simplejson Documentation

Release 3.16.1

Bob Ippolito

August 24, 2021

Contents

JSON (JavaScript Object Notation), specified by **RFC 7159** (which obsoletes **RFC 4627**) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript[1]).

simplejson exposes an API familiar to users of the standard library `marshal` and `pickle` modules. It is the externally maintained version of the `json` library contained in Python 2.6, but maintains compatibility with Python 2.5 and (currently) has significant performance advantages, even without using the optional C extension for speedups. *simplejson* is also supported on Python 3.3+.

Development of simplejson happens on Github: http://github.com/simplejson/simplejson

Encoding basic Python object hierarchies:

```
>>> import simplejson as json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps(u'\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from simplejson.compat import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

Compact encoding:

```
>>> import simplejson as json
>>> obj = [1,2,3,{'4': 5, '6': 7}]
>>> json.dumps(obj, separators=(',', ':'), sort_keys=True)
'[1,2,3,{"4":5,"6":7}]'
```

Pretty printing:

```
>>> import simplejson as json
>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4 * ' '))
{
    "4": 5,
    "6": 7
}
```

Decoding JSON:

```
>>> import simplejson as json
>>> obj = [u'foo', {u'bar': [u'baz', None, 1.0, 2]}]
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]') == obj
True
>>> json.loads('"\\"foo\\bar"') == u'"foo\x08ar'
True
>>> from simplejson.compat import StringIO
>>> io = StringIO('["streaming API"]')
```

(continues on next page)

[1] As noted in the errata for RFC 7159, JSON permits literal U+2028 (LINE SEPARATOR) and U+2029 (PARAGRAPH SEPARATOR) characters in strings, whereas JavaScript (as of ECMAScript Edition 5.1) does not.

(continued from previous page)

```
>>> json.load(io)[0] == 'streaming API'
True
```

Using Decimal instead of float:

```
>>> import simplejson as json
>>> from decimal import Decimal
>>> json.loads('1.1', use_decimal=True) == Decimal('1.1')
True
>>> json.dumps(Decimal('1.1'), use_decimal=True) == '1.1'
True
```

Specializing JSON object decoding:

```
>>> import simplejson as json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal) == decimal.Decimal('1.1')
True
```

Specializing JSON object encoding:

```
>>> import simplejson as json
>>> def encode_complex(obj):
...     if isinstance(obj, complex):
...         return [obj.real, obj.imag]
...     raise TypeError(repr(obj) + " is not JSON serializable")
...
>>> json.dumps(2 + 1j, default=encode_complex)
'[2.0, 1.0]'
>>> json.JSONEncoder(default=encode_complex).encode(2 + 1j)
'[2.0, 1.0]'
>>> ''.join(json.JSONEncoder(default=encode_complex).iterencode(2 + 1j))
'[2.0, 1.0]'
```

Using `simplejson.tool` from the shell to validate and pretty-print:

```
$ echo '{"json":"obj"}' | python -m simplejson.tool
{
    "json": "obj"
}
$ echo '{ 1.2:3.4}' | python -m simplejson.tool
Expecting property name enclosed in double quotes: line 1 column 3 (char 2)
```

Parsing multiple documents serialized as JSON lines (newline-delimited JSON):

```
>>> import simplejson as json
>>> def loads_lines(docs):
...     for doc in docs.splitlines():
```

(continues on next page)

(continued from previous page)

```
...         yield json.loads(doc)
...
>>> sum(doc["count"] for doc in loads_lines('{"count":1}\n{"count":2}\n{"count":3}\n
↪'))
6
```

Serializing multiple objects to JSON lines (newline-delimited JSON):

