index
/usr/lib/python2.7/json/__init__.py
Module Docs

json (version 2.0.9)

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

:mod:`json` exposes an API familiar to users of the standard library
:mod:`marshal` and :mod:`pickle` modules. It is the externally maintained
version of the :mod:`json` library contained in Python 2.6, but maintains
compatibility with Python 2.4 and Python 2.5 and (currently) has
significant performance advantages, even without using the optional C
extension for speedups.

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}], sort_keys=True, 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
    >>> 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 StringIO import StringIO
    >>> io = StringIO('["streaming API"]')
    >>> json.load(io)[0] == 'streaming API'
    True

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)
    >>> from decimal import Decimal
    >>> json.loads('1.1', parse_float=Decimal) == Decimal('1.1')
    True

Specializing JSON object encoding::

    >>> import json
    >>> def encode_complex(obj):
    ...     if isinstance(obj, complex):
    ...         return [obj.real, obj.imag]
    ...     raise TypeError(repr(o) + " 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 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 3 (char 2)
```

Package Contents

decoder encoder scanner tool

Classes

__builtin__.object
    json.decoder.JSONDecoder

json.encoder.JSONEncoder

class **JSONDecoder**(__builtin__.object)

```
Simple JSON <http://json.org> 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.
```

Methods defined here:

__init__(self, encoding=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, strict=True, object_pairs_hook=None)

```
    ``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.
```

``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 ``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'``.

decode(self, s, _w=<built-in method match of _sre.SRE_Pattern object>)

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

raw_decode(self, s, idx=0)

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.

Data descriptors defined here:

__dict__

dictionary for instance variables (if defined)

__weakref__

list of weak references to the object (if defined)

class **JSONEncoder**(__builtin__.object)

Extensible JSON <http://json.org> 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``).
```

Methods defined here:

__init__(self, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, encoding='utf-8', default=None)

```
    Constructor for JSONEncoder, with sensible defaults.

    If skipkeys is false, 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, 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, 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, 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, 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.  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 a (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(self, 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(self, o)

```
    Return a JSON string representation of a Python data structure.

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

iterencode(self, o, _one_shot=False)

```
    Encode the given object and yield each string
    representation as available.

    For example::

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

Data descriptors defined here:

__dict__

```
    dictionary for instance variables (if defined)
```

__weakref__

```
    list of weak references to the object (if defined)
```

Data and other attributes defined here:

item_separator = ', '

key_separator = ': '

Functions

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).

If ``skipkeys`` is true then ``dict`` keys that are not basic types
(``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, 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, 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`` is the most compact
representation.  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.

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``.

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

If ``ensure_ascii`` is false, all non-ASCII characters are not escaped, and the return value may be a ``unicode`` instance. See ``dump`` for details.

If ``check_circular`` is false, 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, 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`` is the most compact representation. 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.

load(fp, encoding=None, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)

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

If the contents of ``fp`` 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 wrapped with

``codecs.getreader(fp)(encoding)``, 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 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 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.

To use a custom ``JSONDecoder`` subclass, specify it with the ``cls``
kwarg; otherwise ``JSONDecoder`` is used.

loads(s, encoding=None, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **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.

``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 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.

``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; otherwise ``JSONDecoder`` is used.
```

Data

__all__ = ['dump', 'dumps', 'load', 'loads', 'JSONDecoder', 'JSONEncoder']
__author__ = 'Bob Ippolito <bob@redivi.com>'
__version__ = '2.0.9'

Author

Bob Ippolito <bob@redivi.com>