# 迭代

1. List, tuple, string, dict 都可以迭代

dict就可以迭代：

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

>>> **for** key **in** d:

... **print** key

...

a

c

b

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

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

1. 其他可迭代对象： 用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

1. 对list实现像java一样的下标循环，可以用enumerate函数把list变成一个index-element对。

>>> **for** i, value **in** enumerate(['A', 'B', 'C']):

... **print** i, value

1. 同时引用了两个变量，在Python里是很常见的，比如下面的代码：

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

... **print** x, y #1 1换行 2 4 换行3 9

# 列表生成式

列表生成式则可以用一行语句代替循环生成上面的list：

>>> [x \* x **for** x **in** range(1, 11)]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

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

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

因此，列表生成式也可以使用两个变量来生成list：

>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }

>>> [k + '=' + v **for** k, v **in** d.iteritems()]

['y=B', 'x=A', 'z=C']

把一个list中所有的字符串变成小写：

>>> L = ['Hello', 'World', 'IBM',18,None 'Apple']

>>> [s.lower() for s in L if isinstance(s,str)]

['hello', 'world', 'apple']#只输出了string

>>> L2 = [s.lower() if isinstance(s,str) else s for s in L] #能输出所有

# 生成器

受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果只访问几个元素，空间浪费。

列表可以按照某种算法算出来，就可以在循环过程中不断推出后续的元素，就不用创建完整的list,可以节省空间。一边循环一边计算，就叫Generator

方法一：将[]改成（）

>>> 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 0x104feab40>

g是一个generator, 要打印出g的元素就要用next()方法，generator保存的是算法，每次调用next()，就计算出下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。

当然，也可以用for循环，因为generator也是可迭代的

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

>>> **for** n **in** g:

... **print** n

方法二： 如果推算的算法比较复杂，用类似列表生成式的for循环无法实现的时候，还可以用函数来实现。

要把fib函数变成generator，只需要把print b改为yield b就可以了：

**def** **fib**(max):

n, a, b = 0, 0, 1

**while** n < max:

**yield** b

a, b = b, a + b

n = n + 1

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：

>>> fib(6)

<generator object fib at 0x104feaaa0>

**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。**

在执行过程中，遇到yield就中断，下次又继续执行。

generator的工作原理，它是在for循环的过程中不断计算出下一个元素，并在适当的条件结束for循环。对于函数改成的generator来说，遇到return语句或者执行到函数体最后一行语句，就是结束generator的指令，for循环随之结束。

# 函数式编程

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

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

### 函数名也是变量

那么函数名是什么呢？函数名其实就是指向函数的变量！对于abs()这个函数，完全可以把函数名abs看成变量

### 传入函数

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

**def** **add**(x, y, f):

**return** f(x) + f(y)

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

11

# map/reduce

map()函数接收两个参数，一个是函数，一个是序列，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list返回

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

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

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

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

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

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

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

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

>>> **def** **fn**(x, y):

... **return** x \* 10 + y

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

13579

**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]

**def** **str2int**(s):

**return** reduce(**lambda** x,y: x\*10+y, map(char2num, s))

利用map()函数，把用户输入的不规范的英文名字，变为首字母大写，其他小写的规范名字。输入：['adam', 'LISA', 'barT']，输出：['Adam', 'Lisa', 'Bart']。

**>>> name = ['adam', 'LISA', 'barT']**

**>>> map(lambda s:s.title(),name)**

**['Adam', 'Lisa', 'Bart']**

Python提供的sum()函数可以接受一个list并求和，请编写一个prod()函数，可以接受一个list并利用reduce()求积。

**>>> def prod(l):**

**return reduce(lambda x,y:x\*y,l)**

# filter

Python内建的filter()函数用于过滤序列。

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

**>>> def isNotPrime(n):**

**if n ==1:**

**return False**

**else:**

**for i in range(2,n):**

**if n%i==0:**

**return False**

**return True**

**>>> filter(isNotPrime,range(1,101))**

**[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]**

# sorted

比较的过程必须通过函数抽象出来。通常规定，对于两个元素x和y，如果认为x < y，则返回-1，如果认为x == y，则返回0，如果认为x > y，则返回1

sorted()函数也是一个高阶函数，它还可以接收一个比较函数来实现自定义的排序。比如，如果要倒序排序，我们就可以自定义一个reversed\_cmp函数：

**def** **reversed\_cmp**(x, y):

**if** x > y:

**return** -1

**if** x < y:

**return** 1

**return** 0

传入自定义的比较函数reversed\_cmp，就可以实现倒序排序：

>>> sorted([36, 5, 12, 9, 21], reversed\_cmp)

[36, 21, 12, 9, 5]

**def** **cmp\_ignore\_case**(s1, s2):

u1 = s1.upper()

u2 = s2.upper()

