pyrlp Documentation

Release 4.0.1

The Ethereum Foundation

Apr 24, 2024

CONTENTS

A package for Recursive Length Prefix encoding and decoding

CHAPTER

ONE

CONTENTS

1.1 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
```

1.2 Tutorial

1.2.1 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.

1.2.2 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'))
```

1.2.3 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.

1.2.4 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
```

1.2.5 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()*.

1.2.6 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.

1.3 API Reference

1.3.1 Functions

`rlp.encode`(*obj*, *sedes=None*, *infer_serializer=True*, *cache=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*.

If *obj* has an attribute `_cached_rlp` (as, notably, `rlp.Serializable`) and its value is not *None*, this value is returned bypassing serialization and encoding, unless *sedes* is given (as the cache is assumed to refer to the standard serialization which can be replaced by specifying *sedes*).

If *obj* is a `rlp.Serializable` and *cache* is true, the result of the encoding will be stored in `_cached_rlp` if it is empty.

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
- **cache** – cache the return value in *obj._cached_rlp* if possible (default *True*)

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*, *strict=True*, *recursive_cache=False*, ***kwargs*)

Decode an RLP encoded object.

If the deserialized result *obj* has an attribute `_cached_rlp` (e.g. if *sedes* is a subclass of `rlp.Serializable`) it will be set to *rlp*, which will improve performance on subsequent `rlp.encode()` calls. Bear in mind however that *obj* needs to make sure that this value is updated whenever one of its fields changes or prevent such changes entirely (`rlp.sedes.Serializable` does the latter).

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

- **strict** – if false inputs that are longer than necessary don't cause an exception

Returns
the decoded and maybe deserialized Python object

Raises
rlp.DecodingError if the input string does not end after the root item and *strict* is true

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 *rlp.sedes.List* 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

1.3.2 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(*length*, *allow_empty=False*)

Create a sedes for binary data with exactly *length* 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.**Boolean**

A sedes for booleans

rlp.sedes.**boolean**

A sedes object for boolean types.

class rlp.sedes.**Text**(*min_length=None*, *max_length=None*, *allow_empty=False*, *encoding='utf8'*)

A sedes object for encoded text data of certain length.

Parameters

- **min_length** – the minimal length in encoded characters or *None* for no lower limit
- **max_length** – the maximal length in encoded characters 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(*length*, *allow_empty=False*)

Create a sedes for text data with exactly *length* encoded characters.

rlp.sedes.**text**

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

class rlp.sedes.**BigEndianInt**(*length=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=None*, *strict=True*)

A sedes for lists, implemented as a list of other sedes objects.

Parameters

strict – If true (de)serializing lists that have a length not matching the sedes length will result in an error. If false (de)serialization will stop as soon as either one of the lists runs out of elements.

class rlp.sedes.**CountableList**(*element_sedes*, *max_length=None*)

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
- **max_length** – maximum number of allowed elements, or *None* for no limit

class rlp.**Serializable**(**args*, ***kwargs*)

The base class for serializable objects.

1.3.3 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

1.4 rlp package

1.4.1 Subpackages

rlp.sedes package

Submodules

rlp.sedes.big_endian_int module

class rlp.sedes.big_endian_int.**BigEndianInt**(*length=None*)

Bases: object

A sedes for big endian integers.

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

deserialize(*serial*)

serialize(*obj*)

rlp.sedes.binary module

class rlp.sedes.binary.**Binary**(*min_length=None*, *max_length=None*, *allow_empty=False*)

Bases: object

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

deserialize(*serial*)

classmethod fixed_length(*length*, *allow_empty=False*)

Create a sedes for binary data with exactly *length* bytes.

is_valid_length(*length*)

classmethod is_valid_type(*obj*)

serialize(*obj*)

rlp.sedes.boolean module

class rlp.sedes.boolean.**Boolean**

Bases: object

A sedes for booleans

deserialize(*serial*)

serialize(*obj*)

rlp.sedes.lists module

Module for sedes objects that use lists as serialization format.

class rlp.sedes.lists.**CountableList**(*element_sedes*, *max_length=None*)

Bases: object

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
- **max_length** – maximum number of allowed elements, or *None* for no limit

deserialize(*serial*)

serialize(*obj*)

class rlp.sedes.lists.**List**(*elements=None*, *strict=True*)

