# 类型与语句结构

### 变量本身类型不固定

变量无类型,值有类型

a = 123 # a是整数

print(a)

a = 'asg' # a变为字符串

print(a)

### 以缩进方式来组织

if name>'a':

print('大于a')

else :

print('小于a')

除法

10/3 🡪3 10.0/3.0🡪3.3333333333333335

10//3 🡪 3 10.0//3.0🡪3.0 :地板除

### if

语句格式

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>

### for

# for x in ...循环就是把每个元素代入变量x，然后执行缩进块的语句

#Python提供一个range()函数，可以生成一个整数序列，再通过list()函数可以转换为list。

list(range(5)) #[0, 1, 2, 3, 4]

### while

sum = 0

n = 0

while n < 10:

sum = sum + n

n = n + 1

print(sum)

break 与 continue

### input

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：

birth = input('birth: ')

if birth < 2000:

print('00前')

else:

print('00后')

name = input('请输入名字:')

print('输入的名字是:\'', name)

### 字符串的表示

'xx' "xxx" "x'xx" "x\'x\"xx" 'x\'x\"xx'

避免转义

r'xxx\\x'

### 多行的表示

'''...'''的格式表示多行

print(r'''如果字符串里面有很多字符都需要转义，

就需要加很多\，

为了简化，

Python还允许用r'',

表示''内部的字符串默认不转义，

可以自己试试：''')

### 对a = 'ABC'的解释

#Python解释器干了两件事情：

#1.在内存中创建了一个'ABC'的字符串；

#2.在内存中创建了一个名为a的变量，并把它指向'abc'。

a = 'ABC'

b = a

a = 'XYZ'

print(b) #输出的是'ABC'还是'XYZ'

### 字符编码encode,decode

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'

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

'ABC'.encode('ascii')

'中文'.encode('utf-8')

要把bytes变为str，就需要用decode()方法：

b'ABC'.decode('ascii')

要计算str包含多少个字符，可以用len()函数：(注意是计算字符数)

len('ABC') #3

len('中文') #2

### list

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

len(classmates)

classmates[0]

classmates.append('Adam')

classmates.insert(1, 'Jack')

classmates.pop()

classmates[1] = 'Sarah'

p = ['asp', 'php']

classmates.append(p)

### tuple

元组tuple一旦初始化就不能修改

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

### 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。

可以通过[]list来创建set---- s = set([1, 2, 3, 2]) 重复元素在set中自动被过滤：

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

通过remove(key)方法可以删除元素：

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}

### 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。

dict可以用在需要高速查找的很多地方，在Python代码中几乎无处不在，正确使用dict非常重要，

需要牢记的第一条就是dict的key必须是不可变对象。

d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}

d['Michael'] #95

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

'Tracy' in d ----True

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') 或 d.get('Thomas', -1)

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

d.pop('Bob')

### list,tuple,set,dict和表示

[x, y, z, ...]--list是可变对象， (x, y, z,.....)--tuple是不可变对象，

set()--set，{x:vx, y:vy, z:vz, ...}--dict，

### 不变对象

在Python中，字符串、整数等都是不可变的，因此，可以放心地作为key。而list是可变的，就不能作为key：

对于不变对象来说，调用对象自身的任意方法，也不会改变该对象自身的内容。

相反，这些方法会创建新的对象并返回，这样，就保证了不可变对象本身永远是不可变的。

## 回顾

python 变量是动态类型，变量类型因值而异，变量名区分大小写。

单行字符串 '...'，"..."两种方式表示

多行字符串'''...'''表示

转义字符与C基本相同

r'...'不使用转义表示. '\r':表示一个换行符, r'\n':表示两个字符—'\'和'n'

布尔值：True,False----and,or,not运算

空值：None

python变量赋值的理解：

a = 'ABC' #@1

b = a

对于a = 'ABC' ：python解释器做了两件事

1.在内存中创建了一个'ABC'的字符串；

2. 在内存中创建了一个名为a的变量，并把它指向'ABC'。

执行b = a，解释器创建了变量b，并把b指向a指向的字符串'ABC'

字符串与编码

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

很多网页上的源码中有<meta charset="utf-8" />的信息，表示该网页正是用的UTF-8编码

ord(str) chr(xxx)

b'str' str.encode() b'str'.decode('ascii') len(str)

格式化:基本同C. 如'%d' % 3---第二个%用于分隔格式化串与实际值

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

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

len(classmates)

classmates[i]

classmates.append('Adam')

classmates.insert(i, 'Jack') #在原list的第i个位置插入,原来的元素后移

classmates.pop(i) #无参时删除末尾元素

classmates[i] = 'Sarah' #修改

list里面的元素的数据类型也可以不同

p = ['asp', 'php']

classmates[i] = p

list类型与C++的List，可以通过下标访问，当其元素为

元组：tuple -- 和list非常类似，但是tuple一旦初始化就不能修改

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

classmates[0] = 'abc' #错误

t = ('a', 'b', ['A', 'B'])

t[2][0] = 'X' #正确，说明：tuple不可变类型，这里t的第三个元素是list类型，我们改变的是

#t[2][0]所指向的元素， t[2]并没有改变（即tuple的list元素并没有改变）

对tuple元素不能修改的理解是--指向不变。

对python元素的理解是----C/C++中的高级指针，变量赋值是改变指针的指向

字典：dict ---使用键-值（key-value）存储，具有极快的查找速度(类比C++的map)

牢记:dict的key必须是不可变对象。

一个key只能对应一个value

d = {key:value, 'Michael': 95, 'Bob': 75, 'Tracy': 85}

d['Michael']

d['Adam'] = 67 #会在dict中增加一个元素

如果key不存在，dict就会报错：

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：

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

'Thomas' in d

d.get(key) 或 d.get(key, value)

pop(key) #删除

集合:set--一组key的集合(类比C++的set)

在set中，没有重复的key

s = set([key, 1, 2, 3])

add(key)

remove(key)

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

对于不变对象来说，调用对象自身的任意方法，也不会改变该对象自身的内容。

这些方法会创建新的对象并返回，这样就保证了不可变对象本身永远是不可变的。

# 函数

### help(abs)查看abs函数的帮助信息

### 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，比如int()函数可以把其他数据类型转换为整数

### 函数名

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

dict1 = {'n1':255, 'n2':1000}

for key in dict1:

print(key+"="+hex(dict1[key]))

## 定义函数

在Python中，定义一个函数要使用def语句，依次写出函数名、括号、括号中的参数和冒号:

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，函数执行完毕后也会返回结果，只是结果为None。

return <==> return None

我们以自定义一个求绝对值的my\_abs函数为例：

def my\_abs(x):

if x >= 0:

return x

else:

return -x

请注意，函数体内部的语句在执行时，一旦执行到return时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有return语句，函数执行完毕后也会返回结果，只是结果为None。

return None可以简写为return。

在Python交互环境中定义函数时，注意Python会出现...的提示。函数定义结束后需要按两次回车重新回到>>>提示符下：

如果你已经把my\_abs()的函数定义保存为abstest.py文件了，那么，可以在该文件的当前目录下启动Python解释器，

用from abstest import my\_abs来导入my\_abs()函数，注意abstest是文件名（不含.py扩展名）：

### 空函数

如果想定义一个什么事也不做的空函数，可以用pass语句：

def nop():

pass

实际上pass可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个pass，让代码能运行起来。

pass还可以用在其他语句里，比如：

if age >= 18:

pass

缺少了pass，代码运行就会有语法错误。

### 数据类型检查isinstance

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)): #如果x不是int或float则抛出类型错误

raise TypeError('bad operand type')

if x >= 0:

return x

else:

return -x

### 返回多个值

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

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

nx = x + step \* math.cos(angle)

ny = y - step \* math.sin(angle)

return nx, ny

但其实这只是一种假象，Python函数返回的仍然是单一值：以tuple元组(nx, ny)的方式返回

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

### 默认参数 parameter=val

必选参数在前，默认参数在后

def power(x, n=2):

pass

python有多个默认参数时可以不提供最后的一个或多个默认参数，

也可以不提供中间部分的默认参数，这时需要把后面提供的默认参数的名称写上（这与C++不同）.

如： 有函数 def func(x, y='A', z=0):

func(1, 'B') ---不提供默认参数z

和 func(1, z=1) --不提供参数y(这时参数y之后的参数必须提供参数名)

**默认参数必须指向不变对象！！！**----否则会出现意想不到的情况（不是语法错误）：如下

def add\_end(L=[]):

L.append('END')

return L

add\_end() #['END']

add\_end() #['END', 'END']

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

Python函数在定义的时候，默认参数L的值就被计算出来了，即[]，因为默认参数L也是一个变量，

它指向对象[]，每次调用该函数，如果改变了L的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的[]了。

### 可变参数 \*parameter --内部组装为一个tuple

函数的参数改为可变参数：

def calc(\*numbers):

sum = 0

for n in numbers:

sum = sum + n \* n

return sum

调用例子

nums = [1, 2, 3]

calc(nums[0], nums[1], nums[2]) #在函数内部，参数numbers接收到的是一个tuple

Python允许你在list或tuple前面加一个\*号，把list或tuple的元素变成可变参数传进去：

nums = [1, 2, 3]

calc(\*nums) #在函数内部，参数numbers接收到的是一个tuple

一个一个传递可变参数时,在函数内部组装成tuple; 直接传递list或tuple到函数时(外部组装),需要加一个’\*’号

### 关键字参数 \*\*parameter --内部组装为一个dict

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

def person(name, age, \*\*kw):

print('name:', name, 'age:', age, 'other:', kw)

return

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

调用例子

person('Bob', 35, city='Beijing') #name: Bob age: 35 other: {'city': 'Beijing'}

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

extra = {'city': 'Beijing', 'job': 'Engineer'}

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

当然，上面复杂的调用可以用简化的写法：

extra = {'city': 'Beijing', 'job': 'Engineer'}

person('Jack', 24, \*\*extra) #name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

\*\*extra表示把extra这个dict的所有key-value用关键字参数传入到函数的\*\*kw参数，kw将获得一个dict，

注意kw获得的dict是extra的一份拷贝，对kw的改动不会影响到函数外的extra

对于关键字参数，函数的调用者可以传入任意不受限制的关键字参数。至于到底传入了哪些，

就需要在函数内部通过kw检查。

函数的参数:

定义函数的时候，我们把参数的名字和位置确定下来，函数的接口定义就完成了。对于函数的调用者来说，只需要知道如何传递正确的参数，以及函数将返回什么样的值就够了，函数内部的复杂逻辑被封装起来，调用者无需了解。

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

### 位置参数

我们先写一个计算x^2的函数：

def power(x):

return x \* x

对于power(x)函数，参数x就是一个位置参数。

当我们调用power函数时，必须传入有且仅有的一个参数x：

>>> power(5)

25

>>> power(15)

225

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

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

def power(x, n):

s = 1

while n > 0:

n = n - 1

s = s \* x

return s

对于这个修改后的power(x, n)函数，可以计算任意n次方：

>>> power(5, 2)

25

>>> power(5, 3)

125

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

### 默认参数2

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

>>> power(5)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: power() missing 1 required positional argument: 'n'

Python的错误信息很明确：调用函数power()缺少了一个位置参数n。

这个时候，默认参数就排上用场了。由于我们经常计算x2，所以，完全可以把第二个参数n的默认值设定为2：

def power(x, n=2):

s = 1

while n > 0:

n = n - 1

s = s \* x

return s

这样，当我们调用power(5)时，相当于调用power(5, 2)：

>>> power(5)

25

>>> power(5, 2)

25

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

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

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

二是如何设置默认参数。

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

使用默认参数有什么好处？最大的好处是能降低调用函数的难度。

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

def enroll(name, gender):

print('name:', name)

print('gender:', gender)

这样，调用enroll()函数只需要传入两个参数：

>>> enroll('Sarah', 'F')

name: Sarah

gender: F

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。

我们可以把年龄和城市设为默认参数：

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

print('name:', name)

print('gender:', gender)

print('age:', age)

print('city:', city)

这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：

>>> enroll('Sarah', 'F')

name: Sarah

gender: F

age: 6

city: Beijing

只有与默认参数不符的学生才需要提供额外的信息：

enroll('Bob', 'M', 7)

enroll('Adam', 'M', city='Tianjin')

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

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

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

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个list，添加一个END再返回：

def add\_end(L=[]):

L.append('END')

return L

当你正常调用时，结果似乎不错：

>>> add\_end([1, 2, 3])

[1, 2, 3, 'END']

>>> add\_end(['x', 'y', 'z'])

['x', 'y', 'z', 'END']

当你使用默认参数调用时，一开始结果也是对的：

>>> add\_end()

['END']

但是，再次调用add\_end()时，结果就不对了：

>>> add\_end()

['END', 'END']

>>> add\_end()

['END', 'END', 'END']

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

原因解释如下：

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

所以，定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用None这个不变对象来实现：

def add\_end(L=None):

if L is None:

L = []

L.append('END')

return L

现在，无论调用多少次，都不会有问题：

>>> add\_end()

['END']

>>> add\_end()

['END']

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

### 可变参数

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

我们以数学题为例子，给定一组数字a，b，c……，请计算a2 + b2 + c2 + ……。

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

def calc(numbers):

sum = 0

for n in numbers:

sum = sum + n \* n

return sum

但是调用的时候，需要先组装出一个list或tuple：

>>> calc([1, 2, 3])

14

>>> calc((1, 3, 5, 7))

84

如果利用可变参数，调用函数的方式可以简化成这样：

>>> calc(1, 2, 3)

14

>>> calc(1, 3, 5, 7)

84

所以，我们把函数的参数改为可变参数：

def calc(\*numbers):

sum = 0

for n in numbers:

sum = sum + n \* n

return sum

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

>>> calc(1, 2)

5

>>> calc()

0

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

>>> nums = [1, 2, 3]

>>> calc(nums[0], nums[1], nums[2])

14

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

>>> nums = [1, 2, 3]

>>> calc(\*nums)

14

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

关键字参数：调用时需要key=xxx的形式提供参数

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

def person(name, age, \*\*kw):

print('name:', name, 'age:', age, 'other:', kw)

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

>>> person('Michael', 30)

name: Michael age: 30 other: {}

也可以传入任意个数的关键字参数：

>>> person('Bob', 35, city='Beijing')

name: Bob age: 35 other: {'city': 'Beijing'}

>>> person('Adam', 45, gender='M', job='Engineer')

name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}

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

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

>>> extra = {'city': 'Beijing', 'job': 'Engineer'}

>>> person('Jack', 24, city=extra['city'], job=extra['job'])

name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

当然，上面复杂的调用可以用简化的写法：

>>> extra = {'city': 'Beijing', 'job': 'Engineer'}

>>> person('Jack', 24, \*\*extra)

name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

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

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

仍以person()函数为例，我们希望检查是否有city和job参数：

def person(name, age, \*\*kw):

if 'city' in kw:

# 有city参数

pass

if 'job' in kw:

# 有job参数

pass

print('name:', name, 'age:', age, 'other:', kw)

return

命名关键字参数：调用时需要key=xxx的形式提供参数

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): #调用时需要city=xxx, job=xxx的形式提供命名参数

print(name, age, city, job)

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

调用方式如下：

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

Jack 24 Beijing Engineer

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)

### 默认参数

def person(name, age=10, sex='nam'):

print(name, age, sex)

调用

>>>person('Jack', 24) #不提供靠右的默认参数

Jack 24 nam

>>>person('Jack', sex='nv') #当提供靠右的默认参数而不提供靠左的默认参数时,需要指定靠右的默认参数名

Jack 10 nv

不定(可变)参数：

def person(name, age, \*args):

print(name, age, args)

调用

>>>person('Jack', 24, 'Beijing', 'Engineer')

Jack 24 ('Beijing', 'Engineer')--不定参数在函数内部自动组装为一个元组tuple

### 关键字参数

def person(name, age, \*\*args):

print(name, age, args)

调用

>>>person('Jack', 24, city='Beijing', job='Engineer')---关键字参数以key=xxx的形式调用

Jack 24 {'job': 'Engineer', 'city': 'Beijing'}---关键字参数在函数内部自动组装为一个字典dict

### 不定参数与关键字参数的区别

1.不定参数在定义时有一个\*，关键字参数在定义时有两个\*

2.不定参数在函数内部自动组装为一个元组tuple，关键字参数在函数内部自动组装为一个字典dict

3.调用方式上：不定参数直接提供参数，关键字参数以key=value的方式提供参数

### 命名关键字参数

def person(name, age, \*, city, job): #以\*分隔普通参数与命名关键字参数

print(name, age, city, job)

调用

>>>person('Jack', 24, city='Beijing', job='Engineer') #命名关键字参数以key=xxx的形式调用,命名关键字参数不可省略

Jack 24 Beijing Engineer

### 不定参数与命名参数

def person(name, age, \*args, city, job): #args是不定参数; city, job是可变参数

print(name, age, args, city, job)

调用

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

Jack 24 () Beijing Engineer ---可变参数为空，对应一个空元组

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

Jack 24 ('xxx', 'yyy') Beijing Engineer

'''

'''

汉诺塔

请编写move(n, a, b, c)函数，它接收参数n，表示3个柱子A、B、C中第1个柱子A的盘子数量，

然后打印出把所有盘子从A借助B移动到C的方法

汉诺塔问题有三根柱子，我给它们分别命名为起始柱src，临时柱tmp，目的柱dst

盘子一共分两种情况：

1.只有1个盘子

这种情况下，直接从起始柱src 移动到 目的柱dst ,完成任务。

2.有1个以上的盘子

假如有n个盘子在起始柱，

首先把第n个盘子上方的n-1个盘子搬到临时柱。

然后把第n个盘子从起始柱移动到目的柱

最后把n-1个盘子从临时柱搬到目的柱 任务完成

'''

def move(n, a, b, c):

if n==1:

print("%c-->%c" %(a, c))

else:

move(n-1, a, c, b)

move(1, a, b, c)

move(n-1, b, a, c)

调用

print(move(3, 'A', 'B', 'C'))

### 尾递归

解决递归调用栈溢出的方法是通过尾递归优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

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。

def fact(n):

return fact\_iter(n, 1)

def fact\_iter(num, product):

if num==1:

return product

else:

return fact\_iter(num-1, num\*product)

调用

print(fact(100))

# 高级特性

### 切片

range(n)---产生[0, n)的序列

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

L[开始索引 ：结束索引-不包括它：步长]

>>>L[0:3] #切片，取L的第[0,3)个元素（第0，第1，第2 个元素）

>>>L[1:3] #切片，取L的第[1,3)个元素 （第1，第2 个元素）

>>>L[:3] #切片，取L的前3个元素

>>>L[-3:] #切片，取L的后3个元素 <==>L[-3:0]

>>>L[::] #取L的第[0,len() )个元素 <==> L[0:len(L)]

>>>L[::-1] #取L的第[-1,-len()]个元素 <==>相当于将L逆置

切片时还可以指定步长

>>> L = list(range(100)) #0，1，2，3......99

>>> L[2:20:2] #在第[2,20)个元素中每两个取一个 ： 2, 4, 6, 8, 10, 12, 14, 16, 18

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：

>>> t = (0, 1, 2, 3, 4, 5)

>>> t[0:3]

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：

>>> c = 'ABCDEFG'

>>> c[0:3]

自己对比看看，了解一下写了开始索引和结束索引，以及不写索引的区别：

L = [1,2,3,4,5,6,7,8,9]

print(L[::1])

print(L[:8:1])

print(L[-2::1])

print(L[-2::-1])

print(L[-2:0:-1])

print(L[-2:0:-2])

结果分别是：

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

[1, 2, 3, 4, 5, 6, 7, 8]

[8, 9]

[8, 7, 6, 5, 4, 3, 2, 1]

[8, 7, 6, 5, 4, 3, 2]

[8, 6, 4, 2]

### 迭代

Python的for循环不仅可以用在list或tuple上，还可以作用在其他可迭代对象上。

list这种数据类型虽然有下标，但很多其他数据类型是没有下标的，但是，只要是可迭代对象，

无论有无下标，都可以迭代，比如dict就可以迭代：

>>> d = {'a': 1, 'b': 2, 'c': 3}

>>> for key in d:

... print(key)

默认情况下，dict迭代的是key。如果要迭代value，可以用for value in d.values()，

如果要同时迭代key和value，可以用for k, v in d.items()。

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

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

### 列表生成式

生成list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]可以用list(range(1, 11))

生成[1x1, 2x2, 3x3, ..., 10x10]可以[x \* x for x in range(1, 11)]

for循环后面还可以加上if判断，这样我们就可以筛选出仅偶数的平方：

>>> [x \* x for x in range(1, 11) if x % 2 == 0]

还可以使用两层循环，可以生成全排列：

>>> [m + n for m in 'ABC' for n in 'XYZ']

['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']

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

x = [d for d in os.listdir('.')] #os.listdir 可以列出文件和目录

print(x)

### 生成器

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。

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

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

>>> L = [x \* x for x in range(10)]

>>> L

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> g = (x \* x for x in range(10))

>>> g

<generator object <genexpr> at 0x1022ef630>

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

我们可以直接打印出list的每一个元素，但我们怎么打印出generator的每一个元素呢？

可以通过next()函数获得generator的下一个返回值：当生成器没有元素时，next(生成器对象)将抛出StopIteration的错误。

也可以直接使用for循环来迭代：如

>>> g = (x \* x for x in range(10))

>>> for n in g:

... print(n)

注意：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：

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'

注意，赋值语句：

a, b = b, a + b

相当于：

t = (b, a + b) # t是一个tuple

a = t[0]

b = t[1]

但不必显式写出临时变量t就可以赋值。

这里是通过函数的方式实现斐波拉契数列

那么我们如何定义generator来实现斐波拉契数列呢

def fib(max):

n, a, b = 0, 0, 1

while n < max:

yield b

a, b = b, a + b

n = n + 1

return 'done'

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：

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

### 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。

这些可以直接作用于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

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，但我们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。

凡是可作用于next()函数的对象都是Iterator类型，它们表示一个惰性计算的序列；

集合数据类型如list、dict、str等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象。

#####函数式编程

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，

还允许返回一个函数！

# 高阶函数

### 变量可以指向函数

函数与函数调用，以Python内置的求绝对值的函数abs()为例

abs(-10)是函数调用，而abs是函数本身

函数本身也可以赋值给变量，即：变量可以指向函数

>>> f = abs

>>> f(-10)

10

### 函数名也是变量

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

>>> def f(x):

... return x-1

...

>>> abs = f #改变了原abs函数名所指向的函数 ！！！注：实际代码绝对不能这么写

>>> abs(1)

0

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

### 函数作为参数

一个函数接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

def add(x, y, f):

return f(x) + f(y)

### map/reduce

1. 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]

Iterator是惰性序列，因此通过list()函数让它把整个序列都计算出来并返回一个list。

例子：

这个list所有数字转为字符串：

>>> list(map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9]))

['1', '2', '3', '4', '5', '6', '7', '8', '9']

2. reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，reduce必须接收两个参数，

reduce把结果继续和序列的下一个元素做累积计算，其效果就是：

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

把序列[1, 3, 5, 7, 9]变换成整数13579，reduce就可以派上用场：

>>> from functools import reduce

>>> def fn(x, y):

... return x \* 10 + y

...

>>> reduce(fn, [1, 3, 5, 7, 9])

13579

### python函数嵌套定义

from functools import reduce

//str2int--实现str-->int

def str2int(s):

def fn(x, y):

return x \* 10 + y

def char2num(s):

return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}[s]

return reduce(fn, map(char2num, s))

### filter

Python内建的filter()函数用于过滤序列。

和map()类似，filter()也接收一个函数和一个序列。和map()不同的是，

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。---有点像C++标准算法中的谓词

例如，在一个list中，删掉偶数，只保留奇数，可以这么写：

def is\_odd(n):

return n % 2 == 1

list(filter(is\_odd, [1, 2, 4, 5, 6, 9, 10, 15]))

结果: [1, 5, 9, 15]

注意到filter()函数返回的是一个Iterator，也就是一个惰性序列，

所以要强迫filter()完成计算结果，需要用list()函数获得所有结果并返回list。

### sorted

排序算法

sorted()是一个高阶函数，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

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([36, 5, -12, 9, -21], key=abs, reverse=True)

## 返回函数(闭包—返回内部函数的引用)

### 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

问题引出

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

**def** **calc\_sum**(\*args):

ax = 0

**for** n **in** args:

ax = ax + n

**return** ax

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？

问题解决方法：

可以不返回求和的结果，而是返回求和的函数

**def** **lazy\_sum**(\*args):

**def** **sum**():

ax = 0

**for** n **in** args:

ax = ax + n

**return** ax

**return** sum

当我们调用lazy\_sum()时，返回的并不是求和结果，而是求和函数：

>>> f = lazy\_sum(1, 3, 5, 7, 9)

>>> f

<**function** lazy\_sum.<locals>.sum at 0x101c6ed90>

调用函数f时，才真正计算求和的结果：

>>> f()

25

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

注意:当我们调用lazy\_sum()时，每次调用都会返回一个新的函数，即使传入相同的参数：

>>> f1 = lazy\_sum(1, 3, 5, 7, 9)

>>> f2 = lazy\_sum(1, 3, 5, 7, 9)

>>> f1==f2

False

f1()和f2()的调用结果互不影响。

**闭包**

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

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

**def** **count**():

fs = []

**for** i **in** range(1, 4):

**def** **f**():

**return** i\*i #嵌套函数可以使用外层函数中的变量 --- i的值取决就被调用时,外层函数中i的值

fs.append(f) #将函数f加入到列表中---注意:加入的是函数,而不是函数的执行结果

**return** fs

f1, f2, f3 = count()

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了。

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

>>> f1()

9

>>> f2()

9

>>> f3()

9

全部都是9！原因就在于返回的函数引用了外层函数中的循环变量i。当函数执行时,循环已经结束了,此时i的值为3，因此最终结果为9。

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

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

**def** **count**():

**def** **f**(j):

**def** **g**():

**return** j\*j

**return** g

fs = []

**for** i **in** range(1, 4):

fs.append(f(i)) *# f(i)立刻被执行，因此i的当前值被传入f()*

**return** fs

可利用lambda函数缩短代码。

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

### 匿名函数lambda

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

在Python中，对匿名函数提供了有限支持。还是以map()函数为例，计算f(x)=x2时，除了定义一个f(x)的函数外，还可以直接传入匿名函数：

>>> list(map(lambda x: x \* x, [1, 2, 3, 4, 5, 6, 7, 8, 9]))

[1, 4, 9, 16, 25, 36, 49, 64, 81]

通过对比可以看出，匿名函数lambda x: x \* x实际上就是：

**def** **f**(x):

**return** x \* x

关键字lambda表示匿名函数，冒号前面的x表示函数参数。

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

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。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

同样，也可以把匿名函数作为返回值返回，比如：

**def** **build**(x, y):

**return** **lambda**: x \* x + y \* y

## 装饰器

引入：

函数对象有一个\_\_name\_\_属性，可以拿到函数的名字

#定义一个函数now

**def** **now**():

print('2016-11-8')

f = now #函数对象可以被赋值给变量

f() #通过变量调用该函数

2016-11-8

print( now.\_\_name\_\_) #拿到函数名

'now'

print( f.\_\_name\_\_) #拿到变量对应的函数名

'now'

### 装饰器的实现

现在我想要增强now()函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改now()函数的定义，该如何实现呢？

如下：

#定义一个log函数，作为装饰函数

**def** **log**(func): #参数func为一个函数类型，log实现了对func的包装

**def** **wrapper**(\*args, \*\*kw): #定义**wrapper，增强func的功能（返回func的调用结果）**

print('调用函数是 %s' % func.\_\_name\_\_)

**return** func(\*args, \*\*kw) #执行函数并返回结果

**return** wrapper #返回**wrapper函数 ---可以理解为增强的func**

注意：func,与func()的区别

第二步：

#在原now函数定义前加 @封装函数—此处为@log

@log

def now(): #now函数的定义不用改变，只在其定义前增加@装饰函数

print('2016-11-8')

@log相当于在函数定义后再执行now=log(now).

