问题求解：算法与数据结构

（Python 版）

致 谢

2015 地空数算课程教材第一章翻译小组

组长

柳晓萱

成员

刘明辰、杨润 、杨礼萌、汪颖、党卓、汪诗舜、陈春含、李子涵、蓝坤

1. 引言

1.1. 目标

- 关于计算机科学、程序设计和问题求解的基本概念；
- 什么是“抽象”，及抽象在问题求解过程中的作用；
- 什么是“抽象数据类型”，及其实现；
- Python 程序设计语言入门

1.2. 引子

自从第一台需要人们用线缆和交换机向其传达指令的电子计算机问世以来，编程已经发生了很多改变。在社会各个方面影响下，计算机技术的革新给计算机科学家提供了越来越多的工具以提升他们的技能。高速处理器、高速网络，以及大容量存储器等的发展给计算机科学家制造出了必须克服、且难度螺旋式上升的种种问题。但在这种快速发展下，有一些原则仍旧保持不变：计算机科学主要研究的是利用计算机解决问题。

无疑你已经花费了大量的时间学习解决问题所需要的基础能力，希望你对自己陈述问题并提出解决方案的能力抱有自信。你已经了解了编写程序代码常常具有难度。但往往正是大型问题及其解决方案相应的复杂性掩盖了解决问题的过程中涉及到的基本思路。

本章主要强调接下来学习过程中的两个重要领域。其一，回顾计算机科学及算法与数据结构的基本框架，特别强调我们学习这些内容的原因以及理解这些内容如何帮助我们更好的解决问题。其二，复习 python 编程语言。尽管无法提供详尽的指导，本章将给出具体案例并对其余章节中将会涉及到的基本概念和思想作出解释。

1.3. 计算机科学是什么

通常来说计算机科学难以准确地定义。这或许是因为人们对“计算机”这个词的滥用。如你所知，计算机科学并不仅仅是对计算机进行研究。虽然计算机是支撑这门学科的重要工具，但它也仅仅是工具。

计算机科学是研究问题、问题求解，以及问题求解过程中得出的解决方案。得到一个问题后，计算机科学家的目标是得出一个算法，一个一步一步的指令以解决该问题可能出现的任何情况。算法通过有限过程解决问题。算法是解决方案。

计算机科学可以被看作是对算法的研究。然而，我们必须严谨地考虑到有一些问题是没有解决方案的。虽然证明这个说法超出了本课本的范围，有些问题无法解决这件事对研究计算机科学的人来说很重要。通过包含两类问题，我们可以充分地定义计算机科学，计算机科学研究的是问题的解决方案以及没有解决方案的问题。

描述问题及其解决方案时，“可计算”（computable）这个词是很常见的。当问题存在解决其的算法时，我们说该问题是可计算的。一种可供替代的对计算机科学的定义是说计算机科学研究的是问题是否可计算、算法是否存在。在任何情况下，你将会注意到“计算机”这个词并不常常出现。我们认为问题的解决方案是独立于机器本身的。

计算机科学，就如它包含问题求解过程一样，也研究抽象。抽象使我们可以一种区分所谓的逻辑和物质的方式来看待问题及其解决方案。在常见的案例中这种基本想法对我们来说是很熟悉的。

考虑以下你今天去学校或工作时乘坐的交通工具。作为一个司机，一个机动车的使用者，为了用机动车达到预期目的，你和机动车之间一定有交互。你坐进车里，插入钥匙，发动汽车，换挡，刹车，加速，行驶。用抽象的角度来看，我们可以说，你看到的是逻辑的观点。你使用的是汽车工程师提供的将你从一个位置转移到另一个位置的功能。这些功能有时也被称为界面。

另一方面，汽车修理工有非常不同的观点。他不仅知道如何驾驶汽车，而且知道运行我们认为理所当然的功能的所有细节。他需要了解引擎如何工作，如何传递档位变化，如何控制温度等等。这被认为是物质的角度，发生“在引擎盖下的”细节。

我们使用计算机的时候也是一样的。大多数人用计算机写文档，收发邮件，上网，播放音乐，储存照片以及玩游戏，但它们对于这些程序具体是如何运行的一无所知。它们从逻辑上或是用户的角度看计算机。计算机科学家、程序猿员、技术员以及系统管理员则持有另一种对于计算机的看法。它们必须知道操作系统具体是如何工作的，网络协议是如何配置的以及如何写各种代码来控制这些功能。它们必须能够控制那些一个用户认为是理所应当的底层的详细内容。

这几个例子相同的一点是：这些抽象化内容的用户，有时也叫做客户，不需要知道程序的详细内容，它们只需要知道界面是如何工作的就可以了。界面就是我们作为用户与在底层的复杂的实现交流的途径。再来看一个抽象化的例子——以 Python 的 math 模型为例。只要我们引用了这个模型，我们就可以进行下面的计算：

```
>>> import math
>>> math.sqrt(16)
4.0
>>>
```

这就是一个过程抽象的例子，我们不必要知道平方根是怎么运算的，我们只需要知道这个函数叫什么、如何使用它。如果引用的正确，我们就可以认为这个函数会提供给我们正确的答案。我们知道有人实现了这个问题的解决方式，我们只需要知道怎么用就好了。这有时被叫做过程的“黑箱”观点。我们只简单地描述接口信息：函数名，需要什么（即参数）以及返回值是什么。而函数实现的细节就被隐藏起来了（见图 1）。

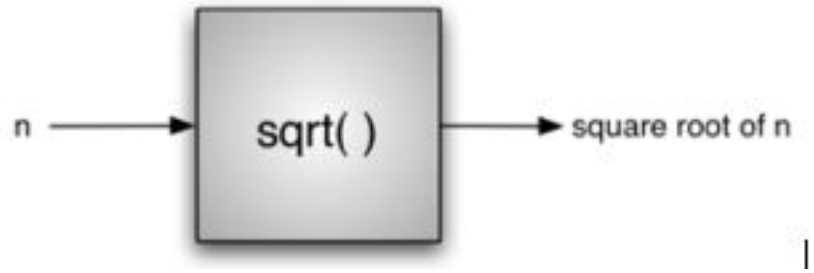

图 1 过程抽象

1.4. 什么是程序设计

程序设计是将算法编码为计算机可执行的表示法或编程语言的过程。虽然如今存在多种编程语言以及多种计算机，但是最重要的第一步还是给出解决问题的方案。没有算法就没有程序。
计算机科学不是研究程序设计的，但程序设计是计算机科学家工作的重要部分。编程通常是我们为我们的解决方案创建一个表示的方式。因此，这种语言表达方式以及创建它的过程成为了这门学科的基本组成部分。

算法描述的是依据代表问题实体的数据和达到预期结果的一系列步骤的问题解决方案。程序设计语言必须提供一种计数性的方式来表达这个过程和这些数据。为此，程序设计语言提供了控制结构和数据类型。

控制结构使得算法的步骤能被简洁而清楚地表达。算法至少要求是能够顺序处理、选择决策和迭代的结构。一种语言只要能够提供这几种基本的语句，就可以被用作算法的表达。

所有数据项在计算机中都用一段二进制数字表示。为了使这些数字能代表数据，我们需要有数据类型。数据类型把这些二进制数据翻译成我们可以理解的、在解决问题中讲得通的内容。这些底层的、预置的数据类型（有时也叫做原始数据类型）是算法发展的基石。

例如很多编程语言都提供了“整型”这一数据类型。此时这一段储存在计算机内存中的二进制数字就被翻译成了整数，并被赋予了我们日常生活中所使用的整数的意义（比如：23，654 和-19）。另外，一种数据类型还提供了数据项可以参与的运算。以整型为例，常见的运算就有加减乘等。我们希望这个数字类型的数据能够参与这些算术运算中。

我们经常遇到的困难是实际上问题和他们的解决方案都很复杂。这些简单的，由程序设计语言提供的结构和数据类型尽管是可以表达复杂解决方案的，但却不利于我们思考解决方法的这个过程我们需要一些办法来控制复杂程度并帮助我们创建解决方案。

1.5. 为何要学习数据结构和抽象数据类型

为了管理问题的复杂度和问题处理的进程, 计算机科学家使用抽象概念使关注点集中在总体框架上，而不会在细节问题上纠缠。通过建立问题域的模型，我们可以更有效地处理问题。这些模型让我

们能够针对问题本身，用一种更一致的方式，去描述我们的算法处理的数据。

早期，我们把对程序的抽象视为一种通过隐藏特定函数的细节让用户可以在更高层次看问题的方法。数据抽象的思想与之相似。抽象的数据类型（有时简写为 ADT），是关于如何查看数据和许可操作的逻辑性描述，不涉及数据、操作是如何被执行的。这意味着我们关注的只是数据代表的含义而不是最终运行的过程。通过提出这种抽象概念，我们建立了对数据的包装。这种理念就是通过对执行的数据的包装，使之从用户视野中隐藏。这就叫做信息隐藏。

图-2 展示了抽象数据的概念及运行机制。用户通过接口实现交互，以使用被抽象数据指定的操作。抽象数据类型是与用户交互的壳。执行的内容被隐藏在更深的一层。用户不关注于执行的细节。

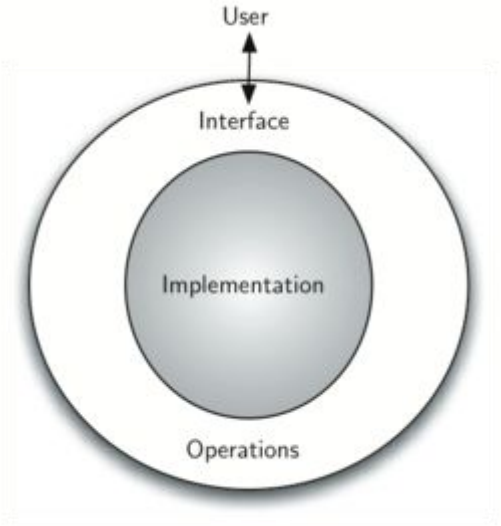

图 2 抽象数据类型

抽象数据类型的执行，即数据结构，需要我们用程序结构的集合和原始数据类型来提供一个物理视图。如前文所述，这两种视图的分离可以在不描述模型建立过程细节的情况下，为所求问题定义一个复杂的数据模型。这给数据提供了一个独立执行的环境。因为执行一个抽象数据通常有不同的方法，因此执行的独立性便可允许程序员在不改变用户与数据交互方式的前提下，对执行细节进行调整。用户便可将注意力保持在解决问题的过程中。

1.6. 为何要学习算法

计算机科学家们从经验中学习。我们则从他人或自己解决问题的过程中学习。接触不同的解决问题的方法并观察不同的算法设计，帮助我们应对接下来遇到的问题挑战。通过思考学习大量不同的算法， 我们可以形成模式识别机制，在下次相似问题出现时，能够更好地解决它。

每一种算法都和其他的有很大不同。考虑到我们之前看到的“sqrt”的例子，很可能有很多不同的方法去执行求平方根函数的计算：某算法可能比其他的使用更多的资源；某算法返回结果可能比其他的多花费 10 倍的时间。有一些方式去比较这两种解决方法。即使它们都能运行，某一种也可能优于另一种。我们可能会认为，某种更有效，或仅仅是运行更快，或占用更少内存。在研究算法的时候，应基于算法本身的特性比较两种解决方案，而并非基于程序或电脑执行算法的特点去比较。

在最坏的情况下，我们可能会遇到难处理的问题，意味着没有算法可以在短时间内解决。将可解决问题、不可解决的问题、有解决方案却耗时耗资源的问题区分开来很重要。

选择算法时经常会面对取舍。作为计算机科学家，除了解决问题的能力，我们还需要掌握解决方案评估技术。最后，一个问题通常有很多解决方法。找到一个方案然后思考它是否是一个好方案将是我们周而复始的任务。