Bases: list

A sedes for lists, implemented as a list of other sedes objects.

Parameters

strict – If true (de)serializing lists that have a length not matching the sedes length will result in an error. If false (de)serialization will stop as soon as either one of the lists runs out of elements.

deserialize(*serial*)

serialize(*obj*)

rlp.sedes.lists.**is_sedes**(*obj*)

Check if *obj* is a sedes object.

A sedes object is characterized by having the methods *serialize(obj)* and *deserialize(serial)*.

rlp.sedes.lists.**is_sequence**(*obj*)

Check if *obj* is a sequence, but not a string or bytes.

rlp.sedes.raw module

A sedes that does nothing. Thus, everything that can be directly encoded by RLP is serializable. This sedes can be used as a placeholder when deserializing larger structures.

rlp.sedes.raw.**deserialize**(*serial*)

rlp.sedes.raw.**serializable**(*obj*)

rlp.sedes.raw.**serialize**(*obj*)

rlp.sedes.serializable module

class rlp.sedes.serializable.**BaseChangeset**(*obj*, *changes=None*)

Bases: object

build_rlp()

close()

commit()

open()

class rlp.sedes.serializable.**BaseSerializable**(**args*, ***kwargs*)

Bases: Sequence

as_dict()

build_changeset(**args*, ***kwargs*)

copy(**args*, ***kwargs*)

classmethod deserialize(*serial*, ***extra_kwargs*)

classmethod serialize(*obj*)

rlp.sedes.serializable.**Changeset**(*obj*, *changes*)

class rlp.sedes.serializable.**ChangesetField**(*field*)

Bases: object

field = None

class rlp.sedes.serializable.**ChangesetState**(*value*)

Bases: Enum

An enumeration.

CLOSED = 'CLOSED'

INITIALIZED = 'INITIALIZED'

OPEN = 'OPEN'

class rlp.sedes.serializable.**MetaBase**

Bases: object

field_attrs = None

field_names = None

fields = None

sedes = None

class rlp.sedes.serializable.**Serializable**(**args*, ***kwargs*)

Bases: *BaseSerializable*

The base class for serializable objects.

class rlp.sedes.serializable.**SerializableBase**(*name*, *bases*, *attrs*)

Bases: ABCMeta

rlp.sedes.serializable.**make_immutable**(*value*)

rlp.sedes.serializable.**merge_args_to_kwargs**(*args*, *kwargs*, *arg_names*, *allow_missing=False*)

rlp.sedes.serializable.**merge_kwargs_to_args**(*args*, *kwargs*, *arg_names*, *allow_missing=False*)

rlp.sedes.serializable.**validate_args_and_kwargs**(*args*, *kwargs*, *arg_names*, *allow_missing=False*)

rlp.sedes.text module

class rlp.sedes.text.**Text**(*min_length=None*, *max_length=None*, *allow_empty=False*, *encoding='utf8'*)

Bases: object

A sedes object for encoded text data of certain length.

Parameters

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

deserialize(*serial*)

classmethod fixed_length(*length*, *allow_empty=False*)

Create a sedes for text data with exactly *length* encoded characters.

is_valid_length(*length*)

classmethod is_valid_type(*obj*)

serialize(*obj*)

Module contents

1.4.2 Submodules

1.4.3 rlp.atomic module

class rlp.atomic.**Atomic**

Bases: object

ABC for objects that can be RLP encoded as is.

1.4.4 rlp.codec module

rlp.codec.**consume_item**(*rlp*, *start*)

Read an item from an RLP string.

Parameters

- **rlp** – the rlp string to read from
- **start** – the position at which to start reading

Returns

a tuple `(item, per_item_rlp, end)`, where `item` is the read item, per_item_rlp is a list containing the RLP encoding of each item and `end` is the position of the first unprocessed byte

rlp.codec.**consume_length_prefix**(*rlp*, *start*)

Read a length prefix from an RLP string.

Parameters

- **rlp** – the rlp byte string to read from
- **start** – the position at which to start reading

Returns

a tuple `(prefix, type, length, end)`, where `type` is either `str` or `list` depending on the type of the following payload, `length` is the length of the payload in bytes, and `end` is the position of the first payload byte in the rlp string

rlp.codec.**consume_payload**(*rlp*, *prefix*, *start*, *type_*, *length*)

Read the payload of an item from an RLP string.

Parameters

- **rlp** – the rlp string to read from
- **type** – the type of the payload (`bytes` or `list`)
- **start** – the position at which to start reading
- **length** – the length of the payload in bytes

Returns

a tuple `(item, per_item_rlp, end)`, where `item` is the read item, per_item_rlp is a list containing the RLP encoding of each item and `end` is the position of the first unprocessed byte

rlp.codec.**decode**(*rlp*, *sedes=None*, *strict=True*, *recursive_cache=False*, ***kwargs*)

Decode an RLP encoded object.

If the deserialized result *obj* has an attribute `_cached_rlp` (e.g. if *sedes* is a subclass of *rlp.Serializable*) it will be set to *rlp*, which will improve performance on subsequent *rlp.encode()* calls. Bear in mind however that *obj* needs to make sure that this value is updated whenever one of its fields changes or prevent such changes entirely (`rlp.sedes.Serializable` does the latter).

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
- **strict** – if false inputs that are longer than necessary don't cause an exception

Returns
the decoded and maybe deserialized Python object

Raises
rlp.DecodingError if the input string does not end after the root item and *strict* is true

