* [首页](https://www.cnblogs.com/)
* [新闻](https://news.cnblogs.com/)
* [博问](https://q.cnblogs.com/)
* [专区](https://brands.cnblogs.com/)
* [闪存](https://ing.cnblogs.com/)
* [班级](https://edu.cnblogs.com/)

窗体顶端

窗体底端

* [注册](https://account.cnblogs.com/signup)[登录](javascript:void(0);)

[返回主页](https://www.cnblogs.com/wf-linux/)

[**与子偕老i**](https://www.cnblogs.com/wf-linux/)

* [博客园](https://www.cnblogs.com/)
* [首页](https://www.cnblogs.com/wf-linux/)
* [新随笔](https://i.cnblogs.com/EditPosts.aspx?opt=1)
* [联系](https://msg.cnblogs.com/send/%E7%95%A5%E5%BE%AE%E6%8A%96%E5%8A%A8)
* [管理](https://i.cnblogs.com/)
* [订阅](javascript:void(0))[订阅](https://www.cnblogs.com/wf-linux/rss/)

随笔- 14  文章- 0  评论- 14  阅读- 11万

[**Python 常用模块大全（整理）**](https://www.cnblogs.com/wf-linux/p/9400354.html)

**OS 模块**

#os模块就是对操作系统进行操作，使用该模块必须先导入模块：

import os

#getcwd() 获取当前工作目录(当前工作目录默认都是当前文件所在的文件夹)

result = os.getcwd()

print(result)

#chdir()改变当前工作目录

os.chdir('/home/sy')

result = os.getcwd()

print(result)

open('02.txt','w')

#操作时如果书写完整的路径则不需要考虑默认工作目录的问题,按照实际书写路径操作

open('/home/sy/下载/02.txt','w')

#listdir() 获取指定文件夹中所有内容的名称列表

result = os.listdir('/home/sy')

print(result)

#mkdir() 创建文件夹

#os.mkdir('girls')

#os.mkdir('boys',0o777)

#makedirs() 递归创建文件夹

#os.makedirs('/home/sy/a/b/c/d')

#rmdir() 删除空目录

#os.rmdir('girls')

#removedirs 递归删除文件夹 必须都是空目录

#os.removedirs('/home/sy/a/b/c/d')

#rename() 文件或文件夹重命名

#os.rename('/home/sy/a','/home/sy/alibaba'

#os.rename('02.txt','002.txt')

#stat() 获取文件或者文件夹的信息

#result = os.stat('/home/sy/PycharmProject/Python3/10.27/01.py)

#print(result)

#system() 执行系统命令(危险函数)

#result = os.system('ls -al') #获取隐藏文件

#print(result)

#环境变量

'''

环境变量就是一些命令的集合

操作系统的环境变量就是操作系统在执行系统命令时搜索命令的目录的集合

'''

#getenv() 获取系统的环境变量

result = os.getenv('PATH')

print(result.split(':'))

#putenv() 将一个目录添加到环境变量中(临时增加仅对当前脚本有效)

#os.putenv('PATH','/home/sy/下载')

#os.system('syls')

#exit() 退出终端的命令

#os模块中的常用值

#curdir 表示当前文件夹 .表示当前文件夹 一般情况下可以省略

print(os.curdir)

#pardir 表示上一层文件夹 ..表示上一层文件夹 不可省略!

print(os.pardir)

#os.mkdir('../../../man')#相对路径 从当前目录开始查找

#os.mkdir('/home/sy/man1')#绝对路径 从根目录开始查找

#name 获取代表操作系统的名称字符串

print(os.name) #posix -> linux或者unix系统 nt -> window系统

#sep 获取系统路径间隔符号 window ->\ linux ->/

print(os.sep)

#extsep 获取文件名称和后缀之间的间隔符号 window & linux -> .

print(os.extsep)

#linesep 获取操作系统的换行符号 window -> \r\n linux/unix -> \n

print(repr(os.linesep))

#导入os模块

import os

#以下内容都是os.path子模块中的内容

#abspath() 将相对路径转化为绝对路径

path = './boys'#相对

result = os.path.abspath(path)

print(result)

#dirname() 获取完整路径当中的目录部分 & basename()获取完整路径当中的主体部分

path = '/home/sy/boys'

result = os.path.dirname(path)

print(result)

result = os.path.basename(path)

print(result)

#split() 将一个完整的路径切割成目录部分和主体部分

path = '/home/sy/boys'

result = os.path.split(path)

print(result)

#join() 将2个路径合并成一个

var1 = '/home/sy'

var2 = '000.py'

result = os.path.join(var1,var2)

print(result)

#splitext() 将一个路径切割成文件后缀和其他两个部分,主要用于获取文件的后缀

path = '/home/sy/000.py'

result = os.path.splitext(path)

print(result)

#getsize() 获取文件的大小

#path = '/home/sy/000.py'

#result = os.path.getsize(path)

#print(result)

#isfile() 检测是否是文件

path = '/home/sy/000.py'

result = os.path.isfile(path)

print(result)

#isdir() 检测是否是文件夹

result = os.path.isdir(path)

print(result)

#islink() 检测是否是链接

path = '/initrd.img.old'

result = os.path.islink(path)

print(result)

#getctime() 获取文件的创建时间 get create time

#getmtime() 获取文件的修改时间 get modify time

#getatime() 获取文件的访问时间 get active time

import time

filepath = '/home/sy/下载/chls'

result = os.path.getctime(filepath)

print(time.ctime(result))

result = os.path.getmtime(filepath)

print(time.ctime(result))

result = os.path.getatime(filepath)

print(time.ctime(result))

#exists() 检测某个路径是否真实存在

filepath = '/home/sy/下载/chls'

result = os.path.exists(filepath)

print(result)

#isabs() 检测一个路径是否是绝对路径

path = '/boys'

result = os.path.isabs(path)

print(result)

#samefile() 检测2个路径是否是同一个文件

path1 = '/home/sy/下载/001'

path2 = '../../../下载/001'

result = os.path.samefile(path1,path2)

print(result)

#os.environ 用于获取和设置系统环境变量的内置值

import os

#获取系统环境变量 getenv() 效果

print(os.environ['PATH'])

#设置系统环境变量 putenv()

os.environ['PATH'] += ':/home/sy/下载'

os.system('chls')

os 常用方法

os.remove(‘path/filename’) 删除文件

os.rename(oldname, newname) 重命名文件

os.walk() 生成目录树下的所有文件名

os.chdir('dirname') 改变目录

os.mkdir/makedirs('dirname')创建目录/多层目录

os.rmdir/removedirs('dirname') 删除目录/多层目录

os.listdir('dirname') 列出指定目录的文件

os.getcwd() 取得当前工作目录

os.chmod() 改变目录权限

os.path.basename(‘path/filename’) 去掉目录路径，返回文件名

os.path.dirname(‘path/filename’) 去掉文件名，返回目录路径

os.path.join(path1[,path2[,...]]) 将分离的各部分组合成一个路径名

os.path.split('path') 返回( dirname(), basename())元组

os.path.splitext() 返回 (filename, extension) 元组

os.path.getatime\ctime\mtime 分别返回最近访问、创建、修改时间

os.path.getsize() 返回文件大小

os.path.exists() 是否存在

os.path.isabs() 是否为绝对路径

os.path.isdir() 是否为目录

os.path.isfile() 是否为文件

**sys 模块**

sys.argv 命令行参数List，第一个元素是程序本身路径

sys.modules.keys() 返回所有已经导入的模块列表

sys.exc\_info() 获取当前正在处理的异常类,exc\_type、exc\_value、exc\_traceback当前处理的异常详细信息

sys.exit(n) 退出程序，正常退出时exit(0)

sys.hexversion 获取Python解释程序的版本值，16进制格式如：0x020403F0

sys.version 获取Python解释程序的版本信息

sys.maxint 最大的Int值

sys.maxunicode 最大的Unicode值

sys.modules 返回系统导入的模块字段，key是模块名，value是模块

sys.path 返回模块的搜索路径，初始化时使用PYTHONPATH环境变量的值

sys.platform 返回操作系统平台名称

sys.stdout 标准输出

sys.stdin 标准输入

sys.stderr 错误输出

sys.exc\_clear() 用来清除当前线程所出现的当前的或最近的错误信息

sys.exec\_prefix 返回平台独立的python文件安装的位置

sys.byteorder 本地字节规则的指示器，big-endian平台的值是'big',little-endian平台的值是'little'

sys.copyright 记录python版权相关的东西

sys.api\_version 解释器的C的API版本

**sys.stdin,sys.stdout,sys.stderr**

stdin , stdout , 以及stderr 变量包含与标准I/O 流对应的流对象. 如果需要更好地控制输出,而print 不能满足你的要求, 它们就是你所需要的. 你也可以替换它们, 这时候你就可以重定向输出和输入到其它设备( device ), 或者以非标准的方式处理它们

我们常用print和raw\_input来进行输入和打印，那么print 和 raw\_input是如何与标准输入/输出流建立关系的呢？  
其实Python程序的标准输入/输出/出错流定义在sys模块中，分别 为： sys.stdin,sys.stdout, sys.stderr  
下列的程序也可以用来输入和输出是一样的：  
import sys  
  
sys.stdout.write('HelloWorld!')  
  
print 'Please enter yourname:',  
name=sys.stdin.readline()[:-1]  
print 'Hi, %s!' % name  
  
那么sys.stdin, sys.stdout, stderr到底是什么呢？我们在Python运行环境中输入以下代码：  
import sys  
for f in (sys.stdin,sys.stdout, sys.stderr): print f  
输出为：  
<open file'<stdin>', mode 'r' at 892210>  
<open file'<stdout>', mode 'w' at 892270>  
<open file'<stderr>', mode 'w at 8922d0>  
  
由此可以看出stdin, stdout, stderr在Python中无非都是文件属性的对象，他们在Python启动时自动与Shell 环境中的标准输入，输出，出错关联。  
而Python程序的在Shell中的I/O重定向与本文开始时举的DOS命令的重定向完全相同，其实这种重定向是由Shell来提供的，与Python 本身并无关系。那么我们是否可以在Python程序内部将stdin,stdout,stderr读写操作重定向到一个内部对象呢？答案是肯定的。  
Python提供了一个StringIO模块来完成这个设想，比如：  
from StringIO import StringIO  
import sys  
buff =StringIO()  
  
temp =sys.stdout                              #保存标准I/O流  
sys.stdout =buff                                #将标准I/O流重定向到buff对象  
print 42, 'hello', 0.001  
  
sys.stdout=temp                                #恢复标准I/O流  
print buff.getvalue()

time模块

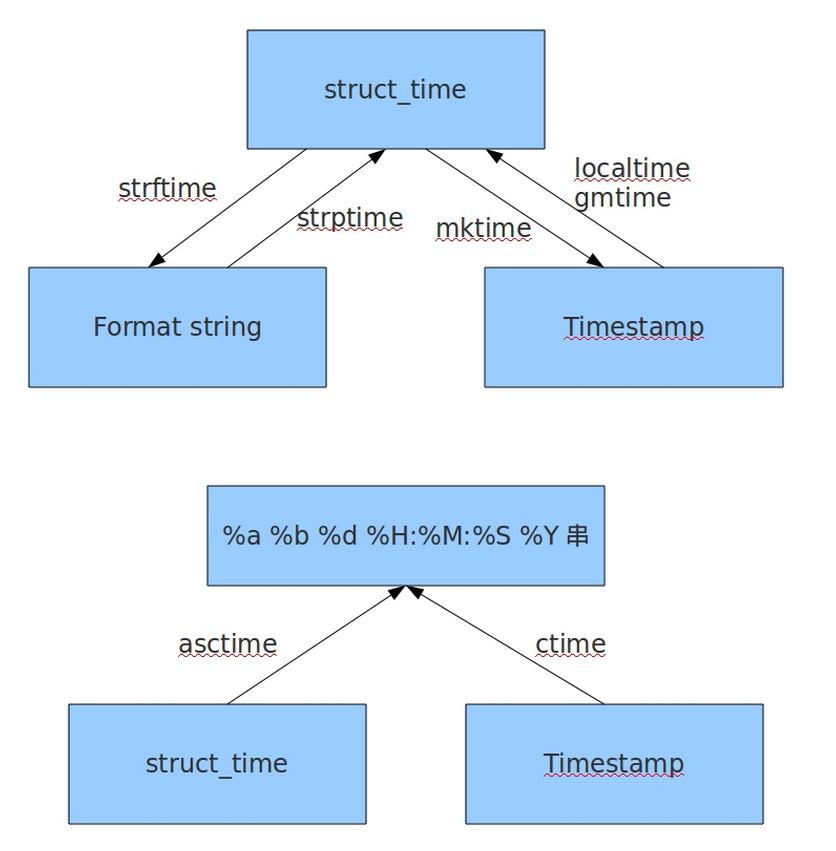
 time模块中时间表现的格式主要有三种：

　　a、timestamp时间戳，时间戳表示的是从1970年1月1日00:00:00开始按秒计算的偏移量

　　b、struct\_time时间元组，共有九个元素组。

　　c、format time 格式化时间，已格式化的结构使时间更具可读性。包括自定义格式和固定格式。

**1、时间格式转换图：**



**2、主要time生成方法和time格式转换方法实例：**

[复制代码](javascript:void(0);)

复制代码

#! /usr/bin/env python

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

# \_\_author\_\_ = "TKQ"

import time

**# 生成timestamp**

time.time()

# 1477471508.05  
#struct\_time to timestamp  
time.mktime(time.localtime())

**#生成struct\_time**

# timestamp to struct\_time 本地时间

time.localtime()

time.localtime(time.time())

# time.struct\_time(tm\_year=2016, tm\_mon=10, tm\_mday=26, tm\_hour=16, tm\_min=45, tm\_sec=8, tm\_wday=2, tm\_yday=300, tm\_isdst=0)

# timestamp to struct\_time 格林威治时间

time.gmtime()

time.gmtime(time.time())

# time.struct\_time(tm\_year=2016, tm\_mon=10, tm\_mday=26, tm\_hour=8, tm\_min=45, tm\_sec=8, tm\_wday=2, tm\_yday=300, tm\_isdst=0)

#format\_time to struct\_time

time.strptime('2011-05-05 16:37:06', '%Y-%m-%d %X')

# time.struct\_time(tm\_year=2011, tm\_mon=5, tm\_mday=5, tm\_hour=16, tm\_min=37, tm\_sec=6, tm\_wday=3, tm\_yday=125, tm\_isdst=-1)

**#生成format\_time**

#struct\_time to format\_time

time.strftime("%Y-%m-%d %X")

time.strftime("%Y-%m-%d %X",time.localtime())

# 2016-10-26 16:48:41

**#生成固定格式的时间表示格式**

time.asctime(time.localtime())

time.ctime(time.time())

# Wed Oct 26 16:45:08 2016

复制代码

[复制代码](javascript:void(0);)

**struct\_time元组元素结构**

[复制代码](javascript:void(0);)

复制代码

属性 值

tm\_year（年） 比如2011

tm\_mon（月） 1 - 12

tm\_mday（日） 1 - 31

tm\_hour（时） 0 - 23

tm\_min（分） 0 - 59

tm\_sec（秒） 0 - 61

tm\_wday（weekday） 0 - 6（0表示周日）

tm\_yday（一年中的第几天） 1 - 366

tm\_isdst（是否是夏令时） 默认为-1

复制代码

[复制代码](javascript:void(0);)

**format time结构化表示**

|  |  |
| --- | --- |
| **格式** | **含义** |
| %a | 本地（locale）简化星期名称 |
| %A | 本地完整星期名称 |
| %b | 本地简化月份名称 |
| %B | 本地完整月份名称 |
| %c | 本地相应的日期和时间表示 |
| %d | 一个月中的第几天（01 - 31） |
| %H | 一天中的第几个小时（24小时制，00 - 23） |
| %I | 第几个小时（12小时制，01 - 12） |
| %j | 一年中的第几天（001 - 366） |
| %m | 月份（01 - 12） |
| %M | 分钟数（00 - 59） |
| %p | 本地am或者pm的相应符 |
| %S | 秒（01 - 61） |
| %U | 一年中的星期数。（00 - 53星期天是一个星期的开始。）第一个星期天之前的所有天数都放在第0周。 |
| %w | 一个星期中的第几天（0 - 6，0是星期天） |
| %W | 和%U基本相同，不同的是%W以星期一为一个星期的开始。 |
| %x | 本地相应日期 |
| %X | 本地相应时间 |
| %y | 去掉世纪的年份（00 - 99） |
| %Y | 完整的年份 |
| %Z | 时区的名字（如果不存在为空字符） |
| %% | ‘%’字符 |

常见结构化时间组合：

print time.strftime("%Y-%m-%d %X")

#2016-10-26 20:50:13

 3、time加减

[复制代码](javascript:void(0);)

复制代码

#timestamp加减单位以秒为单位

import time

t1 = time.time()

t2=t1+10

print time.ctime(t1)#Wed Oct 26 21:15:30 2016

print time.ctime(t2)#Wed Oct 26 21:15:40 2016

复制代码

[复制代码](javascript:void(0);)

**datetime模块**

datatime模块重新封装了time模块，提供更多接口，提供的类有：date,time,datetime,timedelta,tzinfo。

**1、date类**

datetime.date(year, month, day)

静态方法和字段

date.max、date.min：date对象所能表示的最大、最小日期；

date.resolution：date对象表示日期的最小单位。这里是天。

date.today()：返回一个表示当前本地日期的date对象；

date.fromtimestamp(timestamp)：根据给定的时间戮，返回一个date对象；

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif output

方法和属性

[复制代码](javascript:void(0);)

复制代码

d1 = date(2011,06,03)#date对象

d1.year、date.month、date.day：年、月、日；

d1.replace(year, month, day)：生成一个新的日期对象，用参数指定的年，月，日代替原有对象中的属性。（原有对象仍保持不变）

d1.timetuple()：返回日期对应的time.struct\_time对象；

d1.weekday()：返回weekday，如果是星期一，返回0；如果是星期2，返回1，以此类推；

d1.isoweekday()：返回weekday，如果是星期一，返回1；如果是星期2，返回2，以此类推；

d1.isocalendar()：返回格式如(year，month，day)的元组；

d1.isoformat()：返回格式如'YYYY-MM-DD’的字符串；

d1.strftime(fmt)：和time模块format相同。

复制代码

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif output

**2、time类**

datetime.time(hour[ , minute[ , second[ , microsecond[ , tzinfo] ] ] ] )

静态方法和字段

time.min、time.max：time类所能表示的最小、最大时间。其中，time.min = time(0, 0, 0, 0)， time.max = time(23, 59, 59, 999999)；

time.resolution：时间的最小单位，这里是1微秒；

方法和属性

[复制代码](javascript:void(0);)

复制代码

t1 = datetime.time(10,23,15)#time对象  
t1.hour、t1.minute、t1.second、t1.microsecond：时、分、秒、微秒；

t1.tzinfo：时区信息；

t1.replace([ hour[ , minute[ , second[ , microsecond[ , tzinfo] ] ] ] ] )：创建一个新的时间对象，用参数指定的时、分、秒、微秒代替原有对象中的属性（原有对象仍保持不变）；

t1.isoformat()：返回型如"HH:MM:SS"格式的字符串表示；

t1.strftime(fmt)：同time模块中的format；

复制代码

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif output

**3、datetime类**

datetime相当于date和time结合起来。  
datetime.datetime (year, month, day[ , hour[ , minute[ , second[ , microsecond[ , tzinfo] ] ] ] ] )

静态方法和字段

[复制代码](javascript:void(0);)

复制代码

datetime.today()：返回一个表示当前本地时间的datetime对象；

datetime.now([tz])：返回一个表示当前本地时间的datetime对象，如果提供了参数tz，则获取tz参数所指时区的本地时间；

datetime.utcnow()：返回一个当前utc时间的datetime对象；#格林威治时间

datetime.fromtimestamp(timestamp[, tz])：根据时间戮创建一个datetime对象，参数tz指定时区信息；

datetime.utcfromtimestamp(timestamp)：根据时间戮创建一个datetime对象；

datetime.combine(date, time)：根据date和time，创建一个datetime对象；

datetime.strptime(date\_string, format)：将格式字符串转换为datetime对象；

复制代码

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif output

方法和属性

[复制代码](javascript:void(0);)

复制代码

dt=datetime.now()#datetime对象

dt.year、month、day、hour、minute、second、microsecond、tzinfo：

dt.date()：获取date对象；

dt.time()：获取time对象；

dt. replace ([ year[ , month[ , day[ , hour[ , minute[ , second[ , microsecond[ , tzinfo] ] ] ] ] ] ] ])：

dt. timetuple ()

dt. utctimetuple ()

dt. toordinal ()

dt. weekday ()

dt. isocalendar ()

dt. isoformat ([ sep] )

dt. ctime ()：返回一个日期时间的C格式字符串，等效于time.ctime(time.mktime(dt.timetuple()))；

dt. strftime (format)

复制代码

[复制代码](javascript:void(0);)

**4.timedelta类，时间加减**

使用timedelta可以很方便的在日期上做天days，小时hour，分钟，秒，毫秒，微妙的时间计算，如果要计算月份则需要另外的办法。

[复制代码](javascript:void(0);)

复制代码

#coding:utf-8

from datetime import \*

dt = datetime.now()

#日期减一天

dt1 = dt + timedelta(days=-1)#昨天

dt2 = dt - timedelta(days=1)#昨天

dt3 = dt + timedelta(days=1)#明天

delta\_obj = dt3-dt

print type(delta\_obj),delta\_obj#<type 'datetime.timedelta'> 1 day, 0:00:00

print delta\_obj.days ,delta\_obj.total\_seconds()#1 86400.0

复制代码

[复制代码](javascript:void(0);)

**5、tzinfo时区类**

复制代码

#! /usr/bin/python

# coding=utf-8

from datetime import datetime, tzinfo,timedelta

"""

tzinfo是关于时区信息的类

tzinfo是一个抽象类，所以不能直接被实例化

"""

class UTC(tzinfo):

"""UTC"""

def \_\_init\_\_(self,offset = 0):

self.\_offset = offset

def utcoffset(self, dt):

return timedelta(hours=self.\_offset)

def tzname(self, dt):

return "UTC +%s" % self.\_offset

def dst(self, dt):

return timedelta(hours=self.\_offset)

