eth-account Documentation

Release 0.11.0

The Ethereum Foundation

Feb 05, 2024

CONTENTS

Sign Ethereum transactions and messages with local private keys

CHAPTER

ONE

CONTENTS

1.1 eth_account

1.1.1 Account

class eth_account.account.**Account**

The primary entry point for working with Ethereum private keys.

It does **not** require a connection to an Ethereum node.

create(*extra_entropy=''*)

Creates a new private key, and returns it as a `LocalAccount`.

Parameters

extra_entropy (*str or bytes or int*) – Add extra randomness to whatever randomness your OS can provide

Returns

an object with private key and convenience methods

```
>>> from eth_account import Account
>>> acct = Account.create('KEYSMASH FJAFJKLDSKF7JKFDJ 1530')
>>> acct.address
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'
>>> acct.key
HexBytes('0x8676e9a8c86c8921e922e61e0bb6e9e9689aad4c99082620610b00140e5f21b8')

# These methods are also available: sign_message(), sign_transaction(),
# encrypt().
# They correspond to the same-named methods in Account.*
# but without the private key argument
```

create_with_mnemonic(*passphrase: str = ''*, *num_words: int = 12*, *language: str = 'english'*, *account_path: str = "m/44'/60'/0'/0/0"*) → Tuple[*LocalAccount*, str]

Create a new private key and related mnemonic.

> **Caution:** This feature is experimental, unaudited, and likely to change soon

Creates a new private key, and returns it as a `LocalAccount`, alongside the mnemonic that can used to regenerate it using any BIP39-compatible wallet.

Parameters

- **passphrase** (*str*) – Extra passphrase to encrypt the seed phrase
- **num_words** (*int*) – Number of words to use with seed phrase. Default is 12 words. Must be one of [12, 15, 18, 21, 24].
- **language** (*str*) – Language to use for BIP39 mnemonic seed phrase.
- **account_path** (*str*) – Specify an alternate HD path for deriving the seed using BIP32 HD wallet key derivation.

Returns

A tuple consisting of an object with private key and convenience methods, and the mnemonic seed phrase that can be used to restore the account.

Return type

(*LocalAccount*, str)

```
>>> from eth_account import Account
>>> Account.enable_unaudited_hdwallet_features()
>>> acct, mnemonic = Account.create_with_mnemonic()
>>> acct.address
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'
>>> acct == Account.from_mnemonic(mnemonic)
True

# These methods are also available:
# sign_message(), sign_transaction(), encrypt()
# They correspond to the same-named methods in Account.*
# but without the private key argument
```

static decrypt(*keyfile_json*, *password*)

Decrypts a private key.

The key may have been encrypted using an Ethereum client or *encrypt()*.

Parameters

- **keyfile_json** (*dict or str*) – The encrypted key
- **password** (*str*) – The password that was used to encrypt the key

Returns

the raw private key

Return type

HexBytes

```
>>> encrypted = {
... 'address': '5ce9454909639D2D17A3F753ce7d93fa0b9aB12E',
... 'crypto': {'cipher': 'aes-128-ctr',
...  'cipherparams': {'iv': '482ef54775b0cc59f25717711286f5c8'},
...  'ciphertext':
↪'cb636716a9fd46adbb31832d964df2082536edd5399a3393327dc89b0193a2be',
...  'kdf': 'scrypt',
...  'kdfparams': {},
...  'kdfparams': {'dklen': 32,
...                'n': 262144,
```

(continues on next page)

(continued from previous page)

```
...                 'p': 8,
...                 'r': 1,
...                 'salt': 'd3c9a9945000fcb6c9df0f854266d573'},
...  'mac': '4f626ec5e7fea391b2229348a65bfef532c2a4e8372c0a6a814505a350a7689d'},
... 'id': 'b812f3f9-78cc-462a-9e89-74418aa27cb0',
... 'version': 3}
>>> Account.decrypt(encrypted, 'password')
HexBytes('0xb25c7db31feed9122727bf0939dc769a96564b2de4c4726d035b36ecf1e5b364')
```

classmethod enable_unaudited_hdwallet_features()

Use this flag to enable unaudited HD Wallet features.

classmethod encrypt(*private_key*, *password*, *kdf=None*, *iterations=None*)

Creates a dictionary with an encrypted version of your private key. To import this keyfile into Ethereum clients like geth and parity: encode this dictionary with `json.dumps()` and save it to disk where your client keeps key files.

Parameters

- **private_key** (hex str, bytes, int or `eth_keys.datatypes.PrivateKey`) – The raw private key
- **password** (*str*) – The password which you will need to unlock the account in your client
- **kdf** (*str*) – The key derivation function to use when encrypting your private key
- **iterations** (*int*) – The work factor for the key derivation function

Returns

The data to use in your encrypted file

Return type

dict

If kdf is not set, the default key derivation function falls back to the environment variable `ETH_ACCOUNT_KDF`. If that is not set, then 'scrypt' will be used as the default.

```
>>> from pprint import pprint
>>> encrypted = Account.encrypt(
...     0xb25c7db31feed9122727bf0939dc769a96564b2de4c4726d035b36ecf1e5b364,
...     'password'
... )
>>> pprint(encrypted)
{'address': '5ce9454909639D2D17A3F753ce7d93fa0b9aB12E',
 'crypto': {'cipher': 'aes-128-ctr',
            'cipherparams': {'iv': '...'},
            'ciphertext': '...',
            'kdf': 'scrypt',
            'kdfparams': {'dklen': 32,
                          'n': 262144,
                          'p': 1,
                          'r': 8,
                          'salt': '...'},
            'mac': '...'},
 'id': '...',
 'version': 3}
```

(continues on next page)

(continued from previous page)

```
>>> with open('my-keyfile', 'w') as f:
...     f.write(json.dumps(encrypted))
```

from_key(*private_key*)

Returns a convenient object for working with the given private key.

Parameters

private_key (hex str, bytes, int or `eth_keys.datatypes.PrivateKey`) – The raw private key

Returns

object with methods for signing and encrypting

Return type

LocalAccount

```
>>> acct = Account.from_key(
... 0xb25c7db31feed9122727bf0939dc769a96564b2de4c4726d035b36ecf1e5b364)
>>> acct.address
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'
>>> acct.key
HexBytes('0xb25c7db31feed9122727bf0939dc769a96564b2de4c4726d035b36ecf1e5b364')

# These methods are also available: sign_message(), sign_transaction(),
# encrypt(). They correspond to the same-named methods in Account.*
# but without the private key argument
```

from_mnemonic(*mnemonic: str*, *passphrase: str = ''*, *account_path: str = "m/44'/60'/0'/0/0"*) → *LocalAccount*

Generate an account from a mnemonic.

