### 15. \_\_new\_\_和\_\_init\_\_的区别

这个\_\_new\_\_确实很少见到,先做了解吧.

\_\_new\_\_是一个静态方法,而\_\_init\_\_是一个实例方法.

\_\_new\_\_方法会返回一个创建的实例,而\_\_init\_\_什么都不返回.

只有在\_\_new\_\_返回一个cls的实例时后面的\_\_init\_\_才能被调用.

当创建一个新实例时调用\_\_new\_\_,初始化一个实例时用\_\_init\_\_

ps: \_\_metaclass\_\_是创建类时起作用.所以我们可以分别使用\_\_metaclass\_\_,\_\_new\_\_和\_\_init\_\_来分别在类创建,实例创建和实例初始化的时候做一些小手脚.

### 16.  单例模式

上面的代码在单例模式中被称作，**懒汉式单例**.

# -\*- coding: utf-8 -\*-

from MyThread import \*

import threading

Lock = threading.Lock()

class Singleton(object):

# 定义静态变量实例

\_\_instance = None

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

pass

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if not cls.\_\_instance:

try:

Lock.acquire()

# double check

if not cls.\_\_instance:

cls.\_\_instance = super(Singleton, cls).\_\_new\_\_(cls, \*args, \*\*kwargs)

finally:

Lock.release()

return cls.\_\_instance

def test\_singleton\_in\_thread():

print id(Singleton())

if \_\_name\_\_ == "\_\_main\_\_":

idx = 0

while 1:

MyThread(test\_singleton\_in\_thread, []).start()

idx += 1

if idx > 0X100:

break

**装饰器版本的单例模式**：

def singleton(cls, \*args, \*\*kw):

instances = {}

def getinstance():

if cls not in instances:

instances[cls] = cls(\*args, \*\*kw)

return instances[cls]

return getinstance

@singleton

class MyClass:

print ‘a’

**import 方法**

# mysingleton.py

class My\_Singleton(object):

def foo(self):

pass

my\_singleton = My\_Singleton()

# to use

from mysingleton import my\_singleton

my\_singleton.foo()

### 17. python中的作用域

Python 中，一个变量的作用域总是由在代码中被赋值的地方所决定的。当 Python 遇到一个变量的话他会按照这样的顺序进行搜索：本地作用域（Local）→当前作用域被嵌入的本地作用域（Enclosing locals）→全局/模块作用域（Global）→内置作用域（Built-in）

### 18. GIL线程全局锁

线程全局锁(Global Interpreter Lock),即Python为了保证线程安全而采取的独立线程运行的限制,说白了就是一个核只能在同一时间运行一个线程.对于io密集型任务，python的多线程起到作用，但对于cpu密集型任务，python的多线程几乎占不到任何优势，还有可能因为争夺资源而变慢。

解决办法就是多进程和下面的协程(协程也只是单CPU,但是能减小切换代价提升性能).

**关于python的底层**

要理解GIL的含义，我们需要从Python的基础讲起。像C++这样的语言是编译型语言，所谓编译型语言，是指程序输入到编译器，编译器再根据语言的语法进行解析，然后翻译成语言独立的中间表示，最终链接成具有高度优化的机器码的可执行程序。编译器之所以可以深层次的对代码进行优化，是因为它可以看到整个程序（或者一大块独立的部分）。这使得它可以对不同的语言指令之间的交互进行推理，从而给出更有效的优化手段。

与此相反，Python是解释型语言。程序被输入到解释器来运行。解释器在程序执行之前对其并不了解；它所知道的只是Python的规则，以及在执行过程中怎样去动态的应用这些规则。它也有一些优化，但是这基本上只是另一个级别的优化。由于解释器没法很好的对程序进行推导，Python的大部分优化其实是解释器自身的优化。更快的解释器自然意味着程序的运行也能“免费”的更快。也就是说，解释器优化后，Python程序不用做修改就可以享受优化后的好处。

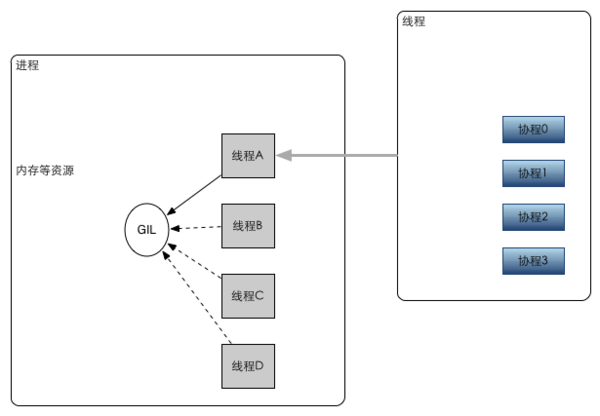
这一点很重要，让我们再强调一下。如果其他条件不变，Python程序的执行速度直接与解释器的“速度”相关。不管你怎样优化自己的程序，你的程序的执行速度还是依赖于解释器执行你的程序的效率。这就很明显的解释了为什么我们需要对优化Python解释器做这么多的工作了。对于Python程序员来说，这恐怕是与免费午餐最接近的了。

现在我们来看一下问题的症结所在。要想利用多核系统，Python必须支持多线程运行。作为解释型语言，Python的解释器必须做到既安全又高效。我们都知道多线程编程会遇到的问题。解释器要留意的是避免在不同的线程操作内部共享的数据。同时它还要保证在管理用户线程时保证总是有最大化的计算资源。

那么，不同线程同时访问时，数据的保护机制是怎样的呢？答案是解释器全局锁。从名字上看能告诉我们很多东西，很显然，这是一个加在解释器上的全局（从解释器的角度看）锁（从互斥或者类似角度看）。这种方式当然很安全，但是它有一层隐含的意思（Python初学者需要了解这个）：对于任何Python程序，不管有多少的处理器，任何时候都总是只有一个线程在执行。

**19. 线程，进程和协程**

* 线程（线程锁、threading.Event、queue 队列、生产者消费者模型、自定义线程池）
* 进程（数据共享、进程池）
* 协程



#### (1)线程

Threading用于提供线程相关的操作。线程是应用程序中工作的最小单元，它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

Threading 模块建立在 \_thread 模块之上。thread 模块以低级、原始的方式来处理和控制线程，而 threading 模块通过对 thread 进行二次封装，提供了更方便的 api 来处理线程。

import threading

import time

def worker(num):

    time.sleep(1)

    print(num)

    return

for i in range(10):

    t = threading.Thread(target=worker, args=(i,), name="t.%d" % i)

    t.start()

# 继承式调用

import threading

import time

class MyThread(threading.Thread):

    def \_\_init\_\_(self,num):

        threading.Thread.\_\_init\_\_(self)

        self.num = num

    def run(self):    #定义每个线程要运行的函数

        print("running on number:%s" %self.num)

        time.sleep(2)

if \_\_name\_\_ == '\_\_main\_\_':

    t1 = MyThread(1)

    t2 = MyThread(2)

    t1.start()

    t2.start()

