aiobitcoin Documentation

Release latest

mkbeh

Jul 11, 2021

Contents

Bitcoin is a cryptocurrency, a form of electronic cash. It is a decentralized digital currency without a central bank or single administrator that can be sent from user to user on the peer-to-peer bitcoin network without the need for intermediaries.

Transactions are verified by network nodes through cryptography and recorded in a public distributed ledger called a blockchain. Bitcoin was invented by an unknown person or group of people using the name, Satoshi Nakamoto, and released as open-source software in 2009. Bitcoins are created as a reward for a process known as mining. They can be exchanged for other currencies, products, and services. Research produced by University of Cambridge estimates that in 2017, there were 2.9 to 5.8 million unique users using a cryptocurrency wallet, most of them using bitcoin.

Bitcoin has been criticized for its use in illegal transactions, its high electricity consumption, price volatility, thefts from exchanges, and the possibility that bitcoin is an economic bubble. Bitcoin has also been used as an investment, although several regulatory agencies have issued investor alerts about bitcoin.

CHAPTER 1

About this Library

This is a simple library that provides methods for working with Bitcoin daemon JSON-RPC.

CHAPTER 2

General

2.1 Installation

Linux requirements:

```
sudo apt install python-dev python3-dev
sudo apt install libssl-dev
```

Install with pip:

```
pip3 install aiobitcoin
```

Install manually:

```
git clone https://github.com/mkbeh/aiobitcoin
cd aiobitcoin
python3 setup.py install install --user
```

Upgrade

```
pip3 install aiobitcoin --user --upgrade
```

2.2 Quickstart

Create 'Blockchain' object and get some info:

```
import asyncio
from aiobitcoin.blockchain import Blockchain


async def foo():
```

(continues on next page)

(continued from previous page)

```
    blockchain = Blockchain(url='http://alice:bob@127.0.0.1:18332')
    difficulty = await blockchain.get_difficulty()
    block_count = await blockchain.get_block_count()
    print(difficulty)
    print(block_count)
    await blockchain.close_session()

ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(foo())
```

or use the same with context manager:

```
import asyncio
from aiobitcoin.blockchain import Blockchain


async def foo():
    async with Blockchain(url='http://alice:bob@127.0.0.1:18332') as blockchain:
        difficulty = await blockchain.get_difficulty()
        block_count = await blockchain.get_block_count()
        print(difficulty)
        print(block_count)


ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(foo())
```

Working with bip32:

```
from aiobitcoin.tools import bip32
from aiobitcoin.tools.bip32 import MAINNET_PRV, TESTNET_PRV
from aiobitcoin.tools.key.Key import Key
from aiobitcoin.mnemonic import Mnemonic

# -- Generate mnemonic phrase --
ceed = Mnemonic().generate(encoding=False)

# ... Output: rebel swear tomorrow burger cave giraffe ...


# -- Generate master keys from ceed for BTC mainnet and testnet --
testnet_mxpriv = bip32.xmprv_from_seed(ceed, TESTNET_PRV)
# ... Output: tprv8ZgxMBicQKsPe6tqMpq6qyzFoFSr3cgh...

mainnet_mxpriv = bip32.xmprv_from_seed(ceed, MAINNET_PRV)
# ... Output: xprv9s21ZrQH143K4Q9MazKYy5Kuck31yFeT...

# -- Generate master public keys from master private key --
testnet_mxpub = bip32.xpub_from_xprv(testnet_mxpriv)
mainnet_mxpub = bip32.xpub_from_xprv(mainnet_mxpriv)

# ... Output: tpubD6NzVbkrYhZ4X5ghC8mzzsGuMQCxEmnh5Y...
# ... Output: xpub661MyMwAqRbcFHVqjwnunwwY2H7JFPHdXv...

# -- Transform master private key to WIF format and getting address of master key --
key = Key.from_text(mainnet_mxpriv)
```

(continues on next page)

