# 第 3 章

# 数 独 游 戏

数独游戏起源于 18 世纪初瑞士数学家欧拉等人研究的拉丁方阵，通过数独游戏可以锻炼人的逻辑思维能力，同时能提高人的专注力、观察力和反应力。自数独游戏起源到现今，一直十分受欢迎。通过对数独游戏的编程实现，将帮助读者掌握 Python 中 list 变量、for 循环、自定义函数等知识的用法，同时本章将介绍在游戏编程中常用的深度优先递归算法。

## 3.1 数独游戏规则

数独是在一个具有 81 个方格的正方形棋盘内按照一定规则填入 1～9 的数字，从而使 81 个方格都填满数字的一种游戏，其游戏棋盘如图 3-1 所示。

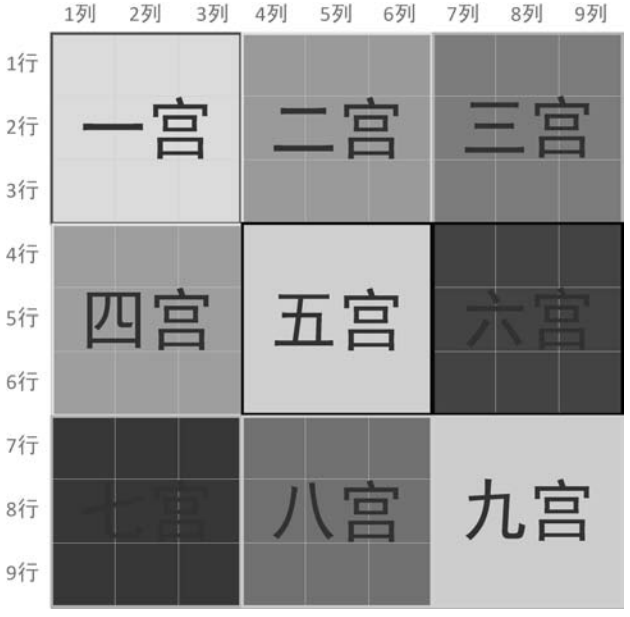


图 3-1 数独游戏棋盘

从图 3-1 可知，棋盘共有 9 行，每一行具有 9 列，同时在棋盘内分成了 9 个宫格，按照从左到右，从上到下分别是一宫到九宫。游戏开始时，棋牌内将会给一些初始数字，按照难度的不同，给出的数字的个数也不同，玩家需要做的就是使用 1～9 的数字填满空白格子，并同时满足以下条件。

（1）行数字不重复：每一行的数字都不同，并且必须包含 1～9。

（2）列数字不重复：每一列的数字都不同，并且必须包含 1～9。

（3）宫格内数字不同：每一宫格内数字都不同，并且必须包含 1～9。

图 3-2 为一个数独游戏的初始开局界面，从图中可知，81 个方格已经填了 32 个数字，其余 49 个空格子需要按照上述规则进行填满。

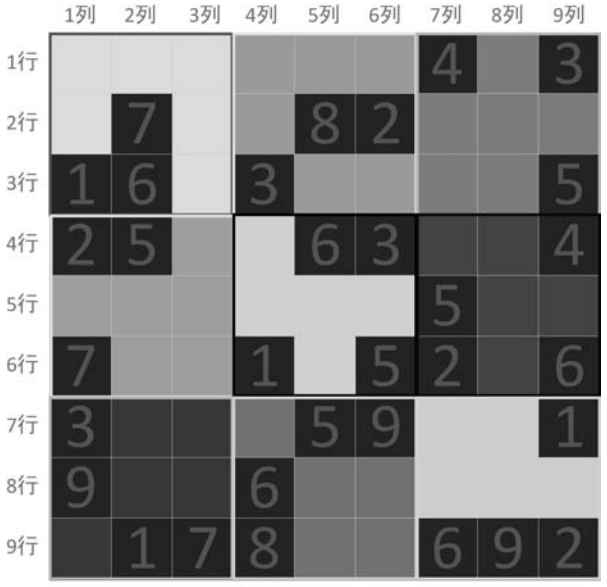


图 3-2 数独游戏的初始开局界面

对图 3-2 的数独初始开局界面进行解答后，可以得到如图 3-3 所示的一个解答。需要说明的是，对于数独的某一个初始开局界面，存在着的解答有可能不止一个正确答案，但也有可能无解，这都是在游戏设计时需要关注的点。

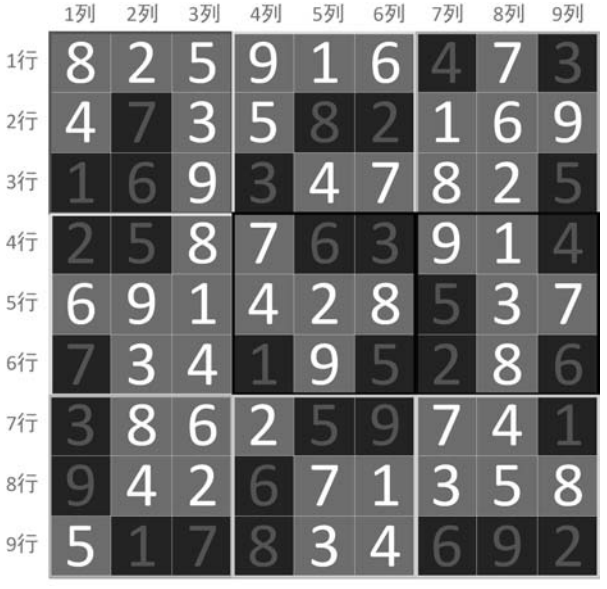


图 3-3 数独游戏的一个解答

## 3.2　数独游戏运行示例

3min

在 3.1 节对数独游戏的游戏规则进行了说明，从游戏规则可知，设计一个数独游戏应包含以下内容。

（1）创建数独的初始开局界面：游戏需要根据玩家的输入提供不同难度的游戏初始开局界面。

（2）创建的数独初始开局界面的解答：创建好数独初始开局界面后，需要提供给玩家相应的开局界面解答。

（3）玩家输入的数独开局界面的解答：玩家输入某一个初始开局界面后，游戏具备给出正确答案的能力。

按照上述内容要求，进行游戏编码后，游戏运行如图 3-4 所示。

```
**********************    欢迎来到数独游戏    **********************

1：创建简单难度数独游戏。
2：创建困难难度数独游戏。
3：输入数独并得到答案(每个数字用空格隔开，空白数字用0代替)。
4：退出!

请输入对应的数字：1
简单难度的数独为：
[0, 8, 0, 0, 1, 3, 5, 0, 0]
[6, 0, 0, 8, 9, 2, 0, 0, 1]
[0, 0, 0, 0, 4, 0, 8, 2, 0]
[0, 0, 5, 0, 8, 0, 0, 0, 7]
[4, 0, 0, 0, 0, 0, 2, 0, 0]
[0, 0, 0, 3, 0, 0, 6, 9, 5]
[8, 9, 6, 1, 0, 0, 0, 5, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 6]
[0, 4, 0, 0, 6, 0, 1, 7, 0]
```

图 3-4　数独游戏运行示例

游戏运行后，玩家有 4 种方式与计算机进行交互，输入 1 将创建简单难度的数独游戏；输入 2 将创建困难难度的数独游戏；输入 3 后则计算机会解答玩家输入的数独题目；输入 4 则退出游戏。

