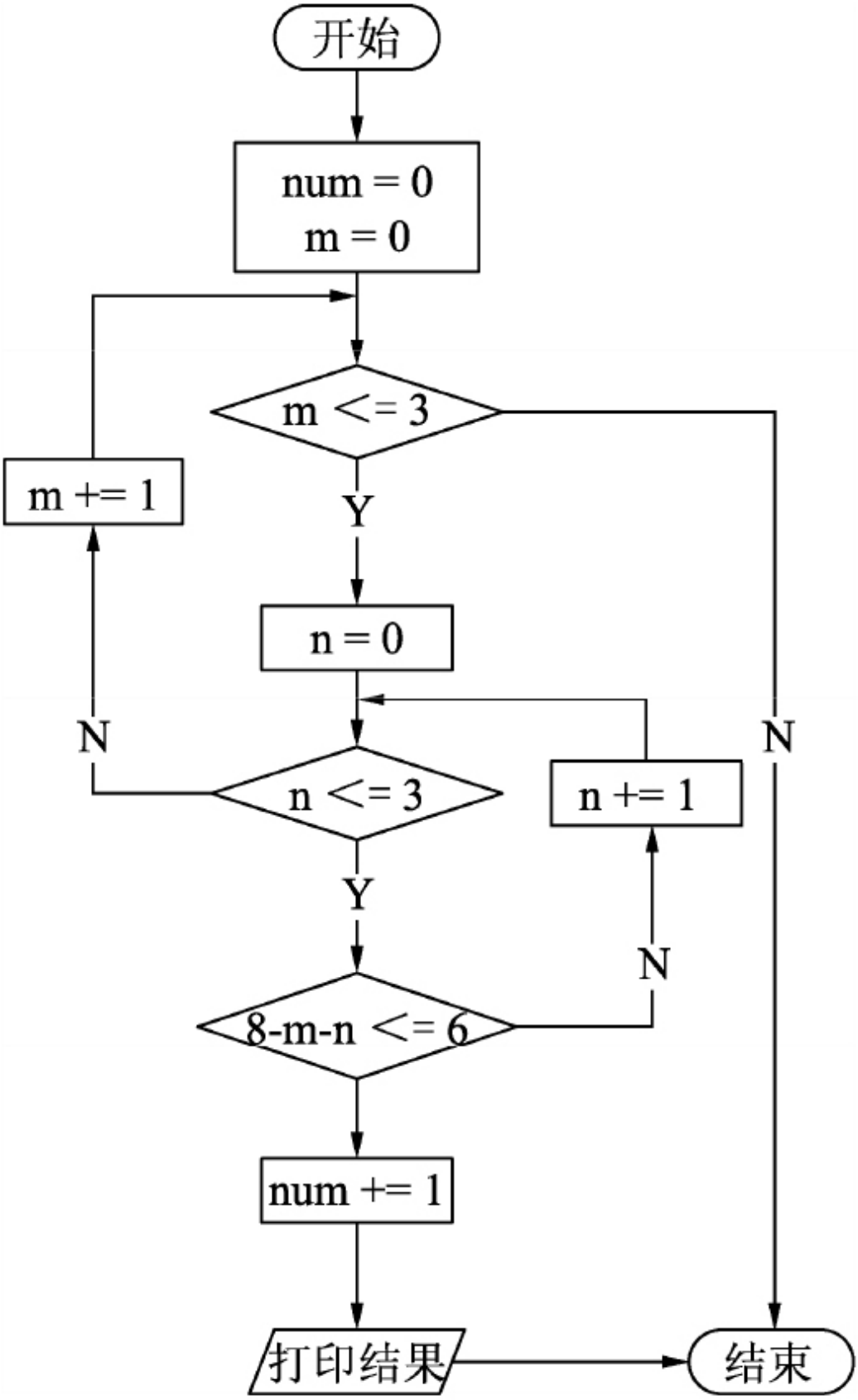


图2.1　程序流程图

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 三色球问题

if __name__=="__main__":
    # 从12个球中任取8个，红球m个，白球
      n个，黑球8-m-n个
    # m的取值范围为[0,3]，因此n的取值范
      围为[0,3]，黑球的个数小于等于6，即
      8-m-n≤6
    print("\t 红球 \t 白球 \t 黑球")
    print(".......................")
    num = 0
    for m in range(0, 4):
        for n in range(0, 4):
            if 8-m-n <= 6:
                num += 1
                print("%2d:  %d \t\t %d \t\t %d" %(num, m, n, 8-m-n))
```

6. 运行结果

在PyCharm下运行程序，结果如图2.2所示。从输出结果中可知，取出的8个球中，红、白、黑三色球可能的颜色搭配共有13种。

```
E:\code\python\Interest-python\venv\Scripts
     红球     白球     黑球
.........................
 1:  0        2        6
 2:  0        3        5
 3:  1        1        6
 4:  1        2        5
 5:  1        3        4
 6:  2        0        6
 7:  2        1        5
 8:  2        2        4
 9:  2        3        3
10:  3        0        5
11:  3        1        4
12:  3        2        3
13:  3        3        2

Process finished with exit code 0
```

图2.2　运行结果

2.2　出售金鱼

1．问题描述

小明将养的一缸金鱼分5次出售：第1次卖出全部的一半加1/2条；第2次卖出余下的三分之一加1/3条；第3次卖出余下的四分之一加1/4条；第4次卖出余下的五分之一加1/5条；最后卖出余下的11条。试编程求出原来鱼缸中共有多少条金鱼。

2．问题分析

依题意可知，金鱼是分5次出售的，每次卖出的方式都相同，因此可以用表达式将每次卖鱼后剩下的条数计算出来。

由：

第1次卖出全部的一半加1/2条；

第2次卖出余下的三分之一加1/3条；

第3次卖出余下的四分之一加1/4条；

第4次卖出余下的五分之一加1/5条。

可推出：

第j次卖出余下的(j+1)分之一加1/(j+1)条。

假设第j次卖鱼前金鱼总数为x，则第j次卖鱼后鱼缸中还剩下金鱼的条数为x-(x+1)/(j+1)。

又由于“最后卖出余下的11条”，因此第4次卖鱼后鱼缸中剩下的金鱼条数为11。

因为金鱼只能整条进行出售，因此x+1必然能够整除j+1。

可以从23开始试探x的取值，由于x值必为奇数，因此步长取2。

3. 确定程序框架

程序流程图如图2.3所示。

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 出售金鱼

if __name__=="__main__":
    flag = 0                                                    # flag作为控
制标志
    i = 23
    while flag == 0:
        j = 1                                                   # j表示卖鱼
的次数
        x = i                                                   # x表示每次
卖鱼的条数
        while j <= 4 and x >= 11:
            if (x + 1) % (j + 1) == 0:       # 判断x + 1是否能整除j + 1
                x -= (x+1)//(j+1)
            else:
                x = 0
                break
            j += 1
        if j == 5 and x == 11:
            print("原来鱼缸中共有%d条金鱼." %i)
            flag = 1                                            # 求出结果，flag置为
1，退出试探
        i += 2
```

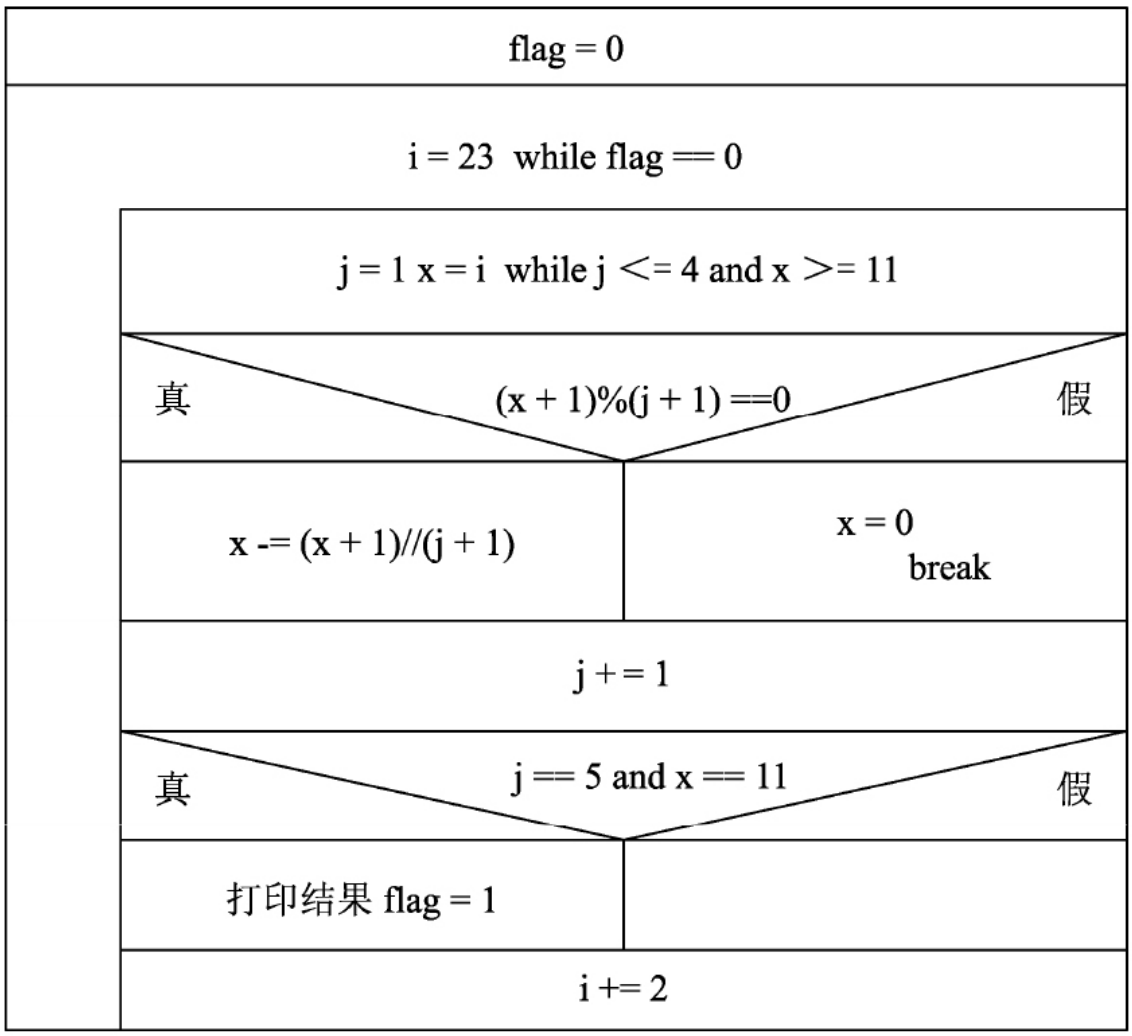


图2.3　程序流程图

5. 运行结果

在PyCharm下运行程序，结果如图2.4所示。由输出结果可知，原来鱼缸中共有59条金鱼。

```
E:\code\python\Interest-python\venv\
原来鱼缸中共有59条金鱼.

Process finished with exit code 0
```

图2.4　运行结果

2.3　求车速

1．问题描述

一辆以固定速度行驶的汽车，司机在上午10点看到里程表上的读数是一个对称数（即这个数从左向右读和从右向左读是完全一样的），为95859。两小时后里程表上出现了一个新的对称数，该数仍为5位数。问该车的速度是多少？新的对称数是多少？

2．问题分析

根据题意，司机在上午10点看到里程表上的读数是一个对称数95859，两小时后里程表上出现的新的对称数必然大于95859。因此，假设所求对称数为i，并设其初值为95860，即从95860开始检测，使i的取值依次递增。

对于i的每一次取值都将其进行分解，然后将对称位置上的数字进行比较，即第1位和第5位比较，第2位和第4位比较。如果每个处于对称位置上的数都是相等的，则可以判断出当前的i中所存放的5位数即为里程表上新出现的对称数。

3．算法设计

根据问题分析可知，i需要从95860开始试探，因此显然需要使用循环结构。在循环体中完成分解5位数并保存、再检测是否为对称数的功能。

根据问题分析可知，需要对一个5位数进行分解并保存，因此可以使用数组来保存分解后生成的5个数字。这样，在进行对称位置上的数字比较时，实际上进行的是指定下标的数组元素的比较。

4．确定程序框架

由上述分析可知，程序的主体是一个循环结构。

（1）循环试探

使用for语句进行循环试探，代码如下：

```
# 以95860为初值，循环试探
# i为里程数初始值，因最后结果仍是5位数，故最大值为100000
for i in range(95860, 100000):
    # 循环体语句
```

根据题意表述，两小时后里程表上出现了一个新的对称数，该数仍为5位数，故最大的循环次数是100000，而起始里程数是95860。

（2）分解5位数并保存

对当前变量i中存放的5位数进行分解，并将结果保存在数组a中。第1次分解出“万”位上的数字，存放在数组元素a[0]中；第2次分解出“千”位上的数字，存放在数组元素a[1]中；接着依次分解出“百”“十”“个”位上的数字，分别存放在数组元素a[2]、a[3]和a[4]中。代码如下：

```
# 从高到低分解当前i中保存的5位数，并顺次存放在数组元素a[0]～a[4]中
t = 0
# 列表a的下标
k = 100000
while k >= 10:
    a[t] = (i % k)//(k // 10)                                    # 保存分解后的数字
    k /= 10
    t += 1
```

（3）判断是否为对称数

分解出的5个数字按照从低位到高位的顺序分别保存在数组元素a[0]～a[4]中。因此，判断该5位数是否为对称数的条件为：a[0]==a[4] and a[1]==a[3]是否成立。

```
if a[0] == a[4] and a[1] == a[3]:
    print("里程表上出现的新的对称数为：%d%d%d%d%d" %(a[0],a[1],a[2],a[3],a[4]))
    print("该车的速度为：%.2f" %((i-95859)/2.0))
    break                                                                    #
跳出循环
```

上面的if语句中使用了break语句来跳出循环试探过程，即一旦找到一个对称的5位数，则立即跳出循环，这保证了经过有限次循环后程序可以正常结束。

程序流程图如图2.5所示。

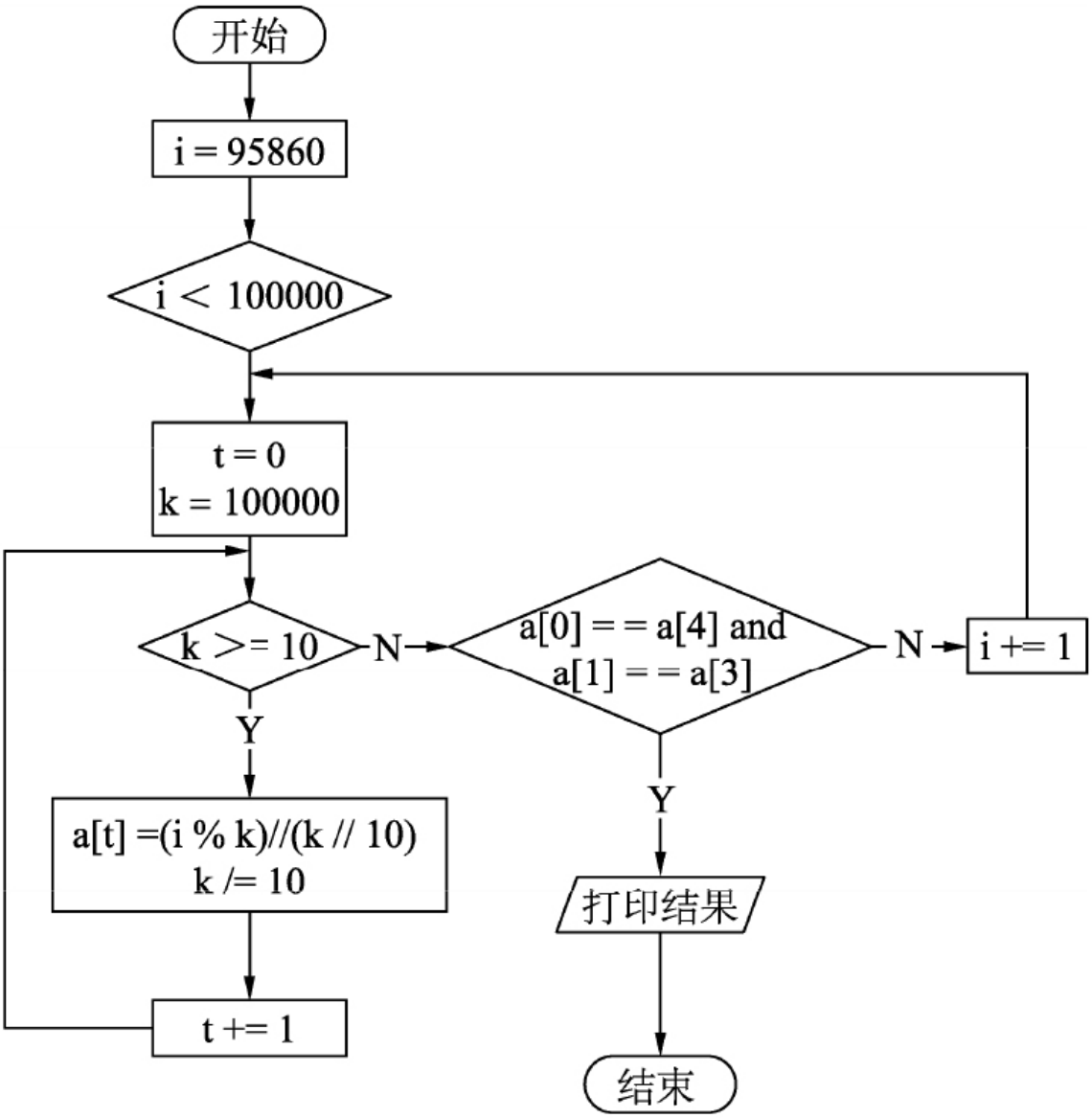


图2.5　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 求车速

if __name__=="__main__":
    a = [0, 0, 0, 0, 0]                                    # 列表a用来存放分解后的5个数字
    # i为里程数初始值，因最后结果仍是5位数，故最大值为100000
    for i in range(95860, 100000):
        # 从高到低分解当前i中保存的5位数，并顺次存放在数组元素a[0]～a[4]中
        t = 0                                                                  # 列表a的下
标
        k = 100000
        while k >= 10:
            a[t] = (i % k)//(k // 10)        # 保存分解后的数字
            k /= 10
            t += 1
        if a[0] == a[4] and a[1] == a[3]:
            print("里程表上出现的新的对称数为：%d%d%d%d%d" %
(a[0],a[1],a[2],a[3],a[4]))
            print("该车的速度为：%.2f" %((i-95859)/2.0))
            break                                                  # 跳出循环
```

6. 运行结果

在PyCharm下运行程序，结果如图2.6所示。由输出结果可知，两小时后里程表上出现的新的对称数为95959，该车的速度为50.00，程序在输出速度时保留了两位小数。

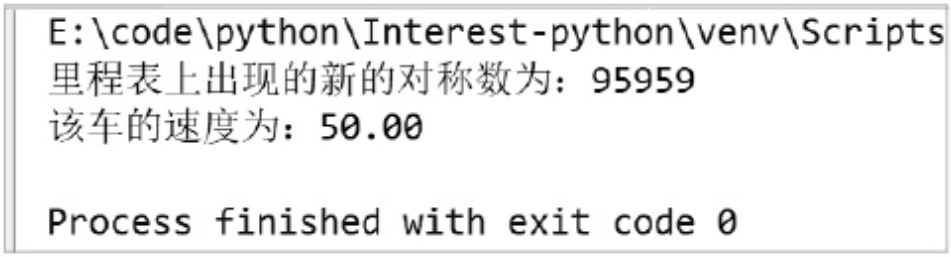


图2.6　运行结果

7. 问题拓展

该程序的主体是一个循环结构，使用了for语句进行循环试探，i是循环变量，初值为95860，代码如下：

```
# 以95860为初值，循环试探
# i为里程数初始值，因最后结果仍是5位数，故最大值为100000
for i in range(95860, 100000):
    # 循环体语句
```

也可以使用while循环结构来替代上面的for循环。在进入while循环前要先设置i的初值为95860，while循环的条件为永真，因此，在循环体中要有退出循环的条件。完整的代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 求车速

if __name__=="__main__":
    a = [0, 0, 0, 0, 0]                                          # 列表a用来存放分解后的5个数字
    i = 95860                                                                  # i为里程数初始值
    while 1:
        # 从高到低分解当前i中保存的5位数，并顺次存放在数组元素a[0]～a[4]中
        t = 0                                                                  # 列表a的下标
        k = 100000
        while k >= 10:
            a[t] = (i % k)//(k // 10)                # 保存分解后的数字
            k /= 10
            t += 1
        if a[0] == a[4] and a[1] == a[3]:
            print("里程表上出现的新的对称数为：%d%d%d%d%d" %(a[0],a[1],a[2],a[3],a[4]))
            print("该车的速度为：%.2f" %((i-95859)/2.0))
            break                                                              # 跳出循环
        i += 1
```

从上面的代码中可以看到，除了将for循环改为while循环以外，循环体中的语句没有变化，当找到新的对称数以后，就可以使用break语句跳出while循环了。

2.4　个人所得税

1. 问题描述

编写一个计算个人所得税的程序，要求输入收入金额后，能够输出应缴的个人所得税。个人所得税征收办法如下：

起征点为2000元。

- 不超过500元的部分，征收5%。
- 超过500～2000元的部分，征收10%。
- 超过2000～5000元的部分，征收15%。
- 超过5000～20000元的部分，征收20%。
- 超过20000～40000元的部分，征收25%。
- 超过40000～60000元的部分，征收30%。
- 超过60000～80000元的部分，征收35%。
- 超过80000～100000元的部分，征收40%。
- 超过100000元以上的，征收45%。

2. 问题分析

分析题目特点，我们可以考虑使用Python语言的列表和元组来描述题目中的条件。下面先讲解Python语言中列表和元组的语法要点。

（1）列表

列表是以方括号“[]”包围的数据集合，列表元素之间使用逗号“,”分割。列表的元素类型可以是任何数据类型，也可以包含另一个列表，可以通过列表的下标来访问列表中的元素，下标从0开始。列表的内容是可变的，定义一个列表如下：

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

其中list[0]=1，list[1]=2，以此类推。

使用for语句遍历这个列表如下：

```
for i in list:
    print(i, end=" ")
```

将会打印出：1 2 3 4 5 6 7 8 9。

下面是列表常用的操作方法。

• list.append(x)：添加元素，将元素x添加到列表list的尾部。

• list.extend(aList)：添加元素，将列表aList中的所有元素添加到列表list的尾部。

• list.insert(index,x)：添加元素，在列表list中的指定位置index处插入元素x。

• list.remove(x)：删除元素，在列表list中删除首次出现的指定元素x。

• list.pop([index])：删除元素，删除并返回列表list中指定位置index处的元素，默认是最后一个元素。

• list.clear()：删除元素，删除列表中的所有元素，并不是删除列表对象。

• list.index(x)：返回第一个x的索引位置，若不存在x元素则抛出异常。

• list.index(x1,x2)：从索引位置x2开始往后搜索的第一个x1。

• list.count(x)：返回指定元素x在列表list中出现的次数。

• len(list)：返回列表中包含元素的个数。

• list.reverse()：翻转列表，所有元素原地翻转。

• list.sort()：对原列表进行排序，默认是升序。

• list.sort(reverse=True)：降序排序。

• list.copy()：返回列表对象的浅复制。

（2）元组

Python的元组与列表相似，元组是以圆括号“()”包围的数据集合。与列表不同的是元组的元素一旦确定就不能再改变，它是不可变数据类型。因此它常常被用在不希望数据被其他操作所改变的场景中。

如果圆括号中不包含任何内容，就是一个空元组。

定义一个元组如下：

```
tuple = ('a', 'b', 'c', 'd')
```

遍历元组如下：

```
for i in tuple:
    print(i, end=" ")
```

将会打印输出：a b c d。

3．算法设计

根据列表和元组的相关知识，这里可以同时使用列表和元组来存放不同的税率范围。接着使用for循环遍历每一个征税范围，将个人收入中超出起征点的金额在每个征税范围内应缴纳的税款累加起来，就得到最后应缴纳的个人所得税。

4．确定程序框架

（1）定义列表TaxTable

其中，元素类型使用元组，该列表描述了征税的范围及对应不同范围的税率。定义如下：

```
#分为9个阶段，每个阶段第一个值为个税起征点，第二个值为该阶段截止点，第三个值为税率
TaxTable = [(0, 500, 0.05),
            (500, 2000, 0.10),
            (2000, 5000, 0.15),
            (5000, 20000, 0.20),
            (20000, 40000, 0.25),
            (40000, 60000, 0.30),
            (60000, 80000, 0.35),
            (80000, 100000, 0.40),
            (100000, 1e10, 0.45)]
```

（2）定义计算税率的函数CaculateTax()

其中，profit为个人收入，TAXBASE是个税起征点。profit-TAXBASE是个人收入中超出个税起征点的部分，仍存入profit变量中，在CaculateTax()中要计算出这部分收入的纳税金额。

```
#计算税收
def CaculateTax(profit):
    tax = 0.0
    profit -= TAXBASE                                                    #
超过个税起征点的收入
    i = 0
    for i in range(len(TaxTable)):
        # 判断profit是否在当前的缴税范围内
        if (profit > TaxTable[i][0]):
            if (profit > TaxTable[i][2]):                # profit超过当前的缴税范
围
                tax += (TaxTable[i][1] - TaxTable[i][0]) * TaxTable[i][2]

            else:                                                        #
profit未超过当前的缴税范围
                tax += (profit - TaxTable[i][0]) * TaxTable[i][2]

            profit -= TaxTable[i][1]
```

```
            if profit < 0:
                profit = 0

            print("征税范围：%6d～%6d  该范围内缴税金额：%6.2f  超出该范围的金额：%6d"
% (TaxTable[i][0], TaxTable[i][1], tax, profit))
    return tax
```

在CaculateTax()函数中使用了for循环，循环变量为i，循环次数与TaxTable数组中的元素个数相同。在循环体中用profit（注意此时的profit中存放的是个人收入中超过个税起征点的部分）与征税范围做比较，如果TaxTable[i][1]>profit>TaxTable[i][0]，即profit恰好处于某个范围内，则在该范围内应缴税金额为：(profit-TaxTable[i][0])*TaxTable[i][2]；如果profit>TaxTable[i][1]，则在该范围内应缴税金额为：(TaxTable[i][1]-TaxTable[i][0])*TaxTable[i][2]。

使用for循环将每个征收范围遍历一遍，将各个范围内产生的缴税金额累加起来，就得到应该缴纳的个人所得税的总金额。

CaculateTax()函数的流程图如图2.7所示。

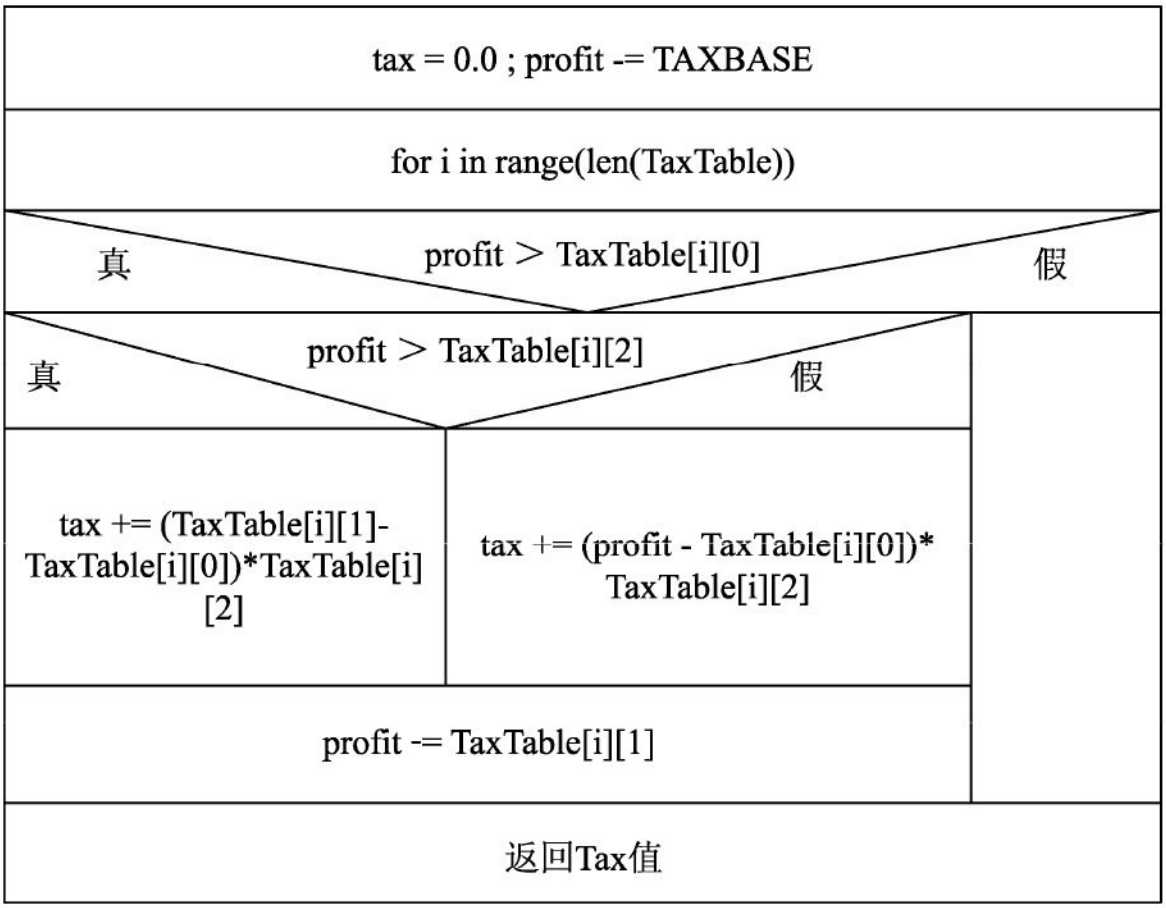


图2.7　CaculateTax()函数的流程图

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 个人所得税问题


TAXBASE = 2000

#分为9个阶段，每个阶段第一个值为个税起征点，第二个值为该阶段截止点，第三个值为税率
TaxTable = [(0, 500, 0.05),
            (500, 2000, 0.10),
            (2000, 5000, 0.15),
            (5000, 20000, 0.20),
            (20000, 40000, 0.25),
            (40000, 60000, 0.30),
            (60000, 80000, 0.35),
            (80000, 100000, 0.40),
            (100000, 1e10, 0.45)]

#计算税收
def CaculateTax(profit):
    tax = 0.0
    profit -= TAXBASE                                                    # 超过个税起
```

```
征点的收入
    i = 0
    for i in range(len(TaxTable)):
        # 判断profit是否在当前的缴税范围内
        if (profit > TaxTable[i][0]):
            if (profit > TaxTable[i][2]):        # profit超过当前的缴税范围
                tax += (TaxTable[i][1] - TaxTable[i][0]) * TaxTable[i][2]

            else:                                                    # profit未
超过当前的缴税范围
                tax += (profit - TaxTable[i][0]) * TaxTable[i][2]

            profit -= TaxTable[i][1]
            if profit < 0:
                profit = 0

            print("征税范围：%6d～%6d  该范围内缴税金额：%6.2f  超出该范围的金额：%6d"
% (TaxTable[i][0], TaxTable[i][1], tax, profit))
    return tax

if __name__ == '__main__':
    print("请输入个人收入金额: ", end='')
    profit = int(input())
    tax = CaculateTax(profit)
    print("您的个人所得税为 %12.2f" % tax)
```

6. 运行结果

在PyCharm下运行程序，屏幕上提示“请输入个人收入金额：”，输入5678，运行结果如图2.8所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/code/python/Inte
请输入个人收入金额: 5678
征税范围:      0~   500  该范围内缴税金额:  25.00  超出该范围的金额:   3178
征税范围:    500~  2000  该范围内缴税金额: 175.00  超出该范围的金额:   1178
您的个人所得税为       175.00

Process finished with exit code 0
```

图2.8　运行结果

2.5　存钱

1. 问题描述

假设银行整存整取存款不同期限的月利率为：

- 0.63%，期限为1年；
- 0.66%，期限为2年；
- 0.69%，期限为3年；
- 0.75%，期限为5年；
- 0.84%，期限为8年。

现在已知某人手上有2000元，要求通过计算选择出一种存钱方案，使得这笔钱存入银行20年后获得的利息最多。假定银行对超出存款期限的那部分时间不付利息。

2. 问题分析

为了获取到最多的利息，应该在存入银行的钱到期后马上就取出来，然后再立刻将原来的本金加上当前所获取到的利息作为新的本金存入银行中，这样反复操作直到满20年为止。

又由于存款的期限不同，对应的利率是不相同的，因此在20年中，不同的存取期限的组合所获得的利息也是不相同的。

假设在这20年中，1年期限的存了x1次，2年期限的存了x2次，3年期限的存了x3次，5年期限的存了x5次，8年期限的存了x8次，则到期时存款人所得的本利合计为：

$$2000\times(1+0.063)^{x1}\times(1+0.066)^{x2}\times(1+0.069)^{x3}\times(1+0.075)^{x5}\times(1+0.084)^{x8}\quad①$$

由题意可知，显然8年期限的存款次数最多为两次，因此可得到下面对存款期限的限定条件：

$0 \leqslant x8 \leqslant 2$

$0 \leqslant x5 \leqslant (20-8\times x8)/5$

$0 \leqslant x3 \leqslant (20-8\times x8-5\times x5)/3$

$0 \leqslant x2 \leqslant (20-8\times x8-5\times x5-3\times x3)/2$

$x1=20-8\times x8-5\times x5-3\times x3-2\times x2$且$x1 \geqslant 0$

3. 算法设计

根据式①及对存款期限的限定条件，可以使用for循环来穷举出所有可能的存款金额，从中找出最大的存款金额就是该问题的解。

因为限定条件已经确定了，因此for循环的循环次数也就确定了。

4. 确定程序框架

程序流程图如图2.9所示。

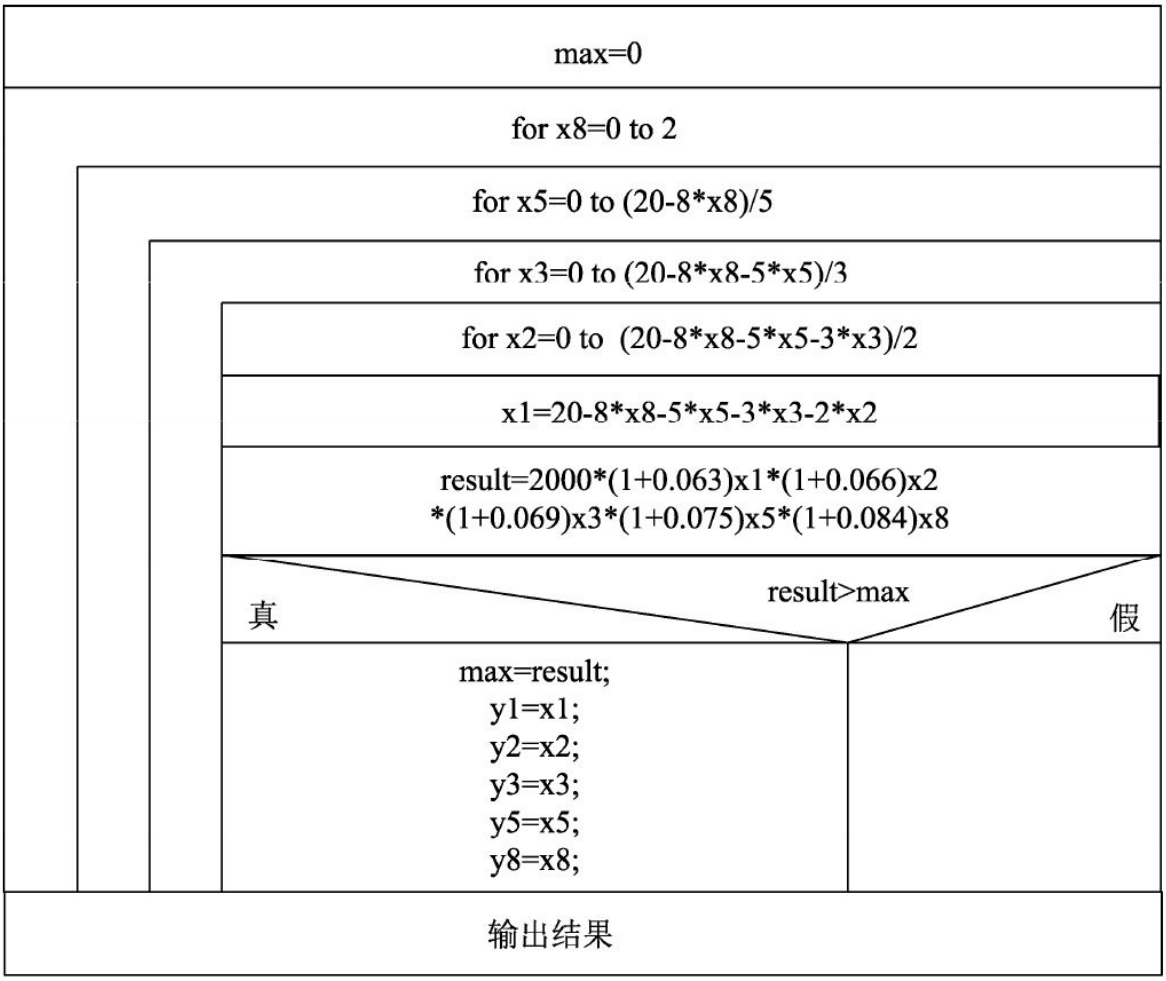


图2.9　程序流程图

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 存钱问题

if __name__=="__main__":
    #在20年中，1年期限的存了x1次，2年期限的存了x2次，以此类推
    max = 0.0
    for x8 in range(0,3):
        t5 = (20-8*x8)//5                                                    #存
款5年的最大次数
        for x5 in range(0, t5+1):
            t3 = (20-8*x8-5*x5)//3                                          #存款3年的最
大次数
            for x3 in range(0, t3+1):
                t2 = (20-8*x8-5*x5-3*x3)//2
                for x2 in range(0, t2+1):
                    x1 = 20-8*x8-5*x5-3*x3-2*x2      # 存款期限限定条件
                    # 判断条件
                    result = 2000* ((1+0.0063*12)**x1) * ((1+2*0.0066*12)**x2)
* ((1+3*0.0069*12)**x3) * ((1+5*0.0075*12)**x5) * ((1+8*0.0084*12)**x8)
```

```
                    # y1、y2、y3、y5、y8用于记录获利最多的存款方式
                    if result > max:
                        max = result
                        y1 = x1
                        y2 = x2
                        y3 = x3
                        y5 = x5
                        y8 = x8
# 输出结果
print("获得利息最多的存款方式为：");
print("8年期限的存了%d次" %y8);
print("5年期限的存了%d次" %y5);
print("3年期限的存了%d次" %y3);
print("2年期限的存了%d次" %y2);
print("1年期限的存了%d次" %y1);
print("存款人最终的获得的本利合计：%0.2f" %result);
```

程序说明：

1）“**”表示幂运算，x**y返回x的y次幂，即计算x^y 的值。例如，5**2=25。

2）使用括号“()”改变程序的优先级。Python语言中，括号“()”拥有最高优先级，可以强制表达式按照需要的顺序求值，括号中的表达式会优先执行，也可以利用括号使得表达式更加易读。

6. 运行结果

在PyCharm下运行程序，结果如图2.10所示。从输出结果中可知，获利最多的存款方式为连续存4次5年期的存款，则满20年所得到的本金一共为8763.19元。注意在程序中控制了输出结果的小数位数为两位。

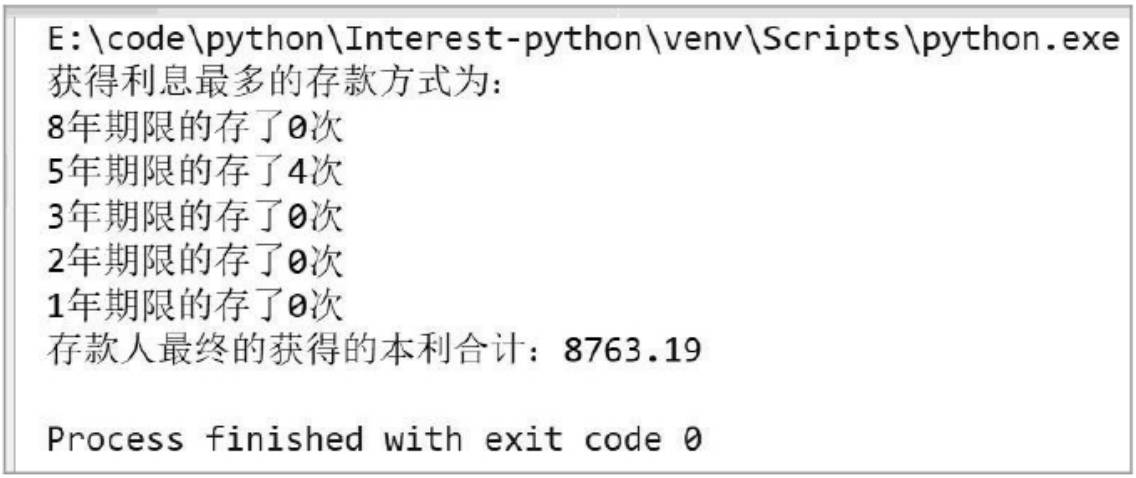


图2.10　运行结果

2.6　分糖果

1．问题描述

10个小孩围成一圈分糖果，老师分给第1个小孩10块，第2个小孩2块，第3个小孩8块，第4个小孩22块，第5个小孩16块，第6个小孩4块，第7个小孩10块，第8个小孩6块，第9个小孩14块，第10个小孩20块。然后所有的小孩同时将手中的糖分一半给右边的小孩；糖块数为奇数的人可向老师要一块。问经过这样几次后大家手中的糖一样多？每人各有多少块糖？

2．问题分析

根据题意，10个小孩开始时所拥有的糖果数是不同的，但分糖的动作却是相同的，即“所有的小孩同时将手中的糖分一半给右边的小孩；糖块数为奇数的人可向老师要一块”。因此，这是一个典型的可使用循环结构来解决的问题。

将老师开始给每个小孩分配的糖果数作为循环的初始条件，以“所有的小孩同时将手中的糖分一半给右边的小孩；糖块数为奇数的人可向老师要一块”这个重复的动作作为循环体，循环的结束条件为所有小孩手中的糖块数一样多。在循环体中，还需要判断糖块数的奇偶性，奇偶性不同完成的操作也不相同，显然这需要使用一个选择结构来实现。

3．算法设计

在问题分析中，我们已经确定了该问题使用循环结构来解决。那么如何存放每个小孩初始时所拥有的糖果数呢？这里考虑使用数组来存放老师开始给每个小孩分配的糖果数，因为有10个小孩，故定义一个长度为10的整型数组即可。在循环过程中，糖果每经过一次重新分配，就打印输出一次，直到最后一次打印时，10个小孩所拥有的糖果数都相同，此时结束循环。

4. 确定程序框架

（1）定义整型数组存放初始条件

```
# sweet[0]=10表示第一个小孩的糖果数为10，以此类推
sweet = [10, 2, 8, 22, 16, 4, 10, 6, 14, 20]
```

将老师开始给每个小孩分配的糖果数存放到sweet数组中。

（2）循环结构实现框架

```
while (10个孩子手中的糖果数不相同):                                    # 若不满足要求则继续循环
# 将每个孩子手中的糖果数平分为两份
    for i in range(0, 10):
        if sweet[i] % 2 == 0:                                          # 若为偶数则直接分出一
半
            sweet[i] = sweet[i] // 2
            t[i] = sweet[i]
        else:                                                                  #
若为奇数则加1后再分出一半
            sweet[i] = (sweet[i] + 1) // 2
            t[i] = sweet[i]

    # 将分出的一半糖果给右边的孩子
    for n in range(0, 9):
        sweet[n + 1] = sweet[n + 1] + t[n]
    sweet[0] += t[9]
    j += 1
    printResult(sweet, j)                                              # 输出当前每个孩子手中
的糖果数
```

上面的代码在while循环结构中又包含了两个for循环。

while循环的循环条件为“10个孩子手中的糖果数不相同”，第一个for循环用来将当前每个孩子手中的糖果分成一半，同时将分配结果保存在数组t中。在分配时注意区分奇偶数糖果分配方式的不同。第二个for循环用来将每个孩子手中已分好的一半的糖果给右边的孩子，由于第一个for循环中已经将每个孩子手中一半的糖果数保存在t数组中了，因此可直接利用t数组中的值修改sweet数组中的对应元素值。又由于“10个小孩围成一圈分糖果”，因此，sweet[9]右边的元素为sweet[0]。

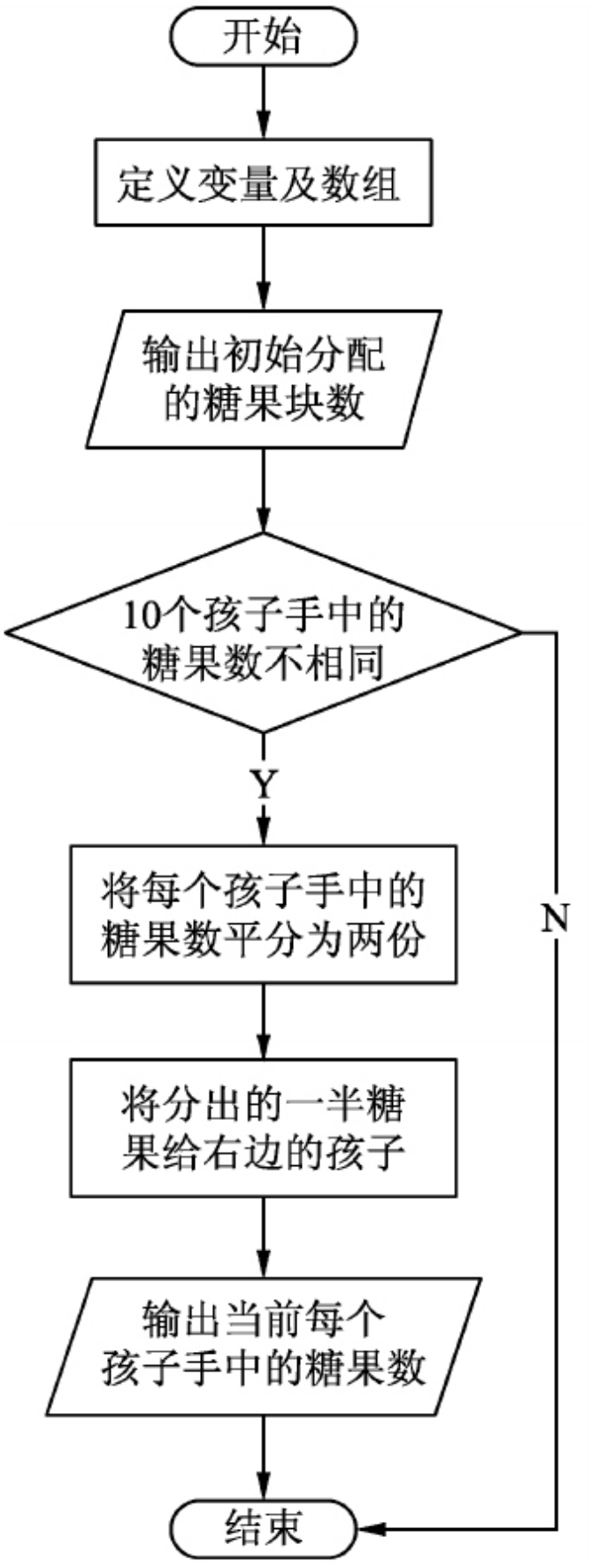

开始
定义变量及数组
输出初始分配
的糖果块数
10个孩子手中的
糖果数不相同
Y
N
将每个孩子手中的
糖果数平分为两份
将分出的一半糖
果给右边的孩子
输出当前每个
孩子手中的糖果数
结束

图2.11　程序流程图

（3）定义judge()函数

judge()函数用来判断每个孩子手中的糖果数是否相同。judge()函数的参数为整型数组，它可以判断该数组中各个元素的值是否相同，如果数组中所有元素的值都相同，则judge()函数返回值为0，否则judge()函数返回值为1。

judge()函数代码如下：

```
# 判断每个孩子手中的糖果数是否相同
def judge(candy):
    for i in range(0,10):
        if candy[0] != candy[i]:
            return 1                                    # 不相同返回1
    return 0                                                #相同返回0
```

程序流程图如图2.11所示。

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 分糖果

# 判断每个孩子手中的糖果数是否相同
def judge(candy):
    for i in range(0,10):
        if candy[0] != candy[i]:
            return 1                                    # 不相同返回1
    return 0                                                #相同返回0


def giveSweets(sweet, j):
    t = [0] * 10
    while (judge(sweet)):                               # 若不满足要求则继续循环
        # 将每个孩子手中的糖果数平分为两份
        for i in range(0, 10):
            if sweet[i] % 2 == 0:                   # 若为偶数则直接分出一半
                sweet[i] = sweet[i] // 2
                t[i] = sweet[i]
            else:                                           # 若为奇数则加1后再分
出一半
                sweet[i] = (sweet[i] + 1) // 2
                t[i] = sweet[i]

        # 将分出的一半糖果给右边的孩子
        for n in range(0, 9):
            sweet[n + 1] = sweet[n + 1] + t[n]
        sweet[0] += t[9]
        j += 1
        printResult(sweet, j)
```

```
# 输出列表中每个元素的值
def printResult(s, j):

    print("%4d" %j , end=" ")

    k = 0
    while k < 10:
        print("%4d" % s[k], end=" ")
        k += 1
        j += 1
    print()


if __name__=="__main__":
    # 定义一个列表来存储老师给每个孩子分配的糖果数
    # sweet[0]=10表示第一个小孩的糖果数为10，以此类推
    sweet = [10, 2, 8, 22, 16, 4, 10, 6, 14, 20]
    print("child  1    2    3    4    5    6    7    8    9   10")
    print("..........................................................")
    print("次数    糖果数")
    # 输出每个孩子手中的糖果数
    j = 0
    print("%4d" % j, end=" ")
    for i in range(len(sweet)):
        print("%4d" % sweet[i], end=" ")
    print()
    giveSweets(sweet,j)                                  # 分糖果
```

6. 运行结果

在PyCharm下运行程序，结果如图2.12所示。从输出结果可知，经过17次分糖过程后，10个孩子手中的糖果数相等，都为18块。

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/c
child  1    2    3    4    5    6    7    8    9   10
.....................................................
次数   糖果数
   0   10    2    8   22   16    4   10    6   14   20
   1   15    6    5   15   19   10    7    8   10   17
   2   17   11    6   11   18   15    9    8    9   14
   3   16   15    9    9   15   17   13    9    9   12
   4   14   16   13   10   13   17   16   12   10   11
   5   13   15   15   12   12   16   17   14   11   11
   6   13   15   16   14   12   14   17   16   13   12
   7   13   15   16   15   13   13   16   17   15   13
   8   14   15   16   16   15   14   15   17   17   15
   9   15   15   16   16   16   15   15   17   18   17
  10   17   16   16   16   16   16   16   17   18   18
  11   18   17   16   16   16   16   16   17   18   18
  12   18   18   17   16   16   16   16   17   18   18
  13   18   18   18   17   16   16   16   17   18   18
  14   18   18   18   18   17   16   16   17   18   18
  15   18   18   18   18   18   17   16   17   18   18
  16   18   18   18   18   18   18   17   17   18   18
  17   18   18   18   18   18   18   18   18   18   18

Process finished with exit code 0
```

图2.12　运行结果

2.7　爱因斯坦的数学题

1．问题描述

爱因斯坦出了一道这样的数学题：有一条长阶梯，若每步跨2阶，则最后剩一阶，若每步跨3阶，则最后剩2阶，若每步跨5阶，则最后剩4阶，若每步跨6阶，则最后剩5阶。只有每次跨7阶，最后才正好一阶不剩。请问在1到n内，有多少个数能满足？

2．问题分析

根据题意，用变量x表示阶梯数，则阶梯数x应该同时满足以下条件：

- 若每步跨2阶，则最后剩1阶，即x%2=1。
- 若每步跨3阶，则最后剩2阶，即x%3=2。
- 若每步跨5阶，则最后剩4阶，即x%5=4。
- 若每步跨6阶，则最后剩5阶，即x%6=5。
- 若每步跨7阶，最后才正好一阶不剩，即x%7=0。

3．算法设计

该问题要求输入n值，求解出在1-n的范围内存在多少个满足要求的阶梯数。在算法设计中，我们使用while循环，将循环条件设置为True，以允许重复读入多个n值。

对每一次读入的n值，都要判断在1-n的范围内存在的满足要求的阶梯数的个数。判断时可采用for循环，循环变量设为i，由题意，i的初值从7开始取即可；循环条件为i<n；循环体中则使用问题分析中列出的5个条件来检验每一个i值，能够满足所有5个条件的i值即为所求的阶梯数。

4. 确定程序框架

由上述分析可知，该程序的主体是一个循环结构。

1）输入n值。

```
while True:
    n = int(input("请输入n值："))
```

2）找到满足要求的阶梯数。

```
for i in range(7, n+1):
    # 判断条件
```

使用for循环检查每一个i值是否满足判断条件。

程序流程图如图2.13所示。

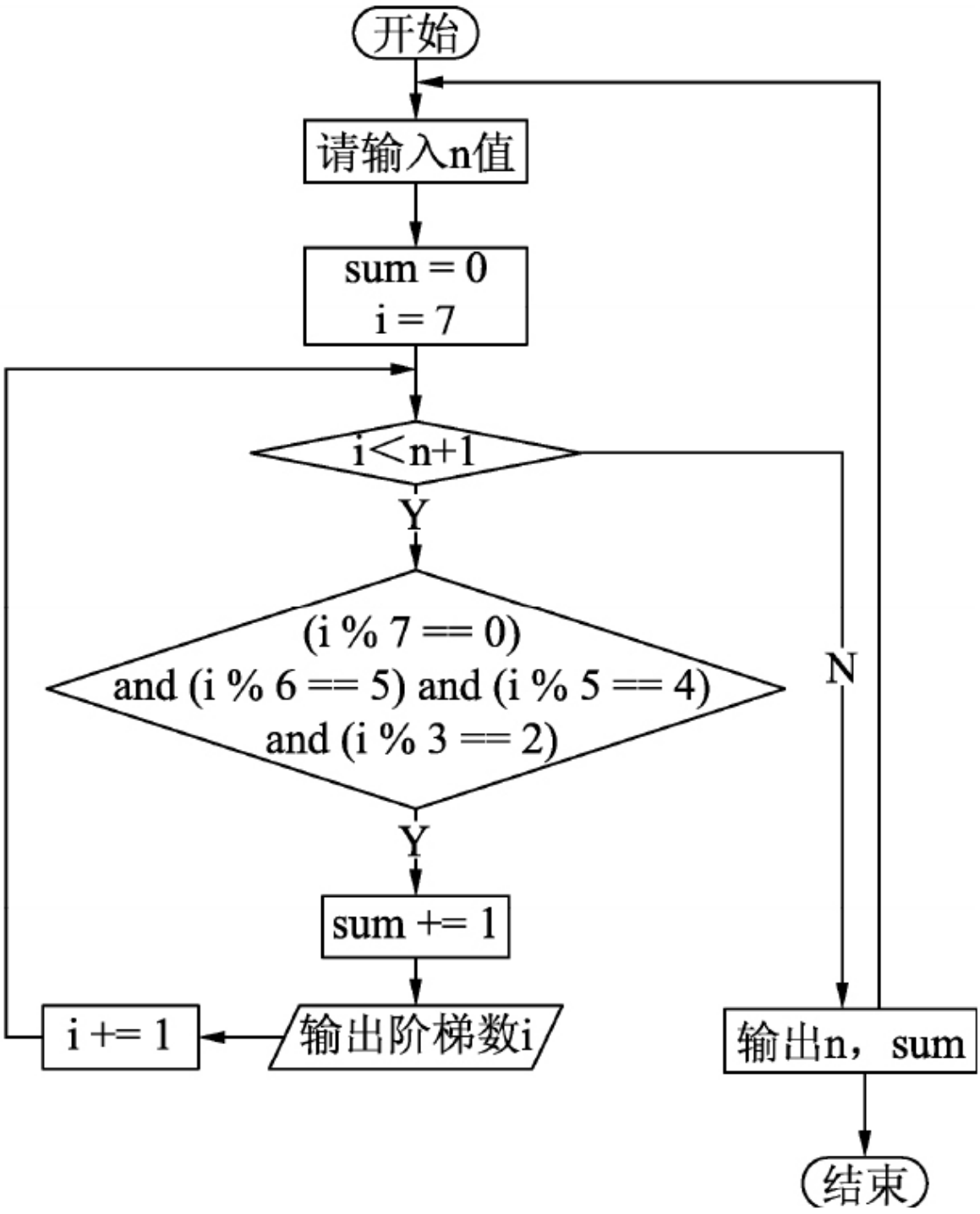


图2.13　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
# @author : liuhefei
# @desc: 爱因斯坦的数学题

def computing_ladder(n):
    print("在1-%d之间的阶梯数为：" %n)
    sum = 0
    for i in range(7, n+1):
        # 阶梯数所满足的条件
        if (i % 7 == 0) and (i % 6 == 5) and (i % 5 == 4) and (i % 3 == 2):
            sum += 1                                    # sum记录1～n之间满足条件的阶梯
个数
            print("%d" %i)
    print("在1-%d之间，有%d个数可以满足爱因斯坦对阶梯的要求。" %(n, sum))

if __name__=="__main__":
    while True:
        n = int(input("请输入n值："))
        computing_ladder(sum, n)
```

6. 运行结果

在PyCharm下运行程序，结果如图2.14所示。由输出结果可知，在1～200之间满足条件的阶梯数只有1个，为119；在1～400之间满足条件的阶梯数有2个，为119和329；在1～600之间满足条件的阶梯数有3个，为119、329和539。

```
E:\code\python\Interest-python\venv\Scripts\python.exe
请输入n值：200
在1-200之间的阶梯数为：
119
在1-200之间，有1个数可以满足爱因斯坦对阶梯的要求。
请输入n值：400
在1-400之间的阶梯数为：
119
329
在1-400之间，有2个数可以满足爱因斯坦对阶梯的要求。
请输入n值：600
在1-600之间的阶梯数为：
119
329
539
在1-600之间，有3个数可以满足爱因斯坦对阶梯的要求。
请输入n值：
```

图2.14　运行结果

2.8　猜牌术

1．问题描述

魔术师利用一副牌中的13张黑桃，预先将它们排好后叠在一起，并使牌面朝下。然后他对观众说：我不看牌，只要数数就可以猜到每张牌是什么，我大声数数，你们听，不信你们就看。魔术师将从最上面的一张牌开始数，第一张把它翻过来正好是黑桃A，他将黑桃A放在桌子上，然后按顺序从上到下数手中的余牌，第二次数1、2，将第一张牌放在这叠牌的下面，将第二张牌翻过来，正好是黑桃2，也将它放在桌子上，第三次数1、2、3，将前面两张依次放在这叠牌的下面，再翻第三张牌正好是黑桃3，这样依次进行，将13张牌全部翻出来，准确无误。问魔术师手中的牌原始次序是怎样安排的？

2．问题分析

先根据题目描述来分析题意。题目中描述的内容比较多，但已经将魔术师出牌的过程描述得很清楚了。

假设桌子上有13个空盒子排成一圈，设定其中一个盒子序号为1，将黑桃A放入1号盒子中，接着从下一个空盒子开始重新计数，当数到第2个空盒子时，将黑桃2放入其中。然后再从下一个空盒子开始重新计数，数到第3个空盒子时，将黑桃3放入其中，这样依次进行下去，直到将13张牌全部放入空盒子中为止。需要注意的是，在计数过程中要跳过那些已放入牌的盒子，而只对空盒子计数。最后牌在盒子中的顺序就是魔术师手中牌的顺序。

3．算法分析

根据问题分析，使用循环结构来实现程序。使用程序将分析过程模拟出来，就可以计算出魔术师手中的牌的原始次序。由于有13

张牌，因此显然要循环13次，每次循环时找到与牌序号对应的那个空盒子，因此循环体完成的功能就是找到对应的空盒子将牌存入。

4．确定程序框架

先定义数组a[14]用于存放13张牌，即相当于问题分析中假定的盒子。

定义变量i、j和n，其中i表示牌的序号，j表示数组的下标（盒子的序号），n用来记录当前的空盒序号，初值为1。

程序的主体结构为for循环语句，在for循环中实现将13张牌放入数组a的功能。

1）程序主框架如下：

```
#外循环13次，每次将一张牌放入空盒中
for i in range(1, 14):                                  # i表示牌的序号
    # n用来记录当前的空盒序号，初值为1
    n = 1                                                   # 每次都从一个空盒开始重新计数
    while n <= i:
        # 如果盒子非空，继续找下一个盒子
        # 如果盒子为空，判断盒子序号与牌的序号是否相同，相同则存入，不同则继续找
```

2）将i号牌放入空盒。

该功能使用while循环结构实现，代码如下：

```
while n <= i:
    if j > 13:
        j = 1
    if a[j]:                                                # 盒子非空，跳过该盒子
        j += 1
    else:
        if n == i:                                          # 判断该盒子是否为第i
个空盒
            a[j] = i                                    # 是则将i存入
        j += 1
        n += 1
```

程序流程图如图2.15所示。

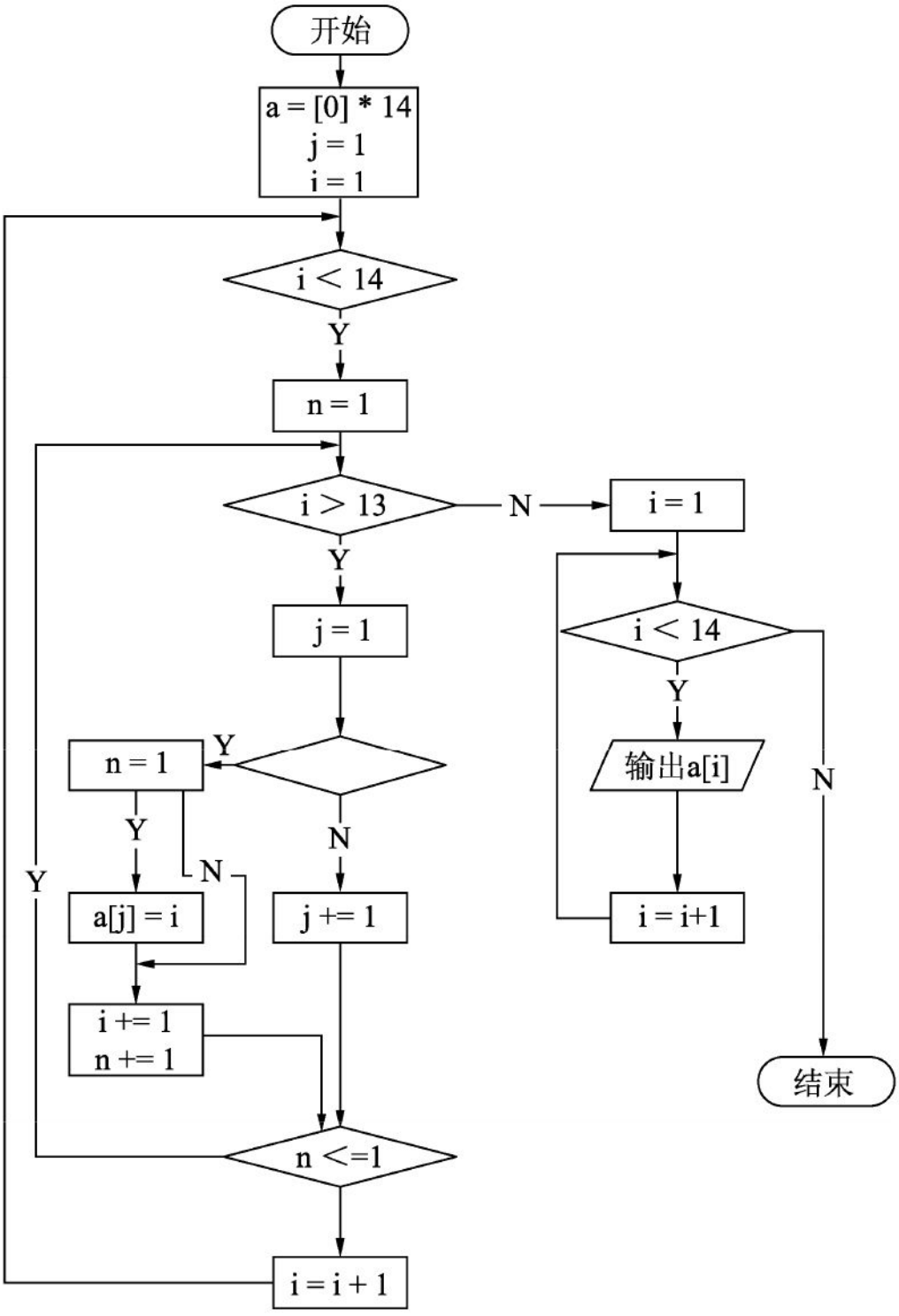


图2.15　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc:  猜牌术

if __name__ == '__main__':
    a = [0] * 14                                    # 初始化列表，用来存放
13张牌
    j = 1                                           # j是数组的
下标，空盒子的序号
    print("魔术师手中的牌原始次序是:")
    # 外循环13次，每次将一张牌放入空盒中
    for i in range(1, 14):                          # i表示牌的序号
        # n用来记录当前的空盒序号，初值为1
        n = 1                                       # 每次都从一个空盒开始
重新计数
        while n <= i:
            if j > 13:
                j = 1
            if a[j]:                                # 盒子非空，跳过该盒子
                j += 1
            else:
                if n == i:                          # 判断该盒子是否为第i
个空盒
                    a[j] = i                        # 是则将i存入
                j += 1
                n += 1

    print(a[1:])
```

6. 运行结果

在PyCharm下运行程序，结果如图2.16所示。由输出结果可知，魔术师手中的牌的原始次序是1，8，2，5，10，3，12，11，9，4，7，6，13。

```
E:\code\python\Interest-python\venv\Scripts\python.exe
魔术师手中的牌原始次序是:
[1, 8, 2, 5, 10, 3, 12, 11, 9, 4, 7, 6, 13]

Process finished with exit code 0
```

图2.16　运行结果

2.9　舍罕王的失算

1．问题描述

相传国际象棋是古印度舍罕王的宰相达依尔发明的。舍罕王十分喜爱象棋，决定让宰相自己选择何种赏赐。这位聪明的宰相指着8×8共64格的象棋棋盘说：陛下，请您赏给我一些麦子吧，就在棋盘的第1格中放1粒，第2格放2粒，第3格放4粒，以后每一格都比前一格增加一倍，依次放完棋盘上的64格，我就感激不尽了。舍罕王让人扛来一袋麦子，他要兑现他的许诺。请编程求出国王总共需要将多少麦子赏赐给他的宰相。

2．问题分析

该问题描述比较复杂，但只要抽象出其数学模型，便很容易解决了。

根据题意，麦子的放法是：在棋盘的第1格中放1粒，第2格放2粒，第3格放4粒，以后每一格都比前一格增加一倍，依次放完棋盘上的64格。

由此可推知，按照如此放法可得到的麦子的总数为：

$$1+2+4+8+16+32+\cdots+2^{63}=\sum_{i=1}^{64}2^{i-1}$$

这样，就从问题描述中抽象出了数学模型，据此模型就可以完成算法设计和程序的编写了。

3．算法设计

在问题分析中已经将所需麦子的总数抽象为数学公式$\sum_{i=1}^{64}2^{i-1}$，现在只要考虑如何设计算法实现累加和即可。显然，可采用循环结构，每循环一次就实现一次累加，总共循环64次可获得累加和。

4. 确定程序框架

程序流程图如图2.17所示。

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 舍罕王的失算

if __name__=="__main__":
    # 使用循环求累加和
    i = 1
    sum = 0.0
    while i <= 64:
        sum = sum + 2**(i-1)
        # print("sum" , i , "=" , sum)
        i += 1
    print("国王总共需要赏赐给宰相的麦子数为：\n%f\n" %sum)            # 打印结果
```

6. 运行结果

在PyCharm下运行程序，结果如图2.18所示。由输出结果可知，国王总共需要赏赐给宰相的麦子数为18446744073709551616.000000。

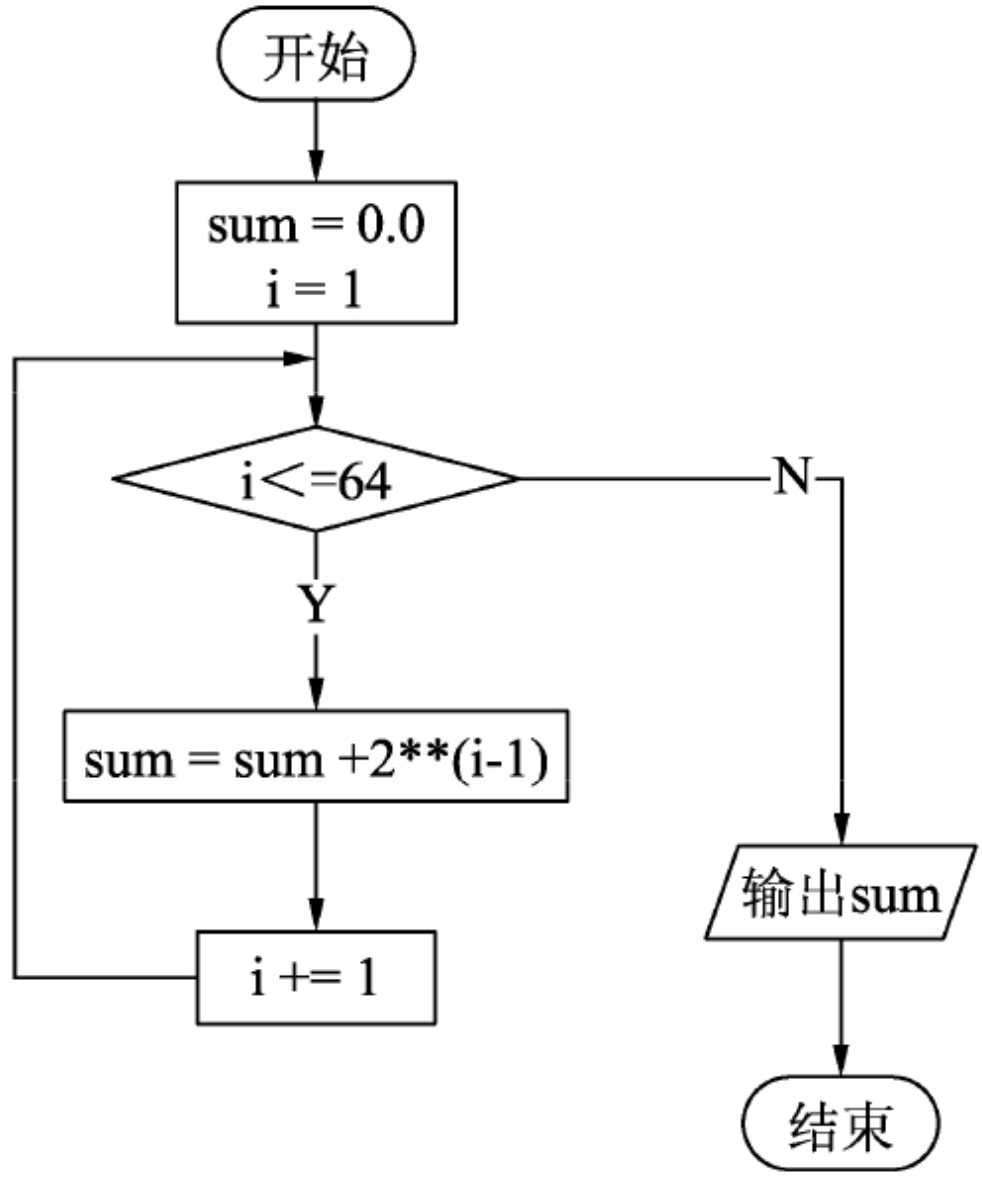


图2.17　程序流程图

```
E:\code\python\Interest-python\venv
国王总共需要赏赐给宰相的麦子数为:
18446744073709551616.000000


Process finished with exit code 0
```

图2.18　运行结果

2.10　马克思手稿中的数学题

1．问题描述

马克思手稿中有一道趣味数学问题：有30个人，其中有男人、女人和小孩，他们在同一家饭馆吃饭，总共花了50先令。已知每个男人吃饭需要花3先令，每个女人吃饭需要花2先令，每个小孩吃饭需要花1先令，请编程求出男人、女人和小孩各有几人。

2．问题分析

根据该问题的描述，可将该问题抽象为一个不定方程组。

设变量x、y和z分别代表男人、女人和小孩，则由题目的要求，可得到如下的方程组：

$$\begin{cases} x+y+z=30 & ① \\ 3x+2y+z=50 & ② \end{cases}$$

其中，方程①表示男人、女人和小孩加起来总共有30个人；方程②表示30个人吃饭总共花了50先令。

用方程②-方程①，可得：

2x+y=20　③

由方程③可知，x取值范围为[0,10]。

3．算法设计

在问题分析中，我们抽象出了一个不定方程组，显然得到了不定方程组的解，该问题也就解决了。但不定方程组中包含了x、y、z三个变量，而方程只有两个，因此不能直接求出x、y、z的值。

而由方程③，我们得到了x的取值范围，因此可将x的有效取值依次代入不定方程组中（即方程①、②、③）中，能使三个方程同时成立的解即为该问题的解。为实现该功能，只需使用一个for循环语句即可。

4．确定程序框架

不定方程组的求解过程代码如下：

```
# 将变量x的可能取值依次代入方程组
for x in range(0, 10+1):
    y = 20 - 2*x             # 方程③，当x一定时，可确定y值
    z = 30 -x - y    # 方程①，当x、y一定时，可确定z值
    # 代入方程②检验，当前获得的x、y、z是否为不定方程组的
      一组解
    if 3*x + 2*y + z == 50:
        number += 1
        print("%2d:%4d%5d%6d" % (number, x, y, z))
```

上面的代码中对于for循环中每个给定的x值，可先通过方程③确定y值，再将当前确定的x和y值代入方程①中确定z值。此时，变量x、y、z都有确定的值了，但它们的值的组合不一定是不定方程组的解，还需要代入方程②中进行验证，只有同时满足方程①、②、③的解才是不定方程组的解。

程序流程图如图2.19所示。

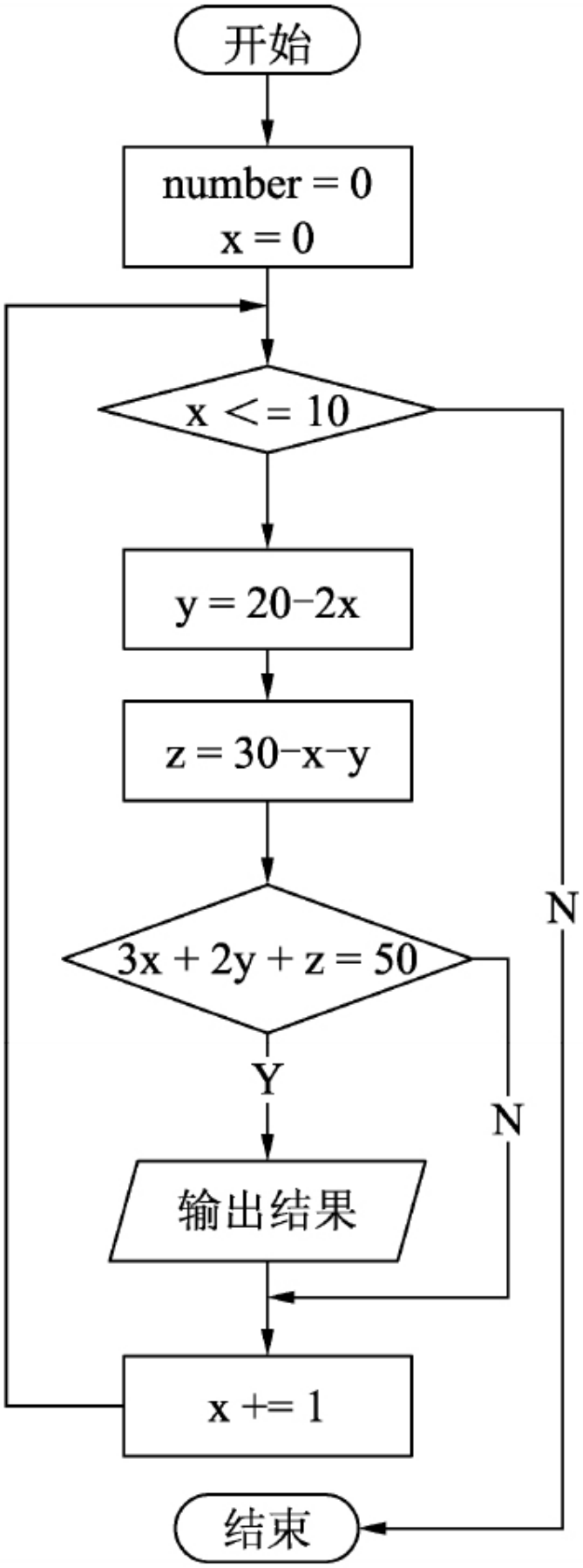

开始
number = 0
x = 0
x <= 10
y = 20−2x
z = 30−x−y
3x + 2y + z = 50
Y
N
N
输出结果
x += 1
结束

图2.19　程序流程图

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 马克思手稿中的数学题

if __name__=="__main__":
    # 变量x、y和z分别代表男人、女人和小孩
    print("    Men  Women  Children ")
    number = 0                                          # 可能的值的组数
    # 将变量x的可能取值依次代入方程组
    for x in range(0, 10+1):
        y = 20 - 2*x                                    # 方程③，当x一定时，可确定y值
        # 方程①，当x、y一定时，可确定z值
        z = 30 -x - y
        # 代入方程②检验，当前获得的x、y、z是否为不
          定方程组的一组解
        if 3*x + 2*y + z == 50:
            number += 1
            print("%2d:%4d%5d%6d" % (number, x,
y, z))
```

6. 运行结果

在PyCharm下运行程序，结果如图2.20所示。由输出结果可知，该不定方程组的解共有11组，即男人、女人和小孩的可能组合共有11种情况。

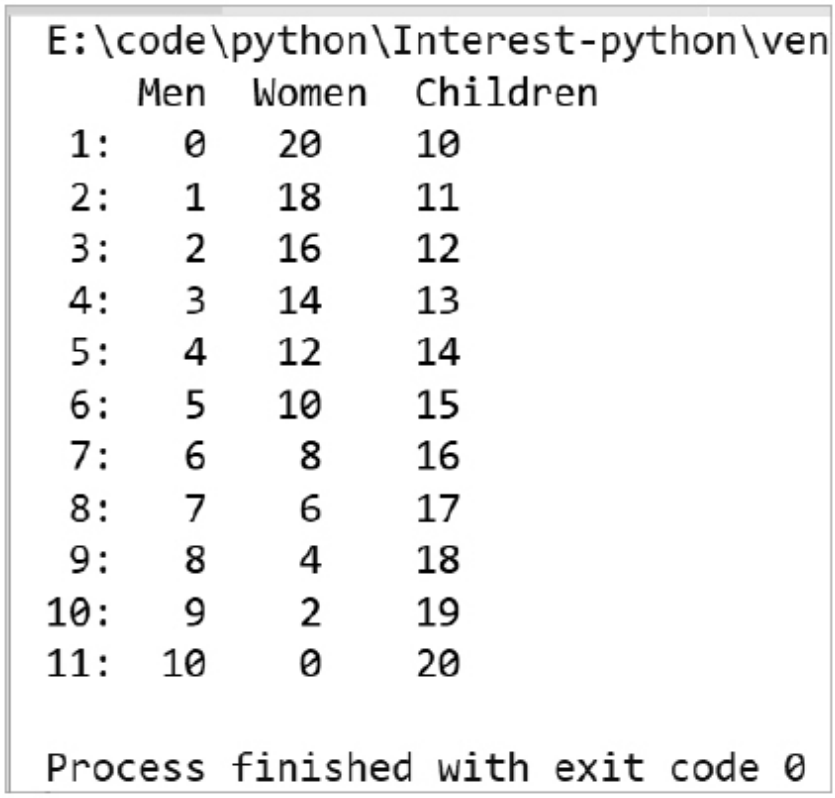

E:\code\python\Interest-python\ven
Men Women Children
1: 0 20 10
2: 1 18 11
3: 2 16 12
4: 3 14 13
5: 4 12 14
6: 5 10 15
7: 6 8 16
8: 7 6 17
9: 8 4 18
10: 9 2 19
11: 10 0 20
Process finished with exit code 0

图2.20　运行结果

2.11　换分币

1．问题描述

将5元的人民币兑换成1元、5角和1角的硬币，共有多少种不同的兑换方法。

2．问题分析

根据该问题的描述，可将该问题抽象为一个不定方程。

设变量x、y和z分别代表兑换的1元、5角和1角的硬币所具有的钱数（角），则由题目的要求，可得到如下的方程：

x+y+z=50

其中，x为兑换的1元硬币钱数，其可能的取值为{0,10,20,30,40,50}；y为兑换的5角硬币钱数，其可能的取值为{0,5,10,15,20,25,30,35,40,45,50}；z为兑换的1角硬币钱数，其可能的取值为{0,1,…50}。

3．算法设计

在问题分析中，我们得到了一个不定方程，显然该不定方程会有多组解。根据题意可知x、y和z的可能取值，将它们所有可能取值的组合代入方程中，能使该方程成立的那些解即为该问题的解。

为实现该功能，需要使用三个嵌套的for循环语句。

4．确定程序框架

程序流程图如图2.21所示。

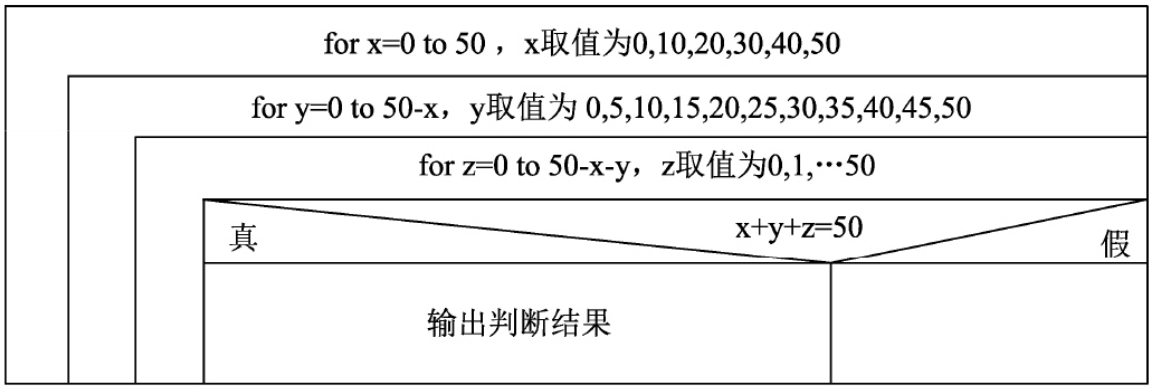


图2.21　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc:  换分币

if __name__=="__main__":
    # 变量x、y和z分别代表兑换的1元、5角和1角的硬币所具有的钱数（角）
    count = 0                                                                    #
计数器
    print("可能的兑换方法如下：")
    # x 为兑换的1元硬币钱数，可能的取值为{0,10,20,30,40,50}
    for x in range(0, 50+1, 10):
        # y为5角硬币钱数，其取值为{0,5,10,15,20,25,30,35,40,45,50}
        for y in range(0, 50-x+1, 5):
            # z为1角硬币钱数，其取值为{0,1,...50}
            for z in range(0, 50-x-y+1, 1):
                if(x + y + z == 50):
                    count += 1
                    if count % 3 == 0:              # 每3列一行
                        print(count, end="   ")
                        print("10*%d+5*%d+1*%d \t" % (x // 10, y // 5, z))
                    else:
                        print(count, end=" ")
                        print("10*%d+5*%d+1*%d \t" % (x // 10, y // 5, z),
end=" ")
```

6．运行结果

在PyCharm下运行程序，结果如图2.22所示。由输出结果可知，可能的兑换方法有36种。

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/code/
可能的兑换方法如下：
1 10*0+5*0+1*50      2 10*0+5*1+1*45      3   10*0+5*2+1*40
4 10*0+5*3+1*35      5 10*0+5*4+1*30      6   10*0+5*5+1*25
7 10*0+5*6+1*20      8 10*0+5*7+1*15      9   10*0+5*8+1*10
10 10*0+5*9+1*5      11 10*0+5*10+1*0     12   10*1+5*0+1*40
13 10*1+5*1+1*35     14 10*1+5*2+1*30     15   10*1+5*3+1*25
16 10*1+5*4+1*20     17 10*1+5*5+1*15     18   10*1+5*6+1*10
19 10*1+5*7+1*5      20 10*1+5*8+1*0      21   10*2+5*0+1*30
22 10*2+5*1+1*25     23 10*2+5*2+1*20     24   10*2+5*3+1*15
25 10*2+5*4+1*10     26 10*2+5*5+1*5      27   10*2+5*6+1*0
28 10*3+5*0+1*20     29 10*3+5*1+1*15     30   10*3+5*2+1*10
31 10*3+5*3+1*5      32 10*3+5*4+1*0      33   10*4+5*0+1*10
34 10*4+5*1+1*5      35 10*4+5*2+1*0      36   10*5+5*0+1*0

Process finished with exit code 0
```

图2.22　运行结果

第3章　各种趣味整数

整数通常是程序设计语言的一种基础形态，例如Java及C编程语言的int类型。整数问题是实际应用中经常要解决的问题。整型数据根据其所占内存大小可分为基本整型（int）、长整型（long int）、短整型（short int），根据数据满足的某些性质又可将其分为完全数、水仙花数和亲密数等。整数问题中经常用到的是对数据的拆分和组合，初学者一定要从实例中总结方法并掌握。在Python语言中，将不同的数值类型统称为Number类型，Python 3只支持整型（int）、浮点型（float）和复数（complex）三种数值类型。在Python 3中，整型没有限制大小，可以当作long类型使用，所以Python 3中没有long类型。本章主要通过对各类整数问题的算法进行讲解，培养读者这方面的思维能力与编程技巧。

3.1　回文数

1．问题描述

打印所有不超过n（取n<256）的其平方具有对称性质的数（也称回文数）。

2．问题分析

对于要判定的数n，计算出其平方后（存于a），按照“回文数”的定义要将最高位与最低位、次高位与次低位等进行比较，若彼此相等则为回文数。此算法需要知道平方数的位数，再一一将每一位分解并比较，此方法对于位数已知且位数不是太多的数来说比较适用。

此问题可借助数组来解决。将平方后的数值（a）的每一位进行分解，按从低位到高位的顺序依次暂存到数组中，再将数组中的元素按照下标从大到小的顺序重新将其组合成一个数k（如n=15，则a=225且k=522），若a等于k则可判定n为回文数。

3．算法设计

从低位到高位将某个整数拆分。对于一个整数（设变量名为a），无论其位数多少，若欲将最低位拆分则只需对10进行求模运算，即a%10；拆分次低位首先要想办法将原来的次低位作为最低位来处理，用原数对10求商可得到由除最低位之外的数形成的新数，且新数的最低位是原数的次低位，根据拆分最低位的方法将次低位求出，即先进行a//10运算，后进行a%10运算；对于其他位上的数算法相同。利用这个方法要解决的一个问题是，什么情况下才算把所有数都拆分完了呢？当拆分到只剩原数最高位时（即新数为个位数时），再对10求商的话，得到的结果肯定为0，可通过这个条件判断是否拆分完毕。根据题意，应将每次拆分出来的数据存储到数组

中，原数的最低位存到下标为0的位置，次低位存到下标为1的位置，以此类推。程序段如下：

```
i = 0
while a != 0:                                   # 从低位到高位分解数a的每一位，存于数组m[1]～
m[16]
    m[i] = a % 10
    a //= 10
    i += 1
```

将数组中元素重新组合成一个新数。拆分时变量a的最高位仍然存储在数组中下标最大的位置，根据“回文数”定义，新数中数据的顺序与a中数据的顺序相反，所以我们按照下标从大到小的顺序分别取出数组中的元素组成新数k。由几个数字组成一个新数时只需用每一个数字乘以所在位置对应的权值，然后相加即可，在编程过程中应该有一个变量t来存储每一位对应的权值，个位权值为1，十位权值为10，百位权值为100，以此类推，所以可以利用循环，每循环一次，t的值就扩大10倍。对应的程序段如下：

```
while i > 0:
    k += m[i-1] * t # t记录某一位置对应的权值
    t *= 10
    i -= 1
```

4. 确定程序框架

程序的流程图如图3.1所示。

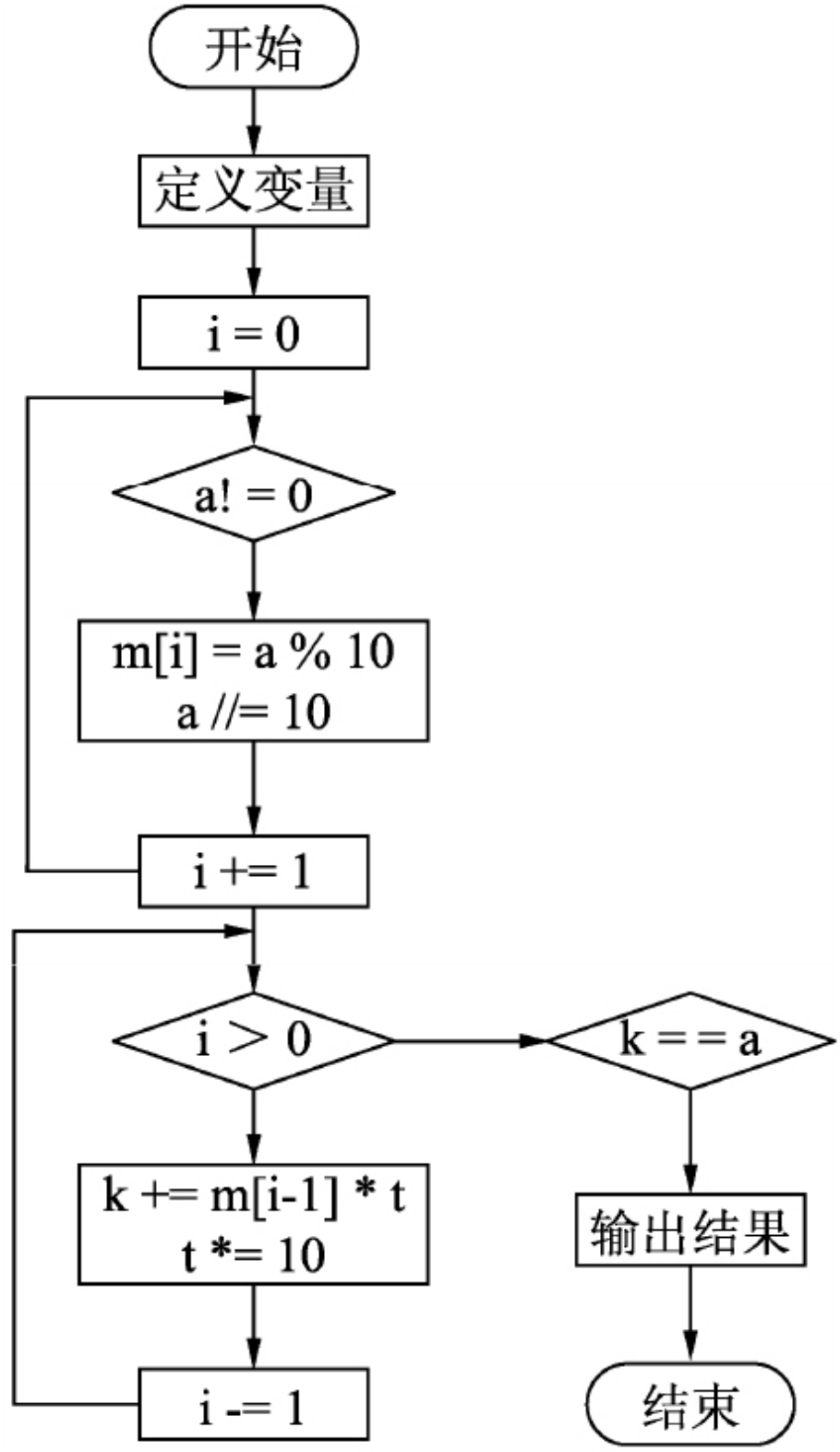


图3.1　程序流程图

5．完整的程序

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 回文数

if __name__ == '__main__':
    m = [1] * 17
    count = 0
    print("No.    number      it's square(palindrome)")
    for n in range(1, 256):                                          # 穷举n的取值范围
        k, i, t, a = 0, 0, 1, n*n                          # 计算n的平方
        squ = a
        while a != 0:                                 # 从低到高分解数a的每一位存于数组m[1]～
m[16]
            m[i] = a % 10
            a //= 10
            i += 1

        while i > 0:
            k += m[i-1] * t          # t记录某一位置对应的权值
            t *= 10
            i -= 1

        if k == squ:
            count += 1
            print("%2d%10d%10d" % (count, n, n*n))
```

---

6. 运行结果

在PyCharm下运行程序，结果如图3.2所示。

```
E:\code\python\Interest-python\venv\Scripts\
No.    number      it's square(palindrome)
 1         1          1
 2         2          4
 3         3          9
 4        11        121
 5        22        484
 6        26        676
 7       101      10201
 8       111      12321
 9       121      14641
10       202      40804
11       212      44944

Process finished with exit code 0
```

图3.2　运行结果

7．问题拓展

在上面的问题分析中，提到另一种判断“回文数”的方法，就是将数据中每一位的数分离出来，然后比较对称位置上的数据，若相等，则此数是“回文数”。此方法适合于对一个整数进行判断。

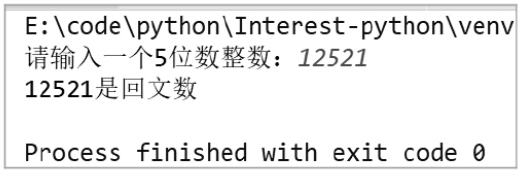
编程实现输入一个5位数，判断它是不是回文数，例如12321是回文数，个位与万位相同，十位与千位相同。

完整的程序如下：

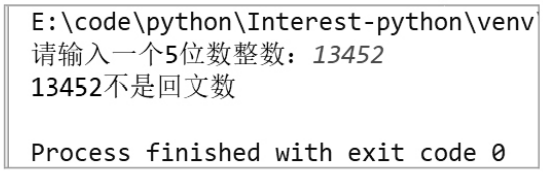
```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 回文数


if __name__ == '__main__':
    x = int(input("请输入一个5位数整数："))
    if x < 10000 or x > 99999:
        print("输入错误")
    else:
        ten_thousand = x // 10000                                    #拆分最高位万
位
        thousand = x % 10000 // 1000                            #拆分千位
        ten = x % 100 // 10                                          #拆分十位
        indiv = x % 10                                                    #拆
分个位
        if indiv == ten_thousand and ten == thousand:
            print("%d是回文数" %x)
        else:
          print("%d不是回文数" %x)
```

对于本题来说，给定的是一个5位数，对于中间位置的百位不需要再进行分离，因为它不与任何其他位置进行比较。但对偶数位的整数进行判断时，所有位置都要分离出来。在编程过程中除了保证程序的正确性外，效率也是很重要的。

运行程序，分别输入12521和13452两个整数，判断它们是否为回文数，结果如图3.3所示。



a）运行结果 1



b）运行结果 2

图3.3　运行结果

3.2　水仙花数

1．问题描述

输出所有的“水仙花数”。所谓的“水仙花数”是指一个三位数，其各位数字的立方和等于该数本身，例如，153是“水仙花数”，因为153=1^3 +1^3 +3^3 。

2．问题分析

根据“水仙花数”的定义，判断一个数是否为“水仙花数”最重要的是要把给出的三位数的个位、十位和百位分别拆分，并求其立方和（设为s），若s与给出的三位数相等，则该三位数为“水仙花数”，反之，则不是。

3．算法设计

“水仙花数”是指满足某一条件的三位数，根据这一信息可以确定整数的取值范围是100～999。对应的循环条件如下：

```
for n in range(100, 1000):
    …
```

1）将n整除以100，得出n在百位上的数字hun。

2）将（n-hun×100）整除以10（或将n先整除以10再对10求模，即n//10%10），得出n在十位上的数字ten。

3）将n对10取余，得出n在个位上的数字ind。

4）求得这三个数字的立方和是否与其本身相等，若相等，则该数为“水仙花数”。

对于每个位置上的数值将其拆分的算法有很多种，应根据不同情况选择不同的算法（对于同一问题不同算法的效率有时会相差很

多）。

4. 确定程序框架

程序流程图如图3.4所示。

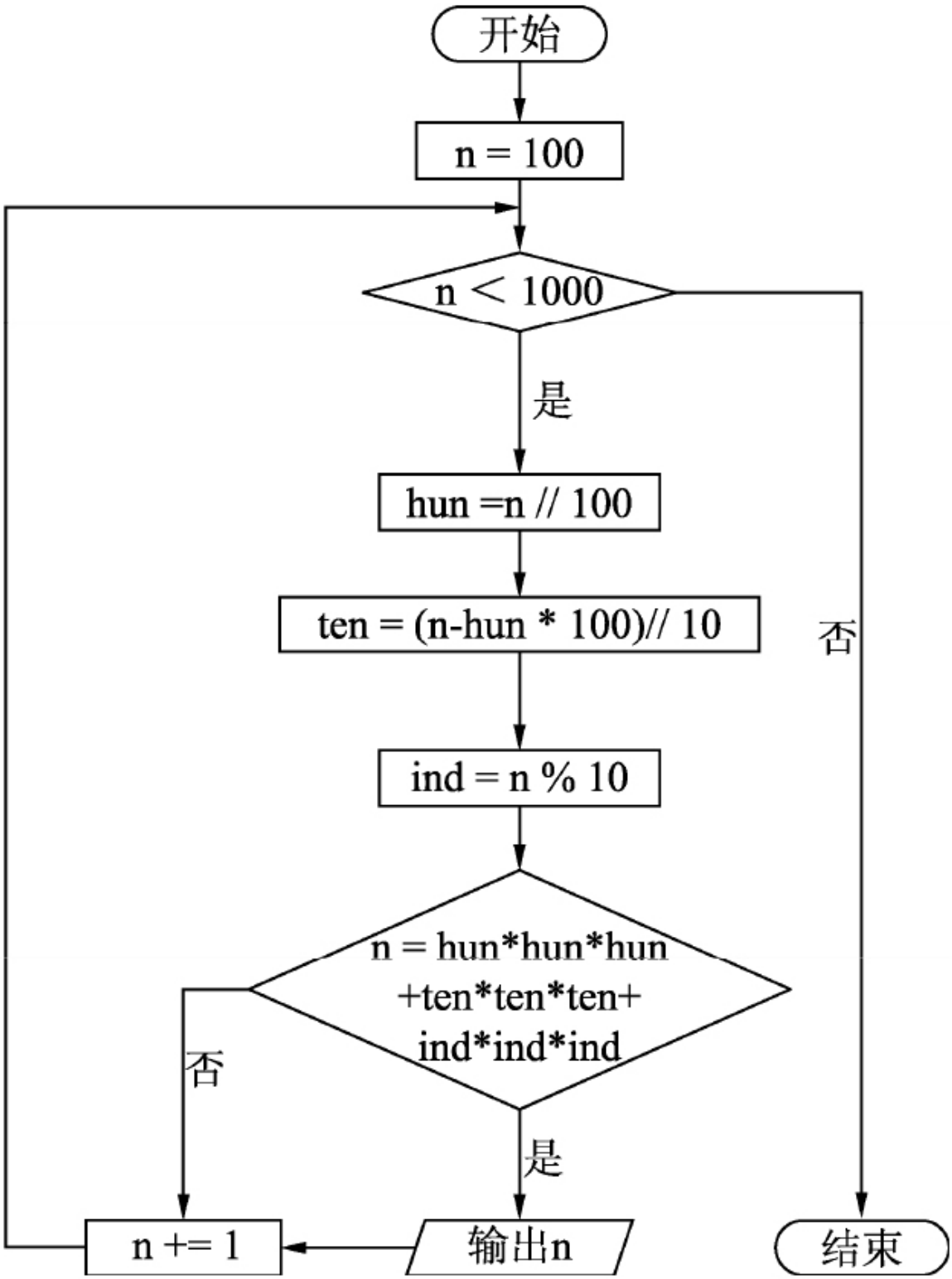


图3.4　程序流程图

5. 完整的程序

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 水仙花数

if __name__ == '__main__':
    print("result is: ")
    # 整数的取值范围
    for n in range(100, 1000):
        hun = n // 100          # 百位
        ten = (n - hun * 100) // 10                    # 十位
        ind = n % 10                                                    # 个位
        m = hun*hun*hun + ten*ten*ten + ind*ind*ind          # 求和
        if n == m:                                                              #
各位上的立方和是否与原数n相等
            print("%d \t" %n, end=" ")
```

6. 运行结果

程序运行结果如图3.5所示。

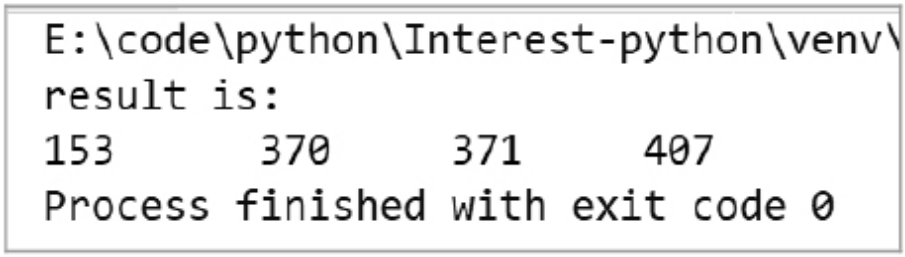


图3.5　运行结果

7. 问题拓展

求某个数n的立方，可以采用程序中的方法对n连乘三次n×n×n，求五次方、十次方运用这种方法仍可忍受，但如果要求的是n的50次方甚至更大呢？也要像上面一样写50次吗？对于编程者来说这是很痛苦的一种事情，既浪费时间又浪费精力。因此，Python语言为我们提供了幂运算符“**”，例如求5的3次方，可以表示为5**3，故我们可以换一种写法，程序代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 水仙花数

if __name__ == '__main__':
    print("result is: ")
    for n in range(100, 1000):                          # 整数的取值范围
        hun = n // 100                                                  # 百位
        ten = (n - hun * 100) // 10                     # 十位
        ind = n % 10                                                    # 个位
        m = hun**3 + ten**3 + ind**3
        if n == m:                                                              #
各位上的立方和是否与原数n相等
            print("%d \t" %n, end=" ")
```

3.3　阿姆斯特朗数

1．问题描述

如果一个整数等于其各个数字的立方和，则该数称为“阿姆斯特朗数”（亦称为自恋性数）。如$153=1^3+5^3+3^3$就是一个“阿姆斯特朗数”。试编程求1000以内的所有“阿姆斯特朗数”。

2．问题分析

“阿姆斯特朗数”与3.2节中的“水仙花数”的不同在于，前者并没有规定几位数，从两者的定义来看，“水仙花数”可以看作是“阿姆斯特朗数”的一个子集。对于这类问题的算法与“水仙花数”类似，需要把每一位分离出来，然后比较其立方和与原数是否相等。

3．算法设计

本题求的是1000以内满足条件的数，从数的位数来说可以分为一位数、两位数及三位数。这样可以根据数的位数不同分别求出不同范围内的“阿姆斯特朗数”，即一位数（1～9）、两位数（10～99）和三位数（100～999）分别用一个循环语句来实现。

上述方法有其局限性，如果题目改成求1 000 000以内甚至更大范围的话，程序里面会有多个循环，不但程序看起来烦琐，写起来也费事，更重要的是每个循环体做的事情都是一样的：将数的每一位拆分。对于这种重复的事情可以考虑用循环将其简化。对于一个数无论它的位数是多少，如果要将其拆分，要么按从低位到高位的顺序，要么按从高位到低位的顺序。本题按从低位到高位的顺序进行拆分。

