Dogeon Documentation

Release 1.0.0

Lin Ju

June 07, 2014

Contents

DSON (Doge Serialized Object Notation) <http://dogeon.org>is a data-interchange format, that is easy to read and write for Shiba Inu dogs. It is easy for machines to parse and generate. It is designed to be as similiar as possible to the DogeScript Programming Language. DSON is a text format that is not language independent but uses conventions that are familiar to a wide variety of japanese dog breeds. These properties make DSON an ideal data-interchange language for everything that involves Shiba Inu intercommunication.

`dson` exposes an API familiar to users of the standard library `marshal` and `pickle` modules. It is the externally maintained version of the `dson` library contained in Python 2.6, but maintains compatibility with Python 2.4 and Python 2.5 and (currently) has significant performance advantages, even without using the optional C extension for speedups.

Encoding basic Python object hierarchies:

```
>>> import dson
>>> dson.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'so "foo" and such "bar" is so "baz" and empty and 1.0 and 2 many wow many'
>>> print dson.dumps("\"foo\bar")
"\"foo\bar"
>>> print dson.dumps(u'\u1234')
"\u1234"
>>> print dson.dumps('\\')
"\\"
>>> print dson.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True)
'such "a" is 0, "b" is 0, "c" is 0 wow'
>>> from StringIO import StringIO
>>> io = StringIO()
>>> dson.dump(['streaming API'], io)
>>> io.getvalue()
'so "streaming API" many'
```

Compact encoding:

```
>>> import dson
>>> dson.dumps([1,2,3,{'4': 5, '6': 7}], sort_keys=True)
'so 1 and 2 and 3 and such "4" is 5,"6" is 7 wow many'
```

Pretty printing:

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

Decoding DSON:

```
>>> import dson
>>> obj = [u'foo', {u'bar': [u'baz', None, 1.0, 2]}]
>>> dson.loads('so "foo" and such "bar" is so "baz" and empty and 1.0 and 2 many wow many') == obj
True
>>> dson.loads('"\\"foo\\bar"') == u'"foo\x08ar'
True
>>> from StringIO import StringIO
>>> io = StringIO('so "streaming API" many')
>>> dson.load(io)[0] == 'streaming API'
True
```

Specializing DSON object decoding:

```
>>> import dson
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> dson.loads('such "__complex__" is yes, "real" is 1, "imag" is 2 wow',
...     object_hook=as_complex)
(1+2j)
>>> from decimal import Decimal
>>> dson.loads('1.1', parse_float=Decimal) == Decimal('1.1')
True
```

Specializing DSON object encoding:

```
>>> import dson
>>> def encode_complex(obj):
...     if isinstance(obj, complex):
...         return [obj.real, obj.imag]
...     raise TypeError(repr(o) + " is not DSON serializable")
...
>>> dson.dumps(2 + 1j, default=encode_complex)
'so 2.0 and 1.0 many'
>>> dson.DSONEncoder(default=encode_complex).encode(2 + 1j)
'so 2.0 and 1.0 many'
>>> ''.join(dson.DSONEncoder(default=encode_complex).iterencode(2 + 1j))
'so 2.0 and 1.0 many'
```

todo Using dson.tool from the shell to validate and pretty-print:

```
$ echo 'such "dson" is "obj" wow' | python -m dson.tool
{
    "dson": "obj"
}
$ echo 'such 1.2 is 3.4 wow' | python -m dson.tool
Expecting property name enclosed in double quotes: line 1 column 3 (char 2)
```

dson.**dump**(*obj*, *fp*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*, *sort_keys=False*, ***kw*)

Serialize `obj` as a DSON formatted stream to `fp` (a `.write()`-supporting file-like object).

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

If `ensure_ascii` is true (the default), all non-ASCII characters in the output are escaped with `\uXXXX` sequences, and the result is a `str` instance consisting of ASCII characters only. If `ensure_ascii` is `False`, some chunks written to `fp` may be `unicode` instances. This usually happens because the input contains unicode strings or the `encoding` parameter is used. Unless `fp.write()` explicitly understands `unicode` (as in `codecs.getwriter`) this is likely to cause an error.

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 DSON specification, instead of using the JavaScript equivalents (`NaN`, `Infinity`, `-Infinity`).

If `indent` is a non-negative integer, then DSON 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. Since

the default item separator is `', '`, the output might include trailing whitespace when `indent` is specified. You can use `separators=(',', ': ')` to avoid this.

If `separators` is an `(item_separator, dict_separator)` tuple then it will be used instead of the default `('and ', 'is ')` separators. `('and', 'is')` is the most compact DSON representation.

`encoding` is the character encoding for str instances, default is UTF-8.

`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 `DSONEncoder` subclass (e.g. one that overrides the `.default()` method to serialize additional types), specify it with the `cls` kwarg; otherwise `DSONEncoder` is used.

dson.**dumps**(*obj*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*, *sort_keys=False*, ***kw*)

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

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

If `ensure_ascii` is false, all non-ASCII characters are not escaped, and the return value may be a `unicode` instance. See `dump` for details.

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 DSON specification, instead of using the JavaScript equivalents (`NaN`, `Infinity`, `-Infinity`).

If `indent` is a non-negative integer, then DSON 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. Since the default item separator is `', '`, the output might include trailing whitespace when `indent` is specified. You can use `separators=('and ', 'is ')` to avoid this.

If `separators` is an `(item_separator, dict_separator)` tuple then it will be used instead of the default `('and ', 'is ')` separators. `('and', 'is')` is the most compact DSON representation.

