知识点

os:和操作系统相关

sys:和解释器相关

json:和操作JSON(一种数据交换格式)相关

pickle:序列化

hashlib:加密算法

Collections:集合类型

os:操作系统接口

此模块提供了灵活的和操作系统相关的函数.

- `os.remove(file_path)`:删除文件
- `os.rmdir(dir_path)`:删除空文件夹
  - 删除非空文件夹使用另一个模块:shutil
  - `shutil.rmtree(path)`
- `os.removedirs(name)`:递归删除空文件夹
- `os.rename(src, dst)`:文件,目录重命名,目标不能事先存在.

举例:

```
import os
os.remove('a.txt')
os.rmdir('aa')          # 只能删除空目录
os.removedirs('bb')
os.rename('abc','def')
```

使用shutil模块删除非空目录.

```
import shutil
shutil.rmtree('aa') # 可以删除带内容目录
```

和路径相关的属性,更多相关的操作被封装在os.path这个模块中.

- `os.curdir`:当前路径
- `os.sep`:路径分隔符
- `os.altsep`:备用的分隔符
- `os.extsep`:扩展名分隔符
- `os.pathsep`:路径分隔符
- `os.linesep`:行分隔符,不要在写文件的时候,使用这个属性.

os.**linesep**

The string used to separate (or, rather, terminate) lines on the current platform. This may be a single character, such as `'\n'` for POSIX, or multiple characters, for example, `'\r\n'` for Windows. Do not use *os.linesep* as a line terminator when writing files opened in text mode (the default); use a single `'\n'` instead, on all platforms.

举例：

```
import os
print(os.curdir)        # .
print(os.sep)           # \
print(os.altsep)        # /
print(os.extsep)        # .
print(os.pathsep)       # ;
```

os.path模块

此模块实现了一些在路径操作上的方法.

获取功能

- `os.path.abspath(path)`:返回一个路径的绝对路径.

  如果参数路径是相对的路径,就把当前路径和参数路径的组合字符串当成结果返回.

  如果参数路径已经是绝对路径,就直接把参数返回.

  如果参数路径以/开始,则把当前盘符和参数路径连接起来组成字符串返回.

  注意:

  此方法只是简单的将一个拼接好的字符串返回,并不会去检查这个字符串表示的文件是否存在.

样例：

```
import os
print(os.path.abspath('aa'))            # D:\PycharmProjects\test03\test01\aa
print(os.path.abspath('D:/test/aa'))    # D:\test\aa
print(os.path.abspath('/bb'))           # D:\bb
```

- `os.path.split(path)`:返回一个元组,第二个元素表示的是最后一部分的内容,第一个元素表示的是剩余的内容.

  如果只是一个盘符或者是以路径分隔符结尾的字符串,则第二个元素为空.

  否则第二个元素就是最后一部分的内容.

  如果path中不包含路径分隔符,则第一个元素为空.

样例：

```
import os
print(os.path.split('D:/'))     # ('D:/', '')
print(os.path.split('.'))       # ('', '.')
print(os.path.split('/aa'))     # ('/', 'aa')
```

- `os.path.basename(path)`:返回path指定的路径的最后一个内容.

  如果只是一个盘符,或者是以路径分隔符结尾的字符串,则返回空;

  否则返回的是路径中的最后一部分内容.

样例：

```
import os
print(os.path.basename('.'))        # .
print(os.path.basename('/aa'))      # aa
print(os.path.basename('/aa/'))     #
print(os.path.basename('D:/test'))  # test
```

- `os.path.dirname(path)`:返回一个路径中的父目录部分.

  如果只是一个盘符,或者是以路径分隔符结尾的字符串,则整体返回.

  否则返回的是路径中的父目录部分.

样例：

```
import os
print(os.path.dirname('.'))         #
print(os.path.dirname('/aa/'))      # /aa
print(os.path.dirname('D:/test'))   # D:/
print(os.path.dirname('D:/'))       # D:/
```

- `os.path.getsize(path)`:获取文件的字节数.如果是文件夹,返回0或者是一个不准确的值.

样例：

```
print(os.path.getsize('aa'))            # 0
print(os.path.getsize('.'))             # 4096
print(os.path.getsize('aa/test.txt'))   # 6
```

- `os.path.join(path,*paths)`:连接若干个路径为一个路径.

  如果路径中有绝对路径,则在这个路径之前的路径都会被丢弃,而从这个路径开始往后拼接.

  Windows中盘符一定要带\,否则不认为是一个盘符.

样例：

```python
res = os.path.join('aa','bb','cc')
print(res)          # aa\bb\cc

res2 = os.path.join('D:/','test')
print(res2)         # D:/test
```

判断功能

- `os.path.exists(path)`:判断路径是否真正存在.
- `os.path.isabs(path)`:判断是否是绝对路径
- `os.path.isfile(path)`:判断是否是文件
- `os.path.isdir(path)`:判断是否是目录

sys:和解释器相关

提供了解释器使用和维护的变量和函数.