1.7. Python 入门

本章我们将复习编程语言 Python，并给出更多关于前文一些想法的详细实例。如果你新接触 Python 或者觉得你需要进一步了解前文提出的话题，我们建议你去查阅“Python Language Reference”“Python Tutorial”等资料。在这里我们的目标是让你重新熟悉此语言并强化认识一些后续章节的核心概念。

Python 是一种现代化的、易学的、面向对象的编程语言。它拥有一系列强大的内置数据类型和易操作的控制命令。因为 Python 是一种解释型语言，通过浏览并描述交互式会话的方式，你可以很轻松地对它进行复查。你应该记得解释器显示你熟悉的“>>>”提示并评估你给出的 Python 结构。比如，

```
>>> print("Algorithms and Data Structures")
Algorithms and Data Structures
>>>
```

显示了提示，输出函数，结果以及下一个提示。

1.7.1. 从数据开始

我们说过，Python 支持面向对象的编程范式。这意味着 Python 把数据当做问题解决过程的重点。在 Python 里，和很多其他面向对象的编程语言一样，我们定义“类”去描述数据的外观（状态）和功能（行为）。“类”类似于抽象数据类型，“类”的用户只能看到数据项的状态和行为。数据项在面向对象的范式里被称为对象。对象是类的一个实例。

1.7.1.1. 内建原子数据类型

我们将通过讲述 Python 中的基本数据类型开始我们的回顾。Python 拥有两个主要的内嵌的有关数值的类用以记录整数和浮点数。这两个 Python 中的类被称作 int 和 float。而标准的算术运算符，如+，-，*，/和**（乘方）时，在使用时，可以通过使用括号来改变其运算顺序。其它的一些非常有用的运算符还有取模，用%实现，和地板除，用//实现。注意如果两个整数相除，其数学上的结果是浮

点数，而在 Python 中，整数类型的除法运算后只显示商的整数部分，截去了它的小数部分。

```
2. print 2+3*4
3. print (2+3)*4
4. print 2**10
5. print 6/3
6. print 7/3
7. print 7//3
8. print 7%3
9. print 3/6
10.print 3//6
11.print 3%6
12.print 2**100
```

Python 中用以表达布尔数据类型的 bool 的类，在表达真值的时候非常有用。对于布尔对象，状态变量的可能值为 True 或者 False，标准的布尔运算符为 and ， or， not。

```
>>> True
True
>>> False
False
>>> False or True
True
>>> not (False or True)
False
>>> True and True
True
```

代码 1 基本运算符(intro_1)

布尔型的数据对象常常也被用作比较运算例如等于（==），大于（>）的结果表示。此外，关系运算符和逻辑运算符可以被组合起来解决更复杂的逻辑问题。表格一中展示了关系算子与逻辑运算符，并在其后对其作用进行了展示。

运算符名称	运算符	解释
小于	<	小于运算符
大于	>	大于运算符
小于等于	<=	小于或等于
大于等于	>=	大于或等于
等于	==	相等
不等	!=	不相等
逻辑和	and	两者均为真方为真

逻辑或	or	两者有一为真即为真
逻辑非	not	真变假，假变真

表格 1 关系和逻辑运算符

```
print(5==10)
print(10 > 5)
print((5 >= 1) and (5 <= 10))
```

代码 2 基本的关系和逻辑运算符(intro_2)

标识符就是程序语言中被用作当名称的符号。在 Python 中，标识符以字母或下划线作为开始，大小写敏感，并且可以是任意长度。需要注意的是在命名的时候应该选择有意义的名称，这样才能使你的程序变得更加容易阅读和理解，当然，也会变得更加美观。

在 Python 中，当一个名字第一次被用在了赋值语句的左边时，一个变量就随之产生了。赋值语句提供了联系标识符与值的途径。变量会持有一个对数据的引用，而并不是数据本身。参见下面的讲解：

```
>>> theSum = 0
>>> theSum
0
>>> theSum = theSum + 1
>>> theSum
1
>>> theSum = True
>>> theSum
True
```

赋值语句 theSum = 0 创建了一个名为 theSum 的变量，并且使该变量持有对数据 0 的引用（见图三）。第二个例子中，赋值语句的右边是一个求值运算，而一个对最终结果的引用会被派给左边的标识符。在图四中，一个本来是整型的变量被赋了一个 Bool 型的值，这就说明，在 Python 中，如果赋值的数据类型变了，那么变量的类型也会跟着改变。赋值语句改变的是变量所执的引用，这是 Python 极其灵活的一个特征，所以一个变量可以指向许多种类型的数据。

图 3 变量持有对数据的引用

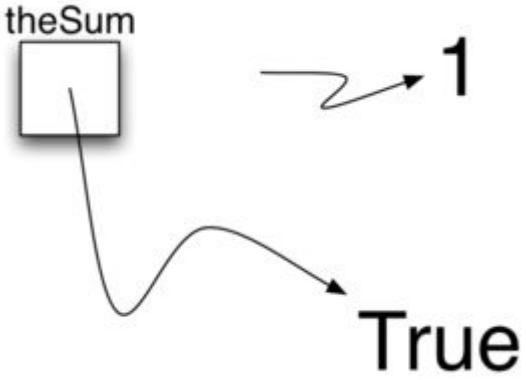

图 4赋值改变了引用

1.7.1.2 内建集合数据类型

除了数值和布尔这两个类，Python还拥有一系列的强大的内嵌的数据容器。列表，字符串和元组都是有序容器，并且在结构上大致相似，但是分别拥有一些需要去好好理解的特别的属性。字典和集合则都是无序容器。

一个列表是包含零个或多个对 Python 中数据的引用的有序容器。列表用方括号括起，里面的每个数据用逗号隔开。空的列表是[]。列表是异质的，这意味着列表中的数据不必要同一个类，并且列表可以被派给一个变量，如下所展示。第一个例子展示了列表中 Python 数据的多种多样。

```
>>> [1,3,True,6.5]
[1, 3, True, 6.5]
>>> myList = [1,3,True,6.5]
>>> myList
[1, 3, True, 6.5]
```

注意到当Python中给一个列表赋值后，会返回一个列表。那么为了记录下这个列表，我们应该将它的引用值传给一个变量。

因为列表是有序的，所以下面提供了一系列可以被用在任何 Python 中的序列的运算符。表格 2 展示并给出了它们的作用：

名称	运算符	讲解
索引	`[ ]`	指向序列中的一个元素
连接	`+`	组合序列
重复	`*`	重复该序列
是否在其中	`in`	查询该元素是否在序列中

长度	`len`	获取序列长度
切片	`[ : ]`	切片操作

表格 2 对任何 Python 序列均有用的运算符

```
>>> myList = [0] * 6
>>> myList
[0, 0, 0, 0, 0, 0]
```

顺便再说一件有关重复操作非常重要的事，就是重复运算的结果是对序列中数据的引用的重复。这个可以在如下的展示中得到体现：

```
myList = [1,2,3,4]
A = [myList]*3
print(A)
myList[2]=45
print(A)
```

代码 3 对引用的重复（intro_3）

变量A持有拥有三份对原始名为mylist序列的引用的容器。注意在对mylist中的元素进行改变后，这种改变体现在了A中。

列表提供了一系列的用来构建数据结构的方法。表格三提供了一个总结，关于它们的用法在之后的例子中呈现：

方法名	用法	解释
`append`	`alist.append(item)`	在列表末尾添加一个新项
`insert`	`alist.insert(i,item)`	在列表的某个位置插入一个项
`pop`	`alist.pop()`	移除并返回列表的最后一项
`pop`	`alist.pop(i)`	移除并返回列表的第 i 项
`sort`	`alist.sort()`	对列表进行排序
`reverse`	`alist.reverse()`	反转列表
`del`	`del alist[i]`	删除在该位置上的元素
`index`	`alist.index(item)`	返回列表中第一个等于 item 项的索引
`count`	`alist.count(item)`	返回列表中有多少项的值等于 item
`remove`	`alist.remove(item)`	删除列表中第一个值等于 item 的项

表格 3 列表提供的一些函数用法

```
myList = [1024, 3, True, 6.5]
myList.append(False)
print(myList)
myList.insert(2,4.5)
print(myList)
print(myList.pop())
print(myList)
print(myList.pop(1))
print(myList)
myList.pop(2)
print(myList)
myList.sort()
print(myList)
myList.reverse()
print(myList)
print(myList.count(6.5))
print(myList.index(4.5))
myList.remove(6.5)
print(myList)
del myList[0]
print(myList)
```

代码 4 列表处理方法示例 (intro_5)

你可以看到一些方法，例如pop，返回了一个值并且同时修改了列表。其他的，像reverse，只是修改了列表没有返回值，Pop默认处理最后一项，但也可以返回一个指定的元素并把它删去。使用这些类函数时，要注意下标也是从0开始的。你也会注意到熟悉的“.”符号，它用于让这个对象调用哪个方法。myList.append(False)可以读作“使对象myList去执行append~~方法~~函数并给它一个值False”。就连最简单的数据体例如整数以这个方式都可以调用~~方法~~函数。

```
>>> (54).__add__(21)
75
>>>
```

如上，我们让一个整数54执行名为“add”的~~方法~~函数（在Python中被称作__add__）然后将21作为值给add，当然我们一般写54+21。我们会在这节后面讲解更多关于方法的知识。

还有一个python中的与list相关的常用功能那就是range。range 产生了一个有顺序的对象，这个对象可以表示一个序列。在使用list里的功能时，可以将range对象里的值视为一个列表。下面是关于它的阐释：

```
>>> range(10)
range(0, 10)
```

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,10)
range(5, 10)
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(5,10,2))
[5, 7, 9]
>>> list(range(10,1,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2]
>>>
```

range对象展现了一个整数的序列。默认的，它会从0开始。如果你提供更多的参数，它可以从特定的点开始和结束，甚至可以跳过一些元素。在第一个例子中，range(10)，这个序列从0开始到10却不包含10，在第二个例子，range(5,10) 从5开始到10却不包含10，range(5,10,2)与上述例子类似，但是每两个数之间相差2（又一次的不包含10）。

字符串是关于零个或多个字母，数字，或其他符号的有序容器。我们称字母，数字和其他的符号为字符。字符串的值用引号（单引号和双引号都可以，只是不能混用的）与标识符区分。

```
>>> "David"
'David'
>>> myName = "David"
>>> myName[3]
'i'
>>> myName*2
'DavidDavid'
>>> len(myName)
5
>>>
```

因为字符串是序列，所以所有对序列的前面讲到的运算符都适用于它。并且字符串也有它的一系列方法，一部分被展现在表格4里，例如：

```
>>> myName
'David'
>>> myName.upper()
'DAVID'
>>> myName.center(10)
'  David   '
>>> myName.find('v')
2
>>> myName.split('v')
```

```
['Da', 'id']
```

在这些里面，split在处理数据方面非常有用。split将接受一个字符串然后返回一个用分隔字符作为切分点的字符串的列表。在上面的例子中，'v'就是分隔符。如果没有给分割的字符，split会自动在空格，换行处切开。

方法名	使用方法	解释
center	astring.center(w)	返回一个字符串，w 长度，原字符串居中
count	astring.count(item)	返回原字符串中出现 item 的次数
ljust	astring.ljust(w)	返回一个字符串，w 长度，原字符串居左
lower	astring.lower()	返回一个字符串，全部小写
rjust	astring.rjust(w)	返回一个字符串，w 长度，原字符串居右
find	astring.find(item)	查询 item，返回第一个匹配的索引位置
split	astring.split(schar)	以 schar 为分隔符，将原字符串分割，返回一个列表

表格 4 Python 中提供给字符串的方法

字符串和列表之间最大的区别就是列表可以被修改而字符串不行。这被称作易变性。列表是可变的，字符串时不可变的。举个例子，你可以通过赋值和索引改变一个列表中的元素，而对于字符串这种操作是不允许的。

