json — JavaScript Object Notation

Purpose: Encode Python objects as JSON strings, and decode JSON strings into Python objects.

The json module provides an API similar to pickle for converting in-memory Python objects to a serialized representation known as JavaScript Object Notation (JSON). Unlike pickle, JSON has the benefit of having implementations in many languages (especially JavaScript). It is most widely used for communicating between the web server and client in a REST API, but is also useful for other inter-application communication needs.

Encoding and Decoding Simple Data Types

The encoder understands Python's native types by default (str, int, float, list, tuple, and dict).

```
# json_simple_types.py

import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
print('DATA:', repr(data))

data_string = json.dumps(data)
print('JSON:', data_string)
```

Values are encoded in a manner superficially similar to Python's repr() output.

```
$ python3 json_simple_types.py

DATA: [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
JSON: [{"a": "A", "b": [2, 4], "c": 3.0}]
```

Encoding, then re-decoding may not give exactly the same type of object.

```
# json_simple_types_decode.py

import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
print('DATA   :', data)

data_string = json.dumps(data)
print('ENCODED:', data_string)

decoded = json.loads(data_string)
print('DECODED:', decoded)

print('ORIGINAL:', type(data[0]['b']))
print('DECODED :', type(decoded[0]['b']))
```

In particular, tuples become lists.

```
$ python3 json_simple_types_decode.py

DATA   : [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
ENCODED: [{"a": "A", "b": [2, 4], "c": 3.0}]
DECODED: [{'a': 'A', 'b': [2, 4], 'c': 3.0}]
ORIGINAL: <class 'tuple'>
DECODED : <class 'list'>
```

Human-consumable vs. Compact Output

Another benefit of JSON over pickle is that the results are human-readable. The dumps() function accepts several arguments to make the output even nicer. For example, the `sort_keys` flag tells the encoder to output the keys of a dictionary in sorted, instead of random, order.

```
# json_sort_keys.py

import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
print('DATA:', repr(data))

unsorted = json.dumps(data)
print('JSON:', json.dumps(data))
print('SORT:', json.dumps(data, sort_keys=True))

first = json.dumps(data, sort_keys=True)
second = json.dumps(data, sort_keys=True)

print('UNSORTED MATCH:', unsorted == first)
print('SORTED MATCH  :', first == second)
```

Sorting makes it easier to scan the results by eye, and also makes it possible to compare JSON output in tests.

```
$ python3 json_sort_keys.py

DATA: [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
JSON: [{"a": "A", "b": [2, 4], "c": 3.0}]
SORT: [{"a": "A", "b": [2, 4], "c": 3.0}]
UNSORTED MATCH: True
SORTED MATCH  : True
```

For highly-nested data structures, specify a value for `indent` so the output is formatted nicely as well.

```
# json_indent.py

import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
print('DATA:', repr(data))

print('NORMAL:', json.dumps(data, sort_keys=True))
print('INDENT:', json.dumps(data, sort_keys=True, indent=2))
```

When indent is a non-negative integer, the output more closely resembles that of `pprint`, with leading spaces for each level of the data structure matching the indent level.

```
$ python3 json_indent.py

DATA: [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
NORMAL: [{"a": "A", "b": [2, 4], "c": 3.0}]
INDENT: [
  {
    "a": "A",
    "b": [
      2,
      4
    ],
    "c": 3.0
  }
]
```

Verbose output like this increases the number of bytes needed to transmit the same amount of data, however, so it is not intended for use in a production environment. In fact, it is possible to adjust the settings for separating data in the encoded output to make it even more compact than the default.

```
# json_compact_encoding.py

import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
print('DATA:', repr(data))

print('repr(data)             :', len(repr(data)))

plain_dump = json.dumps(data)
print('dumps(data)            :', len(plain_dump))

small_indent = json.dumps(data, indent=2)
print('dumps(data, indent=2)  :', len(small_indent))

with_separators = json.dumps(data, separators=(',', ':'))
print('dumps(data, separators):', len(with_separators))
```