(continued from previous page)

```
wif = key.wif()
# ... Output: L4PEssMfRgHvmpyEGxHJkFVcNWeQvZiySNMAa...

addr = key.address()
# ... Output: 1BGLari4SCxGXoJib27C8pAL6Ef3pFqswD

# -- Create sub key by custom derive path --
subkey = key.subkey_for_path('1/0/{}'.format(11))

addr = subkey.address(use_uncompressed=False)
wif = subkey.wif()
```

2.3 Examples

2.3.1 JSON-RPC interaction

Basic usage:

```
import asyncio
from aiobitcoin.blockchain import Blockchain


async def foo():
    blockchain = Blockchain(url='http://alice:bob@127.0.0.1:18332')
    difficulty = await blockchain.get_difficulty()
    block_count = await blockchain.get_block_count()
    print(difficulty)
    print(block_count)
    await blockchain.close_session()

ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(foo())
```

Basic usage with context manager:

```
import asyncio
from aiobitcoin.blockchain import Blockchain


async def foo():
    async with Blockchain(url='http://alice:bob@127.0.0.1:18332') as blockchain:
        difficulty = await blockchain.get_difficulty()
        block_count = await blockchain.get_block_count()
        print(difficulty)
        print(block_count)


ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(foo())
```

Another way to usage:

```
import asyncio
from aiobitcoin.grambitcoin import GramBitcoin
from aiobitcoin.blockchain import Blockchain


async def baz():
    # Create gram objects.
    gram = GramBitcoin(session_required=True)

    # Get some single info.
    blockchain = Blockchain(url='http://alice:bob@127.0.0.1:18332', gram=gram)
    result = await blockchain.get_block_count()
    print(result)

    # Close sessions.
    await gram.close_session()
```

How to call methods asynchronously:

```
async def multi(obj):
    result = await obj.get_block_count()
    print(result)


async def baz():
    # Create Network objects with sessions.
    objs = [Blockchain(url='http://alice:bob@127.0.0.1:18332') for _ in range(100)]

    # Call methods asynchronously
    await asyncio.gather(
        *(multi(obj) for obj in objs)
    )

    # Close sessions
    [await obj.close_session() for obj in objs]
```

Another example of how to call methods asynchronously:

This method is less productive than the previous one by about 25%, but more elegant :)

```
import asyncio
from aiobitcoin.blockchain import Blockchain


async def multi(obj):
    result = await obj.get_block_count()
    print(result)


async def baz():
    async with Blockchain(url='http://alice:bob@127.0.0.1:18332') as blockchain:
        await asyncio.gather(
            *(multi(blockchain) for _ in range(100))
        )
```

Another way to get some info:

```
import asyncio
from aiobitcoin.grambitcoin import GramBitcoin
from aiobitcoin.blockchain import Blockchain


async def baz():
    # Create gram object with `session_required=True`.
    gram = GramBitcoin(url='http://alice:bob@127.0.0.1:18332', session_required=True)

    # Pass the `gram` object to the `Blockchain` class constructor.
    blockchain = Blockchain(gram=gram)

    # Get info.
    result = await blockchain.get_block_count()
    print(result)

    # Close session.
    await gram.close_session()
```

How convenient to get various information using the 'GramBitcoin':

```
import asyncio
from aiobitcoin.grambitcoin import GramBitcoin
from aiobitcoin.blockchain import Blockchain
from aiobitcoin.network import Network


async def baz():
    # Create gram object with `session_required=True`.
    gram = GramBitcoin(url='http://alice:bob@127.0.0.1:18332', session_required=True)

    # Pass the `gram` object to the `Blockchain` class constructor.
    blockchain = Blockchain(gram=gram)
    network = Network(gram=gram)

    # Get info.
    result = await blockchain.get_block_count()
    print(result)

    # Get another info.
    another_result = await network.get_peer_info(to_list=True)
    print(another_result)

    # Close session.
    await gram.close_session()
```

