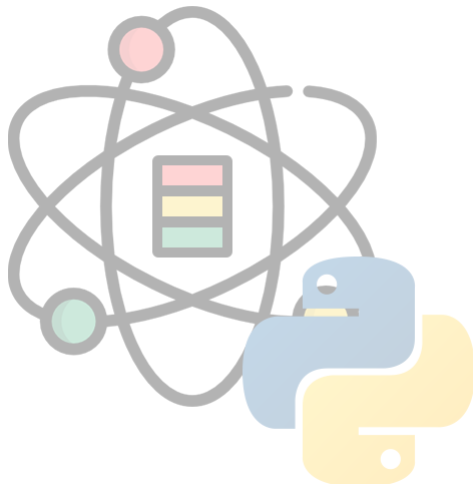
Python 数据科学导论

Data Science Introduction with Python

Python 语言简介

Python Language Introduction

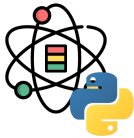
范叶亮



目录

Python 基础语法

Python 语言简介



Python [1] 是由 Guido van Rossum [2] 于 1991 年创造的一种广泛使用的解释型、高级编程、通用型编程语言。

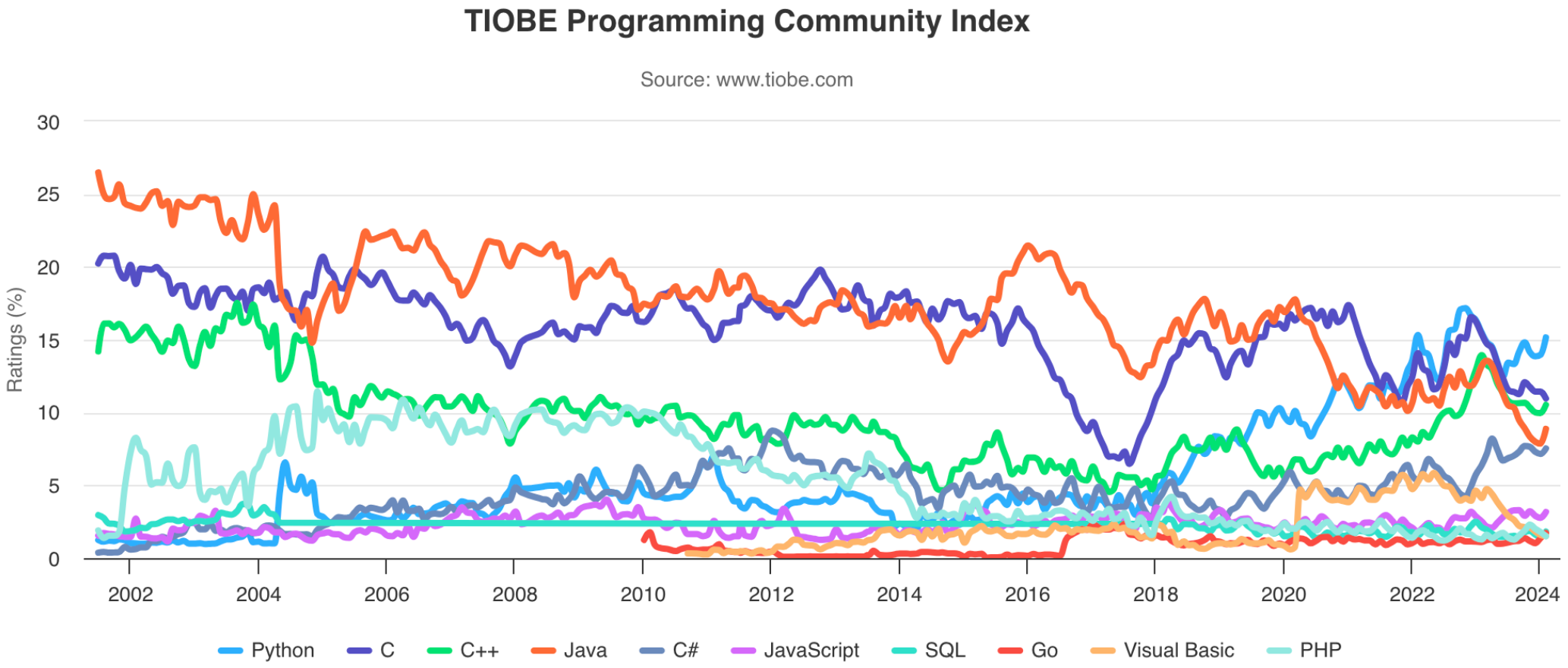
Python 的设计哲学强调代码的可读性和简洁的语法，尤其是使用空格缩进划分代码块，而非使用大括号或者关键词。相比于 C++ 或 Java，Python 让开发者能够用更少的代码表达想法。不管是小型还是大型程序，该语言都试图让程序的结构清晰明了。

Python 拥有动态类型系统和垃圾回收功能，能够自动管理内存使用，并且支持多种编程范式，包括面向对象、命令式、函数式和过程式编程。其本身拥有一个巨大而广泛的标准库。

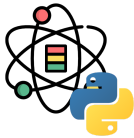
[1] https://zh.wikipedia.org/wiki/Python

[2] https://zh.wikipedia.org/wiki/吉多·范罗苏姆

Python 语言趋势


TIOBE Programming Community Index
Source: www.tiobe.com
Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
2022
2024
Python
C
C++
Java
C#
JavaScript
SQL
Go
Visual Basic
PHP

Python 语法特点



Python 使用缩进 (Tab 或空格) 来组织代码，而不是类似其他语言 (C/C++，Java, R) 使用大括号。使用 Tab 或空格进行缩进没有要求，但应该在一个工程中保持一致。

```
for x in array:
  if x < pivot:
    less.append(x)
  else:
    greater.append(x)
```

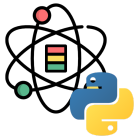
Python 语句并不以分号 ; 结尾，但分号可以用于一行内多条语句之间的分隔：

```
a = 3; b = 4; c = 5
```

