

Python介绍、安装及本语法

Python 越来越火爆

Python 在诞生之初，因为其功能不好，运转功率低，不支持多核，根本没有并发性可言，在计算功能不那么好的年代，一直没有火爆起来，甚至很多人根本不知道有这门语言。

随着时代的发展，物理硬件功能不断提高，而软件的复杂性也不断增大，开发效率越来越被企业重视。因此就有了不一样的声音，在软件开发的初始阶段，性能并没有开发效率重要，没必然为了节省不到 1ms 的时间却让开发量增加好几倍，这样划不过来。也就是开发效率比机器效率更为重要，那么 Python 就逐渐得到越来越多开发者的青睐了。

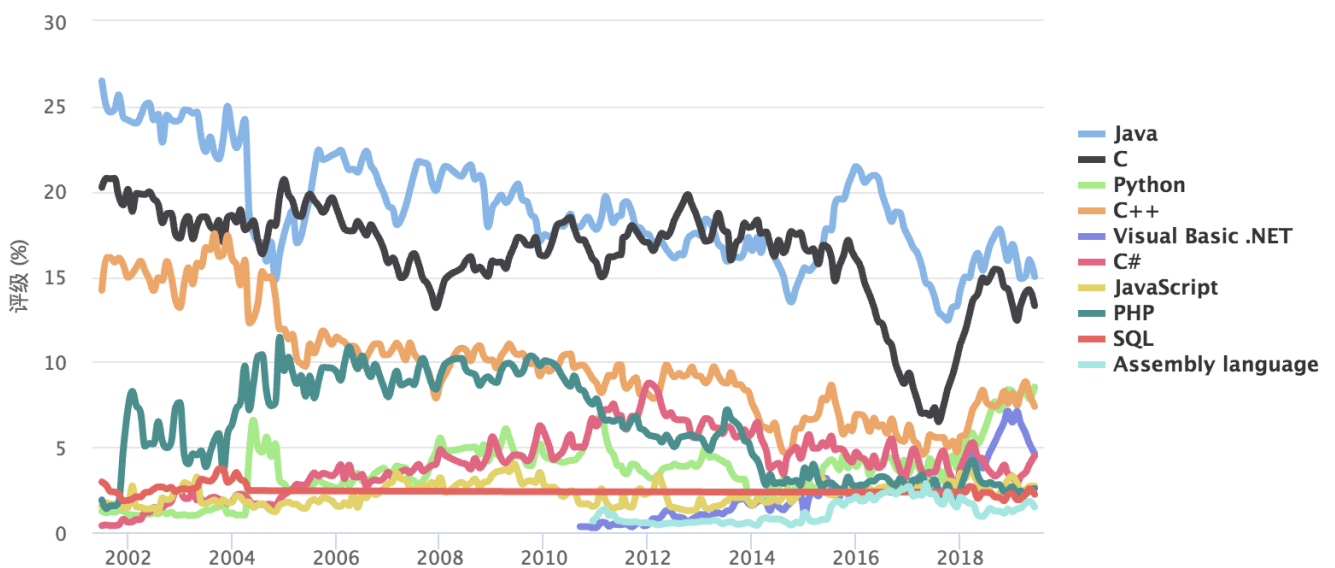
在 12-14 年，云计算升温，大量创业公司和互联网巨头挤进云计算领域，而最著名的云核算开源渠道 OpenStack 就是基于 Python 开发的。

随后几年的备受关注的的人工智能，机器学习首选开发语言也是 Python。

至此，Python 已经成为互联网开发的焦点。在「Top 10 的编程语言走势图」可以看到，Python 已经跃居第三位，而且在 2017 年还成为了最受欢迎的语言。

TOP 10 编程语言的走势图

Source: www.tiobe.com



Python 容易入门且功能强大

如果你是一名初学者，学习 Python 就是你最好的选择，因为它容易学，功能强大，很容易就能构建 Web 应用，非常适合初学者作为入门的开发语言。

Python 的安装

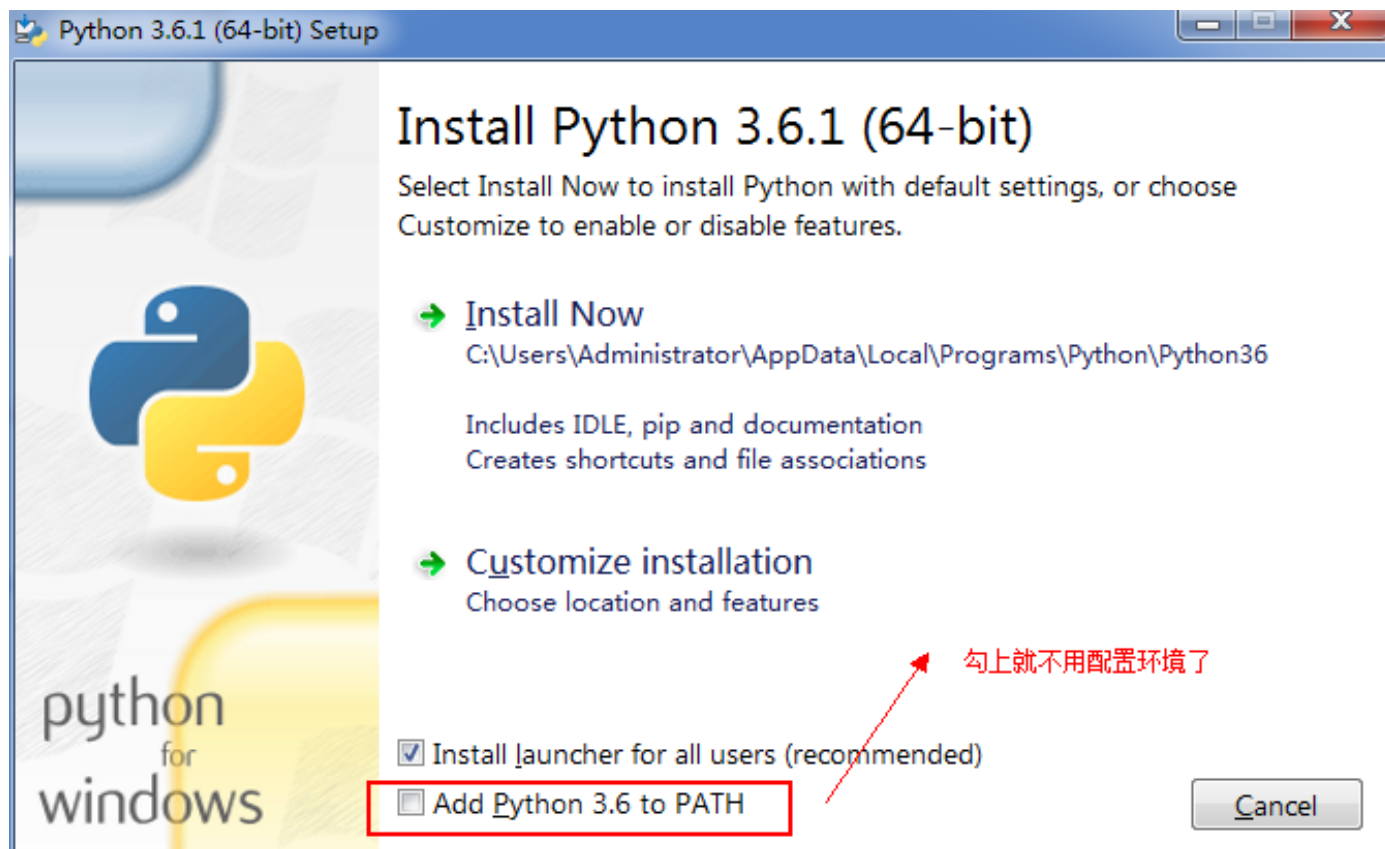
因为 Python 是跨平台的，它可以运行在 Windows、Mac 和各种 Linux/Unix 系统上。目前，Python 有两个版本，一个是 2.x 版，一个是 3.x 版，这两个版本是不兼容的。本草根安装的是 3.6.1 版本的。

至于在哪里下载，草根我建议大家最好直接官网下载，随时下载下来的都是最新版本。官网地址：<https://www.python.org/>

1、windows 系统下安装配置

如果是 windows 系统，下载完后，直接安装，不过这里记得勾上Add Python 3.6 to PATH，然后点「Install Now」 即可完成安装。

这里要注意了，记得把「Add Python 3.6 to Path」 勾上，勾上之后就不需要自己配置环境变量了，如果没勾上，就要自己手动配置。



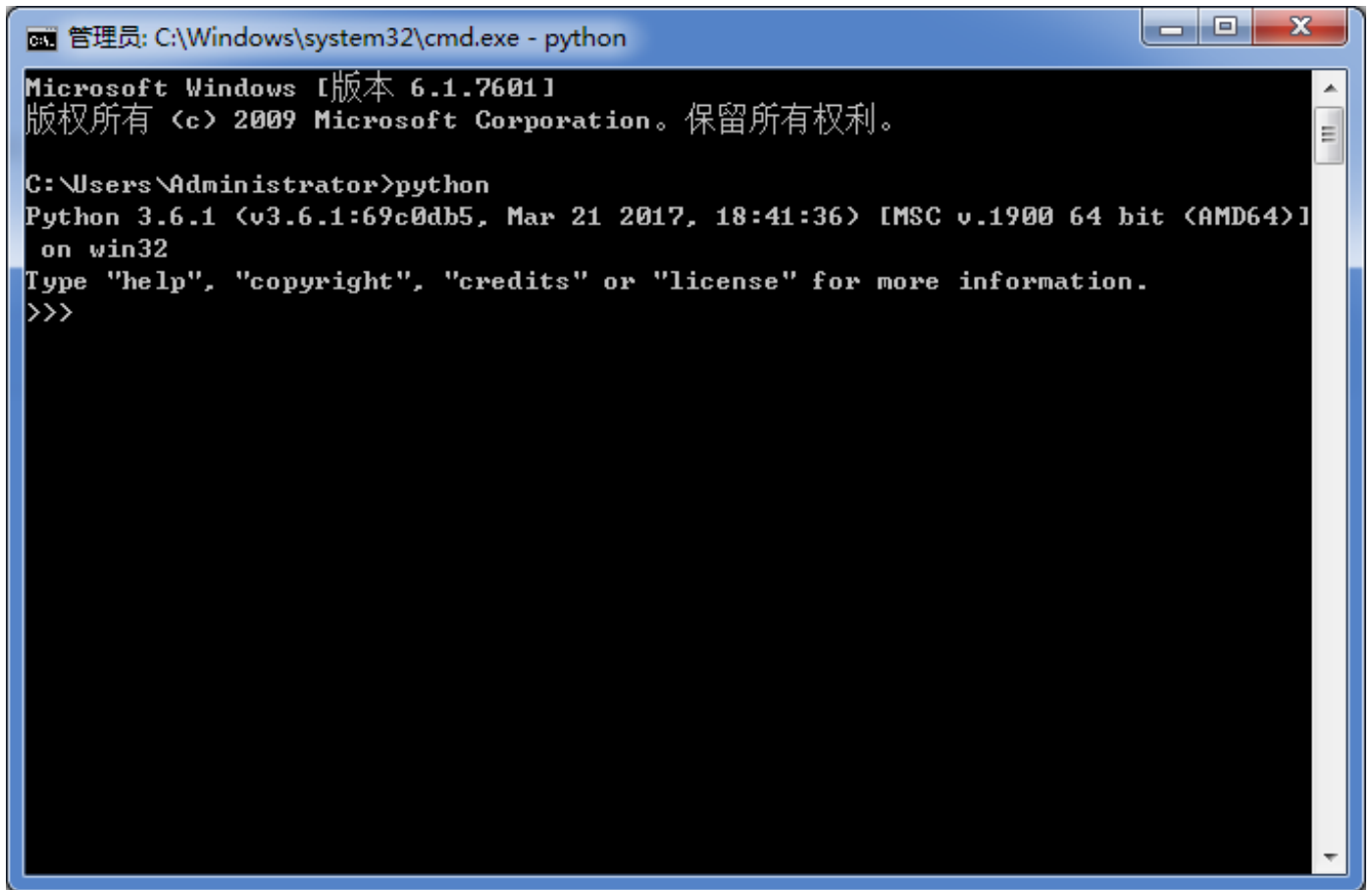
如果你一时手快，忘记了勾上「Add Python 3.6 to Path」，那也不要紧，只需要手动配置一下环境变量就好了。

在命令提示框中 cmd 上输入：

```
path=%path%;C:\Python
```

特别特别注意：`C:\Python` 是 Python 的安装目录，如果你的安装目录是其他地方，就得填上你对应的目录。

安装完成后，打开命令提示符窗口，敲入 python 后，出现下面的情况，证明 Python 安装成功了。



```
管理员: C:\Windows\system32\cmd.exe - python
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>python
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

而你看到提示符 `>>>` 就表示我们已经在 Python 交互式环境中了，可以输入任何 Python 代码，回车后会立刻得到执行结果。

2、Mac 系统下安装配置

MAC 系统一般都自带有 Python2.x 版本的环境，不过现在都不用 2.x 的版本了，所以建议你在 <https://www.python.org/downloads/mac-osx/> 上下载最新版安装。

安装完成之后，如何配置环境变量呢？

先查看当前环境变量：

```
echo $PATH
```

然后打开 `~/.bash_profile`(没有请新建)

```
vi ~/.bash_profile
```

我装的是 Python3.7，Python 执行路径为：`/Library/Frameworks/Python. Framework/Versions/3.7/bin`。于是写入

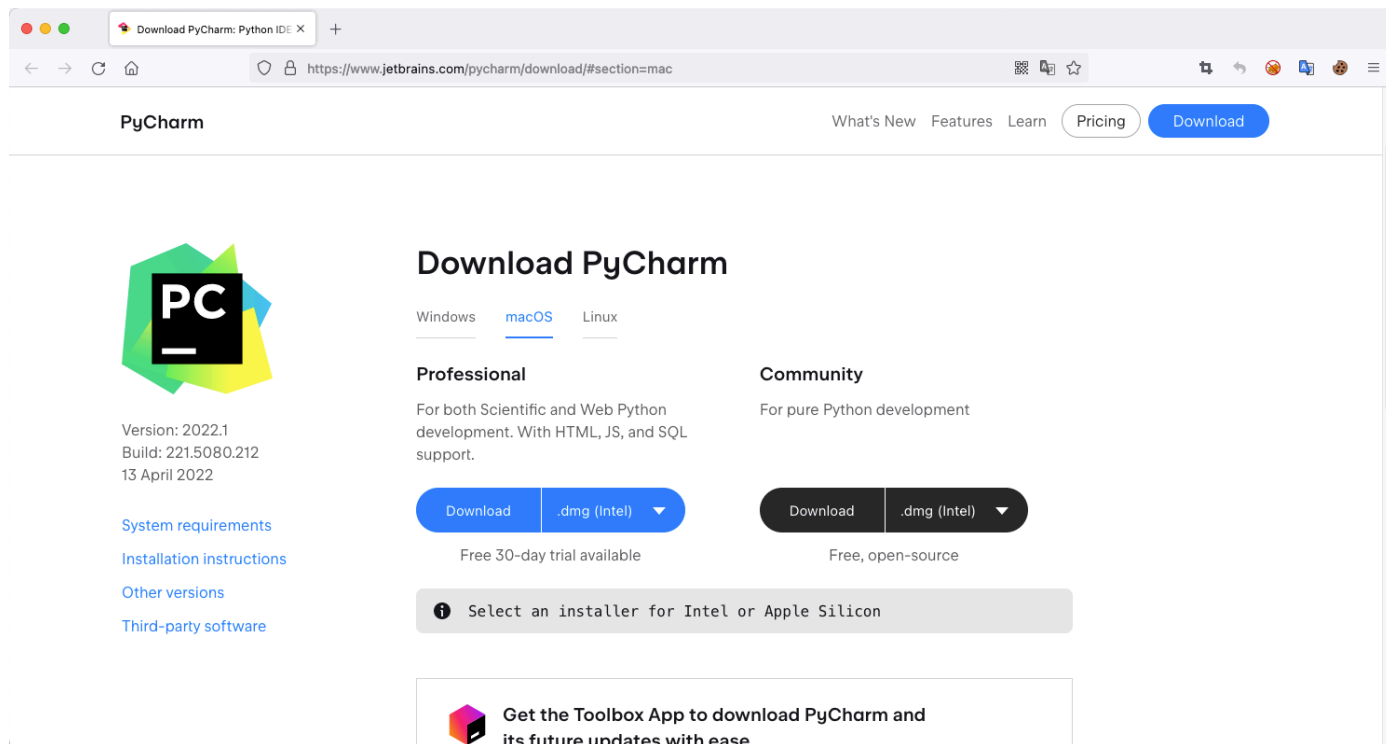
```
export PATH="/Library/Frameworks/Python. Framework/Versions/3.7/bin:$PATH"
```

最后保存退出，激活运行一下文件：

