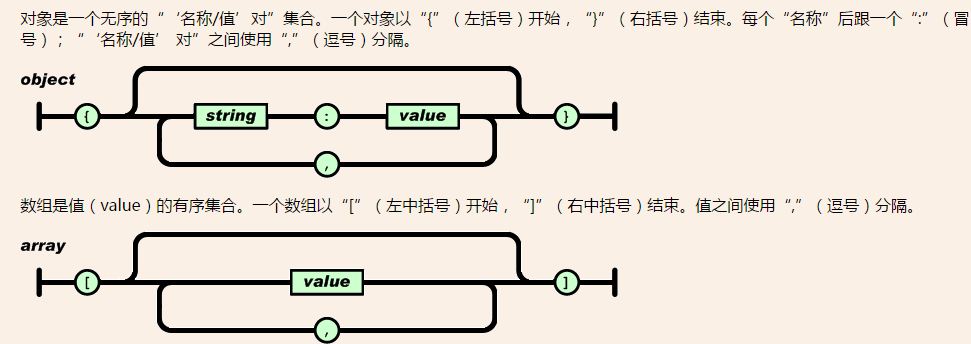
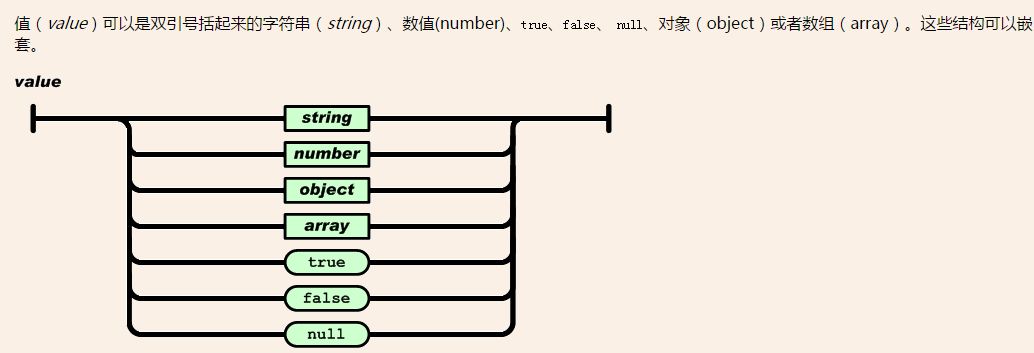
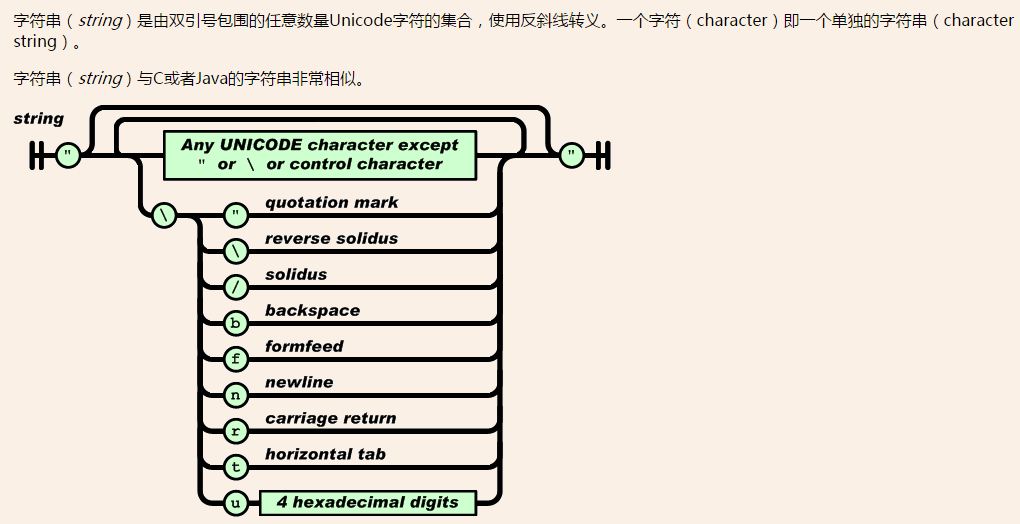
JSON(JavaScript Object Notation)是一种轻量级的数据交换格式，易于人阅读和编写。