所有写在 # 后面的文本将被视为注释并被 Python 解释器忽略：

```
a = 'Leo'
# This is a comment.
```

Python 变量



Python 中通过等号 = 对变量进行赋值，变量类型是动态的，无需提前定义：

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

可以将一个变量赋值给另一个变量：

```
b = a
```

在 Python 中，a 和 b 实际上同时指向了相同的对象，a 发生改变后 b 的值也随之改变：

```
a.append(4)
b
```

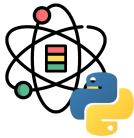
```
## [1, 2, 3, 4]
```

但当赋值的内容为“单值”类型 (int, float, bool, None, str, bytes) 时，则不会出现上述现象。

对象的拷贝分为浅层拷贝和深层拷贝，两个的具体区别请参见官方文档 [1]。

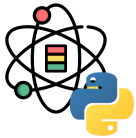
[1] 浅层和深层拷贝：https://docs.python.org/zh-cn/3/library/copy.html

Python 运算



操作符	描述	操作符	描述
a + b	a 加 b	a - b	a 减 b
a * b	a 乘 b	a / b	a 除以 b
a // b	a 整除以 b	a ** b	a 的 b 次方
a & b	a 与 b	a | b	a 或 b
a and b	a 与 b	a or b	a 或 b
a ^ b	a 异或 b	a == b	a 等于 b
a ≠ b	a 不等于 b	a < b, a ≤ b	a 小于(等于) b
a > b, a ≥ b	a 大于(等于) b	a is b	a 和 b 是同一对象
a is not b	a 和 b 不是同一对象		

Python 控制流



Python 中可以通过 `if`，`elif` 和 `else` 控制条件语句，同时在条件中可以使用链式比较：

```
if x < 0:
  print('x < 0')
elif x == 0:
  print('x == 0')
elif 0 < x < 6:
  print('0 < x < 6')
else:
  print('x ≥ 6')
```

Python 中可以通过 `for` 和 `while` 控制循环语句：

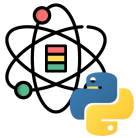
```
for i in range(6):
  for j in range(6):
    print((i, j))

for value in sequence:
  print(value)
```

```
while x < 0:
  x -= 1
```

`for` 语句中还支持拆包：

```
for a, b, c in iterator:
  print((a, b, c))
```

Python 控制流



pass 在 Python 中表示“什么也不做”：

```
if x < 0:
  print('negative!')
elif x == 0:
  pass
else:
  print('positive!')
```

range 返回一个迭代器：

```
range(0, 10)
```

```
## range(0, 10)
```

```
list(range(6, 0, -1))
```

```
## [6, 5, 4, 3, 2, 1]
```

一个 if-else 代码块联合起来为 Python 中的一个三元表达式：

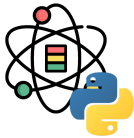
```
x = 6
'x ≥ 0' if x ≥ 0 else 'x < 0'
```

```
## 'x ≥ 0'
```

```
x = -6
'x ≥ 0' if x ≥ 0 else 'x < 0'
```

```
## 'x < 0'
```

Python 函数



Python 通过 def 定义函数：

```
def my_abs(x):
  if x ≥ 0:
    return x
  else:
    return -x

my_abs(-6)
```

函数的参数设置灵活多变，例如：

```
def fun1(x, y, z=1): pass
def fun2(*args): pass
def fun3(**kwargs): pass
```

在调用时，如果不指定参数名称，则通过位置判断对应的参数值，例如：

```
fun(1, 2, 3)
# fun(x=1, y=2, z=3)
```

在指定参数名称时，则可以任意互换参数位置：

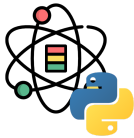
```
fun(z=3, x=1, y=2)
```

* 和 ** 在调用时会发生解包：

```
args = (1, 2)
fun2(*args)
# fun2(1, 2)
```

```
kwargs = {'a': 1, 'b': 2}
fun3(**kwargs)
# fun3(a=1, b=2)
```

Python 导入



在 Python 中，模块即以 .py 为后缀的 Python 代码文件，假设有如下文件：

```
# some_module.py
PI = 3.1415946

def f(x):
  return x + 2

def g(a, b):
  return a + b
```

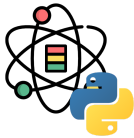
当我们需要在其他文件中使用 some_module.py 中的变量和函数时，我们需要：

```
import some_module
result = some_module.f(5)
```

或是：

```
from some_module import f, g, PI
result = g(5, PI)
```

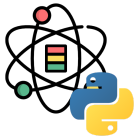
Python 数据类型



Python 标准库中包含了一个内建类型集合，其中“单值”类型也称为标量类型，6 种标量类型如下表所示：

类型	描述
`int`	任意精度无符号整数类型
`float`	双精度 64 位浮点数值类型
`bool`	布尔值类型，`True` 或 `False`
`None`	Python 的 NULL 值 (仅存在一个实例)
`str`	字符串类型，包含 Unicode (UTF-8 编码) 字符串
`bytes`	原生 ASCII 字节 (或 Unicode 编码字节)

Python 数据类型



int 可以存储任意大小数字：

```
var_int = 17239871
var_int ** 3
```

```
## 5123916401649470373311
```

float 表示 64 位双精度的数值，同时可以用科学计数法表示：

```
var_float1 = 3.1415926
var_float2 = 6.18e-1
```

整数除法 / 会自动将整型转为浮点型：

```
3 / 2
```

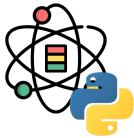
```
## 1.5
```

整除用 // 操作符进行运算

```
3 // 2
```

```
## 1
```

Python 数据类型



Python 中的布尔值包括 True 和 False，布尔值可以与 and 和 or 关键字合用：

```
True and False
```

```
## False
```

```
True or False
```

```
## True
```

None 是 Python 的 NULL 值类型，可以通过 is 判断变量是否为 None：

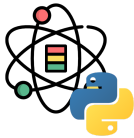
```
a = None
a is None
```

```
## True
```

```
a = 6
a is not None
```

```
## True
```

