sdjson

Release 0.4.0

Custom JSON Encoder for Python utilising functools.singledispatch to support custom encoders for both Python's built-in classes and user-created classes, without as much legwork.

Dominic Davis-Foster

Apr 24, 2024

Contents

Based on https://treyhunner.com/2013/09/singledispatch-json-serializer/ and Python's json module.

Chapter

ONE

Installation

1.1 from PyPI

```
$ python3 -m pip install sdjson --user
```

1.2 from Anaconda

First add the required channels

```
$ conda config --add channels https://conda.anaconda.org/conda-forge
$ conda config --add channels https://conda.anaconda.org/domdfcoding
```

Then install

```
$ conda install sdjson
```

1.3 from GitHub

```
$ python3 -m pip install git+https://github.com/domdfcoding/singledispatch-json@master --user
```

Chapter

TWO

API Reference

JSON encoder utilising functools.singledispatch to support custom encoders for both Python's built-in classes and user-created classes, without as much legwork.

Creating and registering a custom encoder is as easy as:

```
>>> import sdjson
>>>
>>> @sdjson.register_encoder(MyClass)
>>> def encode_myclass(obj):
...     return dict(obj)
>>>
```

In this case, `MyClass` can be made JSON-serializable simply by calling `dict` on it. If your class requires more complicated logic to make it JSON-serializable, do that here.

Then, to dump the object to a string:

```
>>> class_instance = MyClass()
>>> print(sdjson.dumps(class_instance))
'{"menu": ["egg and bacon", "egg sausage and bacon", "egg and spam", "egg bacon and␣
↪spam"],
"today\'s special": "Lobster Thermidor au Crevette with a Mornay sauce served in a␣
↪Provencale
manner with shallots and aubergines garnished with truffle pate, brandy and with a␣
↪fried egg
on top and spam."}'
>>>
```

Or to dump to a file:

```
>>> with open("spam.json", "w") as fp:
...     sdjson.dumps(class_instance, fp)
...
>>>
```

`sdjson` also provides access to `load()`, `loads()`, `JSONDecoder`, `JSONDecodeError`, and `JSONEncoder` from the `json` module, allowing you to use `sdjson` as a drop-in replacement for `json`.

If you wish to dump an object without using the custom encoders, you can pass a different `JSONEncoder` subclass, or indeed `JSONEncoder` itself to get the stock functionality.

```
>>> sdjson.dumps(class_instance, cls=sdjson.JSONEncoder)
>>>
```

When you've finished, if you want to unregister the encoder you can run:

```
>>> sdjson.unregister_encoder(MyClass)
>>>
```

to remove the encoder for `MyClass`. If you want to replace the encoder with a different one it is not necessary to call this function: the *`@sdjson.register_encoder`* decorator will replace any existing decorator for the given class.

Classes:

`JSONDecoder`(*args, **kwargs)	Alias of `json.JSONDecoder`.
`JSONEncoder`(*args, **kwargs)	Alias of `json.JSONEncoder`.

Functions:

`dump`(obj, fp, **kwargs)	Serialize custom Python classes to JSON.
`dumps`(obj, *[, skipkeys, ensure_ascii, ...])	Serialize custom Python classes to JSON.
`load`(*args, **kwargs)	Alias of `json.load()`.
`loads`(*args, **kwargs)	Alias of `json.loads()`.
`register_encoder`(cls[, func])	Registers a new handler for the given type.
`unregister_encoder`(cls)	Unregister the handler for the given type.

class JSONDecoder(**args*, ***kwargs*)

Bases: `JSONDecoder`

Alias of `json.JSONDecoder`.

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

Methods:

`decode`(*args, **kwargs)	Return the Python representation of `s` (a `str` instance containing a JSON document).
`raw_decode`(*args, **kwargs)	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.

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

raw_decode(**args*, ***kwargs*)
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 JSONEncoder(**args*, ***kwargs*)
Bases: `JSONEncoder`

Alias of `json.JSONEncoder`.

