# Bitcoinlib Documentation

***Release 0.4.5***

**Lennart Jongeneel (mccwdev)**

**Mar 04, 2019**

# Manuals

Bitcoin and other Cryptocurrencies Library for Python. Includes a fully functional wallet, Mnemonic key generation and management and connection with various service providers to receive and send blockchain and transaction information.

CHAPTER 1

# Wallet

The bitcoinlibrary contains a wallet implementation using sqlalchemy and sqllite3 to import, create and manage keys in a Hierarchical Deterministic Way.

Example: Create wallet and generate new key to receive bitcoins

```
>>> from bitcoinlib.wallets import HDWallet
>>> w = HDWallet.create('Wallet1')
>>> w
<HDWallet (id=1, name=Wallet1, network=bitcoin)>
>>> key1 = w.new_key()
>>> key1
<HDWalletKey (name=Key 0, wif=xprvA4B..etc..6HZKGW7Kozc, path=m/44'/0'/0'/0/0)>
>>> key1.address
'1Fo7STj6LdRhUuD1AiEsHpH65pXzraGJ9j'
```

When your wallet received a payment and has unspent transaction outputs, you can send bitcoins easily. If succesfull a transaction ID is returned

```
>>> w.send_to('12ooWd8Xag7hsgP9PBPnmyGe36VeUrpMSH', 100000)
'b7feea5e7c79d4f6f343b5ca28fa2a1fcacfe9a2b7f44f3d2fd8d6c2d82c4078'
```

# CHAPTER 2

## Segregated Witness Wallet

Easily create and manage segwit wallets. Both native segwit with base32/bech32 addresses and P2SH nested segwit wallets with traditional addresses are available.

Create a native single key P2WPKH wallet:

```
>>> from bitcoinlib.wallets import HDWallet
>>> w = HDWallet.create('segwit_p2wpkh', witness_type='segwit')
>>> w.get_key().address
bc1q84y2quplejutvu0h4gw9hy59fppu3thg0u2xz3
```

Or create a P2SH nested single key P2SH_P2WPKH wallet:

```
>>> from bitcoinlib.wallets import HDWallet
>>> w = HDWallet.create('segwit_p2sh_p2wpkh', witness_type='p2sh-segwit')
>>> w.get_key().address
36ESSWgR4WxXJSc4ysDSJvecyY6FJkhUbp
```

CHAPTER 3

# Wallet from passphrase with accounts and multiple currencies

The following code creates a wallet with two bitcoin and one litecoin account from a Mnemonic passphrase. The complete wallet can be recovered from the passphrase which is the masterkey.

```
from bitcoinlib.wallets import HDWallet, wallet_delete
from bitcoinlib.mnemonic import Mnemonic

passphrase = Mnemonic().generate()
print(passphrase)
w = HDWallet.create("Wallet2", keys=passphrase, network='bitcoin')
account_btc2 = w.new_account('Account BTC 2')
account_ltc1 = w.new_account('Account LTC', network='litecoin')
w.get_key()
w.get_key(account_btc2.account_id)
w.get_key(account_ltc1.account_id)
w.info()
```

CHAPTER 4

# Multi Signature Wallets

Create a Multisig wallet with 2 cosigner which both need to sign a transaction.

```
from bitcoinlib.wallets import HDWallet
from bitcoinlib.keys import HDKey

NETWORK = 'testnet'
k1 = HDKey(
↪'tprv8ZgxMBicQKsPd1Q44tfDiZC98iYouKRC2CzjT3HGt1yYw2zuX2awTotzGAZQEAU9bi2M5MCj8iedP9MREPjUgpDEBwBgG
↪'
           '5zNYeiX8', network=NETWORK)
k2 = HDKey(
↪'tprv8ZgxMBicQKsPeUbMS6kswJc11zgVEXUnUZuGo3bF6bBrAg1ieFfUdPc9UHqbD5HcXizThrcKike1c4z6xHrz6MWGwy8L6
↪'
           'MeQHdWDp', network=NETWORK)
w1 = HDWallet.create('multisig_2of2_cosigner1', sigs_required=2,
                     keys=[k1, k2.public_master(multisig=True)], network=NETWORK)
w2 = HDWallet.create('multisig_2of2_cosigner2',  sigs_required=2,
                     keys=[k1.public_master(multisig=True), k2], network=NETWORK)
print("Deposit testnet bitcoin to this address to create transaction: ", w1.get_key().
↪address)
```

Create a transaction in the first wallet

```
w1.utxos_update()
t = w1.sweep('mwCwTceJvYV27KXBc3NJZys6CjsgsoeHmf', min_confirms=0)
t.info()
```

And then import the transaction in the second wallet, sign it and push it to the network

```
w2.get_key()
t2 = w2.transaction_import(t)
t2.sign()
t2.send()
t2.info()
```

# CHAPTER 5

## Command Line Tool

With the command line tool you can create and manage wallet without any Python programming.

To create a new Bitcoin wallet

```
$ cli-wallet NewWallet
Command Line Wallet for BitcoinLib

Wallet newwallet does not exist, create new wallet [yN]? y

CREATE wallet 'newwallet' (bitcoin network)

Your mnemonic private key sentence is: force humble chair kiss season ready elbow␣
↪cool awake divorce famous tunnel

Please write down on paper and backup. With this key you can restore your wallet and␣
↪all keys
```

You can use 'cli-wallet' to create simple or multisig wallets for various networks, manage public and private keys and managing transactions.

For the full command line wallet documentation please read

http://bitcoinlib.readthedocs.io/en/latest/_static/manuals.command-line-wallet.html

CHAPTER 6

# Service providers

Communicates with pools of bitcoin service providers to retreive transaction, address, blockchain information. To push a transaction to the network. To determine optimal service fee for a transaction. Or to update your wallet's balance.

Example: Get estimated transactionfee in sathosis per Kb for confirmation within 5 blocks

```
>>> from bitcoinlib.services.services import Service
>>> Service().estimatefee(5)
138964
```

CHAPTER 7

# More examples

For more examples see https://github.com/1200wd/bitcoinlib/tree/master/examples

## 7.1 Install, Update and Tweak BitcoinLib

### 7.1.1 Installation

#### Install with pip

```
$ pip install bitcoinlib
```

Package can be found at https://pypi.python.org/pypi/bitcoinlib/

#### Install from source

```
$ git clone https://github.com/1200wd/bitcoinlib
$ cd bitcoinlib
$ python setup.py install
```

#### Package dependencies

Required Python Packages, are automatically installed upon installing bitcoinlib:

- ecdsa
- pyaes
- scrypt
- sqlalchemy

- requests
- enum34 (for older python installations)

#### Other requirements Linux

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

To install OpenSSL development package on Debian, Ubuntu or their derivatives

```
sudo apt install libssl-dev
```

To install OpenSSL development package on Fedora, CentOS or RHEL

```
sudo yum install openssl-devel
```

#### Other requirements Windows

Tested on Windows 10 with Python 3.6 and pip installed. No special requirements needed.

### 7.1.2 Update Bitcoinlib

Before you update make sure to backup your database! Also backup your settings files in ./bitcoinlib/config if you have made any changes.

If you installed the library with pip upgrade with

```
$ pip install bitcoinlib --upgrade
```

Otherwise pull the git repository.

After an update it might be necessary to update the config files. The config files will be overwritten with new versions if you delete the .bitcoinlib/logs/install.log file.

```
$ rm .bitcoinlib/logs/install.log
```

If the new release contains database updates you have to migrate the database with the updatedb.py command. This program extracts keys and some wallet information from the old database and then creates a new database. The updatedb.py command is just a helper tool and not guaranteed to work, it might fail if there are a lot of database changes. So backup database / private keys first and use at your own risk!

```
$ python updatedb.py
Wallet and Key data will be copied to new database. Transaction data will NOT be␣
↪copied
Updating database file: /home/guest/.bitcoinlib/database/bitcoinlib.sqlite
Old database will be backed up to /home/guest/.bitcoinlib/database/bitcoinlib.sqlite.
↪backup-20180711-01:46
Type 'y' or 'Y' to continue or any other key to cancel: y
```

### 7.1.3 Troubleshooting

When you experience issues with the scrypt package when installing you can try to solve this by installing scrypt seperately:

pip intall scrypt

Please make sure you also have the Python development and SSL development packages installed, see 'Other requirements' above.

You can also use pyscrypt instead of scrypt. Pyscrypt is a pure Python scrypt password-based key derivation library. It works but it is slow when using BIP38 password protected keys.

    pip install pyscrypt

If you run into issues to not hesitate to contact us or file an issue at https://github.com/1200wd/bitcoinlib/issues

### 7.1.4 Tweak BitcoinLib

You can Add another service Provider to this library by updating settings and write a new service provider class.

If you use this library in a production environment it is advised to run your own Bitcoin, Litecoin or Dash node, both for privacy and reliability reasons. More setup information: Setup connection to bitcoin node

Some service providers require an API key to function or allow additional requests. You can add this key to the provider settings file in .bitcoinlib/config/providers.json

## 7.2 Command Line Wallet

Manage wallets from commandline. Allows you to

- Show wallets and wallet info
- Create single and multi signature wallets
- Delete wallets
- Generate receive addresses
- Create transactions
- Import and export transactions
- Sign transactions with available private keys
- Broadcast transaction to the network

The Command Line wallet Script can be found in the tools directory. If you call the script without arguments it will show all available wallets.

Specify a wallet name or wallet ID to show more information about a wallet. If you specify a wallet which doesn't exists the script will ask you if you want to create a new wallet.

### 7.2.1 Create wallet

To create a wallet just specify an unused wallet name:

```
$ cli-wallet mywallet
Command Line Wallet for BitcoinLib

Wallet mywallet does not exist, create new wallet [yN]? y

CREATE wallet 'mywallet' (bitcoin network)

Your mnemonic private key sentence is: mutual run dynamic armed brown meadow height␣
↪elbow citizen put industry work
```

(continues on next page)

(continued from previous page)

```
Please write down on paper and backup. With this key you can restore your wallet and␣
↪all keys

Type 'yes' if you understood and wrote down your key: yes
Updating wallet
```

### 7.2.2 Generate / show receive addresses

To show an unused address to receive funds use the -r or –receive option. If you want to show QR codes on the commandline install the pyqrcode module.

```
$ cli-wallet mywallet -r
Command Line Wallet for BitcoinLib

Receive address is 1JMKBiiDMdjTx6rfqGumALvcRMX6DQNeG1
```

### 7.2.3 Send funds / create transaction

To send funds use the -t option followed by the address and amount. You can also repeat this to send to multiple addresses.

A manual fee can be entered with the -f / –fee option.

The default behavior is to just show the transaction info and raw transaction. You can push this to the network with a 3rd party. Use the -p / –push option to push the transaction to the network.

```
$ cli-wallet -d dbtest mywallet -t 1FpBBJ2E9w9nqxHUAtQME8X4wGeAKBsKwZ 10000
```

### 7.2.4 Restore wallet with passphrase

To restore or create a wallet with a passphrase use new wallet name and the –passphrase option. If it's an old wallet you can recreate and scan it with the -s option. This will create new addresses and update unspend outputs.

```
$ cli-wallet mywallet --passphrase "mutual run dynamic armed brown meadow height␣
↪elbow citizen put industry work"
$ cli-wallet mywallet -s
```

### 7.2.5 Options Overview

Command Line Wallet for BitcoinLib

```
usage: cli_wallet.py [-h] [--wallet-remove] [--list-wallets] [--wallet-info]
                     [--update-utxos] [--update-transactions]
                     [--wallet-recreate] [--receive [NUMBER_OF_ADDRESSES]]
                     [--generate-key] [--export-private]
                     [--passphrase [PASSPHRASE [PASSPHRASE ...]]]
                     [--passphrase-strength PASSPHRASE_STRENGTH]
                     [--network NETWORK] [--database DATABASE]
                     [--create-from-key KEY]
```

(continues on next page)

(continued from previous page)

```
                      [--create-multisig [NUMBER_OF_SIGNATURES_REQUIRED [KEYS ...]]]
                      [--create-transaction [ADDRESS_1 [AMOUNT_1 ...]]]
                      [--sweep ADDRESS] [--fee FEE] [--fee-per-kb FEE_PER_KB]
                      [--push] [--import-tx TRANSACTION]
                      [--import-tx-file FILENAME_TRANSACTION]
                      [wallet_name]

BitcoinLib CLI

positional arguments:
  wallet_name           Name of wallet to create or open. Used to store your
                        all your wallet keys and will be printed on each paper
                        wallet

optional arguments:
  -h, --help            show this help message and exit

Wallet Actions:
  --wallet-remove       Name or ID of wallet to remove, all keys and
                        transactions will be deleted
  --list-wallets, -l    List all known wallets in BitcoinLib database
  --wallet-info, -w     Show wallet information
  --update-utxos, -x    Update unspent transaction outputs (UTXO's) for this
                        wallet
  --update-transactions, -u
                        Update all transactions and UTXO's for this wallet
  --wallet-recreate, -z
                        Delete all keys and transactions and recreate wallet,
                        except for the masterkey(s). Use when updating fails
                        or other errors occur. Please backup your database and
                        masterkeys first.
  --receive [NUMBER_OF_ADDRESSES], -r [NUMBER_OF_ADDRESSES]
                        Show unused address to receive funds. Generate new
                        payment andchange addresses if no unused addresses are
                        available.
  --generate-key, -k    Generate a new masterkey, and show passphrase, WIF and
                        public account key. Use to create multisig wallet
  --export-private, -e  Export private key for this wallet and exit

Wallet Setup:
  --passphrase [PASSPHRASE [PASSPHRASE ...]]
                        Passphrase to recover or create a wallet. Usually 12
                        or 24 words
  --passphrase-strength PASSPHRASE_STRENGTH
                        Number of bits for passphrase key. Default is 128,
                        lower is not adviced but can be used for testing. Set
                        to 256 bits for more future proof passphrases
  --network NETWORK, -n NETWORK
                        Specify 'bitcoin', 'litecoin', 'testnet' or other
                        supported network
  --database DATABASE, -d DATABASE
                        Name of specific database file to use
  --create-from-key KEY, -c KEY
                        Create a new wallet from specified key
  --create-multisig [NUMBER_OF_SIGNATURES_REQUIRED [KEYS ...]], -m [NUMBER_OF_
↪SIGNATURES_REQUIRED [KEYS ...]]
                        Specificy number of signatures required followed by a
```

(continues on next page)

(continued from previous page)

```
                        list of signatures. Example: -m 2 tprv8ZgxMBicQKsPd1Q4
                        4tfDiZC98iYouKRC2CzjT3HGt1yYw2zuX2awTotzGAZQEAU9bi2M5M
                        Cj8iedP9MREPjUgpDEBwBgGi2C8eK5zNYeiX8 tprv8ZgxMBicQKsP
                        eUbMS6kswJc11zgVEXUnUZuGo3bF6bBrAg1ieFfUdPc9UHqbD5HcXi
                        zThrcKike1c4z6xHrz6MWGwy8L6YKVbgJMeQHdWDp

Transactions:
  --create-transaction [ADDRESS_1 [AMOUNT_1 ...]], -t [ADDRESS_1 [AMOUNT_1 ...]]
                        Create transaction. Specify address followed by
                        amount. Repeat for multiple outputs
  --sweep ADDRESS       Sweep wallet, transfer all funds to specified address
  --fee FEE, -f FEE     Transaction fee
  --fee-per-kb FEE_PER_KB
                        Transaction fee in sathosis (or smallest denominator)
                        per kilobyte
  --push, -p            Push created transaction to the network
  --import-tx TRANSACTION, -i TRANSACTION
                        Import raw transaction hash or transaction dictionary
                        in wallet and sign it with available key(s)
  --import-tx-file FILENAME_TRANSACTION, -a FILENAME_TRANSACTION
                        Import transaction dictionary or raw transaction
                        string from specified filename and sign it with
                        available key(s)
```

## 7.3 Add a new Service Provider

The Service class connects to providers such as Blockchain.info or Blockexplorer.com to retreive transaction, network, block, address information, etc

The Service class automatically selects a provider which has requested method available and selects another provider if method fails.

### 7.3.1 Steps to add a new provider

- The preferred way is to create a github clone and update code there (and do a pull request...)
- Add the provider settings in the providers.json file in the configuration directory.

Example:

```
{
    "bitgo": {
        "provider": "bitgo",
        "network": "bitcoin",
        "client_class": "BitGo",
        "provider_coin_id": "",
        "url": "https://www.bitgo.com/api/v1/",
        "api_key": "",
        "priority": 10,
        "denominator": 1,
        "network_overrides": null
    }
}
```

- Create a new Service class in bitcoinlib.services. Create a method for available API calls and rewrite output if needed.

Example:

```
from bitcoinlib.services.baseclient import BaseClient

PROVIDERNAME = 'bitgo'


class BitGoClient(BaseClient):

    def __init__(self, network, base_url, denominator, api_key=''):
        super(self.__class__, self).\
            __init__(network, PROVIDERNAME, base_url, denominator, api_key)

    def compose_request(self, category, data, cmd='', variables=None, method='get'):
        if data:
            data = '/' + data
        url_path = category + data
        if cmd:
            url_path += '/' + cmd
        return self.request(url_path, variables, method=method)

    def estimatefee(self, blocks):
        res = self.compose_request('tx', 'fee', variables={'numBlocks': blocks})
        return res['feePerKb']
```

- Add this service class to __init__.py

```
import bitcoinlib.services.bitgo
```

- Remove install.log file in bitcoinlib's log directory, this will copy all provider settings next time you run the bitcoin library. See 'initialize_lib' method in main.py
- Specify new provider and create service class object to test your new class and it's method

```
from bitcoinlib import services

srv = Service(providers=['blockexplorer'])
print(srv.estimatefee(5))
```

## 7.4 How to connect bitcoinlib to a bitcoin node

This manual explains how to connect to a bitcoind server on your localhost or an a remote server.