#北京时间

beijing = datetime(2011,11,11,0,0,0,tzinfo = UTC(8))

print "beijing time:",beijing

#曼谷时间

bangkok = datetime(2011,11,11,0,0,0,tzinfo = UTC(7))

print "bangkok time",bangkok

#北京时间转成曼谷时间

print "beijing-time to bangkok-time:",beijing.astimezone(UTC(7))

#计算时间差时也会考虑时区的问题

timespan = beijing - bangkok

print "时差:",timespan

#Output==================

# beijing time: 2011-11-11 00:00:00+08:00

# bangkok time 2011-11-11 00:00:00+07:00

# beijing-time to bangkok-time: 2011-11-10 23:00:00+07:00

# 时差: -1 day, 23:00:00

**hashlib加密**

hashlib主要提供字符加密功能，将md5和sha模块整合到了一起，支持md5,sha1, sha224, sha256, sha384, sha512等算法

**具体应用**

[复制代码](javascript:void(0);)

复制代码

#!/usr/bin/env python

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

#pyversion:python3.5

#owner:fuzj

import hashlib

# ######## md5 ########

string = "beyongjie"

md5 = hashlib.md5()

md5.update(string.encode('utf-8')) #注意转码

res = md5.hexdigest()

print("md5加密结果:",res)

# ######## sha1 ########

sha1 = hashlib.sha1()

sha1.update(string.encode('utf-8'))

res = sha1.hexdigest()

print("sha1加密结果:",res)

# ######## sha256 ########

sha256 = hashlib.sha256()

sha256.update(string.encode('utf-8'))

res = sha256.hexdigest()

print("sha256加密结果:",res)

# ######## sha384 ########

sha384 = hashlib.sha384()

sha384.update(string.encode('utf-8'))

res = sha384.hexdigest()

print("sha384加密结果:",res)

# ######## sha512 ########

sha512= hashlib.sha512()

sha512.update(string.encode('utf-8'))

res = sha512.hexdigest()

print("sha512加密结果:",res)

复制代码

[复制代码](javascript:void(0);)

输出结果：

[复制代码](javascript:void(0);)

复制代码

md5加密结果: 0e725e477851ff4076f774dc312d4748

sha1加密结果: 458d32be8ea38b66300174970ab0a8c0b734252f

sha256加密结果: 1e62b55bfd02977943f885f6a0998af7cc9cfb95c8ac4a9f30ecccb7c05ec9f4

sha384加密结果: e91cdf0d2570de5c96ee84e8a12cddf16508685e7a03b3e811099cfcd54b7f52183e20197cff7c07f312157f0ba4875b

sha512加密结果: 3f0020a726e9c1cb5d22290c967f3dd1bcecb409a51a8088db520750c876aaec3f17a70d7981cd575ed4b89471f743f3f24a146a39d59f215ae3e208d0170073

复制代码

[复制代码](javascript:void(0);)

注意：hashlib 加密啊的字符串类型为二进制编码，直接加密字符串会报如下错误：

[复制代码](javascript:void(0);)

复制代码

sha1 = hashlib.sha1()

sha1.update(string)

res = sha1.hexdigest()

print("sha1加密结果:",res)

TypeError: Unicode-objects must be encoded before hashing

复制代码

[复制代码](javascript:void(0);)

可以使用encode进行转换

shaa1 = hashlib.sha1()

shaa1.update(string.encode('utf-8'))

res = shaa1.hexdigest()

print("sha1采用encode转换加密结果:",res)

或者使用byte转换为二进制

shab1 = hashlib.sha1()

shab1.update(bytes(string,encoding='utf-8'))

res = shab1.hexdigest()

print("sha1采用byte转换的结果:",res)

以上输出：

sha1采用encode转换加密结果: 458d32be8ea38b66300174970ab0a8c0b734252f

sha1采用byte转换的结果: 458d32be8ea38b66300174970ab0a8c0b734252f

**常用方法**

* hash.update(arg) 更新哈希对象以字符串参数, 注意：如果同一个hash对象重复调用该方法，则m.update(a); m.update(b) 等效于 m.update(a+b)，看下面例子

[复制代码](javascript:void(0);)

复制代码

m = hashlib.md5()

m.update('a'.encode('utf-8'))

res = m.hexdigest()

print("第一次a加密:",res)

m.update('b'.encode('utf-8'))

res = m.hexdigest()

print("第二次b加密:",res)

m1 = hashlib.md5()

m1.update('b'.encode('utf-8'))

res = m1.hexdigest()

print("b单独加密:",res)

m2 = hashlib.md5()

m2.update('ab'.encode('utf-8'))

res = m2.hexdigest()

print("ab单独加密:",res)

输出结果：

第一次a加密: 0cc175b9c0f1b6a831c399e269772661

第二次b加密: 187ef4436122d1cc2f40dc2b92f0eba0

b单独加密: 92eb5ffee6ae2fec3ad71c777531578f

ab单独加密: 187ef4436122d1cc2f40dc2b92f0eba0

复制代码

[复制代码](javascript:void(0);)

* hash.digest() 返回摘要，作为二进制数据字符串值,
* hash.hexdigest() 返回摘要，作为十六进制数据字符串值,
* hash.copy() 复制

**高级加密**

以上加密算法虽然依然非常厉害，但时候存在缺陷，即：通过撞库可以反解。所以，有必要对加密算法中添加自定义key再来做加密。

[复制代码](javascript:void(0);)

复制代码

low = hashlib.md5()

low.update('ab'.encode('utf-8'))

res = low.hexdigest()

print("普通加密:",res)

high = hashlib.md5(b'beyondjie')

high.update('ab'.encode('utf-8'))

res = high.hexdigest()

print("采用key加密:",res)

输出结果：

普通加密: 187ef4436122d1cc2f40dc2b92f0eba0

采用key加密: 1b073f6b8cffe609751e4c98537b7653

复制代码

[复制代码](javascript:void(0);)

**附加HMAC-SHA1各语言版本实现**

复制代码

在各大开放平台大行其道的互联网开发潮流中，调用各平台的API接口过程中，无一例外都会用到计算签名值(sig值)。而在各种计算签名的方法中，经常被采用的就是HMAC-SHA1，现对HMAC-SHA1做一个简单的介绍：

HMAC，散列消息鉴别码，基于密钥的Hash算法认证协议。实现原理为：利用已经公开的Hash函数和私有的密钥，来生成固定长度的消息鉴别码；

SHA1、MD5等Hash算法是比较常用的不可逆Hash签名计算方法；

BASE64，将任意序列的8字节字符转换为人眼无法直接识别的符号编码的一种方法；

各个语言版本的实现为：

Python版：

import hmac

import hashlib

import base64

hmac.new(Token,data,hashlib.sha1).digest().encode('base64').rstrip()

Token：即接口的key

data：要加密的数据

PHP版：

base64\_encode(hash\_hmac("SHA1",clientStr,Token , true))

C++版（Openssl)：

HMAC( EVP\_sha1(),

/\*key data\*/ strKey.data(),

/\*key len\*/ strKey.size(),

/\*data \*/(unsigned char\*) strRandom.data(),

/\*data len\*/ strRandom.size(), digest, &digest\_len))

Shell版：

echo -n '3f88a95c532bea70' | openssl dgst -hmac '123' -sha1 -binary | base64

**1 logging模块简介**

logging模块是Python内置的标准模块，主要用于输出运行日志，可以设置输出日志的等级、日志保存路径、日志文件回滚等；相比print，具备如下优点：

1. 可以通过设置不同的日志等级，在release版本中只输出重要信息，而不必显示大量的调试信息；
2. print将所有信息都输出到标准输出中，严重影响开发者从标准输出中查看其它数据；logging则可以由开发者决定将信息输出到什么地方，以及怎么输出；

**2 logging模块使用**

**2.1 基本使用**

配置logging基本的设置，然后在控制台输出日志，

[复制代码](javascript:void(0);)

复制代码

复制代码

import logging

logging.basicConfig(level = logging.INFO,format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s')

logger = logging.getLogger(\_\_name\_\_)

logger.info("Start print log")

logger.debug("Do something")

logger.warning("Something maybe fail.")

logger.info("Finish")

复制代码

复制代码

[复制代码](javascript:void(0);)

运行时，控制台输出，

2016-10-09 19:11:19,434 - \_\_main\_\_ - INFO - Start print log

2016-10-09 19:11:19,434 - \_\_main\_\_ - WARNING - Something maybe fail.

2016-10-09 19:11:19,434 - \_\_main\_\_ - INFO - Finish

logging中可以选择很多消息级别，如debug、info、warning、error以及critical。通过赋予logger或者handler不同的级别，开发者就可以只输出错误信息到特定的记录文件，或者在调试时只记录调试信息。

例如，我们将logger的级别改为DEBUG，再观察一下输出结果，

logging.basicConfig(level = logging.DEBUG,format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s')

控制台输出，可以发现，输出了debug的信息。

2016-10-09 19:12:08,289 - \_\_main\_\_ - INFO - Start print log

2016-10-09 19:12:08,289 - \_\_main\_\_ - DEBUG - Do something

2016-10-09 19:12:08,289 - \_\_main\_\_ - WARNING - Something maybe fail.

2016-10-09 19:12:08,289 - \_\_main\_\_ - INFO - Finish

logging.basicConfig函数各参数：

filename：指定日志文件名；

filemode：和file函数意义相同，指定日志文件的打开模式，'w'或者'a'；

format：指定输出的格式和内容，format可以输出很多有用的信息，

参数：作用

%(levelno)s：打印日志级别的数值

%(levelname)s：打印日志级别的名称

%(pathname)s：打印当前执行程序的路径，其实就是sys.argv[0]

%(filename)s：打印当前执行程序名

%(funcName)s：打印日志的当前函数

%(lineno)d：打印日志的当前行号

%(asctime)s：打印日志的时间

%(thread)d：打印线程ID

%(threadName)s：打印线程名称

%(process)d：打印进程ID

%(message)s：打印日志信息

datefmt：指定时间格式，同time.strftime()；

level：设置日志级别，默认为logging.WARNNING；

stream：指定将日志的输出流，可以指定输出到sys.stderr，sys.stdout或者文件，默认输出到sys.stderr，当stream和filename同时指定时，stream被忽略；

**2.2 将日志写入到文件**

**2.2.1 将日志写入到文件**

设置logging，创建一个FileHandler，并对输出消息的格式进行设置，将其添加到logger，然后将日志写入到指定的文件中，

[复制代码](javascript:void(0);)

复制代码

复制代码

import logging

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(level = logging.INFO)

handler = logging.FileHandler("log.txt")

handler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

handler.setFormatter(formatter)

logger.addHandler(handler)

logger.info("Start print log")

logger.debug("Do something")

logger.warning("Something maybe fail.")

logger.info("Finish")

复制代码

复制代码

[复制代码](javascript:void(0);)

log.txt中日志数据为，

2016-10-09 19:01:13,263 - \_\_main\_\_ - INFO - Start print log

2016-10-09 19:01:13,263 - \_\_main\_\_ - WARNING - Something maybe fail.

2016-10-09 19:01:13,263 - \_\_main\_\_ - INFO - Finish

**2.2.2 将日志同时输出到屏幕和日志文件**

logger中添加StreamHandler，可以将日志输出到屏幕上，

[复制代码](javascript:void(0);)

复制代码

复制代码

import logging

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(level = logging.INFO)

handler = logging.FileHandler("log.txt")

handler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

handler.setFormatter(formatter)

console = logging.StreamHandler()

console.setLevel(logging.INFO)

logger.addHandler(handler)

logger.addHandler(console)

logger.info("Start print log")

logger.debug("Do something")

logger.warning("Something maybe fail.")

logger.info("Finish")

复制代码

复制代码

[复制代码](javascript:void(0);)

可以在log.txt文件和控制台中看到，

2016-10-09 19:20:46,553 - \_\_main\_\_ - INFO - Start print log

2016-10-09 19:20:46,553 - \_\_main\_\_ - WARNING - Something maybe fail.

2016-10-09 19:20:46,553 - \_\_main\_\_ - INFO - Finish

可以发现，logging有一个日志处理的主对象，其他处理方式都是通过addHandler添加进去，logging中包含的handler主要有如下几种，

handler名称：位置；作用

StreamHandler：logging.StreamHandler；日志输出到流，可以是sys.stderr，sys.stdout或者文件

FileHandler：logging.FileHandler；日志输出到文件

BaseRotatingHandler：logging.handlers.BaseRotatingHandler；基本的日志回滚方式

RotatingHandler：logging.handlers.RotatingHandler；日志回滚方式，支持日志文件最大数量和日志文件回滚

TimeRotatingHandler：logging.handlers.TimeRotatingHandler；日志回滚方式，在一定时间区域内回滚日志文件

SocketHandler：logging.handlers.SocketHandler；远程输出日志到TCP/IP sockets

DatagramHandler：logging.handlers.DatagramHandler；远程输出日志到UDP sockets

SMTPHandler：logging.handlers.SMTPHandler；远程输出日志到邮件地址

SysLogHandler：logging.handlers.SysLogHandler；日志输出到syslog

NTEventLogHandler：logging.handlers.NTEventLogHandler；远程输出日志到Windows NT/2000/XP的事件日志

MemoryHandler：logging.handlers.MemoryHandler；日志输出到内存中的指定buffer

HTTPHandler：logging.handlers.HTTPHandler；通过"GET"或者"POST"远程输出到HTTP服务器

**2.2.3 日志回滚**

使用RotatingFileHandler，可以实现日志回滚，

[复制代码](javascript:void(0);)

复制代码

复制代码

import logging

from logging.handlers import RotatingFileHandler

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(level = logging.INFO)

#定义一个RotatingFileHandler，最多备份3个日志文件，每个日志文件最大1K

rHandler = RotatingFileHandler("log.txt",maxBytes = 1\*1024,backupCount = 3)

rHandler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

rHandler.setFormatter(formatter)

console = logging.StreamHandler()

console.setLevel(logging.INFO)

console.setFormatter(formatter)

logger.addHandler(rHandler)

logger.addHandler(console)

logger.info("Start print log")

logger.debug("Do something")

logger.warning("Something maybe fail.")

logger.info("Finish")

复制代码

复制代码

[复制代码](javascript:void(0);)

可以在工程目录中看到，备份的日志文件，

2016/10/09 19:36 732 log.txt

2016/10/09 19:36 967 log.txt.1

2016/10/09 19:36 985 log.txt.2

2016/10/09 19:36 976 log.txt.3

**2.3 设置消息的等级**

可以设置不同的日志等级，用于控制日志的输出，

日志等级：使用范围

FATAL：致命错误

CRITICAL：特别糟糕的事情，如内存耗尽、磁盘空间为空，一般很少使用

ERROR：发生错误时，如IO操作失败或者连接问题

WARNING：发生很重要的事件，但是并不是错误时，如用户登录密码错误

INFO：处理请求或者状态变化等日常事务

DEBUG：调试过程中使用DEBUG等级，如算法中每个循环的中间状态

**2.4 捕获traceback**

Python中的traceback模块被用于跟踪异常返回信息，可以在logging中记录下traceback，

代码，

[复制代码](javascript:void(0);)

复制代码

复制代码

import logging

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(level = logging.INFO)

handler = logging.FileHandler("log.txt")

handler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

handler.setFormatter(formatter)

console = logging.StreamHandler()

console.setLevel(logging.INFO)

logger.addHandler(handler)

logger.addHandler(console)

logger.info("Start print log")

logger.debug("Do something")

logger.warning("Something maybe fail.")

try:

open("sklearn.txt","rb")

except (SystemExit,KeyboardInterrupt):

raise

except Exception:

logger.error("Faild to open sklearn.txt from logger.error",exc\_info = True)

logger.info("Finish")

复制代码

复制代码

[复制代码](javascript:void(0);)

控制台和日志文件log.txt中输出，

Start print log

Something maybe fail.

Faild to open sklearn.txt from logger.error

Traceback (most recent call last):

File "G:\zhb7627\Code\Eclipse WorkSpace\PythonTest\test.py", line 23, in <module>

open("sklearn.txt","rb")

IOError: [Errno 2] No such file or directory: 'sklearn.txt'

Finish

也可以使用logger.exception(msg,\_args)，它等价于logger.error(msg,exc\_info = True,\_args)，

将

logger.error("Faild to open sklearn.txt from logger.error",exc\_info = True)

替换为，

logger.exception("Failed to open sklearn.txt from logger.exception")

控制台和日志文件log.txt中输出，

Start print log

Something maybe fail.

Failed to open sklearn.txt from logger.exception

Traceback (most recent call last):

File "G:\zhb7627\Code\Eclipse WorkSpace\PythonTest\test.py", line 23, in <module>

open("sklearn.txt","rb")

IOError: [Errno 2] No such file or directory: 'sklearn.txt'

Finish

**2.5 多模块使用logging**

主模块mainModule.py，

[复制代码](javascript:void(0);)

复制代码

复制代码

import logging

import subModule

logger = logging.getLogger("mainModule")

logger.setLevel(level = logging.INFO)

handler = logging.FileHandler("log.txt")

handler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

handler.setFormatter(formatter)

console = logging.StreamHandler()

console.setLevel(logging.INFO)

console.setFormatter(formatter)

logger.addHandler(handler)

logger.addHandler(console)

logger.info("creating an instance of subModule.subModuleClass")

a = subModule.SubModuleClass()

logger.info("calling subModule.subModuleClass.doSomething")

a.doSomething()

logger.info("done with subModule.subModuleClass.doSomething")

logger.info("calling subModule.some\_function")

subModule.som\_function()

logger.info("done with subModule.some\_function")

复制代码

复制代码

[复制代码](javascript:void(0);)

子模块subModule.py，

[复制代码](javascript:void(0);)

复制代码

复制代码

import logging

module\_logger = logging.getLogger("mainModule.sub")

class SubModuleClass(object):

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

self.logger = logging.getLogger("mainModule.sub.module")

self.logger.info("creating an instance in SubModuleClass")

def doSomething(self):

self.logger.info("do something in SubModule")

a = []

a.append(1)

self.logger.debug("list a = " + str(a))

self.logger.info("finish something in SubModuleClass")

def som\_function():

module\_logger.info("call function some\_function")

复制代码

复制代码

[复制代码](javascript:void(0);)

执行之后，在控制和日志文件log.txt中输出，

2016-10-09 20:25:42,276 - mainModule - INFO - creating an instance of subModule.subModuleClass

2016-10-09 20:25:42,279 - mainModule.sub.module - INFO - creating an instance in SubModuleClass

2016-10-09 20:25:42,279 - mainModule - INFO - calling subModule.subModuleClass.doSomething

2016-10-09 20:25:42,279 - mainModule.sub.module - INFO - do something in SubModule

2016-10-09 20:25:42,279 - mainModule.sub.module - INFO - finish something in SubModuleClass

2016-10-09 20:25:42,279 - mainModule - INFO - done with subModule.subModuleClass.doSomething

2016-10-09 20:25:42,279 - mainModule - INFO - calling subModule.some\_function

2016-10-09 20:25:42,279 - mainModule.sub - INFO - call function some\_function

2016-10-09 20:25:42,279 - mainModule - INFO - done with subModule.some\_function

首先在主模块定义了logger'mainModule'，并对它进行了配置，就可以在解释器进程里面的其他地方通过getLogger('mainModule')得到的对象都是一样的，不需要重新配置，可以直接使用。定义的该logger的子logger，都可以共享父logger的定义和配置，所谓的父子logger是通过命名来识别，任意以'mainModule'开头的logger都是它的子logger，例如'mainModule.sub'。

实际开发一个application，首先可以通过logging配置文件编写好这个application所对应的配置，可以生成一个根logger，如'PythonAPP'，然后在主函数中通过fileConfig加载logging配置，接着在application的其他地方、不同的模块中，可以使用根logger的子logger，如'PythonAPP.Core'，'PythonAPP.Web'来进行log，而不需要反复的定义和配置各个模块的logger。

**3 通过JSON或者YAML文件配置logging模块**

尽管可以在Python代码中配置logging，但是这样并不够灵活，最好的方法是使用一个配置文件来配置。在Python 2.7及以后的版本中，可以从字典中加载logging配置，也就意味着可以通过JSON或者YAML文件加载日志的配置。

**3.1 通过JSON文件配置**

JSON配置文件，

{

"version":1,

"disable\_existing\_loggers":false,

"formatters":{

"simple":{

"format":"%(asctime)s - %(name)s - %(levelname)s - %(message)s"

}

},

"handlers":{

"console":{

"class":"logging.StreamHandler",

"level":"DEBUG",

"formatter":"simple",

"stream":"ext://sys.stdout"

},

"info\_file\_handler":{

"class":"logging.handlers.RotatingFileHandler",

"level":"INFO",

"formatter":"simple",

"filename":"info.log",

"maxBytes":"10485760",

"backupCount":20,

"encoding":"utf8"

},

"error\_file\_handler":{

"class":"logging.handlers.RotatingFileHandler",

"level":"ERROR",

"formatter":"simple",

"filename":"errors.log",

"maxBytes":10485760,

"backupCount":20,

"encoding":"utf8"

}

},

"loggers":{

"my\_module":{

"level":"ERROR",

"handlers":["info\_file\_handler"],

"propagate":"no"

}

},

"root":{

"level":"INFO",

"handlers":["console","info\_file\_handler","error\_file\_handler"]

}

}

通过JSON加载配置文件，然后通过logging.dictConfig配置logging，

[复制代码](javascript:void(0);)

复制代码

复制代码

import json

import logging.config

import os

def setup\_logging(default\_path = "logging.json",default\_level = logging.INFO,env\_key = "LOG\_CFG"):

path = default\_path

value = os.getenv(env\_key,None)

if value:

path = value

if os.path.exists(path):

with open(path,"r") as f:

config = json.load(f)

logging.config.dictConfig(config)

else:

logging.basicConfig(level = default\_level)

def func():

logging.info("start func")

logging.info("exec func")

logging.info("end func")

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

setup\_logging(default\_path = "logging.json")

func()

复制代码

复制代码

[复制代码](javascript:void(0);)