```
source ~/.bash_profile
```

集成开发环境（IDE）：PyCharm

PyCharm 下载地址：<https://www.jetbrains.com/pycharm/download/>



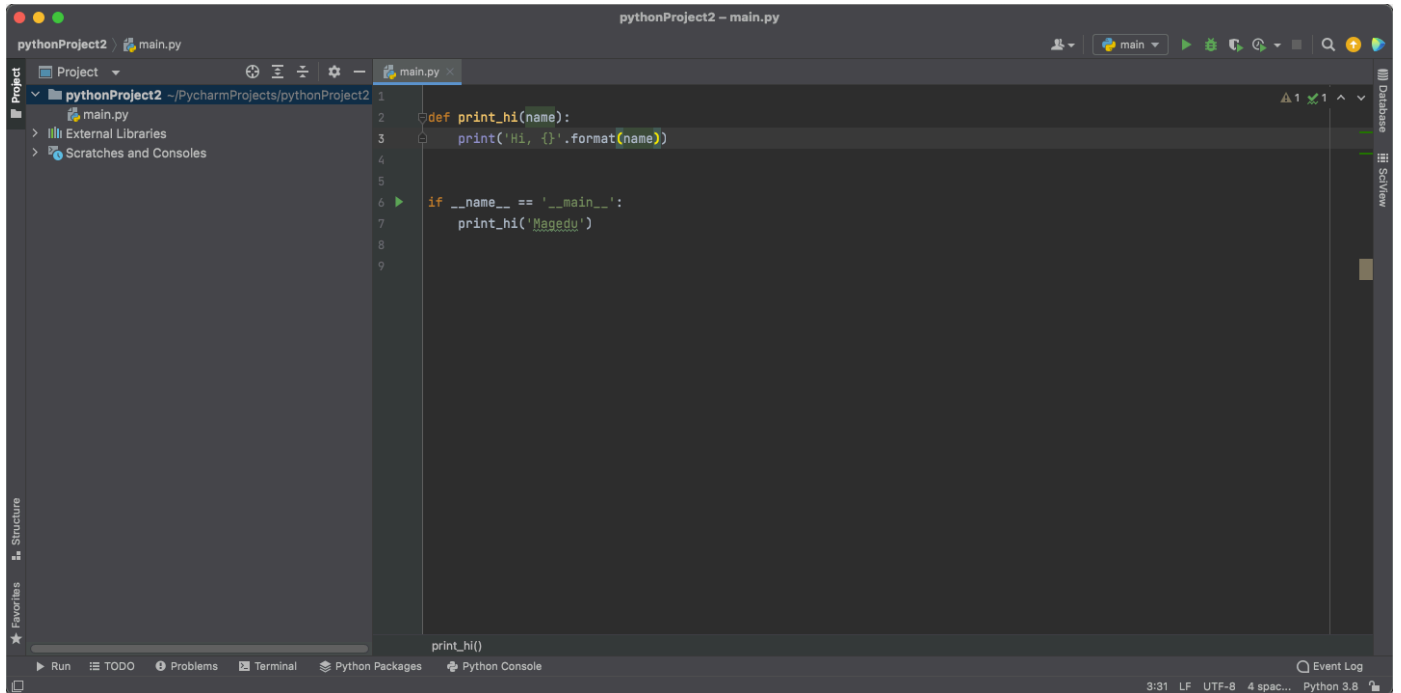
第一个 Python 程序

现在我们可以来写一下第一个 Python 程序了。

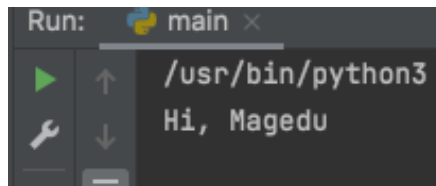
第一个 Python 程序当然是打印 Hello Magedu 。

新建一个文件，命名为 `main.py`，注意，这里是以 `.py` 为后缀的文件。

然后打开文件，输入 `print('Hello {}'.format(name))`



运行结果



代码文件

```
def print_hi(name):  
    print('Hi, {}'.format(name))  
  
if __name__ == '__main__':  
    print_hi('Magedu')
```

###

语法规范

1、编码

源文件的字符编码

默认情况下，Python 源码文件的编码是 UTF-8。这种编码支持世界上大多数语言的字符，可以用于字符串字面值、变量、函数名及注释—— 尽管标准库只用常规的 ASCII 字符作为变量名或函数名，可移植代码都应遵守此约定。要正确显示这些字符，编辑器必须能识别 UTF-8 编码，而且必须使用支持文件中所有字符的字体。

如果不使用默认编码，则要声明文件的编码，文件的 第一行要写成特殊注释。句法如下：

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

其中，*encoding* 可以是 Python 支持的任意一种编码。

可查看官方文档 [codecs](#)

比如，声明使用 Windows-1252 编码，源码文件要写成：

```
# -*- coding: cp1252 -*-
```

第一行的规则也有一种例外情况，源码以 [UNIX "shebang" 行](#) 开头。此时，编码声明要写在文件的第二行。例如：

```
#!/usr/bin/env python3  
# -*- coding: cp1252 -*-
```

备注：

- Unix 系统中，为了不与同时安装的 Python 2.x 冲突，Python 3.x 解释器默认安装的执行文件名不是 `python`。

2、代码格式

2.1、缩进

- 统一使用 4 个空格进行缩进

2.2、行宽

每行代码尽量不超过 80 个字符(在特殊情况下可以略微超过 80，但最长不得超过 120)

理由：

- 方便在控制台下查看代码
- 太长可能是设计有缺陷

2.3、引号

简单说，自然语言使用双引号，机器标示使用单引号，因此 代码里 多数应该使用 单引号

- 自然语言使用双引号 `"..."`
例如错误信息；很多情况还是 unicode，使用 `u"你好世界"`
- 机器标识使用单引号 `'...'` 例如 dict 里的 key
- 正则表达式使用原生的双引号 `r"..."`
- 文档字符串 (docstring) 使用三个双引号 `"""....."""`

2.4、空行

- 模块级函数和类定义之间空两行；
- 类成员函数之间空一行；

```
class A:

    def __init__(self):
        pass

    def hello(self):
        pass

def main():
    pass
```

- 可以使用多个空行分隔多组相关的函数
- 函数中可以使用空行分隔出逻辑相关的代码

3、import 语句

- import 语句应该分行书写

```
# 正确的写法
import os
import sys

# 不推荐的写法
import sys,os

# 正确的写法
from subprocess import Popen, PIPE
```

- import语句应该使用 **absolute** import

正确的写法

```
from foo.bar import Bar
```

不推荐的写法

```
from ..bar import Bar
```

- import语句应该放在文件头部，置于模块说明及docstring之后，于全局变量之前；
- import语句应该按照顺序排列，每组之间用一个空行分隔

```
import os
import sys

import msgpack
import zmq

import foo
```

- 导入其他模块的类定义时，可以使用相对导入

```
from myclass import MyClass
```

- 如果发生命名冲突，则可使用命名空间

```
import bar
import foo.bar

bar.Bar()
foo.bar.Bar()
```

4、空格

- 在二元运算符两边各空一格 `[=, -, +=, ==, >, in, is not, and]`:

正确的写法

```
i = i + 1
submitted += 1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

不推荐的写法

```
i=i+1
submitted +=1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```


- 函数的参数列表中, , 之后要有空格

正确的写法

```
def complex(real, imag):  
    pass
```

不推荐的写法

```
def complex(real,imag):  
    pass
```

- 函数的参数列表中, 默认值等号两边不要添加空格

正确的写法

```
def complex(real, imag=0.0):  
    pass
```

不推荐的写法

```
def complex(real, imag = 0.0):  
    pass
```

- 左括号之后, 右括号之前不要加多余的空格

正确的写法

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

不推荐的写法

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

- 字典对象的左括号之前不要多余的空格

正确的写法

```
dict['key'] = list[index]
```

不推荐的写法

```
dict ['key'] = list [index]
```

- 不要为对齐赋值语句而使用的额外空格

正确的写法

```
x = 1  
y = 2  
long_variable = 3
```

不推荐的写法

```
x           = 1  
y           = 2  
long_variable = 3
```

5、换行

Python 支持括号内的换行。这时有两种情况。

1. 第二行缩进到括号的起始处

```
foo = long_function_name(var_one, var_two,
                          var_three, var_four)
```

1. 第二行缩进 4 个空格，适用于起始括号就换行的情形

```
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

使用反斜杠 `\` 换行，二元运算符 `+` `.` 等应出现在行末；长字符串也可以用此法换行

```
session.query(MyTable).\
    filter_by(id=1).\
    one()

print 'Hello, '\
    '%s %s!' %\
    ('Harry', 'Potter')
```

禁止复合语句，即一行中包含多个语句：

```
# 正确的写法
do_first()
do_second()
do_third()

# 不推荐的写法
do_first();do_second();do_third();
```

`if/for/while` 一定要换行：

```
# 正确的写法
if foo == 'blah':
    do_blah_thing()

# 不推荐的写法
if foo == 'blah': do_blash_thing()
```

6、docstring

docstring 的规范中最其本的两点：

1. 所有的公共模块、函数、类、方法，都应该写 docstring 。私有方法不一定需要，但应该在 def 后提供一个块注释来说明。
2. docstring 的结束"""应该独占一行，除非此 docstring 只有一行。

```
"""Return a foobar
Optional plotz says to frobnicate the bizbaz first.
"""

"""Online docstring"""
```

注释

1、注释

1.1、块注释

“#”号后空一格，段落件用空行分开（同样需要“#”号）

```
# 块注释
# 块注释
#
# 块注释
# 块注释
```

1.2、行注释

至少使用两个空格和语句分开，注意不要使用无意义的注释

```
# 正确的写法
x = x + 1 # 边框加粗一个像素

# 不推荐的写法(无意义的注释)
x = x + 1 # x加1
```

1.3、建议

- 在代码的关键部分(或比较复杂的地方), 能写注释的要尽量写注释
- 比较重要的注释段, 使用多个等号隔开, 可以更加醒目, 突出重要性

```

app = create_app(name, options)

# =====
# 请勿在此处添加 get post等app路由行为 !!!
# =====

if __name__ == '__main__':
    app.run()

```

2、文档注释 (Docstring)

作为文档的Docstring一般出现在模块头部、函数和类的头部，这样在python中可以通过对象的**doc**对象获取文档。编辑器和IDE也可以根据Docstring给出自动提示。

- 文档注释以 `"""` 开头和结尾, 首行不换行, 如有多行, 末行必需换行, 以下是Google的docstring风格示例

```

# -*- coding: utf-8 -*-
"""Example docstrings.

This module demonstrates documentation as specified by the `Google Python
Style Guide`_. Docstrings may extend over multiple lines. Sections are created
with a section header and a colon followed by a block of indented text.

Example:
    Examples can be given using either the ``Example`` or ``Examples``
    sections. Sections support any reStructuredText formatting, including
    literal blocks::

        $ python example_google.py

Section breaks are created by resuming unindented text. Section breaks
are also implicitly created anytime a new section starts.
"""

```

- 不要在文档注释复制函数定义原型, 而是具体描述其具体内容, 解释具体参数和返回值等

```

# 不推荐的写法 (不要写函数原型等废话)
def function(a, b):
    """function(a, b) -> list"""
    ...

# 正确的写法
def function(a, b):
    """计算并返回a到b范围内数据的平均值"""
    ...

```

- 对函数参数、返回值等的说明采用numpy标准, 如下所示

```
def func(arg1, arg2):  
    """在这里写函数的一句话总结(如: 计算平均值).  
  
    这里是具体描述.  
  
    参数  
    -----  
    arg1 : int  
        arg1的具体描述  
    arg2 : int  
        arg2的具体描述  
  
    返回值  
    -----  
    int  
        返回值的具体描述  
  
    参看  
    -----  
    otherfunc : 其它关联函数等...  
  
    示例  
    -----  
    示例使用doctest格式, 在`>>>`后的代码可以被文档测试工具作为测试用例自动运行  
  
    >>> a=[1,2,3]  
    >>> print [x + 3 for x in a]  
    [4, 5, 6]  
    """
```

- 文档注释不限于中英文, 但不要中英文混用
- 文档注释不是越长越好, 通常一两句话能把情况说清楚即可
- 模块、公有类、公有方法, 能写文档注释的, 应该尽量写文档注释

命名规范

1、模块

- 模块尽量使用小写命名, 首字母保持小写, 尽量不要用下划线(除非多个单词, 且数量不多的情况)

```
# 正确的模块名  
import decoder  
import html_parser  
  
# 不推荐的模块名  
import Decoder
```

2、类名

- 类名使用驼峰(CamelCase)命名风格, 首字母大写, 私有类可用一个下划线开头

```
class Farm():
    pass

class AnimalFarm(Farm):
    pass

class _PrivateFarm(Farm):
    pass
```

- 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.

3、函数

- 函数名一律小写, 如有多个单词, 用下划线隔开

```
def run():
    pass

def run_with_env():
    pass
```

- 私有函数在函数前加一个下划线_

```
class Person():

    def _private_func():
        pass
```

4、变量名

- 变量名尽量小写, 如有多个单词, 用下划线隔开

```
if __name__ == '__main__':
    count = 0
    school_name = ''
```

- 常量采用全大写, 如有多个单词, 使用下划线隔开

```
MAX_CLIENT = 100
MAX_CONNECTION = 1000
CONNECTION_TIMEOUT = 600
```

5、常量

- 常量使用以下划线分隔的大写命名

```
MAX_OVERFLOW = 100

Class FooBar:

    def foo_bar(self, print_):
        print(print_)
```

Python字符串

字符串字面量

python 中的字符串字面量由单引号或双引号括起。

'hello Magedu' 等于 "hello Magedu"。

我们可以使用 print() 函数显示字符串字面量：

实例

```
print("Hello Magedu")
print('Hello Magedu')
```

用字符串向变量赋值

通过使用变量名称后跟等号和字符串，可以把字符串赋值给变量：

实例

```
a = "Hello Magedu"
print(a)
```

多行字符串

您可以使用三个引号将多行字符串赋值给变量：

实例

您可以使用三个双引号：

```
mage = """马哥教育隶属于北京马哥教育科技有限公司，是一家国内IT培训品牌，主营学科为Linux云计算、linux运维，Python开发及人工智能，数据分析，大数据等。提供线上训练营，技术研讨会、技术培训课、网络公开课以及免费教学视频服务。"""
print(mage)
```

或三个单引号：

实例

```
mage = '''P马哥教育隶属于北京马哥教育科技有限公司，是一家国内IT培训品牌，主营学科为Linux云计算、linux运维，Python开发及人工智能，数据分析，大数据等。提供线上训练营，技术研讨会、技术培训课、网络公开课以及免费教学视频服务。'''
print(mage)
```

