原创

第十章 Python常用标准库使用（必会）

李振良OK 关注 2016-11-14 10:37:02 12464人阅读 3人评论

本章涉及标准库：

1、sys

2、os

3、glob

4、math

5、random

6、platform

7、pikle与cPikle

8、subprocess

9、Queue

10、StringIO

11、logging

12、ConfigParser

13、urllib与urllib2

14、json

15、time

16、datetime

10.1 sys

1）sys.argv

命令行参数。

argv[0] #代表本身名字

argv[1] #第一个参数

argv[2] #第二个参数

argv[3] #第三个参数

argv[N] #第N个参数

argv #参数以空格分隔存储到列表。

看看使用方法：

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys
print sys.argv[0]
print sys.argv[1]
print sys.argv[2]
```

```
print sys.argv[3]
print sys.argv
print len(sys.argv)

# python test.py
test.py
a
b
c
c
['test.py', 'a', 'b', 'c']
4
```

值得注意的是，argv既然是一个列表，那么可以通过len()函数获取这个列表的长度从而知道输入的参数数量。可以看到列表把自身文件名也写了进去，所以当我们统计的使用应该-1才是实际的参数数量，因此可以len(sys.argv[1:])获取参数长度。

2）sys.path

模块搜索路径。

```
>>> sys.path
['', '/usr/local/lib/python2.7/dist-packages/tornado-3.1-py2.7.egg', '/usr/lib/python2.7', '/usr/lib/python2
```

输出的是一个列表，里面包含了当前Python解释器所能找到的模块目录。

如果想指定自己的模块目录，可以直接追加：

```
>>> sys.path.append('/opt/scripts')
>>> sys.path
['', '/usr/local/lib/python2.7/dist-packages/tornado-3.1-py2.7.egg', '/usr/lib/python2.7', '/usr/lib/python2
```

3）sys.platform

系统平台标识符。

系统	平台标识符
Linux	linux
Windows	win32
Windows/Cygwin	cygwin
Mac OS X	darwin

>>> sys.platform

'linux2'

Python本身就是跨平台语言，但也不就意味着所有的模块都是在各种平台通用，所以可以使用这个方法判断当前平台，做相应的操作。

4）sys.subversion

在第一章讲过Python解释器有几种版本实现，而默认解释器是CPython，来看看是不是：

```
>>> sys.subversion
('CPython', '', '')
```

5）sys.version

查看Python版本：

```
>>> sys.version
'2.7.6 (default, Jun 22 2015, 17:58:13) \n[GCC 4.8.2]'
```

6）sys.exit()

退出解释器：

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys
print "Hello world!"
sys.exit()
print "Hello world!"

# python test.py
Hello world!
```

代码执行到sys.exit()就会终止程序。

7）sys.stdin、sys.stdout和sys.stderr

标准输入、标准输出和错误输出。

标准输入：一般是键盘。stdin对象为解释器提供输入字符流，一般使用raw_input()和input()函数。

例如：让用户输入信息

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys
name = raw_input("Please input your name: ")
print name

# python test.py
Please input your name: xiaoming
xiaoming


import sys
print "Please enter your name: "
name = sys.stdin.readline()
print name

# python b.py
Please enter your name:
xiaoming
xiaoming
```

再例如，a.py文件标准输出作为b.py文件标准输入：

```
# cat a.py
import sys
sys.stdout.write("123456\n")
sys.stdout.flush()
# cat b.py
import sys
print sys.stdin.readlines()

# python a.py | python b.py
['123456\n']
```

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

标准输出：一般是屏幕。stdout对象接收到print语句产生的输出。

例如：打印一个字符串

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys
print "Hello world!"

# python test.py
Hello world!
```

sys.stdout是有缓冲区的，比如：

```
import sys
import time
for i in range(5):
    print i,
    # sys.stdout.flush()
    time.sleep(1)
# python test.py
0 1 2 3 4
```

本是每隔一秒输出一个数字，但现在是循环完才会打印所有结果。如果把sys.stdout.flush()去掉，就会没执行到print就会刷新stdout输出，这对实时输出信息的程序有帮助。

错误输出：一般是错误信息。stderr对象接收出错的信息。

例如：引发一个异常

```
>>> raise Exception, "raise..."
Traceback (most recent call last):File "<stdin>", line 1, in <module>
Exception: raise...
```

博客地址：http://lizhenliang.blog.51cto.com
QQ群：323779636（Shell/Python运维开发群）

10.2 os

os模块主要对目录或文件操作。

方法	描述	示例
os.name	返回操作系统类型	返回值是"posix"代表linux，"nt"代表windows
os.extsep	返回一个"."标识符	
os.environ	以字典形式返回系统变量	
os.devnull	返回/dev/null标识符	
os.linesep	返回一个换行符"\n"	>>> print "a" + os.linesep + "b" a b
os.sep	返回一个路径分隔符正斜杠"/"	>>> "a" + os.sep + "b" 'a/b'
os.listdir(path)	列表形式列出目录	
os.getcwd()	获取当前路径	>>> os.getcwd() '/home/user'

os.chdir(path)	改变当前工作目录到指定目录	>>> os.chdir('/opt')
		>>> os.getcwd()
		'/opt'

