**学习环境：Python 3.7**

# string常用操作

## 删除字符串中多个指定字符

方法一：循环遍历字符，调用str.replace进行替换

chars = ",;:.! \t"  
while True:  
 s1 = input('Enter a string: ')  
 s1 = s1.strip()  
 if s1.lower() == 'quit':  
 print('QUIT!')  
 break  
 for c in chars:  
 s1 = s1.replace(c, '')  
 s2 = s1[::-1]  
 if s1 == s2:  
 print("Yes, it is a palindrome")  
 else:  
 print("No, it is not a palindrome")

方法二：调用re.sub方法进行正则表达式替换

import re  
  
chars = ",;:.! \t"  
while True:  
 s1 = input('Enter a string: ')  
 s1 = s1.strip()  
 if s1.lower() == 'quit':  
 print('QUIT!')  
 break  
 s2 = re.sub(chars, '', s1)  
 if s1 == s2:  
 print("Yes, it is a palindrome")  
 else:  
 print("No, it is not a palindrome")

方法三：调用str.makestrans设置转标换，str.translate进行字符转换

chars = ",;:.! \t"  
chars\_map = dict()  
for c in chars:  
 chars\_map[c] = None  
print(chars\_map)  
trans\_table = str.maketrans(chars\_map)  
  
while True:  
 s1 = input('Enter a string: ')  
 s1 = s1.strip()  
 if s1.lower() == 'quit':  
 print('QUIT!')  
 break  
 s2 = s1.translate(trans\_table)  
 if s1 == s2:  
 print("Yes, it is a palindrome")  
 else:  
 print("No, it is not a palindrome")

# class常用技巧

## 定制类（特殊类成员函数）

常用的一些特殊成员函数：

* \_\_init\_\_, \_\_del\_\_
* \_\_str\_\_, \_\_len\_\_
* \_\_lt\_\_, \_\_gt\_\_, \_\_eq\_\_
* \_\_getitem\_\_, \_\_setitem\_\_, \_\_delitem\_\_
* \_\_setattr\_\_, \_\_getattr\_\_
* … …

class Person:  
  
 def \_\_init\_\_(self, name, age):  
 self.info = dict()  
 self.info['name'] = name  
 self.info['age'] = age  
  
 # not recommended  
 def \_\_del\_\_(self):  
 pass  
  
 def \_\_len\_\_(self):  
 return len(self.info['name'])  
  
 def \_\_str\_\_(self):  
  
 info = list()  
 for (key, value) in self.info.items():  
 s = "{0}={1}".format(key, value)  
 info.append(s)  
 return ",".join(info)  
  
 def \_\_lt\_\_(self, other):  
 return True if self.info['age'] < other.info['age'] else False  
  
 def \_\_gt\_\_(self, other):  
 return True if self.info['age'] > other.info['age'] else False  
  
 def \_\_eq\_\_(self, other):  
 return True if self.info['age'] == other.info['age'] else False  
  
 def \_\_getitem\_\_(self, item):  
 return self.info[item]  
  
 def \_\_setitem\_\_(self, key, value):  
 self.info[key] = value  
  
 def \_\_delitem\_\_(self, key):  
 del self.info[key]  
  
  
p1 = Person('Alex', 10)  
print(len(p1))  
print(p1['name'])  
p1['sex'] = 'male'; print(p1)  
del p1['sex']; print(p1)  
  
p2 = Person('Daniel', 20)  
print(len(p2))  
print(p2['name'])  
p2['sex'] = 'female'; print(p2)  
del p2['sex']; print(p2)  
  
print(p1 < p2)  
print(p1 > p2)  
print(p1 == p2)

## 类成员的get和set优雅实现

通过@property和@xxx.setter装饰器，优雅实现get和set功能。

class Student:  
  
 def \_\_init\_\_(self, name, age, score):  
 self.name = name  
 self.\_\_age = age  
 self.\_\_score = score  
  
 @property  
 def age(self):  
 return self.\_\_age  
  
 @property  
 def score(self):  
 return self.\_\_score  
  
 @score.setter  
 def score(self, value):  
 if not isinstance(value, int):  
 raise ValueError('score must be an integer!')  
 if value < 0 or value > 100:  
 raise ValueError('score must between 0 ~ 100!')  
 self.\_\_score = value  
  
  
s = Student('Alex', 18, 80)  
print(s.name)  
print(s.age)  
print(s.score)  
s.name = 'Danniel'  
s.age = 20 # AttributeError: can't set attribute  
s.score = 100 # OK, set s.\_\_score = 100

通过@property修改的成员，一般添加下划线或双下划线，否则会陷入递归崩溃中。

class Student:  
  
 def \_\_init\_\_(self, name, age, score):  
 self.name = name  
 self.age = age  
 self.score = score  
  