Caution: This feature is experimental, unaudited, and likely to change soon

Parameters

- **mnemonic** (*str*) – space-separated list of BIP39 mnemonic seed words
- **passphrase** (*str*) – Optional passphrase used to encrypt the mnemonic
- **account_path** (*str*) – Specify an alternate HD path for deriving the seed using BIP32 HD wallet key derivation.

Returns

object with methods for signing and encrypting

Return type

LocalAccount

```
>>> from eth_account import Account
>>> Account.enable_unaudited_hdwallet_features()
>>> acct = Account.from_mnemonic(
...  "coral allow abandon recipe top tray caught video climb similar "
...  "prepare bracket antenna rubber announce gauge volume "
```

(continues on next page)

(continued from previous page)

```
...  "hub hood burden skill immense add acid")
>>> acct.address
'0x9AdA5dAD14d925f4df1378409731a9B71Bc8569d'

# These methods are also available: sign_message(), sign_transaction(),
#  encrypt(). They correspond to the same-named methods in Account.*
# but without the private key argument
```

Or, generate multiple accounts from a mnemonic.

```
>>> from eth_account import Account
>>> Account.enable_unaudited_hdwallet_features()
>>> iterator = 0
>>> for i in range(10):
...     acct = Account.from_mnemonic(
...         "health embark april buyer eternal leopard "
...         "want before nominee head thing tackle",
...         account_path=f"m/44'/60'/0'/0/{iterator}")
...     iterator = iterator + 1
...     acct.address
'0x61Cc15522D06983Ac7aADe23f9d5433d38e78195'
'0x1240460F6E370f28079E5F9B52f9DcB759F051b7'
'0xd30dC9f996539826C646Eb48bb45F6ee1D1474af'
'0x47e64beb58c9A469c5eD086aD231940676b44e7C'
'0x6D39032ffEF9987988a069F52EFe4d95D0770555'
'0x3836A6530D1889853b047799Ecd8827255072e77'
'0xed5490dEfF8d8FfAe45cb4066C3daC7C6BFF6a22'
'0xf04F9Ff322799253bcC6B12762AD127570a092c5'
'0x900F7fa9fbe85BB25b6cdB94Da24D807f7feb213'
'0xa248e118b0D19010387b1B768686cd9B473FA137'
```

Caution: For the love of Bob please do not use this mnemonic, it is for testing purposes only.

recover_message(*signable_message:* SignableMessage, *vrs: Tuple[VRS, VRS, VRS] | None = None*, *signature: bytes = None*) → ChecksumAddress

Get the address of the account that signed the given message. You must specify exactly one of: vrs or signature

Parameters

- **signable_message** – the message that was signed
- **vrs** (*tuple(v, r, s), each element is hex str, bytes or int*) – the three pieces generated by an elliptic curve signature
- **signature** (*hex str or bytes or int*) – signature bytes concatenated as r+s+v

Returns

address of signer, hex-encoded & checksummed

Return type

str

```
>>> from eth_account.messages import encode_defunct
>>> from eth_account import Account
>>> message = encode_defunct(text="ISF")
>>> vrs = (
...   28,
...   '0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb3',
...   '0x3e5bfbbf4d3e39b1a2fd816a7680c19ebebaf3a141b239934ad43cb33fcec8ce')
>>> Account.recover_message(message, vrs=vrs)
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'


# All of these recover calls are equivalent:

# variations on vrs
>>> vrs = (
...   '0x1c',
...   '0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb3',
...   '0x3e5bfbbf4d3e39b1a2fd816a7680c19ebebaf3a141b239934ad43cb33fcec8ce')
>>> Account.recover_message(message, vrs=vrs)
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'

>>> # Caution about this approach: likely problems if there are leading 0s
>>> vrs = (
...   0x1c,
...   0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb3,
...   0x3e5bfbbf4d3e39b1a2fd816a7680c19ebebaf3a141b239934ad43cb33fcec8ce)
>>> Account.recover_message(message, vrs=vrs)
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'

>>> vrs = (
...   b'\x1c',
...   b'\xe6\xca\x9b\xbaX\xc8\x86\x11\xfa\xd6jl\xe8\xf9\x96\x90\x81\x95Y8\x07\
↪xc4\xb3\x8b\xd5(\xd2\xcf\xf0\x9dN\xb3',
...   b'>[\xfb\xbfM>9\xb1\xa2\xfd\x81jv\x80\xc1\x9e\xbe\xba\xf3\xa1A\xb29\x93J\
↪xd4<\xb3?\xce\xc8\xce')
>>> Account.recover_message(message, vrs=vrs)
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'

# variations on signature
>>> signature =
↪'0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb33e5bfbbf4d3e39b1a2fd816a7
↪'
>>> Account.recover_message(message, signature=signature)
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'
>>> signature = b'\xe6\xca\x9b\xbaX\xc8\x86\x11\xfa\xd6jl\xe8\xf9\x96\x90\x81\
↪x95Y8\x07\xc4\xb3\x8b\xd5(\xd2\xcf\xf0\x9dN\xb3>[\xfb\xbfM>9\xb1\xa2\xfd\
↪x81jv\x80\xc1\x9e\xbe\xba\xf3\xa1A\xb29\x93J\xd4<\xb3?\xce\xc8\xce\x1c'
>>> Account.recover_message(message, signature=signature)
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'
>>> # Caution about this approach: likely problems if there are leading 0s
>>> signature =␣
↪0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb33e5bfbbf4d3e39b1a2fd816a768
>>> Account.recover_message(message, signature=signature)
```

(continues on next page)

(continued from previous page)

```
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'
```

recover_transaction(*serialized_transaction*)

Get the address of the account that signed this transaction.

Parameters

serialized_transaction (*hex str, bytes or int*) – the complete signed transaction

Returns

address of signer, hex-encoded & checksummed

Return type

str

```
>>> raw_transaction =
↪'0xf86a8086d55698372431831e848094f0109fc8df283027b6285cc889f5aa624eac1f55843b9aca008025a009e
↪'
>>> Account.recover_transaction(raw_transaction)
'0x2c7536E3605D9C16a7a3D7b1898e529396a65c23'
```

set_key_backend(*backend*)

Change the backend used by the underlying eth-keys library.

(The default is fine for most users)

Parameters

backend – any backend that works in eth_keys.KeyApi(backend)

signHash(*message_hash*, *private_key*)

Sign the provided hash.