os.mkdir(path [, mode=0777])	创建目录	>>> os.mkdir('/home/user/test')
os.makedirs(path [, mode=0777])	递归创建目录	>>> os.makedirs('/home/user/abc/abc')
os.rmdir(path)	移除空目录	>>> os.makedirs('/home/user/abc/abc')
os.remove(path)	移除文件	
os.rename(old, new)	重命名文件或目录	
os.stat(path)	获取文件或目录属性	
os.chown(path, uid, gid)	改变文件或目录所有者	
os.chmod(path, mode)	改变文件访问权限	>>> os.chmod('/home/user/c/a.tar.gz', 0777)
os.symlink(src, dst)	创建软链接	
os.unlink(path)	移除软链接	>>> os.unlink('/home/user/ddd')
urandom(n)	返回随机字节，适合加密使用	>>> os.urandom(2) '%\xec'
os.getuid()	返回当前进程UID	
os.getlogin()	返回登录用户名	
os.getpid()	返回当前进程ID	
os.kill(pid, sig)	发送一个信号给进程	
os.walk(path)	目录树生成器，返回格式：(dirpath, [dirnames], [filenames])	>>> for root, dir, file in os.walk('/home/user/abc'): ... print root ... print dir ... print file
os.statvfs(path)		
os.system(command)	执行shell命令，不能存储结果	
popen(command [, mode='r' [, bufsize]])	打开管道来自shell命令，并返回一个文件对象	>>> result = os.popen('ls') >>> result.read()

os.path类用于获取文件属性。

os.path.basename(path)	返回最后一个文件或目录名	>>> os.path.basename('/home/user/a.sh') 'a.sh'
os.path.dirname(path)	返回最后一个文件前面目录	>>> os.path.dirname('/home/user/a.sh') '/home/user'
os.path.abspath(path)	返回一个绝对路径	>>> os.path.abspath('a.sh') '/home/user/a.sh'
os.path.exists(path)	判断路径是否存在，返回布尔值	>>> os.path.exists('/home/user/abc') True
os.path.isdir(path)	判断是否是目录	
os.path.isfile(path)	判断是否是文件	
os.path.islink(path)	判断是否是链接	
os.path.ismount(path)	判断是否挂载	
os.path.getatime(filename)	返回文件访问时间戳	>>> os.path.getctime('a.sh')

		1475240301.9892483
os.path.getctime(filename)	返回文件变化时间戳	
os.path.getmtime(filename)	返回文件修改时间戳	
os.path.getsize(filename)	返回文件大小，单位字节	
os.path.join(a, *p)	加入两个或两个以上路径，以正斜杠"/"分隔。常用于拼接路径	>>> os.path.join('/home/user','test.py','a.py') '/home/user/test.py/a.py'
os.path.split(	分隔路径名	>>> os.path.split('/home/user/test.py') ('/home/user', 'test.py')
os.path.splitext(	分隔扩展名	>>> os.path.splitext('/home/user/test.py') ('/home/user/test', '.py')

10.3 glob

文件查找，支持通配符（*、？、[]）

```
# 查找目录中所有以.sh为后缀的文件
>>> glob.glob('/home/user/*.sh')
['/home/user/1.sh', '/home/user/b.sh', '/home/user/a.sh', '/home/user/sum.sh']

# 查找目录中出现单个字符并以.sh为后缀的文件
>>> glob.glob('/home/user/?.sh')
['/home/user/1.sh', '/home/user/b.sh', '/home/user/a.sh']

# 查找目录中出现a.sh或b.sh的文件
>>> glob.glob('/home/user/[a|b].sh')
['/home/user/b.sh', '/home/user/a.sh']
```

10.4 math

数字处理。

下面列出一些自己决定会用到的：

方法	描述	示例
math.pi	返回圆周率	>>> math.pi 3.141592653589793
math.ceil(x)	返回x浮动的上限	>>> math.ceil(5.2) 6.0
math.floor(x)	返回x浮动的下限	>>> math.floor(5.2) 5.0
math.trunc(x)	将数字截尾取整	>>> math.trunc(5.2) 5
math.fabs(x)	返回x的绝对值	>>> math.fabs(-5.2) 5.2
math.fmod(x, y)	返回x%y(取余)	>>> math.fmod(5,2) 1.0
math.modf(x)	返回x小数和整数	>>> math.modf(5.2) (0.20000000000000018, 5.0)

df(x)		
math.factorial(x)	返回x的阶乘	>>> math.factorial(5) 120
math.pow(x, y)	返回x的y次方	>>> math.pow(2,3) 8.0
math.sprt(x)	返回x的平方根	>>> math.sqrt(5) 2.2360679774997898

10.5 random

生成随机数。

常用的方法：

方法	描述	示例
random.randint(a,b)	返回整数a和b范围内数字	>>> random.randint(1,10) 6
random.random()	返回随机数，它在0和1范围内	>>> random.random() 0.7373251914304791
random.randrange(start, stop[, step])	返回整数范围的随机数，并可以设置只返回跳数	>>> random.randrange(1,10,2) 5
random.sample(array, x)	从数组中返回随机x个元素	>>> random.sample([1,2,3,4,5],2) [2, 4]

10.6 platform

获取操作系统详细信息。

方法	描述	示例
platform.platform()	返回操作系统平台	>>> platform.platform() 'Linux-3.13.0-32-generic-x86_64-with-Ubuntu-14.04-trusty'
platform.uname()	返回操作系统信息	>>> platform.uname() ('Linux', 'ubuntu', '3.13.0-32-generic', '#57-Ubuntu SMP Tue Jul 15 03:51:08 UTC 2014', 'x86_64', 'x86_64')
platform.	返回操作系统平台	>>> platform.system() 'Linux'

system()		
platform.version()	返回操作系统版本	>>> platform.version() '#57-Ubuntu SMP Tue Jul 15 03:51:08 UTC 2014'
platform.machine()	返回计算机类型	>>> platform.machine() 'x86_64'
platform.processor()	返回计算机处理器类型	>>> platform.processor() 'x86_64'
platform.node()	返回计算机网络名	>>> platform.node() 'ubuntu'
platform.python_version()	返回Python版本号	>>> platform.python_version() '2.7.6'

