jsonpickle Documentation

Release 0.9.2

John Paulett

July 22, 2015

Contents

jsonpickle is a Python library for serialization and deserialization of complex Python objects to and from JSON. The standard Python libraries for encoding Python into JSON, such as the stdlib's json, simplejson, and demjson, can only handle Python primitives that have a direct JSON equivalent (e.g. dicts, lists, strings, ints, etc.). jsonpickle builds on top of these libraries and allows more complex data structures to be serialized to JSON. jsonpickle is highly configurable and extendable–allowing the user to choose the JSON backend and add additional backends.

Contents

CHAPTER 1

jsonpickle Usage

Python library for serializing any arbitrary object graph into JSON.

jsonpickle can take almost any Python object and turn the object into JSON. Additionally, it can reconstitute the object back into Python.

The object must be accessible globally via a module and must inherit from object (AKA new-style classes).

Create an object:

```
class Thing(object):
    def __init__(self, name):
        self.name = name

obj = Thing('Awesome')
```

Use jsonpickle to transform the object into a JSON string:

```
import jsonpickle
frozen = jsonpickle.encode(obj)
```

Use jsonpickle to recreate a Python object from a JSON string:

```
thawed = jsonpickle.decode(frozen)
```

Warning: Loading a JSON string from an untrusted source represents a potential security vulnerability. jsonpickle makes no attempt to sanitize the input.

The new object has the same type and data, but essentially is now a copy of the original.

```
assert obj.name == thawed.name
```

If you will never need to load (regenerate the Python class from JSON), you can pass in the keyword unpicklable=False to prevent extra information from being added to JSON:

```
oneway = jsonpickle.encode(obj, unpicklable=False)
result = jsonpickle.decode(oneway)
assert obj.name == result['name'] == 'Awesome'
```

CHAPTER 2

Download & Install

The easiest way to get jsonpickle is via PyPi with pip:

```
$ pip install -U jsonpickle
```

For Python 2.6+, jsonpickle has no required dependencies (it uses the standard library's `json` module by default). For Python 2.5 or earlier, you must install a supported JSON backend (including simplejson or demjson). For example:

```
$ pip install simplejson
```

You can also download or *checkout* the latest code and install from source:

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

CHAPTER 3

API Reference

3.1 jsonpickle API

Contents

3.1.1 `jsonpickle` – High Level API

`jsonpickle.`**`encode`**(*value*, *unpicklable=True*, *make_refs=True*, *keys=False*, *max_depth=None*, *backend=None*, *warn=False*, *max_iter=None*)

Return a JSON formatted representation of value, a Python object.

Parameters

- **`unpicklable`** – If set to False then the output will not contain the information necessary to turn the JSON data back into Python objects, but a simpler JSON stream is produced.
- **`max_depth`** – If set to a non-negative integer then jsonpickle will not recurse deeper than 'max_depth' steps into the object. Anything deeper than 'max_depth' is represented using a Python repr() of the object.
- **`make_refs`** – If set to False jsonpickle's referencing support is disabled. Objects that are id()-identical won't be preserved across encode()/decode(), but the resulting JSON stream will be conceptually simpler. jsonpickle detects cyclical objects and will break the cycle by calling repr() instead of recursing when make_refs is set False.
- **`keys`** – If set to True then jsonpickle will encode non-string dictionary keys instead of coercing them into strings via *repr()*.

- **warn** – If set to True then jsonpickle will warn when it returns None for an object which it cannot pickle (e.g. file descriptors).
- **max_iter** – If set to a non-negative integer then jsonpickle will consume at most *max_iter* items when pickling iterators.

```
>>> encode('my string')
'"my string"'
>>> encode(36)
'36'
```

```
>>> encode({'foo': True})
'{"foo": true}'
```

```
>>> encode({'foo': True}, max_depth=0)
'"{\'foo\': True}"'
```

```
>>> encode({'foo': True}, max_depth=1)
'{"foo": "True"}'
```

jsonpickle.**decode**(*string*, *backend=None*, *keys=False*)
Convert a JSON string into a Python object.

