Bitcoinlib Documentation

Release 0.5.2

Lennart (mccwdev)

Feb 07, 2021

MANUALS

Bitcoin and other Crypto Currency Library for Python.

Includes a fully functional wallet, with multi signature, multi currency and multiple accounts. Use this library to create and manage transactions, addresses/keys, wallets, mnemonic password phrases and blocks with simple and straightforward Python code.

You can use this library at a high level and create and manage wallets on the command line or at a low level and create your own custom made transactions, scripts, keys or wallets.

The BitcoinLib connects to various service providers automatically to update wallets, transactions and blockchain information.

CHAPTER

ONE

WALLET

This Bitcoin Library contains a wallet implementation using SQLAlchemy and SQLite3, MySQL or PostgreSQL to import, create and manage keys in a Hierarchical Deterministic way.

Example: Create wallet and generate new address (key) to receive bitcoins

```
>>> from bitcoinlib.wallets import Wallet
>>> w = Wallet.create('Wallet1')
>>> key1 = w.get_key()
>>> key1.address
'1Fo7STj6LdRhUuD1AiEsHpH65pXzraGJ9j'
```

Now send a small transaction to your wallet and use the scan() method to update transactions and UTXO's

```
>>> w.scan()
>>> w.info()  # Shows wallet information, keys, transactions and UTXO's
```

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

```
>>> t = w.send_to('1PWXhWvUH3bcDWn6Fdq3xhMRPfxRXTjAi1', '0.001 BTC')
'b7feea5e7c79d4f6f343b5ca28fa2a1fcacfe9a2b7f44f3d2fd8d6c2d82c4078'
>>> t.info  # Shows transaction information and send results
```

CHAPTER

TWO

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 Wallet
>>> w = Wallet.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 Wallet
>>> w = Wallet.create('segwit_p2sh_p2wpkh', witness_type='p2sh-segwit')
>>> w.get_key().address
36ESSWgR4WxXJSc4ysDSJvecyY6FJkhUbp
```

CHAPTER

THREE

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 Wallet, wallet_delete
from bitcoinlib.mnemonic import Mnemonic

passphrase = Mnemonic().generate()
print(passphrase)
w = Wallet.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

FOUR

MULTI SIGNATURE WALLETS

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

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

NETWORK = 'testnet'
k1 = HDKey(
↪'tprv8ZgxMBicQKsPd1Q44tfDiZC98iYouKRC2CzjT3HGt1yYw2zuX2awTotzGAZQEAU9bi2M5MCj8iedP9MREPjUgpDEBwBgG
↪'
           '5zNYeiX8', network=NETWORK)
k2 = HDKey(
↪'tprv8ZgxMBicQKsPeUbMS6kswJc11zgVEXUnUZuGo3bF6bBrAg1ieFfUdPc9UHqbD5HcXizThrcKike1c4z6xHrz6MWGwy8L6
↪'
           'MeQHdWDp', network=NETWORK)
w1 = Wallet.create('multisig_2of2_cosigner1', sigs_required=2,
                   keys=[k1, k2.public_master(multisig=True)], network=NETWORK)
w2 = Wallet.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

FIVE

COMMAND LINE TOOL

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

To create a new Bitcoin wallet

```
$ clw 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 the command line wallet 'clw' 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

SIX

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

SEVEN

OTHER DATABASES

Bitcoinlib uses the SQLite database by default but other databases are supported as well. See http://bitcoinlib.readthedocs.io/en/latest/_static/manuals.databases.html for instructions on how to use MySQL or PostgreSQL.

CHAPTER

EIGHT

MORE EXAMPLES

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

8.1 Install, Update and Tweak BitcoinLib

8.1.1 Installation

8.1.1.1 Install with pip

```
$ pip install bitcoinlib
```

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

8.1.1.2 Install from source

Required packages:

```
sudo apt install -y postgresql postgresql-contrib mysql-server libpq-dev
libmysqlclient-dev
```

Create a virtual environment for instance on linux with virtualenv:

```
$ virtualenv -p python3 venv/bitcoinlib
$ source venv/bitcoinlib/bin/activate
```

Then clone the repository and install dependencies:

```
$ git clone https://github.com/1200wd/bitcoinlib.git
$ cd bitcoinlib
$ pip install -r requirements-dev.txt
```

8.1.1.3 Package dependencies

Required Python Packages, are automatically installed upon installing bitcoinlib:

- fastecdsa
- pyaes
- scrypt (or much slower pyscript)
- sqlalchemy
- requests
- enum34 (for older Python installations)
- pathlib2 (for Python 2)
- six

8.1.1.4 Other requirements Linux

On Debian, Ubuntu or their derivatives:

```
sudo apt install build-essential python-dev python3-dev libgmp3-dev
```

On Fedora, CentOS or RHEL:

```
sudo dnf install python3-devel gmp-devel
```

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 gcc openssl-devel
```

8.1.1.5 Development environment

Install database packages for MySQL and PostgreSQL

```
sudo apt install mysql-server postgresql postgresql-contrib libmysqlclient-dev
```

Check for the latest version of the PostgreSQL dev server:

```
sudo apt install postgresql-server-dev-<version>
```

From library root directory install the Python requirements

```
pip install -r requirements-dev.txt
```

Then run the unittests to see if everything works

```
python setup.py test
```

8.1.1.6 Other requirements Windows

This library requires a Microsoft Visual C++ Compiler. For python version 3.5+ you will need Visual C++ 14.0. Install Microsoft Visual Studio and include the "Microsoft Visual C++ Build Tools" which can be downloaded from https://visualstudio.microsoft.com/downloads. Also see https://wiki.python.org/moin/WindowsCompilers

The fastecdsa library is not enabled at this moment in the windows install, the slower ecdsa library is installed. Installation of fastecdsa on Windows is possible but not easy, read https://github.com/AntonKueltz/fastecdsa/issues/11 for steps you could take to install this library.

If you have problems with installing this library on Windows you could try to use the pyscrypt library instead of scrypt. The pyscrypt library is pure Python so it doesn't need any C compilers installed. But this will run slower.

8.1.2 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/install.log file.

```
$ rm .bitcoinlib/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
```

8.1.3 Troubleshooting

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

```
$ pip uninstall scrypt
$ pip install 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 do not hesitate to contact us or file an issue at https://github.com/1200wd/bitcoinlib/issues

8.1.4 Using library in other software

If you use the library in other software and want to change file locations and other settings you can specify a location for a config file in the BCL_CONFIG_FILE:

```
os.environ['BCL_CONFIG_FILE'] = '/var/www/blocksmurfer/bitcoinlib.ini'
```

8.1.5 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 Bcoin, 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/providers.json

8.2 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.

8.2.1 Create wallet

To create a wallet just specify an unused wallet name:

```
$ clw 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

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
```

8.2.2 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.

```
$ clw mywallet -r
Command Line Wallet for BitcoinLib

Receive address is 1JMKBiiDMdjTx6rfqGumALvcRMX6DQNeG1
```

8.2.3 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.

```
$ clw -d dbtest mywallet -t 1FpBBJ2E9w9nqxHUAtQME8X4wGeAKBsKwZ 10000
```

8.2.4 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.

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

8.2.5 Options Overview

Command Line Wallet for BitcoinLib

```
usage: clw.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]
                   [--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 [COSIGNER_ID], -r [COSIGNER_ID]
                        Show unused address to receive funds. Generate new
                        payment and change 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
```

(continues on next page)

(continued from previous page)

```
                        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
                        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)
```

8.3 Add a new Service Provider

The Service class connects to providers such as Blockchain.info or Blockchair.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.

8.3.1 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",
```

(continues on next page)

(continued from previous page)

```
        "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=['blockchair'])
print(srv.estimatefee(5))
```

8.4 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.

8.4.1 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
```

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

8.4.3 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",
```

(continues on next page)

(continued from previous page)

```
    "api_key": "",
    "priority": 11,
    "denominator": 100000000
  }
}
```

8.4.4 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 the responsibility to store user and password information in a secure way.

```
from bitcoinlib.services.bitcoind import BitcoindClient

base_url = 'http://user:password@server_url:18332'
bdc = BitcoindClient(base_url=base_url)
txid = 'e0cee8955f516d5ed333d081a4e2f55b999debfff91a49e8123d20f7ed647ac5'
rt = bdc.getrawtransaction(txid)
print("Raw: %s" % rt)
```

8.4.5 Please note: Using a remote bitcoind server

Using RPC over a public network is unsafe, so since bitcoind version 0.18 remote RPC for all network interfaces is disabled. The rpcallowip option cannot be used to listen on all network interfaces and rpcbind has to be used to define specific IP addresses to listen on. See https://bitcoin.org/en/release/v0.18.0#configuration-option-changes

You could setup a openvpn or ssh tunnel to connect to a remote server to avoid this issues.

8.5 Using MySQL or PostgreSQL databases

Bitcoinlib uses the SQLite database by default, because it easy to use and requires no installation.

But you can also use other databases. At this moment Bitcoinlib is tested with MySQL and PostgreSQL.

8.5.1 Using MySQL database

We assume you have a MySQL server at localhost. Unlike with the SQLite database MySQL databases are not created automatically, so create one from the mysql command prompt:

```
mysql> create database bitcoinlib;
```

Now create a user for your application and grant this user access. And off course replace the password 'secret' with a better password.

```
mysql> create user bitcoinlib@localhost identified by 'secret';
mysql> grant all on bitcoinlib.* to bitcoinlib@localhost with grant option;
```

In your application you can create a database link. The database tables are created when you first run the application

```
db_uri = 'mysql://bitcoinlib:secret@localhost:3306/bitcoinlib'
w = wallet_create_or_open('wallet_mysql', db_uri=db_uri)
w.info()
```

8.5.2 Using PostgreSQL database

First create a user and the database from a shell. We assume you have a PostgreSQL server running at your Linux machine.