##### thread方法

* t.start() : 激活线程
* t.getName() : 获取线程的名称
* t.setName() ： 设置线程的名称
* t.name : 获取或设置线程的名称
* t.is\_alive() ： 判断线程是否为激活状态
* t.isAlive() ：判断线程是否为激活状态
* t.setDaemon() 设置为后台线程或前台线程（默认：False）;通过一个布尔值设置线程是否为守护线程，必须在执行start()方法之前才可以使用。如果是后台线程，主线程执行过程中，后台线程也在进行，主线程执行完毕后，后台线程不论成功与否，均停止；如果是前台线程，主线程执行过程中，前台线程也在进行，主线程执行完毕后，等待前台线程也执行完成后，程序停止
* t.isDaemon() ： 判断是否为守护线程
* t.ident ：获取线程的标识符。线程标识符是一个非零整数，只有在调用了start()方法之后该属性才有效，否则它只返回None
* t.join() ：逐个执行每个线程，执行完毕后继续往下执行，该方法使得多线程变得无意义
* t.run() ：线程被cpu调度后自动执行线程对象的run方法

##### 线程锁

threading.RLock & threading.Lock

我们使用线程对数据进行操作的时候，如果多个线程同时修改某个数据，可能会出现不可预料的结果，为了保证数据的准确性，引入了锁的概念。

import threading

import time

num = 0

lock = threading.RLock()    # 实例化锁类

def work():

    lock.acquire()  # 加锁

    global num

    num += 1

    time.sleep(1)

    print(num)

    lock.release()  # 解锁

for i in range(10):

    t = threading.Thread(target=work)

    t.start()

##### 3 threading.RLock和threading.Lock 的区别

RLock允许在同一线程中被多次acquire。而Lock却不允许这种情况。 如果使用RLock，那么acquire和release必须成对出现，即调用了n次acquire，必须调用n次的release才能真正释放所占用的锁

import threading

lock = threading.Lock()

lock.acquire()

lock.acquire()  # 产生死锁

lock.release()

lock.release()

import threading

rlock = threading.RLock()

rlock.acquire()

rlock.acquire()      # 在同一线程内，程序不会堵塞。

rlock.release()

rlock.release()

print("end.")

##### 4 threading.Event

Event是线程间通信最间的机制之一：一个线程发送一个event信号，其他的线程则等待这个信号。用于主线程控制其他线程的执行。 Events 管理一个flag，这个flag可以使用set()设置成True或者使用clear()重置为False，wait()则用于阻塞，在flag为True之前。flag默认为False。

* Event.wait([timeout]) ： 堵塞线程，直到Event对象内部标识位被设为True或超时（如果提供了参数timeout）
* Event.set() ：将标识位设为Ture
* Event.clear() ： 将标识伴设为False
* Event.isSet() ：判断标识位是否为Ture

import threading

def do(event):

    print('start')

    event.wait()

    print('execute')

event\_obj = threading.Event()

for i in range(10):

    t = threading.Thread(target=do, args=(event\_obj,))

    t.start()

event\_obj.clear()

inp = input('input:')

if inp == 'true':

    event\_obj.set()

当线程执行的时候，如果flag为False，则线程会阻塞，当flag为True的时候，线程不会阻塞。它提供了本地和远程的并发性。

##### 5 threading.Condition

Python提供的Condition对象提供了对复杂线程同步问题的支持。Condition被称为条件变量，除了提供与Lock类似的acquire和release方法外，还提供了wait和notify方法。线程首先acquire一个条件变量，然后判断一些条件。如果条件不满足则wait；如果条件满足，进行一些处理改变条件后，通过notify方法通知其他线程，其他处于wait状态的线程接到通知后会重新判断条件。不断的重复这一过程，从而解决复杂的同步问题。

在典型的设计风格里，利用condition变量用锁去通许访问一些共享状态，线程在获取到它想得到的状态前，会反复调用wait()。修改状态的线程在他们状态改变时调用 notify() or notify\_all()，用这种方式，线程会尽可能的获取到想要的一个等待者状态。

import threading

import time<br>

def consumer(cond):

    with cond:

        print("consumer before wait")

        cond.wait()

        print("consumer after wait")

def producer(cond):

    with cond:

        print("producer before notifyAll")

        cond.notifyAll()

        print("producer after notifyAll")

condition = threading.Condition()

c1 = threading.Thread(name="c1", target=consumer, args=(condition,))

c2 = threading.Thread(name="c2", target=consumer, args=(condition,))

p = threading.Thread(name="p", target=producer, args=(condition,))

c1.start()

time.sleep(2)

c2.start()

time.sleep(2)

p.start()

# consumer()线程要等待producer()设置了Condition之后才能继续。

##### 6 ****queue 队列****

适用于多线程编程的先进先出数据结构，可以用来安全的传递多线程信息。

queue 方法：

* q = queue.Queue(maxsize=0) # 构造一个先进显出队列，maxsize指定队列长度，为0 时，表示队列长度无限制。
* q.join() 　　# 等到队列为kong的时候，在执行别的操作
* q.qsize() 　 # 返回队列的大小 （不可靠）
* q.empty()    # 当队列为空的时候，返回True 否则返回False （不可靠）
* q.full()     # 当队列满的时候，返回True，否则返回False （不可靠）
* q.put(item, block=True, timeout=None) # 将item放入Queue尾部，item必须存在，可以参数block默认为True,表示当队列满时，会等待队列给出可用位置，为False时为非阻塞，此时如果队列已满，会引发queue.Full 异常。 可选参数timeout，表示 会阻塞设置的时间，过后，如果队列无法给出放入item的位置，则引发 queue.Full 异常
* q.get(block=True, timeout=None) # 移除并返回队列头部的一个值，可选参数block默认为True，表示获取值的时候，如果队列为空，则阻塞，为False时，不阻塞，若此时队列为空，则引发 queue.Empty异常。 可选参数timeout，表示会阻塞设置的时候，过后，如果队列为空，则引发Empty异常。
* q.put\_nowait(item) # 等效于 put(item,block=False)
* q.get\_nowait()     # 等效于 get(item,block=False)

**生产者消费者模型**

import queue

import threading

que = queue.Queue(10)

def s(i):

    que.put(i)

    # print("size:", que.qsize())

def x(i):

    g = que.get(i)

    print("get:", g)

for i in range(1, 13):

    t = threading.Thread(target=s, args=(i,))

    t.start()

for i in range(1, 11):

    t = threading.Thread(target=x, args=(i,))

    t.start()

print("size:", que.qsize())

# 输出结果：

get: 1 get: 2 get: 3 get: 4 get: 5 get: 6 get: 7 get: 8 get: 9 get: 10 size: 2

**自定义线程池**

# 自定义线程池（一）

import queue

import threading

import time

class TreadPool:

def \_\_init\_\_(self, max\_num=20):

self.queue = queue.Queue(max\_num)

for i in range(max\_num):

self.queue.put(threading.Thread)

def get\_thread(self):

return self.queue.get()

def add\_thread(self):

self.queue.put(threading.Thread)

def func(pool, n):

time.sleep(1)

print(n)

pool.add\_thread()

p = TreadPool(10)

for i in range(1, 100):

thread = p.get\_thread()

t = thread(target=func, args=(p, i,))

t.start()

# 线程池（二）

import queue

import threading

import contextlib

import time

StopEvent = object()

class Threadpool:

def \_\_init\_\_(self, max\_num=10):

self.q = queue.Queue()

self.max\_num = max\_num

self.terminal = False

self.generate\_list = [] # 以创建线程列表

self.free\_list = [] # 以创建的线程空闲列表

def run(self, func, args, callback=None):

"""

