

---

# Bitcoinlib Documentation

*Release 0.6.1*

**Lennart (mccwdev)**

**Oct 08, 2021**



## MANUALS

<b>1</b>	<b>Wallet</b>	<b>3</b>
<b>2</b>	<b>Segregated Witness Wallet</b>	<b>5</b>
<b>3</b>	<b>Wallet from passphrase with accounts and multiple currencies</b>	<b>7</b>
<b>4</b>	<b>Multi Signature Wallets</b>	<b>9</b>
<b>5</b>	<b>Command Line Tool</b>	<b>11</b>
<b>6</b>	<b>Service providers</b>	<b>13</b>
<b>7</b>	<b>Other Databases</b>	<b>15</b>
<b>8</b>	<b>More examples</b>	<b>17</b>
8.1	Install, Update and Tweak BitcoinLib . . . . .	17
8.1.1	Installation . . . . .	17
8.1.1.1	Install with pip . . . . .	17
8.1.1.2	Install from source . . . . .	17
8.1.1.3	Package dependencies . . . . .	18
8.1.1.4	Other requirements Linux . . . . .	18
8.1.1.5	Development environment . . . . .	18
8.1.1.6	Other requirements Windows . . . . .	19
8.1.2	Update Bitcoinlib . . . . .	19
8.1.3	Troubleshooting . . . . .	19
8.1.4	Using library in other software . . . . .	20
8.1.5	Tweak BitcoinLib . . . . .	20
8.2	Command Line Wallet . . . . .	20
8.2.1	Create wallet . . . . .	21
8.2.2	Generate / show receive addresses . . . . .	21
8.2.3	Send funds / create transaction . . . . .	21
8.2.4	Restore wallet with passphrase . . . . .	21
8.2.5	Options Overview . . . . .	22
8.3	Add a new Service Provider . . . . .	24
8.3.1	Steps to add a new provider . . . . .	24
8.4	How to connect bitcoinlib to a bitcoin node . . . . .	25
8.4.1	Bitcoin node settings . . . . .	25
8.4.2	Connect using config files . . . . .	26
8.4.3	Connect using provider settings . . . . .	26
8.4.4	Connect using base_url argument . . . . .	26
8.4.5	Please note: Using a remote bitcoind server . . . . .	27

8.5	Using MySQL or PostgreSQL databases . . . . .	27
8.5.1	Using MySQL database . . . . .	27
8.5.2	Using PostgreSQL database . . . . .	27
8.6	Using SQLCipher encrypted database . . . . .	28
8.7	10 Tips to Increase Privacy and Security . . . . .	28
8.8	Caching . . . . .	29
8.8.1	What is cached? . . . . .	29
8.8.2	Using other databases . . . . .	29
8.8.3	Disable caching . . . . .	29
8.8.4	Troubleshooting . . . . .	29
8.8.4.1	Nothing is cached, what is the problem? . . . . .	29
8.8.4.2	I get incomplete or incorrect results! . . . . .	30
8.9	bitcoinlib.keys module . . . . .	30
8.10	bitcoinlib.transactions module . . . . .	47
8.11	bitcoinlib.scripts module . . . . .	60
8.12	bitcoinlib.wallets module . . . . .	66
8.13	bitcoinlib.mnemonic module . . . . .	91
8.14	bitcoinlib.networks module . . . . .	93
8.15	bitcoinlib.blocks module . . . . .	95
8.16	bitcoinlib.values module . . . . .	100
8.17	bitcoinlib.services.services module . . . . .	104
8.18	bitcoinlib.services package . . . . .	110
8.18.1	Submodules . . . . .	110
8.18.1.1	bitcoinlib.services.authproxy module . . . . .	110
8.18.1.2	bitcoinlib.services.baseclient module . . . . .	111
8.18.1.3	bitcoinlib.services.bcoin module . . . . .	111
8.18.1.4	bitcoinlib.services.bitaps module . . . . .	111
8.18.1.5	bitcoinlib.services.bitcoind module . . . . .	112
8.18.1.6	bitcoinlib.services.bitcoinlibtest module . . . . .	113
8.18.1.7	bitcoinlib.services.bitgo module . . . . .	113
8.18.1.8	bitcoinlib.services.blockchaininfo module . . . . .	114
8.18.1.9	bitcoinlib.services.blockchair module . . . . .	114
8.18.1.10	bitcoinlib.services.blockcypher module . . . . .	115
8.18.1.11	bitcoinlib.services.blocksmurfer module . . . . .	115
8.18.1.12	bitcoinlib.services.blockstream module . . . . .	116
8.18.1.13	bitcoinlib.services.chainso module . . . . .	116
8.18.1.14	bitcoinlib.services.coinfees module . . . . .	117
8.18.1.15	bitcoinlib.services.cryptoid module . . . . .	117
8.18.1.16	bitcoinlib.services.dashd module . . . . .	117
8.18.1.17	bitcoinlib.services.dogecoin module . . . . .	118
8.18.1.18	bitcoinlib.services.insightdash module . . . . .	119
8.18.1.19	bitcoinlib.services.litecoinblockexplorer module . . . . .	119
8.18.1.20	bitcoinlib.services.litecoind module . . . . .	120
8.18.1.21	bitcoinlib.services.litecoreio module . . . . .	121
8.18.1.22	bitcoinlib.services.smartbit module . . . . .	121
8.18.2	Module contents . . . . .	122
8.19	bitcoinlib.config package . . . . .	122
8.19.1	Submodules . . . . .	122
8.19.1.1	bitcoinlib.config.config module . . . . .	122
8.19.1.2	bitcoinlib.config.opcodes module . . . . .	122
8.19.1.3	bitcoinlib.config.secp256k1 module . . . . .	122
8.19.2	Module contents . . . . .	122
8.20	bitcoinlib.db module . . . . .	122
8.21	bitcoinlib.db_cache module . . . . .	129

8.22	Classes Overview . . . . .	133
8.23	bitcoinlib . . . . .	135
8.23.1	bitcoinlib package . . . . .	135
8.23.1.1	Subpackages . . . . .	135
8.23.1.2	Submodules . . . . .	135
8.23.1.3	Module contents . . . . .	141
8.24	Script types . . . . .	141
8.24.1	Locking scripts . . . . .	141
8.24.2	Unlocking scripts . . . . .	142
8.24.3	Bitcoinlib script support . . . . .	142
<b>9</b>	<b>Disclaimer</b>	<b>143</b>
<b>10</b>	<b>Schematic overview</b>	<b>145</b>
<b>11</b>	<b>Indices and tables</b>	<b>147</b>
	<b>Python Module Index</b>	<b>149</b>
	<b>Index</b>	<b>151</b>



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.





## 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
```



## 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
```



## 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()
```



## 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(
    ↳ 'tprv8ZgxMBicQKsPd1Q44tfDiZC98iYouKRC2CzjT3HGt1yYw2zuX2awTotzGAZQEAU9bi2M5MCj8iedP9MREPjUgpDEBwBgGi2C'
    ↳ '
        '5zNYeiX8', network=NETWORK)
k2 = HDKey(
    ↳ 'tprv8ZgxMBicQKsPeUbMS6kswJc11zgVEXUnUZuGo3bF6bBrAg1ieFfUdPc9UHqbD5HcXizThrcKike1c4z6xHrz6MWGwy8L6YKV'
    ↳ '
        '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()
```





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



## **SERVICE PROVIDERS**

Communicates with pools of bitcoin service providers to retrieve 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 Satoshi per Kb for confirmation within 5 blocks

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



## 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](http://bitcoinlib.readthedocs.io/en/latest/_static/manuals.databases.html) for instructions on how to use MySQL or PostgreSQL.



## 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 pycrypt library instead of scrypt. The pycrypt 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 separately:

```
$ 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 pycrypt instead of scrypt. Pycrypt is a pure Python scrypt password-based key derivation library. It works but it is slow when using BIP38 password protected keys.

```
$ pip install pycrypt
```

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 unspent 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 ...]]	
--	--

(continues on next page)

(continued from previous page)

```

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 advised but can be used for testing. Set
    to 256 bits for more future proof passphrases
--network NETWORK, -n NETWORK
    Specify 'bitcoin', 'litecoin', 'testnet' or other
    supported network
--database DATABASE, -d DATABASE
    Name of specific database file to use
--create-from-key KEY, -c KEY
    Create a new wallet from specified key
--create-multisig [NUMBER_OF_SIGNATURES_REQUIRED [KEYS ...]], -m [NUMBER_OF_SIGNATURES_
REQUIRED [KEYS ...]]
    Specify number of signatures required followed by a
    list of signatures. Example: -m 2 tprv8ZgxMBicQKsPd1Q4
    4tfDiZC98iYouKRC2CzjT3HGt1yYw2zuX2awTotzGAZQEAU9bi2M5M
    Cj8iedP9MREPjUgpDEBwBgGi2C8eK5zNYeiX8 tprv8ZgxMBicQKsP
    eUbMS6kswJc1l1zgVEXUnUZuGo3bF6bBrAg1ieFfUdPc9UHQbD5HcXi
    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 retrieve 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",
    "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)
```

(continues on next page)

(continued from previous page)

```
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 BitcoinClient instance.

```
from bitcoinlib.services.bitcoind import BitcoinClient

bdc = BitcoinClient.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": "BitcoinClient",
    "url": "http://user:password@server_url:18332",
    "api_key": "",
    "priority": 11,
    "denominator": 1000000000
  }
}
```

## 8.4.4 Connect using base\_url argument

Another options is to pass the 'base\_url' argument to the BitcoinClient 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 BitcoinClient

base_url = 'http://user:password@server_url:18332'
bdc = BitcoinClient(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 `rpccallowip` 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()`, `gettxos()` 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](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 `gettxos()` 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 from the address.

**Parameters**

- **address** (*str*) – Address to import
- **compressed** (*bool*) – Is key compressed or not, default is None
- **encoding** (*str*) – Address encoding. Default is base58 encoding, for native segwit addresses specify bech32 encoding. Leave empty to derive from address
- **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. Will 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**

**classmethod parse**(*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 from the address.

```
>>> addr = Address.parse(
↳ '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. Will 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, a chain with only 32 0-bytes will be used.

```
>>> private_hex = '221ff330268a9bb5549a02c801764cffbc79d5c26f4041b26293a425fd5b557c'
>>> k = HDKey(private_hex)
>>> k
<HDKey(public_
  ↳ hex=0363c152144dcd5253c1216b733fdc6eb8a94ab2cd5caa8ead5e59ab456ff99927, wif_
  ↳ public=xpub661MyMwAqRbcEYS8w7XLSVeEsBXy79zSzH1J8vCdxAZningWLDn3zgtU6SmyphHzZG2cYrwpGkWJqRxS6EAW77g
  ↳ 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 addresses

**Return HDKey**

**address**(*compressed=None, prefix=None, script\_type=None, encoding=None*)

Get address derived from public key

```
>>> wif =
↳ 'xpub661MyMwAqRbcFcXi3aM3fVdd42FGDSdufhrr5tdobiPjMrPUykFMTdaFEr7yoy1xxeiFDY8kh2k4h9N77MY6rk1'
↳
>>> 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(
  ↳ 'zprvAWgYBBk7JR8GjAHfvjhGLKFGUJNcnPtKnryWfstePYJc4SVFYbaFk3Fpqn9dSmtPLKrPWB7WzsgzZzFiB1Qnhzoj'
  ↳ )
>>> k.bip38_encrypt('my-secret-password')
'6PYUAKyDY07Q6sSJ3ZY04EFwFTmkUES2mdvsMNBS0n5QyXPmeogxfumfW'
```

**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
>>> 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 four 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 separated 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 addresses

**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()  
  
↳ 'xpub6CjFexgdDZEtHdW7V4LT8wS9rtG3m187pM9qhTpoZdViFhSv3tW9sWonQNtFN1TCkRGAQGKj1UC2ViHTqb7vJV3'  
↳ '
```

**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 =
↳ 'xprv9s21ZrQH143K4LvcS93AHEZh7gBiYND6zMoRiZQGL5wqbpCU2KJDY87Txuv9dduk9hAcsL76F8b5JKzDREf8EmX'
↳
>>> k = HDKey(wif)
>>> k.subkey_for_path("m/44'/0'/0'/0/2")
<HDKey(public_
↳ hex=03004331ca7f0dcdd925abc4d0800a0d4a0562a02c257fa39185c55abdfc4f0c0c, wif_
↳ public=xpub6GyQoEbMUNwulLnbiCSaD8wLrcjyRCEQA8tNsFCH4pvnCbuWSZkSB6LUNe89YsCBTg1Ncs7vHJBjMvw2Q
↳ 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()

↳ 'xpub661MyMwAqRbcEYS8w7XLSVeEsBXy79zSzH1J8vCdxAZningWLdN3zgtU6SmypHzZG2cYrwpGkWJqRxS6EAW77gd'
↳
```

**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, strict=True*)

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('cNUpWJbC1hVJtyxyV4bVAnb4uJ7FPhr82geolvnoA29XWkeiiCQn')
>>> k.secret
12127227708610754620337553985245292396444216111803695028419544944213442390363
```

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

```
>>> k2 = Key('6PYM8wAnnAK5mHYoF7zqj88y5HtK7eiPeqPdu4WnYEFkYKEEoMFEVfuDg', password=
↳ 'test', network='testnet')
>>> k2.secret
12127227708610754620337553985245292396444216111803695028419544944213442390363
```

**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
- **strict** (*bool*) – Raise BKeyError if key is invalid. Default is True. Set to False if you're parsing blockchain transactions, as some may contain invalid keys, but the transaction is/was still valid.

**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('cNUpWJbC1hVJtyxyV4bVAnb4uJ7FPhr82geolvnoA29XWkeiiCQn')
>>> 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**

**hex**()

**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()

↪ '3044022015f9d39d8b53c68c7549d5dc4cbdafe1c71bae3656b93a02d2209e413d9bbcd00220615cf626da0a81945a70'
↪ ''
```

Initialize Signature object with provided r and r value

```

>>> r = 32979225540043540145671192266052053680452913207619328973512110841045982813493
>>> s = 12990793585889366641563976043319195006380846016310271470330687369836458989268
>>> sig = Signature(r, s)
>>> sig.hex()

'48e994862e2cdb372149bad9d9894cf3a5562b4565035943efe0acc502769d351cb88752b5fe8d70d85f3541046df61'

```

### 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*, *include\_hash\_type=True*)

Get DER encoded signature

### Parameters

- **as\_hex** (*bool*) – Output as hexstring
- **include\_hash\_type** (*bool*) – Include hash\_type byte at end of signatures as used in raw scripts. Default is True

### 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()

'48e994862e2cdb372149bad9d9894cf3a5562b4565035943efe0acc502769d351cb88752b5fe8d70d85f3541046df61'
>>> 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****from\_str()**

staticmethod(function) -> method

Convert a function to be a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class C: @staticmethod def f(arg1, arg2, ...):  
    ...
```

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. For a more advanced concept, see the classmethod builtin.

**hex()**

Signature r and s value as single hexadecimal string

**Return hexstring**

**classmethod parse**(*signature*, *public\_key=None*)

**static parse\_bytes**(*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*) – 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**

**classmethod parse\_hex**(*signature*, *public\_key=None*)

**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 =
↳ '48e994862e2cdb372149bad9d9894cf3a5562b4565035943efe0acc502769d351cb88752b5fe8d70d85f3541046'
↳
>>> sig = Signature.parse_hex(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')
'bc1q4pwmstmw8q80nxtxud2h42lev9xzcjqwqyq7t'
```

**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('L4dTuJf2ceEdWDvCPsLhYf8GiuYqXtqfbcKdC21BPDvEM1ykJRC')
'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*, *None*) – Optional network. Method raises `BKeyError` if keys belongs to another network
- **kf\_networks** (*list*, *None*) – 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*, *None*) – 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\x00. |7\x03\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. Will 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(
↳ 'Zpub6vZyhw1ShkEwNxtqfjk7jiwoEbZYMJdbWLHvEwo6Ns2fFc9rdQn3SerYFQXyxtZYbA8a1d83shW3g4WbsnVsyms2L8m7
↳ ')
{'format': 'hdkey_public', 'networks': ['bitcoin'], 'is_private': False, 'script_
↳ types': ['p2wsh'], 'witness_types': ['segwit'], 'multisig': [True]}
```

(continues on next page)

(continued from previous page)

**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()

↪ '30440220792f04c5ba654e27eb636ceb7804c5590051dd77da8b80244f1fa8dfbfff369b302204ba03b039c808a0403d0'
↪ ''
```

### 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 =
↪ '48e994862e2cdb372149bad9d9894cf3a5562b4565035943efe0acc502769d351cb88752b5fe8d70d85f3541046df61'
↪ ''
>>> 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=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, strict=True, 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 str*) – 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
- **strict** (*bool*) – Raise exception when input is malformed, incomplete or not understood
- **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

**classmethod parse**(*raw*, *witness\_type*='segwit', *index\_n*=0, *strict*=True, *network*='bitcoin')

Parse raw BytesIO string and return Input object

**Parameters**

- **raw** (*BytesIO*) – Input
- **witness\_type** (*str*) – Specify witness/signature position: ‘segwit’ or ‘legacy’. Derived from script if not specified.
- **index\_n** (*int*) – Index number of input
- **strict** (*bool*) – Raise exception when input is malformed, incomplete or not understood
- **network** (*str*, [Network](#)) – Network, leave empty for default

**Return Input**

**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, *strict*=True, *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
- **strict** (*bool*) – Raise exception when output is malformed, incomplete or not understood
- **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

#### classmethod parse(*raw*, *output\_n=0*, *strict=True*, *network='bitcoin'*)

Parse raw BytesIO string and return Output object

#### Parameters

- **raw** (*BytesIO*) – raw output stream
- **output\_n** (*int*) – Output number of Transaction output
- **strict** (*bool*) – Raise exception when output is malformed, incomplete or not understood
- **network** (*str*, *Network*) – Network, leave empty for default network

#### Return Output

#### set\_locktime\_relative(*locktime*)

Relative timelocks with CHECKSEQUENCEVERIFY (CSV) as defined in BIP112 :param locktime: :return:

#### 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**

```
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*, *strict=True*)

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 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 str*) – 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
- **strict** (*bool*) – Raise exception when input is malformed or incomplete

**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*, *strict=True*)

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
- **strict** (*bool*) – Raise exception when output is malformed or incomplete

**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 weight units and vsize for this Transaction. Weight units are used to determine fee.

**Return int**

**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:  $\text{math.ceil}(((\text{est\_size} - \text{witness\_size}) * 3 + \text{est\_size}) / 4)$

**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

REPLACED BY THE PARSE() METHOD

**Parameters**

- **rawtx** (*bytes*, *str*) – Raw transaction string
- **network** (*str*, [Network](#)) – Network, leave empty for default
- **check\_size** (*bool*) – Check if no 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 save fees and increase privacy.

**Parameters** **transaction** ([Transaction](#)) – The transaction to be merged

**classmethod parse**(*rawtx, strict=True, network='bitcoin'*)

Parse a raw transaction and create a Transaction object

**Parameters**

- **rawtx** (*BytesIO, bytes, str*) – Raw transaction string
- **strict** (*bool*) – Raise exception when transaction is malformed, incomplete or not understood
- **network** (*str, Network*) – Network, leave empty for default network

**Return Transaction****classmethod parse\_bytes**(*rawtx, strict=True, network='bitcoin'*)

Parse a raw bytes transaction and create a Transaction object. Wrapper for the [parse\\_bytesio\(\)](#) method

**Parameters**

- **rawtx** (*bytes*) – Raw transaction hexadecimal string
- **strict** (*bool*) – Raise exception when transaction is malformed, incomplete or not understood
- **network** (*str, Network*) – Network, leave empty for default network

**Return Transaction****classmethod parse\_bytesio**(*rawtx, strict=True, network='bitcoin'*)

Parse a raw transaction and create a Transaction object

**Parameters**

- **rawtx** (*BytesIO*) – Raw transaction string
- **strict** (*bool*) – Raise exception when transaction is malformed, incomplete or not understood
- **network** (*str, Network*) – Network, leave empty for default network

**Return Transaction**

**classmethod** `parse_hex(rawtx, strict=True, network='bitcoin')`

Parse a raw hexadecimal transaction and create a Transaction object. Wrapper for the `parse_bytesio()` method

**Parameters**

- **rawtx** (*str*) – Raw transaction hexadecimal string
- **strict** (*bool*) – Raise exception when transaction is malformed, incomplete or not understood
- **network** (*str*, `Network`) – Network, leave empty for default network

**Return Transaction**

**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

**Returns****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 0xffffffff

**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()**

Get witness data for all inputs of this transaction

**Return bytes**

**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

**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 empty 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.scripts module

```
class bitcoinlib.scripts.Script(commands=None, message=None, script_types="", is_locking=True,
                                keys=None, signatures=None, blueprint=None, tx_data=None,
                                public_hash=b"", sigs_required=None, redeemscript=b"", hash_type=1)
```

Bases: object

Create a Script object with specified parameters. Use parse() method to create a Script from raw hex

```
>>> s = Script([op.op_2, op.op_4, op.op_add])
>>> s
<Script([op.op_2, op.op_4, op.op_add])>
>>> s.blueprint
[82, 84, 147]
>>> s.evaluate()
True
```

Stack is empty now, because evaluate pops last item from stack and check if is non-zero >>> s.stack []

### Parameters

- **commands** (*list*) – List of script language commands
- **message** (*bytes*) – Signed message to verify, normally a transaction hash. Used to validate script
- **script\_types** (*list of str*) – List of script\_types as defined in SCRIPT\_TYPES
- **is\_locking** (*bool*) – Is this a locking script (Output), otherwise unlocking (Input)
- **keys** (*list of Key*) – Provide list of keys to create script
- **signatures** (*list of Signature*) – Provide list of signatures to create script
- **blueprint** (*list of str*) – Simplified version of script, normally generated by Script object
- **tx\_data** (*dict*) – Dictionary with extra information needed to verify script. Such as ‘redeemscript’ for multisignature scripts and ‘blockcount’ for time locked scripts
- **public\_hash** (*bytes*) – Public hash of key or redeemscript used to create scripts
- **sigs\_required** (*int*) – Nubmer of signatures required to create multisig script
- **redeemscript** (*bytes*) – Provide redeemscript to create a new (multisig) script
- **hash\_type** (*int*) – Specific script hash type, default is SIGHASH\_ALL

### property blueprint

**evaluate**(*message=None, tx\_data=None*)

Evaluate script, run all commands and check if it is valid

```
>>> s = Script([op.op_2, op.op_4, op.op_add])
>>> s
<Script([op.op_2, op.op_4, op.op_add])>
>>> s.blueprint
[82, 84, 147]
>>> s.evaluate()
True
```

```
>>> lock_script = bytes.fromhex(
↳ '76a914f9cc73824051cc82d64a716c836c54467a21e22c88ac')
>>> unlock_script = bytes.fromhex(
↳ '483045022100ba2ec7c40257b3d22864c9558738eea4d8771ab97888368124e176fdd6d7cd8602200f47c8d0c43')
↳ ')
>>> s = Script.parse_bytes(unlock_script + lock_script)
>>> transaction_hash = bytes.fromhex(
↳ '12824db63e7856d00ee5e109fd1c26ac8a6a015858c26f4b336274f6b52da1c3')
>>> s.evaluate(message=transaction_hash)
True
```

#### Parameters

- **message** (*bytes*) – Signed message to verify, normally a transaction hash. Leave empty to use `Script.message`. If supplied `Script.message` will be ignored.
- **tx\_data** – Dictionary with extra information needed to verify script. Such as ‘redeem-script’ for multisignature scripts and ‘blockcount’ for time locked scripts. Leave empty to use `Script.tx_data`. If supplied `Script.tx_data` will be ignored

**Return bool** Valid or not valid

**classmethod** `parse(script, message=None, tx_data=None, strict=True, _level=0)`

Parse raw script and return `Script` object. Extracts script commands, keys, signatures and other data.

Wrapper for the `parse_bytesio()` method. Convert hexadecimal string or bytes script to `BytesIO`.

```
>>> Script.parse('76a914af8e14a2ced715c363b3a72b55b59a31e2acac988ac')
<Script([op.op_dup, op.op_hash160, data-20, op.op_equalverify, op.op_checksig])>
```

#### Parameters

- **script** (*BytesIO, bytes, str*) – Raw script to parse in bytes, `BytesIO` or hexadecimal string format
- **message** (*bytes*) – Signed message to verify, normally a transaction hash
- **tx\_data** (*dict*) – Dictionary with extra information needed to verify script. Such as ‘redeemscript’ for multisignature scripts and ‘blockcount’ for time locked scripts
- **strict** (*bool*) – Raise exception when script is malformed, incomplete or not understood. Default is `True`
- **\_level** (*int*) – Internal argument used to avoid recursive depth

**Return Script**

**classmethod** `parse_bytes(script, message=None, tx_data=None, strict=True, _level=0)`

Parse raw script and return `Script` object. Extracts script commands, keys, signatures and other data.

Wrapper for the `parse_bytesio()` method. Convert bytes script to `BytesIO`.

#### Parameters

- **script** (*bytes*) – Raw script to parse in bytes format
- **message** (*bytes*) – Signed message to verify, normally a transaction hash
- **tx\_data** (*dict*) – Dictionary with extra information needed to verify script. Such as ‘redeemscript’ for multisignature scripts and ‘blockcount’ for time locked scripts

- **strict** (*bool*) – Raise exception when script is malformed or incomplete
- **\_level** (*int*) – Internal argument used to avoid recursive depth

#### Return Script

**classmethod** `parse_bytesio`(*script*, *message=None*, *tx\_data=None*, *strict=True*, *\_level=0*)

Parse raw script and return Script object. Extracts script commands, keys, signatures and other data.

#### Parameters

- **script** (*BytesIO*) – Raw script to parse in bytes, BytesIO or hexadecimal string format
- **message** (*bytes*) – Signed message to verify, normally a transaction hash
- **tx\_data** (*dict*) – Dictionary with extra information needed to verify script. Such as 'redeemscript' for multisignature scripts and 'blockcount' for time locked scripts
- **strict** (*bool*) – Raise exception when script is malformed, incomplete or not understood. Default is True
- **\_level** (*int*) – Internal argument used to avoid recursive depth

#### Return Script

**classmethod** `parse_hex`(*script*, *message=None*, *tx\_data=None*, *strict=True*, *\_level=0*)

Parse raw script and return Script object. Extracts script commands, keys, signatures and other data.

Wrapper for the `parse_bytesio()` method. Convert hexadecimal string script to BytesIO.