10.7 pickle与cPickle

创建可移植的Python序列化对象，持久化存储到文件。

1）pickle

pickle库有两个常用的方法，dump()、load() 和dumps()、 loads()，下面看看它们的使用方法：

dump()方法是把对象保存到文件中。

格式：dump(obj, file, protocol=None)

load()方法是从文件中读数据，重构为原来的Python对象。

格式：load(file)

示例，将字典序列化到文件：

```
>>> import pickle
>>> dict = {'a':1, 'b':2, 'c':3}
>>> output = open('data.pkl', 'wb')  # 二进制模式打开文件
>>> pickle.dump(dict, output)    # 执行完导入操作，当前目录会生成data.pkl文件
>>> output.close()  # 写入数据并关闭
```

看看pickle格式后的文件：

```
# cat data.pkl
(dp0
S'a'
p1
I1
sS'c'
p2
I3
sS'b'
p3
```

```
I2
s.
```

读取序列化文件：

```
>>> f = open('data.pkl')
>>> data = pickle.load(f)
>>> print data
{'a': 1, 'c': 3, 'b': 2}
```

用法挺简单的，就是先导入文件，再读取文件。

接下来看看序列化字符串操作：

dumps()返回一个pickle格式化的字符串

格式：dumps(obj, protocol=None)

load()解析pickle字符串为对象

示例：

```
>>> s = 'abc'
>>> pickle.dumps(s)
"S'abc'\np0\n."
>>> pkl = pickle.dumps(s)
>>> pkl
"S'abc'\np0\n."
>>> pickle.loads(pkl)
'abc'
```

需要注意的是，py2.x使用的是pickle2.0格式版本，如果用3.0、4.0版本的pickle导入会出错。可以通过pickle.format_version 查看版本。

博客地址：http://lizhenliang.blog.51cto.com
QQ群：323779636（Shell/Python运维开发群）

2）cPickle

cPickle库是C语言实现，对pickle进行了优化，提升了性能，建议在写代码中使用。

cPicke提供了与pickle相同的dump()、load() 和dumps()、 loads()方法，用法一样，不再讲解。

10.8 subprocess

subprocess库会fork一个子进程去执行任务，连接到子进程的标准输入、输出、错误，并获得它们的返回代码。这个模块将取代os.system、os.spawn*、os.popen*、popen2.*和commands.*。

提供了以下常用方法帮助我们执行bash命令的相关操作：

subprocess.call()：运行命令与参数。等待命令完成，返回执行状态码。

```
>>> import subprocess
>>> retcode = subprocess.call(["ls", "-l"])
total 504
-rw-r--r-- 1 root root      54 Nov  2 06:15 data.pkl
>>> retcode
0
>>> retcode = subprocess.call(["ls", "a"])
ls: cannot access a: No such file or directory
>>> retcode
2

# 也可以这样写
>>> subprocess.call('ls -l', shell=True)
```

subprocess.check_call()：运行命令与参数。如果退出状态码非0，引发CalledProcessError异常，包含状态码。

```
>>> subprocess.check_call("ls a", shell=True)
ls: cannot access a: No such file or directory
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/subprocess.py", line 540, in check_call
    raise CalledProcessError(retcode, cmd)
subprocess.CalledProcessError: Command 'ls a' returned non-zero exit status 2
```

subprocess.Popen()：这个类我们主要来使用的，参数较多。

参数	描述
args	命令，字符串或列表
bufsize	0代表无缓冲，1代表行缓冲，其他正值代表缓冲区大小，负值采用默认系统缓冲（一般是全缓冲）
executable	
stdin stdout stderr	默认没有任何重定向，可以指定重定向到管道（PIPE）、文件对象、文件描述符（整数），stderr还可以设置为STDOUT
preexec_fn	钩子函数，在fork和exec之间执行
close_fds	
shell	为True，表示用当前默认解释器执行。相当于args前面添加“/bin/sh”“-c或win下"cmd.exe /c "
cwd	指定工作目录
env	设置环境变量
universal_newlines	换行符统一处理成"\n"
startupinfo	在windows下的Win32 API 发送CreateProcess()创建进程
creationflags	在windows下的Win32 API 发送CREATE_NEW_CONSOLE()创建控制台窗口

subprocess.Popen()类又提供了以下些方法：

方法	描述
Popen.communicate(input=None)	与子进程交互。读取从stdout和stderr缓冲区内容，阻塞父进程，等待子进程结束
Popen.kill()	杀死子进程，在Posix系统上发送SIGKILL信号
Popen.pid	获取子进程PID
Popen.poll()	如果子进程终止返回状态码

Popen.returncode	返回子进程状态码
Popen.send_signal(signal)	发送信号到子进程
Popen.stderr	如果参数值是PIPE，那么这个属性是一个文件对象，提供子进程错误输出。否则为None
Popen.stdin	如果参数值是PIPE，那么这个属性是一个文件对象，提供子进程输入。否则为None
Popen.stdout	如果参数值是PIPE，那么这个属性是一个文件对象，提供子进程输出。否则为None
Popen.terminate()	终止子进程，在Posix系统上发送SIGTERM信号，在windows下的Win32 API发送TerminateProcess()到子进程
Popen.wait()	等待子进程终止，返回状态码

