pyrlp Documentation

Release 0.1-dev

jnnk

Dec 06, 2021

Contents

pyrlp is a package for encoding and decoding data to and from *recursive length prefix encoding* (*RLP*). This format finds widely spread use in the Ethereum world.

CHAPTER 1

Quickstart

```
>>> import rlp
>>> from rlp.sedes import big_endian_int, text, List
```

```
>>> rlp.encode(1234)
b'\x82\x04\xd2'
>>> rlp.decode(b'\x82\x04\xd2', big_endian_int)
1234
```

```
>>> rlp.encode([1, [2, []]])
b'\xc4\x01\xc2\x02\xc0'
>>> list_sedes = List([big_endian_int, [big_endian_int, []]])
>>> rlp.decode(b'\xc4\x01\xc2\x02\xc0', list_sedes)
(1, (2, ()))
```

```
>>> class Tx(rlp.Serializable):
...     fields = [
...         ('from', text),
...         ('to', text),
...         ('amount', big_endian_int)
...     ]
...
>>> tx = Tx('me', 'you', 255)
>>> rlp.encode(tx)
b'\xc9\x82me\x83you\x81\xff'
>>> rlp.decode(b'\xc9\x82me\x83you\x81\xff', Tx) == tx
True
```

CHAPTER 2

Tutorial

2.1 Basics

There are two types of fundamental items one can encode in RLP:

1) Strings of bytes
2) Lists of other items

In this package, byte strings are represented either as Python strings or as `bytearrays`. Lists can be any sequence, e.g. `lists` or `tuples`. To encode these kinds of objects, use `rlp.encode()`:

```
>>> from rlp import encode
>>> encode('ethereum')
b'\x88ethereum'
>>> encode('')
b'\x80'
>>> encode('Lorem ipsum dolor sit amet, consetetur sadipscing elitr.')
b'\xb88Lorem ipsum dolor sit amet, consetetur sadipscing elitr.'
>>> encode([])
b'\xc0'
>>> encode(['this', ['is', ('a', ('nested', 'list', []))]])
b'\xd9\x84this\xd3\x82is\xcfa\xcd\x86nested\x84list\xc0'
```

Decoding is just as simple:

```
>>> from rlp import decode
>>> decode(b'\x88ethereum')
b'ethereum'
>>> decode(b'\x80')
b''
>>> decode(b'\xc0')
[]
>>> decode(b'\xd9\x84this\xd3\x82is\xcfa\xcd\x86nested\x84list\xc0')
[b'this', [b'is', [b'a', [b'nested', b'list', []]]]]
```

Now, what if we want to encode a different object, say, an integer? Let's try:

```
>>> encode(1503)
b'\x82\x05\xdf'
>>> decode(b'\x82\x05\xdf')
b'\x05\xdf'
```

Oops, what happened? Encoding worked fine, but `rlp.decode()` refused to give an integer back. The reason is that RLP is typeless. It doesn't know if the encoded data represents a number, a string, or a more complicated object. It only distinguishes between byte strings and lists. Therefore, *pyrlp* guesses how to serialize the object into a byte string (here, in big endian notation). When encoded however, the type information is lost and `rlp.decode()` returned the result in its most generic form, as a string. Thus, what we need to do is deserialize the result afterwards.

2.2 Sedes objects

Serialization and its couterpart, deserialization, is done by, what we call, *sedes objects* (borrowing from the word "codec"). For integers, the sedes *`rlp.sedes.big_endian_int`* is in charge. To decode our integer, we can pass this sedes to `rlp.decode()`:

```
>>> from rlp.sedes import big_endian_int
>>> decode(b'\x82\x05\xdf', big_endian_int)
1503
```

For unicode strings, there's the sedes *`rlp.sedes.binary`*, which uses UTF-8 to convert to and from byte strings:

```
>>> from rlp.sedes import binary
>>> encode(u'Ðapp')
b'\x85\xc3\x90app'
>>> decode(b'\x85\xc3\x90app', binary)
b'\xc3\x90app'
>>> print(decode(b'\x85\xc3\x90app', binary).decode('utf-8'))
Ðapp
```

Lists are a bit more difficult as they can contain arbitrarily complex combinations of types. Therefore, we need to create a sedes object specific for each list type. As base class for this we can use `rlp.sedes.List`:

```
>>> from rlp.sedes import List
>>> encode([5, 'fdsa', 0])
b'\xc7\x05\x84fdsa\x80'
>>> sedes = List([big_endian_int, binary, big_endian_int])
>>> decode(b'\xc7\x05\x84fdsa\x80', sedes)
(5, b'fdsa', 0)
```

Unsurprisingly, it is also possible to nest `rlp.List` objects:

```
>>> inner = List([binary, binary])
>>> outer = List([inner, inner, inner])
>>> decode(encode(['asdf', 'fdsa']), inner)
(b'asdf', b'fdsa')
>>> decode(encode([['a1', 'a2'], ['b1', 'b2'], ['c1', 'c2']]), outer)
((b'a1', b'a2'), (b'b1', b'b2'), (b'c1', b'c2'))
```

2.3 What Sedes Objects Actually Are

We saw how to use sedes objects, but what exactly are they? They are characterized by providing the following three member functions:

- `serializable(obj)`
- `serialize(obj)`
- `deserialize(serial)`

The latter two are used to convert between a Python object and its representation as byte strings or sequences. The former one may be called by `rlp.encode()` to infer which sedes object to use for a given object (see *Sedes Inference*).

