beem Documentation

Release 0.20.20

Holger Nahrstaedt

Apr 18, 2020

Contents

Steem/Hive is a blockchain-based rewards platform for publishers to monetize content and grow community.

It is based on *Graphene* (tm), a blockchain technology stack (i.e. software) that allows for fast transactions and ascalable blockchain solution. In case of Steem/Hive, it comes with decentralized publishing of content.

The beem library has been designed to allow developers to easily access its routines and make use of the network without dealing with all the related blockchain technology and cryptography. This library can be used to do anything that is allowed according to the Steem/Hive blockchain protocol.

CHAPTER 1

About this Library

The purpose of *beem* is to simplify development of products and services that use the Steem blockchain. It comes with

- its own (bip32-encrypted) wallet
- RPC interface for the Blockchain backend
- JSON-based blockchain objects (accounts, blocks, prices, markets, etc)
- a simple to use yet powerful API
- transaction construction and signing
- push notification API
- *and more*

CHAPTER 2

Quickstart

Note:

All methods that construct and sign a transaction can be given the `account=` parameter to identify the user that is going to affected by this transaction, e.g.:

- the source account in a transfer
- the accout that buys/sells an asset in the exchange
- the account whos collateral will be modified

Important, If no `account` is given, then the `default_account` according to the settings in `config` is used instead.

```
from beem import Steem
steem = Steem()
steem.wallet.unlock("wallet-passphrase")
account = Account("test", steem_instance=steem)
account.transfer("<to>", "<amount>", "<asset>", "<memo>")
```

```
from beem.blockchain import Blockchain
blockchain = Blockchain()
for op in blockchain.stream():
    print(op)
```

```
from beem.block import Block
print(Block(1))
```

```
from beem.account import Account
account = Account("test")
print(account.balances)
for h in account.history():
    print(h)
```

```
from beem.steem import Steem
stm = Steem()
stm.wallet.wipe(True)
stm.wallet.create("wallet-passphrase")
stm.wallet.unlock("wallet-passphrase")
stm.wallet.addPrivateKey("512345678")
stm.wallet.lock()
```

```
from beem.market import Market
market = Market("SBD:STEEM")
print(market.ticker())
market.steem.wallet.unlock("wallet-passphrase")
print(market.sell(300, 100)  # sell 100 STEEM for 300 STEEM/SBD
```

CHAPTER 3

General

3.1 Installation

The minimal working python version is 2.7.x. or 3.4.x

beem can be installed parallel to python-steem.

For Debian and Ubuntu, please ensure that the following packages are installed:

```
sudo apt-get install build-essential libssl-dev python-dev curl
```

For Fedora and RHEL-derivatives, please ensure that the following packages are installed:

```
sudo yum install gcc openssl-devel python-devel
```

For OSX, please do the following:

```
brew install openssl
export CFLAGS="-I$(brew --prefix openssl)/include $CFLAGS"
export LDFLAGS="-L$(brew --prefix openssl)/lib $LDFLAGS"
```

For Termux on Android, please install the following packages:

```
pkg install clang openssl-dev python-dev
```

Install pip (https://pip.pypa.io/en/stable/installing/):

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py

python get-pip.py
```

Signing and Verify can be fasten (200 %) by installing cryptography. Install cryptography with pip:

```
pip install -U cryptography
```

Install beem with pip:

```
pip install -U beem
```

Sometimes this does not work. Please try:

```
pip3 install -U beem
```

or:

```
python -m pip install beem
```

3.1.1 Manual installation

You can install beem from this repository if you want the latest but possibly non-compiling version:

```
git clone https://github.com/holgern/beem.git
cd beem
python setup.py build

python setup.py install --user
```

Run tests after install:

```
pytest
```

3.1.2 Installing beem with conda-forge

Installing beem from the conda-forge channel can be achieved by adding conda-forge to your channels with:

```
conda config --add channels conda-forge
```

Once the conda-forge channel has been enabled, beem can be installed with:

```
conda install beem
```

Signing and Verify can be fasten (200 %) by installing cryptography:

```
conda install cryptography
```

3.1.3 Enable Logging

Add the following for enabling logging in your python script:

```
import logging
log = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)
```

When you want to see only critical errors, replace the last line by:

```
logging.basicConfig(level=logging.CRITICAL)
```

3.2 Quickstart

3.2.1 Hive/Steem blockchain

Nodes for using beem with the Hive blockchain can be set by the command line tool with:

```
beempy updatenodes --hive
```

Nodes for the Steem blockchain are set with

```
beempy updatenodes
```

Hive nodes can be set in a python script with

```
from beem import Steem
from beem.nodelist import NodeList
nodelist = NodeList()
nodelist.update_nodes()
nodes = nodelist.get_nodes(hive=True)
hive = Steem(node=nodes)
print(hive.is_hive)
```

Steem nodes can be set in a python script with

```
from beem import Steem
from beem.nodelist import NodeList
nodelist = NodeList()
nodelist.update_nodes()
nodes = nodelist.get_nodes(hive=False)
hive = Steem(node=nodes)
print(hive.is_hive)
```

3.2.2 Steem

The steem object is the connection to the Steem/Hive blockchain. By creating this object different options can be set.

Note: All init methods of beem classes can be given the `steem_instance=` parameter to assure that all objects use the same steem object. When the `steem_instance=` parameter is not used, the steem object is taken from get_shared_steem_instance().

beem.instance.shared_steem_instance() returns a global instance of steem. It can be set by *beem.instance.set_shared_steem_instance()* otherwise it is created on the first call.

```
from beem import Steem
from beem.account import Account
stm = Steem()
account = Account("test", steem_instance=stm)
```

```
from beem import Steem
from beem.account import Account
from beem.instance import set_shared_steem_instance
stm = Steem()
```

(continues on next page)

(continued from previous page)

```
set_shared_steem_instance(stm)
account = Account("test")
```

3.2.3 Wallet and Keys

Each account has the following keys:

- Posting key (allows accounts to post, vote, edit, resteem and follow/mute)
- Active key (allows accounts to transfer, power up/down, voting for witness, . . .)
- Memo key (Can be used to encrypt/decrypt memos)
- Owner key (The most important key, should not be used with beem)

Outgoing operation, which will be stored in the steem blockchain, have to be signed by a private key. E.g. Comment or Vote operation need to be signed by the posting key of the author or upvoter. Private keys can be provided to beem temporary or can be stored encrypted in a sql-database (wallet).

Note: Before using the wallet the first time, it has to be created and a password has to set. The wallet content is available to beempy and all python scripts, which have access to the sql database file.

Creating a wallet

`steem.wallet.wipe(True)` is only necessary when there was already an wallet created.

```
from beem import Steem
steem = Steem()
steem.wallet.wipe(True)
steem.wallet.unlock("wallet-passphrase")
```

Adding keys to the wallet

```
from beem import Steem
steem = Steem()
steem.wallet.unlock("wallet-passphrase")
steem.wallet.addPrivateKey("xxxxxxx")
steem.wallet.addPrivateKey("xxxxxxx")
```

Using the keys in the wallet

```
from beem import Steem
steem = Steem()
steem.wallet.unlock("wallet-passphrase")
account = Account("test", steem_instance=steem)
account.transfer("<to>", "<amount>", "<asset>", "<memo>")
```

Private keys can also set temporary

```
from beem import Steem
steem = Steem(keys=["xxxxxxxxx"])
account = Account("test", steem_instance=steem)
account.transfer("<to>", "<amount>", "<asset>", "<memo>")
```

3.2.4 Receiving information about blocks, accounts, votes, comments, market and witness

Receive all Blocks from the Blockchain

```
from beem.blockchain import Blockchain
blockchain = Blockchain()
for op in blockchain.stream():
    print(op)
```

Access one Block

```
from beem.block import Block
print(Block(1))
```

Access an account

```
from beem.account import Account
account = Account("test")
print(account.balances)
for h in account.history():
    print(h)
```

A single vote

```
from beem.vote import Vote
vote = Vote(u"@gtg/ffdhu-gtg-witness-log|gandalf")
print(vote.json())
```

All votes from an account

```
from beem.vote import AccountVotes
allVotes = AccountVotes("gtg")
```

Access a post

```
from beem.comment import Comment
comment = Comment("@gtg/ffdhu-gtg-witness-log")
print(comment["active_votes"])
```

Access the market

```
from beem.market import Market
market = Market("SBD:STEEM")
print(market.ticker())
```

Access a witness

```
from beem.witness import Witness
witness = Witness("gtg")
print(witness.is_active)
```

3.2.5 Sending transaction to the blockchain

Sending a Transfer

```
from beem import Steem
steem = Steem()
steem.wallet.unlock("wallet-passphrase")
account = Account("test", steem_instance=steem)
account.transfer("null", 1, "SBD", "test")
```

Upvote a post

```
from beem.comment import Comment
from beem import Steem
steem = Steem()
steem.wallet.unlock("wallet-passphrase")
comment = Comment("@gtg/ffdhu-gtg-witness-log", steem_instance=steem)
comment.upvote(weight=10, voter="test")
```

Publish a post to the blockchain

```
from beem import Steem
steem = Steem()
steem.wallet.unlock("wallet-passphrase")
steem.post("title", "body", author="test", tags=["a", "b", "c", "d", "e"], self_
↪vote=True)
```

Sell STEEM on the market

```
from beem.market import Market
from beem import Steem
steem.wallet.unlock("wallet-passphrase")
market = Market("SBD:STEEM", steem_instance=steem)
print(market.ticker())
market.steem.wallet.unlock("wallet-passphrase")
print(market.sell(300, 100))  # sell 100 STEEM for 300 STEEM/SBD
```

3.3 Tutorials

3.3.1 Bundle Many Operations

With Steem, you can bundle multiple operations into a single transactions. This can be used to do a multi-send (one sender, multiple receivers), but it also allows to use any other kind of operation. The advantage here is that the user can be sure that the operations are executed in the same order as they are added to the transaction.

A block can only include one vote operation and one comment operation from each sender.

```
from pprint import pprint
from beem import Steem
from beem.account import Account
from beem.comment import Comment
from beem.instance import set_shared_steem_instance

# not a real working key
wif = "5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3"

stm = Steem(
    bundle=True, # Enable bundle broadcast
    # nobroadcast=True, # Enable this for testing
    keys=[wif],
)
# Set stm as shared instance
set_shared_steem_instance(stm)

# Account and Comment will use now stm
account = Account("test")

# Post
c = Comment("@gtg/witness-gtg-log")

account.transfer("test1", 1, "STEEM")
account.transfer("test2", 1, "STEEM")
account.transfer("test3", 1, "SBD")
# Upvote post with 25%
c.upvote(25, voter=account)

pprint(stm.broadcast())
```

3.3.2 Use nobroadcast for testing

When using *nobroadcast=True* the transaction is not broadcasted but printed.

```
from pprint import pprint
from beem import Steem
from beem.account import Account
from beem.instance import set_shared_steem_instance

# Only for testing not a real working key
wif = "5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3"

# set nobroadcast always to True, when testing
testnet = Steem(
    nobroadcast=True, # Set to false when want to go live
    keys=[wif],
)
# Set testnet as shared instance
set_shared_steem_instance(testnet)

# Account will use now testnet
account = Account("test")

pprint(account.transfer("test1", 1, "STEEM"))
```

When executing the script above, the output will be similar to the following:

```
Not broadcasting anything!
{'expiration': '2018-05-01T16:16:57',
 'extensions': [],
 'operations': [['transfer',
                 {'amount': '1.000 STEEM',
                  'from': 'test',
                  'memo': '',
                  'to': 'test1'}]],
 'ref_block_num': 33020,
 'ref_block_prefix': 2523628005,
 'signatures': [
↪'1f57da50f241e70c229ed67b5d61898e792175c0f18ae29df8af414c46ae91eb5729c867b5d7dcc578368e7024e414c23
↪']}
```

3.3.3 Clear BlockchainObject Caching

Each BlockchainObject (Account, Comment, Vote, Witness, Amount, ...) has a glocal cache. This cache stores all objects and could lead to increased memory consumption. The global cache can be cleared with a *clear_cache()* call from any BlockchainObject.

```
from pprint import pprint
from beem.account import Account

account = Account("test")
pprint(str(account._cache))
account1 = Account("test1")
pprint(str(account._cache))
pprint(str(account1._cache))
account.clear_cache()
pprint(str(account._cache))
pprint(str(account1._cache))
```

3.3.4 Simple Sell Script

```
from beem import Steem
from beem.market import Market
from beem.price import Price
from beem.amount import Amount

# Only for testing not a real working key
wif = "5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3"

#
# Instantiate Steem (pick network via API node)
#
steem = Steem(
    nobroadcast=True,    # <<--- set this to False when you want to fire!
    keys=[wif]           # <<--- use your real keys, when going live!
)

#
# This defines the market we are looking at.
```

(continues on next page)

(continued from previous page)

```
# The first asset in the first argument is the *quote*
# Sell and buy calls always refer to the *quote*
#
market = Market("SBD:STEEM",
    steem_instance=steem
)

#
# Sell an asset for a price with amount (quote)
#
print(market.sell(
    Price(100.0, "STEEM/SBD"),
    Amount("0.01 SBD")
))
```

3.3.5 Sell at a timely rate

```
import threading
from beem import Steem
from beem.market import Market
from beem.price import Price
from beem.amount import Amount

# Only for testing not a real working key
wif = "5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3"

def sell():
    """ Sell an asset for a price with amount (quote)
    """
    print(market.sell(
        Price(100.0, "SBD/STEEM"),
        Amount("0.01 STEEM")
    ))

    threading.Timer(60, sell).start()


if __name__ == "__main__":
    #
    # Instantiate Steem (pick network via API node)
    #
    steem = Steem(
        nobroadcast=True,    # <<--- set this to False when you want to fire!
        keys=[wif]           # <<--- use your real keys, when going live!
    )

    #
    # This defines the market we are looking at.
    # The first asset in the first argument is the *quote*
    # Sell and buy calls always refer to the *quote*
    #
    market = Market("STEEM:SBD",
        steem_instance=steem
    )
```

(continues on next page)

(continued from previous page)

```
    sell()
```

3.3.6 Batch api calls on AppBase

Batch api calls are possible with AppBase RPC nodes. If you call a Api-Call with add_to_queue=True it is not submitted but stored in rpc_queue. When a call with add_to_queue=False (default setting) is started, the complete queue is sended at once to the node. The result is a list with replies.

```
from beem import Steem
stm = Steem("https://api.steemit.com")
stm.rpc.get_config(add_to_queue=True)
stm.rpc.rpc_queue
```

```
[{'method': 'condenser_api.get_config', 'jsonrpc': '2.0', 'params': [], 'id': 6}]
```

```
result = stm.rpc.get_block({"block_num":1}, api="block", add_to_queue=False)
len(result)
```

```
2
```

3.3.7 Account history

Lets calculate the curation reward from the last 7 days:

```
from datetime import datetime, timedelta
from beem.account import Account
from beem.amount import Amount

acc = Account("gtg")
stop = datetime.utcnow() - timedelta(days=7)
reward_vests = Amount("0 VESTS")
for reward in acc.history_reverse(stop=stop, only_ops=["curation_reward"]):
            reward_vests += Amount(reward['reward'])
curation_rewards_SP = acc.steem.vests_to_sp(reward_vests.amount)
print("Rewards are %.3f SP" % curation_rewards_SP)
```

Lets display all Posts from an account:

```
from beem.account import Account
from beem.comment import Comment
from beem.exceptions import ContentDoesNotExistsException
account = Account("holger80")
c_list = {}
for c in map(Comment, account.history(only_ops=["comment"])):
    if c.permlink in c_list:
      continue
    try:
        c.refresh()
    except ContentDoesNotExistsException:
        continue
    c_list[c.permlink] = 1
```

(continues on next page)

(continued from previous page)

```
    if not c.is_comment():
        print("%s " % c.title)
```

3.3.8 Transactionbuilder

Sign transactions with beem without using the wallet and build the transaction by hand. Example with one operation with and without the wallet:

```
from beem import Steem
from beem.transactionbuilder import TransactionBuilder
from beembase import operations
stm = Steem()
# Uncomment the following when using a wallet:
# stm.wallet.unlock("secret_password")
tx = TransactionBuilder(steem_instance=stm)
op = operations.Transfer(**{"from": 'user_a',
                            "to": 'user_b',
                            "amount": '1.000 SBD',
                            "memo": 'test 2'}))
tx.appendOps(op)
# Comment appendWif out and uncomment appendSigner when using a stored key from the␣
↪wallet
tx.appendWif('5.....') # `user_a`
# tx.appendSigner('user_a', 'active')
tx.sign()
tx.broadcast()
```

Example with signing and broadcasting two operations:

```
from beem import Steem
from beem.transactionbuilder import TransactionBuilder
from beembase import operations
stm = Steem()
# Uncomment the following when using a wallet:
# stm.wallet.unlock("secret_password")
tx = TransactionBuilder(steem_instance=stm)
ops = []
op = operations.Transfer(**{"from": 'user_a',
                            "to": 'user_b',
                            "amount": '1.000 SBD',
                            "memo": 'test 2'}))
ops.append(op)
op = operations.Vote(**{"voter": v,
                        "author": author,
                        "permlink": permlink,
                        "weight": int(percent * 100)})
ops.append(op)
tx.appendOps(ops)
# Comment appendWif out and uncomment appendSigner when using a stored key from the␣
↪wallet
tx.appendWif('5.....') # `user_a`
# tx.appendSigner('user_a', 'active')
tx.sign()
tx.broadcast()
```

3.4 beempy CLI

beempy is a convenient CLI utility that enables you to manage your wallet, transfer funds, check balances and more.

3.4.1 Using the Wallet

beempy lets you leverage your BIP38 encrypted wallet to perform various actions on your accounts.

The first time you use *beempy*, you will be prompted to enter a password. This password will be used to encrypt the *beempy* wallet, which contains your private keys.

You can change the password via *changewalletpassphrase* command.

```
beempy changewalletpassphrase
```

From this point on, every time an action requires your private keys, you will be prompted ot enter this password (from CLI as well as while using *steem* library).

To bypass password entry, you can set an environment variable `UNLOCK`.

```
UNLOCK=mysecretpassword beempy transfer <recipient_name> 100 STEEM
```

3.4.2 Common Commands

First, you may like to import your Steem account:

```
beempy importaccount
```

You can also import individual private keys:

```
beempy addkey <private_key>
```

Listing accounts:

```
beempy listaccounts
```

Show balances:

```
beempy balance account_name1 account_name2
```

Sending funds:

```
beempy transfer --account <account_name> <recipient_name> 100 STEEM memo
```

Upvoting a post:

```
beempy upvote --account <account_name> https://steemit.com/funny/@mynameisbrian/the-
↪content-stand-a-comic
```

3.4.3 Setting Defaults

For a more convenient use of `beempy` as well as the `beem` library, you can set some defaults. This is especially useful if you have a single Steem account.

```
beempy set default_account test
beempy set default_vote_weight 100

beempy config
 +---------------------+---------+
 | Key                 | Value   |
 +---------------------+---------+
 | default_account     | test    |
 | default_vote_weight | 100     |
 +---------------------+---------+
```

If you've set up your *default_account*, you can now send funds by omitting this field:

```
beempy transfer <recipient_name> 100 STEEM memo
```

3.4.4 Commands

3.4.5 beempy –help

You can see all available commands with `beempy --help`

```
 ~ % beempy --help
Usage: cli.py [OPTIONS] COMMAND1 [ARGS]... [COMMAND2 [ARGS]...]...

Options:
  -n, --node TEXT         URL for public Steem API (e.g.
                          https://api.steemit.com)
  -o, --offline           Prevent connecting to network
  -d, --no-broadcast      Do not broadcast
  -p, --no-wallet         Do not load the wallet
  -x, --unsigned          Nothing will be signed
  -e, --expires INTEGER   Delay in seconds until transactions are supposed to
                          expire (defaults to 60)
  -v, --verbose INTEGER   Verbosity
  --version               Show the version and exit.
  --help                  Show this message and exit.