示例：

```
>>> p = subprocess.Popen('dmesg |grep eth0', stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell
>>> p.communicate()
.....  # 元组形式返回结果
>>> p.pid
57039
>>> p.wait()
0
>>> p.returncode
0
```

subprocess.PIPE提供了一个缓冲区，将stdout、stderr放到这个缓冲区中，p.communicate()方法读取缓冲区数据。

缓冲区的stdout、stderr是分开的，可以以p.stdout.read()方式获得标准输出、错误输出的内容。

再举个例子，我们以标准输出作为下个Popen任务的标准输入：

>>> p1 = subprocess.Popen('ls', stdout=subprocess.PIPE, shell=True)

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

>>> p1.stdout.close() # 调用后启动p2，为了获得SIGPIPE

>>> output = p2.communicate()[0]

>>> output

'data.pkl\n'

p1的标准输出作为p2的标准输入。这个p2的stdin、stdout也可以是个可读、可写的文件。

10.9 Queue

队列，数据存放在内存中，一般用于交换数据。

类	**描述**
Queue.Empty	当非阻塞get()或get_nowait()对象队列上为空引发异常
Queue.Full	当非阻塞put()或put_nowait()对象队列是一个满的队列引发异常
Queue.	构造函数为后进先出队列。maxsize设置队列最大上限项目数量。小于

LifoQueue(maxsize=0)	或等于0代表无限。
Queue.PriorityQueue(maxsize=0)	构造函数为一个优先队列。级别越高越先出。
Queue.Queue(maxsize=0)	构造函数为一个FIFO(先进先出)队列。maxsize设置队列最大上限项目数量。小于或等于0代表无限。
Queue.deque	双端队列。实现快速append()和popleft()，无需锁。
Queue.heapq	堆排序队列。

用到比较多的是Queue.Queue类，在这里主要了解下这个。

它提供了一些操作队列的方法：

方法	描述
Queue.empty()	如果队列为空返回True，否则返回False
Queue.full()	如果队列是满的返回True，否则返回False
Queue.get(block=True, timeout=None)	从队列中删除并返回一个项目。没有指定项目，因为是FIFO队列，如果队列为空会一直阻塞。timeout超时时间
Queue.get_nowait()	从队列中删除并返回一个项目，不阻塞。会抛出异常。
Queue.join()	等待队列为空，再执行别的操作
Queue.put(item, block=True, timeout=None)	写入项目到队列
Queue.put_nowait()	写入项目到队列，不阻塞。与get同理
Queue.qsize()	返回队列大小
Queue.task_done()	表示原队列的任务完成

示例：

```
>>> from Queue import Queue
>>> q = Queue()
>>> q.put('test')
>>> q.qsize()
1
>>> q.get()
'test'
>>> q.qsize()
0
>>> q.full()
False
>>> q.empty()
True
```

10.10 StringIO

StringIO库将字符串存储在内存中，像操作文件一样操作。主要提供了一个StringIO类。

方法	描述
StringIO.close()	关闭
StringIO.flush()	刷新缓冲区
StringIO.getvalue()	获取写入的数据
StringIO.isatty()	
StringIO.next()	读取下一行，没有数据抛出异常
StringIO.read(n=-1)	默认读取所有内容。n指定读取多少字节
StringIO.readline(length=None)	默认读取下一行。length指定读取多少个字符
StringIO.readlines(sizehint=0)	默认读取所有内容，以列表返回。sizehint指定读取多少字节
StringIO.seek(pos, mode=0)	在文件中移动文件指针，从mode（0代表文件起始位置，默认。1代表当前位置。2代表文件末尾）偏移pos个字节
StringIO.tell()	返回当前在文件中的位置
StringI	截断文件大小

O.truncate()	
StringIO.write(str)	写字符串到文件
StringIO.writelines(iterable)	写入序列，必须是一个可迭代对象，一般是一个字符串列表

可以看到，StringIO方法与文件对象方法大部分都一样，从而也就能方面的操作内存对象。

示例：

```
>>> f = StringIO()
>>> f.write('hello')
>>> f.getvalue()
'hello'
```

像操作文件对象一样写入。

用一个字符串初始化StringIO，可以像读文件一样读取：

```
>>> f = StringIO('hello\nworld!')
>>> f.read()
'hello\nworld!'
>>> s = StringIO('hello world!')
>>> s.seek(5)                # 指针移动到第五个字符，开始写入
>>> s.write('-')
>>> s.getvalue()
'hello-world!'
```

10.11 logging

记录日志库。

有几个主要的类：

logging.Logger	应用程序记录日志的接口
logging.Filter	过滤哪条日志不记录
logging.FileHandler	日志写到磁盘文件
logging.Formatter	定义最终日志格式

日志级别：

级别	数字值	描述
critical	50	危险
error	40	错误
warning	30	警告
info	20	普通信息
debug	10	调试

noset	0	不设置

Formatter类可以自定义日志格式，默认时间格式是%Y-%m-%d %H:%M:%S，有以下这些属性：

%(name)s	日志的名称
%(levelno)s	数字日志级别
%(levelname)s	文本日志级别
%(pathname)s	调用logging的完整路径（如果可用）
%(filename)s	文件名的路径名
%(module)s	模块名
%(lineno)d	调用logging的源行号
%(funcName)s	函数名
%(created)f	创建时间，返回time.time()值
%(asctime)s	字符串表示创建时间
%(msecs)d	毫秒表示创建时间
%(relativeCreated)d	毫秒为单位表示创建时间，相对于logging模块被加载，通常应用程序启动。
%(thread)d	线程ID（如果可用）
%(threadName)s	线程名字（如果可用）
%(process)d	进程ID（如果可用）
%(message)s	输出的消息

