# pyrlp Documentation

***Release 0.1-dev***

**jnnk**

March 04, 2015

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, ListSedes

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

>>> rlp.encode([1, [2, []]])
'\xc4\x01\xc2\x02\xc0'
'\xc5\x01\xc3\x02\xc1\x03'
>>> list_sedes = ListSedes([big_endian_int, [big_endian_int, []]])
>>> rlp.decode('\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)
'\xc9\x82me\x83you\x81\xff'
>>> rlp.decode('\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')
'\x88ethereum'
>>> encode('')
'\x80'
>>> encode('Lorem ipsum dolor sit amet, consetetur sadipscing elitr.')
'\xb88Lorem ipsum dolor sit amet, consetetur sadipscing elitr.'
>>> encode([])
'\xc0'
>>> encode(['this', ['is', ('a', ('nested', 'list', []))]])
'\xd9\x84this\xd3\x82is\xcfa\xcd\x86nested\x84list\xc0'
```

Decoding is just as simple:

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

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

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

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

```
>>> from rlp.text import text
>>> encode(u'Ðapp')
'\x85\xc3\x90app'
>>> decode('\x85\xc3\x90app', text)
u'\xd0app'
>>> print decode('\x85\xc3\x90app', text)
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.ListSedes`:

```
>>> from rlp.sedes import ListSedes
>>> encode([5, 'fdsa', 0])
'\xc7\x05\x84fdsa\x00'
>>> sedes = ListSedes([big_endian_int, text, big_endian_int])
>>> decode('\xc7\x05\x84fdsa\x00', sedes)
[5, u'fdsa', 0]
```

