导论

1.1 目标

- 了解计算机科学，编程以及解决问题的思想
- 了解抽象及其在解决问题过程中的作用
- 理解并实现抽象数据类型的概念
- 回顾Python

1.2 开始

自从第一台需要使用大量线缆和开关来实现人机交互的电子计算机诞生以来，人们对于编程的看法已经经历了巨大的变化。跟社会中很多领域一样，计算科学的发展使得计算机科学家能够使用越来越多的工具。高速处理器、高速网络以及大容量存储器的不断进步导致了复杂度的螺旋式上升，而计算机科学家必须将其解决。在飞速发展期间，也有许多基本原则经久不变。计算科学本身关注的是利用计算机来解决问题。

毫无疑问，你已经花费了大量时间来学习解决问题的基本知识。希望你也能对自己理解问题并给出解决方案的能力有所信心。你应该也有所了解编写计算机程序常常是具有相当难度的。大型问题的复杂性以及由此带来的对应的解决方法的复杂度，往往会掩盖解决问题所需要的基本思路。

本章记下来的部分将要重点强调两个方面的内容。第一，回顾计算机科学以及算法与数据结构的研究必须要满足的基本框架，尤其是我们为何需要学习这些内容以及这些内容将会如何帮助我们更好地解决问题。第二，回顾Python。尽管无法提供详细全面的指导，本章将给出案例并对接下来的章节中将会出现的概念和思路进行解释。

1.3 计算机科学是什么?

一般来说计算机科学很难准确定义。这也许是由于“计算机”这个名字的滥用。也许你已经清楚，计算机科学并不仅仅是对计算机的研究。尽管计算机在该学科中作为工具而起着支撑性作用，然而它也仅仅是 **工具** 而已。

计算机科学是对问题本身、问题的解决、在问题求解过程中得出的解决方案的研究。给定一个问题，以为计算机科学家的目标便是开发一种**算法**，即用来解决该问题可能出现的各种实例的一份详细的指令集清单。算法通过执行有限的过程解决问题。算法便是解决方案。

计算机科学可以被看作是对算法的研究。然而，我们必须考虑到有些问题是无解的。尽管证明此论断超过了本书的范畴，但对于学习计算机科学的人来说，有些问题确实是无解的这一事实是很重要的。通过引入问题的类型，我们便可以完整地定义计算机科学：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

谈及问题和解决方案时，***“可计算（*computable*）”**这个词是很常见的。当用于解决某个问题的算法存在时，我们称该问题为可计算的。于是，另一种对计算机科学的定义为：研究问题是否是可计算的或者说研究算法的存在与不存在。在大多数情况下，你会注意到“计算机”这个词并不会出现。算法被认为是独立于机器的。

计算机科学本身属于问题求解过程，因此也是对 **抽象** 的研究。抽象使得我们可以用类似于将所谓的逻辑对象与物理对象区分开来的方式来处理问题及其解决方案。在下文所述的常见案例中的这种思想对我们来说是很熟悉的。

考虑你今天驾驶的去往学校或公司的汽车。作为司机，也就是汽车的使用者，你拥有一系列按一定顺序发生的交互方式来使得汽车能够按照预定意图运行。为了实现驾驶，你需要坐进汽车，插入钥匙，启动汽车，换挡，刹车，加速，转弯。从抽象的观点来考虑，我们说你看到的是汽车的逻辑面。你使用的是由汽车设计者提供的实现将你从某个地方移动至另一个地方的功能。这些功能有时也被称作 **界面** 。

另一方面，汽车修理工有着大不一样的观点。他不仅知道如何驾驶同时也必须知道实现那些我们认为理所当然的功能所必需的所有细节。他需要知道引擎是如何运行的，换挡器如何实现变速，如何控制温度等等。这被认为是物理角度,即发生“在引擎盖下”的细节。

我们使用计算机时也是一样。大多数人使用计算机来书写文档、发送与接受邮件、浏览网页、播放音乐、存储图像以及玩游戏，但他们并不了解使得这些应用运行所需要的细节知识。他们从逻辑或者说用户的角度来看待计算机。计算机科学家、程序员、技术员以及系统管理员则是从截然不同的角度来看待。他们必须知道操作系统运行的细节，网络协议是如何配置的以及如何编写各种代码来控制这些功能。他们必须能够控制用户仅仅认为是理所当然的那些底层细节。

这两个例子的相同点是：抽象内容的用户，有时也称之为客户，只要了解界面如何使用即可，而不需了解其底层细节。界面是我们作为用户时与具体实现内部的底层复杂代码进行交流的一种方式。另一个例子以Python的math模块为例。只要我们引入了该模块，便可以进行类似于下面的计算：

```
import math(16)

math.sqrt(16)
```

这是一个 **过程抽象（procedural abstraction）** 。我们没必要知道平方根是怎样被计算的，但我们知道这个函数叫什么以及如何使用它。若是正确地进行了导入，便可相信该函数会为我们提供正确的结果。我们知道有人实现了平方根问题的解决方案，但我们仅仅需要了解如何使用它即可。这有时被称为过程的“黑箱”观点。我们仅简单地描述接口：函数的名称，需要提供些什么（参数），以及会返回什么。细节都被隐藏在内部。

1.4 什么是编程

编程 是形成算法并且将其编写为计算机可以执行的表达，即编程语言。尽管存在许多编程语言和各式各样的计算机，最重要的第一步还是获得解决方案。没有算法便没有程序。

计算机科学不是研究编程的。但编程是计算机科学家工作的重要内容。编程常常是解决方案的表达方式。因此，语言表达及其编写方式都成为了该学科的基础部分。

算法以表达某问题实例所需要的数据以及生成预期结果所需要的一系列必要步骤来给出该问题的解决方案。编程语言必须提供符号系统来表达过程和数据。为此，编程语言提供了控制结构和数据类型。

控制结构使得算法步骤能够以简介而准确的方式表达出来。算法要求至少可以实现顺序处理、决策选择、迭代的结构。只要该语言提供了这些基本的语句，便可以用于算法表达。

在计算机中，所有的数据都以二进制数字串表示。为了使得这些数字串具有意义，便有了**数据类型**。数据类型为这些二进制数据提供解释，这样我们便能够以对于求解问题有意义的方式来考虑数据。这些底层的、预置的数据类型（有时被称之为原始数据类型）为开发算法提供了基石。

比如说，许多编程语言都提供了“整型”这种数据类型。计算机内存中的二进制数字串被解释为整数并且给予了我们通常赋予整数所具有的意义（如23654和-19）。此外，一种数据类型还提供了数据项所能参与的操作的描述。以整型为例，加减乘这些操作是很常见的。我们会希望数字类型可以参与到这些算数运算中。

我们常常面临的问题是，问题及其解决方案过于复杂。尽管这些简单的、语言内置的结构和数据类型已经足够表达复杂的解决方案，但却不利于我们求解问题的过程。我们需要办法来控制复杂度并且帮助我们开发解决方案。

1.5 为何要学习数据结构和抽象数据类型?

