json-tricks Documentation

Release 1.2

Mark

Jul 12, 2018

Contents

The *pyjson-tricks* package brings several pieces of functionality to python handling of json files:

1. **Store and load numpy arrays** in human-readable format.
2. **Store and load class instances** both generic and customized.
3. **Store and load date/times** as a dictionary (including timezone).
4. **Preserve map order** `{}` using `OrderedDict`.
5. **Allow for comments** in json files by starting lines with `#`.
6. Sets, complex numbers, Decimal, Fraction, enums, compression, duplicate keys, ...

As well as compression and disallowing duplicate keys.

- Code: https://github.com/mverleg/pyjson_tricks
- Documentation: http://json-tricks.readthedocs.org/en/latest/
- PIP: https://pypi.python.org/pypi/json_tricks

The 2.0 series added some of the above features and broke backward compatibility. The version 3.0 series is a more readable rewrite that also makes it easier to combine encoders, again not fully backward compatible.

Several keys of the format `__keyname__` have special meanings, and more might be added in future releases.

If you're considering JSON-but-with-comments as a config file format, have a look at HJSON, it might be more appropriate. For other purposes, keep reading!

Thanks for all the Github stars!

CHAPTER 1

Installation and use

You can install using

```
pip install json-tricks  # or e.g. 'json-tricks<3.0' for older versions
```

Decoding of some data types needs the corresponding package to be installed, e.g. `numpy` for arrays, `pandas` for dataframes and `pytz` for timezone-aware datetimes.

You can import the usual json functions dump(s) and load(s), as well as a separate comment removal function, as follows:

```
from json_tricks import dump, dumps, load, loads, strip_comments
```

The exact signatures of these and other functions are in the documentation.

`json-tricks` supports Python 2.7, and Python 3.4 and later, and is automatically tested on 2.7, 3.4, 3.5 and 3.6. Pypy is supported without numpy and pandas. Pandas doesn't support 3.4.

CHAPTER 2

Preserve type vs use primitive

By default, types are encoded such that they can be restored to their original type when loaded with `json-tricks`. Example encodings in this documentation refer to that format.

You can also choose to store things as their closest primitive type (e.g. arrays and sets as lists, decimals as floats). This may be desirable if you don't care about the exact type, or you are loading the json in another language (which doesn't restore python types). It's also smaller.

To forego meta data and store primitives instead, pass `primitives` to `dump(s)`. This is available in version `3.8` and later. Example:

```
data = [
        arange(0, 10, 1, dtype=int).reshape((2, 5)),
        datetime(year=2017, month=1, day=19, hour=23, minute=00, second=00),
        1 + 2j,
        Decimal(42),
        Fraction(1, 3),
        MyTestCls(s='ub', dct={'7': 7}),  # see later
        set(range(7)),
]
# Encode with metadata to preserve types when decoding
print(dumps(data))
```

```
// (comments added and indenting changed)
[
        // numpy array
        {
                "__ndarray__": [
                        [0, 1, 2, 3, 4],
                        [5, 6, 7, 8, 9]],
                "dtype": "int64",
                "shape": [2, 5],
                "Corder": true
        },
        // datetime (naive)
```

(continues on next page)

(continued from previous page)

```
        {
                "__datetime__": null,
                "year": 2017,
                "month": 1,
                "day": 19,
                "hour": 23
        },
        // complex number
        {
                "__complex__": [1.0, 2.0]
        },
        // decimal & fraction
        {
                "__decimal__": "42"
        },
        {
                "__fraction__": true
                "numerator": 1,
                "denominator": 3,
        },
        // class instance
        {
                "__instance_type__": [
                  "tests.test_class",
                  "MyTestCls"
                ],
                "attributes": {
                  "s": "ub",
                  "dct": {"7": 7}
                }
        },
        // set
        {
                "__set__": [0, 1, 2, 3, 4, 5, 6]
        }
]
```

```
# Encode as primitive types; more simple but loses type information
print(dumps(data, primitives=True))
```

```
// (comments added and indentation changed)
[
        // numpy array
        [[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]],
        // datetime (naive)
        "2017-01-19T23:00:00",
        // complex number
        [1.0, 2.0],
        // decimal & fraction
        42.0,
        0.3333333333333333,
        // class instance
        {
                "s": "ub",
                "dct": {"7": 7}
```

(continues on next page)

(continued from previous page)

```
        },
        // set
        [0, 1, 2, 3, 4, 5, 6]
]
```

Note that valid json is produced either way: `json-tricks` stores meta data as normal json, but other packages probably won't interpret it.