字符串是数组

像许多其他流行的编程语言一样，Python 中的字符串是表示 unicode 字符的字节数组。

但是，Python 没有字符数据类型，单个字符就是长度为 1 的字符串。

方括号可用于访问字符串的元素。

实例

获取位置 1 处的字符（请记住第一个字符的位置为 0）：

```
mage = "Hello, World!"
print(a[1])
```

裁切

您可以使用裁切语法返回一定范围的字符。

指定开始索引和结束索引，以冒号分隔，以返回字符串的一部分。

实例

获取从位置 3 到位置 5（不包括）的字符：

```
b = "Hello, World!"
print(b[3:5])
```

负的索引

使用负索引从字符串末尾开始切片：

实例

获取从位置 5 到位置 3 的字符，从字符串末尾开始计数：

```
b = "Hello, World!"  
print(b[-5:-3])
```

字符串长度

如需获取字符串的长度，请使用 len() 函数。

实例

len() 函数返回字符串的长度：

```
a = "Hello, Magedu!"  
print(len(a))
```

字符串方法

Python 有一组可用于字符串的内置方法。

实例

strip() 方法删除开头和结尾的空白字符：

```
a = " Hello, Magedu! "  
print(a.strip()) # returns "Hello, Magedu!"
```

实例

lower() 返回小写的字符串：

```
a = "Hello, Magedu!"  
print(a.lower())
```

实例

upper() 方法返回大写的字符串：

```
a = "Hello, Magedu!"  
print(a.upper())
```

实例

replace() 用另一段字符串来替换字符串：

```
a = "Hello, Magedu!"  
print(a.replace("World", "Kitty"))
```

实例

split() 方法在找到分隔符的实例时将字符串拆分为子字符串：

```
a = "Hello, Magedu!"  
print(a.split(",")) # returns ['Hello', ' Magedu!']
```

请使用我们的字符串方法参考手册，学习更多的字符串方法。

检查字符串

如需检查字符串中是否存在特定短语或字符，我们可以使用 in 或 not in 关键字。

实例

检查以下文本中是否存在短语 "edu"：

```
txt = "Magedu is a great company"  
x = "edu" in txt  
print(x)
```

实例

检查以下文本中是否没有短语 "edu"：

```
txt = "Magedu is a great company"  
x = "edu" not in txt  
print(x)
```

字符串级联（串联）

如需串联或组合两个字符串，您可以使用 + 运算符。

实例

将变量 a 与变量 b 合并到变量 c 中：

```
a = "Hello"
b = "Magedu"
c = a + b
print(c)
```

实例

在它们之间添加一个空格：

```
a = "Hello"
b = "Magedu"
c = a + " " + b
print(c)
```

字符串格式

正如在 Python 变量一章中所学到的，我们不能像这样组合字符串和数字：

实例

```
age = 100
txt = "My name is Mage, I am " + age
print(txt)
```

但是我们可以使用 format() 方法组合字符串和数字！

format() 方法接受传递的参数，格式化它们，并将它们放在占位符 {} 所在的字符串中：

实例

使用 format() 方法将数字插入字符串：

```
age = 100
txt = "My name is Magedu, and I am {} years old"
print(txt.format(age))
```

format() 方法接受不限数量的参数，并放在各自的占位符中：

实例

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

可以使用索引号 {0} 来确保参数被放在正确的占位符中：

实例

```
quantity = 300
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

Python 用作计算器

现在，尝试一些简单的 Python 命令。启动解释器，等待主提示符（>>>）出现。

数字

解释器像一个简单的计算器：输入表达式，就会给出答案。表达式的语法很直接：运算符 +、-、*、/ 的用法和其他大部分语言一样（比如，Pascal 或 C）；括号 (()) 用来分组。例如：

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5 # division always returns a floating point number
1.6
```

整数（如，2、4、20）的类型是 `int`，带小数（如，5.0、1.6）的类型是 `float`。本教程后半部分将介绍更多数字类型。

除法运算（/）返回浮点数。用 // 运算符执行 [floor division](#) 的结果是整数（忽略小数）；计算余数用 %：

```
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3 # floor division discards the fractional part
5
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # floored quotient * divisor + remainder
17
```

Python 用 `**` 运算符计算乘方 [1](#):

```
>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128
```

等号 (`=`) 用于给变量赋值。赋值后，下一个交互提示符的位置不显示任何结果:

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

如果变量未定义（即，未赋值），使用该变量会提示错误:

```
>>> n # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Python 全面支持浮点数；混合类型运算数的运算会把整数转换为浮点数:

```
>>> 4 * 3.75 - 1
14.0
```

交互模式下，上次输出的表达式会赋给变量 `_`。把 Python 当作计算器时，用该变量实现下一步计算更简单，例如:

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

最好把该变量当作只读类型。不要为它显式赋值，否则会创建一个同名独立局部变量，该变量会用它的魔法行为屏蔽内置变量。

除了 `int` 和 `float`，Python 还支持其他数字类型，例如 `Decimal` 或 `Fraction`。Python 还内置支持 [复数](#)，后缀 `j` 或 `J` 用于表示虚数（例如 `3+5j`）。

字符串

除了数字，Python 还可以操作字符串。字符串有多种表现形式，用单引号（`'.....'`）或双引号（`"....."`）标注的结果相同 [2](#)。反斜杠 `\` 用于转义：

```
>>> '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," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

交互式解释器会为输出的字符串加注引号，特殊字符使用反斜杠转义。虽然，有时输出的字符串看起来与输入的字符串不一样（外加的引号可能会改变），但两个字符串是相同的。如果字符串中有单引号而没有双引号，该字符串外将加注双引号，反之，则加注单引号。`print()` 函数输出的内容更简洁易读，它会省略两边的引号，并输出转义后的特殊字符：

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

如果不希望前置 `\` 的字符转义成特殊字符，可以使用 *原始字符串*，在引号前添加 `r` 即可：

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

字符串字面值可以包含多行。一种实现方式是使用三重引号：`"""..."""` 或 `'''...'''`。字符串中将自动包括行结束符，但也可以在换行的地方添加一个 `\` 来避免此情况。参见以下示例：

```
print("""\
Usage: thingy [OPTIONS]
    -h                 Display this usage message
    -H hostname        Hostname to connect to
""")
```

输出如下（请注意开始的换行符没有被包括在内）：

```
Usage: thingy [OPTIONS]
    -h                 Display this usage message
    -H hostname        Hostname to connect to
```

字符串可以用 `+` 合并（粘到一起），也可以用 `*` 重复：

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

相邻的两个或多个 *字符串字面值*（引号标注的字符）会自动合并：

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

拆分长字符串时，这个功能特别实用：

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

这项功能只能用于两个字面值，不能用于变量或表达式：

```
>>> prefix = 'Py'
>>> prefix 'thon' # can't concatenate a variable and a string literal
File "<stdin>", line 1
    prefix 'thon'
          ^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
File "<stdin>", line 1
    ('un' * 3) 'ium'
              ^
SyntaxError: invalid syntax
```

合并多个变量，或合并变量与字面值，要用 `+`：

```
>>> prefix + 'thon'
'Python'
```

字符串支持 索引/（下标访问），第一个字符的索引是 0。单字符没有专用的类型，就是长度为一的字符串：

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

索引还支持负数，用负数索引时，从右边开始计数：

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

注意，`-0` 和 `0` 一样，因此，负数索引从 `-1` 开始。

除了索引，字符串还支持 切片。索引可以提取单个字符，切片则提取子字符串：


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

切片索引的默认值很有用；省略开始索引时，默认值为 0，省略结束索引时，默认为到字符串的结尾：

```
>>> 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'
```

注意，输出结果包含切片开始，但不包含切片结束。因此，`s[:i] + s[i:]` 总是等于 `s`：

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

还可以这样理解切片，索引指向的是字符 *之间*，第一个字符的左侧标为 0，最后一个字符的右侧标为 n ， n 是字符串长度。例如：

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

第一行数字是字符串中索引 0...6 的位置，第二行数字是对应的负数索引位置。 i 到 j 的切片由 i 和 j 之间所有对应的字符组成。

对于使用非负索引的切片，如果两个索引都不越界，切片长度就是起止索引之差。例如，`word[1:3]` 的长度是 2。

索引越界会报错：

```
>>> 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
```

但是，切片会自动处理越界索引：

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

Python 字符串不能修改，是 [immutable](#) 的。因此，为字符串中某个索引位置赋值会报错：

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

要生成不同的字符串，应新建一个字符串：

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

内置函数 [len\(\)](#) 返回字符串的长度：

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

列表

Python 支持多种 复合数据类型，可将不同值组合在一起。最常用的 *列表*，是用方括号标注，逗号分隔的一组值。*列表* 可以包含不同类型的元素，但一般情况下，各个元素的类型相同：

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

和字符串（及其他内置 [sequence](#) 类型）一样，列表也支持索引和切片：

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

切片操作返回包含请求元素的新列表。以下切片操作会返回列表的 [浅拷贝](#)：

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

列表还支持合并操作：

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

与 [immutable](#) 字符串不同, 列表是 [mutable](#) 类型, 其内容可以改变：

```
>>> 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]
```

`append()` 方法 可以在列表结尾添加新元素（详见后文）：

```
>>> 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]
```

为切片赋值可以改变列表大小，甚至清空整个列表：

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

内置函数 `len()` 也支持列表：

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

还可以嵌套列表（创建包含其他列表的列表），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'
```

走向编程的第一步

当然，Python 还可以完成比二加二更复杂的任务。例如，可以编写 [斐波那契数列](#) (这个数列从第3项开始，每一项都等于前两项之和)的初始子序列，如下所示：

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
5
8
```

本例引入了几个新功能。

- 第一行中的 **多重赋值**：变量 `a` 和 `b` 同时获得新值 0 和 1。最后一行又用了一次多重赋值，这体现在右表达式在赋值前就已经求值了。右表达式求值顺序为从左到右。
- `while` 循环只要条件（这里指：`a < 10`）保持为真就会一直执行。Python 和 C 一样，任何非零整数都为真，零为假。这个条件也可以是字符串或列表的值，事实上，任何序列都可以；长度非零就为真，空序列则为假。示例中的判断只是最简单的比较。比较操作符的标准写法和 C 语言一样：`<`（小于）、`>`（大于）、`==`（等于）、`<=`（小于等于）、`>=`（大于等于）及 `!=`（不等于）。

- 循环体是缩进的：缩进是 Python 组织语句的方式。在交互式命令行里，得为每个缩输入制表符或空格。使用文本编辑器可以实现更复杂的输入方式；所有像样的文本编辑器都支持自动缩进。交互式输入复合语句时，要在最后输入空白行表示结束（因为解析器不知道哪一行代码是最后一行）。注意，同一块语句的每一行的缩进相同。
- `print()` 函数输出给定参数的值。与表达式不同（比如，之前计算器的例子），它能处理多个参数，包括浮点数与字符串。它输出的字符串不带引号，且各参数项之间会插入一个空格，这样可以实现更好的格式化操作：

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

关键字参数 `end` 可以取消输出后面的换行, 或用另一个字符串结尾：

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

备注

- 1
`**` 比 `-` 的优先级更高, 所以 `-3**2` 会被解释成 `-(3**2)`，因此，结果是 `-9`。要避免这个问题，并且得到 `9`，可以用 `(-3)**2`。
- 2
和其他语言不一样，特殊字符如 `\n` 在单引号（`'...'`）和双引号（`"..."`）里的意义一样。这两种引号唯一的区别是，不需要在单引号里转义双引号 `"`，但必须把单引号转义成 `\'`，反之亦然。

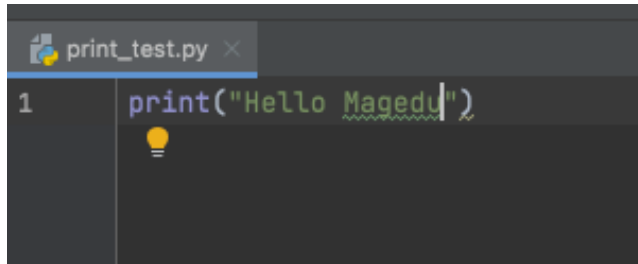
print() 函数

这里先说一下 `print()` 函数，如果你是新手，可能对函数不太了解，没关系，在这里你只要了解它的组成部分和作用就可以了，后面函数这一块会详细说明的。

`print()` 函数由两部分构成：

1. 指令：`print`
2. 指令的执行对象，在 `print` 后面的括号里的内容

而 `print()` 函数的作用是让计算机把你给它的指令结果，显示在屏幕的终端上。这里的指令就是你在 `print()` 函数里的内容。



它的执行流程如下：

1. 向解释器发出指令，打印 'Hello Magedu'
2. 解析器把代码解释为计算机能读懂的机器语言
3. 计算机执行完后就打印结果

print()的sep参数

```
print("Magedu", "Web安全", sep='--')
```

运行结果

```
Magedu--网络安全
```

格式化输出

```
name = "Magedu"
age = 10
gender = "男"
print("%s++%s++%s" % (name, age, gender))
```

运行结果

```
Magedu++10++男
```

format使用

```
name = "test"
age = 10
gender = "男"
print("我叫{0}今年{1}岁是个{2}".format(name, age, gender))
print("我叫{2}今年{0}岁是个{1}".format(name, age, gender))
```

运行结果

```
我叫test今年10岁是个男
我叫男今年test岁是个10
```

其他流程控制工具

if 语句

`if` 语句用于有条件的执行:

```
if_stmt ::= "if" assignment_expression ":" suite
          ("elif" assignment_expression ":" suite)*
          ["else" ":" suite]
```

它通过对表达式逐个求值直至找到一个真值（请参阅 [布尔运算](#) 了解真值与假值的定义）在子句体中选择唯一匹配的一个；然后执行该子句体（而且 `if` 语句的其他部分不会被执行或求值）。如果所有表达式均为假值，则如果 `else` 子句体如果存在就会被执行。

例如：

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

`if` 语句包含零个或多个 `elif` 子句及可选的 `else` 子句。关键字 `'elif'` 是 'else if' 的缩写，适用于避免过多的缩进。`if ... elif ... elif ...` 序列可以当作其他语言中 `switch` 或 `case` 语句的替代品。

如果要把一个值与多个常量进行比较，或者检查特定类型或属性，`match` 语句更实用。详见 [match 语句](#)。

for 语句

`for` 语句用于对序列（例如字符串、元组或列表）或其他可迭代对象中的元素进行迭代：

```
for_stmt ::= "for" target_list "in" expression_list ":" suite
           ["else" ":" suite]
```

表达式列表会被求值一次；它应该产生一个可迭代对象。系统将为 `expression_list` 的结果创建一个迭代器，然后将为迭代器所提供的每一项执行一次子句体，具体次序与迭代器的返回顺序一致。每一项会按标准赋值规则（参见 [赋值语句](#)）被依次赋值给目标列表，然后子句体将被执行。当所有项被耗尽时（这会在序列为空或迭代器引发 [StopIteration](#) 异常时立刻发生），`else` 子句的子句体如果存在将会被执行，并终止循环。

第一个子句体中的 `break` 语句在执行时将终止循环且不执行 `else` 子句体。第一个子句体中的 `continue` 语句在执行时将跳过子句体中的剩余部分并转往下一项继续执行，或者在没有下一项时转往 `else` 子句执行。

for 循环会对目标列表中的变量进行赋值。这将覆盖之前对这些变量的所有赋值，包括在 for 循环体中的赋值：

```
for i in range(10):
    print(i)
    i = 5                # this will not affect the for-loop
                        # because i will be overwritten with the next
                        # index in the range
```

目标列表中的名称在循环结束时不会被删除，但如果序列为空，则它们根本不会被循环所赋值。提示：内置函数 `range()` 会返回一个可迭代的整数序列，适用于模拟 Pascal 中的 `for i := a to b do` 这种效果；例如 `list(range(3))` 会返回列表 `[0, 1, 2]`。

Python 的 `for` 语句与 C 或 Pascal 中的不同。Python 的 `for` 语句不迭代算术递增数值（如 Pascal），或是给予用户定义迭代步骤和暂停条件的能力（如 C），而是迭代列表或字符串等任意序列，元素的迭代顺序与在序列中出现的顺序一致。例如：

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

遍历集合时修改集合的内容，会很容易生成错误的结果。因此不能直接进行循环，而是应遍历该集合的副本或创建新的集合：

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

range() 函数

内置函数 `range()` 常用于遍历数字序列，该函数可以生成算术级数：


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

生成的序列不包含给定的终止数值：`range(10)` 生成 10 个值，这是一个长度为 10 的序列，其中的元素索引都是合法的。`range` 可以从 0 开始，还可以按指定幅度递增（递增幅度称为 '步进'，支持负数）：