```
>>> Script.parse_hex('76a914af8e14a2cecd715c363b3a72b55b59a31e2acac988ac')
<Script([op.op_dup, op.op_hash160, data-20, op.op_equalverify, op.op_checksigt])>
```

#### Parameters

- **script** (*str*) – Raw script to parse in hexadecimal string format
- **message** (*bytes*) – Signed message to verify, normally a transaction hash
- **tx\_data** (*dict*) – Dictionary with extra information needed to verify script. Such as 'redeemscript' for multisignature scripts and 'blockcount' for time locked scripts
- **strict** (*bool*) – Raise exception when script is malformed, incomplete or not understood. Default is True
- **\_level** (*int*) – Internal argument used to avoid recursive depth

#### Return Script

#### property raw

**serialize**()

Serialize script. Return all commands and data as bytes

```
>>> s = Script.parse_hex('76a914af8e14a2cecd715c363b3a72b55b59a31e2acac988ac')
>>> s.serialize().hex()
'76a914af8e14a2cecd715c363b3a72b55b59a31e2acac988ac'
```

#### Return bytes

**serialize\_list**()

Serialize script and return commands and data as list

```
>>> s = Script.parse_hex('76a9')
>>> s.serialize_list()
[b'v', b'\xa9']
```

#### Return list of bytes

**exception** bitcoinlib.scripts.**ScriptError**(msg="")

Bases: Exception

Handle Key class Exceptions

**class** bitcoinlib.scripts.**Stack**(iterable=(),/)

Bases: list

The Stack object is a child of the Python list object with extra operational (OP) methods. The operations as used in the Script language can be used to manipulate the stack / list.

For documentation of the op-methods you could check <https://en.bitcoin.it/wiki/Script>

**as\_ints()**

Return the Stack as list of integers

```
>>> st = Stack.from_ints([1, 2])
>>> st.as_ints()
[1, 2]
```

#### Return list of int

**classmethod** **from\_ints**(list\_ints)

Create a Stack item with a list of integers.

```
>>> Stack.from_ints([1, 2])
[b'\x01', b'\x02']
```

#### Parameters list\_ints –

#### Returns

**is\_arithmetic**(items=1)

Check if top stack item is or last stack are arithmetic and has no more than 4 bytes

#### Return bool

**op\_0notequal()**

**op\_1add()**

**op\_1sub()**

**op\_2drop()**

**op\_2dup()**

**op\_2over()**

**op\_2rot()**

**op\_2swap()**

**op\_3dup()**

**op\_abs()**

**op\_add()**

**op\_booland()**

**op\_boolor()**

**op\_checklocktimeverify**(*sequence, tx\_locktime*)

Implements CHECKLOCKTIMEVERIFY opcode (CLTV) as defined in BIP65.

CLTV is an absolute timelock and is added to an output locking script. It locks an output until a certain time or block.

**Parameters**

- **sequence** (*int*) – Sequence value from the transaction. Must be 0xffffffff to be valid
- **tx\_locktime** (*int*) – The nLocktime value from the transaction in blocks or as Median Time Past timestamp

**Return bool**

**op\_checkmultisig**(*message, data=None*)

**op\_checkmultisigverify**(*message, data=None*)

**op\_checksequenceverify**(*sequence, version*)

Implements CHECKSEQUENCEVERIFY opcode (CSV) as defined in BIP112

CSV is a relative timelock and is added to an output locking script. It locks an output for a certain number of blocks or time.

**Parameters**

- **sequence** (*int*) – Sequence value from the transaction
- **version** (*int*) – Transaction version. Must be 2 or higher

**Return bool**

**op\_checksig**(*message, \_=None*)

**op\_checksigverify**(*message, \_=None*)

**op\_depth()**

**op\_drop()**

**op\_dup()**

**op\_equal()**

**op\_equalverify()**

**op\_hash160()**

**op\_hash256()**

**op\_if**(*commands*)

**op\_ifdup()**

**op\_max()**

**op\_min()**

**op\_negate()**

**op\_nip()**

```
op_nop()
op_nop1()
op_nop10()
op_nop4()
op_nop5()
op_nop6()
op_nop7()
op_nop8()
op_nop9()
op_not()
op_notif(commands)
op_numequal()
op_numequalverify()
op_numgreaterthan()
op_numgreaterthanequal()
op_numlessthan()
op_numlessthanequal()
op_numnotequal()
op_over()
op_pick()
static op_return()
op_ripemd160()
op_roll()
op_rot()
op_sha1()
op_sha256()
op_size()
op_sub()
op_swap()
op_tuck()
op_verify()
op_within()
op_as_number()
    Pop the latest item from the list and decode as number
```

```
>>> st = Stack.from_ints([1, 2])
>>> st.pop_as_number()
2
```

**Return int****bitcoinlib.scripts.data\_pack**(*data*)

Add data length prefix to data string to include data in a script

**Parameters** **data** (*bytes*) – Data to be packed**Return bytes****bitcoinlib.scripts.decode\_num**(*encoded*)

Decode byte representation of number used in Script language to integer.

```
>>> decode_num(b'')
0
>>> decode_num(b'@B\x0f')
10000000
```

**Parameters** **encoded** (*bytes*) – Number as bytes**Return int****bitcoinlib.scripts.encode\_num**(*num*)

Encode number as byte used in Script language. Bitcoin specific little endian format with sign for negative integers.

```
>>> encode_num(0)
b''
>>> encode_num(1)
b'\x01'
>>> encode_num(1000)
b'\xe8\x03'
>>> encode_num(1000000)
b'@B\x0f'
```

**Parameters** **num** (*int*) – number to represent**Return bytes****bitcoinlib.scripts.get\_data\_type**(*data*)

Get type of data in script. Recognises signatures, keys, hashes or sequence data. Return 'other' if data is not recognised.

**Parameters** **data** (*bytes*) – Data part of script**Return str**

## 8.12 bitcoinlib.wallets module

**class** bitcoinlib.wallets.**Wallet**(*wallet, db\_uri=None, session=None, main\_key\_object=None*)

Bases: object

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

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



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

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

Open a wallet with given ID or name

#### Parameters

- **wallet** (*int*, *str*) – Wallet name or ID
- **db\_uri** (*str*) – URI of the database
- **session** (*sqlalchemy.orm.session.Session*) – Sqlalchemy session
- **main\_key\_object** (*HDKey*) – Pass main key object to save time

**account** (*account\_id*)

Returns wallet key of specific BIP44 account.

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

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

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

**Return** *WalletKey*

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

Get list of accounts for this wallet

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

**Return** *list of integers* List of accounts IDs

**addresslist** (*account\_id=None, used=None, network=None, change=None, depth=None, key\_id=None*)

Get list of addresses defined in current wallet. Wrapper for the `keys()` methods.

Use `keys_addresses()` method to receive full key objects

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> w.addresslist()[0]
'16QaHuFkfuebXGcYHnehRXBBX7RG9NbtLg'
```

#### Parameters

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

**Return** *list of str* List of address strings

**as\_dict** (*include\_private=False*)

Return wallet information in dictionary format

**Parameters** **include\_private** (*bool*) – Include private key information in dictionary

**Return** *dict*

**as\_json**(*include\_private=False*)

Get current key as json formatted string

**Parameters** **include\_private** (*bool*) – Include private key information in JSON

**Return** **str**

**balance**(*account\_id=None, network=None, as\_string=False*)

Get total of unspent outputs

**Parameters**

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

**Return** **float, str** Key balance

**balance\_update\_from\_serviceprovider**(*account\_id=None, network=None*)

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

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

**Parameters**

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

**Return** **int** Total balance

**classmethod create**(*name, keys=None, owner="", network=None, account\_id=0, purpose=0, scheme='bip32', sort\_keys=True, password="", witness\_type=None, encoding=None, multisig=None, sigs\_required=None, cosigner\_id=None, key\_path=None, db\_uri=None*)

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

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

```
>>> if wallet_delete_if_exists('create_legacy_wallet_test'): pass
>>> w = Wallet.create('create_legacy_wallet_test')
>>> w
<Wallet(name=create_legacy_wallet_test, db_uri="None")>
```

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

```
>>> if wallet_delete_if_exists('create_legacy_multisig_wallet_test'): pass
>>> w = Wallet.create('create_legacy_multisig_wallet_test', keys=[HDKey(),
↳ HDKey().public()])
```

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

```
>>> if wallet_delete_if_exists('create_segwit_wallet_test'): pass
>>> w = Wallet.create('create_segwit_wallet_test', witness_type='segwit')
```

Use a masterkey WIF when creating a wallet:

```
>>> wif =
↳ 'xprv9s21ZrQH143K3cxbMVswDTYgAc9CeXABQjCD9zmXCpXw4MxN93LanEARbBmV3utHZS9Db4FX1C1RbC5KSNAjQ5W
↳ '
>>> if wallet_delete_if_exists('bitcoinlib_legacy_wallet_test', force=True):
↳ pass
>>> w = Wallet.create('bitcoinlib_legacy_wallet_test', wif)
>>> w
<Wallet(name=bitcoinlib_legacy_wallet_test, db_uri="None")>
>>> # Add some test utxo data:
>>> if w.utxo_add('16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg', 1000000000,
↳ '748799c9047321cb27a6320a827f1f69d767fe889c14bf11f27549638d566fe4', 0): pass
```

Please mention account\_id if you are using multiple accounts.

### Parameters

- **name** (*str*) – Unique name of this Wallet
- **keys** (*str*, *bytes*, *int*) – Masterkey to or list of keys to use for this wallet. Will be automatically created if not specified. One or more keys are obligatory for multisig wallets. Can contain all key formats accepted by the HDKey object, a HDKey object or BIP39 passphrase
- **owner** (*str*) – Wallet owner for your own reference
- **network** (*str*) – Network name, use default if not specified
- **account\_id** (*int*) – Account ID, default is 0
- **purpose** (*int*) – BIP43 purpose field, will be derived from witness\_type and multisig by default
- **scheme** (*str*) – Key structure type, i.e. BIP32 or single
- **sort\_keys** (*bool*) – Sort keys according to BIP45 standard (used for multisig keys)
- **password** (*str*) – Password to protect passphrase, only used if a passphrase is supplied in the 'key' argument.
- **witness\_type** (*str*) – Specify witness type, default is 'legacy'. Use 'segwit' for native segregated witness wallet, or 'p2sh-segwit' for legacy compatible wallets
- **encoding** (*str*) – Encoding used for address generation: base58 or bech32. Default is derive from wallet and/or witness type
- **multisig** (*bool*) – Multisig wallet or child of a multisig wallet, default is None / derive from number of keys.
- **sigs\_required** (*int*) – Number of signatures required for validation if using a multisig-nature wallet. For example 2 for 2-of-3 multisignature. Default is all keys must signed
- **cosigner\_id** (*int*) – Set this if wallet contains only public keys, more then one private key or if you would like to create keys for other cosigners. Note: provided keys of a multisig wallet are sorted if sort\_keys = True (default) so if your provided key list is not sorted the cosigned\_id may be different.
- **key\_path** (*list*, *str*) – Key path for multisig wallet, use to create your own non-standard key path. Key path must follow the following rules: \* Path start with masterkey (m) and end with change / address\_index \* If accounts are used, the account level must

be 3. I.e.: m/purpose/coin\_type/account/ \* All keys must be hardened, except for change, address\_index or cosigner\_id \* Max length of path is 8 levels

- **db\_uri** (*str*) – URI of the database

#### Return Wallet

**property** `default_account_id`

**default\_network\_set**(*network*)

**get\_key**(*account\_id=None, network=None, cosigner\_id=None, change=0*)

Get a unused key / address or create a new one with [new\\_key\(\)](#) if there are no unused keys. Returns a key from this wallet which has no transactions linked to it.

Use the `get_keys()` method to a list of unused keys. Calling the `get_key()` method repeatedly to receive a list of key doesn't work: since the key is unused it would return the same result every time you call this method.

```
>>> w = Wallet('create_legacy_wallet_test')
>>> w.get_key()
<WalletKey(key_id=..., name=..., wif=..., path=m/44'/0'/0'/0/...)>
```

#### Parameters

- **account\_id** (*int*) – Account ID. Default is last used or created account ID.
- **network** (*str*) – Network name. Leave empty for default network
- **cosigner\_id** (*int*) – Cosigner ID for key path
- **change** (*int*) – Payment (0) or change key (1). Default is 0

#### Return WalletKey

**get\_key\_change**(*account\_id=None, network=None*)

Get a unused change key or create a new one if there are no unused keys. Wrapper for the [get\\_key\(\)](#) method

#### Parameters

- **account\_id** (*int*) – Account ID. Default is last used or created account ID.
- **network** (*str*) – Network name. Leave empty for default network

#### Return WalletKey

**get\_keys**(*account\_id=None, network=None, cosigner\_id=None, number\_of\_keys=1, change=0*)

Get a list of unused keys / addresses or create a new ones with [new\\_key\(\)](#) if there are no unused keys. Returns a list of keys from this wallet which has no transactions linked to it.

Use the `get_key()` method to get a single key.

#### Parameters

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

#### Return list of WalletKey

**get\_keys\_change**(*account\_id=None, network=None, number\_of\_keys=1*)

Get a unused change key or create a new one if there are no unused keys. Wrapper for the [get\\_key\(\)](#) method

#### Parameters

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

Return list of **WalletKey**

**import\_key**(*key, account\_id=0, name="", network=None, purpose=44, key\_type=None*)

Add new single key to wallet.

#### Parameters

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

Return **WalletKey**

**import\_master\_key**(*hdkey, name='Masterkey (imported)'*)

Import (another) masterkey in this wallet

#### Parameters

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

Return **HDKey** Main key as HDKey object

**info**(*detail=3*)

Prints wallet information to standard output

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

**key**(*term*)

Return single key with given ID or name as **WalletKey** object

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> w.key('change 0').address
'1HabJXe8mTwXiMzUWW5KdpYbFWu3hvtSbF'
```

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

Return **WalletKey** Single key as object

**key\_for\_path**(*path*, *level\_offset=None*, *name=None*, *account\_id=None*, *cosigner\_id=None*, *address\_index=0*, *change=0*, *network=None*, *recreate=False*)

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

```
>>> w = wallet_create_or_open('key_for_path_example')
>>> key = w.key_for_path([0, 0])
>>> key.path
"m/44'/0'/0'/0/0"
```

```
>>> w.key_for_path([], level_offset=-2).path
"m/44'/0'/0'"
```

```
>>> w.key_for_path([], w.depth_public_master + 1).path
"m/44'/0'/0'"
```

Arguments provided in ‘path’ take precedence over other arguments. The *address\_index* argument is ignored: `>>> key = w.key_for_path([0, 10], address_index=1000) >>> key.path "m/44'/0'/0'/0/10" >>> key.address_index 10`

#### Parameters

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

#### Return WalletKey

**keys**(*account\_id=None*, *name=None*, *key\_id=None*, *change=None*, *depth=None*, *used=None*, *is\_private=None*, *has\_balance=None*, *is\_active=None*, *network=None*, *include\_private=False*, *as\_dict=False*)

Search for keys in database. Include 0 or more of *account\_id*, *name*, *key\_id*, *change* and *depth*.

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> all_wallet_keys = w.keys()
>>> w.keys(depth=0)
[<DbKey(id=..., name='bitcoinlib_legacy_wallet_test', wif=
  ↳ 'xprv9s21ZrQH143K3cxbMVswDTYgAc9CeXABQjCD9zmXCpXw4MxN93LanEARbBmV3utHZS9Db4FX1C1RbC5KSNAjQ5W
  ↳ '>]
```

Returns a list of DbKey object or dictionary object if *as\_dict* is True

#### Parameters

- **account\_id** (*int*) – Search for account ID

- **name** (*str*) – Search for Name
- **key\_id** (*int*) – Search for Key ID
- **change** (*int*) – Search for Change
- **depth** (*int*) – Only include keys with this depth
- **used** (*bool*) – Only return used or unused keys
- **is\_private** (*bool*) – Only return private keys
- **has\_balance** (*bool*) – Only include keys with a balance or without a balance, default is both
- **is\_active** (*bool*) – Hide inactive keys. Only include active keys with either a balance or which are unused, default is None (show all)
- **network** (*str*) – Network name filter
- **include\_private** (*bool*) – Include private key information in dictionary
- **as\_dict** (*bool*) – Return keys as dictionary objects. Default is False: DbKey objects

**Return list of DbKey** List of Keys

**keys\_accounts**(*account\_id=None, network='bitcoin', as\_dict=False*)

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

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> account_key = w.keys_accounts()
>>> account_key[0].path
"/m/44'/0'/0/'"
```

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

**Parameters**

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

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

**keys\_address\_change**(*account\_id=None, used=None, network=None, as\_dict=False*)

Get payment addresses (change=1) of specified account\_id for current wallet. Wrapper for the [keys\(\)](#) methods.

**Parameters**

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

:return list of (DbKey, dict)

**keys\_address\_payment**(*account\_id=None, used=None, network=None, as\_dict=False*)

Get payment addresses (change=0) of specified account\_id for current wallet. Wrapper for the [keys\(\)](#) methods.

**Parameters**

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

:return list of (DbKey, dict)

**keys\_addresses**(*account\_id=None, used=None, is\_active=None, change=None, network=None, depth=None, as\_dict=False*)

Get address keys of specified account\_id for current wallet. Wrapper for the [keys\(\)](#) methods.

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> w.keys_addresses()[0].address
'16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg'
```

**Parameters**

- **account\_id** (*int*) – Account ID
- **used** (*bool*) – Only return used or unused keys
- **is\_active** (*bool*) – Hide inactive keys. Only include active keys with either a balance or which are unused, default is True
- **change** (*int*) – Search for Change
- **network** (*str*) – Network name filter
- **depth** (*int*) – Filter by key depth. Default for BIP44 and multisig is 5
- **as\_dict** (*bool*) – Return as dictionary or DbKey object. Default is False: DbKey objects

:return list of (DbKey, dict)

**keys\_networks**(*used=None, as\_dict=False*)

Get keys of defined networks for this wallet. Wrapper for the [keys\(\)](#) method

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> network_key = w.keys_networks()
>>> # Address index of hardened key 0' is 2147483648
>>> network_key[0].address_index
2147483648
>>> network_key[0].path
"m/44'/0'"
```

**Parameters**

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

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

**property name**

Get wallet name

**Return str**



**network\_list**(*field='name'*)

Wrapper for [networks\(\)](#) method, returns a flat list with currently used networks for this wallet.

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> w.network_list()
['bitcoin']
```

**Return list of str**

**networks**(*as\_dict=False*)

Get list of networks used by this wallet

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

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

**new\_account**(*name="", account\_id=None, network=None*)

Create a new account with a child key for payments and 1 for change.

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

**Parameters**

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

**Return WalletKey**

**new\_key**(*name="", account\_id=None, change=0, cosigner\_id=None, network=None*)

Create a new HD Key derived from this wallet’s masterkey. An account will be created for this wallet with index 0 if there is no account defined yet.

```
>>> w = Wallet('create_legacy_wallet_test')
>>> w.new_key('my key')
<WalletKey(key_id=..., name=my key, wif=..., path=m/44'/0'/0'/0/...)>
```

**Parameters**

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

**Return WalletKey**

**new\_key\_change**(*name="", account\_id=None, network=None*)

Create new key to receive change for a transaction. Calls [new\\_key\(\)](#) method with *change=1*.

**Parameters**

- **name** (*str*) – Key name. Default name is ‘Change #’ with an address index

- **account\_id** (*int*) – Account ID. Default is last used or created account ID.
- **network** (*str*) – Network name. Leave empty for default network

**Return** WalletKey

**property owner**

Get wallet Owner

**Return** str

**path\_expand**(*path*, *level\_offset=None*, *account\_id=None*, *cosigner\_id=0*, *address\_index=None*, *change=0*, *network='bitcoin'*)

Create key path. Specify part of key path to expand to key path used in this wallet.

```
>>> w = Wallet('create_legacy_wallet_test')
>>> w.path_expand([0,1200])
['m', '44', '0', '0', '0', '1200']
```

```
>>> w = Wallet('create_legacy_multisig_wallet_test')
>>> w.path_expand([0,2], cosigner_id=1)
['m', '45', '1', '0', '2']
```

**Parameters**

- **path** (*list*, *str*) – Part of path, for example [0, 2] for change=0 and address\_index=2
- **level\_offset** (*int*) – Just create part of path. For example -2 means create path with the last 2 items (change, address\_index) or 1 will return the master key ‘m’
- **account\_id** (*int*) – Account ID
- **cosigner\_id** (*int*) – ID of cosigner
- **address\_index** (*int*) – Index of key, normally provided to ‘path’ argument
- **change** (*int*) – Change key = 1 or normal = 0, normally provided to ‘path’ argument
- **network** (*str*) – Network name. Leave empty for default network

**Return** list

**public\_master**(*account\_id=None*, *name=None*, *as\_private=False*, *network=None*)

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

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

Returns private key information if available and as\_private is True is specified

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> w.public_master().wif
↳ xpub6D2qEr8Z8WYKKns2xZYyyvvRviPh1NKt1kfHwwfiTxJwj7peReEJt3iXoWsr8tXWTsejDjMfAezM53KVfVksZz
↳ '
```

**Parameters**

- **account\_id** (*int*) – Account ID of key to export
- **name** (*str*) – Optional name for account key

- **as\_private** (*bool*) – Export public or private key, default is False
- **network** (*str*) – Network name. Leave empty for default network

**Return list of WalletKey, WalletKey**

**scan**(*scan\_gap\_limit=5, account\_id=None, change=None, rescan\_used=False, network=None, keys\_ignore=None*)

Generate new addresses/keys and scan for new transactions using the Service providers. Updates all UTXO's and balances.

Keep scanning for new transactions until no new transactions are found for 'scan\_gap\_limit' addresses. Only scan keys from default network and account unless another network or account is specified.

Use the faster [utxos\\_update\(\)](#) method if you are only interested in unspent outputs. Use the [transactions\\_update\(\)](#) method if you would like to manage the key creation yourself or if you want to scan a single key.

#### Parameters

- **scan\_gap\_limit** (*int*) – Amount of new keys and change keys (addresses) created for this wallet. Default is 5, so scanning stops if after 5 addresses no transaction are found.
- **account\_id** (*int*) – Account ID. Default is last used or created account ID.
- **change** (*bool*) – Filter by change addresses. Set to True to include only change addresses, False to only include regular addresses. None (default) to disable filter and include both
- **rescan\_used** (*bool*) – Rescan already used addressed. Default is False, so funds send to old addresses will be ignored by default.
- **network** (*str*) – Network name. Leave empty for default network
- **keys\_ignore** (*list of int*) – Id's of keys to ignore

#### Returns

**scan\_key**(*key*)

Scan for new transactions for specified wallet key and update wallet transactions

**Parameters** **key** ([WalletKey](#), *int*) – The wallet key as object or index

**Return bool** New transactions found?

**select\_inputs**(*amount, variance=None, input\_key\_id=None, account\_id=None, network=None, min\_confirms=1, max\_utxos=None, return\_input\_obj=True, skip\_dust\_amounts=True*)

Select available unspent transaction outputs (UTXO's) which can be used as inputs for a transaction for the specified amount.

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> w.select_inputs(50000000)
[<Input(prev_txid=
  ↳ '748799c9047321cb27a6320a827f1f69d767fe889c14bf11f27549638d566fe4', output_
  ↳ n=0, address='16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg', index_n=0, type='sig_pubkey
  ↳ '>)]
```

#### Parameters

- **amount** (*int*) – Total value of inputs in smallest denominator (satoshi) to select
- **variance** (*int*) – Allowed difference in total input value. Default is dust amount of selected network. Difference will be added to transaction fee.

- **input\_key\_id** (*int*, *list*) – Limit UTXO’s search for inputs to this key ID or list of key IDs. Only valid if no input array is specified
- **account\_id** (*int*) – Account ID
- **network** (*str*) – Network name. Leave empty for default network
- **min\_confirms** (*int*) – Minimal confirmation needed for an UTXO before it will be included in inputs. Default is 1 confirmation. Option is ignored if input\_arr is provided.
- **max\_utxos** (*int*) – Maximum number of UTXO’s to use. Set to 1 for optimal privacy. Default is None: No maximum
- **return\_input\_obj** (*bool*) – Return inputs as Input class object. Default is True
- **skip\_dust\_amounts** (*bool*) – Do not include small amount to avoid dust inputs

**Returns** List of previous outputs

**Return type** list of DbTransactionOutput, list of Input

**send**(*output\_arr*, *input\_arr=None*, *input\_key\_id=None*, *account\_id=None*, *network=None*, *fee=None*, *min\_confirms=1*, *priv\_keys=None*, *max\_utxos=None*, *locktime=0*, *offline=False*, *number\_of\_change\_outputs=1*)

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

Uses the [transaction\\_create\(\)](#) method to create a new transaction, and uses a random service client to send the transaction.

