Eth-Utils Documentation

Piper Merriam, et al.

May 11, 2020

Contents:

CHAPTER 1

Overview

Join the chat at https://gitter.im/ethereum/eth-utils

Build Status

Common utility functions for codebases which interact with ethereum.

- This library and repository was previously located at https://github.com/pipermerriam/ethereum-utils. It was transferred to the Ethereum foundation github in November 2017 and renamed to `eth-utils`. The PyPi package was also renamed from `ethereum-utils` to 'eth-utils'.

1.1 Installation

```
pip install eth-utils
```

1.2 Development

Clone the repository and then run:

```
pip install -e .[dev] eth-hash[pycryptodome]
```

1.3 Documentation

Building Sphinx docs locally:

```
pip install -e .[doc]
cd docs
make html
```

Docs are written in reStructuredText and built using the Sphinx documentation generator.

1.3.1 Running the tests

You can run the tests with:

```
py.test tests
```

Or you can install `tox` to run the full test suite.

1.3.2 Releasing

Pandoc is required for transforming the markdown README to the proper format to render correctly on pypi.

For Debian-like systems:

```
apt install pandoc
```

Or on OSX:

```
brew install pandoc
```

To release a new version:

```
make release bump=$$VERSION_PART_TO_BUMP$$
```

How to bumpversion

The version format for this repo is `{major}.{minor}.{patch}` for stable, and `{major}.{minor}.{patch}-{stage}.{devnum}` for unstable (`stage` can be alpha or beta).

To issue the next version in line, specify which part to bump, like `make release bump=minor` or `make release bump=devnum`.

If you are in a beta version, `make release bump=stage` will switch to a stable.

To issue an unstable version when the current version is stable, specify the new version explicitly, like `make release bump="--new-version 4.0.0-alpha.1 devnum"`

CHAPTER 2

Utilities

All functions can be imported directly from the `eth_utils` module

Alternatively, you can get the curried version of the functions by importing them through the `curried` module like so:

```
>>> from eth_utils.curried import hexstr_if_str
```

2.1 ABI Utils

2.1.1 `event_abi_to_log_topic(event_abi)` -> bytes

Returns the 32 byte log topic for the given event abi.

```
>>> from eth_utils import event_abi_to_log_topic
>>> event_abi_to_log_topic({'type': 'event', 'anonymous': False, 'name': 'MyEvent',
↪'inputs': []})
b
↪'M\xbf\xb6\x8bC\xdd\xdf\xa1+Q\xeb\xe9\x9a\xb8\xfd\xedb\x0f\x9a\n\xc21B\x87\x9aO\x19*\x1byR\xd2
↪'
```

2.1.2 `event_signature_to_log_topic(event_signature)` -> bytes

Returns the 32 byte log topic for the given event signature.

```
>>> from eth_utils import event_signature_to_log_topic
>>> event_signature_to_log_topic('MyEvent()')
b
↪'M\xbf\xb6\x8bC\xdd\xdf\xa1+Q\xeb\xe9\x9a\xb8\xfd\xedb\x0f\x9a\n\xc21B\x87\x9aO\x19*\x1byR\xd2
↪'
```

2.1.3 function_abi_to_4byte_selector(function_abi) -> bytes

Returns the 4 byte function selector for the given function abi.

```
>>> from eth_utils import function_abi_to_4byte_selector
>>> function_abi_to_4byte_selector({'type': 'function', 'name': 'myFunction', 'inputs
↪': [], 'outputs': []})
b'\xc3x\n:'
```

2.1.4 function_signature_to_4byte_selector(function_signature) -> bytes

Returns the 4 byte function selector for the given function signature.

```
>>> from eth_utils import function_signature_to_4byte_selector
>>> function_signature_to_4byte_selector('myFunction()')
b'\xc3x\n:'
```

2.2 Applicators

Applicators help you apply "formatters" in various ways, most notably:

- apply formatters to values by key
- apply formatters to lists by index
- conditionally applying a formatter
- conditionally applying one of several formatters.

Here we define a "formatter" as any `callable` that may be called with a single positional argument. It returns the "formatted" result. For example `int()` could be used as a formatter.

Defining your own formatter is easy:

```
def i_put_my_thing_down_flip_it_and_reverse_it(lyric):
    return ''.join(reversed(lyric))
```

These tools often work nicely when curried. Import them from the `curried` module to get that capability built in, like `from eth_utils.curried import apply_formatter_if`.

2.2.1 apply_formatter_if(condition, formatter, value) -> new_value

This function will apply the formatter only if `bool(condition()) is True`.

```
>>> from eth_utils.curried import apply_formatter_if, is_string

>>> bool_if_string = apply_formatter_if(is_string, bool)

>>> bool_if_string(1)
1
>>> bool_if_string('1')
True
>>> bool_if_string('')
False
```

2.2.2 apply_one_of_formatters(condition_formatter_pairs, value) -> new_value

This function will iterate through `condition_formatter_pairs`, and apply the first formatter which has a truthy condition. One of the formatters *must* match, or this function will raise a `ValueError`.

```
>>> from eth_utils.curried import apply_one_of_formatters, is_string, is_list_like

>>> multi_formatter = apply_one_of_formatters((
    (is_list_like, tuple),
    (is_string, i_put_my_thing_down_flip_it_and_reverse_it),
)
>>> multi_formatter('my thing')
'gniht ym'
>>> multi_formatter([1, 2])
(1, 2)
>>> multi_formatter(54)
ValueError("The provided value did not satisfy any of the formatter conditions")
```

2.2.3 apply_formatter_at_index(formatter, at_index, <list_like>) -> <new_list_like>

This function will apply the formatter to one element of `list_like`, at position `at_index`, and return a new iterable with that element replaced. The returned value will be the same type as the one passed into the third argument.

```
>>> from eth_utils.curried import apply_formatter_at_index

>>> targetted_formatter = apply_formatter_at_index(bool, 1)

>>> targetted_formatter((1, 2, 3))
(1, True, 3)

>>> targetted_formatter([1, 2, 3])
[1, True, 3]
```

2.2.4 apply_formatter_to_array(formatter, <list_like>) -> <new_list_like>

This function will apply the formatter to each element of `list_like`. It returns the same type as the `list_like` argument

```
>>> from eth_utils.curried import apply_formatter_to_array

>>> map_int = apply_formatter_to_array(int)

>>> map_int((1.2, 3.4, 5.6))
(1, 3, 5)

>>> map_int([1.2, 3.4, 5.6])
[1, 3, 5]
```

2.2.5 apply_formatters_to_sequence(formatters, <list_like>) -> <new_list_like>

This function will apply each formatter at to the list-like value, at the position it was supplied. It returns the same time as the `list_like` argument. For example:

```
>>> from eth_utils.curried import apply_formatters_to_sequence

>>> list_formatter = apply_formatters_to_sequence([bool, int, str])

>>> list_formatter([1.2, 3.4, 5.6])
[True, 3, '5.6']

>>> list_formatter((1.2, 3.4, 5.6))
(True, 3, '5.6')

# Formatters and list-like value must be the same length

>>> list_formatter((1.2, 3.4, 5.6, 7.8))
Traceback (most recent call last):
IndexError: Too few formatters for sequence: 3 formatters for (1.2, 3.4, 5.6, 7.8)

>>> list_formatter((1.2, 3.4))
Traceback (most recent call last):
IndexError: Too many formatters for sequence: 3 formatters for (1.2, 3.4)
```

