

# 1. Python基础

Python是一种计算机编程语言。计算机编程语言和我们日常使用的自然语言有所不同，最大的区别就是，自然语言在不同的语境下有不同的理解，而计算机要根据编程语言执行任务，就必须保证编程语言写出的程序决不能有歧义，所以，任何一种编程语言都有自己的一套语法，编译器或者解释器就是负责把符合语法的程序代码转换成CPU能够执行的机器码，然后执行。Python也不例外。

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。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写出缩进较少的代码，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。

**Python程序是大小写敏感的**，如果写错了大小写，程序会报错。

Python使用4个空格的缩进来组织代码块。

在文本编辑器中，需要设置把Tab自动转换为4个空格，确保不混用Tab和空格。

## 1.1. 数据类型和变量

### 1.1.1. 数据类型

计算机顾名思义就是可以做数学计算的机器，因此，计算机程序理所当然地可以处理各种数值。但是，计算机能处理的远不止数值，还可以处理文本、图形、音频、视频、网页等各种各样的数据，不同的数据，需要定义不同的数据类型。在Python中，能够直接处理的数据类型有以下几种：

#### 1.1.1.1. 整数

Python可以处理任意大小的整数，当然包括负整数，在程序中的表示方法和数学上的写法一模一样，例如：1，100，-8080，0，等等。

计算机由于使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用**0x前缀**和**0-9，a-f表示**，例如：0xff00，0xa5b4c3d2，等等。

#### 1.1.1.2. 浮点数

浮点数也就是小数，之所以称为浮点数，是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的，比如，1.23x109和12.3x108是完全相等的。浮点数可以用数学写法，如1.23，3.14，-9.01，等等。但是对于很大或很小的浮点数，就必须用科学计数法表示，把10用e替代，1.23x109就是1.23e9，或者12.3e8，0.000012可以写成1.2e-5，等等。

整数和浮点数在计算机内部存储的方式是不同的，整数运算永远是精确的（除法也是精确的），而浮点数运算则可能会有四舍五入的误差。

#### 1.1.1.3. 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"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\，可以在Python的交互式命令行用print()打印字符串看看：

```
>>> print('I\'m ok.')
I'm ok.
>>> print('I\'m learning\nPython.')
I'm learning
Python.
>>> print('\n\n')
\
\
```

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：

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

如果字符串内部有很多换行，用\n写在一行里不好阅读，为了简化，Python允许用"""..."""的格式表示多行内容：

```
>>> print('''line1
... line2
```

```
... line3''')
line1
line2
line3
```

上面是在交互式命令行内输入，注意在输入多行内容时，提示符由>>>变为...，提示可以接着上一行输入，注意...是提示符，不是代码的一部分：

□

当输入完结束符`` 和括号)后，执行该语句并打印结果。

如果写成程序并保存为.py文件，就是：

```
print(''line1
line2
line3'')
```

多行字符串"..."还可以在前面加上r使用。

#### 1.1.1.4. 布尔值

布尔值和布尔代数的表示完全一致，一个布尔值只有True、False两种值，要么是True，要么是False，在Python中，可以直接用True、False表示布尔值（请注意大小写），也可以通过布尔运算计算出来：

```
>>> True
True
>>> False
False
>>> 3 > 2
True
>>> 3 > 5
False
```

布尔值可以用and、or和not运算。

and运算是与运算，只有所有都为True，and运算结果才是True：

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
>>> 5 > 3 and 3 > 1
True
```

or运算是或运算，只要其中有一个为True，or运算结果就是True：

```
>>> True or True
True
>>> True or False
True
>>> False or False
False
>>> 5 > 3 or 1 > 3
True
```

not运算是非运算，它是一个单目运算符，把True变成False，False变成True：

```
>>> not True
False
>>> not False
True
>>> not 1 > 2
True
```

布尔值经常用在条件判断中，比如：

```
if age >= 18:
    print('adult')
else:
    print('teenager')
```

#### 1.1.1.5. 空值

空值是Python里一个特殊的值，用None表示。None不能理解为0，因为0是有意义的，而None是一个特殊的空值。

此外，Python还提供了列表、字典等多种数据类型，还允许创建自定义数据类型。

#### 1.1.1.6. 变量

变量的概念基本上和初中代数的方程变量是一致的，只是在计算机程序中，变量不仅可以是数字，还可以是任意数据类型。

变量在程序中就是用一个变量名表示了，变量名必须是大小写英文、数字和\_的组合，且不能用数字开头，比如：

```
a = 1
```

变量a是一个整数。

```
t_007 = 'T007'
```

变量t\_007是一个字符串。

```
Answer = True
```

变量Answer是一个布尔值True。

在Python中，等号=是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量。

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如Java是静态语言，赋值语句如下（// 表示注释）：

```
int a = 123; // a是整数类型变量
a = "ABC"; // 错误：不能把字符串赋给整型变量
```

和静态语言相比，动态语言更灵活，就是这个原因。

请不要把赋值语句的等号等同于数学的等号。比如下面的代码：

```
x = 10
x = x + 2
```

如果从数学上理解 $x = x + 2$ 那无论如何是不成立的，在程序中，赋值语句先计算右侧的表达式 $x + 2$ ，得到结果12，再赋给变量x。由于x之前的值是10，重新赋值后，x的值变成12。

最后，理解变量在计算机内存中的表示也非常重要。当我们写：

```
a = 'ABC'
```

时，Python解释器干了两件事情：

在内存中创建了一个'ABC'的字符串；

在内存中创建了一个名为a的变量，并把它指向'ABC'。

#### 1.1.1.7. 常量

所谓常量就是不能变的变量，比如常用的数学常数 $\pi$ 就是一个常量。在Python中，通常用全部大写的变量名表示常量：

PI = 3.14159265359

但事实上PI仍然是一个变量，**Python根本没有任何机制保证PI不会被改变**，所以，用全部大写的变量名表示常量只是一个习惯上的用法。

最后解释一下整数的除法为什么也是精确的。在Python中，有两种除法，一种除法是/：

```
>>> 10 / 3
3.3333333333333335
```

/除法计算结果是浮点数，即使是两个整数恰好整除，结果也是浮点数：

```
>>> 9 / 3
3.0
```

还有一种除法是//，称为地板除，两个整数的除法仍然是整数：

```
>>> 10 // 3
3
```

整数的地板除//永远是整数，即使除不尽。要做精确的除法，使用/就可以。

因为//除法只取结果的整数部分，所以Python还提供一个余数运算，可以得到两个整数相除的余数：

```
>>> 10 % 3
1
```

无论整数做//除法还是取余数，结果永远是整数，所以，整数运算结果永远是精确的。

Python的整数没有大小限制，而某些语言的整数根据其存储长度是有大小限制的，例如Java对32位整数的范围限制在-2147483648-2147483647。

Python的浮点数也没有大小限制，但是超出一定范围就直接表示为inf（无限大）。

## 1.2. 字符串和编码

### 1.2.1. 字符编码

因为计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。最早的计算机在设计时采用8个比特（bit）作为一个字节（byte），所以，一个字节能表示的最大的整数就是255（二进制11111111=十进制255），如果要表示更大的整数，就必须用更多的字节。比如两个字节可以表示的最大整数是65535，4个字节可以表示的最大整数是4294967295。

由于计算机是美国人发明的，因此，最早只有127个字符被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为ASCII编码，比如大写字母A的编码是65，小写字母z的编码是122。

但是要处理中文显然一个字节是不够的，至少需要两个字节，而且还不能和ASCII编码冲突，所以，中国制定了GB2312编码，用来把中文编进去。

全世界有上百种语言，日本把日文编到Shift\_JIS里，韩国把韩文编到Euc-kr里，各国各有各的标准，就会不可避免地出现冲突，结果就是，在多语言混合的文本中，显示出来会有乱码。

因此，Unicode应运而生。Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。

Unicode标准也在不断发展，但最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要4个字节）。现代操作系统和大多数编程语言都直接支持Unicode。

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。

字母A用ASCII编码是十进制的65，二进制的01000001；

字符0用ASCII编码是十进制的48，二进制的00110000，注意字符'0'和整数0是不同的；

汉字中已经超出了ASCII编码的范围，用Unicode编码是十进制的20013，二进制的01001110 00101101。

如果把ASCII编码的A用Unicode编码，只需要在前面补0就可以，因此，A的Unicode编码是00000000 01000001。

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

从上面的表格还可以发现，UTF-8编码有一个额外的好处，就是ASCII编码实际上可以被看成是UTF-8编码的一部分，所以，大量只支持ASCII编码的历史遗留软件可以在UTF-8编码下继续工作。

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：

□

浏览网页的时候，服务器会把动态生成的Unicode内容转换为UTF-8再传输到浏览器：

□

很多网页的源码上会有类似的信息，表示该网页正是用的UTF-8编码。

### 1.2.2. Python的字符串

在最新的Python 3版本中，字符串是以Unicode编码的，也就是说，Python的字符串支持多语言，例如：

```
>>> print('包含中文的str')
包含中文的str
```

对于单个字符的编码，Python提供了ord()函数获取字符的整数表示，chr()函数把编码转换为对应的字符：

```
>>> ord('A')
65
>>> ord('中')
20013
>>> chr(66)
'B'
>>> chr(25991)
'文'
```

如果知道字符的整数编码，还可以用十六进制这么写str：

```
>>> '\u4e2d\u6587'
'中文'
```

两种写法完全是等价的。

由于Python的字符串类型是str，在内存中以Unicode表示，一个字符对应若干个字节。如果要在网络上传输，或者保存到磁盘上，就需要把str变为以字节为单位的bytes。

Python对bytes类型的数据用带b前缀的单引号或双引号表示：

```
x = b'ABC'
```

要注意区分'ABC'和b'ABC'，前者是str，后者虽然内容显示得和前者一样，但bytes的每个字符都只占用一个字节。

以Unicode表示的str通过encode()方法可以编码为指定的bytes，例如：

```
>>> 'ABC'.encode('ascii')
b'ABC'
>>> '中文'.encode('utf-8')
b'\xe4\x b8\xad\xe6\x96\x87'
>>> '中文'.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1: ordinal not in range(128)
```

纯英文的str可以用ASCII编码为bytes，内容是一样的，含有中文的str可以用UTF-8编码为bytes。含有中文的str无法用ASCII编码，因为中文编码的范围超过了ASCII编码的范围，Python会报错。

在bytes中，无法显示为ASCII字符的字节，用\x##显示。

反过来，如果我们从网络或磁盘上读取了字节流，那么读到的数据就是bytes。要把bytes变为str，就需要用decode()方法：

```
>>> b'ABC'.decode('ascii')
'ABC'
>>> b'\xe4\x b8\xad\xe6\x96\x87'.decode('utf-8')
'中文'
如果bytes中包含无法解码的字节，decode()方法会报错：

>>> b'\xe4\x b8\xad\xff'.decode('utf-8')
Traceback (most recent call last):
  ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 3: invalid start byte
```

如果bytes中只有一小部分无效的字节，可以传入errors='ignore'忽略错误的字节：

```
>>> b'\xe4\x b8\xad\xff'.decode('utf-8', errors='ignore')
'中'
```

要计算str包含多少个字符，可以用len()函数：

```
>>> len('ABC')
3
>>> len('中文')
2
```

len()函数计算的是str的字符数，如果换成bytes，len()函数就计算字节数：

```
>>> len(b'ABC')
3
>>> len(b'\xe4\x b8\xad\xe6\x96\x87')
6
>>> len('中文'.encode('utf-8'))
6
```

可见，1个中文字符经过UTF-8编码后通常会占用3个字节，而1个英文字符只占用1个字节。

为了避免乱码问题，应当始终坚持使用UTF-8编码对str和bytes进行转换。

由于Python源代码也是一个文本文件，所以，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，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，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.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。

### 1.2.3. 格式化

最后一个常见的问题是如何输出格式化的字符串。经常会输出类似'亲爱的xxx你好！你xx月的话费是xx，余额是xx'之类的字符串，而xxx的内容都是根据变量变化的，所以，需要一种简便的格式化字符串的方式。

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：

```
>>> 'Hello, %s' % 'world'
'Hello, world'
>>> 'Hi, %s, you have $%d.' % ('Michael', 1000000)
'Hi, Michael, you have $1000000.'
```

%运算符就是用来格式化字符串的。在字符串内部，%s表示用字符串替换，%d表示用整数替换，有几个%?占位符，后面就跟几个变量或者值，顺序要对应好。如果只有一个%?, 括号可以省略。

常见的占位符有：

占位符	替换内容
%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：

如果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'
```

有些时候，字符串里面的%是一个普通字符怎么办？这个时候就需要转义，用%%来表示一个%：

```
>>> 'growth rate: %d %%' % 7
'growth rate: 7 %'
```

1.2.4. format()

另一种格式化字符串的方法是使用字符串的format()方法，它会用传入的参数依次替换字符串内的占位符{0}、{1}.....，不过这种方式写起来比%要麻烦得多：

```
>>> 'Hello, {0}, 成绩提升了 {1:.1f}%'.format('小明', 17.125)
'Hello, 小明, 成绩提升了 17.1%'
```

1.3. 使用list和tuple

1.3.1. list

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：

```
>>> len(classmates)
3
```

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：

```
>>> classmates[0]
'Michael'
>>> classmates[1]
'Bob'
>>> classmates[2]
'Tracy'
>>> classmates[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

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'
>>> classmates[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

当然，倒数第4个就越界了。

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')
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']
```

要删除list末尾的元素，用pop()方法：

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

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)
'Jack'
>>> classmates
['Michael', 'Bob', 'Tracy']
```

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'
>>> classmates
['Michael', 'Sarah', 'Tracy']
```

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']
>>> len(s)
4
```

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 = []
>>> len(L)
0
```

### 1.3.2. tuple

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，可以正常地使用classmates[0]，classmates[-1]，但不能赋值成另外的元素。

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。

tuple的陷阱：当定义一个tuple时，在定义的时候，tuple的元素就必须被确定下来，比如：

```
>>> t = (1, 2)
>>> t
(1, 2)
```

如果要定义一个空的tuple，可以写成():

```
>>> t = ()
>>> t
()
```

但是，要定义一个只有1个元素的tuple，如果这么定义：

```
>>> t = (1)
>>> t
1
```

定义的不是tuple，是1这个数！这是因为括号()既可以表示tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python规定，这种情况下，按小括号进行计算，计算结果自然是1。

所以，只有1个元素的tuple定义时必须加一个逗号，来消除歧义：

```
>>> t = (1,)
>>> t
(1,)
```

Python在显示只有1个元素的tuple时，也会加一个逗号，以免你误解成数学计算意义上的括号。

最后来看一个“可变的”tuple：

```
>>> t = ('a', 'b', ['A', 'B'])
>>> t[2][0] = 'X'
>>> t[2][1] = 'Y'
>>> t
('a', 'b', ['X', 'Y'])
```

这个tuple定义的时候有3个元素，分别是'a', 'b'和一个list。

表面上看，tuple的元素确实变了，但其实变的不是tuple的元素，而是list的元素。tuple一开始指向的list并没有改成别的list，所以，**tuple所谓的“不变”是说，tuple的每个元素，指向永远不变。**即指向'a'，就不能改成指向'b'，指向一个list，就不能改成指向其他对象，但指向的这个list本身是可变的！

要创建一个内容也不变的tuple那就必须保证tuple的每一个元素本身也不能变。

## 1.4. 条件判断

计算机之所以能做很多自动化的任务，因为它可以自己做条件判断。

比如，输入用户年龄，根据年龄打印不同的内容，在Python程序中，用if语句实现：

```
age = 20
if age >= 18:
    print('your age is', age)
    print('adult')
```

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:。

当然上面的判断是很粗略的，完全可以用elif做更细致的判断：

```
age = 3
if age >= 18:
    print('adult')
elif age >= 6:
    print('teenager')
else:
    print('kid')
```

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>
```

if语句执行有个特点，它是从上往下判断，如果在某个判断上是True，把该判断对应的语句执行后，就忽略掉剩下的elif和else。

if判断条件还可以简写，比如写：

```
if x:
    print('True')
```

只要x是非零数值、非空字符串、非空list等，就判断为True，否则为False。

### 1.4.1. input

```
birth = input('birth: ')
if birth < 2000:
    print('00前')
else:
    print('00后')
```

输入1982，结果报错：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() > int()
```

这是因为input()返回的数据类型是str，str不能直接和整数比较，必须先把str转换成整数。Python提供了int()函数来完成这件事情：

```
s = input('birth: ')
birth = int(s)
if birth < 2000:
    print('00前')
else:
    print('00后')
```

再次运行，就可以得到正确地结果。

## 1.5. 循环

### 1.5.1. for...in循环

```
sum = 0
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum = sum + x
print(sum)
```

### 1.5.2. while循环

```
sum = 0
n = 99
while n > 0:
    sum = sum + n
    n = n - 2
print(sum)
```

## 1.6. 使用dict和set

### 1.6.1. dict

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。

```
names = ['Michael', 'Bob', 'Tracy']
scores = [95, 75, 85]
```

先要在names中找到对应的位置，再从scores取出对应的成绩，list越长，耗时越长。

用Python写一个dict如下：

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

直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。

这种key-value存储方式，在放进去的时候，必须根据key算出value的存放位置，这样，取的时候才能根据key直接拿到value。

把数据放入dict的方法，除了初始化时指定外，还可以通过key放入：

```
>>> d['Adam'] = 67
>>> d['Adam']
67
```

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：

```
>>> d['Jack'] = 90
>>> d['Jack']
90
>>> d['Jack'] = 88
>>> d['Jack']
88
```

如果key不存在，dict就会报错：

```
>>> d['Thomas']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Thomas'
```

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的交互环境不显示结果。

删除key：

```
>>> d.pop('Bob')
75
>>> d
{'Michael': 95, 'Tracy': 85}
```

和list比较，dict有以下几个特点：

1. 查找和插入的速度极快，不会随着key的增加而变慢；
2. 需要占用大量的内存，内存浪费多。

**dict的key必须是不可变对象。**（使用哈希算法计算value的位置）

### 1.6.2. set

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：

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

注意，传入的参数[1, 2, 3]是一个list，而显示的{1, 2, 3}只是告诉你这个set内部有1，2，3这3个元素，显示的顺序也不表示set是有序的。。

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}
```

set和dict的唯一区别仅在于没有存储对应的value，但是，set的原理和dict一样，所以，同样不可以放入可变对象，因为无法判断两个可变对象是否相等，也就无法保证set内部“不会有重复元素”。

## 2. 函数

### 2.1. 调用函数

在交互式命令行通过help(XXX)查看XXX函数的帮助信息。

调用函数的时候，如果传入的参数数量不对，会报TypeError的错误：

```
>>> abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (2 given)
```

如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报TypeError的错误，并且给出错误信息：str是错误的参数类型：

```
>>> abs('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

#### 2.1.1. 数据类型转换

Python内置的常用函数还包括数据类型转换函数：

```
>>> int('123')
123
>>> int(12.34)
12
>>> float('12.34')
12.34
>>> str(1.23)
'1.23'
>>> str(100)
'100'
>>> bool(1)
True
>>> bool('')
False
```

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
```

### 2.2. 定义函数

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。

如果没有return语句，函数执行完毕后会返回结果，只是结果为None。return None可以简写为return。

#### 2.2.1. 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。

### 2.2.2. 参数检查

调用函数时，如果参数个数不对，Python解释器会自动检查出来，并抛出TypeError:

参数类型不对：数据类型检查可以用内置函数isinstance()实现。

```
def my_abs(x):
    if not isinstance(x, (int, float)):
        raise TypeError('bad operand type')
    if x >= 0:
        return x
    else:
        return -x
```

### 2.2.3. 返回多个值

这只是一种假象，Python函数返回的仍然是单一值，返回值是一个tuple。

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。

## 2.3. 函数的参数

### 2.3.1. 默认参数

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

必选参数在前，默认参数在后。

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。

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')。

默认参数使用不变对象。

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。

### 2.3.2. 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:

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

已经有一个list或者tuple，要调用一个可变参数:

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

### 2.3.3. 关键字参数

关键字参数允许传入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。

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

