Python 基础篇

卢婧宇

2017 年 9 月 23 日

目录

1 开胃菜

虽然 Python 易于使用，但它却是一门完整的编程语言；与 Shell 脚本或批处理文件相比，它为编写大型程序提供了更多的结构和支持。另一方面，Python 提供了比 C 更多的错误检查，并且作为一门高级语言，它内置支持高级的数据结构类型，例如：灵活的数组和字典。

Python 允许你将程序分割为不同的模块，以便在其他的 Python 程序中重用。Python 内置提供了大量的标准模块，你可以将其用作程序的基础，或者作为学习 Python 编程的示例。这些模块提供了诸如文件 I/O、系统调用、Socket 支持，甚至类似 Tk 的用户图形界面（GUI) 工具包借口。

Python 是一门解释型语言，因为无需编译和链接，你可以在程序开发中节省宝贵的时间。Python 解释器可以交互的使用，这使得试验语言的特性、编写临时程序或在自底向上的程序开发中测试方法非常容易。你甚至还可以把它当做一个桌面计算器。

Python 让程序编写的紧凑和可读。用 Python 编写的程序通常比同样的 C、C++ 或 Java 程序更短小，原因如下：

- 高级数据结构使你可以在一条语句中表达复杂的操作
- 语句组使用缩进代替开始和结束大括号来组织
- 变量或参数无需声明

Python 是可扩展的：如果你会 C 语言编程便可以轻易地为解释器[1] 添加内置函数或模块，或者为了对性能瓶颈作优化，或者将 Python 程序与只有二进制形式的库（比如末各专业的商业图形库）连接起来。一旦你真正掌握了它，你可以将 Python 解释器集成进某个 C 应用程序，并把它当作那个程序的扩展命令行语言。

2 使用 Python 解释器

2.1 调用 Python 解释器

Python 解释器通常被安装在目标机器的 /usr/local/bin/python3.5 目录下。将 /usr/local/bin 目录包含进 Unix shell 的搜索路径里，以确保可以通过输入：Python3.5 命令启动。由于 Python 解释器的安装路径是可选的，这也可能是其他路径，你可以联系安装 Python 的用户或系统管理员确认（例如，/usr/local/python 就是一个常见的选择）。文件结束符 Unix 系统是：Control + D

第二种启动 Python 解释器的方法是 Python -c command [arg] ...，这种方法可以在命令行执行 Python 语句，类似于 shell 中的 -c 选项。由于 Python 语句通常会包含空格或其他特殊 shell 字符，一般建议将命令用单引号包裹起来。

有一些 Python 模块也可以当做脚本使用。你可以使用 python -m module [arg] ... 命令调用它们，这类似在命令行中键入完整的路径名执行模块源文件一样。使用脚本文件时，经常会运行脚本然后进入交互模式。这也可以通过在脚本之前加上 -i 参数来实现。

2.1.1 交互模式

从 tty 读取命令时，我们称解释器工作于交互模式。这种模式下他根据主提示符来执行，主提示符通常标识为三个大于号（>>>); 继续的部分被称为从属提示符，由三个点标识（...)。在第一行之前，解

[1] 解释器，又译为直译器，是一种电脑程序，能够把高级编程语言一行一行直接转译运行。

释器打印欢迎信息、版本号和授权提示

2.1.2 参数传递

调用解释器时，脚本名和附加参数传入一个名为 sys.argv 的字符串列表。你能够获取这个列表通过执行 import sys, 列表的长度大于等于 1；没有给定脚本和参数时，它至少也有一个元素：sys.argv[0] 此时为空字符串。脚本名指定为 '-'（表示标准输入）时，sys.argv[0] 被定义为 '-', 使用 -c 指令时，sys.argv[0] 被设定为 '-c'。使用-m 模块参数时，sys.argv[0] 被设定为指定模块的全名。-c 指令或者 -m 模块之后的参数不会被 Python 解释器的选项处理机制所截获，而是留在 sys.argv 中，供脚本命令操作。

3 Python 简介

输入和输出分别由大括号和句号提示符（>>> 和...) 标注。

Python 中的注释以 # 字符起始，直至实际的行尾。注释可以从行首开始，也可以在空白或代码之后，但是不出现在字符串中。文本字符串中的 # 字符仅仅表示 #。

3.1 将 Python 当做计算器

3.1.1 数字

解释器表现得就像一个简单的计算器：可以向其录入一些表达式，它会给出返回值。表达式语法很直白：运算符 + − ∗ 和 / 与其他语言一样；括号用于分组。

整数（例如，2，4，20）的类型是 int，带有小数部分的数字（例如，5.0，1.6）的类型是 float。Python2 除法想要得到浮点型，除数或被除数应是浮点型。Python3 除法（/）永远返回一个浮点数, 如果要使用 floor 除法并且得到整数结果（丢掉任何小数部分），可以使用（//）运算符；要计算余数使用% 。可以使用 ∗∗ 运算符计算幂乘方。用等号（=）用于给变量赋值。赋值之后在下一个提示符之前不会有任何结果显示。

变量在使用前必须"定义"（赋值），否则会出错。浮点数有完整的支持；整数和浮点数的混合计算中，整数会被转换为浮点数。

交互模式中，最近一个表达式的值赋给变量 '_'。这样我们就可以吧它当做一个桌面计算器：

```
>>> t=12.5/100
>>> p=100.50
>>> p*t
12.5625
>>> p+ _
113.0625
```

此变量对于用户是只读的。不要尝试给它赋值——你只会创建一个独立的同名局部变量，它屏蔽了系统内置变量的魔术效果。

除了 int 和 float，Python 还会支持其它数字类型，例如 Decimal 和 Fraction。Python 还内建支持复数，使用后缀 j 或 J 表示虚数部分（例如，3+5j)。

3.1.2 字符串

相比数值，Python 也提供了可以通过几种不同方式表示的字符串。它们可以用单引号（'...'）或双引号（"..."）标识。\可以用来转义引号：

```
>>> 'spam eggs' #single quotes
'spam eggs'
>>>'doesn\'t'  #use \' to escape the single quote...
"doesn't"
>>> '"Yes,"he said.'
'"Yes,"he said.'
>>> "\"Yes,\"he said."
'"Yes,"he said.'
>>> '"Isn\'t,"she said.'
'"Isn\'t,"she said.'
```

在交互式解释其中，输出的字符串会用引号引起来，特殊字符会用反斜杠转义。虽然可能和输入看上去不太一样，但是两个字符串是相等的。如果字符串中只有单引号而没有双引号，就用双引号引用，否则用单引号引用。print() 函数生成可读性更好的输出，它会省去引号并且打印出转义后的特殊字符：

```
>>> '"Isn\'t,"she said.'
'"Isn\'t,"she said.'
>>> print('"Isn\'t." she said.')
"Isn't." she said.
>>> s='First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  #with print(),\n produces a new line
First line.
Second line.
```

如果你前面带有 \的字符被当作特殊字符，你可以使用原始字符串，方法是在第一个引号前面加上一个 r。

字符串文本能够分成多行。一种方法是使用三引号："""..."""或者 "'...'"。行尾换行符会自动包含到字符串中，但是可以在行尾加上 \来避免这个行为。可以使用反斜杠为行结尾的字符串，他表示下一行在逻辑上是本行的后续内容。

字符串可以由 + 操作符连接（粘到一起），可以由 * 表示重复。相邻两个字符串文本自动连接在一起。它只用于两个字符串文本，不能用于字符串表达式：

```
>>> prefix='Py'
>>> prefix 'thon' # can't concatenate a variable and a string literal
...
SyntaxError: invalid syntax
>>> ('un'*3) 'ium'
```

```
...
SyntaxError: invalid syntax
```

如果你想连接多个变量或者连接一个变量和一个字符串文本，使用 +。这个功能在你想切分很长的字符串的时候特别有用:

```
>>> text=('Put several string within parerheses'
...       'to have them joined together.')
>>> text
'Put several string within parerhesesto have them joined together.'
```

字符串也可以被截取（检索）。类似于 C，字符串的第一个字符索引为 0。Python 没有单独的字符类型；一个字符就是一个简单的长度为 1 的字符串。

```
>>> word='Python'
>>> word[0] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

索引也可以是负数，这将导致从右边开始计算。例如:

```
>>> word[-1] #last character
'n'
>>> word[-2] #second-last character
'o'
>>> word[-6]
'P'
```

请注意 −0 实际上就是 0，所以他不会导致从右边开始计算。

除了索引，还支持切片。索引用于获得单个字符，切片让你获得一个子字符串:

```
>>> word[0:2] # character from position 0(included)to 2 (excluded)
'Py'
>>> word[2:5] # character from position 2(included) to 5(excluded)
'tho'
```

注意，包含起始的字符，不包含末尾的字符。这使得 s[:i]+s[i:] 永远等于 s,:

```
>>> word[:2]+word[2:]
'Python'
>>> word[:4]+word[4:]
'Python'
```

切片的索引有非常有用的默认值；省略的第一个索引默认为零，省略的第二个索引默认为切片的字符串的大小。:

```
>>> word[:2]
'Py'
>>> word[4:]
'on'
>>> word[-2:]
'on'
```

有个办法可以很容易地记住切片的工作方式：切片时的索引是在两个字符之间。左边第一个字符的索引为 0，而长度为 n 的字符串最后一个字符的右界索引为 n。例如：

+—+—+—+—+—+—+
| P | y | t | h | o | n |
+—+—+—+—+—+—+
0 1 2 3 4 5 6
-6 -5 -4 -3 -2 -1

文本中的第一行数字给出字符串的索引点 0...6。第二行给出相应的负索引。切片是从 i 到 j 两个数值表示的边界之间的所有字符。

对于非负索引，如果上下都在边界内，切片长度就是两个索引之差。

试图使用太大索引会导致错误。

```
>>> word[42] # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Python 能够优雅地处理那些没有意义的切片索引：一个过大的索引值（即下标值大于字符串实际长度）将被字符串实际长度所替代，当上边界比下边界大时（即切片左值大于右值）就返回空字符串:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python 字符串不可以被更改——它们是不可变的。因此，赋值给字符串索引的位置会导致错误。:

```
>>> word[0]='J'
...
TypeError: 'str' object does not support item assignment
>>> word[2:]='py'
...
TypeError: 'str' object does not support item assignment
```

如果你需要一个不同的字符串，你应该创建一个新的:

```
>>> 'J'+word[1:]
'Jython'
>>> word[:2]+'py'
'Pypy'
```

内置函数 len() 返回字符串长度:

```
>>> s='supercalifragilisticexpialidocious'
>>> len(s)
34
```

3.1.3 列表

Python 有几个复合数据类型，用于表示其它的值。最通用的是 list（列表），它可以写作中括号之间的一列逗号分隔的值。列标的元素不必是同一类型:

```
>>> squares=[1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

就像字符串（以及其它所有内建的序列类型）一样，列表可以被索引和切片:

```
>>> squares[0] # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:] #slicing return a new list
[9, 16, 25]
```

所有的切片操作都会返回一个包含请求的元素的新列表。这意味着下面的切片操作返回列表一个新的（浅）拷贝副本:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

列表也支持连接这样的操作:

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

不像不可变的字符串，列表是可变的，它允许修改元素:

```
>>> cubes=[1, 8, 27, 65, 125] #something's wrong here
>>> 4 ** 3 #the cub of 4 is 64,not 65!
64
>>> cubes[3]=64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

