

---

# PyJSON5 Documentation

*Release 1.5.2*

**René Kijewski**

**Jul 07, 2021**



# CONTENTS

<b>1</b>	<b>Installation</b>	<b>3</b>
<b>2</b>	<b>Table of Contents</b>	<b>5</b>
2.1	Serializer / Encoder . . . . .	5
2.1.1	Quick Encoder Summary . . . . .	5
2.1.2	Full Encoder Description . . . . .	6
2.1.3	Encoder Compatibility Functions . . . . .	9
2.1.4	Encoder Exceptions . . . . .	9
2.2	Parser / Decoder . . . . .	10
2.2.1	Quick Decoder Summary . . . . .	10
2.2.2	Full Decoder Description . . . . .	10
2.2.3	Decoder Compatibility Functions . . . . .	12
2.2.4	Decoder Exceptions . . . . .	13
2.3	Exceptions . . . . .	15
2.4	Performance . . . . .	15
2.4.1	Decoder Performance . . . . .	15
2.4.2	Encoder Performance . . . . .	16
2.4.3	Benchmark . . . . .	16
<b>3</b>	<b>Quick Summary</b>	<b>17</b>
<b>4</b>	<b>Compatibility</b>	<b>19</b>
	<b>Index</b>	<b>21</b>



A JSON5 serializer and parser library for Python 3.5 and later.

The serializer returns ASCII data that can safely be used in an HTML template. Apostrophes, ampersands, greater-than, and less-than signs are encoded as unicode escaped sequences. E.g. this snippet is safe for any and all input:

```
"<a onclick='alert(" + encode(data) + ")'>show message</a>"
```

Unless the input contains infinite or NaN values, the result will be valid **JSON** data.

All valid **JSON5 1.0.0** and **JSON** data can be read, unless the nesting level is absurdly high.



## INSTALLATION

```
$ pip install pyjson5
```





## TABLE OF CONTENTS

### 2.1 Serializer / Encoder

The serializer returns ASCII data that can safely be used in an HTML template. Apostrophes, ampersands, greater-than, and less-than signs are encoded as unicode escaped sequences. E.g. this snippet is safe for any and all input:

```
"<a onclick='alert (" + encode(data) + ")'>show message</a>"
```

Unless the input contains infinite or NaN values, the result will be valid [JSON](#) data.

#### 2.1.1 Quick Encoder Summary

<i>encode</i> (data, *[, options])	Serializes a Python object to a JSON5 compatible unicode string.
<i>encode_bytes</i> (data, *[, options])	Serializes a Python object to a JSON5 compatible bytes string.
<i>encode_callback</i> (data, cb[, supply_bytes, ...])	Serializes a Python object into a callback function.
<i>encode_io</i> (data, fp[, supply_bytes, options])	Serializes a Python object into a file-object.
<i>encode_noop</i> (data, *[, options])	Test if the input is serializable.
<i>dump</i> (obj, fp, **kw)	Serializes a Python object to a JSON5 compatible unicode string.
<i>dumps</i> (obj, **kw)	Serializes a Python object to a JSON5 compatible unicode string.
<i>Options</i>	Customizations for the <i>encoder_* (...)</i> function family.
<i>Json5EncoderException</i>	Base class of any exception thrown by the serializer.
<i>Json5UnstringifiableType</i> ([message, ...])	The encoder was not able to stringify the input, or it was told not to by the supplied <i>Options</i> .

## 2.1.2 Full Encoder Description

`pyjson5.encode(data, *, options=None, **options_kw)`  
Serializes a Python object to a JSON5 compatible unicode string.

```
encode(['Hello', 'world!']) == '["Hello","world!"]'
```

### Parameters

- **data** (*object*) – Python object to serialize.
- **options** (*Optional[Options]*) – Extra options for the encoder. If **options** and **options\_kw** are specified, then `options.update(**options_kw)` is used.
- **options\_kw** – See Option’s arguments.

### Raises

- **Json5EncoderException** – An exception occurred while encoding.
- **TypeError** – An argument had a wrong type.

### Returns

Unless `float('inf')` or `float('nan')` is encountered, the result will be valid JSON data (as of RFC8259).