线程池执行一个任务

:param func: 任务函数

:param args: 任务函数所需参数

:param callback: 任务执行失败或成功后执行的回调函数，

回调函数有两个参数

1、任务函数执行状态；

2、任务函数返回值（默认为None，即：不执行回调函数）

:return: 如果线程池已经终止，则返回True否则None

"""

if len(self.free\_list) == 0 and len(self.generate\_list) < self.max\_num:

self.generate\_thread()

w = (func, args, callback,)

self.q.put(w)

def generate\_thread(self):

"""

创建一个线程

"""

t = threading.Thread(target=self.call)

t.start()

def call(self):

"""

循环去获取任务函数并执行任务函数

"""

current\_thread = threading.currentThread # 当前线程

self.generate\_list.append(current\_thread)

event = self.q.get()

while event != StopEvent:

func, arguments, callback = event

try:

result = func(\*arguments)

status = True

except Exception as e:

status = False

result = e

if callback is not None:

try:

callback(status, result)

except Exception as e:

pass

if self.terminal:

event = StopEvent

else:

with self.worker\_state(self.free\_list, current\_thread):

event = self.q.get()

# self.free\_list.append(current\_thread)

# event = self.q.get()

# self.free\_list.remove(current\_thread)

else:

self.generate\_list.remove(current\_thread)

def close(self):

"""

执行完所有的任务后，所有线程停止

"""

num = len(self.generate\_list)

while num:

self.q.put(StopEvent)

num -= 1

def terminate(self):

"""

无论是否还有任务，终止线程

"""

self.terminal = True

while self.generate\_list:

self.q.put(StopEvent)

self.q.empty() # 清空队列

@contextlib.contextmanager # with上下文管理

def worker\_state(self, frelist, val):

"""

用于记录线程中正在等待的线程数

"""

frelist.append(val)

try:

yield

finally:

frelist.remove(val)

def work(i):

time.sleep(1)

print(i)

pool = Threadpool()

for item in range(50):

pool.run(func=work, args=(item,))

pool.close()

# pool.terminate()

#### (2) 进程

# 进程

from multiprocessing import Process

def work(name):

    print("Hello, %s" % name)

if \_\_name\_\_ == "\_\_main\_\_":

    p = Process(target=work, args=("nick",))

    p.start()

    p.join()

*注意：由于进程之间的数据需要各自持有一份，所以创建进程需要的非常大的开销。*

##### ****1 数据共享****

不同进程间内存是不共享的，要想实现两个进程间的数据交换，可以用以下方法：

**Shared memory**

数据可以用Value或Array存储在一个共享内存地图里，如下

from multiprocessing import Process, Value, Array

def f(n, a):

    n.value = 3.1415927

    for i in range(len(a)):

        a[i] = -a[i]

if \_\_name\_\_ == '\_\_main\_\_':

    num = Value('d', 0.0)

    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))

    p.start()

    p.join()

    print(num.value)

    print(arr[:])

# 输出：

3.1415927

[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

创建num和arr时，“d”和“i”参数由Array模块使用的typecodes创建：“d”表示一个双精度的浮点数，“i”表示一个有符号的整数，这些共享对象将被线程安全的处理。

‘c’: ctypes.c\_char

‘u’: ctypes.c\_wchar

‘b’: ctypes.c\_byte

‘B’: ctypes.c\_ubyte

‘h’: ctypes.c\_short

‘H’: ctypes.c\_ushort

‘i’: ctypes.c\_int

‘I’: ctypes.c\_uint

‘l’: ctypes.c\_long,

‘L’: ctypes.c\_ulong

‘f’: ctypes.c\_float

‘d’: ctypes.c\_double

from multiprocessing import Process,Array

temp = Array('i', [11,22,33,44])

def Foo(i):

    temp[i] = 100+i

    for item in temp:

        print i,'----->',item

for i in range(2):

    p = Process(target=Foo,args=(i,))

    p.start()

##### 2 Server process

由Manager()返回的manager提供list, dict, Namespace, Lock, RLock, Semaphore, BoundedSemaphore, Condition, Event, Barrier, Queue, Value and Array类型的支持。

from multiprocessing import Process, Manager

def f(d, l):

    d[1] = '1'

    d['2'] = 2

    d[0.25] = None

    l.reverse()

if \_\_name\_\_ == '\_\_main\_\_':

    with Manager() as manager:

        d = manager.dict()

        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))

        p.start()

        p.join()

        print(d)

        print(l)

# 输出结果：

{0.25: None, 1: '1', '2': 2}

[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

Server process manager比 shared memory 更灵活，因为它可以支持任意的对象类型。另外，一个单独的manager可以通过进程在网络上不同的计算机之间共享，不过他比shared memory要慢。

# manage.dict()共享数据

from multiprocessing import Process,Manager

manage = Manager()

dic = manage.dict()

def Foo(i):

    dic[i] = 100+i

    print dic.values()

for i in range(2):

    p = Process(target=Foo,args=(i,))

    p.start()

    p.join()

当创建进程时（非使用时），共享数据会被拿到子进程中，当进程中执行完毕后，再赋值给原值。

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

from multiprocessing import Process, Array, RLock

def Foo(lock,temp,i):

"""

将第0个数加100

"""

lock.acquire()

temp[0] = 100+i

for item in temp:

print i,'----->',item

lock.release()

lock = RLock()

temp = Array('i', [11, 22, 33, 44])

for i in range(20):

p = Process(target=Foo,args=(lock,temp,i,))

p.start()

##### 3 ****进程****池

进程池内部维护一个进程序列，当使用时，则去进程池中获取一个进程，如果进程池序列中没有可供使用的进进程，那么程序就会等待，直到进程池中有可用进程为止。

方法:

* apply(func[, args[, kwds]]) ：使用arg和kwds参数调用func函数，结果返回前会一直阻塞，由于这个原因，apply\_async()更适合并发执行，另外，func函数仅被pool中的一个进程运行。
* apply\_async(func[, args[, kwds[, callback[, error\_callback]]]]) ： apply()方法的一个变体，会返回一个结果对象。如果callback被指定，那么callback可以接收一个参数然后被调用，当结果准备好回调时会调用callback，调用失败时，则用error\_callback替换callback。 Callbacks应被立即完成，否则处理结果的线程会被阻塞。
* close() ： 阻止更多的任务提交到pool，待任务完成后，工作进程会退出。
* terminate() ： 不管任务是否完成，立即停止工作进程。在对pool对象进程垃圾回收的时候，会立即调用terminate()。
* join() : wait工作线程的退出，在调用join()前，必须调用close() or terminate()。这样是因为被终止的进程需要被父进程调用wait（join等价与wait），否则进程会成为僵尸进程

进程池中有两个方法：

* apply
* apply\_async

from multiprocessing import Pool

import time

def myFun(i):

    time.sleep(2)

    return i+100

def end\_call(arg):

    print("end\_call",arg)

p = Pool(5)

# print(p.map(myFun,range(10)))

for i in range(10):

    p.apply\_async(func=myFun,args=(i,),callback=end\_call)

print("end")

p.close()

p.join()

from multiprocessing import Pool, TimeoutError

import time

import os

def f(x):

return x\*x

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

# 创建4个进程

with Pool(processes=4) as pool:

# 打印 "[0, 1, 4,..., 81]"

print(pool.map(f, range(10)))

# 使用任意顺序输出相同的数字，

for i in pool.imap\_unordered(f, range(10)):

print(i)

# 异步执行"f(20)"

res = pool.apply\_async(f, (20,)) # 只运行一个进程

print(res.get(timeout=1)) # 输出 "400"

