Python学习笔记

E-mail：pidaqing@0335.net

前言

Python是一种新兴的计算机程序语言，是自由软件运动的丰硕成果。

Python是一种免费的、解释型的、可移植的、开放源代码的脚本编程语言。它提供了大量的模块，不仅加快了程序开发速度和代码的清晰程度，而且使程序员专注于要解决的问题，不会陷入繁琐的技术细节。它可以用来开发各种应用程序，从简单的脚本任务到复杂的、面向对象的应用程序。

Python是一种脚本语言，它的语法表达优美易读。它具有很多优秀的脚本语言的特点：

- 解释的；
- 面向对象的；
- 内建的高级数据结构；
- 支持模块和包；
- 支持多种平台；
- 可扩展。

它非常适合于教学。在学习Python的过程中，它可以使学生专注计算机程序语言的基本概念，着中理解现代程序语言的精髓，而不必理会那些细枝末节、令人头痛技术细节，这些细节问题不但难于讲解，也很难理解，而且还会使学生产生厌烦情绪，无助于计算机程序语言的学习。

我在学习《How to Think Like a Computer Scientist》的过程中，发现这本书很适合初学编程语言者，而且深深被Python迷住了，因此就整理出这本笔记，供大家参考。希望诸位多提意见，逐渐完善它，最终使它成为“指南”之类参考书，不再是笔记。

I love Python!

目录

第一章

程序

1.1 程序

程序是根据语言提供的指令，按照一定的逻辑顺序，对获得的数据进行运算，并将结果最终返回给我们的指令和数据的组合。在这里运算的含义是广泛的，既包括数学计算之类的操作，比如加减乘除；也包括诸如寻找和替换字符串之类的操作。数据也依据需要的不同，组成不同的形式，处理后的数据，也可能以另一种方式体现。

程序是用语言写成的。语言分高级语言和低级语言。低级语言，有时叫做机器语言或汇编语言。计算机真正“认识”并能够执行的代码，在我们看来是一串0和1组成的二进制数字，这些数字代表指令和数据。想一想早期的计算机科学家就是用这些枯燥乏味的数字编程，其严谨的治学精神令人钦佩。低级语言的出现则是计算机程序语言的一大进步，它用英文单词或单词的缩写代表计算机执行的指令，使编程的效率和程序的可读性都有了较大的提高，但由于它仍然和机器硬件关联紧密，依然不符合人类的语言和思维习惯，而且要想把低级语言写的程序移植到其他平台，很不幸，必须重写。

高级语言的出现是程序语言发展的必然结果，也是计算机语言向人类的自然语言和思维方式逐步靠近和模拟的结果。这一过程现今仍在继续，将来也不会停止。针对不同领域的应用情况，未来会出现更多新的计算机语言。言归正传，高级语言是人类逻辑思维的程序化、数字化和精确化数学描述。逻辑思维是人类思维方式的重要的一部分，但决不是全部，只有这部分计算机能够比较全面、系统地模拟人类的思维方法。由于高级语言是对人类逻辑思维地描述，用它写程序你会感到比较自然，读起来也比较容易，因此，如今的大部分程序都是用高级语言写的。

高级语言的设计的目的是让程序按照人类的思维和语言习惯书写，它是面向人的，不是面向机器的。我们用着方便，但机器却无法读懂它，更谈不上运行了。所以，用高级语言写的程序，必须经过“翻译”程序的处

理，将其转换成机器可执行的代码，才能运行在计算机上。如果想把它移植到别的平台上，只需在它的基础上，做少量更改，就可以了。

高级语言翻译成机器代码有两种方法：解释和编译。

解释型语言是边读源程序边执行。(见图1.1)

图 1.1: 解释型语言的执行方式

而编译型语言则是将源代码编译成目标代码后执行。以后在执行时就不需要编译了。(见图1.2)

图 1.2: 编译型语言的执行方式

1.2 什么是调试

程序是由人写成的，所以难免出现错误。跟踪并改正错误的过程叫调试。

程序中可能有三种类型的错误：

- 语法错误（syntax errors）；
- 运行错误（runtime errors）；
- 语义错误（semantic errors）。

语法错误 程序要运行，首先语句的语法必须正确，才能够被计算机执行。否则，执行的过程中断，返回错误信息。语法指的是程序语句的组成要符合语言规定的构成规则。例如，下面的语句是符合语法规则的：

```
>>> a = b - c
```

它的意思是将b减c的结果赋值给变量a。如果你把这个表达式写成“b - c = a”这种形式，就错了，因为它不合语法规定，“b - c”这样的表达式不能被赋值。当然语法也不是凭空而定的，它要符合我们的思维习惯。

对于自然语言来说，比如说汉语，你写的文章或说的话，存在少量的语法错误，还不至于影响要表达的意思。而计算机则没有达到如此聪明的地步，它要求百分之百的精确。你的程序要完全符合计算机的语法，哪怕有一点错误，它也不可能执行你的程序。其实语法错误还是比较容易找到和消除的。当你开始学习编程时，由于需要逐渐熟悉语法，出现的语法错误可能会很多，随着经验的增长，它会一点点的减少，即使有，你也能够轻松的找到并改正它们。

运行错误 即使是完美无缺的程序，在运行的过程中也会出现错误，有时称为**异常**，或曰不可预料的错误。有人说，计算机不是善于精确计算吗？不错，确实如此。错的不是计算机，而是我们人类。计算机说到底，不过是人类设计的，为我们所用的工具。它和电视机、汽车从本质上来说，是一样的。限于现在有关计算机软硬件的理论水平、工业制造水平、甚至使用者的水平等等一些内在的、外在的因素，你想，它能够不出错吗？程序越复杂，出现异常的几率越大。异常的种类很多，比如内存用尽，除数为零的除法，都可能导致异常。Python中设计了专门的异常处理语句，把错误的影响降至最低。

语义错误 程序即使有语义错误，也能正常运行，也不会产生任何错误信息，但得到的结果和我们预料的大相径庭。这时候程序做了一些别的事情。发生这种错误一般是我们对语句的运行机理了解的不够透彻，自以为它应该如此运行，而实际却不是这样。还有可能是你的解决问题的思路本身就是错的，写的程序当然是错的。查找这样的错误很不容易，需要从结果进行推理，看一看是程序的哪一部分导致了这样的结果。

1.3 调试

程序错误是不可避免的，查找并改正错误，即调试，就成了我们一项中的我们要掌握的一项重要技能。调试是令人讨厌的工作，常常是你花了一天的时间搜寻错误，也可能还是找不到。因此调试需要细心、耐心和恒心。

任何讲解语言的文章和书籍，都不能把所有的知识告诉你，而调试是深入了解一种语言特性的好机会。通过调试，我们不仅能够提高驾驭语言的能力，而且还丰富了编程知识。

调试在某些方面很想破案，面对很少的、凌乱的线索，你必须推测程序实际的执行过程，猜测是什么地方可能导致了错误。

调试是程序员的工作，其目的是使程序按照预定的功能正常运行。但这时的程序还没有最终完成，必须在进行测试。测试则是由另一部分人，他们的目的就是寻找运行程序出现的错误，然后反馈给程序员，由程序员修复错误。这是一个互动的过程。

1.4 程序语言和自然语言

自然语言是人们日常生活中用于交流的语言，如汉语。自然语言是伴随着人类的成生而形成的，它是人和人之间交流的工具。

程序语言是人类根据自然语言的一小部分，给计算机设计的，用于人和计算机进行交流的语言。在进一步说，程序语言是对计算机硬件资源有计划、合理的分配和利用。计算机按照程序步骤，分毫不差地调用自己的硬件资源进行运算，之后把工作结果提交给我们。从这方面说，程序应该是工作流程，而非语言。

程序语言和自然语言存在很多区别：

模糊性：自然语言充满了模糊性。例如我们写的抒情散文，常常是“醉翁之意，不在酒”。但别人依然能够根据上下文的联系，在整体上把握你要说的意思。程序语言则被要求语句的意思必须明确，不能有任何歧义，更不能联系上下文来确定语句的意思。所以说计算机是一根筋的家伙。

冗余：由于自然语言的模糊性，单凭一句话你不能理解作者的真实想法，为此，作者就需要从方方面面对他的真实意图，进行解说，以帮助人们理解正确的意思，结果导致冗余的出现。程序语言很少会出现冗余，因而意思表达德更精确。

无修饰：自然语言为了追求感情和修辞上的效果，会在话语中添加许多华丽的词汇，以使感情更加充沛。程序语言则没有这些东西，他一就是一，二就是二，不涉及自然语言的感情和修辞。

1.5 第一个程序

Python是解释型语言。我们可以通过命令行或脚本模式执行Python程序。

在DOS提示符下，输入**Python**命令，就进入命令行模，这时解释器出现欢迎信息、版本号及版权说明，然后实**Python**的提示符“¿¿¿”。如果程序不只一行，那么在第二行就会出现第二个提示符“...”。

```
>>> if age>18:
...     print "Your age is more than 18."
... Your age is more than 18.
```

我们也可以启动**python**的IDLE，它和命令行模式很相像，而且功能似乎更强大。IDLE的启动界面（见图1.3）：

我们也可以把语句存入脚本文件，然后在命令行执行它。例如:

```
C:\>python hello.py
Hello,world!
```

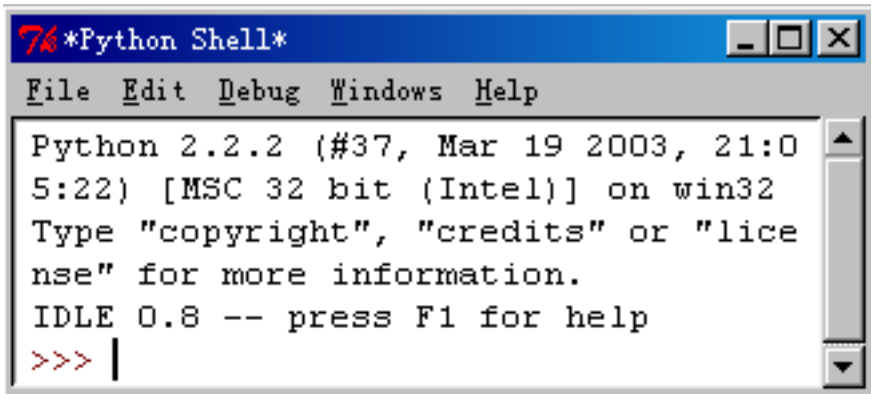

图 1.3: IDLE界面

依照惯例，第一个程序是“Hello，World！”。它所做的就是显示两个单词，“Hello，World！”。用Python语言写的程序如下：

```
print "Hello,World!"
```

这是打印语句的例子，“打印”的意思不是真正打印在纸上，而是显示在屏幕上。程序中的引号表示的是值的开始和结束，它不出现在结果中。

程序的输出结果如下（见图1.4）：

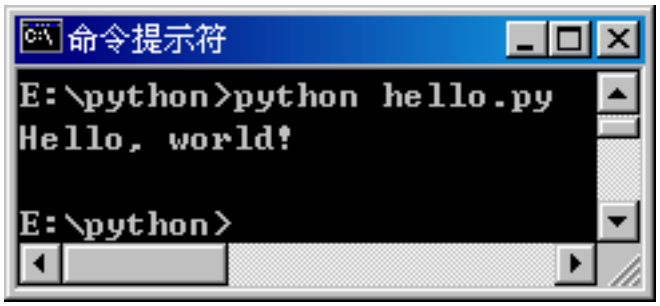

图 1.4: 第一个程序的输出结果

第二章

变量，表达式和语句

2.1 变量和类型

变量是指向各种类型值的名字，以后再用到某个值时，直接引用这个名字即可，不用在写具体的值。在**Python**中，变量的使用环境非常宽松。没有明显的变量声明，而且类型不是固定的。你可以把一个整数赋值给变量，如果觉得不合适，把字符串赋值给它完全可以。

```
>>> x = 100
>>> print x
100
>>> x = "China"
>>> print x
China
```

在别的语言中这是不允许的，若把字符串赋值给整数变量是错误的。我觉得这样做是合理的，难道书柜只能放书，搁别的东西就不行吗？

字符串必须以引号标记开始，并以之标记结束。

如果你不能确定变量或数据的类型，就用解释器内置的函数type确认。如下:

```
>>> type("Hello,World!")
(type 'str')
>>> type(17)
(type 'int')
>>> x = "QHD"
>>> type(x)
<type 'str'>
```

“Hello，World！”属于字符串类型，变量x也是字符串类型，17属于整数类型。

带有小数点的数字叫做浮点数。检查3.0是否为浮点数。如下:

```
>>> type(3.0)
(type 'float')
```

只要是用双引号或单引号括起来的值，都属于字符串。例如:

```
>>> type("31")
(type 'str')
>>> type("2.5")
(type 'str')
>>> type("P001")
<type 'str'>
```

2.2 变量名和关键字

程序中的变量名要有实际意义。变量名可以是由数字和字符组成的任意长度的字符串，但必须以字母开头。**python**是区分大小写的。举个例子来说，Name和name是两个不同的变量名。请看下面的例子：

```
>>> Name = "pi"
>>> name = "da"
>>> print name, Name
da pi
```

符号“_”连接由多个单词组成的变量名。请看下面的例子：

```
>>> my_name = "pi da qing"
>>> print my_name
pi da qing
```

如果定义了一个错误的变量名，解释器显示语法错误。请看下面的例子：

```
>>> 1_first = "第一名"
SyntaxError: invalid syntax
>>> my$ = "dollar"
SyntaxError: invalid syntax
>>> print = 12345
SyntaxError: invalid syntax
```

变量1_first不应以数字开头；变量my$包含非法的字符；变量名print与python定义的关键字print重名，所以print不能再当作变量名了。python定义了28个关键字：

```
and      continue    else      for      import      not      raise
assert   def         except    from     in          or       return
break    del         exec      global   is          pass     try
class    elif        finally   if       lambda      print    while
```

