Ethereum Contract Interface (ABI) Utility Documentation

Release 2.2.0

The Ethereum Foundation

Jul 20, 2022

Contents

The `eth-abi` library provides low level utilities for converting python values to and from solidity's binary ABI format.

For detailed information on ABI types and their encodings, refer to the solidity ABI specification.

CHAPTER 1

Credit

Though much of the code has been revised, the `eth-abi` library was originally extracted from the `pyethereum` library which was authored by Vitalik Buterin.

CHAPTER 2

Table of Contents

2.1 Encoding

2.1.1 Encoding Python Values

Python values can be encoded into binary values for a given ABI type as follows:

```
>>> from eth_abi import encode_single, encode_abi

>>> encode_single('uint256', 12345)
b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
↪'

>>> encode_single('(bytes32,bytes32)', [b'a', b'b'])
b
↪'a\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪'

>>> encode_abi(['bytes32', 'bytes32'], [b'a', b'b'])
b
↪'a\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪'
```

The *encode_single* function can be used to perform any encoding operation from a python value to a binary ABI value for an ABI type. As is seen in the example above, *encode_single* supports encoding of tuple ABI values which can be used to encode sequences of python values in a single binary payload.

The *encode_abi* function provides an alternate API for encoding tuple values. It accepts a list of type strings instead of a single tuple type string. Internally, it uses the *encode_single* function to do this. Because of this redundancy, it will eventually be removed in favor of *encode_single*.

2.1.2 Checking for Encodability

It is also possible to check whether or not a certain python value is encodable for a given ABI type using *encode_single*:

```
>>> from eth_abi import is_encodable

>>> is_encodable('int', 2)
True

>>> is_encodable('int', 'foo')
False

>>> is_encodable('(int,bool)', (0, True))
True

>>> is_encodable('(int,bool)', (0, 0))
False
```

2.1.3 Non-Standard Packed Mode Encoding

Warning: Non-standard packed mode encoding is an experimental feature in the eth-abi library. Use at your own risk and please report any problems at https://github.com/ethereum/eth-abi/issues.

In certain cases, the Solidity programming language uses a non-standard packed encoding. You can encode values in this format like so:

```
>>> from eth_abi.packed import encode_single_packed, encode_abi_packed

>>> encode_single_packed('uint32', 12345)
b'\x00\x0009'

>>> encode_single_packed('(int8[],uint32)', ([1, 2, 3, 4], 12345))
b'\x01\x02\x03\x04\x00\x0009'

>>> encode_abi_packed(['int8[]', 'uint32'], ([1, 2, 3, 4], 12345))
b'\x01\x02\x03\x04\x00\x0009'
```

More information about this encoding format is available at https://solidity.readthedocs.io/en/develop/abi-spec.html#non-standard-packed-mode.

2.2 Decoding

2.2.1 Decoding ABI Values

Binary values for a given ABI type can be decoded into python values as follows:

```
>>> from eth_abi import decode_single, decode_abi

>>> decode_single('uint256', b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
↪')
```

(continues on next page)

(continued from previous page)

```
12345

>>> decode_single('(bytes1,bytes1)', b
↪'a\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
↪')
(b'a', b'b')

>>> decode_abi(['bytes1', 'bytes1'], b
↪'a\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
↪')
(b'a', b'b')
```

The *decode_single* function can be used to perform any decoding operation from a binary ABI value for an ABI type to a python value. As is seen in the example above, *decode_single* supports decoding of tuple ABI values which can be used to decode a single binary payload into a sequence of python values.

The *decode_abi* function provides an alternate API for decoding tuple values. It accepts a list of type strings instead of a single tuple type string. Internally, it uses the *decode_single* function to do this. Because of this redundancy, it will eventually be removed in favor of *decode_single*.

Both the *decode_single* and *decode_abi* functions accept either a python `bytes` or `bytearray` value to indicate the binary data to be decoded.

2.3 Registry

The `eth-abi` library uses a central registry to route encoding/decoding operations for different ABI types to an appropriate encoder/decoder callable or class. Using the registry, the coding behavior of any ABI type can be customized and additional coding behavior for new ABI types can be added.

2.3.1 Adding Simple Types

Here's an example of how you might add support for a simple "null" type using callables:

```
from eth_abi.exceptions import EncodingError, DecodingError
from eth_abi.registry import registry

# Define and register the coders
NULL_ENCODING = b'\x00' * 32

def encode_null(x):
    if x is not None:
        raise EncodingError('Unsupported value')

    return NULL_ENCODING

def decode_null(stream):
    if stream.read(32) != NULL_ENCODING:
        raise DecodingError('Not enough data or wrong data')

    return None

registry.register('null', encode_null, decode_null)
```

(continues on next page)

(continued from previous page)

```
# Try them out
from eth_abi import encode_single, decode_single

assert encode_single('null', None) == NULL_ENCODING
assert decode_single('null', NULL_ENCODING) is None

encoded_tuple = encode_single('(int,null)', (1, None))

assert encoded_tuple == b'\x00' * 31 + b'\x01' + NULL_ENCODING
assert decode_single('(int,null)', encoded_tuple) == (1, None)
```

In the above example, we define two coder callables and register them to handle exact matches against the `'null'` type string. We do this by calling `register` on the *registry* object.