```
$ su - postgres
postgres@localhost:~$ createuser --interactive --pwprompt
Enter name of role to add: bitcoinlib
Enter password for new role:
Enter it again:
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
$ createdb bitcoinlib
```

And assume you unwisely have chosen the password 'secret' you can use the database as follows:

```
db_uri = 'postgresql://bitcoinlib:secret@localhost:5432/'
w = wallet_create_or_open('wallet_mysql', db_uri=db_uri)
w.info()
```

8.6 Using SQLCipher encrypted database

To protect your data such as the private keys you can use SQLCipher to encrypt the full database. SQLCipher is a SQLite extension which uses 256-bit AES encryption and also works together with SQLAlchemy.

Is quite easy to setup and use with Bitcoinlib. First install the required packages, the following works on Ubuntu, but your system might require other packages. Please read https://www.zetetic.net/sqlcipher/ for installations instructions.

```
$ sudo apt install sqlcipher libsqlcipher0 libsqlcipher-dev
$ pip install pysqlcipher3
```

Now you can use a SQLCipher database URI to create and query a encrypted database:

```
password = 'secret'
filename = '~/.bitcoinlib/database/bcl_encrypted.db'
db_uri = 'sqlite+pysqlcipher://:%s@/%s?cipher=aes-256-cfb&kdf_iter=64000' % (password,
↪ filename)
wlt = Wallet.create('bcltestwlt4', network='bitcoinlib_test', db_uri=db_uri)
```

If you look at the contents of the SQLite database you can see it is encrypted.

```
$ cat ~/.bitcoinlib/database/bcl_encrypted.db
<outputs unreadable random garbage>
```

8.7 10 Tips to Increase Privacy and Security

Ten tips for more privacy and security when using Bitcoin and Bitcoinlib:

1. Run your own Bitcoin or Bcoin node, so you are not depending on external Blockchain API service providers anymore. This not only increases your privacy, but also makes your application much faster and more reliable. And as extra bonus you support the Bitcoin network.
2. Use multi-signature wallets. So you are able to store your private keys in separate (offline) locations.
3. Use a minimal amount of inputs when creating a transaction. This is default behavior the Bitcoinlib Wallet object. You can set a hard limit when sending from a wallet with the max_utxos=1 attribute.
4. Use a random number of change outputs and shuffle order of inputs and outputs. This way it is not visible which output is the change output. In the Wallet object you can set the number_of_change_outputs to zero to generate a random number of change outputs.
5. Encrypt your database with SQLCipher.
6. Use password protected private keys. For instance use a password when creating wallets.
7. Backup private keys and passwords! I have no proof but I assume more bitcoins are lost because of lost private keys then there are lost due to hacking...
8. When using Bitcoinlib wallets the private keys are stored in a database. Make sure the database is in a safe location and check encryption, access rights, etc. Also check tip 2 and 5 again and see how you can minimize risks.
9. Test, try, review. Before working with any real value carefully test your applications using the testnet or small value transactions.
10. Read this tips, read some more about Security and Privacy and then think thorough about the best wallet setup, which is always a tradeoff between security, privacy and usability.

8.8 Caching

Results from queries to service providers are store in a cache database. Once transactions are confirmed and stored on the blockchain they are immutable, so they can be stored in a local cache for an indefinite time.

8.8.1 What is cached?

The cache stores transactions, but also address information and transactions-address relations. This speeds up the gettransactions(), getutxos() and getbalance() method since all old transactions can be read from cache, and we only have to check if new transactions are available for a certain address.

The latest block - block number of the last block on the network - is stored in cache for 60 seconds. So the Service object only checks for a new block every minute.

The fee estimation for a specific network is stored for 10 minutes.

8.8.2 Using other databases

By default the cache is stored in a SQLite database in the database folder: ~/.bitcoinlib/databases/bitcoinlib_cache.sqlite The location and type of database can be changed in the config.ini with the default_databasefile_cache variable.

Other type of databases can be used as well, check http://bitcoinlib.readthedocs.io/en/latest/_static/manuals.databases.html for more information.

8.8.3 Disable caching

Caching is enabled by default. To disable caching set the environment variable SERVICE_CACHING_ENABLED to False or set this variable (service_caching_enabled) in the config.ini file placed in your .bitcoinlib/ directory.

8.8.4 Troubleshooting

8.8.4.1 Nothing is cached, what is the problem?

- If the min_providers parameter is set to 2 or more caching will be disabled.
- If a service providers returns an incomplete result no cache will be stored.
- If the after_txid parameter is used in gettransactions() or getutxos() no cache will be stored if this the 'after_txid' transaction is not found in the cache. Because the transaction cache has to start from the first transaction for a certain address and no gaps can occur.

8.8.4.2 I get incomplete or incorrect results!

- Please post an issues in the Github issue-tracker so we can take a look.
- You can delete the database in ~/.bitcoinlib/databases/bitcoinlib_cache.sqlite for an easy fix, or disable caching if that really doesn't work out.

8.9 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.

```
>>> addr = Address(
↪'03715219f51a2681b7642d1e0e35f61e5288ff59b87d275be9eaf1a5f481dcdeb6', encoding=
↪'bech32', script_type='p2wsh')
>>> addr.address
'bc1qaehsuffn0stxmugx3z69z9hm6gnjd9qzeqlfv92cpf5adw63x4tsfl7vwl'
```

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

property data

property hashed_data

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.

```
>>> addr = Address.import_address(
↪'bc1qyftqrh3hm2yapnhh0ukaht83d02a7pda8l5uhkxk9ftzqsmyu7pst6rke3')
>>> addr.as_dict()
{'network': 'bitcoin', '_data': None, 'script_type': 'p2wsh', 'encoding':
↪'bech32', 'compressed': None, 'witness_type': 'segwit', 'depth': None,
↪'change': None, 'address_index': None, 'prefix': 'bc', 'redeemscript': '',
↪'_hashed_data': None, 'address':
↪'bc1qyftqrh3hm2yapnhh0ukaht83d02a7pda8l5uhkxk9ftzqsmyu7pst6rke3', 'address_
↪orig': 'bc1qyftqrh3hm2yapnhh0ukaht83d02a7pda8l5uhkxk9ftzqsmyu7pst6rke3'}
```

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
- **depth** (*int*) – Level of depth in BIP32 key path

- **change** (*int*) – Use 0 for normal address/key, and 1 for change address (for returned/change payments)
- **address_index** (*int*) – Index of address. Used in BIP32 key paths
- **network** (*str*) – Specify network filter, i.e.: bitcoin, testnet, litecoin, etc. Wil trigger check if address is valid for this 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'\x00\x00\x00\x00', child_index=0, is_private=True, network=None, key_type='bip32', password='', 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.

```
>>> private_hex =
↪'221ff330268a9bb5549a02c801764cffbc79d5c26f4041b26293a425fd5b557c'
>>> k = HDKey(private_hex)
>>> k
<HDKey(public_
↪hex=0363c152144dcd5253c1216b733fdc6eb8a94ab2cd5caa8ead5e59ab456ff99927, wif_
↪public=xpub661MyMwAqRbcEYS8w7XLSVeEsBXy79zSzH1J8vCdxAZningWLdN3zgtU6SmypHzZG2cYrwpGkWJqRxS6EAW
↪ network=bitcoin)>
```

Parameters

- **import_key** (*str, bytes, int*) – 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'
- **password** (*str*) – Optional password 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

address(*compressed=None*, *prefix=None*, *script_type=None*, *encoding=None*)

Get address derived from public key

```
>>> wif =
↪'xpub661MyMwAqRbcFcXi3aM3fVdd42FGDSdufhrr5tdobiPjMrPUykFMTdaFEr7yoy1xxeifDY8kh2k4h9N77MY6r
↪'
>>> k = HDKey(wif)
>>> k.address()
'15CacK61qnzJKpSpx9PFiC8X1ajeQxhq8a'
```

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 or Bech32 encoded address

as_dict(*include_private=False*)

Get current HDKey class as dictionary. Byte values are represented by hexadecimal strings.

Parameters **include_private** (*bool*) – Include private key information in dictionary

Return collections.OrderedDict

as_json(*include_private=False*)

Get current key as json formatted string

Parameters **include_private** (*bool*) – Include private key information in dictionary

Return str

bip38_encrypt(*password*)

BIP0038 non-ec-multiply encryption. Returns BIP0038 encrypted private key Based on code from https://github.com/nomorecoin/python-bip38-testing

```
>>> k = HDKey(
↪'zprvAWgYBBk7JR8GjAHfvjhGLKFGUJNcnPtkNryWfstePYJc4SVFYbaFk3Fpqn9dSmtPLKrPWB7WzsgzZzFiB1Qnh
↪')
>>> k.bip38_encrypt('my-secret-password')
'6PYUAKyDYo7Q6sSJ3ZYo4EFeWFTMkUES2mdvsMNBSoN5QyXPmeogxfumfW'
```

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

Return str BIP38 password encrypted private key

child_private(*index=0*, *hardened=False*, *network=None*)

Use Child Key Derivation (CDK) to derive child private key of current HD Key object.

Used by *subkey_for_path()* to create key paths for instance to use in HD wallets. You can use this method to create your own key structures.

This method create private child keys, use *child_public()* to create public child keys.

```
>>> private_hex =
↪'d02220828cad5e0e0f25057071f4dae9bf38720913e46a596fd7eb8f83ad045d'
>>> k = HDKey(private_hex)
>>> ck = k.child_private(10)
>>> ck.address()
'1FgHK5JUa87ASxz5mz3ypeaUV23z9yW654'
>>> ck.depth
1
>>> ck.child_index
10
```

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.

Used by *subkey_for_path()* to create key paths for instance to use in HD wallets. You can use this method to create your own key structures.

This method create public child keys, use *child_private()* to create private child keys.