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> t = w.send([('1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb', 2000000)], offline=True)
>>> t
<WalletTransaction(input_count=1, output_count=2, status=new, network=bitcoin)>
>>> t.outputs
[<Output(value=..., address=..., type=p2pkh)>, <Output(value=..., address=...,
↳ type=p2pkh)>]
```

### Parameters

- **output\_arr** (*list*) – List of output tuples with address and amount. Must contain at least one item. Example: [(‘mxdLD8SAGS9fe2EeCXALDHcdTTbppMHP8N’, 5000000)]. Address can be an address string, Address object, HDKey object or WalletKey object
- **input\_arr** (*list*) – List of inputs tuples with reference to a UTXO, a wallet key and value. The format is [(txid, output\_n, key\_id, value)]
- **input\_key\_id** (*int*, *list*) – Limit UTXO’s search for inputs to this key ID or list of key IDs. Only valid if no input array is specified
- **account\_id** (*int*) – Account ID
- **network** (*str*) – Network name. Leave empty for default network
- **fee** (*int*) – Set fee manually, leave empty to calculate fees automatically. Set fees in smallest currency denominator, for example satoshi’s if you are using bitcoins
- **min\_confirms** (*int*) – Minimal confirmation needed for an UTXO before it will be included in inputs. Default is 1. Option is ignored if input\_arr is provided.
- **priv\_keys** ([HDKey](#), *list*) – Specify extra private key if not available in this wallet

- **max\_utxos** (*int*) – Maximum number of UTXO's to use. Set to 1 for optimal privacy. Default is None: No maximum
- **locktime** (*int*) – Transaction level locktime. Locks the transaction until a specified block (value from 1 to 5 million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transactions without locktime
- **offline** (*bool*) – Just return the transaction object and do not send it when offline = True. Default is False
- **number\_of\_change\_outputs** (*int*) – Number of change outputs to create when there is a change value. Default is 1. Use 0 for random number of outputs: between 1 and 5 depending on send and change amount

#### Return WalletTransaction

**send\_to**(*to\_address, amount, input\_key\_id=None, account\_id=None, network=None, fee=None, min\_confirms=1, priv\_keys=None, locktime=0, offline=False, number\_of\_change\_outputs=1*)  
Create transaction and send it with default Service objects `services.sendrawtransaction()` method.  
Wrapper for wallet `send()` method.

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> t = w.send_to('1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb', 200000, offline=True)
>>> t
<WalletTransaction(input_count=1, output_count=2, status=new, network=bitcoin)>
>>> t.outputs
[<Output(value=..., address=..., type=p2pkh)>, <Output(value=..., address=...,
↳ type=p2pkh)>]
```

#### Parameters

- **to\_address** (*str, Address, HDKey, WalletKey*) – Single output address as string Address object, HDKey object or WalletKey object
- **amount** (*int, str, Value*) – Output is smallest denominator for this network (ie: Satoshi's for Bitcoin), as Value object or value string as accepted by Value class
- **input\_key\_id** (*int, list*) – Limit UTXO's search for inputs to this key ID or list of key IDs. Only valid if no input array is specified
- **account\_id** (*int*) – Account ID, default is last used
- **network** (*str*) – Network name. Leave empty for default network
- **fee** (*int*) – Fee to use for this transaction. Leave empty to automatically estimate.
- **min\_confirms** (*int*) – Minimal confirmation needed for an UTXO before it will included in inputs. Default is 1. Option is ignored if input\_arr is provided.
- **priv\_keys** (*HDKey, list*) – Specify extra private key if not available in this wallet
- **locktime** (*int*) – Transaction level locktime. Locks the transaction until a specified block (value from 1 to 5 million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transactions without locktime
- **offline** (*bool*) – Just return the transaction object and do not send it when offline = True. Default is False
- **number\_of\_change\_outputs** (*int*) – Number of change outputs to create when there is a change value. Default is 1. Use 0 for random number of outputs: between 1 and 5 depending on send and change amount

### Return WalletTransaction

**sweep**(*to\_address*, *account\_id=None*, *input\_key\_id=None*, *network=None*, *max\_utxos=999*, *min\_confirms=1*, *fee\_per\_kb=None*, *fee=None*, *locktime=0*, *offline=False*)

Sweep all unspent transaction outputs (UTXO's) and send them to one or more output addresses.

Wrapper for the [send\(\)](#) method.

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> t = w.sweep('1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb', offline=True)
>>> t
<WalletTransaction(input_count=1, output_count=1, status=new, network=bitcoin)>
>>> t.outputs
[<Output(value=..., address=1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb, type=p2pkh)>]
```

Output to multiple addresses

```
>>> to_list = [('1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb', 100000), (w.get_key(), 0)]
>>> w.sweep(to_list, offline=True)
<WalletTransaction(input_count=1, output_count=2, status=new, network=bitcoin)>
```

### Parameters

- **to\_address** (*str*, *list*) – Single output address or list of outputs in format [(*<address>*, *<amount>*)]. If you specify a list of outputs, use amount value = 0 to indicate a change output
- **account\_id** (*int*) – Wallet's account ID
- **input\_key\_id** (*int*, *list*) – Limit sweep to UTXO's with this key ID or list of key IDs
- **network** (*str*) – Network name. Leave empty for default network
- **max\_utxos** (*int*) – Limit maximum number of outputs to use. Default is 999
- **min\_confirms** (*int*) – Minimal confirmations needed to include utxo
- **fee\_per\_kb** (*int*) – Fee per kilobyte transaction size, leave empty to get estimated fee costs from Service provider. This option is ignored when the 'fee' option is specified
- **fee** (*int*) – Total transaction fee in smallest denominator (i.e. satoshis). Leave empty to get estimated fee from service providers.
- **locktime** (*int*) – Transaction level locktime. Locks the transaction until a specified block (value from 1 to 5 million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transactions without locktime
- **offline** (*bool*) – Just return the transaction object and do not send it when offline = True. Default is False

### Return WalletTransaction

**transaction**(*txid*)

Get WalletTransaction object for given transaction ID (transaction hash)

**Parameters** **txid** (*str*) – Hexadecimal transaction hash

### Return WalletTransaction

```
transaction_create(output_arr, input_arr=None, input_key_id=None, account_id=None, network=None,
                    fee=None, min_confirms=1, max_utxos=None, locktime=0,
                    number_of_change_outputs=1, random_output_order=True)
```

Create new transaction with specified outputs.

Inputs can be specified but if not provided they will be selected from wallets utxo's with `select_inputs()` method.

Output array is a list of 1 or more addresses and amounts.

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> t = w.transaction_create([('1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTh', 2000000)])
>>> t
<WalletTransaction(input_count=1, output_count=2, status=new, network=bitcoin)>
>>> t.outputs
[<Output(value=..., address=..., type=p2pkh)>, <Output(value=..., address=...,
↳ type=p2pkh)>]
```

### Parameters

- **output\_arr** (*list of Output, tuple*) – List of output as Output objects or tuples with address and amount. Must contain at least one item. Example: `[('mxdLD8SAGS9fe2EeCXALDHcdTTbppMHP8N', 5000000)]`
- **input\_arr** (*list of Input, tuple*) – List of inputs as Input objects or tuples with reference to a UTXO, a wallet key and value. The format is `[(txid, output_n, key_ids, value, signatures, unlocking_script, address)]`
- **input\_key\_id** (*int*) – Limit UTXO's search for inputs to this key\_id. Only valid if no input array is specified
- **account\_id** (*int*) – Account ID
- **network** (*str*) – Network name. Leave empty for default network
- **fee** (*int*) – Set fee manually, leave empty to calculate fees automatically. Set fees in smallest currency denominator, for example satoshi's if you are using bitcoins
- **min\_confirms** (*int*) – Minimal confirmation needed for an UTXO before it will included in inputs. Default is 1 confirmation. Option is ignored if input\_arr is provided.
- **max\_utxos** (*int*) – Maximum number of UTXO's to use. Set to 1 for optimal privacy. Default is None: No maximum
- **locktime** (*int*) – Transaction level locktime. Locks the transaction until a specified block (value from 1 to 5 million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transactions without locktime
- **number\_of\_change\_outputs** (*int*) – Number of change outputs to create when there is a change value. Default is 1. Use 0 for random number of outputs: between 1 and 5 depending on send and change amount :type number\_of\_change\_outputs: int
- **random\_output\_order** (*bool*) – Shuffle order of transaction outputs to increase privacy. Default is True

Return **WalletTransaction** object

```
transaction_import(t)
```

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

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

**Return** [WalletTransaction](#)

**transaction\_import\_raw**(*rawtx*, *network=None*)

Import a raw transaction. Link inputs to wallet keys if possible and return [WalletTransaction](#) object

**Parameters**

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

**Return** [WalletTransaction](#)

**transaction\_last**(*address*)

Get transaction ID for latest transaction in database for given address

**Parameters** **address** (*str*) – The address

**Return** *str*

**transaction\_load**(*txid=None*, *filename=None*)

Load transaction object from file which has been stored with the [Transaction.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](#)

**transaction\_spent**(*txid*, *output\_n*)

Check if transaction with given transaction ID and *output\_n* is spent and return *txid* of spent transaction.

Retrieves information from database, does not update transaction and does not check if transaction is spent with service providers.

**Parameters**

- **txid** (*str*, *bytes*) – Hexadecimal transaction hash
- **output\_n** (*int*, *bytes*) – Output *n*

**Return** *str* Transaction ID

**transactions**(*account\_id=None*, *network=None*, *include\_new=False*, *key\_id=None*, *as\_dict=False*)

Get all known transactions input and outputs for this wallet.

The transaction only includes the inputs and outputs related to this wallet. To get full transactions use the [transactions\\_full\(\)](#) method.

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> w.transactions()
[<WalletTransaction(input_count=0, output_count=1, status=confirmed,
↵network=bitcoin)>]
```

**Parameters**

- **account\_id** (*int*, *None*) – Filter by Account ID. Leave empty for default *account\_id*
- **network** (*str*, *None*) – Filter by network name. Leave empty for default network



- **include\_new** (*bool*) – Also include new and incomplete transactions in list. Default is False
- **key\_id** (*int*, *None*) – Filter by key ID
- **as\_dict** (*bool*) – Output as dictionary or WalletTransaction object

**Return list of WalletTransaction** List of WalletTransaction or transactions as dictionary

**transactions\_export**(*account\_id=None, network=None, include\_new=False, key\_id=None, skip\_change=True*)

**Export wallets transactions as list of tuples with the following fields:** (transaction\_date, transaction\_hash, in/out, addresses\_in, addresses\_out, value, value\_cumulative, fee)

#### Parameters

- **account\_id** (*int*, *None*) – Filter by Account ID. Leave empty for default account\_id
- **network** (*str*, *None*) – Filter by network name. Leave empty for default network
- **include\_new** (*bool*) – Also include new and incomplete transactions in list. Default is False
- **key\_id** (*int*, *None*) – Filter by key ID
- **skip\_change** (*bool*) – Do not include change outputs. Default is True

**Return list of tuple**

**transactions\_full**(*network=None, include\_new=False*)

Get all transactions of this wallet as WalletTransaction objects

Use the [transactions\(\)](#) method to only get the inputs and outputs transaction parts related to this wallet

#### Parameters

- **network** (*str*) – Filter by network name. Leave empty for default network
- **include\_new** (*bool*) – Also include new and incomplete transactions in list. Default is False

**Return list of WalletTransaction**

**transactions\_update**(*account\_id=None, used=None, network=None, key\_id=None, depth=None, change=None, limit=20*)

Update wallets transaction from service providers. Get all transactions for known keys in this wallet. The balances and unspent outputs (UTXO's) are updated as well. Only scan keys from default network and account unless another network or account is specified.

Use the [scan\(\)](#) method for automatic address generation/management, and use the [utxos\\_update\(\)](#) method to only look for unspent outputs and balances.

#### Parameters

- **account\_id** (*int*) – Account ID
- **used** (*bool*, *None*) – Only update used or unused keys, specify None to update both. Default is None
- **network** (*str*) – Network name. Leave empty for default network
- **key\_id** (*int*) – Key ID to just update 1 key

- **depth** (*int*) – Only update keys with this depth, default is depth 5 according to BIP0048 standard. Set depth to None to update all keys of this wallet.
- **change** (*int*) – Only update change or normal keys, default is both (None)
- **limit** (*int*) – Stop update after limit transactions to avoid timeouts with service providers. Default is MAX\_TRANSACTIONS defined in config.py

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

**transactions\_update\_by\_txids**(*txids*)

Update transaction or list of transaction for this wallet with provided transaction ID

**Parameters** **txids** (*str*, *list of str*, *bytes*, *list of bytes*) – Transaction ID, or list of transaction IDs

**Returns**

**transactions\_update\_confirmations**()

Update number of confirmations and status for transactions in database

**Returns**

**utxo\_add**(*address*, *value*, *txid*, *output\_n*, *confirmations=1*, *script=""*)

Add a single UTXO to the wallet database. To update all utxo's use [utxos\\_update\(\)](#) method.

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

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

**Parameters**

- **address** (*str*) – Address of Unspent Output. Address should be available in wallet
- **value** (*int*) – Value of output in satoshis or smallest denominator for type of currency
- **txid** (*str*) – Transaction hash or previous output as hex-string
- **output\_n** (*int*) – Output number of previous transaction output
- **confirmations** (*int*) – Number of confirmations. Default is 0, unconfirmed
- **script** (*str*) – Locking script of previous output as hex-string

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

**utxo\_last**(*address*)

Get transaction ID for latest utxo in database for given address

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> w.utxo_last('16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg')
'748799c9047321cb27a6320a827f1f69d767fe889c14bf11f27549638d566fe4'
```

**Parameters** **address** (*str*) – The address

**Return str**

**utxos**(*account\_id=None*, *network=None*, *min\_confirms=0*, *key\_id=None*)

Get UTXO's (Unspent Outputs) from database. Use [utxos\\_update\(\)](#) method first for updated values

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> w.utxos()
[{'value': 1000000000, 'script': '', 'output_n': 0, 'transaction_id': ..., 'spent':
  False, 'script_type': 'p2pkh', 'key_id': ..., 'address':
  '16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg', 'confirmations': 0, 'txid': (continues on next page)
  '748799c9047321cb27a6320a827f1f69d767fe889c14bf11f27549638d566fe4', 'network_
  name': 'bitcoin'}]
```

(continued from previous page)

**Parameters**

- **account\_id** (*int*) – Account ID
- **network** (*str*) – Network name. Leave empty for default network
- **min\_confirms** (*int*) – Minimal confirmation needed to include in output list
- **key\_id** (*int*, *list*) – Key ID or list of key IDs to filter utxo's for specific keys

**Return list** List of transactions

**utxos\_update**(*account\_id=None, used=None, networks=None, key\_id=None, depth=None, change=None, utxos=None, update\_balance=True, max\_utxos=20, rescan\_all=True*)

Update UTXO's (Unspent Outputs) for addresses/keys in this wallet using various Service providers.

This method does not import transactions: use [transactions\\_update\(\)](#) function or to look for new addresses use [scan\(\)](#).

**Parameters**

- **account\_id** (*int*) – Account ID
- **used** (*bool*) – Only check for UTXO for used or unused keys. Default is both
- **networks** (*str*, *list*) – Network name filter as string or list of strings. Leave empty to update all used networks in wallet
- **key\_id** (*int*) – Key ID to just update 1 key
- **depth** (*int*) – Only update keys with this depth, default is depth 5 according to BIP0048 standard. Set depth to None to update all keys of this wallet.
- **change** (*int*) – Only update change or normal keys, default is both (None)
- **utxos** (*list of dict.*) – List of unspent outputs in dictionary format specified below. For usage on an offline PC, you can import utxos with the utxos parameter as a list of dictionaries

```
{
  "address": "n2S9Czehjvdmppwd2YqekxuUC1Tz5ZdK3YN",
  "script": "",
  "confirmations": 10,
  "output_n": 1,
  "txid": "9df91f89a3eb4259ce04af66ad4caf3c9a297feca5e0b3bc506898b6728c5003",
  "value": 8970937
}
```

**Parameters**

- **update\_balance** (*bool*) – Option to disable balance update after fetching UTXO's. Can be used when utxos\_update method is called several times in a row. Default is True
- **max\_utxos** (*int*) – Maximum number of UTXO's to update
- **rescan\_all** (*bool*) – Remove old utxo's and rescan wallet. Default is True. Set to False if you work with large utxo's sets. Value will be ignored if key\_id is specified in your call

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

**wif**(*is\_private=False, account\_id=0*)

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

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

**Parameters**

- **is\_private** (*bool*) – Export public or private key, default is False
- **account\_id** (*bool*) – Account ID of key to export

**Return list, str**

**exception** bitcoinlib.wallets.**WalletError**(*msg=""*)

Bases: Exception

Handle Wallet class Exceptions

**class** bitcoinlib.wallets.**WalletKey**(*key\_id, session, hdkey\_object=None*)

Bases: object

Used as attribute of Wallet class. Contains HDKey class, and adds extra wallet related information such as key ID, name, path and balance.

All WalletKeys are stored in a database

Initialize WalletKey with specified ID, get information from database.

**Parameters**

- **key\_id** (*int*) – ID of key as mentioned in database
- **session** (*sqlalchemy.orm.session.Session*) – Required SQLAlchemy Session object
- **hdkey\_object** (*HDKey*) – Optional HDKey object. Specify HDKey object if available for performance

**as\_dict**(*include\_private=False*)

Return current key information as dictionary

**Parameters** **include\_private** (*bool*) – Include private key information in dictionary

**balance**(*as\_string=False*)

Get total value of unspent outputs

**Parameters** **as\_string** (*bool*) – Specify ‘string’ to return a string in currency format

**Return float, str** Key balance

**static from\_key**(*name, wallet\_id, session, key, account\_id=0, network=None, change=0, purpose=44, parent\_id=0, path='m', key\_type=None, encoding=None, witness\_type='legacy', multisig=False, cosigner\_id=None*)

Create WalletKey from a HDKey object or key.

Normally you don’t need to call this method directly. Key creation is handled by the Wallet class.

```
>>> w = wallet_create_or_open('hdwalletkey_test')
>>> wif =
↳ 'xprv9s21ZrQH143K2mcs9jcK4EjALbu2z1N9qsMTUG1frmnXM3NNCSGR57yLhwTccfNCwdSQEDftgjCCGm96P29wGGCbl'
↳
>>> wk = WalletKey.from_key('import_key', w.wallet_id, w._session, wif)
>>> wk.address
'1MwVEhGq6gg1eeSrEdZom5bHyPqXtJSnPg'
```

(continues on next page)

(continued from previous page)

```
>>> wk
<WalletKey(key_id=..., name=import_key,
↳ wif=xprv9s21ZrQH143K2mcs9jcK4EjALbu2z1N9qsMTUG1frmnXM3NNCSGR57yLhwTccfNCwdSQEDftgjCGm96P29wG
↳ path=m)>
```

**Parameters**

- **name** (*str*) – New key name
- **wallet\_id** (*int*) – ID of wallet where to store key
- **session** (*sqlalchemy.orm.session.Session*) – Required Sqlalchemy Session object
- **key** (*str, int, byte, HDKey*) – Optional key in any format accepted by the HDKey class
- **account\_id** (*int*) – Account ID for specified key, default is 0
- **network** (*str*) – Network of specified key
- **change** (*int*) – Use 0 for normal key, and 1 for change key (for returned payments)
- **purpose** (*int*) – BIP0044 purpose field, default is 44
- **parent\_id** (*int*) – Key ID of parent, default is 0 (no parent)
- **path** (*str*) – BIP0044 path of given key, default is 'm' (masterkey)
- **key\_type** (*str*) – Type of key, single or BIP44 type
- **encoding** (*str*) – Encoding used for address, i.e.: base58 or bech32. Default is base58
- **witness\_type** (*str*) – Witness type used when creating transaction script: legacy, p2sh-segwit or segwit.
- **multisig** (*bool*) – Specify if key is part of multisig wallet, used for create keys and key representations such as WIF and addresses
- **cosigner\_id** (*int*) – Set this if you would like to create keys for other cosigners.

**Return WalletKey** WalletKey object**key()**

Get HDKey object for current WalletKey

**Return HDKey****property name**

Return name of wallet key

**Return str****public()**

Return current key as public WalletKey object with all private information removed

**Return WalletKey****class** bitcoinlib.wallets.WalletTransaction(*hdwallet, account\_id=None, \*args, \*\*kwargs*)Bases: *bitcoinlib.transactions.Transaction*

Used as attribute of Wallet class. Child of Transaction object with extra reference to wallet and database object.

All WalletTransaction items are stored in a database

Initialize WalletTransaction object with reference to a Wallet object

**Parameters**

- **hdwallet** – Wallet object, wallet name or ID
- **account\_id** (*int*) – Account ID
- **args** (*args*) – Arguments for HDWallet parent class
- **kwargs** (*kwargs*) – Keyword arguments for Wallet parent class

**delete()**

Delete this transaction from database.

WARNING: Results in incomplete wallets, transactions will NOT be automatically downloaded again when scanning or updating wallet. In normal situations only use to remove old unconfirmed transactions

**Return int** Number of deleted transactions

**export**(*skip\_change=True*)

**Export this transaction as list of tuples in the following format:** (transaction\_date, transaction\_hash, in/out, addresses\_in, addresses\_out, value, fee)

A transaction with multiple inputs or outputs results in multiple tuples.

**Parameters** **skip\_change** (*boolean*) – Do not include outputs to own wallet (default). Please note: So if this is set to True, then an internal transfer is not exported.

**Return list of tuple**

**classmethod from\_transaction**(*hdwallet, t*)

Create WalletTransaction object from Transaction object

**Parameters**

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

**Return WalletClass**

**classmethod from\_txid**(*hdwallet, txid*)

Read single transaction from database with given transaction ID / transaction hash

**Parameters**

- **hdwallet** (*Wallet*) – Wallet object
- **txid** (*str, bytes*) – Transaction hash as hexadecimal string

**Return WalletClass**

**info()**

Print Wallet transaction information to standard output. Include send information.

**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**

**send**(*offline=False*)

Verify and push transaction to network. Update UTXO's in database after successful send

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

**Return** None

**sign**(*keys=None, index\_n=0, multisig\_key\_n=None, hash\_type=1, fail\_on\_unknown\_key=False, replace\_signatures=False*)

Sign this transaction. Use existing keys from wallet or use keys argument for extra keys.

**Parameters**

- **keys** (*HDKey, str*) – Extra private keys to sign the transaction
- **index\_n** (*int*) – Transaction index\_n to sign
- **multisig\_key\_n** (*int*) – Index number of key for multisig input for segwit transactions. Leave empty if not known. If not specified all possibilities will be checked
- **hash\_type** (*int*) – Hashtype to use, default is SIGHASH\_ALL
- **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

**store()**

Store this transaction to database

**Return** *int* Transaction index number

**bitcoinlib.wallets.normalize\_path**(*path*)

Normalize BIP0044 key path for HD keys. Using single quotes for hardened keys

```
>>> normalize_path("m/44h/2p/1'/0/100")
"m/44'/2'/1'/0/100"
```

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

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

**bitcoinlib.wallets.wallet\_create\_or\_open**(*name, keys="", owner="", network=None, account\_id=0, purpose=None, scheme='bip32', sort\_keys=True, password="", witness\_type=None, encoding=None, multisig=None, sigs\_required=None, cosigner\_id=None, key\_path=None, db\_uri=None*)

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

Returns Wallet object

See Wallets class create method for option documentation

**bitcoinlib.wallets.wallet\_delete**(*wallet, db\_uri=None, force=False*)

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

**Parameters**

- **wallet** (*int, str*) – Wallet ID as integer or Wallet Name as string
- **db\_uri** (*str*) – URI of the database

- **force** (*bool*) – If set to True wallet will be deleted even if unspent outputs are found. Default is False

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

`bitcoinlib.wallets.wallet_delete_if_exists(wallet, db_uri=None, force=False)`

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

**Parameters**

- **wallet** (*int*, *str*) – Wallet ID as integer or Wallet Name as string
- **db\_uri** (*str*) – URI of the database
- **force** (*bool*) – If set to True wallet will be deleted even if unspent outputs are found. Default is False

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

`bitcoinlib.wallets.wallet_empty(wallet, db_uri=None)`

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

**Parameters**

- **wallet** (*int*, *str*) – Wallet ID as integer or Wallet Name as string
- **db\_uri** (*str*) – URI of the database

**Return bool** True if successful

`bitcoinlib.wallets.wallet_exists(wallet, db_uri=None)`

Check if Wallets is defined in database

**Parameters**

- **wallet** (*int*, *str*) – Wallet ID as integer or Wallet Name as string
- **db\_uri** (*str*) – URI of the database

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

`bitcoinlib.wallets.wallets_list(db_uri=None, include_cosigners=False)`

List Wallets from database

**Parameters**

- **db\_uri** (*str*) – URI of the database
- **include\_cosigners** (*bool*) – Child wallets for multisig wallets are for internal use only and are skipped by default

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



## 8.13 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 separated 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 separated 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()

↳'6969ed4666db67fc74fae7869e2acf3c766b5ef95f5e31eb2fceb93d76069c6de971225f700042b0b513f0ad6c
↳'
```

#### 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

## 8.14 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\xaa\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 retrieve info with full WIF string:

```
>>> wif_prefix_search(
  ↳ 'xprv9wTYmMFdV23N21MM6dLNavSQV7Sj7meSPXx6AV5eTdqqGLjycVjb115Ec5LgRAXscPZgy5G4jQ9csyyZLN3PZLxoM1h3
  ↳ ', 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**

## 8.15 bitcoinlib.blocks module

**class** `bitcoinlib.blocks.Block(block_hash, version, prev_block, merkle_root, time, bits, nonce, transactions=None, height=None, confirmations=None, network='bitcoin')`

Bases: `object`

Create a new Block object with provided parameters.