示例：

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#-----------------------------------------------
# 日志格式
#-----------------------------------------------
# %(asctime)s       年-月-日 时-分-秒,毫秒 2013-04-26 20:10:43,745
# %(filename)s      文件名，不含目录
# %(pathname)s      目录名，完整路径
# %(funcName)s      函数名
# %(levelname)s     级别名
```

```
# %(lineno)d      行号
# %(module)s      模块名
# %(message)s     消息体
# %(name)s        日志模块名
# %(process)d     进程id
# %(processName)s   进程名
# %(thread)d      线程id
# %(threadName)s    线程名
import logging
format = logging.Formatter('%(asctime)s - %(levelname)s %(filename)s [line:%(lineno)d] %(message)s')
# 创建日志记录器
info_logger = logging.getLogger('info')
# 设置日志级别,小于INFO的日志忽略
info_logger.setLevel(logging.INFO)
# 日志记录到磁盘文件
info_file = logging.FileHandler("info.log")
# info_file.setLevel(logging.INFO)
# 设置日志格式
info_file.setFormatter(format)
info_logger.addHandler(info_file)
error_logger = logging.getLogger('error')
error_logger.setLevel(logging.ERROR)
error_file = logging.FileHandler("error.log")
error_file.setFormatter(format)
error_logger.addHandler(error_file)
# 输出控制台（stdout）
console = logging.StreamHandler()
console.setLevel(logging.DEBUG)
console.setFormatter(format)
info_logger.addHandler(console)
error_logger.addHandler(console)
if __name__ == "__main__":
    # 写日志
    info_logger.warning("info message.")
    error_logger.error("error message!")


# python test.py
2016-07-02 06:52:25,624 - WARNING test.py [line:49] info message.
2016-07-02 06:52:25,631 - ERROR test.py [line:50] error message!
# cat info.log
2016-07-02 06:52:25,624 - WARNING test.py [line:49] info message.
# cat error.log
2016-07-02 06:52:25,631 - ERROR test.py [line:50] error message!
```

上面代码实现了简单记录日志功能。分别定义了info和error日志，将等于或高于日志级别的日志写到日志文件中。在小项目开发中把它单独写一个模块，很方面在其他代码中调用。

需要注意的是，在定义多个日志文件时，getLogger(name=None)类的name参数需要指定一个唯一的名字，如果没有指定，日志会返回到根记录器，也就是意味着他们日志都会记录到一起。

10.12 ConfigParser

配置文件解析。

这个库我们主要用到ConfigParser.ConfigParser()类，对ini格式文件增删改查。

ini文件固定结构：有多个部分块组成，每个部分有一个[标识]，并有多个key，每个key对应每个值，以等号"="分隔。值的类型有三种：字符串、整数和布尔值。其中字符串可以不用双引号，布尔值为真用1表示，布尔值为假用0表示。注释以分号";"开头。

方法	描述
ConfigParser.add_section(section)	创建一个新的部分配置

ConfigParser.get(section, option, raw=False, vars=None)	获取部分中的选项值，返回字符串
ConfigParser.getboolean(section, option)	获取部分中的选项值，返回布尔值
ConfigParser.getfloat(section, option)	获取部分中的选项值，返回浮点数
ConfigParser.getint(section, option)	获取部分中的选项值，返回整数
ConfigParser.has_option(section, option)	检查部分中是否存在这个选项
ConfigParser.has_section(section)	检查部分是否在配置文件中
ConfigParser.items(section, raw=False, vars=None)	列表元组形式返回部分中的每一个选项
ConfigParser.options(section)	列表形式返回指定部分选项名称
ConfigParser.read(filenames)	读取ini格式的文件
ConfigParser.remove_option(section, option)	移除部分中的选项
ConfigParser.remove_section(section, option)	移除部分
ConfigParser.sections()	列表形式返回所有部分名称
ConfigParser.set(section, option, value)	设置选项值，存在则更新，否则添加
ConfigParser.write(fp)	写一个ini格式的配置文件

举例说明，写一个ini格式文件，对其操作：

```
# cat config.ini
[host1]
host = 192.168.1.1
port = 22
user = zhangsan
pass = 123
[host2]
host = 192.168.1.2
port = 22
user = lisi
pass = 456
[host3]
host = 192.168.1.3
port = 22
user = wangwu
pass = 789
```

1）获取部分中的键值

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from ConfigParser import ConfigParser
conf = ConfigParser()
conf.read("config.ini")
section = conf.sections()[0]   # 获取随机的第一个部分标识
options = conf.options(section) # 获取部分中的所有键
key = options[2]
value = conf.get(section, options[2]) # 获取部分中键的值
print key, value
print type(value)

# python test.py
port 22
<type 'str'>
```

这里有意打出来了值的类型，来说明下get()方法获取的值都是字符串，如果有需要，可以getint()获取整数。测试发现，ConfigParser是从下向上读取的文件内容！

2）遍历文件中的每个部分的每个字段

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from ConfigParser import ConfigParser
conf = ConfigParser()
conf.read("config.ini")
sections = conf.sections()  # 获取部分名称 ['host3', 'host2', 'host1']
for section in sections:
    options = conf.options(section) # 获取部分名称中的键 ['user', 'host', 'port', 'pass']
    for option in options:
        value = conf.get(section, option) # 获取部分中的键值
        print option + ": " + value
    print "-------------"

# python test.py
user: wangwu
host: 192.168.1.3
port: 22
pass: 789
-------------
user: lisi
host: 192.168.1.2
port: 22
pass: 456
-------------
user: zhangsan
host: 192.168.1.1
port: 22
pass: 123
-------------
```

通过上面的例子，熟悉了sections()、options()和get()，能任意获取文件的内容了。

也可以使用items()获取部分中的每个选项：