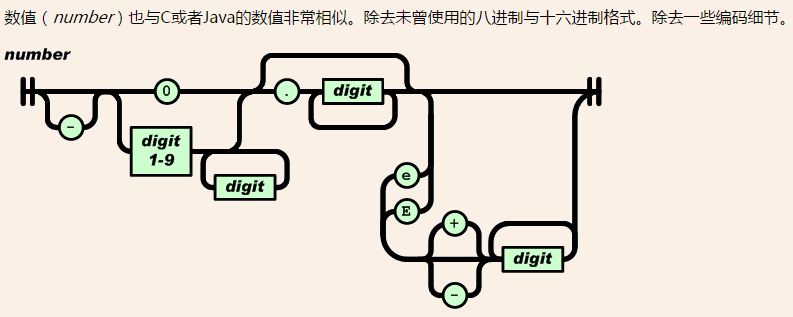
JSON建构于两种结构：

* “名称/值”对的集合。不同的语言中，它被理解为对象（object），纪录（record），结构（struct），字典（dictionary），哈希表（hash table），有键列表（keyed list），或者关联数组 （associative array）。
* 值的有序列表。在大部分语言中，它被理解为数组（array）。











## JSON 文件

JSON 文件的文件类型是 ".json"

JSON 文本的 MIME 类型是 "application/json"

## JSON函数

使用JSON函数需要导入json库：import json

|  |  |
| --- | --- |
| 函数 | 描述 |
| json.dumps | 将Python对象编码成JSON字符串 |
| json.dump | 将Pyhon对象编码成JSON对象，并写入文件 |
| json.loads | 将已编码的JSON字符串解码为Python对象 |
| json.load | 将文件中已编码的JSON字符串解码为Python对象 |

### json.dumps

用于将Python对象编码成JSON字符串。

**语法**：

dumps(obj, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, encoding='utf-8', default=None, sort\_keys=False, \*\*kw)

* skipkeys:如果skipkeys为True，如果字典的**键值**不是基础数据类型(``str``, ``unicode``, ``int``, ``long``, ``float``, ``bool``, ``None``)，就会被跳过而不是抛出``TypeError`异常。例如，key如果是个复数值，则不会抛出异常，而是直接被忽略。
* 如果“ensure\_ascii”是True(默认值)，则所有非ascii字符都是以' ' \uXXXX ' ' '的序列来转义，结果是仅由ASCII字符组成的实例。如果“ensure\_ascii”是False，所有非ascii字符都不会转义，返回值可能是一个‘‘unicode’实例。
* 如果“check\_circular”是False，那么将跳过容器类型的循环引用检查，而循环引用将导致“‘OverflowError”(或更糟)。
* 如果“allow\_nan”是False，那么它将会是一个“ValueError”，将“float”的“浮点”值(“nan”，“inf”，“inf”，“-inf”)序列化为严格的JSON规范，而不是使用JavaScript等价的(``NaN``, ``Infinity``, ``-Infinity``)。
* indent 缩进，默认为None，没有缩进，设置为正整数时，输出的格式将按照indent指定的半角空格数缩进，相当实用。
* separators 设置分隔符，默认的分隔符是(',', ': ')，如果需要自定义json中的分隔符，例如调整冒号前后的空格数，可以按照(item\_separator, key\_separator)的形式设置。
* sort\_keys 默认为False，设为True时，输出结果将按照字典中的key排序。
* default(obj) ' ' ' '是一个可以返回可序列化对象或者抛出TypeError异常的的函数，默认值只会引发TypeError异常。