还可以使用 append() 方法在列表的末尾添加新的元素:

```
>>> cubes.append(216) #add the cube of 6
>>> cubes.append(7 ** 3) # add the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

也可以对切片赋值，此操作可以改变列表的尺寸，或清空它：

```
>>> letters=['a','b','c','d','e','f','g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> #replace some values
... letters[2:5]=['C','D','E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
... letters[2:5]=[]
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
... letters[:]=[]
>>> letters
[]
```

内置函数 len() 同样适用于列表：

```
>>> letters=['a','b','c','d']
>>> len(letters)
4
```

允许嵌套列表（创建一个包含洽谈列表的列表），例如：

```
>>> a=['a','b','c']
>>> n=[1,2,3]
>>> x=[a,n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0][1]
'b'
```

3.2 编程的第一步

当然，我们可以使用 Python 完成比二加二更复杂的任务。例如，我们可以写一个生成斐波那契子程序的程序，如下所示：

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0,1
>>> while b<10:
...     print(b)
...     a, b = b, a+b
```

```
...
1
1
2
3
5
8
```

这个例子介绍了几个新功能。

- 第一行包括了一个多重赋值：变量 a 和 b 同时获得了新的值 0 和 1 最后一行又使用了一次。在这个演示中，变量赋值前，右边首先完成计算。右边的表达式从左到右计算。
- 条件（这里是 b<10）为 true 时，while 循环执行。在 Python 中，类似于 C，任何非零整数都是 true;0 是 false。条件也是可以是字符串或列表，实际上可以使任何序列；所有长度不为零的是 true，空序列是 false。示例中的测试是一个检点的比较。标准比较操作符与 C 相同：< , > , == , <= , >= 和！=。
- 循环体是缩进的：缩进是 Python 组织语句的方法。Python(还) 不提供集成的行编辑功能，所以你要为每一个缩进行输入 TAB 或空格。

  实践中建议你找个文本编辑来录入复杂的 Python 程序，大多数文本编辑器提供自动缩进。交互式录入复合语句时，必须在最后一个空行来标识结束（因为解释器没办法猜测你输入的哪一行是最后一行），需要注意的是同一个语句块中的每一行必须缩进同样数量的空白
- 关键字 print() 语句输出给定表达式的值。它控制多个表达式和字符串输出为你想要字符串（就像在前面计算器的例子中那样）。

  字符串打印时不用引号包围，每两个子项之间插入空间，所以你可以把格式弄得很漂亮，像这样

```
>>> i=256*256
>>> print('The value of i is',i)
The value of i is', 65536
```

一个逗号结尾就可以禁止输出换行：

```
>>> a, b=0 ,1
>>> while b < 1000:
...     print(b, end=',')
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

4 深入 Python 流程控制

除了前面介绍的 while 语句，Python 还从其他语言借鉴了一些流程控制功能，并有所改变。

4.1 if 语句

也许最有名的就是 if 语句。例如：

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x==0:
...     print('Zero')
... elif x==1:
...     print('Single')
... else:
...     print('More')
...
More
```

可能会有零到多个 elif 部分，else 是可选的。关键字 'elif' 是 'else if' 的缩写，这个可以有效地避免过深的缩进。if ... elif ... elif ... 序列用于代替其他语言中的 swith 或 case 语句。

4.2 for 语句

Python 中的 for 语句和 C 或 Pascal 中的略有不同。通常的循环可能会依据一个等差数值步进过程（如 Pascal），或由用户来定义迭代步骤和终止条件（如 C），Python 的 for 语句依据任意序列（链表或字符串）中的子项，按它们在序列中的顺序来进行迭代。例如（没有暗指）：

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

在迭代过程中修改序列不安全（只有在使用链表这样的可变序列时才会有这样的情况）。如果你想要修改你的迭代的序列（例如，复制选择项），你可以迭代它的副本。使用切割标识就可以很方便的做到这一点：

```
>>> for w in words[:]:
...     if len(w) > 6:
...             words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3 range() 函数

如果你需要一个数值序列，内置函数 range() 会很方便，它生成一个等差级数链表：

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

range(10) 生成了一个包含 10 个值的链表，他用链表的索引值填充了这个长度为 10 的列表，所生成的链表中不包括范围中的结束值。也可以让 range() 操作从另一个数值开始，或者可以指定一个不同的步进值（甚至是负数，有时这也被称为步长）

```
range(5, 10)
  5 through 9
range(0, 10, 3)
  0, 3, 6, 9
range(-10, -100, -30)
  -10, -40, -70
```

需要迭代链表索引的话，如下所示结合使用用 range() 和 len()

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

不过，这种场合可以方便的使用 enumerate(), 请参见循环技巧。

如果你只是打印一个序列的话会发生奇怪的事情：

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

在不同方面 range() 函数返回的对象表现为它是一个列表，但事实上它并不是。当你迭代它时，它是一个能够像期望的序列返回连续项的对象；但为了节省空间，他并不真正构造列表。

我们称此类对象是可迭代的，即适合作为那些期望从某些东西中获得连续项直到结束的函数或结构的一个目标（参数）。我们已经见过的 for 语句就是这样一个迭代器。list() 函数是另外一个（迭代器），他从可迭代（对象）中创建列表：

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

稍后我们会看到更多返回可迭代（对象）和以可迭代（对象）作为参数的函数。

4.4 break 和 continue 语句，以及循环中的 else 子句

break 语句和 C 中的类似，用于跳出最近的一级 for 或 while 循环。

循环可以有一个 else 子句；它在循环迭代完整个列表（对于 for）或执行条件为 false（对于 while）时执行，但循环被 break 中止的情况不会执行。以下搜索素数的示例程序演示了这个子句：

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...             if n % x == 0:
...                     print(n, 'equals', x, '*', n//x)
...                     break
...     else:
...         # loop fell through without finding a factor
...             print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, 这是正确的代码。看仔细：else 语句是属于 for 循环之中，不是 if 语句。)

与循环一起使用时，else 子句与 try 语句的 else 子句比与 if 语句的具有更多的共同特点：try 语句的 else 子句在未出现异常时运行，循环的 else 子句在未出现 break 时运行。更多关于 try 语句和异常的内容，请参见异常处理。

continue 语句是从 C 中借鉴来的，它表示循环继续执行下一次迭代：

```
>>> for num in range(2, 10):
...     if num % 2 ==0:
...             print("Found an even number", num)
...             continue
...     print("Found a number", num)
...
Found an even number 2
Found a number 3
Found an even number 4
```

```
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

4.5 pass 语句

pass 语句什么也不做。它用于那些语法上必须要有什么语句，但程序什么也不做的场合，例如：

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

者通常用于创建最小结构的类：

```
>>> class MyEmptyClass:
...     pass
...
```

另一方面，pass 可以在创建新代码时用来做函数或控制体的占位符。可以让你在更抽象的级别上思考。pass 可以默默的被忽视：

```
>>> def initlog(*args):
...     pass  # Remember to implement this!
...
```

4.6 定义函数

我们可以创建一个用来生成指定边界的斐波那契数列的函数：

```
>>> def fib(n):  # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...             print(a, end=' ')
...             a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

关键字 def 引入了一个函数定义。在其后必须跟有函数名和包括形式参数的圆括号。函数体语句从下一行开始，必须是缩进的。

函数体第一行语句可以是可选的字符串文本,这个字符串是函数的文档字符串,或者称为 docstring。(更多关于 docstring 的信息请参考文档字符串）有些工具通过 docstrings 自动生成在线的或可打印的文档，或者让用户通过代码交互浏览；在你的代码中包含 docstrings 是一个好的实践，让它成为习惯吧。

函数调用会为函数局部变量生成一个新的符号表。确切的说，所有函数中的变量赋值都是将值存储在局部符号表。变量引用首先在局部符号表中查找，然后是包含函数的局部符号表，最后是内置名字表。因此，全局变量不能在函数中直接赋值（除非用 global 语句命名），尽管他们可以被引用。

函数引用的实际参数在函数调用时引入局部符号表，因此，实参总是传值调用（这里的值总是一个对象引用，而不是该对象的值)。一个函数被另一个函数调用时，一个新的局部符号表在调用过程中被创建。

一个函数定义会在当前符号表内引入函数名。函数名指代的值（即函数体）有一个被 Python 解释器认定为用户自定义函数的类型。这个值可以赋予其他的名字（即变量名），然后它也可以被当做函数使用。这可以作为通用的重命名机制：

```
>>> fib
<function fib at 0x102aa29d8>
>>> f=fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

如果你使用过其他语言，你可能会反对说：fib 不是一个函数，而是一个方法因为它并不返回任何值。事实上，没有 return 语句的函数确实会返回一个值，虽然是一个相当令人厌烦的值（指 None)。这个值被称为 None(这是一个内建名称)。如果 None 值是唯一被书写的值，那么在写的时候通常会被解释器忽略(即不输出任何内容)。如果你确实想看到这个值的输出内容，请使用 print() 函数：

```
>>> fib(0)
>>> print(fib(0))
None
```

定义一个返回斐波那契数列数字列表的函数，而不是打印它，是很简单的：

```
>>> def fib2(n): #return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...             result.append(a)  # see below
...             a, b = b, a+b
...     return result
...
>>> f100 = fib2(100) # call it
>>> f100             # write the reslut
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

和以前一样，这个例子演示了一些新的 Python 功能：

- return 语句从函数中返回一个值，不带表达式的 return 返回 None。过程结束后也会返回 None。
- 语句 result.append(b) 称为链表对象 result 的一个方法。方法是一个“属于”某个对象的函数，它被命名为 obj.methodename, 这里的 obj 是某个对象（可能是一个表达式），methodename 是某个在该对象类型定义中的方法的命名。

  不同类型定义不同的方法。不同类型可能有同样名字的方法，但不会混淆。（当你定义自己的对象类型和方法时，可能会出现这种情况，class 的定义方法详见类）。示例中演示的 append() 方法由链表对象定义，它向链表中加入一个新元素。在示例中他等同于 result =result +[a], 不过效率更高。

4.7 深入 Python 函数定义

在 Python 中，你也可以定义包含若干参数的函数。这里有三种可用的形式，也可以混合使用。

4.7.1 默认参数值

最常用的一种形式是为一个或多个参数指定默认值。这会创建一个可以使用比定义时允许的参数更少的参数调用的函数，例如:

```
>>> def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
...     while True:
...             ok = input(prompt)
...             if ok in ('y', 'ye', 'yes'):
...                     return Ture
...             if ok in ('n', 'no', 'nop', 'nope'):
...                     return False
...             retries = retries - 1
...             if retries < 0:
...                     raise OSError('uncooperat user')
...             print(complaint)
```

这个函数可以通过几种不同的方式调用:

- 只给出必要的参数: ask_ok (‘Do you really want to quit?’)
- 给出一个可选的参数:

  ask_ok(‘Ok to overwrite the file?’, 2)
- 或者给出所有的参数:

  ask_ok(‘OK to overwrite the file?’, 2, ‘Come only yes or no!’)

这个例子还介绍了 in 关键字。它测定序列中是否包含某个确定的值。
默认值在函数定义作用域被解析，如下所示:

```
>>> i = 5
>>> def f(arg = i):
...     print(arg)
```

```
...
>>> i = 6
>>> f()
5
```

重要警告：默认值只被赋值一次。这使得当默认值是可变对象时会有所不同，比如列表、字典或者大多数类的实例。例如，下面的函数在后续调用过程中会累积（前面）传给它的参数：

```
>>> def f(a, L=[]):
...     L.append(a)
...     return L
...
>>> print(f(1))
[1]
>>> print(f(2))
[1, 2]
>>> print(f(3))
[1, 2, 3]
```

如果你不想让默认值在后续调用中累积，你可以像下面一样定义函数：

```
>>> def f(a, L=None):
...     if L is None:
...             L = []
...     L.append (a)
...     return L
```

4.7.2 关键字参数

函数可以通过关键字参数的形式来调用，形如 keyword = value。例如，以下的函数：

```
>>> def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.")
...     print("-- Lovely plumage, the", type)
...     print("-- It's", state, "!")
```

接受一个必选参数（voltage）以及三个可选参数（state, action 和 type）。可以用以下的任一方法调用：

```
>>> parrot(1000)                       # 1 positional argument
-- This parrot wouldn't voom if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
>>> parrot(voltage=1000)               # 1 keyword argument
-- This parrot wouldn't voom if you put 1000 volts through it.
```

```
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
>>> parrot(voltage=1000000, action='VOOOOOM')  # 2 keyword arguments
-- This parrot wouldn't VOOOOOM if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
>>> parrot(action='VOOOOOM',voltage=1000000)   # 2 keyword arguments
-- This parrot wouldn't VOOOOOM if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
>>> parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
-- This parrot wouldn't voom if you put a thousand volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's pushing up the daisies !
```

不过以下几种调用是无效的:

```
>>> parrot()                     # requried argument missing
>>> parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
>>> parrot(110, voltage=220)     # duplicate value for the same argument
>>> parrot(actor='John Cleese')  # unknow keyword argument
```

在函数调用中，关键字的参数必须跟随在位置参数的后面。传递的所有关键字参数必须与函数接受的某个参数相匹配（例如 actor 不是 parrot 函数的有效参数），它们的顺序并不重要。这也包括非可选参数（例如 parrot(voltage=100) 也是有效的）。任何参数都不可以多次赋值。下面的示例由于这种限制将失败:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

引入一个形如 **name 的参数时，它接收一个字典（参见 Mapping Types-dict），该字典包含了所有未出现在形式参数列表中的关键字参数。这里可能还会组合使用一个形如 *name （下一小节详细介绍）的形式参数，它接收一个元组（下一节中会详细介绍），包含了所有没有出现在形式参数列表中的参数值（*name 必须在 **name 之前出现）。例如，我们这样定义一个函数:

```
>>> def cheeseshop(kind, *arguments, **keywords):
...     print("-- Do you have any", kind, "?")
...     print("-- I'm sorry. we're all out of", kind)
...     for arg in arguments:
...             print(arg)
```

```
...     print("-" * 40)
...     keys = sorted(keywords.keys())
...     for kw in keys:
...             print(kw, ":", keywords[kw])
```

它可以像这样调用：

```
>>> cheeseshop("Limburger", "It's very runny, sir.",
...            "It's really very, VERY runny, sir.",
...            shopkeeper="Michael Palin."
...            client="John Cleese",
...            sketch="Cheese Shop Sketch")
```

当然它会按如下内容打印：

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

注意在打印关键字参数之前，通过对关键字字典 keys() 方法的结果进行排序，生成了关键字参数名的列表；如果不这样做，打印出来的参数顺序是未定义的。

4.7.3 可变参数列表

最后，一个最不常用的选择是可以让函数调用可变个数的参数。这些参数被包装进一个元祖（参见元祖和序列）。在这个可变个数的参数之前，可以有零到多个普通的参数：

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.7.4 参数列表的拆分

另一种相反的情况：当你要传递的参数已经是一个列表，但要调用的函数却接受分开一个个的参数值。这时候你要把已有的列表拆开来。例如内建函数 range() 需要要独立的 start, stop 参数。你可以在调用函数时加一个 * 操作符来自动把参数列表拆开：

```
>>> list(range(3, 6))    # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))  # call with arguments unpacked from a list
[3, 4, 5]
```

以同样的方法，可以使用 ** 操作符拆分关键字参数为字典：

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrto wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrto wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

4.7.5 Lambda 形式

出于实际需要，有几种通常在函数式编程语言例如 Lisp 中出现的功能加入到了 Python。通过 lambda 关键字，可以创建短小的匿名函数。这里有一个函数返回它的两个参数的和：lambda a, b: a+b。Lambda 形式可以用于任何需要的函数对象。出于语法限制，它们只能有一个单独的表达式。

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

上面的示例使用 lambda 表达式返回一个函数。另一个用途是将一个小函数作为参数传递：

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.6 文档字符串

这里介绍的文档字符串的概念和格式。

第一行应该是关于对象用途的简介。简短起见，不用明确的陈述对象名或类型，因为它们可以从别的途径了解到（除非这个名字碰巧就是描述这个函数操作的动词）。这一行应该以大写字母开头，以句号结尾。

如果文档字符串有多行，第二行应该空出来，与接下俩的详细描述明确分隔。接下来的文档婴孩有一或多段描述对象的调用约定、边界效应等。

Python 的解释器不会从多行的文档字符串中取除缩进，所以必要的时候应当自己清除缩进。这符合通常的习惯。第一行之后的第一个非空行决定了整个文档的缩进格式。(我们不用第一行是因为它通常紧靠着起始的引号，缩进的话，所有的留白都应该清除掉。留白的长度应当等于扩展制表符的宽度（通常是 8 个空格)。

以下是一个多行文档字符串的示例:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

^^INo, really, it doesn't do anything.
```

5 数据结构

5.1 关于列表的更多的内容

Python 的列表数据类型包含更多的方法。这里是所有的列表对象方法:

- list.append(x) 把一个元素添加到列表的结尾，相当于 a[len(a):] = [x]
- list.extend(L) 将一个给定列表中的所有元素都添加到另一个列表中，相当于 a[len(a):]=L
- list.insert(i,x) 在指定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，例如 a.insert(0, x) 会插入到整个列表之前，而 a.insert(len(a), x) 相当于 a.append(x)。
- list.remove(x) 删除列表中值为 x 的第一个元素。如果没有这样的元素，就会返回一个错误。
- list.pop([i]) 从列表的指定位置删除元素，并将其返回。如果没有指定索引，a.pop() 返回最后一个元素。元素随机从列表中被删除（方法中 i 两边的方括号表示这个参数是可选的，而不是要求你的输入一对方括号，你会经常在 Python 库参考手册中遇到这样的标记)。
- list.clear() 从列表中删除所有元素。相当于 del a[:]。
- list.count(x) 返回 x 在列表中出现的次数
- list.index(x) 返回列表中第一个值为 x 的元素的索引。如果没有匹配的元素就会返回一个错误。
- list.sort() 对类表中的元素就地进行排列。
- list.reverse() 就地倒排列表中的元素。

- list.copy() 返回列表的一个浅拷贝。等同于 a[:]。

下面这个示例演示了列表的大部分方法：

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

也许大家会发现像 insert, remove 或者 sort 这些修改列表的方法没有打印返回值——它们返回 None。在 python 中对所有可变的数据类型这是统一的设计原则。

5.1.1 把列表当作堆栈使用

列表方法使得列表可以很方便的作为一个堆栈来使用，堆栈作为特定的数据结构，最先进入的元素最后一个被释放（后进先出）。用 append() 方法可以把一个元素添加到堆栈顶。用不指定索引的 pop() 方法可以把一个元素从堆栈顶释放出来。例如：

```
 >>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
```

```
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 把列表当做队列使用

你也可以把列表当做队列使用，队列作为特定的数据结构，最先进入的元素最先释放（先进先出）。不过，列表这样用效率不高。相对来说从列表末尾添加和弹出很快；在头部插入和弹出很慢（因为，为了一个元素，要移动整个列表中的所有元素）。

要实现队列，使用 collections.deque, 它为在首尾两端快速插入和删除而设计。例如：

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 列表推导式

列表推导式为从序列中创建列表提供了一个简单的方法。普通的应用程式通过将一些操作应用于序列的每个成员并通过返回的元素创建列表，或者通过满足特定条件的元素创建子序列。

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

注意这个 for 循环中的被创建（或被重写）的名为 x 的变量在循环完毕后依然存在。使用如下方法，我们可以计算 squares 的值而不会产生任何的副作用：

```
>>> squares = list(map(lambda x: x**2, range(10)))
```

或者等价于：

```
squares = [x**2 for x in range(10)]
```

上面这个方法更加简明且易读。

列表推导式由包含一个表达式的括号组成，表达式后面跟随一个 for 子句，之后可以有零个或多个 for 或 if 子句。结果是一个列表，由表达式依据其后面的 for 和 if 子句上下文计算而来的结果构成。

例如，如下的列表推导式结合两个列表的元素，如果两个元素之间不相等的话：

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x !=y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

等同于：

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...             if x != y:
...                     combs.append((x,y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

值得注意的是在上面两个方法中的 for 和 if 语句的顺序。

如果想要得到一个元组（例如，上面例子中的（x，y)），必须要加上括号：

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>>[x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each elements
>>> freshfruit = [' banana', ' loganberry', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
               ^
```

```
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

列表推导式可使用复杂的表达式和嵌套函数：

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4 嵌套的列表推导式

列表解析中的第一个表达式可以是任何表达式，包括列表解析。

考虑下面由三个长度为 4 的列表组成 3×4 矩阵：

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

现在如果你想交换行和列，可以用嵌套的列表推导式：

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

像前面看到的，嵌套的列表推导式是对 for 后面的内容进行求值，所以上例就等价于：

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...             transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> tramsposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

在实际中，你应该更喜欢使用内置函数组成复杂流程语句。对这种情况 zip() 函数将会做的更好：

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

更多关于本行中使用的星号的说明，参考参数列表的拆分。

5.2 del 语句

有个方法可以从类表中按给定的索引而不是值来删除一个子项：del 语句。它不同于有返回值的 pop() 方法。语句 del 还可以从列表中删除切片或清空整个列表（我们以前介绍过一个方法是将空列表赋值给列表的切片）。例如：

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

del 也可以删除整个变量：

```
>>> del a
>>> a
```

此后再引用命名 a 会引发错误（直到另一个值赋给它为止）。我们在后面的内容中可以看到 del 的其它用法。

5.3 元组和序列

我们知道列表和字符串有很多通用的属性，例如索引和切割操作。它们是序列类型（参见 Sequence Types - list,tuple,range) 中的两种。因为 Python 是一个在不停进化的语言，也可能会加入其它的序列类型，这里介绍另一种标准序列类型：元组。

一个元组由数个逗号分隔的值组成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

正如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可以有或没有括号，不过经常括号都是必须的（如果元组是一个更大的表达式的一部分）。不能给元组的一个独立的元素赋值（尽管你可以通过联接和切割来模拟）。还可以创建包含可变对象的元组，例如列表。

虽然元组和列表很类似，它们经常被用来在不同的情况和不同的用途。元组有很多用途。例如（x,y）坐标对，数据库中的员工记录等等。元组就像字符串，不可变的。通常包含不同种类的元素并通过分拆（参阅本节后面的内容）或索引访问（如果是 namedtuples, 甚至可以通过属性）。列表是可变的，它们的元素通常是相同类型的并通过迭代访问。

一个特殊的问题是构造包含零个或一个元素的元组：为了适应这种情况，语法上有一些额外的改变。一对空的括号可以创建空元组；要创建一个单元素元组可以在值后面跟一个逗号（在括号中放入一个单值不够明确）。丑陋，但是有效。例如：

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

语句 t = 12345, 54321, ‘hello!’ 是元组封装（tuple packing）的一个例子：值 12345，54321 和 ‘hello’ 被封装进元组。其逆操作可能是这样：