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

`range()` 和 `len()` 组合在一起，可以按索引迭代序列：

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

不过，大多数情况下，`enumerate()` 函数更便捷，详见 [循环的技巧](#)。

如果只输出 `range`，会出现意想不到的结果：

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

`range()` 返回对象的操作和列表很像，但其实这两种对象不是一回事。迭代时，该对象基于所需序列返回连续项，并没有生成真正的列表，从而节省了空间。

这种对象称为可迭代对象 `iterable`，函数或程序结构可通过该对象获取连续项，直到所有元素全部迭代完毕。`for` 语句就是这样的架构，`sum()` 是一种把可迭代对象作为参数的函数：

```
>>> sum(range(4)) # 0 + 1 + 2 + 3
6
```

循环中的 `break`、`continue` 语句及 `else` 子句

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

循环语句支持 `else` 子句；`for` 循环中，可迭代对象中的元素全部循环完毕，或 `while` 循环的条件为假时，执行该子句；`break` 语句终止循环时，不执行该子句。请看下面这个查找素数的循环示例：

```
>>> 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
```

(没错，这段代码就是这么写。仔细看：`else` 子句属于 `for` 循环，不属于 `if` 语句。)

与 `if` 语句相比，循环的 `else` 子句更像 `try` 的 `else` 子句：`try` 的 `else` 子句在未触发异常时执行，循环的 `else` 子句则在未运行 `break` 时执行。`try` 语句和异常详见 [异常的处理](#)。

`continue` 语句也借鉴自 C 语言，表示继续执行循环的下一次迭代：

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found an odd number", num)
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
```

```
Found an even number 8
Found an odd number 9
```

pass 语句

`pass` 语句不执行任何操作。语法上需要一个语句，但程序不实际执行任何动作时，可以使用该语句。例如：

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

下面这段代码创建了一个最小的类：

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

`pass` 还可以用作函数或条件子句的占位符，让开发者聚焦更抽象的层次。此时，程序直接忽略 `pass`：

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

match 语句

`match` 语句接受一个表达式并将其值与作为一个或多个 `case` 块给出的连续模式进行比较。这表面上类似于 C、Java 或 JavaScript（以及许多其他语言）中的 `switch` 语句，但它也可以将组件（序列元素或对象属性）从值中提取到变量中。

最简单的形式是将一个目标值与一个或多个字面值进行比较：

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

注意最后一个代码块：“变量名” `_` 被作为 通配符 并必定会匹配成功。如果没有 `case` 语句匹配成功，则不会执行任何分支。

使用 `|` (“or”) 在一个模式中可以组合多个字面值：

```
case 401 | 403 | 404:
    return "Not allowed"
```

模式的形式类似解包赋值，并可被用于绑定变量：

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

请仔细研究此代码！第一个模式有两个字面值，可以看作是上面所示字面值模式的扩展。但接下来的两个模式结合了一个字面值和一个变量，而变量 **绑定** 了一个来自目标的值（`point`）。第四个模式捕获了两个值，这使得它在概念上类似于解包赋值 `(x, y) = point`。

如果使用类实现数据结构，可在类名后加一个类似于构造器的参数列表，这样做可以把属性放到变量里：

```
class Point:
    x: int
    y: int

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

可在 `dataclass` 等支持属性排序的内置类中使用位置参数。还可在类中设置 `__match_args__` 特殊属性为模式的属性定义指定位置。如果它被设为 `("x", "y")`，则以下模式均为等价的，并且都把 `y` 属性绑定到 `var` 变量：

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

读取模式的推荐方式是将它们看做是你会在赋值操作左侧放置的内容的扩展形式，以便理解各个变量将会被设置的值。只有单独的名称（例如上面的 `var`）会被 `match` 语句所赋值。带点号的名称（例如 `foo.bar`）、属性名称（例如上面的 `x=` 和 `y=`）或类名称（通过其后的 `"(...)"` 来识别，例如上面的 `Point`）都绝不会被赋值。

模式可以任意地嵌套。例如，如果有一个由点组成的短列表，则可使用如下方式进行匹配：

```
match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

为模式添加成为守护项的 `if` 子句。如果守护项的值为假，则 `match` 继续匹配下一个 `case` 语句块。注意，值的捕获发生在守护项被求值之前：

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

`match` 语句的其他特性：

- 与解包赋值类似，元组和列表模式具有完全相同的含义，并且实际上能匹配任意序列。但它们不能匹配迭代器或字符串。
- 序列模式支持扩展解包操作：`[x, y, *rest]` 和 `(x, y, *rest)` 的作用类似于解包赋值。在 `*` 之后的名称也可以为 `_`，因此，`(x, y, *_)` 可以匹配包含至少两个条目的序列，而不必绑定其余的条目。
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dictionary. Unlike sequence patterns, extra keys are ignored. An unpacking like `**rest` is also supported. (But `**_` would be redundant, so it is not allowed.)
- 使用 `as` 关键字可以捕获子模式：

```
case (Point(x1, y1), Point(x2, y2) as p2): ...
```

将把输入的第二个元素捕获为 `p2`（只要输入是包含两个点的序列）

- 大多数字面值是按相等性比较的，但是单例对象 `True`、`False` 和 `None` 则是按标识号比较的。
- 模式可以使用命名常量。这些命名常量必须为带点号的名称以防止它们被解读为捕获变量：

```
from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

定义函数

下列代码创建一个可以输出限定数值内的斐波那契数列函数：

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

定义函数使用关键字 `def`，后跟函数名与括号内的形参列表。函数语句从下一行开始，并且必须缩进。

函数内的第一条语句是字符串时，该字符串就是文档字符串，也称为 *docstring*，详见 [文档字符串](#)。利用文档字符串可以自动生成在线文档或打印版文档，还可以让开发者在浏览代码时直接查阅文档；Python 开发者最好养成在代码中加入文档字符串的好习惯。

函数在 *执行* 时使用函数局部变量符号表，所有函数变量赋值都存在局部符号表中；引用变量时，首先，在局部符号表里查找变量，然后，是外层函数局部符号表，再是全局符号表，最后是内置名称符号表。因此，尽管可以引用全局变量和外层函数的变量，但最好不要在函数内直接赋值（除非是 `global` 语句定义的全局变量，或 `nonlocal` 语句定义的外层函数变量）。

在调用函数时会把实际参数（实参）引入到被调用函数的局部符号表中；因此，实参是使用 *按值调用* 来传递的（其中的 *值* 始终是对象的 *引用* 而不是对象的值）。¹ 当一个函数调用另外一个函数时，会为该调用创建一个新的局部符号表。

函数定义在当前符号表中把函数名与函数对象关联在一起。解释器把函数名指向的对象作为用户自定义函数。还可以使用其他名称指向同一个函数对象，并访问该函数：

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

`fib` 不返回值，因此，其他语言不把它当作函数，而是当作过程。事实上，没有 `return` 语句的函数也返回值，只不过这个值比较是 `None`（是一个内置名称）。一般来说，解释器不会输出单独的返回值 `None`，如需查看该值，可以使用 `print()`：

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

编写不直接输出斐波那契数列运算结果，而是返回运算结果列表的函数也非常简单：

```
>>> 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]
```

本例也新引入了一些 Python 功能：

- `return` 语句返回函数的值。`return` 语句不带表达式参数时，返回 `None`。函数执行完毕退出也返回 `None`。
- `result.append(a)` 语句调用了列表对象 `result` 的 *方法*。方法是“从属于”对象的函数，命名为 `obj.methodname`，`obj` 是对象（也可以是表达式），`methodname` 是对象类型定义的方法名。不同类型定义不同的方法，不同类型的方法名可以相同，且不会引起歧义。（用 *类* 可以自定义对象类型和方法，详见 [类](#)）示例中的方法 `append()` 是为列表对象定义的，用于在列表末尾添加新元素。本例中，该方法相当于 `result = result + [a]`，但更有效。

函数定义详解

函数定义支持可变数量的参数。这里列出三种可以组合使用的形式。

默认值参数

为参数指定默认值是非常有用的方式。调用函数时，可以使用比定义时更少的参数，例如：

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    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 ValueError('invalid user response')
    print(reminder)
```

该函数可以用以下方式调用：

- 只给出必选实参：`ask_ok('Do you really want to quit?')`
- 给出一个可选实参：`ask_ok('OK to overwrite the file?', 2)`
- 给出所有实参：`ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

本例还使用了关键字 `in`，用于确认序列中是否包含某个值。

默认值在 定义 作用域里的函数定义中求值，所以：

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

上例输出的是 `5`。

重要警告： 默认值只计算一次。默认值为列表、字典或类实例等可变对象时，会产生与该规则不同的结果。例如，下面的函数会累积后续调用时传递的参数：

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```


输出结果如下：

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

不想在后续调用之间共享默认值时，应以如下方式编写函数：

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

关键字参数

`kwarg=value` 形式的 [关键字参数](#) 也可以用于调用函数。函数示例如下：

```
def parrot(voltage, state='a stiff', action='vroom', 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, "!")
```

该函数接受一个必选参数（`voltage`）和三个可选参数（`state`、`action` 和 `type`）。该函数可用下列方式调用：

```
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
```

以下调用函数的方式都无效：

```
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
```

函数调用时，关键字参数必须跟在位置参数后面。所有传递的关键字参数都必须匹配一个函数接受的参数（比如，`actor` 不是函数 `parrot` 的有效参数），关键字参数的顺序并不重要。这也包括必选参数，（比如，`parrot(voltage=1000)` 也有效）。不能对同一个参数多次赋值，下面就是一个因此限制而失败的例子：

```
>>> 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'
```

最后一个形参为 `**name` 形式时，接收一个字典（详见 [映射类型 --- dict](#)），该字典包含与函数中已定义形参对应之外的所有关键字参数。`**name` 形参可以与 `*name` 形参（下一小节介绍）组合使用（`*name` 必须在 `**name` 前面），`*name` 形参接收一个 [元组](#)，该元组包含形参列表之外的位置参数。例如，可以定义下面这样的函数：

```
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)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

该函数可以用如下方式调用：

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

输出结果如下：

```
-- 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.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

注意，关键字参数在输出结果中的顺序与调用函数时的顺序一致。

第一个函数定义 `standard_arg` 是最常见的形式，对调用方式没有任何限制，可以按位置也可以按关键字传递参数：

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

第二个函数 `pos_only_arg` 的函数定义中有 `/`，仅限使用位置形参：

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments passed as keyword arguments: 'arg'
```

第三个函数 `kwd_only_args` 的函数定义通过 `*` 表明仅限关键字参数：

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

最后一个函数在同一个函数定义中，使用了全部三种调用惯例：

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: combined_example() got some positional-only arguments passed as keyword arguments: 'pos_only'
```

下面的函数定义中，`kwargs` 把 `name` 当作键，因此，可能与位置参数 `name` 产生潜在冲突：

```
def foo(name, **kwargs):  
    return 'name' in kwargs
```

调用该函数不可能返回 `True`，因为关键字 `'name'` 总与第一个形参绑定。例如：

```
>>> foo(1, **{'name': 2})  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: foo() got multiple values for argument 'name'  
>>>
```

加上 `/`（仅限位置参数）后，就可以了。此时，函数定义把 `name` 当作位置参数，`'name'` 也可以作为关键字参数的键：

```
def foo(name, /, **kwargs):  
    return 'name' in kwargs  
>>> foo(1, **{'name': 2})  
True
```

换句话说，仅限位置形参的名称可以在 `**kwargs` 中使用，而不产生歧义。

数据结构

列表详解

列表数据类型支持很多方法，列表对象的所有方法所示如下：

- `list.append(x)`

在列表末尾添加一个元素，相当于 `a[len(a):] = [x]`。

- `list.extend(iterable)`

用可迭代对象的元素扩展列表。相当于 `a[len(a):] = iterable`。

- `list.insert(i, x)`

在指定位置插入元素。第一个参数是插入元素的索引，因此，`a.insert(0, x)` 在列表开头插入元素，`a.insert(len(a), x)` 等同于 `a.append(x)`。

- `list.remove(x)`

从列表中删除第一个值为 `x` 的元素。未找到指定元素时，触发 `ValueError` 异常。

- `list.pop([i])`

删除列表中指定位置的元素，并返回被删除的元素。未指定位置时，`a.pop()` 删除并返回列表的最后一个元素。（方法签名中 *i* 两边的方括号表示该参数是可选的，不是要求输入方括号。这种表示法常见于 Python 参考库）。

- `list.clear()`

删除列表里的所有元素，相当于 `del a[:]`。

- `list.index(x[, start[, end]])`

返回列表中第一个值为 *x* 的元素的零基索引。未找到指定元素时，触发 `ValueError` 异常。可选参数 *start* 和 *end* 是切片符号，用于将搜索限制为列表的特定子序列。返回的索引是相对于整个序列的开始计算的，而不是 *start* 参数。

- `list.count(x)`

返回列表中元素 *x* 出现的次数。

- `list.sort(**, key=None, reverse=False)`

就地排序列表中的元素（要了解自定义排序参数，详见 `sorted()`）。

- `list.reverse()`

翻转列表中的元素。

- `list.copy()`

返回列表的浅拷贝。相当于 `a[:]`。

多数列表方法示例：

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4) # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

`insert`、`remove`、`sort` 等方法只修改列表，不输出返回值——返回的默认值为 `None`。¹ 这是所有 Python 可变数据结构的设计原则。

还有，不是所有数据都可以排序或比较。例如，`[None, 'hello', 10]` 就不可排序，因为整数不能与字符串对比，而 `None` 不能与其他类型对比。有些类型根本就没有定义顺序关系，例如，`3+4j < 5+7j` 这种对比操作就是无效的。

用列表实现堆栈

使用列表方法实现堆栈非常容易，最后插入的最先取出（“后进先出”）。把元素添加到堆栈的顶端，使用 `append()`。从堆栈顶部取出元素，使用 `pop()`，不用指定索引。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]
```

用列表实现队列

列表也可以用作队列，最先加入的元素，最先取出（“先进先出”）；然而，列表作为队列的效率很低。因为，在列表末尾添加和删除元素非常快，但在列表开头插入或移除元素却很慢（因为所有其他元素都必须移动一位）。

实现队列最好用 `collections.deque`，可以快速从两端添加或删除元素。例如：

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

列表推导式

列表推导式创建列表的方式更简洁。常见的用法为，对序列或可迭代对象中的每个元素应用某种操作，用生成的结果创建新的列表；或用满足特定条件的元素创建子序列。

例如，创建平方值的列表：

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

注意，这段代码创建（或覆盖）变量 `x`，该变量在循环结束后仍然存在。下述方法可以无副作用地计算平方列表：

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

或等价于：

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

上面这种写法更简洁、易读。

列表推导式的方括号内包含以下内容：一个表达式，后面为一个 `for` 子句，然后，是零个或多个 `for` 或 `if` 子句。结果是由表达式依据 `for` 和 `if` 子句求值计算而得出一个新列表。举例来说，以下列表推导式将两个列表中不相等的元素组合起来：

```
>>> [(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)]
```

等价于：

```
>>> 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，上面两段代码中，`for` 和 `if` 的顺序相同。

表达式是元组（例如上例的 `(x, y)`）时，必须加上括号：

```
>>> 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 <module>
    [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']

```

嵌套的列表推导式

列表推导式中的初始表达式可以是任何表达式，甚至可以是另一个列表推导式。

下面这个 3x4 矩阵，由 3 个长度为 4 的列表组成：

```

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

```

下面的列表推导式可以转置行列：

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

如上节所示，嵌套的列表推导式基于其后的 `for` 求值，所以这个例子等价于：

```
>>> 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]]
```

反过来说，也等价于：

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

实际应用中，最好用内置函数替代复杂的流程语句。此时，`zip()` 函数更好用：

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

关于本行中星号的详细说明，参见 [解包实参列表](#)。

del 语句

`del` 语句按索引，而不是值从列表中移除元素。与返回值的 `pop()` 方法不同，`del` 语句也可以从列表中移除切片，或清空整个列表（之前是将空列表赋值给切片）。例如：

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

`del` 也可以用来删除整个变量：