When a call is made to one of the coding functions (such as `encode_single` or `decode_single`), the type string which is provided (which we'll call `query`) is sent to the registry. This `query` will be checked against every registration in the registry. Since we created a registration for the exact type string `'null'`, coding operations for that type string will be routed to the encoder and decoder which were provided by the call to `register`. This also works when the registered type string appears in a compound type as with the tuple type in the example.

Note: As a safety measure, the registry will raise an exception if more than one registration matches a query. Take care to ensure that your custom registrations don't conflict with existing ones.

2.3.2 Adding More Complex Types

Sometimes, it's convenient to register a single class to handle encodings or decodings for a range of types. For example, we shouldn't have to make separate registrations for the `'uint256'` and `'uint8'` types or for the `'(int,bool)'` and `'(int,int)'` types. For cases like this, we can make registrations for custom subclasses of `BaseEncoder` and `BaseDecoder`.

Let's say we want to modify our "null" type above so that we can specify the number of 32-byte words that the encoded null value will occupy in the data stream. We could do that in the following way:

```
from eth_abi.decoding import BaseDecoder
from eth_abi.encoding import BaseEncoder
from eth_abi.exceptions import EncodingError, DecodingError
from eth_abi.registry import registry

# Define and register the coders
NULL_ENCODING = b'\x00' * 32

class EncodeNull(BaseEncoder):
    word_width = None

    @classmethod
    def from_type_str(cls, type_str, registry):
        word_width = int(type_str[4:])
        return cls(word_width=word_width)

    def encode(self, value):
        self.validate_value(value)
        return NULL_ENCODING * self.word_width
```

(continues on next page)

(continued from previous page)

```
    def validate_value(self, value):
        if value is not None:
            raise EncodingError('Unsupported value')

class DecodeNull(BaseDecoder):
    word_width = None

    @classmethod
    def from_type_str(cls, type_str, registry):
        word_width = int(type_str[4:])
        return cls(word_width=word_width)

    def decode(self, stream):
        byts = stream.read(32 * self.word_width)
        if byts != NULL_ENCODING * self.word_width:
            raise DecodingError('Not enough data or wrong data')

        return None

registry.register(
    lambda x: x.startswith('null'),
    EncodeNull,
    DecodeNull,
    label='null',
)

# Try them out
from eth_abi import encode_single, decode_single

assert encode_single('null2', None) == NULL_ENCODING * 2
assert decode_single('null2', NULL_ENCODING * 2) is None

encoded_tuple = encode_single('(int,null2)', (1, None))

assert encoded_tuple == b'\x00' * 31 + b'\x01' + NULL_ENCODING * 2
assert decode_single('(int,null2)', encoded_tuple) == (1, None)
```

There are a few differences here from our first example. Now, we are providing a type string matcher function instead of a literal type string with our call to *register*. Also, we are not using simple callables for our coding functions. We have created two custom coder classes which inherit from *BaseEncoder* and *BaseDecoder* respectively. Additionally, we have given a label to this registration in case we want to easily delete the registration later.

The matcher function `lambda x: x.startswith('null')` accepts a query type string and returns `True` or `False` to indicate if the query should be matched with our registration. If a query is *uniquely* matched with our registration in this way, the registry then calls *from_type_str* on our `EncodeNull` or `DecodeNull` class to obtain an appropriate instance of the class based on any additional information contained in the type string. In this example, that additional information is the number that appears at the end of the type string (e.g. `'2'` in `'null2'`). Through this process, the registry can determine an encoder or decoder for any type string of the form `'null<M>'`.

There are a few more details here that are worth explaining.

Both of our coder subclasses have some similar aspects. They both have a class property `word_width`. They also have the same implementation for the *from_type_str* method. The *BaseEncoder* and *BaseDecoder* classes both inherit from *BaseCoder* which causes any keyword arguments passed to `__init__` to be used to set the value of properties on an instance if a class property with the same name is found. This is why our implementations of *from_type_str* instantiate our coder classes with the keyword argument `word_width`. Using this pattern, coder classes can describe what "settings" they support while providing an easy way to assign values to those settings.

Both of our coder classes use the same settings. The settings are initialized from the type string in the same way. Therefore, they have the same implementation for *from_type_str*. For clarity, the same `word_width` property and *from_type_str* implementation appear in both classes but they could also have been extracted out into a mixin class.

Our coder classes also implement the *BaseEncoder.encode* and *BaseDecoder.decode* methods. These methods work in the same way as the simple callable coders in our first example except that they have access to the settings which were extracted from the type string when the class was instantiated via the *from_type_str* method by the registry. This allows them to handle null values of an arbitrary width in the data stream. As with the callable coders, the *BaseEncoder.encode* and *BaseDecoder.decode* implementations are polite and raise an appropriate exception when anything goes wrong. `EncodeNull` does this via an implementation of *BaseEncoder.validate_value*. For encoder classes, it is necessary to implement this method since it is used by the *is_encodable* function to determine if a value is encodable without doing the extra work of encoding it. For certain data types, this can be more efficient than simply attempting to encode a value.

