Python 学习笔记

学习内容：http://www.liaoxuefeng.com

目录

Python 基础

基础知识

以#开头的语句是注释，注释是给人看的，可以是任意内容，解释器会忽略掉注释,python注释用的是符号#，但只能注释一行，如果要写一大片文字，最好还是使用三个单引号进行注释。其他每一行都是一个语句，当语句以冒号:结尾时，缩进的语句视为代码块。

缩进有利有弊。好处是强迫你写出格式化的代码，但没有规定缩进是几个空格还是 Tab。按照约定俗成的管理，应该始终坚持使用 4 个空格的缩进。

缩进的另一个好处是强迫你写出缩进较少的代码，你会倾向于把一段很长的代码拆分成若干函数，从而得到缩进较少的代码。

缩进的坏处就是“复制－粘贴”功能失效了，这是最坑爹的地方。当你重构代码时，粘贴过去的代码必须重新检查缩进是否正确。此外，IDE 很难像格式化 Java 代码那样格式化 Python 代码。

最后，请务必注意，Python 程序是大小写敏感的，如果写错了大小写，程序会报错。

字符串

字符串是以单引号'或双引号"括起来的任意文本，比如'abc'，"xyz"等等。请注意，''或""本身只是一种表示方式，不是字符串的一部分，因此，字符串'abc'只有 a，b，c 这 3 个字符。如果'本身也是一个字符，那就可以用""括起来，比如"I'm OK"包含的字符是 I，'，m，空格，O，K 这 6 个字符。

如果字符串内部既包含'又包含"怎么办？可以用转义字符\来标识，比如：

```
'I\'m \"OK\"!'
```

表示的字符串内容是：

```
I'm "OK"!
```

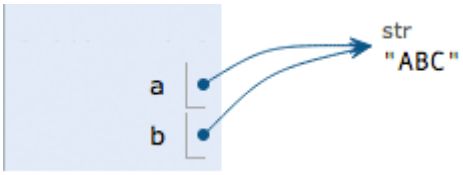
转义字符\可以转义很多字符，比如\n 表示换行，\t 表示制表符，字符\本身也要转义，所以\\表示的字符就是\。

如果字符串里面有很多字符都需要转义，就需要加很多\，为了简化，Python 还允许用 r'' 表示'' 内部的字符串默认不转义，可以自己试试：

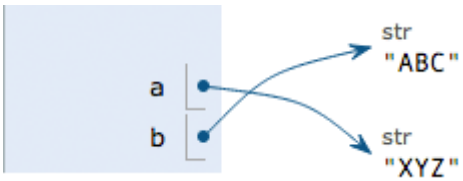
```
>>> print('\\t\\')
\       \
>>> print(r'\\t\\')
\\t\\
```

执行 a = 'ABC'，解释器创建了字符串'ABC'和变量 a，并把 a 指向'ABC'：

执行 b = a，解释器创建了变量 b，并把 b 指向 a 指向的字符串'ABC'：



执行 a = 'XYZ'，解释器创建了字符串'XYZ'，并把 a 的指向改为'XYZ'，但 b 并没有更改：



所以，最后打印变量 b 的结果自然是'ABC'了。

字符编码

现代操作系统和大多数编程语言都直接支持 Unicode。

现在，捋一捋 ASCII 编码和 Unicode 编码的区别：ASCII 编码是 1 个字节，而 Unicode 编码通常是 2 个字节。

新的问题又出现了：如果统一成 Unicode 编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用 Unicode 编码比 ASCII 编码需要多一倍的存储空间，在存储和传输上就十分不划算。

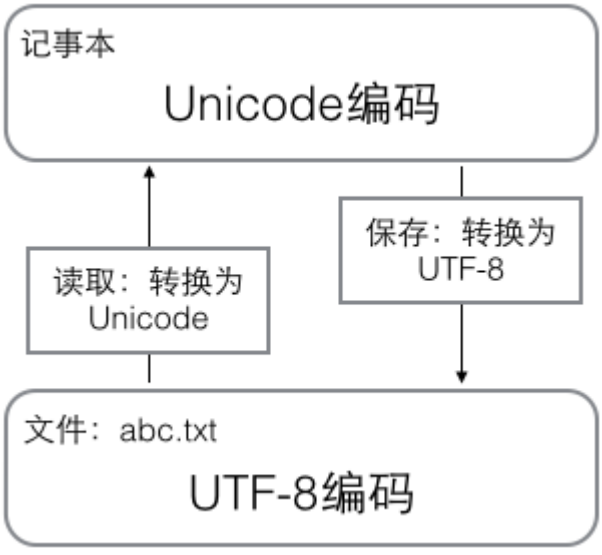
所以，本着节约的精神，又出现了把 Unicode 编码转化为“可变长编码”的 UTF-8 编码。UTF-8 编码把一个 Unicode 字符根据不同的数字大小编码成 1-6 个字节，常用的英文字母被编码成 1 个字节，汉字通常是 3 个字节，只有很生僻的字符才会被编码成 4-6 个字节。如果你要传输的文本包含大量英文字符，用 UTF-8 编码就能节省空间：

字符	ASCII	Unicode	UTF-8
A	01000001	00000000 01000001	01000001
中	x	01001110 00101101	11100100 10111000 10101101

搞清楚了 ASCII、Unicode 和 UTF-8 的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

在计算机内存中，统一使用 Unicode 编码，当需要保存到硬盘或者需要传输的时候，就转换为 UTF-8 编码。

用记事本编辑的时候，从文件读取的 UTF-8 字符被转换为 Unicode 字符到内存里，编辑完成后，保存的时候再把 Unicode 转换为 UTF-8 保存到文件：



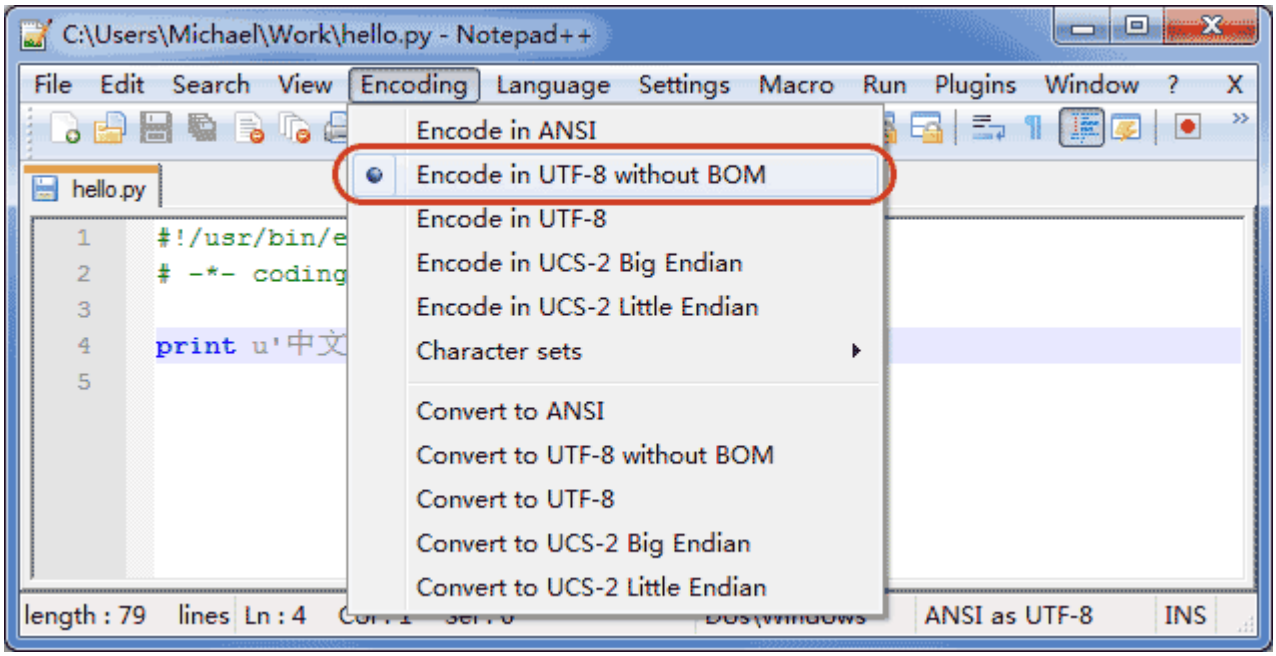
由于 Python 源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为 UTF-8 编码。当 Python 解释器读取源代码时，为了让它按 UTF-8 编码读取，我们通常在文件开头写上这两行：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

第一行注释是为了告诉 Linux/OS X 系统，这是一个 Python 可执行程序，Windows 系统会忽略这个注释；

第二行注释是为了告诉 Python 解释器，按照 UTF-8 编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

申明了 UTF-8 编码并不意味着你的.py 文件就是 UTF-8 编码的，必须并且要确保文本编辑器正在使用 UTF-8 without BOM 编码：



如果.py 文件本身使用 UTF-8 编码，并且也申明了# -*- coding: utf-8 -*-，打开命令提示符测试就可以正常显示中文。

格式化

在 Python 中，采用的格式化方式和 C 语言是一致的，用%实现，举例如下：

常见的占位符有：

%d	整数
%f	浮点数
%s	字符串

%x 十六进制整数

其中，格式化整数和浮点数还可以指定是否补 0 和整数与小数的位数：

```
>>> '%2d-%02d' % (3, 1)
' 3-01'
>>> '%.2f' % 3.1415926
'3.14'
```

如果你不太确定应该用什么，%s 永远起作用，它会把任何数据类型转换为字符串：

```
>>> 'Age: %s. Gender: %s' % (25, True)
'Age: 25. Gender: True'
```

List 列表

Python 内置的一种数据类型是列表：list。list 是一种有序的集合，可以随时添加和删除其中的元素。

比如，列出班里所有同学的名字，就可以用一个 list 表示：

```
>>> classmates = ['Michael', 'Bob', 'Tracy']
>>> classmates
['Michael', 'Bob', 'Tracy']
```

变量 classmates 就是一个 list。**用 len() 函数可以获得 list 元素的个数**：
用索引来访问 list 中每一个位置的元素，**记得索引是从 0 开始的**：
当索引超出了范围时，Python 会报一个 IndexError 错误，所以，要确保索引不要越界，记得**最后一个元素的索引是 len(classmates) - 1**。

如果要取最后一个元素，除了计算索引位置外，还**可以用-1 做索引，直接获取最后一个元素**：

```
>>> classmates[-1]
'Tracy'
```

以此类推，可以获取倒数第 2 个、倒数第 3 个：

```
>>> classmates[-2]
```

```
'Bob'
>>> classmates[-3]
'Michael'
```

list 是一个可变的有序表，所以，可以往 list 中追加元素到末尾：

```
>>> classmates.append('Adam')
>>> classmates
['Michael', 'Bob', 'Tracy', 'Adam']
```

也可以把元素插入到指定的位置，比如索引号为 1 的位置：

```
>>> classmates.insert(1, 'Jack')
```

要删除指定位置的元素，用 pop(i)方法，其中 i 是索引位置：

```
>>> classmates.pop(1)
```

要把某个元素替换成别的元素，可以直接赋值给对应的索引位置：

```
>>> classmates[1] = 'Sarah'
```

list 里面的元素的数据类型也可以不同，比如：

```
>>> L = ['Apple', 123, True]
```

list 元素也可以是另一个 list，比如：

```
>>> s = ['python', 'java', ['asp', 'php'], 'scheme']
```

要注意 s 只有 4 个元素，其中 s[2]又是一个 list，如果拆开写就更容易理解了：

```
>>> p = ['asp', 'php']
>>> s = ['python', 'java', p, 'scheme']
```

要拿到'php'可以写 p[1]或者 s[2][1]，因此 s 可以看成是一个二维数组，类似的还有三维、四维……数组，不过很少用到。

如果一个 list 中一个元素也没有，就是**一个空的 list，它的长度为 0**：

```
>>> L = []
```

Tuple 元组

另一种有序列表叫元组：tuple。tuple 和 list 非常类似，但是 **tuple 一旦初始化就不能修改**，比如同样是列出同学的名字：

```
>>> classmates = ('Michael', 'Bob', 'Tracy')
```

现在，classmates 这个 tuple 不能变了，它也**没有 append()，insert()这样的方法**。其他获取元素的方法和 list 是一样的，你可以正常地使用 classmates[0]，classmates[-1]，但不能赋值成另外的元素。

不可变的 tuple 有什么意义？因为 tuple 不可变，所以代码更安全。如果可能，能用 tuple 代替 list 就尽量用 tuple。

条件判断

根据 Python 的缩进规则，如果 if 语句判断是 True，就把缩进的两行 print 语句执行了，否则，什么也不做。

也可以给 if 添加一个 else 语句，意思是，如果 if 判断是 False，不要执行 if 的内容，去把 else 执行了：

```
age = 3
if age >= 18:
    print('your age is', age)
    print('adult')
else:
    print('your age is', age)
    print('teenager')
```

注意不要少写了冒号:。

elif 是 else if 的缩写，完全可以有多个 elif，所以 if 语句的完整形式就是：

```
if <条件判断 1>:
    <执行 1>
```

```
elif <条件判断 2>:
    <执行 2>
elif <条件判断 3>:
    <执行 3>
else:
    <执行 4>
```

循环

Python 的循环有两种，**一种是 for...in 循环，依次把 list 或 tuple 中的每个元素迭代出来**，看例子：

```
names = ['Michael', 'Bob', 'Tracy']
for name in names:
    print(name)
```

如果要计算 1-100 的整数之和，从 1 写到 100 有点困难，幸好 Python 提供一个 range()函数，可以生成一个整数序列

range(101)就可以生成 0-100 的整数序列，计算如下：

```
sum = 0
for x in range(101):
    sum = sum + x
print(sum)
```

第二种循环是 while 循环，只要条件满足，就不断循环，条件不满足时退出循环。

Dict 字典

Python 内置了字典：dict 的支持，dict 全称 dictionary，在其他语言中也称为 map，使用键-值（key-value）存储，具有极快的查找速度。

用 Python 写一个 dict 如下：

```
>>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
>>> d['Michael']
95
```

由于一个 key 只能对应一个 value，所以，多次对一个 key 放入 value，后面的值会把前面的值冲掉。

要避免 key 不存在的错误，有两种办法，一是通过 in 判断 key 是否存在：

```
>>> 'Thomas' in d
False
```

二是通过 dict 提供的 get 方法，如果 key 不存在，可以返回 None，或者自己指定的 value：

```
>>> d.get('Thomas')
>>> d.get('Thomas', -1)
-1
```

注意：返回 None 的时候 Python 的交互式命令行不显示结果。

要删除一个 key，用 pop(key)方法，对应的 value 也会从 dict 中删除。

dict 可以用在需要高速查找的很多地方，在 Python 代码中几乎无处不在，正确使用 dict 非常重要，需要牢记的第一条就是 **dict 的 key 必须是不可变对象。**

这是因为 dict 根据 key 来计算 value 的存储位置，如果每次计算相同的 key 得出的结果不同，那 dict 内部就完全混乱了。这个通过 key 计算位置的算法称为哈希算法（Hash）。

要保证 hash 的正确性，作为 key 的对象就不能变。在 Python 中，字符串、整数等都是不可变的，因此，可以放心地作为 key。而 list 是可变的，就不能作为 key：

Set 集合

set 和 dict 类似，也是一组 key 的集合，但不存储 value。由于 key 不能重复，所以，在 set 中，没有重复的 key。

要创建一个 set，需要提供一个 list 作为输入集合：

重复元素在 set 中自动被过滤：

```
>>> s = set([1, 1, 2, 2, 3, 3])
>>> s
{1, 2, 3}
```

通过 add(key)方法可以添加元素到 set 中，可以重复添加，但不会有效果：

```
>>> s.add(4)
>>> s
```

```
{1, 2, 3, 4}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
```