Python 数据类型



Python 可以用单引号 ' 或双引号 " 表示一个字符串值：

```
var_str1 = 'a string'
var_str2 = "another string"
```

对于多行字符串，可以使用三个单引号 ''' 或三个双引号 """ 表示：

```
var_str3 = '''
This is a string.
This is another string.
'''
```

字符串是 Unicode 字符的序列，因此可以将其视为除了列表和元组以外的另一种序列：

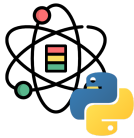
```
s = 'Python'
list(s)
```

```
## ['P', 'y', 't', 'h', 'o', 'n']
```

```
s[:3]
```

```
## 'Pyt'
```

Python 数据类型



反斜杠 \ 为 Python 字符串的转义符号：

```
s = '12\\34'
print(s)
```

```
## 12\34
```

当包含大量转义字符时，为了避免书写麻烦，可以在字符串前加一个前缀符号 r 用于表名后续字符为原生字符：

```
s = r'no\special\characters'
print(s)
```

```
## no\special\characters
```

通过 + 发操作可以将两个字符串合并：

```
a = 'this is '
b = 'a string.'
a + b
```

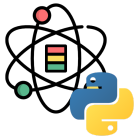
```
## 'this is a string.'
```

字符串是不可变的：

```
s = 'this is a string.'
s[10] = 'f'
```

```
# TypeError: 'str' object does ↩
  not support item assignment
```

Python 数据类型



字符串的格式化方法有多种，字符串对象具有使用 % 运算符的内置操作：

```
name = 'Leo'
'Hello, %s.' % name
```

```
## 'Hello, Leo.'
```

```
age = 30
'Hello, %s. You are %s.' % \
  (name, age)
```

```
## 'Hello, Leo. You are 30.'
```

str.format() 是对 % 的改进，替换字段用大括号标记：

```
'Hello, {}. You are {}.' \
  .format(name, age)
```

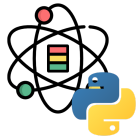
```
## 'Hello, Leo. You are 30.'
```

通过引用索引来以任何顺序引用变量：

```
'Hello, {0}. You are {1}, {0}.' \
  .format(age, name)
```

```
## 'Hello, 30. You are Leo, 30.'
```

Python 数据类型



通过插入变量名称进行引用：

```
person = {'name': 'Leo', 'age': 30}
'Hello, {name}. You are {age}.' \
  .format(name=person['name'],
          age=person['age'])
```

```
## 'Hello, Leo. You are 30.'
```

使用 ** 直接对字典进行拆包：

```
'Hello, {name}. You are {age}.' \
  .format(**person)
```

```
## 'Hello, Leo. You are 30.'
```

更多格式化请参考：https://docs.python.org/zh-cn/3/library/string.html

f-String 自 Python 3.6 开始加入标准库，开头包含一个 f，其包含表达式的大括号将被对应的值替换。

```
name, age = 'Leo', 30
f'Hello, {name}. You are {age}.'
```

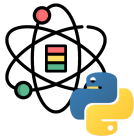
```
## 'Hello, Leo. You are 30.'
```

f-String 是运行时渲染的，因此可以将有效的 Python 表达式放入其中。

```
f'You are {3*10}, {name.lower()}.'
```

```
## 'You are 30, leo.'
```

Python 数据类型



Python 3.0 及以上，Unicode 成为字符串类型的一等类，用于更好的兼容处理 ASCII 和非 ASCII 文本。

```
eva = '初号机'
eva
```

```
## '初号机'
```

```
eva_utf8 = eva.encode('utf-8')
eva_utf8
```

```
## b'\xe5\x88\x9d\xe5\x8f\xb7\xe6\x9c\xba'
```

```
type(eva_utf8)
```

```
## <class 'bytes'>
```

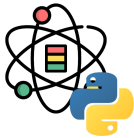
```
eva_utf8.decode('utf-8')
```

```
## '初号机'
```

通过在字符串前加前缀 b 可以直接定义 bytes，但请尽量避免这么做。

Python 数据结构

Python 元组



元组 是一种固定长度，不可变的 Python 对象序列。

```
tup = 1, 2, 3
tup
```

```
## (1, 2, 3)
```

`tuple` 函数可以将任意序列或迭代器转换为元素：

```
tuple([1, 2, 3])
```

```
## (1, 2, 3)
```

```
tuple('string')
```

```
## ('s', 't', 'r', 'i', 'n', 'g')
```

元组可以通过中括号 `[]` 来获取元素，Python 中序列的索引从 0 开始：

```
tup[0]
```

```
## 1
```

可以通过 `+` 连接元组，将元组乘以整数可以生产包含多份拷贝的元组：

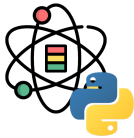
```
(1, 2) + (3, )
```

```
## (1, 2, 3)
```

```
(1, 2, 3) * 2
```

```
## (1, 2, 3, 1, 2, 3)
```

Python 元组



将元组型的表达式赋值给变量，Python 会对等号右边的值进行拆包：

```
tup = (1, 2, 3)
a, b, c = tup
b
```

```
## 2
```

嵌套元组也可以拆包：

```
tup = (4, 5, (6, 7))
a, b, (c, d) = tup
c
```

```
## 6
```

高级拆包功能用于从元组的起始位置“采集”一些元素：

```
values = 1, 2, 3, 4, 5, 6
a, b, *rest = values
a, b
```

```
## (1, 2)
```

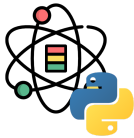
```
rest
```

```
## [3, 4, 5, 6]
```

为了方便，很多 Python 用户会使用下划线 _ 来表示不需要的变量。

```
a, b, *_ = values
```

Python 列表



与元组不同，列表的长度是可变的。它所包含的内容也是可以修改的，可以使用中括号 [] 或者 list 类型函数来定义列表：

```
a = [1, 2, 3, None]
tup = ('Leo', 'Van')
b = list(tup)
b
```

```
## ['Leo', 'Van']
```

```
b[0] = 'Leonardo'
b
```

```
## ['Leonardo', 'Van']
```

list 函数在数据处理中常用于将迭代器或生成器转换为列表：

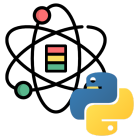
```
gen = range(10)
gen
```

```
## range(0, 10)
```

```
list(gen)
```

```
## [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Python 列表



使用 append 方法可以将元素添加到尾部：