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	string
int, 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:

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`).
encode(o)	Return a JSON string representation of a Python data structure.
iterencode(o[, _one_shot])	Encode the given object and yield each string representation as available.

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 JSONEncoder.default(self, o)
```

Return type `Any`

encode(*o*)

Return a JSON string representation of a Python data structure.

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

Return type Any

iterencode(*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)
```

Return type Iterator[str]

dump(*obj*, *fp*, ***kwargs*)

Serialize custom Python classes to JSON. Custom classes can be registered using the `@encoders.register(<type>)` decorator.

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`, `int`, `float`, `bool`, `None`) will be skipped instead of raising a `TypeError`.

If `ensure_ascii` is false, then the strings written to `fp` can contain non-ASCII characters if they appear in strings contained in `obj`. Otherwise, all such characters are escaped in JSON strings.

If `check_circular` is false, then the circular reference check for container types will be skipped and a circular reference will result in an `RecursionError` (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.

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.

`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*, *default=None*, *sort_keys=False*, ***kwargs*)

Serialize custom Python classes to JSON. Custom classes can be registered using the `@encoders.register(<type>)` decorator.

Serialize `obj` to a JSON formatted `str`.

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

If `ensure_ascii` is false, then the return value can contain non-ASCII characters if they appear in strings contained in `obj`. Otherwise, all such characters are escaped in JSON strings.

If `check_circular` is false, then the circular reference check for container types will be skipped and a circular reference will result in an `RecursionError` (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.

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.

`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.

Return type `str`

load(**args*, ***kwargs*)

Alias of `json.load()`.

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

`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. 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(**args*, ***kwargs*)

Alias of `json.loads()`.

Deserialize `s` (a `str`, `bytes` or `bytearray` instance containing a JSON document) to a Python object.

`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. 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.

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

register_encoder(*cls*, *func=None*)

Registers a new handler for the given type.

Can be used as a decorator or a regular function:

```
@register_encoder(bytes)
def bytes_encoder(obj):
    return obj.decode("UTF-8")

def int_encoder(obj):
    return int(obj)

register_encoder(int, int_encoder)
```

Parameters

- **cls** (Type)
- **func** (Optional[Callable]) – Default None.

Return type Callable

unregister_encoder(*cls*)

Unregister the handler for the given type.

```
unregister_encoder(int)
```

Parameters **cls** (Type)

Raises **KeyError** – if no handler is found.

Chapter

THREE

Contributing

`sdjson` uses tox to automate testing and packaging, and pre-commit to maintain code quality.

Install `pre-commit` with `pip` and install the git hook:

```
$ python -m pip install pre-commit
$ pre-commit install
```

3.1 Coding style

formate is used for code formatting.

It can be run manually via `pre-commit`:

```
$ pre-commit run formate -a
```

Or, to run the complete autoformatting suite:

```
$ pre-commit run -a
```

3.2 Automated tests

Tests are run with `tox` and `pytest`. To run tests for a specific Python version, such as Python 3.6:

```
$ tox -e py36
```

To run tests for all Python versions, simply run:

```
$ tox
```

3.3 Type Annotations

Type annotations are checked using `mypy`. Run `mypy` using `tox`:

```
$ tox -e mypy
```

3.4 Build documentation locally

The documentation is powered by Sphinx. A local copy of the documentation can be built with `tox`:

```
$ tox -e docs
```

Chapter

FOUR

Downloading source code

The `sdjson` source code is available on GitHub, and can be accessed from the following URL: `https://github.com/domdfcoding/singledispatch-json`

If you have `git` installed, you can clone the repository with the following command:

```
$ git clone https://github.com/domdfcoding/singledispatch-json
```

```
Cloning into 'singledispatch-json'...
remote: Enumerating objects: 47, done.
remote: Counting objects: 100% (47/47), done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 173 (delta 16), reused 17 (delta 6), pack-reused 126
Receiving objects: 100% (173/173), 126.56 KiB | 678.00 KiB/s, done.
Resolving deltas: 100% (66/66), done.
```