Running your own bitcoin node allows you to create a large number of requests, faster response times, and more control, privacy and independence. However you need to install and maintain it and it used a lot of resources.

### 7.4.1 Bitcoin node settings

This manual assumes you have a full bitcoin node up and running. For more information on how to install a full node read https://bitcoin.org/en/full-node

Please make sure you have server and txindex option set to 1.

So your bitcoin.conf file for testnet should look something like this. For mainnet use port 8332, and remove the 'testnet=1' line.

```
[rpc]
rpcuser=bitcoinrpc
rpcpassword=some_long_secure_password
server=1
port=18332
txindex=1
testnet=1
```

### 7.4.2 Connect using config files

Bitcoinlib looks for bitcoind config files on localhost. So if you running a full bitcoin node from your local PC as the same user everything should work out of the box.

Config files are read from the following files in this order: * [USER_HOME_DIR]/.bitcoinlib/config/bitcoin.conf * [USER_HOME_DIR]/.bitcoin/bitcoin.conf

If your config files are at another location, you can specify this when you create a BitcoindClient instance.

```
from bitcoinlib.services.bitcoind import BitcoindClient

bdc = BitcoindClient.from_config('/usr/local/src/.bitcoinlib/config/bitcoin.conf')
txid = 'e0cee8955f516d5ed333d081a4e2f55b999debfff91a49e8123d20f7ed647ac5'
rt = bdc.getrawtransaction(txid)
print("Raw: %s" % rt)
```

### 7.4.3 Connect using provider settings

Connection settings can also be added to the service provider settings file in .bitcoinlib/config/providers.json

Example:

```
"bitcoind.testnet": {
  "provider": "bitcoind",
  "network": "testnet",
  "client_class": "BitcoindClient",
  "url": "http://user:password@server_url:18332",
  "api_key": "",
  "priority": 11,
  "denominator": 100000000
}
```

### 7.4.4 Connect using base_url argument

Another options is to pass the 'base_url' argument to the BitcoindClient object directly.

This provides more flexibility but also responsibility to store user and password information secure.

```
from bitcoinlib.services.bitcoind import BitcoindClient

base_url = 'http://user:password@server_url:18332'
bdc = BitcoindClient(base_url=base_url)
```

(continues on next page)

(continued from previous page)

```
txid = 'e0cee8955f516d5ed333d081a4e2f55b999debfff91a49e8123d20f7ed647ac5'
rt = bdc.getrawtransaction(txid)
print("Raw: %s" % rt)
```

# 7.5 bitcoinlib

## 7.5.1 bitcoinlib package

### Subpackages

### bitcoinlib.config package

### Submodules

### bitcoinlib.config.opcodes module

bitcoinlib.config.opcodes.**opcode**(*name*, *as_bytes=True*)

Get integer or byte character value of OP code by name.

**Parameters**

- **name** (*str*) – Name of OP code as defined in opcodenames
- **as_bytes** (*bool*) – Return as byte or int? Default is bytes

**Return int, bytes**

### bitcoinlib.config.secp256k1 module

### Module contents

### bitcoinlib.services package

### Submodules

### bitcoinlib.services.authproxy module

Copyright 2011 Jeff Garzik

AuthServiceProxy has the following improvements over python-jsonrpc's ServiceProxy class:

- HTTP connections persist for the life of the AuthServiceProxy object (if server supports HTTP/1.1)
- sends protocol 'version', per JSON-RPC 1.1
- sends proper, incrementing 'id'
- sends Basic HTTP authentication headers
- parses all JSON numbers that look like floats as Decimal
- uses standard Python json lib

Previous copyright, from python-jsonrpc/jsonrpc/proxy.py:

Copyright (c) 2007 Jan-Klaas Kollhof

This file is part of jsonrpc.

jsonrpc is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this software; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

**class** bitcoinlib.services.authproxy.**AuthServiceProxy**(*service_url*, *service_name=None*, *timeout=30*, *connection=None*)

Bases: object

**batch_**(*rpc_calls*)

Batch RPC call. Pass array of arrays: [ [ "method", params... ], ... ] Returns array of results.

bitcoinlib.services.authproxy.**EncodeDecimal**(*o*)

**exception** bitcoinlib.services.authproxy.**JSONRPCException**(*rpc_error*)

Bases: Exception

### bitcoinlib.services.baseclient module

**class** bitcoinlib.services.baseclient.**BaseClient**(*network*, *provider*, *base_url*, *denominator*, *api_key=''*, *provider_coin_id=''*, *network_overrides=None*)

Bases: object

**request**(*url_path*, *variables=None*, *method='get'*)

**exception** bitcoinlib.services.baseclient.**ClientError**(*msg=''*)

Bases: Exception

### bitcoinlib.services.bitcoind module

**class** bitcoinlib.services.bitcoind.**BitcoindClient**(*network='bitcoin'*, *base_url=''*, *denominator=100000000*, **args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

Class to interact with bitcoind, the Bitcoin deamon

Open connection to bitcoin node

**Parameters**

- **network** – Bitcoin mainnet or testnet. Default is bitcoin mainnet
- **base_url** – Connection URL in format http(s)://user:password@host:port.
- **denominator** – Denominator for this currency. Should be always 100000000 (satoshis) for bitcoin

**Type** str

**Type** str

**Type** str

**block_count**()

**estimatefee**(*blocks*)

**static from_config**(*network='bitcoin'*)
Read settings from bitcoind config file

**Parameters**

- **configfile** – Path to config file. Leave empty to look in default places
- **network** – Bitcoin mainnet or testnet. Default is bitcoin mainnet

**Type** str

**Type** str

**Return BitcoindClient**

**getrawtransaction**(*txid*)

**gettransaction**(*txid*)

**getutxos**(*addresslist*)

**sendrawtransaction**(*rawtx*)

**exception** bitcoinlib.services.bitcoind.**ConfigError**(*msg=''*)
Bases: Exception

### bitcoinlib.services.bitcoinlibtest module

**class** bitcoinlib.services.bitcoinlibtest.**BitcoinLibTestClient**(*network*, *base_url*, *denominator*, **args*)
Bases: *bitcoinlib.services.baseclient.BaseClient*

Dummy service client for bitcoinlib test network. Only used for testing.

Does not make any connection to a service provider, so can be used offline.

**block_count**()

**estimatefee**(*blocks*)
Dummy estimate fee method for the bitcoinlib testnet.

**Parameters** **blocks** (*int*) – Number of blocks

**Return int** Fee as 100000 // number of blocks

**getbalance**(*addresslist*)
Dummy getbalance method for bitcoinlib testnet

**Parameters** **addresslist** (*list*) – List of addresses

**Return int**

**getutxos**(*addresslist*, *utxos_per_address=2*)
Dummy method to retreive UTXO's. This method creates a new UTXO for each address provided out of the testnet void, which can be used to create test transactions for the bitcoinlib testnet.

**Parameters**

- **addresslist** (*list*) – List of addresses
- **utxos_per_address** (*int*) – Number of UTXO's to be created per address

**Return list** The created UTXO set

**sendrawtransaction**(*rawtx*)
Dummy method to send transactions on the bitcoinlib testnet. The bitcoinlib testnet does not exists, so it just returns the transaction hash.

**Parameters** **rawtx** (*bytes, str*) – A raw transaction hash

**Return str** Transaction hash

### bitcoinlib.services.bitgo module

**class** bitcoinlib.services.bitgo.**BitGoClient**(*network*, *base_url*, *denominator*, **args*)
Bases: *bitcoinlib.services.baseclient.BaseClient*

**compose_request**(*category*, *data*, *cmd=''*, *variables=None*, *method='get'*)

**estimatefee**(*blocks*)

**getbalance**(*addresslist*)

**getrawtransaction**(*txid*)

**gettransaction**(*tx_id*)

**gettransactions**(*addresslist*)

**getutxos**(*addresslist*)

### bitcoinlib.services.blockchaininfo module

**class** bitcoinlib.services.blockchaininfo.**BlockchainInfoClient**(*network*, *base_url*, *denominator*, **args*)
Bases: *bitcoinlib.services.baseclient.BaseClient*

**block_count**()

**compose_request**(*cmd*, *parameter=''*, *variables=None*, *method='get'*)

**getbalance**(*addresslist*)

**getrawtransaction**(*tx_id*)

**gettransaction**(*tx_id*)

**gettransactions**(*addresslist*)

**getutxos**(*addresslist*)

### bitcoinlib.services.blockchair module

**class** bitcoinlib.services.blockchair.**BlockChairClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**block_count**()

Get latest block number: The block number of last block in longest chain on the blockchain

**Return int**

**compose_request**(*command*, *query_vars=None*, *data=None*, *offset=0*)

**estimatefee**(*blocks*)

**getbalance**(*addresslist*)

**gettransaction**(*tx_id*)

**gettransactions**(*addresslist*)

**getutxos**(*addresslist*)

### bitcoinlib.services.blockcypher module

**class** bitcoinlib.services.blockcypher.**BlockCypher**(*network*, *base_url*, *denominator*, **args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**block_count**()

**compose_request**(*function*, *data*, *parameter=''*, *variables=None*, *method='get'*)

**estimatefee**(*blocks*)

**getbalance**(*addresslist*)

**getrawtransaction**(*tx_id*)

**gettransaction**(*tx_id*)

**gettransactions**(*addresslist*, *unspent_only=False*)

**getutxos**(*addresslist*)

**sendrawtransaction**(*rawtx*)

### bitcoinlib.services.blockexplorer module

**class** bitcoinlib.services.blockexplorer.**BlockExplorerClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**block_count**()

**compose_request**(*category*, *data*, *cmd=''*, *variables=None*, *method='get'*)

**getbalance**(*addresslist*)

**getrawtransaction**(*tx_id*)

**gettransaction**(*tx_id*)

**gettransactions**(*addresslist*)

**getutxos**(*addresslist*)

**sendrawtransaction**(*rawtx*)

### bitcoinlib.services.blocktrail module

**class** bitcoinlib.services.blocktrail.**BlockTrail**(*network*, *base_url*, *denominator*, *api_key*, **args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**compose_request**(*function*, *data*, *parameter=''*, *variables=None*, *method='get'*, *page=1*)

**estimatefee**(*blocks*)

**getbalance**(*addresslist*)

**gettransaction**(*tx_id*)

**gettransactions**(*addresslist*)

**getutxos**(*addresslist*)

### bitcoinlib.services.chainso module

**class** bitcoinlib.services.chainso.**ChainSo**(*network*, *base_url*, *denominator*, **args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**block_count**()

**compose_request**(*function*, *data=''*, *parameter=''*, *variables=None*, *method='get'*)

**getbalance**(*addresslist*)

**getrawtransaction**(*txid*)

**gettransaction**(*tx_id*)

**gettransactions**(*address_list*)

**getutxos**(*addresslist*)

**sendrawtransaction**(*rawtx*)

### bitcoinlib.services.coinfees module

**class** bitcoinlib.services.coinfees.**CoinfeesClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**compose_request**(*category*, *cmd*, *method='get'*)

**estimatefee**(*blocks*)

### bitcoinlib.services.cryptoid module

**class** bitcoinlib.services.cryptoid.**CryptoID**(*network*, *base_url*, *denominator*, **args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**block_count**()

**compose_request**(*func=None*, *path_type='api'*, *variables=None*, *method='get'*)

**getbalance**(*addresslist*)

**getrawtransaction**(*tx_id*)

**gettransaction**(*tx_id*)

**gettransactions**(*addresslist*)

**getutxos**(*addresslist*)

### bitcoinlib.services.dashd module

**exception** bitcoinlib.services.dashd.**ConfigError**(*msg=''*)
Bases: Exception

**class** bitcoinlib.services.dashd.**DashdClient**(*network='dash'*, *base_url=''*, *denominator=100000000*, **args*)
Bases: *bitcoinlib.services.baseclient.BaseClient*

Class to interact with dashd, the Dash deamon

Open connection to dashcore node

**Parameters**

- **network** – Dash mainnet or testnet. Default is dash mainnet
- **base_url** – Connection URL in format http(s)://user:password@host:port.
- **denominator** – Denominator for this currency. Should be always 100000000 (satoshis) for Dash

**Type** str

**Type** str

**Type** str

**block_count**()

**estimatefee**(*blocks*)

**static from_config**(*network='dash'*)
Read settings from dashd config file

**Parameters**

- **configfile** – Path to config file. Leave empty to look in default places
- **network** – Dash mainnet or testnet. Default is dash mainnet

**Type** str

**Type** str

**Return DashdClient**

**getrawtransaction**(*txid*)

**gettransaction**(*txid*)

**sendrawtransaction**(*rawtx*)

### bitcoinlib.services.estimatefee module

**class** bitcoinlib.services.estimatefee.**EstimateFeeClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**compose_request**(*cmd*, *parameter*, *method='get'*)

**estimatefee**(*blocks*)

### bitcoinlib.services.litecoind module

**exception** bitcoinlib.services.litecoind.**ConfigError**(*msg=''*)

Bases: Exception

**class** bitcoinlib.services.litecoind.**LitecoindClient**(*network='litecoin'*, *base_url=''*, *denominator=100000000*, **args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

Class to interact with litecoind, the Litecoin deamon

Open connection to litecoin node

**Parameters**

- **network** – Litecoin mainnet or testnet. Default is litecoin mainnet
- **base_url** – Connection URL in format http(s)://user:password@host:port.
- **denominator** – Denominator for this currency. Should be always 100000000 (satoshis) for litecoin

**Type** str

**Type** str

**Type** str

**block_count**()

**estimatefee**(*blocks*)

**static from_config**(*network='litecoin'*)

Read settings from litecoind config file

**Parameters**

- **configfile** – Path to config file. Leave empty to look in default places
- **network** – Litecoin mainnet or testnet. Default is litecoin mainnet

**Type** str

**Type** str

**Return LitecoindClient**

**getrawtransaction**(*txid*)

**gettransaction**(*txid*)

**getutxos**(*addresslist*)

**sendrawtransaction**(*rawtx*)

### bitcoinlib.services.litecoreio module

**class** bitcoinlib.services.litecoreio.**LitecoreIOClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**block_count**()

**compose_request**(*category*, *data*, *cmd=''*, *variables=None*, *method='get'*)

**getbalance**(*addresslist*)

**getrawtransaction**(*tx_id*)

**gettransaction**(*tx_id*)

**gettransactions**(*addresslist*)

**getutxos**(*addresslist*)

**sendrawtransaction**(*rawtx*)

### bitcoinlib.services.multiexplorer module

**class** bitcoinlib.services.multiexplorer.**MultiexplorerClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**compose_request**(*func*, *variables*, *service_id='fallback'*, *include_raw=False*, *method='get'*)

### bitcoinlib.services.services module

**class** bitcoinlib.services.services.**Service**(*network='bitcoin'*, *min_providers=1*, *max_providers=1*, *providers=None*)

Bases: `object`

Class to connect to various cryptocurrency service providers. Use to receive network and blockchain information, get specific transaction information, current network fees or push a raw transaction.

The Service class connects to 1 or more service providers at random to retrieve or send information. When a certain service provider fail it automatically tries another one.

Open a service object for the specified network. By default the object connect to 1 service provider, but you can specify a list of providers or a minimum or maximum number of providers.

**Parameters**

- **network** (*str, Network*) – Specify network used
- **min_providers** – Minimum number of providers to connect to. Default is 1. Use for instance to receive

fee information from a number of providers and calculate the average fee. :type min_providers: int :param max_providers: Maximum number of providers to connect to. Default is 1. :type max_providers: int :param providers: List of providers to connect to. Default is all providers and select a provider at random. :type providers: list, str

**block_count**()

Get latest block number: The block number of last block in longest chain on the blockchain

**Return int**

**estimatefee**(*blocks=3*)
Estimate fee per kilobyte for a transaction for this network with expected confirmation within a certain amount of blocks

**Parameters** **blocks** (*int*) – Expection confirmation time in blocks. Default is 3.

**Return int** Fee in smallest network denominator (satoshi)

**getbalance**(*addresslist*, *addresses_per_request=5*)
Get balance for each address in addresslist provided

**Parameters**

- **addresslist** (*list, str*) – Address or list of addresses
- **addresses_per_request** (*int*) – Maximum number of addresses per request. Default is 5. Use lower setting when you experience timeouts or service request errors, or higher when possible.

**Return dict** Balance per address

**getrawtransaction**(*txid*)
Get a raw transaction by its transaction hash

**Parameters** **txid** (*str, bytes*) – Transaction identification hash

**Return str** Raw transaction as hexstring

**gettransaction**(*txid*)
Get a transaction by its transaction hash

**Parameters** **txid** (*str, bytes*) – Transaction identification hash

**Return Transaction** A single transaction object

**gettransactions**(*addresslist*, *addresses_per_request=5*)
Get all transactions for each address in addresslist

**Parameters**

- **addresslist** (*list, str*) – Address or list of addresses
- **addresses_per_request** (*int*) – Maximum number of addresses per request. Default is 5. Use lower setting when you experience timeouts or service request errors, or higher when possible.

**Return list** List of Transaction objects

**getutxos**(*addresslist*, *addresses_per_request=5*)
Get list of unspent outputs (UTXO's) per address

**Parameters**

- **addresslist** (*list, str*) – Address or list of addresses
- **addresses_per_request** (*int*) – Maximum number of addresses per request. Default is 5. Use lower setting when you experience timeouts or service request errors, or higher when possible.