按照上述游戏逻辑，画出的数独游戏流程图如图 3-5 所示。

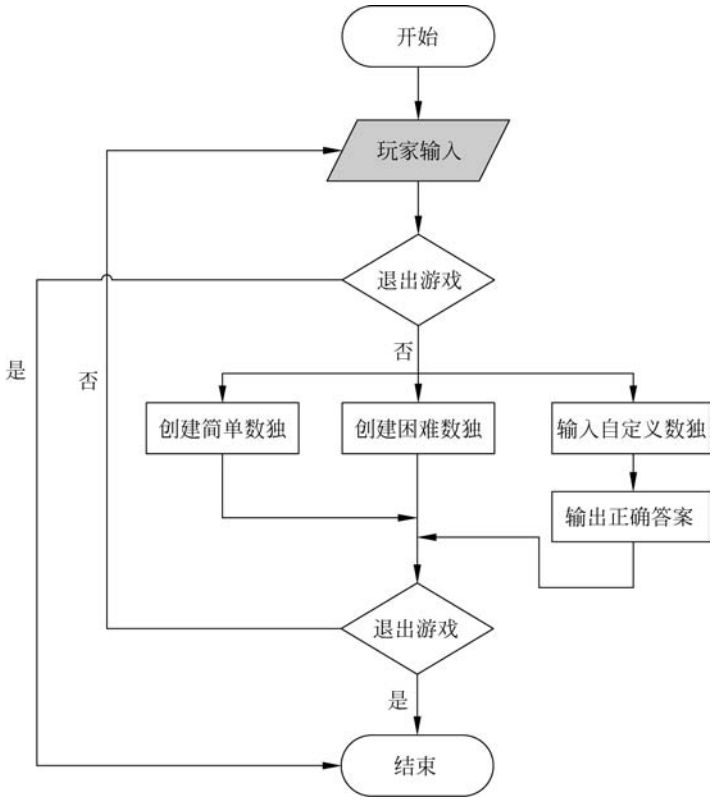


图 3-5　数独游戏流程图

## 3.3　使用 list 存储棋盘状态

数独游戏实际上是在由 81 个格子组成的棋盘上按照一定的规则在空白格子内填入相应的数字(1～9),在游戏设计中需要解决的第 1 个问题就是用什么形式来存储这 81 个格子。因每个格子都需要填写数字,所以可以用整型数据类型来存储每个格子的数值,由于 81 个格子的数据类型相同,并且存在相互关联的关系,所以可以采用 Python 中提供的 list 数据类型来存储这种数据结构。

### 3.3.1　list 数据类型的定义和访问

list 类型是 Python 内置的数据类型,其所有的数据元素均在“[ ]”内。list 内的数据元素用“,”进行分割,理论上每个 list 数据类型可存储 0 个或者无数个数据元素,支持数据元素的动态添加和动态删除。list 数据类型是 Python 中使用最广泛的数据类型之一,在本章和以后章节的游戏编程中将大量使用这种数据类型,读者需要认真掌握 list 数据类型的使用。

**1. list 数据类型的定义**

定义 list 数据类型可以采用赋值运算符“=”来定义具体的 list 变量。接下来看一个具

体的示例。打开 PyCharm 后创建一个名为 listLearn 的工程，在工程中的 main.py 文件里输入以下代码，并按快捷键 Shift＋F10 运行：

```
#第 3 章/listLearn/main.py
li1 = []
li2 = [1,2,3,4,5]
li3 = ['a','b','c']
li4 = [1,2,3,'a','b','c']
li5 = [[1,2,3],['a','b','c']]
print(li1)
print(li2)
print(li3)
print(li4)
print(li5)
```

代码的运行结果如图 3-6 所示。

从图 3-6 的运行结果可知，li1、li2、li3、li4、li5 为 5 个不同的 list 数据类型的输出显示。li1 的内容为空；li2 共存储了 5 个数据元素，每个数据元素的内容为整型；li3 存储了 3 个数据元素，每个数据元素的类型为字符串型；li4 存储了 6 个数据元素，前 3 个数据元素为整型，后 3 个数据类型为字符串型；li5 存储了两个数据元素，第 1 个数据元素为 list 类型，第这两个数据元素也为 list 类型，这两个 list 数据元素又各含了 3 个元素。

**2. list 数据类型的访问方法**

数据类型的定义最终还是要服务于使用，list 数据类型提供了非常方便的访问方法，在具体的访问之前，需要学习一下 Python 提供的 list 索引机制。

1）list 内数据元素的索引与访问

Python 对 list 内的数据元素采用了正向和负向两种索引机制，list 为其内的每个数据元素都分配了两个数字，这两个数字称为数据元素在 list 中的索引，图 3-7 为图 3-6 中 li4 的索引示例。

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

Process finished with exit code 0
```

图 3-6　list 创建运行结果

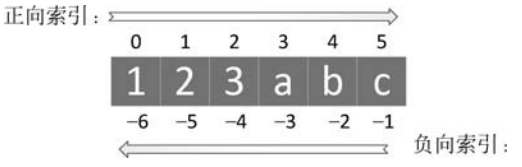


图 3-7　li4 索引示例

从图 3-7 可知，list 数据类型 li4 的每个数据元素都有两个索引。在正向索引中，数据元素的索引为从左到右并从 0 开始依次编号，顺序递增；在负向索引中，数据元素的索引为从右向左并从－1 开始依次编号，顺序递减。

采用索引机制后，list 内数据元素的访问就非常简单了，可以采用“list 数据类型名称[索引]”的访问方法访问 list 内的数据元素。读者可以在 main.py 文件里输入以下代码来感受 list 数据类型索引访问的便利性：

```
#第3章/listLearn/main.py
li4 = [1,2,3,'a','b','c'] #定义 list 数据类型 li4
print(li4[3])             #输出索引为 3 的数据元素
print(li4[ - 1])          #输出索引为 - 1 的数据元素
li4[1] = 4                #将索引为 1 的数据元素修改为 4
print(li4[1])
print(li4)                #输出 li4 的内容
```

从图 3-7 可知，li4 的索引位置为 3 的数据元素为 a，索引位置为－1 的数据元素为 c，所以程序代码的前两行将分别在屏幕上显示 a 和 c。list 支持直接改变索引位置的元素内容，在代码的第 4 行将 li4 的索引位置 1 的内容变成了 4，在第 5 行对索引为 1 的 list 进行内容输出，在屏幕上会显示 4。程序的第 6 行直接输出 li4 的所有内容，屏幕上将显示“1 4 3 a b c”。当要访问的索引编号超过 list 的最大索引时，会发生什么情况？读者可以编写代码进行尝试。

2）list 内数据元素的切片与访问

Python 支持对 list 进行切片访问，通过切片可以直接得到 list 内的一系列数据元素，切片的语法规则如下：

```
[number1:number2:number3]
```

（1）number1：整型数值，表示切片的开始序号，正向切片时默认为 0，负向切片时默认为－1。

（2）number2：整型数值，表示切片的结束序号，但切片内容不包含这个序号，正向切片时默认为 list 的元素个数，负向切片时默认为 list 的元素个数加 1 后取负。

（3）number3：整型数值，表示切片的步长，如果不写 number3，则表示默认步长为 1，这时 number2 后边的“：”可以省略。

切片同时支持正向切片和负向切片，在游戏编程中经常使用，同时也是比较难以理解的内容，读者只要遵循一条准则“number1 无论切片为正向或负向，都是 list 开始位置；number2 无论切片为正向或负向，都是 list 结束位置”，这样掌握切片就不会很困难了。

在 main.py 文件里输入以下代码，并运行，会得到如图 3-8 所示的运行结果。