像上面的log函数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）。

我们在使用now时存在如下等价关系(等价关系在装饰时就确定了,而不是在调用时才产生)：

now🡸🡺log(now); now()🡸🡺log(now)()🡸🡺wrapper()

本质上，decorator就是一个返回函数的高阶函数。

在调用函数now时，实际上now已经变为log(now)—即wrapper ; 所以之后的now()相当于wrapper ()，此时执行print(now.\_\_name\_\_) 得到的结果是wrapper

当不在now前加@log时，执行log(now)() -- 也将得到与加了@log一样的结果

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(‘此时正在执行的函数是’) #相当于先调用log(xxx),再用log执行后的返回值来进行装饰

**def** **now**():

print('2016-11-8')

@log(“xxx”)相当于在函数定义后再执行now=log(“xxx”)(now).

使用@log(“xxx”)装饰与@log装饰的区别是: @log(“xxx”)实际上使用的是log(“xxx”)的返回值来进行装饰.

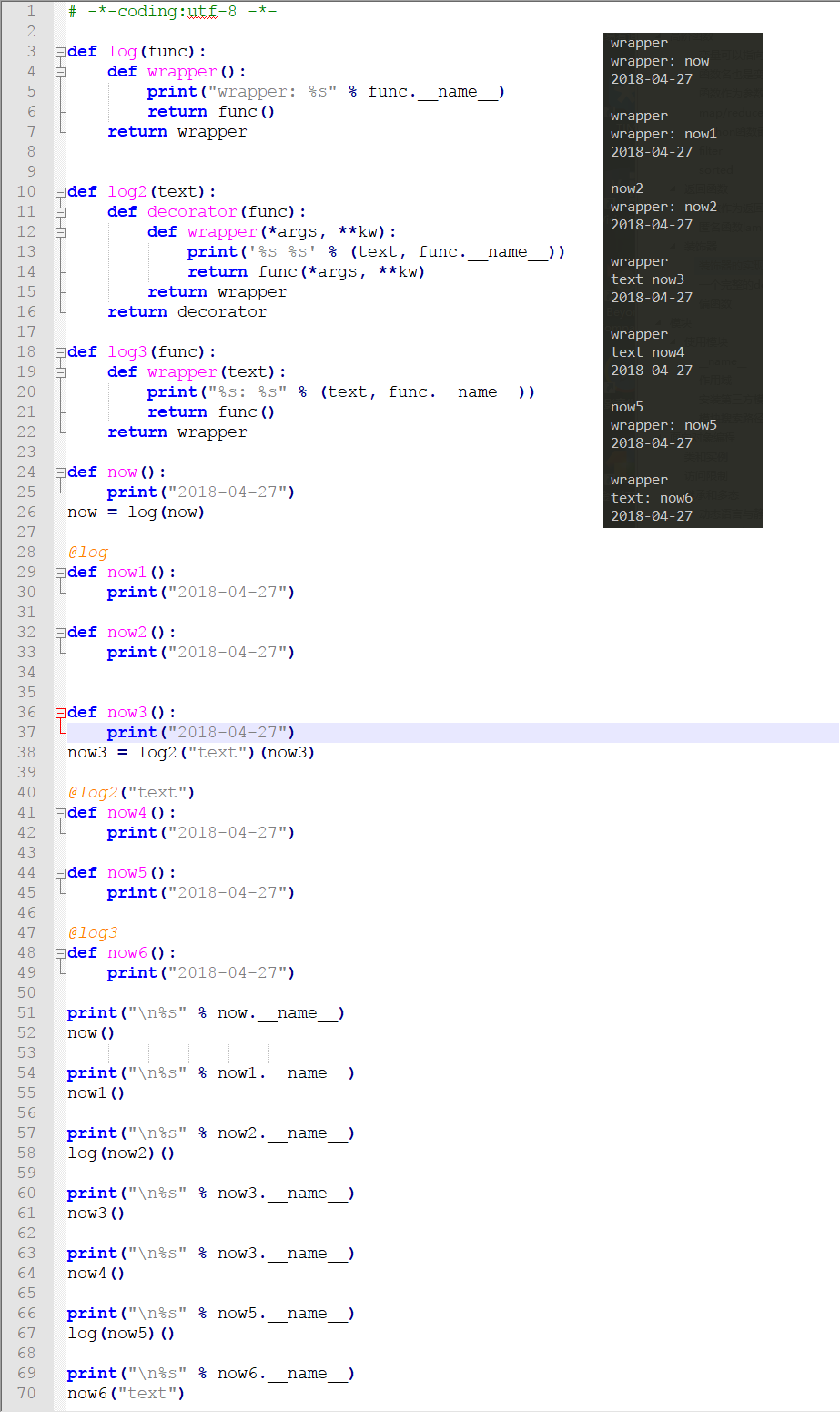
#此处的@log(xxx) 使得now实际上已经变为log(‘此时正在执行的函数是’)(now)—即wrapper，所以在执行now()时实际执行的是wrapper ()。

---总之: 装饰器装饰之后的函数名,表示的已经不是原函数名定义的那个函数了, 而是装饰函数返回的那个函数.

注意：此时执行now.\_\_name\_\_ 得到的结果是wrapper。print(now.\_\_name\_\_) #wrapper

为了避免以上情况（执行now.\_\_name\_\_ 得到的结果是wrapper），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。

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：



### 一个完整的decorator的写法

如下：

import functools

def log(func):

@functools.wraps(func)

def wrapper(\*args, \*\*kw):

print('调用函数是 %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

@log('此时正在执行的函数是')

**def** **now**():

print('2016-11-8')

此时执行now.\_\_name\_\_ 得到的结果是now。但是now的效果仍是log(‘此时正在执行的函数是’)(now) —即wrapper，所以在执行now()时实际执行的是wrapper ()。

在面向对象（OOP）的设计模式中，decorator被称为装饰模式。OOP的装饰模式需要通过继承和组合来实现，而Python除了能支持OOP的decorator外，直接从语法层次支持decorator。Python的decorator可以用函数实现，也可以用类实现。

作业：

1. 编写一个decorator，能在函数调用的前后打印出'begin call'和'end call'的日志。

def log(func): #传入函数func

def wrapper(\*args, \*\*kw): #定义wrapper，增强func的功能（返回func的调用结果）

print('begin call %s()' % func.\_\_name\_\_)

func(\*args, \*\*kw)

print('end call %s()' % func.\_\_name\_\_)

return

return wrapper #返回wrapper函数 ---可以理解为增加的func

@log #('当时正在执行的函数是 ')

def now(): #now函数的定义不用改变，只在其定义前增加@decorator

print('2016-11-8')

print(now())

2. 再思考一下能否写出一个@log的decorator，使它既支持：

@log

def f():

pass

又支持：

@log('execute')

def f():

pass

我的答案：

import functools

def log(\*text):

if isinstance(text[0], str):

def decorator(func):

@functools.wraps(func)

def wrapper(\*args, \*\*kw):

print('%s %s' % (text, func.\_\_name\_\_))

return func(\*args, \*\*kw)

return wrapper

return decorator

else:

@functools.wraps(text[0])

def wrapper1(\*args, \*\*kw):

print('call %s' % text[0].\_\_name\_\_)

return text[0](\*args, \*\*kw)

return wrapper1

@log

def now(): #now函数的定义不用改变，只在其定义前增加@decorator

print('2016-11-8')

@log(‘vvvvv’)

def now1(): #now函数的定义不用改变，只在其定义前增加@decorator

print('2016-11-8')

### 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）。要注意，这里的偏函数和数学意义上的偏函数不一样。

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，于是，我们想到，可以定义一个int2()的函数，默认把base=2传进去：

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

>>> import functools

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

相当于

def int2(x, base=2):

return int(x, base)

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。

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。

# 模块

将函数分别放到不同的文件里来组织代码。在Python中，一个.py文件就称之为一个模块(Module)。

使用模块能大大的提高了代码的可维护性和可重用性。我们可以使用Python内置的模块和来自第三方的模块。

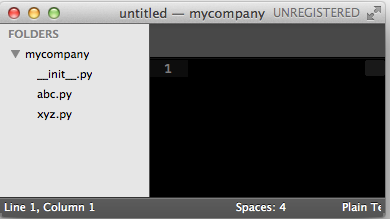
使用模块还可以避免函数名和变量名冲突。存在于不同模块中的相同名字的函数和变量互不影响，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。点[这里](http://docs.python.org/3/library/functions.html" \t "_blank)查看Python的所有内置函数。<https://docs.python.org/3/library/functions.html> （Python的所有内置函数）

分模块避免了函数中变量名的冲突，而且通过包我们又能避免模块名的冲突。

包（Package）是Python引入的按目录来组织模块的方法。

举个例子，abc.py的文件就是名为abc的模块， xyz.py的文件就是名为xyz的模块。

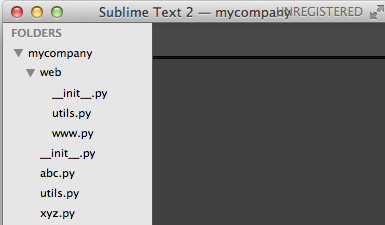
现在，假设我们的abc和xyz这两个模块名字与其他模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名，比如mycompany，按照如下目录存放：



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

请注意，每一个包目录下面都会有一个\_\_init\_\_.py的文件，这个文件使得它所在的目录名是一个模块名—对应的模块是该目录下的\_\_init\_\_.py。 对于Python而言一个目录即一个命名空间，而一个包含\_\_init\_\_.py的目录则为一个模块。\_\_init\_\_.py可以是空文件，也可以有Python代码，因为\_\_init\_\_.py本身就是一个模块，而它的模块名就是mycompany。

类似的，可以有多级目录，组成多级层次的包结构。比如如下的目录结构

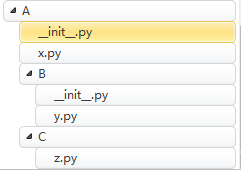


文件www.py的模块名就是mycompany.web.www，两个文件utils.py的模块名分别是mycompany.utils和mycompany.web.utils。

自己创建模块时要注意命名，不能和Python自带的模块名称冲突。例如，系统自带了sys模块，自己的模块就不可命名为sys.py，否则将无法导入系统自带的sys模块。

mycompany.web也是一个模块，请指出该模块对应的.py文件—为myconpany.web.\_\_init\_\_.py。

我的理解：包相当于一个命名空间，将模块放在一个包中即将这个模块加入这个命名空间中。



目录C（不包含\_\_init\_\_.py）---命名空间

目录A,B（包含\_\_init\_\_.py）或称为包. 也是一个模块---对应的模块是该目录下的目录\_\_init\_\_.py

\_\_init\_\_.py ----模块 ----对应的模块名是它所在的目录名

C – 是目录 ---命名空间

x.py – 是模块 :模块名为A.x

A – 是包 – 是模块名 – 对应模块A/ \_\_init\_\_.py

B(A.B) – 是包 – 是模块名 – 对应模块A/B/ \_\_init\_\_.py

y.py – 是模块 – 模块名为A.B.y

## 使用模块

如：

我们编写模块hello.py

#!/usr/bin/env python3 #可以使py文件直接在Unix/Linux/Mac上运行

# -\*- coding: utf-8 -\*- #表示.py文件本身使用标准UTF-8编码

'a test module' #任何模块代码的第一个字符串都被视为模块的文档注释

\_\_author\_ = 'GJY' #使用\_\_author\_\_变量把作者写进去

import sys #导入sys模块

def test():

args = sys.argv #sys模块有一个argv变量，用list存储了命令行的所有参数--与C的main的参数类似

if len(args) == 1:

print('Hello world!')

elif len(args) == 2:

print('Hello, %s!' % args[1])

else:

print('Tool many arguments!')

if \_\_name\_\_ == '\_\_main\_\_':

test()

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

在命令行下执行python3 hello.py 》》将输出Hello world!此时if \_\_name\_\_ == '\_\_main\_\_'

判断成立。

若在命令行下导入import hello 》》则不会输入Hello world!，此时hello.py的if \_\_name\_\_ == '\_\_main\_\_' 不成立。

### \_\_name\_\_

作用：**“Make a script both importable and executable”—使你的脚本既能导入到别的模块又能自己执行。**

\_\_name\_\_是一个特殊变量(表示模块名、默认为\_\_main\_\_)，若a.py文件直接执行则这个文件中的\_\_name\_\_变量为默认值’\_\_main\_\_’; 或在另一个b.py中导入a.py，此时a.py中的\_\_name\_\_变量的值为对应的模块名a。

**例：**先写一个模块：

|  |  |
| --- | --- |
| 1  2  3  4  5 | #module.py  def test():    print "we are in %s" % \_\_name\_\_  if \_\_name\_\_ == '\_\_main\_\_':    test() |

我们执行一下该py文件发现结果是打印出”we are in \_\_main\_\_“,说明我们的if语句中的内容被执行了，调用了test()：

但是如果我们从另我一个模块导入该模块，并调用一次test()函数会是怎样的结果呢？

|  |  |
| --- | --- |
| 1  2  3 | #anothermodle.py  from module import test  test() |

其执行的结果是：we are in module

**总结一下：**

如果我们是直接执行某个.py文件的时候，该文件中那么”\_\_name\_\_ == '\_\_main\_\_'“是True,但是我们如果从另外一个.py文件通过import导入该文件的时候，这时\_\_name\_\_的值就是我们这个py文件的名字而不是\_\_main\_\_。

这个功能还有一个用处：调试代码的时候，在”if \_\_name\_\_ == '\_\_main\_\_'“中加入一些我们的调试代码，我们可以让外部模块调用的时候不执行我们的调试代码，但是如果我们想排查问题的时候，直接执行该模块文件，调试代码能够正常运行！

### 作用域

在一个模块中，我们定义的函数和变量，有的希望给别人使用，有的希望仅仅在模块内部使用。在Python中，是通过’\_’前缀来实现的。

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。比如的\_\_author\_\_，\_\_name\_\_就是特殊变量，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；

在一个模块类似\_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。

python的模块中private函数和变量“不应该”被直接引用，而不是“不能”被直接引用。

（但是在class中类似\_xxx和\_\_xxx这样的私有函数或私有变量就是真正的私有--不能被外部访问）

import hello：相当于加载了一个包吧，可以调用对象hello里的任何函数，形式为hello.xxx,比如hello.test()。

from hello import test：个人感觉就相当于包hello中的test()函数，之后可以直接用test()函数了，如果只是导入了import hello,使用test()时必须加上hello.test()。

### 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。

一般来说，第三方库都会在Python官方的[pypi.python.org](https://pypi.python.org/" \t "_blank)网站注册，要安装一个第三方库，必须先知道该库的名称，可以在官网或者pypi上搜索，比如Pillow的名称叫[Pillow](https://pypi.python.org/pypi/Pillow/" \t "_blank)，因此，安装Pillow的命令就是：

pip install Pillow

### 模块搜索路径

当我们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/3.4/lib/python34.zip', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/plat-darwin', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/lib-dynload', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/site-packages']

如果我们要添加自己的搜索目录，有两种方法：

一是直接修改sys.path，添加要搜索的目录：

>>> **import** sys

>>> sys.path.append('/Users/michael/my\_py\_scripts')

这种方法是在运行时修改，运行结束后失效。

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

# 面向对象编程

面向对象编程——Object Oriented Programming，简称OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

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

定义一个学生类Student，这种数据类型拥有name和score这两个属性（Property）

**class Student(object):**

**def** **\_\_init\_\_**(self, name, score):

self.name = name

self.score = score

**def** **print\_score**(self):

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

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。

bart = Student('Bart Simpson', 59)

lisa = Student('Lisa Simpson', 87)

bart.print\_score()

lisa.print\_score()

类（Class）和实例（Instance）。

数据封装、继承和多态是面向对象的三大特点

我的理解：**\_\_init\_\_ 相当于C++中的构造函数，\_\_init\_\_的第一个函数self相当于C++的this指针—表示这个类的对象（不过C++中的this在类成员中是隐式的，而这里的self需要显式的给出—当然不一定非得取名为self,也可以叫SB、haha）.**

**详细请看《Python中的 \_\_init\_\_和self》**

<http://stackoverflow.com/questions/8609153/why-do-we-use-init-in-python-classes>

<http://stackoverflow.com/questions/625083/python-init-and-self-what-do-they-do>

## 类和实例

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

与静态语言不同，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：

class Student(object):

def \_\_init\_\_(self, name, score):

self.name = name

self.score = score

bart = Student('aaa', 77)

lisa = Student('Lisa Simpson', 87)

bart.age = 8 # bart对象增加了一个age成员

print(bart.age) #>>8， 输出8

print(lisa.age) #将报错，因为lisa无age成员

Traceback (most recent call last):

File "E:\ProjectFile\Py\py.py", line 11, in <module>

print(lisa.age)

AttributeError: 'Student' object has no attribute 'age'

## 访问限制

在类中

特殊变量：以\_\_（双下划线）开头、并以\_\_（双下划线）结束的变量。变量名类似\_\_xxx\_\_。

特殊变量可以直接访问。

私有变量：以\_\_（双下划线）开头、但不以\_\_结束的变量。变量名类似\_\_xxx。

私有变量只在类的内部可以访问，外部不能访问。因为Python解释器自动在其变量名前加\_类名。\_\_xxx实际上变为\_CalssName\_\_name,( CalssName为该变量所属类的类名)—不建议通过变更后的名字从外部访问类的私有变量，因为因为不同版本的Python解释器可能会把\_\_xxx改成不同的变量名。

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

class Student(object):

def \_\_init\_\_(self, name, score):

self.\_\_name = name

self.\_\_score = score

bart = Student('gjy', 22)

print(bart.\_\_name) #执行时出错。AttributeError: 'Student' object has no attribute '\_\_name'

\_\_name是Student的私有变量，在类的外部不能直接访问。

但是执行print(bart.\_Student\_\_name)时可以访问Student的\_\_name变量，因为类的\_\_name成员变量实际上被处理成这个名字。

在模块中

也可以定义以\_\_（双下划线）开头的变量，将其当作模块的私有变量使用，但是在模块中(不在class中)的这种变量并不是实际上的私有。

如在py1.py中有

\_\_myname = ‘gjy’

在py2.py中

import py1

print(py1.\_\_myname) #可以正常执行py1.py中的\_\_myname并不是真正的私有变量。但它的名字表明了它的设计者希望把它当作私有变量看待。

最后注意下面的这种错误写法：

>>> bart = Student('Bart Simpson', 98)

>>> bart.get\_name()

'Bart Simpson'

>>> bart.\_\_name = 'New Name' *# 设置\_\_name变量！*

>>> bart.\_\_name

'New Name'

表面上看，外部代码“成功”地设置了\_\_name变量，但实际上这个\_\_name变量和class内部的\_\_name变量*不是*一个变量！内部的\_\_name变量已经被Python解释器自动改成了\_Student\_\_name，而外部代码给bart新增了一个\_\_name变量。不信试试：

>>> bart.get\_name() *# get\_name()内部返回self.\_\_name*

'Bart Simpson'

## 继承和多态

直接举例

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

**class Animal(object):**

**def** **run**(self):

print('Animal is running...')

当我们需要编写Dog和Cat类时，就可以直接从Animal类继承：

**class Dog(Animal):**

**pass**

**class Cat(Animal):**

**pass**

对于Dog来说，Animal就是它的父类，对于Animal来说，Dog就是它的子类。Cat和Dog类似。最大的好处是子类获得了父类的全部功能。

dog = Dog()

dog.run()

cat = Cat()

cat.run()

运行结果如下：

Animal **is** running...

Animal **is** running...

可以子类确实继承了父类的功能。

但是当我们希望子类在执行run时具有子类的特色时，比如在Dog和Cat，它们run()的时候，分别显示Dog is running...和Cat is running...。这里我们需要在子类中重写run函数覆盖父类的run。

**class Dog(Animal):**

**def** **run**(self):

print('Dog is running...')

**class Cat(Animal):**

**def** **run**(self):

print('Cat is running...')

再次运行，结果如下：

Dog **is** running...

Cat **is** running...

通过继承我们还能实现多态。

我们再编写一个函数，这个函数接受一个名为animal的参数。如下。

**def** **run\_test**(animal):

animal.run()

当我们传入Animal的实例时，run\_test()就打印出：

>>> run\_test(Animal())

Animal **is** running...

当我们传入Dog的实例时，run\_test()就打印出：

>>> run\_test(Dog())

Dog **is** running...

当我们传入Cat的实例时，run\_test()就打印出：

>>> run\_test(Cat())

Cat **is** running...

run\_test()中实际执行的run函数根据我们传入的对象而不同。当我们传入的是Dog对象时执行的是Dog对象的run; 当我们传入的是Cat对象时执行的是Cat对象的run。

### 动态语言与静态语言多态的不同

python是动态语言，不同于C++静态语言。c++在实现多态时传入的对象必须是指定的类型或者它的子类型。对于Python这样的动态语言来说，我们只需要保证传入的对象有一个run()方法就可以了。

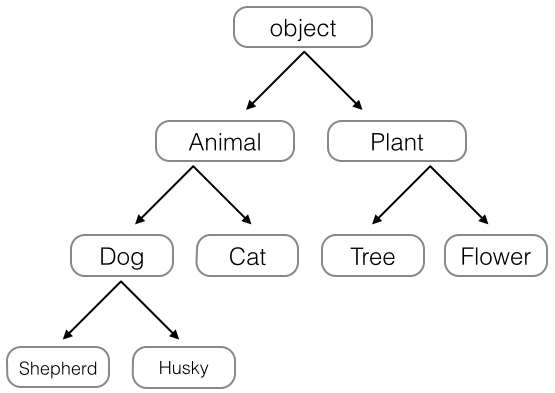
再次看上面的run\_test函数。

**def** **run\_test**(animal):

animal.run()

我们说run\_test接受一个名为animal的参数，而不是说接受一个Animal类型的参数。因为python是动态类型，其变量的类型是在运行时确定的，也就是说在执行时我们给它传递什么类型的对象，它就是什么类型。所以只要我们传递的对象（类型）具有run方法，那么这个函数调用就是正确的。---这是不同与C++的。

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



## 获取对象信息

当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

### 使用type()

使用type()函数，判断对象类型，

>>> type(123)

<**class 'int'>**

**>>> type('str')**

**<class 'str'>**

**>>> type(None)**

**<type(None) 'NoneType'>**

**一个变量指向函数或者类，也可以用type()判断：**

**>>> type(abs)**

**<class 'builtin\_function\_or\_method'>**

**>>> type(a)**

**<class '\_\_main\_\_.Animal'>**

在if语句中判断，就需要比较两个变量的type类型是否相同：

>>> type(123)==type(456)

True

>>> type(123)==int

True

>>> type('abc')==type('123')

True

>>> type('abc')==str

True

>>> type('abc')==type(123)

False

判断基本数据类型可以直接写int，str等，但如果要判断一个对象是否是函数怎么办？可以使用types模块中定义的常量：

>>> **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

### 使用isinstance()

对于class的继承关系来说，使用type()就很不方便。我们要判断class的类型，可以使用isinstance()函数。isinstance(对象, 类型) 当对象是这个类型，或对象是这个类型的子类则isinstance()为真。

### 使用dir()

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

>>> dir('ABC')

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', '**replace**', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

类似

的，比如\_\_len\_\_方法返回长度。在Python中，如果你调用len()函数试图获取一个对象的长度，实际上，在len()函数内部，它自动去调用该对象的\_\_len\_\_()方法，所以，下面的代码是等价的：

>>> len('ABC')

3

>>> 'ABC'.\_\_len\_\_()

3

我们自己写的类，如果也想用len(myObj)的话，就自己写一个\_\_len\_\_()方法：

>>> **class MyDog(object):**

... **def** **\_\_len\_\_**(self):

... **return** 100

...

>>> dog = MyDog()

>>> len(dog)

100

### getattr()、setattr()以及hasattr()

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

>>> class MyObject(object):

... def \_\_init\_\_(self):

... self.x = 9

... def power(self):

... return self.x \* self.x

...

>>> obj = MyObject()

紧接着，可以测试该对象的属性：

>>> hasattr(obj, 'x') # 有属性'x'吗？

True

>>> obj.x

9

>>> hasattr(obj, 'y') # 有属性'y'吗？

False

>>> setattr(obj, 'y', 19) # 设置一个属性'y'

>>> hasattr(obj, 'y') # 有属性'y'吗？

True

>>> getattr(obj, 'y') # 获取属性'y'

19

>>> obj.y # 获取属性'y'

19

如果试图获取不存在的属性，会抛出AttributeError的错误：

>>> getattr(obj, 'z') # 获取属性'z'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'MyObject' object has no attribute 'z'

可以传入一个default参数，如果属性不存在，就返回默认值：

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

404

也可以获得对象的方法：

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

True

>>> getattr(obj, 'power') # 获取属性'power'

<bound method MyObject.power of <\_\_main\_\_.MyObject object at 0x10077a6a0>>

>>> fn = getattr(obj, 'power') # 获取属性'power'并赋值到变量fn

>>> fn # fn指向obj.power

<bound method MyObject.power of <\_\_main\_\_.MyObject object at 0x10077a6a0>>

>>> fn() # 调用fn()与调用obj.power()是一样的

81

### 实例属性和类属性

由于Python是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过self变量：

class Student(object):

def \_\_init\_\_(self, name):

self.name = name #类对象属性name

s = Student('Bob') #实例化一个类对象

s.score = 90 #为类对象s绑定一个属性score

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

class Student(object):

name = 'Student' #这个属性属于类，可通过类名和对象来访问—从这点上看

#类似C++的static成员。但是类的实例可定义同名属性屏蔽类属性

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

>>> class Student(object):

... name = 'Student'

...