注意：JSON键/值对中的键始终是类型str。因此，如果将字典转换为JSON，然后再返回到字典中，那么字典可能不等于原始字典。也就是说，如果x有非字符串键，那么load (dumps(x) != x。

>>> dumps({1:100})

'{"1": 100}'

>>> loads(dumps({1:100}))

{u'1': 100}

**实例1：**

>>> data = {1+2j: "cimplex"}

>>> dumps(data)

Traceback (most recent call last):

File "<pyshell#66>", line 1, in <module>

dumps(data)

File "C:\Python27\lib\json\\_\_init\_\_.py", line 244, in dumps

return \_default\_encoder.encode(obj)

File "C:\Python27\lib\json\encoder.py", line 207, in encode

chunks = self.iterencode(o, \_one\_shot=True)

File "C:\Python27\lib\json\encoder.py", line 270, in iterencode

return \_iterencode(o, 0)

TypeError: keys must be a string

>>> dumps(data, skipkeys=True)

'{}'

**实例2**：

>>> data = u"你好"

>>> dumps(data)

'"\\u4f60\\u597d"'

>>> dumps(data, ensure\_ascii=False)

u'"\u4f60\u597d"'

>>> data = (u'你好').encode('utf8')

>>> dumps(data)

'"\\u4f60\\u597d"'

>>> dumps(data, ensure\_ascii=False)

'"\xe4\xbd\xa0\xe5\xa5\xbd"'

**实例3：**

>>> L

[1, 2, [...]]

>>> dumps(L)

Traceback (most recent call last):

File "<pyshell#89>", line 1, in <module>

dumps(L)

File "C:\Python27\lib\json\\_\_init\_\_.py", line 244, in dumps

return \_default\_encoder.encode(obj)

File "C:\Python27\lib\json\encoder.py", line 207, in encode

chunks = self.iterencode(o, \_one\_shot=True)

File "C:\Python27\lib\json\encoder.py", line 270, in iterencode

return \_iterencode(o, 0)

ValueError: Circular reference detected

>>> dumps(L, check\_circular=True)

Traceback (most recent call last):

File "<pyshell#90>", line 1, in <module>

dumps(L, check\_circular=True)

File "C:\Python27\lib\json\\_\_init\_\_.py", line 244, in dumps

return \_default\_encoder.encode(obj)

File "C:\Python27\lib\json\encoder.py", line 207, in encode

chunks = self.iterencode(o, \_one\_shot=True)

File "C:\Python27\lib\json\encoder.py", line 270, in iterencode

return \_iterencode(o, 0)

ValueError: Circular reference detected

**实例4：**

>>> data = dict(zip(['a', 'b', 'c', 'd'], [1,2,3,4]))

>>> data

{'a': 1, 'c': 3, 'b': 2, 'd': 4}

>>> dumps(data, indent=4, separators=(',',': '))

'{\n "a": 1,\n "c": 3,\n "b": 2,\n "d": 4\n}'

>>> dumps(data, indent=4, separators=(',',':\*'))

'{\n "a":\*1,\n "c":\*3,\n "b":\*2,\n "d":\*4\n}'

>>> dumps(data, indent=4, separators=('#',':\*'))

'{\n "a":\*1#\n "c":\*3#\n "b":\*2#\n "d":\*4\n}'

>>> dumps(data, indent=4, separators=('#',':\*'), sort\_keys=True)

'{\n "a":\*1#\n "b":\*2#\n "c":\*3#\n "d":\*4\n}'

**实例5：**

#!/usr/bin/env python2

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

import json

data = [ { *'a'* : 1, *'b'* : 2, *'c'* : 3, *'d'* : 4, *'e'* : 5 } ]

data\_json = json.dumps(data)

print data\_json

print type(data\_json)

输出：

[{"a": 1, "c": 3, "b": 2, "e": 5, "d": 4}]