Commands:
  addkey                   Add key to wallet When no [OPTION] is given,...
  allow                    Allow an account/key to interact with your...
  approvewitness           Approve a witnesses
  balance                  Shows balance
  broadcast                broadcast a signed transaction
  buy                      Buy STEEM or SBD from the internal market...
  cancel                   Cancel order in the internal market
  changewalletpassphrase   Change wallet password
  claimreward              Claim reward balances By default, this will...
  config                   Shows local configuration
  convert                  Convert STEEMDollars to Steem (takes a week...
  createwallet             Create new wallet with a new password
  currentnode              Sets the currently working node at the first...
  delkey                   Delete key from the wallet PUB is the public...
  delprofile               Delete a variable in an account's profile
  disallow                 Remove allowance an account/key to interact...
  disapprovewitness        Disapprove a witnesses
```

(continues on next page)

(continued from previous page)

```
  downvote                Downvote a post/comment POST is...
  follow                  Follow another account
  follower                Get information about followers
  following               Get information about following
  importaccount           Import an account using a passphrase
  info                    Show basic blockchain info General...
  interest                Get information about interest payment
  listaccounts            Show stored accounts
  listkeys                Show stored keys
  mute                    Mute another account
  muter                   Get information about muter
  muting                  Get information about muting
  newaccount              Create a new account
  nextnode                Uses the next node in list
  openorders              Show open orders
  orderbook               Obtain orderbook of the internal market
  parsewif                Parse a WIF private key without importing
  permissions             Show permissions of an account
  pingnode                Returns the answer time in milliseconds
  power                   Shows vote power and bandwidth
  powerdown               Power down (start withdrawing VESTS from...
  powerdownroute          Setup a powerdown route
  powerup                 Power up (vest STEEM as STEEM POWER)
  pricehistory            Show price history
  resteem                 Resteem an existing post
  sell                    Sell STEEM or SBD from the internal market...
  set                     Set default_account, default_vote_weight or...
  setprofile              Set a variable in an account's profile
  sign                    Sign a provided transaction with available...
  ticker                  Show ticker
  tradehistory            Show price history
  transfer                Transfer SBD/STEEM
  unfollow                Unfollow/Unmute another account
  updatememokey           Update an account's memo key
  upvote                  Upvote a post/comment POST is...
  votes                   List outgoing/incoming account votes
  walletinfo              Show info about wallet
  witnesscreate           Create a witness
  witnesses               List witnesses
  witnessupdate           Change witness properties
```

3.5 Configuration

The pysteem library comes with its own local configuration database that stores information like

- API node URLs
- default account name
- the encrypted master password
- the default voting weight
- if keyring should be used for unlocking the wallet

and potentially more.

You can access those variables like a regular dictionary by using

```
from beem import Steem
steem = Steem()
print(steem.config.items())
```

Keys can be added and changed like they are for regular dictionaries.

If you don't want to load the *beem.steem.Steem* class, you can load the configuration directly by using:

```
from beem.storage import configStorage as config
```

It is also possible to access the configuration with the commandline tool *beempy*:

```
beempy config
```

3.5.1 API node URLs

The default node URLs which will be used when *node* is *None* in *beem.steem.Steem* class is stored in *config["nodes"]* as string. The list can be get and set by:

```
from beem import Steem
steem = Steem()
node_list = steem.get_default_nodes()
node_list = node_list[1:] + [node_list[0]]
steem.set_default_nodes(node_list)
```

beempy can also be used to set nodes:

```
beempy set nodes wss://steemd.privex.io
beempy set nodes "['wss://steemd.privex.io', 'wss://gtg.steem.house:8090']"
```

The default nodes can be reset to the default value. When the first node does not answer, steem should be set to the offline mode. This can be done by:

```
beempy -o set nodes ""
```

or

```
from beem import Steem
steem = Steem(offline=True)
steem.set_default_nodes("")
```

3.5.2 Default account

The default account name is used in some functions, when no account name is given. It is also used in *beempy* for all account related functions.

```
from beem import Steem
steem = Steem()
steem.set_default_account("test")
steem.config["default_account"] = "test"
```

or by beempy with

```
beempy set default_account test
```

3.5.3 Default voting weight

The default vote weight is used for voting, when no vote weight is given.

```
from beem import Steem
steem = Steem()
steem.config["default_vote_weight"] = 100
```

or by beempy with

```
beempy set default_vote_weight 100
```

3.5.4 Setting password_storage

The password_storage can be set to:

- environment, this is the default setting. The master password for the wallet can be provided in the environment variable *UNLOCK*.
- keyring (when set with beempy, it asks for the wallet password)

```
beempy set password_storage environment
beempy set password_storage keyring
```

Environment variable for storing the master password

When *password_storage* is set to *environment*, the master password can be stored in *UNLOCK* for unlocking automatically the wallet.

Keyring support for beempy and wallet

In order to use keyring for storing the wallet password, the following steps are necessary:

- Install keyring: *pip install keyring*
- Change *password_storage* to *keyring* with *beempy* and enter the wallet password.

It also possible to change the password in the keyring by

```
python -m keyring set beem wallet
```

The stored master password can be displayed in the terminal by

```
python -m keyring get beem wallet
```

When keyring is set as *password_storage* and the stored password in the keyring is identically to the set master password of the wallet, the wallet is automatically unlocked everytime it is used.

Testing if unlocking works

Testing if the master password is correctly provided by keyring or the *UNLOCK* variable:

```
from beem import Steem
steem = Steem()
print(steem.wallet.locked())
```

When the output is False, automatic unlocking with keyring or the *UNLOCK* variable works. It can also tested by beempy with

```
beempy walletinfo --test-unlock
```

When no password prompt is shown, unlocking with keyring or the *UNLOCK* variable works.

3.6 Api Definitions

3.6.1 condenser_api

broadcast_block

not implemented

broadcast_transaction

```
from beem.transactionbuilder import TransactionBuilder
t = TransactionBuilder()
t.broadcast()
```

broadcast_transaction_synchronous

```
from beem.transactionbuilder import TransactionBuilder
t = TransactionBuilder()
t.broadcast()
```

get_account_bandwidth

```
from beem.account import Account
account = Account("test")
account.get_account_bandwidth()
```

get_account_count

```
from beem.blockchain import Blockchain
b = Blockchain()
b.get_account_count()
```

get_account_history

```
from beem.account import Account
acc = Account("steemit")
for h in acc.get_account_history(1,0):
    print(h)
```

get_account_reputations

```
from beem.blockchain import Blockchain
b = Blockchain()
for h in b.get_account_reputations():
    print(h)
```

get_account_votes

```
from beem.account import Account
acc = Account("gtg")
for h in acc.get_account_votes():
    print(h)
```

get_active_votes

```
from beem.vote import ActiveVotes
acc = Account("gtg")
post = acc.get_feed(0,1)[0]
a = ActiveVotes(post["authorperm"])
a.printAsTable()
```

get_active_witnesses

```
from beem.witness import Witnesses
w = Witnesses()
w.printAsTable()
```

get_block

```
from beem.block import Block
print(Block(1))
```

get_block_header

```
from beem.block import BlockHeader
print(BlockHeader(1))
```

get_blog

```
from beem.account import Account
acc = Account("gtg")
for h in acc.get_blog():
    print(h)
```

get_blog_authors

```
from beem.account import Account
acc = Account("gtg")
for h in acc.get_blog_authors():
    print(h)
```

get_blog_entries

```
from beem.account import Account
acc = Account("gtg")
for h in acc.get_blog_entries():
    print(h)
```

get_chain_properties

```
from beem import Steem
stm = Steem()
print(stm.get_chain_properties())
```

get_comment_discussions_by_payout

```
from beem.discussions import Query, Comment_discussions_by_payout
q = Query(limit=10)
for h in Comment_discussions_by_payout(q):
    print(h)
```

get_config

```
from beem import Steem
stm = Steem()
print(stm.get_config())
```

get_content

```
from beem.account import Account
from beem.comment import Comment
acc = Account("gtg")
post = acc.get_feed(0,1)[0]
print(Comment(post["authorperm"]))
```

get_content_replies

```
from beem.account import Account
from beem.comment import Comment
acc = Account("gtg")
post = acc.get_feed(0,1)[0]
c = Comment(post["authorperm"])
for h in c.get_replies():
    print(h)
```

get_conversion_requests

```
from beem.account import Account
acc = Account("gtg")
print(acc.get_conversion_requests())
```

get_current_median_history_price

```
from beem import Steem
stm = Steem()
print(stm.get_current_median_history())
```

get_discussions_by_active

```
from beem.discussions import Query, Discussions_by_active
q = Query(limit=10)
for h in Discussions_by_active(q):
    print(h)
```

get_discussions_by_author_before_date

```
from beem.discussions import Query, Discussions_by_author_before_date
for h in Discussions_by_author_before_date(limit=10, author="gtg"):
    print(h)
```

get_discussions_by_blog

```
from beem.discussions import Query, Discussions_by_blog
q = Query(limit=10)
for h in Discussions_by_blog(q):
    print(h)
```

get_discussions_by_cashout

```
from beem.discussions import Query, Discussions_by_cashout
q = Query(limit=10)
for h in Discussions_by_cashout(q):
    print(h)
```

get_discussions_by_children

```
from beem.discussions import Query, Discussions_by_children
q = Query(limit=10)
for h in Discussions_by_children(q):
    print(h)
```

get_discussions_by_comments

```
from beem.discussions import Query, Discussions_by_comments
q = Query(limit=10, start_author="steemit", start_permlink="firstpost")
for h in Discussions_by_comments(q):
    print(h)
```

get_discussions_by_created

```
from beem.discussions import Query, Discussions_by_created
q = Query(limit=10)
for h in Discussions_by_created(q):
    print(h)
```

get_discussions_by_feed

```
from beem.discussions import Query, Discussions_by_feed
q = Query(limit=10, tag="steem")
for h in Discussions_by_feed(q):
    print(h)
```

get_discussions_by_hot

```
from beem.discussions import Query, Discussions_by_hot
q = Query(limit=10, tag="steem")
for h in Discussions_by_hot(q):
    print(h)
```

get_discussions_by_promoted

```
from beem.discussions import Query, Discussions_by_promoted
q = Query(limit=10, tag="steem")
for h in Discussions_by_promoted(q):
    print(h)
```

get_discussions_by_trending

```
from beem.discussions import Query, Discussions_by_trending
q = Query(limit=10, tag="steem")
for h in Discussions_by_trending(q):
    print(h)
```

get_discussions_by_votes

```
from beem.discussions import Query, Discussions_by_votes
q = Query(limit=10)
for h in Discussions_by_votes(q):
    print(h)
```

get_dynamic_global_properties

```
from beem import Steem
stm = Steem()
print(stm.get_dynamic_global_properties())
```

get_escrow

```
from beem.account import Account
acc = Account("gtg")
print(acc.get_escrow())
```

get_expiring_vesting_delegations

```
from beem.account import Account
acc = Account("gtg")
print(acc.get_expiring_vesting_delegations())
```

get_feed

```
from beem.account import Account
acc = Account("gtg")
for f in acc.get_feed():
    print(f)
```

get_feed_entries

```
from beem.account import Account
acc = Account("gtg")
for f in acc.get_feed_entries():
    print(f)
```

get_feed_history

```
from beem import Steem
stm = Steem()
print(stm.get_feed_history())
```

get_follow_count

```
from beem.account import Account
acc = Account("gtg")
print(acc.get_follow_count())
```

get_followers

```
from beem.account import Account
acc = Account("gtg")
for f in acc.get_followers():
    print(f)
```

get_following

```
from beem.account import Account
acc = Account("gtg")
for f in acc.get_following():
    print(f)
```

get_hardfork_version

```
from beem import Steem
stm = Steem()
print(stm.get_hardfork_properties()["hf_version"])
```

get_key_references

```
from beem.account import Account
from beem.wallet import Wallet
acc = Account("gtg")
w = Wallet()
print(w.getAccountFromPublicKey(acc["posting"]["key_auths"][0][0]))
```

get_market_history

```
from beem.market import Market
m = Market()
for t in m.market_history():
    print(t)
```

get_market_history_buckets

```
from beem.market import Market
m = Market()
for t in m.market_history_buckets():
    print(t)
```

get_next_scheduled_hardfork

```
from beem import Steem
stm = Steem()
print(stm.get_hardfork_properties())
```

get_open_orders

```
from beem.market import Market
m = Market()
print(m.accountopenorders(account="gtg"))
```

get_ops_in_block

```
from beem.block import Block
b = Block(2e6, only_ops=True)
print(b)
```

get_order_book

```
from beem.market import Market
m = Market()
print(m.orderbook())
```

get_owner_history

```
from beem.account import Account
acc = Account("gtg")
print(acc.get_owner_history())
```

get_post_discussions_by_payout

```
from beem.discussions import Query, Post_discussions_by_payout
q = Query(limit=10)
for h in Post_discussions_by_payout(q):
    print(h)
```

get_potential_signatures

```
from beem.transactionbuilder import TransactionBuilder
from beem.blockchain import Blockchain
b = Blockchain()
block = b.get_current_block()
trx = block.json()["transactions"][0]
t = TransactionBuilder(trx)
print(t.get_potential_signatures())
```

get_reblogged_by

```
from beem.account import Account
from beem.comment import Comment
acc = Account("gtg")
post = acc.get_feed(0,1)[0]
c = Comment(post["authorperm"])
for h in c.get_reblogged_by():
    print(h)
```

get_recent_trades

```
from beem.market import Market
m = Market()
for t in m.recent_trades():
    print(t)
```

get_recovery_request

```
from beem.account import Account
acc = Account("gtg")
print(acc.get_recovery_request())
```

get_replies_by_last_update

```
from beem.discussions import Query, Replies_by_last_update
q = Query(limit=10, start_author="steemit", start_permlink="firstpost")
for h in Replies_by_last_update(q):
    print(h)
```

get_required_signatures

```
from beem.transactionbuilder import TransactionBuilder
from beem.blockchain import Blockchain
b = Blockchain()
block = b.get_current_block()
trx = block.json()["transactions"][0]
t = TransactionBuilder(trx)
print(t.get_required_signatures())
```

get_reward_fund

```
from beem import Steem
stm = Steem()
print(stm.get_reward_funds())
```

get_savings_withdraw_from

```
from beem.account import Account
acc = Account("gtg")
print(acc.get_savings_withdrawals(direction="from"))
```

get_savings_withdraw_to

```
from beem.account import Account
acc = Account("gtg")
print(acc.get_savings_withdrawals(direction="to"))
```

get_state

```
from beem.comment import RecentByPath
for p in RecentByPath(path="promoted"):
    print(p)
```

get_tags_used_by_author

```
from beem.account import Account
acc = Account("gtg")
print(acc.get_tags_used_by_author())
```

get_ticker

```
from beem.market import Market
m = Market()
print(m.ticker())
```

get_trade_history

```
from beem.market import Market
m = Market()
for t in m.trade_history():
    print(t)
```

get_transaction

```
from beem.blockchain import Blockchain
b = Blockchain()
print(b.get_transaction("6fde0190a97835ea6d9e651293e90c89911f933c"))
```

get_transaction_hex

```
from beem.blockchain import Blockchain
b = Blockchain()
block = b.get_current_block()
trx = block.json()["transactions"][0]
print(b.get_transaction_hex(trx))
```

get_trending_tags

```
from beem.discussions import Query, Trending_tags
q = Query(limit=10, start_tag="steemit")
for h in Trending_tags(q):
    print(h)
```

get_version

not implemented

get_vesting_delegations

```
from beem.account import Account
acc = Account("gtg")
for v in acc.get_vesting_delegations():
    print(v)
```

get_volume

```
from beem.market import Market
m = Market()
print(m.volume24h())
```

get_withdraw_routes

```
from beem.account import Account
acc = Account("gtg")
print(acc.get_withdraw_routes())
```

get_witness_by_account

```
from beem.witness import Witness
w = Witness("gtg")
print(w)
```

get_witness_count

```
from beem.witness import Witnesses
w = Witnesses()
print(w.witness_count)
```

get_witness_schedule

```
from beem import Steem
stm = Steem()
print(stm.get_witness_schedule())
```

get_witnesses

not implemented

get_witnesses_by_vote

```
from beem.witness import WitnessesRankedByVote
for w in WitnessesRankedByVote():
    print(w)
```

lookup_account_names

```
from beem.account import Account
acc = Account("gtg", full=False)
print(acc.json())
```

lookup_accounts

```
from beem.account import Account
acc = Account("gtg")
for a in acc.get_similar_account_names(limit=100):
    print(a)
```

lookup_witness_accounts

```
from beem.witness import ListWitnesses
for w in ListWitnesses():
    print(w)
```

verify_account_authority

disabled and not implemented

verify_authority

```
from beem.transactionbuilder import TransactionBuilder
from beem.blockchain import Blockchain
b = Blockchain()
block = b.get_current_block()
trx = block.json()["transactions"][0]
t = TransactionBuilder(trx)
t.verify_authority()
print("ok")
```

3.7 Modules

3.7.1 beem Modules

beem.account

class beem.account.**Account**(*account*, *full=True*, *lazy=False*, *steem_instance=None*)

Bases: *beem.blockchainobject.BlockchainObject*

This class allows to easily access Account data

Parameters

- **account_name** (*str*) – Name of the account
- **steem_instance** (Steem) – Steem instance
- **lazy** (*bool*) – Use lazy loading
- **full** (*bool*) – Obtain all account data including orders, positions, etc.

Returns Account data

Return type dictionary

Raises ***beem.exceptions.AccountDoesNotExistsException*** – if account does not exist

Instances of this class are dictionaries that come with additional methods (see below) that allow dealing with an account and its corresponding functions.

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("gtg", steem_instance=stm)
>>> print(account)
<Account gtg>
>>> print(account.balances)
```

Note: This class comes with its own caching function to reduce the load on the API server. Instances of this class can be refreshed with `Account.refresh()`. The cache can be cleared with `Account.clear_cache()`

allow (*foreign*, *weight=None*, *permission='posting'*, *account=None*, *threshold=None*, ***kwargs*)
Give additional access to an account by some other public key or account.

Parameters

- **foreign** (*str*) – The foreign account that will obtain access
- **weight** (*int*) – (optional) The weight to use. If not define, the threshold will be used. If the weight is smaller than the threshold, additional signatures will be required. (defaults to threshold)
- **permission** (*str*) – (optional) The actual permission to modify (defaults to `posting`)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)
- **threshold** (*int*) – (optional) The threshold that needs to be reached by signatures to be able to interact

approvewitness (*witness*, *account=None*, *approve=True*, ***kwargs*)
Approve a witness

Parameters

- **witness** (*list*) – list of Witness name or id
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

available_balances
List balances of an account. This call returns instances of *beem.amount.Amount*.

balances
Returns all account balances as dictionary

blog_history (*limit=None*, *start=-1*, *reblogs=True*, *account=None*)
Stream the blog entries done by an account in reverse time order.

Note: RPC nodes keep a limited history of entries for the user blog. Older blog posts of an account may not be available via this call due to these node limitations.

Parameters

- **limit** (*int*) – (optional) stream the latest *limit* blog entries. If unset (default), all available blog entries are streamed.
- **start** (*int*) – (optional) start streaming the blog entries from this index. *start=-1* (default) starts with the latest available entry.
- **reblogs** (*bool*) – (optional) if set *True* (default) reblogs / resteems are included. If set *False*, reblogs/resteems are omitted.
- **account** (*str*) – (optional) the account to stream blog entries for (defaults to `default_account`)

blog_history_reverse example:

```
from beem.account import Account
from beem import Steem
from beem.nodelist import NodeList
nodelist = NodeList()
nodelist.update_nodes()
stm = Steem(node=nodelist.get_nodes(hive=True))
acc = Account("steemitblog", steem_instance=stm)
for post in acc.blog_history(limit=10):
    print(post)
```

cancel_transfer_from_savings(*request_id*, *account=None*, ***kwargs*)
Cancel a withdrawal from 'savings' account.

Parameters

- **request_id** (*str*) – Identifier for tracking or cancelling the withdrawal
- **account** (*str*) – (optional) the source account for the transfer if not `default_account`

change_recovery_account(*new_recovery_account*, *account=None*, ***kwargs*)
Request a change of the recovery account.

Note: It takes 30 days until the change applies. Another request within this time restarts the 30 day period. Setting the current recovery account again cancels any pending change request.

Parameters

- **new_recovery_account** (*str*) – account name of the new recovery account
- **account** (*str*) – (optional) the account to change the recovery account for (defaults to `default_account`)

claim_reward_balance(*reward_steem=0*, *reward_sbd=0*, *reward_vests=0*, *account=None*, ***kwargs*)
Claim reward balances. By default, this will claim `all` outstanding balances. To bypass this behaviour, set desired claim amount by setting any of *reward_steem*, *reward_sbd* or *reward_vests*.

Parameters

- **reward_steem** (*str*) – Amount of STEEM you would like to claim.
- **reward_sbd** (*str*) – Amount of SBD you would like to claim.
- **reward_vests** (*str*) – Amount of VESTS you would like to claim.
- **account** (*str*) – The source account for the claim if not `default_account` is used.

comment_history(*limit=None*, *start_permlink=None*, *account=None*)
Stream the comments done by an account in reverse time order.

Note: RPC nodes keep a limited history of user comments for the user feed. Older comments may not be available via this call due to these node limitations.

