cbor2

Release 5.1.0

Mar 18, 2020
Serializing and deserializing with cbor2 is pretty straightforward:

```python
from cbor2 import dumps, loads

# Serialize an object as a bytestring
data = dumps(['hello', 'world'])

# Deserialize a bytestring
obj = loads(data)

# Efficiently deserialize from a file
with open('input.cbor', 'rb') as fp:
    obj = load(fp)

# Efficiently serialize an object to a file
with open('output.cbor', 'wb') as fp:
    dump(obj, fp)
```

Some data types, however, require extra considerations, as detailed below.

### 1.1 String/bytes handling on Python 2

The `str` type is encoded as binary on Python 2. If you want to encode strings as text on Python 2, use unicode strings instead.

### 1.2 Date/time handling

The CBOR specification does not support naïve datetimes (that is, datetimes where `tzinfo` is missing). When the encoder encounters such a datetime, it needs to know which timezone it belongs to. To this end, you can specify a
default timezone by passing a `tzinfo` instance to `dump()`/`dumps()` call as the `timezone` argument. Decoded
datetimes are always timezone aware.

By default, datetimes are serialized in a manner that retains their timezone offsets. You can optimize the data stream
size by passing `datetime_as_timestamp=False` to `dump()`/`dumps()`, but this causes the timezone offset
information to be lost.

In versions prior to 4.2 the encoder would convert a `datetime.date` object into a `datetime.datetime` prior to
writing. This can cause confusion on decoding so this has been disabled by default in the next version. The behaviour
can be re-enabled as follows:

```python
from cbor2 import dumps
from datetime import date, timezone

# Serialize dates as datetimes
encoded = dumps(date(2019, 10, 28), timezone=timezone.utc, date_as_datetime=True)
```

A default timezone offset must be provided also.

### 1.3 Cyclic (recursive) data structures

If the encoder encounters a shareable object (ie. list or dict) that it has seen before, it will by default raise
`CBOREncodeError` indicating that a cyclic reference has been detected and value sharing was not enabled. CBOR
has, however, an extension specification that allows the encoder to reference a previously encoded value without pro-
cessing it again. This makes it possible to serialize such cyclic references, but value sharing has to be enabled by
passing `value_sharing=True` to `dump()`/`dumps()`.

**Warning:** Support for value sharing is rare in other CBOR implementations, so think carefully whether you want
to enable it. It also causes some line overhead, as all potentially shareable values must be tagged as such.

### 1.4 Tag support

In addition to all standard CBOR tags, this library supports many extended tags:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Semantics</th>
<th>Python type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Standard date/time string</td>
<td><code>datetime.date / datetime.datetime</code></td>
</tr>
<tr>
<td>1</td>
<td>Epoch-based date/time</td>
<td><code>datetime.date / datetime.datetime</code></td>
</tr>
<tr>
<td>2</td>
<td>Positive bignum</td>
<td><code>int / long</code></td>
</tr>
<tr>
<td>3</td>
<td>Negative bignum</td>
<td><code>int / long</code></td>
</tr>
<tr>
<td>4</td>
<td>Decimal fraction</td>
<td><code>decimal.Decimal</code></td>
</tr>
<tr>
<td>5</td>
<td>Bigfloat</td>
<td><code>decimal.Decimal</code></td>
</tr>
<tr>
<td>28</td>
<td>Mark shared value</td>
<td>N/A</td>
</tr>
<tr>
<td>29</td>
<td>Reference shared value</td>
<td>N/A</td>
</tr>
<tr>
<td>30</td>
<td>Rational number</td>
<td><code>fractions.Fraction</code></td>
</tr>
<tr>
<td>35</td>
<td>Regular expression</td>
<td><code>_sre.SRE_Pattern(result of re.compile(...))</code></td>
</tr>
<tr>
<td>36</td>
<td>MIME message</td>
<td><code>email.message.Message</code></td>
</tr>
<tr>
<td>37</td>
<td>Binary UUID</td>
<td><code>uuid.UUID</code></td>
</tr>
<tr>
<td>258</td>
<td>Set of unique items</td>
<td><code>set</code></td>
</tr>
<tr>
<td>260</td>
<td>Network address</td>
<td><code>ipaddress.IPv4Address (or IPv6)</code></td>
</tr>
<tr>
<td>261</td>
<td>Network prefix</td>
<td><code>ipaddress.IPv4Network (or IPv6)</code></td>
</tr>
</tbody>
</table>
Arbitrary tags can be represented with the CBORTag class.

1.5 Use Cases

Here are some things that the cbor2 library could be (and in some cases, is being) used for:

- Experimenting with network protocols based on CBOR encoding
- Designing new data storage formats
- Submitting binary documents to ElasticSearch without base64 encoding overhead
- Storing and validating file metadata in a secure backup system
- RPC which supports Decimals with low overhead
Customizing encoding and decoding

Both the encoder and decoder can be customized to support a wider range of types.

On the encoder side, this is accomplished by passing a callback as the `default` constructor argument. This callback will receive an object that the encoder could not serialize on its own. The callback should then return a value that the encoder can serialize on its own, although the return value is allowed to contain objects that also require the encoder to use the callback, as long as it won’t result in an infinite loop.

