**[Python线程指南](http://www.cnblogs.com/huxi/archive/2010/06/26/1765808.html)**

本文介绍了Python对于线程的支持，包括“学会”多线程编程需要掌握的基础以及Python两个线程标准库的完整介绍及使用示例。

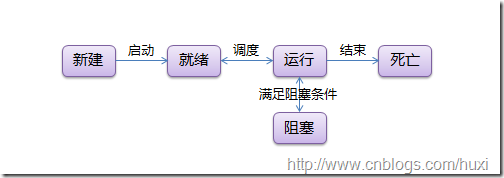
注意：本文基于Python2.4完成，；如果看到不明白的词汇请记得百度谷歌或维基，whatever。

尊重作者的劳动，转载请注明作者及原文地址 >.<

**1. 线程基础**

**1.1. 线程状态**

线程有5种状态，状态转换的过程如下图所示：

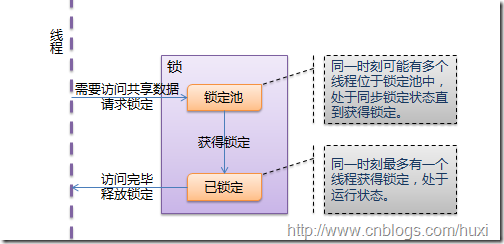


**1.2. 线程同步（锁）**

多线程的优势在于可以同时运行多个任务（至少感觉起来是这样）。但是当线程需要共享数据时，可能存在数据不同步的问题。考虑这样一种情况：一个列表里所有元素都是0，线程"set"从后向前把所有元素改成1，而线程"print"负责从前往后读取列表并打印。那么，可能线程"set"开始改的时候，线程"print"便来打印列表了，输出就成了一半0一半1，这就是数据的不同步。为了避免这种情况，引入了锁的概念。

锁有两种状态——锁定和未锁定。每当一个线程比如"set"要访问共享数据时，必须先获得锁定；如果已经有别的线程比如"print"获得锁定了，那么就让线程"set"暂停，也就是同步阻塞；等到线程"print"访问完毕，释放锁以后，再让线程"set"继续。经过这样的处理，打印列表时要么全部输出0，要么全部输出1，不会再出现一半0一半1的尴尬场面。

线程与锁的交互如下图所示：

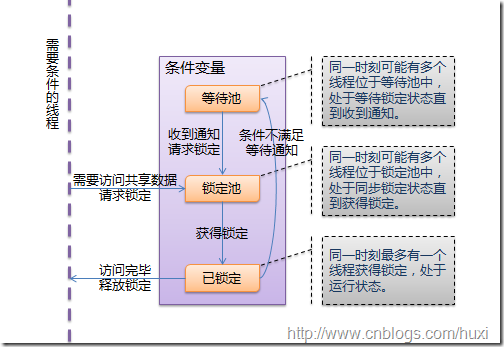


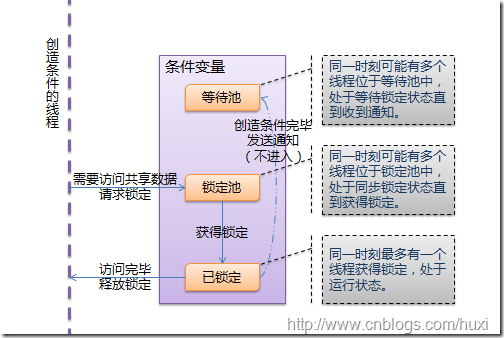
**1.3. 线程通信（条件变量）**

然而还有另外一种尴尬的情况：列表并不是一开始就有的；而是通过线程"create"创建的。如果"set"或者"print" 在"create"还没有运行的时候就访问列表，将会出现一个异常。使用锁可以解决这个问题，但是"set"和"print"将需要一个无限循环——他们不知道"create"什么时候会运行，让"create"在运行后通知"set"和"print"显然是一个更好的解决方案。于是，引入了条件变量。

条件变量允许线程比如"set"和"print"在条件不满足的时候（列表为None时）等待，等到条件满足的时候（列表已经创建）发出一个通知，告诉"set" 和"print"条件已经有了，你们该起床干活了；然后"set"和"print"才继续运行。

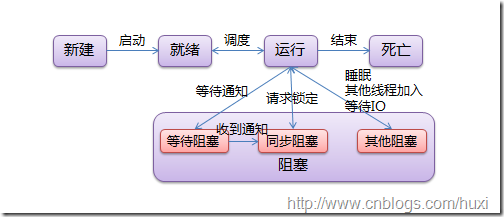
线程与条件变量的交互如下图所示：





**1.4. 线程运行和阻塞的状态转换**

最后看看线程运行和阻塞状态的转换。



阻塞有三种情况：   
同步阻塞是指处于竞争锁定的状态，线程请求锁定时将进入这个状态，一旦成功获得锁定又恢复到运行状态；   
等待阻塞是指等待其他线程通知的状态，线程获得条件锁定后，调用“等待”将进入这个状态，一旦其他线程发出通知，线程将进入同步阻塞状态，再次竞争条件锁定；   
而其他阻塞是指调用time.sleep()、anotherthread.join()或等待IO时的阻塞，这个状态下线程不会释放已获得的锁定。

tips: 如果能理解这些内容，接下来的主题将是非常轻松的；并且，这些内容在大部分流行的编程语言里都是一样的。（意思就是非看懂不可 >\_< 嫌作者水平低找别人的教程也要看懂）

**2. thread**

Python通过两个标准库thread和threading提供对线程的支持。thread提供了低级别的、原始的线程以及一个简单的锁。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | # encoding: UTF-8  import thread  import time    # 一个用于在线程中执行的函数  def func():      for i in range(5):          print 'func'          time.sleep(1)        # 结束当前线程      # 这个方法与thread.exit\_thread()等价      thread.exit() # 当func返回时，线程同样会结束    # 启动一个线程，线程立即开始运行  # 这个方法与thread.start\_new\_thread()等价  # 第一个参数是方法，第二个参数是方法的参数  thread.start\_new(func, ()) # 方法没有参数时需要传入空tuple    # 创建一个锁（LockType，不能直接实例化）  # 这个方法与thread.allocate\_lock()等价  lock = thread.allocate()    # 判断锁是锁定状态还是释放状态  print lock.locked()    # 锁通常用于控制对共享资源的访问  count = 0    # 获得锁，成功获得锁定后返回True  # 可选的timeout参数不填时将一直阻塞直到获得锁定  # 否则超时后将返回False  if lock.acquire():      count += 1        # 释放锁      lock.release()    # thread模块提供的线程都将在主线程结束后同时结束  time.sleep(6) |

**thread 模块提供的其他方法：**thread.interrupt\_main(): 在其他线程中终止主线程。   
thread.get\_ident(): 获得一个代表当前线程的魔法数字，常用于从一个字典中获得线程相关的数据。这个数字本身没有任何含义，并且当线程结束后会被新线程复用。

thread还提供了一个ThreadLocal类用于管理线程相关的数据，名为 thread.\_local，threading中引用了这个类。

由于thread提供的线程功能不多，无法在主线程结束后继续运行，不提供条件变量等等原因，一般不使用thread模块，这里就不多介绍了。

**3. threading**

threading基于Java的线程模型设计。锁（Lock）和条件变量（Condition）在Java中是对象的基本行为（每一个对象都自带了锁和条件变量），而在Python中则是独立的对象。Python Thread提供了Java Thread的行为的子集；没有优先级、线程组，线程也不能被停止、暂停、恢复、中断。Java Thread中的部分被Python实现了的静态方法在threading中以模块方法的形式提供。

**threading 模块提供的常用方法：**   
threading.currentThread(): 返回当前的线程变量。   
threading.enumerate(): 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。   
threading.activeCount(): 返回正在运行的线程数量，与len(threading.enumerate())有相同的结果。

**threading模块提供的类：**Thread, Lock, Rlock, Condition, [Bounded]Semaphore, Event, Timer, local.

**3.1. Thread**

Thread是线程类，与Java类似，有两种使用方法，直接传入要运行的方法或从Thread继承并覆盖run()：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | # encoding: UTF-8  import threading    # 方法1：将要执行的方法作为参数传给Thread的构造方法  def func():      print 'func() passed to Thread'    t = threading.Thread(target=func)  t.start()    # 方法2：从Thread继承，并重写run()  class MyThread(threading.Thread):      def run(self):          print 'MyThread extended from Thread'    t = MyThread()  t.start() |

**构造方法：**Thread(group=None, target=None, name=None, args=(), kwargs={})   
group: 线程组，目前还没有实现，库引用中提示必须是None；   
target: 要执行的方法；   
name: 线程名；   
args/kwargs: 要传入方法的参数。

**实例方法：**isAlive(): 返回线程是否在运行。正在运行指启动后、终止前。   
get/setName(name): 获取/设置线程名。   
is/setDaemon(bool): 获取/设置是否守护线程。初始值从创建该线程的线程继承。当没有非守护线程仍在运行时，程序将终止。   
start(): 启动线程。   
join([timeout]): 阻塞当前上下文环境的线程，直到调用此方法的线程终止或到达指定的timeout（可选参数）。

一个使用join()的例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | # encoding: UTF-8  import threading  import time    def context(tJoin):      print 'in threadContext.'      tJoin.start()        # 将阻塞tContext直到threadJoin终止。      tJoin.join()        # tJoin终止后继续执行。      print 'out threadContext.'    def join():      print 'in threadJoin.'      time.sleep(1)      print 'out threadJoin.'    tJoin = threading.Thread(target=join)  tContext = threading.Thread(target=context, args=(tJoin,))    tContext.start() |

运行结果：

in threadContext.   
in threadJoin.   
out threadJoin.   
out threadContext.

**3.2. Lock**

Lock（指令锁）是可用的最低级的同步指令。Lock处于锁定状态时，不被特定的线程拥有。Lock包含两种状态——锁定和非锁定，以及两个基本的方法。

可以认为Lock有一个锁定池，当线程请求锁定时，将线程至于池中，直到获得锁定后出池。池中的线程处于状态图中的同步阻塞状态。

**构造方法：**Lock()

**实例方法：**acquire([timeout]): 使线程进入同步阻塞状态，尝试获得锁定。   
release(): 释放锁。使用前线程必须已获得锁定，否则将抛出异常。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | # encoding: UTF-8  import threading  import time    data = 0  lock = threading.Lock()    def func():      global data      print '%s acquire lock...' % threading.currentThread().getName()        # 调用acquire([timeout])时，线程将一直阻塞，      # 直到获得锁定或者直到timeout秒后（timeout参数可选）。      # 返回是否获得锁。      if lock.acquire():          print '%s get the lock.' % threading.currentThread().getName()          data += 1          time.sleep(2)          print '%s release lock...' % threading.currentThread().getName()            # 调用release()将释放锁。          lock.release()    t1 = threading.Thread(target=func)  t2 = threading.Thread(target=func)  t3 = threading.Thread(target=func)  t1.start()  t2.start()  t3.start() |

**3.3. RLock**

RLock（可重入锁）是一个可以被同一个线程请求多次的同步指令。RLock使用了“拥有的线程”和“递归等级”的概念，处于锁定状态时，RLock被某个线程拥有。拥有RLock的线程可以再次调用acquire()，释放锁时需要调用release()相同次数。

可以认为RLock包含一个锁定池和一个初始值为0的计数器，每次成功调用 acquire()/release()，计数器将+1/-1，为0时锁处于未锁定状态。

**构造方法：**RLock()

**实例方法：**acquire([timeout])/release(): 跟Lock差不多。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | # encoding: UTF-8  import threading  import time    rlock = threading.RLock()    def func():      # 第一次请求锁定      print '%s acquire lock...' % threading.currentThread().getName()      if rlock.acquire():          print '%s get the lock.' % threading.currentThread().getName()          time.sleep(2)            # 第二次请求锁定          print '%s acquire lock again...' % threading.currentThread().getName()          if rlock.acquire():              print '%s get the lock.' % threading.currentThread().getName()              time.sleep(2)            # 第一次释放锁          print '%s release lock...' % threading.currentThread().getName()          rlock.release()          time.sleep(2)            # 第二次释放锁          print '%s release lock...' % threading.currentThread().getName()          rlock.release()    t1 = threading.Thread(target=func)  t2 = threading.Thread(target=func)  t3 = threading.Thread(target=func)  t1.start()  t2.start()  t3.start() |

**3.4. Condition**

Condition（条件变量）通常与一个锁关联。需要在多个Contidion中共享一个锁时，可以传递一个Lock/RLock实例给构造方法，否则它将自己生成一个RLock实例。

可以认为，除了Lock带有的锁定池外，Condition还包含一个等待池，池中的线程处于状态图中的等待阻塞状态，直到另一个线程调用notify()/notifyAll()通知；得到通知后线程进入锁定池等待锁定。

**构造方法：**Condition([lock/rlock])

**实例方法：**acquire([timeout])/release(): 调用关联的锁的相应方法。   
wait([timeout]): 调用这个方法将使线程进入Condition的等待池等待通知，并释放锁。使用前线程必须已获得锁定，否则将抛出异常。   
notify(): 调用这个方法将从等待池挑选一个线程并通知，收到通知的线程将自动调用acquire()尝试获得锁定（进入锁定池）；其他线程仍然在等待池中。调用这个方法不会释放锁定。使用前线程必须已获得锁定，否则将抛出异常。   
notifyAll(): 调用这个方法将通知等待池中所有的线程，这些线程都将进入锁定池尝试获得锁定。调用这个方法不会释放锁定。使用前线程必须已获得锁定，否则将抛出异常。

例子是很常见的生产者/消费者模式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | # encoding: UTF-8  import threading  import time    # 商品  product = None  # 条件变量  con = threading.Condition()    # 生产者方法  def produce():      global product        if con.acquire():          while True:              if product is None:                  print 'produce...'                  product = 'anything'                    # 通知消费者，商品已经生产                  con.notify()                # 等待通知              con.wait()              time.sleep(2)    # 消费者方法  def consume():      global product        if con.acquire():          while True:              if product is not None:                  print 'consume...'                  product = None                    # 通知生产者，商品已经没了                  con.notify()                # 等待通知              con.wait()              time.sleep(2)    t1 = threading.Thread(target=produce)  t2 = threading.Thread(target=consume)  t2.start()  t1.start() |

**3.5. Semaphore/BoundedSemaphore**

Semaphore（信号量）是计算机科学史上最古老的同步指令之一。Semaphore管理一个内置的计数器，每当调用acquire()时-1，调用release() 时+1。计数器不能小于0；当计数器为0时，acquire()将阻塞线程至同步锁定状态，直到其他线程调用release()。

基于这个特点，Semaphore经常用来同步一些有“访客上限”的对象，比如连接池。

BoundedSemaphore 与Semaphore的唯一区别在于前者将在调用release()时检查计数器的值是否超过了计数器的初始值，如果超过了将抛出一个异常。

**构造方法：**   
Semaphore(value=1): value是计数器的初始值。

**实例方法：**acquire([timeout]): 请求Semaphore。如果计数器为0，将阻塞线程至同步阻塞状态；否则将计数器-1并立即返回。   
release(): 释放Semaphore，将计数器+1，如果使用BoundedSemaphore，还将进行释放次数检查。release()方法不检查线程是否已获得 Semaphore。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | # encoding: UTF-8  import threading  import time    # 计数器初值为2  semaphore = threading.Semaphore(2)    def func():        # 请求Semaphore，成功后计数器-1；计数器为0时阻塞      print '%s acquire semaphore...' % threading.currentThread().getName()      if semaphore.acquire():            print '%s get semaphore' % threading.currentThread().getName()          time.sleep(4)            # 释放Semaphore，计数器+1          print '%s release semaphore' % threading.currentThread().getName()          semaphore.release()    t1 = threading.Thread(target=func)  t2 = threading.Thread(target=func)  t3 = threading.Thread(target=func)  t4 = threading.Thread(target=func)  t1.start()  t2.start()  t3.start()  t4.start()    time.sleep(2)    # 没有获得semaphore的主线程也可以调用release  # 若使用BoundedSemaphore，t4释放semaphore时将抛出异常  print 'MainThread release semaphore without acquire'  semaphore.release() |

**3.6. Event**

Event（事件）是最简单的线程通信机制之一：一个线程通知事件，其他线程等待事件。Event内置了一个初始为False的标志，当调用set()时设为True，调用clear()时重置为 False。wait()将阻塞线程至等待阻塞状态。