 @property  
 def score(self):  
 return self.score  
  
 @score.setter  
 def score(self, value):  
 if not isinstance(value, int):  
 raise ValueError('score must be an integer!')  
 if value < 0 or value > 100:  
 raise ValueError('score must between 0 ~ 100!')  
 self.score = value  
  
  
s = Student('Alex', 18, 80)  
s.score = 100

# 执行报错

RecursionError: maximum recursion depth exceeded while calling a Python object

# 专题讨论

## 多进程

多进程相关重点内容：

* 子进程、进程池
* 进程间通信
* 进程间共享数据
* 进程间互斥访问（互斥锁、读写锁、信号量）

### multiprocessing.Process

multiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, *\**, *daemon=None*)

--- 主要用于启动一个新进程，将代码内部的一个函数过程作为进程的启动入口。

import os  
import multiprocessing as mp  
  
  
def run\_proc(name, sex, age):  
 *""" Run a child process """* print('Run child process, pid={0}, name={1} sex={2} age={3}'.format(os.getpid(), name, sex, age))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print('Run parent process, pid={0}.'.format(os.getpid()))  
 p = mp.Process(target=run\_proc, args=('test',), kwargs={'sex': 'male', 'age': 18})  
 print('Child process will start.')  
 p.start()  
 p.join()  
 print('Child process end.')

### 进程池 - multiprocessing.Pool

**multiprocessing.pool.Pool**([processes[, initializer[, initargs[, maxtasksperchild[, context]]]]])

--- 通过multiprocessing.Pool()创建一个multiprocessing.pool对象

**apply\_async**(*func*[, *args*[, *kwds*[, *callback*[, *error\_callback*]]]])

**map**(func, iterable[, chunksize])

--- A parallel equivalent of the [map()](https://docs.python.org/3/library/functions.html#map) built-in function (it supports only one iterable argument though). It blocks until the result is ready.

--- 一直阻塞，直到map返回结果。

**map\_async**(func, iterable[, chunksize[, callback[, error\_callback]]])

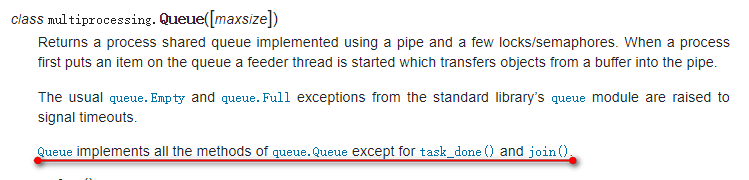
**--- 问题：在测试中，callback为达到预期效果，原因不明？**

import os  
import time  
import random  
import multiprocessing as mp  
  
  
def long\_time\_task(name):  
 *""" Run a child process, running a long time """* start = time.time()  
 time.sleep(random.random() \* 5)  
 end = time.time()  
 elapsed = end - start  
 return elapsed  
  
  
def apply\_callback(t):  
 print('Elapsed time = {0}'.format(t))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 print('Run parent process, pid={0}.'.format(os.getpid()))  
 # 默认的进程池个数为CPU个数（os.cpu\_count()）  
 p = mp.Pool()  
 # 如果创建的进程数过多，需要等待  
 for i in range(8):  
 p.apply\_async(long\_time\_task, args=(i,), callback=apply\_callback)  
 print('Waiting for all subprocesses done...')  
 # Prevents any more tasks from being submitted to the pool.  
 p.close()  
 p.join()  
 print('All sub processes done.')

问题： Pool.map\_async的callbaack参数传递，似乎没起到预期效果？

import multiprocessing as mp  
  
  
def map\_func(x):  
 return x \* 2  
  
  
def map\_callback(x):  
 print('x = {0}'.format(x))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 p = mp.Pool()  
 # 问题：map\_callback没起到预期的效果？  
 h = p.map\_async(map\_func, [1, 2, 3], callback=map\_callback)

### 进程间通信 - Queue



--- 基础用法参考queue.Queue。

import os  
import time  
import random  
import multiprocessing as mp  
import queue  
  
  
  
def write\_queue\_proc(q):  
 print('Child Process: start to write, pid={0}'.format(os.getpid()))  
 for v in ['A', 'B', 'C', ['Python', 'Java']]:  
 print('Put {0} to queue.'.format(v))  
 q.put(v)  
 time.sleep(random.random())  
 print('Child Process: stop to write, pid={0}'.format(os.getpid()))  
  
  
def read\_queue\_proc(q):  
 print('Child Process: start to read, pid={0}'.format(os.getpid()))  
 try:  
 while True:  
 v = q.get(True, 3)  
 print('Get {0} from queue'.format(v))  
 except queue.Empty:  
 print('Empty Queue!')  
 finally:  
 print('Child Process: stop to read, pid={0}'.format(os.getpid()))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 q = mp.Queue()  
 pw = mp.Process(target=write\_queue\_proc, args=(q,))  
 pr = mp.Process(target=read\_queue\_proc, args=(q,))  
 pw.start()  
 pr.start()  
 pw.join()  
 pr.join()

