Bitcoinlib Documentation

Release 0.7.9

Cryp Toon (mccwdev)

Jun 05, 2026

MANUALS

The Python BitcoinLib is a library that provides developers with a wide range of tools to work with Bitcoin.

This library is flexible and modular: you can use specific modules without loading the entire library.

Below you can find some basic examples, link to some manuals and an overview of modules and classes.

For questions, issues, ideas please check Github Discussions or the Github issues page.

CHAPTER

ONE

WALLET

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

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

```
>>> from bitcoinlib.wallets import Wallet
>>> w = Wallet.create('MyWallet')
>>> key1 = w.get_key()
>>> key1.address
'bc1q037g9zlyjzyznqhhqjqhd94s7tsphl0yczlpcs'
```

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('bc1q9fvq0dq8ku3ygjmjxx06x80mzafe4z5gzdah95', '0.001 BTC')
'b7feea5e7c79d4f6f343b5ca28fa2a1fcacfe9a2b7f44f3d2fd8d6c2d82c4078'
>>> t.info()  # Shows transaction information and send results
```

CHAPTER

TWO

WALLET FROM PASSPHRASE WITH ACCOUNTS AND MULTIPLE CURRENCIES

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

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

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

CHAPTER

THREE

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()
```

CHAPTER

FOUR

SIGN A MESSAGE WITH A BITCOIN WALLET

Create a signed message with a Bitcoinlib wallet. After signing you can verify it with another wallet or an online service, to prove you are the owner of the private key

```
>>> message = "This is a signed message as proof I own this wallet and private key. Alice
↪"
>>> w = wallet_create_or_open('message-signing-wallet')
>>> sig = w.sign_message(message)
>>> sig.as_signed_message(message)
-----BEGIN BITCOIN SIGNED MESSAGE-----
This is a example of a signed message by a Bitcoinlib wallet
-----BEGIN SIGNATURE-----
bc1q065ed8cnxf9rr2v0qhatr2guruulr0jpgjycfu
J4YKfaKqHDJCidfWXYTRC191QYZ3slu8VzgFyUb6m7QJn7Yn2sif7Od1slvnhBV6pLqj2cnISyahVCIrrHQNe0I=
-----END BITCOIN SIGNED MESSAGE-----
```

CHAPTER

FIVE

COMMAND LINE TOOL

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

To create a new Bitcoin wallet

```
$ python bitcoinlib/tools/clw.py new -w newwallet
Command Line Wallet - BitcoinLib 0.6.29

CREATE wallet 'newwallet' (bitcoin network)
Passphrase: sibling undo gift cat garage survey taxi index admit odor surface waste
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
Wallet info for newwallet
=== WALLET ===
 ID                            21
 Name                          newwallet
 Owner
 Scheme                        bip32
 Multisig                      False
 Witness type                  segwit
 Main network                  bitcoin
 Latest update                 None

= Wallet Master Key =
 ID                            177
 Private                       True
 Depth                         0

- NETWORK: bitcoin -
- - Keys
  182 m/84`/0`/0`/0/0             bc1qza24j7snqlmx7603z8qplm4rzfkr0p0mneraqv    address␣
↪index 0                        0.00000000

- - Transactions Account 0 (0)

= Balance Totals (includes unconfirmed) =
```

You can use the command line wallet ‘clw’ to create simple or multisig wallets for various networks, manage public and private keys and managing transactions.

For the full command line wallet documentation please read

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

CHAPTER

SIX

SERVICE PROVIDERS

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

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

CHAPTER

SEVEN

OTHER DATABASES

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

CHAPTER

EIGHT

MORE EXAMPLES

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

8.1 Install and setup 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 for Ubuntu, for other systems see below:

```
apt install build-essential python3-dev libgmp3-dev pkg-config postgresql
postgresql-contrib mariadb-server libpq-dev libmysqlclient-dev pkg-config
```

Create a virtual environment for instance on linux with virtualenv:

```
$ virtualenv -p ~/.virtualenvs/bitcoinlib
$ source ~/.virtualenvs/bitcoinlib/bin/activate
```

Then clone the repository and install dependencies:

```
$ git clone https://github.com/1200wd/bitcoinlib.git
$ cd bitcoinlib
$ python -m pip install .
```

You can test your local installation by running all unittests:

```
$ python -m unittest
```

8.1.1.3 Package dependencies

Required Python Packages, are automatically installed upon installing bitcoinlib:

- fastecdsa (or ecdsa on Windows)
- sqlalchemy
- requests
- pycryptodome

8.1.1.4 Other requirements Linux

On Debian, Ubuntu or their derivatives:

```
apt install build-essential python3-dev libgmp3-dev pkg-config postgresql
postgresql-contrib mariadb-server libpq-dev libmysqlclient-dev pkg-config
```

On Fedora, CentOS or RHEL:

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

On Alpine Linux, lightweight Linux used for Docker images:

```
apk add python3-dev gmp-dev py3-pip gcc musl-dev libpq-dev postgresql postgresql-contrib
mariadb-dev mysql-client
```

On Kali linux:

```
apt install libgmp3-dev postgresql postgresql-contrib libpq-dev pkg-config
default-libmysqlclient-dev default-mysql-server
```

8.1.1.5 Development environment

Install database packages for MySQL and PostgreSQL

```
apt install mysql-server postgresql postgresql-contrib libmysqlclient-dev pkg-config
libpq-dev
```

Check for the latest version of the PostgreSQL dev server:

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

From library root directory install the Python requirements

```
python -m pip install .[dev]
```

Then run the unittests to see if everything works

```
python -m unittest
```

8.1.1.6 Run without database

If you do need the Wallet, Service or bitcoinlib tools you can use the library without database. Remove the sqlalchemy module from the requirements or uninstall it with:

```
python -m pip uninstall sqlalchemy
```

When you uninstall the sqlalchemy database not all unittests will run, you need to run them separately. For instance to run the test_keys unittest:

```
python -m pip unittest tests/test_keys.py
```

8.1.1.7 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.

When using Python on Windows it needs to be set to UTF-8 mode. You can do this by adding the PYTHONUTF8=1 to the environment variables or use the -X utf8 command line option. Please see https://docs.python.org/3/using/windows.html#win-utf8-mode for more information.

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
```

8.1.3 Troubleshooting

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

You can also use pycryptodome, pyscrypt or scrypt. pyscript is a pure Python scrypt password-based key derivation library. It works but it is slow when using BIP38 password protected keys.

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 Change data directory

The default location for Bitcoinlib settings and data is in the user's dot bitcoinlib directory. For Linux this would be ~/.bitcoinlib. You can change the location of data, settings and database by specifying the directory in the system environment 'BCL_DATA_DIR' variable:

```
os.environ['BCL_DATA_DIR'] = '/var/www/blocksmurfer/.bitcoinlib'
# or a second directory in user home
os.environ['BCL_DATA_DIR'] = '~/.bitcoinlib2'
```

8.1.5 Service providers and local nodes

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

To increase reliability, speed and privacy or if you use this library in a production environment it is advised to run your own Bcoin or Bitcoin node.

More setup information:

- Setup connection to Bcoin node
- 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 Bitcoin wallets from commandline

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 exist 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 new -w mywallet
CREATE wallet 'newwallet' (bitcoin network)
Passphrase: sibling undo gift cat garage survey taxi index admit odor surface waste
Please write it 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
Wallet info for newwallet
=== WALLET ===
 ID                            21
 Name                          newwallet
 Owner
 Scheme                        bip32
 Multisig                      False
 Witness type                  segwit
 Main network                  bitcoin
 Latest update                 None

= Wallet Master Key =
 ID                            177
 Private                       True
 Depth                         0

- NETWORK: bitcoin -
- - Keys
  182 m/84'/0'/0'/0/0             bc1qza24j7snqlmx7603z8qplm4rzfkr0p0mneraqv    address␣
↪index 0                        0.00000000

- - Transactions Account 0 (0)

= Balance Totals (includes unconfirmed) =
```

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 -w mywallet -r
Command Line Wallet for BitcoinLib

Receive address is bc1qza24j7snqlmx7603z8qplm4rzfkr0p0mneraqv
```

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 -w mywallet -d dbtest -t bc1qza24j7snqlmx7603z8qplm4rzfkr0p0mneraqv 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 -u / –update-transactions option. This will create new addresses and update unspent outputs.

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

The -i / –wallet-info shows the contents of the updated wallet.

8.2.5 Encrypt private key fields

Bitcoinlib has build in functionality to encrypt private key fields in the database. If you provide a password in the runtime environment the data is encrypted at low level in the database module. You can provide a 32 byte key in the DB_FIELD_ENCRYPTION_KEY variable or a password in the DB_FIELD_ENCRYPTION_PASSWORD variable.

```
$ export DB_FIELD_ENCRYPTION_PASSWORD=iforgot
$ clw new -w cryptwallet
Command Line Wallet - BitcoinLib 0.6.14

CREATE wallet 'cryptwallet' (bitcoin network)
Passphrase: job giant vendor side oil embrace true cushion have matrix glimpse rack
Please write it 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
... wallet info ...

$ clw -w cryptwallet -r
Command Line Wallet - BitcoinLib 0.6.14

Receive address: bc1q2cr0chgs6530mdpag2rfn7v9nt232nlpqcc4kc
Install qr code module to show QR codes: pip install pyqrcode
```

If we now remove the password from the environment, we cannot open the wallet anymore:

```
$ export DB_FIELD_ENCRYPTION_PASSWORD=
$ clw -w cryptwallet -i
Command Line Wallet - BitcoinLib 0.6.14

ValueError: Data is encrypted please provide key in environment
```

8.2.6 Example: Multi-signature Bitcoinlib test wallet

First we generate 2 private keys to create a 2-of-2 multisig wallet:

```
$ clw -g -n bitcoinlib_test -y
Command Line Wallet - BitcoinLib 0.6.14

Passphrase: marine kiwi great try know scan rigid indicate place gossip fault liquid
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
Private Master key, to create multisig wallet on this machine:
BC19UtECk2r9PVQYhY4yboRf92XKEnKZf9hQEd1qBqCgQ98HkBeysLPqYewcWDUuaBRSSVXCShDfmhpbtgZ33sWeGPqfwoLwamzPEcn
Public Master key, to share with other cosigner multisig wallets:
BC18rEEZrakM87qWbSSUv19vnRkEFL7ZtNtGx3exB886VbeFZp6aq9JLZucYAj1EtsHKUB2mkjvafCCGaeYeUVtdFcz5xTxTTgEPCE8
Network: bitcoinlib_test

$ clw -g -n bitcoinlib_test -y
Command Line Wallet - BitcoinLib 0.6.14

Passphrase: trumpet utility cotton couch hard shadow ivory alpha glance pear snow emerge
Please write down on paper and backup. With this key you can restore your wallet and all␣
↪keys
Private Master key, to create multisig wallet on this machine:
BC19UtECk2r9PVQYhaAa8kEgBMPWHC4fJVJD48zBMMb9gSpY9LQVvQ1HhzB3Xmkm2BpiH5SyWoboiewpbeexPLsw8QBfAqMbDfet6kL
Public Master key, to share with other cosigner multisig wallets:
BC18rEEvE8begagfJs7kdxx1yW9tFsz7879c9vQQ2mnGbF6WSeKuBEGtmxJYLEy8rpVV9wXffbBtnL1LPKZqujPtEKzHqQeERiRybKB
Network: bitcoinlib_test
```

The -g / –generate-key is used to generate a private key passphrase. With -n / –network we specify the bitcoinlib_test network. This isn't actually a network but allows us to create and verify transactions. The -y / –yes options, skips the required user input. We now use 1 private and 1 public key to create a wallet.

```
$ clw new -w multisig-2-2 -n bitcoinlib_test -m 2 2␣
↪BC19UtECk2r9PVQYhY4yboRf92XKEnKZf9hQEd1qBqCgQ98HkBeysLPqYewcWDUuaBRSSVXCShDfmhpbtgZ33sWeGPqfwoLwamzPE
↪BC18rEEvE8begagfJs7kdxx1yW9tFsz7879c9vQQ2mnGbF6WSeKuBEGtmxJYLEy8rpVV9wXffbBtnL1LPKZqujPtEKzHqQeERiRybK

Command Line Wallet - BitcoinLib 0.6.14

CREATE wallet 'ms22' (bitcoinlib_test network)
Wallet info for ms22
=== WALLET ===
 ID                          22
 Name                        ms22
 Owner
 Scheme                      bip32
 Multisig                    True
 Multisig Wallet IDs         23, 24
 Cosigner ID                 1
 Witness type                segwit
 Main network                bitcoinlib_test
 Latest update               None
```

(continues on next page)

(continued from previous page)

```
= Multisig Public Master Keys =
    0 183␣
↪BC18rEEvE8begagfJs7kdxx1yW9tFsz7879c9vQQ2mnGbF6WSeKuBEGtmxJYLEy8rpVV9wXffbBtnL1LPKZqujPtEKzHqQeERiRybk
↪bip32  cosigner
    1 186␣
↪BC18rEEZrakM87qWbSSUv19vnRkEFL7ZtNtGx3exB886VbeFZp6aq9JLZucYAj1EtsHKUB2mkjvafCCGaeYeUVtdFcz5xTxTTgEPCk
↪bip32  main     *
For main keys a private master key is available in this wallet to sign transactions. *␣
↪cosigner key for this wallet

- NETWORK: bitcoinlib_test -
- - Keys

- - Transactions Account 0 (0)

= Balance Totals (includes unconfirmed) =
```

The multisig wallet has been created, you can view the wallet info by using the -i / –wallet-info option. Now we generate a new receiving address with the -r / –receive option and update the unspent outputs with the -x / –update-utxos option.

```
$ clw -w ms22 -r
Command Line Wallet - BitcoinLib 0.6.14

Receive address: blt1qxu6z7evkrmz5s7sk63dr0u3h9xsf2j2vys88reg75cjvjuz4vf2srkxp7p
Install qr code module to show QR codes: pip install pyqrcode

$ clw -w ms22 -x
Command Line Wallet - BitcoinLib 0.6.14

Updating wallet utxo's
$ clw -w ms22 -i
Command Line Wallet - BitcoinLib 0.6.14

Wallet info for ms22
=== WALLET ===
 ID                            22
 Name                          ms22
 Owner
 Scheme                        bip32
 Multisig                      True
 Multisig Wallet IDs           23, 24
 Cosigner ID                   1
 Witness type                  segwit
 Main network                  bitcoinlib_test
 Latest update                 None

= Multisig Public Master Keys =
    0 183␣
↪BC18rEEvE8begagfJs7kdxx1yW9tFsz7879c9vQQ2mnGbF6WSeKuBEGtmxJYLEy8rpVV9wXffbBtnL1LPKZqujPtEKzHqQeERiRybk
↪bip32  cosigner
    1 186␣
↪BC18rEEZrakM87qWbSSUv19vnRkEFL7ZtNtGx3exB886VbeFZp6aq9JLZucYAj1EtsHKUB2mkjvafCCGaeYeUVtdFcz5xTxTTgEPCk
```

(continues on next page)

(continued from previous page)

```
↪bip32  main     *
For main keys a private master key is available in this wallet to sign transactions. *␣
↪cosigner key for this wallet

- NETWORK: bitcoinlib_test -
- - Keys
  193 m/48`/9999999`/0`/2`/0/0    ␣
↪blt1qxu6z7evkrmz5s7sk63dr0u3h9xsf2j2vys88reg75cjvjuz4vf2srkxp7p Multisig Key 185/192  ␣
↪                2.00000000 T

- - Transactions Account 0 (2)
7b020ae9c7f8ba84a5a5136ae32e6195af5a4f25316f790a1278e04f479ca77d␣
↪blt1qxu6z7evkrmz5s7sk63dr0u3h9xsf2j2vys88reg75cjvjuz4vf2srkxp7p       10          1.
↪00000000 T U
5d0f176259ab4bc596363aa3653c44858ebeb2fd8361311966776192968e545d␣
↪blt1qxu6z7evkrmz5s7sk63dr0u3h9xsf2j2vys88reg75cjvjuz4vf2srkxp7p       10          1.
↪00000000 T U

= Balance Totals (includes unconfirmed) =
bitcoinlib_test      (Account 0)                  2.00000000 T
```

We now have some utxo's in our wallet so we can create a transaction

```
$ clw -w ms22 -s blt1qxu6z7evkrmz5s7sk63dr0u3h9xsf2j2vys88reg75cjvjuz4vf2srkxp7p 0.1
Connected to pydev debugger (build 233.13135.95)
Command Line Wallet - BitcoinLib 0.6.14

Transaction created
Transaction 3b96f493d189667565271041abbc0efbd8631bb54d76decb90e144bb145fa613
Date: None
Network: bitcoinlib_test
Version: 1
Witness type: segwit
Status: new
Verified: False
Inputs
- blt1qxu6z7evkrmz5s7sk63dr0u3h9xsf2j2vys88reg75cjvjuz4vf2srkxp7p 1.00000000 TST␣
↪7b020ae9c7f8ba84a5a5136ae32e6195af5a4f25316f790a1278e04f479ca77d 0
  segwit p2sh_multisig; sigs: 1 (2-of-2) not validated
Outputs
- blt1qxu6z7evkrmz5s7sk63dr0u3h9xsf2j2vys88reg75cjvjuz4vf2srkxp7p 0.10000000 TST p2wsh U
- blt1qe4tr993nftagprtapclxrm7ahrcvl4w0dnxfnhz2cx6pjaeg989syy9zge 0.89993601 TST p2wsh U
Size: 192
Vsize: 192
Fee: 6399
Confirmations: None
Block: None
Pushed to network: False
Wallet: ms22

Transaction created but not sent yet. Transaction dictionary for export:
{<dictionary>}
```

Copy the contents of the dictionary and save it as 3b96f493d189667565271041abbc0efbd8631bb54d76decb90e144bb145fa613.tx

The transaction has been created, but cannot be verified because the wallet contains only 1 private key. So we need to create another wallet with the other private key, in real life situations this would be on another (offline) machine.

Below we create a new wallet, generate a receive address and update the utxo's. Finally we can import the transaction dictionary which will be signed once imported. And as you can see the transaction has been verified now!