```
>>> x, y, z = t
```

这个调用等号右边可以是任何线性序列，称之为序列拆分非常恰当。序列拆封要企业左侧的变量数目与序列的元素个数相同。要注意的是可变参数（multiple assignment) 其实只是元组封装的序列拆封的一个结合。

5.4 集合

Python 还包含了一个数据类型–set（集合）。集合是一个无序不重复元素的集。基本功能包括关系测试和消除重复元素。集合对象还支持 union（联合），intersection（交），difference（差）和 sysmmetric difference（对称差集）等数学运算。

大括号或 set() 函数可以用来创建集合。注意：想要创建空集合，你必须要使用 set() 而不是。后者用于创建空字典。以下是一个简单的演示：

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
```

```
{'pear', 'apple', 'orange', 'banana'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'r', 'a', 'd', 'c', 'b'}
>>> a - b                              # letters in a but not in b
{'d', 'r', 'b'}
>>> a | b                              # letters in either a or b
{'r', 'a', 'l', 'd', 'c', 'z', 'm', 'b'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'l', 'z', 'd', 'm', 'b'}
```

类似列表推导式，这里有一种集合推导式语法：

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'d', 'r'}
```

5.5 字典

另一个非常有用的 Python 内建数据类型是字典（参见 Mapping Types - dict）。字典在某些语言中可能称为联合内存（associative memories）或联合数组（associative arrays）。序列是以连续的整数为索引，与此不同的是，字典以关键字为索引，关键字可以是任意不可变类型，通常用字符串或数值。如果元组中只包含字符串和数字，他可以作为关键字，如果它直接或间接的包含了可变对象，就不能当做关键字。不能用列表做关键字，因为列表可以用索引、切割或者 appen() 和 extend() 等方法改变。

理解字典的最佳方式是把它看做无序的键：值对（key:value 对）集合，键必须是互不相同的（在同一个字典之内）。一对大括号创建一个空的字典： 。初始化列表时，在大括号内放置一组逗号分隔的键:值对，这也是字典输出的方式。

字典的主要操作是依据键来存储和析取值。也可以用 del 来删除键：值对（key:value）。如果你用一个已经存在的关键字存储值，以前为该关键字分配的值就会被遗忘。试图从一个不存在的键中取值会导致错误。

对一个字典执行 list(d.keys()) 将返回一个字典中所有关键字组成的无序列表（如果你想要排序，只需使用 sorted(d.key()) 使用 in 关键字（指 Python 语法）可以检查字典中是否存在某个关键字（指字典）。

字典示例：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
```

```
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

dict() 构造函数可以直接从 key-value 对中创建字典：

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

此外，字典推导可以从任意的键值表达式中创建字典：

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

如果关键字都是简单的字符串，有时通过关键字参数指定 key-value 对更为方便：

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6 循环技巧

在字典中循环时，关键字和对应的值可以使用 items() 方法同时解读出来：

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k,v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

在序列中循环时，索引位置和对应值可以使用 enumerate() 函数同时得到：

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

同时循环两个或更多的序列，可以使用 zip() 整体打包：

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}? It is {1}.' .format(q, a))
...
What is your name ? It is lancelot.
What is your quest ? It is the holy grail.
What is your favorite color ? It is blue.
```

需要逆向循环序列的话，先正向定位序列，然后调用 reversed() 函数：

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

要按排序后的顺序循环序列的话，使用 sorted() 函数，他不改动原序列，而是生成一个新的已排序的序列：

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

若要在循环内部修改正在遍历的序列（例如复制某些元素），建议您首先制作副本。在序列上循环不会隐式地创建副本。切片表示法使用这尤其方便：

```
>>> words = ['cat', 'window', 'defenstrate']
>>> for w in words[:]:      # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...             words.insert(0, w)
...
>>> words
['defenstrate', 'cat', 'window', 'defenstrate']
```

5.7 深入条件控制

while 和 if 语句中使用的条件不仅可以使用比较，而且可以包含任意的操作。

比较操作符 in 和 not in 审核值是否在一个区间之内。操作符 is 和 is not 比较两个对象时否相同；这只和诸如列表这样的可变对象有关。所有的比较操作符具有相同的优先级，低于所有的数值操作。

比较操作可以传递。例如 a < b == c 审核是否 a 小于 b 并且 b 等于 c。

比较操作可以通过逻辑操作符 and 和 or 组合，比较的结果可以用 not 来取反义。这些操作符的优先级又低于比较操作符，在他们之中，not 具有最高的优先级，or 优先级最低，所以 A and not B or C 等于（A and (notB)) or C。当然，括号也可以用于比较表达式。

逻辑操作符 and 和 or 也称为短路操作符：它们的参数从左向右解析，一旦结果可以确定就停止。例如，如果 A 和 C 为真而 B 为假，A and B and C 不会解析 C。作用于一个普通的非逻辑值时，短路操作符的返回值通常是最后一个变量。

可以把比较或其它逻辑表达式的返回值赋给一个变量，例如：

```
>>> string1, string2, string3 = '', 'Tronheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Tronheim'
```

需要注意的是 Python 与 C 不同，在表达式内部不能赋值。C 程序员经常对此抱怨，不过它避免了一类在 C 程序中司空见惯的错误：想要在解析式中使用 == 时误用了 = 操作符。

5.8 比较序列和其它类型

序列对像可以与相同类型的其它对象比较。比较操作按字典序进行：首先比较前两个元素，如果不同，就决定了比较的结果；如果相同，就比较后两个元素，依此类推，直到所有序列都完成比较。如果两个元素本身就是同样类型的序列，就递归字典序比较。如果两个序列的所有子项都相等，就认为序列相等。如果一个序列是另一个序列的初始子序列，较短的一个序列就小于另一个。字符串的字典序按照单字符的 ASCII 顺序。下面是同类型序列之间比较的一些例子：

```
>>> (1, 2, 3)              < (1, 2, 4)
True
>>> [1, 2, 3]              < [1, 2, 4]
True
>>> 'ABC' < 'C' < 'Pascal' < 'Python'
True
>>> (1, 2, 3, 4)           < (1, 2, 4)
True
>>> (1, 2)                 < (1, 2, -1)
True
>>> (1, 2, 3)              < (1.0, 2.0, 3.0)
False
>>> (1, 2, ('aa', 'ab'))       <(1, 2, ('abc', 'a'), 4)
True
```

需要注意的是如果通过 < 或者 > 比较的对象只要具有合适的比较方法就是合法的。比如，混合数值类型是通过它们的数值进行比较的，所以 0 等于 0.0。否则解释器将会触发一个 TypeError 异常，而不是提供以和随意的结果。

6 模块

如果你退出 Python 解释器并重新进入，你做的任何定义（变量和方法）都会丢失。因此，如果你想要编写一些更大的程序，为准备解释器输入使用一个文本编辑器会更好，并以那个文件替代作为输入执行。这就是传说中的脚本。随着你的程序变得越来越长，你可能想要将它分割成几个更易于维护的文件。你也可能想在不同的程序中使用顺手的函数，而不是把代码在它们之间中考来拷去。

为了满足这些需要，Python 提供了一个方法可以从文件中获取定义，在脚本或者解释器的一个交互式实例中使用。这样的文件被称为模块；模块中的定义可以导入到另一个模块或主模块中（在脚本执行时可以调用的变量集位于最高级，并且处于计算器模式）。

模块是包括 Python 定义和声明的文件。文件名就是模块名加上.py 后缀。模块的模块名（做为一个字符串）可以由全局变量 ___name___ 得到。例如，你可以用自己惯用的文件编辑器在当前目录下创建一个叫 fibo.py 的文件，录入如下内容：

```
# Fibonacci numbers module

def fib(n):        # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n):        # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

现在进入 Python 解释器并使用以下命令导入模块：

```
>>> import fibo
```

这样做不会直接把 fibo 中的函数导入当前的语义表；它只是引入了模块名 fibo。你可以通过模块名按如下方式访问这个函数：

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib(100)
```

```
1 1 2 3 5 8 13 21 34 55 89
>>> fibo.__name__
'fibo'
```

如果打算频繁使用一个函数，你可以将它赋予一个本地变量：

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 深入模块

除了包含函数定义外，模块也可以包含执行语句。这些语句一般用来初始化模块。他们仅在第一次被导入的地方执行一次。

每个模块都有自己私有的符号表，被模块内所有的函数定义作为全局符号使用。因此，模块的作者可以在模块内部使用全局变量，而无需担心它与某个用户的全局变量意外冲突。从另一个方面讲，如果你确切的知道自己在做什么，你可以使用引用模块函数的表示法访问模块的全局变量，modname.itemname。

模块可以导入其他的模块。一个（好的）习惯是将所以的 import 语句放在模块的开始（或者是脚本），这并非强制。被导入命名到本模块的语义表中。

import 语句的一个变体直接从被导入的模块中导入命名到本模块的语义表中。例如：

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这样不会从局域语义表中导入模块名（如上所示，fibo 没有定义）。

甚至有种方式可以导入模块中的所有定义：

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这样可以导入所有除了以下划线（_）开头的命名。

需要注意的是在实践中往往不鼓励从一个模块或包中使用导入所有，因为这样会让代码变得很难读。不过，在交互式会话中这样用很方便省力。

注意：出于性能考虑，每个模块在每个解释器会话中只导入一遍。因此，如果你修改了你的模块，需要重启解释器；或者，如果你就是想交互式的测试这么一个模块，可以用 imp.reload() 重新加载，例如：import imp: imp.reload(modulename)。

6.1.1 作为脚本来执行模块

当你使用以下方式运行 Python 模块时，模块中的代码便会被执行：

```
>>> python fibo.py <arguments>
```

模块中的代码会被执行，就像导入它一样，不过此时 ___name___ 被设置为 “___main___”。这相当于，如果你在模块后加入代码：

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

就可以让文件像作为模块导入时一样作为脚本执行。此代码只有在模块作为 “mian” 文件执行时才被调用：

```
lujingyu $ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

如果模块被导入，不会执行这段代码：