```
>>> del a
```

此后，再引用 `a` 就会报错（直到为它赋与另一个值）。后文会介绍 `del` 的其他用法。

元组和序列

列表和字符串有很多共性，例如，索引和切片操作。这两种数据类型是 *序列*（参见 [序列类型 --- list, tuple, range](#)）。随着 Python 语言的发展，其他的序列类型也被加入其中。本节介绍另一种标准序列类型：*元组*。

元组由多个用逗号隔开的值组成，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])
```

输出时，元组都要由圆括号标注，这样才能正确地解释嵌套元组。输入时，圆括号可有可无，不过经常是必须的（如果元组是更大的表达式的一部分）。不允许为元组中的单个元素赋值，当然，可以创建含列表等可变对象的元组。

虽然，元组与列表很像，但使用场景不同，用途也不同。元组是 [immutable](#)（不可变的），一般可包含异质元素序列，通过解包（见本节下文）或索引访问（如果是 [namedtuples](#)，可以属性访问）。列表是 [mutable](#)（可变的），列表元素一般为同质类型，可迭代访问。

构造 0 个或 1 个元素的元组比较特殊：为了适应这种情况，对句法有一些额外的改变。用一对空圆括号就可以创建空元组；只有一个元素的元组可以通过在这个元素后添加逗号来构建（圆括号里只有一个值的话不够明确）。丑陋，但是有效。例如：

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

语句 `t = 12345, 54321, 'hello!'` 是 *元组打包* 的例子：值 `12345`，`54321` 和 `'hello!'` 一起被打包进元组。逆操作也可以：

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

称之为 *序列解包* 也是妥妥的，适用于右侧的任何序列。序列解包时，左侧变量与右侧序列元素的数量应相等。注意，多重赋值其实只是元组打包和序列解包的组合。

集合

Python 还支持 *集合* 这种数据类型。集合是由不重复元素组成的无序容器。基本用法包括成员检测、消除重复元素。集合对象支持合集、交集、差集、对称差分等数学运算。

创建集合用花括号或 [set\(\)](#) 函数。注意，创建空集合只能用 `set()`，不能用 `{}`，`{}` 创建的是空字典，下一小节介绍数据结构：字典。

以下是一些简单的示例

```
>>> 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 a or b or both
{'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'}
```

与 [列表推导式](#) 类似，集合也支持推导式：

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

字典

字典（参见 [映射类型 --- dict](#)）也是一种常用的 Python 内置数据类型。其他语言可能把字典称为 *联合内存* 或 *联合数组*。与以连续整数为索引的序列不同，字典以 *关键字* 为索引，关键字通常是字符串或数字，也可以是其他任意不可变类型。只包含字符串、数字、元组的元组，也可以用作关键字。但如果元组直接或间接地包含了可变对象，就不能用作关键字。列表不能当关键字，因为列表可以用索引、切片、`append()`、`extend()` 等方法修改。

可以把字典理解为 *键值对* 的集合，但字典的键必须是唯一的。花括号 `{}` 用于创建空字典。另一种初始化字典的方式，在花括号里输入逗号分隔的键值对，这也是字典的输出方式。

字典的主要用途是通过关键字存储、提取值。用 `del` 可以删除键值对。用已存在的关键字存储值，与该关键字关联的旧值会被取代。通过不存在的键提取值，则会报错。

对字典执行 `list(d)` 操作，返回该字典中所有键的列表，按插入次序排列（如需排序，请使用 `sorted(d)`）。检查字典里是否存在某个键，使用关键字 `in`。

以下是一些字典的简单示例：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
```

```
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

`dict()` 构造函数可以直接用键值对序列创建字典：

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

字典推导式可以用任意键值表达式创建字典：

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

关键字是比较简单的字符串时，直接用关键字参数指定键值对更便捷：

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

循环的技巧

在字典中循环时，用 `items()` 方法可同时取出键和对应的值：

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

enumerate

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

print(type(enumerate(seq)))
print(enumerate(seq))
print(list(enumerate(seq)))

for index,value in enumerate(seq):
    print(index,value,sep='---')
```

运行结果

```
<class 'enumerate'>
<enumerate object at 0x000001708394E288>
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'test')]
0---a
1---b
2---c
3---test
```

在序列中循环时，用 `enumerate()` 函数可以同时取出位置索引和对应的值：

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

同时循环两个或多个序列时，用 `zip()` 函数可以将其内的元素一一匹配：

zip

`zip()` 函数用于将可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的列表。如果各个迭代器的元素个数不一致，则返回列表长度与最短的对象相同，利用 `*` 号操作符，可以将元组解压为列表。

```
a = [1,2,3,4,5]
b = ['a','b','c','d','e']
c = zip(a,b)
print(c)
print(type(c))
print(list(c))
```

运行结果

```
<zip object at 0x00000226F07124C8>
<class 'zip'>
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

```
>>> 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.
```

逆向循环序列时，先正向定位序列，然后调用 `reversed()` 函数：

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

按指定顺序循环序列，可以用 `sorted()` 函数，在不改动原序列的基础上，返回一个重新序列：

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

使用 `set()` 去除序列中的重复元素。使用 `sorted()` 加 `set()` 则按排序后的顺序，循环遍历序列中的唯一元素：

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

一般来说，在循环中修改列表的内容时，创建新列表比较简单，且安全：


```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

模块

退出 Python 解释器后，再次进入时，之前在 Python 解释器中定义的函数和变量就丢失了。因此，编写较长程序时，建议用文本编辑器代替解释器，执行文件中的输入内容，这就是编写 *脚本*。随着程序越来越长，为了方便维护，最好把脚本拆分成多个文件。编写脚本还有一个好处，不同程序调用同一个函数时，不用每次把函数复制到各个程序。

为实现这些需求，Python 把各种定义存入一个文件，在脚本或解释器的交互式实例中使用。这个文件就是 *模块*；模块中的定义可以 *导入* 到其他模块或 *主* 模块（在顶层和计算器模式下，执行脚本中可访问的变量集）。

模块是包含 Python 定义和语句的文件。其文件名是模块名加后缀名 `.py`。在模块内部，通过全局变量 `__name__` 可以获取模块名（即字符串）。例如，用文本编辑器在当前目录下创建 `fibonacci.py` 文件，输入以下内容：

```
# Fibonacci numbers module

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

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

现在，进入 Python 解释器，用以下命令导入该模块：

```
>>> import fibo
```

这项操作不直接把 `fibo` 函数定义的名称导入到当前符号表，只导入模块名 `fibo`。要使用模块名访问函数：

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

如果经常使用某个函数，可以把它赋值给局部变量：

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

模块详解

模块包含可执行语句及函数定义。这些语句用于初始化模块，且仅在 `import` 语句 第一次 遇到模块名时执行。[1](#) (文件作为脚本运行时，也会执行这些语句。)

模块有自己的私有符号表，用作模块中所有函数的全局符号表。因此，在模块内使用全局变量时，不用担心与用户定义的全局变量发生冲突。另一方面，可以用与访问模块函数一样的标记法，访问模块的全局变量，`modname.itemname`。

可以把其他模块导入模块。按惯例，所有 `import` 语句都放在模块（或脚本）开头，但这不是必须的。导入的模块名存在导入模块的全局符号表里。

`import` 语句有一个变体，可以直接把模块里的名称导入到另一个模块的符号表。例如：

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这段代码不会把模块名导入到局部符号表里（因此，本例没有定义 `fibo`）。

还有一种变体可以导入模块内定义的所有名称：

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这种方式会导入所有不以下划线（`_`）开头的名称。大多数情况下，不要用这个功能，这种方式向解释器导入了一批未知的名称，可能会覆盖已经定义的名称。

注意，一般情况下，不建议从模块或包内导入 `*`，因为，这项操作经常让代码变得难以理解。不过，为了在交互式编译器中少打几个字，这么用也没问题。

模块名后使用 `as` 时，直接把 `as` 后的名称与导入模块绑定。

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

与 `import fibo` 一样，这种方式也可以有效地导入模块，唯一的区别是，导入的名称是 `fib`。

`from` 中也可以使用这种方式，效果类似：

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

注解

为了保证运行效率，每次解释器会话只导入一次模块。如果更改了模块内容，必须重启解释器；仅交互测试一个模块时，也可以使用 `importlib.reload()`，例如 `import importlib; importlib.reload(modulename)`。

以脚本方式执行模块

可以用以下方式运行 Python 模块：

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

这项操作将执行模块里的代码，和导入模块一样，但会把 `__name__` 赋值为 `"__main__"`。也就是把下列代码添加到模块末尾：

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

既可以把这个文件当脚本使用，也可以用作导入的模块，因为，解析命令行的代码只有在模块以“main”文件执行时才会运行：

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

导入模块时，不运行这些代码：

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

这种操作常用于为模块提供便捷用户接口，或用于测试（把模块当作执行测试套件的脚本运行）。

模块搜索路径

导入 `spam` 模块时，解释器首先查找名为 `spam` 的内置模块。如果没找到，解释器再从 `sys.path` 变量中的目录列表里查找 `spam.py` 文件。`sys.path` 初始化时包含以下位置：

- 输入脚本的目录（或未指定文件时的当前目录）。
- `PYTHONPATH`（目录列表，与 shell 变量 `PATH` 的语法一样）。
- The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

注解

在支持 symlink 的文件系统中，输入脚本目录是在追加 symlink 后计算出来的。换句话说，包含 symlink 的目录并没有添加至模块搜索路径。

初始化后，Python 程序可以更改 `sys.path`。运行脚本的目录在标准库路径之前，置于搜索路径的开头。即，加载的是该目录里的脚本，而不是标准库的同名模块。除非刻意替换，否则会报错。详见 [标准模块](#)。

“已编译的” Python 文件

为了快速加载模块，Python 把模块的编译版缓存在 `__pycache__` 目录中，文件名为 `module.*version*.pyc`，`version` 对编译文件格式进行编码，一般是 Python 的版本号。例如，CPython 的 3.3 发行版中，`spam.py` 的编译版本缓存为 `__pycache__/spam.cpython-33.pyc`。使用这种命名惯例，可以让不同 Python 发行版及不同版本的已编译模块共存。

Python 对比编译版本与源码的修改日期，查看它是否已过期，是否要重新编译，此过程完全自动化。此外，编译模块与平台无关，因此，可在不同架构系统之间共享相同的支持库。

Python 在两种情况下不检查缓存。其一，从命令行直接载入模块，只重新编译，不存储编译结果；其二，没有源模块，就不会检查缓存。为了支持无源文件（仅编译）发行版本，编译模块必须在源目录下，并且绝不能有源模块。

给专业人士的一些小建议：

- 在 Python 命令中使用 `-O` 或 `-OO` 开关，可以减小编译模块的大小。`-O` 去除断言语句，`-OO` 去除断言语句和 `doc` 字符串。有些程序可能依赖于这些内容，因此，没有十足的把握，不要使用这两个选项。“优化过的”模块带有 `opt-` 标签，并且文件通常会一小些。将来的发行版或许会改进优化的效果。
- 从 `.pyc` 文件读取的程序不比从 `.py` 读取的执行速度快，`.pyc` 文件只是加载速度更快。
- `compileall` 模块可以为一个目录下的所有模块创建 `.pyc` 文件。
- 本过程的细节及决策流程图，详见 [PEP 3147](#)。

标准模块

Python 自带一个标准模块的库，它在 Python 库参考（此处以下称为“库参考”）里另外描述。一些模块是内嵌到编译器里面的，它们给一些虽并非语言核心但却内嵌的操作提供接口，要么是为了效率，要么是给操作系统基础操作例如系统调入提供接口。这些模块集是一个配置选项，并且还依赖于底层的操作系统。例如，`winreg` 模块只在 Windows 系统上提供。一个特别值得注意的模块 `sys`，它被内嵌到每一个 Python 编译器中。`sys.ps1` 和 `sys.ps2` 变量定义了一些字符，它们可以用作主提示符和辅助提示符：

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

只有解释器用于交互模式时，才定义这两个变量。

变量 `sys.path` 是字符串列表，用于确定解释器的模块搜索路径。该变量以环境变量 `PYTHONPATH` 提取的默认路径进行初始化，如未设置 `PYTHONPATH`，则使用内置的默认路径。可以用标准列表操作修改该变量：

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

`dir()` 函数

内置函数 `dir()` 用于查找模块定义的名称。返回结果是经过排序的字符串列表：

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
```

```
'warnoptions']
```

没有参数时, `dir()` 列出当前定义的名称:

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

注意, 该函数列出所有类型的名称: 变量、模块、函数等。

`dir()` 不会列出内置函数和变量的名称。这些内容的定义在标准模块 `builtins` 里:

```
>>> 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']
```

包

包是一种用“点式模块名”构造 Python 模块命名空间的方法。例如，模块名 `A.B` 表示包 `A` 中名为 `B` 的子模块。正如模块可以区分不同模块之间的全局变量名称一样，点式模块名可以区分 NumPy 或 Pillow 等不同多模块包之间的模块名称。

假设要为统一处理声音文件与声音数据设计一个模块集（“包”）。声音文件的格式很多（通常以扩展名来识别，例如：`.wav`，`.aiff`，`.au`），因此，为了不同文件格式之间的转换，需要创建和维护一个不断增长的模块集合。为了实现对声音数据的不同处理（例如，混声、添加回声、均衡器功能、创造人工立体声效果），还要编写无穷无尽的模块流。下面这个分级文件树展示了这个包的架构：

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

导入包时，Python 搜索 `sys.path` 里的目录，查找包的子目录。

Python 只把含 `__init__.py` 文件的目录当成包。这样可以防止以 `string` 等通用名称命名的目录，无意中屏蔽出现在后方模块搜索路径中的有效模块。最简情况下，`__init__.py` 只是一个空文件，但该文件也可以执行包的初始化代码，或设置 `__all__` 变量，详见下文。

还可以从包中导入单个模块，例如：