CHAPTER 3

Features

3.1 Numpy arrays

The array is encoded in sort-of-readable and very flexible and portable format, like so:

```
arr = arange(0, 10, 1, dtype=uint8).reshape((2, 5))
print(dumps({'mydata': arr}))
```

this yields:

```
{
        "mydata": {
                "dtype": "uint8",
                "shape": [2, 5],
                "Corder": true,
                "__ndarray__": [[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]
        }
}
```

which will be converted back to a numpy array when using `json_tricks.loads`. Note that the memory order (`Corder`) is only stored in v3.1 and later and for arrays with at least 2 dimensions.

As you've seen, this uses the magic key `__ndarray__`. Don't use `__ndarray__` as a dictionary key unless you're trying to make a numpy array (and know what you're doing).

Numpy scalars are also serialized (v3.5+). They are represented by the closest python primitive type. A special representation was not feasible, because Python's json implementation serializes some numpy types as primitives, without consulting custom encoders. If you want to preverse the exact numpy type, use encode_scalars_inplace.

Performance: this method has slow write times similar to other human-readable formats, although read time is worse than csv. File size (with compression) is high on a relative scale, but it's only around 30% above binary. See this benchmark (it's called JSONGzip). A binary alternative might be added, but is not yet available.

This implementation is inspired by an answer by tlausch on stackoverflow that you could read for details.

3.2 Class instances

`json_tricks` can serialize class instances.

If the class behaves normally (not generated dynamic, no `__new__` or `__metaclass__` magic, etc) *and* all it's attributes are serializable, then this should work by default.

```
# json_tricks/test_class.py
class MyTestCls:
        def __init__(self, **kwargs):
                for k, v in kwargs.items():
                        setattr(self, k, v)

cls_instance = MyTestCls(s='ub', dct={'7': 7})

json = dumps(cls_instance, indent=4)
cls_instance_again = loads(json)
```

You'll get your instance back. Here the json looks like this:

```
{
        "__instance_type__": [
                "json_tricks.test_class",
                "MyTestCls"
        ],
        "attributes": {
                "s": "ub",
                "dct": {
                        "7": 7
                }
        }
}
```

As you can see, this stores the module and class name. The class must be importable from the same module when decoding (and should not have changed). If it isn't, you have to manually provide a dictionary to `cls_lookup_map` when loading in which the class name can be looked up. Note that if the class is imported, then `globals()` is such a dictionary (so try `loads(json, cls_lookup_map=glboals())`). Also note that if the class is defined in the 'top' script (that you're calling directly), then this isn't a module and the import part cannot be extracted. Only the class name will be stored; it can then only be deserialized in the same script, or if you provide `cls_lookup_map`.

Note that this also works with `slots` without having to do anything (thanks to `koffie`), which encodes like this (custom indentation):

```
{
        "__instance_type__": ["module.path", "ClassName"],
        "slots": {"slotattr": 37},
        "attributes": {"dictattr": 42}
}
```

If the instance doesn't serialize automatically, or if you want custom behaviour, then you can implement `__json__encode__(self)` and `__json_decode__(self, **attributes)` methods, like so:

```
class CustomEncodeCls:
        def __init__(self):
                self.relevant = 42
                self.irrelevant = 37
```

(continues on next page)

(continued from previous page)

```
        def __json_encode__(self):
                # should return primitive, serializable types like dict, list, int, string, float...
                return {'relevant': self.relevant}

        def __json_decode__(self, **attrs):
                # should initialize all properties; note that __init__ is not called implicitly
                self.relevant = attrs['relevant']
                self.irrelevant = 12
```

As you've seen, this uses the magic key `__instance_type__`. Don't use `__instance_type__` as a dictionary key unless you know what you're doing.