**3.2 通过YAML文件配置**

通过YAML文件进行配置，比JSON看起来更加简介明了，

version: 1

disable\_existing\_loggers: False

formatters:

simple:

format: "%(asctime)s - %(name)s - %(levelname)s - %(message)s"

handlers:

console:

class: logging.StreamHandler

level: DEBUG

formatter: simple

stream: ext://sys.stdout

info\_file\_handler:

class: logging.handlers.RotatingFileHandler

level: INFO

formatter: simple

filename: info.log

maxBytes: 10485760

backupCount: 20

encoding: utf8

error\_file\_handler:

class: logging.handlers.RotatingFileHandler

level: ERROR

formatter: simple

filename: errors.log

maxBytes: 10485760

backupCount: 20

encoding: utf8

loggers:

my\_module:

level: ERROR

handlers: [info\_file\_handler]

propagate: no

root:

level: INFO

handlers: [console,info\_file\_handler,error\_file\_handler]

通过YAML加载配置文件，然后通过logging.dictConfig配置logging，

[复制代码](javascript:void(0);)

复制代码

复制代码

import yaml

import logging.config

import os

def setup\_logging(default\_path = "logging.yaml",default\_level = logging.INFO,env\_key = "LOG\_CFG"):

path = default\_path

value = os.getenv(env\_key,None)

if value:

path = value

if os.path.exists(path):

with open(path,"r") as f:

config = yaml.load(f)

logging.config.dictConfig(config)

else:

logging.basicConfig(level = default\_level)

def func():

logging.info("start func")

logging.info("exec func")

logging.info("end func")

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

setup\_logging(default\_path = "logging.yaml")

func()

复制代码

复制代码

[复制代码](javascript:void(0);)

**subprocess模块**

subprocess是Python 2.4中新增的一个模块，它允许你生成新的进程，连接到它们的 input/output/error 管道，并获取它们的返回（状态）码。这个模块的目的在于替换几个旧的模块和方法，如：

* os.system
* os.spawn\*

**1. subprocess模块中的常用函数**

| **函数** | **描述** |
| --- | --- |
| subprocess.run() | Python 3.5中新增的函数。执行指定的命令，等待命令执行完成后返回一个包含执行结果的CompletedProcess类的实例。 |
| subprocess.call() | 执行指定的命令，返回命令执行状态，其功能类似于os.system(cmd)。 |
| subprocess.check\_call() | Python 2.5中新增的函数。 执行指定的命令，如果执行成功则返回状态码，否则抛出异常。其功能等价于subprocess.run(..., check=True)。 |
| subprocess.check\_output() | Python 2.7中新增的的函数。执行指定的命令，如果执行状态码为0则返回命令执行结果，否则抛出异常。 |
| subprocess.getoutput(cmd) | 接收字符串格式的命令，执行命令并返回执行结果，其功能类似于os.popen(cmd).read()和commands.getoutput(cmd)。 |
| subprocess.getstatusoutput(cmd) | 执行cmd命令，返回一个元组(命令执行状态, 命令执行结果输出)，其功能类似于commands.getstatusoutput()。 |

***说明：***

1. 在Python 3.5之后的版本中，官方文档中提倡通过subprocess.run()函数替代其他函数来使用subproccess模块的功能；
2. 在Python 3.5之前的版本中，我们可以通过subprocess.call()，subprocess.getoutput()等上面列出的其他函数来使用subprocess模块的功能；
3. subprocess.run()、subprocess.call()、subprocess.check\_call()和subprocess.check\_output()都是通过对subprocess.Popen的封装来实现的高级函数，因此如果我们需要更复杂功能时，可以通过subprocess.Popen来完成。
4. subprocess.getoutput()和subprocess.getstatusoutput()函数是来自Python 2.x的commands模块的两个遗留函数。它们隐式的调用系统shell，并且不保证其他函数所具有的安全性和异常处理的一致性。另外，它们从Python 3.3.4开始才支持Windows平台。

**2. 上面各函数的定义及参数说明**

**函数参数列表：**

subprocess.run(args, \*, stdin=None, input=None, stdout=None, stderr=None, shell=False, timeout=None, check=False, universal\_newlines=False)

subprocess.call(args, \*, stdin=None, stdout=None, stderr=None, shell=False, timeout=None)

subprocess.check\_call(args, \*, stdin=None, stdout=None, stderr=None, shell=False, timeout=None)

subprocess.check\_output(args, \*, stdin=None, stderr=None, shell=False, universal\_newlines=False, timeout=None)

subprocess.getstatusoutput(cmd)

subprocess.getoutput(cmd)

**参数说明：**

* ***args：*** 要执行的shell命令，默认应该是一个字符串序列，如['df', '-Th']或('df', '-Th')，也可以是一个字符串，如'df -Th'，但是此时需要把shell参数的值置为True。
* ***shell：*** 如果shell为True，那么指定的命令将通过shell执行。如果我们需要访问某些shell的特性，如管道、文件名通配符、环境变量扩展功能，这将是非常有用的。当然，python本身也提供了许多类似shell的特性的实现，如glob、fnmatch、os.walk()、os.path.expandvars()、os.expanduser()和shutil等。
* ***check：*** 如果check参数的值是True，且执行命令的进程以非0状态码退出，则会抛出一个CalledProcessError的异常，且该异常对象会包含 参数、退出状态码、以及stdout和stderr(如果它们有被捕获的话)。
* ***stdout, stderr：***
* run()函数默认不会捕获命令执行结果的正常输出和错误输出，如果我们向获取这些内容需要传递subprocess.PIPE，然后可以通过返回的CompletedProcess类实例的stdout和stderr属性或捕获相应的内容；
* call()和check\_call()函数返回的是命令执行的状态码，而不是CompletedProcess类实例，所以对于它们而言，stdout和stderr不适合赋值为subprocess.PIPE；
* check\_output()函数默认就会返回命令执行结果，所以不用设置stdout的值，如果我们希望在结果中捕获错误信息，可以执行stderr=subprocess.STDOUT。
* ***input：*** 该参数是传递给Popen.communicate()，通常该参数的值必须是一个字节序列，如果universal\_newlines=True，则其值应该是一个字符串。
* ***universal\_newlines：*** 该参数影响的是输入与输出的数据格式，比如它的值默认为False，此时stdout和stderr的输出是字节序列；当该参数的值设置为True时，stdout和stderr的输出是字符串。

**3. subprocess.CompletedProcess类介绍**

需要说明的是，subprocess.run()函数是Python3.5中新增的一个高级函数，其返回值是一个subprocess.CompletedPorcess类的实例，因此，subprocess.completedPorcess类也是Python 3.5中才存在的。它表示的是一个已结束进程的状态信息，它所包含的属性如下：

* ***args：*** 用于加载该进程的参数，这可能是一个列表或一个字符串
* ***returncode：*** 子进程的退出状态码。通常情况下，退出状态码为0则表示进程成功运行了；一个负值-N表示这个子进程被信号N终止了
* ***stdout：*** 从子进程捕获的stdout。这通常是一个字节序列，如果run()函数被调用时指定universal\_newlines=True，则该属性值是一个字符串。如果run()函数被调用时指定stderr=subprocess.STDOUT，那么stdout和stderr将会被整合到这一个属性中，且stderr将会为None
* ***stderr：*** 从子进程捕获的stderr。它的值与stdout一样，是一个字节序列或一个字符串。如果stderr灭有被捕获的话，它的值就为None
* ***check\_returncode()：*** 如果returncode是一个非0值，则该方法会抛出一个CalledProcessError异常。

**4. 实例**

**subprocess.run()**

>>> subprocess.run(["ls", "-l"]) # doesn't capture output

CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)

Traceback (most recent call last):

...

subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], stdout=subprocess.PIPE)

CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,

stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n')

**subprocess.call()**

>>> subprocess.call(['ls', '-l'])

总用量 160

drwxr-xr-x 2 wader wader 4096 12月 7 2015 公共的

drwxr-xr-x 2 wader wader 4096 12月 7 2015 模板

drwxr-xr-x 2 wader wader 4096 12月 7 2015 视频

drwxr-xr-x 2 wader wader 4096 12月 7 2015 图片

drwxr-xr-x 2 wader wader 4096 12月 7 2015 文档

drwxr-xr-x 2 wader wader 4096 4月 13 2016 下载

drwxr-xr-x 2 wader wader 4096 12月 7 2015 音乐

drwxr-xr-x 7 wader wader 4096 5月 26 2016 桌面

0

>>> subprocess.call('ls -l', shell=True)

总用量 160

drwxr-xr-x 2 wader wader 4096 12月 7 2015 公共的

drwxr-xr-x 2 wader wader 4096 12月 7 2015 模板

drwxr-xr-x 2 wader wader 4096 12月 7 2015 视频

drwxr-xr-x 2 wader wader 4096 12月 7 2015 图片

drwxr-xr-x 2 wader wader 4096 12月 7 2015 文档

drwxr-xr-x 2 wader wader 4096 4月 13 2016 下载

drwxr-xr-x 2 wader wader 4096 12月 7 2015 音乐

drwxr-xr-x 7 wader wader 4096 5月 26 2016 桌面

0

>>> subprocess.call(['ls', '-l'], stdout=subprocess.DEVNULL)

0

>>> subprocess.call(['ls', '-l', '/test'])

ls: 无法访问/test: 没有那个文件或目录

2

**suprocess.check\_call()**

>>> subprocess.check\_call(['ls', '-l'])

总用量 160

drwxr-xr-x 2 wader wader 4096 12月 7 2015 公共的

drwxr-xr-x 2 wader wader 4096 12月 7 2015 模板

drwxr-xr-x 2 wader wader 4096 12月 7 2015 视频

drwxr-xr-x 2 wader wader 4096 12月 7 2015 图片

drwxr-xr-x 2 wader wader 4096 12月 7 2015 文档

drwxr-xr-x 2 wader wader 4096 4月 13 2016 下载

drwxr-xr-x 2 wader wader 4096 12月 7 2015 音乐

drwxr-xr-x 7 wader wader 4096 5月 26 2016 桌面

0

>>> subprocess.check\_call('ls -l', shell=True)

总用量 160

drwxr-xr-x 2 wader wader 4096 12月 7 2015 公共的

drwxr-xr-x 2 wader wader 4096 12月 7 2015 模板

drwxr-xr-x 2 wader wader 4096 12月 7 2015 视频

drwxr-xr-x 2 wader wader 4096 12月 7 2015 图片

drwxr-xr-x 2 wader wader 4096 12月 7 2015 文档

drwxr-xr-x 2 wader wader 4096 4月 13 2016 下载

drwxr-xr-x 2 wader wader 4096 12月 7 2015 音乐

drwxr-xr-x 7 wader wader 4096 5月 26 2016 桌面

0

>>> subprocess.check\_call('ls -l /test', shell=True)

ls: 无法访问/test: 没有那个文件或目录

Traceback (most recent call last):

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

File "/usr/lib/python3.4/subprocess.py", line 557, in check\_call

raise CalledProcessError(retcode, cmd)

subprocess.CalledProcessError: Command 'ls -l /test' returned non-zero exit status 2

**sbuprocess.check\_output()**

>>> ret = subprocess.check\_output(['ls', '-l'])

>>> print(ret)

b' \xe5\x85\xac\xe5\x85\xb1\xe7\x9a\x84\ndrwxr-xr-x 2 wader wader 4096 12\xe6\x9c\x88 7 2015 \xe6\xa8\xa1\xe6\x9d\xbf\ndrwxr-xr-x 2 wader wader 4096 12\xe6\x9c\x88 7 2015 \xe8\xa7\x86\xe9\xa2\x91\ndrwxr-xr-x 2 wader wader 4096 12\xe6\x9c\x88 7 2015 \xe5\x9b\xbe\xe7\x89\x87\ndrwxr-xr-x 2 wader wader 4096 12\xe6\x9c\x88 7 2015 \xe6\x96\x87\xe6\xa1\xa3\ndrwxr-xr-x 2 wader wader 4096 4\xe6\x9c\x88 13 2016 \xe4\xb8\x8b\xe8\xbd\xbd\ndrwxr-xr-x 2 wader wader 4096 12\xe6\x9c\x88 7 2015 \xe9\x9f\xb3\xe4\xb9\x90\ndrwxr-xr-x 7 wader wader 4096 5\xe6\x9c\x88 26 2016 \xe6\xa1\x8c\xe9\x9d\xa2\n'

>>> ret = subprocess.check\_output(['ls', '-l'], universal\_newlines=True)

>>> print(ret)

总用量 160

drwxr-xr-x 2 wader wader 4096 12月 7 2015 公共的

drwxr-xr-x 2 wader wader 4096 12月 7 2015 模板

drwxr-xr-x 2 wader wader 4096 12月 7 2015 视频

drwxr-xr-x 2 wader wader 4096 12月 7 2015 图片

drwxr-xr-x 2 wader wader 4096 12月 7 2015 文档

drwxr-xr-x 2 wader wader 4096 4月 13 2016 下载

drwxr-xr-x 2 wader wader 4096 12月 7 2015 音乐

drwxr-xr-x 7 wader wader 4096 5月 26 2016 桌面

**subprocess.getoutput()与subprocess.getstatusoutput()**

>>> ret = subprocess.getoutput('ls -l')

>>> print(ret)

总用量 160

drwxr-xr-x 2 wader wader 4096 12月 7 2015 公共的

drwxr-xr-x 2 wader wader 4096 12月 7 2015 模板

drwxr-xr-x 2 wader wader 4096 12月 7 2015 视频

drwxr-xr-x 2 wader wader 4096 12月 7 2015 图片

drwxr-xr-x 2 wader wader 4096 12月 7 2015 文档

drwxr-xr-x 2 wader wader 4096 4月 13 2016 下载

drwxr-xr-x 2 wader wader 4096 12月 7 2015 音乐

drwxr-xr-x 7 wader wader 4096 5月 26 2016 桌面

>>> retcode, output = subprocess.getstatusoutput('ls -l')

>>> print(retcode)

0

>>> print(output)

总用量 160

drwxr-xr-x 2 wader wader 4096 12月 7 2015 公共的

drwxr-xr-x 2 wader wader 4096 12月 7 2015 模板

drwxr-xr-x 2 wader wader 4096 12月 7 2015 视频

drwxr-xr-x 2 wader wader 4096 12月 7 2015 图片

drwxr-xr-x 2 wader wader 4096 12月 7 2015 文档

drwxr-xr-x 2 wader wader 4096 4月 13 2016 下载

drwxr-xr-x 2 wader wader 4096 12月 7 2015 音乐

drwxr-xr-x 7 wader wader 4096 5月 26 2016 桌面

>>> retcode, output = subprocess.getstatusoutput('ls -l /test')

>>> print(retcode)

2

>>> print(output)

ls: 无法访问/test: 没有那个文件或目录

**三、subprocess.Popen介绍**

该类用于在一个新的进程中执行一个子程序。前面我们提到过，上面介绍的这些函数都是基于subprocess.Popen类实现的，通过使用这些被封装后的高级函数可以很方面的完成一些常见的需求。由于subprocess模块底层的进程创建和管理是由Popen类来处理的，因此，当我们无法通过上面哪些高级函数来实现一些不太常见的功能时就可以通过subprocess.Popen类提供的灵活的api来完成。

**1.subprocess.Popen的构造函数**

class subprocess.Popen(args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None,

preexec\_fn=None, close\_fds=True, shell=False, cwd=None, env=None, universal\_newlines=False,

startup\_info=None, creationflags=0, restore\_signals=True, start\_new\_session=False, pass\_fds=())

参数说明：

* ***args：*** 要执行的shell命令，可以是字符串，也可以是命令各个参数组成的序列。当该参数的值是一个字符串时，该命令的解释过程是与平台相关的，因此通常建议将args参数作为一个序列传递。
* ***bufsize：*** 指定缓存策略，0表示不缓冲，1表示行缓冲，其他大于1的数字表示缓冲区大小，负数 表示使用系统默认缓冲策略。
* ***stdin, stdout, stderr：*** 分别表示程序标准输入、输出、错误句柄。
* ***preexec\_fn：*** 用于指定一个将在子进程运行之前被调用的可执行对象，只在Unix平台下有效。
* ***close\_fds：*** 如果该参数的值为True，则除了0,1和2之外的所有文件描述符都将会在子进程执行之前被关闭。
* ***shell：*** 该参数用于标识是否使用shell作为要执行的程序，如果shell值为True，则建议将args参数作为一个字符串传递而不要作为一个序列传递。
* ***cwd：*** 如果该参数值不是None，则该函数将会在执行这个子进程之前改变当前工作目录。
* ***env：*** 用于指定子进程的环境变量，如果env=None，那么子进程的环境变量将从父进程中继承。如果env!=None，它的值必须是一个映射对象。
* ***universal\_newlines：*** 如果该参数值为True，则该文件对象的stdin，stdout和stderr将会作为文本流被打开，否则他们将会被作为二进制流被打开。
* ***startupinfo和creationflags：*** 这两个参数只在Windows下有效，它们将被传递给底层的CreateProcess()函数，用于设置子进程的一些属性，如主窗口的外观，进程优先级等。

**2. subprocess.Popen类的实例可调用的方法**

| **方法** | **描述** |
| --- | --- |
| Popen.poll() | 用于检查子进程（命令）是否已经执行结束，没结束返回None，结束后返回状态码。 |
| Popen.wait(timeout=None) | 等待子进程结束，并返回状态码；如果在timeout指定的秒数之后进程还没有结束，将会抛出一个TimeoutExpired异常。 |
| Popen.communicate(input=None, timeout=None) | 该方法可用来与进程进行交互，比如发送数据到stdin，从stdout和stderr读取数据，直到到达文件末尾。 |
| Popen.send\_signal(signal) | 发送指定的信号给这个子进程。 |
| Popen.terminate() | 停止该子进程。 |
| Popen.kill() | 杀死该子进程。 |

**关于communicate()方法的说明：**

* 该方法中的可选参数 input 应该是将被发送给子进程的数据，或者如没有数据发送给子进程，该参数应该是None。input参数的数据类型必须是字节串，如果universal\_newlines参数值为True，则input参数的数据类型必须是字符串。
* 该方法返回一个元组(stdout\_data, stderr\_data)，这些数据将会是字节穿或字符串（如果universal\_newlines的值为True）。
* 如果在timeout指定的秒数后该进程还没有结束，将会抛出一个TimeoutExpired异常。捕获这个异常，然后重新尝试通信不会丢失任何输出的数据。但是超时之后子进程并没有被杀死，为了合理的清除相应的内容，一个好的应用应该手动杀死这个子进程来结束通信。
* 需要注意的是，这里读取的数据是缓冲在内存中的，所以，如果数据大小非常大或者是无限的，就不应该使用这个方法。

**3. subprocess.Popen使用实例**

**实例1：**

>>> import subprocess

>>>

>>> p = subprocess.Popen('df -Th', stdout=subprocess.PIPE, shell=True)

>>> print(p.stdout.read())

Filesystem Type Size Used Avail Use% Mounted on

/dev/vda1 ext4 40G 12G 26G 31% /

devtmpfs devtmpfs 3.9G 0 3.9G 0% /dev

tmpfs tmpfs 3.9G 0 3.9G 0% /dev/shm

tmpfs tmpfs 3.9G 386M 3.5G 10% /run

tmpfs tmpfs 3.9G 0 3.9G 0% /sys/fs/cgroup

tmpfs tmpfs 783M 0 783M 0% /run/user/0

tmpfs tmpfs 783M 0 783M 0% /run/user/1000

**实例2：**

>>> obj = subprocess.Popen(["python"], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

>>> obj.stdin.write('print(1) \n')

>>> obj.stdin.write('print(2) \n')

>>> obj.stdin.write('print(3) \n')

>>> out,err = obj.communicate()

>>> print(out)

1

2

3

>>> print(err)

**实例3：**

>>> obj = subprocess.Popen(["python"], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

>>> out,err = obj.communicate(input='print(1) \n')

>>> print(out)

1

>>> print(err)

**实例4：**

实现类似df -Th | grep data命令的功能，实际上就是实现shell中管道的共功能。

>>>

>>> p1 = subprocess.Popen(['df', '-Th'], stdout=subprocess.PIPE)

>>> p2 = subprocess.Popen(['grep', 'data'], stdin=p1.stdout, stdout=subprocess.PIPE)

>>> out,err = p2.communicate()

>>> print(out)

/dev/vdb1 ext4 493G 4.8G 463G 2% /data

/dev/vdd1 ext4 1008G 420G 537G 44% /data1

/dev/vde1 ext4 985G 503G 432G 54% /data2

>>> print(err)

None

**四、总结**

那么我们到底该用哪个模块、哪个函数来执行命令与系统及系统进行交互呢？下面我们来做个总结：

* 首先应该知道的是，Python2.4版本引入了subprocess模块用来替换os.system()、os.popen()、os.spawn\*()等函数以及commands模块；也就是说如果你使用的是Python 2.4及以上的版本就应该使用subprocess模块了。
* 如果你的应用使用的Python 2.4以上，但是是Python 3.5以下的版本，Python官方给出的建议是使用subprocess.call()函数。Python 2.5中新增了一个subprocess.check\_call()函数，Python 2.7中新增了一个subprocess.check\_output()函数，这两个函数也可以按照需求进行使用。
* 如果你的应用使用的是Python 3.5及以上的版本（目前应该还很少），Python官方给出的建议是尽量使用subprocess.run()函数。
* 当subprocess.call()、subprocess.check\_call()、subprocess.check\_output()和subprocess.run()这些高级函数无法满足需求时，我们可以使用subprocess.Popen类来实现我们需要的复杂功能。

[**json ,pickle模块**](https://www.cnblogs.com/tjuyuan/p/6795860.html)

JSON(JavaScript Object Notation, JS 对象标记) 是一种轻量级的数据交换格式。JSON的数据格式其实就是python里面的字典格式，里面可以包含方括号括起来的数组，也就是python里面的列表。

在python中，有专门处理json格式的模块—— json 和 picle模块

  Json   模块提供了四个方法： dumps、dump、loads、load

pickle 模块也提供了四个功能：dumps、dump、loads、load

一. dumps 和 dump:

 dumps和dump   序列化方法

 dumps只完成了序列化为str，

       dump必须传文件描述符，将序列化的str保存到文件中

查看源码：

def dumps(obj, skipkeys=False, ensure\_ascii=True, check\_circular=True,

allow\_nan=True, cls=None, indent=None, separators=None,

default=None, sort\_keys=False, \*\*kw):

# Serialize ``obj`` to a JSON formatted ``str``.

# 序列号 “obj” 数据类型 转换为 JSON格式的字符串

def dump(obj, fp, skipkeys=False, ensure\_ascii=True, check\_circular=True,

allow\_nan=True, cls=None, indent=None, separators=None,

default=None, sort\_keys=False, \*\*kw):

"""Serialize ``obj`` as a JSON formatted stream to ``fp`` (a

``.write()``-supporting file-like object).