```
>>> 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'}
```

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。

```
>>> 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。

### 2.3.4. 命名关键字参数

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)
```

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。

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

如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符\*了：

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

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
```

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)
```

如果没有可变参数，就必须加一个\*作为特殊分隔符,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。

### 2.3.5. 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。

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。

对于任意函数，都可以通过类似func(\*args, \*\*kw)的形式调用它，无论它的参数是如何定义的。

## 2.4. 递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。

尾递归是指，在函数返回的时候，调用自身本身，并且，return语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化。

## 3. 高级特性

### 3.1. 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']
```

取前3个元素

笨办法：

```
>>> [L[0], L[1], L[2]]
['Michael', 'Sarah', 'Tracy']
```

之所以是笨办法是因为扩展一下，取前N个元素就没辙了。

取前N个元素，也就是索引为0-(N-1)的元素，可以用循环：

```
>>> r = []
>>> n = 3
```

```
>>> for i in range(n):
...     r.append(L[i])
...
>>> r
['Michael', 'Sarah', 'Tracy']
```

对这种经常取指定索引范围的操作，用循环十分繁琐，因此，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']
```

L[0:3]表示，从索引0开始取，直到索引3为止，但不包括索引3。即索引0，1，2，正好是3个元素。

如果第一个索引是0，还可以省略：

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

也可以从索引1开始，取出2个元素出来：

```
>>> L[1:3]
['Sarah', 'Tracy']
```

类似的，既然Python支持L[-1]取倒数第一个元素，那么它同样支持倒数切片，试试：

```
>>> L[-2:]
['Bob', 'Jack']
>>> L[-2:-1]
['Bob']
```

**倒数第一个元素的索引是-1。**

切片操作十分有用。创建一个0-99的数列：

```
>>> L = list(range(100))
>>> L
[0, 1, 2, 3, ..., 99]
```

可以通过切片轻松取出某一段数列。比如前10个数：

```
>>> L[:10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

后10个数：

```
>>> L[-10:]
[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

前11-20个数：

```
>>> L[10:20]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

前10个数，每两个取一个：

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

所有数，每5个取一个：

```
>>> L[::5]
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
```

甚至什么都不写，只写[]就可以原样复制一个list：

```
>>> L[:]
[0, 1, 2, 3, ..., 99]
```

**tuple也是一种list，唯一区别是tuple不可变。**因此，tuple也可以用切片操作，只是操作的结果仍是tuple：

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

**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：

```
>>> 'ABCDEFGF'[:3]
'ABC'
>>> 'ABCDEFGF'[:2]
```

```
'ACEG'
```

在很多编程语言中，针对字符串提供了很多各种截取函数（例如，**substring**），其实目的就是对字符串切片。Python没有针对字符串的截取函数，只需要切片一个操作就可以完成，非常简单。

## 3.2. 迭代

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

在Python中，迭代是通过for ... in来完成的，而很多语言比如C语言，迭代list是通过下标完成的，比如Java代码：

```
for (i=0; i<list.length; i++) {  
    n = list[i];  
}
```

可以看出，Python的for循环抽象程度要高于C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以下标，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)  
...  
a  
c  
b
```

因为dict的存储不是按照list的方式顺序排列，所以，迭代出的结果顺序很可能不一样。

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()。

由于字符串也是可迭代对象，因此，也可以作用于for循环：

```
>>> for ch in 'ABC':  
...     print(ch)  
...  
A  
B  
C
```

所以，当使用for循环时，只要作用于一个可迭代对象，for循环就可以正常运行。

如何判断一个对象是可迭代对象呢？方法是通过collections模块的Iterable类型判断：

```
>>> from collections import Iterable  
>>> isinstance('abc', Iterable) # str是否可迭代  
True  
>>> isinstance([1,2,3], Iterable) # list是否可迭代  
True  
>>> isinstance(123, Iterable) # 整数是否可迭代  
False
```

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)  
...  
1 1  
2 4  
3 9
```

## 3.3. 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。

列表生成式则可以用一行语句代替循环生成[1x1, 2x2, 3x3, ..., 10x10]

```
>>> [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，可以筛选出仅偶数的平方：

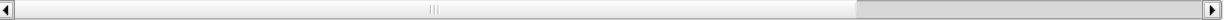
```
>>> [x * x for x in range(1, 11) if x % 2 == 0]
[4, 16, 36, 64, 100]
```

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']
```

列出当前目录下的所有文件和目录名，可以通过一行代码实现：

```
>>> import os # 导入os模块，模块的概念后面讲到
>>> [d for d in os.listdir('.')] # os.listdir可以列出文件和目录
['.emacs.d', '.ssh', '.Trash', 'Adlm', 'Applications', 'Desktop', 'Documents', 'Downloads', 'Library', 'Movies', 'Music', 'Pictures', 'P...
```



for循环其实可以同时使用两个甚至多个变量，比如dict的items()可以同时迭代key和value：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }
>>> for k, v in d.items():
...     print(k, '=', v)
...
y = B
x = A
z = C
```

因此，列表生成式也可以使用两个变量来生成list：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }
>>> [k + '=' + v for k, v in d.items()]
['y=B', 'x=A', 'z=C']
```

把一个list中所有的字符串变成小写：

```
>>> L = ['Hello', 'World', 'IBM', 'Apple']
>>> [s.lower() for s in L]
['hello', 'world', 'ibm', 'apple']
```

## 3.4. 生成器

列表的缺陷：

1. 受到内存限制，列表容量肯定是有限的。
2. 创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

Python中，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。

如果要一个一个打印出来generator，可以通过next()函数获得generator的下一个返回值：

```
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
>>> next(g)
9
>>> next(g)
16
>>> next(g)
25
>>> next(g)
36
>>> next(g)
49
>>> next(g)
64
>>> next(g)
```



```
81
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

generator保存的是算法，每次调用next(g)，就计算出g的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。

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)
...
0
1
4
9
16
25
36
49
64
81
```

创建了一个generator后，基本上永远不会调用next()，而是通过for循环来迭代它，并且不需要关心StopIteration的错误。

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。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易。

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

```
>>> fib(6)
1
1
2
3
5
8
'done'
```

fib函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

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'
```

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：

```
>>> f = fib(6)
>>> f
<generator object fib at 0x104feaaa0>
```

**generator和函数执行流程的区别？**

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。

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
```

odd不是普通函数，而是generator，在执行过程中，遇到yield就中断，下次又继续执行。执行3次yield后，已经没有yield可以执行了，所以，第4次调用next(o)就报错。

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)
...
1
1
2
3
5
8
```

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
...
g: 1
g: 1
g: 2
g: 3
g: 5
g: 8
Generator return value: done
```

## 3.5. 迭代器

可以直接作用于for循环的数据类型有以下几种：

一类是集合数据类型，如list、tuple、dict、set、str等；

一类是generator，包括生成器和带yield的generator function。

可以直接作用于for循环的对象统称为可迭代对象：**Iterable**。

可以使用isinstance()判断一个对象是否是Iterable对象：

```
>>> from collections import Iterable
>>> isinstance([], Iterable)
True
>>> isinstance({}, Iterable)
True
>>> isinstance('abc', Iterable)
True
>>> isinstance((x for x in range(10)), Iterable)
True
>>> isinstance(100, Iterable)
False
```

而生成器不但可以作用于for循环，还可以被next()函数不断调用并返回下一个值，直到最后抛出StopIteration错误表示无法继续返回下一个值了。

可以被next()函数调用并不断返回下一个值的对象称为迭代器：**Iterator**。

可以使用isinstance()判断一个对象是否是Iterator对象：

```
>>> from collections import Iterator
>>> isinstance((x for x in range(10)), Iterator)
True
>>> isinstance([], Iterator)
False
>>> isinstance({}, Iterator)
False
>>> isinstance('abc', Iterator)
False
```

生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator。

把list、dict、str等Iterable变成Iterator可以使用iter()函数：

```
>>> isinstance(iter([]), Iterator)
True
>>> isinstance(iter('abc'), Iterator)
True
```

为什么list、dict、str等数据类型不是Iterator？

这是因为Python的Iterator对象表示的是一个数据流，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列，却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以Iterator的计算是惰性的，只有在需要返回下一个数据时它才会计算。

Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

## 4. 函数式编程

### 4.1. 高阶函数

#### 4.1.1. 变量可以指向函数

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

#### 4.1.2. 函数名也是变量

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

#### 4.1.3. 传入函数

高阶函数，就是让函数的参数能够接收别的函数。

函数式编程就是指这种高度抽象的编程范式。

#### 4.1.4. map/reduce

##### 4.1.4.1. 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。

```
>>> 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]
```

##### 4.1.5. reduce()

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：

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

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])
```

### 4.1.6. filter

Python内建的filter()函数用于过滤序列。

filter()也接收一个函数和一个序列。和map()不同的是，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。

在一个list中，删掉偶数，只保留奇数：

```
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]
```

### 4.1.7. sorted

#### 4.1.7.1. 排序算法

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]
```

key指定的函数将作用于list的每一个元素上，并根据key函数返回的结果进行排序。对比原始的list和经过key=abs处理过的list：

```
list = [36, 5, -12, 9, -21]
keys = [36, 5, 12, 9, 21]
```

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()也是一个高阶函数。用sorted()排序的关键在于实现一个映射函数。

## 4.2. 返回函数

### 4.2.1. 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。

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

当调用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>
```

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

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。

当调用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。

### 4.2.2. 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。

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。

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

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

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

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，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
```

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

缺点是代码较长，可利用lambda函数缩短代码。

## 4.3. 匿名函数

在Python中，对匿名函数提供了有限支持。

以map()函数为例，计算f(x)=x<sup>2</sup>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]
```

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。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：

```
>>> f = lambda x: x * x
>>> f
<function <lambda> at 0x101c6ef28>
>>> f(5)
25
```

把匿名函数作为返回值返回，比如：

```
def build(x, y):
    return lambda: x * x + y * y
```

## 4.4. 装饰器

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。

```
>>> def now():
...     print('2015-3-25')
...
>>> f = now
>>> f()
2015-3-25
```

函数对象有一个\_\_name\_\_属性，可以拿到函数的名字：

```
>>> now.__name__
'now'
>>> f.__name__
'now'
```

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
```

```
@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)
```

我们来剖析上面的语句，首先执行log('execute')，返回的是decorator函数，再调用返回的函数，参数是now函数，返回值最终是wrapper函数。

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，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
```

或者针对带参数的decorator：

```
import functools

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

`import functools` 是导入 `functools` 模块。模块的概念稍候讲解。现在，只需记住在定义 `wrapper()` 的前面加上 `@functools.wraps(func)` 即可。

## 4.5. 偏函数

Python的 `functools` 模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。

通过设定参数的默认值，可以降低函数调用的难度。

`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
>>> int('12345', 16)
74565
```

假设有要转换大量的二进制字符串，每次都传入 `int(x, base=2)` 非常麻烦，可以定义一个 `int2()` 的函数，默认把 `base=2` 传进去：

```
def int2(x, base=2):
    return int(x, base)
```

```
>>> int2('1000000')
64
>>> int2('1010101')
85
```

`functools.partial` 就是创建偏函数的，不需要自己定义 `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
```

最后，创建偏函数时，实际上可以接收函数对象、`*args` 和 `**kw` 这3个参数，当传入：

```
int2 = functools.partial(int, base=2)
```

实际上固定了 `int()` 函数的关键字参数 `base`，也就是：

```
int2('10010')
```

相当于：

```
kw = { 'base': 2 }
int('10010', **kw)
```

当传入：

```
max2 = functools.partial(max, 10)
```

实际上会把10作为\*args的一部分自动加到左边，也就是：

```
max2(5, 6, 7)
```

相当于：

```
args = (10, 5, 6, 7)
max(*args)
```

结果为10。

当函数的参数个数太多，需要简化时，使用`functools.partial`可以创建一个新的函数，这个新函数可以固定住原函数的部分参数，从而在调用时更简单。

## 5. 模块

### 5.1. 使用模块

使用`__author__`变量把作者写进去。

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

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；

外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

### 5.2. 安装第三方模块

在Python中，安装第三方模块，是通过包管理工具pip完成的。

#### 5.2.1. 模块搜索路径

当试图加载一个模块时，Python会在指定的路径下搜索对应的.py文件，如果找不到，就会报错：

```
>>> import mymodule
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named mymodule
```

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/3.6/lib/python36.zip', '/Library/Frameworks/Python.framework/Versions/3.6/lib/python36/lib-dynload', '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages']
```

如果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。

## 6. 面向对象编程

面向对象的设计思想是抽象出Class，根据Class创建Instance。



## 6.1. 类和实例

`__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。

### 6.1.1. 数据封装

要定义一个方法，除了第一个参数是`self`外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了`self`不用传递，其他参数正常传入。

和静态语言不同，Python允许对实例变量绑定任何数据，也就是说，对于两个实例变量，虽然它们都是同一个类的不同实例，但拥有的变量名称都可能不同。

## 6.2. 访问限制

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。

在Python中，变量名类似`__xxx__`的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用`__name__`、`_score__`这样的变量名。

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，这样的变量意思是，“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”。

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。

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。

Python本身没有任何机制阻止你干坏事，一切全靠自觉。

## 6.3. 继承和多态

在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。

传入的任意类型，只要是XXX类或者子类，就会自动调用实际类型的XXX()方法，这就是多态的意思。

### 6.3.1. 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XXX类型，则传入的对象必须是XXX类型或者它的子类，否则，将无法调用xxx()方法。

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

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。

## 6.4. 获取对象信息

### 6.4.1. 使用type()

type()函数返回的是对应的Class类型。

判断一个对象是否是函数：

```
>>> import types
>>> def fn():
...     pass
...
>>> type(fn)==types.FunctionType
True
>>> type(abs)==types.BuiltinFunctionType
True
>>> type(lambda x: x)==types.LambdaType
True
>>> type((x for x in range(10)))==types.GeneratorType
True
```

### 6.4.2. 使用isinstance()

判断class的类型，可以使用isinstance()函数。

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。

### 6.4.3. 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__', ..., '__subclasshook__', 'capitalize', 'casefold', ..., 'zfill']
```

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
```

## 6.5. 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：

```
class Student(object):
    def __init__(self, name):
        self.name = name

s = Student('Bob')
s.score = 90
```

`Student` 类本身需要绑定一个属性，可以直接在 `class` 中定义属性，这种属性是类属性。

在编写程序的时候，千万不要对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。

## 7. 面向对象高级编程

### 7.1. 使用 `__slots__`

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
```

```
>>> 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。

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__`。

### 7.2. 使用 `@property`

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
```

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。

@property广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。

## 7.3. 多重继承

### 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。

```
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。

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。

## 7.4. 定制类

### \_\_str\_\_

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def __str__(self):
...         return 'Student object (name: %s)' % self.name
...
>>> print(Student('Michael'))
Student object (name: Michael)
```

但是

```
>>> s = Student('Michael')
>>> s
<__main__.Student object at 0x109afb310>
```

这是因为直接显示变量调用的不是\_\_str\_\_(), 而是\_\_repr\_\_(), 两者的区别是\_\_str\_\_()返回用户看到的字符串，而\_\_repr\_\_()返回程序开发者看到的字符串，也就是说\_\_repr\_\_()是为调试服务的。

```
class Student(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Student object (name=%s)' % self.name
    __repr__ = __str__
```

### \_\_iter\_\_

如果一个类想被用于for ... in循环，类似list或tuple那样，就必须实现一个\_\_iter\_\_()方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的\_\_next\_\_()方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环。

斐波那契数列

```
class Fib(object):
    def __init__(self):
        self.a, self.b = 0, 1 # 初始化两个计数器a, b

    def __iter__(self):
        return self # 实例本身就是迭代对象，故返回自己

    def __next__(self):
        self.a, self.b = self.b, self.a + self.b # 计算下一个值
        if self.a > 100000: # 退出循环的条件
            raise StopIteration()
        return self.a # 返回下一个值
```

```
>>> for n in Fib():
...     print(n)
...
1
1
2
3
5
```

```
...
46368
75025
```

\_\_getitem\_\_

```
class Fib(object):
    def __getitem__(self, n):
        a, b = 1, 1
        for x in range(n):
            a, b = b, a + b
        return a
```

现在，就可以按下标访问数列的任意一项了：

```
>>> f = Fib()
>>> f[0]
1
>>> f[1]
1
>>> f[2]
2
>>> f[3]
3
>>> f[10]
89
>>> f[100]
573147844013817084101
```

切片对于Fib却报错。原因是\_\_getitem\_\_()传入的参数可能是一个int，也可能是一个切片对象slice，所以要做判断：

```
class Fib(object):
    def __getitem__(self, n):
        if isinstance(n, int): # n是索引
            a, b = 1, 1
            for x in range(n):
                a, b = b, a + b
            return a
        if isinstance(n, slice): # n是切片
            start = n.start
            stop = n.stop
            if start is None:
                start = 0
            a, b = 1, 1
            L = []
            for x in range(stop):
                if x >= start:
                    L.append(a)
                a, b = b, a + b
            return L
```

```
>>> f = Fib()
>>> f[0:5]
[1, 1, 2, 3, 5]
>>> f[:10]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

但是没有对step参数作处理，也没有对负数作处理，所以，要正确实现一个\_\_getitem\_\_()还是有很多工作要做的。

此外，如果把对象看成dict，\_\_getitem\_\_()的参数也可能是一个可以作key的object，例如str。

与之对应的是\_\_setitem\_\_()方法，把对象视作list或dict来对集合赋值。最后，还有一个\_\_delitem\_\_()方法，用于删除某个元素。

总之，通过上面的方法，我们自己定义的类表现得和Python自带的list、tuple、dict没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

\_\_getattr\_\_

要让class只响应特定的几个属性，我们就要按照约定，抛出AttributeError的错误。

```
class Student(object):

    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
        raise AttributeError('\''Student\' object has no attribute \'' + attr + '\')
```

\_\_call\_\_

任何类，只需要定义一个\_\_call\_\_()方法，就可以直接对实例进行调用。

```
class Student(object):
    def __init__(self, name):
```

```
self.name = name

def __call__(self):
    print('My name is %s.' % self.name)
```

```
>>> s = Student('Michael')
>>> s() # self参数不要传入
My name is Michael.
```

## 7.5. 使用枚举类

```
from enum import Enum

Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))
```

这样获得了Month类型的枚举类，可以直接使用Month.Jan来引用一个常量，或者枚举它的所有成员：

```
>>> for name, member in Month.__members__.items():
...     print(name, '=>', member, ',', member.value)
...
Jan => Month.Jan , 1
Feb => Month.Feb , 2
Mar => Month.Mar , 3
Apr => Month.Apr , 4
May => Month.May , 5
Jun => Month.Jun , 6
Jul => Month.Jul , 7
Aug => Month.Aug , 8
Sep => Month.Sep , 9
Oct => Month.Oct , 10
Nov => Month.Nov , 11
Dec => Month.Dec , 12
```

value属性则是自动赋给成员的int常量，默认从1开始计数。

如果需要更精确地控制枚举类型，可以从Enum派生出自定义类：

```
from enum import Enum, unique

@unique
class Weekday(Enum):
    Sun = 0 # Sun的value被设定为0
    Mon = 1
    Tue = 2
    Wed = 3
    Thu = 4
    Fri = 5
    Sat = 6
```

@unique装饰器可以帮助我们检查保证没有重复值。

访问这些枚举类型可以有若干种方法：

```
>>> day1 = Weekday.Mon
>>> print(day1)
Weekday.Mon
>>> print(Weekday.Tue)
Weekday.Tue
>>> print(Weekday['Tue'])
Weekday.Tue
>>> print(Weekday.Tue.value)
2
>>> print(day1 == Weekday.Mon)
True
>>> print(day1 == Weekday.Tue)
False
>>> print(Weekday(1))
Weekday.Mon
>>> print(day1 == Weekday(1))
True
>>> Weekday(7)
Traceback (most recent call last):
...
ValueError: 7 is not a valid Weekday
>>> for name, member in Weekday.__members__.items():
...     print(name, '=>', member)
...
Sun => Weekday.Sun
Mon => Weekday.Mon
Tue => Weekday.Tue
Wed => Weekday.Wed
Thu => Weekday.Thu
```