```
>>> import simplejson as json
>>> def dumps_lines(objs):
...     for obj in objs:
...         yield json.dumps(obj, separators=(',',':')) + '\n'
...
>>> ''.join(dumps_lines([{'count': 1}, {'count': 2}, {'count': 3}]))
'{"count":1}\n{"count":2}\n{"count":3}\n'
```

Note: JSON is a subset of YAML 1.2. The JSON produced by this module's default settings (in particular, the default *separators* value) is also a subset of YAML 1.0 and 1.1. This module can thus also be used as a YAML serializer.

CHAPTER 1

Basic Usage

`simplejson.dump`(*obj*, *fp*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*, *use_decimal=True*, *namedtuple_as_object=True*, *tuple_as_array=True*, *bigint_as_string=False*, *sort_keys=False*, *item_sort_key=None*, *for_json=None*, *ignore_nan=False*, *int_as_string_bitcount=None*, *iterable_as_array=False*, ***kw*)

Serialize *obj* as a JSON formatted stream to *fp* (a `.write()`-supporting file-like object) using this *conversion table*.

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

The *`simplejson`* module will produce `str` objects in Python 3, not `bytes` objects. Therefore, `fp.write()` must support `str` input.

If *ensure_ascii* is false (default: `True`), then some chunks written to *fp* may be `unicode` instances, subject to normal Python `str` to `unicode` coercion rules. Unless `fp.write()` explicitly understands `unicode` (as in `codecs.getwriter()`) this is likely to cause an error. It's best to leave the default settings, because they are safe and it is highly optimized.

If *check_circular* is false (default: `True`), then the circular reference check for container types will be skipped and a circular reference will result in an `OverflowError` (or worse).

If *allow_nan* is false (default: `True`), then it will be a `ValueError` to serialize out of range `float` values (`nan`, `inf`, `-inf`) in strict compliance of the original JSON specification. If *allow_nan* is true, their JavaScript equivalents will be used (`NaN`, `Infinity`, `-Infinity`). See also *ignore_nan* for ECMA-262 compliant behavior.

If *indent* is a string, then JSON array elements and object members will be pretty-printed with a newline followed by that string repeated for each level of nesting. `None` (the default) selects the most compact representation without any newlines. For backwards compatibility with versions of simplejson earlier than 2.1.0, an integer is also accepted and is converted to a string with that many spaces.

Changed in version 2.1.0: Changed *indent* from an integer number of spaces to a string.

If specified, *separators* should be an `(item_separator, key_separator)` tuple. The default is `(', ', ': ')` if *indent* is `None` and `(',', ': ')` otherwise. To get the most compact JSON representation, you should specify `(',', ':')` to eliminate whitespace.

Changed in version 2.1.4: Use `(',', ': ')` as default if *indent* is not `None`.

If *encoding* is not `None`, then all input `bytes` objects in Python 3 and 8-bit strings in Python 2 will be transformed into unicode using that encoding prior to JSON-encoding. The default is `'utf-8'`. If *encoding* is `None`, then all `bytes` objects will be passed to the *default* function in Python 3

Changed in version 3.15.0: `encoding=None` disables serializing `bytes` by default in Python 3.

default(obj) is a function that should return a serializable version of *obj* or raise `TypeError`. The default simply raises `TypeError`.

To use a custom *`JSONEncoder`* subclass (e.g. one that overrides the `default()` method to serialize additional types), specify it with the *cls* kwarg.

Note: Subclassing is not recommended. Use the *default* kwarg or *for_json* instead. This is faster and more portable.

If *use_decimal* is true (default: `True`) then `decimal.Decimal` will be natively serialized to JSON with full precision.

Changed in version 2.1.0: *use_decimal* is new in 2.1.0.

Changed in version 2.2.0: The default of *use_decimal* changed to `True` in 2.2.0.

If *namedtuple_as_object* is true (default: `True`), objects with `_asdict()` methods will be encoded as JSON objects.

Changed in version 2.2.0: *namedtuple_as_object* is new in 2.2.0.

Changed in version 2.3.0: *namedtuple_as_object* no longer requires that these objects be subclasses of `tuple`.

If *tuple_as_array* is true (default: `True`), `tuple` (and subclasses) will be encoded as JSON arrays.

If *iterable_as_array* is true (default: `False`), any object not in the above table that implements `__iter__()` will be encoded as a JSON array.

Changed in version 3.8.0: *iterable_as_array* is new in 3.8.0.

Changed in version 2.2.0: *tuple_as_array* is new in 2.2.0.

If *bigint_as_string* is true (default: `False`), `int 2**53` and higher or lower than `-2**53` will be encoded as strings. This is to avoid the rounding that happens in Javascript otherwise. Note that this option loses type information, so use with extreme caution. See also *int_as_string_bitcount*.

Changed in version 2.4.0: *bigint_as_string* is new in 2.4.0.

If *sort_keys* is true (not the default), then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

Changed in version 3.0.0: Sorting now happens after the keys have been coerced to strings, to avoid comparison of heterogeneously typed objects (since this does not work in Python 3.3+)