```
#第3章/listLearn/main.py
li = [10,20,30,40,50,60,70,80,90,100] #将 li 定义为含有 10 个整型元素的 list 类型
print(li[:3])                         #得到索引大于或等于 0,并且小于 3 的所有数据元素
print(li[1:5])                        #得到索引大于或等于 1,并且小于 5 的所有数据元素
print(li[2:])                         #得到索引大于或等于 2 的所有数据元素
print(li[::2])                        #从索引 0 开始,每隔一个元素取一个
print(li[::-1])                       #反向输出 list 的所有元素
print(li[-1:-5:-1])                   #得到索引小于或等于 -1,并且小于 -5 的反向数据元素
```

在 main.py 文件中代码的第一行定义了一个名为 li 的 list 数据类型，li 中共有 10 个数据元素，在内存中的存储如图 3-9 所示。

程序代码的第 2 行对 li 进行[:3]切片，此时 number1 采用默认值 0，number3 采用默认值 1，根据切片规则，又因 number2 的值为 3，切片的结尾取不大于 number2 的最大索引 2，

```
[10, 20, 30]
[20, 30, 40, 50]
[30, 40, 50, 60, 70, 80, 90, 100]
[10, 30, 50, 70, 90]
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
[100, 90, 80, 70]

Process finished with exit code 0
```

图 3-8　list 的切片访问

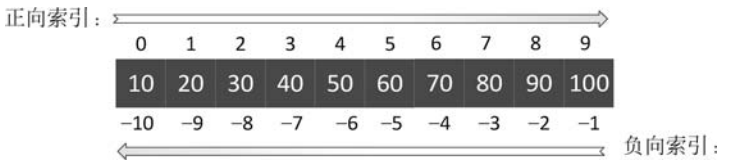


图 3-9　li 的内存存储

故屏幕上将显示索引为 0、1、2 的数据元素 10、20、30。

程序代码的第 3 行对 li 进行[1:5]切片，number1 取值 1，number2 取值 5，number3 采用默认值 1，根据切片规则，屏幕上将显示索引为 1、2、3、4 的数据元素 20、30、40、50。

程序代码的第 4 行对 li 进行[2:]切片，number1 取值 2，number2 使用默认值 li 的元素个数 10，number3 的默认值为 1，根据切片规则，屏幕上将显示索引为 2、3、4、5、6、7、8、9 的数据元素 30、40、50、60、70、80、90、100。

程序代码的第 5 行对 li 进行[::2]切片，number1 取默认值 0，number2 取默认值 li 的元素个数 10，number3 代表的步长为 2，根据切片规则，屏幕上将显示索引为 0、2、4、6、8 的数据元素 10、30、50、70、90。

程序代码的第 6 行对 li 进行[::−1]的负向切片，number1 取默认值−1，number2 取 li 的元素个数加 1 的负数−11，number3 代表步长为−1，根据切片规则，屏幕上将显示索引为−1、−2、−3、−4、−5、−6、−7、−8、−9、−10 的数据元素 100、90、80、70、60、50、40、30、20、10。

程序代码的第 7 行对 li 进行[−1:−5:−1]的负向切片，number1 取值−1，number2 取值−5，number3 取步长值−1，根据切片规则，屏幕上将显示索引为−1、−2、−3、−4 的数据元素 100、90、80、70。

### 3.3.2　数独 81 个格子的 list 存储

在采用 list 数据类型对数独 81 个格子进行存储时，可以定义为含有 9 个 list 数据元素的 list，9 个 list 数据元素又分别含有 9 个整型元素。根据数独的棋盘规则，每一宫格都由 1～9 组成，并且每行和每列的 1～9 数字都不可重复，81 个格子的一种可能数字存储如下：

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

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

在上面的代码中定义了一个名为 sudo 的 list 变量，由 3.3.1 节的索引内容可知，list 内的数据元素都是从 0 开始进行索引的，9 个数据元素，索引为 0～8，在现实世界，人们计数往往从 1 开始计数，这样每个格子的位置就和现实世界的计算规则相差 1。为了解决这个问题，可以将 sudo 定义为含有 10 个 list 数据元素的 list 类型，只是在编程设计时，不使用索引为 0 的数据元素，对 sudo 重新进行定义，代码如下：

```
sudo = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
        [0, 4, 5, 6, 7, 8, 9, 1, 2, 3],
        [0, 7, 8, 9, 1, 2, 3, 4, 5, 6],
        [0, 2, 3, 4, 5, 6, 7, 8, 9, 1],
        [0, 5, 6, 7, 8, 9, 1, 2, 3, 4],
        [0, 8, 9, 1, 2, 3, 4, 5, 6, 7],
        [0, 3, 4, 5, 6, 7, 8, 9, 1, 2],
        [0, 6, 7, 8, 9, 1, 2, 3, 4, 5],
        [0, 9, 1, 2, 3, 4, 5, 6, 7, 8]]
```

## 3.4 使用 for 循环对棋盘格子内容赋值

如 3.3.2 节所述，对已有的棋盘状态可以使用明确的 list 数据类型进行存储，但游戏玩家输入的棋盘状态的数字并不明确，这时就无法直接定义 list 的数据内容，必须找到一种方法可以根据玩家的输入对格子内容依次赋值。

Python 提供的循环可以解决上述问题，其共提供两种循环方法。一种是 2.6 节提到的 while 循环，另外一种是 for 循环，for 循环对于已知循环次数的循环特别适用。接下来看一下 for 循环的具体使用方法。

### 3.4.1 for 循环的定义方法

for 循环的语法格式如下：

```
for <取值> in <序列或者迭代对象的所有值>
    语句块
```

for 循环开始后，首先会将序列或者迭代对象的所有值列出，依次取序列或者迭代对象内的每个值，每取一个值将对语句块运行一次，其具体的流程如图 3-10 所示。

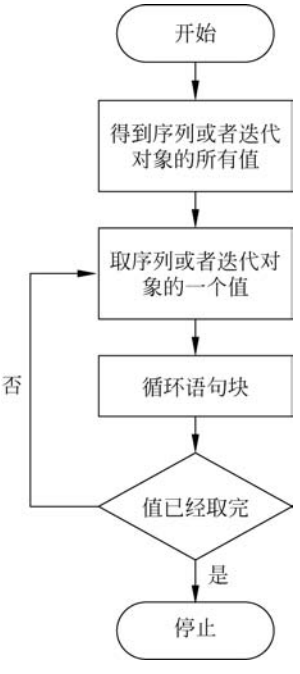


图 3-10 for 循环运行流程图

如何得到 for 循环中“序列或者迭代对象的所有值”从而使循环可以进行下去，是使用 for 循环必须解决的问题，通常使用 range()函数来解决这个问题。

### 3.4.2　range()函数得到迭代对象的所有值

range()函数的使用方法非常类似于 3.3.1 节提到的切片操作，其语法规则如下：

```
range(number1,number2,number3)
```

(1) number1：循环的开始值，默认值为 0。

(2) number2：循环的结束值。

(3) number3：步长值，默认为 1。

依据上述规则，结合 3.4.1 节提到的 for 循环，在 main.py 文件里输入的代码如下：

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

运行上述代码，在屏幕上将显示“0 1 2 3 4”。从运行结果可知，当使用 range()函数得到迭代对象值时，number2 的值将会是一个开区间。

迭代对象默认的序列是正向序列，如果想得到负向序列，则 number1、number2、number3 的值就必须全部提供。在 main.py 文件里输入代码，并查看运行结果，代码如下：