为了控制程序和问题求解过程的复杂度，计算机科学家利用利用抽象概念使得他们可以专注于“总体框架（big picture）”而无需纠结于细节。通过建立问题域的模型，我们可以使用更好更快的求解过程。这些模型允许我们用根据问题本身，用与之更加匹配的方式，来描述算法所需要操作的数据。

之前，隐藏某个功能的具体底层来使得用户或者客户能够在很高的层次来使用的过程被我们称作过程抽象。现在介绍一种与之类似的概念：**数据抽象**。一个**抽象数据类型**，有时被称作ADT，在不涉及具体实现的情况下，对我们如何看待数据以及许可操作进行了描述。也就是说，我们仅关心该数据表达了什么而不是它最终是如何构造的。有了这种抽象，我们便实现了对数据的封装（encapsulate）。这种理念即是，通过将实现的细节进行封装，使得用户无法接触到它们。这被称为 **信息隐藏（information hiding）** 。

图1.2展示了抽象数据的定义以及它是如何运行的。用户与界面进行交互，执行被抽象数据定义了的操作。抽象数据类型便是与用户进行交互的外壳。具体的实现被隐藏在底层。用户不关注实现的细节。

抽象数据类型的实现，常常被称为 **数据结构（data structure）** 。它需要我们利用编程结构和原始数据类型来构造数据的物理面（physical view）。如前文所述，这两种思考面的分离使得我们可以在不给出如何真正地构造模型的情况下，定义问题所需的复杂模型。这即为数据的 **实现独立（implementation-independent）** 思想。因为存在多种方式去实现某个抽象数据类型，因此实现独立使得程序员可以在不改变用户与数据的交互方式的情况下修改具体实现的细节。使用者可以继续关注于问题求解过程。

1.6 为何要学习算法?

计算机科学家通过经验不断学习。以观摩他人解决问题或者亲自解决问题的方式来学习。接触不同的问题解决技术并且研究不同的算法设计可以帮助我们应对接下来的挑战。通过研究大量不同的算法,我们可以形成模式识别机制，这样下次遇到相似问题时，便可以更好地予以解决。

算法之间往往差异很大。以前文所举的 sqrt 为例，完全有可能存在许多不同的方式来实现计算平方根的函数。某一个算法也许会比其它算法占用更少的资源。而某一个算法也可能需要花费其它算法10倍的实际来返回相同的结果。我们希望找到某种机制来对比不同的算法。即使它们都能运行，也会有某个最优算法。也许会从更简洁的角度来考虑，也有可能从运行更快或是消耗更少内存的角度来考量。在研究算法时，我们可以学会通过对比不同解决方案本身的特点而不是程序或者计算机实现它们的特点。

在糟糕的情况下,我们可能会遇到非常棘手的问题，也就是说没有算法可以在要求的时间内解决问题。将有解问题、无解问题、有解但耗时过高或消耗资源过大的问题区分开来是非常重要的。

论证和取舍算法往往会带来取舍问题。作为计算机科学家，除了解决问题的能力外，也需要掌握问题解决方案评估的技能。最后，一个问题往往有多种解法。寻找解决方案并研究其是否优秀是我们需要不断重复的任务。

1.7 Python基础知识回顾

在本节，我们将学习Python编程语言，并给出前文一些思想的具体案例。如果你刚刚接触Python或者说你需要更多的相关信息，我们建议你查询“Python Language Reference”， “Python Tutorial”等资料。在这里，我们的目的是让你重新熟悉该语言并强化对后续章节的一些核心概念的认识。

Python是一种现代，易学，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结构。因为Python是一种解释型语言，所以可以通过查看并理解交互式会话便可以实现复查。你应该记得解释器会显示你熟悉的“>>>”提示并检查你给出的Python结构。比如：

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

以上代码会显示提示符，print函数，结果，以及下一个提示符。

1.8 从数据开始

我们说过，Python支持面向对象编程模式。这意味着Python把数据当作问题求解过程的重点。在Python中，跟其它面向对象编程语言一样，我们定义类来描述数据的外观（状态）以及其功能（行为）。类可类比于抽象数据类型，因为类的用户仅能接触到数据项的状态和行为。数据项在面向对象设计中被称为对象。对象是类的一个实例。

1.8.1 内置基本数据类型

现在开始讲述Python的基本数据类型。Python内置了两种与数值相关的类，实现了整型和浮点型数据类型，即 *int* 和 *float* 。标准的算术运算（加减乘除及幂）可以通过使用括号来改变执行顺序。其它很有用的运算有求余运算 **%** 和地板除 **//** 。注意两个整数相除，其结果应当是浮点数。然而除法运算符仅会返回商的整数部分，去掉小数部分。

布尔数据类型，通过Python中的 **bool** 类来实现，对于表达真值非常有用。布尔对象的状态值仅有 **True** 和 **False** ，其标准运算符为 **and, or, not** 。

布尔数据类型也用于比较运算符比如说等于（==）和大于（>）得出的结果。此外，关系运算符和逻辑运算符可以组合起来形成复杂的逻辑问题。

标识符在编程语言中起到名称的作用。在Python中，标识符以字母或者一个下划线（_）开头，对大小写敏感，并且可以是任意长度。请注意，使用具有实际意义的名称可以使得代码更容易阅读与理解。