Event其实就是一个简化版的 Condition。Event没有锁，无法使线程进入同步阻塞状态。

**构造方法：**Event()

**实例方法：**isSet(): 当内置标志为True时返回True。   
set(): 将标志设为True，并通知所有处于等待阻塞状态的线程恢复运行状态。   
clear(): 将标志设为False。   
wait([timeout]): 如果标志为True将立即返回，否则阻塞线程至等待阻塞状态，等待其他线程调用set()。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | # encoding: UTF-8  import threading  import time    event = threading.Event()    def func():      # 等待事件，进入等待阻塞状态      print '%s wait for event...' % threading.currentThread().getName()      event.wait()        # 收到事件后进入运行状态      print '%s recv event.' % threading.currentThread().getName()    t1 = threading.Thread(target=func)  t2 = threading.Thread(target=func)  t1.start()  t2.start()    time.sleep(2)    # 发送事件通知  print 'MainThread set event.'  event.set() |

**3.7. Timer**

Timer（定时器）是Thread的派生类，用于在指定时间后调用一个方法。

**构造方法：**Timer(interval, function, args=[], kwargs={})   
interval: 指定的时间   
function: 要执行的方法   
args/kwargs: 方法的参数

**实例方法：**Timer从Thread派生，没有增加实例方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # encoding: UTF-8  import threading    def func():      print 'hello timer!'    timer = threading.Timer(5, func)  timer.start() |

**3.8. local**

local是一个小写字母开头的类，用于管理 thread-local（线程局部的）数据。对于同一个local，线程无法访问其他线程设置的属性；线程设置的属性不会被其他线程设置的同名属性替换。

可以把local看成是一个“线程-属性字典”的字典，local封装了从自身使用线程作为 key检索对应的属性字典、再使用属性名作为key检索属性值的细节。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | # encoding: UTF-8  import threading    local = threading.local()  local.tname = 'main'    def func():      local.tname = 'notmain'      print local.tname    t1 = threading.Thread(target=func)  t1.start()  t1.join()    print local.tname |

熟练掌握Thread、Lock、Condition就可以应对绝大多数需要使用线程的场合，某些情况下local也是非常有用的东西。本文的最后使用这几个类展示线程基础中提到的场景：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | # encoding: UTF-8  import threading    alist = None  condition = threading.Condition()    def doSet():      if condition.acquire():          while alist is None:              condition.wait()          for i in range(len(alist))[::-1]:              alist[i] = 1          condition.release()    def doPrint():      if condition.acquire():          while alist is None:              condition.wait()          for i in alist:              print i,          print          condition.release()    def doCreate():      global alist      if condition.acquire():          if alist is None:              alist = [0 for i in range(10)]              condition.notifyAll()          condition.release()    tset = threading.Thread(target=doSet,name='tset')  tprint = threading.Thread(target=doPrint,name='tprint')  tcreate = threading.Thread(target=doCreate,name='tcreate')  tset.start()  tprint.start()  tcreate.start() |

**全文完**

# [Python正则表达式指南](http://www.cnblogs.com/huxi/archive/2010/07/04/1771073.html)

本文介绍了Python对于正则表达式的支持，包括正则表达式基础以及Python正则表达式标准库的完整介绍及使用示例。本文的内容不包括如何编写高效的正则表达式、如何优化正则表达式，这些主题请查看其他教程。

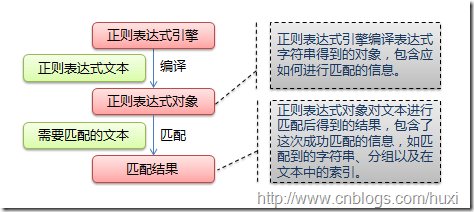
注意：本文基于Python2.4完成；如果看到不明白的词汇请记得百度谷歌或维基，whatever。

尊重作者的劳动，转载请注明作者及原文地址 >.<html

## 1. 正则表达式基础

### 1.1. 简单介绍

正则表达式并不是Python的一部分。正则表达式是用于处理字符串的强大工具，拥有自己独特的语法以及一个独立的处理引擎，效率上可能不如str自带的方法，但功能十分强大。得益于这一点，在提供了正则表达式的语言里，正则表达式的语法都是一样的，区别只在于不同的编程语言实现支持的语法数量不同；但不用担心，不被支持的语法通常是不常用的部分。如果已经在其他语言里使用过正则表达式，只需要简单看一看就可以上手了。

下图展示了使用正则表达式进行匹配的流程：   


正则表达式的大致匹配过程是：依次拿出表达式和文本中的字符比较，如果每一个字符都能匹配，则匹配成功；一旦有匹配不成功的字符则匹配失败。如果表达式中有量词或边界，这个过程会稍微有一些不同，但也是很好理解的，看下图中的示例以及自己多使用几次就能明白。

下图列出了Python支持的正则表达式元字符和语法：     


### 1.2. 数量词的贪婪模式与非贪婪模式

正则表达式通常用于在文本中查找匹配的字符串。Python里数量词默认是贪婪的（在少数语言里也可能是默认非贪婪），总是尝试匹配尽可能多的字符；非贪婪的则相反，总是尝试匹配尽可能少的字符。例如：正则表达式"ab\*"如果用于查找"abbbc"，将找到"abbb"。而如果使用非贪婪的数量词"ab\*?"，将找到"a"。

### 1.3. 反斜杠的困扰

与大多数编程语言相同，正则表达式里使用"\"作为转义字符，这就可能造成反斜杠困扰。假如你需要匹配文本中的字符"\"，那么使用编程语言表示的正则表达式里将需要4个反斜杠"\\\\"：前两个和后两个分别用于在编程语言里转义成反斜杠，转换成两个反斜杠后再在正则表达式里转义成一个反斜杠。Python里的原生字符串很好地解决了这个问题，这个例子中的正则表达式可以使用r"\\"表示。同样，匹配一个数字的"\\d"可以写成r"\d"。有了原生字符串，你再也不用担心是不是漏写了反斜杠，写出来的表达式也更直观。

### 1.4. 匹配模式

正则表达式提供了一些可用的匹配模式，比如忽略大小写、多行匹配等，这部分内容将在Pattern类的工厂方法re.compile(pattern[, flags])中一起介绍。

## 2. re模块

### 2.1. 开始使用re

Python通过re模块提供对正则表达式的支持。使用re的一般步骤是先将正则表达式的字符串形式编译为Pattern实例，然后使用Pattern实例处理文本并获得匹配结果（一个Match实例），最后使用Match实例获得信息，进行其他的操作。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | # encoding: UTF-8  import re    # 将正则表达式编译成Pattern对象  pattern = re.compile(r'hello')    # 使用Pattern匹配文本，获得匹配结果，无法匹配时将返回None  match = pattern.match('hello world!')    if match:      # 使用Match获得分组信息      print match.group()    ### 输出 ###  # hello |

**re.compile(strPattern[, flag]):**

这个方法是Pattern类的工厂方法，用于将字符串形式的正则表达式编译为Pattern对象。 第二个参数flag是匹配模式，取值可以使用按位或运算符'|'表示同时生效，比如re.I | re.M。另外，你也可以在regex字符串中指定模式，比如re.compile('pattern', re.I | re.M)与re.compile('(?im)pattern')是等价的。   
可选值有：

* re.**I**(re.IGNORECASE): 忽略大小写（括号内是完整写法，下同）
* **M**(MULTILINE): 多行模式，改变'^'和'$'的行为（参见上图）
* **S**(DOTALL): 点任意匹配模式，改变'.'的行为
* **L**(LOCALE): 使预定字符类 \w \W \b \B \s \S 取决于当前区域设定
* **U**(UNICODE): 使预定字符类 \w \W \b \B \s \S \d \D 取决于unicode定义的字符属性
* **X**(VERBOSE): 详细模式。这个模式下正则表达式可以是多行，忽略空白字符，并可以加入注释。以下两个正则表达式是等价的：

|  |  |
| --- | --- |
| 1  2  3  4 | a = re.compile(r"""\d +  # the integral part                     \.    # the decimal point                     \d \*  # some fractional digits""", re.X)  b = re.compile(r"\d+\.\d\*") |

re提供了众多模块方法用于完成正则表达式的功能。这些方法可以使用Pattern实例的相应方法替代，唯一的好处是少写一行re.compile()代码，但同时也无法复用编译后的Pattern对象。这些方法将在Pattern类的实例方法部分一起介绍。如上面这个例子可以简写为：

|  |  |
| --- | --- |
| 1  2 | m = re.match(r'hello', 'hello world!')  print m.group() |

re模块还提供了一个方法escape(string)，用于将string中的正则表达式元字符如\*/+/?等之前加上转义符再返回，在需要大量匹配元字符时有那么一点用。

### 2.2. Match

Match对象是一次匹配的结果，包含了很多关于此次匹配的信息，可以使用Match提供的可读属性或方法来获取这些信息。

属性：

1. **string**: 匹配时使用的文本。
2. **re**: 匹配时使用的Pattern对象。
3. **pos**: 文本中正则表达式开始搜索的索引。值与Pattern.match()和Pattern.seach()方法的同名参数相同。
4. **endpos**: 文本中正则表达式结束搜索的索引。值与Pattern.match()和Pattern.seach()方法的同名参数相同。
5. **lastindex**: 最后一个被捕获的分组在文本中的索引。如果没有被捕获的分组，将为None。
6. **lastgroup**: 最后一个被捕获的分组的别名。如果这个分组没有别名或者没有被捕获的分组，将为None。

方法：

1. **group([group1, …]):**  
   获得一个或多个分组截获的字符串；指定多个参数时将以元组形式返回。group1可以使用编号也可以使用别名；编号0代表整个匹配的子串；不填写参数时，返回group(0)；没有截获字符串的组返回None；截获了多次的组返回最后一次截获的子串。
2. **groups([default]):**   
   以元组形式返回全部分组截获的字符串。相当于调用group(1,2,…last)。default表示没有截获字符串的组以这个值替代，默认为None。
3. **groupdict([default]):**返回以有别名的组的别名为键、以该组截获的子串为值的字典，没有别名的组不包含在内。default含义同上。
4. **start([group]):**   
   返回指定的组截获的子串在string中的起始索引（子串第一个字符的索引）。group默认值为0。
5. **end([group]):**返回指定的组截获的子串在string中的结束索引（子串最后一个字符的索引+1）。group默认值为0。
6. **span([group]):**返回(start(group), end(group))。
7. **expand(template):**   
   将匹配到的分组代入template中然后返回。template中可以使用\id或\g<id>、\g<name>引用分组，但不能使用编号0。\id与\g<id>是等价的；但\10将被认为是第10个分组，如果你想表达\1之后是字符'0'，只能使用\g<1>0。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | import re  m = re.match(r'(\w+) (\w+)(?P<sign>.\*)', 'hello world!')    print "m.string:", m.string  print "m.re:", m.re  print "m.pos:", m.pos  print "m.endpos:", m.endpos  print "m.lastindex:", m.lastindex  print "m.lastgroup:", m.lastgroup    print "m.group(1,2):", m.group(1, 2)  print "m.groups():", m.groups()  print "m.groupdict():", m.groupdict()  print "m.start(2):", m.start(2)  print "m.end(2):", m.end(2)  print "m.span(2):", m.span(2)  print r"m.expand(r'\2 \1\3'):", m.expand(r'\2 \1\3')    ### output ###  # m.string: hello world!  # m.re: <\_sre.SRE\_Pattern object at 0x016E1A38>  # m.pos: 0  # m.endpos: 12  # m.lastindex: 3  # m.lastgroup: sign  # m.group(1,2): ('hello', 'world')  # m.groups(): ('hello', 'world', '!')  # m.groupdict(): {'sign': '!'}  # m.start(2): 6  # m.end(2): 11  # m.span(2): (6, 11)  # m.expand(r'\2 \1\3'): world hello! |

### 2.3. Pattern

Pattern对象是一个编译好的正则表达式，通过Pattern提供的一系列方法可以对文本进行匹配查找。

Pattern不能直接实例化，必须使用re.compile()进行构造。

Pattern提供了几个可读属性用于获取表达式的相关信息：

1. pattern: 编译时用的表达式字符串。
2. flags: 编译时用的匹配模式。数字形式。
3. groups: 表达式中分组的数量。
4. groupindex: 以表达式中有别名的组的别名为键、以该组对应的编号为值的字典，没有别名的组不包含在内。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import re  p = re.compile(r'(\w+) (\w+)(?P<sign>.\*)', re.DOTALL)    print "p.pattern:", p.pattern  print "p.flags:", p.flags  print "p.groups:", p.groups  print "p.groupindex:", p.groupindex    ### output ###  # p.pattern: (\w+) (\w+)(?P<sign>.\*)  # p.flags: 16  # p.groups: 3  # p.groupindex: {'sign': 3} |

实例方法[ | re模块方法]：

1. **match(string[, pos[, endpos]]) | re.match(pattern, string[, flags]):**这个方法将从string的pos下标处起尝试匹配pattern；如果pattern结束时仍可匹配，则返回一个Match对象；如果匹配过程中pattern无法匹配，或者匹配未结束就已到达endpos，则返回None。   
   pos和endpos的默认值分别为0和len(string)；re.match()无法指定这两个参数，参数flags用于编译pattern时指定匹配模式。   
   注意：这个方法并不是完全匹配。当pattern结束时若string还有剩余字符，仍然视为成功。想要完全匹配，可以在表达式末尾加上边界匹配符'$'。   
   示例参见2.1小节。
2. **search(string[, pos[, endpos]]) | re.search(pattern, string[, flags]):**这个方法用于查找字符串中可以匹配成功的子串。从string的pos下标处起尝试匹配pattern，如果pattern结束时仍可匹配，则返回一个Match对象；若无法匹配，则将pos加1后重新尝试匹配；直到pos=endpos时仍无法匹配则返回None。   
   pos和endpos的默认值分别为0和len(string))；re.search()无法指定这两个参数，参数flags用于编译pattern时指定匹配模式。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | # encoding: UTF-8  import re    # 将正则表达式编译成Pattern对象  pattern = re.compile(r'world')    # 使用search()查找匹配的子串，不存在能匹配的子串时将返回None  # 这个例子中使用match()无法成功匹配  match = pattern.search('hello world!')    if match:      # 使用Match获得分组信息      print match.group()    ### 输出 ###  # world |

1. **split(string[, maxsplit]) | re.split(pattern, string[, maxsplit]):**按照能够匹配的子串将string分割后返回列表。maxsplit用于指定最大分割次数，不指定将全部分割。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import re    p = re.compile(r'\d+')  print p.split('one1two2three3four4')    ### output ###  # ['one', 'two', 'three', 'four', ''] |

1. **findall(string[, pos[, endpos]]) | re.findall(pattern, string[, flags]):**搜索string，以列表形式返回全部能匹配的子串。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import re    p = re.compile(r'\d+')  print p.findall('one1two2three3four4')    ### output ###  # ['1', '2', '3', '4'] |

1. **finditer(string[, pos[, endpos]]) | re.finditer(pattern, string[, flags]):**搜索string，返回一个顺序访问每一个匹配结果（Match对象）的迭代器。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | import re    p = re.compile(r'\d+')  for m in p.finditer('one1two2three3four4'):      print m.group(),    ### output ###  # 1 2 3 4 |

1. **sub(repl, string[, count]) | re.sub(pattern, repl, string[, count]):**使用repl替换string中每一个匹配的子串后返回替换后的字符串。   
   当repl是一个字符串时，可以使用\id或\g<id>、\g<name>引用分组，但不能使用编号0。   
   当repl是一个方法时，这个方法应当只接受一个参数（Match对象），并返回一个字符串用于替换（返回的字符串中不能再引用分组）。   
   count用于指定最多替换次数，不指定时全部替换。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import re    p = re.compile(r'(\w+) (\w+)')  s = 'i say, hello world!'    print p.sub(r'\2 \1', s)    def func(m):      return m.group(1).title() + ' ' + m.group(2).title()    print p.sub(func, s)    ### output ###  # say i, world hello!  # I Say, Hello World! |

1. **subn(repl, string[, count]) |re.sub(pattern, repl, string[, count]):**返回 (sub(repl, string[, count]), 替换次数)。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import re    p = re.compile(r'(\w+) (\w+)')  s = 'i say, hello world!'    print p.subn(r'\2 \1', s)    def func(m):      return m.group(1).title() + ' ' + m.group(2).title()    print p.subn(func, s)    ### output ###  # ('say i, world hello!', 2)  # ('I Say, Hello World!', 2) |