If *item_sort_key* is a callable (not the default), then the output of dictionaries will be sorted with it. The callable will be used like this: `sorted(dct.items(), key=item_sort_key)`. This option takes precedence over *sort_keys*.

Changed in version 2.5.0: *item_sort_key* is new in 2.5.0.

Changed in version 3.0.0: Sorting now happens after the keys have been coerced to strings, to avoid comparison of heterogeneously typed objects (since this does not work in Python 3.3+)

If *for_json* is true (not the default), objects with a `for_json()` method will use the return value of that method for encoding as JSON instead of the object.

Changed in version 3.2.0: *for_json* is new in 3.2.0.

If *ignore_nan* is true (default: `False`), then out of range `float` values (`nan`, `inf`, `-inf`) will be serialized as `null` in compliance with the ECMA-262 specification. If true, this will override *allow_nan*.

Changed in version 3.2.0: *ignore_nan* is new in 3.2.0.

If *int_as_string_bitcount* is a positive number `n` (default: `None`), `int 2**n` and higher or lower than `-2**n` will be encoded as strings. This is to avoid the rounding that happens in Javascript otherwise. Note that this option loses type information, so use with extreme caution. See also *bigint_as_string* (which is equivalent to *int_as_string_bitcount=53*).

Changed in version 3.5.0: *int_as_string_bitcount* is new in 3.5.0.

Note: JSON is not a framed protocol so unlike `pickle` or `marshal` it does not make sense to serialize more than one JSON document without some container protocol to delimit them.

`simplejson.`**`dumps`**(*obj*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*, *use_decimal=True*, *namedtuple_as_object=True*, *tuple_as_array=True*, *bigint_as_string=False*, *sort_keys=False*, *item_sort_key=None*, *for_json=None*, *ignore_nan=False*, *int_as_string_bitcount=None*, *iterable_as_array=False*, ***kw*)

Serialize *obj* to a JSON formatted `str`.

If *ensure_ascii* is false, then the return value will be a `unicode` instance. The other arguments have the same meaning as in `dump()`. Note that the default *ensure_ascii* setting has much better performance in Python 2.

The other options have the same meaning as in `dump()`.

`simplejson.`**`load`**(*fp*, *encoding='utf-8'*, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, *use_decimal=None*, ***kw*)

Deserialize *fp* (a `.read()`-supporting file-like object containing a JSON document) to a Python object using this *conversion table*. `JSONDecodeError` will be raised if the given JSON document is not valid.

If the contents of *fp* are encoded with an ASCII based encoding other than UTF-8 (e.g. latin-1), then an appropriate *encoding* name must be specified. Encodings that are not ASCII based (such as UCS-2) are not allowed, and should be wrapped with `codecs.getreader(fp)(encoding)`, or simply decoded to a `unicode` object and passed to `loads()`. The default setting of `'utf-8'` is fastest and should be using whenever possible.

If *fp.read()* returns `str` then decoded JSON strings that contain only ASCII characters may be parsed as `str` for performance and memory reasons. If your code expects only `unicode` the appropriate solution is to wrap fp with a reader as demonstrated above.

object_hook is an optional function that will be called with the result of any object literal decode (a `dict`). The return value of *object_hook* will be used instead of the `dict`. This feature can be used to implement custom decoders (e.g. JSON-RPC class hinting).

object_pairs_hook is an optional function that will be called with the result of any object literal decode with an ordered list of pairs. The return value of *object_pairs_hook* will be used instead of the `dict`. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, `collections.OrderedDict` will remember the order of insertion). If *object_hook* is also defined, the *object_pairs_hook* takes priority.

Changed in version 2.1.0: Added support for *object_pairs_hook*.

parse_float, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for JSON floats (e.g. `decimal.Decimal`).

parse_int, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to `int(num_str)`. This can be used to use another datatype or parser for JSON integers (e.g. `float`).

parse_constant, if specified, will be called with one of the following strings: `'-Infinity'`, `'Infinity'`, `'NaN'`. This can be used to raise an exception if invalid JSON numbers are encountered.

If *use_decimal* is true (default: `False`) then *parse_float* is set to `decimal.Decimal`. This is a convenience for parity with the `dump()` parameter.

Changed in version 2.1.0: *use_decimal* is new in 2.1.0.

If *iterable_as_array* is true (default: `False`), any object not in the above table that implements `__iter__()` will be encoded as a JSON array.

Changed in version 3.8.0: *iterable_as_array* is new in 3.8.0.

To use a custom `JSONDecoder` subclass, specify it with the `cls` kwarg. Additional keyword arguments will be passed to the constructor of the class. You probably shouldn't do this.

> **Note:** Subclassing is not recommended. You should use *object_hook* or *object_pairs_hook*. This is faster and more portable than subclassing.