> **Warning:** *Never* sign a hash that you didn't generate, it can be an arbitrary transaction. For example, it might send all of your account's ether to an attacker. Instead, prefer *sign_message()*, which cannot accidentally sign a transaction.

> **Caution:** Deprecated for *sign_message()*. This method will be removed in v0.6

Parameters

- **message_hash** (*hex str, bytes or int*) – the 32-byte message hash to be signed
- **private_key** (hex str, bytes, int or `eth_keys.datatypes.PrivateKey`) – the key to sign the message with

Returns

Various details about the signature - most importantly the fields: v, r, and s

Return type

SignedMessage

sign_message(*signable_message: SignableMessage*, *private_key: bytes | HexStr | int | PrivateKey*) → *SignedMessage*

Sign the provided message.

This API supports any messaging format that will encode to EIP-191 messages.

If you would like historical compatibility with `w3.eth.sign()` you can use *encode_defunct()*.

Other options are the "validator", or "structured data" standards. You can import all supported message encoders in `eth_account.messages`.

Parameters

- **signable_message** – the encoded message for signing
- **private_key** (hex str, bytes, int or `eth_keys.datatypes.PrivateKey`) – the key to sign the message with

Returns

Various details about the signature - most importantly the fields: v, r, and s

Return type

SignedMessage

```
>>> msg = "ISF"
>>> from eth_account.messages import encode_defunct
>>> msghash = encode_defunct(text=msg)
>>> msghash
SignableMessage(version=b'E',
 header=b'thereum Signed Message:\n6',
 body=b'I\xe2\x99\xa5SF')
>>> # If you're curious about the internal fields of SignableMessage, take a␣
↪look at EIP-191, linked above
>>> key = "0xb25c7db31feed9122727bf0939dc769a96564b2de4c4726d035b36ecf1e5b364"
>>> Account.sign_message(msghash, key)
SignedMessage(messageHash=HexBytes(
↪'0x1476abb745d423bf09273f1afd887d951181d25adc66c4834a70491911b7f750'),
␣
↪r=104389933075820307925104709181714897380569894203213074526835978196648170704563,
↪
␣
↪s=28205917190874851400050446352651915501321657673772411533993420917949420456142,
↪
 v=28,
 signature=HexBytes(
↪'0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb33e5bfbbf4d3e39b1a2fd816a7
↪'))
```

`sign_transaction`(*transaction_dict*, *private_key*)

Sign a transaction using a local private key.

It produces signature details and the hex-encoded transaction suitable for broadcast using `w3.eth.sendRawTransaction()`.

To create the transaction dict that calls a contract, use contract object: my_contract.functions.my_function().buildTransaction()

Note: For non-legacy (typed) transactions, if the transaction type is not explicitly provided, it may be determined from the transaction parameters of a well-formed transaction. See below for examples on how to sign with different transaction types.

Parameters

- **transaction_dict** (*dict*) – the transaction with available keys, depending on the type of transaction: nonce, chainId, to, data, value, gas, gasPrice, type, accessList, maxFeePerGas, and maxPriorityFeePerGas
- **private_key** (hex str, bytes, int or `eth_keys.datatypes.PrivateKey`) – the private key to sign the data with

Returns
Various details about the signature - most importantly the fields: v, r, and s

Return type
AttributeDict

```
>>> # EIP-1559 dynamic fee transaction (more efficient and preferred over␣
↪legacy txn)
>>> dynamic_fee_transaction = {
        "type": 2,  # optional - can be implicitly determined based on max fee␣
↪params
        "gas": 100000,
        "maxFeePerGas": 2000000000,
        "maxPriorityFeePerGas": 2000000000,
        "data": "0x616263646566",
        "nonce": 34,
        "to": "0x09616C3d61b3331fc4109a9E41a8BDB7d9776609",
        "value": "0x5af3107a4000",
        "accessList": (  # optional
            {
                "address": "0x0000000000000000000000000000000000000001",
                "storageKeys": (

↪"0x0100000000000000000000000000000000000000000000000000000000000000",
                )
            },
        ),
        "chainId": 1900,
    }
>>> key = '0x4c0883a69102937d6231471b5dbb6204fe5129617082792ae468d01a3f362318'
>>> signed = Account.sign_transaction(dynamic_fee_transaction, key)
{'hash': HexBytes(
↪'0x126431f2a7fda003aada7c2ce52b0ce3cbdbb1896230d3333b9eea24f42d15b0'),

 'r':␣
↪110093478023675319011132687961420618950720745285952062287904334878381994888509,
↪
 'rawTransaction': HexBytes(
↪'0x02f8b282076c228477359400847735940083018 6a09409616c3d61b3331fc4109a9e41a8bdb7d9776609865af
↪'),
 's':␣
↪33674551144139401179914073499472892825822542092106065756005379322302694600392,

 'v': 0}
>>> w3.eth.sendRawTransaction(signed.rawTransaction)
```

```
>>> # legacy transaction (less efficient than EIP-1559 dynamic fee txn)
>>> legacy_transaction = {
```

(continues on next page)

(continued from previous page)

```
        # Note that the address must be in checksum format or native bytes:
        'to': '0xF0109fC8DF283027b6285cc889F5aA624EaC1F55',
        'value': 1000000000,
        'gas': 2000000,
        'gasPrice': 234567897654321,
        'nonce': 0,
        'chainId': 1
    }
>>> key = '0x4c0883a69102937d6231471b5dbb6204fe5129617082792ae468d01a3f362318'
>>> signed = Account.sign_transaction(legacy_transaction, key)
{'hash': HexBytes(
↪'0x6893a6ee8df79b0f5d64a180cd1ef35d030f3e296a5361cf04d02ce720d32ec5'),
 'r':␣
↪44872862617934181798178410248897471157793243053758231102491494799050751740 44,
 'rawTransaction': HexBytes(
↪'0xf86a8086d55698372431831e848094f0109fc8df283027b6285cc889f5aa624eac1f55843b9aca008025a009e
↪'),
 's':␣
↪30785525769477805655994251009256770582792548537338581640010273753578382951464,
 'v': 37}
>>> w3.eth.sendRawTransaction(signed.rawTransaction)
```

```
>>> access_list_transaction = {
        "type": 1,  # optional - can be implicitly determined based on
↪'accessList' and 'gasPrice' params
        "gas": 100000,
        "gasPrice": 1000000000,
        "data": "0x616263646566",
        "nonce": 34,
        "to": "0x09616C3d61b3331fc4109a9E41a8BDB7d9776609",
        "value": "0x5af3107a4000",
        "accessList": (
            {
                "address": "0x0000000000000000000000000000000000000001",
                "storageKeys": (

↪"0x0100000000000000000000000000000000000000000000000000000000000000",
                )
            },
        ),
        "chainId": 1900,
    }
>>> key = '0x4c0883a69102937d6231471b5dbb6204fe5129617082792ae468d01a3f362318'
>>> signed = Account.sign_transaction(access_list_transaction, key)
{'hash': HexBytes(
↪'0x2864ca20a74ca5e044067ad4139a22ff5a0853434f5f1dc00108f24ef5f1f783'),
 'r':␣
↪10594070506339162847235188389409193531714289011444057083140940067673687319770 2,
↪
 'rawTransaction': HexBytes(
↪'0x01f8ad82076c22843b9aca00830186a09409616c3d61b3331fc4109a9e41a8bdb7d9776609865af3107a40008
↪'),
```