```
a = ['Leo', 'Van']
a.append('Black')
a
```

```
## ['Leo', 'Van', 'Black']
```

使用 insert 方法可以将元素插入到指定位置：

```
a.insert(2, 'Mr.')
a
```

```
## ['Leo', 'Van', 'Mr.', 'Black']
```

使用 pop 方法可以将指定位置的元素移除并返回：

```
a.pop(2)
```

```
## 'Mr.'
```

使用 remove 方法可以移除第一个符合要求的值：

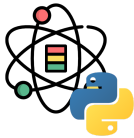
```
a.append('Leo')
a
```

```
## ['Leo', 'Van', 'Black', 'Leo']
```

```
a.remove('Leo')
a
```

```
## ['Van', 'Black', 'Leo']
```

Python 列表



in 用于检查一个值是否在列表中：

```
'Leo' in a
```

```
## True
```

not 关键字可以用作 in 的反义词，表示不在：

```
'Leo' not in a
```

```
## False
```

两个列表可以使用 + 号进行连接：

```
[1, 2] + [3, 4]
```

```
## [1, 2, 3, 4]
```

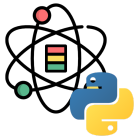
对于一个已经定义的列表，可以使用 extend 方法向该列表中添加多个元素：

```
x = ['Leo', 'Van']
x.extend(['Mr.', 'Black'])
x
```

```
## ['Leo', 'Van', 'Mr.', 'Black']
```

+ 在连接过程中创建了新的列表，同时还需要复制对象，因此使用 extend 效率更高。

Python 列表



调用 sort 方法对列表进行排序（无需新建对象）：

```
a = [1, 3, 2, 6, 4, 5]
a.sort()
a
```

```
## [1, 2, 3, 4, 5, 6]
```

通过参数 key 可以指定用于生成排序值的函数：

```
b = ['likes', 'hiking', 'He']
b.sort(key=len)
b
```

```
## ['He', 'likes', 'hiking']
```

内建的 bisect 模块实现了二分搜索和已排序列表的插值。bisect.bisect 会找到元素应当被插入的位置，并保持序列排序，bisect.insort 将元素插入到相应的位置。

```
import bisect
c = [1, 2, 2, 3, 4, 7]
bisect.bisect(c, 2)
```

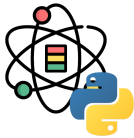
```
## 3
```

```
bisect.insort(c, 6)
c
```

```
## [1, 2, 2, 3, 4, 6, 7]
```

bisect 模块的函数不会检查列表是否已经排序。

Python 列表



使用切片符号可以对大多数序列选取子集，基本形式是将 start:stop 传入到索引符号 [] 中：

```
seq = [1, 2, 3, 4, 5, 6]
seq[1:3]
```

```
## [2, 3]
```

切片还可以将序列赋值给变量：

```
seq[3:4] = [40, None]
seq
```

```
## [1, 2, 3, 40, None, 5, 6]
```

在切片中，起始位置 start 是包含的，而结束位置 stop 是不包含的。

start 和 stop 是可以省略的，如果省略则会默认传入序例的起始位置或结束位置：

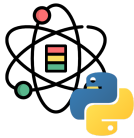
```
seq[:3]
```

```
## [1, 2, 3]
```

```
seq[3:]
```

```
## [40, None, 5, 6]
```

Python 列表



负索引可以从序列的尾部进行索引：

```
seq = [1, 2, 3, 4, 5, 6]
seq[-3:]
```

```
## [4, 5, 6]
```

```
seq[-4:-2]
```

```
## [3, 4]
```

步进值 step 可以在第二个冒号后使用，表示每隔多少个数取一个值：

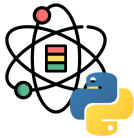
```
seq[::2]
```

```
## [1, 3, 5]
```

```
seq[::-1]
```

```
## [6, 5, 4, 3, 2, 1]
```

Python 列表



enumerate 用于在遍历序列时同时追踪当前元素的索引，其返回 (i, v)，其中 i 为元素索引，v 为元素值：

```
a = ['a', 'b', 'c']
b = []

for i, v in enumerate(a):
  b.append((i, v))

b
```

```
## [(0, 'a'), (1, 'b'), (2, 'c')]
```

sorted 函数返回一个根据任意序列中的元素新建的已排序的列表：

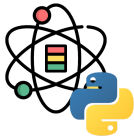
```
sorted([1, 3, 4, 2, 6, 5])
```

```
## [1, 2, 3, 4, 5, 6]
```

```
sorted('Leo Van')
```

```
## [' ', 'L', 'V', 'a', 'e', 'n', 'o']
```

Python 列表



zip 可以将列表、元组或其他序列的元素配对，新建一个元组构成的列表：

```
seq1 = [1, 2, 3]
seq2 = ['a', 'b', 'c']
zipped = zip(seq1, seq2)
list(zipped)
```

```
## [(1, 'a'), (2, 'b'), (3, 'c')]
```

zip 可以处理任意长度的序列，它生成列表的长度由最短的序列决定。

```
seq3 = [True, False]
list(zip(seq1, seq2, seq3))
```

```
## [(1, 'a', True), (2, 'b', False)]
```

对于一个已经“配对”的序列，zip 可以对其进行“拆分”，即将“行”的列表转换为“列”的列表：

```
names = [('Leo', 'Van'), ('Mr.', 'Black')]
first_names, last_names = zip(*names)
first_names
```

```
## ('Leo', 'Mr.')
```

```
last_names
```

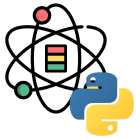
```
## ('Van', 'Black')
```

reversed 函数将序列的元素倒叙排列：