通过 remove(key)方法可以删除元素：

```
>>> s.remove(4)
>>> s
{1, 2, 3}
```

set 可以看成数学意义上的无序和无重复元素的集合，因此，**两个 set 可以做数学意义上的交集、并集等操作：**

```
>>> s1 = set([1, 2, 3])
>>> s2 = set([2, 3, 4])
>>> s1 & s2
{2, 3}
>>> s1 | s2
{1, 2, 3, 4}
```

函数

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”：

```
>>> a = abs # 变量 a 指向 abs 函数
>>> a(-1) # 所以也可以通过 a 调用 abs 函数
1
```

定义函数

在 Python 中，定义一个函数要使用 def 语句，依次写出函数名、括号、括号中的参数和冒号:，然后，在缩进块中编写函数体，函数的返回值用 return 语句返回。

我们以自定义一个求绝对值的 my_abs 函数为例：

```
def my_abs(x):
```

```
    if x >= 0:
        return x
    else:
        return -x
```

如果没有 return 语句，函数执行完毕后也会返回结果，只是结果为 None。

return None 可以简写为 return。

如果你已经把 my_abs()的函数定义保存为 abstest.py 文件了，那么，可以在该文件的当前目录下启动 Python 解释器，用 from abstest import my_abs 来导入 my_abs()函数，注意 abstest 是文件名（不含.py 扩展名）。

空函数

如果想定义一个什么事也不做的空函数，可以用 pass 语句：

```
def nop():
    pass
```

pass 语句什么都不做，那有什么用？实际上 pass 可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个 pass，让代码能运行起来。

pass 还可以用在其他语句里，比如：

```
if age >= 18:
    pass
```

缺少了 pass，代码运行就会有语法错误。

返回多个值

函数可以返回多个值吗？答案是肯定的。

比如在游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的新的坐标：

```
import math

def move(x, y, step, angle=0):
```

```
    nx = x + step * math.cos(angle)
    ny = y - step * math.sin(angle)
    return nx, ny
```

import math 语句表示导入 math 包，并允许后续代码引用 math 包里的 sin、cos 等函数。

然后，我们就可以同时获得返回值：

```
>>> x, y = move(100, 100, 60, math.pi / 6)
>>> print(x, y)
151.96152422706632 70.0
```

但其实这只是一种假象，Python 函数返回的仍然是单一值：

```
>>> r = move(100, 100, 60, math.pi / 6)
>>> print(r)
(151.96152422706632, 70.0)
```

原来返回值是一个 tuple！但是，在语法上，返回一个 tuple 可以省略括号，而多个变量可以同时接收一个 tuple，按位置赋给对应的值，所以，Python 的函数返回多值其实就是返回一个 tuple，但写起来更方便。

函数的参数

Python 的函数定义非常简单，但灵活度却非常大。**除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数**，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

位置参数

我们先写一个计算 x^2 的函数：

```
def power(x):
    return x * x
```

对于 power(x) 函数，参数 x 就是一个位置参数。

当我们调用 power 函数时，必须传入有且仅有的一个参数 x：

```
>>> power(5)
25
>>> power(15)
225
```

现在，如果我们要计算 x^3 怎么办？可以再定义一个 power3 函数，但是如果要计算 x^4、x^5……怎么办？我们不可能定义无限多个函数。

你也许想到了，可以把 power(x) 修改为 power(x, n)，用来计算 x^n，说干就干：

```
def power(x, n):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

对于这个修改后的 power(x, n) 函数，可以计算任意 n 次方：

```
>>> power(5, 2)
25
>>> power(5, 3)
125
```

修改后的 power(x, n) 函数有两个参数：x 和 n，这两个参数都是位置参数，调用函数时，传入的两个值按照位置顺序依次赋给参数 x 和 n。

默认参数

新的 power(x, n) 函数定义没有问题，但是，旧的调用代码失败了，原因是我们增加了一个参数，导致旧的代码因为缺少一个参数而无法正常调用：

```
>>> power(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() missing 1 required positional argument: 'n'
```

Python 的错误信息很明确：调用函数 power() 缺少了一个位置参数 n。

这个时候，默认参数就排上用场了。由于我们经常计算 x^2，所以，完全可以把第二个参数 n 的默认值设定为 2：

```
def power(x, n=2):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

这样，当我们调用 power(5)时，相当于调用 power(5, 2)：

```
>>> power(5)
25
>>> power(5, 2)
25
```

而对于 n > 2 的其他情况，就必须明确地传入 n，比如 power(5, 3)。

从上面的例子可以看出，默认参数可以简化函数的调用。设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则 Python 的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；

二是如何设置默认参数。

当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

使用默认参数有什么好处？最大的好处是能降低调用函数的难度。

举个例子，我们写个一年级小学生注册的函数，需要传入 name 和 gender 两个参数：

```
def enroll(name, gender):
    print('name:', name)
    print('gender:', gender)
```

这样，调用 enroll()函数只需要传入两个参数：

```
>>> enroll('Sarah', 'F')
name: Sarah
```

```
gender: F
```

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。

我们可以把年龄和城市设为默认参数：

```
def enroll(name, gender, age=6, city='Beijing'):
    print('name:', name)
    print('gender:', gender)
    print('age:', age)
    print('city:', city)
```

这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：

```
>>> enroll('Sarah', 'F')
name: Sarah
gender: F
age: 6
city: Beijing
```

只有与默认参数不符的学生才需要提供额外的信息：

```
enroll('Bob', 'M', 7)
enroll('Adam', 'M', city='Tianjin')
```

可见，默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用 enroll('Bob', 'M', 7)，意思是，除了 name，gender 这两个参数外，最后 1 个参数应用在参数 age 上，city 参数由于没有提供，仍然使用默认值。

也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用 enroll('Adam', 'M', city='Tianjin')，意思是，city 参数用传进去的值，其他默认参数继续使用默认值。

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个 list，添加一个 END 再返回：

```
def add_end(L=[]):
```

```
    L.append('END')
    return L
```

当你正常调用时，结果似乎不错：

```
>>> add_end([1, 2, 3])
[1, 2, 3, 'END']
>>> add_end(['x', 'y', 'z'])
['x', 'y', 'z', 'END']
```

当你使用默认参数调用时，一开始结果也是对的：

```
>>> add_end()
['END']
```

但是，再次调用 add_end()时，结果就不对了：

```
>>> add_end()
['END', 'END']
>>> add_end()
['END', 'END', 'END']
```

很多初学者很疑惑，默认参数是[]，但是函数似乎每次都“记住了”上次添加了'END'后的list。

原因解释如下：

Python 函数在定义的时候，默认参数 L 的值就被计算出来了，即[]，因为默认参数 L 也是一个变量，它指向对象[]，每次调用该函数，如果改变了 L 的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的[]了。

所以，定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用 None 这个不变对象来实现：

```
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L
```

现在，无论调用多少次，都不会有问题：

```
>>> add_end()
['END']
>>> add_end()
['END']
```

为什么要设计 str、None 这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象。

可变参数

在 Python 函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是 1 个、2 个到任意个，还可以是 0 个。

我们以数学题为例子，给定一组数字 a，b，c……，请计算 $a^2 + b^2 + c^2 + \ldots\ldots$。

要定义出这个函数，我们必须确定输入的参数。由于参数个数不确定，我们首先想到可以把 a，b，c……作为一个 list 或 tuple 传进来，这样，函数可以定义如下：

```
def calc(numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

但是调用的时候，需要先组装出一个 list 或 tuple：

```
>>> calc([1, 2, 3])
14
>>> calc((1, 3, 5, 7))
84
```

如果利用可变参数，调用函数的方式可以简化成这样：

```
>>> calc(1, 2, 3)
14
>>> calc(1, 3, 5, 7)
```

```
84
```

所以，我们把函数的参数改为可变参数：

```
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

定义可变参数和定义一个 list 或 tuple 参数相比，仅仅在参数前面加了一个*号。在函数内部，参数 numbers 接收到的是一个 tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括 0 个参数：

```
>>> calc(1, 2)
5
>>> calc()
0
```

如果已经有一个 list 或者 tuple，要调用一个可变参数怎么办？可以这样做：

```
>>> nums = [1, 2, 3]
>>> calc(nums[0], nums[1], nums[2])
14
```

这种写法当然是可行的，问题是太繁琐，所以 Python 允许你在 list 或 tuple 前面加一个*号，把 list 或 tuple 的元素变成可变参数传进去：

```
>>> nums = [1, 2, 3]
>>> calc(*nums)
14
```

*nums 表示把 nums 这个 list 的所有元素作为可变参数传进去。这种写法相当有用，而且很常见。

关键字参数

可变参数允许你传入 0 个或任意个参数，这些可变参数在函数调用时自动组装为一个 tuple。而关键字参数允许你传入 0 个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个 dict。请看示例：

```
def person(name, age, **kw):
    print('name:', name, 'age:', age, 'other:', kw)
```

函数 person 除了必选参数 name 和 age 外，还接受关键字参数 kw。在调用该函数时，可以只传入必选参数：

```
>>> person('Michael', 30)
name: Michael age: 30 other: {}
```

也可以传入任意个数的关键字参数：

```
>>> person('Bob', 35, city='Beijing')
name: Bob age: 35 other: {'city': 'Beijing'}
>>> person('Adam', 45, gender='M', job='Engineer')
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}
```

关键字参数有什么用？它可以扩展函数的功能。比如，在 person 函数里，我们保证能接收到 name 和 age 这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

和可变参数类似，也可以先组装出一个 dict，然后，把该 dict 转换为关键字参数传进去：

```
>>> extra = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, city=extra['city'], job=extra['job'])
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

当然，上面复杂的调用可以用简化的写法：

```
>>> extra = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, **extra)
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

extra 表示把 extra 这个 dict 的所有 key-value 用关键字参数传入到函数的kw 参数，kw 将获得一个 dict，注意 kw 获得的 dict 是 extra 的一份拷贝，对 kw 的改动不会影响到函数外的 extra。

命名关键字参数

对于关键字参数，函数的调用者可以传入任意不受限制的关键字参数。至于到底传入了哪些，就需要在函数内部通过 kw 检查。

仍以 person()函数为例，我们希望检查是否有 city 和 job 参数：

```
def person(name, age, **kw):
    if 'city' in kw:
        # 有 city 参数
        pass
    if 'job' in kw:
        # 有 job 参数
        pass
    print('name:', name, 'age:', age, 'other:', kw)
```

但是调用者仍可以传入不受限制的关键字参数：

```
>>> person('Jack', 24, city='Beijing', addr='Chaoyang', zipcode=123456)
```

如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收 city 和 job 作为关键字参数。这种方式定义的函数如下：

```
def person(name, age, *, city, job):
    print(name, age, city, job)
```

和关键字参数**kw 不同，命名关键字参数需要一个特殊分隔符*，*后面的参数被视为命名关键字参数。

调用方式如下：

```
>>> person('Jack', 24, city='Beijing', job='Engineer')
Jack 24 Beijing Engineer
```

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错：

```
>>> person('Jack', 24, 'Beijing', 'Engineer')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: person() takes 2 positional arguments but 4 were given
```

由于调用时缺少参数名 city 和 job，Python 解释器把这 4 个参数均视为位置参数，但 person()函数仅接受 2 个位置参数。

命名关键字参数可以有缺省值，从而简化调用：

```
def person(name, age, *, city='Beijing', job):
    print(name, age, city, job)
```

由于命名关键字参数 city 具有默认值，调用时，可不传入 city 参数：

```
>>> person('Jack', 24, job='Engineer')
Jack 24 Beijing Engineer
```

使用命名关键字参数时，要特别注意，*不是参数，而是特殊分隔符。如果缺少*，Python 解释器将无法识别位置参数和命名关键字参数：

```
def person(name, age, city, job):
    # 缺少 *，city 和 job 被视为位置参数
    pass
```

参数组合

在 Python 中定义函数，可以用必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这 5 种参数都可以组合使用，除了可变参数无法和命名关键字参数混合。**但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数/命名关键字参数和关键字参数。**

比如定义一个函数，包含上述若干种参数：

```
def f1(a, b, c=0, *args, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)

def f2(a, b, c=0, *, d, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)
```

在函数调用的时候，Python 解释器自动按照参数位置和参数名把对应的参数传进去。

```
>>> f1(1, 2)
a = 1 b = 2 c = 0 args = () kw = {}
>>> f1(1, 2, c=3)
a = 1 b = 2 c = 3 args = () kw = {}
>>> f1(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
>>> f1(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
>>> f2(1, 2, d=99, ext=None)
a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}
```

小结

*args 是可变参数，args 接收的是一个 tuple；

**kw 是关键字参数，kw 接收的是一个 dict。

以及调用函数时如何传入可变参数和关键字参数的语法：

可变参数既可以直接传入：func(1, 2, 3)，又可以先组装 list 或 tuple，再通过*args 传入：func(*(1, 2, 3))；

关键字参数既可以直接传入：func(a=1, b=2)，又可以先组装 dict，再通过**kw 传入：func(**{'a': 1, 'b': 2})。

使用*args 和**kw 是 Python 的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

命名的关键字参数是为了限制调用者可以传入的参数名，同时可以提供默认值。

定义命名的关键字参数不要忘了写分隔符*，否则定义的将是位置参数。

高级特性

切片

取一个 list 或 tuple 的部分元素是非常常见的操作。比如，一个 list 如下：

```
>>> L = ['Michael', 'Sarah', 'Tracy', 'Bob', 'Jack']
```

Python 提供了切片（Slice）操作符，能大大简化这种操作。

对应上面的问题，取前 3 个元素，用一行代码就可以完成切片：

```
>>> L[0:3]
['Michael', 'Sarah', 'Tracy']
```

前 10 个数，每两个取一个：

```
>>> L[:10:2]
[0, 2, 4, 6, 8]
```

字符串'xxx'也可以看成是一种 list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串：

```
>>> 'ABCDEFG'[:3]
'ABC'
>>> 'ABCDEFG'[::2]
'ACEG'
```

迭代

如果给定一个 list 或 tuple，我们可以通过 for 循环来遍历这个 list 或 tuple，这种遍历我们称为迭代（Iteration）。

可以看出，Python 的 for 循环抽象程度要高于 Java 的 for 循环，因为 Python 的 for 循环不仅可以用在 list 或 tuple 上，还可以作用在其他可迭代对象上。

list 这种数据类型虽然有下标，但很多其他数据类型是没有下标的，但是，只要是可迭代对象，无论有无下标，都可以迭代，比如 dict 就可以迭代：

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> for key in d:
...     print(key)
```

默认情况下，**dict 迭代的是 key。如果要迭代 value，可以用 for value in d.values()，如果要同时迭代 key 和 value，可以用 for k, v in d.items()。**

最后一个小问题，**如果要对 list 实现类似 Java 那样的下标循环怎么办？Python 内置的 enumerate 函数可以把一个 list 变成索引-元素对，**这样就可以在 for 循环中同时迭代索引和元素本身：

```
>>> for i, value in enumerate(['A', 'B', 'C']):
...     print(i, value)
...
0 A
1 B
2 C
```

上面的 for 循环里，同时引用了两个变量，在 Python 里是很常见的，比如下面的代码：

```
>>> for x, y in [(1, 1), (2, 4), (3, 9)]:
...     print(x, y)
```

列表生成式

列表生成式即 List Comprehensions，是 Python 内置的非常简单却强大的可以用来**创建 list 的生成式。**

循环太繁琐，而列表生成式则可以用一行语句代替循环生成上面的 list：

```
>>> [x * x for x in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

for 循环后面还可以加上 if 判断，这样我们就可以筛选出仅偶数的平方：

```
>>> [x * x for x in range(1, 11) if x % 2 == 0]
[4, 16, 36, 64, 100]
```

还可以使用两层循环，可以生成全排列：

```
>>> [m + n for m in 'ABC' for n in 'XYZ']
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

生成器

在 Python 中，这种一边循环一边计算的机制，称为生成器：generator。

要创建一个 generator，有很多种方法。**第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个 generator：**

```
>>> L = [x * x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> g = (x * x for x in range(10))
>>> g
<generator object <genexpr> at 0x1022ef630>
```

创建 L 和 g 的区别仅在于最外层的[]和()，L 是一个 list，而 g 是一个 generator。

使用 for 循环，因为 generator 也是可迭代对象：