Parameters

- **limit** (*int*) – (optional) stream the latest *limit* comments. If unset (default), all available comments are streamed.
- **start_permlink** (*str*) – (optional) start streaming the comments from this permlink. *start_permlink=None* (default) starts with the latest available entry.
- **account** (*str*) – (optional) the account to stream comments for (defaults to `default_account`)

comment_history_reverse example:

```
from beem.account import Account
from beem import Steem
from beem.nodelist import NodeList
nodelist = NodeList()
nodelist.update_nodes()
stm = Steem(node=nodelist.get_nodes(hive=True))
acc = Account("ned", steem_instance=stm)
for comment in acc.comment_history(limit=10):
    print(comment)
```

convert(*amount*, *account=None*, *request_id=None*)

Convert SteemDollars to Steem (takes 3.5 days to settle)

Parameters

- **amount** (*float*) – amount of SBD to convert
- **account** (*str*) – (optional) the source account for the transfer if not `default_account`
- **request_id** (*str*) – (optional) identifier for tracking the conversion‘

curation_stats()

Returns the curation reward of the last 24h and 7d and the average of the last 7 days

Returns Account curation

Return type dictionary

Sample output:

```
{
    '24hr': 0.0,
    '7d': 0.0,
    'avg': 0.0
}
```

delegate_vesting_shares(*to_account*, *vesting_shares*, *account=None*, ***kwargs*)

Delegate SP to another account.

Parameters

- **to_account** (*str*) – Account we are delegating shares to (delegatee).
- **vesting_shares** (*str*) – Amount of VESTS to delegate eg. *10000 VESTS*.
- **account** (*str*) – The source account (delegator). If not specified, `default_account` is used.

disallow(*foreign*, *permission='posting'*, *account=None*, *threshold=None*, ***kwargs*)

Remove additional access to an account by some other public key or account.

Parameters

- **foreign** (*str*) – The foreign account that will obtain access
- **permission** (*str*) – (optional) The actual permission to modify (defaults to `posting`)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)
- **threshold** (*int*) – The threshold that needs to be reached by signatures to be able to interact

disapprovewitness(*witness*, *account=None*, ***kwargs*)
Disapprove a witness

Parameters

- **witness** (*list*) – list of Witness name or id
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

ensure_full()
Ensure that all data are loaded

estimate_virtual_op_num(*blocktime*, *stop_diff=0*, *max_count=100*)
Returns an estimation of an virtual operation index for a given time or blockindex

Parameters

- **blocktime** (*int, datetime*) – start time or start block index from which account operation should be fetched
- **stop_diff** (*int*) – Sets the difference between last estimation and new estimation at which the estimation stops. Must not be zero. (default is 1)
- **max_count** (*int*) – sets the maximum number of iterations. -1 disables this (default 100)

```
utc = pytz.timezone('UTC')
start_time = utc.localize(datetime.utcnow()) - timedelta(days=7)
acc = Account("gtg")
start_op = acc.estimate_virtual_op_num(start_time)

b = Blockchain()
start_block_num = b.get_estimated_block_num(start_time)
start_op2 = acc.estimate_virtual_op_num(start_block_num)
```

```
acc = Account("gtg")
block_num = 21248120
start = t.time()
op_num = acc.estimate_virtual_op_num(block_num, stop_diff=1, max_count=10)
stop = t.time()
print(stop - start)
for h in acc.get_account_history(op_num, 0):
    block_est = h["block"]
print(block_est - block_num)
```

feed_history(*limit=None*, *start_author=None*, *start_permlink=None*, *account=None*)
Stream the feed entries of an account in reverse time order.

Note: RPC nodes keep a limited history of entries for the user feed. Older entries may not be available via this call due to these node limitations.

Parameters

- **limit** (*int*) – (optional) stream the latest *limit* feed entries. If unset (default), all available entries are streamed.
- **start_author** (*str*) – (optional) start streaming the replies from this author. *start_permlink=None* (default) starts with the latest available entry. If set, *start_permlink* has to be set as well.
- **start_permlink** (*str*) – (optional) start streaming the replies from this permlink. *start_permlink=None* (default) starts with the latest available entry. If set, *start_author* has to be set as well.
- **account** (*str*) – (optional) the account to get replies to (defaults to `default_account`)

comment_history_reverse example:

```
from beem.account import Account
from beem import Steem
from beem.nodelist import NodeList
nodelist = NodeList()
nodelist.update_nodes()
stm = Steem(node=nodelist.get_nodes(hive=True))
acc = Account("ned", steem_instance=stm)
for reply in acc.feed_history(limit=10):
    print(reply)
```

follow(*other, what=['blog'], account=None*)
Follow/Unfollow/Mute/Unmute another account's blog

Parameters

- **other** (*str*) – Follow this account
- **what** (*list*) – List of states to follow. `['blog']` means to follow `other`, `[]` means to unfollow/unmute `other`, `['ignore']` means to ignore `other`, (defaults to `['blog']`)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

getSimilarAccountNames(*limit=5*)
Deprecated, please use get_similar_account_names

get_account_bandwidth(*bandwidth_type=1, account=None*)

get_account_history(*index, limit, order=-1, start=None, stop=None, use_block_num=True, only_ops=[], exclude_ops=[], raw_output=False*)
Returns a generator for individual account transactions. This call can be used in a `for` loop.

Parameters

- **index** (*int*) – first number of transactions to return
- **limit** (*int*) – limit number of transactions to return
- **start** (*int, datetime*) – start number/date of transactions to return (*optional*)

- **stop** (*int, datetime*) – stop number/date of transactions to return (*optional*)
- **use_block_num** (*bool*) – if true, start and stop are block numbers, otherwise virtual OP count numbers.
- **only_ops** (*array*) – Limit generator by these operations (*optional*)
- **exclude_ops** (*array*) – Exclude thse operations from generator (*optional*)
- **batch_size** (*int*) – internal api call batch size (*optional*)
- **order** (*int*) – 1 for chronological, -1 for reverse order
- **raw_output** (*bool*) – if False, the output is a dict, which includes all values. Otherwise, the output is list.

Note: only_ops and exclude_ops takes an array of strings: The full list of operation ID's can be found in beembase.operationids.ops. Example: ['transfer', 'vote']

---

get_account_posts(*sort='feed'*, *account=None*, *observer=None*, *raw_data=False*)
Returns account feed

get_account_votes(*account=None*, *start_author=''*, *start_permlink=''*)
Returns all votes that the account has done

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=False), use_condenser=True)
>>> account = Account("beem.app", steem_instance=stm)
>>> account.get_account_votes()
```

get_balance(*balances*, *symbol*)
Obtain the balance of a specific Asset. This call returns instances of *beem.amount.Amount*. Available balance types:

- "available"
- "saving"
- "reward"
- "total"

Parameters

- **balances** (*str*) – Defines the balance type
- **symbol** (*str, dict*) – Can be "SBD", "STEEM" or "VESTS

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
```

(continues on next page)

(continued from previous page)

```
>>> account = Account("beem.app", steem_instance=stm)
>>> account.get_balance("rewards", "HBD")
0.000 HBD
```

get_balances()
Returns all account balances as dictionary

Returns Account balances

Return type dictionary

Sample output:

```
{
    'available': [102.985 STEEM, 0.008 SBD, 146273.695970 VESTS],
    'savings': [0.000 STEEM, 0.000 SBD],
    'rewards': [0.000 STEEM, 0.000 SBD, 0.000000 VESTS],
    'total': [102.985 STEEM, 0.008 SBD, 146273.695970 VESTS]
}
```

get_bandwidth()
Returns used and allocated bandwidth

Return type dictionary

Sample output:

```
{
    'used': 0,
    'allocated': 2211037
}
```

get_blog(*start_entry_id=0*, *limit=100*, *raw_data=False*, *short_entries=False*, *account=None*)
Returns the list of blog entries for an account

Parameters

- **start_entry_id** (*int*) – default is 0
- **limit** (*int*) – default is 100
- **raw_data** (*bool*) – default is False
- **short_entries** (*bool*) – when set to True and raw_data is True, get_blog_entries is used istead of get_blog
- **account** (*str*) – When set, a different account name is used (Default is object account name)

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("steemit", steem_instance=stm)
>>> account.get_blog(0, 1)
[<Comment @steemit/firstpost>]
```

get_blog_authors(*account=None*)
Returns a list of authors that have had their content reblogged on a given blog account

Parameters **account** (*str*) – When set, a different account name is used (Default is object account name)

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("steemit", steem_instance=stm)
>>> account.get_blog_authors()
```

get_blog_entries(*start_entry_id=0*, *limit=100*, *raw_data=True*, *account=None*)
Returns the list of blog entries for an account

Parameters

- **start_entry_id** (*int*) – default is 0
- **limit** (*int*) – default is 100
- **raw_data** (*bool*) – default is False
- **account** (*str*) – When set, a different account name is used (Default is object account name)

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("steemit", steem_instance=stm)
>>> entry = account.get_blog_entries(0, 1, raw_data=True)[0]
>>> print("%s - %s - %s" % (entry["author"], entry["permlink"], entry["blog
↪"]))
steemit - firstpost - steemit
```

get_conversion_requests(*account=None*)
Returns a list of SBD conversion request

Parameters **account** (*str*) – When set, a different account is used for the request (Default is object account name)

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("beem.app", steem_instance=stm)
```

(continues on next page)

(continued from previous page)

```
>>> account.get_conversion_requests()
[]
```

get_creator()
Returns the account creator or *None* if the account was mined

get_curation_reward(*days=7*)
Returns the curation reward of the last *days* days

Parameters **days** (*int*) – limit number of days to be included int the return value

get_downvote_manabar()
Return downvote manabar

get_downvoting_power(*with_regeneration=True*)
Returns the account downvoting power in the range of 0-100%

get_effective_vesting_shares()
Returns the effective vesting shares

get_escrow(*escrow_id=0*, *account=None*)
Returns the escrow for a certain account by id

Parameters

- **escrow_id** (*int*) – Id (only pre appbase)
- **account** (*str*) – When set, a different account is used for the request (Default is object account name)

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("beem.app", steem_instance=stm)
>>> account.get_escrow(1234)
[]
```

get_expiring_vesting_delegations(*after=None*, *limit=1000*, *account=None*)
Returns the expirations for vesting delegations.

Parameters

- **after** (*datetime*) – expiration after (only for pre appbase nodes)
- **limit** (*int*) – limits number of shown entries (only for pre appbase nodes)
- **account** (*str*) – When set, a different account is used for the request (Default is object account name)

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
```

(continues on next page)

(continued from previous page)

```
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("beem.app", steem_instance=stm)
>>> account.get_expiring_vesting_delegations()
[]
```

get_feed(*start_entry_id=0*, *limit=100*, *raw_data=False*, *short_entries=False*, *account=None*)

Returns a list of items in an account's feed

Parameters

- **start_entry_id** (*int*) – default is 0
- **limit** (*int*) – default is 100
- **raw_data** (*bool*) – default is False
- **short_entries** (*bool*) – when set to True and raw_data is True, get_feed_entries is used istead of get_feed
- **account** (*str*) – When set, a different account name is used (Default is object account name)

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("steemit", steem_instance=stm)
>>> account.get_feed(0, 1, raw_data=True)
[]
```

get_feed_entries(*start_entry_id=0*, *limit=100*, *raw_data=True*, *account=None*)

Returns a list of entries in an account's feed

Parameters

- **start_entry_id** (*int*) – default is 0
- **limit** (*int*) – default is 100
- **raw_data** (*bool*) – default is False
- **short_entries** (*bool*) – when set to True and raw_data is True, get_feed_entries is used istead of get_feed
- **account** (*str*) – When set, a different account name is used (Default is object account name)

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("steemit", steem_instance=stm)
```

(continues on next page)

(continued from previous page)

```
>>> account.get_feed_entries(0, 1)
[]
```

get_follow_count(*account=None*)

get_followers(*raw_name_list=True*, *limit=100*)
Returns the account followers as list

get_following(*raw_name_list=True*, *limit=100*)
Returns who the account is following as list

get_manabar()
Return manabar

get_manabar_recharge_time(*manabar*, *recharge_pct_goal=100*)
Returns the account mana recharge time in minutes

Parameters

- **manabar** (*dict*) – manabar dict from get_manabar() or get_rc_manabar()
- **recharge_pct_goal** (*float*) – mana recovery goal in percentage (default is 100)

get_manabar_recharge_time_str(*manabar*, *recharge_pct_goal=100*)
Returns the account manabar recharge time as string

Parameters

- **manabar** (*dict*) – manabar dict from get_manabar() or get_rc_manabar()
- **recharge_pct_goal** (*float*) – mana recovery goal in percentage (default is 100)

get_manabar_recharge_timedelta(*manabar*, *recharge_pct_goal=100*)
Returns the account mana recharge time as timedelta object

Parameters

- **manabar** (*dict*) – manabar dict from get_manabar() or get_rc_manabar()
- **recharge_pct_goal** (*float*) – mana recovery goal in percentage (default is 100)

get_muters(*raw_name_list=True*, *limit=100*)
Returns the account muters as list

get_mutings(*raw_name_list=True*, *limit=100*)
Returns who the account is muting as list

get_notifications(*only_unread=True*, *limit=100*, *raw_data=False*, *account=None*)
Returns account notifications

Parameters

- **only_unread** (*bool*) – When True, only unread notfications are shown
- **limit** (*int*) – When set, the number of shown notifications is limited (max limit = 100)
- **raw_data** (*bool*) – When True, the raw data from the api call is returned.
- **account** (*str*) – (optional) the account for which the notification should be received to (defaults to `default_account`)

get_owner_history(*account=None*)
Returns the owner history of an account.

Parameters **account** (*str*) – When set, a different account is used for the request (Default is object account name)

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("beem.app", steem_instance=stm)
>>> account.get_owner_history()
[]
```

get_rc()
Return RC of account

get_rc_manabar()
Returns current_mana and max_mana for RC

get_recharge_time(*voting_power_goal=100*, *starting_voting_power=None*)
Returns the account voting power recharge time in minutes

Parameters

- **voting_power_goal** (*float*) – voting power goal in percentage (default is 100)
- **starting_voting_power** (*float*) – returns recharge time if current voting power is the provided value.

get_recharge_time_str(*voting_power_goal=100*, *starting_voting_power=None*)
Returns the account recharge time as string

Parameters

- **voting_power_goal** (*float*) – voting power goal in percentage (default is 100)
- **starting_voting_power** (*float*) – returns recharge time if current voting power is the provided value.

get_recharge_timedelta(*voting_power_goal=100*, *starting_voting_power=None*)
Returns the account voting power recharge time as timedelta object

Parameters

- **voting_power_goal** (*float*) – voting power goal in percentage (default is 100)
- **starting_voting_power** (*float*) – returns recharge time if current voting power is the provided value.

get_recovery_request(*account=None*)
Returns the recovery request for an account

Parameters **account** (*str*) – When set, a different account is used for the request (Default is object account name)

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
```

(continues on next page)

(continued from previous page)

```
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("beem.app", steem_instance=stm)
>>> account.get_recovery_request()
[]
```

get_reputation()
Returns the account reputation in the (steemit) normalized form

get_savings_withdrawals(*direction='from'*, *account=None*)
Returns the list of savings withdrawls for an account.

Parameters

- **account** (*str*) – When set, a different account is used for the request (Default is object account name)
- **direction** (*str*) – Can be either from or to (only non appbase nodes)

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("beem.app", steem_instance=stm)
>>> account.get_savings_withdrawals()
[]
```

get_similar_account_names(*limit=5*)
Returns `limit` account names similar to the current account name as a list

Parameters **limit** (*int*) – limits the number of accounts, which will be returned

Returns Similar account names as list

Return type list

This is a wrapper around *beem.blockchain.Blockchain.get_similar_account_names()* using the current account name as reference.

get_steem_power(*onlyOwnSP=False*)
Returns the account steem power

get_tags_used_by_author(*account=None*)
Returns a list of tags used by an author.

Parameters **account** (*str*) – When set, a different account is used for the request (Default is object account name)

Return type list

get_vesting_delegations(*start_account=''*, *limit=100*, *account=None*)
Returns the vesting delegations by an account.

Parameters

- **account** (*str*) – When set, a different account is used for the request (Default is object account name)

- **start_account** (*str*) – delegatee to start with, leave empty to start from the first by name
- **limit** (*int*) – maximum number of results to return

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("beem.app", steem_instance=stm)
>>> account.get_vesting_delegations()
[]
```

get_vests(*only_own_vests=False*)
Returns the account vests

get_vote(*comment*)
Returns a vote if the account has already voted for comment.

Parameters **comment** (*str, Comment*) – can be a Comment object or a authorpermlink

get_vote_pct_for_SBD(*sbd*, *voting_power=None*, *steem_power=None*, *not_broadcasted_vote=True*)
Returns the voting percentage needed to have a vote worth a given number of SBD.

If the returned number is bigger than 10000 or smaller than -10000, the given SBD value is too high for that account

Parameters **sbd** (*str, int, amount.Amount*) – The amount of SBD in vote value

get_voting_power(*with_regeneration=True*)
Returns the account voting power in the range of 0-100%

get_voting_value_SBD(*voting_weight=100*, *voting_power=None*, *steem_power=None*, *not_broadcasted_vote=True*)
Returns the account voting value in SBD

get_withdraw_routes(*account=None*)
Returns the withdraw routes for an account.

Parameters **account** (*str*) – When set, a different account is used for the request (Default is object account name)

Return type list

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("beem.app", steem_instance=stm)
>>> account.get_withdraw_routes()
[]
```

has_voted(*comment*)
Returns if the account has already voted for comment

Parameters **comment** (*str, Comment*) – can be a Comment object or a authorpermlink

history(*start=None*, *stop=None*, *use_block_num=True*, *only_ops=[]*, *exclude_ops=[]*, *batch_size=1000*, *raw_output=False*)

Returns a generator for individual account transactions. The earlist operation will be first. This call can be used in a `for` loop.

Parameters

- **start** (*int, datetime*) – start number/date of transactions to return (*optional*)
- **stop** (*int, datetime*) – stop number/date of transactions to return (*optional*)
- **use_block_num** (*bool*) – if true, start and stop are block numbers, otherwise virtual OP count numbers.
- **only_ops** (*array*) – Limit generator by these operations (*optional*)
- **exclude_ops** (*array*) – Exclude thse operations from generator (*optional*)
- **batch_size** (*int*) – internal api call batch size (*optional*)
- **raw_output** (*bool*) – if False, the output is a dict, which includes all values. Otherwise, the output is list.

Note: only_ops and exclude_ops takes an array of strings: The full list of operation ID's can be found in beembase.operationids.ops. Example: ['transfer', 'vote']

```
acc = Account("gtg")
max_op_count = acc.virtual_op_count()
# Returns the 100 latest operations
acc_op = []
for h in acc.history(start=max_op_count - 99, stop=max_op_count, use_block_
↪num=False):
    acc_op.append(h)
len(acc_op)
```

```
100
```

```
acc = Account("test")
max_block = 21990141
# Returns the account operation inside the last 100 block. This can be empty.
acc_op = []
for h in acc.history(start=max_block - 99, stop=max_block, use_block_
↪num=True):
    acc_op.append(h)
len(acc_op)
```

```
0
```

```
acc = Account("test")
start_time = datetime(2018, 3, 1, 0, 0, 0)
stop_time = datetime(2018, 3, 2, 0, 0, 0)
# Returns the account operation from 1.4.2018 back to 1.3.2018
acc_op = []
for h in acc.history(start=start_time, stop=stop_time):
    acc_op.append(h)
len(acc_op)
```

```
0
```

history_reverse(*start=None*, *stop=None*, *use_block_num=True*, *only_ops=[]*, *exclude_ops=[]*, *batch_size=1000*, *raw_output=False*)

Returns a generator for individual account transactions. The latest operation will be first. This call can be used in a `for` loop.

Parameters

- **start** (*int, datetime*) – start number/date of transactions to return. If negative the virtual_op_count is added. (*optional*)
- **stop** (*int, datetime*) – stop number/date of transactions to return. If negative the virtual_op_count is added. (*optional*)
- **use_block_num** (*bool*) – if true, start and stop are block numbers, otherwise virtual OP count numbers.
- **only_ops** (*array*) – Limit generator by these operations (*optional*)
- **exclude_ops** (*array*) – Exclude thse operations from generator (*optional*)
- **batch_size** (*int*) – internal api call batch size (*optional*)
- **raw_output** (*bool*) – if False, the output is a dict, which includes all values. Otherwise, the output is list.

Note: only_ops and exclude_ops takes an array of strings: The full list of operation ID's can be found in beembase.operationids.ops. Example: ['transfer', 'vote']

```
acc = Account("gtg")
max_op_count = acc.virtual_op_count()
# Returns the 100 latest operations
acc_op = []
for h in acc.history_reverse(start=max_op_count, stop=max_op_count - 99, use_
↪block_num=False):
    acc_op.append(h)
len(acc_op)
```

```
100
```

```
max_block = 21990141
acc = Account("test")
# Returns the account operation inside the last 100 block. This can be empty.
acc_op = []
for h in acc.history_reverse(start=max_block, stop=max_block-100, use_block_
↪num=True):
    acc_op.append(h)
len(acc_op)
```

```
0
```

```
acc = Account("test")
start_time = datetime(2018, 4, 1, 0, 0, 0)
stop_time = datetime(2018, 3, 1, 0, 0, 0)
# Returns the account operation from 1.4.2018 back to 1.3.2018
```

(continues on next page)

(continued from previous page)

```
acc_op = []
for h in acc.history_reverse(start=start_time, stop=stop_time):
    acc_op.append(h)
len(acc_op)
```