```
list(reversed(range(6)))
```

```
## [5, 4, 3, 2, 1, 0]
```

Python 字典



字典 dict 是 Python 内建的哈希表类型的数据结构，是一种拥有有灵活吃春的键值对集合。在 Python 中使用大括号 {} 可以创建字典，并用逗号将键值对分割，其中键和值均为 Python 对象。

```
d = {'a': 'Leo', 'b': [1, 2, 3]}
d
```

```
## {'a': 'Leo', 'b': [1, 2, 3]}
```

类似访问列表和元组中的元素一样，可以访问、插入或设置字典中的元素：

```
d[3] = 6
d
```

```
## {'a': 'Leo', 'b': [1, 2, 3], 3: 6}
```

```
d['b']
```

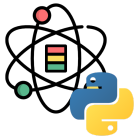
```
## [1, 2, 3]
```

类似列表和元组，可以字典中知否包含一个键值：

```
'b' in d
```

```
## True
```

Python 字典



可以使用 del 或 pop 产出字典的值，pop 方法会在删除的同时返回被删除的值：

```
d
```

```
## {'a': 'Leo', 'b': [1, 2, 3], 3: 6}
```

```
del d[3]
d
```

```
## {'a': 'Leo', 'b': [1, 2, 3]}
```

```
l = d.pop('b')
l
```

```
## [1, 2, 3]
```

```
d
```

```
## {'a': 'Leo'}
```

keys 和 values 方法分别会提供字典的键和值的迭代器，键值没有特定的顺序，但两个函数输出的键和值均会按照相同的顺序：

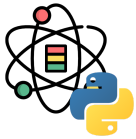
```
d = {'a': 'Leo', 'b': [1, 2, 3], 3: 6}
list(d.keys())
```

```
## ['a', 'b', 3]
```

```
list(d.values())
```

```
## ['Leo', [1, 2, 3], 6]
```

Python 字典



可以使用 update 函数将两个词典合并：

```
d1 = {'a': 'Leo', 'b': [1, 2, 3]}
d2 = {3: 6}
d1.update(d2)
d1
```

```
## {'a': 'Leo', 'b': [1, 2, 3], 3: 6}
```

也可以用 ** 将两个词典合并：

```
d1 = {'a': 'Leo', 'b': [1, 2, 3]}
d2 = {3: 6}
d3 = {**d1, **d2}
d3
```

```
## {'a': 'Leo', 'b': [1, 2, 3], 3: 6}
```

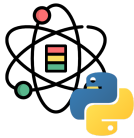
字典的 get 和 pop 方法可以对于一个不存在的键值返回一个默认值：

```
v = d.get(key, default_value)
```

内建的 defaultdict 可以想字典中传入类型或能在各个位置生成默认值的函数：

```
from collections import defaultdict

by_letter = defaultdict(list)
words = ['apple', 'bat', 'bar', 'atom', 'book']

for word in words:
  by_letter[word[0]].append(word)
```

Python 字典



字典的值可以是任何 Python 对象，但必须为不可变对象，例如：标量，元组等。一个对象是否能够作为键值可以通过 hash 函数来检测一个对象是否可以哈希化。

```
hash('string')
```

```
## 8003165216489225753
```

```
hash((1, 2, (3, 4)))
```

```
## 3794340727080330424
```

```
hash((1, 2, [3, 4]))
# TypeError: unhashable type: 'list'
```

对于两个序列，当我们希望将其中一个作为键，另外一个作为值进行配对生成字典，原始的方法为：

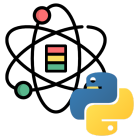
```
mapping = {}
for k, v in zip(k_list, v_list):
  mapping[k] = v
```

字典本质上也是 2 个元素的元组的集合，字典是可以接受一个 2 元组的列表作为参数，上述代码可以改写为：

```
m = dict(zip(range(6), reversed(range(6))))
m
```

```
## {0: 5, 1: 4, 2: 3, 3: 2, 4: 1, 5: 0}
```

Python 集合



集合是一种无序且元素唯一的容器，集合可以通过两种方式创建：通过 set 函数创建或用字面值集和大括号语法创建：

```
set([2, 2, 2, 1, 3, 3])
```

```
## {1, 2, 3}
```

```
{2, 2, 2, 1, 3, 3}
```

```
## {1, 2, 3}
```

集合支持数学上的集合操作，如并集，交集，差集等：

```
a = {1, 2, 3, 4, 5}
b = {3, 4, 5, 6, 7, 8}
```

两个集合的并集可以通过 union 方法或 | 操作符完成：

```
a.union(b)
```

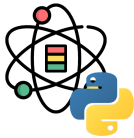
```
## {1, 2, 3, 4, 5, 6, 7, 8}
```

```
a | b
```

```
## {1, 2, 3, 4, 5, 6, 7, 8}
```

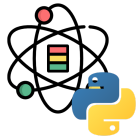
常用的集合方法列表如下所示：

Python 集合



函数	替代方法	描述	函数	替代方法	描述
a.add(x)	N/A	添加元素 x	a.clear()	N/A	清空所有元素
a.remove(x)	N/A	移除元素 x	a.pop()	N/A	移除任意位置元素
a.union(b)	a | b	a 和 b 的并集	a.update(b)	a |= b	将 a 置为 a 和 b 的并集
a.intersection(b)	a & b	a 和 b 的交集	a.intersection_update(b)	a &= b	将 a 置为 a 和 b 的交集
a.difference(b)	a - b	a 和 b 的差集	a.difference_update(b)	a -= b	将 a 置为 a 和 b 的差集
a.issubset(b)	N/A	a 是否为 b 的子集	a.issuperset(b)	N/A	a 是否为 b 的超集
a.isdisjoint(b)	N/A	a 和 b 是否无交集	a.symmetric_difference(b)	a ^ b	a 和 b 的对称差集
			a.symmetric_difference_update(b)	a ^= b	将 a 置为 a 和 b 的对称差集

Python 列表推导



列表推导允许过滤一个容器的元素，其用一个简明的表达式转换传递给过滤器的元素，从而生成一个新的列表。列表推导式的基础形式为：