Get single data and then get multi data asynchronously using 'GramBitcoin'

```
import asyncio
from aiobitcoin.grambitcoin import GramBitcoin
from aiobitcoin.blockchain import Blockchain
from aiobitcoin.network import Network
from aiobitcoin.bitcoinerrors import NoConnectionToTheDaemon


async def multi(obj):
    result = await obj.get_peer_info()
```

(continues on next page)

(continued from previous page)

```
    print(result)


async def baz():
    # Create grams objects.
    grams = [GramBitcoin(url='http://alice:bob@127.0.0.1:18332', session_
↪required=True)
             for _ in range(10)]

    # Try to get some single info.
    try:
        blockchain = Blockchain(gram=grams[0])
        result = await blockchain.get_block_count()
        print(result)
    except NoConnectionToTheDaemon:
        pass

    # Get another info asynchronously.
    objs = [Network(gram=gram) for gram in grams]
    await asyncio.gather(
        *(multi(obj) for obj in objs)
    )

    # Close sessions.
    [await gram.close_session() for gram in grams]
```

2.3.2 Mnemonic phrase generation

```
from aiobitcoin.mnemonic import Mnemonic
ceed = Mnemonic().generate(encoding=False)

# ... Output: rebel swear tomorrow burger cave giraffe ...
```

2.3.3 bip32

Getting master private key from mnemonic phrase:

```
from aiobitcoin.tools import bip32
from aiobitcoin.tools.bip32 import MAINNET_PRV, TESTNET_PRV


testnet_mxpriv = bip32.xmprv_from_seed(ceed, TESTNET_PRV)
# ... Output: tprv8ZgxMBicQKsPe6tqMpq6qyzFoFSr3cgh...

mainnet_mxpriv = bip32.xmprv_from_seed(ceed, MAINNET_PRV)
# ... Output: xprv9s21ZrQH143K4Q9MazKYy5Kuck31yFeT...
```

Getting master public key from master private key:

```
from aiobitcoin.tools import bip32

testnet_mxpub = bip32.xpub_from_xprv(testnet_mxpriv)
mainnet_mxpub = bip32.xpub_from_xprv(mainnet_mxpriv)
```

(continues on next page)

(continued from previous page)

```
# ... Output: tpubD6NzVbkrYhZ4X5ghC8mzzsGuMQCxEmnh5Y...
# ... Output: xpub661MyMwAqRbcFHVqjwnunwwY2H7JFPHdXv...
```

2.3.4 Key tool interaction

Key tool basic usage:

```
from aiobitcoin.tools.key.Key import Key

key = Key.from_text(mainnet_mxpriv)

wif = key.wif()
# ... Output: L4PEssMfRgHvmpyEGxHJkFVcNWeQvZiySNMAa...

addr = key.address()
# ... Output: 1BGLari4SCxGXoJib27C8pAL6Ef3pFqswD

child_index = key.child_index()
# ... Output: 0

mxpub = key.hwif()
# ... Output: xpub661MyMwAqRbcFi4Mh1uhDohwNygiiNuf2C...

hex_mpriv = key.sec_as_hex()
# ... Output: 02d823155a8336b2eb3bfc5536199aec11993e...

sec_mpriv = key.sec()
# ... Output: b'\x02\xd8#\x15Z\x836\xb2\xeb;\xfcU6\x...'

tree_depth = key.tree_depth()
# ... Output: 0
```

Creating sub keys by custom derivation path:

```
subkey = key.subkey_for_path('1/0/{}'.format(11))

addr = subkey.address(use_uncompressed=False)
wif = subkey.wif()
child_index = subkey.child_index()
tree_depth = subkey.tree_depth()

# ... addr: 1KgUQ9GFrQRh2fLX2WfXPdipKsTSDyZeqr
# ... wif: KzRLKBHTNo15FFnQNE4d5iniK85EgDqBaaM4FURme5LmMiYk7nhz
# ... child_index: 11
# ... tree_depth: 3
```

Note: The addresses and WIF keys obtained by the above methods can be easily imported into the Bitcoin Core.

