json使用方法

JSON-是一个轻量级的数据交换格式。点击打开百度百科 JSON维基百科可以了解更多信息。

关于json的官方文档，请点击这里查看。

对简单数据类型的encoding 和 decoding

使用简单的json.dumps方法对简单数据类型进行编码，例如：

```
import json

obj = [[1,2,3],123,123.123,'abc',{'key1':(1,2,3),'key2':(4,5,6)}]
encodedjson = json.dumps(obj)
print repr(obj)
print encodedjson
```

输出：

```
[[1, 2, 3], 123, 123.123, 'abc', {'key2': (4, 5, 6), 'key1': (1, 2, 3)}]
[[1, 2, 3], 123, 123.123, "abc", {"key2": [4, 5, 6], "key1": [1, 2, 3]}]
```

通过输出的结果可以看出，简单类型通过encode之后跟其原始的repr()输出结果非常相似，但是有些数据类型进行了改变，例如上例中的元组则转换为了列表。在json的编码过程中，会存在从python原始类型向json类型的转化过程，具体的转化对照如下：

Python	JSON
dict	object
list, tuple	array
str, unicode	string
int, long, float	number
True	true
False	false
None	null

json.dumps()方法返回了一个str对象encodedjson，我们接下来在对encodedjson进行decode，得到原始数据，需要使用的json.loads()函数：

```
decodejson = json.loads(encodedjson)
print type(decodejson)
print decodejson[4]['key1']
print decodejson
```

输出：

```
<type 'list'>
[1, 2, 3]
[[1, 2, 3], 123, 123.123, u'abc', {u'key2': [4, 5, 6], u'key1': [1, 2, 3]}]
```

loads方法返回了原始的对象，但是仍然发生了一些数据类型的转化。比如，上例中‘abc’转化为了unicode类型。从json到python的类型转化对照如下：

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

json.dumps方法提供了很多好用的参数可供选择，比较常用的有sort_keys（对dict对象进行排序，我们知道默认dict是无序存放的），separators，indent等参数。

排序功能使得存储的数据更加有利于观察，也使得对json输出的对象进行比较，例如：

```
data1 = {'b':789,'c':456,'a':123}
data2 = {'a':123,'b':789,'c':456}
d1 = json.dumps(data1,sort_keys=True)
d2 = json.dumps(data2)
d3 = json.dumps(data2,sort_keys=True)
print d1
print d2
print d3
print d1==d2
print d1==d3
```

输出：

```
{"a": 123, "b": 789, "c": 456}
{"a": 123, "c": 456, "b": 789}
{"a": 123, "b": 789, "c": 456}
False
True
```

上例中，本来data1和data2数据应该是一样的，但是由于dict存储的无序特性，造成两者无法比较。

因此两者可以通过排序后的结果进行存储就避免了数据比较不一致的情况发生，但是排序后再进行存储，系统必定要多做一些事情，也一定会因此造成一定的性能消耗，所以适当排序是很重要的。

indent参数是缩进的意思，它可以使得数据存储的格式变得更加优雅。

```
ata1 = {'b':789,'c':456,'a':123}
d1 = json.dumps(data1,sort_keys=True,indent=4)
print d1
```

输出：

```
{
    "a": 123,
    "b": 789,
    "c": 456
}
```

输出的数据被格式化之后，变得可读性更强，但是却是通过增加一些冗余的空白格来进行填充的。json主要是作为一种数据通信的格式存在的，而网络通信是很在乎数据的大小的，无用的空格会占据很多通信带宽，所以适当时候也要对数据进行压缩。separator参数可以起到这样的作用，该参数传递是一个元组，包含分割对象的字符串。

```
print 'DATA:', repr(data)
print 'repr(data)             :', len(repr(data))
print 'dumps(data)            :', len(json.dumps(data))
print 'dumps(data, indent=2)  :', len(json.dumps(data, indent=4))
print 'dumps(data, separators):', len(json.dumps(data, separators=(',',':')))
```

输出：

```
DATA: {'a': 123, 'c': 456, 'b': 789}
repr(data)             : 30
dumps(data)            : 30
dumps(data, indent=2)  : 46
dumps(data, separators): 25
```