>>> s = Student() # 创建实例s

>>> print(s.name) # 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性

Student

>>> print(Student.name) # 打印类的name属性

Student

>>> s.name = 'Michael' # 给实例绑定name属性---注意：这里是给实例绑定属性，而不是给类绑定属性

>>> print(s.name) # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性

Michael

>>> print(Student.name) # 但是类属性并未消失，用Student.name仍然可以访问

Student

>>> del s.name # 如果删除实例的name属性

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

Student

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

class Student(object): #定义一个类

pass

s = Student(); #实例化一个类对象

s.name = 'Michael' #为类对象 绑定一个name属性

print(s.name)

print(dir(s)) #查看一个对象的所有属性和方法

print(dir(Student)) ##查看一个类的所有属性和方法

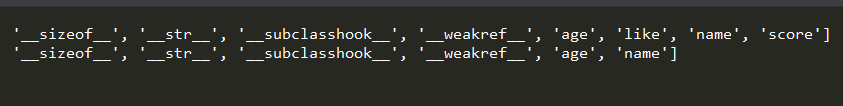
输出结果如下。可以发现为类对象绑定的name属性只属于对象而不属于类。

Michael

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'name']

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_']

*class* Student(object):  
 name = "gjy" #类属性(定义在类中) --所有Student对象共有属性  
 *def \_\_new\_\_*(cls):  
 cls.age = 24 #类属性(在\_\_new\_\_中通过cls参数定义)  
 *return* object.\_\_new\_\_(cls)  
 *def \_\_init\_\_*(self):  
 self.score = 90 #对象属性, 对象特有(但是因为这个属性定义在\_\_init\_\_中,因为所有对象都会拥有这个属性)  
 sex = "man" #局部变量  
s = Student()  
s.like = "看电影" #对象属性, s对象特有  
print(dir(s))  
print(dir(Student))



# 面向对象高级编程

class Student(object): #定义一个类

pass

s = Student(); #实例化一个类对象

s.name = 'Michael' #为实例化的类对象绑定name属性

print(s.name)

def set\_age(self, age):

self.age = age

from types import MethodType

s.set\_age = MethodType(set\_age, s) # 给实例绑定一个方法

s.set\_age(25)

print(s.age)

print(dir(s))

print(dir(Student))

给类对象s绑定的属性和方法只对该对象起作用，对类的其他对象不起作用。要想绑定的属性或方法对类的所有对象起作用则需要绑定到类上（而不是类对象上）。

def set\_score(self, score):

self.score = score

Student.set\_score = set\_score #为类绑定方法

print(dir(s))

print(dir(Student))

笔记：搞清楚类与实例的关系，类的属性和方法对类的所有实例有效，但绑定到类对象的属性和方法仅对这个对象有效

## 使用\_\_slots\_\_

\_\_slots\_\_用来限制实例的属性。比如，只允许对Student实例添加name和age属性。

在定义class的时候，需要定义一个特殊的\_\_slots\_\_变量，来限制该class实例能添加的属性：

class Student(object):

\_\_slots\_\_ = ('name', 'age') # 用tuple定义允许绑定的属性名称

然后，我们试试：

>>> s = Student() # 创建新的实例

>>> s.name = 'Michael' # 绑定属性'name'

>>> s.age = 25 # 绑定属性'age'

>>> s.score = 99 # 绑定属性'score'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Student' object has no attribute 'score'

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

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

>>> class GraduateStudent(Student):

... pass

...

>>> g = GraduateStudent()

>>> g.score = 9999

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

## 使用@property与@xxx.setter来隐藏属性

绑定属性时，如果我们不应该直接把属性暴露出去，这样无法检查参数，容易导致使用不合法的参数。

比如：

s = Student() #定义一个类对象

s.score = 9999 #为类对象绑定一个socre属性，并设置为9999。

我们的分数范围一般为0-100，9999显然不合逻辑，那么我们如何限制score属性的范围呢？

我们将score属性设置为私有(\_score)，再通过一个set\_score()方法来设置成绩，和一个get\_score()来获取成绩。

class Student(object):

def get\_score(self):

return self.\_score #参考私有成员的命名

def set\_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

现在，对任意的Student实例进行操作，就不能随心所欲地设置score了：

>>> s = Student()

>>> s.set\_score(60) # ok!

>>> s.get\_score()

60

>>> s.set\_score(9999)

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

另一种更简单的方法是使用Python内置的@property装饰器。

前面我们看到装饰器（decorator）可以给函数动态加上功能，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): #在该方法前面加@property,便可用操作属性的方式来调用这个方法。

**return** self.\_score

@score.setter

**def** **score**(self, value): #在该方法前面加@属性名.setter,便可以操作属性的方式来调用这个方法。

**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

把一个getter方法变成属性，只需要加上@property就可以了，此时，@property本身又创建了另一个装饰器@xxx.setter(xxx为@property对应的方法名)，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

>>> s = Student()

>>> s.score = 60 # OK，实际转化为s.set\_score(60)

>>> s.score # OK，实际转化为s.get\_score()

60

>>> s.score = 9999

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

当我们只需要只读，只定义getter方法即可(不需要@xxx.setter)

## 多重继承

## 定制类

看到类似\_\_slots\_\_这种形如\_\_xxx\_\_的变量或者函数名就要注意，这些在Python中是有特殊用途的。

\_\_slots\_\_我们已经知道怎么用了，\_\_len\_\_()方法我们也知道是为了能让class作用于len()函数。

除此之外，Python的class中还有许多这样有特殊用途的函数，可以帮助我们定制类。

### \_\_str\_\_

我们先定义一个Student类，打印一个实例：

>>> class Student(object):

... def \_\_init\_\_(self, name):

... self.name = name

...

>>> print(Student('Michael'))

<\_\_main\_\_.Student object at 0x109afb190>

打印出一堆<\_\_main\_\_.Student object at 0x109afb190>，不好看。

怎么才能打印得好看呢？只需要定义好\_\_str\_\_()方法，返回一个好看的字符串就可以了：

>>> 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)

这样打印出来的实例，不但好看，而且容易看出实例内部重要的数据。

但是细心的朋友会发现直接敲变量不用print，打印出来的实例还是不好看：

>>> 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。

解决办法是再定义一个\_\_repr\_\_()。但是通常\_\_str\_\_()和\_\_repr\_\_()代码都是一样的，所以，有个偷懒的写法：

**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。

我们以斐波那契数列为例，写一个Fib类，可以作用于for循环：

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

现在，试试把Fib实例作用于for循环：

>>> for n in Fib():

... print(n)

...

1

1

2

3

5

...

46368

75025

### \_\_getitem\_\_

Fib实例虽然能作用于for循环，看起来和list有点像，但是，把它当成list来使用还是不行，比如，取第5个元素：

>>> Fib()[5]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'Fib' object does not support indexing

要表现得像list那样按照下标取出元素，需要实现\_\_getitem\_\_()方法：

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

但是list有个神奇的切片方法：

>>> list(range(100))[5:10]

[5, 6, 7, 8, 9]

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

现在试试Fib的切片：

>>> 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：

>>> f[:10:2]

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

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。

### \_\_setitem\_\_

\_\_setitem\_\_()方法与\_\_getitem\_\_对应，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\_\_

正常情况下，当我们访问的类方法或属性不存在时，就会报错。而\_\_getattr\_\_指定了在访问类（或实例）中没有的属性（或方法）时，所产生的行为。

首先我们看看没有写\_\_getattr\_\_方法时的情况：

class Student(object):

def \_\_init\_\_(self):

self.name = 'Michael'

调用name属性，没问题，但是，调用不存在的score属性，就有问题了：

>>> s = Student()

>>> print(s.name) # Student实例具有name属性，所以s.name访问正常

Michael

>>> print(s.score) # Student实例没有score属性，所以s. score访问异常

Traceback (most recent call last):

...

AttributeError: 'Student' object has no attribute 'score'

错误信息很清楚地告诉我们，没有找到score这个attribute。

\_\_getattr\_\_()方法，可以指定当不小心访问了类或实例不具有的属性时所产生的效果。修改如下：

class Student(object):

def \_\_init\_\_(self):

self.name = 'Michael'

#只有访问对象不存在的属性时，才调用\_\_getattr\_\_

def \_\_getattr\_\_(self, attr):

if attr=='score': #当访问类或实例中不存在的’score’属性时, 进入该分支

return 99 #不只可以返回数据，还可以返回其它任何对象

if attr=='name':

return 'Jack'

#对于不需要处理的访问，抛出AttributeError

raise AttributeError('\'Student\' object has no attribute \'%s\'' % attr)

当调用不存在的属性时，比如score，Python解释器才会试图调用\_\_getattr\_\_(self, 'score')来尝试获得属性，这样，我们就有机会返回score的值：

>>> s = Student()

>>> s.name # Student实例具有name属性，所以访问s.name时不会调用\_\_getattr\_\_方法，

'Michael'

>>> s.score # Student实例不具有score属性，所以访问s.score时会调用\_\_getattr\_\_方法，if attr=='score'判断成立。

99

返回函数也是完全可以的：

class Student(object):

def \_\_getattr\_\_(self, attr):

if attr=='age':

return lambda: 25

只是调用方式要变为：

>>> s.age()

25

**注意**，只有在没有找到属性的情况下，才调用\_\_getattr\_\_，已有的属性，比如name，不会在\_\_getattr\_\_中查找。

此外，注意到任意调用如s.abc都会返回None，这是因为我们定义的\_\_getattr\_\_默认返回就是None。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 \'%s\'' % attr)

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。

这种完全动态调用的特性有什么实际作用呢？作用就是，可以针对完全动态的情况作调用。

举个例子：

现在很多网站都搞REST API，比如新浪微博、豆瓣啥的，调用API的URL类似：

* [http://api.server/user/friends](http://api.server/user/friends" \t "_blank)
* [http://api.server/user/timeline/list](http://api.server/user/timeline/list" \t "_blank)

如果要写SDK，给每个URL对应的API都写一个方法，那得累死，而且，API一旦改动，SDK也要改。利用完全动态的\_\_getattr\_\_，我们可以写出一个链式调用：

class Chain(object):

def \_\_init\_\_(self, path=''):

self.\_path = path

def \_\_getattr\_\_(self, path):

return Chain('%s/%s' % (self.\_path, path)) #递归

def \_\_str\_\_(self):

return self.\_path

\_\_repr\_\_ = \_\_str\_\_

试试：

>>> Chain().status.user.timeline.list

'/status/user/timeline/list'

[分析见后面的完全动态调用特性小节。](#完全动态调用特性分析)

这样，无论API怎么变，SDK都可以根据URL实现完全动态的调用，而且，不随API的增加而改变！

还有些REST API会把参数放到URL中，比如GitHub的API：

GET /users/:user/repos

调用时，需要把:user替换为实际用户名。如果我们能写出这样的链式调用：

Chain().users('michael').repos

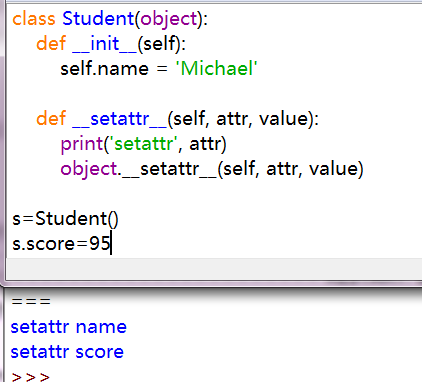
就可以非常方便地调用API了。有兴趣的童鞋可以试试写出来。

*class* Chain(object):  
 *def \_\_init\_\_*(self, *path*=''):  
 self.\_path = *path  
 def \_\_getattr\_\_*(self, *attr*):  
 *if attr*=="users":  
 *def* usersf(*userName*):  
 self.\_userName = *userName  
 return* Chain('%s/%s' % (self.\_path, "users/"+self.\_userName))  
 *return* usersf  
 *return* Chain('%s/%s' % (self.\_path, *attr*))  
 *def \_\_str\_\_*(self):  
 *return* self.\_path  
 \_\_repr\_\_ = \_\_str\_\_

s = Chain().users('michael').repos  
print(s)

### \_\_setattr\_\_

如果定义了这个方法，那么在动态添加实例的属性时，都会调用这个方法。



可以发现print('setattr', attr)执行了两次，在\_\_init\_\_中动态创建实例的name属性时，和s.score=95语句动态创建实例的score属性时，都调用了\_\_setattr\_\_方法。

当然既使没有写\_\_setattr\_\_也能动态了添加实例的属性(有默认的\_\_setattr\_\_)，但是当我们希望在动态的添加实例属性的时候做一些事情的话，重写\_\_setattr\_\_也不失为一种方法。

### \_\_call\_\_

一个对象实例可以有自己的属性和方法，当我们为类实现了\_\_call\_\_方法时，便可以通过实例直接调用方法（ instance.method()方式）。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。----有点类似于C++中重载了函数调用操作符()的类

请看示例：

class Student(object):

def \_\_init\_\_(self, name):

self.name = name

def \_\_call\_\_(self):

print('My name is %s.' % self.name)

调用方式如下：

>>> s = Student('Michael')

>>> s() # 直接通过类对象调用函数—是不是像重载了()的C++类。self参数不需要传入

My name is Michael.

\_\_call\_\_()还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

那么，怎么判断一个变量是对象还是函数呢？其实，更多的时候，我们需要判断一个对象是否能被调用，能被调用的对象就是一个Callable对象，比如函数和我们上面定义的带有\_\_call\_\_()的类实例：

>>> callable(Student())

True

>>> callable(max)

True

>>> callable([1, 2, 3])

False

>>> callable(None)

False

>>> callable('str')

False

通过callable()函数，我们就可以判断一个对象是否是“可调用”对象。

小结

Python的class允许定义许多定制方法，可以让我们非常方便地生成特定的类。

本节介绍的是最常用的几个定制方法，还有很多可定制的方法，请参考[Python的官方文档](http://docs.python.org/3/reference/datamodel.html" \l "special-method-names" \t "_blank)。

### 完全动态调用特性分析：

# =============================================

# 完全动态调用特性：

# 把一个类的所有属性和方法调用全部动态化处理

# \_\_call\_\_(): 用于实例自身的调用，达到()调用的效果

# 即可以把此类的对象当作函数来使用，相当于重载了括号运算符

# \_\_getattr\_\_(): 当调用不存在的属性时调用此方法来尝试获得属性

class Chain(object):

def \_\_init\_\_(self, path=''):

self.\_path = path

def \_\_getattr\_\_(self, path):

print('call \_\_getattr\_\_(%s)' % path)

return Chain('%s/%s' % (self.\_path, path)) #返回一个Chain对象，构造这个对象

#的参数为’self.\_path/path’。

def \_\_str\_\_(self):

return self.\_path

\_\_repr\_\_ = \_\_str\_\_

>>> Chain().status.user.timeline.list调用

输出结果为'/status/user/timeline/list'的分析

这其实就是一个递归构造（类似于递归函数，只是没有回溯而已）。不断的用当前对象构造下一个对象，最后调用\_\_repr\_\_()函数。

1. 首先执行Chain()返回一个实例对象C1—它的构造函数参数为空，

2. Chain().status即C1.status，因为C1中不存在status属性，所以就会调用 \_\_getattr\_\_()它返回一个Chain对象C2—它的构造函数参数为/status—相当于Chain(‘status’)

3. Chain().status.user即C2. user，因为C2中不存在user属性，所以就会调用 \_\_getattr\_\_()它返回一个Chain对象C3—它的构造函数参数为/status/user—相当于Chain(‘/status/user’)

1. Chain().status.user.timeline即C3. timeline，因为C3中不存在timeline属性，所以就会调用 \_\_getattr\_\_()它返回一个Chain对象C4—它的构造函数参数为/status/user/timeline—相当于Chain(‘status/user/timeline’)
2. Chain().status.user.timeline.list即C4. timeline，因为C4中不存在list属性，所以就会调用 \_\_getattr\_\_()它返回一个Chain对象C5—它的构造函数参数为/status/user/timeline/list—相当于Chain(‘/status/user/timeline/list’)
3. >>>Chain().status.user.timeline.list 即C5—直接执行类实例C5将调用C5的\_\_repr\_\_函数。输出C5的\_path即/status/user/timeline/list

## 使用枚举类

枚举类型定义一个class类型，然后，每个常量都是class的一个唯一实例。Python提供了Enum类来实现这个功能：

from **enum** **import** Enum

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

这样我们就获得了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)

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

可见，既可以用成员名称引用枚举常量，又可以直接根据value的值获得枚举常量。

小结

Enum可以把一组相关常量定义在一个class中，且class不可变，而且成员可以直接比较

## 使用元类

### **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

比方说我们要定义一个Hello的class，就写一个hello.py模块：

**class Hello(object):**

**def** **hello**(self, name='world'):

print('Hello, %s.' % name)

当Python解释器载入hello模块时，就会依次执行该模块的所有语句，执行结果就是动态创建出一个Hello的class对象，测试如下：

>>> from hello **import** Hello

>>> h = Hello()

>>> h.hello()

Hello, world.

>>> print(type(Hello))

<**class 'type'>**

**>>> print(type(h))**

**<class 'hello.Hello'> #hello表示模块名**

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。

我们说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。

通过type()函数创建的类和直接写class是完全一样的，因为Python解释器遇到class定义时，仅仅是扫描一下class定义的语法，然后调用type()函数创建出class。

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。

### **metaclass**

除了使用type()动态创建类以外，要控制类的创建行为，还可以使用metaclass。

metaclass，直译为元类，简单的解释就是：

当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。

但是如果我们想创建出类呢？那就必须根据metaclass创建出类，所以：先定义metaclass，然后创建类。

连接起来就是：先定义metaclass，就可以创建类，最后创建实例。

--- metaclass类似于C++中的模板类。

我们先看一个简单的例子，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*):  
 print(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*#测试一下MyList是否可以调用add()方法：  
L = MyList()  
L.add(1)  
print(L)



当我们传入关键字参数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。

元类的\_\_new\_\_()方法接收到的参数依次是：

1. 元类(cls: ListMetaclass)
2. 要创建的类名(name: MyList)；
3. 要创建的类继承的父类集合(bases: (list, ))；
4. 要创建类的属性集合。

## 错误、调试和测试

### 错误处理

内置了一套try...except...finally...的错误处理机制

当我们认为某些代码可能会出错时，就可以用try来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即except语句块，执行完except后，如果有finally语句块，则执行finally语句块，至此，执行完毕。由于没有错误发生，except语句块不会被执行，但是如果有finally，则finally一定会被执行（可以没有finally语句）。

错误有很多种类，如果发生了不同类型的错误，应该由不同的except语句块处理。没错，可以有多个except来捕获不同类型的错误：一个例子如下：

**try**:

print('try...')

r = 10 / int('a')

print('result:', r)

**except** ValueError **as** e:

print('ValueError:', e)

**except** ZeroDivisionError **as** e:

print('ZeroDivisionError:', e)

**finally**:

print('finally...')

print('END')

注意：Python的错误其实也是class，所有的错误类型都继承自BaseException，错误类型有一个继承结构，在判断错误类型时子类类型的**except**语句应该写在前面。否则父类的类型**except**将捕获子类的**except。如下：**

**try**:

foo()

**except** ValueError **as** e:

print('ValueError')

**except** UnicodeError **as** e:

print('UnicodeError')

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

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

[https://docs.python.org/3/library/exceptions.html#exception-hierarchy](https://docs.python.org/3/library/exceptions.html" \l "exception-hierarchy" \t "_blank)

### 调用堆栈

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。来看看err.py：

*# err.py:*

**def** **foo**(s):

**return** 10 / int(s)

**def** **bar**(s):

**return** foo(s) \* 2

**def** **main**():

bar('0')

main()

执行，结果如下：

$ 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

### 记录错误

Python内置的logging模块可以非常容易地记录错误信息

**def** **main**():

**try**:

bar('0')

**except** Exception **as** e:

logging.exception(e)

### 抛出错误

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

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

*# err\_raise.py*

**class FooError(ValueError):**

**pass**

**def** **foo**(s):

n = int(s)

**if** n==0:

**raise** FooError('invalid value: %s' % s)

**return** 10 / n

foo('0')

执行，可以最后跟踪到我们自己定义的错误：

$ python3 err\_raise.py

Traceback (most recent **call** **last**):

File "err\_throw.py", line 11, **in** <**module**>

foo('0')

File "err\_throw.py", line 8, **in** foo

raise FooError('invalid value: %s' % s)

\_\_main\_\_.FooError: invalid **value**: 0

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

### 调试

第一种方法：简单直接粗暴有效，就是用print()把可能有问题的变量打印出来看看

第二种方法:断言:

凡是用print()来辅助查看的地方，都可以用断言（assert）来替代,

如果断言失败，assert语句本身就会抛出AssertionError。

启动Python解释器时可以用-O参数来关闭assert

第三种方法：logging

把print()替换为logging是第3种方式，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)

第4种方法：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。

$ python3 -m pdb err.py

查看代码： l

单步执行代码： n

查看变量： p 变量名

结束调试： q

第4种方法：pdb.set\_trace()

这个方法也是用pdb，但是不需要单步执行，我们只需要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，命令p查看变量，或者用命令c继续运行

终级方法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。目前比较好的Python IDE有PyCharm：

## 单元测试

这种以测试为驱动的开发模式最大的好处就是确保一个程序模块的行为符合我们设计的测试用例。在将来修改的时候，可以极大程度地保证该模块行为仍然是正确的。

我们来编写一个Dict类，这个类的行为和dict一致，但是可以通过属性来访问，用起来就像下面这样：

>>> d = Dict(a=1, b=2)

>>> d['a']

1

>>> d.a

1

mydict.py代码如下：----我们实现的功能模块

**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

为了编写单元测试，我们需要引入Python自带的unittest模块。

编写mydict\_test.py----测试模块

**import** unittest

**from** mydict **import** Dict

**class TestDict(unittest.TestCase): #从**unittest.TestCase继承

**def** **test\_init**(self):

d = Dict(a=1, b='test')

self.assertEqual(d.a, 1)  *#断言*d.a*与1相等*

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): #断言期待抛出KeyError类型的Error

value = d['empty']

**def** **test\_attrerror**(self):

d = Dict()

**with** self.assertRaises(AttributeError): #断言期待抛出AttributeError类型的Error

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，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是assertEqual()：

**self**.assertEqual(abs(-1), 1) *# 断言函数返回的结果与1相等*

另一种重要的断言就是期待抛出指定类型的Error，比如通过d['empty']访问不存在的key时，断言会抛出KeyError：

with **self**.assertRaises(KeyError):

value = d['empty']

而通过d.empty访问不存在的key时，我们期待抛出AttributeError：

with **self**.assertRaises(AttributeError):

value = d.**empty**

### 运行单元测试

一旦编写好单元测试，我们就可以运行单元测试。

**最简单的运行方式**是在mydict\_test.py的最后加上两行代码：

**if** \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

这样就可以把mydict\_test.py当做正常的python脚本运行：

$ python3 mydict\_test.py

**另一种方法**是在命令行通过参数-m unittest直接运行单元测试：

$ python3 -m unittest mydict\_test

.....

----------------------------------------------------------------------

Ran 5 tests **in** 0.000s

OK

这是推荐的做法，因为这样可以一次批量运行很多单元测试，并且，有很多工具可以自动来运行这些单元测试。

### setUp与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调用**每一个**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...')

可以再次运行测试看看每个测试方法调用前后是否会打印出setUp...和tearDown...。

小结

单元测试可以有效地测试某个程序模块的行为，是未来重构代码的信心保证。

单元测试的测试用例要覆盖常用的输入组合、边界条件和异常。

单元测试代码要非常简单，如果测试代码太复杂，那么测试代码本身就可能有bug。

单元测试通过了并不意味着程序就没有bug了，但是不通过程序肯定有bug。

### 文档测试

当我们编写注释时，如果写上这样的注释：

**def** **abs**(n):

'''

Function to get absolute value of number.

Example:

>>> abs(1) #注意>>>与abs(1)之间有一个空格

1

>>> abs(-1)

1

>>> abs(0)

0

'''

**return** n **if** n >= 0 **else** (-n)

这相当于告诉函数的调用者该函数的期望输入和输出。

并且，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。

让我们用doctest来测试上次编写的Dict类：

*# mydict2.py*

**class Dict(dict):**

'''

Simple dict but also support access as x.y style.

>>> d1 = Dict()

>>> d1['x'] = 100

>>> d1.x

100

>>> d1.y = 200

>>> d1['y']

200

>>> d2 = Dict(a=1, b=2, c='3')

>>> d2.c

'3'

>>> d2['empty']

Traceback (most recent call last):

...

KeyError: 'empty'

>>> d2.empty

Traceback (most recent call last):

...

AttributeError: 'Dict' object has no attribute 'empty'

'''

**def** **\_\_init\_\_**(self, \*\*kw):

super(Dict, self).\_\_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

**if** \_\_name\_\_=='\_\_main\_\_': #这段的作用见[\_\_name\_\_](#___name__)

**import** doctest

doctest.testmod()

运行python3 mydict2.py：

$ python3 mydict2.py

什么输出也没有。这说明我们编写的doctest运行都是正确的。如果程序有问题，比如把\_\_getattr\_\_()方法注释掉，再运行就会报错：

$ python3 mydict2.py

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

File "/Users/michael/Github/learn-python3/samples/debug/mydict2.py", line 10, in \_\_main\_\_.Dict

Failed example:

d1.x

Exception raised:

Traceback (most recent **call** **last**):

...

AttributeError: 'Dict' object has **no** attribute 'x'

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

File "/Users/michael/Github/learn-python3/samples/debug/mydict2.py", line 16, **in** \_\_main\_\_.Dict

Failed example:

d2.c

**Exception** raised:

Traceback (most recent **call** **last**):

...

AttributeError: 'Dict' object has **no** attribute 'c'

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1 items had failures:

2 **of** 9 **in** \_\_main\_\_.Dict

\*\*\*Test Failed\*\*\* 2 failures.

注意到最后3行代码。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。

# IO编程

## 同步与异步概念

## 文件读写

读写文件是最常见的IO操作。Python内置了读写文件的函数，用法和C是兼容的。

### 读文件

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

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

标示符'r'表示读，这样，我们就成功地打开了一个文件。

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

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

Traceback (most recent **call** **last**):

File "<stdin>", line 1, **in** <**module**>

FileNotFoundError: [Errno 2] **No** such file **or** directory: '/Users/michael/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（也可以读取指定大小的内容），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，需要保证无论是否出错都能正确地关闭文件，我们有两种方式实现这个目的

1. 使用try ... finally来实现：

f = None

**try**:

f = open('/path/to/file', 'r')

print(f.read())

**finally**:

**if** f:

f.close()

2.使用with …as…

**with** open('/path/to/file', 'r') **as** f:

print(f.read())

这和前面的try ... finally是一样的，但是代码更佳简洁，并且不必调用f.close()方法。

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

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

**for** line **in** f.readlines():

print(line.strip()) *# 把末尾的'\n'删掉*

### 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。

### 二进制文件

前面讲的默认都是读取文本文件，并且是UTF-8编码的文本文件。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('/Users/michael/test.jpg', 'rb')

>>> f.read()

b'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' *# 十六进制表示的字节*

### 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('/Users/michael/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('/Users/michael/gbk.txt', 'r', encoding='gbk', errors='ignore')

### 写文件

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

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

>>> f.write('Hello, world!')

>>> f.close()

你可以反复调用write()来写入文件，但是务必要调用f.close()来关闭文件。当我们写文件时，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。所以，还是用with语句来得保险：

**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，请给open()函数传入encoding参数，将字符串自动转换成指定编码。

小结

在Python中，文件读写是通过open()函数打开的文件对象完成的。使用with语句操作文件IO是个好习惯

## StringIO和BytesIO

### StringIO

很多时候，数据读写不一定是文件，也可以在内存中读写。

在内存中读写str可使用StringIO。

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!

getvalue()方法用于获得写入后的str。

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!