2.2.6 combine_argument_formatters(*formatters) -> lambda <list_like>: <new_list_like>

DEPRECATED

You can replace all current versions of:

```
>>> from eth_utils import combine_argument_formatters

>>> list_formatter = combine_argument_formatters(bool, int, str)
```

With the newer, preferred:

```
>>> from eth_utils.curried import apply_formatters_to_sequence

>>> list_formatter = apply_formatters_to_sequence((bool, int, str))
```

The old usage works like:

Combine several formatters to be applied to a list-like value, each formatter at the position it was supplied. The new formatter will return the same type as it was supplied. For example:

```
>>> from eth_utils import combine_argument_formatters

>>> list_formatter = combine_argument_formatters(bool, int, str)

>>> list_formatter([1.2, 3.4, 5.6])
[True, 3, '5.6']

>>> list_formatter((1.2, 3.4, 5.6))
```

(continues on next page)

(continued from previous page)

```
(True, 3, '5.6')

# it will pass through items longer than the number of formatters supplied
>>> list_formatter((1.2, 3.4, 5.6, 7.8))
(True, 3, '5.6', 7.8)
```

2.2.7 apply_formatters_to_dict(formatter_dict, <dict_like>) -> dict

This function will apply the formatter to the element with the matching key in `dict_like`, passing through values with keys that have no matching formatter.

```
>>> from eth_utils.curried import apply_formatters_to_dict

>>> dict_formatter = apply_formatters_to_dict({
...     'should_be_int': int,
...     'should_be_bool': bool,
... })

>>> result = dict_formatter({
...     'should_be_int': 1.2,
...     'should_be_bool': 3.4,
...     'pass_through': 5.6,
... })
>>> result == {'should_be_int': 1, 'should_be_bool': True, 'pass_through': 5.6}
True
```

2.2.8 apply_key_map(formatter_dict, <dict_like>) -> dict

This function will rename keys from using the lookups provided in `formatter_dict`. It will pass through any unspecified keys.

```
>>> from eth_utils.curried import apply_key_map

>>> dict_key_map = apply_key_map({
...     'black': 'orange',
...     'Internet': 'Ethereum',
... })

>>> result = dict_key_map({
...     'black': 1.2,
...     'Internet': 3.4,
...     'pass_through': 5.6,
... })
>>> result == {'orange': 1.2, 'Ethereum': 3.4, 'pass_through': 5.6}
True
```

2.3 Address Utils

2.3.1 is_address(value) -> bool

Returns `True` if the `value` is one of the following accepted address formats.

- 20 byte hexadecimal, upper/lower/mixed case, with or without `0x` prefix:
  - `'d3cda913deb6f67967b99d67acdfa1712c293601'`
  - `'0xd3cda913deb6f67967b99d67acdfa1712c293601'`
  - `'0xD3CDA913DEB6F67967B99D67ACDFA1712C293601'`
  - `'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'`
- 20 byte hexadecimal padded to 32 bytes with null bytes, upper/lower/mixed case, with or without `0x` prefix:
  - `'000000000000000000000000d3cda913deb6f67967b99d67acdfa1712c293601'`
  - `'000000000000000000000000d3cda913deb6f67967b99d67acdfa1712c293601'`
  - `'0x000000000000000000000000d3cda913deb6f67967b99d67acdfa1712c293601'`
  - `'0x000000000000000000000000D3CDA913DEB6F67967B99D67ACDFA1712C293601'`
  - `'0x000000000000000000000000d3CdA913deB6f67967B99D67aCDFa1712C293601'`
- 20 text or bytes string:
  - `'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01'`

This function has two special cases when it will return False:

- a 20-byte hex string that has mixed case, with an invalid checksum
- a 32-byte value that is all null bytes

```
>>> from eth_utils import is_address
>>> is_address('d3cda913deb6f67967b99d67acdfa1712c293601')
True
>>> is_address('0xd3cda913deb6f67967b99d67acdfa1712c293601')
True
>>> is_address('0xD3CDA913DEB6F67967B99D67ACDFA1712C293601')
True
>>> is_address('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
True
>>> is_address('000000000000000000000000d3cda913deb6f67967b99d67acdfa1712c293601')
False
>>> is_address('000000000000000000000000d3cda913deb6f67967b99d67acdfa1712c293601')
False
>>> is_address('0x000000000000000000000000d3cda913deb6f67967b99d67acdfa1712c293601')
False
>>> is_address('0x000000000000000000000000D3CDA913DEB6F67967B99D67ACDFA1712C293601')
False
>>> is_address('0x000000000000000000000000d3CdA913deB6f67967B99D67aCDFa1712C293601')
False
>>> is_address(b'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01')
True
>>> is_address(
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa
↪)6\x01')
False
>>> is_address('0x0000000000000000000000000000000000000000000000000000000000000000')
False
>>> is_address(
↪'\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
↪')
False
```

2.3.2 is_hex_address(value) -> bool

Return `True` if the value is a 20 byte hexadecimal encoded string in any of upper/lower/mixed casing, with or without the `0x` prefix. Otherwise return `False`

- `'d3cda913deb6f67967b99d67acdfa1712c293601'`
- `'0xd3cda913deb6f67967b99d67acdfa1712c293601'`
- `'0xD3CDA913DEB6F67967B99D67ACDFA1712C293601'`
- `'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'`

```
>>> from eth_utils import is_hex_address
>>> is_hex_address('d3cda913deb6f67967b99d67acdfa1712c293601')
True
>>> is_hex_address('0xd3cda913deb6f67967b99d67acdfa1712c293601')
True
>>> is_hex_address('0xD3CDA913DEB6F67967B99D67ACDFA1712C293601')
True
>>> is_hex_address('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
True
>>> is_hex_address('000000000000000000000000d3cda913deb6f67967b99d67acdfa1712c293601')
False
>>> is_hex_address('000000000000000000000000d3cda913deb6f67967b99d67acdfa1712c293601')
False
>>> is_hex_address('0x000000000000000000000000d3cda913deb6f67967b99d67acdfa1712c293601
↪')
False
>>> is_hex_address('0x000000000000000000000000D3CDA913DEB6F67967B99D67ACDFA1712C293601
↪')
False
>>> is_hex_address('0x000000000000000000000000d3CdA913deB6f67967B99D67aCDFa1712C293601
↪')
False
>>> is_hex_address('\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01')
False
>>> is_hex_address(
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa
↪)6\x01')
False
>>> is_hex_address('0x0000000000000000000000000000000000000000000000000000000000000000
↪')
False
>>> is_hex_address(
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
↪')
False
```