**Return dict** UTXO's per address

**sendrawtransaction**(*rawtx*)
Push a raw transaction to the network

**Parameters** **rawtx** (*str, bytes*) – Raw transaction as hexstring

**Return dict** Send transaction result

**exception** bitcoinlib.services.services.**ServiceError**(*msg=''*)
Bases: Exception

### Module contents

### bitcoinlib.tools package

### Submodules

### bitcoinlib.tools.cli_wallet module

Used by autodoc_mock_imports.

### bitcoinlib.tools.mnemonic_key_create module

Used by autodoc_mock_imports.

### bitcoinlib.tools.sign_raw module

Used by autodoc_mock_imports.

### bitcoinlib.tools.sign_raw_mnemonic module

Used by autodoc_mock_imports.

### bitcoinlib.tools.wallet_multisig_2of3 module

Used by autodoc_mock_imports.

### bitcoinlib.tools.wallet_multisig_3of5 module

Used by autodoc_mock_imports.

### Module contents

Used by autodoc_mock_imports.

### Submodules

### bitcoinlib.db module

**class** bitcoinlib.db.**DbInit**(*databasefile='/home/docs/.bitcoinlib/database/bitcoinlib.sqlite'*)
Bases: object

Initialize database and open session

Import data if database did not exist yet

**class** bitcoinlib.db.**DbKey**(***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Base

Database definitions for keys in Sqlalchemy format

Part of a wallet, and used by transactions

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**account_id**

**address**

**address_index**

**balance**

**change**

**compressed**

**cosigner_id**

**depth**

**encoding**

**id**

**is_private**

**key_type**

**multisig_children**

**multisig_parents**

**name**

**network**

**network_name**

**parent_id**

**path**

**private**

**public**

**purpose**

**transaction_inputs**

**transaction_outputs**

**used**

**wallet**

**wallet_id**

**wif**

**class** bitcoinlib.db.**DbKeyMultisigChildren**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Use many-to-many relationship for multisig keys. A multisig keys contains 2 or more child keys and a child key can be used in more then one multisig key.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**child_id**

**key_order**

**parent_id**

**class** bitcoinlib.db.**DbNetwork**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Database definitions for networks in Sqlalchemy format

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**description**

**name**

**class** bitcoinlib.db.**DbTransaction**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Database definitions for transactions in Sqlalchemy format

Refers to 1 or more keys which can be part of a wallet

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**block_hash**

**block_height**

**coinbase**

**confirmations**

**date**

**fee**

**hash**

**id**

**input_total**

**inputs**

**locktime**

**network**

**network_name**

**output_total**

**outputs**

**raw**

**size**

**status**

**version**

**wallet**

**wallet_id**

**witness_type**

**class** bitcoinlib.db.**DbTransactionInput**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Transaction Input Table

Relates to Transaction table and Key table

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**double_spend**

**index_n**

**key**

**key_id**

**output_n**

**prev_hash**

**script**

**script_type**

**sequence**

**transaction**

**transaction_id**

**value**

**witness_type**

**class** bitcoinlib.db.**DbTransactionOutput**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Transaction Output Table

Relates to Transaction and Key table

When spent is False output is considered an UTXO

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**key**

**key_id**

**output_n**

**script**

**script_type**

**spent**

**transaction**

**transaction_id**

**value**

**class** bitcoinlib.db.**DbWallet**(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

Database definitions for wallets in Sqlalchemy format

Contains one or more keys.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**children**

**cosigner_id**

**default_account_id**

**encoding**
Default encoding to use for address generation

**id**

**key_path**

**keys**

**main_key_id**

**multisig**

**multisig_n_required**
Number of required signature for multisig, only used for multisignature master key

**name**

**network**

**network_name**

**owner**

**parent_id**

**purpose**

**scheme**

**sort_keys**
Sort keys in multisig wallet

**transactions**

**witness_type**

**class** bitcoinlib.db.**TransactionType**
Bases: enum.Enum

Incoming or Outgoing transaction Enumeration

**incoming = 1**

**outgoing = 2**

### bitcoinlib.encoding module

**exception** bitcoinlib.encoding.**EncodingError**(*msg=''*)
Bases: Exception

Log and raise encoding errors

bitcoinlib.encoding.**addr_base58_to_pubkeyhash**(*address*, *as_hex=False*)
Convert Base58 encoded address to public key hash

**Parameters**

- **address** (*str, bytes*) – Crypto currency address in base-58 format
- **as_hex** (*bool*) – Output as hexstring

**Return bytes, str** Public Key Hash

bitcoinlib.encoding.**addr_bech32_to_pubkeyhash**(*bech*, *prefix=None*, *include_witver=False*, *as_hex=False*)
Decode bech32 / segwit address to public key hash

Validate the Bech32 string, and determine HRP and data

**Parameters**

- **bech** (*str*) – Bech32 address to convert
- **prefix** (*str*) – Address prefix called Human-readable part. Default is None and tries to derive prefix, for bitcoin specify 'bc' and for bitcoin testnet 'tb'
- **include_witver** (*bool*) – Include witness version in output? Default is False
- **as_hex** (*bool*) – Output public key hash as hex or bytes. Default is False

**Return str** Public Key Hash

bitcoinlib.encoding.**addr_to_pubkeyhash**(*address*, *as_hex=False*, *encoding='base58'*)
Convert base58 or bech32 address to public key hash

**Parameters**

- **address** (*str*) – Crypto currency address in base-58 format
- **as_hex** (*bool*) – Output as hexstring

- **encoding** (*str*) – Address encoding used: base58 or bech32

**Return bytes, str** public key hash

bitcoinlib.encoding.**change_base**(*chars*, *base_from*, *base_to*, *min_length=0*, *output_even=None*, *output_as_list=None*)

Convert input chars from one base to another.

From and to base can be any base. If base is not found a array of index numbers will be returned

Examples: > change_base('FF', 16, 10) will return 256 > change_base(100, 16, 2048) will return [100]

**Parameters**

- **chars** (*any*) – Input string
- **base_from** (*int, str*) – Base number or name from input
- **base_to** (*int, str*) – Base number or name for output
- **min_length** (*int*) – Minimal output length. Required for decimal, advised for all output to avoid leading zeros conversion problems.
- **output_even** (*bool*) – Specify if output must contain a even number of characters. Sometimes handy for hex conversions.
- **output_as_list** (*bool*) – Always output as list instead of string.

**Return str, list** Base converted input as string or list.

bitcoinlib.encoding.**convert_der_sig**(*signature*, *as_hex=True*)

Convert DER encoded signature to signature

**Parameters**

- **signature** (*bytes*) – DER signature
- **as_hex** (*bool*) – Output as hexstring

**Return bytes, str** Signature

bitcoinlib.encoding.**convertbits**(*data*, *frombits*, *tobits*, *pad=True*)

'General power-of-2 base conversion'

Source: https://github.com/sipa/bech32/tree/master/ref/python

**Parameters**

- **data** (*list, bytearray*) – Data values to convert
- **frombits** (*int*) – Number of bits in source data
- **tobits** (*int*) – Number of bits in result data
- **pad** (*bool*) – Use padding zero's or not. Default is True

**Return list** Converted values

bitcoinlib.encoding.**double_sha256**(*string*, *as_hex=False*)

Get double SHA256 hash of string

**Parameters**

- **string** (*bytes*) – String to be hashed
- **as_hex** – Return value as hexadecimal string. Default is False

:type as_hex

**Return bytes, str**

bitcoinlib.encoding.**hash160**(*string*)
Creates a RIPEMD-160 + SHA256 hash of the input string

**Parameters** **string** (*bytes*) – Script

**Return bytes** RIPEMD-160 hash of script

bitcoinlib.encoding.**int_to_varbyteint**(*inp*)
Convert integer to CompactSize Variable length integer in byte format.

See https://en.bitcoin.it/wiki/Protocol_documentation#Variable_length_integer for specification

**Parameters** **inp** (*int*) – Integer to convert

**Returns** byteint: 1-9 byte representation as integer

bitcoinlib.encoding.**normalize_string**(*string*)
Normalize a string to the default NFKD unicode format See https://en.wikipedia.org/wiki/Unicode_equivalence#Normalization

**Parameters** **string** (*bytes, bytearray, str*) – string value

**Returns** string

bitcoinlib.encoding.**normalize_var**(*var*, *base=256*)
For Python 2 convert variabele to string For Python 3 convert to bytes Convert decimals to integer type

**Parameters**

- **var** (*str, byte, bytearray, unicode*) – input variable in any format
- **base** (*int*) – specify variable format, i.e. 10 for decimal, 16 for hex

**Returns** Normalized var in string for Python 2, bytes for Python 3, decimal for base10

bitcoinlib.encoding.**pubkeyhash_to_addr**(*pubkeyhash*, *prefix=None*, *encoding='base58'*)
Convert public key hash to base58 encoded address

**Parameters**

- **pubkeyhash** (*bytes, str*) – Public key hash
- **prefix** (*str, bytes*) – Prefix version byte of network, default is bitcoin ''
- **encoding** (*str*) – Encoding of address to calculate: base58 or bech32. Default is base58

**Return str** Base58 or bech32 encoded address

bitcoinlib.encoding.**pubkeyhash_to_addr_base58**(*pubkeyhash*, *prefix=b'\x00'*)
Convert public key hash to base58 encoded address

**Parameters**

- **pubkeyhash** (*bytes, str*) – Public key hash
- **prefix** (*str, bytes*) – Prefix version byte of network, default is bitcoin ''

**Return str** Base-58 encoded address

bitcoinlib.encoding.**pubkeyhash_to_addr_bech32**(*pubkeyhash*, *prefix='bc'*, *witver=0*, *separator='1'*)
Encode public key hash as bech32 encoded (segwit) address

Format of address is prefix/hrp + seperator + bech32 address + checksum

For more information see BIP173 proposal at https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki

**Parameters**

- **pubkeyhash** (*str, bytes, bytearray*) – Public key hash
- **prefix** (*str*) – Address prefix or Human-readable part. Default is 'bc' an abbreviation of Bitcoin. Use 'tb' for testnet.
- **witver** (*int*) – Witness version between 0 and 16
- **separator** (*str*) – Separator char between hrp and data, should always be left to '1' otherwise its not standard.

**Return str** Bech32 encoded address

bitcoinlib.encoding.**to_bytearray**(*string*)
Convert String, Unicode or Bytes to Python 2 and 3 compatible ByteArray

**Parameters** **string** (*bytes, str, bytearray*) – String, Unicode, Bytes or ByteArray

**Return bytearray**

bitcoinlib.encoding.**to_bytes**(*string*, *unhexlify=True*)
Convert String, Unicode or ByteArray to Bytes

**Parameters**

- **string** (*str, unicode, bytes, bytearray*) – String to convert
- **unhexlify** (*bool*) – Try to unhexlify hexstring

**Returns** Bytes var

bitcoinlib.encoding.**to_hexstring**(*string*)
Convert Bytes or ByteArray to hexadecimal string

**Parameters** **string** (*bytes, bytearray, str*) – Variable to convert to hex string

**Returns** hexstring

bitcoinlib.encoding.**varbyteint_to_int**(*byteint*)
Convert CompactSize Variable length integer in byte format to integer.

See https://en.bitcoin.it/wiki/Protocol_documentation#Variable_length_integer for specification

**Parameters** **byteint** (*bytes, list, bytearray*) – 1-9 byte representation

**Return int** normal integer

bitcoinlib.encoding.**varstr**(*string*)
Convert string to variably sized string: Bytestring preceeded with length byte

**Parameters** **string** (*bytes, str*) – String input

**Return bytes** varstring

### bitcoinlib.keys module

**class** bitcoinlib.keys.**Address**(*data=''*, *hashed_data=''*, *prefix=None*, *script_type=None*, *compressed=None*, *encoding=None*, *witness_type=None*, *depth=None*, *change=None*, *address_index=None*, *network='bitcoin'*, *network_overrides=None*)
Bases: object

Class to store, convert and analyse various address types as representation of public keys or scripts hashes

Initialize an Address object. Specify a public key, redeemscript or a hash.

**Parameters**

- **data** (*str, bytes*) – Public key, redeem script or other type of script.
- **hashed_data** (*str, bytes*) – Hash of a public key or script. Will be generated if 'data' parameter is provided
- **prefix** (*str, bytes*) – Address prefix. Use default network / script_type prefix if not provided
- **script_type** (*str*) – Type of script, i.e. p2sh or p2pkh.
- **encoding** (*str*) – Address encoding. Default is base58 encoding, for native segwit addresses specify bech32 encoding
- **witness_type** (*str*) – Specify 'legacy', 'segwit' or 'p2sh-segwit'. Legacy for old-style bitcoin addresses, segwit for native segwit addresses and p2sh-segwit for segwit embedded in a p2sh script. Leave empty to derive automatically from script type if possible
- **network** (*str, Network*) – Bitcoin, testnet, litecoin or other network
- **network_overrides** (*dict*) – Override network settings for specific prefixes, i.e.: {"prefix_address_p2sh": "32"}. Used by settings in providers.json

**as_dict**()
Get current Address class as dictionary. Byte values are represented by hexadecimal strings

**Return dict**

**as_json**()
Get current key as json formatted string

**Return str**

**classmethod import_address**(*address*, *compressed=None*, *encoding=None*, *depth=None*, *change=None*, *address_index=None*, *network=None*, *network_overrides=None*)
Import an address to the Address class. Specify network if available, otherwise it will be derived form the address.

**Parameters**

- **address** (*str*) – Address to import
- **compressed** (*bool*) – Is key compressed or not, default is None
- **encoding** (*str*) – Address encoding. Default is base58 encoding, for native segwit addresses specify bech32 encoding. Leave empty to derive from address
- **network** (*str*) – Bitcoin, testnet, litecoin or other network
- **network_overrides** (*dict*) – Override network settings for specific prefixes, i.e.: {"prefix_address_p2sh": "32"}. Used by settings in providers.json

**Return Address**

**with_prefix**(*prefix*)
Convert address using another prefix

**Parameters** **prefix** (*str, bytes*) – Address prefix

**Return str** Converted address

**exception** bitcoinlib.keys.**BKeyError**(*msg=''*)
Bases: Exception

Handle Key class Exceptions

**class** bitcoinlib.keys.**HDKey**(*import_key=None*, *key=None*, *chain=None*, *depth=0*, *parent_fingerprint=b'x00x00x00x00'*, *child_index=0*, *is_private=True*, *network=None*, *key_type='bip32'*, *passphrase=''*, *compressed=True*, *encoding=None*, *witness_type=None*, *multisig=False*)

Bases: *bitcoinlib.keys.Key*

Class for Hierarchical Deterministic keys as defined in BIP0032

Besides a private or public key a HD Key has a chain code, allowing to create a structure of related keys.

The structure and key-path are defined in BIP0043 and BIP0044.

Hierarchical Deterministic Key class init function. If no import_key is specified a key will be generated with systems cryptographically random function. Import key can be any format normal or HD key (extended key) accepted by get_key_format. If a normal key with no chain part is provided, an chain with only 32 0-bytes will be used.

**Parameters**

- **import_key** (*str, bytes, int, bytearray*) – HD Key to import in WIF format or as byte with key (32 bytes) and chain (32 bytes)
- **key** (*bytes*) – Private or public key (length 32)
- **chain** (*bytes*) – A chain code (length 32)
- **depth** (*int*) – Level of depth in BIP32 key path
- **parent_fingerprint** (*bytes*) – 4-byte fingerprint of parent
- **child_index** (*int*) – Index number of child as integer
- **is_private** (*bool*) – True for private, False for public key. Default is True
- **network** (*str, Network*) – Network name. Derived from import_key if possible
- **key_type** (*str*) – HD BIP32 or normal Private Key. Default is 'bip32'
- **passphrase** (*str*) – Optional passphrase if imported key is password protected
- **compressed** (*bool*) – Is key compressed or not, default is True
- **encoding** (*str*) – Encoding used for address, i.e.: base58 or bech32. Default is base58 or derive from witness type
- **witness_type** (*str*) – Witness type used when creating scripts: legacy, p2sh-segwit or segwit.
- **multisig** (*bool*) – Specify if key is part of multisig wallet, used when creating key representations such as WIF and addreses

**Return HDKey**

**account_key**(*account_id=0*, *purpose=44*, *set_network=None*)

Deprecated since version 0.4.5, use public_master() method instead

Derive account BIP44 key for current master key

**Parameters**

- **account_id** (*int*) – Account ID. Leave empty for account 0
- **purpose** (*int*) – BIP standard used, i.e. 44 for default, 45 for multisig, 84 for segwit
- **set_network** (*str*) – Derive account key for different network. Please note this calls the network_change method and changes the network for current key!

**Return HDKey**

**account_multisig_key**(*account_id=0*, *witness_type='legacy'*)
Deprecated since version 0.4.5, use public_master() method instead

Derives a multisig account key according to BIP44/45 definition. Wrapper for the 'account_key' method.

**Parameters**

- **account_id** (*int*) – Account ID. Leave empty for account 0
- **witness_type** (*str*) – Specify witness type, default is legacy. Use 'segwit' for segregated witness.