### 进程间通信 - Pipe

import os  
import time  
import random  
import multiprocessing as mp  
  
  
def write\_pipe\_proc(in\_p, out\_p):  
 print('Child Process: start to write, pid={0}'.format(os.getpid()))  
 out\_p.close()  
 for v in ['A', 'B', 'C', ['Python', 'Java']]:  
 print('Put {0} to pipe.'.format(v))  
 in\_p.send(v)  
 time.sleep(random.random())  
 in\_p.close()  
 print('Child Process: stop to write, pid={0}'.format(os.getpid()))  
  
  
def read\_pipe\_proc(in\_p, out\_p):  
 print('Child Process: start to read, pid={0}'.format(os.getpid()))  
 in\_p.close()  
 try:  
 while True:  
 v = out\_p.recv()  
 print('Get {0} from queue'.format(v))  
 except EOFError:  
 print('Empty pipe!')  
 finally:  
 out\_p.close()  
 print('Child Process: stop to read, pid={0}'.format(os.getpid()))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 in\_p, out\_p = mp.Pipe()  
 pw = mp.Process(target=write\_pipe\_proc, args=(in\_p, out\_p))  
 pr = mp.Process(target=read\_pipe\_proc, args=(in\_p, out\_p))  
 pw.start()  
 pr.start()  
 in\_p.close() # 如果in\_p不进行close操作，read\_pipe\_proc进程的output\_p.recv会阻塞读取  
 out\_p.close()  
 pw.join()  
 pr.join()

### 进程间通信 - Event事件



--- 具体用法参考threading.Event

### 进程间共享数据 - Shared Memory

multiprocessing.**Value**(typecode\_or\_type, \*args, lock=True)

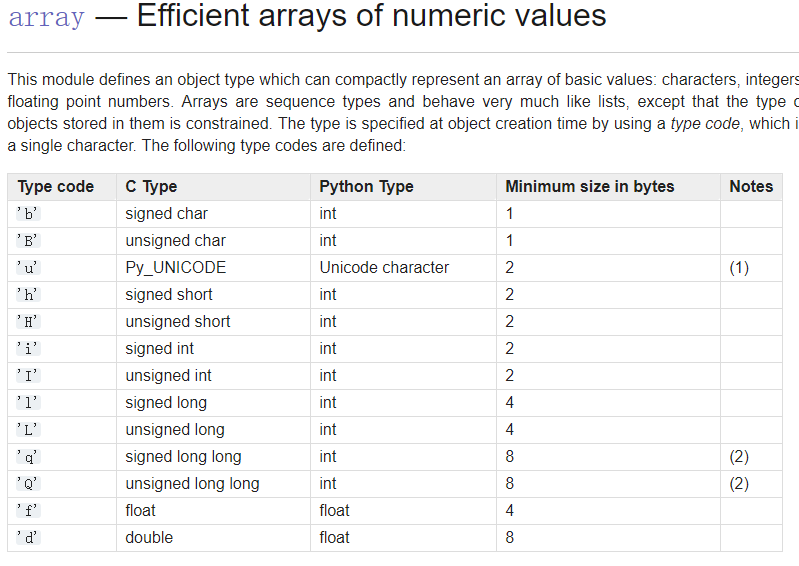
--- Return a [ctypes](https://docs.python.org/3/library/ctypes.html#module-ctypes) object allocated from shared memory.

multiprocessing.**Array**(typecode\_or\_type, size\_or\_initializer, \*, lock=True)

--- Return a ctypes array allocated from shared memory.

import multiprocessing as mp  
  
  
def process\_proc\_a(v, a):  
 v.value = 999  
  
 with v.get\_lock():  
 for i in range(len(a)):  
 a[i] += 1  
  
  
def process\_proc\_b(v, a):  
 with v.get\_lock():  
 for i in range(len(a)):  
 a[i] += 1  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 v = mp.Value('d', 100) # 'd' -> double  
 a = mp.Array('i', range(10)) # 'i' -> signed int  
  
 p1 = mp.Process(target=process\_proc\_a, args=(v, a))  
 p2 = mp.Process(target=process\_proc\_a, args=(v, a))  
 p1.start()  
 p2.start()  
 p1.join()  
 p2.join()  
 print(v.value)  
 print(a[:])

备注：typecode编码含义：



### 进程间共享数据 - Server process

A manager returned by Manager() will support types list, dict, Namespace, Lock, RLock, Semaphore, BoundedSemaphore, Condition, Event, Barrier, Queue, Value and Array.

