python-rapidjson Documentation

Release 0.9.4

Ken Robbins, Lele Gaifax

Jul 06, 2022

CONTENTS

1 Python 3 wrapper around RapidJSON **1**
1.1 Quick start 1
1.2 Performance 5
1.3 Exposed functions and symbols 8
1.4 Indices and tables 37

Index **39**

CHAPTER

ONE

PYTHON 3 WRAPPER AROUND RAPIDJSON

This module implements a Python 3 wrapper around RapidJSON, which is an extremely fast C++ JSON serialization library.

Contents:

1.1 Quick start

This a quick overview of the module.

1.1.1 Installation

First install `python-rapidjson`:

```
$ pip install python-rapidjson
```

If possible this installs a *binary wheel*, containing the latest version of the package already compiled for your system. Otherwise it will download a *source distribution* and will try to compile it: as the module is written in C++, in this case you most probably will need to install a minimal C++ compiler toolchain on your system.

Alternatively it is also possible to install it using Conda.

1.1.2 Basic examples

`python-rapidjson` tries to be compatible with the standard library `json.dumps()` and `json.loads()` functions (but see the *incompatibilities*).

Basic usage looks like this:

```
>>> from pprint import pprint
>>> from rapidjson import dumps, loads
>>> data = {'foo': 100, 'bar': 'baz'}
>>> dumps(data, sort_keys=True) # for doctest
'{"bar":"baz","foo":100}'
>>> pprint(loads('{"bar":"baz","foo":100}'))
{'bar': 'baz', 'foo': 100}
```

All JSON data types are supported using their native Python counterparts:

```
>>> int_number = 42
>>> float_number = 1.4142
>>> string = "2  1.4142"
>>> false = False
>>> true = True
>>> null = None
>>> array = [int_number, float_number, string, false, true, null]
>>> an_object = {'int': int_number, 'float': float_number,
...              'string': string,
...              'true': true, 'false': false,
...              'array': array }
>>> pprint(loads(dumps({'object': an_object})))
{'object': {'array': [42, 1.4142, '2  1.4142', False, True, None],
            'false': False,
            'float': 1.4142,
            'int': 42,
            'string': '2  1.4142',
            'true': True}}
```

Python's lists, tuples and iterators get serialized as JSON arrays:

```
>>> names_t = ('Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat')
>>> names_l = list(names_t)
>>> names_i = iter(names_l)
>>> def names_g():
...     for name in names_t:
...         yield name
>>> dumps(names_t) == dumps(names_l) == dumps(names_i) == dumps(names_g())
True
```

Values can also be `bytes` or `bytearray` instances, which are assumed to contain proper `UTF-8`-encoded strings:

```
>>> clef = "\N{MUSICAL SYMBOL G CLEF}"
>>> bytes_utf8 = clef.encode('utf-8')
>>> bytearray = bytearray(bytes_utf8)
>>> dumps(clef) == dumps(bytes_utf8) == dumps(bytearray) == '"\\uD834\\uDD1E"'
True
```

`python-rapidjson` can optionally handle also a few other commonly used data types:

```
>>> import datetime, decimal, uuid
>>> from rapidjson import DM_ISO8601, UM_CANONICAL, NM_DECIMAL
>>> some_day = datetime.date(2016, 8, 28)
>>> some_timestamp = datetime.datetime(2016, 8, 28, 13, 14, 15)
>>> dumps({'a date': some_day, 'a timestamp': some_timestamp})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: datetime.datetime(...) is not JSON serializable
>>> dumps({'a date': some_day, 'a timestamp': some_timestamp},
...       datetime_mode=DM_ISO8601,
...       sort_keys=True) # for doctests
'{"a date":"2016-08-28","a timestamp":"2016-08-28T13:14:15"}'
>>> as_json = _
```

(continues on next page)

(continued from previous page)

```
>>> pprint(loads(as_json))
{'a date': '2016-08-28', 'a timestamp': '2016-08-28T13:14:15'}
>>> pprint(loads(as_json, datetime_mode=DM_ISO8601))
{'a date': datetime.date(2016, 8, 28),
 'a timestamp': datetime.datetime(2016, 8, 28, 13, 14, 15)}
>>> some_uuid = uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
>>> dumps(some_uuid)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: UUID(...) is not JSON serializable
>>> dumps(some_uuid, uuid_mode=UM_CANONICAL)
'"886313e1-3b8a-5372-9b90-0c9aee199e5d"'
>>> as_json = _
>>> loads(as_json)
'886313e1-3b8a-5372-9b90-0c9aee199e5d'
>>> loads(as_json, uuid_mode=UM_CANONICAL)
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')
>>> pi = decimal.Decimal('3.14159265358979323846264338327950288419 71')
>>> dumps(pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Decimal(...) is not JSON serializable
>>> dumps(pi, number_mode=NM_DECIMAL)
'3.14159265358979323846264338327950288419 71'
>>> as_json = _
>>> loads(as_json)
3.141592653589793
>>> type(loads(as_json))
<class 'float'>
>>> loads(as_json, number_mode=NM_DECIMAL)
Decimal('3.14159265358979323846264338327950288419 71')
```

The module exposes also a *stream* interface:

```
>>> from io import StringIO
>>> from rapidjson import dump, load
>>> stream = StringIO()
>>> dump(data, stream)
>>> stream.seek(0)
0
>>> load(stream) == data
True
```

1.1.3 Incompatibilities

Here are things in the standard `json` library that we have decided not to support:

`separators` argument

This is mostly used for pretty printing and not supported by RapidJSON so it isn't a high priority. We do support `indent` kwarg that would get you nice looking JSON anyways.

Coercing keys when dumping

`json` will stringify a `True` dictionary key as `"true"` if you dump it out but when you load it back in it'll still be a string. We want the dump and load to return the exact same objects so we have decided not to do this coercion by default; you can however use `MM_COERCE_KEYS_TO_STRINGS` or a `default` function to mimic that.

Arbitrary encodings

`json.loads()` accepts an `encoding` kwarg determining the encoding of its input, when that is a `bytes` or `bytearray` instance. Although `RapidJSON` is able to cope with several different encodings, we currently supports only the recommended one, `UTF-8`.

`cls` argument to `loads()` and `dumps()`

The `json` top level functions accept a `cls` parameter that allows to specify custom encoder/decoder class. If you must use that approach, that is you have to use the standard `json` top level functions but want to use `RapidJSON` functionalities, the following snippet shows a reasonably simple way to do that:

```
>>> import datetime
>>> import json
>>> import rapidjson
>>>
>>> class Encoder:
...     def __init__(self, *args, **kwargs):
...         # Filter/adapt JSON arguments to RapidJSON ones
...         rjkwargs = {'datetime_mode': rapidjson.DM_ISO8601}
...         encoder = rapidjson.Encoder(**rjkwargs)
...         self.encode = encoder.__call__
>>>
>>> json.dumps([1,2,datetime.date(2020, 12, 8)], cls=Encoder)
'[1,2,"2020-12-08"]'
>>>
>>> class Decoder:
...     def __init__(self, *args, **kwargs):
...         # Filter/adapt JSON arguments to RapidJSON ones
...         rjkwargs = {'datetime_mode': rapidjson.DM_ISO8601}
...         encoder = rapidjson.Decoder(**rjkwargs)
...         self.decode = encoder.__call__
>>>
>>> json.loads('[1,2,"2020-12-08"]', cls=Decoder)
[1, 2, datetime.date(2020, 12, 8)]
```

`object_pairs_hook` argument

`json` decoding functions accept an `object_pairs_hook` kwarg, a variant of `object_hook` that selects a different way to translate JSON objects into Python dictionaries by first collecting their content into a sequence of *key-value pairs* and eventually passing that sequence to the hook function. That behaviour may be easily simulated:

```
>>> def loads(s, object_pairs_hook=None):
...     if object_pairs_hook is None:
```

(continues on next page)

(continued from previous page)

```
...         d = rapidjson.Decoder()
...     else:
...         class KWPairsDecoder(rapidjson.Decoder):
...             def start_object(self):
...                 return []
...             def end_object(self, pairs):
...                 return object_pairs_hook(pairs)
...         d = KWPairsDecoder()
...     return d(s)
>>>
>>> loads('{"foo": "bar"}', lambda pairs: ','.join(f'{k}={v}' for k, v in pairs))
'foo=bar'
```

1.2 Performance

python-rapidjson tries to be as performant as possible while staying compatible with the json module.

1.2.1 Tables

The following tables show a comparison between this module and other libraries with different data sets. Last row ("overall") is the total time taken by all the benchmarks.

Each number shows the factor between the time taken by each contender and python-rapidjson (in other words, they are *normalized* against a value of 1.0 for python-rapidjson): the lower the number, the speedier the contender.

In **bold** the winner.