```
$ clw new -w multisig-2-2-signer2 -n bitcoinlib_test -m 2 2␣
↪BC18rEEZrakM87qWbSSUv19vnRkEFL7ZtNtGx3exB886VbeFZp6aq9JLZucYAj1EtsHKUB2mkjvafCCGaeYeUVtdFcz5xTxTTgEPC
↪BC19UtECk2r9PVQYhaAa8kEgBMPWHC4fJVJD48zBMMb9gSpY9LQVvQ1HhzB3Xmkm2BpiH5SyWoboiewpbeexPLsw8QBfAqMbDfet6
$ clw -w multisig-2-2-signer2 -r
$ clw -w multisig-2-2-signer2 -x
$ clw -w multisig-2-2-signer2 -a tx.tx
Command Line Wallet - BitcoinLib 0.6.14

Transaction 3b96f493d189667565271041abbc0efbd8631bb54d76decb90e144bb145fa613
Date: None
Network: bitcoinlib_test
Version: 1
Witness type: segwit
Status: new
Verified: True
Inputs
- blt1qxu6z7evkrmz5s7sk63dr0u3h9xsf2j2vys88reg75cjvjuz4vf2srkxp7p 1.00000000 TST␣
↪7b020ae9c7f8ba84a5a5136ae32e6195af5a4f25316f790a1278e04f479ca77d 0
  segwit p2sh_multisig; sigs: 2 (2-of-2) valid
Outputs
- blt1qxu6z7evkrmz5s7sk63dr0u3h9xsf2j2vys88reg75cjvjuz4vf2srkxp7p 0.10000000 TST p2wsh U
- blt1qe4tr993nftagprtapclxrm7ahrcvl4w0dnxfnhz2cx6pjaeg989syy9zge 0.89993601 TST p2wsh U
Size: 192
Vsize: 192
Fee: 6399
Confirmations: None
Block: None
Pushed to network: False
Wallet: multisig-2-2-signer2


Signed transaction:
{<dictionary>}
```

8.2.7 Options Overview

Command Line Wallet for BitcoinLib

usage: clw.py [-h] [–list-wallets] [–generate-key] [–passphrase-strength PASSPHRASE_STRENGTH] [–database DATABASE] [–wallet_name [WALLET_NAME]] [–network NETWORK] [–witness-type WITNESS_TYPE] [–yes]

[–quiet] [–wallet-remove] [–wallet-info] [–update-utxos] [–update-transactions] [–wallet-empty] [–receive] [–cosigner-id COSIGNER_ID] [–export-private] [–import-private IMPORT_PRIVATE] [–send ADDRESS AMOUNT] [–number-of-change-outputs NUMBER_OF_CHANGE_OUTPUTS] [–input-key-id INPUT_KEY_ID] [–sweep ADDRESS] [–fee FEE] [–fee-per-kb FEE_PER_KB] [–push] [–import-tx TRANSACTION] [–import-tx-file FILENAME_TRANSACTION] {new} ...

BitcoinLib command line wallet

positional arguments:
{new}

options:

-h, --help show this help message and exit

--list-wallets, -l List all known wallets in database

--generate-key, -g Generate a new masterkey, and show passphrase, WIF and public account key. Can be used to create a new (multisig) wallet

--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

--database DATABASE, -d DATABASE URI of the database to use

–wallet_name [WALLET_NAME], -w [WALLET_NAME]
Name of wallet to create or open. Provide wallet name or number when running wallet actions

--network NETWORK, -n NETWORK Specify 'bitcoin', 'litecoin', 'testnet' or other supported network

--witness-type WITNESS_TYPE, -j WITNESS_TYPE Witness type of wallet: legacy, p2sh-segwit or segwit (default)

--yes, -y Non-interactive mode, does not prompt for confirmation

--quiet, -q Quiet mode, no output written to console

Wallet Actions:

--wallet-remove Name or ID of wallet to remove, all keys and transactions will be deleted

--wallet-info, -i 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-empty, -z Delete all keys and transactions from wallet, except for the masterkey(s). Use when updating fails or other errors occur. Please backup your database and masterkeys first. Update empty wallet again to restore your wallet.

--receive, -r Show unused address to receive funds.

--cosigner-id COSIGNER_ID, -o COSIGNER_ID Set this if wallet contains only public keys, more than one private key or if you would like to create keys for other cosigners.

--export-private, -e Export private key for this wallet and exit

--import-private IMPORT_PRIVATE, -v IMPORT_PRIVATE Import private key in this wallet

Transactions:

–send ADDRESS AMOUNT, -s ADDRESS AMOUNT
Create transaction to send amount to specified address. To send to multiple addresses, argument can be used multiple times.

--number-of-change-outputs NUMBER_OF_CHANGE_OUTPUTS Number of change outputs. Default is 1, increase for more privacy or to split funds

--input-key-id INPUT_KEY_ID, -k INPUT_KEY_ID Use to create transaction with 1 specific key ID

--sweep ADDRESS Sweep wallet, transfer all funds to specified address

--fee FEE, -f FEE Transaction fee

--fee-per-kb FEE_PER_KB, -b FEE_PER_KB Transaction fee in satoshi per kilobyte

--push, -p Push created transaction to the network

--import-tx 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)

8.2.8 Options overview: New Wallet

usage: clw.py new [-h] –wallet_name [WALLET_NAME] [–password PASSWORD] [–network NETWORK] [–passphrase PASSPHRASE] [–create-from-key KEY] [–create-multisig [. . .]] [–witness-type WITNESS_TYPE]
[–cosigner-id COSIGNER_ID] [–database DATABASE] [–receive] [–yes] [–quiet]

Create new wallet

options:

-h, --help show this help message and exit

–wallet_name [WALLET_NAME], -w [WALLET_NAME]
Name of wallet to create or open. Provide wallet name or number when running wallet actions

--password PASSWORD Password for BIP38 encrypted key. Use to create a wallet with a protected key

--network NETWORK, -n NETWORK Specify 'bitcoin', 'litecoin', 'testnet' or other supported network

--passphrase PASSPHRASE Passphrase to recover or create a wallet. Usually 12 or 24 words

--create-from-key KEY, -c KEY Create a new wallet from specified key

–create-multisig [. . .], -m [. . .]
[NUMBER_OF_SIGNATURES_REQUIRED, NUMBER_OF_SIGNATURES, KEY-1, KEY-2, … KEY-N]Specify number of signatures followed by the number of signatures required and then a list of public or private keys for this wallet. Private keys will be created if not provided in key list. Example, create a 2-of-2 multisig wallet and provide 1 key and create another key: -m 2 2 tprv8ZgxMBicQKsPd1Q44tfDiZC98iYouKRC2CzjT3HGt1yYw2zuX2awTotzGAZQEAU9bi2M5MCj8iedP9MREPjUgpD tprv8ZgxMBicQKsPeUbMS6kswJc11zgVEXUnUZuGo3bF6bBrAg1ieFfUdPc9UHqbD5HcXizThrcKike1c4z6xHrz6MWC

--witness-type WITNESS_TYPE, -j WITNESS_TYPE Witness type of wallet: legacy, p2sh-segwit or segwit (default)

--cosigner-id COSIGNER_ID, -o COSIGNER_ID Set this if wallet contains only public keys, more then one private key or if you would like to create keys for other cosigners.

--database DATABASE, -d DATABASE URI of the database to use

--receive, -r Show unused address to receive funds.

--yes, -y Non-interactive mode, does not prompt for confirmation

--quiet, -q Quiet mode, no output written to console

8.3 Using MySQL or PostgreSQL databases

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

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

The database URI can be passed to the Wallet or Service object, or you can set the database URI for wallets and / or cache in configuration file at ~/.bitcoinlib/config.ini

8.3.1 Using MariaDB / 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 of 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()
```

At the moment it is not possible to use MySQL database for caching, because the BLOB transaction ID's are used as primary key. For caching you need to use a PostgreSQL or SQLite database.

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

Please note 'postgresql+psycopg' has to be used as scheme, because SQLalchemy uses the latest version 3 of psycopg, if not provided it will use psycopg2.

PostgreSQL can also be used for caching of service requests. The URI can be passed to the Service object or provided in the configuration file (~/.bitcoiinlib/config.ini)

```
srv = Service(cache_uri='postgresql+psycopg://postgres:postgres@localhost:5432/)
res = srv.gettransactions('12spqcvLTFhL38oNJDDLfW1GpFGxLdaLCL')
```

8.3.3 Encrypt database or private keys

If you are using wallets with private keys it is advised to use an encrypted database and / or to encrypt the private key fields.

Please read Encrypt Database or Private Keys for more information.

8.4 Encrypt Database or Private Keys

If your database contains private keys it is a good idea to encrypt your data. This will not be done automatically. At the moment you have 2 options:

- Encrypt the database with SQLCipher. The database is fully encrypted and you need to provide the password in the Database URI when opening the database.
- Use a normal database but all private key data will be stored AES encrypted in the database. A key to encrypt and decrypt needs to be provided in the Environment.

8.4.1 Encrypt database with SQLCipher

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 sqlcipher3-binary
# Previous, but now unmaintained: $ pip install pysqlcipher3
```

Create an Encrypted Database for your Wallet

Now you can simply create and use an encrypted database by supplying a password as an argument to the Wallet object:

```
password = 'secret'
db_uri = '/home/user/.bitcoinlib/database/bcl_encrypted.db'
wlt = wallet_create_or_open('bcltestwlt4', network='bitcoinlib_test', db_uri=db_uri, db_
↪password=password)
```

Encrypt using Database URI

You can also use a SQLCipher database URI to create and query an encrypted database:

```
password = 'secret'
filename = '/home/user/.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>
```

8.4.2 Encrypt private key fields with AES

It is also possible to just encrypt the private keys in the database with secure AES encryption. You need to provide a key or password as environment variable.

- You can skip this step if you want, but this provides an extra warning / check when no encryption key is found: Enable database encryption in Bitcoinlib configuration settings at ~/.bitcoinlib/config.ini

```
# Encrypt private key field in database using symmetrically EAS encryption.
# You need to set the password in the DB_FIELD_ENCRYPTION_KEY environment variable.
database_encryption_enabled=True
```

You can provide an encryption key directly or use a password to create a key:

1. Generate a secure 32 bytes encryption key yourself with Bitcoinlib:

```
>>> from bitcoinlib.keys import Key
>>> Key().private_hex()
'2414966ea9f2de189a61953c333f61013505dfbf8e383b5ed6cb1981d5ec2620'
```

This key needs to be stored in the environment when creating or accessing a wallet. No extra arguments have to be provided to the Wallet class, the data is encrypted and decrypted at database level.

2. You can also just provide a password, and let Bitcoinlib create a key for you. You will need to pass the DB_FIELD_ENCRYPTION_PASSWORD environment variable.

There are several ways to store the key in an Environment variable, on Linux you can do:

```
$ export DB_FIELD_ENCRYPTION_KEY=
↪'2414966ea9f2de189a61953c333f61013505dfbf8e383b5ed6cb1981d5ec2620'
```

or

```
$ export DB_FIELD_ENCRYPTION_PASSWORD=ineedtorememberthispassword
```

Or in Windows:

```
$ setx DB_FIELD_ENCRYPTION_KEY
↪'2414966ea9f2de189a61953c333f61013505dfbf8e383b5ed6cb1981d5ec2620'
```

Environment variables can also be stored in an .env key, in a virtual environment or in Python code itself. However anyone with access to the key can decrypt your private keys.

Please make sure to remember and backup your encryption key or password, if you lose your key the private keys can not be recovered!

8.5 Frequently Asked Questions

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

1. Run your own Bitcoin core, Bcoin node or Blockbook server, 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 of 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 or private keys with SQLCipher or AES.
6. Use password protected private keys. For instance use a password when creating wallets.
7. Backup private keys and passwords! We have no proof but I assume more bitcoins are lost because of lost private keys than 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 these tips, read some more about Security and Privacy and then think thoroughly about the best wallet setup, which is always a tradeoff between security, privacy and usability.

8.6 Caching

Results from queries to service providers are stored 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.

8.6.1 What is cached?

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

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

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

8.6.2 Using other databases

By default the cache is stored in an 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 types of databases can be used as well, check http://bitcoinlib.readthedocs.io/en/latest/_static/manuals.databases.html for more information.

8.6.3 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.

8.6.4 Troubleshooting

8.6.4.1 Nothing is cached, what is the problem?

- If the min_providers parameter is set to 2 or more caching will be disabled.
- If a service providers returns an incomplete result no cache will be stored.
- If the after_txid parameter is used in gettransactions() or getutxos() no cache will be stored if this '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.

8.6.4.2 I get incomplete or incorrect results!

- Please post an issue 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.

8.7 Frequently Asked Questions

8.7.1 Can I use Bitcoinlib on my system?

BitcoinLib is platform independent and should run on your system. Bitcoinlib is mainly developed on Ubuntu linux and runs unittests on every commit on Ubuntu and Windows. Dockerfiles are available for Alpine, Kali and Fedora. You can find all dockerfiles on https://github.com/1200wd/bitcoinlib/tree/master/docker

8.7.2 I run into an error 'x' when installing Bitcoinlib

1. Check the installation page and see if you have installed all the requirements.
2. Install the required packages one-by-one using pip install, and see if you get any specific errors.
3. Check for help in Github Discussions.
4. See if you find any known issue.
5. If it doesn't work out, do not hesitate to ask your question in the github discussions or post an issue!

8.7.3 Does Bitcoinlib support 'x'-coin

Bitcoinlib main focus is on Bitcoin. But besides Bitcoin it supports Litecoin and Dogecoin. For testing it supports the Bitcoin testing networks: testnet3, testnet4, regtest and signet. For other coins the Litecoin testnet and Dogecoin testnet is supported.

Support for Dash, Bitcoin Cash and Bitcoin SV has been dropped. There are currently no plans to support other coins. Main problem with supporting new coins is the lack of service provides with a working and stable API.

8.7.4 My wallet transactions are not (correctly) updating!

Most likely cause is a problem with a specific service provider.

Please set log level to 'debug' and check the logs in bitcoinlib.log to see if you can pin down the specific error. You could then disable the provider and post the issue.

To avoid these kinds of errors it is advised to run your local Bcoin node, Blockbook or ElectrumX server.

With a local Bcoin node or Blockbook server you do not depend on external Service providers which increases reliability, security, speed and privacy.

8.7.5 Provider 'x' does not work

If you encounter errors when updating transactions, utxo's, blocks, etc there is probably a problem with a specific provider. You can check the logs in bitcoinlib.log and see which provider has problems. To solve this you can:

- Set priority = 0 for this provider to temporary disable it
- Remove the provider from the providers.json file in you local .bitcoinlib directory
- Use the exclude_provider option when you calling the Service class:

```
srv = Service(exclude_providers=['blocksmurfer'])
```

- Or use a specific provider, for instance your local Blockbook server:

```
srv = Service(providers=['blockbook'])
```

8.7.6 Can I use Bitcoinlib with another database besides SQLite?

Yes, the library can also work with PostgreSQL or MySQL / MariaDB databases. For more information see: Databases.

8.7.7 I have imported a private key from another wallet but the address is different

If you have imported a private key from another wallet in Bitcoinlib and the address in Bitcoinlib does not match, then this is probably because the wallets use different key paths or keys at different levels.

- Check if the level and type of the key it the same? Is it a masterkey (level 0), a master public key (level 3), or a key of an address (level 5)
- Does the wallet uses the same key paths? Bitcoinlib uses the default BIP84 keys path in "m/84'/0'/0'/0/0" format. You can specify a different key path or witness type when creating a wallet

8.7.8 I found a bug!

Please help out project and post your issue on Github. Try to include all code and data so we can reproduce and solve the issue.

8.7.9 I have another question

Maybe your question already has an answer om Github Discussions. Or search for an answer is this documentation.

If that does not answer your question, please post your question on the Github Discussions Q&A.

8.8 How to use Service Providers?

Bitcoinlib uses external service providers to get address, transaction and blockchain information for your wallet or other application.

If you install Bitcoinlib it uses external and free service providers, so you do not have to do anything extra to get it working.

However those free providers have transaction limits, slow responses, downtimes or changing Api's which can cause unexpected results. For larges wallets, complicated projects with a lot of requests or production environments, you basically have three options.

8.8.1 Setup a local node

You need a server with a large storage and a stable connection at your home or a hosting provider. And then follow the manual to setup an Blockbook, ElectrumX or Bcoin server.

You can also setup a Bitcoind Node, but a standard node cannot get utxo's or address information for external wallet addresses.

8.8.2 Get a payed subscription

There a many providers which offer a payed subscription, many unfortunately are really expensive.

At the time of writing Nownodes offers a $20 a month subscription. You can create an account and use the example providers configuration file at https://github.com/1200wd/bitcoinlib/blob/master/bitcoinlib/data/providers.examples-nownodes.json

We have not tested other payed service providers at the moment, but please let us know if you found any good options.

8.8.3 Add another service Provider

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

If you tested the new provider thoroughly don't forget to create a pull request ;)

8.9 How to connect bitcoinlib to a Bitcoin node

This manual explains how to connect to a bitcoind server on your localhost or 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 uses a lot of resources.

> **Warning**
>
> With a standard Bitcoin node you can only retrieve block and transaction information. You can not query the node for information about specific addresses. So it is not suitable to run in combination with a Bitcoinlib wallet. If you would like to use Bitcoinlib wallets and not be dependent on external providers you should use a Bcoin node, ElectrumX server or Blockbook server server instead.

8.9.1 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.

Generate a RPC authorization configuration string online: https://jlopp.github.io/bitcoin-core-rpc-auth-generator/ or with the Python tool you can find in the Bitcoin repository: https://github.com/bitcoin/bitcoin/blob/master/share/rpcauth/rpcauth.py

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

```
server=1
port=18332
txindex=1
testnet=1
rpcauth=bitcoinlib:01cf8eb434e3c9434e244daf3fc1cc71
```

(continues on next page)

(continued from previous page)

```
↪$9cdfb346b76935569683c12858e13147eb5322399580ba51d2d878148a880d1d
rpcbind=0.0.0.0
rpcallowip=192.168.0.0/24
```

To increase your privacy and security, and for instance if you run a Bitcoin node on your home network, you can use TOR. Bitcoind has TOR support build in, and it is ease to setup. See https://en.bitcoin.it/wiki/Setting_up_a_Tor_hidden_service

If you have a TOR service running you can add these lines to your bitcoin.conf settings to only use TOR.

```
proxy=127.0.0.1:9050
bind=127.0.0.1
onlynet=onion
```

8.9.2 Connect using provider settings

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

Example:

```
{
  "bitcoind.testnet": {
    "provider": "bitcoind",
    "network": "testnet",
    "client_class": "BitcoindClient",
    "url": "http://user:password@server_url:18332",
    "api_key": "",
    "priority": 11,
    "denominator": 100000000,
    "network_overrides": null
    "timeout": 0
  }
}
```

8.9.3 Connect using base_url argument

You can also directly pass connection string with the 'base_url' argument in the BitcoindClient object.

This provides more flexibility but also the responsibility to store user and password information in a secure way.

```
from bitcoinlib.services.bitcoind import BitcoindClient

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

You can directly r

```
from bitcoinlib.services.bitcoind import BitcoindClient

# Retrieve some blockchain information and statistics
bdc.proxy.getblockchaininfo()
```

(continues on next page)

(continued from previous page)

```
bdc.proxy.getchaintxstats()
bdc.proxy.getmempoolinfo()

# Add a node to the node list
bdc.proxy.addnode('blocksmurfer.io', 'add')
```

8.9.4 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 are disabled. The rpcallowip option cannot be used to listen on all network interfaces and rpcbind has to be used to define specific IP addresses to listen on. See https://bitcoin.org/en/release/v0.18.0#configuration-option-changes

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

8.10 How to connect Bitcoinlib to a Blockbook server

Trezor's Blockbook is a back-end service for Trezor Suite and can also be used as back-end service provider for Bitcoinlib. Blockbook indexes addresses, transactions, unspent outputs and balances and can be used for fast blockchain queries. Blockbook also support Litecoin, Dash, Dogecoin and various testnets.

If you want to use Bitcoinlib as a wallet, run many blockchain queries or write an application which is dependent on frequent requests to blockchain service providers you should use a Blockbook or ElectrumX service or install a Bcoin node.

8.10.1 Install Blockbook server

You can find some instructions on how to install a Blockbook server on https://media.blocksmurfer.io/blockbook-setup-as-bitcoinlib-backend.html.

You will need a powerful server with enough memory and diskspace. Blockbook runs a full Bitcoin core node on the background, and maintains a large RocksDB database to store additional blockchain data. But once installed and synchronised it runs fast and smooth.

8.10.2 Use Blockbook with Bitcoinlib

To use Blockbook with bitcoinlib add the credentials to the providers.json configuration file in the .bitcoinlib directory.

```
"blockbook": {
  "provider": "blockbook",
  "network": "bitcoin",
  "client_class": "BlockbookClient",
  "provider_coin_id": "",
  "url": "https://<servername>:9130/",
  "api_key": "",
  "priority": 20,
  "denominator": 100000000,
  "network_overrides": null,
  "timeout", 0
}
```

You can increase the priority so the Service object always connects to the Blockbook service first.

8.11 How to connect Bitcoinlib to an ElectrumX server

ElectrumX is a backend for the well-known Electrum Bitcoin wallets. It can also be used as back-end service provider for Bitcoinlib wallets. ElectrumX uses a running Bitcoin core node and adds an extra layer with indexes to allow for fast address, unspent outputs and transactions queries. ElectrumX is focussed on Bitcoin wallets and has no option to query blocks and large non-standard transactions.