The result is always ASCII. All characters outside of the ASCII range are escaped.

The result safe to use in an HTML template, e.g. `<a onclick='alert({{ encode(url) }})'>show message</a>`. Apostrophes `"'` are encoded as `"\u0027"`, less-than, greater-than, and ampersand likewise.

### Return type `str`

`pyjson5.encode_bytes(data, *, options=None, **options_kw)`  
Serializes a Python object to a JSON5 compatible bytes string.

```
encode_bytes(['Hello', 'world!']) == b'["Hello","world!"]'
```

### Parameters

- **data** (*object*) – see `encode(...)`
- **options** (*Optional[Options]*) – see `encode(...)`
- **options\_kw** – see `encode(...)`

### Raises

- **Json5EncoderException** – An exception occurred while encoding.
- **TypeError** – An argument had a wrong type.

### Returns see `encode(...)`

### Return type `bytes`

`pyjson5.encode_callback(data, cb, supply_bytes=False, *, options=None, **options_kw)`  
Serializes a Python object into a callback function.

The callback function `cb` gets called with single characters and strings until the input `data` is fully serialized.

```

encode_callback(['Hello', 'world!'], print)
#prints:
# [
#  "
#  Hello
#  "
#  ,
#  "
#  world!
#  "
#  ]

```

**Parameters**

- **data** (*object*) – see *encode(...)*
- **cb** (*Callable[[Union[bytes|str]], None]*) – A callback function. Depending on the truthyness of *supply\_bytes* either *bytes* or *str* is supplied.
- **supply\_bytes** (*bool*) – Call *cb(...)* with a *bytes* argument if true, otherwise *str*.
- **options** (*Optional[Options]*) – see *encode(...)*
- **options\_kw** – see *encode(...)*

**Raises**

- **Json5EncoderException** – An exception occurred while encoding.
- **TypeError** – An argument had a wrong type.

**Returns** The supplied argument *cb*.

**Return type** *Callable[[Union[bytes|str]], None]*

`pyjson5.encode_io(data, fp, supply_bytes=True, *, options=None, **options_kw)`

Serializes a Python object into a file-object.

The return value of `fp.write(...)` is not checked. If *fp* is unbuffered, then the result will be garbage!

**Parameters**

- **data** (*object*) – see *encode(...)*
- **fp** (*IOBase*) – A file-like object to serialize into.
- **supply\_bytes** (*bool*) – Call *fp.write(...)* with a *bytes* argument if true, otherwise *str*.
- **options** (*Optional[Options]*) – see *encode(...)*
- **options\_kw** – see *encode(...)*

**Raises**

- **Json5EncoderException** – An exception occurred while encoding.
- **TypeError** – An argument had a wrong type.

**Returns** The supplied argument *fp*.

**Return type** *IOBase*

`pyjson5.encode_noop(data, *, options=None, **options_kw)`

Test if the input is serializable.

Most likely you want to serialize data directly, and catch exceptions instead of using this function!

```
encode_noop({47: 11}) == True
encode_noop({47: object()}) == False
```

#### Parameters

- **data** (*object*) – see *encode(...)*
- **options** (*Optional[Options]*) – see *encode(...)*
- **options\_kw** – see *encode(...)*

**Returns** True iff data is serializable.

**Return type** bool

**class** `pyjson5.Options`

Customizations for the `encoder_*` (...) function family.

Immutable. Use `Options.update(**kw)` to create a **new** Options instance.

#### Parameters

- **quotationmark** (*str/None*) –
  - **str**: One character string that is used to surround strings.
  - **None**: Use default: `'"`.
- **tojson** (*str/False/None*) –
  - **str**: A special method to call on objects to return a custom JSON encoded string. Must return ASCII data!
  - **False**: No such member exists. (Default.)
  - **None**: Use default.
- **mappingtypes** (*Iterable[type]/False/None*) –
  - **Iterable[type]**: Classes that should be encoded to objects. Must be iterable over their keys, and implement `__getitem__`.
  - **False**: There are no objects. Any object will be encoded as list of keys as in `list(obj)`.
  - **None**: Use default: `[collections.abc.Mapping]`.

#### **mappingtypes**

The creation argument `mappingtypes`. () if False was specified.

