JSON Module All Versions

Examples

Retrieving data from a file

The following snippet opens a JSON encoded file (replace `filename` with the actual name of the file) and returns the object that is stored in the file.

```
import json

with open(filename, 'r') as f:
    d = json.load(f)
```

Storing data in a file

The following snippet encodes the data stored in `d` into JSON and stores it in a file (replace `filename` with the actual name of the file).

```
import json

d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}

with open(filename, 'w') as f:
    json.dump(d, f)
```

Calling `json.tool` from the command line to pretty-print JSON output

Given some JSON file "foo.json" like:

```
{"foo": {"bar": {"baz": 1}}}
```

we can call the module directly from the command line (passing the filename as an argument) to pretty-print it:

```
$ python -m json.tool foo.json
{
    "foo": {
        "bar": {
            "baz": 1
        }
    }
}
```

The module will also take input from STDOUT, so (in Bash) we equally could do:

```
$ cat foo.json | python -m json.tool
```

Formatting JSON output

Let's say we have the following data:

```
>>> data = {"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Just dumping this as JSON does not do anything special here:

```
>>> print(json.dumps(data))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Setting indentation to get prettier output

If we want pretty printing, we can set an `indent` size:

```
>>> print(json.dumps(data, indent=2))
{
  "cats": [
    {
      "name": "Tubbs",
      "color": "white"
    },
    {
      "name": "Pepper",
      "color": "black"
    }
  ]
}
```

Sorting keys alphabetically to get consistent output

By default the order of keys in the output is undefined. We can get them in alphabetical order to make sure we always get the same output:

```
>>> print(json.dumps(data, sort_keys=True))
{"cats": [{"color": "white", "name": "Tubbs"}, {"color": "black", "name": "Pepper"}]}
```

Getting rid of whitespace to get compact output

We might want to get rid of the unnecessary spaces, which is done by setting separator strings different from the default `', '` and `': '` :

```
>>>print(json.dumps(data, separators=(',', ':')))
{"cats":[{"name":"Tubbs","color":"white"},{"name":"Pepper","color":"black"}]}
```

JSON encoding custom objects

If we just try the following:

```
import json
from datetime import datetime
data = {'datetime': datetime(2016, 9, 26, 4, 44, 0)}
print(json.dumps(data))
```

we get an error saying `TypeError: datetime.datetime(2016, 9, 26, 4, 44) is not JSON serializable` .

To be able to serialize the datetime object properly, we need to write custom code for how to convert it:

```
class DatetimeJSONEncoder(json.JSONEncoder):
    def default(self, obj):
        try:
            return obj.isoformat()
        except AttributeError:
            # obj has no isoformat method; let the builtin JSON encoder handle it
            return super(DatetimeJSONEncoder, self).default(obj)
```

and then use this encoder class instead of `json.dumps` :

```
encoder = DatetimeJSONEncoder()
print(encoder.encode(data))
# prints {"datetime": "2016-09-26T04:44:00"}
```

`load` vs `loads`, `dump` vs `dumps`

The `json` module contains functions for both reading and writing to and from unicode strings, and reading and writing to and from files. These are differentiated by a trailing `s` in the function name. In these examples we use a StringIO object, but the same functions would apply for any file-like object.

Here we use the string-based functions:

```
import json

data = {u"foo": u"bar", u"baz": []}
json_string = json.dumps(data)
# u'{"foo": "bar", "baz": []}'
json.loads(json_string)
# {u"foo": u"bar", u"baz": []}
```

And here we use the file-based functions:

```
import json

from io import StringIO

json_file = StringIO()
data = {u"foo": u"bar", u"baz": []}
json.dump(data, json_file)
json_file.seek(0)  # Seek back to the start of the file before reading
json_file_content = json_file.read()
# u'{"foo": "bar", "baz": []}'
json_file.seek(0)  # Seek back to the start of the file before reading
json.load(json_file)
# {u"foo": u"bar", u"baz": []}
```

As you can see the main difference is that when dumping json data you must pass the file handle to the function, as opposed to capturing the return value. Also worth noting is that you must seek to the start of the file before reading or writing, in order to avoid data corruption. When opening a file the cursor is placed at position `0`, so the below would also work:

```
import json

json_file_path = './data.json'
data = {u"foo": u"bar", u"baz": []}

with open(json_file_path, 'w') as json_file:
    json.dump(data, json_file)

with open(json_file_path) as json_file:
    json_file_content = json_file.read()
    # u'{"foo": "bar", "baz": []}'

with open(json_file_path) as json_file:
    json.load(json_file)
    # {u"foo": u"bar", u"baz": []}
```

Having both ways of dealing with json data allows you to idiomatically and efficiently work with formats which build upon json, such as `pyspark`'s json-per-line:

```
# loading from a file
data = [json.loads(line) for line in open(file_path).splitlines()]

# dumping to a file
with open(file_path, 'w') as json_file:
    for item in data:
        json.dump(item, json_file)
        json_file.write('\n')
```

Creating JSON from Python dict

```
import json
d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}
json.dumps(d)
```

The above snippet will return the following:

```
'{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
```

Creating Python dict from JSON

```
import json
s = '{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
json.loads(s)
```

The above snippet will return the following:

```
{u'alice': 1, u'foo': u'bar', u'wonderland': [1, 2, 3]}
```

Syntax

Parameters

Remarks

For full documentation including version-specific functionality, please check the official documentation .

Types

Defaults

the `json` module will handle encoding and decoding of the below types by default:

De-serialisation types:

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

The `json` module also understands `NaN` , `Infinity` , and `-Infinity` as their corresponding float values, which is outside the JSON spec.

Serialisation types:

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

To disallow encoding of `NaN` , `Infinity` , and `-Infinity` you must encode with `allow_nan=False` . This will then raise a `ValueError` if you attempt to encode these values.

Custom (de-)serialisation

There are various hooks which allow you to handle data that needs to be represented differently. Use of `functools.partial` allows you to partially apply the relevant parameters to these functions for convenience.

Serialisation:

You can provide a function that operates on objects before they are serialised like so:

```
# my_json module

import json
from functools import partial

def serialise_object(obj):
    # Do something to produce json-serialisable data
    return dict_obj

dump = partial(json.dump, default=serialise_object)
dumps = partial(json.dumps, default=serialise_object)
```

De-serialisation:

There are various hooks that are handled by the json functions, such as object_hook and parse_float. For an exhaustive list for your version of python, see here .

```
# my_json module

import json
from functools import partial

def deserialise_object(dict_obj):
    # Do something custom
    return obj

def deserialise_float(str_obj):
    # Do something custom
    return obj

load = partial(json.load, object_hook=deserialise_object, parse_float=deserialise_float)
loads = partial(json.loads, object_hook=deserialise_object, parse_float=deserialise_float)
```

Further custom (de-)serialisation:

The `json` module also allows for extension/substitution of the `json.JSONEncoder` and `json.JSONDecoder` to handle miscellaneous types. The hooks documented above can be added as defaults by creating an equivalently named method. To use these simply pass the class as the `cls` parameter to the relevant function. Use of `functools.partial` allows you to partially apply the cls parameter to these functions for convenience, e.g.

```
# my_json module

import json
from functools import partial

class MyEncoder(json.JSONEncoder):
    # Do something custom

class MyDecoder(json.JSONDecoder):
    # Do something custom

dump = partial(json.dump, cls=MyEncoder)
dumps = partial(json.dumps, cls=MyEncoder)
load = partial(json.load, cls=MyDecoder)
loads = partial(json.loads, cls=MyDecoder)
```