2.3.3 is_binary_address(value) -> bool

Return `True` if the value is a 20 byte string.

```
>>> from eth_utils import is_binary_address
>>> is_binary_address('d3cda913deb6f67967b99d67acdfa1712c293601')
False
>>> is_binary_address('0xd3cda913deb6f67967b99d67acdfa1712c293601')
```

(continues on next page)

(continued from previous page)

```
False
>>> is_binary_address('0xD3CDA913DEB6F67967B99D67ACDFA1712C293601')
False
>>> is_binary_address('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
False
>>> is_binary_address(
↪'000000000000000000000000d3cda913deb6f67967b99d67acdfa1712c293601')
False
>>> is_binary_address(
↪'000000000000000000000000d3cda913deb6f67967b99d67acdfa1712c293601')
False
>>> is_binary_address(
↪'0x000000000000000000000000d3cda913deb6f67967b99d67acdfa1712c293601')
False
>>> is_binary_address(
↪'0x000000000000000000000000D3CDA913DEB6F67967B99D67ACDFA1712C293601')
False
>>> is_binary_address(
↪'0x000000000000000000000000d3CdA913deB6f67967B99D67aCDFa1712C293601')
False
>>> is_binary_address(b'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01')
True
>>> is_binary_address(
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa
↪)6\x01')
False
>>> is_binary_address(
↪'0x0000000000000000000000000000000000000000000000000000000000000000')
False
>>> is_binary_address(
↪'\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
↪')
False
```

2.3.4 `is_canonical_address(value) -> bool`

Returns `True` if the `value` is an address in its canonical form.

The canonical representation of an address according to `eth_utils` is a 20 byte long string of bytes, eg: `b'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01'`

```
>>> from eth_utils import is_canonical_address
>>> is_canonical_address('0xd3cda913deb6f67967b99d67acdfa1712c293601')
False
>>> is_canonical_address(b'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01
↪')
True
>>> is_canonical_address('\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,
↪)6\x01xd')
False
```

2.3.5 `is_checksum_address(value) -> bool`

Returns `True` if the `value` is a checksummed address as specified by ERC55

```
>>> from eth_utils import is_checksum_address
>>> is_checksum_address('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
True
>>> is_checksum_address('0xd3cda913deb6f67967b99d67acdfa1712c293601')
False
>>> is_checksum_address('0xD3CDA913DEB6F67967B99D67ACDFA1712C293601')
False
>>> is_checksum_address('0x52908400098527886E0F7030069857D2E4169EE7')
True
>>> is_checksum_address('0xde709f21023062209210603147156290 80e2fb77')
True
```

2.3.6 `is_checksum_formatted_address(value) -> bool`

Returns `True` if the `value` is formatted as an ERC55 checksum address.

```
>>> from eth_utils import is_checksum_formatted_address
>>> is_checksum_formatted_address('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
True
>>> is_checksum_formatted_address('0xd3cda913deb6f67967b99d67acdfa1712c293601')
False
>>> is_checksum_formatted_address('0xD3CDA913DEB6F67967B99D67ACDFA1712C293601')
False
>>> is_checksum_formatted_address('0x52908400098527886E0F7030069857D2E4169EE7')
False
>>> is_checksum_formatted_address('0xde709f21023062209210603147156290 80e2fb77')
False
```

2.3.7 `is_normalized_address(value) -> bool`

Returns `True` if the `value` is an address in its normalized form.

The normalized representation of an address is the lowercased 20 byte hexadecimal format.

```
>>> from eth_utils import is_normalized_address
>>> is_normalized_address('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
False
>>> is_normalized_address('0xd3cda913deb6f67967b99d67acdfa1712c293601')
True
>>> is_normalized_address('0xD3CDA913DEB6F67967B99D67ACDFA1712C293601')
False
>>> is_normalized_address('0x52908400098527886E0F7030069857D2E4169EE7')
False
>>> is_normalized_address('0xde709f21023062209210603147156290 80e2fb77')
True
```

2.3.8 `is_same_address(a, b) -> bool`

Returns `True` if both `a` and `b` are valid addresses according to the `is_address` function and that they are both representations of the same address.

```
>>> from eth_utils import is_same_address
>>> is_same_address('0xd3cda913deb6f67967b99d67acdfa1712c293601',
↪'0xD3CDA913DEB6F67967B99D67ACDFA1712C293601')
True
>>> is_same_address('0xd3cda913deb6f67967b99d67acdfa1712c293601',
↪'0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
True
>>> is_same_address('0xd3cda913deb6f67967b99d67acdfa1712c293601', b
↪'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01')
True
```

2.3.9 `to_canonical_address(value)` -> Address

Given any valid representation of an address return its canonical form.

```
>>> from eth_utils import to_canonical_address
>>> to_canonical_address('0xd3cda913deb6f67967b99d67acdfa1712c293601')
b'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01'
>>> to_canonical_address('0xD3CDA913DEB6F67967B99D67ACDFA1712C293601')
b'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01'
>>> to_canonical_address('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
b'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01'
>>> to_canonical_address(b'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01
↪')
b'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01'
```

2.3.10 `to_checksum_address(value)` -> ChecksumAddress

Given any valid representation of an address return the checksummed representation.

```
>>> from eth_utils import to_checksum_address
>>> to_checksum_address('0xd3cda913deb6f67967b99d67acdfa1712c293601')
'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'
>>> to_checksum_address('0xD3CDA913DEB6F67967B99D67ACDFA1712C293601')
'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'
>>> to_checksum_address('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'
>>> to_checksum_address(b'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,)6\x01
↪')
'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'
```

2.3.11 `to_normalized_address(value)` -> HexAddress

Given any valid representation of an address return the normalized representation.

```
>>> from eth_utils import to_normalized_address
>>> to_normalized_address(b'\xd3\xcd\xa9\x13\xde\xb6\xf6yg\xb9\x9dg\xac\xdf\xa1q,
↪)6\x01')  # raw bytes
'0xd3cda913deb6f67967b99d67acdfa1712c293601'
>>> to_normalized_address('c6d9d2cd449a754c494264e1809c50e34d64562b')  # hex encoded
'0xc6d9d2cd449a754c494264e1809c50e34d64562b'
>>> to_normalized_address('0xc6d9d2cd449a754c494264e1809c50e34d64562b')  # hex encoded
```

(continues on next page)

(continued from previous page)

```
'0xc6d9d2cd449a754c494264e1809c50e34d64562b'
>>> to_normalized_address('0XC6D9D2CD449A754C494264E1809C50E34D64562B')  # cap-cased
'0xc6d9d2cd449a754c494264e1809c50e34d64562b'
```

2.4 Conversion Utils

These methods convert values using standard practices in the Ethereum ecosystem. For example, strings are encoded to binary using UTF-8.

Because there is no reliable way to distinguish between text and a hex-encoded bytestring, you must explicitly specify which of the two is being supplied when passing in a `str`.