```
>>> import fibo
>>>
```

这通常用来为模块提供一个便于测试的用户接口（将模块作为脚本执行测试需求）。

6.1.2 模块的搜索路径

导入一个叫 spam 的模块时，解释器先在当前目录中搜索名为 spam.py 的文件。如果没有找到的话，接着会到 sys.path 变量中给出的目录列表中查找。sys.path 变量的初始化值来自如下：

- 输入脚本的目录（当前目录）。
- 环境变量 PYTHONPATH 表示的目录列表中搜索（这和 shell 变量 PATH 具有一样的语法，即一系列目录名的列表）。
- Python 默认安装路径中搜索。

注释：在支持符号连接的文件系统中，输入的脚本所在的目录是符号链接指向的目录。换句话说也就是包含符号链接的目录不会被加到目录搜索路径中。

实际上解释器由 sys.path 变量指定的路径目录搜索模块，该变量初始化时默认包含了输入脚本（或者当前目录），PYTHONPATH 和安装目录。这样就允许 Python 程序了解如何修改或替换模块搜索目录。需要注意的是由于这些目录中包含有搜索路径中运行的脚本，所以这些脚本不应该和标准模块重名，否则在导入模块时 Python 会尝试把这些脚本当做模块来加载。这通常会引发错误。

6.1.3 ” 编译的”Python 文件

为了加快加载模块的速度，Python 会在 __pycache__ 目录下以 module.version.pyc 名字缓存每个模块编译后的版本，这里的版本编制了编译后文件的格式。它通常会包含 Python 的版本号。例如，在 CPython 3.3 版中，spam.py 编译后的版本将缓存为 __pycache__spam.cpython33.pyc。这种命名约定允许由不同发布和不同版本的 Python 编译的模块同时存在。

Python 会检查源文件与编译版的修改日期以确定它是否过期并需要重新编译。这是完全自动化的过程。同时，编译后的模块是跨平台的，所以同一个库可以在不同架构的系统之间共享。

Python 不检查在两个不同环境中的缓存。首先，它会永远重新编译而且不会存储直接从命令行加载的模块。其次，如果没有源模块它不会检查缓存。若要支持没有源文件（只有编译版）的发布，编译后的模块必须在源目录下，并且必须没有源文件的模块。

部分高级技巧：

- 为了减少一个编译模块的大小，你可以在 Python 命令行中使用 O 或者 OO。O 参数删除了断言语句，OO 参数删除了断言语句和 __doc__ 字符串。

  因为某些程序依赖因为某些程序依赖于这些变量的可用性，你应该只在确定无误的场合使用这一选项。“优化的”模块有一个.pyo 后缀而不是.pyc 后缀。未来的版本可能会改变优化的效果。

- 来自.pyc 文件或.pyo 文件中的程序不会比来自.py 文件的运行更快；.pyc 或.pyo 文件只是在它们加载的时候更快一些。

- compileall 模块可以为指定目录中的所有模块创建.pyc 文件（或者使用 -O 参数创建.pyo 文件）。

6.2 标准模块

Python 带有一个标准模块库，并发布由独立的文档，名为 Python 库参考手册（此后称其为“库参考手册”）。有一些模块内置于解释器之中，这些操作的访问接口不是语言内核的一部分，但是已经内置于解释器了。这既是为了提高效率，也是为了给系统调用等操作系统原生访问提供接口。这类模块集合是一个依赖于底层平台的配置选项。例如，winreg 模块只提供在 Windows 系统上才有。有一个具体的模块值得注意：sys，这个模块内置于所有的 Python 解释器。变量 sys.ps1 和 sys.ps2 定义可主提示符和辅助提示符字符串：

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C>'
C> print('Yuck!')
Yuck!
C>
```

这两个变量只在解释器的交互模式下有意义。

变量 sys.path 是解释器模块搜索路径的字符串列表。它由环境变量 PYTHONPATH 初始化，如果没有设定 PYTHONPATH ，就由内置的默认值初始化。你可以用标准的字符串操作修改它:

```
>>> import sys
>>> sys.path.append('ufs/guido/lib/python')
```

6.3 dir() 函数

内置函数 dir() 用于按模块名搜索块定义，它返回一个字符串类型的存储列表：

```
>>> import fibo, sys
>>> dir(fibo)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',

'__package__', '__spec__', 'fib', 'fib2']
```

```
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__interactivehook__', '__loader__',

'__name__', '__package__','__spec__', '__stderr__', '__stdin__', '__stdout__',

'_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',

'_git', '_home', '_xoptions', 'abiflags', 'api_version', 'argv', 'base_exec_prefix',

'base_prefix', 'builtin_module_names', 'byteorder','call_tracing', 'callstats',

'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',

'executable', 'exit','flags', 'float_info', 'float_repr_style', 'get_asyncgen_hooks',

'get_coroutine_wrapper', 'getallocatedblocks', 'getcheckinterval','getdefaultencoding',

 'getdlopenflags', 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',

 'getrecursionlimit', 'getrefcount','getsizeof', 'getswitchinterval', 'gettrace',

 'hash_info', 'hexversion', 'implementation', 'int_info', 'intern', 'is_finalizing', 'maxsize',

'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform',

 'prefix', 'ps1', 'ps2', 'set_asyncgen_hooks','set_coroutine_wrapper', 'setcheckinterval',

 'setdlopenflags', 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',

'stdin', 'stdout', 'thread_info', 'version', 'version_info', 'warnoptions']
```

无参调用时，dir() 函数返回当前定义的命名：

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',

'__name__', '__package__', '__spec__', 'a', 'fib', 'fibo', 'sys']
```

注意该列表列出了所有类型的名称：变量，模块，函数，等等。

dir() 不会列出内置函数和变量名。如果你想列出这些内容，它们在标准模块 builtins 中定义：

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',

'BrokenPipeError', 'BufferError','BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',

'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError','DeprecationWarning',

'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError',

'FileNotFoundError','FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',

'ImportError', 'ImportWarning','IndentationError', 'IndexError','InterruptedError',

'IsADirectoryError', 'KeyError', 'KeyboardInterrupt','LookupError', 'MemoryError',

'ModuleNotFoundError', 'NameError','None', 'NotADirectoryError','NotImplemented',

'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning',

'PermissionError','ProcessLookupError', 'RecursionError', 'ReferenceError',

'ResourceWarning', 'RuntimeError', 'RuntimeWarning','StopAsyncIteration', 'StopIteration',

'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit','TabError', 'TimeoutError',

'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError','UnicodeEncodeError',

'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',

'Warning', 'ZeroDivisionError', '_','__build_class__', '__debug__', '__doc__',

'__import__', '__loader__', '__name__', '__package__', '__spec__','abs', 'all', 'any',

'ascii', 'bin','bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod',

'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',

'enumerate', 'eval','exec', 'exit', 'filter', 'float', 'format', 'frozenset',

'getattr', 'globals', 'hasattr', 'hash','help', 'hex', 'id','input', 'int',
```

```
'isinstance', 'issubclass', 'iter', 'len', 'license', 'list','locals', 'map',

'max', 'memoryview', 'min', 'next', 'object','oct', 'open', 'ord', 'pow',

'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr',

'slice', 'sorted','staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

6.4 包

包通常是使用用” 圆点模块名” 的结构化模块命名空间。例如，名为 A.B 的模块表示了名为 A 的包中名为 B 的子块。正如同用模块来保存不同的模块架构可以避免全局变量之间的相互冲突，使用圆点模块名保存像 NumPy 或 Python Imaging Library 之类的不同类库架构可以避免模块之间的明明冲突。

假设你现在想要设计一个模块集（一个 “包”）来统一处理声音文件和声音数据。存在几种不同的声音格式（通常由他们的扩展名来标识，例如：.wav, .aiff, .au），于是，为了在不同类型的文件格式之间转换，你需要维护一个不断增长的包集合。可能你还想要对声音数据做很多不同的操作（例如混音，添加回声，应用平衡功能，创建一个人造效果），所以你要加入一个无限流模块来执行这些操作。你的包可能会是这个样子（通过分级的文件体系来进行分组）：

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
```

...

当导入这个包时，Python 通过 sys.path 搜索路径查找包含这个包的子目录。

为了让 Python 将目录当做内容包，目录中必须包含 ___init___.py 文件。这是为了避免一个含有烂俗名字的目录无意中隐藏了稍后在模块中出现的有效模块，比如 string。虽简单的情况下，只需要一个空的 ___init___.py 文件即可。当然他也可以执行包的初始化代码，或者定义稍后介绍的 ___all___ 变量。

用户可以每次只导入包里的特定模块，例如：

import sound.effects.echo

这样就导入了 sound.effectd.echo 子模块。它必须通过完整的名称来引用：

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

导入包时有一个可以选择的方式：

```
from sound.effectd import echo
```

这样就加载了 echo 子模块，并且使得它在没有包前缀的情况下也可以使用，所以它可以如下方式调用：

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

还有另一种变体用于直接导入函数或变量:

```
from sound.effects.echo import echofilter
```

这样就又一次加载了 echo 子模块，但这样就可以直接调用它的 echofilter() 函数：

```
echofilter(input, output, delay=0.7, atten=4)
```

需要注意的是使用 from package import item 方式导入包时，这个子项（item）既可以是包中的一个子模块（或一个子包），也可以是包中定义的其它命名，像函数、类或变量。import 语句首先核对是否包中有这个子项，如果没有，它假定这是一个模块，并尝试加载它。如果没有找到它，会引发一个 ImportError 异常。

相反，使用类似 import item.subitem.subsubitem 这样的语法时，这些子项必须是包，最后的子项可以是包或模块，但不能是前面子项中定义的类、函数或变量。

6.4.1 从 * 导入包

当用户写下 from sound.effects import * 时会发生什么事？理想中，总是希望在文件系统中找出包中所有的子模块，然后导入它们。这可能会花掉很长时间，并且出现期待之外的边界效应，导出了希望只能显示导入的包。

对于包的作者来说唯一的解决方案就是提供一个明确的包索引。import 语句按如下条件进行转换：执行 from package import * 时，如果包中的 ___init___.py 代码定义了一个名为 ___all___ 的列表，就会按照列表中给出的模块名进行导入。新版本的包发布时作者可以任意更新这个列表。如果包作者不想 import的时候导入他们的包中所有模块，那么也可能会决定不支持它（import）。例如，sound/effects/___init___.py 这个文件可能包括如下代码：

```
__all__ = ["echo", "surround", "reverse"]
```

这意味着 from sound.effects import 语句会从 sound 包中导入以上三个已命名的子模块。

如果没有定义 ___all___, from sound.effects import 语句不会从 sound.effects 包中导入所有的子模块。无论包中定义多少命名，只能确定的是导入了 sound.effects 包（可能会运行 ___init__）每一个命名（以及明确导入的子模块）。同样也包括了前述的 import 语句从包中明确导入的子模块，考虑以下代码：

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

这个例子中, echo 和 surround 模块导入了当前的命名空间，这是因为执行 from...import 语句时它们已经定义在 sound.effects 包中了（定义了 ___all___ 时也会同样工作）。

尽管某些模块设计为使用 import * 时它只导出符合某种规范/模式的命名，仍然不建议在产生代码中使用这种方法。

记住，from package import specific_submodule 没有错误！事实上，除非导入的模块需要使用其它包中的同名子模块，否则这是推荐的写法。

6.4.2 包内引用

如果包中使用了子包结构（就像示例中的 sound 包），可以按绝对位置从相邻的包中引入子模块。例如，如果 sound.filters.vovoder 包需要使用 sound.effects 包中的 echo 模块，它可以 from sound.Effects import echo。

你可以用这样的形式 from module import name 来写显示的相对位置导入。那些显式相对导入用点号标明关联导入当前和上级包。以 surround 模块为例，你可以这样用：

```
from . import echo
from .. import formats
from ..filters import equalizer
```

需要注意的是显式或隐式相对位置导入都基于当前模块的命名。因为主模块的名字总是 “___name___”,Python 应用程序的主模块应该总是用绝对导入。

6.4.3 多重目录中的包

包支持一个更为特殊的特性，___path___。在包的 ___init___.py 文件代码执行之前，该变量初始化一个目录名列表。改变量可以修改，它作用于包中的子包和模块的搜索功能。