```
for i in range(10,0, - 1):
    print(i,end = ' ')
```

上述代码运行后，在屏幕上将显示“10 9 8 7 6 5 4 3 2 1”，读者可以尝试将步长值 number3 从－1 修改成－2 来体会 range()函数迭代对象的取值方法。

### 3.4.3　for 循环得到用户棋盘

数独游戏的一个重要功能就是得到用户的数独棋盘并给出正确答案。从 3.3 节可知，数独棋盘是一个 9×9 的二维矩阵，但是为了计算方便，采用 10×10 的 list 数据类型进行棋盘存储。游戏得到用户输入的过程实际上就是在对应的二维坐标填入用户输入的相应数据，可以采用 for 循环嵌套的方法来根据玩家的输入填入对应棋盘位置的数据。

以下代码将得到用户的数独棋盘，并在屏幕上打印输出：

```
sudo = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
        [0, 4, 5, 6, 7, 8, 9, 1, 2, 3],
        [0, 7, 8, 9, 1, 2, 3, 4, 5, 6],
        [0, 2, 3, 4, 5, 6, 7, 8, 9, 1],
        [0, 5, 6, 7, 8, 9, 1, 2, 3, 4],
        [0, 8, 9, 1, 2, 3, 4, 5, 6, 7],
        [0, 3, 4, 5, 6, 7, 8, 9, 1, 2],
        [0, 6, 7, 8, 9, 1, 2, 3, 4, 5],
        [0, 9, 1, 2, 3, 4, 5, 6, 7, 8]]
```

```
for k in range(1,10):
    sudo[k][1:10] = list(map(int,input().split()))
for i in range(1,10):
    print(sudo[i][1:10])
```

在代码中使用了两个新函数：input().split()和 map()函数。input().split()函数将用户输入的值按照空格分割成多个值，map()函数将 input().split()函数得到的每个值转换为整型。

## 3.5 使用函数提高代码重复利用率

函数是将组织好的可重复使用的代码组织起来，从而实现某一个功能。函数能提高程序的模块性和代码的重复利用率。到目前为止，已经使用了很多 Python 的内置函数来处理数据，对于内置函数，读者只需知道其具体的调用方法，不必关心其具体的定义。

Python 允许用户自定义函数，其具体的语法规则如下：

```
def 函数名称([形参列表]):
            函数体
```

函数定义的第一行以 def 关键字作为开始，函数名称一般以函数的具体功能作为名称，形参之间以“,”隔开。

以下代码定义了一个求 $n!$ 的函数，读者可以尝试在 main.py 文件里输入代码并查看其运行结果，代码如下：

```
#第 3 章/listLearn/main.py
def factorial(n):
    result = 1
    for i in range(n,0, - 1):
            result = result * i
    return result


print("10 的阶乘为:",factorial(10))
```

在上述代码中，factorial()函数用于得到形参 $n$ 的阶乘。需要说明的是，如果定义的函数有返回值，则需要用 return 关键字进行值的返回。定义好函数后，直接使用“函数名([实参列表])”的方法进行函数调用，例如在上述代码中采用 factorial(10)进行阶乘函数的调用。

数独棋盘的内容需要经常向玩家展现，如果将展现代码定义成一个函数，则程序的可读性将会大大提高。对数独棋盘内容进行展现的函数的代码如下：

```
#第 3 章/sudo/main.py
def printSudo():
    """打印数独"""
    for i in range(1, 10):
        print(sudo[i][1:10])
```

### 3.5.1　函数内的局部变量

定义函数时，在函数内部不可避免地要定义变量，在函数内部定义的变量，其作用范围为函数内部，所以把函数内部定义的变量称为局部变量。

以下代码为求 0＋2＋4＋6＋…＋100 的值的函数，在函数中定义了一个局部变量 *s*，如果在函数外调用 *s*，则会显示错误。输入代码并运行，结果如图 3-11 所示，代码如下：

```
#第 3 章/listLearn/main.py
def sum100():
    s = 0
    for i in range(0,101,2):
        s = s + i
    return s


print("100 以内的偶数的和为",sum100())
print(s)
```

```
100以内的偶数的和为： 2550
Traceback (most recent call last):
  File "C:/Users/24849/PycharmProjects/第2章/forth.py", line 7, in <module>
    print(s)
NameError: name 's' is not defined

Process finished with exit code 1
```

图 3-11　局部变量的使用

从图 3-1 可以看出，在函数外部使用函数内定义的 *s* 变量时，编译器提示 *s* 没有定义，并报错。这说明局部变量只能在函数体中被访问，超出函数体的范围就会出错。

### 3.5.2　函数内使用全局变量

在函数外定义的变量可以在函数内被使用，把这种在函数外定义的变量称为全局变量，全局变量的作用域是整个程序范围。

下面的代码是求圆的面积的函数并在程序中加以调用：

```
#第 3 章/listLearn/main.py
pi = 3.14
def area(r):
    s = pi * r * r
    return s


print("π 的值为:",pi)
print("半径为 3 的圆的面积为",area(3))
```

在上述代码中，pi 变量为全局变量，*s* 为函数内的局部变量。在 area()函数内调用了全

局变量 pi 的值，运行上述代码可得如图 3-12 所示的结果。

从图 3-12 可知，在函数内部使用全局变量 pi 时，直接调用即可，不需要做任何变量说明。

以下代码尝试在函数内改变全局变量 pi 的值，读者可在 main.py 文件里输入代码并运行，运行结果如图 3-13 所示，代码如下：

```
pi = 3.14
def area(r):
    pi = 3.13
    s = pi * r * r
    return s
print("半径为 3 的圆的面积为",area(3))
print("π 的值为:",pi)
```

```
π的值为 3.14
半径为3的圆的面积为： 28.259999999999998

Process finished with exit code 0
```

图 3-12 函数内调用全局变量

```
半径为3的圆的面积为： 28.17
π的值为 3.14

Process finished with exit code 0
```

图 3-13 函数内尝试改变全局变量

从图 3-13 可知，虽然在函数内将全局变量的 pi 的值修改为 3.13，但是函数运行完毕后，pi 的值并没有改变。实际上当函数内将全局变量 pi 重新赋值时，编译器会自动创建一个名为 pi 的局部变量，在函数内接下来的语句将使用同名的局部变量 pi 而不是函数外定义的全局变量 pi。那么，有没有办法在函数内修改全局变量的值？Python 提供了 global 关键字，此关键字可实现这个功能。

还是求圆的面积的函数，对其代码进行修改，运行后得到如图 3-14 所示的结果，代码如下：

```
pi = 3.14
def area(r):
  global pi
  pi = 3.13
  s = pi * r * r
  return s
print("半径为 3 的圆的面积为",area(3))
print("π 的值为",pi)
```

从图 3-14 可知，当函数内需要修改全局变量的值时，需要在函数内使用 global 关键字对全局变量加以说明，这样编译器在函数内碰到声明过的全局变量时，就不会产生局部变量的副本，同时在函数内改变全局变量的值时，函数外的全局变量的值也随之进行了改变。