3.3 Date, time, datetime and timedelta

Date, time, datetime and timedelta objects are stored as dictionaries of "day", "hour", "millisecond" etc keys, for each nonzero property.

Timezone name is also stored in case it is set. You'll need to have `pytz` installed to use timezone-aware date/times, it's not needed for naive date/times.

```
{
        "__datetime__": null,
        "year": 1988,
        "month": 3,
        "day": 15,
        "hour": 8,
        "minute": 3,
        "second": 59,
        "microsecond": 7,
        "tzinfo": "Europe/Amsterdam"
}
```

This approach was chosen over timestamps for readability and consistency between date and time, and over a single string to prevent parsing problems and reduce dependencies. Note that if `primitives=True`, date/times are encoded as ISO 8601, but they won't be restored automatically.

Don't use `__date__`, `__time__`, `__datetime__`, `__timedelta__` or `__tzinfo__` as dictionary keys unless you know what you're doing, as they have special meaning.

3.4 Order

Given an ordered dictionary like this (see the tests for a longer one):

```
ordered = OrderedDict((
        ('elephant', None),
        ('chicken', None),
        ('tortoise', None),
))
```

Converting to json and back will preserve the order:

```
from json_tricks import dumps, loads
json = dumps(ordered)
ordered = loads(json, preserve_order=True)
```

where `preserve_order=True` is added for emphasis; it can be left out since it's the default.

As a note on performance, both dicts and OrderedDicts have the same scaling for getting and setting items (`O(1)`). In Python versions before 3.5, OrderedDicts were implemented in Python rather than C, so were somewhat slower; since Python 3.5 both are implemented in C. In summary, you should have no scaling problems and probably no performance problems at all, especially for 3.5 and later. Python 3.6+ preserve order of dictionaries by default making this redundant, but this is an implementation detail that should not be relied on.

3.5 Comments

This package uses `#` and `//` for comments, which seem to be the most common conventions, though only the latter is valid javascript.

For example, you could call `loads` on the following string:

```
{ # "comment 1
        "hello": "Wor#d", "Bye": "\"M#rk\"", "yes\\\"": 5,# comment" 2
        "quote": "\"th#t's\" what she said", // comment "3"
        "list": [1, 1, "#", "\"", "\\", 8], "dict": {"q": 7} #" comment 4 with quotes
}
// comment 5
```

And it would return the de-commented version:

```
{
        "hello": "Wor#d", "Bye": "\"M#rk\"", "yes\\\"": 5,
        "quote": "\"th#t's\" what she said",
        "list": [1, 1, "#", "\"", "\\", 8], "dict": {"q": 7}
}
```

Since comments aren't stored in the Python representation of the data, loading and then saving a json file will remove the comments (it also likely changes the indentation).

The implementation of comments is not particularly efficient, but it does handle all the special cases I could think of. For a few files you shouldn't notice any performance problems, but if you're reading hundreds of files, then they are presumably computer-generated, and you could consider turning comments off (`ignore_comments=False`).

3.6 Other features

- Sets are serializable and can be loaded. By default the set json representation is sorted, to have a consistent representation.
- Save and load complex numbers (version 3.2) with `1+2j` serializing as `{'__complex__': [1, 2]}`.
- Save and load `Decimal` and `Fraction` (including NaN, infinity, -0 for Decimal).
- Save and load `Enum` (thanks to `Jenselme`), either built-in in python3.4+, or with the enum34 package in earlier versions. `IntEnum` needs encode_intenums_inplace.
- `json_tricks` allows for gzip compression using the `compression=True` argument (off by default).

- `json_tricks` can check for duplicate keys in maps by setting `allow_duplicates` to False. These are kind of allowed, but are handled inconsistently between json implementations. In Python, for `dict` and `OrderedDict`, duplicate keys are silently overwritten.