```
0
```

interest()
Calculate interest for an account

Parameters **account** (*str*) – Account name to get interest for

Return type dictionary

Sample output:

```
{
    'interest': 0.0,
    'last_payment': datetime.datetime(2018, 1, 26, 5, 50, 27, tzinfo=<UTC>),
    'next_payment': datetime.datetime(2018, 2, 25, 5, 50, 27, tzinfo=<UTC>),
    'next_payment_duration': datetime.timedelta(-65, 52132, 684026),
    'interest_rate': 0.0
}
```

is_fully_loaded
Is this instance fully loaded / e.g. all data available?

Return type bool

json()

json_metadata

list_all_subscriptions(*account=None*)
Returns all subscriptions

mark_notifications_as_read(*last_read=None*, *account=None*)
Broadcast a mark all notification as read custom_json

Parameters

- **last_read** (*str*) – When set, this datestring is used to set the mark as read date
- **account** (*str*) – (optional) the account to broadcast the custom_json to (defaults to `default_account`)

mute(*mute*, *account=None*)
Mute another account

Parameters

- **mute** (*str*) – Mute this account
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

name
Returns the account name

print_info(*force_refresh=False*, *return_str=False*, *use_table=False*, ***kwargs*)
Prints import information about the account

profile
Returns the account profile

refresh()
Refresh/Obtain an account's data from the API server

rep
Returns the account reputation

reply_history(*limit=None*, *start_author=None*, *start_permlink=None*, *account=None*)
Stream the replies to an account in reverse time order.

Note: RPC nodes keep a limited history of entries for the replies to an author. Older replies to an account may not be available via this call due to these node limitations.

Parameters

- **limit** (*int*) – (optional) stream the latest *limit* replies. If unset (default), all available replies are streamed.
- **start_author** (*str*) – (optional) start streaming the replies from this author. *start_permlink=None* (default) starts with the latest available entry. If set, *start_permlink* has to be set as well.
- **start_permlink** (*str*) – (optional) start streaming the replies from this permlink. *start_permlink=None* (default) starts with the latest available entry. If set, *start_author* has to be set as well.
- **account** (*str*) – (optional) the account to get replies to (defaults to `default_account`)

comment_history_reverse example:

```
from beem.account import Account
acc = Account("ned")
for reply in acc.reply_history(limit=10):
    print(reply)
```

reward_balances

saving_balances

set_withdraw_vesting_route(*to*, *percentage=100*, *account=None*, *auto_vest=False*, ***kwargs*)
Set up a vesting withdraw route. When vesting shares are withdrawn, they will be routed to these accounts based on the specified weights.

Parameters

- **to** (*str*) – Recipient of the vesting withdrawal
- **percentage** (*float*) – The percent of the withdraw to go to the 'to' account.
- **account** (*str*) – (optional) the vesting account
- **auto_vest** (*bool*) – Set to true if the 'to' account should receive the VESTS as VESTS, or false if it should receive them as STEEM. (defaults to `False`)

sp
Returns the accounts Steem Power

total_balances

transfer(*to*, *amount*, *asset*, *memo=''*, *account=None*, ***kwargs*)
Transfer an asset to another account.

Parameters

- **to** (*str*) – Recipient
- **amount** (*float*) – Amount to transfer
- **asset** (*str*) – Asset to transfer
- **memo** (*str*) – (optional) Memo, may begin with # for encrypted messaging
- **account** (*str*) – (optional) the source account for the transfer if not `default_account`

Transfer example:

```
from beem.account import Account
from beem import Steem
active_wif = "5xxxx"
stm = Steem(keys=[active_wif])
acc = Account("test", steem_instance=stm)
acc.transfer("test1", 1, "STEEM", "test")
```

transfer_from_savings(*amount*, *asset*, *memo*, *request_id=None*, *to=None*, *account=None*, ***kwargs*)
Withdraw SBD or STEEM from 'savings' account.

Parameters

- **amount** (*float*) – STEEM or SBD amount
- **asset** (*float*) – 'STEEM' or 'SBD'
- **memo** (*str*) – (optional) Memo
- **request_id** (*str*) – (optional) identifier for tracking or cancelling the withdrawal
- **to** (*str*) – (optional) the source account for the transfer if not `default_account`
- **account** (*str*) – (optional) the source account for the transfer if not `default_account`

transfer_to_savings(*amount*, *asset*, *memo*, *to=None*, *account=None*, ***kwargs*)
Transfer SBD or STEEM into a 'savings' account.

Parameters

- **amount** (*float*) – STEEM or SBD amount
- **asset** (*float*) – 'STEEM' or 'SBD'
- **memo** (*str*) – (optional) Memo
- **to** (*str*) – (optional) the source account for the transfer if not `default_account`
- **account** (*str*) – (optional) the source account for the transfer if not `default_account`

transfer_to_vesting(*amount*, *to=None*, *account=None*, ***kwargs*)
Vest STEEM

Parameters

- **amount** (*float*) – Amount to transfer
- **to** (*str*) – Recipient (optional) if not set equal to account
- **account** (*str*) – (optional) the source account for the transfer if not default_account

type_id = 2

unfollow(*unfollow*, *account=None*)
Unfollow/Unmute another account's blog

Parameters

- **unfollow** (*str*) – Unfollow/Unmute this account
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

update_account_jsonmetadata(*metadata*, *account=None*, ***kwargs*)
Update an account's profile in json_metadata using the posting key

Parameters

- **metadata** (*dict*) – The new metadata to use
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

update_account_keys(*new_password*, *account=None*, ***kwargs*)
Updates all account keys

This method does **not** add any private keys to your wallet but merely changes the memo public key.

Parameters

- **new_password** (*str*) – is used to derive the owner, active, posting and memo key
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

update_account_metadata(*metadata*, *account=None*, ***kwargs*)
Update an account's profile in json_metadata

Parameters

- **metadata** (*dict*) – The new metadata to use
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

update_account_profile(*profile*, *account=None*, ***kwargs*)
Update an account's profile in json_metadata

Parameters

- **profile** (*dict*) – The new profile to use
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

Sample profile structure:

```
{
    'name': 'Holger',
    'about': 'beem Developer',
```

(continues on next page)

(continued from previous page)

```
    'location': 'Germany',
    'profile_image': 'https://c1.staticflickr.com/5/4715/38733717165_
↪7070227c89_n.jpg',
    'cover_image': 'https://farm1.staticflickr.com/894/26382750057_69f5c8e568.
↪jpg',
    'website': 'https://github.com/holgern/beem'
}
```

```
from beem.account import Account
account = Account("test")
profile = account.profile
profile["about"] = "test account"
account.update_account_profile(profile)
```

update_memo_key(*key*, *account=None*, ***kwargs*)

Update an account's memo public key

This method does **not** add any private keys to your wallet but merely changes the memo public key.

Parameters

- **key** (*str*) – New memo public key
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

verify_account_authority(*keys*, *account=None*)

Returns true if the signers have enough authority to authorize an account.

Parameters

- **keys** (*list*) – public key
- **account** (*str*) – When set, a different account is used for the request (Default is object account name)

Return type dictionary

```
>>> from beem.account import Account
>>> from beem import Steem
>>> from beem.nodelist import NodeList
>>> nodelist = NodeList()
>>> nodelist.update_nodes()
>>> stm = Steem(node=nodelist.get_nodes(hive=True))
>>> account = Account("steemit", steem_instance=stm)
>>> print(account.verify_account_authority([
↪"STM7Q2rLBqzPzFeteQZewv9Lu3NLE69fZoLeL6YK59t7UmssCBNTU"])["valid"])
False
```

virtual_op_count(*until=None*)

Returns the number of individual account transactions

Return type list

vp

Returns the account voting power in the range of 0-100%

withdraw_vesting(*amount*, *account=None*, ***kwargs*)

Withdraw VESTS from the vesting account.

Parameters

- **amount** (*float*) – number of VESTS to withdraw over a period of 13 weeks
- **account** (*str*) – (optional) the source account for the transfer if not default_account

class beem.account.**Accounts**(*name_list*, *batch_limit=100*, *lazy=False*, *full=True*, *steem_instance=None*)

Bases: *beem.account.AccountsObject*

Obtain a list of accounts

Parameters

- **name_list** (*list*) – list of accounts to fetch
- **batch_limit** (*int*) – (optional) maximum number of accounts to fetch per call, defaults to 100
- **steem_instance** (Steem) – Steem() instance to use when accessing a RPCcreator = Account(creator, steem_instance=self)

class beem.account.**AccountsObject**

Bases: list

printAsTable()

print_summarize_table(*tag_type='Follower'*, *return_str=False*, ***kwargs*)

beem.aes

class beem.aes.**AESCipher**(*key*)

Bases: object

A classical AES Cipher. Can use any size of data and any size of password thanks to padding. Also ensure the coherence and the type of the data with a unicode to byte converter.

decrypt(*enc*)

encrypt(*raw*)

static str_to_bytes(*data*)

beem.amount

class beem.amount.**Amount**(*amount*, *asset=None*, *fixed_point_arithmetic=False*, *new_appbase_format=True*, *steem_instance=None*)

Bases: dict

This class deals with Amounts of any asset to simplify dealing with the tuple:

```
(amount, asset)
```

Parameters

- **args** (*list*) – Allows to deal with different representations of an amount
- **amount** (*float*) – Let's create an instance with a specific amount
- **asset** (*str*) – Let's you create an instance with a specific asset (symbol)
- **fixed_point_arithmetic** (*boolean*) – when set to True, all operation are fixed point operations and the amount is always be rounded down to the precision

- **steem_instance** (Steem) – Steem instance

Returns All data required to represent an Amount/Asset

Return type dict

Raises **ValueError** – if the data provided is not recognized

Way to obtain a proper instance:

- `args` can be a string, e.g.: "1 SBD"
- `args` can be a dictionary containing `amount` and `asset_id`
- `args` can be a dictionary containing `amount` and `asset`
- `args` can be a list of a `float` and `str` (symbol)
- `args` can be a list of a `float` and a *beem.asset.Asset*
- `amount` and `asset` are defined manually

An instance is a dictionary and comes with the following keys:

- `amount` (float)
- `symbol` (str)
- `asset` (instance of *beem.asset.Asset*)

Instances of this class can be used in regular mathematical expressions (`+-*/%`) such as:

```
from beem.amount import Amount
from beem.asset import Asset
a = Amount("1 STEEM")
b = Amount(1, "STEEM")
c = Amount("20", Asset("STEEM"))
a + b
a * 2
a += b
a /= 2.0
```

```
2.000 STEEM
2.000 STEEM
```

amount
Returns the amount as float

amount_decimal
Returns the amount as decimal

asset
Returns the asset as instance of `steem.asset.Asset`

copy()
Copy the instance and make sure not to use a reference

json()

symbol
Returns the symbol of the asset

tuple()

beem.amount.**check_asset**(*other*, *self*, *stm*)

beem.amount.**quantize**(*amount*, *precision*)

beem.asciichart

class beem.asciichart.**AsciiChart**(*height=None*, *width=None*, *offset=3*, *placeholder='{:8.2f} '*, *charset='utf8'*)

Bases: object

Can be used to plot price and trade history

Parameters

- **height** (*int*) – Height of the plot
- **width** (*int*) – Width of the plot
- **offset** (*int*) – Offset between tick strings and y-axis (default is 3)
- **placeholder** (*str*) – Defines how the numbers on the y-axes are formatted (default is '{:8.2f}')
- **charset** (*str*) – sets the charset for plotting, uft8 or ascii (default: utf8)

adapt_on_series(*series*)

Calculates the minimum, maximum and length from the given list

Parameters **series** (*list*) – time series to plot

```
from beem.asciichart import AsciiChart
chart = AsciiChart()
series = [1, 2, 3, 7, 2, -4, -2]
chart.adapt_on_series(series)
chart.new_chart()
chart.add_axis()
chart.add_curve(series)
print(str(chart))
```

add_axis()

Adds a y-axis to the canvas

```
from beem.asciichart import AsciiChart
chart = AsciiChart()
series = [1, 2, 3, 7, 2, -4, -2]
chart.adapt_on_series(series)
chart.new_chart()
chart.add_axis()
chart.add_curve(series)
print(str(chart))
```

add_curve(*series*)

Add a curve to the canvas

Parameters **series** (*list*) – List width float data points

```
from beem.asciichart import AsciiChart
chart = AsciiChart()
series = [1, 2, 3, 7, 2, -4, -2]
chart.adapt_on_series(series)
chart.new_chart()
chart.add_axis()
```

(continues on next page)

(continued from previous page)

```
chart.add_curve(series)
print(str(chart))
```

clear_data()
Clears all data

new_chart(*minimum=None*, *maximum=None*, *n=None*)
Clears the canvas

```
from beem.asciichart import AsciiChart
chart = AsciiChart()
series = [1, 2, 3, 7, 2, -4, -2]
chart.adapt_on_series(series)
chart.new_chart()
chart.add_axis()
chart.add_curve(series)
print(str(chart))
```

plot(*series*, *return_str=False*)
All in one function for plotting

```
from beem.asciichart import AsciiChart
chart = AsciiChart()
series = [1, 2, 3, 7, 2, -4, -2]
chart.plot(series)
```

set_parameter(*height=None*, *offset=None*, *placeholder=None*)
Can be used to change parameter

beem.asset

class beem.asset.**Asset**(*asset*, *lazy=False*, *full=False*, *steem_instance=None*)
Bases: *beem.blockchainobject.BlockchainObject*

Deals with Assets of the network.

Parameters

- **Asset** (*str*) – Symbol name or object id of an asset
- **lazy** (*bool*) – Lazy loading
- **full** (*bool*) – Also obtain bitasset-data and dynamic asset dat
- **steem_instance** (Steem) – Steem instance

Returns All data of an asset

Note: This class comes with its own caching function to reduce the load on the API server. Instances of this class can be refreshed with `Asset.refresh()`.

---

asset

precision

refresh()
Refresh the data from the API server

symbol

type_id = 3

beem.block

class beem.block.**Block**(*block*, *only_ops=False*, *only_virtual_ops=False*, *full=True*, *lazy=False*, *steem_instance=None*)

Bases: *beem.blockchainobject.BlockchainObject*

Read a single block from the chain

Parameters

- **block** (*int*) – block number
- **steem_instance** (Steem) – Steem instance
- **lazy** (*bool*) – Use lazy loading
- **only_ops** (*bool*) – Includes only operations, when set to True (default: False)
- **only_virtual_ops** (*bool*) – Includes only virtual operations (default: False)

Instances of this class are dictionaries that come with additional methods (see below) that allow dealing with a block and its corresponding functions.

When only_virtual_ops is set to True, only_ops is always set to True.

In addition to the block data, the block number is stored as self["id"] or self.identifier.

```
>>> from beem.block import Block
>>> block = Block(1)
>>> print(block)
<Block 1>
```

Note: This class comes with its own caching function to reduce the load on the API server. Instances of this class can be refreshed with `Account.refresh()`.

block_num
Returns the block number

json()

json_operations
Returns all block operations as list, all dates are strings.

json_transactions
Returns all transactions as list, all dates are strings.

operations
Returns all block operations as list

ops_statistics(*add_to_ops_stat=None*)
Returns a statistic with the occurrence of the different operation types

refresh()
Even though blocks never change, you freshly obtain its contents from an API with this method

time()
Return a datetime instance for the timestamp of this block

transactions
Returns all transactions as list

class beem.block.**BlockHeader**(*block*, *full=True*, *lazy=False*, *steem_instance=None*)
Bases: *beem.blockchainobject.BlockchainObject*

Read a single block header from the chain

Parameters

- **block** (*int*) – block number
- **steem_instance** (Steem) – Steem instance
- **lazy** (*bool*) – Use lazy loading

In addition to the block data, the block number is stored as self["id"] or self.identifier.

```
>>> from beem.block import BlockHeader
>>> block = BlockHeader(1)
>>> print(block)
<BlockHeader 1>
```

block_num
Returns the block number

json()

refresh()
Even though blocks never change, you freshly obtain its contents from an API with this method

time()
Return a datetime instance for the timestamp of this block

beem.blockchain

class beem.blockchain.**Blockchain**(*steem_instance=None*, *mode='irreversible'*, *max_block_wait_repetition=None*, *data_refresh_time_seconds=900*)
Bases: object

This class allows to access the blockchain and read data from it

Parameters

- **steem_instance** (Steem) – Steem instance
- **mode** (*str*) – (default) Irreversible block (irreversible) or actual head block (head)
- **max_block_wait_repetition** (*int*) – maximum wait repetition for next block where each repetition is block_interval long (default is 3)

This class let's you deal with blockchain related data and methods. Read blockchain related data:

Read current block and blockchain info

```
print(chain.get_current_block())
print(chain.steem.info())
```

Monitor for new blocks. When stop is not set, monitoring will never stop.

```
blocks = []
current_num = chain.get_current_block_num()
for block in chain.blocks(start=current_num - 99, stop=current_num):
    blocks.append(block)
len(blocks)
```

```
100
```

or each operation individually:

```
ops = []
current_num = chain.get_current_block_num()
for operation in chain.ops(start=current_num - 99, stop=current_num):
    ops.append(operation)
```

awaitTxConfirmation(*transaction*, *limit=10*)
Returns the transaction as seen by the blockchain after being included into a block

Parameters

- **transaction** (*dict*) – transaction to wait for
- **limit** (*int*) – (optional) number of blocks to wait for the transaction (default: 10)

Note: If you want instant confirmation, you need to instantiate class:*beem.blockchain.Blockchain* with `mode="head"`, otherwise, the call will wait until confirmed in an irreversible block.

Note: This method returns once the blockchain has included a transaction with the **same signature**. Even though the signature is not usually used to identify a transaction, it still cannot be forfeited and is derived from the transaction contented and thus identifies a transaction uniquely.

---

block_time(*block_num*)
Returns a datetime of the block with the given block number.

Parameters **block_num** (*int*) – Block number

block_timestamp(*block_num*)
Returns the timestamp of the block with the given block number as integer.

Parameters **block_num** (*int*) – Block number

blocks(*start=None*, *stop=None*, *max_batch_size=None*, *threading=False*, *thread_num=8*, *only_ops=False*, *only_virtual_ops=False*)
Yields blocks starting from `start`.

Parameters

- **start** (*int*) – Starting block
- **stop** (*int*) – Stop at this block
- **max_batch_size** (*int*) – only for appbase nodes. When not None, batch calls of are used. Cannot be combined with threading
- **threading** (*bool*) – Enables threading. Cannot be combined with batch calls
- **thread_num** (*int*) – Defines the number of threads, when *threading* is set.

- **only_ops** (*bool*) – Only yield operations (default: False). Cannot be combined with `only_virtual_ops=True`.
- **only_virtual_ops** (*bool*) – Only yield virtual operations (default: False)

Note: If you want instant confirmation, you need to instantiate class:*beem.blockchain.Blockchain* with `mode="head"`, otherwise, the call will wait until confirmed in an irreversible block.

---

find_change_recovery_account_requests(*accounts*)

Find pending *change_recovery_account* requests for one or more specific accounts.

Parameters **accounts** (*str/list*) – account name or list of account names to find *change_recovery_account* requests for.

Returns list of *change_recovery_account* requests for the given account(s).

Return type list

```
>>> from beem.blockchain import Blockchain
>>> from beem import Steem
>>> stm = Steem("https://api.steemit.com")
>>> blockchain = Blockchain(steem_instance=stm)
>>> ret = blockchain.find_change_recovery_account_requests('bott')
```

find_rc_accounts(*name*)

Returns the RC parameters of one or more accounts.

Parameters **name** (*str*) – account name to search rc params for (can also be a list of accounts)

Returns RC params

Return type list

```
>>> from beem.blockchain import Blockchain
>>> from beem import Steem
>>> stm = Steem("https://api.steemit.com")
>>> blockchain = Blockchain(steem_instance=stm)
>>> ret = blockchain.find_rc_accounts(["test"])
>>> len(ret) == 1
True
```

get_account_count()

Returns the number of accounts

get_account_reputations(*start=''*, *stop=''*, *steps=1000.0*, *limit=-1*, ***kwargs*)

Yields account reputation between start and stop.

Parameters

- **start** (*str*) – Start at this account name
- **stop** (*str*) – Stop at this account name
- **steps** (*int*) – Obtain `steps` ret with a single call from RPC

get_all_accounts(*start=''*, *stop=''*, *steps=1000.0*, *limit=-1*, ***kwargs*)

Yields account names between start and stop.

Parameters

- **start** (*str*) – Start at this account name

- **stop** (*str*) – Stop at this account name
- **steps** (*int*) – Obtain `steps` ret with a single call from RPC

get_current_block(*only_ops=False*, *only_virtual_ops=False*)
This call returns the current block

Parameters

- **only_ops** (*bool*) – Returns block with operations only, when set to True (default: False)
- **only_virtual_ops** (*bool*) – Includes only virtual operations (default: False)

Note: The block number returned depends on the `mode` used when instantiating from this class.

---

get_current_block_num()
This call returns the current block number

Note: The block number returned depends on the `mode` used when instantiating from this class.

---

get_estimated_block_num(*date*, *estimateForwards=False*, *accurate=True*)
This call estimates the block number based on a given date

Parameters **date** (*datetime*) – block time for which a block number is estimated

---

Note: The block number returned depends on the `mode` used when instantiating from this class.