#### **quotationmark**

The creation argument `quotationmark`.

#### **tojson**

The creation argument `tojson`. None if False was specified.

**update** (*self, \*args, \*\*kw*)

Creates a new Options instance by modifying some members.

### 2.1.3 Encoder Compatibility Functions

`pyjson5.dump(obj, fp, **kw)`

Serializes a Python object to a JSON5 compatible unicode string.

Use `encode_io(...)` instead!

```
dump(obj, fp) == encode_io(obj, fp)
```

#### Parameters

- **obj** (*object*) – Python object to serialize.
- **fp** (*IOBase*) – A file-like object to serialize into.
- **kw** – Silently ignored.

`pyjson5.dumps(obj, **kw)`

Serializes a Python object to a JSON5 compatible unicode string.

Use `encode(...)` instead!

```
dumps(obj) == encode(obj)
```

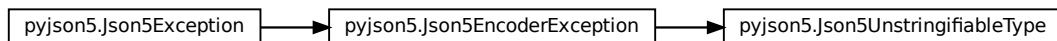
#### Parameters

- **obj** (*object*) – Python object to serialize.
- **kw** – Silently ignored.

**Returns** see `encode(data)`

**Return type** unicode

### 2.1.4 Encoder Exceptions



**class** `pyjson5.Json5EncoderException`

Base class of any exception thrown by the serializer.

**message**

Human readable error description

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**class** `pyjson5.Json5UnstringifiableType` (*message=None, unstringifiable=None*)

The encoder was not able to stringify the input, or it was told not to by the supplied `Options`.

**message**

Human readable error description

**unstringifiable**

The value that caused the problem.

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

## 2.2 Parser / Decoder

All valid JSON5 1.0.0 and JSON data can be read, unless the nesting level is absurdly high.

### 2.2.1 Quick Decoder Summary

<code>decode(data[, maxdepth, some])</code>	Decodes JSON5 serialized data from an <code>str</code> object.
<code>decode_buffer(obj[, maxdepth, some, wordlength])</code>	Decodes JSON5 serialized data from an object that supports the buffer protocol, e.g.
<code>decode_callback(cb[, maxdepth, some, args])</code>	Decodes JSON5 serialized data by invoking a callback.
<code>decode_io(fp[, maxdepth, some])</code>	Decodes JSON5 serialized data from a file-like object.
<code>load(fp, **kw)</code>	Decodes JSON5 serialized data from a file-like object.
<code>loads(s, *, encoding)</code>	Decodes JSON5 serialized data from a string.
<code>Json5DecoderException([message, result])</code>	Base class of any exception thrown by the parser.
<code>Json5NestingTooDeep</code>	The maximum nesting level on the input data was exceeded.
<code>Json5EOF</code>	The input ended prematurely.
<code>Json5IllegalCharacter([message, result, ...])</code>	An unexpected character was encountered.
<code>Json5ExtraData([message, result, character])</code>	The input contained extraneous data.
<code>Json5IllegalType([message, result, value])</code>	The user supplied callback function returned illegal data.

### 2.2.2 Full Decoder Description

`pyjson5.decode(data, maxdepth=None, some=False)`

Decodes JSON5 serialized data from an `str` object.

```
decode('["Hello", "world!"]') == ['Hello', 'world!']
```

**Parameters**

- **data** (*unicode*) – JSON5 serialized data
- **maxdepth** (*Optional[int]*) – Maximum nesting level before the parsing is aborted.
  - If `None` is supplied, then the value of the global variable `DEFAULT_MAX_NESTING_LEVEL` is used instead.
  - If the value is 0, then only literals are accepted, e.g. `false`, `47.11`, or `"string"`.
  - If the value is negative, then the any nesting level is allowed until Python's recursion limit is hit.
- **some** (*bool*) – Allow trailing junk.

**Raises**

- **`Json5DecoderException`** – An exception occurred while decoding.
- **`TypeError`** – An argument had a wrong type.