当一个名称第一次被用在赋值语句的左边时，一个Python变量便产生了。赋值语句提供了一种将名称与值关联起来的方法。变量将会保存数据的引用而不是数据本身。参考下面的例子：

```
>>>the_sum = 0
>>>the_sum
0
>>>the_sum = the_sum + 1
>>>the_sum
1
>>>the_sum = True
>>>the_sum
True
```

赋值语句 **the_sum = 0** 创造了一个被称为the_sum的**变量**，并且该变量持有数据对象0的引用。一般来说，赋值语句的右边会进行运算，其结果数据对象的引用将被指派给左边的变量名。在本例中，该变量的类型是整型，因为“the sum”指向的数据类型便是整型。如果该数据的类型发生变化，如上所示变为布尔值True，变量的类型也会发生相应变化。赋值语句更改了该变量所持的引用。这便是Python的动态特性。同一个变量可以指向不同的数据类型。

1.8.2 内置容器数据类型

除了数值和布尔类之外，Python也提供了很多强大的内置容器类。列表，字符串，元组都是在结构上非常相似的有序容器，但它们之间也有些差异需要掌握以正确使用。集合和字典则是无序容器。

列表是包含零或多个指向Python数据类型的有序容器。列表由方括号围起来的、逗号分割的值构成。空列表即[]。列表是非均质性的，也就是说其中的数据对象不必属于同一类，并且列表本身可以赋给变量。下面的代码块演示了列表中数据类型的多样性。

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

请注意，直接向Python中输入列表的话，列表本身将被返回。然而，为了接下来的操作，其引用应该被赋给某个变量。

因为列表是有序的，因此它支持一系列可用于Python序列类的运算符。

注意到列表的切片是从0开始的。切片操作my_list[1:3]，返回的是列表中以索引为1的项开始到索引为3的项形成的另一个列表（不包含索引为3的那一项）。

有时，我们需要初始化一个列表。这可以利用重复运算迅速完成。

```
>>>my_list = [0] * 6
```

与重复运算符相关的非常重要的一点是，其结果是序列中各数据对象的用地址的重复。下面的代码可以很好地解释：

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

变量A持有3分对原列表my_list的引用。注意到my_list中一个元素的变化将会在A中的所有3个引用得到体现。

列表支持大量可用于构造数据结构的方法。

有一些方法，比如说pop，不仅返回值并且也会对列表本身进行修改。其它的一些方法，仅是对列表进行修改而不返回值。pop默认抹除最后一项，也可以对溢出并返回指定的某一项。在使用这些方法时，索引范围同样是从0开始的。你应该也注意到了熟悉的“.”符号，它用来让某个对象调用某个方法。my_list.append(False) 可以读作“让对象my_list执行它的append方法并且向其传递一个值False。即使是简单的数据对象比如说整型也可以这样调用方法。

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

在上述代码中，整型对象54执行了它的add方法（在Python中叫__add__），并且传递了值21用来相加。结果是75.当然，我们一般都写作54 + 21。在本节的后文将进一步讨论这些方法，

range函数是经常与list一起讨论的Python函数。range函数生成表示一系列值的range对象。利用list函数，可以将range对象的值转为列表。如下所示。

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

range对象代表了一组有序整数。默认情况下，它会从0开始。如果你提供了更多参数，它可以从指定位置开始、结束，甚至可以跳过某些项。在我们的第一个例子range(10)中，序列是从0开始直到10，但是并不包含10。在第二个例子中，range(5,10)从5开始但并不包含10。range(5,10,2)，也是类似地，但是它的步长为2(同样地，不包含10)。

字符串是0个或多个字母、数字或者其它符号的有序容器。我们将这些字母、数字、或者其它符号称为字符。字符串通过单引号或者双引号与标识符区分开来。

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

方法名	使用	解释
center	a_string.center(w)	返回w长的字符串，原字符串位于中间
count	a_string.count(item)	返回item在原字符串中出现的次数
ljust	a_string.ljust(w)	返回w长的字符串，原字符串位于左边
lower	a_string.lower()	返回字符串，原字符串所有字母小写
rjust	a_string.rjust(w)	返回w长的字符串，原字符串位于右边
find	a_string.find(item)	返回item第一次出现的索引
split	a_string.split(s_char)	以s_char作为分隔符，将字符串拆分

表1.4 Python中字符串提供的方法

因为字符串是序列，因此前文所述的所有序列操作都可以用于它。此外，字符串也有另外的一些方法，其中的一些如表1.4所示。

其中，split对于处理数据是非常有用的。split会对字符串进行处理，返回一个由字符串构成的数组，这些字符串由分隔符划分开来。如果没有指定分隔符，split方法会以空白字符比如tab、换行、空格作为分隔符。

字符串与列表的主要区别是，列表可以被修改而字符串不可以。这被称为**可变形**。列表是可变的，而字符串是不可变的。比如说，你可以使用索引和赋值语句修改列表中某一项的值。但是对字符串来说，这种操作是不允许的。

元组与列表非常相似，因为它们都是非均质型数据序列。区别是，元组与字符串一样，是不可变的。元组是无法修改的。元组是以圆括号围起、以逗号分隔的一系列值。作为序列，它也可以使用前文描述的那些操作。

然而，如果你试图改变元组中的某一项，马上便会报错。注意到错误信息提供了问题的定位和原因。

集合是零个或多个不可变Python数据类型组成的无序容器。集合不允许进行复制，是用花括号括起来的、逗号分隔的值。空集用set()表示。集合也是非均质型的，容器本身可以赋给变量：

```
>>> {3, 6, "cat", 4.5, False}
{False, 4.5, 3, 6, 'cat'}
>>> my_set = {3, 6, "cat", 4.5, False}
>>> my_set
{False, 3, 4.5, 6, 'cat'}
```

虽然集合是无序的，但是它们也支持一些前文提到过的操作。如表1.5及下面的代码所示。

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

接触过数学意义上的集合的人应该会对Python下的集合所支持的很多方法感到熟悉。表1.6对此进行了总结。注意union、intersection、issubset和difference都有操作符起到相同的作用。

方法名	使用	解释
union	set1.union(set2)	返回集合，它是set1和set2的并集

方法名	使用	解释
intersection	set1.intersection(set2)	返回集合，它是set1和set2的交集
difference	set1.difference(set2)	返回集合，它是set2中有而set1中没有的项
issubs	set1.issubset(set2)	返回布尔值，判断set1是否是set2的子集
add	set.add(item)	将item加入到集合中
remove	set.remove(item)	从集合中将item移除
pop	set.pop()	从集合中移除任意项
clear	set.clear()	移除集合中所有项