我理解为两个动作，一个动作是将”obj“转换为JSON格式的字符串，还有一个动作是将字符串写入到文件中，也就是说文件描述符fp是必须要的参数 """

示例代码：

[复制代码](javascript:void(0);)

复制代码

>>> import json

>>> json.dumps([]) # dumps可以格式化所有的基本数据类型为字符串

'[]'

>>> json.dumps(1) # 数字

'1'

>>> json.dumps('1') # 字符串

'"1"'

>>> dict = {"name":"Tom", "age":23}

>>> json.dumps(dict) # 字典

'{"name": "Tom", "age": 23}'

复制代码

[复制代码](javascript:void(0);)

a = {"name":"Tom", "age":23}

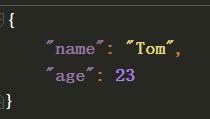
with open("test.json", "w", encoding='utf-8') as f:

# indent 超级好用，格式化保存字典，默认为None，小于0为零个空格

f.write(json.dumps(a, indent=4))

# json.dump(a,f,indent=4) # 和上面的效果一样

保存的文件效果：



二. loads 和 load

loads和load  反序列化方法

       loads 只完成了反序列化，

       load 只接收文件描述符，完成了读取文件和反序列化

 查看源码：

def loads(s, encoding=None, cls=None, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, object\_pairs\_hook=None, \*\*kw):

"""Deserialize ``s`` (a ``str`` instance containing a JSON document) to a Python object.

将包含str类型的JSON文档反序列化为一个python对象"""

def load(fp, cls=None, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, object\_pairs\_hook=None, \*\*kw):

"""Deserialize ``fp`` (a ``.read()``-supporting file-like object containing a JSON document) to a Python object.

将一个包含JSON格式数据的可读文件饭序列化为一个python对象"""

实例：

>>> json.loads('{"name":"Tom", "age":23}')

{'age': 23, 'name': 'Tom'}

[复制代码](javascript:void(0);)

复制代码

import json

with open("test.json", "r", encoding='utf-8') as f:

aa = json.loads(f.read())

f.seek(0)

bb = json.load(f) # 与 json.loads(f.read())

print(aa)

print(bb)

# 输出：

{'name': 'Tom', 'age': 23}

{'name': 'Tom', 'age': 23}

复制代码

[复制代码](javascript:void(0);)

三. json 和 pickle 模块

 json模块和pickle模块都有  dumps、dump、loads、load四种方法，而且用法一样。

不用的是json模块序列化出来的是通用格式，其它编程语言都认识，就是普通的字符串，

而picle模块序列化出来的只有python可以认识，其他编程语言不认识的，表现为乱码

不过picle可以序列化函数，但是其他文件想用该函数，在该文件中需要有该文件的定义（定义和参数必须相同，内容可以不同）

四. python对象（obj） 与json对象的对应关系

[复制代码](javascript:void(0);)

复制代码

+-------------------+---------------+

| Python | JSON |

+===================+===============+

| dict | object |

+-------------------+---------------+

| list, tuple | array |

+-------------------+---------------+

| str | string |

+-------------------+---------------+

| int, float | number |

+-------------------+---------------+

| True | true |

+-------------------+---------------+

| False | false |

+-------------------+---------------+

| None | null |

+-------------------+---------------+

复制代码

[复制代码](javascript:void(0);)

 五. 总结

 1. json序列化方法：

          dumps：无文件操作            dump：序列化+写入文件

  2. json反序列化方法：

          loads：无文件操作              load： 读文件+反序列化

  3. json模块序列化的数据 更通用

      picle模块序列化的数据 仅python可用，但功能强大，可以序列号函数

  4. json模块可以序列化和反序列化的  数据类型 见  python对象（obj） 与json对象的对应关系表

  5. 格式化写入文件利用  indent = 4

**ElementTree是python的XML处理模块**

ElementTree是python的XML处理模块，它提供了一个轻量级的对象模型。它在Python2.5以后成为Python标准库的一部分，但是Python2.4之前需要单独安装。在使用ElementTree模块时，需要import xml.etree.ElementTree的操作。

ElementTree表示整个XML节点树，而Element表示节点数中的一个单独的节点。

**构建**

ElementTree(tag)，其中tag表示根节点，初始化一个ElementTree对象。

Element(tag, attrib={}, \*\*extra)函数用来构造XML的一个根节点，其中tag表示根节点的名称，attrib是一个可选项，表示节点的属性。

SubElement(parent, tag, attrib={}, \*\*extra)用来构造一个已经存在的节点的子节点

Element.text和SubElement.text表示element对象的额外的内容属性，Element.tag和Element.attrib分别表示element对象的标签和属性。

ElementTree.write(file, encoding='us-ascii', xml\_declaration=None, default\_namespace=None, method='xml')，函数新建一个XML文件，并且将节点数数据写入XML文件中。

#encoding=utf-8

import xml.etree.ElementTree as ET

#新建xml文件

def buildNewsXmlFile():

#设置一个新节点，并设置其标签为root

root = ET.Element("root")

#在root下新建两个子节点,设置其名称分别为sina和chinabyte

sina = ET.SubElement(root, "sina")

chinabyte = ET.SubElement(root, "chinabyte")

#在sina下新建两个子节点，设置其节点名称分别为number和first

sina\_number = ET.SubElement(sina, "number")

sina\_number.text = "1"

sina\_first = ET.SubElement(sina, "first")

sina\_first.text = "http://roll.tech.sina.com.cn/internet\_all/index\_1.shtml"

#在chinabyte下新建两个子节点，设置其节点名称为number和first

chinabyte\_number = ET.SubElement(chinabyte, "number")

chinabyte\_number.text = "1"

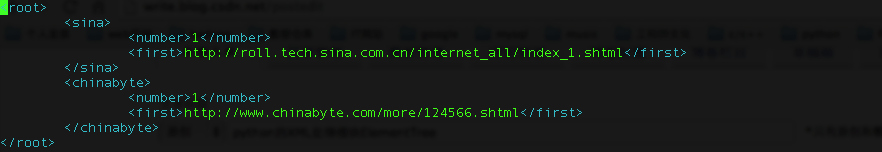
chinabyte\_first = ET.SubElement(chinabyte, "first")

chinabyte\_first.text = "http://www.chinabyte.com/more/124566.shtml"

#将节点数信息保存在ElementTree中，并且保存为XML格式文件

tree = ET.ElementTree(root)

tree.write("urlfile.xml")



**解析和修改XML文件**

ElementTree.parse(source, parser=None)，将xml文件加载并返回ElementTree对象。parser是一个可选的参数，如果为空，则默认使用标准的XMLParser解析器。

ElementTree.getroot()，得到根节点。返回根节点的element对象。

Element.remove(tag)，删除root下名称为tag的子节点

以下函数，ElementTree和Element的对象都包含。

find(match)，得到第一个匹配match的子节点，match可以是一个标签名称或者是路径。返回个element

findtext(match,default=None)，得到第一个配置的match的element的内容

findall(match)，得到匹配match下的所有的子节点，match可以是一个标签或者是路径，它会返回一个list，包含匹配的elements的信息

iter(tag)，创建一个以当前节点为根节点的iterator。

这里有一个xml文件

<?xml version="1.0"?>

<data>

<country name="Liechtenstein">

<rank>1</rank>

<year>2008</year>

<gdppc>141100</gdppc>

<neighbor name="Austria" direction="E"/>

<neighbor name="Switzerland" direction="W"/>

</country>

<country name="Singapore">

<rank>4</rank>

<year>2011</year>

<gdppc>59900</gdppc>

<neighbor name="Malaysia" direction="N"/>

</country>

<country name="Panama">

<rank>68</rank>

<year>2011</year>

<gdppc>13600</gdppc>

<neighbor name="Costa Rica" direction="W"/>

<neighbor name="Colombia" direction="E"/>

</country>

</data>

现在是解析xml文件的代码

#解析Xml文件

def parseXmlFile(xml\_name):

#将XMl文件加载并返回一个ELementTree对象

tree = ET.parse(xml\_name)

#得到第一个匹配sina标签的Element对象

sina = tree.find("contry")

#得到sina的SubElement

for sub\_tag in sina:

print sub\_tag.text

#得到所有匹配sina标签的Element对象的list集合

list\_contry = tree.findall("contry")

for contry in list\_contry:

for sub\_tag in contry:

print sub\_tag.text

#修改xml文件

for rank in tree.iter('rank')

new\_rank = int(rank.text)+1

rank.text = str(new\_rank)

rank.set('updated', 'yes')

tree.write(xml\_name)

第一次的输出是：1，2008，14100

第二次的输出是：1，2008，14100，4，2011，59900，68，2011，13600

修改后的xml文件为

<?xml version="1.0"?>

<data>

<country name="Liechtenstein">

<rank updated="yes">2</rank>

<year>2008</year>

<gdppc>141100</gdppc>

<neighbor name="Austria" direction="E"/>

<neighbor name="Switzerland" direction="W"/>

</country>

<country name="Singapore">

<rank updated="yes">5</rank>

<year>2011</year>

<gdppc>59900</gdppc>

<neighbor name="Malaysia" direction="N"/>

</country>

<country name="Panama">

<rank updated="yes">69</rank>

<year>2011</year>

<gdppc>13600</gdppc>

<neighbor name="Costa Rica" direction="W"/>

<neighbor name="Colombia" direction="E"/>

</country>

</data>

**configparser 简介**

configparser 是 Pyhton 标准库中用来解析配置文件的模块，并且内置方法和字典非常接近。Python2.x 中名为 ConfigParser，3.x 已更名小写，并加入了一些新功能。  
配置文件的格式如下：

[DEFAULT]

ServerAliveInterval = 45

Compression = yes

CompressionLevel = 9

ForwardX11 = yes

[bitbucket.org]

User = Tom

[topsecret.com]

Port: 50022

ForwardX11: no

“[ ]”包含的为 section，section 下面为类似于 key - value 的配置内容；  
configparser 默认支持 ‘=’ ‘：’ 两种分隔。

**configparser 常用方法**

**初始化实例**

使用 configparser 首先需要初始化实例，并读取配置文件：

>>> import configparser

>>> config = configparser.ConfigParser() # 注意大小写

>>> config.read("config.ini") # 配置文件的路径

["config.ini"]

或者可以直接读字典

>>> parser = configparser.ConfigParser()

>>> parser.read\_dict({'section1': {'key1': 'value1',

... 'key2': 'value2',

... 'key3': 'value3'},

... 'section2': {'keyA': 'valueA',

... 'keyB': 'valueB',

... 'keyC': 'valueC'},

... 'section3': {'foo': 'x',

... 'bar': 'y',

... 'baz': 'z'}

... })

**获取所有 sections**

>>> config.sections()

['bitbucket.org', 'topsecret.com'] # 注意会过滤掉[DEFAULT]

**获取指定 section 的 keys & values**

>>> config.items('topsecret.com')

>>>> [('port', '50022'), ('forwardx11', 'no')] # 注意items()返回的字符串会全变成小写

**获取指定 section 的 keys**

>>> config.options('topsecret.com')

['Port', 'ForwardX11']

>>> for option in config['topsecret.com']:

... print(option)

Port

ForwardX11

**获取指定 key 的 value**

>>> config['bitbucket.org']['User']

'Tom'

>>> config.get('bitbucket.org', 'User')

'Tom'

>>> config.getint('topsecret.com', 'Port')

50022

**检查**

>>> 'DEFAULT' in config

True

>>> 'test' in config['section\_test']

False

>>> 'Tom' in config['bitbucket.org']['User']

True

>>> config.has\_section('bitbucket.org')

True

>>> config.has\_option('section\_test', 'test')

False

**添加**

>>> config.add\_section('Section\_1')

>>> config.set('Section\_1', 'key\_1', 'value\_1') # 注意键值是用set()方法

>>> config.write(open('config.ini', 'w')) # 一定要写入才生效

**删除**

>>> config.remove\_option('Section\_1', 'key\_1')

True

>>> config.remove\_section('Section\_1')

True

>>> config.clear() # 清空除[DEFAULT]之外所有内容

>>> config.write(open('config.ini', 'w'))

**关于 [DEFAULT]**

[DEFAULT] 一般包含 ini 格式配置文件的默认项，所以 configparser 部分方法会自动跳过这个 section 。  
前面已经提到 sections() 是获取不到的，还有删除方法对 [DEFAULT] 也无效：

>>> config.remove\_section('DEFAULT')

False

>>> config.clear()

>>> 'DEFAULT' in config

True

>>> 'ForwardX11' in config['DEFAULT']

True

>>> config.sections()

[]

但指定删除和修改 [DEFAULT] 里的 keys & values 是可以的：

>>> config.remove\_option('DEFAULT', 'ForwardX11')

True

>>> config.set('DEFAULT', 'ForwardX11','no')

>>> config['DEFAULT']['ForwardX11']

'no'

还有个特殊的是，has\_section() 也无效，可以和 in 区别使用

>>> config.has\_section('DEFAULT')

False

>>> 'DEFAULT' in config

True

更多用法请看官方文档：<https://docs.python.org/3.6/library/configparser.html>

**randmon（获取随机数）**

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

**random.uniform**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

复制代码 代码如下:

print random.uniform(10, 20)  
print random.uniform(20, 10)  
# 18.7356606526  
# 12.5798298022

**random.randint**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

复制代码 代码如下:

print random.randint(12, 20)  # 生成的随机数 n: 12 <= n <= 20  
print random.randint(20, 20)  # 结果永远是20      
# print random.randint(20, 10)  # 该语句是错误的。下限必须小于上限

**random.randrange**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

**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。有关sequence可以查看python手册数据模型这一章。下面是使用choice的一些例子：

复制代码 代码如下:

print random.choice("学习Python")  
print random.choice(["JGood", "is", "a", "handsome", "boy"])  
print random.choice(("Tuple", "List", "Dict"))

**random.shuffle**random.shuffle(x[, random])，用于将一个列表中的元素打乱。如:

复制代码 代码如下:

p = ["Python", "is", "powerful", "simple", "and so on..."]  
random.shuffle(p)  
print p  
# ['powerful', 'simple', 'is', 'Python', 'and so on...']

**random.sample**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

复制代码 代码如下:

list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
slice = random.sample(list, 5)  # 从list中随机获取5个元素，作为一个片断返回  
print slice  
print list  # 原有序列并没有改变

随机整数：

复制代码 代码如下:

>>> import random  
>>> random.randint(0,99)  
# 21

随机选取0到100间的偶数：

复制代码 代码如下:

>>> import random  
>>> random.randrange(0, 101, 2)  
# 42

随机浮点数：

复制代码 代码如下:

>>> import random  
>>> random.random()  
0.85415370477785668  
>>> random.uniform(1, 10)  
# 5.4221167969800881

随机字符：

复制代码 代码如下:

>>> import random  
>>> random.choice('abcdefg&#%^\*f')  
# 'd'

多个字符中选取特定数量的字符：

复制代码 代码如下:

>>> import random  
random.sample('abcdefghij', 3)  
# ['a', 'd', 'b']

多个字符中选取特定数量的字符组成新字符串：

复制代码 代码如下:

>>> import random  
>>> import string  
>>> string.join( random.sample(['a','b','c','d','e','f','g','h','i','j'], 3) ).replace(" ","")  
# 'fih'

随机选取字符串：

复制代码 代码如下:

>>> import random  
>>> random.choice ( ['apple', 'pear', 'peach', 'orange', 'lemon'] )  
# 'lemon'

洗牌：

复制代码 代码如下:

>>> import random  
>>> items = [1, 2, 3, 4, 5, 6]  
>>> random.shuffle(items)  
>>> items  
# [3, 2, 5, 6, 4, 1]

**shutil模块**

引入： import shutil

**copy()**

功能：复制文件

格式：shutil.copy('来源文件','目标地址')

返回值：复制之后的路径

**copy2()**

功能：复制文件，保留元数据

格式：shutil.copy2('来源文件','目标地址')

返回值：复制之后的路径

**copyfileobj()**

将一个文件的内容拷贝的另外一个文件当中

格式：shutil.copyfileobj(open(来源文件,'r'),open（'目标文件','w'）)

返回值：无

**copyfile()**

功能：将一个文件的内容拷贝的另外一个文件当中

格式:shutil.copyfile(来源文件,目标文件)

返回值：目标文件的路径

**copytree()**

功能：复制整个文件目录

格式:shutil.copytree(来源目录,目标目录)

返回值：目标目录的路径

注意：无论文件夹是否为空，均可以复制，而且会复制文件夹中的所有内容

**copymode()**

功能：拷贝权限

**copystat()**

功能：拷贝元数据（状态）

**rmtree()**

功能：移除整个目录，无论是否空

格式：shutil.rmtree(目录路径)

返回值：无

**move()**

功能：移动文件或者文件夹

格式：shutil.move(来源地址,目标地址)

返回值：目标地址

**which()**

功能：检测命令对应的文件路径

格式：shutil.which(‘命令字符串’)

返回值：命令文件所在位置

注意：window和linux不太一样。 window的命令都是.exe结尾，linux则不是

**disk\_usage()**

功能：检测磁盘使用信息

格式：disk\_usage(‘盘符’)

返回值：元组

**归档和解包操作**

归档：将多个文件合并到一个文件当中，这种操作方式就是归档。

解包：将归档的文件进行释放。

压缩：压缩时将多个文件进行有损或者无损的合并到一个文件当中。

解压缩：就是压缩的反向操作，将压缩文件中的多个文件，释放出来。

**注意：压缩属于归档！**

**make\_archive()**

功能：归档函数，归档操作

格式：shutil.make\_archive('目标文件路径','归档文件后缀','需要归档的目录')

返回值：归档文件的最终路径

**unpack\_archive()**

功能：解包操作

格式：shutil.unpack\_archive('归档文件路径','解包目标文件夹')

返回值:None

注意：文件夹不存在会新建文件夹

**get\_archive\_formats()**

功能：获取当前系统已注册的归档文件格式（后缀）

格式：shutil.get\_archive\_formats()

返回值：列表 [(后缀,解释),(后缀,解释),(后缀,解释)...]

**get\_unpack\_formats()**

功能：获取当前系统已经注册的解包文件格式(后缀)

格式:shutil.get\_unpack\_formats()

返回值：列表 [(后缀,解释),(后缀,解释),(后缀,解释)...]