**Return HDKey**

**address**(*compressed=None*, *prefix=None*, *script_type=None*, *encoding=None*)
Get address derived from public key

**Parameters**

- **compressed** (*bool*) – Always return compressed address
- **prefix** (*str, bytes*) – Specify versionbyte prefix in hexstring or bytes. Normally doesn't need to be specified, method uses default prefix from network settings
- **script_type** (*str*) – Type of script, i.e. p2sh or p2pkh.
- **encoding** (*str*) – Address encoding. Default is base58 encoding, for segwit you can specify bech32 encoding

**Return str** Base58 encoded address

**as_dict**()
Get current HDKey class as dictionary. Byte values are represented by hexadecimal strings.

**Return collections.OrderedDict**

**as_json**()
Get current key as json formatted string

**Return str**

**child_private**(*index=0*, *hardened=False*, *network=None*)
Use Child Key Derivation (CDK) to derive child private key of current HD Key object.

**Parameters**

- **index** (*int*) – Key index number
- **hardened** (*bool*) – Specify if key must be hardened (True) or normal (False)
- **network** (*str*) – Network name.

**Return HDKey** HD Key class object

**child_public**(*index=0*, *network=None*)
Use Child Key Derivation to derive child public key of current HD Key object.

**Parameters**

- **index** (*int*) – Key index number
- **network** (*str*) – Network name.

**Return HDKey** HD Key class object

**fingerprint**()
Get key fingerprint: the last for bytes of the hash160 of this key.

**Return bytes**

**static from_passphrase**(*password=''*, *network='bitcoin'*, *compressed=True*, *encoding=None*, *witness_type='legacy'*, *multisig=False*)

Create key from Mnemonic passphrase

**Parameters**

- **passphrase** (*str*) – Mnemonic passphrase, list of words as string seperated with a space character
- **password** (*str*) – Password to protect passphrase
- **network** (*str, Network*) – Network to use
- **compressed** (*bool*) – Is key compressed or not, default is True
- **encoding** (*str*) – Encoding used for address, i.e.: base58 or bech32. Default is base58 or derive from witness type
- **witness_type** (*str*) – Witness type used when creating scripts: legacy, p2sh-segwit or segwit.
- **multisig** (*bool*) – Specify if key is part of multisig wallet, used when creating key representations such as WIF and addreses

**Return HDKey**

**static from_seed**(*key_type='bip32'*, *network='bitcoin'*, *compressed=True*, *encoding=None*, *witness_type='legacy'*, *multisig=False*)

Used by class init function, import key from seed

**Parameters**

- **import_seed** (*str, bytes*) – Private key seed as bytes or hexstring
- **key_type** (*str*) – Specify type of key, default is BIP32
- **network** (*str, Network*) – Network to use
- **compressed** (*bool*) – Is key compressed or not, default is True
- **encoding** (*str*) – Encoding used for address, i.e.: base58 or bech32. Default is base58 or derive from witness type
- **witness_type** (*str*) – Witness type used when creating scripts: legacy, p2sh-segwit or segwit.
- **multisig** (*bool*) – Specify if key is part of multisig wallet, used when creating key representations such as WIF and addreses

**Return HDKey**

**info**()

Prints key information to standard output

**network_change**(*new_network*)

Change network for current key

**Parameters** **new_network** (*str*) – Name of new network

**Return bool** True

**public**()

Public version of current private key. Strips all private information from HDKey object, returns deepcopy version of current object

**Return HDKey**

**public_master**(*account_id=0*, *purpose=None*, *multisig=None*, *witness_type=None*, *as_private=False*)

Derives a public master key for current HDKey.

**Parameters**

- **account_id** (*int*) – Account ID. Leave empty for account 0
- **purpose** (*int*) – BIP standard used, i.e. 44 for default, 45 for multisig, 84 for segwit.
- **multisig** (*bool*) – Key is part of a multisignature wallet?
- **witness_type** (*str*) – Specify witness type, default is legacy. Use 'segwit' or 'p2sh-segwit' for segregated witness.
- **as_private** – Return private key if available. Default is to return public key

**Return HDKey**

**public_master_multisig**(*account_id=0*, *purpose=None*, *witness_type=None*, *as_private=False*)

Derives a public master key for current HDKey for use with multi signature wallets. Wrapper for the public_master() method.

**Parameters**

- **account_id** (*int*) – Account ID. Leave empty for account 0
- **purpose** (*int*) – BIP standard used, i.e. 44 for default, 45 for multisig, 84 for segwit.
- **witness_type** (*str*) – Specify witness type, default is legacy. Use 'segwit' or 'p2sh-segwit' for segregated witness.
- **as_private** – Return private key if available. Default is to return public key

**Return HDKey**

**subkey_for_path**(*path*, *network=None*)

Determine subkey for HD Key for given path. Path format: m / purpose' / coin_type' / account' / change / address_index Example: m/44'/0'/0'/0/2 See BIP0044 bitcoin proposal for more explanation.

**Parameters**

- **path** (*str, list*) – BIP0044 key path
- **network** (*str*) – Network name.

**Return HDKey** HD Key class object of subkey

**wif**(*is_private=None*, *child_index=None*, *prefix=None*, *witness_type=None*, *multisig=None*)

Get Extended WIF of current key

**Parameters**

- **is_private** (*bool*) – Return public or private key
- **child_index** (*int*) – Change child index of output WIF key
- **prefix** (*str, bytes*) – Specify version prefix in hexstring or bytes. Normally doesn't need to be specified, method uses default prefix from network settings
- **witness_type** (*str*) – Specify witness type, default is legacy. Use 'segwit' for segregated witness.
- **multisig** (*bool*) – Key is part of a multisignature wallet?

**Return str** Base58 encoded WIF key

**wif_key**(*prefix=None*)
Get WIF of Key object. Call to parent object Key.wif()

**Parameters** **prefix** (*str, bytes*) – Specify versionbyte prefix in hexstring or bytes. Normally doesn't need to be specified, method uses default prefix from network settings

**Return str** Base58Check encoded Private Key WIF

**wif_private**(*prefix=None*, *witness_type=None*, *multisig=None*)
Get Extended WIF private key. Wrapper for the wif() method

**Parameters** **prefix** – Specify version prefix in hexstring or bytes. Normally doesn't need to be specified,

method uses default prefix from network settings :type prefix: str, bytes :param witness_type: Specify witness type, default is legacy. Use 'segwit' for segregated witness. :type witness_type: str :param multisig: Key is part of a multisignature wallet? :type multisig: bool

**Return str** Base58 encoded WIF key

**wif_public**(*prefix=None*, *witness_type=None*, *multisig=None*)
Get Extended WIF public key. Wrapper for the wif() method

**Parameters**

- **prefix** (*str, bytes*) – Specify version prefix in hexstring or bytes. Normally doesn't need to be specified, method uses default prefix from network settings
- **witness_type** (*str*) – Specify witness type, default is legacy. Use 'segwit' for segregated witness.
- **multisig** (*bool*) – Key is part of a multisignature wallet?

**Return str** Base58 encoded WIF key

**class** bitcoinlib.keys.**Key**(*import_key=None*, *network=None*, *compressed=True*, *passphrase=''*, *is_private=None*)
Bases: object

Class to generate, import and convert public cryptographic key pairs used for bitcoin.

If no key is specified when creating class a cryptographically secure Private Key is generated using the os.urandom() function.

Initialize a Key object. Import key can be in WIF, bytes, hexstring, etc. If a private key is imported a public key will be derived. If a public is imported the private key data will be empty.

Both compressed and uncompressed key version is available, the Key.compressed boolean attribute tells if the original imported key was compressed or not.

**Parameters**

- **import_key** (*str, int, bytes, bytearray*) – If specified import given private or public key. If not specified a new private key is generated.
- **network** (*str, Network*) – Bitcoin, testnet, litecoin or other network
- **compressed** (*bool*) – Is key compressed or not, default is True
- **passphrase** (*str*) – Optional passphrase if imported key is password protected
- **is_private** (*bool*) – Specify if imported key is private or public. Default is None: derive from provided key

**Returns** Key object

**address**(*compressed=None*, *prefix=None*, *script_type=None*, *encoding=None*)
Get address derived from public key

**Parameters**

- **compressed** (*bool*) – Always return compressed address
- **prefix** (*str, bytes*) – Specify versionbyte prefix in hexstring or bytes. Normally doesn't need to be specified, method uses default prefix from network settings
- **script_type** (*str*) – Type of script, i.e. p2sh or p2pkh.
- **encoding** (*str*) – Address encoding. Default is base58 encoding, for segwit you can specify bech32 encoding

**Return str** Base58 encoded address

**address_obj**
Get address object property. Create standard address object if not defined already.

**Return Address**

**address_uncompressed**(*prefix=None*, *script_type=None*, *encoding=None*)
Get uncompressed address from public key

**Parameters**

- **prefix** (*str, bytes*) – Specify versionbyte prefix in hexstring or bytes. Normally doesn't need to be specified, method uses default prefix from network settings
- **script_type** (*str*) – Type of script, i.e. p2sh or p2pkh.
- **encoding** (*str*) – Address encoding. Default is base58 encoding, for segwit you can specify bech32 encoding

**Return str** Base58 encoded address

**as_dict**()
Get current Key class as dictionary. Byte values are represented by hexadecimal strings.

**Return collections.OrderedDict**

**as_json**()
Get current key as json formatted string

**Return str**

**bip38_encrypt**(*passphrase*)
BIP0038 non-ec-multiply encryption. Returns BIP0038 encrypted privkey. Based on code from https://github.com/nomorecoin/python-bip38-testing

**Parameters** **passphrase** (*str*) – Required passphrase for encryption

**Return str** BIP38 passphrase encrypted private key

**hash160**()
Get public key in RIPEMD-160 + SHA256 format

**Return bytes**

**info**()
Prints key information to standard output

**public**()
Get public version of current key. Removes all private information from current key

**Return Key** Public key

**public_point**()
Get public key point on Elliptic curve

**Return tuple** (x, y) point

**public_uncompressed**()
Get public key, uncompressed version

**Return str** Uncompressed public key hexstring

**wif**(*prefix=None*)
Get Private Key in Wallet Import Format, steps: # Convert to Binary and add 0x80 hex # Calculate Double SHA256 and add as checksum to end of key

**Parameters** **prefix** (*str, bytes*) – Specify versionbyte prefix in hexstring or bytes. Normally doesn't need to be specified, method uses default prefix from network settings

**Return str** Base58Check encoded Private Key WIF

bitcoinlib.keys.**addr_convert**(*addr*, *prefix*, *encoding=None*, *to_encoding=None*)
Convert base-58 encoded address to address with another prefix

**Parameters**

- **addr** (*str*) – Base58 address
- **prefix** (*str, bytes*) – New address prefix
- **encoding** (*str*) – Encoding of original address: base58 or bech32. Leave empty to extract from address
- **to_encoding** (*str*) – Encoding of converted address: base58 or bech32. Leave empty use same encoding as original address

**Return str** New converted address

bitcoinlib.keys.**check_network_and_key**(*key*, *network=None*, *kf_networks=None*, *default_network='bitcoin'*)
Check if given key corresponds with given network and return network if it does. If no network is specified this method tries to extract the network from the key. If no network can be extracted from the key the default network will be returned.

A BKeyError will be raised if key does not corresponds with network or if multiple network are found.

**Parameters**

- **key** (*str, int, bytes, bytearray*) – Key in any format recognized by get_key_format function
- **network** (*str*) – Optional network. Method raises BKeyError if keys belongs to another network
- **kf_networks** (*list*) – Optional list of networks which is returned by get_key_format. If left empty the get_key_format function will be called.
- **default_network** (*str*) – Specify different default network, leave empty for default (bitcoin)

**Return str** Network name

bitcoinlib.keys.**deserialize_address**(*address*, *encoding=None*, *network=None*)
Deserialize address. Calculate public key hash and try to determine script type and network.

The 'network' dictionary item with contain the network with highest priority if multiple networks are found. Same applies for the script type.

Specify the network argument if known to avoid unexpected results.

If more networks and or script types are found you can find these in the 'networks' field.

**Parameters**

- **address** (*str*) – A base58 or bech32 encoded address
- **encoding** (*str*) – Encoding scheme used for address encoding. Attempts to guess encoding if not specified.
- **network** (*str*) – Bitcoin, testnet, litecoin or other network

**Return dict** with information about this address

bitcoinlib.keys.**ec_point**(*p*)

Method for elliptic curve multiplication

**Parameters** **p** – A point on the elliptic curve

**Return Point** Point multiplied by generator G

bitcoinlib.keys.**get_key_format**(*key*, *is_private=None*)

Determines the type (private or public), format and network key.

This method does not validate if a key is valid.

**Parameters**

- **key** (*str, int, bytes, bytearray*) – Any private or public key
- **is_private** (*bool*) – Is key private or not?

**Return dict** Dictionary with format, network and is_private

bitcoinlib.keys.**path_expand**(*path*, *path_template=None*, *level_offset=None*, *account_id=0*, *cosigner_id=0*, *purpose=44*, *address_index=0*, *change=0*, *witness_type='legacy'*, *multisig=False*, *network='bitcoin'*)

Create key path. Specify part of key path and path settings

**Parameters**

- **path** (*list, str*) – Part of path, for example [0, 2] for change=0 and address_index=2
- **path_template** (*list*) – Template for path to create, default is BIP 44: ["m", "purpose'", "coin_type'", "account'", "change", "address_index"]
- **level_offset** (*int*) – Just create part of path. For example -2 means create path with the last 2 items (change, address_index) or 1 will return the master key 'm'
- **account_id** (*int*) – Account ID
- **cosigner_id** (*int*) – ID of cosigner
- **purpose** (*int*) – Purpose value
- **address_index** (*int*) – Index of key, normally provided to 'path' argument
- **change** (*int*) – Change key = 1 or normal = 0, normally provided to 'path' argument
- **witness_type** (*str*) – Witness type for paths with a script ID, specify 'p2sh-segwit' or 'segwit'
- **network** (*str*) – Network name. Leave empty for default network

**Return list**

### bitcoinlib.main module

bitcoinlib.main.**deprecated**(*func*)
This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used.

bitcoinlib.main.**get_encoding_from_witness**(*witness_type=None*)
Derive address encoding (base58 or bech32) from transaction witness type

**Parameters** **witness_type** (*str*) – Witness type: legacy, p2sh-segwit or segwit

**Return str**

bitcoinlib.main.**script_type_default**(*witness_type=None*, *multisig=False*, *locking_script=False*)
Determine default script type for provided witness type and key type combination used in this library.

**Parameters**

- **witness_type** (*str*) – Type of wallet: standard or segwit
- **multisig** (*bool*) – Multisig key or not, default is False
- **locking_script** (*bool*) – Limit search to locking_script. Specify False for locking scripts and True for unlocking scripts

**Return str** Default script type

### bitcoinlib.mnemonic module

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

Class to convert, generate and parse Mnemonic sentences

Implementation of BIP0039 for Mnemonics passphrases

Took some parts from Pavol Rusnak Trezors implementation, see https://github.com/trezor/python-mnemonic

Init Mnemonic class and read wordlist of specified language

**Parameters** **language** (*str*) – use specific wordlist, i.e. chinese, dutch (in development), english, french, italian, japanese or spanish. Leave empty for default 'english'

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

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

**Return str** Checksum of key in bits

**static detect_language**()
Detect language of given phrase

**Parameters** **words** (*str*) – List of space seperated words

**Return str** Language

**generate**(*strength=128*, *add_checksum=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.

**Parameters**

- **strength** (*int*) – 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
- **add_checksum** (*bool*) – Included a checksum? Default is True

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

**sanitize_mnemonic**(*words*)

Check and convert list of words to utf-8 encoding.

Raises an error if unrecognised word is found

**Parameters** **words** (*str*) – List of space seperated words

**Return str** Sanitized list of words

**to_entropy**(*words*, *includes_checksum=True*)

Convert Mnemonic words back to key data entrophy

**Parameters**

- **words** (*str*) – Mnemonic words as string of list of words
- **includes_checksum** (*bool*) – Boolean to specify if checksum is used. Default is True

**Return bytes** Entrophy seed

**to_mnemonic**(*data*, *add_checksum=True*)

Convert key data entropy to Mnemonic sentence

**Parameters**

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

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

**to_seed**(*words*, *password=''*)

Use Mnemonic words and password to create a PBKDF2 seed (Password-Based Key Derivation Function 2)

First use 'sanitize_mnemonic' to determine language and validate and check words

**Parameters**

- **words** (*str*) – Mnemonic passphrase as string with space seperated words
- **password** (*str*) – A password to protect key, leave empty to disable

**Return bytes** PBKDF2 seed

**word**(*index*)

Get word from wordlist

**Parameters** **index** (*int*) – word index ID

**Return str** A word from the dictionary

**wordlist**()

Get full selected wordlist. A wordlist is selected when initializing Mnemonic class

**Return list** Full list with 2048 words

### bitcoinlib.networks module

**class** bitcoinlib.networks.**Network**(*network_name='bitcoin'*)
Bases: object

Network class with all network definitions.

Prefixes for WIF, P2SH keys, HD public and private keys, addresses. A currency symbol and type, the denominator (such as satoshi) and a BIP0044 cointype.

**print_value**(*value*)
Return the value as string with currency symbol

Print value for 100000 satoshi as string in human readable format >>> Network('bitcoin').print_value(100000) '0.00100000 BTC'

**Parameters** **value** (*int, float*) – Value in smallest denominitor such as Satoshi

**Return str**

**wif_prefix**(*is_private=False*, *witness_type='legacy'*, *multisig=False*)

Get WIF prefix for this network and specifications in arguments

```
>>> Network('bitcoin').wif_prefix()
b'²
```

**' # xpub**

```
>>> Network('bitcoin').wif_prefix(is_private=True, witness_type='segwit',␣
↪multisig=True)
b'}ªz'      # Zprv
```

**param is_private** Private or public key, default is True

**type is_private** bool

**param witness_type** Legacy, segwit or p2sh-segwit

**type witness_type** str

**param multisig** Multisignature or single signature wallet. Default is not multisig

**type multisig** True

**return bytes**

**exception** bitcoinlib.networks.**NetworkError**(*msg=''*)
Bases: Exception

Network Exception class

bitcoinlib.networks.**network_by_value**(*field*, *value*)
Return all networks for field and (prefix) value.