**以上就是Python对于正则表达式的支持。熟练掌握正则表达式是每一个程序员必须具备的技能，这年头没有不与字符串打交道的程序了。笔者也处于初级阶段，与君共勉，^\_^**

**另外，图中的特殊构造部分没有举出例子，用到这些的正则表达式是具有一定难度的。有兴趣可以思考一下，如何匹配不是以abc开头的单词，^\_^**

**全文结束**

# [Python字符编码详解](http://www.cnblogs.com/huxi/archive/2010/12/05/1897271.html)

本文简单介绍了各种常用的字符编码的特点，并介绍了在python2.x中如何与编码问题作战 ：）   
请注意本文关于Python的内容仅适用于2.x，3.x中str和unicode有翻天覆地的变化，请查阅其他相关文档。   
尊重作者的劳动，转载请注明作者及原文地址 >.<

## 1. 字符编码简介

### 1.1. ASCII

ASCII(American Standard Code for Information Interchange)，是一种单字节的编码。计算机世界里一开始只有英文，而单字节可以表示256个不同的字符，可以表示所有的英文字符和许多的控制符号。不过ASCII只用到了其中的一半（\x80以下），这也是MBCS得以实现的基础。

### 1.2. MBCS

然而计算机世界里很快就有了其他语言，单字节的ASCII已无法满足需求。后来每个语言就制定了一套自己的编码，由于单字节能表示的字符太少，而且同时也需要与ASCII编码保持兼容，所以这些编码纷纷使用了多字节来表示字符，如GBxxx、BIGxxx等等，他们的规则是，如果第一个字节是\x80以下，则仍然表示ASCII字符；而如果是\x80以上，则跟下一个字节一起（共两个字节）表示一个字符，然后跳过下一个字节，继续往下判断。

这里，IBM发明了一个叫Code Page的概念，将这些编码都收入囊中并分配页码，GBK是第936页，也就是CP936。所以，也可以使用CP936表示GBK。

MBCS(Multi-Byte Character Set)是这些编码的统称。目前为止大家都是用了双字节，所以有时候也叫做DBCS(Double-Byte Character Set)。必须明确的是，MBCS并不是某一种特定的编码，Windows里根据你设定的区域不同，MBCS指代不同的编码，而Linux里无法使用MBCS作为编码。在Windows中你看不到MBCS这几个字符，因为微软为了更加洋气，使用了ANSI来吓唬人，记事本的另存为对话框里编码ANSI就是MBCS。同时，在简体中文Windows默认的区域设定里，指代GBK。

### 1.3. Unicode

后来，有人开始觉得太多编码导致世界变得过于复杂了，让人脑袋疼，于是大家坐在一起拍脑袋想出来一个方法：所有语言的字符都用同一种字符集来表示，这就是Unicode。

最初的Unicode标准UCS-2使用两个字节表示一个字符，所以你常常可以听到Unicode使用两个字节表示一个字符的说法。但过了不久有人觉得256\*256太少了，还是不够用，于是出现了UCS-4标准，它使用4个字节表示一个字符，不过我们用的最多的仍然是UCS-2。

UCS(Unicode Character Set)还仅仅是字符对应码位的一张表而已，比如"汉"这个字的码位是6C49。字符具体如何传输和储存则是由UTF(UCS Transformation Format)来负责。

一开始这事很简单，直接使用UCS的码位来保存，这就是UTF-16，比如，"汉"直接使用\x6C\x49保存(UTF-16-BE)，或是倒过来使用\x49\x6C保存(UTF-16-LE)。但用着用着美国人觉得自己吃了大亏，以前英文字母只需要一个字节就能保存了，现在大锅饭一吃变成了两个字节，空间消耗大了一倍……于是UTF-8横空出世。

UTF-8是一种很别扭的编码，具体表现在他是变长的，并且兼容ASCII，ASCII字符使用1字节表示。然而这里省了的必定是从别的地方抠出来的，你肯定也听说过UTF-8里中文字符使用3个字节来保存吧？4个字节保存的字符更是在泪奔……（具体UCS-2是怎么变成UTF-8的请自行搜索）

另外值得一提的是BOM(Byte Order Mark)。我们在储存文件时，文件使用的编码并没有保存，打开时则需要我们记住原先保存时使用的编码并使用这个编码打开，这样一来就产生了许多麻烦。（你可能想说记事本打开文件时并没有让选编码？不妨先打开记事本再使用文件 -> 打开看看）而UTF则引入了BOM来表示自身编码，如果一开始读入的几个字节是其中之一，则代表接下来要读取的文字使用的编码是相应的编码：

BOM\_UTF8 '\xef\xbb\xbf'   
BOM\_UTF16\_LE '\xff\xfe'   
BOM\_UTF16\_BE '\xfe\xff'

并不是所有的编辑器都会写入BOM，但即使没有BOM，Unicode还是可以读取的，只是像MBCS的编码一样，需要另行指定具体的编码，否则解码将会失败。

你可能听说过UTF-8不需要BOM，这种说法是不对的，只是绝大多数编辑器在没有BOM时都是以UTF-8作为默认编码读取。即使是保存时默认使用ANSI(MBCS)的记事本，在读取文件时也是先使用UTF-8测试编码，如果可以成功解码，则使用UTF-8解码。记事本这个别扭的做法造成了一个BUG：如果你新建文本文件并输入"姹塧"然后使用ANSI(MBCS)保存，再打开就会变成"汉a"，你不妨试试 ：）

## 2. Python2.x中的编码问题

### 2.1. str和unicode

str和unicode都是basestring的子类。严格意义上说，str其实是字节串，它是unicode经过编码后的字节组成的序列。对UTF-8编码的str'汉'使用len()函数时，结果是3，因为实际上，UTF-8编码的'汉' == '\xE6\xB1\x89'。

unicode才是真正意义上的字符串，对字节串str使用正确的字符编码进行解码后获得，并且len(u'汉') == 1。

再来看看encode()和decode()两个basestring的实例方法，理解了str和unicode的区别后，这两个方法就不会再混淆了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | # coding: UTF-8    u = u'汉'  print repr(u) # u'\u6c49'  s = u.encode('UTF-8')  print repr(s) # '\xe6\xb1\x89'  u2 = s.decode('UTF-8')  print repr(u2) # u'\u6c49'    # 对unicode进行解码是错误的  # s2 = u.decode('UTF-8')  # 同样，对str进行编码也是错误的  # u2 = s.encode('UTF-8') |

需要注意的是，虽然对str调用encode()方法是错误的，但实际上Python不会抛出异常，而是返回另外一个相同内容但不同id的str；对unicode调用decode()方法也是这样。很不理解为什么不把encode()和decode()分别放在unicode和str中而是都放在basestring中，但既然已经这样了，我们就小心避免犯错吧。

### 2.2. 字符编码声明

源代码文件中，如果有用到非ASCII字符，则需要在文件头部进行字符编码的声明，如下：

|  |  |
| --- | --- |
| 1 | #-\*- coding: UTF-8 -\*- |

实际上Python只检查#、coding和编码字符串，其他的字符都是为了美观加上的。另外，Python中可用的字符编码有很多，并且还有许多别名，还不区分大小写，比如UTF-8可以写成u8。参见<http://docs.python.org/library/codecs.html#standard-encodings>。

另外需要注意的是声明的编码必须与文件实际保存时用的编码一致，否则很大几率会出现代码解析异常。现在的IDE一般会自动处理这种情况，改变声明后同时换成声明的编码保存，但文本编辑器控们需要小心 ：）

### 2.3. 读写文件

内置的open()方法打开文件时，read()读取的是str，读取后需要使用正确的编码格式进行decode()。write()写入时，如果参数是unicode，则需要使用你希望写入的编码进行encode()，如果是其他编码格式的str，则需要先用该str的编码进行decode()，转成unicode后再使用写入的编码进行encode()。如果直接将unicode作为参数传入write()方法，Python将先使用源代码文件声明的字符编码进行编码然后写入。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # coding: UTF-8    f = open('test.txt')  s = f.read()  f.close()  print type(s) # <type 'str'>  # 已知是GBK编码，解码成unicode  u = s.decode('GBK')    f = open('test.txt', 'w')  # 编码成UTF-8编码的str  s = u.encode('UTF-8')  f.write(s)  f.close() |

另外，模块codecs提供了一个open()方法，可以指定一个编码打开文件，使用这个方法打开的文件读取返回的将是unicode。写入时，如果参数是unicode，则使用open()时指定的编码进行编码后写入；如果是str，则先根据源代码文件声明的字符编码，解码成unicode后再进行前述操作。相对内置的open()来说，这个方法比较不容易在编码上出现问题。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | # coding: GBK    import codecs    f = codecs.open('test.txt', encoding='UTF-8')  u = f.read()  f.close()  print type(u) # <type 'unicode'>    f = codecs.open('test.txt', 'a', encoding='UTF-8')  # 写入unicode  f.write(u)    # 写入str，自动进行解码编码操作  # GBK编码的str  s = '汉'  print repr(s) # '\xba\xba'  # 这里会先将GBK编码的str解码为unicode再编码为UTF-8写入  f.write(s)  f.close() |

### 2.4. 与编码相关的方法

sys/locale模块中提供了一些获取当前环境下的默认编码的方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | # coding:gbk    import sys  import locale    def p(f):      print '%s.%s(): %s' % (f.\_\_module\_\_, f.\_\_name\_\_, f())    # 返回当前系统所使用的默认字符编码  p(sys.getdefaultencoding)    # 返回用于转换Unicode文件名至系统文件名所使用的编码  p(sys.getfilesystemencoding)    # 获取默认的区域设置并返回元祖(语言, 编码)  p(locale.getdefaultlocale)    # 返回用户设定的文本数据编码  # 文档提到this function only returns a guess  p(locale.getpreferredencoding)    # \xba\xba是'汉'的GBK编码  # mbcs是不推荐使用的编码，这里仅作测试表明为什么不应该用  print r"'\xba\xba'.decode('mbcs'):", repr('\xba\xba'.decode('mbcs'))    #在笔者的Windows上的结果(区域设置为中文(简体, 中国))  #sys.getdefaultencoding(): gbk  #sys.getfilesystemencoding(): mbcs  #locale.getdefaultlocale(): ('zh\_CN', 'cp936')  #locale.getpreferredencoding(): cp936  #'\xba\xba'.decode('mbcs'): u'\u6c49' |

## 3.一些建议

### 3.1. 使用字符编码声明，并且同一工程中的所有源代码文件使用相同的字符编码声明。

这点是一定要做到的。

### 3.2. 抛弃str，全部使用unicode。

按引号前先按一下u最初做起来确实很不习惯而且经常会忘记再跑回去补，但如果这么做可以减少90%的编码问题。如果编码困扰不严重，可以不参考此条。

### 3.3. 使用codecs.open()替代内置的open()。

如果编码困扰不严重，可以不参考此条。

### 3.4. 绝对需要避免使用的字符编码：MBCS/DBCS和UTF-16。

这里说的MBCS不是指GBK什么的都不能用，而是不要使用Python里名为'MBCS'的编码，除非程序完全不移植。

Python中编码'MBCS'与'DBCS'是同义词，指当前Windows环境中MBCS指代的编码。Linux的Python实现中没有这种编码，所以一旦移植到Linux一定会出现异常！另外，只要设定的Windows系统区域不同，MBCS指代的编码也是不一样的。分别设定不同的区域运行2.4小节中的代码的结果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | #中文(简体, 中国)  #sys.getdefaultencoding(): gbk  #sys.getfilesystemencoding(): mbcs  #locale.getdefaultlocale(): ('zh\_CN', 'cp936')  #locale.getpreferredencoding(): cp936  #'\xba\xba'.decode('mbcs'): u'\u6c49'    #英语(美国)  #sys.getdefaultencoding(): UTF-8  #sys.getfilesystemencoding(): mbcs  #locale.getdefaultlocale(): ('zh\_CN', 'cp1252')  #locale.getpreferredencoding(): cp1252  #'\xba\xba'.decode('mbcs'): u'\xba\xba'    #德语(德国)  #sys.getdefaultencoding(): gbk  #sys.getfilesystemencoding(): mbcs  #locale.getdefaultlocale(): ('zh\_CN', 'cp1252')  #locale.getpreferredencoding(): cp1252  #'\xba\xba'.decode('mbcs'): u'\xba\xba'    #日语(日本)  #sys.getdefaultencoding(): gbk  #sys.getfilesystemencoding(): mbcs  #locale.getdefaultlocale(): ('zh\_CN', 'cp932')  #locale.getpreferredencoding(): cp932  #'\xba\xba'.decode('mbcs'): u'\uff7a\uff7a' |

可见，更改区域后，使用mbcs解码得到了不正确的结果，所以，当我们需要使用'GBK'时，应该直接写'GBK'，不要写成'MBCS'。

UTF-16同理，虽然绝大多数操作系统中'UTF-16'是'UTF-16-LE'的同义词，但直接写'UTF-16-LE'只是多写3个字符而已，而万一某个操作系统中'UTF-16'变成了'UTF-16-BE'的同义词，就会有错误的结果。实际上，UTF-16用的相当少，但用到的时候还是需要注意。

--END--

[**Python自省（反射）指南**](http://www.cnblogs.com/huxi/archive/2011/01/02/1924317.html)

在笔者，也就是我的概念里，自省和反射是一回事，当然其实我并不十分确定一定以及肯定，所以如果这确实是两个不同的概念的话，还请多多指教 ：） 转载请注明作者、出处并附上原文链接，多谢！  
update 2011-3-10: 更正函数的func\_globals属性含义。

首先通过一个例子来看一下本文中可能用到的对象和相关概念。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #coding: UTF-8  import sys #  模块，sys指向这个模块对象  import inspect  def foo(): pass # 函数，foo指向这个函数对象    class Cat(object): # 类，Cat指向这个类对象      def \_\_init\_\_(self, name='kitty'):          self.name = name      def sayHi(self): #  实例方法，sayHi指向这个方法对象，使用类或实例.sayHi访问          print self.name, 'says Hi!' # 访问名为name的字段，使用实例.name访问    cat = Cat() # cat是Cat类的实例对象    print Cat.sayHi # 使用类名访问实例方法时，方法是未绑定的(unbound)  print cat.sayHi # 使用实例访问实例方法时，方法是绑定的(bound) |

有时候我们会碰到这样的需求，需要执行对象的某个方法，或是需要对对象的某个字段赋值，而方法名或是字段名在编码代码时并不能确定，需要通过参数传递字符串的形式输入。举个具体的例子：当我们需要实现一个通用的DBM框架时，可能需要对数据对象的字段赋值，但我们无法预知用到这个框架的数据对象都有些什么字段，换言之，我们在写框架的时候需要通过某种机制访问未知的属性。

这个机制被称为反射（反过来让对象告诉我们他是什么），或是自省（让对象自己告诉我们他是什么，好吧我承认括号里是我瞎掰的- -#），用于实现在运行时获取未知对象的信息。反射是个很吓唬人的名词，听起来高深莫测，在一般的编程语言里反射相对其他概念来说稍显复杂，一般来说都是作为高级主题来讲；但在Python中反射非常简单，用起来几乎感觉不到与其他的代码有区别，使用反射获取到的函数和方法可以像平常一样加上括号直接调用，获取到类后可以直接构造实例；不过获取到的字段不能直接赋值，因为拿到的其实是另一个指向同一个地方的引用，赋值只能改变当前的这个引用而已。

**1. 访问对象的属性**

以下列出了几个内建方法，可以用来检查或是访问对象的属性。这些方法可以用于任意对象而不仅仅是例子中的Cat实例对象；Python中一切都是对象。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | cat = Cat('kitty')    print cat.name # 访问实例属性  cat.sayHi() # 调用实例方法    print dir(cat) # 获取实例的属性名，以列表形式返回  if hasattr(cat, 'name'): # 检查实例是否有这个属性      setattr(cat, 'name', 'tiger') # same as: a.name = 'tiger'  print getattr(cat, 'name') # same as: print a.name    getattr(cat, 'sayHi')() # same as: cat.sayHi() |

* **dir([obj]):**   
  调用这个方法将返回包含obj大多数属性名的列表（会有一些特殊的属性不包含在内）。obj的默认值是当前的模块对象。
* **hasattr(obj, attr):**这个方法用于检查obj是否有一个名为attr的值的属性，返回一个布尔值。
* **getattr(obj, attr):**调用这个方法将返回obj中名为attr值的属性的值，例如如果attr为'bar'，则返回obj.bar。
* **setattr(obj, attr, val):**   
  调用这个方法将给obj的名为attr的值的属性赋值为val。例如如果attr为'bar'，则相当于obj.bar = val。