2.4 Codecs

Though the default registry can be customized by making additonal coder registrations or by unregistering existing coders (see *Registry*), sometimes a user might wish to create their own registry entirely. In that case, they can still use the usual API for encoding and decoding values (see *Encoding* and *Decoding*) with their own registry by using the *ABICodec* or *ABIEncoder* class.

2.4.1 Using a Custom Registry

Here's an example of how you might add support for a simple "null" type using a custom registry while continuing to use the porcelain encoding and decoding API:

```
from eth_abi.codec import ABICodec
from eth_abi.exceptions import EncodingError, DecodingError
from eth_abi.registry import ABIRegistry

# Define and register the coders
NULL_ENCODING = b'\x00' * 32

def encode_null(x):
    if x is not None:
        raise EncodingError('Unsupported value')

    return NULL_ENCODING

def decode_null(stream):
    if stream.read(32) != NULL_ENCODING:
        raise DecodingError('Not enough data or wrong data')

    return None

registry = ABIRegistry()
registry.register('null', encode_null, decode_null)

# Try them out
codec = ABICodec(registry)

assert codec.encode_single('null', None) == NULL_ENCODING
assert codec.decode_single('null', NULL_ENCODING) is None
```

In the above example, we define two coder callables and register them to handle exact matches against the `'null'` type string in a custom registry. For more information about coder registrations, see *Adding Simple Types*.

We then create a custom codec object with our custom registry and use this to encode and decode byte sequences. This allows us to continue using the porcelain API (described in the *Encoding* and *Decoding* sections) with our custom registry.

2.4.2 Copying an Existing Registry

Sometimes, it's more convenient to use an existing registry but with only one or two small modifications. This can be done via a registry's copying or cloning capability coupled with the use of a custom codec:

```
from eth_abi.codec import ABICodec
from eth_abi.registry import registry as default_registry

registry = default_registry.copy()
registry.unregister('address')

codec = ABICodec(registry)

try:
    codec.encode_single('address', None)
except ValueError:
    pass
else:
    # We shouldn't reach this since the above code will cause an exception
    raise Exception('unreachable')

default_codec = ABICodec(default_registry)

# The default registry is unaffected since a copy was made
assert (
    default_codec.encode_single('address', '0x' + 'ff' * 20) ==
    b'\x00' * 12 + b'\xff' * 20
)
```

2.4.3 Using a Custom Stream Class

If a user wishes to customize the behavior of the internal stream class used for decoding, they can do the following:

```
from eth_abi.codec import ABIEncoder, ABIDecoder
from eth_abi.registry import registry

class MyStream:
    # Custom behavior...
    pass

class MyDecoder(ABIDecoder):
    stream_class = MyStream

class MyCodec(ABIEncoder, MyDecoder):
    pass

codec = MyCodec(registry)
```

2.5 Nested Dynamic Arrays

The `eth-abi` library supports the Solidity ABIv2 encoding format for nested dynamic arrays. This means that values for data types such as the following are legal and encodable/decodable: `int[][]`, `string[]`, `string[2]`, etc.

> **Warning:** Though Solidity's ABIv2 has mostly been finalized, the specification is technically still in development and may change.

2.6 Grammar

The `eth-abi` library exposes its type string parsing and normalization facilities as part of its public API.

2.6.1 Parsing a Type String

Here are some examples of how you might parse a type string into a simple AST and do various operations with the results:

```
>>> from eth_abi.grammar import ABIType, BasicType, TupleType, parse

>>> tuple_type = parse('(int256,bytes,ufixed128x18,bool[])[2]')

>>> # Checking if a type is a tuple or a basic type
>>> isinstance(tuple_type, ABIType)
True
>>> isinstance(tuple_type, TupleType)
True
>>> [isinstance(i, ABIType) for i in tuple_type.components]
[True, True, True, True]
>>> [isinstance(i, BasicType) for i in tuple_type.components]
[True, True, True, True]

>>> int_type, bytes_type, ufixed_type, bool_type = tuple_type.components

>>> # Inspecting parts of types
>>> len(tuple_type.components)
4
>>> tuple_type.arrlist
((2,),)
>>> int_type.base, int_type.sub, int_type.arrlist
('int', 256, None)
>>> bytes_type.base, bytes_type.sub, bytes_type.arrlist
('bytes', None, None)
>>> ufixed_type.base, ufixed_type.sub, ufixed_type.arrlist
('ufixed', (128, 18), None)
>>> bool_type.base, bool_type.sub, bool_type.arrlist
('bool', None, ((),))

>>> # Checking for arrays or dynamicism
>>> tuple_type.is_array, tuple_type.is_dynamic
(True, True)
>>> int_type.is_array, int_type.is_dynamic
(False, False)
```