```
>>> from beem.blockchain import Blockchain
>>> from datetime import datetime
>>> blockchain = Blockchain()
>>> block_num = blockchain.get_estimated_block_num(datetime(2019, 6, 18, 5 ,8,
↪ 27))
>>> block_num == 33898184
True
```

get_similar_account_names(*name*, *limit=5*)
Returns limit similar accounts with name as list

Parameters

- **name** (*str*) – account name to search similars for
- **limit** (*int*) – limits the number of accounts, which will be returned

Returns Similar account names as list

Return type list

```
>>> from beem.blockchain import Blockchain
>>> from beem import Steem
>>> stm = Steem("https://api.steemit.com")
>>> blockchain = Blockchain(steem_instance=stm)
>>> ret = blockchain.get_similar_account_names("test", limit=5)
>>> len(ret) == 5
True
```

get_transaction(*transaction_id*)
Returns a transaction from the blockchain

Parameters **transaction_id** (*str*) – transaction_id

get_transaction_hex(*transaction*)
Returns a hexdump of the serialized binary form of a transaction.

Parameters **transaction** (*dict*) – transaction

static hash_op(*event*)
This method generates a hash of blockchain operation.

is_irreversible_mode()

list_change_recovery_account_requests(*start=''*, *limit=1000*, *order='by_account'*)
List pending *change_recovery_account* requests.

Parameters

- **start** (*str/list*) – Start the listing from this entry. Leave empty to start from the beginning. If *order* is set to *by_account*, *start* has to be an account name. If *order* is set to *by_effective_date*, *start* has to be a list of [effective_on, account_to_recover], e.g. *start=['2018-12-18T01:46:24', 'bott']*.
- **limit** (*int*) – maximum number of results to return (default and maximum: 1000).
- **order** (*str*) – valid values are "by_account" (default) or "by_effective_date".

Returns list of *change_recovery_account* requests.

Return type list

```
>>> from beem.blockchain import Blockchain
>>> from beem import Steem
>>> stm = Steem("https://api.steemit.com")
>>> blockchain = Blockchain(steem_instance=stm)
>>> ret = blockchain.list_change_recovery_account_requests(limit=1)
```

ops(*start=None*, *stop=None*, *only_virtual_ops=False*, ***kwargs*)
Blockchain.ops() is deprecated. Please use Blockchain.stream() instead.

ops_statistics(*start*, *stop=None*, *add_to_ops_stat=None*, *with_virtual_ops=True*, *verbose=False*)
Generates statistics for all operations (including virtual operations) starting from `start`.

Parameters

- **start** (*int*) – Starting block
- **stop** (*int*) – Stop at this block, if set to None, the current_block_num is taken
- **add_to_ops_stat** (*dict*) – if set, the result is added to add_to_ops_stat
- **verbose** (*bool*) – if True, the current block number and timestamp is printed

This call returns a dict with all possible operations and their occurrence.

stream(*opNames=[]*, *raw_ops=False*, **args*, ***kwargs*)
Yield specific operations (e.g. comments) only

Parameters

- **opNames** (*array*) – List of operations to filter for
- **raw_ops** (*bool*) – When set to True, it returns the unmodified operations (default: False)

- **start** (*int*) – Start at this block
- **stop** (*int*) – Stop at this block
- **max_batch_size** (*int*) – only for appbase nodes. When not None, batch calls of are used. Cannot be combined with threading
- **threading** (*bool*) – Enables threading. Cannot be combined with batch calls
- **thread_num** (*int*) – Defines the number of threads, when *threading* is set.
- **only_ops** (*bool*) – Only yield operations (default: False) Cannot be combined with `only_virtual_ops=True`
- **only_virtual_ops** (*bool*) – Only yield virtual operations (default: False)

The dict output is formated such that `type` carries the operation type. Timestamp and block_num are taken from the block the operation was stored in and the other keys depend on the actual operation.

Note: If you want instant confirmation, you need to instantiate class:*beem.blockchain.Blockchain* with `mode="head"`, otherwise, the call will wait until confirmed in an irreversible block.

---

output when *raw_ops=False* is set:

```
{
    'type': 'transfer',
    'from': 'johngreenfield',
    'to': 'thundercurator',
    'amount': '0.080 SBD',
    'memo': 'https://steemit.com/lofi/@johngreenfield/lofi-joji-yeah-right',
    '_id': '6d4c5f2d4d8ef1918acaee4a8dce34f9da384786',
    'timestamp': datetime.datetime(2018, 5, 9, 11, 23, 6, tzinfo=<UTC>),
    'block_num': 22277588, 'trx_num': 35, 'trx_id':
↪'cf11b2ac8493c71063ec121b2e8517ab1e0e6bea'
}
```

output when *raw_ops=True* is set:

```
{
    'block_num': 22277588,
    'op':
        [
            'transfer',
                {
                    'from': 'johngreenfield', 'to': 'thundercurator',
                    'amount': '0.080 SBD',
                    'memo': 'https://steemit.com/lofi/@johngreenfield/lofi-
↪joji-yeah-right'
                }
        ],
        'timestamp': datetime.datetime(2018, 5, 9, 11, 23, 6, tzinfo=<UTC>)
}
```

wait_for_and_get_block(*block_number*, *blocks_waiting_for=None*, *only_ops=False*, *only_virtual_ops=False*, *block_number_check_cnt=-1*, *last_current_block_num=None*)

Get the desired block from the chain, if the current head block is smaller (for both head and irreversible) then we wait, but a maxmimum of blocks_waiting_for * max_block_wait_repetition time before failure.

Parameters

- **block_number** (*int*) – desired block number
- **blocks_waiting_for** (*int*) – difference between block_number and current head and defines how many blocks we are willing to wait, positive int (default: None)
- **only_ops** (*bool*) – Returns blocks with operations only, when set to True (default: False)
- **only_virtual_ops** (*bool*) – Includes only virtual operations (default: False)
- **block_number_check_cnt** (*int*) – limit the number of retries when greater than -1
- **last_current_block_num** (*int*) – can be used to reduce the number of get_current_block_num() api calls

class beem.blockchain.**Pool**(*thread_count*, *batch_mode=True*, *exception_handler=<function default_handler>*)

Bases: object

Pool of threads consuming tasks from a queue

abort(*block=False*)
Tell each worker that its done working

alive()
Returns True if any threads are currently running

done()
Returns True if not tasks are left to be completed

enqueue(*func*, **args*, ***kargs*)
Add a task to the queue

idle()
Returns True if all threads are waiting for work

join()
Wait for completion of all the tasks in the queue

results(*sleep_time=0*)
Get the set of results that have been processed, repeatedly call until done

run(*block=False*)
Start the threads, or restart them if you've aborted

class beem.blockchain.**Worker**(*name*, *queue*, *results*, *abort*, *idle*, *exception_handler*)

Bases: threading.Thread

Thread executing tasks from a given tasks queue

run()
Thread work loop calling the function with the params

beem.blockchain.**default_handler**(*name*, *exception*, **args*, ***kwargs*)

beem.blockchainobject

class beem.blockchainobject.**BlockchainObject**(*data*, *klass=None*, *space_id=1*, *object_id=None*, *lazy=False*, *use_cache=True*, *id_item=None*, *steem_instance=None*, **args*, ***kwargs*)

Bases: dict

cache()

static clear_cache()

clear_cache_from_expired_items()

get_cache_auto_clean()

get_cache_expiration()

getcache(*id*)

iscached(*id*)

items() → a set-like object providing a view on D's items

json()

set_cache_auto_clean(*auto_clean*)

set_cache_expiration(*expiration*)

space_id = 1

test_valid_objectid(*i*)

testid(*id*)

type_id = None

type_ids = []

class beem.blockchainobject.**ObjectCache**(*initial_data={}*, *default_expiration=10*, *auto_clean=True*)

Bases: dict

clear_expired_items()

get(*key*, *default*)
Return the value for key if key is in the dictionary, else default.

beem.comment

class beem.comment.**Comment**(*authorperm*, *use_tags_api=True*, *full=True*, *lazy=False*, *steem_instance=None*)

Bases: *beem.blockchainobject.BlockchainObject*

Read data about a Comment/Post in the chain

Parameters

- **authorperm** (*str*) – identifier to post/comment in the form of `@author/permlink`
- **use_tags_api** (*boolean*) – when set to False, list_comments from the database_api is used
- **steem_instance** (Steem) – *beem.steem.Steem* instance to use when accessing a RPC

```
>>> from beem.comment import Comment
>>> from beem.account import Account
>>> from beem import Steem
>>> stm = Steem()
>>> acc = Account("gtg", steem_instance=stm)
```

(continues on next page)

(continued from previous page)

```
>>> authorperm = acc.get_blog(limit=1)[0]["authorperm"]
>>> c = Comment(authorperm)
>>> postdate = c["created"]
>>> postdate_str = c.json()["created"]
```

author

authorperm

body

category

curation_penalty_compensation_SBD()
Returns The required post payout amount after 15 minutes which will compentsate the curation penalty, if voting earlier than 15 minutes

delete(*account=None*, *identifier=None*)
Delete an existing post/comment

Parameters

- **account** (*str*) – (optional) Account to use for deletion. If `account` is not defined, the `default_account` will be taken or a ValueError will be raised.
- **identifier** (*str*) – (optional) Identifier for the post to delete. Takes the form `@author/permlink`. By default the current post will be used.

Note: A post/comment can only be deleted as long as it has no replies and no positive rshares on it.

depth

downvote(*weight=100*, *voter=None*)
Downvote the post

Parameters

- **weight** (*float*) – (optional) Weight for posting (-100.0 - +100.0) defaults to -100.0
- **voter** (*str*) – (optional) Voting account

edit(*body*, *meta=None*, *replace=False*)
Edit an existing post

Parameters

- **body** (*str*) – Body of the reply
- **meta** (*json*) – JSON meta object that can be attached to the post. (optional)
- **replace** (*bool*) – Instead of calculating a *diff*, replace the post entirely (defaults to `False`)

estimate_curation_SBD(*vote_value_SBD*, *estimated_value_SBD=None*)
Estimates curation reward

Parameters

- **vote_value_SBD** (*float*) – The vote value in SBD for which the curation should be calculated

- **estimated_value_SBD** (*float*) – When set, this value is used for calculate the curation. When not set, the current post value is used.

get_all_replies(*parent=None*)
Returns all content replies

get_author_rewards()
Returns the author rewards.

Example:

```
{
    'pending_rewards': True,
    'payout_SP': 0.912 STEEM,
    'payout_SBD': 3.583 SBD,
    'total_payout_SBD': 7.166 SBD
}
```

get_beneficiaries_pct()
Returns the sum of all post beneficiaries in percentage

get_curation_penalty(*vote_time=None*)
If post is less than 5 minutes old, it will incur a curation reward penalty.

Parameters **vote_time** (*datetime*) – A vote time can be given and the curation penalty is calculated regarding the given time (default is None) When set to None, the current date is used.

Returns Float number between 0 and 1 (0.0 -> no penalty, 1.0 -> 100 % curation penalty)

Return type float

get_curation_rewards(*pending_payout_SBD=False*, *pending_payout_value=None*)
Returns the curation rewards. The split between creator/curator is currently 50%/50%.

Parameters

- **pending_payout_SBD** (*bool*) – If True, the rewards are returned in SBD and not in STEEM (default is False)
- **pending_payout_value** (*float, str*) – When not None this value instead of the current value is used for calculating the rewards

pending_rewards is True when the post is younger than 7 days. *unclaimed_rewards* is the amount of curation_rewards that goes to the author (self-vote or votes within the first 30 minutes). *active_votes* contains all voter with their curation reward.

Example:

```
{
    'pending_rewards': True, 'unclaimed_rewards': 0.245 STEEM,
    'active_votes': {
        'leprechaun': 0.006 STEEM, 'timcliff': 0.186 STEEM,
        'st3llar': 0.000 STEEM, 'crokkon': 0.015 STEEM, 'feedyourminnows': 0.
↪003 STEEM,
        'isnochys': 0.003 STEEM, 'loshcat': 0.001 STEEM, 'greenorange': 0.000␣
↪STEEM,
        'qustodian': 0.123 STEEM, 'jpphotography': 0.002 STEEM, 'thinkingmind
↪': 0.001 STEEM,
        'oups': 0.006 STEEM, 'mattockfs': 0.001 STEEM, 'holger80': 0.003␣
↪STEEM, 'michaelizer': 0.004 STEEM,
        'flugschwein': 0.010 STEEM, 'ulisessabeque': 0.000 STEEM, 'hakancelik
↪': 0.002 STEEM, 'sbi2': 0.008 STEEM,
```

(continues on next page)

(continued from previous page)

```
        'zcool': 0.000 STEEM, 'steemhq': 0.002 STEEM, 'rowdiya': 0.000 STEEM,
↪'qurator-tier-1-2': 0.012 STEEM
    }
}
```

get_parent (*children=None*)
Returns the parent post with depth == 0

get_reblogged_by (*identifier=None*)
Shows in which blogs this post appears

get_replies (*raw_data=False*, *identifier=None*)
Returns content replies

Parameters **raw_data** (*bool*) – When set to False, the replies will be returned as Comment class objects

get_rewards ()
Returns the total_payout, author_payout and the curator payout in SBD. When the payout is still pending, the estimated payout is given out.

Note: Potential beneficiary rewards were already deducted from the *author_payout* and the *total_payout*

Example::

```
{
    'total_payout': 9.956 SBD,
    'author_payout': 7.166 SBD,
    'curator_payout': 2.790 SBD
}
```

get_vote_with_curation (*voter=None*, *raw_data=False*, *pending_payout_value=None*)
Returns vote for voter. Returns None, if the voter cannot be found in *active_votes*.

Parameters

- **voter** (*str*) – Voter for which the vote should be returned
- **raw_data** (*bool*) – If True, the raw data are returned
- **pending_payout_SBD** (*float, str*) – When not None this value instead of the current value is used for calculating the rewards

get_votes (*raw_data=False*)
Returns all votes as ActiveVotes object

id

is_comment ()
Returns True if post is a comment

is_main_post ()
Returns True if main post, and False if this is a comment (reply).

is_pending ()
Returns if the payout is pending (the post/comment is younger than 7 days)

json ()

json_metadata

parent_author

parent_permlink

permlink

refresh()

reply(*body*, *title=''*, *author=''*, *meta=None*)
Reply to an existing post

Parameters

- **body** (*str*) – Body of the reply
- **title** (*str*) – Title of the reply post
- **author** (*str*) – Author of reply (optional) if not provided `default_user` will be used, if present, else a `ValueError` will be raised.
- **meta** (*json*) – JSON meta object that can be attached to the post. (optional)

resteem(*identifier=None*, *account=None*)
Resteem a post

Parameters

- **identifier** (*str*) – post identifier (@<account>/<permlink>)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

reward
Return the estimated total SBD reward.

time_elapsed()
Returns a timedelta on how old the post is.

title

type_id = 8

upvote(*weight=100*, *voter=None*)
Upvote the post

Parameters

- **weight** (*float*) – (optional) Weight for posting (-100.0 - +100.0) defaults to +100.0
- **voter** (*str*) – (optional) Voting account

vote(*weight*, *account=None*, *identifier=None*, ***kwargs*)
Vote for a post

Parameters

- **weight** (*float*) – Voting weight. Range: -100.0 - +100.0.
- **account** (*str*) – (optional) Account to use for voting. If `account` is not defined, the `default_account` will be used or a ValueError will be raised
- **identifier** (*str*) – Identifier for the post to vote. Takes the form `@author/permlink`.

class beem.comment.**RankedPosts**(*sort='trending'*, *tag=''*, *observer=''*, *lazy=False*, *full=True*, *steem_instance=None*)
Bases: `list`

Obtain a list of ranked posts

Parameters

- **account** (*str*) – Account name
- **steem_instance** (Steem) – Steem() instance to use when accesing a RPC

class beem.comment.**RecentByPath**(*path='trending'*, *category=None*, *lazy=False*, *full=True*, *steem_instance=None*)

Bases: list

Obtain a list of posts recent by path

Parameters

- **account** (*str*) – Account name
- **steem_instance** (Steem) – Steem() instance to use when accesing a RPC

class beem.comment.**RecentReplies**(*author*, *skip_own=True*, *lazy=False*, *full=True*, *steem_instance=None*)

Bases: list

Obtain a list of recent replies

Parameters

- **author** (*str*) – author
- **skip_own** (*bool*) – (optional) Skip replies of the author to him/herself. Default: True
- **steem_instance** (Steem) – Steem() instance to use when accesing a RPC

beem.conveyor

class beem.conveyor.**Conveyor**(*url='https://conveyor.steemit.com'*, *steem_instance=None*)

Bases: object

Class to access Steemit Conveyor instances: https://github.com/steemit/conveyor

Description from the official documentation:

- Feature flags: "Feature flags allows our apps (condenser mainly) to hide certain features behind flags."
- User data: "Conveyor is the central point for storing sensitive user data (email, phone, etc). No other services should store this data and should instead query for it here every time."
- User tags: "Tagging mechanism for other services, allows defining and assigning tags to accounts (or other identifiers) and querying for them."

Not contained in the documentation, but implemented and working:

- Draft handling: saving, listing and removing post drafts consisting of a post title and a body.

The underlying RPC authentication and request signing procedure is described here: https://github.com/steemit/rpc-auth

get_feature_flag(*account*, *flag*, *signing_account=None*)

Test if a specific feature flag is set for an account. The request has to be signed by the requested account or an admin account.

Parameters

- **account** (*str*) – requested account
- **flag** (*str*) – flag to be tested

- **signing_account** (*str*) – (optional) account to sign the request. If unset, *account* is used.

Example:

```
from beem import Steem
from beem.conveyor import Conveyor
s = Steem(keys=["5JPOSTINGKEY"])
c = Conveyor(steem_instance=s)
print(c.get_feature_flag('accountname', 'accepted_tos'))
```

get_feature_flags(*account*, *signing_account=None*)

Get the account's feature flags. The request has to be signed by the requested account or an admin account.

Parameters

- **account** (*str*) – requested account
- **signing_account** (*str*) – (optional) account to sign the request. If unset, *account* is used.

Example:

```
from beem import Steem
from beem.conveyor import Conveyor
s = Steem(keys=["5JPOSTINGKEY"])
c = Conveyor(steem_instance=s)
print(c.get_feature_flags('accountname'))
```

get_user_data(*account*, *signing_account=None*)

Get the account's email address and phone number. The request has to be signed by the requested account or an admin account.

Parameters

- **account** (*str*) – requested account
- **signing_account** (*str*) – (optional) account to sign the request. If unset, *account* is used.

Example:

```
from beem import Steem
from beem.conveyor import Conveyor
s = Steem(keys=["5JPOSTINGKEY"])
c = Conveyor(steem_instance=s)
print(c.get_user_data('accountname'))
```

healthcheck()

Get the Conveyor status

Sample output:

```
{
    'ok': True, 'version': '1.1.1-4d28e36-1528725174',
    'date': '2018-07-21T12:12:25.502Z'
}
```

list_drafts(*account*)

List all saved drafts from *account*

Parameters **account** (*str*) – requested account

Sample output:

```
{
    'jsonrpc': '2.0', 'id': 2, 'result': [
        {'title': 'draft-title', 'body': 'draft-body',
         'uuid': '06497e1e-ac30-48cb-a069-27e1672924c9'}
    ]
}
```

prehash_message (*timestamp*, *account*, *method*, *params*, *nonce*)
Prepare a hash for the Conveyor API request with SHA256 according to https://github.com/steemit/rpc-auth Hashing of *second* is then done inside *ecdsasig.sign_message()*.

Parameters

- **timestamp** (*str*) – valid iso8601 datetime ending in "Z"
- **account** (*str*) – valid steem blockchain account name
- **method** (*str*) – Conveyor method name to be called
- **param** (*bytes*) – base64 encoded request parameters
- **nonce** (*bytes*) – random 8 bytes

remove_draft (*account*, *uuid*)
Remove a draft from the Conveyor database

Parameters

- **account** (*str*) – requested account
- **uuid** (*str*) – draft identifier as returned from *list_drafts*

save_draft (*account*, *title*, *body*)
Save a draft in the Conveyor database

Parameters

- **account** (*str*) – requested account
- **title** (*str*) – draft post title
- **body** (*str*) – draft post body

set_user_data (*account*, *params*, *signing_account=None*)
Set the account's email address and phone number. The request has to be signed by an admin account.

Parameters

- **account** (*str*) – requested account
- **param** (*dict*) – user data to be set
- **signing_account** (*str*) – (optional) account to sign the request. If unset, *account* is used.

Example:

```
from beem import Steem
from beem.conveyor import Conveyor
s = Steem(keys=["5JADMINPOSTINGKEY"])
c = Conveyor(steem_instance=s)
userdata = {'email': 'foo@bar.com', 'phone':'+123456789'}
c.set_user_data('accountname', userdata, 'adminaccountname')
```

beem.discussions

class beem.discussions.**Comment_discussions_by_payout**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get comment_discussions_by_payout

Parameters

- **discussion_query** (Query) – Defines the parameter for searching posts
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False
- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Comment_discussions_by_payout
q = Query(limit=10)
for h in Comment_discussions_by_payout(q):
    print(h)
```

class beem.discussions.**Discussions**(*lazy=False*, *use_appbase=False*, *steem_instance=None*)

Bases: object

Get Discussions

Parameters **steem_instance** (Steem) – Steem instance

get_discussions(*discussion_type*, *discussion_query*, *limit=1000*, *raw_data=False*)

Get Discussions

Parameters

- **discussion_type** (*str*) – Defines the used discussion query
- **discussion_query** (Query) – Defines the parameter for searching posts
- **raw_data** (*bool*) – returns list of comments when False, default is False

```
from beem.discussions import Query, Discussions
query = Query(limit=51, tag="steemit")
discussions = Discussions()
count = 0
for d in discussions.get_discussions("tags", query, limit=200):
    print(("%d. " % (count + 1)) + str(d))
    count += 1
```

class beem.discussions.**Discussions_by_active**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

get_discussions_by_active

Parameters

- **discussion_query** (Query) – Defines the parameter searching posts
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False

- **steem_instance** (Steem) – Steem() instance to use when accesing a RPC

```
from beem.discussions import Query, Discussions_by_active
q = Query(limit=10)
for h in Discussions_by_active(q):
    print(h)
```

class beem.discussions.**Discussions_by_author_before_date**(*author=''*, *start_permlink=''*, *before_date='1970-01-01T00:00:00'*, *limit=100*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get Discussions by author before date

Note: To retrieve discussions before date, the time of creation of the discussion @author/start_permlink must be older than the specified before_date parameter.