表1.6 Python中set类提供的方法

最后一个要介绍的Python无序容器是dictionary（字典）。字典是由多个键、值对组成的无序容器。键-值对一般写作 key: value。字典通常是逗号分隔的，键值对由花括号围起来。

通过键获取值以及增加额外的键-值对，是字典的基本操作。通过键取值的操作与一般的序列非常相似，但字典使用的是键而不是索引。增加新值也是类似的。

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

特别需要注意的是，字典的存储与其键的顺序无关。首先放入的键值对('Utah': 'SaltLakeCity')被放在了字典的开始位置，而第二个加入的键值对（'California': 'Sacramento')被放在了最后。键的位置取决于所谓的哈希算法，在第四章将会进一步探讨。同时可见，len()函数与之前的容器起到了一样的作用。

字典有其方法和操作符。如表1.7和1.8所示，下面的代码也进行了演示。keys, values和items方法都返回包含了对应值的对象。你可以使用list函数将它们转为列表。get函数有两种用法。如果键并不存在于字典中，get返回None。然而，第二个参数（可选），可以指定返回值。

略去表1.7、1.8

1.9 输入与输出

无论是请求数据或是发送数据，与用户进行交互可以说是必需的要求了。当代大多数程序都会以对话框的形式向用户请求某种输入。虽然Python确实可以创建对话框，但我们也可以使用简单得多的函数。Python提供了一个函数，可以向用户请求输入某数据并以字符串的形式返回指向该数据的引用。这个函数被成为input.

Python的input函数接收一个简单的字符串参数。这个字符串一般被成为**提示**，因为它包含了一些有用的信息帮助用户完成输入。比如说，你可以这样使用input：

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

user_name 变量将会存储用户输入的任何类型。input函数可以轻松地提示用户进行输入，并存储起来以待进一步处理。比如说，在下面的语句中，第一句向用户请求输入他们的名字，第二句根据输入的字符串进行简单处理并将结果打印出来。

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

特别需要注意的是，input函数返回的值是字符串，以准确地表达用户在提示后所输入的字符。如果你想把输入的字符串解释为其它类型，你必需显式地提供类型转换。在下面的语句中，输入的字符串被转为float函数，这样以来它便可以用来进行数值运算处理了。

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

1.9.1 格式化字符串

从上文已可看出，print函数提供了一种简单的方式实现从Python程序输出值。print函数接受零个或多个参数并且用单空格将他们分隔开来以显示。可以通过指定sep参数来修改分隔符。此外，每一个print默认或在末尾跟上一个换行符。这也可以通过指定end参数来修改。以下面的代码为例：

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

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

对输出结果的样式进行控制是往往是很有用的。幸运的是，Python提供了另一种办法叫做**格式化字符串**。格式化字符串是一种模版，单词或者空格将保留不变，而占位符用于替代将要插入字符串的变量。比如，下面的一个Python语句：

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

其中包含了单词is和years old，但是name和age会根据变量在执行时的值而变化。使用格式化字符串，前一个语句改写为:

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

这个例子演示了一个新的字符串表达式。% 操作符是称为**格式操作符**的字符串操作符。上面的这个语句左边是模版或者格式化字符串，右边则是用来代入格式化字符串的一系列变量。值得注意的是，右边中变量的个数与左边格式化字符串中 % 的个数是一致的。变量从左到右按顺序代入到格式化字符串中。

我们从另一个角度更仔细地研究下格式化表达式的两边。格式化字符串可能会有一个或者多个需要转换的地方。转化字符告诉操作符将要被用来替代进字符串中的值的类型。在上面这个例子中，%s表示一个字符串，而%d表示一个整型。其它的类型还有i, u, f, e, g, c, 或者%。表1.9对此进行了总结。

除了转化字符外，你也可以在%与转化字符间添加一个格式修饰符。给定字段长度后，格式修饰符可以用来进行左对齐或者右对齐。修饰符也可以用来设定浮点数在一定字段宽度下小数点后的位数。表1.10解释了这些修饰符。

修饰符	例子	描述
数字	%20d	变量值占据20个字符的宽度
-	%-20d	变量值占据20个字符，左对齐
+	%+20d	变量值占据20个字符，右对齐
0	%020d	变量值占据20个字符，空位用0填充
.	%20.2f	变量值占据20个字符，且小数点右侧占2个字符

修饰符	例子	描述
(name)	%(name)d	从提供的字典中以**name**作为键取出相应值代入此处

格式化操作符（%）的右边是由变量组成的容器，这些变量的值将会代入左侧的格式化字符串中。容器可以是元组或者字典。如果容器是元气，变量将会按照位置顺序依次代入。也就是说，元组中的第一个元素对应着格式化字符串中的第一转化字符。如果容器是字典，值会根据他们的键代入。下面这个例子中，所有的转化字符必须使用 (name) 这个修饰符来指定键的名称。

```
>>> 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 cost %10.2f cents"%(item,price))
The   banana costs 24.00 cents
>>> item_dict = {"item":"banana", "cost":24}
>>> print("The %(items)s costs %(cost)7.1f cents"%item_dict)
The banana costs 24.0 cents
>>>
```

除了使用转换符和格式修饰符外，Python中也有一个format方法，它可以与**Formatter**类一起使用实现更复杂的字符串格式化。详情可以查看Python库手册。

1.10 控制结构

从前文便可看出，算法需要两个重要的控制结构：迭代和选择。Python以多种方式对此提供了支持。使用者可根据情况选择最合适的表达语句。

对于迭代，Python地宫了while语句和非常强大的for语句。while语句在循环条件为真的情况下一直重复循环体。例如：

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

Hello, world
Hello, world
```

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

重复输出语句"Hellow, world"5次。while语句中的循环条件在每次循环开始时便进行判断。若判断结果为真，循环语句中的循环体便会执行。由于Python要求的强制缩进格式，while语句的结构是很容易看明白的。

while语句是一种可用于实现很多算法的具有普适性的循环结构。在很多情况下，循环是由符合条件控制的。例如：

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

这段代码中只有条件语句的两部分都满足了，循环体才会执行。

虽然说这种结构在很多情况下都适用，但是另一个迭代结构，for语句，可以与很多Python容器类一起使用。for语句可以用于迭代容器内的元素，只要容器是一个序列。比如说：

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

将变量item的值依次赋为列表[1, 3, 6, 2, 5]中的每个值。然后循环体便开始执行。这可用于任何属于序列的容器（lists, tuples, strings）。

for语句的常见作用是明确迭代之的范围。下面这个语句：

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