2.3 语句

语句是Python解释器可以执行的命令。我们已经知道两条语句：打印和赋值。

赋值语句有两个作用：一是建立新的变量，二是将值赋予变量。任何变量在使用时，都必须赋值。否则，被视为不存在的变量。下面例子有三条赋值语句。第一个是将值“How are you？”赋值给字符串变量message；第二个是将18赋值给整数变量n；第三个是将数字3.1415926赋值给浮点变量pi。当打印不存在的变量时，系统给出错误信息。如下:

```
>>> message = "How are you?"
>>> n = 18
>>> pi = 3.1415926
>>> print abc                    #错误语句，没有该变量。
NameError: name 'i' is not defined
```

一般的情况下，我们用状态图表示变量的状态。左边是变量的名称，右边是变量值，中间的箭头指向值。状态图显示了赋值语句的最终操作结果（如图 2.1）。

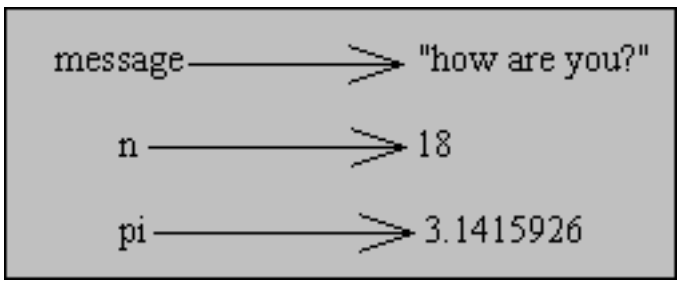

图 2.1: 变量的状态图

也可以用函数type检查变量的类型。变量的类型就是它所指向的值的类型。

```
>>> type(message)
(type 'str')
>>> type(n)
(type 'int')
```

```
>>> type(pi)
(type 'float')
```

打印语句print输出表达式的计算结果。单个变量也可以看作是表达式。如果你想在一行打印多个变量，可以用逗号将这些变量隔开，逗号禁止换行。例如：

```
>>> x = 3
>>> y = 8.9
>>> print x, y, "hello", 9
3 8.9 hello 9
>>> print x, '\t', y, "\t", "hello", "\t", 9
3   8.9     hello   9
```

第二个print语句用制表符将这些变量隔开。

下面是一些特殊符号的打印方法：

```
>>> #打印单引号
>>> print "'"
''
>>> #打印双引号
>>> print '""'
""
>>> #打印换行符
>>> print '\n'


>>> #打印反斜杠
>>> print '\\'
\
```

2.4 表达式

表达式由值、变量和运算符组成。如果在命令行上输入表达式，解释器惊醒计算，并显示结果：

```
>>> 23.3 + 1.2
24.5
```

单一的值或变量也可以当作是表达式：

```
>>> 45
45
>>> x = 1.2
>>> x
1.2
```

计算表达式和打印值是有很大区别的，要注意区分。

```
>>> "I am free!"
'I am free!'
>>> message = "I am free!"
>>> message
'I am free!'
>>> print message
I am free!
>>> print "I am free!"
I am free!
```

当Python显示表达式的值时，显示的格式与你输入的格式是相同的。如果是字符串，就意味着包含引号。而打印语句输出的结果不包括引号，只有字符串的内容。

在脚本文件中，任何表达式都被认为是合法的语句，但是这个语句不做任何事。你可以试着运行下面的脚本文件。

```
# 表达式在脚本文件不做任何事。
12345
34.90
"I am free!"
3 + 5
```

这个脚本根本没有任何输出。

2.5 运算符和操作数

运算符是像加号和减号之类的特殊符号。运算符操作的对象是操作数。

符号+, -, / ，（）的意义与数学中的意义基本相同。一个星号代表的是乘法，两个星号代表的是乘幂。请看下面的例子：

下面的例子都是合法的Python表达式：

```
20 + 32
hour - 1
```

```
hour * 60 + minute
minute / 60
5 ** 2
(5+9) * (15-7)
```

操作数还可以是变量，当运算执行前，变量被它所指向的值替代。

```
>>> 3 * 4
12
>>> 3 ** 4
81
>>> x = 9
>>> y = 6
>>> x * y
54
```

需要注意的是除法运算。如果除数和被除数都是整数，那么结果是截调小数部分的整数。解决的办法是将除数和被除数任意之一加小数点，或是加小数点和零。例如：

```
>>> 5 / 2
2
>>> 5.0 / 2
2.5
>>> 5 / 2.0
2.5
>>> 5 / 2.
2.5
```

模数操作符“%”计算两个整数的余数：

```
>>> print 5 \% 3
2
>>> print 123 \% 12
3
```

Python还支持复数的运算。复数有两种表示方法,一种如：

```
>>> a = 1 + 5j
>>> b = 4j
```

另一种是用函数的方法：

```
>>> x =complex(1, 5)
>>> print x
(1+5j)
```

复数也能够进行数学运算：

```
>>> a = 3 +3j
>>> b = 4 + 4j
>>> print a + b
(7+7j)
>>> print a - b
(-1-1j)
>>> print a * b
24j
>>> print a / b
(0.75+0j)
```

2.6 运算的顺序

如果表达式中有不止一个运算符，那么计算的顺序依靠“优先规则”。Python运算符的“优先规则”和算术的相同。以下关于“优先规则”的建议：

- 小括号有最高的优先级。在表达式中，括号内的首先计算。我们可以利用括号使表达式更容易读懂，而不影响计算结果。
- 乘幂运算的优先级仅次于小括号。例如：

```
>>> （1 + 3） ** 2
16
>>> 3 * 3 ** 2
27
```

- 乘法和除法的优先级相同，比加减法的优先级高。加减法的优先级也是相同的。例如：

```
>>> 2 + 3 * 3 + 2
13
>>> 4 - 10 / 5
2
>>> 4 * 5 / 2
```

```
10
```

- 具有相同优先权的运算符从左到右进行计算。

2.7 字符串操作

通常情况下，字符串放在双引号或单引号之间。字符串不能进行除法、减法和字符串之间的乘法运算，下面的操作都是非法的：

```
>>> "hello" / 3
TypeError: unsupported operand type(s) for /: 'str' and 'int'
>>> string = "string"
>>> string - 1
TypeError: unsupported operand type(s) for -: 'str' and 'int'
>>> string * "hello"
TypeError: unsupported operand type(s) for *: 'str' and 'str'
```

加法“+”能够连接两个字符串成为一个字符串。例如：

```
>>> string1 = "Red"
>>> string2 = "Hat"
>>> print string1 + string2
RedHat
```

由于“Red”和“Hat”没有空格，所以它们非常亲密的连在一起。如果想让字符串之间有空格，可以建一个空字符变量，插在相应的字符串之间让它们隔开，或是在字符串中加入相应的空格。

```
>>> space = " "
>>> str1 = "A"
>>> str2 = "B"
>>> str3 = "C"
>>> print str1 + space + str2 + space + str3
A B C
>>> astr = 'Linux '
>>> bstr = ' Unix '
>>> cstr = 'OS/2'
>>> print astr + bstr + cstr
Linux  Unix OS/2
```

符号“”也可以操作字符串。只是其中一个操作数必须是字符串，另一个必须是整数。字符串被重复整数遍。例如：

```
>>> 3 * "Love"
'LoveLoveLove'
>>> string = "python"
>>> string * 2
'pythonpython'
```

2.8 组合

到目前为止，我们已经学习了程序的几大元素：变量，表达式和语句。但只是孤立的学习，还没有考虑怎样把他们组合起来，完成一项具体的操作。例如，我们可以把加法操作和打印语句结合起来，显示加法表达式的结果：

```
>>> print 10 + 9
19
```

实际上，加法操作发生在打印操作之前，所以这两个动作不是同时发生的。任何涉及到数字，字符串和变量的表达式都可作为Print语句的参数。例如：

```
>>> interest = 0.003
>>> saving = 12345.98
>>> print "Total: ", saving + saving * interest
Total:  12383.01794
```

赋值语句，即等号“=”的右边可以是任意的表达式。表达式的值是什么类型，变量就是什么类型。

```
>>> int = 5
>>> str = "hello"
>>> var = int * 7
>>> type(var)
<type 'int'>
>>> var = str + " world"
>>> type(var)
<type 'str'>
```

有一点要记住，赋值语句的左边不能出现表达式。像下面的语句是非法的：

```
>>> name + 9 = 10       #错误的赋值语句。
SyntaxError: can't assign to operator
```

2.9 注释

当程序越来越复杂时，读懂它就变得非常困难。程序的各部分之间紧密衔接，想依靠部分的代码来了解整个程序要做的，是困难的。

因此，好的习惯是在程序中加入适当的注释，以解释它要做的事情。注释必须以符号“#”开始：

```
# 打印1＋1的结果
print 1 + 1
```

注释可以单独占一行，也可以放在语句行的末尾：

```
print 1 + 1    # 打印1＋1的结果
```

从符号“#”开始，到这一行的末尾之间的所有内容都被忽略，这部分对程序没有影响。注释信息主要是方便了程序员，一个新来的程序员通过注释信息，能够较快的了解程序所做的；原来的程序员在经过一段时间后，可能对自己的程序不了解了，利用注释信息就能较快的熟悉。

第三章

函数

3.1 函数

我们已经见过了函数调用的例子:

```
>>> type("world")
(type 'str')
```

函数type,它的作用是显示值和变量的类型。括号内的变量和值是函数的参数。函数返回的结果叫返回值。

我们可以把type的返回值赋值给变量:

```
>>> name = type("pitianjian")
>>> print name
<type 'str'>
```

函数id以值或变量为参数，返回值是一整数，它表示值或变量的唯一标识符。

```
>>> id(123)
11602164
>>> number = 123
>>> id(number)
11602164
>>> number = "123"
>>> id(number)
13087320
```

每个值或变量都有一个唯一id，id值与变量或值在内存中的位置相关。变量的id也就是它所指向值的id。

3.2 函数定义

到目前为止，我们用的都是Python定义的函数。这些**Python**内置的函数，其定义部分对我们来说时透明的。因此，我们只关注这些函数的用法，而不必关心函数是如何定义的。

我们也可以创建自己的函数，来执行特定的操作。函数的定义形式如下：

```
def <name>(arg1, arg2,... argN):
    <statements>
```

函数的名字也必须以字母开头，可以包括下划线“_”,但不能把**Python**的关键字定义成函数的名字。函数内的语句数量是任意的，每个语句至少有一个空格的缩进，以表示此语句属于这个函数的。缩进结束的地方，函数自然结束。

下面定义了一个两个数相加的函数：

```
>>> def add(p1, p2):
        print p1, "+", p2, "=", p1+p2

>>> add(1, 2)
1 + 2 = 3
```

函数的目的是把一些复杂的操作隐藏，来简化程序的结构，使其容易阅读。函数在调用前，必须先定义。也可以在一个函数内部定义函数，内部函数只有在外部函数调用时才能够被执行。程序调用函数时，转到函数内部执行函数内部的语句，函数执行完毕后，返回到它离开程序的地方，执行程序的下一条语句。

3.3 函数的行参和实参

让我们看下面的例子：

```
>>> def subtracter(p1, p2):
        print p1, "-", p2, "=", p1-p2

>>> var1 = 3.1415
>>> var2 = 4.0987
>>> subtracter(var1, var2)
3.1415 - 4.0987 = -0.9572
```

在这个例子中，函数subtracter有两个参数：p1和p2。他们是函数的行参。变量var1和var2是函数的实参。实参把它的值传递给行参，行参被函数内的语句进行各种操作，而实参没有被改变。

函数也可以将表达式当作它的参数：

```
>>> def display(arg):
    print arg

>>> a1 = 4.5
>>> a2 = 3
>>> display(a1 + a2)
7.5
>>> import math
>>> display(math.pi)
3.14159265359
>>> display("I love you." * 2)
I love you.I love you.
```

3.4 变量的范围

Python有一个主函数：_main_。其它的函数都是在这个函数内执行，或者说_main_调用你的程序及程序内的函数。你在任何函数外创造的变量都属于_main_。看下面的例子：

```
>>> def multiply(p1, p2):
        print p1, "*", p2, "=", p1*p2
>>> def devide(p3, p4):
        print p3, "/", p4, "=", p3/p4
>>> v1 = 2.5
>>> v2 = 5.9
>>> multiply(v1, v2)
2.5 * 5.9 = 14.75
>>> devide(v1, v2)
2.5 / 5.9 = 0.423728813559
```

在这个例子中，定义了两个函数：multiply和devide。p1和p2是函数multiply的变量，p3和p4是函数devide的变量。v1和v2是函数_main_的变量。当两个函数调用完毕后，p1，p2，p3，p4就消失了。变量的范围如图 3.1：

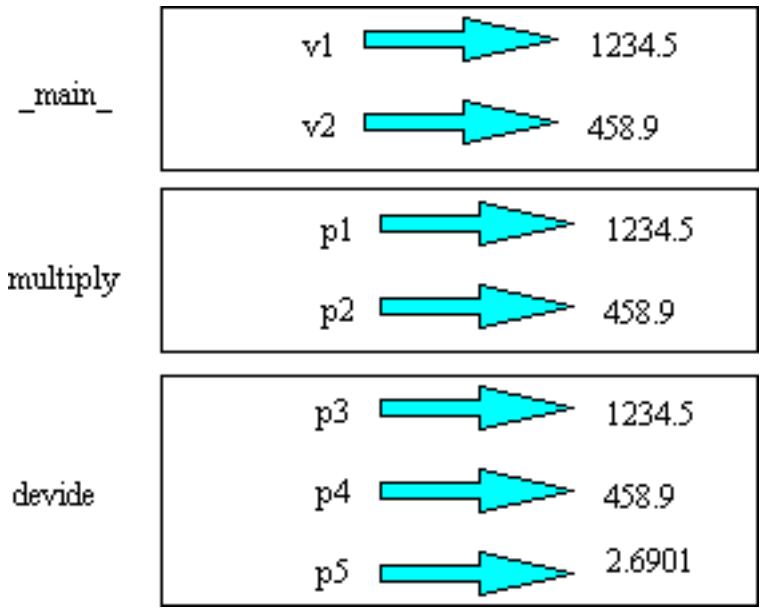

图 3.1: 变量的范围

3.5 函数的返回值

函数的返回值，形象的说是函数结的“果实”；有返回值的函数，称之为“结果”的函数。例如我们要计算圆的面积：

```
>>> import math

>>> def area(radius):
        atemp = math.pi * radius * 2
        return atemp
```