Serialization

serialize	dumps()[1]	Encoder()[2]	dumps(n)[3]	Encoder(n)[4]	simdjson[5]	orjson[6]	ujson[7]	simplejson[8]	stdlib[9]
100 arrays dict	1.00	0.99	0.77	0.76	1.97	**0.26**	1.09	4.00	1.95
100 dicts array	1.00	0.99	0.81	0.76	2.06	**0.34**	1.06	4.96	2.04
256 Trues array	1.00	1.00	1.17	1.04	2.38	**0.38**	1.52	2.70	2.35
256 ascii array	1.00	1.00	1.02	1.00	0.90	**0.26**	0.50	1.06	0.89
256 doubles array	1.00	0.99	0.99	0.99	0.84	**0.06**	0.21	0.88	0.83
256 unicode array	1.00	0.81	0.81	0.82	0.70	**0.09**	0.51	0.65	0.69
apache.json	1.00	1.00	1.01	1.01	1.66	**0.33**	1.18	2.51	1.65
canada.json	1.00	1.00	0.99	1.00	1.06	**0.09**	0.34	1.72	1.06
complex object	1.00	1.00	0.93	0.92	1.58	**0.26**	0.83	2.41	1.57
composite object	1.00	0.99	0.75	0.72	1.91	**0.36**	1.12	2.44	1.87
ctm.json	1.00	1.00	0.76	0.76	2.11	**0.33**	1.43	4.81	2.13
github.json	1.00	1.00	0.98	0.97	1.34	**0.30**	0.98	1.72	1.32
instruments.json	1.00	1.01	0.80	0.80	1.75	**0.34**	1.15	2.28	1.75
mesh.json	1.00	0.99	0.90	0.90	0.94	**0.14**	0.35	1.09	0.93
truenull.json	1.00	1.02	1.03	1.01	1.87	**0.41**	1.69	1.89	1.86
tweet.json	1.00	0.99	0.99	0.94	1.66	**0.33**	1.02	2.33	1.65
twitter.json	1.00	1.00	0.96	0.95	1.39	**0.32**	1.05	1.65	1.36
overall	1.00	0.99	0.83	0.82	1.71	**0.22**	0.90	3.33	1.70

[1] `rapidjson.dumps()`
[2] `rapidjson.Encoder()`
[3] `rapidjson.dumps(number_mode=NM_NATIVE)`
[4] `rapidjson.Encoder(number_mode=NM_NATIVE)`
[5] simdjson 5.0.1
[6] orjson 3.7.6
[7] ujson 5.4.0
[8] simplejson 3.17.6
[9] Python 3.10.5 standard library `json`

Deserialization

deserialize	`loads()`[10]	`Decoder()`[11]	`loads(n)`[12]	`Decoder(n)`[13]	simdjson	orjson	ujson	simplejson	stdlib
100 arrays dict	1.00	1.05	0.90	0.95	0.86	**0.56**	0.86	1.07	0.97
100 dicts array	1.00	0.99	0.87	0.85	0.63	**0.40**	0.68	1.27	1.02
256 Trues array	1.00	1.02	1.11	0.99	0.92	**0.48**	0.83	1.51	1.36
256 ascii array	1.00	0.99	1.02	1.01	0.46	**0.45**	0.82	0.84	0.81
256 doubles array	1.00	1.00	0.27	0.26	0.21	**0.16**	0.54	1.09	1.07
256 unicode array	1.00	1.00	1.00	1.00	1.10	**0.39**	0.80	4.49	1.29
apache.json	1.00	1.00	1.01	1.01	0.65	**0.48**	0.85	0.81	0.83
canada.json	1.00	1.00	0.34	0.34	0.29	**0.25**	0.49	1.04	0.96
complex object	1.00	1.00	0.84	0.84	0.65	**0.37**	0.80	1.15	1.02
composite object	1.00	0.99	0.83	0.81	0.61	**0.45**	0.59	1.37	1.07
ctm.json	1.00	1.01	0.92	0.92	0.74	**0.55**	1.08	1.20	1.14
github.json	1.00	1.00	0.98	0.98	0.62	**0.41**	0.76	0.86	0.79
instruments.json	1.00	1.00	0.91	0.90	0.63	**0.37**	0.72	1.12	0.92
mesh.json	1.00	1.01	0.51	0.51	0.44	**0.36**	0.63	1.44	1.03
truenull.json	1.00	0.99	1.00	1.00	0.58	**0.39**	0.84	0.98	0.87
tweet.json	1.00	1.02	0.99	0.98	0.66	**0.44**	0.84	1.14	1.03
twitter.json	1.00	1.00	0.99	0.99	0.65	**0.42**	0.86	1.01	1.00
overall	1.00	1.05	0.85	0.90	0.80	**0.53**	0.83	1.08	0.97

ASCII vs UTF-8 Serialization

serialize	`rj ascii`Page 8, 15	`rj utf8`Page 8, 16	`uj ascii`Page 8, 17	`uj utf8`Page 8, 18	`sj ascii`Page 8, 19	`sj utf8`Page 8, 20	`json ascii`Page 8, 21	`json utf8`Page 8, 22
Long ASCII string	1.00	0.46	**0.25**	0.41	0.63	0.93	0.57	1.49
Long Unicode string	1.00	0.62	0.63	**0.58**	0.71	0.70	0.80	0.66
overall	1.00	0.57	**0.51**	0.53	0.69	0.78	0.73	0.92

[10] `rapidjson.loads()`
[11] `rapidjson.Decoder()`
[12] `rapidjson.loads(number_mode=NM_NATIVE)`
[13] `rapidjson.Decoder(number_mode=NM_NATIVE)`

1.2.2 DIY

To run these tests yourself, clone the repo and run:

```
$ make benchmarks
```

to focus only on `RapidJSON` or:

```
$ make benchmarks-others
```

to get full comparison against other engines.

To reproduce the tables above, run:

```
$ make benchmarks-tables
```

Compare different versions

pytest-benchmark implements an handy feature that allows you to weight the impact of a particular change. For example, you may start from a released version and execute:

```
$ make benchmarks PYTEST_OPTIONS=--benchmark-autosave
```

After applying whatever change to the code base, you can get a differential view by executing:

```
$ make benchmarks PYTEST_OPTIONS=--benchmark-compare=0001
```

1.3 Exposed functions and symbols

1.3.1 dumps() function

`rapidjson.`**`dumps`**(*obj*, *, *skipkeys=False*, *ensure_ascii=True*, *write_mode=WM_COMPACT*, *indent=4*, *default=None*, *sort_keys=False*, *number_mode=None*, *datetime_mode=None*, *uuid_mode=None*, *bytes_mode=BM_UTF8*, *iterable_mode=IM_ANY_ITERABLE*, *mapping_mode=MM_ANY_MAPPING*, *allow_nan=True*)

Encode given Python *obj* instance into a `JSON` string.

Parameters

- **`obj`** – the value to be serialized
- **`skipkeys`** (*`bool`*) – whether invalid `dict` keys will be skipped
- **`ensure_ascii`** (*`bool`*) – whether the output should contain only ASCII characters
- **`write_mode`** (*`int`*) – enable particular pretty print behaviors

[15] `rapidjson.dumps(ensure_ascii=True)`
[16] `rapidjson.dumps(ensure_ascii=False)`
[17] `ujson.dumps(ensure_ascii=True)`
[18] `ujson.dumps(ensure_ascii=False)`
[19] `simplejson.dumps(ensure_ascii=True)`
[20] `simplejson.dumps(ensure_ascii=False)`
[21] `stdlib json.dumps(ensure_ascii=True)`
[22] `stdlib json.dumps(ensure_ascii=False)`

- **indent** – indentation width or string to produce pretty printed JSON
- **default** (*callable*) – a function that gets called for objects that can't otherwise be serialized
- **sort_keys** (*bool*) – whether dictionary keys should be sorted alphabetically
- **number_mode** (*int*) – enable particular behaviors in handling numbers
- **datetime_mode** (*int*) – how should `datetime`, `time` and `date` instances be handled
- **uuid_mode** (*int*) – how should `UUID` instances be handled
- **bytes_mode** (*int*) – how should `bytes` instances be handled
- **iterable_mode** (*int*) – how should *iterable* values be handled
- **mapping_mode** (*int*) – how should *mapping* values be handled
- **allow_nan** (*bool*) – *compatibility* flag equivalent to `number_mode=NM_NAN`

Returns
A Python `str` instance.

skipkeys

If *skipkeys* is true (default: `False`), then dict keys that are not of a basic type (`str`, `int`, `float`, `bool`, `None`) will be skipped instead of raising a `TypeError`:

```
>>> dumps({(0,): 'empty tuple', True: 'a true value'})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: keys must be strings
>>> dumps({(0,): 'empty tuple', True: 'a true value'}, skipkeys=True)
'{}'
```

Note: *skipkeys* is a backward compatible alias of new `MM_SKIP_NON_STRING_KEYS` *mapping mode*.

ensure_ascii

If *ensure_ascii* is true (the default), the output is guaranteed to have all incoming non-ASCII characters escaped. If *ensure_ascii* is false, these characters will be output as-is:

```
>>> dumps('The symbol for the Euro currency is €')
'"The symbol for the Euro currency is \\u20AC"'
>>> dumps('The symbol for the Euro currency is €',
...       ensure_ascii=False)
'"The symbol for the Euro currency is €"'
```

write_mode

The *write_mode* controls how `python-rapidjson` emits JSON: by default it is *WM_COMPACT*, that produces the most compact JSON representation:

```
>>> dumps([1, 2, {'three': 3, 'four': 4}])
'[1,2,{"four":4,"three":3}]'
```