The keyword argument 'keys' defaults to False. If set to True then jsonpickle will decode non-string dictionary keys into python objects via the jsonpickle protocol.

```
>>> str(decode('"my string"'))
'my string'
>>> decode('36')
36
```

Choosing and Loading Backends

jsonpickle allows the user to specify what JSON backend to use when encoding and decoding. By default, jsonpickle will try to use, in the following order: simplejson, json, and demjson. The prefered backend can be set via *jsonpickle.set_preferred_backend()*. Additional JSON backends can be used via *jsonpickle.load_backend()*.

For example, users of Django can use the version of simplejson that is bundled in Django:

```
jsonpickle.load_backend('django.util.simplejson', 'dumps', 'loads', ValueError))
jsonpickle.set_preferred_backend('django.util.simplejson')
```

Supported backends:

- json in Python 2.6+
- simplejson
- demjson

Experimental backends:

- jsonlib
- yajl via py-yajl
- ujson

jsonpickle.**set_preferred_backend**(*self*, *name*)
Set the preferred json backend.

If a preferred backend is set then jsonpickle tries to use it before any other backend.

For example:

```
set_preferred_backend('simplejson')
```

If the backend is not one of the built-in jsonpickle backends (json/simplejson, or demjson) then you must load the backend prior to calling set_preferred_backend.

AssertionError is raised if the backend has not been loaded.

jsonpickle.**load_backend**(*self*, *name*, *dumps='dumps'*, *loads='loads'*, *loads_exc=<type 'exceptions.ValueError'>*)

Load a JSON backend by name.

This method loads a backend and sets up references to that backend's loads/dumps functions and exception classes.

Parameters

- **dumps** – is the name of the backend's encode method. The method should take an object and return a string. Defaults to 'dumps'.
- **loads** – names the backend's method for the reverse operation – returning a Python object from a string.
- **loads_exc** – can be either the name of the exception class used to denote decoding errors, or it can be a direct reference to the appropriate exception class itself. If it is a name, then the assumption is that an exception class of that name can be found in the backend module's namespace.
- **load** – names the backend's 'load' method.
- **dump** – names the backend's 'dump' method.

Rtype bool True on success, False if the backend could not be loaded.

jsonpickle.**remove_backend**(*self*, *name*)

Remove all entries for a particular backend.

jsonpickle.**set_encoder_options**(*self*, *name*, **args*, ***kwargs*)

Associate encoder-specific options with an encoder.

After calling set_encoder_options, any calls to jsonpickle's encode method will pass the supplied args and kwargs along to the appropriate backend's encode method.

For example:

```
set_encoder_options('simplejson', sort_keys=True, indent=4)
set_encoder_options('demjson', compactly=False)
```

See the appropriate encoder's documentation for details about the supported arguments and keyword arguments.

Customizing JSON output

jsonpickle supports the standard pickle *__getstate__* and *__setstate__* protocol for representating object instances.

object.**__getstate__**()

Classes can further influence how their instances are pickled; if the class defines the method *__getstate__()*, it is called and the return state is pickled as the contents for the instance, instead of the contents of the instance's dictionary. If there is no *__getstate__()* method, the instance's __dict__ is pickled.

object.**__setstate__**(*state*)
Upon unpickling, if the class also defines the method *__setstate__()*, it is called with the unpickled state. If there is no *__setstate__()* method, the pickled state must be a dictionary and its items are assigned to the new instance's dictionary. If a class defines both *__getstate__()* and *__setstate__()*, the state object needn't be a dictionary and these methods can do what they want.

jsonpickle.handlers – Custom Serialization Handlers

The *jsonpickle.handlers* module allows plugging in custom serialization handlers at run-time. This feature is useful when jsonpickle is unable to serialize objects that are not under your direct control. Custom handlers may be created to handle other objects. Each custom handler must derive from *jsonpickle.handlers.BaseHandler* and implement `flatten` and `restore`.

A handler can be bound to other types by calling `jsonpickle.handlers.register()`.