“return atemp”语句的意思是：将atemp表达式的值立即返回，表达式可以是任意复杂的，所以这个函数还可以进一步简化成如下的形式：

```
>>> import math

>>> def area(radius):
        return math.pi * radius * 2
```

有时根据不同的条件，函数能够有多个返回语句：

```
def exampass(x):
    if (x>=60):
        return 1
```

```
    else:
        return 0
```

这些return语句处于不同的条件语句中，并且只可能有一个return语句被执行，之后函数就终止了。在有返回值的函数中，一定要保证每个可能的流程都对应着return语句，请看下面的例子：

```
def abs(x):
    if x<0:
        return -x
    elif x>0:
        return x
```

此函数缺少了一条对应x＝0的return语句，这种情况下，函数返回值是None（注意N要大写）：

```
>>> print abs(0)
None
```

None是什么类型呢？可以用函数type检验一下：

```
>>> type(None)
<type 'NoneType'>
>>> id(None)
504026256
```

由此可见，None是不属于任何类型的类型。

函数也可以返回布尔值，这种函数的内部作了复杂的测试。例如：

```
def isDivisible(x, y):
    if x%y == 0:
        return 1
    else:
        return 0
```

这个函数判断x是否能够被y整除，是则返回1，否则返回0。我们还可以使这个函数更加简洁，而且不再用if语句。

```
def isDivisible(x, y):
    return x%y == 0
```

如果x能够被y整除，则x%y为0，而0与0是相等的，所以返回1；否则，x%y为非0，非0的数与0是不相等的，函数返回0。请看他们的执行结果：

```
>>> isDivisible(23, 2)
0
>>> isDivisible(4, 2)
1
```

返回布尔值的函数经常用在条件语句中：

```
if isDivisible(x, y):
    print "x被y整除"
else:
    print "x不能被y整除"
```

下面的写法也是对的，但却是画蛇添足，根本没有必要。

```
if isDivisible(x, y) == 1:
```

3.6 类型转换

Python提供了将变量或值从一种类型转换成另一种类型的内置函数。int函数能够将符合数学格式数字型字符串转换成整数。否则，返回错误信息。

```
>>> int("34")
34
>>> int("1234ab")          #不能转换成整数
ValueError: invalid literal for int(): 1234ab
```

函数int也能够把浮点数转换成整数，但浮点数的小数部分被截去。

```
>>> int(34.1234)
34
>>> int(-2.46)
-2
```

函数float将整数和字符串转换成浮点数：

```
>>> float("12")
12.0
>>> float("1.111111")
1.111111
```

函数str将数字转换成字符：

```
>>> str(98)
'98'
>>> str("76.765")
'76.765'
```

整数1和浮点数1.0在python中是不同的。虽然它们的值相等的，但却属于不同的类型。这两个数在计算机的存储形式也是不一样。

3.7 数学函数模块

Python有一个math模块，提供了大部分与数学计算相似的函数。模块是一个文件，它是功能类似的函数的集合。

如果我们想利用math模块中的函数，首先要用关键字import引入模块：

```
>>> import math
```

然后用点操作符调用模块中的函数：

```
>>> import math
>>> print math.log10(10)
1.0
>>> print math.sin(1.5)
0.997494986604
```

如果你不想用点操作符，而直接写出math模块中的函数，需要用下面的语句重新输入math中的函数：

```
>>> from math import *
>>> cos(3)
-0.98999249660044542
```

如果你定义的函数中有与cos同名的，math模块中cos函数将被覆盖，即使用点操作符也不能调用math中的cos函数。

在数学模块math中，有两个数学常量：pi和e。

```
>>> print math.pi
3.14159265359
>>> print math.e
2.71828182846
```

再看几个数学函数的例子：

```
>>> math.exp(9)                #e的9次幂。
8103.0839275753842
>>> math.pow(3,4)              #3的4次幂。
81.0
>>> math.sqrt(3.44)            #3.44开平方。
1.8547236990991407
>>> math.sin(math.pi/2)        #正弦函数。
1.0
>>> math.cos(math.pi * 3.4)    #余弦函数。
-0.30901699437494784
>>> math.fabs(-34.90)          #求绝对值。
34.899999999999999
```

3.8 lambda函数

Python允许你定义一种单行的小函数。定义lambda函数的形式如下：

```
labmda 参数：表达式
```

lambda函数默认返回表达式的值。你也可以将其赋值给一个变量。

lambda函数可以接受任意个参数，包括可选参数，但是表达式只有一个：

```
>>> g = lambda x, y: x*y
>>> g(3,4)
12
>>> g = lambda x, y=0, z=0: x+y+z
>>> g(1)
1
>>> g(3, 4, 7)
14
```

也能够直接使用lambda函数，不把它赋值给变量：

```
>>> (lambda x,y=0,z=0:x+y+z)(3,5,6)
14
```

如果你的函数非常简单，只有一个表达式，不包含命令，可以考虑lambda函数。否则，你还是定义函数才对，毕竟函数没有这么多限制。

第四章

条件表达式

4.1 布尔表达式

布尔表达式的值只有两个：真和假。在Python语言中，真值为1，假值为0。下面是Python语言中较常用的比较两个数大小运算符：

```
x == y          #x等于y
x != y          #x不等于y
x > y           #x大于y
x < y           #x小于y
x >= y          #x大于等于y
x <= y          #x小于等于y
```

下面是这些运算符的例子：

```
>>> x = 3.4
>>> y = 9
>>> z = 4.55
>>> x == y
0
>>> x != z
1
>>> x > y
0
>>> y > x
1
>>> z <= y
1
```

Python语言中大部分比较操作符的意义与数学中类似。有两个操作符的用法容易混肴，就是赋值操作符“＝”和比较操作符中“＝＝”。在比

较两个数是否相等时，经常错误的用赋值操作符“＝”作比较，而忘记了它的真正含义是赋值操作，操作符“＝＝”才是真正比较这两个数是否相等的运算符。另外，还要注意大于等于和小于等于操作符的写法，不要将它们写成这样的形式：“=<”和“=>”。

4.2 逻辑操作符

Python有三种逻辑操作：and、or、not。这三个操作符的语义与其英语意义相同，分别是：与、或、非。

例如，$(x > 0) and (x < 10)$这个表达式，and操作符连接两个条件表达式，只有x大于0，并且x小于10的时候，整个表达式才为真。

再看这个表达式，$(n\%2 == 0) or (n\%3 == 0)$，or 连接两个判断是否等于0的表达式，只要n能够被2整除，或是被3整除，这两个表达式只要有一个为真，整个表达式就为真。

最后，not操作符对表达式的值取反。$not(x > y)$，如果x大于y，取反后整个表达式的值为假，否则，为真。

严格来说，逻辑操作符的操作数应该为布尔表达式，但Python对此处理的比较灵活，即使操作数是数字，解释器也把他们当成“表达式”。非0的数字的布尔值为1，0的布尔值是0。看下面的例子：

```
>>> a = 1
>>> b = 0
>>> a and b
0
>>> a or b
1
>>> not a
0
>>> not 0
1
```

在python中空字符串为假，非空字符串为真。非零的数为真。我们研究一下数字和字符串之间、字符串之间的逻辑操作规律。首先看and操作符：

```
>>> 9 and "OK"
'OK'
>>> "OK" and 9
9
>>> "OK" and "ME"
'ME'
```

```
>>> "" and 9
''
>>> "" and "OK"
''
>>> 0 and "OK"
0
```

只要左边的表达式为真，整个表达式返回的值是右边表达式的值。否则，返回左边表达式的值。

or操作符的规则是：只要两边的表达式都为真，整个表达式的结果是左边表达式的值；如果是一真一假，返回真值表达式的值。特别注意的是空值和0的情况，这时候返回的是右边的0或空值：

```
>>> "" or 0
0
>>> 0 or ""
''
```

not的情况比较简单：

```
>>> not 0
1
>>> not ''
1
```

4.3 条件语句

在程序执行的过程中，时常依据一些条件的变化，改变程序的执行流程。改变程序流程的功能，主要由条件语句配合布尔表达式来完成。例如：

```
if (x>0):
    print x
```

当x大于0时，执行print语句；若x小于或等于0，print语句不被执行。

if语句由“头”和“块”组成，它的书写方式在Python中具有典型的代表性：

```
Header:
    First Statement
    Second Statement
    ......
    Last Statement
```

```
Other Statement
```

“头”是if语句的开始，以冒号结束。在“头”里进行条件判断，以确定是否执行if语句的“块”部分。在if语句的“块”的每一行，必须至少有一个空格的缩进，缩进表示这一行是属于if语句的一部分。第一个没有缩进的语句标志着if语句的结束。“块”内至少有一条语句，如果暂时还没有任何操作，你可以加一条语句pass，它什么都不做，是一条空语句。

需要注意的是if语句行以冒号结束。可能是习惯了C+ +和java中写法，这个冒号我总是忘了写。这个冒号与文章中冒号的用法类似。例如：

这部手机的特点是：
待机时间长；
价格便宜；
.....
操作方便。

所以我买了它。

冒号用以引出要说的内容，if语句同样是引出要执行的语句；在上一个例子中，手机的特点写完后，缩进结束，表示这一部分内容已经写完了，再往后就是别的的内容了。同样，if语句后面第一个没有缩进的语句行，就不属于它的“块”了，而是程序要执行的其它内容了。

第二种if语句的形式是有两个执行流程，关键字else（否则）引出另一个程序流程。例如：

```
if x%2 == 0:
    print x, "is even"
else:
    print x, "is odd"
```

如果x除以2的余数是0，则x是偶数，执行第一条打印语句；如果x是奇数，就执行第二条打印语句，也就是else引出的语句。因为x不是奇数就是偶数，只能二者居其一，所以这两条打印语句只能执行其中的一条，即这段代码的执行流程有两个分枝。这段代码可用语言表述成：如果x是偶数，则执行第一条打印语句；否则，x必定是奇数，执行第二条打印语句。高级程序语言是不是和自然语言很接近？

我们也可以将这段代码“封装”成一个检验奇偶的函数：

```
def isParity(x):
    if x%2 == 0:
        print x, "是偶数"
    else:
        print x, "是奇数
```

该函数的执行结果如下：

```
>>> import sys
>>> sys.path.append('c:\python')    #设定此脚本文件的路径。
>>> from isparity import isParity   #从脚本文件中引入函数。
>>> isParity(8)
8 是偶数
>>> isParity(9)
9 是奇数
```

有时候，程序的分支不止两个，可能是三个，或更多。此时，就需要elif语句引出更多的分支。elif语句是“else if”的缩写，每一个elif语句为程序引出一个分支。elif语句的数量没有限制，但最后的分支必须是else语句，并且只能是最后一个程序分支。请看下面的脚本文件：

```
def largeNumber(x, y):
    if x < y:
        print x, "小于", y
    elif x > y:
        print x, "大于", y
    else:
        print x, "等于", y
```

程序按顺序检查条件表达式，当找到第一个满足要求的表达式后，执行此分支内的语句。剩下的条件，即使有满足要求的，也不做检查。请看上一函数执行的结果：

```
>>> import mylib
>>> mylib.largeNumber(3, 9)
3 小于 9
>>> mylib.largeNumber(4, 1.2)
4 大于 1.2
>>> mylib.largeNumber(0, 0)
0 等于 0
```

4.4 while语句

计算机最擅长做自动的、重复性的工作，而且不会出错。几乎任何语言都有关于循环方面的语句。让我们看一看Python的第一个循环语句：while。请看下面的函数：

```
        def f(x):
            result = 0
                while(x):
                    result = result + x
                    x = x - 1
            return result
```

这个函数计算从1到x之间所有整数之和。当我们传递参数给x时，while语句判断x是否大于0，大于0则执行while内的语句。每执行一次循环，x减去1，当x等于0时，while循环终止。

因此，while执行的过程如下：

- 计算循环的条件，将得到1或0；
- 如果条件为0，退出while循环，执行循环外的语句；
- 如果条件为1，执行while块内的每一条语句，然后返回第一步。

如果第一次循环的条件为0，则while语句块的内容永远也不会被执行。另外，循环体内要有使循环结束的方法，从而终止循环的执行。否则，就成了永远不会停止的无限循环。

下面的例子是打印九九乘法表：

```
def minus():
    x = 0;
    y = 0;
    while(x <= 9):
        while(y <= 9):
            if(y == 4):
                print x, "*", y, "=", x * y
            else:
                print x, "*", y, "=", x * y, '  ',
            y = y + 1
        print
        x = x + 1
        y = 0
```

这个例子中有两个while语句，其中的一个嵌套于另一个之中。

4.5 条件嵌套

一个条件可以包含在另一个条件中。看下面的例子：

```
def isEqual(x, y):
    if x == y:
        print x, "等于", y
    else:
        if x < y:
            print x, "小于", y
        else:
            print x, "大于", y
```

这个函数有两个分支：第一个分支只是简单的打印语句；第二个分支又包含了两个分支，每个分支也是打印语句。这个程序不利于阅读。编程时，我们要避免出现这样的情况。

逻辑操作符提供了简化条件嵌套的方法。例如，判断一个数是否大于0且小于10，第一种写法如下：

```
>>> x = 8.9
>>> if x > 0:
        if x < 10:
            print "x大于0且小于10"

x大于0且小于10
```

第二种写法利用了and操作符，易于阅读，是通常的用法：

```
>>> if (0<x) and (x<10):
        print "x大于0且小于10"

x大于0且小于10
```

第三种写法与数学上的很相似：

```
>>> x = 1
>>> if 0 < x < 10:
    print "x大于0且小于10"

x大于0且小于10
```

4.6 return语句

return在函数中返回函数值。它的另一个作用是当函数内有错误发生时，终止函数的运行，提前退出。例如：

```
import math

def printc(a, b):
    if  (a-b) < 0:
        print "a小于b"
        return

    print math.sqrt(a - b)
```

当a小于b时，return终止函数的执行，下一条语句也不执行了。

4.7 键盘输入

Python提供了内置的函数获得键盘的输入。这个函数是raw_input，它被调用时，程序暂停执行，等待用户输入一些信息。当你按下回车键后，程序恢复执行，并且raw_input 返回用户输入的内容。例如：

```
>>> input = raw_input()
I love Python!
>>> print input
I love Python!
```