```
Fri => Weekday.Fri
Sat => Weekday.Sat
```

## 7.6. 使用元类

### 7.6.1. type()

动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

```
class Hello(object):
    def hello(self, name='world'):
        print('Hello, %s.' % name)
```

```
>>> from hello import Hello
>>> h = Hello()
>>> h.hello()
Hello, world.
>>> print(type(Hello))
<class 'type'>
>>> print(type(h))
<class 'hello.Hello'>
```

type()函数可以查看一个类型或变量的类型，Hello是一个class，它的类型就是type，而h是一个实例，它的类型就是class Hello。

class的定义是运行时动态创建的，而创建class的方法就是使用type()函数。

type()函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过type()函数创建出Hello类，而无需通过class Hello(object)...的定义：

```
>>> def fn(self, name='world'): # 先定义函数
...     print('Hello, %s.' % name)
...
>>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello class
>>> h = Hello()
>>> h.hello()
Hello, world.
>>> print(type(Hello))
<class 'type'>
>>> print(type(h))
<class '__main__.Hello'>
```

要创建一个class对象，type()函数依次传入3个参数：

1. class的名称；
2. 继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元元素写法；
3. class的方法名称与函数绑定，这里我们把函数fn绑定到方法名hello上。

正常情况下，我们都用class Xxx...来定义类，但是，type()函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

### 7.6.2. metaclass

先定义metaclass，就可以创建类，最后创建实例。

所以，metaclass允许你创建类或者修改类。换句话说，你可以把类看成是metaclass创建出来的“实例”。

例子：这个metaclass可以给我们自定义的MyList增加一个add方法：

定义ListMetaclass，按照默认习惯，metaclass的类名总是以Metaclass结尾，以便清楚地表示这是一个metaclass：

```
# metaclass是类的模板，所以必须从`type`类型派生：
class ListMetaclass(type):
    def __new__(cls, name, bases, attrs):
        attrs['add'] = lambda self, value: self.append(value)
        return type.__new__(cls, name, bases, attrs)
```

有了ListMetaclass，我们在定义类的时候还要指示使用ListMetaclass来定制类，传入关键字参数metaclass：

```
class MyList(list, metaclass=ListMetaclass):
    pass
```

当传入关键字参数metaclass时，魔术就生效了，它指示Python解释器在创建MyList时，要通过ListMetaclass.new()来创建，在此，我们可以修改类的定义，比如，加上新的方法，然后，返回修改后的定义。

new()方法接收到的参数依次是：

1. 当前准备创建的类的对象；
2. 类的名字；
3. 类继承的父类集合；

4. 类的方法集合。

```
>>> L = MyList()
>>> L.add(1)
>> L
[1]
```

## 8. 错误、调试和测试

### 8.1. 错误处理

在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因。在操作系统提供的调用中，返回错误码非常常见。比如打开文件的函数`open()`，成功时返回文件描述符（就是一个整数），出错时返回-1。

用错误码来表示是否出错十分不便，因为函数本身应该返回的正常结果和错误码混在一起，造成调用者必须用大量的代码来判断是否出错：

```
def foo():
    r = some_function()
    if r==(-1):
        return (-1)
    # do something
    return r

def bar():
    r = foo()
    if r==(-1):
        print('Error')
    else:
        pass
```

一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。

所以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。

#### 8.1.1. try

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

```
try...
except: division by zero
finally...
END
```

可以没有`finally`语句。

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。

```

BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |   +-- FloatingPointError
    |   +-- OverflowError
    |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    |   +-- ModuleNotFoundError
    +-- 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

```

可以跨越多层调用，比如函数main()调用foo()，foo()调用bar()，结果bar()出错了，这时，只要main()捕获到了，就可以处理：

```

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

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

def main():
    try:
        bar('0')
    except Exception as e:
        print('Error:', e)
    finally:
        print('finally...')

```



如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。

```
# err.py:
def foo(s):
    return 10 / int(s)

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

def main():
    bar('0')

main()
```

从上往下可以看到整个错误的调用函数链：

```
$ python3 err.py
Traceback (most recent call last):
  File "err.py", line 11, in <module>
    main()
  File "err.py", line 9, in main
    bar('0')
  File "err.py", line 6, in bar
    return foo(s) * 2
  File "err.py", line 3, in foo
    return 10 / int(s)
ZeroDivisionError: division by zero
```

### 8.1.3. 记录错误

既然能捕获错误，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。

```
# 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')
```

```
$ 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。

### 8.1.4. 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：

```
# 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')
```

```
$ 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
```

另一种错误处理方式：

```
# err_reraise.py

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

def bar():
    try:
        foo('0')
    except ValueError as e:
        print('ValueError!')
        raise

bar()
```

捕获错误目的只是记录一下，便于后续追踪。但是，由于当前函数不知道应该怎么处理该错误，所以，最恰当的方式是继续往上抛，让顶层调用者去处理。

raise语句如果不带参数，就会把当前错误原样抛出。此外，在except中raise一个Error，还可以把一种类型的错误转化成另一种类型：

```
try:
    10 / 0
except ZeroDivisionError:
    raise ValueError('input error!')
```

只要是合理的转换逻辑就可以，但是，决不应该把一个IOError转换成毫不相干的ValueError。

## 8.2. 调试

### 8.2.1. 断言

凡是用print()来辅助查看的地方，都可以用断言（assert）来替代：

```
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n

def main():
    foo('0')
```

assert的意思是，表达式n != 0应该是True，否则，根据程序运行的逻辑，后面的代码肯定会出错。

如果断言失败，assert语句本身就会抛出AssertionError：

```
$ python err.py
Traceback (most recent call last):
...
AssertionError: n is zero!
```

程序中如果到处充斥着assert，和print()相比也好不到哪去。不过，启动Python解释器时可以用-O参数来关闭assert：

```
$ python -O err.py
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

关闭后，可以把所有的assert语句当成pass来看。

### 8.2.2. logging

和assert比，logging不会抛出错误，而且可以输出到文件：

```
import logging
logging.basicConfig(level=logging.INFO)

s = '0'
n = int(s)
logging.info('n = %d' % n)
print(10 / n)
```

```
$ python err.py
INFO:root:n = 0
Traceback (most recent call last):
  File "err.py", line 8, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

允许指定记录信息的级别，有debug，info，warning，error等几个级别，当指定level=INFO时，logging.debug就不起作用了。同理，指定level=WARNING后，debug和info就不起作用了。

logging的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如console和文件。

### 8.2.3. pdb

启动Python的调试器pdb，让程序以单步方式运行，可以随时查看运行状态。

```
# err.py
s = '0'
n = int(s)
print(10 / n)
```

```
$ python -m pdb err.py
> /err.py(2)<module>()
-> s = '0'
```

以参数-m pdb启动后，pdb定位到下一步要执行的代码-> s = '0'。输入命令l来查看代码：

```
(Pdb) l
1      # err.py
2  ->  s = '0'
3      n = int(s)
4      print(10 / n)
```

输入命令n可以单步执行代码：

```
(Pdb) n
> /err.py(3)<module>()
-> n = int(s)
(Pdb) n
> /err.py(4)<module>()
-> print(10 / n)
```

任何时候都可以输入命令p 变量名来查看变量：

```
(Pdb) p s
'0'
(Pdb) p n
0
```

输入命令q结束调试，退出程序：

```
(Pdb) q
```

### 8.2.4. pdb.set\_trace()

需要import pdb，然后，在可能出错的地方放一个pdb.set\_trace()，就可以设置一个断点：

```
# err.py
import pdb

s = '0'
n = int(s)
pdb.set_trace() # 运行到这里会自动暂停
print(10 / n)
```

运行代码，程序会自动在pdb.set\_trace()暂停并进入pdb调试环境，可以用命令p查看变量，或者用命令c继续运行：

```
$ python err.py
> /err.py(7)<module>()
-> print(10 / n)
(Pdb) p n
0
(Pdb) c
Traceback (most recent call last):
  File "err.py", line 7, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

### 8.2.5. IDE

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的IDE。

## 8.3. 单元测试

### “测试驱动开发”（TDD: Test-Driven Development）

如果单元测试通过，说明我们测试的这个函数能够正常工作。如果单元测试不通过，要么函数有bug，要么测试条件输入不正确，需要修复使单元测试能够通过。

修改后再跑一遍单元测试，如果通过，说明修改不会对函数原有的行为造成影响，如果测试不通过，说明我们的修改与原有行为不一致，要么修改代码，要么修改测试。

```
class Dict(dict):

    def __init__(self, **kw):
        super().__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value
```

```
import unittest

from mydict import Dict

class TestDict(unittest.TestCase):

    def test_init(self):
        d = Dict(a=1, b='test')
        self.assertEqual(d.a, 1)
        self.assertEqual(d.b, 'test')
        self.assertTrue(isinstance(d, dict))

    def test_key(self):
        d = Dict()
        d['key'] = 'value'
        self.assertEqual(d.key, 'value')

    def test_attr(self):
        d = Dict()
        d.key = 'value'
        self.assertTrue('key' in d)
        self.assertEqual(d['key'], 'value')

    def test_keyerror(self):
        d = Dict()
        with self.assertRaises(KeyError):
            value = d['empty']

    def test_attrerror(self):
        d = Dict()
        with self.assertRaises(AttributeError):
            value = d.empty
```

编写单元测试时，我们需要编写一个测试类，从`unittest.TestCase`继承。

以`test`开头的方法就是测试方法，不以`test`开头的方法不被认为是测试方法，测试的时候不会被执行。

对每一类测试都需要编写一个`test_xxx()`方法。由于`unittest.TestCase`提供了很多内置的条件判断。

#### 8.3.1. 运行单元测试

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

```
$ python -m unittest mydict_test
.....
-----
Ran 5 tests in 0.000s

OK
```

#### 8.3.2. setUp和tearDown

可以在单元测试中编写两个特殊的`setUp()`和`tearDown()`方法。这两个方法会分别在每调用一个测试方法的前后分别被执行。

`setUp()`和`tearDown()`方法有什么用呢？设想你的测试需要启动一个数据库，这时，就可以在`setUp()`方法中连接数据库，在`tearDown()`方法中关闭数据库，这样，不必在每个测试方法中重复相同的代码：

```
class TestDict(unittest.TestCase):

    def setUp(self):
        print('setUp...')

    def tearDown(self):
        print('tearDown...')
```

## 8.4. 文档测试

```
def abs(n):
    '''
    Function to get absolute value of number.

    Example:

    >>> abs(1)
    1
    >>> abs(-1)
    1
    >>> abs(0)
    0
    ...
    return n if n >= 0 else (-n)
```

Python内置的“文档测试”（doctest）模块可以直接提取注释中的代码并执行测试。

doctest严格按照Python交互式命令行的输入和输出来判断测试结果是否正确。只有测试异常的时候，可以用...表示中间一大段烦人的输出。

当模块正常导入时，doctest不会被执行。只有在命令行直接运行时，才执行doctest。所以，不必担心doctest会在非测试环境下执行。

## 9. IO编程

IO在计算机中指Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由CPU这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要IO接口。

IO编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream就是数据从外面（磁盘、网络）流进内存，Output Stream就是数据从内存流到外面去。

由于CPU和内存的速度远远高于外设的速度，所以，在IO编程中，就存在速度严重不匹配的问题。

第一种是CPU等着，也就是程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，再接着往下执行，这种模式称为同步IO；

另一种方法是CPU不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步IO。

异步IO：回调模式、轮询模式

操作IO的能力都是由操作系统提供的，每一种编程语言都会把操作系统提供的低级C接口封装起来方便使用，Python也不例外。

### 9.1. 文件读写

在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

#### 9.1.1. 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('/test.txt', 'r')
```

标示符'r'表示读

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：

```
>>> f=open('/notfound.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '/notfound.txt'
```

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()
```

使用with语句

```
with open('/path/to/file', 'r') as f:
    print(f.read())
```

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
```

### 9.1.2. file-like Object

像open()函数返回的这种有个read()方法的对象, 在Python中统称为file-like Object。除了file外, 还可以是内存的字节流, 网络流, 自定义流等等。file-like Object不要求从特定类继承, 只要写个read()方法就行。

StringIO就是在内存中创建的file-like Object, 常用作临时缓冲。

### 9.1.3. 二进制文件

要读取二进制文件, 比如图片、视频等等, 用'rb'模式打开文件即可:

```
>>> f = open('/test.jpg', 'rb')
>>> f.read()
b'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' # 十六进制表示的字节
```

### 9.1.4. 字符编码

要读取非UTF-8编码的文本文件, 需要给open()函数传入encoding参数, 例如, 读取GBK编码的文件:

```
>>> f = open('/gbk.txt', 'r', encoding='gbk')
>>> f.read()
'测试'
```

遇到有些编码不规范的文件, 你可能会遇到UnicodeDecodeError, 因为在文本文件中可能夹杂了一些非法编码的字符。遇到这种情况, open()函数还接收一个errors参数, 表示如果遇到编码错误后如何处理。最简单的方式是直接忽略:

```
>>> f = open('/gbk.txt', 'r', encoding='gbk', errors='ignore')
```

### 9.1.5. 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('/test.txt', 'w')
>>> f.write('Hello, world!')
>>> f.close()
```

可以反复调用write()来写入文件, 但是务必要调用f.close()来关闭文件。当写文件时, 操作系统往往不会立刻把数据写入磁盘, 而是放到内存缓存起来, 空闲的时候再慢慢写入。只有调用close()方法时, 操作系统才保证把没有写入的数据全部写入磁盘。忘记调用close()的后果是数据可能只写了一部分到磁盘, 剩下的丢失了

```
with open('/Users/michael/test.txt', 'w') as f:
    f.write('Hello, world!')
```

要写入特定编码的文本文件, 给open()函数传入encoding参数, 将字符串自动转换成指定编码。

以'w'模式写入文件时, 如果文件已存在, 会直接覆盖(相当于删掉后新写入一个文件)。如果我们希望追加到文件末尾可以传入'a'以追加(append)模式写入。

## 9.2. StringIO和BytesIO

### 9.2.1. StringIO

数据读写不一定是文件, 也可以在内存中读写。

StringIO顾名思义就是在内存中读写str。

要把str写入StringIO, 我们需要先创建一个StringIO, 然后, 像文件一样写入即可:

```
>>> from io import StringIO
>>> f = StringIO()
>>> f.write('hello')
5
>>> f.write(' ')
1
>>> f.write('world!')
6
>>> print(f.getvalue())
hello world!
```

要读取StringIO，可以用一个str初始化StringIO，然后，像读文件一样读取：

```
>>> from io import StringIO
>>> f = StringIO('Hello!\nHi!\nGoodbye!')
>>> while True:
...     s = f.readline()
...     if s == '':
...         break
...     print(s.strip())
...
Hello!
Hi!
Goodbye!
```

### 9.2.2. BytesIO

BytesIO实现了在内存中读写bytes。

```
>>> from io import BytesIO
>>> f = BytesIO()
>>> f.write('中文'.encode('utf-8'))
6
>>> print(f.getvalue())
b'\xe4\xb8\xad\xe6\x96\x87'
```

写入的不是str，而是经过UTF-8编码的bytes。

和StringIO类似，可以用一个bytes初始化BytesIO，然后，像读文件一样读取：

```
>>> from io import BytesIO
>>> f = BytesIO(b'\xe4\xb8\xad\xe6\x96\x87')
>>> f.read()
b'\xe4\xb8\xad\xe6\x96\x87'
```

StringIO和BytesIO是在内存中操作str和bytes的方法，使得和读写文件具有一致的接口。

### 9.3. 操作文件和目录

操作系统提供的命令只是简单地调用了操作系统提供的接口函数，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。

```
>>> import os
>>> os.name # 操作系统类型
'posix'
```

### 详细的系统信息

```
>>> os.uname()
posix.uname_result(sysname='Darwin', nodename='MichaelMacPro.local', release='14.3.0', version='Darwin Kernel Version 14.3.0: Mon Mar 23 22:04:42 PDT 2015; root:xnu-2.02.2/~/RELEASE_ARM_T8020_14.3.0.Darwin14.3.0~1/RELEASE_ARM_T8020_14.3.0.Darwin14.3.0~1', machine='armv8-t8020')
```

注意uname()函数在Windows上不提供，也就是说，os模块的某些函数是跟操作系统相关的。

### 9.3.1. 环境变量

在操作系统中定义的环境变量，全部保存在`os.environ`这个变量中，可以直接查看：

```
>>> os.environ
environ({'VERSIONER_PYTHON_PREFER_32_BIT': 'no', 'TERM_PROGRAM_VERSION': '326', 'LOGNAME': 'michael', 'USER': 'michael', 'PATH': '/usr/t
```

要获取某个环境变量的值，可以调用`os.environ.get('key')`：

```
>>> os.environ.get('PATH')
'/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/local/mysql/bin'
>>> os.environ.get('x', 'default')
'default'
```

### 9.3.2. 操作系统和目录

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

```
# 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，这样可以正确处理不同操作系统的路径分隔符。在Linux/Unix/Mac下，os.path.join()返回这样的字符串：

```
part-1/part-2
```

而Windows下会返回这样的字符串：

```
part-1\part-2
```

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。

重命名文件、删除文件：

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

但是复制文件的函数在os模块中不存在。原因是复制文件并非由操作系统提供的系统调用。通过读写文件可以完成文件复制。

shutil模块提供了copyfile()的函数，可以看做是os模块的补充。

利用Python的特性来过滤文件。列出当前目录下的所有目录：

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

列出所有的.py文件：

```
>>> [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']
```

## 9.4. 序列化

在程序运行的过程中，所有的变量都是在内存中。

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, marshallng, flattening等等。

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。

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\x03q\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()
```



把对象从磁盘读到内存，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。

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

这个变量和原来的变量是完全不相干的对象，只是内容相同

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。

9.4.1. JSON

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

```
>>> 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，所以总是能正确地在Python的str与JSON的字符串之间转换。

Python的dict对象可以直接序列化为JSON的{}，不过，class表示对象序列化：

```
Traceback (most recent call last):
...
TypeError: <__main__.Student object at 0x10603cc50> is not JSON serializable
```

```
print(json.dumps(s, default=lambda obj: obj.__dict__))
```

因为通常class的实例都有一个\_\_dict\_\_属性，它就是一个dict，用来存储实例变量。也有少数例外，比如定义了\_\_slots\_\_的class。

同样的道理，如果要把JSON反序列化为一个Student对象实例，loads()方法首先转换出一个dict对象，然后，传入的object\_hook函数负责把dict转换为Student实例：

```
def dict2student(d):
    return Student(d['name'], d['age'], d['score'])
```

运行结果如下：

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'
>>> print(json.loads(json_str, object_hook=dict2student))
<__main__.Student object at 0x10cd3c190>
```

打印出的是反序列化的Student实例对象。

10. 进程和线程

- 多进程模式；
- 多线程模式；
- 多进程+多线程模式。

Python既支持多进程，又支持多线程。

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。**

## 10.1. 多进程

Unix/Linux操作系统提供了一个fork()系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是fork()调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

子进程永远返回0，而父进程返回子进程的ID。这样做的理由是，一个父进程可以fork出很多子进程，所以，**父进程要记下每个子进程的ID，而子进程只需要调用getppid()就可以拿到父进程的ID。**

Python的os模块封装了常见的系统调用，其中就包括fork，可以在Python程序中轻松创建子进程：

```
import os

print('Process (%s) start...' % os.getpid())
# Only works on Unix/Linux/Mac:
pid = os.fork()
if pid == 0:
    print('I am child process (%s) and my parent is %s.' % (os.getpid(), os.getppid()))
else:
    print('I (%s) just created a child process (%s).' % (os.getpid(), pid))
```

运行结果如下：

```
Process (876) start...
I (876) just created a child process (877).
I am child process (877) and my parent is 876.
```

**Windows没有fork调用**

有了fork调用，一个进程在接到新任务时就可以复制出一个子进程来处理新任务，常见的Apache服务器就是由父进程监听端口，每当有新的http请求时，就fork出子进程来处理新的http请求。

### 10.1.1. multiprocessing

multiprocessing模块是**跨平台版本的多进程模块**。

```
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.')
```

```
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。

### 10.1.2. 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.')
```

```
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.
```

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。

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。

```
p = Pool(4)
```

Pool的默认大小是CPU的核数。

### 10.1.3. 子进程

subprocess模块非常方便地启动一个子进程，然后控制其输入和输出。

```
import subprocess

print('$ nslookup www.python.org')
r = subprocess.call(['nslookup', 'www.python.org'])
print('Exit code:', r)
```

```
$ nslookup www.python.org
Server:      192.168.19.4
Address:     192.168.19.4#53

Non-authoritative answer:
www.python.org canonical name = python.map.fastly.net.
Name:   python.map.fastly.net
Address: 199.27.79.223

Exit code: 0
```

如果子进程还需要输入，则可以通过communicate()方法输入：

```
import subprocess

print('$ nslookup')
p = subprocess.Popen(['nslookup'], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
output, err = p.communicate(b'set q=mx\npython.org\nexit\n')
print(output.decode('utf-8'))
print('Exit code:', p.returncode)
```

上面的代码相当于在命令行执行命令nslookup，然后手动输入：

```
set q=mx
python.org
exit
```

```
$ nslookup
Server:      192.168.19.4
Address:     192.168.19.4#53

Non-authoritative answer:
python.org mail exchanger = 50 mail.python.org.