```
>>> private_hex =
↪'d02220828cad5e0e0f25057071f4dae9bf38720913e46a596fd7eb8f83ad045d'
>>> k = HDKey(private_hex)
>>> ck = k.child_public(15)
>>> ck.address()
'1PfLJJgKs8nUbMPpaQUucbGmr8qyNSMGeK'
>>> ck.depth
1
```

(continues on next page)

(continued from previous page)

```
>>> ck.child_index
15
```

Parameters

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

Return HDKey HD Key class object

property fingerprint

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

Return bytes

static from_passphrase(*passphrase*, *password=''*, *network='bitcoin'*, *key_type='bip32'*, *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
- **key_type** (*str*) – HD BIP32 or normal Private Key. Default is 'bip32'
- **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(*import_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 addresses

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. A public master key can be shared with other software administration tools to create readonly wallets or can be used to create multisignature wallets.

```
>>> private_hex =
↪'b66ed9778029d32ebede042c79f448da8f7ab9efba19c63b7d3cdf6925203b71'
>>> k = HDKey(private_hex)
>>> pm = k.public_master()
>>> pm.wif()

↪'xpub6CjFexgdDZEtHdW7V4LT8wS9rtG3m187pM9qhTpoZdViFhSv3tW9sWonQNtFN1TCkRGAQGKj1UC2ViHTqb7vJ
↪'
```

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. Derived from witness_type and multisig arguments if not provided
- **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

See BIP0044 bitcoin proposal for more explanation.

```
>>> wif =
↪'xprv9s21ZrQH143K4LvcS93AHEZh7gBiYND6zMoRiZQGL5wqbpCU2KJDY87Txuv9dduk9hAcsL76F8b5JKzDREf8E
↪'
>>> k = HDKey(wif)
>>> k.subkey_for_path("m/44'/0'/0'/0/2")
<HDKey(public_
↪hex=03004331ca7f0dcdd925abc4d0800a0d4a0562a02c257fa39185c55abdfc4f0c0c, wif_
↪public=xpub6GyQoEbMUNwu1LnbiCSaD8wLrcjyRCEQA8tNsFCH4pnvCbuWSZkSB6LUNe89YsCBTg1Ncs7vHJBjMvw
↪ network=bitcoin)>
```

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

```
>>> private_hex =
↪'221ff330268a9bb5549a02c801764cffbc79d5c26f4041b26293a425fd5b557c'
>>> k = HDKey(private_hex)
>>> k.wif()

↪'xpub661MyMwAqRbcEYS8w7XLSVeEsBXy79zSzH1J8vCdxAZningWLdN3zgtU6SmypHzZG2cYrwpGkWJqRxS6EAW77
↪'
```

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** (*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 multi signature wallet?

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*, *password=''*, *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 import_key is empty a new private key will be generated.

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 compressed boolean attribute tells if the original imported key was compressed or not.

```
>>> k = Key('cNUpWJbC1hVJtyxyV4bVAnb4uJ7FPhr82geo1vnoA29XWkeiiCQn')
>>> k.secret
121272277086107546203375539852452923964442161118036950284195449442134423903 63
```

Can also be used to import BIP-38 password protected keys

```
>>> k2 = Key('6PYM8wAnnmAK5mHYoF7zqj88y5HtK7eiPeqPdu4WnYEFkYKEEoMFEVfuDg',␣
↪password='test', network='testnet')
>>> k2.secret
121272277086107546203375539852452923964442161118036950284195449442134423903 63
```

Parameters

- **import_key** (*str, int, bytes*) – 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
- **password** (*str*) – Optional password 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 or Bech32 encoded address

property 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(*include_private=False*)
Get current Key class as dictionary. Byte values are represented by hexadecimal strings.

Parameters **include_private** (*bool*) – Include private key information in dictionary

Return collections.OrderedDict

as_json(*include_private=False*)
Get current key as json formatted string

Parameters **include_private** (*bool*) – Include private key information in dictionary

Return str

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

```
>>> k = Key('cNUpWJbC1hVJtyxyV4bVAnb4uJ7FPhr82geo1vnoA29XWkeiiCQn')
>>> k.bip38_encrypt('test')
'6PYM8wAnnmAK5mHYoF7zqj88y5HtK7eiPeqPdu4WnYEFkYKEEoMFEVfuDg'
```

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

Return str BIP38 password encrypted private key

property 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

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

property x

property y

class bitcoinlib.keys.**Signature**(*r, s, txid=None, secret=None, signature=None, der_signature=None, public_key=None, k=None, hash_type=1*)

Bases: object

Signature class for transactions. Used to create signatures to sign transaction and verification

Sign a transaction hash with a private key and show DER encoded signature:

```
>>> sk = HDKey('f2620684cef2b677dc2f043be8f0873b61e79b274c7e7feeb434477c082e0dc2')
>>> txid = 'c77545c8084b6178366d4e9a06cf99a28d7b5ff94ba8bd76bbbce66ba8cdef70'
>>> signature = sign(txid, sk)
>>> signature.as_der_encoded().hex()

↪'3044022015f9d39d8b53c68c7549d5dc4cbdafe1c71bae3656b93a02d2209e413d9bbcd00220615cf626da0a81945
↪'
```

Initialize Signature object with provided r and r value

```
>>> r =␣
↪32979225540043540145671192266052053680452913207619328973512110841045982813493
>>> s =␣
↪12990793585889366641563976043319195006380846016310271470330687369836458989268
```

(continues on next page)

(continued from previous page)

```
>>> sig = Signature(r, s)
>>> sig.hex()

↪'48e994862e2cdb372149bad9d9894cf3a5562b4565035943efe0acc502769d351cb88752b5fe8d70d85f3541046df
↪'
```

Parameters

- **r** (*int*) – r value of signature
- **s** (*int*) – s value of signature
- **txid** (*bytes, hexstring*) – Transaction hash z to sign if known
- **secret** (*int*) – Private key secret number
- **signature** (*str, bytes*) – r and s value of signature as string
- **der_signature** (*str, bytes*) – DER encoded signature
- **public_key** (HDKey, Key, *str, hexstring, bytes*) – Provide public key P if known
- **k** (*int*) – k value used for signature

as_der_encoded(*as_hex=False*)

Get DER encoded signature

Parameters **as_hex** (*bool*) – Output as hexstring

Return bytes

bytes()

Signature r and s value as single bytes string

Return bytes

static create(*txid*, *private*, *use_rfc6979=True*, *k=None*)

Sign a transaction hash and create a signature with provided private key.

```
>>> k = 'b2da575054fb5daba0efde613b0b8e37159b8110e4be50f73cbe6479f6038f5b'
>>> txid = '0d12fdc4aac9eaaab9730999e0ce84c3bd5bb38dfd1f4c90c613ee177987429c'
>>> sig = Signature.create(txid, k)
>>> sig.hex()

↪'48e994862e2cdb372149bad9d9894cf3a5562b4565035943efe0acc502769d351cb88752b5fe8d70d85f35410
↪'
>>> sig.r
32979225540043540145671192266052053680452913207619328973512110841045982813493
>>> sig.s
12990793585889366641563976043319195006380846016310271470330687369836458989268
```

Parameters

- **txid** (*bytes, str*) – Transaction signature or transaction hash. If unhashed transaction or message is provided the double_sha256 hash of message will be calculated.
- **private** (HDKey, Key, *str, hexstring, bytes*) – Private key as HDKey or Key object, or any other string accepted by HDKey object

- **use_rfc6979** (*bool*) – Use deterministic value for k nonce to derive k from txid/message according to RFC6979 standard. Default is True, set to False to use random k
- **k** (*int*) – Provide own k. Only use for testing or if you known what you are doing. Providing wrong value for k can result in leaking your private key!

Return Signature

static from_str(*signature*, *public_key=None*)
Create a signature from signature string with r and s part. Signature length must be 64 bytes or 128 character hexstring

Parameters

- **signature** (*bytes, str*) – Signature string
- **public_key** (HDKey, Key, *str, hexstring, bytes*) – Public key as HDKey or Key object or any other string accepted by HDKey object

Return Signature

hex()
Signature r and s value as single hexadecimal string

Return hexstring

property public_key
Return public key as HDKey object

Return HDKey

property txid

verify(*txid=None*, *public_key=None*)
Verify this signature. Provide txid or public_key if not already known

```
>>> k = 'b2da575054fb5daba0efde613b0b8e37159b8110e4be50f73cbe6479f6038f5b'
>>> pub_key = HDKey(k).public()
>>> txid = '0d12fdc4aac9eaaab9730999e0ce84c3bd5bb38dfd1f4c90c613ee177987429c'
>>> sig =
↪'48e994862e2cdb372149bad9d9894cf3a5562b4565035943efe0acc502769d351cb88752b5fe8d70d85f35410
↪'
>>> sig = Signature.from_str(sig)
>>> sig.verify(txid, pub_key)
True
```

Parameters

- **txid** (*bytes, hexstring*) – Transaction hash
- **public_key** (HDKey, Key, *str, hexstring, bytes*) – Public key P

Return bool

bitcoinlib.keys.**addr_convert**(*addr*, *prefix*, *encoding=None*, *to_encoding=None*)
Convert address to another encoding and/or address with another prefix.

```
>>> addr_convert('1GMDUKLom6bJuY37RuFNc6PHv1rv2Hziuo', prefix='bc', to_encoding=
↪'bech32')
'bc1q4pwfmstmw8q80nxtxud2h42lev9xzcjqwqyq7t'
```

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.

```
>>> check_network_and_key('L4dTuJf2ceEdWDvCPsLhYf8GiiuYqXtqfbcKdC21BPDvEM1ykJRC')
'bitcoin'
```

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

Parameters

- **key** (*str, int, bytes*) – 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 contains the network with highest priority if multiple networks are found. Same applies for the script type.

Specify the network argument if network is known to avoid unexpected results.

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

```
>>> deserialize_address('1Khyc5eUddbhYZ8bEZi9wiN8TrmQ8uND4j')
{'address': '1Khyc5eUddbhYZ8bEZi9wiN8TrmQ8uND4j', 'encoding': 'base58', 'public_
↪key_hash': 'cd322766c02e7c37c3e3f9b825cd41ffbdcd17d7', 'public_key_hash_bytes':␣
↪b"\xcd2'f\xc0.|7\xc3\xe3\xf9\xb8%\xcdA\xff\xbd\xcd\x17\xd7", 'prefix': b'\x00',
↪'network': 'bitcoin', 'script_type': 'p2pkh', 'witness_type': 'legacy',
↪'networks': ['bitcoin']}
```

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*) – Specify network filter, i.e.: bitcoin, testnet, litecoin, etc. Wil trigger check if address is valid for this network

Return dict with information about this address

bitcoinlib.keys.**ec_point**(*m*)

Method for elliptic curve multiplication on the secp256k1 curve. Multiply Generator point G with m

Parameters **m** (*int*) – 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.