这个功能可以用于扩展包中的模块集，不过它不常用。

7 输入输出

一个程序可以有几种输出方式：以人类可读的方式打印数据，或者写入一个文件供以后使用。文章将讨论几种可以能性。

7.1 格式化输出

我们有两种大相径庭地输出值方法：表达式语句和 print() 函数（第三种访求是使用文件对象的 write() 方法，标准文件输出可以参考 sys.stdout, 详细内容参见手册）。

通常，你想要对输出做更多的格式控制，而不是简单的打印使用空格分隔的值。有两种方法可以格式化你的输出：第一种方法是由你自己处理整个字符串，通过使用字符串切割和连接操作可以创建任何你想要的输出形式。string 类型包含一些将字符串填充到指定列宽度的有用操作，随后就会讨论这些。第二种方法是使用 str.format() 方法。

标准模块 string 包括了一些操作，将字符串填充入给定列时，这些操作很有用。随后我们会讨论这部分内容。第二种方法是使用 Template 方法。

当然，还有一个问题，如何将值转化为字符串？很幸运，Python 有办法将任意值转为字符串：将它传入 repr() 或 str() 函数。

函数 str() 用于将值转化为适于阅读的形式，而 repr() 转化为供解释器读取的形式（如果没有等价的语法，则会发生 SyntaxError 异常）某对象没有适于人阅读的解释形式的话，str() 会返回与 repr() 等同的值。很多类型，诸如数值或链表、字典这样的结构，针对各函数都有着统一的解读方式。字符串和浮点数，有着独特的解读方式。

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is40000...
>>> # The repr() of a string adds string quotes and backlashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

有两种方式可以写平方和立方表：

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
```

```
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(注意第一个例子，print() 在每列之间加了一个空格，它总是在参数间加空格。)

以上是一个 str.rjust() 方法的演示，它把字符串输出到一列，并通过向左侧填充空格来使其右对齐。类似的方法还有 str.ljust() 和 str.center()。这些函数只是输出新的字符串，并不改变什么。如果输出的字符串太长，它们也不会截断它，而是原样输出，这会使你的输出格式变得混乱，不过总强过另一种选择（截断字符串），因为那样会产生错误的输出值（如果你确实需要截断它，可以使用切割操作，例如：x.lijust(n) [:n]）。

还有另一个方法，str.zfill() 它用于向数值的字符串表达左侧填充 0。该函数可以正确理解正负号：

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

方法 str.format() 的基本用法如下：

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

大括号和其中的字符会被替换成传入 str.format() 的参数。大括号中的数值指明使用传入 str.format() 方法的对象中的哪一个：

```
>>> print('{0} and {1}'. format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'. format('spam', 'eggs'))
eggs and spam
```

如果在 str.format() 调用时使用关键字参数，可以通过参数名来引用值:

```
>>> print('This {food} is {adjective}.'.format(
...     food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

位置参数和关键字参数可以随意组合:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred', other='Georg'))
The story of Bill, Manfred, and Georg.
```

‘!a’(应用 ascii()), ‘!s’(应用 str()) 和 ‘!r’(应用 repr()) 可以在格式化之前转换值:

```
>>> import math
>>> print('The value of PI is approximately {}.'.format(math.pi))
The value of PI is approximately 3.141592653589793.
>>> print('The value of PI is approximately {!r}.'.format(math.pi))
The value of PI is approximately 3.141592653589793.
```

字段名后允许可选的 ‘:’ 和格式指令。这允许对值的格式化加以更深入的控制。下列将 P 转为三位精度。

```
>>> print('The value of PI is approximately {0:.3f}.'.format(math.pi))
The value of PI is approximately 3.142.
```

在字段后的 ‘:’ 后面加一个整数会限定该字段的最小宽度，这在美化表格时很有用:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, phone))
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

如果你有个实在是很长的格式化字符串，不想分割它。如果你可以用命名来引用被格式化的变量而不是位置就好了。有个简单的方法，可以传入一个字典，用中括号（‘[]’）访问它的键:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

这种方式与新的内置函数 vars() 组合使用非常有效。该函数返回包含所有局部变量的字典。

要更进一步了解字符串格式化方法 str.format(), 参见格式字符串语法。

7.1.1 旧式的字符串格式化

操作符% 也可用于字符串格式化。它以类似 sprintf()-style 的方式解析左参数，将右参数应用于此，得到格式化操作生成的字符串，例如：

```
>>> import math
>>> print('The value of PI is approximately %5.3f.'% math.pi)
The value of PI is approximately 3.142.
```

7.2 文件读写

函数 open() 返回文件对象，通常的用法需要两个参数：open(filename, mode)。

```
>>> f = open('workfile', 'w')
```

第一个参数是一个含有文件名的字符串。第二个参数也是一个字符串，含有描述如何使用该文件的几个字符。mode 为 'r' 时表示只是读取文件；'w' 表示只是写入文件（已经存在的同名文件将被删掉）；'a' 表示打开文件进行追加，写入到文件中的任何数据将自动添加到末尾。'r+' 表示打开文件进行读取和写入。mode 参数是可选的，默认为 'r'。

通常，文件以文本打开，这意味着，你从文件读出和向文件写入的字符串会被特定的编码方式（默认是 UTF-8）编码。模式后面的 'b' 以二进制模式打开文件：数据会以字节对象的形式读出和写入。这种模式应该用于所有不包含文本的文件。

在文本模式下，读取时默认会将平台有关的行结束符（Unix 上是 \n , Windows 上是 \r\n）转换为 \n。在文本模式下写入时，默认会将出现的 \n 转换成平台有关的行结束符。这种暗地里的修改对 ASCII 文本文件没有问题，但会损坏 JPEG 或 EXE 这样的二进制文件中的数据。使用二进制模式读写此类文件时要特别小心。

7.2.1 文件对象方法

示例默认文件对象 f 已经创建。

要读取文件内容，需要调用 f.read(size), 该方法读取若干数量的数据并以字符串形式返回其内容，size 是可选的数值，指定字符串长度。如果没有指定 size 或者指定为负数，就会读取返回整个文件。当文件大小为当前机器内存两倍时，就会产生问题。反之，会尽可能按比较大的 size 读取和返回数据。如果到了文件末尾，f.read() 会返回一个空字符串（''）：

```
>>> f = open('workfile')
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

f.readline() 从文件中读取单独一行，字符串结尾会自动加上一个换行符（\n），入过当文件最后一行没有以换行符结尾时，这一操作才会被忽略。这样返回值就不会有混淆，如果 f.readline() 返回一个空字符串，那就表示到达了文件末尾，如果是一个空行，就会描述为 '\n' ，一个只包含换行符的字符串：

```
>>> f.readline()
'This is the entire file.\n'
```

```
>>> f.readline()
'Second line of the file.\n'
>>> f.readline()
''
```

你可以循环遍历文件对象来读取文件中的每一行。这是一种内存高效、快速，并且代码简介的方法：

```
>>> f = open('workfile')
>>> for line in f:
...     print(line, end='')
...
This is the entire file.
Second line of the file.
```

如果你想把文件中的所有行读到一个列表中，你也可以使用 list(f) 或者 f.readlines()。

f.write(string) 方法将 string 的内容写入文件，并返回写入字符的长度：

```
>>> f = open('workfile', 'w')
>>> f.write('This is a test\n')
15
```

想要写入其他非字符串内容，首先要将它转换为字符串：

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
18
```

f.tell() 返回一个整数，代表文件对象在文件中的指针位置，该数值计量了自文件开头到指针处的比特数。需要改变文件对象指针，使用 f.seek(offset,from_what)。指针在该操作中从指定的引用位置移动 offset 比特，引用位置由 from_what 参数指定。from_what 值为 0 表示自文件起始处开始，1 表示自当前文件指针位置开始，2 表示自文件末尾开始。from_what 可以忽略，其默认值为 0，此时从文件头开始：

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
30
>>> f.read(1)
b'4'
```

在文本文件中（没有以 b 模式打开），只允许从文件头开始寻找（有个例外是用 seek(0,2) 寻找文件的最末尾处）而且合法的偏移值只能是 f.tell() 返回的值或者是零。其它任何偏移值都会产生未定义的行为。

当你使用完一个文件时，调用 f.close() 方法就可以关闭它并释放其占用的所有系统资源。在调用 f.cloae() 方法后，试图再次使用文件对象将会自动失败。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: read of closed file
```

用关键字 with 处理文件对象是个好习惯。它的先进之处在于文件用完后会自动关闭，就算发生异常也没关系。它是 try-finally 块的简写：

```
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

文件对此昂还有一些不太常用的附加方法，比如 isatty() 和 trucate() 在库参考手册中有文件对象的完整指南。

7.2.2 使用 json 存储结构化数据

从文件中读写字符串很容易。数值就要多费点周折，因为 read() 方法只会返回字符串，应将其传入 int() 这样的函数，就可以将 '123' 这样的字符串转换为对应的数值 123。当你想要保存更为复杂的数据类型，例如嵌套的列表和字典，手工解析和序列化它们将变得更为复杂。

好在用户不是非得自己编写和调试保存复杂数据类型的代码，Python 允许你使用常用的数据交换格式 JSON(JavaScript Object Notation)。标准模块 json 可接受 Python 数据结构，并将它们转换为字符串表示形式；此过程称为序列化。从字符串表示形式重新构建数据结构称为反序列化。序列化和反序列化的过程中，表示该对象的字符串可以存储在文件或数据中，也可以通过网络连接传送给远程的机器。

注解：JSON 格式经常用于现代应用程序中进行数据交换。许多程序员都已经熟悉它了，使它成为相互协作的一个不错的选择。

如果你有一个对象 x，你可以用简单的一行代码查看其 JSON 字符串表示形式：

```
>>> json.dumps([1, 'simple', 'list'])
'[1, 'simple', 'list']'
```

dumps() 函数的另外一个变体 dunp()，直接将对象序列化到一个文件。所以如果 f 是为写入而打开的一个文件对象，我们可以这样做：

```
json.dump(x, f)
```

为了重新解码对象，如果 f 是为读取而打开的文件对象：

x = json.load(f)

这种简单的序列化技术可以处理列表和字典，但序列化任意类实例为 JSON 需要一点额外的努力。json 模块的手册对此有详细的解释。

参见：pickle-pickle 模块与 JSON 不同，pickle 是一个协议，它允许任意复杂的 Python 对象的序列化。因此，它只能用于 Python 而不能用来与其他语言编写的应用程序进行通信。默认情况下它也是不安

全的：如果数据由熟练的攻击者精心设计，反序列化来自一个不守信任源的 pickle 数据可以执行任意代码。

8 错误和异常

至今为止还没有进一步的谈论过错误信息，不过在你已经试验过的那些例子中，可能已经遇到过一些。Python 中（至少）有两种错误：语法错误和异常（syntax errors 和 exceptions）。

8.1 语法错误

语法错误，也被称作解析错误，也许是你学习 Python 过程中最常见抱怨：

```
>>> while True print('Hello world')
  File "<stdin>", line 1, in ?
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