With *WM_PRETTY* it will use `RapidJSON`'s `PrettyWriter`, with a default *indent* (see below) of four spaces:

```
>>> print(dumps([1, 2, {'three': 3, 'four': 4}], write_mode=WM_PRETTY))
[
    1,
    2,
    {
        "four": 4,
        "three": 3
    }
]
```

With *WM_SINGLE_LINE_ARRAY* arrays will be kept on a single line:

```
>>> print(dumps([1, 2, 'three', [4, 5]], write_mode=WM_SINGLE_LINE_ARRAY))
[1, 2, 'three', [4, 5]]
>>> print(dumps([1, 2, {'three': 3, 'four': 4}], write_mode=WM_SINGLE_LINE_ARRAY))
[1, 2, {
        "three": 3,
        "four": 4
    }]
```

indent

The *indent* parameter may be either a positive integer number or a string: in the former case it specifies a number of spaces, while in the latter the string may contain zero or more ASCII *whitespace* characters (space, tab `\t`, newline `\n` and carriage-return `\r`), all equals (that is, `"\n\t"` is not accepted).

The integer number or the length of the string determine how many spaces (or the characters composing the string) will be used to indent nested structures, when the *write_mode* above is not *WM_COMPACT*, and it defaults to 4. Specifying a value different from `None` automatically sets *write_mode* to *WM_PRETTY*, if not explicited.

By setting *indent* to 0 each array item (when *write_mode* is not `WM_SINGLE_LINE_MODE`) and each dictionary value will be followed by a newline. A positive integer means that each *level* will be indented by that many spaces:

```
>>> print(dumps([1, 2, {'three': 3, 'four': 4}], indent=0))
[
1,
2,
{
"four": 4,
"three": 3
}
]
>>> print(dumps([1, 2, {'three': 3, 'four': 4}], indent=2))
```

(continues on next page)

(continued from previous page)

```
[
  1,
  2,
  {
    "four": 4,
    "three": 3
  }
]
>>> print(dumps([1, 2, {'three': 3, 'four': 4}], indent=""))
[
1,
2,
{
"four": 4,
"three": 3
}
]
>>> print(dumps([1, 2, {'three': 3, 'four': 4}], indent="  "))
[
  1,
  2,
  {
    "four": 4,
    "three": 3
  }
]
>>> print(dumps([1, 2, {'three': 3, 'four': 4}], indent="\t"))
[
        1,
        2,
        {
                "three": 3,
                "four": 4
        }
]
```

default

The *default* argument may be used to specify a custom serializer for otherwise not handled objects. If specified, it should be a function that gets called for such objects and returns a JSON encodable version of the object itself or raise a `TypeError`:

```
>>> class Point(object):
...   def __init__(self, x, y):
...     self.x = x
...     self.y = y
...
>>> point = Point(1,2)
>>> dumps(point)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(continues on next page)

(continued from previous page)

```
TypeError: <__main__.Point object at ...> is not JSON serializable
>>> def point_jsonifier(obj):
...   if isinstance(obj, Point):
...     return {'x': obj.x, 'y': obj.y}
...   else:
...     raise ValueError('%r is not JSON serializable' % obj)
...
>>> dumps(point, default=point_jsonifier)
'{"x":1,"y":2}'
```

sort_keys

When *sort_keys* is true (default: `False`), the JSON representation of Python dictionaries is sorted by key:

```
>>> data = {'a': 'A', 'c': 'C', 'i': 'I', 'd': 'D'}
>>> dumps(data, sort_keys=True)
'{"a":"A","c":"C","d":"D","i":"I"}'
```

Note: *sort_keys* is a backward compatible alias of new `MM_SORT_KEYS` *mapping mode*.

```
>>> dumps(data, mapping_mode=MM_SORT_KEYS)
'{"a":"A","c":"C","d":"D","i":"I"}'
```

The default setting, on modern snakes (that is, on Python >= 3.7), preserves original dictionary insertion order:

```
>>> dumps(data)
'{"a":"A","c":"C","i":"I","d":"D"}'
```

number_mode

The *number_mode* argument selects different behaviors in handling numeric values.

By default *non-numbers* (`nan`, `inf`, `-inf`) will be serialized as their JavaScript equivalents (`NaN`, `Infinity`, `-Infinity`), because `NM_NAN` is *on* by default (**NB**: this is *not* compliant with the `JSON` standard):

```
>>> nan = float('nan')
>>> inf = float('inf')
>>> dumps([nan, inf])
'[NaN,Infinity]'
>>> dumps([nan, inf], number_mode=NM_NAN)
'[NaN,Infinity]'
```

Explicitly setting *number_mode* or using the compatibility option *allow_nan* you can avoid that and obtain a `ValueError` exception instead:

```
>>> dumps([nan, inf], number_mode=NM_NATIVE)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Out of range float values are not JSON compliant
```

(continues on next page)

(continued from previous page)

```
>>> dumps([nan, inf], allow_nan=False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Out of range float values are not JSON compliant
```

Likewise `Decimal` instances cause a `TypeError` exception:

```
>>> from decimal import Decimal
>>> pi = Decimal('3.14159265358979323846264338327950288419 71')
>>> dumps(pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Decimal(...) is not JSON serializable
```

while using *NM_DECIMAL* they will be serialized as their textual representation like any other float value:

```
>>> dumps(pi, number_mode=NM_DECIMAL)
'3.14159265358979323846264338327950288419 71'
```

Yet another possible flag affects how numeric values are passed to the underlying RapidJSON library: by default they are serialized to their string representation by the module itself, so they are virtually of unlimited precision:

```
>>> dumps(1234567890123456789012345678 90)
'1234567890123456789012345678 90'
```

With *NM_NATIVE* their binary values will be passed directly instead: this is somewhat faster, it is subject to the underlying C library `long long` and `double` limits:

```
>>> dumps(1234567890123456789012345678 90, number_mode=NM_NATIVE)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too big to convert
```

These flags can be combined together:

```
>>> fast_and_precise = NM_NATIVE | NM_DECIMAL | NM_NAN
>>> dumps([-1, nan, pi], number_mode=fast_and_precise)
'[-1,NaN,3.14159265358979323846264338327950288419 71]'
```

datetime_mode

By default `date`, `datetime` and `time` instances are not serializable:

```
>>> from datetime import datetime
>>> right_now = datetime(2016, 8, 28, 13, 14, 52, 277256)
>>> date = right_now.date()
>>> time = right_now.time()
>>> dumps({'date': date, 'time': time, 'timestamp': right_now})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: datetime(...) is not JSON serializable
```

When *datetime_mode* is set to *DM_ISO8601* those values are serialized using the common ISO 8601 format:

```
>>> dumps(['date', date, 'time', time, 'timestamp', right_now],
...        datetime_mode=DM_ISO8601)
'["date","2016-08-28","time","13:14:52.277256","timestamp","2016-08-28T13:14:52.
↪277256"]'
```

The *right_now* value is a naïve datetime (because it does not carry the timezone information) and is normally assumed to be in the local timezone, whatever your system thinks it is. When you instead *know* that your value, even being naïve are actually in the UTC timezone, you can use the *DM_NAIVE_IS_UTC* flag to inform RapidJSON about that:

```
>>> mode = DM_ISO8601 | DM_NAIVE_IS_UTC
>>> dumps(['time', time, 'timestamp', right_now], datetime_mode=mode)
'["time","13:14:52.277256+00:00","timestamp","2016-08-28T13:14:52.277256+00:00"]'
```

A variant is *DM_SHIFT_TO_UTC*, that *shifts* all datetime values to the UTC timezone before serializing them:

```
>>> from datetime import timedelta, timezone
>>> here = timezone(timedelta(hours=2))
>>> now = datetime(2016, 8, 28, 20, 31, 11, 84418, here)
>>> dumps(now, datetime_mode=DM_ISO8601)
'"2016-08-28T20:31:11.084418+02:00"'
>>> mode = DM_ISO8601 | DM_SHIFT_TO_UTC
>>> dumps(now, datetime_mode=mode)
'"2016-08-28T18:31:11.084418+00:00"'
```

With *DM_IGNORE_TZ* the timezone, if present, is simply omitted:

```
>>> mode = DM_ISO8601 | DM_IGNORE_TZ
>>> dumps(now, datetime_mode=mode)
'"2016-08-28T20:31:11.084418"'
```

Another *one-way only* alternative format is Unix time: with *DM_UNIX_TIME* `date`, `datetime` and `time` instances are serialized as a number of seconds, respectively since the `EPOCH` for the first two kinds and since midnight for the latter:

```
>>> mode = DM_UNIX_TIME | DM_NAIVE_IS_UTC
>>> dumps([now, now.date(), now.time()], datetime_mode=mode)
'[1472409071.084418,1472342400.0,73871.084418]'
>>> unixtime = float(dumps(now, datetime_mode=mode))
>>> datetime.fromtimestamp(unixtime, here) == now
True
```

Combining it with the *DM_ONLY_SECONDS* will produce integer values instead, dropping *microseconds*:

```
>>> mode = DM_UNIX_TIME | DM_NAIVE_IS_UTC | DM_ONLY_SECONDS
>>> dumps([now, now.date(), now.time()], datetime_mode=mode)
'[1472409071,1472342400,73871]'
```

It can be used combined with *DM_SHIFT_TO_UTC* to obtain the timestamp of the corresponding UTC time:

```
>>> mode = DM_UNIX_TIME | DM_SHIFT_TO_UTC
>>> dumps(now, datetime_mode=mode)
'1472409071.084418'
```

As above, when you know that your values are in the UTC timezone, you can use the *DM_NAIVE_IS_UTC* flag to get the right result:

```
>>> a_long_time_ago = datetime(1968, 3, 18, 9, 10, 0, 0)
>>> mode = DM_UNIX_TIME | DM_NAIVE_IS_UTC
>>> dumps([a_long_time_ago, a_long_time_ago.date(), a_long_time_ago.time()],
...       datetime_mode=mode)
'[-56472600.0,-56505600.0,33000.0]'
```

uuid_mode

Likewise, to handle `UUID` instances there are two modes that can be specified with the *uuid_mode* argument, that will use the string representation of their values:

```
>>> from uuid import uuid4
>>> random_uuid = uuid4()
>>> dumps(random_uuid)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: UUID(...) is not JSON serializable
>>> dumps(random_uuid, uuid_mode=UM_CANONICAL)
'"be576345-65b5-4fc2-92c5-94e2f82e38fd"'
>>> dumps(random_uuid, uuid_mode=UM_HEX)
'"be57634565b54fc292c594e2f82e38fd"'
```

bytes_mode

By default all `bytes` instances are assumed to be `UTF-8` encoded strings, and acted on accordingly:

```
>>> ascii_string = 'ciao'
>>> bytes_string = b'cio\xc3\xa8'
>>> unicode_string = 'cioè'
>>> dumps([ascii_string, bytes_string, unicode_string])
'["ciao","cio\\u00E8","cio\\u00E8"]'
```

Sometime you may prefer a different approach, explicitly disabling that behavior using the *BM_NONE* mode:

```
>>> dumps([ascii_string, bytes_string, unicode_string],
...       bytes_mode=BM_NONE)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: b'cio\xc3\xa8' is not JSON serializable
>>> my_bytes_handler = lambda b: b.decode('UTF-8').upper()
>>> dumps([ascii_string, bytes_string, unicode_string],
...       bytes_mode=BM_NONE, default=my_bytes_handler)
'["ciao","CIO\\u00C8","cio\\u00E8"]'
```

iterable_mode

By default a value that implements the *iterable* protocol gets encoded as a `JSON` array:

```
>>> from time import localtime, struct_time
>>> lt = localtime()
>>> dumps(lt)
'[2020,11,28,19,55,40,5,333,0]'
>>> class MyList(list):
...   pass
>>> ml = MyList((1,2,3))
>>> dumps(ml)
'[1,2,3]'
```

When that's not appropriate, for example because you want to use a different way to encode them, you may specify *iterable_mode* to `IM_ONLY_LISTS`:

```
>>> dumps(lt, iterable_mode=IM_ONLY_LISTS)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <time.struct_time ...> is not JSON serializable
>>> dumps(ml, iterable_mode=IM_ONLY_LISTS)
Traceback (most recent call last):
  ...
TypeError: [1, 2, 3] is not JSON serializable
```

and thus you can use the *default* argument:

```
>>> def ts_or_ml(obj):
...   if isinstance(obj, struct_time):
...     return {'__class__': 'time.struct_time', '__init__': list(obj)}
...   elif isinstance(obj, MyList):
...     return [i*2 for i in obj]
...   else:
...     raise ValueError('%r is not JSON serializable' % obj)
>>> dumps(lt, iterable_mode=IM_ONLY_LISTS, default=ts_or_ml)
'{"__class__":"time.struct_time","__init__":[2020,11,28,19,55,40,5,333,0]}'
>>> dumps(ml, iterable_mode=IM_ONLY_LISTS, default=ts_or_ml)
'[2,4,6]'
```

Obviously, in such case the value returned by the *default* callable **must not** be or contain a `tuple`:

```
>>> def bad_timestruct(obj):
...   if isinstance(obj, struct_time):
...     return {'__class__': 'time.struct_time', '__init__': tuple(obj)}
...   else:
...     raise ValueError('%r is not JSON serializable' % (obj,))
>>> dumps(lt, iterable_mode=IM_ONLY_LISTS, default=bad_timestruct)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: (...) is not JSON serializable
```

mapping_mode

By default a value that implements the *mapping* protocol gets encoded as a `JSON` object:

```
>>> from collections import Counter
>>> d = {"a":1,"b":2,"c":3}
>>> c = Counter(d)
>>> dumps([c, d])
'[{"a":1,"b":2,"c":3},{"a":1,"b":2,"c":3}]'
```

When that's not appropriate, for example because you want to use a different way to encode them, you may specify *mapping_mode* to `MM_ONLY_DICTS`:

```
>>> dumps(d, mapping_mode=MM_ONLY_DICTS)
'{"a":1,"b":2,"c":3}'
>>> dumps(c, mapping_mode=MM_ONLY_DICTS)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Counter(...) is not JSON serializable
```

and thus you can use the *default* argument:

```
>>> def counter(obj):
...   if isinstance(obj, Counter):
...     return {'__class__': 'collections.Counter', '__init__': dict(obj)}
...   else:
...     raise ValueError('%r is not JSON serializable' % obj)
>>> dumps(c, mapping_mode=MM_ONLY_DICTS, default=counter)
'{"__class__":"collections.Counter","__init__":{"a":1,"b":2,"c":3}}'
```

Obviously, in such case the value returned by the *default* callable **must not** be or contain mappings other than plain `dicts`:

```
>>> from collections import OrderedDict
>>> def bad_counter(obj):
...   if isinstance(obj, Counter):
...     return {'__class__': 'time.struct_time', '__init__': OrderedDict(obj)}
...   else:
...     raise ValueError('%r is not JSON serializable' % (obj,))
>>> dumps(c, mapping_mode=MM_ONLY_DICTS, default=bad_counter)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: OrderedDict([('a', 1), ('b', 2), ('c', 3)]) is not JSON serializable
```

Normally, dumping a dictionary containing *non-string* keys raises a `TypeError` exception:

```
>>> dumps({-1: 'minus-one'})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: keys must be strings
```

Setting *mapping_mode* to `MM_COERCE_KEYS_TO_STRINGS` such keys will be converted to their string representation:

```
>>> dumps({-1: 'minus-one', True: "good", False: "bad", None: "ugly"},
...        mapping_mode=MM_COERCE_KEYS_TO_STRINGS)
'{"-1":"minus-one","True":"good","False":"bad","None":"ugly"}'
```

Alternatively, by providing a *default* function you can have finer control on how they should be encoded. For example the following mimics the default behaviour of the standard library `json` module:

```
>>> def mimic_stdlib_json(obj):
...   if isinstance(obj, dict):
...     result = {}
...     for key in obj:
...       if key is True:
...         result['true'] = obj[key]
...       elif key is False:
...         result['false'] = obj[key]
...       elif key is None:
...         result['null'] = obj[key]
...       elif isinstance(key, (int, float)):
...         result[str(key)] = obj[key]
...       else:
...         raise TypeError('keys must be str, int, float, bool or None')
...     return result
...   else:
...     raise ValueError('%r is not JSON serializable' % (obj,))
>>> dumps({True: 'good', False: 'bad', None: 'ugly'},
...        default=mimic_stdlib_json)
'{"true":"good","false":"bad","null":"ugly"}'
```

Warning: This can lead to an infinite recursion error, if the *default* function returns a dictionary that still contains *non-string* keys:

```
>>> dumps({True: 'vero', False: 'falso'},
...        default=lambda map: map)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RecursionError: maximum recursion depth exceeded
```

1.3.2 dump() function

`rapidjson.dump`(*obj*, *stream*, *, *skipkeys=False*, *ensure_ascii=True*, *write_mode=WM_COMPACT*, *indent=4*, *default=None*, *sort_keys=False*, *number_mode=None*, *datetime_mode=None*, *uuid_mode=None*, *bytes_mode=BM_UTF8*, *iterable_mode=IM_ANY_ITERABLE*, *mapping_mode=MM_ANY_MAPPING*, *chunk_size=65536*, *allow_nan=True*)

Encode given Python *obj* instance into a `JSON` stream.

Parameters

- **obj** – the value to be serialized
- **stream** – a *file-like* instance
- **skipkeys** (*bool*) – whether skip invalid `dict` keys

- **ensure_ascii** (*bool*) – whether the output should contain only ASCII characters
- **write_mode** (*int*) – enable particular pretty print behaviors
- **indent** – indentation width or string to produce pretty printed JSON
- **default** (*callable*) – a function that gets called for objects that can't otherwise be serialized
- **sort_keys** (*bool*) – whether dictionary keys should be sorted alphabetically
- **number_mode** (*int*) – enable particular behaviors in handling numbers
- **datetime_mode** (*int*) – how should `datetime`, `time` and `date` instances be handled
- **uuid_mode** (*int*) – how should `UUID` instances be handled
- **bytes_mode** (*int*) – how should `bytes` instances be handled
- **iterable_mode** (*int*) – how should *iterable* values be handled
- **mapping_mode** (*int*) – how should *mapping* values be handled
- **chunk_size** (*int*) – write the stream in chunks of this size at a time
- **allow_nan** (*bool*) – *compatibility* flag equivalent to `number_mode=NM_NAN`

The function has the same behaviour as *dumps()*, except that it requires an additional mandatory parameter, a *file-like* writable *stream* instance:

```
>>> stream = io.BytesIO()
>>> dump('bar', stream)
>>> stream.getvalue()
b'"bar"'
```

The target may also be a text stream[1]:

```
>>> stream = io.StringIO()
>>> dump(r'¯\_()_/¯', stream)
>>> stream.getvalue() == dumps(r'¯\_()_/¯')
True
```

chunk_size

The *chunk_size* argument determines the size of the *buffer* used to feed the *stream*: the greater the value, the fewer calls will be made to its `write()` method.

Consult the *dumps()* documentation for details on all other arguments.

[1] A *text stream* is recognized by checking the presence of an `encoding` member attribute on the instance.

1.3.3 loads() function

`rapidjson.loads`(*string*, *, *object_hook=None*, *number_mode=None*, *datetime_mode=None*, *uuid_mode=None*, *parse_mode=None*, *allow_nan=True*)

Decode the given `JSON` formatted value into Python object.

Parameters

- **string** – The JSON string to parse, either a Unicode `str` instance or a `bytes` or a `bytearray` instance containing an `UTF-8` encoded value
- **object_hook** (*callable*) – an optional function that will be called with the result of any object literal decoded (a `dict`) and should return the value to use instead of the `dict`
- **number_mode** (*int*) – enable particular behaviors in handling numbers
- **datetime_mode** (*int*) – how should `datetime` and `date` instances be handled
- **uuid_mode** (*int*) – how should `UUID` instances be handled
- **parse_mode** (*int*) – whether the parser should allow non-standard JSON extensions
- **allow_nan** (*bool*) – *compatibility* flag equivalent to `number_mode=NM_NAN`

Returns

An equivalent Python object.

Raises

- **ValueError** – if an invalid argument is given
- ***JSONDecodeError*** – if *string* is not a valid `JSON` value

object_hook

object_hook may be used to inject a custom deserializer that can replace any `dict` instance found in the JSON structure with a *derived* object instance:

```
>>> class Point(object):
...   def __init__(self, x, y):
...     self.x = x
...     self.y = y
...   def __repr__(self):
...     return 'Point(%s, %s)' % (self.x, self.y)
...
>>> def point_dejsonifier(d):
...   if 'x' in d and 'y' in d:
...     return Point(d['x'], d['y'])
...   else:
...     return d
...
>>> loads('{"x":1,"y":2}', object_hook=point_dejsonifier)
Point(1, 2)
```

number_mode

The *number_mode* argument selects different behaviors in handling numeric values.

By default *non-numbers* (`nan`, `inf`, `-inf`) are recognized, because `NM_NAN` is *on* by default:

```
>>> loads('[NaN, Infinity]')
[nan, inf]
>>> loads('[NaN, Infinity]', number_mode=NM_NAN)
[nan, inf]
```

Explicitly setting *number_mode* or using the compatibility option *allow_nan* you can avoid that and obtain a `ValueError` exception instead:

```
>>> loads('[NaN, Infinity]', number_mode=NM_NATIVE)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
rapidjson.JSONDecodeError: ... Out of range float values are not JSON compliant
>>> loads('[NaN, Infinity]', allow_nan=False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
rapidjson.JSONDecodeError: ... Out of range float values are not JSON compliant
```

Normally all floating point literals present in the JSON structure will be loaded as Python `float` instances, with *NM_DECIMAL* they will be returned as `Decimal` instances instead:

```
>>> loads('1.2345')
1.2345
>>> loads('1.2345', number_mode=NM_DECIMAL)
Decimal('1.2345')
```

When you can be sure that all the numeric values are constrained within the architecture's hardware limits you can get a sensible speed gain with the *NM_NATIVE* flag. While this is quite faster, integer literals that do not fit into the underlying C library `long long` limits will be converted (*truncated*) to `double` numbers:

```
>>> loads('12345678901234567890123456789 0')
12345678901234567890123456789 0
>>> loads('12345678901234567890123456789 0', number_mode=NM_NATIVE)
1.2345678901234566e+29
```

These flags can be combined together:

```
>>> loads('[-1, NaN, 3.14159265358979323846264338327950288419 71]',
...       number_mode=NM_DECIMAL | NM_NAN)
[-1, Decimal('NaN'), Decimal('3.14159265358979323846264338327950288419 71')]
```

with the exception of *NM_NATIVE* and *NM_DECIMAL*, that does not make sense since there's little point in creating `Decimal` instances out of possibly truncated float literals:

datetime_mode

With *datetime_mode* you can enable recognition of string literals containing an ISO 8601 representation as either `date`, `datetime` or `time` instances:

```
>>> loads('"2016-01-02T01:02:03+01:00"')
'2016-01-02T01:02:03+01:00'
>>> loads('"2016-01-02T01:02:03+01:00"', datetime_mode=DM_ISO8601)
datetime.datetime(2016, 1, 2, 1, 2, 3, tzinfo=...delta(...3600)))
>>> loads('"2016-01-02T01:02:03-01:00"', datetime_mode=DM_ISO8601)
datetime.datetime(2016, 1, 2, 1, 2, 3, tzinfo=...delta(...82800)))
>>> loads('"2016-01-02"', datetime_mode=DM_ISO8601)
datetime.date(2016, 1, 2)
>>> loads('"01:02:03+01:00"', datetime_mode=DM_ISO8601)
datetime.time(1, 2, 3, tzinfo=...delta(...3600)))
```

It can be combined with *DM_SHIFT_TO_UTC* to *always* obtain values in the UTC timezone:

```
>>> mode = DM_ISO8601 | DM_SHIFT_TO_UTC
>>> loads('"2016-01-02T01:02:03+01:00"', datetime_mode=mode)
datetime.datetime(2016, 1, 2, 0, 2, 3, tzinfo=...utc)
```

Note: This option is somewhat limited when the value is a non-naïve time literal because negative values cannot be represented by the underlying Python type, so it cannot adapt such values reliably:

```
>>> mode = DM_ISO8601 | DM_SHIFT_TO_UTC
>>> loads('"00:01:02+00:00"', datetime_mode=mode)
datetime.time(0, 1, 2, tzinfo=...utc)
>>> loads('"00:01:02+01:00"', datetime_mode=mode)
Traceback (most recent call last):
  ...
ValueError: ... Time literal cannot be shifted to UTC: 00:01:02+01:00
```

If you combine it with *DM_NAIVE_IS_UTC* then all values without a timezone will be assumed to be relative to UTC:

```
>>> mode = DM_ISO8601 | DM_NAIVE_IS_UTC
>>> loads('"2016-01-02T01:02:03"', datetime_mode=mode)
datetime.datetime(2016, 1, 2, 1, 2, 3, tzinfo=...utc)
>>> loads('"2016-01-02T01:02:03+01:00"', datetime_mode=mode)
datetime.datetime(2016, 1, 2, 1, 2, 3, tzinfo=...delta(...3600)))
>>> loads('"01:02:03"', datetime_mode=mode)
datetime.time(1, 2, 3, tzinfo=...utc)
```

Yet another combination is with *DM_IGNORE_TZ* to ignore the timezone and obtain naïve values:

```
>>> mode = DM_ISO8601 | DM_IGNORE_TZ
>>> loads('"2016-01-02T01:02:03+01:00"', datetime_mode=mode)
datetime.datetime(2016, 1, 2, 1, 2, 3)
>>> loads('"01:02:03+01:00"', datetime_mode=mode)
datetime.time(1, 2, 3)
```

The *DM_UNIX_TIME* cannot be used here, because there isn't a reasonable heuristic to disambiguate between plain numbers and timestamps:

```
>>> loads('[1,2,3]', datetime_mode=DM_UNIX_TIME)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Invalid datetime_mode, can deserialize only from ISO8601
```

uuid_mode

With *uuid_mode* you can enable recognition of string literals containing two different representations of `UUID` values:

```
>>> loads('"aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"')
'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa'
>>> loads('"aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"',
...       uuid_mode=UM_CANONICAL)
UUID('aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa')
>>> loads('"aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"',
...       uuid_mode=UM_HEX)
UUID('aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa')
>>> loads('"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"',
...       uuid_mode=UM_CANONICAL)
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
>>> loads('"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"',
...       uuid_mode=UM_HEX)
UUID('aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa')
```

parse_mode

With *parse_mode* you can tell the parser to be *relaxed*, allowing either `C++/JavaScript` like comments (*PM_COMMENTS*):