在调用函数raw_input 之前，最好给用户一些提示，否则我们不知道程序要我们作什么。提示的内容可以作为函数raw_input的参数。

```
>>> name = raw_input("Please input your name:")
Please input your name:pidaqing
>>> print name
pidaqing
```

第五章

字符串

5.1 组合数据类型

到目前为止，我们已经学习了三种数据类型：int、float和string。string类型与其它两种类型的数据还有本质的区别，因为它是由多个字符组成的。我们可以把字符串当成一个整体，也可以取得字符串的任何部分。

操作符“[]”从字符串中取出任意个连续的字符：

```
>>> name = "qswtp"
>>> print name[0]
q
>>> print name[4]
p
>>> letter = name[2]
>>> print letter
w
>>> a = 1
>>> b = 2
>>> print name[a + b]
t
```

中括号内表达式是字符串的索引，它表示字符在字符串内的位置。字符串第一个字符的索引是0，而不是1，切记。

函数len返回字符串的长度：

```
>>> address = "www.qswtp.com"
>>> len(address)
13
```

5.2 用for语句遍历字符串

许多操作都涉及对字符串的处理，比如从第一个字符开始，按照顺序读取字符，然后在做相应的处理，直到最后一个字符，这个处理过程叫做遍历。可以用while语句遍历字符串中的每一个字符：

```
def travel(string):
    index = 0
    while index < len(string):
        letter = string[index]
        print letter
        index = index + 1
```

当字符串的索引值等于字符串的长度时，while的条件为假，这时字符串已经浏览完毕，退出循环。

Python提供了另一种语句for来遍历字符串，它的用法更加简练：

```
>>> for char in address:
        print char
```

在每一次的循环中，依次把字符串中的一个字符赋值给变量char，然后打印，直到最后一个字符被打印之后，for语句结束循环。

5.3 字符串片断

字符串的一部分叫做片断。选取片段与选单个取字符很相似：

```
>>> os = "Linux Unix FreeBSD"
>>> print os[0:5]
Linux
>>> print os[6:10]
Unix
>>> print os[11:18]
FreeBSD
```

操作符[n:m]返回字符串的一部分，从第n个字符开始，到第m个字符结束，包括第n个，但不包括第m个。如果你忽略了n，则返回的字符串从索引0开始；如果忽略了m，则字符串从n开始，到最后一个字符。

```
>>> os = "Linux Unix FreeBSD"
>>> print os[:5]
Linux
>>> print os[11:]
FreeBSD
```

如果我们试图用“[]”操作符修改字符串中的任何一个字符，就像下面车操作一样：

```
love = "python"
love[0] = P
```

你别指望着变量love的值变成“Python”，这时将产生运行时错误。这表明字符串的值是不可改变的。最好的办法是取原字符串的一部分和添加的字符串相加，生成一个新的字符串：

```
>>> str = "python"
>>> newstr = "P" + str[1:]
>>> print newstr
Python
```

下面的例子是从一个字符串中，寻找是否存在给出的字符：

```
def find(string, ch):
    index = 0
    while index < len(string):
        if string[index] == ch:
            return index
    index = index + 1
return -1
```

如果string[index] == ch成立，函数立即返回字符的索引，并且退出循环。如果在字符串中没有寻找的字符，程序正常退出循环，并返回-1。

我们再看一个计算字符串中出现字符次数的函数：

```
def count(string, ch):
    count = 0
    for char in string:
        if char == ch:
            count = count + 1

    print count
```

用for语句做一个循环，每次发现一个相等的字符，变量就加1，最后打印出变量count。

5.4 字符串模块

字符串模块string包含一些处理字符串的函数。在用模块前必须先引入：

```
import string
```

字符串模块中包含一个名为find的查找字符的函数，我们也曾写过这样的一个例子。为了调用此函数，需利用“.”点操作符：

```
>>> bookname = "new concept english"
>>> import string
>>> index = string.find(bookname, "s")
>>> print index
17
```

即使我们定义了自己的find函数，也不会与string模块的find函数冲突，因为点操作符使我们能够区分，也使解释器能够区分调用的究竟是哪一个find函数。

实际上，string.find函数还有几个版本。

寻找字符串中的子串开始的位置：

```
>>> string.find("www.qswtp.com", "com")
10
```

在上面的例子中再加一个参数，表明从哪开始搜寻子串：

```
>>> string.find("I love Python!", 'o', 5)
11
>>> string.find("I love Python!", 'o', 1)
3
```

确定寻找字符串的索引范围：

```
>>> string.find("compaq", 'c', 1, 3)
-1
>>> string.find("compaq", 'p', 1, 3)
-1
```

如果找不到，就返回-1。

程序中经常要判断字符是大写还是小写，或者判断是字符还是数字。为此目的，string模块提供了几个有用的常量字符串。

常量string.lowercase包含了所有小写的英文字母；常量string.uppercase包含了所有大写的英文字母；常量string.digits包含了从0到9的数字。

```
>>> import string
>>> print string.lowercase
abcdefghijklmnopqrstuvwxyz
>>> print string.uppercase
ABCDEFGHIJKLMNOPQRSTUVWXYZ
>>> print string.digits
0123456789
```

利用string.lowercase常量，我们可以编写一个判断字符是否为小写的函数：

```
import string
def isLower(char):
    return string.find(string.lowercase, char) != -1
```

我们还可以利用in操作符编写这个函数。in操作符判断一个字符是否存在于字符串中：

```
import string
def isLower(char):
    return char in string.lowercase
```

最后一种选择方案是利用比较操作符：

```
import string
def isLower(char):
    return 'a' <= char <= 'z'
```

第六章

列表

列表是一组任意类型的值，按照一定顺序组合而成的。组成列表的值叫做元素(Elements)。每一个元素被标识一个索引，第一个索引也是0。列表中的元素可以是任意类型，甚至是列表类型，也就是说列表可以嵌套。

6.1 列表值

列表中的元素用中括号括起来，以逗号分割元素。

```
>>> [10, 20, 30, 40]
>>> ["spam", "bungee", "swallow"]
>>> ["name", 12]
```

第一个列表包含四个整数。第二个列表包含三个字符串。

下面的列表包含三种类型的元素：一个字符串，一个浮点数，一个整数，还有另外一个列表：

```
>>>["hello", 2.0, 5, [10,20]]
```

列表的元素还可以是变量。但是修改变量的值，并不影响列表中元素的值。

```
>>> a = 1
>>> b = 2
>>> num = [a, b, 3]
>>> print num
[1, 2, 3]
>>> a = b =9
>>> print num
[1, 2, 3]
```

包含连续整数的列表是很常见的，因此Python提供一种简单的方法创建他们：

```
>>>range(1,5)
>>>[1, 2, 3, 4]
```

函数range返回一个整数列表，列表从函数的第一个参数开始，到最后一个参数结束，不包含最后一个参数，相邻两数之间的差值是1。

range函数还有另外两种形式。单一参数的range函数，产生从0开始的列表：

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

3个参数的range函数，产生一个在数学上成为等差数列的列表。这个例子产生一个从1到10，步长为2的列表：

```
>>>range(1,10,2)
>>>[1,3,5,7,9]
```

最后，还有一种特殊不包含元素的列表，称为空列表。并且它被表示为”[]”。列表可以赋值，或作为参数传递给函数。

6.2 读写元素

读写列表中元素的方法与读写字符串中字符的方法一样－都是通过操作符“[ ]”。中括号内的表达式代表索引，请记住索引是从0开始的。

```
>>> numbers = [13, 15]
13
>>> print numbers[0]
>>> numbers[1] = 5
>>> print numbers[1]
5
```

索引可以是任何整数表达式，但不能是浮点数：

```
>>> numbers[3 - 2]
5
>>> numbers[1.0]              #列表的索引必须是整数。
TypeError: sequence index must be integer
```

如果你读写一个不存在的元素，将会发生一个运行时错误：

```
>>> numbers[2] = 5          #索引超出范围。
IndexError: list assignment index out of range
```

错误的提示表明，索引超出了这个列表的范围。

索引也可以是一个负数，那么列表的最后一个元素的索引是-1，倒数第二个的索引是－2，依此类推：

```
>>> numbers = [1, 2, 3, 4]
>>> numbers = [-1]
4
>>> numbers[-2]
3
>>> numbers[-3]
2
>>> numbers[-4]
1
```

6.3 列表的一些方法

append在列表的尾部追加元素，参数是插入元素的值：

```
>>> number = [0, 1, 2, 3]
>>> number.append(4)
>>> print number
[0, 1, 2, 3, 4]
```

方法insert在列表中插入元素，它有两个参数，一个是索引位置，一个是插入元素的值：

```
>>> number.insert(3, 5)
>>> print number
[0, 1, 2, 5, 3, 4]
```

方法extend合并两个列表为一个：

```
>>> nation1 = ["French", "German"]
>>> nation2 = ["Chinese", "Korean"]
>>> nation1.extend(nation2)
>>> print nation1
['French', 'German', 'Chinese', 'Korean']
>>> print nation2
['Chinese', 'Korean']
```

所谓合并就是将一个列表的元素添加的另一个列表中。

方法index取得元素的索引值：

```
>>> cars = ["Ford", "Fiat", "Volvo"]
>>> cars.index("Ford")
0
```

方法remove从列表中删除第一次出现的值：

```
>>> color = ["red", "green", "yellow", "green", "black"]
>>> color.remove("green")
>>> print color
['red', 'yellow', 'green', 'black']
```

方法pop删除最后一个值，然后返回这个值：

```
>>> letter = ['a', 'b', 'c', 'd']
>>> letter.pop()
'd'
>>> print letter
['a', 'b', 'c']
```

6.4 列表长度

函数len返回列表的长度，即元素的个数。可以用这个值作为遍历列表的变量。这就意味着，即使列表的长度改变，我们也不用对程序的循环次数作出更改。看下面的例子：

```
>>>os=["Linux","Unix","FreeBSD","Mac"]
>>>i=0
>>>while i<len(os):
...    print os[i]
...    i=i+1
...
Linux
Unix
FreeBSD
Mac
```

当循环体最后一次执行时，i的值是：len(os)-1，即最后一个元素的索引。当i等于Len(os)时，循环停止执行，因为len(os)已不是一个合法的索引。

虽然列表之中还可以包含列表，但嵌套的列表被当作一个元素。下面列表的长度是4。

```
>>>length=['book','1',['word','wps','open'],[1,2,3]]
>>>len(length)
4
```

in是一个布尔操作符，它测试左边的操作数是否包含于列表。

```
>>>os=["Linux","Unix","Mac","Windows"]
>>>'windows' in os
1
>>>'BeOS' in os
0
```

因为windows是os列表的一个成员，所以in操作符返回1。而Beos不是，就返回0。in也可以测试字符串中是否包含某个字符：

```
>>> string = "1234567890"
>>> "1" in string
1
>>> 'a' in string
0
```

我们可以用not和in 的组合来测试一个元素不包含于列表。

```
>>>os=["Linux","Unix","Mac","Windows"]
>>>'BeOS' not in os
1
```

6.5 列表和for循环

用for循环遍历列表，更加简明，它没有循环变量。

```
os=["Linux","Unix","Mac","Windows"]
for os in os:
    print os
```

这就像在读英语句子，将os列表中的元素依次读入变量os，并打印变量os。

任何列表表达式都可以用于for循环中：

```
for number in range(20):
    if number\%2==0:
        print number

for fruit in ["banana","apple","auince"]:
    print "I like to eat" + fruit + "s!"
```

第一个循环中的列表由函数产生，要打印从1到19内的所有偶数。第二个例子打印列表中字符串。

6.6 列表操作符

加号操作符"+"将两个列表连接成一个列表：

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1,2,3,4,5,6]
```

类似的，乘号操作符“*”表示把列表的所有元素重复一定的次数，然后形成一新的列表：

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

6.7 列表片断

通过操作符“[]”，将列表元素连续取出的部分，叫片断。

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3]
['b','c']
>>> list[:4]
['a','b','c','d']
>>> list[3:]
['d','e','f']
>>> list[:]
['a','b','c','d','e','f']
```

只要在“[]”内包含冒号“：”，所取的片断依然是列表，否则，就是元素的原始类型。

```
>>> a = ["python", "Java", "perl"]
>>> type(a[:])
<type 'list'>
>>> type(a[0:0])
```

```
<type 'list'>
>>> type(a[1:])
<type 'list'>
>>> type(a[1:2])
<type 'list'>
>>> type(a[-1])
<type 'str'>
```

6.8 列表元素是可变的

和字符串不一样，列表的元素是可更改的，这意味着我们可以修改、添加或删除列表的元素值。

```
>>> fruit = ["banana", "apple", "quince"]
>>> fruit[0] = "pear"
>>> fruit[-1] = "orange"
>>> print fruit
['pear','apple','orange']
>>> list = ['a','b','c','d','e','f']
>>> list[1:3] = ['x','y']
>>> print list
['a','x','y','d','e','f']
```

也可以通过给元素赋空值，删除一些元素：

```
>>> list=['a','b','c','d','e','f']
>>> list[1:3]=[]
>>> print list
['a','d','e','f']
```

或是利用列表片断，在特定的位置插入元素。

```
>>> list = ['a','d','f']
>>> list[1:] = ['b','c']
>>> print list
['a','b','c','d','f']
>>> list[4:4] = ['e']
>>> print list
['a','b','c','d','e','f']
```

6.9 元素的删除

Python提供了关键字del删除列表中的元素。

```
>>> a=['one', 'two', 'three']
>>> del a[1]
>>> a
['one','three']
```

关键字del可以处理负数索引。如果索引超出范围，就会产生运行时错误。也可以用片断作为关键字del的索引：

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del list[1:5]
>>> print list
['a','f']
```

6.10 变量和值

如果我们执行下面的赋值语句：

```
a = "qswtp"
b = "qswtp"
```

我们知道a和b将指向一个值相同的字符串"banana"。但是不知道它们是否是同一个字符串。可能有两种状态（见图 6.1）：

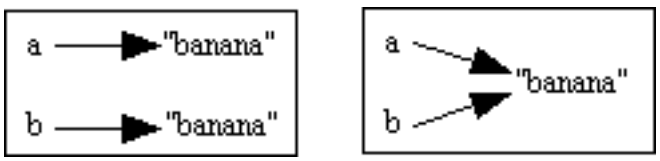

图 6.1: 变量a、b的状态

在第一种情况下，它们是两个不同的字符串，但是有相同的值。第二种情况下，指向同一个字符串，这个字符串有两个不同的名字。

每个变量有唯一的标识符，可以用函数id查看标识符。通过显示a和b的标识符，可以得知a和b是指向同一个字符串。