**Returns** Deserialized data.

**Return type** `object`

`pyjson5.decode_buffer(obj, maxdepth=None, some=False, wordlength=None)`

Decodes JSON5 serialized data from an object that supports the buffer protocol, e.g. bytearray.

```
obj = memoryview(b'["Hello", "world!"]')
decode_buffer(obj) == ['Hello', 'world!']
```

#### Parameters

- **`data`** (*object*) – JSON5 serialized data. The argument must support Python’s buffer protocol, i.e. `memoryview(...)` must work. The buffer must be contiguous.
- **`maxdepth`** (*Optional[int]*) – see `decode(...)`
- **`some`** (*bool*) – see `decode(...)`
- **`wordlength`** (*Optional[int]*) – Must be 0, 1, 2, 4 to denote UTF-8, UCS1, USC2 or USC4 data, resp. Surrogates are not supported. Decode the data to an `str` if need be. If `None` is supplied, then the buffer’s `itemsize` is used.

#### Raises

- **`Json5DecoderException`** – An exception occurred while decoding.
- **`TypeError`** – An argument had a wrong type.
- **`ValueError`** – The value of `wordlength` was invalid.

**Returns** see `decode(...)`

**Return type** `object`

`pyjson5.decode_callback(cb, maxdepth=None, some=False, args=None)`

Decodes JSON5 serialized data by invoking a callback.

```
cb = iter('["Hello", "world!"]').__next__
decode_callback(cb) == ['Hello', 'world!']
```

#### Parameters

- **`cb`** (*Callable[Any, Union[str|bytes|bytearray|int|None]]*) – A function to get values from. The function is called like `cb(*args)`, and it returns:
  - **`str, bytes, bytearray`**: `len(...)` == 0 denotes exhausted input. `len(...)` == 1 is the next character.
  - **`int`**: < 0 denotes exhausted input. >= 0 is the ordinal value of the next character.
  - **`None`**: input exhausted
- **`maxdepth`** (*Optional[int]*) – see `decode(...)`
- **`some`** (*bool*) – see `decode(...)`
- **`args`** (*Optional[Iterable[Any]]*) – Arguments to call `cb` with.

**Raises**

- *Json5DecoderException* – An exception occurred while decoding.
- *TypeError* – An argument had a wrong type.

**Returns** see `decode(...)`

**Return type** `object`

`pyjson5.decode_io(fp, maxdepth=None, some=True)`  
Decodes JSON5 serialized data from a file-like object.

```
fp = io.StringIO("""
    ['Hello', /* TODO look into specs whom to greet */]
    'Wolrd' // FIXME: look for typos
""")

decode_io(fp) == ['Hello']
decode_io(fp) == 'Wolrd'

fp.seek(0)

decode_io(fp, some=False)
# raises Json5ExtraData('Extra data U+0027 near 56', ['Hello'], "")
```

**Parameters**

- **fp** (*IOBase*) – A file-like object to parse from.
- **maxdepth** (*Optional[int] = None*) – see *decode(...)*
- **some** (*bool*) – see *decode(...)*

**Raises**

- *Json5DecoderException* – An exception occurred while decoding.
- *TypeError* – An argument had a wrong type.

**Returns** see `decode(...)`

**Return type** `object`

## 2.2.3 Decoder Compatibility Functions

`pyjson5.load(fp, **kw)`  
Decodes JSON5 serialized data from a file-like object.  
Use *decode\_io(...)* instead!

```
load(fp) == decode_io(fp, None, False)
```

**Parameters**

- **fp** (*IOBase*) – A file-like object to parse from.
- **kw** – Silently ignored.

**Returns** see `decode(...)`

**Return type** `object`



`pyjson5.loads(s, *, encoding=u'UTF-8', **kw)`  
 Decodes JSON5 serialized data from a string.

