# jsoner Documentation

***Release 0.2.0***

**Sebastian Schaffer**

**Mar 29, 2019**

# Contents:

CHAPTER 1

# jsoner

- Free software: MIT license
- Documentation: https://jsoner.readthedocs.io.

*Jsoner* is a package aiming for making conversion to and from json easier.

## 1.1 Installation

### 1.1.1 Stable release

To install jsoner, run this command in your terminal:

```
$ pip install jsoner
```

This is the preferred method to install jsoner, as it will always install the most recent stable release.

### 1.1.2 From sources

The sources for jsoner can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/sschaffer92/jsoner
```

Or download the tarball:

```
$ curl  -OL https://github.com/sschaffer92/jsoner/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

## 1.2 Usage

*Jsoner* builds on the builtin *json* python package. Since you cannot serialize object to json by default it can be useful to have a nice way for doing so. This package provides three different ways to achieve this:

- provide an `to_dict` and `from_dict` method:

```
from jsoner import dumps, loads
class A:
    def __init__(self, a):
        self.a = a

    def to_dict(self) -> dict:
        return {'a': self.a}

    @classmethod
    def from_dict(cls, data: dict) -> 'A':
        return A(**data)

a = A(42)
data = dumps(a)
a = loads(data)
```

- or provide an `to_str` and `from_str` method:

```
from jsoner import dumps, loads
class A:
    def __init__(self, a):
        self.a = a

    def to_str(self) -> str:
        return str(self.a)

    @classmethod
    def from_str(cls, data: str) -> 'A':
        return A(data)

a = A('foo')
data = dumps(a)
a = loads(data)
```

- or implement a conversion function pair (This way is especially useful if you don't have direct access to the class definition):

```
from jsoner import dumps, loads
from jsoner import encoders, decoders
class A:
    def __init__(self, a):
        self.a = a
```

(continues on next page)

(continued from previous page)

```
@encoders.register(A)
def encode_a(a: 'A') -> str:
    return a.a

@decoders.register(A)
def decode_a(data: str) -> str:
    return A(data)

a = A('foo')
data = dumps(a)
a = loads(data)
```

*Jsoner* can also deal with nested objects as long they are also serializable as described above.

### 1.2.1 *Celery* and *Django*

One good use case for the *Jsoner* package is the *Celery* serialization of tasks and task results.

To make *Celery* use *Jsoner* you can apply the following settings:

```
from celery import app
from kombu import serialization

from jsoner import dumps, loads

# register Jsoner
serialization.register('jsoner', dumps, loads, content_type='application/json')

app = Celery('Test')

# tell celery to use Jsoner
app.conf.update(
    accept_content=['jsoner'],
    task_serializer='jsoner',
    result_serializer='jsoner',
    result_backend='rpc'
)

# Celery can now serialize objects which can be serialized by Jsoner.
class A:
    def __init__(self, foo):
        self.foo = foo

    @classmethod
    def from_dict(cls, data: dict) -> 'A':
        return A(**data)

    def to_dict(self):
        return {'foo': self.foo}

a = A('bar')

@app.task
def task(obj: A) -> 'A':
    ...
```

(continues on next page)

(continued from previous page)

```
    return obj

a = task.delay(a).get()
```

This way you can easily serialize django model instances and pass them to the *Celery* task.

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

Then you can just pass the model to the celery task directly:

```
from django.db.models import Model
from jsoner import encoders, decoders

from .models import Person

# Create a conversion function pair which just saved the primary key.
@encoders.register(Model)
def to_primary_key(model: Model) -> int:
    return model.pk

# Load object from the primary key.
@decoders.register(Model)
def from_primary_key(pk: int, model_cls: Model) -> Model:
    return model_cls.objects.get(pk=pk)

p = Person(first_name="Foo", last_name="Bar")
p = task.delay(p).get()
```

Similar you could create a conversion function pair for querysets.

# CHAPTER 2

# API reference

## 2.1 jsoner package

jsoner.**dumps**(*obj*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=<class 'jsoner.serialization.JsonEncoder'>*, *indent=None*, *separators=None*, *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`, `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 `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.

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.

jsoner.**loads**(*s*, ***, *encoding=None*, *cls=None*, *object_hook=<function json_hook>*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

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.

The `encoding` argument is ignored and deprecated.

jsoner.**dump**(*obj*, *fp*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=<class 'jsoner.serialization.JsonEncoder'>*, *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).

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 `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.

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.

jsoner.**load**(*fp*, ***, *cls=None*, *object_hook=<function json_hook>*, *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.

`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.

### 2.1.1 Submodules

### 2.1.2 jsoner.serialization module

**class** jsoner.serialization.**DictConvertible**

Bases: `abc.ABC`

This abstract class implements the `to_dict()` and `from_dict()`. Every class implementing those two methods will be a subclass of *`DictConvertible`*. It is not necessary to inherit from this class.

**class** jsoner.serialization.**JsonEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `json.encoder.JSONEncoder`

JsonEncoder will decode all objects, which implement either *to_dict* and *from_dict* or *to_str* and *from_str*.

---

**Note:** `from_str()` and `from_dict()` must be a `classmethod`. It is enough to implement either `from_str()` and `to_str()` or `from_dict()` and `to_dict()`. If both are implemented, then `from_dict()` and `to_dict()` are preferred.

---

If you do not want to implement methods in your class, or you might have no access to the class definition, you can use *`jsoner.registry.encoders()`* and *`jsoner.registry.decoders()`*.