**2. 访问对象的元数据**

当你对一个你构造的对象使用dir()时，可能会发现列表中的很多属性并不是你定义的。这些属性一般保存了对象的元数据，比如类的\_\_name\_\_属性保存了类名。大部分这些属性都可以修改，不过改动它们意义并不是很大；修改其中某些属性如function.func\_code还可能导致很难发现的问题，所以改改name什么的就好了，其他的属性不要在不了解后果的情况下修改。

接下来列出特定对象的一些特殊属性。另外，Python的文档中有提到部分属性不一定会一直提供，下文中将以红色的星号**\***标记，使用前你可以先打开解释器确认一下。

**2.0. 准备工作：确定对象的类型**

在types模块中定义了全部的Python内置类型，结合内置方法isinstance()就可以确定对象的具体类型了。

* **isinstance(object, classinfo):**   
  检查object是不是classinfo中列举出的类型，返回布尔值。classinfo可以是一个具体的类型，也可以是多个类型的元组或列表。

types模块中仅仅定义了类型，而inspect模块中封装了很多检查类型的方法，比直接使用types模块更为轻松，所以这里不给出关于types的更多介绍，如有需要可以直接查看types模块的文档说明。本文第3节中介绍了inspect模块。

**2.1. 模块(module)**

* \_\_doc\_\_: 文档字符串。如果模块没有文档，这个值是None。
* **\***\_\_name\_\_: 始终是定义时的模块名；即使你使用import .. as 为它取了别名，或是赋值给了另一个变量名。
* **\***\_\_dict\_\_: 包含了模块里可用的属性名-属性的字典；也就是可以使用模块名.属性名访问的对象。
* \_\_file\_\_: 包含了该模块的文件路径。需要注意的是内建的模块没有这个属性，访问它会抛出异常！

|  |  |
| --- | --- |
| 1  2  3  4  5 | import fnmatch as m  print m.\_\_doc\_\_.splitlines()[0] # Filename matching with shell patterns.  print m.\_\_name\_\_                # fnmatch  print m.\_\_file\_\_                # /usr/lib/python2.6/fnmatch.pyc  print m.\_\_dict\_\_.items()[0]     # ('fnmatchcase', <function>)</function> |

**2.2. 类(class)**

* \_\_doc\_\_: 文档字符串。如果类没有文档，这个值是None。
* **\***\_\_name\_\_: 始终是定义时的类名。
* **\***\_\_dict\_\_: 包含了类里可用的属性名-属性的字典；也就是可以使用类名.属性名访问的对象。
* \_\_module\_\_: 包含该类的定义的模块名；需要注意，是字符串形式的模块名而不是模块对象。
* **\***\_\_bases\_\_: 直接父类对象的元组；但不包含继承树更上层的其他类，比如父类的父类。

|  |  |
| --- | --- |
| 1  2  3  4  5 | print Cat.\_\_doc\_\_           # None  print Cat.\_\_name\_\_          # Cat  print Cat.\_\_module\_\_        # \_\_main\_\_  print Cat.\_\_bases\_\_         # (<type>,)  print Cat.\_\_dict\_\_          # {'\_\_module\_\_': '\_\_main\_\_', ...}</type> |

**2.3. 实例(instance)**

实例是指类实例化以后的对象。

* **\***\_\_dict\_\_: 包含了可用的属性名-属性字典。
* **\***\_\_class\_\_: 该实例的类对象。对于类Cat，cat.\_\_class\_\_ == Cat 为 True。

|  |  |
| --- | --- |
| 1  2  3 | print cat.\_\_dict\_\_  print cat.\_\_class\_\_  print cat.\_\_class\_\_ == Cat # True |

**2.4. 内建函数和方法(built-in functions and methods)**

根据定义，内建的(built-in)模块是指使用C写的模块，可以通过sys模块的builtin\_module\_names字段查看都有哪些模块是内建的。这些模块中的函数和方法可以使用的属性比较少，不过一般也不需要在代码中查看它们的信息。

* \_\_doc\_\_: 函数或方法的文档。
* \_\_name\_\_: 函数或方法定义时的名字。
* \_\_self\_\_: 仅方法可用，如果是绑定的(bound)，则指向调用该方法的类（如果是类方法）或实例（如果是实例方法），否则为None。
* **\***\_\_module\_\_: 函数或方法所在的模块名。

**2.5. 函数(function)**

这里特指非内建的函数。注意，在类中使用def定义的是方法，方法与函数虽然有相似的行为，但它们是不同的概念。

* \_\_doc\_\_: 函数的文档；另外也可以用属性名func\_doc。
* \_\_name\_\_: 函数定义时的函数名；另外也可以用属性名func\_name。
* **\***\_\_module\_\_: 包含该函数定义的模块名；同样注意，是模块名而不是模块对象。
* **\***\_\_dict\_\_: 函数的可用属性；另外也可以用属性名func\_dict。   
  不要忘了函数也是对象，可以使用函数.属性名访问属性（赋值时如果属性不存在将新增一个），或使用内置函数has/get/setattr()访问。不过，在函数中保存属性的意义并不大。
* func\_defaults: 这个属性保存了函数的参数默认值元组；因为默认值总是靠后的参数才有，所以不使用字典的形式也是可以与参数对应上的。
* func\_code: 这个属性指向一个该函数对应的code对象，code对象中定义了其他的一些特殊属性，将在下文中另外介绍。
* func\_globals: 这个属性指向定义函数时的全局命名空间。
* **\***func\_closure: 这个属性仅当函数是一个闭包时有效，指向一个保存了所引用到的外部函数的变量cell的元组，如果该函数不是一个内部函数，则始终为None。这个属性也是只读的。

下面的代码演示了func\_closure：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #coding: UTF-8  def foo():      n = 1      def bar():          print n # 引用非全局的外部变量n，构造一个闭包      n = 2      return bar    closure = foo()  print closure.func\_closure  # 使用dir()得知cell对象有一个cell\_contents属性可以获得值  print closure.func\_closure[0].cell\_contents # 2 |

由这个例子可以看到，遇到未知的对象使用dir()是一个很好的主意 ：）

**2.6. 方法(method)**

方法虽然不是函数，但可以理解为在函数外面加了一层外壳；拿到方法里实际的函数以后，就可以使用2.5节的属性了。

* \_\_doc\_\_: 与函数相同。
* \_\_name\_\_: 与函数相同。
* **\***\_\_module\_\_: 与函数相同。
* im\_func: 使用这个属性可以拿到方法里实际的函数对象的引用。另外如果是2.6以上的版本，还可以使用属性名\_\_func\_\_。
* im\_self: 如果是绑定的(bound)，则指向调用该方法的类（如果是类方法）或实例（如果是实例方法），否则为None。如果是2.6以上的版本，还可以使用属性名\_\_self\_\_。
* im\_class: 实际调用该方法的类，或实际调用该方法的实例的类。注意不是方法的定义所在的类，如果有继承关系的话。

|  |  |
| --- | --- |
| 1  2  3  4 | im = cat.sayHi  print im.im\_func  print im.im\_self # cat  print im.im\_class # Cat |

这里讨论的是一般的实例方法，另外还有两种特殊的方法分别是类方法(classmethod)和静态方法(staticmethod)。类方法还是方法，不过因为需要使用类名调用，所以他始终是绑定的；而静态方法可以看成是在类的命名空间里的函数（需要使用类名调用的函数），它只能使用函数的属性，不能使用方法的属性。

**2.7. 生成器(generator)**

生成器是调用一个生成器函数(generator function)返回的对象，多用于集合对象的迭代。

* \_\_iter\_\_: 仅仅是一个可迭代的标记。
* gi\_code: 生成器对应的code对象。
* gi\_frame: 生成器对应的frame对象。
* gi\_running: 生成器函数是否在执行。生成器函数在yield以后、执行yield的下一行代码前处于frozen状态，此时这个属性的值为0。
* next|close|send|throw: 这是几个可调用的方法，并不包含元数据信息，如何使用可以查看生成器的相关文档。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def gen():      for n in xrange(5):          yield n  g = gen()  print g             # <generator object gen at 0x...>  print g.gi\_code     # <code object gen at 0x...>  print g.gi\_frame    # <frame object at 0x...>  print g.gi\_running  # 0  print g.next()      # 0  print g.next()      # 1  for n in g:      print n,        # 2 3 4 |

接下来讨论的是几个不常用到的内置对象类型。这些类型在正常的编码过程中应该很少接触，除非你正在自己实现一个解释器或开发环境之类。所以这里只列出一部分属性，如果需要一份完整的属性表或想进一步了解，可以查看文末列出的参考文档。

**2.8. 代码块(code)**

代码块可以由类源代码、函数源代码或是一个简单的语句代码编译得到。这里我们只考虑它指代一个函数时的情况；2.5节中我们曾提到可以使用函数的func\_code属性获取到它。code的属性全部是只读的。

* co\_argcount: 普通参数的总数，不包括\*参数和\*\*参数。
* co\_names: 所有的参数名（包括\*参数和\*\*参数）和局部变量名的元组。
* co\_varnames: 所有的局部变量名的元组。
* co\_filename: 源代码所在的文件名。
* co\_flags:  这是一个数值，每一个二进制位都包含了特定信息。较关注的是0b100(0x4)和0b1000(0x8)，如果co\_flags & 0b100 != 0，说明使用了\*args参数；如果co\_flags & 0b1000 != 0，说明使用了\*\*kwargs参数。另外，如果co\_flags & 0b100000(0x20) != 0，则说明这是一个生成器函数(generator function)。

|  |  |
| --- | --- |
| 1  2  3  4  5 | co = cat.sayHi.func\_code  print co.co\_argcount        # 1  print co.co\_names           # ('name',)  print co.co\_varnames        # ('self',)  print co.co\_flags & 0b100   # 0 |

**2.9. 栈帧(frame)**

栈帧表示程序运行时函数调用栈中的某一帧。函数没有属性可以获取它，因为它在函数调用时才会产生，而生成器则是由函数调用返回的，所以有属性指向栈帧。想要获得某个函数相关的栈帧，则必须在调用这个函数且这个函数尚未返回时获取。你可以使用sys模块的\_getframe()函数、或inspect模块的currentframe()函数获取当前栈帧。这里列出来的属性全部是只读的。

* f\_back: 调用栈的前一帧。
* f\_code: 栈帧对应的code对象。
* f\_locals: 用在当前栈帧时与内建函数locals()相同，但你可以先获取其他帧然后使用这个属性获取那个帧的locals()。
* f\_globals: 用在当前栈帧时与内建函数globals()相同，但你可以先获取其他帧……。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def add(x, y=1):      f = inspect.currentframe()      print f.f\_locals    # same as locals()      print f.f\_back      # <frame object at 0x...>      return x+y  add(2) |

**2.10. 追踪(traceback)**

追踪是在出现异常时用于回溯的对象，与栈帧相反。由于异常时才会构建，而异常未捕获时会一直向外层栈帧抛出，所以需要使用try才能见到这个对象。你可以使用sys模块的exc\_info()函数获得它，这个函数返回一个元组，元素分别是异常类型、异常对象、追踪。traceback的属性全部是只读的。

* tb\_next: 追踪的下一个追踪对象。
* tb\_frame: 当前追踪对应的栈帧。
* tb\_lineno: 当前追踪的行号。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def div(x, y):      try:          return x/y      except:          tb = sys.exc\_info()[2]  # return (exc\_type, exc\_value, traceback)          print tb          print tb.tb\_lineno      # "return x/y" 的行号  div(1, 0) |

**3. 使用inspect模块**

inspect模块提供了一系列函数用于帮助使用自省。下面仅列出较常用的一些函数，想获得全部的函数资料可以查看inspect模块的文档。

**3.1. 检查对象类型**

* **is{module|class|function|method|builtin}(obj):**检查对象是否为模块、类、函数、方法、内建函数或方法。
* **isroutine(obj):**用于检查对象是否为函数、方法、内建函数或方法等等可调用类型。用这个方法会比多个is\*()更方便，不过它的实现仍然是用了多个is\*()。

|  |  |
| --- | --- |
| 1  2  3 | im = cat.sayHi  if inspect.isroutine(im):      im() |

* 对于实现了\_\_call\_\_的类实例，这个方法会返回False。如果目的是只要可以直接调用就需要是True的话，不妨使用**isinstance(obj, collections.Callable)**这种形式。我也不知道为什么Callable会在collections模块中，抱歉！我猜大概是因为collections模块中包含了很多其他的ABC(Abstract Base Class)的缘故吧：）

**3.2. 获取对象信息**

* **getmembers(object[, predicate]):**   
  这个方法是dir()的扩展版，它会将dir()找到的名字对应的属性一并返回，形如[(name, value), ...]。另外，predicate是一个方法的引用，如果指定，则应当接受value作为参数并返回一个布尔值，如果为False，相应的属性将不会返回。使用is\*作为第二个参数可以过滤出指定类型的属性。
* **getmodule(object):**   
  还在为第2节中的\_\_module\_\_属性只返回字符串而遗憾吗？这个方法一定可以满足你，它返回object的定义所在的模块对象。
* **get{file|sourcefile}(object):**   
  获取object的定义所在的模块的文件名|源代码文件名（如果没有则返回None）。用于内建的对象（内建模块、类、函数、方法）上时会抛出TypeError异常。
* **get{source|sourcelines}(object):**   
  获取object的定义的源代码，以字符串|字符串列表返回。代码无法访问时会抛出IOError异常。只能用于module/class/function/method/code/frame/traceack对象。
* **getargspec(func):**仅用于方法，获取方法声明的参数，返回元组，分别是(普通参数名的列表, \*参数名, \*\*参数名, 默认值元组)。如果没有值，将是空列表和3个None。如果是2.6以上版本，将返回一个命名元组(Named Tuple)，即除了索引外还可以使用属性名访问元组中的元素。

|  |  |
| --- | --- |
| 1  2  3  4 | def add(x, y=1, \*z):      return x + y + sum(z)  print inspect.getargspec(add)  #ArgSpec(args=['x', 'y'], varargs='z', keywords=None, defaults=(1,)) |

* **getargvalues(frame):**仅用于栈帧，获取栈帧中保存的该次函数调用的参数值，返回元组，分别是(普通参数名的列表, \*参数名, \*\*参数名, 帧的locals())。如果是2.6以上版本，将返回一个命名元组(Named Tuple)，即除了索引外还可以使用属性名访问元组中的元素。

|  |  |
| --- | --- |
| 1  2  3  4  5 | def add(x, y=1, \*z):      print inspect.getargvalues(inspect.currentframe())      return x + y + sum(z)  add(2)  #ArgInfo(args=['x', 'y'], varargs='z', keywords=None, locals={'y': 1, 'x': 2, 'z': ()}) |

* **getcallargs(func[, \*args][, \*\*kwds]):**返回使用args和kwds调用该方法时各参数对应的值的字典。这个方法仅在2.7版本中才有。
* **getmro(cls):**返回一个类型元组，查找类属性时按照这个元组中的顺序。如果是新式类，与cls.\_\_mro\_\_结果一样。但旧式类没有\_\_mro\_\_这个属性，直接使用这个属性会报异常，所以这个方法还是有它的价值的。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | print inspect.getmro(Cat)  #(<class '\_\_main\_\_.Cat'>, <type 'object'>)  print Cat.\_\_mro\_\_  #(<class '\_\_main\_\_.Cat'>, <type 'object'>)  class Dog: pass  print inspect.getmro(Dog)  #(<class \_\_main\_\_.Dog at 0x...>,)  print Dog.\_\_mro\_\_ # AttributeError |

* **currentframe():**返回当前的栈帧对象。

其他的操作frame和traceback的函数请查阅inspect模块的文档，用的比较少，这里就不多介绍了。

<全文完>   
参考资料：   
1. [The standard type hierarchy](http://docs.python.org/reference/datamodel.html#the-standard-type-hierarchy)[官方文档][英文]   
2. [inspect — Inspect live objects](http://docs.python.org/library/inspect.html#module-inspect)[官方文档][英文]

# [Python装饰器与面向切面编程](http://www.cnblogs.com/huxi/archive/2011/03/01/1967600.html)

新年好~  那么，很久没有更新了，其实想想也没多少可以写的，因为Python的文档似乎很全面的说……能做的差不多只有翻译和整理了，英文过关的朋友不妨直接去doc.python.org这里查看相关资料 ：）   
转载请注明原作者和原文地址，多谢！