### BytesIO

StringIO操作的只能是str，如果要操作二进制数据，就需要使用BytesIO。

BytesIO实现了在内存中读写bytes，我们创建一个BytesIO，然后写入一些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'

请注意，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'

小结

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。

## 操作文件和目录

Python内置的os模块也可以直接调用操作系统提供的接口函数。

打开Python交互式命令行，我们来看看如何使用os模块的基本功能：

>>> **import** os

>>> os.name *# 操作系统类型*

'posix'

如果是posix，说明系统是Linux、Unix或Mac OS X，如果是nt，就是Windows系统。

要获取详细的系统信息，可以调用uname()函数：

>>> os.uname()

posix.uname\_result(sysname='Darwin', nodename='MichaelMacPro.local', release='14.3.0', version='Darwin Kernel Version 14.3.0: Mon Mar 23 11:59:05 PDT 2015; root:xnu-2782.20.48~5/RELEASE\_X86\_64', machine='x86\_64')

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。

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/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/local/mysql/bin', ...})

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'

### 操作文件和目录

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，这一点要注意一下。查看、创建和删除目录可以这么调用：

>>> os.path.abspath('.') # 查看当前目录的绝对路径

'/Users/michael'

>>> os.path.join('/Users/michael', 'testdir') # 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

'/Users/michael/testdir'

>>> os.mkdir('/Users/michael/testdir') # 创建一个目录

>>> os.rmdir('/Users/michael/testdir') # 删掉一个目录

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

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

>>> os.path.split('/Users/michael/testdir/file.txt') #拆出路径中的目录名与文件名

('/Users/michael/testdir', 'file.txt')

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

>>> os.path.splitext('/path/to/file.txt') #拆出文件路径中的后缀名与路径名

('/path/to/file', '.txt')

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

文件操作使用下面的函数。假定当前目录下有一个test.txt文件：

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

>>> os.remove('test.py') # 删掉文件:

但是复制文件的函数居然在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。理论上讲，我们通过上一节的读写文件可以完成文件复制，只不过要多写很多代码。

幸运的是shutil模块提供了copyfile()的函数，你还可以在shutil模块中找到很多实用函数，它们可以看做是os模块的补充。

最后看看如何利用Python的特性来过滤文件。

>>> [x for x in os.listdir('.') if os.path.isdir(x)] #列出当前目录下的所有目录

['.lein', '.local', '.m2', '.npm', '.ssh', '.Trash', '.vim', 'Applications', 'Desktop', ...]

#列出当前目录下所有的.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']

小结

Python的os模块封装了操作系统的目录和文件操作，要注意这些函数有的在os模块中，有的在os.path模块中。

## 序列化

在程序运行的过程中，所有的变量都是在内存中，比如，定义一个dict：

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

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

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

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

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

Python提供了pickle模块来实现序列化。

首先，我们尝试把一个对象序列化并写入文件：

>>> import pickle

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

>>> pickle.dumps(d)

b'\x80\x03}q\x00(X\x03\x00\x00\x00ageq\x01K\x14X\x05\x00\x00\x00scoreq\x02KXX\x04\x00\x00\x00nameq\x03X\x03\x00\x00\x00Bobq\x04u.'

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

>>> f = open('dump.txt', 'wb')

>>> pickle.dump(d, f)

>>> f.close()

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

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

>>> f = open('dump.txt', 'rb')

>>> d = pickle.load(f)

>>> f.close()

>>> d

{'age': 20, 'score': 88, 'name': 'Bob'}

变量的内容又回来了！

当然，这个变量和原来的变量是完全不相干的对象，它们只是内容相同而已。

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

### **JSON**

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

|  |  |
| --- | --- |
| JSON类型 | Python类型 |
| {} | dict |
| [] | list |
| "string" | str |
| 1234.56 | int或float |
| true/false | True/False |
| null | None |

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：

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

>>> **import** json

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

>>> json.dumps(d)

'{"age": 20, "score": 88, "name": "Bob"}'

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

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

>>> json\_str = '{"age": 20, "score": 88, "name": "Bob"}'

>>> json.loads(json\_str)

{'age': 20, 'score': 88, 'name': 'Bob'}

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

JSON进阶

Python的dict对象可以直接序列化为JSON的{}，不过，很多时候，我们更喜欢用class表示对象，比如定义Student类，然后序列化：

import json

class Student(object):

def \_\_init\_\_(self, name, age, score):

self.name = name

self.age = age

self.score = score

s = Student('Bob', 20, 88)

print(json.dumps(s))

运行代码，毫不留情地得到一个TypeError：

Traceback (most recent call last):

...

TypeError: <\_\_main\_\_.Student object at 0x10603cc50> is not JSON serializable

错误的原因是Student对象不是一个可序列化为JSON的对象。

如果连class的实例对象都无法序列化为JSON，这肯定不合理！

别急，我们仔细看看dumps()方法的参数列表，可以发现，除了第一个必须的obj参数外，dumps()方法还提供了一大堆的可选参数：

[https://docs.python.org/3/library/json.html#json.dumps](https://docs.python.org/3/library/json.html" \l "json.dumps" \t "_blank)

这些可选参数就是让我们来定制JSON序列化。前面的代码之所以无法把Student类实例序列化为JSON，是因为默认情况下，dumps()方法不知道如何将Student实例变为一个JSON的{}对象。

可选参数default就是把任意一个对象变成一个可序列为JSON的对象，我们只需要为Student专门写一个转换函数，再把函数传进去即可：

def student2dict(std):

return {

'name': std.name,

'age': std.age,

'score': std.score

}

这样，Student实例首先被student2dict()函数转换成dict，然后再被顺利序列化为JSON：

>>> print(json.dumps(s, default=student2dict))

{"age": 20, "name": "Bob", "score": 88}

不过，下次如果遇到一个Teacher类的实例，照样无法序列化为JSON。我们可以偷个懒，把任意class的实例变为dict：

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，如果我们要把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，我们传入的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']) #返回一个Student对象

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。

### **小结**

Python语言特定的序列化模块是pickle，但如果要把序列化搞得更通用、更符合Web标准，就可以使用json模块。

json模块的dumps()和loads()函数是定义得非常好的接口的典范。当我们使用时，只需要传入一个必须的参数。但是，当默认的序列化或反序列机制不满足我们的要求时，我们又可以传入更多的参数来定制序列化或反序列化的规则，既做到了接口简单易用，又做到了充分的扩展性和灵活性。

# 进程和线程

很多同学都听说过，现代操作系统比如Mac OS X，UNIX，Linux，Windows等，都是支持“多任务”的操作系统。

什么叫“多任务”呢？简单地说，就是操作系统可以同时运行多个任务。打个比方，你一边在用浏览器上网，一边在听MP3，一边在用Word赶作业，这就是多任务，至少同时有3个任务正在运行。还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。

现在，多核CPU已经非常普及了，但是，即使过去的单核CPU，也可以执行多任务。由于CPU执行代码都是顺序执行的，那么，单核CPU是怎么执行多任务的呢？

答案就是操作系统轮流让各个任务交替执行，任务1执行0.01秒，切换到任务2，任务2执行0.01秒，再切换到任务3，执行0.01秒……这样反复执行下去。表面上看，每个任务都是交替执行的，但是，由于CPU的执行速度实在是太快了，我们感觉就像所有任务都在同时执行一样。

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

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

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

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

我们前面编写的所有的Python程序，都是执行单任务的进程，也就是只有一个线程。如果我们要同时执行多个任务怎么办？

有两种解决方案：

一种是启动多个进程，每个进程虽然只有一个线程，但多个进程可以一块执行多个任务。

还有一种方法是启动一个进程，在一个进程内启动多个线程，这样，多个线程也可以一块执行多个任务。

当然还有第三种方法，就是启动多个进程，每个进程再启动多个线程，这样同时执行的任务就更多了，当然这种模型更复杂，实际很少采用。

总结一下就是，多任务的实现有3种方式：

* 多进程模式；
* 多线程模式；
* 多进程+多线程模式。

同时执行多个任务通常各个任务之间并不是没有关联的，而是需要相互通信和协调，有时，任务1必须暂停等待任务2完成后才能继续执行，有时，任务3和任务4又不能同时执行，所以，多进程和多线程的程序的复杂度要远远高于我们前面写的单进程单线程的程序。

因为复杂度高，调试困难，所以，不是迫不得已，我们也不想编写多任务。但是，有很多时候，没有多任务还真不行。想想在电脑上看电影，就必须由一个线程播放视频，另一个线程播放音频，否则，单线程实现的话就只能先把视频播放完再播放音频，或者先把音频播放完再播放视频，这显然是不行的。

Python既支持多进程，又支持多线程，我们会讨论如何编写这两种多任务程序。

### **小结**

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

多进程和多线程的程序涉及到同步、数据共享的问题，编写起来更复杂。

## 多进程

要让Python程序实现多进程（multiprocessing），我们先了解操作系统的相关知识。

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.

由于Windows没有fork调用，上面的代码在Windows上无法运行。由于Mac系统是基于BSD（Unix的一种）内核，所以，在Mac下运行是没有问题的，推荐大家用Mac学Python！

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。

multiprocessing

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

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

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

from multiprocessing import Process

import os

# 子进程要执行的代码

def run\_proc(name):

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

if \_\_name\_\_=='\_\_main\_\_':

print('Parent process %s.' % os.getpid())

p = Process(target=run\_proc, args=('test',))

print('Child process will start.')

p.start()

p.join()

print('Child process end.')

执行结果如下：

Parent process 928.

Child Process will start.

Run child process test (929)...

Child Process end.

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

join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

Pool

如果要启动大量的子进程，可以用进程池的方式批量创建子进程：

**from** multiprocessing **import** Pool

import os, time, random

def long\_time\_task(name):

print('Run task %s (%s)...' % (name, os.getpid()))

start = time.time()

time.sleep(random.random() \* 3)

end = time.time()

print('Task %s runs %0.2f seconds.' % (name, (end - start)))

if \_\_name\_\_=='\_\_main\_\_':

print('Parent process %s.' % os.getpid())

p = Pool(4)

for i in range(5):

p.apply\_async(long\_time\_task, args=(i,))

print('Waiting for all subprocesses done...')

p.close()

p.join()

print('All subprocesses done.')

执行结果如下：

Parent process 669.

Waiting for all subprocesses done...

Run task 0 (671)...

Run task 1 (672)...

Run task 2 (673)...

Run task 3 (674)...

Task 2 runs 0.14 seconds.

Run task 4 (673)...

Task 1 runs 0.27 seconds.

Task 3 runs 0.86 seconds.

Task 0 runs 1.41 seconds.

Task 4 runs 1.91 seconds.

All subprocesses done.

代码解读：

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

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

p = Pool(5)

就可以同时跑5个进程。

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

子进程

很多时候，子进程并不是自身，而是一个外部进程。我们创建了子进程后，还需要控制子进程的输入和输出。

subprocess模块可以让我们非常方便地启动一个子进程，然后控制其输入和输出。

下面的例子演示了如何在Python代码中运行命令nslookup www.python.org，这和命令行直接运行的效果是一样的：

import subprocess

print('$ nslookup www.python.org')

r = subprocess.call(['nslookup', 'www.python.org'])

print('Exit code:', r)

运行结果：

$ 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

运行结果如下：

$ 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

进程间通信

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

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

from multiprocessing import Process, Queue

import os, time, random

# 写数据进程执行的代码:

def write(q):

print('Process to write: %s' % os.getpid())

for value in ['A', 'B', 'C']:

print('Put %s to queue...' % value)

q.put(value)

time.sleep(random.random())

# 读数据进程执行的代码:

def read(q):

print('Process to read: %s' % os.getpid())

while True:

value = q.get(True)

print('Get %s from queue.' % value)

if \_\_name\_\_=='\_\_main\_\_':

# 父进程创建Queue，并传给各个子进程：

q = Queue()

pw = Process(target=write, args=(q,))

pr = Process(target=read, args=(q,))

# 启动子进程pw，写入:

pw.start()

# 启动子进程pr，读取:

pr.start()

# 等待pw结束:

pw.join()

# pr进程里是死循环，无法等待其结束，只能强行终止:

pr.terminate()

运行结果如下：

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，使我们不需要关注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。

小结

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。

## 多线程

多任务可以由多进程完成，也可以由一个进程内的多线程完成。

我们前面提到了进程是由若干线程组成的，一个进程至少有一个线程。

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

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

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

import time, threading

# 新线程执行的代码:

def loop():

print('thread %s is running...' % threading.current\_thread().name)

n = 0

while n < 5:

n = n + 1

print('thread %s >>> %s' % (threading.current\_thread().name, n))

time.sleep(1)

print('thread %s ended.' % threading.current\_thread().name)

print('thread %s is running...' % threading.current\_thread().name)

t = threading.Thread(target=loop, name='LoopThread')

t.start()

t.join()

print('thread %s ended.' % threading.current\_thread().name)

执行结果如下：

thread MainThread is running...

thread LoopThread is running...

thread LoopThread >>> 1

thread LoopThread >>> 2

thread LoopThread >>> 3

thread LoopThread >>> 4

thread LoopThread >>> 5

thread LoopThread ended.

thread MainThread ended.

由于任何进程默认就会启动一个线程，我们把该线程称为主线程，主线程又可以启动新的线程，Python的threading模块有个current\_thread()函数，它永远返回当前线程的实例。主线程实例的名字叫MainThread，子线程的名字在创建时指定，我们用LoopThread命名子线程。名字仅仅在打印时用来显示，完全没有其他意义，如果不起名字Python就自动给线程命名为Thread-1，Thread-2……

Lock

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

来看看多个线程同时操作一个变量怎么把内容给改乱了：

import time, threading

# 假定这是你的银行存款:

balance = 0

def change\_it(n):

# 先存后取，结果应该为0:

global balance

balance = balance + n

balance = balance - n

def run\_thread(n):

for i in range(100000):

change\_it(n)

t1 = threading.Thread(target=run\_thread, args=(5,))

t2 = threading.Thread(target=run\_thread, args=(8,))

t1.start()

t2.start()

t1.join()

t2.join()

print(balance)

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

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

balance = balance + n

也分两步：

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

也就是可以看成：

x = balance + n

balance = x

由于x是局部变量，两个线程各自都有自己的x，当代码正常执行时：

初始值 balance = 0

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

t1: balance = x1 # balance = 5

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

t1: balance = x1 # balance = 0

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

t2: balance = x2 # balance = 8

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

t2: balance = x2 # balance = 0

结果 balance = 0

但是t1和t2是交替运行的，如果操作系统以下面的顺序执行t1、t2：

初始值 balance = 0

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

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

t2: balance = x2 # balance = 8

t1: balance = x1 # balance = 5

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

t1: balance = x1 # balance = 0

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

t2: balance = x2 # balance = -8

结果 balance = -8

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

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

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

balance = 0

lock = threading.Lock()

def run\_thread(n):

for i in range(100000):

# 先要获取锁:

lock.acquire()

try:

# 放心地改吧:

change\_it(n)

finally:

# 改完了一定要释放锁:

lock.release()

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

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

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

多核CPU

如果你不幸拥有一个多核CPU，你肯定在想，多核应该可以同时执行多个线程。

如果写一个死循环的话，会出现什么情况呢？

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

我们可以监控到一个死循环线程会100%占用一个CPU。

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

要想把N核CPU的核心全部跑满，就必须启动N个死循环线程。

试试用Python写个死循环：

import threading, multiprocessing

def loop():

x = 0

while True:

x = x ^ 1

for i in range(multiprocessing.cpu\_count()):

t = threading.Thread(target=loop)

t.start()

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

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

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

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

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

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

小结

多线程编程，模型复杂，容易发生冲突，必须用锁加以隔离，同时，又要小心死锁的发生。

Python解释器由于设计时有GIL全局锁，导致了多线程无法利用多核。多线程的并发在Python中就是一个美丽的梦。

## ThreadLocal

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

但是局部变量也有问题，就是在函数调用的时候，传递起来很麻烦：

def process\_student(name):

std = Student(name)

# std是局部变量，但是每个函数都要用它，因此必须传进去：

do\_task\_1(std)

do\_task\_2(std)

def do\_task\_1(std):

do\_subtask\_1(std)

do\_subtask\_2(std)

def do\_task\_2(std):

do\_subtask\_1(std)

do\_subtask\_2(std)

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

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

global\_dict = {}

def std\_thread(name):

std = Student(name)

# 把std放到全局变量global\_dict中：

global\_dict[threading.current\_thread()] = std

do\_task\_1()

do\_task\_2()

def do\_task\_1():

# 不传入std，而是根据当前线程查找：

std = global\_dict[threading.current\_thread()]

...

def do\_task\_2():

# 任何函数都可以查找出当前线程的std变量：

std = global\_dict[threading.current\_thread()]

...

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

有没有更简单的方式？

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

import threading

# 创建全局ThreadLocal对象:

local\_school = threading.local()

def process\_student():

# 获取当前线程关联的student:

std = local\_school.student

print('Hello, %s (in %s)' % (std, threading.current\_thread().name))

def process\_thread(name):

# 绑定ThreadLocal的student:

local\_school.student = name

process\_student()

t1 = threading.Thread(target= process\_thread, args=('Alice',), name='Thread-A')

t2 = threading.Thread(target= process\_thread, args=('Bob',), name='Thread-B')

t1.start()

t2.start()

t1.join()

t2.join()

执行结果：

Hello, Alice (in Thread-A)

Hello, Bob (in Thread-B)

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

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

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

小结

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

## 进程 vs. 线程

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

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

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

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

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

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

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

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

### **线程切换**

无论是多进程还是多线程，只要数量一多，效率肯定上不去，为什么呢？

我们打个比方，假设你不幸正在准备中考，每天晚上需要做语文、数学、英语、物理、化学这5科的作业，每项作业耗时1小时。

如果你先花1小时做语文作业，做完了，再花1小时做数学作业，这样，依次全部做完，一共花5小时，这种方式称为单任务模型，或者批处理任务模型。

假设你打算切换到多任务模型，可以先做1分钟语文，再切换到数学作业，做1分钟，再切换到英语，以此类推，只要切换速度足够快，这种方式就和单核CPU执行多任务是一样的了，以幼儿园小朋友的眼光来看，你就正在同时写5科作业。

但是，切换作业是有代价的，比如从语文切到数学，要先收拾桌子上的语文书本、钢笔（这叫保存现场），然后，打开数学课本、找出圆规直尺（这叫准备新环境），才能开始做数学作业。操作系统在切换进程或者线程时也是一样的，它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。

### **计算密集型 vs. IO密集型**

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

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

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

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web应用。

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。

### **异步IO**

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

现代操作系统对IO操作已经做了巨大的改进，最大的特点就是支持异步IO。如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执行多任务，这种全新的模型称为事件驱动模型，Nginx就是支持异步IO的Web服务器，它在单核CPU上采用单进程模型就可以高效地支持多任务。在多核CPU上，可以运行多个进程（数量与CPU核心数相同），充分利用多核CPU。由于系统总的进程数量十分有限，因此操作系统调度非常高效。用异步IO编程模型来实现多任务是一个主要的趋势。

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

## 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。

举个例子：如果我们已经有一个通过Queue通信的多进程程序在同一台机器上运行，现在，由于处理任务的进程任务繁重，希望把发送任务的进程和处理任务的进程分布到两台机器上。怎么用分布式进程实现？

原有的Queue可以继续使用，但是，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。

我们先看服务进程，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：

// master进程向worker进程发送n，worker进程接收到后计算n\*n，并发回给master进程

# task\_master.py master进程

import random, time, queue

from multiprocessing.managers import BaseManager

# 发送任务的队列:用于从master进程向worker进程发送数据

task\_queue = queue.Queue()

# 接收结果的队列: master进程通过它接收从worker进程发来数据

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.')

请注意，当我们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。

现在，可以试试分布式进程的工作效果了。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\_worker.py进程结束，在task\_master.py进程中会继续打印出结果：

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

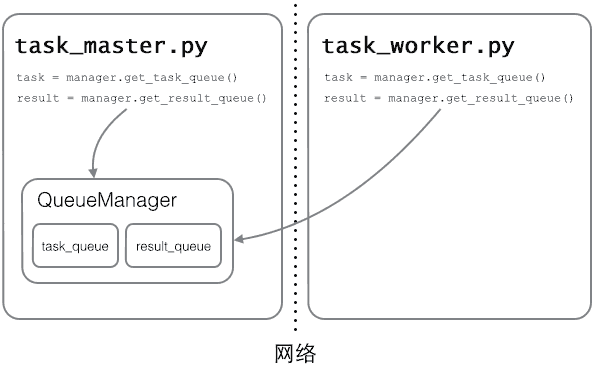
Result: 4219 \* 4219 = 17799961

Result: 339 \* 339 = 114921

Result: 7866 \* 7866 = 61873956

这个简单的Master/Worker模型有什么用？其实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对象存储在哪？注意到task\_worker.py中根本没有创建Queue的代码，所以，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。

小结

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。

# 正则表达式

字符串是编程时涉及到的最多的一种数据结构，对字符串进行操作的需求几乎无处不在。比如判断一个字符串是否是合法的Email地址，

虽然可以编程提取@前后的子串，再分别判断是否是单词和域名，但这样做不但麻烦，而且代码难以复用。

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

所以我们判断一个字符串是否是合法的Email的方法是：

创建一个匹配Email的正则表达式；

用该正则表达式去匹配用户的输入来判断是否合法。

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

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

'00\d'可以匹配'007'，但无法匹配'00A'；

'\d\d\d'可以匹配'010'；

'\w\w\d'可以匹配'py3'；

.可以匹配任意字符，所以：

'py.'可以匹配'pyc'、'pyo'、'py!'等等。

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

来看一个复杂的例子：\d{3}\s+\d{3,8}。

我们来从左到右解读一下：

\d{3}表示匹配3个数字，例如'010'；

\s可以匹配一个空格（也包括Tab等空白符），所以\s+表示至少有一个空格，例如匹配' '，' '等；

\d{3,8}表示3-8个数字，例如'1234567'。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

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

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

### 进阶

要做更精确地匹配，可以用[]表示范围，比如：

[0-9a-zA-Z\\_]可以匹配一个数字、字母或者下划线；

[0-9a-zA-Z\\_]+可以匹配至少由一个数字、字母或者下划线组成的字符串，比如'a100'，'0\_Z'，'Py3000'等等；

[a-zA-Z\\_][0-9a-zA-Z\\_]\*可以匹配由字母或下划线开头，后接任意个由一个数字、字母或者下划线组成的字符串，也就是Python合法的变量；

[a-zA-Z\\_][0-9a-zA-Z\\_]{0, 19}更精确地限制了变量的长度是1-20个字符（前面1个字符+后面最多19个字符）。

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

^表示行的开头，^\d表示必须以数字开头。

$表示行的结束，\d$表示必须以数字结束。

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

### re模块

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

s = 'ABC\\-001' # Python的字符串

# 对应的正则表达式字符串变成：

# 'ABC\-001'

因此我们强烈建议使用Python的r前缀，就不用考虑转义的问题了：

s = r'ABC\-001' # Python的字符串

# 对应的正则表达式字符串不变：

# 'ABC\-001'

先看看如何判断正则表达式是否匹配：

>>> import re

>>> re.match(r'^\d{3}\-\d{3,8}$', '010-12345')

<\_sre.SRE\_Match object; span=(0, 9), match='010-12345'>

>>> re.match(r'^\d{3}\-\d{3,8}$', '010 12345')

>>>

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

test = '用户输入的字符串'

if re.match(r'正则表达式', test):

print('ok')

else:

print('failed')

### 切分字符串

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代码：

>>> 'a b c'.split(' ')

['a', 'b', '', '', 'c']

嗯，无法识别连续的空格，用正则表达式试试：

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

['a', 'b', 'c']

无论多少个空格都可以正常分割。加入,试试：

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

['a', 'b', 'c', 'd']

再加入;试试：

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

['a', 'b', 'c', 'd']

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

### 分组

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

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

>>> m = re.match(r'^(\d{3})-(\d{3,8})$', '010-12345')

>>> m

<\_sre.SRE\_Match object; span=(0, 9), match='010-12345'>

>>> m.group(0)

'010-12345'

>>> m.group(1)

'010'

>>> m.group(2)

'12345'

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

注意到group(0)匹配的串，group(1)、group(2)……表示第1、2、……个捕获的串。

提取子串非常有用。来看一个更凶残的例子：

>>> t = '19:05:30'

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

>>> m.groups()

('19', '05', '30')

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

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

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

### 贪婪匹配

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

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

('102300', '')

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

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

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

('1023', '00')

### 编译

当我们在Python中使用正则表达式时，re模块内部会干两件事情：

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；

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。

小结

正则表达式非常强大，要在短短的一节里讲完是不可能的。要讲清楚正则的所有内容，可以写一本厚厚的书了。如果你经常遇到正则表达式的问题，你可能需要一本正则表达式的参考书。

练习

请尝试写一个验证Email地址的正则表达式。版本一应该可以验证出类似的Email：

someone@gmail.com

bill.gates@microsoft.com

版本二可以验证并提取出带名字的Email地址：

<Tom Paris> tom@voyager.org

# 常用内建模块

## datetime

datetime是Python处理日期和时间的标准库。

### 获取当前日期和时间

我们先看如何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'>**

注意到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。

### 获取指定日期和时间

要指定某个日期和时间，我们直接用参数构造一个datetime：

>>> **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

### datetime转换为timestamp

在计算机中，时间实际上是用数字表示的。我们把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。

你可以认为：

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

可见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）。

把一个datetime类型转换为timestamp只需要简单调用timestamp()方法：

>>> **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

注意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。

### timestamp转换为datetime

要把timestamp转换为datetime，使用datetime提供的fromtimestamp()方法：

>>> **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

### str转换为datetime

很多时候，用户输入的日期和时间是字符串，要处理日期和时间，首先必须把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。详细的说明请参考[Python文档](https://docs.python.org/3/library/datetime.html" \l "strftime-strptime-behavior" \t "_blank)。

注意转换后的datetime是没有时区信息的。

### 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

### 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。

### 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)))

如果系统时区恰好是UTC+8:00，那么上述代码就是正确的，否则，不能强制设置为UTC+8:00时区。

### 时区转换

我们可以先通过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，例如上述bj\_dt到tokyo\_dt的转换。

### 小结

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。

练习

假设你获取了用户输入的日期和时间如2015-1-21 9:01:30，以及一个时区信息如UTC+5:00，均是str，请编写一个函数将其转换为timestamp：

窗体顶端

# -\*- coding:utf-8 -\*-

import re

from datetime import datetime, timezone, timedelta

def to\_timestamp(dt\_str, tz\_str):

cday = datetime.strptime(dt\_str, '%Y-%m-%d %H:%M:%S')

print(cday)

m = re.match(r'\w+([+-]\d+):\*', tz\_str)

hour = int(m.group(1))

print(m.group(1))

x= cday.replace(tzinfo=timezone(timedelta(hours=hour)))

return x.timestamp()

# 测试:

t1 = to\_timestamp('2015-6-1 08:10:30', 'UTC+7:00')

assert t1 == 1433121030.0, t1

t2 = to\_timestamp('2015-5-31 16:10:30', 'UTC-09:00')

assert t2 == 1433121030.0, t2

print('Pass')

窗体底端

## collections

collections是Python内建的一个集合模块，提供了许多有用的集合类。

### namedtuple

我们知道tuple可以表示不变集合，例如，一个点的二维坐标就可以表示成：

>>> p = (1, 2)

但是，看到(1, 2)，很难看出这个tuple是用来表示一个坐标的。

定义一个class又小题大做了，这时，namedtuple就派上了用场：

>>> **from** collections **import** namedtuple

>>> Point = namedtuple('Point', ['x', 'y'])

>>> #Point = namedtuple('Point', ('X', 'Y')) #与上面的语句效果是一样的

>>> 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，并可以用属性而不是索引来引用tuple的某个元素。

这样一来，我们用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。

可以验证创建的Point对象是tuple的一种子类：

>>> isinstance(p, Point)

True

>>> isinstance(p, tuple)

True

类似的，如果要用坐标和半径表示一个圆，也可以用namedtuple定义：

*# namedtuple('名称', [属性list]):*

Circle = namedtuple('Circle', ['x', 'y', 'r'])

### 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。

### 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。

### OrderedDict

使用dict时，Key是无序的。在对dict做迭代时，我们无法确定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): # capacity：容量