Authoritative answers can be found from:
mail.python.org internet address = 82.94.164.166
mail.python.org has AAAA address 2001:888:2000:d::a6

Exit code: 0
```

### 10.1.4. 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。

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()
```

```
Process to write: 50563
Put A to queue...
Process to read: 50564
Get A from queue.
Put B to queue...
Get B from queue.
Put C to queue...
Get C from queue.
```

在Unix/Linux下，multiprocessing模块封装了fork()调用，不需要关注fork()的细节。由于Windows没有fork调用，因此，multiprocessing需要“模拟”出fork的效果，父进程所有Python对象都必须通过pickle序列化再传到子进程去，所有，如果multiprocessing在Windows下调用失败了，要先考虑是不是pickle失败了。

在Unix/Linux下，可以使用fork()调用实现多进程。

要实现跨平台的多进程，可以使用multiprocessing模块。

进程间通信是通过Queue、Pipes等实现的。

## 10.2. 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。

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，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)
```

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。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.....

### 10.2.1. 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。

由于锁只有一个，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。

获得锁的线程用完时一定要释放锁，否则那些苦苦等待锁的线程将永远等待下去，成为死线程。所以用try...finally来确保锁一定会被释放。

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。

### 10.2.2. 多核CPU

启动与CPU核心数量相同的N个线程死循环，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。

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%。

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。**

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。

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。

## 10.3. 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。

```
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()
```

```
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。

可以理解为全球变量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。

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。

## 10.4. 进程vs线程

**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。

### 10.4.1. 线程切换

操作系统在切换进程或者线程时，需要先保存当前执行的现场环境（CPU寄存器状态、内存页等），然后，把新任务的执行环境准备好（恢复上次的寄存器状态，切换内存页等），才能开始执行。这个切换过程虽然很快，但是也需要耗费时间。如果有几千个任务同时进行，操作系统可能就主要忙着切换任务，根本没有多少时间去执行任务了，这种情况最常见的就是硬盘狂响，点窗口无反应，系统处于**假死状态**。

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。

### 10.4.2. 计算密集型 vs. 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。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的C语言替换用Python这样运行速度极低的脚本语言，完全无法提升运行效率。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**。

### 10.4.3. 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，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，**单线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。

## 10.5. 分布式进程

在Thread和Process中，应当**优选Process**，因为Process更稳定，而且，Process可以分布到多台机器上，而Thread最多只能分布到同一台机器的多个CPU上。

Python的multiprocessing模块不但支持多进程，其中managers子模块还支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到其他多个进程中，依靠网络通信。由于managers模块封装很好，不必了解网络通信的细节，就可以很容易地编写分布式多进程程序。

**通过managers模块把Queue通过网络暴露出去**，就可以让其他机器的进程访问Queue了。

**服务进程**，服务进程负责启动Queue，把Queue注册到网络上，然后往Queue里面写入任务：

```
# task_master.py

import random, time, queue
from multiprocessing.managers import BaseManager

# 发送任务的队列：
task_queue = queue.Queue()

# 接收结果的队列：
```

```

result_queue = queue.Queue()

# 从BaseManager继承的QueueManager:
class QueueManager(BaseManager):
    pass

# 把两个Queue都注册到网络上, callable参数关联了Queue对象:
QueueManager.register('get_task_queue', callable=lambda: task_queue)
QueueManager.register('get_result_queue', callable=lambda: result_queue)
# 绑定端口5000, 设置验证码'abc':
manager = QueueManager(address=('', 5000), authkey=b'abc')
# 启动Queue:
manager.start()
# 获得通过网络访问的Queue对象:
task = manager.get_task_queue()
result = manager.get_result_queue()
# 放几个任务进去:
for i in range(10):
    n = random.randint(0, 10000)
    print('Put task %d...' % n)
    task.put(n)
# 从result队列读取结果:
print('Try get results...')
for i in range(10):
    r = result.get(timeout=10)
    print('Result: %s' % r)
# 关闭:
manager.shutdown()
print('master exit.')

```

当在一台机器上写多进程程序时, 创建的Queue可以直接拿来用, 但是, 在分布式多进程环境下, 添加任务到Queue不可以直接对原始的task\_queue进行操作, 那样就绕过了QueueManager的封装, 必须通过manager.get\_task\_queue()获得的Queue接口添加。

然后, 在另一台机器上启动任务进程(本机上启动也可以):

```

# task_worker.py

import time, sys, queue
from multiprocessing.managers import BaseManager

# 创建类似的QueueManager:
class QueueManager(BaseManager):
    pass

# 由于这个QueueManager只从网络上获取Queue, 所以注册时只提供名字:
QueueManager.register('get_task_queue')
QueueManager.register('get_result_queue')

# 连接到服务器, 也就是运行task_master.py的机器:
server_addr = '127.0.0.1'
print('Connect to server %s...' % server_addr)
# 端口和验证码注意保持与task_master.py设置的完全一致:
m = QueueManager(address=(server_addr, 5000), authkey=b'abc')
# 从网络连接:
m.connect()
# 获取Queue的对象:
task = m.get_task_queue()
result = m.get_result_queue()
# 从task队列取任务, 并把结果写入result队列:
for i in range(10):
    try:
        n = task.get(timeout=1)
        print('run task %d * %d...' % (n, n))
        r = ' %d * %d = %d' % (n, n, n*n)
        time.sleep(1)
        result.put(r)
    except Queue.Empty:
        print('task queue is empty.')
# 处理结束:
print('worker exit.')

```

任务进程要通过网络连接到服务进程, 所以要指定服务进程的IP。

先启动task\_master.py服务进程:

```

$ python3 task_master.py
Put task 3411...
Put task 1605...
Put task 1398...
Put task 4729...
Put task 5300...
Put task 7471...
Put task 68...
Put task 4219...
Put task 339...

```

```
Put task 7866...
Try get results...
```

task\_master.py进程发送完任务后，开始等待result队列的结果。现在启动task\_worker.py进程：

```
$ python3 task_worker.py
Connect to server 127.0.0.1...
run task 3411 * 3411...
run task 1605 * 1605...
run task 1398 * 1398...
run task 4729 * 4729...
run task 5300 * 5300...
run task 7471 * 7471...
run task 68 * 68...
run task 4219 * 4219...
run task 339 * 339...
run task 7866 * 7866...
worker exit.
```

task\_master中显示结果：

```
Result: 3411 * 3411 = 11634921
Result: 1605 * 1605 = 2576025
Result: 1398 * 1398 = 1954404
Result: 4729 * 4729 = 22363441
Result: 5300 * 5300 = 28090000
Result: 7471 * 7471 = 55815841
Result: 68 * 68 = 4624
Result: 4219 * 4219 = 17799961
Result: 339 * 339 = 114921
Result: 7866 * 7866 = 61873956
```

这就是一个简单但真正的分布式计算，把代码稍加改造，启动多个worker，就可以把任务分布到几台甚至几十台机器上，比如把计算n\*n的代码换成发送邮件，就实现了邮件队列的异步发送。

Queue对象存储在task\_master.py进程中：

□

而Queue之所以能通过网络访问，就是通过QueueManager实现的。由于QueueManager管理的的多不止一个Queue，所以，要给每个Queue的网络调用接口起个名字，比如get\_task\_queue。

authkey有什么用？这是为了保证两台机器正常通信，不被其他机器恶意干扰。如果task\_worker.py的authkey和task\_master.py的authkey不一致，肯定连接不上。

Python的分布式进程接口简单，封装良好，适合需要把繁重任务分布到多台机器的环境下。

注意Queue的作用是用来传递任务和接收结果，每个任务的描述数据量要尽量小。比如发送一个处理日志文件的任务，就不要发送几百兆的日志文件本身，而是发送日志文件存放的完整路径，由Worker进程再去共享的磁盘上读取文件。

## 11. 正则表达式

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。

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\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。

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。



## 11.1. re模块

Python提供re模块，包含所有正则表达式的功能。

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'
```

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'
```

```
>>> 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')
```

## 11.2. 切分字符串

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

如果用户输入了一组标签，用正则表达式来把不规范的输入转化成正确的数组。

## 11.3. 分组

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。

识别合法时间：

```
>>> t = '19:05:30'
>>> m = re.match(r'^([0-9]|1[0-9]|2[0-3])\:([0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|6[0-9])\:([0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|6[0-9])$', t)
>>> m.groups()
('19', '05', '30')
```

## 11.4. 贪婪匹配

正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。

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')
```

## 11.5. 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。

## 12. 常用内建模块

Python之所以自称“batteries included”，就是因为内置了许多非常有用的模块，无需额外安装和配置，即可直接使用。

### 12.1. datetime

datetime是Python处理日期和时间的标准库。

#### 12.1.1. 获取当前日期和时间

```
>>> from datetime import datetime
>>> now = datetime.now() # 获取当前datetime
>>> print(now)
2015-05-18 16:28:07.198690
>>> print(type(now))
<class 'datetime.datetime'>
```

datetime是模块，datetime模块还包含一个datetime类，通过from datetime import datetime导入的才是datetime这个类。

如果仅导入import datetime，则必须引用全名datetime.datetime。

datetime.now()返回当前日期和时间，其类型是datetime。

#### 12.1.2. 获取指定日期和时间

```
>>> from datetime import datetime
>>> dt = datetime(2015, 4, 19, 12, 20) # 用指定日期时间创建datetime
>>> print(dt)
2015-04-19 12:20:00
```

#### 12.1.3. datetime转换为timestamp

1970年1月1日 00:00:00 UTC+00:00时区的时刻称为epoch time，记为0（1970年以前的时间timestamp为负数），当前时间就是相对于epoch time的秒数，称为timestamp。

```
timestamp = 0 = 1970-1-1 00:00:00 UTC+0:00
```

对应的北京时间是：

```
timestamp = 0 = 1970-1-1 08:00:00 UTC+8:00
```

timestamp的值与时区毫无关系，因为timestamp一旦确定，其UTC时间就确定了，转换到任意时区的时间也是完全确定的，这就是为什么计算机存储的当前时间是以timestamp表示的，因为全球各地的计算机在任意时刻的timestamp都是完全相同的（假定时间已校准）。

```
>>> from datetime import datetime
>>> dt = datetime(2015, 4, 19, 12, 20) # 用指定日期时间创建datetime
>>> dt.timestamp() # 把datetime转换为timestamp
1429417200.0
```

Python的timestamp是一个浮点数。如果有小数位，小数位表示毫秒数。

某些编程语言（如Java和JavaScript）的timestamp使用整数表示毫秒数，这种情况下只需要把timestamp除以1000就得到Python的浮点表示方法。

#### 12.1.4. timestamp转换为datetime

```
>>> from datetime import datetime
>>> t = 1429417200.0
```

```
>>> print(datetime.fromtimestamp(t))
2015-04-19 12:20:00
```

注意到timestamp是一个浮点数，它没有时区的概念，而datetime是有时区的。上述转换是在timestamp和本地时间做转换。

本地时间是指当前操作系统设定的时区。例如北京时区是东8区，则本地时间：

```
2015-04-19 12:20:00
```

实际上就是UTC+8:00时区的时间：

```
2015-04-19 12:20:00 UTC+8:00
```

而此刻的格林威治标准时间与北京时间差了8小时，也就是UTC+0:00时区的时间应该是：

```
2015-04-19 04:20:00 UTC+0:00
```

timestamp也可以直接被转换到UTC标准时区的时间：

```
>>> from datetime import datetime
>>> t = 1429417200.0
>>> print(datetime.fromtimestamp(t)) # 本地时间
2015-04-19 12:20:00
>>> print(datetime.utcfromtimestamp(t)) # UTC时间
2015-04-19 04:20:00
```

### 12.1.5. str转换为datetime

转换方法是通过datetime.strptime()实现，需要一个日期和时间的格式化字符串：

```
>>> from datetime import datetime
>>> cday = datetime.strptime('2015-6-1 18:19:59', '%Y-%m-%d %H:%M:%S')
>>> print(cday)
2015-06-01 18:19:59
```

字符串'%Y-%m-%d %H:%M:%S'规定了日期和时间部分的格式。

注意转换后的datetime是没有时区信息的。

### 12.1.6. datetime转换为str

如果已经有了datetime对象，要把它格式化为字符串显示给用户，就需要转换为str，转换方法是通过strftime()实现的，同样需要一个日期和时间的格式化字符串：

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> print(now.strftime('%a, %b %d %H:%M'))
Mon, May 05 16:28
```

### 12.1.7. datetime加减

对日期和时间进行加减实际上就是把datetime往后或往前计算，得到新的datetime。加减可以直接用+和-运算符，不过需要导入timedelta这个类：

```
>>> from datetime import datetime, timedelta
>>> now = datetime.now()
>>> now
datetime.datetime(2015, 5, 18, 16, 57, 3, 540997)
>>> now + timedelta(hours=10)
datetime.datetime(2015, 5, 19, 2, 57, 3, 540997)
>>> now - timedelta(days=1)
datetime.datetime(2015, 5, 17, 16, 57, 3, 540997)
>>> now + timedelta(days=2, hours=12)
datetime.datetime(2015, 5, 21, 4, 57, 3, 540997)
```

可见，使用timedelta你可以很容易地算出前几天和后几天的时刻。

### 12.1.8. 本地时间转换为UTC时间

本地时间是指系统设定时区的时间，例如北京时间是UTC+8:00时区的时间，而UTC时间指UTC+0:00时区的时间。

一个datetime类型有一个时区属性tzinfo，但是默认为None，所以无法区分这个datetime到底是哪个时区，除非强行给datetime设置一个时区：

```
>>> from datetime import datetime, timedelta, timezone
>>> tz_utc_8 = timezone(timedelta(hours=8)) # 创建时区UTC+8:00
>>> now = datetime.now()
>>> now
datetime.datetime(2015, 5, 18, 17, 2, 10, 871012)
>>> dt = now.replace(tzinfo=tz_utc_8) # 强制设置为UTC+8:00
>>> dt
datetime.datetime(2015, 5, 18, 17, 2, 10, 871012, tzinfo=datetime.timezone(datetime.timedelta(0, 28800)))
```

### 12.1.9. 时区转换

可以先通过`utcnow()`拿到当前的UTC时间，再转换为任意时区的时间：

```
# 拿到UTC时间，并强制设置时区为UTC+0:00:
>>> utc_dt = datetime.utcnow().replace(tzinfo=timezone.utc)
>>> print(utc_dt)
2015-05-18 09:05:12.377316+00:00
# astimezone()将转换时区为北京时间:
>>> bj_dt = utc_dt.astimezone(timezone(timedelta(hours=8)))
>>> print(bj_dt)
2015-05-18 17:05:12.377316+08:00
# astimezone()将转换时区为东京时间:
>>> tokyo_dt = utc_dt.astimezone(timezone(timedelta(hours=9)))
>>> print(tokyo_dt)
2015-05-18 18:05:12.377316+09:00
# astimezone()将bj_dt转换时区为东京时间:
>>> tokyo_dt2 = bj_dt.astimezone(timezone(timedelta(hours=9)))
>>> print(tokyo_dt2)
2015-05-18 18:05:12.377316+09:00
```

时区转换的关键在于，拿到一个`datetime`时，要获知其正确的时区，然后强制设置时区，作为基准时间。

利用带时区的`datetime`，通过`astimezone()`方法，可以转换到任意时区。

注：不是必须从UTC+0:00时区转换到其他时区，任何带时区的`datetime`都可以正确转换，例如`bj_dt`到`tokyo_dt`的转换。

`datetime`表示的时间需要时区信息才能确定一个特定的时间，否则只能视为本地时间。

如果要存储`datetime`，最佳方法是将其转换为`timestamp`再存储，因为`timestamp`的值与时区完全无关。

## 12.2. collections

### 12.2.1. namedtuple

定义坐标：

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(1, 2)
>>> p.x
1
>>> p.y
2
```

`namedtuple`是一个函数，它用来创建一个自定义的`tuple`对象，并且规定了`tuple`元素的个数，并且可以用属性而不是索引来引用`tuple`的某个元素。

用`namedtuple`可以很方便地定义一种数据类型，它具备`tuple`的不变性，又可以根据属性来引用，使用十分方便。

### 12.2.2. deque

使用`list`存储数据时，按索引访问元素很快，但是插入和删除元素就很慢了，因为`list`是线性存储，数据量大的时候，插入和删除效率很低。

`deque`是为了高效实现插入和删除操作的双向列表，适合用于队列和栈：

```
>>> from collections import deque
>>> q = deque(['a', 'b', 'c'])
>>> q.append('x')
>>> q.appendleft('y')
>>> q
deque(['y', 'a', 'b', 'c', 'x'])
```

`deque`除了实现`list`的`append()`和`pop()`外，还支持`appendleft()`和`popleft()`，这样就可以非常高效地往头部添加或删除元素。

### 12.2.3. defaultdict

使用`dict`时，如果引用的`Key`不存在，就会抛出`KeyError`。如果希望`key`不存在时，返回一个默认值，就可以用`defaultdict`：

```
>>> from collections import defaultdict
>>> dd = defaultdict(lambda: 'N/A')
>>> dd['key1'] = 'abc'
>>> dd['key1'] # key1存在
'abc'
>>> dd['key2'] # key2不存在，返回默认值
'N/A'
```

注意默认值是调用函数返回的，而函数在创建`defaultdict`对象时传入。

除了在`Key`不存在时返回默认值，`defaultdict`的其他行为跟`dict`是完全一样的。

### 12.2.4. OrderedDict

使用dict时，Key是无序的。在对dict做迭代时，无法确定Key的顺序。

如果要保持**Key**的顺序，可以用OrderedDict：

```
>>> from collections import OrderedDict
>>> d = dict([('a', 1), ('b', 2), ('c', 3)])
>>> d # dict的Key是无序的
{'a': 1, 'c': 3, 'b': 2}
>>> od = OrderedDict([('a', 1), ('b', 2), ('c', 3)])
>>> od # OrderedDict的Key是有序的
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

注意，**OrderedDict的Key会按照插入的顺序排列**，不是Key本身排序：

```
>>> od = OrderedDict()
>>> od['z'] = 1
>>> od['y'] = 2
>>> od['x'] = 3
>>> list(od.keys()) # 按照插入的Key的顺序返回
['z', 'y', 'x']
```

OrderedDict可以实现一个**FIFO（先进先出）的dict**，当容量超出限制时，先删除最早添加的Key：

```
from collections import OrderedDict

class LastUpdatedOrderedDict(OrderedDict):

    def __init__(self, capacity):
        super(LastUpdatedOrderedDict, self).__init__()
        self._capacity = capacity

    def __setitem__(self, key, value):
        containsKey = 1 if key in self else 0
        if len(self) - containsKey >= self._capacity:
            last = self.popitem(last=False)
            print('remove:', last)
        if containsKey:
            del self[key]
            print('set:', (key, value))
        else:
            print('add:', (key, value))
        OrderedDict.__setitem__(self, key, value)
```