```
>>> g = (x * x for x in range(10))
>>> for n in g:
...     print(n)
```

generator 非常强大。如果推算的算法比较复杂，用类似列表生成式的 for 循环无法实现的时候，还可以用函数来实现。

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print(b)
        a, b = b, a + b
        n = n + 1
    return 'done'
```

上面的函数可以输出斐波那契数列的前 N 个数：

上面的函数和 generator 仅一步之遥。要把 fib 函数变成 generator，只需要把 print(b) 改为 yield b 就可以了：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
```

```
        n = n + 1
    return 'done'
```

这就是定义 generator 的另一种方法。如果一个函数定义中包含 yield 关键字，那么这个函数就不再是一个普通函数，而是一个 generator：

这里，最难理解的就是 generator 和函数的执行流程不一样。函数是顺序执行，遇到 return 语句或者最后一行函数语句就返回。而变成 generator 的函数，**在每次调用 next()的时候执行，遇到 yield 语句返回，再次执行时从上次返回的 yield 语句处继续执行。**

举个简单的例子，定义一个 generator，依次返回数字 1，3，5：

```
def odd():
    print('step 1')
    yield 1
    print('step 2')
    yield(3)
    print('step 3')
    yield(5)
```

调用该 generator 时，首先要生成一个 generator 对象，然后用 next()函数不断获得下一个返回值：

```
>>> o = odd()
>>> next(o)
step 1
1
>>> next(o)
step 2
3
>>> next(o)
step 3
5
>>> next(o)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

回到 fib 的例子，我们在循环过程中不断调用 yield，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成 generator 后，我们基本上从来不会用 next()来获取下一个返回值，而是直接使用 for 循环来迭代：

```
>>> for n in fib(6):
...     print(n)
```

但是用 for 循环调用 generator 时，发现拿不到 generator 的 return 语句的返回值。如果想要拿到返回值，必须捕获 StopIteration 错误，返回值包含在 StopIteration 的 value 中：

```
>>> g = fib(6)
>>> while True:
...     try:
...         x = next(g)
...         print('g:', x)
...     except StopIteration as e:
...         print('Generator return value:', e.value)
...         break
```

函数式编程

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

高阶函数

变量可以指向函数

以 Python 内置的求绝对值的函数 abs()为例，调用该函数用以下代码：

```
>>> abs(-10)
10
```

但是，如果只写 abs 呢？

```
>>> abs
<built-in function abs>
```

可见，abs(-10)是函数调用，而 abs 是函数本身。

要获得函数调用结果，我们可以把结果赋值给变量：

```
>>> x = abs(-10)
>>> x
10
```

但是，如果把函数本身赋值给变量呢？

```
>>> f = abs
>>> f
<built-in function abs>
```

结论：函数本身也可以赋值给变量，即：变量可以指向函数。

函数名也是变量

那么函数名是什么呢？函数名其实就是指向函数的变量！对于 abs()这个函数，完全可以把函数名 abs 看成变量，它指向一个可以计算绝对值的函数！

如果把 abs 指向其他对象，会有什么情况发生？

```
>>> abs = 10
>>> abs(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

把 abs 指向 10 后，就无法通过 abs(-10)调用该函数了！因为 abs 这个变量已经不指向求绝对值函数而是指向一个整数 10！

当然实际代码绝对不能这么写，这里是为了说明函数名也是变量。要恢复 abs 函数，请重启 Python 交互环境。

注：由于 abs 函数实际上是定义在__builtin__模块中的，所以要让修改 abs 变量的指向在其它模块也生效，要用__builtin__.abs = 10。

传入函数

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

```
def add(x, y, f):
    return f(x) + f(y)
```

用代码验证一下：

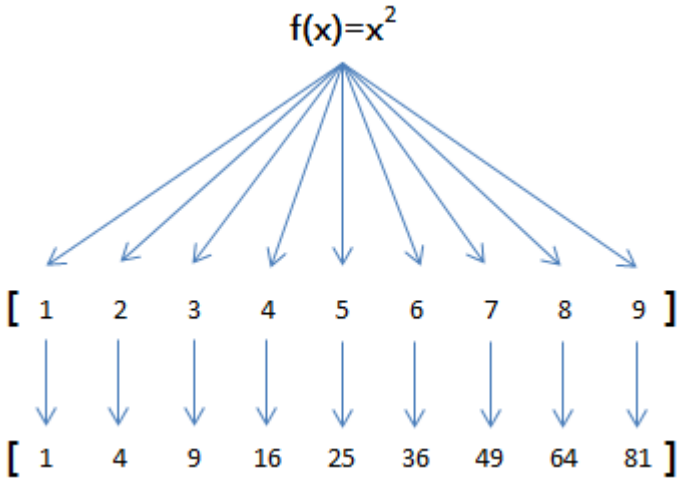
```
>>> add(-5, 6, abs)
11
```

编写高阶函数，就是让函数的参数能够接收别的函数。

map/reduce

我们先看 map。**map()函数接收两个参数，一个是函数，一个是 Iterable，map 将传入的函数依次作用到序列的每个元素，并把结果作为新的 Iterator 返回。**

举例说明，比如我们有一个函数 $f(x)=x^2$，要把这个函数作用在一个 list [1, 2, 3, 4, 5, 6, 7, 8, 9]上，就可以用 map()实现如下：



现在，我们用 Python 代码实现：

```
>>> def f(x):
```

```
...     return x * x
...
>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> list(r)
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

map()传入的第一个参数是 f，即函数对象本身。**由于结果 r 是一个 Iterator，Iterator 是惰性序列，因此通过 list()函数让它把整个序列都计算出来并返回一个 list。**

再看 reduce 的用法。**reduce 把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce 把结果继续和序列的下一个元素做累积计算，其效果就是：**

```
reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
```

比方说对一个序列求和，就可以用 reduce 实现：

```
>>> from functools import reduce
>>> def add(x, y):
...     return x + y
...
>>> reduce(add, [1, 3, 5, 7, 9])
25
```

filter

Python 内建的 filter()函数用于过滤序列。

和 map()类似，**filter()也接收一个函数和一个序列**。和 map()不同的时，**filter()把传入的函数依次作用于每个元素，然后根据返回值是 True 还是 False 决定保留还是丢弃该元素。**

例如，在一个 list 中，删掉偶数，只保留奇数，可以这么写：

```
def is_odd(n):
    return n % 2 == 1

list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15]))
# 结果: [1, 5, 9, 15]
```

用 filter 求素数

用 Python 来实现这个算法，可以先构造一个从 3 开始的奇数序列：

```
def _odd_iter():
    n = 1
    while True:
        n = n + 2
        yield n
```

注意这是一个生成器，并且是一个无限序列。

然后定义一个筛选函数：

```
def _not_divisible(n):
    return lambda x: x % n > 0
```

最后，定义一个生成器，不断返回下一个素数：

```
def primes():
    yield 2
    it = _odd_iter() # 初始序列
    while True:
        n = next(it) # 返回序列的第一个数
        yield n
        it = filter(_not_divisible(n), it) # 构造新序列
```

这个生成器先返回第一个素数 2，然后，利用 filter()不断产生筛选后的新的序列。

由于 primes()也是一个无限序列，所以调用时需要设置一个退出循环的条件：

```
# 打印 1000 以内的素数:
for n in primes():
    if n < 1000:
        print(n)
    else:
        break
```

sorted

排序算法

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个 dict 呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。

Python 内置的 sorted()函数就可以对 list 进行排序：

```
>>> sorted([36, 5, -12, 9, -21])
[-21, -12, 5, 9, 36]
```

此外，**sorted()函数也是一个高阶函数，它还可以接收一个 key 函数来实现自定义的排序，例如按绝对值大小排序：**

```
>>> sorted([36, 5, -12, 9, -21], key=abs)
[5, 9, -12, -21, 36]
```

要进行反向排序，不必改动 key 函数，可以传入第三个参数 reverse=True：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower,
reverse=True)
['Zoo', 'Credit', 'bob', 'about']
```

返回函数

函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

```
def calc_sum(*args):
    ax = 0
    for n in args:
        ax = ax + n
    return ax
```

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数：

```
def lazy_sum(*args):
    def sum():
        ax = 0
        for n in args:
            ax = ax + n
        return ax
    return sum
```

当我们调用 lazy_sum()时，返回的并不是求和结果，而是求和函数：

```
>>> f = lazy_sum(1, 3, 5, 7, 9)
>>> f
<function lazy_sum.<locals>.sum at 0x101c6ed90>
```

调用函数 f 时，才真正计算求和的结果：

```
>>> f()
25
```

在这个例子中，**我们在函数 lazy_sum 中又定义了函数 sum，并且，内部函数 sum 可以引用外部函数 lazy_sum 的参数和局部变量，当 lazy_sum 返回函数 sum 时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。**

请再注意一点，当我们调用 lazy_sum()时，每次调用都会返回一个新的函数，即使传入相同的参数：

```
>>> f1 = lazy_sum(1, 3, 5, 7, 9)
>>> f2 = lazy_sum(1, 3, 5, 7, 9)
>>> f1==f2
False
```

f1()和 f2()的调用结果互不影响。

闭包

注意到返回的函数在其定义内部引用了局部变量 args，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了 f()才执行。我们来看一个例子：

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
             return i*i
        fs.append(f)
    return fs

f1, f2, f3 = count()
```

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的 3 个函数都返回。
因为调用 f()才执行，而调用 f()时，i 作为局部变量（保存返回）都为 4.

你可能认为调用 f1()，f2()和 f3()结果应该是 1，4，9，但实际结果是：

```
>>> f1()
9
>>> f2()
9
>>> f3()
9
```

全部都是 9！原因就在于返回的函数引用了变量 i，但它并非立刻执行。等到 3 个函数都返回时，它们所引用的变量 i 已经变成了 3，因此最终结果为 9。

返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

```
def count():
    def f(j):
        def g():
            return j*j
        return g
    fs = []
    for i in range(1, 4):
        fs.append(f(i)) # f(i)立刻被执行，因此 i 的当前值被传入 f()
    return fs
```

再看看结果：

```
>>> f1, f2, f3 = count()
>>> f1()
1
>>> f2()
4
>>> f3()
9
```

小结

一个函数可以返回一个计算结果，也可以返回一个函数。

返回一个函数时，牢记该函数并未执行，返回函数中不要引用任何可能会变化的变量。

匿名函数

当我们在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。

在 Python 中，对匿名函数提供了有限支持。还是以 map()函数为例，计算 $f(x)=x^2$ 时，除了定义一个 f(x)的函数外，还可以直接传入匿名函数：

```
>>> list(map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9]))
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

通过对比可以看出，匿名函数 lambda x: x * x 实际上就是：

```
def f(x):
    return x * x
```

关键字 lambda 表示匿名函数，冒号前面的 x 表示函数参数。

匿名函数有个限制，就是只能有一个表达式，不用写 return，返回值就是该表达式的结果。

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。

装饰器

```
>>> def now():
```

```
...     print('2015-3-25')
```

假设我们要增强 now()函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改 now()函数的定义，这种**在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。**

本质上，decorator 就是一个返回函数的高阶函数。所以，我们**要定义一个能打印日志的 decorator**，可以定义如下：

```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

观察上面的 log，因为它是一个 decorator，所以接受一个函数作为参数，并返回一个函数。我们要**借助 Python 的@语法，把 decorator 置于函数的定义处：**

```
@log
def now():
    print('2015-3-25')
```

调用 now()函数，不仅会运行 now()函数本身，还会在运行 now()函数前打印一行日志：

```
>>> now()
call now():
2015-3-25
```

把@log 放到 now()函数的定义处，相当于执行了语句：

```
now = log(now)
```

由于 log()是一个 decorator，返回一个函数，所以，原来的 now()函数仍然存在，只是现在同名的 now 变量指向了新的函数，于是调用 now()将执行新函数，即在 log()函数中返回的 wrapper()函数。

wrapper()函数的参数定义是(*args, **kw)，因此，wrapper()函数可以接受任意参数的调用。在 wrapper()函数内，首先打印日志，再紧接着调用原始函数。

如果 decorator 本身需要传入参数，那就需要编写一个返回 decorator 的高阶函数，写出来会更复杂。比如，要自定义 log 的文本：

```
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

这个 3 层嵌套的 decorator 用法如下：

```
@log('execute')
def now():
    print('2015-3-25')
```

执行结果如下：

```
>>> now()
execute now():
2015-3-25
```

和两层嵌套的 decorator 相比，3 层嵌套的效果是这样的：

```
>>> now = log('execute')(now)
```

以上两种 decorator 的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有__name__等属性，但你去看经过 decorator 装饰之后的函数，它们的__name__已经从原来的'now'变成了'wrapper'：

```
>>> now.__name__
'wrapper'
```

因为返回的那个 wrapper()函数名字就是'wrapper'，所以，需要把原始函数的__name__等属性复制到 wrapper()函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写 wrapper.__name__ = func.__name__这样的代码，Python 内置的 functools.wraps 就是干这个事的（在 func 的上一层函数外添加），所以，一个完整的 decorator 的写法如下：

```
import functools

def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

偏函数

int()函数可以把字符串转换为整数，当仅传入字符串时，int()函数默认按十进制转换：

```
>>> int('12345')
12345
```

但 int()函数还提供额外的 base 参数，默认值为 10。如果传入 base 参数，就可以做 N 进制的转换：

```
>>> int('12345', base=8)
5349
```

functools.partial 就是帮助我们创建一个偏函数的，不需要我们自己定义 int2()，可以直接使用下面的代码创建一个新的函数 int2：

```
>>> import functools
>>> int2 = functools.partial(int, base=2)
>>> int2('1000000')
64
>>> int2('1010101')
85
```

所以，简单总结 functools.partial 的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

注意到上面的新的 int2 函数，仅仅是把 base 参数重新设定默认值为 2，但也可以在函数调用时传入其他值：

```
>>> int2('1000000', base=10)
```

```
1000000
```

模块

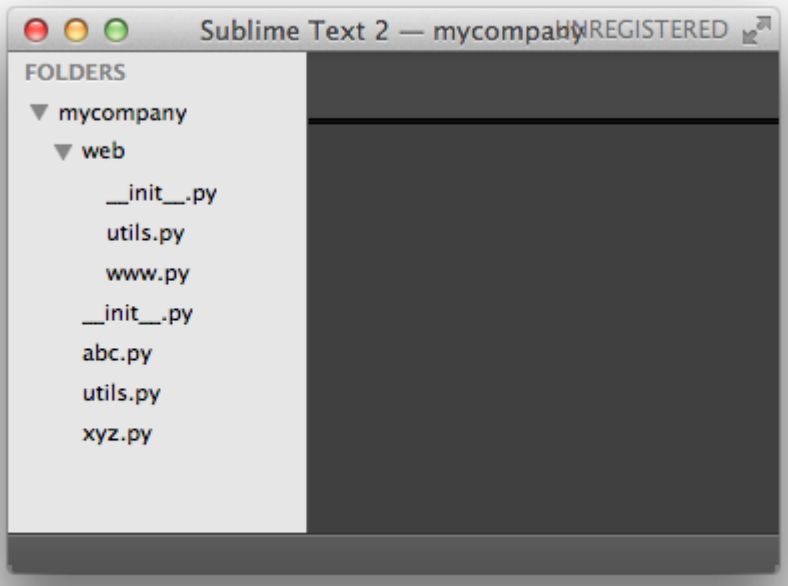
在 Python 中，一个.py 文件就称之为一个模块（Module）。使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。点这里查看 Python 的所有内置函数。

你也许还想到，如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python 又引入了按目录来组织模块的方法，称为包（Package）。

引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在，abc.py 模块的名字就变成了 mycompany.abc，类似的，xyz.py 的模块名变成了 mycompany.xyz。

请注意，每一个包目录下面都会有一个__init__.py 的文件，这个文件是必须存在的，否则，Python 就把这个目录当成普通目录，而不是一个包。__init__.py 可以是空文件，也可以有 Python 代码，因为__init__.py 本身就是一个模块，而它的模块名就是 mycompany。

类似的，可以有多级目录，组成多级层次的包结构。比如如下的目录结构：



文件 www.py 的模块名就是 mycompany.web.www，两个文件 utils.py 的模块名分别是 mycompany.utils 和 mycompany.web.utils。

使用模块

我们以内建的 sys 模块为例，编写一个 hello 的模块：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

' a test module '

__author__ = 'Michael Liao'

import sys

def test():
    args = sys.argv
    if len(args)==1:
            print('Hello, world!')
    elif len(args)==2:
        print('Hello, %s!' % args[1])
    else:
        print('Too many arguments!')