The separators argument to dumps() should be a tuple containing the strings to separate items in a list and keys from values in a dictionary. The default is (', ', ': '). By removing the whitespace, a more compact output is produced.

```
$ python3 json_compact_encoding.py

DATA: [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
repr(data)             : 35
dumps(data)            : 35
dumps(data, indent=2)  : 73
dumps(data, separators): 29
```

Encoding Dictionaries

The JSON format expects the keys to a dictionary to be strings. Trying to encode a dictionary with non-string types as keys produces a TypeError. One way to work around that limitation is to tell the encoder to skip over non-string keys using the skipkeys argument:

```
# json_skipkeys.py
```

```
import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0, ('d',): 'D tuple'}]

print('First attempt')
try:
    print(json.dumps(data))
except TypeError as err:
    print('ERROR:', err)

print()
print('Second attempt')
print(json.dumps(data, skipkeys=True))
```

Rather than raising an exception, the non-string key is ignored.

```
$ python3 json_skipkeys.py

First attempt
ERROR: keys must be str, int, float, bool or None, not tuple

Second attempt
[{"a": "A", "b": [2, 4], "c": 3.0}]
```

Working with Custom Types

All of the examples so far have used Pythons built-in types because those are supported by json natively. It is common to need to encode custom classes, as well, and there are two ways to do that.

Given this class to encode:

```
# json_myobj.py
```

```


class MyObj:
    def __init__(self, s):
        self.s = s

    def __repr__(self):
        return '<MyObj({})>'.format(self.s)
```

The simple way of encoding a `MyObj` instance is to define a function to convert an unknown type to a known type. It does not need to do the encoding, so it should just convert one object to another.

```
# json_dump_default.py

import json
import json_myobj

obj = json_myobj.MyObj('instance value goes here')

print('First attempt')
try:
    print(json.dumps(obj))
except TypeError as err:
    print('ERROR:', err)


def convert_to_builtin_type(obj):
    print('default(', repr(obj), ')')
    # Convert objects to a dictionary of their representation
    d = {
        '__class__': obj.__class__.__name__,
        '__module__': obj.__module__,
    }
    d.update(obj.__dict__)
    return d


print()
print('With default')
print(json.dumps(obj, default=convert_to_builtin_type))
```

In `convert_to_builtin_type()`, instances of classes not recognized by `json` are converted to dictionaries with enough information to re-create the object if a program has access to the Python modules necessary.

```
$ python3 json_dump_default.py

First attempt
ERROR: Object of type MyObj is not JSON serializable

With default
default( <MyObj(instance value goes here)> )
{"__class__": "MyObj", "__module__": "json_myobj", "s": "instance
value goes here"}
```

To decode the results and create a `MyObj()` instance, use the `object_hook` argument to `loads()` to tie in to the decoder so the class can be imported from the module and used to create the instance.

The `object_hook` is called for each dictionary decoded from the incoming data stream, providing a chance to convert the dictionary to another type of object. The hook function should return the object the calling application should receive instead of the dictionary.

```
# json_load_object_hook.py

import json


def dict_to_object(d):
    if '__class__' in d:
        class_name = d.pop('__class__')
        module_name = d.pop('__module__')
        module = __import__(module_name)
        print('MODULE:', module.__name__)
        print('MODULE:', module.__name__)
        class_ = getattr(module, class_name)
        print('CLASS:', class_)
        args = {
            key: value
            for key, value in d.items()
        }
        print('INSTANCE ARGS:', args)
        inst = class_(**args)
    else:
        inst = d
    return inst


encoded_object = '''
    [{"s": "instance value goes here",
      "__module__": "json_myobj", "__class__": "MyObj"}]
    '''

myobj_instance = json.loads(
    encoded_object,
    object_hook=dict_to_object,
)
print(myobj_instance)
```

Since json converts string values to unicode objects, they need to be re-encoded as ASCII strings before they can be used as keyword arguments to the class constructor.

```
$ python3 json_load_object_hook.py

MODULE: json_myobj
CLASS: <class 'json_myobj.MyObj'>
INSTANCE ARGS: {'s': 'instance value goes here'}
[<MyObj(instance value goes here)>]
```

Similar hooks are available for the built-in types integers (parse_int), floating point numbers (parse_float), and constants (parse_constant).

