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

(1)把列表当作堆栈使用

(2)把列表当作队列使用

(3)列表推导式

(4)嵌套的列表推导式