ethereum-accounts Documentation

Release 0.1.0

Jannik Luhn

Sep 30, 2017

Contents:

ethereum-accounts is a Python package for working with Ethereum accounts. Its main features are keystore import and export, as well as message and transaction signing. Seamless integration with web3.py using its middleware API allows sending transactions even when the RPC node does not manage the user's private keys.

We'll start with a *short demo* to give you an overview over the functionality of the package. After that, I recommed having a quick look at the *encoding conventions chapter* because encoding parameters in a wrong format tends to be a common source of trouble (for me, that is). Then, visit other chapters or the automatically generated *API docs* according to your needs. Have fun!

CHAPTER 1

Quickstart

Installation

```
$ pip install ethereum-accounts
```

Account creation

```
>>> from eth_accounts import Account
>>> account = Account.from_private_key('0xff')
>>> with open('tests/testdata/pbkdf2_keystore_template.json') as f:
...     another_account = Account.from_keystore(f, b'password')
...
>>> third_account = Account.new()  # with random private key
```

```
>>> account.private_key
'0x00000000000000000000000000000000000000000000000000000000000000ff'
>>> account.address
'0x5044a80bD3eff58302e638018534BbDA8896c48A'
```

Message signing

```
>>> from eth_accounts import prepare_ethereum_message, recover_signer
>>> message = prepare_ethereum_message(b'Do it.')
>>> signature = account.sign_message(message)
```

```
>>> recover_signer(signature, message)
'0x5044a80bD3eff58302e638018534BbDA8896c48A'
```

```
>>> account.is_signer(signature, message)
True
```

Web3 integration

```
>>> from web3 import Web3
>>> web3 = Web3(Web3.RPCProvider())
>>> web3.add_middleware(account.local_signing_middleware)
>>> web3.eth.sendTransaction({
...     'from': account.address,
...     'to': another_account.address,
...     'value': 100
... })  # will be signed locally and subsequently sent to the node
```

CHAPTER 2

Encoding conventions

First things first, we follow web3.py's conventions: We call strings either bytes or text, depending on if they are binary (`b'test'`) or unicode (`u'test'` or simply `'test'`).

In general, *ethereum-accounts* returns data in consistent formats. On the other hand, it tries to be as forgiving as possible when it comes to accepting data, but only if it can do so unambigiously. If this is not possible, it raises an exception (typically a `ValueError` or `TypeError`).

The return format of private keys, public keys, addresses, messages, and signatures is hex encoded with a `'0x'`-prefix. In addition, private keys are left-padded with zeros to a length of 32 bytes (or 64 characters + 2 for the prefix). Case is always lower, with the exception of addresses which are checksummed according to EIP-55.

The objects listed above are interpreted correctly when giving in one of the following formats:

1. hex encoded text (with or without `'0x'`-prefix, case insensitive, not necessarily padded to the correct length, but of even length)
2. as bytes (not necessarily padded)

Private keys can also be specified as integers. If addresses are not all lower case, they are interpreted as EIP-55-checksummed and rejected if the checksum is wrong.

Finally, passwords must always be given as bytes to avoid any decoding ambiguities.

CHAPTER 3

Account Creation

Accounts are represented by the `Account` class. They are always based on a private key, that can either be given explicitly, extracted from a keystore file or generated randomly. Accordingly, three methods are available to create account objects:

```
>>> from eth_accounts import Account
>>> account = Account.from_private_key('0xff')
>>> with open('tests/testdata/pbkdf2_keystore_template.json') as keystore_file:
...     another_account = Account.from_keystore(keystore_file, b'password')
...
>>> third_account = Account.new()
```

After initialization, the private key as well as the inferred public key and address are accessible via properties:

```
>>> account.private_key
'0x00000000000000000000000000000000000000000000000000000000000000ff'
>>> account.public_key

↪'0x041b38903a43f7f114ed4500b4eac7083fdefece1cf29c63528d563446f972c1804036edc931a60ae889353f77fd53de
↪'
>>> account.address
'0x5044a80bD3eff58302e638018534BbDA8896c48A'
```

Note that all output is hex encoded and the address EIP55 checksummed.