- `sys.argv`:当以脚本方式执行程序时,从命令行获取参数.

  argv[0]表示的是当前正在执行的脚本名.argv[1]表示第一个参数,以此类推.

样例:

有脚本`test.py`内容如下:

```python
import sys
print('脚本名称:',sys.argv[0])
print('第一个参数是:',sys.argv[1])
print('第二个参数是:',sys.argv[2])
```

使用命令行方式运行该脚本:`python test.py hello world`

```
D:\>python test.py hello world
脚本名称: test.py
第一个参数是: hello
第二个参数是: world
```

- `sys.getrefcount(object)`:返回一个对象的引用的次数.总比真实引用的次数多1.

样例:

```python
import sys
print(sys.getrefcount(0))   # 168
print(sys.getrefcount(1))   # 97
```

- `sys.getrecursionlimit()`:递归的最大次数

- `sys.setrecursionlimit(limit)`:设置递归的最大次数.

样例:

```
import sys
print(sys.getrecursionlimit())      # 1000
sys.setrecursionlimit(2000)
print(sys.getrecursionlimit())      # 2000
```

- `sys.modules`:返回系统已经加载的模块,以字典形式返回.

  对这个字典中的值进行修改并没有什么具体意义.反而有时会引发异常.

  常用来作为是否重新加载一个模块的判断依据.

- `sys.path`:系统寻找模块的路径.可以通过PYTHONPATH来进行初始化.

  由于是在程序执行的时候进行初始化的,所以,路径的第一项path[0]始终是调用解释器的脚本所在的路径.如果是动态调用的脚本,或者是从标准输入读取到脚本命令,则path[0]是一个空字符串.程序中可以随时对这个路径进行修改.以达到动态添加模块路径的目的.

json模块

`JSON:JavaScript Object Notation`,JS对象标记.

原本是在JavaScript中对对象标记的一种方式,现在已经变成被大多数语言广泛应用的一种数据交换格式.

JSON格式的数据本身就是一个字符串,JSON文件就是一个文本文件.

它把其他数据格式统统转换成字符串,进而可以保存在文件中或者是通过网络进行传输.另一方面,从JSON文件中读取或者是网络接收端很容易将得到的字符串还原成原来的数据格式.

通过字符串这种非常通用的格式,完成了多种数据格式的交换.

序列化和反序列化

将不同数据类型转换成json字符串的过程称为序列化,反之称为反序列化.

并不是所有类型的数据都能被序列化成json字符串的.

以下是Python中的可以被序列化的数据类型：

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
True	true
False	false
None	null

从上表中可以看出，Python中的`set`，`frozenset`等数据类型，就没有办法转换成json字符串.

Python中提供的json模块就是专门用来操作JSON格式的文件的.

json模块中的方法

- 将其他数据类型序列化成json字符串
- 将json字符串反序列化成其他数据类型

json字符串在进行反序列化时，也遵循一定的对应关系：

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

从上图可以看出，json中的array类型转换成的都是Python中的list.这也就意味着，元组经过json序列化和反序列化之后，将会变成列表.

- 将其他数据类型序列化成json字符串并写入文件
- 从文件中读取json字符串并反序列化成其他数据类型

需要注意的是：

不要尝试多次将json字符串写入同一个文件.

或者手动修改已经写入json字符串的文件.都容易导致反序列化失败.

通常json文件用来保存一些配置信息,这样的配置信息不会太大.通过一次写,一次读,完全可以满足数据交换的需求.

pickle:序列化

pickle是Python中提供的另一种序列化的模块.

但是pickle和json有着多方面的不同:

json更加通用,pickle更有针对性.

pickle模块中的方法

pickle模块中的方法基本和json模块中的方法一样.只是适用的数据类型范围比json更广泛.

- 将其他数据类型序列化成字节
- 将字节反序列化成其他数据类型
- 将其他数据类型序列化成字节并写入文件
- 从文件中读取字节并反序列化成其他数据类型

样例:

```
import pickle
bys1 = pickle.dumps(10)
print(bys1)                 # b'\x80\x03K\n.'
value = pickle.loads(bys1)
print(value)                # 10

bys2 = pickle.dumps(True)
print(pickle.loads(bys2))      # True

bys3 = pickle.dumps('abc')
print(pickle.loads(bys3))       # abc

bys4 = pickle.dumps([1,2,3])
print(pickle.loads(bys4))           # [1, 2, 3]

bys5 = pickle.dumps((1,2,3))
```

```python
print(pickle.loads(bys5))           # (1, 2, 3)

bys6 = pickle.dumps(set('abc'))
print(pickle.loads(bys6))           # {'b', 'a', 'c'}

bys7 = pickle.dumps({'name':'Andy','age':10})
print(pickle.loads(bys7))           # {'name': 'Andy', 'age': 10}
```

```python
import pickle
# 文件操作
with open('pickle.data',mode='wb') as f:
    pickle.dump({'name':'Andy','age':10},f)

with open('pickle.data',mode='rb') as f:
    d = pickle.load(f)
    print(d)                    # {'name':'Andy','age':10}
```

hashlib:加密模块

hashlib模块中提供的类采用的是单向加密算法,也称'哈希算法','摘要算法'.