今天来讨论一下装饰器。装饰器是一个很著名的设计模式，经常被用于有切面需求的场景，较为经典的有插入日志、性能测试、事务处理等。装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量函数中与函数功能本身无关的雷同代码并继续重用。概括的讲，装饰器的作用就是为已经存在的对象添加额外的功能。

## 1. 装饰器入门

### 1.1. 需求是怎么来的？

装饰器的定义很是抽象，我们来看一个小例子。

|  |  |
| --- | --- |
| 1  2  3  4 | def foo():      print 'in foo()'    foo() |

这是一个很无聊的函数没错。但是突然有一个更无聊的人，我们称呼他为B君，说我想看看执行这个函数用了多长时间，好吧，那么我们可以这样做：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | import time  def foo():      start = time.clock()      print 'in foo()'      end = time.clock()      print 'used:', end - start    foo() |

很好，功能看起来无懈可击。可是蛋疼的B君此刻突然不想看这个函数了，他对另一个叫foo2的函数产生了更浓厚的兴趣。

怎么办呢？如果把以上新增加的代码复制到foo2里，这就犯了大忌了~复制什么的难道不是最讨厌了么！而且，如果B君继续看了其他的函数呢？

### 1.2. 以不变应万变，是变也

还记得吗，函数在Python中是一等公民，那么我们可以考虑重新定义一个函数timeit，将foo的引用传递给他，然后在timeit中调用foo并进行计时，这样，我们就达到了不改动foo定义的目的，而且，不论B君看了多少个函数，我们都不用去修改函数定义了！

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import time    def foo():      print 'in foo()'    def timeit(func):      start = time.clock()      func()      end =time.clock()      print 'used:', end - start    timeit(foo) |

看起来逻辑上并没有问题，一切都很美好并且运作正常！……等等，我们似乎修改了调用部分的代码。原本我们是这样调用的：foo()，修改以后变成了：timeit(foo)。这样的话，如果foo在N处都被调用了，你就不得不去修改这N处的代码。或者更极端的，考虑其中某处调用的代码无法修改这个情况，比如：这个函数是你交给别人使用的。

### 1.3. 最大限度地少改动！

既然如此，我们就来想想办法不修改调用的代码；如果不修改调用代码，也就意味着调用foo()需要产生调用timeit(foo)的效果。我们可以想到将timeit赋值给foo，但是timeit似乎带有一个参数……想办法把参数统一吧！如果timeit(foo)不是直接产生调用效果，而是返回一个与foo参数列表一致的函数的话……就很好办了，将timeit(foo)的返回值赋值给foo，然后，调用foo()的代码完全不用修改！

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #-\*- coding: UTF-8 -\*-  import time    def foo():      print 'in foo()'    # 定义一个计时器，传入一个，并返回另一个附加了计时功能的方法  def timeit(func):        # 定义一个内嵌的包装函数，给传入的函数加上计时功能的包装      def wrapper():          start = time.clock()          func()          end =time.clock()          print 'used:', end - start        # 将包装后的函数返回      return wrapper    foo = timeit(foo)  foo() |

这样，一个简易的计时器就做好了！我们只需要在定义foo以后调用foo之前，加上foo = timeit(foo)，就可以达到计时的目的，这也就是装饰器的概念，看起来像是foo被timeit装饰了。在在这个例子中，函数进入和退出时需要计时，这被称为一个横切面(Aspect)，这种编程方式被称为面向切面的编程(Aspect-Oriented Programming)。与传统编程习惯的从上往下执行方式相比较而言，像是在函数执行的流程中横向地插入了一段逻辑。在特定的业务领域里，能减少大量重复代码。面向切面编程还有相当多的术语，这里就不多做介绍，感兴趣的话可以去找找相关的资料。

这个例子仅用于演示，并没有考虑foo带有参数和有返回值的情况，完善它的重任就交给你了 ：）

## 2. Python的额外支持

### 2.1. 语法糖

上面这段代码看起来似乎已经不能再精简了，Python于是提供了一个语法糖来降低字符输入量。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import time    def timeit(func):      def wrapper():          start = time.clock()          func()          end =time.clock()          print 'used:', end - start      return wrapper    @timeit  def foo():      print 'in foo()'    foo() |

重点关注第11行的@timeit，在定义上加上这一行与另外写foo = timeit(foo)完全等价，千万不要以为@有另外的魔力。除了字符输入少了一些，还有一个额外的好处：这样看上去更有装饰器的感觉。

### 2.2. 内置的装饰器

内置的装饰器有三个，分别是staticmethod、classmethod和property，作用分别是把类中定义的实例方法变成静态方法、类方法和类属性。由于模块里可以定义函数，所以静态方法和类方法的用处并不是太多，除非你想要完全的面向对象编程。而属性也不是不可或缺的，Java没有属性也一样活得很滋润。从我个人的Python经验来看，我没有使用过property，使用staticmethod和classmethod的频率也非常低。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Rabbit(object):        def \_\_init\_\_(self, name):          self.\_name = name        @staticmethod      def newRabbit(name):          return Rabbit(name)        @classmethod      def newRabbit2(cls):          return Rabbit('')        @property      def name(self):          return self.\_name |

这里定义的属性是一个只读属性，如果需要可写，则需要再定义一个setter：

|  |  |
| --- | --- |
| 1  2  3 | @name.setter  def name(self, name):      self.\_name = name |

### 2.3. functools模块

functools模块提供了两个装饰器。这个模块是Python 2.5后新增的，一般来说大家用的应该都高于这个版本。但我平时的工作环境是2.4 T-T

**2.3.1. wraps(wrapped[, assigned][, updated]):**这是一个很有用的装饰器。看过前一篇反射的朋友应该知道，函数是有几个特殊属性比如函数名，在被装饰后，上例中的函数名foo会变成包装函数的名字wrapper，如果你希望使用反射，可能会导致意外的结果。这个装饰器可以解决这个问题，它能将装饰过的函数的特殊属性保留。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import time  import functools    def timeit(func):      @functools.wraps(func)      def wrapper():          start = time.clock()          func()          end =time.clock()          print 'used:', end - start      return wrapper    @timeit  def foo():      print 'in foo()'    foo()  print foo.\_\_name\_\_ |

首先注意第5行，如果注释这一行，foo.\_\_name\_\_将是'wrapper'。另外相信你也注意到了，这个装饰器竟然带有一个参数。实际上，他还有另外两个可选的参数，assigned中的属性名将使用赋值的方式替换，而updated中的属性名将使用update的方式合并，你可以通过查看functools的源代码获得它们的默认值。对于这个装饰器，相当于wrapper = functools.wraps(func)(wrapper)。

**2.3.2. total\_ordering(cls):**这个装饰器在特定的场合有一定用处，但是它是在Python 2.7后新增的。它的作用是为实现了至少\_\_lt\_\_、\_\_le\_\_、\_\_gt\_\_、\_\_ge\_\_其中一个的类加上其他的比较方法，这是一个类装饰器。如果觉得不好理解，不妨仔细看看这个装饰器的源代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | 53  def total\_ordering(cls):  54      """Class decorator that fills in missing ordering methods"""  55      convert = {  56          '\_\_lt\_\_': [('\_\_gt\_\_', lambda self, other: other < self),  57                     ('\_\_le\_\_', lambda self, other: not other < self),  58                     ('\_\_ge\_\_', lambda self, other: not self < other)],  59          '\_\_le\_\_': [('\_\_ge\_\_', lambda self, other: other <= self),  60                     ('\_\_lt\_\_', lambda self, other: not other <= self),  61                     ('\_\_gt\_\_', lambda self, other: not self <= other)],  62          '\_\_gt\_\_': [('\_\_lt\_\_', lambda self, other: other > self),  63                     ('\_\_ge\_\_', lambda self, other: not other > self),  64                     ('\_\_le\_\_', lambda self, other: not self > other)],  65          '\_\_ge\_\_': [('\_\_le\_\_', lambda self, other: other >= self),  66                     ('\_\_gt\_\_', lambda self, other: not other >= self),  67                     ('\_\_lt\_\_', lambda self, other: not self >= other)]  68      }  69      roots = set(dir(cls)) & set(convert)  70      if not roots:  71          raise ValueError('must define at least one ordering operation: < > <= >=')  72      root = max(roots)       # prefer \_\_lt\_\_ to \_\_le\_\_ to \_\_gt\_\_ to \_\_ge\_\_  73      for opname, opfunc in convert[root]:  74          if opname not in roots:  75              opfunc.\_\_name\_\_ = opname  76              opfunc.\_\_doc\_\_ = getattr(int, opname).\_\_doc\_\_  77              setattr(cls, opname, opfunc)  78      return cls |

本文到这里就全部结束了，有空的话我会整理一个用于检查参数类型的装饰器的源代码放上来，算是一个应用吧 ：）

# [Python装饰器实例：调用参数合法性验证](http://www.cnblogs.com/huxi/archive/2011/03/31/2001522.html)

之前有说过有空的话我会整理一个用于检查参数类型的装饰器的源代码放上来，那么。。过了好一阵子才完成，因为公司里的代码不方便不出来，所以这个模块完全是现敲的，尽管有一些简单的测试，但我无法保证100%没有问题。放上来的目的更多的是分享一下应用装饰器的思路：）

python是不带静态检查的动态语言，有时候需要在调用函数时保证参数合法。检查参数合法性是一个显著的切面场景，各个函数都可能有这个需求。但另一方面，参数合法性是不是应该由调用方来保证比较好也是一个需要结合实际才能回答的问题，总之双方约定好，不要都不检查或者都检查就可以了。下面这个模块用于在函数上使用装饰器进行参数的合法性验证。

你可以直接执行这个模块进行测试，如果完全没有输出则表示通过。你也可以找到几个以\_test开头的函数，所有的测试用例都包含在这几个函数中。使用方法参见模块文档和测试用例。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320  321  322  323  324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340  341  342  343  344  345  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360  361  362  363  364  365  366 | # -\*- coding: UTF-8 -\*-    '''  @summary: 验证器  该模块提供了一个装饰器用于验证参数是否合法，使用方法为：    from validator import validParam, nullOk, multiType    @validParam(i=int)  def foo(i):      return i+1    编写验证器：    1. 仅验证类型：  @validParam(type, ...)  例如：  检查第一个位置的参数是否为int类型：  @validParam(int)  检查名为x的参数是否为int类型：  @validParam(x=int)    验证多个参数：  @validParam(int, int)  指定参数名验证：  @validParam(int, s=str)    针对\*和\*\*参数编写的验证器将验证这些参数实际包含的每个元素：  @validParam(varargs=int)  def foo(\*varargs): pass    @validParam(kws=int)  def foo7(s, \*\*kws): pass    2. 带有条件的验证：  @validParam((type, condition), ...)  其中，condition是一个表达式字符串，使用x引用待验证的对象；  根据bool(表达式的值)判断是否通过验证，若计算表达式时抛出异常，视为失败。  例如：  验证一个10到20之间的整数：  @validParam(i=(int, '10<x<20'))  验证一个长度小于20的字符串：  @validParam(s=(str, 'len(x)<20'))  验证一个年龄小于20的学生：  @validParam(stu=(Student, 'x.age<20'))    另外，如果类型是字符串，condition还可以使用斜杠开头和结尾表示正则表达式匹配。  验证一个由数字组成的字符串：  @validParam(s=(str, '/^\d\*$/'))    3. 以上验证方式默认为当值是None时验证失败。如果None是合法的参数，可以使用nullOk()。  nullOk()接受一个验证条件作为参数。  例如：  @validParam(i=nullOk(int))  @validParam(i=nullOk((int, '10<x<20')))  也可以简写为：  @validParam(i=nullOk(int, '10<x<20'))    4. 如果参数有多个合法的类型，可以使用multiType()。  multiType()可接受多个参数，每个参数都是一个验证条件。  例如：  @validParam(s=multiType(int, str))  @validParam(s=multiType((int, 'x>20'), nullOk(str, '/^\d+$/')))    5. 如果有更复杂的验证需求，还可以编写一个函数作为验证函数传入。  这个函数接收待验证的对象作为参数，根据bool(返回值)判断是否通过验证，抛出异常视为失败。  例如：  def validFunction(x):      return isinstance(x, int) and x>0  @validParam(i=validFunction)  def foo(i): pass    这个验证函数等价于：  @validParam(i=(int, 'x>0'))  def foo(i): pass      @author: HUXI  @since: 2011-3-22  @change:  '''    import inspect  import re    class ValidateException(Exception): pass      def validParam(\*varargs, \*\*keywords):      '''验证参数的装饰器。'''        varargs = map(\_toStardardCondition, varargs)      keywords = dict((k, \_toStardardCondition(keywords[k]))                      for k in keywords)        def generator(func):          args, varargname, kwname = inspect.getargspec(func)[:3]          dctValidator = \_getcallargs(args, varargname, kwname,                                      varargs, keywords)            def wrapper(\*callvarargs, \*\*callkeywords):              dctCallArgs = \_getcallargs(args, varargname, kwname,                                         callvarargs, callkeywords)                k, item = None, None              try:                  for k in dctValidator:                      if k == varargname:                          for item in dctCallArgs[k]:                              assert dctValidator[k](item)                      elif k == kwname:                          for item in dctCallArgs[k].values():                              assert dctValidator[k](item)                      else:                          item = dctCallArgs[k]                          assert dctValidator[k](item)              except:                  raise ValidateException,\                         ('%s() parameter validation fails, param: %s, value: %s(%s)'                         % (func.func\_name, k, item, item.\_\_class\_\_.\_\_name\_\_))                return func(\*callvarargs, \*\*callkeywords)            wrapper = \_wrapps(wrapper, func)          return wrapper        return generator      def \_toStardardCondition(condition):      '''将各种格式的检查条件转换为检查函数'''        if inspect.isclass(condition):          return lambda x: isinstance(x, condition)        if isinstance(condition, (tuple, list)):          cls, condition = condition[:2]          if condition is None:              return \_toStardardCondition(cls)            if cls in (str, unicode) and condition[0] == condition[-1] == '/':              return lambda x: (isinstance(x, cls)                                and re.match(condition[1:-1], x) is not None)            return lambda x: isinstance(x, cls) and eval(condition)        return condition      def nullOk(cls, condition=None):      '''这个函数指定的检查条件可以接受None值'''        return lambda x: x is None or \_toStardardCondition((cls, condition))(x)      def multiType(\*conditions):      '''这个函数指定的检查条件只需要有一个通过'''        lstValidator = map(\_toStardardCondition, conditions)      def validate(x):          for v in lstValidator:              if v(x):                  return True      return validate      def \_getcallargs(args, varargname, kwname, varargs, keywords):      '''获取调用时的各参数名-值的字典'''        dctArgs = {}      varargs = tuple(varargs)      keywords = dict(keywords)        argcount = len(args)      varcount = len(varargs)      callvarargs = None        if argcount <= varcount:          for n, argname in enumerate(args):              dctArgs[argname] = varargs[n]            callvarargs = varargs[-(varcount-argcount):]        else:          for n, var in enumerate(varargs):              dctArgs[args[n]] = var            for argname in args[-(argcount-varcount):]:              if argname in keywords:                  dctArgs[argname] = keywords.pop(argname)            callvarargs = ()        if varargname is not None:          dctArgs[varargname] = callvarargs        if kwname is not None:          dctArgs[kwname] = keywords        dctArgs.update(keywords)      return dctArgs      def \_wrapps(wrapper, wrapped):      '''复制元数据'''        for attr in ('\_\_module\_\_', '\_\_name\_\_', '\_\_doc\_\_'):          setattr(wrapper, attr, getattr(wrapped, attr))      for attr in ('\_\_dict\_\_',):          getattr(wrapper, attr).update(getattr(wrapped, attr, {}))        return wrapper      #===============================================================================  # 测试  #===============================================================================      def \_unittest(func, \*cases):      for case in cases:          \_functest(func, \*case)      def \_functest(func, isCkPass, \*args, \*\*kws):      if isCkPass:          func(\*args, \*\*kws)      else:          try:              func(\*args, \*\*kws)              assert False          except ValidateException:              pass    def \_test1\_simple():      #检查第一个位置的参数是否为int类型：      @validParam(int)      def foo1(i): pass      \_unittest(foo1,                (True, 1),                (False, 's'),                (False, None))        #检查名为x的参数是否为int类型：      @validParam(x=int)      def foo2(s, x): pass      \_unittest(foo2,                (True, 1, 2),                (False, 's', 's'))        #验证多个参数：      @validParam(int, int)      def foo3(s, x): pass      \_unittest(foo3,                (True, 1, 2),                (False, 's', 2))        #指定参数名验证：      @validParam(int, s=str)      def foo4(i, s): pass      \_unittest(foo4,                (True, 1, 'a'),                (False, 's', 1))        #针对\*和\*\*参数编写的验证器将验证这些参数包含的每个元素：      @validParam(varargs=int)      def foo5(\*varargs): pass      \_unittest(foo5,                (True, 1, 2, 3, 4, 5),                (False, 'a', 1))        @validParam(kws=int)      def foo6(\*\*kws): pass      \_functest(foo6, True, a=1, b=2)      \_functest(foo6, False, a='a', b=2)        @validParam(kws=int)      def foo7(s, \*\*kws): pass      \_functest(foo7, True, s='a', a=1, b=2)      def \_test2\_condition():      #验证一个10到20之间的整数：      @validParam(i=(int, '10<x<20'))      def foo1(x, i): pass      \_unittest(foo1,                (True, 1, 11),                (False, 1, 'a'),                (False, 1, 1))        #验证一个长度小于20的字符串：      @validParam(s=(str, 'len(x)<20'))      def foo2(a, s): pass      \_unittest(foo2,                (True, 1, 'a'),                (False, 1, 1),                (False, 1, 'a'\*20))        #验证一个年龄小于20的学生：      class Student(object):          def \_\_init\_\_(self, age): self.age=age        @validParam(stu=(Student, 'x.age<20'))      def foo3(stu): pass      \_unittest(foo3,                (True, Student(18)),                (False, 1),                (False, Student(20)))        #验证一个由数字组成的字符串：      @validParam(s=(str, r'/^\d\*$/'))      def foo4(s): pass      \_unittest(foo4,                (True, '1234'),                (False, 1),                (False, 'a1234'))      def \_test3\_nullok():      @validParam(i=nullOk(int))      def foo1(i): pass      \_unittest(foo1,                (True, 1),                (False, 'a'),                (True, None))        @validParam(i=nullOk(int, '10<x<20'))      def foo2(i): pass      \_unittest(foo2,                (True, 11),                (False, 'a'),                (True, None),                (False, 1))      def \_test4\_multitype():      @validParam(s=multiType(int, str))      def foo1(s): pass      \_unittest(foo1,                (True, 1),                (True, 'a'),                (False, None),                (False, 1.1))        @validParam(s=multiType((int, 'x>20'), nullOk(str, '/^\d+$/')))      def foo2(s): pass      \_unittest(foo2,                (False, 1),                (False, 'a'),                (True, None),                (False, 1.1),                (True, 21),                (True, '21'))    def \_main():      d = globals()      from types import FunctionType      print      for f in d:          if f.startswith('\_test'):              f = d[f]              if isinstance(f, FunctionType):                  f()    if \_\_name\_\_ == '\_\_main\_\_':      \_main() |