Only supply one of the arguments:

2.4.1 `to_bytes(<bytes/int/bool>, text=<str>, hexstr=<str>)` -> bytes

Takes a variety of inputs and returns its bytes equivalent. Text gets encoded as UTF-8.

```
>>> from eth_utils import to_bytes
>>> to_bytes(0)
b'\x00'
>>> to_bytes(0x000F)
b'\x0f'
>>> to_bytes(b'')
b''
>>> to_bytes(b'\x00\x0F')
b'\x00\x0f'
>>> to_bytes(False)
b'\x00'
>>> to_bytes(True)
b'\x01'
>>> to_bytes(hexstr='0x000F')
b'\x00\x0f'
>>> to_bytes(hexstr='000F')
b'\x00\x0f'
>>> to_bytes(text='')
b''
>>> to_bytes(text='cowmö')
b'cowm\xc3\xb6'
```

2.4.2 `to_hex(<bytes/int/bool>, text=<str>, hexstr=<str>)` -> HexStr

Takes a variety of inputs and returns it in its hexadecimal representation. It follows the rules for converting to hex in the JSON-RPC spec. Roughly, it leaves leading 0s on bytes input, and trims leading zeros on int input.

```
>>> from eth_utils import to_hex
>>> to_hex(0)
'0x0'
>>> to_hex(1)
'0x1'
>>> to_hex(0x0)
```

(continues on next page)

(continued from previous page)

```
'0x0'
>>> to_hex(0x000F)
'0xf'
>>> to_hex(b'')
'0x'
>>> to_hex(b'\x00\x0F')
'0x000f'
>>> to_hex(False)
'0x0'
>>> to_hex(True)
'0x1'
>>> to_hex(hexstr='0x000F')
'0x000f'
>>> to_hex(hexstr='000F')
'0x000f'
>>> to_hex(text='')
'0x'
>>> to_hex(text='cowmö')
'0x636f776dc3b6'
```

2.4.3 `to_int(<bytes/int/bool>, text=<str>, hexstr=<str>)` -> int

Takes a variety of inputs and returns its integer equivalent.

```
>>> from eth_utils import to_int
>>> to_int(0)
0
>>> to_int(0x000F)
15
>>> to_int(b'\x00\x0F')
15
>>> to_int(False)
0
>>> to_int(True)
1
>>> to_int(hexstr='0x000F')
15
>>> to_int(hexstr='000F')
15
```

2.4.4 `to_text(<bytes/int/bool>, text=<str>, hexstr=<str>)` -> str

Takes a variety of inputs and returns its string equivalent. Text gets decoded as UTF-8.

```
>>> from eth_utils import to_text
>>> to_text(0x636f776dc3b6)
'cowmö'
>>> to_text(b'cowm\xc3\xb6')
'cowmö'
>>> to_text(hexstr='0x636f776dc3b6')
'cowmö'
>>> to_text(hexstr='636f776dc3b6')
'cowmö'
```

(continues on next page)

(continued from previous page)

```
>>> to_text(text='cowmö')
'cowmö'
```

2.5 Crypto Utils

Because there is no reliable way to distinguish between text and a hex-encoded bytestring, you must explicitly specify which of the two is being supplied when passing in a `str`.

Only supply one of the arguments:

2.5.1 keccak(<bytes/int/bool>, text=<str>, hexstr=<str>) -> bytes

```
>>> from eth_utils import keccak
>>> keccak(text='')
b"\xc5\xd2F\x01\x86\xf7#<\x92~}\xb2\xdc\xc7\x03\xc0\xe5\x00\xb6S\xca\x82';
 ↪{\xfa\xd8\x04]\x85\xa4p"

# A series of equivalent hash inputs:

>>> keccak(text='')
b'\x85\xe8\x07"\xeb\x93\r\xe9;\xcc\xa8{\xa5\xdf\xda\x89\n\xa12\x95\xae\xad.
 ↪\xec\xc9\x0b\xb2\xd9z\x14\x93\x16'

>>> keccak(0xe298a2)
b'\x85\xe8\x07"\xeb\x93\r\xe9;\xcc\xa8{\xa5\xdf\xda\x89\n\xa12\x95\xae\xad.
 ↪\xec\xc9\x0b\xb2\xd9z\x14\x93\x16'

>>> keccak(b'\xe2\x98\xa2')
b'\x85\xe8\x07"\xeb\x93\r\xe9;\xcc\xa8{\xa5\xdf\xda\x89\n\xa12\x95\xae\xad.
 ↪\xec\xc9\x0b\xb2\xd9z\x14\x93\x16'

>>> keccak(hexstr='0xe298a2')
b'\x85\xe8\x07"\xeb\x93\r\xe9;\xcc\xa8{\xa5\xdf\xda\x89\n\xa12\x95\xae\xad.
 ↪\xec\xc9\x0b\xb2\xd9z\x14\x93\x16'
```

Please Note - When using Python's native hex literals, python converts the hex to an int, so leading 0 bytes are truncated. But all other formats maintain zeros on the left. Hex literals are only padded until a whole number of bytes are provided to keccak. For example:

```
>>> keccak(0xe298a2)
b'\x85\xe8\x07"\xeb\x93\r\xe9;\xcc\xa8{\xa5\xdf\xda\x89\n\xa12\x95\xae\xad.
 ↪\xec\xc9\x0b\xb2\xd9z\x14\x93\x16'

>>> keccak(0x0e298a2)
b'\x85\xe8\x07"\xeb\x93\r\xe9;\xcc\xa8{\xa5\xdf\xda\x89\n\xa12\x95\xae\xad.
 ↪\xec\xc9\x0b\xb2\xd9z\x14\x93\x16'

>>> keccak(0x00e298a2)
b'\x85\xe8\x07"\xeb\x93\r\xe9;\xcc\xa8{\xa5\xdf\xda\x89\n\xa12\x95\xae\xad.
 ↪\xec\xc9\x0b\xb2\xd9z\x14\x93\x16'

>>> keccak(0x000e298a2)
```

(continues on next page)

(continued from previous page)

```
b'\x85\xe8\x07"\xeb\x93\r\xe9;\xcc\xa8{\xa5\xdf\xda\x89\n\xa12\x95\xae\xad.
↪\xec\xc9\x0b\xb2\xd9z\x14\x93\x16'

>>> keccak(hexstr='0x0e298a2')
b'i\x0f$\xbd\xbe\xf7c\xbb\xb9M\xd9\x12H"\x9f\x1f\x87\\E\xa36\xc2\xea,\x8f.
↪\r\xf5\x95\xdc\x19\x9b'

>>> keccak(hexstr='0x00e298a2')
b'i\x0f$\xbd\xbe\xf7c\xbb\xb9M\xd9\x12H"\x9f\x1f\x87\\E\xa36\xc2\xea,\x8f.
↪\r\xf5\x95\xdc\x19\x9b'

>>> keccak(hexstr='0x000e298a2')
b'!$Ezy\xdeU
↪<\xec\x1f\xd1\x10\x05\xff\x11\xfc=J\xcf\xd5H\x0f\xb3c\xcc\xb5\xae\xb1\x1eA\x8b\xd3'
```

