Python3 中文手册

一、开胃菜

1.Python 易于使用，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。因其更多的通用数据类型，Python 比 Awk 甚至 Perl 都适用于更多问题领域，至少大多数事情在 Python 中与其他语言同样简单。

2.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）工具包接口。

3.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。

4.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，这是因为以下几个原因:

（1）高级数据结构使你可以在一条语句中表达复杂的操作;

（2）语句组使用缩进代替开始和结束大括号来组织;

（3）变量或参数无需声明。

5.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（比如某个专业的商业图形库）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或命令行语言。

二、使用Python解释器

1.调用Python解释器

启动Python解释器:

（1）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启动Python解释器

Unix系统：Control+D结束，让解释器以0码退出

Python 解释器具有简单的行编辑功能。在 Unix 系统上，任何 Python 解释器都可能已经添加了 GNU readline 库支持，这样就具备了精巧的交互编辑和历史记录等功能。在 Python 主窗口中输入 Control-P 可能是检查是否支持命令行编辑的最简单的方法。

Python 解释器有些操作类似 Unix shell：当使用终端设备（tty）作为标准输入调用时，它交互的解释并执行命令；当使用文件名参数或以文件作为标准输入调用时，它读取文件并将文件作为脚本执行。

（2）命令行执行pyhton -c command [arg] ...

一般建议命令要用单引号包裹起来

有一些 Python 模块也可以当作脚本使用。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。

(1)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：

1）sys.argv[0] 此时为空字符串)；

2）脚本名指定为 '-' （表示标准输入）时， sys.argv[0] 被设定为 '-')；

3）使用 -c 指令 时，sys.argv[0] 被设定为 '-c')；

4）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。

(2)交互模式

解释器工作于交互模式：从tty读取命令时，根据主提示符来执行，主提示符：（>>>）；继续的部分为从属提示符（...）。

输入多行语句用从属提示符，例如下面的if语句：

```
the world is flat=1
if the world is flat:
    print("Be careful not to fall off!")
```

2.解释器及其环境

(1)源程序编码

默认情况下，Python源文件是UTF-8编码

在首行后插入至少一行特殊的注释行来定义源文件的编码

```
# -*- coding:encoding -*-
```

三、Python简介

表示注释，但是注释不能出现在字符串中

注意：在练习中遇到的从属提示符表示你需要在最后多输入一个空行，解释器才能知道这是一个多行命令的结束

```
    # this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

1.将Python当做计算器

(1)数字

1）int：整数,如2,4,6,10；

```
>>> 2+2
4
>>> 50-5*6
20
>>> 8/5
1.6
```

2）float：小数，如5.0,1.6,2.7等
3）除法“/”永远返回一个浮点数
4）如果使用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；
5）余数：%运算符

```
>>> 17/3
5.666666666666667
>>> 17//3
5
>>> 17%3
2
>>> 5*3+2
17
```

6）乘方：**

```
>>> 5**2
25
>>> 2**7
128
```

7）变量赋值：=

```
>>> width=20
>>> height=5*9
>>> width*height
900
```

变量在使用前必须“定义（赋值）”，否则会出错

```
>>> # try to access an undefined variable
... n
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'n' is not defined
```

8）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

```
>>> 3*3.75/1.5
7.5
>>> 7.0/2
3.5
```

9）交互模式下，最近一个表达式的赋值给变量_。这样我们可以把它当作一个桌面计算器，很方便的用于连续计算，例如

```
>>> tax=12.5/100
>>> price=100.50
>>> price*tax
12.5625
>>> price+_
113.0625
>>> round(_,2)
113.06
```

此变量对于用户是只读的，不要尝试给它一一赋值，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。 10）Python支持其他数字类型，例如Decimal和Fraction，Python还内建了支持复数，使用后缀j或者J表示虚数部分（例如：3+5j）

(2)字符串

1）字符串用'...'或者"..."来标识
"\"可以用来做转义字符

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

2）print()函数可以生成可读性更好的输出，它会省去引号并且打印出转义字符后的特殊字符

```
>>> '"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.
```

3）如果前面带有\的字符被当做特殊字符，可以用原始字符串，在第一个引号的前面加上r：

```
>>> print('C:\some\name')   # here \n means newline!换行了
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