Encoder and Decoder Classes

Besides the convenience functions already covered, the json module provides classes for encoding and decoding. Using the classes directly gives access to extra APIs for customizing their behavior.

The JSONEncoder uses an iterable interface for producing “chunks” of encoded data, making it easier to write to files or network sockets without having to represent an entire data structure in memory.

```
# json_encoder_iterable.py

import json

encoder = json.JSONEncoder()
data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]

for part in encoder.iterencode(data):
    print('PART:', part)
```

The output is generated in logical units, rather than being based on any size value.

```
$ python3 json_encoder_iterable.py

PART: [
PART: {
PART: "a"
PART: :
PART: "A"
PART: ,
PART: "b"
PART: :
PART: [2
PART: , 4
PART: ]
PART: ,
PART: "c"
PART: :
PART: 3.0
PART: }
PART: ]
```

The `encode()` method is basically equivalent to `''.join(encoder.iterencode())`, with some extra error checking up front.

To encode arbitrary objects, override the `default()` method with an implementation similar to the one used in `convert_to_builtin_type()`.

```python
# json_encoder_default.py

import json
import json_myobj


class MyEncoder(json.JSONEncoder):

    def default(self, obj):
        print('default(', repr(obj), ')')
        # Convert objects to a dictionary of their representation
        d = {
            '__class__': obj.__class__.__name__,
            '__module__': obj.__module__,
        }
        d.update(obj.__dict__)
        return d


obj = json_myobj.MyObj('internal data')
print(obj)
print(MyEncoder().encode(obj))
```

The output is the same as the previous implementation.

```
$ python3 json_encoder_default.py

<MyObj(internal data)>
default( <MyObj(internal data)> )
{"__class__": "MyObj", "__module__": "json_myobj", "s": "internal
data"}
```

Decoding text, then converting the dictionary into an object takes a little more work to set up than the previous implementation, but not much.

```python
# json_decoder_object_hook.py

import json


class MyDecoder(json.JSONDecoder):

    def __init__(self):
        json.JSONDecoder.__init__(
            self,
            object_hook=self.dict_to_object,
        )

    def dict_to_object(self, d):
        if '__class__' in d:
            class_name = d.pop('__class__')
            module_name = d.pop('__module__')
            module = __import__(module_name)
            print('MODULE:', module.__name__)
            class_ = getattr(module, class_name)
            print('CLASS:', class_)
            args = {
                key: value
                for key, value in d.items()
            }
            print('INSTANCE ARGS:', args)
            inst = class_(**args)
        else:
            inst = d
        return inst


encoded_object = '''
[{"s": "instance value goes here",
  "__module__": "json_myobj", "__class__": "MyObj"}]
'''

myobj_instance = MyDecoder().decode(encoded_object)
print(myobj_instance)
```

And the output is the same as the earlier example.

```
$ python3 json_decoder_object_hook.py

MODULE: json_myobj
CLASS: <class 'json_myobj.MyObj'>
INSTANCE ARGS: {'s': 'instance value goes here'}
[<MyObj(instance value goes here)>]
```

Working with Streams and Files

All of the examples so far have assumed that the encoded version of the entire data structure could be held in memory at one time. With large data structures, it may be preferable to write the encoding directly to a file-like object. The convenience functions `load()` and `dump()` accept references to a file-like object to use for reading or writing.

```
# json_dump_file.py

import io
import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]

f = io.StringIO()
json.dump(data, f)

print(f.getvalue())
```