```
from ConfigParser import ConfigParser
conf = ConfigParser()
conf.read("config.ini")
print conf.items('host1')

# python test.py
[('user', 'zhangsan'), ('host', '192.168.1.1'), ('port', '22'), ('pass', '123')]
```

3）更新或添加选项

```
from ConfigParser import ConfigParser
conf = ConfigParser()
conf.read("config.ini")
fp = open("config.ini", "w")    # 写模式打开文件，供后面提交写的内容
conf.set("host1", "port", "2222")  # 有这个选项就更新，否则添加
conf.write(fp)  # 写入的操作必须执行这个方法
```

4）添加一部分，并添加选项

```
from ConfigParser import ConfigParser
conf = ConfigParser()
conf.read("config.ini")
fp = open("config.ini", "w")
conf.add_section("host4")    # 添加[host4]
conf.set("host4", "host", "192.168.1.4")
```

```
conf.set("host4", "port", "22")
conf.set("host4", "user", "zhaoliu")
conf.set("host4", "pass", "123")
conf.write(fp)
```

5）删除一部分

```
from ConfigParser import ConfigParser
conf = ConfigParser()
conf.read("config.ini")
fp = open("config.ini", "w")
conf.remove_section('host4')  # 删除[host4]
conf.remove_option('host3', 'pass')  # 删除[host3]的pass选项
conf.write(fp)
```

10.13 urllib与urllib2

打开URL。urllib2是urllib的增强版，新增了一些功能，比如Request()用来修改Header信息。但是urllib2还去掉了一些好用的方法，比如urlencode()编码序列中的两个元素（元组或字典）为URL查询字符串。

一般情况下这两个库结合着用，那我们也结合着了解下。

类	描述
urllib.urlopen(url, data=None, proxies=None)	读取指定URL，创建类文件对象。data是随着URL提交的数据（POST）
urllib/urllib2.quote(s, safe='/')	将字符串中的特殊符号转十六进制表示。如： quote('abc def') -> 'abc%20def'
urllib/urllib2.unquote(s)	与quote相反
urllib.urlencode(query, doseq=0)	将序列中的两个元素（元组或字典）转换为URL查询字符串
urllib.urlretrieve(url, filename=None, reporthook=None, data=None)	将返回结果保存到文件，filename是文件名
urllib2.Request(url, data=None, headers={}, origin_req_host=None, unverifi	一般访问URL用urllib.urlopen()，如果要修改header信息就会用到这个。 data是随着URL提交的数据，将会把HTTP请求GET改为POST。headers是一个字典，包含提交头的键值对应内容。

able=False)	
urllib2.urlopen(url, data=None, timeout=<object object>)	timeout 超时时间，单位秒
urllib2.build_opener(*handlers)	构造opener
urllib2.install_opener(opener)	把新构造的opener安装到默认的opener中，以后urlopen()会自动调用
urllib2.HTTPCookieProcessor(cookiejar=None)	Cookie处理器
urllib2.HTTPBasicAuthHandler	认证处理器
urllib2.ProxyHandler	代理处理器

urllib.urlopen()有几个常用的方法：

方法	描述
getcode()	获取HTTP状态码
geturl()	返回真实URL。有可能URL3xx跳转，那么这个将获得跳转后的URL
info()	返回服务器返回的header信息。可以通过它的方法获取相关值
next()	获取下一行，没有数据抛出异常
read(size=-1)	默认读取所有内容。size正整数指定读取多少字节
readline(size=-1)	默认读取下一行。size正整数指定读取多少字节
readlines(sizehint=0)	默认读取所有内容，以列表形式返回。sizehint正整数指定读取多少字节

示例：

1）请求URL

```
>>> import urllib, urllib2
>>> response = urllib.urlopen("http://www.baidu.com")    # 获取的网站页面源码
>>> response.readline()
'<!DOCTYPE html>\n'
>>> response.getcode()
200
>>> response.geturl()
'http://www.baidu.com'
```

2）伪装chrome浏览器访问

```
>>> user_agent = "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) C
>>> header = {"User-Agent": user_agent}
>>> request = urllib2.Request("http://www.baidu.com", headers=header) # 也可以通过request.add_he
>>> response = urllib2.urlopen(request)
>>> response.geturl()
'https://www.baidu.com/'
>>> print respose.info()  # 查看服务器返回的header信息
Server: bfe/1.0.8.18
Date: Sat, 12 Nov 2016 06:34:54 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: close
Vary: Accept-Encoding
Set-Cookie: BAIDUID=5979A74F742651531360C08F3BE06754:FG=1; expires=Thu, 31-Dec-37 23:55:55 (
Set-Cookie: BIDUPSID=5979A74F742651531360C08F3BE06754; expires=Thu, 31-Dec-37 23:55:55 GMT;
Set-Cookie: PSTM=1478932494; expires=Thu, 31-Dec-37 23:55:55 GMT; max-age=2147483647; path=/; (
Set-Cookie: BDSVRTM=0; path=/
Set-Cookie: BD_HOME=0; path=/
Set-Cookie: H_PS_PSSID=1426_18240_17945_21118_17001_21454_21408_21394_21377_21525_21192; path=/;
P3P: CP=" OTI DSP COR IVA OUR IND COM "
Cache-Control: private
Cxy_all: baidu+a24af77d41154f5fc0d314a73fd4c48f
Expires: Sat, 12 Nov 2016 06:34:17 GMT
X-Powered-By: HPHP
X-UA-Compatible: IE=Edge,chrome=1
Strict-Transport-Security: max-age=604800
BDPAGETYPE: 1
BDQID: 0xf51e0c970000d938
BDUSERID: 0
Set-Cookie: __bsi=12824513216883597638_00_24_N_N_3_0303_C02F_N_N_N_0; expires=Sat, 12-Nov-16 06:
```

这里header只加了一个User-Agent，防止服务器当做爬虫屏蔽了，有时为了对付防盗链也会加Referer，说明是本站过来的请求。还有跟踪用户的cookie。

3）提交用户表单