```
>>> get_key_format('L4dTuJf2ceEdWDvCPsLhYf8GiiuYqXtqfbcKdC21BPDvEM1ykJRC')
{'format': 'wif_compressed', 'networks': ['bitcoin'], 'is_private': True, 'script_
↪types': [], 'witness_types': ['legacy'], 'multisig': [False]}
```

```
>>> get_key_format(
↪'becc7ac3b383cd609bd644aa5f102a811bac49b6a34bbd8afe706e32a9ac5c5e')
{'format': 'hex', 'networks': None, 'is_private': True, 'script_types': [],
↪'witness_types': ['legacy'], 'multisig': [False]}
```

```
>>> get_key_format(
↪'Zpub6vZyhw1ShkEwNxtqfjk7jiwoEbZYMJdbWLHvEwo6Ns2fFc9rdQn3SerYFQXYxtZYbA8a1d83shW3g4WbsnVsymy2L
↪')
{'format': 'hdkey_public', 'networks': ['bitcoin'], 'is_private': False, 'script_
↪types': ['p2wsh'], 'witness_types': ['segwit'], 'multisig': [True]}
```

Parameters

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

Return dict Dictionary with format, network and is_private

bitcoinlib.keys.**mod_sqrt**(*a*)

Compute the square root of 'a' using the secp256k1 'bitcoin' curve

Used to calculate y-coordinate if only x-coordinate from public key point is known. Formula: y ** 2 == x ** 3 + 7

Parameters **a** (*int*) – Number to calculate square root

Return int

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

```
>>> path_expand([10, 20], witness_type='segwit')
['m', "84'", "0'", "0'", '10', '20']
```

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’
- **multisig** (*bool*) – Is path for multisig keys?
- **network** (*str*) – Network name. Leave empty for default network

Return list

bitcoinlib.keys.**sign**(*txid*, *private*, *use_rfc6979=True*, *k=None*)

Sign transaction hash or message with secret private key. Creates a signature object.

Sign a transaction hash with a private key and show DER encoded signature

```
>>> sk = HDKey('728afb86a98a0b60cc81faadaa2c12bc17d5da61b8deaf1c08fc07caf424d493')
>>> txid = 'c77545c8084b6178366d4e9a06cf99a28d7b5ff94ba8bd76bbbce66ba8cdef70'
>>> signature = sign(txid, sk)
>>> signature.as_der_encoded().hex()

↪'30440220792f04c5ba654e27eb636ceb7804c5590051dd77da8b80244f1fa8dfbff369b302204ba03b039c808a040
↪'
```

Parameters

- **txid** (*bytes, str*) – Transaction signature or transaction hash. If unhashed transaction or message is provided the double_sha256 hash of message will be calculated.
- **private** (HDKey, Key, *str*, *hexstring*, *bytes*) – Private key as HDKey or Key object, or any other string accepted by HDKey object
- **use_rfc6979** (*bool*) – Use deterministic value for k nonce to derive k from txid/message according to RFC6979 standard. Default is True, set to False to use random k
- **k** (*int*) – Provide own k. Only use for testing or if you known what you are doing. Providing wrong value for k can result in leaking your private key!

Return Signature

bitcoinlib.keys.**verify**(*txid*, *signature*, *public_key=None*)

Verify provided signature with txid message. If provided signature is no Signature object a new object will be created for verification.

```
>>> k = 'b2da575054fb5daba0efde613b0b8e37159b8110e4be50f73cbe6479f6038f5b'
>>> pub_key = HDKey(k).public()
>>> txid = '0d12fdc4aac9eaaab9730999e0ce84c3bd5bb38dfd1f4c90c613ee177987429c'
>>> sig =
↪'48e994862e2cdb372149bad9d9894cf3a5562b4565035943efe0acc502769d351cb88752b5fe8d70d85f3541046df
↪'
```

(continues on next page)

(continued from previous page)

```
>>> verify(txid, sig, pub_key)
True
```

Parameters

- **txid** (*bytes, hexstring*) – Transaction hash
- **signature** (*str, bytes*) – signature as hexstring or bytes
- **public_key** (HDKey, Key, *str, hexstring, bytes*) – Public key P. If not provided it will be derived from provided Signature object or raise an error if not available

Return bool

8.10 bitcoinlib.transactions module

class bitcoinlib.transactions.**Input**(*prev_txid*, *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*, *witnesses=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_txid + 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_txid** (*bytes, str*) – 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** (*list (bytes, str,* Signature*)*) – Specify optional signatures
- **public_hash** (*bytes*) – Public key hash or script hash. Specify if key is not available
- **unlocking_script** (*bytes, hexstring*) – Unlocking script (scriptSig) to prove ownership. Optional
- **unlocking_script_unsigned** (*bytes, hexstring*) – Unlocking script for signing transaction
- **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, str*) – Value of input in smallest denominator integers (Satoshi's) or as Value object or string
- **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 Sequence 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.
- **witnesses** (*list of bytes*) – List of witnesses for inputs, used for segwit transactions for instance.
- **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_txid, output_n, type, address, public_key, public_hash, unlocking_script and sequence

set_locktime_relative_blocks(*blocks*)

Set nSequence relative locktime for this transaction input. The transaction will only be valid if the specified number of blocks has been mined since the previous UTXO is confirmed.

Maximum number of blocks is 65535 as defined in BIP-0068, which is around 455 days.

When setting an relative timelock, the transaction version must be at least 2. The transaction will be updated so existing signatures for this input will be removed.

Parameters **blocks** (*int*) – The blocks value is the number of blocks since the previous transaction output has been confirmed.

Return None

set_locktime_relative_time(*seconds*)

Set nSequence relative locktime for this transaction input. The transaction will only be valid if the specified amount of seconds have been passed since the previous UTXO is confirmed.

Number of seconds will be rounded to the nearest 512 seconds. Any value below 512 will be interpreted as 512 seconds.

Maximum number of seconds is 33553920 (512 * 65535), which equals 384 days. See BIP-0068 definition.

When setting an relative timelock, the transaction version must be at least 2. The transaction will be updated so existing signatures for this input will be removed.

Parameters **seconds** – Number of seconds since the related previous transaction output has been confirmed.

Returns

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*, *spending_txid=''*, *spending_index_n=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, Value, str*) – Amount of output in smallest denominator integers (Satoshi's) or as Value object or string
- **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
- **spending_txid** (*str*) – Transaction hash of input spending this transaction output

- **spending_index_n** (*int*) – Index number of input spending this transaction output
- **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=None, network='bitcoin', fee=None, fee_per_kb=None, size=None, txid='', txhash='', date=None, confirmations=None, block_height=None, block_hash=None, input_total=0, output_total=0, rawtx=b'', 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.

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.
- **size** (*int*) – Transaction size in bytes
- **txid** (*str*) – The transaction id (same for legacy/segwit) based on [nVersion][txins][txouts][nLockTime as hexadecimal string
- **txhash** (*str*) – The transaction hash (differs from txid for witness transactions), based on [nVersion][marker][flag][txins][txouts][witness][nLockTime] in Segwit (as hexadecimal string). Unused at the moment

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

add_input(*prev_txid*, *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*, *witnesses=None*, *encoding=None*)

Add input to this transaction

Wrapper for append method of Input class.

Parameters

- **prev_txid** (*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*) – 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, used for timelocked transactions
- **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.
- **witnesses** (*list of bytes*) – List of witnesses for inputs, used for segwit transactions for instance.
- **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*, *spending_txid=None*, *spending_index_n=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, None*) – 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
- **spending_txid** (*str*) – Transaction hash of input spending this transaction output
- **spending_index_n** (*int*) – Index number of input spending this transaction output

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

calc_weight_units()

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(*number_of_change_outputs=0*)
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 weight. The formula used is: math.ceil(((est_size-witness_size) * 3 + est_size) / 4)

Parameters **number_of_change_outputs** (*int*) – Number of change outputs, default is 0

Return int Estimated transaction size

static import_raw(*rawtx*, *network='bitcoin'*, *check_size=True*)
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
- **check_size** (*bool*) – Check if not bytes are left when parsing is finished. Disable when parsing list of transactions, such as the transactions in a raw block. Default is True

Return Transaction

info()
Prints transaction information to standard output

static load(*txid=None*, *filename=None*)
Load transaction object from file which has been stored with the *save()* method.

Specify transaction ID or filename.

Parameters

- **txid** (*str*) – Transaction ID. Transaction object will be read from .bitcoinlib datadir
- **filename** (*str*) – Name of transaction object file

Return Transaction

merge_transaction(*transaction*)
Merge this transaction with provided Transaction object.

Add all inputs and outputs of a transaction to this Transaction object. Because the transaction signature changes with this operation, the transaction inputs need to be signed again.

Can be used to implement CoinJoin. Where two or more unrelated Transactions are merged into 1 transaction to safe fees and increase privacy.

Parameters **transaction** (Transaction) – The transaction to be merged

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, None*) – 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

save(*filename=None*)
Store transaction object as file so it can be imported in bitcoinlib later with the *load()* method.

Parameters **filename** (*str*) – Location and name of file, leave empty to store transaction in bitcoinlib data directory: .bitcoinlib/<transaction_id.tx)

Returns

set_locktime_blocks(*blocks*)
Set nLocktime, a transaction level absolute lock time in blocks using the transaction sequence field.

So for example if you set this value to 600000 the transaction will only be valid after block 600000.

Parameters **blocks** (*int*) – Transaction is valid after supplied block number. Value must be between 0 and 500000000. Zero means no locktime.

Returns

set_locktime_relative_blocks(*blocks*, *input_index_n=0*)
Set nSequence relative locktime for this transaction. The transaction will only be valid if the specified number of blocks has been mined since the previous UTXO is confirmed.

Maximum number of blocks is 65535 as defined in BIP-0068, which is around 455 days.

When setting an relative timelock, the transaction version must be at least 2. The transaction will be updated so existing signatures for this input will be removed.

Parameters

- **blocks** (*int*) – The blocks value is the number of blocks since the previous transaction output has been confirmed.
- **input_index_n** (*int*) – Index number of input for nSequence locktime

Return None

set_locktime_relative_time(*seconds*, *input_index_n=0*)
Set nSequence relative locktime for this transaction. The transaction will only be valid if the specified amount of seconds have been passed since the previous UTXO is confirmed.

Number of seconds will be rounded to the nearest 512 seconds. Any value below 512 will be interpreted as 512 seconds.

Maximum number of seconds is 33553920 (512 * 65535), which equals 384 days. See BIP-0068 definition.

When setting an relative timelock, the transaction version must be at least 2. The transaction will be updated so existing signatures for this input will be removed.

Parameters

- **seconds** (*int*) – Number of seconds since the related previous transaction output has been confirmed.
- **input_index_n** (*int*) – Index number of input for nSequence locktime

Returns

set_locktime_time(*timestamp*)
Set nLocktime, a transaction level absolute lock time in timestamp using the transaction sequence field.

Parameters **timestamp** – Transaction is valid after the given timestamp. Value must be between 500000000 and 0xfffffffe

Returns

shuffle()
Shuffle transaction inputs and outputs in random order.

Returns

shuffle_inputs()
Shuffle transaction inputs in random order.

Returns

shuffle_outputs()
Shuffle transaction outputs in random order.

Returns

sign(*keys=None*, *index_n=None*, *multisig_key_n=None*, *hash_type=1*, *fail_on_unknown_key=True*, *replace_signatures=False*)
Sign the transaction input with provided private key

Parameters

- **keys** (HDKey, Key, *bytes*, *list*) – A private key or list of private keys
- **index_n** (*int*) – Index of transaction input. Leave empty to sign all inputs
- **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
- **fail_on_unknown_key** (*bool*) – Method fails if public key from signature is not found in public key list
- **replace_signatures** (*bool*) – Replace signature with new one if already signed.

Return None

sign_and_update(*index_n=None*)
Update transaction ID and resign. Use if some properties of the transaction changed

Parameters **index_n** (*int*) – Index of transaction input. Leave empty to sign all inputs

Returns

signature(*sign_id=None*, *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=None*, *hash_type=1*, *witness_type=None*, *as_hex=False*)
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
- **as_hex** (*bool*) – Return value as hexadecimal string. Default is False

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

property weight_units

witness_data()

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