On the decoder side, you have two options: `tag_hook` and `object_hook`. The former is called by the decoder to process any semantic tags that have no predefined decoders. The latter is called for any newly decoded `dict` objects, and is mostly useful for implementing a JSON compatible custom type serialization scheme. Unless your requirements restrict you to JSON compatible types only, it is recommended to use `tag_hook` for this purpose.

### 2.1 Using the CBOR tags for custom types

The most common way to use `default` is to call `encode()` to add a custom tag in the data stream, with the payload as the value:

```python
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

def default_encoder(encoder, value):
    # Tag number 4000 was chosen arbitrarily
    encoder.encode(CBORTag(4000, [value.x, value.y]))
```

The corresponding `tag_hook` would be:

```python
def tag_hook(decoder, tag, shareable_index=None):
    if tag.tag != 4000:
        return tag
```

(continues on next page)
2.2 Using dicts to carry custom types

The same could be done with `object_hook`, except less efficiently:

```python
def default_encoder(encoder, value):
    encoder.encode(dict(typename='Point', x=value.x, y=value.y))

def object_hook(decoder, value):
    if value.get('typename') != 'Point':
        return value
    return Point(value['x'], value['y'])
```

You should make sure that whatever way you decide to use for telling apart your “specially marked” dicts from arbitrary data dicts won’t mistake on for the other.

2.3 Value sharing with custom types

In order to properly encode and decode cyclic references with custom types, some special care has to be taken. Suppose you have a custom type as below, where every child object contains a reference to its parent and the parent contains a list of children:

```python
from cbor2 import dumps, loads, shareable_encoder, CBORTag

class MyType(object):
    def __init__(self, parent=None):
        self.parent = parent
        self.children = []
        if parent:
            parent.children.append(self)
```

This would not normally be serializable, as it would lead to an endless loop (in the worst case) and raise some exception (in the best case). Now, enter CBOR’s extension tags 28 and 29. These tags make it possible to add special markers into the data stream which can be later referenced and substituted with the object marked earlier.

To do this, in `default` hooks used with the encoder you will need to use the `shareable_encoder()` decorator on your `default` hook function. It will automatically automatically add the object to the shared values registry on the encoder and prevent it from being serialized twice (instead writing a reference to the data stream):

```python
@shareable_encoder
def default_encoder(encoder, value):
    # The state has to be serialized separately so that the decoder would have a chance to
    # create an empty instance before the shared value references are decoded
    serialized_state = encoder.encode_to_bytes(value.__dict__)
    encoder.encode(CBORTag(3000, serialized_state))
```
On the decoder side, you will need to initialize an empty instance for shared value lookup before the object’s state (which may contain references to it) is decoded. This is done with the `set_shareable()` method:

```python
def tag_hook(decoder, tag, shareable_index=None):
    # Return all other tags as-is
    if tag.tag != 3000:
        return tag

    # Create a raw instance before initializing its state to make it possible for cyclic
    instance = MyType.__new__(MyType)
    decoder.set_shareable(shareable_index, instance)

    # Separately decode the state of the new object and then apply it
    state = decoder.decode_from_bytes(tag.value)
    instance.__dict__.update(state)
    return instance
```

You could then verify that the cyclic references have been restored after deserialization:

```python
def tag_hook(decoder, tag, shareable_index=None):
    if tag.tag != 3000:
        return tag

    if decoder.immutable:
        raise CBORDecodeException('MyType cannot be used as a key or set member')

    return MyType(*tag.value)
```

### 2.4 Decoding Tagged items as keys

Since the CBOR specification allows any type to be used as a key in the mapping type, the decoder provides a flag that indicates it is expecting an immutable (and by implication hashable) type. If your custom class cannot be used this way you can raise an exception if this flag is set:

```python
from collections import namedtuple
Pair = namedtuple('Pair', 'first second')
```

```python
from collections import namedtuple
Pair = namedtuple('Pair', 'first second')
```

```python
def tag_hook(decoder, tag, shareable_index=None):
    if tag.tag != 4000:
        return tag

    raise CBORDecodeException('MyType cannot be used as a key or set member')
```

An example where the data could be used as a dict key:

```python
from collections import namedtuple
```

```python
from collections import namedtuple
Pair = namedtuple('Pair', 'first second')
```

```python
def tag_hook(decoder, tag, shareable_index=None):
    if tag.tag != 4000:
        return tag

    return Pair(*tag.value)
```
The `object_hook` can check for the immutable flag in the same way.
This library adheres to Semantic Versioning.

5.1.0 (2020-03-18)
  • Minor API change CBORSimpleValue is now a subclass of namedtuple and allows all numeric comparisons. This brings functional parity between C and Python modules.
  • Fixes for C-module on big-endian systems including floating point decoding, smallint encoding, and boolean argument handling. Tested on s390x and MIPS32.
  • Increase version required of setuptools during install due to unicode errors.