```
>>> b = Block('000000000000000000000000000000000000154ba9d02ddd6cee0d71d1ea232753e02c9ac6affd709',
↳ version=0x20000000, prev_block=
↳ '000000000000000000000000f9578cda278ae7a2002e50d8e6079d11e2ea1f672b483', merkle_root=
↳ '20e86f03c24c53c12014264d0e405e014e15a02ad02c174f017ee040750f8d9d',
↳ time=1592848036, bits=387044594, nonce=791719079)
>>> b
<Block(000000000000000000000000000000000000154ba9d02ddd6cee0d71d1ea232753e02c9ac6affd709, None,
↳ transactions: 0)>
```

## Parameters

- **block\_hash** (*bytes, str*) – Hash value of serialized block
- **version** (*bytes, str, int*) – Block version to indicate which software / BIPs are used to create block
- **prev\_block** (*bytes, str*) – Hash of previous block in blockchain
- **merkle\_root** (*bytes, str*) – Merkle root. Top item merkle chain tree to validate transactions.
- **time** (*int, bytes*) – Timestamp of time when block was included in blockchain
- **bits** (*bytes, str, int*) – Bits are used to indicate target / difficulty
- **nonce** (*bytes, str, int*) – Number used once, n-once is used to create randomness for miners to find a suitable block hash
- **transactions** (*list of Transaction, list of str*) – List of transaction included in this block. As list of transaction objects or list of transaction IDs strings
- **height** (*int*) – Height of this block in the Blockchain
- **confirmations** (*int*) – Number of confirmations for this block, or depth. Increased when new blocks are found
- **network** (*str, Network*) – Network, leave empty for default network

**as\_dict()**

Get representation of current Block as dictionary.

**Return dict**

### check\_proof\_of\_work()

Check proof of work for this block. Block hash must be below target.

This library is not optimised for mining, but you can use this for testing or learning purposes.

```
>>> b = Block('0000000000000000000000000000000000000000000000000000000000000000' +  
↳ ', version=0x200000000, prev_block=  
↳ '00000000000000000000000000000000f9578cda278ae7a2002e50d8e6079d11e2ea1f672b483', merkle_  
↳ root='20e86f03c24c53c12014264d0e405e014e15a02ad02c174f017ee040750f8d9d',  
↳ time=1592848036, bits=387044594, nonce=791719079)  
>>> b.check_proof_of_work()  
True
```

## Return bool

**property difficulty**

Block difficulty calculated from bits / target. Human readable representation of block's target.

Genesis block has difficulty of 1.0

```
>>> from bitcoinlib.services.services import Service
>>> srv = Service()
>>> b = srv.getblock(0)
>>> b.difficulty
1.0
```

**Return float**

**classmethod from\_raw**(raw, block\_hash=None, height=None, parse\_transactions=False, limit=0, network='bitcoin')

Create Block object from raw serialized block in bytes.

Get genesis block:

```
>>> from bitcoinlib.services.services import Service
>>> srv = Service()
>>> b = srv.getblock(0)
>>> b.block_hash.hex()
'000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f'
```

**Parameters**

- **raw** (bytes) – Raw serialize block
- **block\_hash** (bytes) – Specify block hash if known to verify raw block. Value error will be raised if calculated block hash is different than specified.
- **height** (int) – Specify height if known. Will be derived from coinbase transaction if not provided.
- **parse\_transactions** (bool) – Indicate if transactions in raw block need to be parsed and converted to Transaction objects. Default is False
- **limit** (int) – Maximum number of transactions to parse. Default is 0: parse all transactions. Only used if parse\_transaction is set to True
- **network** (str) – Name of network

**Return Block**

**classmethod parse**(raw, block\_hash=None, height=None, parse\_transactions=False, limit=0, network='bitcoin')

Create Block object from raw serialized block in bytes or BytesIO format. Wrapper for [parse\\_bytesio\(\)](#)

Get genesis block:

```
>>> from bitcoinlib.services.services import Service
>>> srv = Service()
>>> b = srv.getblock(0)
>>> b.block_hash.hex()
'000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f'
```

**Parameters**

- **raw** (*BytesIO*, *bytes*) – Raw serialize block
- **block\_hash** (*bytes*) – Specify block hash if known to verify raw block. Value error will be raised if calculated block hash is different than specified.
- **height** (*int*) – Specify height if known. Will be derived from coinbase transaction if not provided.
- **parse\_transactions** (*bool*) – Indicate if transactions in raw block need to be parsed and converted to Transaction objects. Default is False
- **limit** (*int*) – Maximum number of transactions to parse. Default is 0: parse all transactions. Only used if parse\_transaction is set to True
- **network** (*str*) – Name of network

#### Return Block

**classmethod** `parse_bytes`(*raw\_bytes*, *block\_hash=None*, *height=None*, *parse\_transactions=False*, *limit=0*, *network='bitcoin'*)

Create Block object from raw serialized block in bytes or BytesIO format. Wrapper for `parse_bytesio()`

Get genesis block:

```
>>> from bitcoinlib.services.services import Service
>>> srv = Service()
>>> b = srv.getblock(0)
>>> b.block_hash.hex()
'00000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f'
```

#### Parameters

- **raw\_bytes** (*bytes*) – Raw serialize block
- **block\_hash** (*bytes*) – Specify block hash if known to verify raw block. Value error will be raised if calculated block hash is different than specified.
- **height** (*int*) – Specify height if known. Will be derived from coinbase transaction if not provided.
- **parse\_transactions** (*bool*) – Indicate if transactions in raw block need to be parsed and converted to Transaction objects. Default is False
- **limit** (*int*) – Maximum number of transactions to parse. Default is 0: parse all transactions. Only used if parse\_transaction is set to True
- **network** (*str*) – Name of network

#### Return Block

**classmethod** `parse_bytesio`(*raw*, *block\_hash=None*, *height=None*, *parse\_transactions=False*, *limit=0*, *network='bitcoin'*)

Create Block object from raw serialized block in BytesIO format

Get genesis block:

```
>>> from bitcoinlib.services.services import Service
>>> srv = Service()
>>> b = srv.getblock(0)
>>> b.block_hash.hex()
'00000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f'
```





**version\_bips()**

Extract version signaling information from the block's version number.

The block version shows which software the miner used to create the block. Changes to the bitcoin protocol are described in Bitcoin Improvement Proposals (BIPs) and a miner shows which BIPs it supports in the block version number.

This method returns a list of BIP version number as string.

Example: This block uses the BIP9 versioning system and signals BIP141 (segwit) >>> from bitcoinlib.services.services import Service >>> srv = Service() >>> b = srv.getblock(450001) >>> print(b.version\_bips()) ['BIP9', 'BIP141']

**Return list of str**

## 8.16 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=10.000000000000000, denominator=1.00000000, network='bitcoin')

```
>>> Value('15 mBTC')
Value(value=0.01500000000000000, denominator=0.001000000, network='bitcoin')
```

```
>>> Value('10 sat')
Value(value=0.000000100000000, denominator=0.00000001, network='bitcoin')
```

```
>>> Value('1 doge')
Value(value=1.000000000000000, denominator=1.00000000, network='dogecoin')
```

```
>>> Value(500, 'm')
Value(value=0.500000000000000, denominator=0.001000000, network='bitcoin')
```

```
>>> Value(500, 0.001)
Value(value=0.500000000000000, denominator=0.001000000, network='bitcoin')
```

All frequently used arithmetic, comparison 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=0.0220000000000000, 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 supplied 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')
'5.000000 mBTC'
```

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

```
>>> Value('130000000 Doge').str('auto') # Yeah, mega Dogecoins...
'13.000000000 MDOGE'
```

```
>>> Value('21000000000').str('auto')
'2.100000000 GBTC'
```

```
>>> Value('1.5 BTC').str(currency_repr='symbol')
'1.500000000 '
```

```
>>> Value('1.5 BTC').str(currency_repr='name')
'1.500000000 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:

#### 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

## 8.17 bitcoinlib.services.services module

**class** bitcoinlib.services.services.**Cache**(*network*, *db\_uri=""*)

Bases: object

Store transaction, utxo and address information in database to increase speed and avoid duplicate calls to service providers.

Once confirmed a transaction is immutable so we have to fetch it from a service provider only once. When checking for new transactions or utxo's for a certain address we only have to check the new blocks.

This class is used by the Service class and normally you won't need to access it directly.

Open Cache class

### Parameters

- **network** (*str*, [Network](#)) – Specify network used
- **db\_uri** (*str*) – Database to use for caching

**blockcount**(*never\_expires=False*)

Get number of blocks on the current network from cache if recent data is available.

**Parameters** **never\_expires** (*bool*) – Always return latest blockcount found. Can be used to avoid return to old blocks if service providers are not up-to-date.

**Return** **int**

**cache\_enabled**()

Check if caching is enabled. Returns False if SERVICE\_CACHING\_ENABLED is False or no session is defined.

**Return** **bool**

**commit**()

Commit queries in self.session. Rollback if commit fails.

**Returns**

**estimatefee**(*blocks*)

Get fee estimation from cache for confirmation within specified amount of blocks.

Stored in cache in three groups: low, medium and high fees.

**Parameters** **blocks** (*int*) – Expectation confirmation time in blocks.

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

**getaddress**(*address*)

Get address information from cache, with links to transactions and utxo's and latest update information.

**Parameters** **address** (*str*) – Address string

**Return** **DbCacheAddress** An address cache database object

**getblock**(*blockid*)

Get specific block from database cache.

**Parameters** **blockid** (*int*, *str*) – Block height or block hash

**Return** **Block**

**getblocktransactions**(*height*, *page*, *limit*)

Get range of transactions from a block

**Parameters**

- **height** (*int*) – Block height
- **page** (*int*) – Transaction page
- **limit** (*int*) – Number of transactions per page

**Returns****getrawtransaction**(*txid*)

Get a raw transaction string from the database cache if available

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

**Return str** Raw transaction as hexstring

**gettransaction**(*txid*)

Get transaction from cache. Returns False if not available

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

**Return Transaction** A single Transaction object

**gettransactions**(*address, after\_txid="", limit=20*)

Get transactions from cache. Returns empty list if no transactions are found or caching is disabled.

**Parameters**

- **address** (*str*) – Address string
- **after\_txid** (*bytes*) – Transaction ID of last known transaction. Only check for transactions after given tx id. Default: Leave empty to return all transaction. If used only provide a single address
- **limit** (*int*) – Maximum number of transactions to return

**Return list** List of Transaction objects

**getutxos**(*address, after\_txid=""*)

Get list of unspent outputs (UTXO's) for specified address from database cache.

Sorted from old to new, so highest number of confirmations first.

**Parameters**

- **address** (*str*) – Address string
- **after\_txid** (*bytes*) – Transaction ID of last known transaction. Only check for utxos after given tx id. Default: Leave empty to return all utxos.

**Return dict** UTXO's per address

**store\_address**(*address, last\_block=None, balance=0, n\_utxos=None, txs\_complete=False, last\_txid=None*)

Store address information in cache

**param address** Address string

**type address** str

**param last\_block** Number or last block retrieved from service provider. For instance if address contains a large number of transactions and they will be retrieved in more than one request.

**type last\_block** int

**param balance** Total balance of address in sathosis, or smallest network detominator

**type balance** int

**param n\_utxos** Total number of UTXO's for this address

**type n\_utxos** int

**param txs\_complete** True if all transactions for this address are added to cache

**type txs\_complete** bool

**param last\_txid** Transaction ID of last transaction downloaded from blockchain

**type last\_txid** bytes

. :return:

**store\_block**(*block*)

Store block in cache database

**Parameters** **block** ([Block](#)) – Block

**Returns**

**store\_blockcount**(*blockcount*)

Store network blockcount in cache for 60 seconds

**Parameters** **blockcount** (*int*, *str*) – Number of latest block

**Returns**

**store\_estimated\_fee**(*blocks*, *fee*)

Store estimated fee retrieved from service providers in cache.

**Parameters**

- **blocks** (*int*) – Confirmation within x blocks
- **fee** (*int*) – Estimated fee in Sathosis

**Returns**

**store\_transaction**(*t*, *order\_n=None*, *commit=True*)

Store transaction in cache. Use order number to determine order in a block

**Parameters**

- **t** ([Transaction](#)) – Transaction
- **order\_n** (*int*) – Order in block
- **commit** – Commit transaction to database. Default is True. Can be disabled if a larger number of transactions are added to cache, so you can commit outside this method.

**Returns**

```
class bitcoinlib.services.services.Service(network='bitcoin', min_providers=1, max_providers=1,  
                                           providers=None, timeout=5, cache_uri=None,  
                                           ignore_priority=False, exclude_providers=None,  
                                           max_errors=4)
```

Bases: object

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

The Service class connects to 1 or more service providers at random to retrieve or send information. If a service providers fails to correctly respond the Service class will try another available provider.



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

#### Parameters

- **network** (*str*, *Network*) – Specify network used
- **min\_providers** (*int*) – Minimum number of providers to connect to. Default is 1. Use for instance to receive fee information from a number of providers and calculate the average fee.
- **max\_providers** (*int*) – Maximum number of providers to connect to. Default is 1.
- **providers** (*list of str*) – List of providers to connect to. Default is all providers and select a provider at random.
- **timeout** (*int*) – Timeout for web requests. Leave empty to use default from config settings
- **cache\_uri** (*str*) – Database to use for caching
- **ignore\_priority** (*bool*) – Ignores provider priority if set to True. Could be used for unit testing, so no providers are missed when testing. Default is False
- **exclude\_providers** (*list of str*) – Exclude providers in this list, can be used when problems with certain providers arise.

#### blockcount()

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

Block count is cached for BLOCK\_COUNT\_CACHE\_TIME seconds to avoid to many calls to service providers.

#### Return int

#### estimatefee(blocks=3)

Estimate fee per kilobyte for a transaction for this network with expected confirmation within a certain amount of blocks

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

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

#### getbalance(addresslist, addresses\_per\_request=5)

Get total balance for address or list of addresses

#### Parameters

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

**Return dict** Balance per address

#### getblock(blockid, parse\_transactions=True, page=1, limit=None)

Get block with specified block height or block hash from service providers.

If parse\_transaction is set to True a list of Transaction object will be returned otherwise a list of transaction ID's.

Some providers require 1 or 2 extra request per transaction, so to avoid timeouts or rate limiting errors you can specify a page and limit for the transaction. For instance with page=2, limit=4 only transaction 5 to 8 are returned in the Blocks's 'transaction' attribute.

If you only use a local bcoin or bitcoind provider, make sure you set the limit to maximum (i.e. 9999) because all transactions are already downloaded when fetching the block.

```
>>> from bitcoinlib.services.services import Service
>>> srv = Service()
>>> b = srv.getblock(0)
>>> b
<Block(00000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f, 0,
↳ transactions: 1)>
```

#### Parameters

- **blockid** (*str*, *int*) – Hash or block height of block
- **parse\_transactions** (*bool*) – Return Transaction objects or just transaction ID's. Default is return txids.
- **page** (*int*) – Page number of transaction paging. Default is start from the beginning: 1
- **limit** (*int*) – Maximum amount of transaction to return. Default is 10 if parse transaction is enabled, otherwise returns all txid's (9999)

#### Return Block

##### **getcacheaddressinfo**(*address*)

Get address information from cache. I.e. balance, number of transactions, number of utxo's, etc

Cache will only be filled after all transactions for a specific address are retrieved (with `gettransactions` ie)

**Parameters** **address** (*str*) – address string

#### Return dict

##### **getinfo**()

Returns info about current network. Such as difficulty, latest block, mempool size and network hashrate.

#### Return dict

##### **getinputvalues**(*t*)

Retrieve values for transaction inputs for given Transaction.

Raw transactions as stored on the blockchain do not contain the input values but only the previous transaction hash and index number. This method retrieves the previous transaction and reads the value.

**Parameters** **t** ([Transaction](#)) – Transaction

#### Return Transaction

##### **getrawblock**(*blockid*)

Get raw block as hexadecimal string for block with specified hash or block height.

Not many providers offer this option, and it can be slow, so it is advised to use a local client such as bitcoind.

**Parameters** **blockid** (*str*, *int*) – Block hash or block height

#### Return str

##### **getrawtransaction**(*txid*)

Get a raw transaction by its transaction hash

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

**Return str** Raw transaction as hexstring

##### **gettransaction**(*txid*)

Get a transaction by its transaction hash. Convert to Bitcoinlib transaction object.

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

**Return Transaction** A single transaction object

**gettransactions**(*address*, *after\_txid=""*, *limit=20*)

Get all transactions for specified address.

Sorted from old to new, so transactions with highest number of confirmations first.

**Parameters**

- **address** (*str*) – Address string
- **after\_txid** (*str*) – Transaction ID of last known transaction. Only check for transactions after given tx id. Default: Leave empty to return all transaction. If used only provide a single address
- **limit** (*int*) – Maximum number of transactions to return

**Return list** List of Transaction objects

**getutxos**(*address*, *after\_txid=""*, *limit=20*)

Get list of unspent outputs (UTXO's) for specified address.

Sorted from old to new, so highest number of confirmations first.

**Parameters**

- **address** (*str*) – Address string
- **after\_txid** (*str*) – Transaction ID of last known transaction. Only check for utxos after given tx id. Default: Leave empty to return all utxos.
- **limit** (*int*) – Maximum number of utxo's to return

**Return dict** UTXO's per address

**isspent**(*txid*, *output\_n*)

Check if the output with provided transaction ID and output number is spent.

**Parameters**

- **txid** (*str*) – Transaction ID hex
- **output\_n** (*int*) – Output number

**Return bool**

**mempool**(*txid=""*)

Get list of all transaction IDs in the current mempool

A full list of transactions ID's will only be returned if a bcoin or bitcoind client is available. Otherwise specify the txid option to verify if a transaction is added to the mempool.

**Parameters** **txid** (*str*) – Check if transaction with this hash exists in memory pool

**Return list**

**sendrawtransaction**(*rawtx*)

Push a raw transaction to the network

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

**Return dict** Send transaction result

**exception** `bitcoinlib.services.services.ServiceError(msg="")`

Bases: Exception

## 8.18 bitcoinlib.services package

### 8.18.1 Submodules

#### 8.18.1.1 bitcoinlib.services.authproxy module

Copyright 2011 Jeff Garzik

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

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

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

Copyright (c) 2007 Jan-Klaas Kollhof

This file is part of jsonrpc.

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

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

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

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

Bases: object

**batch\_**(*rpc\_calls*)

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

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

**exception** bitcoinlib.services.authproxy.JSONRPCException(*rpc\_error*)

Bases: Exception

### 8.18.1.2 bitcoinlib.services.baseclient module

```
class bitcoinlib.services.baseclient.BaseClient(network, provider, base_url, denominator, api_key="",  
                                              provider_coin_id="", network_overrides=None,  
                                              timeout=5, latest_block=None)
```

Bases: object

```
request(url_path, variables=None, method='get', secure=True, post_data="")
```

```
exception bitcoinlib.services.baseclient.ClientError(msg="")
```

Bases: Exception

### 8.18.1.3 bitcoinlib.services.bcoin module

```
class bitcoinlib.services.bcoin.BcoinClient(network, base_url, denominator, *args)
```

Bases: [bitcoinlib.services.baseclient.BaseClient](#)

Class to interact with Bcoin API

```
blockcount()
```

```
compose_request(func, data="", parameter="", variables=None, method='get')
```

```
estimatefee(blocks)
```

```
getbalance(addresslist)
```

```
getblock(blockid, parse_transactions, page, limit)
```

```
getinfo()
```

```
getrawtransaction(txid)
```

```
gettransaction(txid)
```

```
gettransactions(address, after_txid="", limit=20)
```

```
getutxos(address, after_txid="", limit=20)
```

```
isspent(txid, index)
```

```
mempool(txid="")
```

```
sendrawtransaction(rawtx)
```

### 8.18.1.4 bitcoinlib.services.bitaps module

```
class bitcoinlib.services.bitaps.BitapsClient(network, base_url, denominator, *args)
```

Bases: [bitcoinlib.services.baseclient.BaseClient](#)

```
blockcount()
```

```
compose_request(category, command="", data="", variables=None, req_type='blockchain', method='get')
```

```
getbalance(addresslist)
```

```
getrawtransaction(txid)
```

```
getutxos(address, after_txid="", limit=20)
```

### 8.18.1.5 bitcoinlib.services.bitcoind module

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

Bases: *bitcoinlib.services.baseclient.BaseClient*

Class to interact with bitcoind, the Bitcoin daemon

Open connection to bitcoin node

#### Parameters

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

**Type** str

**Type** str

**Type** str

**blockcount()**

**estimatefee(blocks)**

**static from\_config(configfile=None, network='bitcoin')**

Read settings from bitcoind config file

#### Parameters

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

**Type** str

**Type** str

#### Return BitcoindClient

**getblock(blockid, parse\_transactions=True, page=1, limit=None)**

**getinfo()**

**getrawblock(blockid)**

**getrawtransaction(txid)**

**gettransaction(txid)**

**isspent(txid, index)**

**mempool(txid="")**

**sendrawtransaction(rawtx)**

**exception bitcoinlib.services.bitcoind.ConfigError(msg="")**

Bases: Exception

### 8.18.1.6 bitcoinlib.services.bitcoinlibtest module

**class** bitcoinlib.services.bitcoinlibtest.BitcoinLibTestClient(*network, base\_url, denominator, \*args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

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

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

**blockcount()**

**estimatefee(blocks)**

Dummy estimate fee method for the bitcoinlib testnet.

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

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

**getbalance(addresslist)**

Dummy getbalance method for bitcoinlib testnet

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

**Return int**

**getutxos(address, after\_txid="", limit=10, utxos\_per\_address=2)**

Dummy method to retrieve UTXO's. This method creates a new UTXO for each address provided out of the testnet void, which can be used to create test transactions for the bitcoinlib testnet.

**Parameters**

- **address** (*str*) – Address string
- **after\_txid** (*str*) – Transaction ID of last known transaction. Only check for utxos after given tx id. Default: Leave empty to return all utxos. If used only provide a single address
- **limit** (*int*) – Maximum number of utxo's to return

**Return list** The created UTXO set

**mempool(txid="")**

**sendrawtransaction(rawtx)**

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

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

**Return str** Transaction hash

### 8.18.1.7 bitcoinlib.services.bitgo module

**class** bitcoinlib.services.bitgo.BitGoClient(*network, base\_url, denominator, \*args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**blockcount()**

**compose\_request(category, data, cmd="", variables=None, method='get')**

**estimatefee(blocks)**

**getbalance(addresslist)**

**getutxos(address, after\_txid="", limit=20)**

#### 8.18.1.8 bitcoinlib.services.blockchaininfo module

**class** bitcoinlib.services.blockchaininfo.**BlockchainInfoClient**(*network, base\_url, denominator, \*args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**blockcount**()

**compose\_request**(*cmd, parameter="", variables=None, method='get'*)

**getbalance**(*addresslist*)

**getblock**(*blockid, parse\_transactions, page, limit*)

**getinfo**()

**getrawblock**(*blockid*)

**getrawtransaction**(*txid*)

**gettransaction**(*txid, latest\_block=None*)

**gettransactions**(*address, after\_txid="", limit=20*)

**getutxos**(*address, after\_txid="", limit=20*)

**mempool**(*txid=""*)

#### 8.18.1.9 bitcoinlib.services.blockchair module

**class** bitcoinlib.services.blockchair.**BlockChairClient**(*network, base\_url, denominator, \*args*)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**blockcount**()

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

**Return int**

**compose\_request**(*command, query\_vars=None, variables=None, data=None, offset=0, limit=100, method='get'*)

**estimatefee**(*blocks*)

**getbalance**(*addresslist*)

**getblock**(*blockid, parse\_transactions, page, limit*)

**getinfo**()

**getrawblock**(*blockid*)

**getrawtransaction**(*txid*)

**gettransaction**(*tx\_id*)

**gettransactions**(*address, after\_txid="", limit=20*)

**getutxos**(*address, after\_txid="", limit=20*)

**isspent**(*txid, output\_n*)

**mempool**(*txid=""*)

**sendrawtransaction**(*rawtx*)



#### 8.18.1.10 bitcoinlib.services.blockcypher module

```
class bitcoinlib.services.blockcypher.BlockCypher(network, base_url, denominator, *args)
    Bases: bitcoinlib.services.baseclient.BaseClient

    blockcount()

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

    estimatefee(blocks)

    getbalance(addresslist)

    getblock(blockid, parse_transactions, page, limit)

    getrawtransaction(txid)

    gettransaction(txid)

    gettransactions(address, after_txid='', limit=20)

    getutxos(address, after_txid='', limit=20)

    isspent(txid, output_n)

    mempool(txid)

    sendrawtransaction(rawtx)
```

#### 8.18.1.11 bitcoinlib.services.blocksmurfer module

```
class bitcoinlib.services.blocksmurfer.BlocksMurferClient(network, base_url, denominator, *args)
    Bases: bitcoinlib.services.baseclient.BaseClient

    blockcount()

    compose_request(function, parameter='', parameter2='', variables=None, post_data='', method='get')

    estimatefee(blocks)

    getbalance(addresslist)

    getblock(blockid, parse_transactions, page, limit)

    getinfo()

    getrawtransaction(txid)

    gettransaction(txid)

    gettransactions(address, after_txid='', limit=20)

    getutxos(address, after_txid='', limit=20)

    isspent(txid, output_n)

    mempool(txid)

    sendrawtransaction(rawtx)
```

#### 8.18.1.12 bitcoinlib.services.blockstream module

```
class bitcoinlib.services.blockstream.BlockstreamClient(network, base_url, denominator, *args)
    Bases: bitcoinlib.services.baseclient.BaseClient

    blockcount()

    compose_request(function, data="", parameter="", parameter2="", variables=None, post_data="",
                    method='get')

    estimatefee(blocks)

    getbalance(addresslist)

    getblock(blockid, parse_transactions, page, limit)

    getrawblock(blockid)

    getrawtransaction(txid)

    gettransaction(txid, blockcount=None)

    gettransactions(address, after_txid="", limit=20)

    getutxos(address, after_txid="", limit=20)

    isspent(txid, output_n)

    mempool(txid)

    sendrawtransaction(rawtx)
```

#### 8.18.1.13 bitcoinlib.services.chainso module

```
class bitcoinlib.services.chainso.ChainSo(network, base_url, denominator, *args)
    Bases: bitcoinlib.services.baseclient.BaseClient

    blockcount()

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

    getbalance(addresslist)

    getblock(blockid, parse_transactions, page, limit)

    getinfo()

    getrawtransaction(txid)

    gettransaction(txid, block_height=None)

    gettransactions(address, after_txid="", limit=20)

    getutxos(address, after_txid="", limit=20)

    mempool(txid)

    sendrawtransaction(rawtx)
```

#### 8.18.1.14 bitcoinlib.services.coinfees module