[**Python函数式编程指南：目录和参考**](http://www.cnblogs.com/huxi/archive/2011/07/15/2107536.html)

目录：

* [概述](http://www.cnblogs.com/huxi/archive/2011/06/18/2084316.html)   
  这一篇简要地描述了函数式编程和一些相关的概念。
* [函数](http://www.cnblogs.com/huxi/archive/2011/06/24/2089358.html)   
  这一篇展示了关于函数可能不常用到的部分特征并尝试引导诸位使用函数式的思维解决问题，并介绍了诸多有用的内建函数。
* [迭代器](http://www.cnblogs.com/huxi/archive/2011/07/01/2095931.html)   
  这一篇介绍了迭代器以及Python对迭代器的语法级的支持，还包括了内置模块itertools的介绍。
* [生成器](http://www.cnblogs.com/huxi/archive/2011/07/14/2106863.html)   
  这一篇介绍了自定义的迭代器——生成器，并展示了生成器的用途和使用过程的一些疑难解答。

更多的参考：

* 概念   
  [函数式编程](http://zh.wikipedia.org/wiki/%E5%87%BD%E6%95%B8%E5%BC%8F%E7%B7%A8%E7%A8%8B) | [Functional Programming](http://en.wikipedia.org/wiki/Functional_Programming)   
  [闭包](http://zh.wikipedia.org/wiki/%E9%97%AD%E5%8C%85_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6))  | [Closure](http://en.wikipedia.org/wiki/Closure_(computer_science))   
  [迭代器](http://zh.wikipedia.org/wiki/%E8%BF%AD%E4%BB%A3%E5%99%A8)  | [Iterator](http://en.wikipedia.org/wiki/Iterator)   
  [协程](http://zh.wikipedia.org/wiki/%E5%8D%8F%E7%A8%8B)  | [Coroutine](http://en.wikipedia.org/wiki/Coroutine)
* 文档   
  [Functional Programming HOWTO](http://docs.python.org/howto/functional.html)
* 库文档   
  [Built-in Functions](http://docs.python.org/library/functions.html)   
  [functools](http://docs.python.org/library/functools.html)   
  [itertools](http://docs.python.org/library/itertools.html)
* 第三方库   
  [functional](http://pypi.python.org/pypi/functional)   
  [pipe](http://pypi.python.org/pypi/pipe)
* 笔者的相关文章   
  [Python自省（反射）指南](http://www.cnblogs.com/huxi/archive/2011/01/02/1924317.html)   
  [Python装饰器与面向切面编程](http://www.cnblogs.com/huxi/archive/2011/03/01/1967600.html)   
  [Python装饰器实例：调用参数合法性验证](http://www.cnblogs.com/huxi/archive/2011/03/31/2001522.html)

# [Python函数式编程指南（一）：概述](http://www.cnblogs.com/huxi/archive/2011/06/18/2084316.html)

这大概算是Python最难啃的一块骨头吧。在我Python生涯的这一年里，我遇到了一些Pythoner，他们毫无例外地完全不会使用函数式编程（有些人喜欢称为Pythonic），比如，从来不会传递函数，不知道lambda是什么意思，知道列表展开但从来不知道用在哪里，对Python不提供经典for循环感到无所适从，言谈之中表现出对函数式风格的一种抗拒甚至厌恶。

我尝试剖析这个问题，最终总结了这么两个原因：1、不想改变，认为现有的知识可以完成任务；2、对小众语言的歧视，Python目前在国内市场份额仍然很小很小，熟悉Python风格用处不大。

然而我认为，学习使用一种截然不同的风格可以颠覆整个编程的思想。我会慢慢总结一个系列共4篇文字，篇幅都不大，轻松就能看完，希望对喜欢Python的人们有所帮助，因为我个人确实从中受益匪浅。

还是那句老话，尊重作者的劳动，转载请注明原作者和原地址：）

#### 1. 函数式编程概述

##### 1.1. 什么是函数式编程？

函数式编程使用一系列的函数解决问题。函数仅接受输入并产生输出，不包含任何能影响产生输出的内部状态。任何情况下，使用相同的参数调用函数始终能产生同样的结果。

在一个函数式的程序中，输入的数据“流过”一系列的函数，每一个函数根据它的输入产生输出。函数式风格避免编写有“边界效应”(side effects)的函数：修改内部状态，或者是其他无法反应在输出上的变化。完全没有边界效应的函数被称为“纯函数式的”(purely functional)。避免边界效应意味着不使用在程序运行时可变的数据结构，输出只依赖于输入。

可以认为函数式编程刚好站在了面向对象编程的对立面。对象通常包含内部状态（字段），和许多能修改这些状态的函数，程序则由不断修改状态构成；函数式编程则极力避免状态改动，并通过在函数间传递数据流进行工作。但这并不是说无法同时使用函数式编程和面向对象编程，事实上，复杂的系统一般会采用面向对象技术建模，但混合使用函数式风格还能让你额外享受函数式风格的优点。

##### 1.2. 为什么使用函数式编程？

函数式的风格通常被认为有如下优点：

* 逻辑可证   
  这是一个学术上的优点：没有边界效应使得更容易从逻辑上证明程序是正确的（而不是通过测试）。
* 模块化   
  函数式编程推崇简单原则，一个函数只做一件事情，将大的功能拆分成尽可能小的模块。小的函数更易于阅读和检查错误。
* 组件化   
  小的函数更容易加以组合形成新的功能。
* 易于调试   
  细化的、定义清晰的函数使得调试更加简单。当程序不正常运行时，每一个函数都是检查数据是否正确的接口，能更快速地排除没有问题的代码，定位到出现问题的地方。
* 易于测试   
  不依赖于系统状态的函数无须在测试前构造测试桩，使得编写单元测试更加容易。
* 更高的生产率   
  函数式编程产生的代码比其他技术更少（往往是其他技术的一半左右），并且更容易阅读和维护。

##### 1.3. 如何辨认函数式风格？

支持函数式编程的语言通常具有如下特征，大量使用这些特征的代码即可被认为是函数式的：

* 函数是一等公民   
  函数能作为参数传递，或者是作为返回值返回。这个特性使得模板方法模式非常易于编写，这也促使了这个模式被更频繁地使用。   
  以一个简单的集合排序为例，假设lst是一个数集，并拥有一个排序方法sort需要将如何确定顺序作为参数。   
  如果函数不能作为参数，那么lst的sort方法只能接受普通对象作为参数。这样一来我们需要首先定义一个接口，然后定义一个实现该接口的类，最后将该类的一个实例传给sort方法，由sort调用这个实例的compare方法，就像这样：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #伪代码  interface Comparator {      compare(o1, o2)  }  lst = list(range(5))  lst.sort(Comparator() {      compare(o1, o2) {          return o2 - o1 //逆序  }) |

* 可见，我们定义了一个新的接口、新的类型（这里是一个匿名类），并new了一个新的对象只为了调用一个方法。如果这个方法可以直接作为参数传递会怎样呢？看起来应该像这样：

|  |  |
| --- | --- |
| 1  2  3  4 | def compare(o1, o2):      return o2 - o1 #逆序  lst = list(range(5))  lst.sort(compare) |

* 请注意，前一段代码已经使用了匿名类技巧从而省下了不少代码，但仍然不如直接传递函数简单、自然。
* 匿名函数(lambda)   
  lambda提供了快速编写简单函数的能力。对于偶尔为之的行为，lambda让你不再需要在编码时跳转到其他位置去编写函数。   
  lambda表达式定义一个匿名的函数，如果这个函数仅在编码的位置使用到，你可以现场定义、直接使用：

|  |  |
| --- | --- |
| 1 | lst.sort(lambda o1, o2: o1.compareTo(o2)) |

* 相信从这个小小的例子你也能感受到强大的生产效率：）
* 封装控制结构的内置模板函数   
  为了避开边界效应，函数式风格尽量避免使用变量，而仅仅为了控制流程而定义的循环变量和流程中产生的临时变量无疑是最需要避免的。   
  假如我们需要对刚才的数集进行过滤得到所有的正数，使用指令式风格的代码应该像是这样：

|  |  |
| --- | --- |
| 1  2  3  4 | lst2 = list()  for i in range(len(lst)): #模拟经典for循环      if lst[i] > 0:          lst2.append(lst[i]) |

* 这段代码把从创建新列表、循环、取出元素、判断、添加至新列表的整个流程完整的展示了出来，俨然把解释器当成了需要手把手指导的傻瓜。然而，“过滤”这个动作是很常见的，为什么解释器不能掌握过滤的流程，而我们只需要告诉它过滤规则呢？   
  在Python里，过滤由一个名为filter的内置函数实现。有了这个函数，解释器就学会了如何“过滤”，而我们只需要把规则告诉它：

|  |  |
| --- | --- |
| 1 | lst2 = filter(lambda n: n > 0, lst) |

* 这个函数带来的好处不仅仅是少写了几行代码这么简单。   
  封装控制结构后，代码中就只需要描述功能而不是做法，这样的代码更清晰，更可读。因为避开了控制结构的干扰，第二段代码显然能让你更容易了解它的意图。   
  另外，因为避开了索引，使得代码中不太可能触发下标越界这种异常，除非你手动制造一个。   
  函数式编程语言通常封装了数个类似“过滤”这样的常见动作作为模板函数。唯一的缺点是这些函数需要少量的学习成本，但这绝对不能掩盖使用它们带来的好处。
* 闭包(closure)   
  闭包是绑定了外部作用域的变量（但不是全局变量）的函数。大部分情况下外部作用域指的是外部函数。   
  闭包包含了自身函数体和所需外部函数中的“变量名的引用”。引用变量名意味着绑定的是变量名，而不是变量实际指向的对象；如果给变量重新赋值，闭包中能访问到的将是新的值。   
  闭包使函数更加灵活和强大。即使程序运行至离开外部函数，如果闭包仍然可见，则被绑定的变量仍然有效；每次运行至外部函数，都会重新创建闭包，绑定的变量是不同的，不需要担心在旧的闭包中绑定的变量会被新的值覆盖。   
  回到刚才过滤数集的例子。假设过滤条件中的 0 这个边界值不再是固定的，而是由用户控制。如果没有闭包，那么代码必须修改为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class greater\_than\_helper:      def \_\_init\_\_(self, minval):          self.minval = minval      def is\_greater\_than(self, val):          return val > self.minval    def my\_filter(lst, minval):      helper = greater\_than\_helper(minval)      return filter(helper.is\_greater\_than, lst) |

* 请注意我们现在已经为过滤功能编写了一个函数my\_filter。如你所见，我们需要在别的地方（此例中是类greater\_than\_helper）持有另一个操作数minval。   
  如果支持闭包，因为闭包可以直接使用外部作用域的变量，我们就不再需要greater\_than\_helper了：

|  |  |
| --- | --- |
| 1  2 | def my\_filter(lst, minval):      return filter(lambda n: n > minval, lst) |

* 可见，闭包在不影响可读性的同时也省下了不少代码量。   
  函数式编程语言都提供了对闭包的不同程度的支持。在Python 2.x中，闭包无法修改绑定变量的值，所有修改绑定变量的行为都被看成新建了一个同名的局部变量并将绑定变量隐藏。Python 3.x中新加入了一个关键字 nonlocal 以支持修改绑定变量。但不管支持程度如何，你始终可以访问（读取）绑定变量。
* 内置的不可变数据结构   
  为了避开边界效应，不可变的数据结构是函数式编程中不可或缺的部分。不可变的数据结构保证数据的一致性，极大地降低了排查问题的难度。   
  例如，Python中的元组(tuple)就是不可变的，所有对元组的操作都不能改变元组的内容，所有试图修改元组内容的操作都会产生一个异常。   
  函数式编程语言一般会提供数据结构的两种版本（可变和不可变），并推荐使用不可变的版本。
* 递归   
  递归是另一种取代循环的方法。递归其实是函数式编程很常见的形式，经常可以在一些算法中见到。但之所以放到最后，是因为实际上我们一般很少用到递归。如果一个递归无法被编译器或解释器优化，很容易就会产生栈溢出；另一方面复杂的递归往往让人感觉迷惑，不如循环清晰，所以众多最佳实践均指出使用循环而非递归。   
  这一系列短文中都不会关注递归的使用。

<第一节完>

# [Python函数式编程指南（二）：函数](http://www.cnblogs.com/huxi/archive/2011/06/24/2089358.html)

这是此系列的第二篇，试图说明在Python中如何更好地使用函数并引导诸位使用函数式的思维进行思考。掌握并应用这些内容，就已经是至少形似的函数式风格的代码了，至于思维么，这个真靠自己。

作者水平有限，如有错漏之处还请指出；转载请注明原作者和原地址：）

#### 2. 从函数开始

##### 2.1. 定义一个函数

如下定义了一个求和函数：

|  |  |
| --- | --- |
| 1  2 | def add(x, y):      return x + y |

关于参数和返回值的语法细节可以参考其他文档，这里就略过了。

使用lambda可以定义简单的单行匿名函数。lambda的语法是：

|  |  |
| --- | --- |
| 1 | lambda args: expression |

参数(args)的语法与普通函数一样，同时表达式(expression)的值就是匿名函数调用的返回值；而lambda表达式返回这个匿名函数。如果我们给匿名函数取个名字，就像这样：

|  |  |
| --- | --- |
| 1 | lambda\_add = lambda x, y: x + y |

这与使用def定义的求和函数完全一样，可以使用lambda\_add作为函数名进行调用。然而，提供lambda的目的是为了编写偶尔为之的、简单的、可预见不会被修改的匿名函数。这种风格虽然看起来很酷，但并不是一个好主意，特别是当某一天需要对它进行扩充，再也无法用一个表达式写完时。如果一开始就需要给函数命名，应该始终使用def关键字。