# 异步执行 "os.getpid()"

res = pool.apply\_async(os.getpid, ()) # 只运行一个进程

print(res.get(timeout=1)) # 输出进程的 PID

# 运行多个异步执行可能会使用多个进程

multiple\_results = [pool.apply\_async(os.getpid, ()) for i in range(4)]

print([res.get(timeout=1) for res in multiple\_results])

# 是一个进程睡10秒

res = pool.apply\_async(time.sleep, (10,))

try:

print(res.get(timeout=1))

except TimeoutError:

print("发现一个 multiprocessing.TimeoutError异常")

print("目前，池中还有其他的工作")

# 退出with块中已经停止的池

print("Now the pool is closed and no longer available")

#### (3) 协程

协程又叫微线程，从技术的角度来说，“协程就是你可以暂停执行的函数”。如果你把它理解成“就像生成器一样”，那么你就想对了。 线程和进程的操作是由程序触发系统接口，最后的执行者是系统；协程的操作则是程序员。

　　协程存在的意义：对于多线程应用，CPU通过切片的方式来切换线程间的执行，线程切换时需要耗时（保存状态，下次继续）。协程，则只使用一个线程，在一个线程中规定某个代码块执行顺序。

　　协程的适用场景：当程序中存在大量不需要CPU的操作时（IO），适用于协程。Python里最常见的yield就是协程的思想!可以查看第九个问题.

# 安装

pip install gevent

# 导入模块

import gevent

# greenlet

from greenlet import greenlet

def test1():

    print(11)

    gr2.switch()

    print(22)

    gr2.switch()

def test2():

    print(33)

    gr1.switch()

    print(44)

gr1 = greenlet(test1)

gr2 = greenlet(test2)

gr1.switch()

# 输出结果：

11 33 22 44

# gevent

import gevent

def foo():

    print("Running in foo")

    gevent.sleep(0)

    print("Explicit context switch to foo angin")

def bar():

    print("Explicit context to bar")

    gevent.sleep(0)

    print("Implicit context swich back to bar")

gevent.joinall([

    gevent.spawn(foo),

    gevent.spawn(bar),

])

# 输出结果：

Running in foo

Explicit context to bar

Explicit context switch to foo angin

Implicit context swich back to bar

# 遇到IO自动切换

from gevent import monkey

monkey.patch\_all()

import gevent

import requests

def f(url):

print("FET: %s" % url)

resp = requests.get(url)

data = len(resp.text)

print(url, data)

gevent.joinall([

gevent.spawn(f, 'https://www.python.org/'),

gevent.spawn(f, 'https://www.yahoo.com/'),

gevent.spawn(f, 'https://github.com/'),

])

### 19． 闭包

闭包(closure)是函数式编程的重要的语法结构。闭包也是一种组织代码的结构，它同样提高了代码的可重复使用性。

当一个内嵌函数引用其外部作作用域的变量,我们就会得到一个闭包. 总结一下,创建一个闭包必须满足以下几点:

1. 必须有一个内嵌函数
2. 内嵌函数必须引用外部函数中的变量
3. 外部函数的返回值必须是内嵌函数

感觉闭包还是有难度的,几句话是说不明白的,还是查查相关资料.

重点是函数运行后并不会被撤销,就像16题的instance字典一样,当函数运行完后,instance并不被销毁,而是继续留在内存空间里.这个功能类似类里的类变量,只不过迁移到了函数上.闭包就像个空心球一样,你知道外面和里面,但你不知道中间是什么样.

简单说,闭包就是根据不同的配置信息得到不同的结果

再来看看专业的解释:闭包（Closure）是词法闭包（Lexical Closure）的简称，是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。

在函数编程中经常用到闭包。闭包是什么，它是怎么产生的及用来解决什么问题呢。给出字面的定义先：**闭包是由函数及其相关的引用环境组合而成的实体(即：闭包=函数+引用环境)(想想Erlang的外层函数传入一个参数a, 内层函数依旧传入一个参数b, 内层函数使用a和b, 最后返回内层函数)**。这个从字面上很难理解，特别对于一直使用命令式语言进行编程的程序员们。本文将结合实例代码进行解释。  
函数是什么

地球人都知道：函数只是一段可执行代码，编译后就“固化”了，每个函数在内存中只有一份实例，得到函数的入口点便可以执行函数了。在函数式编程语言中，函 数是一等公民（First class value：第一类对象，我们不需要像命令式语言中那样借助函数指针，委托操作函数），函数可以作为另一个函数的参数或返回值，可以赋给一个变量。函数可 以嵌套定义，即在一个函数内部可以定义另一个函数，有了嵌套函数这种结构，便会产生闭包问题。如：

>>> def ExFunc(n):  
      sum=n  
     def InsFunc():  
             return sum+1  
     return InsFunc

>>> myFunc=ExFunc(10)  
>>> myFunc()  
11  
>>> myAnotherFunc=ExFunc(20)  
>>> myAnotherFunc()  
21  
>>> myFunc()  
11  
>>> myAnotherFunc()  
21

这段程序中，函数InsFunc是函数ExFunc的内嵌函数，并且是ExFunc函数的返回值。我们注意到一个问题：内嵌函数InsFunc中 引用到外层函数中的局部变量sum，IronPython会这么处理这个问题呢？先让我们来看看这段代码的运行结果。当我们调用分别由不同的参数调用 ExFunc函数得到的函数时（myFunc()，myAnotherFunc()），得到的结果是隔离的，也就是说每次调用ExFunc函数后都将生成并保存一个新的局部变量sum。其实这里ExFunc函数返回的就是闭包。

引用环境

按照命令式语言的规则，ExFunc函数只是返回了内嵌函数InsFunc的地址，在执行InsFunc函数时将会由于在其作用域内找不到sum变量而出 错。而在函数式语言中，当内嵌函数体内引用到体外的变量时，将会把定义时涉及到的引用环境和函数体打包成一个整体（**闭包**）返回。现在给出引用环境的定义就 容易理解了：引用环境是指在程序执行中的某个点所有处于活跃状态的约束（一个变量的名字和其所代表的对象之间的联系）所组成的集合。闭包的使用和正常的函 数调用没有区别。

由于闭包把函数和运行时的引用环境打包成为一个新的整体，所以就解决了函数编程中的嵌套所引发的问题。如上述代码段中，当每次调用ExFunc函数 时都将返回一个新的闭包实例，这些实例之间是隔离的，分别包含调用时不同的引用环境现场。不同于函数，闭包在运行时可以有多个实例，不同的引用环境和相同 的函数组合可以产生不同的实例。

##### 一，定义

python中的闭包从表现形式上定义（解释）为：如果在一个内部函数里，对在外部作用域（但不是在全局作用域）的变量进行引用，那么内部函数就被认为是闭包(closure).这个定义是相对直白的，好理解的，不像其他定义那样学究味道十足（那些学究味道重的解释，在对一个名词的解释过程中又充满了一堆让人抓狂的其他陌生名词，不适合初学者）。下面举一个简单的例子来说明。

>>>def addx(x):

>>> def adder(y): return x + y

>>> return adder

>>> c = addx(8)

>>> type(c)

<type 'function'>

>>> c.\_\_name\_\_

'adder'

>>> c(10)

>>> # 输出18

结合这段简单的代码和定义来说明闭包：如果在一个内部函数里：adder(y)就是这个内部函数，对在外部作用域（但不是在全局作用域）的变量进行引用：x就是被引用的变量，x在外部作用域addx里面，但不在全局作用域里，则这个内部函数adder就是一个闭包。

再稍微讲究一点的解释是，闭包=函数块+定义函数时的环境，adder就是函数块，x就是环境，当然这个环境可以有很多，不止一个简单的x。

##### 二，使用闭包注意事项

1，闭包中是不能修改外部作用域的局部变量的】

>>> def foo():

... m = 0

... def foo1():

... m = 1

... print m

...