### 12.2.5. Counter

Counter是一个简单的计数器。

```
>>> from collections import Counter
>>> c = Counter()
>>> for ch in 'programming':
...     c[ch] = c[ch] + 1
...
>>> c
Counter({'g': 2, 'm': 2, 'r': 2, 'a': 1, 'i': 1, 'o': 1, 'n': 1, 'p': 1})
```

Counter实际上也是dict的一个子类，上面的结果可以看出，字符'g'、'm'、'r'各出现了两次，其他字符各出现了一次。

## 12.3. base64

Base64是一种用64个字符来表示任意二进制数据的方法。

二进制文件包含很多无法显示和打印的字符，所以，如果要让记事本这样的文本处理软件能处理二进制数据，就需要一个二进制到字符串的转换方法。Base64是一种最常见的二进制编码方法。

Base64的原理：

首先，准备一个包含64个字符的数组：

```
['A', 'B', 'C', ... 'a', 'b', 'c', ... '0', '1', ... '+', '/']
```

然后，对二进制数据进行处理，每3个字节一组，一共是3x8=24bit，划为4组，每组正好6个bit：

□

Base64编码会把3字节的二进制数据编码为4字节的文本数据，**长度增加33%**。好处是编码后的文本数据可以在邮件正文、网页等直接显示。

如果要编码的二进制数据不是3的倍数，最后会剩下1个或2个字节。Base64用\x00字节在末尾补足后，再在编码的末尾加上1个或2个=号，表示补了多少字节，解码的时候，会自动去掉。

Python内置的base64可以直接进行base64的编解码：

```
>>> import base64
```

```
>>> base64.b64encode(b'binary\x00string')
b'YmluYXJ5AHN0cmluZw=='
>>> base64.b64decode(b'YmluYXJ5AHN0cmluZw==')
b'binary\x00string'
```

Python内置的base64可以直接进行base64的编解码：

```
>>> import base64
>>> base64.b64encode(b'binary\x00string')
b'YmluYXJ5AHN0cmluZw=='
>>> base64.b64decode(b'YmluYXJ5AHN0cmluZw==')
b'binary\x00string'
```

由于标准的Base64编码后可能出现字符+和/，在URL中就不能直接作为参数，所以又有一种"url safe"的base64编码，其实就是把字符+和/分别变成-和\_：

```
>>> base64.b64encode(b'i\x07\x1d\xfb\xef\xff')
b'abcd+// '
>>> base64.urlsafe_b64encode(b'i\x07\x1d\xfb\xef\xff')
b'abcd-__ '
>>> base64.urlsafe_b64decode('abcd-__ ')
b'i\x07\x1d\xfb\xef\xff'
```

还可以自己定义64个字符的排列顺序，这样就可以自定义Base64编码，不过，通常情况下完全没有必要。

Base64是一种通过查表的编码方法，不能用于加密，即使使用自定义的编码表也不行。

Base64适用于小段内容的编码，比如数字证书签名、Cookie的内容等。

由于=字符也可能出现在Base64编码中，但=用在URL、Cookie里面会造成歧义，所以，很多Base64编码后会把=去掉：

```
# 标准Base64:
'abcd' -> 'YWJjZA=='
# 自动去掉=:
'abcd' -> 'YWJjZA'
```

去掉=后怎么解码呢？因为Base64是把3个字节变为4个字节，所以，Base64编码的长度永远是4的倍数，因此，需要加上=把Base64字符串的长度变为4的倍数，就可以正常解码了。

Base64是一种任意二进制到文本字符串的编码方法，常用于在URL、Cookie、网页中传输少量二进制数据。

## 12.4. struct

## 12.5. hashlib

### 12.5.1. 摘要算法简介

Python的hashlib提供了常见的摘要算法，如MD5，SHA1等等。

**摘要算法**又称**哈希算法**、**散列算法**。它通过一个函数，把任意长度的数据转换为一个长度固定的数据串（通常用16进制的字符串表示）。

摘要算法就是通过摘要函数f()对任意长度的数据data计算出固定长度的摘要digest，目的是为了发现原始数据是否被人篡改过。

摘要算法之所以能指出数据是否被篡改过，就是因为摘要函数是一个单向函数，计算f(data)很容易，但通过digest反推data却非常困难。而且，对原始数据做一个bit的修改，都会导致计算出的摘要完全不同。

```
import hashlib

md5 = hashlib.md5()
md5.update('how to use md5 in python hashlib?'.encode('utf-8'))
print(md5.hexdigest())
```

计算结果如下：

```
d26a53750bc40b38b65a520292f69306
```

如果数据量很大，可以分块多次调用update()，最后计算的结果是一样的：

```
import hashlib

md5 = hashlib.md5()
md5.update('how to use md5 in '.encode('utf-8'))
md5.update('python hashlib?'.encode('utf-8'))
print(md5.hexdigest())
```

MD5是最常见的摘要算法，速度很快，生成结果是固定的128 bit字节，通常用一个**32位的16进制字符串**表示。

摘要算法是SHA1：

```
import hashlib

sha1 = hashlib.sha1()
sha1.update('how to use sha1 in '.encode('utf-8'))
sha1.update('python hashlib?'.encode('utf-8'))
print(sha1.hexdigest())
```

SHA1的结果是160 bit字节，通常用一个**40位的16进制字符串**表示。

比SHA1更安全的算法是SHA256和SHA512，不过越安全的算法不仅越慢，而且摘要长度更长。

任何摘要算法都是把无限多的数据集合映射到一个有限的集合中。这种情况称为碰撞。

### 12.5.2. 摘要算法应用

如果以明文保存用户口令，如果数据库泄露，所有用户的口令就落入黑客的手里。此外，网站运维人员是可以访问数据库的，也就是能获取到所有用户的口令。

正确的保存口令的方式是不存储用户的明文口令，而是存储用户口令的摘要，比如MD5。

当用户登录时，首先计算用户输入的明文口令的MD5，然后和数据库存储的MD5对比，如果一致，说明口令输入正确，如果不一致，口令肯定错误。

## 12.6. hmac

为了防止黑客通过彩虹表根据哈希值反推原始口令，在计算哈希的时候，不能仅针对原始输入计算，需要增加一个salt来使得相同的输入也能得到不同的哈希，这样，大大增加了黑客破解的难度。

如果salt是自己随机生成的，通常计算MD5时采用md5(message + salt)。但实际上，把salt看做一个“口令”，加salt的哈希就是：计算一段message的哈希时，根据不通口令计算出不同的哈希。要验证哈希值，必须同时提供正确的口令。

这实际上就是Hmac算法：Keyed-Hashing for Message Authentication。它通过一个标准算法，在计算哈希的过程中，把key混入计算过程中。

和自定义的加salt算法不同，Hmac算法针对所有哈希算法都通用，无论是MD5还是SHA-1。采用Hmac替代自己的salt算法，可以使程序算法更标准化，也更安全。

Python自带的hmac模块实现了标准的Hmac算法。

首先需要准备待计算的原始消息message，随机key，哈希算法，这里采用MD5，使用hmac的代码如下：

```
>>> import hmac
>>> message = b'Hello, world!'
>>> key = b'secret'
>>> h = hmac.new(key, message, digestmod='MD5')
>>> # 如果消息很长，可以多次调用h.update(msg)
>>> h.hexdigest()
'fa4ee7d173f2d97ee79022d1a7355bcf'
```

可见使用hmac和普通hash算法非常类似。hmac输出的长度和原始哈希算法的长度一致。需要注意传入的key和message都是bytes类型，str类型需要首先编码为bytes。

## 12.7. itertools

Python的内建模块itertools提供了非常有用的用于操作迭代对象的函数。

“无限”迭代器：

```
>>> import itertools
>>> natuals = itertools.count(1)
>>> for n in natuals:
...     print(n)
...
1
2
3
...
```

因为count()会创建一个无限的迭代器，所以上述代码会打印出自然数序列，根本停不下来，只能按Ctrl+C退出。

cycle()会把传入的一个序列无限重复下去：

```
>>> import itertools
>>> cs = itertools.cycle('ABC') # 注意字符串也是序列的一种
>>> for c in cs:
...     print(c)
...
'A'
'B'
'C'
'A'
'B'
```

```
'C'
...
```

同样停不下来。

`repeat()`负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数：

```
>>> ns = itertools.repeat('A', 3)
>>> for n in ns:
...     print(n)
...
A
A
A
```

无限序列只有在for迭代时才会无限地迭代下去，如果只是创建了一个迭代对象，它不会事先把无限个元素生成出来，事实上也不可能在内存中创建无限多个元素。

无限序列虽然可以无限迭代下去，但是通常会通过\*\*`takewhile()`

```
>>> naturals = itertools.count(1)
>>> ns = itertools.takewhile(lambda x: x <= 10, naturals)
>>> list(ns)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

### 12.7.1. chain()

`chain()`可以把一组迭代对象串联起来，形成一个更大的迭代器：

```
>>> for c in itertools.chain('ABC', 'XYZ'):
...     print(c)
# 迭代效果: 'A' 'B' 'C' 'X' 'Y' 'Z'
```

### 12.7.2. groupby()

`groupby()`把迭代器中相邻的重复元素挑出来放在一起：

```
>>> for key, group in itertools.groupby('AAABBBCCAAA'):
...     print(key, list(group))
...
A ['A', 'A', 'A']
B ['B', 'B', 'B']
C ['C', 'C']
A ['A', 'A', 'A']
```

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组的，而函数返回值作为组的key。如果要忽略大小写分组，就可以让元素'A'和'a'都返回相同的key：

```
>>> for key, group in itertools.groupby('AaBBbCCAAa', lambda c: c.upper()):
...     print(key, list(group))
...
A ['A', 'a', 'a']
B ['B', 'B', 'b']
C ['c', 'C']
A ['A', 'A', 'a']
```

## 12.8. contextlib

```
with open('/path/to/file', 'r') as f:
    f.read()
```

并不是只有`open()`函数返回的fp对象才能使用with语句。

任何对象，只要正确实现了上下文管理，就可以用于with语句。

实现上下文管理是通过`__enter__`和`__exit__`这两个方法实现的。

```
class Query(object):

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

    def __enter__(self):
        print('Begin')
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        if exc_type:
            print('Error')
        else:
            print('End')
```



```
def query(self):
    print('Query info about %s...' % self.name)
```

```
with Query('Bob') as q:
    q.query()
```

### 12.8.1. @contextmanager

Python的标准库contextlib提供了更简单的写法，上面的代码可以改写如下：

```
from contextlib import contextmanager

class Query(object):

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

    def query(self):
        print('Query info about %s...' % self.name)

@contextmanager
def create_query(name):
    print('Begin')
    q = Query(name)
    yield q
    print('End')
```

@contextmanager这个decorator接受一个generator，用yield语句把with ... as var把变量输出出去，然后，with语句就可以正常地工作了：

```
with create_query('Bob') as q:
    q.query()
```

在某段代码执行前后自动执行特定代码：

```
@contextmanager
def tag(name):
    print("<%s>" % name)
    yield
    print("</%s>" % name)

with tag("h1"):
    print("hello")
    print("world")
```

```
<h1>
hello
world
</h1>
```

代码的执行顺序是：

- with语句首先执行yield之前的语句，因此打印出<h1>；
- yield调用会执行with语句内部的所有语句，因此打印出hello和world；
- 最后执行yield之后的语句，打印出</h1>。
- 

因此，@contextmanager通过编写generator来简化上下文管理。

### 12.8.2. @closing

如果一个对象没有实现上下文，不能把它用于with语句。这个时候，可以用closing()来把该对象变为上下文对象。例如，用with语句使用urlopen()：

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

closing也是一个经过@contextmanager装饰的generator，这个generator编写起来其实非常简单：

```
@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

它的作用就是把任意对象变为上下文对象，并支持with语句。

@contextlib还有一些其他decorator，便于编写更简洁的代码。

## 12.9. urllib

urllib提供了一系列用于操作URL的功能。

### 12.9.1. Get

urllib的request模块可以非常方便地抓取URL内容，也就是发送一个GET请求到指定的页面，然后返回HTTP的响应：

例如，对豆瓣的一个URL<https://api.douban.com/v2/book/2129650>进行抓取，并返回响应：

```
from urllib import request

with request.urlopen('https://api.douban.com/v2/book/2129650') as f:
    data = f.read()
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', data.decode('utf-8'))
```

可以看到HTTP响应的头和JSON数据：

```
Status: 200 OK
Server: nginx
Date: Tue, 26 May 2015 10:02:27 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 2049
Connection: close
Expires: Sun, 1 Jan 2006 01:00:00 GMT
Pragma: no-cache
Cache-Control: must-revalidate, no-cache, private
X-DAE-Node: pid11
Data: {"rating":{"max":10,"numRaters":16,"average":"7.4","min":0},"subtitle":"","author":["廖雪峰编著"],"pubdate":"2007-6",...}
```

如果我们要想模拟浏览器发送GET请求，就需要使用Request对象，通过往Request对象添加HTTP头，我们就可以把请求伪装成浏览器。例如，模拟iPhone 6去请求豆瓣首页：

```
from urllib import request

req = request.Request('http://www.douban.com/')
req.add_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/12A43503.1~1/AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/12A43503.1~1')
with request.urlopen(req) as f:
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', f.read().decode('utf-8'))
```

这样豆瓣会返回适合iPhone的移动版网页：

```
...
<meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1.0, minimum-scale=1.0, maximum-scale=1.0">
<meta name="format-detection" content="telephone=no">
<link rel="apple-touch-icon" sizes="57x57" href="http://img4.douban.com/pics/cardkit/launcher/57.png" />
...
```

### 12.9.2. Post

如果要以POST发送一个请求，只需要把参数data以bytes形式传入。

模拟微博登录，先读取登录的邮箱和口令，然后按照weibo.cn的登录页的格式以username=xxx&password=xxx的编码传入：

```
from urllib import request, parse

print('Login to weibo.cn...')
email = input('Email: ')
passwd = input('Password: ')
login_data = parse.urlencode([
    ('username', email),
    ('password', passwd),
    ('entry', 'mweibo'),
    ('client_id', ''),
    ('savestate', '1'),
    ('ec', ''),
    ('pagerefer', 'https://passport.weibo.cn/signin/welcome?entry=mweibo&r=http%3A%2F%2Fm.weibo.cn%2F')
])

req = request.Request('https://passport.weibo.cn/sso/login')
req.add_header('Origin', 'https://passport.weibo.cn')
```

```
req.add_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 M
req.add_header('Referer', 'https://passport.weibo.cn/signin/login?entry=mweibo&res=wel&wm=3349&r=http%3A%2F%2Fm.weibo.cn%2F')

with request.urlopen(req, data=login_data.encode('utf-8')) as f:
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', f.read().decode('utf-8'))
```

如果登录成功，获得的响应如下：

```
Status: 200 OK
Server: nginx/1.2.0
...
Set-Cookie: SSOLoginState=1432620126; path=/; domain=weibo.cn
...
Data: {"retcode":20000000,"msg":"","data":{"...","uid":"1658384301"}}
```

如果登录失败，获得的响应如下：

```
...
Data: {"retcode":50011015,"msg":"\u7528\u6237\u540d\u6216\u5bc6\u7801\u9519\u8bef","data":{"username":"example@python.org","errline":536
```

### 12.9.3. Handler

如果还需要更复杂的控制，比如通过一个Proxy去访问网站，需要利用ProxyHandler来处理，示例代码如下：

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')
opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
with opener.open('http://www.example.com/login.html') as f:
    pass
```

urllib提供的功能就是利用程序去执行各种HTTP请求。如果要模拟浏览器完成特定功能，需要把请求伪装成浏览器。伪装的方法是先监控浏览器发出的请求，再根据浏览器的请求头来伪装，**User-Agent头就是用来标识浏览器的**。

## 12.10. XML

### 12.10.1. DOM vs SAX

操作XML有两种方法：DOM和SAX。DOM会把整个XML读入内存，解析为树，因此占用内存大，解析慢，优点是可以任意遍历树的节点。SAX是流模式，边读边解析，占用内存小，解析快，缺点是需要自己处理事件。

正常情况下，优先考虑SAX，因为DOM实在太占内存。

在Python中使用SAX解析XML非常简洁，通常的事件是start\_element，end\_element和char\_data，准备好这3个函数，然后就可以解析xml了。

当SAX解析器读到一个节点时：

```
<a href="/">python</a>
```

会产生3个事件：

start\_element事件，在读取时；

char\_data事件，在读取python时；

end\_element事件，在读取时。

```
from xml.parsers.expat import ParserCreate

class DefaultSaxHandler(object):
    def start_element(self, name, attrs):
        print('sax:start_element: %s, attrs: %s' % (name, str(attrs)))

    def end_element(self, name):
        print('sax:end_element: %s' % name)

    def char_data(self, text):
        print('sax:char_data: %s' % text)

xml = r'''<?xml version="1.0"?>
<ol>
  <li><a href="/python">Python</a></li>
  <li><a href="/ruby">Ruby</a></li>
</ol>
...
'''
```

```
handler = DefaultSaxHandler()
parser = ParserCreate()
parser.StartElementHandler = handler.start_element
parser.EndElementHandler = handler.end_element
parser.CharacterDataHandler = handler.char_data
parser.Parse(xml)
```

需要注意的是读取一大段字符串时，CharacterDataHandler可能被多次调用，所以需要自己保存起来，在EndElementHandler里面再合并。

了解析XML外，如何生成XML呢？99%的情况下需要生成的XML结构都是非常简单的，因此，最简单也是最有效的生成XML的方法是拼接字符串：

```
L = []
L.append(r'<?xml version="1.0">')
L.append(r'<root>')
L.append(encode('some & data'))
L.append(r'</root>')
return ''.join(L)
```

不要用复杂的XML，改成JSON。

## 12.11. HTMLParser

HTML本质上是XML的子集，但是HTML的语法没有XML那么严格，所以不能用标准的DOM或SAX来解析HTML。

Python提供了HTMLParser来非常方便地解析HTML：

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):

    def handle_starttag(self, tag, attrs):
        print('<%s>' % tag)

    def handle_endtag(self, tag):
        print('</%s>' % tag)

    def handle_startendtag(self, tag, attrs):
        print('<%s/>' % tag)

    def handle_data(self, data):
        print(data)

    def handle_comment(self, data):
        print('<!--', data, '-->')

    def handle_entityref(self, name):
        print('&%s;' % name)

    def handle_charref(self, name):
        print('&#%s;' % name)

parser = MyHTMLParser()
parser.feed('<<<html>
<head></head>
<body>
<!-- test html parser -->
<p>Some <a href="#">html</a> HTML&nbsp;tutorial...<br>END</p>
</body></html>')

```

feed()方法可以多次调用，也就是不一定一次把整个HTML字符串都塞进去，可以一部分一部分塞进去。

特殊字符有两种，一种是英文表示的，一种是数字表示的Å，这两种字符都可以通过Parser解析出来。

## 13. 常用第三方模块

基本上，所有的第三方模块都会在PyPI - the Python Package Index上注册，只要找到对应的模块名字，即可用pip安装。

### 13.1. Pillow

PIL: Python Imaging Library，已经是Python平台事实上的图像处理标准库了。PIL功能非常强大，但API却非常简单易用。

由于PIL仅支持到Python 2.7，加上年久失修，于是一群志愿者在PIL的基础上创建了兼容的版本，名字叫Pillow，支持最新Python 3.x，又加入了许多新特性。

#### 13.1.1. 操作图像

图像缩放

```
from PIL import Image
```

```
# 打开一个jpg图像文件，注意是当前路径：
im = Image.open('test.jpg')
# 获得图像尺寸：
w, h = im.size
print('Original image size: %sx%s' % (w, h))
# 缩放到50%:
im.thumbnail((w//2, h//2))
print('Resize image to: %sx%s' % (w//2, h//2))
# 把缩放后的图像用jpeg格式保存：
im.save('thumbnail.jpg', 'jpeg')
```

