## sys模块

### 1.1 sys.argv

实现从程序外向程序传递参数。

# File echo.py

import sys

print sys.argv

%python echo.py -a -b -c

### [*'echo.py'*, *'a'*, *'b'*, *'c'*]

### 1.2 sys.exit([arg])

程序中间退出，arg=0位正常退出，sys.exit()退出程序会引发SystemExit异常，可以捕获异常执行清理工作。

### 1.3 sys.getdefaultencoding()

获取系统当前编码

### 1.4 sys.setdefaultencoding()

设置系统默认编码，执行dir（sys）时不会看到这个方法，在解释器中执行不通过，可以先执行reload(sys)，在执行 setdefaultencoding('utf8')，此时将系统默认编码设置为utf8。（见设置系统默认编码 ）

### 1.5 sys.getfilesystemencoding()

获取文件系统默认编码方式

### 1.6 sys.path

sys.path是python的搜索模块的路径集，是一个list。可以在python 环境下使用sys.path.append(path)添加相关的路径，但在退出python环境后自己添加的路径就会自动消失！

### 1.7 sys.platform

获取当前系统平台,只是python解释器这个系统，不是操作系统.

### 1.8 python重定向sys.stdin、sys.stdout和sys.stderr

标准输入、标准输出和错误输出。

### 1.9 sys.moudles

sys.modules是一个全局字典，该字典是python启动后就加载在内存中。每当程序员导入新的模块，sys.modules将自动记录该模块。当第二次再导入该模块时，python会直接到字典中查找，从而加快了程序运行的速度。它拥有字典所拥有的一切方法.

### 1.10 sys.getrefcount

sys.getrefcount返回对象的引用值

### 1.11 示例

sys.stdin标准输入一般是键盘。stdin对象为解释器提供输入字符流，一般使用raw\_input()和input()函数。

>>> name = sys.stdin.readline()

xiaoming #输入

>>> name

'xiaoming\n'

>>>

sys.stdout.write()方法其实就是下面所讲的标准输出，print语句就是调用了这个方法。

标准输出：一般是屏幕。stdout对象接收到print语句产生的输出。

# cat a.py

import sys

sys.stdout.write("123456\n")

sys.stdout.flush()

# cat b.py

import sys

print sys.stdin.readlines()

# python a.py | python b.py

['123456\n']

sys.stdout是有缓冲区的。

sys.stderr错误输出：一般是错误信息。stderr对象接收出错的信息。

 sys.stdout与print

当我们在 [Python](http://lib.csdn.net/base/python" \o "Python知识库" \t "http://blog.csdn.net/bbbeoy/article/details/_blank) 中打印对象调用 print obj 时候，事实上是调用了 sys.stdout.write(obj+'\n') ；print 将你需要的内容打印到了控制台，然后追加了一个换行符；print 会调用 sys.stdout 的 write 方法.

sys.stdin与raw\_input:

当我们用 raw\_input('Input promption: ') 时，事实上是先把提示信息输出，然后捕获输入,如下是等价的:

hi=raw\_input('hello? ')

print 'hello? ', #comma to stay in the same line hi=sys.stdin.readline()[:-1] # -1 to discard the '\n' in input stream

#mysys.py

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

*"""*

*sys模块*

*"""*

import sys

def **test\_argv**():

print sys.argv #实现从程序外部向程序传递参数：['C:\\E\\workspace\\PythonLearning\\modules\\mysys.py']

def **test\_exit**():

print *"exec test exit!"*

try:

sys.exit(1)

except SystemExit, e:

print e #捕获退出码：1

def **test\_encoding**():

print sys.getdefaultencoding() #系统默认编码方式：ascii

reload(sys)

sys.setdefaultencoding(*'utf-8'*)

print sys.getdefaultencoding() #utf-8

print sys.getfilesystemencoding() #获取文件系统编码方式:mbcs

def **test\_path**():

print sys.path

def **test\_platform**():

print sys.platform #获取当前系统平台,只是python解释器这个系统，不是操作系统

def **test\_moudles**():

print sys.modules

FUNC\_LIST = [test\_argv, test\_exit, test\_encoding, test\_path,test\_platform,test\_moudles]

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

for func in FUNC\_LIST:

print *'\*'* \* 60

func()

## random模块

### 2.1 random.random()

用于生成一个0到1的随机符点数: 0 <= n < 1.0

### 2.2 random.uniform()

random.uniform的函数原型为：random.uniform(a, b)，用于生成一个指定范围内的随机符点数，两个参数其中一个是上限，一个是下限。如果a > b，则生成的随机数n: a <= n <= b。如果 a <b， 则 b <= n <= a。

### 2.3 random.randint()

　random.randint()的函数原型为：random.randint(a, b)，用于生成一个指定范围内的整数。其中参数a是下限，参数b是上限，生成的随机数n: a <= n <= b

### 2.4 random.randrange()

random.randrange的函数原型为：random.randrange([start], stop[, step])，从指定范围内，按指定基数递增的集合中获取一个随机数。如：random.randrange(10, 100, 2)，结果相当于从[10, 12, 14, 16, ... 96, 98]序列中获取一个随机数。random.randrange(10, 100, 2)在结果上与 random.choice(range(10, 100, 2) 等效。

### 2.5 random.choice()

random.choice从序列中获取一个随机元素。其函数原型为：random.choice(sequence)。参数sequence表示一个有序类型。这里要说明 一下：sequence在**[python](http://lib.csdn.net/base/python" \o "Python知识库" \t "http://blog.csdn.net/xiaocaiju/article/details/_blank)**不是一种特定的类型，而是泛指一系列的类型。list, tuple,字符串都属于sequence。

### 2.6 random.shuffle()

random.shuffle的函数原型为：random.shuffle(x[, random])，用于将一个列表中的元素打乱.