... print m

... foo1()

... print m

...

>>> foo() #输出0 1 0

从执行结果可以看出，虽然在闭包里面也定义了一个变量m，但是其不会改变外部函数中的局部变量m。

2，以下这段代码是在python中使用闭包时一段经典的错误代码

def foo():

a = 1

def bar():

a = a + 1

return a

return bar

这段程序的本意是要通过在每次调用闭包函数时都对变量a进行递增的操作。但在实际使用时

>>> c = foo()

>>> print c()

Traceback (most recent call last):

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

File "<stdin>", line 4, in bar

UnboundLocalError: local variable 'a' referenced before assignment

这是因为在执行代码 c = foo()时，python会导入全部的闭包函数体bar()来分析其的局部变量，python规则指定所有在赋值语句左面的变量都是局部变量，则在闭包bar()中，变量a在赋值符号"="的左面，被python认为是bar()中的局部变量。再接下来执行print c()时，程序运行至a = a + 1时，因为先前已经把a归为bar()中的局部变量，所以python会在bar()中去找在赋值语句右面的a的值，结果找不到，就会报错。解决的方法很简单

def foo():

a = [1]

def bar():

a[0] = a[0] + 1

return a[0]

return bar

只要将a设定为一个容器就可以了。这样使用起来多少有点不爽，所以在python3以后，在a = a + 1 之前，使用语句nonloacal a就可以了，该语句显式的指定a不是闭包的局部变量。

3，还有一个容易产生错误的事例也经常被人在介绍python闭包时提起，我一直都没觉得这个错误和闭包有什么太大的关系，但是它倒是的确是在python函数式编程是容易犯的一个错误，我在这里也不妨介绍一下。先看下面这段代码

for i in range(3):

print i

在程序里面经常会出现这类的循环语句，Python的问题就在于，当循环结束以后，循环体中的临时变量i不会销毁，而是继续存在于执行环境中。还有一个python的现象是，python的函数只有在执行时，才会去找函数体里的变量的值。

flist = []

for i in range(3):

def foo(x): print x + i

flist.append(foo)

for f in flist:

f(2)

可能有些人认为这段代码的执行结果应该是2,3,4.但是实际的结果是4,4,4。这是因为当把函数加入flist列表里时，python还没有给i赋值，只有当执行时，再去找i的值是什么，这时在第一个for循环结束以后，i的值是2，所以以上代码的执行结果是4,4,4.

解决方法也很简单，改写一下函数的定义就可以了。

for i in range(3):

def foo(x,y=i): print x + y

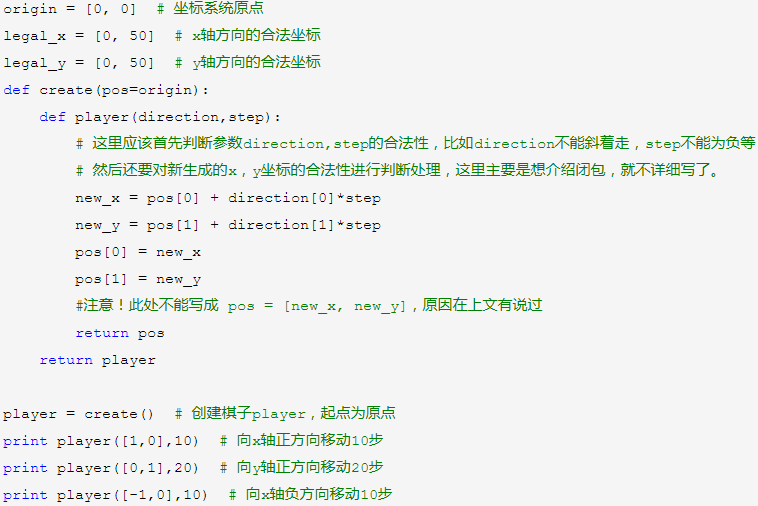
flist.append(foo)

##### 三，作用

说了这么多，不免有人要问，那这个闭包在实际的开发中有什么用呢？闭包主要是在函数式开发过程中使用。以下介绍两种闭包主要的用途。

用途1，当闭包执行完后，仍然能够保持住当前的运行环境。

比如说，如果你希望函数的每次执行结果，都是基于这个函数上次的运行结果。我以一个类似棋盘游戏的例子来说明。假设棋盘大小为50\*50，左上角为坐标系原点(0,0)，我需要一个函数，接收2个参数，分别为方向(direction)，步长(step)，该函数控制棋子的运动。棋子运动的新的坐标除了依赖于方向和步长以外，当然还要根据原来所处的坐标点，用闭包就可以保持住这个棋子原来所处的坐标。



输出为：

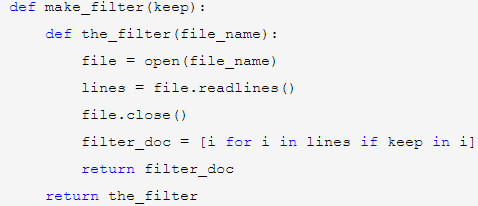
[10, 0]

[10, 20]

[0, 20]

用途2，闭包可以根据外部作用域的局部变量来得到不同的结果

这有点像一种类似配置功能的作用，我们可以修改外部的变量，闭包根据这个变量展现出不同的功能。比如有时我们需要对某些文件的特殊行进行分析，先要提取出这些特殊行。



如果我们需要取得文件"result.txt"中含有"pass"关键字的行，则可以这样使用例子程序

filter = make\_filter("pass")

filter\_result = filter("result.txt")

以上两种使用场景，用面向对象也是可以很简单的实现的，但是在用Python进行函数式编程时，闭包对数据的持久化以及按配置产生不同的功能，是很有帮助的。

### 21. lambda函数

其实就是一个匿名函数,为什么叫lambda?因为和后面的函数式编程有关.

一些有趣的函数：

Python内置了一些特殊函数，这些函数很具python特性。可以让代码更加简洁。

可以看例子：

1 **filter(function, sequence)**：

str = ['a', 'b','c', 'd']

def fun1(s): return s if s != 'a' else None

ret = filter(fun1, str)

print ret

## ['b', 'c', 'd']

对sequence中的item依次执行function(item)，将执行结果为True的item组成一个List/String/Tuple（取决于sequence的类型）返回。

可以看作是过滤函数。

 2 **map(function, sequence)**

str = ['a', 'b','c', 'd']

def fun2(s): return s + ".txt"

ret = map(fun2, str)

print ret

## ['a.txt', 'b.txt', 'c.txt', 'd.txt']

对sequence中的item依次执行function(item)，见执行结果组成一个List返回：map也支持多个sequence，这就要求function也支持相应数量的参数输入：

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

print map(add, range(10), range(10))   
##[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

**3 reduce(function, sequence, starting\_value)**：def add1(x,y): return x + y

print reduce(add1, range(1, 100))

print reduce(add1, range(1, 100), 20)

## 4950 （注：1+2+...+99）  
## 4970 （注：1+2+...+99+20）

对sequence中的item顺序迭代调用function，如果有starting\_value，还可以作为初始值调用，例如可以用来对List求和：

**4 lambda**：

g = lambda s: s + ".fsh"

print g("haha")

print (lambda x: x \* 2) (3)

## haha.fsh

## 6

这是Python支持一种有趣的语法，它允许你快速定义单行的最小函数，类似与C语言中的宏，这些叫做lambda的函数.