`jsonpickle.customhandlers.SimpleReduceHandler` is suitable for handling objects that implement the reduce protocol:

```
from jsonpickle import handlers

class MyCustomObject(handlers.BaseHandler):
    ...

    def __reduce__(self):
        return MyCustomObject, self._get_args()

handlers.register(MyCustomObject, handlers.SimpleReduceHandler)
```

class jsonpickle.handlers.**BaseHandler**(*context*)

flatten(*obj*, *data*)
Flatten *obj* into a json-friendly form and write result to *data*.

Parameters

- **obj** (*object*) – The object to be serialized.
- **data** (*dict*) – A partially filled dictionary which will contain the json-friendly representation of *obj* once this method has finished.

classmethod handles(*cls*)
Register this handler for the given class. Suitable as a decorator, e.g.:

```
@SimpleReduceHandler.handles
class MyCustomClass:
    def __reduce__(self):
        ...
```

restore(*obj*)
Restore an object of the registered type from the json-friendly representation *obj* and return it.

class jsonpickle.handlers.**CloneFactory**(*exemplar*)
Serialization proxy for collections.defaultdict's default_factory

class jsonpickle.handlers.**DatetimeHandler**(*context*)
Custom handler for datetime objects

Datetime objects use __reduce__, and they generate binary strings encoding the payload. This handler encodes that payload to reconstruct the object.

flatten(*obj*, *data*)

restore(*data*)

class jsonpickle.handlers.**OrderedDictReduceHandler**(*context*)
Serialize OrderedDict on Python 3.4+

Python 3.4+ returns multiple entries in an OrderedDict's reduced form. Previous versions return a two-item tuple. OrderedDictReduceHandler makes the formats compatible.

flatten(*obj*, *data*)

class jsonpickle.handlers.**QueueHandler**(*context*)
Opaquely serializes Queue objects

Queues contains mutex and condition variables which cannot be serialized. Construct a new Queue instance when restoring.

flatten(*obj*, *data*)

restore(*data*)

class jsonpickle.handlers.**RegexHandler**(*context*)
Flatten _sre.SRE_Pattern (compiled regex) objects

flatten(*obj*, *data*)

restore(*data*)

class jsonpickle.handlers.**Registry**

get(*cls_or_name*, *default=None*)

Parameters

- **cls_or_name** – the type or its fully qualified name
- **default** – default value, if a matching handler is not found

Looks up a handler by type reference or its fully qualified name. If a direct match is not found, the search is performed over all handlers registered with base=True.

register(*cls*, *handler=None*, *base=False*)
Register the a custom handler for a class

Parameters

- **cls** – The custom object class to handle
- **handler** – The custom handler class (if None, a decorator wrapper is returned)
- **base** – Indicates whether the handler should be registered for all subclasses

This function can be also used as a decorator by omitting the *handler* argument:

@jsonpickle.handlers.register(Foo, base=True) class FooHandler(jsonpickle.handlers.BaseHandler):

pass

unregister(*cls*)

class jsonpickle.handlers.**SimpleReduceHandler**(*context*)
Follow the __reduce__ protocol to pickle an object.

As long as the factory and its arguments are pickleable, this should pickle any object that implements the reduce protocol.

flatten(*obj*, *data*)

restore(*data*)

3.1.2 Low Level API

Typically this low level functionality is not needed by clients.

jsonpickle.pickler – Python to JSON

class jsonpickle.pickler.**Pickler**(*unpicklable=True*, *make_refs=True*, *max_depth=None*, *backend=None*, *keys=False*, *warn=False*, *max_iter=None*)

flatten(*obj*, *reset=True*)

Takes an object and returns a JSON-safe representation of it.

Simply returns any of the basic builtin datatypes

```
>>> p = Pickler()
>>> p.flatten('hello world') == 'hello world'
True
>>> p.flatten(49)
49
>>> p.flatten(350.0)
350.0
>>> p.flatten(True)
True
>>> p.flatten(False)
False
>>> r = p.flatten(None)
>>> r is None
True
>>> p.flatten(False)
False
>>> p.flatten([1, 2, 3, 4])
[1, 2, 3, 4]
>>> p.flatten((1,2,))[tags.TUPLE]
[1, 2]
>>> p.flatten({'key': 'value'}) == {'key': 'value'}
True
```

reset()

jsonpickle.pickler.**encode**(*value*, *unpicklable=False*, *make_refs=True*, *keys=False*, *max_depth=None*, *reset=True*, *backend=None*, *warn=False*, *context=None*, *max_iter=None*)

jsonpickle.unpickler – JSON to Python

class jsonpickle.unpickler.**Unpickler**(*backend=None*, *keys=False*, *safe=False*)

reset()

Resets the object's internal state.

restore(*obj*, *reset=True*)

Restores a flattened object to its original python state.