```
>>> get_unlocking_script_type('p2wsh')
'p2sh_multisig'
```

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*) – 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_to_string**(*script*, *name_data=False*)

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

```
>>> script = '76a914c7402ab295a0eb8897ff5b8fbd5276c2d9d2340b88ac'
>>> script_to_string(script)
'OP_DUP OP_HASH160 hash-20 OP_EQUALVERIFY OP_CHECKSIG'
```

Parameters

- **script** (*bytes, str*) – A locking or unlocking script
- **name_data** (*bool*) – Replace signatures and keys strings with name

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.**transaction_deserialize**(*rawtx*, *network='bitcoin'*, *check_size=True*)

Deserialize a raw transaction

Returns a dictionary with list of input and output objects, locktime and version.

Will raise an error if wrong number of inputs are found or if there are no output found.

Parameters

- **rawtx** (*str, bytes*) – Raw transaction as hexadecimal string or bytes
- **network** (*str, Network*) – Network code, i.e. 'bitcoin', 'testnet', 'litecoin', etc. Leave emtpy for default network
- **check_size** (*bool*) – Check if not bytes are left when parsing is finished. Disable when parsing list of transactions, such as the transactions in a raw block. Default is True

Return Transaction

bitcoinlib.transactions.**transaction_update_spents**(*txs*, *address*)

Update spent information for list of transactions for a specific address. This method assumes the list of transaction complete and up-to-date.

This methods loops through all the transaction and update all transaction outputs for given address, checks if the output is spent and add the spending transaction ID and index number to the outputs.

The same list of transactions with updates outputs will be returned

Parameters

- **txs** (*list of Transaction*) – Complete list of transactions for given address
- **address** (*str*) – Address string

Return list of Transaction

8.11 bitcoinlib.wallets module

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

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

Return str Checksum of key in bits

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

```
>>> Mnemonic().detect_language('chunk gun celery million wood kite tackle␣
↪twenty story episode raccoon dutch')
'english'
```

Parameters **words** (*str*) – List of space separated 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 as multiply of 32, 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 separated words

Return str Sanitized list of words

to_entropy(*words*, *includes_checksum=True*)
Convert Mnemonic words back to key data entropy

```
>>> Mnemonic().to_entropy('chunk gun celery million wood kite tackle twenty␣
↪story episode raccoon dutch').hex()
'28acfc94465fd2f6774759d6897ec122'
```

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 Entropy seed

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

```
>>> Mnemonic().to_mnemonic('28acfc94465fd2f6774759d6897ec122')
'chunk gun celery million wood kite tackle twenty story episode raccoon dutch'
```

Parameters

- **data** (*bytes, hexstring*) – Key data entropy
- **add_checksum** (*bool*) – Included a checksum? Default is True
- **check_on_curve** (*bool*) – Check if data integer value is on secp256k1 curve. Should be enabled when not testing and working with crypto

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

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

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

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

```
>>> Mnemonic().to_seed('chunk gun celery million wood kite tackle twenty␣
↪story episode raccoon dutch').hex()

↪'6969ed4666db67fc74fae7869e2acf3c766b5ef95f5e31eb2fcebd93d76069c6de971225f700042b0b513f0ac
↪'
```

Parameters

- **words** (*str*) – Mnemonic passphrase as string with space separated words
- **password** (*str*) – A password to protect key, leave empty to disable
- **validate** (*bool*) – Validate checksum for given word phrase, default is True

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

8.13 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*, *rep='string'*, *denominator=1*, *decimals=None*)
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 denominator such as Satoshi
- **rep** (*str*) – Currency representation: 'string', 'symbol', 'none' or your own custom name
- **denominator** (*float*) – Unit to use in representation. Default is 1. I.e. 1 = 1 BTC, 0.001 = milli BTC / mBTC
- **decimals** (*int*) – Number of digits after the decimal point, leave empty for automatic determination based on value. Use integer value between 0 and 8

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()  # xpub
b'\x04\x88\xb2\x1e'
>>> Network('bitcoin').wif_prefix(is_private=True, witness_type='segwit',␣
↪multisig=True)  # Zprv
b'\x02\xaaz\x99'
```

Parameters

- **is_private** (*bool*) – Private or public key, default is True
- **witness_type** (*str*) – Legacy, segwit or p2sh-segwit
- **multisig** (*bool*) – Multisignature or single signature wallet. Default is False: no multisig

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 address 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*) – 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*)

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

```
>>> network_values_for('prefix_wif')
[b'\x99', b'\x80', b'\xef', b'\xb0', b'\xb0', b'\xef', b'\xcc', b'\xef', b'\x9e',␣
↪b'\xf1']
>>> network_values_for('prefix_address_p2sh')
[b'\x95', b'\x05', b'\xc4', b'2', b'\x05', b':', b'\x10', b'\x13', b'\x16', b'\xc4
↪']
```

Parameters **field** (*str*) – Prefix name from networks definitions (networks.json)

Return str

bitcoinlib.networks.**print_value**(*value*, *network='bitcoin'*, *rep='string'*, *denominator=1*, *decimals=None*)

Return the value as string with currency symbol

Wrapper for the Network().print_value method.

Parameters

- **value** (*int, float*) – Value in smallest denominator such as Satoshi
- **network** (*str*) – Network name as string, default is 'bitcoin'
- **rep** (*str*) – Currency representation: 'string', 'symbol', 'none' or your own custom name
- **denominator** (*float*) – Unit to use in representation. Default is 1. I.e. 1 = 1 BTC, 0.001 = milli BTC / mBTC, 1e-8 = Satoshi's
- **decimals** (*int*) – Number of digits after the decimal point, leave empty for automatic determination based on value. Use integer value between 0 and 8

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(
↪'xprv9wTYmMFdV23N21MM6dLNavSQV7Sj7meSPXx6AV5eTdqqGLjycVjb115Ec5LgRAXscPZgy5G4jQ9csyyZLN3PZLxoM
↪', 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', 'dogecoin']
```

Parameters

- **wif** (*str*) – WIF string or prefix as 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

8.14 bitcoinlib.blocks module

8.15 bitcoinlib.values module

class bitcoinlib.values.**Value**(*value*, *denominator=None*, *network='bitcoin'*)

Bases: object

Class to represent and convert cryptocurrency values

Create a new Value class. Specify value as integer, float or string. If a string is provided the amount, denominator and currency will be extracted if provided

Examples: Initialize value class >>> Value(10) Value(value=10.00000000000000, denominator=1.00000000, network='bitcoin')

```
>>> Value('15 mBTC')
Value(value=0.01500000000000, denominator=0.00100000, network='bitcoin')
```

```
>>> Value('10 sat')
Value(value=0.00000010000000, denominator=0.00000001, network='bitcoin')
```

```
>>> Value('1 doge')
Value(value=1.00000000000000, denominator=1.00000000, network='dogecoin')
```

```
>>> Value(500, 'm')
Value(value=0.50000000000000, denominator=0.00100000, network='bitcoin')
```