```
>>> id(a)
135044008
>>> id(b)
135044008
```

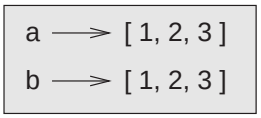

图 6.2: a和b指向不同的对象

实际上，我们得到了相同的标识符，这表明Python只创造了一个字符串，a和b都指向它。

有趣的是，列表与字符串是有区别的。即使列表变量的值相同，他们也是指向不同的列表值。

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```

列表的状态图如下：

整数和浮点数与列表的情况很相似。

6.11 别名

因为变量指向内存中的值，如果我们将一个变量赋值给另一个变量，那末这两个变量指向同一个对象。

```
>>> a = [1,2,3]
>>> b = a
>>> id(a)
7987344
>>> id(b)
7987344
```

变量a和b的状态如图6.3：

因为列表有不同的名字，分别叫做a和b，我们叫它**别名**。利用操作符“[]”修改任何一个列表的元素，另一个列表做相同的改变。如果重新对其中的一个赋值，那么这两个列表的标识符就不在是相同的了，即已经分别属于不同的列表：

```
>>> a = [1, 2, 3]
>>> b = a
```

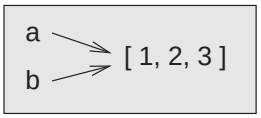

图 6.3: a和b指向相同的对象

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
>>> b = [5, 2, 3]
>>> id(a), id(b)
(13663768, 13615376)
```

6.12 克隆列表

如果要修改列表，但是要保留原来列表的一份拷贝，就需要列表自我复制，这过程叫做克隆。克隆的结果是产生两个值一样，但却有不同标识符的列表。克隆的方法是利用列表的片断操作符：

```
>>> x = [1, 3, 5, 7]
>>> y = x[:]
>>> print y
[1, 3, 5, 7]
>>> y[0] = 9
>>> print y
[9, 3, 5, 7]
>>> print x
[1, 3, 5, 7]
>>> id(x)
13161832
>>> id(y)
13075520
```

利用片断操作符，克隆了整个列表。可以清楚的看到，x和y分别代表不同的列表。修改y的元素值，不影响x 列表。

6.13 列表参数

传递列表参数实际上是传递列表的别名，而不是列表的拷贝。在下面的例子中，函数head以列表为参数，返回列表的第一个参数。

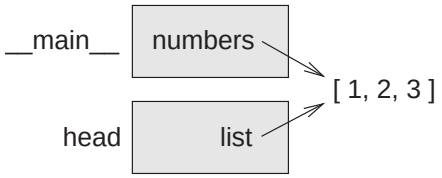

图 6.4: list和numbers指向不同的对象

```
>>> def head(list):
        return list[0]

>>> numbers=[1,2,3]
>>> head(numbers)
1
```

参数list和变量numbers是同一个对象的别名。状态图如 6.4：

由图可以看到列表变量在不同的函数域内。我们可以想象到，如果再函数中更改了列表，也就是更改了列表变量numbers。

```
>>> def deleteHead(list):
        del list[0]

>>> numbers = [1, 3, 5]
>>> deleteHead(numbers)
>>> print numbers
[3,5]
```

如果防止列表变量在函数中被更改，我们可以利用前面讲的克隆列表。请看下面的例子：

```
>>> def delelements(list):
        temp = list[:]
        del temp[0]

>>> numbers = [1,2,3]
>>> delelements(numbers)
>>> print numbers
[1,2,3]
```

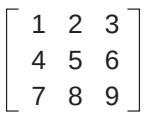

图 6.5: 矩阵

6.14 列表嵌套

嵌套的列表是作为另一个列表中的元素。下面列表中的第三个元素是一个列表。

```
>>> list = ["hello", 2.0, 5, [10, 20]]
```

我们打印列表list[3]，就会得到[10,20]。如果想得到嵌套列表中的元素，可以按照下面的步骤进行：

```
>>> elt = list[3]
>>> elt[0]
10
```

其实列表可以看作是数组，嵌套列表就是多维数组的元素。所以也可以按照下列形式取得元素：

```
>>> list = [0, [1,2,3], [4,5,6]]
>>> list[0][0]
>>> print list[1][0]
1
>>> print list[1][0],list[1][1],list[1][2]
1 2 3
```

6.15 矩阵

嵌套列表可以代表矩阵，例如： 图 6.5的矩阵可以用下面的列表表示：

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

matrix是有三个元素的列表，每个元素代表矩阵的一行。我们可以用常规方法得到矩阵的一行：

```
>>> matrix[1]
[4, 5, 6]
```

还可以用“双索引”的方法得到单一元素：

```
>>> matrix[1][1]
5
```

6.16 字符串和列表

下面我们来介绍两个与字符串和列表相关的函数。一个是将字符串分割成单词列表。空格默认的是单词的边界。

```
>>> import string
>>> song = "The rain in Spain..."
>>> string.split(song)
['The', 'rain', 'in', 'Spain...']
```

split函数的另外一个参数是规定作为分隔符的字符串。分隔符不显示在列表中。

```
>>> string.split(song, 'ai')
['The r', 'n in Sp', 'n...']
```

函数jion的功能和split正好相反，它是将列表连接成字符串。默认的分隔符是空格。

```
>>> list = ['The', 'rain', 'in', 'Spain...']
>>> string.join(list)
'The rain in Spain...'
```

当然也可以像split那样指定分隔符。

```
>>> string.join(list, '_')
'The_rain_in_Spain...'
```

6.17 列表映射

假设有这样包含字符串的列表，要将其中的每个元素的末尾加一字符‘s’。我们会想到如下的办法：

```
>>> fruit = ['apple', 'orange', 'pear', 'banana']
>>> i = 0
>>> for var in fruit:
        fruit[i] = var + 's'
        i = i + 1
>>> print fruit
['apples', 'oranges', 'pears', 'bananas']
```

如果用列表映射，只用一行语句就能解决问题：

```
>>> [fruit + 's' for fruit in fruit]
['apples', 'oranges', 'pears', 'bananas']
```

当for语句每次循环时，把一个元素赋值给变量fruit，然后这个变量再和字符‘s’相加，最后把列表fruit的值全部更新。

上述列表映射对所有元素做了相同的的操作，如果我们只想对满足条件的元素进行处理该怎么办呢？很简单，只要在加上if语句就行了。假如不想对字符个数等于六的元素加‘s’，那么可用下列方法过滤列表：

```
>>> [fruit + 's' for fruit in fruit if len(fruit) != 6]
['apples', 'pears']
```

可见该操作将元素“orange”和“banana”忽略，因为他们不满足字符个数不等于六的条件。

第七章

序列

7.1 序列

到目前为止，我们已经学习了两种复合的数据类型：由字符组成的字符串，由任意类型的元素组成的列表。这两种类型的不同之处在于，列表中的元素能够被修改，而字符串中的字符则不能被修改。换句话说，字符串的值是固定的，列表的值是可变的。

在Python中还有另一种叫做序列的数据类型。它和列表比较相近，只是它的元素的值是固定的。序列中的元素以逗号分隔开。

```
>>> tuple = 'a', 'b', 'c', 'd', 'e'
```

通常情况下，序列用小括号括起来。

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

如果要创造一个包含一个元素的序列，那需要在序列的最后加上逗号。

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

要是不加逗号，就把这个变量当成字符串。

```
>>> t2 = ('a')
>>> type(t2)
<type 'string'>
```

和列表相似，也可以用索引从序列中读取一个元素。

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
>>> tuple[0]
'a'
```

用片断操作符取得列表的一部分：

```
>>> tuple[1:3]
('b', 'c')
```

如果我们试图更改序列的值，解释器会返回错误信息。

```
>>> tuple[0] = 'A'
TypeError: object doesn't support item assignment
```

但是我们可以用另一个方法修改序列中的元素：

```
>>> tuple = ('A',) + tuple[1:]
>>> tuple
('A', 'b', 'c', 'd', 'e')
```

这种方法也适用于字符串：

```
>>> string = "pidaqing"
>>> string = 'P' + string[1:]
>>> print string
Pidaqing
>>> string = string[0:2] + 'D' + string[4:]
>>> print string
PiDqing
```

7.2 序列赋值

在编程中，我们可能要交换两个变量的值。用传统的方法，需要一个临时的中间变量。例如：

```
>>> temp = a
>>> a = b
>>> b = temp
```

Python用序列轻松的解决了这个问题：

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> a, b, c = c, b, a
>>> print a, b, c
3 2 1
```

从这个例子可以看到，右边序列元素的值按照从左到右的顺序赋值给左边的序列元素。如果右边的序列包含表达式，则先进行计算，然后再赋值，如下例：

```
>>> x = 2
>>> y = 3
>>> x, y = x + y, x * y
>>> print x, y
5 6
```

很自然的想到，如果两个序列的元素个数不相等会怎样呢？解释器会报告出错。

```
>>> m = 3
>>> n = 7
>>> m, n = 2, 3, 5
ValueError: unpack tuple of wrong size
```

7.3 序列作为返回值

函数的返回值可以是序列。下面的例子是交换变量值的函数。

```
def swap(x, y):
    return y, x
```

然后我们可以把这个函数赋值给序列。

```
a, b = swap(a, b)
```

当然，用这种方法交换变量值没有什么优点。我们可能习惯用下面的方法：

```
def swap(x, y):      # 错误的函数
    x, y = y, x
```

我们执行一下这个函数，看有什么情况发生：

```
>>> def swap(x, y):
    x, y = y, x
>>> a = 1
>>> b = 2
>>> swap(a, b)
>>> print a, b
1 2
```

我们的原意是交换a和b的值，实际的情况是它们的值没变。这是为什么呢？因为x和a是同一个值的别名，在函数swap中改变x的值后，x和a就彻底没有任何关系了。所以x的改变不能影响变量a。

这种错误叫做语义错误。程序可以执行，没有错误提示，但得到的结果与我们的预期不同。

在此，我们研究一下别名的问题。在Python中如果变量具有相同的值，就可以认为这些变量是同一个值的别名。也就是这些变量指向同一个值。这样做的好处也许是节省了内存空间。但是如果某一个变量改变了它自己的值，那么解释器就开辟出新内存空间存储这个变量的值。这个变量就指向新的值。

```
>>> a = '1234'
>>> b = '1234'
>>> c = '1234'
>>> id(a),id(b),id(c)
(21589680, 21589680, 21589680)
>>> a = 'aaa'
>>> id(a)
21671248
```

7.4 随机函数

Python提供了内建的函数来产生伪随机数，之所以叫伪随机数，是因为它不是数学意义上的随机数。random模块中的函数random能够产生一个值的范围在0.0到1.0之间的浮点数。看下面的例子：

```
>>> import random
>>> for i in range(10):
    x = random.random()
    print x

0.719471873544
0.505680100089
0.694100298971
0.950310250765
0.370379273216
0.14761732812
0.583317059392
0.135645518338
0.39235871116
0.813503714078
```

7.5 随机数列表

我们编写了一个产生随机数列表的函数：randomList。它的参数是一个整数，返回列表的长度等于这个整数。

```
>>> import random
>>> def randomList(n):
s = [0] * n
for i in range(n):
    s[i] = random.random()
return s

>>> randomList(8)
[0.16067655722093033, 0.80172497198506543, 0.43563417769110524,
0.77550762310178989, 0.062999438929851159, 0.55282106935533726,
0.29624064851123899, 0.11623351040588936]
```

产生的随机数是均匀分布的，也就是说每一个值的机率是相等的。函数random产生的随机数范围是从0.0到1.0。如果把这个范围再分成几个部分，那么每部分产生的随机数的个数，从理论上讲，应该是完全相等。下面来验证这个猜想。

7.6 计数

解决像这样问题的好办法是把它分成几个子问题，再寻找子问题的解决办法。我们想计算在给定范围内随机数出现的个数。我们曾写了一个程序，遍历一个字符串，计算给定字符出现的次数。对这个程序作一些修改，使之能够解决现在的问题。这个程序的源代码是：

```
count = 0
for char in fruit:
  if char == 'a':
    count = count + 1
print count
```

第一步：list替换fruit；num替换char。不要着急改变其他部分。

```
count = 0
for num in list:
  if num == 'a':
    count = count + 1
print count
```

第二步：修改测试条件。检查变量num是否出现在变量low和high之间。

```
count = 0 for num in list
  if low < num < high:
    count = count + 1
print count
```

第三步：封装代码在名为inBucket的函数中。参数是list、low和high。

```
def inBucket(list, low, high):
  count = 0
  for num in list:
    if low < num < high:
      count = count + 1
  return count
```

通过拷贝和修改存在的程序，我们很快就写完了一个函数，节约了大量的调试时间。

7.7 分割范围

如果我们测试的范围较大，还算方便。但是范围越小，就越麻烦。例如：

```
bucket1 = inBucket(a, 0.0, 0.25)
bucket2 = inBucket(a, 0.25, 0.5)
bucket3 = inBucket(a, 0.5, 0.75)
bucket4 = inBucket(a, 0.75, 1.0)
```

有两个问题要解决：一个是要保存每次输入的结果；另一个是计算分割的范围。我们用循环计算分割的范围。

```
bucketWidth = 1.0 / numBuckets
for i in range(numBuckets):
    low = i * bucketWidth
    high = low + bucketWidth
    print low, "to", high
```

当numBuckets = 8时，输出是：

```
0.0 to 0.125
0.125 to 0.25
0.25 to 0.375
```

```
0.375 to 0.5
0.5 to 0.625
0.625 to 0.75
0.75 to 0.875
0.875 to 1.0
```

现在回到第一个问题，用一个列表存储这八个整数结果。

```
numBuckets = 8
buckets = [0] * numBuckets
bucketWidth = 1.0 / numBuckets
for i in range(numBuckets):
    low = i * bucketWidth
    high = low + bucketWidth
    buckets[i] = inBucket(list, low, high)
print buckets
```