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

第 4 行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释；

最后，注意到这两行代码：

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

当我们在命令行运行 hello 模块文件时，Python 解释器把一个特殊变量__name__置为__main__，而如果在其他地方导入该 hello 模块时，if 判断将失败，因此，这种 if 测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。

对于：
from lxml import etree

是
from Module import Function 或 Class 等
这个只是从模块中导入一个或几个函数或类的做法。
另外一个常见的是
import Module

作用域

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在 Python 中，是通过_前缀来实现的。

正常的函数和变量名是公开的（public），可以被直接引用，比如：abc，x123，PI 等；

类似__xxx__这样的变量是特殊变量，可以被直接引用，但是有特殊用途，比如上面的__author__，__name__就是特殊变量，hello 模块定义的文档注释也可以用特殊变量__doc__访问，我们自己的变量一般不要用这种变量名；

类似_xxx 和__xxx 这样的函数或变量就是非公开的（private），不应该被直接引用，比如_abc，__abc 等；

之所以我们说，private 函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为 Python 并没有一种方法可以完全限制访问 private 函数或变量，但是，从编程习惯上不应该引用 private 函数或变量。

安装第三方模块

如果你正在使用 Windows，请参考安装 Python 一节的内容，确保安装时勾选了 pip 和 Add python.exe to Path。

在命令提示符窗口下尝试运行 pip，如果 Windows 提示未找到命令，可以重新运行安装程序添加 pip。

```
pip install Pillow
```

耐心等待下载并安装后，就可以使用 Pillow 了。

模块搜索路径

默认情况下，Python 解释器会搜索当前目录、所有已安装的内置模块和第三方模块，**搜索路径存放在 sys 模块的 path 变量中：**

```
>>> import sys
>>> sys.path
['', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python34.zip', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/plat-darwin', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/lib-dynload', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/site-packages']
```

如果我们要添加自己的搜索目录，有两种方法：

一是直接修改 sys.path，添加要搜索的目录：

```
>>> import sys
>>> sys.path.append('/Users/michael/my_py_scripts')
```

这种方法是在运行时修改，运行结束后失效。

第二种方法是设置环境变量 PYTHONPATH，该环境变量的内容会被自动添加到模块搜索路径中。设置方式与设置 Path 环境变量类似。注意只需要添加你自己的搜索路径，Python 自己本身的搜索路径不受影响。

面向对象编程

OOP 把对象作为程序的基本单元，**一个对象包含了数据和操作数据的函数。**

面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，**计算机程序的执行就是一系列消息在各个对象之间传递。**

```
class Student(object):

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

    def print_score(self):
        print('%s: %s' % (self.name, self.score))
```

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。面向对象的程序写出来就像这样：

```
bart = Student('Bart Simpson', 59)
lisa = Student('Lisa Simpson', 87)
bart.print_score()
lisa.print_score()
```

面向对象的抽象程度又比函数要高，因为一个 Class 既包含数据，又包含操作数据的方法。

类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，比如 Student 类，而实例是根据类创建出来的一个个具体的"对象"，每个对象都拥有相同的方法，但各自的数据可能不同。

仍以 Student 类为例，在 Python 中，定义类是通过 class 关键字：

```
class Student(object):
    pass
```

class 后面紧接着是类名，即 Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来的，继承的概念我们后面再讲，通常，如果没有合适的继承类，就使用 object 类，这是所有类最终都会继承的类。

定义好了 Student 类，就可以根据 Student 类创建出 Student 的实例，创建实例是通过类名+()实现的：

```
>>> bart = Student()
>>> bart = Student()
>>> bart
<__main__.Student object at 0x10a67a590>
>>> Student
<class '__main__.Student'>
```

可以看到，变量 bart 指向的就是一个 Student 的实例，后面的 0x10a67a590 是内存地址，每个 object 的地址都不一样，而 Student 本身则是一个类。

可以自由地给一个实例变量绑定属性，比如，给实例 bart 绑定一个 name 属性：

```
>>> bart.name = 'Bart Simpson'
>>> bart.name
'Bart Simpson'
```

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的__init__方法，在创建实例的时候，就把 name，score 等属性绑上去：

```
class Student(object):

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

注意到__init__方法的第一个参数永远是 self，表示创建的实例本身，因此，在__init__方法内部，就可以把各种属性绑定到 self，因为 self 就指向创建的实例本身。

有了__init__方法，在创建实例的时候，就不能传入空的参数了，必须传入与__init__方法匹配的参数，但 self 不需要传，Python 解释器自己会把实例变量传进去：

```
>>> bart = Student('Bart Simpson', 59)
```

访问限制

在 Class 内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。

但是，从前面 Student 类的定义来看，外部代码还是可以自由地修改一个实例的 name、score 属性：

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线__，在 Python 中，实例的变量名如果以__开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问，所以，我们把 Student 类改一改：

```
class Student(object):

    def __init__(self, name, score):
        self.__name = name
        self.__score = score
```

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问实例变量.__name 和实例变量.__score 了：

有些时候，你会看到以一个下划线开头的实例变量名，比如_name，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问__name 是因为 Python 解释器对外把__name 变量改成了_Student__name，所以，仍然可以通过_Student__name 来访问__name 变量：

```
>>> bart._Student__name
'Bart Simpson'
```

但是强烈建议你不要这么干，因为不同版本的 Python 解释器可能会把__name 改成不同的变量名。

继承和多态

在 OOP 程序设计中，当我们定义一个 class 的时候，可以从某个现有的 class 继承，新的 class 称为子类（Subclass），而被继承的 class 称为基类、父类或超类（Base class、Super class）。

比如，我们已经编写了一个名为 Animal 的 class，有一个 run()方法可以直接打印：

```
class Animal(object):
    def run(self):
        print('Animal is running...')
```

当我们需要编写 Dog 和 Cat 类时，就可以直接从 Animal 类继承：

```
class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

对于 Dog 来说，Animal 就是它的父类，对于 Animal 来说，Dog 就是它的子类。Cat 和 Dog 类似。

继承有什么好处？最大的好处是子类获得了父类的全部功能。由于 Animial 实现了 run()方法，因此，Dog 和 Cat 作为它的子类，什么事也没干，就自动拥有了 run()方法：

继承的第二个好处需要我们对代码做一点改进。你看到了，无论是 Dog 还是 Cat，它们 run()的时候，显示的都是 Animal is running...，符合逻辑的做法是分别显示 Dog is running...和 Cat is running...，因此，对 Dog 和 Cat 类改进如下：

```
class Dog(Animal):

    def run(self):
        print('Dog is running...')
```

当子类和父类都存在相同的 run()方法时，我们说，子类的 run()覆盖了父类的 run()，在代码运行的时候，总是会调用子类的 run()。这样，我们就获得了继承的另一个好处：多态。

```
>>> isinstance(c, Dog)
True
```

看来 c 确实对应着 Dog 这 3 种类型。

但是等等，试试：

```
>>> isinstance(c, Animal)
True
```

看来 c 不仅仅是 Dog，c 还是 Animal！

不过仔细想想，这是有道理的，因为 Dog 是从 Animal 继承下来的，当我们创建了一个 Dog 的实例 c 时，我们认为 c 的数据类型是 Dog 没错，但 c 同时也是 Animal 也没错，Dog 本来就是 Animal 的一种！

要理解多态的好处，我们还需要再编写一个函数，这个函数接受一个 Animal 类型的变量：

```
def run_twice(animal):
    animal.run()
    animal.run()
```

当我们传入 Animal 的实例时，run_twice()就打印出：

```
>>> run_twice(Animal())
Animal is running...
Animal is running...
```

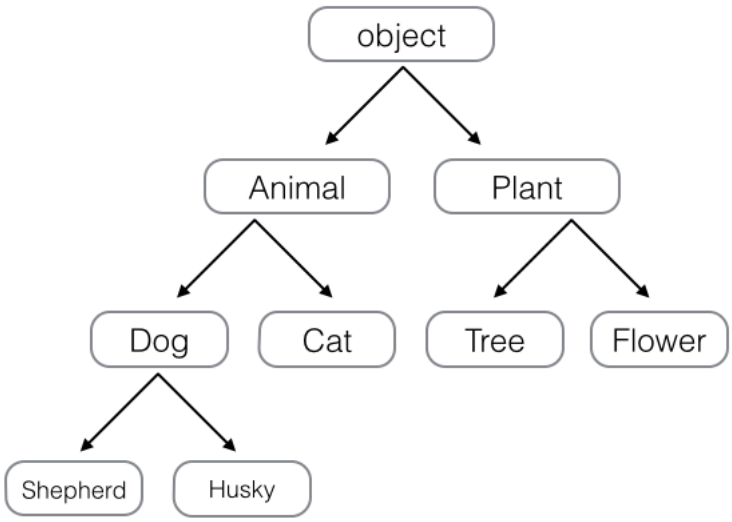
当我们传入 Dog 的实例时，run_twice()就打印出：

```
>>> run_twice(Dog())
Dog is running...
Dog is running...
```

多态的好处就是，当我们需要传入 Dog、Cat、.....时，我们只需要接收 Animal 类型就可以了，因为 Dog、Cat、.....都是 Animal 类型，然后，按照 Animal 类型进行操作即可。由于 Animal 类型有 run()方法，因此，传入的任意类型，只要是 Animal 类或者子类，就会自动调用实际类型的 run()方法，这就是多态的意思：

对于一个变量，我们只需要知道它是 Animal 类型，无需确切地知道它的子类型，就可以放心地调用 run()方法，而具体调用的 run()方法是作用在 Animal、Dog、Cat 上，由运行时该对象的确切类型决定，这就是多态真正的威力

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类 object，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：



静态语言 vs 动态语言

对于静态语言（例如 Java）来说，如果需要传入 Animal 类型，则传入的对象必须是 Animal 类型或者它的子类，否则，将无法调用 run()方法。

对于 Python 这样的动态语言来说，则不一定需要传入 Animal 类型。我们只需要保证传入的对象有一个 run()方法就可以了：

```
class Timer(object):
    def run(self):
        print('Start...')
```

这就是动态语言的“鸭子类型”，它并不要求严格的继承体系，一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子。

Python 的“file-like object“就是一种鸭子类型。对真正的文件对象，它有一个 read()方法，返回其内容。但是，许多对象，只要有 read()方法，都被视为“file-like object“。许多函数接收的参数就是“file-like object“，你不一定要传入真正的文件对象，完全可以传入任何实现了 read()方法的对象。

获取对象信息

当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

使用 type()

首先，我们来判断对象类型，使用 type()函数：

使用 isinstance()

对于 class 的继承关系来说，使用 type()就很不方便。我们要判断 class 的类型，可以使用 isinstance()函数。

我们回顾上次的例子，如果继承关系是：

```
object -> Animal -> Dog -> Husky
```

那么，isinstance()就可以告诉我们，一个对象是否是某种类型。先创建 3 种类型的对象：

```
>>> a = Animal()
>>> d = Dog()
>>> h = Husky()
```

然后，判断：

```
>>> isinstance(h, Husky)
True
```

换句话说，**isinstance()判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上。**

能用 type()判断的基本类型也可以用 isinstance()判断：

```
>>> isinstance('a', str)
True
```

并且还可以判断一个变量是否是某些类型中的一种，比如下面的代码就可以判断是否是 list 或者 tuple：

```
>>> isinstance([1, 2, 3], (list, tuple))
True
```

使用 dir()

如果要获得一个对象的所有属性和方法，可以使用 dir()函数，它返回一个包含字符串的 list，比如，获得一个 str 对象的所有属性和方法：

```
>>> dir('ABC')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__']
```

类似__xxx__的属性和方法在 Python 中都是有特殊用途的，比如__len__方法返回长度。在 Python 中，如果你调用 len()函数试图获取一个对象的长度，实际上，在 len()函数内部，它自动去调用该对象的__len__()方法，所以，下面的代码是等价的：

```
>>> len('ABC')
3
>>> 'ABC'.__len__()
3
```

仅仅把属性和方法列出来是不够的，配合 getattr()、setattr()以及 hasattr()，我们可以直接操作一个对象的状态：

```
>>> class MyObject(object):
...     def __init__(self):
...         self.x = 9
...     def power(self):
...         return self.x * self.x
...
>>> obj = MyObject()
```

紧接着，可以测试该对象的属性：

```
>>> hasattr(obj, 'x') # 有属性'x'吗?
True
>>> obj.x
9
>>> hasattr(obj, 'y') # 有属性'y'吗?
False
>>> setattr(obj, 'y', 19) # 设置一个属性'y'
>>> hasattr(obj, 'y') # 有属性'y'吗?
True
>>> getattr(obj, 'y') # 获取属性'y'
19
>>> obj.y # 获取属性'y'
19
```

如果试图获取不存在的属性，会抛出 AttributeError 的错误：

```
>>> getattr(obj, 'z') # 获取属性'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyObject' object has no attribute 'z'
```

可以传入一个 default 参数，如果属性不存在，就返回默认值：

```
>>> getattr(obj, 'z', 404) # 获取属性'z'，如果不存在，返回默认值 404
404
```

也**可以获得对象的方法：**

```
>>> hasattr(obj, 'power') # 有属性'power'吗?
True
```

```
>>> getattr(obj, 'power') # 获取属性'power'
<bound method MyObject.power of <__main__.MyObject object at 0x10077a
6a0>>
>>> fn = getattr(obj, 'power') # 获取属性'power'并赋值到变量 fn
>>> fn # fn 指向 obj.power
<bound method MyObject.power of <__main__.MyObject object at 0x10077a
6a0>>
>>> fn() # 调用 fn()与调用 obj.power()是一样的
81
```

实例属性和类属性

由于 Python 是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过 self 变量：

但是，**如果 Student 类本身需要绑定一个属性呢？可以直接在 class 中定义属性，这种属性是类属性，归 Student 类所有：**

```
class Student(object):
    name = 'Student'
```

当我们定义了一个类属性后，这个属性虽然归类所有，**但类的所有实例都可以访问到。**来测试一下：

```
>>> class Student(object):
...     name = 'Student'
...
>>> s = Student() # 创建实例 s
>>> print(s.name) # 打印 name 属性，因为实例并没有 name 属性，所以会继续
查找 class 的 name 属性
Student
>>> print(Student.name) # 打印类的 name 属性
Student
>>> s.name = 'Michael' # 给实例绑定 name 属性
>>> print(s.name) # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的
name 属性
Michael
>>> print(Student.name) # 但是类属性并未消失，用 Student.name 仍然可以
访问
Student
```

```
>>> del s.name # 如果删除实例的 name 属性
>>> print(s.name) # 再次调用 s.name，由于实例的 name 属性没有找到，类的 name 属性就显示出来了
Student
```

从上面的例子可以看出，在编写程序的时候，**千万不要把实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。**

面向对象高级编程

使用__slots__

正常情况下，当我们定义了一个 class，创建了一个 class 的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性。先定义 class：

```
class Student(object):
    pass
```

然后，尝试给实例绑定一个属性：

```
>>> s = Student()
>>> s.name = 'Michael' # 动态给实例绑定一个属性
>>> print(s.name)
Michael
```

还可以尝试给实例绑定一个方法：

```
>>> def set_age(self, age): # 定义一个函数作为实例方法
...     self.age = age
...
>>> from types import MethodType
>>> s.set_age = MethodType(set_age, s) # 给实例绑定一个方法
>>> s.set_age(25) # 调用实例方法
>>> s.age # 测试结果
25
```