语法分析器指出错误行，并且在检测到错误的位置前面显示一个小箭头。错误是由箭头前面的标记引起的（或者至少是这么监测的）：这个例子中，函数 print() 被发现存在错误，因为它前面少了一个冒号（':'）。错误会输出文件名和行号，所以如果是从脚本输入的你就知道去哪里检查错误了。

8.2 异常

即使一条语句或表达式在语法上是正确的，当试图执行它时也可能会引发错误。运行期检测到错误称为异常，并且程序不会无条件的崩溃：很快，你将学到如何在 Python 程序中处理它们。然而，大多数异常都不会被程序处理，向这里展示的一样最终会产生一错误信息。异常也有不同的类型，异常类型做为错误信息的一部分显示出来：示例中的异常分别为零除错误（ZeroDivisionError ），命名错误（NameError）和类型错误（TypeError ）。

8.3 异常处理

通过编程处理选择的异常是可行的。看一下下面的例子：它会一直要求用户输入，直到输入一个合法的整数为止，但允许用户中断这个程序（使用 Control-C 或系统支持的任何方法）。注意：用户产生的中断会引发一个 KeyboardInterrupt 异常。

```
>>> while True:
...     try:
...             x = int(input("Please enter a number: "))
...             break
...     except ValueError:
...             print("Oops! That was no valid number. Try again...")
...
```

try 语句按如下方式工作。

- 首先，执行 try 子句（在 try 和 except 关键字之间的部分）。
- 如果没有异常发生，except 子句在 try 语句执行完毕后就被忽略了。
- 如果在 try 子句执行过程中发生了异常，那么该子句其余部分就会被忽略。

  如果异常匹配于 except 关键字后面指定的异常类型，就执行对应的 except 子句。然后继续执行 try 语句之后的代码。
- 如果发生了一个异常，在 except 子句中没有与之匹配的分支，它就会传递到上一级 try 语句中。

  如果最终仍找不到对应的处理语句，它就成为一个未处理异常，终止程序运行，显示提示信息。

9 类

Python 的类机制通过最小的新语法和语义在语言中实现了类。它是 C++ 或者 Modula-3 语言中类机制的混合。就像模块一样，Python 的类并没有在用户和定义之间设立绝对的屏障，而是依赖于用户不去 “强行闯入定义” 的优雅。另一方面，另一方面，类的大多数重要特性都被完整的保留下来：类继承机制允许多重继承，派生类可以覆盖（override）基类中的任何方法或类，可以使用相同的方法名称调用基类的方法。对象可以包含任意数量的私有数据。

9.1 术语相关

对象具有特性，并且多个名称（在多个作用域中）可以绑定在同一个对象上。在其它语言中被称为别名。在对 Python 的第一印象中这通常会被忽略并且当处理不可变基础类型（数字，字符串，元组）时可以被放心的忽略。但是，在调用列表、字典这类可变对象，或者大多数程序外部类型（文件，窗体等）描述实体时，别名对 Python 代码的语义便具有（有意而为）影响。这通常有助于程序的优化，因为在某些方面别名表现的就像是指针。例如，你可以轻易的传递一个对象，因为通过继承只是传递一个指针。并且如果一个方法修改了一个作为参数传递的对象，调用者可以接收这一变化——这消除了两种不同的参数传递机制的需要，像 Pascal 语言。

9.2 Python 作用域和命名空间

命名空间是从命名到对象的映射。当前命名空间主要是通过 Python 字典实现的，不过通常不关心具体的实现方式（除非出于性能考虑），以后也有可能会改变其实现方式。以下有一些命名空间的例子：内置命名（像 abs() 这样的函数，以及内置异常名）集，模块中的全局命名，函数调用中的局部命名。某种意义上讲对象的属性集也是一个命名空间。关于命名空间需要了解一件很重要的事就是不同命名空间中的命名没有任何联系，例如两个不同的模块可能都会定义一个名为 maximize 的函数而不会发生混淆-用户必须以模块名为前缀来引用它们。

顺便提一句，我称 Python 中任何一个 “.” 之后的命名为属性 –例如，表达式 z.real 中的 real 是对象 z 的一个属性。严格来讲，从模块中引用命名是引用属性：表达式 modname.funcname 中，nodname 是一个模块对象，funcname 是它的一个属性。因此，模块的属性和模块中的全局命名有直接的映射关系：他们共享同一命名空间。

属性可以是只读过或写的。后一种情况下，可以对属性赋值。你可以这样作：modname.the_answer = 42。可写的属性也可以用 del 语句删除。例如：del modname.the_answer 会从 modname 对象中删除 the_answer 属性。

作用域就是一个 Python 程序可以直接访问命名空间的正文区域。这里的直接访问意思是一个对名称的错误引用会尝试在命名空间内查找。尽管作用域是静态定义，在使用时他们都是动态的。每次执行时，至少有三个命名空间可以直接访问的作用域嵌套在一起：

- 包含局部命名的使用域在最里面，首先被搜索；其次搜索的是中层的作用域，这里包含了统计函数；最后搜索最外面的作用域，它包含内置命名。
- 首先搜索最内层的作用域，它包含局部命名任意函数包含的作用域，是内层嵌套作用域搜索起点，包含非局部，但是也非全局的命名
- 接下来的作用与包含当前模块的全局命名
- 最外层的作用域（最后搜索）是包含内置命名空间

如果一个命名声明为全局的，那么对它的所有引用和赋值会直接搜索包含这个模块全局命名的作用域。如果要重新绑定最里层作用域之外的变量，可以使用 nonlocal 语句；如果不声明为 nonlocal，这些变量将是只读的（对这样的变量赋值会在最里面的作用域创建一个新的局部变量，外部具有相同命名的那个变量不会改变）。

Python 的一个特别之处在于：如果没有使用 global 语法，其赋值操作总是在最里层的作用域。赋值不会复制数据，只是将命名绑定到对象。删除也是如此：del x 只是从局部作用域的命名空间中删除命名 x 。事实上，所有引入新命名的操作都作用于局部作用域。特别是 import 语句和函数定义将模块名或函数绑定于局部作用域（可以使用 global 语句将变量引入到全局作用域）。

global 语句用以指明某个特定的变量为全局作用域，并重新绑定它。nonlocal 语句用以指明某个特定的变量为封闭作用域，并重新绑定它。

9.2.1 作用域和命名空间示例

以下是一个示例，演示了如何引用不同作用域和命名空间，以及 global 和 nonlocal 如何影响变量绑定：

```
def scope_test():
...     def do_local():
...             spam = "local spam"
...     def do_nonlocal():
...             nonlocal spam
...             spam = "nonlocal spam"
...     def do_global():
...             global spam
...             spam = "global spam"
...     spam = "test spam"
...     do_local()
...     print("After nonlocal assignment:", spam)
...     do_nonlocal()
...     print("After global assignment:", spam)
... scope_test()
... print("In global scope:", spam)
```

以上示例代码的输出为：

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

注意：local 赋值语句是无法改变 scope_test 的 spam 绑定。nonlocal 赋值语句改变了 scope_test 的 spam 绑定，并且 global 赋值语句从模块级改变了 spam 绑定。

你也可以看到在 global 赋值语句之前对 spam 是没有预先绑定的。

9.3 初识类

引入了一些新语法：三种新的对象类型和一些新的语义。

9.3.1 类定义语法

类定义最简单的形式如下：

```
class ClassName:
  <statement-1>
  .
  .
  .
  <statement-N>
```

类的定义（def 语句），要先执行才能生效。（你当然可以把它放进 if 语句的某一分支，或者一个函数的内部。）

进入类定义部分后，会创建出一个新的命名空间，作为局部作用域。因此，所有的赋值成为这个新命名空间的局部变量。特别是函数定义在此绑定了新的命名。

定义完成时（正常退出），就创建了一个类对象。基本上它是对类定义创建的命名空间进行了一个包装；我们在下一节进一步学习类对象的知识。原始的局部作用域（类定义引入之前生效的那个）得到恢复，类对象在这里绑定到类定义头部的类名（例子中是 ClassName）。

9.3.2 类对象

类对像支持两种操作：属性引用和实例化。

属性引用使用和 Python 中所有的属性引用一样的标准语法：obj.name。类对象创建后，类名空间中所有的命名都是有效属性名。所以如果类定义是这样：

```
>>> class MyClass:
...     """A simple example class"""
...     i = 12345
...     def f(self):
...             return 'hello world'
```

那么 MyClass.i 和 MyClass.f 是有效的属性应用，分别返回一个整数和一个方法对象。也可以对类属性赋值，你可以通过给 MyClass.i 赋值来修改它。__doc__ 也是一个有效的属性，返回类的文档字符串："A simple example class"。

类的实例化使用函数符号。只要将类对像看做是一个返回新的类实力的无参数函数即可。例如（假设沿用前面的类）：

```
x = MyClass()
```

以上创建了一个新的类实例并将该对象赋给局部变量 x。

这个实例化操作（"调用"一个类对象）来创建一个空的对象。很多类都倾向于将对象创建为有初始状态的。因此类可能会定义一个名为可能会定义一个名为 __inti__() 的特殊方法，像下面这样：

```
def __init__(self):
    self.data = []
```

类定义了 __init__() 方法的话，类的实例化操作会自动为新创建的类实例调用 __init__() 方法。所以在下例中，可以这样创建一个新的实例：

```
x = MyClass()
```

当然，出于弹性的需要，__inti__() 方法可以有参数。事实上，参数通过 __inti__() 传递到类的实例化操作上。例如，

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...             self.r = realpart
...             self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 实例对象

实例对象唯一可用的操作就是属性引用。有两种有效的属性名。

数据属性相当于 Smalltalk 中的"实例变量"或 C++ 中的"数据成员"。和局部变量一样，数据属性不需要声明，第一次使用时它们就会生成。例如，如果 x 是前面创建的 MyClass 实例，下面这段代码会打印出 16 而在堆栈中留下多余的东西：

```
>>> x.counter = 1
>>> while x.counter < 10:
...     x.counter = x.counter * 2
... print(x.counter)
... del x.counter
```

另一种为实例对象所接受的引用属性是方法。方法是"属于"一个对象的函数。（在 Python 中，方法不止是类实例多独有：其它类型的对象也可有方法。例如，链表对象有 append, insert, remove, sort 等等方法。然而，在后面的介绍中，除非特别说明，我们提到的方法特指类方法）

实例对象的有效名称依赖于它的类。按照定义，类中所有（用户定义）的函数对象对应它的示例中的方法。所以在我们的例子中，x.f 是一个有效的方法引用，因为 MyClass.f 是一个函数。但 x.i 不是，因为 MyClass.i 不是函数。不过 x.f 和 MyClass.f 不同，它是一个方法对象，不是一个函数对象。

9.3.4 方法对象

通常，方法通过右绑定方式调用：

x.f()

在 MyClass 示例中，这会返回字符串 'hello world'。然而，也不是一定要直接调用方法。x.f 是一个方法对象，他可以存储起来以后调用。例如：

```
xf = x.fail
while True:
  print(xf())
```

会不断的打印 hello world。调用方法时发生了什么？你可能注意到调用 x.f() 时没有引用前面标出的变量，尽管在 f() 的函数定义中指明了一个参数。这个参数怎么了？事实上如果函数调用中缺少参数，Python 会抛出异常－－甚至这个参数实际上没什么用……