# 调用父类OrderedDict的构造函数，下面要用到父类的popitem等方法

super(LastUpdatedOrderedDict, self).\_\_init\_\_()

self.\_capacity = capacity

**def** **\_\_setitem\_\_**(self, key, value):

# containsKey=1时表示key已存在，则执行修改操作

# containsKey=0时表示key不存在，则执行添加操作

containsKey = 1 **if** key **in** self **else** 0

# 当已达最大容量，当新加key不存在时，会运行这段，先删除最先添加的

# 当key存在时，不会运行这段，会运行第2个if进行修改

**if** len(self) - containsKey >= self.\_capacity:

#if (len(self) == self.\_capacity) and (containsKey==0): #与上一条语句是等价的

# popitem移除键值对并返回，last=true时按LIFO顺序返回

# last=false时按FIFO顺序返回

last = self.popitem(last=False)

print('remove:', last)

**if** containsKey:

**del** self[key]

print('set:', (key, value))

**else**:

print('add:', (key, value))

# 调用父类的\_\_setitem\_\_方法写入键值对

OrderedDict.\_\_setitem\_\_(self, key, value)

### Counter

Counter是一个简单的计数器，例如，统计字符出现的个数：

>>> **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。

小结

collections模块提供了一些有用的集合类，可以根据需要选用。

## base64

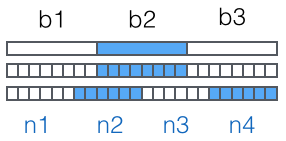
Base64是一种用64个字符来表示任意二进制数据的方法。

用记事本打开exe、jpg、pdf这些文件时，我们都会看到一大堆乱码，因为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的原理很简单，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：



这样我们得到4个数字作为索引，然后查表，获得相应的4个字符，就是编码后的字符串。

所以，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个字节怎么办？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'

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\xb7\x1d\xfb\xef\xff')

b'abcd++//'

>>> base64.urlsafe\_b64encode(b'i\xb7\x1d\xfb\xef\xff')

b'abcd--\_\_'

>>> base64.urlsafe\_b64decode('abcd--\_\_')

b'i\xb7\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。

小结

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。

## struct

准确地讲，Python没有专门处理字节的数据类型。但由于b'str'可以表示字节，所以，字节数组＝二进制str。而在C语言中，我们可以很方便地用struct、union来处理字节，以及字节和int，float的转换。

在Python中，比方说要把一个32位无符号整数变成字节，也就是4个长度的bytes，你得配合位运算符这么写：

>>> n = 10240099

>>> b1 = (n & 0xff000000) >> 24

>>> b2 = (n & 0xff0000) >> 16

>>> b3 = (n & 0xff00) >> 8

>>> b4 = n & 0xff

>>> bs = bytes([b1, b2, b3, b4])

>>> bs

b'\x00\x9c@c'

非常麻烦。如果换成浮点数就无能为力了。

好在Python提供了一个struct模块来解决bytes和其他二进制数据类型的转换。

struct的pack函数把任意数据类型变成bytes: 将数据变成字节流

>>> **import** struct

>>> struct.pack('>I', 10240099)

b'\x00\x9c@c'

pack的第一个参数是处理指令，'>I'的意思是：

>表示字节顺序是big-endian，也就是网络序，I表示4字节无符号整数。

后面的参数个数要和处理指令一致。

unpack把bytes变成相应的数据类型： 🡪将字节流变成数据

>>> struct.unpack('>IH', b'\xf0\xf0\xf0\xf0\x80\x80')

(4042322160, 32896)

根据>IH的说明，后面的bytes依次变为I：4字节无符号整数 和H：2字节无符号整数。

所以，尽管Python不适合编写底层操作字节流的代码，但在对性能要求不高的地方，利用struct就方便多了。

struct模块定义的数据类型可以参考Python官方文档：

[https://docs.python.org/3/library/struct.html#format-characters](https://docs.python.org/3/library/struct.html" \l "format-characters" \t "_blank)

Windows的位图文件（.bmp）是一种非常简单的文件格式，我们来用struct分析一下。

首先找一个bmp文件，没有的话用“画图”画一个。

读入前30个字节来分析：

>>> s = b'\x42\x4d\x38\x8c\x0a\x00\x00\x00\x00\x00\x36\x00\x00\x00\x28\x00\x00\x00\x80\x02\x00\x00\x68\x01\x00\x00\x01\x00\x18\x00'

BMP格式采用小端方式存储数据，文件头的结构按顺序如下：

两个字节：'BM'表示Windows位图，'BA'表示OS/2位图；

一个4字节整数：表示位图大小；

一个4字节整数：保留位，始终为0；

一个4字节整数：实际图像的偏移量；

一个4字节整数：Header的字节数；

一个4字节整数：图像宽度；

一个4字节整数：图像高度；

一个2字节整数：始终为1；

一个2字节整数：颜色数。

所以，组合起来用unpack读取：

>>> struct.unpack('<ccIIIIIIHH', s)

(b'B', b'M', 691256, 0, 54, 40, 640, 360, 1, 24)

结果显示，b'B'、b'M'说明是Windows位图，位图大小为640x360，颜色数为24。

请编写一个bmpinfo.py，可以检查任意文件是否是位图文件，如果是，打印出图片大小和颜色数。

import struct

path = input('请输入图片名:')

print(path)

with open(path, 'rb') as f:

x=f.read(30)

y = struct.unpack('<ccIIIIIIHH', x)

if y[0]+y[1] == b'BM':

print('是位图')

print('位图大小：%d Byte' % y[2])

print('像素(%dx%d)' % (y[6], y[7]))

print ('颜色数:%d' % y[9])

else:

print('不是位图')

## hashlib

### 摘要算法(哈希算法)简介

Python的hashlib提供了常见的摘要算法，如MD5，SHA1等等。

什么是摘要算法呢？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）。

举个例子，你写了一篇文章，内容是一个字符串'how to use python hashlib - by Michael'，并附上这篇文章的摘要是'2d73d4f15c0db7f5ecb321b6a65e5d6d'。如果有人篡改了你的文章，并发表为'how to use python hashlib - by Bob'，你可以一下子指出Bob篡改了你的文章，因为根据'how to use python hashlib - by Bob'计算出的摘要不同于原始文章的摘要。

可见，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。

我们以常见的摘要算法MD5为例，计算出一个字符串的MD5值：

**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())

试试改动一个字母，看看计算的结果是否完全不同。

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。

另一种常见的摘要算法是SHA1，调用SHA1和调用MD5完全类似：

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。

有没有可能两个不同的数据通过某个摘要算法得到了相同的摘要？完全有可能，因为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，比如Bob试图根据你的摘要反推出一篇文章'how to learn hashlib in python - by Bob'，并且这篇文章的摘要恰好和你的文章完全一致，这种情况也并非不可能出现，但是非常非常困难。

摘要算法应用

摘要算法能应用到什么地方？举个常用例子：

任何允许用户登录的网站都会存储用户登录的用户名和口令。如何存储用户名和口令呢？方法是存到数据库表中：

name | password

*--------+----------*

michael | 123456

bob | abc999

alice | alice2008

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：

username | password

*---------+---------------------------------*

michael | e10adc3949ba59abbe56e057f20f883e

bob | 878ef96e86145580c38c87f0410ad153

alice | 99b1c2188db85afee403b1536010c2c9

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。

采用MD5存储口令是否就一定安全呢？也不一定。假设你是一个黑客，已经拿到了存储MD5口令的数据库，如何通过MD5反推用户的明文口令呢？暴力破解费事费力，真正的黑客不会这么干。

考虑这么个情况，很多用户喜欢用123456，888888，password这些简单的口令，于是，黑客可以事先计算出这些常用口令的MD5值，得到一个反推表：

'e10adc3949ba59abbe56e057f20f883e': '123456'

'21218cca77804d2ba1922c33e0151105': '888888'

'5f4dcc3b5aa765d61d8327deb882cf99': 'password'

这样，无需破解，只需要对比数据库的MD5，黑客就获得了使用常用口令的用户账号。

对于用户来讲，当然不要使用过于简单的口令。但是，我们能否在程序设计上对简单口令加强保护呢？

由于常用口令的MD5值很容易被计算出来，所以，要确保存储的用户口令不是那些已经被计算出来的常用口令的MD5，这一方法通过对原始口令加一个复杂字符串来实现，俗称“加盐”：

**def** **calc\_md5**(password):

**return** get\_md5(password + 'the-Salt')

经过Salt处理的MD5口令，只要Salt不被黑客知道，即使用户输入简单口令，也很难通过MD5反推明文口令。

但是如果有两个用户都使用了相同的简单口令比如123456，在数据库中，将存储两条相同的MD5值，这说明这两个用户的口令是一样的。有没有办法让使用相同口令的用户存储不同的MD5呢？

如果假定用户无法修改登录名，就可以通过把登录名作为Salt的一部分来计算MD5，从而实现相同口令的用户也存储不同的MD5。

小结

摘要算法在很多地方都有广泛的应用。要注意摘要算法不是加密算法，不能用于加密（因为无法通过摘要反推明文），只能用于防篡改，但是它的单向计算特性决定了可以在不存储明文口令的情况下验证用户口令。

## 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。

### count()

**count()**会创建一个无限的迭代器

>>> **import** itertools

>>> natuals = itertools.count(1)

>>> **for** n **in** natuals:

... print(n)

...

1

2

3

...

### cycle()

**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'

...

### 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，但是通常我们会通过takewhile()等函数根据条件判断来截取出一个有限的序列：

>>> natuals = itertools.count(1)

>>> ns = itertools.takewhile(**lambda** x: x <= 10, natuals)

>>> list(ns)

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

itertools提供的几个迭代器操作函数更加有用：

### 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'*

### 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。如果我们要忽略大小写分组，就可以让元素'A'和'a'都返回相同的key：

>>> **for** key, group **in** itertools.groupby('AaaBBbcCAAa', **lambda** c: c.upper()):

... print(key, list(group))

...

A ['A', 'a', 'a']

B ['B', 'B', 'b']

C ['c', 'C']

A ['A', 'A', 'a']

**小结**

itertools模块提供的全部是处理迭代功能的函数，它们的返回值不是list，而是Iterator，只有用for循环迭代的时候才真正计算。

## contextlib

在Python中，读写文件这样的资源要特别注意，必须在使用完毕后正确关闭它们。正确关闭文件资源的一个方法是使用try...finally：

**try**:

f = open('/path/to/file', 'r')

f.read()

**finally**:

**if** f:

f.close()

写try...finally非常繁琐。Python的with语句允许我们非常方便地使用资源，而不必担心资源没有关闭，所以上面的代码可以简化为：

**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。实际上，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。例如，下面的class实现了这两个方法：

**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)

这样我们就可以把自己写的资源对象用于with语句：

**with** Query('Bob') **as** q:

q.query()

输出

Begin

Query info about Bob...

End

### @contextmanager

编写\_\_enter\_\_和\_\_exit\_\_仍然很繁琐，因此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语句通过with ... as var把变量输出到var，然后，with语句就可以正常地工作了：

**with** create\_query('Bob') **as** q:

q.query()

很多时候，我们希望在某段代码执行前后自动执行特定代码，也可以用@contextmanager实现。例如：

@contextmanager

**def** **tag**(name):

print("<%s>" % name)

**yield**

print("</%s>" % name)

**with** tag("h1"):

print("hello")

print("world")

上述代码执行结果为：

<h1>

hello

world

</h1>

代码的执行顺序是：

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

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

### @closing

如果一个对象没有实现上下文，我们就不能把它用于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，便于我们编写更简洁的代码。

## XML

XML虽然比JSON复杂，在Web中应用也不如以前多了，不过仍有很多地方在用，所以，有必要了解如何操作XML。

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，缺点是我们需要自己处理事件。

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，通常我们关心的事件是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：举个例子，

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

会产生3个事件：

1. start\_element事件，在读取<a href="/">时；
2. char\_data事件，在读取python时；
3. end\_element事件，在读取</a>时。

用代码实验一下：

**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。

除了解析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)

如果要生成复杂的XML呢？建议你不要用XML，改成JSON。

**小结**

解析XML时，注意找出自己感兴趣的节点，响应事件时，把节点数据保存起来。解析完毕后，就可以处理数据。

请利用SAX编写程序解析Yahoo的XML格式的天气预报，获取当天和第二天的天气：

[https://query.yahooapis.com/v1/public/yql?q=select%20\*%20from%20weather.forecast%20where%20woeid%20in%20(select%20woeid%20from%20geo.places(1)%20where%20text%3D%22beijing%2C%20china%22)%20and%20u%3D%27c%27%20&format=xml&env=store%3A%2F%2Fdatatables.org%2Falltableswithkeysw%2C%20scotland%22)%20and%20u%3D%27c%27%20&format=xml&env=store%3A%2F%2Fdatatables.org%2Falltableswithkeys](https://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20weather.forecast%20where%20woeid%20in%20(select%20woeid%20from%20geo.places(1)%20where%20text%3D%22beijing%2C%20china%22)%20and%20u%3D%27c%27%20&format=xml&env=store%3A%2F%2Fdatatables.org%2Falltableswithkeysw%2C%20scotland%22)%20and%20u%3D%27c%27%20&format=xml&env=store%3A%2F%2Fdatatables.org%2Falltableswithkeys) 网站获取XML。

data = r'''

<query xmlns:yahoo="http://www.yahooapis.com/v1/base.rng" yahoo:count="1" yahoo:created="2016-11-27T07:29:17Z" yahoo:lang="zh-CN">

<results>

<channel>

<yweather:units xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" distance="km" pressure="mb" speed="km/h" temperature="C"/>

<title>Yahoo! Weather - Beijing, Beijing, CN</title>

<link>...</link>

<description>Yahoo! Weather for Beijing, Beijing, CN</description>

<language>en-us</language>

<lastBuildDate>Sun, 27 Nov 2016 03:29 PM CST</lastBuildDate>

<ttl>60</ttl>

<yweather:location xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" city="Beijing" country="China" region=" Beijing"/>

<yweather:wind xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" chill="39" direction="325" speed="28.97"/>

<yweather:atmosphere xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" humidity="18" pressure="34676.63" rising="0" visibility="25.91"/>

<yweather:astronomy xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" sunrise="7:14 am" sunset="4:51 pm"/>

<image>...</image>

<item>

<title>...</title>

<geo:lat xmlns:geo="http://www.w3.org/2003/01/geo/wgs84\_pos#">39.90601</geo:lat>

<geo:long xmlns:geo="http://www.w3.org/2003/01/geo/wgs84\_pos#">116.387909</geo:long>

<link>...</link>

<pubDate>Sun, 27 Nov 2016 02:00 PM CST</pubDate>

<yweather:condition xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" code="32" date="Sun, 27 Nov 2016 02:00 PM CST" temp="7" text="Sunny"/>

<yweather:forecast xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" code="32" date="27 Nov 2016" day="Sun" high="6" low="-2" text="Sunny"/>

<yweather:forecast xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" code="32" date="28 Nov 2016" day="Mon" high="6" low="-4" text="Sunny"/>

<yweather:forecast xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" code="28" date="29 Nov 2016" day="Tue" high="3" low="-3" text="Mostly Cloudy"/>

<yweather:forecast xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" code="32" date="30 Nov 2016" day="Wed" high="6" low="-2" text="Sunny"/>

<yweather:forecast xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" code="32" date="01 Dec 2016" day="Thu" high="8" low="-1" text="Sunny"/>

<yweather:forecast xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" code="32" date="02 Dec 2016" day="Fri" high="7" low="-3" text="Sunny"/>

<yweather:forecast xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" code="34" date="03 Dec 2016" day="Sat" high="7" low="-2" text="Mostly Sunny"/>

<yweather:forecast xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" code="30" date="04 Dec 2016" day="Sun" high="6" low="-3" text="Partly Cloudy"/>

<yweather:forecast xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" code="34" date="05 Dec 2016" day="Mon" high="6" low="-3" text="Mostly Sunny"/>

<yweather:forecast xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0" code="34" date="06 Dec 2016" day="Tue" high="5" low="-4" text="Mostly Sunny"/>

<description>...</description>

<guid isPermaLink="false"/>

</item>

</channel>

</results>

</query>

<!-- total: 24 -->

<!--...-->

'''

from xml.parsers.expat import ParserCreate

class WeatherSaxHandler(object):

def \_\_init\_\_(self):

self.result = {'yweather:forecast': {}}

self.count = 0

def start\_element(self, name, attrs):

if name == 'yweather:location':

self.result['yweather:location'] = attrs

if name == 'yweather:forecast':

self.count += 1

if self.count <= 2:

self.result['yweather:forecast'][str(self.count)] = attrs

else:

pass

def parse\_weather(xml):

handler = WeatherSaxHandler()

parser = ParserCreate()

#parser.returns\_unicode = False

parser.StartElementHandler = handler.start\_element

parser.Parse(xml)

weather = {'today': {}, 'tomorrow': {}}

weather['city'] = handler.result['yweather:location']['city']

weather['country'] = handler.result['yweather:location']['country']

weather['today']['text'] = handler.result['yweather:forecast']['1']['text']

weather['today']['low'] = handler.result['yweather:forecast']['1']['low']

weather['today']['high'] = handler.result['yweather:forecast']['1']['high']

weather['tomorrow']['text'] = handler.result['yweather:forecast']['2']['text']

weather['tomorrow']['low'] = handler.result['yweather:forecast']['2']['low']

weather['tomorrow']['high'] = handler.result['yweather:forecast']['2']['high']

return weather

weather = parse\_weather(data)

print ('Weather:')

print ('city: %s' % weather['city'])

print ('country: %s' % weather['country'])

print ('today: {%s, low: %s, high: %s}' % (weather['today']['text'], weather['today']['low'], weather['today']['high']))

print ('tomorrow: {%s, low: %s, high: %s}' % (weather['tomorrow']['text'], weather['tomorrow']['low'], weather['tomorrow']['high']))

看一下XML的结构网址： [http://www.w3school.com.cn/xml/index.asp](http://www.w3school.com.cn/xml/index.asp" \t "_blank)

## HTMLParser

如果我们要编写一个搜索引擎，第一步是用爬虫把目标网站的页面抓下来，第二步就是解析该HTML页面，看看里面的内容到底是新闻、图片还是视频。

假设第一步已经完成了，第二步应该如何解析HTML呢？

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。

好在Python提供了HTMLParser来非常方便地解析HTML，只需简单几行代码：

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&nbsp;，一种是数字表示的&#1234;，这两种字符都可以通过Parser解析出来。

小结

利用HTMLParser，可以把网页中的文本、图像等解析出来。

## urllib

urllib提供了一系列用于操作URL的功能。

### 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URLhttps://api.douban.com/v2/book/2129650进行抓取，并返回响应：

from urllib import request

req = request.Request('https://api.douban.com/v2/book/2129650')

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'))

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: pidl1

Data: {"rating":{"max":10,"numRaters":16,"average":"7.4","min":0},"subtitle":"","author":["廖雪峰编著"],"pubdate":"2007-6","tags":[{"count":20,"name":"spring","title":"spring"}...}

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/10A5376e Safari/8536.25')

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" />

...

### 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。

我们模拟一个微博登录，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 Mobile/10A5376e Safari/8536.25')

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，我们获得的响应如下：

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，我们获得的响应如下：

...

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

Handler

如果还需要更复杂的控制，比如通过一个Proxy去访问网站，我们需要利用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

**小结**

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。

# 常用第三方模块

除了内建的模块外，Python还有大量的第三方模块。

基本上，所有的第三方模块都会在[PyPI - the Python Package Index](https://pypi.python.org/" \t "_blank)上注册，只要找到对应的模块名字，即可用pip安装。

本章介绍常用的第三方模块。

## PIL

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](https://github.com/python-pillow/Pillow" \t "_blank)，支持最新Python 3.x，又加入了许多新特性，因此，我们可以直接安装使用Pillow。

安装Pillow

在命令行下直接通过pip安装：

$ pip install pillow

如果遇到Permission denied安装失败，请加上sudo重试。

**操作图像**

来看看最常见的图像缩放操作，只需三四行代码：

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')

效果如下：



PIL的ImageDraw提供了一系列绘图方法，让我们可以直接绘图。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')

我们用随机颜色填充背景，再画上文字，最后对图像进行模糊，得到验证码图片如下：

验证码

如果运行的时候报错：

IOError: cannot open resource

这是因为PIL无法定位到字体文件的位置，可以根据操作系统提供绝对路径，比如：

'/Library/Fonts/Arial.ttf'

要详细了解PIL的强大功能，请请参考Pillow官方文档：

[https://pillow.readthedocs.org/](https://pillow.readthedocs.org/" \t "_blank)

小结

PIL提供了操作图像的强大功能，可以通过简单的代码完成复杂的图像处理。

# virtualenv

在开发Python应用程序的时候，系统安装的Python3只有一个版本：3.4。所有第三方的包都会被pip安装到Python3的site-packages目录下。

如果我们要同时开发多个应用程序，那这些应用程序都会共用一个Python，就是安装在系统的Python 3。如果应用A需要jinja 2.7，而应用B需要jinja 2.6怎么办？

这种情况下，每个应用可能需要各自拥有一套“独立”的Python运行环境。virtualenv就是用来为一个应用创建一套“隔离”的Python运行环境。

首先，我们用pip安装virtualenv：

$ pip3 install virtualenv

然后，假定我们要开发一个新的项目，需要一套独立的Python运行环境，可以这么做：

第一步，创建目录：

Mac:~ michael$ mkdir myproject

Mac:~ michael$ cd myproject/

Mac:myproject michael$

第二步，创建一个独立的Python运行环境，命名为venv：

Mac:myproject michael$ virtualenv --no-site-packages venv

Using base prefix '/usr/local/.../Python.framework/Versions/3.4'

New python executable **in** venv/bin/python3.4

Also creating executable **in** venv/bin/python

Installing setuptools, pip, wheel...done.

命令virtualenv就可以创建一个独立的Python运行环境，我们还加上了参数--no-site-packages，这样，已经安装到系统Python环境中的所有第三方包都不会复制过来，这样，我们就得到了一个不带任何第三方包的“干净”的Python运行环境。

新建的Python环境被放到当前目录下的venv目录。有了venv这个Python环境，可以用source进入该环境：

Mac:myproject michael$ source venv/bin/activate

(venv)Mac:myproject michael$

注意到命令提示符变了，有个(venv)前缀，表示当前环境是一个名为venv的Python环境。

下面正常安装各种第三方包，并运行python命令：

(venv)Mac:myproject michael$ pip install jinja2

...

Successfully installed jinja2-2.7.3 markupsafe-0.23

(venv)Mac:myproject michael$ python myapp.py

...

在venv环境下，用pip安装的包都被安装到venv这个环境下，系统Python环境不受任何影响。也就是说，venv环境是专门针对myproject这个应用创建的。

退出当前的venv环境，使用deactivate命令：

(venv)Mac:myproject michael$ deactivate

Mac:myproject michael$

此时就回到了正常的环境，现在pip或python均是在系统Python环境下执行。

完全可以针对每个应用创建独立的Python运行环境，这样就可以对每个应用的Python环境进行隔离。

virtualenv是如何创建“独立”的Python运行环境的呢？原理很简单，就是把系统Python复制一份到virtualenv的环境，用命令source venv/bin/activate进入一个virtualenv环境时，virtualenv会修改相关环境变量，让命令python和pip均指向当前的virtualenv环境。

小结

virtualenv为应用提供了隔离的Python运行环境，解决了不同应用间多版本的冲突问题。

# 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。

但是Python自带的库是支持Tk的Tkinter，使用Tkinter，无需安装任何包，就可以直接使用。本章简单介绍如何使用Tkinter进行GUI编程。

## Tkinter

我们来梳理一下概念：

我们编写的Python代码会调用内置的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。

所以，我们的代码只需要调用Tkinter提供的接口就可以了。

第一个GUI程序

使用Tkinter十分简单，我们来编写一个GUI版本的“Hello, world!”。

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() #pack()方法把Widget加入到父容器中，并实现布局

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，我们创建一个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。

输入文本

我们再对这个GUI程序改进一下，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() # pack()方法把Widget加入到父容器中，并实现布局

self.createWidgets()

**def** **createWidgets**(self):

self.nameInput = Entry(self)

self.nameInput.pack()

#创建一个Button，当Button被点击时，触发self.hello ()。

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:*

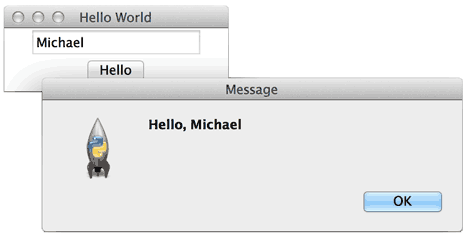
app.master.title('Hello World')

*# 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：



小结

Python内置的Tkinter可以满足基本的GUI程序的要求，如果是非常复杂的GUI程序，建议用操作系统原生支持的语言和库来编写。

# 网络编程

网络通信是两台计算机上的两个进程之间的通信。

## TCP/IP简介

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

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

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

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。

## TCP编程

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

### 客户端

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

举个例子，当我们在浏览器中访问新浪时，我们自己的计算机就是客户端，浏览器会主动向新浪的服务器发起连接。如果一切顺利，新浪的服务器接受了我们的连接，一个TCP连接就建立起来的，后面的通信就是发送网页内容了。

所以，我们要创建一个基于TCP连接的Socket，可以这样做：

# 导入socket库*:*

**import** socket

# 创建一个socket:

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) #创建一个基于IPv4的TCP流式套接字

# 建立连接:

s.connect(('www.sina.com.cn', 80))

创建Socket时，AF\_INET指定使用IPv4协议，如果要用更先进的IPv6，就指定为AF\_INET6。SOCK\_STREAM指定使用面向流的TCP协议，这样，一个Socket对象就创建成功，但是还没有建立连接。

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

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

因此，我们连接新浪服务器的代码如下：s.connect(('www.sina.com.cn', 80))

注意参数是一个tuple，包含地址和端口号。

建立TCP连接后，我们就可以向新浪服务器发送请求，要求返回首页的内容：

# 发送数据:

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

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

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

# 接收数据:

buffer = []

**while** **True**:

# 每次最多接收1k字节:

d = s.recv(1024)

**if** d:

buffer.append(d)

**else**:

**break**

data = b''.join(buffer)

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

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

# 关闭连接:

s.close()

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

header, html = data.split(b'\r\n\r\n', 1)

print(header.decode('utf-8'))

# 把接收的数据写入文件:

**with** open('sina.html', 'wb') **as** f:

f.write(html)

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

### 服务器

和客户端编程相比，服务器编程就要复杂一些。

服务器进程首先要绑定一个端口并监听来自其他客户端的连接。如果某个客户端连接过来了，服务器就与该客户端建立Socket连接，随后的通信就靠这个Socket连接了。所以，服务器会打开固定端口（比如80）监听，每来一个客户端连接，就创建该Socket连接。由于服务器会有大量来自客户端的连接，所以，服务器要能够区分一个Socket连接是和哪个客户端绑定的。一个Socket依赖4项：服务器地址、服务器端口、客户端地址、客户端端口来唯一确定一个Socket。

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

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

import socket, threading,time

首先，创建一个基于IPv4和TCP协议的Socket：

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

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

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

# 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：

import socket, threading,time

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:

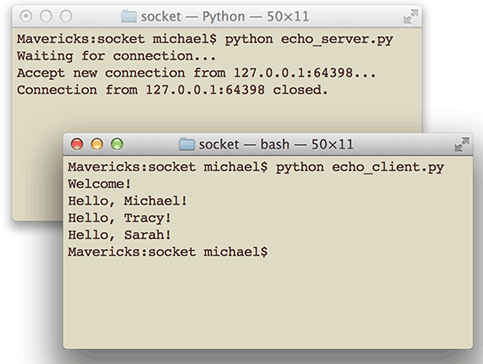
s.send(data)

print(s.recv(1024).decode('utf-8'))

s.send(b'exit')

s.close()