```
>>> Value(500, 0.001)
Value(value=0.50000000000000, denominator=0.00100000, network='bitcoin')
```

All frequently used arithmetic, comparision and logical operators can be used on the Value object. So you can compare Value object, add them together, divide or multiply them, etc.

Values need to use the same network / currency if you work with multiple Value objects. I.e. Value('1 BTC') + Value('1 LTC') raises an error.

Examples: Value operators >>> Value('50000 sat') == Value('5000 fin') # 1 Satoshi equals 10 Finney, see https://en.bitcoin.it/wiki/Units True

```
>>> Value('1 btc') > Value('2 btc')
False
```

```
>>> Value('1000 LTC') / 5
Value(value=200.00000000000000, denominator=1.00000000, network='litecoin')
```

```
>>> Value('0.002 BTC') + 0.02
Value(value=0.02200000000000, denominator=1.00000000, network='bitcoin')
```

The Value class can be represented in several formats.

Examples: Format Value class >>> int(Value("10.1 BTC")) 10

```
>>> float(Value("10.1 BTC"))
10.1
```

```
>>> round(Value("10.123 BTC"), 2).str()
'10.12000000 BTC'
```

```
>>> hex(Value("10.1 BTC"))
'0x3c336080'
```

Parameters

- **value** (*int, float, str*) – Value as integer, float or string. Numeric values must be supllied in smallest denominator such as Satoshi's. String values must be in the format: <value> [<denominator>][<currency_symbol>]
- **denominator** (*int, float, str*) – Denominator as integer or string. Such as 0.001 or m for milli, 1000 or k for kilo, etc. See NETWORK_DENOMINATORS for list of available denominator symbols.
- **network** (*str, Network*) – Specify network if not supplied already in the value string

classmethod from_satoshi(*value*, *denominator=None*, *network='bitcoin'*)

Initialize Value class with smallest denominator as input. Such as represented in script and transactions cryptocurrency values.

Parameters

- **value** (*int*) – Amount of Satoshi's / smallest denominator for this network
- **denominator** (*int, float, str*) – Denominator as integer or string. Such as 0.001 or m for milli, 1000 or k for kilo, etc. See NETWORK_DENOMINATORS for list of available denominator symbols.
- **network** (*str, Network*) – Specify network if not supplied already in the value string

Return Value

str(*denominator=None*, *decimals=None*, *currency_repr='code'*)

Get string representation of Value with requested denominator and number of decimals.

```
>>> Value(1200000, 'sat').str('m')  # milli Bitcoin
'12.00000 mBTC'
```

```
>>> Value(12000.3, 'sat').str(1)  # Use denominator = 1 for Bitcoin
'0.00012000 BTC'
```

```
>>> Value(12000, 'sat').str('auto')
'120.00 µBTC'
```

```
>>> Value(0.005).str('m')
'5.00000 mBTC'
```

```
>>> Value(12000, 'sat').str('auto', decimals=0)
'120 µBTC'
```

```
>>> Value('13000000 Doge').str('auto')  # Yeah, mega Dogecoins...
'13.00000000 MDOGE'
```

```
>>> Value('2100000000').str('auto')
'2.10000000 GBTC'
```

```
>>> Value('1.5 BTC').str(currency_repr='symbol')
'1.50000000 Ƀ'
```

```
>>> Value('1.5 BTC').str(currency_repr='name')
'1.50000000 bitcoins'
```

Parameters

- **denominator** (*int, float, str*) – Denominator as integer or string. Such as 0.001 or m for milli, 1000 or k for kilo, etc. See NETWORK_DENOMINATORS for list of available denominator symbols. If not provided the default self.denominator value is used. Use value 'auto' to automatically determine best denominator for human readability.
- **decimals** (*float*) – Number of decimals to use
- **currency_repr** (*str*) – Representation of currency. I.e. code: BTC, name: bitcoins, symbol: Ƀ

Return str

str_auto (*decimals=None*, *currency_repr='code'*)
String representation of this Value. Wrapper for the *str()* method, but automatically determines the denominator depending on the value.

```
>>> Value('0.0000012 BTC').str_auto()
'120 sat'
```

```
>>> Value('0.0005 BTC').str_auto()
'500.00 µBTC'
```

Parameters

- **decimals** (*float*) – Number of decimals to use
- **currency_repr** (*str*) – Representation of currency. I.e. code: BTC, name: Bitcoin, symbol: ₿

Return str

str_unit (*decimals=None*, *currency_repr='code'*)
String representation of this Value. Wrapper for the *str()* method, but always uses 1 as denominator, meaning main denominator such as BTC, LTC.

```
>>> Value('12000 sat').str_unit()
'0.00012000 BTC'
```

Parameters

- **decimals** (*float*) – Number of decimals to use
- **currency_repr** (*str*) – Representation of currency. I.e. code: BTC, name: Bitcoin, symbol: ₿

Return str

to_bytes (*length=8*, *byteorder='little'*)
Representation of value_sat (value in smallest denominator: satoshi's) as bytes string. Used for script or transaction serialization.

```
>>> Value('1 sat').to_bytes()
b'\x01\x00\x00\x00\x00\x00\x00\x00'
```

Parameters

- **length** (*int*) – Length of bytes string to return, default is 8 bytes
- **byteorder** (*str*) – Order of bytes: little or big endian. Default is 'little'

Return bytes

to_hex (*length=16*, *byteorder='little'*)
Representation of value_sat (value in smallest denominator: satoshi's) as hexadecimal string.