```
[expr for value in collection if condition]
```

这与下面的 for 循环是等价的：

```
result = []
for val in collection:
  if condition:
    result.append(expr)
```

过滤条件是可以省略的。

一个示例如下：

```
strings = ['Leo', 'Van', 'Data', 'Science']
[s.upper() for s in strings if len(x) > 3]
```

```
## ['LEO', 'VAN', 'DATA', 'SCIENCE']
```

字典的推导式如下所示：

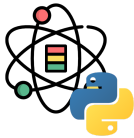
```
{k-expr:v-expr for value in collection if condition}
```

集合的推导式如下所示：

```
{expr for value in collection if condition}
```

Python 编码风格规范

Python 编码风格规范



> 不要在行尾加分号, 也不要用分号将两条命令放在同一行。

> 每行不超过 80 个字符。

Python 会将圆括号, 中括号和花括号中的行隐式的连接起来，可以利用这个特点在表达式外围增加一对额外的圆括号。

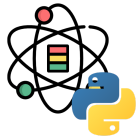
```
if (width == 0 and height == 0 and
    color == 'red' and emphasis == 'strong'):
```

在注释中，如果必要，将长的 URL 放在一行上。

```
# http://www.example.com/us/developer/documentation/api/content/v2.0/do_something.html
```

```
# http://www.example.com/us/developer/documentation/api/content/\
# v2.0/do_something.html
```

Python 编码风格规范



> 宁缺毋滥的使用括号。

除非是用于实现行连接，否则不要在返回语句或条件语句中使用括号。 不过在元组两边使用括号是可以的。

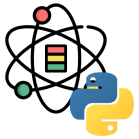
```
if foo:
    bar()
while x:
    x = bar()
if x and y:
    bar()
if not x:
    bar()
return foo
for (x, y) in dict.items(): ...
```

```
if (x):
    bar()
if not(x):
    bar()
return (foo)
```

> 顶级定义之间空两行，方法定义之间空一行。

顶级定义之间空两行，比如函数或者类定义。 方法定义。类定义与第一个方法之间都应该空一行。 函数或方法中，某些地方要是你觉得合适，就空一行。

Python 编码风格规范



> 用 4 个空格来缩进代码。

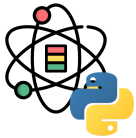
绝对不要用 tab，也不要 tab 和空格混用。对于行连接的情况，你应该要么垂直对齐换行的元素，或者使用 4 空格的悬挂式缩进（这时第一行不应该有参数）。

✅
```
# Aligned with opening delimiter
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
# Aligned with opening delimiter in a dictionary
foo = {
    long_dictionary_key: value1 +
                         value2,
    ...
}
# 4-space hanging indent; nothing on first line
foo = long_function_name(
    var_one, var_two, var_three,
    var_four)
```

✅
```
# 4-space hanging indent in a dictionary
foo = {
    long_dictionary_key:
        long_dictionary_value,
    ...
}
```

❗
```
# Stuff on first line forbidden
foo = long_function_name(var_one, var_two,
    var_three, var_four)
# 2-space hanging indent forbidden
foo = long_function_name(
  var_one, var_two, var_three,
  var_four)
# No hanging indent in a dictionary
foo = {
    long_dictionary_key:
        long_dictionary_value,
        ...
}
```

Python 编码风格规范



> 按照标准的排版规范来使用标点两边的空格。

括号内不要有空格。

✅
```
spam(ham[1], {eggs: 2}, [])
```

❗
```
spam( ham[ 1 ], { eggs: 2 }, [ ] )
```

不要在逗号，分号，冒号前面加空格，但应该在它们后面加（除了在行尾）。

✅
```
if x == 4:
    print x, y
x, y = y, x
```

❗
```
if x == 4 :
    print x , y
x , y = y , x
```

参数列表，索引或切片的左括号前不应加空格。

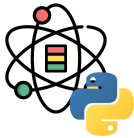
✅
```
spam(1)
```

❗
```
spam (1)
```

✅
```
dict['key'] = list[index]
```

❗
```
dict ['key'] = list [index]
```

Python 编码风格规范



在二元操作符两边都加上一个空格，比如赋值 =，比较 ==，<，>，≠，<>，≤，≥，in，not in，is，is not，布尔 and，or，not。 至于算术操作符两边的空格该如何使用，需要你自己好好判断，不过两侧务必要保持一致。

```
x == 1
```

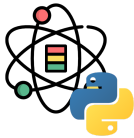
```
x<1
```

当 = 用于指示关键字参数或默认参数值时，不要在其两侧使用空格。

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

```
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

Python 编码风格规范



不要用空格来垂直对齐多行间的标记，因为这会成为维护的负担（适用于 `:`，`#`，`=` 等）。

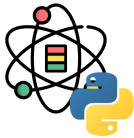
```python
foo = 1000  # comment
long_name = 2  # comment should not be aligned

dictionary = {
    "foo": 1,
    "long_name": 2,
    }
```

```python
foo       = 1000  # comment
long_name = 2     # comment should not be aligne
d

dictionary = {
    "foo"      : 1,
    "long_name": 2,
    }
```

> 大部分 `.py` 文件不必以 `#!` 作为文件的开始。根据 PEP-394，程序的 `main` 文件应该以 `#!/usr/bin/python2` 或者 `#!/usr/bin/python3` 开始。

在计算机科学中，Shebang（也称为 Hashbang）是一个由井号和叹号构成的字符串行 `#!`，其出现在文本文件的第一行的前两个字符。在文件中存在 Shebang 的情况下，类 Unix 操作系统的程序载入器会分析 Shebang 后的内容，将这些内容作为解释器指令，并调用该指令，并将载有 Shebang 的文件路径作为该解释器的参数。例如，以指令 `#!/bin/sh` 开头的文件在执行时会实际调用 `/bin/sh` 程序。