(continues on next page)

(continued from previous page)

```
 's':␣
↪37050226636175381535892585331727388340134760347943439553552848647212419749796,
 'v': 0}
>>> w3.eth.sendRawTransaction(signed.rawTransaction)
```

sign_typed_data(*private_key: bytes | HexStr | int | PrivateKey*, *domain_data: Dict[str, Any] = None*, *message_types: Dict[str, Any] = None*, *message_data: Dict[str, Any] = None*, *full_message: Dict[str, Any] = None*) → *SignedMessage*

Sign the provided EIP-712 message with the provided key.

Parameters

- **private_key** (hex str, bytes, int or `eth_keys.datatypes.PrivateKey`) – the key to sign the message with
- **domain_data** (*dict*) – EIP712 domain data
- **message_types** (*dict*) – custom types used by the *value* data
- **message_data** (*dict*) – data to be signed
- **full_message** (*dict*) – a dict containing all data and types

Returns
Various details about the signature - most importantly the fields: v, r, and s

Return type
SignedMessage

You may supply the information to be encoded in one of two ways:

As exactly three arguments:

- `domain_data`, a dict of the EIP-712 domain data
- `message_types`, a dict of custom types (do not include a `EIP712Domain` key)
- `message_data`, a dict of the data to be signed

Or as a single argument:

- **`full_message`, a dict containing the following keys:**
  - `types`, a dict of custom types (may include a `EIP712Domain` key)
  - `primaryType`, (optional) a string of the primary type of the message
  - `domain`, a dict of the EIP-712 domain data
  - `message`, a dict of the data to be signed

> **Warning:** Note that this code has not gone through an external audit, and the test cases are incomplete.

See documentation for *encode_typed_data()* for usage details

See the EIP-712 spec for more information.

```
>>> # examples of basic usage
>>> from eth_account import Account
>>> # 3-argument usage
```

(continues on next page)

(continued from previous page)

```
>>> # all domain properties are optional
>>> domain_data = {
...     "name": "Ether Mail",
...     "version": "1",
...     "chainId": 1,
...     "verifyingContract": "0xCcCCccccCCCCcCCCCCCcCcCccCcCCCcCcccccccC",
...     "salt": b"decafbeef",
... }
>>> # custom types
>>> message_types = {
...     "Person": [
...         {"name": "name", "type": "string"},
...         {"name": "wallet", "type": "address"},
...     ],
...     "Mail": [
...         {"name": "from", "type": "Person"},
...         {"name": "to", "type": "Person"},
...         {"name": "contents", "type": "string"},
...     ],
... }
>>> # the data to be signed
>>> message_data = {
...     "from": {
...         "name": "Cow",
...         "wallet": "0xCD2a3d9F938E13CD947Ec05AbC7FE734Df8DD826",
...     },
...     "to": {
...         "name": "Bob",
...         "wallet": "0xbBbBBBBbbBBBbbbBbbBbbbbBBbBbbbbBbBbbBBbB",
...     },
...     "contents": "Hello, Bob!",
... }
>>> key = "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
>>> signed_message = Account.sign_typed_data(key, domain_data, message_types,␣
↪message_data)
>>> signed_message.messageHash
HexBytes('0xc5bb16ccc59ae9a3ad1cb8343d4e3351f057c994a97656e1aff8c134e56f7530')

>>> # 1-argument usage

>>> # all domain properties are optional
>>> full_message = {
...     "types": {
...         "EIP712Domain": [
...             {"name": "name", "type": "string"},
...             {"name": "version", "type": "string"},
...             {"name": "chainId", "type": "uint256"},
...             {"name": "verifyingContract", "type": "address"},
...             {"name": "salt", "type": "bytes32"},
...         ],
...         "Person": [
```

(continues on next page)

(continued from previous page)

```
...             {"name": "name", "type": "string"},
...             {"name": "wallet", "type": "address"},
...         ],
...         "Mail": [
...             {"name": "from", "type": "Person"},
...             {"name": "to", "type": "Person"},
...             {"name": "contents", "type": "string"},
...         ],
...     },
...     "primaryType": "Mail",
...     "domain": {
...         "name": "Ether Mail",
...         "version": "1",
...         "chainId": 1,
...         "verifyingContract": "0xCcCCccccCCCCcCCCCCCcCcCccCcCCCcCcccccccC",
...         "salt": b"decafbeef"
...     },
...     "message": {
...         "from": {
...             "name": "Cow",
...             "wallet": "0xCD2a3d9F938E13CD947Ec05AbC7FE734Df8DD826"
...         },
...         "to": {
...             "name": "Bob",
...             "wallet": "0xbBbBBBBbbBBBbbbBbbBbbbbBBbBbbbbBbBbbBBbB"
...         },
...         "contents": "Hello, Bob!",
...     },
... }
>>> signed_message_2 = Account.sign_typed_data(key, full_message=full_message)
>>> signed_message_2.messageHash
HexBytes('0xc5bb16ccc59ae9a3ad1cb8343d4e3351f057c994a97656e1aff8c134e56f7530')
>>> signed_message_2 == signed_message
True
```

See *Signers* for alternative signers.