```
class bitcoinlib.services.coinfees.CoinfeesClient(network, base_url, denominator, *args)
    Bases: bitcoinlib.services.baseclient.BaseClient

    compose_request(category, cmd, method='get')
    estimatefee(blocks)
```

#### 8.18.1.15 bitcoinlib.services.cryptoid module

```
class bitcoinlib.services.cryptoid.CryptoID(network, base_url, denominator, *args)
    Bases: bitcoinlib.services.baseclient.BaseClient

    blockcount()
    compose_request(func=None, path_type='api', variables=None, method='get')
    getbalance(addresslist)
    getrawtransaction(txid)
    gettransaction(txid)
    gettransactions(address, after_txid="", limit=20)
    getutxos(address, after_txid="", limit=20)
    mempool(txid)
```

#### 8.18.1.16 bitcoinlib.services.dashd module

```
exception bitcoinlib.services.dashd.ConfigError(msg="")
    Bases: Exception

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

    Class to interact with dashd, the Dash deamon
    Open connection to dashcore node

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

    Type str
    Type str
    Type str

    blockcount()
    estimatefee(blocks)
    static from_config(configfile=None, network='dash')
        Read settings from dashd config file
```

**Parameters**

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

**Type** str**Type** str**Return DashdClient****getblock**(*blockid*, *parse\_transactions=True*, *page=1*, *limit=None*)**getinfo**()**getrawblock**(*blockid*)**getrawtransaction**(*txid*)**gettransaction**(*txid*)**getutxos**(*address*, *after\_txid=""*, *limit=20*)**isspent**(*txid*, *index*)**sendrawtransaction**(*rawtx*)**8.18.1.17 bitcoinlib.services.dogecoin module****exception** bitcoinlib.services.dogecoin.**ConfigError**(*msg=""*)

Bases: Exception

**class** bitcoinlib.services.dogecoin.**DogecoinClient**(*network='dogecoin'*, *base\_url=""*,  
*denominator=100000000*, \*args)Bases: [bitcoinlib.services.baseclient.BaseClient](#)

Class to interact with dogecoin, the Dogecoin daemon

Open connection to dogecoin node

**Parameters**

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

**Type** str**Type** str**Type** str**blockcount**()**estimatefee**(*blocks*)**static from\_config**(*configfile=None*, *network='dogecoin'*)

Read settings from dogecoin config file

**Parameters**

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

Type str

Type str

Return DogecoinClient

**getinfo()**

**getrawtransaction(txid)**

**gettransaction(txid, block\_height=None, get\_input\_values=True)**

**getutxos(address, after\_txid="", max\_txs=20)**

**mempool(txid="")**

**sendrawtransaction(rawtx)**

#### 8.18.1.18 bitcoinlib.services.insightdash module

**class** bitcoinlib.services.insightdash.**InsightDashClient**(network, base\_url, denominator, \*args)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**blockcount()**

**compose\_request(category, data, cmd="", variables=None, method='get', offset=0)**

**getbalance(addresslist)**

**getblock(blockid, parse\_transactions, page, limit)**

**getinfo()**

**getrawtransaction(tx\_id)**

**gettransaction(tx\_id)**

**gettransactions(address, after\_txid="", limit=20)**

**getutxos(address, after\_txid="", limit=20)**

**isspent(txid, output\_n)**

**mempool(txid)**

**sendrawtransaction(rawtx)**

#### 8.18.1.19 bitcoinlib.services.litecoinblockexplorer module

**class** bitcoinlib.services.litecoinblockexplorer.**LitecoinBlockexplorerClient**(network,  
base\_url,  
denominator,  
\*args)

Bases: *bitcoinlib.services.baseclient.BaseClient*

**blockcount()**

**compose\_request(category, data, cmd="", variables=None, method='get', offset=0)**

**estimatefee(blocks)**

**getbalance(addresslist)**

**getblock(blockid, parse\_transactions, page, limit)**

```
getinfo()
getrawtransaction(tx_id)
gettransaction(tx_id)
getutxos(address, after_txid="", limit=20)
mempool(txid)
sendrawtransaction(rawtx)
```

#### 8.18.1.20 bitcoinlib.services.litecoind module

```
exception bitcoinlib.services.litecoind.ConfigError(msg="")
```

Bases: Exception

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

Bases: *bitcoinlib.services.baseclient.BaseClient*

Class to interact with litecoind, the Litecoin daemon

Open connection to litecoin node

##### Parameters

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

Type str

Type str

Type str

```
blockcount()
```

```
estimatefee(blocks)
```

```
static from_config(configfile=None, network='litecoin')
```

Read settings from litecoind config file

##### Parameters

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

Type str

Type str

##### Return LitecoindClient

```
getblock(blockid, parse_transactions=True, page=1, limit=None)
```

```
getinfo()
```

```
getrawblock(blockid)
```

```
getrawtransaction(txid)
```

```
gettransaction(txid)
```

```
getutxos(address, after_txid="", limit=20)  
isspent(txid, index)  
mempool(txid="")  
sendrawtransaction(rawtx)
```

#### 8.18.1.21 bitcoinlib.services.litecoreio module

```
class bitcoinlib.services.litecoreio.LitecoreIOClient(network, base_url, denominator, *args)  
    Bases: bitcoinlib.services.baseclient.BaseClient  
  
    blockcount()  
  
    compose_request(category, data, cmd="", variables=None, method='get', offset=0)  
  
    getbalance(addresslist)  
  
    getblock(blockid, parse_transactions, page, limit)  
  
    getinfo()  
  
    getrawtransaction(tx_id)  
  
    gettransaction(tx_id)  
  
    gettransactions(address, after_txid="", limit=20)  
  
    getutxos(address, after_txid="", limit=20)  
  
    isspent(txid, output_n)  
  
    mempool(txid)  
  
    sendrawtransaction(rawtx)
```

#### 8.18.1.22 bitcoinlib.services.smartbit module

```
class bitcoinlib.services.smartbit.SmartbitClient(network, base_url, denominator, *args)  
    Bases: bitcoinlib.services.baseclient.BaseClient  
  
    blockcount()  
  
    compose_request(category, command="", data="", variables=None, req_type='blockchain', method='get')  
  
    getbalance(addresslist)  
  
    getblock(blockid, parse_transactions, page, limit)  
  
    getrawtransaction(txid)  
  
    gettransaction(txid)  
  
    gettransactions(address, after_txid="", limit=20)  
  
    getutxos(address, after_txid="", limit=20)  
  
    isspent(txid, output_n)  
  
    mempool(txid)  
  
    sendrawtransaction(rawtx)
```

## 8.18.2 Module contents

# 8.19 bitcoinlib.config package

## 8.19.1 Submodules

### 8.19.1.1 bitcoinlib.config.config module

`bitcoinlib.config.config.initialize_lib()`

`bitcoinlib.config.config.read_config()`

### 8.19.1.2 bitcoinlib.config.opcodes module

`bitcoinlib.config.opcodes.op()`

### 8.19.1.3 bitcoinlib.config.secp256k1 module

## 8.19.2 Module contents

# 8.20 bitcoinlib.db module

**class** `bitcoinlib.db.Db(db_uri=None, password=None)`

Bases: `object`

Bitcoinlib Database object used by `Service()` and `HDWallet()` class. Initialize database and open session when creating database object.

Create new database if it doesn't exist yet

**drop\_db**(*yes\_i\_am\_sure=False*)

**class** `bitcoinlib.db.DbConfig(**kwargs)`

Bases: `sqlalchemy.ext.declarative.api.Base`

BitcoinLib configuration variables

A simple constructor that allows initialization from kwargs.

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

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

**value**

**variable**

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

Bases: `sqlalchemy.ext.declarative.api.Base`

Database definitions for keys in SQLAlchemy format

Part of a wallet, and used by transactions

A simple constructor that allows initialization from kwargs.

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



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

**account\_id**

ID of account if key is part of a HD structure

**address**

Address representation of key. An cryptocurrency address is a hash of the public key

**address\_index**

Index of address in HD key structure address level

**balance**

Total balance of UTXO's linked to this key

**change**

Change or normal address: Normal=0, Change=1

**compressed**

Is key compressed or not. Default is True

**cosigner\_id**

ID of cosigner, used if key is part of HD Wallet

**depth**

Depth of key if it is part of a HD structure. Depth=0 means masterkey, depth=1 are the masterkeys children.

**encoding**

Encoding used to represent address: base58 or bech32

**id**

Unique Key ID

**is\_private**

Is key private or not?

**key\_type**

Type of key: single, bip32 or multisig. Default is bip32

**latest\_txid**

Txid of latest transaction downloaded from the blockchain

**multisig\_children**

List of children keys

**multisig\_parents**

List of parent keys

**name**

Key name string

**network**

DbNetwork object for this key

**network\_name**

Name of key network, i.e. bitcoin, litecoin, dash

**parent\_id**

Parent Key ID. Used in HD wallets

**path**

String of BIP-32 key path

**private**

Bytes representation of private key

**public**

Bytes representation of public key

**purpose**

Purpose ID, default is 44

**transaction\_inputs**

All DbTransactionInput objects this key is part of

**transaction\_outputs**

All DbTransactionOutput objects this key is part of

**used**

Has key already been used on the blockchain in as input or output? Default is False

**wallet**

Related Wallet object

**wallet\_id**

Wallet ID which contains this key

**wif**

Public or private WIF (Wallet Import Format) representation

**class** bitcoinlib.db.DbKeyMultisigChildren(\*\*kwargs)

Bases: sqlalchemy.ext.declarative.api.Base

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

A simple constructor that allows initialization from kwargs.

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

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

**child\_id****key\_order****parent\_id**

**class** bitcoinlib.db.DbNetwork(\*\*kwargs)

Bases: sqlalchemy.ext.declarative.api.Base

Database definitions for networks in SQLAlchemy format

Most network settings and variables can be found outside the database in the libraries configurations settings. Use the bitcoinlib/data/networks.json file to view and manage settings.

A simple constructor that allows initialization from kwargs.

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

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

**description****name**

Network name, i.e.: bitcoin, litecoin, dash

**class** bitcoinlib.db.DbTransaction(\*\*kwargs)

Bases: sqlalchemy.ext.declarative.api.Base

Database definitions for transactions in SQLAlchemy format

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

A simple constructor that allows initialization from kwargs.

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

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

**account\_id**

ID of account

**block\_height**

Number of block this transaction is included in

**coinbase**

Is True when this is a coinbase transaction, default is False

**confirmations**

Number of confirmation when this transaction is included in a block. Default is 0: unconfirmed

**date**

Date when transaction was confirmed and included in a block. Or when it was created when transaction is not send or confirmed

**fee**

Transaction fee

**id**

Unique transaction index for internal usage

**input\_total**

Total value of the inputs of this transaction. Input total = Output total + fee. Default is 0

**inputs**

List of all inputs as DbTransactionInput objects

**is\_complete**

Allow to store incomplete transactions, for instance if not all inputs are known when retrieving UTXO's

**locktime**

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

**network**

Link to DbNetwork object

**network\_name**

Blockchain network name of this transaction

**output\_total**

Total value of the outputs of this transaction. Output total = Input total - fee

**outputs**

List of all outputs as DbTransactionOutput objects

**raw**

Raw transaction hexadecimal string. Transaction is included in raw format on the blockchain

**size**

Size of the raw transaction in bytes

**status**

Current status of transaction, can be one of the following: 'new', 'unconfirmed', 'confirmed'. Default is 'new'

**txid**

Bytes representation of transaction ID

**verified**

Is transaction verified. Default is False

**version**

Transaction version. Default is 1 but some wallets use another version number

**wallet**

Link to Wallet object which contains this transaction

**wallet\_id**

ID of wallet which contains this transaction

**witness\_type**

Is this a legacy or segwit transaction?

**class** bitcoinlib.db.DbTransactionInput(\*\*kwargs)

Bases: sqlalchemy.ext.declarative.api.Base

Transaction Input Table

Relates to Transaction table and Key table

A simple constructor that allows initialization from kwargs.

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

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

**address**

Address string of input, used if no key is associated. An cryptocurrency address is a hash of the public key or a redeemscript

**double\_spend**

Indicates if a service provider tagged this transaction as double spend

**index\_n**

Index number of transaction input

**key**

Related DbKey object

**key\_id**

ID of key used in this input

**output\_n**

Output\_n of previous transaction output that is spent in this input

**prev\_txid**

Transaction hash of previous transaction. Previous unspent outputs (UTXO) is spent in this input

**script**

Unlocking script to unlock previous locked output

**script\_type**

Unlocking script type. Can be 'coinbase', 'sig\_pubkey', 'p2sh\_multisig', 'signature', 'unknown', 'p2sh\_p2wpkh' or 'p2sh\_p2wsh'. Default is sig\_pubkey

**sequence**

Transaction sequence number. Used for timelock transaction inputs

**transaction**

Related DbTransaction object

**transaction\_id**

Input is part of transaction with this ID

**value**

Value of transaction input

**witness\_type**

Type of transaction, can be legacy, segwit or p2sh-segwit. Default is legacy

**class** bitcoinlib.db.DbTransactionOutput(\*\*kwargs)

Bases: sqlalchemy.ext.declarative.api.Base

Transaction Output Table

Relates to Transaction and Key table

When spent is False output is considered an UTXO

A simple constructor that allows initialization from kwargs.

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

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

**address**

Address string of output, used if no key is associated. An cryptocurrency address is a hash of the public key or a redeemscript

**key**

List of DbKey object used in this output

**key\_id**

ID of key used in this transaction output

**output\_n**

Sequence number of transaction output

**script**

Locking script which locks transaction output

**script\_type**

Locking script type. Can be one of these values: 'p2pkh', 'multisig', 'p2sh', 'p2pk', 'nulldata', 'unknown', 'p2wpkh' or 'p2wsh'. Default is p2pkh

**spending\_index\_n**

Index number of transaction input which spends this output

**spending\_txid**

Transaction hash of input which spends this output

**spent**

Indicated if output is already spent in another transaction

**transaction**

Link to transaction object

**transaction\_id**

Transaction ID of parent transaction

**value**

Total transaction output value

**class** bitcoinlib.db.DbWallet(\*\*kwargs)

Bases: sqlalchemy.ext.declarative.api.Base

Database definitions for wallets in Sqlalchemy format

Contains one or more keys.

A simple constructor that allows initialization from kwargs.

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

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

**children**

Wallet IDs of children wallets, used in multisig wallets

**cosigner\_id**

ID of cosigner of this wallet. Used in multisig wallets to differentiate between different wallets

**default\_account\_id**

ID of default account for this wallet if multiple accounts are used

**encoding**

Default encoding to use for address generation, i.e. base58 or bech32. Default is base58.

**id**

Unique wallet ID

**key\_path**

Key path structure used in this wallet. Key path for multisig wallet, use to create your own non-standard key path. Key path must follow the following rules: \* Path start with masterkey (m) and end with change / address\_index \* If accounts are used, the account level must be 3. I.e.: m/purpose/coin\_type/account/ \* All keys must be hardened, except for change, address\_index or cosigner\_id Max length of path is 8 levels

**keys**

Link to keys (DbKeys objects) in this wallet

**main\_key\_id**

Masterkey ID for this wallet. All other keys are derived from the masterkey in a HD wallet bip32 wallet

**multisig**

Indicates if wallet is a multisig wallet. Default is True

**multisig\_n\_required**

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

**name**

Unique wallet name

**network**

Link to DbNetwork object

**network\_name**

Name of network, i.e.: bitcoin, litecoin

**owner**

Wallet owner

**parent\_id**

Wallet ID of parent wallet, used in multisig wallets

**purpose**

Wallet purpose ID. BIP-44 purpose field, indicating which key-scheme is used default is 44

**scheme**

Key structure type, can be BIP-32 or single

**sort\_keys**

Sort keys in multisig wallet

**transactions**

Link to transaction (DbTransactions) in this wallet

**witness\_type**

Wallet witness type. Can be 'legacy', 'segwit' or 'p2sh-segwit'. Default is legacy.

`bitcoinlib.db.add_column(engine, table_name, column)`

Used to add new column to database with migration and update scripts

**Parameters**

- **engine** –
- **table\_name** –
- **column** –

**Returns**

`bitcoinlib.db.compile_largebinary_mysql(type_, compiler, **kwargs)`

`bitcoinlib.db.db_update(db, version_db, code_version='0.6.1')`

`bitcoinlib.db.db_update_version_id(db, version)`

## 8.21 bitcoinlib.db\_cache module

**class** `bitcoinlib.db_cache.DbCache(db_uri=None)`

Bases: `object`

Cache Database object. Initialize database and open session when creating database object.

Create new database if it doesn't exist yet

**drop\_db()**

**class** `bitcoinlib.db_cache.DbCacheAddress(**kwargs)`

Bases: `sqlalchemy.ext.declarative.api.Base`

Address Cache Table

Stores transactions and unspent outputs (UTXO's) per address

A simple constructor that allows initialization from kwargs.

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

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

**address**

Address string base32 or base58 encoded

**balance**

Total balance of UTXO's linked to this key

**last\_block**

Number of last updated block

**last\_txid**

Transaction ID of latest transaction in cache

**n\_txs**

Total number of transactions for this address

**n\_utxos**

Total number of UTXO's for this address

**network\_name**

Blockchain network name of this transaction

**class** bitcoinlib.db\_cache.DbCacheBlock(\*\*kwargs)

Bases: sqlalchemy.ext.declarative.api.Base

Block Cache Table

Stores block headers

A simple constructor that allows initialization from kwargs.

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

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

**bits**

Encoding for proof-of-work, used to determine target and difficulty

**block\_hash**

Hash of this block

**height**

Height or sequence number for this block

**merkle\_root**

Merkle root used to validate transaction in block

**network\_name**

Blockchain network name

**nonce**

Nonce (number used only once or n-once) is used to create different block hashes

**prev\_block**

Block hash of previous block

**time**

Timestamp to indicated when block was created

**tx\_count**

Number of transactions included in this block

**version**

Block version to specify which features are used (hex)



```
class bitcoinlib.db_cache.DbCacheTransaction(**kwargs)
```

Bases: sqlalchemy.ext.declarative.api.Base

Transaction Cache Table

Database which stores transactions received from service providers as cache

A simple constructor that allows initialization from kwargs.

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

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

**block\_height**

Height of block this transaction is included in

**confirmations**

Number of confirmation when this transaction is included in a block. Default is 0: unconfirmed

**date**

Date when transaction was confirmed and included in a block

**fee**

Transaction fee

**locktime**

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

**network\_name**

Blockchain network name of this transaction

**nodes**

List of all inputs and outputs as DbCacheTransactionNode objects

**order\_n**

Order of transaction in block

**txid**

Hexadecimal representation of transaction hash or transaction ID

**version**

Transaction version. Default is 1 but some wallets use another version number

**witness\_type**

Transaction type enum: legacy or segwit

```
class bitcoinlib.db_cache.DbCacheTransactionNode(**kwargs)
```

Bases: sqlalchemy.ext.declarative.api.Base

Link table for cache transactions and addresses

A simple constructor that allows initialization from kwargs.

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

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

**address**

Address string base32 or base58 encoded

**index\_n**

Order of input/output in this transaction

**is\_input**  
True if input, False if output

**output\_n()**

**prev\_txid()**

**ref\_index\_n**  
Index number of transaction input which spends this output

**ref\_txid**  
Transaction hash of input which spends this output

**script**  
Locking or unlocking script

**sequence**  
Transaction sequence number. Used for timelock transaction inputs

**spending\_index\_n()**

**spending\_txid()**

**spent**  
Is output spent?

**transaction**  
Related transaction object

**txid**

**value**  
Value of transaction input

**witnesses**  
Witnesses (signatures) used in Segwit transaction inputs

**class** bitcoinlib.db\_cache.**DbCacheVars**(\*\*kwargs)

Bases: sqlalchemy.ext.declarative.api.Base

Table to store various blockchain related variables

A simple constructor that allows initialization from kwargs.

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

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

**expires**  
Datetime value when variable expires

**network\_name**  
Blockchain network name of this transaction

**type**  
Type of variable: int, string or float

**value**  
Value of variable

**varname**  
Variable unique name

```
class bitcoinlib.db_cache.WitnessTypeTransactions(value)
    Bases: enum.Enum

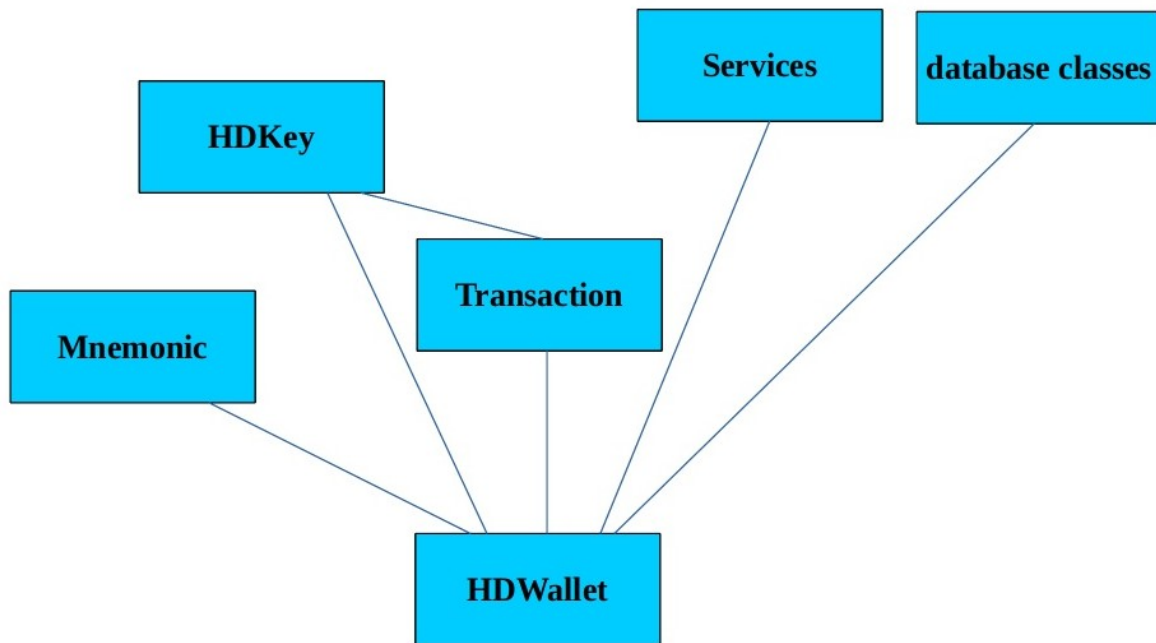
    An enumeration.

    legacy = 'legacy'
    segwit = 'segwit'
```

## 8.22 Classes Overview

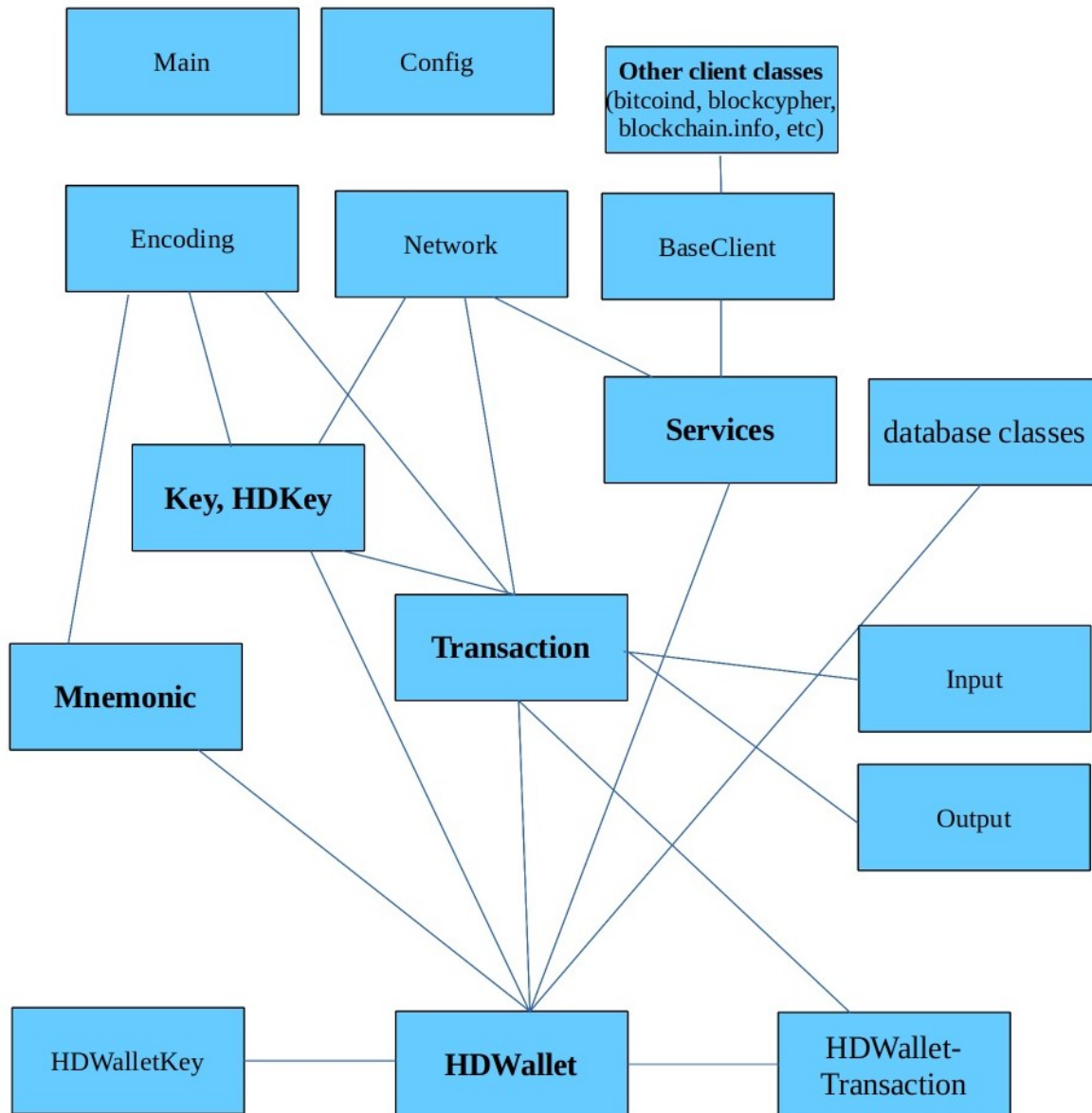
These are the main Bitcoinlib classes

### BitcoinLib Main Classes



This is an overview of all BitcoinLib classes.

## BitcoinLib Classes and Containers



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.

## 8.23 bitcoinlib

### 8.23.1 bitcoinlib package

#### 8.23.1.1 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

#### 8.23.1.2 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 tuple (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(
  ↳ 'xpub661MyMwAqRbcFnkbk13gaJba22ibnEdJS7KAMY99C4jBBHMxWacBStrTinNTc9G5LTFtUqbLpWnzY5yPTNEF9u8sB1kF
  ↳ ', 58, 16)

  ↳ '0488b21e0000000000000000000000007d3cc6702f48bf618f3f14cce5ee2cacf3f70933345ee4710af6fa4a330cc7d503c04
  ↳ '
```