CHAPTER 4

Usage & contributions

Code is under Revised BSD License so you can use it for most purposes including commercially.

Contributions are very welcome! Bug reports, feature suggestions and code contributions help this project become more useful for everyone! There is a short contribution guide.

CHAPTER 5

Tests

Tests are run automatically for commits to the repository for all supported versions. This is the status: To run the tests manually for your version, see this guide.

CHAPTER 6

Main components

Note that these functions exist as two versions, the full version with numpy (np) and the version without requirements (nonp) that doesn't do nunpy encoding/decoding.

If you import these functions directly from json_tricks, e.g. `from json_tricks import dumps`, then it will select np if numpy is available, and nonp otherwise. You can use `json_tricks.NUMPY_MODE` to see if numpy mode is being used.

This dual behaviour can lead to confusion, so it is recommended that you import directly from np or nonp.

6.1 dumps

`json_tricks.nonp.`**`dumps`**(*obj, sort_keys=None, cls=<class 'json_tricks.encoders.TricksEncoder'>, obj_encoders=[<function pandas_encode>, <function numpy_encode>, <function enum_instance_encode>, <function json_date_time_encode>, <function json_complex_encode>, <function json_set_encode>, <function numeric_types_encode>, <function class_instance_encode>], extra_obj_encoders=(), primitives=False, compression=None, allow_nan=False, conv_str_byte=False, fallback_encoders=(), **jsonkwargs*)

Convert a nested data structure to a json string.

Parameters

- **`obj`** – The Python object to convert.
- **`sort_keys`** – Keep this False if you want order to be preserved.
- **`cls`** – The json encoder class to use, defaults to NoNumpyEncoder which gives a warning for numpy arrays.
- **`obj_encoders`** – Iterable of encoders to use to convert arbitrary objects into json-able promitives.
- **`extra_obj_encoders`** – Like *obj_encoders* but on top of them: use this to add encoders without replacing defaults. Since v3.5 these happen before default encoders.

- **fallback_encoders** – These are extra *obj_encoders* that 1) are ran after all others and 2) only run if the object hasn't yet been changed.
- **allow_nan** – Allow NaN and Infinity values, which is a (useful) violation of the JSON standard (default False).
- **conv_str_byte** – Try to automatically convert between strings and bytes (assuming utf-8) (default False).

Returns The string containing the json-encoded version of obj.

Other arguments are passed on to *cls*. Note that *sort_keys* should be false if you want to preserve order.

json_tricks.np.**dumps**(*obj, sort_keys=None, cls=<class 'json_tricks.encoders.TricksEncoder'>, obj_encoders=[<function pandas_encode>, <function numpy_encode>, <function enum_instance_encode>, <function json_date_time_encode>, <function json_complex_encode>, <function json_set_encode>, <function numeric_types_encode>, <function class_instance_encode>], extra_obj_encoders=(), primitives=False, compression=None, allow_nan=False, conv_str_byte=False, fallback_encoders=(), **jsonkwargs*)

Convert a nested data structure to a json string.

Parameters

- **obj** – The Python object to convert.
- **sort_keys** – Keep this False if you want order to be preserved.
- **cls** – The json encoder class to use, defaults to NoNumpyEncoder which gives a warning for numpy arrays.
- **obj_encoders** – Iterable of encoders to use to convert arbitrary objects into json-able promitives.
- **extra_obj_encoders** – Like *obj_encoders* but on top of them: use this to add encoders without replacing defaults. Since v3.5 these happen before default encoders.
- **fallback_encoders** – These are extra *obj_encoders* that 1) are ran after all others and 2) only run if the object hasn't yet been changed.
- **allow_nan** – Allow NaN and Infinity values, which is a (useful) violation of the JSON standard (default False).
- **conv_str_byte** – Try to automatically convert between strings and bytes (assuming utf-8) (default False).

Returns The string containing the json-encoded version of obj.

Other arguments are passed on to *cls*. Note that *sort_keys* should be false if you want to preserve order.