(continues on next page)

(continued from previous page)

```
>>> bytes_type.is_array, bytes_type.is_dynamic
(False, True)
>>> ufixed_type.is_array, ufixed_type.is_dynamic
(False, False)
>>> bool_type.is_array, bool_type.is_dynamic
(True, True)
```

2.6.2 Checking Types for Validity

Types can be checked for validity. For example, `uint9` is not a valid type because the bit-width of `int` types must be a multiple of `8`:

```
>>> from eth_abi.grammar import parse

>>> basic_type = parse('uint9')
>>> # The basic type is not valid because the int type's bit-width is not valid
>>> basic_type.validate()
Traceback (most recent call last):
    ...
eth_abi.exceptions.ABITypeError: For 'uint9' type at column 1 in 'uint9': integer␣
↪size must be multiple of 8

>>> tuple_type = parse('(bool,uint9)')
>>> # The tuple type is not valid because it contains an int type with an invalid bit-
↪width
>>> tuple_type.validate()
Traceback (most recent call last):
    ...
eth_abi.exceptions.ABITypeError: For 'uint9' type at column 7 in '(bool,uint9)':␣
↪integer size must be multiple of 8
```

2.6.3 Normalizing Type Strings

Type strings can be normalized to their canonical form. This amounts to converting type aliases like `uint` to `uint256` and so forth:

```
>>> from eth_abi.grammar import normalize
>>> normalize('uint')
'uint256'
>>> normalize('(uint,(ufixed,function))')
'(uint256,(ufixed128x18,bytes24))'
```

Internally, `eth-abi` will only normalize type strings just before creating coders for a type. This is done automatically such that type strings passed to `eth-abi` do not need to be normalized before hand.

2.7 Tools

The `eth_abi.tools` module provides extra resources to users of `eth-abi` that are not required for typical use. It can be installed with `pip` as an extra requirement:

```
pip install eth-abi[tools]
```

2.7.1 ABI Type Strategies

The `tools` module provides the `get_abi_strategy()` function. This function returns a hypothesis strategy (value generator) for any given ABI type specified by its canonical string representation:

```
>>> from eth_abi.tools import get_abi_strategy

>>> uint_st = get_abi_strategy('uint8')
>>> uint_st
integers(min_value=0, max_value=255)

>>> uint_list_st = get_abi_strategy('uint8[2]')
>>> uint_list_st
lists(elements=integers(min_value=0, max_value=255), min_size=2, max_size=2)

>>> fixed_st = get_abi_strategy('fixed8x1')
>>> fixed_st
decimals(min_value=-128, max_value=127, places=0).map(scale_by_Eneg1)

>>> tuple_st = get_abi_strategy('(bool,string)')
>>> tuple_st
tuples(booleans(), text())
```

Hypothesis strategies can be used to conduct property testing on contract code. For more information on property testing, visit the Hypothesis homepage or the Hypothesis readthedocs site.

2.8 API

2.8.1 eth_abi.abi module

2.8.2 eth_abi.base module

class eth_abi.base.**BaseCoder**(***kwargs*)

Bases: `object`

Base class for all encoder and decoder classes.

classmethod from_type_str(*type_str: str*, *registry*) → eth_abi.base.BaseCoder

Used by `ABIRegistry` to get an appropriate encoder or decoder instance for the given type string and type registry.

2.8.3 eth_abi.codec module

class eth_abi.codec.**ABICodec**(*registry: eth_abi.registry.ABIRegistry*)

Bases: `eth_abi.codec.ABIEncoder`, `eth_abi.codec.ABIDecoder`

class eth_abi.codec.**ABIDecoder**(*registry: eth_abi.registry.ABIRegistry*)

Bases: `eth_abi.codec.BaseABICoder`

Wraps a registry to provide last-mile decoding functionality.

decode (*types*, *data*)

decode_abi (*types: Iterable[str], data: Union[bytes, bytearray]*) → Tuple[Any, ...]

Decodes the binary value `data` as a sequence of values of the ABI types in `types` via the head-tail mechanism into a tuple of equivalent python values.

Parameters

- **types** – An iterable of string representations of the ABI types that will be used for decoding e.g. `('uint256', 'bytes[]', '(int,int)')`
- **data** – The binary value to be decoded.

Returns A tuple of equivalent python values for the ABI values represented in `data`.

decode_single (*typ: str, data: Union[bytes, bytearray]*) → Any

Decodes the binary value `data` of the ABI type `typ` into its equivalent python value.

Parameters

- **typ** – The string representation of the ABI type that will be used for decoding e.g. `'uint256'`, `'bytes[]'`, `'(int,int)'`, etc.
- **data** – The binary value to be decoded.