### 2.7 random.sample()

random.sample的函数原型为：random.sample(sequence, k)，从指定序列中随机获取指定长度的片断。sample函数不会修改原有序列。

### 2.8 random.seed()

seed( ) 用于指定随机数生成时所用算法开始的整数值，如果使用相同的seed( )值，则每次生成的随即数都相同，如果不设置这个值，则系统根据时间来自己选择这个值，此时每次生成的随机数因时间差异而不同。

>>> import random

>>> num = 0

>>> while num < 5:

random.seed(5)

print random.random()

num += 1

0.62290169489

0.62290169489

0.62290169489

0.62290169489

0.62290169489

### 2.9 示例

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

*"""*

*random模块---Python中的random模块用于生成随机数。*

*"""*

import random

def **func\_random**():

print random.random() #用于生成一个0到1的随机符点数: 0 <= n < 1.0

def **func\_uniform**():

print random.uniform(1.0, 20.0) #于生成一个指定范围内的随机符点数

def **func\_randint**():

print random.randint(1,10) #用于生成指定范围内的整数

def **func\_randrange**():

print random.randrange(1,100, 2) #从指定范围内，按指定基数递增的集合中获取一个随机数

def **func\_choice**():

print random.choice(*"hello"*) #从序列中获取一个随机元素

print random.choice([10, 9 ,8 ,7 ,6])

def **func\_shuffle**():

alist = [1, 2, 3, 4]

random.shuffle(alist) #用于将列表中的元素打乱

print alist

def **func\_sample**():

alist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

aslice = random.sample(alist, 5) #从list中随机获取5个元素，作为一个片断返回

print aslice

print alist #原有序列并没有改变。

FUNC\_LIST = [func\_random, func\_uniform, func\_randint ,func\_randrange,

func\_choice, func\_shuffle, func\_sample]

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

for func in FUNC\_LIST:

print *'\*'* \* 30 + *"%s"* % func.\_\_name\_\_ + *'\*'* \* 30

func()

执行结果：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*func\_random\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

0.630245705103

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*func\_uniform\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

8.73697484016

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*func\_randint\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

10

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*func\_randrange\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

57

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*func\_choice\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

o

8

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*func\_shuffle\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[2, 1, 4, 3]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*func\_sample\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[2, 8, 7, 6, 9]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

## OS模块

参见 Python专题——文件处理.docx

## decimal模块

decimal模块用于十进制数学计算，它具有以下特点：

* 提供十进制数据类型，并且存储为十进制数序列；
* 有界精度：用于存储数字的位数是固定的，可以通过decimal.getcontext（）.prec=x 来设定，不同的数字可以有不同的精度
* 浮点：十进制小数点的位置不固定（但位数是固定的）

# 设置全局精度

a = decimal.Decimal(1) / decimal.Decimal(7)

print a # 0.1428571428571428571428571429

decimal.getcontext().prec = 6

b = decimal.Decimal(1) / decimal.Decimal(7)

print b # 0.142857

小数上下文管理器：

>>> import decimal as D

>>> with D.localcontext() as ctx:

ctx.prec = 2

c = D.Decimal('1.0')/D.Decimal('3.0')

print c # 0.33

## threading模块

threading通过对thread模块进行二次封装，提供了更方便的API来操作线程。

### 5.1 threading.Thread创建线程

Thread 是threading模块中最重要的类之一，可以使用它来创建线程。有两种方式来创建线程：

* 通过继承Thread类，重写它的run方法；
* 创建一个threading.Thread对象，在它的初始化函数（\_\_init\_\_）中将可调用对象作为参数传入。

#!/usr/bin/env python2

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

import threading

import time

import random

count = 0

class **Counter**(threading.Thread):

def **\_\_init\_\_**(*self*, lock, thread\_name):

*'''@summary: 初始化线程对象。*

**@param** *lock: 琐对象。*

**@param** *thread\_name: 线程名称。*

*'''*

super(Counter, *self*).\_\_init\_\_(name=thread\_name)

*self*.lock = lock

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

*'''@summary: 重写父类run方法，在线程启动后执行该方法内的代码。*

*'''*

global count

*self*.lock.acquire()

for i in xrange(10000):

count += 1

*self*.lock.release()

def **main**()

print *"Start Main Thread"*

lock = threading.Lock()

for t in [Counter(lock, *'Thread-%d'* %i) for i in range(5)]:

t.start()

time.sleep(2)

print count

print *"End Main Thread"*

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

main()