If you want to use Bitcoinlib as a wallet, run many blockchain queries or write an application which is dependent on frequent requests to blockchain service providers you should use a Blockbook, ElectrumX or Bcoin server.

8.11.1 Install ElectrumX server

You can find instructions on how to install a ElectrumX server on https://media.blocksmurfer.io/install-electrumx-as-bitcoinlib-backend.html

8.11.2 Use ElectrumX with Bitcoinlib

To use ElectrumX with bitcoinlib add the credentials to the providers.json configuration file in the .bitcoinlib directory.

```
"localhost.electrumx": {
    "provider": "electrumx",
    "network": "bitcoin",
    "client_class": "ElectrumxClient",
    "provider_coin_id": "",
    "url": "localhost:50001",
    "api_key": "",
    "priority": 100,
    "denominator": 1,
    "network_overrides": null
},
```

You can increase the priority so the Service object always connects to the ElectrumX service first.

ElectrumX also support Bitcoin testnet, testnet4, regtest and signet. Other coins such as Dogecoin, Litecoin and Dash are also supported. To setup simply update the ports and add add the coin as argument when calling ElectrumX, for instance for testnet4 use: electrumx_server –testnet4

8.12 How to connect Bitcoinlib to a Bcoin node

Bcoin is a full bitcoin node implementation, which can be used to parse the blockchain, send transactions and run a wallet. With a Bcoin node you can retrieve transaction and utxo information for specific addresses, this is not easily possible with a Bitcoind node. So if you want to use Bitcoinlib's wallet functionality for instance and not be dependent on external providers the best option is to run a local Bcoin node, Blockbook or ElectrumX server.

8.12.1 Install Bcoin node

You can find some instructions on how to install a bcoin node on https://media.blocksmurfer.io/install-bcoin-node-ubuntu.html.

There are also some Docker images available. We have created a Docker image with the most optimal settings for bitcoinlib. You can install them with the following command.

```
docker pull blocksmurfer/bcoin
```

8.12.2 Use Bcoin node with Bitcoinlib

To use Bcoin with bitcoinlib add the credentials to the providers.json configuration file in the .bitcoinlib directory.

```
"bcoin": {
    "provider": "bcoin",
    "network": "bitcoin",
    "client_class": "BcoinClient",
    "provider_coin_id": "",
    "url": "https://user:pass@localhost:8332/",
    "api_key": "",
    "priority": 20,
    "denominator": 100000000,
    "network_overrides": null,
    "timeout": 0
},
```

You can increase the priority so the Service object always connects to the Bcoin node first.

8.13 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.

8.13.1 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
        "timeout": 0
    }
}
```

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

Example:

```
from bitcoinlib.services.baseclient import BaseClient

PROVIDERNAME = 'bitgo'


class BitGoClient(BaseClient):

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

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

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

- Add this service class to __init__.py

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

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

```
from bitcoinlib import services

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

8.14 bitcoinlib.keys module

class bitcoinlib.keys.**Address**(*data=''*, *hashed_data=''*, *prefix=None*, *script_type=None*, *compressed=None*, *encoding=None*, *witness_type=None*, *witver=0*, *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.
- **witver** (*int*) – Witness version. Used for p2tr addresses
- **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 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 a 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, 'witver': 0, 'witness_type': 'segwit', 'depth':␣
↪None, 'change': None, 'address_index': None, 'prefix': 'bc', 'redeemscript': '
↪', '_hashed_data': None, 'address':
↪'bc1qyftqrh3hm2yapnhh0ukaht83d02a7pda8l5uhkxk9ftzqsmyu7pst6rke3', 'address_
↪orig': 'bc1qyftqrh3hm2yapnhh0ukaht83d02a7pda8l5uhkxk9ftzqsmyu7pst6rke3'}
```

Parameters

- **address** (*str*) – Address to import
- **compressed** (*bool*) – Is key compressed or not, default is None

- **encoding** (*str*) – Address encoding. Default is base58 encoding, for native segwit addresses specify bech32 encoding. Leave empty to derive from address
- **depth** (*int*) – Level of depth in BIP32 key path
- **change** (*int*) – Use 0 for normal address/key, and 1 for change address (for returned/change payments)
- **address_index** (*int*) – Index of address. Used in BIP32 key paths
- **network** (*str*) – Specify network filter, i.e.: bitcoin, testnet, litecoin, etc. Wil trigger check if address is valid for this network
- **network_overrides** (*dict*) – Override network settings for specific prefixes, i.e.: {"prefix_address_p2sh": "32"}. Used by settings in providers.json

Return Address

with_prefix(*prefix*)

Convert address using another prefix

Parameters

prefix (*str, bytes*) – Address prefix

Return str

Converted address

exception bitcoinlib.keys.**BKeyError**(*msg=''*)

Bases: Exception

Handle Key class Exceptions

class bitcoinlib.keys.**HDKey**(*import_key=None*, *key=None*, *chain=None*, *depth=0*, *parent_fingerprint=b'\x00\x00\x00\x00'*, *child_index=0*, *is_private=None*, *network=None*, *key_type='bip32'*, *password=''*, *compressed=True*, *encoding=None*, *witness_type=None*, *multisig=False*)

Bases: *Key*

Class for Hierarchical Deterministic keys as defined in BIP0032

Besides a private or public key a HDKey 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=zpub6jftahH18ngZw8pNbq6arfqFD7przPySpW3jhhzQiBKYpzJxqwhBEpCk8rh9p7JQ5JrAMu1Pfq1wCXfDfZL8i9
↪ network=bitcoin)>
```

Parameters

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

Return HDKey

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

Get address derived from public key

```
>>> wif =
↪'xpub661MyMwAqRbcFcXi3aM3fVdd42FGDSdufhrr5tdobiPjMrPUykFMTdaFEr7yoy1xxeifDY8kh2k4h9N77MY6rk1
↪'
>>> k = HDKey.from_wif(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 the current HDKey class as a 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 the current key as a JSON formatted string

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

Return str

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

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

Used by *key_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()
'bc1q5rlenzyn95pt4rur6jcnrgx5s3u6kw0nnwjrtt'
>>> ck.depth
1
>>> ck.child_index
10
```

Parameters

- **index** (*int*) – Key index number
- **hardened** (*bool*) – Specify if a 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 a child public key of the current HD Key object.

Used by *key_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 creates 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()
'bc1qlzf2a6tskk8g6g55nsda7akmwsevasv5p4muuf'
>>> ck.depth
1
```

(continues on next page)

(continued from previous page)

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

Parameters

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

Return HDKey
HD Key class object

property fingerprint

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

Return bytes

static from_passphrase(*passphrase*, *password=''*, *network='bitcoin'*, *key_type='bip32'*, *compressed=True*, *encoding=None*, *witness_type='segwit'*, *multisig=False*)

Create a key from the provided 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 addreses

Return HDKey

static from_seed(*import_seed*, *key_type='bip32'*, *network='bitcoin'*, *compressed=True*, *encoding=None*, *witness_type='segwit'*, *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 this key is part of a multisig wallet, used when creating key representations such as WIF and addresses

Return HDKey

static from_wif(*wif*, *network=None*, *compressed=True*, *multisig=None*)

Create HDKey from BIP32 WIF

Parameters

- **wif** (*str*) – HDKey WIF
- **network** (*str*) – Network to use as string
- **compressed** (*bool*) – Is key compressed or not, default is True
- **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

inverse()

Return inverse of this private or public key

Return Key

key_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.from_wif(wif)
>>> k.key_for_path("m/44'/0'/0'/0/2")
<HDKey(public_
↪hex=03004331ca7f0dcdd925abc4d0800a0d4a0562a02c257fa39185c55abdfc4f0c0c, wif_
↪public=xpub6GyQoEbMUNwu1LnbiCSaD8wLrcjyRCEQA8tNsFCH4pnvCbuWSZkSB6LUNe89YsCBTg1Ncs7vHJBjMvw2Q
↪ network=bitcoin)>
```

Parameters

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

Return HDKey

HD Key class object of subkey

network_change(*new_network*)

Change network for current key

Parameters

new_network (*str*) – Name of new network

Return bool
True

public()

Public version of the current private key. Strips all private information from HDKey object, returns deep-copy version of the 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 the 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()

↪'zpub6qN4WhSaerCBXv28QdRXwY4EvnAUbCSLCWUF5ZNSsyrupWwBRv4irjjQ6vpz5WFM7Z7zyy3eQBzijfszpvqdoiy
↪'
```

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 a witness type, default is legacy. Use 'segwit' or 'p2sh-segwit' for segregated witness.
- **as_private** – Return a private key if available. The default is to return the public key

Return HDKey

sign_message(*message*, *use_rfc6979=True*, *k=None*, *hash_type=1*, *force_canonical=False*)

Create a Signature for the provided message. Provide the message in bytes format, this method will add network specific data and will hash the message. By default, a deterministic k value will be used according to the RFC6979 standard.

Parameters

- **message** (*bytes, hexstring*) – Message to be signed. Must be unhashed and in bytes format.
- **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 know what you are doing. Providing wrong value for k can result in leaking your private key!
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL
- **force_canonical** (*bool*) – Some wallets do not require a canonical s value, so you could set this to False

Return Signature

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.

Deprecated: function renamed to *key_for_path()*

```
>>> wif =
↪'xprv9s21ZrQH143K4LvcS93AHEZh7gBiYND6zMoRiZQGL5wqbpCU2KJDY87Txuv9dduk9hAcsL76F8b5JKzDREf8EmX
↪'
>>> k = HDKey.from_wif(wif)
>>> k.subkey_for_path("m/44'/0'/0'/0/2")
<HDKey(public_
↪hex=03004331ca7f0dcdd925abc4d0800a0d4a0562a02c257fa39185c55abdfc4f0c0c, wif_
↪public=xpub6GyQoEbMUNwu1LnbiCSaD8wLrcjyRCEQA8tNsFCH4pnvCbuWSZkSB6LUNe89YsCBTg1Ncs7vHJBjMvw2Q
↪ 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 the current key

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

↪'zpub6jftahH18ngZw8pNbq6arfqFD7przPySpW3jhhzQiBKYpzJxqwhBEpCk8rh9p7JQ5JrAMu1Pfq1wCXfDfZL8i9z
↪'
```

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 the 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('cNUpWJbC1hVJtyxyV4bVAnb4uJ7FPhr82geo1vnoA29XWkeiiCQn')
>>> k.secret
121272277086107546203375539852452923964442161118036950284195449442134423903 63
```

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

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

Parameters

- **import_key** (*str, int, bytes, tuple*) – 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 the 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_bytes(*private=False*)

Return bytes representation of private or public key

Parameters

private – Private or public key

Return bytes

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_hex(*private=False*)

Return hex representation of private or public key

Parameters

private – Private or public key

Return str

as_json(*include_private=False*)

Get current key as json formatted string

Parameters

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

Return str

encrypt(*password*)

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

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

Parameters

password (*str*) – Required password for encryption

Return str
BIP38 password encrypted private key

static from_wif(*wif*, *network=None*)

Import private key in WIF format.

Parameters

- **wif** (*str*) – Private key in WIF format
- **network** (*str, Network*) – Bitcoin, testnet, litecoin or other network

Return Key

property hash160

Get the public key in RIPEMD-160 + SHA256 format

Return bytes

hex()

info()

Prints key information to standard output

inverse()

Return inverse of private or public key

Return Key

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

property public_uncompressed_byte

property public_uncompressed_hex

sign_message(*message*, *use_rfc6979=True*, *k=None*, *hash_type=1*, *force_canonical=False*)

Create a Signature for the provided message. Provide the message in bytes format, this method will add network specific data and will hash the message. By default, a deterministic k value will be used according to the RFC6979 standard.

Parameters

- **message** (*bytes, hexstring*) – Message to be signed. Must be unhashed and in bytes format.
- **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 know what you are doing. Providing wrong value for k can result in leaking your private key!
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL

- **force_canonical** (*bool*) – Some wallets do not require a canonical s value, so you could set this to False

Return Signature

verify_message(*message*, *signature*)

Verify if provided message is signed by signature.

Parameters

- **message** (*bytes, hexstring*) – Message to verify. Must be unhashed and in bytes format.
- **signature** (Signature) – signature as Signature object

Return bool

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*, *message=None*, *secret=None*, *signature=None*, *der_signature=None*, *public_key=None*, *k=None*, *hash_type=1*, *compressed=True*, *recid=0*, *witness_type=None*, *network=None*)

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()

↪'3045022100ae1aa67bbb68dcf960d088681fcaebed55abbe63a2e7154899ffd2cbc3c45d0302206f4267d4e67510bb34
↪'
```

```
>>> signature2 = b
↪'IKF5khz0uiaTwl2LMnFC4ZyLHsqbYTVLMz7o1F5CN4j5Y0RLjLx8fPIYBcs1fulUl2NKnaE92QbP/w/
↪1NGymqFo='
>>> sig = Signature.parse_base64(signature2, sk)
>>> sig.r
730371655523619882547045009609420393757990827061352323347191398100585451051 45
```

Initialize a Signature object with provided r and r value

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

↪'48e994862e2cdb372149bad9d9894cf3a5562b4565035943efe0acc502769d351cb88752b5fe8d70d85f3541046df617
↪'
```

Parameters

- **r** (*int*) – r value of signature
- **s** (*int*) – s value of signature
- **message** (*bytes, hexstring*) – Transaction hash or message 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
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL
- **compressed** (*bool*) – Compressed or uncompressed key. Used for message signatures
- **recid** (*int*) – Recovery ID to indicate which of the 4 possible public points x,y combinations will be used. Used for message signatures
- **network** (*str*, Network) – Network to use

as_base64()

Return the current Signature in base64 format following the bip-0137 standard, used for message signing.

Uses this format: [1 byte of header data][32 bytes for r value][32 bytes for s value]

The header byte contains information about the type of address.

as_bytes()

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

as_hex()

as_signed_message(*message*)

Return signed message string to exchange with other entities.

Output example: —–BEGIN BITCOIN SIGNED MESSAGE—– Bitcoin Signed Message —–BEGIN SIGNATURE—– bc1qfpv6fhe8vqwrq3ag853u5m6l635jy8dku9ysak H/vw2bYHTyrQrRKo+RgahP8Y3L992q6YDJZipAXYV/s6LNN9XgV7ZOpYUnhAs+W6EpLRSzk4iDA6Xfx5qj6bDM0= —–END BITCOIN SIGNED MESSAGE—–

Parameters

message (*str*) – Provide original signed message. Signature object does not store message string

Return str

bytes()

Signature r and s value as single bytes string

Return bytes

static create(*message*, *private*, *use_rfc6979=True*, *k=None*, *hash_type=1*, *prehashed=True*, *force_canonical=False*, *network='bitcoin'*)

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

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

↪'21934dde1f8b62d73359991c4ebe043cc7758118611455b97e7d8c5fd4ae99805f8cf0507436f94062635ad5ccb8
↪'
>>> sig.r
151865879446690974494786022512081880952315030549643452090489736852290455863 04
>>> sig.s
725733514446791109378068894241910621553637000897995687813626653666969746392 32
```

Parameters

- **message** (*bytes, str*) – Transaction signature or transaction hash (txid). 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 know what you are doing. Providing wrong value for k can result in leaking your private key!
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL
- **prehashed** (*bool*) – Wheter the message / txid provided was already prehashed with the Double SHA 256 for instance. Default is True if message size is 32 else False, if set to False the message will be hashed with double_sha256 by this create method.
- **force_canonical** (*bool*) – Calculate signature with canonical s value. Default is True, needed for valid transacton signatures. Only set to False for signing messages.

Return Signature

hex()

Signature r and s value as single hexadecimal string

Return hexstring

property message

classmethod parse(*signature*, *public_key=None*)

static parse_base64(*signature*, *public_key=None*)

Parse base64 encoded signature and return a Signature object.

Extract compressed and recovery ID info from the first byte.

Parameters

- **signature** (*str, bytes*) – Base64 encoded signature
- **public_key** (HDKey, Key, *str, hexstring, bytes*) – Public key to pass to signature object

Return Signature

static parse_bytes(*signature*, *public_key=None*)

Create a signature from signature string with r and s part. Signature should be a DER encoded byte string of a 64 bytes string with an r and s value

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

verify(*message=None*, *public_key=None*)

Verify this signature. Provide a message/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

- **message** (*bytes, hexstring*) – Message to verify, for instance a Transaction hash
- **public_key** (HDKey, Key, *str, hexstring, bytes*) – Public key P

Return bool

verify_message(*message*, *address=None*, *network=None*)

Verify if the provided message is signed with this signature. Provide address to check if address matches the signature.

Parameters

- **message** (*bytes, hexstring*) – Message to verify. Must be unhashed and in bytes format.
- **address** (*Address, str*) – Address used to sign message
- **network** (*Network, str*) – Network used to sign message

Return bool

bitcoinlib.keys.**addr_convert**(*addr*, *prefix*, *encoding=None*, *to_encoding=None*)

Convert address to another encoding and/or address with another prefix.

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

Parameters

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

Return str

New converted address

bitcoinlib.keys.**bip38_create_new_encrypted_wif**(*intermediate_passphrase*, *compressed=True*, *seed=b'yN\xeb\xd6?4\xbe\xf1\xa8\x11B\xdc\xa2\xd0\xf9i\xb2\xcdD\xdeH*
network='bitcoin')

Create new encrypted WIF BIP38 EC multiplied key. Use *bip38_intermediate_password()* to create an intermediate passphrase first.

Parameters

- **intermediate_passphrase** (*str*) – Intermediate passphrase text
- **compressed** (*boolean*) – Compressed or uncompressed key
- **seed** (*str, bytes*) – Seed, default to `os.urandom(24)`
- **network** (*str*) – Network name

Returns dict

Dictionary with encrypted WIF key and confirmation code

bitcoinlib.keys.**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, boolean, dict)
(Private Key bytes, 4 byte address hash for verification, compressed?, dictionary with additional info)

bitcoinlib.keys.**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.keys.**bip38_intermediate_password**(*passphrase*, *lot=None*, *sequence=None*, *owner_salt=b'o\xea\xa8\xce\x9eAa'*)

Intermediate passphrase generator for EC multiplied BIP38 encrypted private keys. Source: https://github.com/meherett/python-bip38/blob/master/bip38/bip38.py

Use intermediate password to create an encrypted WIF key with the *bip38_create_new_encrypted_wif()* method.

Parameters

- **passphrase** (*str*) – Passphrase or password text
- **lot** (*int*) – Lot number between 100000 <= lot <= 999999 range, default to None
- **sequence** (*int*) – Sequence number between 0 <= sequence <= 4095 range, default to None
- **owner_salt** (*str, bytes*) – Owner salt, default to os.urandom(8)

Returns str
Intermediate passphrase

```
>>> bip38_intermediate_password(passphrase="TestingOneTwoThree", lot=199999,␣
↪sequence=1, owner_salt="75ed1cdeb254cb38")
'passphraseb7ruSN4At4Rb8hPTNcAVezfsjonvUs4Qo3xSp1fBFsFPvVGSbpP2WTJMhw3mVZ'
```

bitcoinlib.keys.**check_network_and_key**(*key*, *network=None*, *kf_networks=None*, *default_network='bitcoin'*)

Check if a given key corresponds with the given network and return network if it does. If no network is specified, this method tries to extract the network from the key. If no network can be extracted from the key, the default network will be returned.

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

A BKeyError will be raised if the key does not correspond with the network or if multiple networks 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 belong 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 the highest priority if multiple networks are found. Same applies for the script type.