这种算法的特点是:

- 相同的数据切分成若干个小的部分,对这些小部分分别加密和一次性对整个数据加密的结果是一致的.
- 原始数据的一点小不同,将会导致最终结果的非常大的差异.
- 从加密后的结果反推原始数据几乎是不可能的.

使用摘要算法加密一个数据的三大步骤:

- 获取一个加密算法对象.
- 调用加密对象的update方法给指定的数据加密.
- 调用加密对象的digest或者是hexdigest获取加密后的结果.

样例:

```python
import hashlib
m = hashlib.md5()       # 获取一个假面对象
m.update(b'abc')        # 对参数进行加密,参数必须是字节类型
res = m.hexdigest()     # 获取加密后的字符串
print(res)              # 900150983cd24fb0d6963f7d28e17f72
# 字节形式的结果
print(m.digest())       # b'\x90\x01P\x98<\xd2O\xb0\xd6\x96?}(\xe1\x7fr'
```

简化写法

加密对象的update方法可以调用多次,意味着在在前一次的update结果之上,再次进行加密.如果只是一次更新的话,还可以直接把数据当成参数传递给构造方法.

例如:

```
m = md5()
m.update(data)
res = m.hexdigest()
```

和下面的语句是等价的:

```
res = md5(data).hexdigest()
```

这种在创建加密对象的时候,就指定初始化的数据,称为`salt`(盐).

目的就是为了让加密的结果更加复杂.

其他加密对象

加密对象除了md5之外,还有如下几种:

`'sha1', 'sha224', 'sha256', 'sha384', 'sha3_224', 'sha3_256', 'sha3_384', 'sha3_512', 'sha512', 'shake_128', 'shake_256'`

这些加密对象之间的区别就是加密算法结果的长度不同.加密结果的长度越长,算法越复杂,相应的,加密耗费的时间也越长.

例如:

```
import hashlib
print(len(hashlib.md5().hexdigest()))       # 32
print(len(hashlib.sha1().hexdigest()))      # 40
print(len(hashlib.sha512().hexdigest()))    # 128
```

练习

把用户名和密码信息加密后,通过序列化的方式存储到本地文件中.

并通过控制台输入信息进行验证.

```
import hashlib
import json

# 对用户名和密码进行加密

def get_md5(username,password):
    encrypt = hashlib.md5(username.encode('utf-8'))
```

```
    encrypt.update(password.encode('utf-8'))
    return encrypt.hexdigest()

username = 'Andy'
password = '123'

# 将用户信息保存到文件中:

# 格式:用户名|用户名和密码加密后的字符串

with open('login.info',mode='wt',encoding='utf-8') as f:
    json.dump(username + '|' + get_md5(username,password),f)

# 验证

name = input('input username:')
passwd = input('input password:')

# 反序列化出info

with open('login.info',mode='rt',encoding='utf-8') as f:
    info = json.load(f)

username,password = info.split('|')

if name == username and get_md5(name,passwd) == password:
    print('登录成功')
else:
    print('登录失败')
```

Collections:集合

此模块定义了一些内置容器类数据类型之外,可用的集合类数据类型.

往往针对的是特殊的应用场景.

namedtuple:元组的工厂函数

所谓的工厂函数指的是:接收类名和一些创建此类对象所需要的一些参数,返回指定类的一个对象.

命名元组的特点是给元素绑定了一个有意义的名字.在使用元素时可以使用元素的名字而不必使用索引.

函数的签名是:

```
collections.namedtuple(typename, field_names, *, verbose=False, rename=False,
module=None)
```

此函数的返回值是一个tuple的子类定义.常用的是前两个参数.

第一个参数指定的是返回的子类的类名.

第二个参数指定的是子类元组可以拥有的元素名.以字符串组成的列表表示,或者是以空格,逗号分隔的单个字符串都可以.

例如:

使用`namedtuple`创建一个名为:`Rectangle`的`tuple`子类.

并用`Rectangle`创建一个实例对象,使用其中的元素.

```
from collections import namedtuple
Rectangle = namedtuple('Rectangle',['length','width'])      # 创建类
r = Rectangle(10,5)                 # 使用类实例化对象
print(r.length)                     # 使用元素名访问元素
print(r.width)
print(r[0])                         # 依然可以使用索引访问元素
print(r[1])
```

defaultdict:默认字典

`defaultdict`是内置的`dict`的子类.