Returns The equivalent python value of the ABI value represented in `data`.

stream_class

alias of `eth_abi.decoding.ContextFramesBytesIO`

class eth_abi.codec.**ABIEncoder** (*registry: eth_abi.registry.ABIRegistry*)

Bases: *eth_abi.codec.BaseABICoder*

Wraps a registry to provide last-mile encoding functionality.

encode (*types*, *args*)

encode_abi (*types: Iterable[str], args: Iterable[Any]*) → bytes

Encodes the python values in `args` as a sequence of binary values of the ABI types in `types` via the head-tail mechanism.

Parameters

- **types** – An iterable of string representations of the ABI types that will be used for encoding e.g. `('uint256', 'bytes[]', '(int,int)')`
- **args** – An iterable of python values to be encoded.

Returns The head-tail encoded binary representation of the python values in `args` as values of the ABI types in `types`.

encode_single (*typ: str*, *arg: Any*) → bytes

Encodes the python value `arg` as a binary value of the ABI type `typ`.

Parameters

- **typ** – The string representation of the ABI type that will be used for encoding e.g. `'uint256'`, `'bytes[]'`, `'(int,int)'`, etc.
- **arg** – The python value to be encoded.

Returns The binary representation of the python value `arg` as a value of the ABI type `typ`.

is_encodable (*typ: str*, *arg: Any*) → bool

Determines if the python value `arg` is encodable as a value of the ABI type `typ`.

Parameters

- **typ** – A string representation for the ABI type against which the python value `arg` will be checked e.g. `'uint256'`, `'bytes[]'`, `'(int,int)'`, etc.
- **arg** – The python value whose encodability should be checked.

Returns `True` if `arg` is encodable as a value of the ABI type `typ`. Otherwise, `False`.

is_encodable_type(*typ: str*) → bool

Returns `True` if values for the ABI type `typ` can be encoded by this codec.

Parameters **typ** – A string representation for the ABI type that will be checked for encodability e.g. `'uint256'`, `'bytes[]'`, `'(int,int)'`, etc.

Returns `True` if values for `typ` can be encoded by this codec. Otherwise, `False`.

class eth_abi.codec.**BaseABICoder**(*registry: eth_abi.registry.ABIRegistry*)

Bases: `object`

Base class for porcelain coding APIs. These are classes which wrap instances of `ABIRegistry` to provide last-mile coding functionality.

2.8.4 eth_abi.decoding module

class eth_abi.decoding.**BaseDecoder**(***kwargs*)

Bases: `eth_abi.base.BaseCoder`

Base class for all decoder classes. Subclass this if you want to define a custom decoder class. Subclasses must also implement `BaseCoder.from_type_str`.

decode(*stream: eth_abi.decoding.ContextFramesBytesIO*) → Any

Decodes the given stream of bytes into a python value. Should raise `exceptions.DecodingError` if a python value cannot be decoded from the given byte stream.

2.8.5 eth_abi.encoding module

class eth_abi.encoding.**BaseEncoder**(***kwargs*)

Bases: `eth_abi.base.BaseCoder`

Base class for all encoder classes. Subclass this if you want to define a custom encoder class. Subclasses must also implement `BaseCoder.from_type_str`.

encode(*value: Any*) → bytes

Encodes the given value as a sequence of bytes. Should raise `exceptions.EncodingError` if `value` cannot be encoded.

classmethod invalidate_value(*value: Any, exc: Type[Exception] = <class 'eth_abi.exceptions.EncodingTypeError'>, msg: Optional[str] = None*) → None

Throws a standard exception for when a value is not encodable by an encoder.

validate_value(*value: Any*) → None

Checks whether or not the given value can be encoded by this encoder. If the given value cannot be encoded, must raise `exceptions.EncodingError`.

2.8.6 eth_abi.exceptions module

exception eth_abi.exceptions.**ABITypeError**

Bases: `ValueError`

Raised when a parsed ABI type has inconsistent properties; for example, when trying to parse the type string `'uint7'` (which has a bit-width that is not congruent with zero modulo eight).

exception eth_abi.exceptions.**DecodingError**
Bases: `Exception`

Base exception for any error that occurs during decoding.

exception eth_abi.exceptions.**EncodingError**
Bases: `Exception`

Base exception for any error that occurs during encoding.

exception eth_abi.exceptions.**EncodingTypeError**
Bases: *eth_abi.exceptions.EncodingError*

Raised when trying to encode a python value whose type is not supported for the output ABI type.

exception eth_abi.exceptions.**IllegalValue**
Bases: *eth_abi.exceptions.EncodingError*

Raised when trying to encode a python value with the correct type but with a value that is not considered legal for the output ABI type.

Example:

```
fixed128x19_encoder(Decimal('NaN'))  # cannot encode NaN
```

exception eth_abi.exceptions.**InsufficientDataBytes**
Bases: *eth_abi.exceptions.DecodingError*

Raised when there are insufficient data to decode a value for a given ABI type.

exception eth_abi.exceptions.**MultipleEntriesFound**
Bases: `ValueError`, *eth_abi.exceptions.PredicateMappingError*

Raised when multiple registrations are found for a type string in a registry's internal mapping. This error is non-recoverable and indicates that a registry was configured incorrectly. Registrations are expected to cover completely distinct ranges of type strings.