```
>>> post_data = {"loginform-username":"test","loginform-password":"123456"}
>>> response = urllib2.urlopen("http://home.51cto.com/index", data=(urllib.urlencode(post_data)))
>>> response.read() # 登录后网页内容
```

提交用户名和密码表单登录到51cto网站，键是表单元素的id。其中用到了urlencode()方法，上面讲过是用于转为字典格式为URL接受的编码格式。

例如：

```
>>> urllib.urlencode(post_data)
'loginform-password=123456&loginform-username=test'
```

4）保存cookie到变量中

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
```

```
import urllib, urllib2
import cookielib
# 实例化CookieJar对象来保存cookie
cookie = cookielib.CookieJar()
# 创建cookie处理器
handler = urllib2.HTTPCookieProcessor(cookie)
# 通过handler构造opener
opener = urllib2.build_opener(handler)
response = opener.open("http://www.baidu.com")
for item in cookie:
    print item.name, item.value

# python test.py
BAIDUID EB4BF619C95630EFD619B99C596744B0:FG=1
BIDUPSID EB4BF619C95630EFD619B99C596744B0
H_PS_PSSID 1437_20795_21099_21455_21408_21395_21377_21526_21190_21306
PSTM 1478936429
BDSVRTM 0
BD_HOME 0
```

urlopen()本身就是一个opener，无法满足对Cookie处理，所有就要新构造一个opener。

这里用到了cookielib库，cookielib库是一个可存储cookie的对象。CookieJar类来捕获cookie。

cookie存储在客户端，用来跟踪浏览器用户身份的会话技术。

5）保存cookie到文件

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import urllib, urllib2
import cookielib
cookie_file = 'cookie.txt'
# 保存cookie到文件
cookie = cookielib.MozillaCookieJar(cookie_file)
# 创建cookie处理器
handler = urllib2.HTTPCookieProcessor(cookie)
# 通过handler构造opener
opener = urllib2.build_opener(handler)
response = opener.open("http://www.baidu.com")
# 保存
cookie.save(ignore_discard=True, ignore_expires=True)  # ignore_discard默认是false，不保存将被丢失

# python test.py
# cat cookie.txt

# Netscape HTTP Cookie File
# http://curl.haxx.se/rfc/cookie_spec.html
# This is a generated file!  Do not edit.
.baidu.com    TRUE    /    FALSE    3626420835    BAIDUID    687544519EA906BD0DE5AE02FB25A
.baidu.com    TRUE    /    FALSE    3626420835    BIDUPSID    687544519EA906BD0DE5AE02FB25
.baidu.com    TRUE    /    FALSE        H_PS_PSSID    1420_21450_21097_18560_21455_21408_21395_2
.baidu.com    TRUE    /    FALSE    3626420835    PSTM    1478937189
www.baidu.com    FALSE    /    FALSE        BDSVRTM    0
www.baidu.com    FALSE    /    FALSE        BD_HOME    0
```

MozillaCookieJar()这个类用来保存cookie到文件。

6）使用cookie访问URL

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import urllib2
import cookielib
# 实例化对象
```

```
cookie = cookielib.MozillaCookieJar()
# 从文件中读取cookie
cookie.load("cookie.txt", ignore_discard=True, ignore_expires=True)
# 创建cookie处理器
handler = urllib2.HTTPCookieProcessor(cookie)
# 通过handler构造opener
opener = urllib2.build_opener(handler)
# request = urllib2.Request("http://www.baidu.com")
response = opener.open("http://www.baidu.com")
```

7）使用代理服务器访问URL

```
import urllib2
proxy_address = {"http": "http://218.17.252.34:3128"}
handler = urllib2.ProxyHandler(proxy_address)
opener = urllib2.build_opener(handler)
response = opener.open("http://www.baidu.com")
print response.read()
```

8）URL访问认证

```
import urllib2
auth = urllib2.HTTPBasicAuthHandler()
# (realm, uri, user, passwd)
auth.add_password(None, 'http://www.example.com','user','123456')
opener = urllib2.build_opener(auth)
response = opener.open('http://www.example.com/test.html')
```

10.14 json

JSON是一种轻量级数据交换格式，一般API返回的数据大多是JSON、XML，如果返回JSON的话，将获取的数据转换成字典，方面在程序中处理。

json库经常用的有两种方法dumps和loads()：

```
# 将字典转换为JSON字符串

>>> dict = {'user':[{'user1': 123}, {'user2': 456}]}

>>> type(dict)

<type 'dict'>

>>> json_str = json.dumps(dict)

>>> type(json_str)

<type 'str'>

# 把JSON字符串转换为字典

>>> d = json.loads(json_str)

>>> type(d)

<type 'dict'>
```