```
半径为3的圆的面积为： 28.17
π的值为 3.13

Process finished with exit code 0
```

图 3-14 函数内改变局部变量值

在函数定义时经常使用全局变量，在数独游戏设计中也使用了全局变量，并以此进行了棋盘内容的传递，读者只要牢记以下两个准则，全局变量的使用就不会出问题。

（1）在函数内部使用全局变量，如果只是使用而不改变其值，则可以直接使用。

（2）当在函数内部需要改变全局变量的值时，必须在函数体内使用 global 关键字对全局变量加以说明。

## 3.6　建立数独谜题

数独游戏最重要的一个功能就是给玩家建立数独谜题，让玩家求解。建立数独谜题的一个选择是将互联网上所有数独谜题搜集下来，并存储到后台数据库，每次玩家在需要数独谜题的时候，就从后台数据库里分配一个。搜集题库这种方案虽然可行，但是需要进行大量的整理工作，同时题库很有可能还要涉及版权问题，故本书没有选择这种方案。建立数独谜题的另外一个选择是根据数独的数学规律进行谜题建立，这也是本书建立数独谜题所采取的方法。

### 3.6.1　数独棋盘交换不同数字的位置

从 3.1 节已经知道数独游戏的游戏规则，仔细分析数独棋盘的游戏特性可知，如果一个数独游戏的数独谜题已经成立，交换数独矩阵里的所有某两个数字的位置，则数独谜题依旧存在。图 3-15 是一个符合数独游戏规则的棋盘，将棋盘内所有 2 和所有 5 的位置进行交换，便可得到图 3-16 所示的新棋盘。

| | 1列 | 2列 | 3列 | 4列 | 5列 | 6列 | 7列 | 8列 | 9列 |
|---|---|---|---|---|---|---|---|---|---|
| 1行 | 8 | 2 | 5 | 9 | 1 | 6 | 4 | 7 | 3 |
| 2行 | 4 | 7 | 3 | 5 | 8 | 2 | 1 | 6 | 9 |
| 3行 | 1 | 6 | 9 | 3 | 4 | 7 | 8 | 2 | 5 |
| 4行 | 2 | 5 | 8 | 7 | 6 | 3 | 9 | 1 | 4 |
| 5行 | 6 | 9 | 1 | 4 | 2 | 8 | 5 | 3 | 7 |
| 6行 | 7 | 3 | 4 | 1 | 9 | 5 | 2 | 8 | 6 |
| 7行 | 3 | 8 | 6 | 2 | 5 | 9 | 7 | 4 | 1 |
| 8行 | 9 | 4 | 2 | 6 | 7 | 1 | 3 | 5 | 8 |
| 9行 | 5 | 1 | 7 | 8 | 3 | 4 | 6 | 9 | 2 |

图 3-15　符合数独游戏规则的棋盘

从图 3-15 和图 3-16 可知，交换两个不同数字的位置后，数独棋盘里的数字仍旧符合游戏规则。

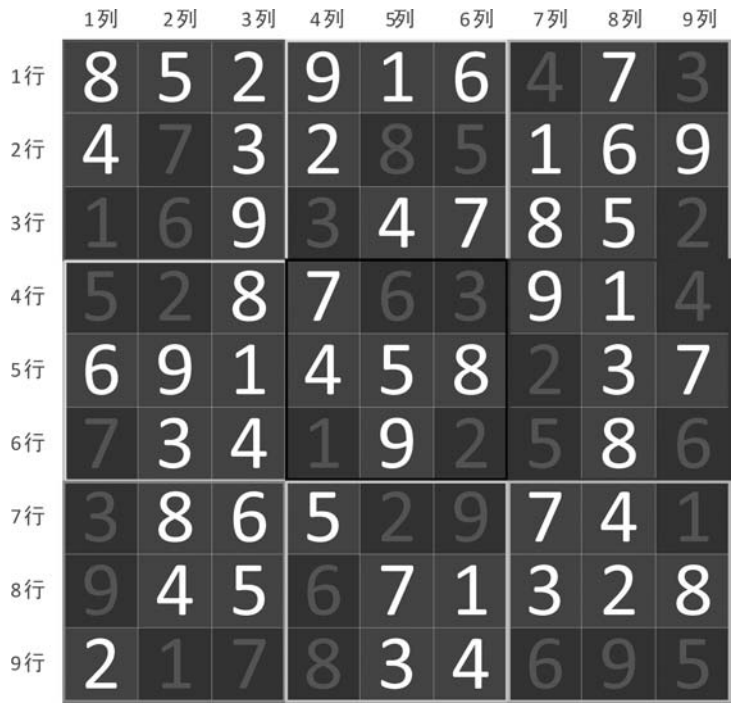


图 3-16　2 和 5 交换位置后的数独棋盘

### 3.6.2　数独棋盘行列交换

依据数学规律可知，对数独棋盘里的不同行之间或不同列之间的数据进行交换后，数独棋盘依旧符合游戏规则。需要注意的是，行或列进行交换的时候，需要注意数独棋盘的宫格的游戏规则。为了保证每一宫格都由不重复的数字(1～9)组成，交换行或者列的时候，只能在同一个宫格内进行。以行交换为例，1～3 行可以随意交换(第 1 行和第 2 行交换或者第 2 行和第 3 行交换)；4～6 行可以随意交换；7～9 行可以随意交换。列交换和行交换类似，此处不再赘述。将图 3-15 所示的棋盘里的数字的第 2 行和第 3 行交换，第 4 行和第 6 行交换，第 7 行和第 9 行交换后，便可得到如图 3-17 所示的棋盘内容。

从图 3-17 可知，行交换后数独棋盘里的数字依旧符合数独的游戏规则。列交换的方法和行交换的方法类似，此处就不再进行描述，读者可以根据列交换规则自行求解。

数独谜题同时支持以 3 行或者 3 列为一个整体进行交换，交换后的棋盘内容也将符合数独游戏规则。例如，可以将 1～3 行或 1～3 列内的内容和 7～9 行或 7～9 列的内容进行整体交换，也可以将 1～3 行或 1～3 列内的内容和 4～6 行或 4～6 列的内容进行交换。将图 3-15 所示的棋盘的 1～3 列内容和 7～9 列内容整体交换后，便可得到如图 3-18 所示的新棋盘。

从图 3-18 可知，3 列作为一个整体进行交换后，新棋盘里的数字也符合游戏规则。

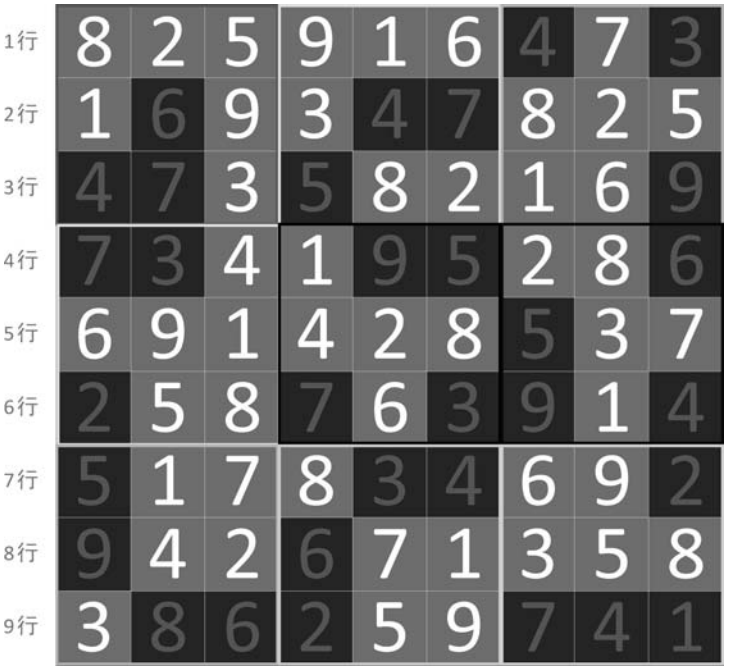


图 3-17　行交换后的数独棋盘

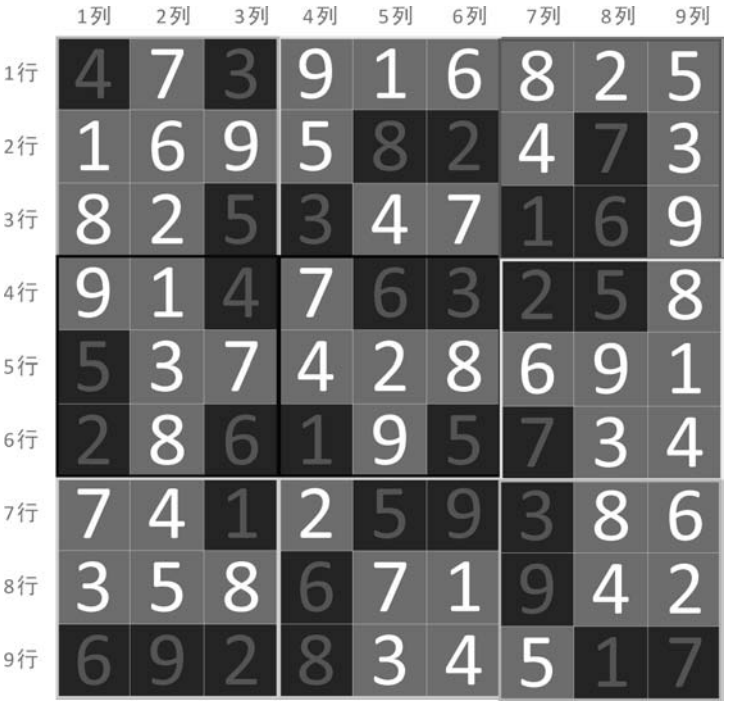


图 3-18　列交换后的新棋盘

### 3.6.3 挖洞建立数独谜题

在数独谜题给玩家进行解答时，需要玩家根据游戏规则在空白方格内填入相应的数字(1～9)，通常来讲未知的空白方格越多，数独谜题的难度就越大。从统计分析可知，49 个未知空白方格对应于数独游戏的简单难度，59 个未知空白方格对应于数独游戏的困难难度。

在数独游戏设计时，采用对数独棋盘挖洞的方法建立数独谜题，可以利用随机函数选取不同的行列位置，将得到的位置内容设为空白方格，在程序中空白方格用 0 表示。具体的挖洞函数的代码如下：