Example, get available networks for WIF or adress prefix >>> network_by_value('prefix_wif', 'B0') ['litecoin', 'litecoin_legacy'] >>> network_by_value('prefix_address', '6f') ['testnet', 'litecoin_testnet']

This method does not work for HD prefixes, use 'wif_prefix_search' instead >>> network_by_value('prefix_address', '043587CF') []

**Parameters**

- **field** (*str*) – Prefix name from networks definitions (networks.json)
- **value** (*str, bytes*) – Value of network prefix

**Return list** Of network name strings

bitcoinlib.networks.**network_defined**(*network*)

Is network defined?

Networks of this library are defined in networks.json in the operating systems user path.

```
>>> network_defined('bitcoin')
True
>>> network_defined('ethereum')
False
```

**Parameters** **network** (*str*) – Network name

**Return bool**

bitcoinlib.networks.**network_values_for**(*field*, *output_as='default'*)

Return all prefixes mentioned field, i.e.: prefix_wif, prefix_address_p2sh, etc

```
>>> network_values_for('prefix_wif')
[b'', b'', b'ï', b'°', b'°', b'ï', b'Ì', b'ï']
>>> network_values_for('prefix_address_p2sh')
[b'', b'', b'Ä', b'2', b'', b':', b'', b'']
```

**Parameters**

- **field** (*str*) – Prefix name from networks definitions (networks.json)
- **output_as** (*str*) – Output as string or hexstring. Default is string or hexstring depending on field type.

**Return str**

bitcoinlib.networks.**wif_prefix_search**(*wif*, *witness_type=None*, *multisig=None*, *network=None*)

Extract network, script type and public/private information from HDKey WIF or WIF prefix.

Example, get bitcoin 'xprv' info: >>> wif_prefix_search('0488ADE4', network='bitcoin', multisig=False) [{'prefix': '0488ADE4', 'is_private': True, 'prefix_str': 'xprv', 'network': 'bitcoin', 'witness_type': 'legacy', 'multisig': False, 'script_type': 'p2pkh'}]

Or retreive info with full WIF string: >>> wif_prefix_search('xprv9wTYmMFdV23N21MM6dLNavSQV7Sj7meSPXx6AV5eTdq network='bitcoin', multisig=False) [{'prefix': '0488ADE4', 'is_private': True, 'prefix_str': 'xprv', 'network': 'bitcoin', 'witness_type': 'legacy', 'multisig': False, 'script_type': 'p2pkh'}]

Can return multiple items if no network is specified: >>> [nw['network'] for nw in wif_prefix_search('0488ADE4', multisig=True)] ['bitcoin', 'dash']

**Parameters**

- **wif** (*str, bytes*) – WIF string or prefix in bytes or hexadecimal string
- **witness_type** (*str*) – Limit search to specific witness type
- **multisig** (*bool*) – Limit search to multisig: false, true or None for both. Default is both
- **network** (*str*) – Limit search to specified network

**Return dict**

### bitcoinlib.transactions module

**class** bitcoinlib.transactions.**Input**(*prev_hash*, *output_n*, *keys=None*, *signatures=None*, *public_hash=b''*, *unlocking_script=b''*, *unlocking_script_unsigned=None*, *script_type=None*, *address=''*, *sequence=4294967295*, *compressed=None*, *sigs_required=None*, *sort=False*, *index_n=0*, *value=0*, *double_spend=False*, *locktime_cltv=None*, *locktime_csv=None*, *key_path=''*, *witness_type=None*, *encoding=None*, *network='bitcoin'*)

Bases: object

Transaction Input class, used by Transaction class

An Input contains a reference to an UTXO or Unspent Transaction Output (prev_hash + output_n). To spent the UTXO an unlocking script can be included to prove ownership.

Inputs are verified by the Transaction class.

Create a new transaction input

**Parameters**

- **prev_hash** (*bytes, hexstring*) – Transaction hash of the UTXO (previous output) which will be spent.
- **output_n** (*bytes, int*) – Output number in previous transaction.
- **keys** (*list (bytes, str, Key)*) – A list of Key objects or public / private key string in various formats. If no list is provided but a bytes or string variable, a list with one item will be created. Optional
- **signatures** (*bytes, str*) – Specify optional signatures
- **public_hash** (*bytes, str*) – Public key or script hash. Specify if key is not available
- **unlocking_script** (*bytes, hexstring*) – Unlocking script (scriptSig) to prove ownership. Optional
- **script_type** (*str*) – Type of unlocking script used, i.e. p2pkh or p2sh_multisig. Default is p2pkh
- **address** (*str, Address*) – Address string or object for input
- **sequence** (*bytes, int*) – Sequence part of input, you normally do not have to touch this
- **compressed** (*bool*) – Use compressed or uncompressed public keys. Default is compressed
- **sigs_required** (*int*) – Number of signatures required for a p2sh_multisig unlocking script
- **sort** (*boolean*) – Sort public keys according to BIP0045 standard. Default is False to avoid unexpected change of key order.
- **index_n** (*int*) – Index of input in transaction. Used by Transaction class.
- **value** (*int*) – Value of input in smallest denominator, i.e. sathosis
- **double_spend** (*bool*) – Is this input also spend in another transaction
- **locktime_cltv** (*int*) – Check Lock Time Verify value. Script level absolute time lock for this input

- **locktime_csv** (*int*) – Check Sequency Verify value.
- **key_path** (*str, list*) – Key path of input key as BIP32 string or list
- **witness_type** (*str*) – Specify witness/signature position: 'segwit' or 'legacy'. Determine from script, address or encoding if not specified.
- **encoding** (*str*) – Address encoding used. For example bech32/base32 or base58. Leave empty for default
- **network** (*str, Network*) – Network, leave empty for default

**as_dict**()

Get transaction input information in json format

**Return dict** Json with output_n, prev_hash, output_n, type, address, public_key, public_hash, unlocking_script and sequence

**sequence_timelock_blocks**(*blocks*)

**sequence_timelock_time**(*seconds*)

**update_scripts**(*hash_type=1*)

Method to update Input scripts.

Creates or updates unlocking script, witness script for segwit inputs, multisig redeemscripts and locktime scripts. This method is called when initializing a Input class or when signing an input.

**Parameters** **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL

**Return bool** Always returns True when method is completed

**class** bitcoinlib.transactions.**Output**(*value*, *address=''*, *public_hash=b''*, *public_key=b''*, *lock_script=b''*, *spent=False*, *output_n=0*, *script_type=None*, *encoding=None*, *network='bitcoin'*)

Bases: object

Transaction Output class, normally part of Transaction class.

Contains the amount and destination of a transaction.

Create a new transaction output

An transaction outputs locks the specified amount to a public key. Anyone with the private key can unlock this output.

The transaction output class contains an amount and the destination which can be provided either as address, public key, public key hash or a locking script. Only one needs to be provided as the they all can be derived from each other, but you can provide as much attributes as you know to improve speed.

**Parameters**

- **value** (*int*) – Amount of output in smallest denominator of currency, for example satoshi's for bitcoins
- **address** (*str, Address, HDKey*) – Destination address of output. Leave empty to derive from other attributes you provide. An instance of an Address or HDKey class is allowed as argument.
- **public_hash** (*bytes, str*) – Hash of public key or script
- **public_key** (*bytes, str*) – Destination public key
- **lock_script** (*bytes, str*) – Locking script of output. If not provided a default unlocking script will be provided with a public key hash.

- **spent** (*bool*) – Is output already spent? Default is False
- **output_n** (*int*) – Output index number, default is 0. Index number has to be unique per transaction and 0 for first output, 1 for second, etc
- **script_type** (*str*) – Script type of output (p2pkh, p2sh, segwit p2wpkh, etc). Extracted from lock_script if provided.
- **encoding** (*str*) – Address encoding used. For example bech32/base32 or base58. Leave empty to derive from address or default base58 encoding
- **network** (*str, Network*) – Network, leave empty for default

**as_dict**()

Get transaction output information in json format

**Return dict** Json with amount, locking script, public key, public key hash and address

**class** bitcoinlib.transactions.**Transaction**(*inputs=None*, *outputs=None*, *locktime=0*, *version=1*, *network='bitcoin'*, *fee=None*, *fee_per_kb=None*, *size=None*, *hash=''*, *date=None*, *confirmations=None*, *block_height=None*, *block_hash=None*, *input_total=0*, *output_total=0*, *rawtx=''*, *status='new'*, *coinbase=False*, *verified=False*, *witness_type='legacy'*, *flag=None*)

Bases: object

Transaction Class

Contains 1 or more Input class object with UTXO's to spent and 1 or more Output class objects with destinations. Besides the transaction class contains a locktime and version.

Inputs and outputs can be included when creating the transaction, or can be add later with add_input and add_output respectively.

A verify method is available to check if the transaction Inputs have valid unlocking scripts.

Each input in the transaction can be signed with the sign method provided a valid private key.

Create a new transaction class with provided inputs and outputs.

You can also create a empty transaction and add input and outputs later.

To verify and sign transactions all inputs and outputs need to be included in transaction. Any modification after signing makes the transaction invalid.

**Return type**

**Parameters**

- **inputs** (*list (Input)*) – Array of Input objects. Leave empty to add later
- **outputs** (*list (Output)*) – Array of Output object. Leave empty to add later
- **locktime** (*int*) – Transaction level locktime. Locks the transaction until a specified block (value from 1 to 5 million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transactions without locktime
- **version** (*bytes, int*) – Version rules. Defaults to 1 in bytes
- **network** (*str, Network*) – Network, leave empty for default network
- **fee** (*int*) – Fee in smallest denominator (ie Satoshi) for complete transaction

- **fee_per_kb** (*int*) – Fee in smallest denominator per kilobyte. Specify when exact transaction size is not known.

:param size; Transaction size in bytes :type size: int :param date: Confirmation date of transaction :type date: datetime.datetime :param confirmations: Number of confirmations :type confirmations: int :param block_height: Block number which includes transaction :type block_height: int :param block_hash: Hash of block for this transaction :type block_hash: str :param input_total: Total value of inputs :type input_total: int :param output_total: Total value of outputs :type output_total: int :param rawtx: Raw hexstring of complete transaction :type rawtx: str :param status: Transaction status, for example: 'new', 'incomplete', 'unconfirmed', 'confirmed' :type status: str :param coinbase: Coinbase transaction or not? :type coinbase: bool :param verified: Is transaction successfully verified? Updated when verified() method is called :type verified: bool :param witness_type: Specify witness/signature position: 'segwit' or 'legacy'. Determine from script, address or encoding if not specified. :type witness_type: str :param flag: Transaction flag to indicate version, for example for SegWit :type flag: bytes, str

**add_input**(*prev_hash*, *output_n*, *keys=None*, *signatures=None*, *public_hash=b''*, *unlocking_script=b''*, *unlocking_script_unsigned=None*, *script_type=None*, *address=''*, *sequence=4294967295*, *compressed=True*, *sigs_required=None*, *sort=False*, *index_n=None*, *value=None*, *double_spend=False*, *locktime_cltv=None*, *locktime_csv=None*, *key_path=''*, *witness_type=None*, *encoding=None*)

Add input to this transaction

Wrapper for append method of Input class.

**Parameters**

- **prev_hash** (*bytes, hexstring*) – Transaction hash of the UTXO (previous output) which will be spent.
- **output_n** (*bytes, int*) – Output number in previous transaction.
- **keys** (*bytes, str*) – Public keys can be provided to construct an Unlocking script. Optional
- **signatures** (*bytes, str*) – Add signatures to input if already known
- **public_hash** (*bytes, str*) – Specify public hash from key or redeemscript if key is not available
- **unlocking_script** (*bytes, hexstring*) – Unlocking script (scriptSig) to prove ownership. Optional
- **unlocking_script_unsigned** (*bytes, str*) – TODO: find better name...
- **script_type** (*str*) – Type of unlocking script used, i.e. p2pkh or p2sh_multisig. Default is p2pkh
- **address** (*str, Address*) – Specify address of input if known, default is to derive from key or scripts
- **sequence** (*int, bytes*) – Sequence part of input, you normally do not have to touch this
- **compressed** (*bool*) – Use compressed or uncompressed public keys. Default is compressed
- **sigs_required** – Number of signatures required for a p2sh_multisig unlocking script
- **sigs_required** – int
- **sort** (*boolean*) – Sort public keys according to BIP0045 standard. Default is False to avoid unexpected change of key order.

- **index_n** (*int*) – Index number of position in transaction, leave empty to add input to end of inputs list
- **value** (*int*) – Value of input
- **double_spend** (*bool*) – True if double spend is detected, depends on which service provider is selected
- **locktime_cltv** (*int*) – Check Lock Time Verify value. Script level absolute time lock for this input
- **locktime_csv** (*int*) – Check Sequency Verify value.
- **key_path** (*str, list*) – Key path of input key as BIP32 string or list
- **witness_type** (*str*) – Specify witness/signature position: 'segwit' or 'legacy'. Determine from script, address or encoding if not specified.
- **encoding** (*str*) – Address encoding used. For example bech32/base32 or base58. Leave empty to derive from script or script type

**Return int** Transaction index number (index_n)

**add_output**(*value*, *address=''*, *public_hash=b''*, *public_key=b''*, *lock_script=b''*, *spent=False*, *output_n=None*, *encoding=None*)

Add an output to this transaction

Wrapper for the append method of the Output class.

**Parameters**

- **value** (*int*) – Value of output in smallest denominator of currency, for example satoshi's for bitcoins
- **address** (*str, Address*) – Destination address of output. Leave empty to derive from other attributes you provide.
- **public_hash** (*bytes, str*) – Hash of public key or script
- **public_key** (*bytes, str*) – Destination public key
- **lock_script** (*bytes, str*) – Locking script of output. If not provided a default unlocking script will be provided with a public key hash.
- **spent** (*bool*) – Has output been spent in new transaction?
- **output_n** (*int*) – Index number of output in transaction
- **encoding** (*str*) – Address encoding used. For example bech32/base32 or base58. Leave empty for to derive from script or script type

**Return int** Transaction output number (output_n)

**as_dict**()

Return Json dictionary with transaction information: Inputs, outputs, version and locktime

**Return dict**

**as_json**()

Get current key as json formatted string

**Return str**

**calculate_fee**()

Get fee for this transaction in smallest denominator (i.e. Satoshi) based on its size and the transaction.fee_per_kb value

**Return int** Estimated transaction fee

**estimate_size**(*add_change_output=False*)
Get estimated vsize in for current transaction based on transaction type and number of inputs and outputs.

For old-style legacy transaction the vsize is the length of the transaction. In segwit transaction the witness data has less weigth. The formula used is: math.ceil(((est_size-witness_size) * 3 + est_size) / 4)

**Parameters** **add_change_output** (*bool*) – Assume an extra change output will be created but has not been created yet.

**Return int** Estimated transaction size

**static import_raw**(*network='bitcoin'*)
Import a raw transaction and create a Transaction object

Uses the _transaction_deserialize method to parse the raw transaction and then calls the init method of this transaction class to create the transaction object

**Parameters**

- **rawtx** (*bytes, str*) – Raw transaction string
- **network** (*str, Network*) – Network, leave empty for default

**Return Transaction**

**info**()
Prints transaction information to standard output

**raw**(*sign_id=None*, *hash_type=1*, *witness_type=None*)
Serialize raw transaction

Return transaction with signed inputs if signatures are available

**Parameters**

- **sign_id** (*int*) – Create raw transaction which can be signed by transaction with this input ID
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL
- **witness_type** (*str*) – Serialize transaction with other witness type then default. Use to create legacy raw transaction for segwit transaction to create transaction signature ID's

**Return bytes**

**raw_hex**(*sign_id=None*, *hash_type=1*, *witness_type=None*)
Wrapper for raw() method. Return current raw transaction hex

**Parameters**

- **sign_id** (*int*) – Create raw transaction which can be signed by transaction with this input ID
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL
- **witness_type** (*str*) – Serialize transaction with other witness type then default. Use to create legacy raw transaction for segwit transaction to create transaction signature ID's

**Return hexstring**

**sign**(*keys=None*, *tid=None*, *multisig_key_n=None*, *hash_type=1*)
Sign the transaction input with provided private key

**Parameters**

- **keys** (HDKey, Key, *bytes*, *list*) – A private key or list of private keys
- **tid** (*int*) – Index of transaction input
- **multisig_key_n** (*int*) – Index number of key for multisig input for segwit transactions. Leave empty if not known. If not specified all possibilities will be checked
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL

**Return None**

**signature**(*sign_id*, *hash_type=1*, *witness_type=None*)
Serializes transaction and calculates signature for Legacy or Segwit transactions

**Parameters**

- **sign_id** (*int*) – Index of input to sign
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL
- **witness_type** (*str*) – Legacy or Segwit witness type? Leave empty to use Transaction witness type

**Return bytes** Transaction signature

**signature_hash**(*sign_id*, *hash_type=1*, *witness_type=None*)
Double SHA256 Hash of Transaction signature

**Parameters**

- **sign_id** (*int*) – Index of input to sign
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL
- **witness_type** (*str*) – Legacy or Segwit witness type? Leave empty to use Transaction witness type

**Return bytes** Transaction signature hash

**signature_segwit**(*sign_id*, *hash_type=1*)
Serialize transaction signature for segregated witness transaction

**Parameters**

- **sign_id** (*int*) – Index of input to sign
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL

**Return bytes** Segwit transaction signature

**update_totals**()
Update input_total, output_total and fee according to inputs and outputs of this transaction

**Return int**

**verify**()
Verify all inputs of a transaction, check if signatures match public key.

Does not check if UTXO is valid or has already been spent

**Return bool** True if enough signatures provided and if all signatures are valid

**exception** bitcoinlib.transactions.**TransactionError**(*msg=''*)
Bases: Exception

Handle Transaction class Exceptions

bitcoinlib.transactions.**get_unlocking_script_type**(*locking_script_type*, *witness_type='legacy'*, *multisig=False*)

Specify locking script type and get corresponding script type for unlocking script.