Use `decode(...)` instead!

```
loads(s) == decode(s)
```

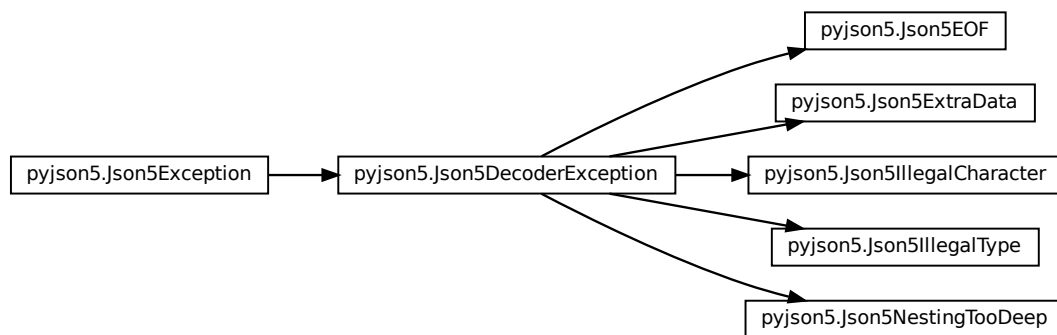
#### Parameters

- **s** (*object*) – Unless the argument is an `str`, it gets decoded according to the parameter `encoding`.
- **encoding** (*str*) – Codec to use if `s` is not an `str`.
- **kw** – Silently ignored.

**Returns** see `decode(...)`

**Return type** `object`

## 2.2.4 Decoder Exceptions



**class** `pyjson5.Json5DecoderException` (*message=None, result=None, \*args*)

Base class of any exception thrown by the parser.

**message**

Human readable error description

**result**

Deserialized data up until now.

**with\_traceback()**

`Exception.with_traceback(tb)` – set `self.__traceback__` to `tb` and return `self`.

**class** `pyjson5.Json5NestingTooDeep`

The maximum nesting level on the input data was exceeded.

**message**

Human readable error description

**result**

Deserialized data up until now.

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**class** pyjson5.Json5EOF

The input ended prematurely.

**message**

Human readable error description

**result**

Deserialized data up until now.

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**class** pyjson5.Json5IllegalCharacter(*message=None, result=None, character=None, \*args*)

An unexpected character was encountered.

**character**

Extraneous character.

**message**

Human readable error description

**result**

Deserialized data up until now.

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**class** pyjson5.Json5ExtraData(*message=None, result=None, character=None, \*args*)

The input contained extraneous data.

**character**

Extraneous character.

**message**

Human readable error description

**result**

Deserialized data up until now.

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**class** pyjson5.Json5IllegalType(*message=None, result=None, value=None, \*args*)

The user supplied callback function returned illegal data.

**message**

Human readable error description

**result**

Deserialized data up until now.

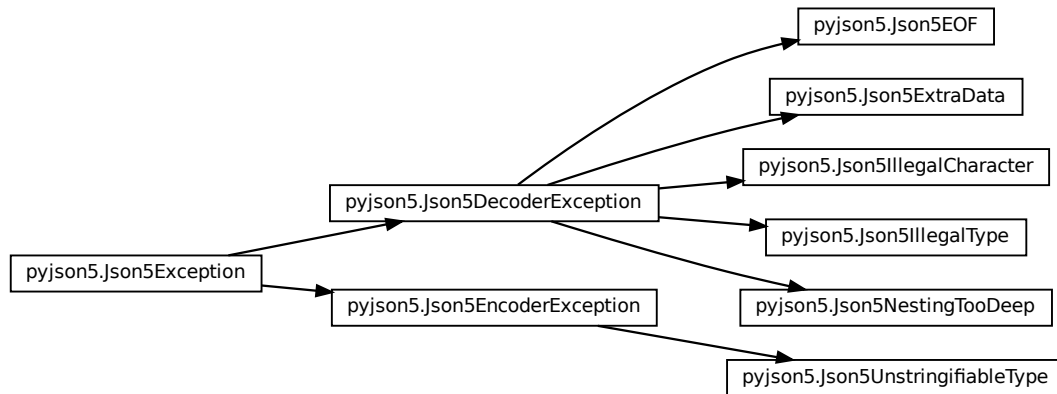
**value**

Value that caused the problem.

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

## 2.3 Exceptions



```
class pyjson5.Json5Exception(message=None, *args)
```

Base class of any exception thrown by PyJSON5.

**message**

Human readable error description

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

## 2.4 Performance

This library is written in Cython for a better performance than a pure-Python implementation could give you.

### 2.4.1 Decoder Performance

The library has about the same speed as the shipped `json` module for *pure* JSON data.