```
>>> myList
[1, 3, True, 6.5]
>>> myList[0]=2**10
>>> myList
[1024, 3, True, 6.5]
>>>
>>> myName
'David'
>>> myName[0]='X'

Traceback (most recent call last):
  File "<pyshell#84>", line 1, in -toplevel-
    myName[0]='X'
TypeError: object doesn't support item assignment
>>>
```

元组tuple和列表list类似，也是异质数据序列容器，区别是，tuple是不可变数据类型，就像字符串一样。元表不可被修改，元表写成用圆括号括起的，用逗号隔开的一系列值。当然，作为一个序列，它也适用于上述的关于序列的运算符，例如

```
>>> myTuple = (2,True,4.96)
>>> myTuple
(2, True, 4.96)
>>> len(myTuple)
3
>>> myTuple[0]
2
>>> myTuple * 3
(2, True, 4.96, 2, True, 4.96, 2, True, 4.96)
>>> myTuple[0:2]
(2, True)
>>>
```

但是，如果你试图改变元表里的一个元素，马上就会报错。注意有关error的消息中会包含问题的位置和原因。（改程序也是一件很重要的事啊）

```
>>> myTuple[1]=False

Traceback (most recent call last):
  File "<pyshell#137>", line 1, in -toplevel-
    myTuple[1]=False
TypeError: object doesn't support item assignment
>>>
```

集合是0个或多个数据的无序散列容器。集合不允许多重记录并且表示为花括号括起的用逗号隔开的一系列值。空的集合表示为set()。集合是异质的，并且集合是可变的。

```
如下示例：>>> {3,6,"cat",4.5,False}
{False, 4.5, 3, 6, 'cat'}
>>> mySet = {3,6,"cat",4.5,False}
>>> mySet
{False, 4.5, 3, 6, 'cat'}
>>>
```

尽管集合是无序的，他们仍然支持一些与之前所示的其他容器相类似的操作，表五回顾了这些操作，并且之后给出了如何使用这些操作的具体例子：

运算	运算符	解释
属于关系	in	判断一个元素是否属于这个集合
元素数目	len	返回值是集合中元素的数目
|（并集）	集合 A| 集合 B	返回一个新集合，这个集合是集合 A,B 的并集
&（交集）	集合 A & 集合 B	返回一个新集合，这个集合只有集合 A,B 共有的元素，是集合 A,B 的交集
-	集合 A – 集合 B	返回一个新集合，这个集合是集合 A 除去 A 与 B 共有的元素（A-（A∩B））
<=	集合 A <= 集合 B	判断集合 A 中的所有元素是否都在集合 B 中，返回布尔值 True 或者 False

表五：Python 中集合的运算

```
>>> mySet
{False, 4.5, 3, 6, 'cat'}
>>> len(mySet)
5
>>> False in mySet
True
>>> "dog" in mySet
False
>>>
```

集合提供了许多方法，用于实现一些数学操作，表六大致列举了一些这样的函数，之后也有一些他们的使用实例。注意，union, intersection, issubset 和 difference 这些函数的功能也可用上述在表五中出现的运算符代替。

设有两个集合 A、B，以及不属于这两个集合的元素 item:

函数名	使用方法	解释
union	A.union(B)	返回一个新集合，这个集合含有 A,B 中的所有元素，是集合 A,B 的并集
intersection	A.intersection(B)	返回一个新集合，这个集合只有集合 A,B 共有的元素，是集合 A,B 的交集
difference	A.difference(B)	返回一个新集合，这个集合是集合 A 除去 A 与 B 共有的元素（A-（A∩B））
issubset	A.issubset(B)	判断集合 A 中的所有元素是否都在集合 B 中，返回布尔值 True 或者 False
add	A.add(item)	把 item 这个元素添加到集合 A 中
remove	A.remove(item)	从集合 A 中除去 item 这个元素

pop	A.pop()	从集合 A 中移除任意一个元素
clear	A.clear()	移除集合 A 中的所有元素

表六：Python 中集合操作的函数

```
>>> mySet
{False, 4.5, 3, 6, 'cat'}
>>> yourSet = {99,3,100}
>>> mySet.union(yourSet)
{False, 4.5, 3, 100, 6, 'cat', 99}
>>> mySet | yourSet
{False, 4.5, 3, 100, 6, 'cat', 99}
>>> mySet.intersection(yourSet)
{3}
>>> mySet & yourSet
{3}
>>> mySet.difference(yourSet)
{False, 4.5, 6, 'cat'}
>>> mySet - yourSet
{False, 4.5, 6, 'cat'}
>>> {3,100}.issubset(yourSet)
True
>>> {3,100}<=yourSet
True
>>> mySet.add("house")
>>> mySet
{False, 4.5, 3, 6, 'house', 'cat'}
>>> mySet.remove(4.5)
>>> mySet
{False, 3, 6, 'house', 'cat'}
>>> mySet.pop()
False
>>> mySet
{3, 6, 'house', 'cat'}
>>> mySet.clear()
>>> mySet
set()
>>>
```

这里要说的最后一种 python 的数据类型是字典，字典是由许多对相互之间有联系的元素组成的，每对都是由一个键（key）和一个值（value）组成的，这种元素对被称为键值对，一般记作键：值（key:value）。字典的写法是：若干对键值对排列在一起，之间直接用逗号隔开，前后加上大括号。举个例子：

```
>>> capitals = {'Iowa':'DesMoines','Wisconsin':'Madison'}
>>> capitals
{'Wisconsin': 'Madison', 'Iowa': 'DesMoines'}
>>>
```

在字典的操作中，我们可以巧妙的通过元素的键来获取它的值，也可以通过添加另外的键值对来改动字典。“通过键来获取值”这一操作的语法和从序列中读取元素的操作差不多，只不过后者是通过序列下标来获取，而前者是通过键获取值。添加新元素也是类似的，序列中添加一个新元素，要增加一个下标，而字典则是增加一个键值对。

```
>>> capitals = {'Iowa':'DesMoines','Wisconsin':'Madison'}
>>> print(capitals['Iowa'])
DesMoines
>>> capitals['Utah']='SaltLakeCity'
>>> print(capitals)
{'Wisconsin': 'Madison', 'Utah': 'SaltLakeCity', 'Iowa': 'DesMoines'}
>>> capitals['California']='Sacramento'
>>> print(len(capitals))
4
>>> for k in capitals:
    print(capitals[k]," is the capital of ", k)

('Madison', ' is the capital of ', 'Wisconsin')
('SaltLakeCity', ' is the capital of ', 'Utah')
('DesMoines', ' is the capital of ', 'Iowa')
('Sacramento', ' is the capital of ', 'California')
```

代码1：字典的使用（把运行结果也写上了）

切记，字典型对于键（key）的存储是没有特定的顺序的，如上例中，第一个添加的键值对（’Utah’:’SaltLakeCity’）被放在了字典中的第一个位置，而第二个添加的键值对（’California’:’Sacramento’）被放在了最后一个位置。关于键的摆放位置实际上是和“散列法”这一概念有关，这会在第四章中详细论述。在上例中我们也展示了 length 函数，它和在之前几种数据类型一样，可以用来求字典中所含键值对的数目。

字典既可以通过函数来操作，也可以通过运算符来操作，表 7 和表 8 分别给出了字典中所能使用的运算符，以及具有类似功能的函数，之后的一段代码给出了它们的实际操作。Keys，values 和 items 这三个函数分别能够给出字典中所有的键、值、键值对，然后你可以用 list 函数把它们得到的结果转变为列表。Get 函数有两种不同的用法，第一种，如果查找的 key 不在字典中，它就会返回一个 None，而在第二种中，加入一个可选参数，如果查找的 key 不在字典中，你就可以定义一个特定的返回值。

设有一个字典 mydict

运算符	用法	解释
[]	mydict[‘key’]	返回 key 这个键所对应的值，如果 key 不存在，则会报错
In	key in mydict	如果 key 这个键在字典中，那么就返回 True，如果不在，就返回 False
del	del mydict[‘key’]	在字典中移除 key 这个键所对应的键值对

表 7　Python 中字典的运算符

```
>>> phoneext={'david':1410,'brad':1137}
>>> phoneext
{'brad': 1137, 'david': 1410}
>>> phoneext.keys()
dict_keys(['brad', 'david'])
>>> list(phoneext.keys())
['brad', 'david']
>>> phoneext.values()
dict_values([1137, 1410])
>>> list(phoneext.values())
[1137, 1410]
>>> phoneext.items()
dict_items([('brad', 1137), ('david', 1410)])
>>> list(phoneext.items())
[('brad', 1137), ('david', 1410)]
>>> phoneext.get("kent")
>>> phoneext.get("kent","NO ENTRY")
'NO ENTRY'
>>>
```

设有个字典 adict

函数名	使用方法	解释
keys	adict.keys()	以列表的形式返回 adict 中的所有键(key)
values	adict.values()	以列表的形式返回 adict 中的所有值(value)
items	adict.items()	以列表的形式返回 adict 中的所有键值对，列表的每个元素是包含键和值的元组
get	adict.get(key)	返回 key 所对应的值，如果 key 不存在，就返回 None
get	adict.get(key,alt)	返回 key 所对应的值，如果 key 不存在，就返回 alt

表 8　python 中字典中的函数操作

1.7.2 输入与输出

无论是获取数据，或者是返回某种结果，我们都经常需要与用户进行互动，如今的大多数程序都用一个对话框作为一种要求用户提供某种类型的输入的方式。虽然 Python 确实也有办法创建对话框，但我们有更简单的函数可以来解决这个问题。Python 就为我们提供了这样一个函数，它能要求用户输入某些数据，同时以字符串的形式为用户提供输入提示（就是输入的时候提示个“please input your data”之类的）。这个函数就是 input。

Python 中的 input 函数后面的括号里只能接受单一的字符串，这个字符串被称为提示符（prompt），因为这个字符串一般输入的是有用的文本信息，用于提示用户来输入某些东西。举个例子，你可以这样用 input：

```
aName = input('Please enter your name: ')
```

在这条语句执行之后，无论用户在看到了提示之后输入了什么，它都会被存放在 aName 这个变量中。使用 input，我们就能轻松的写出提示用户输入数据的指令，然后把用户输入的数据存储起来，以便进行进一步的操作。例如，在下面的这两条语句中，第一条让用户输入他们的名字，然后第二条语句对于他们输入的字符串进行了简单的处理，然后输出了处理的结果。

```
aName = input("Please enter your name ")
print("Your name in all capitals is",aName.upper(),
    "and has length", len(aName))
```

代码 3 input 函数，并且返回一个字符串

切记，input 函数在接收到用户的输入后，它的返回值是一个字符串，其表现了用户在看到输入提示后所输入的数据类型的准确特征，如果你想把这个字符串转化为另外一种数据类型，你必须进行明确的类型转换。在下面的这些语句中，用户输入的字符串被转化为了浮点型，从而输入的数据可以进行进一步的运算处理。

```
sradius = input("Please enter the radius of the circle ")
radius = float(sradius)
diameter = 2 * radius
```

1.7.2.1 字符串格式化输出

在之前我们已经了解了 print 函数是一种非常简便的从 python 程序中输出数据的方式，它可以获取 0 个或多个参量并将其输出，其间默认用空格作为分隔符。事实上，我们可以通过设定 sep 参数来修改分隔符。此外，每一行输出都默认以新行字符（>>>）结束，这同样可以改变，使用 end 参数就能达成目的。以下的代码展示了如何进行这种变更。