Simply returns any of the basic builtin types

```
>>> u = Unpickler()
>>> u.restore('hello world')
'hello world'
>>> u.restore({'key': 'value'})
{'key': 'value'}
```

jsonpickle.unpickler.**decode**(*string*, *backend=None*, *context=None*, *keys=False*, *reset=True*, *safe=False*)

jsonpickle.unpickler.**getargs**(*obj*)
Return arguments suitable for __new__()

jsonpickle.unpickler.**has_tag**(*obj*, *tag*)
Helper class that tests to see if the obj is a dictionary and contains a particular key/tag.

```
>>> obj = {'test': 1}
>>> has_tag(obj, 'test')
True
>>> has_tag(obj, 'fail')
False
```

```
>>> has_tag(42, 'fail')
False
```

jsonpickle.unpickler.**loadclass**(*module_and_name*)
Loads the module and returns the class.

```
>>> cls = loadclass('datetime.datetime')
>>> cls.__name__
'datetime'
```

```
>>> loadclass('does.not.exist')
```

```
>>> loadclass('__builtin__.int')()
0
```

jsonpickle.unpickler.**loadrepr**(*reprstr*)
Returns an instance of the object from the object's repr() string. It involves the dynamic specification of code.

```
>>> obj = loadrepr('datetime/datetime.datetime.now()')
>>> obj.__class__.__name__
'datetime'
```

jsonpickle.unpickler.**make_blank_classic**(*cls*)
Implement the mandated strategy for dealing with classic classes which cannot be instantiated without __getinitargs__ because they take parameters

jsonpickle.backend – JSON Backend Management

class jsonpickle.backend.**JSONBackend**(*fallthrough=True*)
Manages encoding and decoding using various backends.

It tries these modules in this order: simplejson, json, demjson

simplejson is a fast and popular backend and is tried first. json comes with python2.6 and is tried second. demjson is the most permissive backend and is tried last.

decode(*string*)
Attempt to decode an object from a JSON string.

This tries the loaded backends in order and passes along the last exception if no backends are able to decode the string.

dumps(*obj*)
Attempt to encode an object into JSON.

This tries the loaded backends in order and passes along the last exception if no backend is able to encode the object.

enable_fallthrough(*enable*)
Disable jsonpickle's fallthrough-on-error behavior

By default, jsonpickle tries the next backend when decoding or encoding using a backend fails.

This can make it difficult to force jsonpickle to use a specific backend, and catch errors, because the error will be suppressed and may not be raised by the subsequent backend.

Calling *enable_backend(False)* will make jsonpickle immediately re-raise any exceptions raised by the backends.

encode(*obj*)
Attempt to encode an object into JSON.

This tries the loaded backends in order and passes along the last exception if no backend is able to encode the object.

load_backend(*name*, *dumps='dumps'*, *loads='loads'*, *loads_exc=<type 'exceptions.ValueError'>*)
Load a JSON backend by name.

This method loads a backend and sets up references to that backend's loads/dumps functions and exception classes.

Parameters

- **dumps** – is the name of the backend's encode method. The method should take an object and return a string. Defaults to 'dumps'.
- **loads** – names the backend's method for the reverse operation – returning a Python object from a string.
- **loads_exc** – can be either the name of the exception class used to denote decoding errors, or it can be a direct reference to the appropriate exception class itself. If it is a name, then the assumption is that an exception class of that name can be found in the backend module's namespace.
- **load** – names the backend's 'load' method.
- **dump** – names the backend's 'dump' method.

Rtype bool True on success, False if the backend could not be loaded.

loads(*string*)
Attempt to decode an object from a JSON string.