Python 编码风格规范



> 确保对模块，函数，方法和行内注释使用正确的风格。

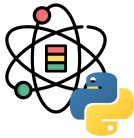
文档字符串

Python 有一种独一无二的的注释方式：使用文档字符串。文档字符串是包，模块，类或函数里的第一个语句。这些字符串可以通过对象的 `__doc__` 成员被自动提取，并且被 pydoc 所用。我们对文档字符串的惯例是使用三重双引号 """ (PEP-257)。一个文档字符串应该这样组织：首先是一行以句号，问号或惊叹号结尾的概述（或者该文档字符串单纯只有一行）。接着是一个空行。接着是文档字符串剩下的部分，它应该与文档字符串的第一行的第一个引号对齐。下面有更多文档字符串的格式化规范。

模块

每个文件应该包含一个许可样板。根据项目使用的许可（例如，Apache 2.0，BSD，LGPL，GPL），选择合适的样板。

Python 编码风格规范



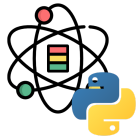
函数和方法

下文所指的函数，包括函数，方法，以及生成器。一个函数必须要有文档字符串，除非它满足以下条件：1. 外部不可见，2. 非常短小，3. 简单明了。

文档字符串应该包含函数做什么，以及输入和输出的详细描述。通常，不应该描述“怎么做”，除非是一些复杂的算法。文档字符串应该提供足够的信息，当别人编写代码调用该函数时，他不需要看一行代码，只要看文档字符串就可以了。对于复杂的代码，在代码旁边加注释会比使用文档字符串更有意义。关于函数的几个方面应该在特定的小节中进行描述记录， 这几个方面如下文所述。每节应该以一个标题行开始。标题行以冒号结尾。除标题行外，节的其他内容应被缩进 2 个空格。

Args：列出每个参数的名字，并在名字后使用一个冒号和一个空格，分隔对该参数的描述。如果描述太长超过了单行 80 字符，使用 2 或者 4 个空格的悬挂缩进（与文件其他部分保持一致）。描述应该包括所需的类型和含义。如果一个函数接受 `*foo`（可变长度参数列表）或者 `**bar` （任意关键字参数），应该详细列出 `*foo` 和 `**bar`。

Python 编码风格规范

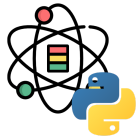


Returns (或者 Yields 用于生成器)：描述返回值的类型和语义。如果函数返回 None，这一部分可以省略。

Raises：列出与接口有关的所有异常。

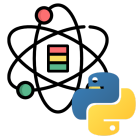
```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by big_table.  Silly things may happen if
    other_silly_variable is not None.

    Args:
        big_table: An open Bigtable Table instance.
        keys: A sequence of strings representing the key of each table row
            to fetch.
        other_silly_variable: Another optional variable, that has a much
            longer name than the other args, and which does nothing.
    """
```

Python 编码风格规范



```
    """
    Returns:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    Raises:
        IOError: An error occurred accessing the bigtable.Table object.
    """
    pass
```

Python 编码风格规范



块注释和行注释

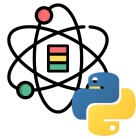
最需要写注释的是代码中那些技巧性的部分。 如果你在下次 代码审查 的时候必须解释一下，那么你应该现在就给它写注释。 对于复杂的操作，应该在其操作开始前写上若干行注释。 对于不是一目了然的代码，应在其行尾添加注释。

```
# We use a weighted dictionary search to find out where i is in
# the array.  We extrapolate position based on the largest num
# in the array and the array size and then do binary search to
# get the exact number.

if i & (i-1) == 0:        # True if i is 0 or a power of 2.
```

为了提高可读性，注释应该至少离开代码 2 个空格。另一方面，绝不要描述代码. 假设阅读代码的人比你更懂 Python，他只是不知道你的代码要做什么。

```
# Now go through the b array and make sure whenever i occurs
# the next element is i+1
```

Python 编码风格规范



> 即使参数都是字符串，使用 % 或者格式化方法格式化字符串。不过也不能一概而论，你需要在 + 和 % 之间好好判定。

✅
```
x = a + b
x = '%s, %s!' % (imperative, expletive)
x = '{}, {}!'.format(imperative, expletive)
x = 'name: %s; score: %d' % (name, n)
x = 'name: {}; score: {}'.format(name, n)
```

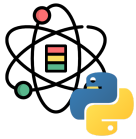
❗
```
x = '%s%s' % (a, b)  # use + in this case
x = '{}{}'.format(a, b)  # use + in this case
x = imperative + ', ' + expletive + '!'
x = 'name: ' + name + '; score: ' + str(n)
```

避免在循环中用 + 和 += 操作符来累加字符串。由于字符串是不可变的，这样做会创建不必要的临时对象，并且导致二次方而不是线性的运行时间。作为替代方案，你可以将每个子串加入列表，然后在循环结束后用 .join 连接列表。

✅
```
items = ['<table>']
for l, f in employee_list:
  items.append('<tr><td>%s,%s</td></tr>' % (l, f
)
items.append('</table>')
table = ''.join(items)
```

❗
```
table = '<table>'
for l, f in employee_list:
  table += '<tr><td>%s,%s</td></tr>' % (l, f)
table += '</table>'
```

Python 编码风格规范



在同一个文件中，保持使用字符串引号的一致性。使用单引号 ' 或者双引号 " 之一用以引用字符串，并在同一文件中沿用。在字符串内可以使用另外一种引号，以避免在字符串中使用。PyLint 已经加入了这一检查。

✅
```
Python('Why are you hiding your eyes?')
Gollum("I'm scared of lint errors.")
Narrator('"Good!" thought a Python reviewer.')
```

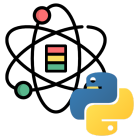
❗
```
Python("Why are you hiding your eyes?")
Gollum('The lint. It burns. It burns us.')
Gollum("Always the great lint. Watching.")
```