4）多行字符串：”””...””” 或者”’...”’
行尾换行符会自动被包含到字符串中，但是可以在行尾加上\来避免这个行为

```
print("""\
Usage: thingy [OPTIONS]
     -h
     -H hostname
""")
```

5）字符串可以由+操作符连接，可以由*表示重复

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

相邻的两个字符串文本自动连接在一起

```
>>> 'Py' 'thon'
'Python'
```

但是这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
```

6）连接多个变量或者连接一个变量和一个字符串文本，使用 +：

```
>>> prefix = 'Py'
>>> prefix + 'thon'
'Python'
```

在很想切分很长的字符串时很有用：

```
>>> text = ('Put several strings within parentheses '
            'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

7）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'
```

注意：-0实际上就是0，所以它不会导致从右边开始计算 8）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]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters 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：

```
>>> 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]  # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]  # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

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 的字符串其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中的索引点 0...6。第二行给出相应的负索引。切片是从 i 到 j 两个数值标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。例如，word[1:3] 是 2 。

试图使用太大的索引会导致错误:

```
 >>> 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
```

9）Python可以优雅的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代替，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:]
''
```

10）Python的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
```

如果需要一个不同的字符串，应该建一个新的:

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

11）内置函数len()返回字符串长度

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

(3)列表

1）Python有几个复合数据类型，用来表示其他的值，最常用的是list列表：中括号表示，列表的元素不必是同一类型：

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

2）就像字符串（以及其他所有内建的序列类型一样），列表可以被索引和切片：

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
>>> squares[0]    # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

3）所有的切片操作都会返回一个包含请求元素的新列表，这意味着下面的切片操作会返回一个新的（浅）拷贝副本：

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
>>> squares[0]    # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
>>> squares[:]
[1, 4, 9, 16, 25]
```

4）列表也支持连接这样的操作：

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

5）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 cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

6）使用 append() 方法 （后面我们会看到更多关于列表的方法的内容）在列表的末尾添加新的元素:

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

7）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
[]
```

8）内置函数 len() 同样适用于列表:

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

9)允许嵌套列表(创建一个包含其它列表的列表)，例如:

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

2.编程的第一步

写一个生成菲波那契子序列的程序:

```
>>> # 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
```

这个例子出现的新功能:

（1）第一行: 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。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。

（2）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。条件也可以是字符串或列表，实际上可以是任何序列;

（3）所有长度不为零的是 true，空序列是 false。示例中的测试是一个简单的比较。标准比较操作符与 C 相同： < ， > ， == ， <=， >= 和 ! =。

（4）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，一般是4个空格;

（5）大多数文本编辑器提供自动缩进。交互式录入复合语句时，必须在最后输入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。

（6）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（就像我们在前面计算器的例子中那样）。

（7）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
```

（8）用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=',')
...     a, b = b, a+b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

补充:

（1）因为 ** 的优先级高于 -，所以 -3**2 将解释为 -(3**2) 且结果为 -9。为了避免这点并得到 9，你可以使用 (-3)**2。

（2）与其它语言不同，特殊字符例如 \n 在单引号('...')和双引号("...")中具有相同的含义。两者唯一的区别是在单引号中，你不需要转义 " （但你必须转义 '\' ），反之亦然。

四、深入Python流程控制

1.if语句

```
>>> 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
#注意空格的缩进，4个空格，不然程序报错
```

（1）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。
（2）if ... elif ... elif ... 序列用于替代其它语言中的 switch 或 case 语句。

2.for语句

（1）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
```

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

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

3.range()函数

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

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

(2)range(10) 生成了一个包含 10 个值的链表，它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
```

（3）需要迭代链表索引的话，如下所示结合使用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
```

（4）这种场合可以方便地使用enumerate()，详情参见循环技巧
（5）如果只打印一个序列会发生：

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

（6）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，它并不真正构造列表。（7）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）。（8）for语句就是一个迭代器。list()函数是另一个迭代器，他从可迭代（对象）中创建链表：

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

4.break和continue语句，以及循环中的else子句

break语句

（1）break语句：用于跳出最近一级for或while循环
（2）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中止的情况下不会执行。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
#else语句是属于for循环之中，不是if语句
```

（3）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

continue：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
```