2.6 Currency Utils

2.6.1 denoms

Object with property access to all of the various denominations for ether. Available denominations are:

denomination	amount in wei
wei	1
kwei	1000
babbage	1000
femtoether	1000
mwei	1000000
lovelace	1000000
picoether	1000000
gwei	1000000000
shannon	1000000000
nanoether	1000000000
nano	1000000000
szabo	1000000000000
microether	1000000000000
micro	1000000000000
finney	1000000000000000
milliether	1000000000000000
milli	1000000000000000
ether	1000000000000000000
kether	1000000000000000000000
grand	1000000000000000000000
mether	1000000000000000000000000
gether	1000000000000000000000000000
tether	1000000000000000000000000000000

```
>>> from eth_utils import denoms
>>> denoms.wei
1
```

(continues on next page)

(continued from previous page)

```
>>> denoms.finney
1000000000000000
>>> denoms.ether
1000000000000000000
```

2.6.2 `to_wei(value, denomination)` -> integer

Converts `value` in the given `denomination` to its equivalent in the *wei* denomination.

```
>>> from eth_utils import to_wei
>>> to_wei(1, 'ether')
1000000000000000000
```

2.6.3 `from_wei(value, denomination)` -> decimal.Decimal

Converts the `value` in the *wei* denomination to its equivalent in the given `denomination`. Return value is a `decimal.Decimal` with the appropriate precision to be a lossless conversion.

```
>>> from eth_utils import from_wei
>>> from_wei(1000000000000000000, 'ether')
Decimal('1')
>>> from_wei(123456789, 'ether')
Decimal('1.23456789E-10')
```

2.7 Debug Utils

2.7.1 Generate environment info

At the shell:

```
$ python -m eth_utils

Python version:
3.5.3 (default, Nov 23 2017, 11:34:05)
[GCC 6.3.0 20170406]

Operating System: Linux-4.10.0-42-generic-x86_64-with-Ubuntu-17.04-zesty

pip freeze result:
bumpversion==0.5.3
cytoolz==0.9.0
flake8==3.4.1
ipython==6.2.1
pytest==3.3.2
virtualenv==15.1.0
... etc
```

2.8 Decorators

2.8.1 `@combomethod`

Decorates methods in a class that can be called as both an instance method or a `@classmethod`.

Use the decorator like so:

```
>>> from eth_utils import combomethod

>>> class Storage:
...     val = 1
...
...     @combomethod
...     def get(combo):
...         if isinstance(combo, type):
...             print("classmethod call")
...         elif isinstance(combo, Storage):
...             print("instance method call")
...         else:
...             raise TypeError("Unreachable, unless you really monkey around")
...         return combo.val
...
```

As usual, instances create their own copy on assignment.

```
>>> store = Storage()
>>> store.val = 2

>>> store.get()
instance method call
2

>>> Storage.get()
classmethod call
1
```

2.9 Encoding Utils

2.9.1 `big_endian_to_int(value)` -> integer

Returns `value` converted to an integer (from a big endian representation).

```
>>> from eth_utils import big_endian_to_int
>>> big_endian_to_int(b'\x00')
0
>>> big_endian_to_int(b'\x01')
1
>>> big_endian_to_int(b'\x01\x00')
256
```

2.9.2 `int_to_big_endian(value)` -> bytes

Returns `value` converted to the big endian representation.

```
>>> from eth_utils import int_to_big_endian
>>> int_to_big_endian(0)
b'\x00'
>>> int_to_big_endian(1)
b'\x01'
>>> int_to_big_endian(256)
b'\x01\x00'
```

2.10 Functional Utils

2.10.1 `compose(*callables)` -> callable

DEPRECATED in 0.3.0.

Returns a single function which is the composition of the given callables.

```
>>> def f(v):
...     return v * 3
...
>>> def g(v):
...     return v + 2
...
>>> def h(v):
...     return v % 5
...
>>> compose(f, g, h)(1)
0
>>> h(g(f(1)))
0
>>> compose(f, g, h)(2)
3
>>> h(g(f(1)))
3
>>> compose(f, g, h)(3)
1
>>> h(g(f(1)))
1
>>> compose(f, g, h)(4)
4
>>> h(g(f(1)))
4
```

2.10.2 `flatten_return(callable)` -> callable() -> tuple

Decorator which performs a non-recursive flattening of the return value from the given `callable`.

```
>>> flatten_return(lambda: [[1, 2, 3], [4, 5], [6]])
(1, 2, 3, 4, 5, 6)
```

2.10.3 `sort_return(callable) => callable() -> tuple`

Decorator which sorts the return value from the given `callable`.

```
>>> flatten_return(lambda: [[1, 2, 3], [4, 5], [6]])
(1, 2, 3, 4, 5, 6)
```

2.10.4 `reversed_return(callable) => callable() -> tuple`

Decorator which reverses the return value from the given `callable`.

```
>>> reversed_return(lambda: [1, 5, 2, 4, 3])
(3, 4, 2, 5, 1)
```

2.10.5 `to_dict(callable) => callable() -> dict`

Decorator which casts the return value from the given `callable` to a dictionary.

```
>>> from eth_utils import to_dict
>>> @to_dict
... def build_thing():
...     yield 'a', 1
...     yield 'b', 2
...     yield 'c', 3
...
>>> build_thing() == {'a': 1, 'b': 2, 'c': 3}
True
```

2.10.6 `to_list(callable) => callable() -> list`

Decorator which casts the return value from the given `callable` to a list.

```
>>> from eth_utils import to_list
>>> @to_list
... def build_thing():
...     yield 'a'
...     yield 'b'
...     yield 'c'
...
>>> build_thing()
['a', 'b', 'c']
```

2.10.7 `to_ordered_dict(callable) => callable() -> collections.OrderedDict`

Decorator which casts the return value from the given `callable` to an ordered dictionary of type `collections.OrderedDict`.

```
>>> from eth_utils import to_ordered_dict
>>> @to_ordered_dict
... def build_thing():
...     yield 'd', 4
```

(continues on next page)

(continued from previous page)

```
...     yield 'a', 1
...     yield 'b', 2
...     yield 'c', 3
...
>>> build_thing()
OrderedDict([('d', 4), ('a', 1), ('b', 2), ('c', 3)])
```

2.10.8 `to_tuple(callable)` => callable() -> tuple

Decorator which casts the return value from the given `callable` to a tuple.

```
>>> from eth_utils import to_tuple
>>> @to_tuple
... def build_thing():
...     yield 'a'
...     yield 'b'
...     yield 'c'
...
>>> build_thing()
('a', 'b', 'c')
```

2.10.9 `to_set(callable)` => callable() -> set

Decorator which casts the return value from the given `callable` to a set.

```
>>> from eth_utils import to_set
>>> @to_set
... def build_thing():
...     yield 'a'
...     yield 'b'
...     yield 'a'  # duplicate
...     yield 'c'
...
>>> build_thing() == {'c', 'b', 'a'}
True
```