--- 与共享内存（Value、Array）相比，Manager支持更多的数据类型。

import multiprocessing as mp  
  
  
def process\_proc\_a(d, l, v, q):  
 d['a'] = 'ABC'  
 l.reverse()  
 v.value = 999  
 q.put('Hello, world')  
  
  
def process\_proc\_b(a, lo):  
 lo.acquire()  
 for i in range(len(a)):  
 a[i] += 1  
 lo.release()  
  
  
def process\_proc\_c(a, lo):  
 lo.acquire()  
 for i in range(len(a)):  
 a[i] += 1  
 lo.release()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 with mp.Manager() as manager:  
 d = manager.dict()  
 l = manager.list(range(10))  
 v = manager.Value('d', 100) # 'd' -> double  
 q = manager.Queue()  
  
 p1 = mp.Process(target=process\_proc\_a, args=(d, l, v, q))  
 p1.start()  
 p1.join()  
 print(d)  
 print(l)  
 print(v.value)  
 print(q.get())  
  
 a = manager.Array('i', range(10)) # 'i' -> signed int  
 lo = manager.Lock()  
 p2 = mp.Process(target=process\_proc\_b, args=(a, lo))  
 p3 = mp.Process(target=process\_proc\_c, args=(a, lo))  
 p2.start()  
 p3.start()  
 p2.join()  
 p3.join()  
 print(a[:])

Server process managers are more flexible than using shared memory objects because they can be made to support arbitrary object types. Also, a single manager can be shared by processes on different computers over a network. They are, however, slower than using shared memory.

--- 与shared memory相比，性能较低。

### subprocess

很多时候，子进程入口不是当前代码的一个内部函，而是一个外部程序。此时通过subprocess指定外部程序创建子进程，并可以控制子进程的输入和输出。

**创建子进程：subprocess.Popen**(…)：

**参数shell详解：**

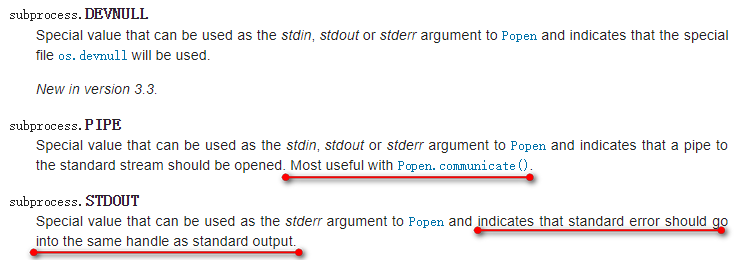
The *shell* argument (which defaults to False) specifies whether to use the shell as the program to execute. If *shell* is True, it is recommended to pass *args* as a string rather than as a sequence.

--- shell参数表示是否通过shell命令执行程序。如果shell=True，建议args参数以字符串形式传递，而不是序列。

**参数stdin、stdout、stdin详解：**

*stdin*, *stdout* and *stderr* specify the executed program’s standard input, standard output and standard error file handles, respectively. Valid values are [PIPE](https://docs.python.org/3/library/subprocess.html#subprocess.PIPE), [DEVNULL](https://docs.python.org/3/library/subprocess.html#subprocess.DEVNULL), an existing file descriptor (a positive integer), an existing [file object](https://docs.python.org/3/glossary.html#term-file-object), and None.

--- 标准输入、输出、错误输出流。合法参数值为PIPE，DEVNULL，文件描述符，文件对象和None。



**与子进程进行IO交互：Popen.communicate**(*input=None*, *timeout=None*)

--- 用于与子程序进程输入输出交互，返回(stdout\_data, stderr\_data)

import subprocess  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 p = subprocess.Popen(['ls', '-l', '/dev/null'])  
 print(p.stdout)  
  
 p = subprocess.Popen(['ls', '-l', '/dev/null'], stdout=subprocess.PIPE)  
 for l in p.stdout.readlines():  
 print(l.decode('utf-8'))  
  
 p = subprocess.Popen(['nslookup'], stdin=subprocess.PIPE, stdout=subprocess.PIPE)  
 output, err = p.communicate(b'www.baidu.com\n')  
 print(output.decode('utf-8'))  
  
 fo = open('./output1.txt', 'w')  
 p = subprocess.Popen('ls -l /dev/null', shell=True, stdin=subprocess.PIPE, stdout=fo)  
 print(p.stdout) # here, p.stdout is None  
 fo.close()  
  
 fi = open('./input.txt', 'r')  
 fo = open('./output2.txt', 'w')  
 p = subprocess.Popen('cat', shell=True, stdin=fi, stdout=fo)  
 print(p.stdout) # here, p.stdout is None  
 fi.close()  
 fo.close()

### 多进程同步 - Lock



--- 具体用法参考threading.Lock。

### 多进程同步 - Condition



--- 具体用法参考threading.Condition。

## 多线程

threading module constructs higher-level threading interfaces on top of the lower level \_thread module.