我们需要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。

**小结**

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。

## 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。

我们来看看如何通过UDP协议传输数据。和TCP类似，使用UDP的通信双方也分为客户端和服务器。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))

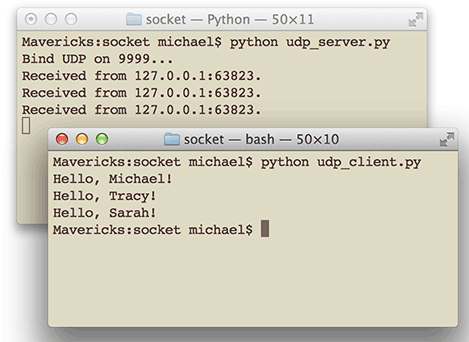
# 接收数据:

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：



**小结**

UDP的使用与TCP类似，但是不需要建立连接。此外，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。

# 电子邮件

Email的历史比Web还要久远，直到现在，Email也是互联网上应用非常广泛的服务。

几乎所有的编程语言都支持发送和接收电子邮件，但是，先等等，在我们开始编写代码之前，有必要搞清楚电子邮件是如何在互联网上运作的。

我们来看看传统邮件是如何运作的。假设你现在在北京，要给一个香港的朋友发一封信，怎么做呢？

首先你得写好信，装进信封，写上地址，贴上邮票，然后就近找个邮局，把信仍进去。

信件会从就近的小邮局转运到大邮局，再从大邮局往别的城市发，比如先发到天津，再走海运到达香港，也可能走京九线到香港，但是你不用关心具体路线，你只需要知道一件事，就是信件走得很慢，至少要几天时间。

信件到达香港的某个邮局，也不会直接送到朋友的家里，因为邮局的叔叔是很聪明的，他怕你的朋友不在家，一趟一趟地白跑，所以，信件会投递到你的朋友的邮箱里，邮箱可能在公寓的一层，或者家门口，直到你的朋友回家的时候检查邮箱，发现信件后，就可以取到邮件了。

电子邮件的流程基本上也是按上面的方式运作的，只不过速度不是按天算，而是按秒算。

现在我们回到电子邮件，假设我们自己的电子邮件地址是me@163.com，对方的电子邮件地址是friend@sina.com（注意地址都是虚构的哈），现在我们用Outlook或者Foxmail之类的软件写好邮件，填上对方的Email地址，点“发送”，电子邮件就发出去了。这些电子邮件软件被称为**MUA**：Mail User Agent——邮件用户代理。

Email从MUA发出去，不是直接到达对方电脑，而是发到**MTA**：Mail Transfer Agent——邮件传输代理，就是那些Email服务提供商，比如网易、新浪等等。由于我们自己的电子邮件是163.com，所以，Email首先被投递到网易提供的MTA，再由网易的MTA发到对方服务商，也就是新浪的MTA。这个过程中间可能还会经过别的MTA，但是我们不关心具体路线，我们只关心速度。

Email到达新浪的MTA后，由于对方使用的是@sina.com的邮箱，因此，新浪的MTA会把Email投递到邮件的最终目的地**MDA**：Mail Delivery Agent——邮件投递代理。Email到达MDA后，就静静地躺在新浪的某个服务器上，存放在某个文件或特殊的数据库里，我们将这个长期保存邮件的地方称之为电子邮箱。

同普通邮件类似，Email不会直接到达对方的电脑，因为对方电脑不一定开机，开机也不一定联网。对方要取到邮件，必须通过MUA从MDA上把邮件取到自己的电脑上。

所以，一封电子邮件的旅程就是：

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

有了上述基本概念，要编写程序来发送和接收邮件，本质上就是：

1. 编写MUA把邮件发到MTA；
2. 编写MUA从MDA上收邮件。

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

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

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

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

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

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

## SMTP发送邮件

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

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

首先，我们来构造一个最简单的纯文本邮件：

**from** email.mime.text **import** MIMEText

msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')

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

然后，通过SMTP发出去：

*# 输入Email地址和口令:*

from\_addr = input('From: ')

password = input('Password: ')

*# 输入收件人地址:*

to\_addr = input('To: ')

*# 输入SMTP服务器地址:*

smtp\_server = input('SMTP server: ')

**import** smtplib

server = smtplib.SMTP(smtp\_server, 25) *# SMTP协议默认端口是25*

server.set\_debuglevel(1)

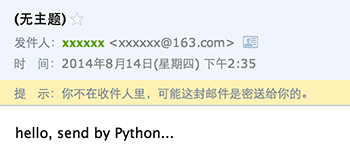
server.login(from\_addr, password)

server.sendmail(from\_addr, [to\_addr], msg.as\_string())

server.quit()

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

如果一切顺利，就可以在收件人信箱中收到我们刚发送的Email：



仔细观察，发现如下问题：

1. 邮件没有主题；
2. 收件人的名字没有显示为友好的名字，比如Mr Green <green@example.com>；
3. 明明收到了邮件，却提示不在收件人中。

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

**from** email **import** encoders

**from** email.header **import** Header

**from** email.mime.text **import** MIMEText

**from** email.utils **import** parseaddr, formataddr

**import** smtplib

**def** **\_format\_addr**(s):

name, addr = parseaddr(s)

**return** formataddr((Header(name, 'utf-8').encode(), addr))

from\_addr = input('From: ')

password = input('Password: ')

to\_addr = input('To: ')

smtp\_server = input('SMTP server: ')

msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')

msg['From'] = \_format\_addr('Python爱好者 <%s>' % from\_addr)

msg['To'] = \_format\_addr('管理员 <%s>' % to\_addr)

msg['Subject'] = Header('来自SMTP的问候……', 'utf-8').encode()

server = smtplib.SMTP(smtp\_server, 25)

server.set\_debuglevel(1)

server.login(from\_addr, password)

server.sendmail(from\_addr, [to\_addr], msg.as\_string())

server.quit()

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

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

再发送一遍邮件，就可以在收件人邮箱中看到正确的标题、发件人和收件人：



你看到的收件人的名字很可能不是我们传入的管理员，因为很多邮件服务商在显示邮件时，会把收件人名字自动替换为用户注册的名字，但是其他收件人名字的显示不受影响。

如果我们查看Email的原始内容，可以看到如下经过编码的邮件头：

From: =?utf-8?b?UHl0aG9u54ix5aW96ICF?= <xxxxxx@163.com>

To: =?utf-8?b?566h55CG5ZGY?= <xxxxxx@qq.com>

Subject: =?utf-8?b?5p2l6IeqU01UUOeahOmXruWAmeKApuKApg==?=

这就是经过Header对象编码的文本，包含utf-8编码信息和Base64编码的文本。如果我们自己来手动构造这样的编码文本，显然比较复杂。

发送HTML邮件

如果我们要发送HTML邮件，而不是普通的纯文本文件怎么办？方法很简单，在构造MIMEText对象时，把HTML字符串传进去，再把第二个参数由plain变为html就可以了：

msg = MIMEText('<html><body><h1>Hello</h1>' +

'<p>send by <a href="http://www.python.org">Python</a>...</p>' +

'</body></html>', 'html', 'utf-8')

再发送一遍邮件，你将看到以HTML显示的邮件：



**发送附件**

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

*# 邮件对象:*

msg = MIMEMultipart()

msg['From'] = \_format\_addr('Python爱好者 <%s>' % from\_addr)

msg['To'] = \_format\_addr('管理员 <%s>' % to\_addr)

msg['Subject'] = Header('来自SMTP的问候……', 'utf-8').encode()

*# 邮件正文是MIMEText:*

msg.attach(MIMEText('send with file...', 'plain', 'utf-8'))

*# 添加附件就是加上一个MIMEBase，从本地读取一个图片:*

**with** open('/Users/michael/Downloads/test.png', 'rb') **as** f:

*# 设置附件的MIME和文件名，这里是png类型:*

mime = MIMEBase('image', 'png', filename='test.png')

*# 加上必要的头信息:*

mime.add\_header('Content-Disposition', 'attachment', filename='test.png')

mime.add\_header('Content-ID', '<0>')

mime.add\_header('X-Attachment-Id', '0')

*# 把附件的内容读进来:*

mime.set\_payload(f.read())

*# 用Base64编码:*

encoders.encode\_base64(mime)

*# 添加到MIMEMultipart:*

msg.attach(mime)

然后，按正常发送流程把msg（注意类型已变为MIMEMultipart）发送出去，就可以收到如下带附件的邮件：



**发送图片**

如果要把一个图片嵌入到邮件正文中怎么做？直接在HTML邮件中链接图片地址行不行？答案是，大部分邮件服务商都会自动屏蔽带有外链的图片，因为不知道这些链接是否指向恶意网站。

要把图片嵌入到邮件正文中，我们只需按照发送附件的方式，先把邮件作为附件添加进去，然后，在HTML中通过引用src="cid:0"就可以把附件作为图片嵌入了。如果有多个图片，给它们依次编号，然后引用不同的cid:x即可。

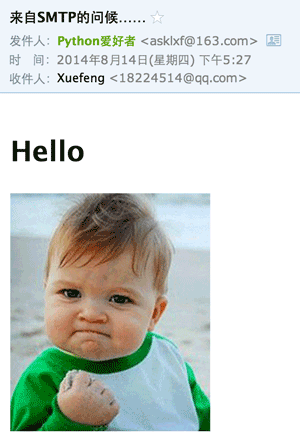
把上面代码加入MIMEMultipart的MIMEText从plain改为html，然后在适当的位置引用图片：

msg.attach(MIMEText('<html><body><h1>Hello</h1>' +

'<p><img src="cid:0"></p>' +

'</body></html>', 'html', 'utf-8'))

再次发送，就可以看到图片直接嵌入到邮件正文的效果：



**同时支持HTML和Plain格式**

如果我们发送HTML邮件，收件人通过浏览器或者Outlook之类的软件是可以正常浏览邮件内容的，但是，如果收件人使用的设备太古老，查看不了HTML邮件怎么办？

办法是在发送HTML的同时再附加一个纯文本，如果收件人无法查看HTML格式的邮件，就可以自动降级查看纯文本邮件。

利用MIMEMultipart就可以组合一个HTML和Plain，要注意指定subtype是alternative：

msg = MIMEMultipart('alternative')

msg['From'] = ...

msg['To'] = ...

msg['Subject'] = ...

msg.attach(MIMEText('hello', 'plain', 'utf-8'))

msg.attach(MIMEText('<html><body><h1>Hello</h1></body></html>', 'html', 'utf-8'))

# 正常发送msg对象...

**加密SMTP**

使用标准的25端口连接SMTP服务器时，使用的是明文传输，发送邮件的整个过程可能会被窃听。要更安全地发送邮件，可以加密SMTP会话，实际上就是先创建SSL安全连接，然后再使用SMTP协议发送邮件。

某些邮件服务商，例如Gmail，提供的SMTP服务必须要加密传输。我们来看看如何通过Gmail提供的安全SMTP发送邮件。

必须知道，Gmail的SMTP端口是587，因此，修改代码如下：

smtp\_server = 'smtp.gmail.com'

smtp\_port = 587

server = smtplib.SMTP(smtp\_server, smtp\_port)

server.starttls()

*# 剩下的代码和前面的一模一样:*

server.set\_debuglevel(1)

...

只需要在创建SMTP对象后，立刻调用starttls()方法，就创建了安全连接。后面的代码和前面的发送邮件代码完全一样。

如果因为网络问题无法连接Gmail的SMTP服务器，请相信我们的代码是没有问题的，你需要对你的网络设置做必要的调整。

**小结**

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

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

Message

+- MIMEBase

+- MIMEMultipart

+- MIMENonMultipart

+- MIMEMessage

+- MIMEText

+- MIMEImage

这种嵌套关系就可以构造出任意复杂的邮件。你可以通过[email.mime文档](https://docs.python.org/3/library/email.mime.html" \t "_blank)查看它们所在的包以及详细的用法。

## POP3收取邮件

SMTP用于发送邮件，如果要收取邮件呢？

收取邮件就是编写一个**MUA**作为客户端，从**MDA**把邮件获取到用户的电脑或者手机上。收取邮件最常用的协议是**POP**协议，目前版本号是3，俗称**POP3**。

Python内置一个poplib模块，实现了POP3协议，可以直接用来收邮件。

注意到POP3协议收取的不是一个已经可以阅读的邮件本身，而是邮件的原始文本，这和SMTP协议很像，SMTP发送的也是经过编码后的一大段文本。

要把POP3收取的文本变成可以阅读的邮件，还需要用email模块提供的各种类来解析原始文本，变成可阅读的邮件对象。

所以，收取邮件分两步：

第一步：用poplib把邮件的原始文本下载到本地；

第二部：用email解析原始文本，还原为邮件对象。

### 通过POP3下载邮件

POP3协议本身很简单，以下面的代码为例，我们来获取最新的一封邮件内容：

import poplib

*# 输入邮件地址, 口令和POP3服务器地址:*

email = input('Email: ')

password = input('Password: ')

pop3\_server = input('POP3 server: ')

*# 连接到POP3服务器:*

server = poplib.POP3(pop3\_server)

*# 可以打开或关闭调试信息:*

server.set\_debuglevel(1)

*# 可选:打印POP3服务器的欢迎文字:*

**print**(server.getwelcome().decode('utf-8'))

*# 身份认证:*

server.user(email)

server.pass\_(password)

*# stat()返回邮件数量和占用空间:*

**print**('Messages: %s. Size: %s' % server.stat())

*# list()返回所有邮件的编号:*

resp, mails, octets = server.**list**()

*# 可以查看返回的列表类似[b'1 82923', b'2 2184', ...]*

**print**(mails)

*# 获取最新一封邮件, 注意索引号从1开始:*

index = len(mails)

resp, lines, octets = server.retr(index)

*# lines存储了邮件的原始文本的每一行,*

*# 可以获得整个邮件的原始文本:*

msg\_content = b'\r\n'.join(lines).decode('utf-8')

*# 稍后解析出邮件:*

msg = Parser().parsestr(msg\_content)

*# 可以根据邮件索引号直接从服务器删除邮件:*

*# server.dele(index)*

*# 关闭连接:*

server.quit()

用POP3获取邮件其实很简单，要获取所有邮件，只需要循环使用retr()把每一封邮件内容拿到即可。真正麻烦的是把邮件的原始内容解析为可以阅读的邮件对象。

### 解析邮件

解析邮件的过程和上一节构造邮件正好相反，因此，先导入必要的模块：

**from** email.parser **import** Parser

**from** email.header **import** decode\_header

**from** email.utils **import** parseaddr

**import** poplib

只需要一行代码就可以把邮件内容解析为Message对象：

msg = Parser().parsestr(msg\_content)

但是这个Message对象本身可能是一个MIMEMultipart对象，即包含嵌套的其他MIMEBase对象，嵌套可能还不止一层。

所以我们要递归地打印出Message对象的层次结构：

*# indent用于缩进显示:*

**def** **print\_info**(msg, indent=0):

**if** indent == 0:

**for** header **in** ['From', 'To', 'Subject']:

value = msg.get(header, '')

**if** value:

**if** header=='Subject':

value = decode\_str(value)

**else**:

hdr, addr = parseaddr(value)

name = decode\_str(hdr)

value = u'%s <%s>' % (name, addr)

print('%s%s: %s' % (' ' \* indent, header, value))

**if** (msg.is\_multipart()):

parts = msg.get\_payload()

**for** n, part **in** enumerate(parts):

print('%spart %s' % (' ' \* indent, n))

print('%s--------------------' % (' ' \* indent))

print\_info(part, indent + 1)

**else**:

content\_type = msg.get\_content\_type()

**if** content\_type=='text/plain' **or** content\_type=='text/html':

content = msg.get\_payload(decode=True)

charset = guess\_charset(msg)

**if** charset:

content = content.decode(charset)

print('%sText: %s' % (' ' \* indent, content + '...'))

**else**:

print('%sAttachment: %s' % (' ' \* indent, content\_type))

邮件的Subject或者Email中包含的名字都是经过编码后的str，要正常显示，就必须decode：

**def** **decode\_str**(s):

value, charset = decode\_header(s)[0]

**if** charset:

value = value.decode(charset)

**return** value

decode\_header()返回一个list，因为像Cc、Bcc这样的字段可能包含多个邮件地址，所以解析出来的会有多个元素。上面的代码我们偷了个懒，只取了第一个元素。

文本邮件的内容也是str，还需要检测编码，否则，非UTF-8编码的邮件都无法正常显示：

**def** **guess\_charset**(msg):

charset = msg.get\_charset()

**if** charset **is** None:

content\_type = msg.get('Content-Type', '').lower()

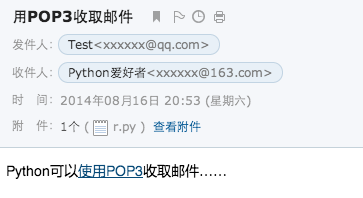
pos = content\_type.find('charset=')

**if** pos >= 0:

charset = content\_type[pos + 8:].strip()

**return** charset

把上面的代码整理好，我们就可以来试试收取一封邮件。先往自己的邮箱发一封邮件，然后用浏览器登录邮箱，看看邮件收到没，如果收到了，我们就来用Python程序把它收到本地：



运行程序，结果如下：

+OK Welcome to coremail Mail Pop3 Server (163coms[...])

Messages: 126. Size: 27228317

From: Test <xxxxxx@qq.com>

To: Python爱好者 <xxxxxx@163.com>

Subject: 用POP3收取邮件

part 0

--------------------

part 0

--------------------

Text: Python可以使用POP3收取邮件……...

part 1

--------------------

Text: Python可以<a href="...">使用POP3</a>收取邮件……...

part 1

--------------------

Attachment: application/octet-stream

我们从打印的结构可以看出，这封邮件是一个MIMEMultipart，它包含两部分：第一部分又是一个MIMEMultipart，第二部分是一个附件。而内嵌的MIMEMultipart是一个alternative类型，它包含一个纯文本格式的MIMEText和一个HTML格式的MIMEText。

### 小结

用Python的poplib模块收取邮件分两步：第一步是用POP3协议把邮件获取到本地，第二步是用email模块把原始邮件解析为Message对象，然后，用适当的形式把邮件内容展示给用户即可。

# 访问数据库

程序运行的时候，数据都是在内存中的。当程序终止的时候，通常都需要将数据保存到磁盘上，无论是保存到本地磁盘，还是通过网络保存到服务器上，最终都会将数据写入磁盘文件。

而如何定义数据的存储格式就是一个大问题。如果我们自己来定义存储格式，比如保存一个班级所有学生的成绩单：

|  |  |
| --- | --- |
| 名字 | 成绩 |
| Michael | 99 |
| Bob | 85 |
| Bart | 59 |
| Lisa | 87 |

你可以用一个文本文件保存，一行保存一个学生，用,隔开：

Michael,99

Bob,85

Bart,59

Lisa,87

你还可以用JSON格式保存，也是文本文件：

[

{"name":"Michael","score":99},

{"name":"Bob","score":85},

{"name":"Bart","score":59},

{"name":"Lisa","score":87}

]

你还可以定义各种保存格式，但是问题来了：

存储和读取需要自己实现，JSON还是标准，自己定义的格式就各式各样了；

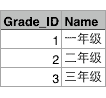
不能做快速查询，只有把数据全部读到内存中才能自己遍历，但有时候数据的大小远远超过了内存（比如蓝光电影，40GB的数据），根本无法全部读入内存。

为了便于程序保存和读取数据，而且，能直接通过条件快速查询到指定的数据，就出现了数据库（Database）这种专门用于集中存储和查询的软件。

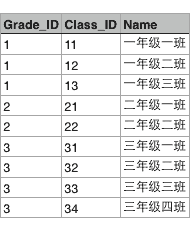
数据库软件诞生的历史非常久远，早在1950年数据库就诞生了。经历了网状数据库，层次数据库，我们现在广泛使用的关系数据库是20世纪70年代基于关系模型的基础上诞生的。

关系模型有一套复杂的数学理论，但是从概念上是十分容易理解的。举个学校的例子：

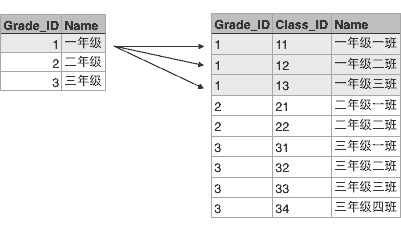
假设某个XX省YY市ZZ县第一实验小学有3个年级，要表示出这3个年级，可以在Excel中用一个表格画出来：



每个年级又有若干个班级，要把所有班级表示出来，可以在Excel中再画一个表格：



这两个表格有个映射关系，就是根据Grade\_ID可以在班级表中查找到对应的所有班级：

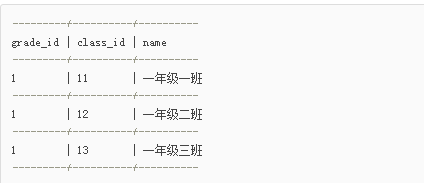


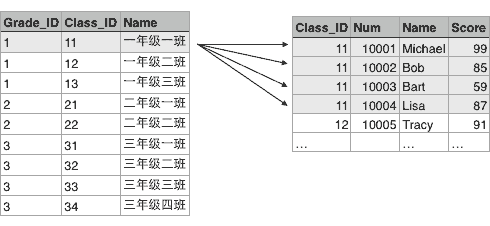
也就是Grade表的每一行对应Class表的多行，在关系数据库中，这种基于表（Table）的一对多的关系就是关系数据库的基础。

根据某个年级的ID就可以查找所有班级的行，这种查询语句在关系数据库中称为SQL语句，可以写成：

SELECT \* FROM classes WHERE grade\_id = '1';

结果也是一个表：

类似的，Class表的一行记录又可以关联到Student表的多行记录：



由于本教程不涉及到关系数据库的详细内容，如果你想从零学习关系数据库和基本的SQL语句，如果你想从零学习关系数据库和基本的SQL语句，请自行搜索相关课程。

NoSQL

你也许还听说过NoSQL数据库，很多NoSQL宣传其速度和规模远远超过关系数据库，所以很多同学觉得有了NoSQL是否就不需要SQL了呢？千万不要被他们忽悠了，连SQL都不明白怎么可能搞明白NoSQL呢？

数据库类别

既然我们要使用关系数据库，就必须选择一个关系数据库。目前广泛使用的关系数据库也就这么几种：

付费的商用数据库：

* Oracle，典型的高富帅；
* SQL Server，微软自家产品，Windows定制专款；
* DB2，IBM的产品，听起来挺高端；
* Sybase，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在Web的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是Google、Facebook，还是国内的BAT，无一例外都选择了免费的开源数据库：

* MySQL，大家都在用，一般错不了；
* PostgreSQL，学术气息有点重，其实挺不错，但知名度没有MySQL高；
* sqlite，嵌入式数据库，适合桌面和移动应用。

作为Python开发工程师，选择哪个免费数据库呢？当然是MySQL。因为MySQL普及率最高，出了错，可以很容易找到解决方法。而且，围绕MySQL有一大堆监控和运维的工具，安装和使用很方便。

为了能继续后面的学习，你需要从MySQL官方网站下载并安装[MySQL Community Server 5.6](http://dev.mysql.com/downloads/mysql/" \t "_blank)，这个版本是免费的，其他高级版本是要收钱的（请放心，收钱的功能我们用不上）。

## 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。

在使用SQLite前，我们先要搞清楚几个概念：

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。

由于SQLite的驱动内置在Python标准库中，所以我们可以直接来操作SQLite数据库。

# 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))')

# 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\')')

# 通过rowcount获得插入的行数:

print(cursor.rowcount)

# 关闭Cursor:

cursor.close()

# 提交事务:

conn.commit()

# 关闭Connection:

conn.close()

#我们再试试查询记录：

conn = sqlite3.connect('test.db')

cursor = conn.cursor()

# 执行查询语句:

cursor.execute('select \* from user where id=?', ('1',))

# 获得查询结果集:

values = cursor.fetchall()

print(values)

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featchall()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。具体文档请参阅SQLite官方网站。

### 小结

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。

如何才能确保出错的情况下也关闭掉Connection对象和Cursor对象呢？请回忆try:...except:...finally:...的用法。

## 使用MySQL

MySQL是Web世界中使用最广泛的数据库服务器。SQLite的特点是轻量级、可嵌入，但不能承受高并发访问，适合桌面和移动应用。而MySQL是为服务器端设计的数据库，能承受高并发访问，同时占用的内存也远远大于SQLite。

此外，MySQL内部有多种数据库引擎，最常用的引擎是支持数据库事务的InnoDB。

安装MySQL

可以直接从MySQL官方网站下载最新的[Community Server 5.6.x](http://dev.mysql.com/downloads/mysql/5.6.html" \t "_blank)版本。MySQL是跨平台的，选择对应的平台下载安装文件，安装即可。

安装时，MySQL会提示输入root用户的口令，请务必记清楚。如果怕记不住，就把口令设置为password。

在Windows上，安装时请选择UTF-8编码，以便正确地处理中文。

在Mac或Linux上，需要编辑MySQL的配置文件，把数据库默认的编码全部改为UTF-8。MySQL的配置文件默认存放在/etc/my.cnf或者/etc/mysql/my.cnf：

[client]

default-character-set = utf8

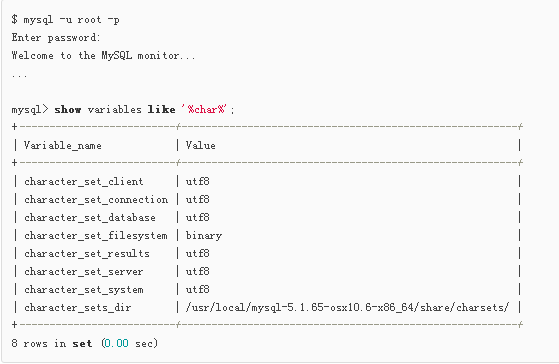
[mysqld]

default-storage-engine = INNODB

character-set-server = utf8

collation-server = utf8\_general\_ci

重启MySQL后，可以通过MySQL的客户端命令行检查编码：



看到utf8字样就表示编码设置正确。

注：如果MySQL的版本≥5.5.3，可以把编码设置为utf8mb4，utf8mb4和utf8完全兼容，但它支持最新的Unicode标准，可以显示emoji字符。

安装MySQL驱动

由于MySQL服务器以独立的进程运行，并通过网络对外服务，所以，需要支持Python的MySQL驱动来连接到MySQL服务器。MySQL官方提供了mysql-connector-python驱动，但是安装的时候需要给pip命令加上参数--allow-external：

$ pip install mysql-connector-python --allow-external mysql-connector-python

如果上面的命令安装失败，可以试试另一个驱动：

$ pip install mysql-connector

我们演示如何连接到MySQL服务器的数据库：

# 导入mysql驱动

import mysql.connector

try:

# 连接数据库

conn = mysql.connector.connect(database='db1', user='dbuser1',

password='dbuser1', host='192.168.64.138', port='3306')

print ("Opened database successfully")

# 创建游标

cur = conn.cursor()

# 创建user表

cur.execute('''

drop table if exists vo84\_user

''')

cur.execute('''

create table vo84\_user(

id varchar(10) primary key,

name varchar(20)

)''')

print ("Table created successfully")

# 插入一行记录，注意MySQL的占位符是%s:

cur.execute('insert into vo84\_user (id, name) values (%s, %s)', ['1', 'Michael'])

# print(cur.rowcount)

# 提交事务

conn.commit()

# 运行查询

cur.execute('select \* from vo84\_user where id = %s', ('1',))

values = cur.fetchall()

print(values)

except Exception as e:

print(e)

finally:

# 关闭游标及断开数据库连接

cur.close()

