cbor2

Release 5.4.3

Alex Grönholm

May 09, 2022

CONTENTS

target https://cbor2.readthedocs.io/en/latest/?badge=latest

alt Documentation Status

This library provides encoding and decoding for the Concise Binary Object Representation (CBOR) (RFC 8949) serialization format. The specification is fully compatible with the original RFC 7049. Read the docs to learn more.

It is implemented in pure python with an optional C backend.

On PyPy, cbor2 runs with almost identical performance to the C backend.

CHAPTER ONE

FEATURES

- Simple api like `json` or `pickle` modules.
- Support many CBOR tags with stdlib objects.
- Generic tag decoding.
- Shared value references including cyclic references.
- String references compact encoding with repeated strings replaced with indices.
- Optional C module backend tested on big- and little-endian architectures.
- Extensible tagged value handling using `tag_hook` and `object_hook` on decode and `default` on encode.
- Command-line diagnostic tool, converting CBOR file or stream to JSON `python -m cbor2.tool` (This is a lossy conversion, for diagnostics only)
- Thorough test suite.

CHAPTER

TWO

INSTALLATION

```
pip install cbor2
```

2.1 Requirements

- Python >= 3.7 (or PyPy3 3.7+)
- C-extension: Any C compiler that can build Python extensions. Any modern libc with the exception of Glibc<2.9

2.2 Building the C-Extension

To force building of the optional C-extension, set OS env `CBOR2_BUILD_C_EXTENSION=1`. To disable building of the optional C-extension, set OS env `CBOR2_BUILD_C_EXTENSION=0`. If this environment variable is unset, setup.py will default to auto detecting a compatible C library and attempt to compile the extension.

CHAPTER

THREE

USAGE

CHAPTER

FOUR

COMMAND-LINE USAGE

`python -m cbor2.tool` converts CBOR data in raw binary or base64 encoding into a representation that allows printing as JSON. This is a lossy transformation as each datatype is converted into something that can be represented as a JSON value.

Usage:

```
# Pass hexadecimal through xxd.
$ echo a16568656c6c6f65776f726c64 | xxd -r -ps | python -m cbor2.tool --pretty
{
    "hello": "world"
}
# Decode Base64 directly
$ echo ggEC | python -m cbor2.tool --decode
[1, 2]
# Read from a file encoded in Base64
$ python -m cbor2.tool -d tests/examples.cbor.b64
{...}
```

It can be used in a pipeline with json processing tools like jq to allow syntax coloring, field extraction and more.

CBOR data items concatenated into a sequence can be decoded also:

```
$ echo ggECggMEggUG | python -m cbor2.tool -d --sequence
[1, 2]
[3, 4]
[5, 6]
```

Multiple files can also be sent to a single output file:

```
$ python -m cbor2.tool -o all_files.json file1.cbor file2.cbor ... fileN.cbor
```

CHAPTER

FIVE

SECURITY

This library has not been tested against malicious input. In theory it should be as safe as JSON, since unlike `pickle` the decoder does not execute any code.

5.1 Table of contents

5.1.1 Basic usage

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.

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.

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 processing 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.

String references

When `string_referencing=True` is passed to *dump()*/*dumps()*, if the encoder would encode a string that it has previously encoded and where a reference would be shorter than the encoded string, it instead encodes a reference to the nth sufficiently long string already encoded.

Warning: Support for string referencing is rare in other CBOR implementations, so think carefully whether you want to enable it.

Tag support

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

Tag	Semantics	Python type(s)
0	Standard date/time string	datetime.date / datetime.datetime
1	Epoch-based date/time	datetime.date / datetime.datetime
2	Positive bignum	int / long
3	Negative bignum	int / long
4	Decimal fraction	decimal.Decimal
5	Bigfloat	decimal.Decimal
25	String reference	str / bytes
28	Mark shared value	N/A
29	Reference shared value	N/A
30	Rational number	fractions.Fraction
35	Regular expression	`_sre.SRE_Pattern` (result of `re.compile(...)`)
36	MIME message	email.message.Message
37	Binary UUID	uuid.UUID
256	String reference namespace	N/A
258	Set of unique items	set
260	Network address	`ipaddress.IPv4Address` (or IPv6)
261	Network prefix	`ipaddress.IPv4Network` (or IPv6)
55799	Self-Described CBOR	object

Arbitary tags can be represented with the *CBORTag* class.