5.pass语句

（1）pass语句什么都不做，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)
...
```

（2）这通常用于创建最小结构的类：

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

（3）pass语句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!
...
```

6.定义函数

（1）首先创建一个用来生成指定边界的斐波那列函数：

```
>>> def fib(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
>>>
```

（2）关键字def引入了函数定义。在其后面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。
（3）函数体的第一句：可选的字符串文本，这个字符串是函数的文档字符串，或者成为docstrings（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;包含docstrings是个好的实践，要成为习惯）
（4）函数调用：为函数局部变量生成一个新的符号表。
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中查找，然后是包含函数的局部符号表，然后是全局符号表，最后是内置名字表。
（5）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。
（6）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）。
（7）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。
（7*）实际上，引用对象调用 描述的更为准确。如果传入一个可变对象，调用者会看到调用操作带来的任何变化（如子项插入到列表中）。
（8）一个函数定义会在当前符号表内引入函数名。
（9）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 0x7f41d48def28>
>>> f=fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

（10）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 result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>>
```

这个例子演示了一些新的Python功能:

（1）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。

（2）语句result.append()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的方法的命名。

（3）不同的类型定义不同的方法，不同类型可能有同样的名字的方法，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。在示例中它等同于 result = result + [a]，不过效率更高。

7.深入Python函数定义

在Python中，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。

(1)默认参数值

常用的方式：一个或者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 True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise OSError('uncooperative 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 on, only yes or no!
  ```

1.1 in关键字

in关键字：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
>>>
```

1.2 重要警告

警告：默认值只能被赋值一次。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]
>>>
```

如果不想让默认值在后续调用中累积，可以如下定义函数：

```
>>> def f(a,L=None):
...     if L is None:
...         L=[]
...     L.append(a)
...     return L
...
>>> print(f(1))
[1]
>>> print(f(2))
[2]
>>> print(f(3))
[3]
>>>
```

(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：如下函数：

```
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
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

以下几种调用无效：

```
parrot()                     # required 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')  # unknown keyword argument
```

注意：

（1）函数调用中，关键字的参数必须跟随在位置参数的后面。

（2）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=1000)也是有效的）。（3）任何参数都不可以多次赋值。（4）下面的示例将由这种限制而失败：

```
>>> 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。例如，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])
   #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")
   #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，打印出来的参数的顺序是未定义的。

(3)可变参数列表

函数调用可变个数的参数：这些参数被包装进一个元组，在这些可变个数的参数之前，可以有零到多个普通的参数：

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

通常，这些可变参数是参数列表中的最后一个，因为它们将把所有的剩余输入参数传递给函数。任何出现在 *args 后的参数是关键字参数，这意味着，他们只能被用作关键字，而不是位置参数：

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

```
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

(4)参数列表的分析

相反的情况：当要传递的参数已经是一个列表，但是要调用的函数却接受分开一个个的参数值，这时需要把已有的列表拆开来。例如：内建函数range()需要独立的start，stop参数。可以在调用函数时加一个*操作付来自动把参数列表拆开：

```
>>> list(range(3,6))
[3, 4, 5]
>>> args=[3,6]
>>> list(range(*args))
[3, 4, 5]
>>>
```

以同样的方式，可以使用**操作符分拆关键字参数为字典：

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

(5)Lambda形式

（1）lambda关键字：可以创建短小的匿名函数。一个函数返回它的两个参数的和：

```
lambda a,b:a+b
```

（2）Lambda形式可以用于任何需要的函数对象，单数处于语法限制，它们只能是一个单独的表达式。

（3）类似于嵌套函数定义，lambda形式可以从外部作用域引用变量：

（4）作用一：使用lambda表达式返回一个函数

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

（5）作用二：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')]
```

(6)文档字符串

（1）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。

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

（3）Python 的解释器不会从多行的文档字符串中去除缩进，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，缩进格式显示的不清楚。）留白“相当于”是字符串的起始缩进。每一行都不应该有缩进，如果有缩进的话，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）。

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.