通过把一个大问题分解成几个小问题，再逐个解决这些小问题，最后就把大问题解决了。这种方法我称之为“个个击破”。

第八章

字典

到目前为止，我们已经学习了三种复合数据类型：字符串、列表和序列。它们用整数作为索引。如果你试图用其它类型做索引，就会产生错误。

```
>>> list = [1, 2, 4]
>>> list[0]
1
>>> list['one']
TypeError: sequence index must be integer
```

字典的索引可以是字符串，除了这一点，它与其它组合类型非常相似。当然，字典的索引也可以是整数。

```
>>> dict1 = {'mother': '妈妈', 'father': '爸爸'}
>>> print dict1
{'father': '爸爸', 'mother': '妈妈'}
>>> dict1 = {1: '妈妈', 2: '爸爸'}
>>> print dict1
{1: '妈妈', 2: '爸爸'}
```

我们可以创造一个空字典，然后再添加元素。

```
>>> eng2sp = {}   #空字典
>>> eng2sp['one'] = 'uno'
>>> eng2sp['two'] = 'dos'
>>> print eng2sp
{'one': 'uno', 'two': 'dos'}
```

字典的元素以逗号为分隔符，每个元素包含键和键值，他俩用冒号分隔。

```
>>> dict = {'one': 1, 'two': 2, 'three': 3}
>>> print dict
{'three': 3, 'two': 2, 'one': 1}
```

我们看到键值的顺序与创建时的顺序不同。其实，我们不必太在意键值顺序。因为我们是利用键浏览与之对应的值。

```
>>> print dict['two']
2
```

8.1 字典操作

函数del删除字典中的元素。例如下面的例子可以看作水果的库存。

```
>>> inventory = {'apples': 430, 'bananas': 312,\
                 'oranges': 525, 'pears': 217}
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 217, 'bananas': 312}
```

如果pears卖完了，可以把它从字典中删除。

```
>>> del inventory['pears']
>>> print inventory
{'oranges': 525, 'apples': 430, 'bananas': 312}
```

如果我们希望以后再买pears，但现在库存没有，就把它赋值为零。

```
>>> inventory['pears'] = 0
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 0, 'bananas': 312}
```

如果你想删除所有的元素，可用clear方法：

```
>>> inventory.clear()
{}
```

函数len则返回字典元素的数量。

```
>>> os = {1: 'Linux', 2: 'Uinx', 3: 'FreeBSD'}
>>> len(os)
3
```

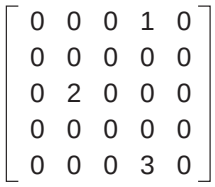

图 8.1: 稀疏矩阵

8.2 别名和拷贝

字典是可变的。如果你想修改字典，并且保留原来的备份，就要用到字典的copy方法。看下面的例子：

```
>>> opposites = {'up': 'down', 'right': 'wrong', \
                 'true': 'false'}
>>> alias = opposites
>>> copy = opposites.copy()
```

alias和opposites指向同一个值。而copy则指向全新的拷贝。如果修改alias，opposites也发生变化。

```
>>> alias['right'] = 'left'
>>> opposites['right']
'left'
```

但是如果修改copy，opposites不变。

```
>>> copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

8.3 稀疏矩阵

如图8.1表示的是稀疏矩阵： 用列表表示如下：

```
matrix = [ [0,0,0,1,0],
           [0,0,0,0,0],
           [0,2,0,0,0],
           [0,0,0,0,0],
           [0,0,0,3,0] ]
```

也可以用字典表示矩阵。该矩阵的非零元素的键用含有两个整数元素的序列表示，分别代表行和列。

```
matrix = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

仅仅需要三个键值对表示矩阵的非零值。每个键的类型是数组，键值是整数。用这种方法，我们不能得到值为零的元素，因为这个矩阵中，没有非零值的键。

```
>>> matrix = {(0,3): 1, (2, 1): 2, (4, 3): 3}
>>> matrix[2, 1]
2
>>> matrix[2, 2]
KeyError: (2, 2)
```

get方法解决了这个问题。

```
>>> matrix.get((1, 3), 0)
0
```

第一个参数是键；第二个参数表示：如果该键没有出现在字典中，那末这个键的值就是第二个参数。

8.4 暗示

请看下面的函数：

```
def fibonacci (n):
  if n == 0 or n == 1:
    return 1
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

你可能注意到：n的值约大，程序运行的时间就越长。当n等于32时，大约运行25秒。为什么会这样呢？看一下函数的调用图8.2： 函数出现了重复的调用。比如n=2就出现了两次。所以这是没有效率的解决方法。当n变大时，情况变得更糟。一个好的解决办法是将已经运算完的结果保存在字典中，以备以后所需。

```
previous = {0:1, 1:1}
def fibonacci(n):
  if previous.has_key(n):
    return previous[n]
  else:
    newValue = fibonacci(n-1) + fibonacci(n-2)
    previous[n] = newValue
    return newValue
```

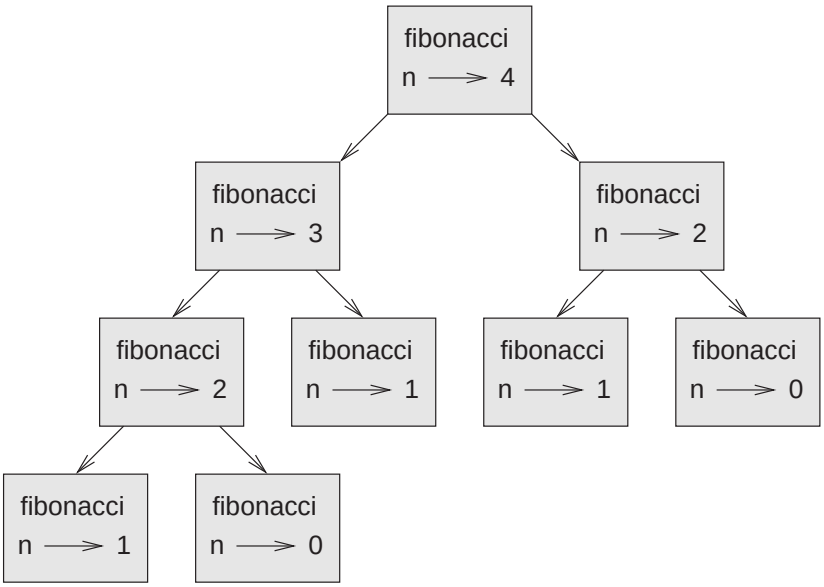

图 8.2: n=4时，函数fibonacci的调用图

字典首先定义了当n=0,1时的值。当函数fibonacci被调用，先检查字典中是否包含要计算的结果。如果有就立刻返回结果，不再做递归调用。若没有，就得计算新值，并且新值在函数返回前加入到字典中。

用这个版本的fibonacci函数，我们的计算机能够瞬间计算n=40的值。要使用老版本的fibonacci函数，你必须耐心等待。当我们计算n=50时，会得到一个错误：

```
>>> fibonacci(50)
OverflowError: integer addition
```

python用一种叫做长整数的类型处理任意大小的整数。通常，我们用整数后面加一个大写的L 表示长整数。

```
>>> type(3L)
<type 'long'>
```

另一种方法是用函数long把任意的数字类型，即使是数字字符窜，转换成长整数。

```
>>> long(34)
34L
>>> long(3.4)
3L
>>> long('34')
34L
```

所有的数学操作符都适用于长整数。因此，对于上面的函数fibonacci不必做太多的更改，就能正常运行了。

```
>>> previous = {0:1L, 1:1L}
>>> fibonacci(50)
20365011074
```

8.5 计算字符串

我们曾写过一个函数,目的是计算字符串中字母出现的次数。而字典提供了一个很好的方法，来统计字母出现的次数。

```
>>> letterCounts = {}
>>> for letter in "Mississippi":
      letterCounts[letter] = letterCounts.get (letter, 0) + 1
>>> print letterCounts
{'i': 4, 'p': 2, 's': 4, 'M': 1}
```

Python有两个函数items和sort能够更好的完成这一功能。

```
letterItem = letterCounts.items()
print letterItem
[('i', 4), ('p', 2), ('s', 4), ('M', 1)]
letterItem.sort()
print letterItem
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

第九章

文件

9.1 文件的打开和关闭

程序运行的时候，相关数据是保存在内存中的，关闭计算机电源，内存中的数据就丢失了。为了永久的保存数据，必须将数据存储于文件内，文件通常是保存在硬盘、软盘或光盘中。

打开文件就是创造了文件对象。下面的例子中，变量f指向一个文件对象。

```
>>> f = open("test.dat", "w")
>>> print f
<open file 'test.dat', mode 'w' at 0x0151F8A8>
```

函数open的第一个参数是文件名，第二个是打开的模式。"w"表示以写入的方式打开文件。如果文件"test.dat"不存在，open函数就创建它，否则，新创建的文件覆盖已经存在的文件。当我们打印文件对象时，显示了文件名、模式和对象位置等信息。

为了把一些内容写到文件中，需要调用文件对象的write方法：

```
>>> f.write("Now is the time")
>>> f.write("to close the file")
>>> f.close()
```

文件的方法close关闭文件。我们向文件test.dat写入了两个字符串，这两个字符串以如下形式存在于文件的同一行：

```
Now is the timeto close the file
```

如果要让这两个字符串分别处于一行，则应该在第一个字符串后面加上换行符：

```
f.write("Now is the time\n")
```

现在可以再一次的打开文件，但这次不是以写的方式，而是以读的方式，即“r”。

```
f = open("test.dat", "r")
```

如果要以读的方式打开一个不存在的文件，解释器会显示错误信息：

```
>>> f = open("tes.dat", "r")
IOError: [Errno 2] No such file or directory: 'tes.dat'
```

当创建或读取文件时，Python首先在当前的目录进行工作，默认的工作目录是C:/python22。若想在别的目录下工作，就必须指明完整的路径。例如：

```
>>> f = open("c:\\test.dat", "w")
>>> f.write("OK")
>>> f.close()
```

为了把文件的内容写到字符串中，可以用文件对象的read方法。

```
text = f.read()
print text
Now is the timeto close the file
```

因为在”time”和”to”之间没有写入空格，所以这两个词连在了一起。

read可以接受数字参数，表示读出一定数量的字符。如果读到文件的末尾，就返回空字符。

```
>>> f = open("test.dat", "r")
>>> print f.read(5)
Now i
>>> print f.read(100)
s the timeto close the file
>>> print f.read()

>>>
```

接下来的函数是拷贝文件，一次读写五十个字符。第一个参数是源文件名，第二个参数是新文件名。

```
def copyFile(oldFile, newFile):
    f1 = open(oldFile, "r")
    f2 = open(newFile, "w")
    while 1:
        text = f1.read(50)
        if text == "":
            break
        f2.write(text)
    f1.close()
    f2.close()
    return
```

break表示是当没有字符拷贝，也就是"text"为空字符时，彻底跳出"while"循环，这也是唯一结束"while"循环的方式。

9.2 文本文件

文本文件包含可打印字符和空格，每一行以换行符为结束标志。Python是处理文本文件的行家里手。下面的例子建立了一个文本文件，它有三行：

```
>>> f = open("test.dat", "w")
>>> f.write("I love Python!\nHello, world\nGood Bye!\n")
>>> f.close()
```

readline方法每次从文本文件中读取一行的内容，包括换行符：

```
>>> f = open("test.dat", "r")
>>> print f.readline()
I love Python!

>>>
```

readlines方法以列表的方法返回文件内其余的内容：

```
>>> print f.readlines()
['Hello, world\n', 'Good Bye!\n']
```

这时已到达文件的末尾，如果在调用上述两个方法，readline 方法返回空字符，readlines方法返回空列表：

```
>>> print f.readline()

>>> print f.readlines()
[]
```

接下来的例子是行处理程序，函数filterFile拷贝一个文件，同时将旧文件中不是以“#”开头的行写入新文件中：

```
def filterFile(old, new):
    sfile = open(old, "r")
    dfile = open(new, "w")
    while 1:
        text = sfile.readline()
        if text == "":
            break
        elif text[0] == "#":
            continue
        else:
            dfile.write(text)
    sfile.close()
    dfile.close()
```

在这个函数里有两个关键字：break和continue。break 表示如果读取的行内容为空，则完全终止while 的循环；continue 表示若行的首字符为“#”，则终止这一次的循环，也就是continue 以下的循环内的语句不执行了，返回到循环的顶部继续下一次循环。

9.3 写入变量

write的参数只能是字符串，如果想把其它类型的变量写入文件，就必须将其转换成字符串。一个简单的方法是利用str函数：

```
>>> f = open("test.dat", "w")
>>> f.write(str(1234.56) + '\n')
>>> f.write(str(1000))
>>> f.close()
```

另一种方法是利用格式化操作符“%”。当操作符“%”的两边是整数时，它是求余数的运算。如果第一个操作符是字符串，它就是格式化操作符。第一个参数是需要格式化的字符串，第二个参数是数组表达式。结果是包含表达式值的字符串。请看下面的例子：

```
>>> age = 31
>>> "%d" % age
'31'
```

格式化序列“%d”表示数组中的第一个表达式的值应该是整数类型，字母d代表“decimal”。

格式化序列可以出现在字符串的任何位置，所以我们能够在句子中嵌入值：

```
>>> age = 31
>>> "My age is : %d." % age
'My age is : 31.'
```

格式化序列“%f”对应是浮点数，默认的小数点后面有六位小数。“%s”对应是字符串。表达式要与字符串中的格式化序列相匹配，匹配包含两个方面，一个是有几个表达式，就有几个格式化序列；另一个是格式化序列与表达式值的类型相对应。

```
>>> "In %d days we make %f million %s." % \
     (31, 31*12.59, 'dollars')
'In 31 days we make 390.290000 million dollars.'
>>> "d% d% d%" % (3, 4 ,5 ,6)
ValueError: incomplete format            #错误信息
>>> "d%" % 8.0
ValueError: incomplete format            #错误信息
```