2.10.10 `apply_to_return_value(callable)` => decorator_fn

This function takes a single callable and returns a decorator. The returned decorator, when applied to a function, will incercept the function's return value, pass it to the callable, and return the value returned by the callable.

```
>>> from eth_utils import apply_to_return_value
>>> double = apply_to_return_value(lambda v: v * 2)
>>> @double
... def f(v):
...     return v
...
>>> f(2)
4
>>> f(3)
6
```

2.11 Hexidecimal Utils

2.11.1 add_0x_prefix(value: HexStr) -> HexStr

Returns `value` with a `0x` prefix. If the value is already prefixed it is returned as-is. Value must be a HexStr.

```
>>> from eth_utils import add_0x_prefix
>>> from eth_typing import HexStr
>>> add_0x_prefix(HexStr('12345'))
'0x12345'
>>> add_0x_prefix(HexStr('0x12345'))
'0x12345'
```

2.11.2 decode_hex(value) -> bytes

Returns `value` decoded into a byte string. Accepts any string with or without the `0x` prefix.

```
>>> from eth_utils import decode_hex
>>> decode_hex('0x123456')
b'\x124V'
>>> decode_hex('123456')
b'\x124V'
```

2.11.3 encode_hex(value) -> string

Returns `value` encoded into a hexadecimal representation with a `0x` prefix

```
>>> from eth_utils import encode_hex
>>> encode_hex(b'\x01\x02\x03')
'0x010203'
```

2.11.4 is_0x_prefixed(value) -> bool

Returns `True` if `value` has a `0x` prefix. Value must be a string literal.

```
>>> from eth_utils import is_0x_prefixed
>>> is_0x_prefixed('12345')
False
>>> is_0x_prefixed('0x12345')
True
```

2.11.5 is_hex(value) -> bool

Returns `True` if `value` is a hexadecimal encoded string of text type.

```
>>> from eth_utils import is_hex
>>> is_hex('')
False
>>> is_hex('0x')
True
```

(continues on next page)

(continued from previous page)

```
>>> is_hex('0X')
True
>>> is_hex('1234567890abcdef')
True
>>> is_hex('0x1234567890abcdef')
True
>>> is_hex('0x1234567890ABCDEF')
True
>>> is_hex('0x1234567890AbCdEf')
True
>>> is_hex('12345')  # odd length is ok
True
>>> is_hex('0x12345')  # odd length is ok
True
>>> is_hex('123456__abcdef')  # non hex characters
False

# invalid, will raise TypeError:
>>> is_hex(b'')
Traceback (most recent call last):
TypeError: is_hex requires text typed arguments.
>>> is_hex(b'0x')
Traceback (most recent call last):
TypeError: is_hex requires text typed arguments.
>>> is_hex(b'0X')
Traceback (most recent call last):
TypeError: is_hex requires text typed arguments.
```

2.11.6 `is_hexstr(value) -> bool`

Returns `True` if `value` is a hexadecimal encoded string of text type.

Note: This function differs from `is_hex(value: Any)` in that it will return False on all non-text type arguments, while `is_hex` will raise a `TypeError` for all non-text type arguments.

```
>>> from eth_utils import is_hexstr
>>> is_hexstr('')
False
>>> is_hexstr('0x')
True
>>> is_hexstr('0X')
True
>>> is_hexstr('1234567890abcdef')
True
>>> is_hexstr('0x1234567890abcdef')
True
>>> is_hexstr('0x1234567890ABCDEF')
True
>>> is_hexstr('0x1234567890AbCdEf')
True
>>> is_hexstr('12345')  # odd length is ok
True
>>> is_hexstr('0x12345')  # odd length is ok
```

(continues on next page)

(continued from previous page)

```
True
>>> is_hexstr('123456__abcdef')  # non hex characters
False
>>> is_hexstr(b'') # any non-string returns False
False
>>> is_hexstr(b'0x') # any non-string returns False
False
```

2.11.7 remove_0x_prefix(value: HexStr) -> HexStr

Returns `value` with the `0x` prefix stripped. If the value does not have a `0x` prefix it is returned as-is. Value must be a HexStr.

```
>>> from eth_utils import remove_0x_prefix
>>> from eth_typing import HexStr
>>> remove_0x_prefix(HexStr('12345'))
'12345'
>>> remove_0x_prefix(HexStr('0x12345'))
'12345'
```

2.12 Humanize Utils

2.12.1 humanize_seconds(seconds) -> string

Returns the provide number of seconds as a shorthand string.

```
>>> from eth_utils import humanize_seconds
>>> humanize_seconds(0)
'0s'
>>> humanize_seconds(1)
'1s'
>>> humanize_seconds(60)
'1m'
>>> humanize_seconds(61)
'1m1s'
```

2.12.2 humanize_bytes(bytes) -> string

Returns the provided byte string in a human readable format.

If the value is 5 bytes or less it is returned in full in its hexidecimal representation (without a `0x` prefix)

If the value is longer that 5 bytes it is returned in its hexidecimal representation (without a `0x` prefix) with the middle segment replaced by an ellipsis, only showing the first and last four hexidecimal nibbles.

```
>>> from eth_utils import humanize_bytes
>>> humanize_bytes(bytes(range(3)))
 '000102'
>>> humanize_bytes(bytes(range(5)))
 '0001020304'
```

(continues on next page)

(continued from previous page)

```
>>> humanize_bytes(bytes(range(32)))
 '0001..1e1f'
```

2.12.3 humanize_hash(bytes) -> string

A loose wrapper around `humanize_bytes` that is typed specifically for the `eth_typing.Hash32` type.

```
>>> from eth_utils import humanize_hash
>>> humanize_hash(bytes(range(32)))
 '0001..1e1f'
```

2.12.4 humanize_integer_sequence(values) -> string

Returns a concise representation of the provided sequence of integer values.

```
>>> from eth_utils import humanize_integer_sequence
>>> humanize_integer_sequence((1, 2, 3, 4))
'1-4'
>>> humanize_integer_sequence((1, 2, 3, 4, 6, 8, 9, 10))
'1-4|6|8-10'
```

2.12.5 humanize_ipfs_uri(string) -> string

Returns the provided IPFS uri, with the middle segment of the hash replaced by an ellipsis, only showing the first and last four characters of the hash.

```
>>> from eth_utils import humanize_ipfs_uri
>>> humanize_ipfs_uri('ipfs://QmTKB75Y73zhNbD3Y73xeXGjYrZHmaXXNxoZqGCagu7r8u')
 'ipfs://QmTK..7r8u'
```

2.13 Logging Utils

2.13.1 get_logger(string, [, logger_class]) -> logger

This API is similar to the standard library `logging.getLogger` however, the logger it returns will be an instance of the provided `logger_class`. If `logger_class` is not provided this returns an instance of whatever the current default logger class is set on the `logging`.

```
>>> import logging
>>> from eth_utils import get_logger
>>> logger = get_logger('my_application')
>>> assert logger.name == 'my_application'
>>> assert isinstance(logger, logging.getLoggerClass())
```

2.13.2 get_extended_debug_logger(string) -> ExtendedDebugLogger

Like `get_logger` except that it always returns an instance of `ExtendedDebugLogger`

```
>>> from eth_utils import get_extended_debug_logger, ExtendedDebugLogger
>>> logger = get_extended_debug_logger('my_application')
>>> assert logger.name == 'my_application'
>>> assert isinstance(logger, ExtendedDebugLogger), type(logger)
```

2.13.3 class HasLogger

Classes which inherit from this class will have an instance of a logger available on the attribute `logger`

```
>>> from eth_utils import HasLogger
>>> class MyClass(HasLogger):
...     pass
...
>>> MyClass.logger.debug("This works")
>>> instance = MyClass()
>>> instance.logger.debug("This also works")
```

The `name` of the logger instance is derived from the `__qualname__` for the class.