**paramiko**

paramiko是一个用于做远程控制的模块，使用该模块可以对远程服务器进行命令或文件操作，值得一说的是，fabric和ansible内部的远程管理就是使用的paramiko来现实。

1、下载安装

|  |  |
| --- | --- |
| 1  2  3 | pycrypto，由于 paramiko 模块内部依赖pycrypto，所以先下载安装pycrypto  pip3 install pycrypto  pip3 install paramiko |

2、模块使用

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

#!/usr/bin/env python

#coding:utf-8

import paramiko

ssh = paramiko.SSHClient()

ssh.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())

ssh.connect('192.168.1.108', 22, 'alex', '123')

stdin, stdout, stderr = ssh.exec\_command('df')

print stdout.read()

ssh.close();

复制代码

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

import paramiko

private\_key\_path = '/home/auto/.ssh/id\_rsa'

key = paramiko.RSAKey.from\_private\_key\_file(private\_key\_path)

ssh = paramiko.SSHClient()

ssh.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())

ssh.connect('主机名 ', 端口, '用户名', key)

stdin, stdout, stderr = ssh.exec\_command('df')

print stdout.read()

ssh.close()

复制代码

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

import os,sys

import paramiko

t = paramiko.Transport(('182.92.219.86',22))

t.connect(username='wupeiqi',password='123')

sftp = paramiko.SFTPClient.from\_transport(t)

sftp.put('/tmp/test.py','/tmp/test.py')

t.close()

import os,sys

import paramiko

t = paramiko.Transport(('182.92.219.86',22))

t.connect(username='wupeiqi',password='123')

sftp = paramiko.SFTPClient.from\_transport(t)

sftp.get('/tmp/test.py','/tmp/test2.py')

t.close()

复制代码

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

import paramiko

pravie\_key\_path = '/home/auto/.ssh/id\_rsa'

key = paramiko.RSAKey.from\_private\_key\_file(pravie\_key\_path)

t = paramiko.Transport(('182.92.219.86',22))

t.connect(username='wupeiqi',pkey=key)

sftp = paramiko.SFTPClient.from\_transport(t)

sftp.put('/tmp/test3.py','/tmp/test3.py')

t.close()

import paramiko

pravie\_key\_path = '/home/auto/.ssh/id\_rsa'

key = paramiko.RSAKey.from\_private\_key\_file(pravie\_key\_path)

t = paramiko.Transport(('182.92.219.86',22))

t.connect(username='wupeiqi',pkey=key)

sftp = paramiko.SFTPClient.from\_transport(t)

sftp.get('/tmp/test3.py','/tmp/test4.py')

t.close()

复制代码

[复制代码](javascript:void(0);)

**requests**

Python标准库中提供了：urllib等模块以供Http请求，但是，它的 API 太渣了。它是为另一个时代、另一个互联网所创建的。它需要巨量的工作，甚至包括各种方法覆盖，来完成最简单的任务。

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

import urllib.request

f = urllib.request.urlopen('http://www.webxml.com.cn//webservices/qqOnlineWebService.asmx/qqCheckOnline?qqCode=424662508')

result = f.read().decode('utf-8')

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

import urllib.request

req = urllib.request.Request('http://www.example.com/')

req.add\_header('Referer', 'http://www.python.org/')

r = urllib.request.urlopen(req)

result = f.read().decode('utf-8')

复制代码

[复制代码](javascript:void(0);)

*注：更多见Python官方文档：https://docs.python.org/3.5/library/urllib.request.html#module-urllib.request*

Requests 是使用 Apache2 Licensed 许可证的 基于Python开发的HTTP 库，其在Python内置模块的基础上进行了高度的封装，从而使得Pythoner进行网络请求时，变得美好了许多，使用Requests可以轻而易举的完成浏览器可有的任何操作。

1、安装模块

|  |  |
| --- | --- |
| 1 | pip3 install requests |

2、使用模块

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

# 1、无参数实例

import requests

ret = requests.get('https://github.com/timeline.json')

print(ret.url)

print(ret.text)

# 2、有参数实例

import requests

payload = {'key1': 'value1', 'key2': 'value2'}

ret = requests.get("http://httpbin.org/get", params=payload)

print(ret.url)

print(ret.text)

复制代码

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

# 1、基本POST实例

import requests

payload = {'key1': 'value1', 'key2': 'value2'}

ret = requests.post("http://httpbin.org/post", data=payload)

print(ret.text)

# 2、发送请求头和数据实例

import requests

import json

url = 'https://api.github.com/some/endpoint'

payload = {'some': 'data'}

headers = {'content-type': 'application/json'}

ret = requests.post(url, data=json.dumps(payload), headers=headers)

print(ret.text)

print(ret.cookies)

复制代码

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

requests.get(url, params=None, \*\*kwargs)

requests.post(url, data=None, json=None, \*\*kwargs)

requests.put(url, data=None, \*\*kwargs)

requests.head(url, \*\*kwargs)

requests.delete(url, \*\*kwargs)

requests.patch(url, data=None, \*\*kwargs)

requests.options(url, \*\*kwargs)

# 以上方法均是在此方法的基础上构建

requests.request(method, url, \*\*kwargs)

复制代码

[复制代码](javascript:void(0);)

更多requests模块相关的文档见：http://cn.python-requests.org/zh\_CN/latest/

3、Http请求和XML实例

实例：检测QQ账号是否在线

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

import urllib

import requests

from xml.etree import ElementTree as ET

# 使用内置模块urllib发送HTTP请求，或者XML格式内容

"""

f = urllib.request.urlopen('http://www.webxml.com.cn//webservices/qqOnlineWebService.asmx/qqCheckOnline?qqCode=424662508')

result = f.read().decode('utf-8')

"""

# 使用第三方模块requests发送HTTP请求，或者XML格式内容

r = requests.get('http://www.webxml.com.cn//webservices/qqOnlineWebService.asmx/qqCheckOnline?qqCode=424662508')

result = r.text

# 解析XML格式内容

node = ET.XML(result)

# 获取内容

if node.text == "Y":

print("在线")

else:

print("离线")

复制代码

[复制代码](javascript:void(0);)

实例：查看火车停靠信息

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

import urllib

import requests

from xml.etree import ElementTree as ET

# 使用内置模块urllib发送HTTP请求，或者XML格式内容

"""

f = urllib.request.urlopen('http://www.webxml.com.cn/WebServices/TrainTimeWebService.asmx/getDetailInfoByTrainCode?TrainCode=G666&UserID=')

result = f.read().decode('utf-8')

"""

# 使用第三方模块requests发送HTTP请求，或者XML格式内容

r = requests.get('http://www.webxml.com.cn/WebServices/TrainTimeWebService.asmx/getDetailInfoByTrainCode?TrainCode=G666&UserID=')

result = r.text

# 解析XML格式内容

root = ET.XML(result)

for node in root.iter('TrainDetailInfo'):

print(node.find('TrainStation').text,node.find('StartTime').text,node.tag,node.attrib)

复制代码

[复制代码](javascript:void(0);)

注：更多接口[猛击这里](http://www.cnblogs.com/wupeiqi/archive/2012/11/18/2776014.html)

**paramiko模块**

[复制代码](javascript:void(0);)

复制代码

import paramiko

# 创建SSH对象

ssh = paramiko.SSHClient()

# 允许连接不在known\_hosts文件上的主机

ssh.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())

# 连接服务器

ssh.connect(hostname="192.168.0.99", port=22, username="root", password="rootroot")

# 执行命令

stdin, stdout, stderr = ssh.exec\_command('df')

# 获取结果

result = stdout.read().decode()

# 获取错误提示（stdout、stderr只会输出其中一个）

err = stderr.read()

# 关闭连接

ssh.close()

print(stdin, result, err)

复制代码

[复制代码](javascript:void(0);)

注：如果注释“ssh.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())”这句，会报错。

https://images2017.cnblogs.com/blog/507595/201707/507595-20170727204359618-2075096304.png

类似问题可以为linux系统中~/.ssh/known\_hosts文件中的内容。

**二、实现SFTP功能**

[复制代码](javascript:void(0);)

复制代码

import paramiko

# 连接虚拟机centos上的ip及端口

transport = paramiko.Transport(("192.168.0.99", 22))

transport.connect(username="root", password="rootroot")

# 将实例化的Transport作为参数传入SFTPClient中

sftp = paramiko.SFTPClient.from\_transport(transport)

# 将“calculator.py”上传到filelist文件夹中

sftp.put('D:\python库\Python\_shell\day05\calculator.py', '/filelist/calculator.py')

# 将centos中的aaa.txt文件下载到桌面

sftp.get('/filedir/aaa.txt', r'C:\Users\duany\_000\Desktop\test\_aaa.txt')

transport.close()

复制代码

[复制代码](javascript:void(0);)

注：如果遇到Windows中路径问题，链接如下网址http://blog.csdn.net/elang6962/article/details/68068126

**三、使用秘钥实现SSH功能**

[复制代码](javascript:void(0);)

复制代码

import paramiko

private\_key = paramiko.RSAKey.from\_private\_key\_file('id\_rsa31')

# 创建SSH对象

ssh = paramiko.SSHClient()

ssh.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())

# 连接服务器

ssh.connect(hostname='192.168.79.9', port=22, username='root', pkey=private\_key)

stdin, stdout, stderr = ssh.exec\_command('ifconfig')

res\_out = stdout.read()

print(res\_out.decode())

ssh.close()

复制代码

[复制代码](javascript:void(0);)

**四、使用秘钥实现SFTP功能**

复制代码

import paramiko

private\_key = paramiko.RSAKey.from\_private\_key\_file('id\_rsa31')

# 连接虚拟机centos上的ip及端口

transport = paramiko.Transport(("192.168.79.9", 22))

transport.connect(username="root", pkey=private\_key)

# 将实例化的Transport作为参数传入SFTPClient中

sftp = paramiko.SFTPClient.from\_transport(transport)

# 将“calculator.py”上传到filelist文件夹中

sftp.put('D:\python库\Python\_shell\day05\calculator.py', '/filedir/calculator.py')

# 将centos中的aaa.txt文件下载到桌面

sftp.get('/filedir/oldtext.txt', r'C:\Users\duany\_000\Desktop\oldtext.txt')

transport.close()

## math模块

math.ceil(num) 向上取整

math.floor(num) 向下取整

math.abfs(num) 求绝对值

math.modf(num) 以元组的形式返回整数和小数部分

math.pi 圆周率

math.pow(num, num2) 幂次方 num\*\*num2

math.sqrt(num) 开平方根

**内置模块uuid/base64/hashlib/hmac/md5**

**1、**[uuid](https://so.csdn.net/so/search?q=uuid&spm=1001.2101.3001.7020)**模块**

uuid:是128为全局唯一[标识符](https://so.csdn.net/so/search?q=%E6%A0%87%E8%AF%86%E7%AC%A6&spm=1001.2101.3001.7020)，通常由32字节的字母串表示，它可以保证时间和空间的唯一性，也称为GUID  
   原理：通过mac地址、[时间戳](https://so.csdn.net/so/search?q=%E6%97%B6%E9%97%B4%E6%88%B3&spm=1001.2101.3001.7020)、命令空间、随机数、伪随机数来保证生产的ID的唯一性  
【作用】  
   随机生成字符，当成token使用，当成订单号、用户账号等  
【算法】  
(1)uuid1()基于时间戳：有mac地址，当前的时间戳，[随机数](https://so.csdn.net/so/search?q=%E9%9A%8F%E6%9C%BA%E6%95%B0&spm=1001.2101.3001.7020)，可以保证全球范围内的唯一性。但是由于mac地址使用带来的安全性，局域网中可以使用ip地址来替代。  
(2)uuid2()基于分布式计算环境DCE：  
   算法和uuid1()相同，不同是把时间戳的前四位换位posix的uid，实际中很少使用，python中没有这个函数  
(3)uuid3()基于基于名字和[md5](https://so.csdn.net/so/search?q=md5&spm=1001.2101.3001.7020)散列值：比较常用  
    通过计算名字和命令空间的md5散列值得到的，保证了同一命令空间中不同名字的唯一性，和不同命令空间的唯一性。  
    但是同一命名空间的相同名字生成相同的uuid  
(4)uuid14()基于随机数：  
    它是有伪随机数得到的，有一定重复概率，这个概率并且是可以计算出来的  
(5)uuid1()基于名字和SAH1的散列值：比较常用  
   算法原理和uuid3()算法原理相同，不同的是使用sha1算法

1. import uuid
2. # print(uuid.uuid1()) # d6647d94-a39d-11e9-8258-98fa9b100544
3. print(uuid.uuid4())
4. print(uuid.uuid3(namespace=uuid.NAMESPACE\_DNS, name="wzy")) # b234f592-cece-382b-bccb-ae56c8fb0bb8
5. print(uuid.uuid5(namespace=uuid.NAMESPACE\_DNS, name="wzy")) # 43c98da6-920f-570d-a2e8-08f890b4340c

**2、**[base64](https://so.csdn.net/so/search?q=base64&spm=1001.2101.3001.7020)**模块**

base64是一种比较常见的二进制编码方式。  
【原理】：一个包含64个字符的数组  
   ['A','B',....,'a','b',.....,'0','1',.....,'+','/']  
   对二进制数据进行处理，每三个字节一组，一组就是3x8=24bit,划为4组，每组正好64bit  
【作用】  
   适用于小段内容的编码，比如数字证书签名，cookie，网页中传输的少量二进制数据。需要注意的是base64是一种通过查表的编码方法，不能用于加密，及时修改了字符对照表也不行。

import base64

s = b'jun is good man'

*# 编码*

print(base64.b64encode(s)) *# b'anVuIGlzIGdvb2QgbWFu'*

print(base64.b64encode("jun is good man".encode("utf-8"))) *# b'anVuIGlzIGdvb2QgbWFu'*

*# 解码*

s2= b'anVuIGlzIGdvb2QgbWFu'

print(base64.b64decode(s2)) *# b'jun is good man'*

**3、hashlib模块**

import hashlib

str\_char = b"wanzhuang2019"

m5 = hashlib.md5()

m5.update(str\_char)

print(m5.hexdigest()) *# c73ac3cb6e162d37d7ca05395f797426*

*# 如果数据量比较大，可以分多次调用update，最后结果是一样的*

m6 = hashlib.md5()

m6.update(b'wanzhuang')

m6.update(b'2019')

print(m6.hexdigest()) *# c73ac3cb6e162d37d7ca05395f797426*

import requests

import hashlib

*# hash算法，对某段信息打印指纹，能够压缩信息，hash算法不可逆*

chunkSize= 4096 *# 每次读取文件的字节数*

def hashFile(file\_name):

"""对文件做hash"""

h = hashlib.sha256()

while True:

with open(file\_name, "rb") as f:

chunk = f.read(chunkSize)

if not chunk: *# 表示文件没有内容可读取了*

break

h.update(chunk)

return h.hexdigest() *# 得到文件最终hash值*

def hashStr(str):

"""对某个字符串进行hash值，str是需要被hash的字符串"""

h = hashlib.sha256()

h.update((str).encode("utf-8"))

return h.hexdigest()

def info(object):

methodList = [method for method in dir(object) if callable(getattr(object,method))]

print(methodList)

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

info(requests)

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

**4、hmac模块**

     hmac:实现HMAC算法，是用一个key对数据进行“杂凑”后再进行[hash](https://so.csdn.net/so/search?q=hash&spm=1001.2101.3001.7020)算法，使用hmac比hash算法更安全。因为不同key会产生不同的hash

import hmac

char\_str = b'wzy2003'

key = b'xiaomi'

h = hmac.new(key,char\_str,digestmod="MD5")

print(h.hexdigest()) *# 71b24094b7584dcc478e9cacdbd884ea*

**5、md5模块**

对普通数据md5

import hashlib

def get\_string\_md5():

info = "xiaozhu"

m = hashlib.md5()

m.update(info.encode("utf-8"))

return m.hexdigest()

下面是对一个文件内容进行md5，比如我们在传输过程中判断文件是否出现问题

import hashlib

def get\_file\_md5(fname:str)-> str:

"""

:param fname: 文件名称

:return:

"""

hash\_md5 = hashlib.md5()

with open(fname,"rb") as f:

for chunk in iter(lambda :f.read(4096),b""):

hash\_md5.update(chunk)

return hash\_md5.hexdigest()

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

m\_value = get\_file\_md5(r"D:\迅雷下载\odoo\_13.0.20191002 - 副本.exe")

print(m\_value) *#02ffe90c7100bb9a88f6d98bf71a6e91*

**shelve模块**

shelve模块比pickle模块简单，只有一个open函数，返回类似字典的对象，可读可写;key必须为字符串，而值可以是python所支持的数据类型

import shelve

f = shelve.open(r'shelve) # 目的：将一个字典放入文本中

f['stu1\_info']={'name':'rxz','age':'29'}

f['stu2\_info']={'name':'gsh','age':'3'}

f['school\_info']={'website':'tongcheng.com','city':'suzhou'}

f.close()

print(f.get('stu\_info')['age'])

**re模块**

就其本质而言，正则表达式（或 RE）是一种小型的、高度专业化的编程语言，（在Python中）它内嵌在Python中，并通过 re 模块实现。正则表达式模式被编译成一系列的字节码，然后由用 C 编写的匹配引擎执行。

**字符匹配（普通字符，元字符）：**

   1 普通字符：大多数字符和字母都会和自身匹配  
              >>> re.findall('alvin','yuanaleSxalexwupeiqi')  
                      ['alvin']

   2 元字符：. ^ $ \* + ? { } [ ] | ( ) \

import re

*# . 就是通配符,一个点代表一个字符*

s = re.findall("a...x","adsfaeyuxslg")

print(s) *# ['aeyux']*

*# ^ 这个代表以什么开头*

s = re.findall("^a..x","adsfaeyuxslg")

print(s) *# ['']*

s = re.findall("a..x","adsxaeyxslg")

print(s) *# ['adsx', 'aeyx']*

*# $ 以什么结尾的*

s = re.findall("a..x$","adsxaeyxslg")

print(s) *#[]*

a = re.findall("a..x$","adsxaeyxslgarrx")

print(a) *#['arrx']*

*# 重复符号有4个，第一个（\* ）代表0到无穷次,贪婪匹配*

s = re.findall("d\*","sfdderddddddyuyygdd")

print(s) *#['', '', 'dd', '', '', 'dddddd', '', '', '', '', '', 'dd', '']*

*# 重复符号有4个，第二个（+ ）代表1 到无穷次,贪婪匹配*

s = re.findall("alex+","asdhfalexxx")

print(s) *#['alexxx']*

s = re.findall("alex\*","asdhfalexxx")

print(s) *#['alexxx']*

a = re.findall("alex+","asdhfale")

print(a) *#[]*

a = re.findall("alex\*","asdhfale")

print(a) *#['ale']*

*# 重复符号有4个，第三个（？）代表0 到1*

s = re.findall("alex?","asdhfalexxx")

print(s) *#['alex']*

s = re.findall("alex?","asdhfale")

print(s) *#['ale']*

*# 重复符号有4个，第四个（{}）代表0 到你想到取得次数*

"""

{0,}=====>\*

{1,}=====>+

{0,1}=====>?

{6}=====>重复6次

{1，6}=====>重复,1,2,3,4,5,6其中任何一次

"""

s = re.findall("alex{6}","asdhfalexxxxxx")

print(s) *#['alexxxxxx']*

s = re.findall("alex{0,6}","asdhfalexx")

print(s) *#['alexx']*

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

   元字符之. ^ $ \* + ? { }

import re

ret=re.findall('a..in','helloalvin')

print(ret)*#['alvin']*

ret=re.findall('^a...n','alvinhelloawwwn')

print(ret)*#['alvin']*

ret=re.findall('a...n$','alvinhelloawwwn')

print(ret)*#['awwwn']*

ret=re.findall('a...n$','alvinhelloawwwn')

print(ret)*#['awwwn']*

ret=re.findall('abc\*','abcccc')*#贪婪匹配[0,+oo]*

print(ret)*#['abcccc']*

ret=re.findall('abc+','abccc')*#[1,+oo]*

print(ret)*#['abccc']*

ret=re.findall('abc?','abccc')*#[0,1]*

print(ret)*#['abc']*

ret=re.findall('abc{1,4}','abccc')

print(ret)*#['abccc'] 贪婪匹配*

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

**注意：前面的\*,+,?等都是贪婪匹配，也就是尽可能匹配，后面加?号使其变成惰性匹配**

1. *#--------------------------------------------字符集[]*

ret=re.findall('a[bc]d','acd')

print(ret)*#['acd']*

ret=re.findall('[a-z]','acd')

print(ret)*#['a', 'c', 'd']*

ret=re.findall('[.\*+]','a.cd+')

print(ret)*#['.', '+']*

*#在字符集里有功能的符号: - ^ \*

ret=re.findall('[1-9]','45dha3')

print(ret)*#['4', '5', '3']*

ret=re.findall('[^ab]','45bdha3')

print(ret)*#['4', '5', 'd', 'h', '3']*

ret=re.findall('[\d]','45bdha3')

print(ret)*#['4', '5', '3']*

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

元字符之转义符\

反斜杠后边跟元字符去除特殊功能,比如\.  
反斜杠后边跟普通字符实现特殊功能,比如\d

\d  匹配任何十进制数；它相当于类 [0-9]。  
\D 匹配任何非数字字符；它相当于类 [^0-9]。  
\s  匹配任何空白字符；它相当于类 [ \t\n\r\f\v]。  
\S 匹配任何非空白字符；它相当于类 [^ \t\n\r\f\v]。  
\w 匹配任何字母数字字符；它相当于类 [a-zA-Z0-9\_]。  
\W 匹配任何非字母数字字符；它相当于类 [^a-zA-Z0-9\_]  
\b  匹配一个特殊字符边界，比如空格 ，&，＃等

import re

ret=re.findall('I\b','I am LIST')

print(ret)*#[]*

ret=re.findall(r'I\b','I am LIST')

print(ret)*#['I']*

现在我们聊一聊\,先看下面两个匹配：

*#-----------------------------eg1:*

import re

ret=re.findall('c\l','abc\le')

print(ret)*#[]*

ret=re.findall('c\\l','abc\le')

print(ret)*#[]*

ret=re.findall('c\\\\l','abc\le')

print(ret)*#['c\\l']*

ret=re.findall(r'c\\l','abc\le')

print(ret)*#['c\\l']*

*#-----------------------------eg2:*

*#之所以选择\b是因为\b在ASCII表中是有意义的*

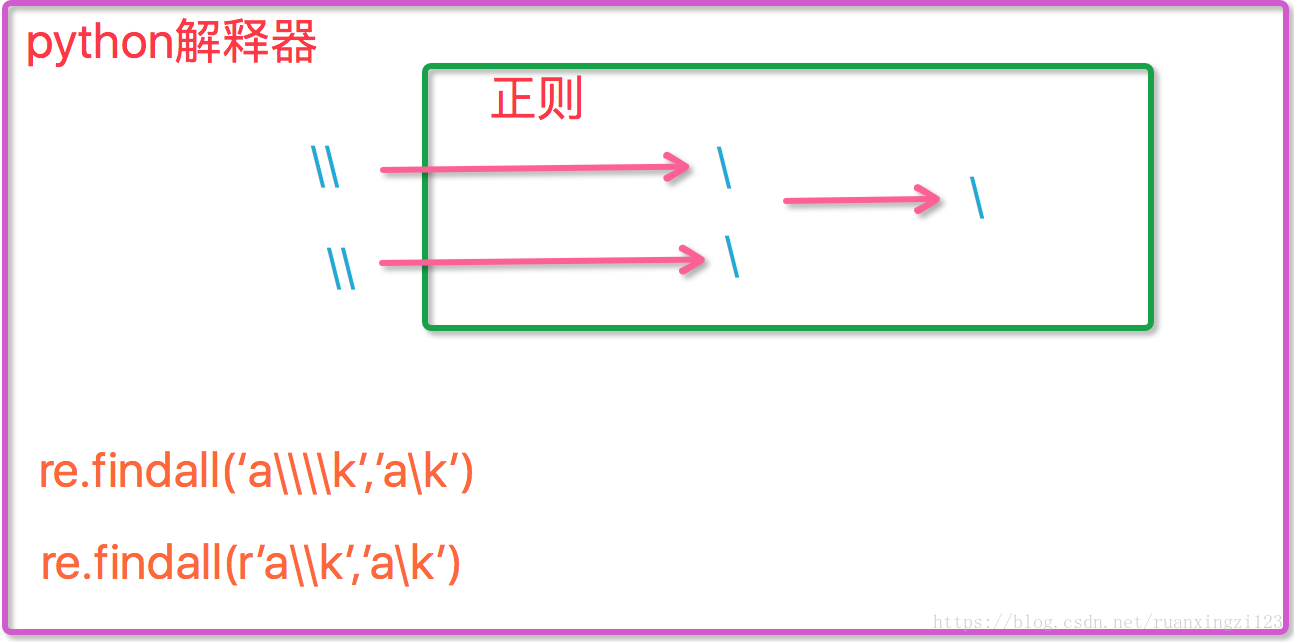
m = re.findall('\bblow', 'blow')

print(m)

m = re.findall(r'\bblow', 'blow')

print(m)