> **Warning:** In a future version of `eth-abi`, this error class will no longer inherit from `ValueError`.

exception eth_abi.exceptions.**NoEntriesFound**
Bases: `ValueError`, *eth_abi.exceptions.PredicateMappingError*

Raised when no registration is found for a type string in a registry's internal mapping.

> **Warning:** In a future version of `eth-abi`, this error class will no longer inherit from `ValueError`.

exception eth_abi.exceptions.**NonEmptyPaddingBytes**
Bases: *eth_abi.exceptions.DecodingError*

Raised when the padding bytes of an ABI value are malformed.

exception eth_abi.exceptions.**ParseError**(*text*, *pos=-1*, *expr=None*)
Bases: `parsimonious.exceptions.ParseError`

Raised when an ABI type string cannot be parsed.

exception eth_abi.exceptions.**PredicateMappingError**

Bases: Exception

Raised when an error occurs in a registry's internal mapping.

exception eth_abi.exceptions.**ValueOutOfBounds**

Bases: *eth_abi.exceptions.IllegalValue*

Raised when trying to encode a python value with the correct type but with a value that appears outside the range of valid values for the output ABI type.

Example:

```
ufixed8x1_encoder(Decimal('25.6'))  # out of bounds
```

2.8.7 eth_abi.registry module

class eth_abi.registry.**ABIRegistry**

copy()

Copies a registry such that new registrations can be made or existing registrations can be unregistered without affecting any instance from which a copy was obtained. This is useful if an existing registry fulfills most of a user's needs but requires one or two modifications. In that case, a copy of that registry can be obtained and the necessary changes made without affecting the original registry.

has_encoder(*type_str: str*) → bool

Returns `True` if an encoder is found for the given type string `type_str`. Otherwise, returns `False`. Raises *MultipleEntriesFound* if multiple encoders are found.

register(*lookup: Union[str, Callable[[str], bool]], encoder: Union[Callable[[Any], bytes], Type[eth_abi.encoding.BaseEncoder]], decoder: Union[Callable[[eth_abi.decoding.ContextFramesBytesIO], Any], Type[eth_abi.decoding.BaseDecoder]], label: str = None*) → None

Registers the given `encoder` and `decoder` under the given `lookup`. A unique string label may be optionally provided that can be used to refer to the registration by name.

Parameters

- **lookup** – A type string or type string matcher function (predicate). When the registry is queried with a type string `query` to determine which encoder or decoder to use, `query` will be checked against every registration in the registry. If a registration was created with a type string for `lookup`, it will be considered a match if `lookup == query`. If a registration was created with a matcher function for `lookup`, it will be considered a match if `lookup(query) is True`. If more than one registration is found to be a match, then an exception is raised.
- **encoder** – An encoder callable or class to use if `lookup` matches a query. If `encoder` is a callable, it must accept a python value and return a `bytes` value. If `encoder` is a class, it must be a valid subclass of *encoding.BaseEncoder* and must also implement the *from_type_str* method on *base.BaseCoder*.
- **decoder** – A decoder callable or class to use if `lookup` matches a query. If `decoder` is a callable, it must accept a stream-like object of bytes and return a python value. If `decoder` is a class, it must be a valid subclass of *decoding.BaseDecoder* and must also implement the *from_type_str* method on *base.BaseCoder*.

- **label** – An optional label that can be used to refer to this registration by name. This label can be used to unregister an entry in the registry via the `unregister` method and its variants.

register_decoder(*lookup: Union[str, Callable[[str], bool]], decoder: Union[Callable[[eth_abi.decoding.ContextFramesBytesIO], Any], Type[eth_abi.decoding.BaseDecoder]], label: str = None*) → None

Registers the given `decoder` under the given `lookup`. A unique string label may be optionally provided that can be used to refer to the registration by name. For more information about arguments, refer to `register`.

register_encoder(*lookup: Union[str, Callable[[str], bool]], encoder: Union[Callable[[Any], bytes], Type[eth_abi.encoding.BaseEncoder]], label: str = None*) → None

Registers the given `encoder` under the given `lookup`. A unique string label may be optionally provided that can be used to refer to the registration by name. For more information about arguments, refer to `register`.

unregister(*label: str*) → None

Unregisters the entries in the encoder and decoder registries which have the label `label`.

unregister_decoder(*lookup_or_label: Union[str, Callable[[str], bool]]*) → None

Unregisters a decoder in the registry with the given lookup or label. If `lookup_or_label` is a string, the decoder with the label `lookup_or_label` will be unregistered. If it is an function, the decoder with the lookup function `lookup_or_label` will be unregistered.

unregister_encoder(*lookup_or_label: Union[str, Callable[[str], bool]]*) → None

Unregisters an encoder in the registry with the given lookup or label. If `lookup_or_label` is a string, the encoder with the label `lookup_or_label` will be unregistered. If it is an function, the encoder with the lookup function `lookup_or_label` will be unregistered.