CHAPTER 3

Package

3.1 API

3.1.1 aiobitcoin.tools

aiobitcoin.tools.key package

aiobitcoin.tools.key.BIP32Node module

A BIP0032-style hierarchical wallet.

Implement a BIP0032-style hierarchical wallet which can create public or private wallet keys. Each key can create many child nodes. Each node has a wallet key and a corresponding private & public key, which can be used to generate Bitcoin addresses or WIF private keys.

At any stage, the private information can be stripped away, after which descendants can only produce public keys.

Private keys can also generate "hardened" children, which cannot be generated by the corresponding public keys. This is useful for generating "change" addresses, for example, which there is no need to share with people you give public keys to.

class aiobitcoin.tools.key.BIP32Node.**BIP32Node**(*netcode*, *chain_code*, *depth=0*, *parent_fingerprint=b'x00x00x00x00'*, *child_index=0*, *secret_exponent=None*, *public_pair=None*)

Bases: *aiobitcoin.tools.key.Key.Key*

This is a deterministic wallet that complies with BIP0032 https://en.bitcoin.it/wiki/BIP_0032

as_text(*as_private=False*)

Yield a 111-byte string corresponding to this node.

chain_code()

child_index()

children(*max_level=50*, *start_index=0*, *include_hardened=True*)

fingerprint()

classmethod from_hwif(*b58_str*, *allow_subkey_suffix=True*)
Generate a Wallet from a base58 string in a standard way.

classmethod from_master_secret(*master_secret*, *netcode='BTC'*)
Generate a Wallet from a master password.

classmethod from_wallet_key(*b58_str*, *allow_subkey_suffix=True*)
Generate a Wallet from a base58 string in a standard way.

hwif(*as_private=False*)
Yield a 111-byte string corresponding to this node.

parent_fingerprint()

public_copy()
Yield the corresponding public node for this node.

serialize(*as_private=None*)
Yield a 78-byte binary blob corresponding to this node.

subkey(*i=0*, *is_hardened=False*, *as_private=None*)
Yield a child node for this node.

i: the index for this node. is_hardened: use "hardened key derivation". That is, the public version

of this node cannot calculate this child.

as_private: set to True to get a private subkey.

subkey_for_path(*path*)

path: a path of subkeys denoted by numbers and slashes. Use H or p for private key derivation. End with .pub to force the key public.

Examples: 1H/5/2/1 would call subkey(i=1, is_hardened=True) .subkey(i=5).subkey(i=2).subkey(i=1) and then yield the private key 0/0/458.pub would call subkey(i=0).subkey(i=0) .subkey(i=458) and then yield the public key

You should choose one of the H or p convention for private key derivation and stick with it.

subkeys(*path*)
A generalized form that can return multiple subkeys.

tree_depth()

wallet_key(*as_private=False*)
Yield a 111-byte string corresponding to this node.

exception aiobitcoin.tools.key.BIP32Node.**PublicPrivateMismatchError**
Bases: Exception

aiobitcoin.tools.key.Key module

exception aiobitcoin.tools.key.Key.**InvalidPublicPairError**
Bases: ValueError

exception aiobitcoin.tools.key.Key.**InvalidSecretExponentError**
Bases: ValueError

class aiobitcoin.tools.key.Key.**Key**(*secret_exponent=None*, *public_pair=None*, *hash160=None*, *prefer_uncompressed=None*, *is_compressed=None*, *is_pay_to_script=False*, *netcode=None*)

Bases: object

address(*use_uncompressed=None*)
Return the public address representation of this key, if available. If use_uncompressed is not set, the preferred representation is returned.

as_text()
Return a textual representation of this key.

bitcoin_address(*use_uncompressed=None*)
Return the public address representation of this key, if available. If use_uncompressed is not set, the preferred representation is returned.

classmethod from_sec(*sec*, *netcode=None*)
Create a key from an sec bytestream (which is an encoding of a public pair).

classmethod from_text(*text*, *is_compressed=False*)
This function will accept a BIP0032 wallet string, a WIF, or a bitcoin address.