- Dataset: <https://github.com/zemirco/sf-city-lots-json>
- Version: Python 3.9.1+ (default, Feb 5 2021, 13:46:56)
- CPU: AMD Ryzen 7 2700 @ 3.7GHz
- `pyjson5.decode()`: **2.08 s** ± 7.49 ms per loop (*lower is better*)
- `json.loads()`: **2.71 s** ± 12.1 ms per loop
- The decoder works correctly: `json.loads(content) == pyjson5.loads(content)`

## 2.4.2 Encoder Performance

The encoder generates pure JSON data if there are no infinite or NaN values in the input, which are invalid in JSON. The serialized data is XML-safe, i.e. there are no cheverons <>, ampersands &, apostrophes ' or control characters in the output. The output is always ASCII regardless if you call `pyjson5.encode()` or `pyjson5.encode_bytes()`.

- Dataset: <https://github.com/zemirco/sf-city-lots-json>
- Python 3.9.1+ (default, Feb 5 2021, 13:46:56)
- CPU: AMD Ryzen 7 2700 @ 3.7GHz
- `pyjson5.encode()`: **1.37 s**  $\pm$  19.2 per loop (*lower is better*)
- `json.dumps()`: **3.66 s**  $\pm$  72.6 ms per loop
- `json.dumps() + xml.sax.saxutils.escape()`: **4.01 s**  $\pm$  21.3 ms per loop
- The encoder works correctly: `obj == json.loads(pyjson5.encode(obj))`

## 2.4.3 Benchmark

Using [Ultrajson's benchmark](#) you can tell for which kind of data PyJSON5 is fast, and for which data it is slow in comparison (*higher is better*):

	json	pyjson5	ujson	orjson
<b>Array with 256 doubles</b>				
encode	6,425	81,202	28,966	83,836
decode	16,759	34,801	34,794	80,655
<b>Array with 256 strings</b>				
encode	36,969	73,165	35,574	113,082
decode	42,730	38,542	38,386	60,732
<b>Array with 256 UTF-8 strings</b>				
encode	3,458	3,134	4,024	31,677
decode	2,428	2,498	2,491	1,750
<b>Array with 256 True values</b>				
encode	130,441	282,703	131,279	423,371
decode	220,657	262,690	264,485	262,283
<b>Array with 256 dict{string, int} pairs</b>				
encode	11,621	10,014	18,148	73,905
decode	17,802	19,406	19,391	23,478
<b>Dict with 256 arrays with 256 dict{string, int} pairs</b>				
encode	40	38	68	213
decode	43	49	48	51
<b>Medium complex object</b>				
encode	8,704	11,922	15,319	49,677
decode	12,567	14,042	13,985	19,481
<b>Complex object</b>				
encode	672	909	731	
decode	462	700	700	

- `ujson` == 4.0.3.dev9
- `orjson` == 3.5.1

## QUICK SUMMARY

<i>decode</i> (data[, maxdepth, some])	Decodes JSON5 serialized data from an <code>str</code> object.
<i>decode_buffer</i> (obj[, maxdepth, some, wordlength])	Decodes JSON5 serialized data from an object that supports the buffer protocol, e.g.
<i>decode_callback</i> (cb[, maxdepth, some, args])	Decodes JSON5 serialized data by invoking a callback.
<i>decode_io</i> (fp[, maxdepth, some])	Decodes JSON5 serialized data from a file-like object.
<i>load</i> (fp, **kw)	Decodes JSON5 serialized data from a file-like object.
<i>loads</i> (s, *[, encoding])	Decodes JSON5 serialized data from a string.
<i>encode</i> (data, *[, options])	Serializes a Python object to a JSON5 compatible unicode string.
<i>encode_bytes</i> (data, *[, options])	Serializes a Python object to a JSON5 compatible bytes string.
<i>encode_callback</i> (data, cb[, supply_bytes, ...])	Serializes a Python object into a callback function.
<i>encode_io</i> (data, fp[, supply_bytes, options])	Serializes a Python object into a file-object.
<i>encode_noop</i> (data, *[, options])	Test if the input is serializable.
<i>dump</i> (obj, fp, **kw)	Serializes a Python object to a JSON5 compatible unicode string.
<i>dumps</i> (obj, **kw)	Serializes a Python object to a JSON5 compatible unicode string.
<i>Options</i>	Customizations for the <code>encoder_*</code> (...) function family.
<i>Json5EncoderException</i>	Base class of any exception thrown by the serializer.
<i>Json5DecoderException</i> ([message, result])	Base class of any exception thrown by the parser.