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](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](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 + separator + 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.read_varbyteint(s)`

Read variable length integer from BytesIO stream. Wrapper for the `varbyteint_to_int` method

**Parameters** *s* (*BytesIO*) – A binary stream

**Return** *int*

`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](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('5468697320737472696e67206861732061206c656e677468206f66203330
↪')).hex()
'1e5468697320737472696e67206861732061206c656e677468206f66203330'
```

**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

### 8.23.1.3 Module contents

## 8.24 Script types

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

They are defined in `main.py`

### 8.24.1 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

### 8.24.2 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

### 8.24.3 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 than the one listed in the Locking Script table is possible but not advised: It is not standard and not sufficiently tested.

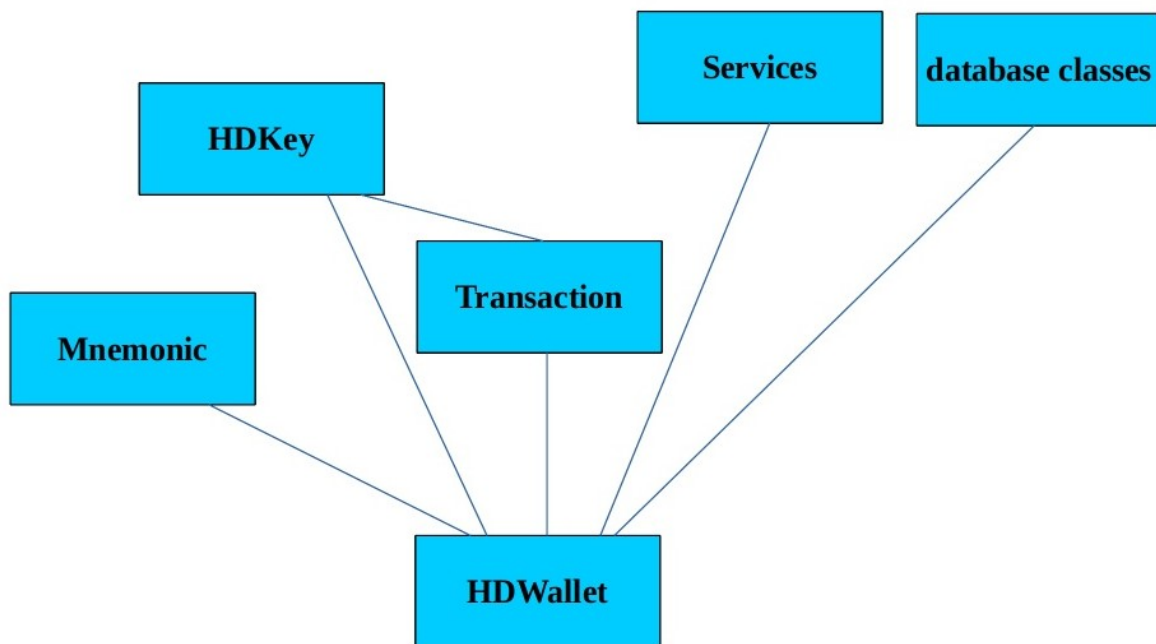
## **DISCLAIMER**

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

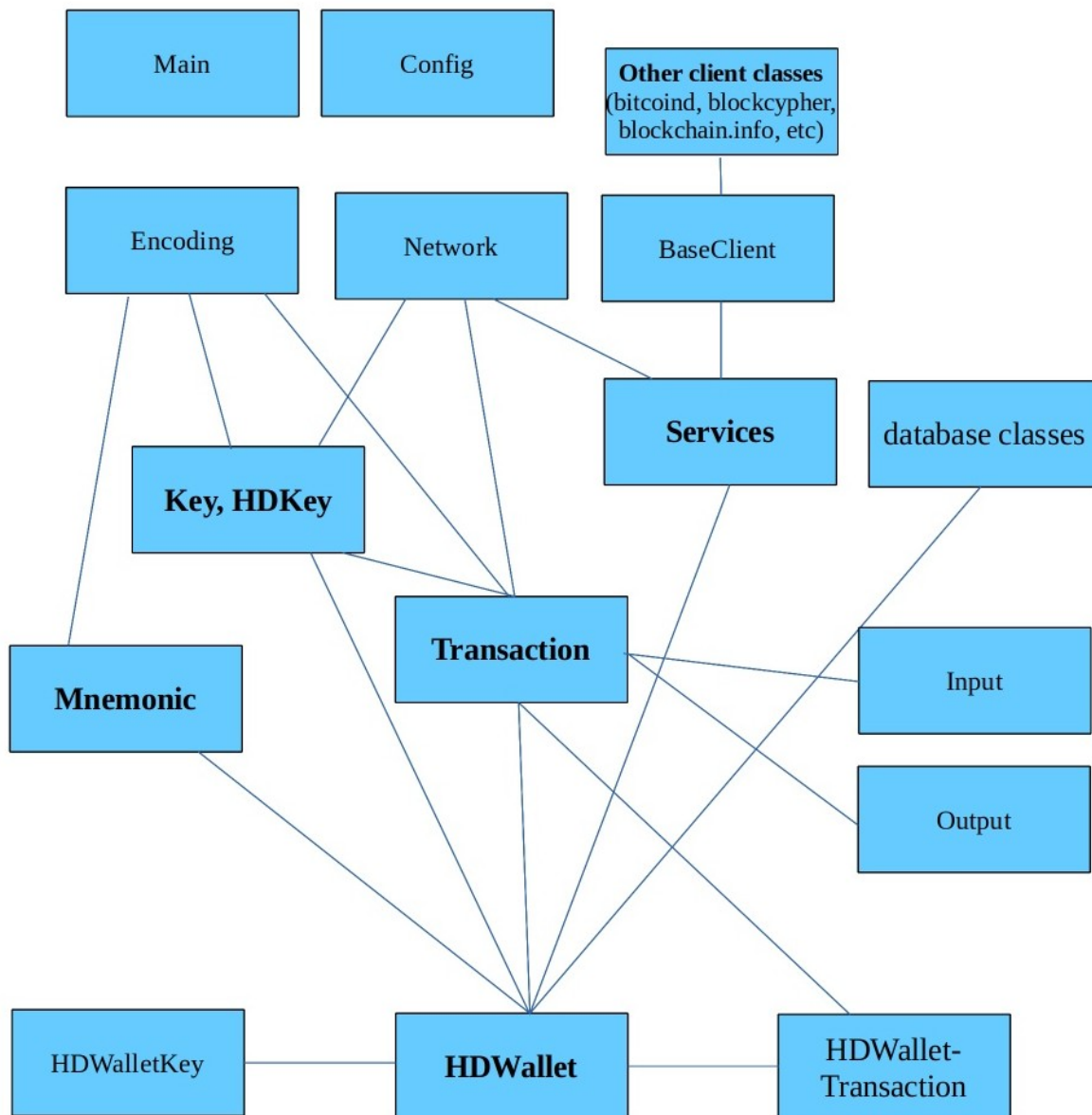


## SCHEMATIC OVERVIEW

### BitcoinLib Main Classes



## BitcoinLib Classes and Containers





## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### b

- [bitcoinlib](#), 141
- [bitcoinlib.blocks](#), 95
- [bitcoinlib.config](#), 122
- [bitcoinlib.config.config](#), 122
- [bitcoinlib.config.opcodes](#), 122
- [bitcoinlib.config.secp256k1](#), 122
- [bitcoinlib.db](#), 122
- [bitcoinlib.db\\_cache](#), 129
- [bitcoinlib.encoding](#), 135
- [bitcoinlib.keys](#), 30
- [bitcoinlib.main](#), 141
- [bitcoinlib.mnemonic](#), 91
- [bitcoinlib.networks](#), 93
- [bitcoinlib.scripts](#), 60
- [bitcoinlib.services](#), 122
- [bitcoinlib.services.authproxy](#), 110
- [bitcoinlib.services.baseclient](#), 111
- [bitcoinlib.services.bcoin](#), 111
- [bitcoinlib.services.bitaps](#), 111
- [bitcoinlib.services.bitcoind](#), 112
- [bitcoinlib.services.bitcoinlibtest](#), 113
- [bitcoinlib.services.bitgo](#), 113
- [bitcoinlib.services.blockchaininfo](#), 114
- [bitcoinlib.services.blockchair](#), 114
- [bitcoinlib.services.blockcypher](#), 115
- [bitcoinlib.services.blocksmurfer](#), 115
- [bitcoinlib.services.blockstream](#), 116
- [bitcoinlib.services.chainso](#), 116
- [bitcoinlib.services.coinfees](#), 117
- [bitcoinlib.services.cryptoid](#), 117
- [bitcoinlib.services.dashd](#), 117
- [bitcoinlib.services.dogecoin](#), 118
- [bitcoinlib.services.insightdash](#), 119
- [bitcoinlib.services.litecoinblockexplorer](#), 119
- [bitcoinlib.services.litecoind](#), 120
- [bitcoinlib.services.litecoreio](#), 121
- [bitcoinlib.services.services](#), 104
- [bitcoinlib.services.smartbit](#), 121
- [bitcoinlib.tools](#), 135
- [bitcoinlib.tools.clw](#), 135
- [bitcoinlib.tools.mnemonic\\_key\\_create](#), 135
- [bitcoinlib.tools.sign\\_raw](#), 135
- [bitcoinlib.tools.wallet\\_multisig\\_2of3](#), 135
- [bitcoinlib.transactions](#), 47
- [bitcoinlib.values](#), 100
- [bitcoinlib.wallets](#), 66



## A

account() (bitcoinlib.wallets.Wallet method), 67  
 account\_id (bitcoinlib.db.DbKey attribute), 123  
 account\_id (bitcoinlib.db.DbTransaction attribute), 125  
 accounts() (bitcoinlib.wallets.Wallet method), 67  
 add\_column() (in module bitcoinlib.db), 129  
 add\_input() (bitcoinlib.transactions.Transaction method), 51  
 add\_output() (bitcoinlib.transactions.Transaction method), 52  
 addr\_base58\_to\_pubkeyhash() (in module bitcoinlib.encoding), 135  
 addr\_bech32\_to\_pubkeyhash() (in module bitcoinlib.encoding), 136  
 addr\_convert() (in module bitcoinlib.keys), 43  
 addr\_to\_pubkeyhash() (in module bitcoinlib.encoding), 136  
 address (bitcoinlib.db.DbKey attribute), 123  
 address (bitcoinlib.db.DbTransactionInput attribute), 126  
 address (bitcoinlib.db.DbTransactionOutput attribute), 127  
 address (bitcoinlib.db\_cache.DbCacheAddress attribute), 129  
 address (bitcoinlib.db\_cache.DbCacheTransactionNode attribute), 131  
 Address (class in bitcoinlib.keys), 30  
 address() (bitcoinlib.keys.HDKey method), 33  
 address() (bitcoinlib.keys.Key method), 39  
 address\_index (bitcoinlib.db.DbKey attribute), 123  
 address\_obj (bitcoinlib.keys.Key property), 39  
 address\_uncompressed() (bitcoinlib.keys.Key method), 39  
 addresslist() (bitcoinlib.wallets.Wallet method), 67  
 as\_der\_encoded() (bitcoinlib.keys.Signature method), 41  
 as\_dict() (bitcoinlib.blocks.Block method), 96  
 as\_dict() (bitcoinlib.keys.Address method), 30  
 as\_dict() (bitcoinlib.keys.HDKey method), 33  
 as\_dict() (bitcoinlib.keys.Key method), 39  
 as\_dict() (bitcoinlib.transactions.Input method), 48  
 as\_dict() (bitcoinlib.transactions.Output method), 49

as\_dict() (bitcoinlib.transactions.Transaction method), 53  
 as\_dict() (bitcoinlib.wallets.Wallet method), 67  
 as\_dict() (bitcoinlib.wallets.WalletKey method), 86  
 as\_ints() (bitcoinlib.scripts.Stack method), 63  
 as\_json() (bitcoinlib.keys.Address method), 30  
 as\_json() (bitcoinlib.keys.HDKey method), 33  
 as\_json() (bitcoinlib.keys.Key method), 39  
 as\_json() (bitcoinlib.transactions.Transaction method), 53  
 as\_json() (bitcoinlib.wallets.Wallet method), 67  
 AuthServiceProxy (class in bitcoinlib.services.authproxy), 110

## B

balance (bitcoinlib.db.DbKey attribute), 123  
 balance (bitcoinlib.db\_cache.DbCacheAddress attribute), 130  
 balance() (bitcoinlib.wallets.Wallet method), 68  
 balance() (bitcoinlib.wallets.WalletKey method), 86  
 balance\_update\_from\_serviceprovider() (bitcoinlib.wallets.Wallet method), 68  
 BaseClient (class in bitcoinlib.services.baseclient), 111  
 batch\_() (bitcoinlib.services.authproxy.AuthServiceProxy method), 110  
 BcoinClient (class in bitcoinlib.services.bcoin), 111  
 bip38\_decrypt() (in module bitcoinlib.encoding), 136  
 bip38\_encrypt() (bitcoinlib.keys.HDKey method), 33  
 bip38\_encrypt() (bitcoinlib.keys.Key method), 39  
 bip38\_encrypt() (in module bitcoinlib.encoding), 136  
 BitapsClient (class in bitcoinlib.services.bitaps), 111  
 BitcoindClient (class in bitcoinlib.services.bitcoind), 112  
 bitcoinlib  
     module, 141  
 bitcoinlib.blocks  
     module, 95  
 bitcoinlib.config  
     module, 122  
 bitcoinlib.config.config  
     module, 122  
 bitcoinlib.config.opcodes

- module, 122
- bitcoinlib.config.secp256k1
  - module, 122
- bitcoinlib.db
  - module, 122
- bitcoinlib.db\_cache
  - module, 129
- bitcoinlib.encoding
  - module, 135
- bitcoinlib.keys
  - module, 30
- bitcoinlib.main
  - module, 141
- bitcoinlib.mnemonic
  - module, 91
- bitcoinlib.networks
  - module, 93
- bitcoinlib.scripts
  - module, 60
- bitcoinlib.services
  - module, 122
- bitcoinlib.services.authproxy
  - module, 110
- bitcoinlib.services.baseclient
  - module, 111
- bitcoinlib.services.bcoin
  - module, 111
- bitcoinlib.services.bitaps
  - module, 111
- bitcoinlib.services.bitcoind
  - module, 112
- bitcoinlib.services.bitcoinlibtest
  - module, 113
- bitcoinlib.services.bitgo
  - module, 113
- bitcoinlib.services.blockchaininfo
  - module, 114
- bitcoinlib.services.blockchair
  - module, 114
- bitcoinlib.services.blockcypher
  - module, 115
- bitcoinlib.services.blocksmurfer
  - module, 115
- bitcoinlib.services.blockstream
  - module, 116
- bitcoinlib.services.chainso
  - module, 116
- bitcoinlib.services.coinfees
  - module, 117
- bitcoinlib.services.cryptoid
  - module, 117
- bitcoinlib.services.dashd
  - module, 117
- bitcoinlib.services.dogecoin

- module, 118
- bitcoinlib.services.insightdash
  - module, 119
- bitcoinlib.services.litecoinblockexplorer
  - module, 119
- bitcoinlib.services.litecoind
  - module, 120
- bitcoinlib.services.litecoreio
  - module, 121
- bitcoinlib.services.services
  - module, 104
- bitcoinlib.services.smartbit
  - module, 121
- bitcoinlib.tools
  - module, 135
- bitcoinlib.tools.clw
  - module, 135
- bitcoinlib.tools.mnemonic\_key\_create
  - module, 135
- bitcoinlib.tools.sign\_raw
  - module, 135
- bitcoinlib.tools.wallet\_multisig\_2of3
  - module, 135
- bitcoinlib.transactions
  - module, 47
- bitcoinlib.values
  - module, 100
- bitcoinlib.wallets
  - module, 66
- BitcoinLibTestClient (*class in bitcoinlib.services.bitcoinlibtest*), 113
- BitGoClient (*class in bitcoinlib.services.bitgo*), 113
- bits (*bitcoinlib.db\_cache.DbCacheBlock attribute*), 130
- BKeyError, 32
- Block (*class in bitcoinlib.blocks*), 95
- block\_hash (*bitcoinlib.db\_cache.DbCacheBlock attribute*), 130
- block\_height (*bitcoinlib.db.DbTransaction attribute*), 125
- block\_height (*bitcoinlib.db\_cache.DbCacheTransaction attribute*), 131
- BlockchainInfoClient (*class in bitcoinlib.services.blockchaininfo*), 114
- BlockChairClient (*class in bitcoinlib.services.blockchair*), 114
- blockcount() (*bitcoinlib.services.bcoin.BcoinClient method*), 111
- blockcount() (*bitcoinlib.services.bitaps.BitapsClient method*), 111
- blockcount() (*bitcoinlib.services.bitcoind.BitcoindClient method*), 112

- `blockcount()` (*bitcoinlib.services.bitcoinlibtest.BitcoinLibTestClient method*), 113
- `blockcount()` (*bitcoinlib.services.bitgo.BitGoClient method*), 113
- `blockcount()` (*bitcoinlib.services.blockchaininfo.BlockchainInfoClient method*), 114
- `blockcount()` (*bitcoinlib.services.blockchair.BlockChairClient method*), 114
- `blockcount()` (*bitcoinlib.services.blockcypher.BlockCypher method*), 115
- `blockcount()` (*bitcoinlib.services.blocksmurfer.BlocksMurferClient method*), 115
- `blockcount()` (*bitcoinlib.services.blockstream.BlockstreamClient method*), 116
- `blockcount()` (*bitcoinlib.services.chainso.ChainSo method*), 116
- `blockcount()` (*bitcoinlib.services.cryptoid.CryptoID method*), 117
- `blockcount()` (*bitcoinlib.services.dashd.DashdClient method*), 117
- `blockcount()` (*bitcoinlib.services.dogecoin.DogecoinClient method*), 118
- `blockcount()` (*bitcoinlib.services.insightdash.InsightDashClient method*), 119
- `blockcount()` (*bitcoinlib.services.litecoinblockexplorer.LitecoinBlockexplorerClient method*), 119
- `blockcount()` (*bitcoinlib.services.litecoind.LitecoindClient method*), 120
- `blockcount()` (*bitcoinlib.services.litecoreio.LitecoreIOClient method*), 121
- `blockcount()` (*bitcoinlib.services.services.Cache method*), 104
- `blockcount()` (*bitcoinlib.services.services.Service method*), 107
- `blockcount()` (*bitcoinlib.services.smartbit.SmartbitClient method*), 121
- `BlockCypher` (class in *bitcoinlib.services.blockcypher*), 115
- `BlocksmurferClient` (class in *bitcoinlib.services.blocksmurfer*), 115
- `BlockstreamClient` (class in *bitcoinlib.services.blockstream*), 116
- `blueprint` (*bitcoinlib.scripts.Script property*), 60
- `bytes()` (*bitcoinlib.keys.Signature method*), 41
- ## C
- `Cache` (class in *bitcoinlib.services.services*), 104
- `cache_enabled()` (*bitcoinlib.services.services.Cache method*), 104
- `calc_weight_units()` (*bitcoinlib.transactions.Transaction method*), 53
- `calculate_fee()` (*bitcoinlib.transactions.Transaction method*), 53
- `ChainSo` (class in *bitcoinlib.services.chainso*), 116
- `change` (*bitcoinlib.db.DbKey attribute*), 123
- `change_base()` (in module *bitcoinlib.encoding*), 137
- `check_network_and_key()` (in module *bitcoinlib.keys*), 43
- `check_proof_of_work()` (*bitcoinlib.blocks.Block method*), 96
- `checksum()` (*bitcoinlib.mnemonic.Mnemonic static method*), 91
- `child_id` (*bitcoinlib.db.DbKeyMultisigChildren attribute*), 124
- `child_private()` (*bitcoinlib.keys.HDKey method*), 34
- `child_public()` (*bitcoinlib.keys.HDKey method*), 34
- `children` (*bitcoinlib.db.DbWallet attribute*), 128
- `ClientError`, 111
- `coinbase` (*bitcoinlib.db.DbTransaction attribute*), 125
- `CoinfeesClient` (class in *bitcoinlib.services.coinfees*), 117
- `commit()` (*bitcoinlib.services.services.Cache method*), 104
- `compile_largebinary_mysql()` (in module *bitcoinlib.db*), 129
- `compose_request()` (*bitcoinlib.services.bcoin.BcoinClient method*), 111
- `compose_request()` (*bitcoinlib.services.bitaps.BitapsClient method*), 111
- `compose_request()` (*bitcoinlib.services.bitgo.BitGoClient method*), 113
- `compose_request()` (*bitcoinlib.services.blockchaininfo.BlockchainInfoClient method*), 114
- `compose_request()` (*bitcoinlib.services.blockchair.BlockChairClient method*), 114
- `compose_request()` (*bitcoinlib.services.blockcypher.BlockCypher method*), 115
- `compose_request()` (*bitcoinlib.services.blocksmurfer.BlocksMurferClient method*), 115

`compose_request()` (*bitcoinlib.services.blockstream.BlockstreamClient method*), 116  
`compose_request()` (*bitcoinlib.services.chainso.ChainSo method*), 116  
`compose_request()` (*bitcoinlib.services.coinfees.CoinfeesClient method*), 117  
`compose_request()` (*bitcoinlib.services.cryptoid.CryptoID method*), 117  
`compose_request()` (*bitcoinlib.services.insightdash.InsightDashClient method*), 119  
`compose_request()` (*bitcoinlib.services.litecoinblockexplorer.LitecoinBlockexplorerClient method*), 119  
`compose_request()` (*bitcoinlib.services.litecoreio.LitecoreIOClient method*), 121  
`compose_request()` (*bitcoinlib.services.smartbit.SmartbitClient method*), 121  
`compressed` (*bitcoinlib.db.DbKey attribute*), 123  
`ConfigError`, 112, 117, 118, 120  
`confirmations` (*bitcoinlib.db.DbTransaction attribute*), 125  
`confirmations` (*bitcoinlib.db\_cache.DbCacheTransaction attribute*), 131  
`convert_der_sig()` (*in module bitcoinlib.encoding*), 138  
`convertbits()` (*in module bitcoinlib.encoding*), 138  
`cosigner_id` (*bitcoinlib.db.DbKey attribute*), 123  
`cosigner_id` (*bitcoinlib.db.DbWallet attribute*), 128  
`create()` (*bitcoinlib.keys.Signature static method*), 41  
`create()` (*bitcoinlib.wallets.Wallet class method*), 68  
`CryptoID` (*class in bitcoinlib.services.cryptoid*), 117  
`DbCacheTransaction` (*class in bitcoinlib.db\_cache*), 130  
`DbCacheTransactionNode` (*class in bitcoinlib.db\_cache*), 131  
`DbCacheVars` (*class in bitcoinlib.db\_cache*), 132  
`DbConfig` (*class in bitcoinlib.db*), 122  
`DbKey` (*class in bitcoinlib.db*), 122  
`DbKeyMultisigChildren` (*class in bitcoinlib.db*), 124  
`DbNetwork` (*class in bitcoinlib.db*), 124  
`DbTransaction` (*class in bitcoinlib.db*), 124  
`DbTransactionInput` (*class in bitcoinlib.db*), 126  
`DbTransactionOutput` (*class in bitcoinlib.db*), 127  
`DbWallet` (*class in bitcoinlib.db*), 128  
`decode_num()` (*in module bitcoinlib.scripts*), 66  
`default_account_id` (*bitcoinlib.db.DbWallet attribute*), 128  
`default_account_id` (*bitcoinlib.wallets.Wallet property*), 70  
`default_network_set()` (*bitcoinlib.wallets.Wallet method*), 70  
`delete()` (*bitcoinlib.wallets.WalletTransaction method*), 88  
`deprecated()` (*in module bitcoinlib.main*), 141  
`depth` (*bitcoinlib.db.DbKey attribute*), 123  
`der_encode_sig()` (*in module bitcoinlib.encoding*), 138  
`description` (*bitcoinlib.db.DbNetwork attribute*), 124  
`deserialize_address()` (*in module bitcoinlib.keys*), 44  
`detect_language()` (*bitcoinlib.mnemonic.Mnemonic static method*), 91  
`difficulty` (*bitcoinlib.blocks.Block property*), 96  
`DogecoinClient` (*class in bitcoinlib.services.dogecoin*), 118  
`double_sha256()` (*in module bitcoinlib.encoding*), 138  
`double_spend` (*bitcoinlib.db.DbTransactionInput attribute*), 126  
`drop_db()` (*bitcoinlib.db.Db method*), 122  
`drop_db()` (*bitcoinlib.db\_cache.DbCache method*), 129

## D

`DashdClient` (*class in bitcoinlib.services.dashd*), 117  
`data` (*bitcoinlib.keys.Address property*), 30  
`data_pack()` (*in module bitcoinlib.scripts*), 66  
`date` (*bitcoinlib.db.DbTransaction attribute*), 125  
`date` (*bitcoinlib.db\_cache.DbCacheTransaction attribute*), 131  
`Db` (*class in bitcoinlib.db*), 122  
`db_update()` (*in module bitcoinlib.db*), 129  
`db_update_version_id()` (*in module bitcoinlib.db*), 129  
`DbCache` (*class in bitcoinlib.db\_cache*), 129  
`DbCacheAddress` (*class in bitcoinlib.db\_cache*), 129  
`DbCacheBlock` (*class in bitcoinlib.db\_cache*), 130

## E

`ec_point()` (*in module bitcoinlib.keys*), 44  
`encode_num()` (*in module bitcoinlib.scripts*), 66  
`EncodeDecimal()` (*in module bitcoinlib.services.authproxy*), 110  
`encoding` (*bitcoinlib.db.DbKey attribute*), 123  
`encoding` (*bitcoinlib.db.DbWallet attribute*), 128  
`EncodingError`, 135  
`estimate_size()` (*bitcoinlib.transactions.Transaction method*), 53  
`estimatefee()` (*bitcoinlib.services.bcoin.BcoinClient method*), 111  
`estimatefee()` (*bitcoinlib.services.bitcoind.BitcoindClient method*), 112



- `estimatefee()` (*bitcoinlib.services.bitcoinlibtest.BitcoinLibTestClient* method), 113
- `estimatefee()` (*bitcoinlib.services.bitgo.BitGoClient* method), 113
- `estimatefee()` (*bitcoinlib.services.blockchair.BlockChairClient* method), 114
- `estimatefee()` (*bitcoinlib.services.blockcypher.BlockCypher* method), 115
- `estimatefee()` (*bitcoinlib.services.blocksmurfer.BlocksmurferClient* method), 115
- `estimatefee()` (*bitcoinlib.services.blockstream.BlockstreamClient* method), 116
- `estimatefee()` (*bitcoinlib.services.coinfees.CoinfeesClient* method), 117
- `estimatefee()` (*bitcoinlib.services.dashd.DashdClient* method), 117
- `estimatefee()` (*bitcoinlib.services.dogecoin.DogecoinClient* method), 118
- `estimatefee()` (*bitcoinlib.services.litecoinblockexplorer.LitecoinBlockexplorerClient* method), 119
- `estimatefee()` (*bitcoinlib.services.litecoind.LitecoindClient* method), 120
- `estimatefee()` (*bitcoinlib.services.services.Cache* method), 104
- `estimatefee()` (*bitcoinlib.services.services.Service* method), 107
- `evaluate()` (*bitcoinlib.scripts.Script* method), 60
- `expires` (*bitcoinlib.db\_cache.DbCacheVars* attribute), 132
- `export()` (*bitcoinlib.wallets.WalletTransaction* method), 88
- ## F
- `fee` (*bitcoinlib.db.DbTransaction* attribute), 125
- `fee` (*bitcoinlib.db\_cache.DbCacheTransaction* attribute), 131
- `fingerprint` (*bitcoinlib.keys.HDKey* property), 35
- `from_config()` (*bitcoinlib.services.bitcoind.BitcoindClient* static method), 112
- `from_config()` (*bitcoinlib.services.dashd.DashdClient* static method), 117
- `from_config()` (*bitcoinlib.services.dogecoin.DogecoinClient* static method), 118
- `from_config()` (*bitcoinlib.services.litecoind.LitecoindClient* static method), 120
- `from_ints()` (*bitcoinlib.scripts.Stack* class method), 63
- `from_key()` (*bitcoinlib.wallets.WalletKey* static method), 86
- `from_passphrase()` (*bitcoinlib.keys.HDKey* static method), 35
- `from_raw()` (*bitcoinlib.blocks.Block* class method), 97
- `from_satoshi()` (*bitcoinlib.values.Value* class method), 101
- `from_seed()` (*bitcoinlib.keys.HDKey* static method), 35
- `from_str()` (*bitcoinlib.keys.Signature* method), 42
- `from_transaction()` (*bitcoinlib.wallets.WalletTransaction* class method), 88
- `from_txid()` (*bitcoinlib.wallets.WalletTransaction* class method), 88
- ## G
- `generate()` (*bitcoinlib.mnemonic.Mnemonic* method), 91
- `get_data_type()` (in module *bitcoinlib.scripts*), 66
- `get_encoding_from_witness()` (in module *bitcoinlib.main*), 141
- `get_key()` (*bitcoinlib.wallets.Wallet* method), 70
- `get_key_change()` (*bitcoinlib.wallets.Wallet* method), 70
- `get_key_format()` (in module *bitcoinlib.keys*), 44
- `get_keys()` (*bitcoinlib.wallets.Wallet* method), 70
- `get_keys_change()` (*bitcoinlib.wallets.Wallet* method), 70
- `get_unlocking_script_type()` (in module *bitcoinlib.transactions*), 58
- `getaddress()` (*bitcoinlib.services.services.Cache* method), 104
- `getbalance()` (*bitcoinlib.services.bcoin.BcoinClient* method), 111
- `getbalance()` (*bitcoinlib.services.bitaps.BitapsClient* method), 111
- `getbalance()` (*bitcoinlib.services.bitcoinlibtest.BitcoinLibTestClient* method), 113
- `getbalance()` (*bitcoinlib.services.bitgo.BitGoClient* method), 113
- `getbalance()` (*bitcoinlib.services.blockchaininfo.BlockchainInfoClient* method), 114
- `getbalance()` (*bitcoinlib.services.blockchair.BlockChairClient* method), 114
- `getbalance()` (*bitcoinlib.services.blockcypher.BlockCypher* method), 115

`getbalance()` (*bitcoinlib.services.blocksmurfer.BlocksmurferClient method*), 115  
`getbalance()` (*bitcoinlib.services.blockstream.BlockstreamClient method*), 116  