但是，**给一个实例绑定的方法，对另一个实例是不起作用的：**

```
>>> s2 = Student() # 创建新的实例
>>> s2.set_age(25) # 尝试调用方法
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'set_age'
```

为了给所有实例都绑定方法，可以给 class 绑定方法：

```
>>> def set_score(self, score):
...     self.score = score
...
>>> Student.set_score = MethodType(set_score, Student)
```

给 class 绑定方法后，所有实例均可调用：

```
>>> s.set_score(100)
>>> s.score
100
>>> s2.set_score(99)
>>> s2.score
99
```

通常情况下，上面的 set_score 方法可以直接定义在 class 中，但动态绑定允许我们在程序运行的过程中动态给 class 加上功能，这在静态语言中很难实现。

使用__slots__

但是，如果我们想要限制实例的属性怎么办？比如，只允许对 Student 实例添加 name 和 age 属性。

为了达到限制的目的，Python 允许在定义 class 的时候，定义一个特殊的__slots__变量，来限制该 class 实例能添加的属性：

```
class Student(object):
    __slots__ = ('name', 'age') # 用 tuple 定义允许绑定的属性名称
```

然后，我们试试：

```
>>> s = Student() # 创建新的实例
>>> s.name = 'Michael' # 绑定属性'name'
```

```
>>> s.age = 25 # 绑定属性'age'
>>> s.score = 99 # 绑定属性'score'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'score'
```

由于'score'没有被放到__slots__中，所以不能绑定 score 属性，试图绑定 score 将得到 AttributeError 的错误。

使用__slots__要注意，__slots__定义的属性仅对当前类实例起作用，对继承的子类是不起作用的：

```
>>> class GraduateStudent(Student):
...     pass
...
>>> g = GraduateStudent()
>>> g.score = 9999
```

除非在子类中也定义__slots__，这样，子类实例允许定义的属性就是自身的__slots__加上父类的__slots__。

看到这里就让我想起了基于原型的继承。
向基于原型的继承生成的对象上添加属性方法，并不会影响其他基于同一原型实现的其他对象
在原型上添加属性方法，则在由此原型生成的对象上都能使用
调用一个属性或方法，如果当前对象上没有，则会在类(原型对象)上查找。表现为在类上添加属性，所有实例都可访问
slots 只是限制该类生成的对象上的的属性，并不限制类本身(原型对象)上的属性
slots 只限制本当前层次对象的属性，并不限制其子类对象的属性

```
class Person(object):
    __slots__ = ('name',)
#----------------------------
p1 = Person()
p2 = Person()
p1.name = 'xx'   # ok
try:
    p1.age =   10 # error
    print(p2.name) # error
except Exception as e:
    print('[Error]',e)
# ----------------------------
```

Person.age = 20 # 属性被挂载在类(原型)上，不受__slots__限制,等价于 setattr(Person,age,10)，且影响所有对象

print(p1.name,p1.age) # 这里 p1.age 首先从对象 p1 本身查找，没有找到就查找类上的属性,找到了直接输出 20

```
# 因为实例 p1 被__slots__限制添加属性，但他的类上有属性,因此 p1.age 实际上调用的是类上挂载的属性
# 凡是对象调用类中的属性，那么该对象只有读取类中属性的权限，而没有改写类属性的权限
print(p1.age) # 20
print(p2.age) # 20
try:
    Person.age = 88 # 类自己改自己的属性无压力
    p1.age = 99 # 对象尝试在这里改写类属性，错误
except Exception as e:
    print('[Error]',e)
    print('After revise p1.age by Class = ',p1.age) # 输出 88

# ---------------------------
class Stu(Person):
    pass
stu = Stu()
stu.age = 30 # 父类中的__slots__不会影响子类
print('stu s age is ',stu.age)

第一，slots 只能限制添加属性，不能限制通过添加方法来添加属性：
def set_city(self, city):
    self.city=city

class Student(object):
    __slots__ = ('name', 'age', 'set_city')
    pass

Student.set_city = MethodType(set_city, Student)

a = Student()
a.set_city(Beijing)
a.city
上段代码中，Student 类限制两个属性 name 和 age，但可以通过添加方法添加一个 city 属性（甚至可以添加很多属性，只要 set_city 方法里有包括）
第二，属性分实例属性和类属性，多个实例同时更改类属性，值是最后更改的一个
def set_age(self,age):
    self.age=age

class Stu(object):
    pass

s=Stu()
a=Stu()

from types import MethodType

Stu.set_age=MethodType(set_age,Stu)

a.set_age(15) \\通过 set_age 方法，设置的类属性 age 的值
s.set_age(11) \\也是设置类属性 age 的值，并把上个值覆盖掉
print(s.age,a.age) \\由于 a 和 s 自身没有 age 属性，所以打印的是类属性 age 的值

a.age = 10   \\给实例 a 添加一个属性 age 并赋值为 10
s.age = 20   \\给实例 b 添加一个属性 age 并赋值为 20
```

\\这两个分别是实例 a 和 s 自身的属性，仅仅是与类属性 age 同名，并没有任何关系

print(s.age,a.age) \\打印的是 a 和 s 自身的 age 属性值，不是类 age 属性值
所以，

1， slots 并不能严格限制属性的添加，可通过在方法里定义限制之外的属性来添加本不能添加的属性（当然，前提是方法没有被限制）

2， 类属性是公共属性，所有实例都可以引用的，前提是实例自身没有同名的属性，因此类属性不能随意更改（别的实例可能在引用类属性的值），就是说不能随便用 a.set_age() 更改 age 的值（因为调用此方法更改的是类属性 age 的值，不是实例 a 自身的 age 属性值

使用@property

还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。**Python 内置的@property 装饰器就是负责把一个方法变成属性调用的：**

```
class Student(object):

    @property
    def score(self):
        return self._score

    @score.setter
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
```

@property 的实现比较复杂，我们先考察如何使用。把一个 getter 方法变成属性，只需要加上@property 就可以了，此时，@property 本身又创建了另一个装饰器@score.setter，负责把一个 setter 方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

以上两个@property、@score.setter 为成对使用，可不成对使用

备注：如果使用 setter 属性，则在 setter 之前，必须调用 property 属性方法，不然报错；

```
>>> s = Student()
>>> s.score = 60 # OK，实际转化为 s.set_score(60)
>>> s.score # OK，实际转化为 s.get_score()
60
>>> s.score = 9999
```

```
Traceback (most recent call last):
  ...
ValueError: score must between 0 ~ 100!
```

注意到这个神奇的@property，我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过 getter 和 setter 方法来实现的。

还可以定义只读属性，只定义 getter 方法，不定义 setter 方法就是一个只读属性：

```
class Student(object):

    @property
    def birth(self):
        return self._birth

    @birth.setter
    def birth(self, value):
        self._birth = value

    @property
    def age(self):
        return 2015 - self._birth
```

上面的 birth 是可读写属性，而 age 就是一个**只读**属性，因为 age 可以根据 birth 和当前时间计算出来。

多重继承

通过多重继承，一个子类就可以同时获得多个父类的所有功能。

MixIn

在设计类的继承关系时，通常，主线都是单一继承下来的，例如，Ostrich 继承自 Bird。但是，如果需要“混入”额外的功能，通过多重继承就可以实现，比如，让 Ostrich 除了继承自 Bird 外，再同时继承 Runnable。这种设计通常称之为 MixIn。

为了更好地看出继承关系，我们把 Runnable 和 Flyable 改为 RunnableMixIn 和 FlyableMixIn。类似的，你还可以定义出肉食动物 CarnivorousMixIn 和植食动物 HerbivoresMixIn，让某个动物同时拥有好几个 MixIn：

```
class Dog(Mammal, RunnableMixIn, CarnivorousMixIn):
```

```
    pass
```

MixIn 的目的就是给一个类增加多个功能，这样，在设计类的时候，我们优先考虑通过多重继承来组合多个 MixIn 的功能，而不是设计多层次的复杂的继承关系。

Python 自带的很多库也使用了 MixIn。举个例子，Python 自带了 TCPServer 和 UDPServer 这两类网络服务，而要同时服务多个用户就必须使用多进程或多线程模型，这两种模型由 ForkingMixIn 和 ThreadingMixIn 提供。通过组合，我们就可以创造出合适的服务来。

比如，编写一个多进程模式的 TCP 服务，定义如下：

```
class MyTCPServer(TCPServer, ForkingMixIn):
    pass
```

编写一个多线程模式的 UDP 服务，定义如下：

```
class MyUDPServer(UDPServer, ThreadingMixIn):
    pass
```

如果你打算搞一个更先进的协程模型，可以编写一个 CoroutineMixIn：

```
class MyTCPServer(TCPServer, CoroutineMixIn):
    pass
```

这样一来，我们不需要复杂而庞大的继承链，只要选择组合不同的类的功能，就可以快速构造出所需的子类。

小结

由于 Python 允许使用多重继承，因此，MixIn 就是一种常见的设计。

只允许单一继承的语言（如 Java）不能使用 MixIn 的设计。

错误、调试和测试

错误处理

高级语言通常都内置了一套 try...except...finally... 的错误处理机制，Python 也不例外。

try

让我们用一个例子来看看 try 的机制：

```
try:
    print('try...')
    r = 10 / 0
    print('result:', r)
except ZeroDivisionError as e:
    print('except:', e)
finally:
    print('finally...')
print('END')
```

当我们认为某些代码可能会出错时，就可以用 try 来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即 except 语句块，执行完 except 后，如果有 finally 语句块，则执行 finally 语句块，至此，执行完毕。

你还可以猜测，错误应该有很多种类，**如果发生了不同类型的错误，应该由不同的 except 语句块处理。没错，可以有多个 except 来捕获不同类型的错误：**

```
try:
    print('try...')
    r = 10 / int('a')
    print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
finally:
    print('finally...')
print('END')
```

int()函数可能会抛出 ValueError，所以我们用一个 except 捕获 ValueError，用另一个 except 捕获 ZeroDivisionError。

此外，如果没有错误发生，**可以在 except 语句块后面加一个 else，当没有错误发生时，会自动执行 else 语句：**

```
try:
    print('try...')
    r = 10 / int('2')
```

```
    print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
else:
    print('no error!')
finally:
    print('finally...')
print('END')
```

Python 的错误其实也是 class，所有的错误类型都继承自 BaseException，所以在使用 except 时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

```
try:
    foo()
except ValueError as e:
    print('ValueError')
except UnicodeError as e:
    print('UnicodeError')
```

第二个 except 永远也捕获不到 UnicodeError，因为 UnicodeError 是 ValueError 的子类，如果有，也被第一个 except 给捕获了。

Python 所有的错误都是从 BaseException 类派生的，常见的错误类型和继承关系看这里：

https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
```

```
     |    +-- FloatingPointError
     |    +-- OverflowError
     |    +-- ZeroDivisionError
     +-- AssertionError
     +-- AttributeError
     +-- BufferError
     +-- EOFError
     +-- ImportError
     +-- LookupError
     |    +-- IndexError
     |    +-- KeyError
     +-- MemoryError
     +-- NameError
     |    +-- UnboundLocalError
     +-- OSError
     |    +-- BlockingIOError
     |    +-- ChildProcessError
     |    +-- ConnectionError
     |    |    +-- BrokenPipeError
     |    |    +-- ConnectionAbortedError
     |    |    +-- ConnectionRefusedError
     |    |    +-- ConnectionResetError
     |    +-- FileExistsError
     |    +-- FileNotFoundError
     |    +-- InterruptedError
     |    +-- IsADirectoryError
     |    +-- NotADirectoryError
     |    +-- PermissionError
     |    +-- ProcessLookupError
     |    +-- TimeoutError
     +-- ReferenceError
     +-- RuntimeError
     |    +-- NotImplementedError
     |    +-- RecursionError
     +-- SyntaxError
     |    +-- IndentationError
     |         +-- TabError
```

```
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

抛出错误

因为错误是 class，捕获一个错误就是捕获到该 class 的一个实例。因此，错误并不是凭空产生的，而是**有意创建并抛出的。Python 的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。**

如果要抛出错误，首先根据需要，可以定义一个错误的 class，选择好继承关系，然后，用 raise 语句抛出一个错误的实例：

```python
# err_raise.py
class FooError(ValueError):
    pass

def foo(s):
    n = int(s)
    if n==0:
        raise FooError('invalid value: %s' % s)
    return 10 / n
```

```
foo('0')
```

执行，可以最后跟踪到我们自己定义的错误：

```
$ python3 err_raise.py
Traceback (most recent call last):
  File "err_throw.py", line 11, in <module>
    foo('0')
  File "err_throw.py", line 8, in foo
    raise FooError('invalid value: %s' % s)
__main__.FooError: invalid value: 0
```

只有在必要的时候才定义我们自己的错误类型。如果可以选择 Python 已有的内置的错误类型（比如 ValueError，TypeError），尽量使用 Python 内置的错误类型。

记录错误

如果不捕获错误，自然可以让 Python 解释器来打印出错误堆栈，但程序也被结束了。既然我们能捕获错误，就可以把错误堆栈打印出来，然后分析错误原因，同时，让程序继续执行下去。

Python 内置的 logging 模块可以非常容易地记录错误信息：

```
# err_logging.py

import logging

def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except Exception as e:
        logging.exception(e)

main()
print('END')
```

同样是出错，但程序打印完错误信息后会继续执行，并正常退出：

```
$ python3 err_logging.py
ERROR:root:division by zero
Traceback (most recent call last):
  File "err_logging.py", line 13, in main
    bar('0')
  File "err_logging.py", line 9, in bar
    return foo(s) * 2
  File "err_logging.py", line 6, in foo
    return 10 / int(s)
ZeroDivisionError: division by zero
END
```

通过配置，logging 还可以把错误记录到日志文件里，方便事后排查。

IO 编程

文件读写

读文件

要以读文件的模式打开一个文件对象，使用 Python 内置的 open()函数，传入文件名和标示符：

```
>>> f = open('/Users/michael/test.txt', 'r')
```

标示符'r'表示读，这样，我们就成功地打开了一个文件。

如果文件不存在，open()函数就会抛出一个 IOError 的错误，并且给出错误码和详细的信息告诉你文件不存在：

如果文件打开成功，接下来，调用 read()方法可以一次读取文件的全部内容，Python 把内容读到内存，用一个 str 对象表示：

```
>>> f.read()
'Hello, world!'
```

最后一步是调用 close()方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

```
>>> f.close()
```

由于文件读写时都有可能产生 IOError，一旦出错，后面的 f.close()就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用 try ... finally 来实现：

```
try:
    f = open('/path/to/file', 'r')
    print(f.read())
finally:
    if f:
        f.close()
```

调用 read()会一次性读取文件的全部内容，如果文件有 10G，内存就爆了，所以，要保险起见，可以反复调用 read(size)方法，每次最多读取 size 个字节的内容。另外，调用 readline()可以每次读取一行内容，调用 readlines()一次读取所有内容并按行返回 list。因此，要根据需要决定怎么调用。

如果文件很小，read()一次性读取最方便；如果不能确定文件大小，反复调用 read(size)比较保险；如果是配置文件，调用 readlines()最方便：

```
for line in f.readlines():
    print(line.strip()) # 把末尾的'\n'删掉
```

写文件

写文件和读文件是一样的，唯一区别是调用 open()函数时，传入标识符'w'或者'wb'表示写文本文件或写二进制文件：

```
>>> f = open('/Users/michael/test.txt', 'w')
>>> f.write('Hello, world!')
>>> f.close()
```

操作文件和目录

Python 内置的 os 模块也可以直接调用操作系统提供的接口函数。

打开 Python 交互式命令行，我们来看看如何使用 os 模块的基本功能：

```
>>> import os
```

操作文件和目录的函数一部分放在 os 模块中，一部分放在 os.path 模块中，这一点要注意一下。**查看、创建和删除目录可以这么调用**：

```
# 查看当前目录的绝对路径:
>>> os.path.abspath('.')
'/Users/michael'
# 在某个目录下创建一个新目录，首先把新目录的完整路径表示出来:
>>> os.path.join('/Users/michael', 'testdir')
'/Users/michael/testdir'
# 然后创建一个目录:
>>> os.mkdir('/Users/michael/testdir')
# 删掉一个目录:
>>> os.rmdir('/Users/michael/testdir')
```