If you want to write a file that is detected as CBOR by the Unix `file` utility, wrap your data in a `~cbor2.types.CBORTag` object like so:

```
from cbor2 import dump, CBORTag

with open('output.cbor', 'wb') as fp:
    dump(CBORTag(55799, obj), fp)
```

This will be ignored on decode and the original data content will be returned.

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

5.1.2 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.

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:

```
class Point:
    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:

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

    # tag.value is now the [x, y] list we serialized before
    return Point(*tag.value)
```

Using dicts to carry custom types

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

```
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.

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:
    def __init__(self, parent=None):
        self.parent = parent
        self.children = []
        if parent:
            self.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
    # references to work
    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
parent = MyType()
child1 = MyType(parent)
```

(continues on next page)

(continued from previous page)

```
child2 = MyType(parent)
serialized = dumps(parent, default=default_encoder, value_sharing=True)

new_parent = loads(serialized, tag_hook=tag_hook)
assert new_parent.children[0].parent is new_parent
assert new_parent.children[1].parent is new_parent
```

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:

```
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)
```

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

```
from collections import namedtuple

Pair = namedtuple('Pair', 'first second')

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.

5.1.3 Version history

This library adheres to Semantic Versioning.

5.4.3 (2022-05-03)

- Removed support for Python < 3.7
- Various build system improvements for binary wheels (agronholm)
- Migrated project to use `pyproject.toml` and pre-commit hooks (agronholm)

5.4.2 (2021-10-14)

- Fix segfault when initializing CBORTag with incorrect arguments (Sekenre)
- Fix sphinx build warnings (Sekenre)

5.4.1 (2021-07-23)

- Fix SystemErrors when using C-backend, meaningful exceptions now raised (Sekenre)
- Fix precision loss when decoding base10 decimal fractions (Sekenre)
- Made CBORTag handling consistent between python and C-module (Sekenre)

5.4.0 (2021-06-04)

- Fix various bounds checks in the C-backend (Sekenre)
- More testing of invalid/corrupted data (Sekenre)
- Support for String References (xurtis)
- Update Docs to refer to new RFC8949

5.3.0 (2021-05-18)

- Removed support for Python < 3.6

5.2.0 (2020-09-30)

- Final version tested with Python 2.7 and 3.5
- README: Announce deprecation of Python 2.7, 3.5
- README: More detail and examples
- Bugfix: Fix segfault on loading huge arrays with C-backend (Sekenre)
- Build system: Allow packagers to force C-backend building or disable using env var (jameshilliard)
- Feature: `cbor2.tool` Command line diagnostic tool (Sekenre)
- Feature: Ignore semantic tag used for file magic 55799 AKA “Self-Described CBOR” (kalcutter)

5.1.2 (2020-07-21)

- Bugfix: Refcount bug in C lib causing intermittent segfaults on shutdown (tdryer)

5.1.1 (2020-07-03)

- Build system: Making C lib optional if it fails to compile (chiefnoah)
- Build system: Better Glibc version detection (Sekenre and JayH5)
- Tests: Positive and negative bignums (kalcutter)
- Bugfix: Fractional seconds parsing in datetimes (kalcutter)

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 requred 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
- Added support for encoding and decoding `ipaddress.IPv4Address`, `ipaddress.IPv6Address`, `ipaddress.IPv4Network`, and `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 `decimal.Decimal` instances (PR by Sekenre)

4.1.0 (2018-05-27)

- Added canonical encoding (via `canonical=True`) (PR by Sekenre)
- Added support for encoding/decoding sets (semantic tag 258) (PR by Sekenre)
- Added support for encoding `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 (`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 `semantic_decoders` decoder option with the `CBORDecoder.tag_hook` option
- **BACKWARD INCOMPATIBLE** Replaced the `encoders` encoder option with the `CBOREncoder.default` option
- **BACKWARD INCOMPATIBLE** Factored out the file object argument (`fp`) from all callbacks