##### 2.2. 使用函数赋值

事实上你已经见过了，上一节中我们将lambda表达式赋值给了add。同样，使用def定义的函数也可以赋值，相当于为函数取了一个别名，并且可以使用这个别名调用函数：

|  |  |
| --- | --- |
| 1  2 | add\_a\_number\_to\_another\_one\_by\_using\_plus\_operator = add  print add\_a\_number\_to\_another\_one\_by\_using\_plus\_operator(1, 2) |

既然函数可以被变量引用，那么将函数作为参数和返回值就是很寻常的做法了。

##### 2.3. 闭包

闭包是一类特殊的函数。如果一个函数定义在另一个函数的作用域中，并且函数中引用了外部函数的局部变量，那么这个函数就是一个闭包。下面的代码定义了一个闭包：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def f():      n = 1      def inner():          print n      inner()      n = 'x'      inner() |

函数inner定义在f的作用域中，并且在inner中使用了f中的局部变量n，这就构成了一个闭包。闭包绑定了外部的变量，所以调用函数f的结果是打印1和'x'。这类似于普通的模块函数和模块中定义的全局变量的关系：修改外部变量能影响内部作用域中的值，而在内部作用域中定义同名变量则将遮蔽（隐藏）外部变量。

如果需要在函数中修改全局变量，可以使用关键字global修饰变量名。Python 2.x中没有关键字为在闭包中修改外部变量提供支持，在3.x中，关键字nonlocal可以做到这一点：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #Python 3.x supports `nonlocal'  def f():      n = 1      def inner():          nonlocal n          n = 'x'      print(n)      inner()      print(n) |

调用这个函数的结果是打印1和'x'，如果你有一个Python 3.x的解释器，可以试着运行一下。

由于使用了函数体外定义的变量，看起来闭包似乎违反了函数式风格的规则即不依赖外部状态。但是由于闭包绑定的是外部函数的局部变量，而一旦离开外部函数作用域，这些局部变量将无法再从外部访问；另外闭包还有一个重要的特性，每次执行至闭包定义处时都会构造一个新的闭包，这个特性使得旧的闭包绑定的变量不会随第二次调用外部函数而更改。所以闭包实际上不会被外部状态影响，完全符合函数式风格的要求。（这里有一个特例，Python 3.x中，如果同一个作用域中定义了两个闭包，由于可以修改外部变量，他们可以相互影响。）

虽然闭包只有在作为参数和返回值时才能发挥它的真正威力，但闭包的支持仍然大大提升了生产率。

##### 2.4. 作为参数

如果你对OOP的模板方法模式很熟悉，相信你能很快速地学会将函数当作参数传递。两者大体是一致的，只是在这里，我们传递的是函数本身而不再是实现了某个接口的对象。   
我们先来给前面定义的求和函数add热热身：

|  |  |
| --- | --- |
| 1 | print add('三角形的树', '北极') |

与加法运算符不同，你一定很惊讶于答案是'三角函数'。这是一个内置的彩蛋...bazinga!

言归正传。我们的客户有一个从0到4的列表：

|  |  |
| --- | --- |
| 1 | lst = range(5) #[0, 1, 2, 3, 4] |

虽然我们在上一小节里给了他一个加法器，但现在他仍然在为如何计算这个列表所有元素的和而苦恼。当然，对我们而言这个任务轻松极了：

|  |  |
| --- | --- |
| 1  2  3 | amount = 0  for num in lst:      amount = add(amount, num) |

这是一段典型的指令式风格的代码，一点问题都没有，肯定可以得到正确的结果。现在，让我们试着用函数式的风格重构一下。

首先可以预见的是求和这个动作是非常常见的，如果我们把这个动作抽象成一个单独的函数，以后需要对另一个列表求和时，就不必再写一遍这个套路了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def sum\_(lst):      amount = 0      for num in lst:          amount = add(amount, num)      return amount    print sum\_(lst) |

还能继续。sum\_函数定义了这样一种流程：   
1. 使用初始值与列表的第一个元素相加；   
2. 使用上一次相加的结果与列表的下一个元素相加；   
3. 重复第二步，直到列表中没有更多元素；   
4. 将最后一次相加的结果返回。

如果现在需要求乘积，我们可以写出类似的流程——只需要把相加换成相乘就可以了：

|  |  |
| --- | --- |
| 1  2  3  4  5 | def multiply(lst):      product = 1      for num in lst:          product = product \* num      return product |

除了初始值换成了1以及函数add换成了乘法运算符，其他的代码全部都是冗余的。我们为什么不把这个流程抽象出来，而将加法、乘法或者其他的函数作为参数传入呢？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def reduce\_(function, lst, initial):      result = initial      for num in lst:          result = function(result, num)      return result    print reduce\_(add, lst, 0) |

现在，想要算出乘积，可以这样做：

|  |  |
| --- | --- |
| 1 | print reduce\_(lambda x, y: x \* y, lst, 1) |

那么，如果想要利用reduce\_找出列表中的最大值，应该怎么做呢？请自行思考：）

虽然有模板方法这样的设计模式，但那样的复杂度往往使人们更情愿到处编写循环。将函数作为参数完全避开了模板方法的复杂度。

Python有一个内建函数reduce，完整实现并扩展了reduce\_的功能。本文稍后的部分包含了有用的内建函数的介绍。请注意我们的目的是没有循环，使用函数替代循环是函数式风格区别于指令式风格的最显而易见的特征。

\*像Python这样构建于类C语言之上的函数式语言，由于语言本身提供了编写循环代码的能力，内置函数虽然提供函数式编程的接口，但一般在内部还是使用循环实现的。同样的，如果发现内建函数无法满足你的循环需求，不妨也封装它，并提供一个接口。

##### 2.5. 作为返回值

将函数返回通常需要与闭包一起使用（即返回一个闭包）才能发挥威力。我们先看一个函数的定义：

|  |  |
| --- | --- |
| 1  2  3  4  5 | def map\_(function, lst):      result = []      for item in lst:          result.append(function(item))      return result |

函数map\_封装了最常见的一种迭代：对列表中的每个元素调用一个函数。map\_需要一个函数参数，并将每次调用的结果保存在一个列表中返回。这是指令式的做法，当你知道了列表解析(list comprehension)后，会有更好的实现。

这里我们先略过map\_的蹩脚实现而只关注它的功能。对于上一节中的lst，你可能发现最后求乘积结果始终是0，因为lst中包含了0。为了让结果看起来足够大，我们来使用map\_为lst中的每个元素加1：

|  |  |
| --- | --- |
| 1  2 | lst = map\_(lambda x: add(1, x), lst)  print reduce\_(lambda x, y: x \* y, lst, 1) |

答案是120，这还远远不够大。再来：

|  |  |
| --- | --- |
| 1  2 | lst = map\_(lambda x: add(10, x), lst)  print reduce\_(lambda x, y: x \* y, lst, 1) |

囧，事实上我真的没有想到答案会是360360，我发誓没有收周鸿祎任何好处。

现在回头看看我们写的两个lambda表达式：相似度超过90%，绝对可以使用抄袭来形容。而问题不在于抄袭，在于多写了很多字符有木有？如果有一个函数，根据你指定的左操作数，能生成一个加法函数，用起来就像这样：

|  |  |
| --- | --- |
| 1 | lst = map\_(add\_to(10), lst) #add\_to(10)返回一个函数，这个函数接受一个参数并加上10后返回 |

写起来应该会舒服不少。下面是函数add\_to的实现：

|  |  |
| --- | --- |
| 1  2 | def add\_to(n):      return lambda x: add(n, x) |

通过为已经存在的某个函数指定数个参数，生成一个新的函数，这个函数只需要传入剩余未指定的参数就能实现原函数的全部功能，这被称为偏函数。Python内置的functools模块提供了一个函数partial，可以为任意函数生成偏函数：

|  |  |
| --- | --- |
| 1 | functools.partial(func[, \*args][, \*\*keywords]) |

你需要指定要生成偏函数的函数、并且指定数个参数或者命名参数，然后partial将返回这个偏函数；不过严格的说partial返回的不是函数，而是一个像函数一样可直接调用的对象，当然，这不会影响它的功能。

另外一个特殊的例子是装饰器。装饰器用于增强甚至干脆改变原函数的功能，我曾写过一篇文档介绍装饰器，地址在这里：<http://www.cnblogs.com/huxi/archive/2011/03/01/1967600.html>。

\*题外话，单就例子中的这个功能而言，在一些其他的函数式语言中（例如Scala）可以使用名为柯里化(Currying)的技术实现得更优雅。柯里化是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。如下的伪代码所示：

|  |  |
| --- | --- |
| 1  2  3  4  5 | #不是真实的代码  def add(x)(y): #柯里化      return x + y    lst = map\_(add(10), lst) |

通过将add函数柯里化，使得add接受第一个参数x，并返回一个接受第二个参数y的函数，调用该函数与前文中的add\_to完全相同（返回x + y），且不再需要定义add\_to。看上去是不是更加清爽呢？遗憾的是Python并不支持柯里化。

##### 2.6. 部分内建函数介绍

* reduce(function, iterable[, initializer])   
  这个函数的主要功能与我们定义的reduce\_相同。需要补充两点：   
  它的第二个参数可以是任何可迭代的对象（实现了\_\_iter\_\_()方法的对象）；   
  如果不指定第三个参数，则第一次调用function将使用iterable的前两个元素作为参数。   
  由reduce和一些常见的function组合成了下面列出来的内置函数：

|  |  |
| --- | --- |
| 1  2  3  4  5 | all(iterable) == reduce(lambda x, y: bool(x and y), iterable)  any(iterable) == reduce(lambda x, y: bool(x or y), iterable)  max(iterable[, args...][, key]) == reduce(lambda x, y: x if key(x) > key(y) else y, iterable\_and\_args)  min(iterable[, args...][, key]) == reduce(lambda x, y: x if key(x) < key(y) else y, iterable\_and\_args)  sum(iterable[, start]) == reduce(lambda x, y: x + y, iterable, start) |

* map(function, iterable, ...)   
  这个函数的主要功能与我们定义的map\_相同。需要补充一点：   
  map还可以接受多个iterable作为参数，在第n次调用function时，将使用iterable1[n], iterable2[n], ...作为参数。
* filter(function, iterable)   
  这个函数的功能是过滤出iterable中所有以元素自身作为参数调用function时返回True或bool(返回值)为True的元素并以列表返回，与系列第一篇中的my\_filter函数相同。
* zip(iterable1, iterable2, ...)   
  这个函数返回一个列表，每个元素都是一个元组，包含(iterable1[n], iterable2[n], ...)。   
  例如：zip([1, 2], [3, 4]) --> [(1, 3), (2, 4)]   
  如果参数的长度不一致，将在最短的序列结束时结束；如果不提供参数，将返回空列表。

除此之外，你还可以使用本文2.5节中提到的functools.partial()为这些内置函数创建常用的偏函数。

另外，pypi上有一个名为functional的模块，除了这些内建函数外，还额外提供了更多的有意思的函数。但由于使用的场合并不多，并且需要额外安装，在本文中就不介绍了。但我仍然推荐大家下载这个模块的纯Python实现的源代码看看，开阔思维嘛。里面的函数都非常短，源文件总共只有300行不到，地址在这里：<http://pypi.python.org/pypi/functional>

此篇结束：）

# [Python函数式编程指南（三）：迭代器](http://www.cnblogs.com/huxi/archive/2011/07/01/2095931.html)

这一篇我们将讨论迭代器。迭代器并不是函数式编程特有的东西，但它仍然是函数式编程的一个重要的组成部分，或者说是一个重要的工具。

转载请注明原作者和原文地址：）

#### 3. 迭代器

##### 3.1. 迭代器(Iterator)概述

迭代器是访问集合内元素的一种方式。迭代器对象从集合的第一个元素开始访问，直到所有的元素都被访问一遍后结束。

迭代器不能回退，只能往前进行迭代。这并不是什么很大的缺点，因为人们几乎不需要在迭代途中进行回退操作。

迭代器也不是线程安全的，在多线程环境中对可变集合使用迭代器是一个危险的操作。但如果小心谨慎，或者干脆贯彻函数式思想坚持使用不可变的集合，那这也不是什么大问题。

对于原生支持随机访问的数据结构（如tuple、list），迭代器和经典for循环的索引访问相比并无优势，反而丢失了索引值（可以使用内建函数enumerate()找回这个索引值，这是后话）。但对于无法随机访问的数据结构（比如set）而言，迭代器是唯一的访问元素的方式。

迭代器的另一个优点就是它不要求你事先准备好整个迭代过程中所有的元素。迭代器仅仅在迭代至某个元素时才计算该元素，而在这之前或之后，元素可以不存在或者被销毁。这个特点使得它特别适合用于遍历一些巨大的或是无限的集合，比如几个G的文件，或是斐波那契数列等等。这个特点被称为延迟计算或惰性求值(Lazy evaluation)。

迭代器更大的功劳是提供了一个统一的访问集合的接口。只要是实现了\_\_iter\_\_()方法的对象，就可以使用迭代器进行访问。

##### 3.2. 使用迭代器

使用内建的工厂函数iter(iterable)可以获取迭代器对象：

|  |  |
| --- | --- |
| 1  2  3  4 | >>> lst = range(2)  >>> it = iter(lst)  >>> it  <listiterator object at 0x00BB62F0> |

使用迭代器的next()方法可以访问下一个元素：

|  |  |
| --- | --- |
| 1  2 | >>> it.next()  0 |

如果是Python 2.6+，还有内建函数next(iterator)可以完成这一功能：

|  |  |
| --- | --- |
| 1  2 | >>> next(it)  1 |

如何判断迭代器还有更多的元素可以访问呢？Python里的迭代器并没有提供类似has\_next()这样的方法。   
那么在这个例子中，我们已经访问到了最后一个元素1，再使用next()方法会怎样呢？

|  |  |
| --- | --- |
| 1  2  3  4 | >>> it.next()  Traceback (most recent call last):    File "<stdin>", line 1, in <module>  StopIteration |

Python遇到这样的情况时将会抛出StopIteration异常。事实上，Python正是根据是否检查到这个异常来决定是否停止迭代的。   
这种做法与迭代前手动检查是否越界相比各有优点。但Python的做法总有一些利用异常进行流程控制的嫌疑。

了解了这些情况以后，我们就能使用迭代器进行遍历了。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | it = iter(lst)  try:      while True:          val = it.next()          print val  except StopIteration:      pass |

实际上，因为迭代操作如此普遍，Python专门将关键字for用作了迭代器的语法糖。在for循环中，Python将自动调用工厂函数iter()获得迭代器，自动调用next()获取元素，还完成了检查StopIteration异常的工作。上述代码可以写成如下的形式，你一定非常熟悉：

|  |  |
| --- | --- |
| 1  2 | for val in lst:      print val |

首先Python将对关键字in后的对象调用iter函数获取迭代器，然后调用迭代器的next方法获取元素，直到抛出StopIteration异常。对迭代器调用iter函数时将返回迭代器自身，所以迭代器也可以用于for语句中，不需要特殊处理。

常用的几个内建数据结构tuple、list、set、dict都支持迭代器，字符串也可以使用迭代操作。你也可以自己实现一个迭代器，如上所述，只需要在类的\_\_iter\_\_方法中返回一个对象，这个对象拥有一个next()方法，这个方法能在恰当的时候抛出StopIteration异常即可。但是需要自己实现迭代器的时候不多，即使需要，使用生成器会更轻松。下一篇我们将讨论生成器的部分。

\*异常并不是非抛出不可的，不抛出该异常的迭代器将进行无限迭代，某些情况下这样的迭代器很有用。这种情况下，你需要自己判断元素并中止，否则就死循环了！

使用迭代器的循环可以避开索引，但有时候我们还是需要索引来进行一些操作的。这时候内建函数enumerate就派上用场咯，它能在iter函数的结果前加上索引，以元组返回，用起来就像这样：

|  |  |
| --- | --- |
| 1  2 | for idx, ele in enumerate(lst):      print idx, ele |

##### 3.3. 生成器表达式(Generator expression)和列表解析(List Comprehension)

绝大多数情况下，遍历一个集合都是为了对元素应用某个动作或是进行筛选。如果看过本文的第二部分，你应该还记得有内建函数map和filter提供了这些功能，但Python仍然为这些操作提供了语言级的支持。

|  |  |
| --- | --- |
| 1  2 | (x+1 for x in lst) #生成器表达式，返回迭代器。外部的括号可在用于参数时省略。  [x+1 for x in lst] #列表解析，返回list |