```
import sound.effects.echo
```

这段代码加载子模块 `sound.effects.echo`，但引用时必须使用子模块的全名：

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

另一种导入子模块的方法是：

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

这段代码还可以加载子模块 `echo`，不加包前缀也可以使用。因此，可以按如下方式使用：

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

Import 语句的另一种变体是直接导入所需的函数或变量：

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

同样，这样也会加载子模块 `echo`，但可以直接使用函数 `echofilter()`：

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

注意，使用 `from package import item` 时，`item` 可以是包的子模块（或子包），也可以是包中定义的函数、类或变量等其他名称。`import` 语句首先测试包中是否定义了 `item`；如果未在包中定义，则假定 `item` 是模块，并尝试加载。如果找不到 `item`，则触发 `ImportError` 异常。

相反，使用 `import item.subitem.subsubitem` 句法时，除最后一项外，每个 `item` 都必须是包；最后一项可以是模块或包，但不能是上一项中定义的类、函数或变量。

从包中导入 *

使用 `from sound.effects import *` 时会发生什么？理想情况下，该语句在文件系统查找并导入包的所有子模块。这项操作花费的时间较长，并且导入子模块可能会产生不必要的副作用，这种副作用只有在显式导入子模块时才会发生。

唯一的解决方案是提供包的显式索引。`import` 语句使用如下惯例：如果包的 `__init__.py` 代码定义了列表 `__all__`，运行 `from package import *` 时，它就是用于导入的模块名列表。发布包的新版本时，包的作者应更新此列表。如果包的作者认为没有必要在包中执行导入 `*` 操作，也可以不提供此列表。例如，`sound/effects/__init__.py` 文件包含以下代码：

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

即，`from sound.effects import *` 将导入 `sound` 包中的这三个命名子模块。

如果没有定义 `__all__`，`from sound.effects import *` 语句不会把包 `sound.effects` 中所有子模块都导入到当前命名空间；该语句只确保导入包 `sound.effects`（可能还会运行 `__init__.py` 中的初始化代码），然后，再导入包中定义的名称。这些名称包括 `__init__.py` 中定义的任何名称（以及显式加载的子模块），还包括之前 `import` 语句显式加载的包里的子模块。请看以下代码：


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

本例中，执行 `from...import` 语句时，将把 `echo` 和 `surround` 模块导入至当前命名空间，因为，它们是在 `sound.effects` 包里定义的。（该导入操作在定义了 `__all__` 时也有效。）

虽然，可以把模块设计为用 `import *` 时只导出遵循指定模式的名称，但仍不提倡在生产代码中使用这种做法。

记住，使用 `from package import specific_submodule` 没有任何问题！实际上，除了导入模块使用不同包的同名子模块之外，这种方式是推荐用法。

子包

包中含有多个子包时（与示例中的 `sound` 包一样），可以使用绝对导入引用兄弟包中的子模块。例如，要在模块 `sound.filters.vocoder` 中使用 `sound.effects` 包的 `echo` 模块时，可以用 `from sound.effects import echo` 导入。

还可以用 `import` 语句的 `from module import name` 形式执行相对导入。这些导入语句使用前导句点表示相对导入中的当前包和父包。例如，相对于 `surround` 模块，可以使用：

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

注意，相对导入基于当前模块名。因为主模块名是 `"__main__"`，所以 Python 程序的主模块必须始终使用绝对导入。

文件对象的方法

本节下文中的例子假定已创建 `f` 文件对象。

`f.read(size)` 可用于读取文件内容，它会读取一些数据，并返回字符串（文本模式），或字节串对象（在二进制模式下）。`size` 是可选的数值参数。省略 `size` 或 `size` 为负数时，读取并返回整个文件的内容；文件大小是内存的两倍时，会出现问题。`size` 取其他值时，读取并返回最多 `size` 个字符（文本模式）或 `size` 个字节（二进制模式）。如已到达文件末尾，`f.read()` 返回空字符串（`''`）。

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

`f.readline()` 从文件中读取单行数据；字符串末尾保留换行符（`\n`），只有在文件不以换行符结尾时，文件的最后一行才会省略换行符。这种方式让返回值清晰明确；只要 `f.readline()` 返回空字符串，就表示已经到达了文件末尾，空行使用 `'\n'` 表示，该字符串只包含一个换行符。

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

从文件中读取多行时，可以用循环遍历整个文件对象。这种操作能高效利用内存，快速，且代码简单：

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

如需以列表形式读取文件中的所有行，可以用 `list(f)` 或 `f.readlines()`。

`f.write(string)` 把 *string* 的内容写入文件，并返回写入的字符数。

```
>>> f.write('This is a test\n')
15
```

写入其他类型的对象前，要先把它们转化为字符串（文本模式）或字节对象（二进制模式）：

```
>>> value = ('the answer', 42)
>>> s = str(value) # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` 返回整数，给出文件对象在文件中的当前位置，表示为二进制模式下时从文件开始的字节数，以及文本模式下的意义不明的数字。

`f.seek(offset, whence)` 可以改变文件对象的位置。通过向参考点添加 *offset* 计算位置；参考点由 *whence* 参数指定。*whence* 值为 0 时，表示从文件开头计算，1 表示使用当前文件位置，2 表示使用文件末尾作为参考点。省略 *whence* 时，其默认值为 0，即使用文件开头作为参考点。

```
>>>
```

```
>>> 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（模式字符串未使用 `b` 时打开的文件）中，只允许相对于文件开头搜索（使用 `seek(0, 2)` 搜索到文件末尾是个例外），唯一有效的 *offset* 值是能从 `f.tell()` 中返回的，或 0。其他 *offset* 值都会产生未定义的行为。

文件对象还支持 `isatty()` 和 `truncate()` 等方法，但不常用；文件对象的完整指南详见库参考。

使用 `json` 保存结构化数据

从文件写入或读取字符串很简单，数字则稍显麻烦，因为 `read()` 方法只返回字符串，这些字符串必须传递给 `int()` 这样的函数，接受 `'123'` 这样的字符串，并返回数字值 123。保存嵌套列表、字典等复杂数据类型时，手动解析和序列化的操作非常复杂。

Python 支持 [JSON \(JavaScript Object Notation\)](#) 这种流行数据交换格式，用户无需没完没了地编写、调试代码，才能把复杂的数据类型保存到文件。`json` 标准模块采用 Python 数据层次结构，并将之转换为字符串表示形式；这个过程称为 *serializing*（序列化）。从字符串表示中重建数据称为 *deserializing*（解序化）。在序列化和解序化之间，表示对象的字符串可能已经存储在文件或数据中，或通过网络连接发送到远方的机器。

注解

JSON 格式通常用于现代应用程序的数据交换。程序员早已对它耳熟能详，可谓是交互操作的不二之选。

只需一行简单的代码即可查看某个对象的 JSON 字符串表现形式：

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

`dumps()` 函数还有一个变体，`dump()`，它只将对象序列化为 [text file](#)。因此，如果 `f` 是 [text file](#) 对象，可以这样做：

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

要再次解码对象，如果 `f` 是已打开、供读取的 [text file](#) 对象：

```
x = json.load(f)
```

这种简单的序列化技术可以处理列表和字典，但在 JSON 中序列化任意类的实例，则需要付出额外努力。[json](#) 模块的参考包含对此的解释。

类

类把数据与功能绑定在一起。创建新类就是创建新的对象 **类型**，从而创建该类型的新 **实例**。类实例支持维持自身状态的属性，还支持（由类定义的）修改自身状态的方法。

和其他编程语言相比，Python 的类只使用了很少的新语法和语义。Python 的类有点类似于 C++ 和 Modula-3 中类的结合体，而且支持面向对象编程（OOP）的所有标准特性：类的继承机制支持多个基类、派生的类能覆盖基类的方法、类的方法能调用基类中的同名方法。对象可包含任意数量和类型的数据。和模块一样，类也支持 Python 动态特性：在运行时创建，创建后还可以修改。

如果用 C++ 术语来描述的话，类成员（包括数据成员）通常为 *public*（例外的情况见下文 [私有变量](#)），所有成员函数都是 *virtual*。与在 Modula-3 中一样，没有用于从对象的方法中引用对象成员的简写形式：方法函数在声明时，有一个显式的参数代表本对象，该参数由调用隐式提供。与在 Smalltalk 中一样，Python 的类也是对象，这为导入和重命名提供了语义支持。与 C++ 和 Modula-3 不同，Python 的内置类型可以用作基类，供用户扩展。此外，与 C++ 一样，算术运算符、下标等具有特殊语法的内置运算符都可以为类实例而重新定义。

由于缺乏关于类的公认术语，本章中偶尔会使用 Smalltalk 和 C++ 的术语。本章还会使用 Modula-3 的术语，Modula-3 的面向对象语义比 C++ 更接近 Python，但估计听说过这门语言的读者很少。

名称和对象

对象之间相互独立，多个名称（在多个作用域内）可以绑定到同一个对象。其他语言称之为别名。Python 初学者通常不容易理解这个概念，处理数字、字符串、元组等不可变基本类型时，可以不必理会。但是，对涉及可变对象，如列表、字典等大多数其他类型的 Python 代码的语义，别名可能会产生意料之外的效果。这样做，通常是为了让程序受益，因为别名在某些方面就像指针。例如，传递对象的代价很小，因为实现只传递一个指针；如果函数修改了作为参数传递的对象，调用者就可以看到更改 --- 无需 Pascal 用两个不同参数的传递机制。

Python 作用域和命名空间

在介绍类前，首先要介绍 Python 的作用域规则。类定义对命名空间有一些巧妙的技巧，了解作用域和命名空间的工作机制有利于加强对类的理解。并且，即便对于高级 Python 程序员，这方面的知识也很有用。

接下来，我们先了解一些定义。

namespace（命名空间）是映射到对象的名称。现在，大多数命名空间都使用 Python 字典实现，但除非涉及到优化性能，我们一般不会关注这方面的事情，而且将来也可能会改变这种方式。命名空间的几个常见示例：[abs\(\)](#) 函数、内置异常等的内置函数集合；模块中的全局名称；函数调用中的局部名称。对象的属性集合也算是一种命名空间。关于命名空间的一个重要知识点是，不同命名空间中的名称之间绝对没有关系；例如，两个不同的模块都可以定义 `maximize` 函数，且不会造成混淆。用户使用函数时必须要在函数名前面附上模块名。

点号之后的名称是 **属性**。例如，表达式 `z.real` 中，`real` 是对象 `z` 的属性。严格来说，对模块中名称的引用是属性引用：表达式 `modname.funcname` 中，`modname` 是模块对象，`funcname` 是模块的属性。模块属性和模块中定义的全局名称之间存在直接的映射：它们共享相同的命名空间！[1](#)

属性可以是只读或者可写的。如果可写，则可对属性赋值。模块属性是可写时，可以使用 `modname.the_answer = 42`。 `del` 语句可以删除可写属性。例如， `del modname.the_answer` 会删除 `modname` 对象中的 `the_answer` 属性。

命名空间是在不同时刻创建的，且拥有不同的生命周期。内置名称的命名空间是在 Python 解释器启动时创建的，永远不会被删除。模块的全局命名空间在读取模块定义时创建；通常，模块的命名空间也会持续到解释器退出。从脚本文件读取或交互式读取的，由解释器顶层调用执行的语句是 `_main` 模块调用的一部分，也拥有自己的全局命名空间。内置名称实际上也在模块里，即 `builtins`。

函数的本地命名空间在调用该函数时创建，并在函数返回或抛出不在函数内部处理的错误时被删除。（实际上，用“遗忘”来描述实际发生的情况会更好一些。）当然，每次递归调用都会有自己的本地命名空间。

作用域 是命名空间可直接访问的 Python 程序的文本区域。“可直接访问”的意思是，对名称的非限定引用会在命名空间中查找名称。

作用域虽然是静态确定的，但会被动态使用。执行期间的任何时刻，都会有 3 或 4 个命名空间可被直接访问的嵌套作用域：

- 最内层作用域，包含局部名称，并首先在其中进行搜索
- 封闭函数的作用域，包含非局部名称和非全局名称，从最近的封闭作用域开始搜索
- 倒数第二个作用域，包含当前模块的全局名称
- 最外层的作用域，包含内置名称的命名空间，最后搜索

如果把名称声明为全局变量，则所有引用和赋值将直接指向包含该模块的全局名称的中间作用域。重新绑定在最内层作用域以外找到的变量，使用 `nonlocal` 语句把该变量声明为非局部变量。未声明为非局部变量的变量是只读的，（写入只读变量会在最内层作用域中创建一个 **新的** 局部变量，而同名的外部变量保持不变。）

通常，当前局部作用域将（按字面文本）引用当前函数的局部名称。在函数之外，局部作用域引用与全局作用域一致的命名空间：模块的命名空间。类定义在局部命名空间内再放置另一个命名空间。

划重点，作用域是按字面文本确定的：模块内定义的函数的全局作用域就是该模块的命名空间，无论该函数从什么地方或以什么别名被调用。另一方面，实际的名称搜索是在运行时动态完成的。但是，Python 正在朝着“编译时静态名称解析”的方向发展，因此不要过于依赖动态名称解析！（局部变量已经是被静态确定了。）

Python 有一个特殊规定。如果不存在生效的 `global` 或 `nonlocal` 语句，则对名称的赋值总是会进入最内层作用域。赋值不会复制数据，只是将名称绑定到对象。删除也是如此：语句 `del x` 从局部作用域引用的命名空间中移除对 `x` 的绑定。所有引入新名称的操作都是使用局部作用域：尤其是 `import` 语句和函数定义会在局部作用域中绑定模块或函数名称。

`global` 语句用于表明特定变量在全局作用域里，并应在全局作用域中重新绑定；`nonlocal` 语句表明特定变量在外层作用域中，并应在外层作用域中重新绑定。

作用域和命名空间示例

下例演示了如何引用不同作用域和名称空间，以及 `global` 和 `nonlocal` 对变量绑定的影响：

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

示例代码的输出是：

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

注意，**局部** 赋值（这是默认状态）不会改变 `scope_test` 对 `spam` 的绑定。`nonlocal` 赋值会改变 `scope_test` 对 `spam` 的绑定，而 `global` 赋值会改变模块层级的绑定。

而且，`global` 赋值前没有 `spam` 的绑定。

初探类

类引入了一点新语法，三种新的对象类型和一些新语义。

类定义语法

最简单的类定义形式如下：

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

与函数定义 (`def` 语句) 一样，类定义必须先执行才能生效。把类定义放在 `if` 语句的分支里或函数内部试试。

在实践中，类定义内的语句通常都是函数定义，但也可以是其他语句。这部分内容稍后再讨论。类里的函数定义一般是特殊的参数列表，这是由方法调用的约定规范所指明的 --- 同样，稍后再解释。

当进入类定义时，将创建一个新的命名空间，并将其用作局部作用域 --- 因此，所有对局部变量的赋值都是在这个新命名空间之内。特别的，函数定义会绑定到这里的新函数名称。

当（从结尾处）正常离开类定义时，将创建一个 **类对象**。这基本上是一个包围在类定义所创建命名空间内容周围的包装器；我们将在下一节了解有关类对象的更多信息。原始的（在进入类定义之前起作用的）局部作用域将重新生效，类对象将在这里被绑定到类定义头所给出的类名称（在这个示例中为 `ClassName`）。

Class 对象

类对象支持两种操作：属性引用和实例化。

属性引用 使用 Python 中所有属性引用所使用的标准语法: `obj.name`。有效的属性名称是类对象被创建时存在于类命名空间中的所有名称。因此，如果类定义是这样的：

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

那么 `MyClass.i` 和 `MyClass.f` 就是有效的属性引用，将分别返回一个整数和一个函数对象。类属性也可以被赋值，因此可以通过赋值来更改 `MyClass.i` 的值。 `__doc__` 也是一个有效的属性，将返回所属类的文档字符串：
`"A simple example class"`。

类的 **实例化** 使用函数表示法。可以把类对象视为是返回该类的一个新实例的不带参数的函数。举例来说（假设使用上述的类）：

```
x = MyClass()
```

创建类的新 **实例** 并将此对象分配给局部变量 `x`。

实例化操作（“调用”类对象）会创建一个空对象。许多类喜欢创建带有特定初始状态的自定义实例。为此类定义可能包含一个名为 `__init__()` 的特殊方法，就像这样：

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

当一个类定义了 `__init__()` 方法时，类的实例化操作会自动为新创建的类实例发起调用 `__init__()`。因此在这个示例中，可以通过以下语句获得一个经初始化的新实例：

```
x = MyClass()
```

当然，`__init__()` 方法还可以有额外参数以实现更高灵活性。在这种情况下，提供给类实例化运算符的参数将被传递给 `__init__()`。例如，：

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

实例对象

现在我们能实例对象做什么？实例对象所能理解的唯一操作是属性引用。有两种有效的属性名称：数据属性和方法。

数据属性 对应于 Smalltalk 中的“实例变量”，以及 C++ 中的“数据成员”。数据属性不需要声明；像局部变量一样，它们将在第一次被赋值时产生。例如，如果 `x` 是上面创建的 `MyClass` 的实例，则以下代码段将打印数值 `16`，且不保留任何追踪信息：

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

另一类实例属性引用称为 **方法**。方法是“从属于”对象的函数。（在 Python 中，方法这个术语并不是类实例所特有的：其他对象也可以有方法。例如，列表对象具有 `append`, `insert`, `remove`, `sort` 等方法。然而，在以下讨论中，我们使用方法一词将专指类实例对象的方法，除非另外显式地说明。）

实例对象的有效方法名称依赖于其所属的类。根据定义，一个类中所有是函数对象的属性都是定义了其实例的相应方法。因此在我们的示例中，`x.f` 是有效的方法引用，因为 `MyClass.f` 是一个函数，而 `x.i` 不是方法，因为 `MyClass.i` 不是函数。但是 `x.f` 与 `MyClass.f` 并不是一回事 --- 它是一个 **方法对象**，不是函数对象。

类和实例变量

一般来说，实例变量用于每个实例的唯一数据，而类变量用于类的所有实例共享的属性和方法：