其他功能如切片、旋转、滤镜、输出文字、调色板等一应俱全。

比如，模糊效果也只需几行代码：

```
from PIL import Image, ImageFilter

# 打开一个jpg图像文件，注意是当前路径：
im = Image.open('test.jpg')
# 应用模糊滤镜：
im2 = im.filter(ImageFilter.BLUR)
im2.save('blur.jpg', 'jpeg')
```

□

PIL的ImageDraw提供了一系列绘图方法，可以直接绘图。比如要生成字母验证码图片：

```
from PIL import Image, ImageDraw, ImageFont, ImageFilter

import random

# 随机字母：
def rndChar():
    return chr(random.randint(65, 90))

# 随机颜色1:
def rndColor():
    return (random.randint(64, 255), random.randint(64, 255), random.randint(64, 255))

# 随机颜色2:
def rndColor2():
    return (random.randint(32, 127), random.randint(32, 127), random.randint(32, 127))

# 240 x 60:
width = 60 * 4
height = 60
image = Image.new('RGB', (width, height), (255, 255, 255))
# 创建Font对象:
font = ImageFont.truetype('Arial.ttf', 36)
# 创建Draw对象:
draw = ImageDraw.Draw(image)
# 填充每个像素:
for x in range(width):
    for y in range(height):
        draw.point((x, y), fill=rndColor())
# 输出文字:
for t in range(4):
    draw.text((60 * t + 10, 10), rndChar(), font=font, fill=rndColor2())
# 模糊:
image = image.filter(ImageFilter.BLUR)
image.save('code.jpg', 'jpeg')
imgae.show()
```

## 13.2. requests

Python内置的urllib模块，用于访问网络资源。但是，用起来比较麻烦，而且，缺少很多实用的高级功能。

requests是一个Python第三方库，处理URL资源特别方便。

### 13.2.1. 使用requests

要通过GET访问一个页面，只需要几行代码：

```
>>> import requests
>>> r = requests.get('https://www.douban.com/') # 豆瓣首页
>>> r.status_code
200
>>> r.text
r.text
'<!DOCTYPE HTML>\n<html>\n<head>\n<meta name="description" content="提供图书、电影、音乐唱片的推荐、评论和...'
```

对于带参数的URL，传入一个dict作为params参数：

```
>>> r = requests.get('https://www.douban.com/search', params={'q': 'python', 'cat': '1001'})
>>> r.url # 实际请求的URL
'https://www.douban.com/search?q=python&cat=1001'
```

requests自动检测编码，可以使用encoding属性查看：

```
>>> r.encoding
'utf-8'
```

无论响应是文本还是二进制内容，我们都可以用content属性获得bytes对象：

```
>>> r.content
b'<!DOCTYPE html>\n<html>\n<head>\n<meta http-equiv="Content-Type" content="text/html; charset=utf-8">\n...'
```

requests的方便之处还在于，对于特定类型的响应，例如JSON，可以直接获取：

```
>>> r = requests.get('https://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20weather.forecast%20where%20woeid%20%3D%202151330&')
>>> r.json()
{'query': {'count': 1, 'created': '2017-11-17T07:14:12Z', ...}}
```

需要传入HTTP Header时，传入一个dict作为headers参数：

```
>>> r = requests.get('https://www.douban.com/', headers={'User-Agent': 'Mozilla/5.0 (iPhone; CPU iPhone OS 11_0 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.0 Mobile/15E148 Safari/604.1'})
>>> r.text
'<!DOCTYPE html>\n<html>\n<head>\n<meta charset="UTF-8">\n <title>豆瓣(手机版)</title>...'
```

要发送POST请求，只需要把get()方法变成post()，然后传入data参数作为POST请求的数据：

```
>>> r = requests.post('https://accounts.douban.com/login', data={'form_email': 'abc@example.com', 'form_password': '123456'})
```

requests默认使用application/x-www-form-urlencoded对POST数据编码。如果要传递JSON数据，可以直接传入json参数：

```
params = {'key': 'value'}
r = requests.post(url, json=params) # 内部自动序列化为JSON
```

类似的，上传文件需要更复杂的编码格式，但是requests把它简化成files参数：

```
>>> upload_files = {'file': open('report.xls', 'rb')}
>>> r = requests.post(url, files=upload_files)
```

在读取文件时，注意务必使用'rb'即二进制模式读取，这样获取的bytes长度才是文件的长度。

把post()方法替换为put(), delete()等，就可以以PUT或DELETE方式请求资源。

除了能轻松获取响应内容外，requests对获取HTTP响应的其他信息也非常简单。例如，获取响应头：

```
>>> r.headers
{'Content-Type': 'text/html; charset=utf-8', 'Transfer-Encoding': 'chunked', 'Content-Encoding': 'gzip', ...}
>>> r.headers['Content-Type']
'text/html; charset=utf-8'
```

requests对Cookie做了特殊处理，使得我们不必解析Cookie就可以轻松获取指定的Cookie：

```
>>> r.cookies['ts']
'example_cookie_12345'
```

要在请求中传入Cookie，只需准备一个dict传入cookies参数：

```
>>> cs = {'token': '12345', 'status': 'working'}
>>> r = requests.get(url, cookies=cs)
```

最后，要指定超时，传入以秒为单位的timeout参数：

```
>>> r = requests.get(url, timeout=2.5) # 2.5秒后超时
```

## 13.3. chardet

虽然Python提供了Unicode表示的str和bytes两种数据类型，并且可以通过encode()和decode()方法转换，但是，在不知道编码的情况下，对bytes做decode()不好做。

对于未知编码的bytes，要把它转换成str，需要先“猜测”编码。猜测的方式是先收集各种编码的特征字符，根据特征字符判断，就能有很大概率“猜对”。

### 13.3.1. 使用chardet

```
>>> data = '高高高原上草，一岁一枯荣'.encode('gbk')
>>> chardet.detect(data)
{'encoding': 'GB2312', 'confidence': 0.7407407407407407, 'language': 'Chinese'}
```

检测正确的概率是74%。

## 13.4. psutil

在Python中获取系统信息使用psutil这个第三方模块。顾名思义，psutil = process and system utilities，不仅可以通过一两行代码实现系统监控，还可以跨平台使用，支持Linux / UNIX / OSX / Windows等。

### 13.4.1. 获取CPU信息

```
>>> import psutil
>>> psutil.cpu_count() # CPU逻辑数量
4
>>> psutil.cpu_count(logical=False) # CPU物理核心
2
# 2说明是双核超线程，4则是4核非超线程
```

统计CPU的用户 / 系统 / 空闲时间：

```
>>> psutil.cpu_times()
scputimes(user=10963.31, nice=0.0, system=5138.67, idle=356102.45)
```

再实现类似top命令的CPU使用率，每秒刷新一次，累计10次：

```
>>> for x in range(10):
...     psutil.cpu_percent(interval=1, percpu=True)
...
[14.0, 4.0, 4.0, 4.0]
[12.0, 3.0, 4.0, 3.0]
[8.0, 4.0, 3.0, 4.0]
[12.0, 3.0, 3.0, 3.0]
[18.8, 5.1, 5.9, 5.0]
[10.9, 5.0, 4.0, 3.0]
[12.0, 5.0, 4.0, 5.0]
[15.0, 5.0, 4.0, 4.0]
[19.0, 5.0, 5.0, 4.0]
[9.0, 3.0, 2.0, 3.0]
```

### 13.4.2. 获取内存信息

使用psutil获取物理内存和交换内存信息，分别使用：

```
>>> psutil.virtual_memory()
svmmem(total=8589934592, available=2866520064, percent=66.6, used=7201386496, free=216178688, active=3342192640, inactive=2650341376, wire=0)
>>> psutil.swap_memory()
sswap(total=1073741824, used=150732800, free=923009024, percent=14.0, sin=10705981440, sout=40353792)
```

返回的是字节为单位的整数，可以看到，总内存大小是8589934592 = 8 GB，已用7201386496 = 6.7 GB，使用了66.6%。

而交换区大小是1073741824 = 1 GB。

### 13.4.3. 获取磁盘信息

可以通过psutil获取磁盘分区、磁盘使用率和磁盘IO信息：

```
>>> psutil.disk_partitions() # 磁盘分区信息
[sdiskpart(device='/dev/disk1', mountpoint='/', fstype='hfs', opts='rw,local,rootfs,dovolfs,journaled,multilabel')]
>>> psutil.disk_usage('/') # 磁盘使用情况
sdiskusage(total=998982549504, used=390880133120, free=607840272384, percent=39.1)
>>> psutil.disk_io_counters() # 磁盘IO
sdiskio(read_count=988513, write_count=274457, read_bytes=14856830464, write_bytes=17509420032, read_time=2228966, write_time=1618405)
```

可以看到，磁盘'/'的总容量是998982549504 = 930 GB，使用了39.1%。文件格式是HFS，opts中包含rw表示可读写，journaled表示支持日志。

### 13.4.4. 获取网络信息

psutil可以获取网络接口和网络连接信息：

```
>>> psutil.net_io_counters() # 获取网络读写字节 / 包的个数
snetio(bytes_sent=3885744870, bytes_recv=10357676702, packets_sent=10613069, packets_recv=10423357, errin=0, errout=0, dropin=0, dropout=0)
>>> psutil.net_if_addrs() # 获取网络接口信息
{
    'lo': [
        {'ifindex': 1, 'address': '0:0:0:0:0:0:0:0', 'netmask': '0:0:0:0:0:0:0:0', 'broadcast': '0:0:0:0:0:0:0:0', 'ptype': 'loopback'}
    ],
    'en0': [
        {'ifindex': 2, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en1': [
        {'ifindex': 3, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en2': [
        {'ifindex': 4, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en3': [
        {'ifindex': 5, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en4': [
        {'ifindex': 6, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en5': [
        {'ifindex': 7, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en6': [
        {'ifindex': 8, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en7': [
        {'ifindex': 9, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en8': [
        {'ifindex': 10, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en9': [
        {'ifindex': 11, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en10': [
        {'ifindex': 12, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en11': [
        {'ifindex': 13, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en12': [
        {'ifindex': 14, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en13': [
        {'ifindex': 15, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en14': [
        {'ifindex': 16, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en15': [
        {'ifindex': 17, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en16': [
        {'ifindex': 18, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en17': [
        {'ifindex': 19, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en18': [
        {'ifindex': 20, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en19': [
        {'ifindex': 21, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en20': [
        {'ifindex': 22, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en21': [
        {'ifindex': 23, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en22': [
        {'ifindex': 24, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en23': [
        {'ifindex': 25, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en24': [
        {'ifindex': 26, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en25': [
        {'ifindex': 27, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en26': [
        {'ifindex': 28, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en27': [
        {'ifindex': 29, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en28': [
        {'ifindex': 30, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en29': [
        {'ifindex': 31, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en30': [
        {'ifindex': 32, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en31': [
        {'ifindex': 33, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en32': [
        {'ifindex': 34, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en33': [
        {'ifindex': 35, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en34': [
        {'ifindex': 36, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en35': [
        {'ifindex': 37, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en36': [
        {'ifindex': 38, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en37': [
        {'ifindex': 39, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en38': [
        {'ifindex': 40, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en39': [
        {'ifindex': 41, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en40': [
        {'ifindex': 42, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en41': [
        {'ifindex': 43, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en42': [
        {'ifindex': 44, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en43': [
        {'ifindex': 45, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en44': [
        {'ifindex': 46, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en45': [
        {'ifindex': 47, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en46': [
        {'ifindex': 48, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en47': [
        {'ifindex': 49, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en48': [
        {'ifindex': 50, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en49': [
        {'ifindex': 51, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en50': [
        {'ifindex': 52, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en51': [
        {'ifindex': 53, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en52': [
        {'ifindex': 54, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en53': [
        {'ifindex': 55, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en54': [
        {'ifindex': 56, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en55': [
        {'ifindex': 57, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en56': [
        {'ifindex': 58, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en57': [
        {'ifindex': 59, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en58': [
        {'ifindex': 60, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en59': [
        {'ifindex': 61, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en60': [
        {'ifindex': 62, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en61': [
        {'ifindex': 63, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en62': [
        {'ifindex': 64, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en63': [
        {'ifindex': 65, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en64': [
        {'ifindex': 66, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en65': [
        {'ifindex': 67, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en66': [
        {'ifindex': 68, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en67': [
        {'ifindex': 69, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en68': [
        {'ifindex': 70, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en69': [
        {'ifindex': 71, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en70': [
        {'ifindex': 72, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en71': [
        {'ifindex': 73, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en72': [
        {'ifindex': 74, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en73': [
        {'ifindex': 75, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en74': [
        {'ifindex': 76, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en75': [
        {'ifindex': 77, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en76': [
        {'ifindex': 78, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en77': [
        {'ifindex': 79, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en78': [
        {'ifindex': 80, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en79': [
        {'ifindex': 81, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en80': [
        {'ifindex': 82, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en81': [
        {'ifindex': 83, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en82': [
        {'ifindex': 84, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en83': [
        {'ifindex': 85, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en84': [
        {'ifindex': 86, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en85': [
        {'ifindex': 87, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en86': [
        {'ifindex': 88, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en87': [
        {'ifindex': 89, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en88': [
        {'ifindex': 90, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en89': [
        {'ifindex': 91, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en90': [
        {'ifindex': 92, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en91': [
        {'ifindex': 93, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en92': [
        {'ifindex': 94, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en93': [
        {'ifindex': 95, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en94': [
        {'ifindex': 96, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en95': [
        {'ifindex': 97, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en96': [
        {'ifindex': 98, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en97': [
        {'ifindex': 99, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en98': [
        {'ifindex': 100, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en99': [
        {'ifindex': 101, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en100': [
        {'ifindex': 102, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en101': [
        {'ifindex': 103, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en102': [
        {'ifindex': 104, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en103': [
        {'ifindex': 105, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en104': [
        {'ifindex': 106, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en105': [
        {'ifindex': 107, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en106': [
        {'ifindex': 108, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en107': [
        {'ifindex': 109, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en108': [
        {'ifindex': 110, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en109': [
        {'ifindex': 111, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en110': [
        {'ifindex': 112, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en111': [
        {'ifindex': 113, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en112': [
        {'ifindex': 114, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en113': [
        {'ifindex': 115, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en114': [
        {'ifindex': 116, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en115': [
        {'ifindex': 117, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en116': [
        {'ifindex': 118, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en117': [
        {'ifindex': 119, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en118': [
        {'ifindex': 120, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en119': [
        {'ifindex': 121, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en120': [
        {'ifindex': 122, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en121': [
        {'ifindex': 123, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en122': [
        {'ifindex': 124, 'address': '08:00:27:00:00:00', 'netmask': 'ff:ff:ff:ff:ff:ff', 'broadcast': 'ff:ff:ff:ff:ff:ff', 'ptype': 'ether'}
    ],
    'en123':
```

```

'lo0': [snic(family=<AddressFamily.AF_INET: 2>, address='127.0.0.1', netmask='255.0.0.0'), ...],
'en1': [snic(family=<AddressFamily.AF_INET: 2>, address='10.0.1.80', netmask='255.255.255.0'), ...],
'en0': [...],
'en2': [...],
'bridge0': [...]
}
>>> psutil.net_if_stats() # 获取网络接口状态
{
'lo0': snicstats(isup=True, duplex=<NicDuplex.NIC_DUPLEX_UNKNOWN: 0>, speed=0, mtu=16384),
'en0': snicstats(isup=True, duplex=<NicDuplex.NIC_DUPLEX_UNKNOWN: 0>, speed=0, mtu=1500),
'en1': snicstats(...),
'en2': snicstats(...),
'bridge0': snicstats(...)
}

```

要获取当前网络连接信息，使用`net_connections()`：

```

>>> psutil.net_connections()
Traceback (most recent call last):
...
PermissionError: [Errno 1] Operation not permitted

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
...
psutil.AccessDenied: psutil.AccessDenied (pid=3847)

```

可能会得到一个`AccessDenied`错误，原因是`psutil`获取信息也是要走系统接口，而获取网络连接信息需要`root`权限，这种情况下，可以退出Python交互环境，用`sudo`重新启动：

psutil

阅读：4

用Python来编写脚本简化日常的运维工作是Python的一个重要用途。在Linux下，有许多系统命令可以让我们时刻监控系统运行的状态，如`ps`，`top`，`free`等等。要获取这些系统

在Python中获取系统信息的另一个好办法是使用`psutil`这个第三方模块。顾名思义，`psutil = process and system utilities`，它不仅可以通过一两行代码实现系统监控，

安装psutil

如果安装了Anaconda，`psutil`就已经可用了。否则，需要在命令行下通过`pip`安装：

```
$ pip install psutil
```

如果遇到`Permission denied`安装失败，请加上`sudo`重试。

获取CPU信息

我们先来获取CPU的信息：

```

>>> import psutil
>>> psutil.cpu_count() # CPU逻辑数量
4
>>> psutil.cpu_count(logical=False) # CPU物理核心
2
# 2说明是双核超线程，4则是4核非超线程
统计CPU的用户 / 系统 / 空闲时间：

>>> psutil.cpu_times()
scputimes(user=10963.31, nice=0.0, system=5138.67, idle=356102.45)
再实现类似top命令的CPU使用率，每秒刷新一次，累计10次：

```

```

>>> for x in range(10):
...     psutil.cpu_percent(interval=1, percpu=True)
...
[14.0, 4.0, 4.0, 4.0]
[12.0, 3.0, 4.0, 3.0]
[8.0, 4.0, 3.0, 4.0]
[12.0, 3.0, 3.0, 3.0]
[18.8, 5.1, 5.9, 5.0]
[10.9, 5.0, 4.0, 3.0]
[12.0, 5.0, 4.0, 5.0]
[15.0, 5.0, 4.0, 4.0]
[19.0, 5.0, 5.0, 4.0]
[9.0, 3.0, 2.0, 3.0]
获取内存信息

```

使用`psutil`获取物理内存和交换内存信息，分别使用：

```

>>> psutil.virtual_memory()
svmem(total=8589934592, available=2866520064, percent=66.6, used=7201386496, free=216178688, active=3342192640, inactive=2650341376, wire=0)
>>> psutil.swap_memory()
sswap(total=1073741824, used=150732800, free=923009024, percent=14.0, sin=10705981440, sout=40353792)
返回的是字节为单位的整数，可以看到，总内存大小是8589934592 = 8 GB，已用7201386496 = 6.7 GB，使用了66.6%。

```



而交换区大小是1073741824 = 1 GB。

#### 获取磁盘信息

可以通过psutil获取磁盘分区、磁盘使用率和磁盘IO信息：

```
>>> psutil.disk_partitions() # 磁盘分区信息
[sdiskpart(device='/dev/disk1', mountpoint='/', fstype='hfs', opts='rw,local,rootfs,dovolfs,journaled,multilabel'')]
>>> psutil.disk_usage('/') # 磁盘使用情况
sdiskusage(total=998982549504, used=390880133120, free=607840272384, percent=39.1)
>>> psutil.disk_io_counters() # 磁盘IO
sdiskio(read_count=988513, write_count=274457, read_bytes=14856830464, write_bytes=17509420032, read_time=2228966, write_time=1618405)
可以看到，磁盘 '/' 的总容量是998982549504 = 930 GB，使用了39.1%。文件格式是HFS，opts中包含rw表示可读写，journaled表示支持日志。
```

#### 获取网络信息

psutil可以获取网络接口和网络连接信息：

```
>>> psutil.net_io_counters() # 获取网络读写字节 / 包的个数
snetio(bytes_sent=3885744870, bytes_recv=10357676702, packets_sent=10613069, packets_recv=10423357, errin=0, errout=0, dropin=0, dropout=0)
>>> psutil.net_if_addrs() # 获取网络接口信息
{
  'lo0': [snic(family=<AddressFamily.AF_INET: 2>, address='127.0.0.1', netmask='255.0.0.0'), ...],
  'en1': [snic(family=<AddressFamily.AF_INET: 2>, address='10.0.1.80', netmask='255.255.255.0'), ...],
  'en0': [...],
  'en2': [...],
  'bridge0': [...]
}
>>> psutil.net_if_stats() # 获取网络接口状态
{
  'lo0': snicstats(isup=True, duplex=<NicDuplex.NIC_DUPLEX_UNKNOWN: 0>, speed=0, mtu=16384),
  'en0': snicstats(isup=True, duplex=<NicDuplex.NIC_DUPLEX_UNKNOWN: 0>, speed=0, mtu=1500),
  'en1': snicstats(...),
  'en2': snicstats(...),
  'bridge0': snicstats(...)
}
```