```
16
>>>
```

print函数被执行了5次。range函数返回一个代表了0,1,2,3,4这一序列的range对象，并且每个值都会被赋给item，然后将其取平方并输出。

这种迭代结构的另一种非常有用的功能是处理字符串中每一个字符。下面这段代码遍历字符串，并且将每一个字符添加进列表。其结果是一个含有所有单词的字母的列表。

```
word_list = ['cat', 'dog', 'rabbit']
letter_list = []
for a_word in word_list:
        for a_letter in a_word:
                letter_list.append(a_letter)
print(letter_list)
```

选择语句允许程序员发出问题，基于问题的结果执行不同的操作。许多编程语言提供了两种形式的结构：ifelse 和 if。下面是一个简单的使用ifelse语句的二元选择结构：

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

在这个例子中，n指向的对象被用于检查其是否小于0。如果是，输出错误提示信息；如果不是，执行else中的语句，计算平方根。

选择结构，跟其它控制结构一样，可以嵌套起来使用，因此其中一个选择结构可用于决定是否再执行下一个选择。比如说，假设score是一个持有计算机科学考试分数的变量：

```
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')
```

Python中也有单向选择结构，即if语句。

回到列表，有另一种使用迭代和选择结构的方式生成列表。这被称为**列表推导式（list comprehension）**。列表推导式可以轻松地基于某些处理或者选择条件来生成列表。比如说，如果想要生成1-10的完全平方，可以这样来：

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

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

使用列表表达式，可以一步完成：

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

变量x依据for循环结构依次取为1到10的各个数，据每一个x值计算出的x * x值便被添加进了所构造的列表中。列表推导式的通用语法也允许加入选择条件，这样只有满足一定条件的项才会被加入列表中。比如：

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

这个列表推导式构造了一个仅含有1到10中的奇数的完全平方值的列表。支持迭代的任何序列都可以用于列表推导式：

```
>>> [ch.upper() for ch in 'comprehension' if ch not in 'aeiou']
```

1.11 异常处理

在写程序时，常常会有两种类型的错误。第一种，即所谓的语法错误，简单地说就是程序员在语句或者表达的构造上犯错了。比如说，for语句中没有跟上冒号是错误的：

```
>>> for i in range(10)
SyntaxError: invalid syntax
```

在这个例子中，Python解释器发现该指令是无法完成的，因为它不符合Python的语法。语法错误在刚接触编程语言时是比较常见的。

另一种错误，即逻辑错误，指程序能够运行但是返回的结果是错误的。这有可能是底层算法的错误或者对该算法的错误表达。有时候，逻辑错误会产生非常严重的异常，比如说零除、获取列表中并不存在的项。在这种情况下，逻辑错误会导致运行错误，使得程序终止。这种类型的运行错误一般被称为**异常（exceptions）**。

很多时候，初学者把异常当作是导致程序终止的致命运行错误。然而，大多数编程语言都提供了处理这些错误的方法，这样一来，程序员便可以根据需求做一些应对措施。此外，程序员也可以新建指定的异常用于他们认为有必要的场景。

发生异常时，一般的说法是“异常冒出（exception raised）“。可以通过**try**语句对异常进行“处理（handle）”。比如说，考虑下面的代码：要求用户输入整数，然后从数学库中调用平方根函数；如果用户输入大于等于0的值，通过print输出其平方根；如果用户输入负数，平方根函数会报告**ValueError**错误。

```
>>> a_number = int(input("Please enter an integer"))
Please enter an integer -23
>>> print(math.sqrt(a_number))
ValueError: math domain error
>>>
```

通过在try代码块中调用print函数，便可以处理该异常：

```
>>> try:
                print(math.sqrt(a_number))
        except:
                print("Bad Value for square root")
```

```
                print("Using absolute value instead")
                print(math.sqrt(abs(a_number))

Bad Value for square root
Using absolute value instead
4.795831523456
>>>
```

sqrt引起的异常被捕获，然后提示相应的信息给用户并且使用绝对值以保证非负性。这意味着程序并不会终端，并且会向下继续执行。

程序员可以通过**raise**语句构造运行错误。比如说，与直接向平方根函数输入负数不同，可以先检查值然后使自定义的异常冒出。下面的代码片段演示了构造新的**RuntimeError**异常。注意，程序同样会终端，但是导致该中断的是由程序员显式构造的异常。

```
>>> if a_number < 0:
                raise RuntimeError("You can't user a negative number")
        else:
                print(math.sqrt(a_number))
```

除了以上演示的RuntimeError外，还可以使用很多异常。查看Python手册以了解所有可用的异常类型及其定义方法。

1.12 定义函数

之前的过程抽象例子中调用了math模块中被称为**sqrt**的函数来计算平方根。通常来说，可以通过定义函数来隐藏计算细节。定义一个函数需要提供名称、一组参数以及函数体。定义的函数可能会显式地返回值。比如说，下面定义的一个简单函数会返回传入的值的平方：

```
>>> def square(n):
                return n ** 2

>>> square(3)
9
>>> square(square(3))
81
>>>
```

该函数定义包括了函数名，即**square**，以及用圆括号围起来的一组形参。就该函数来说，**n**是唯一的形参，这意味着**square**函数只需要一个数据便可以运行。函数被隐藏起来的细节计算了 n ** 2 的结果并且将其返回。在Python环境中调用它即可进行计算，传入一个实际参数值，在本例中是3。注意，对square函数的调用返回了一个整数，并且该整数可以作为参数传入另一次对square函数的调用。

通过著名的“牛顿法”可以实现自己的平方根函数。对于估算平方根，牛顿法给出了一种收敛于正确值的迭代计算方法。公式：

$new\_guess = \frac{1}{2} * (\frac{old\_guess+n}{old_guess})$

代入一个n值，然后重复地估算其平方根，每一次迭代时将上次的new_guess代为本次迭代的old_guess。这里使用的初值是$\frac{n}{2}$。代码1.1定义了一个函数，它接收1个n值，返回20次迭代后估算的n的平方根。同样地，牛顿法的实现细节被隐藏于函数定义中，用户并不需要知道其实现细节。代码1.1也演示了使用#字符来作为注释标记。同一行内，在#之后的字符都被解释器忽略。

**代码1.1 square_root函数 **