```
#第 3 章/sudo/main.py
def buildHole(num):
    #全局变量 sudo 用于存储原始棋盘
     global sudo
     li = []
     n = 1
#在数独棋盘随机产生 num 个位置，并将位置存储到 li 变量里
     while n < = num:
       temp = [random.randint(1,9),random.randint(1,9)]
       if temp not in li:
          li.append(temp)
          n += 1
#在产生的 num 个位置里放入 0，表示空白方格
    for i in range(len(li)):
        sudo[li[i][0]][li[i][1]] = 0
```

在函数代码中，使用存储了原始棋盘数据的 sodu 变量作为要挖洞的数据，通过[random.randint(1,9),random.randint(1,9)]语句在数独棋盘内产生要挖洞的随机坐标，当产生的坐标的个数等于函数的入口参数 num 时，while 循环结束。for 循环将所有产生的坐标所对应的棋盘位置设置为 0，表示此处为空白方格。

### 3.6.4 数独谜题的具体实现

在 3.6.1～3.6.3 节分析了数独谜题的数学特性与挖洞的方法，本节将利用其特性进行数独谜题的具体实现。

具体的实现流程如图 3-19 所示。

**1. 建立数独谜题主函数**

创建数独谜题的第一步是创建入口函数 generateSudo()，通过这个函数来建立初始数独棋盘，并由此调用混淆函数，其具体的代码如下：

```
#第 3 章/sudo/main.py
#建立数独谜题主函数
def generateSudo():
    global sudo #数独棋盘全局变量
    #建立原始符合游戏规则的数独棋盘
    sudo = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

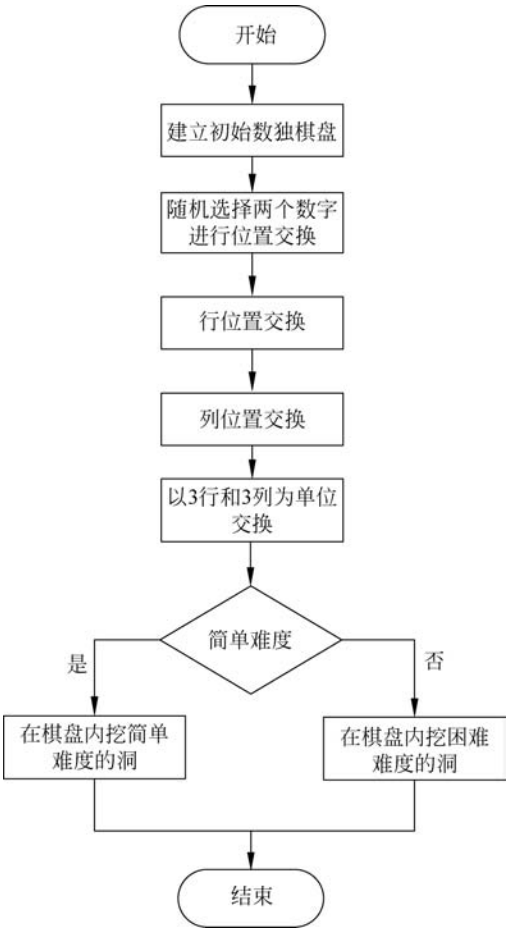


图 3-19　数独谜题实现流程图

```
            [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
            [0, 4, 5, 6, 7, 8, 9, 1, 2, 3],
            [0, 7, 8, 9, 1, 2, 3, 4, 5, 6],
            [0, 2, 3, 4, 5, 6, 7, 8, 9, 1],
            [0, 5, 6, 7, 8, 9, 1, 2, 3, 4],
            [0, 8, 9, 1, 2, 3, 4, 5, 6, 7],
            [0, 3, 4, 5, 6, 7, 8, 9, 1, 2],
            [0, 6, 7, 8, 9, 1, 2, 3, 4, 5],
            [0, 9, 1, 2, 3, 4, 5, 6, 7, 8]]
    nums = random.randint(10,15)
    # 进行 10～15 次数独矩阵位置混淆
    for i in range(nums):
        shuffleSudo()
```

在 generateSudo()函数里对全局变量 sudo 进行了初始值设置，笔者在这里选择了最简的数独棋盘作为初始值，读者也可以根据已有的任意一个数独解的棋盘作为开局初始值，nums 是一个随机值为 10～15 的局部变量，通过这个局部变量的值来调用接下来要提到的 shuffleSudo()混淆函数。通常来讲，nums 的值越大，数独棋盘的混淆度就越大，但是程序