1.1.2 SignedTransaction & SignedMessage

class eth_account.datastructures.**SignedMessage**(*messageHash*, *r*, *s*, *v*, *signature*)

Bases: tuple

messageHash: HexBytes

Alias for field number 0

r: int

Alias for field number 1

s: int

Alias for field number 2

signature: HexBytes

Alias for field number 4

v: int
Alias for field number 3

class eth_account.datastructures.**SignedTransaction**(*rawTransaction*, *hash*, *r*, *s*, *v*)
Bases: tuple

hash: HexBytes
Alias for field number 1

r: int
Alias for field number 2

rawTransaction: HexBytes
Alias for field number 0

s: int
Alias for field number 3

v: int
Alias for field number 4

1.1.3 Messages

class eth_account.messages.**SignableMessage**(*version: bytes*, *header: bytes*, *body: bytes*)
Bases: tuple

A message compatible with EIP-191 that is ready to be signed.

The properties are components of an EIP-191 signable message. Other message formats can be encoded into this format for easy signing. This data structure doesn't need to know about the original message format. For example, you can think of EIP-712 as compiling down to an EIP-191 message.

In typical usage, you should never need to create these by hand. Instead, use one of the available encode_* methods in this module, like:

- *encode_structured_data()*
- *encode_intended_validator()*
- *encode_defunct()*
- *encode_typed_data()*

body: bytes
Alias for field number 2

header: bytes
Alias for field number 1

version: bytes
Alias for field number 0

eth_account.messages.**defunct_hash_message**(*primitive: bytes | None = None*, ***, *hexstr: str | None = None*, *text: str | None = None*) → HexBytes
Convert the provided message into a message hash, to be signed.

> **Caution:** Intended for use with the deprecated *eth_account.account.Account.signHash()*. This is for backwards compatibility only. All new implementations should use *encode_defunct()* instead.

Parameters

- **primitive** (*bytes or int*) – the binary message to be signed
- **hexstr** (*str*) – the message encoded as hex
- **text** (*str*) – the message as a series of unicode characters (a normal Py3 str)

Returns

The hash of the message, after adding the prefix

eth_account.messages.**encode_defunct**(*primitive: bytes | None = None*, ***, *hexstr: str | None = None*, *text: str | None = None*) → *SignableMessage*

Encode a message for signing, using an old, unrecommended approach.

Only use this method if you must have compatibility with `w3.eth.sign()`.

EIP-191 defines this as "version E".

Supply exactly one of the three arguments: bytes, a hex string, or a unicode string.

Parameters

- **primitive** (*bytes or int*) – the binary message to be signed
- **hexstr** (*str*) – the message encoded as hex
- **text** (*str*) – the message as a series of unicode characters (a normal Py3 str)

Returns

The EIP-191 encoded message, ready for signing

```
>>> from eth_account.messages import encode_defunct
>>> from eth_utils.curried import to_hex, to_bytes

>>> message_text = "ISF"
>>> encode_defunct(text=message_text)
SignableMessage(version=b'E',
                header=b'thereum Signed Message:\n6',
                body=b'I\xe2\x99\xa5SF')

These four also produce the same hash:
>>> encode_defunct(to_bytes(text=message_text))
SignableMessage(version=b'E',
                header=b'thereum Signed Message:\n6',
                body=b'I\xe2\x99\xa5SF')

>>> encode_defunct(bytes(message_text, encoding='utf-8'))
SignableMessage(version=b'E',
                header=b'thereum Signed Message:\n6',
                body=b'I\xe2\x99\xa5SF')

>>> to_hex(text=message_text)
'0x49e299a55346'
>>> encode_defunct(hexstr='0x49e299a55346')
SignableMessage(version=b'E',
                header=b'thereum Signed Message:\n6',
                body=b'I\xe2\x99\xa5SF')
```

(continues on next page)

(continued from previous page)

```
>>> encode_defunct(0x49e299a55346)
SignableMessage(version=b'E',
                header=b'thereum Signed Message:\n6',
                body=b'I\xe2\x99\xa5SF')
```

eth_account.messages.**encode_intended_validator**(*validator_address: Address | str*, *primitive: bytes | None = None*, ***, *hexstr: str | None = None*, *text: str | None = None*) → *SignableMessage*

Encode a message using the "intended validator" approach (ie~ version 0) defined in EIP-191.

Supply the message as exactly one of these three arguments: bytes as a primitive, a hex string, or a unicode string.

Warning: Note that this code has not gone through an external audit.

Parameters

- **validator_address** – which on-chain contract is capable of validating this message, provided as a checksummed address or in native bytes.
- **primitive** (*bytes or int*) – the binary message to be signed
- **hexstr** (*str*) – the message encoded as hex
- **text** (*str*) – the message as a series of unicode characters (a normal Py3 str)

Returns

The EIP-191 encoded message, ready for signing

eth_account.messages.**encode_structured_data**(*primitive: bytes | int | Mapping | None = None*, ***, *hexstr: str | None = None*, *text: str | None = None*) → *SignableMessage*

Warning: This method is deprecated. Use *encode_typed_data()* instead.

Encode an EIP-712 message.

EIP-712 is the "structured data" approach (ie~ version 1 of an EIP-191 message).

Supply the message as exactly one of the three arguments:

- primitive, as a dict that defines the structured data
- primitive, as bytes
- text, as a json-encoded string
- hexstr, as a hex-encoded (json-encoded) string

Warning: Note that this code has not gone through an external audit, and the test cases are incomplete.

Parameters

- **primitive** (*bytes or int or Mapping (eg~ dict)*) – the binary message to be signed

- **hexstr** – the message encoded as hex
- **text** – the message as a series of unicode characters (a normal Py3 str)

Returns
The EIP-191 encoded message, ready for signing

Usage Notes:

- An EIP712 message consists of 4 top-level keys: `types`, `primaryType`, `domain`, and `message`. All 4 must be present to encode properly.
- The key `EIP712Domain` must be present within `types`.
- The *type* of a field may be a Solidity type or a *custom* type, i.e., one that is defined within the `types` section of the typed data.
- Extra information in `message` and `domain` will be ignored when encoded. For example, if the custom type `Person` defines the fields `name` and `wallet`, but an additional `id` field is provided in `message`, the resulting encoding will be the same as if the `id` information was not present.
- Unused custom types will be ignored in the same way.

```
>>> # an example of basic usage
>>> import json
>>> from eth_account import Account
>>> from eth_account.messages import encode_structured_data

>>> typed_data = {
...     "types": {
...         "EIP712Domain": [
...             {"name": "name", "type": "string"},
...             {"name": "version", "type": "string"},
...             {"name": "chainId", "type": "uint256"},
...             {"name": "verifyingContract", "type": "address"},
...         ],
...         "Person": [
...             {"name": "name", "type": "string"},
...             {"name": "wallet", "type": "address"},
...         ],
...         "Mail": [
...             {"name": "from", "type": "Person"},
...             {"name": "to", "type": "Person"},
...             {"name": "contents", "type": "string"},
...         ],
...     },
...     "primaryType": "Mail",
...     "domain": {
...         "name": "Ether Mail",
...         "version": "1",
...         "chainId": 1,
...         "verifyingContract": "0xCcCCccccCCCCcCCCCCCcCcCccCcCCCcCcccccccC",
...     },
...     "message": {
...         "from": {
...             "name": "Cow",
```