> **Note:** `load()` will read the rest of the file-like object as a string and then call `loads()`. It does not stop at the end of the first valid JSON document it finds and it will raise an error if there is anything other than whitespace after the document. Except for files containing only one JSON document, it is recommended to use `loads()`.

`simplejson.`**`loads`**`(`*s*, *encoding='utf-8'*, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, *use_decimal=None*, ***kw*`)`

Deserialize *s* (a `str` or `unicode` instance containing a JSON document) to a Python object. `JSONDecodeError` will be raised if the given JSON document is not valid.

If *s* is a `str` instance and is encoded with an ASCII based encoding other than UTF-8 (e.g. latin-1), then an appropriate *encoding* name must be specified. Encodings that are not ASCII based (such as UCS-2) are not allowed and should be decoded to `unicode` first.

If *s* is a `str` then decoded JSON strings that contain only ASCII characters may be parsed as `str` for performance and memory reasons. If your code expects only `unicode` the appropriate solution is decode *s* to `unicode` prior to calling loads.

The other arguments have the same meaning as in `load()`.

CHAPTER 2

Encoders and decoders

class simplejson.**JSONDecoder**(*encoding='utf-8'*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, *strict=True*)

Simple JSON decoder.

Performs the following translations in decoding by default:

JSON	Python 2	Python 3
object	dict	dict
array	list	list
string	unicode	str
number (int)	int, long	int
number (real)	float	float
true	True	True
false	False	False
null	None	None

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

encoding determines the encoding used to interpret any `str` objects decoded by this instance (`'utf-8'` by default). It has no effect when decoding `unicode` objects.

Note that currently only encodings that are a superset of ASCII work, strings of other encodings should be passed in as `unicode`.

object_hook is an optional function that will be called with the result of every JSON object decoded and its return value will be used in place of the given `dict`. This can be used to provide custom deserializations (e.g. to support JSON-RPC class hinting).

object_pairs_hook is an optional function that will be called with the result of any object literal decode with an ordered list of pairs. The return value of *object_pairs_hook* will be used instead of the `dict`. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, `collections.OrderedDict` will remember the order of insertion). If *object_hook* is also defined, the *object_pairs_hook* takes priority.

Changed in version 2.1.0: Added support for *object_pairs_hook*.

parse_float, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for JSON floats (e.g. `decimal.Decimal`).

parse_int, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to `int(num_str)`. This can be used to use another datatype or parser for JSON integers (e.g. `float`).

parse_constant, if specified, will be called with one of the following strings: `'-Infinity'`, `'Infinity'`, `'NaN'`. This can be used to raise an exception if invalid JSON numbers are encountered.

strict controls the parser's behavior when it encounters an invalid control character in a string. The default setting of `True` means that unescaped control characters are parse errors, if `False` then control characters will be allowed in strings.

decode(*s*)

Return the Python representation of *s* (a `str` or `unicode` instance containing a JSON document)

If *s* is a `str` then decoded JSON strings that contain only ASCII characters may be parsed as `str` for performance and memory reasons. If your code expects only `unicode` the appropriate solution is decode *s* to `unicode` prior to calling decode.

JSONDecodeError will be raised if the given JSON document is not valid.

raw_decode(*s*[, *idx=0*])

Decode a JSON document from *s* (a `str` or `unicode` beginning with a JSON document) starting from the index *idx* and return a 2-tuple of the Python representation and the index in *s* where the document ended.

This can be used to decode a JSON document from a string that may have extraneous data at the end, or to decode a string that has a series of JSON objects.

JSONDecodeError will be raised if the given JSON document is not valid.

class simplejson.**JSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*, *use_decimal=True*, *namedtuple_as_object=True*, *tuple_as_array=True*, *bigint_as_string=False*, *item_sort_key=None*, *for_json=True*, *ignore_nan=False*, *int_as_string_bitcount=None*, *iterable_as_array=False*)

Extensible JSON encoder for Python data structures.

Supports the following objects and types by default:

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

Note: The JSON format only permits strings to be used as object keys, thus any Python dicts to be encoded should only have string keys. For backwards compatibility, several other types are automatically coerced to strings: int, long, float, Decimal, bool, and None. It is error-prone to rely on this behavior, so avoid it when possible. Dictionaries with other types used as keys should be pre-processed or wrapped in another type with an appropriate *for_json* method to transform the keys during encoding.