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。

小结

* 执行INSERT等操作后要调用commit()提交事务；
* MySQL的SQL占位符是%s。

## 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。是不是很简单？

但是由谁来做这个转换呢？所以ORM框架应运而生。

在Python中，最有名的ORM框架是SQLAlchemy。我们来看看SQLAlchemy的用法。

首先通过pip安装SQLAlchemy：

$ pip install sqlalchemy

然后，利用上次我们在MySQL的test数据库中创建的user表，用SQLAlchemy来试试：

第一步，导入SQLAlchemy，并初始化DBSession：

# 导入:

from sqlalchemy import Column, String, create\_engine

from sqlalchemy.orm import sessionmaker

from sqlalchemy.ext.declarative import declarative\_base

# 创建对象的基类:

Base = declarative\_base()

# 定义User对象:

class User(Base):

# 表的名字:

\_\_tablename\_\_ = 'user'

# 表的结构:

id = Column(String(20), primary\_key=True)

name = Column(String(20))

# 初始化数据库连接:

engine = create\_engine('mysql+mysqlconnector://root:password@localhost:3306/test')

# 创建DBSession类型:

DBSession = sessionmaker(bind=engine)

以上代码完成SQLAlchemy的初始化和具体每个表的class定义。如果有多个表，就继续定义其他class，例如School：

class School(Base):

\_\_tablename\_\_ = 'school'

id = ...

name = ...

create\_engine()用来初始化数据库连接。SQLAlchemy用一个字符串表示连接信息：

'数据库类型+数据库驱动名称://用户名:口令@机器地址:端口号/数据库名'

你只需要根据需要替换掉用户名、口令等信息即可。

下面，我们看看如何向数据库表中添加一行记录。

由于有了ORM，我们向数据库表中添加一行记录，可以视为添加一个User对象：

# 创建session对象:

session = DBSession()

# 创建新User对象:

new\_user = User(id='5', name='Bob')

# 添加到session:

session.add(new\_user)

# 提交即保存到数据库:

session.commit()

# 关闭session:

session.close()

可见，关键是获取session，然后把对象添加到session，最后提交并关闭。DBSession对象可视为当前数据库连接。

如何从数据库表中查询数据呢？有了ORM，查询出来的可以不再是tuple，而是User对象。SQLAlchemy提供的查询接口如下：

# 创建Session:

session = DBSession()

# 创建Query查询，filter是where条件，最后调用one()返回唯一行，如果调用all()则返回所有行:

user = session.query(User).filter(User.id=='5').one()

# 打印类型和对象的name属性:

print('type:', type(user))

print('name:', user.name)

# 关闭Session:

session.close()

运行结果如下：

type: <class '\_\_main\_\_.User'>

name: Bob

可见，ORM就是把数据库表的行与相应的对象建立关联，互相转换。

由于关系数据库的多个表还可以用外键实现一对多、多对多等关联，相应地，ORM框架也可以提供两个对象之间的一对多、多对多等功能。

例如，如果一个User拥有多个Book，就可以定义一对多关系如下：

class User(Base):

\_\_tablename\_\_ = 'user'

id = Column(String(20), primary\_key=True)

name = Column(String(20))

# 一对多:

books = relationship('Book')

class Book(Base):

\_\_tablename\_\_ = 'book'

id = Column(String(20), primary\_key=True)

name = Column(String(20))

# “多”的一方的book表是通过外键关联到user表的:

user\_id = Column(String(20), ForeignKey('user.id'))

当我们查询一个User对象时，该对象的books属性将返回一个包含若干个Book对象的list。

小结

ORM框架的作用就是把数据库表的一行记录与一个对象互相做自动转换。

正确使用ORM的前提是了解关系数据库的原理。

# Web开发

最早的软件都是运行在大型机上的，软件使用者通过“哑终端”登陆到大型机上去运行软件。后来随着PC机的兴起，软件开始主要运行在桌面上，而数据库这样的软件运行在服务器端，这种Client/Server模式简称CS架构。

随着互联网的兴起，人们发现，CS架构不适合Web，最大的原因是Web应用程序的修改和升级非常迅速，而CS架构需要每个客户端逐个升级桌面App，因此，Browser/Server模式开始流行，简称BS架构。

在BS架构下，客户端只需要浏览器，应用程序的逻辑和数据都存储在服务器端。浏览器只需要请求服务器，获取Web页面，并把Web页面展示给用户即可。

当然，Web页面也具有极强的交互性。由于Web页面是用HTML编写的，而HTML具备超强的表现力，并且，服务器端升级后，客户端无需任何部署就可以使用到新的版本，因此，BS架构迅速流行起来。

今天，除了重量级的软件如Office，Photoshop等，大部分软件都以Web形式提供。比如，新浪提供的新闻、博客、微博等服务，均是Web应用。

Web应用开发可以说是目前软件开发中最重要的部分。Web开发也经历了好几个阶段：

1. 静态Web页面：由文本编辑器直接编辑并生成静态的HTML页面，如果要修改Web页面的内容，就需要再次编辑HTML源文件，早期的互联网Web页面就是静态的；
2. CGI：由于静态Web页面无法与用户交互，比如用户填写了一个注册表单，静态Web页面就无法处理。要处理用户发送的动态数据，出现了Common Gateway Interface，简称CGI，用C/C++编写。
3. ASP/JSP/PHP：由于Web应用特点是修改频繁，用C/C++这样的低级语言非常不适合Web开发，而脚本语言由于开发效率高，与HTML结合紧密，因此，迅速取代了CGI模式。ASP是微软推出的用VBScript脚本编程的Web开发技术，而JSP用Java来编写脚本，PHP本身则是开源的脚本语言。
4. MVC：为了解决直接用脚本语言嵌入HTML导致的可维护性差的问题，Web应用也引入了Model-View-Controller的模式，来简化Web开发。ASP发展为ASP.Net，JSP和PHP也有一大堆MVC框架。

目前，Web开发技术仍在快速发展中，异步开发、新的MVVM前端技术层出不穷。

Python的诞生历史比Web还要早，由于Python是一种解释型的脚本语言，开发效率高，所以非常适合用来做Web开发。

Python有上百种Web开发框架，有很多成熟的模板技术，选择Python开发Web应用，不但开发效率高，而且运行速度快。

本章我们会详细讨论Python Web开发技术。

## 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。而浏览器和服务器之间的传输协议是HTTP，所以：

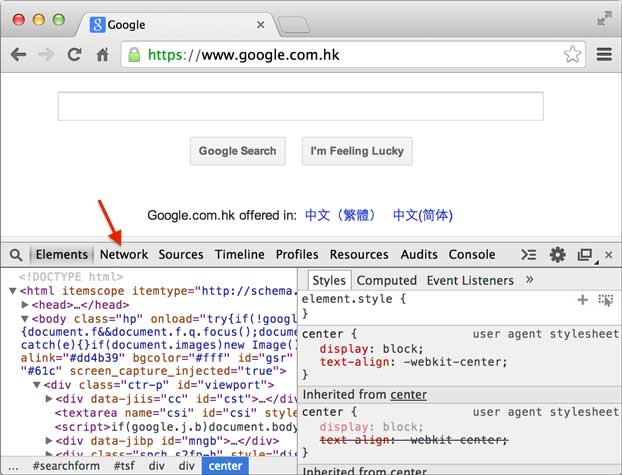
* HTML是一种用来定义网页的文本，会HTML，就可以编写网页；
* HTTP是在网络上传输HTML的协议，用于浏览器和服务器的通信。

在举例子之前，我们需要安装Google的[Chrome浏览器](http://www.google.com/intl/zh-CN/chrome/" \t "_blank)。

为什么要使用Chrome浏览器而不是IE呢？因为IE实在是太慢了，并且，IE对于开发和调试Web应用程序完全是一点用也没有。

我们需要在浏览器很方便地调试我们的Web应用，而Chrome提供了一套完整地调试工具，非常适合Web开发。

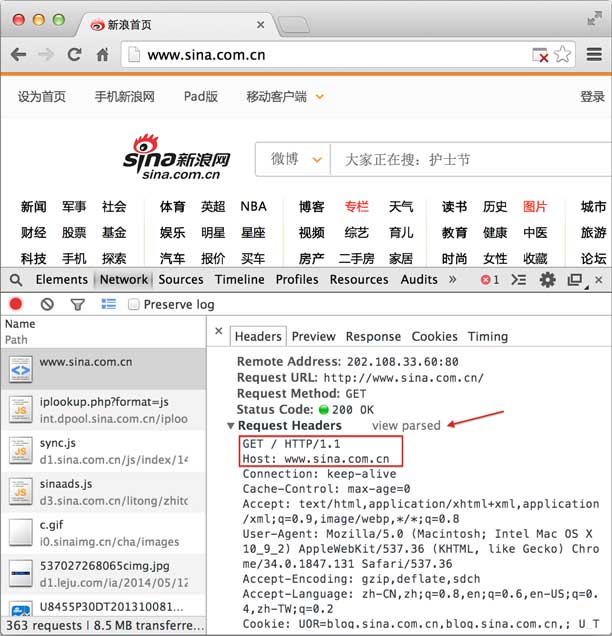
安装好Chrome浏览器后，打开Chrome，在菜单中选择“视图”，“开发者”，“开发者工具”，就可以显示开发者工具：



Elements显示网页的结构，Network显示浏览器和服务器的通信。我们点Network，确保第一个小红灯亮着，Chrome就会记录所有浏览器和服务器之间的通信：



当我们在地址栏输入www.sina.com.cn时，浏览器将显示新浪的首页。在这个过程中，浏览器都干了哪些事情呢？通过Network的记录，我们就可以知道。在Network中，定位到第一条记录，点击，右侧将显示Request Headers，点击右侧的view source，我们就可以看到浏览器发给新浪服务器的请求：



最主要的头两行分析如下，第一行：

GET / HTTP/1.1

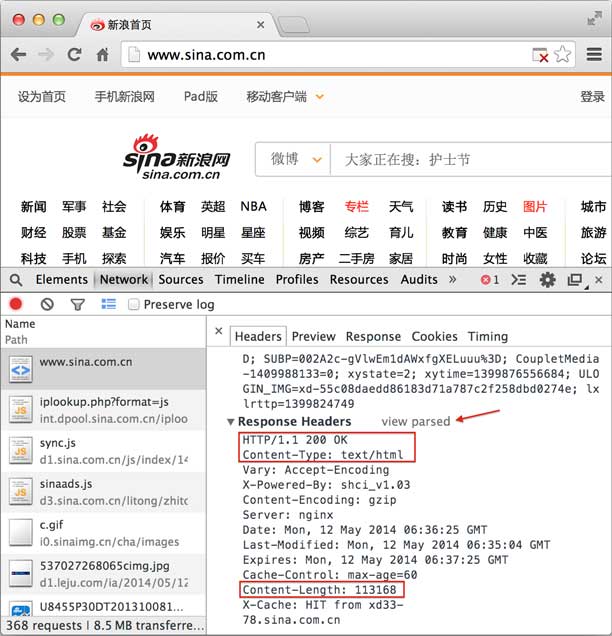
GET表示一个读取请求，将从服务器获得网页数据，/表示URL的路径，URL总是以/开头，/就表示首页，最后的HTTP/1.1指示采用的HTTP协议版本是1.1。目前HTTP协议的版本就是1.1，但是大部分服务器也支持1.0版本，主要区别在于1.1版本允许多个HTTP请求复用一个TCP连接，以加快传输速度。

从第二行开始，每一行都类似于Xxx: abcdefg：

Host: www.sina.com.cn

表示请求的域名是www.sina.com.cn。如果一台服务器有多个网站，服务器就需要通过Host来区分浏览器请求的是哪个网站。

继续往下找到Response Headers，点击view source，显示服务器返回的原始响应数据：



HTTP响应分为Header和Body两部分（Body是可选项），我们在Network中看到的Header最重要的几行如下：

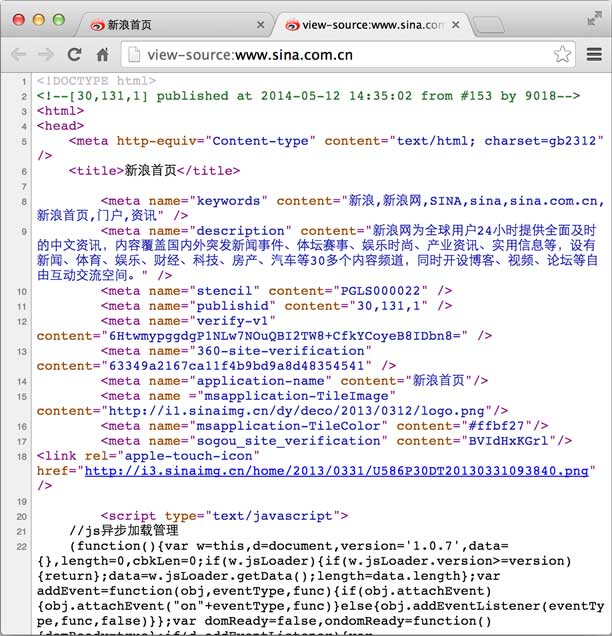
200 OK

200表示一个成功的响应，后面的OK是说明。失败的响应有404 Not Found：网页不存在，500 Internal Server Error：服务器内部出错，等等。

Content-Type: text/html

Content-Type指示响应的内容，这里是text/html表示HTML网页。请注意，浏览器就是依靠Content-Type来判断响应的内容是网页还是图片，是视频还是音乐。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，它也不一定就是图片。

HTTP响应的Body就是HTML源码，我们在菜单栏选择“视图”，“开发者”，“查看网页源码”就可以在浏览器中直接查看HTML源码：



当浏览器读取到新浪首页的HTML源码后，它会解析HTML，显示页面，然后，根据HTML里面的各种链接，再发送HTTP请求给新浪服务器，拿到相应的图片、视频、Flash、JavaScript脚本、CSS等各种资源，最终显示出一个完整的页面。所以我们在Network下面能看到很多额外的HTTP请求。

### HTTP请求

跟踪了新浪的首页，我们来总结一下HTTP请求的流程：

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

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。当我们编写一个页面时，我们只需要在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<img src="http://i1.sinaimg.cn/home/2013/1008/U8455P30DT20131008135420.png">，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。

### 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。

要详细了解HTTP协议，推荐“[HTTP: The Definitive Guide](http://shop.oreilly.com/product/9781565925090.do" \t "_blank)”一书，非常不错，有中文译本：

[HTTP权威指南](http://t.cn/R7FguRq" \t "_blank)

## HTML简介

网页就是HTML？这么理解大概没错。因为网页中不但包含文字，还有图片、视频、Flash小游戏，有复杂的排版、动画效果，所以，HTML定义了一套语法规则，来告诉浏览器如何把一个丰富多彩的页面显示出来。

HTML长什么样？上次我们看了新浪首页的HTML源码，如果仔细数数，竟然有6000多行！

所以，学HTML，就不要指望从新浪入手了。我们来看看最简单的HTML长什么样：

<html>

<head>

<title>Hello</title>

</head>

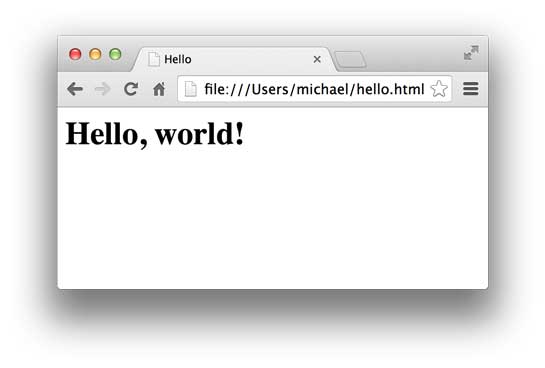
<body>

<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<html>。规范的HTML也包含<head>...</head>和<body>...</body>（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。

### 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：



### JavaScript简介

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>

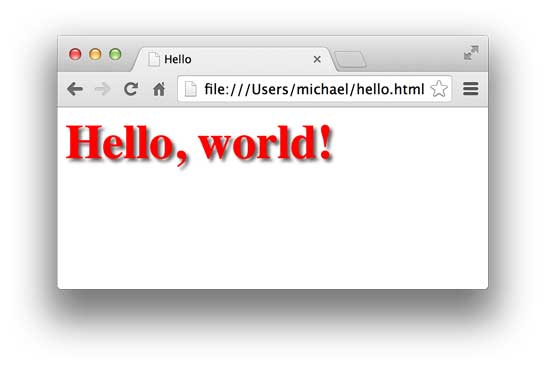
<body>

<h1 onclick="change()">Hello, world!</h1>

</body>

</html>

点击标题后效果如下：



### 小结

如果要学习Web开发，首先要对HTML、CSS和JavaScript作一定的了解。HTML定义了页面的内容，CSS来控制页面元素的样式，而JavaScript负责页面的交互逻辑。

讲解HTML、CSS和JavaScript就可以写3本书，对于优秀的Web开发人员来说，精通HTML、CSS和JavaScript是必须的，这里推荐一个在线学习网站w3schools：

[http://www.w3schools.com/](http://www.w3schools.com/" \t "_blank)

以及一个对应的中文版本：

[http://www.w3school.com.cn/](http://www.w3school.com.cn/" \t "_blank)

当我们用Python或者其他语言开发Web应用时，我们就是要在服务器端动态创建出HTML，这样，浏览器就会向不同的用户显示出不同的Web页面。

## WSGI接口

了解了HTTP协议和HTML文档，我们其实就明白了一个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。

所以，最简单的Web应用就是先把HTML用文件保存好，用一个现成的HTTP服务器软件，接收用户请求，从文件中读取HTML，返回。Apache、Nginx、Lighttpd等这些常见的静态服务器就是干这件事情的。

如果要动态生成HTML，就需要把上述步骤自己来实现。不过，接受HTTP请求、解析HTTP请求、发送HTTP响应都是苦力活，如果我们自己来写这些底层代码，还没开始写动态HTML呢，就得花个把月去读HTTP规范。

正确的做法是底层代码由专门的服务器软件实现，我们用Python专注于生成HTML文档。因为我们不希望接触到TCP连接、HTTP原始请求和响应格式，所以，需要一个统一的接口，让我们专心用Python编写Web业务。

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')]) #发送Header

**return** [b'<h1>Hello, web!</h1>'] #返回Body

上面的application()函数就是符合WSGI标准的一个HTTP处理函数(函数名不一定是application)，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，我们关心的就是如何从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，底层代码不需要我们自己编写，我们只负责在更高层次上考虑如何响应请求就可以了。

不过，等等，这个application()函数怎么调用？如果我们自己调用，两个参数environ和start\_response我们没法提供，返回的bytes也没法发给浏览器。

所以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，我们可以挑选一个来用。但是现在，我们只想尽快测试一下我们编写的application()函数真的可以把HTML输出到浏览器，所以，要赶紧找一个最简单的WSGI服务器，把我们的Web应用程序跑起来。

好消息是Python内置了一个WSGI服务器，这个模块叫wsgiref，它是用纯Python编写的WSGI服务器的参考实现。所谓“参考实现”是指该实现完全符合WSGI标准，但是不考虑任何运行效率，仅供开发和测试使用。

运行WSGI服务

我们先编写hello.py，实现Web应用程序的WSGI处理函数：

*# hello.py*

**def** **application**(environ, start\_response):

start\_response('200 OK', [('Content-Type', 'text/html')])

**return** [b'<h1>Hello, web!</h1>']

然后，再编写一个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:*

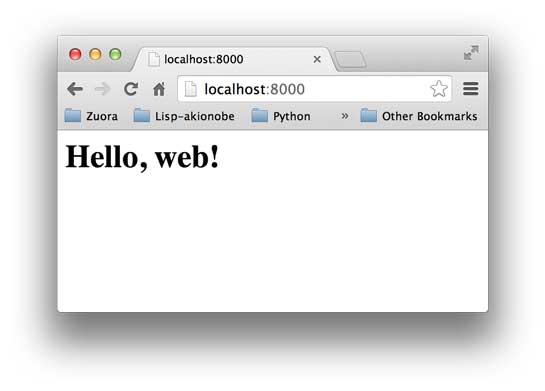
httpd.serve\_forever()

确保以上两个文件在同一个目录下，然后在命令行输入python server.py来启动WSGI服务器：

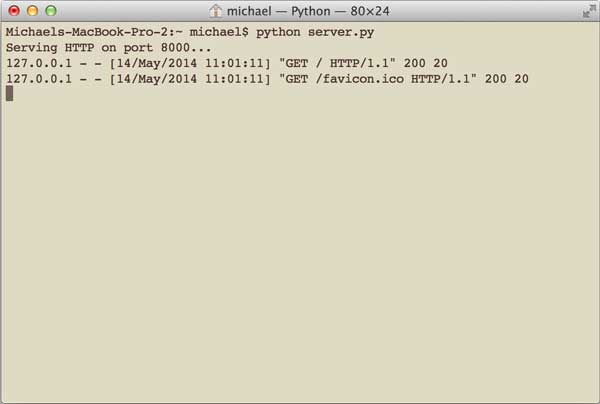


注意：如果8000端口已被其他程序占用，启动将失败，请修改成其他端口。

启动成功后，打开浏览器，输入http://localhost:8000/，就可以看到结果了：



在命令行可以看到wsgiref打印的log信息：



按Ctrl+C终止服务器。

如果你觉得这个Web应用太简单了，可以稍微改造一下，从environ里读取PATH\_INFO，这样可以显示更加动态的内容：

*# hello.py*

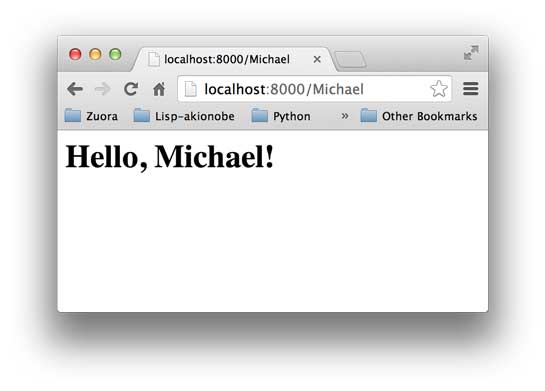
**def** **application**(environ, start\_response):

start\_response('200 OK', [('Content-Type', 'text/html')])

body = '<h1>Hello, %s!</h1>' % (environ['PATH\_INFO'][1:] **or** 'web')

**return** [body.encode('utf-8')]

你可以在地址栏输入用户名作为URL的一部分，将返回Hello, xxx!：



是不是有点Web App的感觉了？

### 小结

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。

复杂的Web应用程序，光靠一个WSGI函数来处理还是太底层了，我们需要在WSGI之上再抽象出Web框架，进一步简化Web开发。

## 使用Web框架

了解了WSGI框架，我们发现：其实一个Web App，就是写一个WSGI的处理函数，针对每个HTTP请求进行响应。

但是如何处理HTTP请求不是问题，问题是如何处理100个不同的URL。

每一个URL可以对应GET和POST请求，当然还有PUT、DELETE等请求，但是我们通常只考虑最常见的GET和POST请求。

一个最简单的想法是从environ变量里取出HTTP请求的信息，然后逐个判断：

def application(environ, start\_response):

method = environ['REQUEST\_METHOD']

path = environ['PATH\_INFO']

if method=='GET' and path=='/':

return handle\_home(environ, start\_response)

if method=='POST' and path='/signin':

return handle\_signin(environ, start\_response)

...

只是这么写下去代码是肯定没法维护了。

代码这么写没法维护的原因是因为WSGI提供的接口虽然比HTTP接口高级了不少，但和Web App的处理逻辑比，还是比较低级，我们需要在WSGI接口之上能进一步抽象，让我们专注于用一个函数处理一个URL，至于URL到函数的映射，就交给Web框架来做。

由于用Python开发一个Web框架十分容易，所以Python有上百个开源的Web框架。这里我们先不讨论各种Web框架的优缺点，直接选择一个比较流行的Web框架——[Flask](http://flask.pocoo.org/" \t "_blank)来使用。

用Flask编写Web App比WSGI接口简单（这不是废话么，要是比WSGI还复杂，用框架干嘛？），我们先用pip安装Flask：

$ pip install flask

然后写一个app.py，处理3个URL，分别是：

GET /：首页，返回Home；

GET /signin：登录页，显示登录表单；

POST /signin：处理登录表单，显示登录结果。

注意噢，同一个URL/signin分别有GET和POST两种请求，映射到两个处理函数中。

Flask通过Python的[装饰器](http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014318435599930270c0381a3b44db991cd6d858064ac0000)在内部自动地把URL和函数给关联起来，所以，我们写出来的代码就像这样：

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\_\_':

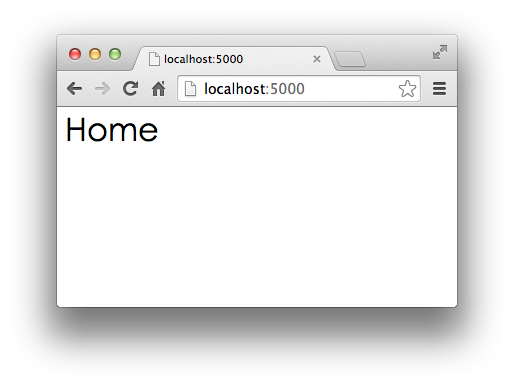
app.run()

运行python app.py，Flask自带的Server在端口5000上监听：

$ python app.py

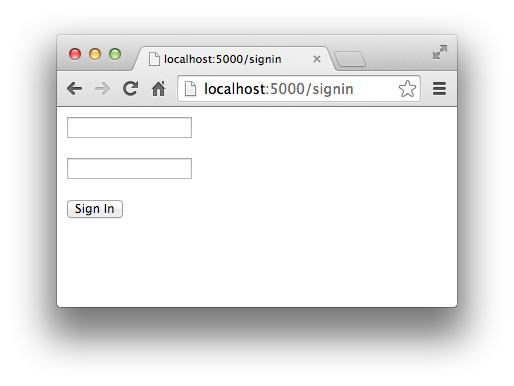
\* Running on http://127.0.0.1:5000/

打开浏览器，输入首页地址http://localhost:5000/：

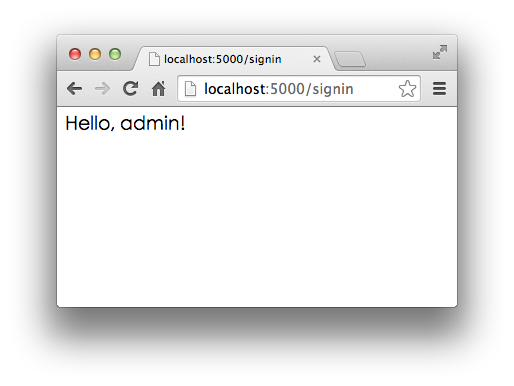


首页显示正确！

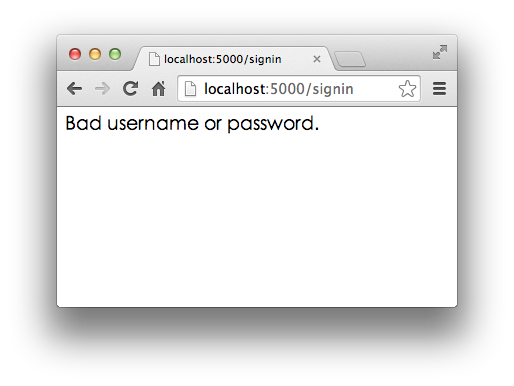
再在浏览器地址栏输入http://localhost:5000/signin，会显示登录表单：