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

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

```
>>> deserialize_address('1Khyc5eUddbhYZ8bEZi9wiN8TrmQ8uND4j')
{'address': '1Khyc5eUddbhYZ8bEZi9wiN8TrmQ8uND4j', 'encoding': 'base58', 'public_key_
↪hash': 'cd322766c02e7c37c3e3f9b825cd41ffbdcd17d7', 'public_key_hash_bytes': b"\
↪xcd2'f\xc0.|7\xc3\xe3\xf9\xb8%\xcdA\xff\xbd\xcd\x17\xd7", 'prefix': b'\x00',
↪'network': 'bitcoin', 'script_type': 'p2pkh', 'witness_type': 'legacy', 'networks
↪': ['bitcoin', 'regtest'], 'checksum': b'\xcf\xa4I0', 'witver': None, 'raw': b"\
↪x00\xcd2'f\xc0.|7\xc3\xe3\xf9\xb8%\xcdA\xff\xbd\xcd\x17\xd7\xcf\xa4I0"}
```

Parameters

- **address** (*str*) – A base58 or bech32 encoded address
- **encoding** (*str*) – Encoding scheme used for address encoding. Attempts to guess encoding if not specified.
- **network** (*str*) – Specify network filter, i.e.: bitcoin, testnet, litecoin, etc. Wil trigger check if address is valid for this network

Return dict

with information about this address

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

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

Parameters

m (*int*) – A scalar multiplier

Return Point

Generator point G multiplied by m

bitcoinlib.keys.**ec_point_multiplication**(*p*, *m*, *return_point=False*)

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

Parameters

- **p** (*tuple*) – Point on SECP256k1 curve

- **m** (*int*) – A scalar multiplier

Return tuple

Generator point G multiplied by m as tuple in (x, y) format

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', 'regtest'], 'is_private': True,
↪ 'script_types': [], 'witness_types': ['segwit'], 'multisig': [False]}
```

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

```
>>> get_key_format(
↪'Zpub6vZyhw1ShkEwNxtqfjk7jiwoEbZYMJdbWLHvEwo6Ns2fFc9rdQn3SerYFQXYxtZYbA8a1d83shW3g4WbsnVsymy2L8m7
↪')
{'format': 'hdkey_public', 'networks': ['bitcoin', 'regtest'], 'is_private': False,
↪'script_types': ['p2wsh'], 'witness_types': ['segwit'], 'multisig': [True]}
```

Parameters

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

Return dict

Dictionary with format, network and is_private

bitcoinlib.keys.**message_magic**(*message*, *network=None*)

Add network magic to a message string, so “Hello world!” results in “BITCOIN Signed Message: Hello world!

Parameters

- **message** (*str*) – Message text
- **network** (*Network, str*) – Network to use, default is Bitcoin

Return str

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 a 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=84*, *address_index=0*, *change=0*, *witness_type='segwit'*, *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**(*message*, *private*, *use_rfc6979=True*, *k=None*, *hash_type=1*, *prehashed=True*, *force_canonical=True*, *network='bitcoin'*)

Sign the transaction hash or message with the 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()

↪'3044022039df9d8a0b4df185605c5a46eb087d499ddf98cb5ebcae6e0fd99c56152d2d730220726a3c03ac0b04e1489a
↪'
```

Parameters

- **message** (*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 know what you are doing. Providing wrong value for k can result in leaking your private key!
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL

- **prehashed** (*bool*) – Whether the message / txid provided was already prehashed with the Double SHA 256 for instance. Default is True, if set to False the message will be hashed with double_sha256 by this create method.
- **force_canonical** (*bool*) – Force canonicalization of signature s value. Default is True
- **network** (*str*, Network) – Specific network, default is DEFAULT_NETWORK

Return Signature

bitcoinlib.keys.**signed_message_parse**(*message_signature*)

Parse Bitcoin Signed Message and split into message, address and signature part.

Example: python> signed_message_parse(' —–BEGIN BITCOIN SIGNED MESSAGE—–

Bitcoinlib is cool! —–BEGIN SIGNATURE—– bc1qed0dq6a7gshfvap4j946u44kk73gs3a0d5p3sw ILtL9qkUb+2nfxY3bUqfoWsVSwhMSos+DVY7p3EqmzQ6qF2gHNPvILwrsZ2AKlIqPmJjln4OKpW+d86wBn27yJw= —–END BITCOIN SIGNED MESSAGE—–")

Result: ('Bitcoinlib is cool!', 'bc1qed0dq6a7gshfvap4j946u44kk73gs3a0d5p3sw', 'ILtL9qkUb+2nfxY3bUqfoWsVSwhMSos+DVY7p3EqmzQ6qF2gHNPvILwrsZ2AKlIqPmJjln4OKpW+d86wBn27yJw=', 'bitcoin')

Parameters
message_signature (*str*) – Signed Message text

Return tuple (str, str, str, str)
message, base64 encoded signature, address

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

Verify the provided signature with the message / txid. 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 =
↪'48e994862e2cdb372149bad9d9894cf3a5562b4565035943efe0acc502769d351cb88752b5fe8d70d85f3541046df617
↪'
>>> verify(txid, sig, pub_key)
True
```

Parameters

- **message** (*bytes*, *hexstring*) – Message / Transaction hash
- **signature** (Signature, *str*, *bytes*) – signature as Signature, 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

bitcoinlib.keys.**verify_message**(*message*, *sig*, *address*, *network=None*)

Verify a signed message. Prove if the owner of the provided address did sign this exact message.

```
>>> message = 'this is a unittest'
>>> sig = 'IBqszVUdhXkJnZpcoQh+EyEZicOQP0fZ2z/
↪Rzw4Xa+YgCXtsOYSR1RvAYAeejCkEW6iQZ9e5j8AqSH2zxvshZPM='
```

(continues on next page)

(continued from previous page)

```
>>> addr = 'bc1q7wdttvkuypq3p2ww7cfzgzzs659jehhgvsfnnn'
>>> verify_message(message, sig, addr)
True
```

Parameters

- **message** (*str*) – Message text
- **sig** (*str, Signature*) – Signature
- **address** (*str, Address*) – Address to verify message
- **network** (*Network, str*) – Network to use, default is Bitcoin

Return bool

8.15 bitcoinlib.transactions module

class bitcoinlib.transactions.**Input**(*prev_txid*, *output_n*, *keys=None*, *signatures=None*, *public_hash=b''*, *unlocking_script=b''*, *locking_script=None*, *redeemscript=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 spend 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, HDKey)*) – 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
- **locking_script** (*bytes, hexstring*) – Unlocking script for signing transaction
- **redeemscript** (*bytes, hexstring*) – Redeem script for p2sh transaction. Will be automatically created for standard scripts
- **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, used for locktime setting and replace by fee. No need to set directly normally.
- **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 the smallest denominator integers (Satoshi's) or as Value object or string
- **double_spend** (*bool*) – Is this input also spend in another transaction
- **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, bytes*) – List of witnesses for inputs, used for segwit transactions for instance. Argument can be list of bytes or string or a single bytes string with concatenated witnesses as found in a raw transaction.
- **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 an 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

verify(*transaction_hash*)

Verify input with provided transaction hash, check if signatures matches public key.

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

Parameters
transaction_hash (*bytes*) – Double SHA256 Hash of Transaction signature

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

class bitcoinlib.transactions.**Output**(*value*, *address=''*, *public_hash=b''*, *public_key=b''*, *lock_script=b''*, *spent=False*, *output_n=0*, *script_type=None*, *witver=0*, *encoding=None*, *spending_txid=''*, *spending_index_n=None*, *strict=True*, *change=None*, *witness_type=None*, *network='bitcoin'*)

Bases: object

Transaction Output class, normally part of Transaction class.

Contains the amount and destination of a transaction.

Create a new transaction output

A 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 they all can be derived from each other, but you can provide as many attributes as you know to improve speed.

Parameters

- **value** (*int*, Value, *str*) – Amount of output in the 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.
- **witver** (*int*) – Witness version
- **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
- **change** (*bool*) – Is this a change output back to own wallet or not? Used for replace-by-fee.
- **witness_type** (*str*) – Specify witness type: 'segwit' or 'legacy'. Determine from script, address or encoding if not specified.
- **network** (*str, Network*) – Network, leave empty for default

property address

property address_obj

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

Return Address

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 a 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 a 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='segwit'*, *flag=None*, *replace_by_fee=False*, *index=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 added 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 an 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 the smallest denominator (ie Satoshi) for complete transaction

- **fee_per_kb** (*int*) – Fee in the 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
- **index** (*int*) – Index of transaction in block. Used when parsing blocks

add_input(*prev_txid*, *output_n*, *keys=None*, *signatures=None*, *public_hash=b''*, *unlocking_script=b''*, *locking_script=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** (*list (bytes, str, Key, HDKey)*) – 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

- **locking_script** (*bytes, hexstring*) – Locking script (scriptPubKey) of previous output if known
- **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 Locktime Verify value.
- **locktime_csv** (*int*) – Check Sequency Verify value.
- **key_path** (*str, list*) – Key path of input key as BIP32 string or list
- **witness_type** (*str*) – Specify witness/signature position: 'segwit' or 'legacy'. Determine from script, address or encoding if not specified.
- **witnesses** (*list of bytes, list of 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*, *change=None*)

Add an output to this transaction

Wrapper for the append method of the Output class.

Parameters

- **value** (*int*) – Value of output in the 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
- **change** (*bool*) – Is this a change output back to own wallet or not? Used for replace-by-fee.

Return int

Transaction output number (output_n)

as_bytes()

Return raw serialized transaction as bytes string

Return bytes

as_dict()

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

Return dict

as_hex()

Return raw hex string of transaction as hex string

Returns

as_json()

Get current transaction as json formatted string

Return str

bumpfee(*fee=0*, *extra_fee=0*)

Increase fee for this transaction. If replace-by-fee is signaled in this transaction the fee can be increased to speed up inclusion on the blockchain.

If not fee or extra_fee is provided the extra fee will be increased by the formule you can find in the code below using the BUMPFEE_DEFAULT_MULTIPLIER from the config settings.

The extra fee will be deducted from change output. This method fails if there are not enough change outputs to cover fees.

Parameters

- **fee** (*int*) – New fee for this transaction
- **extra_fee** (*int*) – Extra fee to add to current transaction fee

Return None

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 the smallest denominator (i.e. Satoshi) based on its size and the transaction.fee_per_kb value

Return int
Estimated transaction fee

`estimate_size`(*number_of_change_outputs=0*)

Get estimated vsize in for current transaction based on transaction type and number of inputs and outputs.

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

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

Return int
Estimated transaction size

`info`()

Prints transaction information to standard output

`static load`(*txid=None*, *filename=None*)

Load transaction object from file which has been stored with the *save()* method.

Specify transaction ID or filename.

Parameters

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

Return Transaction

`merge_transaction`(*transaction*)

Merge this transaction with provided Transaction object.

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

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

Parameters
transaction (Transaction) – The transaction to be merged

`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'*, *index=None*, *raw_bytes=b''*)

Parse a raw transaction and create a Transaction object

Parameters

- **rawtx** (*BytesIO*) – Raw transaction bytes stream
- **strict** (*bool*) – Raise exception when transaction is malformed, incomplete or not understood
- **network** (*str, Network*) – Network, leave empty for default network
- **index** (*int*) – index position in block
- **raw_bytes** – Raw transaction as bytes if available
- **raw_bytes** – bytes

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.

You can also pass the locktime value directly to a Transaction object, or when sending from a wallet.

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*, *locktime=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 a 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
- **locktime** (*int*) – Overwrite default locktime, must be lower than current network block-count. If anti-fee-sniping is used in a Wallet this value is already filled in.

Returns

set_locktime_relative_time(*seconds*, *input_index_n=0*, *locktime=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 a 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
- **locktime** (*int*) – Overwrite default locktime, must be lower than current network block-count. If anti-fee-sniping is used in a Wallet this value is already filled in.

Returns

set_locktime_time(*timestamp*)

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

You can also pass the locktime value directly to a Transaction object, or when sending from a wallet.

Parameters

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

Returns

shuffle()

Shuffle transaction inputs and outputs in random order.

Returns

shuffle_inputs()

Shuffle transaction inputs in random order.

Returns

shuffle_outputs()

Shuffle transaction outputs in random order.

Returns

sign(*keys=None*, *index_n=None*, *multisig_key_n=None*, *hash_type=1*, *fail_on_unknown_key=True*, *replace_signatures=False*)

Sign the transaction input with provided private key

Parameters

- **keys** (HDKey, Key, *bytes*, *list*) – A private key or list of private keys
- **index_n** (*int*) – Index of transaction input. Leave empty to sign all inputs
- **multisig_key_n** (*int*) – Index number of key for multisig input for segwit transactions. Leave empty if not known. If not specified all possibilities will be checked
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL
- **fail_on_unknown_key** (*bool*) – Method fails if public key from signature is not found in public key list
- **replace_signatures** (*bool*) – Replace signature with new one if already signed.

Return None

sign_and_update(*index_n=None*)

Update transaction ID and resign. Use if some properties of the transaction changed

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

Returns

signature(*sign_id=None*, *hash_type=1*, *witness_type=None*)

Serializes transaction and calculates signature for Legacy or Segwit transactions

Parameters

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

Return bytes
Transaction signature

signature_hash(*sign_id=None*, *hash_type=1*, *witness_type=None*, *as_hex=False*)

Double SHA256 Hash of Transaction signature

Parameters

- **sign_id** (*int*) – Index of input to sign
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL
- **witness_type** (*str*) – Legacy or Segwit witness type? Leave empty to use Transaction witness type
- **as_hex** (*bool*) – Return value as a 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_inputs(*input_n=None*)

Update input scripts to reflect changes you made to one of more inputs. All inputs will be updated unless you specificy a specific input.

Parameters
input_n – Input to update, leave empty to update all input scripts

Returns

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 a locking script type and get a 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.**transaction_update_spents**(*txs*, *address*)

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

This method loops through all the transactions and updates the transaction outputs for the supplied address, checks if the output is spent and adds the spending transaction ID and index number to the outputs.

The same list of transactions with all updates outputs will be returned

Parameters

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

Return list of Transaction

8.16 bitcoinlib.scripts module

class bitcoinlib.scripts.**Script**(*commands=None*, *message=None*, *script_types=''*, *is_locking=True*, *keys=None*, *signatures=None*, *blueprint=None*, *env_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_2, op_4, 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 []

```
>>> key1 = '5JruagvxNLXTnkksyLMfgFgf3CagJ3Ekxu5oGxpTm5mPfTAPez3'
>>> key2 = '5JX3qAwDEEaapvLXRfbXRMSiyRgRSW9WjgxeyJQWwBugbudCwsk'
>>> key3 = '5JjHVMwJdjPEPQhq34WMUhzLcEd4SD7HgZktEh8WHstWcCLRceV'
>>> keylist = [Key(k) for k in [key1, key2, key3]]
>>> redeemscript = Script(keys=keylist, sigs_required=2, script_types=['multisig'])
```

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
- **env_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, Script*) – Provide redeemscript to create a new (multisig) script
- **hash_type** (*int*) – Specific script hash type, default is SIGHASH_ALL

as_bytes()

as_hex()

property blueprint

evaluate(*message=None*, *env_data=None*, *trace=False*)

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_2, op_4, 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.
- **env_data** (*dict()*) – Dictionary with extra information needed to verify script. Such as 'redeemscript' for
- **trace** (*bool*) – Write trace information to stdout

multisignature scripts and 'blockcount' for time locked scripts. Leave emtpy to use Script.data. If supplied Script.data will be ignored

Return bool

Valid or not valid

classmethod parse(*script*, *message=None*, *env_data=None*, *is_locking=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('76a914af8e14a2cecd715c363b3a72b55b59a31e2acac988ac')
<Script([op_dup, op_hash160, data-20, op_equalverify, 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
- **env_data** (*dict*) – Dictionary with extra information needed to verify script. Such as 'redeemscript' for multisignature scripts and 'blockcount' for time locked scripts
- **is_locking** – Is this a locking script or not, use None if not known and derive from script.

- **is_locking** – bool, None
- **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*, *env_data=None*, *is_locking=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
- **env_data** (*dict*) – Dictionary with extra information needed to verify script. Such as 'redeemscript' for multisignature scripts and 'blockcount' for time locked scripts
- **is_locking** – Is this a locking script or not, use None if not known and derive from script.
- **is_locking** – bool, None
- **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*, *env_data=None*, *data_length=0*, *is_locking=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
- **env_data** (*dict*) – Dictionary with extra information needed to verify script. Such as 'redeemscript' for multisignature scripts and 'blockcount' for time locked scripts
- **data_length** (*int*) – Length of script data if known. Supply if you can to increase efficiency and lower change of incorrect parsing
- **is_locking** – Is this a locking script or not, use None if not known and derive from script.
- **is_locking** – bool, None
- **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*, *env_data=None*, *is_locking=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_dup, op_hash160, data-20, op_equalverify, op_checksig])>
```

Parameters

- **script** (*str*) – Raw script to parse in hexadecimal string format
- **message** (*bytes*) – Signed message to verify, normally a transaction hash
- **env_data** (*dict*) – Dictionary with extra information needed to verify script. Such as 'redeemscript' for multisignature scripts and 'blockcount' for time locked scripts
- **is_locking** – Is this a locking script or not, use None if not known and derive from script.
- **is_locking** – bool, None
- **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_str(*script*, *message=None*, *env_data=None*, *is_locking=None*, *strict=True*, *_level=0*)

Parse script in string format and return Script object. Extracts script commands, keys, signatures and other data.

```
>>> s = Script.parse_str("1 98 OP_ADD 99 OP_EQUAL")
>>> s
<Script([data-1, data-1, op_add, data-1, op_equal])>
>>> s.evaluate()
True
```

Parameters

- **script** (*str*) – Raw script to parse in bytes format
- **message** (*bytes*) – Signed message to verify, normally a transaction hash
- **env_data** (*dict*) – Dictionary with extra information needed to verify script. Such as 'redeemscript' for multisignature scripts and 'blockcount' for time locked scripts
- **is_locking** – Is this a locking script or not, use None if not known and derive from script.
- **is_locking** – bool, None
- **strict** (*bool*) – Raise exception when script is malformed or incomplete
- **_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

view(*blueprint=False*, *as_list=False*, *op_code_numbers=False*, *show_1_byte_data_as_int=True*)

View script as string in human-readable format.

Parameters

- **blueprint** (*bool*) – Show blueprint only, without detailed data.
- **as_list** (*bool*) – Show script as list
- **op_code_numbers** (*bool*) – Show opcodes as numbers instead of string.
- **show_1_byte_data_as_int** (*bool*) – Show 1 byte data objects as integers.

Return str

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

Bases: Exception

Handle Key class Exceptions

class bitcoinlib.scripts.**Stack**(*iterable=()*, */ (Positional-only parameter separator (PEP 570))*)

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 stock 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()

Add the top 2 numbers of the stack and appends the result on the top of the stack.

Fails if the top 2 items are not arithmetic or if there are not enough items on the stack.

Return bool
Operation succeeded

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 verion. 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_numgreaterthanorequal()

op_numlessthan()

op_numlessthanorequal()

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()

pop_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')
1000000
```

Parameters
encoded (*bytes*) – Number to decode

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

8.17 bitcoinlib.wallets module

class bitcoinlib.wallets.**Wallet**(*wallet*, *db_uri=None*, *db_cache_uri=None*, *session=None*, *main_key_object=None*, *db_password=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 many child keys as you want in a structured manner.

You can import keys in many formats 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 the provided ID or name

Parameters

- **wallet** (*int, str*) – Wallet name or ID
- **db_uri** (*str*) – URI of the database
- **db_cache_uri** (*str*) – URI of the cache database. If not specified the default cache database is used when using sqlite, for other databasetypes the cache database is merged with the wallet database (db_uri)
- **session** (*sqlalchemy.orm.session.Session*) – Sqlalchemy session
- **main_key_object** (HDKey) – Pass the main key object to save time

account(*account_id*)

Returns wallet key of the specified 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=None*)

Get a list of accounts for this wallet

Parameters
network (*str*) – The network name filter. Default filter is the network of first main key

Return list of integers
List of account IDs

address_index(*address_index*, *account_id=None*, *cosigner_id=None*, *change=0*, *network=None*)

Get the key with the specified address_index from this wallet. Always returns a key with the specified path, even if it is not created yet in wallet. Wrapper for the *key_for_path()* method.

To get an unused key / address use the *get_key()* method and to create a new key use the *new_key()* method.

Parameters

- **address_index** (*int*) – address_index of the specific key
- **account_id** (*int*) – Account ID
- **cosigner_id** (*int*) – ID of cosigner
- **change** (*int*) – Change key = 1 or normal = 0, normally provided to 'path' argument
- **network** (*str*) – The network name. Leave empty for default network

Return WalletKey

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

Get a list of addresses defined in the 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]
'16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg'
```

Parameters

- **account_id** (*int*) – Account ID
- **used** (*bool, None*) – Only return used or unused keys
- **network** (*str*) – The 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 an 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 the current key as a 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 the total balance of the unspent outputs for this wallet

Parameters

- **account_id** (*int*) – Account ID filter
- **network** (*str*) – The 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 the total wallet balance in the 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*) – The 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*, *anti_fee_sniping=True*, *strict=True*, *ignore_dust=True*, *db_uri=None*, *db_cache_uri=None*, *db_password=None*)

Create Wallet and insert in the database. Generate a masterkey or imports a key when specified.