Accounts that have been imported from keystores, have two additional properties: The address found in the keystore in plain text (which usually but not necessarily is the same as the actual address) and an identifier:

```
>>> account.exposed_address
'88e422c8c5f12f7a484c7bbd070b14a027d55364'
>>> account.id
'386aff9b-9e44-4b3c-b731-37d64726e757'
```

If those are not available, they fall back to `None`.

CHAPTER 4

Signatures

The purpose of Ethereum accounts is to sign messages. Most generally, this task is fulfilled by `Account.sign_message()` which either takes the message as bytes or hex encoded:

```
>>> signature = account.sign_message(b'do it')
>>> signature

↪'0xdc1b25b5085ee83fcabed1c08902e42755aa94eb4f89c1c5def1b995911218014c3dc5ce9d5ec7f1b481a07a7bbab1ab
↪'
>>> from eth_utils import encode_hex
>>> assert account.sign_message(encode_hex(b'do it')) == signature
```

Commonly, the message is hashed before signing. If you'd like you can do this by yourself:

```
>>> from eth_utils import keccak
>>> hashed_message = keccak(b'do it')
>>> assert signature == account.sign_message(hashed_message, hash=False)
```

In some cases, human readable text is to be signed. Instead of passing it directly (where it would be interpreted as hex and hopefully lead to an exception), encode it first:

```
>>> import codecs
>>> message = codecs.encode('Drö Chönösön möt döm Köntröböss', 'utf-8')
>>> signature = account.sign_message(message)
```

To subsequently validate a signature two methods are available: `Account.is_signer()` checks if the account has signed the message and `recover_signer()` returns the signer's address.

```
>>> from eth_accounts import recover_signer
>>> from eth_utils import is_same_address
>>> assert account.is_signer(signature, message)
>>> assert is_same_address(recover_signer(signature, message), account.address)
```

Following the signing function, here `hash=False` can be specified as well.

Often, not arbitrary messages but Ethereum transactions are to be signed. Of this, `Account.sign_transaction()` takes care. It expects the unsigned transaction to be passed as RLP-serializable object, implemented for example in pyethereum or in a basic form in this package. Finally, due to replay protection according to EIP-155 the target network id has to be specified:

```
>>> from eth_accounts import Transaction
>>> from eth_utils import decode_hex
>>> tx = Transaction(
...     nonce=0,
...     gasprice=30 * 10**9,
...     startgas=21000,
...     to=decode_hex('0x' + 20 * '00'),
...     value=10**18,
...     data=b'',
...     v=0, r=0, s=0  # the signature to calculate
... )
>>> account.sign_transaction(tx, network_id=1)  # main net
>>> tx.v, tx.r, tx.s
(37, 58532937890638004285825567298708718952681745693284428409123298183772432557576,␣
↪80112792867190359596305302001287599643804286436274449000091967150142525216 6)
```

Validating the signer of a transaction is faciliated by `Account.is_sender()` and `recover_sender()`:

```
>>> assert account.is_sender(tx, network_id=1)
>>> from eth_accounts import recover_sender
>>> assert is_same_address(recover_sender(tx, network_id=1), account.address)
```

CHAPTER 5

Web3 Integration

Typically, accounts are managed by Ethereum clients such as Geth and Parity. Transaction templates are sent to them via RPC calls (`eth_sendTransaction`), where they are signed and subsequently distributed to the network. However, often this is not desired, especially if the client is remote and cannot be trusted (e.g., Infura's publicly accessible nodes).

As an alternative, transactions can be created and signed locally and then sent via `eth_sendRawTransaction`. To simplify this progress, this package provides middleware for web3.py, the canonical Python package for communication with Ethereum nodes.