```
class Dog:

    kind = 'canine'          # class variable shared by all instances

    def __init__(self, name):
        self.name = name     # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                # shared by all dogs
'canine'
>>> e.kind                # shared by all dogs
'canine'
>>> d.name                # unique to d
'Fido'
>>> e.name                # unique to e
'Buddy'
```

正如 名称和对象中已讨论过的，共享数据可能在涉及 mutable 对象例如列表和字典的时候导致令人惊讶的结果。例如以下代码中的 *tricks* 列表不应该被用作类变量，因为所有的 *Dog* 实例将只共享一个单独的列表：

```
class Dog:

    tricks = []             # mistaken use of a class variable

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

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

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

正确的类设计应该使用实例变量：

```
class Dog:
```

```

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

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

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

```

类的call

Python中，如果在创建class的时候写了call（）方法，那么该class实例化出实例后，实例名()就是调用call（）方法。

```

class test():
    def __call__(self, *args, **kwargs):
        print("test")

t = test()
t()

```

运行结果

```
test
```

classmethod

classmethod 修饰符对应的函数不需要实例化，不需要 self 参数，但第一个参数需要是表示自身类的 cls 参数，可以用来调用类的属性，类的方法，实例化对象等。不需要实例化的意思是不创建对象，比如a = A()

```

#!/usr/bin/python
# -*- coding: UTF-8 -*-

class A(object):
    bar = 1
    def func1(self):

```

```

        print ('foo')
    @classmethod
    def func2(cls):
        print ('func2')
        print (cls.bar)
        cls().func1()    # 调用 foo 方法

A.func2()                # 不需要实例化

```

运行结果

```

func2
1
foo

```

staticmethod

对比classmethod, staticmethod没有参数

```

#!/usr/bin/python
# -*- coding: UTF-8 -*-

class C(object):
    @staticmethod
    def f():
        print('runoob');

C.f();          # 静态方法无需实例化
cobj = C()
cobj.f()        # 也可以实例化后调用

```

运行结果

```

runoob
runoob

```

property

```

#普通的get和set私有化age, __的变量就是私有化,私有化变量外部不能直接访问
class Test():
    def __init__(self, name, age):
        self.name = name
        self.__age = age

```

```

def getAge(self):
    return self.__age

def setAge(self, age):
    self.__age = age

class Test1():
    def __init__(self, name, age):
        self.name = name
        self.__age = age

    @property
    def age(self):
        return self.__age

    #设置方法
    @age.setter
    def age(self, age):
        self.__age = age

t = Test("name", 20)
print(t.getAge())
t.setAge(30)
print(t.getAge())

#简单来说，这装饰器的作用就是调用函数变成调用变量，从不能调用私有化变量，直接用一个变量代替
t1 = Test1("name", 20)
print(t1.age)
t1.age = 30
print(t1.age)

```

运行结果

```

20
30
20
30

```

继承

当然，如果不支持继承，语言特性就不值得称为“类”。派生类定义的语法如下所示：

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

名称 `BaseClassName` 必须定义于包含派生类定义的作用域中。也允许用其他任意表达式代替基类名称所在的位置。这有时也可能会用得上，例如，当基类定义在另一个模块中的时候：

```
class DerivedClassName(modname.BaseClassName):
```

派生类定义的执行过程与基类相同。当构造类对象时，基类会被记住。此信息将被用来解析属性引用：如果请求的属性在类中找不到，搜索将转往基类中进行查找。如果基类本身也派生自其他某个类，则此规则将被递归地应用。

派生类的实例化没有任何特殊之处：`DerivedClassName()` 会创建该类的一个新实例。方法引用将按以下方式解析：搜索相应的类属性，如有必要将按基类继承链逐步向下查找，如果产生了一个函数对象则方法引用就生效。

派生类可能会重写其基类的方法。因为方法在调用同一对象的其他方法时没有特殊权限，所以调用同一基类中定义的另一方法的基类方法最终可能会调用覆盖它的派生类的方法。（对 C++ 程序员的提示：Python 中所有的方法实际上都是 `virtual` 方法。）

在派生类中的重载方法实际上可能想要扩展而非简单地替换同名的基类方法。有一种方式可以简单地直接调用基类方法：即调用 `BaseClassName.methodname(self, arguments)`。有时这对客户端来说也是有用的。（请注意仅当此基类可在全局作用域中以 `BaseClassName` 的名称被访问时方可使用此方式。）

Python 有两个内置函数可被用于继承机制：

- 使用 `isinstance()` 来检查一个实例的类型：`isinstance(obj, int)` 仅会在 `obj.__class__` 为 `int` 或某个派生自 `int` 的类时为 `True`。
- 使用 `issubclass()` 来检查类的继承关系：`issubclass(bool, int)` 为 `True`，因为 `bool` 是 `int` 的子类。但是，`issubclass(float, int)` 为 `False`，因为 `float` 不是 `int` 的子类。

多重继承

Python 也支持一种多重继承。带有多个基类的类定义语句如下所示：

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

对于多数应用来说，在最简单的情况下，你可以认为搜索从父类所继承属性的操作是深度优先、从左至右的，当层次结构中存在重叠时不会在同一个类中搜索两次。因此，如果某一属性在 `DerivedClassName` 中未找到，则会到 `Base1` 中搜索它，然后（递归地）到 `Base1` 的基类中搜索，如果在那里未找到，再到 `Base2` 中搜索，依此类推。

真实情况比这个更复杂一些；方法解析顺序会动态改变以支持对 `super()` 的协同调用。这种方式在某些其他多重继承型语言中被称为后续方法调用，它比单继承型语言中的 `super` 调用更强大。

动态改变顺序是有必要的，因为所有多重继承的情况都会显示出一个或更多的菱形关联（即至少有一个父类可通过多条路径被最底层类所访问）。例如，所有类都是继承自 `object`，因此任何多重继承的情况都提供了一条以上的路径可以通向 `object`。为了确保基类不会被访问一次以上，动态算法会用一种特殊方式将搜索顺序线性化，保留每个类所指定的从左至右的顺序，只调用每个父类一次，并且保持单调（即一个类可以被子类化而不影响其父类的优先顺序）。总而言之，这些特性使得设计具有多重继承的可靠且可扩展的类成为可能。

私有变量

那种仅限从一个对象内部访问的“私有”实例变量在 Python 中并不存在。但是，大多数 Python 代码都遵循这样一个约定：带有一个下划线的名称（例如 `_spam`）应该被当作是 API 的非公有部分（无论它是函数、方法或是数据成员）。这应当被视为一个实现细节，可能不经通知即加以改变。

由于存在对于类私有成员的有效使用场景（例如避免名称与子类所定义的名称相冲突），因此存在对此种机制的有限支持，称为 *名称改写*。任何形式为 `__spam` 的标识符（至少带有两个前缀下划线，至多一个后缀下划线）的文本将被替换为 `_classname__spam`，其中 `classname` 为去除了前缀下划线的当前类名称。这种改写不考虑标识符的句法位置，只要它出现在类定义内部就会进行。

名称改写有助于让子类重载方法而不破坏类内方法调用。例如：

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update    # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

上面的示例即使在 `MappingSubclass` 引入了一个 `__update` 标识符的情况下也不会出错，因为它会在 `Mapping` 类中被替换为 `_Mapping__update` 而在 `MappingSubclass` 类中被替换为 `_MappingSubclass__update`。

请注意，改写规则的设计主要是为了避免意外冲突；访问或修改被视为私有的变量仍然是可能的。这在特殊情况下甚至会很有用，例如在调试器中。

请注意传递给 `exec()` 或 `eval()` 的代码不会将发起调用类的类名视作当前类；这类似于 `global` 语句的效果，因此这种效果仅限于同时经过字节码编译的代码。同样的限制也适用于 `getattr()`、`setattr()` 和 `delattr()`，以及对于 `__dict__` 的直接引用。

杂项说明

有时会需要使用类似于 Pascal 的“record”或 C 的“struct”这样的数据类型，将一些命名数据项捆绑在一起。这种情况适合定义一个空类：

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

一段需要特定抽象数据类型的 Python 代码往往可以被传入一个模拟了该数据类型的方法的类作为替代。例如，如果你有一个基于文件对象来格式化某些数据的函数，你可以定义一个带有 `read()` 和 `readline()` 方法从字符串缓存获取数据的类，并将其作为参数传入。

实例方法对象也具有属性：`m.__self__` 就是带有 `m()` 方法的实例对象，而 `m.__func__` 则是该方法所对应的函数对象。

迭代器

到目前为止，我们可能已经注意到大多数容器对象都可以使用 `for` 语句：

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

这种访问风格清晰、简洁又方便。迭代器的使用非常普遍并使得 Python 成为一个统一的整体。在幕后，`for` 语句会在容器对象上调用 `iter()`。该函数返回一个定义了 `__next__()` 方法的迭代器对象，此方法将逐一访问容器中的元素。当元素用尽时，`__next__()` 将引发 `StopIteration` 异常来通知终止 `for` 循环。你可以使用 `next()` 内置函数来调用 `__next__()` 方法；这个例子显示了它的运作方式：

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

看过迭代器协议的幕后机制，给你的类添加迭代器行为就很容易了。定义一个 `__iter__()` 方法来返回一个带有 `__next__()` 方法的对象。如果类已定义了 `__next__()`，则 `__iter__()` 可以简单地返回 `self`：

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```



```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

生成器

[生成器](#) 是一个用于创建迭代器的简单而强大的工具。它们的写法类似于标准的函数，但当它们要返回数据时会使用 [yield](#) 语句。每次在生成器上调用 [next\(\)](#) 时，它会从上次离开的位置恢复执行（它会记住上次执行语句时的所有数据值）。一个显示如何非常容易地创建生成器的示例如下：

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

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

可以用生成器来完成的操作同样可以用前一节所描述的基于类的迭代器来完成。但生成器的写法更为紧凑，因为它会自动创建 [__iter__\(\)](#) 和 [__next__\(\)](#) 方法。

另一个关键特性在于局部变量和执行状态会在每次调用之间自动保存。这使得该函数相比使用 `self.index` 和 `self.data` 这种实例变量的方式更易编写且更为清晰。

除了会自动创建方法和保存程序状态，当生成器终结时，它们还会自动引发 [StopIteration](#)。这些特性结合在一起，使得创建迭代器能与编写常规函数一样容易。

生成器表达式

某些简单的生成器可以写成简洁的表达式代码，所用语法类似列表推导式，但外层为圆括号而非方括号。这种表达式被设计用于生成器将立即被外层函数所使用的情况。生成器表达式相比完整的生成器更紧凑但较不灵活，相比等效的列表推导式则更为节省内存。

示例：

```

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

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

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

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

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

```

装包，拆包

#装包

装包就是把未命名的参数放到元组中，把命名参数放到字典中

```

a = 1, 2
print(a)
(1, 2)

```

#拆包将一个结构中的数据拆分为多个单独变量中 *args **kwargs

```

def run1(*args):      # *args 相当于 a, b, c = args
    print(*args)      # 拆包 1, 2, 3
    print(args)       # 未拆包 (1, 2, 3)
    run2(*args)       # 将拆包数据传给run2
    run2(args)        # 将未拆包数据传给run2

```

```

def run2(*args):
    print(args)       # 打印包
    print(*args)      # 打印拆包后的数据

```

```

a, b, c = 1, 2, 3
run1(a, b, c)

```

```

1 2 3
(1, 2, 3)
(1, 2, 3)
1 2 3

```

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

闭包

请大家跟我理解一下，如果在一个函数的内部定义了另一个函数，外部的我们叫他外函数，内部的我们叫他内函数。

闭包：在一个外函数中定义了一个内函数，内函数里运用了外函数的临时变量，并且外函数的返回值是内函数的引用。这样就构成了一个闭包。一般情况下，在我们认知当中，如果一个函数结束，函数的内部所有东西都会释放掉，还给内存，局部变量都会消失。但是闭包是一种特殊情况，如果外函数在结束的时候发现有自己的临时变量将来会在内部函数中用到，就把这个临时变量绑定给了内部函数，然后自己再结束。

#闭包函数的实例

outer是外部函数 a和b都是外函数的临时变量

```
def outer( a ):
    b = 10
    # inner是内函数
    def inner():
        #在内函数中 用到了外函数的临时变量
        print(a+b)
    # 外函数的返回值是内函数的引用
    return inner

if __name__ == '__main__':
    # 在这里我们调用外函数传入参数5
    #此时外函数两个临时变量 a是5 b是10 ，并创建了内函数，然后把内函数的引用返回存给了demo
    # 外函数结束的时候发现内部函数将会用到自己的临时变量，这两个临时变量就不会释放，会绑定给这个内部函数
    demo = outer(5)
    # 我们调用内部函数，看一看内部函数是不是能使用外部函数的临时变量
    # demo存了外函数的返回值，也就是inner函数的引用，这里相当于执行inner函数
    demo() # 15
    demo2 = outer(7)
    demo2()#17
```

*args 和 **kwargs ?

考虑一种情况，我们必须编写多个函数只是为了接受不同数量的参数。

例如以下 `add` Python中的方法：

```
def add(a, b):  
    return a+b  
  
def add(a, b, c, d, e, f):  
    return a+b+c+d+e+f  
  
print(add(5,6))           #output 11  
print(add(5, 6, 1, 5, 2, 2))  #output 21
```

请注意，我们必须编写两个重载函数来添加不同数量的参数。

每次参数数量不同时，我们都需要编写一个与之对应的特定函数。

我们可以解决这个问题的一种方法是使用 [列表](#) 作为参数来接受多个整数，如下所示：

```
def add(lst):  
    return sum(lst)  
  
print(add([5, 6, 1, 5, 2, 2]))  #output 21
```

虽然这解决了问题，但它仍然改变了我们想要使用该功能的方式。

还有另一种方法可以使 Python 中的函数在参数方面更加灵活，那就是使用 `*args` 和 `**kwargs`。

什么是 `*args` 和 `**kwargs`?

`*args` 和 `**kwargs` 是 Python 中的关键字，当定义为参数时，允许函数接受不同数量的参数。

当我们不确定特定函数的参数数量时，它们很有用。

它们都使函数在参数方面具有灵活性，但它们彼此略有不同。让我们看看两者的例子。

`*args` 在 Python 中是一个 **非关键字参数**，它可以接受和存储可变数量的位置参数作为 [元组](#)。

位置参数是一个不跟等号 (=) 和默认值的名称。

```
def add(*args):  
    print(sum(args))  
  
add()           #0-arguments  
add(5, 6, 4)    #3-arguments  
add(1, 1, 1, 1, 1)  #5-arguments
```

`*args` 可以接受可变数量的参数（甚至是 0 个参数），从而使函数更加灵活。

如果我们不使用 `*args`，我们必须为每个函数调用定义单独的函数。

我们可以重命名 *args 吗？

我们可以。这是因为 `args` 在 `*args` 只是一个名字，更重要的是 `*`。

`*` 在 `*args` 是一个解包运算符，将传入的参数转换为 Python 元组。

保持 `*` 操作符，我们可以重命名 `args` 任何名称，如下所示：

```
def add(*integers):  
    print(sum(integers))  
  
add(-1, 7, 2, 6)    #output 14
```

Python中的kwargs

`**kwargs` 在 Python 中是一个 **关键字参数**，它可以接受和存储不同类型的 **关键字参数** 作为 **字典**。

关键字参数（也称为命名参数）后跟一个等号和一个给出其默认值的表达式。

当我们希望函数接受不同数量的参数并且同时也不确定它们的类型时，它很有用。

考虑以下代码，其中一个函数接受两个不同类型的不同关键字参数：