从低位到高位进行拆分，每次拆分的都是当前数的个位，可以用当前数n对10求模，即n%10，这样最后一位就被分离出来；再次分

离的是原数的次低位，对于次低位，要想办法让其成为新数的最低位，可采用原数n对10求商的方法，即n//10。其他位置的数据求法同上。

题目给出的数据最多三位，我们可以定义三个变量分别来存储原数的个位、十位和百位，也可以用数组来存储，数组的长度为3。

4. 确定程序框架

程序流程图如图3.6所示。

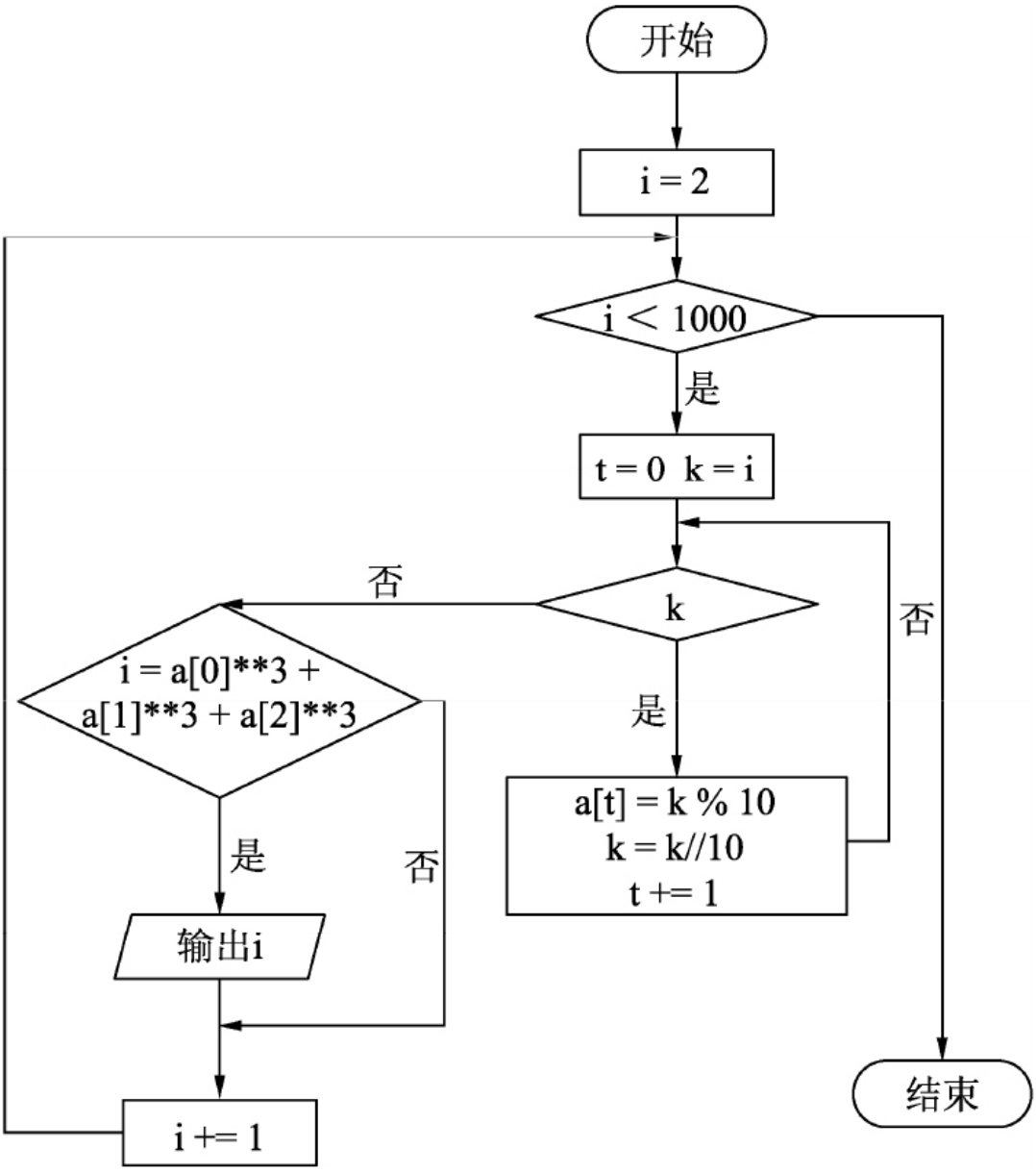


图3.6　程序流程图

5. 完整的程序

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 阿姆斯特朗数

if __name__ == "__main__":
    a = [0, 0, 0]                                    # 列表a用来存储被拆分的数的个位、十位和百位
    print("1000以内的阿姆斯特朗数：")
    for i in range(2, 1000):
        t = 0
        k = i
        # 按从低位到高位的顺序拆分数
        while k:
            a[t] = k % 10
            k = k // 10
            t += 1
        sum = a[0]**3 + a[1]**3 + a[2]**3
        if i == sum:
            print("%d \t " %i, end=" ")
```

6. 运行结果

在PyCharm中运行程序，结果如图3.7所示。

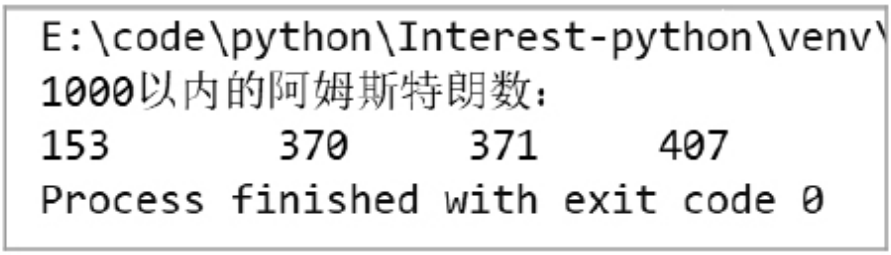


图3.7　运行结果

7. 问题拓展

本题程序采用的是从低位到高位的顺序进行分离。下面将从高位到低位顺序进行拆分的程序段给出，供读者参考。

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 阿姆斯特朗数

if __name__ == "__main__":
    a = [0, 0, 0]                                    # 列表a用来存储被拆分的数的个位、十位和百位
    print("1000以内的阿姆斯特朗数：")
    for i in range(2, 1000):
        # 按从高位到低位的顺序拆分数
```

```
t = 0
k = 1000
while k >= 10:
    a[t] = (i % k) // (k // 10)
    k = k // 10
    t += 1
sum = a[0]**3 + a[1]**3 + a[2]**3
if i == sum:                              # 判断是否为阿姆斯特朗数
    print("%d \t" %i, end=" ")
```

3.4 完数

1. 问题描述

求某一范围内完数的个数。

如果一个数等于它的因子之和，则称该数为“完数”（或“完全数”）。例如，6的因子为1、2、3，而6=1+2+3，因此6是“完数”。

2. 问题分析

根据完数的定义，解决本题的关键是计算出所选取整数i（i的取值范围不固定）的因子（因子就是所有可以整除这个数的数），然后将各因子累加到变量s（记录所有因子之和），若s等于i，则可确认i为完数，反之则不是完数。

3. 算法设计

对于这类求某一范围（本题范围不固定，在编程过程中采用键盘输入的方式）内满足条件的数的问题，一般采用遍历的方式，即一个一个地去判断给定范围内的数值是否满足条件，这一过程可利用循环来实现。

本题的关键是求出选取数值i的因子，即从1到i-1范围内能整除i的数。看某一个数j是不是i的因子，可利用语句“if i%j==0”进行判断，求某一个数的所有因子，需要在1到i-1范围内进行遍历，同样采用循环实现。因此，本题从整体上看可利用两层循环来实现。外层循环控制该数的范围为2～n；内层循环j控制除数的范围为1～i，通过判断i对j取余是否等于0，找到该数的各个因子。程序段如下：

```
i = 2                                    # 变量 i 控制选定数的范围
while i <= n:
    ...
```

```
for j in range(i):
    ...

if s == i:                                  # 判断因子之和是否和原数相等
   # 输出当前i是完数
i += 1
```

对于某个选定的数，将求得的各因子累加到变量s（累加过程中用到s的初值，故s初值为0）之后，s的值发生改变，若直接将下一个选定数的因子加到s上，得到的值并非所求（此时s的初值不是0而是上一个选定数的因子之和）。因此每次判断下一个选定数之前，必须将变量s的值重新置为0。编程过程中一定要注意变量s重新置0的位置，如果语句放的位置不正确，得到的结果也不是正确结果。

注意： Python语言中的整数问题经常涉及判断两个数是否相等或某变量（或表达式）是否满足某一条件的情况，对于这类问题，初学者经常会出现混用赋值符号“=”与等于号“==”的情况。

赋值符号“=”的优先级别低于其他的运算符，所以对该运算符往往最后读取。它的作用是将一个表达式的值赋给一个变量（左值）。左值必须能够被修改，不能是常量。如while i=10的作用是将右值“10”赋给左值i，每次判断i的值都为10，故表达式的值为非0，即判定条件为真，导致程序进入死循环。

等于号“==”是关系运算符的一种，结果只有True或False两种。它的作用是用来判断等号“==”两边参与运算的值是否相等，若相等，则返回True，否则返回False。如while i==10的作用是判断变量i的值是否等于10，若相等，表达式的值为True，否则为False，当表达式为真，程序继续执行循环体语句，否则结束循环。

4. 确定程序框架

程序流程图如图3.8所示。

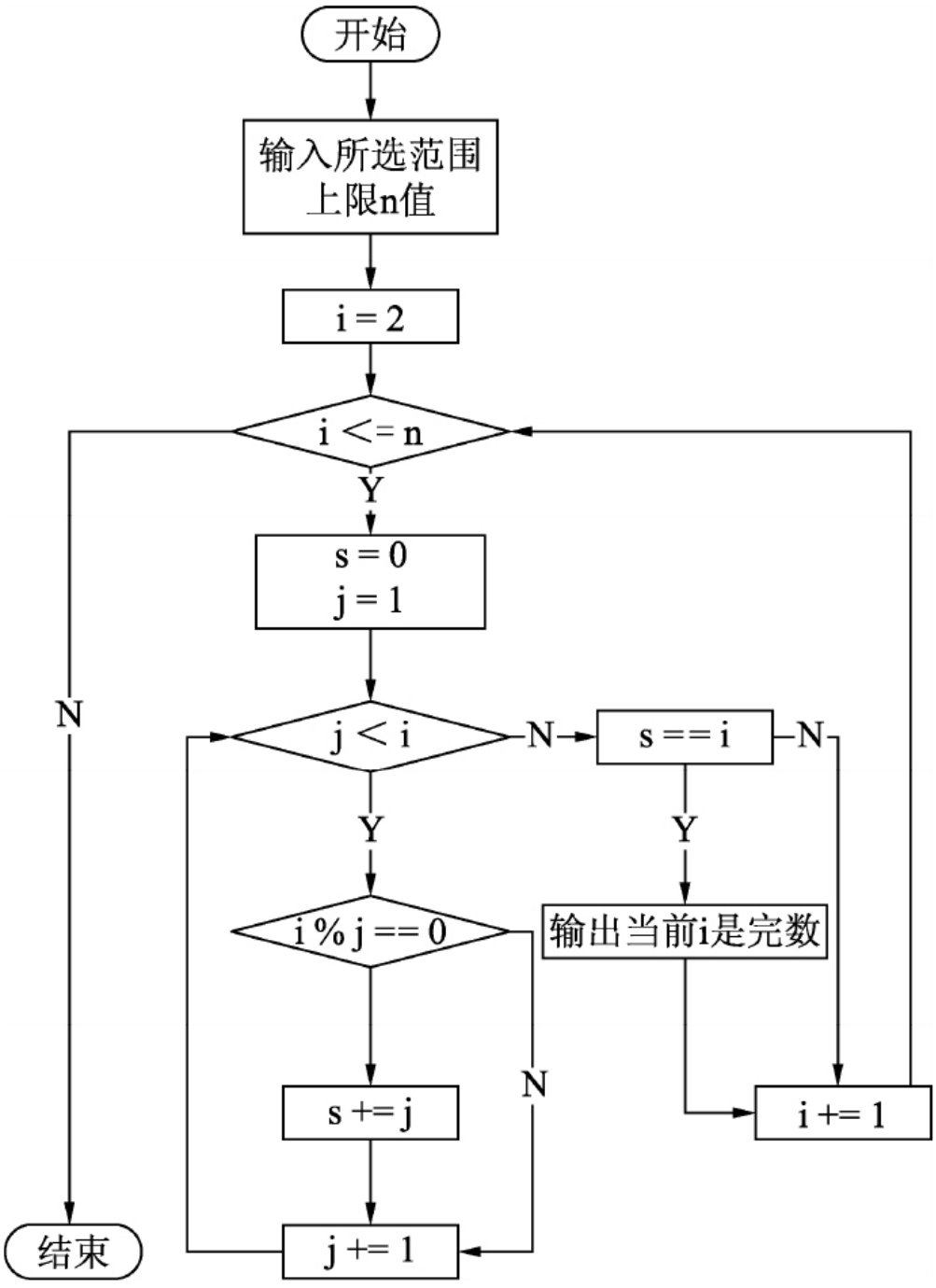


图3.8　程序流程图

5. 完整的程序

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 完数

if __name__=="__main__":
    n = int(input("请输入所选范围上限：  "))
    i = 2                                                    # 变量 i 控制选定数的范围
```

```
    while i <= n:
        s = 0                                                    # s记录累加因子之和，保证每次循
环时s的初值为0
        j = 1                                                    # j 控制除数范围
        for j in range(i):
            if j != 0 and i % j == 0:        # 判断 j 是不是 i 的因子
                s += j                                                  # 因子和

        if s == i:                                                      # 判断因子之和是否和原
数相等
            print("2到%d之间的完数：%d" %(n, i))
        i += 1
```

6. 运行结果

程序运行结果如图3.9所示。

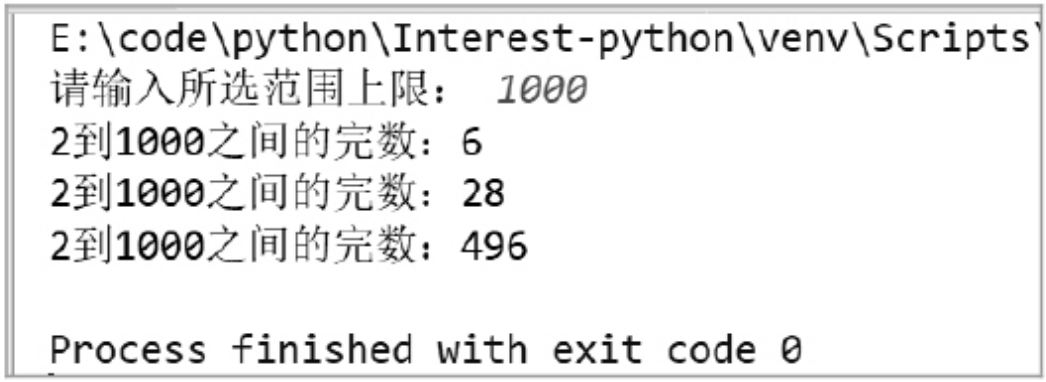


图3.9　运行结果

7. 问题拓展

上述程序中，求某数的因子时采用从1到i-1范围内进行遍历的方法，一个数一个数地去试。这种方法可以做到没有遗漏，但是效率不高。

对于某一整数来说，其最大因子为n/2（此时n为偶数，若n为奇数，其最大因子小于n/2），在n/2～n-1范围内没有数据可以整除此数。据此，我们可以把遍历范围缩小至1～n-1，这样程序效率可以提高一倍。相应的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 完数

if __name__=="__main__":
```

```
    n = int(input("请输入所选范围上限：  "))
    i = 2                                                    # 变量 i 控
制选定数的范围
    print(n)
    while i <= n:
        s = 0                                          # s记录累加因子之和，保证每次循
环时s的初值为0
        j = 1                                                # j 控制除数
范围
        while j <= (n // 2):
            if j != 0 and i % j == 0:        # 判断 j 是不是 i 的因子
                s += j                                       # 因子和
            j += 1

        if s == i:                                           # 判断因子之和是否和原
数相等
            print("2到%d之间的完数：%d" %(n, i))
        i += 1
```

3.5 亲密数

1. 问题描述

如果整数A的全部因子（包括1，不包括A本身）之和等于B，且整数B的全部因子（包括1，不包括B本身）之和等于A，则将整数A和B称为亲密数。求3000以内的全部亲密数。

2. 问题分析

按照亲密数定义，要判断整数a是否有亲密数，只要计算出a的全部因子的累加和，将其存到变量b，再计算b的全部因子的累加和设为n，若n等于a，则可判定a和b是亲密数。

3. 算法设计

计算数a的各因子的算法：用a依次对i（i的范围可以是1～a-1或1～a/2-1）进行模（“%”，在编程过程中一定要注意求模符号两边参加运算的数据必须为整数）运算，若结果等于0，则i为a的一个因子；否则i就不是a的因子。将所求得的因子累加到变量b。

接下来求变量b的因子：算法同上，将b的因子之和累加到变量n。根据亲密数的定义判断变量n是否等于变量a（if(n==a)），若相等，则a和b是一对亲密数，反之则不是。

4. 确定程序框架

程序的简单流程图如图3.10所示。

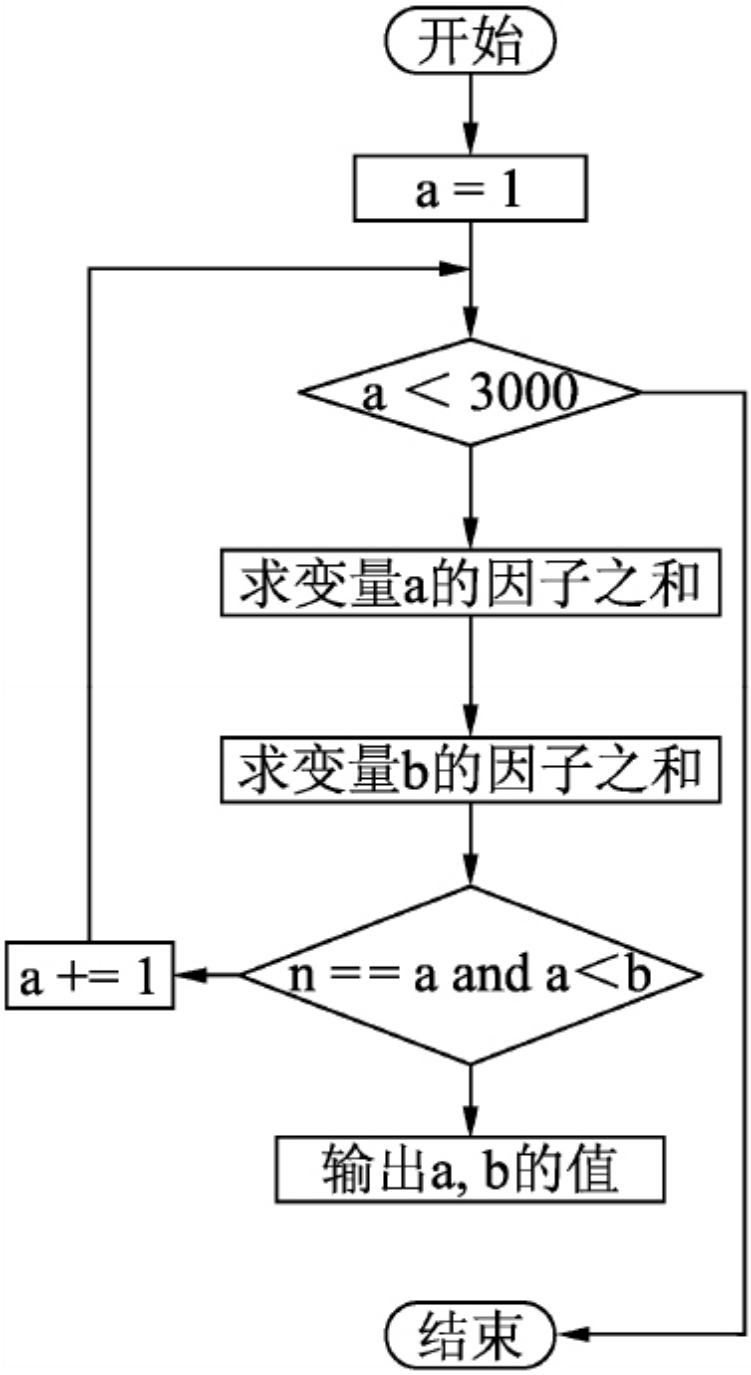


图3.10　程序流程图

流程图中求变量a和b的因子之和为简略流程图，只是说明这部分的作用，具体画法请参照3.4节的流程图。

5．完整的程序

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 亲密数

if __name__=="__main__":
    print("3000以内的全部亲密数为：")
    for a in range(1, 3000):                    # 穷举3000以内的全部整数
        b = 0
        i = 1
        while i <= (a//2):                  # 计算数 a 的各因子，将各因子之和存放到b中
            if a % i == 0:
                b += i
            i += 1

        n = 0                                                   # 计算b的各因子，将各
因子之和存于 n
        j = 1
        while j <= (b//2):
            if b % j == 0:
                n += j
            j += 1
        if n == a and a < b:
            print("%4d -- %4d \t" %(a, b))
```

6. 运行结果

在PyCharm下运行程序，结果如图3.11所示。

```
E:\code\python\Interest-python\venv\Scripts
3000以内的全部亲密数为：
 220 --  284
1184 -- 1210
2620 -- 2924

Process finished with exit code 0
```

图3.11　运行结果

7. 问题拓展

将原程序稍做改动，在最初定义的时候给变量b和n赋初值0。完整的代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 亲密数

if __name__=="__main__":
```

```
print("3000以内的全部亲密数为：")
b = 0
n = 0
for a in range(3000):              # 穷举30000以内的全部整数
    # 计算数a的各因子，将各因子之和存放到b中
    i = 1
    while i <= (a//2):
        if a % i == 0:
            b += i
        i += 1

    # 计算b的各因子，将各因子之和存于n
    j = 1
    while j <= (b//2):
        if b % j == 0:
            n += j
        j += 1
    if n == a and a < b:
        print("%4d -- %4d \t" %(a, b))
```

在PyCharm下运行程序，结果如图3.12所示。

图3.12　运行结果

从图3.5中可以看出程序并没有输出结果，即在3000这个范围内没有找到亲密数，而实际上亲密数是存在的，这是为什么呢？

后面这个程序看上去似乎没有什么问题，但是仔细分析一下会发现：在最初定义的时候给变量b和n赋初值0，第一次执行循环体时，将a和b的因子分别累加到b和n，得到的b和n的值确实是两个变量的因子之和，但是当再次执行循环体时，b和n的初值已不再是0，当再次把求得的因子累加到其上时，最后b和n存储的值并不是所求当前变量的因子之和（还包括上次判断的变量的因子之和），故最后没有符合条件的a和b。

注意：　对于这类需要多次将某些值存储到一个变量中的情况，一定要注意变量赋初值的位置。

3.6 自守数

1. 问题描述

自守数是指一个数的平方的尾数等于该数自身的自然数。例如，5^2 =25，25^2 =625，76^2 =5776，9376^2 =87 909 376。求100 000以内的自守数。

2. 问题分析

根据自守数的定义，求解本题的关键是知道当前所求自然数的位数，以及该数平方的尾数与被乘数、乘数之间的关系。

3. 算法设计

采用“求出一个数的平方后再截取最后相应位数”的方法显然是不可取的，因为计算机无法表示过大的整数。

分析手工方式下整数平方（乘法）的计算过程，以376为例：

```
   376          被乘数
×  376          乘数
----------
  2256          第一个部分积=被乘数×乘数的倒数第一位
  2632          第二个部分积=被乘数×乘数的倒数第二位
  1128          第三个部分积=被乘数×乘数的倒数第三位
----------
141376          积
```

本问题所关心的是积的最后三位。分析产生积的后三位的过程可以看出，在每一次的部分积中，并不是它的每一位都会对积的后三位产生影响。总结规律可以得到，在三位数乘法中，对积的后三位产生影响的部分积分别如下：

- 第一个部分积中：被乘数最后三位×乘数的倒数第一位。
- 第二个部分积中：被乘数最后两位×乘数的倒数第二位。
- 第三个部分积中：被乘数最后一位×乘数的倒数第三位。

将以上部分积的后三位求和后截取后三位就是三位数乘积的后三位。这样的规律可以推广到同样问题的不同位数乘积。

4. 求给定数的位数

求一个数的位数可以借助最高位的权值来计算，对于十进制来说，个位的权值为10^0，十位的权值为10^1，百位的权值为10^2，以此类推。一个存储三位数的变量n=123，每次除以10，将得到的值再赋给n，直到n的值为0，最多可以除3次；若变量n中存储的是4位数，用同样的方法去除以10最多可以除4次。可以发现，直到变量变为0，除以10的次数即为当前给定数的位数。程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 求给定数的位数

if __name__=="__main__":
    print("请输入一个正整数n：", end="")
    n = int(input())
    if n <=0:
        print("输出错误")
        exit()
    count = 0                                    # count存储数的位数
    while n != 0:
        n = n // 10
        count += 1
    print(" %d 位数" %count)
```

由于本题在下面的编程过程中还要用到原数number，故在求位数的程序段中为保证number值不被破坏，暂时将number的值赋给另一变量mul，即mul=number，由mul代替number去执行相应的操作。对应程序段如下：

```
mul = number
k = 1
while (mul // 10) > 0:                          # 由number的位数确定截取数字进行乘法时的系数k
```

```
    mul //= 10
    k *= 10
```

5. 分离给定数中的最后几位

从一个两位数（存在变量n中）开始分析，分离最低位个位：n%10；对于三位数n，分离最后两位：n%100；对于四位数n，分离最后三位：n%1000；以此类推，若分离出最后x位，只需要用原数对10x求余。

从算法设计部分所举例子可以看出，对于第二个部分积“2632”来说其实应该是“26320”，因为对于乘数中的倒数第二位“7”来说，因其在十位，对应的权值为10，第二个部分积实质上为376×70=26 320。故求部分积的程序段为：

```
while k > 0:
    # (部分积+截取被乘数的后N位×截取乘数的第M位)，%a再截取部分积
    mul = (mul + (number % (k * 10))*(number % b - number % (b //  10)))%a
    k //= 10                                    # k为截取被乘数时的系数
    b *= 10
```

对于整个循环来说，变量k是由number的位数确定截取数字进行乘法时的系数。第一次执行循环体时，被乘数的所有位数都影响到平方的尾数，故第一个部分的积等于被乘数×乘数的最后一位，将部分积累加到变量mul上，再对a取余截取相应的尾数位数；第二次执行循环体，影响平方尾数的是被乘数中除了最高位之外的数（故k先除以10再参加运算），第二个部分的积等于被乘数×乘数的倒数第二位，(number%b-number%(b//10))用来求乘数中影响平方尾数的对应位上的数；第三次、第四次执行循环体的过程同上。

6. 确定程序框架

该程序的简略流程图如图3.13所示。

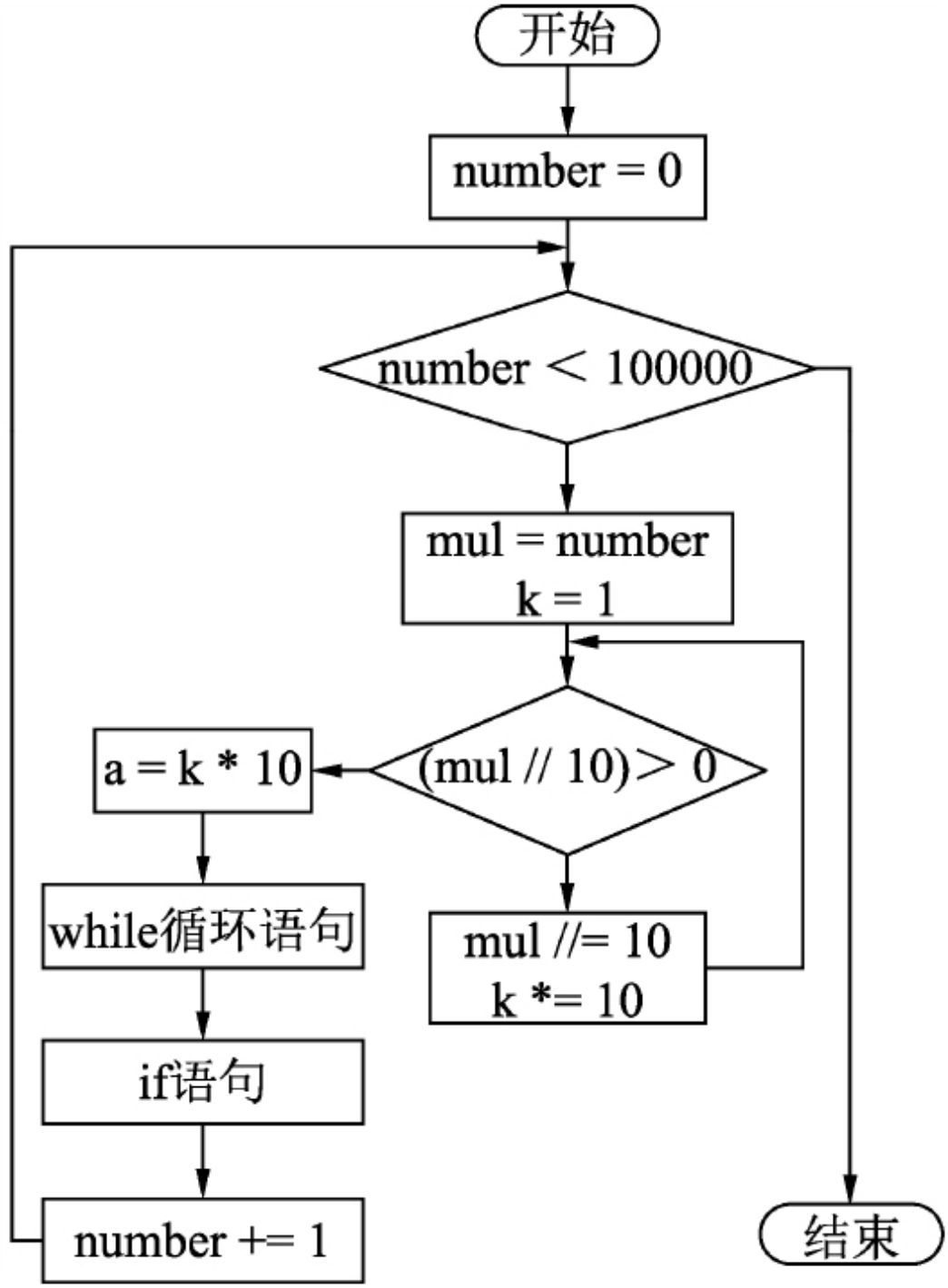


图3.13　程序流程图

7．完整的程序

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 自守数
```

```
if __name__=="__main__":
    print("100000以内的自守数：")
    for number in range(0, 100000):
        mul = number
        k = 1
        while (mul // 10) > 0:    # 由number的位数确定截取数字进行乘法时的系数k
            mul //= 10
            k *= 10

        a = k * 10                                          # a为截取部分积时的系数
        mul = 0                                             # 积的最后n位
        b = 10                                              # b为截取乘数相应位时的系数
        while k > 0:
            # (部分积+截取被乘数的后N位×截取乘数的第M位)，%a再截取部分积
            mul = (mul + (number % (k * 10))*(number % b - number % (b //
10)))%a
            k //= 10                            # k为截取被乘数时的系数
            b *= 10
        if number == mul:            # 判定若为自守数，则输出
            print("%ld " %number, end="\t")
    # print()
```

8. 运行结果

在PyCharm中运行程序，运行结果如图3.14所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe
100000以内的自守数：
0    1    5    6    25   76   376      625      9376     90625
Process finished with exit code 0
```

图3.14　运行结果

3.7　高次方数的尾数

1．问题描述

求13的13次方的最后三位数。

2．问题分析

许多初学者看到本题最容易想到的方法就是：将13累乘13次后截取最后三位即可。但是计算机中存储的整数是有一定范围的，超出某范围将不能正确表示，所以用这种算法不可能得到正确的结果。实际上，题目仅要求后三位的值，完全没有必要把13的13次方完全求出来。

3．算法设计

手工计算13的13次方的步骤如下：

```
  13          169          2197
 ×13         × 13         ×  13
 ----        ----         ----
  39          507          6591          ……
  13          169          2197
 ----        ----         ----
 169         2197         28561
```

研究乘法的规律可以发现，乘积的最后三位的值只与乘数和被乘数的后三位有关，与乘数和被乘数的高位无关。利用这一规律，在计算下一次的乘积时，我们只需用上次乘积的后三位来参与运算（即在求第三次乘积时，上次的乘积2197并不需要都参与运算，只取其后三位197再次与13相乘即可）。求某数的后三位的算法为：用某数对1000取模。

编程中，将累乘得到的积存储到变量last中，在进行下一次相乘之前先截取last的后三位再相乘，即last%1000*13，将结果存储到last中，即last=last*x%1000，这里x的值为13。因第一次相乘时

用到了变量last的初值，故在定义时给last赋初值1，或在参与计算之前给last赋初值1。

4. 确定程序框架

程序流程图如图3.15所示。

5. 完整的程序

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 高次方数的尾数

if __name__ == "__main__":
    last = 1                                                    # 变量last保存求得的x的y次方的
部分积的后三位
    print("请输入两个数x和y：")
    x = int(input("x = "))
    y = int(input("y = "))
    for i in range(1, y+1):
        last = last * x % 1000       # 将last乘x后对1000取模，即求积的后三位
    print("%d的%d次方所得积的后三位为：%d" %(x,y,last))
```

6. 运行结果

在PyCharm下运行程序，分别输入x和y的值，这里输入13、13，运行结果如图3.16所示。

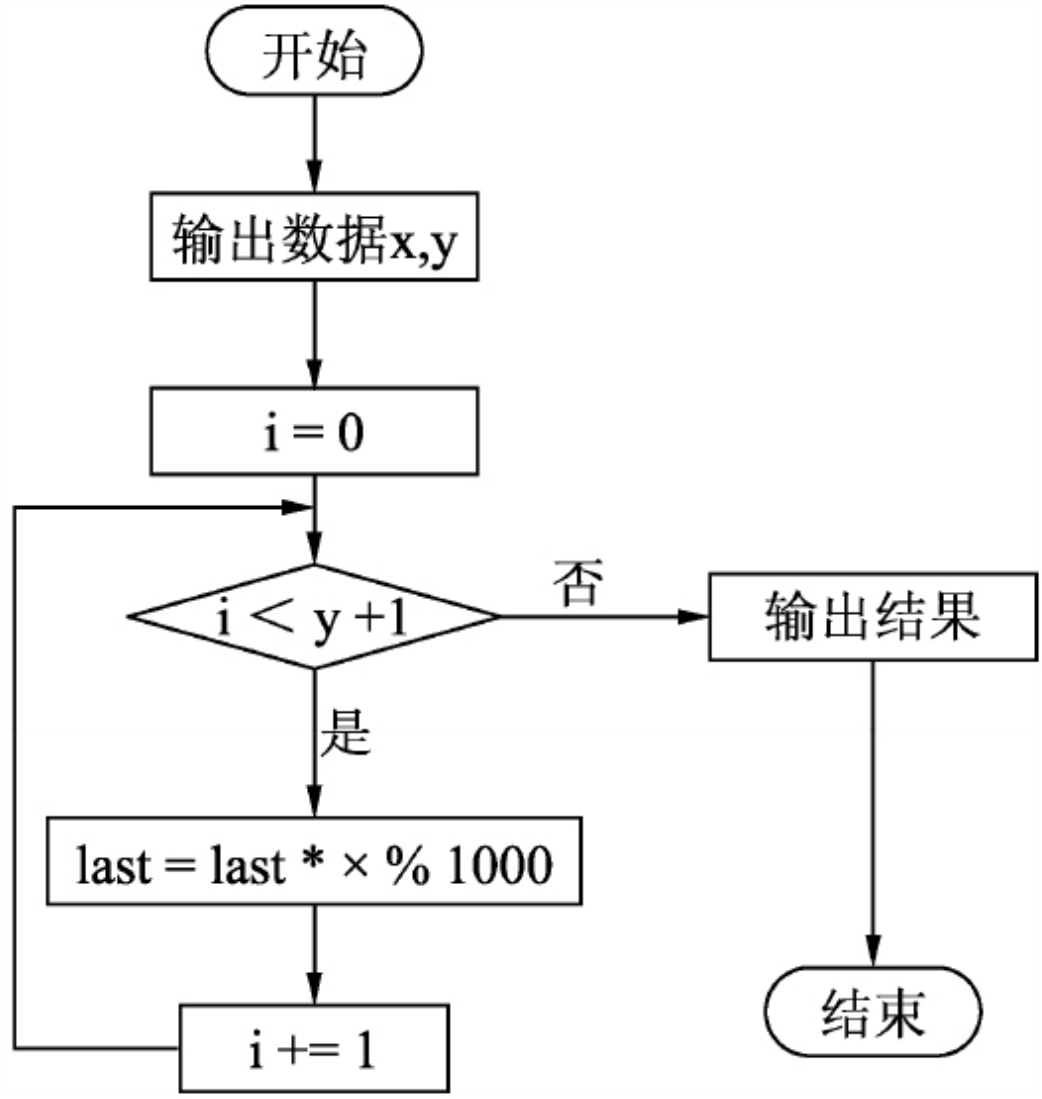


图3.15　程序流程图

```
E:\code\python\Interest-python\venv\
请输入两个数x和y：
x = 13
y = 13
13的13次方所得积的后三位为：253

Process finished with exit code 0
```

图3.16　运行结果

上述程序段具有普遍性，无论是求哪个数的多少次方的后三位尾数都可以利用上述程序实现。若针对本题中的具体数据，程序可改为：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 高次方数的尾数

if __name__ == "__main__":
    last = 1                                    # 变量last保存求得的13的13次方的部分积的后三位
    for i in range(1, 13+1):          # 13自乘的次数
        last = last * 13 % 1000       # 将last乘13后对1000取模，即求积的后三位
    print("13的13次方所得积的后三位为：%d" %last)
```

将会直接打印出结果：253。

3.8　黑洞数

1．问题描述

编程求三位数中的“黑洞数”。

黑洞数又称陷阱数，是指任何一个数字不全相同的整数，在经过有限次“重排求差”操作后，总会得到某一个或一些数，这些数即为黑洞数。“重排求差”操作是将组成一个数的各位数字重排，将得到的最大数减去最小数。例如，207的“重排求差”操作序列是：720-027=693，963-369=594，954-459=495，此时再进行“重排求差”操作不会发生改变。再用208计算一次，也是停止到495，所以495是三位黑洞数。

2．问题分析

根据“黑洞数”定义，对于任意一个数字不全相同的整数，最后结果总会掉入到一个黑洞圈或黑洞数里，最后结果一旦为黑洞数，无论再重复进行多少次的“重排求差”操作，结果都是一样的，因此可把结果相等作为判断“黑洞数”的依据。

3．算法设计

过程如下：

1）将任意一个三位数进行拆分。

2）拆分后的数据重新组合，将可以组合的最大值减去最小值，差值赋给变量j。

3）将当前差值暂存到另一变量h中：h=j。

4）对变量j执行拆分、重组、求差操作，差值仍然存储到变量j中。

5）判断当前差值j是否与前一次的差值h相等，若相等，则将差值输出并结束循环，否则重复步骤3～5。

4．比较三个数的大小并将其重组

求“黑洞数”的关键是求出拆分后所能组成的最大值max和最小值min，求最大值和最小值的关键是找出拆分后数值的大小关系，通过比较找出最大值、次大值及最小值。三个数比较大小可以采用两两比较的方法，首先a与b比较，其次a与c比较，最后b与c比较。比较顺序很重要，有时比较顺序不一样得到的结果也是不一样的。在比较过程中如需对两个数进行交换，则需借助中间变量t来实现，否则变量中存储的数将被改变。如“a=b;b=a;”，第一个语句执行完毕后变量a的值由原值变为b的值，第二个语句的作用是把现在a的值赋给b，但此时a中存储的已经不再是原来的值，而是被赋予的b值。

比较后数值按照从大到小的顺序分别存储在变量a、b、c中，再按一定的顺序重新组合成最大值和最小值。因求最大值和最小值的操作在程序中不止一次用到，故可定义两个函数three_max(a,b,c)和three_min(a,b,c)，功能分别是求由三个数组成的最大值和最小值。代码如下：

```
# 求三位数的组合最大值
def three_max(a, b, c):          # a、b、c分别对应百位、十位、个位
    if a < b:                                # 如果a<b，则将变量a、b的值互换
        t = a
        a = b
        b = t
    if a < c:
        t = a
        a = c
        c = t
    if b < c:
        t = b
        b = c
        c = t
    return a*100 + b*10 + c
```

函数three_min(a,b,c)的代码与以上代码的不同之处在于最后的返回值，函数three_min(a,b,c)中需返回的是最小值c*100+b*10+a。

5．寻找“黑洞数”

将第一次得到的差值j赋给变量h，因在后面的编程过程中会再次将得到的差值赋给变量j，为避免j中存储的原值找不到，故先把前一次的差值暂存到另一个变量h中。在比较过程中一旦两次结果相等，循环过程即可结束，可用break语句实现。判定条件j==h可以在循环体中用if语句实现，也可写在while语句中。寻找“黑洞数”的方法代码如下：

```
# 求黑洞数
def black_number(max, min):
    j = max - min
    k = 0
    while k < min:                                          # k控制循环次数
        h = j                                               # h记录上一
次最大值与最小值的差
        hun = j // 100                                      # 百位
        ten = j % 100 // 10                                 # 十位
        bit = j % 10                                        # 个位
        max = three_max(hun, ten, bit)      # 最大值
        min = three_min(hun, ten, bit)      # 最小值
        j = max - min
        if j == h:                                          # 最后两次差
相等时，差即为所求黑洞数
            print("%d " % j)
            break                                           # 跳出循环
        k += 1
```

6. 完整的程序

完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 黑洞数

# 求三位数组合的最大值
def three_max(a, b, c):                                 # a、b、c分别对应百位、十
位、个位
    if a < b:                                           # 如果a<b，则将
变量a、b的值互换
        t = a
        a = b
        b = t
    if a < c:
        t = a
        a = c
        c = t
    if b < c:
        t = b
        b = c
        c = t
    return a*100 + b*10 + c

#求三位数组合的最小值
def three_min(a, b, c):                                 # a、b、c分别对应百位、十
位、个位
    if a < b:                                           # 如果a<b，则将
```

```
变量a、b的值互换
        t = a
        a = b
        b = t
    if a < c:
        t = a
        a = c
        c = t
    if b < c:
        t = b
        b = c
        c = t
    return c*100 + b*10 + a


# 求黑洞数
def black_number(max, min):
    j = max - min
    k = 0
    while k < min:                                                    # k控制循环次数
        h = j                                                           # h记录上一
次最大值与最小值的差
        hun = j // 100                                                  # 百位
        ten = j % 100 // 10                                     # 十位
        bit = j % 10                                                    # 个位
        max = three_max(hun, ten, bit)          # 最大值
        min = three_min(hun, ten, bit)          # 最小值
        j = max - min
        if j == h:                                                              # 最后两次差
相等时，差即为所求黑洞数
            print("%d " % j)
            break                                                       # 跳出循环
        k += 1


if __name__ == "__main__":
    i = int(input("请输入一个三位整数："))
    hun = i // 100                                                      # 百位
    ten = i % 100 // 10                                     # 十位
    bit = i % 10                                                        # 个位
    max = three_max(hun, ten, bit)          # 最大值
    min = three_min(hun, ten, bit)          # 最小值
    print("max = ", max)
    print("min = ", min)
    black_number(max, min)
```

7. 运行结果

在PyCharm下运行程序，屏幕上提示“请输入一个三位数：”，分别输入306和108，运行结果如图3.17所示。

```
E:\code\python\Interest-python\venv
请输入一个三位整数：306
max =  630
min =  36
495

Process finished with exit code 0
```

a）运行结果 1

```
E:\code\python\Interest-python\venv
请输入一个三位整数：108
max =  810
min =  18
495

Process finished with exit code 0
```

b）运行结果 2

图3.17　运行结果

3.9　勾股数

1．问题描述

求100以内的所有勾股数。

所谓勾股数，是指能够构成直角三角形三条边的三个正整数（a，b，c）。

2．问题分析

根据“勾股数”定义，所求三角形三边应满足条件$a^2+b^2=c^2$。可以在所求范围内利用穷举法找出满足条件的数。

3．算法分析

采用穷举法求解时最容易想到的一种方法是利用三个循环语句分别控制变量a、b、c的取值范围，第一层控制变量a，取值范围是1～100；在a值确定的情况下再确定b值，即第二层控制变量b，为了避免结果有重复现象，b的取值范围是(a+1)～100；a、b的值已确定，利用穷举法在(b+1)～100范围内一个一个地去比较，看当前c值是否满足条件$a^2+b^2=c^2$，若满足，则输出当前a、b、c的值，否则继续寻找。主要代码如下：

```
…
for a in range(1,101):                                  # 确定a的取值
    for b in range(a+1, 101):                           # 确定b的取值
        for c in range(b+1, 101):                       # 确定c的取值
            if a*a + b*b == c*c:                        # 判断三个变量是否满足勾股数条件
                print("%4d %4d %4d\t |" %(a, b, c), end="")
…
```

但是上述算法的效率比较低，根据$a^2+b^2=c^2$这一条件，在a、b值确定的情况下没必要再利用循环一个一个去寻找c值。若a、b、c是一组勾股数，则a^2+b^2的平方根一定等于c，c的平方应该等于a、b的平方和，故可将a^2+b^2的平方根赋给c，再判断c的平方是否

等于a^2 +b^2 。根据“勾股数”定义将变量定义为整型，a^2 +b^2 的平方根不一定为整数，但变量c的类型为整型，将一个实数赋给一个整型变量时可将实数强制转换为整型（舍弃小数点之后的部分），然后再赋值，这种情况下得到的c的平方与原来的a^2 +b^2 的值肯定不相等。故可利用这一条件进行判断。

4. 完整的程序

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @Time : 2019/6/12 0:37
# @desc: 勾股数  a**2+b**2=c**2

import math

if __name__ == "__main__":
    count = 0
    print("100以内的勾股数有：")
    # a、b、c分别表示三角形的三条边
    # 满足勾股数的三个数组成的三角形一定是直角三角形
    print("   a\tb\t c \t\t a \t b \t  c \t\t a \t  b\t  c \t\t a\t b\t  c");
    # 求100以内的勾股数
    for a in range(1,101):
        for b in range(a+1, 101):            # 邻边不能相等，否则就是等边三角形了
            c = int(math.sqrt(a*a + b*b))    # 求c值，并转换为整型
            # 判断c的平方是否等于a*a + b*b，且两边之和大于第三边
            if c*c == (a*a + b*b) and (a + b > c) and (a + c > b) and (b + c >
a) and c <= 100:
                print("%4d %4d %4d\t |" %(a, b, c), end="")
                count += 1
                if count % 4 == 0:
                    print()
```

5. 运行结果

程序运行结果如图3.18所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/code/python/Interest-
100以内的勾股数有：
  a   b   c  |  a   b   c  |  a   b   c  |  a   b   c  |
  3   4   5  |  5  12  13  |  6   8  10  |  7  24  25  |
  8  15  17  |  9  12  15  |  9  40  41  |  10 24  26  |
 11  60  61  |  12 16  20  |  12 35  37  |  13 84  85  |
 14  48  50  |  15 20  25  |  15 36  39  |  16 30  34  |
 16  63  65  |  18 24  30  |  18 80  82  |  20 21  29  |
 20  48  52  |  21 28  35  |  21 72  75  |  24 32  40  |
 24  45  51  |  24 70  74  |  25 60  65  |  27 36  45  |
 28  45  53  |  28 96  100 |  30 40  50  |  30 72  78  |
 32  60  68  |  33 44  55  |  33 56  65  |  35 84  91  |
 36  48  60  |  36 77  85  |  39 52  65  |  39 80  89  |
 40  42  58  |  40 75  85  |  42 56  70  |  45 60  75  |
 48  55  73  |  48 64  80  |  51 68  85  |  54 72  90  |
 57  76  95  |  60 63  87  |  60 80  100 |  65 72  97  |

Process finished with exit code 0
```

图3.18　运行结果

6．问题拓展

对于“勾股数”除了利用上面的算法外还有其他方法。

由于任何一个勾股数组(a,b,c)内的三个数同时乘以一个整数n得到的新数组(na,nb,nc)仍然是勾股数，所以一般我们想找的是a、b、c互质的勾股数组。

关于这样的数组，比较常用也比较实用的方法有以下两种。

（1）第一种方法

当a为大于1的奇数2n+1时，$b=2n^2+2n$，$c=2n^2+2n+1$。

实际上就是把a的平方数拆成两个连续自然数，例如：

- n=1时，(a,b,c)=(3,4,5)
- n=2时，(a,b,c)=(5,12,13)
- n=3时，(a,b,c)=(7,24,25)

……

这是最经典的一个方法，而且由于两个连续自然数必然互质，所以用这个方法得到的勾股数组全部都是互质的。

（2）第二种方法

当a为大于4的偶数2n时，$b=n^2-1$，$c=n^2+1$。

也就是把a的一半的平方分别减1和加1，例如：

- n=3时，(a,b,c)=(6,8,10)
- n=4时，(a,b,c)=(8,15,17)
- n=5时，(a,b,c)=(10,24,26)
- n=6时，(a,b,c)=(12,35,37)

……

这是次经典的方法，当n为奇数时由于(a,b,c)是三个偶数，所以该勾股数组必然不是互质的；而n为偶数时由于b、c是两个连续奇数必然互质，所以该勾股数组互质。

所以如果只想得到互质的数组，该方法可以改成对于a=4n（n≥2），$b=4n^2-1$，$c=4n^2+1$，例如：

- n=2时(a,b,c)=(8,15,17)
- n=3时(a,b,c)=(12,35,37)
- n=4时(a,b,c)=(16,63,65)

……

代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
```

```
# @desc: 互质勾股数

if __name__ == "__main__":
    a = int(input("请输入一个a值："))
    # 输入的a为奇数，则a = 2n+1，b=2n**2+2n，c=2n**2+2n+1
    if a >= 3 and a % 2 == 1:
        n = (a - 1) // 2
        b = 2 * n**2 + 2*n
        c = 2 * n**2 + 2*n + 1
        print("%d %d %d\n" %(a,b,c))
    else:
        # 输入的a为偶数，则a = 2n，b=n**2-1，c=n**2+1
        if a >= 3 and a % 2 == 0:
            n = a // 2
            b = n**2 - 1
            c = n**2 + 1
            print("%d %d %d\n" % (a, b, c))
        else:
            print("error")
```

3.10　不重复的3位数

1．问题描述

用1、2、3、4共4个数字能组成多少个互不相同且无重复数字的三位数？都是多少？

2．问题分析

求互不相同的三位数，可以一位一位地去确定，先确定百位，再确定十位和个位，各位上的数值进行比较，若互不相同则输出。

3．算法设计

（1）利用多重循环嵌套的for语句实现。

（2）用三重循环分别控制百位、十位、个位上的数字，它们都可以是1、2、3、4。

（3）在已组成的排列数中，还要再去掉出现重复的1、2、3、4这些数字不满足条件的排列。

题目要求最后输出满足条件的数据个数，需要一个变量count充当计数器的作用，有一个满足条件的数据出现计数器的值加1。为了使每行能输出8个数字，每输出一个数字就对count的值进行判断，看是否能被8整除，若能整除则输出换行符。

```
if count % 8 == 0:
    print()
```

4．确定程序框架

该程序的流程图如图3.19所示。

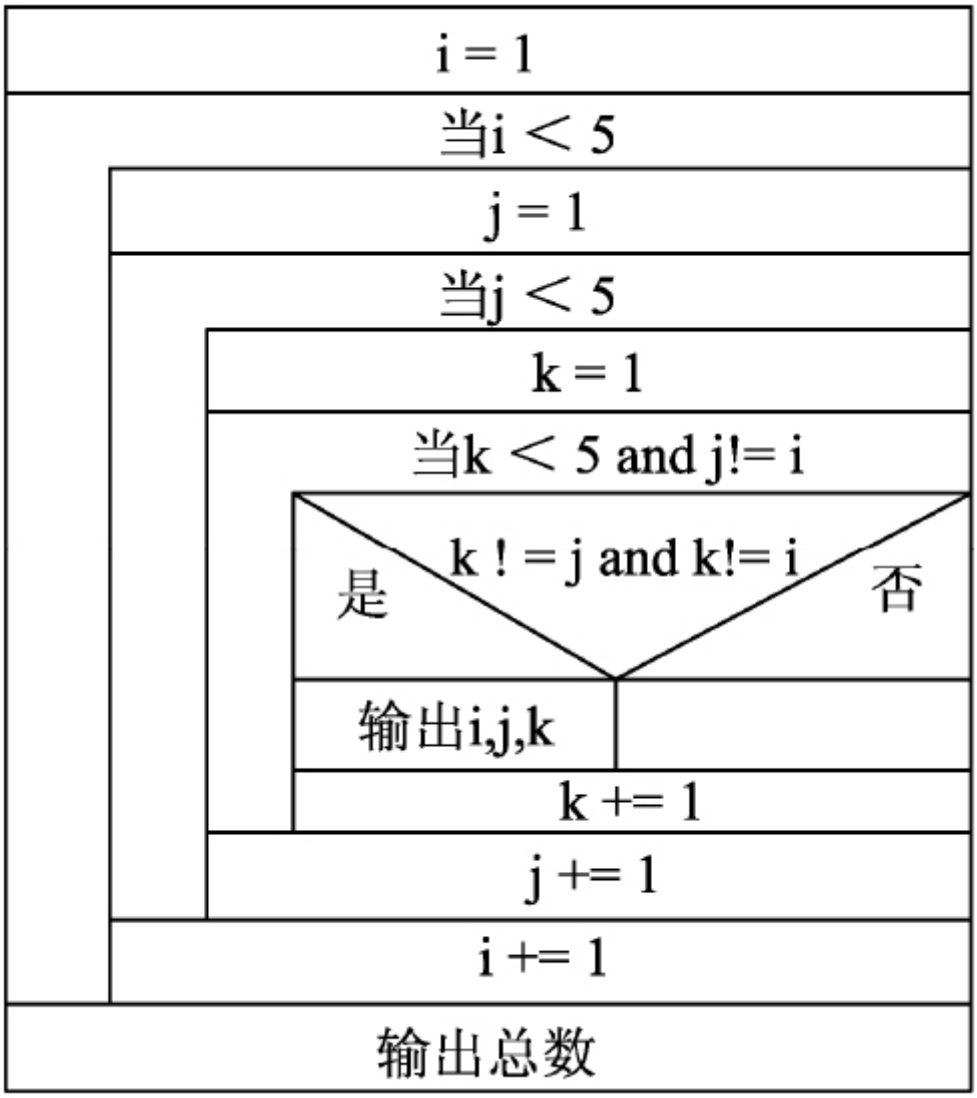


图3.19　程序流程图

5. 完整的程序

完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 不重复的3位数

if __name__ == "__main__":
    count = 0                                           # 计数器
    for i in range(1, 5):
        for j in range(1, 5):
            for k in range(1, 5):
                if i != k and i != j and j != k:     # 判断三个数是否互不相同
                    count += 1
                    print("%d%d%d  " %(i,j,k), end=" ")
```

```
                    if count % 8 == 0:
                        print()
    print("三位互不相同的数，共有：%d" %count, "个")
```

6. 运行结果

在PyCharm下运行程序，结果如图3.20所示。

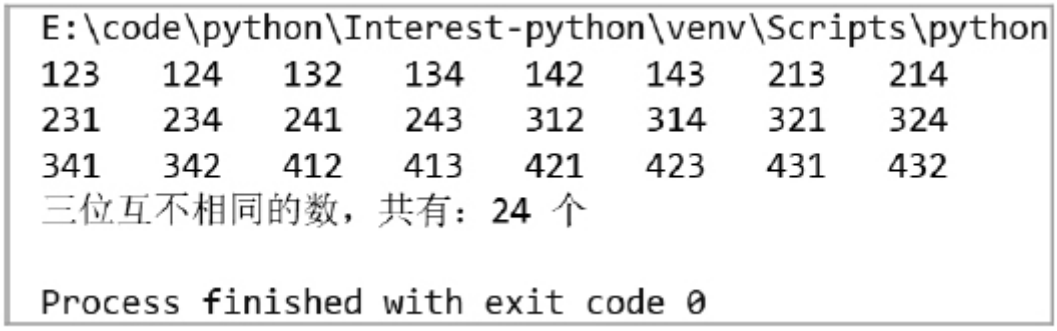


图3.20　运行结果

7. 问题拓展

上面的程序段的效率比较低，因为无论i与j的值是否相等，k都要从1到4把所有的值遍历完。根据题目要求，只要i与j的值相等，那么k的取值就没必要进行，因为无论k的值是多少，最后组成的三位数中总有相同的数字。对于本题来说，因取值范围较小，算法效率的高低相差并不大，但是对于取值范围大的题目，两种算法的效率相差是很明显的。程序代码可改写如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 不重复的3位数

if __name__=="__main__":
    count = 0                                                                    #
计数器
    for i in range(1, 5):
        for j in range(1, 5):
            k = 1
            while k < 5 and j != i:
                if k != j and k != i:
                    print("%d%d%d  " %(i,j,k), end=" ")
                    count += 1
                    if count % 8 == 0:                  # 每输出8个换一行
                        print()
                k += 1
    print("三位互不相同的数，共有：%d" % count, "个")
```

第4章　趣味分数

分数是数学中的一个概念，它一般包括真分数、假分数和带分数。分数在我们的日常生活学习中经常会用到，如商场的优惠活动幅度、降水概率等都要用分数来衡量，数学中比较两个或多个分数的大小，分数之间的运算等都是经常会遇到的问题。本章针对生活中尤其是数学中的分数问题从程序设计的角度出发去解决实际问题。

4.1　将真分数分解为埃及分数

1．问题描述

现输入一个真分数，请将该分数分解为埃及分数。

2．问题分析

真分数（a proper fraction）是指分子比分母小的分数，其分数值小于1。如1/2、3/5、8/9等都是真分数。

分子是1的分数，叫单位分数。古代埃及人在进行分数运算时，只使用分子是1的分数，因此这种分数也叫作埃及分数，或者叫单分子分数。例如，8/11=1/2+1/5+1/55+1/110。

我们约定分子分母都是自然数，分数的分子用a表示，分母用b表示。若真分数的分子a能整除分母b，则真分数经过化简就可以得到埃及分数，若真分数的分子不能整除分母，则可以从原来的分数中分解出一个分母为(b/a)+1的埃及分数。用这种方法将剩余部分反复分解，最后可得到结果。

3．算法设计

真分数分解为埃及分数的思路可归纳如下：

1）分数的分子用a表示、分母用b表示，变量c用来存储各个埃及分数的分母。

2）如果分母是分子的倍数，直接约简成埃及分数。此时，埃及分数的分母c=b/a，分子为1，即直接将变量a赋值为1。

3）若分母不是分子的倍数，则可以分解出一个分母为(b/a)+1的埃及分数，即变量c的值为(b/a)+1。

4）如果分子是1，表明已经是埃及分数，不用再分解，分解过程结束。

如果分数的分子a为1，则说明此时的分数已经是埃及分数，无须再分解，可结束循环。对于这种不受循环条件限制，当某一条件满足时便可结束循环的情况，可用break语句实现。

```
if a == 1:
    print("1/%ld\n" %c, end="")
    break
```

5）如果分子是3而且分母是偶数，直接分解成两个埃及分数1/(b/2)和1/b。

因分母为偶数，故变量b一定是2的倍数，对于分解出来的分数1/(b/2)经过约分之后肯定能得到一个埃及分数。原分数分解为两个埃及分数之后便可利用break语句结束循环。

```
if a == 3 and b % 2 == 0:       # 若余数分子为3，分母为偶数，输出最后两个埃及分数
    print("1/%ld + 1/%ld\n" %(b//2, b))
    break
```

6）从分数中减去这个分母为(b/a)+1的埃及分数，回到步骤2重复上述过程。

分解出此埃及分数之后用原分数a/b减去此埃及分数，得到新的分数。此新分数的分子a=a*c-b，分母b=b*c。

整个程序没有明确的循环条件，故为了能使循环继续，需将循环条件用一非0的常量表示条件为真。从上述过程可以看出，虽然利用循环条件不能结束循环，但当满足某一条件时利用break语句，仍然可以避免程序进入死循环。

将某一真分数分解为一个以上的埃及分数后，输出时要求以各分数相加的形式输出，故在输出语句中“+”是作为普通字符输出的。

```
print("1/%ld + " %c, end="")
```

4. 补充知识点

（1）print()函数

print()函数的一般调用形式为：

```
print(格式控制　%输出表列)
```

也可以省略格式控制，直接输出。

格式说明符用于为输出项提供输出格式说明，也就是使数据按格式说明符的要求进行输出，其由%号和紧跟在其后的格式描述符组成。

部分常用的格式说明符及其描述如表4.1所示。

表4.1　数据类型及其对应的格式说明符

格式说明符	描　　述
%d	整型
%f或%e	浮点型
%c	字符及其ASCII码

（续）

格式说明符	描　　述
%s	字符串
%ld	长整型
%lf	双浮点型

在双引号中除了格式说明符之外的内容要全部原样输出；输出表列中各个输出项之间要用逗号隔开；输出项可以是任意合法的常量、变量或表达式。

注意事项：

• 输出比较自由一些，输出的各个数之间到底是什么，取决于格式说明符之间的内容。

• 格式说明符要与输出项一一对应。

• 输出语句中还可以有\n、\r、\t、\a等转义字符。

• 尽量不要在输出语句中改变输出变量的值。

• 输出的数据中如果存在变量，一定是已经定义过的。

（2）不换行输出

在使用print()函数时，它会自动打印一个换行符。如果不想换行或不想自动添加换行符，可以在print()函数中加上参数end=""来避免换行，其中end参数的双引号中可以添加任意字符，实现在打印的结果后面添加任意字符。

实例代码：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: print()函数

if __name__ == "__main__":

    pi = 3.141592657
    # 输出整数部分
    print("%d" %pi)
    # 指定占位符宽度，输出整数部分
    print("%5d" % pi)
    # 指定占位符宽度，左边补0，输出整数部分
    print("%05d" %pi)
    # 指定占位符宽度，右对齐，输出整数部分
    print("%-5d" %pi)

    # 保留2位小数
    print("%.2f" % pi)
    # 指定占位符，并保留3位小数
    print("%6.3f" %pi)
    # 指定占位符，并保留4位小数，左边补0
    print("%06.4f" %pi)
    # 指定占位符，保留2位小数，左边补0，始终带符号
    print("%+05.2f" %pi)

    # 使用科学记数法表示
    print("%e" %pi)

    # 输出字符串
    str = "HelloWorld!"
    print("str = %s" %str)
```

```
    # 也可以省略格式说明，直接输出
    print("str = " , str)
    # 保留3个字符
    print("%.3s" %str)

    # 输出字符
    str1 = 'a'
    print("str1 = %c" %str1)

    print("-------------")
    for i in range(3):
        print(i)                                        # 换行输出

    for i in range(3):
        print(i, end=" ")                               # 不换行输出
```

在PyCharm下运行程序，结果如图4.1所示。

5. 确定程序框架

程序流程图如图4.2所示。

6. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 将真分数分解为埃及分数

if __name__ == "__main__":
    print("请输入一个分数: ", end=" ")
    a, b = [int(i) for i in input().split()]
    print("输入的分数为: %ld/%ld" %(a, b))
    print("埃及分数: ", end=" ")
    while 1:
        if b % a != 0:       # 若分子不能整除分母，则分解出一个分母为b//a+1的埃及数
            c = b // a + 1
        else:
            c = b // a
            a = 1
        if a == 1:
            print("1/%ld\n" %c, end="")
            break
        else:
            print("1/%ld + " %c, end="")
        a = a * c - b                                   # 求出余数的分子
        b = b * c                                       # 求出余数的分母
        # 若余数分子为3，分母为偶数，则输出最后两个埃及数
        if a == 3 and b % 2 == 0:
            print("1/%ld + 1/%ld\n" %(b//2, b))
            break
```

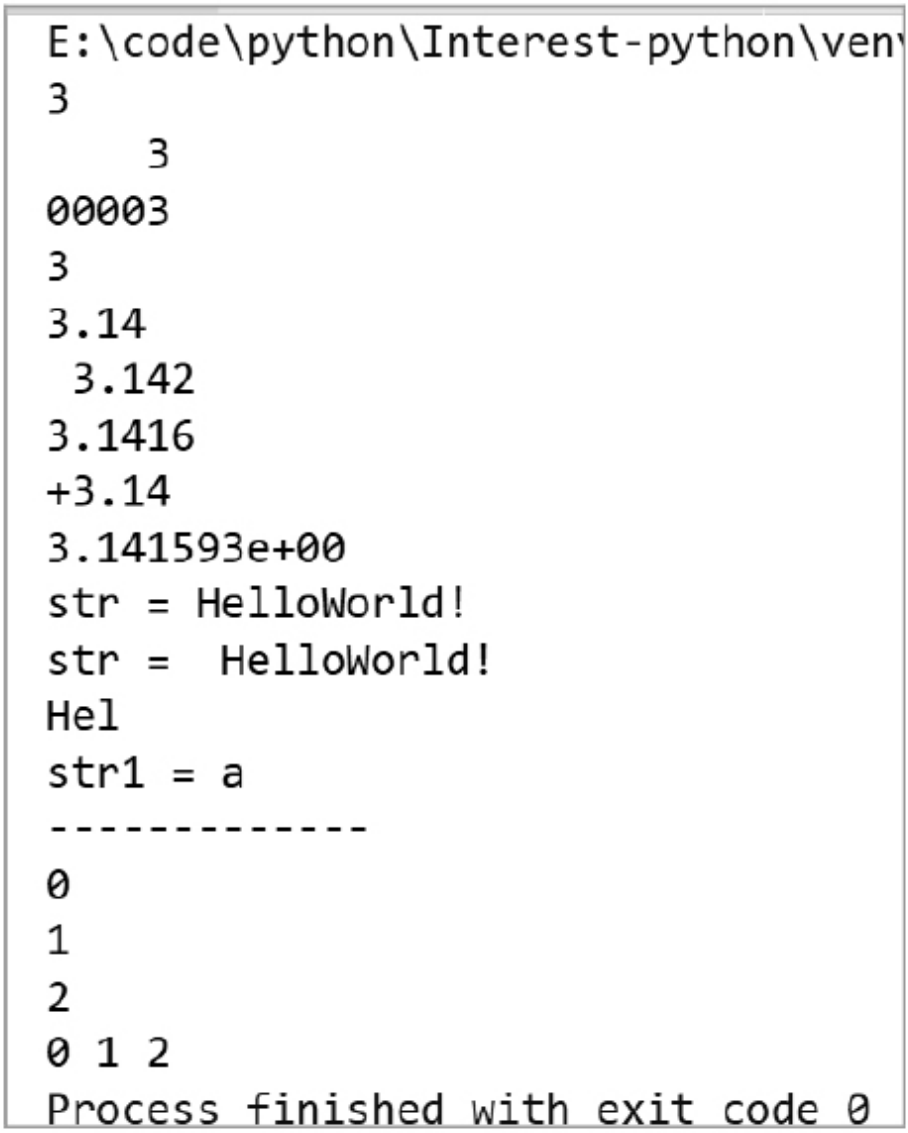

E:\code\python\Interest-python\ven
3
3
00003
3
3.14
3.142
3.1416
+3.14
3.141593e+00
str = HelloWorld!
str = HelloWorld!
Hel
str1 = a
0
1
2
0 1 2
Process finished with exit code 0

图4.1　运行结果

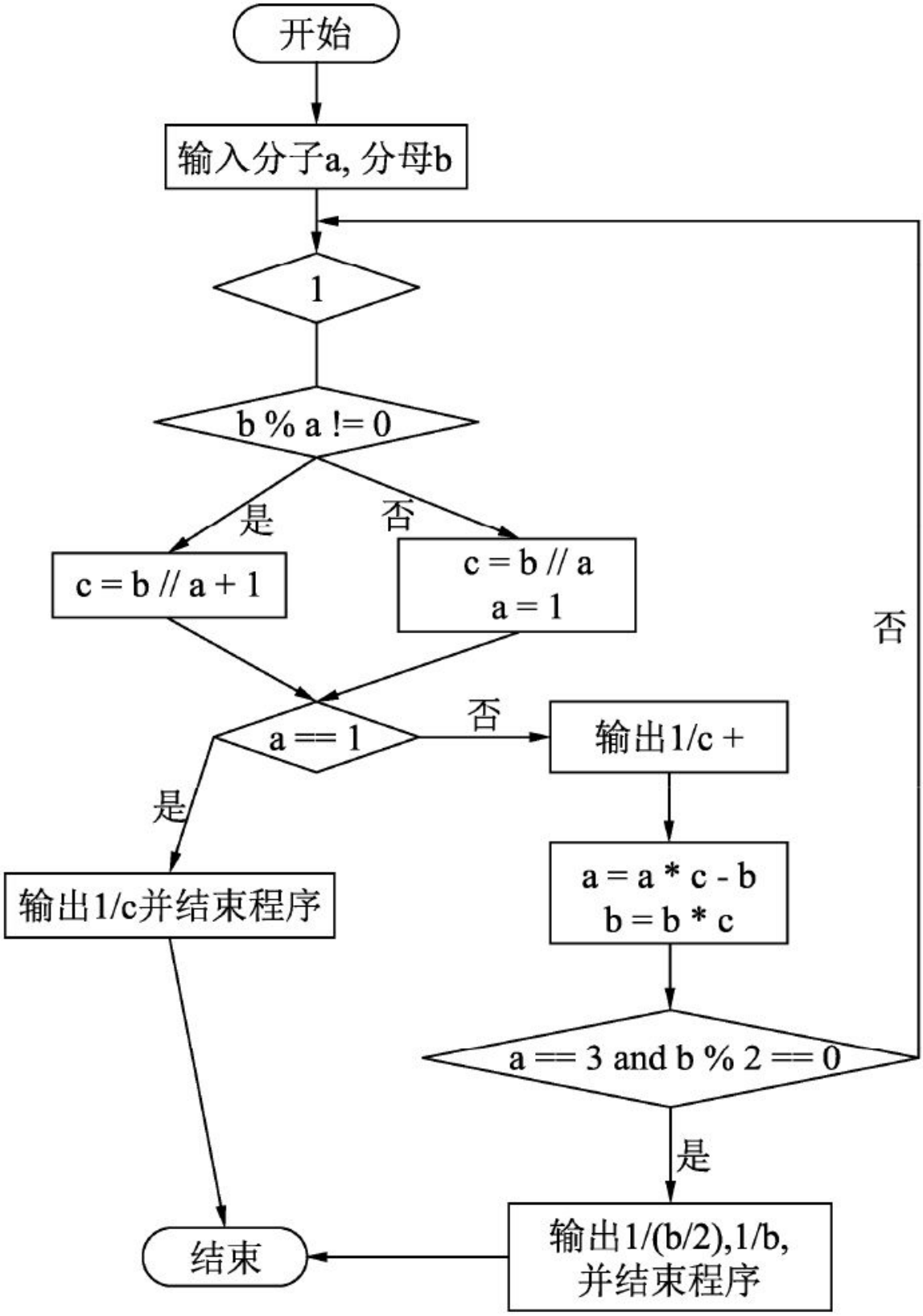
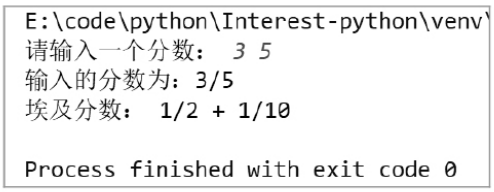


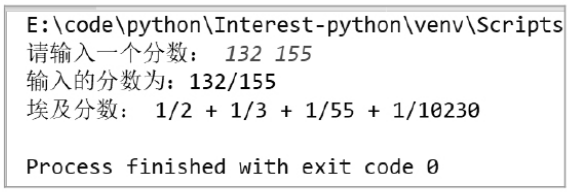
图4.2　程序流程图

7．运行结果

在PyCharm下运行程序，根据屏幕提示按照“数1/数2”的格式输入分子a和分母b的值。先输入3/5，再输入132/155，两次不同输入的运行结果如图4.3所示。读者还可以输入其他的分数来观察程序的运行结果。



a）运行结果 1



b）运行结果 2

图4.3　运行结果

4.2　列出真分数序列

1．问题描述

按递增顺序依次列出所有分母为40、分子小于40的最简分数。

2．问题分析

分子和分母只有公因数1的分数，或者说分子和分母是互质数的分数，叫作最简分数，又称既约分数。如2/3、8/9、3/8等就是最简分数。

方法1：求分子小于40的最简分数，可以对分子采用穷举的方法。根据最简分数的定义可知，分子和分母的最大公约数为1，因此可以利用最大公约数的方法，判定分子与40是否构成真分数。

方法2：分子和分母的公因数只有1的分数为最简分数，若分子和分母在1～分子（num2）（题目要求分子小于40，分子和分母的公约数小于两者中的任意一个）之间除了1之外还有其他的公因数，则此分数肯定不是最简分数。

3．算法分析

变量num1、num2分别用于存储分母和分子的值。

方法1：

求最大公约数一般采用辗转相除的思想，具体步骤概括如下：

1）用较大的数num1除以较小的数num2，得到的余数存储到变量temp中，即temp=num1%num2。

2）用较小的除数num2和得出的余数temp构成新的一对数，并分别赋值给num1和num2，继续做上面的除法。

3）当num2为0时，num1就是最大公约数；否则重复步骤1和步骤2。

对于辗转相除法的思想将在4.4节详细说明，此处先跳过。

方法2：

分数的分子仍然采用穷举法。对于每一个可能的分子，都要判断在1～num2范围内分数num1/num2除了1之外是否有其他的公因数，循环初值为2。

在2～num2内若有一个数j能同时整除分子、分母，说明此分数不是最简分数，j～num2之间的数也无须再判断，利用break语句结束循环，循环结束时j<num2。循环过程中若没有一个数可以同时整除分子和分母，即条件if(num1%j==0)and(num2%j==0)不成立，则break语句不执行，循环正常结束，即条件j≤num2不成立，循环结束时j>num2。利用j与num2的大小关系可判断分数是否为最简分数。

4. 确定程序框架

辗转相除法求最大公约数的流程图见4.4节，下面给出方法2的流程图，如图4.4所示。

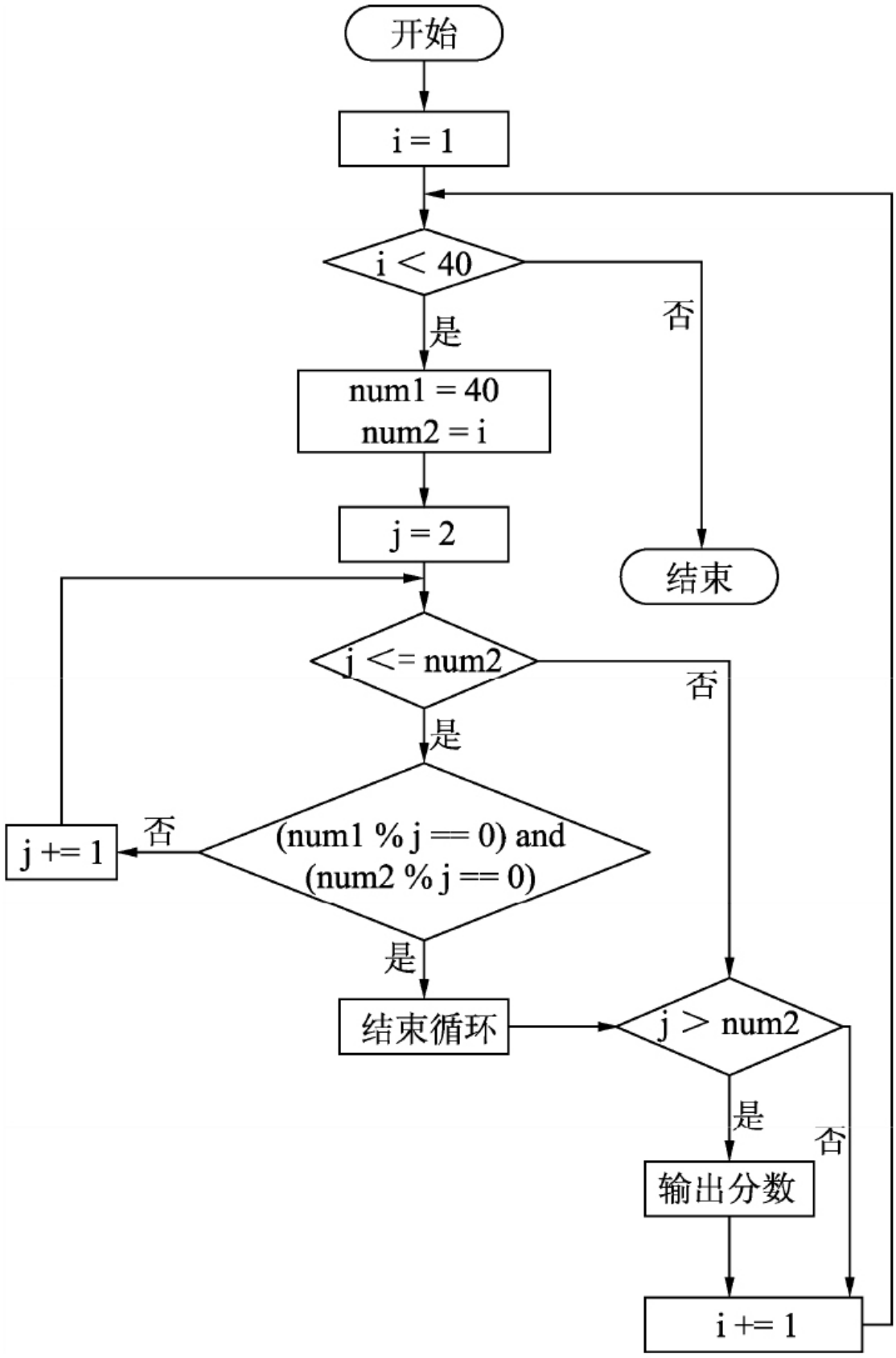

开始
i = 1
i < 40
否
是
num1 = 40
num2 = i
j = 2
结束
j <= num2
否
是
(num1 % j == 0) and
(num2 % j == 0)
否
j += 1
是
结束循环
j > num2
是
否
输出分数
i += 1

图4.4　程序流程图

5. 完整的程序

代码1（对应方法1）：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 列出真分数序列——利用辗转相除法

if __name__ == "__main__":
    print("分母为40，分子小于40的最简分数有：")
    n = 0                                                    # 计数器，记录最简分数的个数
    for i in range(1, 40):                                   # 穷举40以内的全部分子
        num1 = 40                                            # 分母
        num2 = i                                             # 分子
        # 采用辗转相除法求出分子与分母的最大公约数
        while num2 != 0:
            temp = num1 % num2
            num1 = num2
            num2 = temp
        if num1 == 1:                                        # 若最大公约数为1，则为最简真分数
            n += 1
            print("%2d/40 " %i, end=" ")
            if n % 8 == 0:                                   # 每8个一行
                print()
```

代码2（对应方法2）：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 列出真分数序列

if __name__ == "__main__":
    print("分母为40，分子小于40的最简分数有：")
    n = 0                                                    # 计数器，记录最简分数的个数
    for i in range(1, 40):                                   # 穷举40以内的全部分子
        num1 = 40                                            # 分母
        num2 = i                                             # 分子
        j =2
        while j <= num2:
            # 判断2～num2之间分子和分母是否有公约数
            # 如果有j满足条件，则结束循环，说明此时的分数不是最简分数
            if (num1 % j == 0) and (num2 % j == 0):
                break
            j += 1
        # 如果j > num2，说明2～num2之间没有分子和分母的公约数，分数为最简分数
        if j > num2:
            print("%2d/40  " %i, end=" ")
            n += 1
            if n % 8 == 0:                                   # 每行输出8个数
                print()
```

6. 运行结果

在PyCharm下运行程序，程序的运行结果如图4.5所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/code
分母为40，分子小于40的最简分数有：
 1/40    3/40    7/40    9/40   11/40   13/40   17/40   19/40
21/40   23/40   27/40   29/40   31/40   33/40   37/40   39/40

Process finished with exit code 0
```

图4.5　运行结果

7．拓展训练

按递增顺序依次列出所有分母小于等于40的最简真分数。

根据问题描述进行分析后可知，求分母为40、分子小于40的最简分数的算法如上所述；当分母为30，求分子小于30的最简分数思想与上述思想相同，只需将num1的值改为30；分母为39，38，37，…，1时，最简分数也都可用上述方法求得。因此，要求分母小于等于40的最简真分数，只需在上述程序的基础上加一个外层循环来控制分母的取值即可，即添加循环“for k in range(1,40+1)”。

代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 列出真分数序列

if __name__ == "__main__":
    print("分母小于等于40的最简分数有：")
    for k in range(1, 40+1):
        num1 = k                                        # 分母
        n = 0                                                   # 计数器，记录最简分数的个数
        for i in range(1, num1):            # 穷举40以内的全部分子
            num2 = i                                    # 分子
            j = 2
            while j <= num2:
                # 判断2～num2之间分子和分母是否有公约数
                # 如果有j满足条件，则结束循环，说明此时的分数不是最简分数
                if (num1 % j == 0) and (num2 % j == 0):
                    break
                j += 1
            # 如果j > num2，说明2～num2之间没有分子和分母的公约数，分数为最简分数
            if j > num2:
                print("%2d/%2d " %(num2,num1) , end=" ")
                n += 1
                if n % 10 == 0:             # 每行输出8个数
                    print()
        print()
```

4.3　多项式之和

1．问题描述

计算下列多项式的值：

$$S=1+\frac{1}{1\times 2}+\frac{1}{1\times 2\times 3}+\cdots+\frac{1}{1\times 2\times 3\times\cdots 50}$$

2．问题分析

方法一：把上面多项式中的每一个分项标上记号，第一个式子的记号为1，第二个式子的记号为2，第三个式子的记号为3，以此类推。可以发现式子的分母与记号之间的关系如表4.2所示。

表4.2　分母与记号间的关系

项　数	1	2	3	4	5	……
分　母	1	1×2	1×2×3	1×2×3×4	1×2×3×4×5	……
	1!	2!	3!	4!	5!	……

通过表4.2可以发现，每一项分式的分母都是对应项标记的阶乘。所以只要求出每一项的阶乘再将其倒数和加在一起即为所求多项式的结果。

方法二：整体去看每一个分式之间的联系，每一个分式用一个字母来代替，第一个分式用t1表示，第二个分式用t2表示，以此类推，如表4.3所示。

表4.3　分式间的联系

项　数	1	2	3	4	……
分　式	$t1=1/1$	$t2=1/(1\times 2)$	$t3=1/(1\times 2\times 3)$	$t4=1/(1\times 2\times 3\times 4)$	……
	$t1=1\times 1$	$t2=t1\times(1/2)$	$t3=t2\times(1/3)$	$t4=t3\times(1/4)$	……

由表4.3可以看出，后面一项等于前一项乘以当前项数的倒数，如果这里的每一项都用同一个变量t表示，那么第i项就可以用公式表示为t=t*1/i。这样可以只用一层循环来实现，循环变量i控制对应的项数，取值范围为1～n。

3. 算法分析

n的阶乘算法为：n!=1×2×3×4×5×…×n，可用循环来实现。代码如下：

```
t = 1                                                    # t记录每一项分式的分母，每次循环之
前给t赋初值
j = 1
while j <= i:
    t = t * j                    # 每一项的阶乘
    j += 1
```

对于每一项都是求出其分母即项数i对应的阶乘，是一个循环重复的过程，故可用另外一层循环来控制项数，范围为1～n。代码如下：

```
i = 1
while i <= n:                    # i控制对应的项数
    …
```

对于存储阶乘的变量t，每一次记录新的阶乘之前都要把其值赋为1，否则下一项的阶乘值会受上一项的影响，所以在每次执行内层循环求下一项阶乘之前，要把t的值再次赋为1。

方法二的算法分析此处省略，后面将给出相关程序。

4. 确定程序框架

程序流程图如图4.6所示。

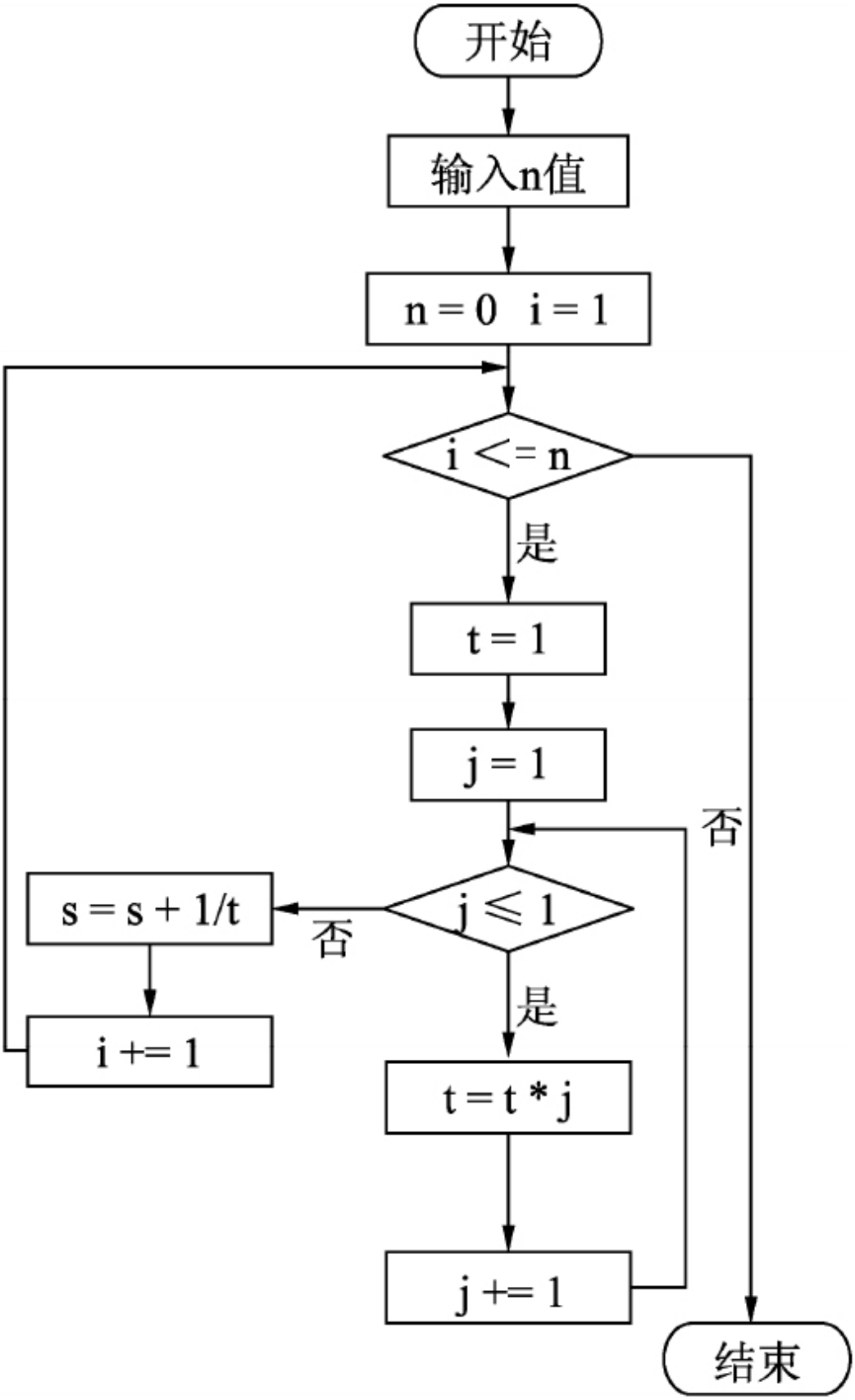


图4.6　程序流程图

5. 完整的程序

方法1：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 多项式之和

if __name__ == "__main__":
    n = int(input("请输入一个整数n: "))
    s = 0                                                    # s记录多项式的和
    i = 1
    while i <= n:                                          # i控制对应的项数
        t = 1                                                  # t记录每一项分式的分母，每次循
环之前给t赋初值
        j = 1
        while j <= i:
            t = t * j                                  # 每一项的阶乘
            j += 1
        s = s + 1/t
        i += 1
    print("%f" %s)
```

方法2：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 多项式之和

if __name__ == "__main__":
    s = 0                                                    # s记录多项式的和
    n = int(input("请输入一个整数n: "))
    t = 1
    for i in range(1, n+1):          # i<=n，i控制对应的项数
        t = t*1/i                                  #将分式的值赋给变量t
        s = s + t
    print("%f" %s)
```

6. 运行结果

在PyCharm下运行程序，屏幕上提示“请输入一个整数n:”，这里输入50，运行结果如图4.7所示。

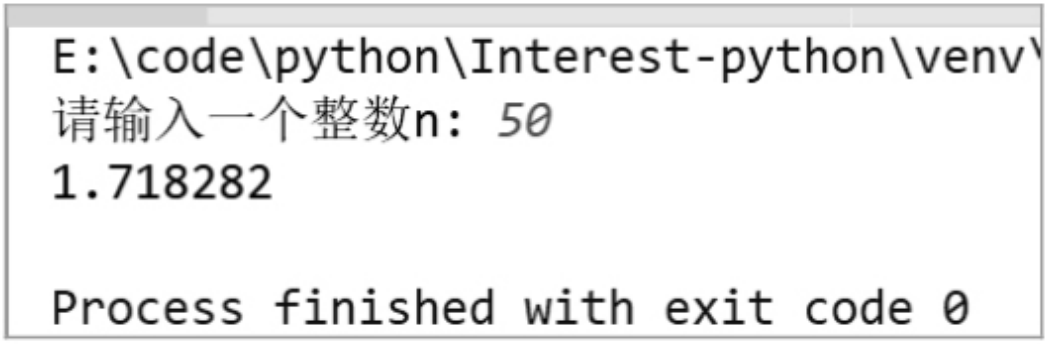

E:\code\python\Interest-python\venv\
请输入一个整数n: 50
1.718282
Process finished with exit code 0

图4.7　运行结果

4.4　最大公约数

1. 问题描述

求任意两个正整数的最大公约数（Greatest Common Divisor，GCD）。

2. 问题分析

如果有一个自然数a能被自然数b整除，则称a为b的倍数，b为a的约数。几个自然数公有的约数，叫作这几个自然数的公约数。公约数中最大的一个公约数，称为这几个自然数的最大公约数。

根据约数的定义可知，某个数的所有约数必不大于这个数本身，几个自然数的最大公约数必不大于其中任何一个数。要求任意两个正整数的最大公约数即求出一个不大于这两个数中的任何一个，但又能同时整除两个整数的最大自然数。

3. 算法设计

思路有两种，第一种，采用穷举法按从小到大（初值为1，最大值为两个整数当中较小的数）的顺序将所有满足条件的公约数列出，输出其中最大的一个；第二种，按照从大（两整数中较小的数）到小（最小的整数1）的顺序求出第一个能同时整除两个整数的自然数，即为所求。下面对第二种思路进行详细说明。

无论按照从小到大还是从大到小的顺序寻找最大公约数，最关键的是找出两数中较小的数。对于输入的两个正整数m和n，相同的数据可能因为输入顺序不同导致变量m、n中存储数据的大小不同，如m=8、n=4与m=4、n=8，但无论变量中值的大小顺序怎么样最后的结果应该是相同的。为了避免相同数据因输入顺序不同而出现不同的结果，也为了使程序具有一般性，对于每次输入的值先进行大小排序，规定变量m中存储大数、变量n中存储小数。

两个变量所存储内容互换需要借助一个中间变量来完成，若采用将第二个变量的值赋给第一个变量，然后再将第一个变量的值赋给第二个变量的方法是错误的，因为经过第一次赋值（即第二个变量的值赋给第一个变量）后，第一个变量中存储的内容被替换掉，原来的值已经找不到了，所以再次赋值时并没有把第一个变量中原来的值赋给第二个变量，而是将改变之后的值（即第二个变量的值）赋给了第二个变量。正确的代码如下：

```
if m < n:                                                    # 比较大小，使得m中存储大数，n
中存储小数
    # 交换m和n的值
    temp = m
    m = n
    n = temp
```

两个数的最大公约数有可能是其中较小的数，故在按从大到小的顺序寻找最大公约数时，循环变量i的初值从较小的数n开始依次递减，去寻找第一个能同时整除两个整数的自然数，并将其输出。需要注意的是，虽然判定条件是i>0，但在找到第一个满足条件的i值后，循环没必要继续进行，如25和15，最大公约数是5，对于后面的4、3、2、1便没必要再去执行，但此时判定条件仍然成立，要结束循环只能借助break语句。

```
# 按照从大到小的顺序寻找满足条件的自然
  数
i = n
while i > 0:
    if m % i == 0 and n % i == 0:
        # 输出满足条件的自然数并结束
          循环
        print("%d 和 %d 的最大公约数
是:  %d" %(m, n, i))
        break
    i -= 1
```

4. 确定程序框架

程序的流程图如图4.8所示。

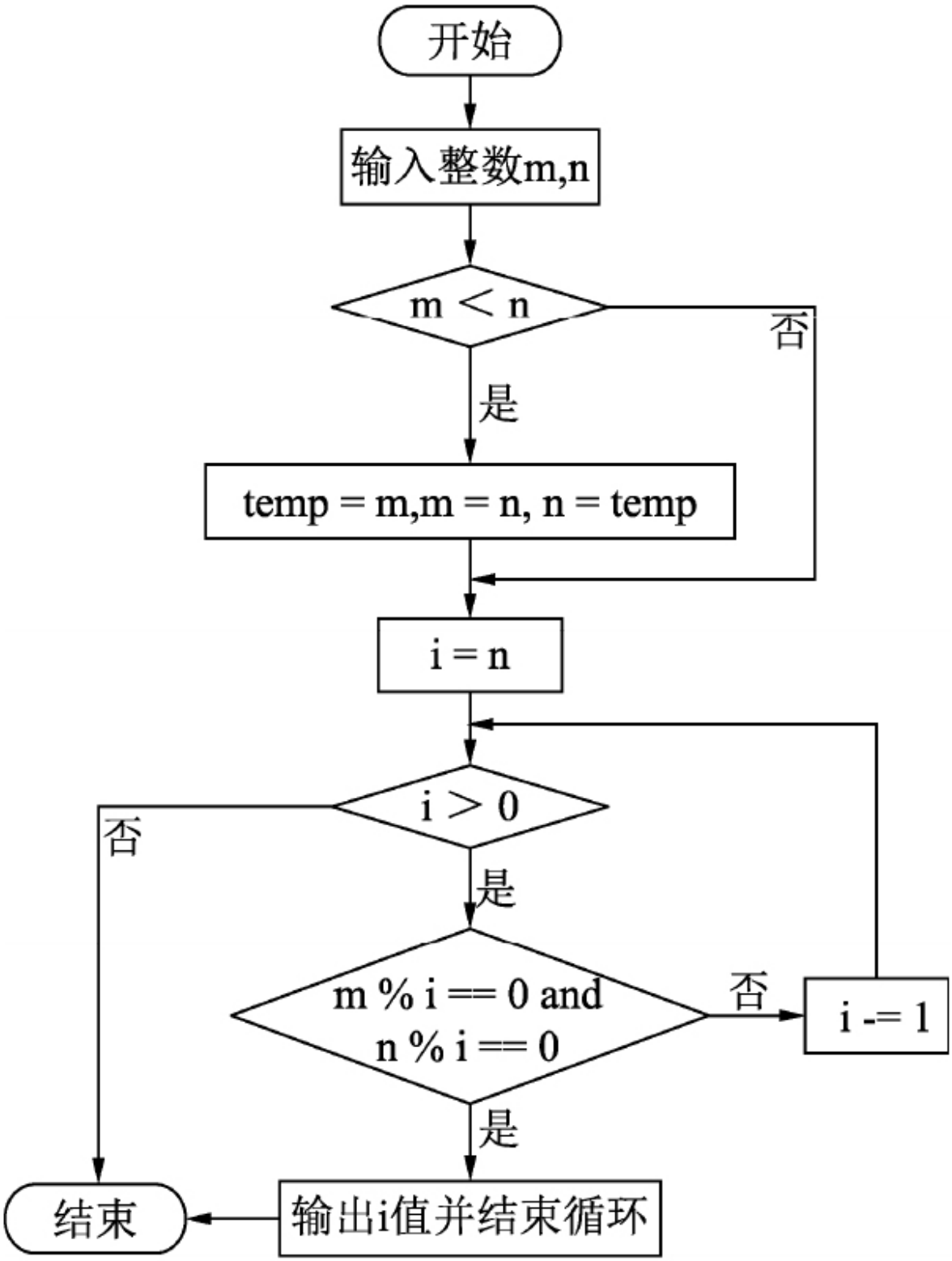


图4.8　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 最大公约数

if __name__ == "__main__":
    print("请输入两个整数")
    m = int(input("m = "))
    n = int(input("n = "))
    if m < n:                                                    # 比较大小，使得m中存储大
数，n中存储小数
        # 交换m和n的值
        temp = m
        m = n
        n = temp

    i = n                                                        # 按照从大到小的顺序寻
找满足条件的自然数
    while i > 0:
        if m % i == 0 and n % i == 0:
            # 输出满足条件的自然数并结束循环
            print("%d 和 %d 的最大公约数是： %d" %(m, n, i))
            break
        i -= 1
```

第一种思路，按照从小到大的顺序穷举两数公约数的程序代码如下：

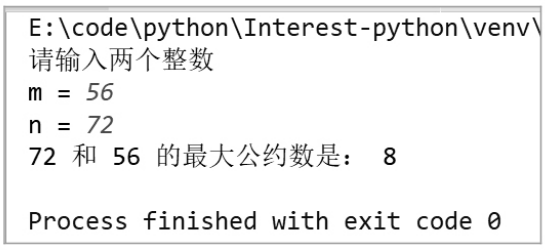
```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 最大公约数

if __name__ == "__main__":
    print("请输入两个整数")
    m = int(input("m = "))
    n = int(input("n = "))
    # 比较两个数的大小，进行交换
    if m < n:
        temp = n
        n = m
        m = temp

    for i in range(1, n):
        if m % i == 0 and n % i == 0:
            k = i                                                # 将当前情况下的最大公
约数存储在k中
    print("%d 和 %d 的最大公约数是：%d" %(m, n, k))
```

此算法第一步也需要对两个变量的值进行大小比较，使得变量m中存储大数，n中存储小数。程序中需要注意的是，最后输出结果时不能直接输出变量i的值，因循环结束时循环变量i的值为n，并不一定是要求的最大公约数。为了使求得的公约数在穷举过程中被记

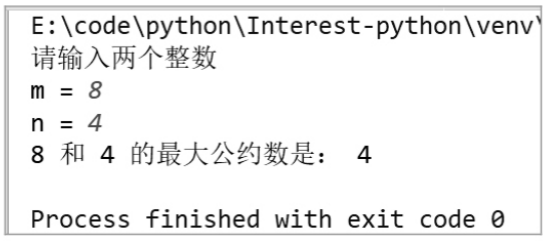
录，可以将满足条件的i值暂存到变量k中，使得k中始终存储当前情况下的最大公约数。

6. 运行结果

在PyCharm下运行程序，屏幕上提示“请输入两个整数”。首先输入56和72，运行结果如图4.9a所示，因为有可能两个数中较小的数正好是两数的最大公约数，故再输入一组数8和4，运行结果如图4.9b所示。



a）运行结果 1



b）运行结果 2

图4.9　运行结果

7. 问题拓展

早在公元前300年左右，欧几里得就在他的著作《几何原本》中给出了求最大公约数高效的解法——辗转相除法。辗转相除法用到的原理很简单，假设用f(x,y)表示x和y的最大公约数，取k=x/y，b=x%y，则x=ky+b，如果一个数能够同时整除x和y，则必能同时整除b和y；而能够同时整除b和y的数也必能同时整除x和y，即x和y的公约数与b和y的公约数是相同的，其最大公约数也是相同的，则有f(x,y)=f(y,x%y)（y>0），如此便可把原问题转换为求两个更小数的最大公约数，直到其中一个数为0，剩下的另外一个数就是两者最大的公约数。

例如，12和30的公约数有：1、2、3、6，其中6就是12和30的最大公约数。

欧几里德算法，其思想可概括如下：

1）用较大的数m除以较小的数n，得到的余数存储到变量b中，b=m%n。

2）上一步中较小的除数n和得出的余数b构成新的一对数，并分别赋值给m和n，继续做上面的除法。

3）若余数为0，其中较小的数（即除数）就是最大公约数。否则重复步骤1和步骤2。

以求288和123的最大公约数为例，操作如下：

288÷123=2余42

123÷42=2余39

42÷39=1余3

39÷3=13

所以3就是288和123的最大公约数。

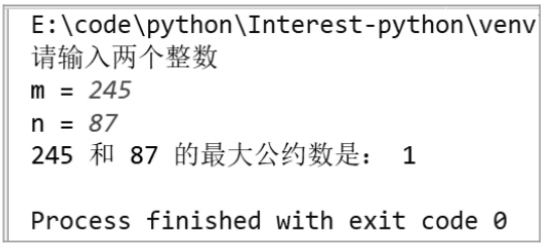
在进行辗转相除之前同样要确定两数中的大数和小数，将其分别存放在不同变量中。

相应程序段如下：

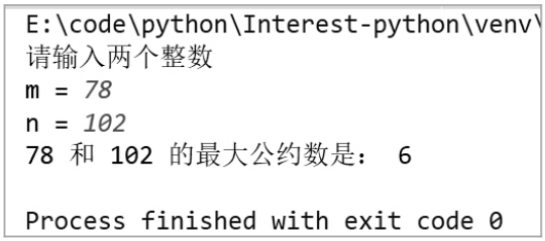
```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 最大公约数—辗转相除法

if __name__ == "__main__":
    print("请输入两个整数")
    m = int(input("m = "))                                  # m存储较大数
    n = int(input("n = "))                                  # n存储较小数
    print("%d 和 %d 的最大公约数是： " % (m, n), end="")
    # 比较两个数的大小，进行交换，使得m是最大数，n是最小数
    if m < n:
        temp = n
        n = m
        m = temp
    b = m % n                                               # b存储m 和
n取模得到的余数
    while b != 0:
        m = n                                               # 原来的小数
作为下次运算时的大数
        n = b                                               # 将上一次的
```

```
    余数作为下次相除时的小数
            b = m % n
        print("%d" %n)
```

在PyCharm下运行上面的程序，屏幕上提示“请输入两个整数”，输入245和87后，结果为1，如图4.10a所示；输入78和102后，结果为6，如图4.10b所示。



a）运行结果 1



b）运行结果 2

图4.10　运行结果

8. 拓展训练

求一个最小的正整数，这个正整数被任意n（2≤n≤10）除都是除不尽的，而且余数总是n-1。例如，被9除时的余数为8。要求设计一个算法，不得使用枚举与“除2，除3，…，除9，除10”有关的命令，求出这个正整数。

4.5　最小公倍数

1. 问题描述

求任意两个正整数的最小公倍数（Least Common Multiple，LCM）。

2. 问题分析

如果有一个自然数a能被自然数b整除，则称a为b的倍数，b为a的约数，对于两个整数来说，最小公倍数是指这两个数共有倍数中最小的一个。计算最小公倍数时，通常会借助最大公约数来辅助计算，即最小公倍数=两数的乘积/最大公约（因）数，解题时要避免和最大公约（因）数问题混淆。

对于最小公倍数的求解，除了利用最大公约数外还可根据定义进行算法设计。要求任意两个正整数的最小公倍数，就是求出一个最小的能同时被两个整数整除的自然数。

3. 算法设计

根据定义可知，两个整数的最小公倍数不小于两数中的任意一个，若大数不是小数的倍数，则可由大数开始利用递增的方法找到第一个满足条件的数。利用定义求最小公倍数的关键是找到两个整数中较大的数。

对于输入的两个正整数m和n，每次输入的大小顺序可能不同，为了使程序具有一般性，首先对整数m和n进行大小排序，规定变量m中存储大数、变量n中存储小数。

若输入时m的值小于变量n的值，则需要交换两个变量中存储的内容。再次强调，交换两个变量中的内容并不是简单地相互赋值，而要借助中间变量，将其中一个变量的值暂存（防止在交换过程中

将原来的内容丢失）。此过程在4.4节及第1章1.8节中已介绍过，这里不再赘述。

```
if m < n:                                   # 比较两个数的大小，使得m中存储大数，n中存储小数
    temp = m
    m = n
    n = temp
```