When only a name is specified, a 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")>
```

To create a multi-signature wallet, specify multiple keys (private or public) and provide the sigs_required argument if it is different than 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")>
>>> # Add some test utxo data:
>>> if w.utxo_add('16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg', 100000000,
↪'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*, HDKey, *HDWalletKey*, *list of str*, *list of bytes*, *list of int*, *list of* HDKey, *list of HDWalletKey*) – 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*) – The 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 a witness type, default is 'segwit', for native segregated witness wallet. Use 'legacy' for an old-style wallets or 'p2sh-segwit' for legacy compatible wallets
- **encoding** (*str*) – Encoding used for address generation: base58 or bech32. Default is derive from the wallet and/or witness type
- **multisig** (*bool*) – Multisig wallet or child of a multisig wallet, default is None / derive from number of keys.
- **sigs_required** (*int*) – Number of signatures required for validation if using a multisignature wallet. For example 2 for 2-of-3 multisignature. Default is all keys must be signed
- **cosigner_id** (*int*) – Set this if wallet contains only public keys, more than 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 wallet's cosigner_id may be different.
- **key_path** (*list, str*) – Key path for a multisig wallet, use to create your own non-standard key path. Key path must follow the following rules: * Path starts with masterkey (m) and ends 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 the path is 8 levels
- **anti_fee_sniping** (*boolean*) – Set default locktime in transactions as current block height + 1 to avoid fee-sniping. Default is True, which will make the network more secure. You could disable it to avoid transaction fingerprinting.
- **strict** (*boolean*) – Set to False, to ignore non-standard signatures or script. Can be usefull for blockchain parsing or external transactions
- **ignore_dust** (*boolean*) – Ignore dust outputs in unspent transaction outputs. The dust output amount is defined in the network settings. Default is True.
- **db_uri** (*str*) – URI of the database for wallets, wallet transactions and keys
- **db_cache_uri** (*str*) – URI of the cache database. If not specified, the default cache database is used when using sqlite, for other databasetypes the cache database is merged with the wallet database (db_uri)
- **db_password** – Password to encrypt the database. Requires the installation of sqlcipher (see

documentation). :type db_password: str

Return Wallet

property default_account_id

default_network_set(*network*)

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

Get an 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 keys 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/84'/0'/0'/0/...)>
```

Parameters

- **account_id** (*int*) – Account ID. Default is the last used or created account ID.
- **witness_type** (*str*) – Use to create a key with a specific witness_type
- **network** (*str*) – The network name. Leave empty for the 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*, *witness_type=None*, *network=None*)

Get an 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 the last used or created account ID.
- **witness_type** (*str*) – Use to create a key with the specific witness_type
- **network** (*str*) – The network name. Leave empty for default network

Return WalletKey

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

Get a list of unused keys / addresses or create 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 the last used or created account ID.
- **witness_type** (*str*) – Use to create a key with the specific witness_type
- **network** (*str*) – The 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*, *witness_type=None*, *network=None*, *number_of_keys=1*)

Get an 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 the last used or created account ID.
- **witness_type** (*str*) – Use to create a key with the specific witness_type
- **network** (*str*) – The 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=84*, *key_type=None*)

Add a new single key to the wallet.

Parameters

- **key** (*str, bytes, int,* HDKey, Address) – Key to import
- **account_id** (*int*) – Account ID. Default is the last used or created account ID.
- **name** (*str*) – Specify name for this key, leave empty for default
- **network** (*str*) – The network name, method will try to extract from the key if not specified. Raises warning if network could not be detected
- **purpose** (*int*) – BIP44 definition used, default is 84 (segwit)
- **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*, *witness_type=None*, *network=None*, *recreate=False*)

Wrapper for the keys_for_path method. Returns a single wallet key.

Parameters

- **path** (*list, str*) – Part of the key path, i.e. [0, 0] for [change=0, address_index=0]
- **level_offset** (*int*) – Just create part of the path, when creating keys. For example -2 means create a 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
- **witness_type** (*str*) – Use to create key with different witness_type
- **network** (*str*) – The 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*, *witness_type=None*, *network=None*, *include_private=False*, *as_dict=False*)

Search for keys in the 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)
[<WalletKey(id=..., name='bitcoinlib_legacy_wallet_test', wif=
↪'xprv9s21ZrQH143K3cxbMVswDTYgAc9CeXABQjCD9zmXCpXw4MxN93LanEARbBmV3utHZS9Db4FX1C1RbC5KSNAjQ5W
↪'>]
```

Returns a list of WalletKey objects 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)
- **witness_type** (*str*) – Filter by witness_type
- **network** (*str*) – The network name filter
- **include_private** (*bool*) – Include private key information in dictionary
- **as_dict** (*bool*) – Return keys as dictionary objects. Default is False: WalletKey objects

Return list of WalletKey or list of dict
List of keys from this wallet

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

Get Database records of account key(s) with for the 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 this wallet, 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*) – The network name filter
- **as_dict** (*bool*) – Return as dictionary or WalletKey object. Default is False: WalletKey objects

Return list of (WalletKey, dict)

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

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

Parameters

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

:return list of WalletKey, list of dict

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

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

Parameters

- **account_id** (*int*) – Account ID

- **used** (*bool*) – Only return used or unused keys
- **network** (*str*) – The network name filter
- **as_dict** (*bool*) – Return as dictionary or WalletKey object. Default is False: WalletKey objects

:return list of WalletKey, list of 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 the 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*) – The network name filter
- **depth** (*int*) – Filter by key depth. Default for BIP44 and multisig is 5
- **as_dict** (*bool*) – Return as dictionary or WalletKey object. Default is False: WalletKey objects

:return list of WalletKey, list of dict

keys_for_path(*path*, *level_offset=None*, *name=None*, *account_id=None*, *cosigner_id=None*, *address_index=0*, *change=0*, *witness_type=None*, *network=None*, *recreate=False*, *number_of_keys=1*)

Return the key for the specified path. Derive all wallet keys in a path if they do not already exist

```
>>> w = wallet_create_or_open('key_for_path_example')
>>> key = w.key_for_path([0, 0])
>>> key.path
"m/84'/0'/0'/0/0"
```

```
>>> w.key_for_path([], level_offset=-2).path
"m/84'/0'/0'"
```

```
>>> w.key_for_path([], w.depth_public_master + 1).path
"m/84'/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/84'/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
- **witness_type** (*str*) – Use to create key with different witness_type
- **network** (*str*) – The 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
- **number_of_keys** (*int*) – Number of keys to create, use to create keys in bulk fast

Return list of WalletKey

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()
>>> network_key[0].path
"m/44'/0'"
```

Parameters

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

Return list of WalletKey, list of dict

last_address_index(*account_id=None*, *cosigner_id=0*, *change=0*, *network=None*)

Get last used address_index for this wallet

Parameters

- **account_id** (*int*) – Account ID
- **cosigner_id** (*int*) – ID of cosigner
- **change** (*int*) – Change key = 1 or normal = 0, normally provided to 'path' argument
- **network** (*str*) – The network name. Leave empty for default network

Return int

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 a 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*, *witness_type=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 this wallet contains a masterkey.

Parameters

- **name** (*str*) – Account Name. If not specified, "Account #" with the account_id will be used as name
- **account_id** (*int*) – Account ID. Default is the last accounts ID + 1
- **witness_type** (*str*) – Use to create a key with the specific witness_type
- **network** (*str*) – The network name. Leave empty for default network

Return WalletKey

new_key(*name=''*, *account_id=None*, *change=0*, *cosigner_id=None*, *witness_type=None*, *network=None*)

def new_key(self, name='', account_id=None, change=0, cosigner_id=None, witness_type=None, network=None):

Parameters

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

Return WalletKey

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

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

Parameters

- **name** (*str*) – Key name. The default name is 'Change #' with an address index

- **account_id** (*int*) – Account ID. Default is the last used or created account ID.
- **witness_type** (*str*) – Use to create key with different witness_type
- **network** (*str*) – The network name. Leave empty for default network

Return WalletKey

new_keys(*name=''*, *account_id=None*, *change=0*, *cosigner_id=None*, *witness_type=None*, *number_of_keys=1*, *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/84'/0'/0'/0/...)>
```

Parameters

- **name** (*str*) – Key name. Does not have to be unique, but if you use it as a reference, you might choose to enforce this. If not specified 'Key #' with a unique sequence number will be used
- **account_id** (*int*) – Account ID. Default is the last used or created account ID.
- **change** (*int*) – Change (1) or payments (0). The default is 0
- **cosigner_id** (*int*) – Cosigner ID for key path
- **witness_type** (*str*) – Use to create a key with different witness_type
- **number_of_keys** (*int*) – Number of keys to generate. Use a positive integer
- **network** (*str*) – The network name. Leave empty for default network

Return list of 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 a key path. Specify part of the key path to expand to the key path used in this wallet.

```
>>> w = Wallet('create_legacy_wallet_test')
>>> w.path_expand([0,1200])
['m', "84'", "0'", "0'", '0', '1200']
```

```
>>> w = Wallet('create_legacy_multisig_wallet_test')
>>> w.path_expand([0,2], cosigner_id=1)
['m', "48'", "0'", "0'", "2'", '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*) – The network name. Leave empty for default network

Return list

public_master(*account_id=None*, *name=None*, *as_private=False*, *witness_type=None*, *network=None*)

Return the 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 returned as a list.

Returns private key information if available and as_private is True is specified

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

↪'xpub6D2qEr8Z8WYKKns2xZYyyvvRviPh1NKt1kfHwwfiTxJwj7peReEJt3iXoWWsr8tXWTsejDjMfAezM53KVFVkSZz
↪'
```

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
- **witness_type** (*str*) – Witness type, leave empty for default witness_type
- **network** (*str*) – The 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 the 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 the 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*) – The 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 the specified wallet key and update wallet transactions

Parameters

key (*WalletKey, int*) – The wallet key as an 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*, *ignore_dust=None*)

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 the smallest denominator (sathosi) to select
- **variance** (*int*) – Allowed difference in total input value. Default is dust amount of the selected network. Difference will be added to the 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*) – The 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
- **ignore_dust** (*bool*) – Do not include small amounts to avoid dust inputs. Default is to use the Wallet.ignore dust setting which is True by default.

Returns

List of selected unspent transaction outputs

Return type

list[*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*, *broadcast=False*, *number_of_change_outputs=1*, *random_output_order=True*, *replace_by_fee=False*)

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 the 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', 200000)])
>>> 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*) – The network name. Leave empty for default network
- **fee** (*int, str*) – Set fee manually, leave empty to calculate fees automatically. Set fees in the smallest currency denominator, for example satoshi's if you are using bitcoins. You can also supply a string: 'low', 'normal' or 'high' to determine fees automatically.
- **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
- **broadcast** (*bool*) – Just return the transaction object and do not send it when broadcast = False. 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
- **random_output_order** (*bool*) – Shuffle order of transaction outputs to increase privacy. Default is True

- **replace_by_fee** (*bool*) – Signal replace by fee and allow to send a new transaction with higher fees. Sets sequence value to SEQUENCE_REPLACE_BY_FEE

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*, *broadcast=False*, *number_of_change_outputs=1*, *random_output_order=True*, *replace_by_fee=False*)

Create a transaction and send it with default Service objects `services.sendrawtransaction()` method.

Wrapper for the wallet *send()* method.

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> t = w.send_to('1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb', 200000)
>>> 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 the 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*) – The network name. Leave empty for default network
- **fee** (*int*, *str*) – Set fee manually, leave empty to calculate fees automatically. Set fees in the smallest currency denominator, for example satoshi's if you are using bitcoins. You can also supply a string: 'low', 'normal' or 'high' to determine fees automatically.
- **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
- **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
- **broadcast** (*bool*) – Just return the transaction object and do not send it when broadcast = False. 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
- **random_output_order** (*bool*) – Shuffle order of transaction outputs to increase privacy. Default is True
- **replace_by_fee** (*bool*) – Signal replace by fee and allow to send a new transaction with higher fees. Sets sequence value to SEQUENCE_REPLACE_BY_FEE

Return WalletTransaction

property session

sign_message(*message*, *key_term=None*, *use_rfc6979=True*, *k=None*, *hash_type=1*, *force_canonical=False*)

Sign a message with this wallet and the provided key. If no key ID is provided, the message will be signed with the first available key.

Parameters

- **message** (*bytes, hexstring*) – Message to be signed. Must be unhashed and in bytes format.
- **key_term** (*int, str*) – Key ID or address of signing key. Search term can be key ID, key address, key WIF or key name
- **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 know what you are doing. Providing wrong value for k can result in leaking your private key!
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL
- **force_canonical** (*bool*) – Some wallets require a canonical s value, so you could set this to True

Return Signature

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*, *broadcast=False*, *replace_by_fee=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')
>>> 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)
<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*) – The 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, str*) – Total transaction fee in the smallest denominator (i.e. satoshis). Leave empty to get the estimated fee from service providers. You can also supply a string: 'low', 'normal' or 'high' to determine fees automatically.
- **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
- **broadcast** (*bool*) – Just return the transaction object and do not send it when broadcast = False. Default is False
- **replace_by_fee** (*bool*) – Signal replace by fee and allow to send a new transaction with higher fees. Sets sequence value to SEQUENCE_REPLACE_BY_FEE

Return WalletTransaction

transaction(*txid*)

Get WalletTransaction object for the provided 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*, *replace_by_fee=False*, *ignore_dust=None*)

Create a new transaction with specified outputs.

Inputs can be specified, but if not provided they will be selected from the 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([('1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb', 200000)])
>>> 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*) – The network name. Leave empty for default network
- **fee** (*int, str*) – Set fee manually, leave empty to calculate fees automatically. Set fees in the smallest currency denominator, for example satoshi's if you are using bitcoins. You can also supply a string: 'low', 'normal' or 'high' to determine fees automatically.
- **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
- **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 a 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. The default is True
- **replace_by_fee** (*bool*) – Signal replace-by-fee and allow you to send a new transaction with higher fees. Sets sequence value to SEQUENCE_REPLACE_BY_FEE
- **ignore_dust** (*bool*) – Do not include small amounts to avoid dust inputs. Default is to use the Wallet.ignore dust setting which is True by default.

Return WalletTransaction
object

transaction_delete(*txid*)

Remove specified transaction from this wallet and update related transactions.

Parameters
txid (*str*) – Transaction ID of transaction to remove

Returns

transaction_import(*t*)

Import a Transaction into this wallet. Link inputs to wallet keys if possible and return a 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*) – The network name. Leave empty for default network

Return WalletTransaction

transaction_last(*address*)

Get transaction ID for the latest transaction in the database for given address

Parameters
address (*str*) – The address

Return str

transaction_load(*txid=None*, *filename=None*)

Load a transaction object from a file which has been stored with the `Transaction.save()` method.

Specify transaction ID or filename.

Parameters

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

Return WalletTransaction

transaction_spent(*txid*, *output_n*)

Check if transaction with the given transaction ID and output_n is spent and return transaction IDs of spent transactions.

Retrieves information from the database, does not update the 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 a 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*, *limit=0*, *offset=0*)

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 the list. The default is False
- **limit** (*int*) – Maximum number of transactions to return. Combine with an offset parameter to use as pagination
- **offset** (*int*) – Number of transactions to skip

Return list of WalletTransaction

transactions_remove_unconfirmed(*hours_old=0*, *account_id=None*, *network=None*)

Removes all unconfirmed transactions from this wallet and updates related transactions / utxos.

Parameters

- **hours_old** (*int, float*) – Only delete unconfirmed transactions which are x hours old. You can also use decimals, ie: 0.5 for half an hour
- **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

Returns

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 the 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*) – The 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 a transaction or list or of transactions for this wallet with provided transaction IDs

Parameters

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

Returns

transactions_update_confirmations()

Update the number of confirmations and the status of transactions in the 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 the 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 sathosis 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 the latest unspent output in the database for the specified 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 the database. Use *utxos_update()* method first for updated values

```
>>> w = Wallet('bitcoinlib_legacy_wallet_test')
>>> w.utxos()
[{'value': 100000000, 'script': '', 'output_n': 0, 'transaction_id': ..., 'spent
↪': False, 'script_type': 'p2pkh', 'key_id': ..., 'address':
↪'16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg', 'confirmations': 0, 'txid':
↪'748799c9047321cb27a6320a827f1f69d767fe889c14bf11f27549638d566fe4', 'network_
↪name': 'bitcoin'}]
```

Parameters

- **account_id** (*int*) – Account ID
- **network** (*str*) – The network name. Leave empty for default network
- **min_confirms** (*int*) – Minimal confirmation needed to include in the 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": "n2S9Czehjvdmpwd2YqekxuUC1Tz5ZdK3YN",
  "script": "",
  "confirmations": 10,
  "output_n": 1,
  "txid": "9df91f89a3eb4259ce04af66ad4caf3c9a297feea5e0b3bc506898b6728c5003",
  "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

verify_message(*message*, *signature*, *key_term=None*)

Verify if a provided message is signed by the provided signature and matches a public key from this wallet.

If no key_term is provided, it will check the signature against all known keys for this wallet. This can be slow for larger wallets.

Parameters

- **message** (*bytes, hexstring*) – Message to verify. Must be unhashed and in bytes format.
- **signature** (Signature) – signature as Signature object
- **key_term** (*int, str*) – Key ID or address of verifying key. Search term can be key ID, key address, key WIF or key name

Return bool

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

Return Wallet Import Format string for a 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

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

Get witness types in use by this wallet. For example, 'legacy', 'segwit', 'p2sh-segwit'

Parameters

- **account_id** (*int*) – Account ID. Leave empty for the default account
- **network** (*str*) – The network name filter. The default filter is DEFAULT_NETWORK

Returns
list of witness types for this Wallet class

Return type
list of 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 an attribute of the 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 the database.

Parameters

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

as_dict(*include_private=False*)

Return current key information as a dictionary

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

balance_str()

Get total value of unspent outputs in string format

Return str
Key balance

static from_key(*name*, *wallet_id*, *session*, *key*, *account_id=0*, *network=None*, *change=0*, *purpose=84*, *parent_id=0*, *path='m'*, *key_type=None*, *encoding=None*, *witness_type='segwit'*, *multisig=False*, *cosigner_id=None*, *new_key_id=None*)

Create WalletKey from an 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 =
↪'xprv9s21ZrQH143K2mcs9jcK4EjALbu2z1N9qsMTUG1frmnXM3NNCSGR57yLhwTccfNCwdSQEDftgjCGm96P29wGGcb
↪'
>>> wk = WalletKey.from_key('import_key', w.wallet_id, w.session, wif)
>>> wk.address
'bc1qukcgc3guzt0a27j7vdegtgwxsrv4khc4688ycs'
>>> wk
<WalletKey(key_id=..., name=import_key,␣
```

(continues on next page)

(continued from previous page)

```
↪wif=zprvAWgYBBk7JR8GjN16pTBZUQvAgYBvsFM9g6Pu33oScnYHTEzphkbYKFHckMNncUg3kug1jAs1c3uNXiKWTYmH
↪ 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 84
- **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 encoding
- **witness_type** (*str*) – Witness type used when creating transaction script: legacy, p2sh-segwit or segwit.
- **multisig** (*bool*) – Specify if the key is part of a multisig wallet, used for create keys and key representations such as WIF and addreses
- **cosigner_id** (*int*) – Set this if you would like to create keys for other cosigners.
- **new_key_id** (*int*) – Key ID in the database (DbKey.id), use to directly insert key in the database without checks and without commiting. Mainly for internal usage, to significantly increase speed when inserting multiple keys.