JSON与Python解码后数据类型：

JSON	Python
object	dict
array	list
string	unicode
number（int）	init，long

number（real）	float
true	Ture
false	False
null	None

10.15 time

这个time库提供了各种操作时间值。

方法	描述	示例
time.asctime([tuple])	将一个时间元组转换成一个可读的24个时间字符串	>>> time.asctime(time.localtime()) 'Sat Nov 12 01:19:00 2016'
time.ctime(seconds)	字符串类型返回当前时间	>>> time.ctime() 'Sat Nov 12 01:19:32 2016'
time.localtime([seconds])	默认将当前时间转换成一个(struct_timetm_year,tm_mon,tm_mday,tm_hour,tm_min, tm_sec,tm_wday,tm_yday,tm_isdst)	>>> time.localtime() time.struct_time(tm_year=2016, tm_mon=11, tm_mday=12, tm_hour=1, tm_min=19, tm_sec=56, tm_wday=5, tm_yday=317, tm_isdst=0)
time.mktime(tuple)	将一个struct_time转换成时间戳	>>> time.mktime(time.localtime()) 1478942416.0
time.sleep(seconds)	延迟执行给定的秒数	>>> time.sleep(1.5)
time.strftime(format[,tuple])	将元组时间转换成指定格式。[tuple]不指定默认以当前时间	>>> time.strftime('%Y-%m-%d %H:%M:%S') '2016-11-12 01:20:54'
time.time()	返回当前时间时间戳	>>> time.time() 1478942466.45977

strftime():

指令	描述
%a	简化星期名称，如Sat
%A	完整星期名称，如Saturday
%b	简化月份名称，如Nov
%B	完整月份名称，如November
%c	当前时区日期和时间
%d	天
%H	24小时制小时数（0-23）
%I	12小时制小时数（01-12）

%j	365天中第多少天
%m	月
%M	分钟
%p	AM或PM，AM表示上午，PM表示下午
%S	秒
%U	一年中第几个星期
%w	星期几
%W	一年中第几个星期
%x	本地日期，如'11/12/16'
%X	本地时间，如'17:46:20'
%y	简写年名称，如16
%Y	完整年名称，如2016
%Z	当前时区名称（PST：太平洋标准时间）
%%	代表一个%号本身

10.16 datetime

datetime库提供了以下几个类：

类	描述
datetime.date()	日期，年月日组成
datetime.datetime()	包括日期和时间
datetime.time()	时间，时分秒及微秒组成
datetime.timedelta()	时间间隔
datetime.tzinfo()	

datetime.date()类：

方法	描述	描述
date.max	对象所能表示的最大日期	datetime.date(9999, 12, 31)
date.min	对象所能表示的最小日期	datetime.date(1, 1, 1)
date.strftime()	根据datetime自定义时间格式	>>> date.strftime(datetime.now(), '%Y-%m-%d %H:%M:%S') '2016-11-12 07:24:15

date.today()	返回当前系统日期	>>> date.today() datetime.date(2016, 11, 12)
date.isoformat()	返回ISO 8601格式时间（YYYY-MM-DD）	>>> date.isoformat(date.today()) '2016-11-12'
date.fromtimestamp()	根据时间戳返回日期	>>> date.fromtimestamp(time.time()) datetime.date(2016, 11, 12)
date.weekday()	根据日期返回星期几，周一是0，以此类推	>>> date.weekday(date.today()) 5
date.isoweekday()	根据日期返回星期几，周一是1，以此类推	>>> date.isoweekday(date.today()) 6
date.isocalendar()	根据日期返回日历（年，第几周，星期几）	>>> date.isocalendar(date.today()) (2016, 45, 6)

datetime.datetime()类：

方法	描述	示例
datetime.now()/datetime.today()	获取当前系统时间	>>> datetime.now() datetime.datetime(2016, 11, 12, 7, 39, 35, 106385)
date.isoformat()	返回ISO 8601格式时间	>>> datetime.isoformat(datetime.now()) '2016-11-12T07:42:14.250440'
datetime.date()	返回时间日期对象，年月日	>>> datetime.date(datetime.now()) datetime.date(2016, 11, 12)
datetime.time()	返回时间对象，时分秒	>>> datetime.time(datetime.now()) datetime.time(7, 46, 2, 594397)
datetime.utcnow()	UTC时间，比中国时间快8个小时	>>> datetime.utcnow() datetime.datetime(2016, 11, 12, 15, 47, 53, 514210)

datetime.time()类：

方法	描述	示例
time.max	所能表示的最大时间	>>> time.max datetime.time(23, 59, 59, 999999)
time.min	所能表示的最小时间	>>> time.min datetime.time(0, 0)
time.resol	时间最小单位，1微妙	>>> time.resolution datetime.timedelta(0, 0, 1)

ution		

datetime.timedelta()类：

```
# 获取昨天日期
>>> date.today() - timedelta(days=1)
datetime.date(2016, 11, 11)
>>> date.isoformat(date.today() - timedelta(days=1))
'2016-11-11'
# 获取明天日期
>>> date.today() + timedelta(days=1)
datetime.date(2016, 11, 13)
>>> date.isoformat(date.today() + timedelta(days=1))
'2016-11-13'
```


python 标准库 Python基础教程

6 收藏 分享

上一篇：第九章 Python自定义模块及... 下一篇：第十一章 Python常用内建...

李振良OK
157篇文章，176W+人气，675粉丝
热爱运维工作，喜欢技术交流、分享。

关注

提问和评论都可以，用心的回复会被更多人看到和认可

Ctrl+Enter 发布 取消 发布

3条评论 按时间正序 | 按时间倒序

shouhou2581314
1楼 2016-11-15 13:56:44
振良兄辛苦，总结了这么多

于学康
2楼 2016-12-09 11:20:06
真棒，万分感谢！

时崎小三
3楼 2018-01-30 18:55:58
报道！下一篇