In order to use this feature, the middleware has to be registered first:

```
>>> from web3 import Web3
>>> web3 = Web3(Web3.RPCProvider())
>>> web3.add_middleware(account.local_signing_middleware)
```

Now, web3 can be used as usual, but all transactions originating from the account are signed locally:

```
>>> from eth_utils import denoms
>>> other_account = Account.from_private_key('0xaa')
>>> web3.eth.sendTransaction({
...     'from': account.address,
...     'to': other_account.address,
...     'value': 10 * denoms.finney
... })
'0xcb34b55f681a226b994cee10553978952ff82f5bc731a97131ce2b361e42ad75'
>>> web3.eth.getBalance(other_account.address) / denoms.finney
10.0
```

```
>>> token_contract = web3.eth.contract(address=contract_address, abi=contract_abi)
>>> token_contract.transact({'from': account.address}).transfer(other_account.address,
↪ 100000)
'0xde029a35e40809757ddd22a98a0e62da419e4f791eed20846a1eedad42a93c46'
>>> token_contract.call().balanceOf(other_account.address)
100000
```

To extend this to other accounts, add them as middlewares as well:

```
>>> for account in [Account.new() for _ in range(10)]:
...     web3.add_midleware(account)
```

If for the specified sender no local signing middleware is registered, it goes through to the remote node unmodified.

CHAPTER 6

Keystore Export

Exporting accounts to keystore files is possible via the `Account.to_keystore()` method:

```
>>> with open('keystore.json', 'w') as keystore_file:
...     account.to_keystore(keystore_file, b'password')
```

Instead of writing the keystore to a file, it can also be returned in form of a dictionary:

```
>>> d = account.to_keystore_dict(b'password')
```

Both methods allow extensive customization of the result. Most importantly, the key derivation function (KDF) can be specified. Currently, PBKDF2 and scrypt are supported:

```
>>> pbkdf2_keystore = account.to_keystore_dict(b'password', kdf='pbkdf2')
>>> scrypt_keystore = account.to_keystore_dict(b'password', kdf='scrypt')
```

Typically, the KDF have to be parametrized. Sensible defaults are chosen, but those can individually be overridden if desired:

```
>>> pbkdf2_keystore = account.to_keystore_dict(b'password', kdf='pbkdf2',
...                                            kdf_params={'salt': '0xff'})
```

The same applies to the cipher, but here only the canonical `'aes-128-ctr'` is supported:

```
>>> keystore = account.to_keystore_dict(b'password', cipher='aes-128-ctr',
...                                     cipher_params={'iv': '0xff'})
```

Warning: The security of the keystore depends on both KDF salt and cipher IV being random (which they are by default). Don't override those or any other parameters unless you know what you are doing.

Exposure of the address can be prevented by setting `expose_address` to `False`:

```
>>> keystore = account.to_keystore_dict(b'password', expose_address=False)
>>> assert account.from_keystore(keystore, b'password').exposed_account is None
```

Finally, the keystore's ID can be customized. By default (`uuid=True`) a random UUID will be generated. To use a custom value, pass it as the argument. Setting it to `False` or `None` will result in no ID appearing in the keystore.

```
>>> keystore = account.to_keystore_dict(b'password', uuid=None)
>>> assert account.from_keystore(keystore, b'password').uuid is None
>>> keystore = account.to_keystore_dict(b'password', uuid='some-random-id')
>>> assert account.from_keystore(keystore, b'password').uuid == 'some-random-id'
```

Note: Importing an account from a keystore file and exporting it again will by default lead to keystores with different IDs. If this is not desired, make it explicit:

```
>>> keystore = account.to_keystore_dict(b'password', uuid=account.uuid)
```

CHAPTER 7

API

Account

Signing

Utils

Exceptions

CHAPTER 8

Indices and tables

- genindex
- modindex
- search