```
>>> print("Hello")
Hello
>>> print("Hello","World")
Hello World
>>> print("Hello","World", sep="***")
Hello***World
>>> print("Hello","World", end="***")
Hello World***>>>
```

在程序中，固定格式的输出往往是很有用的，Python 提供了一种替代性的输出，称为格式化字符串。格式化字符串是一个供填空的模版，其中的单词或空格会保持原样，而占位符则留给变量来填充。例如，这个语句

```
print(aName, "is", age, "years old.")
```

包含单词“is”和“years old”，但是其中的 name 和 age 会随着操作过程中变量值的改变而改变。利用格式化字符串，我们就能把这行语句写成：

```
print("%s is %d years old." % (aName, age))
```

这个简单的例子展示了一种新的串表达式。“%”是字符串运算符，被称为格式操作符。在上面这个表达式中，%的左边是模版或者格式字符串，右边则是一个容器，其中包含了替换格式字符串的变量值。注意，右边容器中的变量的个数必须和左边字符串中的占位符的数目相一致，在用变量值替换占位符的过程中，变量从左到右依次读取，按顺序替代占位符的位置。

现在我们更细致的观察这种格式化表达的左右两侧，格式字符串可以拥有一个或多个需要用变量来替代的占位符。%后面的字母是转换字符，它告诉了格式运算符何种类型的变量将会填充到字符串的这个位置。在上面的那个例子中，%s 指定了填入的变量是字符串，而%d 指定了是整数。其它可能的转换字符包括 i， u，f，e，g，c 或%。表 9 概述了各种不同类型的转换字符。

字符	输出格式
d,i	整型
u	无符号整型
f	浮点型，如 m.ddddd
e	浮点型，如 m.dddddе+/-xx
E	浮点型，如 m.dddddE+/-xx
g	指数比-4 小或比 5 大时使用%e，否则使用%f

c	单字符
s	字符串或者是能通过 str()转为字符串的数据
%	输入一个%

表 9　格式化字符串转换字符

除了转换字符之外，你也可以在%和转换字符之间插入格式修饰符。格式修饰符可以用一段给定长度的空格使变量实现左对齐或者右对齐。格式修饰符也可在小数点后添加数字，来指定字段宽度。表 10 给出了这些格式修饰符的使用。

修饰符类型	例子	描述
number	%20d	变量值占据 20 个字符宽度
-	%-20d	变量值占据 20 个字符宽度，左对齐
+	%+20d	变量值占据 20 个字符宽度，右对齐
0	%020d	变量值占据 20 个字符宽度，前置“0”
.	%20.2f	变量值占据 20 个字符宽度，且保留两位小数
(name)	%(name)d	从字典中取 key 为 name 的值放在此处

表 10　格式修饰符操作

格式操作符（%）的右侧是一个由变量组成的容器，其中的变量值将会被插入左侧的格式字符串，这个容器既可以是元组，也可以是字。如果这个容器是元组，那么变量值的插入是按顺序的，也就是说，元组中的第一个元素对应于格式化字符串的第一个占位符，然后从左往右依次对应。如果这个容器是字典，那么变量值是按照他们的键（key）来插入的，在这种情况下，所有的转换字符必须用表格中的(name)修饰符来指定键（key）的名称，从而插入其值。

```
>>> price = 24
>>> item = "banana"
>>> print("The %s costs %d cents"%(item,price))
The banana costs 24 cents
>>> print("The %+10s costs %5.2f cents"%(item,price))
The    banana costs 24.00 cents
>>> print("The %+10s costs %10.2f cents"%(item,price))
The    banana costs      24.00 cents
>>> itemdict = {"item":"banana","cost":24}
>>> print("The %(item)s costs %(cost)7.1f cents"%itemdict)
The banana costs    24.0 cents
>>>
```

除了能使用转换字符以及格式修饰符的格式化字符串外，Python 中的字符串也包括了一个名为 format 的函数，它可以和一个新的类——Formatter 结合，以实现更复杂的字符串格式化。关于 python

的这一特色，可以查阅 Python Library 的参考手册。

1.7.3 控制结构

就像我们前面所说的，算法要求两个重要的控制结构：迭代和选择。它们的不同形式都被 Python 所支持，操作者可以选择对于所给问题最有用的表述。

对于迭代，Python 提供了一个有用的 while 语句和 for 语句。While 语句可以在循环条件是真的情况下一直重复循环体。例如：

```
>>> counter = 1
>>> while counter <= 5:
...     print("Hello, world")
...     counter = counter + 1


Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
```

重复输出“Hello world”5 次，循环条件在每次循环开始时都得到判断，如果判断结果为真，循环主体将得到执行。由于强制缩进的模式执行语言，我们可以很容易看到 Python 中的 while 循环结构。

While 循环语句是一个非常常见的迭代结构，我们将会在很多不同的算法中使用到。在很多情况下，一个复合条件将会控制迭代结构 。例如

```
while counter <= 10 and not done:
...
```

这段代码会导致语句的主体只有在所有的条件都满足的情况下才会执行。变量 counder 的值必须小于或者等于 10 并且变量 done（布尔型）必须是 False(not false 就是 ture)，因此真+真=真。

这种类型的结构在各种情况下非常有用。另外一个迭代结构----for 语句可以和许多 python 集合体共同使用。For 语句可以遍历一个集合的元素，只要这个集合是一个序列集合。例如：

```
>>> for item in [1,3,6,2,5]:
...    print(item)
...
1
3
6
```

```
2
5
```

分配每个连续值的变量条目列表中(1、3、6、2、5)。然后执行迭代的主体。这适用于任何集合序列(列表、元组和字符串)。

For 语句的作用是明确迭代值的范围，例如：

```
>>> for item in range(5):
...    print(item**2)
...
0
1
4
9
16
>>>
```

该语句将会执行输出功能 5 次。函数将会返回一系列对象范围代表序列（0,1,2,3,4,）每个值将分配给变量项，然后会将每个值平方后输出。

另外一个迭代结构非常有用的功能是处理每个字符串的字符。下面的代码片段列表遍历每个字符串的字符，并且将每个字符都添加到列表。结果是含有所有字母的单词的一个列表。

```
wordlist = ['cat','dog','rabbit']
letterlist = [ ]
for aword in wordlist:
    for aletter in aword:
        letterlist.append(aletter)
print(letterlist)
```

代码 8 Processing Each Character in a List of Strings (intro_8)

选择语句允许程序员提出问题,然后根据结果,执行不同的操作。大多数编程语言提供了两种形式的有用的结构:if else 和 if。下面是一个简单的二元选择使用 ifelse 语句的例子。

```
if n<0:
   print("Sorry, value is negative")
else:
   print(math.sqrt(n))
```

在这个例子中,对象 n 被用于检查是否小于零。如果是,输出一个消息,指出它是负的。如果不是,该声明执行 else 子句，计算平方根。

选择结构,与任何控制构造一样,可以嵌套,这样得到的结果可以帮助决定是否问下一个问题。例如,假设得分是一个引用变量用于对计算机科学进行测试。

```
 if score >= 90:
   print('A')
else:
   if score >=80:
      print('B')
   else:
      if score >= 70:
         print('C')
      else:
         if score >= 60:
            print('D')
         else:
            print('F')
```

这个片段将通过输出字母将分数进行分类。如果比分是大于或等于 90,将输出 A。如果没有(else),下一个问题将会进行。如果比分是大于或等于 80,那么它必须在 80 年和 89 之间，否则对第一个问题的答案是错误的。在这种情况下将会输出 B。我们可以看到 Python 缩进模式有助于理解 if 和 else 之间的联系，无需任何额外的语法元素。

这种类型的嵌套的另一个语法的选择是使用 elif 作为关键字。else 和 if 结合,消除了需要额外的嵌套级别。注意,最后的 else 仍然是必要的。 因为默认有可能其他条件失败。

```
if score >= 90:
   print('A')
elif score >=80:
   print('B')
elif score >= 70:
   print('C')
elif score >= 60:
   print('D')
else:
   print('F')
```

Python 中也有一个单向选择结构，if 语句。有了这个语句，如果条件为真，则执行操作。在条件为假的情况下，仅仅执行后面的语句。例如，下面的代码片段将首先检查是否一个变量 n 的值是负的。如果是，则其由绝对值函数修改它的值。无论如何，下一句代码是计算平方根。

```
if n<0:
   n = abs(n)
```

```
print(math.sqrt(n))
```

自我检查

通过尝试下面的练习来测试你是否理解了以上内容。修改 Activecode8 的代码，使最终列表只包含每个字母的单一副本。

```
# the answer is: ['c', 'a', 't', 'd', 'o', 'g', 'r', 'b', 'i']
```

代码 4（self_check_1）

在列表中，存在用于创建使用迭代和选择结构的列表的替代方法。在被称为列表解析。列表解析可以让你轻松创建基于某些处理或选择条件的列表。例如，如果我们想创建的小于 10 的整数的完全平方的列表，我们可以使用 for 语句：

使用列表解析，我们可以一步完成：

```
>>> sqlist=[]
>>> for x in range(1,11):
        sqlist.append(x*x)

>>> sqlist
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>>
```

使用列表解析，我们可以一步完成以上操作

```
>>> sqlist=[x*x for x in range(1,11)]
>>> sqlist
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>>
```

在 for 结构中，变量 x 取从 1 到 10（不包括 10）的整数值。 计算 x * x 的值后，将结果添加到新建的列表中。列表解析中也可以使用选择语句，以便只有某些项目进行运算并添加到新的列表中。例如，

```
>>> sqlist=[x*x for x in range(1,11) if x%2 != 0]
>>> sqlist
[1, 9, 25, 49, 81]
>>>
```

这个列表解析建立了一个仅含有 1 到 10 内奇数的平方值的列表，需要进行迭代运算的任何序列可

以在列表解析中被用来构造一个新的列表。

```
>>>[ch.upper() for ch in 'comprehension' if ch not in 'aeiou']
['C', 'M', 'P', 'R', 'H', 'N', 'S', 'N']
>>>
```

自我检查

通过重做 Activecode8 测试你对列表解析的理解。另外，看看你能不能删除重复的元素。

```
# the answer is: ['c', 'a', 't', 'd', 'o', 'g', 'r', 'a', 'b', 'b', 'i', 't']
```

代码 5 (self_check_2)

1.7.4 异常处理

写程序时，通常会出现两种错误。第一，所谓的语法错误，简单的说就是程序员在语句结构或表达中犯了一个错误。例如，在 for 语句中没有写冒号是错误的

```
>>> for i in range(10)
SyntaxError: invalid syntax (<pyshell#61>, line 1)
```

在这种情况下，Python 编译器发现它不能完成该指令的处理，因为它不符合语言的规则。当你第一次学习一门语言时，语法错误往往更频繁。

另一种错误是逻辑错误，程序可以执行，但给出的结果是错误的。这可能是由于底层的算法中的错误或者错误表达了该算法。在某些情况下，逻辑错误导致严重的后果，如试图除以零或在试图访问列表中的一个项目时越界了。在这种情况下，逻辑错误导致运行错误，程序终止。这些类型的运行错误的通常被称为程序异常。

大多数的时候，初学者简单地认为 exceptions（异常）是导致执行结束的运行错误。但是，大多数编程语言提供了一种方法来处理这些错误，让程序员在遇到这些运行错误时有一些干预措施。此外，如果他们需要这些异常对程序进行检测的话，程序员可以创建自己的异常。当异常发生时，我们说，它已经“提出，”你可以通过使用 try 语句“处理”已提出的异常。例如，考虑下一段代码要求用户输入整数，然后调用 python 库函数中的平方根函数。如果用户输入一个值，该值大于或等于 0 时，打印会显示平方根。如果用户输入一个负值，平方根函数将报告一个 ValueError 异常。