Return WalletKey

The wallet key object

key()

Get HDKey object for the current walletKey

Return HDKey, list of HDKey

property name

Return name of the wallet key

Return str

public()

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

Return WalletKey

sign_message(*message*, *use_rfc6979=True*, *k=None*, *hash_type=1*, *force_canonical=False*)

Sign a message with this wallet key

Parameters

- **message** (*bytes, hexstring*) – Message to be signed. Must be unhashed and in bytes format.
- **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 know what you are doing. Providing a wrong value for k can result in leaking your private key!
- **hash_type** (*int*) – Specific hash type, default is SIGHASH_ALL
- **force_canonical** (*bool*) – Some wallets require a canonical s value, so you could set this to True

Return Signature

verify_message(*message*, *signature*)

Verify if the provided message is signed by signature for this wallet key.

Parameters

- **message** (*bytes, hexstring*) – Message to verify. Must be unhashed and in bytes format.
- **signature** (Signature) – signature as Signature object

Return bool

class bitcoinlib.wallets.**WalletTransaction**(*hdwallet*, *account_id=None*, **args*, ***kwargs*)

Bases: *Transaction*

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

All WalletTransaction items are stored in a database

Initialize a 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

add_input_from_wallet(*amount_min=0*, *key_id=None*, *min_confirms=1*)

Add a new input from an utxo of this wallet. If key_id is not specified, it adds the first input it finds with the minimum amount and minimum confirms specified.

WARNING: Change output and fees are not updated, so you risk overpaying fees!

Parameters

- **amount_min** (*int*) – Minimum value of new input
- **key_id** (*int*) – Filter by this key id
- **min_confirms** (*int*) – Minimum confirms of utxo

Return int
Index number of new input

bumpfee(*fee=0*, *extra_fee=0*, *broadcast=False*)

Increase the fee for this transaction. If replace-by-fee is signaled in this transaction, the fee can be increased to speed up inclusion on the blockchain.

If no fee or extra_fee is provided, the extra fee will be increased by the formule you can find in the code below using the BUMPFEE_DEFAULT_MULTIPLIER from the config settings.

The extra fee will be deducted from change output. This method fails if there are not enough change outputs to cover fees.

If this transaction does not have enough inputs to cover an extra fee, an extra wallet utxo will be aaded to inputs if available.

Previous broadcasted transaction will be removed from the wallet with this replace-by-fee transaction and wallet information updated.

Parameters

- **fee** (*int*) – New fee for this transaction
- **extra_fee** (*int*) – Extra fee to add to the current transaction fee
- **broadcast** (*bool*) – Increase fee and directly broadcast transaction to the network

Return None

delete()

Delete this transaction from the database.

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

Return int
Number of deleted transactions

export(*skip_change=True*)

Export this transaction as a 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 a WalletTransaction object from a Transaction object

Parameters

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

Return WalletTransaction

classmethod from_txid(*hdwallet*, *txid*)

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

Parameters

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

Return WalletTransaction

info()

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

save(*filename=None*)

Store the transaction object as a file, so it can be imported in bitcoinlib later with the `load()` method.

Parameters

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

Returns

send(*broadcast=True*)

Verify and push transaction to network. Update UTXO's in the database after sending was successful.

Parameters

broadcast (*bool*) – Verify transaction and broadcast, if set to False, the transaction is verified but not broadcasted, i. Default is True

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 this 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 the 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 the database

Return int

Transaction index number

to_transaction()

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*, *anti_fee_sniping=True*, *strict=True*, *ignore_dust=True*, *db_uri=None*, *db_cache_uri=None*, *db_password=None*)

Create a wallet with specified options if it doesn't exist, otherwise open the existing wallet.

Returns Wallet object

See Wallets class create method for option documentation

bitcoinlib.wallets.**wallet_delete**(*wallet*, *db_uri=None*, *force=False*, *db_password=None*)

Delete wallet and associated keys and transactions from the database. If the 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
- **db_password** (*str*) – Password to use for the encrypted database. Requires the installation of sqlcipher (see documentation).

Return int

Number of rows deleted, so 1 if successful

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

Delete wallet and associated keys from the database. If the wallet has unspent outputs, it raises a WalletError exception unless 'force=True' is specified. If the 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
- **db_password** (*str*) – Password to use for the encrypted database. Requires the installation of sqlcipher (see documentation).

Return int

Number of rows deleted, so 1 if successful

bitcoinlib.wallets.**wallet_empty**(*wallet*, *db_uri=None*, *db_password=None*)

Remove all generated keys and transactions from the 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
- **db_password** (*str*) – Password to use for the encrypted database. Requires the installation of sqlcipher (see documentation).

Return bool

True if successful

bitcoinlib.wallets.**wallet_exists**(*wallet*, *db_uri=None*, *db_password=None*)

Check if Wallets is defined in the database

Parameters

- **wallet** (*int, str*) – Wallet ID as integer or Wallet Name as string
- **db_uri** (*str*) – URI of the database
- **db_password** (*str*) – Password to use for the encrypted database. Requires the installation of sqlcipher (see documentation).

Return bool

True if the wallet exists otherwise False

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

List Wallets from the 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
- **db_password** (*str*) – Password to use for the encrypted database. Requires the installation of sqlcipher (see documentation).

Return dict

Dictionary of wallets defined in the database

8.18 bitcoinlib.mnemonic module

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

Bases: object

Class to convert, generate and parse Mnemonic sentences

Implementation of BIP0039 for Mnemonics passphrases

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

Init Mnemonic class and read wordlist of specified language

Parameters

language (*str*) – use specific wordlist, i.e. Chinese, Dutch (in development), English, French, Italian, Japanese or Spanish. Leave empty for default 'English'

static checksum(*data*)

Calculates checksum for given data key

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

Return str
Checksum of key in bits

static detect_language(*words*)

Detect language of given phrase

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

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

Return str
Language

generate(*strength=128*, *add_checksum=True*)

Generate a random Mnemonic key

Uses cryptographically secure os.urandom() function to generate data. Then creates a Mnemonic sentence with the 'to_mnemonic' method.

Parameters

- **strength** (*int*) – Key strength in number of bits as multiply of 32, default is 128 bits. It advised to specify 128 bits or more, i.e.: 128, 256, 512 or 1024
- **add_checksum** (*bool*) – Included a checksum? Default is True

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

sanitize_mnemonic(*words*)

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

Raises an error if unrecognised word is found

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

Return str
Sanitized list of words

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

Convert Mnemonic words back to key data entropy

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

Parameters

- **words** (*str*) – Mnemonic words as string of list of words

- **includes_checksum** (*bool*) – Boolean to specify if checksum is used. Default is True

Return bytes
Entropy seed

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

Convert key data entropy to Mnemonic sentence

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

Parameters

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

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

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

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

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

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

↪'6969ed4666db67fc74fae7869e2acf3c766b5ef95f5e31eb2fcebd93d76069c6de971225f700042b0b513f0ad6c8
↪'
```

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

8.19 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.

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

Get WIF prefix for this network and specifications in arguments

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

Parameters

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

Return bytes

exception bitcoinlib.networks.**NetworkError**(*msg=''*)

Bases: Exception

Network Exception class

bitcoinlib.networks.**network_by_value**(*field*, *value*)

Return all networks for field and (prefix) value.

Example, get available networks for WIF or address prefix

```
>>> network_by_value('prefix_wif', 'B0')
['litecoin', 'litecoin_legacy']
>>> network_by_value('prefix_address', '6f')
['testnet', 'testnet4', 'signet', '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'\x9e', b'\xf1']
>>> network_values_for('prefix_address_p2sh')
[b'\x95', b'\x05', b'\xc4', b'2', b':', b'\x16']
```

Parameters

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

Return str

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

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

Example, get bitcoin 'xprv' info:

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

Or retreive info with full WIF string:

```
>>> wif_prefix_search(
↪'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', 'regtest', '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

8.20 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('000000000000000000000154ba9d02ddd6cee0d71d1ea232753e02c9ac6affd709',␣
↪version=0x20000000, prev_block=
↪'000000000000000000000f9578cda278ae7a2002e50d8e6079d11e2ea1f672b483', merkle_root=
↪'20e86f03c24c53c12014264d0e405e014e15a02ad02c174f017ee040750f8d9d',␣
↪time=1592848036, bits=387044594, nonce=791719079)
>>> b
<Block(000000000000000000000154ba9d02ddd6cee0d71d1ea232753e02c9ac6affd709, None,␣
↪transactions: 0)>
```

Parameters

- **block_hash** (*bytes, str*) – Hash value of serialized block
- **version** (*bytes, str, in*) – 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('000000000000000000154ba9d02ddd6cee0d71d1ea232753e02c9ac6affd709
↪', version=0x20000000, prev_block=
↪'00000000000000000000f9578cda278ae7a2002e50d8e6079d11e2ea1f672b483', 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 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 from specified hash.
- **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()
'000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f'
```

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 from specified hash.
- **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()
'000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f'
```

Parameters

- **raw** (*BytesIO*) – 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 from specified hash.
- **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

parse_transaction()

Parse a single transaction from Block, if transaction data is available in txs_data attribute. Add Transaction object in Block and return the transaction

Return Tranasaction

parse_transaction_dict(*index=None*)

Parse a single transaction from Block, if transaction data is available in txs_data attribute. Add Transaction object in Block and return the transaction

Return Transaction

parse_transactions(*limit=0*)

Parse raw transactions from Block, if transaction data is available in txs_data attribute. Creates Transaction objects in Block.

Parameters
limit (*int*) – Maximum number of transactions to parse

Returns

parse_transactions_dict()

Parse raw transactions from Block, if transaction data is available in txs_data attribute. Returns a list of transactions dictionaries.

Only works if transactions are not parsed yet with *parse_transactions()* or parse_transactions=True when creating a new block object.

Returns

serialize()

Serialize raw block in bytes.

A block consists of an 80 bytes header: * version - 4 bytes * previous block - 32 bytes * merkle root - 32 bytes * timestamp - 4 bytes * bits - 4 bytes * nonce - 4 bytes

Followed by a list of raw serialized transactions.

Method will raise an error if one of the header fields is missing or has an incorrect size.

Return bytes

property target

Block target calculated from block's bits. Block hash must be below this target. Used to calculate block difficulty.

Return int

property target_hex

Block target in hexadecimal string of 64 characters.

Return str

update_totals()

property version_bin

Get the block version as binary string. Since BIP9 protocol changes are signaled by changing one of the 29 last bits of the version number.

```
>>> from bitcoinlib.services.services import Service
>>> srv = Service()
>>> b = srv.getblock(450001)
>>> print(b.version_bin)
00100000000000000000000000000010
```

Return str

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

8.21 bitcoinlib.values module

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

Bases: object

Class to represent and convert cryptocurrency values

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

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

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

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

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

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

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

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

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

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

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

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

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

The Value class can be represented in several formats.

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

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

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

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

Parameters

- **value** (*int, float, str*) – Value as integer, float or string. Numeric values must be supllied in the 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 the smallest denominator as input. Such as represented in script and transactions cryptocurrency values.

Parameters

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

Return Value

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

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

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

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

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

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

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

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

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

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

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

Parameters

- **denominator** (*int, float, str*) – Denominator as integer or string. Such as 0.001 or m for milli, 1000 or k for kilo, etc. See NETWORK_DENOMINATORS for list of available denominator symbols. If not provided the default self.denominator value is used. Use value 'auto' to automatically determine the 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 the 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 the 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 the 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 the 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

8.22 bitcoinlib.services.services module

class bitcoinlib.services.services.**Cache**(*network*, *db_uri=''*)

Bases: object

Store transaction, utxo and address information in the 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 the number of blocks on the current network from the cache if recent data is available.

Parameters
never_expires (*bool*) – Always return the 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 the specified number of blocks.

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

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

Return int
Fee in the smallest network denominator (satoshi)

getaddress(*address*)

Get address information from the 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 the database cache.

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

Return Block

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

Get a 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 the cache. Returns an empty list if no transactions are found or caching is disabled.

Parameters

- **address** (*str*) – Address string
- **after_txid** (*bytes*) – Transaction ID of the 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 a list of unspent outputs (UTXO's) for the specified address from the database cache.

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

Parameters

- **address** (*str*) – Address string
- **after_txid** (*bytes*) – Transaction ID of the 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 the cache

param address
Address string

type address
str

param last_block
Number or last block retrieved from the service provider. For instance, if an 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 the cache database

Parameters

block (Block) – The Block object to store in the cache

Returns

store_blockcount(*blockcount*)

Store network blockcount in the 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*, *index=None*, *commit=True*)

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

Parameters

- **t** (Transaction) – The Transaction object
- **index** (*int*) – Order in block
- **commit** (*bool*) – Commit transaction to the database. Default is True. Can be disabled if a larger number of transactions are added to the cache, so you can commit outside this method.

Returns

store_utxo(*txid*, *index_n*, *commit=True*)

Store utxo in cache. Updates only known transaction outputs for transactions which are fully cached

Parameters

- **txid** (*str*) – Transaction ID
- **index_n** (*int*) – Index number of output
- **commit** (*bool*) – 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*, *strict=True*, *wallet_name=None*, *provider_name=None*)

Bases: object

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

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

Create a service object for the specified network. By default, the object connects 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.
- **strict** (*bool*) – Strict checks of valid signatures, scripts and transactions. Normally use strict=True for wallets, transaction validations etcetera. For blockchain parsing strict=False should be used, but be sure to check warnings in the log file. Default is True.
- **wallet_name** (*str*) – Name of wallet if connecting to bitcoin node
- **provider_name** (*str*) – Name of a specific provider to connect to. Note this is different from the providers list argument: the lists mention a type of provider such as 'blockbook' or 'bcoin', the provider name is a key in the providers.json dictionary file such as 'bcoin.testnet.localhost'.

blockcount()

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

Block count is cashed for BLOCK_COUNT_CACHE_TIME seconds to avoid too many calls to service providers.

Return int

estimatefee(*blocks=5*, *priority=''*)

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

Parameters

- **blocks** (*int*) – Expected confirmation time in blocks.
- **priority** (*str*) – Priority for transaction: can be 'low', 'medium' or 'high'. Overwrites value supplied in 'blocks' argument

Return int

Fee in the 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 the block with the specified block height or block hash from service providers.

If parse_transaction is set to True, a list of Transaction objects will be returned otherwise a list of transaction IDs.

Some providers require 1 or 2 extra requests 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 and limit=4 only transaction 5 to 8 are returned to the Block'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(000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f, 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 25 if parse transaction is enabled, otherwise returns all txid's (9999)

Return Block

getcacheaddressinfo(*address*)

Get address information from the 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 the current network. Such as difficulty, latest block, mempool size and network hashrate.

Return dict

getinputvalues(*t*)

Retrieve values for transaction inputs for a 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) – A Transaction object

Return Transaction

getrawblock(*blockid*)

Get a raw block as a hexadecimal string for a block with a 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 a 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 the specified address.

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

Parameters

- **address** (*str*) – Address string
- **after_txid** (*str*) – Transaction ID of the 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 a list of unspent outputs (UTXO's) for the specified address.

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

Parameters

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

- **after_txid** (*str*) – Transaction ID of the 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. Sometimes ignored by service providers if more results are returned by default.

Return dict
UTXO's per address

isspent(*txid*, *output_n*)

Check if the output with the provided transaction ID and output number is already 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 a 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

8.23 bitcoinlib.services package

8.23.1 Submodules

8.23.1.1 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=10*, *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

8.23.1.2 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*, *strict=True*, *wallet_name=''*)

Bases: object

request(*url_path*, *variables=None*, *method='get'*, *secure=True*, *post_data=''*, *header=None*)

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

Bases: Exception

8.23.1.3 bitcoinlib.services.bcoin module

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

Bases: *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()

getrawblock(*blockid*)

getrawtransaction(*txid*)

gettransaction(*txid*)

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

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

isspent(*txid*, *index*)

mempool(*txid=''*)

8.23.1.4 bitcoinlib.services.bitaps module

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

Bases: *BaseClient*

blockcount()

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

getbalance(*addresslist*)

gettransaction(*txid*)

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

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

8.23.1.5 bitcoinlib.services.bitcoind module

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

Bases: *BaseClient*

Class to interact with bitcoind, the Bitcoin deamon

Open connection to bitcoin node

Parameters

- **base_url** – Connection URL in format http(s)://user:password@host:port.
- **network** – Bitcoin mainnet or testnet. Default is bitcoin mainnet
- **denominator** – Denominator for this currency. Should be always 100000000 (Satoshi's) for bitcoin
- **wallet_name** – Name of Bitcoin core wallet to use. Make sure the wallet exists, otherwise connection will fail.

Type
str

Type
str

Type
str

Type
str

`blockcount`()

`estimatefee`(*blocks*)

`static from_config`(*configfile=None*, *network='bitcoin'*, ***kwargs*)

Read settings from bitcoind config file

Obsolete: does not work anymore, passwords are not stored in bitcoin config, only hashed password.

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

`getbalance`(*addresslist*)

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

`getinfo`()

`getrawblock`(*blockid*)

`getrawtransaction`(*txid*)

`gettransaction`(*txid*)

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

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

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

`mempool`(*txid=''*)

`sendrawtransaction`(*rawtx*)

`exception bitcoinlib.services.bitcoind.ConfigError`(*msg=''*)

Bases: `Exception`

8.23.1.6 bitcoinlib.services.bitcoinlibtest module

`class bitcoinlib.services.bitcoinlibtest.BitcoinLibTestClient`(*network*, *base_url*, *denominator*, **args*)

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

Parameters

- **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

8.23.1.7 bitcoinlib.services.bitflyer module

class bitcoinlib.services.bitflyer.**BitflyerClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *BaseClient*

blockcount()

compose_request(*function*, *parameter=''*, *parameter2=''*, *method='get'*)

getbalance(*addresslist*)

8.23.1.8 bitcoinlib.services.bitgo module

class bitcoinlib.services.bitgo.**BitGoClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *BaseClient*

blockcount()

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

estimatefee(*blocks*)

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

8.23.1.9 bitcoinlib.services.blockbook module

class bitcoinlib.services.blockbook.**BlockbookClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *BaseClient*

blockcount()

compose_request(*category*, *data*, *cmd=''*, *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*, *output_n*)

mempool(*txid*)

sendrawtransaction(*rawtx*)

8.23.1.10 bitcoinlib.services.blockbook1 module

class bitcoinlib.services.blockbook1.**BlockbookClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *BaseClient*

blockcount()

compose_request(*category*, *data*, *cmd=''*, *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*, *output_n*)

mempool(*txid*)

sendrawtransaction(*rawtx*)

8.23.1.11 bitcoinlib.services.blockchaininfo module

class bitcoinlib.services.blockchaininfo.**BlockchainInfoClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *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=''*)

8.23.1.12 bitcoinlib.services.blockchair module

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

Bases: *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*)

8.23.1.13 bitcoinlib.services.blockcypher module

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

Bases: *BaseClient*

blockcount()

compose_request(*function*, *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*, *output_n*)

mempool(*txid*)

sendrawtransaction(*rawtx*)

8.23.1.14 bitcoinlib.services.blocksmurfer module

class bitcoinlib.services.blocksmurfer.**BlocksmurferClient**(*network*, *base_url*, *denominator*, *api_key*, **args*)

Bases: *BaseClient*

blockcount()

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

estimatefee(*blocks*)

getbalance(*addresslist*)

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

getinfo()

getrawblock(*blockid*)

getrawtransaction(*txid*)

gettransaction(*txid*, *block_count=None*)

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

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

isspent(*txid*, *output_n*)

mempool(*txid*)

sendrawtransaction(*rawtx*)

8.23.1.15 bitcoinlib.services.blockstream module

class bitcoinlib.services.blockstream.**BlockstreamClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *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*)

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

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

isspent(*txid*, *output_n*)

mempool(*txid*)

sendrawtransaction(*rawtx*)

8.23.1.16 bitcoinlib.services.chainso module

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

Bases: *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*)

8.23.1.17 bitcoinlib.services.cryptoid module

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

Bases: *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*)

8.23.1.18 bitcoinlib.services.dogecoind module

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

Bases: Exception

class bitcoinlib.services.dogecoind.**DogecoindClient**(*network='dogecoin'*, *base_url=''*, *denominator=100000000*, **args*)

Bases: *BaseClient*

Class to interact with dogecoind, 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'*, ***kargs*)

Read settings from dogecoind 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 DogecoindClient

getinfo()

getrawtransaction(*txid*)

gettransaction(*txid*, *block_height=None*, *get_input_values=True*)

getutxos(*address*, *after_txid=''*, *max_txs=20*)

isspent(*txid*, *index*)

mempool(*txid=''*)

sendrawtransaction(*rawtx*)

8.23.1.19 bitcoinlib.services.electrumx module

class bitcoinlib.services.electrumx.**ElectrumxClient**(*network*, *base_url*, *denominator*, *api_key*, **args*)

Bases: *BaseClient*

blockcount()

compose_request(*method*, *parameters=None*)

estimatefee(*blocks*)

getbalance(*addresslist*)

getrawtransaction(*txid*)

gettransaction(*txid*, *block_count=None*)

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

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

mempool(*txid*)

sendrawtransaction(*rawtx*)

8.23.1.20 bitcoinlib.services.litecoinblockexplorer module

class bitcoinlib.services.litecoinblockexplorer.**LitecoinBlockexplorerClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *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(*txid*)

gettransaction(*txid*)

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

isspent(*txid*, *output_n*)

mempool(*txid*)

sendrawtransaction(*rawtx*)

8.23.1.21 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: *BaseClient*

Class to interact with litecoind, the Litecoin deamon

Open connection to litecoin node

Parameters

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

Type
str

Type
str

Type
str

blockcount()

estimatefee(*blocks*)

static from_config(*configfile=None*, *network='litecoin'*, ***kargs*)

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

getbalance(*addresslist*)

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

getinfo()

getrawblock(*blockid*)

getrawtransaction(*txid*)

gettransaction(*txid*)

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

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

isspent(*txid*, *index*)

mempool(*txid=''*)

sendrawtransaction(*rawtx*)

8.23.1.22 bitcoinlib.services.litecoreio module

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

Bases: *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*)

8.23.1.23 bitcoinlib.services.mempool module

class bitcoinlib.services.mempool.**MempoolClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *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*)

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

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

isspent(*txid*, *output_n*)

mempool(*txid=''*)

sendrawtransaction(*rawtx*)

8.23.1.24 bitcoinlib.services.nownodes module

class bitcoinlib.services.nownodes.**NownodesClient**(*network*, *base_url*, *denominator*, **args*)

Bases: *BaseClient*

blockcount()

compose_request(*function=''*, *params=None*, *method='post'*)

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

getinfo()

getrawblock(*blockid*)

getrawtransaction(*txid*)

gettransaction(*txid*)

isspent(*txid*, *output_n*)

mempool(*txid=''*)

sendrawtransaction(*rawtx*)

8.23.2 Module contents

8.24 bitcoinlib.config package

8.24.1 Submodules

8.24.1.1 bitcoinlib.config.config module

bitcoinlib.config.config.**initialize_lib**()

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

8.24.1.2 bitcoinlib.config.opcodes module

bitcoinlib.config.opcodes.**op**()

8.24.1.3 bitcoinlib.config.secp256k1 module

8.24.2 Module contents

8.25 bitcoinlib.db module

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

Bases: object

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

Create the new database if it doesn't exist yet

drop_db(*yes_i_am_sure=False*)

class bitcoinlib.db.DbConfig(**kwargs)

Bases: 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: 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

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

witness_type

Key witness type, only specify when using mixed wallets. Can be 'legacy', 'segwit' or 'p2sh-segwit'. Default is segwit.

class bitcoinlib.db.DbKeyMultisigChildren(**kwargs)

Bases: 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 than 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: 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

class bitcoinlib.db.DbTransaction(**kwargs)

Bases: 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

index

Index of transaction in block

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

Tranaction 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: 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, p2sh-segwit or taproot. Default is segwit

witnesses

Witnesses (signatures) used in Segwit transaction inputs

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

Bases: 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

is_change

Is this a change output / output to own wallet?

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', 'p2wsh', 'p2tr'. 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: 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.

anti_fee_sniping

Set default locktime in transactions to avoid fee-sniping

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

ignore_dust

Ignore dust outputs

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

strict

Raise exception if incorrect scripts or signatures are detected

transactions

Link to transaction (DbTransactions) in this wallet

witness_type

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

class bitcoinlib.db.**EncryptedBinary**(**args: Any*, ***kwargs: Any*)

Bases: TypeDecorator

FieldType for encrypted Binary storage using EAS encryption

Construct a TypeDecorator.

Arguments sent here are passed to the constructor of the class assigned to the impl class level attribute, assuming the impl is a callable, and the resulting object is assigned to the self.impl instance attribute (thus overriding the class attribute of the same name).

If the class level impl is not a callable (the unusual case), it will be assigned to the same instance attribute 'as-is', ignoring those arguments passed to the constructor.

Subclasses can override this to customize the generation of self.impl entirely.

cache_ok = True

Indicate if statements using this ExternalType are "safe to cache".

The default value None will emit a warning and then not allow caching of a statement which includes this type. Set to False to disable statements using this type from being cached at all without a warning. When

set to True, the object's class and selected elements from its state will be used as part of the cache key. For example, using a TypeDecorator:

```
class MyType(TypeDecorator):
    impl = String

    cache_ok = True

    def __init__(self, choices):
        self.choices = tuple(choices)
        self.internal_only = True
```

The cache key for the above type would be equivalent to:

```
>>> MyType(["a", "b", "c"])._static_cache_key
(<class '__main__.MyType'>, ('choices', ('a', 'b', 'c')))
```

The caching scheme will extract attributes from the type that correspond to the names of parameters in the __init__() method. Above, the "choices" attribute becomes part of the cache key but "internal_only" does not, because there is no parameter named "internal_only".

The requirements for cacheable elements is that they are hashable and also that they indicate the same SQL rendered for expressions using this type every time for a given cache value.

To accommodate for datatypes that refer to unhashable structures such as dictionaries, sets and lists, these objects can be made "cacheable" by assigning hashable structures to the attributes whose names correspond with the names of the arguments. For example, a datatype which accepts a dictionary of lookup values may publish this as a sorted series of tuples. Given a previously un-cacheable type as:

```
class LookupType(UserDefinedType):
    """a custom type that accepts a dictionary as a parameter.

    this is the non-cacheable version, as "self.lookup" is not
    hashable.

    """

    def __init__(self, lookup):
        self.lookup = lookup

    def get_col_spec(self, **kw):
        return "VARCHAR(255)"

    def bind_processor(self, dialect): ...  # works with "self.lookup" ...
```

Where "lookup" is a dictionary. The type will not be able to generate a cache key:

```
>>> type_ = LookupType({"a": 10, "b": 20})
>>> type_._static_cache_key
<stdin>:1: SAWarning: UserDefinedType LookupType({'a': 10, 'b': 20}) will not
produce a cache key because the ``cache_ok`` flag is not set to True.
Set this flag to True if this type object's state is safe to use
in a cache key, or False to disable this warning.
symbol('no_cache')
```

If we **did** set up such a cache key, it wouldn't be usable. We would get a tuple structure that contains a

dictionary inside of it, which cannot itself be used as a key in a "cache dictionary" such as SQLAlchemy's statement cache, since Python dictionaries aren't hashable:

```
>>> # set cache_ok = True
>>> type_.cache_ok = True

>>> # this is the cache key it would generate
>>> key = type_._static_cache_key
>>> key
(<class '__main__.LookupType'>, ('lookup', {'a': 10, 'b': 20}))

>>> # however this key is not hashable, will fail when used with
>>> # SQLAlchemy statement cache
>>> some_cache = {key: "some sql value"}
Traceback (most recent call last): File "<stdin>", line 1,
in <module> TypeError: unhashable type: 'dict'
```

The type may be made cacheable by assigning a sorted tuple of tuples to the ".lookup" attribute:

```
class LookupType(UserDefinedType):
    """a custom type that accepts a dictionary as a parameter.

    The dictionary is stored both as itself in a private variable,
    and published in a public variable as a sorted tuple of tuples,
    which is hashable and will also return the same value for any
    two equivalent dictionaries.  Note it assumes the keys and
    values of the dictionary are themselves hashable.

    """

    cache_ok = True

    def __init__(self, lookup):
        self._lookup = lookup

        # assume keys/values of "lookup" are hashable; otherwise
        # they would also need to be converted in some way here
        self.lookup = tuple((key, lookup[key]) for key in sorted(lookup))

    def get_col_spec(self, **kw):
        return "VARCHAR(255)"

    def bind_processor(self, dialect): ...  # works with "self._lookup" ...
```

Where above, the cache key for `LookupType({"a": 10, "b": 20})` will be:

```
>>> LookupType({"a": 10, "b": 20})._static_cache_key
(<class '__main__.LookupType'>, ('lookup', (('a', 10), ('b', 20))))
```

Added in version 1.4.14: - added the `cache_ok` flag to allow some configurability of caching for `TypeDecorator` classes.

Added in version 1.4.28: - added the `ExternalType` mixin which generalizes the `cache_ok` flag to both the `TypeDecorator` and `UserDefinedType` classes.

> **See also**
>
> sql_caching

impl

alias of LargeBinary

key = None

process_bind_param(*value*, *dialect*)

Receive a bound parameter value to be converted.

Custom subclasses of _types.TypeDecorator should override this method to provide custom behaviors for incoming data values. This method is called at **statement execution time** and is passed the literal Python data value which is to be associated with a bound parameter in the statement.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be None.
- **dialect** – the Dialect in use.

> **See also**
>
> types_typedecorator
>
> _types.TypeDecorator.process_result_value()

process_result_value(*value*, *dialect*)

Receive a result-row column value to be converted.

Custom subclasses of _types.TypeDecorator should override this method to provide custom behaviors for data values being received in result rows coming from the database. This method is called at **result fetching time** and is passed the literal Python data value that's extracted from a database result row.

The operation could be anything desired to perform custom behavior, such as transforming or deserializing data.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be None.
- **dialect** – the Dialect in use.

> **See also**
>
> types_typedecorator
>
> _types.TypeDecorator.process_bind_param()

class bitcoinlib.db.**EncryptedString**(**args: Any*, ***kwargs: Any*)

Bases: TypeDecorator

FieldType for encrypted String storage using EAS encryption

Construct a TypeDecorator.

Arguments sent here are passed to the constructor of the class assigned to the impl class level attribute, assuming the impl is a callable, and the resulting object is assigned to the self.impl instance attribute (thus overriding the class attribute of the same name).

If the class level impl is not a callable (the unusual case), it will be assigned to the same instance attribute 'as-is', ignoring those arguments passed to the constructor.

Subclasses can override this to customize the generation of self.impl entirely.

cache_ok = True

Indicate if statements using this ExternalType are "safe to cache".

The default value None will emit a warning and then not allow caching of a statement which includes this type. Set to False to disable statements using this type from being cached at all without a warning. When set to True, the object's class and selected elements from its state will be used as part of the cache key. For example, using a TypeDecorator:

```
class MyType(TypeDecorator):
    impl = String

    cache_ok = True

    def __init__(self, choices):
        self.choices = tuple(choices)
        self.internal_only = True
```

The cache key for the above type would be equivalent to:

```
>>> MyType(["a", "b", "c"])._static_cache_key
(<class '__main__.MyType'>, ('choices', ('a', 'b', 'c')))
```

The caching scheme will extract attributes from the type that correspond to the names of parameters in the __init__() method. Above, the "choices" attribute becomes part of the cache key but "internal_only" does not, because there is no parameter named "internal_only".

The requirements for cacheable elements is that they are hashable and also that they indicate the same SQL rendered for expressions using this type every time for a given cache value.

To accommodate for datatypes that refer to unhashable structures such as dictionaries, sets and lists, these objects can be made "cacheable" by assigning hashable structures to the attributes whose names correspond with the names of the arguments. For example, a datatype which accepts a dictionary of lookup values may publish this as a sorted series of tuples. Given a previously un-cacheable type as:

```
class LookupType(UserDefinedType):
    """a custom type that accepts a dictionary as a parameter.

    this is the non-cacheable version, as "self.lookup" is not
    hashable.

    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, lookup):
        self.lookup = lookup

    def get_col_spec(self, **kw):
        return "VARCHAR(255)"

    def bind_processor(self, dialect): ...  # works with "self.lookup" ...
```

Where "lookup" is a dictionary. The type will not be able to generate a cache key:

```
>>> type_ = LookupType({"a": 10, "b": 20})
>>> type_._static_cache_key
<stdin>:1: SAWarning: UserDefinedType LookupType({'a': 10, 'b': 20}) will not
produce a cache key because the ``cache_ok`` flag is not set to True.
Set this flag to True if this type object's state is safe to use
in a cache key, or False to disable this warning.
symbol('no_cache')
```

If we **did** set up such a cache key, it wouldn't be usable. We would get a tuple structure that contains a dictionary inside of it, which cannot itself be used as a key in a "cache dictionary" such as SQLAlchemy's statement cache, since Python dictionaries aren't hashable:

```
>>> # set cache_ok = True
>>> type_.cache_ok = True

>>> # this is the cache key it would generate
>>> key = type_._static_cache_key
>>> key
(<class '__main__.LookupType'>, ('lookup', {'a': 10, 'b': 20}))

>>> # however this key is not hashable, will fail when used with
>>> # SQLAlchemy statement cache
>>> some_cache = {key: "some sql value"}
Traceback (most recent call last): File "<stdin>", line 1,
in <module> TypeError: unhashable type: 'dict'
```

The type may be made cacheable by assigning a sorted tuple of tuples to the ".lookup" attribute:

```
class LookupType(UserDefinedType):
    """a custom type that accepts a dictionary as a parameter.

    The dictionary is stored both as itself in a private variable,
    and published in a public variable as a sorted tuple of tuples,
    which is hashable and will also return the same value for any
    two equivalent dictionaries.  Note it assumes the keys and
    values of the dictionary are themselves hashable.

    """

    cache_ok = True

    def __init__(self, lookup):
        self._lookup = lookup
```

(continues on next page)

(continued from previous page)

```
        # assume keys/values of "lookup" are hashable; otherwise
        # they would also need to be converted in some way here
        self.lookup = tuple((key, lookup[key]) for key in sorted(lookup))

    def get_col_spec(self, **kw):
        return "VARCHAR(255)"

    def bind_processor(self, dialect): ...  # works with "self._lookup" ...
```

Where above, the cache key for `LookupType({"a": 10, "b": 20})` will be:

```
>>> LookupType({"a": 10, "b": 20})._static_cache_key
(<class '__main__.LookupType'>, ('lookup', (('a', 10), ('b', 20))))
```

Added in version 1.4.14: - added the `cache_ok` flag to allow some configurability of caching for `TypeDecorator` classes.

Added in version 1.4.28: - added the `ExternalType` mixin which generalizes the `cache_ok` flag to both the `TypeDecorator` and `UserDefinedType` classes.

See also

sql_caching

impl

alias of `String`

key = None

process_bind_param(*value*, *dialect*)

Receive a bound parameter value to be converted.

Custom subclasses of `_types.TypeDecorator` should override this method to provide custom behaviors for incoming data values. This method is called at **statement execution time** and is passed the literal Python data value which is to be associated with a bound parameter in the statement.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **dialect** – the `Dialect` in use.

See also

types_typedecorator

`_types.TypeDecorator.process_result_value()`

process_result_value(*value*, *dialect*)

Receive a result-row column value to be converted.

Custom subclasses of `_types.TypeDecorator` should override this method to provide custom behaviors for data values being received in result rows coming from the database. This method is called at **result fetching time** and is passed the literal Python data value that's extracted from a database result row.

The operation could be anything desired to perform custom behavior, such as transforming or deserializing data.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **dialect** – the `Dialect` in use.

See also

types_typedecorator

`_types.TypeDecorator.process_bind_param()`

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

Used to add a new column to the 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='0.7.9'*)

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

8.26 bitcoinlib.db_cache module

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

Bases: `object`

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

Create a new database if it doesn't exist yet

drop_db()

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

Bases: `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: `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: 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

index

Index of transaction in block

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

txid

Hexadecimal representation of transaction hash or transaction ID

version

Tranaction 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: 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: 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*)

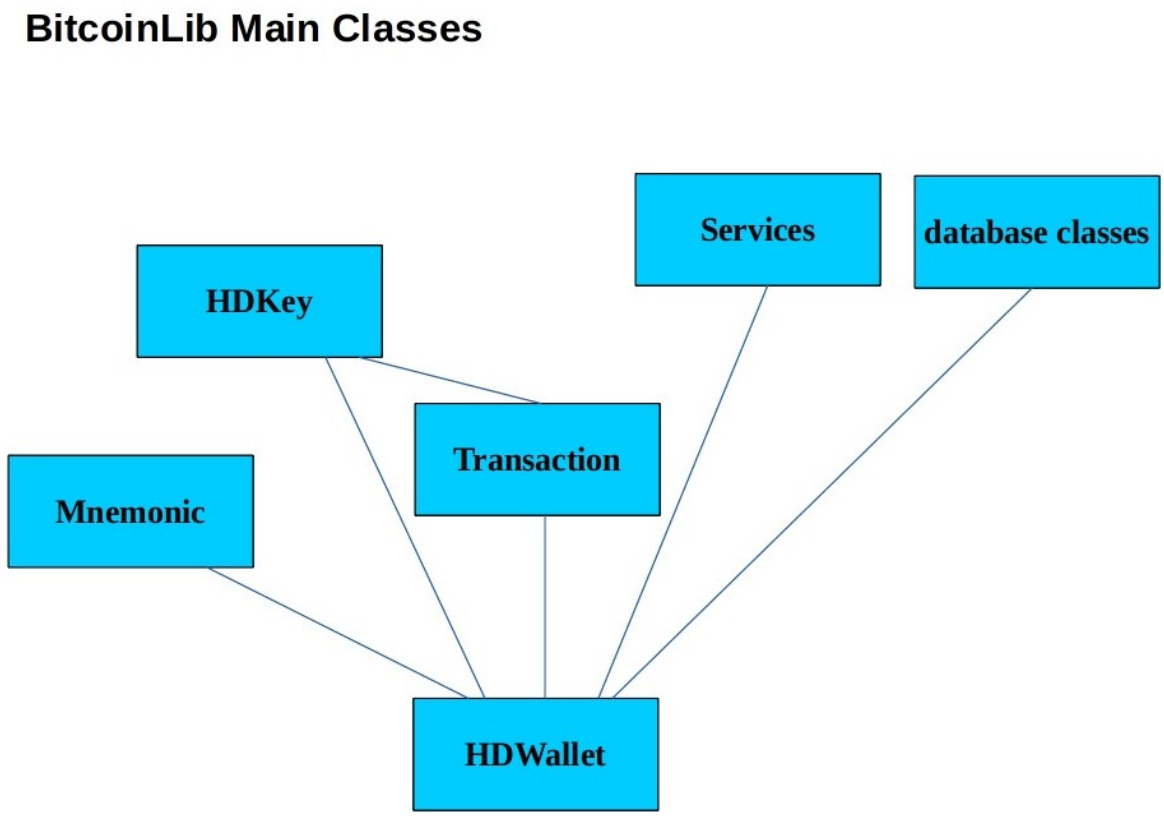
Bases: Enum

legacy = 'legacy'

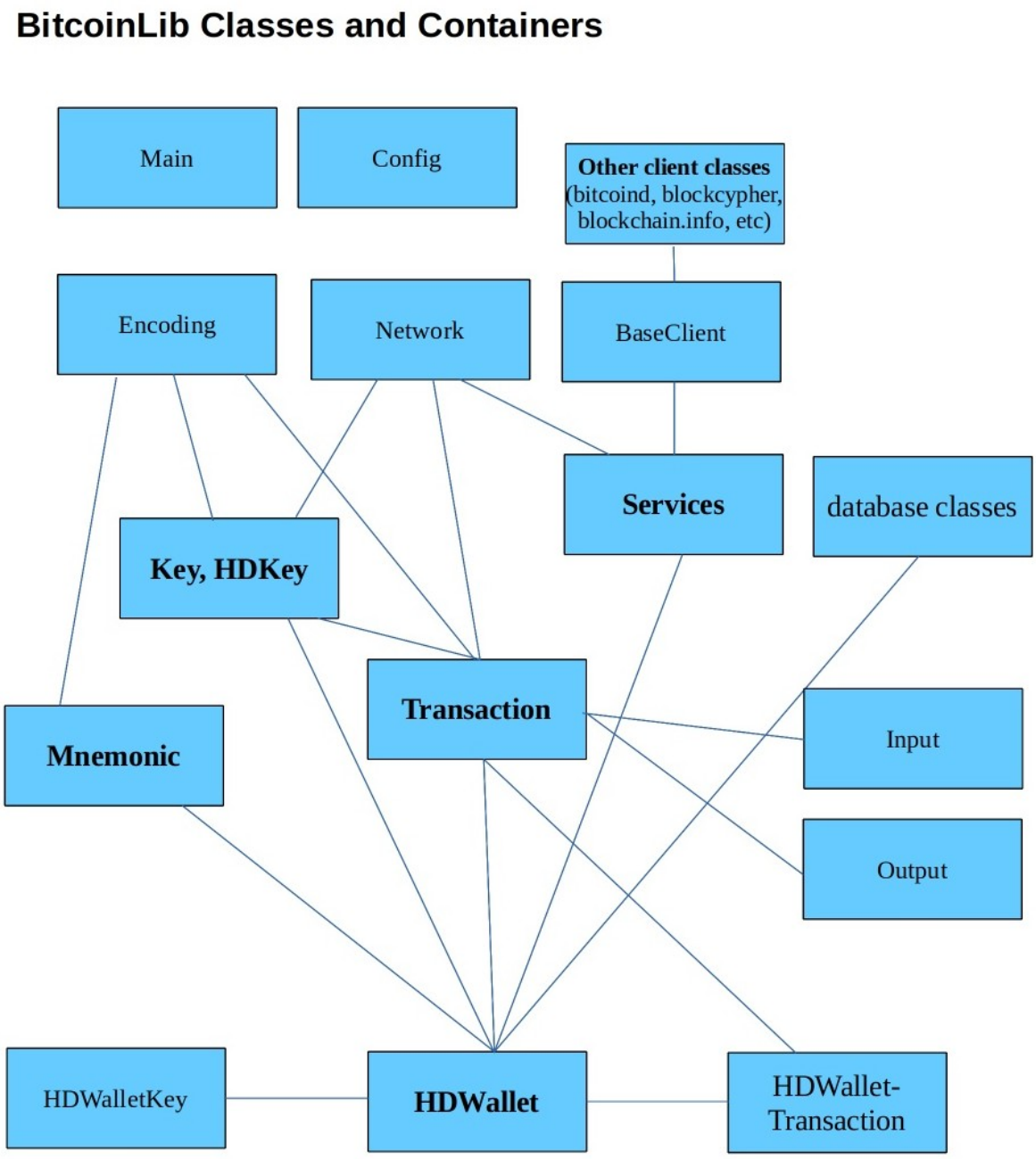
segwit = 'segwit'

8.27 Classes Overview

These are the main Bitcoinlib classes



This is an overview of all BitcoinLib classes.



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

8.28 bitcoinlib

8.28.1 bitcoinlib package

8.28.1.1 Subpackages

bitcoinlib.tools package

Submodules

bitcoinlib.tools.benchmark module

bitcoinlib.tools.clw module

bitcoinlib.tools.clw.**create_transaction**(*wlt*, *send_args*, *args*)

bitcoinlib.tools.clw.**create_wallet**(*wallet_name*, *args*, *db_uri*, *output_to*)

bitcoinlib.tools.clw.**exception_handler**(*exception_type*, *exception*, *traceback*)

bitcoinlib.tools.clw.**get_passphrase**(*strength*, *interactive=False*, *quiet=False*)

bitcoinlib.tools.clw.**main**()

bitcoinlib.tools.clw.**parse_args**()

bitcoinlib.tools.clw.**print_transaction**(*wt*)

bitcoinlib.tools.import_database module

bitcoinlib.tools.mnemonic_key_create module

bitcoinlib.tools.sign_raw module

bitcoinlib.tools.wallet_multisig_2of3 module

Module contents

8.28.1.2 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*) – Cryptocurrency address in base-58 format
- **as_hex** (*bool*) – Output as hexstring

Return bytes, str

Public Key Hash

bitcoinlib.encoding.**addr_bech32_checksum**(*bech*)

Get bech32 checksum. Returns 1 for bech32 addresses and 0x2bc830a3 for bech32m addresses. More info https://github.com/bitcoin/bips/blob/master/bip-0350.mediawiki

```
>>> addr_bech32_checksum(
↪'bc1pw508d6qejxtdg4y5r3zarvary0c5xw7kw508d6qejxtdg4y5r3zarvary0c5xw7kt5nd6y')
734539939
```

Parameters

bech (*str*) – Bech32 address to convert

Return bool

Bech32 checksum

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 sizes of 20 and 32 bytes are supported

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 a 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*) – Cryptocurrency 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.**aes_decrypt**(*encrypted_data*, *key*)

Decrypt encrypted data using AES Symmetric Block cipher Encryption in SIV mode. Use to decrypt data encrypted with the *aes_encrypt()* method. The encrypted data attribute must contain a Ciphertext and 16-byte tag.

A nonce is not used, so data is encrypted deterministic, in SIV mode this doesn't reduce security. (see https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html#siv-mode)

Parameters

- **encrypted_data** (*bytes*) – Data to decrypt. Must consist of a ciphertext and 16 byte tag.
- **key** (*bytes*) – The cryptographic key, size must be 32 bytes because AES-128 is used as cipher

Return bytes

Ciphertext and 16 bytes tag

bitcoinlib.encoding.**aes_encrypt**(*data*, *key*)

Encrypt data using AES Symmetric Block cipher Encryption in SIV mode (see https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html#siv-mode)

A nonce is not used so data is encrypted deterministic, in SIV mode this doesn't reduce security.

Method returns a byte string with ciphertext and a 16-byte tag. The ciphertext has the same length as the data.

Data can be decrypted with the *aes_decrypt()* method

Parameters

- **data** (*bytes*) – Data to encrypt
- **key** (*bytes*) – The cryptographic key, size must be 32 bytes because AES-128 is used as cipher

Return bytes

Ciphertext and 16 bytes tag

bitcoinlib.encoding.**base58encode**(*inp*)

Convert bytes to base58 encode string

Parameters

inp (*bytes*) – Input string

Return str

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(
↪'xpub661MyMwAqRbcFnkbk13gaJba22ibnEdJS7KAMY99C4jBBHMxWaCBSTrTinNTc9G5LTFtUqbLpWnzY5yPTNEF9u8sB1kE
↪', 58, 16)

↪'0488b21e000000000000000000007d3cc6702f48bf618f3f14cce5ee2cacf3f70933345ee4710af6fa4a330cc7d503c045
↪'
```

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, str*) – 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 an output must contain an even number of characters. Sometimes handy for hex conversions.
- **output_as_list** (*bool*) – Always output as a list instead of string.

Return str, list

Base converted input as a 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.

Replaced by *signature_der_decode_bytes()* method

Parameters

- **signature** (*bytes*) – DER encoed 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.

Replaced by the *signature_der_encode()* method

Parameters

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

Return bytes

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

Get double SHA256 hash of string

Parameters

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

Return bytes, str

bitcoinlib.encoding.**hash160**(*string*)

Creates a RIPEMD-160 + SHA256 hash of the input string

Parameters
string (*bytes*) – Script

Return bytes
RIPEMD-160 hash of script

bitcoinlib.encoding.**int_to_varbyteint**(*inp*)

Convert integer to CompactSize Variable length integer in byte format.

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

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

Parameters
inp (*int*) – Integer to convert

Returns
byteint: 1-9 byte representation as integer

bitcoinlib.encoding.**normalize_string**(*string*)

Normalize a string to the default NFKD Unicode format See https://en.wikipedia.org/wiki/Unicode_equivalence#Normalization

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

Returns
string

bitcoinlib.encoding.**normalize_var**(*var*, *base=256*)

Convert variable to normalized value: - UTF-8 encoded bytes for string values - Integer value for numbers - Deepcopy of list for lists

Parameters

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

Returns
Normalized variable

Return type
bytes, int, list

bitcoinlib.encoding.**pubkeyhash_to_addr**(*pubkeyhash*, *prefix=None*, *encoding='base58'*, *witver=0*)

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
- **witver** (*int*) – Witness version used. Currently used for Taproot addresses with witver=1. Ignored for base58 addresses

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'*, *checksum_xor=1*)

Encode public key hash as bech32 encoded (segwit) address

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

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

For more information see the 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 it’s not standard.
- **checksum_xor** (*int*) – checksum 1 for bech32 v0 addresses and 0x2bc830a3 for bech32m v1+ addresses

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.**read_varbyteint_return**(*s*)

Read variable length integer from BytesIO stream. Return original converted bytes (to reconstruct transaction or script).

Parameters
s (*BytesIO*) – A binary stream

Return (int, bytes)

bitcoinlib.encoding.**ripemd160**(*string*)

bitcoinlib.encoding.**scrypt_hash**(*password*, *salt*, *key_len=64*, *N=16384*, *r=8*, *p=1*, *buflen=64*)

Wrapper for Scrypt method for scrypt or Cryptodome library

For documentation see methods in referring libraries

bitcoinlib.encoding.**sha256**(*string*, *as_hex=False*)

Get 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.**signature_der_decode**(*signature*)

Decode a DER encoded signature and extract an r and s value

The DER encoded signature must have the following format:
<sequence byte> <integer byte> r <integer byte> s

Parameters
signature (*bytes*) – DER encoded signature

Return (int, int)
Tuple with r and s value

bitcoinlib.encoding.**signature_der_decode_bytes**(*signature*)

Extract r and s value from DER encoded string and convert to bytes.

Parameters
signature (*bytes*) – DER encoded signature

Return bytes
Signature bytes encoded with r and s value

bitcoinlib.encoding.**signature_der_encode**(*r*, *s*)

Create DER encoded signature string with signature r and s value.

A DER encoded signature has the following format:
<sequence byte> <integer byte> r <integer byte> s

Parameters

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

Return bytes

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 the format of the input string is not known.

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

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

Returns
hexstring

bitcoinlib.encoding.**varbyteint_to_int**(*byteint*)

Convert CompactSize Variable length integer in byte format to integer.

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

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

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

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

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

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

```
>>> varstr(to_bytes('5468697320737472696e6720686173206c656e677468206f66203330
↪')).hex()
'1e5468697320737472696e6720686173206c656e677468206f66203330'
```

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.**get_key_structure_data**(*witness_type*, *multisig=False*, *purpose=None*, *encoding=None*)

Get data from wallet key structure. Provide witness_type and multisig to determine key path, purpose (BIP44 reference) and encoding.

Parameters

- **witness_type** (*str*) – Witness type used for transaction validation
- **multisig** (*bool*) – Multisig or single keys wallet, default is False: single key / 1-of-1 wallet
- **purpose** (*int*) – Overrule purpose found in wallet structure. Do not use unless you known what you are doing.
- **encoding** (*str*) – Overrule encoding found in wallet structure. Do not use unless you known what you are doing.

Returns

(key_path, purpose, encoding)

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

8.28.1.3 Module contents

8.29 Script types

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

They are defined in main.py

8.29.1 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

8.29.2 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

8.29.3 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.

CHAPTER

NINE

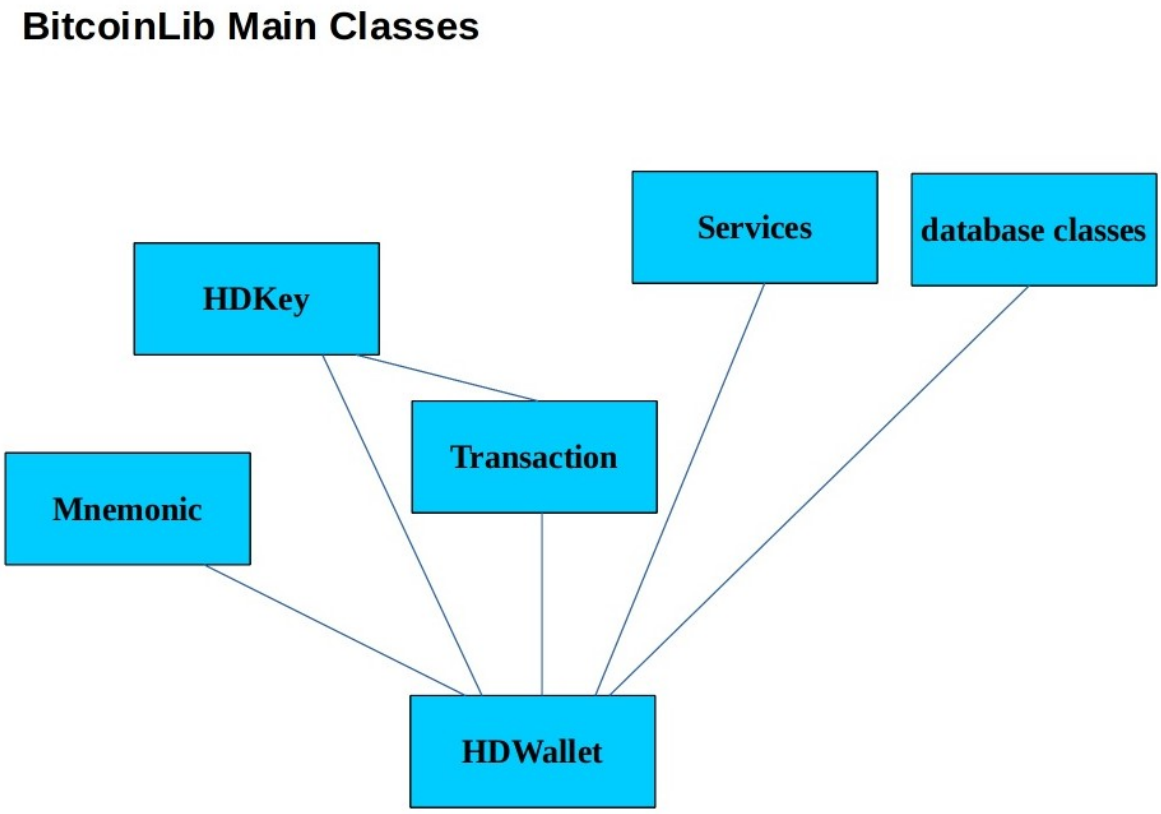
DISCLAIMER

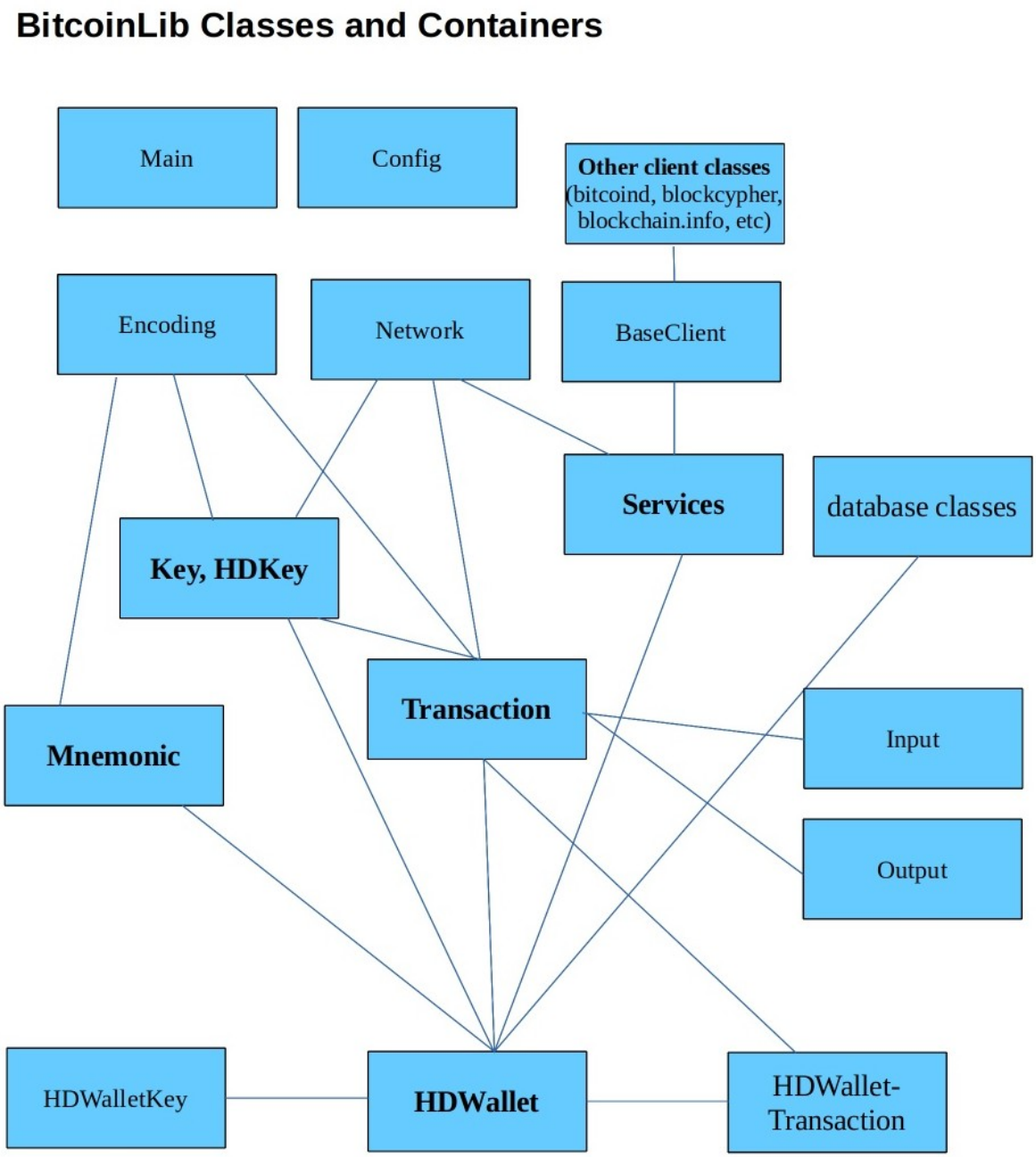
This library is still in development, please use at your own risk and test sufficiently before using it in a production environment.

CHAPTER

TEN

SCHEMATIC OVERVIEW




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

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y