



3.10.0



Go

json — Codificador y decodificador JSON

Código fuente: [Lib/json/_init_.py](#)

JSON (JavaScript Object Notation), especificado por RFC 7159 (que obsoleta RFC 4627) y por ECMA-404, es un formato ligero de intercambio de datos inspirado en la sintaxis literal de objetos JavaScript (aunque no es un subconjunto estricto de JavaScript [1].)

json expone una API familiar para los usuarios de los módulos estándar de `marshal` de biblioteca y `pickle`.

Codificación de jerarquías básicas de objetos Python:

```
>>> 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"]'
```

Codificación compacta:

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

Impresión bonita:

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

Decodificación 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"]')
```



3.10.0



Go

Decodificación de objetos JSON especializada:

```
>>> 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')
```

Ampliación de [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)
...
>>> 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', ']']]
```

Usando [json.tool](#) desde el shell para validar e imprimir bonitamente:

```
$ 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)
```

Consulte [Interfaz de línea de comandos](#) para obtener documentación detallada.

Nota: JSON es un subconjunto de [YAML 1.2](#). El JSON producido por la configuración predeterminada de este módulo (en particular, el valor predeterminado de *los separadores*) también es un subconjunto de YAML 1.0 y 1.1. Por lo tanto, este módulo también se puede utilizar como un serializador YAML.

Nota: Los codificadores y decodificadores de este módulo conservan el orden de entrada y salida de forma predeterminada. El pedido solo se pierde si los contenedores subyacentes no están ordenados.

Antes de Python 3.7, no se garantizaba que se ordenara [el dictado](#), por lo que las entradas y salidas generalmente se codificaban a menos que se realizaran [colecciones](#). [OrderedDict](#) fue solicitado específicamente. A partir de Python 3.7, el [dictado](#) regular se convirtió en preservación de orden, por lo que ya no es necesario especificar [colecciones](#). [OrderedDict](#) para la generación y análisis de JSON.



3.10.0



Go

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

Serialice `obj` como una secuencia con formato JSON a `fp` (un objeto similar a un [archivo compatible](#)) utilizando esta tabla de [conversión](#). `.write()`

Si `skipkeys` es true (valor predeterminado:), entonces las claves de dict que no son de un tipo básico (`str`, `int`, `float`, `bool`,) se omitirán en lugar de generar un `TypeError`. `False` `None`

El módulo `json` siempre produce objetos `str`, no objetos `bytes`. Por lo tanto, debe admitir la entrada `str`. `fp.write()`

Si `ensure_ascii` es true (el valor predeterminado), se garantiza que la salida tendrá todos los caracteres entrantes que no son ASCII escapados. Si `ensure_ascii` es false, estos caracteres se generarán tal cual.

Si `check_circular` es false (valor predeterminado:), se omitirá la comprobación de referencia circular para los tipos de contenedor y una referencia circular dará lugar a un `OverflowError` (o peor). `True`

Si `allow_nan` es false (valor predeterminado:), entonces será un `ValueError` serializar valores `flotantes` fuera del rango (, ,) en estricto cumplimiento de la especificación JSON. Si `allow_nan` es true, se utilizarán sus equivalentes de JavaScript (, ,). `True` `nan` `inf` `-inf` `NaN` `Infinity` `-Infinity`

Si `la sangría` es un entero o cadena no negativo, los elementos de la matriz JSON y los miembros del objeto se imprimirán bonitamente con ese nivel de sangría. Un nivel de sangría de 0, negativo o solo insertará nuevas líneas. (el valor predeterminado) selecciona la representación más compacta. El uso de una sangría entera positiva sangría de tantos espacios por nivel. Si `la sangría` es una cadena (como), esa cadena se utiliza para sangría en cada nivel. `""` `None` `\t`

Modificado en la versión 3.2: Permitir cadenas para `sangría` además de enteros.