若输入的两个数，大数m是小数n的倍数，那么大数m即为所求的最小公倍数；若大数m不能被小数n整除，则需要寻找一个能同时被两数整除的自然数，即从大数m开始依次向后递增直到找到第一个能同时被两数整除的数为止，故循环变量i的初值为m。需要注意的是，在找到第一个满足条件的i值后，循环没必要继续进行，故用break来结束循环。

在上面的分析过程中没有提到循环变量的终止条件，因i的最大值不能确定，像这种终止条件不确定的情况如何来表示呢？方法有两种，第一，可以把判定条件表示成循环变量满足的基本条件，如本例终止条件可表示成i>0；第二，终止条件省略不写，利用循环体中的语句结束循环，如在找到第一个满足条件的自然数时利用break语句结束循环。

```
i = m
while i > 0:                                # 从大数开始寻找满足条件的自然数
    if i % m == 0 and i % n == 0:
        # 输出满足条件的自然数并结束循环
        print("%d 和 %d 的最小公倍数为：%d" %(m, n, i))
        break
    i += 1
```

通常情况下，如果我们知道循环的次数，可以使用for循环来实现；如果不清楚循环的次数，可以使用while循环来实现，while循环的条件就是循环的终止条件。

4. 确定程序框架

程序流程图如图4.11所示。

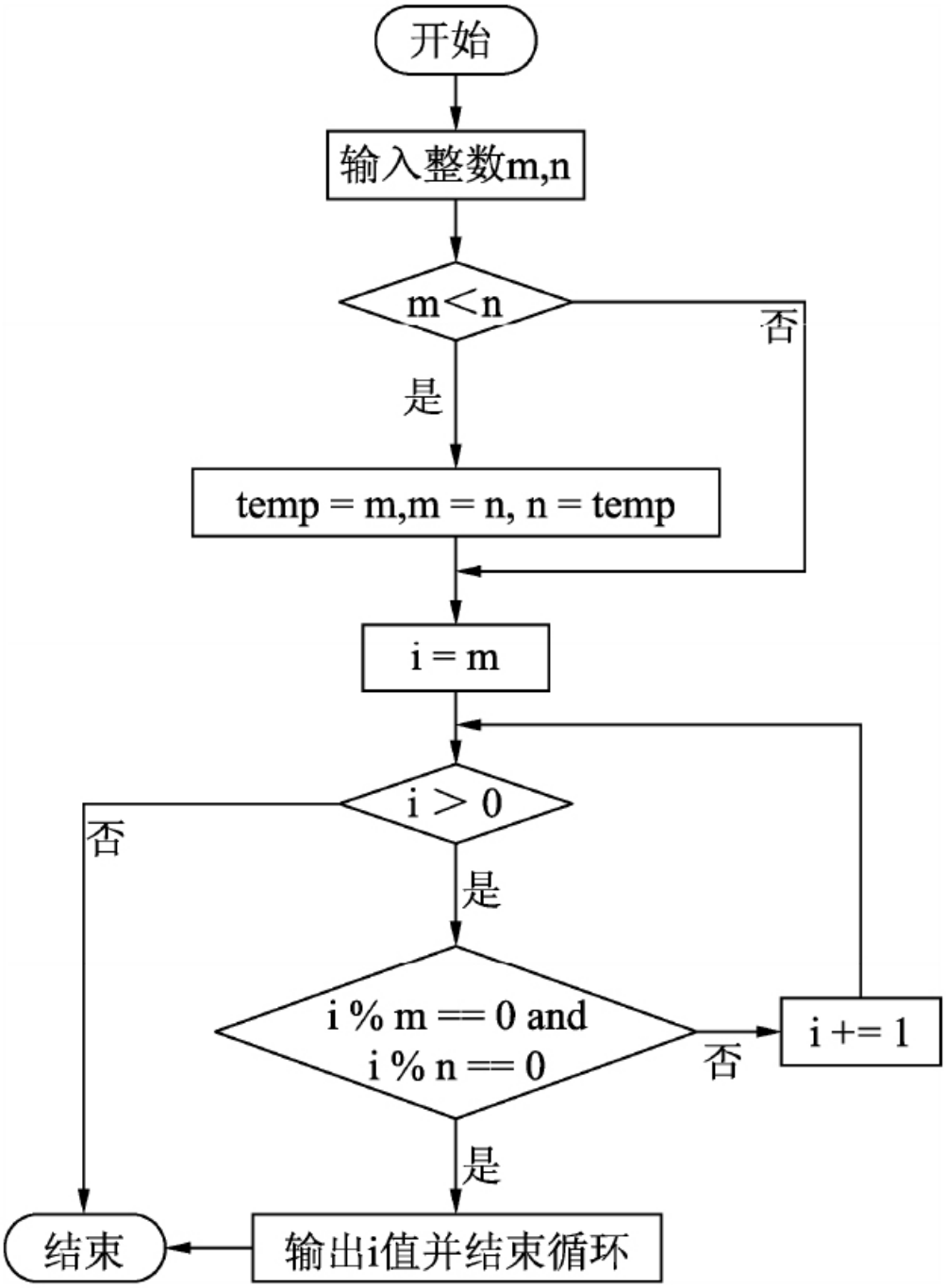


图4.11　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 最小公倍数
```

```
if __name__ == "__main__":
    print("请输入两个整数")
    m = int(input("m = "))
    n = int(input("n = "))
    if m < n:                                   # 比较两个数的大小，使得m中存储大数，n中存储
小数
        temp = m
        m = n
        n = temp
    i = m
    while i > 0:                                # 从大数开始寻找满足条件的自然数
        if i % m == 0 and i % n == 0:
            # 输出满足条件的自然数并结束循环
            print("%d 和 %d 的最小公倍数为：%d" %(m, n, i))
            break
        i += 1
```

最小公倍数不可以像最大公约数那样直接利用辗转相除法求出，但可以借助辗转相除法求得的最大公约数来求最小公倍数。辗转相除法求最大公约数的代码在4.4节中已经给出，在已知最大公约数的情况下，借助公式最小公倍数=两数的乘积/最大公约（因）数，即可求出两整数的最小公倍数。

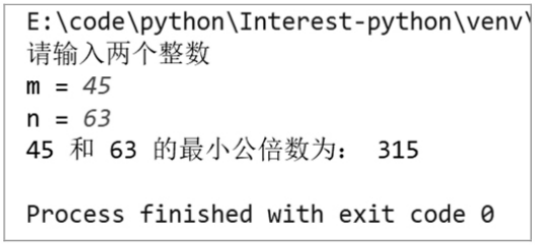
由4.4节中求解最大公约数的代码可知，在辗转相除的过程中，变量m、n的值是一直在变化的，程序结束时与最初输入的值已不同，但在求最小公倍数时要用到两变量初值的乘积，因此在进入循环进行辗转相除之前应先将两变量的乘积保存，假设存储到变量k中，根据公式用变量k的值除以求得的最大公约数n得到的值即为所求的最小公倍数。对应代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 最小公倍数—利用两数的最大公约数求出最小公倍数

if __name__ == "__main__":
    print("请输入两个整数")
    m = int(input("m = "))
    n = int(input("n = "))
    k = m * n                                       # k存储两数的乘积
    print("%d 和 %d 的最小公倍数为： " %(m, n), end="")
    if m < n:                                       # 比较两个数的大小，使得m存储大数，n存储小数
        temp = m
        m = n
        n = temp
    b = m % n                                       # b存储m除以n的余数
    while b != 0:
        m = n                                       # 原来的小数作为下次运算时的大数
        n = b                                       # 将上一次的余数作为下次相除时的小数
        b = m % n
    resultNum = k // n      # 两数乘积除以最大公约数即为它们的最小公倍数
    print("%d" %resultNum)
```

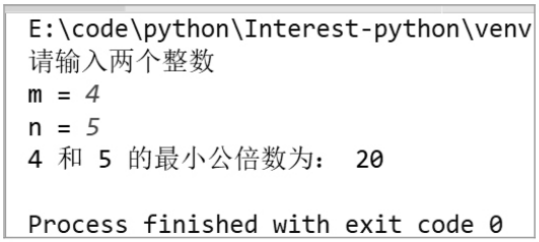
当然，根据定义求得的最大公约数，也可由公式最小公倍数=两数的乘积/最大公约（因）数，再次求出最小公倍数。方法同上述代码。

6. 运行结果

在PyCharm下运行程序，屏幕上提示“请输入两个整数”，输入45和63后，结果为315，如图4.12a所示；输入4和5后，结果为20，如图4.12b所示。



a）运行结果 1



b）运行结果 2

图4.12　运行结果

4.6　歌星大奖赛

1．问题描述

在歌星大奖赛中，有10个评委为参赛的选手打分，分数为1～100分。选手最后得分为：去掉一个最高分和一个最低分后其余8个分数的平均值。请编写一个程序实现。

2．问题分析

求一组数中的最大值和最小值是Python语言中比较常见的一类问题，这类问题的算法十分简单，定义两个变量max、min分别存储最大值、最小值，利用两个变量与给定的数依次比较的方法求出最大值和最小值。但是要注重在程序中判定最大值和最小值的变量是如何赋值的。

3．算法设计

确定变量初值。变量max和min要分别与每个数进行比较，因此在第一次比较时要用到两变量的初值，那么max和min的初值赋多少合适呢？一般情况可按照下面的方法赋值，最大值max的初值尽量小，最小值min的初值尽量大。对于变量max来说，只有其初值尽可能小的时候，在第一次与给定的数比较时数1才会大于max，才能把数1赋给max，作为变量max的新值；接着与数2比较，若数2>max，同样把数2的值作为新值赋给max，若数2<max，则max中的值保持不变。重复上面的过程直到max与所有的数都比较完，则max中存储的就是最大值。若刚开始max的值就很大，那么在比较过程中，给定的数若都比当前max的值小，经过一轮比较结束时变量max中存储的仍然是最初所赋的初值，那么这样的比较是没有意义的。比较过程如表4.4（以5个数为例进行说明）所示。

表4.4　比较过程

数值 / 比较次数	max	给定数1	给定数2	给定数3	给定数4	给定数5
	68	67	56	79	83	70
第一次：max<数1	**67**	67	56	79	83	70
第二次：max>数2	**67**	67	56	79	83	70
第三次：max<数3	**79**	67	56	79	83	70
第四次：max<数4	**83**	67	56	79	83	70
第五次：max>数5	**83**	67	56	79	83	70

比较完之后max的值为83，正好是所给数中的最大值。对于变量min的比较与赋值过程同上，在最初赋值是为了保证给定的任意一个数都比min小，所以应该把min的初值赋得尽可能大。

对于10个评委的评分利用循环结构实现，循环变量i记录已经输入的评分的个数，初值为0，判定条件为i<10。评分的总和采用累加的方式存储到变量sum中，即循环体执行一次输入一个分数，接着将其累加到变量sum上，等到循环结束时，sum中即为所有评分的总和。求解最大值和最小值的过程与表4.4所示类似，每输入一个分数，就与当前的最大（小）值进行比较，若其大（小）于变量max（min）的值就把此分数赋值给max（min）。由上述过程可以看出，无论是输入分数、求解总和还是寻找最大（小）值，都可以在一个循环过程中实现，代码如下：

```
for i in range(1, 11):
print("第%d个评委打分：" %i, end="")
    integer = int(input())                    # 输入评委的评分
    sum += integer                                    # 计算总分
    if integer > max:                        # 通过比较筛选出其中的最高分
        max = integer
    if integer < min:                        # 通过比较筛选出其中的最低分
        min = integer
```

4. 程序框架

程序流程图如图4.13所示。

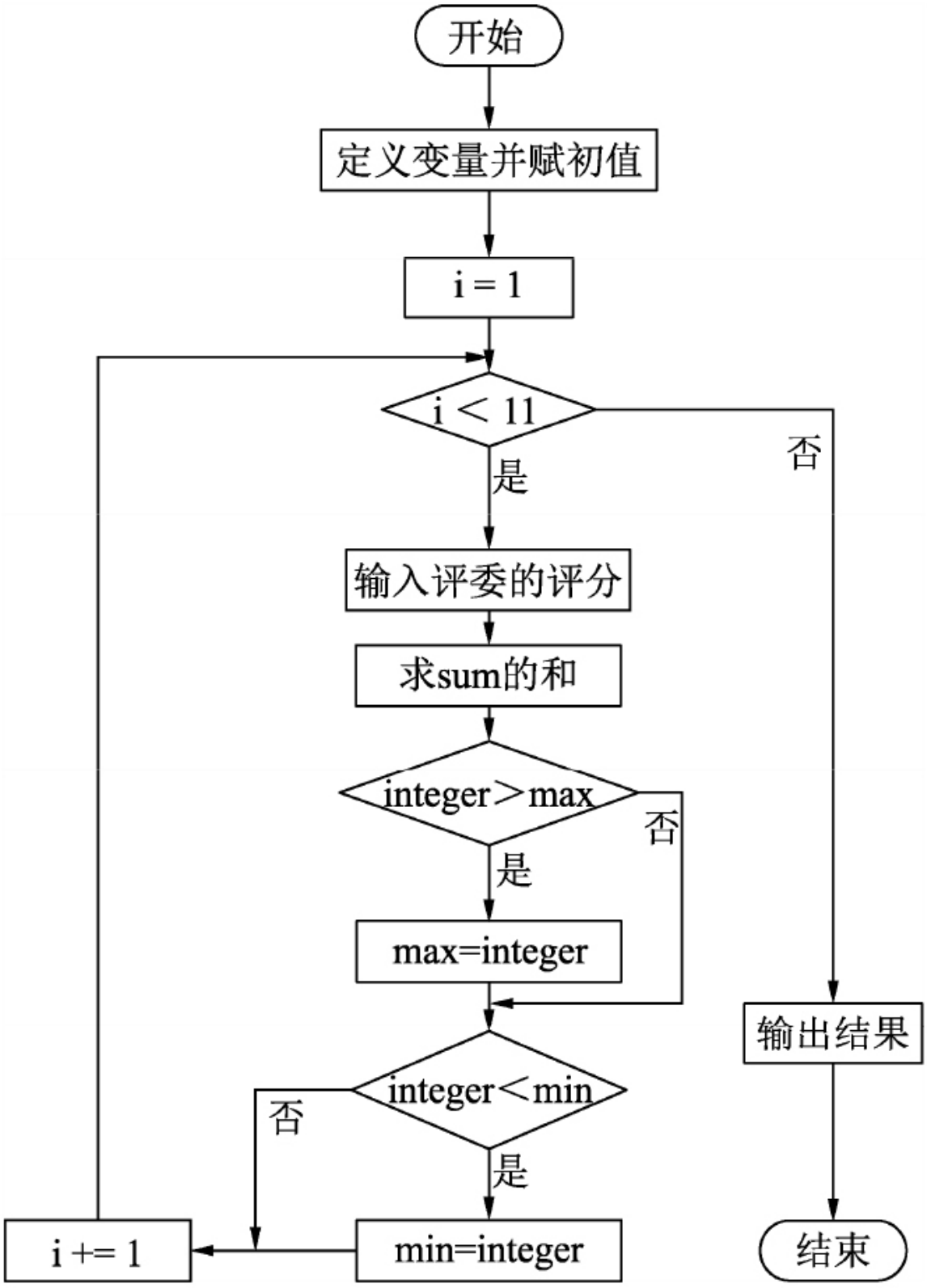


图4.13　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 歌星大奖赛

if __name__ == "__main__":
    max = 0
    min = 100
    sum = 0                                                                    #
sum存放10个评委打分的总分数
    for i in range(1, 11):
        print("第%d个评委打分: " %i, end="")
        integer = int(input())                                          # 输入评委的评分
        if integer < 0 or integer > 100:                          # 对分数值进行验证
            print("输入的分数错误")
            exit()
        sum += integer                                                     # 计算总分
        if integer > max:                                                 #通过比较筛选出
其中的最高分
            max = integer
        if integer < min:                                                 #通过比较筛选出
其中的最低分
            min = integer
    print("去掉一个最高分: %d" % max)
    print("去掉一个最低分: %d" % min)
    print("最后得分: %d" % ((sum - max - min) // 8))
```

6. 运行结果

在PyCharm下运行程序，按照屏幕提示输入10个分数，程序运行结果如图4.14所示。

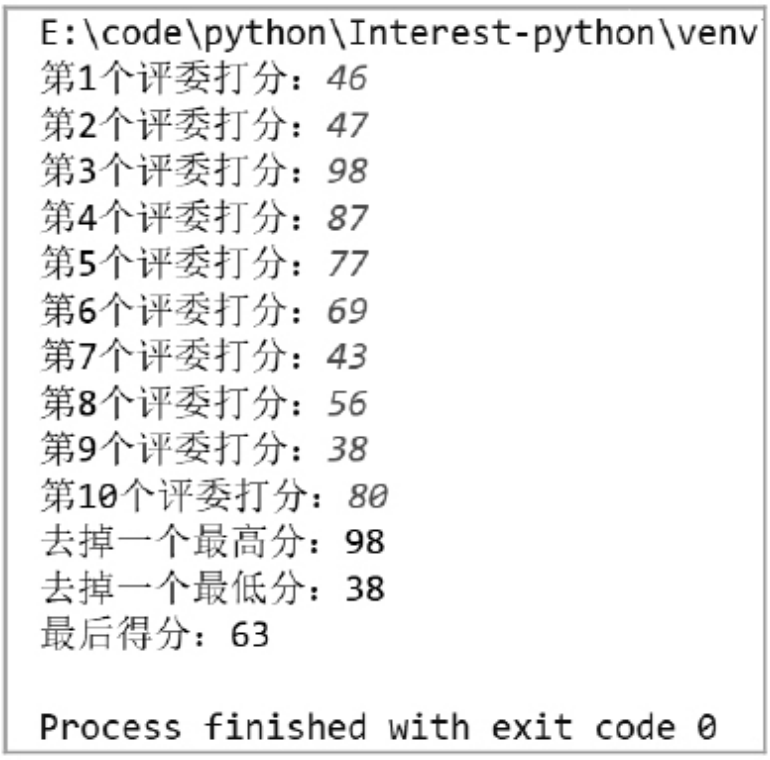


图4.14　运行结果

7．问题拓展

题目条件不变，但考虑同时对评委评分进行裁判，即在10个评委中找出最公平（即评分最接近平均分）和最不公平（即与平均分的差距最大）的评委，程序应该怎样实现？

要找出最公平与最不公平的评委，需要在求出平均值后将该值与所有分数进行比较，求出与平均值差值的绝对值最大和最小的两个评分，该评分所对应的评委即为所求。因有一比较过程，因此在输入完评委的评分后需要将其存储，若在上述代码基础上进行改进，则需要另外定义10个变量来存储评委评分。此算法虽然可以满足题目要求，但是写起来麻烦，为解决这个问题可以利用数组来实现，这样便可不必定义10个变量，只需要定义一个包含10个元素的数组，第1到第10个评委的评分分别存储到数组score[0]～score[10]。

最公平的评委即求出的与平均值差值最小的评分所对应的评委，若有一个评分正好等于平均分，则此分数对应的评委即为最公平的；若都不相同，则需要将差值进行比较选出最小值，算法与求一组数中最小值的思路相同。最不公平的评委一定在所求的最大值和最小值对应的评委中产生。

代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 歌星大奖赛

import random

# 求出最大分数值
def maxScore(score):
    max = score[0]                                                    #
给max赋初值
    m = 0                                                             # m
记录最大值的下标
    for j in range(1, 10):
        if max < score[j]:
            max = score[j]
            m = j                                                     # 记录最大值
的下标
    print("最大的分数为：%d" % max)
    return max,m

# 求出最小分数值
def minScore(score):
    min = score[0]                                                    #
给min赋初值
    n = 0                                                             # n
记录最小值的下标
    for j in range(1, 10):
        if min > score[j]:
            min = score[j]
            n = j                                                     # 记录最小值
的下标
    print("最小的分数为：%d" % min)
    return min,n


if __name__=="__main__":
    sum = 0                                                           #
记录10个评委打分的总分数
    score = [0]*10
    for i in range(10):
        score[i] = random.randint(0, 101)    # 生成10个随机分数
        sum = sum + score[i]
    print("10个评委的打分为：", score)
    max,m = maxScore(score)
    min,n = minScore(score)

    avg = (sum - max - min) // 8                        # 计算平均分
    print("去掉最高分和最低分，最后得分：%d" %avg)

    temp = 0                          # temp用来记录最公平与最不公平评委给出的评分存储的
下标
    s = abs(score[0] - avg)                                           # s记录评分与平均值差
值的绝对值
    for i in range(10):
```

```
        if abs(score[i] - avg) == 0:
            temp = i
            print("最公平的评委是：%d, 打分：%d" % ((temp + 1),(score[temp+1])))

    temp = 0
    for i in range(10):
        if abs(score[i] - avg) != 0:
            if s > abs(score[i] - avg):
                s = abs(score[i] - avg)
                temp = i
    print("最公平的评委是：%d" %(temp + 1))

    if (avg - min) == (max - avg):
        print("最不公平的评委是：%d %d" %((m+1), (n+1)))
    else:
        if (avg - min) > (max - avg):
            print("最不公平的评委就是：%d" %(n+1))
        else:
            print("最不公平的评委就是：%d" %(m+1))
```

在PyCharm下运行程序，运行结果如图4.15所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/
10个评委的打分为：  [22, 31, 57, 1, 69, 67, 29, 45, 38, 101]
最大的分数为：101
最小的分数为：1
去掉最高分和最低分，最后得分：44
最公平的评委是：8
最不公平的评委就是：10

Process finished with exit code 0
```

图4.15　运行结果

8. 知识点补充

random()是随机数库函数，它返回随机生成的一个实数，其值在[0,1)范围内。

random()函数是不能直接访问的，访问它需要导入random模块，然后通过random静态对象调用该方法。

使用如下语句导入random模块：

```
import random
```

在程序中导入random模块后，就可以通过random静态对象来调用random模块提供的随机函数了。

下面介绍random模块提供的常用随机函数。

• random.random()：该函数生成一个0到1之间的随机小数。

• random.randint()：该函数具有两个参数，一个是范围上限，一个是范围下限，用于随机生成指定范围内的整数，其中下限必须小于上限。

• random.uniform()：该函数具有两个参数，一个是范围上限，一个是范围下限，用于随机生成指定范围内的浮点数（小数），其中下限必须小于上限。

• random.randrange()：该函数具有三个参数，前面两个参数表示范围的上限和下限，第三个参数是一个递增值，用于生成指定范围内，以指定基数递增的随机数。

• random.choice()：该函数用于从给定的序列中随机获取一个元素返回。序列可以是字符串、列表、元组等。

• random.shuffle()：该函数用于将一个给定的列表元素打乱，随机排序。

• random.sample()：该函数具有两个参数，第一个参数表示指定序列，第二个参数是需获取的指定长度，用于从指定序列中随机获取指定长度的片段，原有的序列不会改变。序列可以是字符串、列表、元组等。

代码实例如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: random函数

import random

if __name__ == "__main__":
```

```
# 生成一个0到1之间的随机小数，赋值给a
a = random.random()
print("a = ", a)

# 生成一个[0,101)之间的随机整数，赋值给b
b = random.randint(0, 101)
print("b = ", b)

# 生成一个[0,10)之间的随机小数，赋值给c
c = random.uniform(0, 10)
print("c = ", c)

# 随机生成[0, 101)之间的偶数，递增数为2
d = random.randrange(0, 101, 2)
print("d = ", d)

# 随机生成一个字符
e = random.choice('abcdefg&#%^*f')
print("e = ", e)

# 随机返回一个字符串
f = random.choice(['apple', 'pear', 'peach', 'orange', 'lemon'])
print("f = ", f)

#  从给定的多个字符中，随机生成3个字符
g = random.sample('abcdefghijklmnopqrstuvwxyz',3)
print("g = ", g)

# 将一个列表中的元素打乱，随机排序
num = [9, 6, 4, 0, 2, 5, 3, 7, 1, 8]
num1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(num)
print("num = ", num)

random.shuffle(num1)
print("num1 = ", num1)
```

在PyCharm中运行程序，运行结果如图4.16所示。

```
E:\code\python\Interest-python\venv\Scripts\
a =  0.8316468007683989
b =  46
c =  8.180263116411941
d =  50
e =  c
f =  pear
g =  ['m', 'a', 'l']
num =  [4, 3, 2, 9, 5, 7, 6, 1, 0, 8]
num1 =  [5, 8, 7, 1, 9, 6, 3, 4, 10, 2]

Process finished with exit code 0
```

图4.16　运行结果

4.7　分数比较

1．问题描述

比较两个分数的大小。

2．问题分析

人工方式下比较分数大小最常用的方法是：进行分数的通分后比较分子的大小。通分的步骤可描述如下：

1）先求出原来几个分数（式）的分母的最简公分母。

2）根据分数（式）的基本性质，把原来分数（式）化成以最简公分母为分母的分数（式）。

例如，比较分数$\frac{7}{12}$ 与 $\frac{5}{7}$ 的过程如下：

1）两分数的分母没有公约数，故通分后最简公分母为两分母之积，即12*7=84。

2）分子为最简公分母除以原来分数的分母再乘以分子，即两分数的分子分别为84/12*7和84/7*5。

3）通分后两分数分别为$\frac{49}{84}$ 和 $\frac{60}{84}$ ，故 $\frac{7}{12}<\frac{5}{7}$ 。

若两分数的分母有公约数，则应求出通分后的最简公分母，即两分母之积/分母的最大公约数。

3．算法分析

由上述分析可知，求通分后的最简公分母，就是求两分母的最小公倍数。求最小公倍数的前提是求出两数的最大公约数，最大公约数的求解可采用辗转相除法，步骤如下：

1）用较大的数m除以较小的数n，得到的余数存储到变量b中，即b=m%n。

2）第1步中较小的除数n和得出的余数b构成新的一对数，并分别赋值给m和n，继续做上面的除法。

3）若余数为0，其中较小的数（即除数）就是最大公约数；否则重复步骤1和步骤2。

对于最大公约数的求解用自定义函数common_divisor(a,b)实现，程序代码如下：

```
# 求两个数的最大公约数
def common_divisor(a, b):
    # 若a < b，则交换两变量的值
    if a < b:
        temp = a
        a = b
        b = temp
    # 求分母a和b的最大公约数
    c = a * b
    while b != 0:
        d = b
        b = a % b
        a = d
    return c // a
```

通分后的分子为：通分后的分母/原分数分母*原分数分子。两分数的分母分别用变量j和l表示，分子用变量i和k表示，则求解分子的代码如下：

```
m = common_divisor(j, l) // j * i                    # 求出第一个分数通分后的分子
n = common_divisor(j, l) // l * k                    # 求出第二个分数通分后的分子
```

只需比较变量m、n的值即可，若m>n则第一个分数大于第二个分数；若m=n，两分数相等；否则第一个分数小于第二个分数。

4. 程序框架

程序流程图如图4.17所示。求最大公约数的流程图参照图4.8。

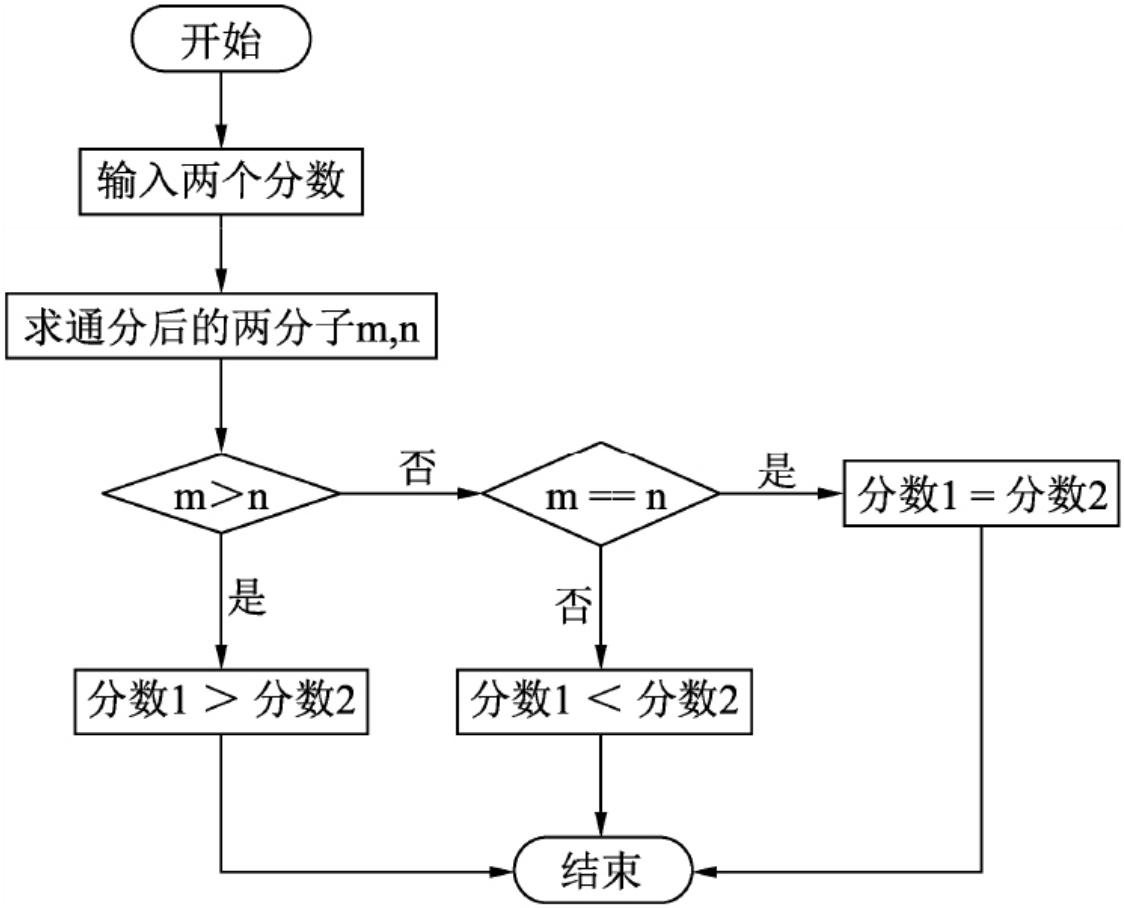


图4.17　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 分数比较

# 求两个数的最大公约数
def common_divisor(a, b):
    # 若a < b，则交换两变量的值
    if a < b:
        temp = a
        a = b
        b = temp
    # 求分母a和b的最大公约数
    c = a * b
    while b != 0:
        d = b
        b = a % b
        a = d
    return c // a


if __name__ == "__main__":
    print("请分别输入两个分数：")
```

```
i, j = [int(i) for i in input("请输入第一个分数: ").split()]
k, l = [int(i) for i in input("请输入第二个分数: ").split()]
print("第一个分数: %d/%d" %(i, j))
print("第二个分数: %d/%d" %(k, l))
m = common_divisor(j, l) // j * i        # 求出第一个分数通分后的分子
n = common_divisor(j, l) // l * k        # 求出第二个分数通分后的分子
if m > n:
    print("%d/%d > %d/%d" %(i,j,k,l))
else:
    if m == n:
        print("%d/%d = %d/%d" %(i,j,k,l))
    else:
        print("%d/%d < %d/%d" %(i,j,k,l))
```

6. 运行结果

为了验证程序的正确性，对于大于、小于和等于三种情况分别进行验证，以及分数有无公约数两种情况也分别进行验证，选择的三组数据分别为4/5与6/7、8/4与16/32、16/32与4/8，运行结果如图4.18所示。

```
E:\code\python\Interest-python\venv
请分别输入两个分数:
请输入第一个分数: 4 5
请输入第二个分数: 6 7
第一个分数: 4/5
第二个分数: 6/7
4/5 < 6/7

Process finished with exit code 0
```

a）运行结果 1

```
E:\code\python\Interest-python\venv
请分别输入两个分数:
请输入第一个分数: 8 4
请输入第二个分数: 16 32
第一个分数: 8/4
第二个分数: 16/32
8/4 > 16/32

Process finished with exit code 0
```

b）运行结果 2

```
E:\code\python\Interest-python\venv
请分别输入两个分数:
请输入第一个分数: 16 32
请输入第二个分数: 4 8
第一个分数: 16/32
第二个分数: 4/8
16/32 = 4/8

Process finished with exit code 0
```

c）运行结果 3

图4.18　运行结果

4.8　计算分数的精确值

1．问题描述

使用数组精确计算M/N（0<M<N≤100）的值。假如M/N是无限循环小数，则计算并输出它的第一循环节，同时要求输出循环节的起止位置（小数位的序号）。

2．问题分析

由于计算机字长的限制，常规的浮点运算都有精度限制，为了得到高精度的计算结果，就必须自行设计实现方法。

为了实现高精度的计算，可将商存放在一维数组中，数组中的每个元素存放一位十进制数，即商的第一位存放在第一个元素中，商的第二位存放在第二个元素中，以此类推。这样就可以使用数组来表示一个高精度的计算结果。

进行除法运算时可以模拟人工操作，即每次求出商的第一位后，将余数乘以10，再计算商的下一位，重复以上过程，当某次计算后的余数为0时，表示M/N为有限不循环小数。某次计算后的余数与前面的某个余数相同时，则M/N为无限循环小数，从该余数第一次出现之后所求得的各位数就是小数的循环节。

程序具体实现时，采用了数组和其他一些技巧来保存除法运算所得到的余数和商的各位数。

3．算法分析

在运算过程中，每次得到的余数都要与前面运算过程产生的余数进行比较，看否已经出现过，故余数及商都要存储在数组中。分别定义两个数组remainder[101]和quotient[101]，用来存放运算过程中每一步的余数，及得到的每一位商。

被除数、除数分别用变量m、n存储，由题意知m<n。数组元素remainder[m]的下标表示运算过程中得到的余数，remainder[m]=i语句的作用是：记录余数m所存储的位置i，即小数点后的位置。quotient[i]=m/n把第i次相除得到的商存储到数组中下标为i的位置。因每次除完之后，得到的余数肯定比除数n小，所以在下一次进行相除之前余数应先乘以10再运算。代码如下：

```
remainder[m] = i                    # m：得到的余数；remainder[m]：该余数对应的商的位数
m *= 10                                 # 余数扩大10倍
quotient[i] = m // n     # 商
m = m % n                               # 求余数
```

对得到的余数m进行判断，看m的值是否为0，若为0则说明此分数是有限循环小数，结束循环。若m的值不为0，则要判断此余数是否已经出现过（定义时将数组remainder中元素的值全部初始化为0，若remainder[m]的值为0则说明此余数没有出现过，若值不为0则说明前面已出现过相同的余数，此分数为无限循环小数），若没有出现过则继续相除，若已经出现过则说明分数为无限循环小数，第一个循环节的起始位置为数组quotient[1]的结束位置，数组元素remainder[m]对应的下标i。代码如下：

```
if m == 0:                                                  # 余数为0则表示是有限小数
    j = 1
    while j <= i:
        print("%d" %quotient[j], end="")                    # 输出商
        j += 1
    break
# 退出循环

if remainder[m] != 0:                                  # 若该余数对应的位在前面已经出现过
    j = 1
    while j <= i:
        print("%d" %quotient[j], end="")                    # 输出循环小数
        j += 1
    print("\n分数的第一个循环节：%d" %remainder[m])
    print("循环节的起始位置为：%d" %i)
    break
```

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 计算分数精确值

if __name__ == "__main__":
    remainder = [0]*101                                    # remainder存放除法的余数
    quotient = [0]*101                                     # quotient 依次存放商的每一位
    print("请输入分子与分母大于0，小于等于100的分数")
    m = int(input("分子m = "))                               # 分子
    n = int(input("分母n = "))                               # 分母
    print("输入的分数为：%d/%d" %(m, n))
    print("%d/%d 的准确值是：0." %(m, n), end="")
    for i in range(1, 101):                                # i商的位数
        remainder[m] = i    # m：得到的余数；remainder[m]：该余数对应的商的位数
        m *= 10                                                                  # 余数扩大
10倍
        quotient[i] = m // n                               # 商
        m = m % n                                                                # 求余数
        if m == 0:                                                               # 余数为0则
表示是有限小数
            j = 1
            while j <= i:
                print("%d" %quotient[j], end="")             # 输出商
                j += 1
            break
# 退出循环

        if remainder[m] != 0:                              # 若该余数对应的位在前面已经出
现过
            j = 1
            while j <= i:
                print("%d" %quotient[j], end="")     # 输出循环小数
                j += 1
            print("\n分数的第一个循环节：%d" %remainder[m])
            print("循环节的起始位置为：%d" %i)
            break
```

---

5. 运行结果

在PyCharm下运行程序，屏幕上提示“请输入分子与分母大于0，小于等于100的分数”，在输入一个有限分数1/40后，运行结果如图4.19所示。

输入一个有限分数1/6后，运行结果如图4.20所示。

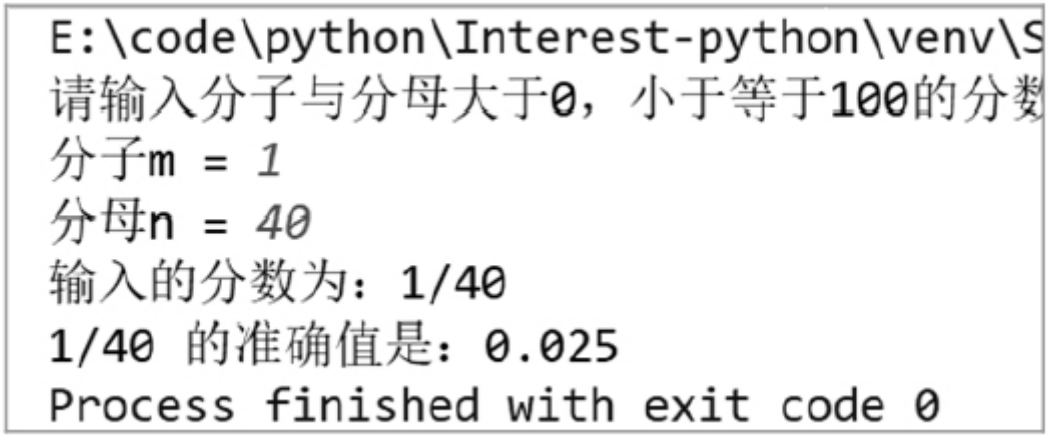


图4.19　运行结果1

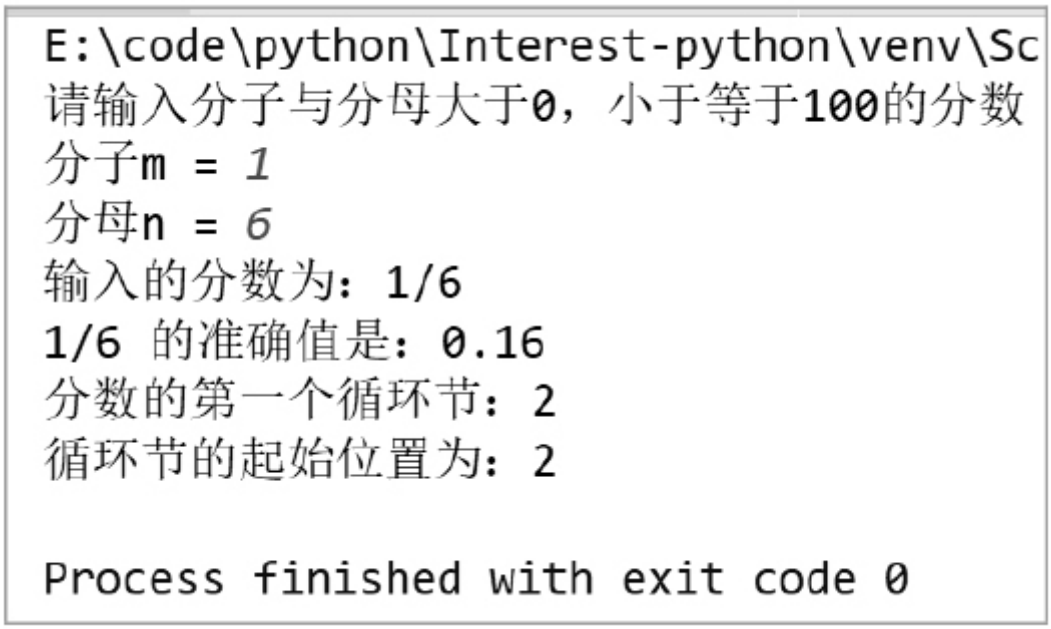


图4.20　运行结果2

输入一个有限分数25/95后，运行结果如图4.21所示。

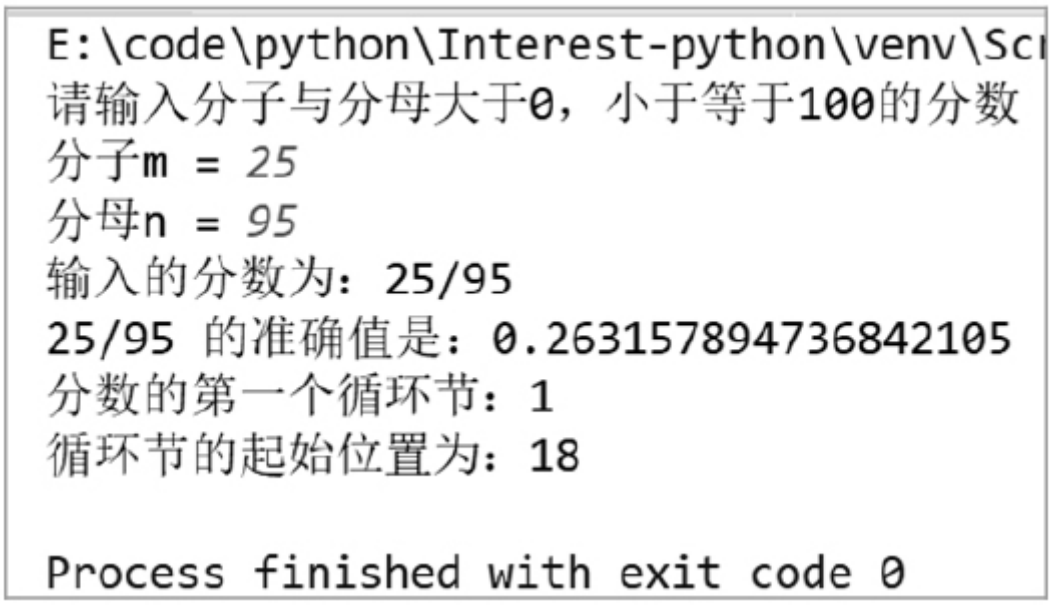

E:\code\python\Interest-python\venv\Sc
请输入分子与分母大于0，小于等于100的分数
分子m = 25
分母n = 95
输入的分数为：25/95
25/95 的准确值是：0.26315789473684210 5
分数的第一个循环节：1
循环节的起始位置为：18
Process finished with exit code 0

图4.21　运行结果3

第5章　趣味素数

素数是指除了1和它本身以外再没有其他因子的自然数。在数论中，素数是最纯粹也最令人着迷的概念。在所有的素数中，只有2是唯一的一个偶数，其他的素数都是奇数。

本章首先介绍如何判断一个自然数是否为素数，然后在掌握素数判断方法的基础上介绍几种有趣的素数，这些素数都各有特点，但它们的共同点是都用到了素数的判断方法。因此，对于素数部分的学习，读者应该熟练掌握素数的概念和判断方法，再通过本章介绍的几种特殊的素数来开阔思路，提高学习兴趣。

5.1 素数

1. 问题描述

求给定范围start～end之间的所有素数。

2. 问题分析

素数指的是只能被1和它自身整除的整数。

判定一个整数m是否为素数的关键，就是要判定整数m能否能被除1和它自身以外的其他任何整数所整除，若都不能整除，则m为素数。

本题求的是给定范围start～end之间的所有素数，考虑到程序的通用性，需要从键盘输入start和end的值，例如输入start=1，end=1000，则所编写的程序应能够打印出1～1000之间的所有素数。

3. 算法设计

由问题分析可知，该问题考虑用双层循环结构实现。

外层循环对start～end之间的每个数进行迭代，逐一检查其是否为素数。外层循环的循环变量用变量m表示，m即代表当前需要进行判断的整数，显然其取值范围为start≤m≤end。

内层循环稍显复杂，完成的功能是判断当前的m是否为素数。设内循环变量为i，程序设计时i从2开始，直到$\sqrt{m}$ 为止。用i依次去除需要判定的整数m，如果m能够被2～ $\sqrt{m}$ 中的任何一个整数所整除，则表示i必然小于或等于 $\sqrt{m}$ ，并可以确定当前的整数m不是素数，因此应提前结束该次循环。如果m不能被 $2\sim\sqrt{m}$ 中的任何一个整数所整除，则在完成最后一次循环后，i还需要加1，即 $i=\sqrt{m}+1$ ，之后才终止循环。此时，可以确定当前的整数m为素数。

我们可以使用标志位flag来监控内外循环执行的情况。在定义变量时将flag初值设为1，在内层循环中判断时，如果m能够被$2\sim\sqrt{m}$中的任何一个整数所整除，则在内循环中将flag设置为0。如果m不能被$2\sim\sqrt{m}$中的任何一个整数所整除，则在内循环中不会修改flag标志的值，退出内循环后它的值仍为1。此时在外循环中对flag的值进行判断，如果flag=0，则显然当前的m不是素数，如果flag=1，则当前的m是素数，应该将其打印出来。

还需要注意的是，在外循环中，每次要进行下一次迭代之前，要先将flag标志再次置为1。

在设计算法时我们还可以定义count变量用来保存最后求得的start～end之间素数的总个数。

4．确定程序框架

1）输入start和end并判断输入是否合法。输入start和end值，并使用while语句来判断start和end的值是否满足条件start>0且start<end，如果满足则输入范围合法，否则重新输入。

```
print("请输入一个整数范围(start-end): ")
start = int(input("start = "))
end = int(input("end = "))
while not (start>0 and start<end):                    #判断输入范围是否正确
    print("输入的参数有误，请重新输入: " )
    start = int(input("start = "))
    end = int(input("end = "))
```

2）求素数。在算法设计中我们已经知道，求指定范围内的素数需要使用双重循环，下面的代码描述了求素数的过程。

```
# 外层循环，对start～end之间的每个数进行迭代，检查是否为素数
m = start
while m <= end:
    k = math.sqrt(m)                                          # 求m的平方根
    i = 2
    while i <= k:                                        # 内层循环，判断2～k之间的每个数是否能被m整除
        # 若存在一个数能被m整除，则跳出内层循环
        # 否则，m是素数，打印m
        i += 1
    m += 1
```

3）每输出15个素数则换行。定义变量count，初值为0。使用count计数，每找到一个素数就将count值加1，接着判断当前count值是否能被15整除，若能整除则换行，否则在当前行继续打印。代码如下：

```
    count += 1
    if count % 15 == 0:                                  # 每15个素数换一行
        print()
```

程序的流程图如图5.1所示。

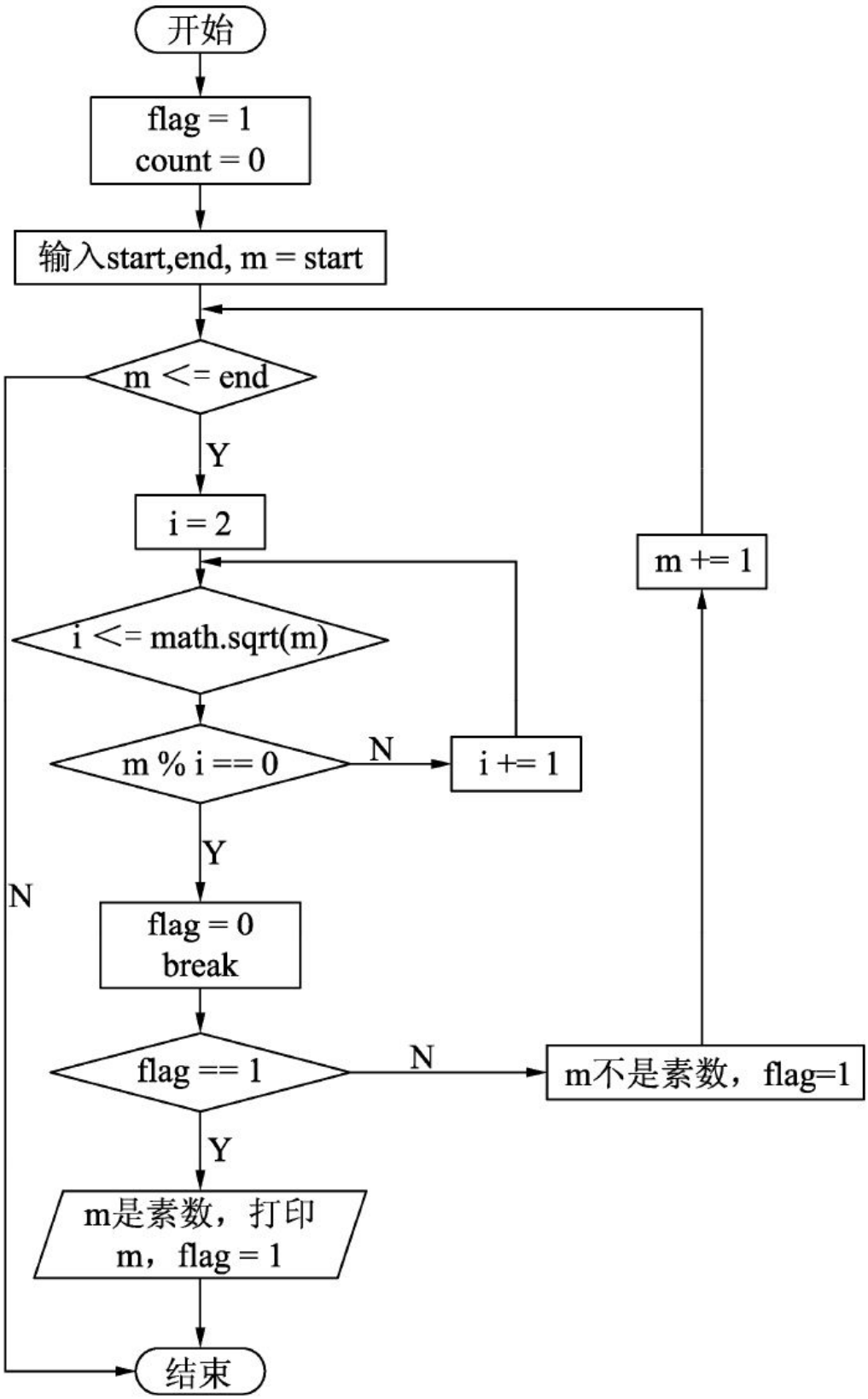


图5.1　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 素数

import math

if __name__ == "__main__":
    flag = 1
    count = 0
    print("请输入一个整数范围(start-end): ")
    start = int(input("start = "))
    end = int(input("end = "))
    while not (start>0 and start<end):                                    #判断输入范围是否正
确
        print("输入的参数有误，请重新输入：" )
        start = int(input("start = "))
        end = int(input("end = "))

    print("%d 和 %d 之间的素数有：" %(start, end))
    # 外层循环，对start～end之间的每个数进行迭代，检查是否为素数
    m = start
    while m <= end:
        k = math.sqrt(m)             # 求m的平方根
        i = 2
        while i <= k:                       # 内层循环，判断2～k之间的每个数是否能被m整
除
            if m % i == 0:           # 若存在一个数能被m整除，则跳出内层循环
                flag = 0
                break
            i += 1
        if flag == 1:                           # 如果flag == 1，则当前的m为素数
            print("%-4d" %m, end="")
            count += 1
            if count % 15 == 0:      # 每15个素数换一行
                print()
        flag = 1
        m += 1
    print("\n%d 到 %d之间共有：%d 个素数" %(start, end, count))
```

6. 运行结果

在PyCharm下运行程序，屏幕上提示“请输入一个整数范围（start-end）：”，输入1和1000，即打印1～1000之间的所有素数，运行结果如图5.2所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/coc
请输入一个整数范围(start-end):
start = 1
end = 1000
1 和 1000 之间的素数有:
1   2   3   5   7   11  13  17  19  23  29  31  37  41  43
47  53  59  61  67  71  73  79  83  89  97  101 103 107 109
113 127 131 137 139 149 151 157 163 167 173 179 181 191 193
197 199 211 223 227 229 233 239 241 251 257 263 269 271 277
281 283 293 307 311 313 317 331 337 347 349 353 359 367 373
379 383 389 397 401 409 419 421 431 433 439 443 449 457 461
463 467 479 487 491 499 503 509 521 523 541 547 557 563 569
571 577 587 593 599 601 607 613 617 619 631 641 643 647 653
659 661 673 677 683 691 701 709 719 727 733 739 743 751 757
761 769 773 787 797 809 811 821 823 827 829 839 853 857 859
863 877 881 883 887 907 911 919 929 937 941 947 953 967 971
977 983 991 997
1 到 1000之间共有: 169 个素数

Process finished with exit code 0
```

图5.2　运行结果

7. 问题拓展

在问题分析中，我们指出素数是只能由1和它自身整除的整数。判定一个整数m是否为素数的关键就是要判定整数m能否被1和它自身以外的任何其他整数所整除，若都不能整除，则m为素数。因此，实际上求素数最直观的方法就是用m依次除以2～m之间的所有整数，从而做出判断。

因此最简单直观的判断方法就是使用下面的代码，在下面的代码中循环变量i的变化范围就是2～m。

```
# 判断m是否为素数
i = 2
while i <= m:
    if m % i == 0:
        print("%d 不是素数! " %m)
        break
    i = i + 1
else:
    print("%d 是素数! " %m)
```

而我们在解决该问题时循环变量i的变化范围是$2\sim\sqrt{m}$，显然该范围需要判断的次数要小于对2～m之间的每个数进行判断的次数。

之所以只需要对$2\sim\sqrt{m}$之间的每个数进行判断，其依据为如下定理：

如果m不是素数，则m必有满足1<i≤math.sqrt(m)的一个因子i存在。

根据上面的定理，我们就可以有效地减少循环判断的次数，即将循环变量i的变化范围改为2～sqrt(m)，从而提高算法的效率。代码如下：

```
# 判断m是否为素数
k = int(math.sqrt(m))
for i in range(2, k + 2):
    if m % i == 0:
        break                                   # 可以整除，肯定不是素数，结束循环
if i == k + 1:
    print(m, "是素数！")
else:
    print(m, "不是素数！")
```

8. 拓展训练

请思考问题：找出10个最小的连续自然数，它们个个都是合数（非素数）。

9. 补充知识点

聪明的你可能已经注意到了，在前面章节及本节中，我们都使用到了Python的math模块，下面我们就来介绍一下math模块的一些相关方法，在后续章节中也会用到。

math模块提供了对C标准定义的数学函数的访问，包括数论与表示函数、幂函数与对数函数、三角函数、角度转换函数、双曲函数、特殊函数及常数等。

下面介绍其中的部分常用函数，没有介绍到的函数，如果读者感兴趣，可以自行查阅相关的资料文档。

（1）数论与表示函数

• math.ceil(x)：该函数返回x的上限值，即大于或等于x的最小整数。

• math.fabs(x)：该函数返回x的绝对值。

• math.factorial(x)：该函数以一个整数返回x的阶乘，如果x不是整数或为负数时，则抛出ValueError错误异常。

• math.floor(x)：该函数返回x的向下取整，小于或等于x的最大整数。

• math.gcd(a,b)：该函数返回整数a和b的最大公约数。如果a或b之一非零，则gcd(a,b)的值是能同时整除a和b的最大正整数。gcd(0,0)返回0。

（2）幂函数与对数函数

• math.exp(x)：该函数返回e的x次幂，其中e=2.718281...，是自然对数的基数。这通常比math.e ** x或pow(math.e,x)更精确。

• math.expm1(x)：该函数返回e的x次幂，减1。这里e是自然对数的基数。

• math.log(x[,base])：该函数如果使用一个参数，返回x的自然对数（底为e）。如果使用两个参数，则返回给定base的对数x，计算为log(x)/log(base)。

• math.log2(x)：该函数返回x以2为底的对数。这通常比log(x,2)更准确。

• math.log10(x)：该函数返回x底为10的对数。这通常比log(x,10)更准确。

• math.pow(x,y)：该函数将返回x的y次幂。

• math.sqrt(x)：该函数返回x的平方根。

（3）三角函数

• math.sin(x)：该函数返回x弧度的正弦值。

• math.cos(x)：该函数返回x弧度的余弦值。

• math.tan(x)：该函数返回x弧度的正切值。

• math.asin(x)：该函数以弧度为单位，返回x的反正弦值。

• math.acos(x)：该函数以弧度为单位，返回x的反余弦值。

• math.atan(x)：该函数以弧度为单位，返回x的反正切值。

（4）角度转换函数

• math.degrees(x)：该函数将角度x从弧度转换为度数。

• math.radians(x)：该函数将角度x从度数转换为弧度。

（5）双曲函数

双曲函数是基于双曲线而非圆来对三角函数进行模拟。下面介绍常用的双曲函数。

• math.sinh(x)：该函数返回x的双曲正弦值。

• math.cosh(x)：该函数返回x的双曲余弦值。

• math.tanh(x)：该函数返回x的双曲正切值。

• math.asinh(x)：该函数返回x的反双曲正弦值。

• math.acosh(x)：该函数返回x的反双曲余弦值。

• math.atanh(x)：该函数返回x的反双曲正切值。

（6）常数

• math.pi：数学常数圆周率π=3.141592...，精确到可用精度。

• math.e：数学常数e=2.718281...，精确到可用精度。

• math.tau：数学常数τ=6.283185...，精确到可用精度。τ是一个圆周常数，等于2π，即圆的周长与半径之比。

5.2　哥德巴赫猜想

1. 问题描述

2000以内的不小于4的正偶数都能够分解为两个素数之和（即验证歌德巴赫猜想对2000以内的正偶数成立）。

2. 问题分析

根据问题描述，为了验证歌德巴赫猜想对2000以内的正偶数都是成立的，要将整数分解为两部分，然后判断分解出的两个整数是否均为素数。若是，则满足题意，否则应重新进行分解和判断。

针对该问题，我们可以给定如下的输入和输出限定。

输入时：每行输入一组数据，即2000以内的正偶数n，一直输入到文件结束符为止。

输出时：输出n能被分解成的素数a和b。如果不止一组解，则输出其中a最小的那组解。

当然，读者可以根据实际的需要规定不同的输入和输出形式。

输入示例：

```
4
6
8
10
12
```

输出示例：

```
2 2
3 3
3 5
3 7
5 7
```

3．算法设计

本问题我们可以采用函数来解决。

（1）fun(n)函数判断输入的n值是否为素数

定义一个函数，函数名设为fun，在其中判断传进来的形参——设为n（n≥2），是否为素数，如果是素数则返回1，否则返回0。在判断是否为素数时，可以采用5.1节中介绍的方法。需要注意的是，在所有偶数中，只有2是唯一的素数。因此，在函数fun()中，可以分为以下4种情况来判断：

- n=2，是素数，返回1。
- n是偶数，不是素数，返回0。
- n是奇数，不是素数，返回0。
- n≠2，是素数，返回1。

（2）guess(n)函数用于验证哥德巴赫猜想

由于我们已经对输出做了限定，即当输出结果时，如果有多组解，则输出a最小的那组解。显然，对每个读入的数据n，a必然小于或等于n//2，因此，定义循环变量i，使其从2～n/2进行循环，每次循环都做如下判断：fun(i) and fun(n-i)是否为1。

如果fun(i) and fun(n-i)=1，则表示fun(i)=1同时fun(n-i)=1。由fun()函数的定义可知，此时i和n-i都为素数，又由于i是从2～n/2按由小到大的顺序来迭代的，因此，（i,n-i）是我们求出的一组解，且该组解必然是所有可能解中a值最小的。

还需要注意的是，由于除了2以外的偶数不可能是素数，因此，i值的可能取值只能是2和所有的奇数。

4. 确定程序框架

（1）程序主框架

程序的主框架是一个while循环，每输入一个数据就处理一次，直到人为结束程序或输入非法数据而终止输入。代码如下：

```
while True:
# 循环输入
    n = int(input())
    guess(n)
# 调用函数验证哥德巴赫猜想
```

（2）使用函数判断n是否为素数

在算法设计中我们详细介绍了fun()函数，它的功能就是判断传进来的形参n是否为素数，其代码如下：

```
# 判断是否为素数
def fun(n):
    if n == 2:
        return 1                                                    #
n是2，返回1
    if n % 2 == 0:
        return 0
# n是偶数，不是素数，返回0

    i = 3
    while i <= math.sqrt(n):
        if n % i == 0:
            return 0                                                #
n是奇数，不是素数，返回0
        i += 2
    return 1
# n是除2以外的素数，返回1
```

（3）使用函数验证哥德巴赫猜想

在算法设计中，我们介绍了guess()函数，它的功能就是验证传入的参数n是否满足哥德巴赫猜想，其代码如下：

```
# 验证哥德巴赫猜想
def guess(n):
    ok = 0
# 进入循环前先置标志位
    i = 2
    while i <= (n // 2):
        if fun(i):
            if fun(n - i):
```

```
                print("%d  %d\n" % (i, n - i))       # i和n-i都是素数，则打印
                ok = 1
        if i != 2:
            i += 1
        if ok:
            break                                    # 已打印出所需要的输
出结果，跳出循环
        i += 1
```

程序的流程图如图5.3所示。

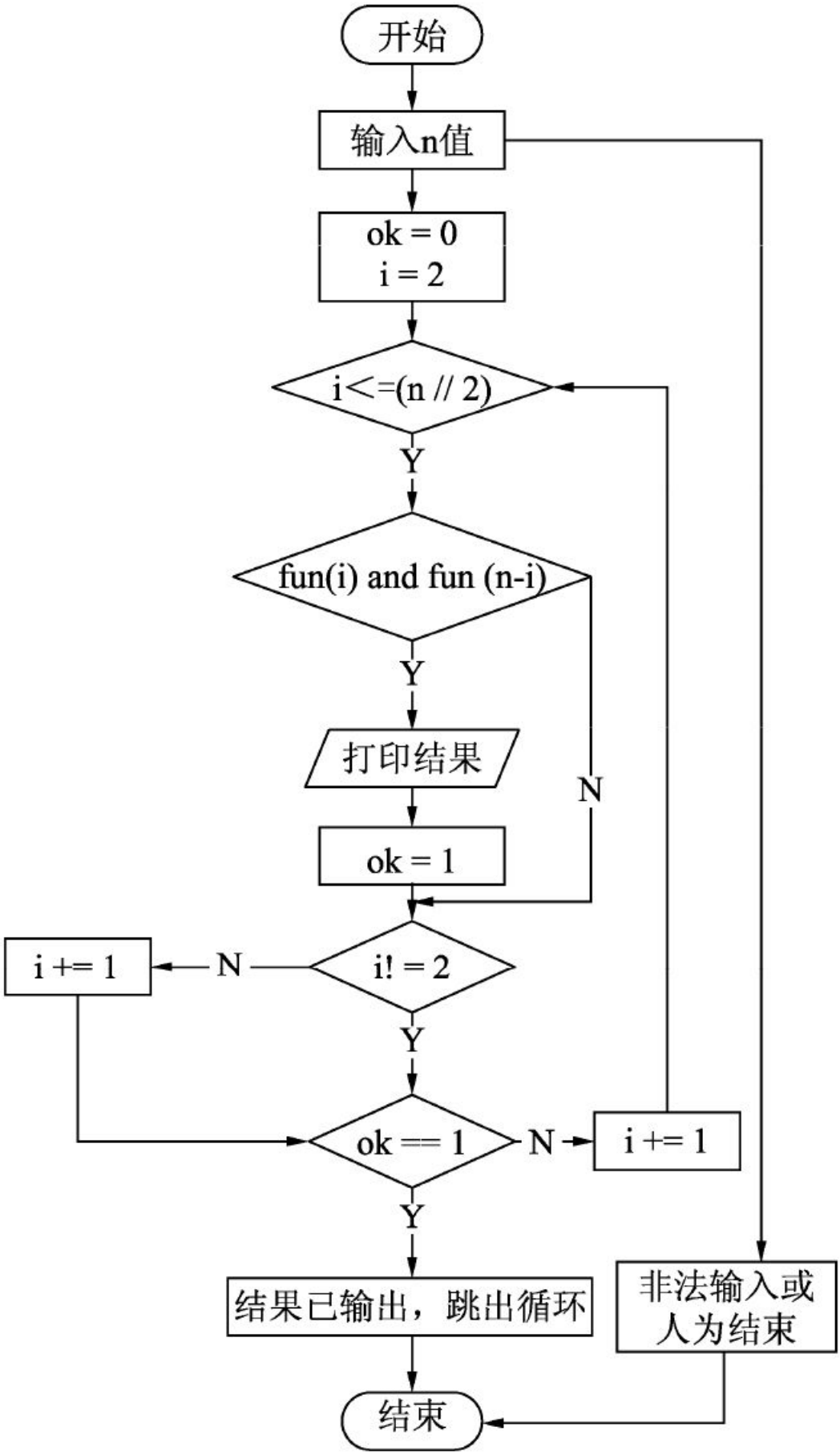

开始
输入n值
ok = 0
i = 2
i<=(n // 2)
Y
fun(i) and fun (n-i)
Y
N
打印结果
ok = 1
i += 1
N
i! = 2
Y
ok == 1
N
i += 1
Y
结果已输出，跳出循环
非法输入或
人为结束
结束

图5.3　程序流程图

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 哥德巴赫猜想

import math
# 判断是否为素数
def fun(n):
    if n == 2:
        return 1                                    # n是2，返回1
    if n % 2 == 0:
        return 0                                     # n是偶数，不是素数，返回0

    i = 3
    while i <= math.sqrt(n):
        if n % i == 0:
            return 0                                # n是奇数，不是素数，返回0
        i += 2
    return 1                                        # n是除2以外的素数，返回1

# 验证哥德巴赫猜想
def guess(n):
    ok = 0                                           # 进入循环前先置标志位
    i = 2
    while i <= (n // 2):
        if fun(i):
            if fun(n - i):
                print("%d  %d\n" % (i, n - i))      # i和n-i都是素数，则打印
                ok = 1
        if i != 2:
            i += 1
        if ok == 1:
            break                                   # 已打印出所需要的输出结果，跳出循
环
        i += 1

if __name__ == "__main__":
    while True:                                     # 循环输入
       n = int(input())
       guess(n)                                    # 调用函数验证哥德巴赫猜想
```

6. 运行结果

在PyCharm下运行程序，分别输入4、6、8、10、12，每输入一个数据后，按Enter键则立即打印出该数据的结果，如图5.4所示。

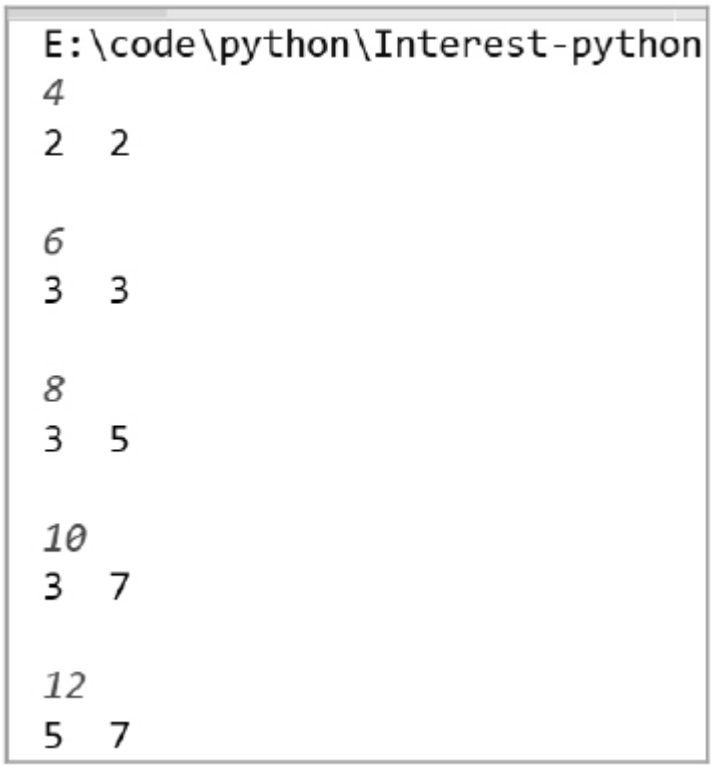


图5.4　运行结果

7．问题拓展

在该问题中我们定义了fun()函数来判断数n是否为素数，在fun()函数中，针对n的奇偶性进行了不同的处理，只要n是偶数，则肯定不是素数，这样就只需对n是奇数的情况进行判断，如下面的代码中加粗部分所示：

```
# 判断是否为素数
def fun(n):
    if n == 2:
        return 1                                    # n是2，返回1
    if n % 2 == 0:
        return 0                                    # n是偶数，不是素数，
返回0

    i = 3
    while i <= math.sqrt(n):
        if n % i == 0:
            return 0                                # n是奇数，不是素数，返回0
        i += 2
    return 1                                        # n是除2以外的素数，
返回1
```

如果n是奇数，则n包含的因子也只能为奇数，否则n就应该是偶数，因此循环变量i每次自增2，且i的变化范围为$3\sim\sqrt{n}$ 。使用这种方式来判断素数，与从 $2\sim\sqrt{n}$ 逐个比较相比，进一步缩小了比较范围，处理速度也进一步获得了提高。

5.3　要发就发

1．问题描述

“1898——要发就发”。请将不超过1993的所有素数从小到大排成第一行，第二行上的每个数都等于它上面相邻两个素数之差。编程求出：第二行数中是否存在若干个连续的整数，它们的和恰好为1898？假如存在的话，又有几种这样的情况？

两行数据分别如下：

第一行：2，3，5，7，11，13，17…1979，1987，1993

第二行：1，2，2，4，2，4…8，6

2．问题分析

首先从数学上对该问题进行分析。

假设第一行中的素数为n[1]、n[2]、n[3]…n[i]、…；第二行中的差值为m[1]、m[2]、m[3]…m[j]…。则m[j]可以表示为：m[j]=n[j+1]-n[j]。

第二行中连续N个数的和sum可以表示为：

sum=m[k]+m[k+1]+m[k+2]+m[k+3]+…+m[j]

=(n[k+1]-n[k])+(n[k+2]-n[k+1])+(n[k+3]-n[k+2])+…+(n[j+1]-n[j])

=n[k+1]-n[k]+n[k+2]-n[k+1]+n[k+3]-n[k+2]+…+n[j+1]-n[j]

=n[j+1]-n[k]

其中j>k≥1。

因此，原题目可以转换成如下的等价问题：在不超过1993的所有素数中是否存在这样两个素数，它们的差恰好是1898。若存在，则第二行中必有所需整数序列且其和恰为1898。

显然，对原问题的等价问题的求解是比较简单的。

由上面分析可知，在求解等价问题时，第一行的素数序列可以从3开始考虑，因为任意的素数与2的差一定为奇数，不可能为1898，所以在求解时素数序列中不需要包含2。

3．算法设计

首先采用数组number[NUM]来存放第一行中的全部素数，由前面分析可知，从3开始存放即可，一直到1993。

在产生不超过1993的素数序列时可以采用类似于5.2节中的函数来判断一个整数是否为素数。如果是素数，则将其存放到数组中，否则不需要存放。

定义一个函数，函数名设为fun()，在其中判断传进来的形参是否为素数，如果是素数则返回1，否则返回0。需要注意的是，在所有偶数中，只有2是唯一的素数。在函数fun()中，可以分为以下5种情况来判断。

- n≤1，由题意可知，本题不考虑素数为1的情况，且n<1时显然不是素数，故返回0。
- n=2，是素数，返回1。
- n是偶数，不是素数，返回0。
- n是奇数，不是素数，返回0。
- n≠2，是素数，返回1。

在主函数中，使用循环结构。

从第一行中最大的素数开始搜索，用它逐个减去number[0]、number[1]…每减一次判断一次，看它们的差值是否大于1898。直到在数组number[NUM]中找到一个位置j，使得最大素数与number[j]的差值不大于1898。此时，需要判断它们的差值是否等于1898，如果恰好等于1898，则最大素数与number[j]中存放的素数就是我们找到的第一个结果集。

对第一行中次大的素数重复该搜索过程，以此类推，直到第一行中大于1898且与1898最接近的那个素数为止。

4. 确定程序框架

程序的流程图如图5.5所示。

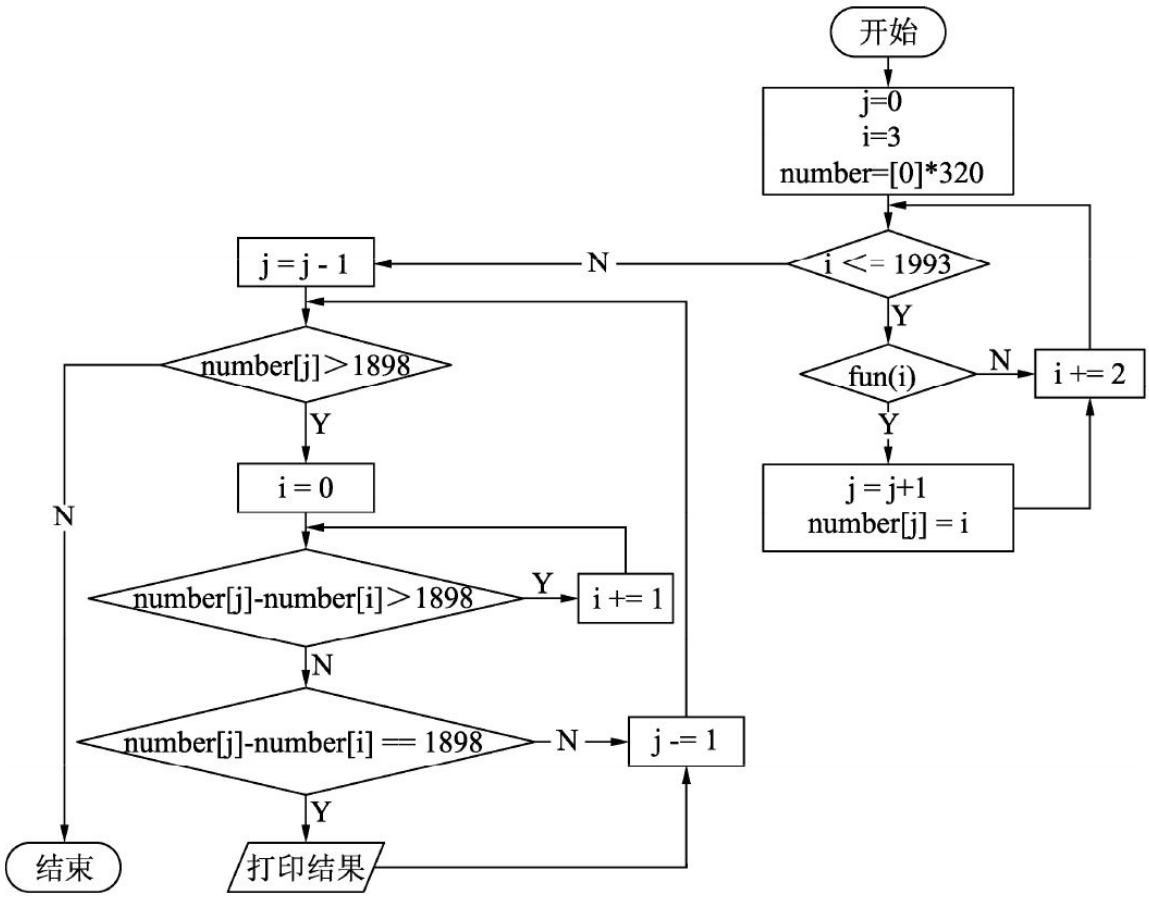


图5.5　程序流程图

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 要发就发

import math

# 判断素数
def fun(i):
    if i <= 1:
        return 0;                                                    # n<1时显然
不是素数，故返回0
    if i == 2:
        return 1                                                     # n=2，是素
数，返回1
    if i % 2 == 0:
        return 0                                                     # n是偶数，
不是素数，返回0
    for j in range(3, int(math.sqrt(i) + 1)):
        if i % j == 0:
            return 0                                                 # n是奇数，
不是素数，返回0
        j += 2
    return 1                                                                 # n
是除2以外的素数，返回1


if __name__ == "__main__":
    count = 0                                                                #
计数器
    print("列出第一行中差值为1898的所有素数组合： ")
    j = 0
    i = 3
    number = [0]*320
    while i <= 1993:                                                 # 求出不超过
1993的全部素数
        if fun(i):
            j = j+1
            number[j] = i
        i += 2
    j = j - 1
    while number[j] > 1898:                                  # 从最大的素数开始向1898
搜索
        # 循环查找满足条件的素数
        i = 0
        while number[j]-number[i] > 1898:
            i += 1
        if number[j] - number[i] == 1898:    # 若两个素数的差为1898，则输出
            count += 1
            print("(%d). %3d,  %d" %(count,
number[i], number[j]))
        j -= 1
```

6. 运行结果

在PyCharm下运行程序，结果如图5.6所示。


E:\code\python\Interest-python\venv\
列出第一行中差值为1898的所有素数组合:
(1). 89, 1987
(2). 53, 1951
(3). 3, 1901
Process finished with exit code 0

图5.6　运行结果

5.4　可逆素数

1．问题描述

请从小到大输出所有4位数的可逆素数。

可逆素数是指一个素数将其各位数字的顺序倒过来构成的反序数也是素数。

2．问题分析

通过前面几节的分析，相信读者对求素数的方法已经很熟悉了。本题要求的是可逆素数，根据问题描述可知，题目的难点已经不在于判断一个数是否为素数，而在于如何求一个整数的反序数。

求一个整数的反序数可按照如下的操作步骤进行。

（1）从该整数的最后一位开始，依次向前截取当前整数的最后一位数字。每截取一次，整数的位数减少一位。

（2）每次都将新截取到的数字作为其反序数的最后一位（个位），然后与上一次生成的反序数乘以10以后的数值相加。

（3）如此进行下去，原来整数的数字被从低位到高位不断地截取，依次形成其反序数中从高位到低位的各位数字。

下面以4位整数1234为例说明反序数的生成过程。

（1）截取个位4，原整数变为123，新生成的反序数为4。

1234→0*10+4=4

（2）截取当前整数的个位3，原整数变为12，用3与上一次生成的反序数4乘以10后的数值相加，则新生成的反序数为43。

123→4*10+3=43

（3）截取当前整数的个位2，原整数变为1，用2与上一次生成的反序数43乘以10后的数值相加，新生成的反序数为432。

12→43*10+2=432

（4）截取当前整数的个位1，用1与上一次生成的反序数432乘以10后的数值相加，新生成的反序数为4321。

1→432*10+1=4321

最后得到1234的反序数为4321。

3. 算法设计

解决该问题的算法可以使用穷举法，对所有的4位整数及其相应的反序数都依次进行判断。当该整数本身和其反序数都是素数时，该整数即为可逆素数，此时可以将该整数打印输出。

判断一个整数是否为素数的方法在前面已经多次用到，此处仍可以使用函数来完成素数的判断。将函数名设为fun，在其中判断传进来的形参是否为素数，如果是素数则返回1，否则返回0。该函数已经多次使用过，故此处不再详述，有问题的读者可参考5.2节及5.3节中对fun()函数的说明。

算法的核心是如何穷举所有的4位整数及其对应的反序数。由于是4位整数，因此算法设计时可以使用四重嵌套循环，每重循环对应一个数位。当所有循环执行完毕后，也就完成了对所有4位整数的迭代。

显然，第1重循环对应的是4位整数的千位，同时也是其反序数的个位；第2重循环对应的是4位整数的百位，同时也是其反序数的十位；第3重循环对应的是4位整数的十位，同时也是其反序数的百位；第4重循环（也是最内层循环）对应的是4位整数的个位，同时也是其反序数的千位。这样，每次在最内层循环的循环体内都可以

得到用4个循环变量组成的一个4位整数及其对应的反序数，我们只需判断这两个整数是否为素数即可，如果它们都为素数，则输出原4位整数，否则不输出。

程序无输入，输出时每行输出10个可逆素数。

4. 确定程序框架

程序的主框架为四重循环，具体如下：

```
for a in range(1, 9+1):                                          # 千位
    for b in range(0, 9+1):                                  # 百位
        for c in range(0, 9+1):                          # 十位
            for d in range(1, 9+1):                          # 个位
                # 判断4位整数是否为素数
                # 判断4位整数的反序数是否为素数
                # 如果该4位整数及其反序数都为素数，则打印该4位整数
```

程序的流程图如图5.7所示。

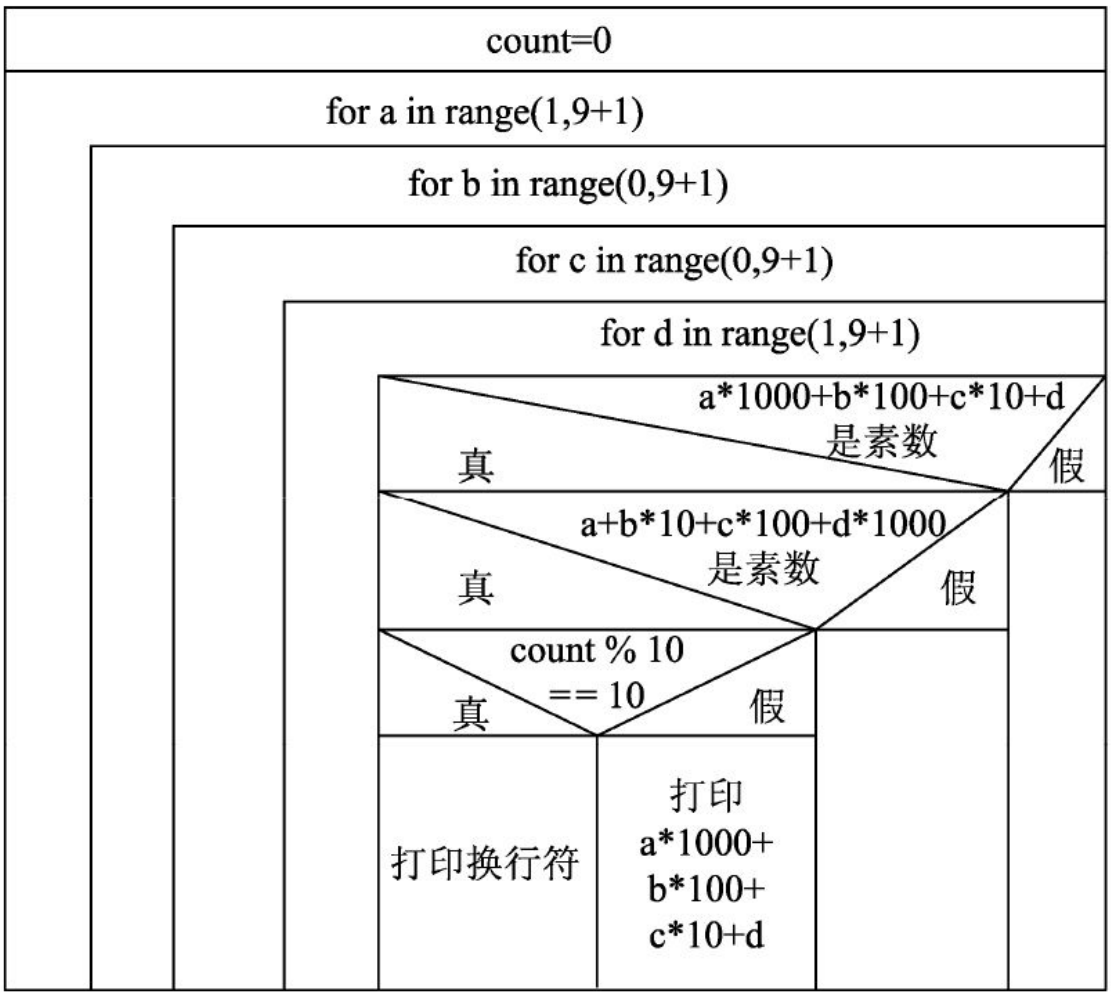


图5.7　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 可逆素数

import math

# 判断素数
def fun(n):
    if n <= 1:
        return 0;                                                    #
n<1时显然不是素数，故返回0
    if n == 2:
        return 1                                                     #
n=2，是素数，返回1
    if n % 2 == 0:
        return 0                                                     # n
是偶数，不是素数，返回0
    for i in range(3, int(math.sqrt(n) + 1)):
```

```
        if n % i == 0:
            return 0                                                    # n
是奇数，不是素数，返回0
        i += 2
    return 1
# n是除2以外的素数，返回1

if __name__=="__main__":
    count = 0
# 计数器
    # 穷举法
    for a in range(1, 9+1):                                             # 千位
        for b in range(0, 9+1):                                         # 百位
            for c in range(0, 9+1):                                     # 十位
                for d in range(1, 9+1):                                 # 个位
                    if fun(a*1000+b*100+c*10+d):    # 判断4位整数是否为素数
                        # 判断4位逆序的整数是否为素数
                        if fun(a+b*10+c*100+d*1000):
                            if count % 10 == 0:     # 每10个数就换行
                                print()
                            print("%d   " %(a*1000+b*100+c*10+d), end="")
                            count += 1
    print("\n4位可逆素数共有%d个" %count)
```

对程序的说明如下：

• 在上面的程序中使用了四重循环，有4个循环变量a、b、c、d。由于要求是4位整数，因此a从1开始取值，其取值范围为[1,9]。又因为个位为0的4位数肯定不是素数，因此d也从1开始取值，其取值范围为[1,9]。

• 使用a、b、c、d 4个循环变量可以将4位整数表示为a*1000+b*100+c*10+d，其反序数为a+b*10+c*100+d*1000。

• 程序中使用count变量控制每行输出10个整数。

6. 运行结果

在PyCharm下运行程序，结果如图5.8所示。

7. 拓展训练

求1000以内的孪生素数。孪生素数是指：若a为素数，且a+2也是素数，则素数a和a+2称为孪生素数。

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/code/python/

1009   1021   1031   1033   1061   1069   1091   1097   1103   1109
1151   1153   1181   1193   1201   1213   1217   1223   1229   1231
1237   1249   1259   1279   1283   1301   1321   1381   1399   1409
1429   1439   1453   1471   1487   1499   1511   1523   1559   1583
1597   1601   1619   1657   1669   1723   1733   1741   1753   1789
1811   1831   1847   1867   1879   1901   1913   1933   1949   1979
3011   3019   3023   3049   3067   3083   3089   3109   3121   3163
3169   3191   3203   3221   3251   3257   3271   3299   3301   3319
3343   3347   3359   3371   3373   3389   3391   3407   3433   3463
3467   3469   3511   3527   3541   3571   3583   3613   3643   3697
3719   3733   3767   3803   3821   3851   3853   3889   3911   3917
3929   7027   7043   7057   7121   7177   7187   7193   7207   7219
7229   7253   7297   7321   7349   7433   7457   7459   7481   7507
7523   7529   7547   7561   7577   7589   7603   7643   7649   7673
7681   7687   7699   7717   7757   7817   7841   7867   7879   7901
7927   7949   7951   7963   9001   9011   9013   9029   9041   9103
9127   9133   9161   9173   9209   9221   9227   9241   9257   9293
9341   9349   9403   9421   9437   9439   9467   9479   9491   9497
9521   9533   9547   9551   9601   9613   9643   9661   9679   9721
9749   9769   9781   9787   9791   9803   9833   9857   9871   9883
9923   9931   9941   9967
4位可逆素数共有204个

Process finished with exit code 0
```

图5.8　运行结果

5.5 回文素数

1. 问题描述

本节要研究回文素数的问题，先来看看什么是回文素数。

所谓回文素数指的是，对一个整数n从左向右和从右向左读其数值都相同且n为素数，则称整数n为回文素数。

对于偶数位的整数，除了11以外，都不存在回文素数。即所有的4位整数、6位整数、8位整数等都不存在回文素数。下面列出两位和三位整数中包含的所有回文素数。

两位回文素数：11。

三位回文素数：101，131，151，181，191，313，353，373，383，727，757，787，797，919，929。

本节要求解的问题是：求出所有不超过1000的回文素数。

2. 问题分析

本题仍旧要使用判断素数的方法。判断素数的方法读者已经比较熟悉了，因此本题实际要解决的问题在于如何求一个整数的回文数。

我们采用的方法是穷举法。对1000以内的每一个整数n进行考察，判断n是否为回文数。如果n是回文数，再判断它是否为素数，对于既是回文数也是素数的整数n就是我们要求的回文素数，将其打印输出即可。

由于题目要求解的是所有不超过1000的回文素数，因此最后的结果中应该包含两位和三位的回文数。

我们采用穷举法来构造一个整数并求与其对应的反序数，若整数与其反序数相等，则该整数是回文数。

3．算法设计

在问题分析中我们已经确定要采用穷举法逐一考查1000以内的每个整数，因此本题的算法设计可以采用循环结构来完成。

通过三重循环来遍历所有1000以内的整数。用3个循环变量来构造整数n，同时，这3个循环变量反序便可以构造出n的反序数m。其中，特别要注意的是如果n的个位为0，接下来要做的就是比较m和n的值是否相等，如果相等则表明整数m是回文数。再来判断n是否是素数，如果n也同时为素数，则n为回文素数，将其打印出来即可。

4．确定程序框架

程序的流程图如图5.9所示。

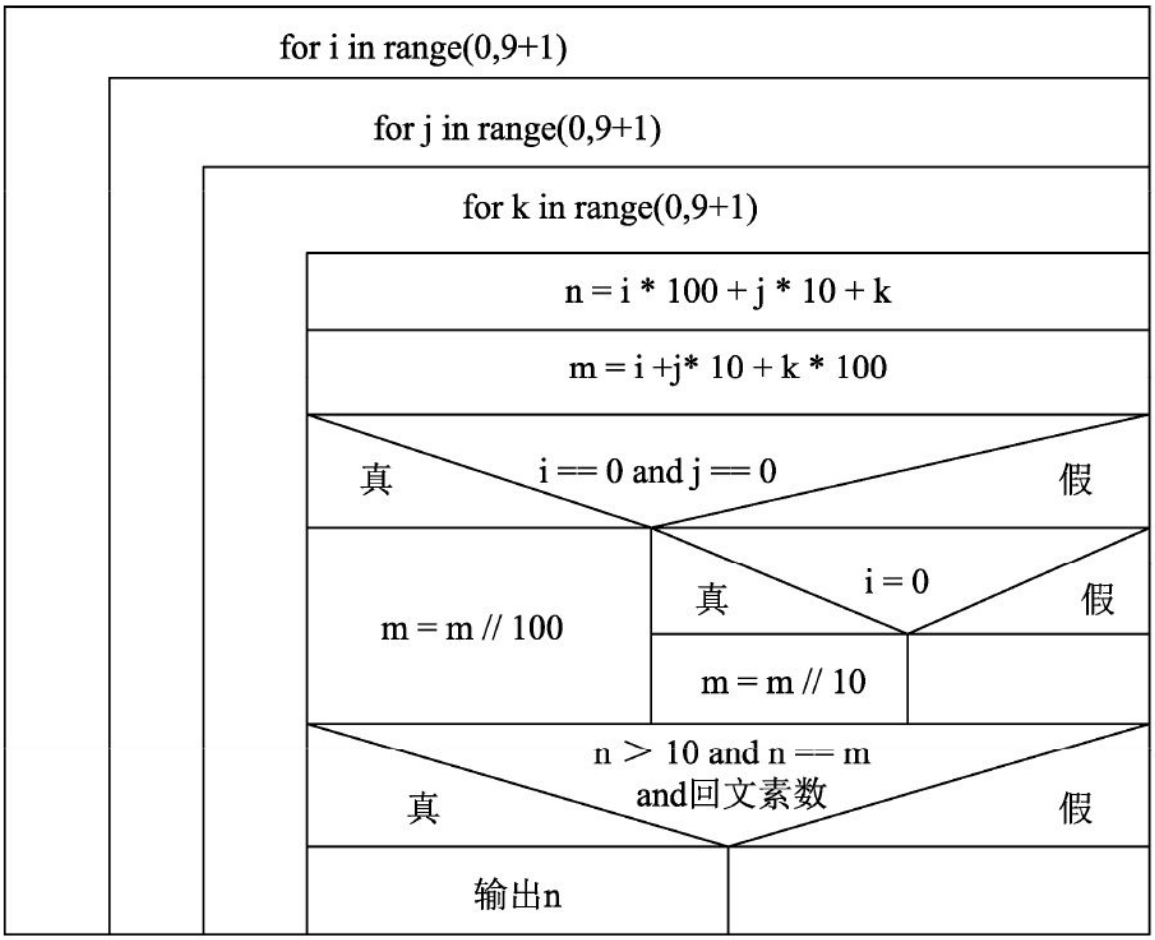


图5.9　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 回文素数

# 判断参数n是否为素数
def fun(n):
    for i in range(2, (n-1)//2):
        if n % i == 0:                                                    #
偶数不是素数
            return 0
    return 1
# 返回1是素数


if __name__=="__main__":
    print("1000以内的回文素数：")
    for i in range(0, 9+1):                                                # 穷举第一位
        for j in  range(0, 9+1):                                          # 穷举第二位
            for k in range(0, 9+1):                                       # 穷举第三位
                n = i * 100 + j * 10 + k            # 计算组成的整数
                m = i + j * 10 + k * 100                                  # 计算对应的反序数
                if i == 0 and j == 0:                                     # 处理整数的前两位为0
的情况
                    m = m // 100
                elif i == 0:                                              #
处理整数的第一位为0的情况
                    m = m // 10
                if n > 10 and n == m and fun(n): # 若大于10且为回文数则输出
                    print("%d\t" %n, end="")
```

6．运行结果

在PyCharm下运行程序，结果如图5.10所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/code/python
1000以内的回文素数：
11  101 131 151 181 191 313 353 373 383 727 757 787 797 919 929
Process finished with exit code 0
```

图5.10　运行结果

7．拓展训练

考虑是否有其他方法可以判断一个整数是否为回文素数。

5.6　孪生素数

1．问题描述

本节要研究孪生素数的问题，先来看看什么是孪生素数。

所谓孪生素数指的是间隔为2的两个相邻素数，因为它们之间的距离已经近得不能再近了，如同孪生兄弟一样，故将这一对素数称为孪生素数。

显然，最小的一对孪生素数是(1,3)。我们可以写出3～100以内的孪生素数，一共有8对，分别是(3,5)，(5,7)，(11,13)，(17,19)，(29,31)，(41,43)，(59,61)和(71,73)。随着数字的增大，孪生素数的分布也越来越稀疏，人工寻找孪生素数变得非常困难。

关于孪生素数还存在着一个著名的猜想——孪生素数猜想，即孪生素数是否有无穷多对，这是数论中还有待解决的一个重要问题。此处我们只讨论在有限范围内的孪生素数求解问题。

本节要解决的问题：编程求出3～1000以内的所有孪生素数。

2．问题分析

只要明白了什么是孪生素数，该问题便很好理解了。下面我们给出孪生素数更准确的定义。

孪生素数是指若a为素数，且a+2也是素数，则素数a和a+2称为孪生素数。

要编程求解的问题是找出3～1000以内的所有孪生素数，因此很自然的可以使用穷举法对3～1000以内的每一个整数n进行考查，先判断n是否为素数，再判断n+2是否为素数，如果n和n+2同时为素数，则(n,n+2)就是一对孪生素数，将其打印输出即可。

读者根据输出结果便可获知3～1000以内的孪生素数共有多少对。

3．算法设计

在问题分析中我们已经确定要采用穷举法逐一考查3～1000以内的每个整数，因此在本题的算法设计中需要采用循环结构。

在判断是否为素数时可以定义一个函数prime，每次判断整数n是否为素数时都将n作为实参传递给函数prime()，在prime()中使用前面介绍过的判断素数的方法进行判断。如果n为素数，则prime()函数返回值为1，否则prime()函数返回值为0。

4．确定程序框架

程序的流程图如图5.11所示。

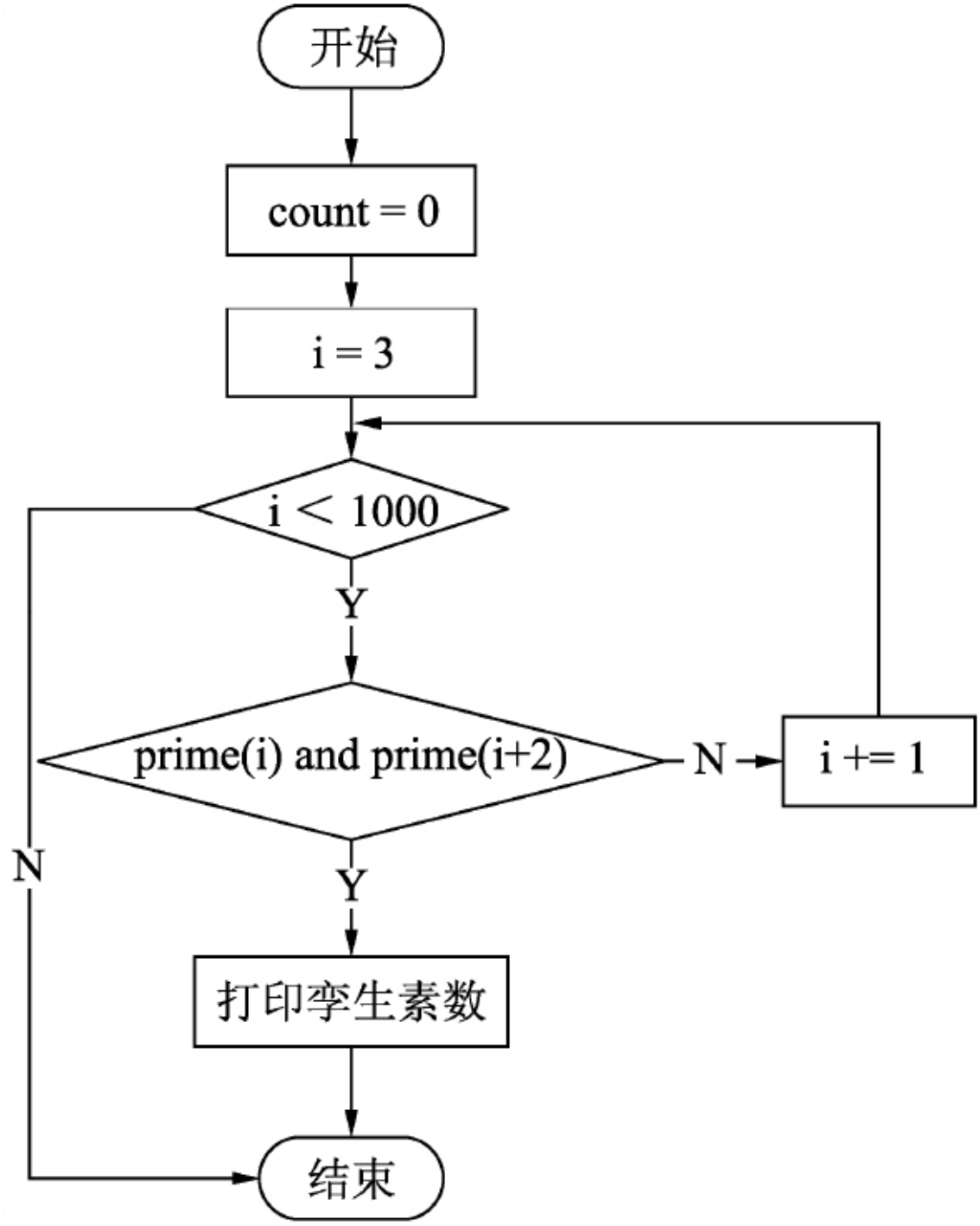


图5.11　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 孪生素数


import math

# 判断n是否为素数
def prime(n):
    k = math.sqrt(n) + 1
    i = 2
    while i <= k:
        if n % i == 0:
            return 0                                                    # n能被j整
除，不是素数，返回0
        i += 1
    return 1                                                                # n
是素数，返回1


if __name__=="__main__":
    count = 0                                                               #
计数器
    print("3到1000之间的孪生素数：")
    for i in range(3, 1000):
        if prime(i) and prime(i+2):
            print("(%-3d, %3d)  " %(i, i+2), end="")
            count += 1
            if count % 5 == 0:                                  # 每5个数换一行
                print()
    print("\n1000以内的孪生素数共有%d对" %count)
```

6. 运行结果

在PyCharm下运行程序，结果如图5.12所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/co
3到1000之间的孪生素数：
(3  ,   5)  (5  ,   7)  (11 ,  13)  (17 ,  19)  (29 ,  31)
(41 ,  43)  (59 ,  61)  (71 ,  73)  (101, 103)  (107, 109)
(137, 139)  (149, 151)  (179, 181)  (191, 193)  (197, 199)
(227, 229)  (239, 241)  (269, 271)  (281, 283)  (311, 313)
(347, 349)  (419, 421)  (431, 433)  (461, 463)  (521, 523)
(569, 571)  (599, 601)  (617, 619)  (641, 643)  (659, 661)
(809, 811)  (821, 823)  (827, 829)  (857, 859)  (881, 883)

1000以内的孪生素数共有35对

Process finished with exit code 0
```

图5.12　运行结果

观察图5.12可知，输出结果时每行都打印了5对孪生素数，一共打印了7行。因此，在3～1000范围内的孪生素数一共有35对。

5.7　梅森素数

1．问题描述

梅森数（Mersenne Prime）指的是形如2^n-1的正整数，其中指数n是素数，记为Mn。如果一个梅森数是素数，则称其为梅森素数。例如$2^2-1=3$、$2^3-1=7$都是梅森素数。

当n=2、3、5、7时，Mn都是素数，但n=11时，$Mn=M_{11}=2^{11}-1=2047=23\times89$，显然不是梅森素数。

1722年，瑞士数学大师欧拉证明了$2^{31}-1=2147483647$是一个素数，它共有10位数，成为当时世界上已知的最大素数。

迄今为止，人们仅发现了47个梅森素数。梅森素数历来都是数论研究中的一项重要内容，也是当今科学探索中的热点和难点问题。

了解了梅森素数后，现在来看本节要解决的编程问题。

试求出指数n<20的所有梅森素数。

2．问题分析

只要理解了梅森素数的定义，该问题并不难求解。

要编程求解的问题是找出指数n<20的所有梅森素数。根据梅森素数的定义，我们可以先求出n<20的所有梅森数，再逐一判断这些数是否为素数。如果是素数，则表示该数为梅森素数，打印输出即可，否则不是梅森素数。

3．算法设计

由问题分析可知，我们要求n<20的所有梅森数，因此在本题的算法设计中需要采用循环结构。

设变量mp存储梅森数，整数i表示指数，其取值从2～19，i每变化一次，都相应地计算出一个梅森数，存放在mp中。对每次计算得到的当前mp值，都调用函数prime()进行判断。

在判断mp是否为素数时可以定义一个函数prime()，每次都将mp的当前值作为实参传递给函数prime()，在prime()中使用前面介绍过的判断素数的方法进行判断。如果n为素数，则prime()函数返回值为1，否则prime()函数返回值为0。