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png



**元字符之分组()**

import re

m = re.findall(r'(ad)+', 'add')

print(m)

ret=re.search('(?P<id>\d{2})/(?P<name>\w{3})','23/com')

print(ret.group())*#23/com*

print(ret.group('id'))*#23*

**元字符之｜**

import re

ret = re.findall(r"kr|z","sdfkrsf")

print(ret)*#['kr']*

ret = re.findall(r"kr|z","sdfkrzsf")

print(ret)*#['kr', 'z']*

import re

ret=re.search('(ab)|\d','rabhdg8sd')

print(ret.group())*#ab*

**re模块下的常用方法**

import re

*#1*

re.findall('a','alvin yuan') *#返回所有满足匹配条件的结果,放在列表里*

*#2*

re.search('a','alvin yuan').group() *#函数会在字符串内查找模式匹配,只到找到第一个匹配然后返回一个包含匹配信息的对象,该对象可以*

*# 通过调用group()方法得到匹配的字符串,如果字符串没有匹配，则返回None。*

*#3*

re.match('a','abc').group() *#同search,不过尽在字符串开始处进行匹配*

*#4*

ret=re.split('[ab]','abcd') *#先按'a'分割得到''和'bcd',在对''和'bcd'分别按'b'分割*

print(ret)*#['', '', 'cd']*

*#5*

ret=re.sub('\d','abc','alvin5yuan6',1)

print(ret)*#alvinabcyuan6*

ret=re.subn('\d','abc','alvin5yuan6')

print(ret)*#('alvinabcyuanabc', 2)*

*#6*

obj=re.compile('\d{3}')

ret=obj.search('abc123eeee')

print(ret.group())*#123*

import re

ret=re.finditer('\d','ds3sy4784a')

print(ret) *#<callable\_iterator object at 0x10195f940>*

print(next(ret).group())

print(next(ret).group())

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

 注意：

import re

ret=re.findall('www.(baidu|oldboy).com','www.oldboy.com')

print(ret)*#['oldboy'] 这是因为findall会优先把匹配结果组里内容返回,如果想要匹配结果,取消权限即可*

ret=re.findall('www.(?:baidu|oldboy).com','www.oldboy.com')

print(ret)*#['www.oldboy.com']*

import re

print(re.findall("<(?P<tag\_name>\w+)>\w+</(?P=tag\_name)>","<h1>hello</h1>"))

print(re.search("<(?P<tag\_name>\w+)>\w+</(?P=tag\_name)>","<h1>hello</h1>"))

print(re.search(r"<(\w+)>\w+</\1>","<h1>hello</h1>"))

*#匹配出所有的整数*

import re

*#ret=re.findall(r"\d+{0}]","1-2\*(60+(-40.35/5)-(-4\*3))")*

ret=re.findall(r"-?\d+\.\d\*|(-?\d+)","1-2\*(60+(-40.35/5)-(-4\*3))")

ret.remove("")

print(ret)

# timeit模块

Python3中的timeit模块可以用来测试小段代码的运行时间

其中主要通过两个函数来实现：timeit和[repeat](https://so.csdn.net/so/search?q=repeat&spm=1001.2101.3001.7020)，代码如下：

def timeit(stmt="pass", setup="pass", timer=default\_timer,

number=default\_number, globals=None):

"""Convenience function to create Timer object and call timeit method."""

return Timer(stmt, setup, timer, globals).timeit(number)

def repeat(stmt="pass", setup="pass", timer=default\_timer,

repeat=default\_repeat, number=default\_number, globals=None):

"""Convenience function to create Timer object and call repeat method."""

return Timer(stmt, setup, timer, globals).repeat(repeat, number)

在上面的代码中可见，无论是timeit还是repeat都是先生成Timer对象，然后调用了Timer对象的timeit或repeat函数。

在使用timeit模块时，可以直接使用timeit.timeit()、tiemit.repeat()，还可以先用timeit.Timer()来生成一个Timer对象，然后再用TImer对象用timeit()和repeat()函数，后者再灵活一些。

上述两个函数的入参：

* stmt：用于传入要测试时间的代码，可以直接接受字符串的表达式，也可以接受单个变量，也可以接受函数。传入函数时要把函数申明在当前文件中，然后在 stmt = func() 执行函数，然后使用setup = from \_\_main\_\_ import func
* setup：传入stmt的运行环境，比如stmt中使用到的参数、变量，要导入的模块等。可以写一行语句，也可以写多行语句，写多行语句时要用分号；隔开语句。
* number：要测试的代码的运行次数，默认100000次，对于耗时的代码，运行太多次会比较慢，此时建议自己修改一下运行次数
* repeat：指测试要重复几次，每次的结果构成列表返回，默认3次。

**一、直接使用timeit.timeit()、tiemit.repeat()：**

import timeit

print(timeit.timeit(stmt= 'list(i\*\*2 for i in normal\_list)',setup = 'normal\_list=range(10000)',number=10))

#0.3437936799875755

print(timeit.repeat(stmt= 'list(i\*\*2 for i in normal\_list)', setup='normal\_list=range(10000)',repeat=2,number=10))

#[0.33649995761778984, 0.3394490767789293]

#setup 为复合语句

print(timeit.timeit(stmt= 'list(i\*\*2 for i in normal\_list)',setup = 'a=10000;normal\_list=range(a)',number=10))

#0.33272367424748817

print(timeit.repeat(stmt= 'list(i\*\*2 for i in normal\_list)', setup='a=10000;normal\_list=range(a)',repeat=2,number=10))

#[0.3323106610316342, 0.3356380911962764]

def func():

normal\_list=range(10000)

L = [i\*\*2 for i in normal\_list]

#stmt为函数

print(timeit.timeit("func()", setup="from \_\_main\_\_ import func",number=10))

#0.12436874684622312

print(timeit.repeat("func()", setup="from \_\_main\_\_ import func",repeat=2,number=10))

#[0.12142133435126468, 0.12079555675148601]

直接用函数的方式，速度更快。

**二、先生成Timer，再调用timeit()、repeat()：**

'''

遇到问题没人解答？小编创建了一个Python学习交流QQ群：778463939

寻找有志同道合的小伙伴，互帮互助,群里还有不错的视频学习教程和PDF电子书！

'''

import timeit

#生成timer

timer1 = timeit.Timer(stmt= 'list(i\*\*2 for i in normal\_list)',setup = 'normal\_list=range(10000)')

#调用timeit和repeat时还传number和repeat参数

print(timer1.timeit(number=10))

#0.34721554568091145

print(timer1.repeat(repeat=2,number=10))

#[0.3391925079630199, 0.34103400077255097]

#setup 为复合语句

timer1 = timeit.Timer(stmt= 'list(i\*\*2 for i in normal\_list)',setup = 'a=10000;normal\_list=range(a)')

print(timer1.timeit(number=10))

0.34383463997592467

print(timer1.repeat(repeat=2,number=10))

#[0.34573984832288773, 0.34413273766891006]

#stmt为函数

def func():

normal\_list=range(10000)

L = [i\*\*2 for i in normal\_list]

timer1 = timeit.Timer("func()", setup="from \_\_main\_\_ import func")

print(timer1.timeit(number=10))

#0.1223264363160359

print(timer1.repeat(repeat=2,number=10))

#[0.12266321844246209, 0.1264150395975001]

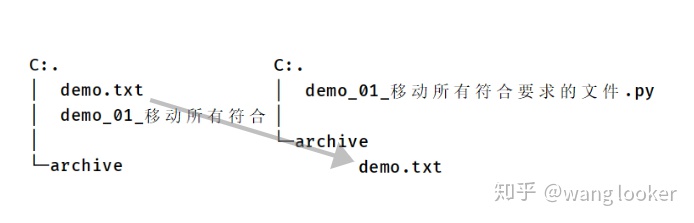
# Pathlib

之前如果要使用 python 操作文件路径，我总是会条件反射导入 os.path。 而现在，我会更加喜欢用新式的 pathlib, 虽然用得还是没有 os.path 熟练，但是以后会坚持使用。

pathlib 库从 python3.4 开始，到 python3.6 已经比较成熟。如果你的新项目可以直接用 3.6 以上，建议用 pathlib。相比于老式的 os.path 有几个优势：

1. 老的路径操作函数管理比较混乱，有的是导入 os, 有的又是在 os.path 当中，而新的用法统一可以用 pathlib 管理。
2. 老用法在处理不同操作系统 win，mac 以及 linux 之间很吃力。换了操作系统常常要改代码，还经常需要进行一些额外操作。
3. 老用法主要是函数形式，返回的数据类型通常是字符串。但是路径和字符串并不等价，所以在使用 os 操作路径的时候常常还要引入其他类库协助操作。新用法是面向对象，处理起来更灵活方便。
4. pathlib 简化了很多操作，用起来更轻松。

举个例子， 把所有的 txt 文本全部移动到 archive 目录当中（archive 目录必须存在)。



使用原来的用法：

import glob

import os

import shutil

# 获取运行目录下所有的 txt 文件。注意：不是这个文件目录下

print(glob.glob('\*.txt'))

for file\_name in glob.glob('\*.txt'):

new\_path = os.path.join('archive', file\_name)

shutil.move(file\_name, new\_path)

新的写法：

from pathlib import Path

Path("demo.txt").replace('archive/demo.txt')

这篇文章列举了 pathlib 的主要用法，细节知识有点多，我只会介绍 pathlib 的用法，不会每个都列举和 os 的区别。但是我会在文章最后放上主要操作使用 pathlib 和 os 的对比图。

最主要的路径操作，你可以参考国外一位大神总结的图：



**1 路径获取**

* 获取当前工作目录

>>> import pathlib

>>> pathlib.Path.cwd()

C:\Users\me\study

# WindowsPath('C:\Users\me\study')

虽然在这里打印出来的很像一个字符串，但实际上得到的是一个 WindowsPath('C:\Users\me\study')对象。显示内容由 Path 类的 \_\_repr\_\_ 定义。

注意  
工作目录是在哪个目录下运行你的程序，不是项目目录。

如果你只想得到字符串表示，不想要 WindowsPath 对象，可以用 str() 转化：

>>> str(pathlib.Path.cwd())

C:\Users\me\study

* 获取用户 home 目录。

下面的例子因为基本都是使用 pathlib 下面的 Path 类，所以可以换一种导入方式。

from pathlib import Path

>>> Path.home()

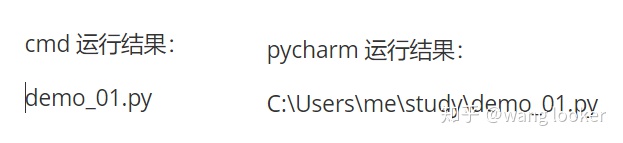
c:\Users\me

* 获取当前文件路径

>>> Path(\_\_file\_\_)

demo\_01.py

在 pycharm 中右击运行和在 cmd 运行的结果会不同。pycharm 会显示全路径，cmd 运行只会显示工作目录下的相对路径。如果想统一，可以添加后缀 .resolve() 转化成绝对路径，这个在后面还会提到。



* 获取任意字符串路径

>>> Path('subdir/demo\_02.py')

subdir\demo\_02.py

>>> Path('c:d:y/rad.txt')

c:d:y\rad.txt

这里需要注意 2 点：

1. 不管字符串使用的是正斜杠 / 还是反斜杠 \， 在 windows 系统里，得到的路径都是反斜杠\, pathlib 会根据操作系统智能处理。
2. 第二个例子中字符串会被 / 分割，c:d:y 会被当做一个目录名字，pathlib 不会去判断这个文件真的存在哦。

* 获取绝对路径

只需要在任意路径对象后添加方法 .resolve() 就能获取路径的绝对路径。如果填入的路径是相对路径（windows 下没有盘符，linux 没有 / 开头)，则会在当前工作目录后添加路径。如果是已经是绝对路径，则只会根据操作系统优化表达。

>>> file = Path('archive/demo.txt')

>>> file

archive\demo.txt

>>> file.resolve()

C:\Users\me\study\archive\demo.txt

* 获取文件属性

文件属性比如文件大小，创建时间，修改时间等等。

file = Path('archive/demo.txt')

print(file.stat())

print(file.stat().st\_size)

print(file.stat().st\_atime)

print(file.stat().st\_ctime)

print(file.stat().st\_mtime)

找出最后修改的文件的例子：

>>> path = Path.cwd()

>>> max(

[(f.stat().st\_mtime, f)

for f in path.iterdir()

if f.is\_file()]

)

1589171135.860173 C:\Users\me\study\demo\_03.py

**2 路径组成部分**

获取路径的组成部分非常方便：

* .name 文件名，包含后缀名，如果是目录则获取目录名。
* .stem 文件名，不包含后缀。
* .suffix 后缀，比如 .txt, .png。
* .parent 父级目录，相当于 cd ..
* .anchor 锚，目录前面的部分 C:\ 或者 /。

>>> file = Path('archive/demo.txt')

>>> file.name

demo.txt

>>> file.stem

demo

>>> file.suffix

.txt

>>> file.parent

C:\Users\me\study\archive

>>> file.anchor

'C:\'

* 获取上一级目录

>>> file = Path('archive/demo.txt')

>>> file.parent

archive

获取上 2 级 和 3 级目录会有点问题。因为传入的是一个相对路径，上 3 级已经无法往上了，所以还是会停留在工作目录。所以你需要确保你的路径有这么多层级。

>>> file.parent.parent

.

>>> file.parent.parent.parent

.

* 获取所有的上级目录：

>>> file.parents

<WindowsPath.parents>

>>> list(file.parents)

[WindowsPath('archive'), WindowsPath('.')]

* 父级目录的另一种表示方法：

>>> file.parents[0]

archive

如果路径是在当前工作目录下的子目录，最好转化成绝对路径再获取上层目录：

>>> file.resolve().parents[4]

C:\Users

# 不能使用负数

在 os 模块中获取上 3 级目录简直令人奔溃，需要重复使用 dirname 函数，使用 pathlib 的 parent 可以极大简单化操作：

>>> import os

>>> os.path.dirname(file)

* 相对其他某个路径的结果：

>>> file.relative\_to('archive')

dmeo.txt

**3 子路径扫描**

* dir\_path.iterdir() 可以扫描某个目录下的所有路径（文件和子目录)， 打印的会是处理过的绝对路径。

>>> cwd = Path.cwd()

>>> [path for path in cwd.iterdir() if cwd.is\_dir()]

[

WindowsPath('C:/Users/me/study/archive'),

WindowsPath('C:/Users/me/study/demo\_01.py'),

WindowsPath('C:/Users/me/study/new\_archive')

]

* 使用 iterdir() 可以统计目录下的不同文件类型：

>>> path = Path.cwd()

>>> files = [f.suffix for f in path.iterdir() if f.is\_file()]

>>> collections.Counter(files)

Counter({'.py': 3, '.txt': 1})

* 查找目录下的指定文件 glob 。

使用模式匹配（正则表达式）匹配指定的路径。正则表达式不熟练的可以查看 [**这个教程，真的让我学会了正则表达式**](https://link.zhihu.com/?target=https%3A//mp.weixin.qq.com/s%3Fsrc%3D11%26timestamp%3D1589010579%26ver%3D2327%26signature%3DJ84Vp3En0%2AOyHZ5OJEkl4xjhX3gXUDTEn2jxYy%2AheATYN%2AIOB66Vnvj8R9Cu9D6BVB%2AQcAP4NtxQne2WVDOxb0e9lUFISXOtzT-8dhAqbUvHVZ69%2AyGAvJJ2vzWhJMfA%26new%3D1)。glob 只会匹配当前目录下， rglob 会递归所有子目录。下面这个例子，demo.txt 在 archive 子目录下。所以用 glob 找到的是空列表，rglob 可以找到。glob 得到的是一个生成器，可以通过 list() 转化成列表。

>>> cwd = Path.cwd()

>>> list(cwd.glob('\*.txt'))

[]

>>> list(cwd.rglob('\*.txt'))

[WindowsPath('C:/Users/me/study/archive/demo.txt')]

* 检查路径是否符合规则 match

>>> file = Path('/archive/demo.txt')

>>> file.match('\*.txt')

True

**4 路径拼接**

pathlib 支持用 / 拼接路径。熟悉魔术方法的同学应该很容易理解其中的原理。

>>> Path.home() / 'dir' / 'file.txt'

C:\Users\me\dir\file.txt

如果用不惯 / ，也可以用类似 os.path.join 的方法：

>>> Path.home().joinpath('dir', 'file.txt')

C:\Users\me\dir\file.txt

**5 路径测试（判断）**

* 是否为文件

>>> Path("archive/demo.txt").is\_file()

True

>>> Path("archive/demo.txt").parent.is\_file()

False

* 是否为文件夹 (目录)

>>> Path("archive/demo.txt").is\_dir()

False

>>> Path("archive/demo.txt").parent.is\_dir()

True

* 是否存在

>>> Path("archive/demo.txt").exists

True

>>> Path("archive/dem.txt").exists

False

**6 文件操作**

* 创建文件 touch

>>> file = Path('hello.txt')

>>> file.touch(exist\_ok=True)

None

>>> file.touch(exist\_ok=False)

FileExistsError: [Errno 17] File exists: 'hello.txt'

exist\_ok 表示当文件已经存在时，程序的反应。如果为 True，文件存在时，不进行任何操作。如果为 False, 则会报 FileExistsError 错误。

* 创建目录 path.mkdir

用 os 创建目录分为 2 个函数 ： mkdir() 和 makedirs()。 mkdir() 一次只能创建一级目录， makedirs() 可以同时创建多级目录：

>>> os.mkdir('dir/subdir/3dir')

FileNotFoundError: [WinError 3] 系统找不到指定的路径。: 'dir/subdir/3dir'

>>> os.makedirs('dir/subdir/3dir')

None

使用 pathlib 只需要用 path.mkdir() 函数就可以。它提供了 parents 参数，设置为 True 可以创建多级目录；不设置则只能创建 一层：

>>> path = Path('/dir/subdir/3dir')

>>> path.mkdir()

FileNotFoundError: [WinError 3] 系统找不到指定的路径。: 'dir/subdir/3dir'

>>> path.mkdir(parents=True)

None

* 删除目录 path.rmdir()

删除目录非常危险，并且没有提示，一定要谨慎操作。一次只删除一级目录，且当前目录必须为空。

>>> path = Path('dir/subdir/3dir')

>>> path.rmdir()

None

* 删除文件 path.unlink, 危险操作。

>>> path = Path('archive/demo.txt')

>>> path.unlink()

* 打开文件

使用 open() 函数打开文件时，如果需要传入文件路径。可以用字符串作为参数传入：

with open('archive/demo.txt') as f:

print(f.read())

也可以传入 Path 对象：

file\_path = Path('archive/demo.txt')

with open(file\_path) as f:

print(f.read())

如果经常使用 pathlib，可以在获取到 Path 路径以后直接调用 path.open() 方法。至于到底用哪一个，其实不必太在意，因为 path.open() 也是调用内置函数 open()。

file = Path('archive/demo.txt')

with file.open() as f:

print(f.read())

不过 pathlib 对读取和写入进行了简单的封装，不再需要重复去打开文件和管理文件的关闭了。

* .read\_text() 读取文本
* .read\_bytes() 读取 bytes
* .write\_text() 写入文本
* .write\_bytes() 写入 tytes

>>> file\_path = Path('archive/demo.txt')

>>> file\_path.read\_text() # 读取文本

'text in the demo.txt'

>>> file\_path.read\_bytes() # 读取 bytes

b'text in the demo.txt'

>>> file.write\_text('new words') # 写入文本

9

>>> file.write\_bytes(b'new words') # 写入 bytes

9

注意  
file.write 操作使用的是 w 模式，如果之前已经有文件内容，将会被覆盖。

* 移动文件

txt\_path = Path('archive/demo.txt')

res = txt\_path.replace('new\_demo.txt')

print(res)

这个操作会把 archive 目录下的 demo.txt 文件移动到当前工作目录，并重命名为 new\_demo.txt。

移动操作支持的功能很受限。比如当前工作目录如果已经有一个 new\_demo.txt 的文件，则里面的内容都会被覆盖。还有，如果需要移动到其他目录下，则该目录必须要存在，否则会报错：

# new\_archive 目录必须存在，否则会报错

txt\_path = Path('archive/demo.txt')

res = txt\_path.replace('new\_archive/new\_demo.txt')

print(res)

为了避免出现同名文件里的内容被覆盖，通常需要进行额外处理。比如判断同名文件不能存在，但是父级目录必须存在；或者判断父级目录不存在时，创建该目录。

dest = Path('new\_demo.txt')

if (not dest.exists()) and dest.parent.exists():

txt\_path.replace(dest)

* 重命名文件

txt\_path = Path('archive/demo.txt')

new\_file = txt\_path.with\_name('new.txt')

txt\_path.replace(new\_file)

* 修改后缀名

txt\_path = Path('archive/demo.txt')

new\_file = txt\_path.with\_suffix('.json')

txt\_path.replace(new\_file)

注意  
不管是移动文件还是删除文件，都不会给任何提示。所以在进行此类操作的时候要特别小心。  
对文件进行操作最好还是用 shutil 模块。

# Python中的itertools模块

## itertools模块的介绍

在Python中，迭代器（Iterator）是常用来做惰性序列的对象，只有当迭代到某个值的时候，才会进行计算得出这个值。因此，迭代器可以用来存储无限大的序列，这样我们就不用把他一次性放在内存中，而只在需要的时候进行计算。所以，对于读取大文件或者无线集合，最好是使用迭代器。实际上，Python2的大多数函数都是返回列表等序列，而Python3都已经改进为返回迭代器。

Python的内置模块itertools就是用来操作迭代器的一个模块，包含的函数都是能够创建迭代器来用于for循环或者next()。其中函数主要可以分为三类，分别是无限迭代器，有限迭代器，组合迭代器。

## 无限迭代器（Infinite Iterators）

这些函数可以生成无限的迭代器，我们主要学习以下三个函数的用法。

**count([start=0, step=1])** 接收两个可选整形参数，第一个指定了迭代开始的值，第二个指定了迭代的步长。此外，start参数默认为0，step参数默认为1，可以根据需要来把这两个指定为其它值，或者使用默认参数。

**import** itertools

**for** i **in** itertools**.**count(10,2):

**print**(i)

**if** i**>**20:

**break**

[Running] python **-**u "e:\pythonee\code\test.py"