Alternatively, the code can be downloaded in a 'zip' file by clicking:
Clone or download –> Download Zip

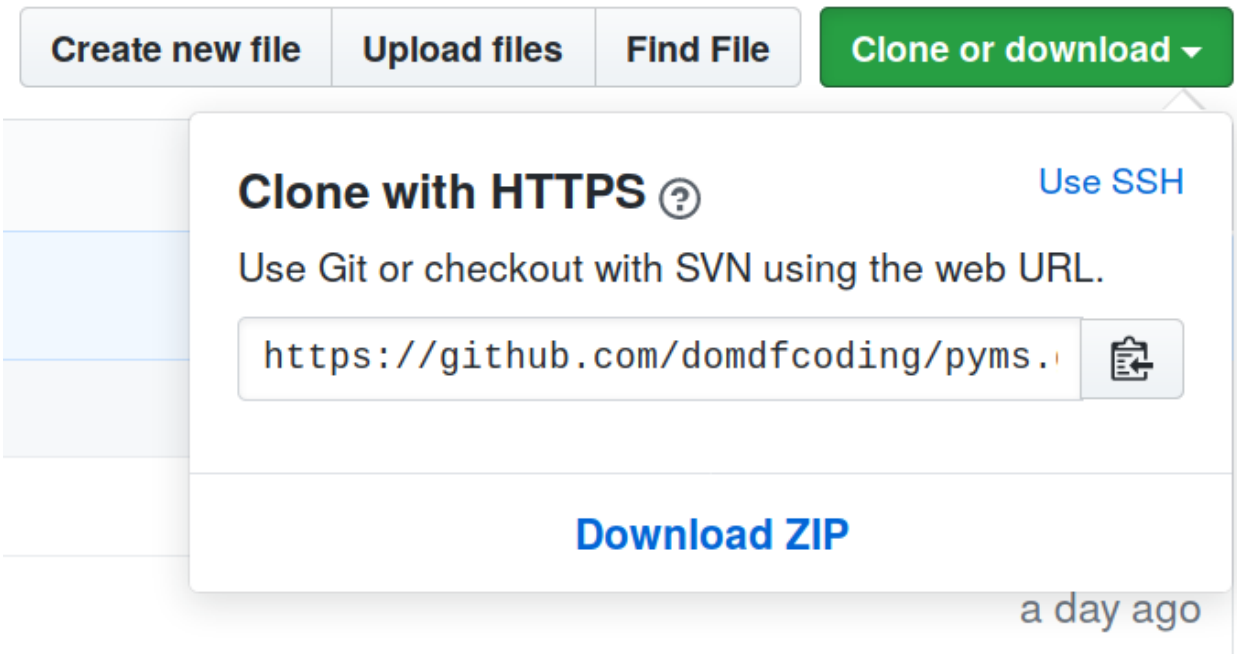

Fig. 1: Downloading a 'zip' file of the source code

4.1 Building from source

The recommended way to build `sdjson` is to use tox:

```
$ tox -e build
```

The source and wheel distributions will be in the directory `dist`.

If you wish, you may also use pep517.build or another **PEP 517**-compatible build tool.

Chapter

FIVE

License

`sdjson` is licensed under the MIT License

A short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

Permissions

- Commercial use – The licensed material and derivatives may be used for commercial purposes.
- Modification – The licensed material may be modified.
- Distribution – The licensed material may be distributed.
- Private use – The licensed material may be used and modified in private.

Conditions

- License and copyright notice – A copy of the license and copyright notice must be included with the licensed material.

Limitations

- Liability – This license includes a limitation of liability.
- Warranty – This license explicitly states that it does NOT provide any warranty.

See more information on choosealicense.com ⇒

Python Module Index

S

Index