    No, really, it doesn't do anything.
```

(7)函数注解

是关于用户自定义的函数的完全可选的、随意的元数据信息。无论 Python 本身或者标准库中都没有使用函数注解；本节只是描述了语法。第三方的项目是自由地为文档，类型检查，以及其它用途选择函数注解。

注解是以字典形式存储在函数的 _annotations_属性中，对函数的其它部分没有任何影响。参数注解（Parameter annotations）是定义在参数名称的冒号后面，紧随着一个用来表示注解的值得表达式。返回注释（Return annotations）是定义在一个-> 后面，紧随着一个表达式，在冒号与->之间。下面的示例包含一个位置参数，一个关键字参数，和没有意义的返回值

注释示例:

```
>>> def f(ham: 42, eggs: int = 'spam') -> "Nothing to see here":
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...
>>> f('wonderful')
Annotations: {'eggs': <class 'int'>, 'return': 'Nothing to see here', 'ham': 42}
Arguments: wonderful spam
```

8.插曲：编码风格

写更长更复杂的程序时，讨论编码风格让程序更易读:

对于Python，PEP 8 引入的项目遵循的风格：

- 使用 4 空格缩进，而非 TAB
  在小缩进（可以嵌套更深）和大缩进（更易读）之间，4空格是一个很好的折中。TAB会引发一些混乱，弃用。
- 折行以确保其不会超过 79 个字符
  这有助于小显示器用户阅读，也可以让大显示器能并排显示几个代码文件
- 使用空行分隔函数和类，以及函数中的大块代码
- 可能的话，注释独占一行
- 使用文档字符串
- 把空格放到操作符两边，以及逗号后面，但是括号里侧不加空格：a = f(1,2)+g(3,4)
- 统一函数和类命名
  推荐类名用驼峰命名，函数和方法名用 小写_和 _下划线。总是用self作为方法的第一个参数
- 不要使用花哨的编码，如果你的代码的目的是要在国际化环境。Python的默认情况下，UTF-8，甚至普通的 ASCII 总是工作的最好
- 不要使用非 ASCII 字符的标识符，除非是不同语种的会阅读或者维护代码。

五、数据结构

1.关于列表

```
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指引，例如a.insrert(0,x)会插入到整个列表之
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。元素随即从列表中被删除（
list.clear()
#从列表中删除所有的元素。相当于del a[:]
list.index(x)
#返回列表中第一个值为x的元素的索引。如果没有匹配的就会返回一个错误。
list.count(x)
#返回x在列表中出现的次数。
list.sort()
#对列表中的元素就地进行排序。
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[:]。
```

示例：

```
>>> 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]
>>>
```

注意：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；别的语言可能会返回一个变化的对象，允许方法连续执行，像

```
d->insert("a")->remove("b")->sort();
```

(1)把列表当作堆栈使用

列表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）。
（1）用 append() 方法可以把一个元素添加到堆栈顶；
（2）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]
>>>
```

(2)把列表当作队列使用

列表当作队列来使用：堆栈作为特定的数据结构，最先进入的元素最先释放（先进先出）。

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）。

```
>>> 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'])
>>>
```

(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办法。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。

例如：假设我们创建一个squares列表，可以像下面方式：

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

注意：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。如下方法可以计算squares的值，但是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)]
```

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。

如下的列表推导式结合两个列表的元素，如果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)]
```

注意：上面两个方法中的for和if语句的顺序。

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 element
>>> 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, in ?
    [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]
```

列表推导式可以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']
```

(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。

一个由三个长度为4的列表组成的3×4矩阵：

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

交换行和列：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]]
```

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):
...     transposed.append([row[i] for row in matrix])
```

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

反之也是一样的：

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

使用内置函数组成复杂流程语句，zip()函数：

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

2.del语句

del语句：从列表中按给定的索引而不是值来删除一个子项，它不同于有返回值的pop()方法。

del语句：可以从列表中删除切片或清空整个列表（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语句：可以删除整个变量

```
>>> del 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）。

3.元组和序列

列表和字符串是序列类型，另一个标准序列类型：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 1, 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])
```

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。

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）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 序列拆封 非常恰当。序列拆封要求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 ）其实只是元组封装和序列拆封的一个结合。

4.集合Set

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，你必须使用 set() 而不是 。后者用于创建空字典，我们在下一节中介绍的一种数据结构。 示例:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

集合推导式: （类似于列表推导式）

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

5.字典

Python内建数据类型：字典，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，它可以做为关键字，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 append() 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.keys()) ）。[2] 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）。