**if** u1 < u2:

**return** -1

**if** u1 > u2:

**return** 1

**return** 0

>>> sorted(['bob', 'about', 'Zoo', 'Credit'], cmp\_ignore\_case)

['about', 'bob', 'Credit', 'Zoo']

# 返回函数

### 函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回

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

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

ax = 0

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

ax = ax + n

**return** ax

**return** sum

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

>>> f

<**function** sum at 0x10452f668>

调用函数f时，才真正计算求和的结果：

>>> f()

25

在函数lazy\_sum中又定义了函数sum，并且，内部函数sum可以引用外部函数lazy\_sum的参数和局部变量，当lazy\_sum返回函数sum时，相关参数和变量都保存在返回的函数中, 种称为“闭包（Closure）”

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

# 闭包

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

fs = []

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

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

**return** i\*i

fs.append(f)

**return** fs

f1, f2, f3 = count()

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：

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

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

# 匿名函数

关键字lambda表示匿名函数，冒号前面的表示函数参数。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 0x10453d7d0>

>>> 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

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

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

... **print** '2013-12-25'

>>> f = now

>>> f()

2013-12-25

**函数对象有一个\_\_name\_\_属性，可以拿到函数的名字**：

>>> now.\_\_name\_\_

'now'

>>> f.\_\_name\_\_

'now'

**要增强函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改函数的定义，这种在代码运行期间动态增加功能的方式**，称之为“装饰器”（Decorator）。**本质上，decorator就是一个返回函数的高阶函数。**

**def** **log**(func):

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

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

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

**return** wrapper

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

@log

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

**print** '2013-12-25'

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

>>> now()

**call** now():

2013-12-25

把@log放到now()函数的定义处，相当于执行了语句：

now = log(now)

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

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

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

**def** **log**(text):

**def** **decorator**(func):

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

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

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

**return** wrapper

**return** decorator

@log('execute')

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

**print** '2013-12-25'

和两层嵌套的decorator相比，3层嵌套的效果是这样的：

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

首先执行log('execute')，返回的是decorator函数，再调用返回的函数，参数是now函数，返回值最终是wrapper函数。

因为我们讲了函数也是对象，它有\_\_name\_\_等属性，但你去看经过decorator装饰之后的函数，它们的\_\_name\_\_已经从原来的'now'变成了'wrapper'：

>>> now.\_\_name\_\_

'wrapper'

因为返回的那个wrapper()函数名字就是'wrapper'，所以，需要把原始函数的\_\_name\_\_等属性复制到wrapper()函数中

Python内置的functools.wraps

**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

### 小结

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。

decorator可以增强函数的功能，定义起来虽然有点复杂，但使用起来非常灵活和方便。

# 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）

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：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

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：

**def** **int2**(x, base=2):

**return** int(x, base)

functools.partial

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

>>> **import** functools

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

>>> int2('1000000')

### 小结

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

# 模块

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

### 别名

导入模块时，还可以使用别名，这样，可以在运行时根据当前环境选择最合适的模块。比如Python标准库一般会提供StringIO和cStringIO两个库，这两个库的接口和功能是一样的，但是cStringIO是C写的，速度更快，所以，你会经常看到这样的写法：

**try**:

**import** cStringIO **as** StringIO

**except** ImportError: *# 导入失败会捕获到ImportError*

**import** StringIO

这样就可以优先导入cStringIO。如果有些平台不提供cStringIO，还可以降级使用StringIO。导入cStringIO时，用import ... as ...指定了别名StringIO，因此，后续代码引用StringIO即可正常工作。

还有类似simplejson这样的库，在Python 2.6之前是独立的第三方库，从2.6开始内置，所以，会有这样的写法：

**try**:

**import** json *# python >= 2.6*

**except** ImportError:

**import** simplejson **as** json *# python <= 2.5*

由于Python是动态语言，函数签名一致接口就一样，因此，无论导入哪个模块后续代码都能正常工作。

# 作用域

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

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

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

类似\_xxx和\_\_xxx这样的函数或变量就是非公开的（private），不应该被直接引用，比如\_abc，\_\_abc等；外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

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

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

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_，在Python中，实例的变量名如果以\_\_开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问，所以，我们把Student类改一改。不能直接访问\_\_name是因为Python解释器对外把\_\_name变量改成了\_Student\_\_name，所以，仍然可以通过\_Student\_\_name来访问\_\_name变量

# 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()**

但是type()函数返回的是什么类型呢？它返回type类型。Python把每种type类型都定义好了常量，放在types模块里，使用之前，需要先导入：

>>> **import** types

>>> type('abc')==types.StringType

True

>>> type(u'abc')==types.UnicodeType

True

>>> type([])==types.ListType

True

>>> type(str)==types.TypeType

True

有一种类型就叫TypeType，所有类型本身的类型就是TypeType，

### 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()判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上。

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

>>> isinstance('a', (str, unicode))