若prime()函数返回值为1，则当前mp为梅森素数，应该将其输出；若prime()函数返回值为0，则当前mp不是梅森素数。

4. 确定程序框架

程序的流程图如图5.13所示。

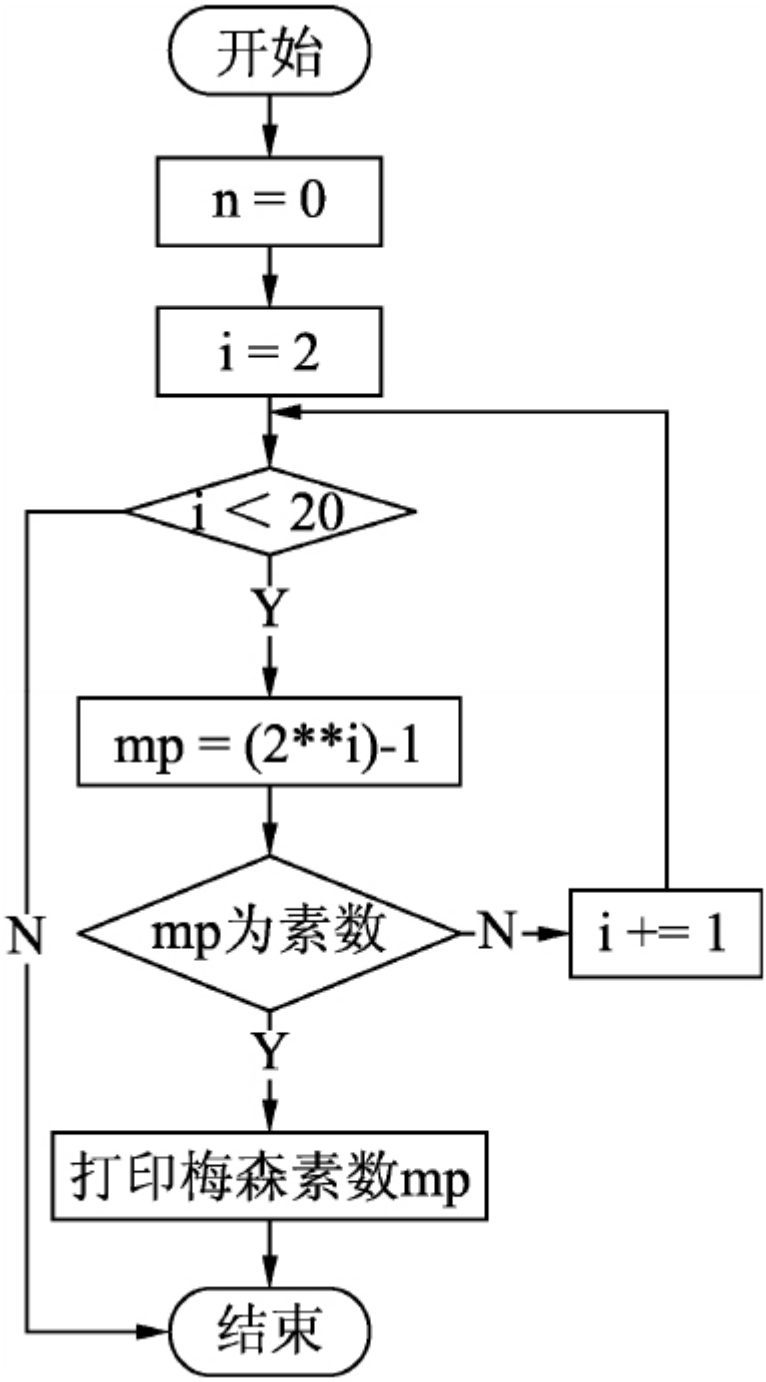


图5.13　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
```

```
# @desc: 梅森素数

import math

# 判断n是否为素数
def prime(n):
    k = math.sqrt(n) + 1
    i = 2
    while i <= k:
        if n % i == 0:
            return 0         # n能被j整除，不是素数，返回0
        i += 1
    return 1                 # n是素数，返回1

if __name__=="__main__":
    n = 0
    print("梅森素数：")
    for i in range(2, 20):
        mp = (2 ** i) - 1                                    # 求梅森数
        if prime(mp):                                                # 判断mp是否为梅森素
数
            n += 1
            print("M(%d) = %d" %(i, mp))     # 若当前mp为梅森素数，则打印，i为指数
    print("2的指数n<20的所有梅森素数有：%d个" %n)
```

6. 运行结果

在PyCharm下运行程序，结果如图5.14所示。由图5.14可见，M(2)=3，表示3是指数为2的梅森素数，其他的结果可类似解释。由运行结果可知，n<20的所有梅森素数共有7个。

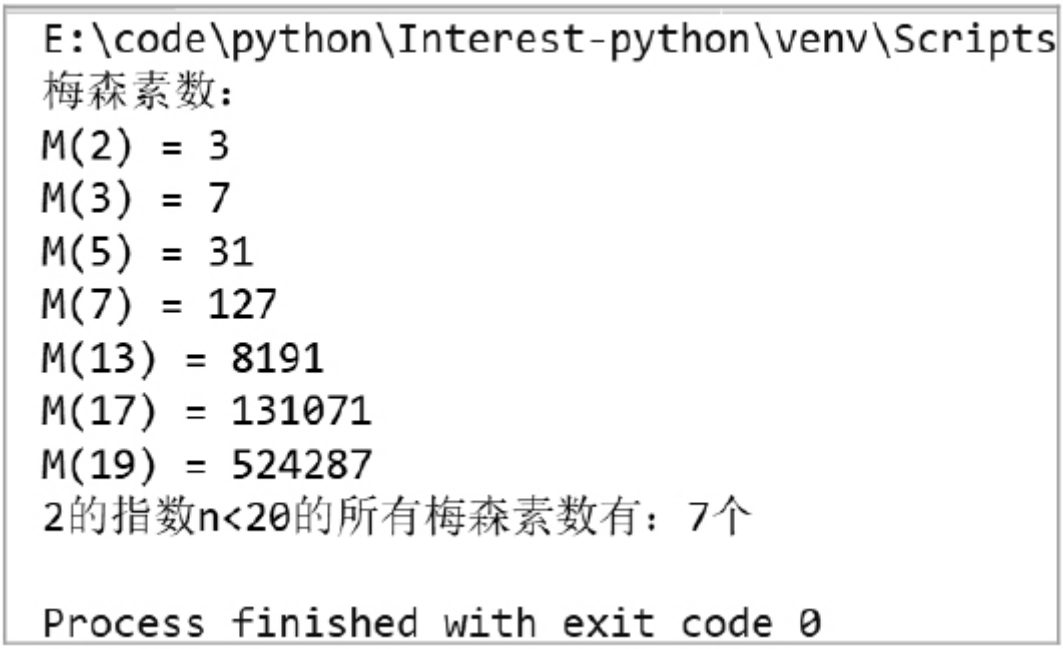


图5.14　运行结果

第6章　趣味逻辑推理

逻辑推理问题在Python程序设计中也是一类很常见的问题。要充分利用题目中给定的已知条件，通过合理的分析和判断，得出正确的结论。而使用计算机编程解决逻辑推理问题的关键是在理解题意的基础上写出正确的逻辑表达式。在Python中提供了丰富的算数和逻辑操作符，通过这些操作符可以将复杂的逻辑推理问题转化为简明的逻辑表达式来表达。将要求解问题中给出的限定条件用程序语言描述清楚后就可以使用穷举法得到最终的判断结果，这是解决逻辑推理问题的一种通用方法。

本章共有8个例子，每个例子都提供了一段有趣的小故事，读者可根据这些小故事自己进行分析判断，得到一个结论，再与使用Python编程实现的结果进行比较，从而更好地掌握逻辑推理问题的编程方法，提高解决问题的能力。

6.1　谁家孩子跑得最慢

1．问题描述

假设有张、王、李三家，每家都有3个孩子。某一天，这三家的9个孩子一起短跑比赛，规定不考虑年龄大小，第一名得9分，第二名得8分，第三名得7分，以此类推。比赛结束后统计分数发现三家孩子的总分是相同的，同时限定这9个孩子的名次不存在并列的情况，且同一家的孩子不会获得相连的名次。现已知获得第一名的是李家的孩子，获得第二名的是王家的孩子，要求编程求出获得最后一名的是哪家的孩子。

2．问题分析

根据问题描述可知：

1）参加比赛的一共有9个孩子，得分情况依次是从1～9分。由此可知，该场比赛总共的分数为9+8+7+6+5+4+3+2+1=45分。

2）由于“比赛结束后统计分数发现三家孩子的总分是相同的”，因此每家孩子的得分都为15分。

3）由于“获得第一名的是李家的孩子，获得第二名的是王家的孩子”，因此可推知获得第三名的一定是张家的孩子，否则李家（或王家）的孩子总分就会超过15分。

4）由于“这9个孩子的名次不存在并列的情况，且同一家的孩子不会获得相连的名次”，因此可推知获得第4名的一定不是张家的孩子。

3．算法设计

将问题分析中的文字进一步具体化。

先确定使用什么结构来存储9个孩子的分数。这里考虑使用列表，设列表名为score，且score[1]中存放张家3个孩子的分数，score[2]中存放王家3个孩子的分数，score[3]中存放李家3个孩子的分数。因为“同一家的孩子不会获得相连的名次”，则张家3个孩子的分数分别按从大到小的顺序依次存放在列表元素score[1][1]、score[1][2]和score[1][3]中，王家3个孩子的分数分别按从大到小的顺序依次存放在列表元素score[2][1]、score[2][2]和score[2][3]中，李家3个孩子的分数分别按从大到小的顺序依次存放在列表元素score[3][1]、score[3][2]和score[3][3]中。

因此，由问题分析中的第（2）点可知：

```
score[1][1]+score[1][2]+score[1][3]=15
score[2][1]+score[2][2]+score[2][3]=15
score[3][1]+score[3][2]+score[3][3]=15
①
```

由问题分析中的第（3）点可知：

```
score[3][1]=9，score[2][1]=8，score[1][1]=7
②
```

由问题分析中的第（4）点可知：

```
score[1][2]≠6
```

由于“9个孩子的名次不存在并列的情况，且同一家的孩子不会获得相连的名次”，因此score列表中存放的9个值应该各不相同，分数为1～9。由①、②可推知：

```
score[1][2]+score[1][3] ≠score[2][2]+score[2][3] ≠score[3][2]+score[3][3]
```

因为9个孩子的名次不会并列，且每家孩子的分数是按从大到小的顺序依次存放在列表中的，因此：

```
score[1][2] ≠score[2][2] ≠score[3][2]
score[1][3] ≠score[2][3] ≠score[3][3]
③
```

且score[1][2]、score[2][2]、score[3][2]应该占据了分数段的4～6分这几个分值，score[1][3]、score[2][3]、score[3][3]则占据了分数段的1～3分这几个分值。

由上面的分析可知，由于已经明确了score[1][2]、score[2][2]和score[3][2]这3个列表元素中存放的值所占据的分数段为4～6分，则可以使用循环结构来判断出3个元素中存放的值分别是几。判断的依据是在循环体中检测，保证③的成立。

4. 确定程序框架

程序的流程图如图6.1所示。

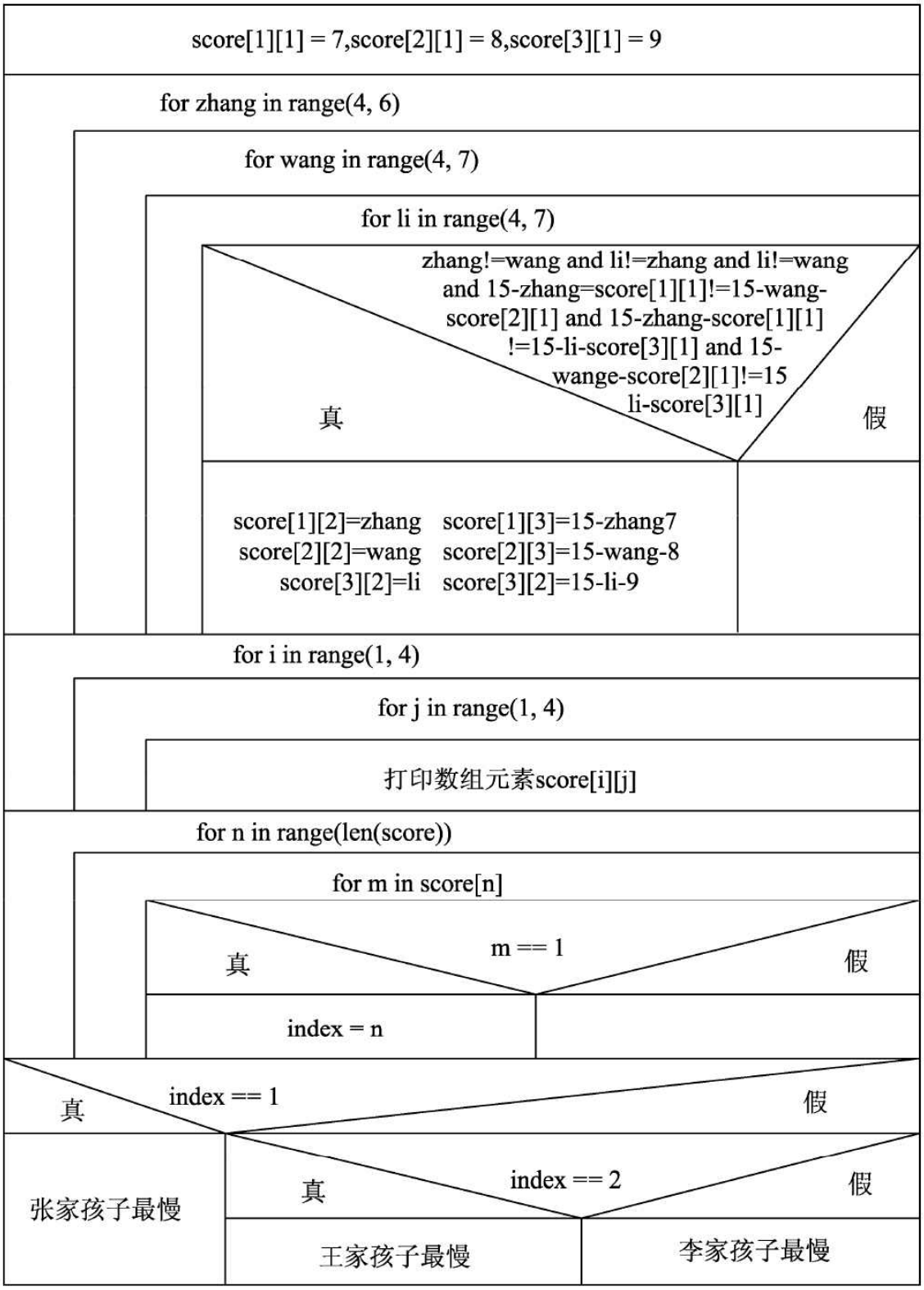


图6.1　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-
#author: liuhefei
#desc: 谁家孩子跑得最慢

if __name__ == "__main__":
    score = [([0] * 4) for i in range(4)]   #定义列表，用来存储各个孩子的分数
    #score[1]、score[2]和score[3]分别存放张家、王家和李家3个孩子的分数
    #获得第一名的是李家的孩子，获得第二名的是王家的孩子

    #初始化三家孩子的最高分数值
    score[1][1] = 7                                                    #score[1]存
放张家3个孩子的分数
    score[2][1] = 8                                                    #score[2]存
放王家3个孩子的分数
    score[3][1] = 9                                                    #score[3]存
放李家3个孩子的分数

    #通过循环得到9个孩子的分数
    for zhang in range(4, 6):
        for wang in range(4, 7):
            for li in range(4, 7):
                #9个孩子名次不存在并列的情况
                if zhang != wang and li != zhang and li != wang \
                        and 15-zhang-score[1][1] != 15-wang-score[2][1] \
                        and 15-zhang-score[1][1] != 15-li-score[3][1] \
                        and 15-wang-score[2][1] != 15-li-score[3][1]:
                    #将结果存入对应的数组元素
                    score[1][2] = zhang
                    score[1][3] = 15-zhang-7
                    score[2][2] = wang
                    score[2][3] = 15-wang-8
                    score[3][2] = li
                    score[3][3] = 15-li-9

    #打印二维数组score，输出各家孩子的分数
    print("array score:")
    for i in range(1, 4):
        for j in range(1, 4):
            print(score[i][j], end=' ')
        print()

    #for循环每一个孩子的分数值，并将最后一名孩子所来自的家庭保存到index中
    for n in range(len(score)):
        #print(n, score[n])
        for m in score[n]:
            if m == 1:
                index = n                                              #记录最后一名孩子所来
自的家庭
    #print(index)

    #输出最后一名孩子来自的家庭
    if index == 1:
        print("The last one reached the end is a child from family Zhang.\n")
    elif index == 2:
        print("The last one reached the end is a child from family Wang.\n")
    else:
        print("The last one reached the end is a child from family Li.\n")
```

在上面的程序中需要注意的是，循环变量zhang可能的取值为4和5，不能为6，这是由于同一家的孩子不会获得连续的名次。

6. 运行结果

在PyCharm下运行程序，结果如图6.2所示。

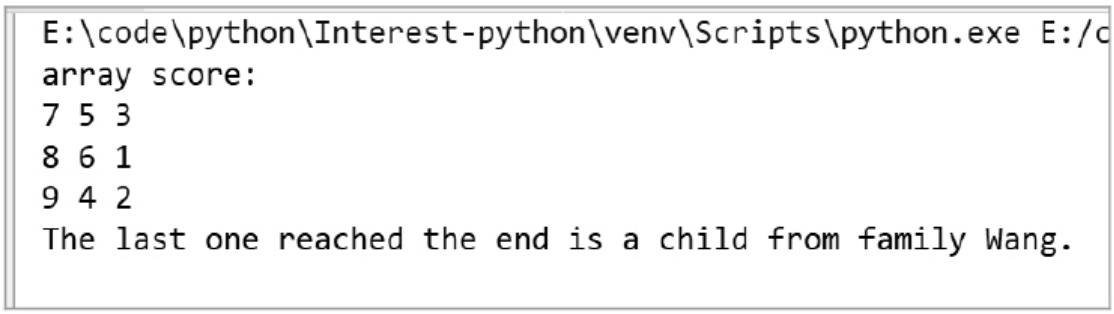


图6.2　运行结果

由运行结果可以看到，我们使用矩阵的形式将三家孩子的分数打印了出来。张家3个孩子的分数由大到小分别为7分、5分和3分，王家3个孩子的分数由大到小分别为8分、6分和1分，而李家3个孩子的分数由大到小分别为9分、4分和2分。获得最后一名的孩子也就是跑得最慢的孩子是王家的孩子。

6.2　新郎和新娘

1. 问题描述

有三对情侣结婚，假设三个新郎为A、B、C，三个新娘为X、Y、Z。有参加婚礼的人搞不清谁和谁结婚，所以去询问了这六位新人中的三位，得到的回答为：新郎A说他要和新娘X结婚；新娘X说她的未婚夫是新郎C；而新郎C说他要和新娘Z结婚。听到这样的回答后，提问者知道他们都是在开玩笑，说的都是假话，但他仍搞不清谁和谁结婚。现在请编程求出到底哪位新郎和哪位新娘结婚。

2. 问题分析

根据问题描述，提问者得到的回答都是假话，因此新郎A的新娘不是X，X的新郎不是C，C的新娘不是Z。

显然，一个新郎只能和一个新娘结婚，他们之间是一对一的关系。

3. 算法设计

该问题我们可以采用穷举法来解决。

设三个变量x、y和z分别表示与新娘X结婚的新郎、与新娘Y结婚的新郎和与新娘Z结婚的新郎，它们可能取值的集合是{'A','B','C'}。定义一个groom列表来存放三个新郎，如下：

```
groom = ['A','B','C']
```

其中，groom[0]中存放A，groom[1]中存放B，groom[2]中存放C。

根据问题分析中获得的结论，可以得出判断依据如下：

- x!='A'——新郎A的新娘不是X。
- x!='C'——与新娘X结婚的新郎不是C。
- z!='C'——C的新娘不是Z。
- x!=y——与新娘X结婚的新郎不会与新娘Y结婚。
- x!=z——与新娘X结婚的新郎不会与新娘Z结婚。
- y!=z——与新娘Y结婚的新郎不会与新娘Z结婚。

将上面的判断依据组合起来，找到变量x、y和z所满足的条件，用Python表示如下：

```
x != groom[0] and x != groom[2] and z != groom[2] and x != y and x != z and y
!= z
```

找到上面的条件以后，我们就可以在程序中使用三重循环来穷举x、y和z的所有取值，找出满足上述条件的x值、y值和z值，这样便能解决到底哪位新郎和哪位新娘结婚的问题。

4. 确定程序框架

程序的流程图如图6.3所示。

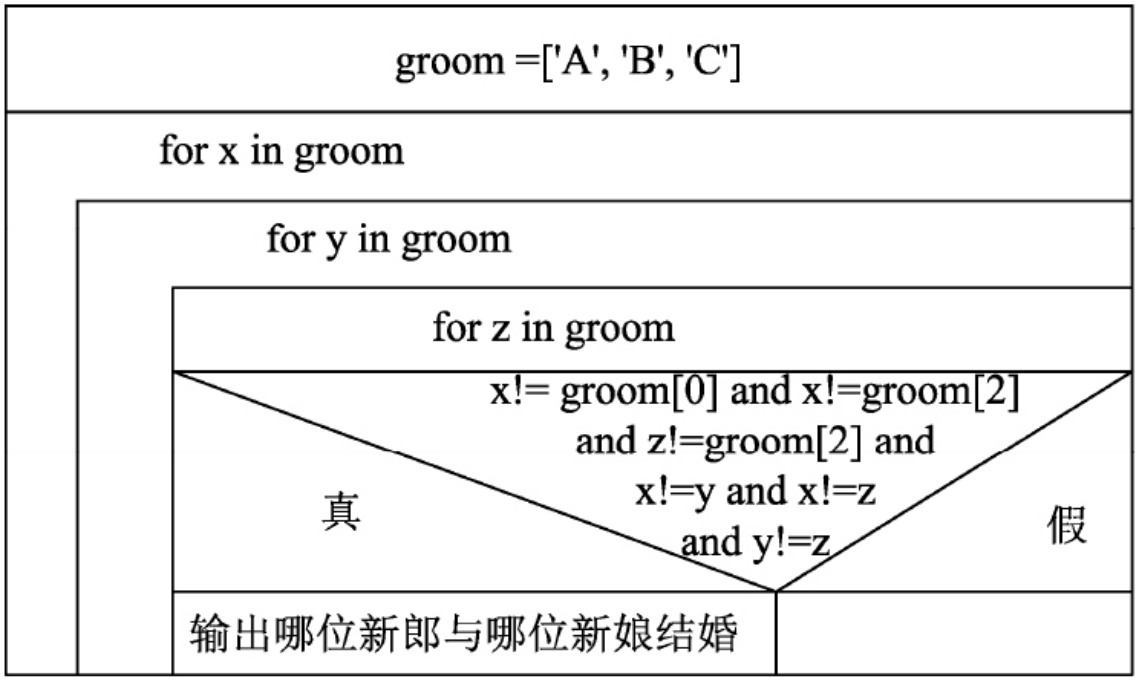


图6.3　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-
#author: liuhefei
#desc: 新郎和新娘

if __name__ == "__main__":
    #三个新郎为A、B、C，三个新娘为X、Y、Z
    groom = ['A', 'B', 'C']                                    #定义新郎列表

    #穷举所有可能情况
    for x in groom:
        for y in groom:
            for z in groom:
                # 新郎A的新娘不是X，那就只能是Y和Z；
                # X的新郎不是C，那就只能是A和B；
                # C的新娘不是Z，那就只能是X和Y；
                # 由此推测出：新郎C的新娘是Y，那么新郎A的新娘是Z，进一步推断新郎B
                  的新娘是X
                if x != groom[0] and x != groom[2] and z != groom[2] and x !=
y and x != z and y != z:
                    print("结果为：")
                    print("新娘X与新郎" + x + "结婚\n");
                    print("新娘Y与新郎" + y + "结婚\n");
                    print("新娘Z与新郎" + z + "结婚\n");
```

6．运行结果

在PyCharm下运行程序，结果如图6.4所示。由图6.4可见，新娘X与新郎B结婚，新娘Y与新郎C结婚，新娘Z与新郎A结婚。

图6.4　运行结果

6.3 谁在说谎

1. 问题描述

现有张三、李四和王五三个人，张三说李四在说谎，李四说王五在说谎，而王五说张三和李四两人都在说谎。要求编程求出这三个人中到底谁说的是真话，谁说的是假话。

2. 问题分析

显然，该题是一个逻辑推断问题。张三、李四和王五三个人都可能说真话，也都可能说假话，那么如何来判断他们到底谁在说谎呢？

由问题描述可得到如下三个结论：

• 由于“张三说李四在说谎”，因此，如果张三说的是真话，则李四就在说谎；反之，如果张三在说谎，则李四说的就是真话。

• 由于“李四说王五在说谎”，因此，如果李四说的是真话，则王五就在说谎；反之，如果李四在说谎，则王五说的就是真话。

• 由于“王五说张三和李四两人都在说谎”，因此，如果王五说的是真话，则张三和李四两人就都在说谎；反之，如果王五在说谎，则张三和李四两人至少一人说的是真话。

3. 算法设计

该问题同样可用穷举法进行解决。

首先将问题分析中得到的三个分析结果用表达式表达出来。用变量x、y和z分别表示张三、李四和王五三人说话真假的情况，当x、y或z的值为1时表示该人说的是真话，值为0时表示该人说的是假话。则问题分析中的三个结论可以使用如下的表达式进行表示：

• x==1 and y==0——张三说的是真话，李四在说谎。

• x==0 and y==1——张三在说谎，李四说的是真话。

• y==1 and z==0——李四说的是真话，王五在说谎。

• y==0 and z==1——李四在说谎，王五说的是真话。

• z==1 and x==0 and y==0——王五说的是真话，则张三和李四两人就都在说谎。

• z==0 and x+y!=0——王五在说谎，则张三和李四两人至少一人说的是真话。

我们已经知道，在Python中，有了关系运算符和逻辑运算符以后，就可以使用一个逻辑表达式来表达一个复杂的关系。将上面的表达式进行整理，获得Python的表达式如下：

```
(x and (not y) or (not x) and y) and (y and (not z) or (not y) and z) and (z
and x ==0 and y==0 or (not z) and x+y != 0)
```

4. 确定程序框架

程序的流程图如图6.5所示。

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-
#author: liuhefei
#desc: 谁在说谎

if __name__ == "__main__":
    # x、y和z分别表示张三、李四和王五三人说话真假的情况
    # 当x、y或z的值为1时表示该人说的是真话，值为0时表示该人说的是假话
    # 使用三重循环穷举所有情况
    for x in range(2):
        for y in range(2):
            for z in range(2):
                if (x and (not y) or (not x) and y) and (y and (not z) or (not
y) and z)
                    and (z and x ==0 and y==0 or (not z) and x+y != 0):
```

```
a = '真' if x == 1 else '假'
b = '真' if y == 1 else '假'
c = '真' if z == 1 else '假'
print("张三说的是" + a + "话")
print("李四说的是" + b + "话")
print("王五说的是" + c + "话")
```

程序说明：

在输出结果的时候使用了条件表达式。它的应用背景是当使用if语句时，不论表达式为“真”还是“假”，都只执行一个赋值语句给同一个变量赋值，这种情况下便可以使用条件运算符来代替if语句进行处理，以简化程序的书写。

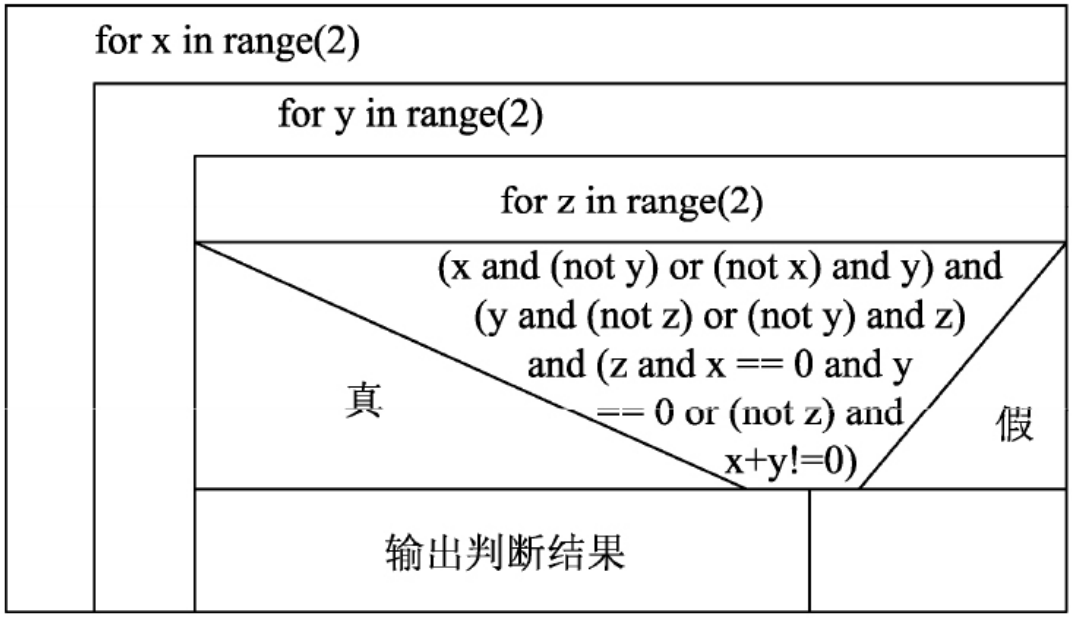


图6.5　程序流程图

例如，有如下的if语句：

```
if x>0:
  y=x
else:
  y=-x;
```

该if语句可以使用如下的条件运算符来表达：

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

它的执行过程为：如果x>0条件为真，则条件表达式的值为x，即y=x；如果条件为假，则条件表达式的值为-x，则y=-x。

6．运行结果

在PyCharm下运行程序，显示结果如图6.6所示。由图6.6可知，张三说的是假话，李四说的是真话，王五说的是假话。

图6.6　运行结果

7．问题拓展

在本题中，首先根据题意进行逻辑分析，得到“问题分析”中的三条结论；接着，将这些结论用Python中的表达式表达出来，作为我们程序设计中进行判断的依据。

对这类逻辑推理问题都可以类似地进行处理。由于该类问题中往往要将推理结论转换成逻辑表达式，因此这里对逻辑表达式的相关内容再做一归纳。

1）首先要明确的是逻辑表达式的值是逻辑量——“真”或“假”。对Python编译系统而言，当逻辑表达式运算结果为“真”时，则该表达式值为True，当逻辑表达式运算结果为“假”时，该表达式值为False。

2）在对逻辑表达式求解时，并不是所有的逻辑运算符都会被执行，只有当该逻辑运算符影响表达式的求解时才执行该运算符。

例如，True and False and True，对该逻辑表达式求解时，只需执行True and False即可确定整个表达式的结果为False，其中第

二个“and”运算符并没有被执行。

在逻辑表达式x or y or z中，只要x非0（为真），就不需要再判断y和z的值。只有当x为假时，才需要继续判断y的值。而仅当x和y都为假时，才需要判断z的值。

6.4　谁是窃贼

1. 问题描述

警察审问4名窃贼嫌疑犯。现在已知，这4人当中仅有一名是窃贼，还知道这4个人中的每个人要么是诚实的，要么总是说谎。

下面是这4个人给警察的回答。

- 甲说：“乙没有偷，是丁偷的。”
- 乙说：“我没有偷，是丙偷的。”
- 丙说：“甲没有偷，是乙偷的。”
- 丁说：“我没有偷。”

请根据这4个人的回答判断谁是窃贼。

2. 问题分析

显然该题是一个逻辑推断问题。已知4个人中的每个人要么是诚实的，要么总是说谎，那么如何来判断他们到底谁是窃贼呢？

由问题描述可知，甲、乙、丙、丁4人中仅有一名窃贼，且这4个人中每个人要么是诚实的，要么总是说谎。分析甲、乙、丙3人所说的话可以发现，他们每人都说了两句话，即“X没有偷，X偷的”，因此，不论这三人是否说谎，他们所提到的这两个人中必有一个是窃贼。而丁只说了他自己没有偷，故无法判断其真假。

假设用变量A、B、C、D分别代表4个人，变量的值为1代表该人是窃贼，则根据4个人的说法可列出下面的4个条件：

- 甲说：“乙没有偷，是丁偷的。”——B+D=1　①

• 乙说：“我没有偷，是丙偷的。”——B+C=1　②

• 丙说：“甲没有偷，是乙偷的。”——A+B=1　③

• 丁说：“我没有偷。”——A+B+C+D=1　④

由于甲、乙、丙3人的话中都提到了两个人，其中必有一人是小偷，故在根据他们的话列出条件表达式时，可以不关心谁说的是真话，谁说的是假话。

由于丁的话无法判断真假，故根据丁的话列出的表达式只反映了4人中仅有一名是窃贼的条件。

3. 算法设计

该问题的关键是使用Python中的逻辑表达式将问题分析中得到的4个条件表达出来，逻辑表达式如下：

B+D==1 and B+C==1 and A+B==1　⑤

条件④表示A、B、C、D中必有一个为1。

在程序中可依次假定甲、乙、丙、丁分别为窃贼，代入⑤进行测试，满足条件⑤的那个人为窃贼，具体如下：

1）先假定甲为窃贼，即A=1、B=0、C=0、D=0，代入条件⑤测试是否成立，若成立则不再对乙、丙、丁进行测试。

2）若不成立，则再假定乙为窃贼，即A=0、B=1、C=0、D=0，代入条件⑤测试是否成立，若成立则可确定乙为窃贼，不再对丙、丁进行测试。

3）若不成立，再假定丙为窃贼，即A=0、B=0、C=1、D=0，代入条件⑤测试是否成立，若成立则可确定丙为窃贼，不再对丁进行测试。

4）若不成立，再假定丁为窃贼，即A=0、B=0、C=0、D=1，代入条件⑤测试是否成立，若成立则确定丁为窃贼。

4．确定程序框架

程序的流程图如图6.7所示。

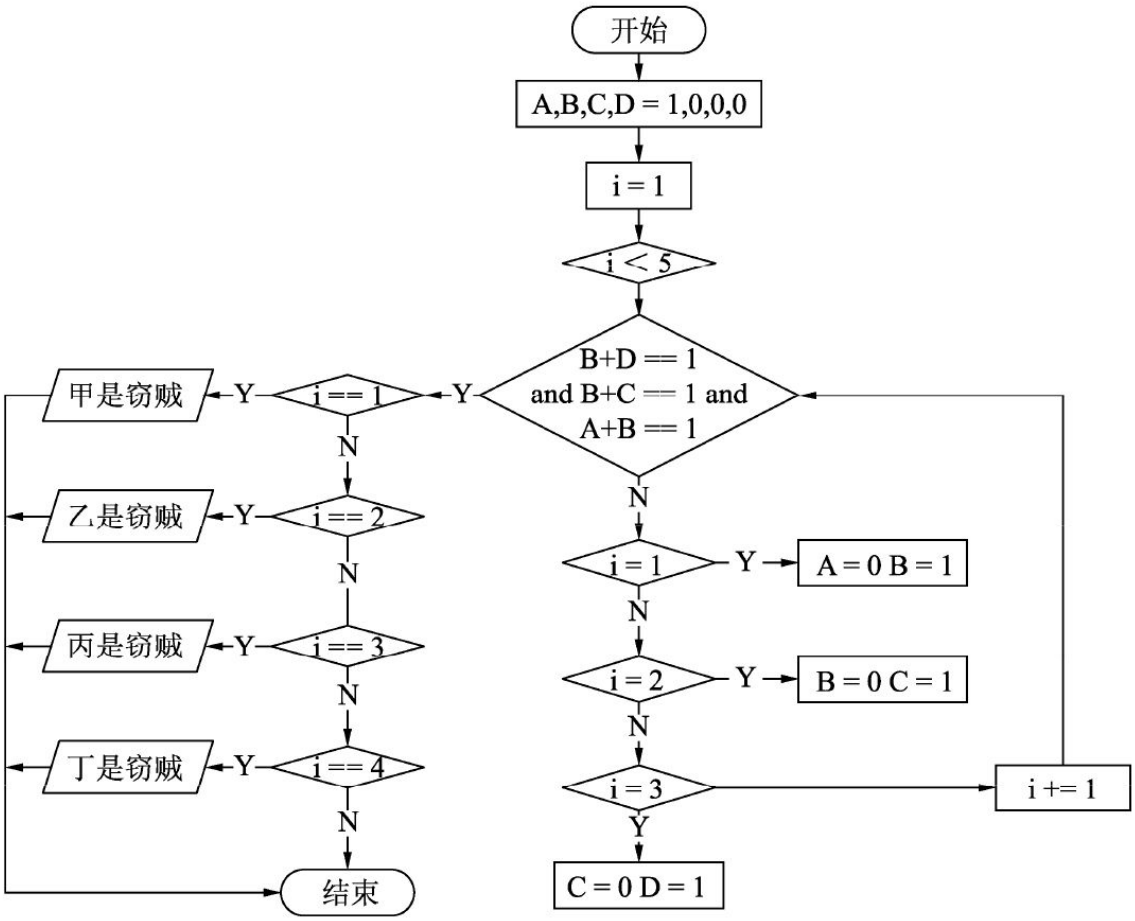


图6.7　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-
#author: liuhefei
#desc: 谁是窃贼

if __name__ == "__main__":
    #甲、乙、丙、丁分别用A、B、C、D代表。A、B、C、D的值要么为1，要么为0
    # 为1表示是窃贼，为0表示不是
    # 满足4个条件：B+D=1，B+C=1，A+B=1，A+B+C+D=1
```

```
A, B, C, D = 1, 0, 0, 0
for i in range(1, 4+1):                                    #i=1,2,3,4
    if B+D == 1 and B+C == 1 and A+B == 1:
        break
    else:
        if i == 1:
            A=0
            B=1
        if i == 2:
            B=0
            C=1
        if i == 3:
            C=0
            D=1
print("判断结果：")
if i == 1:
    print("甲是窃贼\n")
if i == 2:
    print("乙是窃贼\n")
if i == 3:
    print("丙是窃贼\n")
if i == 4:
    print("丁是窃贼\n")
```

程序说明如下：

• 使用变量i控制循环次数，因为要分别对甲、乙、丙、丁进行测试，故循环次数为4。

• 输出时根据i的取值来确定输出结果。

6. 运行结果

在PyCharm下运行程序，结果如图6.8所示。由运行结果可知，乙是窃贼。

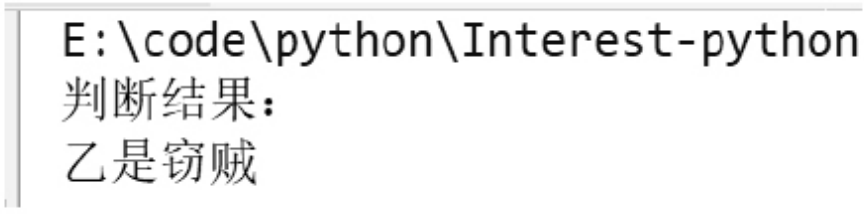


图6.8　运行结果

6.5　旅客国籍

1. 问题描述

在一个旅馆中住着6个不同国籍的人，他们分别来自美国、德国、英国、法国、俄罗斯和意大利。他们的名字分别叫A、B、C、D、E和F，要说明的是名字的顺序与前面提到的国籍不一定是相互对应的。现在已知：

1）A和美国人是医生。

2）E和俄罗斯人是教师。

3）C和德国人是技师。

4）B和F曾经当过兵，而德国人从未参过军。

5）法国人比A年龄大，意大利人比C年龄大。

6）B同美国人下周要去西安旅行，而C同法国人下周要去杭州度假。

现要求根据上述已知条件，编程求出A、B、C、D、E和F各是哪国人。

2. 问题分析

根据问题描述中给定的条件可进行如下分析：

- 由“A和美国人是医生”可知A不是美国人。
- 由“E和俄罗斯人是教师”可知E不是俄罗斯人。
- 由“C和德国人是技师”可知C不是德国人。

• 又因为A的职业是医生，与俄罗斯人和德国人的职业不同，故A不是俄罗斯人也不是德国人。E的职业是教师，与美国人和德国人的职业不同，故E不是美国人也不是德国人。C的职业是技师，与美国人和俄罗斯人不同，故C不是美国人也不是俄罗斯人。

• 由“B和F曾经当过兵，而德国人从未参过军”可知，B和F不是德国人。

• 由“法国人比A年龄大，意大利人比C年龄大”可知，A不是法国人，C不是意大利人。

• 由“B同美国人下周要去西安旅行，而C同法国人下周要去杭州度假”可知，B不是美国人，也不是法国人，C不是法国人。

用条件矩阵将上面的分析结果表示出来：

	美国	英国	法国	德国	意大利	俄罗斯
A	0		0	0		0
B	0		0	0		
C	0		0	0	0	0
D						
E	0			0		0
F				0		

根据上面的条件矩阵使用消去法，即可得到问题的结果。

3. 算法设计

下面给出从条件矩阵初始化到完成消去得到结果的整个过程中条件矩阵每一步的变化，矩阵中加粗的元素表示在该步骤中该元素发生了变化。具体变化过程如下：

1）初始化条件矩阵。

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

状态（a）

2）将问题分析中得出的结果在条件矩阵中表示出来。

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & \mathbf{0} & 2 & \mathbf{0} & \mathbf{0} & 5 & \mathbf{0} \\ 0 & \mathbf{0} & 2 & \mathbf{0} & \mathbf{0} & 5 & 6 \\ 0 & \mathbf{0} & 2 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & \mathbf{0} & 2 & 3 & \mathbf{0} & 5 & 6 \\ 0 & 1 & 2 & 3 & \mathbf{0} & 5 & 6 \end{pmatrix}$$

状态（b）

条件矩阵中为0的项表示不是该国的人，同时将条件矩阵中每一列的0号元素都置为1，用来表示该列尚未进行处理。

3）第4列只有一个元素为0，执行消去操作。

执行消去操作时从只有一个元素为非0的列开始消去。

$$\begin{pmatrix} 0 & 1 & 1 & 1 & \mathbf{0} & 1 & 1 \\ 0 & 0 & 2 & 0 & 0 & 5 & 0 \\ 0 & 0 & 2 & 0 & 0 & 5 & 6 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & 4 & \mathbf{0} & \mathbf{0} \\ 0 & 0 & 2 & 3 & 0 & 5 & 6 \\ 0 & 1 & 2 & 3 & 0 & 5 & 6 \end{pmatrix}$$

状态（c）

将条件矩阵中非零元素所在行的其他元素都置为0，同时置a[0][4]为0，表示条件矩阵中第4列已处理完毕。

4）第1列只有一个元素为0，执行消去操作。

显然，当条件矩阵位于状态（c）时，第1列中只有一个元素非0，可执行消去操作。

$$\begin{pmatrix} 0 & \mathbf{0} & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 & 0 & 5 & 0 \\ 0 & 0 & 2 & 0 & 0 & 5 & 6 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 2 & 3 & 0 & 5 & 6 \\ 0 & 1 & \mathbf{0} & \mathbf{0} & 0 & \mathbf{0} & \mathbf{0} \end{pmatrix}$$

状态（d）

5）第3列只有一个元素为0，执行消去操作。

显然，当条件矩阵位于状态（d）时，第3列中只有一个元素非0，可执行消去操作。

$$\begin{pmatrix} 0 & 0 & 1 & \mathbf{0} & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 & 0 & 5 & 0 \\ 0 & 0 & 2 & 0 & 0 & 5 & 6 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & \mathbf{0} & 3 & 0 & \mathbf{0} & \mathbf{0} \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

状态（e）

6）第6列只有一个元素为0，执行消去操作。

显然，当条件矩阵位于状态（e）时，第6列中只有一个元素非0，可执行消去操作。

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & \mathbf{0} \\ 0 & 0 & 2 & 0 & 0 & 5 & 0 \\ 0 & 0 & \mathbf{0} & 0 & 0 & \mathbf{0} & 6 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

状态（f）

7）第5列只有一个元素为0，执行消去操作。

显然，当条件矩阵位于状态（f）时，第5列中只有一个元素非0，可执行消去操作。

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & \mathbf{0} & 0 \\ 0 & 0 & \mathbf{0} & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

状态（g）

8）第2列只有一个元素为0，执行消去操作。

当条件矩阵位于状态（g）时，只有第2列还未被处理，经观察发现，该非零元素所在行（第3行）的所有其他元素都已经为0了，但在我们编程实现时，程序还是会对该行的每个元素都检查一遍。

$$\begin{pmatrix} 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

状态（h）

执行消去操作后的矩阵状态为状态（h）所示。该矩阵中除了第0列以外每列都只有一个元素非0，由此就可推断出A、B、C、D、E、F到底是哪国人。

4. 确定程序框架

总结上述矩阵状态的变化过程，可得出程序执行的简要流程，如图6.9所示。

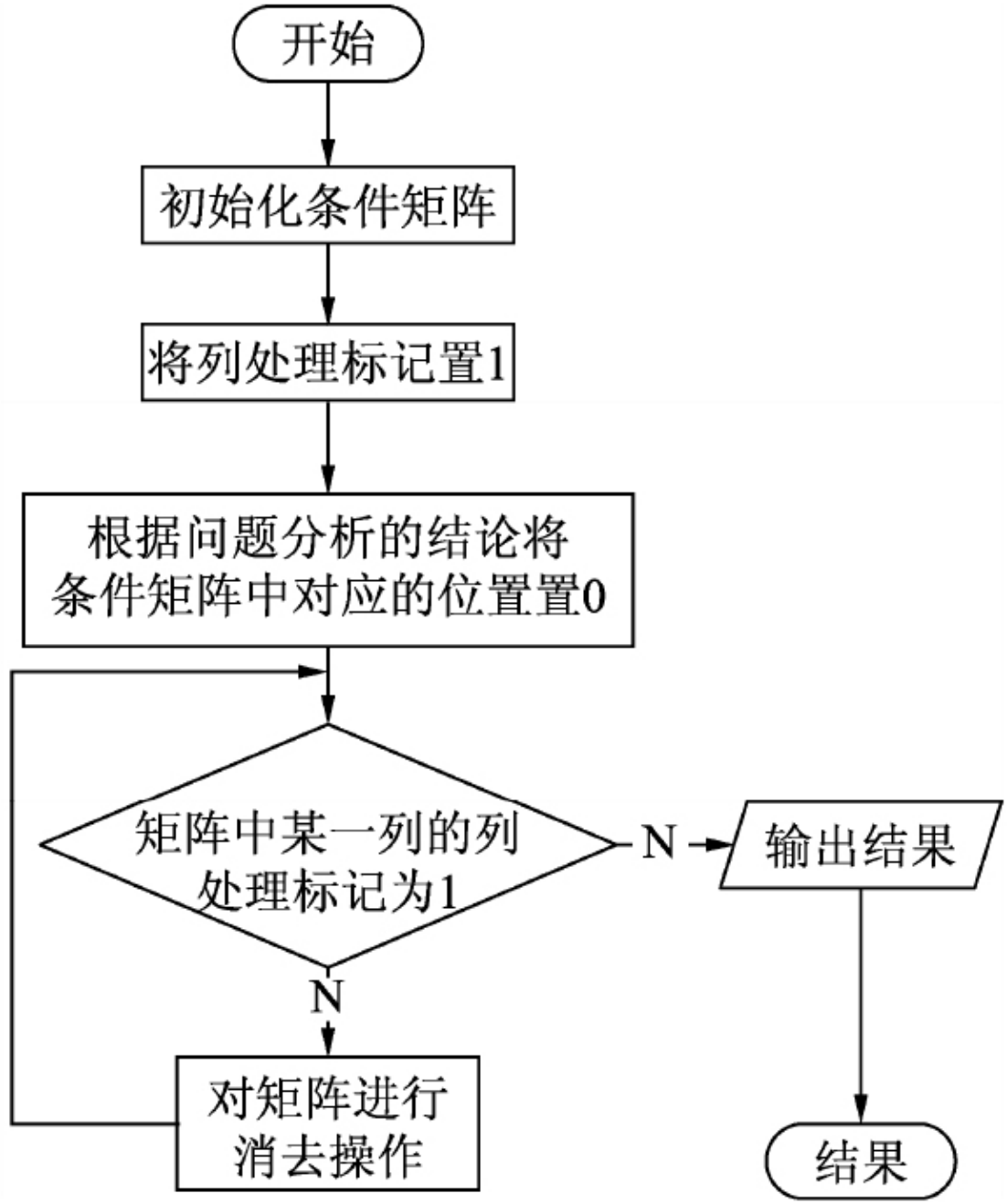


图6.9　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-
#author:liuhefei
#desc: 旅客国籍

country = [" ", "美国", "英国", "法国", "德国", "意大利", "俄罗斯"]                #国名
```

```
def Nationality():
    #初始化条件矩阵
    a = []
    for i in range(7):
        b = []
        for j in range(7):
            b.append(j)
        a.append(b)

    for i in range(1, 7):    #条件矩阵每一列的第0号元素作为该列数据处理的标记
        a[0][i] = 1                                                                    #标
记该列尚未处理

    a[1][1] = a[2][1] = a[3][1] = a[5][1] = 0        #输入条件矩阵中的各种条件
    a[1][3] = a[2][3] = a[3][3] = 0                              #0表示不是该国人
    a[1][4] = a[2][4] = a[3][4] = a[5][4] = a[6][4] = 0
    a[3][5] = 0
    a[1][6] = a[3][6] = a[5][6] = 0

    x = 0
    y = 0
    while a[0][1] + a[0][2] + a[0][3] + a[0][4] + a[0][5] + a[0][6] > 0:
        #当所有6列均处理完毕后退出循环
        for i in range(1, 7):                                          #i为列坐标
            if a[0][i] != 0:                                           #若该列尚未处理，则进
行处理
                e = 0
                for j in range(1, 7):        #j变量保存行坐标，e变量是该列中非0元素的
计数器
                    if a[j][i] != 0:
                        #统计每列中的非0元素个数
                        x = j                                          #x变量保存行坐标
                        y = i                                          #y变量保存列坐标
                        e += 1
                if e == 1:  #若该列只有一个元素为非零，则进行消去操作
                    for t in range(1, 7):
                        if t != i:
                            a[x][t] = 0                                #将非零元素所在行的其
他元素置0
                    a[0][y] = 0                                        #设置该列已处理完毕的
标记

    print("矩阵最终状态为：")
    #输出执行消去操作后矩阵的最终状态
    for i in range(7):
        print(a[i])

    print("推断结果为：")
    for i in range(1, 7):
        print(chr(ord('A') - 1 + i), '来自: ', end='')
        for j in range(1, 7):
            if a[i][j] != 0:
                print(country[a[i][j]], '。')
                break


if __name__ == '__main__':
    Nationality()
```

---

6. 运行结果

在PyCharm下运行程序，结果如图6.10所示。根据运行结果可知：A是意大利人，B是俄罗斯人，C是英国人，D是德国人，E是法国人，F是美国人。

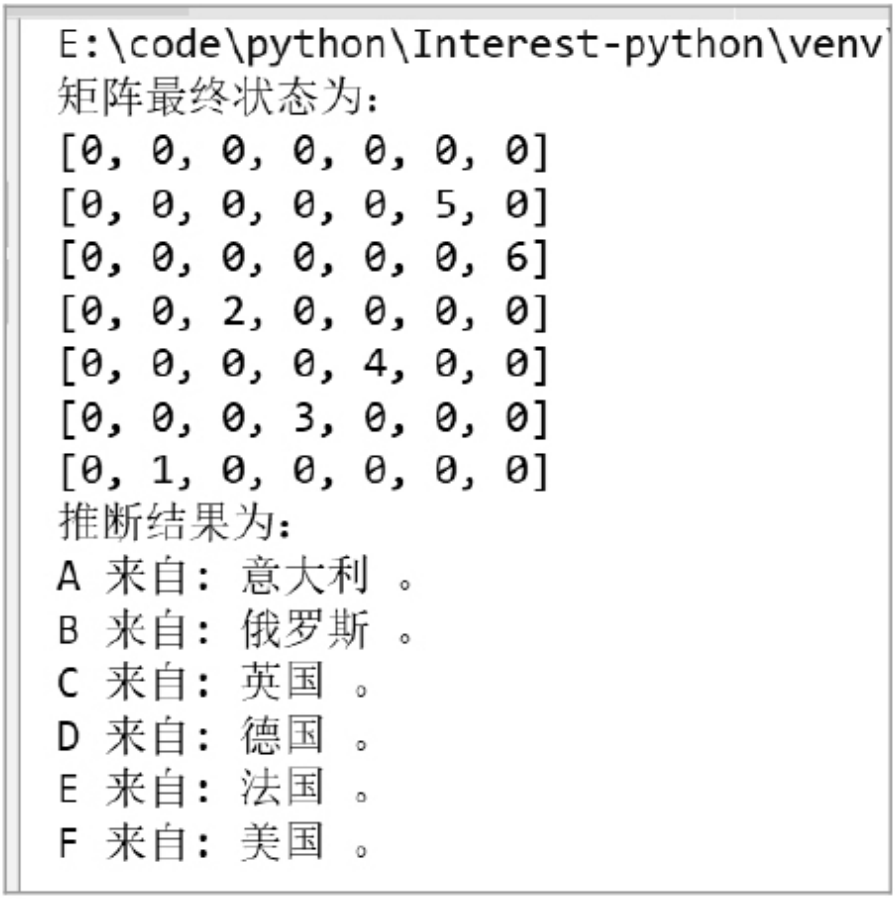


图6.10　运行结果

7．拓展训练

根据题意生成条件矩阵，再使用消去法进行推理判断是一种常用的方法，该方法对于解决较为复杂的逻辑问题是十分有效的，希望读者能够掌握。

6.6　委派任务

1. 问题描述

某项任务需要在A、B、C、D、E、F这6个人中挑选人来完成，但挑选人受限于以下条件：

1）A和B两个人至少去一人。

2）A和D不能同时去。

3）A、E和F三人中要挑选两个人去。

4）B和C同时去或者都不去。

5）C和D两人中只能去一个。

6）如果D不去，那么E也不去。

试编程求出应该让哪几个人去完成这项任务。

2. 问题分析

先将问题描述中的限定条件转换成表达式。假设用A、B、C、D、E、F这6个变量来表示6个人，如果某个人被选定去完成任务则对应的变量值为1；否则，如果某个人未被选中参加该项任务，则其对应的变量值为0。由此，将限定条件转换为如下表达式：

- A+B＞＝1——A和B两个人至少去一人。
- A+D!=2——A和D不能同时去。
- A+E+F==2——A、E和F三人中要挑选两个人去。
- B+C==0 or B+C==2——B和C同时去或者都不去。

• C+D==1——C和D两人中只能去一个。

• D+E==0 or D==1——如果D不去，那么E也不去（如果D去了，那么E可以去也可以不去）。

3. 算法设计

求解逻辑推理类问题的关键就是写出正确的逻辑表达式。由于Python中提供了丰富的算术和逻辑操作符，因此可以借助它们有效地将问题化繁为简。将求解问题中的限定条件用程序语言描述清楚后就可以使用穷举法来获得最终的判断结果。对此类问题的求解都可以依据这种思考方法来进行。

下面使用Python中的逻辑表达式将问题分析中得到的6个条件表达出来，这6个表达式之间是“与”的关系。逻辑表达式如下：

```
A+B>=1 and A+D!=2 and A+E+F= =2 and (B+C= =0 or B+C= =2) and C+D= =1 and (D+E= =0 or D= =1)
```

在程序中穷举每个人去或者不去的各种可能情况，并代入上面的逻辑表达式中进行推理运算，则使该逻辑表达式的值均为真的结果就是正确的结果。

4. 确定程序框架

程序的流程图如图6.11所示。

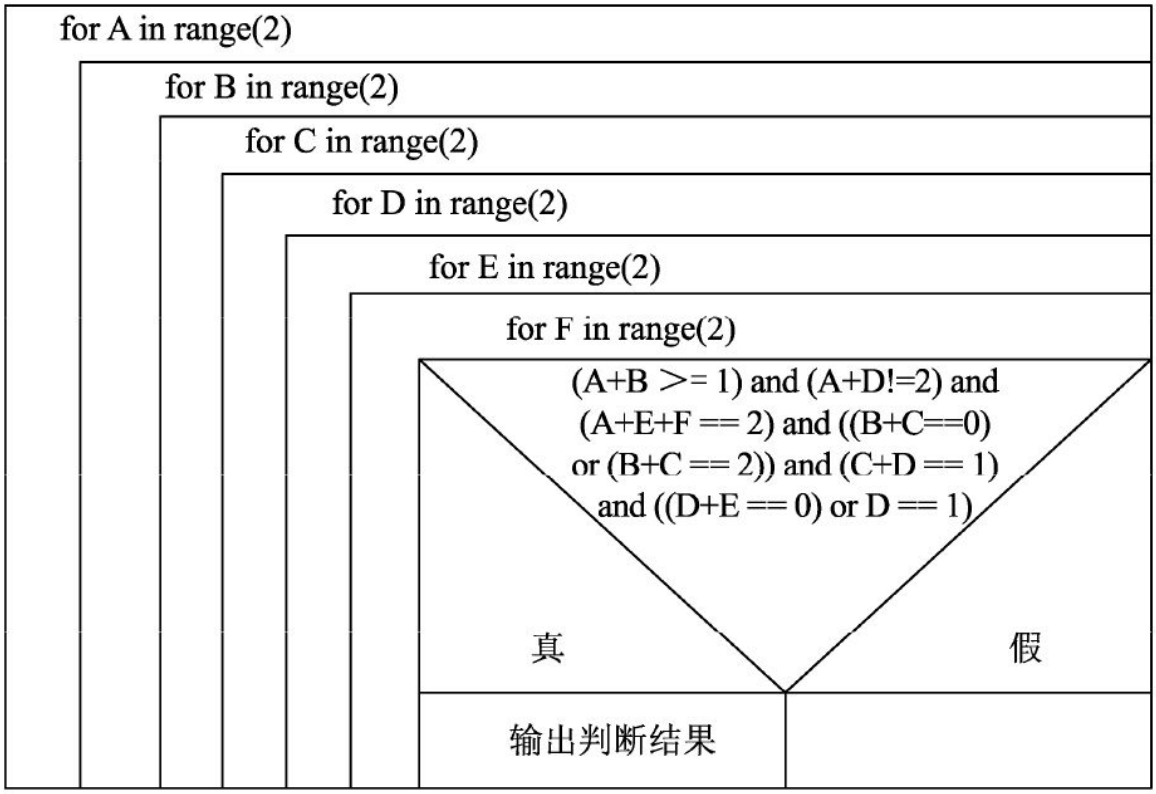


图6.11　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-
#author: liuhefei
#desc: 委派任务

if __name__ == "__main__":
    #A、B、C、D、E、F 可能的取值分别为0或1（0：不去，1：去）
    #穷举所有情况
    for A in range(2):
        for B in range(2):
            for C in range(2):
                for D in range(2):
                    for E in range(2):
                        for F in range(2):
                            #逻辑表达式作为判断条件
                            if (A+B >= 1) and (A+D != 2) and (A+E+F == 2)
                                and ((B+C == 0) or (B+C == 2)) and (C+D == 1)
and ((D+E == 0) or D == 1):
                                a = '' if A == 1 else "未"            #三元表达式
                                print("A" + a + "被选择去完成任务")

                                b = '' if B == 1 else "未"
                                print("B" + b + "被选择去完成任务")

                                c = '' if C == 1 else "未"
                                print("C" + c + "被选择去完成任务")
```

```
d = '' if D == 1 else "未"
print("D" + d + "被选择去完成任务")

e = '' if E == 1 else "未"
print("E" + e + "被选择去完成任务")

f = '' if F == 1
else "未"

print("F" + f +
"被选择去完成任务")
```

6. 运行结果

在PyCharm下运行程序，结果如图6.12所示。

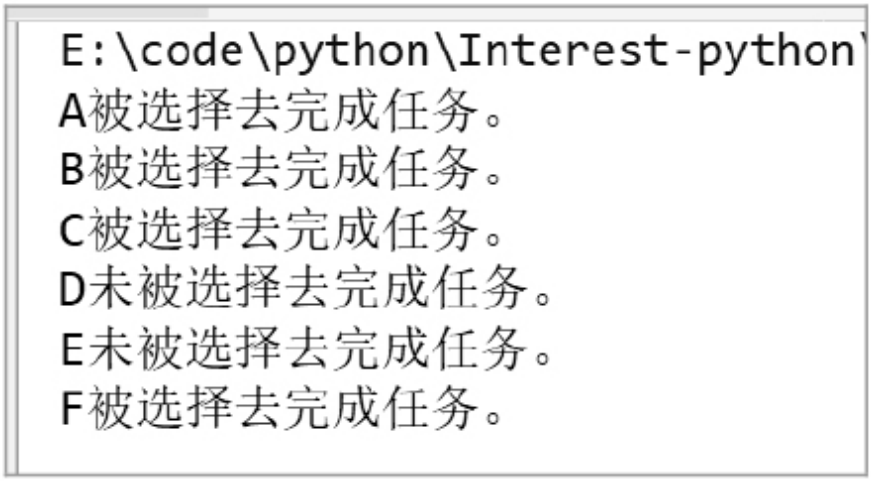


图6.12　运行结果

6.7　谜语博士的难题

谜语博士遇到了两个难题，先来看第一个难题。

6.7.1　谜语博士的难题（一）

1. 问题描述

诚实族和说谎族是来自两个岛屿的不同民族，已知诚实族的人永远说真话，而说谎族的人永远说假话。

一天，谜语博士遇到三个人，知道他们可能是来自诚实族或说谎族的人。为了调查这三个人到底来自哪个族，博士分别问了他们问题，下面是他们的对话：

博士问：“你们是什么族的？”

第一个人回答说：“我们之中有两个来自诚实族。”

第二个人说：“不要胡说，我们三个人中只有一个是来自诚实族的。”

第三个人接着第二个人的话说：“对，确实只有一个是诚实族的。”

请根据他们的回答编程判断出他们分别是来自哪个族的。

2. 问题分析

假设这三个人分别用A、B、C三个变量来代表，若某个人说谎则其对应的变量值为0，若诚实则其对应的变量值为1。根据题目中三个人的话分析如下：

1）第一个人回答说：“我们之中有两个来自诚实族。”

因此如果第一个人（用A代表第一个人）说的是真话，则A来自诚实族，另外两个人中一个来自诚实族，一个来自说谎族，则有表达式(not A)and A+B+C==2

如果A说的是假话，则A来自说谎族，A的话也一定是假话，因此三个人中来自诚实族的人必定不是两个，则有表达式(not A) and A+B+C !=2

2）第二个人说：“不要胡说，我们三个人中只有一个是来自诚实族的。”

如果第二个人（用B来代表第二个人）说的是真话，则B来自诚实族，而另外两个人都来自说谎族，则有表达式B and A+B+C==1

如果第二个人B说的是假话，则B来自说谎族，且三个人中来自诚实族的人数必定不是一个，则有表达式(not B) and A+B+C !=1

3）第三个人说：“对，确实只有一个是诚实族的。”

如果第三个人（用C代表第三个人）说的是真话，则C来自诚实族，另外两个人都来自说谎族。则有表达式C and A+B+C==1

如果C说的是假话，则C来自说谎族，且有表达式：(not C) and A+B+C !=1

3. 算法设计

在问题分析中我们已经列出了各种可能情况，接下来就可以使用穷举法来获得最终的判断结果了。

下面使用Python中的逻辑表达式将问题分析中得到的6个条件表达出来，逻辑表达式如下：

```
(A and A+B+C = =2 or (not A) and A+B+C !=2) and (B and A+B+C = =1 or (not B)
and A+B+C !=1)
and (C and A+B+C = =1 or (not C) and A+B+C !=1)
```

在程序中穷举每个人的各种可能情况，并代入上面的逻辑表达式中进行推理运算，则使该逻辑表达式的值均为真的结果就是正确的结果。

4. 确定程序框架

程序的流程图如图6.13所示。

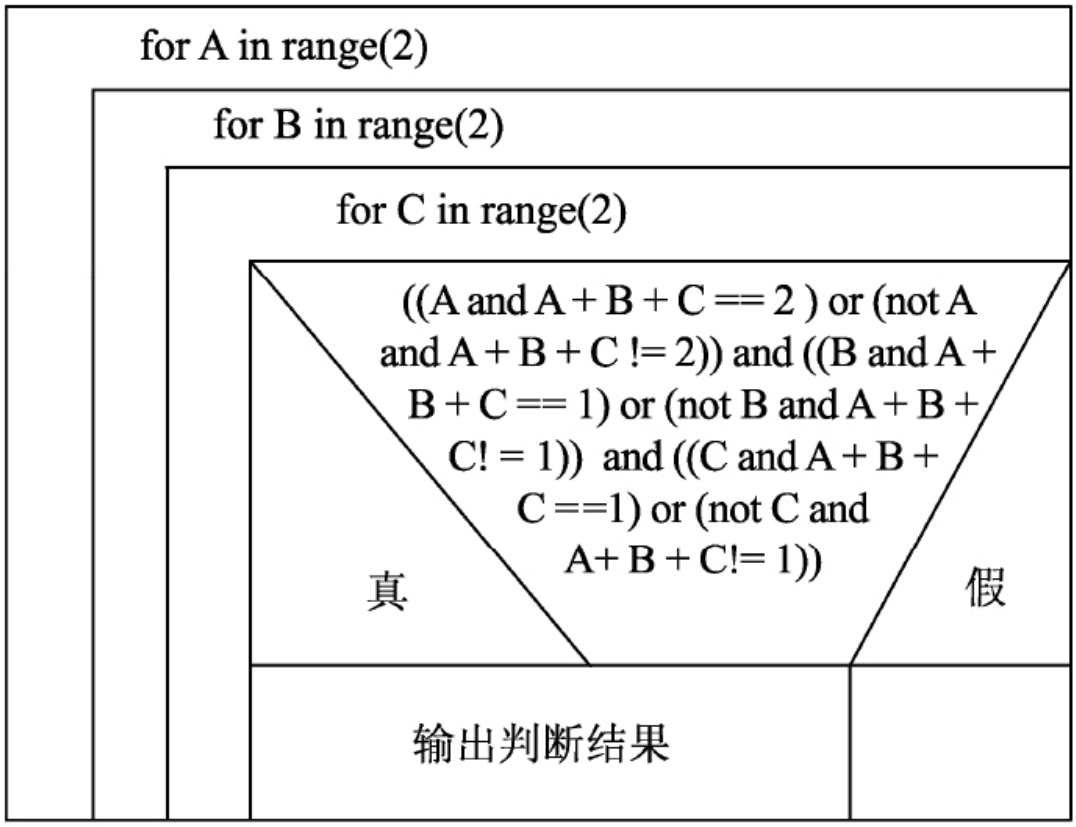


图6.13　程序流程图

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-
#author: liuhefei
#desc: 谜语博士的难题(一)

if __name__ == "__main__":
    #分别使用A、B、C代表第一个、第二个和第三个人，说谎对应的变量值为0，诚实对应的变
     量值为1
    for A in range(2):
        for B in range(2):
            for C in range(2):
                #逻辑判断条件
                if ((A and A + B + C == 2) or (not A and A + B + C != 2))
                        and ((B and A + B + C == 1) or (not B and A + B + C !=
1))
                        and ((C and A + B + C == 1) or (not C and A + B + C !=
1)):
                    a = "诚实族" if A else "说谎族"
```

```
b = "诚实族" if B else "说谎族"
c = "诚实族" if C else "说谎族"
print("第一个人来自" + a )
print("第二个人来自" + b)
print("第三个人来自" + c)
```

还可以将上面的程序中的逻辑判断条件分开判断，即使用多个if语句，代码如下：

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-
#author: liuhefei
#desc: 谜语博士的难题(一)

if __name__=="__main__":
    # 分别使用A、B、C代表第一个、第二个和第三个人，说谎对应的变量值为0，诚实对应的变量值为1
    for A in range(2):
        for B in range(2):
            for C in range(2):
                #使用多个if语句判断
                if (A and A + B + C == 2) or (not A and A + B + C != 2):
                    if(B and A + B + C == 1) or (not B and A + B + C != 1):
                        if(C and A + B + C == 1) or (not C and A + B + C != 1):
                            if A == 0:
                                a = "说谎族"
                            else:
                                a = "诚实族"
                            if B == 0:
                                b = "说谎族"
                            else:
                                b = "诚实族"
                            if C == 0:
                                c = "说谎族"
                            else:
                                c = "诚实族"
                            #print(A,B,C)
                            print("第一个人来自" + a)
                            print("第二个人来自" + b)
                            print("第三个人来自" + c)
```

两种方式的运行结果都是相同的。

6. 运行结果

在PyCharm下运行程序，结果如图6.14所示。由图6.14可见，这三个人都来自说谎族。

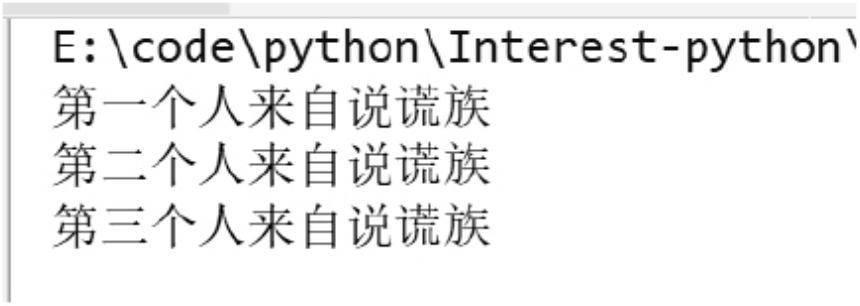

E:\code\python\Interest-python\
第一个人来自说谎族
第二个人来自说谎族
第三个人来自说谎族

图6.14　运行结果

6.7.2　谜语博士的难题（二）

1. 问题描述

两面族是岛屿上的一个新民族，他们的特点是说话时一句真话一句假话，真假交替。即如果第一句说的是真话，则第二句必为假话；如果第一句说的是假话，则第二句必然是真话。但第一句话到底是真假却不得而知。

现在谜语博士碰到了三个人，这三个人分别来自三个不同的民族，诚实族、说谎族和两面族。谜语博士和这三个人分别进行了对话。

首先，谜语博士问左边的人：“中间的人是哪个族的？”左边的人回答说：“是诚实族的。”

谜语博士又问中间的人：“你是哪个族的？”中间的人回答说：“两面族的。”

最后，谜语博士问右边的人：“中间的人到底是哪个族的？”右边的人回答说：“是说谎族的。”

现在请编程求出这三个人各自来自哪个族。

2. 问题分析

显然，谜语博士碰到的第二个难题要比第一个难题更为复杂。但是解题思路与第一个难题是类似的，相信读者已经对该类问题的解题方法有所了解了。

由于现在不仅需要判断三个人的民族，而且这三个人还存在相对的位置关系，因此像谜语博士的难题（一）中那样简单定义三个变量还不足以描述难题（二）中的情况。

首先还是用变量将三个民族表示出来，表示的时候还要考虑到他们之间的位置关系。我们可以采用如下方式来定义变量：

- 变量L=1，表示左边的人来自诚实族。
- 变量M=1，表示中间的人来自诚实族。
- 变量R=1，表示右边的人来自诚实族。
- 变量LL=1，表示左边的人来自两面族。
- 变量MM=1，表示中间的人来自两面族。
- 变量RR=1，表示右边的人来自两面族。

根据上述变量定义方式，有：

- 左边的人来自说谎族，则L!=1且LL!=1。
- 中间的人来自说谎族，则M!=1且MM!=1。
- 右边的人来自说谎族，则R!=1且RR!=1。

从上述变量定义可以看到，为解决第二个难题，变量的数目已经变为6个。下面我们来分析题目中谜语博士与三个人的对话。

根据题目中三个人的回答做分析如下。

1）左边的人说中间的人是诚实族的。

若左边的人说的是真话，则他来自诚实族，且中间的人也是诚实族的。这种情况可用表达式表达为：

```
L and (not LL) and M and (not MM)
```

上面的表达式的含义为左边及中间的人是诚实族的同时不可能是两面族的。

若左边的人说的是假话，则可以肯定他不是诚实族的，且中间的人也不是诚实族的。这种情况可用表达式表达为：

```
(not L) and (not M)
```

上面的表达式的含义为左边及中间的人肯定不是诚实族的，但不能确定他们到底来自说谎族还是两面族。

综合起来，根据左边人的回答可得到逻辑表达式：

```
(L and (not LL) and M and (not MM)) or ( (not L) and (not M))
```

2）中间的人说自己是两面族的。

若中间的人说的是真话，则他来自两面族；若中间的人说的是假话，则他来自说谎族。因此，可以判断出中间的人肯定不是诚实族的。这种情况可用表达式表达为：

```
not M
```

3）右边的人说中间的人是说谎族的。

• 若右边的人来自诚实族，则中间的人是说谎族的。这种情况可用表达式表达为：

```
R and (not M) and (not MM)
```

• 若右边的人来自两面族，则无法判断其话的真假，即无法确定中间的人来自哪个族。这种情况可用表达式表达为：

```
RR and (not R)
```

• 若右边的人来自说谎族，且中间的人不是说谎族的，而是诚实族或两面族的。这种情况可用表达式表达为：

```
(not R) and (not RR) and (M or MM)
```

综合起来，根据右边的人的回答可得到逻辑表达式：

```
R and (not M) and (not MM) or (RR and (not R)) or ((not R) and (not RR) and (M
or MM))
```

4）由于题目中说“三个人分别来自三个不同的民族”，因此可以得出如下表达式：

```
(L+LL !=2 and M+MM !=2 and R+RR !=2) and (L+M+R ==1 and LL+MM+RR = =1)
```

3. 算法设计

在问题分析中我们已经列出了各种可能的情况，接下来仍然使用穷举法来获得最终的判断结果。

下面使用Python语言中的逻辑表达式将问题分析中得到的全部逻辑条件表达出来，具体如下：

```
(L and (not LL) and M and (not MM)) or ( (not L) and (not M)) and (not M)
and (R and (not M) and (not MM) or (RR and (not R)) or ((not R) and (not RR)
and (M or MM)))
and L+LL !=2 and M+MM !=2 and R+RR !=2 and L+M+R ==1 and LL+MM+RR = =1
```

在程序中穷举每个人的各种可能情况，并代入上面的逻辑表达式中进行推理运算，则使该逻辑表达式的值均为真的结果就是正确的结果。

4. 确定程序框架

该程序的流程图如图6.15所示。

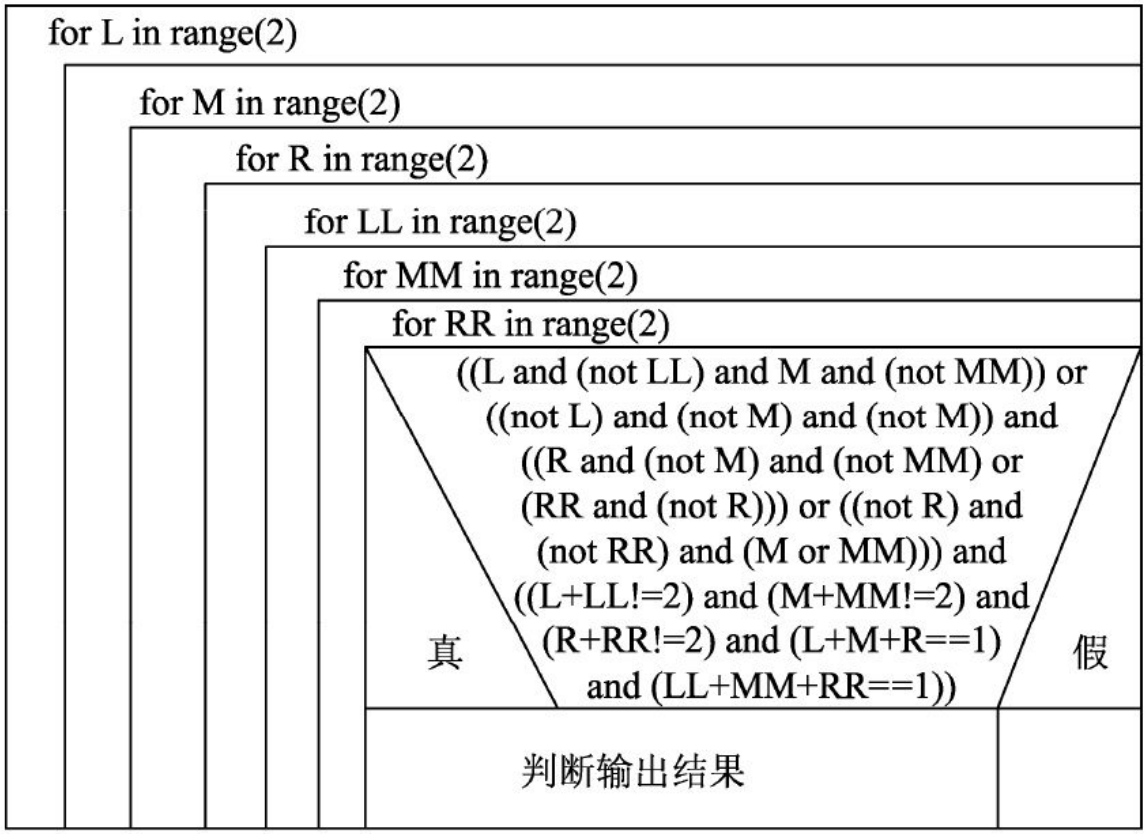


图6.15　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-
#author: liuhefei
#desc: 谜语博士的难题(二)

if __name__=="__main__":
    #分别使用变量L、M、R表示左边、中间、右边的人来自诚实族
    #分别使用变量LL、MM、RR表示左边、中间、右边的人来自两面族
    #0表示说谎，1表示诚实
    for L in range(2):                                      #穷举
        for M in range(2):
            for R in range(2):
                for LL in range(2):
                    for MM in range(2):
                        for RR in range(2):
                            if ((L and (not LL) and M and (not MM)) or ((not L)
and (not M)) and (not M)):
                                if ((R and (not M) and (not MM) or (RR and (not
R)))
                                        or ((not R) and (not RR) and (M or
MM))):
                                    if ((L + LL != 2) and (M + MM != 2) and (R
+ RR != 2)
                                                and (L + M + R == 1) and (LL + MM
+
```

```
RR == 1)):
                                                    # 使用三元表达式
                                                    l = "两面族" if LL else ("诚实族" if L


else
"说谎族")
                                                    m = "两面族" if MM else ("诚实族" if M


else
"说谎族")
                                                    r = "两面族" if RR else ("诚实族" if R


else
"说谎族")
                                                    print("左边的人来自" + l)
                                                    print("中间的人来自" + m)
                                                    print("右边的人来自" + r)
```

6. 运行结果

在PyCharm下运行程序，结果如图6.16所示。由图6.16可见，左边的人来自两面族，中间的人来自说谎族，右边的人来自诚实族。

图6.16　运行结果

6.8 黑与白

1. 问题描述

有A、B、C、D、E这5个人，每个人额头上都贴了一张黑或白的纸。5人对坐，每个人都可以看到其他人额头上纸的颜色。

A说：“我看见有3人额头上贴的是白纸，1人额头上贴的是黑纸。”

B说：“我看见其他4人额头上贴的都是黑纸。”

C说：“我看见1人额头上贴的是白纸，其他3人额头上贴的是黑纸。”

D说：“我看见4人额头上贴的都是白纸。”

E什么也没说。

现在已知额头上贴黑纸的人说的都是谎话，额头上贴白纸的人说的都是实话。问这5人谁的额头上贴的是白纸，谁的额头上贴的是黑纸？

2. 问题分析

该问题是一个逻辑推理问题。分析A、B、C、D这4个人所说的话可以得出4个条件。

假设用变量a、b、c、d、e分别代表A、B、C、D、E这5个人额头上贴纸的颜色，当变量的取值为1时表示该人额头上贴纸的颜色为白色，当变量的取值为0时表示该人额头上贴纸的颜色为黑色。则分析题目中4个人所说的话如下：

A说：“我看见有三人额头上贴的是白纸，一人额头上贴的是黑纸。”

如果A额头上贴的是白纸，那么他说的是实话，则有表达式：a and b+c+d+e==3。

如果A额头上贴的是黑纸，那么他说的是谎话，则有表达式：(not a) and b+c+d+e!=3。

B说：“我看见其他4人额头上贴的都是黑纸。”

如果B额头上贴的是白纸，那么他说的是实话，则有表达式：b and a+c+d+e==0。

如果B额头上贴的是黑纸，那么他说的是谎话，则有表达式：(not b) and a+c+d+e!=0。

C说：“我看见一人额头上贴的是白纸，其他三人额头上贴的是黑纸。”

如果C额头上贴的是白纸，那么他说的是实话，则有表达式：c and a+b+d+e==1。

如果C额头上贴的是黑纸，那么他说的是谎话，则有表达式：(not c) and a+b+d+e!=1。

D说：“我看见4人额头上贴的都是白纸。”

如果D额头上贴的是白纸，那么他说的是实话，则有表达式：d and a+b+c+e==4。

如果D额头上贴的是黑纸，那么他说的是谎话，则有表达式：(not d) and a+b+c+e!=4。

3. 算法设计

求解逻辑推理类问题的关键就是写出正确的逻辑表达式。将问题分析中列出的限定条件用程序语言描述清楚后就可以使用穷举法来获得最终的判断结果。

下面使用Python中的逻辑表达式将问题分析中得到的几个条件表达出来，得到的逻辑表达式如下：

```
(a and b+c+d+e = =3 or (not a) and b+c+d+e !=3) and (b and a+c+d+e = =0 or (not
b) and a+c+d+e !=0) and
( c and a+b+d+e ==1 or (not c) and a+b+d+e !=1) and (d and a+b+c+e = =4 or (not
d) and a+b+c+e !=4)
```

在程序中穷举每个人额头贴纸颜色的所有可能情况，并代入上面的逻辑表达式中进行推理运算，则使该逻辑表达式的值均为“真”的结果就是正确的结果。

4. 确定程序框架

在程序中使用五重循环来穷举所有可能情况，该程序的流程图如图6.17所示。

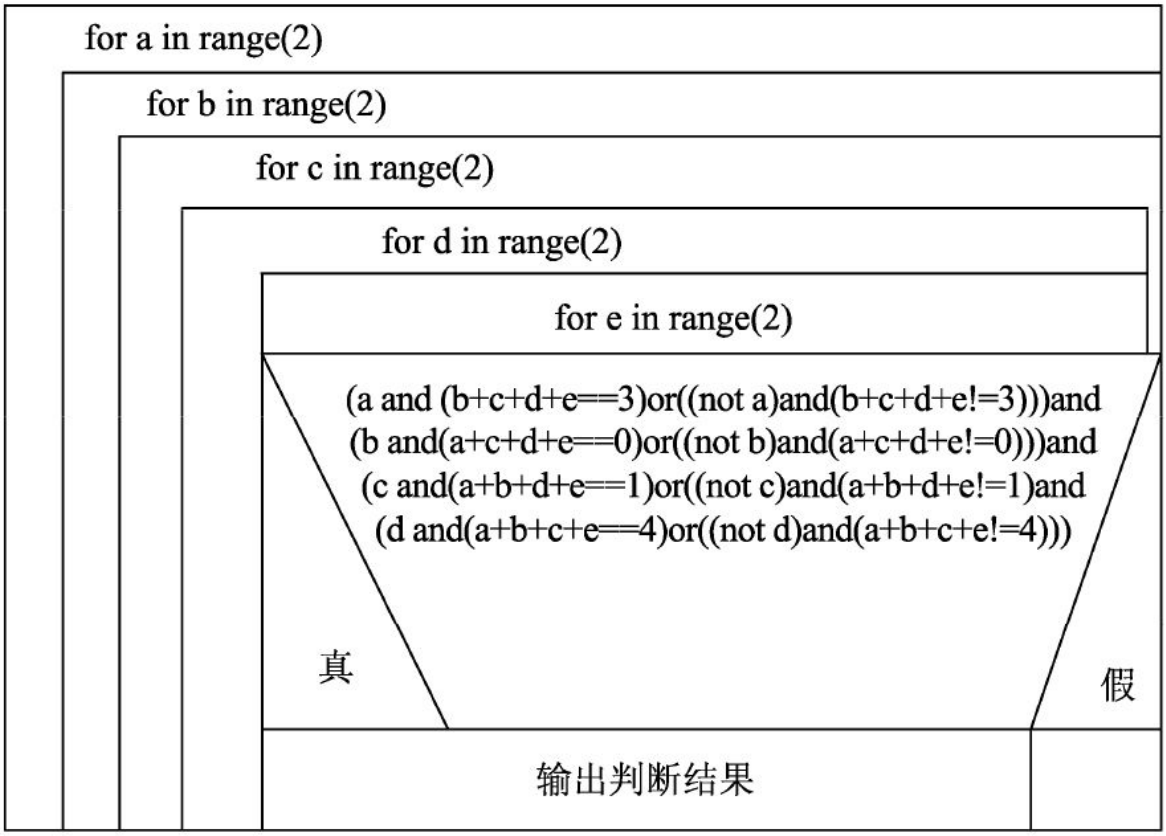


图6.17　程序流程图

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-
#author:liuhefei
#desc: 黑与白

if __name__=="__main__":
    #A、B、C、D、E这5个人，每个人额头上都贴了一张黑或白的纸
    #变量a、b、c、d、e分别代表A、B、C、D、E这5个人额头上贴纸的颜色，
    # 当变量的取值为1时表示该人额头上贴纸为白色，当变量取值为0时表示该人额头上贴纸为
     黑色
    for a in range(2):       #a、b、c、d、e变量的值要么为0，要么为1，0为黑色，1为白色
        for b in range(2):
            for c in range(2):
                for d in range(2):
                    for e in range(2):
                        if (a and (b + c + d + e == 3) or ((not a) and (b + c
+ d + e != 3))):   #A
                            if(b and (a + c + d + e == 0) or ((not b) and (a +
c + d + e != 0))):  #B
                                if(c and (a + b + d + e == 1) or ((not c) and
(a + b + d + e != 1))): #C
                                    if (d and (a + b + c + e == 4) or ((not d)
and (a + b + c + e != 4))): #D
                                        a1 = "白" if a==1 else "黑"    #三元表达式
                                        b1 = "白" if b==1 else "黑"
                                        c1 = "白" if c==1 else "黑"
                                        d1 = "白" if d==1 else "黑"
                                        e1 = "白" if e==1 else "黑"
                                        print("A额头上的贴纸是" + a1 + "色的.")
                                        print("B额头上的贴纸是" + b1 + "色的.")
                                        print("C额头上的贴纸是" + c1 + "色的.")
                                        print("D额头上的贴纸是" + d1 + "色的.")
                                        print("E额头上的贴纸是" + e1 + "色的.")
```

6. 运行结果

在PyCharm下运行程序，结果如图6.18所示。由图6.18可见，A、B、D额头上贴的是黑纸，C、E额头上的贴的是白纸。

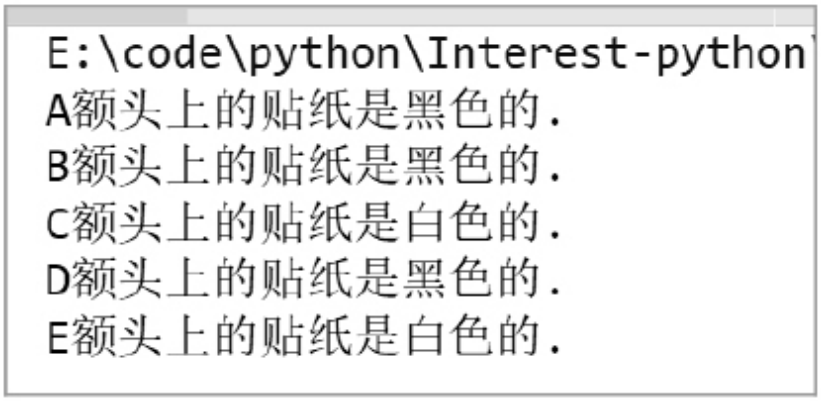


图6.18　运行结果

第7章　趣味游戏

本章以趣味游戏作为主题，为读者提供了8个小游戏。在对8个游戏分析讲解的过程中，结合对Python语言语法知识的回顾和总结，使读者在提高编程技巧的同时查漏补缺，牢固掌握Python语言的语法。

通过一些趣味小游戏学习编程，可以极大地提高读者学习Python语言程序设计的兴趣，起到事半功倍的效果。

7.1　黑白子交换

1. 问题描述

有3个白子和3个黑子如图7.1所示布置。

图7.1　初始位置

游戏的目的是用最少的步数将图7.1中白子和黑子的位置进行交换，使得最终结果如图7.2所示。

图7.2　最终位置

游戏的规则如下：

1）一次只能移动一个棋子。

2）棋子可以向空格中移动，也可以跳过一个对方的棋子进入空格。

3）白色棋子只能向右移动，黑色棋子只能向左移动，不能跳过两个棋子。

请用计算机实现上述游戏。

2. 问题分析

计算机解决这类问题的关键是要找出问题的规律，或者说是要制定一套计算机行动的规则。分析本题，先用人来解决问题，可总

结出以下规则：

1）黑子向左跳过白子落入空格，转（5）。

2）白子向右跳过黑子落入空格，转（5）。

3）黑子向左移动一格落入空格（但不应产生棋子阻塞现象），转（5）。

4）白子向右移动一格落入空格（但不应产生棋子阻塞现象），转（5）。

5）判断游戏是否结束，若没有结束，则转（1）继续。

所谓“阻塞”现象指的是在移动棋子的过程中，两个尚未到位的同色棋子连接在一起，使棋盘中的其他棋子无法继续移动。

例如，按下列方法移动棋子（“○”代表白子，“●”代表黑子，“△”代表空格）。

0：○ ○ ○ △ ● ● ●

1：○ ○ △ ○ ● ● ●

2：○ ○ ● ○ △ ● ●

3：○ ○ ● △ ○ ● ●

4：两个●连在一起产生阻塞

○ ○ ● ● ○ △ ●

或两个○连在一起产生阻塞

○ △ ● ○ ○ ● ●

产生阻塞现象的原因是在第2步时棋子○不能向右移动，而只能将●向左移动。

总结产生阻塞的原因是当棋盘出现“黑、白、空、黑”或“白、空、黑、白”状态时，不能向左或向右移动中间的棋子，而只移动两边的棋子。按照上述规则，可以保证在移动棋子的过程中不会出现棋子无法移动的现象，并且可以用最少的步数完成白子和黑子的位置交换。

3. 算法设计

可以有4种移动方式（“○”代表白子，“●”代表黑子，“△”代表空格，“～”代表任意）。

- 白棋跳过黑棋：～ ○ ● △ ～ ～ ～
- 黑棋跳过白棋：～ ～ △ ○ ● ～ ～
- 白棋移向空格：～ ～ ～ ○ △ ～ ～
- 黑棋移向空格：～ ～ ～ △ ● ～ ～

（1）黑白棋要是能跳，则先跳

根据游戏规则，如果出现下列情况1，黑棋不能向右，此时只能白棋跳过黑棋向右；同样，如果出现下列情况2，白棋不能向左，此时只能黑棋跳过白棋向左。

情况1：～ ○ ● △ ○ ～ ～

情况2：～ ● △ ○ ● ～ ～

是否存在黑棋既能向左跳，白棋又可向右跳的可能性呢？即，情况3或情况4同时存在的现象。

情况3：～ ○ ● △ ○ ● ～

情况4：○ ● △ ○ ● ～ ～

事实证明这两种情况是存在的。

（2）棋子只能移动时

1）若向右移动白子不会产生阻塞，则白子向右移动，分i=1和i>1两种情况：

• i=1时，白棋只能向右移。

○ △ ～ ～ ～ ～ ～

• i>1时，i处的白棋只有在i-1和i+2位置上的棋子不同时才能向右移动，即情况5或情况6。

情况5：～ ● ○ △ ○ ～ ～

情况6：～ ○ ○ △ ● ～ ～

分析：如果i-1和i+2位置上的棋子相同时，即情况7。

情况7：～ ～ ● ○ △ ● ～

如果将白子（“○”）向左移动到空格（“△”）处，会转变成情况8。如果在情况7时，第二个任意子（“～”）位置是白子（“○”），白子跳过黑子右移，会出现两个白子相连的情况，如情况9，将产生阻塞；或者出现倒数第二个黑子跳过白子左移，出现两个黑子相连的情况，如情况10，同样产生阻塞。

情况8：～ ～ ● △ ○ ● ～

情况9：～ △ ● ○ ○ ● ～

情况10：～ ～ ● ● ○ △ ～

相应代码如下：

```
i = 0
while flag and i < 5:                          # 若白子可以向右跳过黑子，则白子向右跳
    if t[i] == 1 and t[i + 1] == 2 and t[i + 2] == 0:
```

2）若向左移动黑子不会产生阻塞，则黑子向左移动，分i=5和i>1两种情况：

• i=5时，黑棋只能向左移。

～ ～ ～ ～ ～ △ ●

• i>1时，i+1处的黑棋只有在i-1和i+2位置上的棋子不同时才能向左移动，即，情况11或情况12。

情况11：～ ○ △ ● ● ～ ～

情况12：～ ● △ ● ○ ～ ～

相应代码如下：

```
i = 0
while flag and i < 5:                                   # 若黑子可以向左跳过白子，则黑子向左跳
    if t[i] == 0 and t[i + 1] == 1 and t[i + 2] == 2:
```

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @Time : 2019/7/13 10:07
# @desc: 黑白子交换

# 打印输出结果
def printLine(a):
    global number

    print("No. %2d:……" % number)
    number += 1
    print(" ", end='')

    for i in range(7):
        if a[i] == 1:
            print('| * ', end='')
        else:
            if a[i] == 2:
                print('| @ ', end='')
            else:
                print('|  ', end='')
    print(" |\n ……\n")

# 交换
def change(t, i, j):
```

```
    term = t[i]
    t[i] = t[j]
    t[j] = term
    return t


if __name__ == '__main__':
    t = [1, 1, 1, 0, 2, 2, 2]        # 初始化数组：1代表白子，2代表黑子，0代表空格
    number = 0
    printLine(t)
    # 若还没有完成棋子的交换则继续进行循环
    # 判断游戏是否结束
    while t[0] + t[1] + t[2] != 6 or t[4] + t[5] + t[6] != 3:
        # flag为棋子移动一步的标记，flag=True表示尚未移动棋子，flag=False表示已
          经移动棋子
        flag = True

        i = 0
        while flag and i < 5:     # 若白子可以向右跳过黑子，则白子向右跳
            if t[i] == 1 and t[i + 1] == 2 and t[i + 2] == 0:
                t = change(t, i, i + 2)      # 调用交换
                printLine(t)
                flag = False
            i += 1

        i = 0
        while flag and i < 5:             # 若黑子可以向左跳过白子，则黑子向左跳
            if t[i] == 0 and t[i + 1] == 1 and t[i + 2] == 2:
                t = change(t, i, i + 2)
                printLine(t)
                flag = False
            i += 1

        i = 0
        while flag and i < 6:             # 若向右移动白子不会产生阻塞，则白子向右移动
            f = True
            if i < 5:
                f = t[i - 1] != t[i + 2]
            if t[i] == 1 and t[i + 1] == 0 and (i == 0 or f):
                t = change(t, i, i + 1)
                printLine(t)
                flag = False
            i += 1

        i = 0
        while flag and i < 6:             # 若向左移动黑子不会产生阻塞，则黑子向左移动
            f = True
            if i < 5:
                f = t[i - 1] != t[i + 2]
            if t[i] == 0 and t[i + 1] == 2 and (i == 5 or f):
                t = change(t, i, i + 1)
                printLine(t)
                flag = False
            i += 1
```

---

5. 运行结果

在PyCharm下运行程序，结果如图7.3所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe
No.  0: ……
 | * | * | * |  | @ | @ | @  |
 ……
No.  1: ……
 | * | * |  | * | @ | @ | @  |
 ……
No.  2: ……
 | * | * | @ | * |  | @ | @  |
 ……
No.  3: ……
 | * | * | @ | * | @ |  | @  |
 ……
No.  4: ……
 | * | * | @ |  | @ | * | @  |
 ……
No.  5: ……
 | * |  | @ | * | @ | * | @  |
 ……
No.  6: ……
 |  | * | @ | * | @ | * | @  |
 ……
No.  7: ……
 | @ | * |  | * | @ | * | @  |
 ……
No.  8: ……
 | @ | * | @ | * |  | * | @  |
 ……
No.  9: ……
 | @ | * | @ | * | @ | * |  |
 ……
No. 10: ……
 | @ | * | @ | * | @ |  | *  |
 ……
No. 11: ……
 | @ | * | @ |  | @ | * | *  |
 ……
No. 12: ……
 | @ |  | @ | * | @ | * | *  |
 ……
No. 13: ……
 | @ | @ |  | * | @ | * | *  |
 ……
No. 14: ……
 | @ | @ | @ | * |  | * | *  |
 ……
No. 15: ……
 | @ | @ | @ |  | * | * | *  |
 ……

   Process finished with exit code 0
```

图7.3　运行结果

7.2　自动发牌

1. 问题描述

一副扑克有52张牌，打桥牌时应将牌分给4个人。请设计一个程序完成自动发牌的工作。要求：黑桃用S（Spaces）表示，红桃用H（Hearts）表示，方块用D（Diamonds）表示，梅花用C（Clubs）表示。

2. 问题分析

按照打桥牌的规定，每人应当有13张牌。在人工发牌时，先进行洗牌，然后将洗好的牌按一定的顺序发给每一个人。为了便于计算机模拟，可将人工方式的发牌过程加以修改，先确定好发牌顺序：1、2、3、4；将52张牌顺序编号：黑桃2对应数字0，红桃2对应数字1，方块2对应数字2，梅花2对应数字3，黑桃3对应数字4，红桃3对应数字5，……；然后从52张牌中随机为每个人抽牌。

这里采用Python语言中random模块库中的随机函数，生成0～51之间的共52个随机数，以产生洗牌后发牌的效果。有关随机函数的使用方法在第4章已经讲解过，在此不再陈述。

3. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 自动发牌


import random

def p(b, n):
    print("\n♠", end='')                          # 打印黑桃标记
    for i in range(13):                           # 将数组中的值转换为相应的花色
        if b[i] // 13 == 0:                       # 找到该花色对应的牌
            print(n[b[i] % 13], end=' ')

    print("\n♥ ", end='')                         # 打印红桃标记
```

```
    for i in range(13):
        if b[i] // 13 == 1:
            print(n[b[i] % 13], end=' ')

    print("\n♦ ", end='')                    # 打印方块标记
    for i in range(13):
        if b[i] // 13 == 2:
            print(n[b[i] % 13], end=' ')

    print("\n♣ ", end='')                    # 打印梅花标记
    for i in range(13):
        if b[i] // 13 == 3 or b[i] // 13 == 4:
            print(n[b[i] % 13], end=' ')

    print()


if __name__ == '__main__':
    n = ['2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J', 'Q', 'K', 'A']
    a = [0] * 53
    b1 = [0] * 13
    b2 = [0] * 13
    b3 = [0] * 13
    b4 = [0] * 13

    b11, b22, b33, b44, t = 0, 0, 0, 0, 1
    while (t <= 52):                                    # 控制发52张牌
        m = random.randint(0, 52)   # 产生0～51之间的随机数
        flag, i = True, 1
        while i <= t and flag:            # 查找新产生的随机数是否已经存在
            if m == a[i]:
                flag = 0                    # flag = 1表示产生的是新的随机数，flag
= 0
                                                                    表示新产生的随机数
已经存在

            i += 1

        if flag:
            a[t] = m                                    # 如果产生了新的随机数，则存入
数组
            t += 1
            # 根据t的模值，判断当前的牌应存入哪个数组中
            if t % 4 == 0:
                b1[b11] = a[t - 1]
                b11 += 1

            elif t % 4 == 1:
                b2[b22] = a[t - 1]
                b22 += 1

            elif t % 4 == 2:
                b3[b33] = a[t - 1]
                b33 += 1

            elif t % 4 == 3:
                b4[b44] = a[t - 1]
                b44 += 1


    b1 = sorted(b1, reverse=True)    # 将每个人的牌进行排序
    b2 = sorted(b2, reverse=True)
    b3 = sorted(b3, reverse=True)
    b4 = sorted(b4, reverse=True)

    p(b1, n)                                                # 分别打印每个人的牌
    p(b2, n)
    p(b3, n)
    p(b4, n)
```

4. 运行结果

在PyCharm下运行程序，结果如图7.4所示。

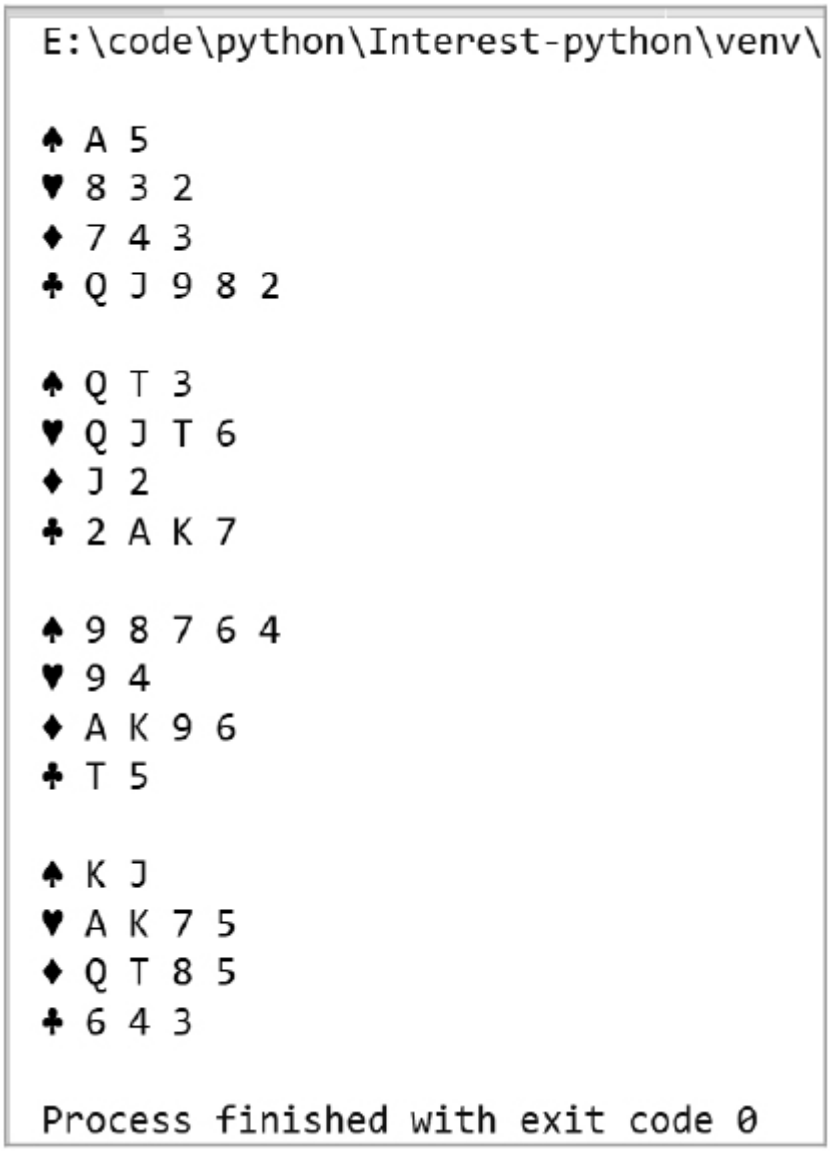
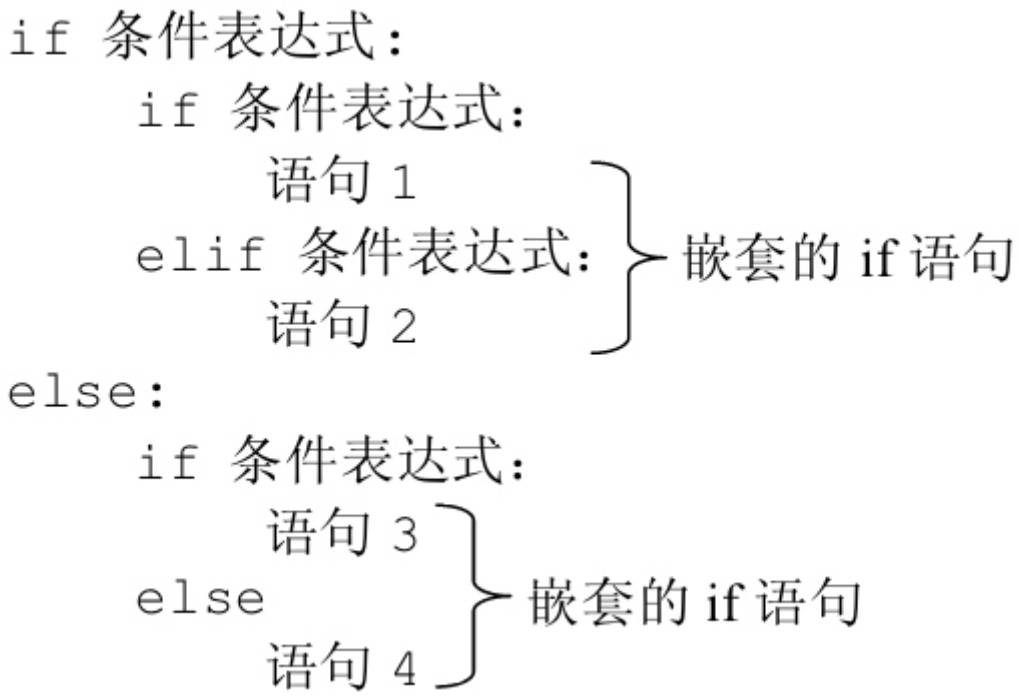