Raises
rlp.DeserializationError if the deserialization fails

rlp.codec.**decode_raw**(*item*, *strict*, *_*)

rlp.codec.**encode**(*obj*, *sedes=None*, *infer_serializer=True*, *cache=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*.

If *obj* has an attribute `_cached_rlp` (as, notably, *rlp.Serializable*) and its value is not *None*, this value is returned bypassing serialization and encoding, unless *sedes* is given (as the cache is assumed to refer to the standard serialization which can be replaced by specifying *sedes*).

If *obj* is a *rlp.Serializable* and *cache* is true, the result of the encoding will be stored in `_cached_rlp` if it is empty.

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
- **cache** – cache the return value in *obj._cached_rlp* if possible (default *True*)

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.codec.**encode_raw**(*item*)

RLP encode (a nested sequence of) `Atomic`s.

rlp.codec.**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 *rlp.sedes.List* 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

rlp.codec.**length_prefix**(*length*, *offset*)

Construct the prefix to lists or strings denoting their length.

Parameters

- **length** – the length of the item in bytes
- **offset** – `0x80` when encoding raw bytes, `0xc0` when encoding a list

1.4.5 rlp.exceptions module

exception rlp.exceptions.**DecodingError**(*message*, *rlp*)

Bases: *RLPException*

Exception raised if decoding fails.

Variables
rlp – the RLP string that could not be decoded

exception rlp.exceptions.**DeserializationError**(*message*, *serial*)

Bases: *RLPException*

Exception raised if deserialization fails.

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

exception rlp.exceptions.**EncodingError**(*message*, *obj*)

Bases: *RLPException*

Exception raised if encoding fails.

Variables
obj – the object that could not be encoded

exception rlp.exceptions.**ListDeserializationError**(*message=None*, *serial=None*, *element_exception=None*, *index=None*)

Bases: *DeserializationError*

Exception raised if deserialization by a `sedes.List` fails.

Variables

- **element_exception** – the exception that occurred during the deserialization of one of the elements, or *None* if the error is unrelated to a specific element

- **index** – the index in the list that produced the error or *None* if the error is unrelated to a specific element

exception rlp.exceptions.**ListSerializationError**(*message=None*, *obj=None*, *element_exception=None*, *index=None*)

Bases: *SerializationError*

Exception raised if serialization by a `sedes.List` fails.

Variables

- **element_exception** – the exception that occurred during the serialization of one of the elements, or *None* if the error is unrelated to a specific element
- **index** – the index in the list that produced the error or *None* if the error is unrelated to a specific element

exception rlp.exceptions.**ObjectDeserializationError**(*message=None*, *serial=None*, *sedes=None*, *list_exception=None*)

Bases: *DeserializationError*

Exception raised if deserialization by a `sedes.Serializable` fails.

Variables

- **sedes** – the `sedes.Serializable` that failed
- **list_exception** – exception raised by the underlying list sedes, or *None* if no such exception has been raised
- **field** – name of the field of the object that produced the error, or *None* if no field responsible for the error

exception rlp.exceptions.**ObjectSerializationError**(*message=None*, *obj=None*, *sedes=None*, *list_exception=None*)

Bases: *SerializationError*

Exception raised if serialization of a `sedes.Serializable` object fails.

Variables

- **sedes** – the `sedes.Serializable` that failed
- **list_exception** – exception raised by the underlying list sedes, or *None* if no such exception has been raised
- **field** – name of the field of the object that produced the error, or *None* if no field responsible for the error

exception rlp.exceptions.**RLPException**

Bases: `Exception`

Base class for exceptions raised by this package.

exception rlp.exceptions.**SerializationError**(*message*, *obj*)

Bases: *RLPException*

Exception raised if serialization fails.

Variables

obj – the object that could not be serialized

1.4.6 rlp.lazy module

class rlp.lazy.**LazyList**(*rlp*, *start*, *end*, *sedes=None*, ***sedes_kwargs*)

Bases: Sequence

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

next()

rlp.lazy.**consume_item_lazy**(*rlp*, *start*)

Read an item from an RLP string lazily.

If the length prefix announces a string, the string is read; if it announces a list, a *LazyList* is created.

Parameters

- **rlp** – the rlp string to read from
- **start** – the position at which to start reading

Returns

a tuple `(item, end)` where `item` is the read string or a *LazyList* and `end` is the position of the first unprocessed byte.

rlp.lazy.**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*

rlp.lazy.**peek**(*rlp*, *index*, *sedes=None*)

Get a specific element from an rlp encoded nested list.

This function uses *rlp.decode_lazy()* and, thus, decodes only the necessary parts of the string.

Usage example:

```
>>> import rlp
>>> rlpdata = rlp.encode([1, 2, [3, [4, 5]]])
>>> rlp.peek(rlpdata, 0, rlp.sedes.big_endian_int)
1
>>> rlp.peek(rlpdata, [2, 0], rlp.sedes.big_endian_int)
3
```

Parameters

- **rlp** – the rlp string
- **index** – the index of the element to peek at (can be a list for nested data)
- **sedes** – a sedes used to deserialize the peeked at object, or *None* if no deserialization should be performed

Raises

IndexError if *index* is invalid (out of range or too many levels)

1.4.7 rlp.utils module

1.4.8 Module contents

1.5 pyrlp v4.0.1 (2024-04-24)

1.5.1 Internal Changes - for pyrlp Contributors

- Add python 3.12 support, `rust-backend` now works with python 3.11 and 3.12 (#150)

1.5.2 Miscellaneous Changes

- #151

1.6 pyrlp v4.0.0 (2023-11-29)

1.6.1 Features

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

1.6.2 Internal Changes - for pyrlp Contributors

- Convert `.format` strings to `f-strings` (#144)
- Add `autoflake` linting and move config to `pyproject.toml` (#145)

Release Notes

1.7 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

TWO

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

r

INDEX

M

N

O

P

R

S

T

V