```
def square_root(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

>>> square_root(9)
3.0
>>> square_root(4563)
67.549982194812904
>>>
```

1.13 Python中的面向对象编程：定义类

如前所述，Python是一门面向对象编程语言。到目前为止，我们已使用过了一些内置类来演示数据和控制结构。面向对象编程语言的最强大的特性之一便是，其允许程序员（问题解决者）构造新的类来对解决问题所需要的数据进行建模。

记住，我们使用抽象数据类型来对数据对象的外貌（状态）和行为（方法）作一个逻辑性描述。通过构造实现抽象数据类型的类，程序员不仅可以利用抽象过程的优点也可以给出在程序中实际使用该抽象所必须的细节。我们通过定义新的类来实现抽象数据类型。

Fraction 类

一个常用于演示用户自定义类的具体过程的例子是，构造实现抽象数据类型Fraction。Python已经提供了一些数值类。然而有些时候，创建一种外形像分数的数据对象会更加合理。

分数，比如说$\frac{3}{5}$包含两部分。位于上面的数字即为分子，可以是任何整数；下面的数字为分母，应是大于0的整数（负分数被认为其分子为负）。虽然可以用浮点数来近似分数，但现在我们尝试将分数表示为精确值。

Fraction类的操作应使得**Fraction**数据对象跟其它的数字类的运算类似，即实现加、减、乘、除。这些分数最好也能以“斜杠”的方式来显示，比如说**3/5**。此外，**Fraction**类的所有方法都应返回最简形式的计算结果，这样以来，无论进行何种计算，得到的结果都是最通用的形式。

在Python中，与函数定义的语法类似，通过提供类名以及其支持的方法，便可定义类。比如说：

```
class Fraction:
        pass
        # the methods go here
```

这便是定义方法的框架。所有类都应该提供的第一个方法是**构造器（constructor）**。构造器定义了数据对象创建的方式。创建一个**Fraction**类需要提供两个数据，即分子和分母。在Python中，构造器即为__init__（init前后都跟了两个下划线）。如代码1.2所示。

代码1.2：Fraction类及其构造器

```
class Fraction:
        def __init__(self,top,bottom):
                self.num = top
                self.den = bottom
```

注意，形参列表有3项 **(self,top,bottom)** 。**self**是一个特殊的参数，用于指向该对象本身，必须放在形参的第一个，但是在调用时不会给出实际值。如前所述，分数有两个状态数据，即分子和分母。构造器中**self.num**这种写法定义了**Fraction**内部的数据对象**num**作为它的状态参数之一。类似的，**self.den**定义了分母。在初始化时，两个形参便被赋给了状态参数，这样新的分数对象便确定了其初始值。

通过调用构造器便可构造**Fraction**类的实例，即使用类名并传入必要的状态参数（注意并不会直接使用invoke __ init __)。比如说，

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

这样便构造了一个名为**my_fraction**，代表分数$\frac{3}{5}$的对象。

接下来需要做的是实现抽象数据要求的行为。首先，考虑当试图打印**Fraction**对象时会发生什么。

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

Fraction对象my_f并不知道如何处理打印请求。**print**函数要求对象将自己转化为对象以输出到屏幕。在目前这种情况下，**my_f**唯一的选择只有显示变量中存储的实际引用（也就是内存地址本身）。这并不是我们想要的。

有两种办法可以解决这个问题。一是定义名为**show**的方法，使得**Fraction**类可以将其本身作为字符串打印出来，如列表1.3所示。可以按之前的方式来构造**Fraction**对象，然后命令它将自己显示出来，即以合适的格式将其本身打印出来。不幸的事，这种办法并不通用。为了正常打印，必须给定**Fraction**类如何将其本身转化为字符串。

代码1.3：Show函数

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

Python中，所有的类都有一组预定义的方法，但是它们不一定能正常运行。其中之一便是__str__ ,它将对象转化为字符串。由前文可见，该方法的默认实现是返回实例的内存地址。需要为此方法提供更好的实现，这被称为重写或者说重新定义了该方法的行为。

为此，定义一个名为__str__的方法并给出新的实现即可，如代码1.4。该定义除了特殊参数**self**以外不需要其它任何信息。将对象的每一个内部数据转换为字符串，然后利用字符串连接将它们合并在一起，并且在它们之间加上一个“/"字符，便得到了最终的字符串。当**Fraction**对象被要求将它本身转化为字符串时，便会返回这个最终的字符串。注意该函数有多种使用方法。

代码1.4:标准方法

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

```
>>> my_f.__str__()
'3/5'
>>> str(my_f)
'3/5'
```

也可以重写**Fraction**类的其它方法。其中最重要的一些便是基础算术运算。对于两个**Fraction**对象，应当可以通过标准的“+”记号实现它们之间的想家。就当前的状态来说，如果试图将两个**Fraction**对象相加，会得到下面的结果：

```
>>> f1 = Fraction(1,4)
>>> f2 = Fration(1,2)
>>> f1 + f2
Traceback (most recent call last):
File "<pyshell#26>", line 1, in <module> f1+ f2
TypeError: unsupported operand type(s) for +: 'Fraction' and 'Fraction'
>>>
```

仔细查看该错误，便可发现问题是"+"运算符不支持**Fraction**运算对象。为**Fraction**类提供重写加法运算的方法即可解决该问题；在Python中，这个方法名为__add__。它需要两个参数，第一个是必需的**self**，第二个代表了表达式中的另一个运算对象。比如说：

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

这段代码会让**Fraction**对象**f1**将**f2**与**f1**本身相加。这也可以用标准的记法来写：**f1 + f2**。

为了相加，两个**Fraction**对象必须要有相同的分母。最简单的确保它们分母相同的方法是将两个**Fraction**对象的分母之积作为共同的分母，即
$\frac{a}{b} + \frac{c}{d} =]fracadbd + \frac{cb}{bd} = \frac{ad+cb}{bd}$
其实现如代码1.5所示。加法函数返回一个新的**Fraction**对象，其分子和分母即为求和的结果。该方法可用于含有分数、加法运算、赋值、打印操作的标准的算术表达式。
代码1.5:Fraction对象相加