```
def student(name="", roll=-1):  
    print(name)    #output: pencil  
    print(roll)    #output: 77  
  
student(name="pencil", roll=77)
```

在这里，我们可以使用 `**kwargs` 使函数在关键字参数方面更加灵活，如下所示：

```
pythdef student(**kwargs):  
    print(kwargs['name'])    #output: pencil  
    print(kwargs['roll'])    #output: 77  
  
student(name="pencil", roll=77, department="IT")
```

我们已经将一个额外的关键字参数（即部门）传递给 `student` 功能，但代码工作正常。

和 `*args` 一样，在 `**kwargs` 中我们可以重命名 `kwargs`。

重要的部分是 `**`（解包操作符），它应该在名称之前。

装饰器

装饰器是一个在另一个函数周围创建包装器的函数。这个包装器为现有代码添加了一些额外的功能。

装饰器也是一个函数，它是让其他函数在不改变变动的前提下增加额外的功能。

装饰器是一个闭包，把一个函数当作参数返回一个替代版的函数，本质是一个返回函数的函数（即返回值为函数对象）。

python3支持用@符号直接将装饰器应用到函数。

装饰器工作场景：插入日志、性能测试、事务处理等等。

函数被装饰器装饰过后，此函数的属性均已发生变化，如名称变为装饰器的名称。

```
# funcEx.py

def addOne(myFunc):
    def addOneInside(x):
        print("adding One")
        return myFunc(x) + 1
    return addOneInside

def subThree(x):
    return x - 3

result = addOne(subThree)

print(subThree(5))
print(result(5))

# outputs
# 2
# adding One
# 3
```

上面的代码工作如下，

- 函数“subThree”在第 9-10 行定义，它用 3 减去给定的数字。
- 函数'addOne'（第3 行）有一个参数，即myFunc，它表明'addOne'将函数作为输入。由于 subThree 函数只有一个输入参数，因此在函数“addOneInside”中设置了一个参数（第 4 行）；用于 return 语句（第 6 行）。此外，在返回值之前打印“加一”（第 5 行）。
- 在第12 行，addOne 的返回值（即函数'addOneInside'）存储在'result'中。因此，“结果”是一个接受一个输入的函数。
- 最后，在第 13 行和第 14 行打印值。请注意，“加一”由 result(5) 打印，并且值增加 1，即 2 到 3。

二层装饰器，也就是没有装饰器没有参数

```
import time
def decl(func):
    print("开始ing")
    time.sleep(3)

    def wrapper(*args,**kwargs):
        func(*args,**kwargs)
        print("到达装饰器里面")
    return wrapper()

@decl
def f1():
    print("我是f1")
```

运行结果

```
开始ing
我是f1
到达装饰器里面
```

三层装饰器，有参数的装饰器

```
import time
def outer(a):
    def decl(func):
        print("开始ing")
        time.sleep(3)

        def wrapper(*args,**kwargs):
            func(*args,**kwargs)
            print("到达装饰器里面",a)
        return wrapper()
    return decl

@outer(10)
def f1():
    print("我是f1")
```

运行结果

```
开始ing
我是f1
到达装饰器里面 10
```

匿名函数

```

s = lambda a,b:a+b
print(s)
print(s(1,2))

#map+匿名函数
lists = [1,3,5,7,9]
result = map(lambda x:x*x,lists)
print(result)
print(list(result))

#if+匿名函数
func = lambda x:x if x==10 else x+1
print(func(10))
print(func(11))

func = lambda x:print("大于10") if x>10 else print("少于10")
print(func(11))
print(func(2))

#filter+匿名函数
result = list(filter(lambda x:x>5, lists))
print(result)

#reduce +匿名函数
from functools import reduce
# reduce 函数可以按照给定的方法把输入参数中上序列缩减为单个的值，具体的做法如下：
# 首先从序列中去除头两个元素并把它传递到那个二元函数中去，求出一个值，再把这个加到序列中循环求下一个值，
直到最后一个值 。
result = reduce(lambda x,y:x*y, [1,2,3,4,5] )#(((1*2)*3)*4)*5
print(result)

```

运行结果

```

<function <lambda> at 0x00000285FDBC3E18>
3
<map object at 0x00000285FDF312B0>
[1, 9, 25, 49, 81]
10
12
大于10
None
少于10
None
[7, 9]
120

```

捕捉异常错误

句法错误

句法错误又称解析错误，是学习 Python 时最常见的错误：

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

解析器会复现出现句法错误的代码行，并用小“箭头”指向行里检测到的第一个错误。错误是由箭头 上方的 token 触发的（至少是在这里检测出的）：本例中，在 `print()` 函数中检测到错误，因为，在它前面缺少冒号（`:`）。错误信息还输出文件名与行号，在使用脚本文件时，就可以知道去哪里查错。

异常

即使语句或表达式使用了正确的语法，执行时仍可能触发错误。执行时检测到的错误称为 *异常*，异常不一定导致严重的后果：很快我们就能学会如何处理 Python 的异常。大多数异常不会被程序处理，而是显示下列错误信息：

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

错误信息的最后一行说明程序遇到了什么类型的错误。异常有不同的类型，而类型名称会作为错误信息的一部分中打印出来：上述示例中的异常类型依次是：`ZeroDivisionError`，`NameError` 和 `TypeError`。作为异常类型打印的字符串是发生的内置异常的名称。对于所有内置异常都是如此，但对于用户定义的异常则不一定如此（虽然这种规范很有用）。标准的异常类型是内置的标识符（不是保留关键字）。

此行其余部分根据异常类型，结合出错原因，说明错误细节。

错误信息开头用堆栈回溯形式展示发生异常的语境。一般会列出源代码行的堆栈回溯；但不会显示从标准输入读取的行。

[内置异常](#) 列出了内置异常及其含义。

异常的处理

可以编写程序处理选定的异常。下例会要求用户一直输入内容，直到输入有效的整数，但允许用户中断程序（使用 Control-C 或操作系统支持的其他操作）；注意，用户中断程序会触发 `KeyboardInterrupt` 异常。

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

`try` 语句的工作原理如下：

- 首先，执行 `try` 子句（`try` 和 `except` 关键字之间的（多行）语句）。
- 如果没有触发异常，则跳过 `except` 子句，`try` 语句执行完毕。
- 如果在执行 `try` 子句时发生了异常，则跳过该子句中剩下的部分。如果异常的类型与 `except` 关键字后指定的异常相匹配，则会执行 `except` 子句，然后跳到 `try/except` 代码块之后继续执行。
- 如果发生的异常与 `except` 子句中指定的异常不匹配，则它会被传递到外部的 `try` 语句中；如果没有找到处理程序，则它是一个 *未处理异常* 且执行将终止并输出如上所示的消息。

`try` 语句可以有多个 `except` 子句 来为不同的异常指定处理程序。但最多只有一个处理程序会被执行。处理程序只处理对应的 `try` 子句中发生的异常，而不处理同一 `try` 语句内其他处理程序中的异常。`except` 子句 可以用带圆括号的元组来指定多个异常，例如：

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

如果发生的异常与 `except` 子句中的类是同一个类或是它的基类时，则该类与该异常相兼容（反之则不成立 --- 列出派生类的 `except` 子句 与基类不兼容）。例如，下面的代码将依次打印 B, C, D:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

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

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

请注意如果颠倒 `except` 子句的顺序（把 `except B` 放在最前），则会输出 B, B, B --- 即触发了第一个匹配的 `except` 子句。

所有异常都继承自 `BaseException`，因此它可被用作通配符。但这种用法要非常谨慎小心，因为它很容易掩盖真正的编程错误！ 它还可被用于打印错误消息然后重新引发异常（允许调用者再来处理该异常）：

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except BaseException as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

可以选择让最后一个 `except` 子句省略异常名称，但在此之后异常值必须从 `sys.exc_info()[1]` 获取。

`try ... except` 语句具有可选的 `else` 子句，该子句如果存在，它必须放在所有 `except` 子句之后。它适用于 `try` 子句没有引发异常但又必须要执行的代码。例如：

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

使用 `else` 子句比向 `try` 子句添加额外的代码要好，可以避免意外捕获非 `try ... except` 语句保护的代码触发的异常。

发生异常时，它可能具有关联值，即异常 参数。是否需要参数，以及参数的类型取决于异常的类型。

`except` 子句可以在异常名称后面指定一个变量。这个变量会绑定到一个异常实例并将参数存储在 `instance.args` 中。为了方便起见，该异常实例定义了 `__str__()` 以便能直接打印参数而无需引用 `.args`。也可以在引发异常之前先实例化一个异常并根据需要向其添加任何属性。：

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
```

```

...     print(type(inst))      # the exception instance
...     print(inst.args)      # arguments stored in .args
...     print(inst)           # __str__ allows args to be printed directly,
...                             # but may be overridden in exception subclasses
...     x, y = inst.args       # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

如果异常有参数，则它们将作为未处理异常的消息的最后一部分（'详细信息'）打印。

异常处理程序不仅会处理在 *try* 子句中发生的异常，还会处理在 *try* 子句中调用（包括间接调用）的函数。例如：

```

>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero

```

触发异常

`raise` 语句支持强制触发指定的异常。例如：

```

>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere

```

`raise` 唯一的参数就是要触发的异常。这个参数必须是异常实例或异常类（派生自 `Exception` 类）。如果传递的是异常类，将通过调用没有参数的构造函数来隐式实例化：

```

raise ValueError # shorthand for 'raise ValueError()'

```

如果只想判断是否触发了异常，但并不打算处理该异常，则可以使用更简单的 `raise` 语句重新触发异常：

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

异常链

`raise` 语句支持可选的 `from` 子句, 该子句用于启用链式异常。例如:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

转换异常时, 这种方式很有用。例如:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

异常链会在 `except` 或 `finally` 子句内部引发异常时自动生成。这可以通过使用 `from None` 这样的写法来禁用:

```
try:
    open('database.sqlite')
except OSError:
    raise RuntimeError from None
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

异常链机制详见 [内置异常](#)。

用户自定义异常

程序可以通过创建新的异常类命名自己的异常（Python 类的内容详见 [类](#)）。不论是以直接还是间接的方式，异常都应从 [Exception](#) 类派生。

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

大多数异常命名都以 “Error” 结尾，类似标准异常的命名。

许多标准模块都需要自定义异常，以报告由其定义的函数中出现的错误。有关类的说明，详见 [类](#)。

定义清理操作

[try](#) 语句还有一个可选子句，用于定义在所有情况下都必须执行的清理操作。例如：

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

如果存在 [finally](#) 子句，则 [finally](#) 子句是 [try](#) 语句结束前执行的最后一项任务。不论 [try](#) 语句是否触发异常，都会执行 [finally](#) 子句。以下内容介绍了几种比较复杂的触发异常情景：

- 如果执行 [try](#) 子句期间触发了某个异常，则某个 [except](#) 子句应处理该异常。如果该异常没有 [except](#) 子句处理，在 [finally](#) 子句执行后会被重新触发。
- [except](#) 或 [else](#) 子句执行期间也会触发异常。同样，该异常会在 [finally](#) 子句执行之后被重新触发。
- 如果 [finally](#) 子句中包含 [break](#)、[continue](#) 或 [return](#) 等语句，异常将不会被重新引发。
- 如果执行 [try](#) 语句时遇到 [break](#)、[continue](#) 或 [return](#) 语句，则 [finally](#) 子句在执行

`break`、`continue` 或 `return` 语句之前执行。

- 如果 `finally` 子句中包含 `return` 语句，则返回值来自 `finally` 子句的某个 `return` 语句的返回值，而不是来自 `try` 子句的 `return` 语句的返回值。

例如：

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

这是一个比较复杂的例子：

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

如上所示，任何情况下都会执行 `finally` 子句。`except` 子句不处理两个字符串相除触发的 `TypeError`，因此会在 `finally` 子句执行后被重新触发。

在实际应用程序中，`finally` 子句对于释放外部资源（例如文件或者网络连接）非常有用，无论是否成功使用资源。

预定义的清理操作

某些对象定义了不需要该对象时要执行的标准清理操作。无论使用该对象的操作是否成功，都会执行清理操作。比如，下例要打开一个文件，并输出文件内容：

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

这个代码的问题在于，执行完代码后，文件在一段不确定的时间内处于打开状态。在简单脚本中这没有问题，但对于较大的应用程序来说可能会出问题。`with` 语句支持以及时、正确的清理的方式使用文件对象：

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

语句执行完毕后，即使在处理行时遇到问题，都会关闭文件 *f*。和文件一样，支持预定义清理操作的对象会在文档中指出这一点。

```
try:
    open('cc')
except Exception as err:
    print(err)
```

运行结果

```
[Errno 2] No such file or directory: 'cc'
```

虚拟环境和包

概述

Python应用程序通常会使用不在标准库内的软件包和模块。应用程序有时需要特定版本的库，因为应用程序可能需要修复特定的错误，或者可以使用库的过时版本的接口编写应用程序。

这意味着一个Python安装可能无法满足每个应用程序的要求。如果应用程序A需要特定模块的1.0版本但应用程序B需要2.0版本，则需求存在冲突，安装版本1.0或2.0将导致某一个应用程序无法运行。

这个问题的解决方案是创建一个 [virtual environment](#)，一个目录树，其中安装有特定Python版本，以及许多其他包。

然后，不同的应用将可以使用不同的虚拟环境。要解决先前需求相冲突的例子，应用程序 A 可以拥有自己的 安装了 1.0 版本的虚拟环境，而应用程序 B 则拥有安装了 2.0 版本的另一个虚拟环境。如果应用程序 B 要求将某个库升级到 3.0 版本，也不会影响应用程序 A 的环境。

创建虚拟环境

用于创建和管理虚拟环境的模块称为 `venv`。`venv` 通常会安装你可用的最新版本的 Python。如果您的系统上有多个版本的 Python，您可以通过运行 `python3` 或您想要的任何版本来选择特定的 Python 版本。

要创建虚拟环境，请确定要放置它的目录，并将 `venv` 模块作为脚本运行目录路径：

```
python3 -m venv tutorial-env
```

这将创建 `tutorial-env` 目录，如果它不存在的话，并在其中创建包含 Python 解释器副本和各种支持文件的目录。

虚拟环境的常用目录位置是 `.venv`。这个名称通常会令该目录在你的终端中保持隐藏，从而避免需要对所在目录进行额外解释的一般名称。它还能防止与某些工具所支持的 `.env` 环境变量定义文件发生冲突。

创建虚拟环境后，您可以激活它。

在 Windows 上，运行：

```
tutorial-env\Scripts\activate.bat
```

在 Unix 或 MacOS 上，运行：

```
source tutorial-env/bin/activate
```

（这个脚本是为 bash shell 编写的。如果你使用 **csh** 或 **fish** shell，你应该改用 `activate.csh` 或 `activate.fish` 脚本。）

激活虚拟环境将改变你所用终端的提示符，以显示你正在使用的虚拟环境，并修改环境以使 `python` 命令所运行的将是已安装的特定 Python 版本。例如：

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May  6 2016, 10:59:36)
...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
 '~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

使用 pip 管理包

你可以使用一个名为 **pip** 的程序来安装、升级和移除软件包。默认情况下 `pip` 将从 Python Package Index <https://pypi.org> 安装软件包。你可以在你的 web 浏览器中查看 Python Package Index。

`pip` 有许多子命令：`"install"`、`"uninstall"`、`"freeze"` 等等。（请在 [安装 Python 模块](#) 指南页查看完整的 `pip` 文档。）

您可以通过指定包的名称来安装最新版本的包：

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

您还可以通过提供包名称后跟 `==` 和版本号来安装特定版本的包：

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

如果你重新运行这个命令，`pip` 会注意到已经安装了所请求的版本并且什么都不做。您可以提供不同的版本号来获取该版本，或者您可以运行 `pip install --upgrade` 将软件包升级到最新版本：

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
  Uninstalling requests-2.6.0:
    Successfully uninstalled requests-2.6.0
  Successfully installed requests-2.7.0
```

`pip uninstall` 后跟一个或多个包名称将从虚拟环境中删除包。

`pip show` 将显示有关特定包的信息：

```
(tutorial-env) $ pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`pip list` 将显示虚拟环境中安装的所有软件包：

```
(tutorial-env) $ pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`pip freeze` 将生成一个类似的已安装包列表，但输出使用 `pip install` 期望的格式。一个常见的约定是将此列表放在 `requirements.txt` 文件中：

```
(tutorial-env) $ pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

然后将 `requirements.txt` 提交给版本控制并作为应用程序的一部分提供。然后用户可以使用 `install -r` 安装所有必需的包：

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
...
Installing collected packages: novas, numpy, requests
Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```