### 22. 函数式编程

当我们说起函数式编程来说，我们会看到如下函数式编程的长相：

##### 函数式编程的三大特性：

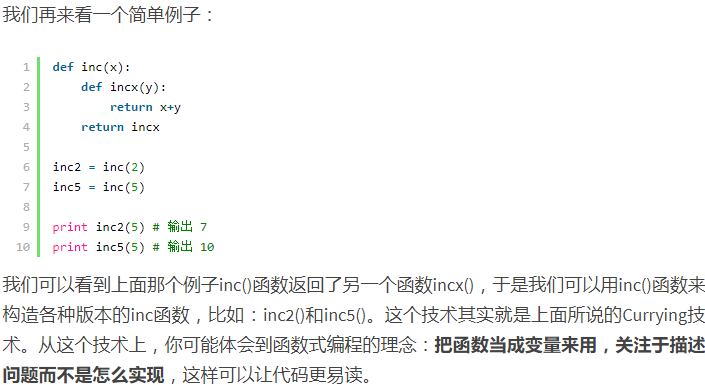
* **immutable data 不可变数据**：像Clojure一样，默认上变量是不可变的，如果你要改变变量，你需要把变量copy出去修改。这样一来，可以让你的程序少很多Bug。因为，程序中的状态不好维护，在并发的时候更不好维护。（你可以试想一下如果你的程序有个复杂的状态，当以后别人改你代码的时候，是很容易出bug的，在并行中这样的问题就更多了）
* **first class functions**：这个技术可以让你的函数就像变量一样来使用。也就是说，你的函数可以像变量一样被创建，修改，并当成变量一样传递，返回或是在函数中嵌套函数。这个有点像Javascript的Prototype（参看[Javascript的面向对象编程](https://coolshell.cn/articles/6668.html)）
* **尾递归优化**：我们知道递归的害处，那就是如果递归很深的话，stack受不了，并会导致性能大幅度下降。所以，我们使用尾递归优化技术——每次递归时都会重用stack，这样一来能够提升性能，当然，这需要语言或编译器的支持。Python就不支持。

##### 函数式编程的几个技术

* **map & reduce** ：这个技术不用多说了，函数式编程最常见的技术就是对一个集合做Map和Reduce操作。这比起过程式的语言来说，在代码上要更容易阅读。（传统过程式的语言需要使用for/while循环，然后在各种变量中把数据倒过来倒过去的）这个很像C++中的STL中的foreach，find\_if，count\_if之流的函数的玩法。
* **pipeline**：这个技术的意思是，把函数实例成一个一个的action，然后，把一组action放到一个数组或是列表中，然后把数据传给这个action list，数据就像一个pipeline一样顺序地被各个函数所操作，最终得到我们想要的结果。
* **recursing 递归** ：递归最大的好处就简化代码，他可以把一个复杂的问题用很简单的代码描述出来。注意：递归的精髓是描述问题，而这正是函数式编程的精髓。
* **currying**：把一个函数的多个参数分解成多个函数， 然后把函数多层封装起来，每层函数都返回一个函数去接收下一个参数这样，可以简化函数的多个参数。在C++中，这个很像STL中的bind\_1st或是bind2nd。
* **higher order function 高阶函数**：所谓高阶函数就是函数当参数，把传入的函数做一个封装，然后返回这个封装函数。现象上就是函数传进传出，就像面向对象对象满天飞一样。

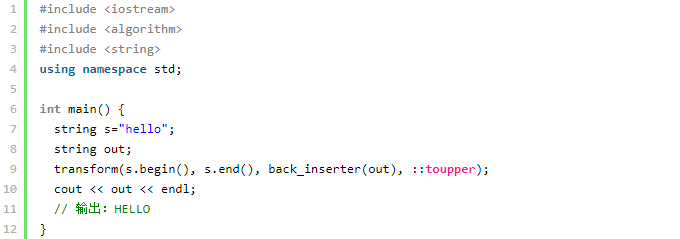
##### 还有函数式的一些好处

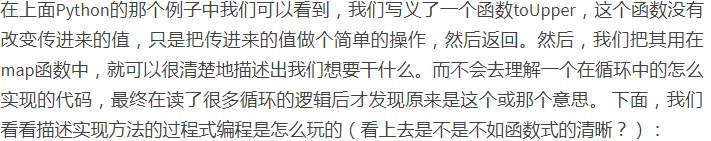
* **parallelization 并行：**所谓并行的意思就是在并行环境下，各个线程之间不需要同步或互斥。
* **lazy evaluation 惰性求值**：这个需要编译器的支持。表达式不在它被绑定到变量之后就立即求值，而是在该值被取用的时候求值，也就是说，语句如*x:=expression;* (把一个表达式的结果赋值给一个变量)明显的调用这个表达式被计算并把结果放置到 *x* 中，但是先不管实际在 *x* 中的是什么，直到通过后面的表达式中到 *x* 的引用而有了对它的值的需求的时候，而后面表达式自身的求值也可以被延迟，最终为了生成让外界看到的某个符号而计算这个快速增长的依赖树。
* **determinism 确定性**：所谓确定性的意思就是像数学那样 f(x) = y ，这个函数无论在什么场景下，都会得到同样的结果，这个我们称之为函数的确定性。而不是像程序中的很多函数那样，同一个参数，却会在不同的场景下计算出不同的结果。所谓不同的场景的意思就是我们的函数会根据一些运行中的状态信息的不同而发生变化。