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

Changed in version 2.2.0: Changed *namedtuple* encoding from JSON array to object.

To extend this to recognize other objects, subclass and implement a `default()` method with another method that returns a serializable object for `o` if possible, otherwise it should call the superclass implementation (to raise `TypeError`).

Note: Subclassing is not recommended. You should use the *default* or *for_json* kwarg. This is faster and more portable than subclassing.

If *skipkeys* is false (the default), then it is a `TypeError` to attempt encoding of keys that are not str, int, long, float, Decimal, bool, or None. If *skipkeys* is true, such items are simply skipped.

If *ensure_ascii* is true (the default), the output is guaranteed to be `str` objects with all incoming unicode characters escaped. If *ensure_ascii* is false, the output will be a unicode object.

If *check_circular* is true (the default), then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an `OverflowError`). Otherwise, no such check takes place.

If *allow_nan* is true (the default), then `NaN`, `Infinity`, and `-Infinity` will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a `ValueError` to encode such floats. See also *ignore_nan* for ECMA-262 compliant behavior.

If *sort_keys* is true (not the default), then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

Changed in version 3.0.0: Sorting now happens after the keys have been coerced to strings, to avoid comparison of heterogeneously typed objects (since this does not work in Python 3.3+)

If *item_sort_key* is a callable (not the default), then the output of dictionaries will be sorted with it. The callable will be used like this: `sorted(dct.items(), key=item_sort_key)`. This option takes precedence over *sort_keys*.

Changed in version 2.5.0: *item_sort_key* is new in 2.5.0.

Changed in version 3.0.0: Sorting now happens after the keys have been coerced to strings, to avoid comparison of heterogeneously typed objects (since this does not work in Python 3.3+)

If *indent* is a string, then JSON array elements and object members will be pretty-printed with a newline followed by that string repeated for each level of nesting. `None` (the default) selects the most compact representation without any newlines. For backwards compatibility with versions of simplejson earlier than 2.1.0, an integer is also accepted and is converted to a string with that many spaces.

Changed in version 2.1.0: Changed *indent* from an integer number of spaces to a string.

If specified, *separators* should be an `(item_separator, key_separator)` tuple. The default is `(', ', ': ')` if *indent* is `None` and `(',', ': ')` otherwise. To get the most compact JSON representation, you should specify `(',', ':')` to eliminate whitespace.

Changed in version 2.1.4: Use `(',', ': ')` as default if *indent* is not `None`.

If specified, *default* should be a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

If *encoding* is not `None`, then all input `bytes` objects in Python 3 and 8-bit strings in Python 2 will be transformed into unicode using that encoding prior to JSON-encoding. The default is `'utf-8'`. If *encoding* is `None`, then all `bytes` objects will be passed to the `default()` method in Python 3

Changed in version 3.15.0: `encoding=None` disables serializing `bytes` by default in Python 3.

If *namedtuple_as_object* is true (default: `True`), objects with `_asdict()` methods will be encoded as JSON objects.

Changed in version 2.2.0: *namedtuple_as_object* is new in 2.2.0.

Changed in version 2.3.0: *namedtuple_as_object* no longer requires that these objects be subclasses of `tuple`.

If *tuple_as_array* is true (default: `True`), `tuple` (and subclasses) will be encoded as JSON arrays.

Changed in version 2.2.0: *tuple_as_array* is new in 2.2.0.

If *iterable_as_array* is true (default: `False`), any object not in the above table that implements `__iter__()` will be encoded as a JSON array.

Changed in version 3.8.0: *iterable_as_array* is new in 3.8.0.

If *bigint_as_string* is true (default: `False`), int` 2**53 and higher or lower than `-2**53` will be encoded as strings. This is to avoid the rounding that happens in Javascript otherwise. Note that this option loses type information, so use with extreme caution.

Changed in version 2.4.0: *bigint_as_string* is new in 2.4.0.

If *for_json* is true (default: `False`), objects with a `for_json()` method will use the return value of that method for encoding as JSON instead of the object.

Changed in version 3.2.0: *for_json* is new in 3.2.0.

If *ignore_nan* is true (default: `False`), then out of range `float` values (`nan`, `inf`, `-inf`) will be serialized as `null` in compliance with the ECMA-262 specification. If true, this will override *allow_nan*.

Changed in version 3.2.0: *ignore_nan* is new in 3.2.0.

`default`(*o*)

Implement this method in a subclass such that it returns a serializable object for *o*, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```python
def default(self, o):
   try:
       iterable = iter(o)
   except TypeError:
       pass
   else:
       return list(iterable)
   return JSONEncoder.default(self, o)
```