执行结果：

Start Main Thread

50000

End Main Thread

在代码中，我们创建了一个Counter类，它继承了threading.Thread。初始化函数接收两个参数，一个是锁对象，另一个是线程的名称。在Counter类中，重写了从父类继承的run方法，run方法将一个全局变量逐一的增加10000。在接下来的代码中，创建了五个Counter对象，分别调用其start方法。最后打印结果。这里要说明一下run方法 和start方法: 它们都是从Thread继承而来的，run()方法将在线程开启后执行，可以把相关的逻辑写到run方法中（通常把run方法称为活动[Activity]。）；start()方法用于启动线程。

另外一种创建线程的方法：

#!/usr/bin/env python2

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

import threading

import time

import random

count = 0

lock = threading.Lock()

def **do\_add**():

global count

global lock

lock.acquire()

for i in xrange(10000):

count += 1

lock.release()

def **main**():

print *"Start Main Thread"*

threads = [threading.Thread(target = do\_add, args = (), name = *'Thread-%d'* %i) for i in range(3)]

for t in threads:

print t.name

t.start()

time.sleep(2) #确保线程都执行完毕，主线程不会提前退出

print count

print *"End Main Thread"*

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

main()

执行结果：

Start Main Thread

Thread-0

Thread-1

Thread-2

30000

End Main Thread

定义了方法do\_add，它将全局变量count 逐一的增加10000。然后创建了3个Thread对象，把函数对象do\_add 作为参数传给它的初始化函数，再调用Thread对象的start方法，线程启动后将执行do\_add函数。这里有必要介绍一下threading.Thread类的初始化函数原型：

def \_\_init\_\_(self, group=None, target=None, name=None, args=(), kwargs={})

* 参数group是预留的，用于将来扩展；
* 参数target是一个可调用对象（也称为活动[activity]），在线程启动后执行；
* 参数name是线程的名字。默认值为“Thread-N“，N是一个数字。
* 参数args和kwargs分别表示调用target时的参数列表和关键字参数。

### 5.2 Thread类常用属性和方法

class Thread(\_Verbose)

| A class that represents a thread of control.

|

| This class can be safely subclassed in a limited fashion.

|

| Method resolution order:

| Thread

| \_Verbose

| \_\_builtin\_\_.object

|

| Methods defined here:

|

| \_\_init\_\_(self, group=None, target=None, name=None, args=(), kwargs=None, verbose=None)

| This constructor should always be called with keyword arguments. Arguments are:

|

| \*group\* should be None; reserved for future extension when a ThreadGroup

| class is implemented.

|

| \*target\* is the callable object to be invoked by the run()

| method. Defaults to None, meaning nothing is called.

|

| \*name\* is the thread name. By default, a unique name is constructed of

| the form "Thread-N" where N is a small decimal number.

|

| \*args\* is the argument tuple for the target invocation. Defaults to ().

|

| \*kwargs\* is a dictionary of keyword arguments for the target

| invocation. Defaults to {}.

|

| If a subclass overrides the constructor, it must make sure to invoke

| the base class constructor (Thread.\_\_init\_\_()) before doing anything

| else to the thread.

|

| \_\_repr\_\_(self)

|

| getName(self)

|

| isAlive(self)

| Return whether the thread is alive.

|

| This method returns True just before the run() method starts until just

| after the run() method terminates. The module function enumerate()

| returns a list of all alive threads.

|

| isDaemon(self)

|

| is\_alive = isAlive(self)

|

| join(self, timeout=None)

| Wait until the thread terminates.

|

| This blocks the calling thread until the thread whose join() method is

| called terminates -- either normally or through an unhandled exception

| or until the optional timeout occurs.

|

| When the timeout argument is present and not None, it should be a

| floating point number specifying a timeout for the operation in seconds

| (or fractions thereof). As join() always returns None, you must call

| isAlive() after join() to decide whether a timeout happened -- if the

| thread is still alive, the join() call timed out.

|

| When the timeout argument is not present or None, the operation will

| block until the thread terminates.

|

| A thread can be join()ed many times.

|

| join() raises a RuntimeError if an attempt is made to join the current

| thread as that would cause a deadlock. It is also an error to join() a

| thread before it has been started and attempts to do so raises the same

| exception.

|

| run(self)

| Method representing the thread's activity.

|

| You may override this method in a subclass. The standard run() method

| invokes the callable object passed to the object's constructor as the

| target argument, if any, with sequential and keyword arguments taken

| from the args and kwargs arguments, respectively.

|

| setDaemon(self, daemonic)

|

| setName(self, name)

|

| start(self)

| Start the thread's activity.

|

| It must be called at most once per thread object. It arranges for the

| object's run() method to be invoked in a separate thread of control.

|

| This method will raise a RuntimeError if called more than once on the

| same thread object.

|

| ----------------------------------------------------------------------

| Data descriptors defined here:

|

| daemon

| A boolean value indicating whether this thread is a daemon thread (True) or not (False).

|

| This must be set before start() is called, otherwise RuntimeError is

| raised. Its initial value is inherited from the creating thread; the

| main thread is not a daemon thread and therefore all threads created in

| the main thread default to daemon = False.

|

| The entire Python program exits when no alive non-daemon threads are

| left.

|

| ident

| Thread identifier of this thread or None if it has not been started.

|

| This is a nonzero integer. See the thread.get\_ident() function. Thread

| identifiers may be recycled when a thread exits and another thread is

| created. The identifier is available even after the thread has exited.

|

| name

| A string used for identification purposes only.

|

| It has no semantics. Multiple threads may be given the same name. The

| initial name is set by the constructor.