**default**(*obj*, **args*, ***kwargs*)

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)
```

**class** jsoner.serialization.**JsonerSerializable**

Bases: `abc.ABC`

The *`JsonerSerializable`* serves as an abstract class which indicated if an instance can be serialized by *Jsoner*. Therefore it implements the `__subclasshook__()` method.

An object is serializable by *Jsoner* if it is registered in the encoding-, decoding-registry or if it is convertible to a dict or to a string.

**class** jsoner.serialization.**StrConvertible**
Bases: abc.ABC

This abstract class implements the `to_str()` and `from_str()`. Every class implementing those two methods will be a subclass of *StrConvertible*. It is not necessary to inherit from this class.

jsoner.serialization.**json_hook**(*primitive: Any*) → Any
This hook will try to recreate an object from the data it receives. It it fails to do so, it will just return the original data.

**Parameters** **primitive** –

**Returns**

jsoner.serialization.**maybe_convert_to_obj**(*data: dict*) → Any
This function will try to create an object from the data dictionary.

**Parameters** **data** –

**Returns**

jsoner.serialization.**obj_spec**(*obj_or_type: Union[object, type]*) → str
This function returns the path of the argument class.

If the argument is an instance of `type`, it returns the path of the argument itself.

**Usage::**

```
>>> from jsoner.serialization import obj_spec
>>> class A:
...     pass
```

```
>>> obj_spec(A)  # doctest: +ELLIPSIS
'...A'
```

```
>>> a = A()
>>> obj_spec(a)  # doctest: +ELLIPSIS
'...A'
```

**Parameters** **obj_or_type** –

**Returns**

### 2.1.3 jsoner.registry module

**class** jsoner.registry.**Registry**(***kwargs*)
Bases: collections.UserDict

The *Registry* allows simple key-value mapping. Each key is only allowed once in the registry.

**Usage::**

```
>>> from jsoner.registry import Registry
>>> reg = Registry()
```

```
>>> reg.add('foo', 42)
>>> reg.get('foo')
42
```

```
>>> reg = Registry()
>>> @reg.register('foo')
... def foo():
...     return 42
```

```
>>> reg.get('foo')()
42
```

**add**(*key: Any*, *value: Any*) → None
Adds the key, value pair to the registry. Each key is only allowed once in the registry.

**Parameters**

- **key** –
- **value** –

**Returns**

**Raises** **KeyError** – If the key is already in the registry.

**register**(*key: Any*) → Callable
*register()* servers as a decorator to add functions to the registry.

**Parameters** **key** –

**Returns** Callable

**registry**
returns: The registry dictionary. :rtype: dict

**class** jsoner.registry.**SubclassRegistry**(***kwargs*)
Bases: *jsoner.registry.Registry*

The *SubclassRegistry* will not only map a single key-value pair, but will also retrieve a value if the key, or the type of the key is a Subclass of any of the keys.

If the key, seems to be an object, which could potentially be in the registry but is not found at once, the *SubclassRegistry* will search the mro of the object and check against its entries.

**Usage::**

```
>>> from jsoner.registry import SubclassRegistry
>>> reg = SubclassRegistry()
```

```
>>> reg.add(dict, 42)
>>> reg.get(dict)
42
```

```
>>> class A:
...     pass
>>> class B(A):
...     pass
```

```
>>> reg.add(A, 'bar')
>>> reg.get(B)
'bar'
>>> reg.get(B())
'bar'
```

The registration also works with strings

```
>>> from datetime import datetime
>>> reg.add('datetime.datetime', 'foo')
>>> reg.get(datetime)
'foo'
```

```
>>> reg.get('dict')
42
```

Furthermore it can be used as decorator.

```
>>> reg = SubclassRegistry()
>>> @reg.register(A)
... def foo():
...     return 42
>>> reg.get(A)()
42
>>> reg.get(B)()
42
```

jsoner.registry.**decoders = {}**
*decoders* contains the inverse functions-type mapping for *encoders*.

jsoner.registry.**encoders = {}**
*encoders* contains a mapping of types and encoding functions. An encoding function takes an argument and returns a value which should be json serializable. The function which is defined in *decoders* must be able to recreate the object from the returned value.

jsoner.registry.**import_object**(*path: str*) → Any
Import the object or raise an ImportError if the object is not found.

**Parameters** **path** – The path to the object.

**Returns** The imported object.

**Raises** **ImportError** –

### 2.1.4 jsoner.errors module

**exception** jsoner.errors.**JsonEncodingError**
Bases: *jsoner.errors.JsonerException*

This error occurs if *Jsoner* cannot encode your object to json.

**exception** jsoner.errors.**JsonerException**
Bases: Exception

Base Exception class

CHAPTER 3

# Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

## 3.1 Types of Contributions

### 3.1.1 Report Bugs

Report bugs at https://github.com/sschaffer92/jsoner/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### 3.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

### 3.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

### 3.1.4 Write Documentation

jsoner could always use more documentation, whether as part of the official jsoner docs, in docstrings, or even on the web in blog posts, articles, and such.

### 3.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/sschaffer92/jsoner/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## 3.2 Get Started!

Ready to contribute? Here's how to set up *jsoner* for local development.

1. Fork the *jsoner* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/jsoner.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv jsoner
$ cd jsoner/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 jsoner tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

## 3.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/sschaffer92/jsoner/pull_requests and make sure that the tests pass for all supported Python versions.

## 3.4 Tips

To run a subset of tests:

```
$ py.test tests.test_jsoner
```

## 3.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 4

# Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

## 4.1 Development Lead

- Sebastian Schaffer

## 4.2 Contributors

None yet. Why not be the first?

CHAPTER 5

# History

## 5.1 0.1.0 (2019-02-18)

- First release on PyPI.

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## j

# Index