Table Of Contents

Previous topic

18.1.11. email: Examples

Next topic

18.3. mailcap — Mailcap file handling

This Page

18.2. json — JSON encoder and decoder

New in version 2.6.

JSON (JavaScript Object Notation), specified by **RFC 4627**, is a lightweight data interchange format based on a subset of JavaScript syntax (ECMA-262 3rd edition).

json exposes an API familiar to users of the standard library marshal and pickle modules.

Encoding basic Python object hierarchies:

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

Report a Bug
Show Source

Quick search

Go

Enter search terms or a module, class or function name.

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

Compact encoding:

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

Pretty printing:

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

Decoding JSON:

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

Specializing JSON object decoding:

```
>>> import 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('1.1')
```

Extending `JSONEncoder`:

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return json.JSONEncoder.default(self, obj)
...
>>> dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[', '2.0', ', ', '1.0', ']']
```

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

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

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.

18.2.1. Basic Usage

`json.`**`dump`**(*obj*, *fp*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *encoding="utf-8"*, *default=None*, *sort_keys=False*, ***kw*)

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`**.

If *ensure_ascii* is **`True`** (the default), all non-ASCII characters in the output are escaped with `\uXXXX` sequences, and the result is a **`str`** instance consisting of ASCII characters only. If *ensure_ascii* is **`False`**, some chunks written to *fp* may be **`unicode`** instances. This usually happens because the input contains unicode strings or the *encoding* parameter is used. Unless `fp.write()` explicitly understands `unicode` (as in `codecs.getwriter()`) this is likely to cause an error.

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 JSON specification, instead of using the JavaScript equivalents (`NaN`, `Infinity`, `-Infinity`).

If *indent* is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, or negative, will only insert newlines. `None` (the default) selects the most compact representation.

> **Note:** Since the default item separator is `', '`, the output might include trailing whitespace when *indent* is specified. You can use `separators=(',', ': ')` to avoid this.

If *separators* is an `(item_separator, dict_separator)` tuple, then it will be used instead of the default `(', ', ': ')` separators. `(',', ':')` is the most compact JSON representation.

encoding is the character encoding for str instances, default is UTF-8.

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

If *sort_keys* is **`True`** (default: **`False`**), then the output of dictionaries will be sorted by key.

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

> **Note:** Unlike `pickle` and `marshal`, JSON is not a framed protocol so trying to serialize more objects with repeated calls to `dump()` and the same *fp* will result in an invalid JSON file.

`json.`**`dumps`**(*obj*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *encoding="utf-8"*, *default=None*, *sort_keys=False*, ***kw*)

Serialize *obj* to a JSON formatted `str` using this *conversion table*. If *ensure_ascii* is **`False`**, the result may contain non-ASCII characters and the return value may be a `unicode` instance.

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

> **Note:** Keys in key/value pairs of JSON are always of the type `str`. When a dictionary is converted into JSON, all the keys of the dictionary are coerced to strings. As a result of this, if a dictionary is converted into JSON and then back into a dictionary, the dictionary may not equal the

original one. That is, `loads(dumps(x)) != x` if x has non-string keys.

`json.`**`load`**(*fp*[, *encoding*[, *cls*[, *object_hook*[, *parse_float*[, *parse_int*[, *parse_constant*[, *object_pairs_hook*[, ***kw*]]]]]]]])

Deserialize *fp* (a `.read()`-supporting *file-like object* containing a JSON document) to a Python object using this *conversion table*.

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(encoding)(fp)`, or simply decoded to a **`unicode`** object and passed to **`loads()`**.

object_hook is an optional function that will be called with the result of any object literal decoded (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 decoded 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 2.7: 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.

Changed in version 2.7: parse_constant doesn't get called on 'null', 'true', 'false' anymore.

To use a custom **`JSONDecoder`** subclass, specify it with the `cls` kwarg; otherwise **`JSONDecoder`** is used. Additional keyword arguments will be passed to the constructor of the class.

`json.`**`loads`**(*s*[, *encoding*[, *cls*[, *object_hook*[, *parse_float*[, *parse_int*[, *parse_constant*[, *object_pairs_hook*[, ***kw*]]]]]]]])

Deserialize *s* (a **`str`** or **`unicode`** instance containing a JSON document) to a Python object using this *conversion table*.

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.

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

18.2.2. Encoders and Decoders

class `json.`**`JSONDecoder`**([*encoding*[, *object_hook*[, *parse_float*[, *parse_int*[, *parse_constant*[, *strict*[, *object_pairs_hook*]]]]]]])

Simple JSON decoder.

Performs the following translations in decoding by default:

JSON	Python
object	dict
array	list
string	unicode
number (int)	int, long
number (real)	float
true	True
false	False
null	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, if specified, 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, if specified will be called with the result of every JSON object decoded 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 2.7: 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'`, `'null'`, `'true'`, `'false'`. This can be used to raise an exception if invalid JSON numbers are encountered.

If *strict* is **`False`** (**`True`** is the default), then control characters will be allowed inside strings. Control characters in this context are those with character codes in the 0-31 range, including `'\t'` (tab), `'\n'`, `'\r'` and `'\0'`.

If the data being deserialized is not a valid JSON document, a `ValueError` will be raised.

`decode`(*s*)

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

`raw_decode`(*s*)

Decode a JSON document from *s* (a `str` or `unicode` beginning with a JSON document) 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.

class `json.`**`JSONEncoder`**([*skipkeys*[, *ensure_ascii*[, *check_circular*[, *allow_nan*[, *sort_keys*[, *indent*[, *separators*[, *encoding*[, *default*]]]]]]]]])

Extensible JSON encoder for Python data structures.

Supports the following objects and types by default:

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

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`).

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

If *ensure_ascii* is **`True`** (the default), all non-ASCII characters in the output are escaped with `\uXXXX` sequences, and the results are `str` instances consisting of ASCII characters only. If *ensure_ascii* is **`False`**, a result may be a `unicode` instance. This usually happens if the input contains unicode strings or the *encoding* parameter is used.

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.

If *sort_keys* is **`True`** (default **`False`**), 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.

If *indent* is a non-negative integer (it is **`None`** by default), then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. **`None`** is the most compact representation.

> **Note:** Since the default item separator is `', '`, the output might include trailing whitespace when *indent* is specified. You can use `separators=(',', ': ')` to avoid this.

If specified, *separators* should be an `(item_separator, key_separator)` tuple. The default is `(', ', ': ')`. To get the most compact JSON representation, you should specify `(',', ':')` to eliminate whitespace.

If specified, *default* is 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 strings will be transformed into unicode using that encoding prior to JSON-encoding. The default is UTF-8.

`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:

```
def default(self, o):
   try:
       iterable = iter(o)
   except TypeError:
       pass
   else:
       return list(iterable)
   # Let the base class default method raise the TypeError
   return JSONEncoder.default(self, o)
```

`encode`(*o*)

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

```
>>> 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)
```

18.2.3. Standard Compliance

The JSON format is specified by **RFC 4627**. 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:

- Top-level non-object, non-array values are accepted and output;
- 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.

18.2.3.1. Character Encodings

The RFC recommends that JSON be represented using either UTF-8, UTF-16, or UTF-32, with UTF-8 being the default. Accordingly, this module uses UTF-8 as the default for its *encoding* parameter.

This module’s deserializer only directly works with ASCII-compatible encodings; UTF-16, UTF-32, and other ASCII-incompatible encodings require the use of workarounds described in the documentation for the deserializer’s *encoding* parameter.

The RFC also non-normatively describes a limited encoding detection technique for JSON texts; this module’s deserializer does not implement this or any other kind of encoding detection.

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.

18.2.3.2. Top-level Non-Object, Non-Array Values

The RFC specifies that the top-level value of a JSON text must be either a JSON object or array (Python `dict` or `list`). This module’s deserializer also accepts input texts consisting solely of a JSON null, boolean, number, or string value:

```
>>> just_a_json_string = '"spam and eggs"'  # Not by itself a valid
>>> json.loads(just_a_json_string)
```

```
u'spam and eggs'
```

This module itself does not include a way to request that such input texts be regarded as illegal. Likewise, this module's serializer also accepts single Python **`None`**, **`bool`**, numeric, and **`str`** values as input and will generate output texts consisting solely of a top-level JSON null, boolean, number, or string value without raising an exception:

```
>>> neither_a_list_nor_a_dict = u"spam and eggs"
>>> json.dumps(neither_a_list_nor_a_dict)  # The result is not a val
'"spam and eggs"'
```

This module's serializer does not itself include a way to enforce the aforementioned constraint.

18.2.3.3. 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 a
>>> 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.

18.2.3.4. Repeated Names Within an Object

The RFC specifies that the names within a JSON object should be unique, but does not specify 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)
{u'x': 3}
```

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

Python » 2.7.6 Documentation » The Python Standard Library » 18. Internet Data Handling » previous | next | modules | index