## COMPATIBILITY

At least CPython / PyPy 3.5, and a C++11 compatible compiler (such as GCC 5.2+) is needed.

---

[Glossary / Index](#)





## C

character (*pyjson5.Json5ExtraData attribute*), 14  
 character (*pyjson5.Json5IllegalCharacter attribute*), 14

## D

decode() (*in module pyjson5*), 10  
 decode\_buffer() (*in module pyjson5*), 11  
 decode\_callback() (*in module pyjson5*), 11  
 decode\_io() (*in module pyjson5*), 12  
 dump() (*in module pyjson5*), 9  
 dumps() (*in module pyjson5*), 9

## E

encode() (*in module pyjson5*), 6  
 encode\_bytes() (*in module pyjson5*), 6  
 encode\_callback() (*in module pyjson5*), 6  
 encode\_io() (*in module pyjson5*), 7  
 encode\_noop() (*in module pyjson5*), 7

## J

Json5DecoderException (*class in pyjson5*), 13  
 Json5EncoderException (*class in pyjson5*), 9  
 Json5EOF (*class in pyjson5*), 14  
 Json5Exception (*class in pyjson5*), 15  
 Json5ExtraData (*class in pyjson5*), 14  
 Json5IllegalCharacter (*class in pyjson5*), 14  
 Json5IllegalType (*class in pyjson5*), 14  
 Json5NestingTooDeep (*class in pyjson5*), 13  
 Json5UnstringifiableType (*class in pyjson5*), 9

## L

load() (*in module pyjson5*), 12  
 loads() (*in module pyjson5*), 13

## M

mappingtypes (*pyjson5.Options attribute*), 8  
 message (*pyjson5.Json5DecoderException attribute*), 13  
 message (*pyjson5.Json5EncoderException attribute*), 9  
 message (*pyjson5.Json5EOF attribute*), 14

message (*pyjson5.Json5Exception attribute*), 15  
 message (*pyjson5.Json5ExtraData attribute*), 14  
 message (*pyjson5.Json5IllegalCharacter attribute*), 14  
 message (*pyjson5.Json5IllegalType attribute*), 14  
 message (*pyjson5.Json5NestingTooDeep attribute*), 13  
 message (*pyjson5.Json5UnstringifiableType attribute*), 9

## O

Options (*class in pyjson5*), 8

## Q

quotationmark (*pyjson5.Options attribute*), 8

## R

result (*pyjson5.Json5DecoderException attribute*), 13  
 result (*pyjson5.Json5EOF attribute*), 14  
 result (*pyjson5.Json5ExtraData attribute*), 14  
 result (*pyjson5.Json5IllegalCharacter attribute*), 14  
 result (*pyjson5.Json5IllegalType attribute*), 14  
 result (*pyjson5.Json5NestingTooDeep attribute*), 13

## T

tojson (*pyjson5.Options attribute*), 8

## U

unstringifiable (*pyjson5.Json5UnstringifiableType attribute*), 9  
 update() (*pyjson5.Options method*), 8

## V

value (*pyjson5.Json5IllegalType attribute*), 14

## W

with\_traceback() (*pyjson5.Json5DecoderException method*), 13  
 with\_traceback() (*pyjson5.Json5EncoderException method*), 9  
 with\_traceback() (*pyjson5.Json5EOF method*), 14  
 with\_traceback() (*pyjson5.Json5Exception method*), 15

`with_traceback()` (*pyjson5.Json5ExtraData*  
*method*), [14](#)

`with_traceback()` (*pyjson5.Json5IllegalCharacter*  
*method*), [14](#)

`with_traceback()` (*pyjson5.Json5IllegalType*  
*method*), [14](#)

`with_traceback()` (*pyjson5.Json5NestingTooDeep*  
*method*), [14](#)

`with_traceback()` (*pyj-*  
*son5.Json5UnstringifiableType* *method*),  
[10](#)