(continues on next page)

(continued from previous page)

```
...             "wallet": "0xCD2a3d9F938E13CD947Ec05AbC7FE734Df8DD826"
...         },
...         "to": {
...             "name": "Bob",
...             "wallet": "0xbBbBBBBbbBBBbbbBbbBbbbbBBbBbbbbBbBbbBBbB"
...         },
...         "contents": "Hello, Bob!",
...     },
... }

>>> key = "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"

>>> signable_msg_from_dict = encode_structured_data(typed_data)
>>> signable_msg_from_str = encode_structured_data(text=json.dumps(typed_data))
>>> signable_msg_from_hexstr = encode_structured_data(
...     hexstr=json.dumps(typed_data).encode("utf-8").hex()
... )

>>> signed_msg_from_dict = Account.sign_message(signable_msg_from_dict, key)
>>> signed_msg_from_str = Account.sign_message(signable_msg_from_str, key)
>>> signed_msg_from_hexstr = Account.sign_message(signable_msg_from_hexstr, key)

>>> signed_msg_from_dict == signed_msg_from_str == signed_msg_from_hexstr
True
>>> signed_msg_from_dict.messageHash
HexBytes('0xbe609aee343fb3c4b28e1df9e632fca64fcfaede20f02e86244efddf30957bd2')
```

eth_account.messages.**encode_typed_data**(*domain_data: Dict[str, Any] | None = None*, *message_types: Dict[str, Any] | None = None*, *message_data: Dict[str, Any] | None = None*, *full_message: Dict[str, Any] | None = None*) → *SignableMessage*

Encode an EIP-712 message in a manner compatible with other implementations in use, such as the Metamask and Ethers `signTypedData` functions.

See the EIP-712 spec for more information.

You may supply the information to be encoded in one of two ways:

As exactly three arguments:

- `domain_data`, a dict of the EIP-712 domain data
- `message_types`, a dict of custom types (do not include a `EIP712Domain` key)
- `message_data`, a dict of the data to be signed

Or as a single argument:

- **`full_message`, a dict containing the following keys:**
  - `types`, a dict of custom types (may include a `EIP712Domain` key)
  - `primaryType`, (optional) a string of the primary type of the message
  - `domain`, a dict of the EIP-712 domain data
  - `message`, a dict of the data to be signed

Warning: Note that this code has not gone through an external audit, and the test cases are incomplete.

Type Coercion:

- For fixed-size bytes types, smaller values will be padded to fit in larger types, but values larger than the type will raise `ValueOutOfBounds`. e.g., an 8-byte value will be padded to fit a `bytes16` type, but 16-byte value provided for a `bytes8` type will raise an error.
- Fixed-size and dynamic `bytes` types will accept `int``s. Any negative values will be converted to ``0` before being converted to `bytes`
- `int` and `uint` types will also accept strings. If prefixed with `"0x"` , the string will be interpreted as hex. Otherwise, it will be interpreted as decimal.

Noteable differences from `signTypedData`:

- Custom types that are not alphanumeric will encode differently.
- Custom types that are used but not defined in `types` will not encode.

Parameters

- **domain_data** – EIP712 domain data
- **message_types** – custom types used by the *value* data
- **message_data** – data to be signed
- **full_message** – a dict containing all data and types

Returns

a `SignableMessage`, an encoded message ready to be signed

```
>>> # examples of basic usage
>>> from eth_account import Account
>>> from eth_account.messages import encode_typed_data
>>> # 3-argument usage

>>> # all domain properties are optional
>>> domain_data = {
...     "name": "Ether Mail",
...     "version": "1",
...     "chainId": 1,
...     "verifyingContract": "0xCcCCccccCCCCcCCCCCCcCcCccCcCCCcCcccccccC",
...     "salt": b"decafbeef",
... }
>>> # custom types
>>> message_types = {
...     "Person": [
...         {"name": "name", "type": "string"},
...         {"name": "wallet", "type": "address"},
...     ],
...     "Mail": [
...         {"name": "from", "type": "Person"},
...         {"name": "to", "type": "Person"},
...         {"name": "contents", "type": "string"},
...     ],
```

(continues on next page)

(continued from previous page)

```
... }
>>> # the data to be signed
>>> message_data = {
...     "from": {
...         "name": "Cow",
...         "wallet": "0xCD2a3d9F938E13CD947Ec05AbC7FE734Df8DD826",
...     },
...     "to": {
...         "name": "Bob",
...         "wallet": "0xbBbBBBBbbBBBbbbBbbBbbbbBBbBbbbbBbBbbBBbB",
...     },
...     "contents": "Hello, Bob!",
... }
>>> key = "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
>>> signable_message = encode_typed_data(domain_data, message_types, message_data)
>>> signed_message = Account.sign_message(signable_message, key)
>>> signed_message.messageHash
HexBytes('0xc5bb16ccc59ae9a3ad1cb8343d4e3351f057c994a97656e1aff8c134e56f7530')
>>> # the message can be signed in one step using Account.sign_typed_data
>>> signed_typed_data = Account.sign_typed_data(key, domain_data, message_types,␣
↪message_data)
>>> signed_typed_data == signed_message
True

>>> # 1-argument usage

>>> # all domain properties are optional
>>> full_message = {
...     "types": {
...         "EIP712Domain": [
...             {"name": "name", "type": "string"},
...             {"name": "version", "type": "string"},
...             {"name": "chainId", "type": "uint256"},
...             {"name": "verifyingContract", "type": "address"},
...             {"name": "salt", "type": "bytes32"},
...         ],
...         "Person": [
...             {"name": "name", "type": "string"},
...             {"name": "wallet", "type": "address"},
...         ],
...         "Mail": [
...             {"name": "from", "type": "Person"},
...             {"name": "to", "type": "Person"},
...             {"name": "contents", "type": "string"},
...         ],
...     },
...     "primaryType": "Mail",
...     "domain": {
...         "name": "Ether Mail",
...         "version": "1",
...         "chainId": 1,
...         "verifyingContract": "0xCcCCccccCCCCcCCCCCCcCcCccCcCCCcCcccccccC",
```