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

```
>>> inner = ListSedes([text, text])
>>> outer = ListSedes([inner, inner, inner])
>>> decode(encode(['asdf', 'fdsa']), inner)
[u'asdf', u'fdsa']
>>> decode(encode([['a1', 'a2'], ['b1', 'b2'], ['c1', 'c2']]), outer)
[[u'a1', u'a2'], [u'b1', u'b2'], [u'c1', u'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.text`). Instances of `rlp.sedes.ListSedes` 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', text),
...         ('receiver', text),
...         ('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('me', 'you', 255)
>>> tx2 = Transaction(amount=255, sender='you', receiver='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)
['me', 'you', '\xff']
>>> tx1 == Transaction.deserialize(['me', 'you', '\xff'])
True
>>> Transaction.serializable(tx1)
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)
'\xc9\x82me\x83you\x81\xff'
>>> decode('\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.ListSedes` 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

rlp.**encode**(*obj*, *sedes=None*, *infer_serializer=True*)
Encode a Python object in RLP format.

By default, the object is serialized in a suitable way first (using `rlp.infer_sedes()`) and then encoded. Serialization can be explicitly suppressed by setting *infer_serializer* to `False` and not passing an alternative as *sedes*.

**Parameters**

- **sedes** – an object implementing a function `serialize(obj)` which will be used to serialize `obj` before encoding, or `None` to use the infered one (if any)
- **infer_serializer** – if `True` an appropriate serializer will be selected using `rlp.infer_sedes()` to serialize *obj* before encoding

**Returns** the RLP encoded item

**Raises** `rlp.EncodingError` in the rather unlikely case that the item is too big to encode (will not happen)

**Raises** `rlp.SerializationError` if the serialization fails

rlp.**decode**(*rlp*, *sedes=None*, ***kwargs*)
Decode an RLP encoded object.

**Parameters**

- **sedes** – an object implementing a function `deserialize(code)` which will be applied after decoding, or `None` if no deserialization should be performed
- ****kwargs** – additional keyword arguments that will be passed to the deserializer

**Returns** the decoded and maybe deserialized Python object

**Raises** `rlp.DecodingError` if the input string does not end after the root item

**Raises** `rlp.DeserializationError` if the deserialization fails

rlp.**decode_lazy**(*rlp*, *sedes=None*, ***sedes_kwargs*)
Decode an RLP encoded object in a lazy fashion.

If the encoded object is a bytestring, this function acts similar to `rlp.decode()`. If it is a list however, a `LazyList` is returned instead. This object will decode the string lazily, avoiding both horizontal and vertical traversing as much as possible.

The way *sedes* is applied depends on the decoded object: If it is a string *sedes* deserializes it as a whole; if it is a list, each element is deserialized individually. In both cases, *sedes_kwargs* are passed on. Note that, if a deserializer is used, only "horizontal" but not "vertical lazyness" can be preserved.

**Parameters**

- **rlp** – the RLP string to decode
- **sedes** – an object implementing a method `deserialize(code)` which is used as described above, or `None` if no deserialization should be performed
- ****sedes_kwargs** – additional keyword arguments that will be passed to the deserializers

**Returns** either the already decoded and deserialized object (if encoded as a string) or an instance of `rlp.LazyList`

**class** `rlp.`**`LazyList`**(*rlp*, *start*, *end*, *sedes=None*, ***sedes_kwargs*)
A RLP encoded list which decodes itself when necessary.

Both indexing with positive indices and iterating are supported. Getting the length with `len()` is possible as well but requires full horizontal encoding.

**Parameters**

- **rlp** – the rlp string in which the list is encoded
- **start** – the position of the first payload byte of the encoded list
- **end** – the position of the last payload byte of the encoded list
- **sedes** – a sedes object which deserializes each element of the list, or `None` for no deserialization
- ****sedes_kwargs** – keyword arguments which will be passed on to the deserializer

`rlp.`**`infer_sedes`**(*obj*)
Try to find a sedes objects suitable for a given Python object.

The sedes objects considered are *obj*'s class, *big_endian_int* and *binary*. If *obj* is a sequence, a `ListSedes` will be constructed recursively.

**Parameters** **obj** – the python object for which to find a sedes object

**Raises** `TypeError` if no appropriate sedes could be found

## 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.

**class** `rlp.sedes.`**`Binary`**(*min_length=None*, *max_length=None*, *allow_empty=False*)
A sedes object for binary data of certain length.

**Parameters**

- **min_length** – the minimal length in bytes or *None* for no lower limit
- **max_length** – the maximal length in bytes or *None* for no upper limit
- **allow_empty** – if true, empty strings are considered valid even if a minimum length is required otherwise

**classmethod** **fixed_length**(*l*, *allow_empty=False*)
Create a sedes for binary data with exactly *l* bytes.

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

**class** rlp.sedes.**BigEndianInt**(*l=None*)
A sedes for big endian integers.

**Parameters** **l** – the size of the serialized representation in bytes or *None* to use the shortest possible one

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).

**class** rlp.sedes.**List**(*elements=[]*)
A sedes for lists, implemented as a list of other sedes objects.

**class** rlp.sedes.**CountableList**(*element_sedes*)
A sedes for lists of arbitrary length.

**Parameters** **element_sedes** – when (de-)serializing a list, this sedes will be applied to all of its elements

**class** rlp.**Serializable**(**args*, ***kwargs*)
Base class for objects which can be serialized into RLP lists.

`fields` defines which attributes are serialized and how this is done. It is expected to be an ordered sequence of 2-tuples `(name, sedes)`. Here, `name` is the name of an attribute and `sedes` is the sedes object that will be used to serialize the corresponding attribute. The object as a whole is then serialized as a list of those fields.

**Variables** **fields** – a list of 2-tuples `(name, sedes)` where `name` is a string corresponding to an attribute and `sedes` is the sedes object used for (de)serializing the attribute.

**Parameters**

- ***args** – initial values for the first attributes defined via `fields`
- ****kwargs** – initial values for all attributes not initialized via positional arguments

**classmethod** **exclude**(*excluded_fields*)
Create a new sedes considering only a reduced set of fields.

## 3.3 Exceptions

**exception** rlp.**RLPException**
Base class for exceptions raised by this package.

**exception** rlp.**EncodingError**(*message*, *obj*)
Exception raised if encoding fails.

**Variables** **obj** – the object that could not be encoded

**exception** rlp.**DecodingError**(*message*, *rlp*)
Exception raised if decoding fails.

**Variables** **rlp** – the RLP string that could not be decoded

**exception** rlp.**SerializationError**(*message*, *obj*)
Exception raised if serialization fails.

**Variables** **obj** – the object that could not be serialized

**exception** rlp.**DeserializationError**(*message*, *serial*)
Exception raised if deserialization fails.

**Variables** **serial** – the decoded RLP string that could not be deserialized

CHAPTER 4

# Indices and tables

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

## B



## C



## D



## E



## F



## I



## L



## R



## S