要获取当前网络连接信息，使用net\_connections()：

```
>>> psutil.net_connections()
Traceback (most recent call last):
...
PermissionError: [Errno 1] Operation not permitted
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
...
psutil.AccessDenied: psutil.AccessDenied (pid=3847)
```

你可能会得到一个AccessDenied错误，原因是psutil获取信息也是要走系统接口，而获取网络连接信息需要root权限，这种情况下，可以退出Python交互环境，用sudo重新启动。

```
$ sudo python3
Password: *****
Python 3.6.3 ... on darwin
Type "help", ... for more information.
>>> import psutil
>>> psutil.net_connections()
[
  sconn(fd=83, family=<AddressFamily.AF_INET6: 30>, type=1, laddr=addr(ip='::127.0.0.1', port=62911), raddr=addr(ip='::127.0.0.1', port=62905), status='ESTABLISHED', pid=3725),
  sconn(fd=84, family=<AddressFamily.AF_INET6: 30>, type=1, laddr=addr(ip='::127.0.0.1', port=62905), raddr=addr(ip='::127.0.0.1', port=62905), status='ESTABLISHED', pid=3725),
  sconn(fd=93, family=<AddressFamily.AF_INET6: 30>, type=1, laddr=addr(ip=':::', port=8080), raddr=(), status='LISTEN', pid=3725),
  sconn(fd=103, family=<AddressFamily.AF_INET6: 30>, type=1, laddr=addr(ip='::127.0.0.1', port=62918), raddr=addr(ip='::127.0.0.1', port=62918), status='ESTABLISHED', pid=3725),
  sconn(fd=105, family=<AddressFamily.AF_INET6: 30>, type=1, ..., pid=3725),
  sconn(fd=106, family=<AddressFamily.AF_INET6: 30>, type=1, ..., pid=3725),
  sconn(fd=107, family=<AddressFamily.AF_INET6: 30>, type=1, ..., pid=3725),
  ...
  sconn(fd=27, family=<AddressFamily.AF_INET: 2>, type=2, ..., pid=1)
]
```

### 13.4.5. 获取进程信息

通过psutil可以获取到所有进程的详细信息：

```
>>> psutil.pids() # 所有进程ID
[3865, 3864, 3863, 3856, 3855, 3853, 3776, ..., 45, 44, 1, 0]
>>> p = psutil.Process(3776) # 获取指定进程ID=3776，其实就是当前Python交互环境
>>> p.name() # 进程名称
'python3.6'
>>> p.exe() # 进程exe路径
'/Users/michael/anaconda3/bin/python3.6'
>>> p.cwd() # 进程工作目录
'/Users/michael'
>>> p.cmdline() # 进程启动的命令行
['python3']
>>> p.ppid() # 父进程ID
```

```

3765
>>> p.parent() # 父进程
<psutil.Process(pid=3765, name='bash') at 4503144040>
>>> p.children() # 子进程列表
[]
>>> p.status() # 进程状态
'running'
>>> p.username() # 进程用户名
'michael'
>>> p.create_time() # 进程创建时间
1511052731.120333
>>> p.terminal() # 进程终端
'/dev/ttys002'
>>> p.cpu_times() # 进程使用的CPU时间
pcputimes(user=0.081150144, system=0.053269812, children_user=0.0, children_system=0.0)
>>> p.memory_info() # 进程使用的内存
pmem(rss=8310784, vms=2481725440, pfaults=3207, pageins=18)
>>> p.open_files() # 进程打开的文件
[]
>>> p.connections() # 进程相关网络连接
[]
>>> p.num_threads() # 进程的线程数量
1
>>> p.threads() # 所有线程信息
[thread(id=1, user_time=0.090318, system_time=0.062736)]
>>> p.environ() # 进程环境变量
{'SHELL': '/bin/bash', 'PATH': '/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin...', 'PWD': '/Users/michael', 'LANG': 'zh_CN.UTF-8', ...}
>>> p.terminate() # 结束进程
Terminated: 15 <-- 自己把自己结束了

```

和获取网络连接类似，获取一个root用户的进程需要root权限，启动Python交互环境或者.py文件时，需要sudo权限。

psutil提供了一个test()函数，可以模拟出ps命令的效果：

```

$ sudo python3
Password: *****
Python 3.6.3 ... on darwin
Type "help", ... for more information.
>>> import psutil
>>> psutil.test()
USER      PID %MEM  VSZ   RSS TTY      START    TIME  COMMAND
root         0  24.0 74270628 2016380 ?        Nov18   40:51  kernel_task
root         1   0.1 2494140   9484 ?        Nov18   01:39  launchd
root        44   0.4 2519872   36404 ?        Nov18   02:02  UserEventAgent
root        45   ? 2474032   1516 ?        Nov18   00:14  syslogd
root        47   0.1 2504768   8912 ?        Nov18   00:03  kextd
root        48   0.1 2505544   4720 ?        Nov18   00:19  fseventsd
_appleeven  52   0.1 2499748   5024 ?        Nov18   00:00  appleeventsd
root        53   0.1 2500592   6132 ?        Nov18   00:02  configd
...

```

## 14. virtualenv

virtualenv用来为一个应用创建一套“隔离”的Python运行环境。

## 15. 图形界面

Python支持多种图形界面的第三方库，包括：

- Tk
- wxWidgets
- Qt
- GTK

等等。

### 15.1. Tkinter

Tkinter封装了访问Tk的接口；

Tk是一个图形库，支持多个操作系统，使用Tcl语言开发；

Tk会调用操作系统提供的本地GUI接口，完成最终的GUI。

第一步是导入Tkinter包的所有内容：

```
from tkinter import *
```

第二步是从Frame派生一个Application类，这是所有Widget的父容器：

```
class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

    def createWidgets(self):
        self.helloLabel = Label(self, text='Hello, world!')
        self.helloLabel.pack()
        self.quitButton = Button(self, text='Quit', command=self.quit)
        self.quitButton.pack()
```

在GUI中，每个Button、Label、输入框等，都是一个Widget。Frame则是可以容纳其他Widget的Widget，所有的Widget组合起来就是一棵树。

pack()方法把Widget加入到父容器中，并实现布局。pack()是最简单的布局，grid()可以实现更复杂的布局。

在createWidgets()方法中，创建一个Label和一个Button，当Button被点击时，触发self.quit()使程序退出。

第三步，实例化Application，并启动消息循环：

```
app = Application()
# 设置窗口标题:
app.master.title('Hello World')
# 主消息循环:
app.mainloop()
```

GUI程序的主线程负责监听来自操作系统的消息，并依次处理每一条消息。因此，如果消息处理非常耗时，就需要在新线程中处理。

运行这个GUI程序，可以看到下面的窗口：

□

点击“Quit”按钮或者窗口的“X”结束程序。

## 15.2. 输入文本

加入一个文本框，让用户可以输入文本，然后点按钮后，弹出消息对话框。

```
from tkinter import *
import tkinter.messagebox as messagebox

class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

    def createWidgets(self):
        self.nameInput = Entry(self)
        self.nameInput.pack()
        self.alertButton = Button(self, text='Hello', command=self.hello)
        self.alertButton.pack()

    def hello(self):
        name = self.nameInput.get() or 'world'
        messagebox.showinfo('Message', 'Hello, %s' % name)

app = Application()
# 设置窗口标题:
app.master.title('Hello World')
# 主消息循环:
app.mainloop()
```

当用户点击按钮时，触发hello()，通过self.nameInput.get()获得用户输入的文本后，使用tkMessageBox.showinfo()可以弹出消息对话框。

程序运行结果如下：

□

如果是非常复杂的GUI程序，用操作系统原生支持的语言和库来编写。

## 16. 网络编程

用Python进行网络编程，就是在Python程序本身这个进程内，连接别的服务器进程的通信端口进行通信。

### 16.1. TCP/IP简介

为了把全世界的所有不同类型的计算机都连接起来，就必须规定一套全球通用的协议，为了实现互联网这个目标，互联网协议簇（Internet Protocol Suite）就是通用协议标准。Internet是由inter和net两个单词组合起来的，原意就是连接“网络”的网络，有了Internet，任何私有网络，只要支持这个协议，就可以联入互联网。

因为互联网协议包含了上百种协议标准，但是最重要的两个协议是TCP和IP协议，所以，大家把互联网的协议简称TCP/IP协议。

IP地址实际上是一个32位整数（称为IPv4），以字符串表示的IP地址如192.168.0.1实际上是把32位整数按8位分组后的数字表示，目的是便于阅读。

IPv6地址实际上是一个128位整数，它是目前使用的IPv4的升级版，以字符串表示类似于2001:0db8:85a3:0042:1000:8a2e:0370:7334。

一个进程也可能同时与多个计算机建立链接，因此它会申请很多端口。

## 16.2. TCP编程

Socket是网络编程的一个抽象概念。通常用一个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。

### 16.2.1. 客户端

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。

```
# 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))
```

注意参数是一个tuple，包含地址和端口号。

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。

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，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，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。

### 16.2.2. 服务器

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。

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)
```

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：

```
# 监听端口：
s.bind(('127.0.0.1', 9999))
```

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：

```
def tcplink(sock, addr):
    print('Accept new connection from %s:%s...' % addr)
    sock.send(b'Welcome!')
    while True:
        data = sock.recv(1024)
        time.sleep(1)
        if not data or data.decode('utf-8') == 'exit':
            break
        sock.send(('Hello, %s!' % data.decode('utf-8')).encode('utf-8'))
    sock.close()
    print('Connection from %s:%s closed.' % addr)
```

连接建立后，服务器首先发一条欢迎消息，然后等待客户端数据，并加上Hello再发送给客户端。如果客户端发送了exit字符串，就直接关闭连接。

要测试这个服务器程序，我们还需要编写一个客户端程序：

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 建立连接：
s.connect(('127.0.0.1', 9999))
# 接收欢迎消息：
print(s.recv(1024).decode('utf-8'))
for data in [b'Michael', b'Tracy', b'Sarah']:
    # 发送数据：
    s.send(data)
    print(s.recv(1024).decode('utf-8'))
s.send(b'exit')
s.close()
```

打开两个命令行窗口，一个运行服务器程序，另一个运行客户端程序，就可以看到效果了：

□

需要注意的是，客户端程序运行完毕就退出了，而服务器程序会永远运行下去，必须按Ctrl+C退出程序。

用TCP协议进行Socket编程在Python中十分简单，对于客户端，要主动连接服务器的IP和指定端口，对于服务器，要首先监听指定端口，然后，对每一个新的连接，创建一个线程或进程来处理。通常，服务器程序会无限运行下去。

同一个端口，被一个Socket绑定了以后，就不能被别的Socket绑定了。

## 16.3. UDP编程

TCP是建立可靠连接，并且通信双方都可以以流的形式发送数据。相对TCP，UDP则是面向无连接的协议。

使用UDP协议时，不需要建立连接，只需要知道对方的IP地址和端口号，就可以直接发数据包。但是，能不能到达就不知道了。

虽然用UDP传输数据不可靠，但它的优点是和TCP比，速度快，对于不要求可靠到达的数据，就可以使用UDP协议。

和TCP类似，使用UDP的通信双方也分为客户端和服务端。服务器首先需要绑定端口：

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# 绑定端口:
s.bind(('127.0.0.1', 9999))
```

创建Socket时，SOCK\_DGRAM指定了这个Socket的类型是UDP。绑定端口和TCP一样，但是不需要调用listen()方法，而是直接接收来自任何客户端的数据：

```
print('Bind UDP on 9999...')
while True:
    # 接收数据:
    data, addr = s.recvfrom(1024)
    print('Received from %s:%s.' % addr)
    s.sendto(b'Hello, %s!' % data, addr)
```

recvfrom()方法返回数据和客户端的地址与端口，这样，服务器收到数据后，直接调用sendto()就可以把数据用UDP发给客户端。

注意这里省掉了多线程，因为这个例子很简单。

客户端使用UDP时，首先仍然创建基于UDP的Socket，然后，不需要调用connect()，直接通过sendto()给服务器发数据：

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
for data in [b'Michael', b'Tracy', b'Sarah']:
    # 发送数据:
    s.sendto(data, ('127.0.0.1', 9999))
    # 接收数据:
    print(s.recv(1024).decode('utf-8'))
s.close()
```

从服务器接收数据仍然调用recv()方法。

仍然用两个命令行分别启动服务器和客户端测试，结果如下：

□

服务器绑定UDP端口和TCP端口互不冲突，也就是说，UDP的9999端口与TCP的9999端口可以各自绑定。

## 17. 电子邮件

### 17.1. STMP发送邮件

#### 17.1.1. 发送HTML邮件

#### 17.1.2. 发送附件

#### 17.1.3. 发送图片

#### 17.1.4. 同时支持HTML和Plain格式

#### 17.1.5. 加密SMTP

### 17.2. POP3收取邮件

#### 17.2.1. 通过POP3下载邮件

#### 17.2.2. 解析邮件

## 18. 访问数据库

### 18.1. 使用SQLite

SQLite是一种嵌入式数据库，它的数据库就是一个文件。由于SQLite本身是C写的，而且体积很小，所以，经常被集成到各种应用程序中，甚至在iOS和Android的App中都可以集成。

Python就内置了SQLite3，所以，在Python中使用SQLite，不需要安装任何东西，直接使用。

表是数据库中存放关系数据的集合，一个数据库里面通常都包含多个表，比如学生的表，班级的表，学校的表，等等。表和表之间通过外键关联。

要操作关系数据库，首先需要连接到数据库，一个数据库连接称为Connection；

连接到数据库后，需要打开游标，称之为Cursor，通过Cursor执行SQL语句，然后，获得执行结果。

Python定义了一套操作数据库的API接口，任何数据库要连接到Python，只需要提供符合Python标准的数据库驱动即可。

```
# 导入SQLite驱动:
```

```
>>> import sqlite3
# 连接到SQLite数据库
# 数据库文件是test.db
# 如果文件不存在，会自动在当前目录创建:
>>> conn = sqlite3.connect('test.db')
# 创建一个Cursor:
>>> cursor = conn.cursor()
# 执行一条SQL语句，创建user表:
>>> cursor.execute('create table user (id varchar(20) primary key, name varchar(20))')
<sqlite3.Cursor object at 0x10f8aa260>
# 继续执行一条SQL语句，插入一条记录:
>>> cursor.execute('insert into user (id, name) values (\'1\', \'Michael\')')
<sqlite3.Cursor object at 0x10f8aa260>
# 通过rowcount获得插入的行数:
>>> cursor.rowcount
1
# 关闭Cursor:
>>> cursor.close()
# 提交事务:
>>> conn.commit()
# 关闭Connection:
>>> conn.close()
```

```
>>> conn = sqlite3.connect('test.db')
>>> cursor = conn.cursor()
# 执行查询语句:
>>> cursor.execute('select * from user where id=?', ('1',))
<sqlite3.Cursor object at 0x10f8aa340>
# 获得查询结果集:
>>> values = cursor.fetchall()
>>> values
[(\'1\', \'Michael\')]
>>> cursor.close()
>>> conn.close()
```

使用Python的DB-API时，只要搞清楚Connection和Cursor对象，打开后一定记得关闭，就可以放心地使用。

使用Cursor对象执行insert, update, delete语句时，执行结果由rowcount返回影响的行数，就可以拿到执行结果。

使用Cursor对象执行select语句时，通过fetchall()可以拿到结果集。结果集是一个list，每个元素都是一个tuple，对应一行记录。

如果SQL语句带有参数，那么需要把参数按照位置传递给execute()方法，有几个?占位符就必须对应几个参数，例如：

```
cursor.execute('select * from user where name=? and pwd=?', ('abc', 'password'))
```

SQLite支持常见的标准SQL语句以及几种常见的数据类型。

在Python中操作数据库时，要先导入数据库对应的驱动，然后，通过Connection对象和Cursor对象操作数据。

要确保打开的Connection对象和Cursor对象都正确地被关闭，否则，资源就会泄露。

## 18.2. 使用MySQL

```
# 导入MySQL驱动:
>>> import mysql.connector
# 注意把password设为你的root口令:
>>> conn = mysql.connector.connect(user='root', password='password', database='test')
>>> cursor = conn.cursor()
# 创建user表:
>>> cursor.execute('create table user (id varchar(20) primary key, name varchar(20))')
# 插入一行记录，注意MySQL的占位符是%s:
>>> cursor.execute('insert into user (id, name) values (%s, %s)', ['1', 'Michael'])
>>> cursor.rowcount
1
# 提交事务:
>>> conn.commit()
>>> cursor.close()
# 运行查询:
>>> cursor = conn.cursor()
>>> cursor.execute('select * from user where id = %s', ('1',))
>>> values = cursor.fetchall()
>>> values
[(\'1\', \'Michael\')]
# 关闭Cursor和Connection:
>>> cursor.close()
True
>>> conn.close()
```

由于Python的DB-API定义都是通用的，所以，操作MySQL的数据库代码和SQLite类似。

执行INSERT等操作后要调用commit()提交事务：

MySQL的SQL占位符是%s。

## 18.3. 使用SQLAlchemy

数据库表是一个二维表，包含多行多列。把一个表的内容用Python的数据结构表示出来的话，可以用一个list表示多行，list的每一个元素是tuple，表示一行记录，比如，包含id和name的user表：

```
[
    ('1', 'Michael'),
    ('2', 'Bob'),
    ('3', 'Adam')
]
```

Python的DB-API返回的数据结构就是像上面这样表示的。

但是用tuple表示一行很难看出表的结构。如果把一个tuple用class实例来表示，就可以更容易地看出表的结构来：

```
class User(object):
    def __init__(self, id, name):
        self.id = id
        self.name = name

[
    User('1', 'Michael'),
    User('2', 'Bob'),
    User('3', 'Adam')
]
```

这就是传说中的ORM技术：Object-Relational Mapping，把关系数据库的表结构映射到对象上。

在Python中，最有名的ORM框架是SQLAlchemy。

## 19. Web开发

### 19.1. HTTP协议简介

在Web应用中，服务器把网页传给浏览器，实际上就是把网页的HTML代码发送给浏览器，让浏览器显示出来。而浏览器和服务器的传输协议是HTTP，所以：

- HTML是一种用来定义网页的文本，会HTML，就可以编写网页；
- HTTP是在网络上传输HTML的协议，用于浏览器和服务器的通信。

浏览器并不靠URL来判断响应的内容，所以，即使URL是<http://example.com/abc.jpg>，它也不一定是图片。

#### 19.1.1. HTTP请求

步骤1：浏览器首先向服务器发送HTTP请求，请求包括：

方法：GET还是POST，GET仅请求资源，POST会附带用户数据；

路径：/full/url/path；

域名：由Host头指定：Host: [www.sina.com.cn](http://www.sina.com.cn)

以及其他相关的Header；

如果是POST，那么请求还包括一个Body，包含用户数据。

步骤2：服务器向浏览器返回HTTP响应，响应包括：

响应代码：200表示成功，3xx表示重定向，4xx表示客户端发送的请求有错误，5xx表示服务器端处理时发生了错误；

响应类型：由Content-Type指定；

以及其他相关的Header；

通常服务器的HTTP响应会携带内容，也就是有一个Body，包含响应的内容，网页的HTML源码就在Body中。

步骤3：如果浏览器还需要继续向服务器请求其他资源，比如图片，就再次发出HTTP请求，重复步骤1、2。

Web采用的HTTP协议采用了非常简单的请求-响应模式，从而大大简化了开发。当编写一个页面时，只需要在HTTP请求中把HTML发送出去，不需要考虑如何附带图片、视频等，浏览器如果需要请求图片和视频，它会发送另一个HTTP请求，因此，一个HTTP请求只处理一个资源。