A socket or normal file handle would work the same way as the `StringIO` buffer used in this example.

```
$ python3 json_dump_file.py

[{"a": "A", "b": [2, 4], "c": 3.0}]
```

Although it is not optimized to read only part of the data at a time, the `load()` function still offers the benefit of encapsulating the logic of generating objects from stream input.

```
# json_load_file.py

import io
import json

f = io.StringIO('[{"a": "A", "c": 3.0, "b": [2, 4]}]')
print(json.load(f))
```

Just as for `dump()`, any file-like object can be passed to `load()`.

```
$ python3 json_load_file.py

[{'a': 'A', 'c': 3.0, 'b': [2, 4]}]
```

Mixed Data Streams

JSONDecoder includes raw_decode(), a method for decoding a data structure followed by more data, such as JSON data with trailing text. The return value is the object created by decoding the input data, and an index into that data indicating where decoding left off.

```
# json_mixed_data.py

import json

decoder = json.JSONDecoder()


def get_decoded_and_remainder(input_data):
    obj, end = decoder.raw_decode(input_data)
    remaining = input_data[end:]
    return (obj, end, remaining)


encoded_object = '[{"a": "A", "c": 3.0, "b": [2, 4]}]'
extra_text = 'This text is not JSON.'

print('JSON first:')
data = ' '.join([encoded_object, extra_text])
obj, end, remaining = get_decoded_and_remainder(data)

print('Object              :', obj)
print('End of parsed input :', end)
print('Remaining text      :', repr(remaining))

print()
print('JSON embedded:')
try:
    data = ' '.join([extra_text, encoded_object, extra_text])
    obj, end, remaining = get_decoded_and_remainder(data)
except ValueError as err:
    print('ERROR:', err)
```

Unfortunately, this only works if the object appears at the beginning of the input.

```
$ python3 json_mixed_data.py

JSON first:
Object              : [{'a': 'A', 'c': 3.0, 'b': [2, 4]}]
End of parsed input : 35
Remaining text      : ' This text is not JSON.'

JSON embedded:
ERROR: Expecting value: line 1 column 1 (char 0)
```

JSON at the Command Line

The json.tool module implements a command line program for reformatting JSON data to be easier to read.

```
[{"a": "A", "c": 3.0, "b": [2, 4]}]
```

The input file example.json contains a mapping with the keys out of alphabetical order. The first example below shows the data reformatted in order, and the second example uses --sort-keys to sort the mapping keys before printing the output.

```
$ python3 -m json.tool example.json

[
    {
        "a": "A",
        "c": 3.0,
        "b": [
            2,
            4
        ]
    }
]

$ python3 -m json.tool --sort-keys example.json

[
    {
        "a": "A",
        "b": [
            2,
            4
        ],
        "c": 3.0
    }
]
```

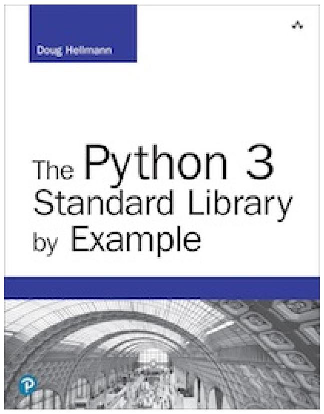
See also

- Standard library documentation for json
- Python 2 to 3 porting notes for json
- JavaScript Object Notation – JSON home, with documentation and implementations in other languages.
- jsonpickle – `jsonpickle` allows for any Python object to be serialized into JSON.

uuid — Universally Unique Identifiers xmlrpc.client — Client Library for XML-RPC

This page was last updated 2018-12-09.

Navigation
uuid — Universally Unique Identifiers
xmlrpc.client — Client Library for XML-RPC



Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

© Copyright 2019, Doug Hellmann

BY NC SA