Si se especifica, `los separadores` deben ser una tupla. El valor predeterminado es si `la sangría` es y de lo contrario. Para obtener la representación JSON más compacta, debe especificar eliminar el espacio en blanco. `(item_separator, key_separator)` `(' ', ',')` `None` `(' ', ':')` `(' ', ',')` `(' ', ':')`

Modificado en la versión 3.4: Utilícies como valor predeterminado si `la sangría` no es . `(' ', ',')` `None`

Si se especifica, `el valor predeterminado` debe ser una función a la que se llama para objetos que de otro modo no se pueden serializar. Debe devolver una versión JSON codificable del objeto o generar un `TypeError`. Si no se especifica, se genera `TypeError`.

Si `sort_keys` es true (valor predeterminado:), entonces la salida de los diccionarios se ordenará por clave. `False`

Para utilizar una subclase `JSONEncoder` personalizada (por ejemplo, una que anule el método para serializar tipos adicionales), especifíquela con `el cls kwarg`; de lo contrario, se utiliza `JSONEncoder.default()`

Modificado en la versión 3.6: Todos los parámetros opcionales ahora son [solo palabras clave](#).

Nota: A diferencia de `pickle` y `marshal`, JSON no es un protocolo enmarcado, por lo que intentar serializar varios objetos con llamadas repetidas a `dump()` usando el mismo `fp` dará como resultado un archivo JSON no válido.



3.10.0



Go

Serialice `obj` a un `str` con formato JSON utilizando esta [tabla de conversión](#). Los argumentos tienen el mismo significado que en `dump()`.

Nota: Las claves en los pares clave/valor de JSON son siempre del tipo `str`. Cuando un diccionario se convierte en JSON, todas las claves del diccionario se obligan a cadenas. Como resultado de esto, si un diccionario se convierte en JSON y luego se vuelve a convertir en un diccionario, es posible que el diccionario no sea igual al original. Es decir, `loads(dumps(x)) != x` si `x` tiene claves que no son de cadena.

`json.loads(fp, *, cls=Ninguno, object_hook=Ninguno, parse_float=Ninguno, parse_int=Ninguno, parse_constant=Ninguno, object_pairs_hook=Ninguno, **kw)`

Deserializar `fp` (un archivo de [texto](#) -supporting o [archivo binario](#) que contiene un documento JSON) a un objeto Python utilizando esta tabla de [conversión](#). `.read()`

`object_hook` es una función opcional a la que se llamará con el resultado de cualquier objeto literal decodificado (un [dictado](#)). Se utilizará el valor devuelto de `object_hook` en lugar del [dictado](#). Esta característica se puede utilizar para implementar decodificadores personalizados (por ejemplo, sugerencias de clase [JSON-RPC](#)).

`object_pairs_hook` es una función opcional a la que se llamará con el resultado de cualquier objeto literal decodificado con una lista ordenada de pares. Se utilizará el valor devuelto de `object_pairs_hook` en lugar del [dictado](#). Esta característica se puede utilizar para implementar decodificadores personalizados. Si también se define `object_hook`, la `object_pairs_hook` tiene prioridad.

Modificado en la versión 3.1: Se ha añadido compatibilidad con `object_pairs_hook`.

`parse_float`, si se especifica, se llamará con la cadena de cada flotador JSON que se decodificará. De forma predeterminada, esto es equivalente a `.`. Esto se puede utilizar para utilizar otro tipo de datos o analizador para flotadores JSON (por ejemplo, `decimal.Decimal.float(num_str)`)

`parse_int`, si se especifica, se llamará con la cadena de cada JSON int que se decodificará. De forma predeterminada, esto es equivalente a `.`. Esto se puede utilizar para utilizar otro tipo de datos o analizador para enteros JSON (por ejemplo, `float.int(num_str)`)

`parse_constant`, si se especifica, se llamará con una de las siguientes cadenas: `, , ,`. Esto se puede usar para generar una excepción si se encuentran números JSON no válidos. `'-Infinity' 'Infinity' 'NaN'`

Cambiado en la versión 3.1: `parse_constant` ya no se llama 'null', 'true', 'false'.