|

| ----------------------------------------------------------------------

| Data descriptors inherited from \_Verbose:

|

| \_\_dict\_\_

| dictionary for instance variables (if defined)

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

**Thread.getName(self)**

**Thread.setName(self, name)**

**Thread.name**

* 用于获取和设置线程的名称。

**Thread.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。

**Thread.is\_alive(self)**

**Thread.isAlive(self)**

* 判断线程是否是激活的（alive）。从调用start()方法启动线程，到run()方法执行完毕或遇到未处理异常而中断这段时间内，线程是激活的。

**Thread.join(self, timeout=None)**

* 调用Thread.join将会使主调线程堵塞，直到被调用线程运行结束或超时。参数timeout是一个数值类型，表示超时时间，如果未提供该参数，那么主调线程将一直堵塞到被调线程结束。

主线程A中，创建了子线程B，并且在主线程A中调用了B.join()，那么，主线程A会在调用的地方等待，直到子线程B完成操作后，才可以接着往下执行，那么在调用这个线程时可以使用被调用线程的join方法。

原型：join([timeout])

 里面的参数时可选的，代表线程运行的最大时间，即如果超过这个时间，不管这个此线程有没有执行完毕都会被回收，然后主线程或函数都会接着执行的。

**threading.RLock()**

**threading.Lock()**

在threading模块中，定义两种类型的琐：threading.Lock和threading.RLock。它们之间有一点细微的区别，通过比较下面两段代码来说明：

程序1：

#!/usr/bin/env python2

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

import threading

import time

lock = threading.Lock() # Lock()对象

lock.acquire()

lock.acquire() # 产生了死锁

lock.release()

lock.release()

print *"end"*

*程序2：*

#!/usr/bin/env python2

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

import threading

import time

lock = threading.RLock() # RLock对象

lock.acquire()

lock.acquire() # 在同一个线程内，程序不会阻塞

lock.release()

lock.release()

print *"end"*

这两种琐的主要区别是：RLock允许在同一线程中被多次acquire。而Lock却不允许这种情况。注意：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琐。

重入锁（ReentrantLock）是一种递归无阻塞的同步机制。RLock 将由最近成功获得锁定，并且还没有释放该锁定的线程所拥有。当锁定没有被另一个线程所拥有时，调用 lock 的线程将成功获取该锁定并返回。如果当前线程已经拥有该锁定，此方法将立即返回。可以使用 isHeldByCurrentThread() 和 getHoldCount() 方法来检查此情况是否发生。它提供了lock()方法：  
如果该锁定没有被另一个线程保持，则获取该锁定并立即返回，将锁定的保持计数设置为 1。  
如果当前线程已经保持该锁定，则将保持计数加 1，并且该方法立即返回。  
如果该锁定被另一个线程保持，则出于线程调度的目的，禁用当前线程，并且在获得锁定之前，该线程将一直处于休眠状态，此时锁定保持计数被设置为 1。

**threading.Condition（\*args, \*\*kwargs）**

可以把Condition理解为一把高级的锁，它提供了比Lock, RLock更高级的功能，允许我们能够控制复杂的线程同步问题。threadiong.Condition在内部维护一个锁对象（默认是RLock），可以在创建Condition对象的时候把锁对象作为参数传入。Condition也提供了acquire, release方法，其含义与锁的acquire, release方法一致，其实它只是简单的调用内部锁对象的对应的方法而已。Condition还提供了如下方法(特别要注意：这些方法只有在占用琐(acquire)之后才能调用，否则将会报RuntimeError异常。)：

**Condition.wait([timeout]):**

wait方法释放内部所占用的琐，同时线程被挂起，直至接收到通知被唤醒或超时（如果提供了timeout参数的话）。当线程被唤醒并重新占有琐的时候，程序才会继续执行下去。

**Condition.notify():**

唤醒一个挂起的线程（如果存在挂起的线程）。注意：notify()方法不会释放所占用的琐。

**Condition.notify\_all()**

**Condition.notifyAll()**

唤醒所有挂起的线程（如果存在挂起的线程）。注意：这些方法不会释放所占用的琐。

#!/usr/bin/env python2

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

import threading, time

class **Hider**(threading.Thread):

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

super(Hider, *self*).\_\_init\_\_()

*self*.cond = cond

*self*.name = name

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

time.sleep(1)

*self*.cond.acquire() #b

print *self*.name + *': 我已经把眼睛蒙上了'*

*self*.cond.notify()

*self*.cond.wait() #c

#f

print *self*.name + *': 我找到你了 ~\_~'*

*self*.cond.notify()

*self*.cond.release()

#g

print *self*.name + *': 我赢了'* #h

class **Seeker**(threading.Thread):

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

super(Seeker, *self*).\_\_init\_\_()

*self*.cond = cond

*self*.name = name

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

*self*.cond.acquire()

*self*.cond.wait() #a #释放对琐的占用，同时线程挂起在这里，直到被notify并重新占有琐。

#d

print *self*.name + *': 我已经藏好了，你快来找我吧'*

*self*.cond.notify()

*self*.cond.wait() #e

#h

*self*.cond.release()

print *self*.name + *': 被你找到了，哎~~~'*

cond = threading.Condition()

seeker = Seeker(cond, *'seeker'*)

hider = Hider(cond, *'hider'*)

seeker.start()

hider.start()