6.2 dump

json_tricks.nonp.**dump**(*obj, fp, sort_keys=None, cls=<class 'json_tricks.encoders.TricksEncoder'>, obj_encoders=[<function pandas_encode>, <function numpy_encode>, <function enum_instance_encode>, <function json_date_time_encode>, <function json_complex_encode>, <function json_set_encode>, <function numeric_types_encode>, <function class_instance_encode>], extra_obj_encoders=(), primitives=False, compression=None, force_flush=False, allow_nan=False, conv_str_byte=False, fallback_encoders=(), **jsonkwargs*)

Convert a nested data structure to a json string.

Parameters

- **fp** – File handle or path to write to.
- **compression** – The gzip compression level, or None for no compression.
- **force_flush** – If True, flush the file handle used, when possibly also in the operating system (default False).

The other arguments are identical to *dumps*.

json_tricks.np.**dump**(*obj, fp, sort_keys=None, cls=<class 'json_tricks.encoders.TricksEncoder'>, obj_encoders=[<function pandas_encode>, <function numpy_encode>, <function enum_instance_encode>, <function json_date_time_encode>, <function json_complex_encode>, <function json_set_encode>, <function numeric_types_encode>, <function class_instance_encode>], extra_obj_encoders=(), primitives=False, compression=None, force_flush=False, allow_nan=False, conv_str_byte=False, fallback_encoders=(), **jsonkwargs*)

Convert a nested data structure to a json string.

Parameters

- **fp** – File handle or path to write to.
- **compression** – The gzip compression level, or None for no compression.
- **force_flush** – If True, flush the file handle used, when possibly also in the operating system (default False).

The other arguments are identical to *dumps*.

6.3 loads

json_tricks.nonp.**loads**(*string, preserve_order=True, ignore_comments=True, decompression=None, obj_pairs_hooks=[<function pandas_hook>, <function json_numpy_obj_hook>, <json_tricks.decoders.EnumInstanceHook object>, <function json_date_time_hook>, <function json_complex_hook>, <function json_set_hook>, <function numeric_types_hook>, <json_tricks.decoders.ClassInstanceHook object>], extra_obj_pairs_hooks=(), cls_lookup_map=None, allow_duplicates=True, conv_str_byte=False, **jsonkwargs*)

Convert a nested data structure to a json string.

Parameters

- **string** – The string containing a json encoded data structure.
- **decode_cls_instances** – True to attempt to decode class instances (requires the environment to be similar the the encoding one).
- **preserve_order** – Whether to preserve order by using OrderedDicts or not.
- **ignore_comments** – Remove comments (starting with # or //).
- **decompression** – True to use gzip decompression, False to use raw data, None to automatically determine (default). Assumes utf-8 encoding!
- **obj_pairs_hooks** – A list of dictionary hooks to apply.
- **extra_obj_pairs_hooks** – Like *obj_pairs_hooks* but on top of them: use this to add hooks without replacing defaults. Since v3.5 these happen before default hooks.

- **cls_lookup_map** – If set to a dict, for example `globals()`, then classes encoded from __main__ are looked up this dict.
- **allow_duplicates** – If set to False, an error will be raised when loading a json-map that contains duplicate keys.
- **parse_float** – A function to parse strings to integers (e.g. Decimal). There is also *parse_int*.
- **conv_str_byte** – Try to automatically convert between strings and bytes (assuming utf-8) (default False).

Returns The string containing the json-encoded version of obj.

Other arguments are passed on to json_func.

json_tricks.np.**loads**(*string, preserve_order=True, ignore_comments=True, decompression=None, obj_pairs_hooks=[<function pandas_hook>, <function json_numpy_obj_hook>, <json_tricks.decoders.EnumInstanceHook object>, <function json_date_time_hook>, <function json_complex_hook>, <function json_set_hook>, <function numeric_types_hook>, <json_tricks.decoders.ClassInstanceHook object>], extra_obj_pairs_hooks=(), cls_lookup_map=None, allow_duplicates=True, conv_str_byte=False, **jsonkwargs*)

Convert a nested data structure to a json string.