Parameters

- **author** (*str*) – Defines the author *(required)*
- **start_permlink** (*str*) – Defines the permlink of a starting discussion
- **before_date** (*str*) – Defines the before date for query
- **limit** (*int*) – Defines the limit of discussions
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False
- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Discussions_by_author_before_date
for h in Discussions_by_author_before_date(limit=10, author="gtg"):
    print(h)
```

class beem.discussions.**Discussions_by_blog**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get discussions by blog

Parameters

- **discussion_query** (Query) – Defines the parameter searching posts, tag musst be set to a username
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False
- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Discussions_by_blog
q = Query(limit=10)
for h in Discussions_by_blog(q):
    print(h)
```

class beem.discussions.**Discussions_by_cashout**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get discussions_by_cashout. This query seems to be broken at the moment. The output is always empty.

Parameters

- **discussion_query** (Query) – Defines the parameter searching posts
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False
- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Discussions_by_cashout
q = Query(limit=10)
for h in Discussions_by_cashout(q):
    print(h)
```

class beem.discussions.**Discussions_by_children**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get discussions by children

Parameters

- **discussion_query** (Query) – Defines the parameter searching posts
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False
- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Discussions_by_children
q = Query(limit=10)
for h in Discussions_by_children(q):
    print(h)
```

class beem.discussions.**Discussions_by_comments**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get discussions by comments

Parameters

- **discussion_query** (Query) – Defines the parameter searching posts, start_author and start_permlink must be set.
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False

- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Discussions_by_comments
q = Query(limit=10, start_author="steemit", start_permlink="firstpost")
for h in Discussions_by_comments(q):
    print(h)
```

class beem.discussions.**Discussions_by_created**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get discussions_by_created

Parameters

- **discussion_query** (Query) – Defines the parameter for searching posts
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False
- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Discussions_by_created
q = Query(limit=10)
for h in Discussions_by_created(q):
    print(h)
```

class beem.discussions.**Discussions_by_feed**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get discussions by feed

Parameters

- **discussion_query** (Query) – Defines the parameter searching posts, tag musst be set to a username
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False
- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Discussions_by_feed
q = Query(limit=10, tag="steem")
for h in Discussions_by_feed(q):
    print(h)
```

class beem.discussions.**Discussions_by_hot**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get discussions by hot

Parameters

- **discussion_query** (Query) – Defines the parameter searching posts
- **use_appbase** (*bool*) – use condenser call when set to False, default is False

- **raw_data** (*bool*) – returns list of comments when False, default is False
- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Discussions_by_hot
q = Query(limit=10, tag="steem")
for h in Discussions_by_hot(q):
    print(h)
```

class beem.discussions.**Discussions_by_promoted**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get discussions by promoted

Parameters

- **discussion_query** (Query) – Defines the parameter searching posts
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False
- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Discussions_by_promoted
q = Query(limit=10, tag="steem")
for h in Discussions_by_promoted(q):
    print(h)
```

class beem.discussions.**Discussions_by_trending**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get Discussions by trending

Parameters

- **discussion_query** (Query) – Defines the parameter for searching posts
- **steem_instance** (Steem) – Steem instance
- **raw_data** (*bool*) – returns list of comments when False, default is False

```
from beem.discussions import Query, Discussions_by_trending
q = Query(limit=10, tag="steem")
for h in Discussions_by_trending(q):
    print(h)
```

class beem.discussions.**Discussions_by_votes**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get discussions_by_votes

Parameters

- **discussion_query** (Query) – Defines the parameter searching posts
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False

- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Discussions_by_votes
q = Query(limit=10)
for h in Discussions_by_votes(q):
    print(h)
```

class beem.discussions.**Post_discussions_by_payout**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Get post_discussions_by_payout

Parameters

- **discussion_query** (Query) – Defines the parameter for searching posts
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False
- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Post_discussions_by_payout
q = Query(limit=10)
for h in Post_discussions_by_payout(q):
    print(h)
```

class beem.discussions.**Query**(*limit=0*, *tag=''*, *truncate_body=0*, *filter_tags=[]*, *select_authors=[]*, *select_tags=[]*, *start_author=None*, *start_permlink=None*, *start_tag=None*, *parent_author=None*, *parent_permlink=None*, *start_parent_author=None*, *before_date=None*, *author=None*)

Bases: dict

Query to be used for all discussion queries

Parameters

- **limit** (*int*) – limits the number of posts
- **tag** (*str*) – tag query
- **truncate_body** (*int*) –
- **filter_tags** (*array*) –
- **select_authors** (*array*) –
- **select_tags** (*array*) –
- **start_author** (*str*) –
- **start_permlink** (*str*) –
- **start_tag** (*str*) –
- **parent_author** (*str*) –
- **parent_permlink** (*str*) –
- **start_parent_author** (*str*) –
- **before_date** (*str*) –

- **author** (*str*) – Author (see Discussions_by_author_before_date)

```
from beem.discussions import Query
query = Query(limit=10, tag="steemit")
```

class beem.discussions.**Replies_by_last_update**(*discussion_query*, *lazy=False*, *use_appbase=False*, *raw_data=False*, *steem_instance=None*)

Bases: list

Returns a list of replies by last update

Parameters

- **discussion_query** (Query) – Defines the parameter searching posts start_parent_author and start_permlink must be set.
- **use_appbase** (*bool*) – use condenser call when set to False, default is False
- **raw_data** (*bool*) – returns list of comments when False, default is False
- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Replies_by_last_update
q = Query(limit=10, start_parent_author="steemit", start_permlink="firstpost")
for h in Replies_by_last_update(q):
    print(h)
```

class beem.discussions.**Trending_tags**(*discussion_query*, *lazy=False*, *use_appbase=False*, *steem_instance=None*)

Bases: list

Returns the list of trending tags.

Parameters

- **discussion_query** (Query) – Defines the parameter searching posts, start_tag can be set. :param bool use_appbase: use condenser call when set to False, default is False
- **steem_instance** (Steem) – Steem instance

```
from beem.discussions import Query, Trending_tags
q = Query(limit=10, start_tag="")
for h in Trending_tags(q):
    print(h)
```

beem.exceptions

exception beem.exceptions.**AccountDoesNotExistsException**

Bases: Exception

The account does not exist

exception beem.exceptions.**AccountExistsException**

Bases: Exception

The requested account already exists

exception beem.exceptions.**AssetDoesNotExistsException**

Bases: Exception

The asset does not exist

exception beem.exceptions.**BatchedCallsNotSupported**
Bases: Exception

Batch calls do not work

exception beem.exceptions.**BlockDoesNotExistsException**
Bases: Exception

The block does not exist

exception beem.exceptions.**BlockWaitTimeExceeded**
Bases: Exception

Wait time for new block exceeded

exception beem.exceptions.**ContentDoesNotExistsException**
Bases: Exception

The content does not exist

exception beem.exceptions.**InsufficientAuthorityError**
Bases: Exception

The transaction requires signature of a higher authority

exception beem.exceptions.**InvalidAssetException**
Bases: Exception

An invalid asset has been provided

exception beem.exceptions.**InvalidMemoKeyException**
Bases: Exception

Memo key in message is invalid

exception beem.exceptions.**InvalidMessageSignature**
Bases: Exception

The message signature does not fit the message

exception beem.exceptions.**InvalidWifError**
Bases: Exception

The provided private Key has an invalid format

exception beem.exceptions.**MissingKeyError**
Bases: Exception

A required key couldn't be found in the wallet

exception beem.exceptions.**NoWalletException**
Bases: Exception

No Wallet could be found, please use *beem.wallet.Wallet.create()* to create a new wallet

exception beem.exceptions.**NoWriteAccess**
Bases: Exception

Cannot store to sqlite3 database due to missing write access

exception beem.exceptions.**OfflineHasNoRPCException**
Bases: Exception

When in offline mode, we don't have RPC

exception beem.exceptions.**RPCConnectionRequired**
Bases: Exception

An RPC connection is required

exception beem.exceptions.**VestingBalanceDoesNotExistsException**
Bases: Exception

Vesting Balance does not exist

exception beem.exceptions.**VoteDoesNotExistsException**
Bases: Exception

The vote does not exist

exception beem.exceptions.**VotingInvalidOnArchivedPost**
Bases: Exception

The transaction requires signature of a higher authority

exception beem.exceptions.**WalletExists**
Bases: Exception

A wallet has already been created and requires a password to be unlocked by means of *beem.wallet.Wallet.unlock()*.

exception beem.exceptions.**WalletLocked**
Bases: Exception

Wallet is locked

exception beem.exceptions.**WitnessDoesNotExistsException**
Bases: Exception

The witness does not exist

exception beem.exceptions.**WrongMasterPasswordException**
Bases: Exception

The password provided could not properly unlock the wallet

exception beem.exceptions.**WrongMemoKey**
Bases: Exception

The memo provided is not equal the one on the blockchain

beem.imageuploader

class beem.imageuploader.**ImageUploader**(*base_url='https://steemitimages.com'*, *challenge='ImageSigningChallenge'*, *steem_instance=None*)
Bases: object

upload(*image*, *account*, *image_name=None*)
Uploads an image

Parameters

- **image** (*str, bytes*) – path to the image or image in bytes representation which should be uploaded
- **account** (*str*) – Account which is used to upload. A posting key must be provided.
- **image_name** (*str*) – optional

```
from beem import Steem
from beem.imageuploader import ImageUploader
stm = Steem(keys=["5xxx"]) # private posting key
iu = ImageUploader(steem_instance=stm)
iu.upload("path/to/image.png", "account_name") # "private posting key belongs␣
↪to account_name
```

beem.instance

class beem.instance.**SharedInstance**
Bases: object

Singelton for the Steem Instance

config = {}

instance = None

beem.instance.**clear_cache**()
Clear Caches

beem.instance.**set_shared_config**(*config*)
This allows to set a config that will be used when calling shared_steem_instance and allows to define the configuration without requiring to actually create an instance

beem.instance.**set_shared_steem_instance**(*steem_instance*)
This method allows us to override default steem instance for all users of SharedInstance.instance.

Parameters **steem_instance** (Steem) – Steem instance

beem.instance.**shared_steem_instance**()
This method will initialize SharedInstance.instance and return it. The purpose of this method is to have offer single default steem instance that can be reused by multiple classes.

```
from beem.account import Account
from beem.instance import shared_steem_instance

account = Account("test")
# is equivalent with
account = Account("test", steem_instance=shared_steem_instance())
```

beem.market

class beem.market.**Market**(*base=None*, *quote=None*, *steem_instance=None*)
Bases: dict

This class allows to easily access Markets on the blockchain for trading, etc.

Parameters

- **steem_instance** (Steem) – Steem instance
- **base** (Asset) – Base asset
- **quote** (Asset) – Quote asset

Returns Blockchain Market

Return type dictionary with overloaded methods

Instances of this class are dictionaries that come with additional methods (see below) that allow dealing with a market and its corresponding functions.

This class tries to identify **two** assets as provided in the parameters in one of the following forms:

- `base` and `quote` are valid assets (according to *beem.asset.Asset*)
- `base:quote` separated with `:`
- `base/quote` separated with `/`
- `base-quote` separated with `-`

Note: Throughout this library, the `quote` symbol will be presented first (e.g. `STEEM:SBD` with `STEEM` being the quote), while the `base` only refers to a secondary asset for a trade. This means, if you call *beem.market.Market.sell()* or *beem.market.Market.buy()*, you will sell/buy **only quote** and obtain/pay **only base**.

---

accountopenorders(*account=None*, *raw_data=False*)
Returns open Orders

Parameters

- **account** (*Account*) – Account name or instance of Account to show orders for in this market
- **raw_data** (*bool*) – (optional) returns raw data if set True, or a list of Order() instances if False (defaults to False)

static btc_usd_ticker(*verbose=False*)
Returns the BTC/USD price from bitfinex, gdax, kraken, okcoin and bitstamp. The mean price is weighted by the exchange volume.

buy(*price*, *amount*, *expiration=None*, *killfill=False*, *account=None*, *orderid=None*, *returnOrderId=False*)
Places a buy order in a given market

Parameters

- **price** (*float*) – price denoted in `base`/`quote`
- **amount** (*number*) – Amount of `quote` to buy
- **expiration** (*number*) – (optional) expiration time of the order in seconds (defaults to 7 days)
- **killfill** (*bool*) – flag that indicates if the order shall be killed if it is not filled (defaults to False)
- **account** (*string*) – Account name that executes that order
- **returnOrderId** (*string*) – If set to “head” or “irreversible” the call will wait for the tx to appear in the head/irreversible block and add the key “orderid” to the tx output

Prices/Rates are denoted in ‘base’, i.e. the SBD_STEEM market is priced in STEEM per SBD.

Example: in the SBD_STEEM market, a price of 300 means a SBD is worth 300 STEEM

Note: All prices returned are in the **reversed** orientation as the market. I.e. in the STEEM/SBD market, prices are SBD per STEEM. That way you can multiply prices with *1.05* to get a +5%.

> **Warning:** Since buy orders are placed as limit-sell orders for the base asset, you may end up obtaining more of the buy asset than you placed the order for. Example:
>
> - You place and order to buy 10 SBD for 100 STEEM/SBD
> - This means that you actually place a sell order for 1000 STEEM in order to obtain **at least** 10 SBD
> - If an order on the market exists that sells SBD for cheaper, you will end up with more than 10 SBD

cancel(*orderNumbers*, *account=None*, ***kwargs*)
Cancels an order you have placed in a given market. Requires only the "orderNumbers".

Parameters **orderNumbers** (*int, list*) – A single order number or a list of order numbers

get_string(*separator=':'*)
Return a formated string that identifies the market, e.g. `STEEM:SBD`

Parameters **separator** (*str*) – The separator of the assets (defaults to `:`)

static hive_btc_ticker()
Returns the HIVE/BTC price from bittrex and upbit. The mean price is weighted by the exchange volume.

hive_usd_implied()
Returns the current HIVE/USD market price

market_history(*bucket_seconds=300*, *start_age=3600*, *end_age=0*, *raw_data=False*)
Return the market history (filled orders).

Parameters

- **bucket_seconds** (*int*) – Bucket size in seconds (see *returnMarketHistoryBuckets()*)
- **start_age** (*int*) – Age (in seconds) of the start of the window (default: 1h/3600)
- **end_age** (*int*) – Age (in seconds) of the end of the window (default: now/0)
- **raw_data** (*bool*) – (optional) returns raw data if set True

Example:

```
{
    'close_sbd': 2493387,
    'close_steem': 7743431,
    'high_sbd': 1943872,
    'high_steem': 5999610,
    'id': '7.1.5252',
    'low_sbd': 534928,
    'low_steem': 1661266,
    'open': '2016-07-08T11:25:00',
    'open_sbd': 534928,
    'open_steem': 1661266,
    'sbd_volume': 9714435,
    'seconds': 300,
    'steem_volume': 30088443
}
```

market_history_buckets()

orderbook(*limit=25*, *raw_data=False*)
Returns the order book for SBD/STEEM market.

Parameters **limit** (*int*) – Limit the amount of orders (default: 25)

Sample output (raw_data=False):

```
{
    'asks': [
        380.510 STEEM 460.291 SBD @ 1.209669 SBD/STEEM,
        53.785 STEEM 65.063 SBD @ 1.209687 SBD/STEEM
    ],
    'bids': [
        0.292 STEEM 0.353 SBD @ 1.208904 SBD/STEEM,
        8.498 STEEM 10.262 SBD @ 1.207578 SBD/STEEM
    ],
    'asks_date': [
        datetime.datetime(2018, 4, 30, 21, 7, 24, tzinfo=<UTC>),
        datetime.datetime(2018, 4, 30, 18, 12, 18, tzinfo=<UTC>)
    ],
    'bids_date': [
        datetime.datetime(2018, 4, 30, 21, 1, 21, tzinfo=<UTC>),
        datetime.datetime(2018, 4, 30, 20, 38, 21, tzinfo=<UTC>)
    ]
}
```

Sample output (raw_data=True):

```
{
    'asks': [
        {
            'order_price': {'base': '8.000 STEEM', 'quote': '9.618 SBD
↪'},
            'real_price': '1.20225000000000004',
            'steem': 4565,
            'sbd': 5488,
            'created': '2018-04-30T21:12:45'
        }
    ],
    'bids': [
        {
            'order_price': {'base': '10.000 SBD', 'quote': '8.333 STEEM
↪'},
            'real_price': '1.20004800192007677',
            'steem': 8333,
            'sbd': 10000,
            'created': '2018-04-30T20:29:33'
        }
    ]
}
```

Note: Each bid is an instance of class:*beem.price.Order* and thus carries the keys `base`, `quote` and `price`. From those you can obtain the actual amounts for sale

recent_trades(*limit=25*, *raw_data=False*)
Returns the order book for a given market. You may also specify “all” to get the orderbooks of all markets.

Parameters

- **limit** (*int*) – Limit the amount of orders (default: 25)
- **raw_data** (*bool*) – when False, FilledOrder objects will be returned

Sample output (raw_data=False):

```
[
    (2018-04-30 21:00:54+00:00) 0.267 STEEM 0.323 SBD @ 1.209738 SBD/
↪STEEM,
    (2018-04-30 20:59:30+00:00) 0.131 STEEM 0.159 SBD @ 1.213740 SBD/
↪STEEM,
    (2018-04-30 20:55:45+00:00) 0.093 STEEM 0.113 SBD @ 1.215054 SBD/
↪STEEM,
    (2018-04-30 20:55:30+00:00) 26.501 STEEM 32.058 SBD @ 1.209690 SBD/
↪STEEM,
    (2018-04-30 20:55:18+00:00) 2.108 STEEM 2.550 SBD @ 1.209677 SBD/
↪STEEM,
]
```

Sample output (raw_data=True):

```
[
    {'date': '2018-04-30T21:02:45', 'current_pays': '0.235 SBD', 'open_
↪pays': '0.194 STEEM'},
    {'date': '2018-04-30T21:02:03', 'current_pays': '24.494 SBD',
↪'open_pays': '20.248 STEEM'},
    {'date': '2018-04-30T20:48:30', 'current_pays': '175.464 STEEM',
↪'open_pays': '211.955 SBD'},
    {'date': '2018-04-30T20:48:30', 'current_pays': '0.999 STEEM',
↪'open_pays': '1.207 SBD'},
    {'date': '2018-04-30T20:47:54', 'current_pays': '0.273 SBD', 'open_
↪pays': '0.225 STEEM'},
]
```

Note: Each bid is an instance of *beem.price.Order* and thus carries the keys `base`, `quote` and `price`. From those you can obtain the actual amounts for sale

---

sell(*price*, *amount*, *expiration=None*, *killfill=False*, *account=None*, *orderid=None*, *returnOrderId=False*)

Places a sell order in a given market

Parameters

- **price** (*float*) – price denoted in `base`/`quote`
- **amount** (*number*) – Amount of `quote` to sell
- **expiration** (*number*) – (optional) expiration time of the order in seconds (defaults to 7 days)
- **killfill** (*bool*) – flag that indicates if the order shall be killed if it is not filled (defaults to False)
- **account** (*string*) – Account name that executes that order
- **returnOrderId** (*string*) – If set to “head” or “irreversible” the call will wait for the tx to appear in the head/irreversible block and add the key “orderid” to the tx output

Prices/Rates are denoted in ‘base’, i.e. the SBD_STEEM market is priced in STEEM per SBD.

