19.2. `json` — JSON encoder and decoder¶

New in version 2.6.

JSON (JavaScript Object Notation) <http://json.org> is a subset of JavaScript syntax (ECMA-262 3rd edition) used as a lightweight data interchange format.

`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()
>>> 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)
{
   "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]
...         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: line 1 column 2 (char 2)
```

Note

The JSON produced by this module's default settings is a subset of YAML, so it may be used as a serializer for that as well.

19.2.1. Basic Usage¶

`json.dump`(*obj*, *fp*[, *skipkeys*[, *ensure_ascii*[, *check_circular*[, *allow_nan*[, *cls*[, *indent*[, *separators*[, *encoding*[, *default*[, ***kw*]]]]]]]]])¶

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

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 `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.

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 will only insert newlines. `None` (the default) selects the most compact representation.

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`.

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

`json.dumps`(*obj*[, *skipkeys*[, *ensure_ascii*[, *check_circular*[, *allow_nan*[, *cls*[, *indent*[, *separators*[, *encoding*[, *default*[, ***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()`.

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

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

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).

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.

To use a custom `JSONDecoder` subclass, specify it with the `cls` kwarg. 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*[, ***kw*]]]]]]])¶

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

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 `dump()`.

19.2.2. Encoders and decoders¶

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

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).

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.

`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), 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.

If *sort_keys* is `True` (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.

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.

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)
  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)
```


Table Of Contents

19.2. `json` — JSON encoder and decoder

- 19.2.1. Basic Usage
- 19.2.2. Encoders and decoders

Previous topic

19.1.11. `email`: Examples

Next topic

19.3. `mailcap` — Mailcap file handling