json — JSON encoder and decoder

Source code: Lib/json/__init__.py

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]).

Warning: Be cautious when parsing JSON data from untrusted sources. A malicious JSON string may cause the decoder to consume considerable CPU and memory resources. Limiting the size of data to be parsed is recommended.

`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('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io 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({'6': 7, '4': 5}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

Specializing JSON object encoding:

```
>>> import json
>>> def custom_json(obj):
```

```
...     if isinstance(obj, complex):
...         return {'__complex__': True, 'real': obj.real, 'imag': obj.imag}
...     raise TypeError(f'Cannot serialize object of {type(obj)}')
...
>>> json.dumps(1 + 2j, default=custom_json)
'{"__complex__": true, "real": 1.0, "imag": 2.0}'
```

Decoding JSON:

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['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 super().default(obj)
...
>>> json.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 -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

See Command Line Interface for detailed documentation.

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.

Note: This module's encoders and decoders preserve input and output order by default. Order is only lost if the underlying containers are unordered.

Basic Usage

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

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

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

Parameters:

- **obj** (*object*) – The Python object to be serialized.
- **fp** (file-like object) – The file-like object *obj* will be serialized to. The `json` module always produces `str` objects, not `bytes` objects, therefore `fp.write()` must support `str` input.
- **skipkeys** (*bool*) – If `True`, keys that are not of a basic type (`str`, `int`, `float`, `bool`, `None`) will be skipped instead of raising a `TypeError`. Default `False`.
- **ensure_ascii** (*bool*) – If `True` (the default), the output is guaranteed to have all incoming non-ASCII characters escaped. If `False`, these characters will be outputted as-is.
- **check_circular** (*bool*) – If `False`, the circular reference check for container types is skipped and a circular reference will result in a `RecursionError` (or worse). Default `True`.
- **allow_nan** (*bool*) – If `False`, serialization of out-of-range `float` values (`nan`, `inf`, `-inf`) will result in a `ValueError`, in strict compliance with the JSON specification. If `True` (the default), their JavaScript equivalents (`NaN`, `Infinity`, `-Infinity`) are used.
- **cls** (a `JSONEncoder` subclass) – If set, a custom JSON encoder with the `default()` method overridden, for serializing into custom datatypes. If `None` (the default), `JSONEncoder` is used.
- **indent** (*int* | *str* | *None*) – If a positive integer or string, JSON array elements and object members will be pretty-printed with that indent level. A positive integer indents that many spaces per level; a string (such as `"\t"`) is used to indent each level. If zero, negative, or `""` (the empty string), only newlines are inserted. If `None` (the default), the most compact representation is used.
- **separators** (*tuple* | *None*) – A two-tuple: `(item_separator, key_separator)`. If `None` (the default), *separators* defaults to `(', ', ': ')` if *indent* is `None`, and `(',', ': ')` otherwise. For the most compact JSON, specify `(',', ':')` to eliminate whitespace.

输出将保证对所有输入的非ASCII字符进行转义

- **default** (callable | None) – A function that is called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`. If `None` (the default), `TypeError` is raised.
- **sort_keys** (*bool*) – If `True`, dictionaries will be outputted sorted by key. Default `False`.

> *Changed in version 3.2:* Allow strings for *indent* in addition to integers.

> *Changed in version 3.4:* Use `(',', ': ')` as default if *indent* is not `None`.

> *Changed in version 3.6:* All optional parameters are now keyword-only.

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

Serialize *obj* to a JSON formatted `str` using this conversion table. 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, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)`

Deserialize *fp* to a Python object using the JSON-to-Python conversion table.

Parameters:

- **fp** (file-like object) – A `.read()`-supporting text file or binary file containing the JSON document to be deserialized.
- **cls** (a `JSONDecoder` subclass) – If set, a custom JSON decoder. Additional keyword arguments to `load()` will be passed to the constructor of *cls*. If `None` (the default), `JSONDecoder` is used.
- **object_hook** (callable | None) – If set, a function that is called with the result of any object literal decoded (a `dict`). The return value of this function will be used instead of the `dict`. This feature can be used to implement custom decoders, for example JSON-RPC class hinting. Default `None`.
- **object_pairs_hook** (callable | None) – If set, a function that is called with the result of any object literal decoded with an ordered list of pairs. The return value of this function will be used instead of the `dict`. This feature can be used to implement custom decoders. If *object_hook* is also set, *object_pairs_hook* takes priority. Default `None`.
- **parse_float** (callable | None) – If set, a function that is called with the string of every JSON float to be decoded. If `None` (the default), it is equivalent to `float(num_str)`. This can be used to parse JSON floats into custom datatypes, for example `decimal.Decimal`.
- **parse_int** (callable | None) – If set, a function that is called with the string of every JSON int to be decoded. If `None` (the default), it is equivalent to `int(num_str)`. This can be used to parse JSON integers into custom datatypes, for example `float`.
- **parse_constant** (callable | None) – If set, a function that is called with one of the following strings: `'-Infinity'`, `'Infinity'`, or `'NaN'`. This can be used to raise an exception

if invalid JSON numbers are encountered. Default `None`.