图7.4　运行结果

5. 问题拓展

在上面的程序中使用了大量嵌套的if…else…语句。在if语句中又包含一个或多个if语句称为if语句的嵌套。其一般形式如下：



在使用嵌套的if语句时要特别注意if与else的配对关系，原则就是else总是与它上面最近的且未配对的那个if相匹配。

如果一个if条件满足之后，还有其他多种情况需要判断，可以使用if…elif…elif…语句进行多次判断，如上述程序中的代码可修改如下：

```
if flag:
    a[t] = m                                                    # 如果产生了新的随机数，则存入
数组
    t += 1
    # 根据t的模值，判断当前的牌应存入哪个数组中
    if t % 4 == 0:
        b1[b11] = a[t - 1]
        b11 += 1

    elif t % 4 == 1:
        b2[b22] = a[t - 1]
        b22 += 1

    elif t % 4 == 2:
        b3[b33] = a[t - 1]
        b33 += 1

    elif t % 4 == 3:
        b4[b44] = a[t - 1]
        b44 += 1
```

7.3　常胜将军

1．问题描述

有火柴21根，两人依次取，每次每人只可取走1～4根，不能多取，也不能不取，谁取到最后一根火柴谁输。请编写一个人机对弈程序，要求人先取，计算机后取；计算机为“常胜将军”。

2．问题分析

可以这样思考这个问题：要想让计算机是“常胜将军”，也就是要让人取到最后一根火柴。这样只有一种可能，那就是让计算机只剩下1根火柴给人，因为此时人至少取1根火柴。别的情况都不能保证计算机常胜。

于是问题转换为“有20根火柴，两人轮流取，每人每次可以取走1～4根，不可多取，也不能不取，要求人先取，计算机后取，谁取到最后一根火柴谁赢”。为了计算机能够取到最后一根火柴，就要保证最后一轮抽取（人先取一次，计算机再取一次）之前剩下5棵火柴。因为只有这样才能保证无论人怎样取火柴，计算机都能将其余的火柴全部取走。

于是问题又转换为“15根火柴，两人轮流取，每人每次可以取走1～4根，不可多取，也不能不取，要求人先取，计算机后取，保证计算机取到最后一根火柴”。同样道理，为了让计算机取到最后一根火柴，就要保证最后一轮的抽取（人先取一次，计算机再取一次）之前剩下5根火柴。

于是问题又转化为10根火柴的问题……，依次递推。

3．算法设计

根据以上分析，可以得出这样的结论：21根火柴，在人先取计算机后取、每次取1～4根的前提下，只要保证每一轮的抽取（人先取一次，计算机再取一次）时人抽到的火柴数与计算机抽到的火柴数之和为5，就可以实现计算机的常胜不败。

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 常胜将军

if __name__ == "__main__":
    spare = 21                                                                    #
21根火柴
    print("---------你不能战胜我，不信试试---------")
    print("开始游戏: ")
    while 1:
        print("---------目前还有火柴 %d 根---------" %spare)
        people = int(input("人取火柴："))          # 人取火柴
        if people < 1 or people > 4 or people > spare:
            print("你违规了，你取的火柴数有问题！")
            continue
        spare = spare - people                  # 人取后，剩余的火柴数
        # 人取后，剩余的火柴数为0，则计算机
          获胜，跳出循环
        if spare == 0:
            print("计算机获胜，游戏结束！")
            break
        # 计算机取火柴
        computer = 5 - people
        spare = spare - computer
        print("计算机取火柴：%d" %computer)
        # 计算机取后，剩余的火柴数为0，则人
          获胜，跳出循环
        if spare == 0:
            print("人获胜，游戏结束！")
            break
```

5. 运行结果

在PyCharm下运行程序，结果如图7.5所示。


E:\code\python\Interest-python\venv\
--------你不能战胜我，不信试试--------
开始游戏:
--------目前还有火柴 21 根--------
人取火柴：1
计算机取火柴：4
--------目前还有火柴 16 根--------
人取火柴：3
计算机取火柴：2
--------目前还有火柴 11 根--------
人取火柴：2
计算机取火柴：3
--------目前还有火柴 6 根--------
人取火柴：4
计算机取火柴：1
--------目前还有火柴 1 根--------
人取火柴：1
计算机获胜，游戏结束！
Process finished with exit code 0

图7.5　运行结果

7.4　人机猜数

1．问题描述

由计算机随机产生一个4位整数，请人猜这4位整数是多少。人输入一个4位数后，计算机首先判断其中有几位猜对了，并且对的数字中有几位位置也正确，将结果显示出来，给人以提示，请人再猜，直到人猜出计算机随机产生的4位数是多少为止。

例如，计算机产生一个4位整数“1234”请人猜，可能的提示如下：

人猜的整数	计算机判断有几个数字正确	有几个位置正确
1122	2A	1B
3344	2A	1B
3312	3A	0B
4123	4A	0B
1243	4A	2B
1234	4A	4B

游戏结束。请编程实现该游戏。

2．问题分析

判断相同位置上的数字是否相同不需要特殊的算法。只要截取相同位置上的数字进行比较即可。但在判断几位数字正确时，则应当注意：计算机随机产生的是“1123”，而人所猜的是“1576”，则正确的数字只有1位。

程序中截取计算机随机产生的数的每位数字与人所猜的数字按位比较。若有两位数字相同，则要记住所猜中数字的位置，使该位数字只能与一位对应的数字“相同”。当截取下一位数字进行比较

时，就不应再与上述位置上的数字进行比较，以避免所猜的数中的一位与对应数中多位数字“相同”的错误情况。

3．随机数

随机数，顾名思义就是随机产生的、无规则的数。在编程中，有时我们不想手动从键盘输入数据，而想让计算机自动产生一些数据供我们使用（例如生成100个两位数），这时就要用到随机数。

随机数的生成方法很简单，在Python中只需导入random模块即可，random模块提供了生成随机数的相关方法，在第4章中已对random模块做了详细的讲解，在此不再陈述。

4．程序框架

程序流程图如图7.6所示。

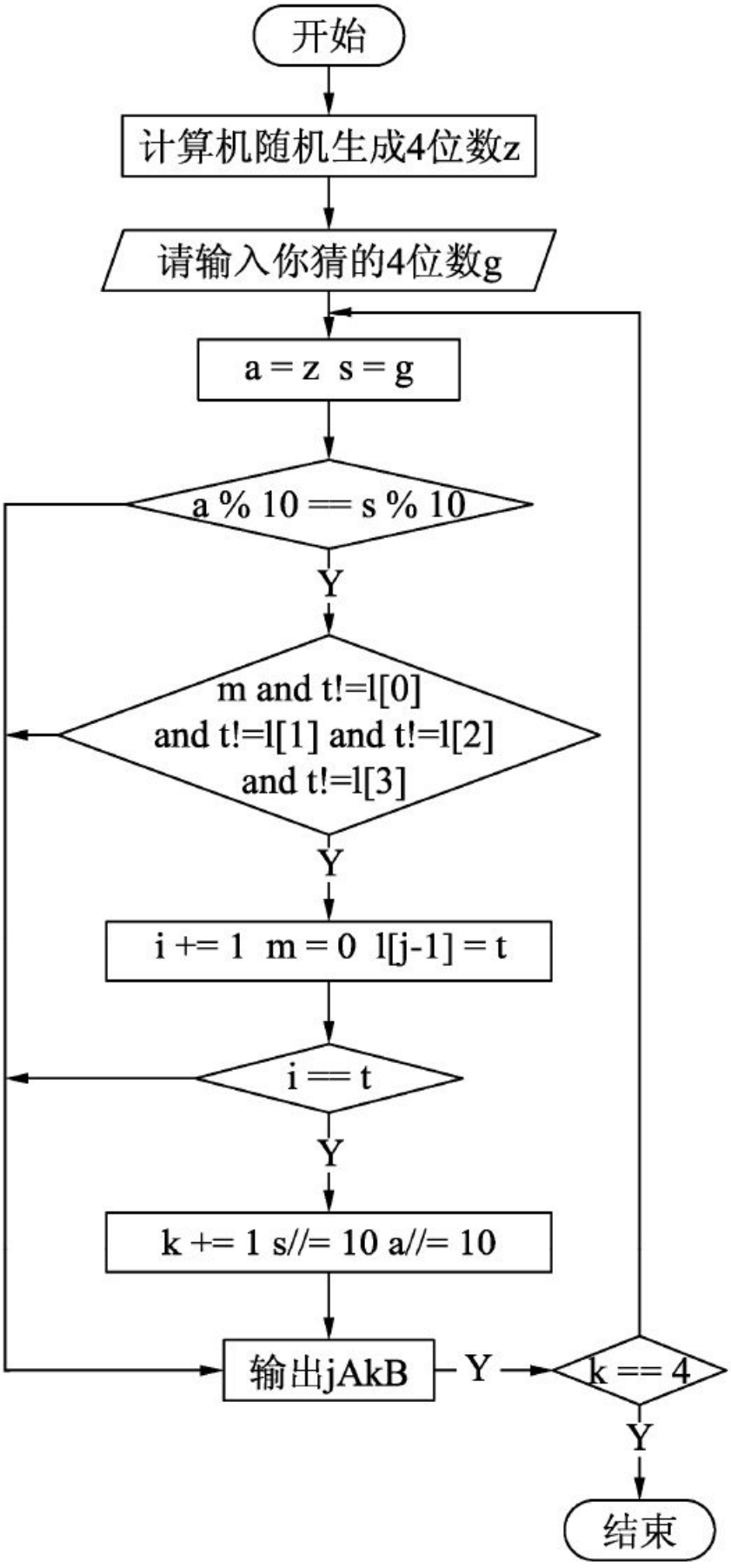

开始
计算机随机生成4位数z
请输入你猜的4位数g
a = z s = g
a % 10 == s % 10
Y
m and t!=l[0]
and t!=l[1] and t!=l[2]
and t!=l[3]
Y
i += 1 m = 0 l[j-1] = t
i == t
Y
k += 1 s//= 10 a//= 10
输出jAkB
Y
k == 4
Y
结束

图7.6　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 人机猜数

import random


if __name__ == "__main__":
    z = random.randint(1000, 10000) # 随机生成[1000,10000)之间的数
    # print("z = ", z)
    print("机器输入四位数：****")
    l = [0] * 4                                                    # 存储4位随机数

    c = 1                                                          # 变量c为猜数次数计算
器
    while True:
        g = int(input("请输入你猜的四位数："))
        a = z
        j = 0                                                      # 变量j表示数字正确的
位数
        k = 0                                                      # 变量k表示位置正确的
位数
        l[0] = l[1] = l[2] = l[3] = 0
        # 变量i表示原数中的第i位数。个位为第1位，千位为第4位
        for i in range(1, 5):
            s = g
            m = 1
            for t in range(1, 5):    # 人所猜想的数
                if a % 10 == s % 10:  # 若第i位与人猜的第t位相同
                    # 若该位置上的数字尚未与其他数字"相同"
                    if m and t != l[0] and t != l[1] and t != l[2] and t !=
l[3]:
                        j += 1
                        m = 0
                        l[j-1] = t           # 记录相同数字时，该数字在所猜数字中的位
置
                    if i == t:
                        k += 1               # 若位置也相同，则计数器k加1
                s //= 10
            a //= 10
        print("你猜的结果是：%dA%dB\n" %(j, k))
        if k == 4:
            print("****** 你赢了 ******")
            break                                                  # 若位置全部正确，则人
猜对了，退出
        c += 1
    print("你总共猜了 %d 次" %c)
```

6．运行结果

在PyCharm下运行程序，先由计算机随机产生一个4位数，接着提示“请输入你猜的四位数：”，输入猜的4位数后，计算机会给出

一个判断结果，接着再猜，直至猜对为止，运行结果如图7.7所示。

```
E:\code\python\Interest-python\venv\Scripts\
机器输入四位数： ****
请输入你猜的四位数：1234
你猜的结果是：3A1B

请输入你猜的四位数：4567
你猜的结果是：1A0B

请输入你猜的四位数：5678
你猜的结果是：0A0B

请输入你猜的四位数：1290
你猜的结果是：3A1B

请输入你猜的四位数：1240
你猜的结果是：3A2B

请输入你猜的四位数：1249
你猜的结果是：4A2B

请输入你猜的四位数：1429
你猜的结果是：4A1B

请输入你猜的四位数：1294
你猜的结果是：4A1B

请输入你猜的四位数：1942
你猜的结果是：4A4B

****** 你赢了 ******
你总共猜了 9 次

Process finished with exit code 0
```

图7.7　运行结果

7．拓展训练

在一个3×3的棋盘上，甲乙两个人进行对弈。已知两人轮流在棋盘上放棋子，当某一方的棋子连成一条直线时（包括横线、竖线或斜线），则该方获胜。请编写游戏程序来实现人机之间的比赛，并输出比赛结果。比赛有3种结果，分别是输、赢和平局。

7.5　搬山游戏

1．问题描述

设有n座山，计算机与人作为比赛的双方，轮流搬山。规定每次搬山数不能超过k座，谁搬最后一座谁输。游戏开始时，计算机请人输入山的总数n和每次允许搬山的最大数k，然后请人开始，等人输入了需要搬走的山的数目后，计算机马上打印出它搬多少座山，并提示尚余多少座山。双方轮流搬山直到最后一座山搬完为止。计算机会显示谁是赢家，并问人是否要继续比赛。如果人不想玩了，计算机便会统计出共玩了几局，双方胜负如何。

2．问题分析

程序结构：

程序中先输入山的座数和允许每次搬山的最大数，从而找出最佳的搬山座数以获得游戏的胜利。

程序在若干次游戏结束后还记录了计算机跟人的胜负次数。程序中应用了条件语句、循环语句、逻辑判断语句来实现功能。

函数实现：

在有n座山的情况下，计算机为了将最后一座山留给人，而且又要控制每次搬山的数目不超过最大数k，应搬山的数目要满足下列关系：(n-1)%(k+1)。

3．算法设计

计算机参加游戏时应遵循下列原则：

1）当（剩余山的数目-1）≤可移动的最大数k时，计算机要移（剩余山数目-1）座，以便将最后一座山留给人。

2）对于任意正整数x、y，一定有：0≤x%(y+1)≤y。

在有n座山的情况下，计算机为了将最后一座山留给人，而且又要控制每次搬山的数目不超过最大数k，则它应搬山的数目要满足关系：(n-1)%(k+1)。

如果算出结果为0，即整除无余数，则规定只搬1座山，以防止冒进后发生问题。

4. 确定程序框架

程序的简单流程图如图7.8所示。

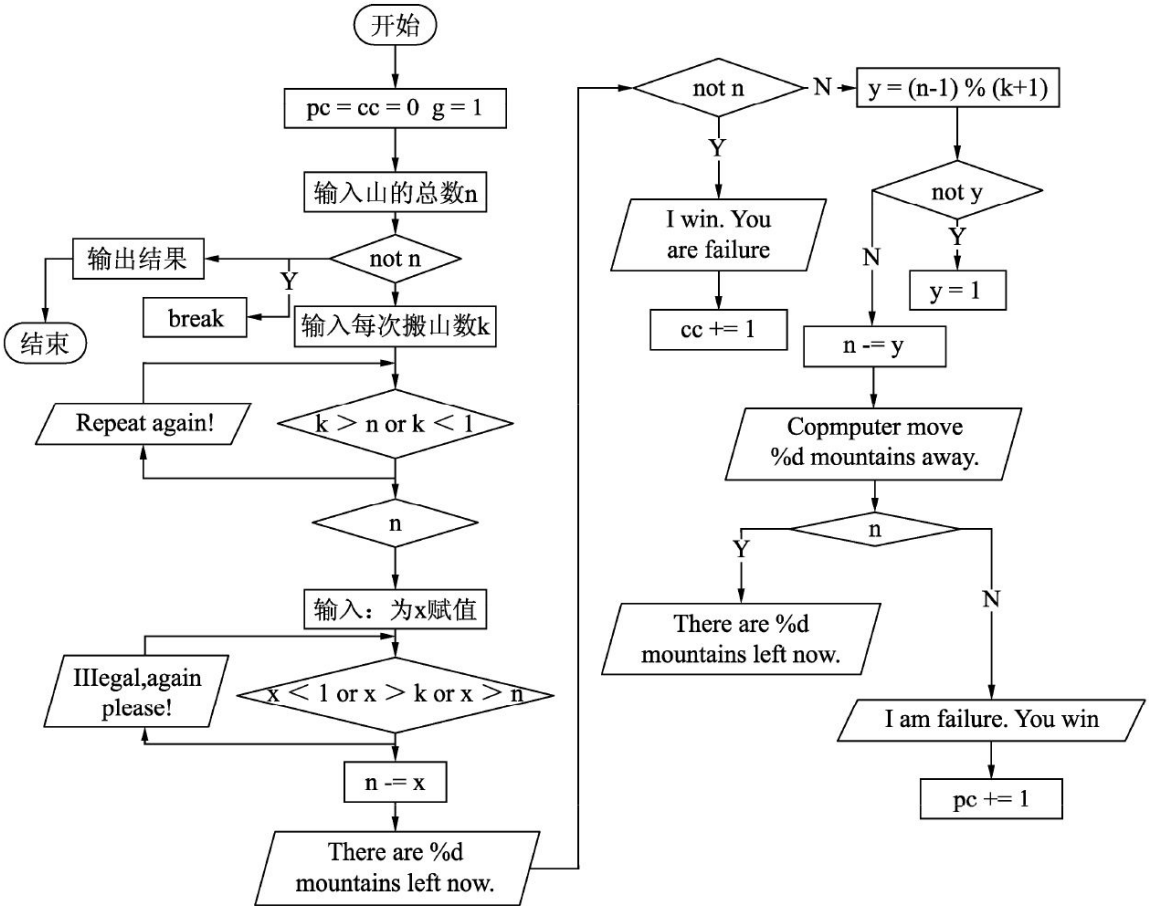


图7.8　程序流程图

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 搬山游戏


if __name__ == "__main__":
    print("More Mountain Game")
    print("Game Begin")
    pc = cc = 0
    g = 1
    while True:
        print("No.%2d game " %g)
        print("----------------------")
        print("How many mountains are there? ", end="")
        n = int(input())                                          # 输入山的总数
        if not n:
            break

        # 允许的最大搬山数
        k = int(input("How many mountains are allowed to each time? "))
        while(k > n or k < 1):
            if k > n or k < 1:
               print("Repeat again!")

        while n:
            x = int(input("How many mountains do you wish move away? "))
            if x < 1 or x > k or x > n:                  # 判断搬山数是否符合要求
                print("IIIegal,again please!")
                continue;
            n -= x
            print("There are %d mountains left now." %n)
            if not n:
                print("……I win. You are failure……")
                cc += 1
            else:
                y = (n-1) % (k+1)                                 # 求出最佳搬山数
                if not y:
                    y = 1
                n -= y
                print("Copmputer move %d mountains away." %y)
                if n:
                    print(" There are %d mountains left now."  %n)
                else:
                    print("……I am failure. You win……")
                    pc += 1
        g += 1

    # 打印结果
    print("Games in total have been played %d." %(cc + pc))
    print("You score is win %d,lose %d." %(pc, cc))
    print("My score is win %d,lose %d." %(cc, pc))
```

6. 运行结果

在PyCharm下运行程序，结果如图7.9所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe
More Mountain Game
Game Begin
No. 1 game
--------------------
How many mountains are there? 10
How many mountains are allowed to each time? 3
How many mountains do you wish move away? 3
There are 7 mountains left now.
Copmputer move 2 mountains away.
 There are 5 mountains left now.
How many mountains do you wish move away? 2
There are 3 mountains left now.
Copmputer move 2 mountains away.
 There are 1 mountains left now.
How many mountains do you wish move away? 1
There are 0 mountains left now.
……I win. You are failure……
No. 2 game
--------------------
How many mountains are there?
```

图7.9　运行结果

7.6　抢30游戏

1．问题描述

由两个人玩“抢30”游戏，游戏规则是：第一个人先说“1”或“1，2”，第二个人要接着往下说一个或两个数，然后又轮到第一个人，再接着往下说一个或两个数。这样两人反复轮流，每次每个人说一个或两个数都可以，但是不可以连说三个数，谁先抢到30，谁得胜。

2．问题分析

首先，分析这个游戏是否公平。一个游戏的公平性主要体现在游戏双方赢的机会性，如果机会性一样大，就公平，否则就不公平。那么，这个游戏中两人赢的机会性是不是一样大呢？

经分析可知，获胜者最后总能说到27，还有呢？获胜者陆续说出了24，21，18，15，12，9，6，3。因此，只要能控制讲出上述数，就一定能在最后“抢到30”。在大家不知情的情况下，不管先说后说，都有赢的可能性，但游戏里潜藏着人为可控的必胜因素。还可以发现，失败者报1个数，获胜者就报2个数；失败者报2个数，获胜者就只报1个数。所以获胜者总能迅速报数。

结论：如果有人利用数学规律掌握了必胜的秘诀，巧妙设计，就可以做到每战必赢，这个游戏其实是不公平的。

然后，探究只赢不输的奥秘。

规律1：使用逆推的方法

要想抢到30，必须先抢到27，这样，无论对方说28或“28，29”，自己总能抢到30。现在问题转换为如何抢到27。要想抢到27，必须先抢到24，这样，无论对方说25或“25，26”，自己总能抢到27……照此推理下去，要想抢到6，必须先要抢到3，这样无论

对方说4或“4，5”，自己总能抢到6。最后，问题转换为如何抢到3，要想抢到3，只有让对方先开始，这样无论对方先说1或“1，2”，自己总能抢到3。由此可见，这个游戏是偏向后开口的人，若这个人能抢到3，6，9，12，…，21，24，27，则一定会赢，因此，这个游戏是不公平的。

规律2：使用循环法

根据游戏规则，第一个人可以在1或“1，2”中选择一个或两个数字，对于“抢30”游戏，第二个人总是可以控制每轮报数的个数为3，由于30可以被3整除，因此第二个人可以控制自己最后说到30从而获胜。

如果把“抢30”游戏改成“抢50”，可类似地进行分析。要先抢到50，就要先抢到47，44，41，…，5，2。因此，我们发现，“抢50”的游戏是偏向先开口的人。

3．Python语言函数介绍

Python语言是函数的语言。对于函数，我们可以从以下几个方面去理解和把握。

（1）小函数大程序

意思是说，一个Python程序可以很大，但通常是由多个函数组成的。从这个意义上讲，函数往往就比较短小。

• 一个程序需要由几个函数来实现，这取决于用户对Python语言的掌握程度和领悟能力，没有硬性规定，以方便编程、方便调试和方便升级为原则。

• 一个程序分解成几个函数，有利于快速调试程序，也有利于提高程序代码的利用率。因为函数是可以多次被调用的，调用次数和调用场合没有限制。除main()函数以外，任何一个函数都可以调用另外一个函数。

• 不要指望通过一个函数就解决程序中的所有问题。事实上，每个函数都应该做自己最应该做的事情，即每个函数都应该具有相对独立的功能。

（2）main()函数及其作用

• 不管是规模很大的Python语言程序，还是比较短小的Python语言程序，永远都只能有一个而且只能有一个main()函数。

• main()函数可以放在程序中的任何一个地方，可以在程序首部，也可以在程序中间，还可以在程序尾部。

• 在Python语言程序中，不管main()函数放在程序的什么地方，都一定是从main()函数开始执行程序的，并且从main()函数结束程序。所以，main()函数又被称为主函数，它是Python程序执行的入口。

（3）函数的种类

函数通常分为模块库函数和自定义函数（用户函数）两大类。

• 模块库函数是相应的模块提供的函数，我们可以直接调用。

• 自定义函数是指由编程者自己编写的、用以实现特定功能的函数。所谓编写Python程序或开发Python程序，在很大程度上就是指编写若干个自定义函数（包括main()函数）。

（4）函数的定义、调用和说明

自定义函数时会涉及3个方面的问题：这个函数的功能及实现方法、如何来调用及调用前必要的准备。这3个方面分别对应着Python语言中的3个概念：函数定义、函数调用和函数说明。函数定义最为关键，因为函数必须要先定义才能使用。

简单地归纳函数定义的语法如下：

```
def 函数名（函数的参数）:
    函数体（即函数的具体程序，由若干条语句组成）
```

更多关于Python函数的知识点，在前面的章节中就已经说过，在此不再细说。

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 抢30

import random

# 定义输入函数
def input_num(t):
    a = int(input("Please count:"))
    while a > 2 or a < 1 or a + t > 30:

        if a > 2 or a < 1 or a + t > 30:
            print("Error input, again!")
        else:
            print("You count: %d" % (t + a))
        a = int(input("Please count:"))
    print("You count: %d" % (t + a))
    return t + a                                        # 返回当前已经取走的数的累加和


# 计算谁会胜利
def copu(s):
    c = 0
    print("Computer count: ", end="")
    if (s + 1) % 3 == 0:                                # 若剩余的数的模为1，则取1
        s = s + 1
        print(s)
    else:
        if (s + 2) % 3 == 0:
            s = s + 2                                   # 若剩余的数的模为2，则取2
            print(s)
        else:
            c = random.randint(1, 2)                    # 否则随机取1或2
            s = s + c
            print(s)
    return s


if __name__ == "__main__":
    tol = 0
    print("***********catch thirty **************")
    print("Game Begin")
    # 取随机数决定机器和人谁先走第一步。若为1，则表示人先走第一步
    if (random.randint(1, 2) == 1):
        tol = input_num(tol)
    while tol != 30:                                    # 游戏结束条件
        tol = copu(tol)
        if tol == 30:                                   # 计算机取一个数，若为30则机器胜利
            print("Computer: I lose !")
```

```
        else:
            tol = input_num(tol)
            if tol == 30:                                    # 人取一个数，若为30则人胜利
                print("People: I lose !")
    print("************Game Over*********************")
```

5. 运行结果

在PyCharm下运行程序，结果如图7.10所示。

6. 拓展训练

已知桌上有25颗棋子，现在要求游戏双方轮流取棋子，每人每次至少取走一颗棋子，最多可取走3颗棋子。照此取下去，直到将所有棋子取完，此时必然有一方手中的棋子数为偶数，而另一方手中的棋子数为奇数，规定偶数方获胜。编程实现此人机游戏。

```
E:\code\python\Interest-python\venv\Scripts\python.exe
***********catch thirty **************
Game Begin
Please count:1
You count: 1
Computer count: 3
Please count:2
You count: 5
Computer count: 6
Please count:1
You count: 7
Computer count: 9
Please count:2
You count: 11
Computer count: 12
Please count:1
You count: 13
Computer count: 15
Please count:2
You count: 17
Computer count: 18
Please count:1
You count: 19
Computer count: 21
Please count:2
You count: 23
Computer count: 24
Please count:2
You count: 26
Computer count: 27
Please count:1
You count: 28
Computer count: 30
Computer: I lose !
************Game Over*********************
```

图7.10　运行结果

7.7　24点游戏

1. 问题描述

在屏幕上输入1～10范围内的4个整数（可以有重复），对它们进行加、减、乘、除四则运算后（可以任意的加括号限定计算的优先级），寻找计算结果等于24的表达式。

例如输入4个整数4、5、6、7，可得到表达式：4*((5-6)+7)=24。这只是一个解，本题目要求输出全部的解。要求表达式中数字的顺序不能改变。

2. 问题分析

本题最简便的解法是应用穷举法搜索整个解空间，筛选出符合题目要求的全部解。因此，关键的问题是如何确定该题的解空间。

假设输入的4个整数为A、B、C、D，如果不考虑括号优先级的情况，仅用四则运算符将它们连接起来，即A+B*C/D…，则可以形成4^3=64种可能的表达式。如果考虑加括号的情况，而暂不考虑运算符，则共有以下5种可能的情况。

- ((A□B)□C)□D；
- (A□(B□C))□D；
- A□(B□(C□D))；
- A□((B□C)□D)；
- (A□B)□(C□D)。

其中，□代表+、-、*、/ 4种运算符中的任意一种。将上面两种情况综合起来考虑，每输入4个整数，其构成的解空间为64*5=320种表达式。也就是说，每输入4个整数，无论以什么方式或优先级进

行四则运算，其结果都会在这320种答案之中。我们的任务就是在这320种表达式中寻找出计算结果为24的表达式。

3．算法设计

以上是应用穷举法解决本题目的基本思想。下面介绍具体的实施办法。

首先将3个不同位置上的运算符设置成不同的变量：op1、op2、op3，并规定op1为整数A与B之间的运算符；op2为整数B与C之间的运算符；op3为整数C与D之间的运算符。

```
A op1 B op2 C op3 D
```

又规定变量op1、op2、op3的取值范围为1、2、3、4，分别表示加、减、乘，除4种运算，如表7.1所示。

表7.1　变量op1、op2、op3与运算符的对应情况

op1、op2、op3 变量值	表示的运算	op1、op2、op3 变量值	表示的运算
1	+	3	*
2	-	4	/

这样通过一个三重循环就可以枚举出不考虑括号情况的64种表达式类型。算法如下：

```
for op1 in range(1, 5):
    for op2 in range(1, 5):
        for op3 in range(1, 5):
            # 得到一种不含括号的表达式情形：A op1 B op2 C op3 D
```

下面的问题就是考虑如何在表达式中添加括号，以及如何通过每种表达式的状态计算出对应的表达式的值。我们分别来介绍。

首先，上述算法得到的每一种表达式都可能具有5种添加括号的方式，而这5种添加括号的方式实际上涵盖了该表达式的所有可能优先级的运算。例如，表达式A+B-C*D的5种添加括号的方式为：

- ((A+B)-C)*D;
- (A+(B-C))*D;
- A+(B-(C*D));
- A+((B-C)*D);
- (A+B)-(C*D)。

实际上，对表达式A+B-C*D以任何优先级方式运算，都包含在这5种表达式之中。

例如，不添加任何括号的表达式A+B-C*D等价于表达式(A+B)-(C*D)，表达式A+(B-C)*D等价于表达式A+((B-C)*D)。也就是说，上述5种表达式覆盖了+、-、*、/这4种运算符组合所构成的表达式的全部解（以任何优先级进行计算）。因此在程序设计时就应当针对这5种添加括号的方式，分别计算出每一种表达式的值，看它是否等于24。

由于每一种表达式的计算顺序（优先级）不尽相同，为了避免表达式运算的麻烦，可以设置5个函数，分别对应每一种类型表达式的计算。算法如下：

```
# 对应的表达式类型：((A□B)□C)□D
def calculate_model1(i, j, k, t, op1, op2, op3):
    r1 = cal(i, j, op1)
    r2 = cal(r1, k, op2)
    r3 = cal(r2, t, op3)
    return r3


# 对应的表达式类型：(A□(B□C))□D
def calculate_model2(i, j, k, t, op1, op2, op3):
    r1 = cal(j, k, op2)
    r2 = cal(i, r1, op1)
    r3 = cal(r2, t, op3)
    return r3


# 对应的表达式类型：A□(B□(C□D))
def calculate_model3(i, j, k, t, op1, op2, op3):
    r1 = cal(k, t, op3)
    r2 = cal(j, r1, op2)
    r3 = cal(i, r2, op1)
    return r3
```

```
# 对应的表达式类型：A□((B□C)□D)
def calculate_model4(i, j, k, t, op1, op2, op3):
    r1 = cal(j, k, op2)
    r2 = cal(r1, t, op3)
    r3 = cal(i, r2, op1)
    return r3


# 对应的表达式类型：(A□B)□(C□D)
def calculate_model5(i, j, k, t, op1, op2, op3):
    r1 = cal(i, j, op1)
    r2 = cal(k, t, op3)
    r3 = cal(r1, r2, op2)
    return r3
```

上述算法中，每一个函数对应一种表达式类型，其返回值为表达式的值，原表达式的形式为i op1 j op2 k op3 t，不同的表达式类型，根据其运算时优先级的不同进行不同的运算。其中函数cal()的作用是通过每种表达式的状态计算出对应的表达式的值。

函数cal()包括3个参数，第3个参数为运算符变量，它标志着不同种类的运算符（如表7.1所示），前两个参数为运算数。该函数的作用是使用前两个参数指定的操作数进行第3个参数指定的运算，并返回其运算结果。例如调用cal(2,3,1)的作用是计算2+3=5，并返回5。函数cal()的代码如下：

```
def cal(x, y, op):
    if op == 1:
        return x + y                    # op等于1，加法运算
    elif op == 2:
        return x - y                    # op等于2，减法运算
    elif op == 3:
        return x * y                    # op等于3，乘法运算
    elif op == 4:
        return x / y                    # op等于4，除法运算
```

将上面所讲的算法结合起来，就可以遍历由4个操作数（范围：1～10）、3个运算符（取值：+、-、*、/）、以任何优先级进行运算所构成的320种表达式，从而寻找其中答案为24的表达式。

如果更加细致深入地思考本题，本题的算法还可以改善。其实由4个整数和4种运算符（+、-、*、/）以不同的优先级运算方式构成的表达式并不一定就是320种，有些表达式可能是完全等价的，例如

((A+B)-C)*D和(A+(B-C))*D。这两个表达式是完全等价的，无论A、B、C、D为何值，它们的计算结果都是一样的。因此可以将这两个表达式合并成为一个表达式：(A+B-C)*D。这样就缩小了搜索的空间，提高了查询的效率。但是这样做要考虑的问题就会多一些，有兴趣的读者可以自己尝试完成。

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 24点

op = ['#', '+', '-', '*', '/']


def cal(x, y, op):
    if op == 1:
        return x + y                                    # op等于1，加法运算
    elif op == 2:
        return x - y                                    # op等于2，减法运算
    elif op == 3:
        return x * y                                    # op等于3，乘法运算
    elif op == 4:
        return x / y                                    # op等于4，除法运算


# 对应的表达式类型：((A□B)□C)□D
def calculate_model1(i, j, k, t, op1, op2, op3):
    r1 = cal(i, j, op1)
    r2 = cal(r1, k, op2)
    r3 = cal(r2, t, op3)
    return r3


# 对应的表达式类型：(A□(B□C))□D
def calculate_model2(i, j, k, t, op1, op2, op3):
    r1 = cal(j, k, op2)
    r2 = cal(i, r1, op1)
    r3 = cal(r2, t, op3)
    return r3


# 对应的表达式类型：A□(B□(C□D))
def calculate_model3(i, j, k, t, op1, op2, op3):
    r1 = cal(k, t, op3)
    r2 = cal(j, r1, op2)
    r3 = cal(i, r2, op1)
    return r3


# 对应的表达式类型：A□((B□C)□D)
def calculate_model4(i, j, k, t, op1, op2, op3):
    r1 = cal(j, k, op2)
    r2 = cal(r1, t, op3)
    r3 = cal(i, r2, op1)
    return r3
```

```
# 对应的表达式类型：(A□B)□(C□D)
def calculate_model5(i, j, k, t, op1, op2, op3):
    r1 = cal(i, j, op1)
    r2 = cal(k, t, op3)
    r3 = cal(r1, r2, op2)
    return r3


# 函数get24()用于寻找符合要求（计算结果为24）的表达式
def get24(i, j, k, t):
    flag = False
    for op1 in range(1, 5):
        for op2 in range(1, 5):
            for op3 in range(1, 5):
                # 找到((A□B)□C)□D类型的表达式中符合要求的表达式
                if calculate_model1(i, j, k, t, op1, op2, op3) == 24:
                    print("((%d%c%d)%c%d)%c%d=24" % (i, op[op1], j, op[op2],
k, op[op3], t))
                    flag = True

                # 找到(A□(B□C))□D类型的表达式中符合要求的表达式
                if calculate_model2(i, j, k, t, op1, op2, op3) == 24:
                    print("(%d%c(%d%c%d))%c%d=24" % (i, op[op1], j, op[op2],
k, op[op3], t))
                    flag = True

                # 找到A□(B□(C□D))类型的表达式中符合要求的表达式
                if calculate_model3(i, j, k, t, op1, op2, op3) == 24:
                    print("%d%c(%d%c(%d%c%d))=24" % (i, op[op1], j, op[op2],
k, op[op3], t))
                    flag = True

                # 找到A□((B□C)□D)类型的表达式中符合要求的表达式
                if calculate_model4(i, j, k, t, op1, op2, op3) == 24:
                    print("%d%c((%d%c%d)%c%d)=24" % (i, op[op1], j, op[op2],
k, op[op3], t))
                    flag = True

                # 找到(A□B)□(C□D)类型的表达式中符合要求的表达式
                if (calculate_model5(i, j, k, t, op1, op2, op3) == 24):
                    print("(%d%c%d)%c(%d%c%d)=24" % (i, op[op1], j, op[op2],
k, op[op3], t))
                    flag = True

    return flag

if __name__ == '__main__':
    print("Please input four integer (1～10)")
    i, j, k, t = map(int, input().split())
    # 若输入数值不合法，重新输入
    if i < 1 or i > 10 or j < 1 or j > 10 or k < 1 or k > 10 or t < 1 or t >
10:
        print("Input illege, Please input again")
        i, j, k, t = map(int, input().split())

    if get24(i, j, k, t):                                # 找到符合要求的表达式
        pass
    else:
        print("Sorry, the four integer cannot be calculated to get 24")
```

5．运行结果

本程序中，从主函数输入4个1～10的整数，调用函数get24()寻找符合要求（计算结果为24）的表达式。如果找到结果等于24的表

达式，函数get24()将其输出，并返回1；如果没有找到结果等于24的表达式，函数get24()返回0。

本程序的运行结果如图7.11和图7.12所示。

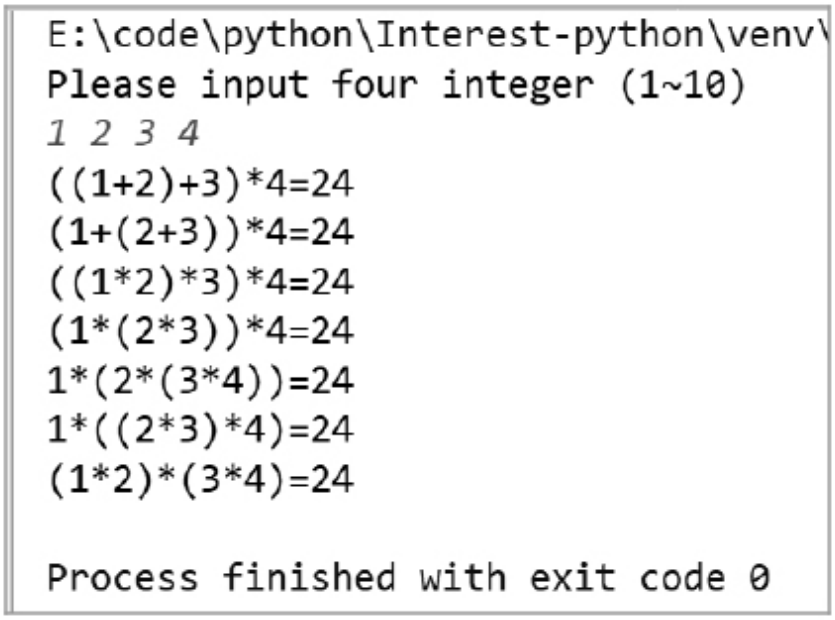


图7.11　运行结果1（找到结果等于24的表达式）

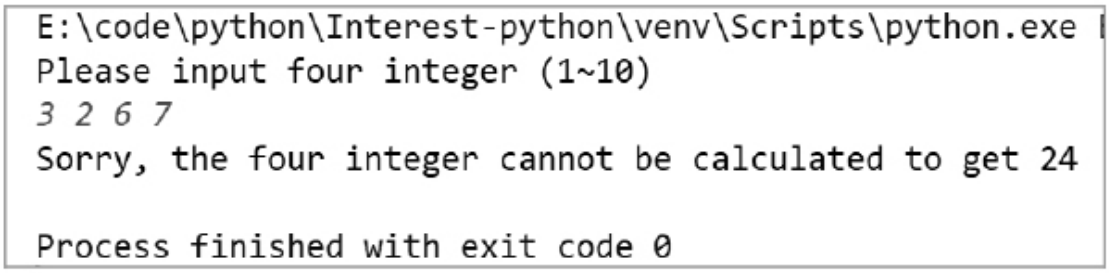


图7.12　运行结果2（未找到结果等于24的表达式）

7.8　掷骰子

1. 问题描述

骰子是一个有6个面的正方体，每个面分别印有1～6个小圆点代表点数。假设这个游戏的规则是两个人轮流掷骰子6次，并将每次投掷的点数累加起来，点数多者获胜，点数相同则为平局。

要求编写程序模拟这个游戏的过程，并求出玩100盘之后谁是最终的获胜者。

2. 问题分析

由于每个人掷骰子所得到的点数是随机的，所以需要借助随机数发生器，每次产生一个1～6之间的整数，以此模拟玩者掷骰子的点数。

要得到6个不同的随机值，只需要调用Python的random模块提供的random.randint()函数即可，这里为random.randint(1,6)。

为了计算在每盘中甲、乙两人所掷的点数，需要定义两个变量d1和d2，用于记录每个人投掷点数的累加和。

为了记录每个人的获胜盘数，需要再定义两个变量c1和c2，用于记录每个人获胜的盘数。

3. 程序框架

程序流程图如图7.13所示。

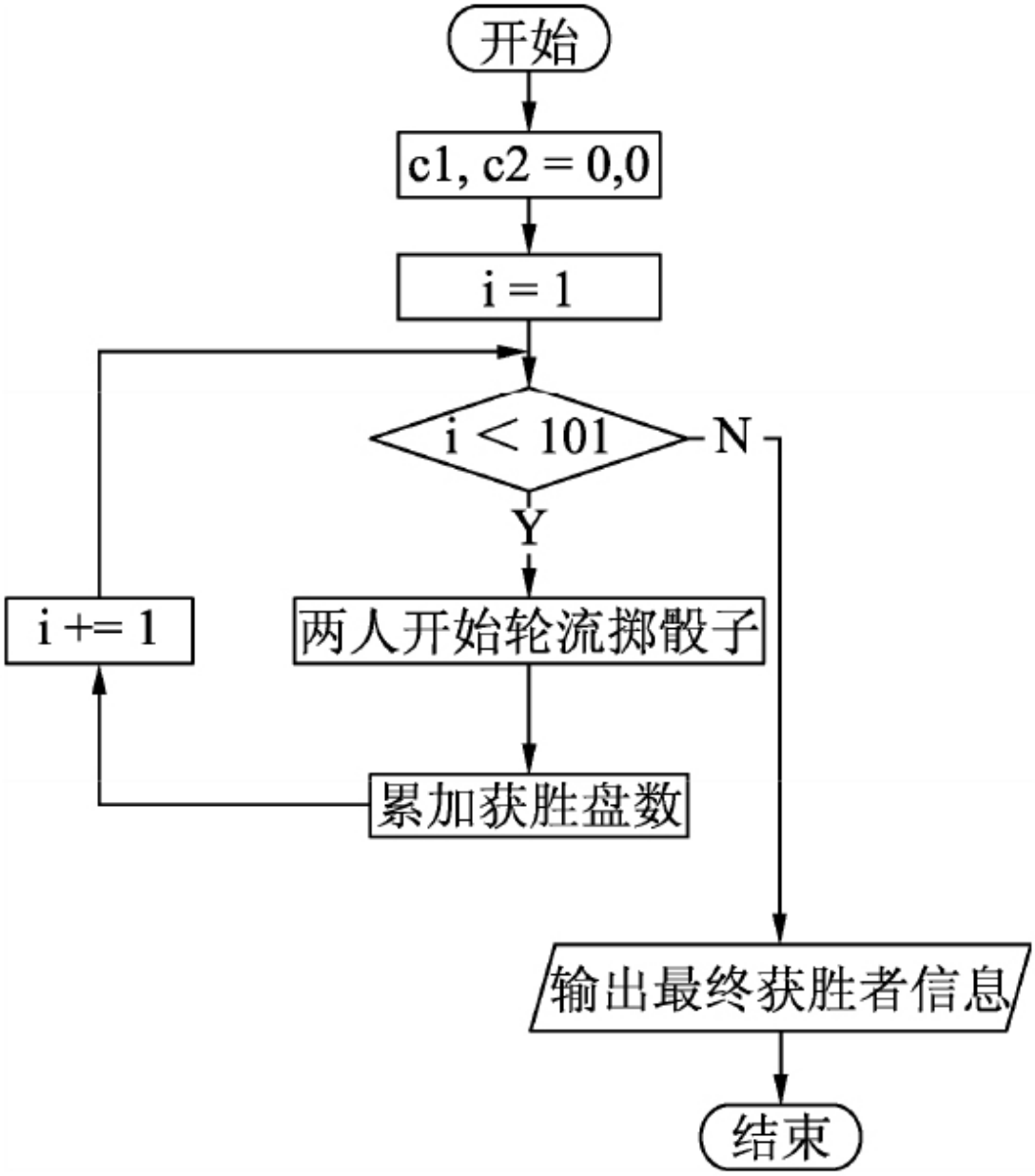


图7.13　程序流程图

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 掷骰子

import random
import time

if __name__ == "__main__":
```

```
    # c1和c2分别记录两个人的获胜盘数
    c1 , c2 = 0, 0
    print("两人开始掷骰子100次：")
    for i in range(1, 101): # 100盘
        d1 , d2 = 0, 0                                  # d1和d2分别记录两个人投掷点数
的累加和
        for j in range(1, 7):                                   # 两个人轮流掷骰子
            d1 = d1 + random.randint(1, 6)                      # 第一个人所掷骰子点数
总和
            d2 = d2 + random.randint(1, 6)                      # 第二个人所掷骰子点数
总和
        if d1 > d2:
            c1 += 1                                                          #
累加获胜盘数
        else:
            if d1 < d2:
                c2 += 1

    print("等待结果......")
    time.sleep(5)
    print("100次之后，获胜者是：")
    if c1 > c2:                                         # 输出最终获胜者信息
        print("第一个人获胜！")
    else:
        if c1 < c2:
            print("第二个人获胜！")
        else:
            print("两人平局！")
```

5. 运行结果

在PyCharm下运行程序，结果如图7.14所示。

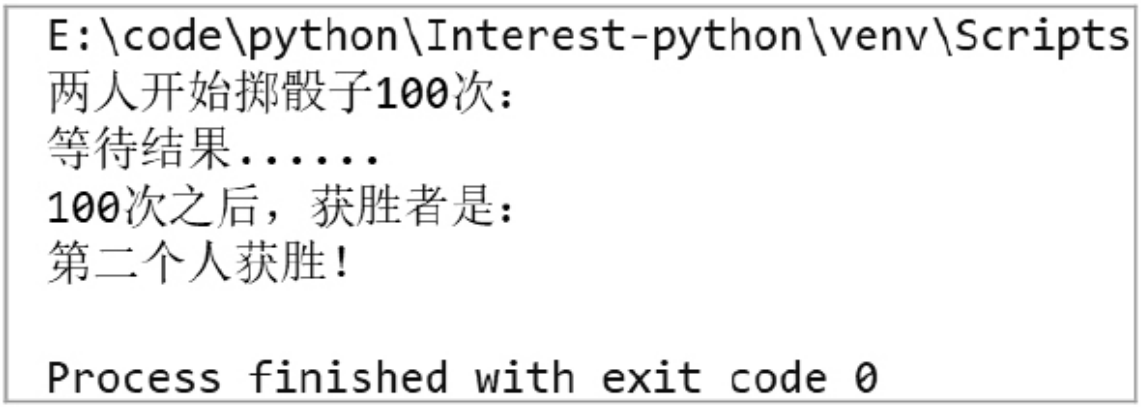


图7.14　运行结果

6. 补充知识点

在程序中，我们引入了一个新的模块time，并使用了time模块提供的sleep()函数，该函数是让程序休眠一段时间，这里是5秒。下面我们具体介绍time模块。

time模块提供了各种与时间相关的函数。下面介绍部分常用的函数。

（1）time.time()→float

time.time()函数返回浮点数的seconds since the epoch。epoch的具体日期和leap seconds的处理取决于平台。在Windows和大多数UNIX系统上，epoch是1970年1月1日00:00:00（UTC），并且闰秒不计入seconds since the epoch。这通常被称为UNIX time。

（2）time.time_ns()→int

time.time_ns()函数的用法与time()相似，但该函数返回是以纳秒为单位的自epoch以来的整数时间。

（3）time.thread_time()→float

time.thread_time()函数返回当前线程的系统和用户CPU时间之和的值（以小数秒为单位）。它不包括睡眠期间经过的时间。根据定义，它是特定于线程的。返回值的参考点未定义，因此只有同一线程中连续调用结果之间的差异才有效。

（4）time.thread_time_ns()→int

time.thread_time_ns()函数的用法与thread_time()相似，但该函数返回的是以纳秒为单位的时间。

（5）time.asctime([t])

time.asctime([t])函数用于转换一个元组或struct_time表示的时间，由gmtime()函数或localtime()函数返回的字符串为'Sun Jun 20 23:21:05 1993'。如果未提供t，则使用由localtime()函数返回的当前时间。区域信息不被函数asctime()使用。

（6）time.clock()

在UNIX平台上，time.clock()函数将当前处理器时间返回为以秒为单位的浮点数。

（7）time.pthread_getcpuclockid(thread_id)

time.pthread_getcpuclockid(thread_id)函数返回指定的thread_id的特定于线程的CPU时间时钟的clk_id。

（8）time.clock_getres(clk_id)

time.clock_getres(clk_id)函数返回指定时钟clk_id的分辨率（精度）。

（9）time.clock_gettime(clk_id)→float

time.clock_gettime(clk_id)函数返回指定时钟clk_id的时间。

（10）time.clock_gettime_ns(clk_id)→int

time.clock_gettime_ns(clk_id)函数的用法与clock_gettime()相似，但该函数是以纳秒的单位返回时间的。

（11）time.clock_settime(clk_id,time:float)

time.clock_settime(clk_id,time:float)函数设置指定时钟clk_id的时间。目前，CLOCK_REALTIME是clk_id唯一可接受的值。

（12）time.clock_settime_ns(clk_id,time:int)

time.clock_settime_ns(clk_id,time:int)函数的用法与clock_settime()相似，但该函数是以纳秒的形式设置时间的。

（13）time.ctime([secs])

time.ctime([secs])函数将自seconds since the epoch表示的时间转换为表示本地时间的字符串。如果未提供secs或为None，则

使用由time()返回的当前时间。ctime(secs)相当于asctime(localtime(secs))。区域信息不被ctime()使用。

（14）time.get_clock_info(name)

time.get_clock_info(name)函数获取有关指定时钟的信息作为命名空间对象。

（15）time.process_time()→float

time.process_time()函数返回当前进程的系统和用户CPU时间总和的值（以小数秒为单位）。

（16）time.process_time_ns()→int

time.process_time_ns()函数的用法与process_time()相似，但该函数是以纳秒的形式返回时间的。

（17）time.sleep(secs)

time.sleep(secs)函数用于暂停执行调用线程达到给定的秒数。参数可以是浮点数，以指示更精确的睡眠时间。

（18）time.strftime(format[,t])

time.strftime(format[,t])函数用于转换一个元组或struct_time表示的由gmtime()函数或localtime()函数返回的时间到由format参数指定的字符串。如果未提供t，则使用由localtime()函数返回的当前时间。format必须是一个字符串。如果t中的任何字段超出允许范围，则引发ValueError。

strftime("%a,%d,%b,%Y,%H:%M:%S+0000",gmtime())

其中0是时间元组中任何位置的合法参数；如果format参数通常是非法的，则该值被强制改为正确的值。

（19）time.strptime(string[,format])

time.strptime(string[,format])函数根据格式解析表示时间的字符串。返回值为一个被gmtime()函数或localtime()函数返回的struct_time。

format参数使用与strftime()函数相同的指令，默认为匹配ctime()函数返回格式的"%a %b %d %H:%M:%S %Y"。

更多关于time模块相关的知识点，请读者自行查阅学习。

第8章　趣味数组

数组是一些具有相同类型的数的集合，它是由某种类型的数据按照一定的顺序组成的，并用下标来指示数组中元素的序号。当处理大量同类型的数据时，使用数组非常方便。

本章的几个问题都与数组相关，通过对这几个问题的学习，读者可以明确数组的使用场合，熟练掌握数组的使用方法。在分析问题的时候，将Python语言中列表（数组）的语法知识贯穿其中进行讲解，读者可以根据自己的情况查漏补缺，起到复习和巩固的作用。

8.1　平分7筐鱼

1. 问题描述

甲、乙、丙三位渔夫出海打鱼，他们随船带了21只箩筐。当晚返航时，他们发现有7筐装满了鱼，还有7筐装了半筐鱼，另外7筐是空的，由于他们没有秤，只好通过目测认为7个满筐鱼的重量是相等的，7个半筐鱼的重量是相等的。在不将鱼倒出来的前提下，怎样将鱼和筐平分为三份？

2. 嵌套列表

如果一个列表的元素是列表，那么这个列表就是一个嵌套列表。一个简单的嵌套列表定义如下：

```
list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

使用for循环遍历这个列表，代码如下：

```
list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for i in list:
    print(i)
```

打印结果如图8.1所示。

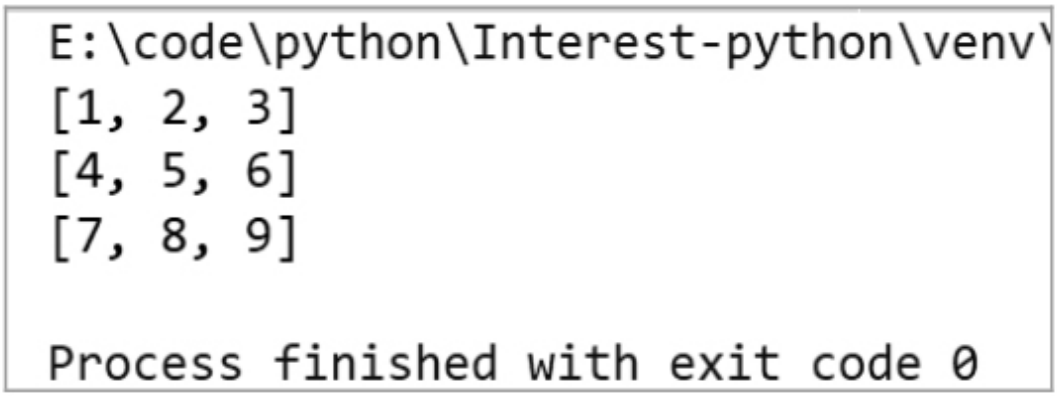


图8.1　打印结果

同样可以使用序号（下标）来访问这个嵌套列表的元素。可以把这个嵌套列表看作一个行列矩阵，行和列的下标都是从0开始的，例如list[1][2]表示的就是第一行第二列的元素，在这里这个元素值是6。

嵌套列表遍历之后还是列表，它一样具有单层列表的所有方法。

3. 问题分析

根据题意可以知道：每个人应分得7个箩筐，其中有3.5筐鱼。解决该问题可以采用一个3×3的数组，数组名为a，用来表示三个人分到的东西。其中每个人对应数组a的一行，数组的第0列放分到的鱼的整筐数，数组的第1列放分到的半筐数，数组的第2列放分到的空筐数。

又由题目可以推出：

1）数组的每行或每列的元素之和都为7。

2）对数组的行来说，满筐数加半筐数为3.5。

3）每个人所得的满筐数不能超过3筐。

4）每个人都必须至少有1个半筐，且半筐数一定为奇数。

对于找到的某种分鱼方案，三个人谁拿哪一份都是相同的，为了避免出现重复的分配方案，可以规定：第二个人的满筐数等于第一个人的满筐数；第二个人的半筐数大于等于第一个人的半筐数。

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 平分7筐鱼
```

```
if __name__ == '__main__':
    print("分鱼的方案如下：")
    count = 0
    a = [[0]*3 for i in range(3)]

    for i in range(4):                                  # 试探第一个人满筐a[0][0]的
值，满筐数不能>3
        a[0][0] = i
        j = i
        while j <= 7 - i and j <= 3:  # 试探第二个人满筐a[1][0]的值，满筐数不能>3
            a[1][0] = j
            a[2][0] = 7 - j - a[0][0]
            j += 1
            if a[2][0] > 3:
                continue                                # 第三个人满筐数不能>3
            if a[2][0] < a[1][0]:
                break       # 要求后一个人分的满筐数大于等于前一个人，以排除重复情况
            for k in range(1, 6, 2):        # 试探半筐a[0][1]的值，半筐数为奇数
                a[0][1] = k
                for m in range(1, 7 - k, 2):        # 试探半筐a[1][1]的值，半筐数
为奇数
                    a[1][1] = m
                    a[2][1] = 7 - k - m
                    # 判断每个人分到的鱼是否为3.5筐，flag为满足题意的标记变量
                    flag, n = True, 0
                    while flag and n < 3:
                        if a[n][0] + a[n][1] < 7 and a[n][0] * 2 + a[n][1] ==
7:
                            # 计算应得到的空筐数量
                            a[n][2] = 7 - a[n][0] - a[n][1]
                        else:
                            flag = False            # 不符合题意则置标记为0
                        n += 1

                    if flag:
                        count += 1
                        print('No.', count, ' Full basket Semi-basket Empty')
                        for n in range(3):
                            print('fisher ', chr(65 + n), ':', a[n][0], a[n]
[1],
a[n][2])
```

---

5. 运行结果

在PyCharm下运行程序，结果如图8.2所示。

```
E:\code\python\Interest-python\venv\Scripts
分鱼的方案如下：
No. 1  Full basket Semi-basket Empty
fisher  A : 1 5 1
fisher  B : 3 1 3
fisher  C : 3 1 3
No. 2  Full basket Semi-basket Empty
fisher  A : 2 3 2
fisher  B : 2 3 2
fisher  C : 3 1 3

Process finished with exit code 0
```

图8.2　运行结果

8.2　农夫过河

1. 问题描述

一个农夫在河边带了一匹狼、一只羊和一棵白菜，他需要把这三样东西用船带到河的对岸。然而，这艘船只能容下农夫本人和另外一样东西。如果农夫不在场的话，狼会吃掉羊，羊也会吃掉白菜。请编程为农夫解决这个过河问题。

2. 问题分析

根据问题描述可知，该问题涉及的对象较多，而且运算步骤也较为复杂，因此在使用Python语言实现时，首先需要将具体问题数字化。

由于整个过程的实现需要多步，而不同步骤中各个事物所处的位置不同，因此可以定义一个二维数组（嵌套列表）来表示4个对象——狼（wolf）、羊（goat）、白菜（cabbage）和农夫（farmer）。对于东岸和西岸，可以用east和west表示，也可以用0和1来表示，以保证程序设计时的简便性。

题目要求给出农夫、狼、羊和白菜的过河步骤，没有对先后顺序进行约束，这就需要给各个事物依次进行编号，然后依次试探，若试探成功，再进行下一步试探。因此，解决该问题可以使用循环或者递归算法，以避免随机盲目运算而且保证每种情况都可以试探到。

题目要求求出农夫带一只羊、一匹狼和一棵白菜过河的所有办法，所以依次成功返回运算结果后，需要继续运算，直至求出所有结果，即给出农夫不同的过河方案。

3. 算法设计

本程序使用递归算法，为了方便将各个实例数字化，定义二维数组int a[N][4]存储每一步中各个事物所处的位置。二维数组的一维下标表示当前进行的步骤，第二维下标可能的取值为0～3，在这里规定它与4种事物的具体对应关系为：0——狼、1——羊、2——白菜、3——农夫。接着再将东岸和西岸数字化，用0表示东岸，1表示西岸，该信息存储在二维数组的对应元素中。

初始情况下，当前步骤为0，此时狼、羊、白菜和农夫都在东岸，则使用a数组来表示该状态为：

a[0][0]	a[0][1]	a[0][2]	a[0][3]
0	0	0	0

假设在第3步之后狼在东岸，羊在西岸，白菜在东岸，农夫在西岸，则该步骤的存储状态为：

a[3][0]	a[3][1]	a[3][2]	a[3][3]
0	1	0	1

定义Step变量表示渡河的步骤，则成功渡河之后，a数组中的存储状态为：

a[Step][0]	a[Step][1]	a[Step][2]	a[Step][3]
1	1	1	1

因为成功渡河后，狼、羊、白菜和农夫都在河的西岸，因此有a[Step][0]+a[Step][1]+a[Step][2]+a[Step][3]=4。

题目中要求狼和羊、羊和白菜不能在一起，因此若有下述情况出现，则发生错误，应返回操作。

a[Step][1]!=a[Step][3] and (a[Step][2]==a[Step][1] or a[Step][0]==a[Step][1])

程序采用递归算法，主程序结构如图8.3所示。

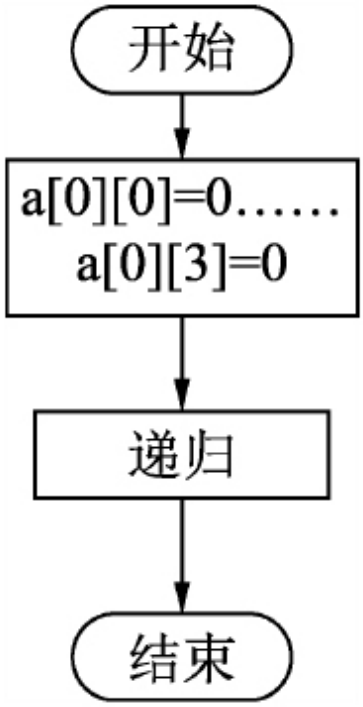


图8.3　主程序结构

在程序实现时，除了定义a数组来存储每一步中各个对象所处的位置以外，再定义一维数组b[N]来存储每一步中农夫是如何过河的。

程序中实现递归操作部分的核心代码为：

```
# 递归，从带第一种对象开始依次向下循环，同时限定递归的界限
for i in range(-1, 3):
    b[Step] = i                                    # 记录农夫渡河方式
    a[Step+1] = a[Step][:]  # 复制上一步状态，进行下一步移动
    # 农夫过去或者回来
    a[Step + 1][3] = 1 - a[Step + 1][3]

    if i == -1:
        search(Step + 1)                                        # 进行第一步

    elif a[Step][i] == a[Step][3]:                  # 若该物与农夫同岸，带回
        a[Step + 1][i] = a[Step + 1][3]             # 带回该物
        search(Step + 1)                                        # 进行下一步
```

每次循环从-1到2依次代表农夫渡河时为一人、带狼、带羊、带白菜通过，利用语句“b[Step]=i”分别记录每一步中农夫的渡河方式，语句“a[Step+1][i]=a[Step+1][3]”是利用赋值方式使该对象与农夫一同到对岸或者回到本岸。若渡河成功，则依次输出渡河方

式。“i<3”为递归操作的界限，若i=2时仍无符合条件的方式，则渡河失败。

在递归的过程中每进行一步都需要判断条件以决定是否继续进行此次操作，具体的判断代码为：

```
# 若该步骤能使各值均为1，则输出结果，进入回归步骤
if a[Step][0] + a[Step][1] + a[Step][2] + a[Step][3] == 4:
    ……
    return
```

上面的的代码表示若当前步骤能使各值均为1，则渡河成功，输出结果，进入回归步骤。

若当前步骤与以前的步骤相同，则返回操作，代码如下：

```
for i in range(Step):
    if a[i] == a[Step]:                    # 若该步与以前步骤相同，则返回操作
        return
```

若羊和农夫不在一块而狼和羊或者羊和白菜在一块，则返回操作，判断代码如下：

```
# 若羊和农夫不在一起而狼和羊或者羊和白菜在一起，则返回操作
if a[Step][1] != a[Step][3] and (a[Step][2] == a[Step][1] or a[Step][0] ==
a[Step][1]):
    return
```

递归部分程序结构如图8.4所示。

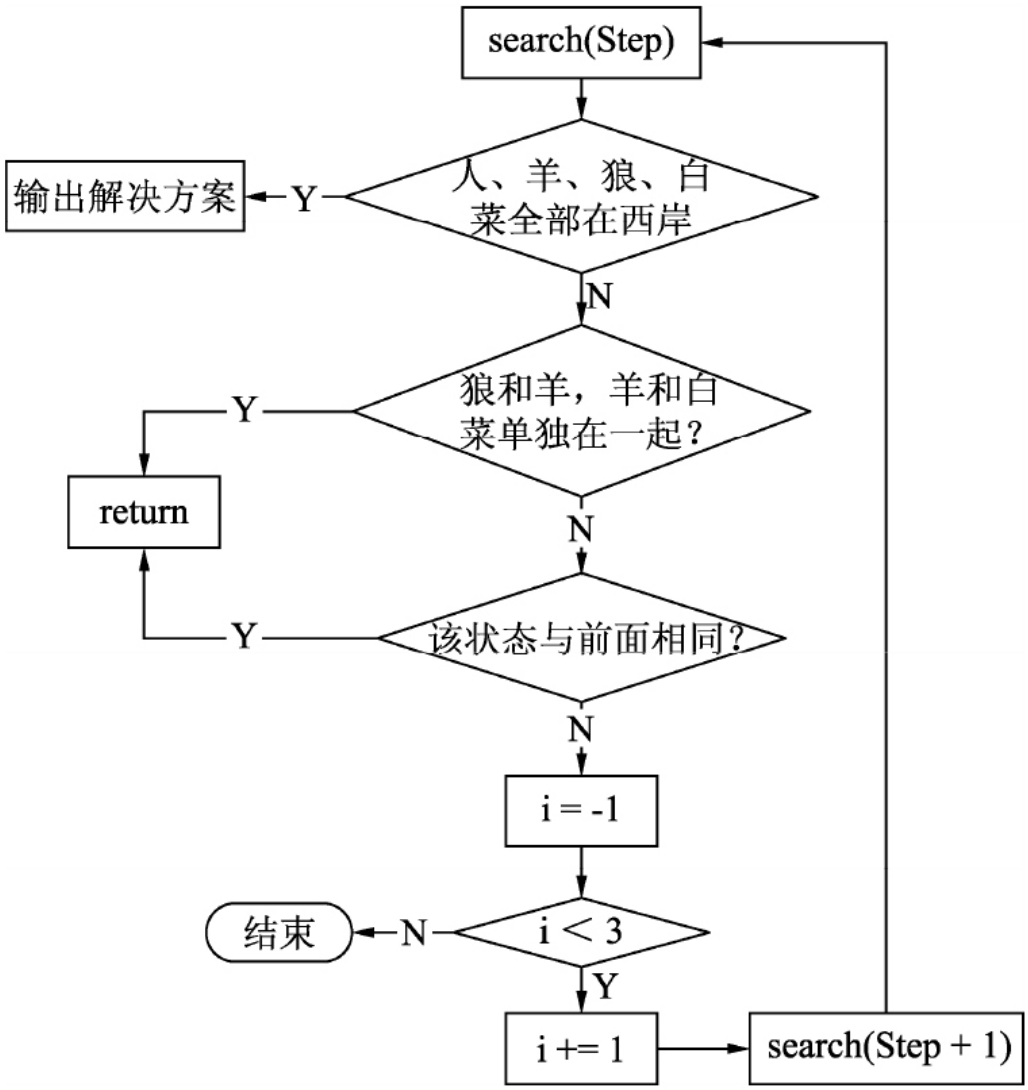


图8.4　递归部分程序结构

4. 完整的程序

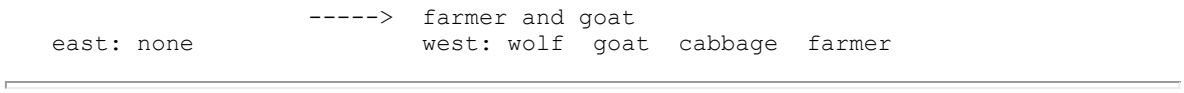
```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 农夫过河


def search(Step):
    # 若该步骤能使各值均为1，则输出结果，进入回归步骤
    if a[Step][0] + a[Step][1] + a[Step][2] + a[Step][3] == 4:
        for i in range(Step + 1):                    # 能够依次输出不同的方案
            print('east:', end=' ')
            if a[i][0] == 0:
                print('wolf', end='  ')
            if a[i][1] == 0:
                print('goat', end='  ')
            if a[i][2] == 0:
                print('cabbage', end='  ')
            if a[i][3] == 0:
```

```
                print('farmer', end='  ')
            if a[i][0] and a[i][1] and a[i][2] and a[i][3]:
                print("none", end='')
            print(end='                ')
            print('west:', end=' ')
            if a[i][0] == 1:
                print("wolf", end='  ')
            if a[i][1] == 1:
                print('goat', end='  ')
            if a[i][2] == 1:
                print('cabbage', end='  ')
            if a[i][3] == 1:
                print('farmer', end='  ')
            if not (a[i][0] or a[i][1] or a[i][2] or a[i][3]):
                print('none', end='')
            print('\n')
            if i < Step:
                print('                        the %d time' % (i + 1))
            if i>0 and i<Step:
                if a[i][3] == 0:                    # 农夫在本岸
                    print("                   ----->  farmer ", end='')
                    print(name[b[i] + 1])
                else:                                                # 农夫在对岸
                    print("                   <-----  farmer ", end='')
                    print(name[b[i] + 1])

        print('\n\n\n')
        return

    for i in range(Step):
        if a[i] == a[Step]:                                  # 若该步与以前的步骤相
同，取消操作
            return

    # 若羊和农夫不在一起而狼和羊或者羊和白菜在一起，则取消操作
    if a[Step][1] != a[Step][3] and (a[Step][2] == a[Step][1] or a[Step][0] ==
a[Step][1]):
        return


    # 递归，从带第一种对象开始依次向下循环，同时限定递归的界限
    for i in range(-1, 3):
        b[Step] = i                                                          #
记录农夫渡河的方式
        a[Step+1] = a[Step][:]                               # 复制上一步的状态，进
行下一步移动
        a[Step + 1][3] = 1 - a[Step + 1][3]          # 农夫过去或者回来

        if i == -1:
            search(Step + 1)                                 # 进行第一步

        elif a[Step][i] == a[Step][3]:                # 若该物与农夫同岸，带回
            a[Step + 1][i] = a[Step + 1][3] # 带回该物
            search(Step + 1)                                 # 进行下一步


if __name__ == '__main__':
    N = 15
    a = [[0] * 4 for i in range(N)]
    b = [0] * N

    name = ["         ",
            "and wolf",
            "and goat",
            "and cabbage"]

    print('              农夫过河问题，解决方案如下：\n')
    search(0)
```

5. 运行结果

在Pycharm下运行程序，结果如下：

```
E:\code\python\Interest-python\venv\Scripts\python.exe  E:/code/python/
            农夫过河问题，解决方案如下：

east: wolf  goat  cabbage  farmer                  west: none

                      the 1 time
east: wolf  cabbage                 west: goat  farmer

                      the 2 time
                <-----  farmer
east: wolf  cabbage  farmer                 west: goat

                      the 3 time
                ----->  farmer and wolf
east: cabbage                 west: wolf  goat  farmer

                      the 4 time
                <-----  farmer and goat
east: goat  cabbage  farmer                 west: wolf

                      the 5 time
                ----->  farmer and cabbage
east: goat                  west: wolf  cabbage  farmer

                      the 6 time
                <-----  farmer
east: goat  farmer                  west: wolf  cabbage

                      the 7 time
                ----->  farmer and goat
east: none                west: wolf  goat  cabbage  farmer



east: wolf  goat  cabbage  farmer                  west: none

                      the 1 time
east: wolf  cabbage                 west: goat  farmer

                      the 2 time
                <-----  farmer
east: wolf  cabbage  farmer                 west: goat

                      the 3 time
                ----->  farmer and cabbage
east: wolf                  west: goat  cabbage  farmer

                      the 4 time
                <-----  farmer and goat
east: wolf  goat  farmer                  west: cabbage

                      the 5 time
                ----->  farmer and wolf
east: goat                  west: wolf  cabbage  farmer

                      the 6 time
                <-----  farmer
east: goat  farmer                  west: wolf  cabbage

                      the 7 time
```


-----> farmer and goat
east: none
west: wolf goat cabbage farmer

8.3　矩阵转置

1. 问题描述

编写一个程序，将一个3行3列的矩阵进行转置。

2. 问题分析

要解决该问题首先应该清楚什么是矩阵的转置。矩阵转置在数学上的定义为：

设A为m×n阶矩阵（即m行n列的矩阵），其第i行第j列的元素是a(i,j)，即$A=a(i,j)_{m\times n}$

定义A的转置为这样一个n×m阶矩阵B，满足$B=a(j,i)_{n\times m}$，即b(i,j)=a(j,i)（B的第i行第j列元素是A的第j行第i列元素），记为A’=B。

假设有如下的矩阵A：

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

则经过转置过，即将矩阵的第i行变成了现在的第i列，则原来的矩阵A变为如下的矩阵B：

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

3. 算法设计

解决矩阵问题时通常都是先将矩阵存放在一个二维数组中，而当矩阵发生变化时，二维数组中对应的元素也会发生变化。

以问题分析中提到的A矩阵为例，要实现A的转置，首先应将其存放在一个二维数组n中，该二维数组中的元素及其内容如表8.1所示。

表8.1　二维数组n中的元素及其内容

n[0][0] 1	n[0][1] 2	n[0][2] 3
n[1][0] 4	n[1][1] 5	n[1][2] 6
n[2][0] 7	n[2][1] 8	n[2][2] 9

将A转置后，二维数组n中的元素及其内容如表8.2所示。

表8.2　A转置后二维数组n中的元素及其内容

n[0][0] 1	n[0][1] 4	n[0][2] 7
n[1][0] 2	n[1][1] 5	n[1][2] 8
n[2][0] 3	n[2][1] 6	n[2][2] 9

观察表8.2可知，转置后矩阵主对角线上的数组元素n[0][0]、n[1][1]、n[2][2]的值并没有发生变化，只是位于对角线右上方的三个元素与位于对角线左下方的三个元素的值进行了交换。具体为

n[0][1]与n[1][0]进行了交换，n[0][2]与n[2][0]进行了交换，n[1][2]与n[2][1]进行了交换。

根据这个发现就可以来设计我们的算法，在对一个3×3阶矩阵转置时，只需将主对角线右上方的数组元素n[0][1]、n[0][2]、n[1][2]分别与主对角线左下方的数组元素n[1][0]、n[2][0]、n[2][1]的值通过一个临时变量进行交换即可，总共只需要进行三次交换就可以实现矩阵的转置。

4. 确定程序框架

根据算法设计来确定程序的主体结构。矩阵问题和循环结构是分不开的，要想访问二维数组中的每个元素，就需要使用一个双重循环。这里我们可以使用一个双重的for循环在循环体中完成数组元素交换，即矩阵转置的任务。

我们先来分析下面这段代码：

```
for i in range(3):
    for j in range(3):
        if i != j:
         temp = n[i][j]
            n[i][j] = n[j][i]
            n[j][i] = temp
```

在上面这段代码中利用双重循环遍历了二维数组中所有的数组元素，在循环体中使用了临时变量temp，将数组中的元素n[i][j]和与之对应的n[j][i]进行了交换。请读者思考这样做是否正确呢？

先考虑外层循环的次数，外层循环的变量为i，它的取值为0、1、2，共循环三次。由于是双重循环，因此对于每一次外层循环，都嵌入了一个循环变量为j的内层循环，j的取值也为0、1、2，内层循环也有三次，这样，内层循环总共运行了3×3=9次。除去(i=0,j=0)、(i=1,j=1)和(i=2,j=2)的三次外，代码中加粗的部分，即if语句中的布尔表达式i!=j在其中的6次循环中都是成立的，所以在整个循环过程中一共进行了6次形如n[i][j]与n[j][i]的数组元素之间值的交换，而我们已经分析过正确的交换次数应该是3次。显然

不符合矩阵转置的要求，因此上面的代码是有问题的，但哪里有问题呢？

出问题的部分就是if语句中的布尔表达式。布尔表达式i!=j没有限定只能将主对角线右上方的数组元素与主对角线左下方的数组元素进行单方向交换，因此，主对角线右上方的数组元素与主对角线左下方的数组元素发生了双方向交换，这样交换次数就变成了6次，而且矩阵并没有被转置。

改正的方法很简单，只需将if语句的条件修改为j>i即可，代码如下：

```
for i in range(3):
    for j in range(3):
        #将主对角线右上方的数组元素与主对角线左下方的数组元素进行单方向交换
        if j > i:
            temp = n[i][j]
            n[i][j] = n[j][i]
            n[j][i] = temp
```

按照上面的代码执行就可以正确地实现矩阵的转置。程序流程图如图8.5所示。

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 矩阵转置

if __name__ == "__main__":
    n = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
    print("原始矩阵：")
    for i in range(3):
        for j in range(3):
            print("%d  " %(n[i][j]), end=" ")                    # 输出原始矩阵
        print()

    for i in range(3):
        for j in range(3):
            #将主对角线右上方的数组元素与主对角线左下方的数组元素进行单方向交换
            if j > i:
                temp = n[i][j]
                n[i][j] = n[j][i]
                n[j][i] = temp

    print("转置矩阵：")
```

```
for i in range(3):
    for j in range(3):
        print("%d  " %(n[i][j]), end=" ")
    print()
```

6. 运行结果

在PyCharm下运行程序，结果如图8.6所示。由图8.6可见，程序正确地生成了原始矩阵的转置矩阵。

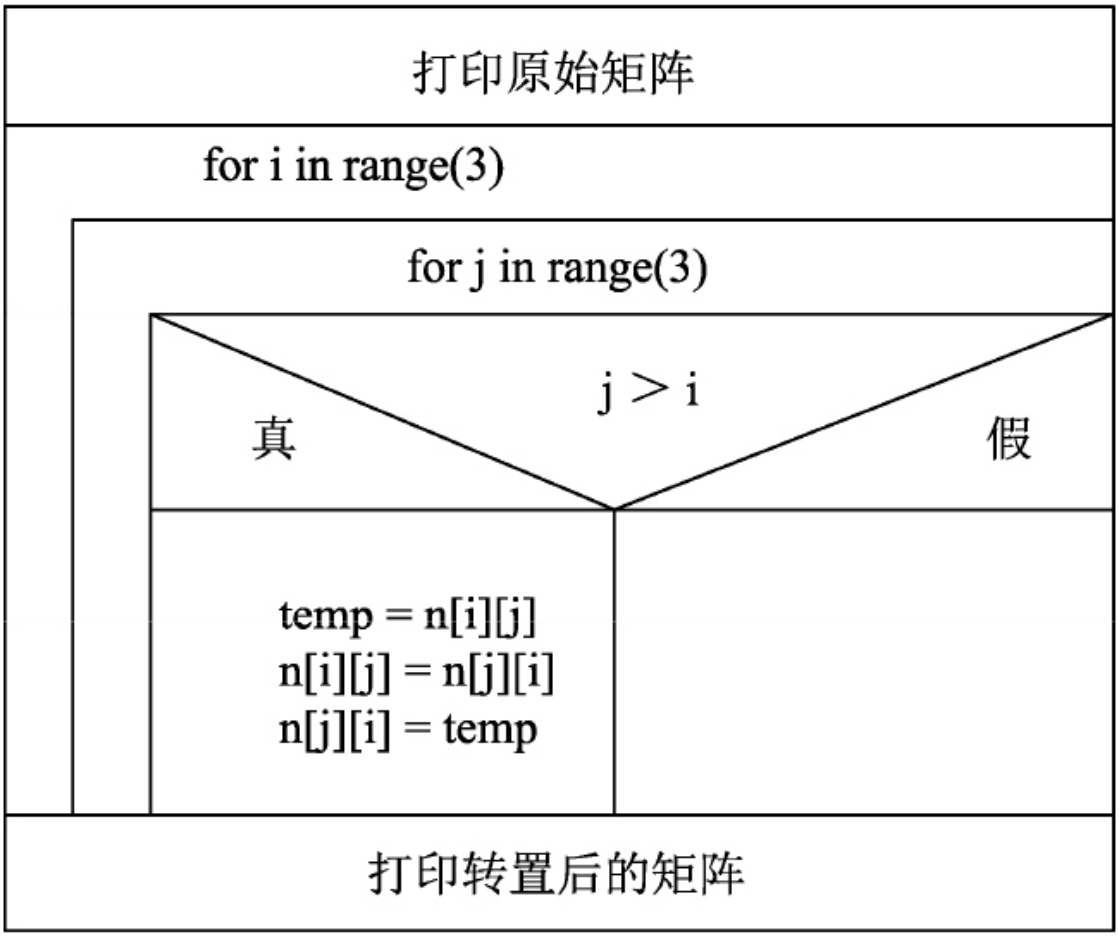


图8.5　程序流程图

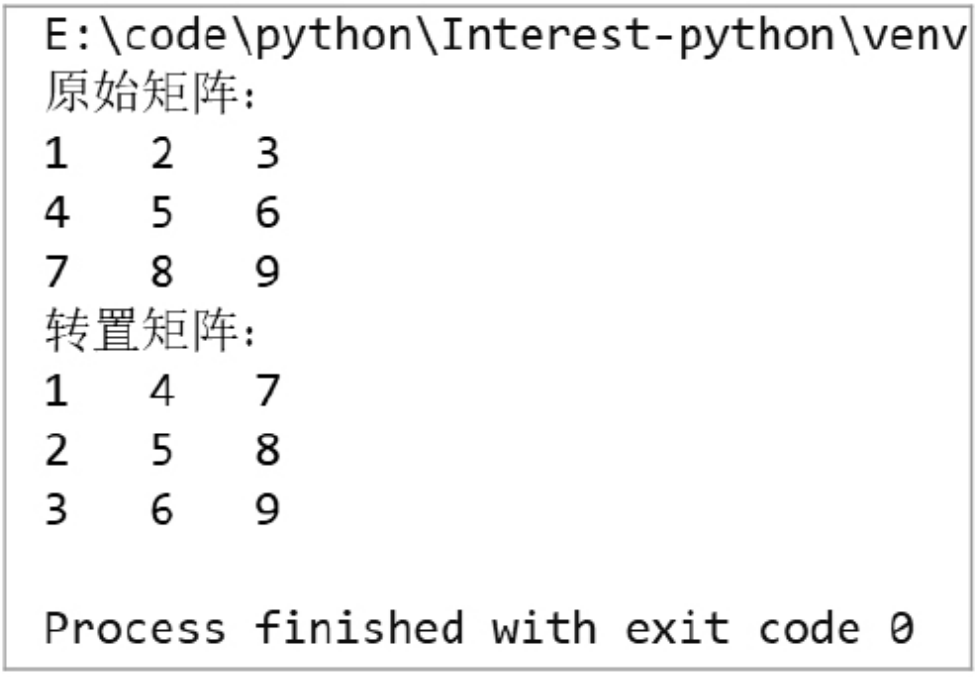


图8.6　运行结果

7. 问题拓展

在解决与矩阵相关的问题时，常常需要使用到二维数组。而要想遍历二维数组中的每个元素就需要使用双重循环，一般使用的是两个for循环，例如我们在解决矩阵转置问题时就是这样处理的。

与矩阵相关的问题除了矩阵转置以外，还有编程实现矩阵的加、减、乘，以及求矩阵元素的最大值、最小值和求对角线元素的和等运算。此处，我们再介绍如何求矩阵元素的最大值和最小值，其他的矩阵问题读者可作为练习自己编程实现。

已知有一个3×4的矩阵，要求编程求出其中值最大及最小的元素所在的行号和列号，以及该元素的值。

显然，要解决这个问题必须要遍历矩阵中的每个元素，因此程序的结构就是一个双重的for循环，在循环体中进行的就是矩阵元素的比较，然后找出最大和最小元素。

该问题的程序流程图如图8.7所示。

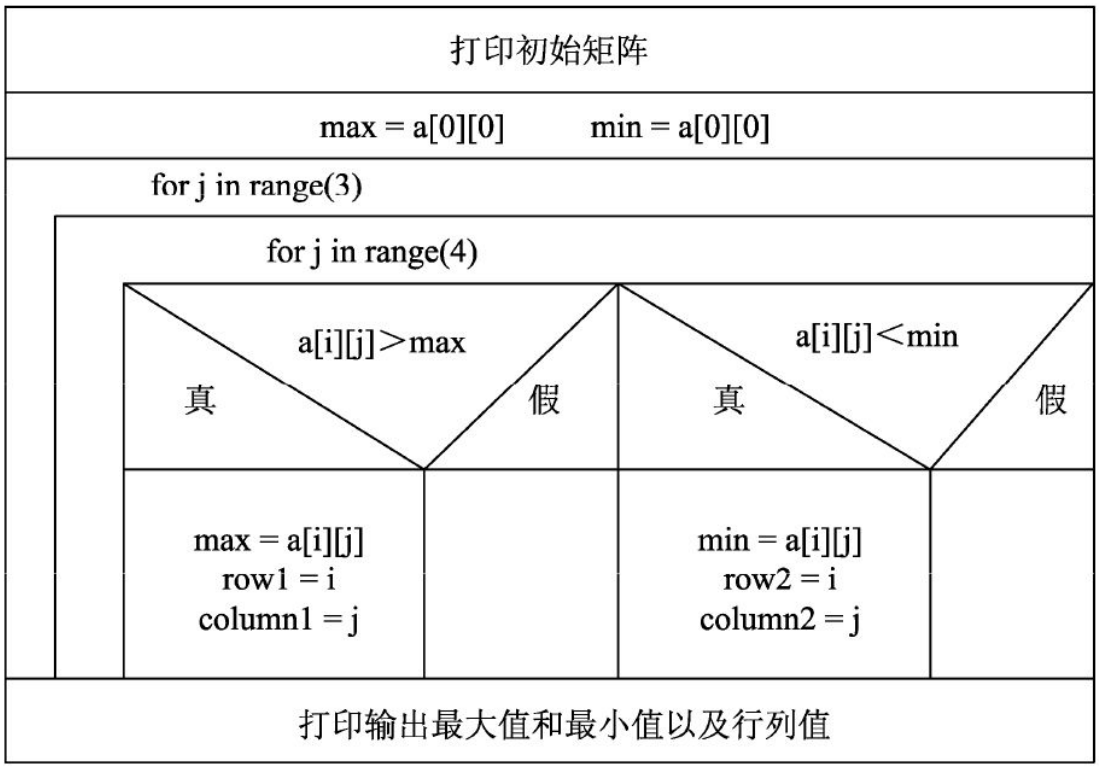


图8.7　程序流程图

完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 求矩阵的最值

if __name__ == "__main__":
    row1, row2 = 0, 0                                                    # 行下标
    column1, column2 = 0, 0                                          # 列下标
    a = [[12, 17, 13, 16], [18, 11, 19, 15], [10, 14, 12, 15]]
    print("打印矩阵：")
    for i in range(3):
        for j in range(4):
            print("%d  " %(a[i][j]), end=" ")
        print()
    max = a[0][0]                                                        # 设置max的
初值
    min = a[0][0]                                                        # 设置min的
初值
    # 矩阵中每个元素逐一与max进行比较
    for i in range(3):
        for j in range(4):
            if a[i][j] > max:                # 如果某个矩阵元素大于max，则将其与max进行交
换
                max= a[i][j]
                row1 = i
                column1 = j
            if a[i][j] < min:                # 如果某个矩阵元素小于min，则将其与min进行交
换
                min = a[i][j]
                row2 = i
                column2 = j
```

```
    print("矩阵的最大值为：%d，其所在行为第%d行，所在列为第%d列\n" %(max, row1,
column1))
    print("矩阵的最小值为：%d，其所在行为第%d行，所在列为第%d列\n" %(min, row2,
column2))
```

在PyChar下运行程序，结果如图8.8所示。

```
E:\code\python\Interest-python\venv\Scripts\python
打印矩阵：
12   17   13   16
18   11   19   15
10   14   12   15
矩阵的最大值为：19，其所在行为第1行，所在列为第2列

矩阵的最小值为：10，其所在行为第2行，所在列为第0列


Process finished with exit code 0
```

图8.8　运行结果

8.4　狼追兔子

1．问题描述

一只兔子躲进了10个环形分布的洞中的某一个。狼在第一个洞中没有找到兔子，就隔一个洞，到第三个洞中去找；如果没有找到，就隔两个洞，到第六个洞中去找；以后每次多隔一个洞去找兔子……这样下去，如果一直找不到兔子，请问兔子可能在哪个洞中？

2．数组

在Python语言中，其数据类型列表就是数组。它是以方括号“[]”包围的数据类型，各个数据之间使用逗号“,”隔开。其内部元素可以是Python语言中的任何数据类型，也可以是另一个列表。我们可以通过各个元素的序号（下标）来访问列表中的元素，也可以对列表进行排序、添加、删除、相加、乘法及分片等操作。

关于列表的相关方法我们在第2章中就介绍过了，这里不再重复。

3．问题分析

首先定义一个数组a[11]，其数组元素为a[1]，a[2]，a[3]…a[10]，这10个数组元素分别表示10个洞，初值均置为1。

接着使用“穷举法”来找兔子，通过循环结构进行穷举，设最大寻找次数为1000次。由于洞只有10个，因此第n次查找对应第n%10个洞，如果在第n%10个洞中没有找到兔子，则将数组元素a[n%10]置0。

当循环结束后，再检查a数组各元素（各个洞）的值，若其值仍为1，则兔子可能藏身于该洞中。

4. 算法设计

程序的流程图如图8.9所示。

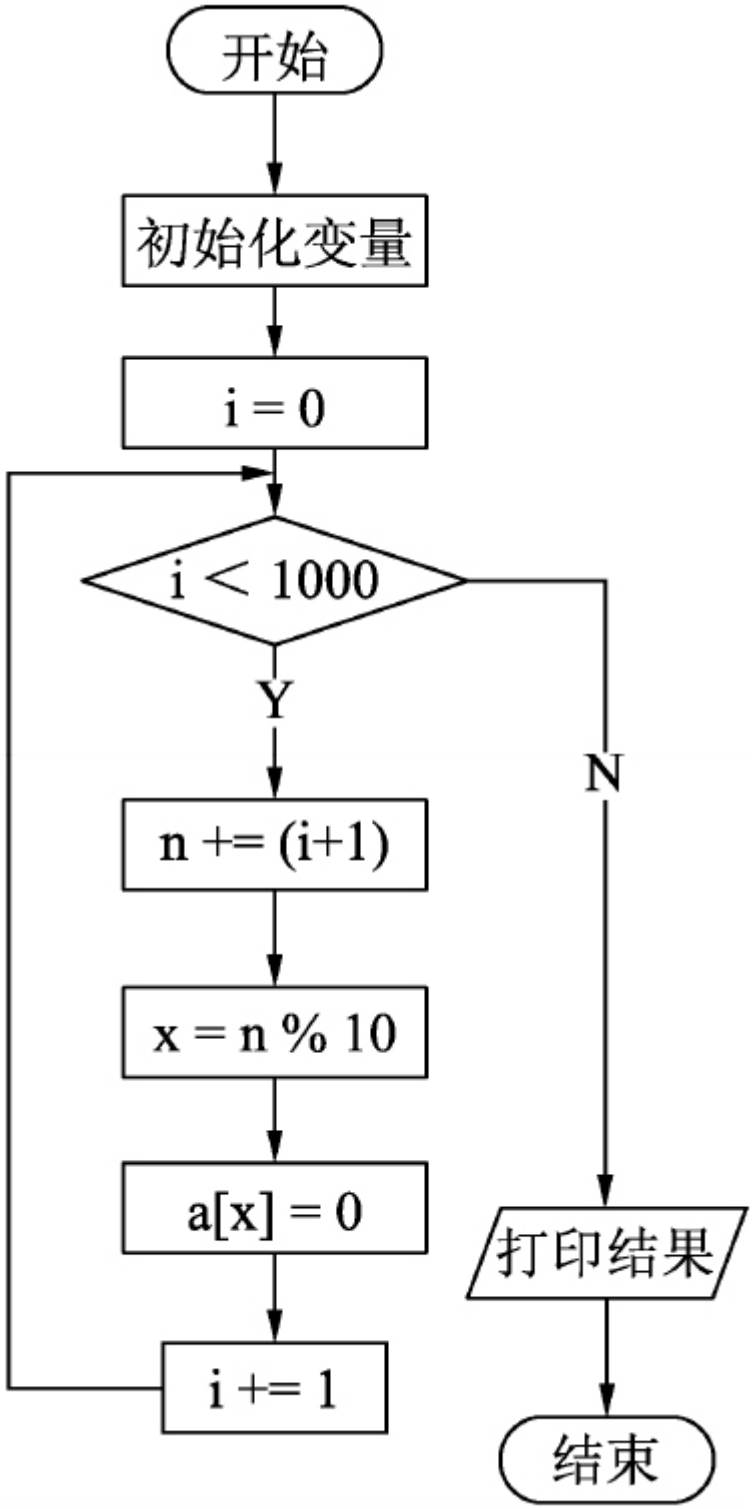


图8.9　程序流程图

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 狼追兔子

if __name__=="__main__":
    n = 0
    x = 0
    a = [1]*11      # 定义一个长度为11的数组，初始值均为1
    for i in range(0, 1000):         # 穷举搜索
        n += (i+1)
        x = n % 10
        a[x] = 0             # 没有找到，设置为0
    for i in range(0, 10):  # 输出结果
        if a[i]:
            print("可能在第%d个洞中" %i)
```

6. 运行结果

在PyCharm下运行程序，结果如图8.10所示。

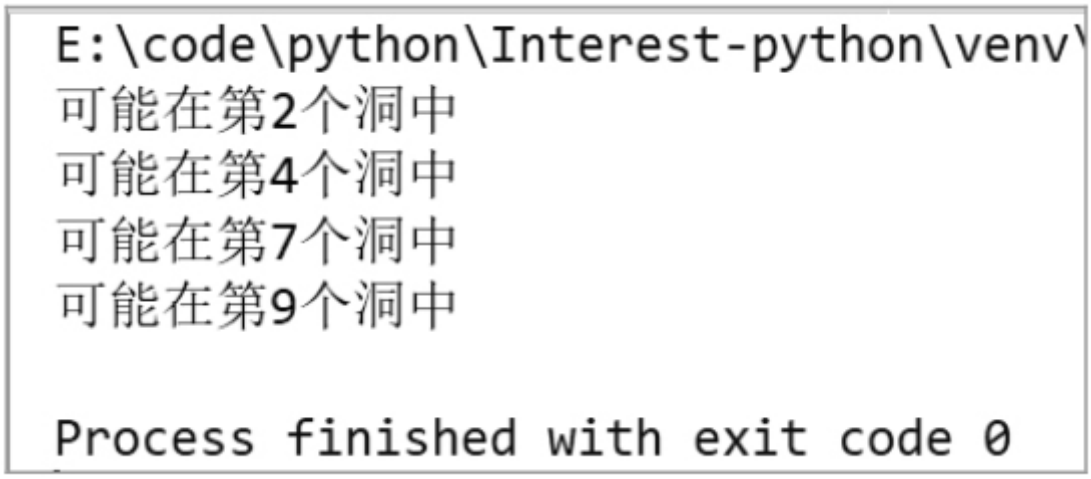


图8.10　运行结果

8.5　选美比赛

1. 问题描述

用Python语言编写程序完成以下任务：

一批选手参加比赛，比赛的规则是最后得分越高，名次越低。当半决赛结束时，要在现场按照选手的出场顺序宣布最后得分和最后的名次，获得相同分数的选手具有相同的名次，名次连续编号，不用考虑同名次的选手人数。

例如：选手序号：1，2，3，4，5，6，7

选手得分：5，3，4，7，3，5，6

输出名次为：3，1，2，5，1，3，4

2. 问题分析

读者可能会不假思索地说出解决此问题的方法，将选手的得分存放到一个数组中，然后按从小到大进行排列，排列靠前的名次靠前，排列靠后的名次靠后。但是问题并非如此简单。首先题目要求按照选手的出场顺序（即选手的序号）宣布最后得分，也就是说并不是按照名次的前后输出选手信息，而是按照选手的序号输出最后的得分和名次。其次，题目要求获得相同分数的选手具有相同的名次，并且名次连续编号，不用考虑同名次的选手人数，因此不存在并列名次占位的情况，也就是说如果存在并列第一，那么下一名的名次是第二名，而不是第三名。另外，如果只是简单地对选手的得分序列进行排序，那么选手的得分与选手的序号就不能构成一一对应的关系，那么这样的排序也就没有意义了。出于这几点考虑，此问题并非是只使用简单的排序运算就可以解决的。

我们使用数组来解决该问题。将每个选手的信息（包括序号、得分、名次）存放在一个数组变量中，然后组成一个新数组。该数组可定义为：

```
psn = [[1, 5, 0], [2, 3, 0], [3, 4, 0], [4, 7, 0], [5, 3, 0], [6, 5, 0],
[7, 6, 0]]
```

说明： [1,5,0]中的1表示选手的序号为1；5表示选手的得分为5；0表示选手的名次。在该数组中我们初始化选手的信息，将选手的名次全部默认设置为0。

最开始每个数组变量中只存放选手的序号和得分的信息，然后以选手的得分为比较对象，从小到大进行排序。算法描述如下：

```
#应用冒泡排序法，排序后的数组psn按照score的值从小到大排列
def sortScore(psn, n):
   for i in range(n-1):
    for j in range(n-1-i):
        if psn[j][1] > psn[j+1][1]:
            #交换psn[j]与psn[j+1]
            tmp = psn[j]
            psn[j] = psn[j+1]
            psn[j+1] = tmp
```

上面的算法中使用了冒泡排序法对数组psn中的n个元素按照score从小到大的顺序进行排序。

接着指定每一位选手的名次。因为此时数组psn已按照score从小到大排列了，因此就比较容易设定每一位选手的名次了。使用算法描述如下：

```
#指定每一位选手的名次
def setRand(psn, n):
    j = 1
    psn[0][2] = 1                                          # 设定第一位选手的名
次
    for i in range(n):                                  # 设定psn[2]～psn[n-1]的名次
        if psn[i][1] != psn[i-1][1]:
            psn[i][2] = j
            j += 1
        else:
            psn[i][2] = psn[i-1][2]                     # psn[i]与psn[i-1]的名次相同
```

首先将第一位选手psn[0]的名次设定为1，因为它的得分是最少的，然后依次为psn[2]～psn[n-1]设定名次。如果psn[i].score不等于psn[i-1].score，说明psn[i]的名次要落后一名；否则psn[i]的名次与psn[i-1]的名次相同。

最后再按照选手的序号重新排序，以便能够按照选手的序号输出结果。该算法描述如下：

```
#最后再按照选手的序号重新排序，以便能够按照选手的序号输出结果
def sortNum(psn, n):
   for i in range(n-1):
    for j in range(n-1-i):
        if psn[j][0] > psn[j+1][0]:                          #交换psn[j]与psn[j+1]
            tmp = psn[j]
            psn[j] = psn[j+1]
            psn[j+1] = tmp
```

3．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 选美比赛

#应用冒泡排序法，排序后的数组psn按照score的值从小到大排列
def sortScore(psn, n):
   for i in range(n-1):
    for j in range(n-1-i):
        if psn[j][1] > psn[j+1][1]:
            #交换psn[j]与psn[j+1]
            tmp = psn[j]
            psn[j] = psn[j+1]
            psn[j+1] = tmp


#指定每一位选手的名次
def setRand(psn, n):
    j = 1
    psn[0][2] = 1                                                  # 设定第一位选手的名次
    for i in range(n):                                        # 设定psn[2]～psn[n-1]的名次
        if psn[i][1] != psn[i-1][1]:
            psn[i][2] = j
            j += 1
        else:
            psn[i][2] = psn[i-1][2]          # psn[i]与psn[i-1]的名次相同


#最后再按照选手的序号重新排序，以便能够按照选手的序号输出结果
def sortNum(psn, n):
   for i in range(n-1):
    for j in range(n-1-i):
        if psn[j][0] > psn[j+1][0]:                #交换psn[j]与psn[j+1]
```

```
            tmp = psn[j]
            psn[j] = psn[j+1]
            psn[j+1] = tmp


def sortRand(psn, n):
    sortScore(psn, n)                                   #以分数为关键字排序
    setRand(psn, n)                                     #按照分数排名次
    sortNum(psn, n)                                     #按照序号重新排序


if __name__ == '__main__':
    #选手的信息组成一个结构体数组
    psn = [[1, 5, 0], [2, 3, 0], [3, 4, 0], [4, 7, 0], [5, 3, 0], [6, 5, 0],
[7, 6, 0]]
    sortRand(psn, 7)
    print("num   score rand  ")
    #输出结果
    for i in range(7):
        print("%d%6d%6d" % (psn[i][0], psn[i]
[1], psn[i][2]))
```

4. 运行结果

在PyCharm下运行程序，结果如图8.11所示。

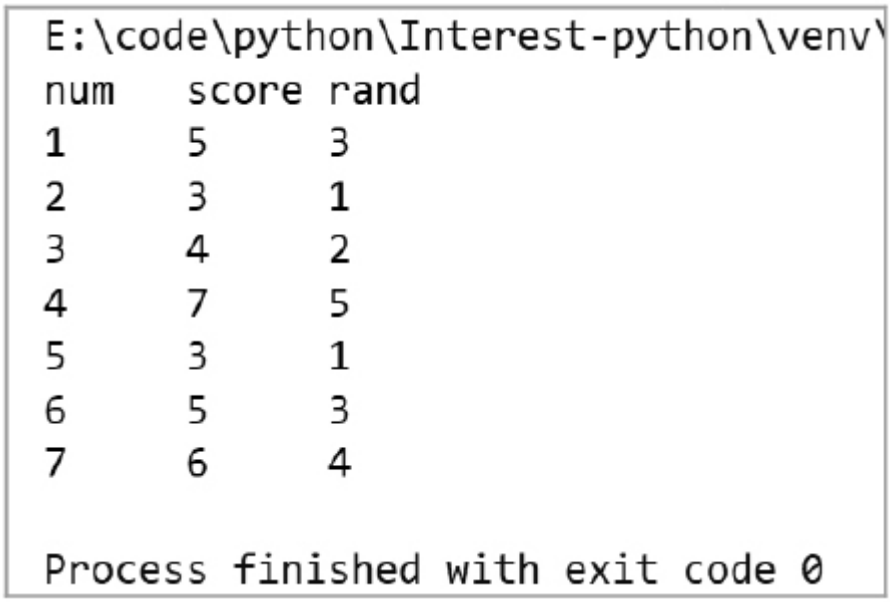


图8.11　运行结果

8.6　邮票组合

1．问题描述

我们寄信都要贴邮票，在邮局有一些小面值的邮票，通过这些小面值邮票中的一张或几张的组合，可以满足不同邮件的不同邮资。现在，邮局有4种不同面值的邮票，在每个信封上最多能贴5张邮票，面值可以相同也可以不同，但至少要贴3种不同面值的邮票，要求编程求出用这4种面值所能组成的邮资的最大值。

2．问题分析

输入：4种邮票的面值，存入数组stamp[4]。

输出：用这4种面值组成的邮资最大值。

按题目的要求，a、b、c、d面值的邮票均可以取0、1、2、3张，但总共5张。为求出组合的最大邮资，我们可以对输入的4种面值邮票进行排序，求出邮票面值的大小，存放到有序数组stamp[4]中。由于至少要贴3种不同的邮票，最多贴5张，我们可以按照面值最大的贴3张、面值第二大的贴1张、面值第三大的贴1张来求出最大的邮资。公式为：

Sum=最大面值×3+第二大面值×1+第三大面值×1

3．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 邮票组合

if __name__=="__main__":
    a, b, c, d = map(int, input("请输入4种邮票的面值,以空格分割: \n").split())
    stamp = [0]*4    # 存放4种邮票的面值
    stamp[0] = a
```

```
stamp[1] = b
stamp[2] = c
stamp[3] = d
print("你输入的4种邮票面值：" )
print(stamp)
# 对邮票面值排序，最大面值为stamp[3]，第二大
  面值为stamp[2]，第三大面值为stamp[1]
stamp.sort()
sum = stamp[3]*3 + stamp[2]*1 + stamp[1]*1
print("组合后的最大邮资为：%d" %sum)
```