For basic types, the sedes object is usually a module (e.g. *rlp.sedes.big_endian_int* and *rlp.sedes.binary*). Instances of `rlp.sedes.List` provide the sedes interface too, as well as the class `rlp.Serializable` which is discussed in the following section.

2.4 Encoding Custom Objects

Often, we want to encode our own objects in RLP. Examples from the Ethereum world are transactions, blocks or anything send over the Wire. With *pyrlp*, this is as easy as subclassing `rlp.Serializable`:

```
>>> import rlp
>>> class Transaction(rlp.Serializable):
...     fields = (
...         ('sender', binary),
...         ('receiver', binary),
...         ('amount', big_endian_int)
...     )
```

The class attribute `fields` is a sequence of 2-tuples defining the field names and the corresponding sedes. For each name an instance attribute is created, that can conveniently be initialized with `__init__()`:

```
>>> tx1 = Transaction(b'me', b'you', 255)
>>> tx2 = Transaction(amount=255, sender=b'you', receiver=b'me')
>>> tx1.amount
255
```

At serialization, the field names are dropped and the object is converted to a list, where the provided sedes objects are used to serialize the object attributes:

```
>>> Transaction.serialize(tx1)
[b'me', b'you', b'\xff']
>>> tx1 == Transaction.deserialize([b'me', b'you', b'\xff'])
True
```

As we can see, each subclass of `rlp.Serializable` implements the sedes responsible for its instances. Therefore, we can use `rlp.encode()` and `rlp.decode()` as expected:

```
>>> encode(tx1)
b'\xc9\x82me\x83you\x81\xff'
>>> decode(b'\xc9\x82me\x83you\x81\xff', Transaction) == tx1
True
```

2.5 Sedes Inference

As we have seen, `rlp.encode()` (or, rather, `rlp.infer_sedes()`) tries to guess a sedes capable of serializing the object before encoding. In this process, it follows the following steps:

1) Check if the object's class is a sedes object (like every subclass of `rlp.Serializable`). If so, its class is the sedes.
2) Check if one of the entries in `rlp.sedes.sedes_list` can serialize the object (via `serializable(obj)`). If so, this is the sedes.
3) Check if the object is a sequence. If so, build a `rlp.sedes.List` by recursively infering a sedes for each of its elements.
4) If none of these steps was successful, sedes inference has failed.

If you have build your own basic sedes (e.g. for `dicts` or `floats`), you might want to hook in at step 2 and add it to `rlp.sedes.sedes_list`, whereby it will be automatically be used by `rlp.encode()`.

2.6 Further Reading

This was basically everything there is to about this package. The technical specification of RLP can be found either in the Ethereum wiki or in Appendix B of Gavin Woods Yellow Paper. For more detailed information about this package, have a look at the *API Reference* or the source code.

CHAPTER 3

API Reference

3.1 Functions

3.2 Sedes Objects

`rlp.sedes.`**`raw`**
A sedes object that does nothing. Thus, it can serialize everything that can be directly encoded in RLP (nested lists of strings). This sedes can be used as a placeholder when deserializing larger structures.

`rlp.sedes.`**`binary`**
A sedes object for binary data of arbitrary length (an instance of `rlp.sedes.Binary` with default arguments).

`rlp.sedes.`**`boolean`**
A sedes object for boolean types.

`rlp.sedes.`**`text`**
A sedes object for utf encoded text data of arbitrary length (an instance of `rlp.sedes.Text` with default arguments).

`rlp.sedes.`**`big_endian_int`**
A sedes object for integers encoded in big endian without any leading zeros (an instance of `rlp.sedes.BigEndianInt` with default arguments).

3.3 Exceptions

CHAPTER 4

Release Notes

4.1 Unreleased (latest source)

- *repr()* now returns an evaluatable string, like `MyRLPObj(my_int_field=1, my_str_field="a_str")` - #117

4.2 0.4.8

- Implement `Serializable.make_mutable` and `rlp.sedes.make_mutable` API.
- Add `mutable` flag to `Serializable.deserialize` to allow deserialization into mutable objects.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

R