```
>>> anumber = int(input("Please enter an integer "))
Please enter an integer -23
>>> print(math.sqrt(anumber))
```

```
Traceback (most recent call last):
  File "<pyshell#102>", line 1, in <module>
    print(math.sqrt(anumber))
ValueError: math domain error
>>>
```

我们可以通过从 try 模块中调用打印功能处理这个异常。相应的，模块除了“捕获”了异常，还打印一条消息返回给用户。例如：

```
>>> try:
       print(math.sqrt(anumber))
    except:
       print("Bad Value for square root")
       print("Using absolute value instead")
       print(math.sqrt(abs(anumber)))

Bad Value for square root
Using absolute value instead
4.79583152331
>>>
```

Try 语句将发现异常产生于调用平方根函数时，并将该信息返回给用户，并提醒用户使用数值的绝对值，以确保我们在对一个非负数进行开方运算。这意味着，程序不会终止，而是会继续到下一个语句。

另外，程序员可以通过使用 raise 语句自己制造运行异常。例如，我们不调用开方的函数，我们先检查输入值，然后提出我们自己的异常。下面的代码片段显示了创建一个新的 RuntimeError 异常的结果。请注意，该计划将仍然会终止，但现在导致终止异常是由程序员明确创建

```
>>> if anumber < 0:
...     raise RuntimeError("You can't use a negative number")
... else:
...     print(math.sqrt(anumber))
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
RuntimeError: You can't use a negative number
>>>
```

除了上述所示的 RuntimeError 异常，还有其他他种类的异常。所有可用的异常类型以及如何创建你自己的异常可见 Python 的参考手册。

1.7.5 定义函数

刚才的例子从数学模块 Python 函数中调用了开方的函数。在一般情况下，我们可以通过定义一个函数隐藏任何计算的详细步骤。一个函数定义需要一个名字，一组参数，和函数主体。它也可以明确地返回一个值。例如，下面定义的简单函数返回它的值的平方。

```
>>> def square(n):
...     return n**2
...
>>> square(3)
9
>>> square(square(3))
81
>>>
```

此函数语法的定义包括名称，square，形式参数组合列表。在以上函数中，n 是唯一的形式参数，这表明 square 函数只需要输入一个数据经行运行。函数主体计算 n 平方的结果，并返回。我们可以通过在 Python 环境中调用 square 函数来评价它。给 square 函数传递一个实际参数值，在上述例子中，我们把 3 这个实际数值传给了函数的形式参数。注意，调用 square 函数返回的值可以被其他操作进行使用。

我们可以通过使用“牛顿”法实现我们自己的平方根函数。用牛顿法逼近平方根计算，使结果收敛于正确的值。等式 newguess= 1/2 *（oldguess+ N / oldguess）取一个值 n，重复地执行以上等式，每一次等式计算结果 newguess 变成下一次迭代的 oldguess 带入进行计算。这里使用的初值为 n/2。清单 1 显示了一个函数，他接受一个值 n，并返回做 20 重复计算之后的值。再次，牛顿法的具体步骤隐藏在函数定义内，并且用户不必知道该函数为达到预计目的如何实现这个功能。清单 1 还显示使用＃字符作为注释标记。＃后的任何字符都被忽略。

Listing 1

```
def squareroot(n):
    root = n/2    #initial guess will be 1/2 of n
    for k in range(20):
        root = (1/2)*(root + (n / root))

    return root
```

```
>>>squareroot(9)
3.0
>>>squareroot(4563)
67.549981495186216
>>>
```

自我检查

这里有一个涵盖了我们所学所有知识的题目。你可能已经听说了无限猴子定理？该定理指出，猴子随机在打字机键盘键入一个字符，经过无限时间后，肯定会键入一系列给定的文字，比如莎士比亚全集。好吧，假设我们有一个 Python 函数替换猴子。你认为多久以后生成一句莎士比亚的名言？我们将这句话定为：“methinks it is a weasel”

你不会希望在浏览器中运行这个程序，所以使用的 Python IDE 进行编译吧。我们将模拟这个问题的方法是编写一个函数，该函数生成一个 27 个字符长度的字符串，从 26 个字母和空格中随机选择一个字符。我们将编写另一个函数，来比较随机生成的字符串和目标字符串。

第三个函数将反复调用生成和比较函数，那么如果所有目标字母都在随机字符串中出现了，我们就完成了。如果字母没有全部出现，我们会生成一个全新的字符串.为了让它更易于跟随你的程序的过程，第三个函数应该返回出到目前为止产生的最好的字符串，并返回在产生这个字符串之前每 100 次尝试中产生其它不合题意的字符串的次数。

自我检查的挑战

看看你是否可以这样优化程序，保留正确的字母，只修改符合到目前为止与目标字符串最接近的字符串中的一个字符。如果新生成的字符是目标字符串中需要的，我们就用这个字符覆盖前一个字符串中不合题意的字符，这是一种类似“爬坡”的算法。

1.7.6. Python 面向对象编程：定义类

我们之前声明了 Python 是一种面向对象的编程语言，到目前为止我们已经用了很多内置的类来展示数据和控制的构成。但是，面向对象的编程语言一个最重要的特征是允许编程者（问题求解程序）来创造一个可以用来建立需要用来解决问题的数据模型的新的类。

记住我们运用抽象的数据类型来提供一个关于一种数据项目是什么样子（它的阐述）和它能做什么（它的方法）逻辑描述。通过建立类来实现一种抽象数据类型，一个编程者可以从中获得抽象过程的好处同时也提供必须的细节来将这些抽象实际运用到一个程序中。无论何时我们想实施一种抽象数据类型，我们将依靠新的类来做它。

1.7.6.1.示例：Fraction 类

用一个非常简单的例子来展现实施一个使用者定义的类是建立一个 Fraction 类来实施抽象数据类型 。我们已经知道 Python 提供很多数字的类供我们使用。有时候，然而，可能创建一个“类似” fraction 的数据类型会更有可观性。

一个 fraction 如 3 / 5 包括两部分。最高值，称为分子，可以是任何整数。底部的值，称为分母，可以是任何大于 0 的整数（负的 fraction 有负的分子）。虽说我们可以创建一个浮点数近似任何 fraction，但在有的情况下我们还是想用分数表示一个确切的值。

对 Fration 运算符应使对 Fraction 数据对象的运算操作像任何其他的数值一样，我们需要能加，减，乘，和除。我们也希望能够显示 fraction 使用标准的“分数”的形式，例如 3 / 5。此外，所有 fraction 的运算应该以最简的形式返回结果，这样不管怎样进行计算，我们最后总是最常见的形式。

在 Python 中，我们通过提供一个名字和设立一个方法（在语法上类似于函数定义）来定义一个新的类。

举这个例子，

```
class Fraction:

   #the methods go here
```

为我们提供了定义方法的框架。对于所有的类而言第一步提供的都应是构造函数。构造函数中定义了数据对象的创建方式。创建一个 Fraction 的对象，我们需要提供两块数据，分子和分母。在 Python 中，构造函数的方法经常被称作__init__（两个下划线在 init 之前和之后），如清单 2 所示。

Listing 2

```
class Fraction:

    def __init__(self,top,bottom):

        self.num = top
        self.den = bottom
```

注意，正式的参数列表包含三项（self,top,bottom）。self 是一个特殊的参数，都可以用来作为参考返回对象本身。它必须是第一个正式参数；然而，调用时它将永远不需给出一个实际参数值。如前所述，Fraction 需要两块状态数据，分子和分母。在构造函数符号 self.num 定义 Fraction 对象有一个内部的数据对象被称为 num 作为它的状态。同样，self.den 被创造为分母。这两个形参的值最初被分配的状态，使新的 fraction 对象知道它的起始值。

为创造一个 Fraction 类的实例，我们必须借助构造器。这会在使用类的名称和传递实际数值的时候发生（注意我们不会直接借助__init__）。例如，

```
myfraction = Fraction(3,5)
```

这里创建一个对象，称为 myfraction 代表分数 3 / 5（五分之三）。图 5 显示了这个目标现在的实现。

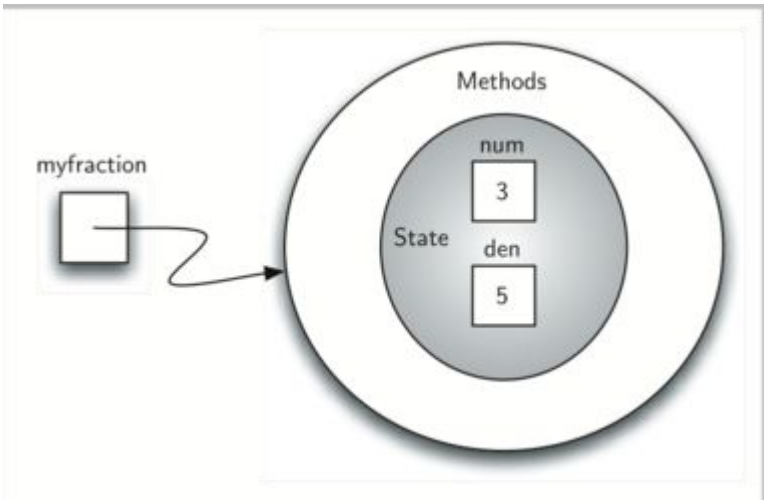

图 2 An Instance of the Fraction Class
一个分数类的实例

我们要做的下一件事是实现抽象数据类型要求的行为。首先，考虑当我们尝试打印分数对象会发生什么。

```
>>> myf = Fraction(3,5)
>>> print(myf)
<__main__.Fraction instance at 0x409b1acc>
```

fraction 对象，myf，不知道如何回应这一要求打印。打印功能要求对象转换成字符串以至于该字符串可以输入入输出。myf 唯一的选择是表明该变量中存储的实际参考（本身的地址）。这不是我们想要的。

我们有两种方法可以解决这个问题。一个是定义一个方法称为 show 将允许 Fraction 对象将自己作为一个字符串输出。我们可以实现这个方法，如清单 3 所示。如果我们之前创建的 Fraction 对象，我们可以让它 show（展示）它自己，换句话说，用适当的格式打印本身。不幸的是，这工作一般不可以。为了使打印正常工作，我们需要告诉 Fraction 类如何转换成一个字符串。这就是打印功能需要的以此来工作。

Listing 3

```
def show(self):
    print(self.num,"/",self.den)
>>> myf = Fraction(3,5)
>>> myf.show()
3 / 5
>>> print(myf)
<__main__.Fraction instance at 0x40bce9ac>
>>>
```

在 Python 中，所有的类都有一个标准设立方法，但不一定能正常工作。其中的一个__str__是一种方法来将对象转为字符串。此方法的默认实现是返回我们已经看到的地址的字符串。我们需要做的是为这个方法提供一个“更好”的实现。我们会说这个实现覆盖前一个，或者说它重新定义了方法的行为。

为做到这个，我们简单地定义一个方法，名称为__str__方并给它一个新的实现如清单 4 所示。这个定义并不需要任何其他信息，除了特殊参数 self。反过来，该方法会通过转变每一块的内部状态的数据为一个字符串，然后放置一个字符在字符串之间作为字符串关联的事物。结果字符串将在任何一个 Fraction 对象要求转换它自己为字符串时被返回。注意这个函数使用的各种方法。

Listing 4