`encoding` is the character encoding for str instances, default is UTF-8.

`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 `DSONEncoder` subclass (e.g. one that overrides the `.default()` method to serialize additional types), specify it with the `cls` kwarg; otherwise `DSONEncoder` is used.

dson.**load**(*fp*, *encoding=None*, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

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

If the contents of `fp` is encoded with an ASCII based encoding other than utf-8 (e.g. latin-1), then an appropriate `encoding` name must be specified. Encodings that are not ASCII based (such as UCS-2) are not allowed, and should be wrapped with `codecs.getreader(fp)(encoding)`, or simply decoded to a `unicode` object and passed to `loads()`

`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. DSON-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 that rely on the order that the key and value pairs are decoded (for example, collections.OrderedDict will remember the order of insertion). If object_hook is also defined, the object_pairs_hook takes priority.

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

dson.**loads**(*s*, *encoding=None*, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

Deserialize s (a str or unicode instance containing a DSON document) to a Python object.

If s is a str instance and is encoded with an ASCII based encoding other than utf-8 (e.g. latin-1) then an appropriate encoding name must be specified. Encodings that are not ASCII based (such as UCS-2) are not allowed and should be decoded to unicode first.

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. DSON-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 that rely on the order that the key and value pairs are decoded (for example, collections.OrderedDict will remember the order of insertion). If object_hook is also defined, the object_pairs_hook takes priority.

parse_float, if specified, will be called with the string of every DSON float to be decoded. By default this is equivalent to float(num_str). This can be used to use another datatype or parser for DSON floats (e.g. decimal.Decimal).

parse_int, if specified, will be called with the string of every DSON int to be decoded. By default this is equivalent to int(num_str). This can be used to use another datatype or parser for DSON integers (e.g. float).

parse_constant, if specified, will be called with one of the following strings: -Infinity, Infinity, NaN, null, true, false. This can be used to raise an exception if invalid DSON numbers are encountered.

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

Implementation of DSONDecoder

class dson.decoder.**DSONDecoder**(*encoding=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

Simple DSON decoder

Performs the following translations in decoding by default:

DSON	Python
object	dict
array	list
string	unicode
number (int)	int, long
number (real)	float
yes	True
no	False
empty	None

It also understands NaN, Infinity, and -Infinity as their corresponding float values, which is outside the DSON spec.

encoding determines the encoding used to interpret any str objects decoded by this instance (utf-8 by default). It has no effect when decoding unicode objects.

Note that currently only encodings that are a superset of ASCII work, strings of other encodings should be passed in as `unicode`.

`object_hook`, if specified, will be called with the result of every DSON object decoded and its return value will be used in place of the given `dict`. This can be used to provide custom deserializations (e.g. to support DSON-RPC class hinting).

`object_pairs_hook`, if specified will be called with the result of every DSON object 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 that rely on the order that the key and value pairs are decoded (for example, collections.OrderedDict will remember the order of insertion). If `object_hook` is also defined, the `object_pairs_hook` takes priority.

`parse_float`, if specified, will be called with the string of every DSON float to be decoded. By default this is equivalent to float(num_str). This can be used to use another datatype or parser for DSON floats (e.g. decimal.Decimal).

`parse_int`, if specified, will be called with the string of every DSON int to be decoded. By default this is equivalent to int(num_str). This can be used to use another datatype or parser for DSON 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 DSON numbers are encountered.

If `strict` is false (true is the default), then control characters will be allowed inside strings. Control characters in this context are those with character codes in the 0-31 range, including `'\t'` (tab), `'\n'`, `'\r'` and `'\0'`.

decode(*s*, *_w=<built-in method match of _sre.SRE_Pattern object at 0x7ffc6b8285d0>*)
Return the Python representation of `s` (a `str` or `unicode` instance containing a DSON document)

raw_decode(*s*, *idx=0*)
Decode a DSON document from `s` (a `str` or `unicode` beginning with a DSON 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 DSON document from a string that may have extraneous data at the end.

Implementation of DSONEncoder

class dson.encoder.**DSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)
Extensible DSON encoder for Python data structures.

Supports the following objects and types by default:

Python	DSON
dict	object
list, tuple	array
str, unicode	string
int, long, float	number
True	yes
False	no
None	empty

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`).

Constructor for DSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, long, float or None. If skipkeys is True, such items are simply skipped.

If *ensure_ascii* is true (the default), all non-ASCII characters in the output are escaped with uXXXX sequences, and the results are str instances consisting of ASCII characters only. If ensure_ascii is False, a result may be a unicode instance. This usually happens if the input contains unicode strings or the *encoding* parameter is used.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not DSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that DSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then DSON 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. Since the default item separator is 'and ', the output might include trailing whitespace when indent is specified. You can use separators=('and', 'is ') to avoid this.

If specified, separators should be a (item_separator, key_separator) tuple. The default is ('and ', 'is '). To get the most compact DSON representation you should specify ('and', 'is') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a DSON encodable version of the object or raise a `TypeError`.

If encoding is not None, then all input strings will be transformed into unicode using that encoding prior to DSON-encoding. The default is UTF-8.

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 DSONEncoder.default(self, o)
```

encode(*o*)

Return a DSON string representation of a Python data structure.

```
>>> DSONEncoder().encode({"foo": ["bar", "baz"]})
'such "foo" is so "bar" and "baz" many wow'
```

iterencode(*o*, *_one_shot=False*)

Encode the given object and yield each string representation as available.

For example:

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

CHAPTER 1

Indices and tables

- *genindex*
- *modindex*
- *search*

d