```
def __add__(self, other_fraction):
        new_num = self.num + other_fraction.den + self.den +
other_fraction.num
new_den = self.den + other_fraction.den
>>> f1 = Fraction(1,4)
>>> f2 = Fraction(1,2)
```

```
>>> f3 = f1 + f2
>>> print(f3)
6/8
>>>
```

加法运算如期望运行了，但还有一点可以完善。注意$\frac{6}{8}$虽然是($\frac{1}{4}$ + $\frac{1}{2}$)的正确结果，但却并不是最简形式即$\frac{3}{4}$。为保证结果始终是最简形式，需要提供一个函数来对分数进行最简化，且这个函数本身需要找出最大公约数（GCD）。分子和分母同时除以该最大公约数即可获得最简形式的结果。

Euclid算法是寻找最大公约数的最佳算法。Euclid算法思想如下：对于两个整数m和n，如果n能整除m，则m和n的最大公约数就是n；反之，则为n本身与m被n除所得余数这两个数之间的最大公约数。此处提供一种迭代实现。注意，该GCD算法只在分母为正的情况下可用。对于本章定义的**Fraction**类是合理的，因为负分数被规定为分子为负。

```
def gcd(m,n)
        while m % n != 0:
                old_m = m
                old_n = n
                m = old_n
                n = old_m % old_n
        return n

print(gcd(20,10))
```

现在可以利用此函数对任意**Fraction**类对象进行化简。为了最简化分数，需要将其分子和分母同时除以它们的最大公约数。因此，对于分数$\frac{6}{8}$，其最大公约数是2，得到的结果是新的分数$\frac{3}{4}$，如代码1.6所示。

代码1.6:最简化分数

```
def __add__(self, other_fraction):
        new_num = self.num * other_fraction.den + self.den *
other_fraction.num
        new_den = self.den * other_fraction.den
        common = gcd(new_num, new_den)
        return Fraction(new_num // common, new_den // common)
```

Fraction对象现在已经有了两个很有用的方法。接下来还需要添加的方法包括分数之间的大小比较。给定2个**Fraction**对象，**f1**和**f2**，**f1** == **f2** 只有当它们的引用指向同一个对象时才成立。2个具有相同的

分子和分母的**Fraction**对象在当前这种实现方式下并不相等。这种模式叫**浅相等（shallow equality）**。

可以通过重写__eq__方法来构造**深相等（deep equality）**，即值相等，而不是一定要求引用相等。__eq__方法是所有类共有的标准方法之一。__eq__方法将两个对象进行对比，若值相等则返回**True**，反之则为**False**。

在**Fraction**类中，可以如此重写__eq__方法：将两个分数写为分母相同的形式，然后比较它们的分子，如代码1.7所示。值得强调的是，还有其它可以重写的方法。比如说，__le__方法提供了小于或等于的功能。

代码1.7:判断两个 Fraction**对象是否相等**

```
def __eq__(self, other):
        first_num = self. num * other.den
        second_num = other.num * self.den

        return first_num == second_num
```

到目前为止的完整的**Fraction**类如下所示。剩下的算术和关系方法就作为练习。

代码1.7: Fraction 类

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

        m = oldn
        n = oldm%oldn
    return n

class Fraction:
     def __init__(self,top,bottom):
         self.num = top
         self.den = bottom

     def __str__(self):
         return str(self.num)+"/"+str(self.den)

     def show(self):
         print(self.num,"/",self.den)
```

```
    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)

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

        return firstnum == secondnum

x = Fraction(1,2)
y = Fraction(2,3)
print(x+y)
print(x == y)
```

1.13.2 继承：逻辑门和电路

最后一节介绍面向对象编程的另一个重点。**继承**是类与类之间建立联系的能力，这很像人与人之间的方式：孩子们从父母那里继承特征。类似地，Python中的**子类（subclass）**可以从**父类（superclass）**继承数据和行为。

图1.8显示了Python的内置容器以及它们彼此之间的关系。这种关系结构被称为**继承层次**：**list**是**Sequential Collections**的子类。此类关系常被描述为**IS-A关系（list IS-A Sequential Collection）**。这意味着**list**继承了**sequences**的特征，即数据的有序性和连接、重复、索引等操作。

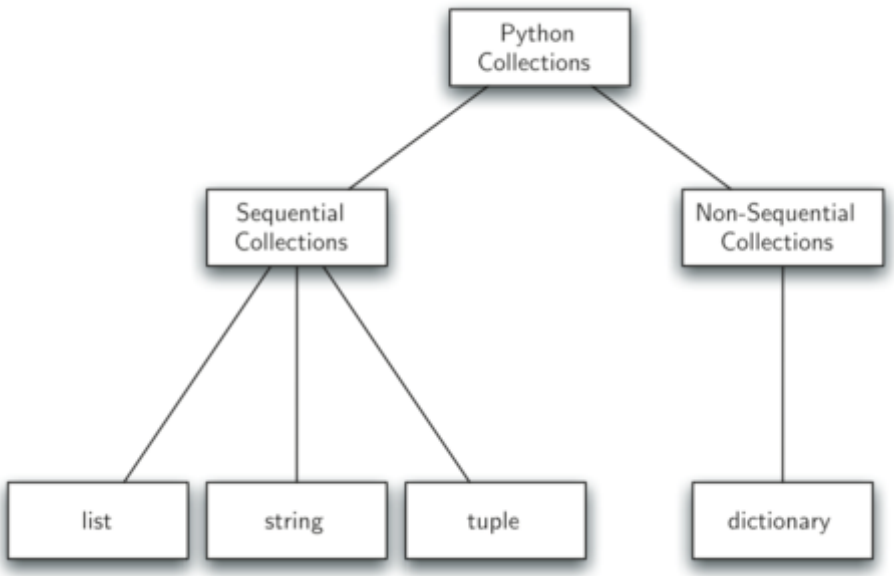


图1.8

列表，元组和字符串都属于有序容器。它们继承了共用的数据组织方式和操作。然而，每一个都根据数据均质性和可变形相互区别开来。子类从父类继承特征并且通过增加额外的特征使得其与父类区别开来。