如你所见，生成器表达式和列表解析（注：这里的翻译有很多种，比如列表展开、列表推导等等，指的是同一个意思）的区别很小，所以人们提到这个特性时，简单起见往往只描述成列表解析。然而由于返回迭代器时，并不是在一开始就计算所有的元素，这样能得到更多的灵活性并且可以避开很多不必要的计算，所以除非你明确希望返回列表，否则应该始终使用生成器表达式。接下来的文字里我就不区分这两种形式了：）

你也可以为列表解析提供if子句进行筛选：

|  |  |
| --- | --- |
| 1 | (x+1 for x in lst if x!=0) |

或者提供多条for子句进行嵌套循环，嵌套次序就是for子句的顺序：

|  |  |
| --- | --- |
| 1 | ((x, y) for x in range(3) for y in range(x)) |

列表解析就是鲜明的Pythonic。我常遇到两个使用列表解析的问题，本应归属于最佳实践，但这两个问题非常典型，所以不妨在这里提一下：

第一个问题是，因为对元素应用的动作太复杂，不能用一个表达式写出来，所以不使用列表解析。这是典型的思想没有转变的例子，如果我们将动作封装成函数，那不就是一个表达式了么？

第二个问题是，因为if子句里的条件需要计算，同时结果也需要进行同样的计算，不希望计算两遍，就像这样：

|  |  |
| --- | --- |
| 1 | (x.doSomething() for x in lst if x.doSomething()>0) |

这样写确实很糟糕，但组合一下列表解析即可解决：

|  |  |
| --- | --- |
| 1 | (x for x in (y.doSomething() for y in lst) if x>0) |

内部的列表解析变量其实也可以用x，但为清晰起见我们改成了y。或者更清楚的，可以写成两个表达式：

|  |  |
| --- | --- |
| 1  2 | tmp = (x.doSomething() for x in lst)  (x for x in tmp if x > 0) |

列表解析可以替代绝大多数需要用到map和filter的场合，可能正因为此，著名的静态检查工具pylint将map和filter的使用列为了警告。

##### 3.4. 相关的库

Python内置了一个模块itertools，包含了很多函数用于creating iterators for efficient looping（创建更有效率的循环迭代器），这说明很是霸气，这一小节就来浏览一遍这些函数并留下印象吧，需要这些功能的时候隐约记得这里面有就好。这一小节的内容翻译自itertools模块官方文档。

###### 3.4.1. 无限迭代

* **count(start, [step])**   
  从start开始，以后每个元素都加上step。step默认值为1。   
  count(10) --> 10 11 12 13 14 ...
* **cycle(p)**   
  迭代至序列p的最后一个元素后，从p的第一个元素重新开始。   
  cycle('ABCD') --> A B C D A B C D ...
* **repeat(elem [,n])**   
  将elem重复n次。如果不指定n，则无限重复。   
  repeat(10, 3) --> 10 10 10

###### 3.4.2. 在最短的序列参数终止时停止迭代

* **chain(p, q, ...)**迭代至序列p的最后一个元素后，从q的第一个元素开始，直到所有序列终止。   
  chain('ABC', 'DEF') --> A B C D E F
* **compress(data, selectors)**如果bool(selectors[n])为True，则next()返回data[n]，否则跳过data[n]。   
  compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
* **dropwhile(pred, seq)**当pred对seq[n]的调用返回False时才开始迭代。   
  dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
* **takewhile(pred, seq)**dropwhile的相反版本。   
  takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
* **ifilter(pred, seq)**内建函数filter的迭代器版本。   
  ifilter(lambda x: x%2, range(10)) --> 1 3 5 7 9
* **ifilterfalse(pred, seq)**ifilter的相反版本。   
  ifilterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
* **imap(func, p, q, ...)**内建函数map的迭代器版本。   
  imap(pow, (2,3,10), (5,2,3)) --> 32 9 1000
* **starmap(func, seq)**将seq的每个元素以变长参数(\*args)的形式调用func。   
  starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
* **izip(p, q, ...)**内建函数zip的迭代器版本。   
  izip('ABCD', 'xy') --> Ax By
* **izip\_longest(p, q, ..., fillvalue=None)**   
  izip的取最长序列的版本，短序列将填入fillvalue。   
  izip\_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
* **tee(it, n)**返回n个迭代器it的复制迭代器。
* **groupby(iterable[, keyfunc])**这个函数功能类似于SQL的分组。使用groupby前，首先需要使用相同的keyfunc对iterable进行排序，比如调用内建的sorted函数。然后，groupby返回迭代器，每次迭代的元素是元组(key值, iterable中具有相同key值的元素的集合的子迭代器)。或许看看Python的排序指南对理解这个函数有帮助。   
  groupby([0, 0, 0, 1, 1, 1, 2, 2, 2]) --> (0, (0 0 0)) (1, (1 1 1)) (2, (2 2 2))

###### 3.4.3. 组合迭代器

* **product(p, q, ... [repeat=1])**笛卡尔积。   
  product('ABCD', repeat=2) --> AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
* **permutations(p[, r])**去除重复的元素。   
  permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
* **combinations(p, r)**   
  排序后去除重复的元素。   
  combinations('ABCD', 2) --> AB AC AD BC BD CD
* **combinations\_with\_replacement()**排序后，包含重复元素。   
  combinations\_with\_replacement('ABCD', 2) --> AA AB AC AD BB BC BD CC CD DD

此篇结束。

# [Python函数式编程指南（四）：生成器](http://www.cnblogs.com/huxi/archive/2011/07/14/2106863.html)

生成器是迭代器，同时也并不仅仅是迭代器，不过迭代器之外的用途实在是不多，所以我们可以大声地说：生成器提供了非常方便的自定义迭代器的途径。

这是函数式编程指南的最后一篇，似乎拖了一个星期才写好，嗯……

转载请注明原作者和原文地址：）

#### 4. 生成器(generator)

##### 4.1. 生成器简介

首先请确信，生成器就是一种迭代器。生成器拥有next方法并且行为与迭代器完全相同，这意味着生成器也可以用于Python的for循环中。另外，对于生成器的特殊语法支持使得编写一个生成器比自定义一个常规的迭代器要简单不少，所以生成器也是最常用到的特性之一。

从Python 2.5开始，[PEP 342：通过增强生成器实现协同程序]的实现为生成器加入了更多的特性，这意味着生成器还可以完成更多的工作。这部分我们会在稍后的部分介绍。

##### 4.2. 生成器函数

###### 4.2.1. 使用生成器函数定义生成器

如何获取一个生成器？首先来看一小段代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | >>> def get\_0\_1\_2():  ...   yield 0  ...   yield 1  ...   yield 2  ...  >>> get\_0\_1\_2  <function get\_0\_1\_2 at 0x00B2CB70> |

我们定义了一个函数get\_0\_1\_2，并且可以查看到这确实是函数类型。但与一般的函数不同的是，get\_0\_1\_2的函数体内使用了关键字yield，这使得get\_0\_1\_2成为了一个生成器函数。生成器函数的特性如下：

1. 调用生成器函数将返回一个生成器；

|  |  |
| --- | --- |
| 1  2  3 | >>> generator = get\_0\_1\_2()  >>> generator  <generator object get\_0\_1\_2 at 0x00B1C7D8> |

1. 第一次调用生成器的next方法时，生成器才开始执行生成器函数（而不是构建生成器时），直到遇到yield时暂停执行（挂起），并且yield的参数将作为此次next方法的返回值；

|  |  |
| --- | --- |
| 1  2 | >>> generator.next()  0 |

1. 之后每次调用生成器的next方法，生成器将从上次暂停执行的位置恢复执行生成器函数，直到再次遇到yield时暂停，并且同样的，yield的参数将作为next方法的返回值；

|  |  |
| --- | --- |
| 1  2  3  4 | >>> generator.next()  1  >>> generator.next()  2 |

1. 如果当调用next方法时生成器函数结束（遇到空的return语句或是到达函数体末尾），则这次next方法的调用将抛出StopIteration异常（即for循环的终止条件）；

|  |  |
| --- | --- |
| 1  2  3  4 | >>> generator.next()  Traceback (most recent call last):    File "<stdin>", line 1, in <module>  StopIteration |

1. 生成器函数在每次暂停执行时，函数体内的所有变量都将被封存(freeze)在生成器中，并将在恢复执行时还原，并且类似于闭包，即使是同一个生成器函数返回的生成器，封存的变量也是互相独立的。   
   我们的小例子中并没有用到变量，所以这里另外定义一个生成器来展示这个特点：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | >>> def fibonacci():  ...   a = b = 1  ...   yield a  ...   yield b  ...   while True:  ...     a, b = b, a+b  ...     yield b  ...  >>> for num in fibonacci():  ...   if num > 100: break  ...   print num,  ...  1 1 2 3 5 8 13 21 34 55 89 |

1. 看到while True可别太吃惊，因为生成器可以挂起，所以是延迟计算的，无限循环并没有关系。这个例子中我们定义了一个生成器用于获取斐波那契数列。

###### 4.2.2. 生成器函数的FAQ

接下来我们来讨论一些关于生成器的有意思的话题。

1. **你的例子里生成器函数都没有参数，那么生成器函数可以带参数吗？**   
   当然可以啊亲，而且它支持函数的所有参数形式。要知道生成器函数也是函数的一种：）

|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> def counter(start=0):  ...   while True:  ...     yield start  ...     start += 1  ... |

1. 这是一个从指定数开始的计数器。
2. **既然生成器函数也是函数，那么它可以使用return输出返回值吗？**不行的亲，是这样的，生成器函数已经有默认的返回值——生成器了，你不能再另外给一个返回值；对，即使是return None也不行。但是它可以使用空的return语句结束。如果你坚持要为它指定返回值，那么Python将在定义的位置赠送一个语法错误异常，就像这样：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> def i\_wanna\_return():  ...   yield None  ...   return None  ...    File "<stdin>", line 3  SyntaxError: 'return' with argument inside generator |

1. **好吧，那人家需要确保释放资源，需要在try...finally中yield，这会是神马情况？（我就是想玩你）我在finally中还yield了一次！**Python会在真正离开try...finally时再执行finally中的代码，而这里遗憾地告诉你，暂停不算哦！所以结局你也能猜到吧！

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | >>> def play\_u():  ...   try:  ...     yield 1  ...     yield 2  ...     yield 3  ...   finally:  ...     yield 0  ...  >>> for val in play\_u(): print val,  ...  1 2 3 0 |

1. \*这与return的情况不同。return是真正的离开代码块，所以会在return时立刻执行finally子句。   
   \*另外，“在带有finally子句的try块中yield”定义在PEP 342中，这意味着只有Python 2.5以上版本才支持这个语法，在Python 2.4以下版本中会得到语法错误异常。
2. **如果我需要在生成器的迭代过程中接入另一个生成器的迭代怎么办？写成下面这样好傻好天真。。**

|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> def sub\_generator():  ...   yield 1  ...   yield 2  ...   for val in counter(10): yield val  ... |

1. 这种情况的语法改进已经被定义在[PEP 380：委托至子生成器的语法]中，据说会在Python 3.3中实现，届时也可能回馈到2.x中。实现后，就可以这么写了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> def sub\_generator():  ...   yield 1  ...   yield 2  ...   yield from counter(10)    File "<stdin>", line 4      yield from counter(10)               ^  SyntaxError: invalid syntax |

1. 看到语法错误木有？现在我们还是天真一点吧~

有更多问题？请回复此文：）

##### 4.3. 协同程序(coroutine)

协同程序（协程）一般来说是指这样的函数：

* 彼此间有不同的局部变量、指令指针，但仍共享全局变量；
* 可以方便地挂起、恢复，并且有多个入口点和出口点；
* 多个协同程序间表现为协作运行，如A的运行过程中需要B的结果才能继续执行。

协程的特点决定了同一时刻只能有一个协同程序正在运行（忽略多线程的情况）。得益于此，协程间可以直接传递对象而不需要考虑资源锁、或是直接唤醒其他协程而不需要主动休眠，就像是内置了锁的线程。在符合协程特点的应用场景，使用协程无疑比使用线程要更方便。

从另一方面说，协程无法并发其实也将它的应用场景限制在了一个很狭窄的范围，这个特点使得协程更多的被拿来与常规函数进行比较，而不是与线程。当然，线程比协程复杂许多，功能也更强大，所以我建议大家牢牢地掌握线程即可：[Python线程指南](http://www.cnblogs.com/huxi/archive/2010/06/26/1765808.html)

这一节里我也就不列举关于协程的例子了，以下介绍的方法了解即可。

Python 2.5对生成器的增强实现了协程的其他特点，在这个版本中，生成器加入了如下方法：

1. **send(value):**   
   send是除next外另一个恢复生成器的方法。Python 2.5中，yield语句变成了yield表达式，这意味着yield现在可以有一个值，而这个值就是在生成器的send方法被调用从而恢复执行时，调用send方法的参数。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | >>> def repeater():  ...   n = 0  ...   while True:  ...     n = (yield n)  ...  >>> r = repeater()  >>> r.next()  0  >>> r.send(10)  10 |

1. \*调用send传入非None值前，生成器必须处于挂起状态，否则将抛出异常。不过，未启动的生成器仍可以使用None作为参数调用send。   
   \*如果使用next恢复生成器，yield表达式的值将是None。
2. **close():**   
   这个方法用于关闭生成器。对关闭的生成器后再次调用next或send将抛出StopIteration异常。
3. **throw(type, value=None, traceback=None):**   
   这个方法用于在生成器内部（生成器的当前挂起处，或未启动时在定义处）抛出一个异常。

\*别为没见到协程的例子遗憾，协程最常见的用处其实就是生成器。

##### 4.4. 一个有趣的库：pipe

这一节里我要向诸位简要介绍pipe。pipe并不是Python内置的库，如果你安装了easy\_install，直接可以安装它，否则你需要自己下载它：<http://pypi.python.org/pypi/pipe>

之所以要介绍这个库，是因为它向我们展示了一种很有新意的使用迭代器和生成器的方式：流。pipe将可迭代的数据看成是流，类似于linux，pipe使用'|'传递数据流，并且定义了一系列的“流处理”函数用于接受并处理数据流，并最终再次输出数据流或者是将数据流归纳得到一个结果。我们来看一些例子。

第一个，非常简单的，使用add求和：

|  |  |
| --- | --- |
| 1  2  3 | >>> from pipe import \*  >>> range(5) | add  10 |

求偶数和需要使用到where，作用类似于内建函数filter，过滤出符合条件的元素：

|  |  |
| --- | --- |
| 1  2 | >>> range(5) | where(lambda x: x % 2 == 0) | add  6 |

还记得我们定义的斐波那契数列生成器吗？求出数列中所有小于10000的偶数和需要用到take\_while，与itertools的同名函数有类似的功能，截取元素直到条件不成立：

|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> fib = fibonacci  >>> fib() | where(lambda x: x % 2 == 0)\  ...       | take\_while(lambda x: x < 10000)\  ...       | add  3382 |

需要对元素应用某个函数可以使用select，作用类似于内建函数map；需要得到一个列表，可以使用as\_list：

|  |  |
| --- | --- |
| 1  2 | >>> fib() | select(lambda x: x \*\* 2) | take\_while(lambda x: x < 100) | as\_list  [1, 1, 4, 9, 25, 64] |

pipe中还包括了更多的流处理函数。你甚至可以自己定义流处理函数，只需要定义一个生成器函数并加上修饰器Pipe。如下定义了一个获取元素直到索引不符合条件的流处理函数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> @Pipe  ... def take\_while\_idx(iterable, predicate):  ...   for idx, x in enumerate(iterable):  ...     if predicate(idx): yield x  ...     else: return  ... |

使用这个流处理函数获取fib的前10个数字：

|  |  |
| --- | --- |
| 1  2 | >>> fib() | take\_while\_idx(lambda x: x < 10) | as\_list  [1, 1, 2, 3, 5, 8, 13, 21, 34, 55] |

更多的函数就不在这里介绍了，你可以查看pipe的源文件，总共600行不到的文件其中有300行是文档，文档中包含了大量的示例。

pipe实现起来非常简单，使用Pipe装饰器，将普通的生成器函数（或者返回迭代器的函数）代理在一个实现了\_\_ror\_\_方法的普通类实例上即可，但是这种思路真的很有趣。

函数式编程指南全文到这里就全部结束了，希望这一系列文章能给你带来帮助。希望大家都能看到一些结构式编程之外的编程方式，并且能够熟练地在恰当的地方使用 ：）

明天我会整理一个目录放上来方便查看，并且列出一些供参考的文章。遗憾的是这些文章几乎都是英文的，请努力学习英语吧 - -#