4. 运行结果

在PyCharm下运行程序，结果如图8.12所示。

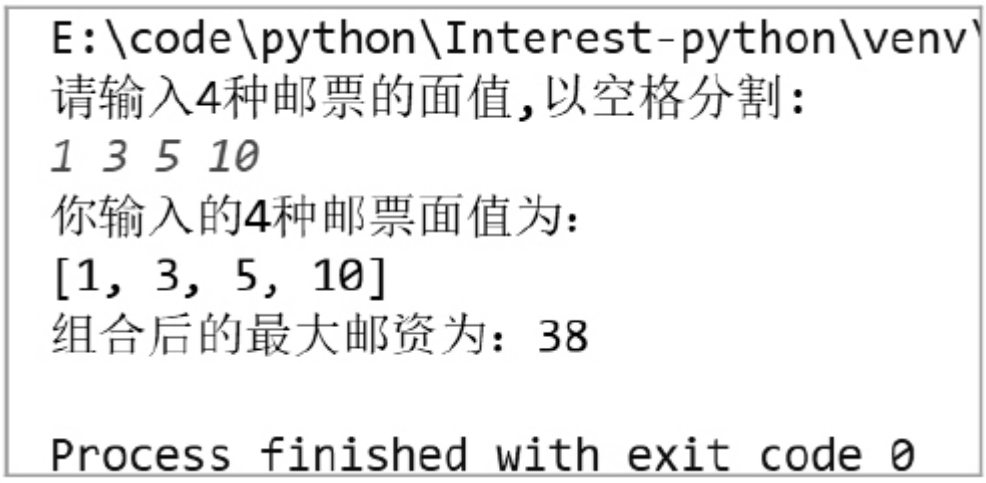


图8.12　运行结果

8.7　魔方阵

1．问题描述

编写程序，实现如图8.13所示的5-魔方阵。

2．问题分析

所谓“n-魔方阵”，指的是使用1～n^2 共n^2 个自然数排列成一个n×n的方阵，其中n为奇数。该方阵的每行、每列以及对角线元素之和都相等，并为一个只与n有关的常数，该常数为n×(n^2 +1)/2。

例如，图8.13所示的5-魔方阵，其第一行、第一列以及主对角线上各元素之和如下。

第一行元素之和：17+24+1+8+15=65

第一列元素之和：17+23+4+10+11=65

主对角线上元素之和：17+5+13+21+9=65

而n×(n^2 +1)/2=5×(5^2 +1)/2=65

可以验证，5-魔方阵中其余各行、各列以及副对角线上的元素之和也都为65。

假定阵列的行列下标都从0开始，则魔方阵的生成方法如下：

1）在第0行中间置1，并对从2开始的其余n^2 -1个数依次按下面2～5所列规则存放。

2）假定当前数的下标为(i,j)，则下一个数的放置位置为当前位置的右上方，即下标为(i-1,j+1)的位置。

3）如果当前数在第0行，即i-1小于0，则将下一个数放在最后一行的下一列上，即下标为(n-1,j+1)的位置。

4）如果当前数在最后一列上，即j+1大于n-1，则将下一个数放在上一行的第一列上，即下标为(i-1,0)的位置。

5）如果当前数是n的倍数，则将下一个数直接放在当前位置的正下方，即下标为(i+1,j)的位置。

按照上面的规则，5-魔方阵中数字的存放过程如图8.14所示。

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

图8.13　5-魔方阵

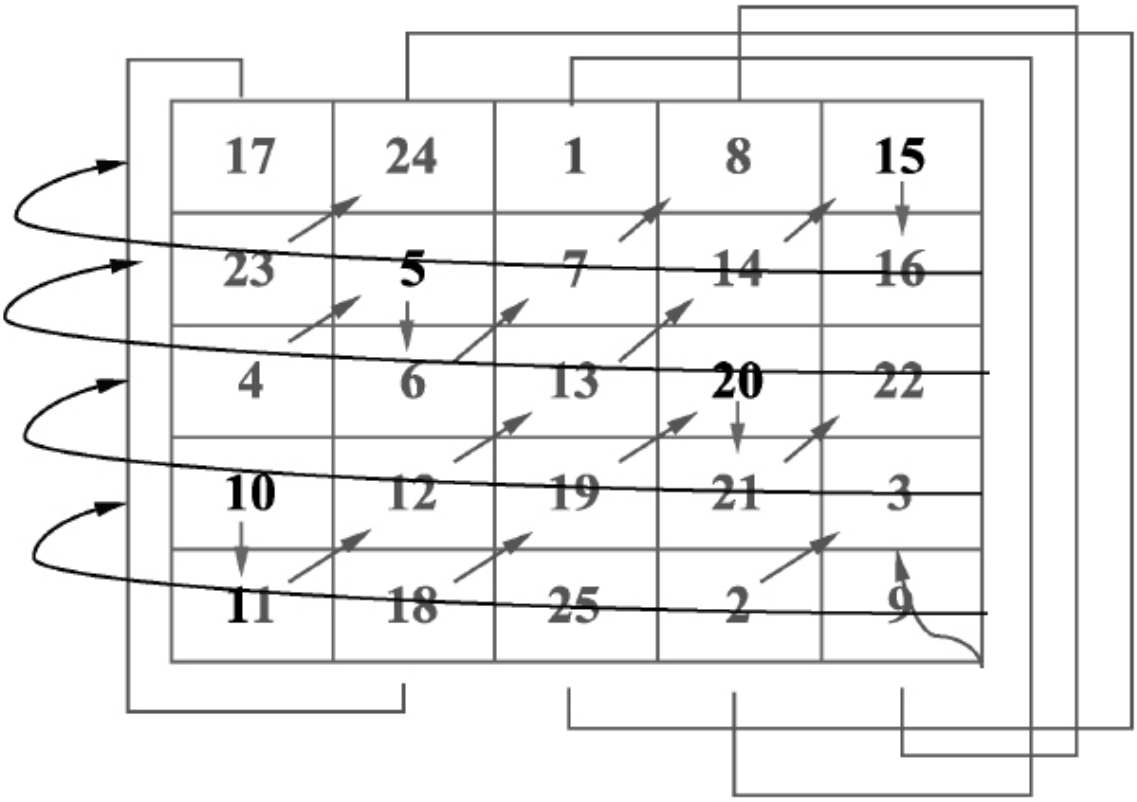


图8.14　魔方阵中数字存放过程

3．算法设计

在设计算法时采用了下面的一些方法：

1）定义array()函数，该函数根据输入的n值，生成并显示一个魔方阵，当发现n不是奇数时，就加1使之成为奇数。

2）显然我们应该使用数组来表示魔方阵。

4．确定程序框架

（1）定义数组

要生成n阶魔方阵，共需要存入n^2 个自然数，在程序中首先要为这n^2 个自然数进行初始化定义，代码如下：

```
max = n * n
mtrx = [1] * max
```

（2）生成魔方阵

生成魔方阵时按照在问题分析中介绍的生成规律，需要考虑几种不同情况，这几种情况在代码中都应该体现出来。按照生成规律，首先应该将自然数1存入数组mtrx中，然后从2开始再依次确定每个数的存放位置。生成魔方阵的代码如下：

```
    mtrx[n // 2] = 1                                    # 将1存入数组
    i = 0                                                           # 自然数
1所在行
    j = n // 2                                          # 自然数1所在列
    # 从2开始确定每个数的存放位置
    for num in range(2, max + 1):
        i = i - 1
        j = j + 1
        if (num - 1) % n == 0:                      # 当前数是n的倍数
            i = i + 2
            j = j - 1
        if i < 0:                                       # 当前数在第0行
            i = n - 1
        if j > n - 1:                                   # 当前数在最后一列，即n-1
列
            j = 0
        no = i * n + j                                  # 找到当前数在数组中存
放的位置
        mtrx[no] = num
```

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 魔方阵

def array(n):
    if n % 2 == 0:                                  # n是偶数，则加1使其变为奇数
        n = n + 1
    max = n * n
    mtrx = [1] * max
    mtrx[n // 2] = 1                                # 将1存入数组
    i = 0                                                       # 自然数1所
在行
    j = n // 2                                              # 自然数1所在列
    # 从2开始确定每个数的存放位置
    for num in range(2, max + 1):
        i = i - 1
        j = j + 1
        if (num - 1) % n == 0:                  # 当前数是n的倍数
            i = i + 2
            j = j - 1
        if i < 0:                                       # 当前数在第0行
            i = n - 1
        if j > n - 1:                               # 当前数在最后一列，即n-1列
            j = 0
        no = i * n + j                                  # 找到当前数在数组中存放的位置
```

```
            mtrx[no] = num
        # 打印生成的魔方阵
        print("生成的%d-魔方阵为：" %n, end="")
        no = 0
        for i in range(0, n):
            print()
            for j in range(0, n):
                print("%3d" %mtrx[no], end=" ")
                no += 1
        print()

if __name__ == "__main__":
    n = int(input("请输入n值："))
    array(n)                                                    # 调用array()函数
```

6. 运行结果

在PyCharm下运行程序，屏幕上提示“请输入n值：”，输入5，则可以正确打印出5-魔方阵，运行结果如图8.15所示。

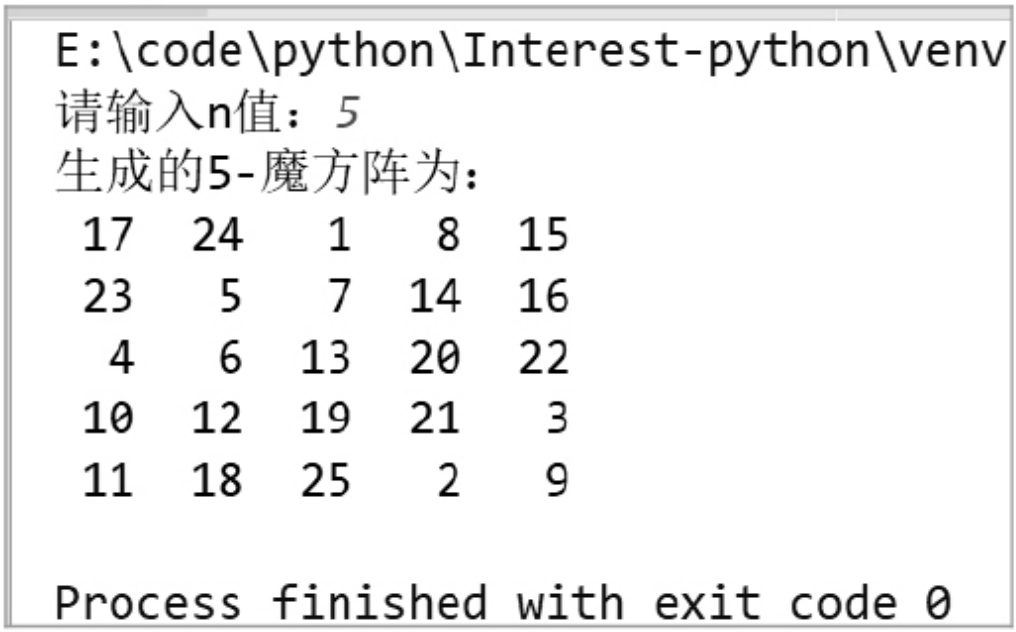


图8.15　运行结果

7. 问题拓展

在解决该问题时，我们使用一维数组来生成魔方阵。这里再给出直接使用二维数组来生成5-魔方阵的程序，以便于读者进行比较。

直接使用二维数组生成5-魔方阵的代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 魔方矩阵

if __name__ == "__main__":
    N = 5
    a = [[0] * 5 for i in range(5)]          # 定义一个5×5的二维数组
    i = 0                                                                    #
自然数1的行标
    j = N // 2                                                               # 自然数1的
列标
    t = N - 1
    k = 1
    while k <= (N * N):                                                  # 变量k控制循环和自然数
        a[i][j] = k
        x = i                                                                # 变量x保存
新的行
        y = j                                                                # 变量y保存
新的列
        if i == 0:
            i = t                                                        # 当前数在第0行，则下
一个数在最后一行
        else:
            i = i - 1                                                    # 产生行，非第0行，则
取下一列
        if j != t:
            j = j + 1                                                    # 产生列，非最后列，则
取下一列
        else:
            j = 0                                                                # 当前数在最
后一列，则下一个数在第一列
        if a[i][j] != 0:
            i = x + 1
            j = y
        k += 1
    # 打印生成的魔方阵
    print("生成的5-魔方阵为：", end="")
    for i in range(N):
        print()
        for j in range(N):
            print("%3d " %a[i][j], end=" ")
    print()
```

上面的代码定义了二维数组a[N][N]，其中N是字符常量，表示数值5，因此数组a[N][N]可以存放一个5-魔方阵。接着找到自然数所在位置的行标和列标，下面就开始将N×N（即25）个数存入二维数组a。

将数存入二维数组a使用的是while循环，循环变量为k值，共循环25次，存入时要遵循魔方阵的存入规则。每次循环时，先将当前数k存入二维数组的指定元素a[i][j]中，然后判断下一个数的存放位置，即i和j的值。加粗的代码是表示如果按魔方阵的存入规则得出的位置已经被占用，则下一个自然数应该存放在当前数的下一行上，但是与当前数在同一列上。这与使用魔方阵生成规则中的第5条

“如果当前数是n的倍数，则将下一个数直接放在当前位置的正下方，即下标为(i+1,j)的位置。”找到的位置是相同的。

运行程序，结果如图8.16所示。由图8.16中可以看出，打印出的5-魔方阵与图8.15中打印出的5-魔方阵是完全相同的。

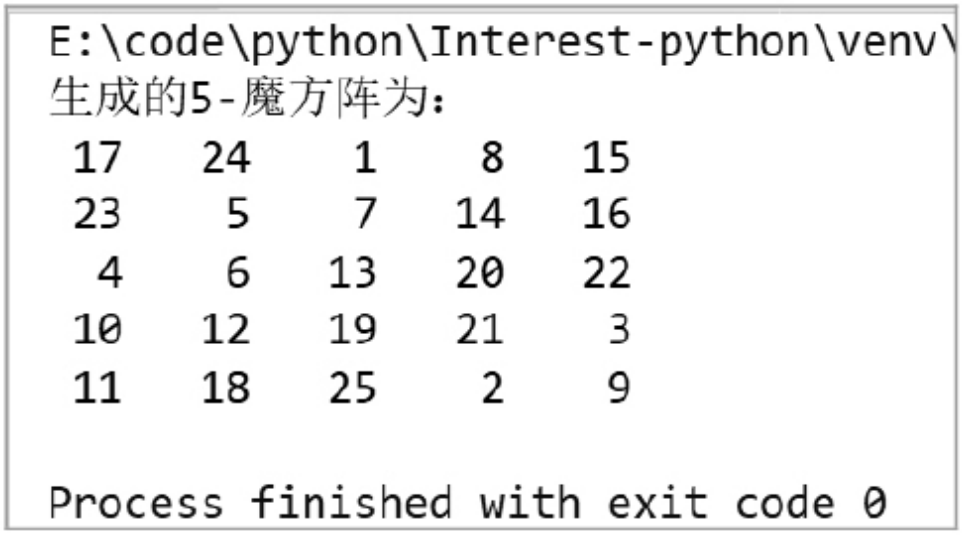


图8.16　运行结果

8.8　马踏棋盘

1. 问题描述

国际象棋的棋盘为8×8的方格棋盘。现将“马”放在任意指定的方格中，按照“马”走棋的规则将“马”进行移动。要求每个方格只能进入一次，最终使得“马”走遍棋盘的64个方格。编写一个Python程序，实现马踏棋盘操作，要求用1～64这64个数字标注“马”移动的路径，也就是按照求出的行走路线，将数字1～64依次填入棋盘的方格中，并输出。

2. 问题分析

国际象棋中，“马”的移动规则如图8.17所示。

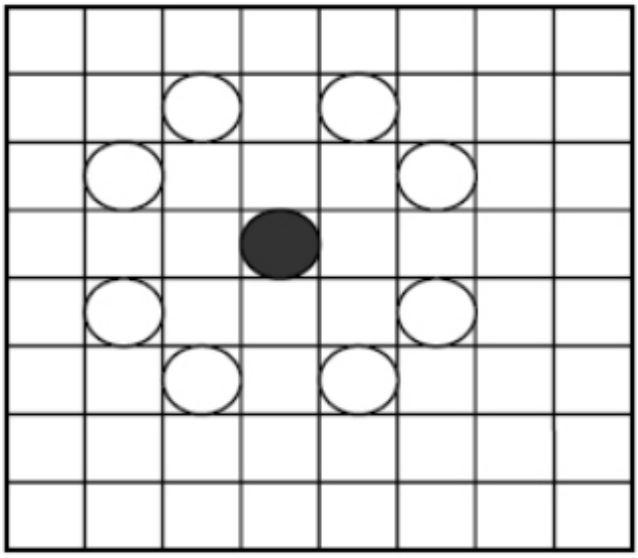


图8.17　“马”的移动规则

图中实心的圆圈代表“马”的位置，它下一步可移动到图中空心圆圈所标注的8个位置上，该规则叫作“马走日”。但是如果“马”位于棋盘的边界附近，那么它下一步可移动到的位置就不一定有8个了，因为要保证“马”每一步都走在棋盘中。

马踏棋盘的问题其实就是要将1～64填入到一个8×8的矩阵中，要求相邻的两个数按照“马”的移动规则放置在矩阵中。例如数字a放置在矩阵的(i,j)位置上，则数字a+1只能放置在矩阵的(i-2,j+1)、(i-1,j+2)、(i+1,j+2)、(i+2,j+1)、(i+2,j-1)、(i+1,j-2)、(i-1,j-2)、(i-2,j-1)之中的一个位置上。这样在矩阵中从1遍历到64，就得到了马踏棋盘的行走路线。因此本题的最终目的是输出一个8×8的矩阵，在该矩阵中填写1～64这64个数字，相邻数字之间遵照“马走日”的规则。

3．算法设计

解决马踏棋盘问题的一种比较容易理解的方法是应用递归的深度优先搜索的思想。因为“马”每走一步都是盲目的，它并不能判断当前的走步一定正确，而只能保证当前这步是可走的。“马”走的每一步棋都是从它当前位置出发，向下一步的8个位置中的1个行走（在它下一步有8个位置可走的情况下）。因此“马”当前所走的路径并不一定正确，因为它可能还有剩下的可选路径没有尝试，如图8.18所示。

在图8.18中，假设最开始“马”位于棋盘的(0,0)位置，接下来“马”有两处位置可走，即(1,2)和(2,1)。这时“马”是无法确定走2的位置最终是正确的，还是走3的位置最终是正确的。因此“马”只能任意先从一个路径走下去（例如从2的位置）。如果这条路是正确的，那当然是幸运的，如果不正确，则“马”要退回到第一步，继续从3的位置走下去。以后“马”走的每一步都遵循这个规则。这个过程就是一种深度搜索的过程，同时也是一种具有重复性操作的递归过程。可以用一棵“探索树”来描述该深度优先搜索的过程，如图8.19所示。

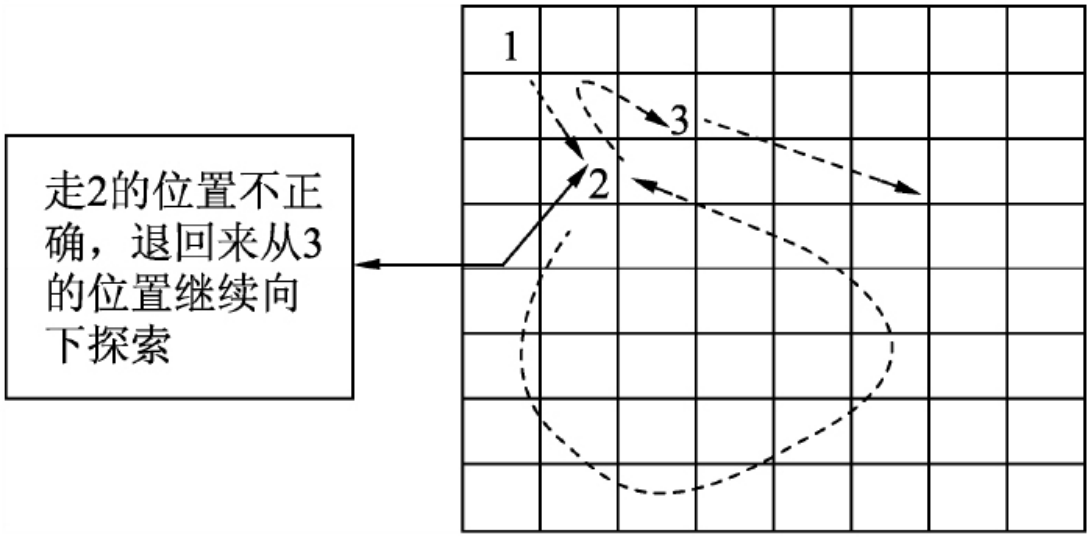


图8.18　“马”的可选路径

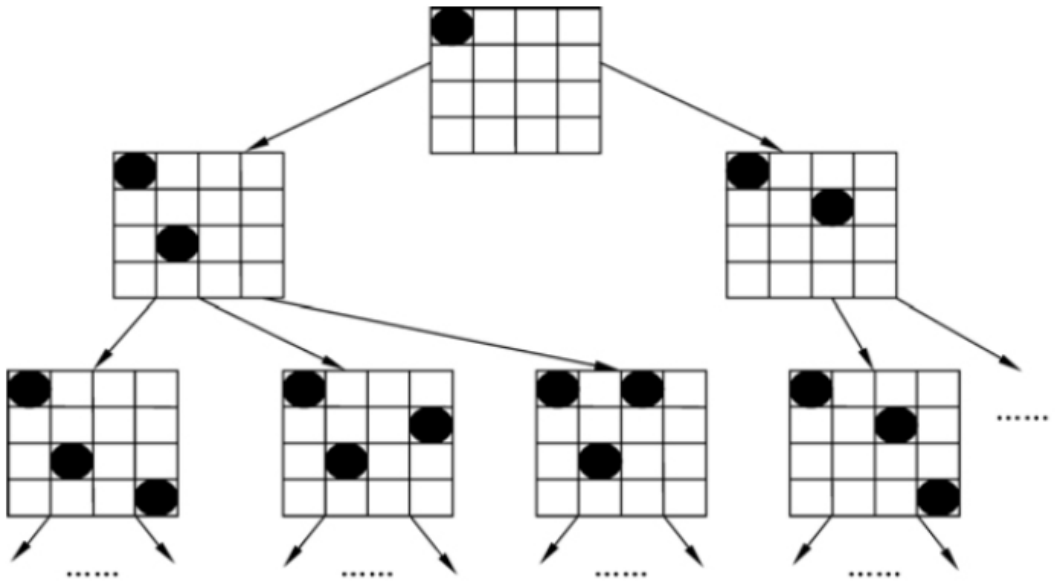


图8.19　深度优先搜索过程

“马”的行走过程实际上就是一个深度探索的过程。如图8.19所示，“探索树”的根结点为“马”在棋盘中的初始位置（这里用4×4的棋盘示意）。接下来“马”有两种行走方式，于是根结点派生出两个分支。而再往下一步行走，根结点的两个孩子又能够分别派生出其他不同的“行走路线”分支，如此派生下去，就得到了“马”的所有可能的走步状态。可以想见，该探索树的叶子结点只可能有两种状态：一是该结点不能再派生出其他的“走步”分支了，也就是“马”走不通了；二是棋盘中的每个方格都被走到，即

“马”踏遍棋盘。于是从该探索树的根结点到第二种情况的叶结点构成的路径就是马踏棋盘的行走过程。

如何才能通过搜索这棵探索树找到这条马踏棋盘的行走路径呢？可以采用深度优先搜索的方法以先序的方式访问树中的各个结点，直到访问到叶结点。如果叶结点是第二种情况的叶结点，则搜索过程可以结束，因为找到了马踏棋盘的行走路径；如果叶结点为第一种情况的叶结点，即走不通了，则需要返回到上一层的结点，顺着该结点的下一条分支继续进行深度优先搜索下去。

因此在设计“马踏棋盘”的算法时可以借鉴图的深度优先遍历算法和二叉树的先序遍历（根左右）算法。但是在这里并不需要真正地构建这样一棵探索树，我们只需要借用探索树的思想。在实际的操作过程中，所谓的探索树实际就是深度优先搜索的探索路径，每个结点实际就是当前的棋盘状态，而所谓的叶结点要么就是在当前棋盘状态下，“马”无法再进行下一步行走；要么就是马踏棋盘成功。该算法可描述如下：

```
# 深度优先搜索的“马踏棋盘”
def TravelChessBoard(x, y, tag):
    x1, y1, flag, count = x, y, False, 0
    chess[x][y] = tag
    if tag == 60:                                                    #
搜索成功，返回1
        return True
    flag, x1, y1 = nextxy(x1, y1, count)     # 找到基于(x1,y1)的下一个可走位置
    # 上一步未找到，则在其余几种可走位置中寻找下一个可走位置
    while not flag and count < 7:
        count = count + 1
        print('(1): ', count)
        flag, x1, y1 = nextxy(x1, y1, count)

    while flag:                                                  # 找到下一个可走位置，
则进行深度优先搜索
        if TravelChessBoard(x1, y1, tag + 1):        # 递归
            return True
        x1 = x
        y1 = y
        count = count + 1
        flag, x1, y1 = nextxy(x1, y1, count)         # 寻找下一个(x,y)
        while not flag and count < 7:                    # 循环地寻找下一个(x,y)
            count = count + 1
            print('(2): ', count)
            flag, x1, y1 = nextxy(x1, y1, count)

    if not flag:
        chess[x][y] = 0                                                  # 搜索不成
功，擦除足迹，返回0
    return False
```

该算法中通过函数TravelChessBoard()递归地搜索“马”的每一种走法。其中参数x、y指定“马”当前走到棋盘中的位置，tag是标记变量，每走一个棋盘方格，tag自动增1，它标识着马踏棋盘的行走路线。

算法首先将当前“马”处在棋盘中的位置上添加标记tag，然后判断tag是否等于64，如果等于64，说明这是马踏棋盘的最后一步，因此搜索成功，程序应当结束，返回1。否则，找到“马”下一步可以走到的位置(x1,y1)，如果找到这个位置坐标，flag置1，否则flag置0。

下面在flag为1的条件下（即找到坐标(x1,y1)），递归地调用函数TravelChessBoard()。也就是从(x1,y1)指定的棋盘中的位置继续向下深度搜索。如果从(x1,y1)向下搜索成功，即程序一直执行下去，直到tag等于64返回1，那就说明“马”已经踏遍棋盘（马踏棋盘的过程是：先走到棋盘的(x,y）位置，再从(x,y）的下一个坐标(x1,y1)向下深度搜索，直至走遍全棋盘），于是搜索结束，返回1；否则继续找到“马”的下一个可以行走的坐标(x1,y1)，如果找到这个位置坐标，flag置1，并从(x1,y1)向下重复上述的递归搜索，否则flag置0，本次递归结束。

如果找遍当前位置(x,y)的下一个坐标(x1,y1)（一般情况是8种），但是从(x1,y1)向下继续深度优先搜索都不能成功地“马踏棋盘”（此时flag等于0），则表明当前所处的状态并不处于马踏棋盘的“行走路径”上，也就是说“马”本不应该走到(x,y)的位置上，因此将chess[x][y]置0，表明棋盘中该位置未被走过（擦掉足迹），同时返回0，程序退到上一层的探索状态。

这里应当知道，所谓当前位置(x,y)的下一个坐标(x1,y1)是指“马”下一步可以走到的地方，用坐标(x1,y1)返回。在探索树中它处在(x,y)所在的结点的子结点中，如图8.20所示。

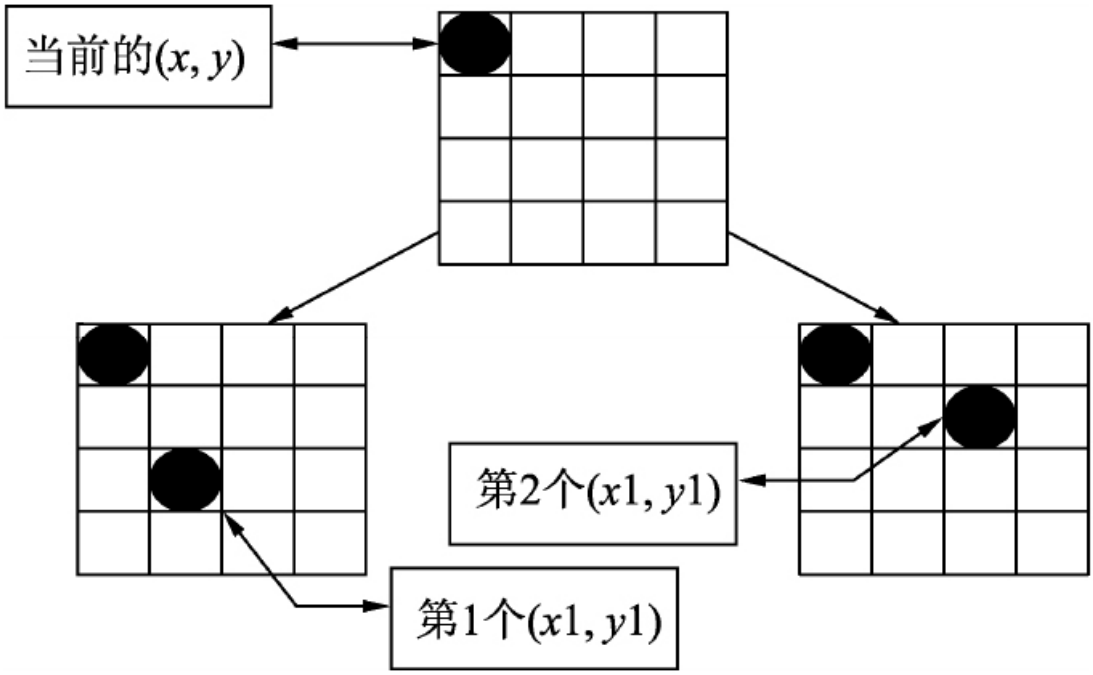


图8.20　(x,y)的下一个坐标(x1,y1)

当前位置(x,y)的下一个坐标(x1,y1)的可选个数由当前棋盘的局面决定。一般情况下有8种可走的位置（如图8.17所示），但是如果“马”位于棋盘的边缘（如图8.18所示的探索树的根结点）或者8个可选位置中有的已被“马”前面的足迹所占据，则这两种情况下(x1,y1)的可选个数就不是8个了。

上述搜索算法相当于先序遍历探索树，只不过它不一定是将探索树完整地遍历，而是当tag等于64时，也就是棋盘被全部“踏遍”时就停止继续搜索了。

4．完整的程序

根据以上分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 马踏棋盘

# 找到基于（x,y）位置的下一个可走的位置
def nextxy(x, y, count):
    if count == 0 and x + 2 <= X - 1 and y - 1 >= 0 and chess[x + 2][y - 1]
== 0:                                                                    # 找到坐
标(x+2,y-1)
        x = x + 2
```

```
        y = y - 1
        flag = True

    elif count == 1 and x + 2 <= X - 1 and y + 1 <= Y - 1 and chess[x + 2][y
+ 1] == 0:                                                          # 找到坐标
(x+2,y+1)
        x = x + 2
        y = y + 1
        flag = True

    elif count == 2 and x + 1 <= X - 1 and y - 2 >= 0 and chess[x + 1][y -
2] == 0:                                                                # 找到坐
标(x+1,y-2)
        x = x + 1
        y = y - 2
        flag = True

    elif count == 3 and x + 1 <= X - 1 and y+2 <= Y-1 and chess[x+1][y+2]
== 0:                                                                   # 找到坐
标(x+1,y+2)
        x = x + 1
        y = y + 2
        flag = True

    elif count == 4 and x - 2 >= 0 and y - 1 >= 0 and chess[x - 2][y - 1]
== 0:                                                                   # 找到坐
标(x-2,y-1)
        x = x - 2
        y = y - 1
        flag = True

    elif count == 5 and x - 2 >= 0 and y + 1 <= Y - 1 and chess[x - 2][y +
1] == 0:                                                                # 找到坐
标(x-2,y+1)
        x = x - 2
        y = y + 1
        flag = True

    elif count == 6 and x - 1 >= 0 and y - 2 >= 0 and chess[x - 1][y - 2]
 == 0:                                                                  #
找到坐标(x-1,y-2)
        x = x - 1
        y = y - 2
        flag = True

    elif count == 7 and x - 1 >= 0 and y + 2 <= Y - 1 and chess[x - 1][y +
2] == 0:                                                                # 找到坐
标(x-1,y+2)
        x = x - 1
        y = y + 2
        flag = True

    else:
        flag = False

    return flag, x, y

# 深度优先搜索的"马踏棋盘"
def TravelChessBoard(x, y, tag):
    x1, y1, flag, count = x, y, False, 0
    chess[x][y] = tag
    if tag == 60:                                                       #
搜索成功，返回1
        return True
    flag, x1, y1 = nextxy(x1, y1, count)     # 找到基于(x1,y1)的下一个可走位置
    # 上一步未找到，则在其余几种可走位置中寻找下一个可走位置
    while not flag and count < 7:
        count = count + 1
        # print('(1): ', count)
        flag, x1, y1 = nextxy(x1, y1, count)

    while flag:                         # 找到下一个可走位置，则进行深度优先搜索
```

```
        if TravelChessBoard(x1, y1, tag + 1):        # 递归
            return True
        x1 = x
        y1 = y
        count = count + 1
        flag, x1, y1 = nextxy(x1, y1, count)         # 寻找下一个(x,y)
        while not flag and count < 7:                    # 循环地寻找下一个(x,y)
            count = count + 1
            # print('(2): ', count)
            flag, x1, y1 = nextxy(x1, y1, count)

    if not flag:
        chess[x][y] = 0                                                  # 搜索不成
功，擦除足迹，返回0
    return False


if __name__ == '__main__':
    X = 8
    Y = 8
    chess = [[0]*X for i in range(Y)]                                # 初始化，棋盘的所有位
置都置0

    if TravelChessBoard(2, 0, 1):                                    # 深度优先搜索
        for i in range(X):
            for j in range(Y):
                print("%-5d" % chess[i][j],
end='')
            print()
        print("The horse has travelled the
chess borad")
    else:
        print("The horse cannot travel the
chess board")
```

5. 运行结果

在PyCharm下运行程序，结果如图8.21所示。

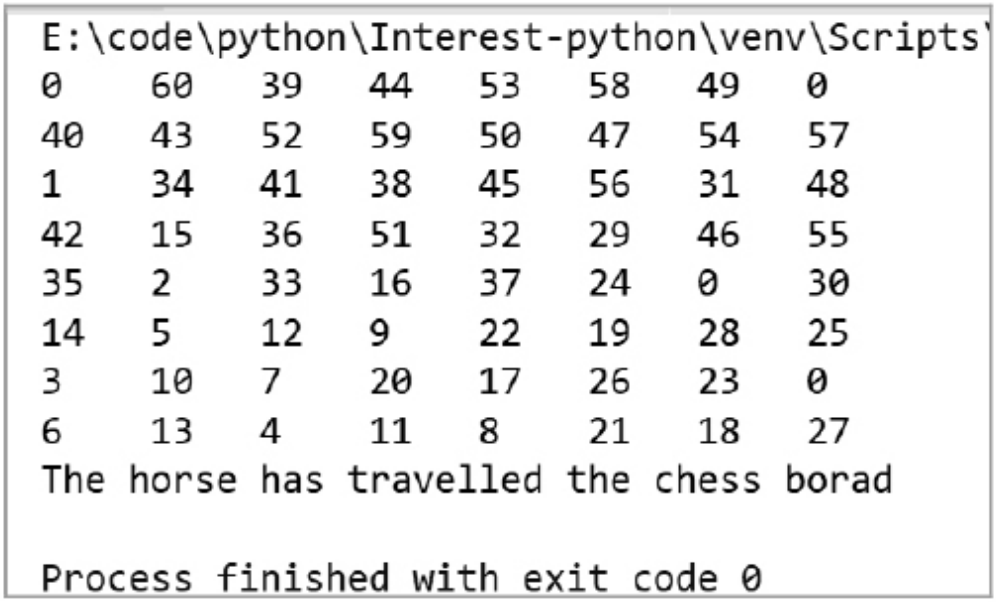


图8.21　运行结果

8.9 删除“*”符号

1. 问题描述

现在有一串字符需要输入，规定输入的字符串中只包含字母和“*”符号。请编写程序，实现除了字符串前后的“*”之外，将串中其他的“*”全部删除。

例如，假设输入的字符串为：****A*BC*DEF*G********，删除串中的“*”后，字符串变为：****ABCDEFG********。

2. 问题分析

该问题需要对字符串进行操作。在Python语言中，可以使用字符串相关的函数结合数组来实现该功能。

3. 字符串

在Python 3中有6种标准的数据类型：Number（数字）、String（字符串）、List（列表或数组）、Tuple（元组）、Sets（集合）和Dictionary（字典）。

其中String（字符串）用来表示或存储文本数据。Python中的字符串数据通常由单引号（'...'）、双引号（"..."）或者三引号（'''...'''）、（"""..."""）包围，其中三引号包围的字符串可以由多行组成。通常情况下字符串中可以包含数字、字母、中文、特殊符号以及一些不可见的控制字符，如换行符、制表符等。

下面介绍Python中常用的字符串的操作方法。

（1）字符串常用方法

下面是常用的字符串方法。

• str.replace(oldValue,newValue)：字符串替换，使用新字符串newValue替换字符串str中的旧字符串oldValue。

• str.split(str1)：分割字符串，使用str1作为分隔符来分割字符串str。

• str.join(str1)：合并字符串（连接字符串）。

• str1.startswith(str2)：指定字符串str2是字符串str1的开头，返回True或False。

• str1.endswith(str2)：指定字符串str2是字符串str1的结尾，返回True或False。

• str1.find(str2)：返回字符串str1中第一次出现字符串str2的位置。

• str1.rfind(str2)：返回字符串str1中最后一次出现字符串str2的位置。

• str1.count(str2)：指定字符串str2在字符串str1中出现的次数。

• str1.strip(str2)：去除字符串str1中首尾指定的字符串str2。

• str1.lstrip(str2)：去除字符串str1中左边指定的字符串str2。

• str1.rstrip(str2)：去除字符串str1中右边指定的字符串str2。

• str.format()：格式化字符串。

• len(str)：获取字符串的长度。

（2）字符串大小写转换方法

下面是常用的字符串大小写转换方法。

• str.capitalize()：产生新的字符串，首字母大写。

• str.title()：产生新的字符串，每个单词的首字母大写。

• str.upper()：产生新的字符串，所有字符转换为大写。

• str.lower()：产生新的字符串，所有字符转换为小写。

• str.swapcase()：产生新的字符串，所有字母大小写转换（大写字母转换为小写，小写字母转换为大写）。

（3）字符串验证方法（以下方法均返回True或False）

下面是常用的字符串验证方法。

• str.isalnum()：是否为字母（A～Z/a～z）或数字（0～9）。

• str.isalpha()：检测字符串是否只由字母（A～Z/a～z）组成（含汉字）。

• str.isdigit()：检测字符串是否只由数字（0～9）组成。

• str.isspace()：检测是否为空白字符。

• str.isupper()：检测所有的字符是否为都大写字母。

• str.islower()：检测所有的字符是否为都小写字母。

• str.istitle()：检测字符串中的单词是否为首字母大写。

字符串操作实例代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 字符串
```

```
if __name__ == "__main__":
    str1 = "abcd*1234*aaaa*bbb*abcd*ABCabc"
    str2 = "Hello World!"
    print(str1.capitalize())                              # 将字符串的第一个字母大写
    print(str1.count('a'))                                        # 获得字符串中a的数目
    print(str1.find('abc'))                               # 获得字符串中abc的起始位置
    print("str1字符串长度：" , len(str1))
    print("连接字符串：", str1.join('你好'))
    print("拼接字符串：", str1 + str2)
    print("将字符串中的全部字母转换为大写：", str1.upper())
    print("首字母大写：", str1.title())
    print("以*分割字符串：", str1.split("*"))
    print("使用'*'号重复字符串：", str2*3)
    print("替换字符串：", str1.replace('abc', 'def'))
    #指定字符串str2是字符串str1的开头，返回True或False
    print(str1.startswith(str2))
    # 指定字符串str2是字符串str1的结尾，返回True或False
    print(str1.endswith(str2))
    print("abc在字符串str1中出现的最后位置：",str1.rfind('abc'))
    # 去除字符串str1中首尾指定的字符串str2
    print("去除字符串str1的首尾字符串abc: ",str1.strip('abc'))


    print(str2.isalnum())    # 检测字符串中是否包含0～9、A～Z、a～z
    print(str2.isalpha())    # 检测字符串中是否包含A～Z、a～z
    print(str2.isdigit())    # 检测字符串中是否仅包含数字
    print(str2.islower())    # 检测字符串是否均为小写字母
    print(str2.isspace())    # 检测字符串中的所有字符是否均为空白字符
    print(str2.istitle())    # 检测字符串中的单词是否为首字母大写
    print(str2.isupper())    # 检测字符串是否均为大写字母
```

在PyCharm下运行程序，运行结果如图8.22所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/code/python
Abcd*1234*aaaa*bbb*abcd*abcabc
7
0
str1字符串长度： 30
连接字符串： 你abcd*1234*aaaa*bbb*abcd*ABCabc好
拼接字符串： abcd*1234*aaaa*bbb*abcd*ABCabcHello World!
将字符串中的全部字母转换为大写： ABCD*1234*AAAA*BBB*ABCD*ABCABC
首字母大写： Abcd*1234*Aaaa*Bbb*Abcd*Abcabc
以*分割字符串： ['abcd', '1234', 'aaaa', 'bbb', 'abcd', 'ABCabc']
使用'*'号重复字符串： Hello World!Hello World!Hello World!
替换字符串： defd*1234*aaaa*bbb*defd*ABCdef
False
False
abc在字符串str1中出现的最后位置： 27
去除字符串str1的首尾字符串abc： d*1234*aaaa*bbb*abcd*ABC
False
False
False
False
False
True
False

Process finished with exit code 0
```

图8.22　运行结果

4. 算法分析

我们使用最简单的方法来实现，定义一个fun()函数，在函数中遍历这个字符串，将字符串拆分开，并将每个字符分别存入数组，然后遍历这个数组，判断每个数组字符元素是否是“*”，如果是就替换为空，然后返回字符串结果。

其中fun()函数定义如下：

```
def fun(s):
    a = [0] * len(s)
    for i in range(len(s)):
        a[i] = s[i]                                    # 将字符串拆分存入数组
    for j in a:
        if j == '*':
            j = ""
        print(j, end="")
```

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 删除“*”符号
"""
实例：输入：  ****A*BC*DEF*G********
      输出：  ****ABCDEFG********
"""

def fun(s):
    a = [0] * len(s)
    for i in range(len(s)):
        a[i] = s[i]                                                    # 将字符串拆
分存入数组
    print("结果：", end="")
    for j in a:
        if j == '*':
            j = ""
        print(j, end="")


if __name__ == "__main__":
    s = str(input("请输入一个只包含字母和*号的字符串：\n"))
    print("输入的字符串为：", s)
    fun(s)
```

6. 运行结果

在PyCharm下运行程序，结果如图8.23所示。

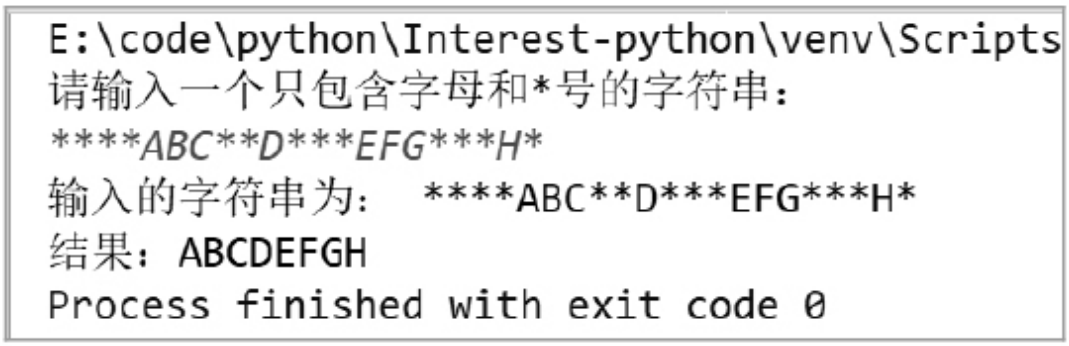


图8.23　运行结果

7. 其他方法

本题除了使用数组方法实现之外，还可以使用正则表达式来实现，这里仅提供一个参考实例，代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 删除“*”符号  正则法

import re

# 删除指定位置的“*”
def fun(s):
    result_str = re.sub(r'([a-zA-Z]+)(.+)([a-zA-Z]+)', lambda match:
''.join([m.replace('*', '') for m in match.groups()]), s)
    print(result_str)

if __name__ == "__main__":
    s = str(input("请输入一个只包含字母和*号的字符串：\n"))
    print("输入的字符串为：", s)
    fun(s)
```

在PyCharm下运行程序，结果如图8.24所示。

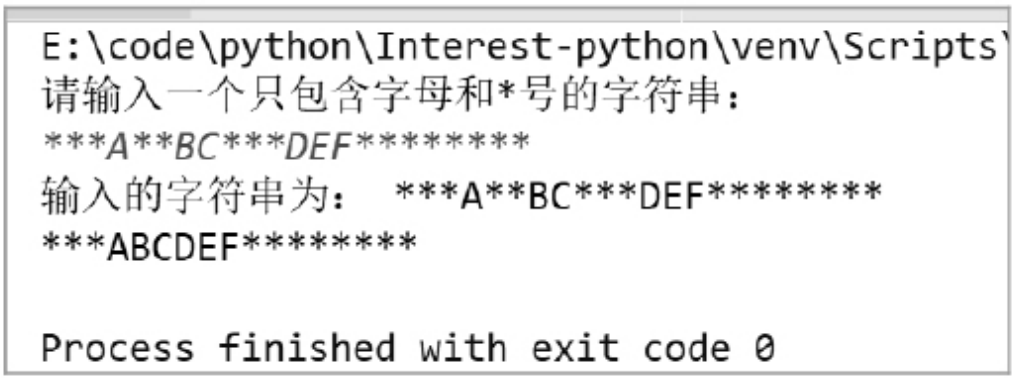


图8.24　运行结果

8. 拓展训练

规定输入的字符串只包含字母和*号。请编写程序，实现将字符串前面连续的“*”全部删除，中间和尾部的“*”不删除。例如，字符串的内容为：*******A*BC*DEF*G****，删除后，字符串的内容应当是：A*BC*DEF*G****。

8.10　在指定位置插入字符

1. 问题描述

请编写程序，实现以下功能：在字符串中的所有数字字符前加一个“$”符号。例如，输入A1B23CD45，输出A$1B$2$3CD$4$5。

2. 问题分析

在字符串S的所有数字字符前加一个“$”符号，可以使用字符串函数结合数组来实现该功能。首先遍历这个字符串，将字符串的字符元素存入数组，然后遍历数组元素，利用字符串的isdigit()函数来判断这个元素是否是数字，如果是，就在该数字的前面添加上“$”符号。其对应的代码如下：

```
def insert_str(s):
    a = [0] * len(s)
    for i in range(len(s)):                    # 遍历字符串
        a[i] = s[i]                                    # 将字符串存入列表数组

    # 遍历数组元素
    for i in a:
        # 用isdigit()函数判断是否数字
        flag = i.isdigit()                             # 如果是数字，返回True
        if flag == True:
            i = '$'+i
        print(i, end="")
```

3. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 在指定位置插入字符

def insert_str(s):
    a = [0] * len(s)
    for i in range(len(s)):                    # 遍历字符串
        a[i] = s[i]                                    # 将字符串存入列表数组

    # 遍历数组元素
    for i in a:
        # 用isdigit()函数判断是否数字
        flag = i.isdigit()                         # 如果是数字返回True
```

```
            if flag == True:
                i = '$'+i
            print(i, end="")

if __name__=="__main__":
    s = str(input("请输入一个字符串："))
    print("输入的字符串为：", s)
    insert_str(s)
```

4. 运行结果

在PyCharm下运行程序，结果如图8.25所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe
请输入一个字符串：1a2b3d4D5665FDER765F654431DDD5K
输入的字符串为： 1a2b3d4D5665FDER765F654431DDD5K
$1a$2b$3d$4D$5$6$6$5FDER$7$6$5F$6$5$4$4$3$1DDD$5K
Process finished with exit code 0
```

图8.25　运行结果

第9章　趣味函数递归

如果在调用一个函数的过程中存在直接或间接地调用函数自身的情况，则称为函数的递归调用。在Python语言中是允许函数的递归调用的。

在递归问题的求解过程中关键是找到递归公式以及递归结束的条件。只有有了递归结束条件，才能使递归过程经过有限的步骤后正常结束。

本章选用几个有趣的递归问题进行分析，通过对问题的讲解将递归的概念和思考方法渗透其中，这些递归问题都各有特点，但其中包含的递归思想是相同的。因此，通过对本章的学习，相信读者对递归的概念能有更为深入的理解，并能够独立分析和解决相关问题。

9.1　猴子吃桃

1. 问题描述

一个猴子摘了一些桃子，它第一天吃掉了其中的一半然后再多吃了一个，第二天照此方法又吃掉了剩下桃子的一半加一个，以后每天如此，直到第十天早上，猴子发现只剩下一个桃子了，问猴子第一天总共摘了多少个桃子？

2. 问题分析

假设Ai为第i天吃完后剩下的桃子的个数，A0表示第一天共摘下的桃子，显然，本题要求的是A0。根据问题描述，前后相邻两天之间的桃子数应存在如下关系：

```
A(i+1)=Ai-(Ai/2+1)
```

此式可转换为：

```
Ai=2 A(i+1)+2=2(A(i+1)+1)
```

因此，有以下递推式子：

```
A0 = 2×(A1+1)     A1：第1天吃完后剩下的桃子数
A1 = 2×(A2+1)     A2：第2天吃完后剩下的桃子数
……
A8 = 2×(A9+1)     A9：第9天吃完后剩下的桃子数
A9 = 1
```

由于第9天吃完后剩下的桃子数是已知的，因此根据上述递推式子可以推出第8天的桃子总数；根据第8天吃完后剩下的桃子数又可以推出第7天的桃子总数……重复进行下去，就可以推出第1天摘下的桃子总数。推导过程如表9.1所示。

表9.1　推导过程表

天数*i*	第*i*天吃完后剩下的桃子数
9	A9 = 1
8	A8 = 2×(A9+1)=4
7	A7 = 2×(A8+1)=10
6	A6 = 2×(A7+1)=22
5	A5 = 2×(A6+1)=46
4	A4 = 2×(A5+1)=94
3	A3 = 2×(A4+1)=190
2	A2 = 2×(A3+1)=382
1	A1 = 2×(A2+1)=766
0	A0 = 2×(A1+1)=1534

3. 算法设计

以上递推过程可分别用循环结构和递归函数实现。先使用递归函数来实现。

将上述递推关系进行描述：假设第n天吃完后剩下的桃子数为A(n)，第n+1天吃完后剩下的桃子数为A(n+1)，则存在递推关系A(n)=(A(n+1)+1)*2。这种递推关系可以用递归函数实现。

4. 确定程序框架

递归函数代码如下：

```
# 递推公式：A(n) = (A(n+1) + 1) * 2
def A(n):
    if n >= 9:
        return 1                                                    #
递归出口
    else:
        return (2 * (A (n+1) + 1))                  # 递推公式
```

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 猴子吃桃

# 递推公式：A(n) = (A(n+1) + 1) * 2
```

```
def A(n):
    if n >= 9:
        return 1                                                    # 递归出口
    else:
        return (2 * (A(n+1) + 1))                                   # 递推公式

if __name__ == "__main__":
    print("猴子第一天总共摘了 %d 个桃子" %A(0))
```

6. 运行结果

在PyCharm下运行程序，结果如图9.1所示。

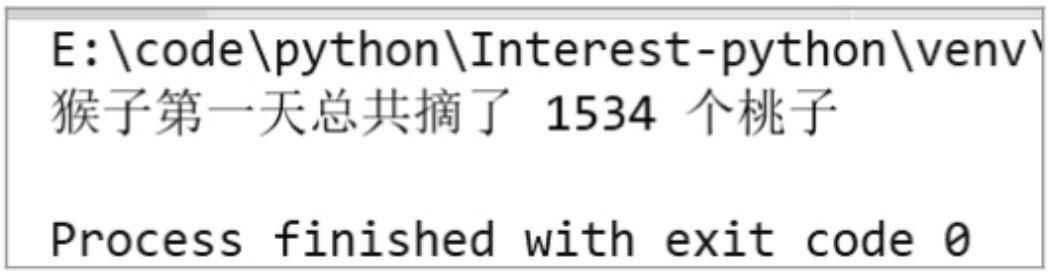


图9.1　运行结果

7. 问题拓展

该问题还可以使用循环结构求解，代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 猴子吃桃

if __name__ == "__main__":
    day = 9
    x2 = 1
    while day > 0:
        x1 = (x2 + 1) * 2            # 第1天的桃子数是第2天桃子数加1后的2倍
        x2 = x1
        day -= 1
    print("猴子第一天总共摘了 %d 个桃子" %x1)
```

9.2　杨辉三角形

1. 问题描述

在屏幕上打印杨辉三角形。杨辉三角形，又称贾宪三角形、帕斯卡三角形，是二项式系数在三角形中的一种几何排列。

图9.2显示了杨辉三角的前7行。

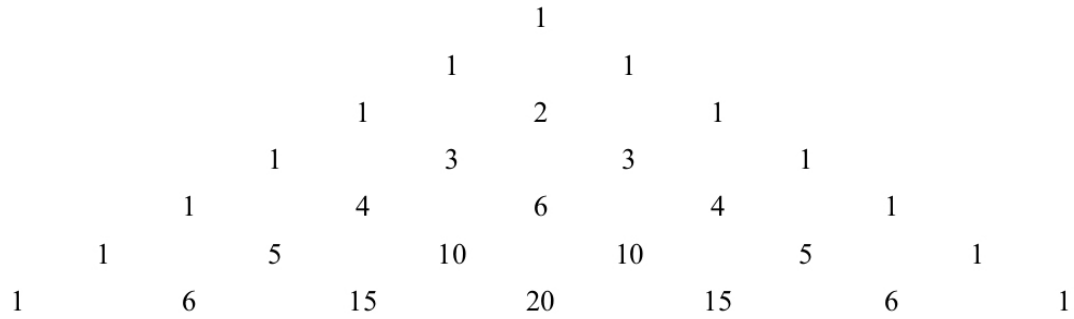


图9.2　杨辉三角形的前7行

2. 问题分析

杨辉三角形中的数，正是(x+y)的N次方幂展开式各项的系数。本题作为程序设计中具有代表性的题目，求解的方法很多，下面以递归的方法来打印杨辉三角形。

从杨辉三角形的特点出发，可以总结出：

1）第x行有x个值（设起始行为第1行）。

2）对于第x行的第y（y＞＝3）个值，有：当y=1或y=x时，其值为1；当y!=1且y!=x时，其值为第x-1行的第y-1个值与第x-1行的第y个值之和。

将这些特点提炼成数学公式，则位于杨辉三角第x行第y列的值为：

$$c(x,y)=\begin{cases}1 & y=1 \text{ 或 } y=x \\ c(x-1,y-1)+c(x-1,y) & y\neq 1 \text{ 且 } y\neq x\end{cases}$$

3. 确定程序框架

根据问题分析中得到的数学公式写出递归函数，代码如下：

```
# 杨辉三角递归函数
def c(x, y):
    if y == 1 or y == x:                    # y=1或y=x时，函数返回值为1
        return 1
    else:
        z = c(x-1, y-1) + c(x-1, y)         # y为其他值时的递推公式
        return z
```

4. 完整的程序

根据上面的分析，完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 杨辉三角形

# 杨辉三角形递归函数
def c(x, y):
    if y == 1 or y == x:                                # y=1或y=x时，函数返回值为1
        return 1
    else:
        z = c(x-1, y-1) + c(x-1, y)         # y为其他值时的递推公式
        return z


if __name__ == "__main__":
    n = int(input("请输入杨辉三角的行数: "))
    for i in range(1, n+1):                             # 输出n行
        for j in range(0, n-i+1):
            print("   ", end=" ")
        for j in range(1, i+1):
            # 调用递归函数，输出第i行的第j个值
            print("%6d  " %(c(i, j)), end=" ")
        print()
```

5. 运行结果

在PyCharm下运行程序，屏幕上提示“请输入杨辉三角的行数：”，这里输入10，打印出的杨辉三角形如图9.3所示。

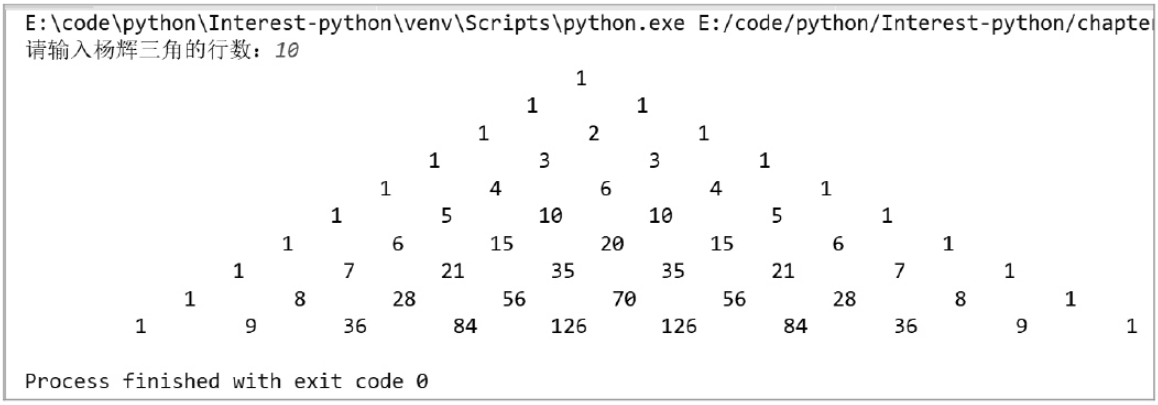


图9.3　运行结果

6. 问题拓展

使用二维数组打印杨辉三角形。

由于位于杨辉三角形两个腰上的数都为1，其他位置上的数等于它肩上两个数之和，基于杨辉三角形的这个特点，就可以使用二维数组打印出杨辉三角形。

先定义二维数组a[N][N]，N为常量，大于要打印的行数n。再将每行的第一个数和最后一个数赋值为1，即a[i][1]=a[i][i]=1。除了每行的第一个数和最后一个数以外，每行上的其他数都为其肩上的两数之和，即a[i][j]=a[i-1][j-1]+a[i-1][j]。

（1）计算杨辉三角形中的数值并存入二维数组

定义row和column两个变量分别代表杨辉三角形的行和列，变量n表示要打印的行数。

```
# 计算杨辉三角中的数值并存入二维数组a中
for row in range(1, n+1):
    # 令每行两边的数为1，循环从1开始，每行第一个数存放在a[row][1]中
    a[row][1] = a[row][row] = 1
for row in range(3, n+1):
    for column in range(2, (row-1)+1):
        # 计算其他位置的值并存入二维数组
        a[row][column] = a[row-1][column-1] + a[row-1][column]
```

（2）打印空格

在每行输出之前，先打印空格占位，可使输出更美观。

第1行打印3(n-1)个空格，第2行打印3(n-2)个空格……，第k行打印3(n-k)个空格。

```
for row in range(1, n+1):
    for k in range(1, (n-row)+1):
        print("   ", end="")                    # 在每行输出数之前先打印空格占位，使输出更美观
```

（3）打印杨辉三角形中的数

输出杨辉三角形每一行之前都先打印空格，之后再使用下面的代码输出每行中的数值。

```
# 打印杨辉三角形
for row in range(1, n+1):
    for k in range(1, (n-row)+1):
        print("   ", end="")          # 在每行输出数之前先打印空格占位，使输出更美观
    # column<=row表示不输出数组中其他的数，只输出所需的数
    for column in range(1, row+1):
        print("%6d" %(a[row][column]), end=" ")
    print()                                                  # 当一行输出完以后换行继续下一行的输出
```

（4）完整的程序

现在我们就需要把刚才的程序进行组合，构成完整的程序。

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 杨辉三角形

if __name__ == "__main__":
    n = 0
    a = [([0] * 14) for i in range(14)]      # 定义一个行为14、列为14的二维数组
    while n <= 0 or n >= 13:                                 # 控制打印的行数，行数过大会造成显示不规范
        n = int(input("请输入杨辉三角的行数："))
    print("打印 %d 行杨辉三角如下：" %n)
    # 计算杨辉三角中的数值并存入二维数组a中
    for row in range(1, n+1):
        # 令每行两边的数为1，循环从1开始，每行第一个数存放在a[row][1]中
        a[row][1] = a[row][row] = 1
    for row in range(3, n+1):
        for column in range(2, (row-1)+1):
            # 计算其他位置的值并存入二维数组
            a[row][column] = a[row-1][column-1] + a[row-1][column]
    # 打印杨辉三角形
    for row in range(1, n+1):
        for k in range(1, (n-row)+1):
```

```
            print("    ", end="")      # 在每行输出数之前先打印空格占位，使输出更美观
        # column<=row表示不输出数组中其他的数，只输出所需的数
        for column in range(1, row+1):
            print("%6d" %(a[row][column]), end=" ")
        print()                                          # 当一行输出完以后换行继续下一行
的输出
```

（5）运行结果

在PyCharm下运行程序，屏幕上提示“请输入杨辉三角的行数：”，这里输入8，打印出的杨辉三角形如图9.4所示。

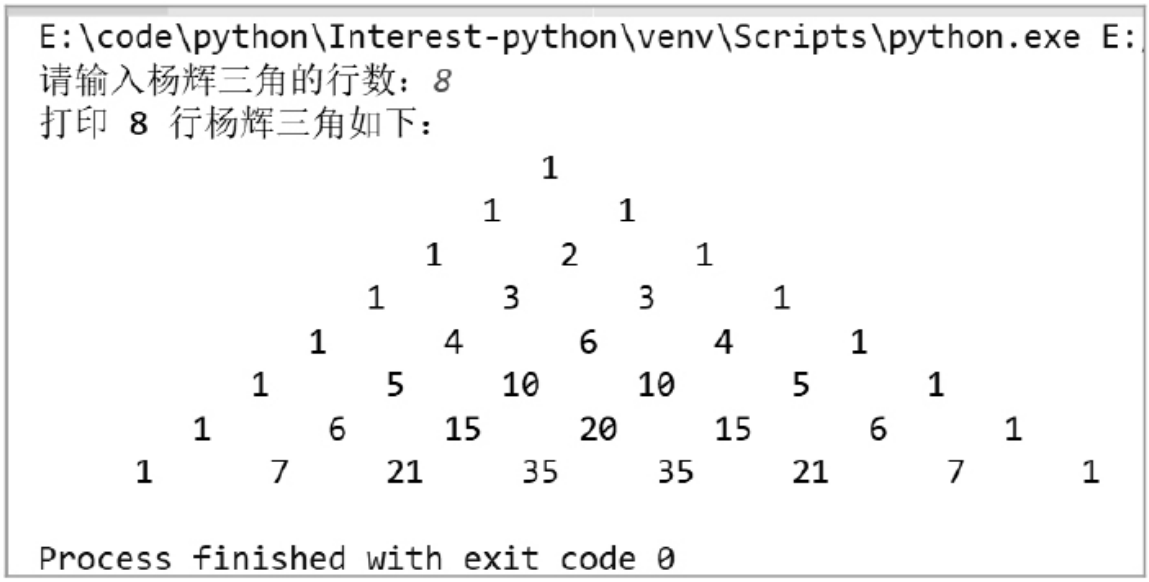


图9.4　打印8行的杨辉三角形

总结：　除了本节介绍的两种打印杨辉三角形的方法外，还有很多其他的方法，读者可以自己思考其解法。

9.3　卡布列克常数

1．问题描述

对于任意一个4位数n，进行如下的运算：

1）将组成该4位数的4个数字由大到小排列，形成由这4个数字构成的最大的4位数。

2）将组成该4位数的4个数字由小到大排列，形成由这4个数字构成的最小的4位数（如果4个数中含有0，则得到的数不足4位）。

3）求这两个数的差，得到一个新的4位数（高位0保留）。

这称为对n进行了一次卡布列克运算。

存在这样一个规律：对一个各位数字不全相同的4位数重复进行若干次卡布列克运算，最后得到的结果总是6174，这个数被称为卡布列克数。

例如：

6543-3456=3087

8730-378=8325

8532-2358=6174

7641-1467=6174

要求编程来验证卡布列克常数。

2．问题分析

定义函数kblk(n)，该函数表示可对n进行卡布列克运算，直至结果为6147或0为止，其中0表示各位完全相同时所得到的结果。函数kblk(n)对n进行一次卡布列克运算可以得到数值num，如果num既不是6147也不是0，则递归调用函数kblk(n)将num作为参数传递给它，即kblk(num)。注意，递归函数的出口最终结果为6147或0。

3. 算法设计

根据问题分析可知，该算法应该包括如下几个功能：

1）分解4位整数，并保存每位上的数字。

2）对分解后的4个数字分别求出其构成的最大值和最小值。

3）进行卡布列克运算。

这几个功能，我们可以分别通过不同的函数来实现，其中进行卡布列克运算的函数为递归函数。

4. 确定程序框架

1）分解4位整数，并保存每位上的数字。定义数组array[4]，用来存放分解后4位整数num中每位上的数字。分解4位整数的代码如下：

```
# 将输入的4位数拆分存入数组
def createArray(n):
    # 定义数组array[4]，用来存放分解后4位整数num中每位上的数字
    array = [0] * 4
    # 分解整数n
    a = n // 1000                                                    # 千位
    b = (n % 1000) // 100                                      # 百位
    c = ((n % 1000) % 100) // 10                        # 十位
    d = ((n % 1000) % 100) % 10                         # 个位
    array[0] = a
    array[1] = b
    array[2] = c
    array[3] = d
    # print(array)
    return array
```

上面的代码中，采用取整求余的方法将4位整数拆分开，将拆分的数字分别存入数组array中，其中千位数字存入array[0]中，百位数字存入array[1]中，十位数字存入array[2]中，个位数字存入array[3]中。

2）对分解后的4位整数分别求出其构成的最大值和最小值。定义函数array_max(array)来求出分解后的4位数字的最大值。代码如下：

```
# 求数组的最大值
def array_max(array):
    array.sort()
    maxNum = array[3]*1000 + array[2]*100 + array[1]*10 + array[0]
    # print("maxNum = %d" %maxNum)
    return maxNum
```

定义函数array_min(array)来求出分解后的4位数字的最小值。代码如下：

```
# 求数组的最小值
def array_min(array):
    array.sort()
    minNum = array[0]*1000 + array[1]*100 + array[2]*10 + array[3]
    # print("minNum = %d" %minNum)
    return minNum
```

3）进行卡布列克运算。定义kblk()函数实现卡布列克计算，在该函数中可直接调用上面实现的几个功能函数。

```
# 递归求卡布列克常数
def kblk(n, count):
    if n != 6174 and n != 0:          # 若不等于6174且不等于0，则进行卡布列克运算
        array = createArray(n)        # 将4位整数分解，并将各位数字存入array数组中
        max = array_max(array)        # 求各位数字组成的最大值
        min = array_min(array)        # 求各位数字组成的最小值
        num = max - min               # 求最大值和最小值的差
        count += 1
        # 输出该步的计算过程
        print("[%d]: %d-%d=%d\n" % (count, max, min, num));
        kblk(num, count);             # 递归调用，继续进行卡布列克运算
```

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 卡布列克常数

# 将输入的4位数拆分存入数组
def createArray(n):
    # 定义数组array[4]，用来存放分解后4位整数num中每位上的数字
    array = [0] * 4
    # 分解整数n
    a = n // 1000                                   # 千位
    b = (n % 1000) // 100                           # 百位
    c = ((n % 1000) % 100) // 10                    # 十位
    d = ((n % 1000) % 100) % 10                     # 个位
    array[0] = a
    array[1] = b
    array[2] = c
    array[3] = d
    # print(array)
    return array

# 求数组的最大值
def array_max(array):
    array.sort()
    maxNum = array[3]*1000 + array[2]*100 + array[1]*10 + array[0]
    # print("maxNum = %d" %maxNum)
    return maxNum

# 求数组的最小值
def array_min(array):
    array.sort()
    minNum = array[0]*1000 + array[1]*100 + array[2]*10 + array[3]
    # print("minNum = %d" %minNum)
    return minNum

# 递归求卡布列克常数
def kblk(n, count):
    if n != 6174 and n != 0:                # 若不等于6174且不等于0，则进行卡布列克运算
        array = createArray(n)              # 将4位整数分解，各位数字存入array数组中
        max = array_max(array)              # 求各位数字组成的最大值
        min = array_min(array)              # 求各位数字组成的最小值
        num = max - min                     # 求最大值和最小值的差
        count += 1
        # 输出该步的计算过程
        print("[%d]: %d-%d=%d\n" % (count, max, min, num));
        kblk(num, count);                   # 递归调用，继续进行卡布列克运算


if __name__ == "__main__":
    n = 1000
    count = 0
    while n <= 1000 or n > 9999:
        n = int(input("请输入一个4位整数："))
    kblk(n, count)
```

6. 运行结果

在PyCharm下运行程序，结果如图9.5所示。

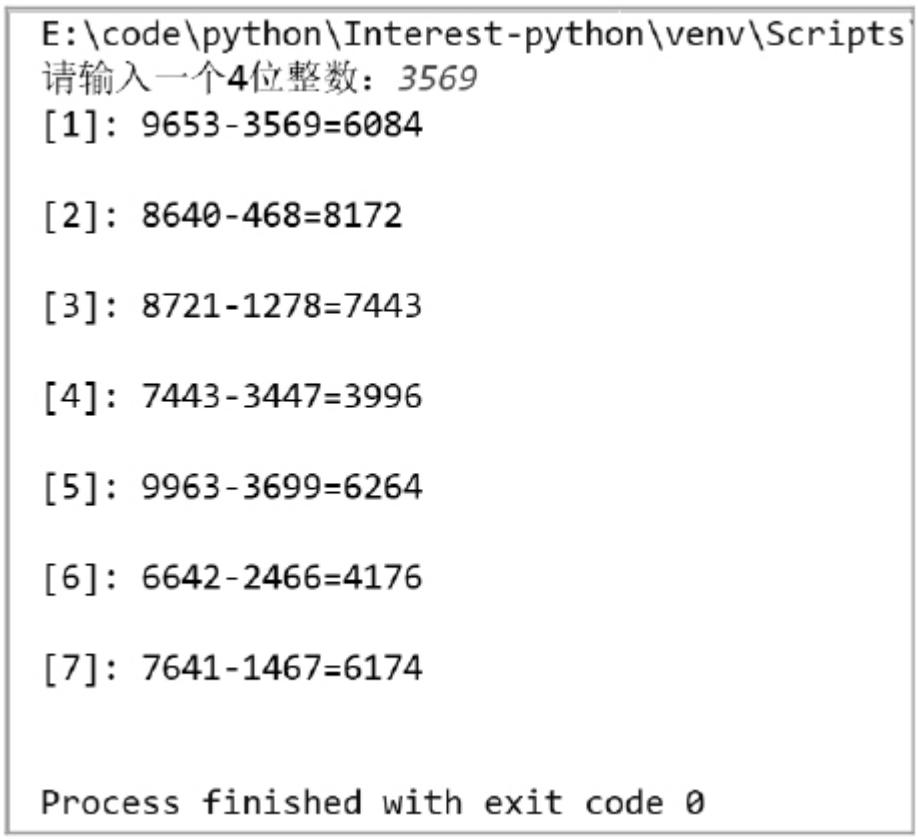


图9.5　运行结果

在图9.5中，我们随便输入了一个4位整数3569，对该整数共进行了7次卡布列克运算，最后得到的结果为6147，因此验证成功。读者可以输入其他4位整数来对卡布列克常数进行验证。

9.4　递归解决年龄问题

1. 问题描述

有5个人坐在一起，问第5个人多少岁，他说比第4个人大2岁。问第4个人多少岁，他说比第3个人大2岁。问第3人多少岁，他说比第2个人大2岁。问第2个人多少岁，他说比第1个人大2岁。最后问第1个人多少岁，他说他是10岁。编写程序，求出当输入某个人时其对应的年龄。

2. 问题分析

在分析该问题前先介绍函数递归调用的基础知识。

1）函数递归调用的定义：如果在调用一个函数的过程中又出现直接或间接地调用该函数本身，则称为函数的递归调用。Python语言中允许函数进行递归调用。

2）程序中递归调用的方式：直接递归调用，即函数直接调用本身。例如：

```
def f(x):
    ......
    y = f(n)                                                    # 调用自身
    .....
    return y
```

在调用函数f()的过程中，又要调用f()函数，这就是函数的直接递归调用。

间接递归调用，即函数间接调用本身。例如：

```
def f1(x):
......
    z = f2(y)
    ......

    return 2*z
```

```
def f2(t):
    .....
    c = f1(a)
    .....

    return 3+c
```

在上面的代码中定义了两个函数f1()和f2()。在调用f1()函数的过程中，f1()又调用了函数f2()，而在调用函数f2()的过程中，又调用了f1()。

注意： 在递归调用中不能出现无终止的调用，只能是有限次的递归调用，即必须有递归结束条件。因此，在代码中一定要有控制递归调用终止的语句。一般可以采用if语句来控制只有在某一条件成立时才继续执行递归调用，否则不再继续递归。

了解了递归调用的基础知识后，现在对本节中提出的问题进行分析。

该问题是一个递归问题。要求出第5个人的年龄，则必须先知道第4个人的年龄，显然第4个人的年龄也是未知的，但可以由第3个人的年龄推算出来。而想知道第3个人的年龄，又必须先知道第2个人的年龄，而第2个人的年龄取决于第1个人的年龄。

又已知每个人的年龄都比其前一个人的年龄大2，因此根据题意，可得到如下几个表达式：

```
age(5)=age(4)+2
age(4)=age(3)+2
age(3)=age(2)+2
age(2)=age(1)+2
age(1)=10
```

归纳上面的5个表达式，用数学公式表达为：

$$age(n)=\begin{cases} age(n-1)+2 & n>1 \\ 10 & n=1 \end{cases} \quad ①$$

在上面的公式中，当n＞1时，求第n个人的年龄是通过一个公式来表达的。根据式①，我们可以画图来表示求第n个人年龄的过程，如图9.6所示即为求解第5个人年龄的过程。

求解第n个人的年龄分成两个阶段。第一个阶段是“回推”过程，第二个阶段是“递推”过程。

在“回推”过程中，利用的是n＞1时的公式age(n-1)+2。要求的是第n个人的年龄，因此首先将第n个人的年龄回推到第(n-1)个人的年龄，但第n-1个人的年龄仍然未知，因此需要继续回推到第(n-2)个人的年龄，第(n-2)个人的年龄仍然未知，需要继续向前回推，如此下去，一直回推到第1个人的年龄。而第一个人的年龄是已知的，因此，第一阶段的“回推”过程结束，开始第二阶段的“递推”。

在“递推”过程中，从第1个人的年龄可以推出第2个人的年龄，从第2个人的年龄可以推出第3个人的年龄，如此下去一直递推到第5个人的年龄。

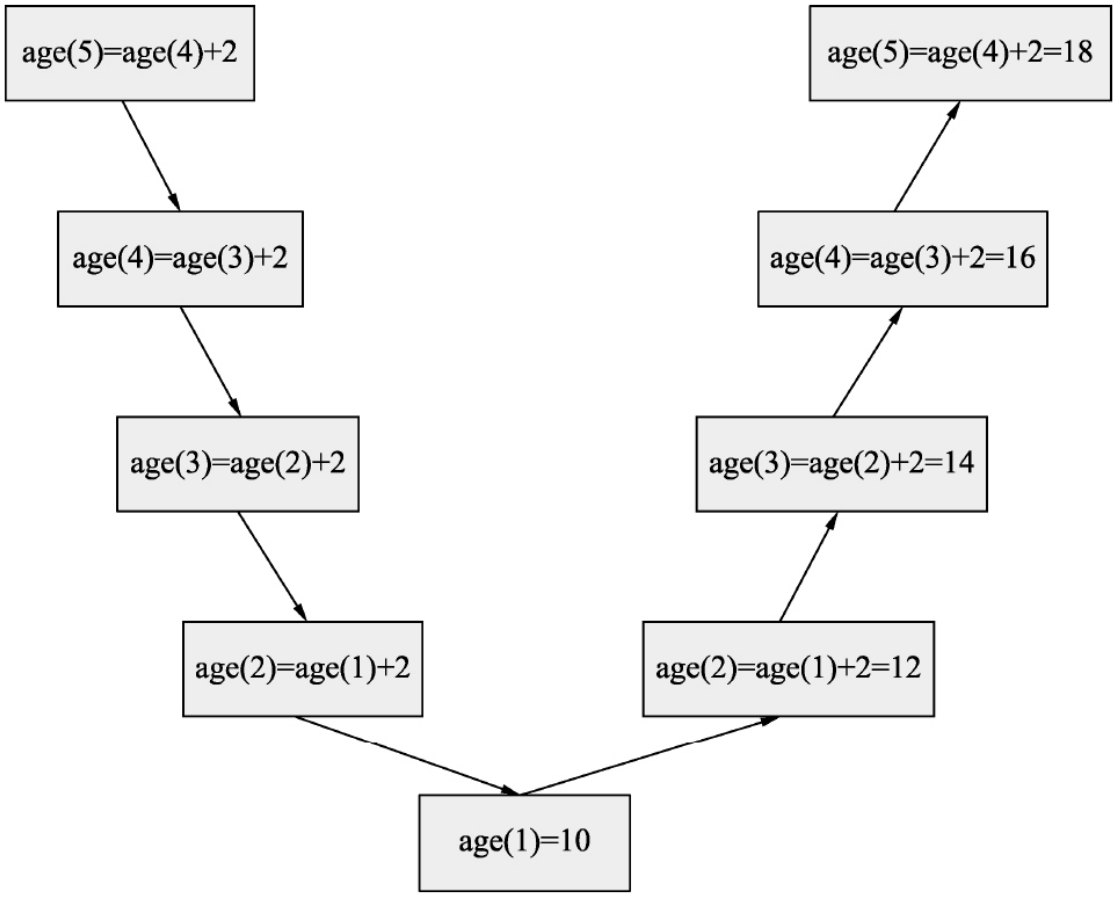


图9.6　年龄求解过程示意图

在图9.6中，通过“回推”和“递推”两个过程，最终可得到第5个人的年龄为18岁。

3．算法设计

理解了问题分析中的递归处理过程后，算法设计就非常简单了。只需要将式①转换成一个函数，然后在main()函数中调用它就可以了。

4．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 递归解决年龄问题
```

```
#递归计算年龄
def age(n):
    if n == 1:
        x = 10
    else:
        x = age(n-1) + 2
    return x

if __name__ == "__main__":
    n = int(input("请输入n值: "))                                    # n表示第几个人
    # 调用age()函数，计算第n个人的年龄
    print("第 %d 个人的年龄为: %d" %(n, age(n)))
```

程序分析：

虽然问题分析过程比较复杂，但该问题的程序代码非常简单，在main()函数中通过调用age(n)函数就可以获知第n个人的年龄。

age()函数总共被调用了n次，其中age(n)是由main()函数进行调用的，而其余的n-1次都是由age()函数自身调用的，即age()函数递归调用了n-1次。

假设n=5，则age()函数总共被调用了5次，age()函数递归调用了4次。

要注意的是，每次调用age()函数时并不会马上获得年龄值，而是不断地进行递归调用，直到调用到age(1)时才有确定的年龄值，然后再从age(1)一步步地递推回去。

age(n)函数的调用过程如图9.7所示。

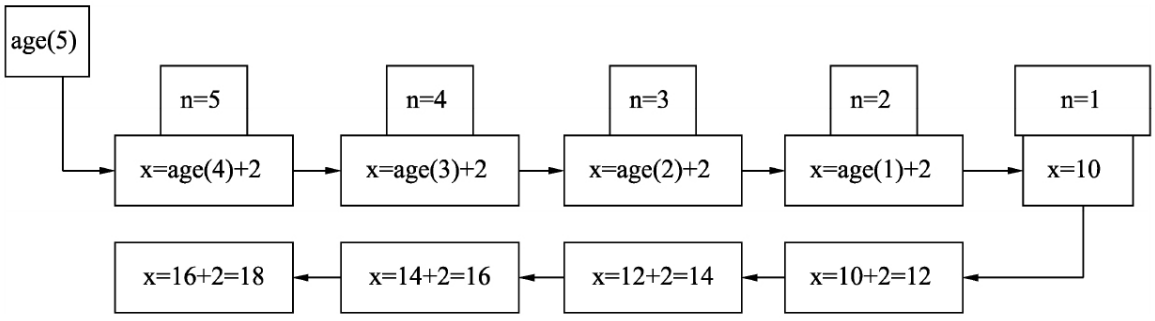


图9.7　age(n)函数的调用过程

5. 运行结果

在PyCharm下运行程序，屏幕上提示“请输入n值：”，对该题而言，n的最大取值为整数5。此处输入n的值为5，则计算出第5个人的年龄为18。运行结果如图9.8所示。

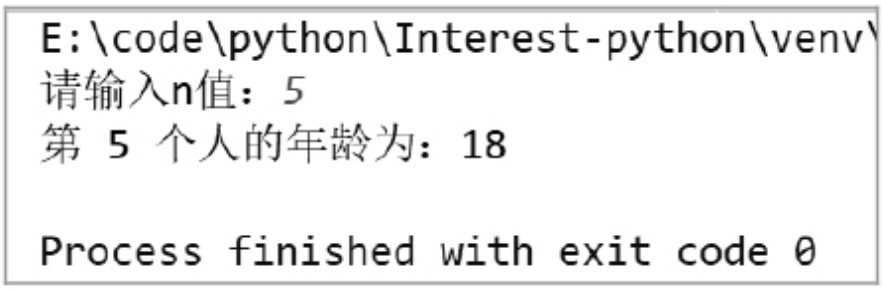


图9.8　运行结果

6. 问题拓展

由该题的分析过程可知，递归的问题都可以分为“回推”和“递推”两个阶段，而且必须存在一个能够结束递归过程的条件，如本题中的age(1)=10，否则递归过程会无限制地进行下去而无法结束。

下面对递归法做一总结。

递归是设计和描述算法的一种强有力的工具。能够采用递归来描述的算法通常具有一些特征：为求解规模为n的问题，首先要将它分解成规模较小的问题，然后通过这些小问题的解，能够方便地构造出大问题的解，同时，这些规模较小的问题也能够采取同样的分解方法分解成规模更小的问题，并能够通过这些更小问题的解构造出规模较大问题的解。特别地，当问题规模n=0或n=1时，能直接获得问题的解。

递归算法的执行过程分为“回推”和“递推”两个阶段。

• 在回推阶段，是把较复杂的问题（规模为n）的求解递推到比原问题简单一些的问题（规模小于n）的求解。例如本例中，要求解age(n)，先把它递推到求解age(n-1)，而要计算age(n-1)，又必须先计算age(n-2)，以此类推，直到age(1)为止。需要注意的是，在

递推阶段，必须要有能够终止递归的条件。如本例中n为1时，递推可终止。

• 在递推阶段，当获得最简单情况的解时，如本题中得到age(1)的值，逐级返回，依次得到较复杂问题的解，最终获得所求问题的解。

注意： 在编写递归函数时需要注意，函数中定义的局部变量和形式参数只在当前的调用层有效，当回推到简单问题时，原来调用层中的局部变量和参数都被隐藏起来。每一个简单问题层中都有自己的局部变量和参数。

9.5　递归解决分鱼问题

1．问题描述

A、B、C、D、E这5个人合伙夜间捕鱼，凌晨时都已经疲惫不堪，于是各自在河边的树丛中找地方睡着了。第二天日上三竿时，A第一个醒来，他将鱼平分为5份，把多余的一条扔回河中，然后拿着自己的一份回家去了；B第二个醒来，但不知道A已经拿走了一份鱼，于是他将剩下的鱼平分为5份，扔掉多余的一条，然后只拿走了自己的一份；接着C、D、E依次醒来，也都按同样的办法分鱼。问这5个人至少合伙捕到多少条鱼？每个人醒来后所看到的鱼是多少条？

2．问题分析

假设5个人合伙捕了x条鱼，则“A第一个醒来，他将鱼平分为5份，把多余的一条扔回河中，然后拿着自己的一份回家去了”之后，还剩下4(x-1)//5条鱼。

这里实际包含了一个隐含条件：假设Xn为第n（n=1、2、3、4、5）个人分鱼前鱼的总数，则(Xn-1)//5必须为正整数，否则不合题意。如果(Xn-1)//5为正整数，则(Xn-1)%5==0必须成立。

又根据题意，应该有下面等式：

```
X4=4(X5-1) // 5
X3=4(X4-1) // 5
X2=4(X3-1) // 5
X1=4(X2-1) // 5
```

则一旦给定X5，就可以依次推算出X4、X3、X2和X1的值。要保证X5、X4、X3、X2和X1都满足条件(Xn-1)%5==0，此时的X5就为5个人合伙捕到的鱼的总条数。显然，5个人合伙可能捕到的鱼的条数

并不唯一，但题目中强调了“至少”合伙捕到的鱼，此时题目的答案是唯一的。该问题可使用递归的方法求解。

3. 确定程序框架

在main()函数中构建一个不定次数的while循环，这里使用一个死循环来实现。定义变量x表示5个人合伙可能捕到的鱼的条数，我们可以取x的最小值为6，让x的值逐渐增加，x每一次取值，都增加5，直到找到一个符合问题要求的答案。由于题目中问“这5个人至少合伙捕到多少条鱼”，而我们找到的第一个x值就是5个人至少捕到的鱼的总条数。

```
if __name__ == "__main__":
# 变量定义及初始化
    while True:
        x = x+5
    # 将x传入分鱼递归函数进行检验
    # 找到第一个符合题意的x则退出循环
```

通过这个循环，就可以对每一个x的可能情况进行检查。当然，是通过调用分鱼的递归函数来进行检查的。

分鱼的递归函数如下：

```
# 分鱼递归方法
def fish(n, x):
    if (x-1) % 5 == 0:
        if n == 1:
            return 1                                                    #
递归出口
        else:
            return fish(n-1, (x-1)//5 * 4)                  # 递归调用
    return 0
# x不是符合题意的解，返回0
```

fish()函数中包含了两个参数：n和x。n表示参与分鱼的人数，x表示n个人分鱼前鱼的总条数。这两个参数都是由main()函数中传递进来的。

根据前面的分析，当n=5时，(x-1)%5==0必须成立，否则该x值不是满足题意的值，退出fish()函数，返回到main()函数，main()

函数中再传递新的x值到fish()中进行检验。如果(x-1)%5==0条件成立，则要判断n=4时，(x-1)%5==0条件是否成立，需要注意的是，此时的形参x是4个人分鱼前鱼的总条数，即fish(5,x)递归调用fish(4,(x-1)//5*4)。这样依次进行下去，直到n=1时，(x-1)%5==0条件仍成立，则说明开始从main()函数中传递进来的x值是符合题意要求的一个值，可以逐层从递归函数中返回，每次返回值都为1，直至返回到main()函数。

4. 完整的程序

根据上面的分析，完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 递归解决分鱼问题

# 分鱼递归方法
def fish(n, x):
    if (x-1) % 5 == 0:
        if n == 1:
            return 1                                    # 递归出口
        else:
            return fish(n-1, (x-1)//5 * 4)  # 递归调用
    return 0                                                    # x 不是符合题意的解，返回0

if __name__ == "__main__":
    # 变量定义及初始化
    i = 0
    flag = 0
    while True:
        if flag != 1:
            i += 1
            x = i*5 + 1                                 # x最小值为6，以后每次增加5
            if(fish(5, x)):                             # 将x传入分鱼递归函数进行检验
                flag = 1                                # 找到第一个符合题意的x，则置标志位为1
                print("五个人合伙捕到的鱼总数为%d" %x);
                exit()                                  # 结束程序
```

5. 运行结果

在PyCharm下运行程序，结果如图9.9所示。由图9.9可知，5个人合伙捕到的鱼的总条数为3121条。

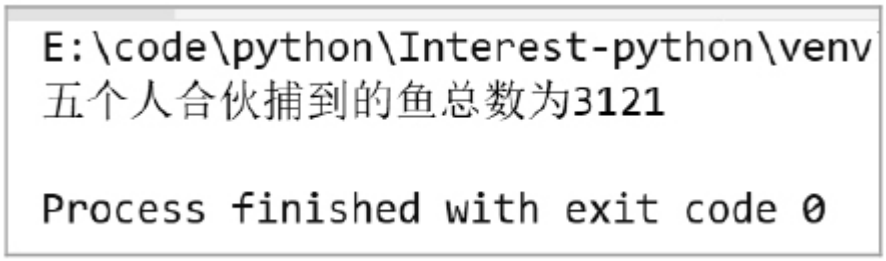


图9.9　运行结果

6. 问题拓展

本题还可以使用“递推法”来求解。

递推法是利用问题本身所具有的递推关系来求解。所谓递推关系指的是：当得到问题规模为n-1的解后，可以得出问题规模为n的解。因此，从规模为0或1的解可以依次递推出任意规模的解。

（1）算法设计

找到递推关系。

定义数组fish[6]来保存每个人分鱼前鱼的总条数，A、B、C、D、E分鱼前鱼的总条数分别存放在fish数组下标为1、2、3、4、5的元素中。

相邻两人看到的鱼的条数存在如下关系：

```
fish[1]=全部的鱼
fish[2]=(fish[1]-1) // 5 * 4
fish[3]=(fish[2]-1) // 5 * 4
fish[4]=(fish[3]-1) // 5 * 4
fish[5]=(fish[4]-1) // 5 * 4
```

据此，得出一般的表达式：

```
fish[n]=(fish[n-1]-1) // 5 * 4
```

则

```
fish[n-1]=fish[n]*5 // 4 + 1
```

下面使用穷举法，假设E分鱼前鱼的总数为6条、11条、16条……则对应E的每次取值都可以将其他4个人分鱼前鱼的总数递推出来。每个人分鱼前，鱼的总数%5都必须为1，且B、C、D、E分鱼前鱼的总数%4必须为0，即每次剩余的鱼必须能够均分成4份。

（2）完整的程序

根据上面的分析，完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 递归解决分鱼问题

if __name__ == "__main__":
    # fish[6]存放每个人分鱼前的总条数
    # A、B、C、D、E分鱼前鱼的总条数分别存放在fish数组下标为1、2、3、4、5的元素中
    fish = [0]*6
    fish[5] = 6
    while True:
        i = 4
        while i > 0:
            if fish[i+1] % 4 != 0:
                break
            fish[i] = fish[i+1] * 5 // 4 + 1                    # 递推关系
式
            if fish[i] % 5 != 1:
                break
            i -= 1
        if i == 0:
            break
        fish[5] += 5
    for i in range(1, 6):
        print("fish[%d] = %d " %(i, fish[i]))                   # 输出结果
```

（3）运行结果

在PyCharm下运行程序，结果如图9.10所示。由图9.10可知，5个人合伙捕到的鱼的总条数为3121条，A醒来后看到的鱼是3121条，B醒来后看到的鱼是2496条，C醒来后看到的鱼是1996条，D醒来后看到的鱼是1596条，E醒来后看到的鱼是1276条。

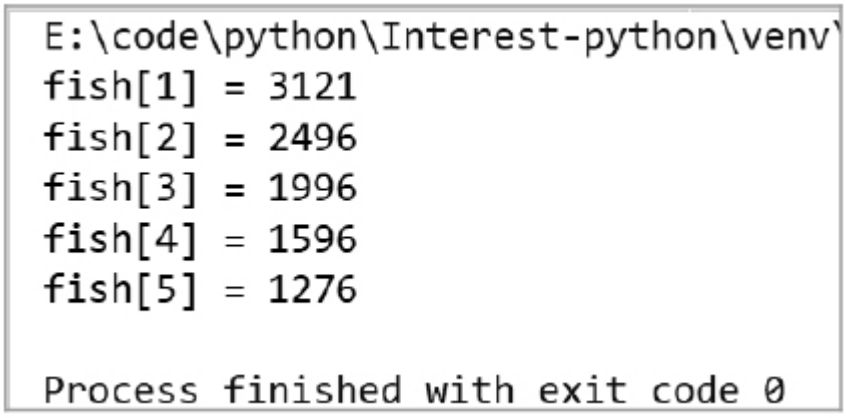


图9.10　运行结果

由于递归会引起一系列的函数调用，并且可能会产生一系列的重复计算，因此递归算法的执行效率相对较低。当某个递归算法能使用递推算法来实现时，出于对效率的考虑，通常按照递推算法来编写程序。

9.6　汉诺塔问题

1. 问题描述

汉诺塔问题是一个古典的数学问题，它只能用递归方法来解决。在古代有一个梵塔，塔内有A、B、C三个座。开始时A座上有64个盘子，盘子大小不同，但保证大的在下，小的在上。现在有一个和尚想将这64个盘子从A座移动到C座，但他每次只能移动一个盘子，且在移动过程中在3个座上都必须保持大盘在下小盘在上的状态。在移动过程中可以利用B座，要求编程将移动步骤打印出来。汉诺塔示意图如图9.11所示。

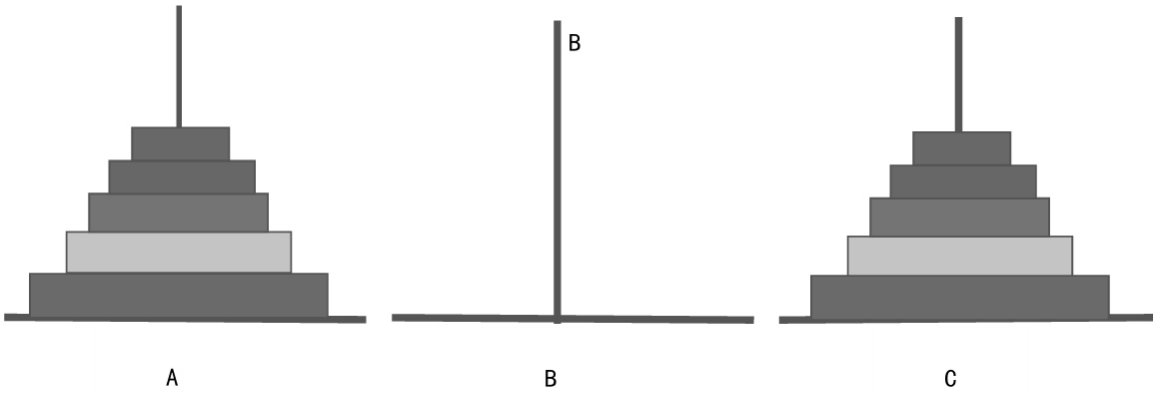


图9.11　汉诺塔示意图

2. 问题分析

汉诺塔问题是一个著名的问题，由于条件是每次只能移动一个盘，且不允许大盘放在小盘上面，因此64个盘子的移动次数是18 446 744 073 709 551 615这是一个天文数字，即使使用计算机也很难计算64层的汉诺塔问题。因此在这里仅给出问题的解决方法并解决盘子数较小时的汉诺塔问题。

首先考虑A座上最下面的盘子，如果能将它上面的63个盘子从A座移动到C座，则任务完成。具体步骤如下：

1）将A座最上面的63个盘子移动到B座上。

2）将A座上剩下的一个盘子移动到C座上。

3）将B座上的63个盘子移动到C座上。

如果能完成上述3步，则任务完成。这种思考方法就是递归的思考方法。但实际上问题并没有解决，在步骤1中如何将A座最上面的63个盘子移动到B座上呢？

为了解决将将A座最上面的63个盘子移动到B座上的问题，还需要做如下工作：

1）将A座上面的62个盘子移动到C座上。

2）将A座上剩下的一个盘子移动到B座上。

3）将C座上的62个盘子移动到B座上。

将这个过程进行下去，即不断地递归，继续完成移动62个盘子、61个盘子……的工作，直到最后达到仅有一个盘子的情形，则将一个盘子从一个座移动到另一个座，问题也就全部得到了解决，所有的步骤都是可执行的。

要说明的是，只有移动一个盘子的任务完成后，移动两个盘子的任务才能完成，以此类推，只有移动63个盘子的任务完成后，移动64个盘子的任务才能完成，由此可知该问题是非常典型的递归问题。

3．算法设计

该问题使用递归算法来解决。递归算法具有一些特征：为了求解规模为N的问题，应先设法将该问题分解成一些规模较小的问题，从这些较小问题的解可以方便地构造出大问题的解。同时，这些规模较小的问题也可以采用同样的方法分解成规模更小的问题，并能

从这些规模更小的问题的解中构造出规模较小问题的解。特别地，当N=1时，可直接获得问题的解。

现在给出解决问题的方法。

先定义递归函数hanio(N,A,B,C)，该方法表示将N个盘子从A座借助B座移动到C座，盘子的初始个数为N。下面是解题步骤：

若A座上只有1个盘子，此时N=1，则可直接将盘子从A座移动到C座上，问题得到解决。

若A座上有1个以上的盘子，即N>1，此时需要再考虑3个步骤。

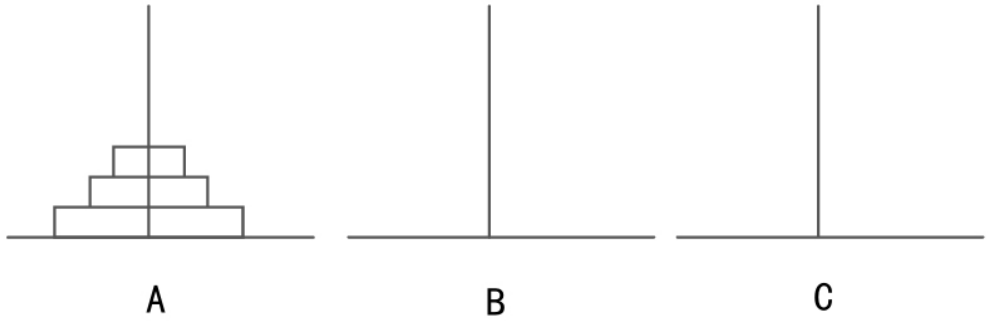
1）先将N-1个盘子从A座借助C座移动到B座上。显然，这N-1个盘子不能作为一个整体移动，而是要按照要求来移动。此时，可递归调用函数hanio(N-1,A,C,B)。需要注意的是，这里借助C座将N-1个盘子从A座移动到B座，A是源，B是目标。

2）将A座上剩下的第N个盘子移动到C座上。

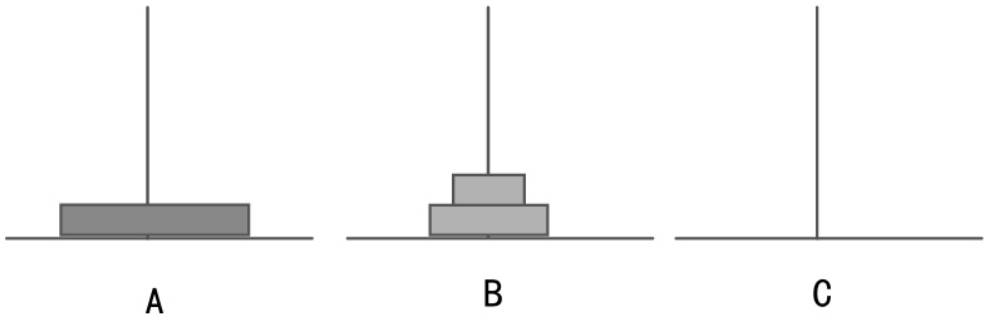
3）将B座上的N-1个盘子借助A座移动到C座上。此时，递归调用函数hanio(N-1,B,A,C)。需要注意的是，这里借助于A座将N-1个盘子从B座移动到C座，B是源，C是目标。

完成了这3步，就可以实现预期的效果，在C座上按照正确的次序叠放好所有的盘子。

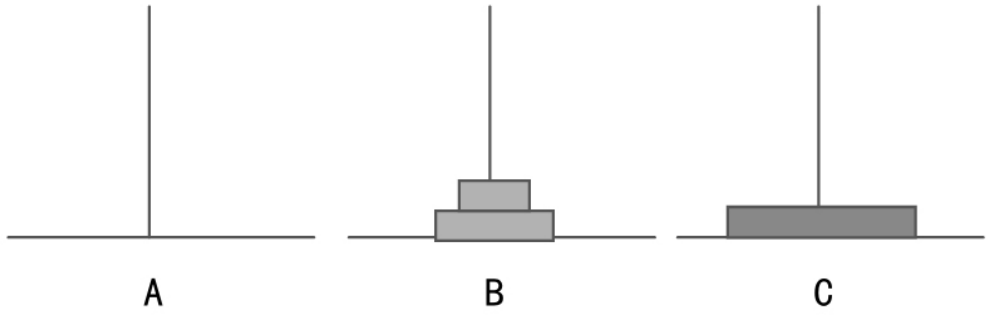
下面以移动3个盘子为例，给出使用递归函数移动3个盘子的示意图，如图9.12所示。



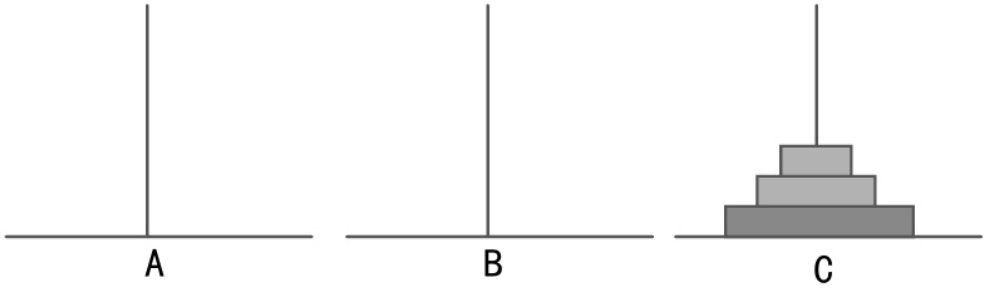
a）未移动前



b）步骤 1



c）步骤 2



d）步骤 3

图9.12　移动3个盘子的示意图

4. 确定程序框架

根据前面的分析，编写递归函数hanio(N,A,B,C)，代码如下：

```
def hanoi(N, A, B, C):
    if N == 1:                                  # 将A座上剩下的第N个盘子移动到C座上
        print("move dish %d from %c to %c " %(N, A, C))      # 打印移动步骤
    else:
        hanoi(N-1, A, C, B)          # 借助C座将N-1个盘子从A座移动到B座
        print("move dish %d from %c to %c " %(N, A, C))      # 打印移动步骤
        hanoi(N-1, B, A, C)
```

完成了递归函数后，只需要在main()函数中调用它，把需要移动的盘子个数传递给它即可。

5. 完整的程序

根据上面的分析，编写完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 汉诺塔问题

def hanoi(N, A, B, C):
    if N == 1:                                  # 将A座上剩下的第N个盘子移动到C座上
        print("move dish %d from %c to %c " %(N, A, C))      # 打印移动步骤
    else:
        hanoi(N-1, A, C, B)          # 借助C座将N-1个盘子从A座移动到B座
        print("move dish %d from %c to %c " %(N, A, C))      # 打印移动步骤
        hanoi(N-1, B, A, C)


if __name__ == "__main__":
    #输入要移动的盘子个数
    n = int(input("Please input the number of dishes: "))
    print("The steps to move %2d dishes are: " %n);
    hanoi(n, 'A', 'B', 'C');
# 调用递归函数
```

6. 运行结果

在PyCharm下运行程序，输入要移动的盘子数为3，则显示结果如图9.13所示。输入要移动的盘子数为4时，则显示结果如图9.14所示。根据显示结果可知，在汉诺塔中按规则移动3个盘子需要7步，

移动4个盘子则需要15步。类似地，读者可以运行程序得出移动更多盘子所需要的步数。

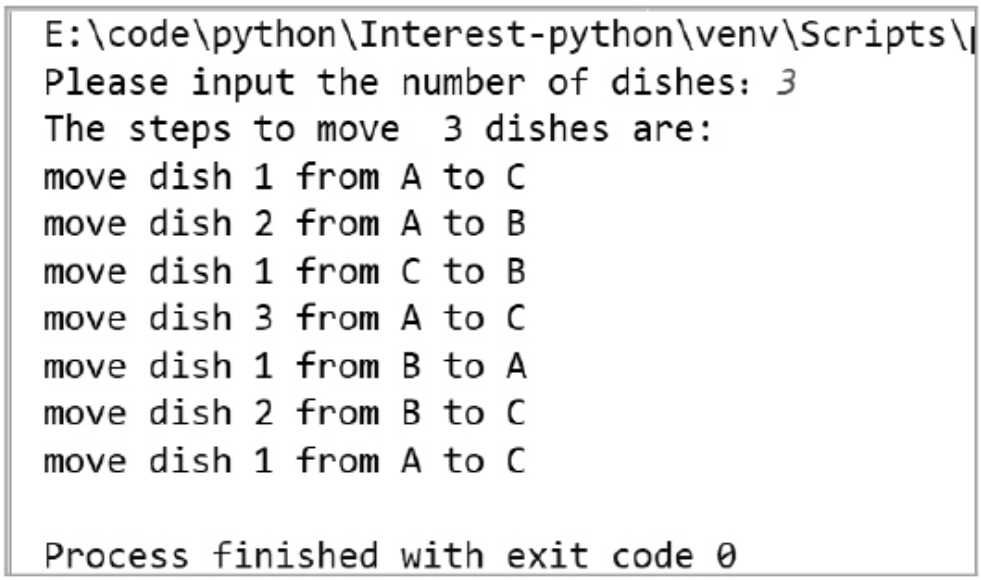


图9.13　移动3个盘子的运行结果

```
E:\code\python\Interest-python\venv\Scripts\
Please input the number of dishes: 4
The steps to move  4 dishes are:
move dish 1 from A to B
move dish 2 from A to C
move dish 1 from B to C
move dish 3 from A to B
move dish 1 from C to A
move dish 2 from C to B
move dish 1 from A to B
move dish 4 from A to C
move dish 1 from B to C
move dish 2 from B to A
move dish 1 from C to A
move dish 3 from B to C
move dish 1 from A to B
move dish 2 from A to C
move dish 1 from B to C

Process finished with exit code 0
```