通过以继承的形式来组织类，面向对象编程语言可以将已有代码根据需求扩展，并且也有助于对已有关系的理解。据此可以提高构建抽象表示的效率。

为了进一步探索这种思想，本节将演示构造一个**simulation**类，用于模拟数字电路。此模拟的基本模块是逻辑门（logic gate）。这些电位开关代表了布尔代数的输入与输出。一般来说，门只有一个输出端。输出的值取决于输入端的值。

与门（AND gate）有两个输入端，每一个端口的值都为1或者0（分别代表了True和False）。如果两个输入端均为1，则输出端结果为1；然而只要有一个输入端值为0，则结果为0。

或门（OR gate）也有两个输入端，若输入值含有一个或两个1，则输出结果为1。在本例中，输入端均为0，故结果为0。

非门（NOT gate）与前两个门不同在于，它只有一个输入端，输出值就是输入值的反面。若输入值为0，则输出值为1。类似地，输入为1则输出为0。图1.9演示了这些门的典例。每个门都有对应的**真值表**来表示该门中输入-输出的映射方式。

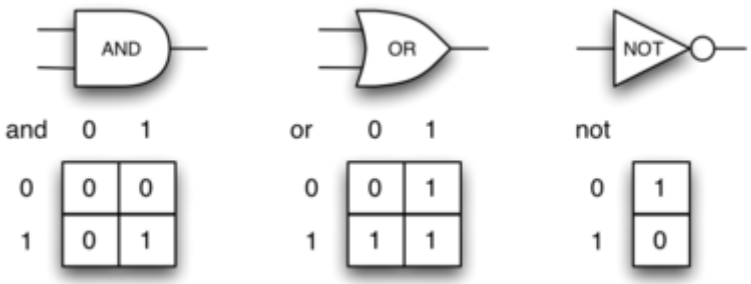
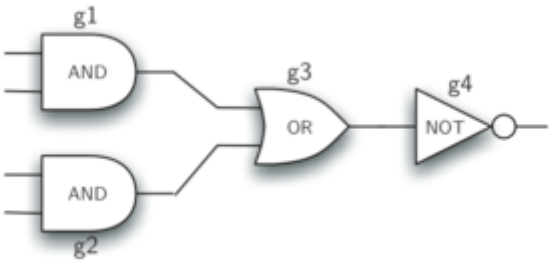
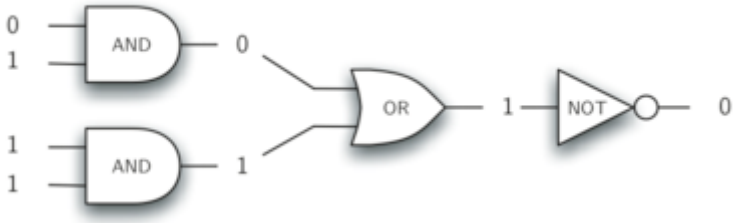


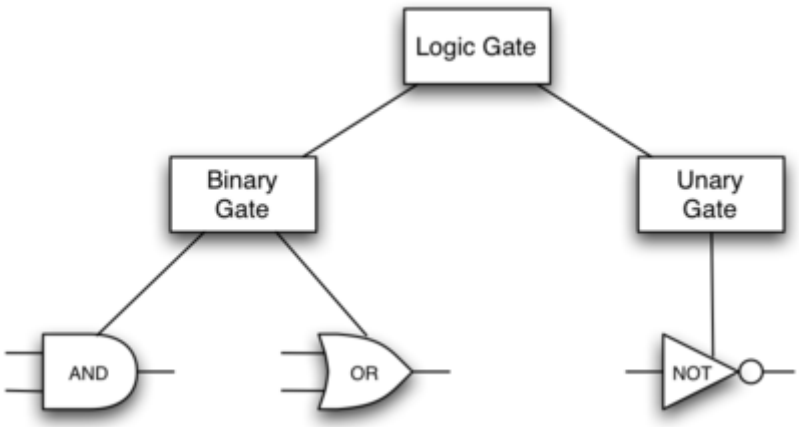
图1.9

将这些门以不同的模式组合并且输入不同的值，便可以建立具有逻辑功能的电路。图1.10演示了含有两个与门、一个或门和一个非门的电路。两个与门的输出端直接连向或门，然后或门的输出结果又输入非门。给定一组输入值（共四个，每个与门两个输入值），经过处理后，最终会在非门输出结果。图1.10也演示具体的例子。





为了实现电路，首先需要为逻辑门建立个示意图。如1.11所示，逻辑门很容易以继承层次的形式来组织。在层次结构的顶部，**LogicGate**类提供了逻辑门最通用的特征，即一个门的标签和输出端。下一级子类将逻辑门分为两类，即有两个输入端和一个输入端的。再往下走，每一个类开始实现具体的逻辑功能。



现在开始具体的实现，首先是最通用的**LogicGate**。前文已提到，每个逻辑门都有唯一一个输出端和一个标签用于识别。此外，需要提供方法使得用户可以查询门的标签。

每一个还必需的一个行为是获取其输出值。那么要求逻辑门根据当前的输入值进行正确的逻辑运算，而为了生成结果，逻辑门还应知道选择正确的逻辑运算，即调用方法进行正确的逻辑运算。代码1.4演示了完整的**LogicGate**类。

代码1.8

```
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
```

此处还不会实现**performGateLogic**方法，理由是目前还不知道逻辑门将会怎样各逻辑门如何运行其逻辑操作。这些细节将会隐藏于每个被加入继承层次结构的逻辑门。这是一种非常强大的面向对象编程思想。当前正在编写的方法的具体代码实际上还并不存在。参数**self**指向当前调用该方法的对象。被加入继承层次的新的逻辑门仅需要实现**performGateLogic**函数，便可正确调用。运行完成，逻辑门便会给出其输出值。