### 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：

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：

>>> 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

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 0x108ca35d0>>

>>> 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 0x108ca35d0>>

>>> fn() *# 调用fn()与调用obj.power()是一样的*

81

# 使用\_\_slots\_\_

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

>>> **class Student(object):**

... **pass**

尝试给实例绑定一个属性：

>>> s = Student()

>>> s.name = 'Michael' *# 动态给实例绑定一个属性*

>>> **print** s.name

Michael

还可以尝试给实例绑定一个方法：

>>> **def** **set\_age**(self, age): *# 定义一个函数作为实例方法*

... self.age = age

>>> **from** types **import** MethodType

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

>>> s.set\_age(25) *# 调用实例方法*

>>> s.age *# 测试结果*

25

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

为了给所有实例都绑定方法，可以给class绑定方法：

>>> **def** **set\_score**(self, score):

... self.score = score

>>> Student.set\_score = MethodType(set\_score, None, Student)

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

### 使用\_\_slots\_\_

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

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的\_\_slots\_\_变量，来限制该class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*

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

# 使用@property

有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？

对于类的方法，装饰器一样起作用。Python内置的@property装饰器就是负责把一个方法变成属性调用的：

**class Student(object):**

@property

**def** **score**(self):

**return** self.\_score

@score.setter

**def** **score**(self, value):

**if** **not** isinstance(value, int):

**raise** ValueError('score must be an integer!')

**if** value < 0 **or** value > 100:

**raise** ValueError('score must between 0 ~ 100!')

self.\_score = value

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

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

# 多重继承

现在，我们要给动物再加上Runnable和Flyable的功能，只需要先定义好Runnable和Flyable的类：

**class Runnable(object):**

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

print('Running...')

**class Flyable(object):**

**def** **fly**(self):

print('Flying...')

对于需要Runnable功能的动物，就多继承一个Runnable，例如Dog：

**class Dog(Mammal, Runnable):**

**pass**

对于需要Flyable功能的动物，就多继承一个Flyable，例如Bat：

**class Bat(Mammal, Flyable):**

**pass**

通过多重继承，一个子类就可以同时获得多个父类的所有功能。

# Mixin

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

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

**class Dog(Mammal, RunnableMixin, CarnivorousMixin):**

**pass**

Python自带了TCPServer和UDPServer这两类网络服务，而要同时服务多个用户就必须使用多进程或多线程模型，这两种模型由ForkingMixin和ThreadingMixin提供。通过组合，我们就可以创造出合适的服务来。

比如，编写一个多进程模式的TCP服务，定义如下：

**class MyTCPServer(TCPServer, ForkingMixin):**

**pass**

编写一个多线程模式的UDP服务，定义如下：

**class MyUDPServer(UDPServer, ThreadingMixin):**

**pass**

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

**class MyTCPServer(TCPServer, CoroutineMixin):**

**pass**

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

# 定制类

### \_\_str\_\_

我们先定义一个Student类，打印一个实例：

只需要定义好\_\_str\_\_()方法，返回一个好看的字符串就可以了：

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。

**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*

### \_\_getitem\_\_

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):

**if** isinstance(n, int):

a, b = 1, 1

**for** x **in** range(n):

a, b = b, a + b

**return** a

**if** isinstance(n, slice):

start = n.start

stop = n.stop

a, b = 1, 1

L = []

**for** x **in** range(stop):

**if** x >= start:

L.append(a)

a, b = b, a + b

**return** L

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

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

### \_\_getattr\_\_

那就是写一个\_\_getattr\_\_()方法，动态返回一个属性。修改如下：

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))

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

**return** self.\_path

试试：

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

'/status/user/timeline/list'

### \_\_call\_\_

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

**class Student(object):**

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

self.name = name

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

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

# 使用元类

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

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

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

>>> def fn(self, name='world'): # 先定义函数

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

>>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello **class**

**>>> h = Hello()**

**>>> h.hello()**

**Hello, world.**

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。

### 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。

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

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是创建类，所以必须从`type`类型派生：*

**class ListMetaclass(type):**

**def** **\_\_new\_\_**(cls, name, bases, attrs):

attrs['add'] = **lambda** self, value: self.append(value)

**return** type.\_\_new\_\_(cls, name, bases, attrs)

**class MyList(list):**

\_\_metaclass\_\_ = ListMetaclass *# 指示使用ListMetaclass来定制类*

当我们写下\_\_metaclass\_\_ = ListMetaclass语句时，魔术就生效了，它指示Python解释器在创建MyList时，要通过ListMetaclass.\_\_new\_\_()来创建，在此，我们可以修改类的定义，比如，加上新的方法，然后，返回修改后的定义。