--- threading模块：更高层次的多线程接口

--- \_thread模块：较低层次的线程接口

### 线程的start和run方法区别

参考：<http://www.cnblogs.com/i-honey/p/8043648.html>

* start() 方法是启动一个子线程，线程名就是我们定义的name；  
  调用流程：start() --> run() --> \_target()
* run() 方法并不启动一个新线程，就是在主线程中调用了一个普通函数而已；   
  调用流程：run() --> \_target()  
  通常，当继承threading.Thread类时，会重载run()方法。

其中：\_target()就是我们传入给线程的入口函数。

import time  
import threading  
  
  
class MyThread(threading.Thread):  
  
 def \_\_init\_\_(self, n):  
 threading.Thread.\_\_init\_\_(self, name=n)  
  
 def run(self):  
 t = threading.current\_thread()  
 print('ChildThread: name={0}, id={1}, daemon={2}'.format(t.name, t.ident, t.daemon))  
 time.sleep(2)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 t = MyThread('NewThread')  
 # 调用run()，类似调用一个普通接口，不会创建新的线程  
 t.run()  
 # 调用start()，会创建一个新的线程，并自动调用run()  
 t.start()  
 t.join()

### 线程的daemon属性

在脚本运行过程中有一个主线程，若在主线程中创建了子线程，当主线程结束时根据子线程daemon属性值的不同可能会发生下面的两种情况之一：

* 如果某个子线程的daemon属性为False，主线程结束时会检测该子线程是否结束，如果该子线程还在运行，则主线程会等待它完成后再退出；
* 如果某个子线程的daemon属性为True，主线程运行结束时不对这个子线程进行检查而直接退出，同时所有daemon值为True的子线程将随主线程一起结束，而不论是否运行完成。

import time  
import threading  
  
  
def thread\_proc(n):  
 t = threading.current\_thread()  
 print('ChildThread: name={0}, id={1}, daemon={2}'.format(t.name, t.ident, t.daemon))  
 for i in range(n):  
 time.sleep(1)  
 print('ChildThread: name={0} is running'.format(t.name))  
 print('ChildThread: name={0} exits ...'.format(t.name))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 # t1线程运行较短时间  
 t1 = threading.Thread(target=thread\_proc, name='thread-1', args=(2, ))  
 # t2线程运行较长时间  
 t2 = threading.Thread(target=thread\_proc, name='thread-2', args=(100,))  
 t1.daemon = False  
 t2.daemon = False  
 t1.start()  
 t2.start()  
 # t1.join()  
 # t2.join()

# 运行结果：t1运行结束后，由于t2.daemon=False，主线程会一直等待t2运行结束

ChildThread: name=thread-1, id=3096, daemon=False

ChildThread: name=thread-2, id=15704, daemon=False

ChildThread: name=thread-2 is running

ChildThread: name=thread-1 is running

ChildThread: name=thread-1 is running

ChildThread: name=thread-1 exits ...

ChildThread: name=thread-2 is running

ChildThread: name=thread-2 is running

ChildThread: name=thread-2 is running

ChildThread: name=thread-2 is running

ChildThread: name=thread-2 is running

ChildThread: name=thread-2 is running

… …

在上例中，如果t2.daemon=True，运行效果如下：

# 运行效果：t1运行结束后，由于t2.daemon=True，主线程不会等待t2结束自行结束，此时t2也会随着主线程一起结束

ChildThread: name=thread-1, id=5652, daemon=False

ChildThread: name=thread-2, id=11092, daemon=True

ChildThread: name=thread-1 is running

ChildThread: name=thread-2 is running

ChildThread: name=thread-1 is running

ChildThread: name=thread-1 exits ...

Process finished with exit code 0

### Thread Local数据

Thread Local数据主要解决两个场景的问题：

* 多个线程访问全局数据不太安全，需要精确控制（加锁、分区访问），容易出错；
* 多个线程可以访问局部变量，但需要在函数之间不断传递，比较麻烦；

import threading  
  
# 全局变量，每个thread根据thread-id分区，访问与当前线程关联的数据  
global\_dict = {}  
  
# 一个Thread Local变量虽然是全局变量，但每个线程都只能读写自己线程的独立副本，互不干扰。  
local = threading.local()  
  
  
def func():  
 print('Thread name is {0}'.format(local.name))  
 print('Thread name is {0}'.format(global\_dict[threading.current\_thread().ident]['name']))  
  
  
def thread\_proc():  
 # 方法1：通过全局变量，访问线程相关数据（）  
 global\_dict[threading.current\_thread().ident] = {}  
 global\_dict[threading.current\_thread().ident]['name'] = threading.current\_thread().name  
  
 # 方法2：通过Thread Local变量，访问线程私有数据（无法访问其他线程数据）  
 local.name = threading.current\_thread().name  
  
 func()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 t1 = threading.Thread(target=thread\_proc, name='thread-1')  
 t2 = threading.Thread(target=thread\_proc, name='thread-2')  
 t1.start()  
 t2.start()  
 t1.join()  
 t2.join()

### 多线程通信 - queue.Queue

The [queue](https://docs.python.org/3/library/queue.html#module-queue) module implements multi-producer, multi-consumer queues.