为多行字符串使用三重双引号 """ 而非三重单引号 '''。当且仅当项目中使用单引号 ' 来引用字符串时，才可能会使用三重 ''' 为非文档字符串的多行字符串来标识引用。 文档字符串必须使用三重双引号 """。不过要注意，通常用隐式行连接更清晰，因为多行字符串与程序其他部分的缩进方式不一致。

✅
```
print ("This is much nicer.\n"
       "Do it this way.\n")
```

❗
```
    print """This is pretty ugly.
Don't do this.
"""
```

Python 编码风格规范



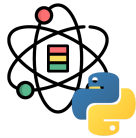
> 为临时代码使用 TODO 注释，它是一种短期解决方案，不算完美，但够好了。

TODO 注释应该在所有开头处包含 TODO 字符串，紧跟着是用括号括起来的你的名字，email 地址或其它标识符。然后是一个可选的冒号。接着必须有一行注释，解释要做什么。 主要目的是为了有一个统一的 TODO 格式，这样添加注释的人就可以搜索到(并可以按需提供更多细节)。写了 TODO 注释并不保证写的人会亲自解决问题。当你写了一个 TODO，请注上你的名字。

```
# TODO(kl@gmail.com): Use a "*" here for string repetition.
# TODO(Zeke) Change this to use relations.
```

如果你的 TODO 是“将来做某事”的形式，那么请确保你包含了一个指定的日期（“2009 年 11 月解决”）或者一个特定的事件（“等到所有的客户都可以处理XML请求就移除这些代码”）。

Python 编码风格规范



> 每个导入应该独占一行。

✅
```
import os
import sys
```

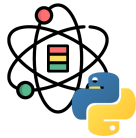
❗
```
import os, sys
```

导入总应该放在文件顶部，位于模块注释和文档字符串之后，模块全局变量和常量之前。 导入应该按照从最通用到最不通用的顺序分组：1. 标准库导入，2. 第三方库导入，3. 应用程序指定导入。

每种分组中，应该根据每个模块的完整包路径按字典序排序，忽略大小写。

```
import foo
from foo import bar
from foo.bar import baz
from foo.bar import Quux
from Foob import ar
```

Python 编码风格规范



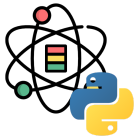
> 通常每个语句应该独占一行。

不过，如果测试结果与测试语句在一行放得下，你也可以将它们放在同一行。如果是 `if` 语句，只有在没有 `else` 时才能这样做。特别地，绝不要对 `try/except` 这样做，因为 `try` 和 `except` 不能放在同一行。

```python
if foo: bar(foo)
```

```python
if foo: bar(foo)
else:   baz(foo)

try:                bar(foo)
except ValueError: baz(foo)

try:
    bar(foo)
except ValueError: baz(foo)
```

Python 编码风格规范

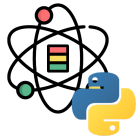


> 在 Python 中，对于琐碎又不太重要的访问函数，你应该直接使用公有变量来取代它们，这样可以避免额外的函数调用开销。当添加更多功能时，你可以用属性（property）来保持语法的一致性。

重视封装的面向对象程序员看到这个可能会很反感，因为他们一直被教育：所有成员变量都必须是私有的！其实，那真的是有点麻烦啊，试着去接受 Pythonic 哲学吧。

另一方面，如果访问更复杂，或者变量的访问开销很显著，那么你应该使用像 `get_foo()` 和 `set_foo()` 这样的函数调用。如果之前的代码行为允许通过属性（property）访问，那么就不要将新的访问函数与属性绑定。这样，任何试图通过老方法访问变量的代码就没法运行，使用者也就会意识到复杂性发生了变化。

Python 编码风格规范



> `module_name`, `package_name`, `ClassName`, `method_name`, `ExceptionName`, `function_name`, `GLOBAL_VAR_NAME`, `instance_var_name`, `function_parameter_name`, `local_var_name`.

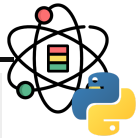
应该避免的名称

1. 单字符名称，除了计数器和迭代器。
2. 包/模块名中的连字符 `-`。
3. 双下划线开头并结尾的名称（Python 保留，例如 `__init__`）。

命名约定

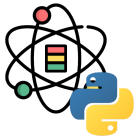
1. 所谓“内部（Internal）”表示仅模块内可用，或者在类内是保护或私有的。
2. 用单下划线 `_` 开头表示模块变量或函数是 protected 的（使用 `from module import *` 时不会包含）。
3. 用双下划线 `__` 开头的实例变量或方法表示类内私有。
4. 将相关的类和顶级函数放在同一个模块里。不像 Java，没必要限制一个类一个模块。
5. 对类名使用大写字母开头的单词（如 CapWords，即 Pascal 风格），但是模块名应该用小写加下划线的方式（如 `lower_with_under.py`）。

Python 之父 Guido 推荐的规范



类型	公共	内部
模块	lower_with_under	_lower_with_under
包	lower_with_under	
类	CapWords	_CapWords
异常	CapWords	
函数	lower_with_under()	_lower_with_under()
全局/类常量	CAPS_WITH_UNDER	_CAPS_WITH_UNDER
全局/类变量	lower_with_under	_lower_with_under
实例变量	lower_with_under	_lower_with_under（保护） __lower_with_under（私有）
方法名称	lower_with_under()	_lower_with_under()（保护） __lower_with_under()（私有）
函数/方法参数	lower_with_under	
局部变量	lower_with_under	

Python 编码风格规范



> 即使是一个打算被用作脚本的文件，也应该是可导入的。并且简单的导入不应该导致这个脚本的主功能被执行，这是一种副作用，主功能应该放在一个 `main()` 函数中。

在 Python 中，pydoc 以及单元测试要求模块必须是可导入的。你的代码应该在执行主程序前总是检查 `if __name__ == '__main__'`，这样当模块被导入时主程序就不会被执行。

```
def main():
      ...

if __name__ == '__main__':
    main()
```

所有的顶级代码在模块导入时都会被执行。要小心不要去调用函数，创建对象，或者执行那些不应该在使用 pydoc 时执行的操作。

感谢倾听