Parameters

- **string** – The string containing a json encoded data structure.
- **decode_cls_instances** – True to attempt to decode class instances (requires the environment to be similar the the encoding one).
- **preserve_order** – Whether to preserve order by using OrderedDicts or not.
- **ignore_comments** – Remove comments (starting with # or //).
- **decompression** – True to use gzip decompression, False to use raw data, None to automatically determine (default). Assumes utf-8 encoding!
- **obj_pairs_hooks** – A list of dictionary hooks to apply.
- **extra_obj_pairs_hooks** – Like *obj_pairs_hooks* but on top of them: use this to add hooks without replacing defaults. Since v3.5 these happen before default hooks.
- **cls_lookup_map** – If set to a dict, for example `globals()`, then classes encoded from __main__ are looked up this dict.
- **allow_duplicates** – If set to False, an error will be raised when loading a json-map that contains duplicate keys.
- **parse_float** – A function to parse strings to integers (e.g. Decimal). There is also *parse_int*.
- **conv_str_byte** – Try to automatically convert between strings and bytes (assuming utf-8) (default False).

Returns The string containing the json-encoded version of obj.

Other arguments are passed on to json_func.

6.4 load

json_tricks.nonp.**load**(*fp, preserve_order=True, ignore_comments=True, decompression=None, obj_pairs_hooks=[<function pandas_hook>, <function json_numpy_obj_hook>, <json_tricks.decoders.EnumInstanceHook object>, <function json_date_time_hook>, <function json_complex_hook>, <function json_set_hook>, <function numeric_types_hook>, <json_tricks.decoders.ClassInstanceHook object>], extra_obj_pairs_hooks=(), cls_lookup_map=None, allow_duplicates=True, conv_str_byte=False, **jsonkwargs*)

Convert a nested data structure to a json string.

Parameters **fp** – File handle or path to load from.

The other arguments are identical to loads.

json_tricks.np.**load**(*fp, preserve_order=True, ignore_comments=True, decompression=None, obj_pairs_hooks=[<function pandas_hook>, <function json_numpy_obj_hook>, <json_tricks.decoders.EnumInstanceHook object>, <function json_date_time_hook>, <function json_complex_hook>, <function json_set_hook>, <function numeric_types_hook>, <json_tricks.decoders.ClassInstanceHook object>], extra_obj_pairs_hooks=(), cls_lookup_map=None, allow_duplicates=True, conv_str_byte=False, **jsonkwargs*)

Convert a nested data structure to a json string.

Parameters **fp** – File handle or path to load from.

The other arguments are identical to loads.

CHAPTER 7

Utilities

7.1 strip comments

json_tricks.comment.**strip_comments**(*string*, *comment_symbols=frozenset(['//', '#'])*)

Parameters

- **string** – A string containing json with comments started by comment_symbols.
- **comment_symbols** – Iterable of symbols that start a line comment (default # or //).

Returns The string with the comments removed.

7.2 numpy

json_tricks.np.**numpy_encode**(*obj*, *primitives=False*)

Encodes numpy ‘ndarray‘s as lists with meta data.

Encodes numpy scalar types as Python equivalents. Special encoding is not possible, because int64 (in py2) and float64 (in py2 and py3) are subclasses of primitives, which never reach the encoder.

Parameters **primitives** – If True, arrays are serialized as (nested) lists without meta info.

json_tricks.np.**json_numpy_obj_hook**(*dct*)

Replace any numpy arrays previously encoded by NumpyEncoder to their proper shape, data type and data.

Parameters **dct** – (dict) json encoded ndarray

Returns (ndarray) if input was an encoded ndarray

7.3 class instances

json_tricks.encoders.**class_instance_encode**(*obj*, *primitives=False*)
Encodes a class instance to json. Note that it can only be recovered if the environment allows the class to be imported in the same way.

class json_tricks.decoders.**ClassInstanceHook**(*cls_lookup_map=None*)
This hook tries to convert json encoded by class_instance_encoder back to it's original instance. It only works if the environment is the same, e.g. the class is similarly importable and hasn't changed.