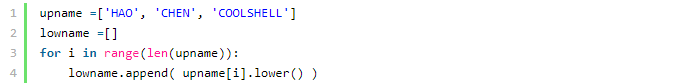


##### Map & Reduce

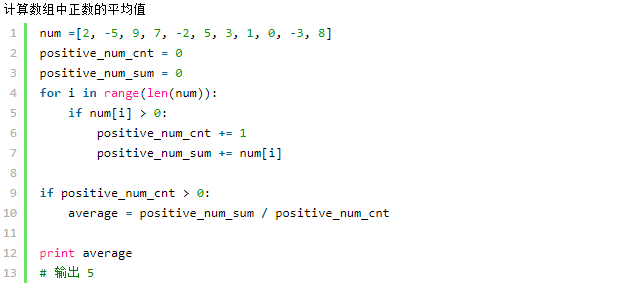


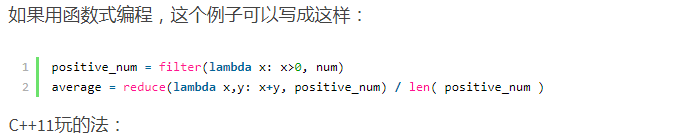


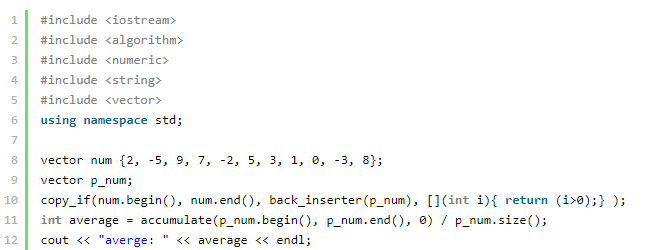


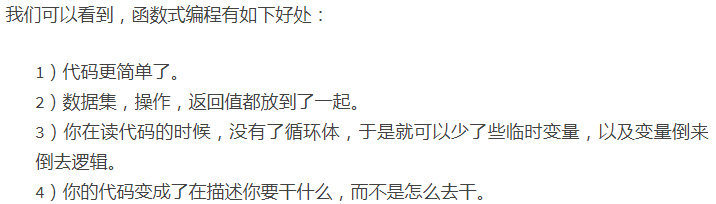


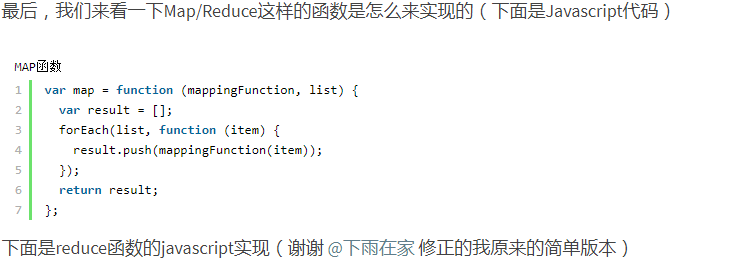


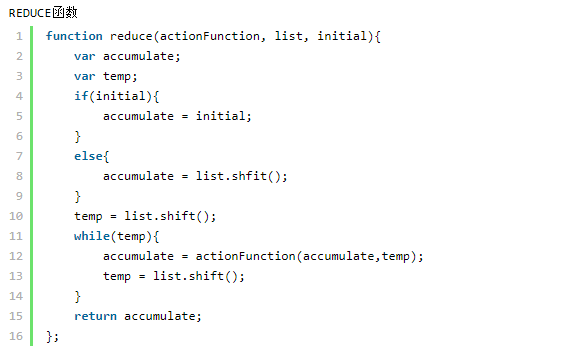




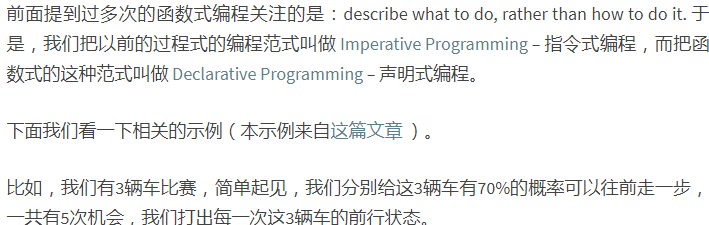




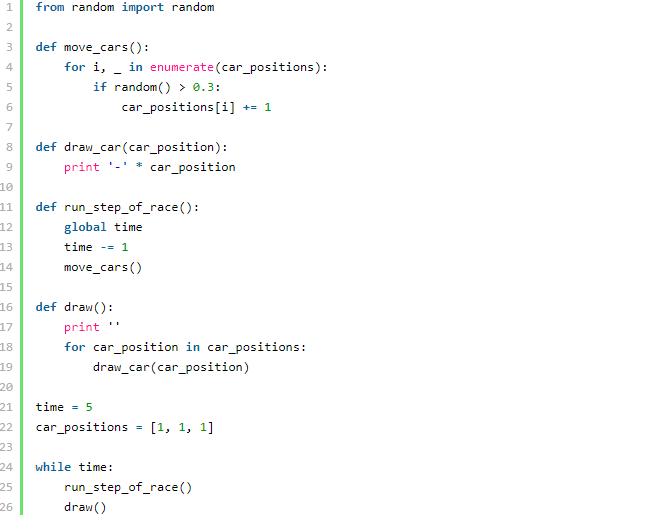


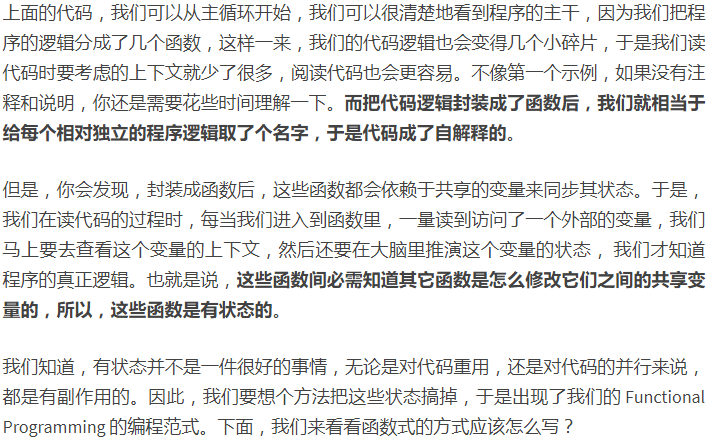


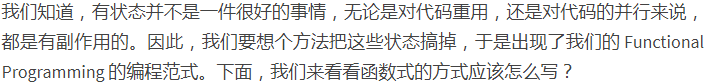
##### Declarative Programming vs Imperative Programming

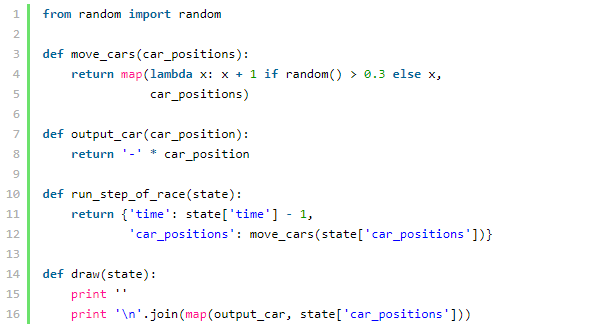


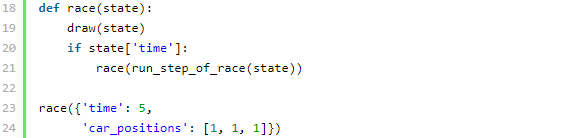


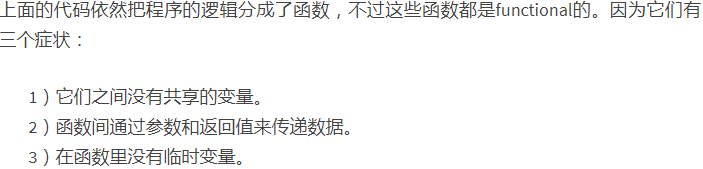






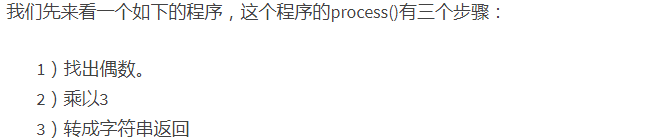
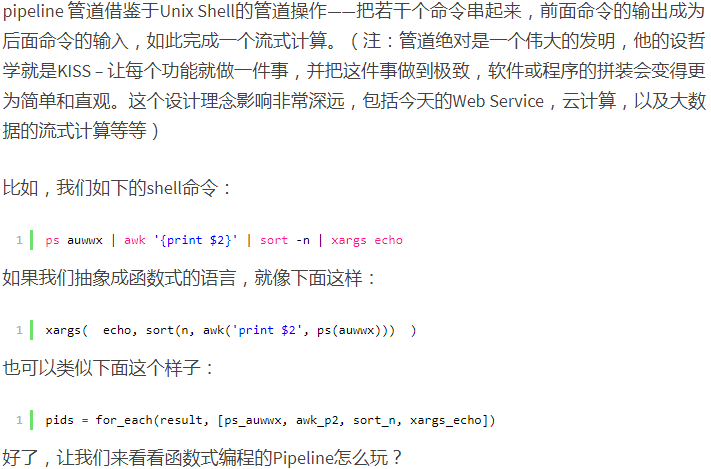