--- 多用于生产/消费模型。

class **queue.Queue**(maxsize=0)

--- FIFO queue

class **queue.LifoQueue**(maxsize=0)

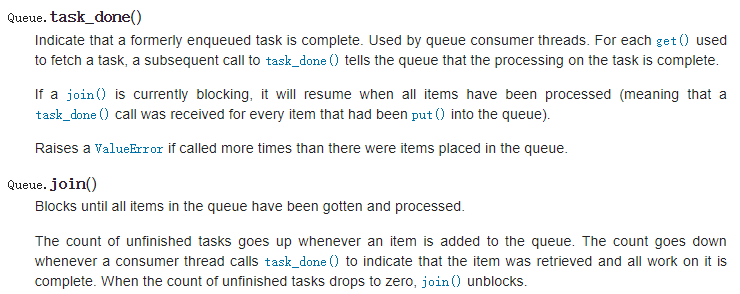
--- LIFO queue

class **queue.PriorityQueue**(maxsize=0)

--- Priority queue. With a priority queue, the entries are kept sorted (**using the heapq module**) and the lowest valued entry is retrieved first.

class queue.SimpleQueue()

--- An unbounded FIFO queue. Simple queues lack advanced functionality such as task tracking.



--- task\_done()：指示之前的enqueued task（出队列的任务）已经完成，对每个get()获取的任务被完成后调用task\_done进行指示。

--- join()：阻塞，知道队列中所有任务被获取和处理。（感觉内部有一个计数器标记“未完成的任务数”，此标记通过task\_done进行修改）

### 多线程通信 - Event事件

This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

--- 最简单的通信机制之一：一个线程发出event信号，其他线程等待此event信号。

--- 可用于有条件的生产/消费模型。

假设场景：创建两个子线程连接DB，需要等MainThread完成DB初始化后进行事件通知。

如果没有Event功能，子进程可能需要进入一个while-check-sleep的循环。

import time  
import threading  
  
  
def thread\_proc(e):  
 if not e.is\_set():  
 print('Thread {0} waits to connect DB ...'.format(threading.current\_thread().name))  
 e.wait()  
 # do something  
 print('Thread {0} connects to DB successfully'.format(threading.current\_thread().name))  
  
  
e = threading.Event()  
t1 = threading.Thread(target=thread\_proc, name='Thread-1', args=(e,))  
t2 = threading.Thread(target=thread\_proc, name='Thread-2', args=(e,))  
t1.start()  
t2.start()  
time.sleep(2)  
e.set() # DB is ready  
t1.join()  
t2.join()

### 定时器线程Timer

threading.Timer继承threading.Thread，其属性和行为与Thread基本保持一致。

import time  
import threading  
  
  
def timer\_func(msg):  
 print('Timer thread name={0}, msg={1}'.format(threading.current\_thread().name, msg))  
  
  
# class Timer is a subclass of Thread.  
t = threading.Timer(2.0, timer\_func, args=('Hello', ))  
t.start()  
# time.sleep(2)  
# t.cancel()  
t.join()

### 多线程同步 - Lock

class threading.Lock接口说明：

* acquire(blocking=True, timeout=-1)  
  --- The return value is True if the lock is acquired successfully, False if not (for example if the timeout expired).
* release()  
  --- Release a lock. This can be called from any thread, not only the thread which has acquired the lock.  
  --- When invoked on an unlocked lock, a [RuntimeError](https://docs.python.org/3/library/exceptions.html#RuntimeError) is raised.

# 返回值说明：  
# True：获取lock成功；否则一直阻塞，不返回  
l.acquire(blocking=True, timeout=-1)  
  
# 返回值说明：  
# True：获取lock成功  
# False：获取lock失败  
l.acquire(blocking=True, timeout=-10)  
  
# 返回值说明：  
# True：获取Lock成功  
# False：获取Lock失败（如果设置blocking=True，会阻塞）  
l.acquire(blocking=False)

### 多线程同步 - Condition

**以下内容为网络摘抄：**

当小伙伴a在往火锅里面添加鱼丸，这个就是生产者行为；另外一个小伙伴b在吃掉鱼丸就是消费者行为。当火锅里面鱼丸达到一定数量加满后b才能吃，这就是一种**条件判断**了。

这就是本篇要讲的Condition（条件变量）

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。

Condition():

* acquire(): 线程锁
* release(): 释放锁
* wait(timeout): 线程挂起，直到收到一个notify通知或者超时（可选的，浮点数，单位是秒s）才会被唤醒继续运行。wait()必须在已获得Lock前提下才能调用，否则会触发RuntimeError。
* notify(n=1): 通知其他线程，那些挂起的线程接到这个通知之后会开始运行，默认是通知一个正等待该condition的线程,最多则唤醒n个等待的线程。notify()必须在已获得Lock前提下才能调用，否则会触发RuntimeError。notify()不会主动释放Lock。
* notifyAll(): 如果wait状态线程比较多，notifyAll的作用就是通知所有线程