Para utilizar una subclase `JSONDecoder` personalizada, especifíquela con el kwarg; de lo contrario, se utiliza `JSONDecoder`. Se pasarán argumentos de palabras clave adicionales al constructor de la clase `cls`

Si los datos que se están deserializando no son un documento JSON válido, se creará un `JSONDecodeError`.

Modificado en la versión 3.6: Todos los parámetros opcionales ahora son [solo palabras clave](#).

Cambiado en la versión 3.6: `fp` ahora puede ser un [archivo binario](#). La codificación de entrada debe ser UTF-8, UTF-16 o UTF-32.

`json.loads(s, *, cls=Ninguno, object_hook=Ninguno, parse_float=Ninguno, parse_int=Ninguno, parse_constant=Ninguno, object_pairs_hook=Ninguno, **kw)`



3.10.0



Go

Los otros argumentos tienen el mismo significado que en [load\(\)](#).

Si los datos que se están deserializando no son un documento JSON válido, se creará un [JSONDecodeError](#).

Cambiado en la versión 3.6: `s` ahora puede ser de tipo [bytes](#) o [bytearray](#). La codificación de entrada debe ser UTF-8, UTF-16 o UTF-32.

Modificado en la versión 3.9: Se ha eliminado *la codificación del argumento de palabra clave*.

Codificadores y decodificadores

```
class (*, object_hook=Ninguno, parse_float=Ninguno, parse_int=Ninguno, parse_constant=Ninguno, estricto=Verdadero, object_pairs_hook=Ninguno json.JSONDecoder)
```

Decodificador JSON simple.

Realiza las siguientes traducciones en la decodificación de forma predeterminada:

JSON	Pitón
objeto	dict
arreglo	lista
cuerda	str
número (int)	int
número (real)	flotar
verdadero	Verdadero
falso	Falso
nulo	Ninguno

También entiende `,`, `,` y como sus valores correspondientes, que está fuera de la especificación JSON. `NaN` `-Infinity` `float`

`object_hook`, si se especifica, se llamará con el resultado de cada objeto JSON decodificado y su valor devuelto se utilizará en lugar del [dictado](#)dado. Esto se puede utilizar para proporcionar deserializaciones personalizadas (por ejemplo, para admitir sugerencias de clases [JSON-RPC](#)).

`object_pairs_hook`, si se especifica se llamará con el resultado de cada objeto JSON decodificado con una lista ordenada de pares. Se utilizará *el* valor devuelto de `object_pairs_hook` en lugar del [dictado](#). Esta característica se puede utilizar para implementar decodificadores personalizados. Si también se define `object_hook`, la `object_pairs_hook` tiene prioridad.

Modificado en la versión 3.1: Se ha añadido compatibilidad con `object_pairs_hook`.

`parse_float`, si se especifica, se llamará con la cadena de cada flotador JSON que se decodificará. De forma predeterminada, esto es equivalente a `.`. Esto se puede utilizar para utilizar otro tipo de datos o analizador para flotadores JSON (por ejemplo, [decimal.Decimal](#)).`float(num_str)`



3.10.0



Go

para enteros JSON (por ejemplo, `float`).`int(num_str)`

`parse_constant`, si se especifica, se llamará con una de las siguientes cadenas: , , . Esto se puede usar para generar una excepción si se encuentran números JSON no válidos. '-Infinity' 'Infinity' 'NaN'

Si `stricto` es `false` (es el valor predeterminado), se permitirán caracteres de control dentro de las cadenas. Los caracteres de control en este contexto son aquellos con códigos de caracteres en el rango de 0 a 31, incluidos (tab), y .`True` '\t' '\n' '\r' '\0'

Si los datos que se están deserializando no son un documento JSON válido, se creará un `JSONDecodeError`.

Modificado en la versión 3.6: Todos los parámetros ahora son **solo palabras clave**.

`decode(s)`

Devuelve la representación de Python de `s` (una instancia `str` que contiene un documento JSON).

`JSONDecodeError` se planteará si el documento JSON dado no es válido.

`raw_decode(s)`

Decodifique un documento JSON de `s` (un `str` que comienza con un documento JSON) y devuelva una tupla 2 de la representación de Python y el índice en `s` donde terminó el documento.