```
>>> Value('15 sat').to_hex()
'0f00000000000000'
```

Parameters

- **length** (*int*) – Length of hexadecimal string to return, default is 16 characters
- **byteorder** (*str*) – Order of bytes: little or big endian. Default is 'little'

Returns

property value_sat

Value in smallest denominator, i.e. Satoshi for the Bitcoin network

Return int

bitcoinlib.values.**value_to_satoshi**(*value*, *network=None*)

Convert Value object or value string to smallest denominator amount as integer

Parameters

- **value** (*str, int, float, Value*) – Value object, value string as accepted by Value class or numeric value amount
- **network** (*str, Network*) – Specify network to validate value string

Return int

8.16 bitcoinlib.services.services module

8.17 bitcoinlib.services package

8.17.1 Submodules

8.17.1.1 bitcoinlib.services.authproxy module

8.17.1.2 bitcoinlib.services.baseclient module

8.17.1.3 bitcoinlib.services.bcoin module

8.17.1.4 bitcoinlib.services.bitaps module

8.17.1.5 bitcoinlib.services.bitcoind module

8.17.1.6 bitcoinlib.services.bitcoinlibtest module

8.17.1.7 bitcoinlib.services.bitgo module

8.17.1.8 bitcoinlib.services.blockchaininfo module

8.17.1.9 bitcoinlib.services.blockchair module

8.17.1.10 bitcoinlib.services.blockcypher module

8.17.1.11 bitcoinlib.services.blocksmurfer module

8.17.1.12 bitcoinlib.services.blockstream module

8.17.1.13 bitcoinlib.services.chainso module

8.17.1.14 bitcoinlib.services.coinfees module

8.17.1.15 bitcoinlib.services.cryptoid module

8.17.1.16 bitcoinlib.services.dashd module

8.17.1.17 bitcoinlib.services.dogecoind module

8.17.1.18 bitcoinlib.services.insightdash module

8.17.1.19 bitcoinlib.services.litecoinblockexplorer module

8.17.1.20 bitcoinlib.services.litecoind module

8.17.1.21 bitcoinlib.services.litecoreio module

8.17.1.22 bitcoinlib.services.smartbit module

8.17.2 Module contents

8.18 bitcoinlib.config package

8.18.1 Submodules

bitcoinlib.config.config.**read_config**()

8.18.1.2 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

8.18.1.3 bitcoinlib.config.secp256k1 module

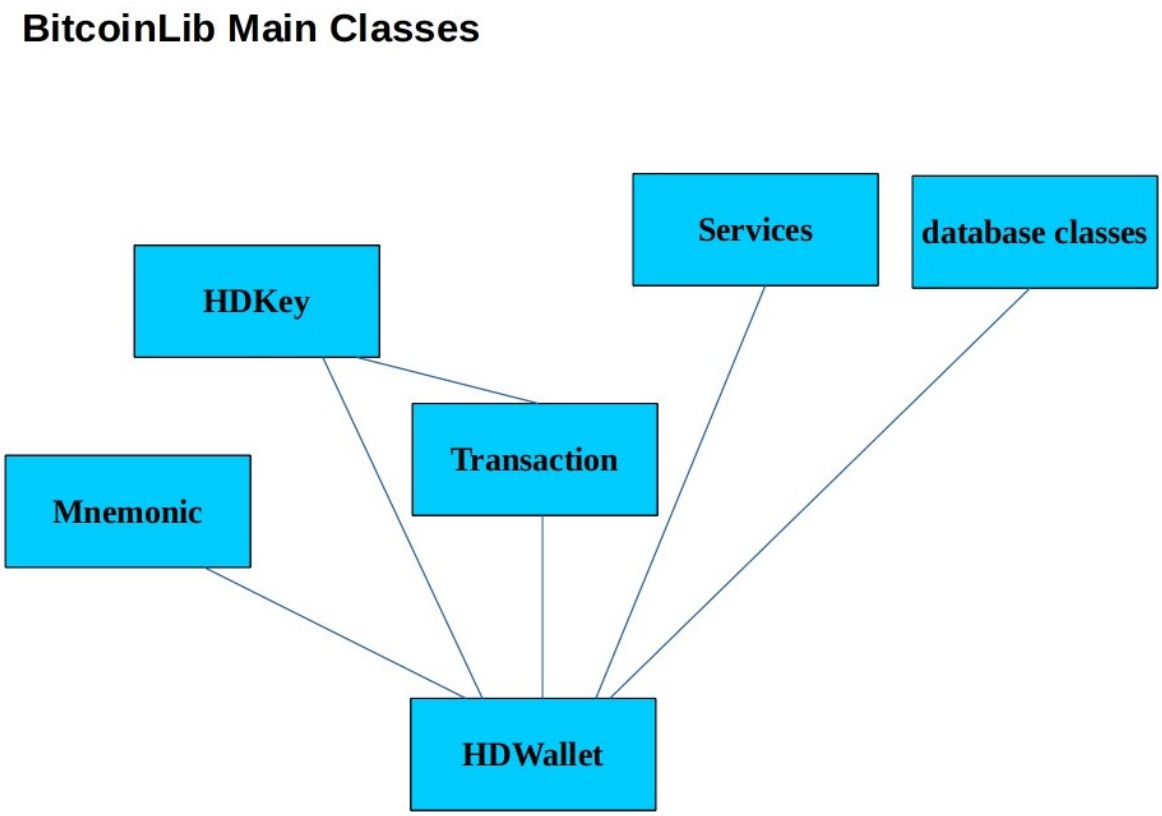
8.18.2 Module contents

8.19 bitcoinlib.db module

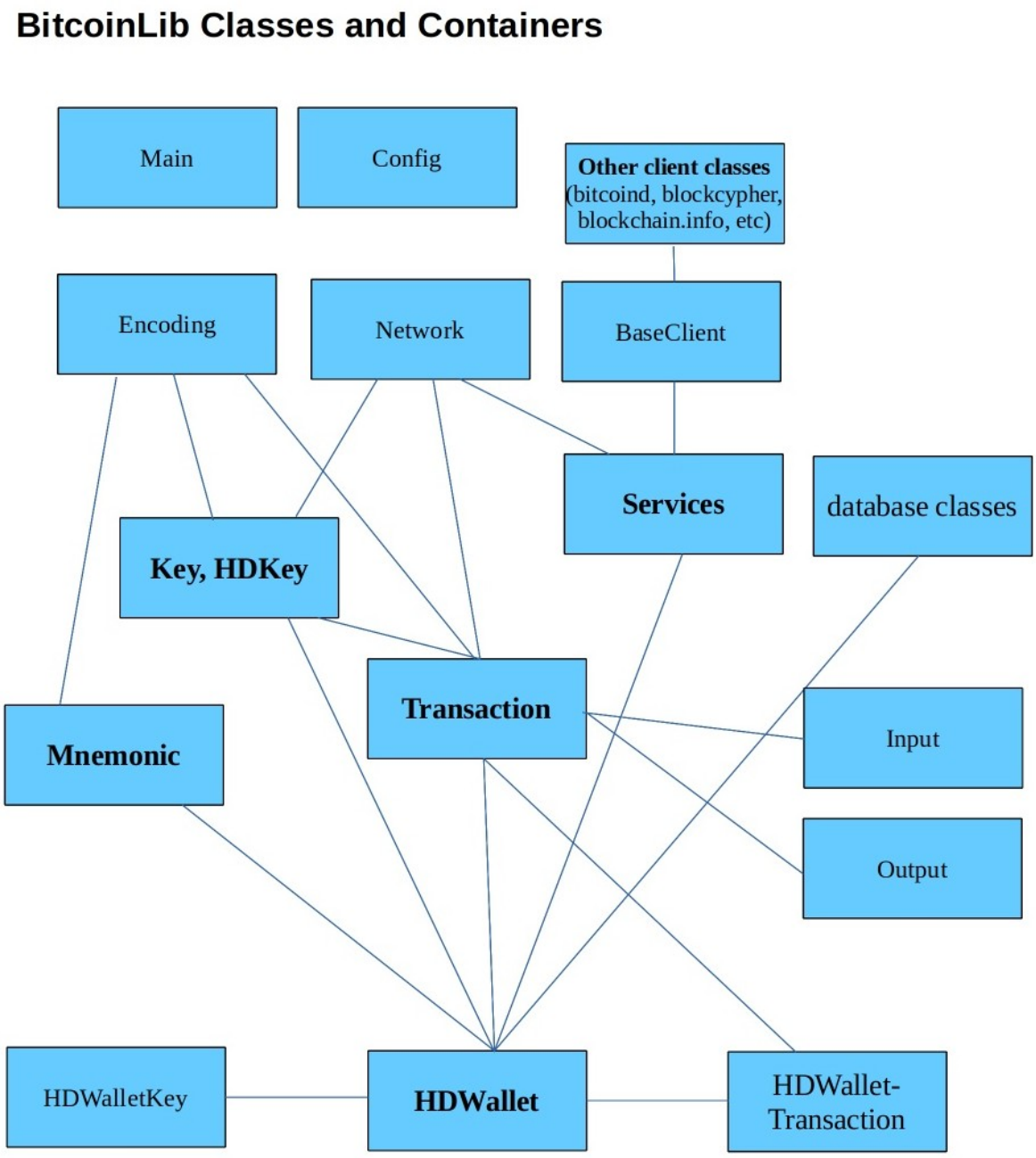
8.20 bitcoinlib.db_cache module

8.21 Classes Overview

These are the main Bitcoinlib classes



This is an overview of all BitcoinLib classes.



So most classes can be used individually and without database setup. The Wallet class needs a proper database setup and is dependent upon most other classes.

8.22 bitcoinlib

8.22.1 bitcoinlib package

8.22.1.1 Subpackages

bitcoinlib.tools package

Submodules

bitcoinlib.tools.clw module

bitcoinlib.tools.mnemonic_key_create module

bitcoinlib.tools.sign_raw module

bitcoinlib.tools.wallet_multisig_2of3 module

Module contents

8.22.1.2 Submodules

bitcoinlib.encoding module

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

Log and raise encoding errors

class bitcoinlib.encoding.**Quantity**(*value*, *units=''*, *precision=3*)
Bases: object

Class to convert very large or very small numbers to a readable format.

Provided value is converted to number between 0 and 1000, and a metric prefix will be added.

```
>>> # Example - the Hashrate on 10th July 2020
>>> str(Quantity(122972532877979100000, 'H/s'))
'122.973 EH/s'
```

Convert given value to number between 0 and 1000 and determine metric prefix

Parameters

- **value** (*int, float*) – Value as integer in base 0
- **units** (*str*) – Base units, so 'g' for grams for instance
- **precision** (*int*) – Number of digits after the comma

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

```
>>> addr_base58_to_pubkeyhash('142Zp9WZn9Fh4MV8F3H5Dv4Rbg7Ja1sPWZ', as_hex=True)
'21342f229392d7c9ed82c932916cee6517fbc9a2'
```

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

```
>>> addr_bech32_to_pubkeyhash('bc1qy8qmc6262m68ny0ftlexs4h9paud8sgce3sf84', as_
↪hex=True)
'21c1bc695a56f47991e95ff26856e50f78d3c118'
```

Validate the bech32 string, and determine HRP and data. Only standard data size of 20 and 32 bytes are excepted

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=None*)

Convert base58 or bech32 address to public key hash

Wrapper for the *addr_base58_to_pubkeyhash()* and *addr_bech32_to_pubkeyhash()* method

Parameters

- **address** (*str*) – Crypto currency address in base-58 format
- **as_hex** (*bool*) – Output as hexstring
- **encoding** (*str*) – Address encoding used: base58 or bech32. Default is base58. Try to derive from address if encoding=None is provided

Return bytes, str public key hash

bitcoinlib.encoding.**bip38_decrypt**(*encrypted_privkey*, *password*)

BIP0038 non-ec-multiply decryption. Returns WIF private key. Based on code from https://github.com/nomorecoin/python-bip38-testing This method is called by Key class init function when importing BIP0038 key.

Parameters

- **encrypted_privkey** (*str*) – Encrypted private key using WIF protected key format
- **password** (*str*) – Required password for decryption

Return tupple (bytes, bytes) (Private Key bytes, 4 byte address hash for verification)

bitcoinlib.encoding.**bip38_encrypt**(*private_hex*, *address*, *password*, *flagbyte=b'\xe0'*)

BIP0038 non-ec-multiply encryption. Returns BIP0038 encrypted private key Based on code from https://github.com/nomorecoin/python-bip38-testing

Parameters

- **private_hex** (*str*) – Private key in hex format

- **address** (*str*) – Address string
- **password** (*str*) – Required password for encryption
- **flagbyte** (*bytes*) – Flagbyte prefix for WIF

Return str BIP38 password encrypted private key

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

Convert input chars from one numeric base to another. For instance from hexadecimal (base-16) to decimal (base-10)

From and to numeric base can be any base. If base is not found in definitions an array of index numbers will be returned

Examples:

```
>>> change_base('FF', 16, 10)
255
>>> change_base('101', 2, 10)
5
```

Convert base-58 public WIF of a key to hexadecimal format

```
>>> change_base(
↪'xpub661MyMwAqRbcFnkbk13gaJba22ibnEdJS7KAMY99C4jBBHMxWaCBSTrTinNTc9G5LTFtUqbLpWnzY5yPTNEF9u8sB
↪', 58, 16)

↪'0488b21e000000000000000000d3cc6702f48bf618f3f14cce5ee2cacf3f70933345ee4710af6fa4a330cc7d503c
↪'
```

Convert base-58 address to public key hash: '00' + length '21' + 20 byte key

```
>>> change_base('142Zp9WZn9Fh4MV8F3H5Dv4Rbg7Ja1sPWZ', 58, 16)
'0021342f229392d7c9ed82c932916cee6517fbc9a2487cd97a'
```

Convert to 2048-base, for example a Mnemonic word list. Will return a list of integers

```
>>> change_base(100, 16, 2048)
[100]
```

Parameters

- **chars** (*any*) – Input string
- **base_from** (*int*) – Base number or name from input. For example 2 for binary, 10 for decimal and 16 for hexadecimal
- **base_to** (*int*) – Base number or name for output. For example 2 for binary, 10 for decimal and 16 for hexadecimal
- **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*)
Extract content from DER encoded string: Convert DER encoded signature to signature string.

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*) – 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.**der_encode_sig**(*r*, *s*)
Create DER encoded signature string with signature r and s value.

Parameters

- **r** (*int*) – r value of signature
- **s** (*int*) – s value of signature

Return bytes

bitcoinlib.encoding.**double_sha256**(*string*, *as_hex=False*)
Get double SHA256 hash of string

Parameters

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

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

```
>>> int_to_varbyteint(10000).hex()
'fd1027'
```

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, str*) – string value

Returns string

bitcoinlib.encoding.**normalize_var**(*var*, *base=256*)
For Python 2 convert variable to string

For Python 3 convert to bytes

Convert decimals to integer type

Parameters

- **var** (*str, byte*) – 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

Wrapper for the *pubkeyhash_to_addr_base58()* and *pubkeyhash_to_addr_bech32()* method

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

```
>>> pubkeyhash_to_addr_base58('21342f229392d7c9ed82c932916cee6517fbc9a2')
'142Zp9WZn9Fh4MV8F3H5Dv4Rbg7Ja1sPWZ'
```

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

```
>>> pubkeyhash_to_addr_bech32('21c1bc695a56f47991e95ff26856e50f78d3c118')
'bc1qy8qmc6262m68ny0ftlexs4h9paud8sgce3sf84'
```

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*) – 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_bytes**(*string*, *unhexlify=True*)

Convert string, hexadecimal string to bytes

Parameters

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

Returns Bytes var

bitcoinlib.encoding.**to_hexstring**(*string*)

Convert bytes, string to a hexadecimal string. Use instead of built-in hex() method if format of input string is not known.