回顾`dict`的创建方式:

```
d = {'name':'abc','age':10}
d = dict(name='abc',age=10)     # 调用dict的构造函数创建字典对象
```

`defaultdict`类的构造函数签名:

```
class collections.defaultdict([default_factory[, ...]])
```

和`dict`的构造函数相比,只是在前面添加了一个参数:`default_factory`,其余的参数和`dict`一样.

第一个参数指定的是一个函数名,用来表示当字典对象中出现了不存在的键时,对应的值初始值是如何计算.

正因为这个函数是获取值的,所以,对这个函数规定:不能有参数.

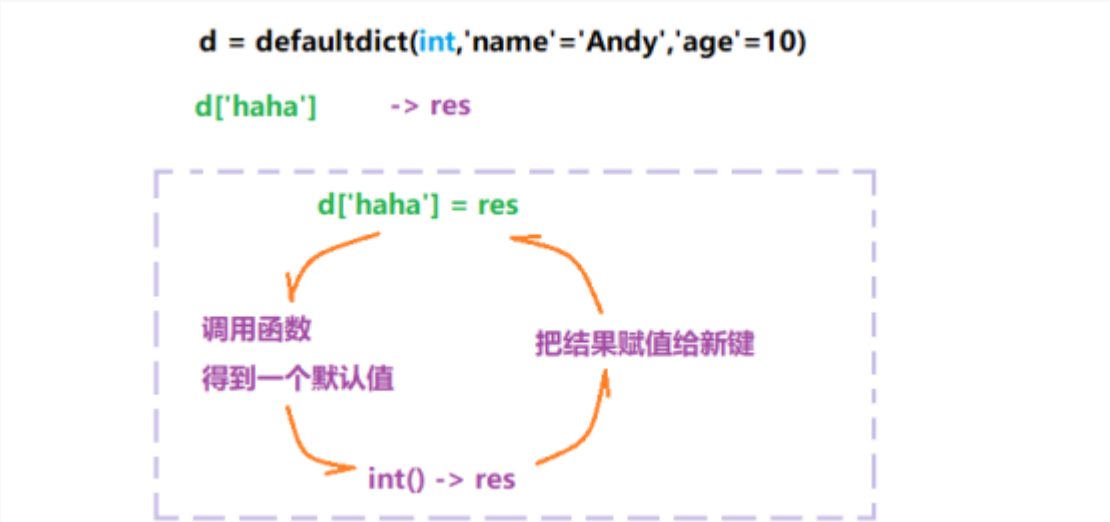
默认情况下,第一个参数是None,意味着不存在的键对应的值为None.

注意:

一旦使用`defaultdict`时,指定了不存在的键,则会引发两件事情:

- 调用第一个参数指定的函数得到默认值.
- 把返回值赋值给这个新键.

如图所示:



例如：

```
from collections import defaultdict
d = defaultdict(int,name='abc',age=10)
print(d)                # defaultdict(<class 'int'>, {'name': 'abc', 'age': 10})
print(d['name'])        # abc
print(d['age'])         # 10
print(d['haha'])        # 0
print(d)                # defaultdict(<class 'int'>, {'name': 'abc', 'age': 10, 'haha': 0})
```

从结果中可以看出，`defaultdict`新增了不存在键和默认值组成的键值对.

自定义函数充当第一个参数：

```
from collections import defaultdict
def func():
    return 'hello world'

d = defaultdict(func,name='Andy',age=10)
print(d['name'])        # Andy
print(d['haha'])        # hello world
print(d)
# defaultdict(<function func at 0x000002C9A0D71E18>, {'name': 'Andy', 'age': 10, 'haha': 'hello world'})
```

Counter:生成统计信息

用于统计可哈希对象的数量.

是dict的子类.一种特殊的字典.

它的键是可哈希对象，值是这个对象的个数统计信息.个数可以是手动指定的，也可以是自动计算出来的，并且可以是负数和0.

创建Counter对象.

```
c = Counter()                           # 创建空的计数器
c = Counter('abcdefabc')                # 使用可迭代对象创建计数器
c = Counter({'A':1,'B':3})              # 使用字典创建计数器
c = Counter(A=1,B=3,C=0)                # 手动初始化计数器
```

查看计数器的统计结果，和使用字典的方式相同：

```
from collections import Counter
c = Counter('abcab')
print(c)            # Counter({'a': 2, 'b': 2, 'c': 1})
print(c['a'])       # 2
print(c['b'])       # 2
print(c['name'])    # 0
```

Counter常用方法：

`most_common([n])`:显示数量最多的前几名.

例如：

```
from collections import Counter
c = Counter('abcab')
print(c.most_common(2)) # [('a', 2), ('b', 2)]
```