```
def __str__(self):
    return str(self.num)+"/"+str(self.den)
>>> myf = Fraction(3,5)
>>> print(myf)
3/5
>>> print("I ate", myf, "of the pizza")
I ate 3/5 of the pizza
>>> myf.__str__()
'3/5'
>>> str(myf)
'3/5'
>>>
```

我们可以为我们的新 Fraction 类装载许多其他的方法。一些最重要的是基本的算术运算。我们希望能够创建两个分数对象然后他们加在一起使用标准的“+”符号。在这一点上，如果我们试图直接相加两个分数，则会得到如下：

```
>>> f1 = Fraction(1,4)
>>> f2 = Fraction(1,2)
>>> f1+f2

Traceback (most recent call last):
  File "<pyshell#173>", line 1, in -toplevel-
    f1+f2
TypeError: unsupported operand type(s) for +:
         'instance' and 'instance'
>>>
```

如果你仔细看看错误，你会看到错误是“+”号运算无法被当成运算符理解。

我们可以通过提供 Fraction 类重写方法解决这个问题。在 Python 中，这种方法被称为__add__，它需要两个参数。第一，self，这个总是需要的，第二是在另一个被操作数的表达。例如，

```
f1.__add__(f2)
```

会要求 Fraction 对象 f1 来将 f2 加到到本身。这可以写为标准的符号格式，f1 + f2。
两部分必须有相同分母来相加。最简单的方法来确保他们具有相同的分母是简单地使用两分母的公倍数作为一个共同的分母，a / b + c / d =ad/ bd + cb / bd =（ad + cb）/ bd 的实现如清单 5 所示。此外，函数返回一个新的分子和分母的分数总和对象。我们可以通过写一个标准的分数算术表达式使用此方法，分配加法结果，然后打印我们需要的结果。

Listing 5

```
def __add__(self,otherfraction):

    newnum = self.num*otherfraction.den + self.den*otherfraction.num
    newden = self.den * otherfraction.den

    return Fraction(newnum,newden)
```

```
>>> f1=Fraction(1,4)
>>> f2=Fraction(1,2)
>>> f3=f1+f2
>>> print(f3)
6/8
>>>
```

方法“__add__”达到了我们的目的，但结果并不完美。6/8 的确是对的，但是它并不是最简分数。最好的答案应该是 3/4。为了保证运算结果为最简分数，我们需要一个辅助函数来识别并化简分数。这个函数需要能够找到分子分母的最大公因数，然后我们就可以将分子分母同时除以最大公因数，得到最简分数答案。

最著名的寻找最大公因数的方法是欧几里得算法，我们将在第 8 节对其做具体讨论。欧几里得算法规定：对于两个整数 m 和 n，如果 n 能整除 m，那么就将 n 除以 m 的结果作为新的 n，如果 n 不能再整除 m，那么最大公因数就是 n 或者 m 被 n 整除的剩余物。在这里简要地给出一个以迭代法实现的示例（请参见*动态代码 1*）。注意这个最大公因数的算法只适用于分母是正数的情况。因为我们可以把负分数的负号归结于分子，所以这种算法还是比较实用的。

```
def gcd(m,n):
    while m%n != 0:
        oldm = m
```

```
        oldn = n

        m = oldn
        n = oldm%oldn
    return n

print gcd(20,10)
```

函数来化简分数式。为了使所求分数为最简分数形式，令分子、分母同时除以它们的最大公约数。如对于 6/8 这个分数，最大公约数是 2。我们分数线上下同时除以 2 就得到了一个“新的分数”，3/4（请参见第 6 节）。

Listing 6

```
def __add__(self,otherfraction):
    newnum = self.num*otherfraction.den + self.den*otherfraction.num
    newden = self.den * otherfraction.den
    common = gcd(newnum,newden)
return Fraction(newnum//common,newden//common)
```

```
>>> f1=Fraction(1,4)
>>> f2=Fraction(1,2)
>>> f3=f1+f2
>>> print(f3)
3/4
>>>
```

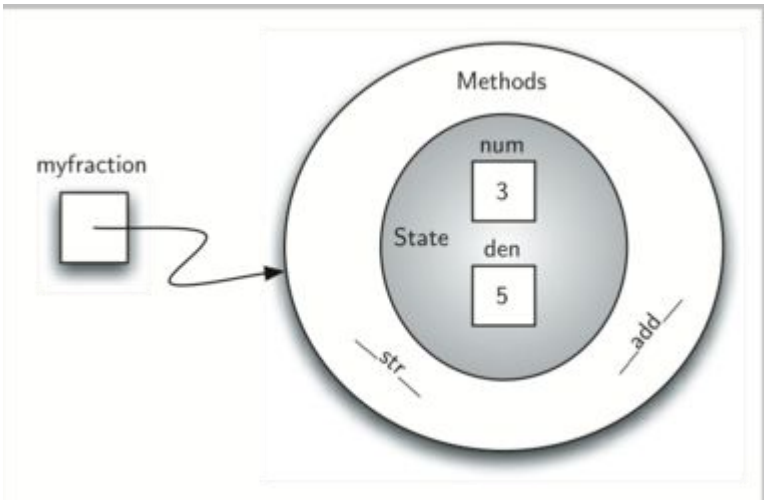

图 3 一个用 2 种方法实现分数类型的例子

对于分数对象，现在有 2 个非常有用的方法，如图 6。我们需要在我们的示例函数类中包含一个可以使两个分数相互比较的方法组。假设我们有两个分数对象，他们分别是 f1 和 f2，仅当 f1 与 f2 指向的是同一对象时 f1==f2 才为真命题，两个不同的对象即使含有相同的分子和分母，在这个实现中也不相等。这个现象被称为浅相等（见图 7）。

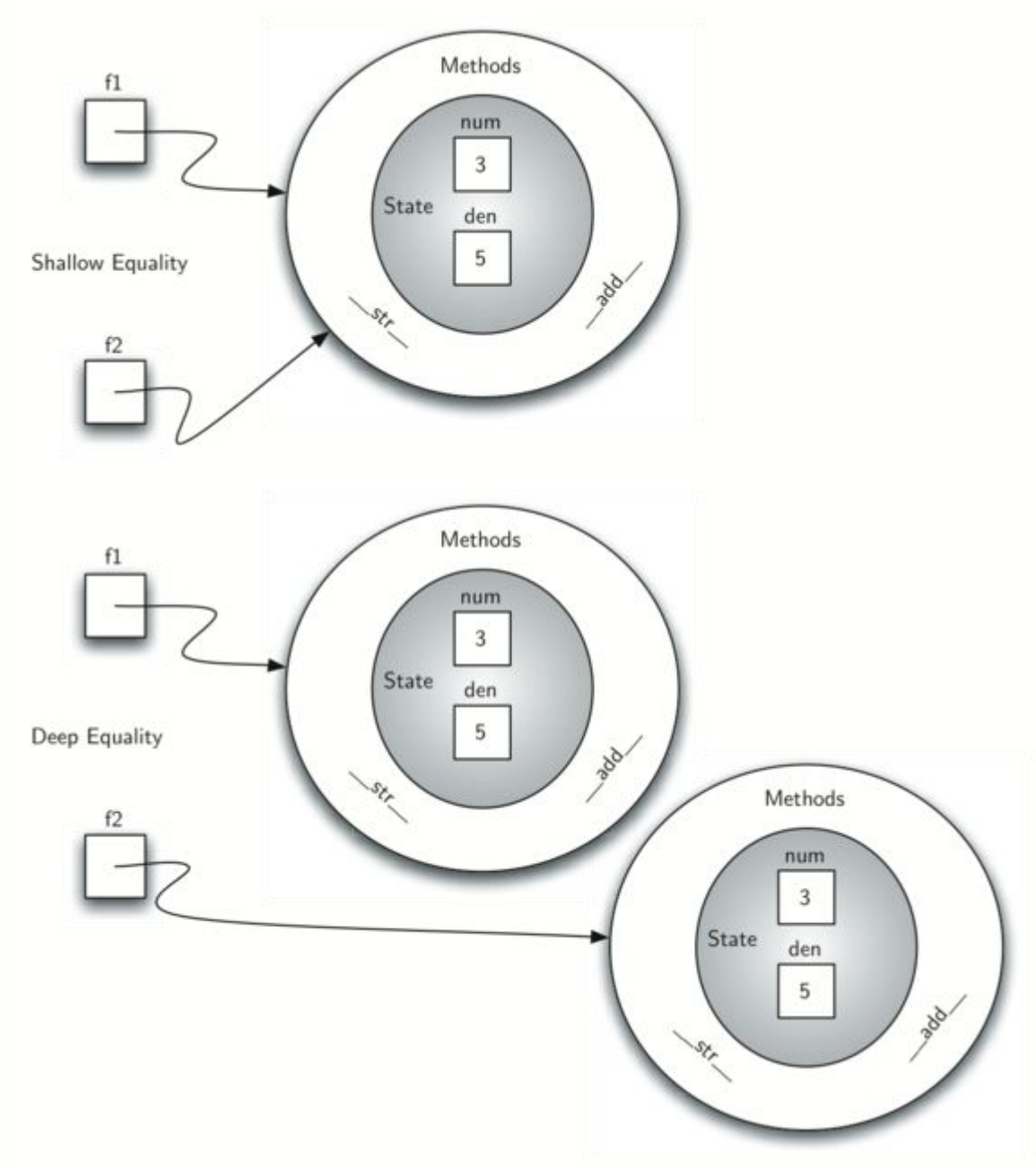

图 4 浅相等和深相等的比较

我们可以通过重建_eq_方法来建立深相等（见图 7）——即一种数值上相等，却并不一定是相同指向的相等方式。_eq_方法是另一种在很多类中均可用的标准方法。_eq_方法比较两个对象，若它们在数值上相等就返回真值，否则就返回假值。

在分数类中，我们通过再一次将两个分数放在同一条件下，再比较他们的分母（通分）的方法实现_eq_方法（见图 7）。需要指出的是，还有其他的相关运算子是可以被重构的。比如，_le_方法可以判定“小于等于”。

Listing 7

```
def __eq__(self, other):
    firstnum = self.num * other.den
    secondnum = other.num * self.den

    return firstnum == secondnum
```

直到这里，完整的分数类已经在*动态代码 2* 中呈现了。剩下的算法和相关方法将作为训练留给读者。

自我测验

为加深理解在 Python 课程中运算符是怎样生效的，以及如何正确地写方法，请写一些包括*，/，-，以及比较运算符>和<的方法。