`getbalance()` (*bitcoinlib.services.chainso.ChainSo method*), 116  
`getbalance()` (*bitcoinlib.services.cryptoid.CryptoID method*), 117  
`getbalance()` (*bitcoinlib.services.insightdash.InsightDashClient method*), 119  
`getbalance()` (*bitcoinlib.services.litecoinblockexplorer.LitecoinBlockexplorerClient method*), 119  
`getbalance()` (*bitcoinlib.services.litecoreio.LitecoreIOClient method*), 121  
`getbalance()` (*bitcoinlib.services.services.Service method*), 107  
`getbalance()` (*bitcoinlib.services.smartbit.SmartbitClient method*), 121  
`getblock()` (*bitcoinlib.services.bcoin.BcoinClient method*), 111  
`getblock()` (*bitcoinlib.services.bitcoind.BitcoindClient method*), 112  
`getblock()` (*bitcoinlib.services.blockchaininfo.BlockchainInfoClient method*), 114  
`getblock()` (*bitcoinlib.services.blockchair.BlockChairClient method*), 114  
`getblock()` (*bitcoinlib.services.blockcypher.BlockCypherClient method*), 115  
`getblock()` (*bitcoinlib.services.blocksmurfer.BlocksmurferClient method*), 115  
`getblock()` (*bitcoinlib.services.blockstream.BlockstreamClient method*), 116  
`getblock()` (*bitcoinlib.services.chainso.ChainSo method*), 116  
`getblock()` (*bitcoinlib.services.dashd.DashdClient method*), 118  
`getblock()` (*bitcoinlib.services.insightdash.InsightDashClient method*), 119  
`getblock()` (*bitcoinlib.services.litecoinblockexplorer.LitecoinBlockexplorerClient method*), 119  
`getblock()` (*bitcoinlib.services.litecoind.LitecoindClient method*), 120  
`getblock()` (*bitcoinlib.services.litecoreio.LitecoreIOClient method*), 121  
`getblock()` (*bitcoinlib.services.services.Cache method*), 104  
`getblock()` (*bitcoinlib.services.services.Service method*), 107  
`getblock()` (*bitcoinlib.services.smartbit.SmartbitClient method*), 121  
`getblocktransactions()` (*bitcoinlib.services.services.Cache method*), 104  
`getcacheaddressinfo()` (*bitcoinlib.services.services.Service method*), 108  
`getinfo()` (*bitcoinlib.services.bcoin.BcoinClient method*), 111  
`getinfo()` (*bitcoinlib.services.bitcoind.BitcoindClient method*), 112  
`getinfo()` (*bitcoinlib.services.blockchaininfo.BlockchainInfoClient method*), 114  
`getinfo()` (*bitcoinlib.services.blockchair.BlockChairClient method*), 114  
`getinfo()` (*bitcoinlib.services.blocksmurfer.BlocksmurferClient method*), 115  
`getinfo()` (*bitcoinlib.services.chainso.ChainSo method*), 116  
`getinfo()` (*bitcoinlib.services.dashd.DashdClient method*), 118  
`getinfo()` (*bitcoinlib.services.dogecoin.DogecoinClient method*), 119  
`getinfo()` (*bitcoinlib.services.insightdash.InsightDashClient method*), 119  
`getinfo()` (*bitcoinlib.services.litecoinblockexplorer.LitecoinBlockexplorerClient method*), 119  
`getinfo()` (*bitcoinlib.services.litecoind.LitecoindClient method*), 120  
`getinfo()` (*bitcoinlib.services.litecoreio.LitecoreIOClient method*), 121  
`getinfo()` (*bitcoinlib.services.services.Service method*), 108  
`getinputvalues()` (*bitcoinlib.services.services.Service method*), 108  
`getrawblock()` (*bitcoinlib.services.bitcoind.BitcoindClient method*), 112  
`getrawblock()` (*bitcoinlib.services.blockchaininfo.BlockchainInfoClient method*), 114  
`getrawblock()` (*bitcoinlib.services.blockchair.BlockChairClient method*), 114  
`getrawblock()` (*bitcoinlib.services.blockstream.BlockstreamClient method*), 116  
`getrawblock()` (*bitcoinlib.services.dashd.DashdClient method*), 118  
`getrawblock()` (*bitcoinlib.services.litecoind.LitecoindClient method*), 120  
`getrawblock()` (*bitcoinlib.services.litecoreio.LitecoreIOClient method*), 121  
`getrawblock()` (*bitcoinlib.services.services.Service method*), 108  
`getrawtransaction()` (*bitcoinlib.services.services.Service method*), 107

<i>lib.services.bcoin.BcoinClient</i>	<i>method</i> ),	<i>lib.services.bcoin.BcoinClient</i>	<i>method</i> ),
111		111	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.bitaps.BitapsClient</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.bitcoind.BitcoindClient</i>
<i>method</i> ),		<i>method</i> ),	
111		112	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.bitcoind.BitcoindClient</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.blockchaininfo.BlockchainInfoClient</i>
<i>method</i> ),		<i>method</i> ),	
112		114	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.blockchaininfo.BlockchainInfoClient</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.blockchair.BlockChairClient</i>
<i>method</i> ),		<i>method</i> ),	
114		114	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.blockchair.BlockChairClient</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.blockcypher.BlockCypher</i>
<i>method</i> ),		<i>method</i> ),	
114		115	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.blockcypher.BlockCypher</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.blocksmurfer.BlocksMurferClient</i>
<i>method</i> ),		<i>method</i> ),	
115		115	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.blocksmurfer.BlocksMurferClient</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.blockstream.BlockstreamClient</i>
<i>method</i> ),		<i>method</i> ),	
115		116	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.blockstream.BlockstreamClient</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.chainso.ChainSo</i>
<i>method</i> ),		<i>method</i> ),	
116		116	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.chainso.ChainSo</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.cryptoid.CryptoID</i>
<i>method</i> ),		<i>method</i> ),	
116		117	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.cryptoid.CryptoID</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.dashd.DashdClient</i>
<i>method</i> ),		<i>method</i> ),	
117		118	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.dashd.DashdClient</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.dogecoin.DogecoinClient</i>
<i>method</i> ),		<i>method</i> ),	
118		119	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.dogecoin.DogecoinClient</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.insightdash.InsightDashClient</i>
<i>method</i> ),		<i>method</i> ),	
119		119	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.insightdash.InsightDashClient</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.litecoinblockexplorer.LitecoinBlockexplorerClient</i>
<i>method</i> ),		<i>method</i> ),	
119		120	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.litecoinblockexplorer.LitecoinBlockexplorerClient</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.litecoind.LitecoindClient</i>
<i>method</i> ),		<i>method</i> ),	
120		120	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.litecoind.LitecoindClient</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.litecoreio.LitecoreIOClient</i>
<i>method</i> ),		<i>method</i> ),	
120		121	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.litecoreio.LitecoreIOClient</i>	<i>gettransaction()</i>	<i>(bitcoinlib.services.services.Cache</i>
<i>method</i> ),		<i>method</i> ),	
121		105	
<i>getrawtransaction()</i>	<i>(bitcoinlib.services.services.Cache</i>	<i>gettransaction()</i>	<i>(bitcoinlib.services.services.Service</i>
<i>method</i> ),		<i>method</i> ),	
105		108	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.services.Service</i>	<i>gettransaction()</i>	<i>(bitcoin-lib.services.smartbit.SmartbitClient</i>
<i>method</i> ),		<i>method</i> ),	
108		121	
<i>getrawtransaction()</i>	<i>(bitcoin-lib.services.smartbit.SmartbitClient</i>	<i>gettransactions()</i>	<i>(bitcoin-lib.services.bcoin.BcoinClient</i>
<i>method</i> ),		<i>method</i> ),	
121		111	
<i>gettransaction()</i>	<i>(bitcoin-</i>	<i>gettransactions()</i>	<i>(bitcoin-</i>

*lib.services.blockchaininfo.BlockchainInfoClient* method), 114

*gettransactions()* (*bitcoinlib.services.blockchair.BlockChairClient* method), 114

*gettransactions()* (*bitcoinlib.services.blockcypher.BlockCypher* method), 115

*gettransactions()* (*bitcoinlib.services.blocksmurfer.BlocksmurferClient* method), 115

*gettransactions()* (*bitcoinlib.services.blockstream.BlockstreamClient* method), 116

*gettransactions()* (*bitcoinlib.services.chainso.ChainSo* method), 116

*gettransactions()* (*bitcoinlib.services.cryptoid.CryptoID* method), 117

*gettransactions()* (*bitcoinlib.services.insightdash.InsightDashClient* method), 119

*gettransactions()* (*bitcoinlib.services.litecoreio.LitecoreIOClient* method), 121

*gettransactions()* (*bitcoinlib.services.services.Cache* method), 105

*gettransactions()* (*bitcoinlib.services.services.Service* method), 109

*gettransactions()* (*bitcoinlib.services.smartbit.SmartbitClient* method), 121

*getutxos()* (*bitcoinlib.services.bcoin.BcoinClient* method), 111

*getutxos()* (*bitcoinlib.services.bitaps.BitapsClient* method), 111

*getutxos()* (*bitcoinlib.services.bitcoinlibtest.BitcoinLibTestClient* method), 113

*getutxos()* (*bitcoinlib.services.bitgo.BitGoClient* method), 113

*getutxos()* (*bitcoinlib.services.blockchaininfo.BlockchainInfoClient* method), 114

*getutxos()* (*bitcoinlib.services.blockchair.BlockChairClient* method), 114

*getutxos()* (*bitcoinlib.services.blockcypher.BlockCypher* method), 115

*getutxos()* (*bitcoinlib.services.blocksmurfer.BlocksmurferClient* method), 115

*getutxos()* (*bitcoinlib.services.blockstream.BlockstreamClient* method), 116

*getutxos()* (*bitcoinlib.services.chainso.ChainSo* method), 116

*getutxos()* (*bitcoinlib.services.cryptoid.CryptoID* method), 117

*getutxos()* (*bitcoinlib.services.dashd.DashdClient* method), 118

*getutxos()* (*bitcoinlib.services.dogecoin.DogecoinClient* method), 119

*getutxos()* (*bitcoinlib.services.insightdash.InsightDashClient* method), 119

*getutxos()* (*bitcoinlib.services.litecoinblockexplorer.LitecoinBlockexplorerClient* method), 120

*getutxos()* (*bitcoinlib.services.litecoind.LitecoindClient* method), 121

*getutxos()* (*bitcoinlib.services.litecoreio.LitecoreIOClient* method), 121

*getutxos()* (*bitcoinlib.services.services.Cache* method), 105

*getutxos()* (*bitcoinlib.services.services.Service* method), 109

*getutxos()* (*bitcoinlib.services.smartbit.SmartbitClient* method), 121

## H

*hash160* (*bitcoinlib.keys.Key* property), 40

*hash160()* (in module *bitcoinlib.encoding*), 138

*hashed\_data* (*bitcoinlib.keys.Address* property), 30

*HDKey* (class in *bitcoinlib.keys*), 32

*height* (*bitcoinlib.db\_cache.DbCacheBlock* attribute), 130

*hex()* (*bitcoinlib.keys.Key* method), 40

*hex()* (*bitcoinlib.keys.Signature* method), 42

## I

*id* (*bitcoinlib.db.DbKey* attribute), 123

*id* (*bitcoinlib.db.DbTransaction* attribute), 125

*id* (*bitcoinlib.db.DbWallet* attribute), 128

*import\_address()* (*bitcoinlib.keys.Address* class method), 31

*import\_key()* (*bitcoinlib.wallets.Wallet* method), 71

*import\_master\_key()* (*bitcoinlib.wallets.Wallet* method), 71

*import\_raw()* (*bitcoinlib.transactions.Transaction* static method), 53

*index\_n* (*bitcoinlib.db.DbTransactionInput* attribute), 126

*index\_n* (*bitcoinlib.db\_cache.DbCacheTransactionNode* attribute), 131

*info()* (*bitcoinlib.keys.HDKey* method), 35

*info()* (*bitcoinlib.keys.Key* method), 40

*info()* (*bitcoinlib.transactions.Transaction* method), 54

*info()* (*bitcoinlib.wallets.Wallet* method), 71

*info()* (*bitcoinlib.wallets.WalletTransaction* method), 88

*initialize\_lib()* (in module *bitcoinlib.config.config*), 122

*Input* (class in *bitcoinlib.transactions*), 47

*input\_total* (*bitcoinlib.db.DbTransaction* attribute), 125

inputs (*bitcoinlib.db.DbTransaction attribute*), 125  
 InsightDashClient (class in *bitcoinlib.services.insightdash*), 119  
 int\_to\_varbyteint() (in module *bitcoinlib.encoding*), 138  
 is\_arithmetic() (*bitcoinlib.scripts.Stack method*), 63  
 is\_complete (*bitcoinlib.db.DbTransaction attribute*), 125  
 is\_input (*bitcoinlib.db\_cache.DbCacheTransactionNode attribute*), 131  
 is\_private (*bitcoinlib.db.DbKey attribute*), 123  
 isspent() (*bitcoinlib.services.bcoin.BcoinClient method*), 111  
 isspent() (*bitcoinlib.services.bitcoind.BitcoindClient method*), 112  
 isspent() (*bitcoinlib.services.blockchair.BlockChairClient method*), 114  
 isspent() (*bitcoinlib.services.blockcypher.BlockCypher method*), 115  
 isspent() (*bitcoinlib.services.blocksmurfer.BlocksMurferClient method*), 115  
 isspent() (*bitcoinlib.services.blockstream.BlockstreamClient method*), 116  
 isspent() (*bitcoinlib.services.dashd.DashdClient method*), 118  
 isspent() (*bitcoinlib.services.insightdash.InsightDashClient method*), 119  
 isspent() (*bitcoinlib.services.litecoind.LitecoindClient method*), 121  
 isspent() (*bitcoinlib.services.litecoreio.LitecoreIOClient method*), 121  
 isspent() (*bitcoinlib.services.services.Service method*), 109  
 isspent() (*bitcoinlib.services.smartbit.SmartbitClient method*), 121

## J

JSONRPCException, 110

## K

key (*bitcoinlib.db.DbTransactionInput attribute*), 126  
 key (*bitcoinlib.db.DbTransactionOutput attribute*), 127  
 Key (class in *bitcoinlib.keys*), 38  
 key() (*bitcoinlib.wallets.Wallet method*), 71  
 key() (*bitcoinlib.wallets.WalletKey method*), 87  
 key\_for\_path() (*bitcoinlib.wallets.Wallet method*), 71  
 key\_id (*bitcoinlib.db.DbTransactionInput attribute*), 126  
 key\_id (*bitcoinlib.db.DbTransactionOutput attribute*), 127  
 key\_order (*bitcoinlib.db.DbKeyMultisigChildren attribute*), 124  
 key\_path (*bitcoinlib.db.DbWallet attribute*), 128  
 key\_type (*bitcoinlib.db.DbKey attribute*), 123  
 keys (*bitcoinlib.db.DbWallet attribute*), 128

keys() (*bitcoinlib.wallets.Wallet method*), 72  
 keys\_accounts() (*bitcoinlib.wallets.Wallet method*), 73  
 keys\_address\_change() (*bitcoinlib.wallets.Wallet method*), 73  
 keys\_address\_payment() (*bitcoinlib.wallets.Wallet method*), 73  
 keys\_addresses() (*bitcoinlib.wallets.Wallet method*), 74  
 keys\_networks() (*bitcoinlib.wallets.Wallet method*), 74

## L

last\_block (*bitcoinlib.db\_cache.DbCacheAddress attribute*), 130  
 last\_txid (*bitcoinlib.db\_cache.DbCacheAddress attribute*), 130  
 latest\_txid (*bitcoinlib.db.DbKey attribute*), 123  
 legacy (*bitcoinlib.db\_cache.WitnessTypeTransactions attribute*), 133  
 LitecoinBlockexplorerClient (class in *bitcoinlib.services.litecoinblockexplorer*), 119  
 LitecoindClient (class in *bitcoinlib.services.litecoind*), 120  
 LitecoreIOClient (class in *bitcoinlib.services.litecoreio*), 121  
 load() (*bitcoinlib.transactions.Transaction static method*), 54  
 locktime (*bitcoinlib.db.DbTransaction attribute*), 125  
 locktime (*bitcoinlib.db\_cache.DbCacheTransaction attribute*), 131

## M

main\_key\_id (*bitcoinlib.db.DbWallet attribute*), 128  
 mempool() (*bitcoinlib.services.bcoin.BcoinClient method*), 111  
 mempool() (*bitcoinlib.services.bitcoind.BitcoindClient method*), 112  
 mempool() (*bitcoinlib.services.bitcoinlibtest.BitcoinLibTestClient method*), 113  
 mempool() (*bitcoinlib.services.blockchaininfo.BlockchainInfoClient method*), 114  
 mempool() (*bitcoinlib.services.blockchair.BlockChairClient method*), 114  
 mempool() (*bitcoinlib.services.blockcypher.BlockCypher method*), 115  
 mempool() (*bitcoinlib.services.blocksmurfer.BlocksMurferClient method*), 115  
 mempool() (*bitcoinlib.services.blockstream.BlockstreamClient method*), 116  
 mempool() (*bitcoinlib.services.chainso.ChainSo method*), 116  
 mempool() (*bitcoinlib.services.cryptoid.CryptoID method*), 117  
 mempool() (*bitcoinlib.services.dogecoin.DogecoinClient method*), 119



`mempool()` (*bitcoinlib.services.insightdash.InsightDashClient* method), 119  
`mempool()` (*bitcoinlib.services.litecoinblockexplorer.LitecoinBlockExplorerClient* method), 120  
`mempool()` (*bitcoinlib.services.litecoind.LitecoindClient* method), 121  
`mempool()` (*bitcoinlib.services.litecoreio.LitecoreIOClient* method), 121  
`mempool()` (*bitcoinlib.services.services.Service* method), 109  
`mempool()` (*bitcoinlib.services.smartbit.SmartbitClient* method), 121  
`merge_transaction()` (*bitcoinlib.transactions.Transaction* method), 54  
`merkle_root` (*bitcoinlib.db\_cache.DbCacheBlock* attribute), 130  
`Mnemonic` (class in *bitcoinlib.mnemonic*), 91  
`mod_sqrt()` (in module *bitcoinlib.keys*), 45  
module  
    *bitcoinlib*, 141  
    *bitcoinlib.blocks*, 95  
    *bitcoinlib.config*, 122  
    *bitcoinlib.config.config*, 122  
    *bitcoinlib.config.opcodes*, 122  
    *bitcoinlib.config.secp256k1*, 122  
    *bitcoinlib.db*, 122  
    *bitcoinlib.db\_cache*, 129  
    *bitcoinlib.encoding*, 135  
    *bitcoinlib.keys*, 30  
    *bitcoinlib.main*, 141  
    *bitcoinlib.mnemonic*, 91  
    *bitcoinlib.networks*, 93  
    *bitcoinlib.scripts*, 60  
    *bitcoinlib.services*, 122  
    *bitcoinlib.services.authproxy*, 110  
    *bitcoinlib.services.baseclient*, 111  
    *bitcoinlib.services.bcoin*, 111  
    *bitcoinlib.services.bitaps*, 111  
    *bitcoinlib.services.bitcoind*, 112  
    *bitcoinlib.services.bitcoinlibtest*, 113  
    *bitcoinlib.services.bitgo*, 113  
    *bitcoinlib.services.blockchaininfo*, 114  
    *bitcoinlib.services.blockchair*, 114  
    *bitcoinlib.services.blockcypher*, 115  
    *bitcoinlib.services.blocksmurfer*, 115  
    *bitcoinlib.services.blockstream*, 116  
    *bitcoinlib.services.chainso*, 116  
    *bitcoinlib.services.coinfees*, 117  
    *bitcoinlib.services.cryptoid*, 117  
    *bitcoinlib.services.dashd*, 117  
    *bitcoinlib.services.dogecoin*, 118  
    *bitcoinlib.services.insightdash*, 119  
    *bitcoinlib.services.litecoinblockexplorer*, 119  
    *bitcoinlib.services.litecoind*, 120  
    *bitcoinlib.services.litecoreio*, 121  
    *bitcoinlib.services.services*, 104  
    *bitcoinlib.services.smartbit*, 121  
    *bitcoinlib.tools*, 135  
    *bitcoinlib.tools.clw*, 135  
    *bitcoinlib.tools.mnemonic\_key\_create*, 135  
    *bitcoinlib.tools.sign\_raw*, 135  
    *bitcoinlib.tools.wallet\_multisig\_2of3*, 135  
    *bitcoinlib.transactions*, 47  
    *bitcoinlib.values*, 100  
    *bitcoinlib.wallets*, 66  
`multisig` (*bitcoinlib.db.DbWallet* attribute), 128  
`multisig_children` (*bitcoinlib.db.DbKey* attribute), 123  
`multisig_n_required` (*bitcoinlib.db.DbWallet* attribute), 128  
`multisig_parents` (*bitcoinlib.db.DbKey* attribute), 123  

## N

`n_txs` (*bitcoinlib.db\_cache.DbCacheAddress* attribute), 130  
`n_utxos` (*bitcoinlib.db\_cache.DbCacheAddress* attribute), 130  
`name` (*bitcoinlib.db.DbKey* attribute), 123  
`name` (*bitcoinlib.db.DbNetwork* attribute), 124  
`name` (*bitcoinlib.db.DbWallet* attribute), 128  
`name` (*bitcoinlib.wallets.Wallet* property), 74  
`name` (*bitcoinlib.wallets.WalletKey* property), 87  
`network` (*bitcoinlib.db.DbKey* attribute), 123  
`network` (*bitcoinlib.db.DbTransaction* attribute), 125  
`network` (*bitcoinlib.db.DbWallet* attribute), 128  
`Network` (class in *bitcoinlib.networks*), 93  
`network_by_value()` (in module *bitcoinlib.networks*), 93  
`network_change()` (*bitcoinlib.keys.HDKey* method), 36  
`network_defined()` (in module *bitcoinlib.networks*), 94  
`network_list()` (*bitcoinlib.wallets.Wallet* method), 74  
`network_name` (*bitcoinlib.db.DbKey* attribute), 123  
`network_name` (*bitcoinlib.db.DbTransaction* attribute), 125  
`network_name` (*bitcoinlib.db.DbWallet* attribute), 128  
`network_name` (*bitcoinlib.db\_cache.DbCacheAddress* attribute), 130  
`network_name` (*bitcoinlib.db\_cache.DbCacheBlock* attribute), 130  
`network_name` (*bitcoinlib.db\_cache.DbCacheTransaction* attribute), 131  
`network_name` (*bitcoinlib.db\_cache.DbCacheVars* attribute), 132  
`network_values_for()` (in module *bitcoinlib.networks*), 94

[NetworkError](#), 93  
[networks\(\)](#) (*bitcoinlib.wallets.Wallet method*), 75  
[new\\_account\(\)](#) (*bitcoinlib.wallets.Wallet method*), 75  
[new\\_key\(\)](#) (*bitcoinlib.wallets.Wallet method*), 75  
[new\\_key\\_change\(\)](#) (*bitcoinlib.wallets.Wallet method*), 75  
[nodes](#) (*bitcoinlib.db\_cache.DbCacheTransaction attribute*), 131  
[nonce](#) (*bitcoinlib.db\_cache.DbCacheBlock attribute*), 130  
[normalize\\_path\(\)](#) (*in module bitcoinlib.wallets*), 89  
[normalize\\_string\(\)](#) (*in module bitcoinlib.encoding*), 139  
[normalize\\_var\(\)](#) (*in module bitcoinlib.encoding*), 139

## O

[op\(\)](#) (*in module bitcoinlib.config.opcodes*), 122  
[op\\_0notequal\(\)](#) (*bitcoinlib.scripts.Stack method*), 63  
[op\\_1add\(\)](#) (*bitcoinlib.scripts.Stack method*), 63  