10

12

14

16

18

20

22

**cycle(iterable)** 是用一个可迭代对象中的元素来创建一个迭代器，并且复制自己的值，一直无限的重复下去。

**import** itertools

**for** i **in** itertools**.**cycle("abcd"):

**print**(i) *# 具有无限的输出，可以按ctrl+c来停止。*

[Running] python **-**u "e:\pythonee\code\test.py"

a

b

c

d

a

b

c

d

a

b

**repeat(elem [,n])**是将一个元素重复n遍或者无穷多遍，并返回一个迭代器。

**import** itertools

**for** i **in** itertools**.**repeat("abcd",5):

**print**(i)

[Running] python **-**u "e:\pythonee\code\test.py"

abcd

abcd

abcd

abcd

abcd

## 组合迭代器（Combinatoric Iterators）

组合操作包括排列，笛卡儿积，或者一些离散元素的选择，组合迭代器就是产生这样序列的迭代器。我们来看看这几个函数的用法。

**product(\*iterables, repeat=1)** 得到的是可迭代对象的笛卡儿积，\*iterables参数表示需要多个可迭代对象。这些可迭代对象之间的笛卡儿积，也可以使用for循环来实现，例如 **product(A, B)** 与 **((x,y) for x in A for y in B)**就实现一样的功能。

**import** itertools

**for** i **in** itertools**.**product([1,2,3],[4,5,6]):

**print**(i)

[Running] python **-**u "e:\pythonee\code\test.py"

(1, 4)

(1, 5)

(1, 6)

(2, 4)

(2, 5)

(2, 6)

(3, 4)

(3, 5)

(3, 6)

而 repeat 参数则表示这些可迭代序列重复的次数。例如 **product(A, repeat=4)** 与 **product(A, A, A, A)**实现的功能一样。

**import** itertools

**for** i **in** itertools**.**product('ab','cd',repeat **=** 2):

**print**(i)

[Running] python **-**u "e:\pythonee\code\test.py"

('a', 'c', 'a', 'c')

('a', 'c', 'a', 'd')

('a', 'c', 'b', 'c')

('a', 'c', 'b', 'd')

('a', 'd', 'a', 'c')

('a', 'd', 'a', 'd')

('a', 'd', 'b', 'c')

('a', 'd', 'b', 'd')

('b', 'c', 'a', 'c')

('b', 'c', 'a', 'd')

('b', 'c', 'b', 'c')

('b', 'c', 'b', 'd')

('b', 'd', 'a', 'c')

('b', 'd', 'a', 'd')

('b', 'd', 'b', 'c')

('b', 'd', 'b', 'd')

**permutations(iterable,r=None)**返回的是可迭代元素中的一个排列组合，并且是按顺序返回的，且不包含重复的结果。

**import** itertools

**for** i **in** itertools**.**permutations('abc'):

**print**(i)

[Running] python **-**u "e:\pythonee\code\test.py"

('a', 'b', 'c')

('a', 'c', 'b')

('b', 'a', 'c')

('b', 'c', 'a')

('c', 'a', 'b')

('c', 'b', 'a')

当然，第 2 个参数默认为None，它表示的是返回元组（tuple) 的长度，我们来尝试一下传入第二个参数。

**import** itertools

**for** i **in** itertools**.**permutations('abc',2):

**print**(i)

[Running] python **-**u "e:\pythonee\code\test.py"

('a', 'b')

('a', 'c')

('b', 'a')

('b', 'c')

('c', 'a')

('c', 'b')

**combinations(iterable,r)** 返回的是可迭代对象所有的长度为 r 的子序列，注意这与前一个函数 **permutation** 不同，**permutation** 返回的是排列，而 **combinations** 返回的是组合。

**import** itertools

**for** i **in** itertools**.**combinations('1234',2):

**print**(i)

[Running] python **-**u "e:\pythonee\code\test.py"

('1', '2')

('1', '3')

('1', '4')

('2', '3')

('2', '4')

('3', '4')

**combinations\_with\_replacement(iterable, r)** 返回一个可与自身重复的元素组合，用法类似于 **combinations** 。

**import** itertools

**for** i **in** itertools**.**combinations\_with\_replacement('1234',2):

**print**(i)

[Running] python **-**u "e:\pythonee\code\test.py"

('1', '1')

('1', '2')

('1', '3')

('1', '4')

('2', '2')

('2', '3')

('2', '4')

('3', '3')

('3', '4')

('4', '4')

## 有限迭代器（Iterators Terminating on the Shortest Input Sequence）

这里的函数有十来个，主要为大家介绍其中几个常用的函数。

**chain(\*iterables)** 可以把多个可迭代对象组合起来，形成一个更大的迭代器。

**import** itertools

**for** i **in** itertools**.**chain('good','bye'):

**print**(i)

[Running] python **-**u "e:\pythonee\code\test.py"

g

o

o

d

b

y

e

**groupby(iterable,key=None)** 可以把相邻元素按照 key 函数分组，并返回相应的 key 和 groupby，如果key函数为 None，则只有相同的元素才能放在一组。

**import** itertools

**for** key, group **in** itertools**.**groupby('AaaBBbcCAAa', **lambda** c: c**.**upper()):

**print**(list(group))

[Running] python **-**u "e:\pythonee\code\test.py"

['A', 'a', 'a']

['B', 'B', 'b']

['c', 'C']

['A', 'A', 'a']

**accumulate(iterable [,func])** 可以计算出一个迭代器，这个迭代器是由特定的二元函数的累计结果生成的，如果不指定的话，默认函数为求和函数。

**import** itertools

**for** i **in** itertools**.**accumulate([0,1,0,1,1,2,3,5]):

**print**(i)

[Running] python **-**u "e:\pythonee\code\test.py"

0

1

1

2

3

5

8

13

如果我们指定这个累计函数，则还能有不同的用法，例如，指定一个最大值函数，或者自己定义的函数。

**import** itertools

**for** i **in** itertools**.**accumulate([2,1,4,3,5],max):

**print**(i)

[Running] python **-**u "e:\pythonee\code\test.py"

2

2

4

4

5

# optparse模块

Python 有两个内建的模块用于处理命令行参数：

一个是 getopt，getopt只能简单处理 命令行参数。

另一个是 optparse，是一个能够让程式设计人员轻松设计出简单明了、易于使用、符合标准的Unix命令列程式的Python模块。生成使用和帮助信息。

下面是一个简单的示例脚本optparse\_exampl\_1.py：

[root@localhost python]# vim optparse\_exampl\_1.py

#!/usr/bin/env python

from optparse import OptionParser

parser = OptionParser()

parser.add\_option("-f", "--file", dest="filename",

help="write report to FILE", metavar="FILE")

parser.add\_option("-q", "--quiet",

action="store\_false", dest="verbose", default=True,

help="don't print status messages to stdout")

(options, args) = parser.parse\_args()

现在你可以在命令行进行如下输入：

<yourscript> --file=outfile -q

<yourscript> -f outfile --quiet

<yourscript> --quiet --file outfile

<yourscript> -q -foutfile

<yourscript> -qfoutfile

上面这些命令是相同效果的。除此之外， optparse 还为我们自动生成命令行的帮助信息：

<yourscript> -h

<yourscript> --help

optparse将打印脚本的选项和帮助信息：

[root@localhost python]# ./optparse\_exampl\_1.py -h

Usage: optparse\_exampl\_1.py [options]

### optparse的简单用法如下：

[复制代码](javascript:void(0);)

复制代码

import optparse

parser = optparse.OptionParser()#创建optparse对象parser

parser.add\_option("-s","--server", dest="server", help="ftp server ip\_address")#使用add\_option()来定义命令行参数

parser.add\_option("-P","--port", type="int", dest="port", help="ftp server port")

parser.add\_option("-u","--username", dest="username", help="username info")

parser.add\_option("-p","--password", dest="password", help="password info")

fakeArgs=['-f',"file.txt",'-v','good luck to you','arg2','arg']#用一个数组模拟命令参数

options,args=optParser.parse\_args(fakeArgs)#使用parse\_args()来解析命令行。

复制代码

[复制代码](javascript:void(0);)

add\_option函数的参数:

dest：用于保存输入的临时变量，其值通过options的属性进行访问，存储的内容是dest之前输入的参数，多个参数用逗号分隔  
type: 用于检查命令行参数传入的参数的数据类型是否符合要求，有 string，int，float 等类型  
help：用于生成帮助信息  
default: 给dest的默认值，如果用户没有在命令行参数给dest分配值，则使用默认值

### optparse的简单深入解析

#### 新建对象

[复制代码](javascript:void(0);)

复制代码

parser=OptionParser()形参包括:def\_\_init\_\_(self,

usage=None,

option\_list=None,

option\_class=Option,

version=None,

conflict\_handler="error",

description=None,

formatter=None,

add\_help\_option=True,

prog=None,

epilog=None):

复制代码

[复制代码](javascript:void(0);)

### 新建选项

可以使用:add\_option, add\_option\_group, add\_options.

add\_option(…):  
add\_option 方法中前面的参数为命令的选项, 可以为等价的短名或者长名,一般是前面为短名,后面为长名.

可以配置的参数有以下:

* dest: 可以决定解析后,取值时的属性名, 尤其适于有多个等价参数. 不指定时就是选项不加-的字符串.
* type: 选项的值类型,值的默认类型是字符串, 这里将值指定为其他类型.
* default: 缺省值. 没有设置缺省值的为None.
* help: 选项中有 -h 时打印的 help 信息.
* metavar: 表示显示到 help 中选项的默认值；
* choices: 当 type 设置为 choices 时,需要设置此值.
* const: 指定一个常量值给选项, 该常量值将用于后面store\_const和append\_const,一起合用.
* action: 用于控制对选项和参数的处理,像无参数选项处理,可以设置为以下几种字符串:
  + "store": 储存值到 dest 指定的属性,强制要求后面提供参数;
  + "store\_true": 当使用该选项时,后面的 dest 将设置为 true, 不跟参数.
  + "store\_false": 当使用该选项时,后面的 dest 将设置为 false. 常配合另一个 "store\_true" 的选项使用同一个 dest 时使用. 不跟参数.
  + "append": 储存值到 dest 指定的属性,并且是以数组的形式, 必须跟参数.
  + "store\_const": 用来存储参数为 const 设置的值到 dest 指定的属性当中.常用于 dest 为同名2个以上选项时的处理. 不跟参数.
  + "append\_const": 用来存储参数为 const 设置的数组到 dest 指定的属性当中. 不跟参数.
  + "count": 使用后将给储存值到 dest 指定的属性值加1,可以统计参数中出现次数.用途不大. 不跟参数.
  + "callback": 后面指定回调函数名(不加括号),会将相应opt和args传给回调函数.
  + "help", "version": 对应为帮助和版本. 要另外自己设计时使用.

当 action 设置为 store\_ture / store\_false 时, 解析参数时, 如果有值时为 Ture / False, 没有值时为 None.  
当 dest 相同时, 一个 action 设置为 store\_false, 另一个 action 设置为 store\_ture 时, 解析参数时,以在后面出现的为准.

### 选项相关函数:

* 可以用来同时设置多个选项的默认参数

def set\_default(self, dest, value):def set\_defaults(self,\*\*kwargs):

* 检查是否有相应选项

def has\_option(self, opt\_str):

* 删除选项

[复制代码](javascript:void(0);)

复制代码

def remove\_option(self, opt\_str):

eg.

parser =OptionParser()

parser.add\_option('-f','--file', action='store', dest='file', help='file to handle')if parser.has\_option('-f'):print('content -f')

parser.set\_default('-f','myFile')

parser.remove\_option('-f')ifnot parser.has\_option('-f'):print('do not content -f')

复制代码

[复制代码](javascript:void(0);)

输出

content -f

donot content -f

* 添加选项组 add\_option\_group()  
  如果 options 很多的时候, 可以进行分组,然后进行添加. 分组的好处是, 对一系列程序参数可以分一个组,如果有独立的 description , 可以另外处理.使用如下：

group=OptionGroup(parser)#创建分组group.add\_option()#添加选项

parser.add\_option\_group(group)#将分组加入到解释器

* 添加选项数组 add\_options([Option1,…])  
  将各个Option对象放在一个列表里再一起添加.

## 参数解析 parse\_args()

使用 parse\_args() 对参数进行解析,默认是使用 sys.argv[1:] 作为参数, 也可以传递一个命令行参数列表: parse\_args(list).

parse\_args() 返回的两个值：

* options，它是一个对象，保存有命令行参数值。只要知道命令行参数名，如 input，就可以访问其对应的值：options.input 。
* args ，它是没被解析的命令行参数的列表。

### 帮助文档

默认自动带有 -h 和 --help 来输出帮助文档, 输出后程序终止.  
帮助文档由三部分组成:

1. usage 帮助部分,  
   usage 帮助部分一般在 OptionParser 初始化时输入,为第一个参数, 也可以用具体形参名指定. 可以使用 **%prog** 来表示当前的程序名.
2. description 描述部分  
   在初始化 OptionParser 时 description 形参指定的内容.
3. 选项以及选项说明部分  
   选项加入时定义的说明文字

OptionParser 的形参 version 可以指定 --version 输出的字符串, 同样支持**%prog**, 如 version="%prog 1.0"

[复制代码](javascript:void(0);)

复制代码

# 一个帮助文档解释字符串

hstr ='%prog custom help string'

parser =OptionParser(hstr, description='custom description')

get\_usage(), get\_description(), get\_version():获得对应的字符串.

print\_help(), print\_usage(), print\_description(), print\_version():输出相应内容

error(str):出错并输出str.

复制代码

要自己定制程序的参数选项控制，可以使用python自带的OptParse模块，当然也可以直接解析sys.argv里的参数（但是此比较麻烦）

import optparse

optparse 现在不再更新了,更新版本叫 argparse.

import argparse

# OptParse支持一般性GUN的选项方法，包括：

无参选项, -v

有参选项, -p value, –para=value

值参一体, -pvalue (不支持长参数)

合并选项, -abc, -abcp value (最后一个可以是有参, 其余均无参)

　　解析时候 - 和 -- 的区别：

- 就看后面的值, 要是无参的, 那继续读下一个; 要是有参的,就把参数读进来(分隔或一体).

-- 直接读后面的值;

optparse modlue基本用法

一，基本用法

1，载入OptionParser类,新建对象: OptionParser()

2，添加选项: add\_option(…)

3，参数解析: parse\_args()

举个例子：

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

import optparse

from optparse import OptionParser

# 一个帮助文档解释字符串

hstr = '%prog custom help string'

parser = OptionParser(hstr, description='custom description', version='%prog 1.0')

#通过OptionParser类创建parser实例,初始参数usage中的%prog等同于os.path.basename(sys.argv[0]),即

#你当前所运行的脚本的名字，version参数用来显示当前脚本的版本。

'''

添加参数，-f、--file是长短options，有一即可。

dest='user' 将该用户输入的参数保存到变量user中，可以通过options.user方式来获取该值

action用来表示将option后面的值如何处理，比如:

XXX.py -f test.txt

经过parser.parse\_args()处理后,则将test.txt这个值存储进-f所代表的一个对象，即定义-f中的dest

即option.filename = 'test.txt'

action的常用选项还有store\_true,store\_false等，这两个通常在布尔值的选项中使用。

metavar仅在显示帮助中有用，如在显示帮助时会有：

-f FILE, --filename=FILE write output to FILE

-m MODE, --mode=MODE interaction mode: novice, intermediate, or expert

[default: intermediate]

如果-f这一项没有metavr参数，则在上面会显示为-f FILENAME --filename=FILENAME,即会显示dest的值

defalut是某一选项的默认值，当调用脚本时，参数没有指定值时，即采用default的默认值。

'''

parser.add\_option('-i', '--input', action='store', dest='input', help='read input data from input file')

parser.add\_option('-o', '--output', action='store', dest='output', help='write data to output file')

parser.add\_option('-q', '--quite', action='store\_false', dest='version', help='don\'t print the version')

# parser.add\_option('-v', '--version', action='store\_true', dest='version', default=False, help='print the version')

# parser.add\_option('-v', '--version', action='store\_true', dest='version', help='print the version')

parser.add\_option('-f', '--file', action='store', dest='file', help='file to handle')

parser.add\_option('-a', '--add', action='append', dest='add', help='add to handle')

parser.add\_option('-c', '--count', action='count', dest='count', help='count to handle')

parser.add\_option('-d', '--count1', action='count', dest='count', help='count1 to handle')

#parser.add\_option('-v', '--version', dest='version')

if parser.has\_option('-f'):

print('content -f') # parser.set\_default('-f', 'myFile')

parser.remove\_option('-f')

if not parser.has\_option('-f'):

print('do not content -f')

# 用一个数组模拟命令参数

#testArgs = ['-i', 'someForInput', '-f', 'someForFile', '-vq', '-a', 'test1 test2 test3', '-c', '-d']

testArgs = [ '-i', 'someForInput', 'someForFile', 'someForFile1', '-q', '-a', 'test1 test2 test3', '-c', '-d', '-h']

options, args = parser.parse\_args(testArgs)

print('options : %s' % options)

print('args : %s' % args)

if options.input:

print('input in args : %s' % options.input)

if options.version:

print('version 1.0.0')

# if options.file:

# print('file in args : %s' % options.file)

if options.add:

print('add in args : %s' % options.add)

print('version in args', options.version)

结果：

+ View Code

二，新建对象

1

parse = OptionParse()

形参包括：

def \_\_init\_\_(self,

usage=None,

option\_list=None,

option\_class=Option,

version=None,

conflict\_handler="error",

description=None,

formatter=None,

add\_help\_option=True,

prog=None,

epilog=None):

三，新建选项

3.1 可以使用下面几个方法：

add\_option, add\_option\_group, add\_options.

3.2 具体参数分析：

add\_option(…):

add\_option 方法中前面的参数为命令的选项, 可以为等价的短名或者长名,一般是前面为短名,后面为长名.

可以配置的参数有以下:

dest: 可以决定解析后,取值时的属性名, 尤其适于有多个等价参数. 不指定时就是选项不加-的字符串.

type: 选项的值类型,值的默认类型是字符串, 这里将值指定为其他类型.

default: 缺省值. 没有设置缺省值的为None.

help: 选项中有 -h 时打印的 help 信息.

metavar: 表示显示到 help 中选项的默认值；

choices: 当 type 设置为 choices 时,需要设置此值.

const: 指定一个常量值给选项, 该常量值将用于后面store\_const和append\_const,一起合用.

action: 用于控制对选项和参数的处理,像无参数选项处理,可以设置为以下几种字符串:

"store": 储存值到 dest 指定的属性,强制要求后面提供参数;

"store\_true": 当使用该选项时,后面的 dest 将设置为 true, 不跟参数.

"store\_false": 当使用该选项时,后面的 dest 将设置为 false. 常配合另一个 "store\_true" 的选项使用同一个 dest 时使用. 不跟参数.

"append": 储存值到 dest 指定的属性,并且是以数组的形式, 必须跟参数.

"store\_const": 用来存储参数为 const 设置的值到 dest 指定的属性当中.常用于 dest 为同名2个以上选项时的处理. 不跟参数.

"append\_const": 用来存储参数为 const 设置的数组到 dest 指定的属性当中. 不跟参数.

"count": 使用后将给储存值到 dest 指定的属性值加1,可以统计参数中出现次数.用途不大. 不跟参数.

"callback": 后面指定回调函数名(不加括号),会将相应opt和args传给回调函数.

"help", "version": 对应为帮助和版本. 要另外自己设计时使用.

当 action 设置为 store\_ture / store\_false 时, 解析参数时, 如果有值时为 Ture / False, 没有值时为 None.

当 dest 相同时, 一个 action 设置为 store\_false, 另一个 action 设置为 store\_ture 时, 解析参数时,以在后面出现的为准.

四，选项相关参数：

4.1，可以用来同时设置多个选项的默认参数

def set\_default(self, dest, value):

def set\_defaults(self, \*\*kwargs):

4.2，检查是否有相应的选项

1

def has\_option(self, opt\_str):

4.3，删除选项

1

def remove\_option(self, opt\_str):

4.4，举个例子：

parser = OptionParser()

parser.add\_option('-f', '--file', action='store', dest='file', help='file to handle')

if parser.has\_option('-f'):

print('content -f')

parser.set\_default('-f', 'myFile')

parser.remove\_option('-f')

if not parser.has\_option('-f'):

print('do not content -f')

输出：

content -f

do not content -f

4.5，添加选项组 add\_option\_group()

　　如果 options 很多的时候, 可以进行分组,然后进行添加. 分组的好处是, 对一系列程序参数可以分一个组,如果有独立的 description , 可以另外处理.使用如下：

group = OptionGroup(parser) #创建分组

group.add\_option() #添加选项

parser.add\_option\_group(group) #将分组加入到解释器

4.6，添加选项数组 add\_options([Option1,…])

将各个Option对象放在一个列表里再一起添加

五，参数解析parse\_args()

使用 parse\_args() 对参数进行解析,默认是使用 sys.argv[1:] 作为参数, 也可以传递一个命令行参数列表: parse\_args(list).

parse\_args() 返回的两个值：

1

2

options，它是一个对象，保存有命令行参数值。只要知道命令行参数名，如 input，就可以访问其对应的值：options.input 。

args ，它是没被解析的命令行参数的列表。

六，帮助文档

默认自动带有 -h 和 --help 来输出帮助文档, 输出后程序终止.

帮助文档由三部分组成:

usage 帮助部分,

usage 帮助部分一般在 OptionParser 初始化时输入,为第一个参数, 也可以用具体形参名指定. 可以使用 %prog 来表示当前的程序名.

description 描述部分

在初始化 OptionParser 时 description 形参指定的内容.

选项以及选项说明部分

选项加入时定义的说明文字

OptionParser 的形参 version 可以指定 --version 输出的字符串, 同样支持%prog, 如 version="%prog 1.0"

举个例子：

# 一个帮助文档解释字符串

hstr = '%prog custom help string'

parser = OptionParser(hstr, description='custom description')

get\_usage(), get\_description(), get\_version(): 获得对应的字符串.

print\_help(), print\_usage(), print\_description(), print\_version(): 输出相应内容

error(str): 出错并输出str.