7.4 enum instances

Support for enums was added in Python 3.4. Support for previous versions of Python is available with the enum 34 package.

json_tricks.encoders.**enum_instance_encode**(*obj*, *primitives=False*, *with_enum_value=False*)
Encodes an enum instance to json. Note that it can only be recovered if the environment allows the enum to be imported in the same way. :param primitives: If true, encode the enum values as primitive (more readable, but cannot be restored automatically). :param with_enum_value: If true, the value of the enum is also exported (it is not used during import, as it should be constant).

class json_tricks.decoders.**EnumInstanceHook**(*cls_lookup_map=None*)
This hook tries to convert json encoded by enum_instance_encode back to it's original instance. It only works if the environment is the same, e.g. the enum is similarly importable and hasn't changed.

By default `IntEnum` cannot be encoded as enums since they cannot be differenciated from integers. To serialize them, you must use *encode_intenums_inplace* which mutates a nested data structure (in place!) to replace any `IntEnum` by their representation. If you serialize this result, it can subsequently be loaded without further adaptations.

json_tricks.utils.**encode_intenums_inplace**(*obj*)
Searches a data structure of lists, tuples and dicts for IntEnum and replaces them by their dictionary representation, which can be loaded by json-tricks. This happens in-place (the object is changed, use a copy).

7.5 date/time

json_tricks.encoders.**json_date_time_encode**(*obj*, *primitives=False*)
Encode a date, time, datetime or timedelta to a string of a json dictionary, including optional timezone.

Parameters **obj** – date/time/datetime/timedelta obj

Returns (dict) json primitives representation of date, time, datetime or timedelta

json_tricks.decoders.**json_date_time_hook**(*dct*)
Return an encoded date, time, datetime or timedelta to it's python representation, including optional timezone.

Parameters **dct** – (dict) json encoded date, time, datetime or timedelta

Returns (date/time/datetime/timedelta obj) python representation of the above

7.6 numpy scalars

It's not possible (without a lot of hacks) to encode numpy scalars. This is the case because some numpy scalars (*float64*, and depending on Python version also *int64*) are subclasses of *float* and *int*. This means that the Python json

encoder will stringify them without them ever reaching the custom encoders.

So if you really want to encode numpy scalars, you'll have to do the conversion beforehand. For that purpose you can use *encode_scalars_inplace*, which mutates a nested data structure (in place!) to replace any numpy scalars by their representation. If you serialize this result, it can subsequently be loaded without further adaptations.

It's not great, but unless the Python json module changes, it's the best that can be done. See issue 18 for more details.

`json_tricks.np_utils.`**`encode_scalars_inplace`**(*obj*)

Searches a data structure of lists, tuples and dicts for numpy scalars and replaces them by their dictionary representation, which can be loaded by json-tricks. This happens in-place (the object is changed, use a copy).

CHAPTER 8

Running tests

There are many test environments: with and without pandas, numpy or timezone support, and for each of the supported Python versions. You will need all the Python versions installed, as well as a number of packages available through pip. You can just install `detox` (`pip install detox`), and others will be installed automatically as dependencies or during tests (like `numpy`, `pandas`, `pytz`, `pytest`, `pytest-cov` and `tox`).

To run all of these tests at once, simply run `detox` from the main directory. It usually takes roughly half a minute, but the first time takes longer because packages need to be installed.

To get coverage information from all these configurations, you first need to combine them using `coverage combine .tox/coverage/*` (once after each `detox`). You can then show results normally, e.g. `coverage report`. It should be about 90%.

If you want to show results in IntelliJ PyCharm with lines highlighted etc, you need several steps. First generate an XML-report with `coverage xml`. Then the paths must be made to start at the root of the project, or be absolute, which can be done using `sed 's/filename="\([^"]*\)"/filename="pyjson_tricks\/\1"/g' coverage.xml` or manually using find and replace. Finally, you can load the report using `Tools > Show code coverage data` and use the green +.

CHAPTER 9

Table of content

This is a simple module so the documentation is single-page.

Index