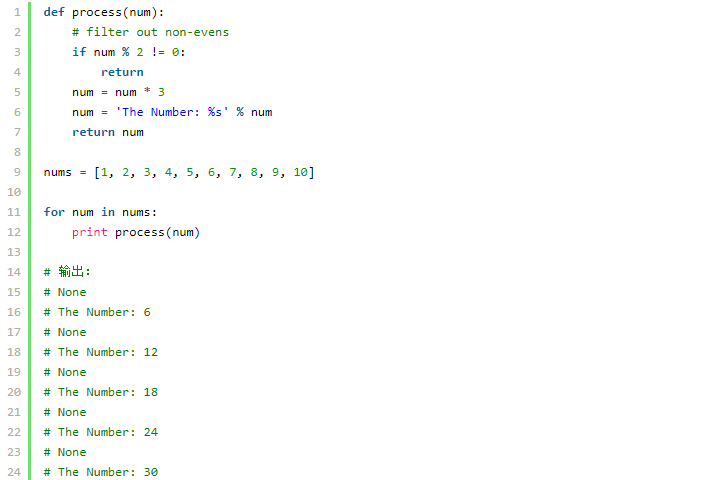




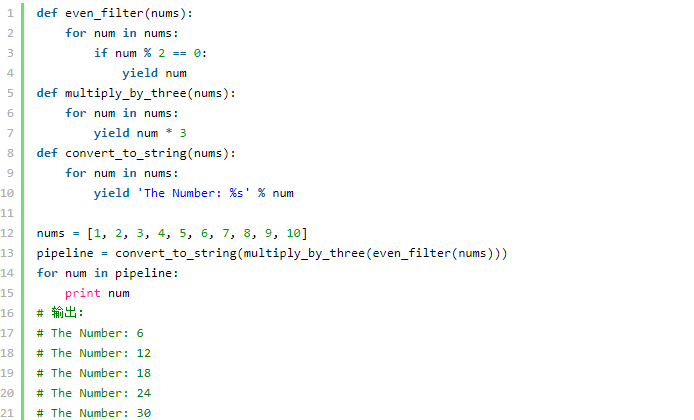


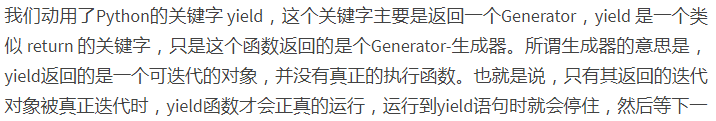
##### Pipeline





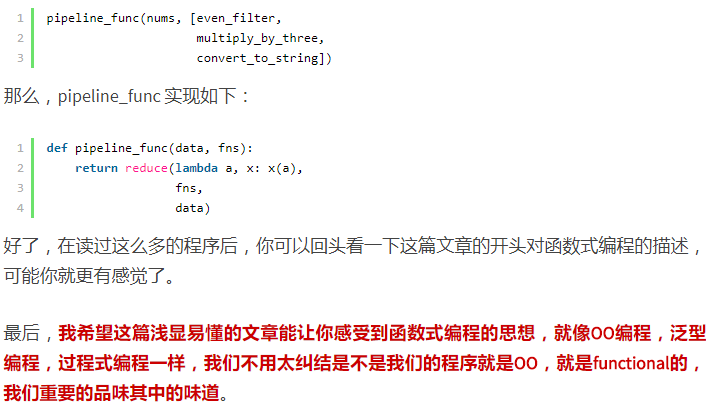


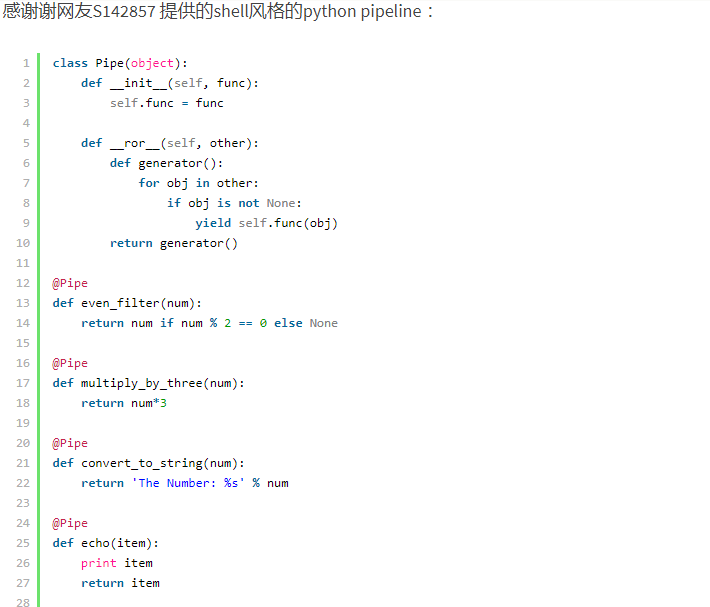


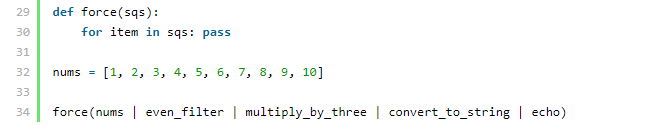












### 23. Python里的拷贝



在python中，对象赋值实际上是对象的引用。当创建一个对象，然后把它赋给另一个变量的时候，python并没有拷贝这个对象，而只是拷贝了这个对象的引用

(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

(3) 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

### 24. 垃圾回收

Python的GC模块主要运用了“引用计数”（reference counting）来跟踪和回收垃圾。在引用计数的基础上，还可以通过“标记-清除”（mark and sweep）解决容器对象可能产生的循环引用的问题。通过“分代回收”（generation collection）以空间换取时间来进一步提高垃圾回收的效率。

即：**python采用的是引用计数机制为主，标记-清除和分代收集两种机制为辅的策略**

##### （1）引用计数

python里每一个东西都是对象，它们的核心就是一个结构体：PyObject

typedef struct\_object {

int ob\_refcnt;

struct\_typeobject \*ob\_type;

} PyObject;

` PyObject是每个对象必有的内容，其中ob\_refcnt就是做为引用计数。当一个对象有新的引用时，它的ob\_refcnt就会增加，当引用它的对象被删除，它的ob\_refcnt就会减少。当引用计数为0时，该对象生命就结束了。

引用计数机制的优点：

1. 简单
2. 实时性：一旦没有引用，内存就直接释放了。不用像其他机制等到特定时机。实时性还带来一个好处：处理回收内存的时间分摊到了平时。

引用计数机制的缺点：

1. 维护引用计数消耗资源
2. 循环引用

list1 = []

list2 = []

list1.append(list2)

list2.append(list1)

list1与list2相互引用，如果不存在其他对象对它们的引用，list1与list2的引用计数也仍然为1，所占用的内存永远无法被回收，这将是致命的。  
对于如今的强大硬件，缺点1尚可接受，但是循环引用导致内存泄露，注定python还将引入新的回收机制。(标记清除和分代收集)