The "is_compressed" parameter is ignored unless a public address is passed in.

hash160(*use_uncompressed=None*)
Return the hash160 representation of this key, if available. If use_uncompressed is not set, the preferred representation is returned.

is_private()

netcode()
Return the netcode

public_copy()

public_pair()
Return a pair of integers representing the public key (or None).

sec(*use_uncompressed=None*)
Return the SEC representation of this key, if available. If use_uncompressed is not set, the preferred representation is returned.

sec_as_hex(*use_uncompressed=None*)
Return the SEC representation of this key as hex text. If use_uncompressed is not set, the preferred representation is returned.

secret_exponent()
Return an integer representing the secret exponent (or None).

sign(*h*)
Return a der-encoded signature for a hash h. Will throw a RuntimeError if this key is not a private key

subkey(*path_to_subkey*)
Return the Key corresponding to the hierarchical wallet's subkey

subkeys(*path_to_subkeys*)
Return an iterator yielding Keys corresponding to the hierarchical wallet's subkey path (or just this key).

verify(*h*, *sig*)
Return whether a signature is valid for hash h using this key.

wif(*use_uncompressed=None*)
Return the WIF representation of this key, if available. If use_uncompressed is not set, the preferred representation is returned.

aiobitcoin.tools.bip32 module

BIP32 Hierarchical Deterministic Wallet functions.

A deterministic wallet is a hash-chain of private/public key pairs that derives from a single root, which is the only element requiring backup. Moreover, there are schemes where public keys can be calculated without accessing private keys.

A hierarchical deterministic wallet is a tree of multiple hash-chains, derived from a single root, allowing for selective sharing of keypair chains.

Here, the HD wallet is implemented according to BIP32 bitcoin standard https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki.

aiobitcoin.tools.bip32.**xmprv_from_seed**(*seed: Union[str, bytes], version: Union[str, bytes], decode: bool = True*) → bytes
derive the master extended private key from the seed

aiobitcoin.tools.bip32.**xpub_from_xprv**(*xprv: Union[str, bytes], decode: bool = True*) → bytes
Neutered Derivation (ND)

Computation of the extended public key corresponding to an extended private key ("neutered" as it removes the ability to sign transactions)

3.1.2 aiobitcoin.mnemonic module

class aiobitcoin.mnemonic.**Mnemonic**(*language='english'*)
Bases: object

static checksum(*data*)
Calculates checksum for given data key

Parameters **data** (*bytes, hexstring*) – key string

Return str Checksum of key in bits

generate(*strength=128*, *add_checksum=True*, *encoding=True*)
Generate a random Mnemonic key

Uses cryptographically secure os.urandom() function to generate data. Then creates a Mnemonic sentence with the 'to_mnemonic' method. :param strength: Key strength in number of bits, default is 128 bits. It advised to specify 128 bits or more, i.e.: 128, 256, 512 or 1024 :type strength: int :param add_checksum: Included a checksum? Default is True :type add_checksum: bool :param encoding :type encoding: bool

Return str Mnemonic passphrase consisting of a space seperated list of words

to_mnemonic(*data*, *add_checksum=True*, *check_on_curve=True*)
Convert key data entropy to Mnemonic sentence

Parameters

- **data** (*bytes, hexstring*) – Key data entropy
- **add_checksum** (*bool*) – Included a checksum? Default is True

- **check_on_curve** – Check if data integer value is on secp256k1 curve. Should be enabled when not

testing and working with crypto :type check_on_curve: bool

Return str Mnemonic passphrase consisting of a space seperated list of words

3.1.3 aiobitcoin.grambitcoin module

class aiobitcoin.grambitcoin.**GramBitcoin**(*url=None*, *read_timeout=20*, *session_required=False*)