(continues on next page)

(continued from previous page)

```
...         "salt": b"decafbeef"
...     },
...     "message": {
...         "from": {
...             "name": "Cow",
...             "wallet": "0xCD2a3d9F938E13CD947Ec05AbC7FE734Df8DD826"
...         },
...         "to": {
...             "name": "Bob",
...             "wallet": "0xbBbBBBBbbBBBbbbBbbBbbbbBBbBbbbbBbBbbBBbB"
...         },
...         "contents": "Hello, Bob!",
...     },
... }
>>> signable_message_2 = encode_typed_data(full_message=full_message)
>>> signed_message_2 = Account.sign_message(signable_message_2, key)
>>> signed_message_2.messageHash
HexBytes('0xc5bb16ccc59ae9a3ad1cb8343d4e3351f057c994a97656e1aff8c134e56f7530')
>>> signed_message_2 == signed_message
True
>>> # the full_message can be signed in one step using Account.sign_typed_data
>>> signed_typed_data_2 = Account.sign_typed_data(key, domain_data, message_types,␣
↪message_data)
>>> signed_typed_data_2 == signed_message_2
True
```

1.2 Signers

These classes abstract away the private key, as opposed to *eth_account.account.Account*, which explicitly requires the private key on each usage.

All the signer classes in this package must meet the interface specified by *BaseAccount*.

Currently there is only one Local Signer. Some upcoming alternatives to the basic local signer include hierarchical deterministic (HD) wallets and hardware wallets.

1.2.1 Local Signer

class eth_account.signers.local.**LocalAccount**(*key*, *account*)

Bases: *BaseAccount*

A collection of convenience methods to sign and encrypt, with an embedded private key.

Variables
key (*bytes*) – the 32-byte private key data

```
>>> my_local_account.address
"0xF0109fC8DF283027b6285cc889F5aA624EaC1F55"
>>> my_local_account.key
b"\x01\x23..."
```

You can also get the private key by casting the account to `bytes`:

```
>>> bytes(my_local_account)
b"\\x01\\x23..."
```

property address

The checksummed public address for this account.

```
>>> my_account.address
"0xF0109fC8DF283027b6285cc889F5aA624EaC1F55"
```

encrypt(*password*, *kdf=None*, *iterations=None*)

Generate a string with the encrypted key.

This uses the same structure as in *encrypt()*, but without a private key argument.

property key

Get the private key.

signHash(*message_hash*)

Sign the hash of a message.

This uses the same structure as in *signHash()* but without specifying the private key.

> **Caution:** Deprecated for *sign_message()*. To be removed in v0.6

Parameters

message_hash (*bytes*) – 32 byte hash of the message to sign

sign_message(*signable_message*)

Generate a string with the encrypted key.

This uses the same structure as in *sign_message()*, but without a private key argument.

sign_transaction(*transaction_dict*)

Sign a transaction dict.

This uses the same structure as in *sign_transaction()* but without specifying the private key.

Parameters

transaction_dict (*dict*) – transaction with all fields specified

1.2.2 Abstract Signer

class eth_account.signers.base.**BaseAccount**

Bases: ABC

Specify convenience methods to sign transactions and message hashes.

abstract property address

The checksummed public address for this account.

```
>>> my_account.address
"0xF0109fC8DF283027b6285cc889F5aA624EaC1F55"
```

abstract signHash(*message_hash*)

Sign the hash of a message.

This uses the same structure as in *signHash()* but without specifying the private key.

> **Caution:** Deprecated for *sign_message()*. To be removed in v0.6

Parameters
message_hash (*bytes*) – 32 byte hash of the message to sign

abstract sign_message(*signable_message:* SignableMessage) → *SignedMessage*

Sign the EIP-191 message.

This uses the same structure as in *sign_message()* but without specifying the private key.

Parameters
signable_message – The encoded message, ready for signing

abstract sign_transaction(*transaction_dict*)

Sign a transaction dict.

This uses the same structure as in *sign_transaction()* but without specifying the private key.

Parameters
transaction_dict (*dict*) – transaction with all fields specified

1.3 Release Notes

1.3.1 eth-account v0.11.0 (2024-02-05)

Breaking Changes

- Drop support for python 3.7 (#248)

Internal Changes - for eth-account Contributors

- Change older `%` and `.format` strings to use `f-strings` (#245)
- Merge template updates, notably use `pre-commit` for linting and change the name of the `master` branch to `main` (#248)

Removals

- Remove deprecated `signTransaction`, it has been replaced by `sign_transaction` (#244)

1.3.2 eth-account v0.10.0 (2023-10-30)

Deprecations

- Deprecate `encode_structured_data` in favor of new `encode_typed_data` (#235)

Improved Documentation

- Added usage notes and example for `encode_structured_data` (#233)

Features

- Add new `encode_typed_data` to better handle EIP712 message signing (#235)
- Added option to call `encode_typed_data` with a single dict arg in addition to the existing 3-dict style (#238)
- Add `sign_typed_data` as a method of the `Account` class (#239)

Internal Changes - for eth-account Contributors

- Added tests for `encode_structured_data` for easier comparison with Metamask's SignTypedData (#233)
- Bump version for node and ethers.js in integration tests, update ethers usage to match (#236)
- Add `build.os` to readthedocs settings (#237)
- Add upper pin to `hexbytes` dependency to due incoming breaking change (#240)
- Add tests comparing output of signed EIP712 messages with metamask and ethers (#241)

1.3.3 eth-account v0.9.0 (2023-06-07)

Breaking Changes

- drop python3.6 support from setup (#228)

Improved Documentation

- remove notices of Draft status for eips 712 and 191 (#222)

Features

- Add support for Python 3.11 (#212)

Internal Changes - for eth-account Contributors

- Upgrade Node from v12.x to v18.x in tests (#217)
- pulled full new node_v18 install script (#223)
- bump versions for docs dependencies (#224)
- add sphinx_rtd_theme to docs/conf.py extensions list (#225)
- merge in updates from python project template (#288)

1.3.4 eth-account v0.8.0 (2022-12-15)

Features

- update all references to deprecated *eth_abi.encode_abi* to *eth_abi.encode* (#200)

Performance improvements

- Reduce the number of pbkdf2 iterations to speed up tests (#77)

Deprecations and Removals

- remove deprecated methods that were noted to go in v0.5 (#195)

Internal Changes - for eth-account Contributors

- add coverage reporting to pytest (#192)
- Use updated circleci Python images, fix Sphinx warning (#194)