\_\_new\_\_()方法接收到的参数依次是：

1. 当前准备创建的类的对象；
2. 类的名字；
3. 类继承的父类集合；
4. 类的方法集合

ORM全称“Object Relational Mapping”，即对象-关系映射，就是把关系数据库的一行映射为一个对象，也就是一个类对应一个表，这样，写代码更简单，不用直接操作SQL语句。

要编写一个ORM框架，所有的类都只能动态定义，因为只有使用者才能根据表的结构定义出对应的类来。

### 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。

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。

# 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。

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。

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。

# WSGI：Web Server Gateway Interface。

Flask

* [Django](https://www.djangoproject.com/)：全能型Web框架；
* [web.py](http://webpy.org/)：一个小巧的Web框架；
* [Bottle](http://bottlepy.org/)：和Flask类似的Web框架；
* [Tornado](http://www.tornadoweb.org/)：Facebook的开源异步Web框架。

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。

# PYTHON缺点：

第一个缺点就是运行速度慢，和C程序相比非常慢，因为Python是解释型语言，你的代码在执行时会一行一行地翻译成CPU能理解的机器码，这个翻译过程非常耗时，所以很慢。而C程序是运行前直接编译成CPU能执行的机器码，所以非常快。

第二个缺点就是代码不能加密。如果要发布你的Python程序，实际上就是发布源代码，这一点跟C语言不同，C语言不用发布源代码，只需要把编译后的机器码（也就是你在Windows上常见的xxx.exe文件）发布出去。要从机器码反推出C代码是不可能的，所以，凡是编译型的语言，都没有这个问题，而解释型的语言，则必须把源码发布出去。

# 爬虫的效率

爬虫下载慢主要原因是阻塞等待发往网站的请求和网站返回

考虑用多进程+分布在不同机房的集群。  
  
理由如下：  
如果单进程，则瓶颈多出在CPU上。  
  
多进程的话可以高效利用CPU。但是其实多数情况是在网络，所以说更好的解决办法是用多个机房的多台机器同时跑多进程的爬虫，这样减少网络阻塞。  
  
实现的话，用scrapy+rq-queue然后用redis来作队列就好。

Scrapy原生支持多线程，还可以设定抓取速率，并发线程数等等参数；除此之外，scrapy对爬虫提取HTML内容也有良好的支持。

1 第一次先请求某个网页，抓取到本地，假设文件名为 a.html。这时文件系统有个文件的修改时间。  
  
2 第二次访问网页，如果发现本地已经有了 a.html，则向服务器发送一个 If-Modified-Since 的请求（[http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html](//link.zhihu.com/?target=http%3A//www.w3.org/Protocols/rfc2616/rfc2616-sec14.html)）。 把 a.html 的修改时间写到请求里。  
  
3 如果网页更新了，服务器会返回一个 200 的应答，这时就重新抓取网页，更新本地文件。  
  
4 如果网页没有更新，服务器会返回一个304的应答。这时就不需要更新文件了。

网页解析工具

正则表达式(模式匹配)

html.parser(结构化解析-DOM)

Beautiful Soup

Lxml

beautifulsoup的源码，其实现原理使用的是正则表达式，而lxml使用的节点递归的技术。

urllib 和urllib2都是接受URL请求的相关模块，但是提供了不同的功能。两个最显著的不同如下：

urllib2可以接受一个**Request**类的实例来设置URL请求的headers，urllib仅可以接受URL。这意味着，你不可以伪装你的User Agent字符串等。

urllib提供**urlencode**方法用来GET查询字符串的产生，而urllib2没有。这是为何urllib常和urllib2一起使用的原因。

[xrange](http://www.nowamagic.net/academy/tag/xrange) 用法与 range 完全相同，所不同的是生成的不是一个list对象，而是一个生成器。

要生成很大的数字序列的时候，用xrange会比range性能优很多，因为不需要一上来就开辟一块很大的内存空间。

# 深浅拷贝

1. 直接赋值,传递对象的引用而已,原始列表改变，被赋值的b也会做相同的改变
2. copy浅拷贝，没有拷贝子对象，所以原始数据改变，子对象会改变

c=copy.copy(alist) **[1, 2, 3, ['a', 'b', 'cccc']] 里面的子对象被改变了**

1. 深拷贝，包含对象里面的自对象的拷贝，所以原始对象的改变不会造成深拷贝里任何子元素的改变

copy.deepcopy(alist)

# 对DIC进行排序

>>> import operator

>>> x={1:2,3:4,4:3,2:1,0:0}

>>> sorted\_x=sorted(x.items(),key=operator.itemgetter(1))

>>> sorted\_x

[(0, 0), (2, 1), (1, 2), (4, 3), (3, 4)]

>>> sorted([3,41,51,2,3],reverse=True)

[51, 41, 3, 3, 2]

>>> sorted(x.items(),key=lambda x:x[1])

[(0, 0), (2, 1), (1, 2), (4, 3), (3, 4)]