Esto se puede usar para decodificar un documento JSON a partir de una cadena que puede tener datos extraños al final.

```
class (*, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True,
sort_keys=False, indent=None, separators= Ninguno, predeterminado=Ninguno
json.JSONEncoder)
```

Codificador JSON extensible para estructuras de datos Python.

Admite los siguientes objetos y tipos de forma predeterminada:

Pitón	JSON
<code>dict</code>	objeto
<code>lista, tupla</code>	arreglo
<code>str</code>	cuerda
<code>Int, float, int- & float-derived Enums</code>	número
<code>Verdadero</code>	verdadero
<code>Falso</code>	falso
<code>Ninguno</code>	nulo

Modificado en la versión 3.4: Se ha agregado compatibilidad con las clases Enum derivadas de `int` y `float`.

Para extender esto para reconocer otros objetos, subclase e implementar un método `default()` con otro método que devuelva un objeto serializable para si es posible, de lo contrario debe llamar a la implementación de superclase (para generar `TypeError`).o



3.10.0



Go

Si `ensure_ascii` es true (el valor predeterminado), se garantiza que la salida tendrá todos los caracteres entrantes que no son ASCII escapados. Si `ensure_ascii` es false, estos caracteres se generarán tal cual.

Si `check_circular` es true (el valor predeterminado), se comprobarán las listas, los dictados y los objetos codificados personalizados en busca de referencias circulares durante la codificación para evitar una recursión infinita (lo que causaría un [OverflowError](#)). De lo contrario, no se lleva a cabo tal verificación.

Si `allow_nan` es true (el valor predeterminado), entonces , , y se codificará como tal. Este comportamiento no es compatible con la especificación JSON, pero es coherente con la mayoría de los codificadores y decodificadores basados en JavaScript. De lo contrario, será un [ValueError](#) codificar dichos flotantes. `Nan` `Infinity` `-Infinity`

Si `sort_keys` es true (valor predeterminado:), entonces la salida de los diccionarios se ordenará por clave; esto es útil para las pruebas de regresión para garantizar que las serializaciones JSON se puedan comparar en el día a día. `False`

Si `la sangría` es un entero o cadena no negativo, los elementos de la matriz JSON y los miembros del objeto se imprimirán bonitamente con ese nivel de sangría. Un nivel de sangría de 0, negativo o solo insertará nuevas líneas. (el valor predeterminado) selecciona la representación más compacta. El uso de una sangría entera positiva sangría de tantos espacios por nivel. Si `la sangría` es una cadena (como), esa cadena se utiliza para sangría en cada nivel. `"" None "\t"`

Modificado en la versión 3.2: Permitir cadenas para `sangría` además de enteros.

Si se especifica, `los separadores` deben ser una tupla. El valor predeterminado es si `la sangría` es y de lo contrario. Para obtener la representación JSON más compacta, debe especificar eliminar el espacio en blanco. `(item_separator, key_separator) ('', ', ', ': ') None ('', ', ', ': ') ('', ', ', ': ')`

Modificado en la versión 3.4: Utilícies como valor predeterminado si `la sangría` no es `.('', ', ', ': ') None`

Si se especifica, `el valor predeterminado` debe ser una función a la que se llama para objetos que de otro modo no se pueden serializar. Debe devolver una versión JSON codificable del objeto o generar un [TypeError](#). Si no se especifica, se genera [TypeError](#).

Modificado en la versión 3.6: Todos los parámetros ahora son [solo palabras clave](#).

`default(o)`

Implemente este método en una subclase tal que devuelva un objeto serializable para `o`, o llame a la implementación base (para generar un [TypeError](#)).

Por ejemplo, para admitir iteradores arbitrarios, puede implementar `default()` de la siguiente manera:

```
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 json.JSONEncoder.default(self, o)
```

`encode(o)`



3.10.0



Go

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

>>>

iterencode(*o*)

Codifique el objeto dado, *o*, y produzca cada representación de cadena como disponible. Por ejemplo:

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

Excepciones

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

Subclase de `ValueError` con los siguientes atributos adicionales:

msg

El mensaje de error sin formato.

doc

El documento JSON que se está analizando.

pos

Índice de inicio del *documento en el* que se ha fallado el análisis.

lineno

La línea correspondiente a *pos*.

colno

La columna correspondiente a *pos*.