通过移除多余的空白符，达到了压缩数据的目的，而且效果还是比较明显的。

另一个比较有用的dumps参数是skipkeys，默认为False。 dumps方法存储dict对象时，key必须是str类型，如果出现了其他类型的话，那么会产生TypeError异常，如果开启该参数，设为True的话，则会比较优雅的过度。

```
data = {'b':789,'c':456,(1,2):123}
print json.dumps(data,skipkeys=True)
```

输出：

```
{"c": 456, "b": 789}
```

处理自己的数据类型

json模块不仅可以处理普通的python内置类型，也可以处理我们自定义的数据类型，而往往处理自定义的对象是很常用的。

首先，我们定义一个类Person。

```
class Person(object):
def __init__(self,name,age):
    self.name = name
    self.age = age
def __repr__(self):
    return 'Person Object name : %s , age : %d' % (self.name,self.age)
if __name__  == '__main__':
p = Person('Peter',22)
print p
```

如果直接通过json.dumps方法对Person的实例进行处理的话，会报错，因为json无法支持这样的自动转化。通过上面所提到的json和python的类型转化对照表，可以发现，object类型是和dict相关联的，所以我们需要把我们自定义的类型转化为dict，然后再进行处理。这里，有两种方法可以使用。

方法一：自己写转化函数

```
'''
Created on 2016-10-11
@author: Yanghui
'''
import Person
import json

p = Person.Person('Peter',22)

def object2dict(obj):
#convert object to a dict
d = {}
d['__class__'] = obj.__class__.__name__
d['__module__'] = obj.__module__
d.update(obj.__dict__)
return d
```

```
def dict2object(d):
#convert dict to object
if'__class__' in d:
    class_name = d.pop('__class__')
    module_name = d.pop('__module__')
    module = __import__(module_name)
    class_ = getattr(module,class_name)
    args = dict((key.encode('ascii'), value) for key, value in d.items()) #get args
    inst = class_(**args) #create new instance
else:
    inst = d
return inst

d = object2dict(p)
print d
#{'age': 22, '__module__': 'Person', '__class__': 'Person', 'name': 'Peter'}

o = dict2object(d)
print type(o),o
#<class 'Person.Person'> Person Object name : Peter , age : 22

dump = json.dumps(p,default=object2dict)
print dump
#{"age": 22, "__module__": "Person", "__class__": "Person", "name": "Peter"}

load = json.loads(dump,object_hook = dict2object)
print load
#Person Object name : Peter , age : 22
```

上面代码已经写的很清楚了，实质就是自定义object类型和dict类型进行转化。object2dict函数将对象模块名、类名以及**dict**存储在dict对象里，并返回。dict2object函数则是反解出模块名、类名、参数，创建新的对象并返回。在json.dumps 方法中增加default参数，该参数表示在转化过程中调用指定的函数，同样在decode过程中json.loads方法增加object_hook,指定转化函数。

方法二：继承JSONEncoder和JSONDecoder类，覆写相关方法

JSONEncoder类负责编码，主要是通过其default函数进行转化，我们可以override该方法。同理对于JSONDecoder。

```
'''
Created on 2016-10-11
@author: Yanghui
'''
import Person
import json

p = Person.Person('Peter',22)
```

```
class MyEncoder(json.JSONEncoder):
    def default(self,obj):
        #convert object to a dict
        d = {}
        d['__class__'] = obj.__class__.__name__
        d['__module__'] = obj.__module__
        d.update(obj.__dict__)
        return d

class MyDecoder(json.JSONDecoder):
    def __init__(self):
        json.JSONDecoder.__init__(self,object_hook=self.dict2object)
    def dict2object(self,d):
        #convert dict to object
        if'__class__' in d:
            class_name = d.pop('__class__')
            module_name = d.pop('__module__')
            module = __import__(module_name)
            class_ = getattr(module,class_name)
            args = dict((key.encode('ascii'), value) for key, value in d.items()) #get
            inst = class_(**args) #create new instance
        else:
            inst = d
        return inst


d = MyEncoder().encode(p)
o =  MyDecoder().decode(d)

print d
print type(o), o
```

对于JSONDecoder类方法，稍微有点不同，但是改写起来也不是很麻烦。看代码应该就比较清楚了。