**threading.Event**

Event实现与Condition类似的功能，不过比Condition简单一点。它通过维护内部的标识符来实现线程间的同步问题。（threading.Event和.NET中的System.Threading.ManualResetEvent类实现同样的功能。）

**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。

#!/usr/bin/env python2

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

import threading

import time

class **Seeker**(threading.Thread):

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

super(Seeker, *self*).\_\_init\_\_(name = name)

*self*.cond = cond

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

time.sleep(1) # 确保先运行Hidler中的方法

print *self*.name + *':我已经把眼睛蒙上了'*

*self*.cond.set()

time.sleep(1)

*self*.cond.wait()

print *self*.name + *':我找到你了'*

*self*.cond.set()

print *self*.name + *': 我赢了'*

class **Hidler**(threading.Thread):

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

super(Hidler, *self*).\_\_init\_\_(name = name)

*self*.cond = cond

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

*self*.cond.wait()

print *self*.name + *': 我已经藏好了'*

time.sleep(1)

*self*.cond.wait()

print *self*.name + *': 被你找到了'*

cond = threading.Event()

seeker = Seeker(cond, name=*'seeker'*)

hidler = Hidler(cond, name = *'hidler'*)

seeker.start()

hidler.start()

执行结果：

seeker:我已经把眼睛蒙上了

hidler: 我已经藏好了

seeker:我找到你了

hidler: 被你找到了

seeker: 我赢了

**threading.Timer**

是threading.Thread的子类，可以在指定时间间隔后执行某个操作。

例如：

def **hello**():

print *'hello'*

t = threading.Timer(3, hello)

t.start()

**threading.active\_count()  
threading.activeCount()**

* 获取当前活动的(alive)线程的个数。

**threading.current\_thread()  
threading.currentThread()**

* 获取当前的线程对象（Thread object）。

**threading.enumerate()**

* 获取当前所有活动线程的列表。

**threading.settrace(func)**

* 设置一个跟踪函数，用于在run()执行之前被调用。

**threading.setprofile(func)**

* 设置一个跟踪函数，用于在run()执行完毕之后调用。

## multiprocessing模块

### 6.1 Process类

**创建进程的类**：Process([group [, target [, name [, args [, kwargs]]]]])

* target表示调用对象
* args表示调用对象的位置参数元组。
* kwargs表示调用对象的字典。name为别名。
* group实质上不使用。

**方法**：

* is\_alive() 判断进程是否是激活的
* join([timeout]) 等待直到子进程终止
* run() 在子进程中运行的对象，可以被重写
* start() 启动进程
* terminate() 终止进程：发送SIGTERM信号或使用终止过程()

其中，Process以start()启动某个进程。

**属性**：authkey、daemon（要通过start()设置）、exitcode(进程在运行时为None、如果为–N，表示被信号N结束）、name、pid。其中daemon是父进程终止后自动终止，且自己不能产生新进程，必须在start()之前设置。

#!/usr/bin/env python2

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

import multiprocessing

import time

import sys

import os

if sys.platform[:3] == *"win"*:

timefunc = time.clock

else:

timefunc = time.time

def **task\_1**():

start = timefunc()

print *"Task1 Start %s"* % time.strftime(*"%Y-%m-%d %X"*, time.localtime())

with open(*'task1.txt'*, *'w'*) as fp:

for i in range(10000000):

if i%1000 == 0:

fp.write(*'a'*\*100 + os.linesep)

print *'Task1 Used %s s'* % (timefunc()-start)

def **task\_2**():

start = timefunc()

print *"Task2 Start %s"* % time.strftime(*"%Y-%m-%d %X"*, time.localtime())

with open(*'task2.txt'*, *'w'*) as fp:

for i in range(10000000):

if i%1000 == 0:

fp.write(*'a'*\*100 + os.linesep)

print *'Task2 Used %s s'* % (timefunc()-start)

def **task\_3**():

start = timefunc()

print *"Task3 Start %s"* % time.strftime(*"%Y-%m-%d %X"*, time.localtime())

with open(*'task3.txt'*, *'w'*) as fp:

for i in range(10000000):

if i%1000 == 0:

fp.write(*'a'*\*100 + os.linesep)

print *'Task3 Used %s s'* % (timefunc()-start)

def **main**():

start = timefunc()

print *"Main start %s"* % time.strftime(*"%Y-%m-%d %X"*, time.localtime())

p1 = multiprocessing.Process(target=task\_1)

p2 = multiprocessing.Process(target=task\_2)

p3 = multiprocessing.Process(target=task\_3)

p1.start()

p2.start()

p3.start()

print *"The number of CPU is:%s"* % multiprocessing.cpu\_count() # cpu核数

for p in multiprocessing.active\_children():

print *"child p.name:"* + p.name + *"\tp.id"* + str(p.pid) # 打印进程名、进程id

for t in [p1, p2, p3]:

t.join()

print *"Main Used %s"* % (timefunc()-start)

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

main()