Python命令行解析argparse常用语法使用简介

　　python中命令行解析最简单最原始的方法是使用sys.argv来实现，更高级的可以使用aegparse这个模块，argparse从python2.7开始被加入标准库，所以python版本在2.7以下，那么需要安装一下。

import argparse

parser=argparse.ArgumentParser()

parser.add\_argument("echo",help="echo the string")

args=parser.parse\_args()

print args.echo

1、导入argparse模块

2、创建解析器对象ArgumentParser，可以添加参数。

description：描述程序

　　　　parser=argparse.ArgumentParser(description="This is a example program ")

add\_help：默认是True，可以设置False禁用

3、add\_argument()方法，用来指定程序需要接受的命令参数

定位参数：

　　　　parser.add\_argument("echo",help="echo the string")

可选参数：

　　　　parser.add\_argument("--verbosity", help="increase output verbosity")

在执行程序的时候，定位参数必选，可选参数可选。

4，add\_argument()常用的参数：

dest：如果提供dest，例如dest="a"，那么可以通过args.a访问该参数

default：设置参数的默认值

action：参数出发的动作

store：保存参数，默认

store\_const：保存一个被定义为参数规格一部分的值（常量），而不是一个来自参数解析而来的值。

store\_ture/store\_false：保存相应的布尔值

append：将值保存在一个列表中。

append\_const：将一个定义在参数规格中的值（常量）保存在一个列表中。

count：参数出现的次数

parser.add\_argument("-v", "--verbosity", action="count", default=0, help="increase output verbosity")

version：打印程序版本信息

type：把从命令行输入的结果转成设置的类型

choice：允许的参数值

parser.add\_argument("-v", "--verbosity", type=int, choices=[0, 1, 2], help="increase output verbosity")

help：参数命令的介绍

1，argarse.ArgumentParser.parse\_known\_args() 解析

　　此函数就是有时候，一个脚本只需要解析所有命令行参数中的一小部分，剩下的命令行参数给一两个脚本或者程序。在这种情况下，parse\_known\_args() 就很有用。它很像parse\_args()，但是它在接受到多余的命令行参数时不报错。相反的，返回一个tuple类型的命名空间和一个保存着余下的命令行字符的list。

　　举个例子，新建一个 test.py 文件，在文件中写入如下代码：

import argparse

parser = argparse.ArgumentParser()

parser.add\_argument(

'--flag\_int',

type=float,

default=0.01,

help='flag\_int.'

)

FLAGS, unparsed = parser.parse\_known\_args()

print(FLAGS)

print(unparsed)

　　结果如下：

$ python prog.py --flag\_int 0.02 --double 0.03 a 1

Namespace(flag\_int=0.02)

['--double', '0.03', 'a', '1']

Options:

-h, --help show this help message and exit

-f FILE, --file=FILE write report to FILE

-q, --quiet don't print status messages to stdout

下面简单介绍optparse的用法：

aptparse 在大多数情况下是相当灵活和强大的，本文介绍常见的optparse用法。

首先要导入OptionParser类，在主程序中要创建一个类：

from optparse import OptionParser

[...]

parser = OptionParser()

现在可以定义命令行选项，基本语法是：

parser.add\_option(opt\_str, ...,

attr=value, ...)

每种选项各有一个或多个选项的字符串，比如 -f 或 --file，通常每个选项将有一个短选项和一个长选项。例如：

1 parser.add\_option("-f", "--file", ...)

你可以自由定义为许多短选项和尽可能多的长选项(包括零),但是至少要有一个选项字符串整体。

最后，一旦你已经定义好了所有的命令行参数，调用 parse\_args() 来解析程序的命令行：

1 (options, args) = parser.parse\_args()

注： 你也可以传递一个命令行参数列表到 parse\_args()；否则，默认使用 sys.argv[:1]。

parse\_args() 返回的两个值：

options，它是一个对象（optpars.Values），保存有命令行参数值。只要知道命令行参数名，如 file，就可以访问其对应的值： options.file 。

args，它是一个由 positional arguments 组成的列表。

了解选项操作：

action 是 parse\_args() 方法的参数之一，它指示 optparse 当解析到一个命令行参数时该如何处理。actions 有一组固定的值可供选择，默认是’store ‘，表示将命令行参数值保存在 options 对象里。

示例：

parser.add\_option("-f", "--file",

action="store", type="string", dest="filename")

args = ["-f", "foo.txt"]

(options, args) = parser.parse\_args(args)

print options.filename

最后将会打印出 “foo.txt”。

当 optparse 解析到’-f’，会继续解析后面的’foo.txt’，然后将’foo.txt’保存到 options.filename 里。当调用 parser.args() 后，options.filename 的值就为’foo.txt’。

你也可以指定 add\_option() 方法中 type 参数为其它值，如 int 或者 float 等等：

1 parser.add\_option("-n", type="int", dest="num")

注意：这个选项没有长选项，长选项也是可选的，如果没有指定dest选项，将用命令行的参数名对options对象的值进行存取。store也有其他的两种形式: stort\_true 和 store\_false， 用于处理带命令行选项后面不带值的情况，例如： -v,-q等命令行参数。

parser.add\_option("-v", action="store\_true", dest="verbose")

parser.add\_option("-q", action="store\_false", dest="verbose")

这样的话，当解析到 ‘-v’，options.verbose 将被赋予 True 值，反之，解析到 ‘-q’，会被赋予 False 值。

其它的 actions 值还有：

store\_const 、append 、count 、callback

默认值

parse\_args() 方法提供了一个 default 参数用于设置默认值。如：

parser.add\_option("-v", action="store\_true", dest="verbose")

parser.add\_option("-q", action="store\_false", dest="verbose", default=True)

又或者使用set\_defaults例如：

parser.set\_defaults(verbose=True)

parser.add\_option(...)

(options, args) = parser.parse\_args()

程序生成帮助

optparse 另一个方便的功能是自动生成程序的帮助信息。你只需要为 add\_option() 方法的 help 选项指定帮助信息文本：

usage = "usage: %prog [options] arg1 arg2"

parser = OptionParser(usage=usage)

parser.add\_option("-v", "--verbose",

action="store\_true", dest="verbose", default=True,

help="make lots of noise [default]")

parser.add\_option("-q", "--quiet",

action="store\_false", dest="verbose",

help="be vewwy quiet (I'm hunting wabbits)")

parser.add\_option("-f", "--filename",

metavar="FILE", help="write output to FILE")

parser.add\_option("-m", "--mode",

default="intermediate",

help="interaction mode: novice, intermediate, "

"or expert [default: %default]")

当 optparse 解析到 -h 或者 –help 命令行参数时，会调用 parser.print\_help() 打印程序的帮助信息：

Usage: <yourscript> [options] arg1 arg2

Options:

-h, --help show this help message and exit

-v, --verbose make lots of noise [default]

-q, --quiet be vewwy quiet (I'm hunting wabbits)

-f FILE, --filename=FILE

write output to FILE

-m MODE, --mode=MODE interaction mode: novice, intermediate, or

expert [default: intermediate]

（注意：当脚本打印帮助信息后会退出，不会解析其他选项参数）

自定义程序使用方法：

1 usage = "usage: %prog [options] arg1 arg2"

这行信息会优先打印在程序的选项信息前。当中的 %prog，optparse 会以当前程序名的字符串来替代：如 os.path.basename.(sys.argv[0])。

如果用户没有提供自定义的使用方法信息，optparse 会默认使用： “usage: %prog [options]”。

用户在定义命令行参数的帮助信息时，不用担心换行带来的问题，optparse 会处理好这一切。

设置 add\_option 方法中的 metavar 参数，有助于提醒用户，该命令行参数所期待的参数，如 metavar=“mode”：

1 -m MODE, --mode=MODE

注意： metavar 参数中的字符串会自动变为大写。

在 help 参数的帮助信息里使用 %default 可以插入该命令行参数的默认值。

如果程序有很多的命令行参数，你可能想为他们进行分组，这时可以使用 OptionGroup：

group = OptionGroup(parser, "Dangerous Options",

"Caution: use these options at your own risk. "

"It is believed that some of them bite.")

group.add\_option("-g", action="store\_true", help="Group option.")

parser.add\_option\_group(group)

输出如下：

Usage: <yourscript> [options] arg1 arg2

Options:

-h, --help show this help message and exit

-v, --verbose make lots of noise [default]

-q, --quiet be vewwy quiet (I'm hunting wabbits)

-f FILE, --filename=FILE

write output to FILE

-m MODE, --mode=MODE interaction mode: novice, intermediate, or

expert [default: intermediate]

Dangerous Options:

Caution: use these options at your own risk. It is believed that some

of them bite.

-g Group option.

完整的列子：

group = OptionGroup(parser, "Dangerous Options",

"Caution: use these options at your own risk. "

"It is believed that some of them bite.")

group.add\_option("-g", action="store\_true", help="Group option.")

parser.add\_option\_group(group)

group = OptionGroup(parser, "Debug Options")

group.add\_option("-d", "--debug", action="store\_true",

help="Print debug information")

group.add\_option("-s", "--sql", action="store\_true",

help="Print all SQL statements executed")

group.add\_option("-e", action="store\_true", help="Print every action done")

parser.add\_option\_group(group)

# csv模块

csv文件具有格式简单，快速存取，兼容性好等特点，工程、金融、商业等很多数据文件都是采用csv文件保存和处理。工作中数据处理也用到了csv，简要总结下使用经验，特别是**那些由于本地兼容性导致的与官方文档的差异使用**。

csv（comma Seperated Values）文件的格式非常简单，类似一个文本文档，每一行保存一条数据，同一行中的各个数据通常采用逗号（或tab）分隔。python自带了csv模块，专门用于处理csv文件的读取和存档。

csv模块中，主要由两种方式存取csv文件：函数方法；类方法。

下面首先介绍简单介绍读写csv文件的两种方法，然后**结合用于解析csv文件的属性参数集dialect，具体介绍读写csv文件的技巧和实例**。

**一、函数方法（list方法）**

csv.reader(f [, dialect='excel'][optional kwargs]) #返回csv阅读器（本质是一个迭代器，具有\_\_next\_\_()、\_\_iter\_\_()方法），可通过迭代读取csv文件内容。f为打开csv文件的文件对象，注意，**实测文件应该用text模式（r或rt）打开，而不是官方文档的binary模式打开**（会报\_csv.Error: iterator should return strings, not bytes (did you open the file in text mode?)）。dialect规定csv文件解析所采用的属性集，也可通过关键字参数修改指定属性，下文由详细介绍和实例。

csv.writer(f [, dialect='excel'][optional kwargs]) #返回csv写入器，其可将序列对象（list等）写入到csv文件中，写入方法为writerow()、writerows()，实例中介绍。f为被写入的csv文件对象，同样地，**实测文件应该用text模式（w或wt）打开，而不是官方文档的binary模式打开**。

l = []

with open('test1.csv','rt') as f:

cr = csv.reader(f)

for row in cr:

print(row)

l.append(row) #将test.csv内容读入列表l，每行为其一个元素，元素也为list

with open('1.csv','wt') as f2:

cw = csv.writer(f2)

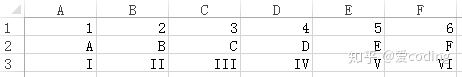
#采用writerow()方法

for item in l:

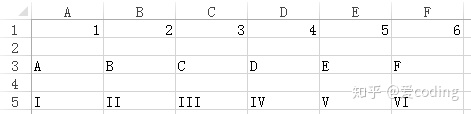
cw.writerow(item) #将列表的每个元素写到csv文件的一行

#或采用writerows()方法

#cw.writerows(l) #将嵌套列表内容写入csv文件，每个外层元素为一行，每个内层元素为一个数据



test1.csv



1.csv

注意：1.csv中，每行数据间有一个空行，这是由于lineterminator属性导致的，下文介绍修正方法。

**二、类方法（dict方法）**

csv.DictReader(f, fieldnames=None, restkey=None, restval=None, dialect='excel', \*args, \*\*kwds) #建立**类字典方式**的csv阅读器（迭代器）。f为打开csv文件的文件对象，同样地，**实测文件应该用text模式（r或rt）打开。**fieldnames（list等）为字典的key值（**相当于列标题**），每个key对应csv文件中的一列，如不指定，则默认读取的文件第一行元素为key。

csv.DictWriter(f, fieldnames, restval='', extrasaction='raise', dialect='excel', \*args, \*\*kwds) #建立类字典方式的csv写入器，具有writeheader()，writerrow(rowdict)，writerows(rowdicts)方法。f为文件类型对象，可为打开csv文件的文件对象（**w或wt模式打开**）。fieldnames（list等）为字典的key值（**列标题**），每个key对应csv文件中的一列，**必须指定**。

l = []

with open('test2.csv','rt') as f:

cr = csv.DictReader(f)

for row in cr:

print(row)

l.append(row) #将test2.csv内容读入列表l，每行为其一个元素，元素为dict

#key为标题（未指定时为读取的第一行），value为对应列的数据

with open('2.csv','wt') as f2:

cw = csv.DictWriter(f2,fieldnames=['标题%d' % i for i in range(1,7)])

cw.writeheader() #将fieldnames写入标题行

#采用writerow()方法

for rowdict in l:

cw.writerow(rowdict) #将列表的每个元素（dict）按照对应的键值对写到csv文件的一行

#或采用writerows()方法

#cw.writerows(l) #将dict组成的list整体写入csv文件，每个dict为一行，每个value为一个数据



test2.csv



2.csv

注意：2.csv同样存在空行问题。

**三、csv文件属性集**

csv文件写入、读取都需要用到csv文件的属性集dialect（也可称之为“方言”^\_^），其主要包括：

* 分隔符delimiter（最重要）。用于将同一行中的各个数据分隔开，通常逗号（','），也有用tab（'\t'）的。
* 换行符lineterminator（也很重要）。用于区分不同行的数据，通常为'\r\n'（**由于操作系统原因，该换行符在写入csv时，可能会造成各行数据之间多出一个空行，详见上文示例**），也有用'\n'。
* 引用符quotechar。用于引用包含特殊字符（如分隔符、引用符、换行符）的区域，通常为双引号（'"'）。
* 转义符escapechar。用于在quoting设置为QUOTE\_NONE时跳过转义符后的分隔符（即将该分隔符不作为分隔符，而是作为内容读入）、在doublequote设置为False时跳过转义符后的引用符（即将该引用符不作为引用符，而是作为内容读入）。**类似于python'\'，可设为任意字符（详见下文示例）**。
* 引用模式quoting。指定特殊字符的表达形式。
  + 0表示csv.QUOTE\_MINIMAL（最小引用，Excel生成的csv）。
  + 1表示csv.QUOTE\_ALL（引用所有，Unix生成的csv）。
  + 2表示QUOTE\_NONNUMERIC（引用非数字）。
  + 3表示csv.QUOTE\_NONE（不引用，表示特殊字符是不被引用符包围的）。
* 引用符引用模式doublequote。指定引用引用符自身的方式。
  + True表示采用双重引用符形式（'""'）。
  + False表示采用在引用符前置转义符的形式（如'\"'。注：如未同时设置escapechar，则将报错）。
* 是否启用严格模式strict。如为True，则在读入的csv文件格式比较差时报错。通常采用默认的False。

csv模块已经内置了3个属性集：

* csv.excel：采用逗号分隔的Excel格式csv
  + delimiter = ','
  + doublequote = True
  + lineterminator = '\r\n'
  + quotechar = '"'
  + quoting = 0
  + skipinitialspace = False
  + escapechar = None
* csv.excel\_tab：采用tab分隔的Excel格式csv
  + delimiter = '\t' #除分隔符外，其他继承自csv.excel类
* csv.unix\_dialect：采用逗号分隔的Unix格式csv
  + delimiter = ','
  + doublequote = True
  + lineterminator = '\n'
  + quotechar = '"'
  + quoting = 1
  + skipinitialspace = False
  + escapechar = None

除此以外，用户自己也可通过csv.register\_dialect()注册新的属性集，或者更简单地，在调用csv读写命令时，**直接通过关键字参数指定各属性**。

**四、csv写入间隔空行问题（换行符实例）**

回过头来看第一、二节，写入的csv文件均在数据行之间存在空行。这是因为采用了默认的csv.excel属性集，换行符为'\r\n'，由于操作系统原因，该换行符在写入csv时，可能会造成各行数据之间多出一个空行**。**

**解决该问题很简单，在调用csv写入器时，直接指定换行符为'\r'或'\n'即可**。

l = []

with open('test1.csv','rt') as f:

cr = csv.reader(f)

for row in cr:

print(row)

l.append(row) #将test.csv内容读入列表l，每行为其一个元素，元素也为list

with open('1a.csv','wt') as f2:

cw = csv.writer(f2, lineterminator = '\n')

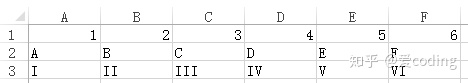
#采用writerow()方法

for item in l:

cw.writerow(item) #将列表的每个元素写到csv文件的一行

#或采用writerows()方法

#cw.writerows(l) #将嵌套列表内容写入csv文件，每个外层元素为一行，每个内层元素为一个数据



1a.csv

**五、转义符实例**

第三节介绍了转义符的功能，下面用实例示范一下。

l = []

with open('test1.csv','rt') as f:

cr = csv.reader(f)

print(cr.\_\_next\_\_())

以上程序中，无转义符，将输出：['1', '2', '3', '4', '5', '6']。

l = []

with open('test1.csv','rt') as f:

cr = csv.reader(f, escapechar = '3')

print(cr.\_\_next\_\_())

以上程序中，将字符'3'设置为转义符，将输出：['1', '2', **',4'**, '5', '6']。

可见，转义符'3'本身不再输出，且其后紧跟的分隔符','作为内容输出，不再作为分隔符。

# tempfile模块：生成临时文件和临时目录

tempfile 模块中常用的函数，如表 1 所示。

|  |  |
| --- | --- |
| 表 1 tempfile 模块常用函数及功能 | |
| **tempfile 模块函数** | **功能描述** |
| tempfile.TemporaryFile(mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None) | 创建临时文件。该函数返回一个类文件对象，也就是支持文件 I/O。 |
| tempfile.NamedTemporaryFile(mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None, delete=True) | 创建临时文件。该函数的功能与上一个函数的功能大致相同，只是它生成的临时文件在文件系统中有文件名。 |
| tempfile.SpooledTemporaryFile(max\_size=0, mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None) | 创建临时文件。与 TemporaryFile 函数相比，当程序向该临时文件输出数据时，会先输出到内存中，直到超过 max\_size 才会真正输出到物理磁盘中。 |
| tempfile.TemporaryDirectory(suffix=None, prefix=None, dir=None) | 生成临时目录。 |
| tempfile.gettempdir() | 获取系统的临时目录。 |
| tempfile.gettempdirb() | 与 gettempdir() 相同，只是该函数返回字节串。 |
| tempfile.gettempprefix() | 返回用于生成临时文件的前缀名。 |
| tempfile.gettempprefixb() | 与 gettempprefix() 相同，只是该函数返回字节串。 |

提示：表中有些函数包含很多参数，但这些参数都具有自己的默认值，因此如果没有特殊要求，可以不对其传参。

tempfile 模块还提供了 tempfile.mkstemp() 和 tempfile.mkdtemp() 两个低级别的函数。上面介绍的 4 个用于创建临时文件和临时目录的函数都是高级别的函数，高级别的函数支持自动清理，而且可以与 with 语句一起使用，而这两个低级别的函数则不支持，因此一般推荐使用高级别的函数来创建临时文件和临时目录。  
  
此外，tempfile 模块还提供了 tempfile.tempdir 属性，通过对该属性赋值可以改变系统的临时目录。  
  
下面程序示范了如何使用临时文件和临时目录：

1. import tempfile
2. # 创建临时文件
3. fp = tempfile.TemporaryFile()
4. **print**(fp.name)
5. fp.write('两情若是久长时，'.encode('utf-8'))
6. fp.write('又岂在朝朝暮暮。'.encode('utf-8'))
7. # 将文件指针移到开始处，准备读取文件
8. fp.seek(0)
9. **print**(fp.read().decode('utf-8')) # 输出刚才写入的内容
10. # 关闭文件，该文件将会被自动删除
11. fp.close()
12. # 通过with语句创建临时文件，with会自动关闭临时文件
13. with tempfile.TemporaryFile() as fp:
14. # 写入内容
15. fp.write(b'I Love [Python](http://c.biancheng.net/python/)!')
16. # 将文件指针移到开始处，准备读取文件
17. fp.seek(0)
18. # 读取文件内容
19. **print**(fp.read()) # b'I Love Python!'
20. # 通过with语句创建临时目录
21. with tempfile.TemporaryDirectory() as tmpdirname:
22. **print**('创建临时目录', tmpdirname)

上面程序以两种方式来创建临时文件：

1. 第一种方式是手动创建临时文件，读写临时文件后需要主动关闭它，当程序关闭该临时文件时，该文件会被自动删除。
2. 第二种方式则是使用 with 语句创建临时文件，这样 with 语句会自动关闭临时文件。

上面程序最后还创建了临时目录。由于程序使用 with 语句来管理临时目录，因此程序也会自动删除该临时目录。  
  
运行上面程序，可以看到如下输出结果：

C:\Users\admin\AppData\Local\Temp\tmphvehw9z1  
两情若是久长时，又岂在朝朝暮暮。  
b'I Love Python!'  
创建临时目录C:\Users\admin\AppData\Local\Temp\tmp3sjbnwob

上面第一行输出结果就是程序生成的临时文件的文件名，最后一行输出结果就是程序生成的临时目录的目录名。需要注意的是，不要去找临时文件或临时文件夹，因为程序退出时该临时文件和临时文件夹都会被删除。

实例1

import os

import tempfile

fd, path = tempfile.mkstemp()

try:

with os.fdopen(fd, 'w') as tmp:

# do stuff with temp file

tmp.write('stuff')

finally:

os.remove(path)

实例2

import os

import tempfile

# 句柄关闭，文件即删除

with tempfile.TemporaryFile() as tmp:

# Do stuff with tmp

tmp.write('stuff')

实例3：

import os

import tempfile

tmp = tempfile.NamedTemporaryFile(delete=True)

try:

# do stuff with temp

tmp.write('stuff')

finally:

tmp.close() # 文件关闭即删除