场景：当Producer增加num到一定数量后（条件num=5），通知Consumer进行消费；当消费完毕后，Consu通知Producer继续工作；

import time  
import threading  
  
  
num = 0  
  
  
class Producer(threading.Thread):  
  
 def \_\_init\_\_(self, con):  
 threading.Thread.\_\_init\_\_(self)  
 self.con = con  
  
 def run(self):  
  
 global num  
 self.con.acquire() # protect global num  
 while True:  
 print('Producer: start to add num ...')  
 num += 1  
 print('Producer: num = {0}'.format(num))  
 time.sleep(1)  
 if num >= 5:  
 print('Producer: num={0}, stop to add num'.format(num))  
 con.notify() # wake up consumer thread  
 con.wait() # # wait for consumer to wakeup  
 # never reached  
 self.con.release()  
  
  
class Consumers(threading.Thread):  
 def \_\_init\_\_(self, con):  
 threading.Thread.\_\_init\_\_(self)  
 self.con = con  
  
 def run(self):  
  
 global num  
 con.acquire() # protect global num  
 while True:  
 print('Consumer: start to minus num ...')  
 num -= 1  
 print('Consumer: num = {0}'.format(num))  
 time.sleep(1)  
 if num <= 0:  
 print('Consumer: num = {0}, stop to minus num'.format(num))  
 con.notify() # wakeup producer thread  
 con.wait() # wait for producer to wakeup  
 # never reached  
 con.release()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 con = threading.Condition()  
 p = Producer(con)  
 c = Consumers(con)  
 p.start()  
 c.start()

### （待深入）协程Coroutine

一个coroutine从结构上像是一个generator，只是一个包含yield关键字的函数而已。yield是一个控制流程的设备，使多任务间协作（yield本意就是“放弃”）。每个coroutine yields 控制返还给scheduler，因此其他的coroutine被激活。

coroutine可以处于4种状态，通过inspect.getgeneratorstate()函数来确定状态。

* GEN\_CREATED，等待开始
* GEN\_RUNNING，被解释器执行
* GEN\_SUSPENED，在yield表达式处挂起
* GEN\_CLOSED，执行结束

**举例：协程实现producer-consumer模型**

整个流程无锁，由一个线程执行，produce和consumer协作完成任务，所以称为“协程”，而非线程的抢占式多任务。

import inspect  
  
  
def consumer():  
 print('consumer: created ...')  
 msg = ''  
 while True:  
 data = yield msg  
 print('consumer: consume data = {0}'.format(data))  
 msg = 'OK'  
  
  
def producer(con):  
 con.send(None) # activate consumer coroutine  
 for i in range(3):  
 print('producer: send data = {0} to consumer'.format(i))  
 msg = con.send(i)  
 print('producer: consumer return msg = {0}'.format(msg))  
 # producer完成生产，关闭consumer  
 con.close()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 con = consumer()  
 print(inspect.getgeneratorstate(con)) # GEN\_CREATED  
  
 producer(con)  
 print(inspect.getgeneratorstate(con)) # GEN\_CLOSED

# greenlet、gevent、eventlet

## 生成器generator

个人理解：generator和coroutine的区别：

* generator产生数据（yield），用来在迭代（iterator）中使用
* coroutine则是需要其他协程发送数据过来（send），从而消费数据并返回结果（yield）

举例：generator产生数据

import time  
  
def range2(num):  
 for i in range(num):  
 yield i  
  
  
for i in range2(3):  
 print(i)  
  
print()  
  
c = range2(3)  
print(c.\_\_next\_\_())  
print(c.\_\_next\_\_())  
print(c.\_\_next\_\_())  
time.sleep(1)  
print(c.\_\_next\_\_())

# 程序运行输出。

# for语句会自动调用和处理generator的\_\_next\_\_接口和StopIteration异常

# 通过\_\_next\_\_获取到最后时会产生StopIteration异常

0

1

2

0

1

2

Traceback (most recent call last):

File "D:/python-study/Hello/hello.py", line 146, in <module>

print(c.\_\_next\_\_())

StopIteration

generator expressions（生成器表达式）：

g = (x \* x for x in range(10)) # generator expressions  
print(g)  
for i in g:  
 print(i, end=' ')

# 将上述generator expression通过generator函数显示实现如下：

def pow2(num):  
 for x in range(num):  
 yield x \* x  
  
for i in pow2(10):  
 print(i, end=' ')

## 装饰器decorator

### 不带参数的decorator（二层嵌套def）

关键知识点：

* 通用化的参数传递(\*args, \*\*kwargs)
* 通过 @functools.wraps保存函数的\_\_name\_\_和\_\_doc\_\_属性

# -\*- coding:utf-8 -\*-  
  
import functools  
  
  
# def increment(func):  
# # wrapped\_f(a, b)是固定的参数传递方法，仅接受指定格式的参数形式  
# def wrapped\_f(a, b):  
# """ Increment a function result """  
# return func(a, b) + 1  
# return wrapped\_f  
  
  
def increment(func):  
 @functools.wraps(func)  
 # wrapped\_f(\*args, \*\*kwargs)是更通用的参数传递方法，可以任意的参数形式  
 def wrapped\_f(\*args, \*\*kwargs):  
 *""" Increment a function result """* return int(func(\*args, \*\*kwargs) + 1)  
 return wrapped\_f  
  