执行结果：

Main start 2017-11-03 19:09:13

The number of CPU is:4

child p.name:Process-3 p.id7380

child p.name:Process-1 p.id5960

child p.name:Process-2 p.id6712

Task2 Start 2017-11-03 19:09:13

Task3 Start 2017-11-03 19:09:13

Task1 Start 2017-11-03 19:09:13

Task2 Used 1.5251817162 s

Task3 Used 1.56424922805 s

Task1 Used 1.70972561199 s

Main Used 1.90023822653

**把进程创建成类**：

#!/usr/bin/env python2

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

import multiprocessing

import sys

import os

import time

if sys.platform[:3] == "win":

timefunc = time.clock

else:

timefunc = time.time

count = 0

class MyProcessor(multiprocessing.Process):

def run(self):

start = timefunc()

print 'Start process:%s id:%s' % (self.name, self.pid)

global count

for i in range(10000000):

count += 1

print 'End process:%s id:%s Used time:%f S count=%d' % (self.name, self.pid,

timefunc()-start, count)

def main():

start = timefunc()

global count

print 'Start Main at %s' % time.strftime("%Y-%m-%d %X", time.localtime())

procs = [MyProcessor() for i in range(3)]

for p in procs:

p.start()

for p in procs:

p.join()

print "Main count=%d" % count

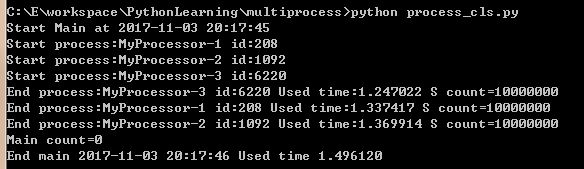
print "End main %s Used time %f" %(time.strftime("%Y-%m-%d %X", time.localtime()),

timefunc()-start)

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

main()

执行结果：



**daemon属性**

如果子进程设置了daemon属性为True，主进程结束，它们就随着结束了。

**例1**：

#!/usr/bin/env python2

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

import multiprocessing

import time

def **test**():

print *'Start Test'*

time.sleep(2)

print *'End Test'*

def **main**():

proc = multiprocessing.Process(target=test)

proc.start()

print *'End'*

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

main()

执行结果：（先执行主进程，后执行子线程）

End

Start Test

End Test

**例2**：

#!/usr/bin/env python2

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

import multiprocessing

import time

def **test**():

print *'Start Test'*

time.sleep(2)

print *'End Test'*

def **main**():

proc = multiprocessing.Process(target=test)

proc.start()

proc.join()

print *'End'*

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

main()

执行结果：（join方法使主进程阻塞，直到子进程结束）

Start Test

End Test

End

例3：

#!/usr/bin/env python2

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

import multiprocessing

import time

def **test**():

print *'Start Test'*

time.sleep(2)

print *'End Test'*

def **main**():

proc = multiprocessing.Process(target=test)

proc.daemon = True

proc.start()

print *'End'*

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

main()

执行结果：（daemon设置为True，子进程为守护进程，主进程退出它就退出）

End

### 6.2 Lock

当多个进程需要访问共享资源的时候，Lock可以用来避免访问的冲突。

#!/usr/bin/env python2

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

import multiprocessing

import sys

def **worker\_with**(lock, f):

with lock:

fs = open(f, *'a+'*)

n = 10

while n > 1:

fs.write(*"Lockd acquired via with\n"*)

n -= 1

fs.close()

def **worker\_no\_with**(lock, f):

lock.acquire()

try:

fs = open(f, *'a+'*)

n = 10

while n > 1:

fs.write(*"Lock acquired directly\n"*)

n -= 1

fs.close()

finally:

lock.release()

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

lock = multiprocessing.Lock()

f = *"lock.txt"*

w = multiprocessing.Process(target = worker\_with, args=(lock, f))

nw = multiprocessing.Process(target = worker\_no\_with, args=(lock, f))

w.start()

nw.start()

print *"end"*

*执行结果：lock.txt*

Lockd acquired via with

Lockd acquired via with

Lockd acquired via with

Lockd acquired via with

Lockd acquired via with

Lockd acquired via with

Lockd acquired via with

Lockd acquired via with

Lockd acquired via with

Lock acquired directly

Lock acquired directly

Lock acquired directly

Lock acquired directly

Lock acquired directly

Lock acquired directly

Lock acquired directly

Lock acquired directly

Lock acquired directly

### 6.3 Semaphore

Semaphore用来控制对共享资源的访问数量，例如池的最大连接数

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

import multiprocessing

import sys

import time

def **worker**(s, i):

s.acquire()

print multiprocessing.current\_process().name + *"acquire"*

time.sleep(i)

print multiprocessing.current\_process().name + *"release\n"*

s.release()

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

S = multiprocessing.Semaphore(2)

procs = [multiprocessing.Process(target = worker, args=(S, 3)) for i in range(1, 6)]

for p in procs:

p.start()

for p in procs:

p.join()

print *'End'*

*执行结果：*

Process-1acquire

Process-2acquire

Process-1release

Process-3acquire

Process-2release

Process-4acquire

Process-3release

Process-5acquire

Process-4release

Process-5release

End

上面的实例中使用semaphore限制了最多有2个进程同时执行。

### 6.4 Event

event用来实现进程间同步通讯

import multiprocessing

import time

def **wait\_for\_event**(e):

print(*"wait\_for\_event: starting"*)

e.wait()

print(*"wairt\_for\_event: e.is\_set()->"* + str(e.is\_set()))

def **wait\_for\_event\_timeout**(e, t):

print(*"wait\_for\_event\_timeout:starting"*)

e.wait(t)

print(*"wait\_for\_event\_timeout:e.is\_set->"* + str(e.is\_set()))