2.8.8 eth_abi.grammar module

class `eth_abi.grammar.`**ABIType**(*arrlist=None, node=None*)

Base class for results of type string parsing operations.

arrlist

The list of array dimensions for a parsed type. Equal to `None` if type string has no array dimensions.

is_array

Equal to `True` if a type is an array type (i.e. if it has an array dimension list). Otherwise, equal to `False`.

is_dynamic

Equal to `True` if a type has a dynamically sized encoding. Otherwise, equal to `False`.

item_type

If this type is an array type, equal to an appropriate `ABIType` instance for the array's items.

node

The parsimonious `Node` instance associated with this parsed type. Used to generate error messages for invalid types.

to_type_str()

Returns the string representation of an ABI type. This will be equal to the type string from which it was created.

validate()

Validates the properties of an ABI type against the solidity ABI spec:

https://solidity.readthedocs.io/en/develop/abi-spec.html

Raises *ABITypeError* if validation fails.

class eth_abi.grammar.**TupleType**(*components*, *arrlist=None*, *, *node=None*)

Represents the result of parsing a tuple type string e.g. "(int,bool)".

components

A tuple of *ABIType* instances for each of the tuple type's components.

is_dynamic

Equal to `True` if a type has a dynamically sized encoding. Otherwise, equal to `False`.

item_type

If this type is an array type, equal to an appropriate *ABIType* instance for the array's items.

to_type_str()

Returns the string representation of an ABI type. This will be equal to the type string from which it was created.

validate()

Validates the properties of an ABI type against the solidity ABI spec:

https://solidity.readthedocs.io/en/develop/abi-spec.html

Raises *ABITypeError* if validation fails.

class eth_abi.grammar.**BasicType**(*base*, *sub=None*, *arrlist=None*, *, *node=None*)

Represents the result of parsing a basic type string e.g. "uint", "address", "ufixed128x19[][2]".

base

The base of a basic type e.g. "uint" for "uint256" etc.

is_dynamic

Equal to `True` if a type has a dynamically sized encoding. Otherwise, equal to `False`.

item_type

If this type is an array type, equal to an appropriate *ABIType* instance for the array's items.

sub

The sub type of a basic type e.g. `256` for "uint256" or `(128, 18)` for "ufixed128x18" etc. Equal to `None` if type string has no sub type.

to_type_str()

Returns the string representation of an ABI type. This will be equal to the type string from which it was created.

validate()

Validates the properties of an ABI type against the solidity ABI spec:

https://solidity.readthedocs.io/en/develop/abi-spec.html

Raises *ABITypeError* if validation fails.

eth_abi.grammar.**normalize**(*type_str*)

Normalizes a type string into its canonical version e.g. the type string 'int' becomes 'int256', etc.

Parameters **type_str** – The type string to be normalized.

Returns The canonical version of the input type string.

eth_abi.grammar.**parse**(*type_str*)

Parses a type string into an appropriate instance of *ABIType*. If a type string cannot be parsed, throws *ParseError*.

Parameters **type_str** – The type string to be parsed.

Returns An instance of `ABIType` containing information about the parsed type string.

2.8.9 eth_abi.tools module

2.9 Release Notes

2.9.1 eth-abi v2.2.0 (2022-07-20)

Features

- Add support for Python 3.8. Includes updating mypy and flake8 version requirements (#164)

Improved Documentation

- Fix broken badges in README (#164)

Deprecations

- Add `DeprecationWarning` for `encode_abi()`, `encode_single()`, `decode_abi()`, and `decode_single()` and add temporary versions of `abi.encode()` and `abi.decode()` so users can start making these changes early. (#164)

Miscellaneous changes

- #164, #166, #172, #184

2.9.2 eth-abi v2.1.1 (2020-02-27)

Bugfixes

- If subclassing `eth_abi.decoding.ContextFramesBytesIO.seek()`, the new method was not being used by `seek_in_frame()`. Now it will be. (#139)

Internal Changes - for eth_abi contributors

- Merged in project template, for changes in release scripts, docs, release notes, etc. (#140)

2.9.3 v2.1.0

- Added support for "byte" alias for "bytes1" type.
- Added support for custom stream class in `ABIDecoder`. See *Using a Custom Stream Class*.

2.9.4 v2.0.0

- Includes all changes from v2.0.0 beta and alpha versions.

2.9.5 v2.0.0-beta.9

- Added `eth_abi.tools` submodule with extra requirements installable with `pip install eth-abi[tools]`. See *Tools*.

2.9.6 v2.0.0-beta.8

- Added `has_encoder()` and `is_encodable_type()` to facilitate checking for type validity against coder registrations.

2.9.7 v2.0.0-beta.7

Released March 24, 2019

- Fixed an issue that caused custom types containing capital letters to be unparseable.
- Removed PyPy support.
- Added Python 3.7 support.

2.9.8 v2.0.0-beta.6

- Added the grammar module to the public API. See *Grammar*.
- Updated string API for the `ABIType`. Type strings for `ABIType` instances are now obtained via the `to_type_str()` method instead of by invoking the builtin Python `str` function with an instance of `ABIType`.