对于要格式化的数字，我们还能够指定它所占的位数。“%”后面的数字表明数字的位数，如果位数多于数字的实际位数，且该数为正，则在要格式化的数字的前面添加空格；如果该数为负，空格添加在数字的后面：

```
>>> "%5d" % 1
'    1'
>>> "%-5d" % 1
'1    '
>>> "%4f" % 9.1
'9.100000'
>>> "%3f" % 1234
'1234.000000'
>>> "%3d" % 1234
'1234'
```

对于浮点数，我们还可以指定小数的位数：

```
>>> "%4.3f" % 1137.98
'1137.980'
```

下面的例子是按照一定的格式打印姓名和工资。姓名是字典的键，工资是字典的值。姓名左对齐，工资右对齐,同时对姓名进行了排序：

```
>>> def printSalary(salary):
    name = salary.keys()
    name.sort()
    for n in name:
        print "%-12s : %12.2f" % (n,salary[n])

>>> salary = {'pidaqing':1137.9, 'zhangming':737.3, 'pitianjian':5.0}
>>> printSalary(salary)
pidaqing     :      1137.90
pitianjian   :         5.00
zhangming    :       737.30
```

从上述的内容可以看到，为了将不同类型的数据保存到文件，必须将其转换成字符串。结果导致从文件中读的一切内容都是字符串，数据的原始类型信息丢失了。解决的办法是输入pickle模块，用它提供的方法把各种类型的数据存入文件，数据结构的信息也同样被保存了。也就是说，你保存了什么，将来读出的还是什么。例如：

```
>>> import pickle
>>> f = open("test.dat", "w")
>>> pickle.dump(100, f)
>>> pickle.dump(123.98, f)
>>> pickle.dump((1, 3, "abc"), f)
>>> pickle.dump([4, 5, 7], f)
>>> f.close()
```

在这个例子中，我们用dump方法分别向文件中写入了整数、浮点数、列表和数组。如果你用write方法写入，那是会出错的。

接下来就看能不能把这些数据“原封不动”的读出来：

```
>>> f = open("test.dat", "r")
>>> a = pickle.load(f)
>>> print a
100
>>> type(a)
<type 'int'>
>>> b = pickle.load(f)
>>> print b
123.98
>>> type(b)
```

```
<type 'float'>
>>> c = pickle.load(f)
>>> print c
(1, 3, 'abc')
>>> type(c)
<type 'tuple'>
>>> d = pickle.load(f)
>>> print d
[4, 5, 7]
>>> type(d)
<type 'list'>
```

每调用一次load方法，就得到先前存入的一个变量，而且这个变量还保存着原始类型的信息。

第十章

异常

10.1 错误信息

程序出错了，就会产生异常。异常是不可避免的，关键是怎样处理。当然不能任其放任自流，不管不问。起码的要求是解释器终止程序的运行，指出错误类型，以及对错误进行简单描述。

例如，除数为零时，IDE产生一个异常：

```
>>> print 2/0
ZeroDivisionError: integer division or modulo by zero  #异常信息
```

异常信息分为两个部分，冒号前面的是异常类型，之后是对此的简单说明。其它的信息则指出在程序的什么地方出错了。当异常产生时，如果没有代码来处理它，Python对其进行缺省处理，输出一些异常信息并终止程序。

程序在执行的过程中产生异常，但不希望程序终止执行，这时就需要用try和except 语句对异常进行处理。比如，提示用户输入文件名，然后打开文件。若文件不存在，我们也不想程序就此崩溃，处理异常处理就成了关键的部分。

```
filename = ''
while 1:
    filename = raw_input("Input a file name: ")
    if filename == 'q':
        break
    try:
        f = open(filename, "r")
        print 'Opened a file.'
    except:
```

```
            print 'There is no file named', filename
```

try块的语句要求打开一个文件，如果没有异常发生，就忽略except块的内容；如果产生异常，就执行except块内的语句，之后是再一次的循环。

10.2 自定义异常信息

如果程序检测到错误，我们也可以用raise定义异常。

```
def inputAge():
    age = input("Input your age:")
    if (age>100 or age<18):
        raise 'BadNumberError', 'out of range'
    return age
```

raise有两个参数，第一个是由我们自己定义的异常类型，第二个是关于此异常的少量说明信息。如果调用inputAge的函数有处理异常的程序，即使inputAge出错，整个程序也能正常运行；否则，程序退出，显示错误信息。

```
>>> inputAge()
Input your age:31
31
>>> inputAge()
Input your age:109
BadNumberError: out of range  #异常信息
```

10.3 一个复杂的例子

让我们看下面的脚本文件：

```
while 1:
    try:
        x = int(raw_input("Input a number:"))
        y = int(raw_input("Input a number:"))
        z = x / y
    except ValueError, ev:
        print "That is no valid number.", ev
    except ZeroDivisionError, ez:
        print "divisor is zero:", ez
    except:
```

```
            print "Unexpected error."
            raise
        else:
            print "There is no error."

    print x , "/" , y , "=" , x/y
```

在这个例子中有三个except语句。一个try语句可以和多个except配合使用，但只可能是其中的一个被执行。

前两个except语句，接受两个参数。第一个参数是异常的类型，第二个参数用于接收异常发生时生成的值，异常是否有这个参数及参数的类型如何，由异常的类型决定。

异常“ValueError”在这里表示：如果你输入的字符串包含非数字类型的字符，这个异常将被引发。异常“ZeroDivisionError”表示除数为0引发的异常。

最后一个except语句表示当有异常发生，但不是前面定义的两种类型，就执行这条语句。用这样的except 语句要小心，理由是你很可能把一个应该注意的的程序错误隐藏了。为了防止这种情况的发生，我们用了raise语句，将异常抛出。

当没有任何异常发生时，else语句的内容被执行。else语句一定放在所有except语句的后面。

第十一章

类和对象

11.1 用户定义数据类型

以前学过的数据类型都是Python内置的，从现在开始我们要自己定义数据类型：点（point）。

在数学上，点的位置由两个坐标决定。例如，（0,0）代表坐标原点，（x,y）代表行轴为x、纵轴为y的点。在Python中，一个自然的方法是用两个浮点数表示点的横坐标和纵坐标。而问题的关键是如何把这两个浮点数组合起来，形成一个新的数据对象，用列表，或是数组，对于某些方面的应用来说，这也许是上佳之选。

另外一个选择是将点定义成**类（Class）**，即用户为自己的应用定义的数据类型。类的好处会随着学习的深入，逐渐被我们体会到。

类的定义方式如下：

```
class Point:
    pass
```

关键字class表示Point是我们定义的类，也就是向程序声明了一种新的数据类型。类Point的实质内容暂时没有，但又不能为空，所以用pass替代，以便将来添加。类的定义可以放在程序的任何地方，通常是放在程序开始部分，import语句之后。

类只是对实体的一种抽象和概括。打个比方，自然界包括动物和植物，动物这个概念是对千千万万个体的共有特征的一种描述，你不可能找到一只叫“动物”的动物，但我们能够找到你养的一只狗，动物园里的一只猴子，这些都是实实在在存在的个体。类的概念与之非常相似。点类只是无数个实际点的抽象，这些无数个实际点是点类的实例。点类需要实例化才能生成一个实际的点。如果创建一个点的对象，需调用名为Point的函数：

```
blank = Point()
```

变量blank指向一个实际的点对象。函数Point创造点对象的过程叫做构造。

11.2 属性

上节创造了一个点对象，但这个对象除了名字，没有任何内容。现在利用点操作符给这个点对象添加了两个数据项：横坐标和纵坐标。这两个数据项叫做点的属性。

```
>>> class Point:
pass

>>> blank = Point()
>>> blank.x = 3.0
>>> blank.y = 4.0
```

利用点操作符我们也可以打印点的属性：

```
>>> print blank.x
3.0
>>> print blank.y
4.0
```

blank.x表示的是对象blank的x属性，即点blank的横坐标，blank和x通过点操作符组成一个不可分割的整体，代表blank的属性。它们可以作为表达式的一部分，参加运算。

```
>>> import math
>>> distance = math.sqrt(blank.x * blank.x + blank.y * blank.y)
>>> print distance
5.0
```

也可以打印blank对象本身：

```
>>> print blank
<__main__.Point instance at 0x00C76980>
```

结果显示blank是点类Point的一个实例，它被定义在_main_中。0x00C76980是这个对象的唯一标识符，用十六进制表示。

我们也可以用type函数测试一下Piont和blank。

```
>>> type(blank)
<type 'instance'>
>>> type(Point)
<type 'class'>
```

可以看出，blank是一个实例，Point是类。

blank是一个实例，也是一个变量，所以对象可作为函数的参数：

```
>>> def printPoint(p):
        print '(' + str(p.x) + ',' + str(p.y) + ')'

>>> printPoint(blank)
(3.0,4.0)
```

这个函数以点的一个实例为参数，打印该点的行、纵坐标。

11.3 同一性

英文单词“same”的意思是同一的、相同的，这似乎没有异议，如果我们再深入探讨一下，可能期望与所想并不一致。

例如，你说，“Chris and I have the same car”，你的意思是Chris和我的车为同一个厂家，同一个型号，但却是两辆不同车。如果你说，“Chris and I have the same mother”，这里意思是Chris和我的妈妈是同一个人，可不是两个不同的妈妈。由此可见，“Sameness”的真正意思要根据上下文来判断。

当我们讨论对象时，也同样存在这样的模糊性。例如，我们说两个相同的点，可能这两个点的数据（横、纵坐标）相同，也可能他们就是同一个对象。我们可以利用“==”操作符判断两个点是否为同一对象：

```
>>> p1 = Point()
>>> p1.x = 1.0
>>> p1.x = 9.0
>>> p2 = Point()
>>> p2.x = 1.0
>>> p2.y = 9.0
>>> p1 == p2
0
```

从这个例子可以看到，点p1和p2虽然有相同的数据，但却是两个不相同的对象。若是把p2赋值给p1，那么p1和p2就是同一对象的两个别名。

```
>>> p1 = p2
>>> p1 == p2
1
```

11.4 长方形类

现在我们要建一个长方形类，首先要寻找一些信息，来对长方形进行数学描述。根据具体的应用不同，会存在下面几种定义：

- 长方形的中心点的坐标，以及长和宽；
- 一个角的坐标，以及长和宽；
- 两个对角的坐标。

习惯的选择是确定左下角的坐标，以及长和宽。下面分三步定义一个长方形对象：

- 定义一个类：

```
>>> class Rectangle:
        pass
```

- 初始化一个长方形对象，并定义它的长和宽：

```
>>> r1 = Rectangle()
>>> r1.width = 10
>>> r1.height = 20
```

- 把长方形的左下角坐标定义为一个点对象：

```
>>> r1.corner = Point()
>>> r1.corner.x = 1
>>> r1.corner.y = 1
```

此时，这个对象的状态图如：(见图 11.1)

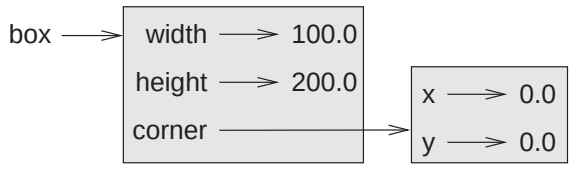

图 11.1: 长方形对象r的状态图

函数的返回值可以是对象。下面的程序计算长方形的右上定点的坐标，放回值是一个点。我把它存为一个脚本文件。

```
class Point:
    pass
class Rectangle:
    pass

def findUpperRight(rectangle):
    p = Point()
    p.x = rectangle.width + rectangle.corner.x
    p.y = rectangle.height + rectangle.corner.y
    return p

r = Rectangle()
r.width = 50.0
r.height = 70.00
r.corner = Point()
r.corner.x = 10
r.corner.y = 5

up = findUpperRight(r)
print '(' + str(up.x) + ',' + str(up.y) + ')'
```

脚本的执行结果是：

```
E:\python>python upper.py
(60.0,75.0)
```

我们能够通过改变对象的属性来改变对象的状态。例如，我们可以改变长方形的长和宽，而不改变它的位置。

```
>>> rect = Rectangle()
>>> rect.width = 1.0
>>> rect.height = 2.0
>>> print rect.width, rect.height
1.0 2.0
>>> rect.width = rect.width + 3.0
>>> rect.height = rect.height + 4.0
>>> print rect.width, rect.height
4.0 6.0
```

11.5 拷贝

别名增加了程序阅读的困难，因为在一个地方改变了对象的值，但在另一个地方可能是不希望的。并且追踪对象所有的别名，又是很困难的。

解决的方法是拷贝对象，生成一个新的对象实例。copy模块的方法copy能够复制任何对象。

```
>>> p1 = Point()
>>> p1.x = 2.0
>>> p1.y = 4.0
>>> p2 = p1
>>> print p1,p2
<__main__.Point instance at 0x00D76878>
<__main__.Point instance at 0x00D76878>
>>> p2 = copy.copy(p1)
>>> print p1, p2
<__main__.Point instance at 0x00D76878>
<__main__.Point instance at 0x00D009A0>
>>> print p1.x,p1.y
2.0 4.0
>>> print p2.x,p2.y
2.0 4.0
```

输入copy模块之后，我们用copy方法复制了一个新的点对象，但他们数据项的值相同。

一些简单的对象，如点，没有包含任何嵌入的对象，copy方法已经足够了。这种复制叫做浅拷贝。

对于象长方形类的对象，它的属性中包含点对象，再用copy方法进行复制，虽然生成了新的长方形对象，但这两个对象的点对象是同一个对象：

```
>>> r1 = Rectangle()
>>> r1.width = 1.0
>>> r1.height = 2.0
>>> r1.corner = Point()
>>> r1.corner.x = 0
>>> r1.corner.y = 0
>>> r2 = copy.copy(r1)
>>> print r1.corner, r2.corner
<__main__.Point instance at 0x00D44BA8>
<__main__.Point instance at 0x00D44BA8>
```

显然这也不是我们希望的结果。

幸运的是，copy模块包含了一个名为deepcopy 的方法，它可以拷贝任何嵌入的对象。这种拷贝我们称之为深拷贝。

```
>>> r2 = copy.deepcopy(r1)
>>> id(r1)
13831952
>>> id(r2)
14043608
>>> id(r1.corner)
13913000
>>> id(r2.corner)
14003528
```

b1和b2已经是完全不同的对象了。

第十二章

类与方法

12.1 面向对象的技术

Python是面向对象的编程语言，自然提供了面向对象的编程方法。但要给面向对象的编程方法下一个定义，是很困难的。问题关键是理解对象的含义。对象的含义是广泛的，它是对现实世界和概念世界的抽象、模拟和提炼。