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

e = multiprocessing.Event()

w1 = multiprocessing.Process(name = *"block"*,

target = wait\_for\_event,

args = (e,))

w2 = multiprocessing.Process(name = *"non-block"*,

target = wait\_for\_event\_timeout,

args = (e, 2))

w1.start()

w2.start()

time.sleep(3)

e.set()

print(*"main: event is set"*)

执行结果：

wait\_for\_event: starting

wait\_for\_event\_timeout:starting

wait\_for\_event\_timeout:e.is\_set->False

main: event is set

wairt\_for\_event: e.is\_set()->True

### 6.5 Queue

Queue是多进程安全的队列，可以使用Queue实现多进程之间的数据传递。put方法用以插入数据到队列中，put方法还有两个可选参数：blocked和timeout。如果blocked为True（默认值），并且timeout为正值，该方法会阻塞timeout指定的时间，直到该队列有剩余的空间。如果超时，会抛出Queue.Full异常。如果blocked为False，但该Queue已满，会立即抛出Queue.Full异常。

get方法可以从队列读取并且删除一个元素。同样，get方法有两个可选参数：blocked和timeout。如果blocked为True（默认值），并且timeout为正值，那么在等待时间内没有取到任何元素，会抛出Queue.Empty异常。如果blocked为False，有两种情况存在，如果Queue有一个值可用，则立即返回该值，否则，如果队列为空，则立即抛出Queue.Empty异常。Queue的一段示例代码：

#!/usr/bin/env python2

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

import multiprocessing

def **write\_proc**(q, i):

try:

q.put(i, block = False)

print *"Put Queue Size:%d"* % q.qsize()

except Exception, e:

print *'ffffffffffffff'*

pass

def **read\_proc**(q):

try:

print q.get(block = False)

print *"Get Queue Size:%d"* % q.qsize()

except Exception, e:

print e

pass

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

q = multiprocessing.Queue(5)

writers = [multiprocessing.Process(target=write\_proc, args=(q,i)) for i in range(6)]

readers = [multiprocessing.Process(target=read\_proc, args=(q,)) for i in range(6)]

for writer in writers:

writer.start()

for reader in readers:

reader.start()

for writer in writers:

writer.join()

for reader in readers:

reader.join()

print *'end'*

*执行结果：*

Put Queue Size:1

Put Queue Size:2

Put Queue Size:4

Put Queue Size:5

Put Queue Size:5

ffffffffffffff

0

Get Queue Size:4

1

Get Queue Size:3

2

Get Queue Size:2

3

Get Queue Size:1

4

Get Queue Size:0

end

### 6.6 Pipe

Pipe方法返回(conn1, conn2)代表一个管道的两个端。Pipe方法有duplex参数，如果duplex参数为True(默认值)，那么这个管道是全双工模式，也就是说conn1和conn2均可收发。duplex为False，conn1只负责接受消息，conn2只负责发送消息。

send和recv方法分别是发送和接受消息的方法。例如，在全双工模式下，可以调用conn1.send发送消息，conn1.recv接收消息。如果没有消息可接收，recv方法会一直阻塞。如果管道已经被关闭，那么recv方法会抛出EOFError。

#!/usr/bin/env python2

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

import multiprocessing

import time

def **proc1**(pipe):

while True:

for i in xrange(10):

print *"send: %s"* %(i)

pipe.send(i)

time.sleep(1)

break

def **proc2**(pipe):

while True:

print *"proc2 rev:"*, pipe.recv()

time.sleep(1)

def **proc3**(pipe):

while True:

print *"PROC3 rev:"*, pipe.recv()

time.sleep(1)

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

pipe = multiprocessing.Pipe()

p1 = multiprocessing.Process(target=proc1, args=(pipe[0],))

p2 = multiprocessing.Process(target=proc2, args=(pipe[1],))

p3 = multiprocessing.Process(target=proc3, args=(pipe[1],))

p1.start()

p2.start()

p3.start()

p1.join()

p2.join()

p3.join()

执行结果：

send: 0

proc2 rev: 0

PROC3 rev:send: 1

1

proc2 rev:PROC3 rev:send: 2

2

send: 3

3

proc2 rev:send: 4PROC3 rev:

4

proc2 rev:send: 5

5

PROC3 rev:send: 6

6

proc2 rev:send: 7

7

send: 8PROC3 rev:

8

send: 9proc2 rev:

9

PROC3 rev:

### 6.7 Pool

在利用Python进行系统管理的时候，特别是同时操作多个文件目录，或者远程控制多台主机，并行操作可以节约大量的时间。当被操作对象数目不大时，可以直接利用multiprocessing中的Process动态成生多个进程，十几个还好，但如果是上百个，上千个目标，手动的去限制进程数量却又太过繁琐，此时可以发挥进程池的功效。  
Pool可以提供指定数量的进程，供用户调用，当有新的请求提交到pool中时，如果池还没有满，那么就会创建一个新的进程用来执行该请求；但如果池中的进程数已经达到规定最大值，那么该请求就会等待，直到池中有进程结束，才会创建新的进程来它。