```

```

代码 6（自我测验 4）

1.7.6.2 继承：逻辑门与门电路

最后一节我们将要介绍面对对象编程的另一个重要方面。继承是一个类联系另一个类的能力，就好像人与人之间能够相互联系,孩子们从父母继承特征。Python 子类能够继承父类的数据和行为特征，类似于孩子们继承父母的特征。这些类通常称为子类和父类。

图 8 显示了内置 python 集合以及它们之间的相互关系我们称这样的一个继承层次结构的关系。例如，列表是有序集合的一个子集，这时，我们称列表为子类，有序集为父类（或者列表亚类、有序超类）。通常称为类的父子继承关系（the list IS-A sequential collection）。这意味着列表从有序集继承重要的特征序列，即基础数据的排序和操作（连接、重复、索引等）。

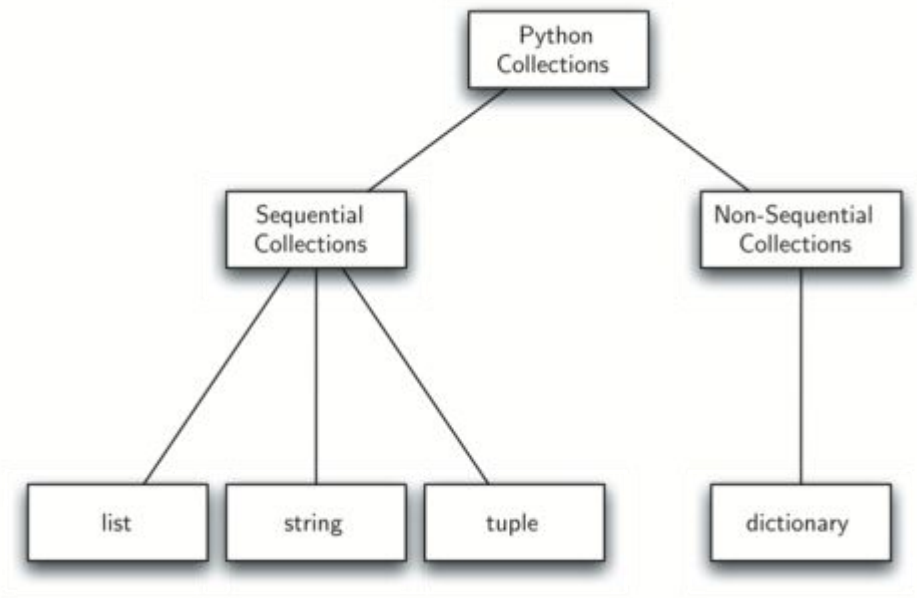

图 8 python 集合的继承层次结构

有序集合的类型包括列表(list)、元组(tuple)和字符串(string)。就像孩子们一方面继承父母的特征，另一方面也通过添加额外的特征来加以区分自己，这三者都继承了共同的数据组织和操作，但是，它们每一个都是独特的，区别在于数据是否均匀、是否不变。

通过这种分层方式来组织类，面对对象的编程语言允许在编写代码之前进行扩展，以满足新情景的需要。此外，用这种分层的方式来组织数据，我们可以更好地了解存在的关系以及更有效地构建抽象表示。

为了进一步探索这种想法，我们将构建一个模拟，用来模拟数字电路的程序。逻辑门(logic gate)将是这个模拟的基本构建模块。这些电子开关代表布尔代数的输入和输出关系。一般来说，门只有一行简单的输出，而输出的值取决于给定的输入值。

与门(AND gate)有两个输入端，每一个都可以是 0 或者 1（代表假或者真）。如果输入值都是 1，则输出是 1。然而，如果输入值有一个是 0 或者两个都是 0，那么输出是 0。

或门(OR gate)同样也有两个输入端，如果输入有一个 1 或者两个都是 1，那么输出 1。只有两个输入都是 0，结果才是 0。

非门(NOT gate)与其它两个门不同的是它只有一个输入端，输出值刚好与输入值相反。如果输入 0，则结果是 1。类似地，输入 1，则结果是 0。

图 9 显示了每一个门的典型代表。每一个门有一个表格来显示相应的门的输入——输出映射。

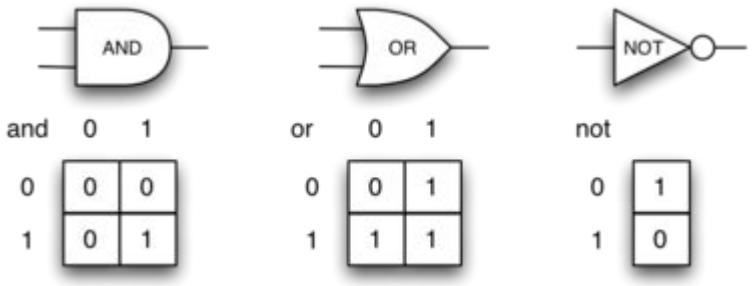

图 9：三种门的典型代表

通过结合这些不同模式的逻辑门，我们可以应用一组输入值来构建具有逻辑功能的电路。图 10 是一个由两个与门、一个或门和一个非门组成的电路。数值从两个与门输出后直接反馈给或门，或门的输出结果再进入非门。如果我们将一组输入值应用到四个输入行（每个与门有两个），数值经过处理后得到的结果就会在非门输出。示例见图 10.

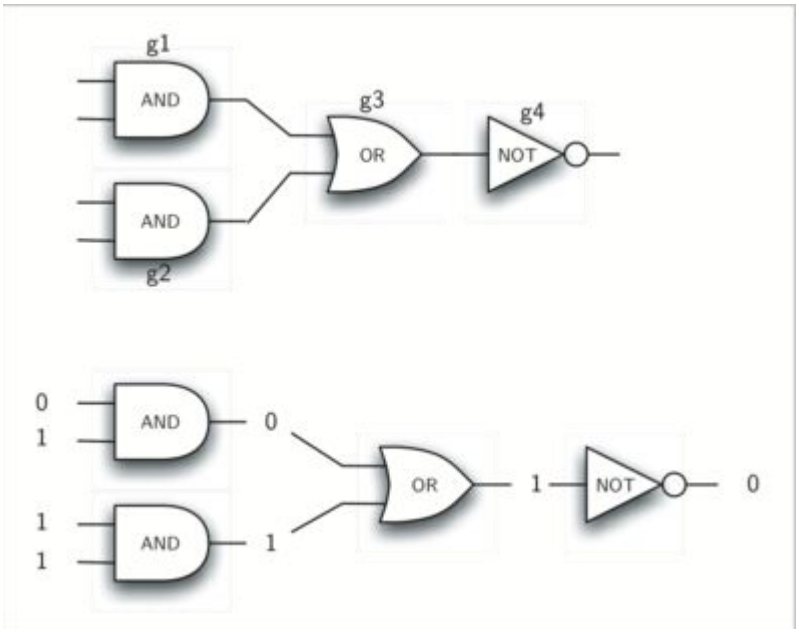

图 10 电路

为了实现一个电路，我们首先要构建一个逻辑门的表示。逻辑门很容易被组织成一个类继承层次结构（见图 11）。在层次结构的顶部，逻辑门类（LogicGate）代表逻辑门的最普遍特征,即有一个标签的门和一个输出端。子类的下一级别把逻辑门分为两个家族，有一个输入端的和有两个输入端的。再往下，每一个门都有特定的逻辑功能。

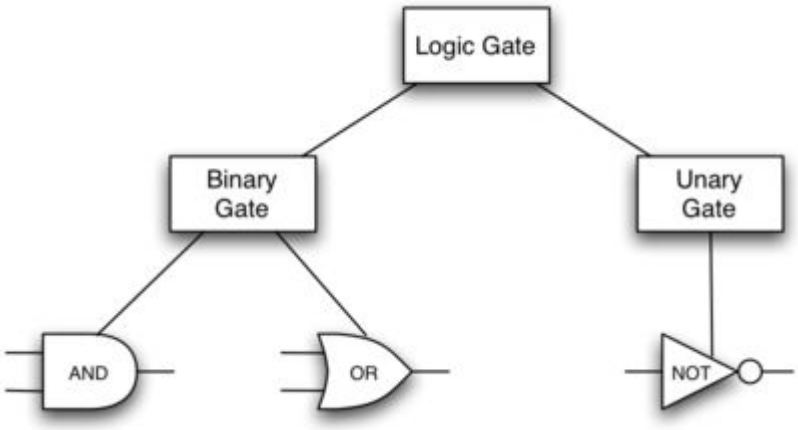

图 11 逻辑门的继承层级结构

我们现在可以开始实现从最一般的类，逻辑门开始。正如之前说的，每一个门都有一个识别标签和一个输出端。此外，我们需要方法允许每一个门的客户从相应的门得到标签。

每一个逻辑门需要的其它行为能力是能够知道它的输出值。这要求门根据当前的输入而执行适当的逻辑。为了生成输出，门需要知道具体的逻辑是什么。这意味着调用一个方法来执行逻辑计算。完整的类见表 8.

Listing 8

```
class LogicGate:

    def __init__(self,n):
        self.label = n
        self.output = None

    def getLabel(self):
        return self.label

    def getOutput(self):
        self.output = self.performGateLogic()
        return self.output
```

这时候，我们将不会实现执行逻辑门(performGateLogic)的功能，因为我们不知道每个逻辑门将会怎样执行它的逻辑操作。这些细节将会包含在每一个被加入到层级结构的门中。这是一个非常强大的面向对象编程思想。我们正在编写一个使用并不存在的代码的方法。参数是指实际的门对象所调用的方法。任何被添加到层次结构的新的逻辑门只需要实现执行逻辑门功能，并且在适当的时候被使用。一旦完成，门就可以提供输出值。这种可以扩展现存等级结构并且为需要使用新类的等级结构提供具体的功能的能力对再利用现存代码来说是非常重要的。

我们以输入端的多少将逻辑门分类。与门和或门有两个输入端，非门有一个输入端。二元门(BinaryGate)类是逻辑门的一个子类，并且增加两个输入端。一元门(UnaryGate)也是逻辑门的子类，

但是只有一个输入端。在计算机电路设计中，这些输入端有时候也称为“针”，因此我们通常也使用这一术语。

Listing 9

```
class BinaryGate(LogicGate):

    def __init__(self,n):
        LogicGate.__init__(self,n)

        self.pinA = None
        self.pinB = None

    def getPinA(self):
        return int(input("Enter Pin A input for gate "+ self.getLabel()+"-->"))

    def getPinB(self):
        return int(input("Enter Pin B input for gate "+ self.getLabel()+"-->"))
```

Listing 10

```
class UnaryGate(LogicGate):

    def __init__(self,n):
        LogicGate.__init__(self,n)

        self.pin = None

    def getPin(self):
        return int(input("Enter Pin input for gate "+ self.getLabel()+"-->"))
```

第 9 节和第 10 节实现了两种类。这些类的构造器都开始于一个对于使用母函数的_init_的母类构造器的精确访问。当创建一个二元门类的示例的时候，我们首先希望初始化所有的从逻辑门继承的数据类型。在这种情况下，即是门的标签。之后，构造器开始加入 2 条输入线（pinA 和 pinB）。这是你在构筑类的阶层时应该经常使用的一个非常常见的模型。子类构造器需要访问母类构造器，然后再转移到它们自己的有区别的数据。

Python 还有一个函数名为 super，这是一个可以用来代替对母类精确命名的函数。这是一个更加普遍的结构，它的应用十分广泛，尤其当一个类含有多于一个的母类。但是我们在这个介绍中不打算讨论它。例如，在我们的示例中，关于 LogicGate.__init__(self,n)可以被 super(UnaryGate,self).__init__(n)替换。

二元门添加的仅有的一个行为就是获取两条输入线中数值的能力。由于这些数值来自一些外部环

境，我们将通过一个向用户提供的输入声明简单地询问他们。相同的实现也发生在除了只有一条输入线之外的一元门中。

由于我们有了针对依赖输入线数量的逻辑门的一个普遍类，我们就可以构筑有独特行为的特殊逻辑门。例如，由于与门（AndGate）有两个输入线，它将成为二元门的一个亚类。与以前一样，构造器的第一条线向上访问它的母类构造器（二元门），相应地，它访问它的母类构造器（逻辑门）。注意，由于与门继承自两条输入线、一条输出线和一个标签，与门类不提供任何新的数据。

Listing 11

```
class AndGate(BinaryGate):

    def __init__(self,n):
        BinaryGate.__init__(self,n)

    def performGateLogic(self):

        a = self.getPinA()
        b = self.getPinB()
        if a==1 and b==1:
            return 1
        else:
            return 0
```

“与”门仅需要添加一个“特定行为”——执行先前描述的布尔运算，是由“执行门逻辑”（performGateLogic）方法进行的。这种方法首先获取两个输入值，然后进行判断，当且仅当两个输入值都为 1 时返回 1。完整的类代码见表 11。

为了显示出“与”门执行的效果，我们可以创建实例，通过观察实例计算并输出的结果来体会“与”门的逻辑。下面这一段代码展示的是一个名为“g1”（“G1”是“g1”的在类内部的标签）的“与”门对象。当我们调用“输出结果”（getOutput）方法时，此对象会首先命令它的“执行门逻辑”（performGateLogic）方法来检查两条输入线，一旦识别到完整的符合要求的输入值，便立刻输出结果。