**Parameters**

- **locking_script_type** (*str*) – Locking script type. I.e.: p2pkh, p2sh, p2wpkh, p2wsh
- **witness_type** (*str*) – Type of witness: legacy or segwit. Default is legacy
- **multisig** (*bool*) – Is multisig script or not? Default is False

**Return str** Unlocking script type such as sig_pubkey or p2sh_multisig

bitcoinlib.transactions.**script_add_locktime_cltv**(*locktime_cltv*, *script*)

bitcoinlib.transactions.**script_add_locktime_csv**(*locktime_csv*, *script*)

bitcoinlib.transactions.**script_deserialize**(*script*, *script_types=None*, *locking_script=None*, *size_bytes_check=True*)

Deserialize a script: determine type, number of signatures and script data.

**Parameters**

- **script** (*str, bytes, bytearray*) – Raw script
- **script_types** (*list*) – Limit script type determination to this list. Leave to default None to search in all script types.
- **locking_script** (*bool*) – Only deserialize locking scripts. Specify False to only deserialize for unlocking scripts. Default is None for both
- **size_bytes_check** (*bool*) – Check if script or signature starts with size bytes and remove size bytes before parsing. Default is True

**Return list** With this items: [script_type, data, number_of_sigs_n, number_of_sigs_m]

bitcoinlib.transactions.**script_deserialize_sigpk**(*script*)

Deserialize a unlocking script (scriptSig) with a signature and public key. The DER encoded signature is decoded to a normal signature with point x and y in 64 bytes total.

Returns signature and public key.

**Parameters** **script** (*bytes*) – A unlocking script

**Return tuple** Tuple with a signature and public key in bytes

bitcoinlib.transactions.**script_to_string**(*script*)

Convert script to human readable string format with OP-codes, signatures, keys, etc

Example: "OP_DUP OP_HASH160 af8e14a2cecd715c363b3a72b55b59a31e2acac9 OP_EQUALVERIFY OP_CHECKSIG"

**Parameters** **script** (*bytes, str*) – A locking or unlocking script

**Return str**

bitcoinlib.transactions.**serialize_multisig_redeemscript**(*key_list*, *n_required=None*, *compressed=True*)

Create a multisig redeemscript used in a p2sh.

Contains the number of signatures, followed by the list of public keys and the OP-code for the number of signatures required.

**Parameters**

- **key_list** (*Key, list*) – List of public keys
- **n_required** (*int*) – Number of required signatures
- **compressed** (*bool*) – Use compressed public keys?

**Return bytes** A multisig redeemscript

bitcoinlib.transactions.**verify_signature**(*transaction_to_sign*, *signature*, *public_key*)

Verify if signatures signs provided transaction hash and corresponds with public key

**Parameters**

- **transaction_to_sign** (*bytes, str*) – Raw transaction to sign
- **signature** (*bytes, str*) – A signature
- **public_key** (*bytes, str*) – The public key

**Return bool** Return True if verified

### bitcoinlib.wallets module

**class** bitcoinlib.wallets.**HDWallet**(*wallet*, *databasefile='/home/docs/.bitcoinlib/database/bitcoinlib.sqlite'*, *session=None*, *main_key_object=None*)

Bases: object

Class to create and manage keys Using the BIP0044 Hierarchical Deterministic wallet definitions, so you can use one Masterkey to generate as much child keys as you want in a structured manner.

You can import keys in many format such as WIF or extended WIF, bytes, hexstring, seeds or private key integer. For the Bitcoin network, Litecoin or any other network you define in the settings.

Easily send and receive transactions. Compose transactions automatically or select unspent outputs.

Each wallet name must be unique and can contain only one cointype and purpose, but practically unlimited accounts and addresses.

Open a wallet with given ID or name

**Parameters**

- **wallet** (*int, str*) – Wallet name or ID
- **databasefile** (*str*) – Location of database file. Leave empty to use default
- **session** (*sqlalchemy.orm.session.Session*) – Sqlalchemy session
- **main_key_object** (*HDKey*) – Pass main key object to save time

**account**(*account_id*)

Returns wallet key of specific BIP44 account.

Account keys have a BIP44 path depth of 3 and have the format m/purpose'/network'/account'

I.e: Use account(0).key().wif_public() to get wallet's public master key

**Parameters** **account_id** (*int*) – ID of account. Default is 0

**Return HDWalletKey**

**accounts**(*network='bitcoin'*)

Get list of accounts for this wallet

**Parameters** **network** (*str*) – Network name filter. Default filter is DEFAULT_NETWORK

**Return list** List of accounts as HDWalletKey objects

**addresslist**(*account_id=None*, *used=None*, *network=None*, *change=None*, *depth=None*, *key_id=None*)

Get list of addresses defined in current wallet

**Parameters**

- **account_id** (*int*) – Account ID
- **used** (*bool, None*) – Only return used or unused keys
- **network** (*str*) – Network name filter
- **change** – Only include change addresses or not. Default is None which returns both
- **depth** (*int*) – Filter by key depth. Default is None for standard key depth. Use -1 to show all keys
- **key_id** (*int*) – Key ID to get address of just 1 key

**Return list** List of address strings

**as_dict**()

Return wallet information in dictionary format

**Parameters** **detail** (*int*) – Level of detail to show, from 0 to 6. With 0 no details and 6 most details

**Return dict**

**as_json**()

Get current key as json formatted string

**Return str**

**balance**(*account_id=None*, *network=None*, *as_string=False*)

Get total of unspent outputs

**Parameters**

- **account_id** (*int*) – Account ID filter
- **network** (*str*) – Network name. Leave empty for default network
- **as_string** (*boolean*) – Set True to return a string in currency format. Default returns float.

**Return float, str** Key balance

**balance_update_from_serviceprovider**(*account_id=None*, *network=None*)

Update balance of currents account addresses using default Service objects getbalance method. Update total wallet balance in database.

Please Note: Does not update UTXO's or the balance per key! For this use the 'updatebalance' method instead

**Parameters**

- **account_id** (*int*) – Account ID. Leave empty for default account
- **network** (*str*) – Network name. Leave empty for default network

**Return int** Total balance

**classmethod create**(*name*, *keys=None*, *owner=''*, *network=None*, *account_id=0*, *purpose=0*, *scheme='bip32'*, *sort_keys=True*, *password=''*, *witness_type=None*, *encoding=None*, *multisig=None*, *sigs_required=None*, *cosigner_id=None*, *key_path=None*, *databasefile=None*)

Create HDWallet and insert in database. Generate masterkey or import key when specified.

When only a name is specified an legacy HDWallet with a single masterkey is created with standard p2wpkh scripts.

To create a multi signature wallet specify multiple keys (private or public) and provide the sigs_required argument if it different then len(keys)

To create a native segwit wallet use the option witness_type = 'segwit' and for old style addresses and p2sh embedded segwit script us 'ps2h-segwit' as witness_type.

Please mention account_id if you are using multiple accounts.

**Parameters**

- **name** (*str*) – Unique name of this Wallet
- **keys** (*str, bytes, int, bytearray*) – Masterkey to or list of keys to use for this wallet. Will be automatically created if not specified. One or more keys are obligatory for multisig wallets. Can contain all key formats accepted by the HDKey object, a HDKey object or BIP39 passphrase
- **owner** (*str*) – Wallet owner for your own reference
- **network** (*str*) – Network name, use default if not specified
- **account_id** (*int*) – Account ID, default is 0
- **purpose** (*int*) – BIP43 purpose field, will be derived from witness_type and multisig by default
- **scheme** (*str*) – Key structure type, i.e. BIP32 or single
- **sort_keys** (*bool*) – Sort keys according to BIP45 standard (used for multisig keys)
- **password** (*str*) – Password to protect passphrase, only used if a passphrase is supplied in the 'key' argument.
- **witness_type** (*str*) – Specify witness type, default is 'legacy'. Use 'segwit' for native segregated witness wallet, or 'p2sh-segwit' for legacy compatible wallets
- **encoding** (*str*) – Encoding used for address generation: base58 or bech32. Default is derive from wallet and/or witness type
- **multisig** (*bool*) – Multisig wallet or child of a multisig wallet, default is None / derive from number of keys.
- **sigs_required** (*int*) – Number of signatures required for validation if using a multisignature wallet. For example 2 for 2-of-3 multisignature. Default is all keys must signed
- **cosigner_id** (*int*) – Set this if wallet contains only public keys or if you would like to create keys for other cosigners.
- **key_path** – Key path for multisig wallet, use to create your own non-standard key path. Key path must

follow the following rules: * Path start with masterkey (m) and end with change / address_index * If accounts are used, the account level must be 3. I.e.: m/purpose/coin_type/account/ * All keys must be hardened, except for change, address_index or cosigner_id * Max length of path is 8 levels :type key_path: list, str :param databasefile: Location of database file. Leave empty to use default :type databasefile: str

**Return HDWallet**

**classmethod create_multisig**(*name*, *keys*, *sigs_required=None*, *owner=''*, *network=None*, *account_id=0*, *purpose=None*, *sort_keys=True*, *witness_type='legacy'*, *encoding=None*, *key_path=None*, *cosigner_id=None*, *databasefile=None*)

Create a multisig wallet with specified name and list of keys. The list of keys can contain 2 or more public or private keys. For every key a cosigner wallet will be created with a BIP44 key structure or a single key depending on the key_type.

**Parameters**

- **name** (*str*) – Unique name of this Wallet
- **keys** (*list*) – List of keys in HDKey format or any other format supported by HDKey class
- **sigs_required** (*int*) – Number of signatures required for validation. For example 2 for 2-of-3 multisignature. Default is all keys must signed
- **network** (*str*) – Network name, use default if not specified
- **account_id** (*int*) – Account ID, default is 0
- **purpose** (*int*) – BIP44 purpose field, default is 44
- **sort_keys** (*bool*) – Sort keys according to BIP45 standard (used for multisig keys)
- **witness_type** (*str*) – Specify wallet type, default is legacy. Use 'segwit' for segregated witness wallet.
- **encoding** (*str*) – Encoding used for address generation: base58 or bech32. Default is derive from wallet and/or witness type
- **key_path** – Key path for multisig wallet, use to create your own non-standard key path. Key path must

follow the following rules: * Path start with masterkey (m) and end with change / address_index * If accounts are used, the account level must be 3. I.e.: m/purpose/coin_type/account/ * All keys must be hardened, except for change, address_index or cosigner_id * Max length of path is 8 levels :type key_path: list, str :param cosigner_id: Set this if wallet contains only public keys or if you would like to create keys for other cosigners. :type cosigner_id: int :param databasefile: Location of database file. Leave empty to use default :type databasefile: str

**Return HDWallet**

**default_account_id**

**default_network_set**(*network*)

**get_key**(*account_id=None*, *network=None*, *cosigner_id=None*, *number_of_keys=1*, *change=0*)

Get a unused key or create a new one if there are no unused keys. Returns a key from this wallet which has no transactions linked to it.

**Parameters**

- **account_id** (*int*) – Account ID. Default is last used or created account ID.
- **network** (*str*) – Network name. Leave empty for default network
- **cosigner_id** (*int*) – Cosigner ID for key path
- **number_of_keys** (*int*) – Number of keys to return. Default is 1
- **change** (*int*) – Payment (0) or change key (1). Default is 0

**Return HDWalletKey**

**get_key_change** (*account_id=None*, *network=None*, *number_of_keys=1*)
Get a unused change key or create a new one if there are no unused keys. Wrapper for the get_key method

**Parameters**

- **account_id** (*int*) – Account ID. Default is last used or created account ID.
- **network** (*str*) – Network name. Leave empty for default network
- **number_of_keys** (*int*) – Number of keys to return. Default is 1

**Return HDWalletKey**

**import_key** (*key*, *account_id=0*, *name=''*, *network=None*, *purpose=44*, *key_type=None*)
Add new single key to wallet.

**Parameters**

- **key** (*str, bytes, int, bytearray, HDKey, Address*) – Key to import
- **account_id** (*int*) – Account ID. Default is last used or created account ID.
- **name** (*str*) – Specify name for key, leave empty for default
- **network** (*str*) – Network name, method will try to extract from key if not specified. Raises warning if network could not be detected
- **purpose** (*int*) – BIP definition used, default is BIP44
- **key_type** (*str*) – Key type of imported key, can be single (unrelated to wallet, bip32, bip44 or master for new or extra master key import. Default is 'single'

**Return HDWalletKey**

**import_master_key** (*hdkey*, *name='Masterkey (imported)'*)
Import (another) masterkey in this wallet

**Parameters**

- **hdkey** (*HDKey, str*) – Private key
- **name** (*str*) – Key name of masterkey

**Return HDKey** Main key as HDKey object

**info** (*detail=3*)
Prints wallet information to standard output

**Parameters** **detail** (*int*) – Level of detail to show. Specify a number between 0 and 5, with 0 low detail and 5 highest detail

**key** (*term*)
Return single key with given ID or name as HDWalletKey object

**Parameters** **term** (*int, str*) – Search term can be key ID, key address, key WIF or key name

**Return HDWalletKey** Single key as object

**key_add_private** (*wallet_key*, *private_key*)
Change public key in wallet to private key in current HDWallet object and in database

**Parameters**

- **wallet_key** (*HDWalletKey*) – Key object of wallet

- **private_key** (HDKey, *str*) – Private key wif or HDKey object

**Return HDWalletKey**

**key_for_path**(*path*, *level_offset=None*, *name=None*, *account_id=None*, *cosigner_id=None*, *address_index=0*, *change=0*, *network=None*, *recreate=False*)

Return key for specified path. Derive all wallet keys in path if they not already exists

```
>>> w = HDWallet.create('key_for_path_example')
>>> w.key_for_path([0, 0])
<HDWalletKey(key_id=750, name=address index 0, wif=xprv...4Vk2, path=m/44'/0'/
↪0'/0/0)>
```

```
>>> w.key_for_path([], level_offset=-2)
<HDWalletKey(key_id=748, name=account 0, wif=xprv...aMo, path=m/44'/0'/0')>
```

```
>>> w.key_for_path([], w.depth_public_master + 1)
<HDWalletKey(key_id=748, name=account 0, wif=xprv...aMo, path=m/44'/0'/0')>
```

Arguments provided in 'path' take precedence over other arguments. The address_index argument is ignored: >>> w.key_for_path([0, 10], address_index=1000) <HDWalletKey(key_id=751, name=address index 0, wif=xprv...k2Mo, path=m/44'/0'/0'/0/10)>

**Parameters**

- **path** (*list, str*) – Part of key path, i.e. [0, 0] for [change=0, address_index=0]
- **level_offset** (*int*) – Just create part of path, when creating keys. For example -2 means create path with the last 2 items (change, address_index) or 1 will return the master key 'm'
- **name** (*str*) – Specify key name for latest/highest key in structure
- **account_id** (*int*) – Account ID
- **cosigner_id** (*int*) – ID of cosigner
- **address_index** (*int*) – Index of key, normally provided to 'path' argument
- **change** (*int*) – Change key = 1 or normal = 0, normally provided to 'path' argument
- **network** (*str*) – Network name. Leave empty for default network
- **recreate** (*bool*) – Recreate key, even if already found in wallet. Can be used to update public key with private key info

**Return HDWalletKey**

**keys**(*account_id=None*, *name=None*, *key_id=None*, *change=None*, *depth=None*, *used=None*, *is_private=None*, *has_balance=None*, *is_active=True*, *network=None*, *as_dict=False*)

Search for keys in database. Include 0 or more of account_id, name, key_id, change and depth.

Returns a list of DbKey object or dictionary object if as_dict is True

**Parameters**

- **account_id** (*int*) – Search for account ID
- **name** (*str*) – Search for Name
- **key_id** (*int*) – Search for Key ID
- **change** (*int*) – Search for Change
- **depth** (*int*) – Only include keys with this depth

- **used** (*bool*) – Only return used or unused keys
- **is_private** (*bool*) – Only return private keys
- **has_balance** (*bool*) – Only include keys with a balance or without a balance, default is both
- **is_active** (*bool*) – Hide inactive keys. Only include active keys with either a balance or which are unused, default is True
- **network** (*str*) – Network name filter
- **as_dict** – Return keys as dictionary objects. Default is False: DbKey objects

**Return list** List of Keys

**keys_accounts**(*account_id=None*, *network='bitcoin'*, *as_dict=False*)

Get Database records of account key(s) with for current wallet. Wrapper for the keys() method.

Returns nothing if no account keys are available for instance in multisig or single account wallets. In this case use accounts() method instead.

**Parameters**

- **account_id** (*int*) – Search for Account ID
- **network** (*str*) – Network name filter
- **as_dict** (*bool*) – Return as dictionary or DbKey object. Default is False: DbKey objects

**Return list** DbKey or dictionaries

**keys_address_change**(*account_id=None*, *used=None*, *network=None*, *as_dict=False*)

Get payment addresses (change=1) of specified account_id for current wallet. Wrapper for the keys() methods.