Note: Subclassing is not recommended. You should implement this as a function and pass it to the *default* kwarg of `dumps()`. This is faster and more portable than subclassing. The semantics are the same, but without the self argument or the call to the super implementation.

encode(*o*)

Return a JSON string representation of a Python data structure, *o*. For example:

```
>>> import simplejson as json
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)

Encode the given object, *o*, and yield each string representation as available. For example:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

Note that `encode()` has much better performance than `iterencode()`.

class simplejson.**JSONEncoderForHTML**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*, *use_decimal=True*, *namedtuple_as_object=True*, *tuple_as_array=True*, *bigint_as_string=False*, *item_sort_key=None*, *for_json=True*, *ignore_nan=False*, *int_as_string_bitcount=None*)

Subclass of `JSONEncoder` that escapes &, <, and > for embedding in HTML.

It also escapes the characters U+2028 (LINE SEPARATOR) and U+2029 (PARAGRAPH SEPARATOR), irrespective of the *ensure_ascii* setting, as these characters are not valid in JavaScript strings (see http://timelessrepo.com/json-isnt-a-javascript-subset).

Changed in version 2.1.0: New in 2.1.0

CHAPTER 3

Exceptions

exception simplejson.**JSONDecodeError**(*msg*, *doc*, *pos*, *end=None*)

Subclass of `ValueError` with the following additional attributes:

msg

The unformatted error message

doc

The JSON document being parsed

pos

The start index of doc where parsing failed

end

The end index of doc where parsing failed (may be `None`)

lineno

The line corresponding to pos

colno

The column corresponding to pos

endlineno

The line corresponding to end (may be `None`)

endcolno

The column corresponding to end (may be `None`)

CHAPTER 4

Standard Compliance and Interoperability

The JSON format is specified by **RFC 7159** and by ECMA-404. This section details this module's level of compliance with the RFC. For simplicity, `JSONEncoder` and `JSONDecoder` subclasses, and parameters other than those explicitly mentioned, are not considered.

This module does not comply with the RFC in a strict fashion, implementing some extensions that are valid JavaScript but not valid JSON. In particular:

- Infinite and NaN number values are accepted and output;
- Repeated names within an object are accepted, and only the value of the last name-value pair is used.

Since the RFC permits RFC-compliant parsers to accept input texts that are not RFC-compliant, this module's deserializer is technically RFC-compliant under default settings.

4.1 Character Encodings

The RFC recommends that JSON be represented using either UTF-8, UTF-16, or UTF-32, with UTF-8 being the recommended default for maximum interoperability.

As permitted, though not required, by the RFC, this module's serializer sets *ensure_ascii=True* by default, thus escaping the output so that the resulting strings only contain ASCII characters.

Other than the *ensure_ascii* parameter, this module is defined strictly in terms of conversion between Python objects and `Unicode strings`, and thus does not otherwise directly address the issue of character encodings.

The RFC prohibits adding a byte order mark (BOM) to the start of a JSON text, and this module's serializer does not add a BOM to its output. The RFC permits, but does not require, JSON deserializers to ignore an initial BOM in their input. This module's deserializer will ignore an initial BOM, if present.

Changed in version 3.6.0: Older versions would raise `ValueError` when an initial BOM is present

The RFC does not explicitly forbid JSON strings which contain byte sequences that don't correspond to valid Unicode characters (e.g. unpaired UTF-16 surrogates), but it does note that they may cause interoperability problems. By default, this module accepts and outputs (when present in the original `str`) codepoints for such sequences.

4.2 Infinite and NaN Number Values

The RFC does not permit the representation of infinite or NaN number values. Despite that, by default, this module accepts and outputs `Infinity`, `-Infinity`, and `NaN` as if they were valid JSON number literal values:

```
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

In the serializer, the *allow_nan* parameter can be used to alter this behavior. In the deserializer, the *parse_constant* parameter can be used to alter this behavior.

4.3 Repeated Names Within an Object

The RFC specifies that the names within a JSON object should be unique, but does not mandate how repeated names in JSON objects should be handled. By default, this module does not raise an exception; instead, it ignores all but the last name-value pair for a given name:

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json) == {'x': 3}
True
```

The *object_pairs_hook* parameter can be used to alter this behavior.

4.4 Top-level Non-Object, Non-Array Values

The old version of JSON specified by the obsolete **RFC 4627** required that the top-level value of a JSON text must be either a JSON object or array (Python `dict` or `list`), and could not be a JSON null, boolean, number, or string value. **RFC 7159** removed that restriction, and this module does not and has never implemented that restriction in either its serializer or its deserializer.

Regardless, for maximum interoperability, you may wish to voluntarily adhere to the restriction yourself.

4.5 Implementation Limitations

Some JSON deserializer implementations may set limits on:

- the size of accepted JSON texts
- the maximum level of nesting of JSON objects and arrays
- the range and precision of JSON numbers
- the content and maximum length of JSON strings

This module does not impose any such limits beyond those of the relevant Python datatypes themselves or the Python interpreter itself.

When serializing to JSON, beware any such limitations in applications that may consume your JSON. In particular, it is common for JSON numbers to be deserialized into IEEE 754 double precision numbers and thus subject to that representation's range and precision limitations. This is especially relevant when serializing Python `int` values of extremely large magnitude, or when serializing instances of "exotic" numerical types such as `decimal.Decimal`.

CHAPTER 5

Command Line Interface

The `simplejson.tool` module provides a simple command line interface to validate and pretty-print JSON.

If the optional *`infile`* and *`outfile`* arguments are not specified, `sys.stdin` and `sys.stdout` will be used respectively:

```
$ echo '{"json": "obj"}' | python -m simplejson.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m simplejson.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

5.1 Command line options

`infile`

The JSON file to be validated or pretty-printed:

```
$ python -m simplejson.tool mp_films.json
[
    {
        "title": "And Now for Something Completely Different",
        "year": 1971
    },
    {
        "title": "Monty Python and the Holy Grail",
        "year": 1975
    }
]
```

If *infile* is not specified, read from `sys.stdin`.

`outfile`

Write the output of the *infile* to the given *outfile*. Otherwise, write it to `sys.stdout`.

Python Module Index

Index