2.9.9 v2.0.0-beta.5

- Added registry copying functionality to facilitate modification of the default registry. See *Copying an Existing Registry*.

2.9.10 v2.0.0-beta.4

- Update eth-typing requirement to `>=2.0.0,<3.0.0`.

2.9.11 v2.0.0-beta.3

- Added codec API to facilitate use of custom registries. See *Codecs*.

2.9.12 v2.0.0-beta.2

Released October 16, 2018

- Bugfixes
  - Was accidentally allowing eth-typing v2. Now it requires eth-typing v1 only.

2.9.13 v2.0.0-beta.1

- New Features
  - Added support for nested dynamic arrays from the Solidity version 2 ABI
  - Added support for non-standard packed mode encoding
  - Added support for tuple array types e.g. `(int,int)[]`
- Backwards Incompatible Changes
  - The `encode_single()` and `decode_single()` functions no longer accept type tuples to identify ABI types. Only type strings are accepted.
  - The `collapse_type()` function has been removed. People who still wish to use this function should replicate its logic locally and where needed.
  - The `process_type()` function has been removed in favor of the *`parse()`* function. This should make the parsing API more consistent with the new parsimonious parser.

2.9.14 v2.0.0-alpha.1

Released July 19, 2018

- Backwards Incompatible Changes
  - `decode_single()` called with ABI type 'string' will now return a python `str` instead of `bytes`.
  - Support for the legacy `real` and `ureal` types has been removed
- Bugfixes
  - Simple callable encoders work again
- Misc
  - Various documentation updates and type annotations

2.9.15 v1.3.0

Released December 6, 2018

- Bugfixes
  - Resolved an issue that was preventing discovery of type hints.
- Misc
  - Updated eth-typing dependency version to `>=2.0.0,<3.0.0`.

2.9.16 v1.2.2

Released October 18, 2018

- Bugfixes
  - Expand parsimonious dependency from v0.8.0 to v0.8.*

2.9.17 v1.2.1

Released October 16, 2018

- Bugfixes
  - Was accidentally allowing eth-typing v2. Now it requires eth-typing v1 only. (backport from v2)

2.9.18 v1.2.0

Released August 28, 2018

- New Features
  - Backported and added support for nested dynamic arrays from the Solidity version 2 ABI

2.9.19 v1.1.1

Released May 10, 2018

- Bugfixes
  - `is_encodable()` now returns `False` if a `Decimal` has too many digits to be encoded in the given `fixed<M>x<N>` type. (It was previously raising a `ValueError`)
  - Raise an *`EncodingTypeError`* instead of a `TypeError` when trying to encode a `float` into a `fixed<M>x<N>` type.

2.9.20 v1.1.0

Released May 8, 2018

- New Features
  - Added a Registry API (docs in progress) for looking up encoders by ABI type
  - Added support for types: tuple and fixedMxN
  - Added new is_encodable check for whether a value can be encoded with the given ABI type
- Bugfixes
  - Fix RealDecoder bug that allowed values other than 32 bytes
  - Fix bug that accepted `stringN` as a valid ABI type. Strings may not have a fixed length.
  - Stricter value checking when encoding a Decimal (Make sure it's not a NaN)
  - Fix typos in "missing property" exceptions
- Misc
  - Precompile regexes, for performance & clarity
  - Test fixups and switch to CircleCI
  - Readme improvements
  - Performance improvements
  - Drop Python 2 support cruft

2.9.21 v1.0.0

Released Feb 28, 2018

- Confirmed pypy3 compatibility
- Add support for eth-utils v1.0.0-beta2 and v1.0.1 stable
- Testing improvements

2.9.22 v1.0.0-beta.0

Released Feb 5, 2018

- Drop py2 support
- Add support for eth-utils v1-beta1

2.9.23 v0.5.0

- Rename to `eth-abi` for consistency across github/pypi/python-module

2.9.24 v0.4.4

- Better error messages for decoder errors.

2.9.25 v0.4.3

- Bugfix for `process_type` to support byte string type arrguments

2.9.26 v0.4.2

- `process_type` now auto-expands all types which have omittied their sizes.

2.9.27 v0.4.1

- Support for `function` types.

2.9.28 v0.3.1

- Bugfix for small signed integer and real encoding/decoding

2.9.29 v0.3.1

- Bugfix for faulty release.

2.9.30 v0.3.0

- Depart from the original pyethereum encoding/decoding logic.
- Fully rewritten encoder and decoder functionality.

2.9.31 v0.2.2

- Fix a handful of bytes encoding issues.

2.9.32 v0.2.1

- Use pyrlp utility functions for big_endian int operations

2.9.33 v0.2.0

- Bugfixes from upstream pyethereum repository for encoding/decoding
- Python 3 Support

2.9.34 v0.1.0

- Initial release

CHAPTER 3

Indices and Tables

- API Glossary
- modindex

Python Module Index

e

Index

N

P

R

S

T

U

V