示例:

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

```
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> '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, 'jack': 4098, 'guido': 4127}
```

字典推导式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，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, 'jack': 4098, 'guido': 4127}
```

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.
```

要按排序后的顺序循环序列的话，使用 sorted() 函数，它不改动原序列，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），首先制作副本，在序列上循环不会隐式地创建副本，切片表示法很方便:

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

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 比较两个对象是否相同；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，在它们之中，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 也称作短路操作符：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 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

注意：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 == 时误用了 = 操作符。

8.比较序列和其它类型

序列对象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。

同类型序列之间比较的实例：

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

注意：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 是等于 0.0 。否则解释器将会触发一个 TypeError 异常，而不是提供一个随意的结果。

六、模块

脚本：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。

模块：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，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这个模块：

```
>>> 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.fib2(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
```

1.深入模块

除了包含函数定义外：模块也可以包含可执行语句。
这些语句一般用来初始化模块。

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表使用。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）习惯是将所有的 import 语句放在模块的开始（或者是脚本），这并非强制。被导入的模块名会放入当前模块的全局符号表中。

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。

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。

注解：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)
```

(1)作为脚本来执行模块

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如下代码：

```
if \_{}name\_{} == "\_{}main\_{}":
    import sys
    fib(int(sys.argv[1]))
```

可以让此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 "main" 文件执行时才被调用：

```
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）。

(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,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 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。

注意：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作模块来加载。这通常会引发错误。

(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.cpython-33.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。 因为某些程序依赖于这些变量的可用性，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）。
- 在 PEP 3147 中有很多关这一部分内容的细节，并且包含了一个决策流程。

2.标准模块

Python 带有一个标准模块库，并发布有独立的文档，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 定义了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')
```

3.dir()函数

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

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_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',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
```

```
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'version_info', 'warnoptions']
```

无参数调用时，dir() 函数返回当前定义的命名:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__', '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', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
 '__debug__', '__doc__', '__import__', '__name__', '__package__', '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']
```

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 的子模块。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命名冲突。

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（通常由它们的扩展名来标识，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搜索路径中出现的有效模块，比如 string。最简单的情况下，只需要一个空的 _init_.py 文件即可。当然它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.effects.echo 子模块。它必需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.effects 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)
```

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。

(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，导出了希望只能显式导入的包。

对于包的作者来说唯一的解决方案就是给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 *
```

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_.py 中的初始化代码）以及包中定义的所有命名会随之导入。这样就从 _init_.py 中导入了每一个命名（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 *
```

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，仍然不建议在生产代码中使用这种写法。

```
from Package import specific_submodule
```

这句语句没有错误！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。

(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.vocoder 包需要使用 sound.effects 包中的 echo 模块，它可以 from sound.Effects import echo。

用这样的形式 from module import name 来写显式的相对位置导入。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
```

注意：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 "_main_"，Python 应用程序的主模块应该总是用绝对导入。

(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。该变量可以修改，它作用于包中的子包和模块的搜索功能。

此功能可以用于扩展包中的模块集，但不常用。

七、输入和输出

1.格式化输出

表达式语句

print()函数

```
>>> 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 is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... 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'))"
>>>
```

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
>>>
```

*#print()*在每列之间加了一个空格，它总是在参数间加入空格

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.ljust(n)[:n]
```

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()的基本使用方法如下：

```
>>> 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 3.14159265359.
>>> 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。下例将 Pi 转为三位精度。

```
>>> import math
>>> 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))
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

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: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
          'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

可以用 '**' 标志将这个字典以关键字参数的方式传入:

```
>>> 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。

(1)旧使的字符串格式化

操作符%也可以用于字符串格式化。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.
```

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。

(1)文件对象方法

首先示例默认文件对象 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，就会读取并返回整个文件。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.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），只有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 first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

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，并且代码简介的方式：

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

把文件中的所有行读到一个列表中，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.write('This is a test\n')
15
```

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话话，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，其默认值为零，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
13
>>> f.read(1)
b'd'
```

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.close() 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 ?
ValueError: I/O operation on 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
```

(2)使用json存储结构化数据

从文件中读写字符串很容易。数值就不是这样，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复杂。

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 可以接受 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。

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 dump()，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)
```