#!/usr/bin/env python2

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

import multiprocessing

import time

def **func**(msg):

print *"msg:"*, msg

time.sleep(3)

print *"end:"*, msg

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

pool = multiprocessing.Pool(processes = 3)

for i in xrange(4):

msg = *"hello %d"* %(i)

pool.apply\_async(func, (msg, )) #维持执行的进程总数为processes，当一个进程执行完毕后会添加新的进程进去

print *"Mark~ Mark~ Mark~~~~~~~~~~~~~~~~~~~~~~"*

pool.close()

pool.join() #调用join之前，先调用close函数，否则会出错。执行完close后不会有新的进程加入到pool,join函数等待所有子进程结束

print *"Sub-process(es) done."*

*执行结果：*

Mark~ Mark~ Mark~~~~~~~~~~~~~~~~~~~~~~

msg: hello 0

msg: hello 1

msg: hello 2

end: hello 0

msg: hello 3

end: hello 1

end: hello 2

end: hello 3

Sub-process(es) done.

## bisect模块

NAME

bisect - Bisection algorithms.

FILE

c:\python27\lib\bisect.py

FUNCTIONS

bisect(...)

bisect(a, x[, lo[, hi]]) -> index

bisect\_right(a, x[, lo[, hi]]) -> index

Return the index where to insert item x in list a, assuming a is sorted.

**返回x插入列表a的位置，a是排序的。lo 的默认值是 0，hi**

**的默认值是序列的长度，即 len() 作用于该序列的返回值。**

The return value i is such that all e in a[:i] have e <= x, and all e in

a[i:] have e > x. So if x already appears in the list, i points just

beyond the rightmost x already there

Optional args lo (default 0) and hi (default len(a)) bound the

slice of a to be searched.

>>> L

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

>>> bisect.bisect(L, 100)

10 # 插入位置索引

>>> bisect.bisect(L, 1)

1. # 如果插入值相等，插入相等值右侧

>>> bisect.bisect\_left(L, 1)

1 # 如果插入值相等，插入相等值左侧

>>> bisect.bisect(L, -1)

0

bisect\_left(...)

bisect\_left(a, x[, lo[, hi]]) -> index

Return the index where to insert item x in list a, assuming a is sorted.

The return value i is such that all e in a[:i] have e < x, and all e in

a[i:] have e >= x. So if x already appears in the list, i points just

before the leftmost x already there.

Optional args lo (default 0) and hi (default len(a)) bound the

slice of a to be searched.

bisect\_right(...)

bisect(a, x[, lo[, hi]]) -> index

bisect\_right(a, x[, lo[, hi]]) -> index

Return the index where to insert item x in list a, assuming a is sorted.

The return value i is such that all e in a[:i] have e <= x, and all e in

a[i:] have e > x. So if x already appears in the list, i points just

beyond the rightmost x already there

Optional args lo (default 0) and hi (default len(a)) bound the

slice of a to be searched.

insort(...)

insort(a, x[, lo[, hi]])

insort\_right(a, x[, lo[, hi]])

Insert item x in list a, and keep it sorted assuming a is sorted.

**将x插入列表a，并保持a排序的。**

If x is already in a, insert it to the right of the rightmost x.

Optional args lo (default 0) and hi (default len(a)) bound the

slice of a to be searched.

>>> bisect.insort(L, 1)

>>> L

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

insort\_left(...)

insort\_left(a, x[, lo[, hi]])

Insert item x in list a, and keep it sorted assuming a is sorted.

If x is already in a, insert it to the left of the leftmost x.

Optional args lo (default 0) and hi (default len(a)) bound the

slice of a to be searched.

insort\_right(...)

insort(a, x[, lo[, hi]])

insort\_right(a, x[, lo[, hi]])

Insert item x in list a, and keep it sorted assuming a is sorted.

If x is already in a, insert it to the right of the rightmost x.

Optional args lo (default 0) and hi (default len(a)) bound the

slice of a to be searched.

## collections模块

### namedtuple函数

namedtuple(typename, field\_names, verbose=False, rename=False)

Returns a new subclass of tuple with named fields.

namedtuple是继承自tuple的子类。namedtuple创建一个和tuple类似的对象，而且对象拥有可访问的属性。

>>> Card = collections.namedtuple('Card', ['rank', 'suit'])

>>> Card

<class '\_\_main\_\_.Card'>

>>> dir(Card)

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_getslice\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_slots\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_asdict', '\_fields', '\_make', '\_replace', 'count', 'index', **'rank', 'suit**']

>>> Card.\_\_doc\_\_

'Card(rank, suit)'

>>> Card.\_\_bases\_\_

(<type 'tuple'>,)

class **Poker**(object):

ranks = [str(n) for n in range(2, 11) + list(*'JQKA'*)]

suits = *'spades diamonds clubs hearts'*.split()

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

*self*.\_cards = [Card(rank, suit) for suit in *self*.suits

for rank in *self*.ranks]

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

return len(*self*.\_cards)

def **\_\_getitem\_\_**(*self*, position):

return *self*.\_cards[position]

另一种创建类的方法：

>>> City = collections.namedtuple('City', 'name country population coordinates')

>>> City

<class '\_\_main\_\_.City'>

>>> City.\_fields

('name', 'country', 'population', 'coordinates')