9.3.5 类和实例变量

一般来说，实例变量用于对每一个实例都是唯一的数据，类变量用于类的所有实例共享的属性和方法：

```
  class Dog:
    kind = 'canine'           #class variable shared by all instances
    def __init__(self, name):
        self.name = name      #instance variable unique to each instance
>>> d = Dog('Fibo')
>>> e = Dog('Buddy')
>>> d.kind                  # share by all dogs
'canine'
>>> e.kind                   # share by all dogs
'canine'
>>> d.name                  # unique to d
'Fibo'
>>> e.name                 # unique to e
'Buddy'
```

正如在术语相关讨论的，可变对象，例如列表和字典，的共享数据可能带来以外的效果。例如，下面代码中的 tricks 列表不应该用作类变量，因为所有的 Dog 实例将共享同一个列表：

```
class Dog:

    tricks = []             # mistaken use of a class variable
```

```
    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

这个类的正确设计应该使用一个实例变量：

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 一些说明

数据属性会覆盖同名的方法属性。为了避免意外的名称冲突，这在大型程序中是极难发现的 Bug，使用一些约定来减少冲突的机会是明智的。

一般，方法的第一个参数被命名为 self。这仅仅是一个约定：对 Python 而言，名称 self 绝对没有任何特殊含义。（但是请注意：如果不遵循这个约定，对其他的 Python 程序员而言你的代码可读性就会变差，而且有些类查看器程序也可能是遵循此约定编写的。）

类属性的任何函数对象都为那个类的实例定义了一个方法。函数定义代码不一定非得定义在类中：也可以将一个函数对象赋值给类中的一个局部变量。例如：

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

现在 f，g 和 h 都是类 C 的属性，引用的都是函数对象，因此他们都是 C 实例的方法 h 严格等于 g。要注意的是这种习惯通常只会迷惑程序的读者。

通过 self 参数的方法属性，方法可以调用其它的方法：

```
class Bag:
  def __init__(self):
    self.data = []
  def add(self, x):
    self.data.append(x)
  def addtwice(self, x):
    self.add(x)
    self.add(x)
```

每个值都是一个对象，因此每个值都有一个类（class）（也称为它的类型（type））， 它存储为 object.___class___ 。

9.5 继承

当然，如果一种语言不支持继承就，“类” 就没有什么意义。派生类的定义如下所示：

```
class DerivedClassName(BaseClassName):
  <statement-1>
  .
  .
  .
  <statement-N>
```

命名 BaseClassName(示例中的基类名）必须与派生类定义在一个作用域内。除了类，还可以用表达式，基类定义在另一个模块中时这一点非常有用：

class DerivedClassName(modname.BaseClassName):

派生类定义的执行过程和基类是一样的。构造派生类对象时，就记住了基类。这在解析属性引用的时候尤其有用：如果再类中找不到请求调用的属性，就搜索基类。如果基类是由别的类派生而来，这个规则会递归的应用上去。

派生类色实例化没有什么特别之处：DerivedClassName()（示例中的派生类）创建一个新的类实例。方法引用按如下规则解析：搜索对应的类属性，必要时沿基类链逐级搜索，如果找到了函数对象这个方法引用就是合法的。

派生类可能会覆盖其基类的方法。因为方法调用同一个对象中的其它方法时没有特权，基类的方法调用同一个基类的方法时，可能实际上最终调用了派生类中的覆盖方法。（对于 C++ 程序员来说，Python 中的所有方法本质上都是虚方法。）

派生类中的覆盖方法可能是想要扩充而不是简单的替代基类中的重名方法。有一个简单的方法可以直接调用基类方法，只要调用：BaseClassName.methodname(self, arguments)。有时这对于客户也很有用。（要注意只有 BaseClassName 在同一全局作用域定义或导入时在能这样用。）

Python 有两个用于继承的函数：

- 函数 isinstance() 用于检查实例类型：isinstance(obj, int) 只有在 obj.__class__ 是 int 或其它从 int 继承的类型
- 函数 issubclass() 用于检查类继承：issubclass(bool, int) 为 True，因为 bool 是 int 的子类。然而，issubclass(float, int) 为 False，因为 float 不是 int 的子类。

9.5.1 多继承

Python 同样有限的支持多继承形式。多继承的类定义性如下例：

```
class DerivedClassName(Base1, Base2, Base3):
  <statement-1>
  .
  .
  .
  <statement-N>
```

在大多数情况下，在最简单的情况下，你能想到的搜索属性从父类继承的深度优先，左到右，而不是搜索两次在同一个类层次结构中，其中有一个重叠。因此，如果在 DerivedClassName（示例中的派生类）中没有找到摸个属性，就会搜索 Base1，然后（递归的）搜索其基类，如果最终没有找到，就搜索 Base2，以此类推。

实际上，super() 可以动态的改变解析顺序。这个方式可见于其它的一些多继承语言，类似 callnext-method，比单继承语言中的 super 更强大。

动态调整顺序十分必要的，因为所有的多继承会有一到多个菱形关系（指至少一个祖先类可以从子类经由多个继承路径到达）。例如，所有的 new-style 类继承自 object ，所以任意的多继承总是会有多于一条继承路径到达 object 。

9.6 私有变量

只能从对象内部访问的 “私有” 实例变量，在 Python 中不存在。然而，也有一个变通的访问用于大多数 Python 代码：以一个下划线开头的命名（例如 ___spam）会被处理为 API 的非公开部分（无论它是一个函数、方法或数据成员）。他会被视为一个现实细节，无需公开。

因为有一个正当的类私有成员用途（即避免子类里定义的命名与之冲突），Python 提供了对这种结构的有限支持，称为 name mangling（命名编码）。任何形如 ___spam 的标识（前面至少两个下划线，后

面至多一个），被替代为 _classname__spam，去掉前导下划线的 classname 即当钱的类名。此语法不关注标识的位置，只要求在类定义内。

名称重整是有助于子类重写方法，而不会打破组内的方法调用。例如：

```
class Mapping:
  def __int__(self, iterable):
    self.items_list = []
    self.__update(iterable)
  def update(self, iterable):
    for item in iterable:
      self.items_list.append(item)
  __update = update     # private copy of original update() method
  class MappingSubclass(Mapping):
    def update(self, key, values):
      # provides new signature for update()
      # but does not break __init__()
      for item in zip(keys, values):
        self.items_list.append(item)
```

需要注意的是编码规则设计为尽可能的避免冲突，被认作为私有的变量仍然有可能被访问或修改。在特定的场合它也是有用的，比如调试的时候。

要注意的是代码传入 exec()，eval() 时不考虑所调用的类的类名，视其为当前类，这类似于 global 语句的效应，已经按字节编译的部分也有同样的限制。这也同样作用于 getattr()，setattr() 和 delattr()，像直接引用 __dict__ 一样。

9.7 补充

有时类似于 Pascal 中“记录（record）”或 C 中“结构（struct）”的数据类型很有用，它将一组已命名的数据项绑定在一起。一个空的类定义可以很好的实现它：

```
class Employee:
  pass

john = Employee()   # Create an empty employee record
# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

某一段 Python 代码需要一个特殊的抽象数据结构的话，通常可以传入一个类，事实上这模仿了该类的方法。例如，如果你有一个用于从文件对象中格式化数据的函数，你可以定义一个带有 read() 和 readline() 方法的类，以此从字符串缓冲读取数据，然后将该类的对象作为参数传入前述的函数。

实例方法对象也有属性：m.__self__ 是一个实例方法所属的对象，而 m.__func__ 是这个方法对应的函数对象。

9.8 异常也是类

用户自定义异常也可以是类。利用这个机制可以创建可扩展的异常体系。

以下是两种新的，有效的（语义上的）异常抛出形式，使用 raise 语句：

```
raise Class
raise Instance
```

第一种形式中，Class 必须是 type 或其派生类的一个实例。第二种形式是以下形式的简写：

```
raise Class()
```

发生的异常其类型如果是 except 子句中列出的类，或者是其派生类，那么他们就是相符的（反过来说 –发生的异常其类型如果是异常子句中列出的类的基类，它们就不相符）。例如，以下代码会按顺序打印 B，C，D：

```
class B(Exception)
  pass
class C(B):
  pass

for cls in [B, C, D]:
  try:
    raise cls()
  except D:
    print("D")

  except C:
    print("C")
  except B:
    print("B")
```

要注意的是如果异常子句的顺序颠倒过来（execpt B 在最前面），它就会打印 B，B，B第一个匹配的异常被触发。

打印一个异常类的错误信息时，先打印类名，然后是一个空格、一个冒号，然后是用内置函数 str() 将类转换得到的完整字符串。

9.9 迭代器

现在你可能注意到大多数容器对象都可以用 for 遍历：

```
for element in [1, 2, 3]:
  print(element)
for element in (1, 2, 3):
  print(element)
for key in {'one':1, 'two':2}:
```

```
  print(key)
for char in "123":
  print(char)
for line in open("myfile.txt"):
  print(line, end=' ')
```

这种形式的访问清晰、简洁、方便。迭代器的用法在 Python 中普遍而且统一。在后台，for 语句在容器对象中调用 iter()。该函数返回一个定义了 ___next___() 方法的迭代器对象，它在容器中逐一访问元素。没有后续的元素时，___next___() 抛出一个 StopIteration 异常通知 for 语句循环结束。你可以是用内建的 next() 函数调用 ___next___() 方法；以下是其工作原理的示例:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x105864be0>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

了解了迭代器协议的后台机制，就可以很容易的给自己的类添加迭代器行为。定义一个 ___iter___() 方法,使其返回一个带有 ___next___() 方法的对象。如果这个类已经定义了 ___next___(),那么 ___iter___() 只需要返回 self:

```
class Reverse:
    " " "Iterator for looping over a sequence backwards." " "
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> from fibo import Reverse
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<fibo.Reverse object at 0x103605320>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.10 生成器

Generator 是创建迭代器的简单而强大的工具。它们写起来就像是正规的函数，需要返回数据的时候使用 yield 语句。每次 next() 被调用时，生成器恢复它脱离的位置（它记忆语句最后一次执行的位置和所有的数据值）。以下示例演示了生成器可以很简单的创建出来：

```
def reverse(data):
  for index in range(len(data)-1, -1, -1):
    yield data[index]

>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

前一节中描述了基于类的迭代器，他能做的每一件事生成器也能做到。因为自动创建了 ___iter___() 和 ___next___ 方法，生成器显得如此简洁。

另一个关键的功能在于两次执行之间，局部变量和执行状态都自动的保存下来。这使函数很容易写，而且比使用 self.index 和 self.data 之类的方式更清晰。

除了创建和保存程序状态的自动方法，当发生器终结时，还会自动抛出 StopIteration 异常。综上所述，这些功能使得编写一个正规函数成为创建迭代器的最简单方法。

9.11 生成器表达式

有时简单的生成器可以用简洁的方式调用，就像不带中括号的链表推导式。这些表达式视为函数调用生成器而设计的。生成器表达式比完整的生成器定义更简洁，但没有那么多变，而且通常比等价的链表推导式更容易记。

例如：

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> from math import pi, sin
>>> sine_table = {x: sin(x*pi/180) for x in range(0, 91)}

>>> unique_words = set(word  for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

有一个例外。模块对象有一个隐秘的只读对象，名为 ___dict___ ，它返回用于实现模块命名空间的字典，命名 ___dict___ 是一个属性而非全局命名。显然，使用它违反了命名空间实现的抽象原则，应该被严格限制于调试中。