```
>>> to_hexstring(b'\x12\xaa\xdd')
'12aadd'
```

Parameters **string** (*bytes, 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

```
>>> varbyteint_to_int(bytes.fromhex('fd1027'))
(10000, 3)
```

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

Return (int, int) tuple wit converted integer and size

bitcoinlib.encoding.**varstr**(*string*)

Convert string to variably sized string: Bytestring preceded with length byte

```
>>> varstr(to_bytes('546869732073747269e67206861732061206c656e677468206f66203330
↪')).hex()
'1e546869732073747269e67206861732061206c656e677468206f66203330'
```

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

Return bytes varstring

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.

Returns 'base58' for legacy and p2sh-segwit witness type and 'bech32' for segwit

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.

```
>>> script_type_default('segwit', locking_script=True)
'p2wpkh'
```

Parameters

- **witness_type** (*str*) – Witness type used: standard, p2sh-segwit or segwit
- **multisig** (*bool*) – Multi-signature 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

8.22.1.3 Module contents

8.23 Script types

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

They are defined in main.py

8.23.1 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

8.23.2 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

8.23.3 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

NINE

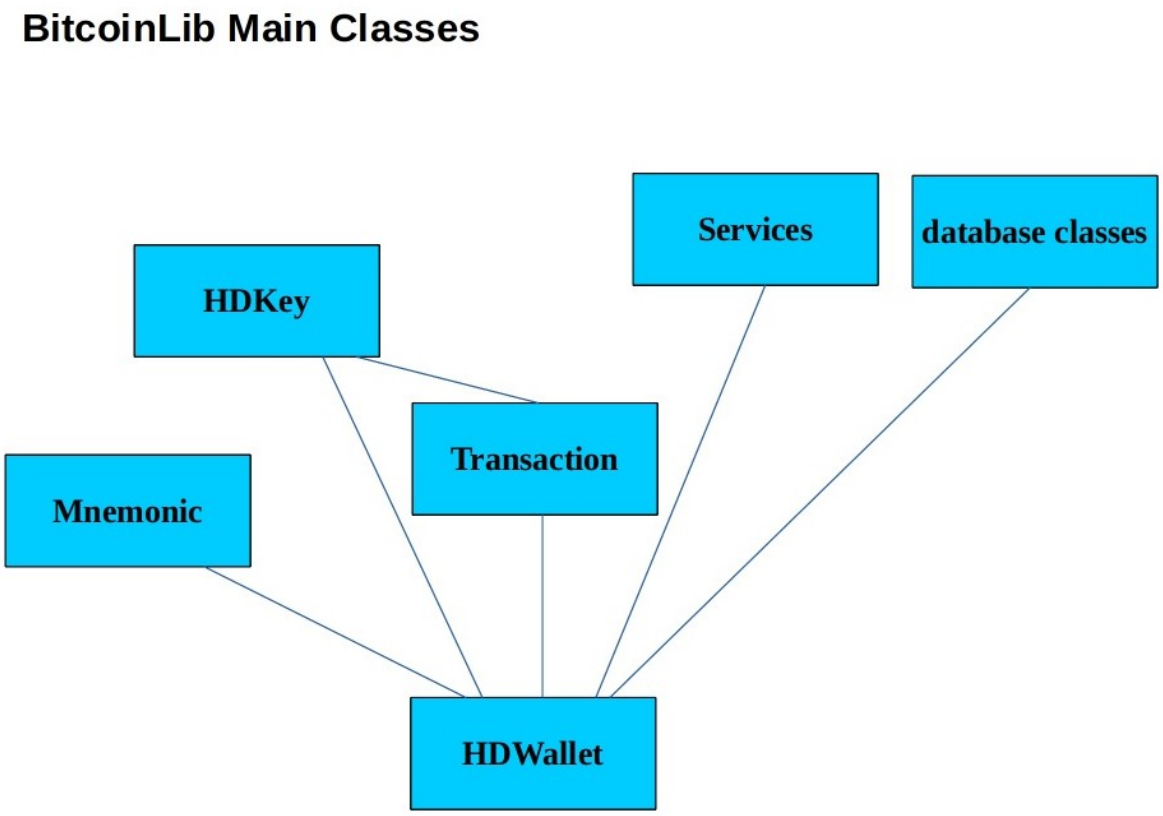
DISCLAIMER

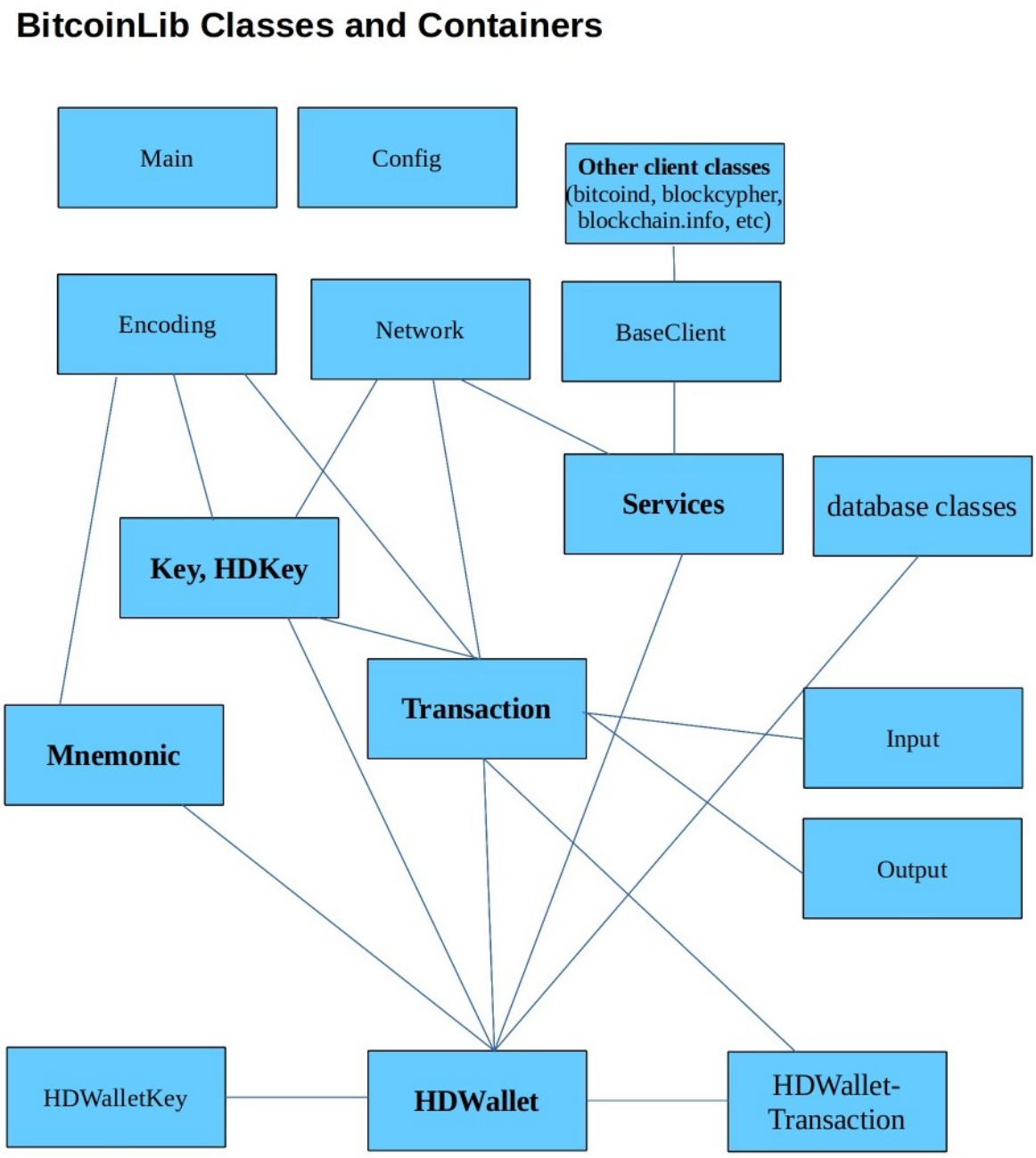
This library is still in development, please use at your own risk and test sufficiently before using it in a production environment.

CHAPTER

TEN

SCHEMATIC OVERVIEW




BitcoinLib Classes and Containers
Main
Config
Other client classes
(bitcoind, blockcypher,
blockchain.info, etc)
Encoding
Network
BaseClient
Services
database classes
Key, HDKey
Transaction
Mnemonic
Input
Output
HDWalletKey
HDWallet
HDWallet-
Transaction

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y