把两个路径合成一个时，不要直接拼字符串，而要通过 os.path.join()函数

同样的道理，要**拆分路径**时，也不要直接去拆字符串，而要通过 os.path.split()函数，这样可以把一个路径拆分为两部分，后一部分总是最后级别的目录或文件名：

```
>>> os.path.split('/Users/michael/testdir/file.txt')
('/Users/michael/testdir', 'file.txt')
```

os.path.splitext()可以直接让你**得到文件扩展名**，很多时候非常方便：

```
>>> os.path.splitext('/path/to/file.txt')
('/path/to/file', '.txt')
```

这些合并、拆分路径的函数并不要求目录和文件要真实存在，它们只对字符串进行操作。

文件操作使用下面的函数。假定当前目录下有一个 test.txt 文件：

```
# 对文件重命名:
>>> os.rename('test.txt', 'test.py')
# 删掉文件:
>>> os.remove('test.py')
```

最后看看如何**利用 Python 的特性来过滤文件**。比如我们要**列出当前目录下的所有目录**，只需要一行代码：

```
>>> [x for x in os.listdir('.') if os.path.isdir(x)]
['.lein', '.local', '.m2', '.npm', '.ssh', '.Trash', '.vim', 'Applications', 'Desktop', ...]
```

要列出所有的.py 文件，也只需一行代码：

```
>>> [x for x in os.listdir('.') if os.path.isfile(x) and os.path.splitext(x)[1]=='.py']
['apis.py', 'config.py', 'models.py', 'pymonitor.py', 'test_db.py', 'urls.py', 'wsgiapp.py']
```

序列化

在程序运行的过程中，所有的变量都是在内存中，比如，定义一个 dict：

```
d = dict(name='Bob', age=20, score=88)
```

可以随时修改变量，比如把 name 改成'Bill'，但是一旦程序结束，变量所占用的内存就被操作系统全部回收。如果没有把修改后的'Bill'存储到磁盘上，下次重新运行程序，变量又被初始化为'Bob'。

我们把变量从内存中变成可存储或传输的过程称之为序列化，在 Python 中叫 pickling，在其他语言中也被称之为 serialization，marshalling，flattening 等等，都是一个意思。

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即 unpickling。

Python 提供了 pickle 模块来实现序列化。

首先，我们尝试把一个对象序列化并写入文件：

```
>>> import pickle
>>> d = dict(name='Bob', age=20, score=88)
>>> pickle.dumps(d)
b'\x80\x03}q\x00(X\x03\x00\x00\x00ageq\x01K\x14X\x05\x00\x00\x00scoreq\x02KXX\x04\x00\x00\x00nameq\x03X\x03\x00\x00\x00Bobq\x04u.'
```

pickle.dumps()方法把任意对象序列化成一个 bytes，然后，就可以把这个 bytes 写入文件。或者用另一个方法 pickle.dump()直接把对象序列化后写入一个 file-like Object：

```
>>> f = open('dump.txt', 'wb')
>>> pickle.dump(d, f)
>>> f.close()
```

看看写入的 dump.txt 文件，一堆乱七八糟的内容，这些都是 Python 保存的对象内部信息。

当我们要把对象从磁盘读到内存时，可以先把内容读到一个 bytes，然后用 pickle.loads()方法反序列化出对象，也可以直接用 pickle.load()方法从一个 file-like Object 中直接反序列化出对象。我们打开另一个 Python 命令行来反序列化刚才保存的对象：

```
>>> f = open('dump.txt', 'rb')
>>> d = pickle.load(f)
>>> f.close()
>>> d
{'age': 20, 'score': 88, 'name': 'Bob'}
```

变量的内容又回来了！

当然，这个变量和原来的变量是完全不相干的对象，它们只是内容相同而已。

Pickle 的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于 Python，**并且可能不同版本的 Python 彼此都不兼容，因此，只能用 Pickle 保存那些不重要的数据，不能成功地反序列化也没关系。**

JSON

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如 XML，但更好的方法是序列化为 JSON，因为 JSON 表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON 不仅是标准格式，并且比 XML 更快，而且可以直接在 Web 页面中读取，非常方便。

JSON 表示的对象就是标准的 JavaScript 语言的对象，JSON 和 Python 内置的数据类型对应如下：

JSON 类型	Python 类型
{}	dict

[]	list
"string"	str
1234.56	int 或 float
true/false	True/False
null	None

Python 内置的 json 模块提供了非常完善的 Python 对象到 JSON 格式的转换。我们先看看如何把 Python 对象变成一个 JSON：

```
>>> import json
>>> d = dict(name='Bob', age=20, score=88)
>>> json.dumps(d)
'{"age": 20, "score": 88, "name": "Bob"}'
```

dumps()方法返回一个 str，内容就是标准的 JSON。类似的，dump()方法可以直接把 JSON 写入一个 file-like Object。

要把 JSON 反序列化为 Python 对象，用 loads()或者对应的 load()方法，前者把 JSON 的字符串反序列化，后者从 file-like Object 中读取字符串并反序列化：

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'
>>> json.loads(json_str)
{'age': 20, 'score': 88, 'name': 'Bob'}
```

由于 JSON 标准规定 JSON 编码是 UTF-8，所以我们总是能正确地在 Python 的 str 与 JSON 的字符串之间转换。

进程和线程

真正的并行执行多任务只能在多核 CPU 上实现，但是，由于任务数量远远多于 CPU 的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

对于操作系统来说，一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程，打开一个记事本就启动了一个记事本进程，打开两个记事本就启动了两个记事本进程，打开一个 Word 就启动了一个 Word 进程。

有些进程还不止同时干一件事，比如 Word，它可以同时进行打字、拼写检查、打印等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。

由于每个进程至少要干一件事，所以，一个进程至少有一个线程。当然，像 Word 这种复杂的进程可以有多个线程，多个线程可以同时执行，多线程的执行方式和多进程是一样的，也是由操作系统在多个线程之间快速切换，让每个线程都短暂地交替运行，看起来就像同时执行一样。当然，真正地同时执行多线程需要多核 CPU 才可能实现。

总结一下就是，多任务的实现有 3 种方式：

- **多进程模式；**
- **多线程模式；**
- **多进程+多线程模式。**

小结

线程是最小的执行单元，而进程由至少一个线程组成。如何调度进程和线程，完全由操作系统决定，程序自己不能决定什么时候执行，执行多长时间。

多进程和多线程的程序涉及到同步、数据共享的问题，编写起来更复杂。

多进程

multiprocessing

如果你打算编写多进程的服务程序，Unix/Linux 无疑是正确的选择。由于 Windows 没有 fork 调用，难道在 Windows 上无法用 Python 编写多进程的程序？

由于 Python 是跨平台的，自然也应该提供一个跨平台的多进程支持。multiprocessing 模块就是跨平台版本的多进程模块。

multiprocessing 模块提供了一个 Process 类来代表一个进程对象，下面的例子演示了启动一个子进程并等待其结束：

```
from multiprocessing import Process
import os

# 子进程要执行的代码
def run_proc(name):
```

```
    print('Run child process %s (%s)...' % (name, os.getpid()))

if __name__=='__main__':
    print('Parent process %s.' % os.getpid())
    p = Process(target=run_proc, args=('test',))
    print('Child process will start.')
    p.start()
    p.join()
    print('Child process end.')
```

执行结果如下：

```
Parent process 928.
Process will start.
Run child process test (929)...
Process end.
```

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个 Process 实例，用 start() 方法启动，这样创建进程比 fork() 还要简单。

join() 方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

Pool

如果要启动大量的子进程，可以用进程池的方式批量创建子进程：

```
from multiprocessing import Pool
import os, time, random

def long_time_task(name):
    print('Run task %s (%s)...' % (name, os.getpid()))
    start = time.time()
    time.sleep(random.random() * 3)
    end = time.time()
    print('Task %s runs %0.2f seconds.' % (name, (end - start)))

if __name__=='__main__':
    print('Parent process %s.' % os.getpid())
    p = Pool(4)
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print('Waiting for all subprocesses done...')
```

```
    p.close()
    p.join()
    print('All subprocesses done.')
```

执行结果如下：

```
Parent process 669.
Waiting for all subprocesses done...
Run task 0 (671)...
Run task 1 (672)...
Run task 2 (673)...
Run task 3 (674)...
Task 2 runs 0.14 seconds.
Run task 4 (673)...
Task 1 runs 0.27 seconds.
Task 3 runs 0.86 seconds.
Task 0 runs 1.41 seconds.
Task 4 runs 1.91 seconds.
All subprocesses done.
```

代码解读：

对 Pool 对象调用 join()方法会等待所有子进程执行完毕，调用 join()之前必须先调用 close()，调用 close()之后就不能继续添加新的 Process 了。

请注意输出的结果，task 0，1，2，3 是立刻执行的，而 task 4 要等待前面某个 task 完成后才执行，这是因为 Pool 的默认大小在我的电脑上是 4，因此，最多同时执行 4 个进程。这是 Pool 有意设计的限制，并不是操作系统的限制。如果改成：

```
p = Pool(5)
```

就可以同时跑 5 个进程。

由于 Pool 的默认大小是 CPU 的核数，如果你不幸拥有 8 核 CPU，你要提交至少 9 个子进程才能看到上面的等待效果。

实验表明，将 Pool 大小设置成 CPU 个数最高效。

进程间通信

Process 之间肯定是需要通信的，操作系统提供了很多机制来实现进程间的通信。Python 的 multiprocessing 模块包装了底层的机制，提供了 Queue、Pipes 等多种方式来交换数据。

我们以 Queue 为例，在父进程中创建两个子进程，一个往 Queue 里写数据，一个从 Queue 里读数据：

```
from multiprocessing import Process, Queue
import os, time, random

# 写数据进程执行的代码:
def write(q):
    print('Process to write: %s' % os.getpid())
    for value in ['A', 'B', 'C']:
        print('Put %s to queue...' % value)
        q.put(value)
        time.sleep(random.random())

# 读数据进程执行的代码:
def read(q):
    print('Process to read: %s' % os.getpid())
    while True:
        value = q.get(True)
        print('Get %s from queue.' % value)

if __name__=='__main__':
    # 父进程创建 Queue，并传给各个子进程:
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    # 启动子进程 pw，写入:
    pw.start()
    # 启动子进程 pr，读取:
    pr.start()
    # 等待 pw 结束:
    pw.join()
    # pr 进程里是死循环，无法等待其结束，只能强行终止:
    pr.terminate()
```

多线程

多任务可以由多进程完成，也可以由一个进程内的多线程完成。

我们前面提到了进程是由若干线程组成的，一个进程至少有一个线程。

由于线程是操作系统直接支持的执行单元，因此，高级语言通常都内置多线程的支持，Python 也不例外，并且，Python 的线程是真正的 Posix Thread，而不是模拟出来的线程。

Python 的标准库提供了两个模块：_thread 和 threading，_thread 是低级模块，threading 是高级模块，对_thread 进行了封装。**绝大多数情况下，我们只需要使用 threading 这个高级模块。**

启动一个线程就是把一个函数传入并创建 Thread 实例，然后调用 start()开始执行：

```
import time, threading

# 新线程执行的代码:
def loop():
    print('thread %s is running...' % threading.current_thread().name)
    n = 0
    while n < 5:
        n = n + 1
        print('thread %s >>> %s' % (threading.current_thread().name, n))
        time.sleep(1)
    print('thread %s ended.' % threading.current_thread().name)

print('thread %s is running...' % threading.current_thread().name)
t = threading.Thread(target=loop, name='LoopThread')
t.start()
t.join()
print('thread %s ended.' % threading.current_thread().name)
```

执行结果如下：

```
thread MainThread is running...
thread LoopThread is running...
thread LoopThread >>> 1
thread LoopThread >>> 2
thread LoopThread >>> 3
thread LoopThread >>> 4
thread LoopThread >>> 5
thread LoopThread ended.
thread MainThread ended.
```

由于任何进程默认就会启动一个线程，我们把该线程称为主线程，主线程又可以启动新的线程，Python 的 threading 模块有个 current_thread()函数，它永远返回当前线程的实

例。主线程实例的名字叫 MainThread，子线程的名字在创建时指定，我们用 LoopThread 命名子线程。名字仅仅在打印时用来显示，完全没有其他意义，如果不起名字 Python 就自动给线程命名为 Thread-1，Thread-2……

Lock

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。

来看看多个线程同时操作一个变量怎么把内容给改乱了：

```
import time, threading

# 假定这是你的银行存款:
balance = 0

def change_it(n):
    # 先存后取，结果应该为0:
    global balance
    balance = balance + n
    balance = balance - n

def run_thread(n):
    for i in range(100000):
        change_it(n)

t1 = threading.Thread(target=run_thread, args=(5,))
t2 = threading.Thread(target=run_thread, args=(8,))
t1.start()
t2.start()
t1.join()
t2.join()
print(balance)
```

我们定义了一个共享变量 balance，初始值为 0，并且启动两个线程，先存后取，理论上结果应该为 0，但是，由于线程的调度是由操作系统决定的，当 t1、t2 交替执行时，只要循环次数足够多，balance 的结果就不一定是 0 了。

原因是因为高级语言的一条语句在 CPU 执行时是若干条语句，即使一个简单的计算：

```
balance = balance + n
```

也分两步：

1. 计算 balance + n，存入临时变量中；
2. 将临时变量的值赋给 balance。

也就是可以看成：

```
x = balance + n
balance = x
```

由于 x 是局部变量，两个线程各自都有自己的 x，当代码正常执行时：

```
初始值 balance = 0

t1: x1 = balance + 5 # x1 = 0 + 5 = 5
t1: balance = x1     # balance = 5
t1: x1 = balance - 5 # x1 = 5 - 5 = 0
t1: balance = x1     # balance = 0

t2: x2 = balance + 8 # x2 = 0 + 8 = 8
t2: balance = x2     # balance = 8
t2: x2 = balance - 8 # x2 = 8 - 8 = 0
t2: balance = x2     # balance = 0

结果 balance = 0
```

但是 t1 和 t2 是交替运行的，如果操作系统以下面的顺序执行 t1、t2：

```
初始值 balance = 0

t1: x1 = balance + 5  # x1 = 0 + 5 = 5

t2: x2 = balance + 8  # x2 = 0 + 8 = 8
t2: balance = x2      # balance = 8

t1: balance = x1      # balance = 5
t1: x1 = balance - 5  # x1 = 5 - 5 = 0
t1: balance = x1      # balance = 0

t2: x2 = balance - 8  # x2 = 0 - 8 = -8
t2: balance = x2   # balance = -8
```

```
结果 balance = -8
```

究其原因，是因为修改 balance 需要多条语句，而执行这几条语句时，线程可能中断，从而导致多个线程把同一个对象的内容改乱了。

两个线程同时一存一取，就可能导致余额不对，你肯定不希望你的银行存款莫名其妙地变成了负数，所以，我们必须确保一个线程在修改 balance 的时候，别的线程一定不能改。

如果我们要确保 balance 计算正确，就要给 change_it()上一把锁，当某个线程开始执行 change_it()时，我们说，该线程因为获得了锁，因此其他线程不能同时执行 change_it()，只能等待，直到锁被释放后，获得该锁以后才能改。**由于锁只有一个（都是 lock），无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成修改的冲突。创建一个锁就是通过 threading.Lock()来实现：**