的运行时间就越长，因此这个值不宜选得过大。

**2. 混淆函数进行数值交换**

在混淆函数 shuffleSudo()里，共分为 3 个阶段，第 1 个阶段为数独棋盘交换不同数字的位置，第 2 个阶段为交换行和列的位置，第 3 个阶段为以 3 行和 3 列为单位进行值的交换，其具体的代码如下：

```
#第 3 章/sudo/main.py
#数独棋盘混淆函数
def shuffleSudo():
    global sudo                              #数独棋盘全局变量
    #第 1 阶段：数字交换
    #产生要替换的数字和要被替换的数字
    source = random.randint(1,9)
    replace = random.randint(1,9)
    #产生两个不同的数字
    while source == replace:
       source = random.randint(1, 9)
       replace = random.randint(1, 9)
    #在 9×9 的数独棋盘内对选定的两个数字的位置进行全部替换
    for i in range(1,10):
       for j in range(1,10):
           if sudo[i][j] == source:
              sudo[i][j] = replace
           elif sudo[i][j] == replace:
              sudo[i][j] = source
    #第 2 阶段：行列交换
    #为了保证小宫格也符合规则，1～3 内部交换，4～6 内部交换，7～9 内部交换
    for i in range(5):
       base = random.choice([1,4,7])      #在 1、4、7 里随机选择 1 个数
       #在 base 行开始的三行内随机产生两行
       source = random.randint(base,base + 2)
       replace = random.randint(base,base + 2)
       while source == replace:
           source = random.randint(base, base + 2)
           replace = random.randint(base, base + 2)
       #行交换
       sudo[source],sudo[replace] = sudo[replace],sudo[source]
       #列交换
       for k in range(1,10):
       sudo[k][source],sudo[k][replace] = sudo[k][replace],sudo[k][source]
    #第 3 阶段：3 行和 3 行交换、3 列和 3 列交换
    source = random.choice([1,4,7])
    replace = random.choice([1,4,7])
    while source == replace:
       source = random.choice([1, 4, 7])
       replace = random.choice([1, 4, 7])
    for i in range(3):
       sudo[source + i],sudo[replace + i] = sudo[replace + i],sudo[source + i]
       for k in range(1,10):
           sudo[k][source + i],sudo[k][replace + i] = sudo[k][replace + i],sudo[k][source + i]
```

混淆函数的第 1 个阶段是通过 source 和 replace 两个随机变量产生了要进行位置交换的数字，因为这两个值都是通过 random.randint(1,9)产生的，有可能会产生相同的数值，对相同的数值进行位置交换毫无意义，因此需要使用 while source==replace 循环语句来保证产生不同的交换值。

混淆函数的第 2 个阶段是对 1～3 行、4～6 行、7～9 行的数独棋盘的值进行交换，将 base 变量定义为 1、4、7 中的一个随机值，以 base 为基准再产生两个要交换的行，同时使用 while 循环保证要交换的行号不同。

混淆函数的第 3 个阶段以 3 行和 3 列为整体进行数独棋盘值的交换，通过 source 和 replace 变量分别产生要交换的 3 行的起始行号，通过 while 循环保证 source 和 replace 变量的不同。

## 3.7　深度优先解答数独谜题

2min

解答数独谜题的过程就是在数独棋盘的空白方格内填入数字(1～9)的过程，每填入一个数字需要对数独棋盘上所有的数字进行规则判断。如果当前填入的数字符合游戏规则，就进行下一个空白方格的数字填写；如果填入的数字不符合游戏规则，就换一个数字填写，直到符合规则为止。当前空白方格内填入的数字 1～9 有没有可能都不符合游戏规则？答案是肯定的，这时就必须回溯到上一个已经填写好数字的空白方格处，对其内容更换数字进行填写，以保证当前的空白方格有数字可以符合游戏规则，这种需要进行回溯的解答游戏的算法就称为深度优先。深度优先算法需要递归调用本身，程序编写也较为困难，需要读者仔细体会。

**1. 填写数字后的数独棋盘判断**

在编写深度优先算法前，需先解决较为简单的判断数独棋盘已经填写的数字是否符合游戏规则的问题，其具体的代码如下：

```
#第 3 章/sudo/main.py
def validSudo():
    """校验当前 sudo 数组是否满足数独的规则,True 表示数组符合规则,False 表示数组不符合规
则"""
    #检查行和列是否有重复值
    for i in range(1, 10):
#每一行的开始,将 li_row 和 li_col 设置为空
        li_row = []
        li_col = []
        for j in range(1, 10):
            if sudo[i][j] != 0:
                if sudo[i][j] not in li_row:
                    li_row.append(sudo[i][j])
                else:
                    return False
            if sudo[j][i] != 0:
                if sudo[j][i] not in li_col:
                    li_col.append(sudo[j][i])
```

```
                else:
                    return False
    #检查九宫格是否有重复值
    #得到每个小的九宫格的边界值
    for i in range(1, 10, 3):
        for j in range(1, 10, 3):
        #在判断每个九宫格的重复值前,将 li_sq 设置为空
            li_sq = []
            for row in range(i, i + 3):
                for col in range(j, j + 3):
                    if sudo[row][col] != 0:
        #判断 li_sq 里是否存在当前值
                        if sudo[row][col] not in li_sq:
                            li_sq.append(sudo[row][col])
                        else:
                            return False
        return True
```

为检测当前的 sudo 数组里行列是否存在重复值,定义两个 list 变量 li_row 和 li_col,分别用于存储行和列出现过的值。通过两个 for 循环的嵌套,依次对二维数独棋盘数组中的每个值进行判断。如果当前的值在 li_row 或 li_col 中不存在,则添加到 li_row 和 li_col 中;如果当前的值在 li_row 或 li_col 中已经存在,则说明当前行或者列中存在重复值,函数的返回值为 False。

对每个九宫格内是否存在重复值的判断和行列重复值的判断很类似,不同的是,在对小的九宫格的重复值进行判断时,需要首先使用两个 for 循环定位到每个九宫格的边界,再在边界部分进行两个 for 循环的嵌套,依次判断每个小的九宫格的值是否存在于 li_sq 变量中。如果 li_sq 变量中没有这个值,则添加进去;如果 li_sq 变量中有这个值,则说明存在重复值,函数的返回值为 False。

**2. 数独空白方格位置保存**

每个数独谜题需要填写的空白方格的位置都是不同的,在游戏编程中,空白方格的值以 0 表示。为解答数独谜题,需要把每个要填写位置的空白方格的坐标保存下来,其具体的代码如下:

```
#第 3 章/sudo/main.py
def getPuzzle():
    """将所有的待填入位置保存下来"""
    #全局变量 allPuzzle 用于存储棋盘空白方格的坐标位置
    global allPuzzle
    allPuzzle = []
    for i in range(1, 10):
        for j in range(1, 10):
            if sudo[i][j] == 0:
                p = [i, j]
                allPuzzle.append(p)
```

allPuzzle 全局变量存储了数独棋盘内所有的空白方格的坐标,在代码里通过判断坐标

位置的值是否为0来判断是否为空白方格，如果坐标值为0，则为空白方格；如果坐标值不为0，则不是空白方格。如果是空白方格，则将行列坐标 $i$ 和 $j$ 组成的[i,j]列表存储到allPuzzle全局变量里。

**3. 深度优先递归解决数独谜题**

有了前边的准备工作，接下来就可以采用深度优先递归算法解决数独谜题，其具体的代码如下：

```
#第3章/sudo/main.py
def getAnswer(n):
    global sudo, allPuzzle, over
    #判断是否已经填写完所有的空白方格
    if n == len(allPuzzle):
        over = True
        return True
    #依次对空白方格填入数字(1～9)
    for i in range(1, 10):
    #将空白格子的值设定为i
        sudo[allPuzzle[n][0]][allPuzzle[n][1]] = i
#判断当前填写的数字是否满足数独规则,如果满足,则填下一个空白格子
        if validSudo():
            getAnswer(n + 1)
#如果格子已经填充完毕,则结束函数
        if over:
            return True
#将当前格子重新设置回空白格子
        sudo[allPuzzle[n][0]][allPuzzle[n][1]] = 0
    return False
```

在用深度优先递归解决数独谜题时，函数的初始值 $n$ 为0，表示解答了0个空白格子，当 $n$ 为allPuzzle的大小时，表示解答了所有的数独空白方格，递归函数结束。在函数的内部调用for循环对每个空白方格进行数字(1～9)填写，如果填写的数字符合游戏规则，则对下一个空白方格进行getAnswer(n+1)的递归函数调用；如果当前空白方格依次填入的数字(1～9)都不能符合游戏规则，则回退到上一个空白方格并换别的数字尝试。getAnswer()函数调用结束后，全局变量sudo就存储了所有的空白方格的数字解。