[op\\_1sub\(\)](#) (*bitcoinlib.scripts.Stack method*), 63  
[op\\_2drop\(\)](#) (*bitcoinlib.scripts.Stack method*), 63  
[op\\_2dup\(\)](#) (*bitcoinlib.scripts.Stack method*), 63  
[op\\_2over\(\)](#) (*bitcoinlib.scripts.Stack method*), 63  
[op\\_2rot\(\)](#) (*bitcoinlib.scripts.Stack method*), 63  
[op\\_2swap\(\)](#) (*bitcoinlib.scripts.Stack method*), 63  
[op\\_3dup\(\)](#) (*bitcoinlib.scripts.Stack method*), 63  
[op\\_abs\(\)](#) (*bitcoinlib.scripts.Stack method*), 63  
[op\\_add\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_booland\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_booror\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_checkclocktimeverify\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_checkmultisig\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_checkmultisigverify\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_checksequenceverify\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_checksigsig\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_checksigsigverify\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_depth\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_drop\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_dup\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_equal\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_equalverify\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_hash160\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_hash256\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_if\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_ifdup\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_max\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_min\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_negate\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_nip\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_nop\(\)](#) (*bitcoinlib.scripts.Stack method*), 64  
[op\\_nop1\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_nop10\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_nop4\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_nop5\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_nop6\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_nop7\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_nop8\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_nop9\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_not\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_notif\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_numequal\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_numequalverify\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_numgreaterthan\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_numgreaterthanorequal\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_numless than\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_numless thanorequal\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_numnotequal\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_over\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_pick\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_return\(\)](#) (*bitcoinlib.scripts.Stack static method*), 65  
[op\\_ripemd160\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_roll\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_rot\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_sha1\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_sha256\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_size\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_sub\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_swap\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_tuck\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_verify\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[op\\_within\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[order\\_n](#) (*bitcoinlib.db\_cache.DbCacheTransaction attribute*), 131  
[Output](#) (*class in bitcoinlib.transactions*), 48  
[output\\_n](#) (*bitcoinlib.db.DbTransactionInput attribute*), 126  
[output\\_n](#) (*bitcoinlib.db.DbTransactionOutput attribute*), 127  
[output\\_n\(\)](#) (*bitcoinlib.db\_cache.DbCacheTransactionNode method*), 132  
[output\\_total](#) (*bitcoinlib.db.DbTransaction attribute*), 125  
[outputs](#) (*bitcoinlib.db.DbTransaction attribute*), 125  
[owner](#) (*bitcoinlib.db.DbWallet attribute*), 128  
[owner](#) (*bitcoinlib.wallets.Wallet property*), 76

## P

[parent\\_id](#) (*bitcoinlib.db.DbKey attribute*), 123

[parent\\_id](#) (*bitcoinlib.db.DbKeyMultisigChildren attribute*), 124  
[parent\\_id](#) (*bitcoinlib.db.DbWallet attribute*), 129  
[parse\(\)](#) (*bitcoinlib.blocks.Block class method*), 97  
[parse\(\)](#) (*bitcoinlib.keys.Address class method*), 31  
[parse\(\)](#) (*bitcoinlib.keys.Signature class method*), 42  
[parse\(\)](#) (*bitcoinlib.scripts.Script class method*), 61  
[parse\(\)](#) (*bitcoinlib.transactions.Input class method*), 48  
[parse\(\)](#) (*bitcoinlib.transactions.Output class method*), 49  
[parse\(\)](#) (*bitcoinlib.transactions.Transaction class method*), 54  
[parse\\_bytes\(\)](#) (*bitcoinlib.blocks.Block class method*), 98  
[parse\\_bytes\(\)](#) (*bitcoinlib.keys.Signature static method*), 42  
[parse\\_bytes\(\)](#) (*bitcoinlib.scripts.Script class method*), 61  
[parse\\_bytes\(\)](#) (*bitcoinlib.transactions.Transaction class method*), 54  
[parse\\_bytesio\(\)](#) (*bitcoinlib.blocks.Block class method*), 98  
[parse\\_bytesio\(\)](#) (*bitcoinlib.scripts.Script class method*), 62  
[parse\\_bytesio\(\)](#) (*bitcoinlib.transactions.Transaction class method*), 54  
[parse\\_hex\(\)](#) (*bitcoinlib.keys.Signature class method*), 42  
[parse\\_hex\(\)](#) (*bitcoinlib.scripts.Script class method*), 62  
[parse\\_hex\(\)](#) (*bitcoinlib.transactions.Transaction class method*), 55  
[parse\\_transactions\(\)](#) (*bitcoinlib.blocks.Block method*), 99  
[path](#) (*bitcoinlib.db.DbKey attribute*), 123  
[path\\_expand\(\)](#) (*bitcoinlib.wallets.Wallet method*), 76  
[path\\_expand\(\)](#) (*in module bitcoinlib.keys*), 45  
[pop\\_as\\_number\(\)](#) (*bitcoinlib.scripts.Stack method*), 65  
[prev\\_block](#) (*bitcoinlib.db\_cache.DbCacheBlock attribute*), 130  
[prev\\_txid](#) (*bitcoinlib.db.DbTransactionInput attribute*), 126  
[prev\\_txid\(\)](#) (*bitcoinlib.db\_cache.DbCacheTransactionNode method*), 132  
[print\\_value\(\)](#) (*bitcoinlib.networks.Network method*), 93  
[print\\_value\(\)](#) (*in module bitcoinlib.networks*), 94  
[private](#) (*bitcoinlib.db.DbKey attribute*), 123  
[pubkeyhash\\_to\\_addr\(\)](#) (*in module bitcoinlib.encoding*), 139  
[pubkeyhash\\_to\\_addr\\_base58\(\)](#) (*in module bitcoinlib.encoding*), 139  
[pubkeyhash\\_to\\_addr\\_bech32\(\)](#) (*in module bitcoinlib.encoding*), 139

[public](#) (*bitcoinlib.db.DbKey attribute*), 124  
[public\(\)](#) (*bitcoinlib.keys.HDKey method*), 36  
[public\(\)](#) (*bitcoinlib.keys.Key method*), 40  
[public\(\)](#) (*bitcoinlib.wallets.WalletKey method*), 87  
[public\\_key](#) (*bitcoinlib.keys.Signature property*), 42  
[public\\_master\(\)](#) (*bitcoinlib.keys.HDKey method*), 36  
[public\\_master\(\)](#) (*bitcoinlib.wallets.Wallet method*), 76  
[public\\_master\\_multisig\(\)](#) (*bitcoinlib.keys.HDKey method*), 36  
[public\\_point\(\)](#) (*bitcoinlib.keys.Key method*), 40  
[purpose](#) (*bitcoinlib.db.DbKey attribute*), 124  
[purpose](#) (*bitcoinlib.db.DbWallet attribute*), 129

## Q

[Quantity](#) (*class in bitcoinlib.encoding*), 135

## R

[raw](#) (*bitcoinlib.db.DbTransaction attribute*), 125  
[raw](#) (*bitcoinlib.scripts.Script property*), 62  
[raw\(\)](#) (*bitcoinlib.transactions.Transaction method*), 55  
[raw\\_hex\(\)](#) (*bitcoinlib.transactions.Transaction method*), 55  
[read\\_config\(\)](#) (*in module bitcoinlib.config.config*), 122  
[read\\_varbyteint\(\)](#) (*in module bitcoinlib.encoding*), 140  
[ref\\_index\\_n](#) (*bitcoinlib.db\_cache.DbCacheTransactionNode attribute*), 132  
[ref\\_txid](#) (*bitcoinlib.db\_cache.DbCacheTransactionNode attribute*), 132  
[request\(\)](#) (*bitcoinlib.services.baseclient.BaseClient method*), 111

## S

[sanitize\\_mnemonic\(\)](#) (*bitcoinlib.mnemonic.Mnemonic method*), 91  
[save\(\)](#) (*bitcoinlib.transactions.Transaction method*), 55  
[save\(\)](#) (*bitcoinlib.wallets.WalletTransaction method*), 88  
[scan\(\)](#) (*bitcoinlib.wallets.Wallet method*), 77  
[scan\\_key\(\)](#) (*bitcoinlib.wallets.Wallet method*), 77  
[scheme](#) (*bitcoinlib.db.DbWallet attribute*), 129  
[script](#) (*bitcoinlib.db.DbTransactionInput attribute*), 126  
[script](#) (*bitcoinlib.db.DbTransactionOutput attribute*), 127  
[script](#) (*bitcoinlib.db\_cache.DbCacheTransactionNode attribute*), 132  
[Script](#) (*class in bitcoinlib.scripts*), 60  
[script\\_add\\_locktime\\_cltv\(\)](#) (*in module bitcoinlib.transactions*), 58  
[script\\_add\\_locktime\\_csv\(\)](#) (*in module bitcoinlib.transactions*), 58  
[script\\_deserialize\(\)](#) (*in module bitcoinlib.transactions*), 58



`script_to_string()` (in module `bitcoinlib.transactions`), 58  
`script_type` (`bitcoinlib.db.DbTransactionInput` attribute), 126  
`script_type` (`bitcoinlib.db.DbTransactionOutput` attribute), 127  
`script_type_default()` (in module `bitcoinlib.main`), 141  
`ScriptError`, 63  
`segwit` (`bitcoinlib.db_cache.WitnessTypeTransactions` attribute), 133  
`select_inputs()` (`bitcoinlib.wallets.Wallet` method), 77  
`send()` (`bitcoinlib.wallets.Wallet` method), 78  
`send()` (`bitcoinlib.wallets.WalletTransaction` method), 88  
`send_to()` (`bitcoinlib.wallets.Wallet` method), 79  
`sendrawtransaction()` (`bitcoinlib.services.bcoin.BcoinClient` method), 111  
`sendrawtransaction()` (`bitcoinlib.services.bitcoind.BitcoindClient` method), 112  
`sendrawtransaction()` (`bitcoinlib.services.bitcoinlibtest.BitcoinLibTestClient` method), 113  
`sendrawtransaction()` (`bitcoinlib.services.blockchair.BlockChairClient` method), 114  
`sendrawtransaction()` (`bitcoinlib.services.blockcypher.BlockCypher` method), 115  
`sendrawtransaction()` (`bitcoinlib.services.blocksmurfer.BlocksMurferClient` method), 115  
`sendrawtransaction()` (`bitcoinlib.services.blockstream.BlockstreamClient` method), 116  
`sendrawtransaction()` (`bitcoinlib.services.chainso.ChainSo` method), 116  
`sendrawtransaction()` (`bitcoinlib.services.dashd.DashdClient` method), 118  
`sendrawtransaction()` (`bitcoinlib.services.dogecoin.DogecoinClient` method), 119  
`sendrawtransaction()` (`bitcoinlib.services.insightdash.InsightDashClient` method), 119  
`sendrawtransaction()` (`bitcoinlib.services.litecoinblockexplorer.LitecoinBlockexplorerClient` method), 120  
`sendrawtransaction()` (`bitcoinlib.services.litecoind.LitecoindClient` method), 121  
`sendrawtransaction()` (`bitcoinlib.services.litecoreio.LitecoreIOClient` method), 121  
`sendrawtransaction()` (`bitcoinlib.services.services.Service` method), 109  
`sendrawtransaction()` (`bitcoinlib.services.smartbit.SmartbitClient` method), 121  
`sequence` (`bitcoinlib.db.DbTransactionInput` attribute), 127  
`sequence` (`bitcoinlib.db_cache.DbCacheTransactionNode` attribute), 132  
`serialize()` (`bitcoinlib.blocks.Block` method), 99  
`serialize()` (`bitcoinlib.scripts.Script` method), 62  
`serialize_list()` (`bitcoinlib.scripts.Script` method), 62  
`serialize_multisig_redeemscript()` (in module `bitcoinlib.transactions`), 59  
`Service` (class in `bitcoinlib.services.services`), 106  
`ServiceError`, 109  
`set_locktime_blocks()` (`bitcoinlib.transactions.Transaction` method), 55  
`set_locktime_relative()` (`bitcoinlib.transactions.Output` method), 49  
`set_locktime_relative_blocks()` (`bitcoinlib.transactions.Output` method), 49  
`set_locktime_relative_blocks()` (`bitcoinlib.transactions.Transaction` method), 55  
`set_locktime_relative_time()` (`bitcoinlib.transactions.Output` method), 50  
`set_locktime_relative_time()` (`bitcoinlib.transactions.Transaction` method), 56  
`set_locktime_time()` (`bitcoinlib.transactions.Transaction` method), 56  
`shuffle()` (`bitcoinlib.transactions.Transaction` method), 56  
`shuffle_inputs()` (`bitcoinlib.transactions.Transaction` method), 56  
`shuffle_outputs()` (`bitcoinlib.transactions.Transaction` method), 56  
`sign()` (`bitcoinlib.transactions.Transaction` method), 56  
`sign()` (`bitcoinlib.wallets.WalletTransaction` method), 89  
`sign()` (in module `bitcoinlib.keys`), 45  
`sign_and_update()` (`bitcoinlib.transactions.Transaction` method), 57  
`Signature` (class in `bitcoinlib.keys`), 40  
`signature()` (`bitcoinlib.transactions.Transaction` method), 57  
`signature_hash()` (`bitcoinlib.transactions.Transaction` method), 57  
`signature_segwit()` (`bitcoinlib.transactions.Transaction` method), 57  
`size` (`bitcoinlib.db.DbTransaction` attribute), 125  
`SmartbitClient` (class in `bitcoinlib.services.smartbit`), 121

`sort_keys` (*bitcoinlib.db.DbWallet attribute*), 129  
`spending_index_n` (*bitcoinlib.db.DbTransactionOutput attribute*), 127  
`spending_index_n()` (*bitcoinlib.db\_cache.DbCacheTransactionNode method*), 132  
`spending_txid` (*bitcoinlib.db.DbTransactionOutput attribute*), 127  
`spending_txid()` (*bitcoinlib.db\_cache.DbCacheTransactionNode method*), 132  
`spent` (*bitcoinlib.db.DbTransactionOutput attribute*), 127  
`spent` (*bitcoinlib.db\_cache.DbCacheTransactionNode attribute*), 132  
`Stack` (*class in bitcoinlib.scripts*), 63  
`status` (*bitcoinlib.db.DbTransaction attribute*), 126  
`store()` (*bitcoinlib.wallets.WalletTransaction method*), 89  
`store_address()` (*bitcoinlib.services.services.Cache method*), 105  
`store_block()` (*bitcoinlib.services.services.Cache method*), 106  
`store_blockcount()` (*bitcoinlib.services.services.Cache method*), 106  
`store_estimated_fee()` (*bitcoinlib.services.services.Cache method*), 106  
`store_transaction()` (*bitcoinlib.services.services.Cache method*), 106  
`str()` (*bitcoinlib.values.Value method*), 101  
`str_auto()` (*bitcoinlib.values.Value method*), 102  
`str_unit()` (*bitcoinlib.values.Value method*), 102  
`subkey_for_path()` (*bitcoinlib.keys.HDKey method*), 36  
`sweep()` (*bitcoinlib.wallets.Wallet method*), 80

## T

`target` (*bitcoinlib.blocks.Block property*), 99  
`target_hex` (*bitcoinlib.blocks.Block property*), 99  
`time` (*bitcoinlib.db\_cache.DbCacheBlock attribute*), 130  
`to_bytes()` (*bitcoinlib.values.Value method*), 103  
`to_bytes()` (*in module bitcoinlib.encoding*), 140  
`to_entropy()` (*bitcoinlib.mnemonic.Mnemonic method*), 91  
`to_hex()` (*bitcoinlib.values.Value method*), 103  
`to_hexstring()` (*in module bitcoinlib.encoding*), 140  
`to_mnemonic()` (*bitcoinlib.mnemonic.Mnemonic method*), 92  
`to_seed()` (*bitcoinlib.mnemonic.Mnemonic method*), 92  
`transaction` (*bitcoinlib.db.DbTransactionInput attribute*), 127  
`transaction` (*bitcoinlib.db.DbTransactionOutput attribute*), 127

`transaction` (*bitcoinlib.db\_cache.DbCacheTransactionNode attribute*), 132  
`Transaction` (*class in bitcoinlib.transactions*), 50  
`transaction()` (*bitcoinlib.wallets.Wallet method*), 80  
`transaction_create()` (*bitcoinlib.wallets.Wallet method*), 80  
`transaction_deserialize()` (*in module bitcoinlib.transactions*), 59  
`transaction_id` (*bitcoinlib.db.DbTransactionInput attribute*), 127  
`transaction_id` (*bitcoinlib.db.DbTransactionOutput attribute*), 128  
`transaction_import()` (*bitcoinlib.wallets.Wallet method*), 81  
`transaction_import_raw()` (*bitcoinlib.wallets.Wallet method*), 82  
`transaction_inputs` (*bitcoinlib.db.DbKey attribute*), 124  
`transaction_last()` (*bitcoinlib.wallets.Wallet method*), 82  
`transaction_load()` (*bitcoinlib.wallets.Wallet method*), 82  
`transaction_outputs` (*bitcoinlib.db.DbKey attribute*), 124  
`transaction_spent()` (*bitcoinlib.wallets.Wallet method*), 82  
`transaction_update_spents()` (*in module bitcoinlib.transactions*), 59  
`TransactionError`, 58  
`transactions` (*bitcoinlib.db.DbWallet attribute*), 129  
`transactions()` (*bitcoinlib.wallets.Wallet method*), 82  
`transactions_export()` (*bitcoinlib.wallets.Wallet method*), 83  
`transactions_full()` (*bitcoinlib.wallets.Wallet method*), 83  
`transactions_update()` (*bitcoinlib.wallets.Wallet method*), 83  
`transactions_update_by_txids()` (*bitcoinlib.wallets.Wallet method*), 84  
`transactions_update_confirmations()` (*bitcoinlib.wallets.Wallet method*), 84  
`tx_count` (*bitcoinlib.db\_cache.DbCacheBlock attribute*), 130  
`txid` (*bitcoinlib.db.DbTransaction attribute*), 126  
`txid` (*bitcoinlib.db\_cache.DbCacheTransaction attribute*), 131  
`txid` (*bitcoinlib.db\_cache.DbCacheTransactionNode attribute*), 132  
`txid` (*bitcoinlib.keys.Signature property*), 42  
`type` (*bitcoinlib.db\_cache.DbCacheVars attribute*), 132

## U

`update_scripts()` (*bitcoinlib.transactions.Input*

method), 48  
 update\_totals() (bitcoinlib.transactions.Transaction method), 57  
 used (bitcoinlib.db.DbKey attribute), 124  
 utxo\_add() (bitcoinlib.wallets.Wallet method), 84  
 utxo\_last() (bitcoinlib.wallets.Wallet method), 84  
 utxos() (bitcoinlib.wallets.Wallet method), 84  
 utxos\_update() (bitcoinlib.wallets.Wallet method), 85

## V

value (bitcoinlib.db.DbConfig attribute), 122  
 value (bitcoinlib.db.DbTransactionInput attribute), 127  
 value (bitcoinlib.db.DbTransactionOutput attribute), 128  
 value (bitcoinlib.db\_cache.DbCacheTransactionNode attribute), 132  
 value (bitcoinlib.db\_cache.DbCacheVars attribute), 132  
 Value (class in bitcoinlib.values), 100  
 value\_sat (bitcoinlib.values.Value property), 103  
 value\_to\_satoshi() (in module bitcoinlib.values), 103  
 varbyteint\_to\_int() (in module bitcoinlib.encoding), 140  
 variable (bitcoinlib.db.DbConfig attribute), 122  
 varname (bitcoinlib.db\_cache.DbCacheVars attribute), 132  
 varstr() (in module bitcoinlib.encoding), 140  
 verified (bitcoinlib.db.DbTransaction attribute), 126  
 verify() (bitcoinlib.keys.Signature method), 42  
 verify() (bitcoinlib.transactions.Transaction method), 58  
 verify() (in module bitcoinlib.keys), 46  
 version (bitcoinlib.db.DbTransaction attribute), 126  
 version (bitcoinlib.db\_cache.DbCacheBlock attribute), 130  
 version (bitcoinlib.db\_cache.DbCacheTransaction attribute), 131  
 version\_bin (bitcoinlib.blocks.Block property), 99  
 version\_bips() (bitcoinlib.blocks.Block method), 99

## W

wallet (bitcoinlib.db.DbKey attribute), 124  
 wallet (bitcoinlib.db.DbTransaction attribute), 126  
 Wallet (class in bitcoinlib.wallets), 66  
 wallet\_create\_or\_open() (in module bitcoinlib.wallets), 89  
 wallet\_delete() (in module bitcoinlib.wallets), 89  
 wallet\_delete\_if\_exists() (in module bitcoinlib.wallets), 90  
 wallet\_empty() (in module bitcoinlib.wallets), 90  
 wallet\_exists() (in module bitcoinlib.wallets), 90  
 wallet\_id (bitcoinlib.db.DbKey attribute), 124  
 wallet\_id (bitcoinlib.db.DbTransaction attribute), 126  
 WalletError, 86  
 WalletKey (class in bitcoinlib.wallets), 86

wallets\_list() (in module bitcoinlib.wallets), 90  
 WalletTransaction (class in bitcoinlib.wallets), 87  
 weight\_units (bitcoinlib.transactions.Transaction property), 58  
 wif (bitcoinlib.db.DbKey attribute), 124  
 wif() (bitcoinlib.keys.HDKey method), 37  
 wif() (bitcoinlib.keys.Key method), 40  
 wif() (bitcoinlib.wallets.Wallet method), 85  
 wif\_key() (bitcoinlib.keys.HDKey method), 37  
 wif\_prefix() (bitcoinlib.networks.Network method), 93  
 wif\_prefix\_search() (in module bitcoinlib.networks), 95  
 wif\_private() (bitcoinlib.keys.HDKey method), 37  
 wif\_public() (bitcoinlib.keys.HDKey method), 38  
 with\_prefix() (bitcoinlib.keys.Address method), 32  
 witness\_data() (bitcoinlib.transactions.Transaction method), 58  
 witness\_type (bitcoinlib.db.DbTransaction attribute), 126  
 witness\_type (bitcoinlib.db.DbTransactionInput attribute), 127  
 witness\_type (bitcoinlib.db.DbWallet attribute), 129  
 witness\_type (bitcoinlib.db\_cache.DbCacheTransaction attribute), 131  
 witnesses (bitcoinlib.db\_cache.DbCacheTransactionNode attribute), 132  
 WitnessTypeTransactions (class in bitcoinlib.db\_cache), 132  
 word() (bitcoinlib.mnemonic.Mnemonic method), 92  
 wordlist() (bitcoinlib.mnemonic.Mnemonic method), 92

## X

x (bitcoinlib.keys.Key property), 40

## Y

y (bitcoinlib.keys.Key property), 40