<type 'str'>

使用参数让JSON数据格式化输出：

print json.dumps({*'a'*: *'Runoob'*, *'b'*: 7}, sort\_keys=True, indent=4, separators=(*','*, *': '*))

输出：

{

"a": "Runoob",

"b": 7

}

**实例6**：

>>> import json

>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])

'["foo", {"bar": ["baz", null, 1.0, 2]}]'

>>> print json.dumps("\"foo\bar")

"\"foo\bar"

>>> print json.dumps(u'\u1234')

"\u1234"

>>> print json.dumps('\\')

"\\"

>>> print json.dumps({'a':0, 'd':0, 'c':0, 'b':0}, sort\_keys=True)

{"a": 0, "b": 0, "c": 0, "d": 0}

**实例7：**

class **ComplexEncoder**(json.JSONEncoder):

def **default**(*self*, obj):

if isinstance(obj, complex):

return [obj.real, obj.imag]

return json.JSONEncoder.default(*self*, obj)

print json.dumps(2+1j, cls=ComplexEncoder)

print ComplexEncoder().encode(2 + 1j)

print list(ComplexEncoder().iterencode(2 + 1j))

输出：

[2.0, 1.0]

[2.0, 1.0]

['[2.0', ', 1.0', ']']

### json.dump

dump(obj, fp, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, encoding='utf-8', default=None, sort\_keys=False, \*\*kw)

* fp:文件对象用来fp.write()进行数据写入文件

**实例1**：

with open(*'test.json'*, *'w'*) as fp:

json.dump({*'a'*: *'Runoob'*, *'b'*: 7}, fp, sort\_keys=True, indent=4, separators=(*','*, *': '*))

输出：

test.json

{

"a": "Runoob",

"b": 7

}

**实例2：**

>>> data

{'a': None, 'c': False, 'b': 0.0, 'e': [1, 2, 3, 4], 'd': u'\u4f60\u597d', 'f': {'a': 100}}

>>> fp = open(r'C:\E\dir0\1.json', 'w')

>>> dump(data, fp, indent=4, separators=(',',': '), sort\_keys=True)

>>> fp.close()

**实例:3**：

>>> from StringIO import StringIO

>>> io = StringIO()

>>> json.dump(['streaming API'], io)

>>> io.getvalue()

'["streaming API"]'

>>>

### json.load

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

从指定文件中加载一个对象，返回新构造的对象。

* 如果’fp’编码encoded是ASCII而不是utf-8，可选参数‘encoding’必须进行指定。

### json.loads

从一个字符串中加载一个对象，返回构造的对象。

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



**实例**1：

import json

data = {*'a'*: 1, *'b'* : True, *'c'* : False, *'d'* : None, *'e'* : [1, 2, 3, 4] }

with open(*'test.json'*, *'w+'*) as fp:

json.dump(data, fp, sort\_keys=True, indent=4, separators=(*','*, *': '*))

with open(*'test.json'*, *'r'*) as fp:

print json.load(fp)

执行结果：

输出：test.json文件

{

"a": 1,

"b": true,

"c": false,

"d": null,

"e": [

1,

2,

3,

4

]

}

load后结果：

{u'a': 1, u'c': False, u'b': True, u'e': [1, 2, 3, 4], u'd': None}

**实例2**：

>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')

[u'foo', {u'bar': [u'baz', None, 1.0, 2]}]

>>> json.loads('"\\"foo\\bar"')

u'"foo\x08ar'

>>> io = StringIO('["streaming API"]')

>>> json.load(io)

[u'streaming API']

>>>

**实例3**：

def **as\_complex**(dct):

if *'\_\_complex\_\_'* in dct:

return complex(dct[*'real'*], dct[*'imag'*])

return dct

print json.loads(*'{"\_\_complex\_\_": true, "real": 1, "imag": 2}'*, object\_hook=as\_complex)

输出：(1+2j)