This tries the loaded backends in order and passes along the last exception if no backends are able to decode the string.

remove_backend(*name*)
Remove all entries for a particular backend.

set_encoder_options(*name*, **args*, ***kwargs*)
Associate encoder-specific options with an encoder.

After calling set_encoder_options, any calls to jsonpickle's encode method will pass the supplied args and kwargs along to the appropriate backend's encode method.

For example:

```
set_encoder_options('simplejson', sort_keys=True, indent=4)
set_encoder_options('demjson', compactly=False)
```

See the appropriate encoder's documentation for details about the supported arguments and keyword arguments.

set_preferred_backend(*name*)
Set the preferred json backend.

If a preferred backend is set then jsonpickle tries to use it before any other backend.

For example:

```
set_preferred_backend('simplejson')
```

If the backend is not one of the built-in jsonpickle backends (json/simplejson, or demjson) then you must load the backend prior to calling set_preferred_backend.

AssertionError is raised if the backend has not been loaded.

jsonpickle.util – Helper functions

Helper functions for pickling and unpickling. Most functions assist in determining the type of an object.

jsonpickle.util.**b64decode**(*payload*)

jsonpickle.util.**b64encode**(*data*)

jsonpickle.util.**has_method**(*obj*, *name*)

jsonpickle.util.**has_reduce**(*obj*)
Tests if __reduce__ or __reduce_ex__ exists in the object dict or in the class dicts of every class in the MRO *except object*.

Returns a tuple of booleans (has_reduce, has_reduce_ex)

jsonpickle.util.**importable_name**(*cls*)

```
>>> class Example(object):
...     pass
```

```
>>> ex = Example()
>>> importable_name(ex.__class__)
'jsonpickle.util.Example'
```

```
>>> importable_name(type(25))
'__builtin__.int'
```

```
>>> importable_name(None.__class__)
'__builtin__.NoneType'
```

```
>>> importable_name(False.__class__)
'__builtin__.bool'
```

```
>>> importable_name(AttributeError)
'__builtin__.AttributeError'
```

jsonpickle.util.**in_dict**(*obj*, *key*, *default=False*)
Returns true if key exists in obj.__dict__; false if not in. If obj.__dict__ is absent, return default

jsonpickle.util.**in_slots**(*obj*, *key*, *default=False*)
Returns true if key exists in obj.__slots__; false if not in. If obj.__slots__ is absent, return default

jsonpickle.util.**is_bytes**(*obj*)
Helper method to see if the object is a bytestring.

```
>>> is_bytes(b'foo')
True
```

jsonpickle.util.**is_dictionary**(*obj*)
Helper method for testing if the object is a dictionary.

```
>>> is_dictionary({'key':'value'})
True
```

jsonpickle.util.**is_dictionary_subclass**(*obj*)
Returns True if *obj* is a subclass of the dict type. *obj* must be a subclass and not the actual builtin dict.

```
>>> class Temp(dict): pass
>>> is_dictionary_subclass(Temp())
True
```

jsonpickle.util.**is_function**(*obj*)
Returns true if passed a function

```
>>> is_function(lambda x: 1)
True
```

```
>>> is_function(locals)
True
```

```
>>> def method(): pass
>>> is_function(method)
True
```

```
>>> is_function(1)
False
```

jsonpickle.util.**is_installed**(*module*)
Tests to see if `module` is available on the sys.path

```
>>> is_installed('sys')
True
>>> is_installed('hopefullythisisnotarealmodule')
False
```

jsonpickle.util.**is_iterator**(*obj*)

jsonpickle.util.**is_list**(*obj*)
Helper method to see if the object is a Python list.

```
>>> is_list([4])
True
```

jsonpickle.util.**is_list_like**(*obj*)

jsonpickle.util.**is_module**(*obj*)
Returns True if passed a module

```
>>> import os
>>> is_module(os)
True
```

jsonpickle.util.**is_module_function**(*obj*)
Return True if *obj* is a module-global function

```
>>> import os
>>> is_module_function(os.path.exists)
True
```

```
>>> is_module_function(lambda: None)
False
```

jsonpickle.util.**is_noncomplex**(*obj*)
Returns True if *obj* is a special (weird) class, that is more complex than primitive data types, but is not a full object. Including:

- struct_time

jsonpickle.util.**is_object**(*obj*)
Returns True is obj is a reference to an object instance.

```
>>> is_object(1)
True
```

```
>>> is_object(object())
True
```

```
>>> is_object(lambda x: 1)
False
```

jsonpickle.util.**is_picklable**(*name*, *value*)
Return True if an object can be pickled

```
>>> import os
>>> is_picklable('os', os)
True
```

```
>>> def foo(): pass
>>> is_picklable('foo', foo)
True
```

```
>>> is_picklable('foo', lambda: None)
False
```

jsonpickle.util.**is_primitive**(*obj*)
Helper method to see if the object is a basic data type. Unicode strings, integers, longs, floats, booleans, and None are considered primitive and will return True when passed into *is_primitive()*

```
>>> is_primitive(3)
True
```

```
    >>> is_primitive([4,4])
    False
```

jsonpickle.util.**is_reducible**(*obj*)
Returns false if of a type which have special casing, and should not have their __reduce__ methods used

jsonpickle.util.**is_sequence**(*obj*)
Helper method to see if the object is a sequence (list, set, or tuple).

```
    >>> is_sequence([4])
    True
```

jsonpickle.util.**is_sequence_subclass**(*obj*)
Returns True if *obj* is a subclass of list, set or tuple.

obj must be a subclass and not the actual builtin, such as list, set, tuple, etc..

```
    >>> class Temp(list): pass
    >>> is_sequence_subclass(Temp())
    True
```

jsonpickle.util.**is_set**(*obj*)
Helper method to see if the object is a Python set.

```
    >>> is_set(set())
    True
```

jsonpickle.util.**is_tuple**(*obj*)
Helper method to see if the object is a Python tuple.

```
    >>> is_tuple((1,))
    True
```

jsonpickle.util.**is_type**(*obj*)
Returns True is obj is a reference to a type.

```
    >>> is_type(1)
    False
```

```
    >>> is_type(object)
    True
```

```
    >>> class Klass: pass
    >>> is_type(Klass)
    True
```

jsonpickle.util.**is_unicode**(*obj*)
Helper method to see if the object is a unicode string

jsonpickle.util.**itemgetter**(*obj*, *getter=<operator.itemgetter object>*)

jsonpickle.util.**translate_module_name**(*module*)
Rename builtin modules to a consistent (Python2) module name

This is used so that references to Python's *builtins* module can be loaded in both Python 2 and 3. We remap to the "__builtin__" name and unmap it when importing.

See untranslate_module_name() for the reverse operation.

jsonpickle.util.**untranslate_module_name**(*module*)
Rename module names mention in JSON to names that we can import

This reverses the translation applied by translate_module_name() to a module name available to the current version of Python.

CHAPTER 4

Contributing

4.1 Contributing to jsonpickle

We welcome contributions from everyone. Please fork jsonpickle on github.

4.1.1 Get the Code

```
git clone git://github.com/jsonpickle/jsonpickle.git
```

4.1.2 Run the Test Suite

Before code is pulled into the master jsonpickle branch, all tests should pass. If you are contributing an addition or a change in behavior, we ask that you document the change in the form of test cases.

The jsonpickle test suite uses several JSON encoding libraries as well as several libraries for sample objects. To simplify the process of setting up these libraries we recommend creating a virtualenv and using a pip requirements file to install the dependencies. In the base jsonpickle directory:

```
# create a virtualenv that is completely isolated from the
# site-wide python install
virtualenv --no-site-packages env

# activate the virtualenv
source env/bin/activate

# use pip to install the dependencies listed in the requirements file
pip install --upgrade -r requirements.txt
pip install --upgrade -r requirements-test.txt
```

To run the suite, simply invoke `tests/runtests.py`:

```
$ tests/runtests.py
test_None (util_tests.IsPrimitiveTestCase) ... ok
test_bool (util_tests.IsPrimitiveTestCase) ... ok
test_dict (util_tests.IsPrimitiveTestCase) ... ok
test_float (util_tests.IsPrimitiveTestCase) ... ok
...
```