Miscellaneous changes

- #197, #198, #199, #202, #203, #204, #206

1.3.5 eth-account v0.7.0 (2022-08-17)

Bugfixes

- bump ansi-regex to 5.0.1 to fix minor ReDos vulnerability (#129)
- Enable lint runs again on CI (#166)
- fix DoS-able regex pattern (#178)
- Allow towncrier to build the release notes again (#185)

Improved Documentation

- Add example to generate multiple accounts from a mnemonic (#153)
- Pin Jinja2 at >=3.0.0,<3.1.0; pin towncrier==18.5.0; open up Sphinx requirement to allow >=1.6.5,<5. (#156)
- added missing quotes to readme dev environment setup example (#172)

Miscellaneous changes

- #79, #155, #162, #164, #165

Breaking changes

- Change bitarray dependency requirement to be >=2.4,<3 since 2.4 has wheels for all platform types. (#154)
- Fix errors in EIP-712 signing (#175)

1.3.6 eth-account v0.6.1 (2022-02-24)

Bugfixes

- Allow encoding of structured data containing `bytes` (#91)

Miscellaneous changes

- #68, #144

1.3.7 eth-account v0.6.0 (2022-01-20)

Features

- Update dependencies: - eth-abi - eth-keyfile - eth-keys - eth-rlp - pyrlp - eth-utils (#138)
- Add support for Python 3.9 and 3.10 (#139)

Deprecations and Removals

- Drop support for Python 3.6 (#139)

1.3.8 eth-account v0.5.9 (2022-08-04)

Bugfixes

- fix DoS-able regex pattern (#178)

Miscellaneous changes

- #183, #184

1.3.9 eth-account v0.5.8 (2022-06-06)

Miscellaneous changes

- #163, #168

1.3.10 eth-account v0.5.7 (2022-01-27)

Features

- Add support for Python 3.9 and 3.10 (#139)

Bugfixes

- `recover_message` now raises an `eth_keys.exceptions.BadSignature` error if the v, r, and s points are invalid (#142)

1.3.11 eth-account v0.5.6 (2021-09-22)

Features

- An explicit transaction type is no longer required for signing a transaction if we can implicitly determine the transaction type from the transaction parameters (#125)

Bugfixes

- When signing a transaction, the regular JSON-RPC structure is now expected as input and is converted to the appropriate rlp transaction structure when signing (#125)
- Fix string interpolation in `ValidationError` message of _hash_eip_191_message (#128)

Improved Documentation

- Updated docs for sign_transaction to show that transaction type can be implicitly determined based on transaction parameters if one is not provided (#126)
- Add `encode_defunct` to list of example message encoders (#127)

1.3.12 eth-account v0.5.5 (2021-07-21)

Features

- Added support for EIP-2718 (Typed Transaction) and EIP-2939 (Access List Transaction) (#115)
- Added support for EIP-1559 (Dynamic Fee Transaction) (#117)

Bugfixes

- Structured messages (EIP-712) new permit leaving some (but not all) domain fields undefined. (#72)

Internal Changes - for eth-account Contributors

- Upgrade project template, of note: a new mypy & pydocstyle, and types being exported correctly. (#121)

Miscellaneous changes

- #116

1.3.13 v0.5.3 (2020-08-31)

Performance improvements

- RLP encoding/decoding speedup by using rlp v2alpha1, which has a rust implementation. (#104)

1.3.14 v0.5.2 (2020-04-30)

Bugfixes

- Makes sure that the raw txt files needed for Mnemonics get packaged with the release. (#99)

1.3.15 v0.5.1

Released 2020-04-23

- Fix a crash in signing typed messages with arrays #97
- Replace attrdict with NamedTuple to silence a deprecation warning #76
- Run more doctests & improve docs #94

1.3.16 v0.5.0

Released 2020-03-30

- Add Python 3.8 support #86
- Add opt-in support for Mnemonic seed phrases #87 (NOTE: This API is unaudited and likely to change)
- Dependency change: support eth-keys v0.3.* #69

1.3.17 v0.4.0

Released 2019-05-06

- BREAKING CHANGE: drop python 3.5 (and therefore pypy3 support). #60 (includes other housekeeping)
- New message signing API: *sign_message()* and `recover_message`. #61
  - New *eth_account.messages.encode_intended_validator()* for EIP-191's Intended Validator message-signing format. #56
  - New *eth_account.messages.encode_structured_data()* for EIP-712's Structured Data message-signing format. #57
- Add optional param iterations to *encrypt()* #52
- Add optional param kdf to *encrypt()*, plus env var `ETH_ACCOUNT_KDF`. Default kdf switched from hmac-sha256 to scrypt. #38
- Accept "to" addresses formatted as `bytes` in addition to checksummed, hex-encoded. #36

1.3.18 v0.3.0

Released July 24, 2018

- Support `eth_keys.datatypes.PrivateKey` in params that accept a private key.
- New docs for *Signers*
- Under the hood: add a new *BaseAccount* abstract class, so that upcoming signing classes can implement it (be on the lookout for upcoming hardware wallet support)

1.3.19 v0.2.3

Released May 27, 2018

- Implement __eq__ and __hash__ for *LocalAccount*, so that accounts can be used in `set`, or as keys in `dict`, etc.

1.3.20 v0.2.2

Released Apr 25, 2018

- Compatibility with pyrlp v0 and v1

1.3.21 v0.2.1

Released Apr 23, 2018

- Accept 'from' in signTransaction, if it matches the sending private key's address

1.3.22 v0.2.0 (stable)

Released Apr 19, 2018

- Audit cleanup is complete
- Stopped requiring chainId, until tooling to automatically derive it gets better (Not that transactions without chainId are potentially replayable on fork chains)

1.3.23 v0.2.0-alpha.0

Released Apr 6, 2018

- Ability to sign an already-hashed message
- Moved `eth_sign`-style message hashing to *eth_account.messages.defunct_hash_message()*
- Stricter transaction input validation, and better error messages. Including: *to* field must be checksummed.
- PyPy3 support & tests
- Upgrade dependencies
- Moved non-public interfaces to *internal* module
- Documentation
  - use `getpass` instead of typing in password manually
  - *eth_account.signers.local.LocalAccount* attributes
  - readme improvements
  - more

1.3.24 v0.1.0-alpha.2

- Imported the local signing code from web3.py's `w3.eth.account`
- Imported documentation and added more
- Imported tests and pass them

1.3.25 v0.1.0-alpha.1

- Launched repository, claimed names for pip, RTD, github, etc

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

e

INDEX

S

V