Warning: This class will not behave nicely with the standard library `typing.Generic`. If you need to create a `Generic` class then you'll need to assign your logging instances manually.

2.13.4 class ExtendedDebugLogger

A subclass of `logging.Logger` which exposes a `debug2` function which can be used to log a message at the `DEBUG2` log level.

Note: This class works fine on its own but will produce cleaner logs if you make sure to call `eth_utils.setup_DEBUG2_logging` at least once before issuing any `debug2` level logs.

2.13.5 class HasExtendedDebugLogger

Same as the `HasLogger` class except the logger it exposes is an instance of `ExtendedDebugLogger`

2.13.6 setup_DEBUG2_logging() -> None

Installs the `DEBUG2` level to the standard library `logging` module which uses the numeric level of `8`. This includes adding it to the known levels as well as providing a `logging.DEBUG2` convenience property on the logging module.

This function is purely for convenience. You can use `ExtendedDebugLogger` without this, though your logs will be printed with the label `'Level 8'`.

```
>>> from eth_utils import setup_DEBUG2_logging
>>> import logging
>>> logging.getLevelName(8)
'Level 8'
>>> setup_DEBUG2_logging()
>>> logging.getLevelName(8)
```

(continues on next page)

(continued from previous page)

```
'DEBUG2'
>>> logging.DEBUG2
8
```

Note: This function is idempotent

2.13.7 `class HasLoggerMeta`

This is the metaclass which is responsible for adding the logger instance to the class. It exposes two additional APIs.

- `HasLoggerMeta.replace_logger_class(cls: logging.Logger)`

  Returns a new metaclass which will use the provided logger class.

- `HasLoggerMeta.meta_compat(other: type)`

  Returns a new metaclass that derives from both metaclasses. This is useful when working in conjunction with `abc.ABC` or `typing.Generic`.

2.14 Numeric Utils

2.14.1 `clamp(lower_bound, upper_bound, value)` -> result

Returns `value` clamped within the inclusive range defined by `[lower_bound, upper_bound]`. The value can be any number type that supports < and > comparisons against the provided bounds.

```
>>> from eth_utils import clamp
>>> clamp(5, 7, 4)
5
>>> clamp(5, 7, 5)
5
>>> clamp(5, 7, 6)
6
>>> clamp(5, 7, 7)
7
>>> clamp(5, 7, 8)
7
```

2.15 Type Utils

2.15.1 `is_boolean(value)` -> bool

Returns `True` if `value` is of type `bool`

```
>>> from eth_utils import is_boolean
>>> is_boolean(True)
True
>>> is_boolean(False)
True
```

(continues on next page)

(continued from previous page)

```
>>> is_boolean(1)
False
```

2.15.2 is_bytes(value) -> bool

Returns `True` if `value` is a byte string or a byte array.

```
>>> from eth_utils import is_bytes
>>> is_bytes('abcd')
False
>>> is_bytes(b'abcd')
True
>>> is_bytes(bytearray((1, 2, 3)))
True
```

2.15.3 is_dict(value) -> bool

Returns `True` if `value` is a mapping type.

```
>>> from eth_utils import is_dict
>>> is_dict({'a': 1})
True
>>> is_dict([1, 2, 3])
False
```

2.15.4 is_integer(value) -> bool

Returns `True` if `value` is an integer

```
>>> from eth_utils import is_integer
>>> is_integer(0)
True
>>> is_integer(1)
True
>>> is_integer('1')
False
>>> is_integer(1.1)
False
```

2.15.5 is_list_like(value) -> bool

Returns `True` if `value` is a non-string sequence such as a sequence (such as a list or tuple).

```
>>> from eth_utils import is_list_like
>>> is_list_like('abcd')
False
>>> is_list_like([])
True
>>> is_list_like(tuple())
True
```

2.15.6 is_list(value) -> bool

Returns `True` if `value` is a non-string sequence such as a list.

```
>>> from eth_utils import is_list
>>> is_list('abcd')
False
>>> is_list([])
True
>>> is_list(tuple())
False
```

2.15.7 is_tuple(value) -> bool

Returns `True` if `value` is a non-string sequence such as a tuple.

```
>>> from eth_utils import is_tuple
>>> is_tuple('abcd')
False
>>> is_tuple([])
False
>>> is_tuple(tuple())
True
```

2.15.8 is_null(value) -> bool

Returns `True` if `value` is `None`

```
>>> from eth_utils import is_null
>>> is_null(None)
True
>>> is_null(False)
False
```

2.15.9 is_number(value) -> bool

Returns `True` if `value` is numeric

```
>>> from eth_utils import is_number
>>> is_number(1)
True
>>> is_number(1.1)
True
>>> is_number('1')
False
>>> from decimal import Decimal
>>> is_number(Decimal('1'))
True
```

2.15.10 is_string(value) -> bool

Returns `True` if `value` is of any string type.

```
>>> from eth_utils import is_string
>>> is_string('abcd')
True
>>> is_string(b'abcd')
True
>>> is_string(bytearray((1, 2, 3)))
True
```

2.15.11 `is_text(value) -> bool`

Returns `True` if `value` is a text string.

```
>>> from eth_utils import is_text
>>> is_text(u'abcd')
True
>>> is_text(b'abcd')
False
>>> is_text(bytearray((1, 2, 3)))
False
```

CHAPTER 3

Eth_Utils 1.9.0 (2020-05-11)

3.1 Features

- Add *is_hexstr* as preferred method of checking if a given value is a hex string. (#137)
- Improve performance of *is_hex* and *is_hexstr* by up to 40x (#185)
- Add `humanize_integer_sequence` utility. (#188)
- Add `humanize_bytes` utility. (#189)

3.2 Bugfixes

- Silence a deprecation error by importing from collections.abc, instead of collections. (#186)

CHAPTER 4

eth-utils 1.8.4 (2019-12-05)

4.1 Bugfixes

- Add missing asterisk to MANIFEST.in (#182)

CHAPTER 5

eth-utils 1.8.3 (2019-12-04)

5.1 Misc

- #181

CHAPTER 6

eth-utils 1.8.2 (2019-12-04)

6.1 Misc

- #177, #180

CHAPTER 7

eth-utils 1.8.1 (2019-11-20)

No significant changes.