图9.14　移动4个盘子的运行结果

9.7　逆序输出数字

1．问题描述

编程实现将输入的整数逆序输出。

2．问题分析

前面我们已经接触过很多的递归问题了，这些递归问题可以简单地分成两类：一类可以归结为数值问题，还有一类为非数值问题。

数值问题的递归是指可以表达为数学公式的问题，如9.1节的猴子吃桃问题，9.4节的求年龄问题。

非数值问题的递归是指问题本身难以用数学公式来表达的问题，如9.3节的汉诺塔问题。

对于数值问题，由于本身可以表达为数学公式，所以可以从数学公式入手来推导出问题的递归定义，然后确定问题的边界条件，从而最终确定递归函数和递归结束条件。

对于非数值问题，由于其本身不能用数学公式来表达，因此其求解的一般方法是要自行设计一种算法，进而找到解决问题的一系列操作步骤。如果能够找到解决问题的一系列递归操作步骤，则非数值问题也可以使用递归方法来求解。

本节我们要讨论的逆序输出数字就是一个数值问题的递归。问题本身并不复杂，但通过该问题我们可以对这一类递归问题的编程方法进行总结和比较，以便于读者在遇到相似的问题时能参照解决。

3．算法设计

该问题要求任意输入一个整数，实现它的逆序输出。因此，首先要判断输入的整数是正整数还是负整数。无论正负，程序都应该能处理。如果是负整数，则在逆序输出前应先打印出负号。

接着解决整数的逆序问题。

假设输入的整数保存在变量num中，我们以输入4位的正整数为例，其他位数的整数可类似地分析。假设输入的4位正整数为abcd。我们可以这样思考：

1）如果三位数abc已经实现了逆序，则只需打印d与abc逆序后的三位数即可。

2）如果两位数ab已经实现了逆序，则只需打印c与ab逆序后的两位数即可。

3）如果一位数a已经实现了逆序，则只需打印b与a逆序后的一位数即可。显然，a逆序后还是它本身，因此此步可直接打印结果：ba。

4）接着再由第3步向前递推，则可以实现逆序输出abcd这个4位整数。

由刚才的分析过程可知，显然这是一个递归问题，将大问题逐渐细化，递归的出口就是对一位数进行逆序操作的结果仍是它本身。

4. 确定程序框架

根据算法分析过程，我们可以写出逆序的递归函数reverse()，代码如下：

```
# 逆序函数
def reverse(n):
    if n != 0:
        print("%d" % (n % 10), end="")       # 输出正整数n当前的最高位
        reverse(n // 10)                              # 递归调用
```

上面的代码中加粗的部分表示去掉当前正整数n中的最高位，将剩下的正整数作为递归函数的实际参数。

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc:  逆序输出数字

# 逆序函数
def reverse(n):
    if n != 0:
        print("%d" % (n % 10), end="")                  # 输出正整数n当前的最高位
        reverse(n // 10)                                        # 递归调用

if __name__ == "__main__":
    num = int(input("请输入一个整数："))
    # 如果num小于0，就先把num转化字符串，截取第一位 -号，然后将数字逆序，再拼接上
      符号输出
    if num < 0:
        str_num = str(num)
        num = str_num[1:]                                   # 剪切掉符号位
        print("-", end="")
        reverse(int(num))
    else:
        reverse(num)
```

6. 运行结果

在PyCharm下运行程序，结果如图9.15所示。由图9.15可见，程序可以正确地将输入的整数逆序输出。

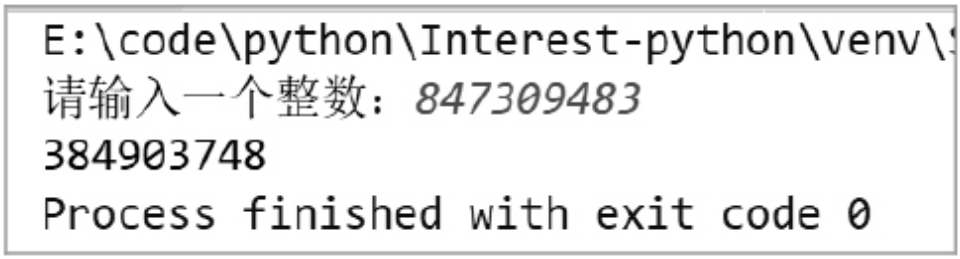


图9.15　运行结果

7. 问题拓展

（1）逆序输出字符串

与逆序输出数字类似的问题还有逆序输出字符串。下面我们先直接给出使用递归来逆序输出字符串的完整代码，再进行分析。

逆序输出字符串的完整代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 逆序输出字符串

# 递归函数
def rvstr(s):
    if len(s) <= 1:
        return s
    return rvstr(s[1:]) + s[0]

if __name__ == "__main__":
    str = str(input("请输入一个字符串："))
    str_result = rvstr(str)
    print("%s 逆序后：  %s" % (str, str_result))
```

在上面的代码中，当输入时按Enter键，表示字符串输入结束，可以开始逆序打印。

在递归函数rvstr(s)中，我们用到了字符串的分片功能，通过递归调用字符串s的分片剪切实现字符串的逆序功能。

在PyCharm下运行程序，运行结果如图9.16所示。

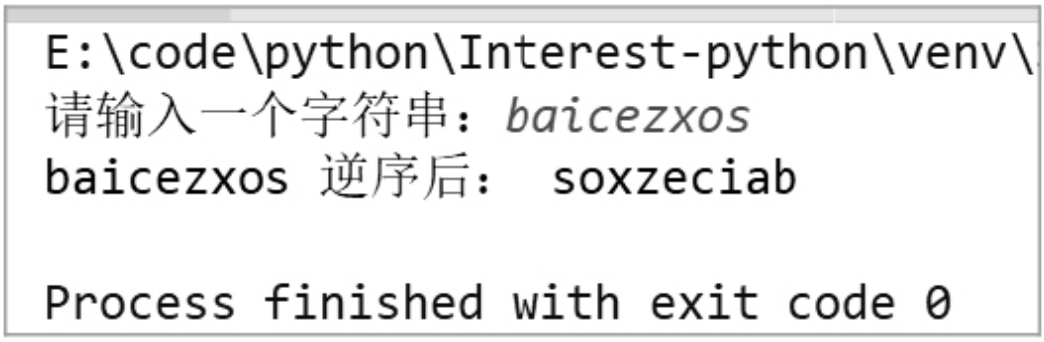


图9.16　运行结果

在Python中，一个字符串相当于一个不可改变序列的列表。一旦这个字符串被定义了，则该字符串中的每个字符都有了自己确定的位置，也有了自己的下标，不可改变。

在Python中，可以使用“[]”来访问字符串中指定位置上的字符，这种方式类似于C语言或Java语言中的数组。在Python语言的字符串中，字符的序号也是从0开始的，即str[0]表示字符串str中的第1个字符。

与C语言或Java语言不同的是，Python允许以负数表示字符的序号，负数表示从字符串的末尾处开始计算，字符串的最后一个字符序号为-1，而不是-0。

上面提到的字符序号也可以理解为是索引，索引用来对单个元素进行访问。在Python语言中，对一个序列一定范围内的元素进行访问的技术叫作分片技术，分片是通过冒号相隔的两个索引实现的。分片操作既支持正数索引，又支持负数索引，这对于访问一个序列的部分元素很方便。分片操作的实现需要提供序列的两个索引作为边界值，第一个索引的元素包含在分片内，第二个索引的元素不包含在分片内。

下面提供一个实例代码以帮助读者理解分片，代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 序列分片


if __name__ == "__main__":
    str = "abcdefghijklmnopqrstuvwxyz"      # 定义字符串

    print(str[4])     # 取字符串中序号为4的字符，也就是第5个字符串
    print(str[-3])            # 从字符串的尾部开始计算，取字符串的倒数第3个字符
    print(str[-0])            # -0即是0，也就是取字符串的第一个字符
    print(str[-1])            # 取字符串的倒数第一个字符，也就是最后一个字符

    #分片
    print(str[2:8])   # 取字符串中从第3个字符到第9个字符的内容，不包含第9个字符
    print(str[1:1])           # 不包含第2个字符，这里为空
    print(str[1:-1])          # 从第2个字符开始到最后一个字符，不包含最后一个字符
    print(str[0:-1])          # 从第1个字符到倒数第1个字符，不包含最后一个字符
    print(str[-8:-2])         # 从倒数第9个到倒数第2个字符，不包含倒数第9个字符
    # 打印为空，在分片中最左边的索引比它右边的索引晚出现在序列中，结果就是一个空值
    print(str[-8:0])
    print(str[:10])   # 从第1个字符取到第11个字符，不包含第11个字符
    print(str[12:])   # 从第13个字符取到最后一个字符
    print(str[-9:])   # 从倒数第9个字符开始取
    print(str[:-4])   # 从第1个字符取到倒数第4个字符，不包含倒数第4个
    print(str[:])     #取出整个字符串
```

在PyCharm下运行程序，结果如图9.17所示。

```
E:\code\python\Interest-python\venv\Scripts
e
x
a
z
cdefgh

bcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxy
stuvwx

abcdefghij
mnopqrstuvwxyz
rstuvwxyz
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvwxyz

Process finished with exit code 0
```

图9.17　运行结果

（2）其他数值问题的递归

其他一些数值问题还有求n！、求斐波拉切数列、求最大公约数等问题。下面我们再介绍使用递归求最大公约数的方法。

用辗转相除法求出两个整数m与n的最大公约数。

本题要求我们使用辗转相除法（也称欧几里德算法或欧式算法）来求解m和n的最大公约数。辗转相除法的公式如下：

$$\gcd(m,n)\begin{cases} n & m\ \%\ n=0 \\ \gcd(n,\ m \bmod n) & m\ \%\ n\neq 0 \end{cases}$$

由上述公式可知，求m与n的最大公约数问题可以等价于求n与(m%n)的最大公约数问题。因此，可以把n当作新的m，把(m%n)当作新的n，则问题再次转换为求新的m与新的n的最大公约数。而求新的m与新的n的最大公约数又等价于求新的n与(m%n)的最大公约数，如此继续下去，直到新的n为0，则所求的最大公约数就是当前的m值，这就是利用辗转相除法求m与n的最大公约数的过程。

例如：求gcd(70,30)。

1）当前m=70，n=30，m%n=10。

2）转换成求gcd(30,10)，此时m=30，n=10，由于m%n=0，故结束，gcd(70,30)=n=10。

根据上面的描述，列出递归算法的步骤：

1）求r=m%n。

2）判断r是否为0。若r=0，则n为所求的最大公约数，输出n。

3）若r!=0，则令m=n，n=r。

4）转到步骤1。

根据上述分析，编写递归函数gcd()如下：

```
# 递归函数 求最大公约数
def gcd(m , n):
    if n == 0:
        g = m
    else:
        g = gcd(n, m % n)                                    # 递归调用
    return g
```

还可以在递归函数中加入对特殊情况的处理，如m、n为负数的情况。此时递归函数代码如下：

```
# 递归函数 求最大公约数
def gcd(m , n):
    if m < 0:
        m = -m
```

```
    if n < 0:
        n = -n
    if n == 0:
        g = m
    else:
        g = gcd(n, m % n)                                    # 递归调用
    return g
```

上面函数中加粗的代码表示如果m、n为负数，则先将其转化为正整数，再进行下面的操作。

还可以使用条件表达式来简化递归函数的书写，具体代码如下：

```
# 递归函数 求最大公约数
def gcd(m , n):
    if m < 0:
        m = -m
    if n < 0:
        n = -n

    return m if n==0 else gcd(n, m%n)                # 递归调用
```

函数中加粗的代码使用了条件表达式，它的执行效果与使用if...else...结构的效果相同。

完整的程序代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 递归求最大公约数

# 递归函数 求最大公约数
def gcd(m , n):
    if n == 0:
        g = m
    else:
        g = gcd(n, m % n)                                    # 递归调用
    return g

if __name__ == "__main__":
    print("请输入两个正整数：")
    m = int(input("m = "))
    n = int(input("n = "))
    g = gcd(m, n)            # 调用递归函数
    print("%d和%d的最大公约数是：%d" %(m, n,
g));
```

在PyCharm下运行程序，结果如图9.18所示。

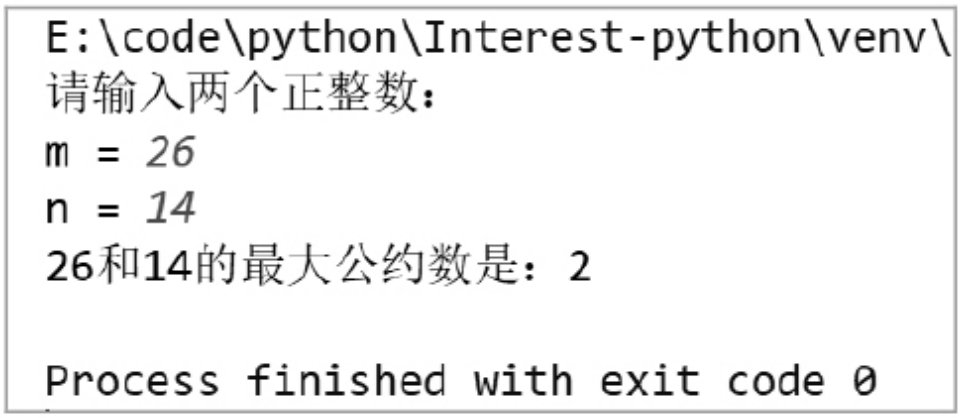


图9.18　求最大公约数的运行结果

第10章　定理与猜想

本章选择了几个数学中的定理及猜想进行验证，包括四方定理、角谷定理及一些还未获得完全证明的问题，如回文数的形成方法。

本章将使用Python语言验证定理和猜想的各种方法展现给了读者。通过对本章的学习，可以拓宽读者的思路，增强读者利用Python语言解决相关问题的能力。

10.1　尼科彻斯定理

1．问题描述

尼科彻斯定理可以叙述为“任何一个整数的立方都可以表示成一串连续的奇数的和”。

2．问题分析

根据尼科彻斯定理的叙述举例如下：

3^3 =7+9+11=27

4^3 =13+15+17+19=64

5^3 =21+23+25+27+29=125

由于“任何一个整数的立方都可以表示成一串连续的奇数的和”，因此我们可以这样考虑该问题：先计算任意输入数n的立方，然后从奇数1开始进行累加，每次加2以保证下一个数也是奇数，如果累加和超过n的立方，则再进行下一次尝试，即从3开始进行累加，如此进行下去，直到找到一串连续的奇数，它们的和等于n的立方。

3．算法设计

根据问题分析，该问题可使用循环结构来实现。

首先定义变量n，用来保存输入的某个整数，并计算出n的立方，用变量cube来表示。接着使用双重循环来查找这串连续的奇数。

在双重循环中，定义外层循环变量为i，它控制尝试的次数；定义内层循环变量为j，它控制找到的这串奇数的长度。

4. 确定程序框架

（1）外层循环框架

i从1开始取值，即第一次从1开始试探，是否有cube=1+3+…这样的奇数序列存在，如果不存在，则令i增加2，再次试探是否有cube=3+5+…这样的奇数序列存在，如果不存在，则i再次增加2，试探是否有cube=5+7+…这样的奇数序列存在，如此循环下去，直到找到这样的一个序列，它们的和等于cube。

i的取值不会超过cube。需要注意的是，这样的奇数序列并不唯一。

根据上面的叙述，外层循环框架如下：

```
# cube存放n的立方，也可以写成cube = n ** 3
cube = n * n * n
# 外层循环通过累加和来查找奇数序列
i = 1
while i < cube:
    # 试探是否存在一个奇数序列，它们的和等于
      cube
    ...
    I += 2
```

（2）内层循环框架

内层循环的主要作用是找到和为cube的奇数序列。j的取值从i开始，并且不会超过cube，每次循环j的值都增加2。在内层循环的循环体中测试是否存在j+(j+2)+…=cube，如果存在，则找到了这串奇数序列，否则将j值增加2，再次测试是否存在j+(j+2)+…=cube，这样循环下去，直到找到一个奇数序列其和等于cube为止。

根据上面的叙述，内层循环框架如下：

```
# 内层循环通过累加和来查找奇数序列
j = i
while j < cube:
    sum += j
    # 找到了奇数序列
    if sum == cube:
        print("%d = %d + %d + ... + %d" %(cube, i, i+2, j))
    # 没找到，退出内层循环，返回外层for循环
    if sum > cube:
```

```
        # 将sum重置为0，以便开始下次试探
        sum = 0
        break
    j += 2
```

程序流程图如图10.1所示。

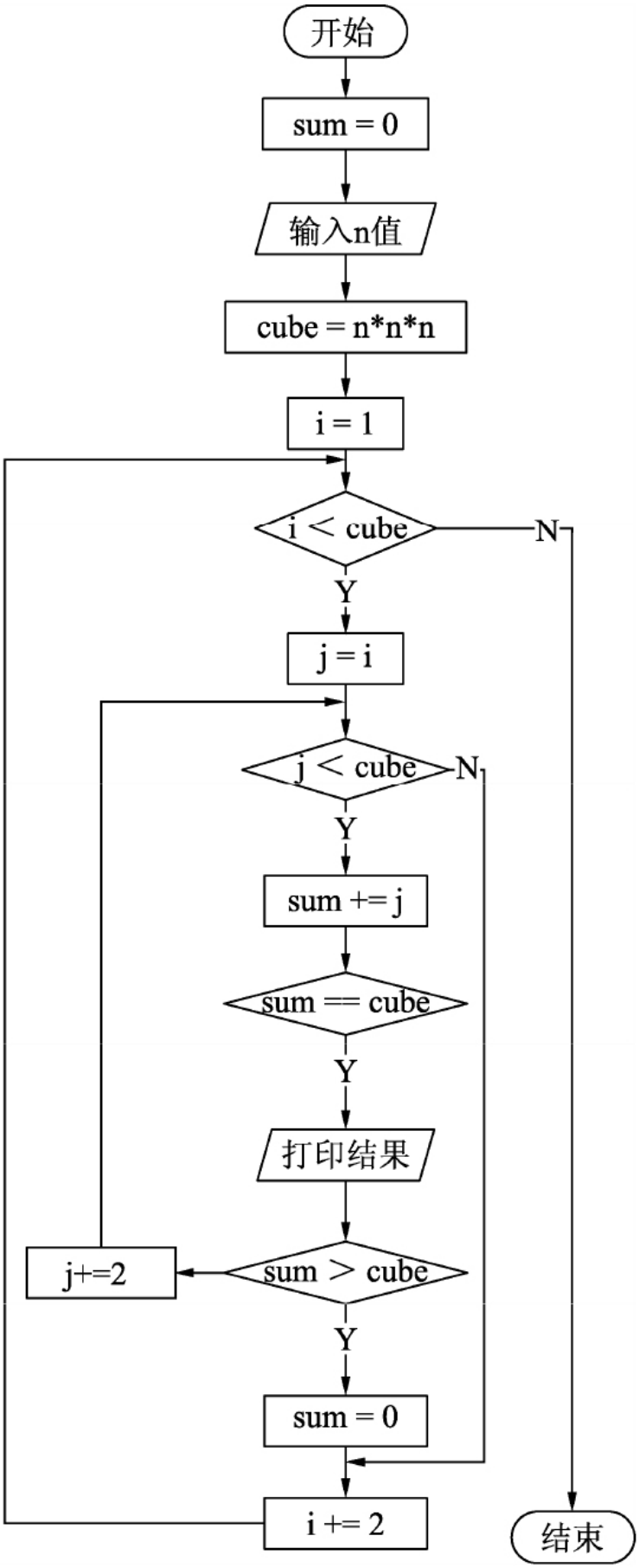

开始
sum = 0
输入n值
cube = n*n*n
i = 1
i < cube
N
Y
j = i
j < cube
N
Y
sum += j
sum == cube
Y
打印结果
j+=2
sum > cube
Y
sum = 0
i += 2
结束

图10.1　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 尼科彻斯定理
# 任何一个整数的立方都可以表示成一串连续的奇数的和

if __name__ == "__main__":
    sum = 0                                                    # sum变量存放奇数的累
加和，初值为0
    n = int(input("请输入大于1的n值："))
    if n <= 1:
        print("输入的n值有误")
        exit()
    cube = n * n * n                                # cube存放n的立方，也可以写成cube = n
** 3
    # 外层循环通过累加和来查找奇数序列
    i = 1
    while i < cube:
        # 内层循环通过累加和来查找奇数序列
        j = i
        while j < cube:
            sum += j
            # 找到了奇数序列
            if sum == cube:
                print("%d = %d + %d + ... + %d" %(cube, i, i+2, j))
            # 没找到，退出内层循环，返回外层for循环
            if sum > cube:
                sum = 0                                 # 将sum重置为0，以便开始下次试探
                break
            j += 2
        i += 2
```

6．运行结果

在PyCharm下运行程序，结果如图10.2所示。由图10.2可知，找到的连续的奇数序列并不是唯一的，可能会存在多组解，但只要存在一组解，就可以证明尼科彻斯定理是正确的。

```
E:\code\python\Interest-python\venv
请输入大于1的n值：5
125 = 21 + 23 + ... + 29

Process finished with exit code 0
```

a）运行结果 1

```
E:\code\python\Interest-python\venv\
请输入大于1的n值：14
2744 = 71 + 73 + ... + 125
2744 = 183 + 185 + ... + 209
2744 = 683 + 685 + ... + 689
2744 = 1371 + 1373 + ... + 1373

Process finished with exit code 0
```

b）运行结果 2

图10.2　运行结果

7．问题拓展

本题还可以采用另一种方法求解。

首先证明尼科彻斯定理是成立的。

对于任意一个正整数n，不论n是奇数还是偶数，(n×n-n+1)必然为奇数。

构造一个等差数列，该数列的首项a1=(n×n-n+1)，公差d为2，该数列为奇数数列，则其前n项和S_n 为：

$S_n = na_1 + n(n-1)d/2$

$=n\times(n\times n-n+1)+n(n-1)\times 2/2$

$=n\times n\times n-n\times n+n+n\times n-n$

$=n\times n\times n$

由此便可证明，任意一个正整数n的立方都可以表示成一串连续的奇数的和。

通过上述推导过程可知，奇数数列的首项为(n×n-n+1)，长度为n。按照定理的证明可直接编程实现。

完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc:  尼科彻斯定理
# 任何一个整数的立方都可以表示成一串连续的奇数的和

if __name__ == "__main__":
    n = int(input("请输入大于1的n值："))
    if n <= 1:
        print("输入的n值有误")
        exit()
    cube = n ** 3                                                    # cube存放n
的立方
    print("%d * %d * %d = %d = " %(n, n, n, cube), end="")
```

```
    s = 0                                                    # s表示数列
的前n项和
    i = 0                                                    # i表示数列
的第i项
    while i < n:                                             # 输出数列，首项
为n*n-n+1，公差为2
        s += n * n - n + 1 + i * 2          # 求数列的前n项和
        y = "+ %d" if i else "%d"
        print(y %(n * n - n + 1 + i * 2), end=" ")  # 打印结果
        i += 1
    if s == cube:
        print(" 定理成立")
    else:
        print(" 定理不成立")
```

在PyCharm下运行程序，结果如图10.3所示。由此可以证明尼科彻斯定理是正确的。

```
E:\code\python\Interest-python\venv\Scripts\python.exe
请输入大于1的n值：5
5 * 5 * 5 = 125 = 21 + 23 + 25 + 27 + 29  定理成立

Process finished with exit code 0
```

a）运行结果 1

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/code/python/Interest-python/chapter10/10.4.1.py
请输入大于1的n值：14
14 * 14 * 14 = 2744 = 183 + 185 + 187 + 189 + 191 + 193 + 195 + 197 + 199 + 201 + 203 + 205 + 207 + 209  定理成立

Process finished with exit code 0
```

b）运行结果 2

图10.3　运行结果

10.2　奇数平方的有趣性质

1．问题描述

任意奇数的平方有这样的有趣性质：任意奇数的平方与1的差是8的倍数。要求编程验证奇数的这个性质。

2．问题分析

本题是一个数学定理，我们先来验证该定理的成立。

设任意奇数可以表示为2n+1，则将其平方后再减1为：

$(2n+1)^2 -1$

$=(2n+1+1)(2n+1-1)$

$=4n(n+1)$

• 当n为奇数时，n+1为偶数，4n(n+1)=8*n*[(n+1)/2]，显然为8的倍数。

• 当n为偶数时，n+1为奇数，4n(n+1)=8*n/2*(n+1)，显然为8的倍数。

因此，对于任意的奇数来说，其平方后再减1所得差值一定是8的倍数。

3．完整的程序

由问题分析可知，该定理成立。题目要求我们编程来验证该定理，由于不可能穷举出所有的奇数，因此我们用有限范围内的奇数来验证该定理的成立。

在程序中验证1000～2000范围内的所有奇数都具有该性质。该程序比较简单，只用一个循环结构就可以完成验证，下面直接给出完整的程序。

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 奇数平方的有趣性质
# 任意奇数的平方有这样的有趣性质：任意奇数的平方与1的差是8的倍数

if __name__ == "__main__":
    # 对1000～2000范围内的所有奇数进行验证
    n = 1001
    while n <= 1999:
        # 输出当前的奇数
        print("%d: " %(n), end=" ")
        print("(%d*%d-1)/8" %(n, n), end=" ")
        # 输出(n*n-1)/8的值
        print("= %d" %((n*n-1)//8), end=" ")
        # 输出（n*n-1）/8的余数
        print("+ %d\n" %((n*n-1)%8), end=" ")
        n += 2
```

程序中加粗的代码(n*n-1)%8表示用当前奇数n的平方减1后的值再对8取余，如果这个值为0，则表示n*n-1能被8整除，这样就验证了该定理成立。

4. 运行结果

在PyCharm下运行程序，实现对1000～2000范围内所有奇数的验证，运行结果如图10.4所示。由于运行结果过长，图10.4中只给出了1953～1999中所有奇数的验证过程。

```
1953:  (1953*1953-1)/8 = 476776 + 0
1955:  (1955*1955-1)/8 = 477753 + 0
1957:  (1957*1957-1)/8 = 478731 + 0
1959:  (1959*1959-1)/8 = 479710 + 0
1961:  (1961*1961-1)/8 = 480690 + 0
1963:  (1963*1963-1)/8 = 481671 + 0
1965:  (1965*1965-1)/8 = 482653 + 0
1967:  (1967*1967-1)/8 = 483636 + 0
1969:  (1969*1969-1)/8 = 484620 + 0
1971:  (1971*1971-1)/8 = 485605 + 0
1973:  (1973*1973-1)/8 = 486591 + 0
1975:  (1975*1975-1)/8 = 487578 + 0
1977:  (1977*1977-1)/8 = 488566 + 0
1979:  (1979*1979-1)/8 = 489555 + 0
1981:  (1981*1981-1)/8 = 490545 + 0
1983:  (1983*1983-1)/8 = 491536 + 0
1985:  (1985*1985-1)/8 = 492528 + 0
1987:  (1987*1987-1)/8 = 493521 + 0
1989:  (1989*1989-1)/8 = 494515 + 0
1991:  (1991*1991-1)/8 = 495510 + 0
1993:  (1993*1993-1)/8 = 496506 + 0
1995:  (1995*1995-1)/8 = 497503 + 0
1997:  (1997*1997-1)/8 = 498501 + 0
1999:  (1999*1999-1)/8 = 499500 + 0

Process finished with exit code 0
```

图10.4　运行结果

10.3　回文数的形成

1．问题描述

任取一个十进制正整数，将其倒过来后与原来的正整数相加，会得到一个新的正整数，重复以上步骤，则最终可得到一个回文数。请编程进行验证。

2．问题分析

回文数是指这个数无论从左向右读还是从右向左读都是一样的，如121、11等。

回文数的这一形成规则目前还未得到数学上的验证，还属于一个猜想。有些回文数的形成要经过上百个步骤，因此此处仅做编程验证，并打印形成过程。

如输入正整数78，则按照问题描述中回文数的形成规则，会有如下形成过程：

78+87＝165

165+561＝726

726+627＝1353

1353+3531＝4884

经过了4步，整数78形成了回文数4884。

3．算法设计

根据问题分析可知，该算法应该包括如下几个功能：

1）求正整数的反序数。

2）判断一个正整数是否为回文数。

3）形成回文数。

上面的这几个功能，我们可以分别通过不同的函数来实现。

4. 确定程序框架

（1）求输入正整数的反序数

定义reverse()函数，用于求输入正整数的反序数，代码如下：

```
# 求反序数
def reverse(a):
    t = 0
    while a > 0:
        t = t * 10 + a % 10 # t中存放的是a的反序数
        a //= 10
    return t
```

（2）判断给定的正整数是否为回文数

定义palindrome()函数，用于判断某个正整数是否为回文数，代码如下：

```
# 判断是否为回文数
def palindrome(s):
    if (reverse(s) == s):    # 调用reverse()函数判断变量s是否与其反序数相等
        return 1                        # s是回文数则返回1
    else:
        return 0                        # s不是回文数返回0
```

上面的代码中调用了reverse()函数判断变量s是否与其反序数相等，若相等，则s为回文数，palindrome()函数返回1；否则s不是回文数，palindrome()函数返回0。

（3）形成回文数

在主函数中根据形成回文数的规则来生成回文数，代码如下：

```
m = reverse(n)
while (palindrome(m + n) == 0):                    # 判断当前的正整数n是否为回文数
    count += 1
```

```
    print("[%d]:%d+%d=%d " % (count, n, m, m + n))          # 打印操作步骤
    n += m
# n加上其反序数
    m = reverse(n)
count += 1
print("[%d]:%d+%d=%d\n" % (count, n, m, m + n))
```

上面的代码中加粗的部分用来判断当前的正整数n是否为回文数。接着在while循环中再次判断n是否为回文数，如果不是则再次执行生成回文数的操作步骤，直至n为回文数为止，此时退出while循环。

在while循环中，每次n与其反序数m相加时都将其打印出来，这样最后可看到回文数生成的完整过程。

程序流程图如图10.5所示。

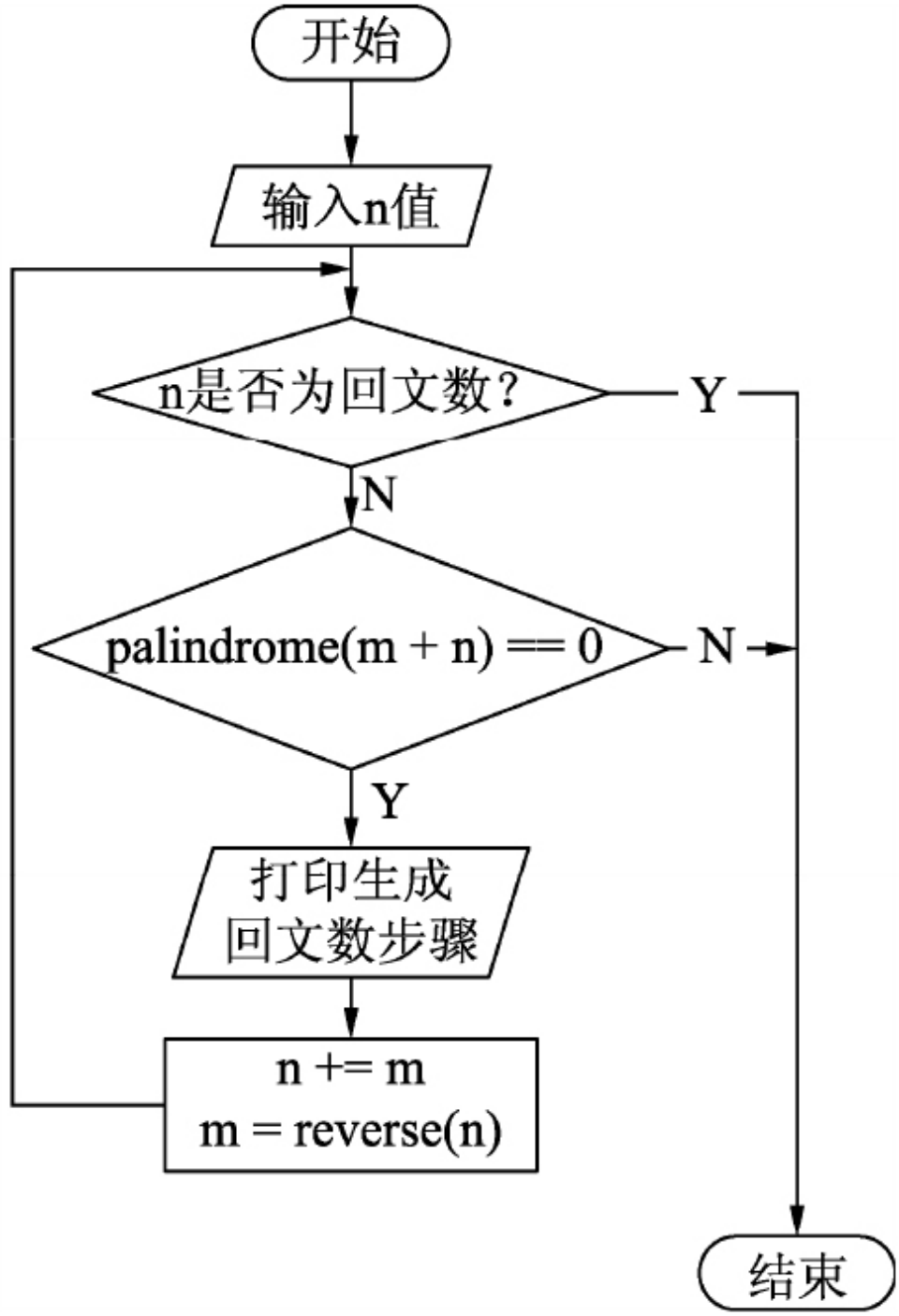


图10.5　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
# @author : liuhefei
# @desc: 回文数的形成
# 任取一个十进制正整数，将其倒过来后与原来的正整数相加，会得到一个新的正整数。重复以上
  步骤，则最终可得到一个回文数

# 求反序数
def reverse(a):
    t = 0
    while a > 0:
        t = t * 10 + a % 10 # t中存放的是a的反序数
        a //= 10
    return t

# 判断是否为回文数
def palindrome(s):
    if (reverse(s) == s):    # 调用reverse()函数判断变量s是否与其反序数相等
        return 1                    # s是回文数则返回1
    else:
        return 0                    # s不是回文数则返回0


if __name__ == '__main__':
    # 与C语言不同的是，Python支持大数计算，无限位数
    # 对于小整数，它会使用机器自身的整数计算功能去快速计算，当超出机器自身所能支持的范
      围时，自动转换为大数计算
    n = int(input("请输入一个正整数："))
    print("回文数的产生过程如下：")
    count = 0
    m = reverse(n)
    while (palindrome(m + n) == 0):                    # 判断当前的整数n是否为回文数
        count += 1
        print("[%d]:%d+%d=%d " % (count, n, m, m + n))       # 打印操作步骤
        n += m                                                                    # n
加上其反序数
        m = reverse(n)
    count += 1
    print("[%d]:%d+%d=%d\n" % (count, n, m, m + n))
```

---

6. 运行结果

在PyChram下运行程序，结果如图10.6所示。

注意到在图10.6a中，我们随便输入了一个4位正整数1993，对该整数共进行了8次生成回文数的操作步骤，最后得到的结果为2 322 232，显然它是一个回文数，因此验证了该回文数操作规则。

类似地，图10.6b中输入了4位正整数1996，经过5步操作，最后也获得了回文数。读者可以输入其他4位正整数来对本题中给出的回文数的操作规则进行验证。

```
E:\code\python\Interest-python\venv\Scripts\
请输入一个正整数：1993
回文数的产生过程如下：
[1]:1993+3991=5984
[2]:5984+4895=10879
[3]:10879+97801=108680
[4]:108680+86801=195481
[5]:195481+184591=380072
[6]:380072+270083=650155
[7]:650155+551056=1201211
[8]:1201211+1121021=2322232


Process finished with exit code 0
```

a）运行结果 1

```
E:\code\python\Interest-python\venv\Scripts\
请输入一个正整数：1996
回文数的产生过程如下：
[1]:1996+6991=8987
[2]:8987+7898=16885
[3]:16885+58861=75746
[4]:75746+64757=140503
[5]:140503+305041=445544


Process finished with exit code 0
```

b）运行结果 2

图10.6　运行结果

10.4　四方定理

1．问题描述

四方定理是数论中的重要定理，它可以叙述为“所有的自然数至多用4个数的平方和就可以表示出来”。例如：

25=1*1+2*2+2*2+4*4

99=1*1+1*1+4*4+9*9

要求编写程序来验证四方定理。

2．问题分析

问题描述中说“所有的自然数至多用4个数的平方和就可以表示出来”，既然是至多用4个数的平方和，那么三个、两个和一个数的平方和的情况要不要考虑在内呢？而且有些数是不能分解为4个数的平方和的，比如17，只能分解成两个或三个数的平方和：

17=1*1+4*4　--　分解为两个数的平方和

17=2*2+2*2+3*3　--　分解为三个数的平方和

而16可以用一个数的平方来表示：

16=4*4

因此，同一个数可能会分解出多组解。而我们在编程实现的时候只要能找到一组解就验证了四方定理，因此，在程序中可以设定，一旦找到一组解就退出。

3．算法设计

要验证四方定理，可以使用穷举法。可以在程序开始时允许输入任何一个自然数，然后判断该自然数可用至多4个数的平方和表示出来。

判断的过程我们使用穷举法。设计一个四重循环，循环变量分别使用x1、x2、x3和x4表示，如果它们的平方和相加恰好等于输入的number变量值，则证明四方定理成立。

现在的问题是循环变量x1、x2、x3和x4的取值范围怎样确定。考虑最极端的情况，如果输入的number用一个数的平方就可以表示出来，例如前面提到的16，则找到的一组解是x1=4，x2=0，x3=0，x4=0。显然x1不应该大于math.sqrt(number)，则可将x1的变化范围再扩大一些，将其变化范围中的最大值设为number/2，而0≤x2≤x1，0≤x3≤x2，0≤x4≤x3，则形成如下循环结构：

```
# 穷举各种情况
for x1 in range(0, number//2):
    for x2 in range(0, x1+1):
        for x3 in range(0, x2+1):
            for x4 in range(0, x3+1):
               #判断是否满足定理要求
```

定理所要求的条件可用如下语句表示：

```
number= =x1*x1+x2*x2+x3*x3+x4*x4
```

4. 确定程序框架

程序的流程图如图10.7所示。

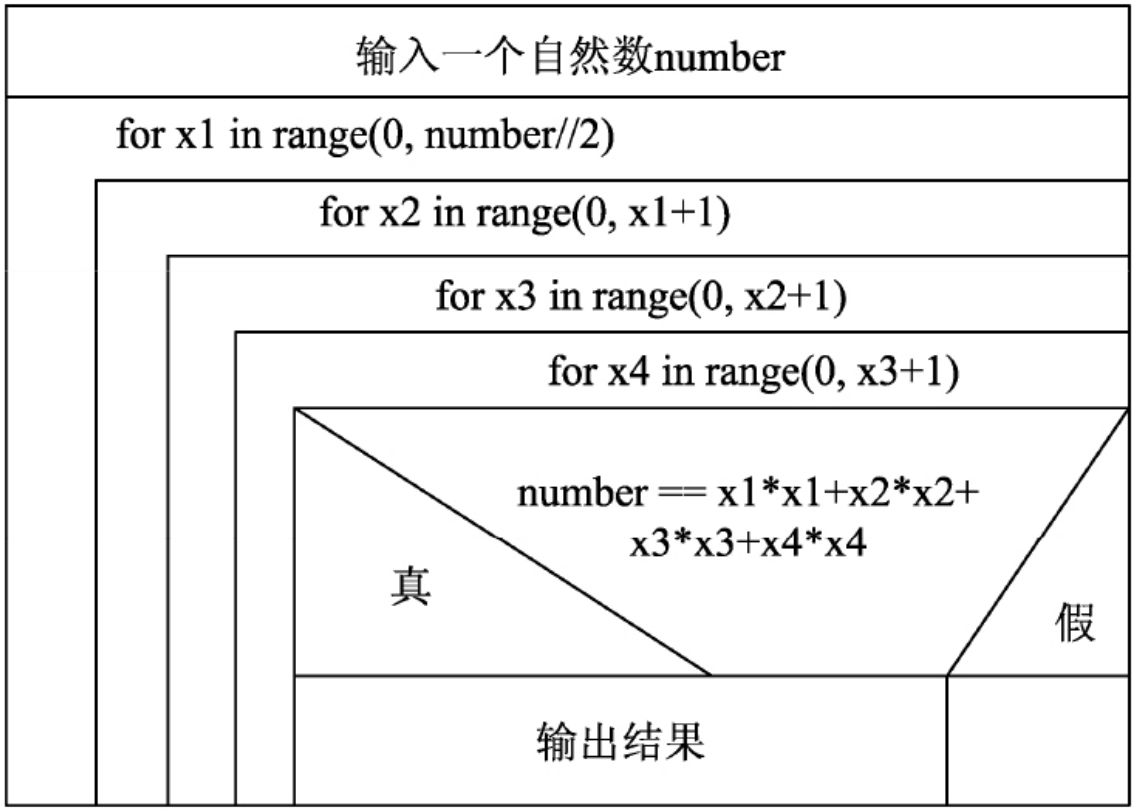


图10.7　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 四方定理
# 所有的自然数至多用4个数的平方和就可以表示出来

if __name__ == "__main__":
    number = int(input("请输入一个自然数："))
    # 穷举各种情况
    for x1 in range(0, number//2):
        for x2 in range(0, x1+1):
            for x3 in range(0, x2+1):
                for x4 in range(0, x3+1):
                    # 判断是否满足定理要求
                    if number == x1*x1 + x2*x2 + x3*x3 + x4*x4:
                        # 若满足定理要求，则输出其中一组解，并退出循环
                        print("%d=%d*%d+%d*%d+%d*%d+%d*%d" %(number, x1, x1,
x2, x2, x3, x3, x4, x4))
                        exit(0)                                    # 退出循环
```

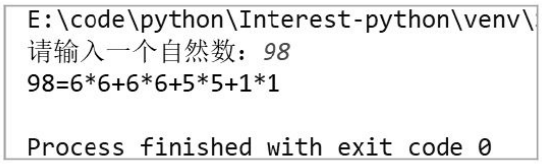
程序分析：

在上面的程序中，当找到一组满足条件的解之后就使用exit(0)函数退出了四重循环。

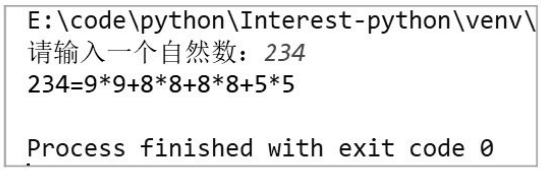
exit([arg])函数用来结束当前进程，在整个程序中只要调用了exit()函数，就会结束程序。[arg]是可选参数，默认为0，exit(0)表示程序正常退出。

6. 运行结果

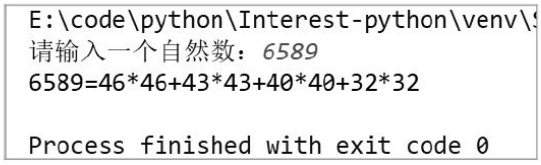
在PyCharm下运行程序，屏幕上提示“请输入一个自然数：”，我们随意输入3个自然数：98、234和6589，运行结果如图10.8所示。



a）运行结果 1



b）运行结果 2



c）运行结果 3

图10.8　运行结果

由运行结果可以看到，随意输入的3个数都被分解成了4个数的平方和。当然，除了屏幕上显示的这组解以外，还可能存在其他的分解方法，但无论如何，至多用4个数的平方和便可将一个数表示出来，这就通过程序验证了四方定理的正确性。

7. 问题拓展

在算法设计中，我们曾经分析过，循环变量x1的值实际上不会超过math.sqrt(number)。因此，我们可以修改循环结构，使x1的范围更精准，而x2、x3、x4的范围不变，完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 四方定理

import math

if __name__ == "__main__":
    number = int(input("请输入一个自然数："))
    n = int(math.sqrt(number))
    # 穷举各种情况
    for x1 in range(0, n):
        for x2 in range(0, x1+1):
            for x3 in range(0, x2+1):
                for x4 in range(0, x3+1):
                    # 判断是否满足定理要求
                    if number == x1*x1 + x2*x2 + x3*x3 + x4*x4:
                        # 若满足定理要求，则输出其中一组解，并退出循环
                        print("%d=%d*%d+%d*%d+%d*%d+%d*%d" %(number, x1, x1,
x2, x2, x3, x3, x4, x4))
                        exit(0)                         # 退出循环  要想输出多组解，可
以注释此行
```

10.5　角谷猜想

1．问题描述

角谷猜想在西方常被称为西拉古斯猜想，据说这个问题首先是在美国的西拉古斯大学被研究的，而在东方，这个问题则由将它带到日本的日本数学家角谷静夫的名字来命名，故被称为角谷猜想。

角谷猜想的内容是任给一个自然数，若为偶数则除以2，若为奇数则乘以3再加1，这样得到一个新的自然数之后再按照前面的法则继续演算，若干次以后得到的结果必然为1。在数学文献里，角谷猜想也常常被称为“3X+1问题”。请编程验证角谷猜想。

2．问题分析

先通过几个实例来理解角谷猜想的含义。

取自然数n=6，则根据角谷猜想，有：

6→3→10→5→16→8→4→2→1。

最终结果为1，则n=6时角谷猜想成立。

取自然数n=15，则根据角谷猜想，有：

15→46→23→70→35→106→53→160→80→40→20→10→5→16→8→4→2→1。

最终结果为1，则n=15时角谷猜想成立。

读者还可取其他的自然数按照上述规则来演算，结果都为1。

虽然角谷猜想还未获得一般的证明，但是已经有人拿各种各样的数字来进行试验，结果发现角谷猜想总是成立的，现在已经获得验证的最大数是1 099 511 627 776。

3. 算法分析

角谷猜想中已经明确地给出了处理过程，即对于给定的自然数n，有如下函数：

$$C(n)=\begin{cases} n/2 & n \text{ 是偶数} \\ 3n+1 & n \text{ 是奇数} \end{cases}$$

在问题分析中进行的变换，实际上是对函数C进行迭代。则问题可表述为：从任意一个自然数开始，经过对函数C有限次的迭代，能否最终得到1。

算法不需要特别的设计，根据函数C可直接进行角谷猜想的验证。

4. 确定程序框架

在main()函数中构建一个不定次数的while循环。定义变量n表示输入的自然数，在循环体中判断n的奇偶性并根据奇偶性的不同执行不同的操作，当n=1时，退出while循环。while循环结构如下：

```
while n != 1:   # 当n=1时终止循环
    # 若n为奇数，则乘以3加1
    if n % 2 == 1:
        n = n * 3 + 1
        count += 1
        # 输出执行步骤
        print("[%d]: %d * 3 + 1 = %d "
%(count, (n-1)//3, n))
    else:
        n //= 2      # 若n为偶数，则直接除以2
        count += 1
        print("[%d]: %d / 2 = %d " %(count,
2 * n, n))               # 输出执行步骤
```

程序的流程图如图10.9所示。

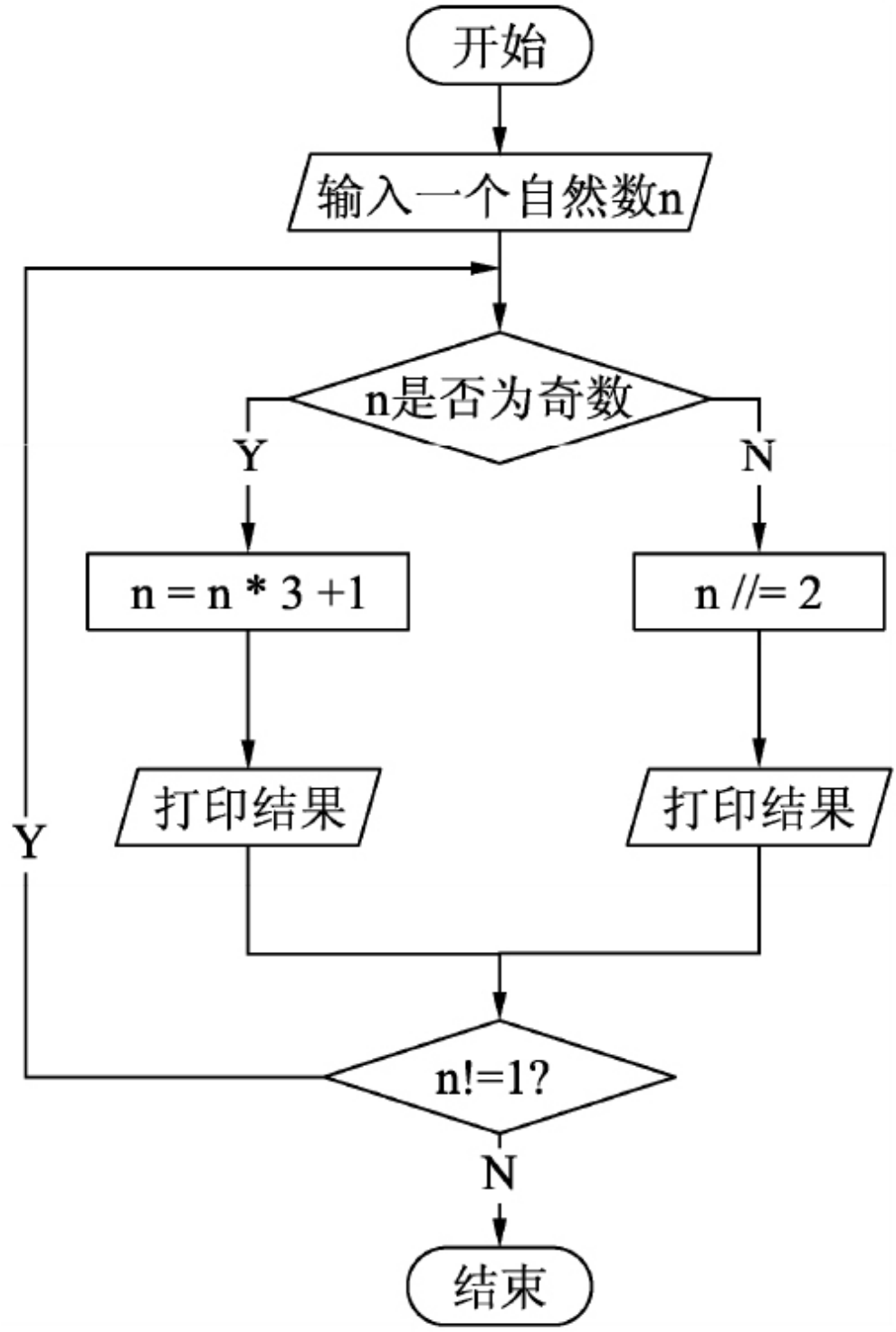


图10.9　程序流程图

5．完整的程序

根据上面的分析，编写完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 角谷猜想
# 任给一个自然数，若为偶数则除以2，若为奇数则乘以3再加1
# 这样得到一个新的自然数之后再按照前面的法则继续演算，若干次以后得到的结果必然为1

if __name__ == "__main__":
    count = 0
    n = int(input("请输入一个自然数："))
    while n != 1:                                                    # 当n=1时终
止循环
        if n % 2 == 1:                                               # 若n为奇数，则乘以3加
1
            n = n * 3 + 1
            count += 1
            # 输出执行步骤
            print("[%d]: %d * 3 + 1 = %d " %(count, (n-1)//3, n))
        else:
            n //= 2                                                  # 若n为偶数，则直接除
以2
            count += 1
            print("[%d]: %d / 2 = %d " %(count, 2 * n, n))  # 输出执行步骤
```

6. 运行结果

在PyCharm下运行程序，结果如图10.10所示。在图10.10a中打印出了n=6时按照角谷猜想的执行步骤，最后结果为1；在图10.10b中打印出了n=15时按照角谷猜想的执行步骤，最后结果也为1。因此，它们都符合角谷猜想的结论。读者也可以任意输入其他自然数进行验证。

```
E:\code\python\Interest-python\venv\
请输入一个自然数：6
[1]: 6 / 2 = 3
[2]: 3 * 3 + 1 = 10
[3]: 10 / 2 = 5
[4]: 5 * 3 + 1 = 16
[5]: 16 / 2 = 8
[6]: 8 / 2 = 4
[7]: 4 / 2 = 2
[8]: 2 / 2 = 1

Process finished with exit code 0
```

a）运行结果 1

```
E:\code\python\Interest-python\venv
请输入一个自然数：15
[1]: 15 * 3 + 1 = 46
[2]: 46 / 2 = 23
[3]: 23 * 3 + 1 = 70
[4]: 70 / 2 = 35
[5]: 35 * 3 + 1 = 106
[6]: 106 / 2 = 53
[7]: 53 * 3 + 1 = 160
[8]: 160 / 2 = 80
[9]: 80 / 2 = 40
[10]: 40 / 2 = 20
[11]: 20 / 2 = 10
[12]: 10 / 2 = 5
[13]: 5 * 3 + 1 = 16
[14]: 16 / 2 = 8
[15]: 8 / 2 = 4
[16]: 4 / 2 = 2
[17]: 2 / 2 = 1

Process finished with exit code 0
```

b）运行结果 2

图10.10　运行结果

10.6　π的近似值

1．问题描述

使用蒙特卡罗法求π的近似值。

蒙特卡罗方法或称计算机模拟方法，是一种基于“随机数”的计算方法。这一方法源于美国在第二次世界大战时研制原子弹的“曼哈顿计划”。该计划的主持人之一数学家冯·诺依曼用驰名世界的赌城——摩纳哥的蒙特卡罗来命名这种方法，为它蒙上了一层神秘的色彩。

蒙特卡罗方法的思路是，在一个单位边长的正方形中，以边长为半径，以一个顶点为圆心，在这个正方形上作四分之一圆。在正方形中随机地投入很多点，使所投入的点落在正方形中每一个位置的机会相等。若点落入四分之一圆内则计数。重复地向正方形中投入足够多的点，用落入四分之一圆内的点数除以总的点数，得到的就是π的四分之一的近似值。

2．问题分析

使用随机函数random()随机产生两个小数x、y构成一个坐标点(x,y)，假设正方形的边长为100，则判断坐标点是否落在四分之一圆内的条件是x2+y2≤10000，其中0≤x≤100，0≤y≤100。若总共向正方形中投放了N个点，而落在四分之一圆内部的点为d个，则π=4*d/N。

需要注意的是，蒙特卡罗方法是使用随机模拟实验结果进行统计来求π的近似值的方法。因此使用该方法所求出的π值只有当统计次数足够多时才会准确，在统计次数较少时会存在一定的误差。

3．算法设计

我们使用Python的random模块中的函数来生成随机数。关于random模块的相关方法，在第4章已经介绍过了，此处不再陈述。在本程序中，我们使用random模块的random()函数来生成两个小数，构成一个点的坐标，代码如下：

```
# 随机生成[0, 1)之间的数
x, y = random(), random()
```

然后利用Python的math模块的sqrt()函数来判断这个点是否落在圆内，代码如下：

```
# 蒙特卡罗法求解
dist = math.sqrt(x**2+y**2)
if dist <= 1.0:
    hits = hits+1
```

4. 确定程序框架

程序的流程图如图10.11所示。

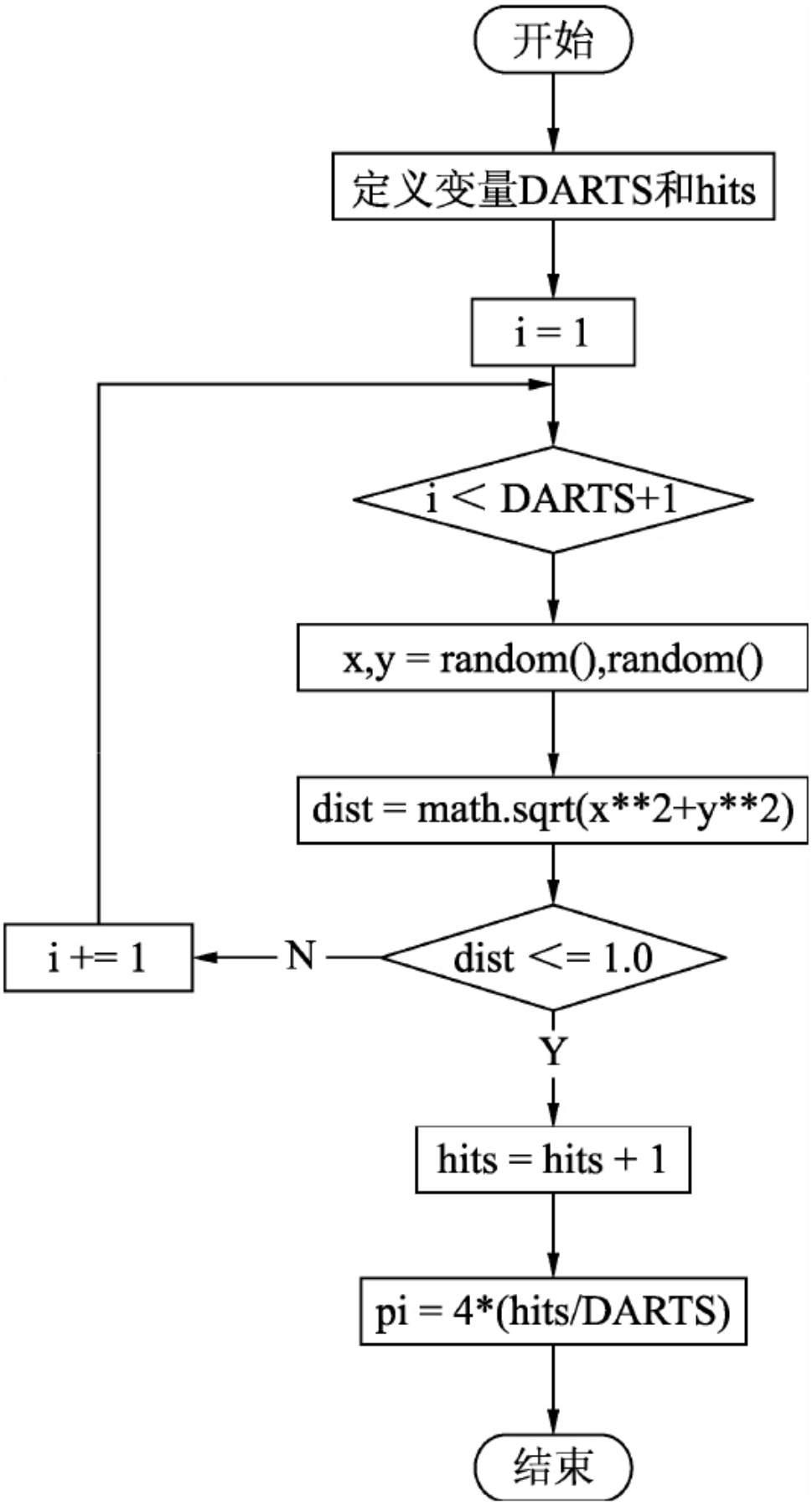

开始
定义变量DARTS和hits
i = 1
i < DARTS+1
x,y = random(),random()
dist = math.sqrt(x**2+y**2)
dist <= 1.0
N
i += 1
Y
hits = hits + 1
pi = 4*(hits/DARTS)
结束

图10.11　程序流程图

5. 完整的程序

根据上面的分析，编写完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: π的近似值 —蒙特卡罗法求解

from random import random
import math

if __name__ == "__main__":
    DARTS = 300000000        # 落入正方形的总的点数，此数越大，越逼近π的近似值
    hits = 0.0                                                    # 落入四分之
一圆的点数
    for i in range(1, DARTS+1):
        x, y = random(), random()            # 随机生成[0, 1)之间的数
        dist = math.sqrt(x**2+y**2)                  # 蒙特卡罗法求解
        if dist <= 1.0:
            hits = hits+1
    pi = 4*(hits/DARTS)                      # 计算π的近似值
    print("pi的值{}.".format(pi))
```

6. 运行结果

在PyCharm下运行程序，共运行了3次，第一次得到的π值为3.14162572，第二次得到的π值为3.141629573，第三次得到的π值为3.1414737466，显示结果如图10.12所示。根据显示结果可知，由于蒙特卡罗法是采用统计规律来计算π值，因此求出的是π的近似值，仅当统计次数足够大时，才会更接近π值。

```
E:\code\python\Interest-python\venv\
pi的值3.14162572.

Process finished with exit code 0
```

a）运行结果 1

```
E:\code\python\Interest-python\venv\
pi的值3.1416295733333333.

Process finished with exit code 0
```

b）运行结果 2

```
E:\code\python\Interest-python\venv\
pi的值3.1414737466666667.

Process finished with exit code 0
```

c）运行结果 3

图10.12　运行结果

第11章　趣味图形

Python是一门功能强大的计算机语言，它不仅可以处理字符和数值，还可以绘制图形，被广泛应用在数据分析领域，以及数据挖掘和机器学习等领域。Python之所以功能如此强大，是因为它有非常丰富的库，可以在很多领域应用。本章我们使用Python来绘制一些基本图形，这些基本图形又可以组合出复杂的图形，从而带领读者进入Python绘图领域。

本章简要介绍了Python语言的绘图功能和常用的图形函数，通过对本章内容的学习，相信读者也能使用Python语言绘制出美丽的图形。

11.1　画直线

1．问题描述

使用Python的turtle（海龟绘图）模块提供的函数绘制直线。

2．问题分析

一幅复杂的图形通常都可以由点、直线、三角形、矩形、平行四边形、圆、椭圆和圆弧等基本图形组成。其中的三角形、矩形、平行四边形又可以由直线组成，而直线又是由两个点确定的。

我们使用Python的turtle模块所提供的函数来绘制直线。在使用之前我们先介绍一下turtle模块的相关知识点。

turtle模块提供面向对象和面向过程两种形式的海龟绘图基本组件。面向对象的接口类如下：

1）TurtleScreen类：定义图形窗口作为绘图海龟的运动场。它的构造器需要一个tkinter.Canvas或ScrolledCanvas作为参数。应在turtle作为某个程序的一部分时使用。

Screen()函数返回一个TurtleScreen子类的单例对象。该函数在turtle作为独立绘图工具时使用。作为一个单例对象，其所属的类是不可以被继承的。

TurtleScreen/Screen的所有方法还存在对应的函数，即作为面向过程的接口组成部分。

2）RawTurtle类：定义海龟对象在TurtleScreen上绘图。它的构造器需要一个Canvas、ScrolledCanvas或TurtleScreen作为参数，以指定RawTurtle对象在哪里绘图。从RawTurtle派生出子类Turtle，该类对象在Screen实例上绘图，如果实例不存在，则会自动创建。

下面分类介绍常用的turtle方法。

（1）海龟动作

首先介绍常用的移动和绘制相关的方法。

• turtle.forward(distance)：参数distance是一个整型或浮点型数值，该方法用于设置画笔前进distance指定的距离，方向为画笔的朝向，可简写为turtle.fd(distance)。

• turtle.backward(distance)：参数distance是一个整型或浮点型数值，该方法用于设置画笔后退distance指定的距离，方向与画笔的朝向相反，可简写为turtle.bk(distance)或turtle.back(distance)。

• turtle.right(angle)：参数angle是一个整型或浮点型数值，该方法用于设置画笔右转angle个单位，单位默认为角度，可简写为turtle.rt(angle)。

• turtle.left(angle)：参数angle是一个整型或浮点型数值，该方法用于设置画笔左转angle个单位，单位默认为角度，可简写为turtle.lt(angle)。

• turtle.goto(x,y=None)|setpos(x,y=None)|setposition(x,y=None)：参数x是一个数值或数值对/向量，而参数y是一个数值或None，该方法用于设置画笔移动到一个绝对坐标，如果画笔已落下将会画线，它不会改变海龟的朝向。

• turtle.setx(x)：参数x是一个整型或浮点型数值，该方法用于设置海龟的横坐标为x，纵坐标保持不变。

• turtle.sety(y)：参数y是一个整型或浮点型数值，该方法用于设置海龟的纵坐标为y，横坐标保持不变。

• turtle.setheading(to_angle)：参数to_angle是一个整型或浮点型数值，该方法用于设置画笔的朝向为to_angle，可简写为

turtle.seth(to_angle)。

• turtle.home()：该方法设置画笔返回原点(0,0)，并设置朝向为初始方向。

• turtle.circle(radius,extent=None,steps=None)：用于绘制圆形。参数radius是一个数值，表示所画圆的半径。extent是一个数值或None，为一个夹角，用来决定绘制圆的一部分，如果不指定extent参数，则会绘制整个圆；如果*extent不是完整圆周，则以当前画笔位置为一个端点绘制圆弧。如果radius为正值则朝逆时针方向绘制圆弧，否则朝顺时针方向绘制圆弧。最终海龟的朝向会依据extent的值而改变。圆实际是以其内切正多边形来近似表示的，其边的数量由steps指定，如果未指定边数则会自动确定。此方法也可用来绘制正多边形。

• turtle.dot(size=None,*color)：参数size是一个大于等于1的整型数；参数color是一个颜色字符串或颜色数值元组，用于设置颜色。该方法用于绘制一个直径为size、颜色为color的圆点。

• turtle.stamp()：该方法用于绘制一个海龟形状，返回该印章的stamp_id。

• turtle.clearstamp(stamp_id)：参数stamp_id是一个整型数，它必须是之前调用stamp()方法返回的一个数值，该方法用于删除stamp_id指定的印章。

• turtle.clearstamps(n=None)：参数n是一个整型数或是None，该方法用于删除全部或前/后n个海龟印章。如果n为None，则删除全部印章；如果n>0，则删除前n个印章；如果n<0，则删除后n个印章。

• turtle.undo()：该方法用于撤销最近的一个（或多个）海龟动作。

• turtle.speed(speed=None)：参数speed是一个0～10范围内的整型数或速度字符串，该方法用于设置海龟的移动速度。移动速度范围是0～10，速度值从1到10，画线和海龟转向的动画效果逐级加快；当参数speed为0时，表示没有动画效果。具体的速度字符串与速度值的对应关系如下：

• "fastest"：0，最快。

• "fast"：10，快。

• "normal"：6，正常。

• "slow"：3，慢。

• "slowest"：1，最慢。

下面介绍常用的获取海龟的状态的方法。

• turtle.position()：返回海龟当前的坐标(x,y)，可简写为pos()。

• turtle.towards(x,y=None)：参数x是一个数值或数值对/矢量，或是一个海龟实例，而参数y的取值取决于x，如果x是一个数值，则y就是一个数值，否则y为None。该方法返回海龟当前位置到(x,y)位置或其他海龟位置这条直线的夹角。

• turtle.xcor()：返回海龟的x坐标。

• turtle.ycor()：返回海龟的y坐标。

• turtle.heading()：返回海龟当前的朝向。

• turtle.distance(x,y=None)：参数x是一个数值或数值对/矢量，或是一个海龟实例，而参数y是一个数值，如果x是一个数值，否则为None。该方法返回海龟与给定(x,y)位置，给定矢量或给定其他海龟之间的距离。

下面介绍常用的度量单位设置相关的方法。

• turtle.degrees(fullcircle=360.0)：参数fullcircle是一个数值，该方法用于设置角度的度量单位，即设置一个圆周为多少度，默认值为360°。

• turtle.radians()：用于设置角度的度量单位为弧度，其值等于degrees(2*math.pi)。

（2）画笔控制

下面介绍常用的设置绘图状态的方法。

• turtle.pendown()：设置画笔落下，画笔在移动时将会画线，可简写为turtle.pd()或turtle.down()。

• turtle.penup()：设置画笔抬起，画笔在移动时不画线，可简写为turtle.pu()或turtle.up()。

• turtle.pensize(width=None)和turtle.width(width=None)：参数width是一个正数值，该方法用于设置画笔的粗细为width或返回该值。

• turtle.pen(pen=None,**pendict)：参数pen是一个包含部分或全部下列键的字典，pendict是一个或多个以下列键为关键字的关键字参数。该方法返回或设置画笔的属性，以一个包含以下键值对的“画笔字典”表示。

• "shown"：True/False；

• "pendown"：True/False；

• "pencolor"：颜色字符串或颜色元组；

• "fillcolor"：颜色字符串或颜色元组；

• \"pensize"：正数值；

• "speed"：0～10范围内的数值；

• "resizemode"："auto"、"user"或"noresize"；

• \"stretchfactor"：(正数值,正数值)；

• "outline"：正数值；

• "tilt"：数值。

这个字典可以作为pen()函数的参数，为画笔设置多个属性。

• turtle.isdown()：返回一个布尔值，如果画笔落下，则返回True，如果画笔抬起，则返回False。

下面介绍常用的颜色控制相关的方法。

• turtle.color(*args)：用于返回或设置画笔颜色和填充颜色。参数args有多种输入格式，具体如下：

• color()：返回以一对颜色描述字符串或元组表示的当前画笔颜色和填充颜色，两者可分别由pencolor()和fillcolor()返回。

• color(colorstring)、color((r,g,b))和color(r,g,b)：输入格式与pencolor()相同，同时设置填充颜色和画笔颜色为指定的值。

• color(colorstring1,colorstring2)和color((r1,g1,b1),(r2,g2,b2))：相当于pencolor(colorstring1)加fillcolor(colorstring2)，使用其他输入格式的方法也与之类似。

• turtle.pencolor(*args)：用于返回或设置画笔颜色，参数args具有4种输入格式，具体如下：

• pencolor()：返回以颜色描述字符串或元组表示的当前画笔颜色，可用作其他color/pencolor/fillcolor调用的输入。

• pencolor(colorstring)：设置画笔颜色为colorstring指定的Tk颜色描述字符串，例如"red"、"yellow"或"#33cc8c"。

• pencolor((r,g,b))：设置画笔颜色为以r,g,b元组表示的RGB颜色。r,g,b的取值范围应为0～colormode，colormode的值为1.0或255。

• pencolor(r,g,b)：设置画笔颜色为以r,g,b表示的RGB颜色。r,g,b的取值范围应为0～colormode。

• turtle.fillcolor(*args)：用于返回或设置填充颜色。参数args具有多种输入格式，具体如下：

• fillcolor()：返回以颜色描述字符串或元组表示的当前填充颜色，可用作其他color/pencolor/fillcolor调用的输入。

• fillcolor(colorstring)：设置填充颜色为colorstring指定的Tk颜色描述字符串，例如"red"、"yellow"或"#33cc8c"。

• fillcolor((r,g,b))：设置填充颜色为以r,g,b元组表示的RGB颜色。r,g,b的取值范围应为0～colormode，colormode的值为1.0或255。

• fillcolor(r,g,b)：设置填充颜色为r,g,b表示的RGB颜色。r,g,b的取值范围应为0～colormode。

下面介绍常用的与填充相关的方法。

• turtle.filling()：返回一个布尔值，表示填充状态。填充为True，否则为False。

• turtle.begin_fill()：在绘制要填充的形状之前调用。

• turtle.end_fill()：填充上次调用begin_fill()之后绘制的形状。

下面介绍常用的与绘图控制相关的方法。

• turtle.reset()：重置。该方法用于从屏幕中删除海龟的绘图，海龟回到原点并设置所有变量为默认值。

• turtle.clear()：清空。该方法用于从屏幕中删除指定海龟的绘图。不移动海龟，海龟的状态和位置以及其他海龟的绘图不受影响。

• turtle.write(arg,move=False,align="left",font=("Arial",8,"normal"))：参数arg表示要书写到TurtleScreen的对象，是一个字符串；move是一个布尔值（True/False），如果move为True，画笔会移动到文本的右下角，默认move为False；align是一个字符串，用于指定对齐方式，其值可以是"left"、"center"或"right"；font用于指定文本字体，是一个三元组(fontname,fontsize,fonttype)。该方法用于书写文本，将arg指定的字符串书写到当前海龟位置。

（3）海龟状态

下面介绍与海龟可见性相关的方法。

• turtle.showturtle()：使得海龟可见，可简写为turtle.st()。

• turtle.hideturtle()：使得海龟不可见，可简写为turtle.ht()。

• turtle.isvisible()：返回一个布尔值，表示海龟是否可见。返回True表示海龟可见，返回False表示海龟不可见。

下面介绍与海龟外观相关的方法。

• turtle.shape(name=None)：参数name是一个有效的形状名字符串。该方法用于设置海龟形状为name指定的形状名，如果没有指定参数name，则会返回当前的形状名。参数name指定的形状名应存在于TurtleScreen的shape字典中。多边形的形状初始值包括"arrow"、"turtle"、"circle"、"square"、"triangle"和"classic"。

• turtle.resizemode(rmode=None)：参数rmode是一个字符串，其值可以是"auto"、"user"或"noresize"之一。该方法用于设置大小调整模式。如果没有指定参数rmode，则返回当前的大小调整模式。大小调整模式含义如下：

• "auto"：根据画笔粗细值调整海龟的外观。

• "user"：根据拉伸因子和轮廓宽度（outline）值调整海龟的外观，两者是由shapesize()设置的。

• "noresize"：不调整海龟的外观大小。

• turtle.shapesize(stretch_wid=None,stretch_len=None,outline=None)和
turtle.turtlesize(stretch_wid=None,stretch_len=None,outline=None)：这两个方法作用相同，其中参数stretch_wid是一个正数值，为垂直于其朝向的宽度拉伸因子；stretch_len是一个正数值，为平行于其朝向的长度拉伸因子，决定形状轮廓线的粗细；outline是一个正数值。这两个方法用于返回或设置画笔的x/y拉伸因子和轮廓。当且仅当大小调整模式设为"user"时海龟会基于其拉伸因子调整外观。

• turtle.shearfactor(shear=None)：参数shear是一个可选的数值，该方法用于设置或返回当前的剪切因子。根据shear指定的剪切因子，即剪切角度的切线来剪切海龟形状，不改变海龟的朝

向（移动方向）。如未指定shear参数，则返回当前的剪切因子，即剪切角度的切线，与海龟朝向平行的线条将被剪切。

• turtle.settiltangle(angle)：参数angle是一个数值。该方法用于旋转海龟形状使其指向angle指定的方向，忽略其当前的倾角，不改变海龟的朝向（移动方向）。

• turtle.tilt(angle)：参数angle是一个数值。该方法用于设置海龟形状自其当前的倾角转动angle指定的角度，但不改变海龟的朝向（移动方向）。

（4）使用事件

下面介绍几个常用的使用事件的方法。

• turtle.onclick(fun,btn=1,add=None)：参数fun是一个函数，调用时将传入两个参数，表示在画布上单击的坐标；btn是一个鼠标按钮编号，默认值为1，表示鼠标左键；add是一个布尔值（True/False），如果为True则添加一个新绑定，否则将取代先前的绑定。该方法将fun指定的函数绑定到鼠标单击此海龟事件。如果fun值为None，则移除现有的绑定。

• turtle.onrelease(fun,btn=1,add=None)：该方法的参数含义与turtle.onclick()方法的相同。该方法表示将fun指定的函数绑定到在此海龟上释放鼠标按键事件。如果fun值为None，则移除现有的绑定。

• turtle.ondrag(fun，btn=1,add=None)：该方法的参数与turtle.onclick()方法和turtle.onrelease()方法的参数含义相同。该方法表示将fun指定的函数绑定到在此海龟上移动鼠标事件。如果fun值为None，则移除现有的绑定。

（5）特殊海龟方法

下面介绍几个特殊海龟的方法。

• turtle.begin_poly()：用于开始记录多边形的顶点。当前海龟的位置为多边形的第一个顶点。

• turtle.end_poly()：用于停止记录多边形的顶点。当前海龟的位置为多边形的最后一个顶点，它将连线到第一个顶点。

• turtle.get_poly()：用于返回最新记录的多边形。

• turtle.clone()：用于创建并返回海龟的克隆体，具有相同的位置、朝向和海龟属性。

• turtle.getturtle()和turtle.getpen()：返回海龟对象自身，常用来作为一个函数返回“匿名海龟”。

• turtle.getscreen()：用来返回作为海龟绘图场所的TurtleScreen类对象。该对象将可调用TurtleScreen方法。

• turtle.setundobuffer(size)：参数size是一个整型数值或None。该方法用于设置或禁用撤销缓存区。如果size为一个整型数值，则将开辟一个指定大小的空缓冲区。size表示可使用undo()方法撤销的海龟命令的次数上限。如果size为None，则禁用撤销缓冲区。

• turtle.undobufferentries()：该方法用于返回撤销缓冲区里的条目数。

• turtle.done():该方法用于关闭turtle，表示绘图完成。

关于turtle模块的更多其他方法在此就不介绍了，感兴趣的读者请自行查阅相关的API文档。

3. 算法设计

在前面的问题分析中，我们介绍了turtle模块的大部分绘图函数，有了这些函数作为基础，下面就来分析本节我们要解决的问题。

要绘制直线，需要用到前面提到的以下几个方法。

- turtle.penup()：画笔抬起。
- turtle.color()：设置海龟或画笔颜色。
- turtle.goto()：画笔移动到一个坐标位置。
- turtle.pendown()：画笔落下。

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 画直线

import turtle

if __name__ == "__main__":
    # 绘制折线
    turtle.penup()                                          # 提笔
    turtle.color('green', 'red')
    turtle.goto(100, 100)
    turtle.pendown()                                    # 落笔

    turtle.goto(100, -100)
    turtle.goto(-100, -100)
    turtle.goto(-100, 100)
    turtle.goto(100, 100)
    turtle.goto(5, 215)
    turtle.goto(-108, 90)

    turtle.done()
```

在PyCharm下运行程序，结果如图11.1所示。

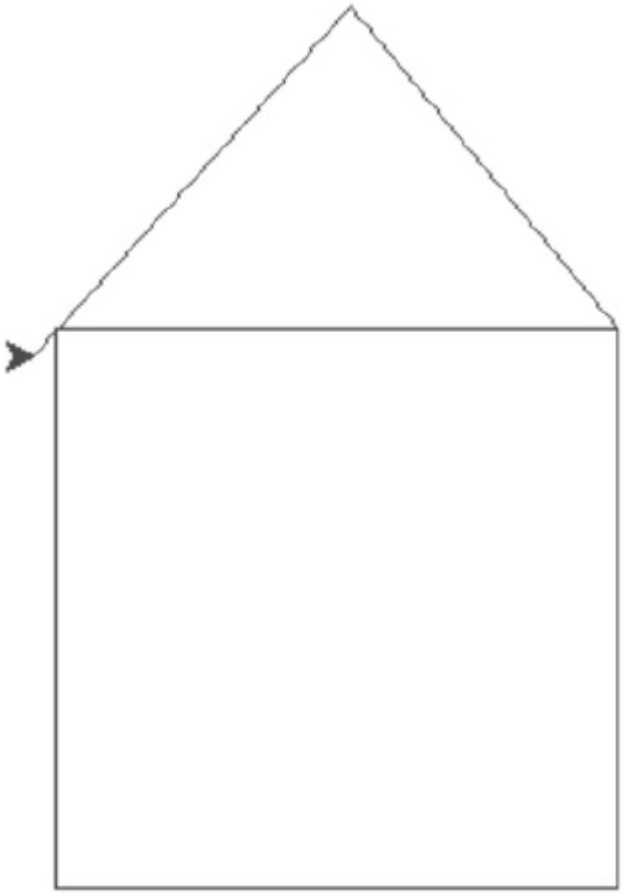


图11.1　运行结果

5. 拓展训练

使用turtle绘制直线，并计算点(x1,y1)到点(x4,y4)之间的距离。

根据前面讲解的turtle模块的相关函数方法，直接进行绘制。

完整的代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 绘图，计算(x1,y1)到(x4,y4)的距离

import turtle
import math

if __name__=="__main__":
    # 定义多个点的坐标
    x1, y1 = 100, 100
    x2, y2 = 100, -100
```

```
x3, y3 = -100, -100
x4, y4 = -100, 100
x5, y5 = 100, 100

# 绘制折线
turtle.penup()                                          # 提笔
turtle.goto(x1, y1)
turtle.pendown()                                # 落笔

turtle.goto(x2, y2)
turtle.goto(x3, y3)
turtle.goto(x4, y4)
turtle.goto(x5, y5)

# 计算起始点和终点的距离
distance = math.sqrt((x1-x4)**2 + (y1-y4)**2)
turtle.write(distance)
turtle.done()
```

在PyCharm下运行程序，结果如图11.2所示。

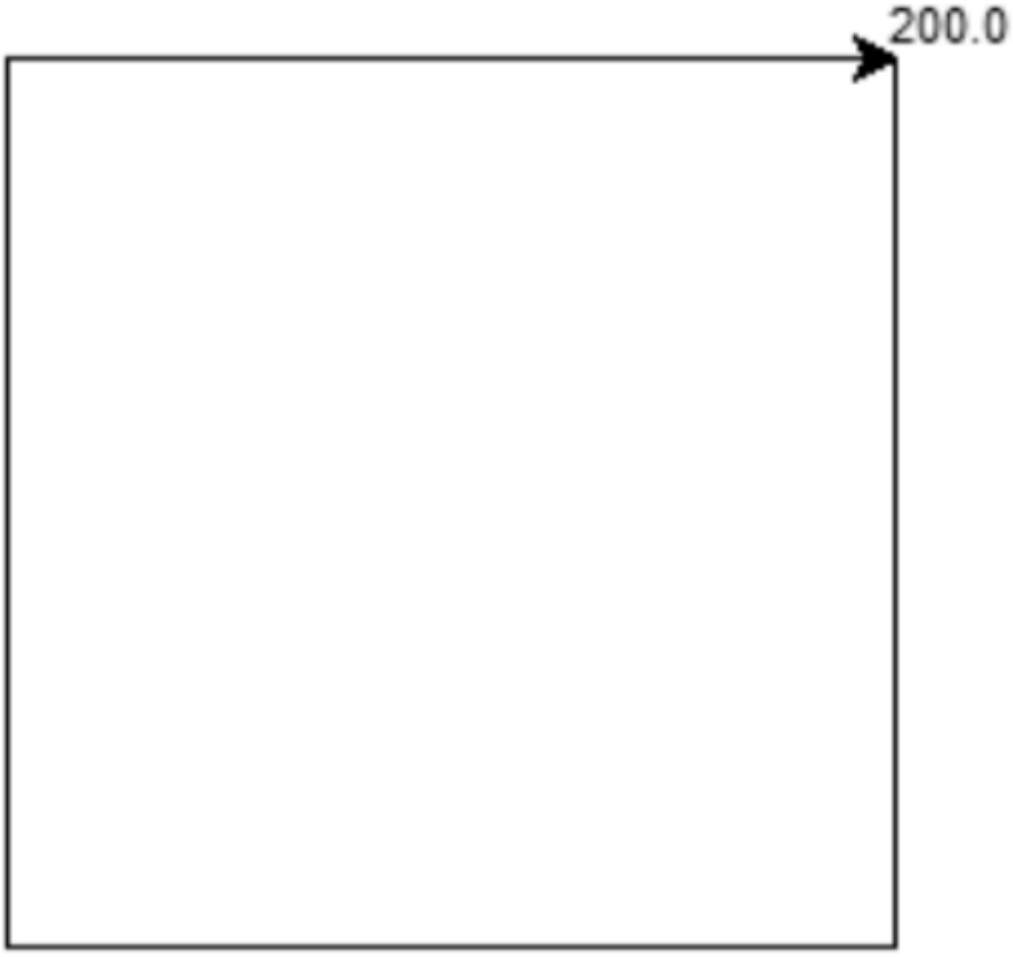


图11.2　运行结果

11.2　画圆和圆弧

1. 问题描述

使用turtle中提供的绘图函数绘制一个笑脸。

2. 问题分析

一个笑脸图形，可以分解为一个大圆（脸部轮廓）、两个小圆（眼睛）、三条直线（鼻子）和一条圆弧（嘴巴）。因此现在要解决的问题是如何在屏幕上画出圆、圆弧等曲线。

3. 算法设计

要绘制圆形、圆弧等图形，可以使用turtle模块提供的函数方法，在前面我们详细地介绍了turtle的函数方法，这里使用turtle的如下方法来绘制所需的图形。

- turtle.width(2)：设置宽度。
- turtle.color("black")：设置画笔颜色。
- turtle.circle(120)：画圆。
- turtle.penup()：抬笔。
- turtle.goto(-60,130)：移动到坐标点。
- turtle.pendown()：画笔落下。
- turtle.setheading(90)：设置朝向，向北90°。
- turtle.forward(len)：前进len指定的距离。
- turtle.left(3)：左转3个单位。

• turtle.done()：关闭turtle。

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 画圆和圆弧

import turtle

if __name__ == "__main__":
    # 画脸
    turtle.width(2)                                  # 设置宽度
    turtle.color("black")                        # 设置画笔颜色
    turtle.circle(120)                               # 画圆

    # 画眼睛
    turtle.penup()                                                  # 抬笔
    turtle.goto(-60, 130)                        # 移动到坐标点
    turtle.pendown()                                 # 下笔
    turtle.color("black")
    turtle.circle(20)

    turtle.penup()                                                  # 抬笔
    turtle.goto(60, 130)                         # 移动到坐标点
    turtle.pendown()                                 # 下笔
    turtle.color("black")
    turtle.circle(20)

    # 画鼻子
    turtle.penup()                                   # 抬笔
    turtle.goto(0, 120)                          # 移动到坐标点
    turtle.pendown()                                 # 下笔
    turtle.goto(-50, 70)
    turtle.goto(50, 70)
    turtle.goto(0, 120)

    # 画嘴巴
    turtle.penup()                                                  # 抬笔
    turtle.goto(-60, 45)                         # 移动到坐标点
    turtle.pendown()                                 # 下笔
    #turtle.circle(90, extent= 90)
    turtle.setheading(90)                        # 设置朝向
    len = 1                                      # 设置初始走的速度为1
    for j in range(60):
        if j > 30:       # 当j<30，也就是画前一半的弧线
            len += 0.2               # 让速度越走越快
        else:                                    # 画后一半弧线
            len -= 0.2               # 让速度越走越慢
        turtle.forward(len)          # 前进
        turtle.left(3)               # 左转
    turtle.goto(-60, 45)

    turtle.done()                    # 关闭
```

在PyCharm下运行程序，结果如图11.3所示。

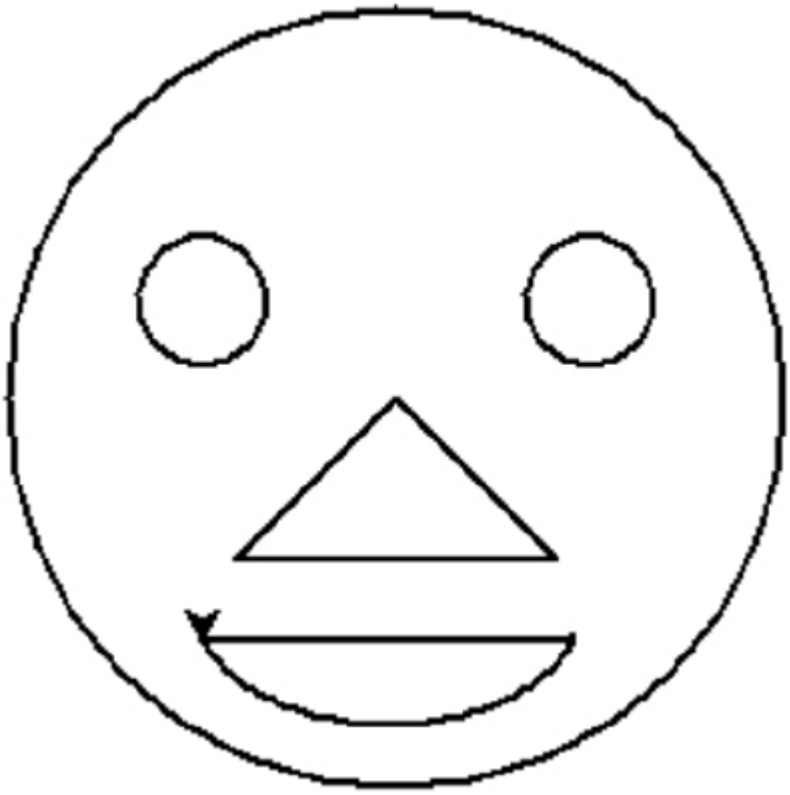


图11.3　运行结果

11.3　画彩色图形

1. 问题描述

使用turtle绘制一个彩色多边形图形。

2. 问题分析

这里绘制一个彩色的六边形，可以使用一个颜色列表，为这个六边形设置多种颜色。我们可以设计一个算法来实现，具体见算法分析。

3. 算法分析

这里使用一个for循环来实现绘制六边形，让画笔不停地绘制，同时这个六边形会逐渐变大。在for循环内设置画笔的颜色，使得每条边都是不同的颜色，并且随着循环次数的不断增加，画笔不断地前进、左转，同时画笔宽度也在不断变化。算法如下：

```
sides=6
# 图形边数
colors=["red","yellow","green","blue","orange","purple"]          # 颜色列表
for x in range(360):
    t.pencolor(colors[x%sides])
# 设置画笔颜色
    t.forward(x*3/sides+x)
# 前进
    t.left(360/sides+1)
# 左转
    t.width(x*sides/200)
# 画笔宽度
```

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 画彩色图形

import turtle
```

```
if __name__ == "__main__":
    t = turtle.Pen()                                                    # 启用画笔
    turtle.bgcolor("white")                                  # 设置画板背景颜色
    sides=6                                                                    # 图形边数
    colors=["red","yellow","green","blue","orange","purple"]          # 颜色列表
    for x in range(360):
        t.pencolor(colors[x%sides])                          # 设置画笔颜色
        t.forward(x*3/sides+x)                               # 前进
        t.left(360/sides+1)                                  # 左转
        t.width(x*sides/200)                                 # 画笔宽度
```

5. 运行结果

在PyCharm下运行程序，结果如图11.4所示。

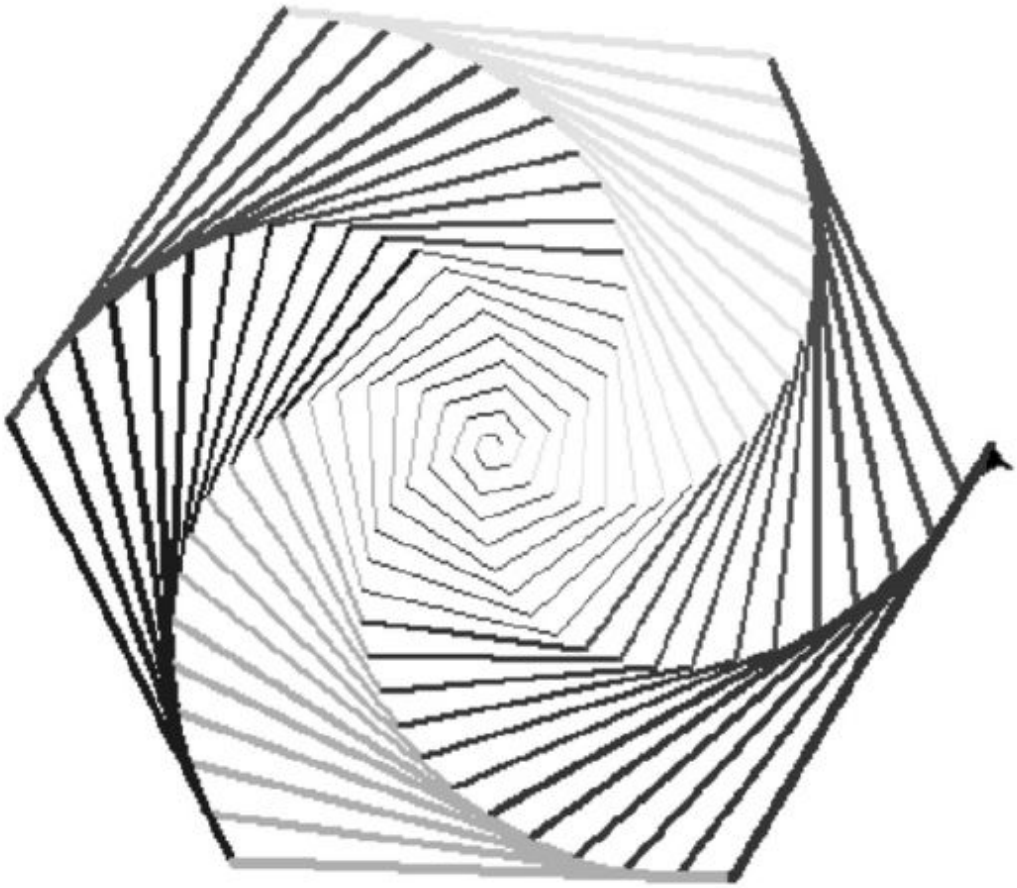


图11.4　运行结果

6. 拓展训练

使用turtle的函数方法绘制一个彩色的奥运五环。

完整的代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @Time : 2019/8/10 14:42
#desc: 绘制奥运五环

import turtle

def huitu_wuhuan():
    turtle.width(8)                                 #设置宽度

    turtle.color("blue")                        #设置画笔颜色
    turtle.circle(50)                           #画圆

    turtle.penup()                                  #抬笔
    turtle.goto(120, 0)                         #移动到坐标点
    turtle.pendown()                                #下笔

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

    turtle.penup()
    turtle.goto(240, 0)
    turtle.pendown()

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

    turtle.penup()
    turtle.goto(60, -50)
    turtle.pendown()

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

    turtle.penup()
    turtle.goto(180, -50)
    turtle.pendown()

    turtle.color("green")
    turtle.circle(50)
    turtle.done()

if __name__=="__main__":
    huitu_wuhuan()
```

在PyCharm下运行程序，结果如图11.5所示。

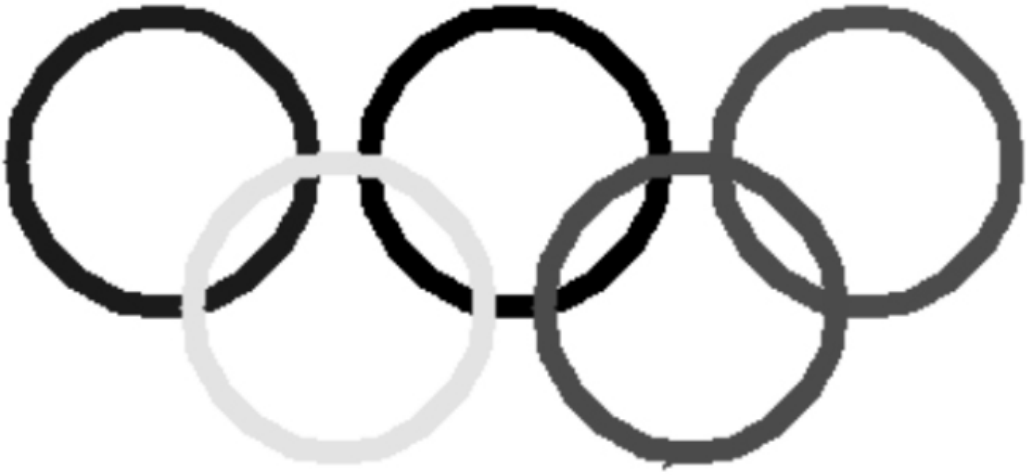


图11.5　运行结果

11.4　绘制余弦曲线

1. 问题描述

绘制一条0～360°（2π）的余弦函数cos(x)曲线。

2. 问题分析

要绘制余弦函数曲线，需要使用到Python语言的NumPy库和matplotlib库，绘制的余弦函数曲线在0～360°（2π）的范围内。

3. 算法设计

该程序的核心部分如下：

1）生成一个0～360°的数组。

```
X = np.linspace(0, 2 * np.pi, 100)                    #生成指定大小的一维数组
```

2）生成余弦值。

```
Y = np.cos(X)                                                    # 返回数
组元组的余弦值
```

3）传入数据绘制余弦函数曲线。

```
# 传入数据
plt.plot(X, Y, linewidth = 4)                          # 设置线宽
plt.plot(X, Y, 'g')                                              # 设置线条为绿色
```

4）显示绘制的图形。

```
# 绘制曲线
plt.show()
```

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 绘制余弦曲线

import numpy as np
import matplotlib.pyplot as plt

if __name__ == "__main__":
    plt.rcParams['font.sans-serif'] = ['SimHei']     # 用于正常显示中文标签
    plt.rcParams['axes.unicode_minus'] = False       # 用来正常显示负号
    # 范围0~2π，共100等分
    X = np.linspace(0, 2 * np.pi, 100)                                    #生成指定大小
的一维数组
    Y = np.cos(X)
# 返回数组元组的余弦值

    # 传入数据
    plt.plot(X, Y, linewidth = 4)                                         # 设置线宽
    plt.plot(X, Y, 'g')                                                          #
设置线条为绿色

    plt.title("余弦曲线图")

    # 保存图片
    plt.savefig("reuslt1.png")
    # 绘制曲线
    plt.show()
```

5. 运行结果

在PyCharm下运行程序，结果如图11.6所示。

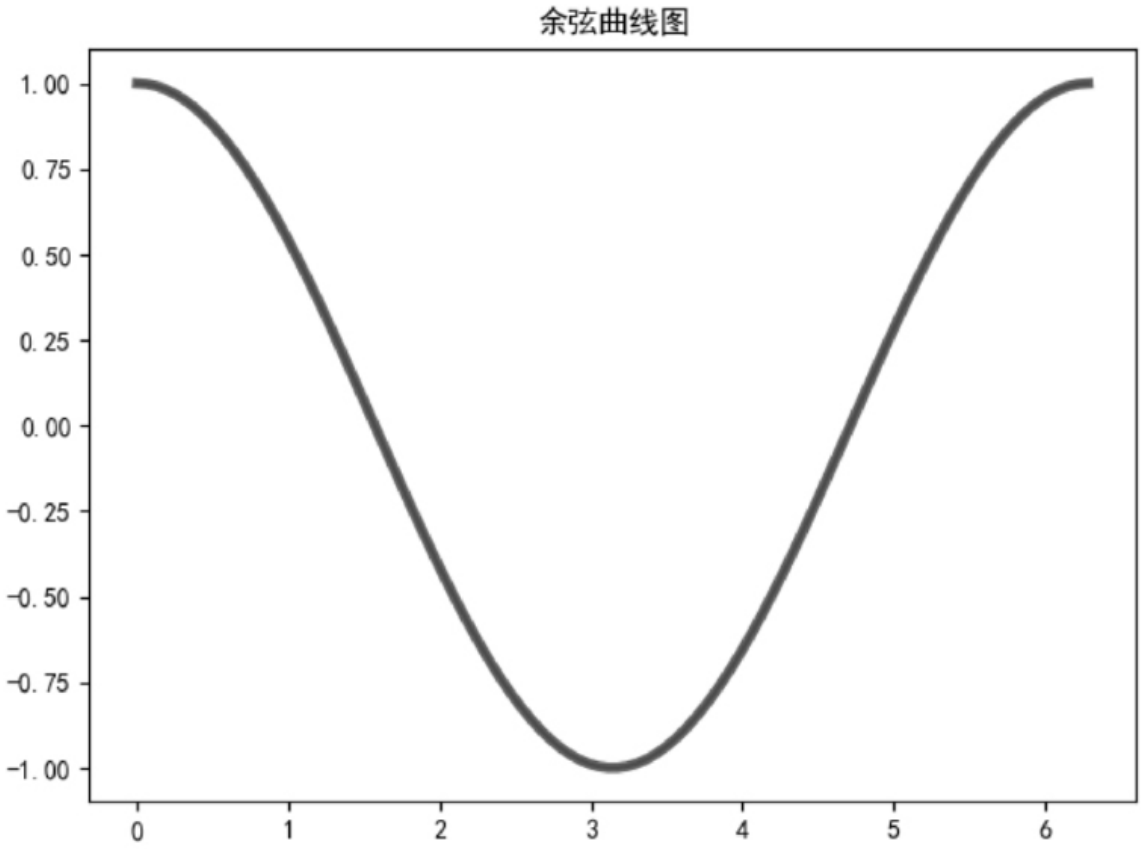


图11.6　运行结果

6．补充知识点

（1）NumPy库

NumPy库是Python科学计算库的基础，它定义了多（N）维数组对象，并且实现了大量的科学运行，例如向量运算、线性代数运算、傅里叶变换、随机数生成等。另外，它能与C/C++和FORTRAN代码集成，并实现一些复杂功能。

NumPy库常用于科学计算，除此之外，还可以用作多维度普通数据的容器，也可以定义任意类型的数据，使得它可以与各种类型的数据分析库进行无缝连接。

NumPy库的主要对象是一个均匀的多维数组，它通过众多的方法来灵活地操作数组。多维数组类似于表格，表格内的数据都是同一

类型的，一般是数字。N维数组在NumPy中是一个ndarray类，该类具有以下几个重要属性：

• ndarray.ndim：数组的维度。

• ndarray.shape：数组的形状，返回一个元组，表示数组在各个方向上的长度。例如一个n行m列的矩阵，shape值是(n,m)。

• ndarray.size：数组的大小，该值等于shape元组内各个元素的乘积。

• ndarray.dtype：数组内元素的数据类型。这里的数据类型可以是Python的标准数据类型，也可以是NumPy提供的常用数据类型。

• ndarray.itemsize：数组内的元素占用计算机内存的大小。

• ndarray.data：数组在内存中的地址。

实例代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: numpy库

import  numpy as np

if __name__ == "__main__":
    # 生成一维数组
    a = np.array([1,2,3,4,5,6,7,8,9])
    print("a = ", a)

    print("数组的维度：", a.ndim)

    print("数组的形状：", a.shape)

    print("数组的大小：", a.size)

    print("数组内元素的数据类型：", a.dtype)

    print("数组内元素占计算机内存的大小：", a.itemsize)

    print("数组在内存中的地址：", a.data)

    # 生成二维数组
    b = np.array([[1,3,5],[2,4,6],[7,8,9]])
    print("b = ", b)
    print("数组的维度：", b.ndim)
    print("数组的形状：", b.shape)

    # 生成三维数组，并指定数据类型
    c = np.array([[[1,2,3],[4,5,6],[7,8,9]],[[9,8,7],[6,5,4],[3,2,1]]],
```

```
dtype=int)
    print("c = ", c)
    print("数组的维度：", c.ndim)
    print("数组的形状：", c.shape)
```

在PyCharm下运行程序，结果如图11.7所示。

```
E:\code\python\Interest-python\venv\Scripts\python.exe
a =  [1 2 3 4 5 6 7 8 9]
数组的维度：  1
数组的形状：  (9,)
数组的大小：  9
数组内元素的数据类型：  int32
数组内元素占计算机内存的大小：  4
数组在内存中的地址：  <memory at 0x000002D52ADA0948>
b =  [[1 3 5]
 [2 4 6]
 [7 8 9]]
数组的维度：  2
数组的形状：  (3, 3)
c =  [[[1 2 3]
  [4 5 6]
  [7 8 9]]

 [[9 8 7]
  [6 5 4]
  [3 2 1]]]
数组的维度：  3
数组的形状：  (2, 3, 3)

Process finished with exit code 0
```

图11.7　运行结果

程序中array()函数与Python标准库中的array不同，Python标准库的array只处理一维数组，请不要混淆。

NumPy库的array()函数在生成数组时，会把序列的序列转换为二维数组，序列内嵌套序列的序列会生成三维数组，以此类推。在生成数组时，我们可以显式地声明数组的数据类型。