4.1.3 Generate Documentation

You do not need to generate the documentation when contributing, though, if you are interested, you can generate the docs yourself. The following requires Sphinx (present if you follow the virtualenv instructions above):

```
# pull down the sphinx-to-github project
git submodule init
git submodule update

cd docs
make html
```

If you wish to browse the documentation, use Python's `SimpleHTTPServer` to host them at http://localhost:8000:

```
cd build/html
python -m SimpleHTTPServer
```

CHAPTER 5

Contact

Please join our mailing list. You can send email to *jsonpickle@googlegroups.com*.

Check http://github.com/jsonpickle/jsonpickle for project updates.

CHAPTER 6

Authors

- John Paulett - john -at- paulett.org - http://github.com/johnpaulett
- David Aguilar - davvid -at- gmail.com - http://github.com/davvid
- Dan Buch - http://github.com/meatballhat
- Ian Schenck - http://github.com/ianschenck
- David K. Hess - http://github.com/davidkhess
- Alec Thomas - http://github.com/alecthomas
- jaraco - https://github.com/jaraco
- Marcin Tustin - https://github.com/marcintustin

CHAPTER 7

Change Log

7.1 Change Log

7.1.1 Version 0.9.2 - March 20, 2015

- Fixes for serializing objects with custom handlers.
- We now properly serialize deque objects constructed with a *maxlen* parameter.
- Test suite fixes

7.1.2 Version 0.9.1 - March 15, 2015

- Support datetime objects with FixedOffsets.

7.1.3 Version 0.9.0 - January 16, 2015

- Support for Pickle Protocol v4.
- We now support serializing defaultdict subclasses that use *self* as their default factory.
- We now have a decorator syntax for registering custom handlers, and allow custom handlers to register themselves for all subclasses. (#104).
- We now support serializing types with metaclasses and their instances (e.g., Python 3 *enum*).
- We now support serializing bytestrings in both Python 2 and Python 3. In Python 2, the *str* type is decoded to UTF-8 whenever possible and serialized as a true bytestring elsewise; in Python 3, bytestrings are explicitly encoded/decoded as bytestrings. Unicode strings are always encoded as is in both Python 2 and Python 3.
- Added support for serializing numpy arrays, dtypes and scalars (see *jsonpickle.ext.numpy* module).

7.1.4 Version 0.8.0 - September 6, 2014

- We now support serializing objects that contain references to module-level functions (#77).
- Better Pickle Protocol v2 support (#78).
- Support for string __slots__ and iterable __slots__ (#67) (#68).
- *encode()* now has a *warn* option that makes jsonpickle emit warnings when encountering objects that cannot be pickled.

- A Javascript implementation of jsonpickle is now included in the jsonpickleJS directory.

7.1.5 Version 0.7.2 - August 6, 2014

- We now properly serialize classes that inherit from classes that use *__slots__* and add additional slots in the derived class.
- jsonpickle can now serialize objects that implement *__getstate__()* but not *__setstate__()*. The result of *__getstate__()* is returned as-is when doing a round-trip from Python objects to jsonpickle and back.
- Better support for collections.defaultdict with custom factories.
- Added support for *queue.Queue* objects.

7.1.6 Version 0.7.1 - May 6, 2014

- Added support for Python 3.4.
- Added support for `posix.stat_result`.

7.1.7 Version 0.7.0 - March 15, 2014

- Added `handles` decorator to *jsonpickle.handlers.BaseHandler*, enabling simple declaration of a handler for a class.
- *__getstate__()* and *__setstate__()* are now honored when pickling objects that subclass `dict`.
- jsonpickle can now serialize `collections.Counter` objects.
- Object references are properly handled when using integer keys.
- Object references are now supported when using custom handlers.
- Decimal objects are supported in Python 3.
- jsonpickle's "fallthrough-on-error" behavior can now be disabled.
- Simpler API for registering custom handlers.
- A new "safe-mode" is provided which avoids eval(). Backwards-compatible deserialization of repr-serialized objects is disabled in this mode. e.g. *decode(string, safe=True)*

7.1.8 Version 0.6.1 - August 25, 2013

- Python 3.2 support, and additional fixes for Python 3.