**Parameters**

- **account_id** (*int*) – Account ID
- **used** (*bool*) – Only return used or unused keys
- **network** (*str*) – Network name filter
- **as_dict** (*bool*) – Return as dictionary or DbKey object. Default is False: DbKey objects

**Return list** DbKey or dictionaries

**keys_address_payment**(*account_id=None*, *used=None*, *network=None*, *as_dict=False*)

Get payment addresses (change=0) of specified account_id for current wallet. Wrapper for the keys() methods.

**Parameters**

- **account_id** (*int*) – Account ID
- **used** (*bool*) – Only return used or unused keys
- **network** (*str*) – Network name filter
- **as_dict** (*bool*) – Return as dictionary or DbKey object. Default is False: DbKey objects

**Return list** DbKey or dictionaries

**keys_addresses**(*account_id=None*, *used=None*, *network=None*, *depth=None*, *as_dict=False*)
Get address-keys of specified account_id for current wallet. Wrapper for the keys() methods.

**Parameters**

- **account_id** (*int*) – Account ID
- **used** (*bool*) – Only return used or unused keys
- **network** (*str*) – Network name filter
- **depth** (*int*) – Filter by key depth. Default for BIP44 and multisig is 5
- **as_dict** (*bool*) – Return as dictionary or DbKey object. Default is False: DbKey objects

**Return list** DbKey or dictionaries

**keys_networks**(*used=None*, *as_dict=False*)
Get keys of defined networks for this wallet. Wrapper for the keys() method

**Parameters**

- **used** (*bool*) – Only return used or unused keys
- **as_dict** (*bool*) – Return as dictionary or DbKey object. Default is False: DbKey objects

**Return list** DbKey or dictionaries

**name**
Get wallet name

**Return str**

**network_list**(*field='name'*)
Wrapper for networks methods, returns a flat list with currently used networks for this wallet.

**Return list of str**

**networks**(*as_dict=False*)
Get list of networks used by this wallet

**Parameters** **as_dict** (*bool*) – Return as dictionary or as Network objects, default is Network objects

**Return list of (Network, dict)**

**new_account**(*name=''*, *account_id=None*, *network=None*)
Create a new account with a childkey for payments and 1 for change.

An account key can only be created if wallet contains a masterkey.

**Parameters**

- **name** (*str*) – Account Name. If not specified 'Account #" with the account_id will be used
- **account_id** (*int*) – Account ID. Default is last accounts ID + 1
- **network** (*str*) – Network name. Leave empty for default network

**Return HDWalletKey**

**new_key**(*name=''*, *account_id=None*, *change=0*, *cosigner_id=None*, *network=None*)
Create a new HD Key derived from this wallet's masterkey. An account will be created for this wallet with index 0 if there is no account defined yet.

**Parameters**

- **name** (*str*) – Key name. Does not have to be unique but if you use it at reference you might chooce to enforce this. If not specified 'Key #' with an unique sequence number will be used
- **account_id** (*int*) – Account ID. Default is last used or created account ID.
- **change** (*int*) – Change (1) or payments (0). Default is 0
- **cosigner_id** (*int*) – Cosigner ID for key path
- **network** (*str*) – Network name. Leave empty for default network

**Return HDWalletKey**

**new_key_change**(*name=''*, *account_id=None*, *network=None*)

Create new key to receive change for a transaction. Calls new_key method with change=1.

**Parameters**

- **name** (*str*) – Key name. Default name is 'Change #' with an address index
- **account_id** (*int*) – Account ID. Default is last used or created account ID.
- **network** (*str*) – Network name. Leave empty for default network

**Return HDWalletKey**

**owner**

Get wallet Owner

**Return str**

**path_expand**(*path*, *level_offset=None*, *account_id=None*, *cosigner_id=0*, *address_index=None*, *change=0*, *network='bitcoin'*)

Create key path. Specify part of key path and path settings

**Parameters**

- **path** (*list, str*) – Part of path, for example [0, 2] for change=0 and address_index=2
- **level_offset** (*int*) – Just create part of path. For example -2 means create path with the last 2 items (change, address_index) or 1 will return the master key 'm'
- **account_id** (*int*) – Account ID
- **cosigner_id** (*int*) – ID of cosigner
- **address_index** (*int*) – Index of key, normally provided to 'path' argument
- **change** (*int*) – Change key = 1 or normal = 0, normally provided to 'path' argument
- **network** (*str*) – Network name. Leave empty for default network

**Return list**

**public_master**(*account_id=None*, *name=None*, *network=None*)

Return public master key(s) for this wallet. Use to import in other wallets to sign transactions or create keys.

For a multisig wallet all public master keys are return as list.

Returns private key information if available.

**Parameters**

- **account_id** (*int*) – Account ID of key to export

- **name** (*str*) – Optional name for account key
- **network** (*str*) – Network name. Leave empty for default network

**Return list of HDWalletKey, HDWalletKey**

**scan**(*scan_gap_limit=3*, *account_id=None*, *change=None*, *network=None*, *_keys_ignore=None*, *_recursion_depth=0*)

Generate new keys for this wallet and scan for UTXO's.

**Parameters**

- **scan_gap_limit** (*int*) – Amount of new keys and change keys (addresses) created for this wallet
- **account_id** (*int*) – Account ID. Default is last used or created account ID.
- **change** – Filter by change addresses. Set to True to include only change addresses, False to only include regular addresses. None (default) to disable filter and include both
- **network** (*str*) – Network name. Leave empty for default network

**Returns**

**select_inputs**(*amount*, *variance=None*, *account_id=None*, *network=None*, *min_confirms=0*, *max_utxos=None*, *return_input_obj=True*)

Select available inputs for given amount

**Parameters**

- **amount** (*int*) – Total value of inputs to select
- **variance** (*int*) – Allowed difference in total input value. Default is dust amount of selected network.
- **account_id** (*int*) – Account ID
- **network** (*str*) – Network name. Leave empty for default network
- **min_confirms** (*int*) – Minimal confirmation needed for an UTXO before it will included in inputs. Default is 0 confirmations. Option is ignored if input_arr is provided.
- **max_utxos** (*int*) – Maximum number of UTXO's to use. Set to 1 for optimal privacy. Default is None: No maximum
- **return_input_obj** (*bool*) – Return inputs as Input class object. Default is True

**Return list of DbTransactionOutput, Input** List of previous outputs

**send**(*output_arr*, *input_arr=None*, *account_id=None*, *network=None*, *fee=None*, *min_confirms=0*, *priv_keys=None*, *max_utxos=None*, *locktime=0*, *offline=False*)

Create new transaction with specified outputs and push it to the network. Inputs can be specified but if not provided they will be selected from wallets utxo's. Output array is a list of 1 or more addresses and amounts.

**Parameters**

- **output_arr** (*list*) – List of output tuples with address and amount. Must contain at least one item. Example: [('mxdLD8SAGS9fe2EeCXALDHcdTTbppMHp8N', 5000000)]. Address can be an address string, Address object, HDKey object or HDWalletKey object
- **input_arr** (*list*) – List of inputs tuples with reference to a UTXO, a wallet key and value. The format is [(tx_hash, output_n, key_id, value)]
- **account_id** (*int*) – Account ID

- **network** (*str*) – Network name. Leave empty for default network
- **fee** (*int*) – Set fee manually, leave empty to calculate fees automatically. Set fees in smallest currency denominator, for example satoshi's if you are using bitcoins
- **min_confirms** (*int*) – Minimal confirmation needed for an UTXO before it will included in inputs. Default is 0. Option is ignored if input_arr is provided.
- **priv_keys** (HDKey, *list*) – Specify extra private key if not available in this wallet
- **max_utxos** (*int*) – Maximum number of UTXO's to use. Set to 1 for optimal privacy. Default is None: No maximum
- **locktime** (*int*) – Transaction level locktime. Locks the transaction until a specified block (value from 1 to 5 million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transactions without locktime
- **offline** (*bool*) – Just return the transaction object and do not send it when offline = True. Default is False

**Return HDWalletTransaction**

**send_to**(*to_address*, *amount*, *account_id=None*, *network=None*, *fee=None*, *min_confirms=0*, *priv_keys=None*, *locktime=0*, *offline=False*)

Create transaction and send it with default Service objects sendrawtransaction method

**Parameters**

- **to_address** (*str*, Address, HDKey, HDWalletKey) – Single output address as string Address object, HDKey object or HDWalletKey object
- **amount** (*int*) – Output is smallest denominator for this network (ie: Satoshi's for Bitcoin)
- **account_id** (*int*) – Account ID, default is last used
- **network** (*str*) – Network name. Leave empty for default network
- **fee** (*int*) – Fee to use for this transaction. Leave empty to automatically estimate.
- **min_confirms** (*int*) – Minimal confirmation needed for an UTXO before it will included in inputs. Default is 0. Option is ignored if input_arr is provided.
- **priv_keys** (HDKey, *list*) – Specify extra private key if not available in this wallet
- **locktime** (*int*) – Transaction level locktime. Locks the transaction until a specified block (value from 1 to 5 million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transactions without locktime
- **offline** (*bool*) – Just return the transaction object and do not send it when offline = True. Default is False

**Return HDWalletTransaction**

**sweep**(*to_address*, *account_id=None*, *input_key_id=None*, *network=None*, *max_utxos=999*, *min_confirms=0*, *fee_per_kb=None*, *fee=None*, *locktime=0*, *offline=False*)

Sweep all unspent transaction outputs (UTXO's) and send them to one output address. Wrapper for the send method.

**Parameters**

- **to_address** (*str*) – Single output address
- **account_id** (*int*) – Wallet's account ID
- **input_key_id** (*int*) – Limit sweep to UTXO's with this key_id

- **network** (*str*) – Network name. Leave empty for default network
- **max_utxos** (*int*) – Limit maximum number of outputs to use. Default is 999
- **min_confirms** (*int*) – Minimal confirmations needed to include utxo
- **fee_per_kb** (*int*) – Fee per kilobyte transaction size, leave empty to get estimated fee costs from Service provider. This option is ignored when the 'fee' option is specified
- **fee** (*int*) – Total transaction fee in smallest denominator (i.e. satoshis). Leave empty to get estimated fee from service providers.
- **locktime** (*int*) – Transaction level locktime. Locks the transaction until a specified block (value from 1 to 5 million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transactions without locktime
- **offline** (*bool*) – Just return the transaction object and do not send it when offline = True. Default is False

**Return HDWalletTransaction**

**transaction_create**(*output_arr*, *input_arr=None*, *account_id=None*, *network=None*, *fee=None*, *min_confirms=0*, *max_utxos=None*, *locktime=0*)

Create new transaction with specified outputs. Inputs can be specified but if not provided they will be selected from wallets utxo's. Output array is a list of 1 or more addresses and amounts.

**Parameters**

- **output_arr** (*list*) – List of output tuples with address and amount. Must contain at least one item. Example: [('mxdLD8SAGS9fe2EeCXALDHcdTTbppMHp8N', 5000000)]
- **input_arr** (*list*) – List of inputs tuples with reference to a UTXO, a wallet key and value. The format is [(tx_hash, output_n, key_ids, value, signatures, unlocking_script, address)]
- **account_id** (*int*) – Account ID
- **network** (*str*) – Network name. Leave empty for default network
- **fee** (*int*) – Set fee manually, leave empty to calculate fees automatically. Set fees in smallest currency denominator, for example satoshi's if you are using bitcoins
- **min_confirms** (*int*) – Minimal confirmation needed for an UTXO before it will included in inputs. Default is 0 confirmations. Option is ignored if input_arr is provided.
- **max_utxos** (*int*) – Maximum number of UTXO's to use. Set to 1 for optimal privacy. Default is None: No maximum
- **locktime** (*int*) – Transaction level locktime. Locks the transaction until a specified block (value from 1 to 5 million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transactions without locktime

**Return HDWalletTransaction** object

**transaction_import**(*t*)

Import a Transaction into this wallet. Link inputs to wallet keys if possible and return HDWalletTransaction object. Only imports Transaction objects or dictionaries, use transaction_import_raw method to import a raw transaction.

**Parameters** **t** (*Transaction, dict*) – A Transaction object or dictionary

**Return HDWalletTransaction**

**transaction_import_raw**(*raw_tx*, *network=None*)

Import a raw transaction. Link inputs to wallet keys if possible and return HDWalletTransaction object

**Parameters**

- **raw_tx** (*str, bytes*) – Raw transaction
- **network** (*str*) – Network name. Leave empty for default network

**Return HDWalletTransaction**

**transactions**(*account_id=None*, *network=None*, *include_new=False*, *key_id=None*)

**Parameters**

- **account_id** (*int*) – Filter by Account ID. Leave empty for default account_id
- **network** (*str*) – Filter by network name. Leave empty for default network
- **include_new** (*bool*) – Also include new and incomplete transactions in list. Default is False
- **key_id** (*int*) – Filter by key ID

**Return list** List of transactions as dictionary

**transactions_update**(*account_id=None*, *used=None*, *network=None*, *key_id=None*, *depth=None*, *change=None*)

Update wallets transaction from service providers. Get all transactions for known keys in this wallet. The balances and unspent outputs (UTXO's) are updated as well, but for large wallets use the utxo_update method if possible.

**Parameters**

- **account_id** (*int*) – Account ID
- **used** (*bool, None*) – Only update used or unused keys, specify None to update both. Default is None
- **network** (*str*) – Network name. Leave empty for default network
- **key_id** (*int*) – Key ID to just update 1 key
- **depth** (*int*) – Only update keys with this depth, default is depth 5 according to BIP0048 standard. Set depth to None to update all keys of this wallet.
- **change** (*int*) – Only update change or normal keys, default is both (None)

**Return bool** True if all transactions are updated

**transactions_update_by_txids**(*txids*)

**utxo_add**(*address*, *value*, *tx_hash*, *output_n*, *confirmations=0*, *script=''*)

Add a single UTXO to the wallet database. To update all utxo's use utxos_update method.

Use this method for testing, offline wallets or if you wish to override standard method of retreiving UTXO's

This method does not check if UTXO exists or is still spendable.

[{ 'address': 'n2S9Czehjvdmpwd2YqekxuUC1Tz5ZdK3YN', 'script': '', 'confirmations': 10, 'output_n': 1, 'tx_hash': '9df91f89a3eb4259ce04af66ad4caf3c9a297feea5e0b3bc506898b6728c5003', 'value': 8970937

}]

**Parameters**

- **address** (*str*) – Address of Unspent Output. Address should be available in wallet

- **value** (*int*) – Value of output in sathosis or smallest denominator for type of currency
- **tx_hash** (*str*) – Transaction hash or previous output as hex-string
- **output_n** (*int*) – Output number of previous transaction output
- **confirmations** (*int*) – Number of confirmations. Default is 0, unconfirmed
- **script** (*str*) – Locking script of previous output as hex-string

**Return int** Number of new UTXO's added, so 1 if successful

**utxos**(*account_id=None*, *network=None*, *min_confirms=0*, *key_id=None*)

Get UTXO's (Unspent Outputs) from database. Use utxos_update method first for updated values

**Parameters**

- **account_id** (*int*) – Account ID
- **network** (*str*) – Network name. Leave empty for default network
- **min_confirms** (*int*) – Minimal confirmation needed to include in output list
- **key_id** (*int*) – Key ID to just get 1 key

**Return list** List of transactions

**utxos_update**(*account_id=None*, *used=None*, *networks=None*, *key_id=None*, *depth=None*, *change=None*, *utxos=None*, *update_balance=True*)

Update UTXO's (Unspent Outputs) in database of given account using the default Service object.

Delete old UTXO's which are spent and append new UTXO's to database.

For usage on an offline PC, you can import utxos with the utxos parameter as a list of dictionaries: [{

'address': 'n2S9CzehjvdmpwdYqekxuUC1Tz5ZdK3YN', 'script': '', 'confirmations': 10, 'output_n': 1, 'tx_hash': '9df91f89a3eb4259ce04af66ad4caf3c9a297feea5e0b3bc506898b6728c5003', 'value': 8970937

}]

**Parameters**

- **account_id** (*int*) – Account ID
- **used** (*bool*) – Only check for UTXO for used or unused keys. Default is both
- **networks** (*str, list*) – Network name filter as string or list of strings. Leave empty to update all used networks in wallet
- **key_id** (*int*) – Key ID to just update 1 key
- **depth** (*int*) – Only update keys with this depth, default is depth 5 according to BIP0048 standard. Set depth to None to update all keys of this wallet.
- **change** (*int*) – Only update change or normal keys, default is both (None)
- **utxos** (*list*) – List of unspent outputs in dictionary format specified in this method DOC header
- **update_balance** (*bool*) – Option to disable balance update after fetching UTXO's, used when utxos_update method is called several times in a row. Default is True

**Return int** Number of new UTXO's added

**wif**(*is_private=False*, *account_id=0*)

Return Wallet Import Format string for master private or public key which can be used to import key and recreate wallet in other software.

A list of keys will be exported for a multisig wallet.

**Parameters**

- **is_private** (*Public or private, default is True*) – Export public or private key
- **account_id** (*bool*) – Account ID of key to export

**Return list, str**

**class** bitcoinlib.wallets.**HDWalletKey**(*key_id*, *session*, *hdkey_object=None*)

Bases: object

Normally only used as attribute of HDWallet class. Contains HDKey class, and adds extra information such as key ID, name, path and balance.

All HDWalletKey are stored in a database

Initialize HDWalletKey with specified ID, get information from database.