```
>>> loads('"foo" // one line of explanation')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
rapidjson.JSONDecodeError: Parse error at offset 6: The document root must not be␣
↪followed by other values.
>>> loads('"bar" /* detailed explanation */')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
rapidjson.JSONDecodeError: Parse error at offset 6: The document root must not be␣
↪followed by other values.
>>> loads('"foo" // one line of explanation', parse_mode=PM_COMMENTS)
'foo'
>>> loads('"bar" /* detailed explanation */', parse_mode=PM_COMMENTS)
'bar'
```

or *trailing commas* at the end of arrays and objects (*PM_TRAILING_COMMAS*):

```
>>> loads('[1,]')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
rapidjson.JSONDecodeError: Parse error at offset 3: Invalid value.
>>> loads('[1,]', parse_mode=PM_TRAILING_COMMAS)
[1]
>>> loads('{"one": 1,}', parse_mode=PM_TRAILING_COMMAS)
{'one': 1}
```

or both:

```
>>> loads('[1, /* 2, */ 3,]')
Traceback (most recent call last):
  ...
rapidjson.JSONDecodeError: Parse error at offset 4: Invalid value.
>>> loads('[1, /* 2, */ 3,]', parse_mode=PM_COMMENTS | PM_TRAILING_COMMAS)
[1, 3]
```

1.3.4 load() function

`rapidjson.load`(*stream*, *, *object_hook=None*, *number_mode=None*, *datetime_mode=None*, *uuid_mode=None*, *parse_mode=None*, *chunk_size=65536*, *allow_nan=True*)

Decode the given Python file-like *stream* containing a `JSON` formatted value into Python object.

Parameters

- **stream** – a file-like object
- **object_hook** (*callable*) – an optional function that will be called with the result of any object literal decoded (a `dict`) and should return the value to use instead of the `dict`
- **number_mode** (*int*) – enable particular behaviors in handling numbers
- **datetime_mode** (*int*) – how should `datetime` and `date` instances be handled
- **uuid_mode** (*int*) – how should `UUID` instances be handled
- **parse_mode** (*int*) – whether the parser should allow non-standard JSON extensions
- **chunk_size** (*int*) – read the stream in chunks of this size at a time
- **allow_nan** (*bool*) – *compatibility* flag equivalent to `number_mode=NM_NAN`

Returns

An equivalent Python object.

Raises

- **ValueError** – if an invalid argument is given
- ***JSONDecodeError*** – if *string* is not a valid `JSON` value

The function has the same behaviour as *loads()*, except for the kind of the first argument that is expected to be file-like object instead of a string:

```
>>> load(io.StringIO('"Naïve"'))
'Naïve'
>>> load(io.BytesIO(b'["string", {"kind": "object"}, 3.14159]'))
['string', {'kind': 'object'}, 3.14159]
```

chunk_size

The *chunk_size* argument determines the size of the *buffer* used to load the *stream*: the greater the value, the fewer calls will be made to its `read()` method.

Consult the `loads()` documentation for details on all other arguments.

1.3.5 Encoder class

class rapidjson.**Encoder**(*skip_invalid_keys=False*, *ensure_ascii=True*, *write_mode=WM_COMPACT*, *indent=4*, *sort_keys=False*, *number_mode=None*, *datetime_mode=None*, *uuid_mode=None*, *bytes_mode=BM_UTF8*, *iterable_mode=IM_ANY_ITERABLE*, *mapping_mode=MM_ANY_MAPPING*)

Class-based `dumps()`-like functionality.

Parameters

- **skip_invalid_keys** (*bool*) – whether invalid `dict` keys *will be skipped*
- **ensure_ascii** (*bool*) – whether the output should contain *only ASCII characters*
- **write_mode** (*int*) – enable particular *pretty print* behaviors
- **indent** – either an integer, the indentation width, or a string, that will be used as line separator, when *write_mode* is not `WM_COMPACT`
- **sort_keys** (*bool*) – whether dictionary keys should be *sorted alphabetically*
- **number_mode** (*int*) – enable particular *behaviors in handling numbers*
- **datetime_mode** (*int*) – how should *datetime, time and date instances be handled*
- **uuid_mode** (*int*) – how should *UUID instances be handled*
- **bytes_mode** (*int*) – how should *bytes instances be handled*
- **iterable_mode** (*int*) – how should *iterable* values be handled
- **mapping_mode** (*int*) – how should *mapping* values be handled

Attributes

bytes_mode

Type
int

How bytes values should be treated.

datetime_mode

Type
int

Whether and how datetime values should be encoded.

ensure_ascii

Type
bool

Whether the output should contain only ASCII characters.

`indent_char`

Type
str

What will be used as end-of-line character.

`indent_count`

Type
int

The indentation width.

`iterable_mode`

Type
int

Whether *iterables* will be generically encoded as `JSON` arrays or not.

`mapping_mode`

Type
int

Whether *mappings* will be generically encoded as `JSON` objects or not.

`number_mode`

Type
int

The encoding behavior with regards to numeric values.

`skip_invalid_keys`

Type
bool

Whether invalid keys shall be skipped.

Note: *skip_invalid_keys* is a backward compatible alias of new `MM_SKIP_NON_STRING_KEYS` *mapping mode*.

`sort_keys`

Type
bool

Whether dictionary keys shall be sorted alphabetically.

Note: *sort_keys* is a backward compatible alias of new `MM_SORT_KEYS` *mapping mode*.

`uuid_mode`

Type
int

Whether and how UUID values should be encoded

write_mode

Type
int

Whether the output should be pretty printed or not.

Methods

__call__(*obj*, *stream=None*, *, *chunk_size=65536*)

Parameters

- **obj** – the value to be encoded
- **stream** – a *file-like* instance
- **chunk_size** (*int*) – write the stream in chunks of this size at a time

Returns
a string with the JSON encoded *value*, when *stream* is None

When *stream* is specified, the encoded result will be written there, possibly in chunks of *chunk_size* bytes at a time, and the return value will be None.

default(*value*)

Parameters
value – the Python value to be encoded

Returns
a *JSON-serializable* value

If implemented, this method is called whenever the serialization machinery finds a Python object that it does not recognize: if possible, the method should returns a *JSON encodable* version of the *value*, otherwise raise a TypeError:

```
>>> class Point:
...   def __init__(self, x, y):
...     self.x = x
...     self.y = y
...
>>> point = Point(1,2)
>>> Encoder()(point)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <__main__.Point object at ...> is not JSON serializable
>>> class PointEncoder(Encoder):
...   def default(self, obj):
...     if isinstance(obj, Point):
...       return {'x': obj.x, 'y': obj.y}
...     else:
...       raise TypeError('%r is not JSON serializable' % obj)
...
>>> pe = PointEncoder(sort_keys=True)
>>> pe(point)
'{"x":1,"y":2}'
```

When you want to treat your `bytes` instances in a special way, you can use *BM_NONE* together with this method:

```
>>> class HexifyBytes(Encoder):
...   def default(self, obj):
...     if isinstance(obj, bytes):
...       return obj.hex()
...     else:
...       return obj
...
>>> small_numbers = bytes([1, 2, 3])
>>> hb = HexifyBytes(bytes_mode=BM_NONE)
>>> hb(small_numbers)
'"010203"'
```

Likewise, when you want full control on how an *iterable* such as a `tuple` should be encoded, you can use *IM_ONLY_LISTS* and implement a suitable `default()` method:

```
>>> from time import localtime, struct_time
>>> class ObjectifyStructTime(Encoder):
...   def default(self, obj):
...     if isinstance(obj, struct_time):
...       return {'__class__': 'time.struct_time', '__init__': list(obj)}
...     else:
...       raise ValueError('%r is not JSON serializable' % obj)
...
>>> obst = ObjectifyStructTime(iterable_mode=IM_ONLY_LISTS)
>>> obst(localtime())
'[2020,11,28,19,55,40,5,333,0]'
```

Similarly, when you want full control on how a *mapping* other than plain `dict` should be encoded, you can use *MM_ONLY_DICTS* and implement a `default()` method:

```
>>> from collections import OrderedDict
>>> class ObjectifyOrderedDict(Encoder):
...   def default(self, obj):
...     if isinstance(obj, OrderedDict):
...       return {'__class__': 'collections.OrderedDict',
...               '__init__': list(obj.items())}
...     else:
...       raise ValueError('%r is not JSON serializable' % obj)
...
>>> ood = ObjectifyOrderedDict(mapping_mode=MM_ONLY_DICTS)
>>> ood(OrderedDict(('a', 1), ('b', 2))))
'{"__class__":"collections.OrderedDict","__init__":[["a",1],["b",2]]}'
```

1.3.6 Decoder class

class rapidjson.**Decoder**(*number_mode=None*, *datetime_mode=None*, *uuid_mode=None*, *parse_mode=None*)

Class-based `loads()`-like functionality.

Parameters

- **number_mode** (*int*) – enable particular *behaviors in handling numbers*
- **datetime_mode** (*int*) – how should *datetime, time and date instances be handled*
- **uuid_mode** (*int*) – how should *UUID instances be handled*
- **parse_mode** (*int*) – whether the parser should allow *non-standard JSON extensions*

Attributes

datetime_mode

Type
int

The datetime mode, whether and how datetime literals will be recognized.

number_mode

Type
int

The number mode, whether numeric literals will be decoded.

parse_mode

Type
int

The parse mode, whether comments and trailing commas are allowed.

uuid_mode

Type
int

The UUID mode, whether and how UUID literals will be recognized.

Methods

__call__(*json*, ***, *chunk_size=65536*)

Parameters

- **json** – either a `str` instance, an *UTF-8* `bytes` instance or a *file-like* stream, containing the `JSON` to be decoded
- **chunk_size** (*int*) – in case of a stream, it will be read in chunks of this size

Returns
a Python value

```
>>> decoder = Decoder()
>>> decoder('"€ 0.50"')
'€ 0.50'
>>> decoder(io.StringIO('"€ 0.50"'))
'€ 0.50'
>>> decoder(io.BytesIO(b'"\xe2\x82\xac 0.50"'))
'€ 0.50'
>>> decoder(b'"\xe2\x82\xac 0.50"')
'€ 0.50'
```

end_array(*sequence*)

Parameters
sequence – an instance implement the *mutable sequence* protocol

Returns
a new value

This is called, if implemented, when a *JSON array* has been completely parsed, and can be used replace it with an arbitrary different value:

```
>>> class TupleDecoder(Decoder):
...   def end_array(self, a):
...     return tuple(a)
...
>>> td = TupleDecoder()
>>> res = td('[{"one": [1]}, {"two":[2,3]}]')
>>> isinstance(res, tuple)
True
>>> res[0]
{'one': (1,)}
>>> res[1]
{'two': (2, 3)}
```

end_object(*mapping*)

Parameters
mapping – an instance representing the JSON object, either one implementing the *mapping protocol* or a list containing `(key, value)` tuples

Returns
a new value

This is called, if implemented, when a *JSON object* has been completely parsed, and can be used to replace it with an arbitrary different value, like what can be done with the `object_hook` argument of the *loads()* function:

```
>>> class Point(object):
...   def __init__(self, x, y):
...     self.x = x
...     self.y = y
...   def __repr__(self):
...     return 'Point(%s, %s)' % (self.x, self.y)
...
>>> class PointDecoder(Decoder):
...   def end_object(self, d):
```

(continues on next page)

(continued from previous page)

```
...     if 'x' in d and 'y' in d:
...       return Point(d['x'], d['y'])
...     else:
...       return d
...
>>> pd = PointDecoder()
>>> pd('{"x":1,"y":2}')
Point(1, 2)
```

When *start_object()* returns a `list` instance, then the *mapping* argument is actually a list of tuples.

start_object()

Returns
either a list or mapping instance

This method, when implemented, is called whenever a new *JSON object* is found: it must return either a list or an instance implementing the *mapping protocol*.

It can be used to select a different implementation than the standard `dict` used by default:

```
>>> from collections import OrderedDict
>>> class OrderedDecoder(Decoder):
...   def start_object(self):
...     return OrderedDict()
...
>>> od = OrderedDecoder()
>>> type(od('{"foo": "bar"}'))
<class 'collections.OrderedDict'>
```

By returning a `list` you obtain a different handling of *JSON objects*: instead of translating them into Python maps as soon as they are found, their *key-value tuples* are accumulated in the given list; when the *JSON object* has been completely parsed, then the sequence is passed to the method *end_object()*, if implemented, that will reasonably transmogrify it into some kind of dictionary. When that method is missing, then the list is kept as is, thus representing all objects in the JSON origin as lists of key-value tuples in Python:

```
>>> class KVPairsDecoder(Decoder):
...   def start_object(self):
...     return []
...
>>> kvpd = KVPairsDecoder()
>>> kvpd('{"one": 1, "two": 2}')
[('one', 1), ('two', 2)]
```

string(*s*)

Parameters
s – a `str` instance

Returns
a new value

This method, when implemented, is called whenever a *JSON string* has been completely parsed, and can be used to replace it with an arbitrary different value:

```
>>> class SwapStringCase(Decoder):
...   def string(self, s):
...     return s.swapcase()
...
>>> ssc = SwapStringCase()
>>> ssc('"Hello World!"')
'hELLO wORLD!'
```

Note that it is called **after** the recognition of dates and UUIDs, when *datetime_mode* and/or *uuid_mode* are specified:

```
>>> class DDMMYYYY(Decoder):
...   def string(self, s):
...     if len(s) == 8 and s.isdigit():
...       dd = int(s[:2])
...       mm = int(s[2:4])
...       yyyy = int(s[-4:])
...       return (yyyy, mm, dd)
...     return s
...
>>> ddmmyyyy = DDMMYYYY(datetime_mode=DM_ISO8601)
>>> ddmmyyyy('["2017-08-21", "21082017"]')
[datetime.date(2017, 8, 21), (2017, 8, 21)]
```

1.3.7 Validator class

class rapidjson.**Validator**(*json_schema*)

Parameters

json_schema – the JSON schema, specified as a `str` instance or an *UTF-8* `bytes` instance

Raises

JSONDecodeError – if *json_schema* is not a valid `JSON` value

__call__(*json*)

Parameters

json – the `JSON` value, specified as a `str` instance or an *UTF-8* `bytes` instance, that will be validated

Raises

JSONDecodeError – if *json* is not a valid `JSON` value

The given *json* value will be validated accordingly to the *schema*: a *ValidationError* will be raised if the validation fails, and the exception will contain three arguments, respectively the type of the error, the position in the schema and the position in the `JSON` document where the error occurred:

```
>>> validate = Validator('{"required": ["a", "b"]}')
>>> validate('{"a": null, "b": 1}')
>>> try:
...   validate('{"a": null, "c": false}')
... except ValidationError as error:
...   print(error.args)
```

(continues on next page)

(continued from previous page)

```
...
('required', '#', '#')
```

```
>>> validate = Validator('{"type": "array",'
...                      ' "items": {"type": "string"},'
...                      ' "minItems": 1}')
>>> validate('["foo", "bar"]')
>>> try:
...   validate('[]')
... except ValidationError as error:
...   print(error.args)
...
('minItems', '#', '#')
```

```
>>> try:
...   validate('[1]')
... except ValidationError as error:
...   print(error.args)
...
('type', '#/items', '#/0')
```

When *json* is not a valid JSON document, a `JSONDecodeError` is raised instead:

```
>>> validate('x')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
rapidjson.JSONDecodeError: Invalid JSON
```

1.3.8 RawJSON class

class rapidjson.**RawJSON**(*value*)

Preserialized JSON string.

Parameters

value (*str*) – the string that rapidjson should use verbatim when serializing this object

Some applications might decide to store JSON-serialized objects in their database, but might need to assemble them in a bigger JSON.

The `RawJSON` class serves this purpose. When serialized, the string value provided will be used verbatim, whitespace included:

```
>>> raw_list = RawJSON('[1, 2,3]')
>>> dumps({'foo': raw_list})
'{"foo":[1, 2,3]}'
```

Warning: `python-rapidjson` runs no checks on the preserialized data. This means that it can potentially output invalid JSON, if you provide it:

```
>>> raw_list = RawJSON('[1, ')
>>> dumps({'foo': raw_list})
'{"foo":[1, }'
```

`rapidjson.__author__`

The authors of the module.

`rapidjson.__version__`

The version of the module.

`rapidjson.__rapidjson_version__`

The version of the RapidJSON library this module is built with.

`rapidjson.__rapidjson_exact_version__`

The *exact* version of the RapidJSON library, as determined by `git describe`.

datetime_mode related constants

`rapidjson.DM_NONE`

This is the default *datetime_mode*: *neither* `date` *nor* `datetime` *nor* `time` instances are recognized by *dumps()* and *loads()*.

`rapidjson.DM_ISO8601`

In this *datetime_mode* mode *dumps()* and *loads()* handle `date`, `datetime` *and* `date` instances representing those values using the ISO 8601 format.

`rapidjson.DM_UNIX_TIME`

This mode tells RapidJSON to serialize `date`, `datetime` and `time` as *numeric timestamps*: for the formers it is exactly the result of their `timestamp()` method, that is as the number of seconds passed since `EPOCH`; `date` instances are serialized as the corresponding `datetime` instance with all the *time* slots set to 0; `time` instances are serialized as the number of seconds since midnight.

Since this is obviously *irreversible*, this flag is usable **only** for *dumps()*: an error is raised when passed to *loads()*.

`rapidjson.DM_ONLY_SECONDS`

This is usable in combination with *DM_UNIX_TIME* so that an integer representation is used, ignoring *microseconds*.

`rapidjson.DM_IGNORE_TZ`

This can be used combined with *DM_ISO8601* or *DM_UNIX_TIME*, to ignore the value's timezones.

`rapidjson.DM_NAIVE_IS_UTC`

This can be used combined with *DM_ISO8601* or *DM_UNIX_TIME*, to tell RapidJSON that the naïve values (that is, without an explicit timezone) are already in UTC timezone.

`rapidjson.DM_SHIFT_TO_UTC`

This can be used combined with *DM_ISO8601* or *DM_UNIX_TIME*, to always *shift* values the UTC timezone.

uuid_mode related constants

`rapidjson.UM_NONE`

This is the default *uuid_mode*: `UUID` instances are *not* recognized by `dumps()` and `loads()`.

`rapidjson.UM_CANONICAL`

In this *uuid_mode*, `loads()` recognizes string values containing the `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx` canonical representation as `UUID` instances; `dumps()` emits same kind of representation for `UUID` instances as a string value.

`rapidjson.UM_HEX`

In this *uuid_mode* `loads()` recognizes string values containing exactly 32 hex digits *or* the `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx` canonical representation as `UUID` instances; `dumps()` emits the 32 hex digits of `UUID` instances as a string value.

number_mode related constants

`rapidjson.NM_NONE`

This is the default *number_mode*: numeric values can be as wide as the memory allows.

`rapidjson.NM_DECIMAL`

In this *number_mode* `loads()` will return floating point values as `Decimal` instances instead of `float`; `dumps()` will serialize `Decimal` instances like any other `float` number.

`rapidjson.NM_NAN`

This enables *non-numbers* (i.e. `nan`, `+inf` and `-inf`) handling in both directions.

`rapidjson.NM_NATIVE`

In this alternative *number_mode* numeric values must fit into the underlying C library limits, with a considerable speed benefit.

parse_mode related constants

`rapidjson.PM_NONE`

This is the default *parse_mode*: with the exception of the *NaN and Infinite* recognition active by default, the parser is in *strict mode*.

`rapidjson.PM_COMMENTS`

In this *parse_mode*, the parser allows and ignores one-line `// ...` and multi-line `/* ... */` comments

`rapidjson.PM_TRAILING_COMMAS`

In this *parse_mode*, the parser allows and ignores trailing commas at the end of *arrays* and *objects*.

bytes_mode related constants

`rapidjson.BM_NONE`

This disables the default handling mode (`BM_UTF8`) of `bytes` instances: they won't be treated in any special way and will raise a `TypeError` exception when encountered. On the other hand, in this mode they can be managed by a *default* handler.

`rapidjson.BM_UTF8`

This is the default setting for *bytes_mode*: any `bytes` instance will be assumed to be an `UTF-8` encoded string, and decoded accordingly.

iterable_mode related constants

`rapidjson.`**`IM_ANY_ITERABLE`**

This is the default setting for *iterable_mode*: any iterable will be dumped as a `JSON` array.

`rapidjson.`**`IM_ONLY_LISTS`**

This disables the default handling mode (*`IM_ANY_ITERABLE`*) of *iterable* values (with the exception of `strings` and `lists`): they won't be treated in any special way and will raise a `TypeError` exception when encountered. On the other hand, in this mode they can be managed by a *default* handler.

mapping_mode related constants

`rapidjson.`**`MM_ANY_MAPPING`**

This is the default setting for *mapping_mode*: any mapping will be dumped as a `JSON` object.

`rapidjson.`**`MM_ONLY_DICTS`**

This disables the default handling mode (*`MM_ANY_MAPPING`*) of generic *mapping* values, they won't be treated like `dicts` and will raise a `TypeError` exception when encountered. On the other hand, in this mode they can be managed by a *default* handler.

`rapidjson.`**`MM_COERCE_KEYS_TO_STRINGS`**

Since `JSON` objects require that all keys must be of type string, when a mapping contains *non-string* keys, it will be processed by the *default* handler if available, otherwise a `TypeError` exception is raised.

Alternatively, you can use the this option to automatically render such keys as their `str()` representation.

`rapidjson.`**`MM_SKIP_NON_STRING_KEYS`**

In this mode, dict keys that are not of a basic type (`str`, `int`, `float`, `bool`, `None`) will be skipped instead of raising a `TypeError`.

`rapidjson.`**`MM_SORT_KEYS`**

Alphabetically order dictionary keys.

Exceptions

exception `rapidjson.`**`JSONDecodeError`**

A subclass of `ValueError`, raised when trying to parse an invalid `JSON`, either by *`Validator`* objects or by *`loads()`* function.

exception `rapidjson.`**`ValidationError`**

Exception raised by *`Validator`* objects, a subclass of `ValueError`.

Its *args* attribute is a tuple with three string values, respectively the *schema keyword* that generated the failure, its *JSON pointer* and a *JSON pointer* to the error location in the (invalid) document.

write_mode related constants

`rapidjson.WM_COMPACT`

The default dump mode, without any extra whitespace.

`rapidjson.WM_PRETTY`

This selects the RapidJSON `PrettyWriter`, to produce more readable `JSON`: each array's item and object's key will be preceded by a newline, and nested structures will be indented.

`rapidjson.WM_SINGLE_LINE_ARRAY`

This tells the `PrettyWriter` to emit arrays on a single line, instead of separating items with a newline.

1.4 Indices and tables

- genindex
- search

INDEX

Symbols

__author__ (*in module rapidjson*), 34
__call__() (*rapidjson.Decoder method*), 29
__call__() (*rapidjson.Encoder method*), 27
__call__() (*rapidjson.Validator method*), 32
__rapidjson_exact_version__ (*in module rapidjson*), 34
__rapidjson_version__ (*in module rapidjson*), 34
__version__ (*in module rapidjson*), 34

B

BM_NONE (*in module rapidjson*), 35
BM_UTF8 (*in module rapidjson*), 35
bytes_mode (*rapidjson.Encoder attribute*), 25

D

datetime_mode (*rapidjson.Decoder attribute*), 29
datetime_mode (*rapidjson.Encoder attribute*), 25
Decoder (*class in rapidjson*), 29
default() (*rapidjson.Encoder method*), 27
DM_IGNORE_TZ (*in module rapidjson*), 34
DM_ISO8601 (*in module rapidjson*), 34
DM_NAIVE_IS_UTC (*in module rapidjson*), 34
DM_NONE (*in module rapidjson*), 34
DM_ONLY_SECONDS (*in module rapidjson*), 34
DM_SHIFT_TO_UTC (*in module rapidjson*), 34
DM_UNIX_TIME (*in module rapidjson*), 34
dump() (*in module rapidjson*), 18
dumps() (*in module rapidjson*), 8

E

Encoder (*class in rapidjson*), 25
end_array() (*rapidjson.Decoder method*), 30
end_object() (*rapidjson.Decoder method*), 30
ensure_ascii (*rapidjson.Encoder attribute*), 25

I

IM_ANY_ITERABLE (*in module rapidjson*), 36
IM_ONLY_LISTS (*in module rapidjson*), 36
indent_char (*rapidjson.Encoder attribute*), 26
indent_count (*rapidjson.Encoder attribute*), 26
iterable_mode (*rapidjson.Encoder attribute*), 26

J

JSONDecodeError, 36

L

load() (*in module rapidjson*), 24
loads() (*in module rapidjson*), 20

M

mapping_mode (*rapidjson.Encoder attribute*), 26
MM_ANY_MAPPING (*in module rapidjson*), 36
MM_COERCE_KEYS_TO_STRINGS (*in module rapidjson*), 36
MM_ONLY_DICTS (*in module rapidjson*), 36
MM_SKIP_NON_STRING_KEYS (*in module rapidjson*), 36
MM_SORT_KEYS (*in module rapidjson*), 36

N

NM_DECIMAL (*in module rapidjson*), 35
NM_NAN (*in module rapidjson*), 35
NM_NATIVE (*in module rapidjson*), 35
NM_NONE (*in module rapidjson*), 35
number_mode (*rapidjson.Decoder attribute*), 29
number_mode (*rapidjson.Encoder attribute*), 26

P

parse_mode (*rapidjson.Decoder attribute*), 29
PM_COMMENTS (*in module rapidjson*), 35
PM_NONE (*in module rapidjson*), 35
PM_TRAILING_COMMAS (*in module rapidjson*), 35

R

RawJSON (*class in rapidjson*), 33

S

skip_invalid_keys (*rapidjson.Encoder attribute*), 26
sort_keys (*rapidjson.Encoder attribute*), 26
start_object() (*rapidjson.Decoder method*), 31
string() (*rapidjson.Decoder method*), 31

U

UM_CANONICAL (*in module rapidjson*), 35
UM_HEX (*in module rapidjson*), 35
UM_NONE (*in module rapidjson*), 35
uuid_mode (*rapidjson.Decoder attribute*), 29
uuid_mode (*rapidjson.Encoder attribute*), 26

V

ValidationError, 36
Validator (*class in rapidjson*), 32

W

WM_COMPACT (*in module rapidjson*), 37
WM_PRETTY (*in module rapidjson*), 37
WM_SINGLE_LINE_ARRAY (*in module rapidjson*), 37
write_mode (*rapidjson.Encoder attribute*), 26