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。

除了Flask，常见的Python Web框架还有：

[Django](https://www.djangoproject.com/" \t "_blank)：全能型Web框架；

[web.py](http://webpy.org/" \t "_blank)：一个小巧的Web框架；

[Bottle](http://bottlepy.org/" \t "_blank)：和Flask类似的Web框架；

[Tornado](http://www.tornadoweb.org/" \t "_blank)：Facebook的开源异步Web框架。

当然了，因为开发Python的Web框架也不是什么难事，我们后面也会讲到开发Web框架的内容。

### 小结

有了Web框架，我们在编写Web应用时，注意力就从WSGI处理函数转移到URL+对应的处理函数，这样，编写Web App就更加简单了。

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。

## 使用模板

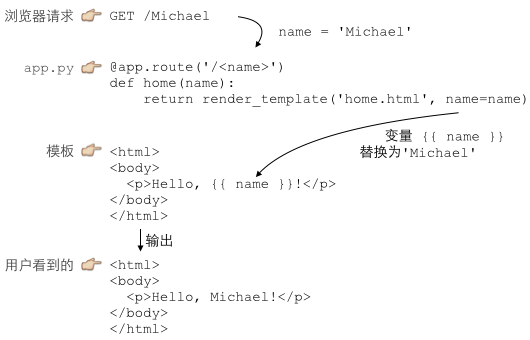
Web框架把我们从WSGI中拯救出来了。现在，我们只需要不断地编写函数，带上URL，就可以继续Web App的开发了。

但是，Web App不仅仅是处理逻辑，展示给用户的页面也非常重要。在函数中返回一个包含HTML的字符串，简单的页面还可以，但是，想想新浪首页的6000多行的HTML，你确信能在Python的字符串中正确地写出来么？反正我是做不到。

俗话说得好，不懂前端的Python工程师不是好的产品经理。有Web开发经验的同学都明白，Web App最复杂的部分就在HTML页面。HTML不仅要正确，还要通过CSS美化，再加上复杂的JavaScript脚本来实现各种交互和动画效果。总之，生成HTML页面的难度很大。

由于在Python代码里拼字符串是不现实的，所以，模板技术出现了。

使用模板，我们需要预先准备一个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，根据我们传入的数据，替换后，得到最终的HTML，发送给用户：



这就是传说中的MVC：Model-View-Controller，中文名“模型-视图-控制器”。

Python处理URL的函数就是C：Controller，Controller负责业务逻辑，比如检查用户名是否存在，取出用户信息等等；

包含变量{{ name }}的模板就是V：View，View负责显示逻辑，通过简单地替换一些变量，View最终输出的就是用户看到的HTML。

MVC中的Model在哪？Model是用来传给View的，这样View在替换变量的时候，就可以从Model中取出相应的数据。

上面的例子中，Model就是一个dict：

{ 'name': 'Michael' }

只是因为Python支持关键字参数，很多Web框架允许传入关键字参数，然后，在框架内部组装出一个dict作为Model。

现在，我们把上次直接输出字符串作为HTML的例子用高端大气上档次的MVC模式改写一下：

**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](http://jinja.pocoo.org/" \t "_blank)，所以我们先直接安装jinja2：

$ pip install jinja2

然后，开始编写jinja2模板：

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>

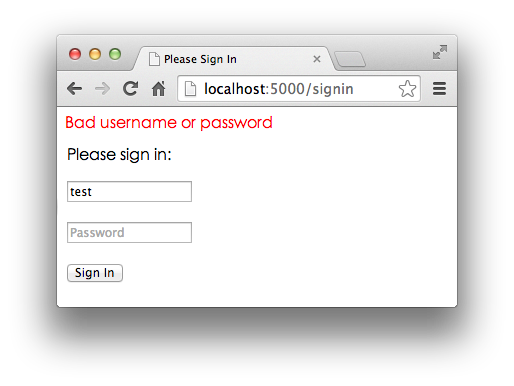
</html>

登录失败的模板呢？我们在form.html中加了一点条件判断，把form.html重用为登录失败的模板。

最后，一定要把模板放到正确的templates目录下，templates和app.py在同级目录下：

mvc-dir

启动python app.py，看看使用模板的页面效果：



通过MVC，我们在Python代码中处理M：Model和C：Controller，而V：View是通过模板处理的，这样，我们就成功地把Python代码和HTML代码最大限度地分离了。

使用模板的另一大好处是，模板改起来很方便，而且，改完保存后，刷新浏览器就能看到最新的效果，这对于调试HTML、CSS和JavaScript的前端工程师来说实在是太重要了。

在Jinja2模板中，我们用{{ name }}表示一个需要替换的变量。很多时候，还需要循环、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 %}

如果page\_list是一个list：[1, 2, 3, 4, 5]，上面的模板将输出5个超链接。

除了Jinja2，常见的模板还有：

* [Mako](http://www.makotemplates.org/" \t "_blank)：用<% ... %>和${xxx}的一个模板；
* [Cheetah](http://www.cheetahtemplate.org/" \t "_blank)：也是用<% ... %>和${xxx}的一个模板；
* [Django](https://www.djangoproject.com/" \t "_blank)：Django是一站式框架，内置一个用{% ... %}和{{ xxx }}的模板。

### 小结

有了MVC，我们就分离了Python代码和HTML代码。HTML代码全部放到模板里，写起来更有效率。

# 异步IO

在IO编程一节中，我们已经知道，CPU的速度远远快于磁盘、网络等IO。在一个线程中，CPU执行代码的速度极快，然而，一旦遇到IO操作，如读写文件、发送网络数据时，就需要等待IO操作完成，才能继续进行下一步操作。这种情况称为同步IO。

在IO操作的过程中，当前线程被挂起，而其他需要CPU执行的代码就无法被当前线程执行了。

因为一个IO操作就阻塞了当前线程，导致其他代码无法执行，所以我们必须使用多线程或者多进程来并发执行代码，为多个用户服务。每个用户都会分配一个线程，如果遇到IO导致线程被挂起，其他用户的线程不受影响。

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。

由于我们要解决的问题是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。

可以想象如果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)

所以，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)

消息模型其实早就应用在桌面应用程序中了。一个GUI程序的主线程就负责不停地读取消息并处理消息。所有的键盘、鼠标等消息都被发送到GUI程序的消息队列中，然后由GUI程序的主线程处理。

由于GUI线程处理键盘、鼠标等消息的速度非常快，所以用户感觉不到延迟。某些时候，GUI线程在一个消息处理的过程中遇到问题导致一次消息处理时间过长，此时，用户会感觉到整个GUI程序停止响应了，敲键盘、点鼠标都没有反应。这种情况说明在消息模型中，处理一个消息必须非常迅速，否则，主线程将无法及时处理消息队列中的其他消息，导致程序看上去停止响应。

消息模型是如何解决同步IO必须等待IO操作这一问题的呢？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。

## 协程

在学习异步IO模型前，我们先来了解协程。

协程，又称微线程，纤程。英文名Coroutine。

协程的概念很早就提出来了，但直到最近几年才在某些语言（如Lua）中得到广泛应用。

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。

看起来A、B的执行有点像多线程，但协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

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。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

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。

但是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。

来看例子：

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**(): #消费者

r = ''

**while** True:

n = **yield** r

**if** **not** n:

**return**

print('[CONSUMER] Consuming %s...' % n)

r = '200 OK'

**def** **produce**(c): #生产者

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)

执行结果：

[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：

1. 首先调用c.send(None)启动生成器；将生成器推进到yield位置。
2. 然后，一旦生产了东西，通过c.send(n)切换到consumer执行；
3. consumer通过yield拿到消息，处理，又通过yield把结果传回；
4. produce拿到consumer处理的结果，继续生产下一条消息；
5. 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。”

## asyncio

asyncio是Python 3.4版本引入的标准库，直接内置了对异步IO的支持。

asyncio的编程模型就是一个消息循环。我们从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，我们就把这个coroutine扔到EventLoop中执行。

hello()会首先打印出Hello world!，然后，yield from语法可以让我们方便地调用另一个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。

我们用Task封装两个coroutine试试：

**import** threading

**import** asyncio

@asyncio.coroutine

**def** **hello**():

print('Hello world! (%s)' % threading.currentThread())

**yield** **from** asyncio.sleep(1) #通过yield from调用会引起阻塞的操作

print('Hello again! (%s)' % threading.currentThread())

loop = asyncio.get\_event\_loop()

tasks = [hello(), hello()]

loop.run\_until\_complete(asyncio.wait(tasks))

loop.close()

观察执行过程：

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)>)

由打印的当前线程名称可以看出，两个coroutine是由同一个线程并发执行的。

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。

我们用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()

执行结果如下：

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。

### 小结

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。

## async/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。

为了简化并更好地标识异步IO，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。

请注意，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：

1. 把@asyncio.coroutine替换为async；
2. 把yield from替换为await。

让我们对比一下上一节的代码：

@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!")

剩下的代码保持不变。

### 小结

Python从3.5版本开始为asyncio提供了async和await的新语法；

注意新语法只能用在Python 3.5以及后续版本，如果使用3.4版本，则仍需使用上一节的方案。

练习

将上一节的异步获取sina、sohu和163的网站首页源码用新语法重写并运行。

## 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。

我们先安装aiohttp：

pip install aiohttp

然后编写一个HTTP服务器，分别处理以下URL：

* / - 首页返回b'<h1>Index</h1>'；
* /hello/{name} - 根据URL参数返回文本hello, %s!。

代码如下：

**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, %s!</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()

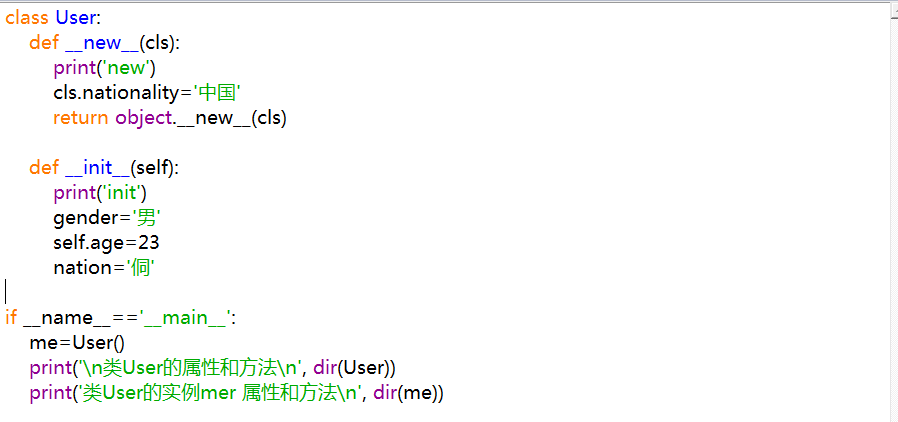
loop.run\_until\_complete(init(loop))

loop.run\_forever()

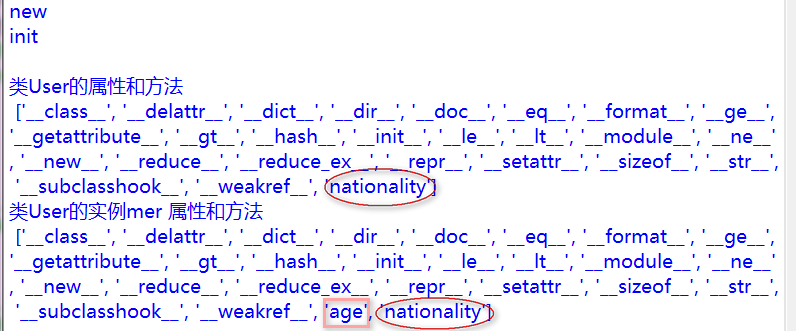
注意aiohttp的初始化函数init()也是一个coroutine，loop.create\_server()则利用asyncio创建TCP服务。

# 我的心得

## 关于\_\_init\_\_与\_\_new\_\_



执行结果如下



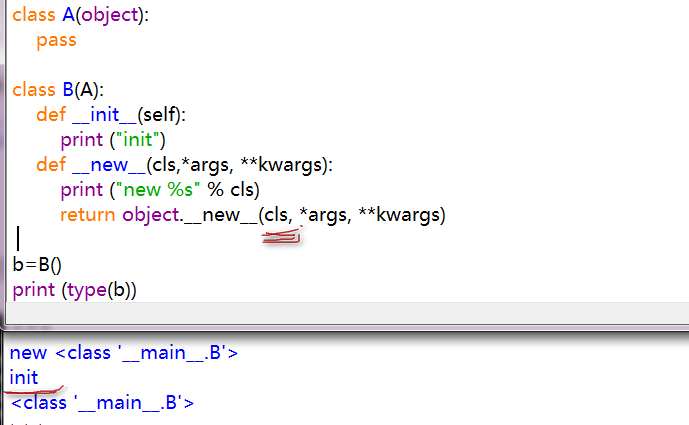
dir()—获取对象的所有属性和方法。

可以发现在实例化一个类时，\_\_init\_\_方法并不是最先被调用的。实际上\_\_init\_\_方法只是对类的实例进行一些初始化。而真正创建类实例的是\_\_new\_\_方法。

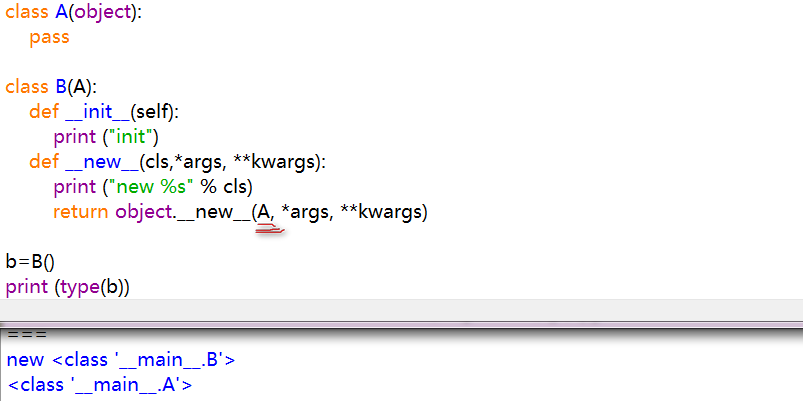
流程为，先通过\_\_new\_\_为类创建一个实例，并返回一个这个实例，然后将该实例传递给\_\_init\_\_进行一些初始化。所以\_\_new\_\_方法的第一个参数为类，而\_\_init\_\_方法的第一个参数为类的实例对象。在编程中我们常用cls参数名表示类，用self参数名表示类的实例。

在\_\_new\_\_中通过cls.nationality='中国'语句创建的是类的属性，而在\_\_init\_\_中通过self.age=23创建的是实例对象的属性。--需要注意的是：若\_\_new\_\_没有正确返回**当前类cls**的实例，那\_\_init\_\_是不会被调用的，即使是父类的实例也不行。

当\_\_new\_\_返回**当前类cls**的实例时，\_\_init\_\_在\_\_new\_\_方法之后被调用：



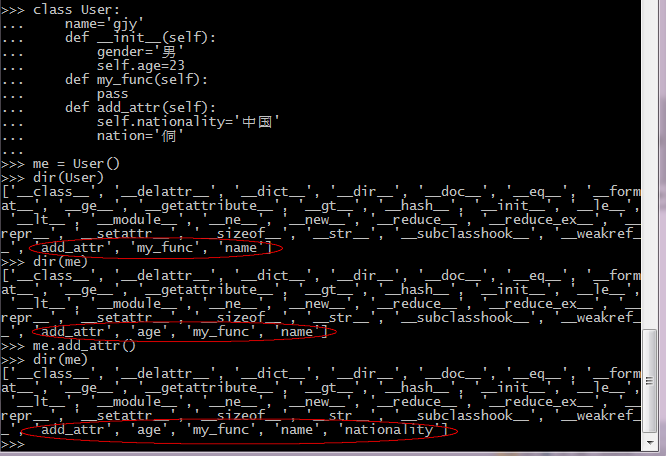
当\_\_new\_\_返回的不是**当前类cls**的实例时，\_\_init\_\_没有被调用：



另外：元类的\_\_new\_\_方法，

。

## 关于类与类的实例



注意：区别类与类的实例。这里User是类，me是类User的实例。

我们可以看到：

对于User类, 除默认的属性和方法外，还有name, my\_func, add\_attr。

对于实例me, 在没有执行add\_attr方法前，除默认的属性和方法外，还有name, my\_func, add\_attr, age。（实例有age属性是因为在创建实例时会自动调用\_\_init\_\_，在\_\_init\_\_中执行self.age=23为实例添加了一个age属性。）当调用add\_attr方法执行self.nationality='中国'诗句后，会为实例对象添加一个nationality属性。

python是动态语言它可以动态的为类或实例添加属性或方法。

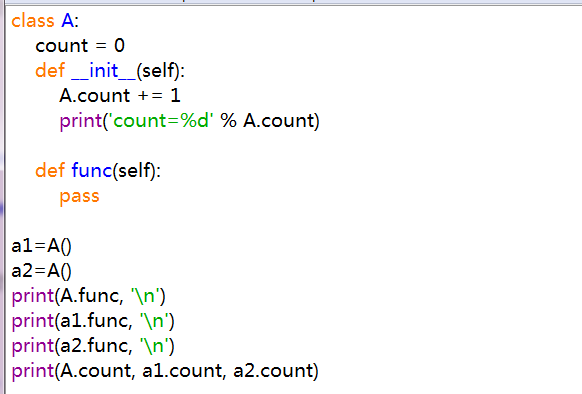
因为类的\_\_init\_\_方法的第一个参数是类的实例对象，所以可以self.xxx的方式来为类的实例添加相应的属性。----需要说明的是：这里的self不是关键字，它只不过是\_\_init\_\_方法的一个参数名，当然我们也可以用其它的参数名。

gender与age都是在\_\_init\_\_中定义的，为什么age是实例的属性，而gender不是实例的属性呢。关键在于age是以self.age方式声明的，

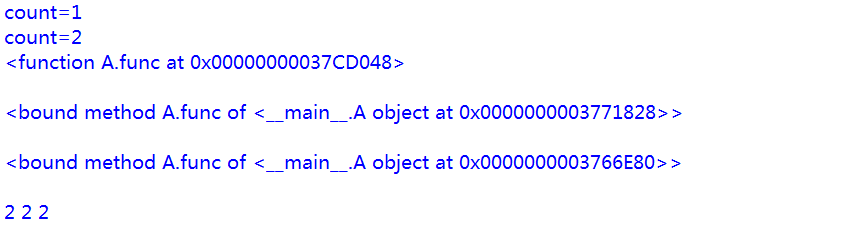
同理： add\_attr方法的第一个参数也表示类的实例对象，

注意：这里name与age的区别是，name是类的属性，而age是类实例的属性。类的属性属于该类的所有实例(由所有实例共享)，实例的属性由实例独享。

## 类属性与类实例的属性：

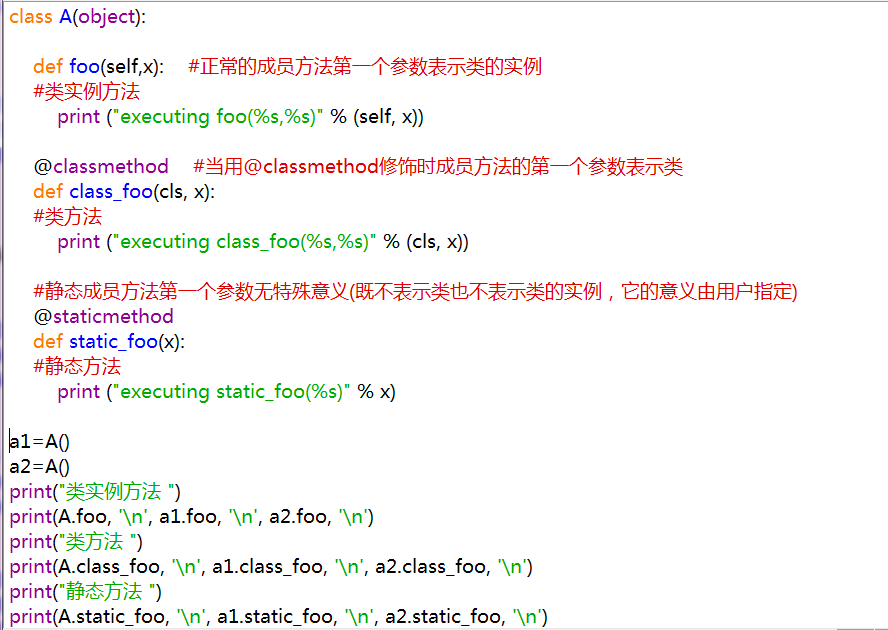


执行结果如下：

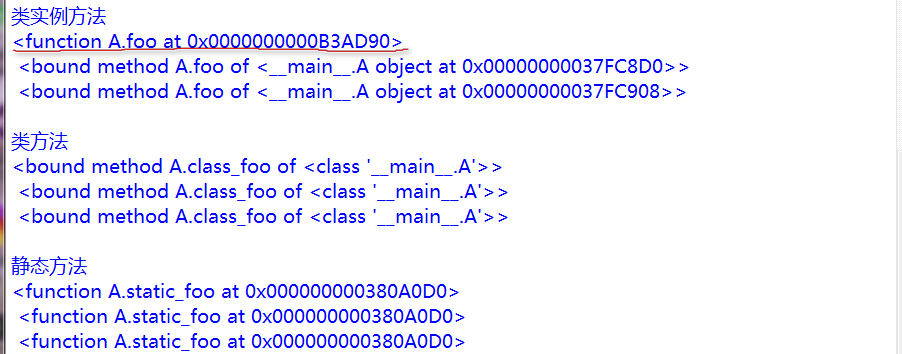


其中count为类属性，由所有该类的实例所共享。而func方法是类实例的方法，由每个实例所持有。

## python中类方法、类实例方法、静态方法的使用与区别



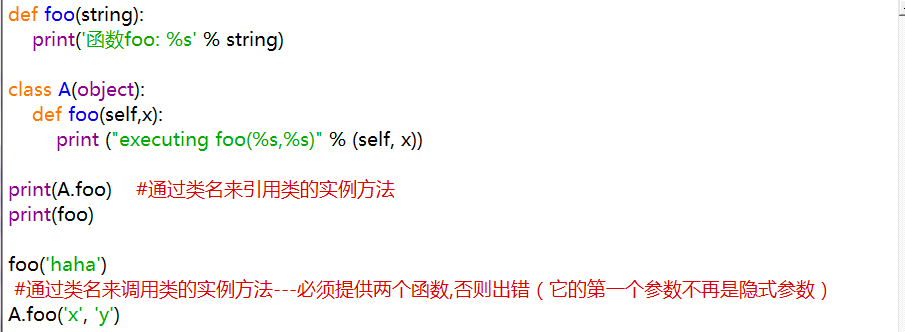
执行结果如下：



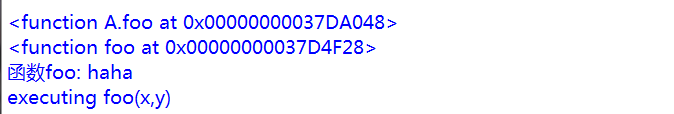
区别：

* 类方法和静态方法由所有类的实例所共享。类实例方法由每个实例所独享，每个实例都有它的一个副本。
* 类方法的隐含调用参数是类，而类实例方法的隐含调用参数是类的实例，静态方法没有隐含调用参数。（隐含参数即方法的第一个函数，所以使用时不需要显式传入）。

注意：划线部分说明当通过类（类名）来调用类实例的方法时，该方法与普通函数无异。证明如下：



执行结果如下：



通过类来引用或调用函数时，类名相当于python的包名。

\_\_new\_\_是类的方法吗？--若是它为什么没有写@classmethod

## 装饰器

### 装饰普通函数：

例：

def now():

print('2016-11-8')

例如现在用一个log函数来装饰上面的now函数，现执行now函数之前进行一些操作。

为了用log函数装饰now,我们需要在now函数前加上@log。--表示用log函数装饰now函数。

#在原now函数定义前加 @封装函数—此处为@log

@log #用log函数装饰now函数

def now():

print('2016-11-8')

然后再编写我们的log函数。--因为我们用log来装饰new，所以log需要一个参数（对应被装饰的函数）。

**import** functools

**def** **log**(func): #参数func为函数类型

@functools.wraps(func)

**def** **wrapper**(\*args, \*\*kw): #定义**wrapper，增强func的功能（返回func的调用结果）**

print('调用函数是 %s()' % func.\_\_name\_\_) #对func功能的增强

**return** func(\*args, \*\*kw) #执行函数并返回结果

**return** wrapper #返回**wrapper函数 ---可以理解为增加的func**

这样我们在使用now时存在如下等价关系：

now🡸🡺log(now); now()🡸🡺log(now)()🡸🡺wrapper()

被装饰的函数对象，等价与以它为参调用装饰器函数。

注意：func,与func()的区别---即函数与函数调用的区别

### 装饰类的方法：

@property装饰器。对于类的方法，Python内置的@property装饰器能够对一个方法的访问变成对属性的访问。

**class Student(object):**

#在**score**方法前面加@property,便能以操作属性的方式来调用这个方法。如:实例.score

@property

**def** **score**(self):

**return** self.\_score #返回动态添加的实例的\_score属性

#在前面加@xxx.setter(xxx为@property所装饰的方法名),便可以操作属性的方式来调用这个方法。如：实例.score=100

@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 #对动态添加的实例属性\_score赋值

@property装饰器使得被装饰的方法能以实例属性的形式调用。同时@property装饰器创建了另一个装饰器@xxx.setter(xxx为@property所装饰的方法名)，@xxx.setter装饰器使得它装饰的方法能以实例属性的方法调用。所不同的是被@property装饰的方法对应的是获取，被@xxx.setter装饰的方法对应的是设置。

>>> s = Student()

>>> s.score = 60 # OK，实际转化为s. **score**(60) ---@property修饰的方法

>>> s.score # OK，实际转化为s. **score**()---@xxx.setter修饰的方法

60

>>> s.score = 9999

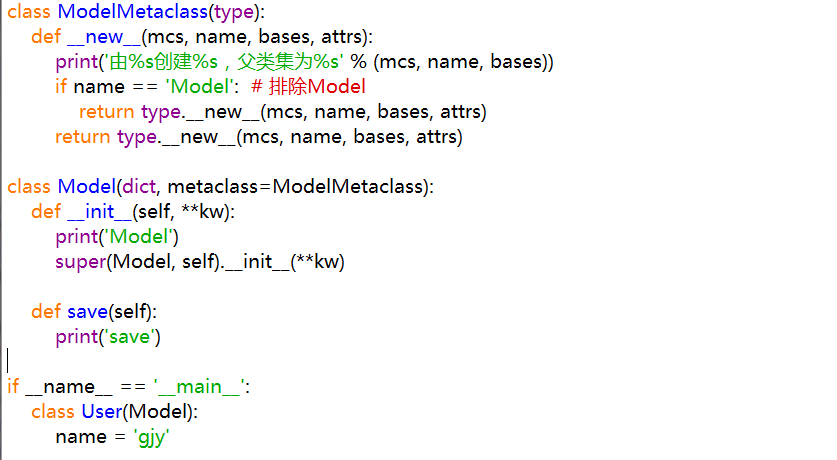
Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

当我们只需要读时，只定义@property装饰器即可(不需要@xxx.setter)

### 类也是一个对象。



执行结果如下：



根据输出结果：我们可以得到结论：

**对class Model(dict, metaclass=ModelMetaclass)的理解**：metaclass=ModelMetaclass参数表示Model类由元类ModelMetaclass生成，dict是Model的父类。

由元类ModelMetaclass生成继承自dict的Model类(实际上Model不只继承了dict,还隐式的继承了其它的类)。Model类由元类ModelMetaclass生成，它继承了dict。

items()