**Parameters**

- **key_id** (*int*) – ID of key as mentioned in database
- **session** (*sqlalchemy.orm.session.Session*) – Required Sqlalchemy Session object
- **hdkey_object** (HDKey) – Optional HDKey object. Specify HDKey object if available for performance

**as_dict**()

Return current key information as dictionary

**balance**(*fmt=''*)

Get total value of unspent outputs

**Parameters** **fmt** (*str*) – Specify 'string' to return a string in currency format

**Return float, str** Key balance

**static from_key**(*wallet_id*, *session*, *key=''*, *account_id=0*, *network=None*, *change=0*, *purpose=44*, *parent_id=0*, *path='m'*, *key_type=None*, *encoding=None*, *witness_type='legacy'*, *multisig=False*, *cosigner_id=None*)

Create HDWalletKey from a HDKey object or key

**Parameters**

- **name** (*str*) – New key name
- **wallet_id** (*int*) – ID of wallet where to store key
- **session** (*sqlalchemy.orm.session.Session*) – Required Sqlalchemy Session object
- **key** (*str, int, byte, bytearray,* HDKey) – Optional key in any format accepted by the HDKey class
- **account_id** (*int*) – Account ID for specified key, default is 0
- **network** (*str*) – Network of specified key
- **change** (*int*) – Use 0 for normal key, and 1 for change key (for returned payments)
- **purpose** (*int*) – BIP0044 purpose field, default is 44
- **parent_id** (*int*) – Key ID of parent, default is 0 (no parent)

- **path** (*str*) – BIP0044 path of given key, default is 'm' (masterkey)
- **key_type** (*str*) – Type of key, single or BIP44 type
- **encoding** (*str*) – Encoding used for address, i.e.: base58 or bech32. Default is base58
- **witness_type** (*str*) – Witness type used when creating transaction script: legacy, p2sh-segwit or segwit.
- **multisig** (*bool*) – Specify if key is part of multisig wallet, used for create keys and key representations such as WIF and addreses
- **cosigner_id** (*int*) – Set this if you would like to create keys for other cosigners.

**Return HDWalletKey** HDWalletKey object

**key**()
Get HDKey object for current HDWalletKey

**Return HDKey**

**name**
Return name of wallet

**Return str**

**class** bitcoinlib.wallets.**HDWalletTransaction**(*hdwallet*, **args*, ***kwargs*)
Bases: *bitcoinlib.transactions.Transaction*

Normally only used as attribute of HDWallet class. Child of Transaction object with extra reference to wallet and database object.

All HDWalletTransaction items are stored in a database

Initialize HDWalletTransaction object with reference to a HDWallet object

**Parameters**

- **hdwallet** – HDWallet object, wallet name or ID
- **args** (*args*) – Arguments for HDWallet parent class
- **kwargs** (*kwargs*) – Keyword arguments for HDWallet parent class

**classmethod from_transaction**(*hdwallet*, *t*)
Create HDWalletTransaction object from Transaction object

**Parameters**

- **hdwallet** (*HDwallet, str, int*) – HDWallet object, wallet name or ID
- **t** (Transaction) – Specify Transaction object

**Return HDWalletClass**

**info**()
Print Wallet transaction information to standard output. Include send information.

**save**()
Save this transaction to database

**Return int** Transaction ID

**send**(*offline=False*)
Verify and push transaction to network. Update UTXO's in database after successfull send

**Parameters** **offline** (*bool*) – Just return the transaction object and do not send it when offline = True. Default is False

**Return None**

**sign**(*keys=None*, *index_n=0*, *multisig_key_n=None*, *hash_type=1*)
Sign this transaction. Use existing keys from wallet or use keys argument for extra keys.

**Parameters**

- **keys** (HDKey, *str*) – Extra private keys to sign the transaction
- **index_n** (*int*) – Transaction index_n to sign
- **multisig_key_n** (*int*) – Index number of key for multisig input for segwit transactions. Leave empty if not known. If not specified all possibilities will be checked
- **hash_type** (*int*) – Hashtype to use, default is SIGHASH_ALL

**Return None**

**exception** bitcoinlib.wallets.**WalletError**(*msg=''*)
Bases: Exception

Handle Wallet class Exceptions

bitcoinlib.wallets.**normalize_path**(*path*)
Normalize BIP0044 key path for HD keys. Using single quotes for hardened keys

**Parameters** **path** (*str*) – BIP0044 key path

**Return str** Normalized BIP0044 key path with single quotes

bitcoinlib.wallets.**parse_bip44_path**(*path*)
Assumes a correct BIP0044 path and returns a dictionary with path items. See Bitcoin improvement proposals BIP0043 and BIP0044.

Specify path in this format: m / purpose' / cointype' / account' / change / address_index. Path length must be between 1 and 6 (Depth between 0 and 5)

**Parameters** **path** (*str*) – BIP0044 path as string, with backslash (/) seperator.

**Return dict** Dictionary with path items: is_private, purpose, cointype, account, change and address_index

bitcoinlib.wallets.**wallet_create_or_open**(*name*, *keys=''*, *owner=''*, *network=None*, *account_id=0*, *purpose=None*, *scheme='bip32'*, *sort_keys=True*, *password=''*, *witness_type='legacy'*, *encoding=None*, *multisig=None*, *sigs_required=None*, *cosigner_id=None*, *key_path=None*, *database-file='/home/docs/.bitcoinlib/database/bitcoinlib.sqlite'*)
Create a wallet with specified options if it doesn't exist, otherwise just open

See Wallets class create method for option documentation

bitcoinlib.wallets.**wallet_create_or_open_multisig**(*name*, *keys*, *sigs_required=None*, *owner=''*, *network=None*, *account_id=0*, *purpose=None*, *sort_keys=True*, *witness_type='legacy'*, *encoding=None*, *cosigner_id=None*, *key_path=None*, *database-file='/home/docs/.bitcoinlib/database/bitcoinlib.sqlite'*)
Deprecated since version 0.4.5, use wallet_create_or_open instead

Create a wallet with specified options if it doesn't exist, otherwise just open

See Wallets class create method for option documentation

bitcoinlib.wallets.**wallet_delete**(*wallet*, *databasefile='/home/docs/.bitcoinlib/database/bitcoinlib.sqlite'*, *force=False*)

Delete wallet and associated keys and transactions from the database. If wallet has unspent outputs it raises a WalletError exception unless 'force=True' is specified

**Parameters**

- **wallet** (*int, str*) – Wallet ID as integer or Wallet Name as string
- **databasefile** (*str*) – Location of Sqlite database. Leave empty to use default
- **force** (*bool*) – If set to True wallet will be deleted even if unspent outputs are found. Default is False

**Return int** Number of rows deleted, so 1 if succesfull

bitcoinlib.wallets.**wallet_delete_if_exists**(*wallet*, *databasefile='/home/docs/.bitcoinlib/database/bitcoinlib.sqlite'*, *force=False*)

Delete wallet and associated keys from the database. If wallet has unspent outputs it raises a WalletError exception unless 'force=True' is specified. If wallet wallet does not exist return False

**Parameters**

- **wallet** (*int, str*) – Wallet ID as integer or Wallet Name as string
- **databasefile** (*str*) – Location of Sqlite database. Leave empty to use default
- **force** (*bool*) – If set to True wallet will be deleted even if unspent outputs are found. Default is False

**Return int** Number of rows deleted, so 1 if succesfull

bitcoinlib.wallets.**wallet_empty**(*wallet*, *databasefile='/home/docs/.bitcoinlib/database/bitcoinlib.sqlite'*)

Remove all generated keys and transactions from wallet. Does not delete the wallet itself or the masterkey, so everything can be recreated.

**Parameters**

- **wallet** (*int, str*) – Wallet ID as integer or Wallet Name as string
- **databasefile** (*str*) – Location of Sqlite database. Leave empty to use default

**Return bool** True if succesfull

bitcoinlib.wallets.**wallet_exists**(*wallet*, *databasefile='/home/docs/.bitcoinlib/database/bitcoinlib.sqlite'*)

Check if Wallets is defined in database

**Parameters**

- **wallet** (*int, str*) – Wallet ID as integer or Wallet Name as string
- **databasefile** (*str*) – Location of Sqlite database. Leave empty to use default

**Return bool** True if wallet exists otherwise False

bitcoinlib.wallets.**wallets_list**(*databasefile='/home/docs/.bitcoinlib/database/bitcoinlib.sqlite'*)

List Wallets from database

**Parameters** **databasefile** (*str*) – Location of Sqlite database. Leave empty to use default

**Return dict** Dictionary of wallets defined in database

### Module contents

bitcoinlib.**tools**
Used by autodoc_mock_imports.

## 7.6 Script types

This is an overview script types used in transaction Input and Outputs.

They are defined in main.py

### 7.6.1 Locking scripts

Scripts lock funds in transaction outputs (UTXO's). Also called ScriptSig.

| Lock Script | Script to Unlock | Encoding | Key type / Script | Prefix BTC |
|---|---|---|---|---|
| p2pkh | Pay to Public Key Hash | base58 | Public key hash | 1 |
| p2sh | Pay to Script Hash | base58 | Redeemscript hash | 3 |
| p2wpkh | Pay to Wallet Pub Key Hash | bech32 | Public key hash | bc |
| p2wsh | Pay to Wallet Script Hash | bech32 | Redeemscript hash | bc |
| multisig | Multisig Script | base58 | Multisig script | 3 |
| pubkey | Public Key (obsolete) | base58 | Public Key | 1 |
| nulldata | Nulldata | n/a | OP_RETURN script | n/a |

### 7.6.2 Unlocking scripts

Scripts used in transaction inputs to unlock funds from previous outputs. Also called ScriptPubKey.

| Locking sc. | Name | Unlocks | Key type / Script |
|---|---|---|---|
| sig_pubkey | Signature, Public Key | p2pkh | Sign. + Public key |
| p2sh_multisig | Pay to Script Hash | p2sh, multisig | Multisig + Redeemscript |
| p2sh_p2wpkh | Pay to Wallet Pub Key Hash | p2wpkh | PK Hash + Redeemscript |
| p2sh_p2wsh | Multisig script | p2wsh | Redeemscript |
| signature | Sig for public key (old) | pubkey | Signature |

### 7.6.3 Bitcoinlib script support

The 'pubkey' lockscript and 'signature' unlocking script are ancient and not supported by BitcoinLib at the moment.

Using different encodings for addresses then the one listed in the Locking Script table is possible but not adviced: It is not standard and not sufficiently tested.

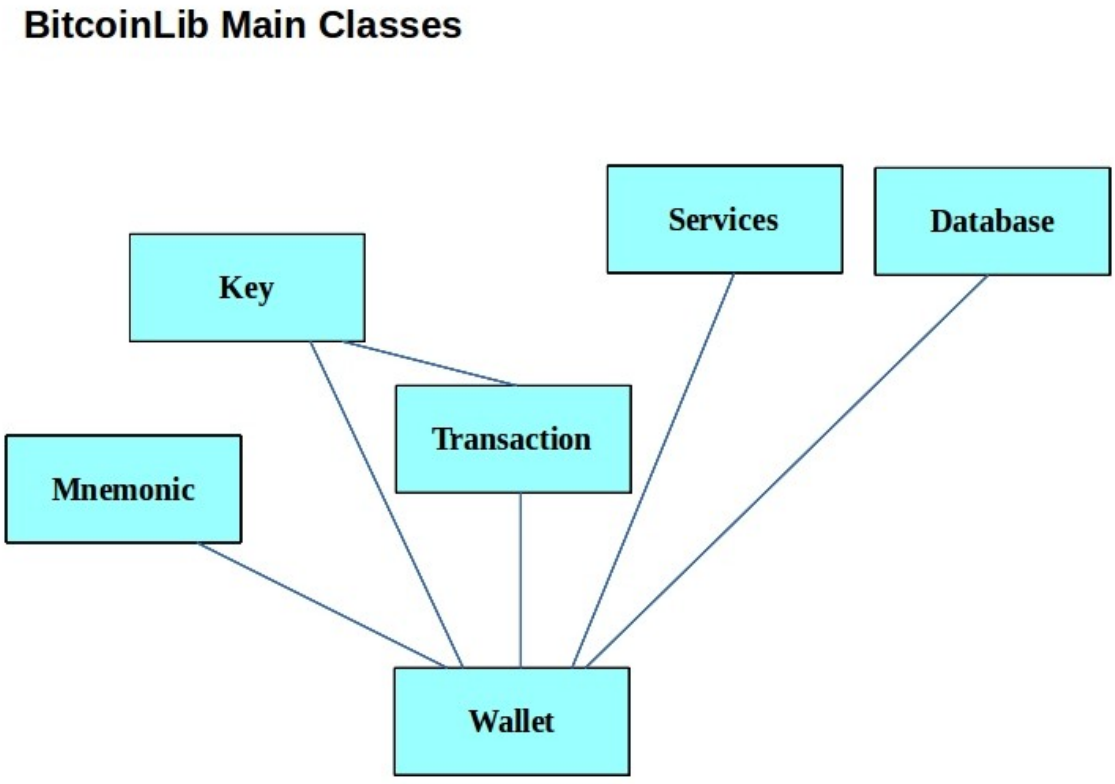
CHAPTER 8

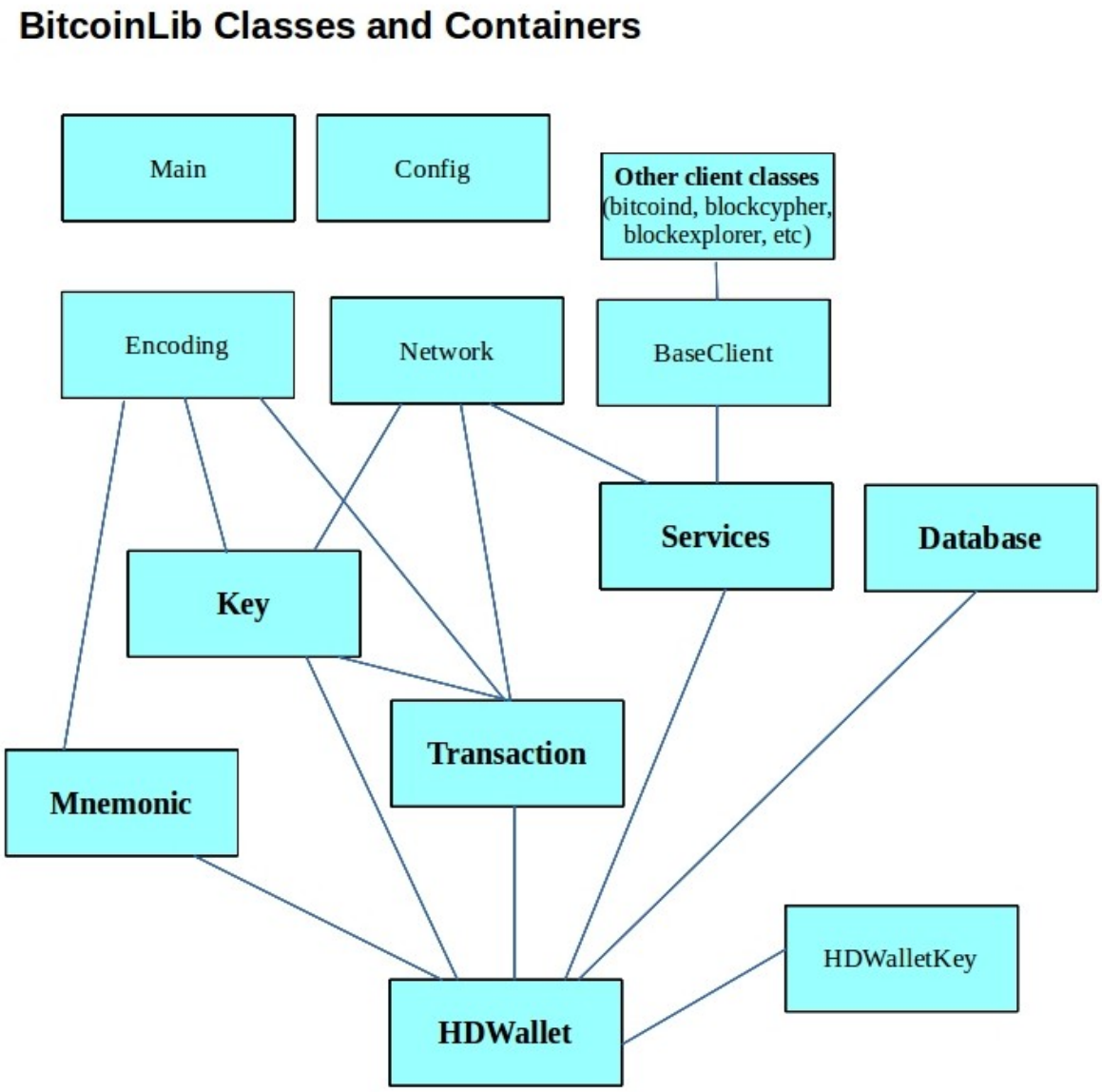
# Disclaimer

This library is still in development, please use at your own risk and test sufficiently before using it in a production environment.

CHAPTER 9

# Schematic overview




BitcoinLib Classes and Containers
Main
Config
Other client classes
(bitcoind, blockcypher,
blockexplorer, etc)
Encoding
Network
BaseClient
Services
Database
Key
Transaction
Mnemonic
HDWalletKey
HDWallet

CHAPTER 10

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## b

# Index

## C

## D



## E



## F



## G

## H



## I



## J



## K



## L



## M

## N



## O



## P



## R



## S

## T



## U



## V

## W