  
@increment  
def test\_plus(a, b):  
 *""" Add two things together """* return a + b  
  
  
# 如果设置@functools.wraps(func), 会负责把原始函数的\_\_name\_\_和\_\_doc\_\_等属性复制到wrapped\_f()函数中。  
print(test\_plus.\_\_name\_\_) # this returns 'test\_plus'  
print(test\_plus.\_\_doc\_\_) # this returns 'Add two things together'  
  
  
# 如果缺少@functools.wraps(func)  
# print(test\_plus.\_\_name\_\_) # this is now 'wrapped\_f' instead of 'test\_plus'  
# print(test\_plus.\_\_doc\_\_) # this now returns 'Increment a function result' instead of 'Add two things  
  
  
r = test\_plus(5, 5)  
assert r == 11, "We wrote our decorator wrong!"

### 携带参数的decorator（三层嵌套def）

def log(level):  
 def log\_decorator(func):  
 @functools.wraps(func)  
 def wrapped\_f(\*args, \*\*kwargs):  
 print('{0} logging info ...'.format(level))  
 return func(\*args, \*\*kwargs)  
 return wrapped\_f  
 return log\_decorator  
  
  
# 翻译成高阶函数的调用: test\_multiply = log('ERROR')(test\_multiply)  
@log('ERROR')  
def test\_multiply(a, b):  
 return a \* b  
  
  
s = test\_multiply(5, 5)  
assert s == 25

把装饰过程进行拆解后，过程如下：

log\_tmp = log('ERROR') # step1: 返回一个decorator函数  
  
  
# step2: 用log\_tmp去装饰test\_multiply并返回新函数  
@log\_tmp  
def test\_multiply(a, b):  
 return a \* b  
  
  
s = test\_multiply(5, 5)  
assert s == 25

考虑把三层嵌套的decorator定义拆分定义，此时最内层的wrapped函数引用了未定义的参数level，导致运行异常。

def log\_decorator(func):  
 @functools.wraps(func)  
 def wrapped\_f(\*args, \*\*kwargs):  
 # \*\*\* 此处引用了未定义的level，调用失败 \*\*\*  
 print('{0} logging info ...'.format(level))  
 return func(\*args, \*\*kwargs)  
 return wrapped\_f  
  
  
def log(level):  
 return log\_decorator  
  
  
@log('ERROR')  
def test\_func(a, b):  
 return a + b  
  
  
test = log('ERROR')(test\_func)  
a = test\_func(10, 10)  
assert a == 20

# log\_decorator运行异常，提示level变量未定义

Traceback (most recent call last):

File "D:/python-study/Hello/decorator.py", line 94, in <module>

a = test\_func(10, 10)

File "D:/python-study/Hello/decorator.py", line 79, in wrapped\_f

print('{0} logging info ...'.format(level))

NameError: name 'level' is not defined

## 字符串编码问题

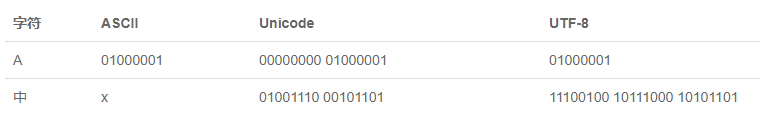
**ASCII编码：**1字节编码，只有127个英文字符。

**Unicode编码：**一个不断发展的标准，通常是2个字节编码（非常偏僻的字符需要4个字节编码）。

--- 问题：如果统一成Unicode编码，乱码问题从此消失了。但是，如果文本内容基本上全部是英文的话，用Unicode编码比ASCII编码需要多一倍的存储空间，在存储和传输上就十分不划算。

**UTF-8编码：**把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。

示例：



注意：

* Python2.X源码文件默认使用ascii编码（sys.getdefaultencoding() == ascii），默认不可以正常解析中文，源文件需要指定UTF-8编码。
* Python3.X 源码文件默认使用utf-8编码（sys.getdefaultencoding() == utf-8），所以可以正常解析中文，源文件无需指定 UTF-8 编码。

# FAQs

## ？？？浅拷贝copy和深拷贝deepcopy

## hashable和immutable的关系？

# 来自stackoverflow的高票数回答：

[**Hashing**](http://en.wikipedia.org/wiki/Hash_function) is the process of **converting some large amount of data into a much smaller amount (typically a single integer) in a repeatable way** so that it can be looked up in a table in constant-time (O(1)), which is important for high-performance algorithms and data structures.

[**Immutability**](http://en.wikipedia.org/wiki/Immutable_object) is the idea that an object **will not change in some important way after it has been created**, especially in any way that might change the hash value of that object.

The two ideas are related because **objects which are used as hash keys must typically be immutable so their hash value doesn't change.** If it was allowed to change then the location of that object in a data structure such as a hashtable would change and then the whole purpose of hashing for efficiency is defeated.