世界本来就是对象的世界，一台电脑、一座大楼、一座大山、2003年6月10日、数学中的一条线，大到宇宙，小到原子，从无形到有形，都可以看作对象。我们自打出生的那一天起，就要面对对象。比如你看新闻联播，了解国内外大事，我们最终的结果是得到信息，但信息不会凭空传递而来，它是靠电视向你传播的。电视机正是你要面对的对象。

对象可以嵌套。比如把我的电脑看作是一个对象，那末它就由以下对象组成：显示器、键盘、鼠标、主机、音箱等。计算机对象是所有这些对象组合而成的。当然其中的每一个对象，还可以细分，例如主机还可以分为机箱、CPU等等。究竟要不要细分，要根据情况来定。如果你给初学计算机的人讲解有关硬件的知识，刚开始，只要把上述几部分介绍就可以了。随着学习的深入，你就必须更详细的介绍每一部分。所以说，对象要不要细分，完全依据需要而定。

对象不是一个空壳，而是有血有肉的实体。它有特点，或曰属性。例如，人的高矮胖瘦就是属性。光有属性还不够，我们还会跑步、听歌和愤怒等等，这些是我们的方法。属性是动，方法是静。方法分为两类，一种是与外界无关的，如跑步；一种是和其他对象交互，如愤怒。

世界是物质运动的结果。对象之间的相互作用，就是对象的运动，即事件。与世隔绝的对象是没有任何价值的。当然你可以在程序中创建一个“老死不相往来”的对象，但它是一种资源浪费，没有实用价值。现实中一项工作的完成，是很多人和物相互协调、相互作用的结果。一个程序要达到的目的，也是很多对象相互作用的结果。

对象的方法与函数类似，但还有两方面的区别：

1. 方法定义在类的内部，是类的一部分，他们之间的关系是很明显的；
2. 调用的语法不一样。

现在，我们定义一个时间类和打印时间的函数：

```
class Time:
    pass

def printTime(time):
    print str(time.hours) + ":" +
          str(time.minutes) + ":" +
          str(time.seconds)
```

时间类有三个属性：hours、minutes和seconds。函数printTime则是显示这三个属性值。调用这个函数时，我们以一个时间对象为参数：

```
>>> now = Time()
>>> now.hours = 10
>>> now.minutes = 30
>>> now.seconds = 10
>>> printTime(now)
```

为了把函数变成类Time的方法，我们只需要将其移到类定义的内部。注意缩进的改变。

```
class Time:
    def printTime(self):
          print str(time.hours) + ":" + \
                str(time.minutes) + ":" + \
                str(time.seconds)
```

这时在调用方法时，就用点操作符：

```
>>> now.printTime()
```

通常，方法的第一个参数是调用它自己：self。在用函数打印时间时，相当于说，“嘿，printTime！打印now对象。”当调用自己的方法时，等于说，“嘿，now！打印你自己。”

在看另一个方法：

```
    def increment(self, seconds):
        self.seconds = seconds + self.seconds

        while self.seconds>=60:
            self.seconds = self.seconds - 60
            self.minutes = self.minutes + 1

        while self.minutes>=60:
            self.minutes = self.minutes - 60
            self.hours = self.hours + 1
```

这个函数有两个参数，一个它自己，一个seconds。函数的意思是，在当前的时间上加上一定的秒数，就得到一个新的时间。例如：

```
now.nicrement(100)
```

函数调用时，第一参数是默认的，不必写出来。

接下来我们在看一个稍微复杂的after方法。after方法判定两个时间哪一个在前。他有两个参数，一个是它自己，另一个是他的同类。

```
    def after(self, time):
        if self.hour>time.hour:
            return 1
        if self.hour<time.hour:
            return 0

        if self.minute > time.minute:
            return 1
        if self.minute < time.minute:
            return 0

        if self.second > time.second:
            return 1
    return 0
```

12.2 可选择的参数

我们曾见到一些内置的函数，能够接受的参数个数是可变的。同样，你也能够定义可变参数的函数。下面这个函数求从数head开始，到tail为尾，步长为step的所有数之和。

```
def total(head, tail, step):
    temp = 0
    while head<=tail:
        temp = temp + head
        head = head + step
    return temp
```

它有三个参数，调用函数时，三个参数都不能省略。

有的参数之所以可以省略，是因为函数中已经给出了缺省的参数。我们把上面的函数作一下改造，定义step的缺省参数为1：

```
total(head, tail, step=1)
```

现在调用total函数时，step参数就可以省略，但不表示step不存在，它的值是默认的1。当然你也可以根据需要取其他值。如下：

```
>>> print total(1, 100)
5050
>>> print total(1, 100, 2)
2500
```

缺省参数的定义要符合以下规则：缺省参数全部位于参数表的后部，而且缺省参数之间不能在有非缺省参数。下面定义是不合法的：

```
total(head=1, tail, step)          #错误的定义
total(head=1, tail, step=1)        #错误的定义
total(head, tail=100, step)        #错误的定义
```

12.3 构造函数

构造函数是任何类都有的特殊方法。当要创建一个类时，就调用构造函数。它的名字是：__init__ 。init的前后分别是两个下划线字符。时间类Time的构造函数如下：

```
class Time:
    def __init__(self, hours=0, minutes=0, seconds=0):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds
```

当我们调用Time的构造函数时，参数依次传递给__init__：

```
>>> now = Time(12, 10, 30)
>>> now.printTime()
12:10:30
```

因为Time的构造函数都有缺省值，所以在创建类时，可以全部忽略他们：

```
>>> now = Time()
>>> now.printTime()
>>> 0:0:0
```

还可以传递一个或两个参数：

```
>>> now = Time(23)
>>> now.printTime()
23:0:0
>>> now = Time(12, 10)
>>> now.printTime()
12:10:0
```

最后，我们还可以传递参数子集：

```
>>> now = Time(seconds = 35, hours = 19)
>>> now.printTime()
>>> 19:0:35
```

这种情况下，并不强调参数一定要符合定义时的顺序。

第十三章

操作符重定义

13.1 加减法重定义

Python的基本数据类型如整数、浮点数，能够进行数学运算。类可以吗？看下例：

```
>>> class RMB:
        def __init__(self, sum = 0.0):
            self.sum = sum
        def __str__(self):
            return str(self.sum)
>>> a = RMB()
>>> b = RMB()
>>> a + b
TypeError: unsupported operand types for
+: 'instance' and 'instance'      #异常信息
```

我们定义了一个人民币类，然后生成两个实例，这两个实例相加之后，显示了一条错误信息，表明实例不能进行相加操作。错误信息的意思是，加法操作暂时不支持两个实例操作数。

为了使类的实例也可以进行数学操作，我们需要在类的内部重新定义数学操作符，使之支持用户定义的数据类型。有些朋友可能马上意识到，这不是C++中的操作符重载吗？在这里我没有用“重载”这个词，我认为C++中重载要比Python中的重载概念广泛得多，例如C++函数重载，在Python中并不存在这个概念（也许我不知道）。因此，我用了重定义一词。下面得例子中，修改了RMB类，添加了RMB类的加法和减法操作，也就是RMB类的两个方法：__add__和__sub__。

```
>>> class RMB:
        def __init__(self, sum = 0.0):
```

```
        self.sum = sum

    def __str__(self):
        return str(self.sum)

    def __add__(self, other):       #重定义加法操作
        return RMB(self.sum + other.sum)

    def __sub__(self, other):       #重定义减法操作
        return RMB(self.sum - other.sum)
```

通常情况下，第一个参数是调用__add__ 或__sub__方法的对象。第二个参数是other，以区别于第一个参数self。两个实例的相加或相减，也就是各自属性sum的加减，所得的值作为新RMB实例的参数，因此返回值仍是人民币类。请看执行结果：

```
>>> a = RMB(20000)
>>> b = RMB(234.987)
>>> print a + b
20234.987
>>>print a - b
19765.013
```

其实表达式a + b和p1.__add__(p2)是等同的，我们可以试一试：

```
>>> a = RMB(34.5)
>>> b = RMB(345.98)
>>> print a.__add__(b), a + b
380.48 380.48
>>> print a.__sub__(b), a - b
-311.48 -311.48
```

13.2 乘法重定义

与乘法重定义相关的方法有两个，一个是__mul__，另一个是__rmul__。你可以在类中定义其中的一个，也可以两个都定义。

如果乘法操作符“*”的左右操作数都是用户定义的数据类型，那么调用__mul__。若左边的操作数是原始数据类型，而右边是用户定义数据类型，则调用__rmul__。

请看例子：

```
class Line:
   def __init__(self, length = 0.0):
        self.length = length

   def __str__(self):
        return str(self.length)

    def __mul__(self, other):  #乘法重定义
        return Rect(self.length, other.length)

    def __rmul__(self, other): #乘法重定义
        return Line(self.length * other)

class Rect:
    def __init__(self, width = 0.0, length = 0.0):
        self.width = width
        self.length = length

    def __str__(self):
        return '(' + str(self.length) + ','
                   + str(self.width) + ')'

    def area(self):     #计算长方形的面积
        return self.width * self.length
```

在这个例子中，定义了Line（线类）和Rect（长方形类）。重定义了Line类的乘法操作符。第一个重定义表示两个线实例相乘，得到并返回长方形实例。第二个重定义表示一个线实例乘以一个原始数据类型，得到并返回一个线实例。执行结果如下：

```
>>> aline = Line(5.87)
>>> bline = 2.9 * Line(8.34)
>>> print 'aline = ' , aline, 'bline = ',bline
aline =  5.87 bline =  24.186
>>> rect = aline * bline
>>> print rect
(24.186,5.87)
>>> print rect.area()
141.97182
```

乘法重定义的第二种形式，必须严格按照“2.9 * Line(8.34)”的书写顺序，要是调换一下顺序，解释器将给出错误信息。

```
>>> bline = Line(8.34) * 2.9
AttributeError: 'float' object has no attribute 'length'
```

第十四章

继承

14.1 继承

面向对象程序语言的一个重要特点是继承。继承提供了在已存在类的基础上创建新类的方法。继承的子类 拥有被继承的父类的所有方法，在此基础上，子类还可以添加自己的专有方法。

继承是类的强有力的特点。一些程序不用继承将会非常复杂，用了继承，写起来就会简单扼要，通俗易懂。另外，通过继承，还可以重新利用以前编写的代码，因为你可以对父类的方法进行定制，而并非一定要修改它们。有时，继承反映了问题的自然结构，这样使程序更容易理解。

任何事情都有双重性。继承也有可能使程序变得难以阅读。调用一个方法时，有时很难判断它是在哪定义的。相关的代码可能分散在几个模块中。所以，有些事情利用继承的特点能够做好，不利用继承，也可能做的不错。如果问题的自然结构不倾向于用继承解决，那么就不要用，用了反而不好。

14.2 继承的定义

假设已经定义了一个父类BaseClass，那么子类的定义方式如下：

```
class DerivedClass(BaseClass):
    ......
```

子类别的实例化方式没有特别之处。如果要引用子类的某个属性，首先在子类中寻找，没有就去到父类中寻找它的定义，在没有的话，就一直向上找下去，知道找到为止。

方法的寻找方式与属性相同。子类的方法可以重定义父类的方法，要是你觉得父类的方法不能满足要求的话。有时候，子类的方法中可以直接调用父类中的方法，方式如下：

```
BaseClass.method(self, arguments)
```

14.3 定义一个父类

```
class Person:
    def __init__(self,
                 name = None,
                 age = 1,
                 sex = "men"):
        self.name = name
        self.age = age
        self.sex = sex
    def displayInfo(self):
        print "name   : %-20s" % self.name
        print "age    : %-20d" % self.age
        print "sex    : %-20s" % self.sex
```

这个Person类有三个属性，方法displayInfo用于打印这三个属性。

14.4 继承Person的子类

接着定义了继承Person的student类：

```
class Student(Person):
    def __init__(self,
                 name = None,
                 age = 1,
                 sex = "men",
                 grade = 0):
        Person.__init__(self, name, age, sex)
        self.grade = grade
    def displayInfo(self):
        Person.displayInfo(self)
        print "grade  : %-20d" % self.grade
```

Student类中的__init__方法中调用了父类的__init__方法，同时加了一条打印给grade属性赋值的语句。Student类中重定义了父类displayInfo方法，它的内部也调用了父类的displayInfo方法。

14.5 私有方法

在C＋＋语言中有私有方法的概念，私有方法只能被类的内部方法调用。在Python 中，类的私有方法和私有属性，不能够从类的外面调用。类的方法和属性是公有，还是私有，可以从它的名字判断。如果名字是以两个下划线开始，但并不是以两个下划线结束，则是私有的。其余的都是公有的。请看下面的例子：

```
import math

class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def __str__(self):
        return '(' + str(self.x) + ',' + str(self.y) + ')'
class Line:
    def __init__(self, p1 = Point(), p2 = Point()):
        self.__p1 = p1            #私有属性
        self.__p2 = p2
    def __str__(self):
        return str(self.__p1) + str(self.__p2)
    def __distance(self):             #私有方法
        tx = math.pow(self.__p1.x, 2) + math.pow(self.__p2.x, 2)
        ty = math.pow(self.__p1.y, 2) + math.pow(self.__p2.y, 2)
        return math.sqrt(tx + ty)
    def length(self):
        print self.__distance()
```

在这个例子中，Line类的两个属性p1和p2是私有的属性，而方法__distance是私有方法。如果试图用如下的语句调用他们，将显示错误信息：

```
>>> Line().__p1
AttributeError: Line instance has no attribute '__p1'
>>> Line().__distance
AttributeError: Line instance has no attribute '__distance'
```

错误信息显示在类Line中没有与之对应的属性。这表明私有的方法和属性不能在类的的外部调用。

参考文献

[1] 《How to Think Like a Computer Scientist》

[2] 《Learning Python》 Published by O'Reilly & Associates, Inc. 101 Morris Street, Sebastopol, CA 95472.

[3] 《Dive Into Python》
Copyright . 2000, 2001, 2002 Mark Pilgrim