7.1.9 Version 0.6.0 - August 24, 2013

- Python 3 support!
- `time.struct_time` is now serialized using the built-in *jsonpickle.handlers.SimpleReduceHandler*.

7.1.10 Version 0.5.0 - August 22, 2013

- Non-string dictionary keys (e.g. ints, objects) are now supported by passing *keys=True* to `jsonpickle.encode()` and `jsonpickle.decode()`.
- We now support namedtuple, deque, and defaultdict.
- Datetimes with timezones are now fully supported.
- Better support for complicated structures e.g. datetime inside dicts.
- jsonpickle added support for references and cyclical data structures in 0.4.0. This can be disabled by passing *make_refs=False* to `jsonpickle.encode()`.

7.1.11 Version 0.4.0 - June 21, 2011

- Switch build from setuptools to distutils
- Consistent dictionary key ordering
- Fix areas with improper support for unpicklable=False
- Added support for cyclical data structures (#16).
- Experimental support for jsonlib and py-yajl backends.
- New contributers David K. Hess and Alec Thomas

Warning: To support cyclical data structures (#16), the storage format has been modified. Efforts have been made to ensure backwards-compatibility. jsonpickle 0.4.0 can read data encoded by jsonpickle 0.3.1, but earlier versions of jsonpickle may be unable to read data encoded by jsonpickle 0.4.0.

7.1.12 Version 0.3.1 - December 12, 2009

- Include tests and docs directories in sdist for distribution packages.

7.1.13 Version 0.3.0 - December 11, 2009

- Officially migrated to git from subversion. Project home now at http://jsonpickle.github.com/. Thanks to Michael Jone's sphinx-to-github.
- Fortified jsonpickle against common error conditions.
- Added support for:
- List and set subclasses.
- Objects with module references.
- Newstyle classes with *__slots__*.
- Objects implementing *__setstate__()* and *__getstate__()* (follows the `pickle` protocol).
- Improved support for Zope objects via pre-fetch.
- Support for user-defined serialization handlers via the jsonpickle.handlers registry.
- Removed cjson support per John Millikin's recommendation.
- General improvements to style, including **PEP 257** compliance and refactored project layout.

- Steps towards Python 2.3 and Python 3 support.
- New contributors Dan Buch and Ian Schenck.
- Thanks also to Kieran Darcy, Eoghan Murray, and Antonin Hildebrand for their assistance!

7.1.14 Version 0.2.0 - January 10, 2009

- Support for all major Python JSON backends (including json in Python 2.6, simplejson, cjson, and demjson)
- Handle several datetime objects using the repr() of the objects (Thanks to Antonin Hildebrand).
- Sphinx documentation
- Added support for recursive data structures
- Unicode dict-keys support
- Support for Google App Engine and Django
- Tons of additional testing and bug reports (Antonin Hildebrand, Sorin, Roberto Saccon, Faber Fedor, FirePython, and Joose)

7.1.15 Version 0.1.0 - August 21, 2008

- Added long as basic primitive (thanks Adam Fisk)
- Prefer python-cjson to simplejson, if available
- Major API change, use python-cjson's decode/encode instead of simplejson's load/loads/dump/dumps
- Added benchmark.py to compare simplejson and python-cjson

7.1.16 Version 0.0.5 - July 21, 2008

- Changed prefix of special fields to conform with CouchDB requirements (Thanks Dean Landolt). Break backwards compatibility.
- Moved to Google Code subversion
- Fixed unit test imports

7.1.17 Version 0.0.3

- Convert back to setup.py from pavement.py (issue found by spidaman)

7.1.18 Version 0.0.2

- Handle feedparser's FeedParserDict
- Converted project to Paver
- Restructured directories
- Increase test coverage

7.1.19 Version 0.0.1

Initial release

CHAPTER 8

License

jsonpickle is provided under a New BSD license,

j

Symbols

B

C

D

E

F

G

H

I

J

L

M

O

P

Q

R

S

T

U