```
balance = 0
lock = threading.Lock()

def run_thread(n):
    for i in range(100000):
        # 先要获取锁:
        lock.acquire()
        try:
            # 放心地改吧:
            change_it(n)
        finally:
            # 改完了一定要释放锁:
            lock.release()
```

当多个线程同时执行 lock.acquire()时，只有一个线程能成功地获取锁，然后继续执行代码，其他线程就继续等待直到获得锁为止。

获得锁的线程用完后一定要释放锁，否则那些苦苦等待锁的线程将永远等待下去，成为死线程。所以我们用 try...finally 来确保锁一定会被释放。

锁的好处就是确保了某段关键代码只能由一个线程从头到尾完整地执行，坏处当然也很多，首先是阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了。**其次，由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁，导致多个线程全部挂起，既不能执行，也无法结束，只能靠操作系统强制终止。**

多核 CPU

如果你不幸拥有一个多核 CPU，你肯定在想，多核应该可以同时执行多个线程。

如果写一个死循环的话，会出现什么情况呢？

打开 Mac OS X 的 Activity Monitor，或者 Windows 的 Task Manager，都可以监控某个进程的 CPU 使用率。

我们可以监控到一个死循环线程会 100%占用一个 CPU。

如果有两个死循环线程，在多核 CPU 中，可以监控到会占用 200%的 CPU，也就是占用两个 CPU 核心。

要想把 N 核 CPU 的核心全部跑满，就必须启动 N 个死循环线程。

试试用 Python 写个死循环：

```
import threading, multiprocessing

def loop():
    x = 0
    while True:
        x = x ^ 1

for i in range(multiprocessing.cpu_count()):
    t = threading.Thread(target=loop)
    t.start()
```

启动与 CPU 核心数量相同的 N 个线程，在 4 核 CPU 上可以监控到 CPU 占用率仅有 102%，也就是仅使用了一核。

但是用 C、C++或 Java 来改写相同的死循环，直接可以把全部核心跑满，4 核就跑到 400%，8 核就跑到 800%，为什么 Python 不行呢？

因为 Python 的线程虽然是真正的线程，但解释器执行代码时，有一个 GIL 锁：Global Interpreter Lock，任何 Python 线程执行前，必须先获得 GIL 锁，然后，每执行 100 条字节码，解释器就自动释放 GIL 锁，让别的线程有机会执行。这个 GIL 全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在 Python 中只能交替执行，即使 100 个线程跑在 100 核 CPU 上，也只能用到 1 个核。

因为有 GIL 锁，每个进程里面有一个 GIL 锁，一个进程里面运行多线程，线程必须获得 GIL 锁才能运行，所以在 Pyhton 里面多线程运行是假的，一个核只能运行一个进程的某个线程，不能两个核同时运行一个进程里面的两个线程。只能两个核运行两个进程，达到并发效用！！！

GIL 是 Python 解释器设计的历史遗留问题，通常我们用的解释器是官方实现的 CPython，要真正利用多核，除非重写一个不带 GIL 的解释器。

所以，在 Python 中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那只能通过 C 扩展来实现，不过这样就失去了 Python 简单易用的特点。

不过，也不用过于担心，**Python 虽然不能利用多线程实现多核任务，但可以通过多进程实现多核任务。多个 Python 进程有各自独立的 GIL 锁，互不影响。**

ThreadLocal

在多线程环境下，每个线程都有自己的数据。一个线程使用自己的局部变量比使用全局变量好，因为局部变量只有线程自己能看见，不会影响其他线程，而全局变量的修改必须加锁。

但是局部变量也有问题，就是在函数调用的时候，传递起来很麻烦：

```
def process_student(name):
    std = Student(name)
    # std 是局部变量，但是每个函数都要用它，因此必须传进去：
    do_task_1(std)
    do_task_2(std)

def do_task_1(std):
    do_subtask_1(std)
    do_subtask_2(std)

def do_task_2(std):
    do_subtask_2(std)
    do_subtask_2(std)
```

每个函数一层一层调用都这么传参数那还得了？用全局变量？也不行，因为每个线程处理不同的 Student 对象，不能共享。

如果用一个全局 dict 存放所有的 Student 对象，然后以 thread 自身作为 key 获得线程对应的 Student 对象如何？

```
global_dict = {}

def std_thread(name):
    std = Student(name)
    # 把 std 放到全局变量 global_dict 中：
    global_dict[threading.current_thread()] = std
```

```
    do_task_1()
    do_task_2()

def do_task_1():
    # 不传入 std，而是根据当前线程查找：
    std = global_dict[threading.current_thread()]
    ...

def do_task_2():
    # 任何函数都可以查找出当前线程的 std 变量：
    std = global_dict[threading.current_thread()]
    ...
```

这种方式理论上是可行的，它最大的优点是消除了 std 对象在每层函数中的传递问题，但是，每个函数获取 std 的代码有点丑。

有没有更简单的方式？

ThreadLocal 应运而生，不用查找 dict，ThreadLocal 帮你自动做这件事：

```
import threading

# 创建全局 ThreadLocal 对象:
local_school = threading.local()

def process_student():
    # 获取当前线程关联的 student:
    std = local_school.student
    print('Hello, %s (in %s)' % (std, threading.current_thread().name))

def process_thread(name):
    # 绑定 ThreadLocal 的 student:
    local_school.student = name
    process_student()

t1 = threading.Thread(target= process_thread, args=('Alice',), name='Thread-A')
t2 = threading.Thread(target= process_thread, args=('Bob',), name='Thread-B')
t1.start()
t2.start()
t1.join()
```

```
t2.join()
```

执行结果：

```
Hello, Alice (in Thread-A)
Hello, Bob (in Thread-B)
```

全局变量 local_school 就是一个 ThreadLocal 对象，每个 Thread 对它都可以读写 student 属性，但互不影响。你可以把 local_school 看成全局变量，但每个属性如 local_school.student 都是线程的局部变量，可以任意读写而互不干扰，也不用管理锁的问题，ThreadLocal 内部会处理。

可以理解为全局变量 local_school 是一个 dict，不但可以用 local_school.student，还可以绑定其他变量，如 local_school.teacher 等等。

ThreadLocal 最常用的地方就是为每个线程绑定一个数据库连接，HTTP 请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

小结

一个 ThreadLocal 变量虽然是全局变量，但每个线程都只能读写自己线程的独立副本，互不干扰。ThreadLocal 解决了参数在一个线程中各个函数之间互相传递的问题。

进程 vs. 线程

我们介绍了多进程和多线程，这是实现多任务最常用的两种方式。现在，我们来讨论一下这两种方式的优缺点。

首先，要实现多任务，通常我们会设计 Master-Worker 模式，Master 负责分配任务，Worker 负责执行任务，因此，多任务环境下，通常是一个 Master，多个 Worker。

如果用多进程实现 Master-Worker，主进程就是 Master，其他进程就是 Worker。

如果用多线程实现 Master-Worker，主线程就是 Master，其他线程就是 Worker。

多进程模式最大的优点就是稳定性高，因为一个子进程崩溃了，不会影响主进程和其他子进程。（当然主进程挂了所有进程就全挂了，但是 Master 进程只负责分配任务，挂掉的概率低）著名的 Apache 最早就是采用多进程模式。

多进程模式的缺点是创建进程的代价大，在 Unix/Linux 系统下，用 fork 调用还行，在 Windows 下创建进程开销巨大。另外，操作系统能同时运行的进程数也是有限的，在内存和 CPU 的限制下，如果有几千个进程同时运行，操作系统连调度都会成问题。

多线程模式通常比多进程快一点，但是也快不到哪去，而且，多线程模式致命的缺点就是任何一个线程挂掉都可能直接造成整个进程崩溃，因为所有线程共享进程的内存。在 Windows 上，如果一个线程执行的代码出了问题，你经常可以看到这样的提示：“该程序执行了非法操作，即将关闭”，其实往往是某个线程出了问题，但是操作系统会强制结束整个进程。

在 Windows 下，多线程的效率比多进程要高，所以微软的 IIS 服务器默认采用多线程模式。由于多线程存在稳定性的问题，IIS 的稳定性就不如 Apache。为了缓解这个问题，IIS 和 Apache 现在又有多进程+多线程的混合模式，真是把问题越搞越复杂。

线程切换

无论是多进程还是多线程，只要数量一多，效率肯定上不去。

所以，多任务一旦多到一个限度，就会消耗掉系统所有的资源，结果效率急剧下降，所有任务都做不好。

计算密集型 vs. IO 密集型

是否采用多任务的第二个考虑是任务的类型。我们可以把任务分为计算密集型和 IO 密集型。

计算密集型任务的特点是要进行大量的计算，消耗 CPU 资源，比如计算圆周率、对视频进行高清解码等等，全靠 CPU 的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU 执行任务的效率就越低，所以，**要最高效地利用 CPU，计算密集型任务同时进行的数量应当等于 CPU 的核心数。**

计算密集型任务由于主要消耗 CPU 资源，因此，代码运行效率至关重要。Python 这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用 C 语言编写。

第二种任务的类型是 IO 密集型，涉及到网络、磁盘 IO 的任务都是 IO 密集型任务，这类任务的特点是 CPU 消耗很少，任务的大部分时间都在等待 IO 操作完成（因为 IO 的速度远远低于 CPU 和内存的速度）。对于 IO 密集型任务，任务越多，CPU 效率越高，但也有一个限度。常见的大部分任务都是 IO 密集型任务，比如 Web 应用。对于 IO 密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C 语言最差。

异步 IO

考虑到 CPU 和 IO 之间巨大的速度差异，一个任务在执行的过程中大部分时间都在等待 IO 操作，单进程单线程模型会导致别的任务无法并行执行，因此，我们才需要多进程模型或者多线程模型来支持多任务并发执行。

现代操作系统对 IO 操作已经做了巨大的改进，最大的特点就是支持异步 IO。如果充分利用操作系统提供的异步 IO 支持，就可以用单进程单线程模型来执行多任务，这种全新的模型

称为事件驱动模型，Nginx 就是支持异步 IO 的 Web 服务器，它在单核 CPU 上采用单进程模型就可以高效地支持多任务。在多核 CPU 上，可以运行多个进程（数量与 CPU 核心数相同），充分利用多核 CPU。由于系统总的进程数量十分有限，因此操作系统调度非常高效。用异步 IO 编程模型来实现多任务是一个主要的趋势。

对应到 Python 语言，单进程的异步编程模型称为协程，有了协程的支持，就可以基于事件驱动编写高效的多任务程序。我们会在后面讨论如何编写协程。

正则表达式

字符串是编程时涉及到的最多的一种数据结构，对字符串进行操作的需求几乎无处不在。比如判断一个字符串是否是合法的 Email 地址，虽然可以编程提取@前后的子串，再分别判断是否是单词和域名，但这样做不但麻烦，而且代码难以复用。

正则表达式是一种用来匹配字符串的强有力的武器。它的设计思想是用一种描述性的语言来给字符串定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的。

所以我们判断一个字符串是否是合法的 Email 的方法是：

1. 创建一个匹配 Email 的正则表达式；
2. 用该正则表达式去匹配用户的输入来判断是否合法。

因为正则表达式也是用字符串表示的，所以，我们要首先了解如何用字符来描述字符。

在正则表达式中，如果直接给出字符，就是精确匹配。用\d 可以匹配一个数字，\w 可以匹配一个字母或数字，所以：

- '00\d'可以匹配'007'，但无法匹配'00A'；
- '\d\d\d'可以匹配'010'；
- '\w\w\d'可以匹配'py3'；

. 可以匹配任意字符，所以：

- 'py.'可以匹配'pyc'、'pyo'、'py!'等等。

要匹配变长的字符，在正则表达式中，用*表示任意个字符（包括 0 个），用+表示至少一个字符，用?表示 0 个或 1 个字符，用{n}表示 n 个字符，用{n,m}表示 n-m 个字符：

来看一个复杂的例子：\d{3}\s+\d{3,8}。

我们来从左到右解读一下：

1. \d{3}表示匹配 3 个数字，例如'010'；

2. \s 可以匹配一个空格（也包括 Tab 等空白符），所以\s+表示至少有一个空格，例如匹配' ', ' '等；
3. \d{3,8}表示 3-8 个数字，例如'1234567'。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

如果要匹配'010-12345'这样的号码呢？由于'-'是特殊字符，在正则表达式中，要用'\'转义，所以，上面的正则是\d{3}\-\d{3,8}。

但是，仍然无法匹配'010 - 12345'，因为带有空格。所以我们需要更复杂的匹配方式。

进阶

要做更精确地匹配，可以用[]表示范围，比如：

- [0-9a-zA-Z_]可以匹配一个数字、字母或者下划线；
- [0-9a-zA-Z_]+可以匹配至少由一个数字、字母或者下划线组成的字符串，比如'a100'，'0_Z'，'Py3000'等等；
- [a-zA-Z_][0-9a-zA-Z_]*可以匹配由字母或下划线开头，后接任意个由一个数字、字母或者下划线组成的字符串，也就是 Python 合法的变量；
- [a-zA-Z_][0-9a-zA-Z_]{0, 19}更精确地限制了变量的长度是 1-20 个字符（前面 1 个字符+后面最多 19 个字符）。

A|B 可以匹配 A 或 B，所以[P|p]ython 可以匹配'Python'或者'python'。

^表示行的开头，^\d 表示必须以数字开头。

$表示行的结束，\d$表示必须以数字结束。

你可能注意到了，py 也可以匹配'python'，但是加上^py$就变成了整行匹配，就只能匹配'py'了。

re 模块

有了准备知识，我们就可以在 Python 中使用正则表达式了。Python 提供 re 模块，包含所有正则表达式的功能。由于 Python 的字符串本身也用\转义，所以要特别注意：

```
s = 'ABC\\-001' # Python 的字符串
# 对应的正则表达式字符串变成：
# 'ABC\-001'
```

因此我们强烈建议使用 Python 的 r 前缀，就不用考虑转义的问题了：

```
s = r'ABC\-001' # Python 的字符串
# 对应的正则表达式字符串不变：
# 'ABC\-001'
```

先看看如何判断正则表达式是否匹配：

```
>>> import re
>>> re.match(r'^\d{3}\-\d{3,8}$', '010-12345')
<_sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> re.match(r'^\d{3}\-\d{3,8}$', '010 12345')
>>>
```

match()方法判断是否匹配，如果匹配成功，返回一个 Match 对象，否则返回 None。常见的判断方法就是：

```
test = '用户输入的字符串'
if re.match(r'正则表达式', test):
    print('ok')
else:
    print('failed')
```

切分字符串

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代码：

```
>>> 'a b   c'.split(' ')
['a', 'b', '', '', 'c']
```

嗯，无法识别连续的空格，用正则表达式试试：

```
>>> re.split(r'\s+', 'a b   c')
['a', 'b', 'c']
```

无论多少个空格都可以正常分割。加入,试试：

```
>>> re.split(r'[\s\,]+', 'a,b, c  d')
['a', 'b', 'c', 'd']
```

再加入;试试：

```
>>> re.split(r'[\s\,\;]+', 'a,b;; c  d')
['a', 'b', 'c', 'd']
```

如果用户输入了一组标签，下次记得用正则表达式来把不规范的输入转化成正确的数组。

分组

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用()表示的就是要提取的分组（Group）。比如：

^(\d{3})-(\d{3,8})$分别定义了两个组，可以直接从匹配的字符串中提取出区号和本地号码：

```
>>> m = re.match(r'^(\d{3})-(\d{3,8})$', '010-12345')
>>> m
<_sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> m.group(0)
'010-12345'
>>> m.group(1)
'010'
>>> m.group(2)
'12345'
```

如果正则表达式中定义了组，就可以在 Match 对象上用 group()方法提取出子串来。

注意到 group(0)永远是原始字符串，group(1)、group(2)……表示第 1、2、……个子串。

提取子串非常有用。来看一个更凶残的例子：

```
>>> t = '19:05:30'
>>> m = re.match(r'^(0[0-9]|1[0-9]|2[0-3]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|[0-9])$', t)
>>> m.groups()
('19', '05', '30')
```

这个正则表达式可以直接识别合法的时间。但是有些时候，用正则表达式也无法做到完全验证，比如识别日期：

```
'^(0[1-9]|1[0-2]|[0-9])-(0[1-9]|1[0-9]|2[0-9]|3[0-1]|[0-9])$'
```

对于'2-30'，'4-31'这样的非法日期，用正则还是识别不了，或者说写出来非常困难，这时就需要程序配合识别了。

贪婪匹配

最后需要特别指出的是，正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的0：