- **BACKWARD INCOMPATIBLE** The encoder no longer supports every imaginable type implementing the `Sequence` or `Map` interface, as they turned out to be too broad
- Added the `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 (`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 `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 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

5.1.4 `cbor2.encoder`

class `cbor2.encoder.`**`CBOREncoder`**(*fp*, *datetime_as_timestamp=False*, *timezone=None*, *value_sharing=False*, *default=None*, *canonical=False*, *date_as_datetime=False*, *string_referencing=False*)

The CBOREncoder class implements a fully featured CBOR encoder with several extensions for handling shared references, big integers, rational numbers and so on. Typically the class is not used directly, but the *`dump()`* and *`dumps()`* functions are called to indirectly construct and use the class.

When the class is constructed manually, the main entry points are *`encode()`* and *`encode_to_bytes()`*.

Parameters

- **`datetime_as_timestamp`** (*`bool`*) – set to `True` to serialize datetimes as UNIX timestamps (this makes datetimes more concise on the wire, but loses the timezone information)
- **`timezone`** (*`datetime.tzinfo`*) – the default timezone to use for serializing naive datetimes; if this is not specified naive datetimes will throw a `ValueError` when encoding is attempted
- **`value_sharing`** (*`bool`*) – set to `True` to allow more efficient serializing of repeated values and, more importantly, cyclic data structures, at the cost of extra line overhead
- **`default`** – a callable that is called by the encoder with two arguments (the encoder instance and the value being encoded) when no suitable encoder has been found, and should use the methods on the encoder to encode any objects it wants to add to the data stream
- **`canonical`** (*`bool`*) – when True, use "canonical" CBOR representation; this typically involves sorting maps, sets, etc. into a pre-determined order ensuring that serializations are comparable without decoding
- **`date_as_datetime`** (*`bool`*) – set to `True` to serialize date objects as datetimes (CBOR tag 0), which was the default behavior in previous releases (cbor2 <= 4.1.2).
- **`string_referencing`** (*`bool`*) – set to `True` to allow more efficient serializing of repeated string values

`disable_string_namespacing`()

Disable generation of new string namespaces for the duration of the context block.

`disable_string_referencing`()

Disable tracking of string references for the duration of the context block.

`disable_value_sharing`()

Disable value sharing in the encoder for the duration of the context block.

`encode`(*obj*)

Encode the given object using CBOR.

Parameters **`obj`** – the object to encode

`encode_canonical_map`(*value*)

Reorder keys according to Canonical CBOR specification

`encode_sortable_key`(*value*)

Takes a key and calculates the length of its optimal byte representation, along with the representation itself. This is used as the sorting key in CBOR's canonical representations.

encode_to_bytes(*obj*)

Encode the given object to a byte buffer and return its value as bytes.

This method was intended to be used from the `default` hook when an object needs to be encoded separately from the rest but while still taking advantage of the shared value registry.

write(*data*)

Write bytes to the data stream.

Parameters **data** (*bytes*) – the bytes to write

cbor2.encoder.**container_encoder**(*func*)

The given encoder is a container with child values. Handle cyclic or duplicate references to the value and strings within the value efficiently.

Containers may contain cyclic data structures or may contain values or themselves by referenced multiple times throughout the greater encoded value and could thus be more efficiently encoded with shared value references and string references where duplication occurs.

If value sharing is enabled, this marks the given value shared in the datastream on the first call. If the value has already been passed to this method, a reference marker is instead written to the data stream and the wrapped function is not called.

If value sharing is disabled, only infinite recursion protection is done.

If string referencing is enabled and this is the first use of this method in encoding a value, all repeated references to long strings and bytearrays will be replaced with references to the first occurrence of those arrays.

If string referencing is disabled, all strings and bytearrays will be encoded directly.

cbor2.encoder.**dump**(*obj*, *fp*, ***kwargs*)

Serialize an object to a file.

Parameters

- **obj** – the object to serialize