Example: in the SBD_STEEM market, a price of 300 means a SBD is worth 300 STEEM

Note: All prices returned are in the **reversed** orientation as the market. I.e. in the STEEM/SBD market, prices are SBD per STEEM. That way you can multiply prices with *1.05* to get a +5%.

static steem_btc_ticker()
Returns the STEEM/BTC price from bittrex, binance, huobi and upbit. The mean price is weighted by the exchange volume.

steem_usd_implied()
Returns the current STEEM/USD market price

ticker(*raw_data=False*)
Returns the ticker for all markets.

Output Parameters:

- `latest`: Price of the order last filled
- `lowest_ask`: Price of the lowest ask
- `highest_bid`: Price of the highest bid
- `sbd_volume`: Volume of SBD
- `steem_volume`: Volume of STEEM
- `percent_change`: 24h change percentage (in %)

Note: Market is STEEM:SBD and prices are SBD per STEEM!

Sample Output:

```
{
    'highest_bid': 0.30100226633322913,
    'latest': 0.0,
    'lowest_ask': 0.3249636958897082,
    'percent_change': 0.0,
    'sbd_volume': 108329611.0,
    'steem_volume': 355094043.0
}
```

trade_history(*start=None*, *stop=None*, *intervall=None*, *limit=25*, *raw_data=False*)
Returns the trade history for the internal market

This function allows to fetch a fixed number of trades at fixed intervall times to reduce the call duration time. E.g. it is possible to receive the trades from the last 7 days, by fetching 100 trades each 6 hours.

When intervall is set to None, all trades are received between start and stop. This can take a while.

Parameters

- **start** (*datetime*) – Start date
- **stop** (*datetime*) – Stop date
- **intervall** (*timedelta*) – Defines the intervall
- **limit** (*int*) – Defines how many trades are fetched at each intervall point
- **raw_data** (*bool*) – when True, the raw data are returned

trades(*limit=100*, *start=None*, *stop=None*, *raw_data=False*)
Returns your trade history for a given market.

Parameters

- **limit** (*int*) – Limit the amount of orders (default: 100)
- **start** (*datetime*) – start time
- **stop** (*datetime*) – stop time

volume24h(*raw_data=False*)
Returns the 24-hour volume for all markets, plus totals for primary currencies.

Sample output:

```
{
    "STEEM": 361666.63617,
    "SBD": 1087.0
}
```

beem.memo

class beem.memo.**Memo**(*from_account=None*, *to_account=None*, *steem_instance=None*)
Bases: object

Deals with Memos that are attached to a transfer

Parameters

- **from_account** (Account) – Account that has sent the memo
- **to_account** (Account) – Account that has received the memo
- **steem_instance** (Steem) – Steem instance

A memo is encrypted with a shared secret derived from a private key of the sender and a public key of the receiver. Due to the underlying mathematics, the same shared secret can be derived by the private key of the receiver and the public key of the sender. The encrypted message is perturbed by a nonce that is part of the transmitted message.

```
from beem.memo import Memo
m = Memo("steemeu", "wallet.xeroc")
m.steem.wallet.unlock("secret")
enc = (m.encrypt("foobar"))
print(enc)
>> {'nonce': '17329630356955254641', 'message': '8563e2bb2976e0217806d642901a2855
↪'}
print(m.decrypt(enc))
>> foobar
```

To decrypt a memo, simply use

```
from beem.memo import Memo
m = Memo()
m.steem.wallet.unlock("secret")
print(m.decrypt(op_data["memo"]))
```

if op_data being the payload of a transfer operation.

Memo Keys

In Steem, memos are AES-256 encrypted with a shared secret between sender and receiver. It is derived from the memo private key of the sender and the memo public key of the receiver.

In order for the receiver to decode the memo, the shared secret has to be derived from the receiver's private key and the senders public key.

The memo public key is part of the account and can be retrieved with the *get_account* call:

```
get_account <accountname>
{
  [...]
  "options": {
    "memo_key": "GPH5TPTziKkLexhVKsQKtSpo4bAv5RnB8oXcG4sMHEwCcTf3r7dqE",
    [...]
  },
  [...]
}
```

while the memo private key can be dumped with *dump_private_keys*

Memo Message

The take the following form:

```
{
  "from": "GPH5mgup8evDqMnT86L7scVebRYDC2fwAWmygPEUL43LjstQegYCC",
  "to": "GPH5Ar4j53kFWuEZQ9XhxbAja4YXMPJ2EnUg5QcrdeMFYUNMMNJbe",
  "nonce": "13043867485137706821",
  "message": "d55524c37320920844ca83bb20c8d008"
}
```

The fields *from* and *to* contain the memo public key of sender and receiver. The *nonce* is a random integer that is used for the seed of the AES encryption of the message.

Encrypting a memo

The high level memo class makes use of the beem wallet to obtain keys for the corresponding accounts.

```
from beem.memo import Memo
from beem.account import Account

memoObj = Memo(
    from_account=Account(from_account),
    to_account=Account(to_account)
)
encrypted_memo = memoObj.encrypt(memo)
```

Decoding of a received memo

```
from getpass import getpass
from beem.block import Block
from beem.memo import Memo

# Obtain a transfer from the blockchain
block = Block(23755086)                    # block
transaction = block["transactions"][3]     # transactions
op = transaction["operations"][0]          # operation
op_id = op[0]                              # operation type
op_data = op[1]                            # operation payload
```

(continues on next page)

(continued from previous page)

```
# Instantiate Memo for decoding
memo = Memo()

# Unlock wallet
memo.unlock_wallet(getpass())

# Decode memo
# Raises exception if required keys not available in the wallet
print(memo.decrypt(op_data["transfer"]))
```

decrypt(*memo*)
Decrypt a memo

Parameters **memo** (*str*) – encrypted memo message

Returns encrypted memo

Return type str

encrypt(*memo*, *bts_encrypt=False*)
Encrypt a memo

Parameters **memo** (*str*) – clear text memo message

Returns encrypted memo

Return type str

unlock_wallet(**args*, ***kwargs*)
Unlock the library internal wallet

beem.message

class beem.message.**Message**(*message*, *steem_instance=None*)
Bases: object

sign(*account=None*, ***kwargs*)
Sign a message with an account's memo key

Parameters **account** (*str*) – (optional) the account that owns the bet (defaults to default_account)

Returns the signed message encapsulated in a known format

verify(***kwargs*)
Verify a message with an account's memo key

Parameters **account** (*str*) – (optional) the account that owns the bet (defaults to default_account)

Returns True if the message is verified successfully

Raises ***InvalidMessageSignature*** – if the signature is not ok

beem.nodelist

class beem.nodelist.**NodeList**
Bases: list

Returns HIVE/STEEM nodes as list

```
from beem.nodelist import NodeList
n = NodeList()
nodes_urls = n.get_nodes()
```

get_hive_nodes (*testnet=False*, *not_working=False*, *wss=True*, *https=True*)
Returns hive only nodes as list

Parameters

- **testnet** (*bool*) – when True, testnet nodes are included
- **not_working** (*bool*) – When True, all nodes including not working ones will be returned

get_nodes (*hive=False*, *exclude_limited=False*, *dev=False*, *testnet=False*, *testnetdev=False*, *wss=True*, *https=True*, *not_working=False*, *normal=True*, *appbase=True*)
Returns nodes as list

Parameters

- **hive** (*bool*) – When True, only HIVE nodes will be returned
- **exclude_limited** (*bool*) – When True, limited nodes are excluded
- **dev** (*bool*) – when True, dev nodes with version 0.19.11 are included
- **testnet** (*bool*) – when True, testnet nodes are included
- **testnetdev** (*bool*) – When True, testnet-dev nodes are included
- **not_working** (*bool*) – When True, all nodes including not working ones will be returned
- **normal** (*bool*) – deprecated
- **appbase** (*bool*) – deprecated

get_steem_nodes (*testnet=False*, *not_working=False*, *wss=True*, *https=True*)
Returns steem only nodes as list

Parameters

- **testnet** (*bool*) – when True, testnet nodes are included
- **not_working** (*bool*) – When True, all nodes including not working ones will be returned

get_testnet (*testnet=True*, *testnetdev=False*)
Returns testnet nodes

update_nodes (*weights=None*, *steem_instance=None*)
Reads metadata from fullnodeupdate and recalculates the nodes score

Parameters **weight** (*list, dict*) – can be used to weight the different benchmarks

```
from beem.nodelist import NodeList
nl = NodeList()
weights = [0, 0.1, 0.2, 1]
nl.update_nodes(weights)
weights = {'block': 0.1, 'history': 0.1, 'apicall': 1, 'config': 1}
nl.update_nodes(weights)
```

beem.notify

class beem.notify.**Notify**(*on_block=None*, *only_block_id=False*, *steem_instance=None*, *keep_alive=25*)

Bases: events.events.Events

Notifications on Blockchain events.

This modules allows yout to be notified of events taking place on the blockchain.

Parameters

- **on_block** (*fnt*) – Callback that will be called for each block received
- **steem_instance** (Steem) – Steem instance

Example

```
from pprint import pprint
from beem.notify import Notify

notify = Notify(
    on_block=print,
)
notify.listen()
```

close()

Cleanly close the Notify instance

listen()

This call initiates the listening/notification process. It behaves similar to run_forever().

process_block(*message*)

reset_subscriptions(*accounts=[]*)

Change the subscriptions of a running Notify instance

beem.price

class beem.price.**FilledOrder**(*order*, *steem_instance=None*, ***kwargs*)

Bases: *beem.price.Price*

This class inherits *beem.price.Price* but has the base and quote Amounts not only be used to represent the price (as a ratio of base and quote) but instead has those amounts represent the amounts of an actually filled order!

Parameters **steem_instance** (Steem) – Steem instance

Note: Instances of this class come with an additional date key that shows when the order has been filled!

---

json()

class beem.price.**Order**(*base*, *quote=None*, *steem_instance=None*, ***kwargs*)

Bases: *beem.price.Price*

This class inherits *beem.price.Price* but has the base and quote Amounts not only be used to represent the price (as a ratio of base and quote) but instead has those amounts represent the amounts of an actual order!

Parameters **steem_instance** (Steem) – Steem instance

Note: If an order is marked as deleted, it will carry the 'deleted' key which is set to `True` and all other data be `None`.

class beem.price.**Price**(*price=None*, *base=None*, *quote=None*, *base_asset=None*, *steem_instance=None*)

Bases: `dict`

This class deals with all sorts of prices of any pair of assets to simplify dealing with the tuple:

```
(quote, base)
```

each being an instance of *beem.amount.Amount*. The amount themselves define the price.

Note: The price (floating) is derived as `base/quote`

Parameters

- **args** (*list*) – Allows to deal with different representations of a price
- **base** (Asset) – Base asset
- **quote** (Asset) – Quote asset
- **steem_instance** (Steem) – Steem instance

Returns All data required to represent a price

Return type dictionary

Way to obtain a proper instance:

- `args` is a str with a price and two assets
- `args` can be a floating number and `base` and `quote` being instances of *beem.asset.Asset*
- `args` can be a floating number and `base` and `quote` being instances of `str`
- `args` can be dict with keys `price`, `base`, and `quote` (*graphene balances*)
- `args` can be dict with keys `base` and `quote`
- `args` can be dict with key `receives` (filled orders)
- `args` being a list of `[quote, base]` both being instances of *beem.amount.Amount*
- `args` being a list of `[quote, base]` both being instances of `str` (`amount symbol`)
- `base` and `quote` being instances of `beem.asset.Amount`

This allows instanciations like:

- `Price("0.315 SBD/STEEM")`
- `Price(0.315, base="SBD", quote="STEEM")`
- `Price(0.315, base=Asset("SBD"), quote=Asset("STEEM"))`
- `Price({"base": {"amount": 1, "asset_id": "SBD"}, "quote": {"amount": 10, "asset_id": "SBD"}})`
- `Price(quote="10 STEEM", base="1 SBD")`
- `Price("10 STEEM", "1 SBD")`

- `Price(Amount("10 STEEM"), Amount("1 SBD"))`
- `Price(1.0, "SBD/STEEM")`

Instances of this class can be used in regular mathematical expressions (`+-*/%`) such as:

```
>>> from beem.price import Price
>>> from beem import Steem
>>> stm = Steem("https://api.steemit.com")
>>> Price("0.3314 SBD/STEEM", steem_instance=stm) * 2
0.662804 SBD/STEEM
>>> Price(0.3314, "SBD", "STEEM", steem_instance=stm)
0.331402 SBD/STEEM
```

as_base(*base*)

Returns the price instance so that the base asset is `base`.

Note: This makes a copy of the object!

```
>>> from beem.price import Price
>>> from beem import Steem
>>> stm = Steem("https://api.steemit.com")
>>> Price("0.3314 SBD/STEEM", steem_instance=stm).as_base("STEEM")
3.017483 STEEM/SBD
```

as_quote(*quote*)

Returns the price instance so that the quote asset is `quote`.

Note: This makes a copy of the object!

```
>>> from beem.price import Price
>>> from beem import Steem
>>> stm = Steem("https://api.steemit.com")
>>> Price("0.3314 SBD/STEEM", steem_instance=stm).as_quote("SBD")
3.017483 STEEM/SBD
```

copy() → a shallow copy of D

invert()

Invert the price (e.g. go from `SBD/STEEM` into `STEEM/SBD`)

```
>>> from beem.price import Price
>>> from beem import Steem
>>> stm = Steem("https://api.steemit.com")
>>> Price("0.3314 SBD/STEEM", steem_instance=stm).invert()
3.017483 STEEM/SBD
```

json()

market

Open the corresponding market

Returns Instance of *beem.market.Market* for the corresponding pair of assets.

symbols()

beem.price.**check_asset**(*other*, *self*, *stm*)

beem.rc

class beem.rc.**RC**(*steem_instance=None*)
Bases: object

account_create_dict(*account_create_dict*)
Calc RC costs for account create

account_update_dict(*account_update_dict*)
Calc RC costs for account update

claim_account(*tx_size=300*)
Claim account

comment(*tx_size=1000*, *permlink_length=10*, *parent_permlink_length=10*)
Calc RC for a comment

comment_dict(*comment_dict*)
Calc RC costs for a comment dict object

Example for calculating RC costs

```
from beem.rc import RC
comment_dict = {
                "permlink": "test", "author": "holger80",
                "body": "test", "parent_permlink": "",
                "parent_author": "", "title": "test",
                "json_metadata": {"foo": "bar"}
               }

rc = RC()
print(rc.comment_from_dict(comment_dict))
```

create_claimed_account_dict(*create_claimed_account_dict*)
Calc RC costs for claimed account create

custom_json(*tx_size=444*, *follow_id=False*)

custom_json_dict(*custom_json_dict*)
Calc RC costs for a custom_json

Example for calculating RC costs

```
from beem.rc import RC
from collections import OrderedDict
custom_json_dict = {
                    "json": [
                                "reblog", OrderedDict([("account", "xeroc"), (
↪"author", "chainsquad"),
                                                       ("permlink", "streemian-
↪com-to-open-its-doors-and-offer-a-20-discount")
                                                      ])
                            ],
                    "required_auths": [],
                    "required_posting_auths": ["xeroc"],
                    "id": "follow"
                   }

rc = RC()
print(rc.comment(custom_json_dict))
```

get_authority_byte_count(*auth*)

get_resource_count(*tx_size*, *execution_time_count*, *state_bytes_count=0*, *new_account_op_count=0*, *market_op_count=0*)
Creates the resource_count dictionary based on tx_size, state_bytes_count, new_account_op_count and market_op_count

get_tx_size(*op*)
Returns the tx size of an operation

transfer(*tx_size=290*, *market_op_count=1*)
Calc RC of a transfer

transfer_dict(*transfer_dict*)
Calc RC costs for a transfer dict object

Example for calculating RC costs

```
from beem.rc import RC
from beem.amount import Amount
transfer_dict = {
                 "from": "foo", "to": "baar",
                 "amount": Amount("111.110 STEEM"),
                 "memo": "Fooo"
                }

rc = RC()
print(rc.comment(transfer_dict))
```

vote(*tx_size=210*)
Calc RC for a vote

vote_dict(*vote_dict*)
Calc RC costs for a vote

Example for calculating RC costs

```
from beem.rc import RC
vote_dict = {
             "voter": "foobara", "author": "foobarc",
             "permlink": "foobard", "weight": 1000
            }

rc = RC()
print(rc.comment(vote_dict))
```

beem.snapshot

class beem.snapshot.**AccountSnapshot**(*account*, *account_history=[]*, *steem_instance=None*)
Bases: list

This class allows to easily access Account history

Parameters

- **account_name** (*str*) – Name of the account
- **steem_instance** (Steem) – Steem instance

build(*only_ops=[]*, *exclude_ops=[]*, *enable_rewards=False*, *enable_out_votes=False*, *enable_in_votes=False*)
Builds the account history based on all account operations

Parameters

- **only_ops** (*array*) – Limit generator by these operations (*optional*)
- **exclude_ops** (*array*) – Exclude these operations from generator (*optional*)

build_curation_arrays(*end_date=None*, *sum_days=7*)
Build curation arrays

build_rep_arrays()
Build reputation arrays

build_sp_arrays()
Builds the own_sp and eff_sp array

build_vp_arrays()
Build vote power arrays

get_account_history(*start=None*, *stop=None*, *use_block_num=True*)
Uses account history to fetch all related ops

Parameters

- **start** (*int, datetime*) – start number/date of transactions to return (*optional*)
- **stop** (*int, datetime*) – stop number/date of transactions to return (*optional*)
- **use_block_num** (*bool*) – if true, start and stop are block numbers, otherwise virtual OP count numbers.

get_data(*timestamp=None*, *index=0*)
Returns snapshot for given timestamp

get_ops(*start=None*, *stop=None*, *use_block_num=True*, *only_ops=[]*, *exclude_ops=[]*)
Returns ops in the given range

parse_op(*op*, *only_ops=[]*, *enable_rewards=False*, *enable_out_votes=False*, *enable_in_votes=False*)
Parse account history operation

reset()
Resets the arrays not the stored account history

search(*search_str*, *start=None*, *stop=None*, *use_block_num=True*)
Returns ops in the given range

update(*timestamp*, *own*, *delegated_in=None*, *delegated_out=None*, *steem=0*, *sbd=0*)
Updates the internal state arrays

Parameters

- **timestamp** (*datetime*) – datetime of the update
- **own** (amount.Amount, *float*) – vests
- **delegated_in** (*dict*) – Incoming delegation
- **delegated_out** (*dict*) – Outgoing delegation
- **steem** (amount.Amount, *float*) – steem
- **sbd** (amount.Amount, *float*) – sbd

update_in_vote(*timestamp*, *weight*, *op*)

update_out_vote(*timestamp*, *weight*)

update_rewards(*timestamp*, *curation_reward*, *author_vests*, *author_steem*, *author_sbd*)

beem.steem

class beem.steem.**Steem**(*node=''*, *rpcuser=None*, *rpcpassword=None*, *debug=False*, *data_refresh_time_seconds=900*, ***kwargs*)

Bases: object

Connect to the Steem network.

Parameters

- **node** (*str*) – Node to connect to *(optional)*
- **rpcuser** (*str*) – RPC user *(optional)*
- **rpcpassword** (*str*) – RPC password *(optional)*
- **nobroadcast** (*bool*) – Do **not** broadcast a transaction! *(optional)*
- **unsigned** (*bool*) – Do **not** sign a transaction! *(optional)*
- **debug** (*bool*) – Enable Debugging *(optional)*
- **keys** (*array, dict, string*) – Predefine the wif keys to shortcut the wallet database *(optional)*
- **wif** (*array, dict, string*) – Predefine the wif keys to shortcut the wallet database *(optional)*
- **offline** (*bool*) – Boolean to prevent connecting to network (defaults to False) *(optional)*
- **expiration** (*int*) – Delay in seconds until transactions are supposed to expire *(optional)* (default is 30)
- **blocking** (*str*) – Wait for broadcasted transactions to be included in a block and return full transaction (can be "head" or "irreversible")
- **bundle** (*bool*) – Do not broadcast transactions right away, but allow to bundle operations. It is not possible to send out more than one vote operation and more than one comment operation in a single broadcast *(optional)*
- **appbase** (*bool*) – Use the new appbase rpc protocol on nodes with version 0.19.4 or higher. The settings has no effect on nodes with version of 0.19.3 or lower.
- **num_retries** (*int*) – Set the maximum number of reconnects to the nodes before NumRetriesReached is raised. Disabled for -1. (default is -1)
- **num_retries_call** (*int*) – Repeat num_retries_call times a rpc call on node error (default is 5)
- **timeout** (*int*) – Timeout setting for https nodes (default is 60)
- **use_sc2** (*bool*) – When True, a steemconnect object is created. Can be used for broadcast posting op or creating hot_links (default is False)
- **steemconnect** (SteemConnect) – A SteemConnect object can be set manually, set use_sc2 to True
- **custom_chains** (*dict*) – custom chain which should be added to the known chains

Three wallet operation modes are possible:

- **Wallet Database**: Here, the steemlibs load the keys from the locally stored wallet SQLite database (see `storage.py`). To use this mode, simply call `Steem()` without the `keys` parameter
- **Providing Keys**: Here, you can provide the keys for your accounts manually. All you need to do is add the wif keys for the accounts you want to use as a simple array using the `keys` parameter to `Steem()`.
- **Force keys**: This more is for advanced users and requires that you know what you are doing. Here, the `keys` parameter is a dictionary that overwrite the `active`, `owner`, `posting` or `memo` keys for any account. This mode is only used for *foreign* signatures!