5.0.1 (2020-01-21)
  • Fix deprecation warning on python 3.7, 3.8 (mariano54)
  • Minor documentation tweaks

5.0.0 (2020-01-20)
  • BACKWARD INCOMPATIBLE CBOR does not have a bare DATE type, encoding dates as datetimes is disabled by default (PR by Changaco)
  • BACKWARD INCOMPATIBLE set_shareable() only takes the instance to share, not the shareable’s index
  • BACKWARD INCOMPATIBLE CBORError now descends from Exception rather than ValueError; however, subordinate exceptions now descend from ValueError (where appropriate) so most users should notice no difference
  • BACKWARD INCOMPATIBLE CBORDecoder can now raise CBORDecodeEOF which descends from EOFError supporting streaming applications
  • Optional Pure C implementation by waveform80 that functions identically to the pure Python implementation with further contributions from: toravir, jonashoechst, Changaco
  • Drop Python 3.3 and 3.4 support from the build process; they should still work if built from source but are no longer officially supported
cbor2, Release 5.1.0

- Added support for encoding and decoding \texttt{ipaddress.IPv4Address}, \texttt{ipaddress.IPv6Address}, \texttt{ipaddress.IPv4Network}, and \texttt{ipaddress.IPv6Network} (semantic tags 260 and 261)

4.2.0 (2020-01-10)
- **BROKEN BUILD** Removed

4.1.2 (2018-12-10)
- Fixed bigint encoding taking quadratic time
- Fixed overflow errors when encoding floating point numbers in canonical mode
- Improved decoder performance for dictionaries
- Minor documentation tweaks

4.1.1 (2018-10-14)
- Fixed encoding of negative \texttt{decimal.Decimal} instances (PR by Sekenre)

4.1.0 (2018-05-27)
- Added canonical encoding (via \texttt{canonical=True}) (PR by Sekenre)
- Added support for encoding/decoding sets (semantic tag 258) (PR by Sekenre)
- Added support for encoding \texttt{FrozenDict} (hashable dict) as map keys or set elements (PR by Sekenre)

4.0.1 (2017-08-21)
- Fixed silent truncation of decoded data if there are not enough bytes in the stream for an exact read (\texttt{CBORDecodeError} is now raised instead)

4.0.0 (2017-04-24)
- **BACKWARD INCOMPATIBLE** Value sharing has been disabled by default, for better compatibility with other implementations and better performance (since it is rarely needed)
- **BACKWARD INCOMPATIBLE** Replaced the \texttt{semantic_decoders} decoder option with the \texttt{CBORDecoder.tag_hook} option
- **BACKWARD INCOMPATIBLE** Replaced the \texttt{encoders} encoder option with the \texttt{CBOREncoder.default} option
- **BACKWARD INCOMPATIBLE** Factored out the file object argument (\texttt{fp}) from all callbacks
- **BACKWARD INCOMPATIBLE** The encoder no longer supports every imaginable type implementing the \texttt{Sequence} or \texttt{Map} interface, as they turned out to be too broad
- Added the \texttt{CBORDecoder.object_hook} option for decoding dicts into complex objects (intended for situations where JSON compatibility is required and semantic tags cannot be used)
- Added encoding and decoding of simple values (\texttt{CBORSimpleValue}) (contributed by Jerry Lundström)
- Replaced the decoder for bignums with a simpler and faster version (contributed by orent)
- Made all relevant classes and functions available directly in the \texttt{cbor2} namespace
- Added proper documentation

3.0.4 (2016-09-24)
- Fixed TypeError when trying to encode extension types (regression introduced in 3.0.3)

3.0.3 (2016-09-23)
- No changes, just re-releasing due to git tagging screw-up
3.0.2 (2016-09-23)
• Fixed decoding failure for datetimes with microseconds (tag 0)

3.0.1 (2016-08-08)
• Fixed error in the cyclic structure detection code that could mistake one container for another, sometimes causing a bogus error about cyclic data structures where there was none

3.0.0 (2016-07-03)
• BACKWARD INCOMPATIBLE Encoder callbacks now receive three arguments: the encoder instance, the value to encode and a file-like object. The callback must now either write directly to the file-like object or call another encoder callback instead of returning an iterable.
• BACKWARD INCOMPATIBLE Semantic decoder callbacks now receive four arguments: the decoder instance, the primitive value, a file-like object and the shareable index for the decoded value. Decoders that support value sharing must now set the raw value at the given index in `decoder.shareables`.
• BACKWARD INCOMPATIBLE Removed support for iterative encoding (`CBOREncoder.encode()` is no longer a generator function and always returns `None`)
• Significantly improved performance (encoder ~30 % faster, decoder ~60 % faster)
• Fixed serialization round-trip for `undefined` (simple type 23)
• Added proper support for value sharing in callbacks

2.0.0 (2016-06-11)
• BACKWARD INCOMPATIBLE Deserialize unknown tags as `CBORTag` objects so as not to lose information
• Fixed error messages coming from nested structures

1.1.0 (2016-06-10)
• Fixed deserialization of cyclic structures

1.0.0 (2016-06-08)
• Initial release
• API reference