## 3.8　数独游戏代码解析

至此，数独游戏的玩家对数独谜题输入、数独游戏建立、数独游戏解答等数独游戏设计部分已经做了详细的设计。然而对于一个完整的数独游戏来讲，还需要有给玩家交互的主程序部分，通过主程序调用前述已经完成的函数，从而串联起整个游戏代码。运行PyCharm后，新建名为sudo的工程，在工程的main.py文件里输入的代码如下：

```
#第3章/sudo/main.py
#导入随机模块
import random
```

```
#建立数独棋盘的全局变量
sudo = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
        [0, 4, 5, 6, 7, 8, 9, 1, 2, 3],
        [0, 7, 8, 9, 1, 2, 3, 4, 5, 6],
        [0, 2, 3, 4, 5, 6, 7, 8, 9, 1],
        [0, 5, 6, 7, 8, 9, 1, 2, 3, 4],
        [0, 8, 9, 1, 2, 3, 4, 5, 6, 7],
        [0, 3, 4, 5, 6, 7, 8, 9, 1, 2],
        [0, 6, 7, 8, 9, 1, 2, 3, 4, 5],
        [0, 9, 1, 2, 3, 4, 5, 6, 7, 8]]
#建立存储所有待解决的空白方格的坐标的全局变量
allPuzzle = []
#全局变量,用来判断游戏是否结束
over = False
#简单难度为 49 个空格需要填写,困难难度为 59 个空格需要填写
EASY = 49
HARD = 59


def getPuzzle():
    """将所有的空白格子的位置坐标保存起来"""
    global allPuzzle
    allPuzzle = []
    for i in range(1, 10):
        for j in range(1, 10):
            if sudo[i][j] == 0:
                p = [i, j]
                allPuzzle.append(p)


def validSudo():
    """校验当前 sudo 数组是否满足数独的规则,True 表示数组符合规则,False 表示数组不符合规
则"""
    #检查行和列是否有重复值
    for i in range(1, 10):
        li_row = []
        li_col = []
        for j in range(1, 10):
            if sudo[i][j] != 0:
                if sudo[i][j] not in li_row:
                    li_row.append(sudo[i][j])
                else:
                    return False
            if sudo[j][i] != 0:
                if sudo[j][i] not in li_col:
                    li_col.append(sudo[j][i])
                else:
                    return False
```

```
    #检查九宫格是否有重复值
for i in range(1, 10, 3):
       for j in range(1, 10, 3):
          li_sq = []
          for row in range(i, i + 3):
              for col in range(j, j + 3):
                  if sudo[row][col] != 0:
                     if sudo[row][col] not in li_sq:
                        li_sq.append(sudo[row][col])
                     else:
                        return False
    return True


def getAnswer(n):
  """深度优先解决数独谜题"""
    global sudo, allPuzzle, over
    if n == len(allPuzzle):
       over = True
       return True
    for i in range(1, 10):
       sudo[allPuzzle[n][0]][allPuzzle[n][1]] = i
       if validSudo():
          getAnswer(n + 1)
       if over:
          return True
       sudo[allPuzzle[n][0]][allPuzzle[n][1]] = 0
    return False


def printSudo():
    """打印数独"""
    for i in range(1, 10):
       print(sudo[i][1:10])


def shuffleSudo():
    """对已知的数独矩阵进行混淆运算,从而产生新的谜题"""
    global sudo
    #第一步：数字交换
    #产生要替换的数字和要被替换的数字
    source = random.randint(1,9)
    replace = random.randint(1,9)
    while source == replace:
       source = random.randint(1, 9)
       replace = random.randint(1, 9)
    #对 9×9 的棋子进行全部替换
    for i in range(1,10):
       for j in range(1,10):
```

```
            if sudo[i][j] == source:
                sudo[i][j] = replace
            elif sudo[i][j] == replace:
                sudo[i][j] = source
    #第二步：行列交换
    #为了保证小宫格也符合规则,1～3 内部交换,4～6 内部交换,7～9 内部交换
    for i in range(5):
        base = random.choice([1,4,7])
        source = random.randint(base,base + 2)
        replace = random.randint(base,base + 2)
        while source == replace:
            source = random.randint(base, base + 2)
            replace = random.randint(base, base + 2)
        #行交换
        sudo[source],sudo[replace] = sudo[replace],sudo[source]
        #列交换
        for k in range(1,10):
            sudo[k][source],sudo[k][replace] = sudo[k][replace],sudo[k][source]
    #第三步：3 行和 3 行交换、3 列和 3 列交换
    source = random.choice([1,4,7])
    replace = random.choice([1,4,7])
    while source == replace:
        source = random.choice([1, 4, 7])
        replace = random.choice([1, 4, 7])
    for i in range(3):
        sudo[source + i],sudo[replace + i] = sudo[replace + i],sudo[source + i]
        for k in range(1,10):sudo[k][source + i],sudo[k][replace + i] = sudo[k][replace + i],
sudo[k][source + i]


def generateSudo():
"""产生数独谜题"""
    global sudo
    sudo = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
            [0, 4, 5, 6, 7, 8, 9, 1, 2, 3],
            [0, 7, 8, 9, 1, 2, 3, 4, 5, 6],
            [0, 2, 3, 4, 5, 6, 7, 8, 9, 1],
            [0, 5, 6, 7, 8, 9, 1, 2, 3, 4],
            [0, 8, 9, 1, 2, 3, 4, 5, 6, 7],
            [0, 3, 4, 5, 6, 7, 8, 9, 1, 2],
            [0, 6, 7, 8, 9, 1, 2, 3, 4, 5],
            [0, 9, 1, 2, 3, 4, 5, 6, 7, 8]]
    nums = random.randint(10,15)
    #多次混淆运算,从而保证每次产生数独谜题重复的概率降低
    for i in range(nums):
        shuffleSudo()

i = -1
```

```
while i!= 4:
    print("**********    欢迎来到数独游戏    **********")
    print()
    print("1: 创建简单难度数独游戏")
    print("2: 创建困难难度数独游戏")
    print("3: 输入数独并得到答案(每个数字用空格隔开,空白数字用 0 代替)")
    print("4: 退出!")
    print()
    i = int(input("请输入对应的数字: "))
    if i == 1:
        print("简单难度的数独为")
        generateSudo()
        buildHole1(EASY)
        printSudo()
    elif i == 2:
        print("困难难度的数独为")
        generateSudo()
        buildHole1(HARD)
        printSudo()
    elif i == 3:
        for k in range(1,10):
            sudo[k][1:10] = list(map(int,input().split()))
        print("输入的数独答案为")
        getPuzzle()
        getAnswer(0)
        printSudo()
```

主程序里通过 while 循环得到玩家的键盘输入，对于不同的输入会跳转到不同的处理分支。在游戏设计中经常使用 while 的循环判断来得到玩家的输入响应，在后续章节读者会看到更多这样的例子。

## 3.9　小结

本章主要介绍了数独游戏的具体实现，同时对本章涉及的 list 数据类型、for 循环、range()函数、自定义函数、深度优先递归算法等相关知识做了简要介绍。

学习本章后，读者能够掌握 list 数据类型的索引访问和切片访问、多重 for 循环的嵌套使用、自定义函数的定义方法、局部变量和全局变量的使用等知识。

本章采用了挖洞的思想来创建数独谜题，读者也可使用诸如拉斯维加斯等数独谜题创建算法来创建数独谜题。

本章知识可为后续章节的学习打下良好基础。

1
2
3
4
24 ?