Bases: object

This class is needed to create a session and then transfer the class instance with already active session to other classes , in order to avoid creating multiple sessions. RPC-API https://bitcoincore.org/en/doc/0.17.0/

Parameters

- **url (optional)** (*str*) – Node URI in format http://alice:bob@127.0.0.1:18332
- **read_timeout (20)** (*int*) – Request operations timeout
- **session_required (optional)** (*object*) – Will create session or not

Important note: if you create an instance of this class, then you need to pass the session_required parameter with the value True. It is not necessary to pass the url parameter, if you do not pass it, then do not forget to pass the url to other classes to which you intend to pass an object of this class.

call_method(*method*, **args*)

check_gram(*gram*)

static check_url(*url*, *gram*)

close_session()

3.1.4 aiobitcoin.blockchain module

class aiobitcoin.blockchain.**Blockchain**(*url=None*, *gram=None*, *read_timeout=20*)

Bases: aiobitcoin.grambitcoincommon.GramBitcoinCommon

Methods from *Blockchain* section https://bitcoincore.org/en/doc/0.17.0/.

Parameters

- **url (optional)** (*str*) – Node URI in format http://alice:bob@127.0.0.1:18332
- **gram (optional)** (*object*) – GramBitcoin object
- **read_timeout (20)** (*int*) – Request operations timeout

Note: You must pass at least one parameter or *url* or *gram* (with active session).

get_block_count() → int

Returns Returns the number of blocks in the longest blockchain.

get_blockchain_info() → dict

Provides information about the current state of the block chain. :return: full blockchain information.

get_difficulty()

Returns Returns the proof-of-work difficulty as a multiple of the minimum difficulty.

get_mempool_info()

Returns Returns details on the active state of the TX memory pool.

3.1.5 aiobitcoin.network module

class aiobitcoin.network.**Network**(*url=None*, *gram=None*, *read_timeout=20*)
Bases: aiobitcoin.grambitcoincommon.GramBitcoinCommon

Methods from *Network* section https://bitcoincore.org/en/doc/0.17.0/.

Parameters

- **url (optional)** (*str*) – Node URI in format http://alice:bob@127.0.0.1:18332
- **gram (optional)** (*object*) – GramBitcoin object
- **read_timeout (20)** (*int*) – Request operations timeout

Note: You must pass at least one parameter or *url* or *gram* (with active session).

clear_banned() → None
Clear all banned IPs. :return: None

get_network_info() → dict

Returns Returns an object containing various state info regarding P2P networking.

get_peer_info(*to_list: bool = True*) → list

Parameters **to_list** – will return list or genexpr

Returns Returns data about each connected network node as a json array of objects.

list_banned(*to_list: bool = True*) → list

Parameters **to_list** –

Returns List all banned IPs/Subnets.

ping() → None
Requests that a ping be sent to all other nodes, to measure ping time. Results provided in getpeerinfo, pingtime and pingwait fields are decimal seconds. Ping command is handled in queue with all other commands, so it measures processing backlog, not just network ping. :return: None

set_ban(*subnet: str*, *command: str = 'add'*, *bantime: int = 0*, *absolute: bool = False*) → aiobitcoin.bitcoinerrors.InvalidIpOrSubnet
Attempts to add or remove an IP/Subnet from the banned list. :param subnet: The IP/Subnet (see getpeerinfo for nodes IP) with an optional netmask (default is /32 = single IP) :param command: 'add' to add an IP/Subnet to the list, 'remove' to remove an IP/Subnet from the list :param bantime: Time in seconds how long (or until when if [absolute] is set) the IP is banned (0 or empty means using the default time of 24h which can also be overwritten by the -bantime startup argument) :param absolute: If set, the bantime must be an absolute timestamp in seconds since epoch (Jan 1 1970 GMT) :return: None

3.1.6 aiobitcoin.util module

class aiobitcoin.util.**Util**(*url=None*, *gram=None*, *read_timeout=20*)
Bases: aiobitcoin.grambitcoincommon.GramBitcoinCommon

Methods from *Util* section https://bitcoincore.org/en/doc/0.17.0/.