```
>>> g1 = AndGate("G1")
>>> g1.getOutput()
Enter Pin A input for gate G1-->1
Enter Pin B input for gate G1-->0
0
```

或”门和“非”门的逻辑判断遵循同样的过程。“或”门是二元逻辑门的一个子类，而“非”门则属于一元逻辑门。因为执行判断逻辑是这些类的特征行为，所以它们都需要提供自己的“执行门逻辑”功能。

下面，我们通过建立一个交互式门类的实例并输出结果，来演示单个门类的使用方法：

```
>>> g2 = OrGate("G2")
>>> g2.getOutput()
Enter Pin A input for gate G2-->1
Enter Pin B input for gate G2-->1
1
>>> g2.getOutput()
Enter Pin A input for gate G2-->0
Enter Pin B input for gate G2-->0
0
>>> g3 = NotGate("G3")
>>> g3.getOutput()
Enter Pin input for gate G3-->0
1
```

既然我们已经掌握了基本的门类操作，接下来便可以创建组合逻辑电路了。组合逻辑电路需要把各个门组合连接起来，让一个门的计算结果流入另一个门成为输入值。为了实现这个操作，我们将引入一个新的类，叫做连接器（connector）类。

连接器类并不属于门的层次，而是在其两端分别含有门层（见图 12）。这种关系是面向对象的编程中十分重要的关系，叫做“HAS-A”关系，即非继承关系。在这里复习一下之前学过的“IS-A”关系，也即继承关系，是指子类与父类的相似性，如一元逻辑门继承逻辑门。

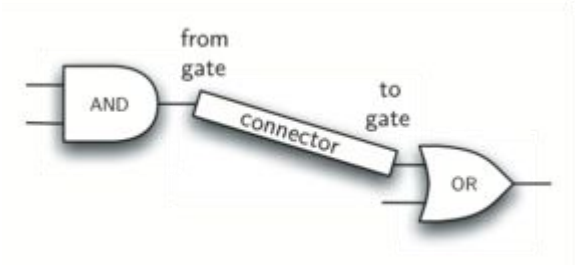

图 12 连接器连接着一个门的输出与另一个门的输入

介绍了连接器类之后，我们得知“组合非继承逻辑门”意味着连接器内部有门类的实例，但却并不属于门的层次。在设计类的时候，分清楚“IS-A”关系（继承关系）与“HAS-A”（非继承关系）是十分重要的。

列表 12 为连接器类。连接器内有两个门类的实例，在每个实例中连接器对象作为 fromgate 或 togate，指示数据值从一个门的输出流入另一个门的输入。设置连接器的时候，“setNextPin”的命令是十分重要的。我们需要把这个方法添加到门类中，以便每一个 togate 都能够选择合适的输入线作为连接。

Listing 12

```
class Connector:

    def __init__(self, fgate, tgate):
        self.fromgate = fgate
        self.togate = tgate

        tgate.setNextPin(self)

    def getFrom(self):
        return self.fromgate

    def getTo(self):
        return self.togate
```

在二元逻辑门中，有些门有两条输入线，而连接器只能连接一条。如果两条线都是可用的，我们默认选择 pinA。如果 pinA 已经被其他连接器连接了，我们便选择 pinB。想要实现一个门的连接，至少要有一条可用的输入线。

Listing 13

```
def setNextPin(self,source):
    if self.pinA == None:
        self.pinA = source
    else:
        if self.pinB == None:
            self.pinB = source
        else:
           raise RuntimeError("Error: NO EMPTY PINS")
```

现在对于输入数据，我们有了两种方法：一种是从外部用户处获取，另一种是从上一个门的输出值中获取。这就要求 getPinA 和 getPinB 方法也进行适当的改进（见列表 14）：如果输入线没有任何可用连接（None），便与原来一样提示用户输入数据；如果有可用连接，便自动执行逻辑过程。这种操作一直重复到所有的输入值都变为可用值并且最后的输出值变为问题中的门中所需要的输入值。总而言之，组合电路逆向执行，找到必需的输入值来得到最后的结果。

Listing 14

```
def getPinA(self):
    if self.pinA == None:
        return input("Enter Pin A input for gate " + self.getName()+"-->")
    else:
```

```
        return self.pinA.getFrom().getOutput()
```

下列代码片段实现了上段所述的组合逻辑电路：

```
>>> g1 = AndGate("G1")
>>> g2 = AndGate("G2")
>>> g3 = OrGate("G3")
>>> g4 = NotGate("G4")
>>> c1 = Connector(g1,g3)
>>> c2 = Connector(g2,g3)
>>> c3 = Connector(g3,g4)
```

两个“与”门（g1 和 g2）的输出结果与“或”门（g3）相连接，而 g3 的输出结果与“非”门（g4）相连。最后，“非”门的输出值就是整个电路的输出值。例如：

```
>>> g4.getOutput()
Pin A input for gate G1-->0
Pin B input for gate G1-->1
Pin A input for gate G2-->1
Pin B input for gate G2-->1
0
```

请用 ActiveCode 4 自己尝试。

```
class LogicGate:

    def __init__(self,n):
        self.name = n
        self.output = None

    def getName(self):
        return self.name

    def getOutput(self):
        self.output = self.performGateLogic()
        return self.output


class BinaryGate(LogicGate):

    def __init__(self,n):
        LogicGate.__init__(self,n)
```

```
        self.pinA = None
        self.pinB = None

    def getPinA(self):
        if self.pinA == None:
            return int(input("Enter Pin A input for gate "+self.getName()+"-->"))
        else:
            return self.pinA.getFrom().getOutput()

    def getPinB(self):
        if self.pinB == None:
            return int(input("Enter Pin B input for gate "+self.getName()+"-->"))
        else:
            return self.pinB.getFrom().getOutput()

    def setNextPin(self,source):
        if self.pinA == None:
            self.pinA = source
        else:
            if self.pinB == None:
                self.pinB = source
            else:
                print("Cannot Connect: NO EMPTY PINS on this gate")


class AndGate(BinaryGate):

    def __init__(self,n):
        BinaryGate.__init__(self,n)

    def performGateLogic(self):

        a = self.getPinA()
        b = self.getPinB()
        if a==1 and b==1:
            return 1
        else:
            return 0

class OrGate(BinaryGate):

    def __init__(self,n):
        BinaryGate.__init__(self,n)
```

```
    def performGateLogic(self):

        a = self.getPinA()
        b = self.getPinB()
        if a ==1 or b==1:
            return 1
        else:
            return 0

class UnaryGate(LogicGate):

    def __init__(self,n):
        LogicGate.__init__(self,n)

        self.pin = None

    def getPin(self):
        if self.pin == None:
            return int(input("Enter Pin input for gate "+self.getName()+"-->"))
        else:
            return self.pin.getFrom().getOutput()

    def setNextPin(self,source):
        if self.pin == None:
            self.pin = source
        else:
            print("Cannot Connect: NO EMPTY PINS on this gate")


class NotGate(UnaryGate):

    def __init__(self,n):
        UnaryGate.__init__(self,n)

    def performGateLogic(self):
        if self.getPin():
            return 0
        else:
            return 1


class Connector:
```

```
    def __init__(self, fgate, tgate):
        self.fromgate = fgate
        self.togate = tgate

        tgate.setNextPin(self)

    def getFrom(self):
        return self.fromgate

    def getTo(self):
        return self.togate


def main():
   g1 = AndGate("G1")
   g2 = AndGate("G2")
   g3 = OrGate("G3")
   g4 = NotGate("G4")
   c1 = Connector(g1,g3)
   c2 = Connector(g2,g3)
   c3 = Connector(g3,g4)
   print(g4.getOutput())

main()
```

代码 13 完整的组合逻辑电路

自我测试

创建两个新的门类：NorGate 和 NandGate。其中，NorGate 相当于“或”门的输出值连接“非”门；NandGate 则为“与”门输出值连接“非”门。

创建一系列门，证明：

NOT ((A and B) or (C and D)) 与 NOT(A and B) and NOT (C and D)等价

注意要在模拟过程中使用一些自己定义的新的门

```

```

代码 7（self_check_5）

1.8 小结

- 计算机科学是研究问题求解的学科；
- 计算机科学采用“抽象”作为表示过程与数据的工具；
- 采用“抽象数据类型”，程序员可以通过隐藏数据细节来控制问题域的复杂度；
- Python 是一个强大的、而又易于使用的面向对象程序设计语言；
- 列表、元组和串，是 Python 内置的有序集类型；
- 字典与集合，是数据的无序集类型；
- 采用“类”，程序员可以具体实现一个抽象数据类型；
- 程序员可以创建新方法，也可以重载已有的标准方法；
- 一个类构造器在处理自身的数据和行为之前，总会调用其父类的构造器。

1.9 关键词

抽象数据类型	抽象化	算法
类	可计算	数据抽象化
数据结构	数据类型	深意义相等
字典	封装	异常
格式运算符	格式化字符串	HAS-A 关系
独立于实现	信息隐藏	继承
继承阶层架构	接口	IS-A 关系
列表	列表解析	方法
可变性	对象	过程抽象化
编程	提示符	自身
浅意义相等	仿真	字符串
子类	超类	真值表

1.10.问题讨论

1. 为在大学校园内的人员构建一个继承阶层架构。这些人员包括教职员工和学生。他们之间有哪些相同点？有哪些区别？
2. 为银行账户构建一个继承阶层架构。
3. 为不同类型的电脑建立一个继承阶层架构。
4. 利用本章提供的类，构建一个交互性回路并进行测试。

1.11.编程练习

1、执行一个简单方法，以返回分数的分子和分母。

2、很多情况下最好分数一开始就能保持最简比。请修改 Fraction 类的构造器，让 GCD 可以用来约分。注意这意味着__add__函数不再需要执行约分的功能。请做必要修改。

3、执行余下的简单代数运算操作(__sub__, __mul__, and __truediv__).

4、执行余下的关系型操作(__gt__, __ge__, __lt__, __le__, and __ne__)

5、修改 fraction 类的构造器以确定分数的分子和分母都是整数，如果有一个不是，则用 raise 语句抛出异常。

6、在分数的定义中我们认为负分数有负分子和负分母。使用负分数可能导致一些关系型操作出错。通常来讲，这是个不必要的限制。请通过修改构造器的方式，使用户略过负数，让各种操作都能正确运行。

7、搜索__radd__方法。它和__add__有什么不同？什么时候需要用到？执行__radd__。

8、思考一下__iadd__方法，过程同上题。

9、搜索__repr__方法。它和__str__有什么不同？什么时候需要用到？执行__repr__。

10、搜索其他逻辑门（如 NAND，NOR 和 XOR）。把它们加入组合逻辑电路中。你需要做多少额外编码？

11、最简单的数字电路就是半加器，请搜索并运行。

12、请扩展上题数字电路并运行一个八比特的全加器。

13、本章展示的模拟电路是逆向执行的。换句话说，给定一个电路，为计算输出值，该电路通过输入值进行回溯，这期间也会用到一些别的输出结果。这个过程一直进行，直到找到需要用户从外界输入的值。请修改运行机制，以使程序正向运行，即电路一接受输入值就开始计算结果。

14、设计一个类来表示一副纸牌。用两个类来运行你最喜欢的纸牌游戏。

15、找一个数独游戏并编程解决。