NumPy库常用的生成数组的函数如下：

• array()：用于把输入的序列（元组、列表等）转换成一维或多维数组，默认情况下会复制输入值。

• zeros()：用于生成元素全为0的数组。

• ones()：用于生成元素全为1的数组。

• zeros_like()：返回一个与输入数组的形状和数据类型相同但元素全为0的数组。

• ones_like()：返回一个与输入数组的形状和数据类型相同但元素全为1的数组。

• arange()：用于生成一维数组序列。

• linspace()：用于生成指定大小的一维数组。

• numpy.random.rand()：用于生成指定大小且元素在[0,1)区间上均匀分布的数组。

• numpy.random.randn()：用于生成服从标准分布的数组。

NumPy库常用的数组函数如下：

• add()：用于实现两个数组元素相加。

• subtract()：用于实现两个数组元素相减。

• multiply()：用于实现两个数组元素相乘。

• divide()：用于实现两个数组元素相除。

• power()：用于实现两个数组的幂运算，第一个数组元素作为底，第二个数组元组作为指数。

• mod()、remainder()：用于实现两个数组元素相除，并返回结果的余数。

• absolute()、fabs()：返回数组元素的绝对值。

• exp()：返回输入数组以e为底的指数结果。

• exp2()：返回输入数组以2为底的指数结果。

• log()：返回数组元素的自然对数。

• log2()：返回数组元素以2为底的对数。

• log10()：返回数组元素以10为底的对数。

• sqrt()：返回数组元素的平方根。

• square()：返回数组元素的平方。

• floor()：作向下取整运算。

• ceil()：作向上取整运算。

• trunc()：截掉小数点后面的数。

• sin()、cos()、tan()：分别返回数组元素的正弦值、余弦值和正切值。

• arcsin()、arccos()、arctan()：分别返回数组元素的反正弦值、反余弦值和反正切值。

• greater(x1,x2)：如果x1大于x2，则返回布尔真值。

• greater_equal(x1,x2)：如果x1大于等于x2，则返回布尔真值。

• less(x1,x2)：如果x1小于x2，则返回布尔真值。

• less_equal(x1,x2)：如果x1小于等于x2，则返回布尔真值。

• not_equal()：如果x1不等于x2，则返回布尔真值。

• equal()：如果x1等于x2，则返回布尔真值。

更多关于NumPy库的相关知识，在这里就不展开讲述了，感兴趣的读者请自行查询相关的API资料。

（2）matplotlib库

matplotlib是一个主要用于绘制二维图形的Python库，它依赖于NumPy库，提供出色的绘图能力。这里重点介绍matplotlib库的pyplot模块。

matplotlib.pyplot模块提供了一些简单的命令，以方便用户创建图形。每个pyplot函数都负责一项工作，例如创建图形、在图形上建立一个绘图区、在绘图区上添加线条和图形标签等。pyplot会自动根据调用的函数记录下绘图的状态，例如当前图形、绘图区域等。

pyplot模块下的常用函数如下：

• pyplot.plot()：用于绘图，根据接收的参数不同，能够绘制不同的图形，例如绘制直线、曲线，设置所绘制线条的宽度、颜色等。

• pyplot.title()：用于设置所绘制图形的标题。

• pyplot.savefig()：以图片的形式保存所绘制的图形。

• pyplot.setp()：用来查看某个线条对象的可设置属性。

• pyplot.gca()：用于返回当前的Axes（绘图区域的一部分）。

• pyplot.gcf()：用于返回当前的图形。

• pyplot.figure()：用于创建图形，是可选的。

• pyplot.subplot()：用于创建子图。该方法包含三个参数，分别是numrows、numcols和fignum。参数fignum的取值范围应该介于1与numrows×numcols之间。numrows和numcols把画布划分成numrows×numcols个方格，fignum定义在哪张子图上进行操作。子图从第一行开始从左向右编号，然后是第二行……，直到最后一行。如果numrows×numcols<10，则可以采用简写的方式，如subplot(1,2,1)可以简写为subplot(121)。

• pyplot.clf()：用于清除当前的图形。

• pyplot.cla()：用于清除当前的Axes。

• pyplot.text()：用于设置图形的文本属性，它会把添加的文本任意地放到图形的空白位置上。

• pyplot.annotate()：用于在图形的特定位置添加文本。

• pyplot.ylabel()：用于添加y轴标签。

• pyplot.xlabel()：用于添加x轴标签。

• pyplot.pie()：用于绘制饼状图。

• pyplot.bar()：用于绘制柱状图。

更多关于matplotlib库的相关函数，请读者自行查阅学习。

7. 拓展训练

绘制一条0～360°（2π）的正弦函数sin(x)曲线。

完整的代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 绘制正弦曲线
```

```
import numpy as np
import matplotlib.pyplot as plt

if __name__ == "__main__":
    plt.rcParams['font.sans-serif'] = ['SimHei']     # 用于正常显示中文标签
    plt.rcParams['axes.unicode_minus'] = False                # 用来正常显示负号
    x = np.arange(0, 2 * np.pi, 0.01)                         # 从0到2π，以0.01步进
    y = np.sin(x)

    plt.rcParams['font.sans-serif'] = ['SimHei']     # 用于正常显示中文标签
    plt.plot(x, y, linewidth=4)                                          # 设置线的宽
度
    # 设置线条颜色，y表示黄色，g表示绿色，b表示黑色，c表示灰色，默认为蓝色
    plt.plot(x, y, 'g')
    plt.title("正弦曲线图")
# 图像标题
    plt.show()
# 绘制
```

在PyCharm下运行程序，结果如图11.8所示。

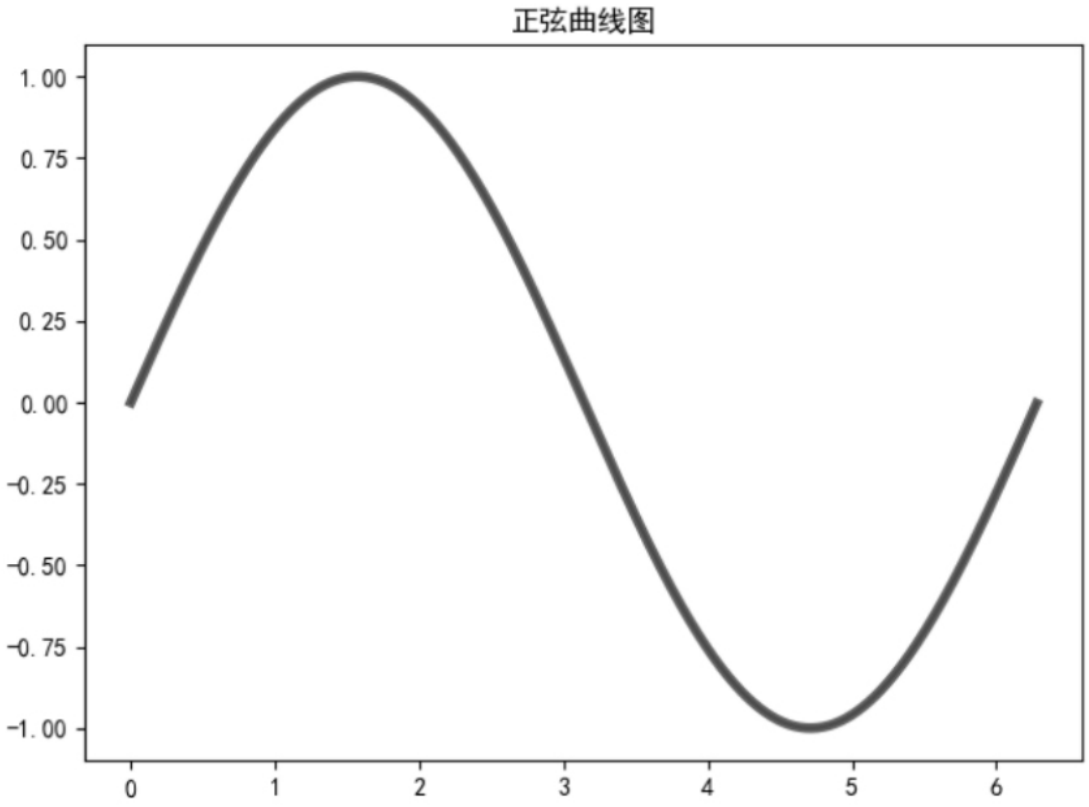


图11.8　运行结果

11.5　绘制空心圆

1. 问题描述

在屏幕上绘制一个空心的圆。

2. 问题分析

绘制圆的方法很多，我们可以根据圆的方程来进行绘制。圆的方程有标准方程和参数方程，这里采用圆的参数方程来绘制圆。

3. 算法设计

圆的参数方程如下：

$$\begin{cases} x = r\cos\theta \\ y = r\sin\theta \end{cases}$$

其中r表示圆的半径，圆心在坐标原点(0,0)上。如果圆心不在坐标原点(0,0)上，而在点(a,b)上，那么就相当于将圆心在(0,0)上的圆平移到圆心为(a,b)上，其原理图如图11.9所示。

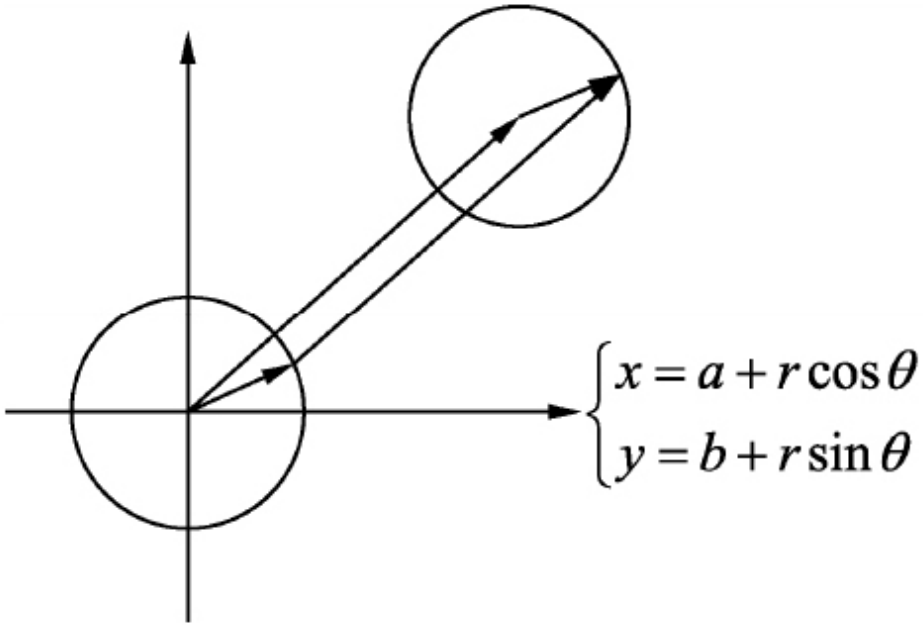


图11.9　圆的参数方程原理图

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 绘制圆形

import numpy as np
import matplotlib.pyplot as plt

if __name__ == "__main__":
    plt.rcParams['font.sans-serif'] = ['SimHei']     # 用于正常显示中文标签
    plt.rcParams['axes.unicode_minus'] = False       # 用来正常显示负号

    # 圆半径
    r = 2.0
    # 圆心坐标
    a , b = (0. , 0.)
    # 参数方程
    Circle = np.arange(0, 2*np.pi, 0.01)
    x = a + r * np.cos(Circle)
    y = b + r * np.sin(Circle)
    fig = plt.figure()
    axes = fig.add_subplot(111)
    axes.plot(x, y)
    axes.axis('equal')
    plt.title("圆形")
    plt.show()
```

5. 运行结果

在PyCharm下运行程序，结果如图11.10所示。

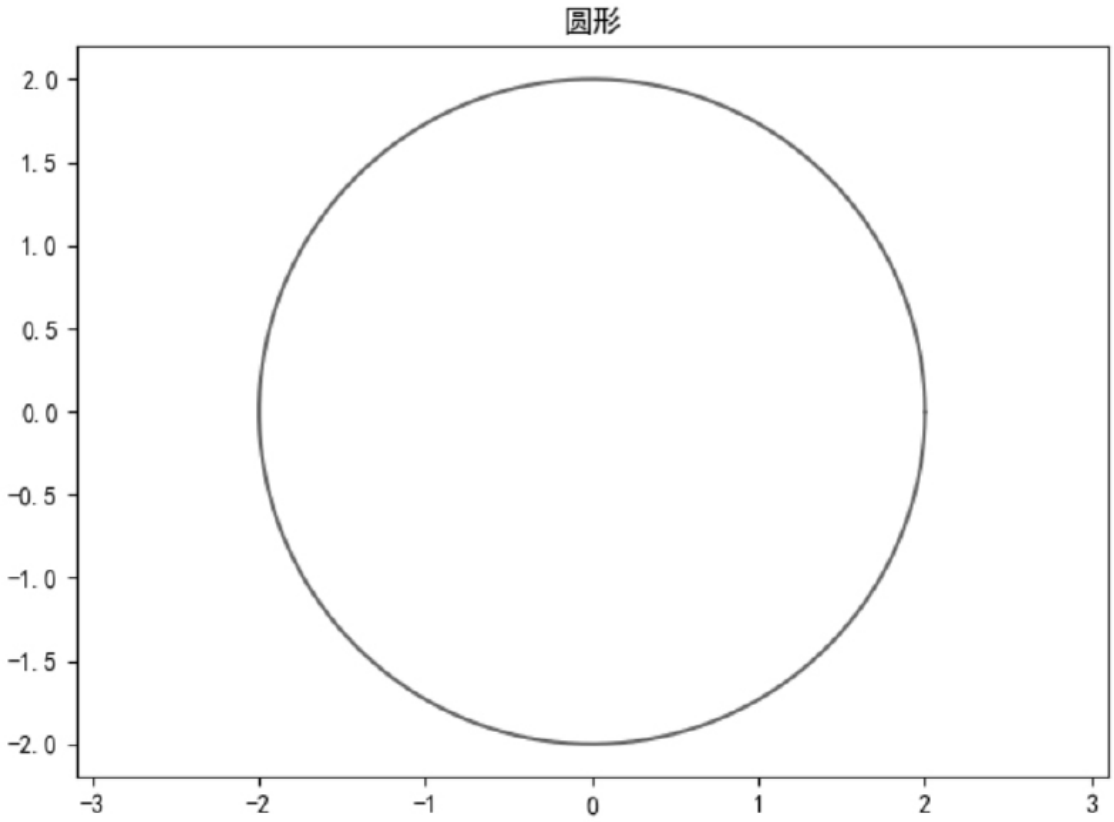


图11.10　运行结果

6. 拓展训练

实现函数$y=x^2-2x+1$的图形与圆的图形叠加显示。

完整的代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 绘制一元二次方程曲线和圆形

import numpy as np
import matplotlib.pyplot as plt

if __name__ == "__main__":
    plt.rcParams['font.sans-serif'] = ['SimHei']     # 用于正常显示中文标签
    plt.rcParams['axes.unicode_minus'] =False                    # 用来正常显示负号
    x = y = np.arange(-8, 8, 0.1)
    x, y = np.meshgrid(x, y)
    # 绘制x2 + y2 = 25 的圆形
    plt.contour(x, y, x ** 2 + y ** 2, [25])

    # 一元二次方程  y = x2 - 2x + 1
    x1 = np.linspace(-3, 3, 50)
```

```
print(x1)
y1 = x1 ** 2
plt.plot(x1, y1, linewidth=4) # 设置线宽
plt.plot(x1, y1, 'g')    # 设置线条为绿色

plt.title("一元二次方程曲线与圆形")
plt.axis('scaled')
plt.show()
```

在PyCharm下运行程序，结果如图11.11所示。

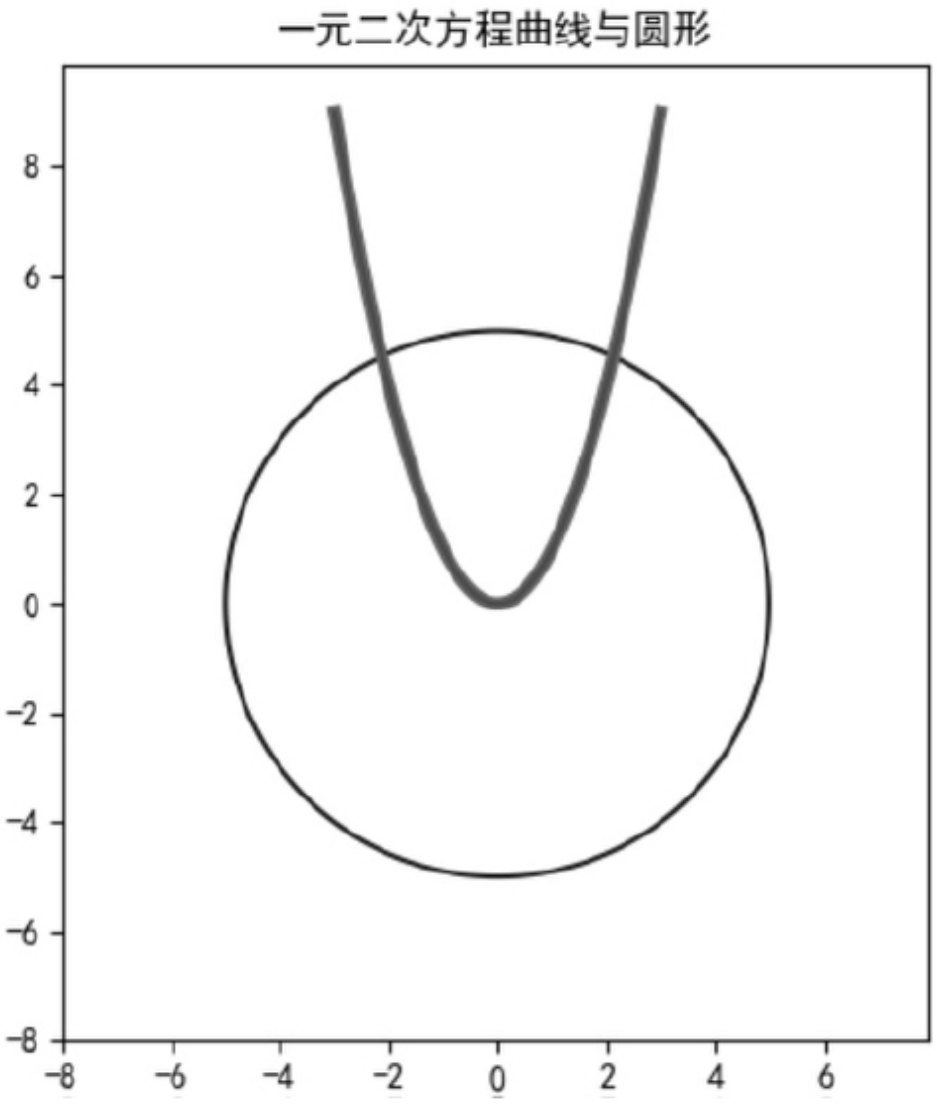


图11.11　运行结果

11.6　绘制空心菱形

1. 问题描述

编程打印如图11.12所示的空心菱形图案。

0	1	2	3	4	5	6	7	8	9
1					*				
2				*		*			
3			*				*		
4		*						*	
5	*								*
1		*						*	
2			*				*		
3				*		*			
4					*				

图11.12　空心菱形

2. 问题分析

该问题要求绘制空心菱形，在此基础上，还可以添加其他要求，如尽可能少地使用printf语句，或者由键盘输入正数n，并绘制出有2*n+1行的空心菱形图案。

那么针对此类问题我们究竟应该从何入手分析呢？观察图11.12可知，图中每一行和每一列的星号和空格所出现的位置并非杂乱无章的，而是都呈现出一定的规律。这样就可以将绘制空心菱形的问

题转换为找出这些星号和空格与它们所在行与列之间存在的某种规律的问题。只要这个规律找到了，那么问题就迎刃而解了。

下面以第5行（n=5）作为对称轴的空心菱形为例来寻找规律。观察图形和积累的经验告诉我们，空心菱形应该分上下两个部分来打印，那么菱形的上下两部分和行列之间有什么联系呢？我们仔细来观察图11.12，首先标示出每一行和每一列的行号和列号，其次寻找上半部分之间的联系。

图11.12中1～5行的规律如表11.1所示，其中行号用i表示，列号用j表示。

表11.1　上部分1～5行的规律表

行号i	星号所在的列j	
1	5	5
2	4	6
3	3	7
4	2	8
5	1	9
行列关系	$6-i$	$4+i$
与n值的关系	$n+1-i$	$n-1+i$

根据表11.1，我们不难写出以下语句：

```
if j == n + 1 - i or j == n - 1 + i:
    print("* ", end="")
else:
    print(" ", end="")
```

再结合行列的循环关系，得到以下代码：

```
# 外层循环控制行数，从控制台输入的参数n即为菱形上半个三角形的行数
for i in range(1, n + 1):
    for j in range(1, (n + i - 1) + 1):
        if j == n + 1 - i or j == n - 1 + i:
            print("* ", end="")
        else:
            print(" ", end="")
    print()
```

接下来，我们观察图11.12中6～9行的规律。为了使寻找到的规律中和行列号建立的关系数值相对简单且小，我们假定行号重新从1开始排列，即我们寻找的是下半部分1～4行的规律，具体如表11.2所示。

表11.2　下部分1～4行的规律表

行号i	星号所在的列j	
1	2	8

（续）

行号i	星号所在的列j	
2	3	7
3	4	6
4	5	5
行列关系	$i+1$	$9-i$
与n值的关系	—	$2n-1+i$

根据表11.2，我们不难写出以下语句：

```
if j == i + 1 or j == 2 * n - 1 - i:
    print("* ", end="")
else:
    print(" ", end="")
```

再结合行列的循环关系，得到以下代码：

```
# 外层循环控制行数，由于下半个三角形比上面的少一行，所以循环变量i的最大值为n-1
for i in range(1, n):
    for j in range(1, (2 * n - 1 - i) + 1):
        if j == i + 1 or j == 2 * n - 1 - i:
            print("* ", end="")
        else:
            print(" ", end="")
    print()
```

找到空心菱形的规律后，程序就很容易编写了，只需要在核心代码的基础上再将程序补充完整即可。

3．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 绘制空心菱形

def draw(n):
    # 外层循环控制行数，从控制台输入的参数n即为菱形上半个三角形的行数
    for i in range(1, n + 1):
        for j in range(1, (n + i - 1) + 1):
            if j == n + 1 - i or j == n - 1 + i:
                print("* ", end="")
            else:
                print(" ", end="")
        print()
    # 打印下半个三角形
    # 外层循环控制行数，由于下半个三角形比上面的少一行，所以循环变量i的最大值为n-1
    for i in range(1, n):
        for j in range(1, (2 * n - 1 - i) + 1):
            if j == i + 1 or j == 2 * n - 1 - i:
                print("* ", end="")
            else:
                print(" ", end="")
        print()

if __name__ == "__main__":
    n = int(input("请输入菱形对称轴的行数n: "))
    draw(n)
```

4. 运行结果

在PyCharm下运行程序，屏幕上提示“请输入菱形对称轴的行数n：”，此处输入5，则打印出以第5行为对称轴的空心菱形，如图11.13所示。

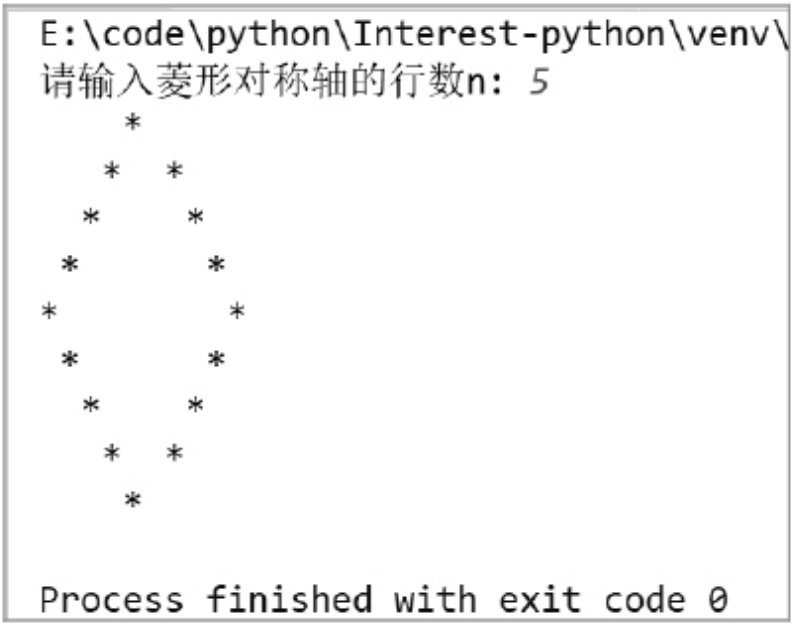


图11.13　运行结果

11.7　填充彩色图形

1. 问题描述

使用turtle绘制填充彩色图形。

2. 问题分析

这里我们使用turtle的方法来绘制一座房子和一个太阳，然后填充上对应的颜色。

3. 算法设计

要绘制一座房子和一个太阳，需要用到turtle模块中的以下方法。

- turtle.Pen()：启动画笔。
- turtle.color()：设置颜色。
- turtle.hideturtle()：隐藏海龟。
- turtle.begin_fill()：开始填充颜色。
- turtle.forward()：前进。
- turtle.left(180-60)：左转120°。
- turtle.right(90)：右转90°。
- turtle.end_fill()：结束填充。
- turtle.penup()：抬笔。
- turtle.pendown()：落笔。

- turtle.goto(100,200)：移动到绝对坐标点。
- turtle.circle(20)：画圆。

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 填充彩色图形

import turtle
import time

if __name__ == "__main__":
    t = turtle.Pen()                                    # 启动画笔
    t.color('green', 'pink')                            # 设置颜色
    t.hideturtle()                                      # 隐藏海龟

    t.begin_fill()                                      # 开始填充颜色
    for x in range(3):
        t.forward(180)                                  # 前进
        t.left(180 - 60)                                # 左转
    t.forward(10)                                       # 直行10个像素
    t.right(90)                                         # 右转90°
    t.end_fill()                                        # 填充结束

    t.color('green', 'brown')
    t.begin_fill()
    for x in range(3):
        t.forward(160)
        t.left(90)
    t.end_fill()

    t.penup()                                           # 抬笔
    t.goto(30, -160)
    t.pendown()                                         # 落笔
    for x in range(3):
        t.right(90)
        t.forward(40)

    t.penup()
    t.color('green', 'red')
    t.begin_fill()
    t.goto(100, 200)
    t.circle(20)                                        # 画圆
    t.end_fill()

    time.sleep(20)
```

5. 运行结果

在PyCharm下运行程序，结果如图11.14所示。

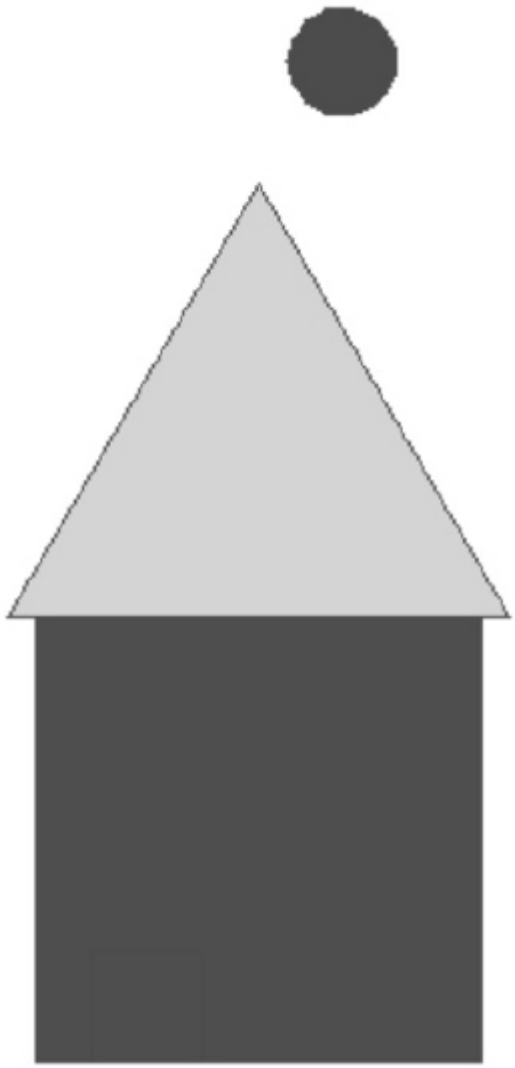


图11.14　运行结果

11.8　绘制饼状图

1. 问题描述

使用matplotlib库的pyplot模块提供的绘图函数绘制饼状图。

2. 问题分析

这里以水果的销量为例，来绘制一个饼状图。

3. 算法分析

要绘制一个饼状图，可以使用matplotlib模块下pyplot所提供的函数方法来进行绘制，具体如下：

```
# 绘制饼状图
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%', shadow= True,
startangle=90 )
```

4. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 绘制饼状图

import matplotlib.pyplot as plt

if __name__ == "__main__":
    plt.rcParams['font.sans-serif'] = ['SimHei']     # 用于正常显示中文标签
    labels = '苹果', '香蕉', '雪梨', '西瓜', '葡萄'
    sizes = [10, 15, 8, 62, 5]
    explode = (0, 0, 0, 0.1, 0)                                  # 分割出第二个分片
    fig1, ax1 = plt.subplots()                                   # 设置多个子图
    ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
shadow=True, startangle=90 )                                        # 绘制饼状图
    ax1.axis('equal')                                                          #
保证饼图绘制出来以后是圆形
    plt.title('水果销量图')                                                  # 设置标
题
    plt.show()
```

在PyCharm下运行程序，结果如图11.15所示。

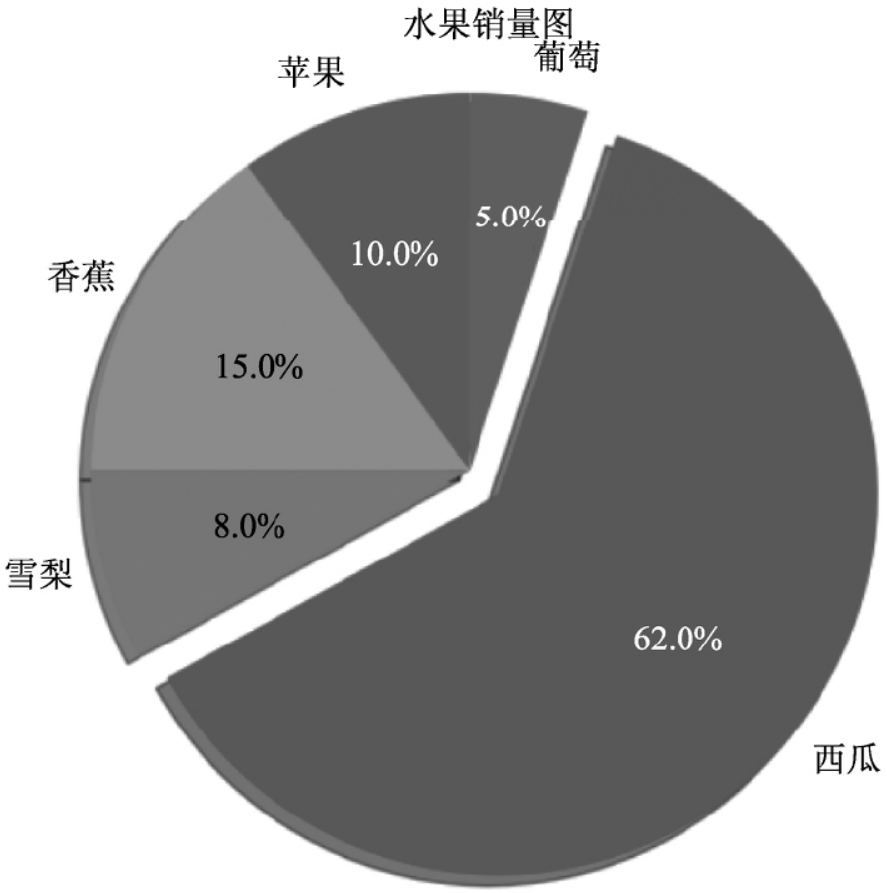


图11.15　运行结果

5. 拓展训练

使用matplotlib库的pyplot模块提供的函数绘制散点图。

关于matplotlib库的pyplot模块提供的函数在前面已经介绍过了，这里不再陈述。

完整的实例代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @Time : 2019/8/10 23:32
# @desc: 绘制散点图

import matplotlib.pyplot as plt
import numpy as np
```

```
# 定义生成y值的函数
def cData(n):
    a1 = np.cos(2 * np.pi * n)
    b1 = np.exp(-n)
    return a1 * b1

if __name__ == "__main__":
    plt.rcParams['font.sans-serif'] = ['SimHei']     # 用于正常显示中文标签
    plt.rcParams['axes.unicode_minus'] = False                # 用来正常显示负号
    d1 = np.arange(0.0, 6.0, 0.3)                                    # 生成一维数
组序列
    point = plt.plot(d1, cData(d1), 'ro')                     # 绘制
    plt.setp(point, 'markersize', 20)                                # 设置数据点
的大小
    plt.setp(point, 'markerfacecolor', 'g')                   # 设置数据点的颜色
    plt.title('散点图')
    plt.show()
```

在PyCharm下运行程序，结果如图11.16所示。

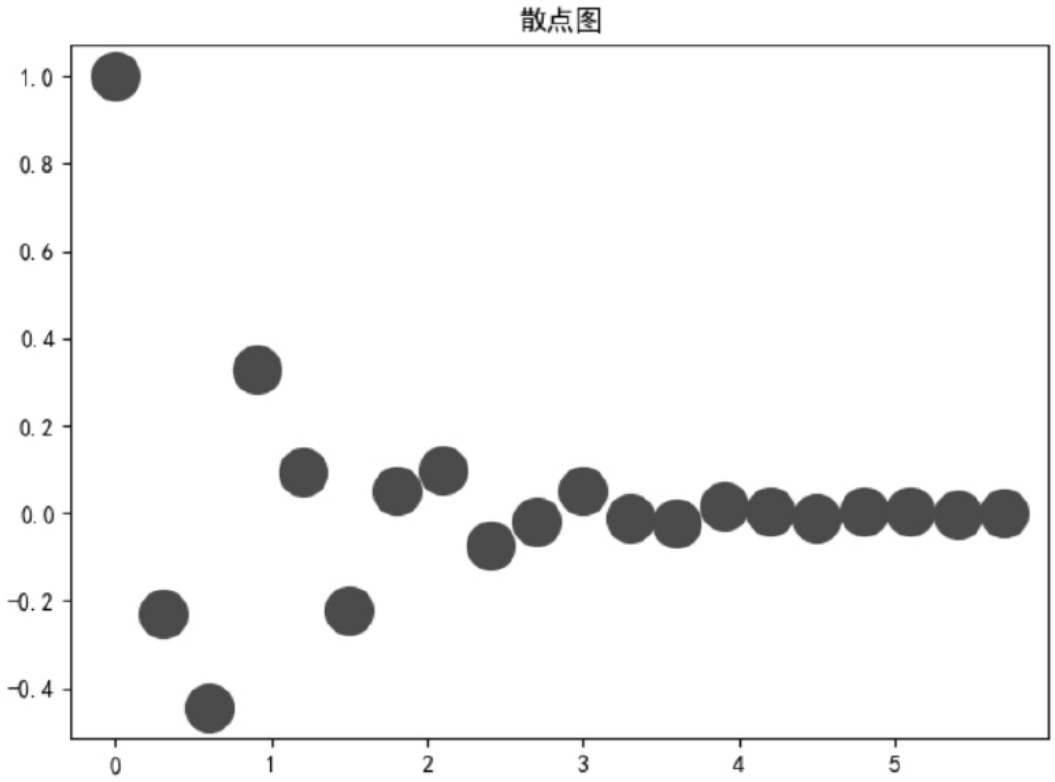


图11.16　运行结果

本章为大家介绍了Python相关的绘图模块的函数，只为起到抛砖引玉的作用，关于更多的绘图相关知识，感兴趣的读者可以继续查阅相关API文档进行深入学习。

第12章　其他趣味问题

在前面各章中我们介绍了趣味算法、趣味数学问题、趣味整数、趣味分数、趣味素数、趣味逻辑推理、趣味游戏、趣味数组、趣味函数递归、定理与猜想以及趣味图形等各类趣味问题。本章在前面各章的基础上再介绍几个趣味问题，这几个问题用到了数组、文件、结构体等知识点，可以使读者进一步巩固前面各章所学到的内容。

12.1　约瑟夫环

1．问题描述

17世纪的法国数学家加斯帕在《数目的游戏问题》中讲到一个故事：15个教徒和15个非教徒在深海上遇险，必须将一半的人投入海中，其余的人才能幸免于难。于是想了一个办法，将30个人围成一个圆圈，从第一个人开始依次报数，每数到第九个人就将他扔入大海，如此循环进行直到仅剩15个人为止。问怎样的排法才能使每次投入大海的都是非教徒？

2．问题分析

本题明显是一个约瑟夫问题。约瑟夫问题的求解方法很多，这里仅给出一种实现方法。

问题描述中说“将30个人围成一个圆圈”，据此可以考虑使用一个循环的链来表示。

3．算法设计

我们使用一个二维数组array[flag][next]来存放30个人，其中flag用来表示是否被扔下海的标记。为1表示没有被扔下海，为0表示已被扔下海。next表示指向下一个人的数组下标。

程序从第一人开始对还未扔下海的人进行计数，每数到9时，就将数组中的标记改为0，表示该人已经被扔下海了，如此循环计数直到有15个人被扔下海为止。

4．确定程序框架

1）定义二维数组。定义一个二维数组用来存放30个人，代码如下：

```
array = [[0] * 2 for i in range(31)]                # 30个人，0号元素没有使用
```

上面的代码定义了二维数组array，该数组有31个元素，数组中的0号元素未使用，1～30号元素分别表示题中所述的30个人。

2）初始化二维数组。二维数组的初始化代码如下：

```
for i in range(1, 31):              # 初始化结构数组
    array[i][0] = 1                             # flag标志置为1，表示人在船上
    array[i][1] = i + 1                 # next的值为数组中下一个元素的下标，即指向下一个人
array[30][1] = 1                                # 第30个人的数组下标指向第一个人以构成环
```

上面的代码使用for循环完成对二维数组的初始化工作。初始情况下，由于30个人都在船上，因此数组中每个元素的flag成员的值都置为1，next成员的值设置为数组中下一个元素的下标，即指向下一个人。需要注意的是，array数组中第30个元素的next成员值需要重新指定为1，用于指向第一个人以构成环。

3）计数15次，每到第九个人就将其扔入大海。实现核心功能的代码如下：

```
j = 30                                  # 变量j指向已经处理完毕的数组元素，从array[i]指向的人开始计数
for i in range(15):                     # 循环变量i作为计数器，记录已扔下海的人数，共15个人
    k = 0
    while True:                 # 循环变量k作为计数器，决定哪个人被扔下海，计数到9为止
        if k < 9:
            j = array[j][1]             # 修改数组下标，取下一个人
            k += array[j][0]                    # 进行计数，已扔下海的人标记为0
        else:
            break                               # 计数到9则停止计数
    array[j][0] = 0                             # 将标记置0，表示该人已被扔下海
```

上面的程序中循环变量i作为计数器，用于记录已扔下海的人数。循环变量k作为计数器，用于找到每次被扔下海的那个人，计数到9为止。每次循环找到被扔下海的那个人后就将其flag标记置0。

程序流程图如图12.1所示。

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 约瑟夫环


if __name__ == '__main__':
    array = [[0] * 2 for i in range(31)]  # 30个人，0号元素没有使用
    print('最终结果为(s:被扔下海,b:在船上):')
    for i in range(1, 31):  # 初始化结构数组
        array[i][0] = 1     # flag标志置为1，表示人在船上
        array[i][1] = i + 1 # next的值为数组中下一个元素的下标，即指向下一个人
    array[30][1] = 1                # 第30个人的数组下标指向第一个人以构成环
    j = 30                          # 变量j指向已经处理完毕的数组元素，从array[i]指向
的人开始计数
    for i in range(15):             # 循环变量i作为计数器，记录已扔下海的人数，共15个
人
        k = 0
        while True:          # 循环变量k作为计数器，决定哪个人被扔下海，计数到9为止
            if k < 9:
                j = array[j][1]             # 修改数组下标，取下一个人
                k += array[j][0]    # 进行计数，已扔下海的人标记为0
            else:
                break                               # 计数到9则停止计数
        array[j][0] = 0                     # 将标记置0，表示该人已被扔下海

    for i in range(1, 31):                          # 输出结果
        if array[i][0]:
            print('b', end='')
        else:
            print('s', end='')
    print()
```

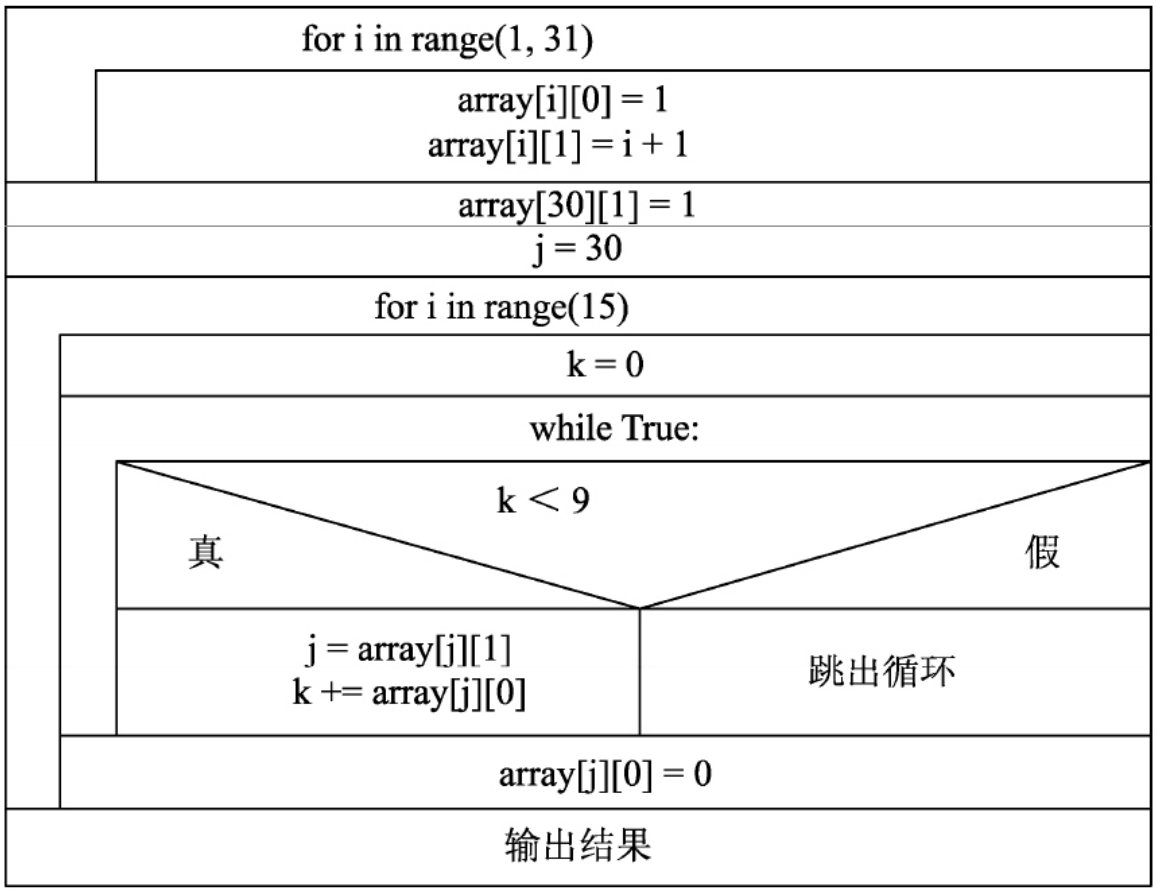


图12.1　程序流程图

6．运行结果

在PyCharm下运行程序，结果如图12.2所示。其中，s表示数组中该位置代表的人将会被扔进海里，b表示数组中该位置代表的人会待在船上。

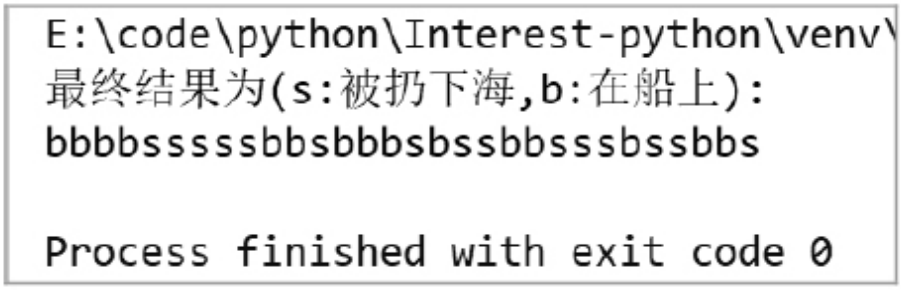


图12.2　运行结果

7．拓展训练

思考题：有n个人围成一圈，顺序排号。从第一个人开始报数（从1到3报数），凡是报到3的人就退出圈子，问最后留下的人是原来的第几号。

12.2　数据加密

1．问题描述

某个公司采用公用电话来传递数据，传递的数据是4位的整数，且要求在传递过程中数据是加密的。数据加密的规则为：将每位传递的数字都加上5，之后用和除以10的余数来代替该数字，最后将第一位和第四位数字交换，第二位和第三位数字交换。

要求通过程序实现数据加密的过程。

2．问题分析

解决该问题只要按照题目中给出的数据加密规则编程即可。

3．算法设计

该问题需要进行数据拆分，将拆分后各位上的数字存放在一个一维数组中。

对拆分后的各位数字应用加密规则时，可使用for循环结构来实现。

4．确定程序框架

（1）数据拆分

定义变量n，用于存放要传递的数据。定义数组s[4]，用于存放拆分后的各位数字。拆分的代码如下：

```
# 获取各位上的数字
s[0] = n % 10                                    # 将个位存入s[0]
s[1] = n % 100 // 10                          # 将十位存入s[1]
s[2] = n % 1000 // 100                      # 将百位存入s[2]
s[3] = n // 1000                                  # 将千位存入s[3]
```

（2）数据加密

数据加密过程采用两个for循环来实现，具体如下：

```
for i in range(0, 4):
    s[i] += 5        # 各位上的数字加5
    s[i] %= 10       # 除以10取余

# 数字交换，1、4位交换，2、3位交换
for i in range(0, (3 // 2) + 1):
    # 数字交换
    t = s[i]
    s[i] = s[3 - i]
    s[3 - i] = t
```

程序流程图如图12.3所示。

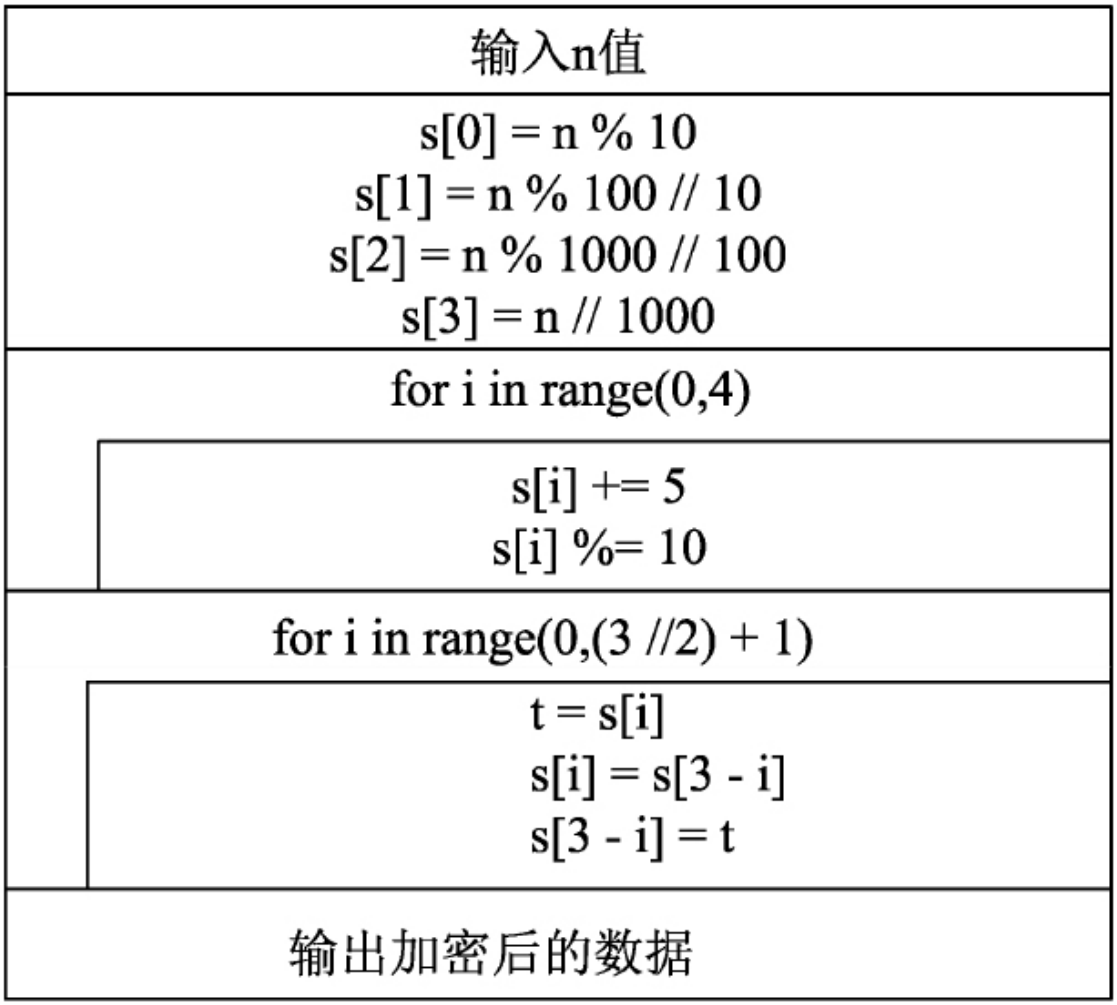


图12.3　程序流程图

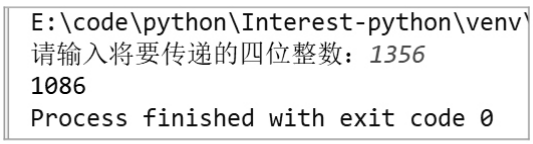
5．完整的程序

根据上面的分析，编写完整的程序如下：

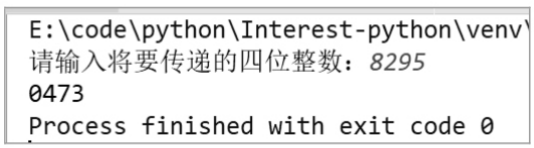
```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 数据加密问题

def encry(n):
    # 获取各位上的数字
    s[0] = n % 10                                   # 将个位存入s[0]
    s[1] = n % 100 // 10                            # 将十位存入s[1]
    s[2] = n % 1000 // 100                          # 将百位存入s[2]
    s[3] = n // 1000                                # 将千位存入s[3]

    for i in range(0, 4):
        s[i] += 5                                   # 各位上的数字加5
        s[i] %= 10                                  # 除以10取余

    # 数字交换，1、4位交换，2、3位交换
    for i in range(0, (3 // 2) + 1):
        # 数字交换
        t = s[i]
        s[i] = s[3 - i]
        s[3 - i] = t

    # 输出加密后的数据
    i = 3
    while i >= 0:
        print("%d" % s[i], end="")
        i -= 1

if __name__ == "__main__":
    s = [0]*4                                       # 数组s用来存放生成的4位数
    n = int(input("请输入将要传递的四位整数："))
    encry(n)
```

6. 运行结果

在PyCharm下运行程序，结果如图12.4所示。在图12.4a中，输入的数据为1356，加密后的数据为1086；在图12.4b中，输入的数据为8295，加密后的数据为0473。



a）运行结果 1



b）运行结果 2

图12.4　运行结果

7. 问题拓展

本题中的数据拆分还可以使用循环结构来实现。下面的代码使用了while循环结构来拆分数据，拆分后的数据同样保存到s数组中。

```
i = 0
while n % 10 > 0 and i < 8:
    s[i] = n % 10
    n = n // 10
    i += 1
    count += 1
```

该题还可以进行修改，如要求用电话传输的数据位数小于8位，且数据加密规则为：将每位传递的数字都加上5，之后用和除以10的余数来代替该数字，最后将第一位和最后一位数字交换。

此时需要定义数组s[8]并定义变量count用于记录输入的数据位数。完整的程序代码如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 数据加密问题

def encry(n, count):
    i = 0
    while n % 10 > 0 and i < 8:
        s[i] = n % 10
        n = n // 10
        i += 1
        count += 1
    i = 0
    for i in range(count+1):
        s[i] += 5
        s[i] %= 10
    # 交换第一位和最后一位数字
    t = s[0]
    s[0] = s[count-1]
    s[count-1]=t

    #输出加密后的数据
    i = count-1
    while i >= 0:
        print("%d" %s[i], end="")
        i -=1
    print()

if __name__ == "__main__":
    # 数组s用来存放生成的8位数
    s = [0] * 8
    n = int(input("请输入将要加密的整数："))
    count = 0
    encry(n, count)
```

程序运行结果如图12.5所示，输入了5位数12345，加密后的数据为07896。

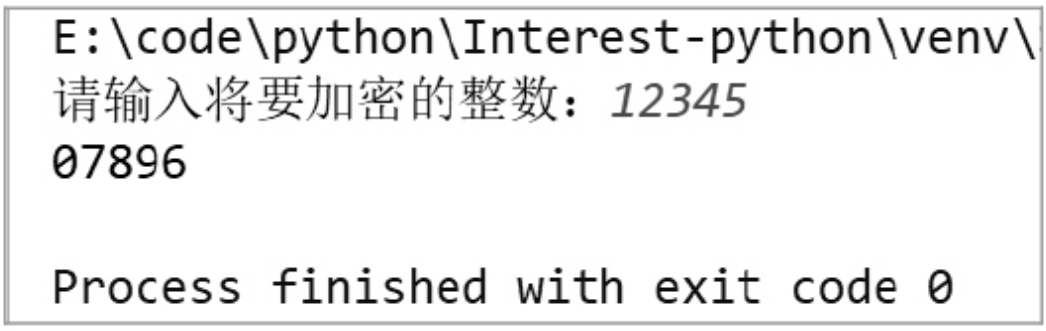


图12.5　运行结果

12.3　三色旗

1．问题描述

假设有一条绳子，上面有红、白、蓝三种颜色的旗子。开始时绳子上旗子的颜色并没有顺序，现在要对旗子进行分类，并按照蓝、白、红的顺序排列。需要注意的是只能在绳子上进行移动，并且一次只能调换两个旗子，则如何移动才能使旗子移动的次数最少。

2．问题分析

由问题描述可知，只在一条绳子上移动，而且一次只能调换两个旗子，因此只要保证在移动旗子时，从绳子的开头开始，遇到蓝色的旗子向前移，遇到白色的旗子留在中间，而遇到红色的旗子则向后移。如果要让移动次数最少，则可以使用三个标记b、w、r来分别标记蓝旗、白旗和红旗。

3．算法设计

整个绳子可以分为三个部分，蓝旗部分、白旗部分和红旗部分，在排序未完成时，还有未处理部分，如图12.6所示。

如果w标记的当前旗子为白色，图12.6中阴影部分为w标记当前所指向的旗子，则w标记增1，表示白旗部分增1。

如果w标记的当前旗子为蓝色，则将b标记与w标记所指向的旗子交换，同时b标记与w标记都增1，表示蓝旗部分与白旗部分都多了一个元素。

如果w标记的当前旗子为红色，则将w标记与r标记所指向的旗子交换，同时r标记减1，即r标记前移，表示未处理的部分减1。

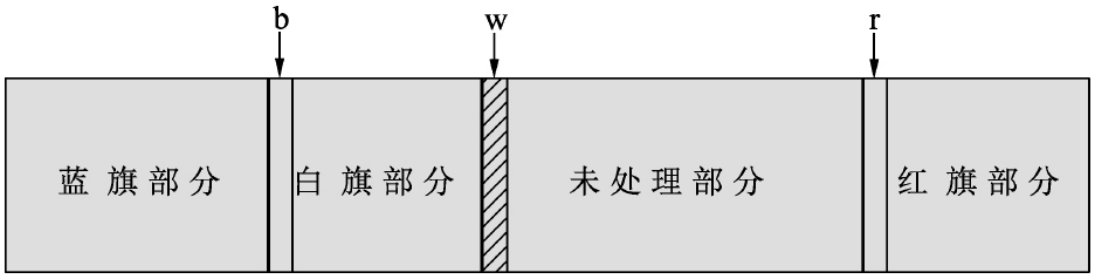


图12.6　三色旗示意图

开始时，r指向绳子中最后一个旗子，之后r标记不断前移，当其位于w标记之前时，即r值小于w值时，全部旗子处理完毕，可以结束比较和移动。

假设绳子上旗子的颜色为如下序列：

red white blue white white blue red blue white red

初始情况下三个标记的位置为：

w b　　r

red white blue white white blue red blue white red　初始情况

则标记的移动情况和交换旗子的过程如下：

w b　　r

red white blue white white blue red blue white red　第一趟：移动 *r* 标记

w b　　r

white white blue white white blue red blue red red　第一趟：交换 *r* 和 *w* 所指旗子

w b　　r

white white blue white white blue red blue red red　第一趟：*r* 标记前移

b　w　r

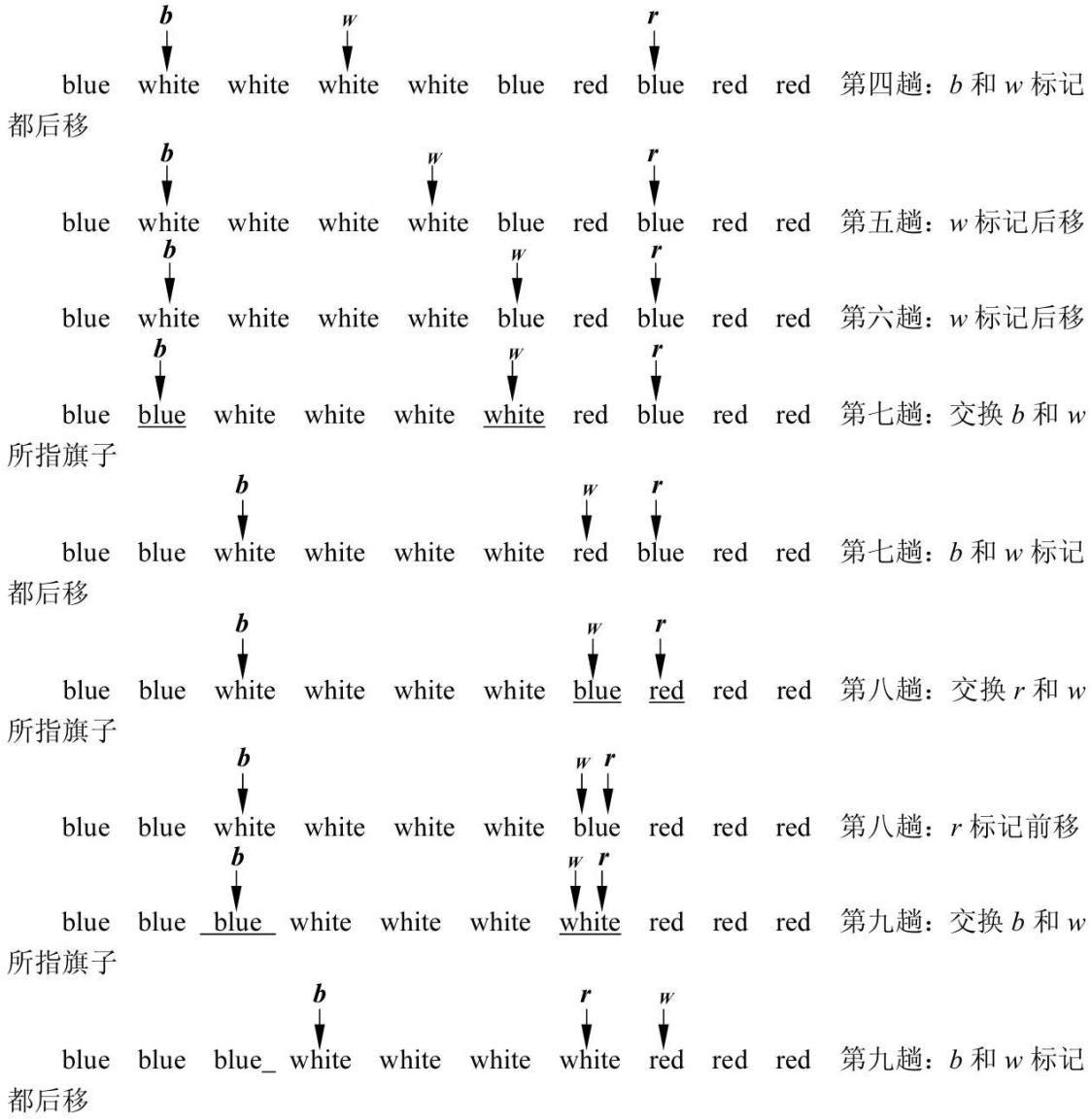
white white blue white white blue red blue red red　第二趟：*w* 标记后移

b　w　r

white white blue white white blue red blue red red　第三趟：*w* 标记后移

b　w　r

blue white white white white blue red blue red red　第四趟：交换 *b* 和 *w* 所指旗子



4．确定程序框架

（1）定义字符数组

定义字符数组表示绳子上的各个旗子的颜色，用大写字母R、W和B分别代表红色、白色和蓝色。

```
color = ['R', 'W', 'B', 'W', 'W', 'B', 'R', 'B', 'W', 'R']          #定义字符数组
```

（2）标记移动和交换旗子

移动时需要通过一个while循环来判断移动过程是否结束，在while循环体中根据旗子的不同颜色进行不同的处理。

移动过程的核心代码如下：

```
# 移动过程
while w <= r:
    # 遇到的是白旗
    if color[w] == WHITE:
        w += 1                                  # 白旗指针自增1
    # 遇到的是蓝旗
    elif color[w] == BLUE:
        SWAP(b, w)                              # 交换蓝旗指针和白旗指针所指向的旗子
        b += 1                                  # 蓝旗指针自增1
        w += 1                                  # 白旗指针自增1
    # 遇到的是红旗
    else:
        # 移动红旗指针使其指向当前最靠前的非红旗位置
        while w < r and color[r] == RED:
            r -= 1                              # 红旗指针自减1
        SWAP(r, w)                              # 交换红旗指针和白旗指针所指向的旗子颜色
        r -= 1                                  # 红旗指针自减1
```

程序流程图如图12.7所示。

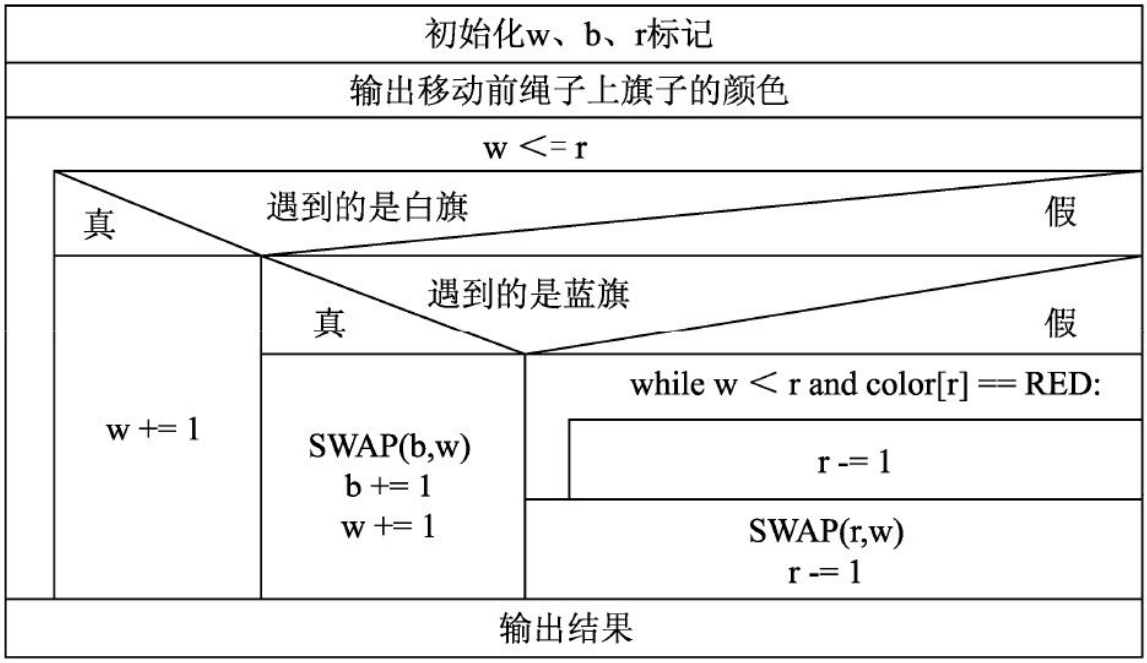


图12.7　程序流程图

5. 完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 三色旗


#交换旗子
def SWAP(x,y):
    temp=color[x]
    color[x]=color[y]
    color[y]=temp


if __name__ == '__main__':
    WHITE = 'W'
    RED = 'R'
    BLUE = 'B'
    color = ['R', 'W', 'B', 'W', 'W', 'B', 'R', 'B', 'W', 'R']
    #定义字符数组
    w, b, r = 0, 0, len(color)-1

    # 打印出移动前绳子上旗子的颜色
    for i in range(len(color)):
        print(color[i], end=' ')
    print()

    # 移动过程
    while w <= r:
        # 遇到的是白旗
        if color[w] == WHITE:
            w += 1                                  # 白旗指针自增1
        # 遇到的是蓝旗
        elif color[w] == BLUE:
            SWAP(b, w)                          # 交换蓝旗指针和白旗指针所指向的旗子
            b += 1                                  # 蓝旗指针自增1
            w += 1                                  # 白旗指针自增1
        # 遇到的是红旗
        else:
            # 移动红旗指针使其指向当前最靠前的非红旗位置
            while w < r and color[r] == RED:
                r -= 1                          # 红旗指针自减1
            SWAP(r, w)                          # 交换红旗指针和白旗指针所指向的旗子颜色
            r -= 1                                  # 红旗指针自减1

    # 打印出移动后绳子上旗子的颜色
    for i in range(len(color)):
        print(color[i], end=' ')
    print()
```

6. 运行结果

在PyCharm下运行程序，结果如图12.8所示。

由运行结果可见，程序执行后，绳子上三种颜色的旗子已经按照蓝、白、红的顺序排列了。

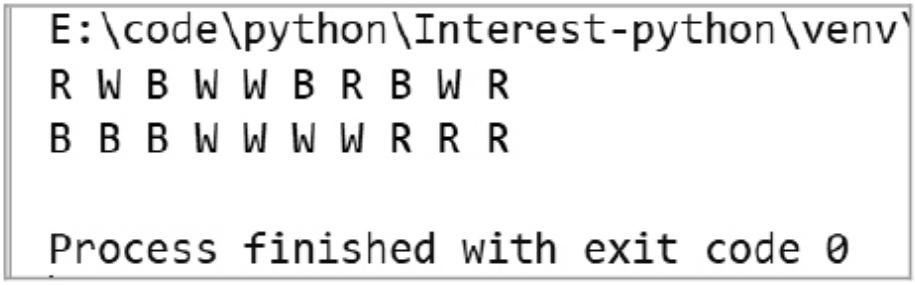

E:\code\python\Interest-python\venv
R W B W W B R B W R
B B B W W W W R R R
Process finished with exit code 0

图12.8　运行结果

12.4　双色球

1. 问题描述

编写程序模拟福利彩票的双色球开奖过程，由程序产生出6个红色球和1个蓝色球。

要求：

1）每期开出的红色球号码不能重复，但蓝色球可以是红色球中的一个。

2）红色球的范围是1～33，蓝色球的范围是1～16。

3）输出格式为“红色球：x x x x x x　蓝色球：x”。

2. 问题分析

由问题描述可知，该问题是编程来模拟福利彩票中双色球开奖过程，因此需要随机生成6个红色球号码和1个蓝色球号码，显然需要使用Python语言中的random模块来生成随机数。

由题目要求可知“每期开出的红色球号码不能重复”，而使用随机函数并不能保证每次产生的随机数都不相同，因此在程序设计时需要判断每次新生成的红色球号码是否和已生成的红色球号码相同，如果有重复，则需要重新生成新的红色球号码。

3. 算法设计

随机生成6个不同红色球号码的功能可使用循环结构来实现。我们使用数组来保存生成的6个红色球号码。在循环体中需要判断每次新生成的红色球号码是否与已生成的红色球号码重复。

由于蓝色球号码只有1个，而且可以与红色球的号码重复，因此可以直接使用随机函数来生成蓝色球号码，并保存在变量中。

4. 确定程序框架

（1）产生随机数

产生1～33范围内的随机整数，代码如下：

```
tmp = random.randint(1, 33)
```

（2）随机产生红色球号码

定义red数组来保存产生的红色球号码。由题意可知，需要随机产生6个红色球号码，因此可以使用red数组中下标为0～5的6个元素来保存红色球号码。

随机产生红色球号码的过程使用while循环结构，循环变量为i，i初值为0，i<6，在循环体中随机生成不同的红色球号码。需要注意的是，因为红色球号码是随机生成的，因此有可能两次while循环中产生的红色球号码恰好相同，这就要求在循环体中必须有相应的代码来判断每次新生成的红色球号码是否与已生成的红色球号码不相同。如果不相同，则在red数组的相应位置保存该新生成的红色球号码，否则应该重新生成新的红色球号码。代码如下：

```
red = [1] * 6             # 定义red数组，保存随机生成的6个红色球号码，号码范围为1～33
i = 0
# 随机生成6个红色球号码
while i < 6:
    tmp = random.randint(1, 33)
    j = 0
    while j < i:
        # 判断已生成的红色球号码是否与当前while循环中产生的随机红色球号码相同
        # 如果相同，则重新生成新的红色球号码，否则在red[i]中保存新生成的红色球号码
        if red[j] == tmp:
            break
        j += 1
    if j == i:
        red[i] = tmp          # 将新生成的红色球号码保存在red数组中
        i += 1
```

程序流程图如图12.9所示。

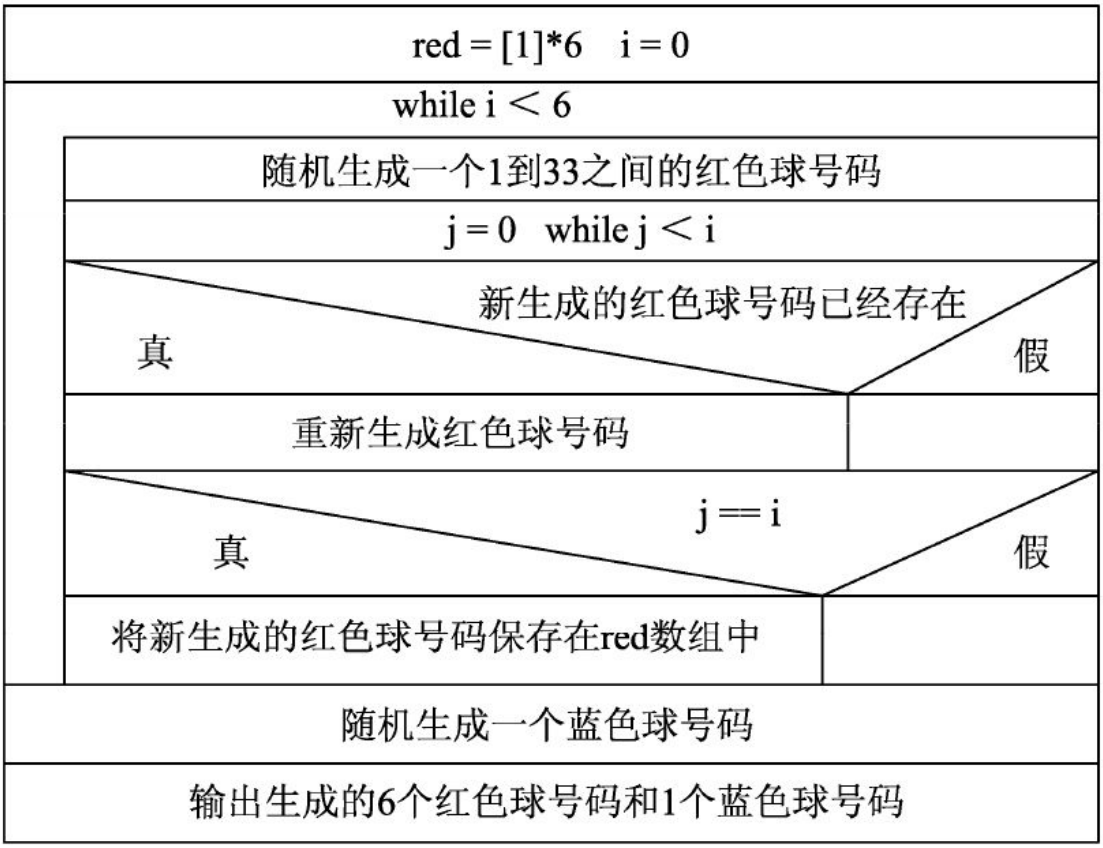


图12.9　程序流程图

5．完整的程序

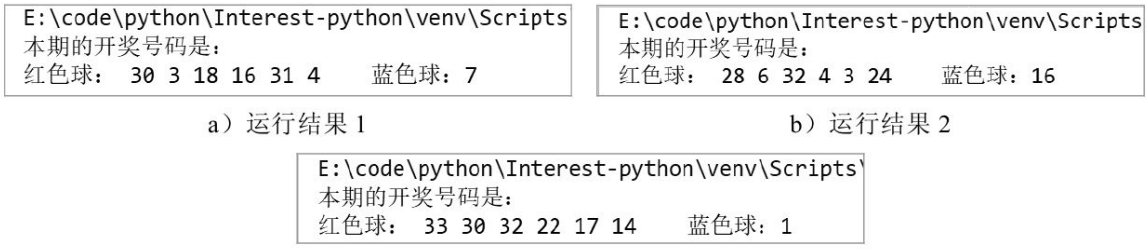
根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 双色球　红色球范围1～33，6个；　蓝色球范围1～16，1个

import random

if __name__ == "__main__":
    red = [1] * 6   # 定义red数组，保存随机生成的6个红色球号码，号码范围为1～33
    i = 0
    # 随机生成6个红色球号码
    while i < 6:
        tmp = random.randint(1, 33)
        j = 0
        while j < i:
            # 判断已生成的红色球号码是否与当前while循环中产生的随机红色球号码相同
            # 如果相同，则重新生成新的红色球号码，否则在red[i]中保存新生成的红色球号码
            if red[j] == tmp:
                break
            j += 1
        if j == i:
            red[i] = tmp                        # 将新生成的红色球号码保存在red数组中
            i += 1

    blue = random.randint(1, 16)     # 随机生成蓝色球号码
```

```
print("本期的开奖号码是：")
print("红色球：", end=" ")
for i in range(6):
    print("%d" % red[i], end=" ")
print("    蓝色球：%d" % blue)
```

6. 运行结果

在PyCharm下运行程序，运行三次，每次都生成了不同的红色球和蓝色球序列，结果如图12.10所示。



a）运行结果 1　　b）运行结果 2

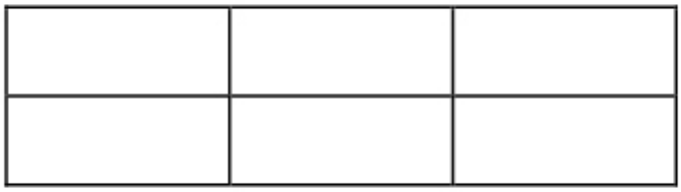
c）运行结果 3

图12.10　运行结果

12.5　填表格

1．问题描述

将1、2、3、4、5和6填入下表中，要求每一列右边的数字比左边的数字大且每一行下面的数字比上面的数字大。编程求出共有几种填写方法？



2．问题分析

根据题目要求可知，数字1必然位于表中第一行第一列的单元格中，而数字6则必然位于表中第二行第三列的单元格中，其他几个数字则按照题目要求使用试探法来分别找到合适的位置。

3．算法分析

根据前面的分析，在实现时可以定义一个一维数组a[6]。数组元素a[0]～a[2]位于表格中的第一行，数组元素a[3]～a[5]位于表格中的第二行。具体如表12.1所示。

表12.1　数组元素分布

a[0]	a[1]	a[2]
a[3]	a[4]	a[5]

显然，a[0]=1，a[5]=6。又根据题意可知：

1）a[1]应该大于a[0]，则a[1]的取值从a[0]+1开始试探，最大不会超过5。

2）a[2]应该大于a[1]，则a[2]的取值从a[1]+1开始试探，最大不会超过5。

3）a[3]位于表格中的第二行第一列，则它应该大于a[0]，因此a[3]的取值从a[0]+1开始试探，最大不会超过5。

4）a[4]位于表格中的第二行第二列，则它应该大于左侧的a[3]和第一行中的a[1]，因此当a[1]>a[3]时，a[4]初值为a[1]+1，即从a[1]+1开始试探，最大不会超过5；当a[1]<a[3]时，a[4]初值为a[3]+1，即从a[3]+1开始试探，最大不会超过5。

4. 确定程序框架

（1）确定循环结构

在main()函数中构建一个四重循环，在该四重循环中对a[1]~a[4]的取值进行试探。每重循环的循环变量初始值以及循环条件在算法分析中已经讨论过。

循环结构如下：

```
a[1] = a[0] + 1
while a[1] <= 5:                        # a[1]必须大于a[0]
    a[2] = a[1] + 1
    while a[2] <= 5:                        # a[2]必须大于a[1]
        a[3] = a[0] + 1
        while a[3] <= 5:                    # 第二行的a[3]必须大于a[0]
                        # 第二行的a[4]必须大于左侧的a[3]和上边的a[1]
            if a[1] > a[3]:
                a[4] = a[1] + 1
            else:
                a[4] = a[3] + 1
            while a[4] <= 5:
                # 判断是否满足题意，如果满足则打印结果
```

上面的代码中加粗的部分使用了条件表达式来保证a[4]必须同时大于左侧的a[3]和上面一行中的a[1]。

（2）判断取值

定义函数judge()来判断a[1]～a[4]的取值是否符合题意，代码如下：

```
# 判断a[1]～a[4]的取值
def judge(b):
    for i in range(1, 4):
        l = i + 1
        while l < 5:
            if (b[l] == b[i]):
                # 判断a[1]~ a[4]的取值是否有重复，若有重复则返回False
                return False
            l += 1

    return True                                    # 若a[1]~ a[4]的取值各不相同，则返回
True
```

程序的流程图如图12.11所示。

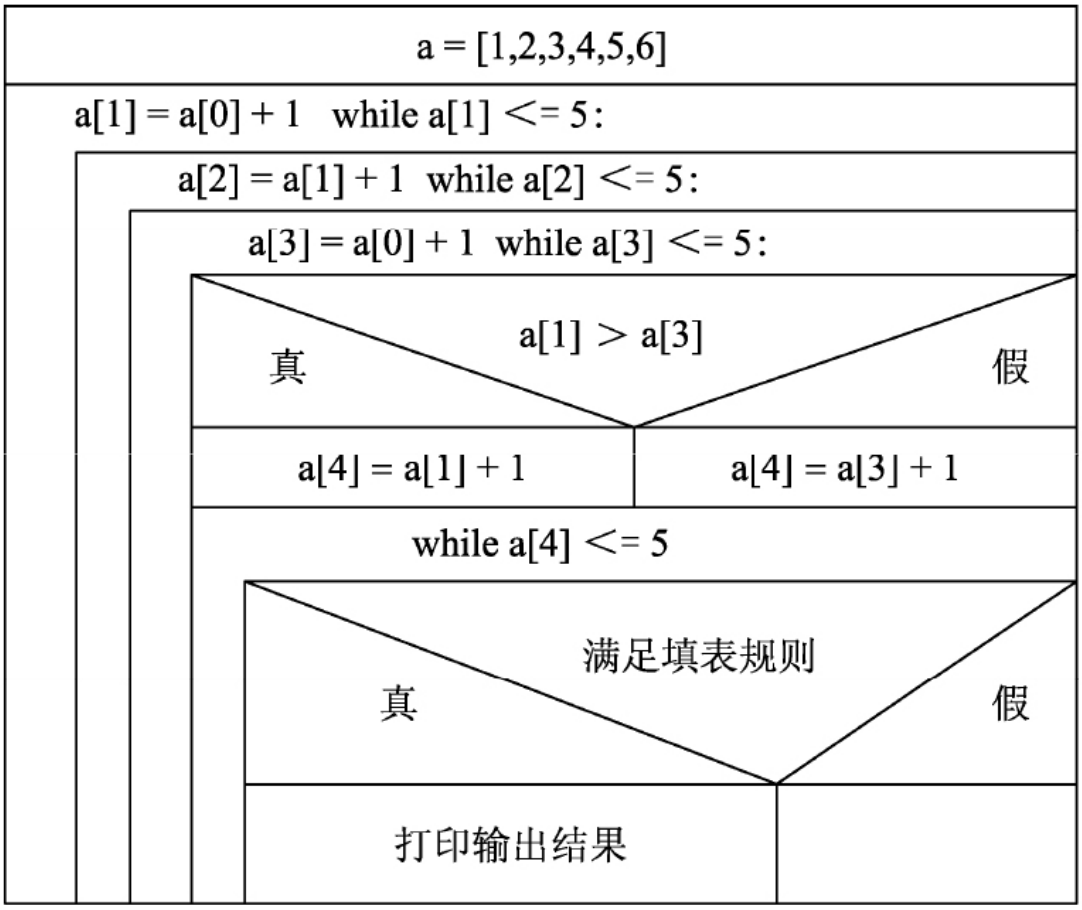


图12.11　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
# @author : liuhefei
# @desc: 填2行3列的表格


# 判断a[1]～a[4]的取值
def judge(b):
    for i in range(1, 4):
        l = i + 1
        while l < 5:
            if (b[l] == b[i]):
                # 判断a[1]～a[4]的取值是否有重复，若有重复则返回False
                return False
            l += 1

    return True  # 若a[1]～a[4]的取值各不相同，则返回True

if __name__ == '__main__':
    count = 1                                          # 计数器
    a = [1, 2, 3, 4, 5, 6]                    # 初始化列表
    print("满足条件的结果为:")
    a[1] = a[0] + 1
    while a[1] <= 5:                                          # a[1]必须大于a[0]
        a[2] = a[1] + 1
        while a[2] <= 5:                                      # a[2]必须大于a[1]
            a[3] = a[0] + 1
            while a[3] <= 5:                          # 第二行的a[3]必须大于a[0]
                                  # 第二行的a[4]必须大于左侧的a[3]和上边的a[1]
                if a[1] > a[3]:
                    a[4] = a[1] + 1
                else:
                    a[4] = a[3] + 1
                while a[4] <= 5:
                    if judge(a):  # 如果满足题意，则打印结果
                        print("结果%d:" % count)
                        count += 1
                        print(a[:3])
                        print(a[3:])
                        print()
                    a[4] += 1
                a[3] += 1
            a[2] += 1
        a[1] += 1
```

---

6. 运行结果

在PyCharm下运行程序，结果如图12.12所示。由运行结果可知，满足条件的结果共有5组，即有5种符合题意的填表方法。

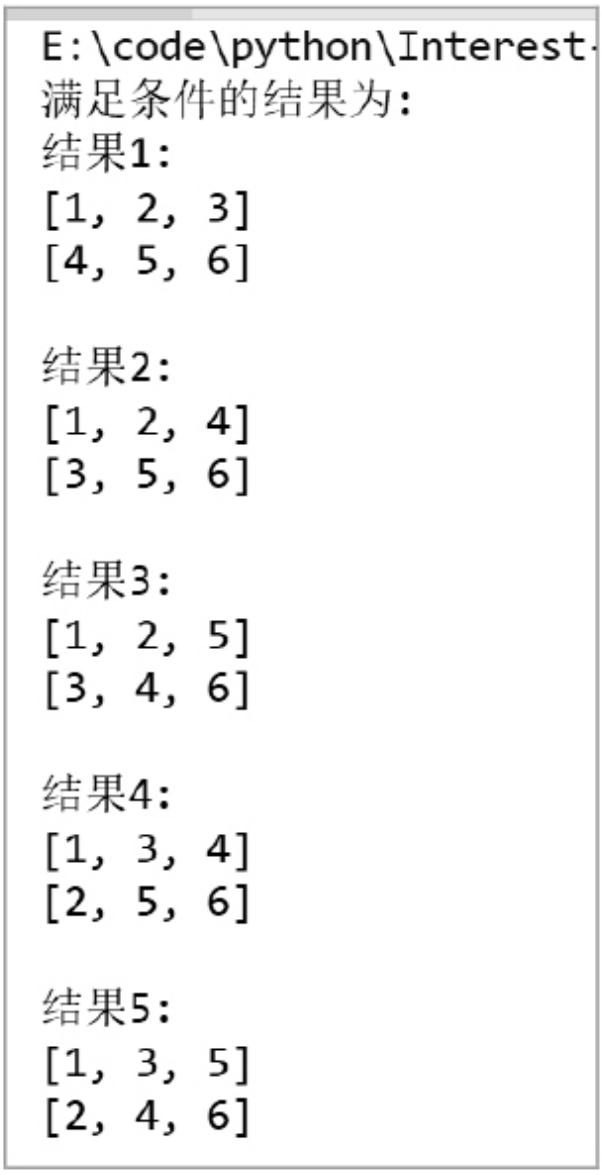

E:\code\python\Interest
满足条件的结果为:
结果1:
[1, 2, 3]
[4, 5, 6]
结果2:
[1, 2, 4]
[3, 5, 6]
结果3:
[1, 2, 5]
[3, 4, 6]
结果4:
[1, 3, 4]
[2, 5, 6]
结果5:
[1, 3, 5]
[2, 4, 6]

图12.12　运行结果

12.6　求出符合要求的素数

1. 问题描述

编写程序实现将大于某个整数n且紧靠n的k个素数存入某个数组中，同时实现从infile.txt文件中读取10对n和k的值，分别求出符合要求的素数，并将结果保存到outfile.txt文件中。

2. 问题分析

解决该问题首先要能够判断出某个数是否为素数，同时还应该熟练掌握Python语言中读写文件的相关知识。判断某个数是否为素数的方法在第5章中已经详细介绍过了，这里对Python语言的文件概念进行简单介绍。

文件指的是存储在外部介质中的数据的集合。运行程序时，通常的做法是将数据保存在变量中，但是当程序结束并关闭时，数据也会随着丢失。为了保存运行程序时产生的数据，最好的做法是通过文件的形式来保存，当程序运行结束后，数据仍然在文件中，不会丢失。

3. 算法设计

该问题可根据题意直接编程，程序中应该能够判断某个数是否为素数，该功能可通过定义函数来实现，同时也可以通过定义相应功能的函数来实现对文件的操作。

4. 确定程序框架

（1）判断是否为素数

定义函数isPrime()，形式参数为n，当函数返回值为0时，表示n不是素数；当函数返回值为1时，则表示n为素数。

```
# 判断素数
def isPrime(n):
    for i in range(2, n):
        if n % i == 0:
            return False                        # n不是素数
    return True                                 # n是素数
```

（2）求出紧靠n的k个素数，并存放到数组中

定义函数num()，其形参为n、k和数组array。

```
# 求出紧靠n的k个素数
def num(n, k, array):
    i, n = 0, n + 1
    while k > 0:
        if isPrime(n):                          # 调用函数isPrime()判断n是否为
素数
            array[i] = n                        # 若n是素数，则将n存入数组array中
            i += 1
            k -= 1
        n += 1
```

（3）实现对文件的操作

定义filedata()函数，实现对文件的操作。在该函数中，实现了对infile.txt文件的读操作，从infile.txt文件中读取10对n和k的值，再调用num()函数分别求出符合要求的素数，最后将结果保存到outfile.txt文件中。因为outfile.txt文件不存在，因此先生成outfile.txt文件，再对其进行写操作。

filedata()函数代码如下：

```
# 文件操作
def filedata():
    array = [0] * 1000
    rf = open('infile.txt', 'r')                # 读infile.txt文件
    wf = open('outfile.txt', 'w')               # 写outfile.txt文件
    # 从infile.txt文件中读取10对(n,k)值
    for i in range(10):
        [n, k] = rf.readline().split()
        n = int(n)
        k = int(k)
        num(n, k, array)                        # 调用num()函数
        for n in range(k):
            print('%d ' % array[n], file=wf, end='')            # 写文件
        print(file=wf)

    rf.close()              # 关闭infile.txt文件
    wf.close()              # 关闭outfile.txt文件
```

程序的流程图如图12.13所示。

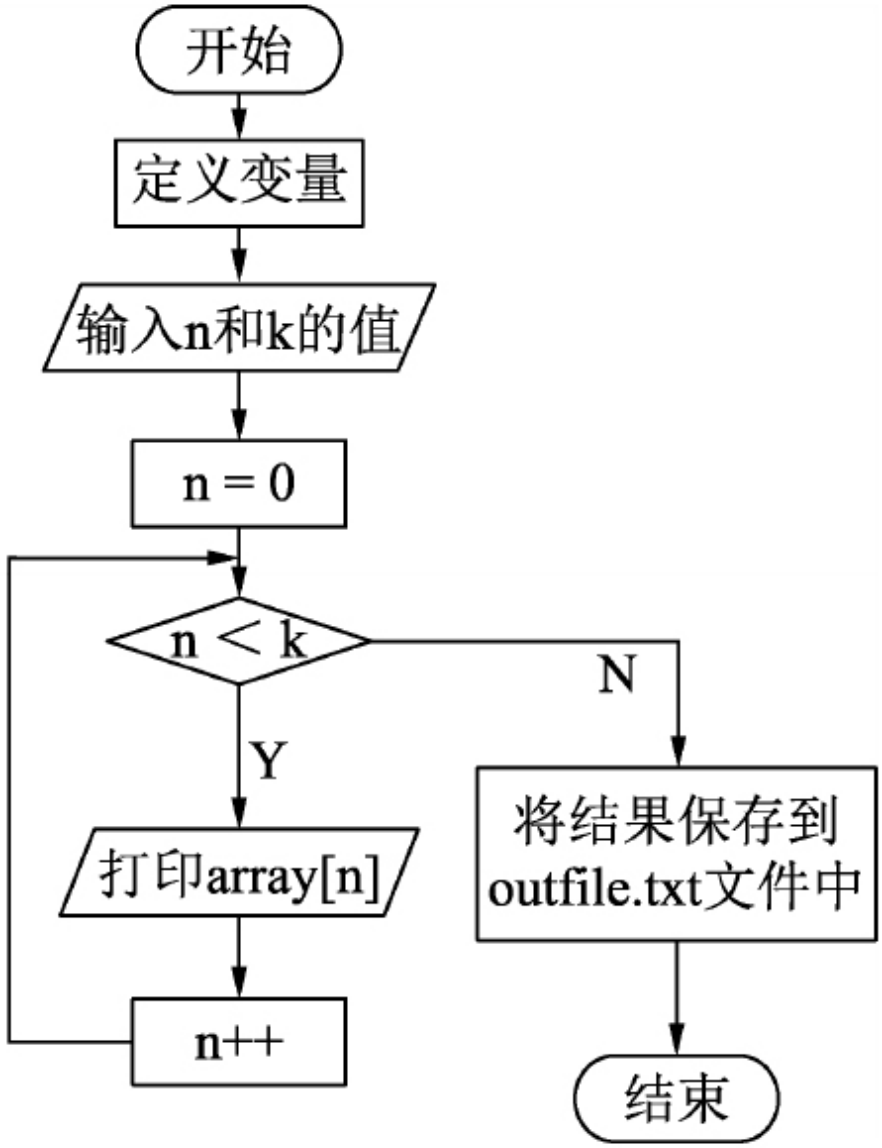


图12.13　程序流程图

5. 完整的程序

根据上面的分析，编写完整的程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 求出符合要求的素数

# 判断素数
def isPrime(n):
    for i in range(2, n):
```

```
        if n % i == 0:
            return False            # n不是素数
    return True                     # n是素数

# 求出紧靠n的k个素数
def num(n, k, array):
    i, n = 0, n + 1
    while k > 0:
        if isPrime(n):              # 调用函数isPrime()判断n是否为素数
            array[i] = n    # 若n是素数，则将n存入数组array中
            i += 1
            k -= 1
        n += 1


# 文件操作
def filedata():
    array = [0] * 1000
    rf = open('infile.txt', 'r')            # 读infile.txt文件
    wf = open('outfile.txt', 'w')           # 写outfile.txt文件
    # 从infile.txt文件中读取10对(n,k)值
    for i in range(10):
        [n, k] = rf.readline().split()
        n = int(n)
        k = int(k)
        num(n, k, array)                                # 调用num()函数
        for n in range(k):
            print('%d ' % array[n], file=wf, end='')        # 写文件
        print(file=wf)

    rf.close()                                                  # 关闭
infile.txt文件
    wf.close()                                                  # 关闭
outfile.txt文件


if __name__ == '__main__':
    array = [0] * 1000
    print('输入整数n和k：')
    n, k = map(int, input().split())
    num(n, k, array)                                        # 调用num()函数
    for n in range(k):
        print(array[n], end=' ')            # 打印出array数组中的每个值
    print('\n')
    filedata()                                                  # 调用
filedata()函数
```

在运行程序之前，一定要先创建infile.txt文件，否则将会报错。

6. 运行结果

在PyCharm下运行程序，当输入整数n=17，k=5时，打印出紧靠17的5个素数：19、23、29、31和37，如图12.14所示，

同时，程序还从infile.txt文件中读取了10对n和k值，infile.txt文件的图标如图12.15所示，其保存的10对n和k的值如图12.16所示。对这10对n和k的值分别求出符合要求的素数，并存放在

outfile.txt文件中。outfile.txt文件由程序来生成，其图标如图12.17所示，文件内容如图12.18所示。

```
E:\code\python\Interest-python\venv\
输入整数n和k：
17 5
19 23 29 31 37
```

图12.14　运行结果

图12.15　infile.txt文件图标

```
17 5
20 3
12 7
35 5
10 2
46 3
56 8
19 4
28 6
69 5
```

图12.16　infile.txt文件的内容

图12.17　outfile.txt文件图标

```
19 23 29 31 37
23 29 31
13 17 19 23 29 31 37
37 41 43 47 53
11 13
47 53 59
59 61 67 71 73 79 83 89
23 29 31 37
29 31 37 41 43 47
71 73 79 83 89
```

图12.18　outfile.txt文件的内容

7. 补充知识点

该问题涉及Python语言中文件的内容。下面对Python语言的文件操作进行简单介绍。

（1）文件的打开

在Python中，使用open()函数来打开文件。该函数的定义如下：

```
open(file_name [,access_mode][,buffering])
```

参数说明：

- file_name：将要打开的文件名，是一个字符串。

• access_mode：指打开文件的模式，对应有只读、只写、追加等。access_mode变量值不是必须要有的，默认的文件访问模式是只读（r）。

• buffering：文件缓冲区，如果buffering的值为0，就不会有寄存；如果buffering的值为1，访问文件时就会有寄存行；如果将buffering的值设为大于1的整数，表示这就是寄存区的缓冲大小；如果buffering的值为负数，表示寄存区的缓冲大小就是系统默认的值。

open()函数返回一个文件（File）对象。File对象代表计算机中的一个文件。

（2）文件的打开模式

Pyhon语言中的文件打开模式如表12.2所示。

表12.2　Python语言中的文件打开模式

模　　式	描　　述
a	打开一个文件用于追加操作。如果该文件已经存在，文件指针就会指向文件的末尾。换句话说，新内容会追加到原有内容的后面。如果文件不存在，则会先创建新文件然后进行写入操作

（续）

模　式	描　述
ab	以二进制格式打开一个文件用于追加操作。如果该文件已经存在，文件指针就会指向文件的末尾。换句话说，新内容会追加到原有内容的后面。如果文件不存在，则会先创建新文件然后进行写入操作
a+	打开一个文件用于读写操作。如果该文件已经存在，文件指针就会指向文件的结尾。文件打开时是追加模式。如果该文件不存在，则会创建新文件用于读写操作
ab+	以二进制格式打开一个文件用于追加操作。如果该文件已经存在，文件指针就会放到文件末尾；如果文件不存在，则会创建新文件用于读写操作
r	以只读的方式打开文件。文件指针将会指向文件的开头，这是文件操作的默认方式
rb	以二进制格式打开一个文件用于只读操作。文件指针将会指向文件的开头
r+	打开一个文件用于读写操作。文件指针将会指向文件的开头
rb+	以二进制格式打开一个文件用于读写操作。文件指针将会指向文件的开头
w	打开一个文件用于写入操作。如果该文件已经存在，就将其原有内容覆盖，写入新的内容；如果该文件不存在，则创建新文件，然后写入新内容
wb	以二进制格式打开一个文件用于写入操作。如果该文件已经存在，就将其原有内容覆盖，写入新内容；如果该文件不存在，则创建新文件，然后写入新内容
w+	打开一个文件用于读写操作。如果该文件已经存在，就将其原有内容覆盖；如果该文件不存在，则创建新文件，然后进行读写操作
wb+	以二进制格式打开一个文件用于读写。如果该文件已经存在，就将其原有内容覆盖；如果该文件不存在，则创建新文件，然后进行读写操作

在使用open()函数时，指定文件操作模式为读模式与什么模式都不指定的效果一样。使用读模式只能读取文件内容；使用写模式可以向文件写入内容；+参数可以用到任何其他模式中，表示同时允许读写操作；b参数表示可以用来读取一个二进制文件。

（3）文件的读写

在Python的文件操作中，我们使用read()函数来读取文件内容，使用write()函数来向文件中写入内容。

open()函数返回的是一个File对象，有了File对象，就能进行后续的读写操作了。我们使用File对象的read()函数将文件的内容读取为一个字符串值。read()函数从一个打开的文件中读取字符串。Python中的字符串不仅仅是简单的字符串（字母、数字、汉字等），也可以是二进制数据。read()函数的定义如下：

```
fileObject.read([count]);
```

说明： fileObject为open()函数返回的File对象，count参数是从已经打开的文件中读取的字节数，该函数从文件的开头处开始读入。如果没有count参数，该函数就会尝试尽可能地多读取内容，有可能一直读取到文件末尾。

在Python中，使用write()函数向一个文件中写入内容数据。write()函数可以将任何字符串写入一个打开的文件，它不会在字符串的结尾添加换行符（‘\n’）。write()函数的定义如下：

```
fileObject.write(string);
```

说明： fileObject是open()函数返回的File对象，string参数是将要写入文件的内容。write()函数返回写入文件的字符串的长度。

（4）文件的关闭

对文件操作完成之后，需要调用文件的close()方法来关闭文件。一般情况下，一个文件对象在退出程序后会自动关闭，但是为了安全起见，还是要显式地调用close()方法来关闭文件，以避免出现不必要的异常。close()函数的定义如下：

```
fileObject.close();
```

说明： fileObject是open()函数返回的File对象。

当我们对一个文件进行修改时，在操作完成后，一定要记得关闭文件，因为写入的数据可能被缓存，如果程序或系统因为某些原因而崩溃，被缓存部分的数据就不会写入文件了。

（5）文件操作的其他常用函数

下面介绍几个文件操作中常用的其他函数。

• fileObject.readline()：该函数用于读入文件的一行到字符串中。

• fileObject.readlines()：该函数用于将整个文件按行读入到列表中。

• fileObject.writeLines()：该函数用于向文件中写入一个列表。

实例代码：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 文件操作


if __name__ == "__main__":
    # 以只读方式打开test.txt文件
    file1 = open("test.txt", 'r', encoding='utf-8')
    print("test.txt的文件内容：", file1.read())

    # 以写入方式打开test1.txt文件
    file2 = open("test1.txt", 'w+', encoding='utf-8')
    file2.write("你好")  # 将test.txt文件的内容写入test1.txt中
    file2 = open("test1.txt", 'r', encoding='utf-8')
    print("test1.txt的文件内容为：", file2.read())

    list_str = ["天国虽美","没有了你","万杯觥筹","只不过是提醒寂寞罢了"]
    # 以写入方式打开test2.txt文件
    file3 = open("test2.txt", 'w+', encoding='utf-8')
    file3.writelines(list_str)
    file3 = open("test2.txt", 'r', encoding='utf-8')
    print("test2.txt的文件内容为：", file3.readlines())

    # 关闭文件
    file1.close()
    file2.close()
    file3.close()
```

在PyCharm下运行程序，结果如图12.19所示：

```
E:\code\python\Interest-python\venv\Scripts\python.exe E:/code/python/
test.txt的文件内容：  虽然辛苦，我还是会选择那种滚烫的人生！
我所理解的生活就是和自己喜欢的一切在一起！
当你想要某种东西时，整个宇宙会合力祝你实现愿望！
如果命运是一条孤独的河流，谁会是你灵魂的摆渡人？
对每个人而言，真正的职责只有一个：找到自我！
你是爱，是暖，是希望，你是人间的四月天！
我们终此一生，就是要摆脱他人的期待，找到真正的自己！
test1.txt的文件内容为：  你好
test2.txt的文件内容为：  ['天国虽美没有了你万杯觥筹只不过是提醒寂寞罢了']
```

图12.19　运行结果

关于Python文件相关的知识点就介绍这么多，感兴趣的读者可以参考相关资料深入进行学习。

12.7　统计学生成绩

1. 问题描述

有5个学生，每个学生有三门课程的成绩需要统计。要求从键盘输入学生的学号、姓名以及三门课程的成绩，计算出平均成绩，并将原有的数据和计算出的平均成绩存放在磁盘文件stud中。

2. 问题分析

该问题是统计学生信息，要统计的信息包括学生的学号、姓名、三门课程的成绩，以及计算出的平均成绩，显然可以使用Python语言中的列表来存放学生的信息。

题目中还要求将学生信息保存到文件中，这就需要打开文件、向文件中写入数据以及关闭文件等一系列操作。

3. 算法设计

先定义一个学生列表，用来存放5个学生的信息。再通过for循环读入5个学生的信息，最后通过文件操作将5个学生的信息保存到文件中。

4. 确定程序框架

（1）定义学生列表

定义student列表用于保存学生信息，该列表如下：

```
stu = [['', '', 0, 0] for i in range(5)]
```

（2）将学生信息保存到文件stud中

具体代码如下：

```
with open('stud', 'w', encoding='utf-8') as fp:          # 打开文件
    print('Sno,Sname,Score,arv', file=fp)
    for i in range(5):
        fp.write('{0},{1},{2},{3}\n'.format(stu[i][0], stu[i][1], stu[i][2],
stu[i][3]))
# 将学生信息写入文件
fp.close()
# 关闭文件
```

程序的流程图如图12.20所示。

图12.20　程序流程图

5．完整的程序

根据上面的分析，编写程序如下：

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @author : liuhefei
# @desc: 统计学生成绩

if __name__ == '__main__':
    stu = [['', '', 0, 0] for i in range(5)]

    # 输入5个学生信息
    for i in range(5):
        print('\n请输入第%d个学生的信息:' % (i + 1))
        stu[i][0] = input('stuNo:')
```

```
        stu[i][1] = input('name:')
        sum = 0
        # 求出平均成绩
        for j in range(3):
            stu[i][2] = int(input('score %d:' % (j + 1)))
            sum += stu[i][2]
            stu[i][3] = sum // 3.0

    with open('stud', 'w', encoding='utf-8') as fp:          # 打开文件
        print('Sno,Sname,Score,arv', file=fp)
        for i in range(5):
            fp.write('{0},{1},{2},{3}\n'.format(stu[i][0], stu[i][1], stu[i]
[2], stu[i][3]))
# 将学生信息写入文件
    fp.close()
# 关闭文件
```

6. 运行结果

在PyCharm下运行程序，输入的数据如下。

```
请输入第1个学生的信息:
stuNo:12
name:张三
score 1:89
score 2:75
score 3:72

请输入第2个学生的信息:
stuNo:15
name:李四
score 1:76
score 2:83
score 3:85

请输入第3个学生的信息:
stuNo:29
name:王五
score 1:68
score 2:71
score 3:78

请输入第4个学生的信息:
stuNo:35
name:李明
score 1:86
score 2:88
score 3:91

请输入第5个学生的信息:
stuNo:36
name:王立
score 1:72
score 2:69
score 3:80
```

生成的stud文件如图12.21所示。

文件内容如图12.22所示。

图12.21　stud文件

12.7.py　stud

```
1  Sno,Sname,Score,arv
2  12,张三,72,78.0
3  15,李四,85,81.0
4  29,王五,78,72.0
5  35,李明,91,88.0
6  36,王立,80,73.0
7
```

图12.22　运行结果