Nuevo en la versión 3.5.

Cumplimiento de normas e interoperabilidad

El formato JSON está especificado por [RFC 7159](#) y por [ECMA-404](#). Esta sección detalla el nivel de cumplimiento de este módulo con el RFC. Para simplificar, no se consideran las subclases `JSONEncoder` y `JSONDecoder`, ni los parámetros distintos de los mencionados explícitamente.

Este módulo no cumple con el RFC de manera estricta, implementando algunas extensiones que son JavaScript válido pero NO JSON válido. En particular:

- Se aceptan y se generan valores numéricos infinitos y NaN;
- Se aceptan nombres repetidos dentro de un objeto y solo se utiliza el valor del par apellido-valor.

Dado que el RFC permite que los analizadores compatibles con RFC acepten textos de entrada que no son compatibles con RFC, el deserializador de este módulo es técnicamente compatible con RFC en la configuración predeterminada.

Codificaciones de caracteres



3.10.0



Go

Según lo permitido, aunque no requerido, por el RFC, el serializador de este módulo establece `ensure_ascii = True` de forma predeterminada, escapando así de la salida para que las cadenas resultantes solo contengan caracteres ASCII.

A parte del parámetro `ensure_ascii`, este módulo se define estrictamente en términos de conversión entre objetos Python y `cadenasUnicode` y, por lo tanto, no aborda directamente el problema de las codificaciones de caracteres.

El RFC prohíbe agregar una marca de orden de byte (BOM) al inicio de un texto JSON, y el serializador de este módulo no agrega una lista de materiales a su salida. El RFC permite, pero no requiere, que los deserializadores JSON ignoren una lista de materiales inicial en su entrada. El deserializador de este módulo genera un `ValueError` cuando hay una lista de materiales inicial.

El RFC no prohíbe explícitamente las cadenas JSON que contienen secuencias de bytes que no corresponden a caracteres Unicode válidos (por ejemplo, sustitutos UTF-16 no emparejados), pero sí señala que pueden causar problemas de interoperabilidad. De forma predeterminada, este módulo acepta y genera (cuando está presente en el `str`) puntos de código originales para dichas secuencias.

Valores numéricos infinitos y NaN

El RFC no permite la representación de valores numéricos infinitos o NaN. A pesar de eso, por defecto, este módulo acepta y genera , y como si fueran valores literales de número JSON válidos: `Infinity` `-Infinity` `NaN`

```
>>> # 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
```

En el serializador, el parámetro `allow_nan` se puede utilizar para alterar este comportamiento. En el deserializador, el parámetro `parse_constant` se puede utilizar para modificar este comportamiento.

Nombres repetidos dentro de un objeto

El RFC especifica que los nombres dentro de un objeto JSON deben ser únicos, pero no exige cómo se deben manejar los nombres repetidos en los objetos JSON. De forma predeterminada, este módulo no plantea una excepción; en su lugar, ignora todos menos el par apellido-valor para un nombre dado:

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

El parámetro `object_pairs_hook` se puede utilizar para modificar este comportamiento.

Valores de nivel superior no objeto, no matriz

La versión anterior de JSON especificada por el [obsoleto RFC 4627](#) requería que el valor de nivel superior de un texto JSON debe ser un objeto o matriz JSON ([dictado](#) o [listado](#) de Python), y no puede ser un valor



3.10.0



Go

De todos modos, para una máxima interoperabilidad, es posible que desee adherirse voluntariamente a la restricción usted mismo.

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

```
$ 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",
```



3.10.0



Go

J

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.

New in version 3.5.

--no-ensure-ascii

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

New in version 3.9.

--json-lines

Parse every input line as separate JSON object.

New in version 3.8.

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

Mutually exclusive options for whitespace control.

New 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.