CHAPTER 8

eth-utils 1.8.0-0.1 (2019-11-20)

8.1 Misc

- #175

CHAPTER 9

eth-utils 1.8.0 (2019-11-04)

9.1 Features

- Add support for python3.8 (#174)

CHAPTER 10

eth-utils 1.7.0 (2019-09-05)

10.1 Features

- Expose `DEBUG2` log level as top level module import (#117)
- Add `get_logger` and `get_extended_debug_logger` utils (#170)

10.2 Improved Documentation

- Setup towncrier to improve the quality of the release notes (#172)

10.2.1 Release Notes

CHAPTER 11

v1.6.4

Released: August 5, 2019

- Feature
  - Caching for `ExtendedDebugLogger.show_debug2` property - #167

CHAPTER 12

v1.6.3

Released: August 5, 2019

- Feature
  - Add support for Python3.7 - #165
- Bugfix
  - Fix `HasLogger` compatibility with other metaclasses. - #165

CHAPTER 13

v1.6.2

Released: July 24, 2019

- Feature
  - Add support for Python3.7 - #165
  - Add `humanize_ipfs_uri`. - #162
- Bugfix
  - Fix typing of `clamp` numeric utility. - #164

CHAPTER 14

v1.6.1

Released: June 11, 2019

- Maintenance
  - Use eth-typing types instead of eth-utils types, when available - #163

CHAPTER 15

v1.6.0

Released: May 16, 2019

- Feature
  - Add logging utilities `HasLogger`, `ExtendedDebugLogger`, `HasExtendedDebugLogger` and `setup_DEBUG2_logging` - #158

CHAPTER 16

v1.5.2

Released: April 30, 2019

- Bugfix
  - Fix *eth_utils.currency.denom* to be a real class with proper type declarations. - #154
  - Fix *eth_utils.functional.replace_exceptions* type declarations. - #155
- Feature
  - Add new *eth_utils.clamp* - #150

CHAPTER 17

v1.5.1

Released: April 17, 2019

- Bugfix
  - Fix type declarations for *eth_utils.functional.to_dict* and *eth_utils.funcional.to_ordered_dict* - #151

CHAPTER 18

v1.5.0

Released: April 16, 2019

- Features
  - Add new *eth_utils.humanize.humanize_seconds* and *eth_utils.humanize.humanize_hash*. - #149
  - Enable PEP561 type hints

CHAPTER 19

v1.4.1

Released: Dec 18, 2018

- Bugfixes
  - Fixed *eth_utils.abi.collapse_if_tuple* not handling fixed-size tuple arrays.

CHAPTER 20

v1.4.0

Released: Dec 6, 2018

- Features
  - Support tuples in *eth_utils.abi.function_abi_to_4byte_selector* and a new *eth_utils.abi.collapse_if_tuple* - #141

CHAPTER 21

v1.3.0

- Misc
  - Fix linting issues

CHAPTER 22

v1.3.0-beta.0

- Misc
  - Use eth-typing v2.0.0, which may be a breaking change for downstream packages

CHAPTER 23

v1.2.2

- Bugfixes
  - Prevent from installing with Python 3.5.2 which has a fatal bug when `...` is used in a type. - #125
- Misc
  - Start using `black` for style checking. - #129

CHAPTER 24

1.2.1

- Move docs to RTD/Sphinx, with doctest
- Update eth-typing dependency to 1.3.0

CHAPTER 25

1.2.0

- Import more resources from implementation-specific “toolz” library in “toolz” wrapper module

CHAPTER 26

1.1.2

- Update eth-typing dependency

CHAPTER 27

1.1.1

- Add *ValidationError* exception

CHAPTER 28

1.1.0

- Add *abi* and *address* type hints
- Add typehints to more modules
- Add *replace_exceptions* decorator to decorators.py
- Add type hints to *applicators* module
- Add type hints to *conversions* module
- Add *import_string* util from django
- Add conditional cytoolz or toolz install based on python implementation

CHAPTER 29

1.0.3

- Reject str as a primitive in *to_hex()*
- Faster *int_to_big_endian* implementation

CHAPTER 30

1.0.2

- Update apply key map to catch conflicting keys
- Add validation of 19 byte address
- Support bytearrays in conversion functions
- Apply formatters to sequence

CHAPTER 31

1.0.1

- Add autouse fixture to print warnings
- Change *hexidecimal* -> *hexadecimal*
- Strictly accept text types for decode_hex
- Remove remaining force_* utils

CHAPTER 32

0.8.1

- Convert formatting from force
- Backport pr45 into v0
- Write validate conversion arguments decorator
- Update *hex* and *int* conversions to work with new decorator
- Deprecate force bytes/text & formatting utils

CHAPTER 33

0.8.0

- Swap in eth-hash for pysha3
- Convert keccak from force_bytes
- Convert address utils from force text/bytes
- Import many of the application functions from web3.py
- Add *@combomethod* decorator
- Add tool to generate environment info
- Add type conversion helpers
- Convert precision to localcontext
- Remove unnecessary future imports
- Drop support for py27

CHAPTER 34

0.7.4

- Constrain dependencies to major version

CHAPTER 35

0.7.3

- Support for python 3.6

CHAPTER 36

0.7.2

- Minor fix for how *__version__* is computed in the *eth_utils* module.

CHAPTER 37

0.7.1

- Futzing with PyPi formatting of README info.

CHAPTER 38

0.7.0

- Rename library on pypi to *eth_utils*

CHAPTER 39

0.6.0

- Bugfix for *to_wei* to handle floating point inputs in a manner consistent with what users would expect.

CHAPTER 40

0.5.1

- Bugfix for *is_hex* to prevent exceptions from being raised for non-hexadecimal inputs.

CHAPTER 41

0.5.0

- *is_hex* now supports both empty string as *0x* and odd length hexadecimal strings.

CHAPTER 42

0.4.1

- Bugfix for currency conversions which retained too high a precision.

CHAPTER 43

0.4.0

- *is_address* will now verify the checksum on any address which passes the *is_checksum_formatted_address* check.

CHAPTER 44

0.3.2

- Added *is_hex*.

CHAPTER 45

0.3.1

- Added *big_endian_to_int* and *int_to_big_endian*.

CHAPTER 46

0.3.0

- Deprecate *compose*
- Bugfix for *is_0x_prefixed* to correctly detect uppercase *X* as part of the prefix.
- Added *is_hex_address*
- Added *is_binary_address*
- Added *is_32byte_address*
- Added *is_checksum_formatted_address*
- Added *apply_to_return_value*
- Added *to_set*
- Added *is_list*
- Added *is_tuple*

CHAPTER 47

0.2.1

- Strip whitespace from event signatures in *event_signature_to_log_topic*

CHAPTER 48

0.2.1

- Strip whitespace from event signatures in *function_signature_to_4byte_selector*

CHAPTER 49

0.2.0

Initial release

CHAPTER 50

Indices and tables

- genindex
- modindex
- search