Parameters

- **url (optional)** (*str*) – Node URI in format http://alice:bob@127.0.0.1:18332
- **gram (optional)** (*object*) – GramBitcoin object
- **read_timeout (20)** (*int*) – Request operations timeout

Note: You must pass at least one parameter or *url* or *gram* (with active session).

validate_address(*addr: str*) → bool
Return information about the given bitcoin address. :param addr: bitcoin address :return: boolean value

3.1.7 aiobitcoin.wallet module

class aiobitcoin.wallet.**Wallet**(*url=None*, *gram=None*, *read_timeout=20*)
Bases: aiobitcoin.grambitcoincommon.GramBitcoinCommon

Methods from *Wallet* section https://bitcoincore.org/en/doc/0.17.0/.

Parameters

- **url (optional)** (*str*) – Node URI in format http://alice:bob@127.0.0.1:18332
- **gram (optional)** (*object*) – GramBitcoin object
- **read_timeout (20)** (*int*) – Request operations timeout

Note: You must pass at least one parameter or *url* or *gram* (with active session).

get_balance() → float
Returns the total available balance. :return: total balance

import_address(*addr: str*, *rescan: bool = False*) → aiobitcoin.bitcoinerrors.PrivateKeyForThisAddressAlreadyInWallet
Adds an address or script (in hex) that can be watched as if it were in your wallet but cannot be used to spend. :param addr: bitcoin address :param rescan: activate rescanning blockchain after importing :return: boolean value

import_priv_key(*wif: str*) → aiobitcoin.bitcoinerrors.InvalidPrivateKeyEncoding
Adds a private key (as returned by dumpprivkey) to your wallet. :param wif: bitcoin private key :return: boolean value

list_transactions(*count: int = 100*, *include_watchonly: bool = True*, *to_list: bool = True*) → list
Returns up to 'count' most recent transactions. :param count: The number of transactions to return :param include_watchonly: Include transactions to watch-only addresses :param to_list: will return list or genexpr :return: list of transactions

rescan_blockchain(*days_ago: int*, *full_rescan: bool = False*) → bool
Rescan the local blockchain for wallet related transactions. :param days_ago: how many days the blockchain rescanning will pass :param full_rescan: blockchain rescanning will start from block 1 :return: boolean value

send_to_address(*addr: str*, *amount: int*) → aiobitcoin.bitcoinerrors.InvalidAddress
Send an amount to a given address. :param addr: Bitcoin address :param amount: amount to send :return: tx hash

3.1.8 aiobitcoin.bitcoinerrors module

exception aiobitcoin.bitcoinerrors.**IncorrectCreds**(*uri*)
Bases: aiobitcoin.bitcoinerrors._BitcoinErrors

Login or password in URI is incorrect.

exception aiobitcoin.bitcoinerrors.**InvalidAddress**(*error_msg=''*)
Bases: aiobitcoin.bitcoinerrors._BitcoinErrors

Invalid address.

exception aiobitcoin.bitcoinerrors.**InvalidIpOrSubnet**(*error_msg=''*)
Bases: aiobitcoin.bitcoinerrors._BitcoinErrors

Invalid ip or subnet.

exception aiobitcoin.bitcoinerrors.**InvalidPrivateKeyEncoding**(*error_msg=''*)
Bases: aiobitcoin.bitcoinerrors._BitcoinErrors

Invalid private key encoding.

exception aiobitcoin.bitcoinerrors.**NoConnectionToTheDaemon**(*error_msg*)
Bases: aiobitcoin.bitcoinerrors._BitcoinErrors

There is no connection to the daemon.

exception aiobitcoin.bitcoinerrors.**PrivateKeyForThisAddressAlreadyInWallet**(*error_msg=''*)
Bases: aiobitcoin.bitcoinerrors._BitcoinErrors

Private key for address is already in wallet.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index