```
>>> re.match(r'^(\d+)(0*)$', '102300').groups()
('102300', '')
```

由于\d+采用贪婪匹配，直接把后面的0全部匹配了，结果0*只能匹配空字符串了。

必须让\d+采用非贪婪匹配（也就是尽可能少匹配），才能把后面的0匹配出来，加个?就可以让\d+采用非贪婪匹配：

```
>>> re.match(r'^(\d+?)(0*)$', '102300').groups()
('1023', '00')
```

编译

当我们在Python中使用正则表达式时，re模块内部会干两件事情：

1. 编译正则表达式，如果正则表达式的字符串本身不合法，会报错；
2. 用编译后的正则表达式去匹配字符串。

如果一个正则表达式要重复使用几千次，出于效率的考虑，我们可以预编译该正则表达式，接下来重复使用时就不需要编译这个步骤了，直接匹配：

```
>>> import re
# 编译:
>>> re_telephone = re.compile(r'^(\d{3})-(\d{3,8})$')
# 使用：
>>> re_telephone.match('010-12345').groups()
('010', '12345')
>>> re_telephone.match('010-8086').groups()
('010', '8086')
```

编译后生成 Regular Expression 对象，由于该对象自己包含了正则表达式，所以调用对应的方法时不用给出正则字符串。

网络编程

TCP/IP 简介

IP 协议负责把数据从一台计算机通过网络发送到另一台计算机。数据被分割成一小块一小块，然后通过 IP 包发送出去。由于互联网链路复杂，两台计算机之间经常有多条线路，因此，路由器就负责决定如何把一个 IP 包转发出去。IP 包的特点是按块发送，途径多个路由，但不保证能到达，也不保证顺序到达。

TCP 协议则是建立在 IP 协议之上的。TCP 协议负责在两台计算机之间建立可靠连接，保证数据包按顺序到达。TCP 协议会通过握手建立连接，然后，对每个 IP 包编号，确保对方按顺序收到，如果包丢掉了，就自动重发。

许多常用的更高级的协议都是建立在 TCP 协议基础上的，比如用于浏览器的 HTTP 协议、发送邮件的 SMTP 协议等。

一个 IP 包除了包含要传输的数据外，还包含源 IP 地址和目标 IP 地址，源端口和目标端口。

端口有什么作用？在两台计算机通信时，只发 IP 地址是不够的，因为同一台计算机上跑着多个网络程序。一个 IP 包来了之后，到底是交给浏览器还是 QQ，就需要端口号来区分。每个网络程序都向操作系统申请唯一的端口号，这样，两个进程在两台计算机之间建立网络连接就需要各自的 IP 地址和各自的端口号。

TCP 编程

Socket 是网络编程的一个抽象概念。通常我们用一个 Socket 表示“打开了一个网络链接”，而打开一个 Socket 需要知道目标计算机的 IP 地址和端口号，再指定协议类型即可。

客户端

大多数连接都是可靠的 TCP 连接。创建 TCP 连接时，主动发起连接的叫客户端，被动响应连接的叫服务器。

所以，我们要创建一个基于 TCP 连接的 Socket，可以这样做：

```
# 导入 socket 库:
import socket
```

```
# 创建一个 socket:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 建立连接:
s.connect(('www.sina.com.cn', 80))
```

创建 Socket 时，AF_INET 指定使用 IPv4 协议，如果要用更先进的 IPv6，就指定为 AF_INET6。SOCK_STREAM 指定使用面向流的 TCP 协议，这样，一个 Socket 对象就创建成功，但是还没有建立连接。

客户端要主动发起 TCP 连接，必须知道服务器的 IP 地址和端口号。新浪网站的 IP 地址可以用域名 www.sina.com.cn 自动转换到 IP 地址，但是怎么知道新浪服务器的端口号呢？

答案是作为服务器，提供什么样的服务，端口号就必须固定下来。由于我们想要访问网页，因此新浪提供网页服务的服务器必须把端口号固定在 80 端口，因为 80 端口是 Web 服务的标准端口。其他服务都有对应的标准端口号，例如 SMTP 服务是 25 端口，FTP 服务是 21 端口，等等。端口号小于 1024 的是 Internet 标准服务的端口，端口号大于 1024 的，可以任意使用。

建立 TCP 连接后，我们就可以向新浪服务器发送请求，要求返回首页的内容：

```
# 发送数据:
s.send(b'GET / HTTP/1.1\r\nHost: www.sina.com.cn\r\nConnection: close\r\n\r\n')
```

TCP 连接创建的是双向通道，双方都可以同时给对方发数据。但是谁先发谁后发，怎么协调，要根据具体的协议来决定。例如，HTTP 协议规定客户端必须先发请求给服务器，服务器收到后才发数据给客户端。

发送的文本格式必须符合 HTTP 标准，如果格式没问题，接下来就可以接收新浪服务器返回的数据了：

```
# 接收数据:
buffer = []
while True:
    # 每次最多接收 1k 字节:
    d = s.recv(1024)
    if d:
        buffer.append(d)
    else:
        break
data = b''.join(buffer)
```

接收数据时，调用 recv(max)方法，一次最多接收指定的字节数，因此，在一个 while 循环中反复接收，直到 recv()返回空数据，表示接收完毕，退出循环。

当我们接收完数据后，调用 close()方法关闭 Socket，这样，一次完整的网络通信就结束了：

```
# 关闭连接:
s.close()
```

接收到的数据包括 HTTP 头和网页本身，我们只需要把 HTTP 头和网页分离一下，把 HTTP 头打印出来，网页内容保存到文件：

```
header, html = data.split(b'\r\n\r\n', 1)
print(header.decode('utf-8'))
# 把接收的数据写入文件:
with open('sina.html', 'wb') as f:
    f.write(html)
```

现在，只需要在浏览器中打开这个 sina.html 文件，就可以看到新浪的首页了。

服务器

和客户端编程相比，服务器编程就要复杂一些。

服务器进程首先要绑定一个端口并监听来自其他客户端的连接。如果某个客户端连接过来了，服务器就与该客户端建立 Socket 连接，随后的通信就靠这个 Socket 连接了。

所以，服务器会打开固定端口（比如 80）监听，每来一个客户端连接，就创建该 Socket 连接。由于服务器会有大量来自客户端的连接，所以，服务器要能够区分一个 Socket 连接是和哪个客户端绑定的。一个 Socket 依赖 4 项：服务器地址、服务器端口、客户端地址、客户端端口来唯一确定一个 Socket。

但是服务器还需要同时响应多个客户端的请求，所以，每个连接都需要一个新的进程或者新的线程来处理，否则，服务器一次就只能服务一个客户端了。

我们来编写一个简单的服务器程序，它接收客户端连接，把客户端发过来的字符串加上 Hello 再发回去。

首先，创建一个基于 IPv4 和 TCP 协议的 Socket：

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

然后，我们要绑定监听的地址和端口。服务器可能有多块网卡，可以绑定到某一块网卡的 IP 地址上，也可以用 0.0.0.0 绑定到所有的网络地址，还可以用 127.0.0.1 绑定到本机地址。127.0.0.1 是一个特殊的 IP 地址，表示本机地址，如果绑定到这个地址，客户端必须同时在本机运行才能连接，也就是说，外部的计算机无法连接进来。

端口号需要预先指定。因为我们写的这个服务不是标准服务，所以用 9999 这个端口号。请注意，小于 1024 的端口号必须要有管理员权限才能绑定：

紧接着，调用 listen()方法开始监听端口，传入的参数指定等待连接的最大数量：

```
s.listen(5)
print('Waiting for connection...')
```

接下来，服务器程序通过一个永久循环来接受来自客户端的连接，accept()会等待并返回一个客户端的连接:

```
while True:
    # 接受一个新连接:
    sock, addr = s.accept()
    # 创建新线程来处理 TCP 连接:
    t = threading.Thread(target=tcplink, args=(sock, addr))
    t.start()
```

每个连接都必须创建新线程（或进程）来处理，否则，单线程在处理连接的过程中，无法接受其他客户端的连接。

电子邮件

一封电子邮件的旅程就是：

```
发件人 -> MUA -> MTA -> MTA -> 若干个 MTA -> MDA <- MUA <- 收件人
```

有了上述基本概念，要编写程序来发送和接收邮件，本质上就是：

1. 编写 MUA 把邮件发到 MTA；
2. 编写 MUA 从 MDA 上收邮件。

发邮件时，MUA 和 MTA 使用的协议就是 SMTP：Simple Mail Transfer Protocol，后面的 MTA 到另一个 MTA 也是用 SMTP 协议。

收邮件时，MUA 和 MDA 使用的协议有两种：POP：Post Office Protocol，目前版本是 3，俗称 POP3；IMAP：Internet Message Access Protocol，目前版本是 4，优点是不但能取邮件，还可以直接操作 MDA 上存储的邮件，比如从收件箱移到垃圾箱，等等。

邮件客户端软件在发邮件时，会让你先配置 SMTP 服务器，也就是你要发到哪个 MTA 上。假设你正在使用 163 的邮箱，你就不能直接发到新浪的 MTA 上，因为它只服务新浪的用户，所以，你得填 163 提供的 SMTP 服务器地址：smtp.163.com，为了证明你是 163 的用户，SMTP 服务器还要求你填写邮箱地址和邮箱口令，这样，MUA 才能正常地把 Email 通过 SMTP 协议发送到 MTA。

类似的，从 MDA 收邮件时，MDA 服务器也要求验证你的邮箱口令，确保不会有人冒充你收取你的邮件，所以，Outlook 之类的邮件客户端会要求你填写 POP3 或 IMAP 服务器地址、邮箱地址和口令，这样，MUA 才能顺利地通过 POP 或 IMAP 协议从 MDA 取到邮件。

在使用 Python 收发邮件前，请先准备好至少两个电子邮件，如 xxx@163.com，xxx@sina.com，xxx@qq.com 等，注意两个邮箱不要用同一家邮件服务商。

最后*特别注意*，目前大多数邮件服务商都需要手动打开 SMTP 发信和 POP 收信的功能，否则只允许在网页登录.

SMTP 发送邮件

SMTP 是发送邮件的协议，Python 内置对 SMTP 的支持，可以发送纯文本邮件、HTML 邮件以及带附件的邮件。

Python 对 SMTP 支持有 smtplib 和 email 两个模块，email 负责构造邮件，smtplib 负责发送邮件。

首先，我们来构造一个最简单的纯文本邮件：

```
from email.mime.text import MIMEText
msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')
```

注意到构造 MIMEText 对象时，第一个参数就是邮件正文，第二个参数是 MIME 的 subtype，传入'plain'表示纯文本，最终的 MIME 就是'text/plain'，最后一定要用 utf-8 编码保证多语言兼容性。

然后，通过 SMTP 发出去：

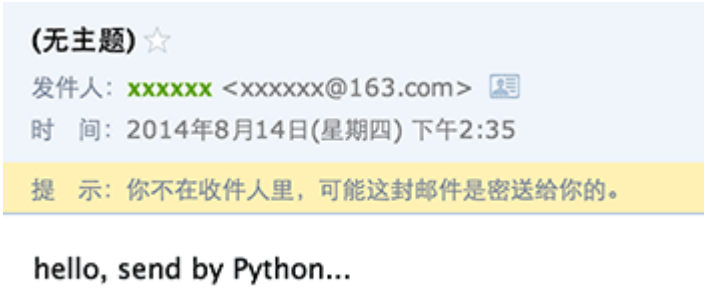
```
# 输入 Email 地址和口令:
from_addr = input('From: ')
password = input('Password: ')
# 输入收件人地址:
```

```
to_addr = input('To: ')
# 输入 SMTP 服务器地址:
smtp_server = input('SMTP server: ')

import smtplib
server = smtplib.SMTP(smtp_server, 25) # SMTP 协议默认端口是 25
server.set_debuglevel(1)
server.login(from_addr, password)
server.sendmail(from_addr, [to_addr], msg.as_string())
server.quit()
```

我们用 set_debuglevel(1)就可以打印出和 SMTP 服务器交互的所有信息。SMTP 协议就是简单的文本命令和响应。login()方法用来登录 SMTP 服务器，sendmail()方法就是发邮件，由于可以一次发给多个人，所以传入一个 list，邮件正文是一个 str，as_string()把 MIMEText 对象变成 str。

如果一切顺利，就可以在收件人信箱中收到我们刚发送的 Email：



仔细观察，发现如下问题：

邮件没有主题；

收件人的名字没有显示为友好的名字，比如 Mr Green <green@example.com>；

明明收到了邮件，却提示不在收件人中。

这是因为邮件主题、如何显示发件人、收件人等信息并不是通过 SMTP 协议发给 MTA，而是包含在发给 MTA 的文本中的，所以，我们必须把 From、To 和 Subject 添加到 MIMEText 中，才是一封完整的邮件：

```
from email import encoders
from email.header import Header
from email.mime.text import MIMEText
from email.utils import parseaddr, formataddr
```

```
import smtplib

def _format_addr(s):
    name, addr = parseaddr(s)
    return formataddr((Header(name, 'utf-8').encode(), addr))

from_addr = input('From: ')
password = input('Password: ')
to_addr = input('To: ')
smtp_server = input('SMTP server: ')

msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')
msg['From'] = _format_addr('Python 爱好者 <%s>' % from_addr)
msg['To'] = _format_addr('管理员 <%s>' % to_addr)
msg['Subject'] = Header('来自 SMTP 的问候……', 'utf-8').encode()

server = smtplib.SMTP(smtp_server, 25)
server.set_debuglevel(1)
server.login(from_addr, password)
server.sendmail(from_addr, [to_addr], msg.as_string())
server.quit()
```

我们编写了一个函数_format_addr()来格式化一个邮件地址。注意不能简单地传入 name <addr@example.com>，因为如果包含中文，需要通过 Header 对象进行编码。

msg['To']接收的是字符串而不是 list，如果有多个邮件地址，用,分隔即可。

发送附件

如果 Email 中要加上附件怎么办？带附件的邮件可以看做包含若干部分的邮件：文本和各个附件本身，所以，可以构造一个 MIMEMultipart 对象代表邮件本身，然后往里面加上一个 MIMEText 作为邮件正文，再继续往里面加上表示附件的 MIMEBase 对象即可：

```
# 邮件对象:
msg = MIMEMultipart()
msg['From'] = _format_addr('Python 爱好者 <%s>' % from_addr)
msg['To'] = _format_addr('管理员 <%s>' % to_addr)
msg['Subject'] = Header('来自 SMTP 的问候……', 'utf-8').encode()

# 邮件正文是 MIMEText:
msg.attach(MIMEText('send with file...', 'plain', 'utf-8'))
```

```
# 添加附件就是加上一个MIMEBase，从本地读取一个图片:
with open('/Users/michael/Downloads/test.png', 'rb') as f:
    # 设置附件的MIME和文件名，这里是png类型:
    mime = MIMEBase('image', 'png', filename='test.png')
    # 加上必要的头信息:
    mime.add_header('Content-Disposition', 'attachment', filename='test.png')
    mime.add_header('Content-ID', '<0>')
    mime.add_header('X-Attachment-Id', '0')
    # 把附件的内容读进来:
    mime.set_payload(f.read())
    # 用Base64编码:
    encoders.encode_base64(mime)
    # 添加到MIMEMultipart:
    msg.attach(mime)
```

小结

使用 Python 的 smtplib 发送邮件十分简单，只要掌握了各种邮件类型的构造方法，正确设置好邮件头，就可以顺利发出。

构造一个邮件对象就是一个 Messag 对象，如果构造一个 MIMEText 对象，就表示一个文本邮件对象，如果构造一个 MIMEImage 对象，就表示一个作为附件的图片，要把多个对象组合起来，就用 MIMEMultipart 对象，而 MIMEBase 可以表示任何对象。它们的继承关系如下：

```
Message
+- MIMEBase
   +- MIMEMultipart
   +- MIMENonMultipart
      +- MIMEMessage
      +- MIMEText
      +- MIMEImage
```

这种嵌套关系就可以构造出任意复杂的邮件。你可以通过 email.mime 文档查看它们所在的包以及详细的用法。