HTTP协议同时具备极强的扩展性，虽然浏览器请求的是<http://www.sina.com.cn/>的首页，但是新浪在HTML中可以链入其他服务器的资源，比如，从而将请求压力分散到各个服务器上，并且，一个站点可以链接到其他站点，无数个站点互相链接起来，就形成了World Wide Web，简称WWW。

#### 19.1.2. HTTP格式

每个HTTP请求和响应都遵循相同的格式，一个HTTP包含Header和Body两部分，其中Body是可选的。

HTTP协议是一种文本协议，所以，它的格式也非常简单。HTTP GET请求的格式：



```
GET /path HTTP/1.1
Header1: Value1
Header2: Value2
Header3: Value3
```

每个Header一行一个，换行符是\r\n。

HTTP POST请求的格式：

```
POST /path HTTP/1.1
Header1: Value1
Header2: Value2
Header3: Value3

body data goes here...
```

当遇到连续两个\r\n时，Header部分结束，后面的数据全部是Body。

HTTP响应的格式：

```
200 OK
Header1: Value1
Header2: Value2
Header3: Value3

body data goes here...
```

HTTP响应如果包含body，也是通过\r\n\r\n来分隔的。请再次注意，Body的数据类型由Content-Type头来确定，如果是网页，Body就是文本，如果是图片，Body就是图片的二进制数据。

当存在**Content-Encoding**时，**Body数据是被压缩的**，最常见的压缩方式是gzip，所以，看到Content-Encoding: gzip时，需要将Body数据先解压缩，才能得到真正的数据。压缩的目的在于减少Body的大小，加快网络传输。

## 19.2. HTML简介

```
<html>
<head>
  <title>Hello</title>
</head>
<body>
  <h1>Hello, world!</h1>
</body>
</html>
```

可以用文本编辑器编写HTML，然后保存为hello.html，双击或者把文件拖到浏览器中，就可以看到效果：

□

HTML文档就是一系列的Tag组成，最外层的Tag是。规范的HTML也包含...和...（注意不要和HTTP的Header、Body搞混了），由于HTML是富文档模型，所以，还有一系列的Tag用来表示链接、图片、表格、表单等等。

### 19.2.1. CSS简介

CSS是Cascading Style Sheets（层叠样式表）的简称，CSS用来控制HTML里的所有元素如何展现，比如，给标题元素<h1>加一个样式，变成48号字体，灰色，带阴影：

```
<html>
<head>
  <title>Hello</title>
  <style>
    h1 {
      color: #333333;
      font-size: 48px;
      text-shadow: 3px 3px 3px #666666;
    }
  </style>
</head>
<body>
  <h1>Hello, world!</h1>
</body>
</html>
```

效果如下：

□

### 19.2.2. JavaScript格式

JavaScript虽然名称有个Java，但它和Java真的一点关系没有。JavaScript是为了让HTML具有交互性而作为脚本语言添加的，JavaScript既可以内嵌到HTML中，也可以从外部链接到HTML中。如果我们希望当用户点击标题时把标题变成红色，就必须通过JavaScript来实现：

```
<html>
<head>
  <title>Hello</title>
  <style>
    h1 {
      color: #333333;
      font-size: 48px;
      text-shadow: 3px 3px 3px #666666;
    }
  </style>
  <script>
    function change() {
      document.getElementsByTagName('h1')[0].style.color = '#ff0000';
    }
  </script>
</head>
<body>
  <h1 onclick="change()">Hello, world!</h1>
</body>
</html>
```

点击标题后效果如下：

□

## 19.3. WSGI接口

一个Web应用的本质就是：

1. 浏览器发送一个HTTP请求；
2. 服务器收到请求，生成一个HTML文档；
3. 服务器把HTML文档作为HTTP响应的Body发送给浏览器；
4. 浏览器收到HTTP响应，从HTTP Body取出HTML文档并显示。

动态生成HTML，底层代码由专门的服务器软件实现，需要一个统一的接口，这个接口就是WSGI：Web Server Gateway Interface。

WSGI接口定义非常简单，它只要求Web开发者实现一个函数，就可以响应HTTP请求。我们来看一个最简单的Web版本的“Hello, web!”：

```
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return [b'<h1>Hello, web!</h1>']
```

上面的application()函数就是符合WSGI标准的一个HTTP处理函数，它接收两个参数：

environ：一个包含所有HTTP请求信息的dict对象；

start\_response：一个发送HTTP响应的函数。

在application()函数中，调用：

```
start_response('200 OK', [('Content-Type', 'text/html')])
```

就发送了HTTP响应的Header，注意Header只能发送一次，也就是只能调用一次start\_response()函数。start\_response()函数接收两个参数，一个是HTTP响应码，一个是一组list表示的HTTP Header，每个Header用一个包含两个str的tuple表示。

通常情况下，都应该把Content-Type头发送给浏览器。其他很多常用的HTTP Header也应该发送。

然后，函数的返回值b'<h1>Hello, web!</h1>'将作为HTTP响应的Body发送给浏览器。

有了WSGI，如何从environ这个dict对象拿到HTTP请求信息，然后构造HTML，通过start\_response()发送Header，最后返回Body。

整个application()函数本身没有涉及到任何解析HTTP的部分，也就是说，底层代码不需要自己编写，只需要在更高层次上考虑如何响应请求。

application()函数必须由WSGI服务器来调用。有很多符合WSGI规范的服务器。

### 19.3.1. 运行WSGI服务

```
# hello.py

def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return [b'<h1>Hello, web!</h1>']
```

server.py，负责启动WSGI服务器，加载application()函数：

```
# server.py
# 从wsgiref模块导入：
from wsgiref.simple_server import make_server
# 导入我们自己编写的application函数：
from hello import application
```

```
# 创建一个服务器，IP地址为空，端口是8000，处理函数是application：
httpd = make_server('', 8000, application)
print('Serving HTTP on port 8000...')
# 开始监听HTTP请求：
httpd.serve_forever()
```

在命令行输入python server.py来启动WSGI服务器：

□

□

在命令行可以看到wsgiref打印的log信息：

□

从environ里读取PATH\_INFO，这样可以显示更加动态的内容：

```
# hello.py

def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    body = '<h1>Hello, %s!</h1>' % (environ['PATH_INFO'][1:] or 'web')
    return [body.encode('utf-8')]
```

无论多么复杂的Web应用程序，入口都是一个WSGI处理函数。HTTP请求的所有输入信息都可以通过environ获得，HTTP响应的输出都可以通过start\_response()加上函数返回值作为Body。

需要在WSGI之上再抽象出Web框架，进一步简化Web开发。

## 19.4. 使用Web框架

Flask通过Python的装饰器在内部自动地把URL和函数给关联起来。

```
from flask import Flask
from flask import request

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def home():
    return '<h1>Home</h1>'

@app.route('/signin', methods=['GET'])
def signin_form():
    return '''<form action="/signin" method="post">
        <p><input name="username"></p>
        <p><input name="password" type="password"></p>
        <p><button type="submit">Sign In</button></p>
    </form>'''

@app.route('/signin', methods=['POST'])
def signin():
    # 需要从request对象读取表单内容：
    if request.form['username']=='admin' and request.form['password']=='password':
        return '<h3>Hello, admin!</h3>'
    return '<h3>Bad username or password.</h3>'

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

运行python app.py，Flask自带的Server在端口5000上监听：

```
$ python app.py
* Running on http://127.0.0.1:5000/
```

打开浏览器，输入首页地址<http://localhost:5000/>：

□

首页显示正确！

再在浏览器地址栏输入<http://localhost:5000/signin>，会显示登录表单：

□

输入预设的用户名admin和口令password，登录成功：

□

输入其他错误的用户名和口令，登录失败：

□

实际的Web App应该拿到用户名和口令后，去数据库查询再比对，来判断用户是否能登录成功。

在编写URL处理函数时，除了配置URL外，从HTTP请求拿到用户数据也是非常重要的。Web框架都提供了自己的API来实现这些功能。Flask通过`request.form['name']`来获取表单的内容。

## 19.5. 使用模板

预先准备一个HTML文档，这个HTML文档不是普通的HTML，而是嵌入了一些变量和指令，然后，根据传入的数据，替换后，得到最终的HTML，发送给用户：

□

MVC：Model-View-Controller，中文名“模型-视图-控制器”。

```
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def home():
    return render_template('home.html')

@app.route('/signin', methods=['GET'])
def signin_form():
    return render_template('form.html')

@app.route('/signin', methods=['POST'])
def signin():
    username = request.form['username']
    password = request.form['password']
    if username=='admin' and password=='password':
        return render_template('signin-ok.html', username=username)
    return render_template('form.html', message='Bad username or password', username=username)

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

Flask通过`render_template()`函数来实现模板的渲染。和Web框架类似，Python的模板也有很多种。Flask默认支持的模板是jinja2:

home.html

用来显示首页的模板：

```
<html>
<head>
  <title>Home</title>
</head>
<body>
  <h1 style="font-style:italic">Home</h1>
</body>
</html>
```

form.html

用来显示登录表单的模板：

```
<html>
<head>
  <title>Please Sign In</title>
</head>
<body>
  {% if message %}
  <p style="color:red">{{ message }}</p>
  {% endif %}
  <form action="/signin" method="post">
    <legend>Please sign in:</legend>
    <p><input name="username" placeholder="Username" value="{{ username }}"></p>
    <p><input name="password" placeholder="Password" type="password"></p>
    <p><button type="submit">Sign In</button></p>
  </form>
</body>
</html>
```

signin-ok.html

登录成功的模板：

```
<html>
<head>
  <title>Welcome, {{ username }}</title>
</head>
<body>
  <p>Welcome, {{ username }}!</p>
```

```
</body>
</html>
```

把模板放到正确的templates目录下，templates和app.py在同级目录下：

□

启动python app.py

□

循环、条件判断等指令语句：在Jinja2中，用{% ... %}表示指令。

比如循环输出页码：

```
{% for i in page_list %}
    <a href="/page/{ i }">{{ i }}</a>
{% endfor %}
```

## 20. 异步IO

多线程和多进程的模型虽然解决了并发问题，但是系统不能无上限地增加线程。由于系统切换线程的开销也很大，所以，一旦线程数量过多，CPU的时间就花在线程切换上了，真正运行代码的时间就少了，结果导致性能严重下降。

问题是CPU高速执行能力和IO设备的龟速严重不匹配，多线程和多进程只是解决这一问题的一种方法。

另一种解决IO问题的方法是异步IO。当代码需要执行一个耗时的IO操作时，它只发出IO指令，并不等待IO结果，然后就去执行其他代码了。一段时间后，当IO返回结果时，再通知CPU进行处理。

按普通顺序写出的代码实际上是没法完成异步IO的：

```
do_some_code()
f = open('/path/to/file', 'r')
r = f.read() # <== 线程停在此处等待IO操作结果
# IO操作完成后线程才能继续执行：
do_some_code(r)
```

同步IO模型的代码是无法实现异步IO模型的。

异步IO模型需要一个消息循环，在消息循环中，主线程不断地重复“读取消息-处理消息”这一过程：

```
loop = get_event_loop()
while True:
    event = loop.get_event()
    process_event(event)
```

当遇到IO操作时，代码只负责发出IO请求，不等待IO结果，然后直接结束本轮消息处理，进入下一轮消息处理过程。当IO操作完成后，将收到一条“IO完成”的消息，处理该消息时就可以直接获取IO操作结果。

在“发出IO请求”到收到“IO完成”的这段时间里，同步IO模型下，主线程只能挂起，但异步IO模型下，主线程并没有休息，而是在消息循环中继续处理其他消息。这样，在异步IO模型下，一个线程就可以同时处理多个IO请求，并且没有切换线程的操作。对于大多数IO密集型的应用程序，使用异步IO将大大提升系统的多任务处理能力。

### 20.1. 协程

协程，又称微线程，纤程。英文名Coroutine。

子程序，或者称为函数，在所有语言中都是层级调用，比如A调用B，B在执行过程中又调用了C，C执行完毕返回，B执行完毕返回，最后是A执行完毕。

所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。

子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

注意，在一个子程序中中断，去执行其他子程序，不是函数调用，有点类似CPU的中断。比如子程序A、B：

```
def A():
    print('1')
    print('2')
    print('3')

def B():
    print('x')
    print('y')
    print('z')
```

假设由协程执行，在执行A的过程中，可以随时中断，去执行B，B也可能在执行过程中中断再去执行A，结果可能是：

```
1
2
x
```

```
y
3
z
```

但是在A中是没有调用B的，所以协程的调用比函数调用理解起来要难一些。

协程的特点在于是一个线程执行。

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

利用多核CPU最简单的方法是多进程+协程。

Python对协程的支持是通过generator实现的。

在generator中，我们不但可以通过for循环来迭代，还可以不断调用next()函数获取由yield语句返回的下一个值。

Python的yield不但可以返回一个值，它还可以接收调用者发出的参数。

传统的生产者-消费者模型是一个线程写消息，一个线程取消息，通过锁机制控制队列和等待，但一不小心就可能死锁。

如果改用协程，生产者生产消息后，直接通过yield跳转到消费者开始执行，待消费者执行完毕后，切换回生产者继续生产，效率极高：

```
def consumer():
    r = ''
    while True:
        n = yield r
        if not n:
            return
        print('[CONSUMER] Consuming %s...' % n)
        r = '200 OK'

def produce(c):
    c.send(None)
    n = 0
    while n < 5:
        n = n + 1
        print('[PRODUCER] Producing %s...' % n)
        r = c.send(n)
        print('[PRODUCER] Consumer return: %s' % r)
    c.close()

c = consumer()
produce(c)
```

```
[PRODUCER] Producing 1...
[CONSUMER] Consuming 1...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 2...
[CONSUMER] Consuming 2...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 3...
[CONSUMER] Consuming 3...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 4...
[CONSUMER] Consuming 4...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 5...
[CONSUMER] Consuming 5...
[PRODUCER] Consumer return: 200 OK
```

注意到consumer函数是一个generator，把一个consumer传入produce后：

首先调用c.send(None)启动生成器；

然后，一旦生产了东西，通过c.send(n)切换到consumer执行；

consumer通过yield拿到消息，处理，又通过yield把结果传回；

produce拿到consumer处理的结果，继续生产下一条消息；

produce决定不生产了，通过c.close()关闭consumer，整个过程结束。

整个流程无锁，由一个线程执行，produce和consumer协作完成任务，所以称为“协程”，而非线程的抢占式多任务。

最后套用Donald Knuth的一句话总结协程的特点：

“子程序就是协程的一种特例。”

## 20.2. asyncio

asyncio的编程模型就是一个消息循环。从asyncio模块中直接获取一个EventLoop的引用，然后把需要执行的协程扔到EventLoop中执行，就实现了异步IO。

用asyncio实现Hello world代码如下：

```
import asyncio

@asyncio.coroutine
def hello():
    print("Hello world!")
    # 异步调用asyncio.sleep(1):
    r = yield from asyncio.sleep(1)
    print("Hello again!")

# 获取EventLoop:
loop = asyncio.get_event_loop()
# 执行coroutine
loop.run_until_complete(hello())
loop.close()
```

@asyncio.coroutine把一个generator标记为coroutine类型，然后，把这个coroutine扔到EventLoop中执行。

hello()会首先打印出Hello world!，然后，yield from语法可以方便地调用另一个generator。由于asyncio.sleep()也是一个coroutine，所以线程不会等待asyncio.sleep()，而是直接中断并执行下一个消息循环。当asyncio.sleep()返回时，线程就可以从yield from拿到返回值（此处是None），然后接着执行下一行语句。

把asyncio.sleep(1)看成是一个耗时1秒的IO操作，在此期间，主线程并未等待，而是去执行EventLoop中其他可以执行的coroutine了，因此可以实现并发执行。

用Task封装两个coroutine：

```
import threading
import asyncio

@asyncio.coroutine
def hello():
    print('Hello world! (%s)' % threading.currentThread())
    yield from asyncio.sleep(1)
    print('Hello again! (%s)' % threading.currentThread())

loop = asyncio.get_event_loop()
tasks = [hello(), hello()]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

```
Hello world! (<_MainThread(MainThread, started 140735195337472)>)
Hello world! (<_MainThread(MainThread, started 140735195337472)>)
(暂停约1秒)
Hello again! (<_MainThread(MainThread, started 140735195337472)>)
Hello again! (<_MainThread(MainThread, started 140735195337472)>)
```

如果把asyncio.sleep()换成真正的IO操作，则多个coroutine就可以由一个线程并发执行。

用asyncio的异步网络连接来获取sina、sohu和163的网站首页：

```
import asyncio

@asyncio.coroutine
def wget(host):
    print('wget %s...' % host)
    connect = asyncio.open_connection(host, 80)
    reader, writer = yield from connect
    header = 'GET / HTTP/1.0\r\nHost: %s\r\n\r\n' % host
    writer.write(header.encode('utf-8'))
    yield from writer.drain()
    while True:
        line = yield from reader.readline()
        if line == b'\r\n':
            break
        print('%s header > %s' % (host, line.decode('utf-8').rstrip()))
    # Ignore the body, close the socket
    writer.close()

loop = asyncio.get_event_loop()
tasks = [wget(host) for host in ['www.sina.com.cn', 'www.sohu.com', 'www.163.com']]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

```
wget www.sohu.com...
wget www.sina.com.cn...
wget www.163.com...
(等待一段时间)
(打印出sohu的header)
www.sohu.com header > HTTP/1.1 200 OK
```

```
www.sohu.com header > Content-Type: text/html
...
(打印出sina的header)
www.sina.com.cn header > HTTP/1.1 200 OK
www.sina.com.cn header > Date: Wed, 20 May 2015 04:56:33 GMT
...
(打印出163的header)
www.163.com header > HTTP/1.0 302 Moved Temporarily
www.163.com header > Server: Cdn Cache Server V2.0
```

可见3个连接由一个线程通过coroutine并发完成。

asyncio提供了完善的异步IO支持；

异步操作需要在coroutine中通过yield from完成；

多个coroutine可以封装成一组Task然后并发执行。

## 20.3. asyncio/await

用asyncio提供的@asyncio.coroutine可以把一个generator标记为coroutine类型，然后在coroutine内部用yield from调用另一个coroutine实现异步操作。

从Python 3.5开始引入了新的语法async和await，可以让coroutine的代码更简洁易读。

async和await是针对coroutine的新语法，要使用新的语法，只需要做两步简单的替换：

把@asyncio.coroutine替换为async；

把yield from替换为await。

```
@asyncio.coroutine
def hello():
    print("Hello world!")
    r = yield from asyncio.sleep(1)
    print("Hello again!")
```

用新语法重新编写如下：

```
async def hello():
    print("Hello world!")
    r = await asyncio.sleep(1)
    print("Hello again!")
```

## 20.4. aiohttp

asyncio可以实现单线程并发IO操作。如果仅用在客户端，发挥的威力不大。如果把asyncio用在服务器端，例如Web服务器，由于HTTP连接就是IO操作，因此可以用单线程+coroutine实现多用户的高并发支持。

asyncio实现了TCP、UDP、SSL等协议，aiohttp则是基于asyncio实现的HTTP框架。

编写一个HTTP服务器，分别处理以下URL：

1. `/` - 首页返回**b'<h1>Index</h1>'；**
2. `/hello/{name}` - 根据URL参数返回文本**hello, {name}!**。

```
import asyncio

from aiohttp import web

async def index(request):
    await asyncio.sleep(0.5)
    return web.Response(body=b'<h1>Index</h1>')

async def hello(request):
    await asyncio.sleep(0.5)
    text = '<h1>hello, {name}!</h1>' % request.match_info['name']
    return web.Response(body=text.encode('utf-8'))

async def init(loop):
    app = web.Application(loop=loop)
    app.router.add_route('GET', '/', index)
    app.router.add_route('GET', '/hello/{name}', hello)
    srv = await loop.create_server(app.make_handler(), '127.0.0.1', 8000)
    print('Server started at http://127.0.0.1:8000...')
    return srv

loop = asyncio.get_event_loop()
loop.run_until_complete(init(loop))
loop.run_forever()
```

注意aiohttp的初始化函数init()也是一个coroutine，loop.create\_server()则利用asyncio创建TCP服务。