- **fp** – a file-like object
- **kwargs** – keyword arguments passed to *CBOREncoder*

cbor2.encoder.**dumps**(*obj*, ***kwargs*)

Serialize an object to a bytestring.

Parameters

- **obj** – the object to serialize
- **kwargs** – keyword arguments passed to *CBOREncoder*

Returns the serialized output

Return type bytes

cbor2.encoder.**shareable_encoder**(*func*)

Wrap the given encoder function to gracefully handle cyclic data structures.

If value sharing is enabled, this marks the given value shared in the datastream on the first call. If the value has already been passed to this method, a reference marker is instead written to the data stream and the wrapped function is not called.

If value sharing is disabled, only infinite recursion protection is done.

5.1.5 `cbor2.decoder`

class `cbor2.decoder.`**`CBORDecoder`**(*fp*, *tag_hook=None*, *object_hook=None*, *str_errors='strict'*)

The CBORDecoder class implements a fully featured CBOR decoder with several extensions for handling shared references, big integers, rational numbers and so on. Typically the class is not used directly, but the `load()` and `loads()` functions are called to indirectly construct and use the class.

When the class is constructed manually, the main entry points are `decode()` and `decode_from_bytes()`.

Parameters

- **`tag_hook`** – callable that takes 2 arguments: the decoder instance, and the `CBORTag` to be decoded. This callback is invoked for any tags for which there is no built-in decoder. The return value is substituted for the `CBORTag` object in the deserialized output
- **`object_hook`** – callable that takes 2 arguments: the decoder instance, and a dictionary. This callback is invoked for each deserialized `dict` object. The return value is substituted for the dict in the deserialized output.

`decode`()

Decode the next value from the stream.

Raises *`CBORDecodeError`* – if there is any problem decoding the stream

`decode_from_bytes`(*buf*)

Wrap the given bytestring as a file and call `decode()` with it as the argument.

This method was intended to be used from the `tag_hook` hook when an object needs to be decoded separately from the rest but while still taking advantage of the shared value registry.

property immutable

Used by decoders to check if the calling context requires an immutable type. Object_hook or tag_hook should raise an exception if this flag is set unless the result can be safely used as a dict key.

`read`(*amount*)

Read bytes from the data stream.

Parameters **`amount`** (*`int`*) – the number of bytes to read

`set_shareable`(*value*)

Set the shareable value for the last encountered shared value marker, if any. If the current shared index is `None`, nothing is done.

Parameters **`value`** – the shared value

Returns the shared value to permit chaining

`cbor2.decoder.`**`load`**(*fp*, ***kwargs*)

Deserialize an object from an open file.

Parameters

- **`fp`** – the input file (any file-like object)
- **`kwargs`** – keyword arguments passed to *`CBORDecoder`*

Returns the deserialized object

`cbor2.decoder.`**`loads`**(*s*, ***kwargs*)

Deserialize an object from a bytestring.

Parameters

- **s** (*bytes*) – the bytestring to deserialize
- **kwargs** – keyword arguments passed to *CBORDecoder*

Returns the deserialized object

5.1.6 cbor2.types

exception cbor2.types.**CBORDecodeEOF**

Raised when decoding unexpectedly reaches EOF.

exception cbor2.types.**CBORDecodeError**

Raised for exceptions occurring during CBOR decoding.

exception cbor2.types.**CBORDecodeValueError**

Raised when the CBOR stream being decoded contains an invalid value.

exception cbor2.types.**CBOREncodeError**

Raised for exceptions occurring during CBOR encoding.

exception cbor2.types.**CBOREncodeTypeError**

Raised when attempting to encode a type that cannot be serialized.

exception cbor2.types.**CBOREncodeValueError**

Raised when the CBOR encoder encounters an invalid value.

exception cbor2.types.**CBORError**

Base class for errors that occur during CBOR encoding or decoding.

class cbor2.types.**CBORSimpleValue**(*value*)

Represents a CBOR "simple value".

Parameters **value** (*int*) – the value (0-255)

class cbor2.types.**CBORTag**(*tag*, *value*)

Represents a CBOR semantic tag.

Parameters

- **tag** (*int*) – tag number
- **value** – encapsulated value (any object)

class cbor2.types.**FrozenDict**(**args*, ***kwargs*)

A hashable, immutable mapping type.

The arguments to `FrozenDict` are processed just like those to `dict`.

cbor2.types.**undefined = undefined**

Represents the "undefined" value.

- API reference

PYTHON MODULE INDEX

INDEX