Raises:
- **JSONDecodeError** – When the data being deserialized is not a valid JSON document.
- **UnicodeDecodeError** – When the data being deserialized does not contain UTF-8, UTF-16 or UTF-32 encoded data.

> *Changed in version 3.1:*
>
> - Added the optional *object_pairs_hook* parameter.
> - *parse_constant* doesn't get called on 'null', 'true', 'false' anymore.

> *Changed in version 3.6:*
>
> - All optional parameters are now keyword-only.
> - *fp* can now be a binary file. The input encoding should be UTF-8, UTF-16 or UTF-32.

> *Changed in version 3.11:* The default *parse_int* of `int()` now limits the maximum length of the integer string via the interpreter's integer string conversion length limitation to help avoid denial of service attacks.

`json.loads(s, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)`

Identical to `load()`, but instead of a file-like object, deserialize *s* (a `str`, `bytes` or `bytearray` instance containing a JSON document) to a Python object using this conversion table.

> *Changed in version 3.6:* *s* can now be of type `bytes` or `bytearray`. The input encoding should be UTF-8, UTF-16 or UTF-32.

> *Changed in version 3.9:* The keyword argument *encoding* has been removed.

Encoders and Decoders

`class json.JSONDecoder(*, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, strict=True, object_pairs_hook=None)`

Simple JSON decoder.

Performs the following translations in decoding by default:

JSON	Python
object	dict
array	list
string	str
number (int)	int
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.

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 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. If *object_hook* is also defined, the *object_pairs_hook* takes priority.

> *Changed in version 3.1:* Added support for *object_pairs_hook*.

parse_float is an optional function that 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 is an optional function that 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 is an optional function that 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 *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 `JSONDecodeError` will be raised.

> *Changed in version 3.6:* All parameters are now keyword-only.

decode(*s*)

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

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

raw_decode(*s*)

Decode a JSON document from *s* (a `str` 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=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Extensible JSON encoder for Python data structures.

Supports the following objects and types by default:

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
True	true
False	false
None	null

> *Changed in version 3.4:* Added support for int- and float-derived Enum classes.

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), a `TypeError` will be raised when trying to encode keys that are not `str`, `int`, `float` or `None`. If *skipkeys* is true, such items are simply skipped.

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.

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 a `RecursionError`). 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 or string, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, negative, or `""` will only insert newlines. `None` (the default) selects the most compact representation. Using a positive integer indent indents that many spaces per level. If *indent* is a string (such as `"\t"`), that string is used to indent each level.

> *Changed in version 3.2:* Allow strings for *indent* in addition to integers.

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 3.4:* 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 not specified, `TypeError` is raised.

> *Changed in version 3.6:* All parameters are now keyword-only.

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 super().default(o)
```

encode(*o*)

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

```
>>> 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 json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

Exceptions

exception json.**JSONDecodeError**(*msg*, *doc*, *pos*)

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.

lineno

The line corresponding to *pos*.

colno

The column corresponding to *pos*.

Added in version 3.5.

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.

Character Encodings

The RFC requires 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 raises a `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`) code points for such sequences.

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.

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}
```

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

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.

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`.

Command Line Interface

Source code: Lib/json/tool.py

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

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

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

> *Changed in version 3.5:* The output is now in the same order as the input. Use the `--sort-keys` option to sort the output of dictionaries alphabetically by key.

Command line options

`infile`

The JSON file to be validated or pretty-printed:

```
$ python -m json.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`.

`--sort-keys`

Sort the output of dictionaries alphabetically by key.

> *Added in version 3.5.*

`--no-ensure-ascii`

Disable escaping of non-ascii characters, see `json.dumps()` for more information.

> *Added in version 3.9.*

`--json-lines`

Parse every input line as separate JSON object.

> *Added in version 3.8.*

`--indent, --tab, --no-indent, --compact`

Mutually exclusive options for whitespace control.

> *Added in version 3.9.*

`-h, --help`

Show the help message.

Footnotes

[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.