If no node is provided, it will connect to default nodes of http://geo.steem.pl. Default settings can be changed with:

```
steem = Steem(<host>)
```

where `<host>` starts with `https://`, `ws://` or `wss://`.

The purpose of this class it to simplify interaction with Steem.

The idea is to have a class that allows to do this:

```
>>> from beem import Steem
>>> steem = Steem()
>>> print(steem.get_blockchain_version())
```

This class also deals with edits, votes and reading content.

Example for adding a custom chain:

```
from beem import Steem
stm = Steem(node=["https://mytstnet.com"], custom_chains={"MYTESTNET":
    {'chain_assets': [{'asset': 'SBD', 'id': 0, 'precision': 3, 'symbol': 'SBD'},
                      {'asset': 'STEEM', 'id': 1, 'precision': 3, 'symbol': 'STEEM
↪'},
                      {'asset': 'VESTS', 'id': 2, 'precision': 6, 'symbol': 'VESTS
↪'}],
     'chain_id': '79276aea5d4877d9a25892eaa01b0adf019d3e5cb12a97478df3298ccdd01674
↪',
     'min_version': '0.0.0',
     'prefix': 'MTN'}
    }
)
```

broadcast (*tx=None*)
Broadcast a transaction to the Steem network

Parameters **tx** (*tx*) – Signed transaction to broadcast

chain_params

claim_account (*creator*, *fee=None*, ***kwargs*)
Claim account for claimed account creation.

When fee is 0 STEEM a subsidized account is claimed and can be created later with create_claimed_account. The number of subsidized account is limited.

Parameters

- **creator** (*str*) – which account should pay the registration fee (RC or STEEM) (defaults to `default_account`)
- **fee** (*str*) – when set to 0 STEEM (default), claim account is paid by RC

clear()

clear_data()
Clears all stored blockchain parameters

comment_options(*options*, *identifier*, *beneficiaries=[]*, *account=None*, ***kwargs*)
Set the comment options

Parameters

- **options** (*dict*) – The options to define.
- **identifier** (*str*) – Post identifier
- **beneficiaries** (*list*) – (optional) list of beneficiaries
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

For the options, you have these defaults::

```
{
    "author": "",
    "permlink": "",
    "max_accepted_payout": "1000000.000 SBD",
    "percent_steem_dollars": 10000,
    "allow_votes": True,
    "allow_curation_rewards": True,
}
```

connect(*node=''*, *rpcuser=''*, *rpcpassword=''*, ***kwargs*)
Connect to Steem network (internal use only)

create_account(*account_name*, *creator=None*, *owner_key=None*, *active_key=None*, *memo_key=None*, *posting_key=None*, *password=None*, *additional_owner_keys=[]*, *additional_active_keys=[]*, *additional_posting_keys=[]*, *additional_owner_accounts=[]*, *additional_active_accounts=[]*, *additional_posting_accounts=[]*, *storekeys=True*, *store_owner_key=False*, *json_meta=None*, ***kwargs*)
Create new account on Steem

The brainkey/password can be used to recover all generated keys (see *beemgraphenebase.account* for more details.

By default, this call will use `default_account` to register a new name `account_name` with all keys being derived from a new brain key that will be returned. The corresponding keys will automatically be installed in the wallet.

Warning: Don't call this method unless you know what you are doing! Be sure to understand what this method does and where to find the private keys for your account.

Note: Please note that this imports private keys (if password is present) into the wallet by default when nobroadcast is set to False. However, it **does not import the owner key** for security reasons by default. If you set store_owner_key to True, the owner key is stored. Do NOT expect to be able to recover it from the wallet if you lose your password!

Note: Account creations cost a fee that is defined by the network. If you create an account, you will need to pay for that fee!

Parameters

- **account_name** (*str*) – (**required**) new account name
- **json_meta** (*str*) – Optional meta data for the account
- **owner_key** (*str*) – Main owner key
- **active_key** (*str*) – Main active key
- **posting_key** (*str*) – Main posting key
- **memo_key** (*str*) – Main memo_key
- **password** (*str*) – Alternatively to providing keys, one can provide a password from which the keys will be derived
- **additional_owner_keys** (*array*) – Additional owner public keys
- **additional_active_keys** (*array*) – Additional active public keys
- **additional_posting_keys** (*array*) – Additional posting public keys
- **additional_owner_accounts** (*array*) – Additional owner account names
- **additional_active_accounts** (*array*) – Additional acctive account names
- **storekeys** (*bool*) – Store new keys in the wallet (default: `True`)
- **creator** (*str*) – which account should pay the registration fee (defaults to `default_account`)

Raises ***AccountExistsException*** – if the account already exists on the blockchain

create_claimed_account(*account_name*, *creator=None*, *owner_key=None*, *active_key=None*, *memo_key=None*, *posting_key=None*, *password=None*, *additional_owner_keys=[]*, *additional_active_keys=[]*, *additional_posting_keys=[]*, *additional_owner_accounts=[]*, *additional_active_accounts=[]*, *additional_posting_accounts=[]*, *storekeys=True*, *store_owner_key=False*, *json_meta=None*, *combine_with_claim_account=False*, *fee=None*, ***kwargs*)

Create new claimed account on Steem

The brainkey/password can be used to recover all generated keys (see *beemgraphenebase.account* for more details.

By default, this call will use `default_account` to register a new name `account_name` with all keys being derived from a new brain key that will be returned. The corresponding keys will automatically be installed in the wallet.

Warning: Don't call this method unless you know what you are doing! Be sure to understand what this method does and where to find the private keys for your account.

Note: Please note that this imports private keys (if password is present) into the wallet by default when nobroadcast is set to False. However, it **does not import the owner key** for security reasons by default. If

you set store_owner_key to True, the owner key is stored. Do NOT expect to be able to recover it from the wallet if you lose your password!

Note: Account creations cost a fee that is defined by the network. If you create an account, you will need to pay for that fee!

Parameters

- **account_name** (*str*) – (**required**) new account name
- **json_meta** (*str*) – Optional meta data for the account
- **owner_key** (*str*) – Main owner key
- **active_key** (*str*) – Main active key
- **posting_key** (*str*) – Main posting key
- **memo_key** (*str*) – Main memo_key
- **password** (*str*) – Alternatively to providing keys, one can provide a password from which the keys will be derived
- **additional_owner_keys** (*array*) – Additional owner public keys
- **additional_active_keys** (*array*) – Additional active public keys
- **additional_posting_keys** (*array*) – Additional posting public keys
- **additional_owner_accounts** (*array*) – Additional owner account names
- **additional_active_accounts** (*array*) – Additional acctive account names
- **storekeys** (*bool*) – Store new keys in the wallet (default: `True`)
- **combine_with_claim_account** (*bool*) – When set to True, a claim_account operation is additionally broadcasted
- **fee** (*str*) – When combine_with_claim_account is set to True, this parameter is used for the claim_account operation
- **creator** (*str*) – which account should pay the registration fee (defaults to `default_account`)

Raises ***AccountExistsException*** – if the account already exists on the blockchain

custom_json(*id*, *json_data*, *required_auths=[]*, *required_posting_auths=[]*, ***kwargs*)

Create a custom json operation

Parameters

- **id** (*str*) – identifier for the custom json (max length 32 bytes)
- **json_data** (*json*) – the json data to put into the custom_json operation
- **required_auths** (*list*) – (optional) required auths
- **required_posting_auths** (*list*) – (optional) posting auths

---

Note: While reqired auths and required_posting_auths are both optional, one of the two are needed in order to send the custom json.

```
steem.custom_json("id", "json_data",
required_posting_auths=['account'])
```

finalizeOp(*ops*, *account*, *permission*, ***kwargs*)
This method obtains the required private keys if present in the wallet, finalizes the transaction, signs it and broadacasts it

Parameters

- **ops** (*list, GrapheneObject*) – The operation (or list of operations) to broadcast
- **account** (*Account*) – The account that authorizes the operation
- **permission** (*string*) – The required permission for signing (active, owner, posting)
- **append_to** (*TransactionBuilder*) – This allows to provide an instance of TransactionBuilder (see *Steem.new_tx()*) to specify where to put a specific operation.

Note: append_to is exposed to every method used in the Steem class

Note: If ops is a list of operation, they all need to be signable by the same key! Thus, you cannot combine ops that require active permission with ops that require posting permission. Neither can you use different accounts for different operations!

Note: This uses *Steem.txbuffer()* as instance of *beem.transactionbuilder.TransactionBuilder*. You may want to use your own txbuffer

get_api_methods()
Returns all supported api methods

get_apis()
Returns all enabled apis

get_block_interval(*use_stored_data=True*)
Returns the block interval in seconds

get_blockchain_name(*use_stored_data=True*)
Returns the blockchain version

get_blockchain_version(*use_stored_data=True*)
Returns the blockchain version

get_chain_properties(*use_stored_data=True*)
Return witness elected chain properties

Properties::

```
{
    'account_creation_fee': '30.000 STEEM',
    'maximum_block_size': 65536,
    'sbd_interest_rate': 250
}
```

get_config(*use_stored_data=True*)
Returns internal chain configuration.

Parameters **use_stored_data** (*bool*) – If True, the cached value is returned

get_current_median_history(*use_stored_data=True*)
Returns the current median price

Parameters **use_stored_data** (*bool*) – if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

get_default_nodes()
Returns the default nodes

get_dust_threshold(*use_stored_data=True*)
Returns the vote dust threshold

get_dynamic_global_properties(*use_stored_data=True*)
This call returns the *dynamic global properties*

Parameters **use_stored_data** (*bool*) – if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

get_feed_history(*use_stored_data=True*)
Returns the feed_history

Parameters **use_stored_data** (*bool*) – if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

get_hardfork_properties(*use_stored_data=True*)
Returns Hardfork and live_time of the hardfork

Parameters **use_stored_data** (*bool*) – if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

get_median_price(*use_stored_data=True*)
Returns the current median history price as Price

get_network(*use_stored_data=True*, *config=None*)
Identify the network

Parameters **use_stored_data** (*bool*) – if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

Returns Network parameters

Return type dictionary

get_rc_cost(*resource_count*)
Returns the RC costs based on the resource_count

get_reserve_ratio()
This call returns the *reserve ratio*

get_resource_params()
Returns the resource parameter

get_resource_pool()
Returns the resource pool

get_reward_funds(*use_stored_data=True*)
Get details for a reward fund.

Parameters **use_stored_data** (*bool*) – if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

get_sbd_per_rshares(*not_broadcasted_vote_rshares=0*, *use_stored_data=True*)
Returns the current rshares to SBD ratio

get_steem_per_mvest(*time_stamp=None*, *use_stored_data=True*)
Returns the MVEST to STEEM ratio

Parameters **time_stamp** (*int*) – (optional) if set, return an estimated STEEM per MVEST ratio for the given time stamp. If unset the current ratio is returned (default). (can also be a datetime object)

get_witness_schedule(*use_stored_data=True*)
Return witness elected chain properties

hardfork

info(*use_stored_data=True*)
Returns the global properties

is_connected()
Returns if rpc is connected

is_hive

move_current_node_to_front()
Returns the default node list, until the first entry is equal to the current working node url

newWallet(*pwd*)
Create a new wallet. This method is basically only calls *beem.wallet.Wallet.create()*.

Parameters **pwd** (*str*) – Password to use for the new wallet

Raises ***WalletExists*** – if there is already a wallet created

new_tx(**args*, ***kwargs*)
Let's obtain a new txbuffer

Returns id of the new txbuffer

Return type int

post(*title*, *body*, *author=None*, *permlink=None*, *reply_identifier=None*, *json_metadata=None*, *comment_options=None*, *community=None*, *app=None*, *tags=None*, *beneficiaries=None*, *self_vote=False*, *parse_body=False*, ***kwargs*)
Create a new post. If this post is intended as a reply/comment, *reply_identifier* needs to be set with the identifier of the parent post/comment (eg. *@author/permlink*). Optionally you can also set json_metadata, comment_options and upvote the newly created post as an author. Setting category, tags or community will override the values provided in json_metadata and/or comment_options where appropriate.

Parameters

- **title** (*str*) – Title of the post
- **body** (*str*) – Body of the post/comment
- **author** (*str*) – Account are you posting from
- **permlink** (*str*) – Manually set the permlink (defaults to None). If left empty, it will be derived from title automatically.
- **reply_identifier** (*str*) – Identifier of the parent post/comment (only if this post is a reply/comment).
- **json_metadata** (*str, dict*) – JSON meta object that can be attached to the post.
- **comment_options** (*dict*) – JSON options object that can be attached to the post.

Example:

```
comment_options = {
    'max_accepted_payout': '1000000.000 SBD',
    'percent_steem_dollars': 10000,
    'allow_votes': True,
    'allow_curation_rewards': True,
    'extensions': [[0, {
        'beneficiaries': [
            {'account': 'account1', 'weight': 5000},
            {'account': 'account2', 'weight': 5000},
        ]}
    ]]
}
```

Parameters

- **community** (*str*) – (Optional) Name of the community we are posting into. This will also override the community specified in *json_metadata*.
- **app** (*str*) – (Optional) Name of the app which are used for posting when not set, beem/<version> is used
- **tags** (*str, list*) – (Optional) A list of tags to go with the post. This will also override the tags specified in *json_metadata*. The first tag will be used as a 'category'. If provided as a string, it should be space separated.
- **beneficiaries** (*list*) – (Optional) A list of beneficiaries for posting reward distribution. This argument overrides beneficiaries as specified in *comment_options*.

For example, if we would like to split rewards between account1 and account2:

```
beneficiaries = [
    {'account': 'account1', 'weight': 5000},
    {'account': 'account2', 'weight': 5000}
]
```

Parameters

- **self_vote** (*bool*) – (Optional) Upvote the post as author, right after posting.
- **parse_body** (*bool*) – (Optional) When set to True, all mentioned users, used links and images are put into users, links and images array inside json_metadata. This will override provided links, images and users inside json_metadata. Hashtags will added to tags until its length is below five entries.

prefix

refresh_data(*property*, *force_refresh=False*, *data_refresh_time_seconds=None*)
Read and stores steem blockchain parameters If the last data refresh is older than data_refresh_time_seconds, data will be refreshed

Parameters

- **force_refresh** (*bool*) – if True, a refresh of the data is enforced
- **data_refresh_time_seconds** (*float*) – set a new minimal refresh time in seconds

rshares_to_sbd(*rshares*, *not_broadcasted_vote=False*, *use_stored_data=True*)
Calculates the current SBD value of a vote

rshares_to_vote_pct(*rshares*, *steem_power=None*, *vests=None*, *voting_power=10000*, *use_stored_data=True*)
Obtain the voting percentage for a desired rshares value for a given Steem Power or vesting shares and voting_power Give either steem_power or vests, not both. When the output is greater than 10000 or less than -10000, the given absolute rshares are too high

Returns the required voting percentage (100% = 10000)

Parameters

- **rshares** (*number*) – desired rshares value
- **steem_power** (*number*) – Steem Power
- **vests** (*number*) – vesting shares
- **voting_power** (*int*) – voting power (100% = 10000)

sbd_symbol
get the current chains symbol for SBD (e.g. "TBD" on testnet)

sbd_to_rshares(*sbd*, *not_broadcasted_vote=False*, *use_stored_data=True*)
Obtain the r-shares from SBD

Parameters

- **sbd** (*str, int,* amount.Amount) – SBD
- **not_broadcasted_vote** (*bool*) – not_broadcasted or already broadcasted vote (True = not_broadcasted vote). Only impactful for very high amounts of SBD. Slight modification to the value calculation, as the not_broadcasted vote rshares decreases the reward pool.

sbd_to_vote_pct(*sbd*, *steem_power=None*, *vests=None*, *voting_power=10000*, *not_broadcasted_vote=True*, *use_stored_data=True*)
Obtain the voting percentage for a desired SBD value for a given Steem Power or vesting shares and voting power Give either Steem Power or vests, not both. When the output is greater than 10000 or smaller than -10000, the SBD value is too high.

Returns the required voting percentage (100% = 10000)

Parameters

- **sbd** (*str, int,* amount.Amount) – desired SBD value
- **steem_power** (*number*) – Steem Power
- **vests** (*number*) – vesting shares
- **not_broadcasted_vote** (*bool*) – not_broadcasted or already broadcasted vote (True = not_broadcasted vote). Only impactful for very high amounts of SBD. Slight modification to the value calculation, as the not_broadcasted vote rshares decreases the reward pool.

set_default_account(*account*)
Set the default account to be used

set_default_nodes(*nodes*)
Set the default nodes to be used

set_default_vote_weight(*vote_weight*)
Set the default vote weight to be used

set_password_storage(*password_storage*)
Set the password storage mode.

When set to “no”, the password has to be provided each time. When set to “environment” the password is taken from the UNLOCK variable

When set to “keyring” the password is taken from the python keyring module. A wallet password can be stored with python -m keyring set beem wallet password

Parameters **password_storage** (*str*) – can be “no”, “keyring” or “environment”

sign(*tx=None*, *wifs=[]*, *reconstruct_tx=True*)
Sign a provided transaction with the provided key(s)

Parameters

- **tx** (*dict*) – The transaction to be signed and returned
- **wifs** (*string*) – One or many wif keys to use for signing a transaction. If not present, the keys will be loaded from the wallet as defined in “missing_signatures” key of the transactions.
- **reconstruct_tx** (*bool*) – when set to False and tx is already contructed, it will not reconstructed and already added signatures remain

sp_to_rshares(*steem_power*, *voting_power=10000*, *vote_pct=10000*, *use_stored_data=True*)
Obtain the r-shares from Steem power

Parameters

- **steem_power** (*number*) – Steem Power
- **voting_power** (*int*) – voting power (100% = 10000)
- **vote_pct** (*int*) – voting percentage (100% = 10000)

sp_to_sbd(*sp*, *voting_power=10000*, *vote_pct=10000*, *not_broadcasted_vote=True*, *use_stored_data=True*)
Obtain the resulting SBD vote value from Steem power

Parameters

- **steem_power** (*number*) – Steem Power
- **voting_power** (*int*) – voting power (100% = 10000)
- **vote_pct** (*int*) – voting percentage (100% = 10000)
- **not_broadcasted_vote** (*bool*) – not_broadcasted or already broadcasted vote (True = not_broadcasted vote).

Only impactful for very big votes. Slight modification to the value calculation, as the not_broadcasted vote rshares decreases the reward pool.

sp_to_vests(*sp*, *timestamp=None*, *use_stored_data=True*)
Converts SP to vests

Parameters

- **sp** (*float*) – Steem power to convert
- **timestamp** (*datetime*) – (Optional) Can be used to calculate the conversion rate from the past

steem_symbol
get the current chains symbol for STEEM (e.g. “TESTS” on testnet)

switch_blockchain(*blockchain*, *update_nodes=False*)
Switches the connected blockchain. Can be either hive or steem.

Parameters

- **blockchain** (*str*) – can be "hive" or "steem"
- **update_nodes** (*bool*) – When true, the nodes are updated, using NodeList.update_nodes()

tx()
Returns the default transaction buffer

txbuffer
Returns the currently active tx buffer

unlock(**args*, ***kwargs*)
Unlock the internal wallet

update_proposal_votes(*proposal_ids*, *approve*, *account=None*, ***kwargs*)
Update proposal votes

Parameters

- **proposal_ids** (*list*) – list of proposal ids
- **approve** (*bool*) – True/False
- **account** (*str*) – (optional) witness account name

vests_symbol
get the current chains symbol for VESTS

vests_to_rshares(*vests*, *voting_power=10000*, *vote_pct=10000*, *subtract_dust_threshold=True*, *use_stored_data=True*)
Obtain the r-shares from vests

Parameters

- **vests** (*number*) – vesting shares
- **voting_power** (*int*) – voting power (100% = 10000)
- **vote_pct** (*int*) – voting percentage (100% = 10000)

vests_to_sbd(*vests*, *voting_power=10000*, *vote_pct=10000*, *not_broadcasted_vote=True*, *use_stored_data=True*)
Obtain the resulting SBD vote value from vests

Parameters

- **vests** (*number*) – vesting shares
- **voting_power** (*int*) – voting power (100% = 10000)
- **vote_pct** (*int*) – voting percentage (100% = 10000)
- **not_broadcasted_vote** (*bool*) – not_broadcasted or already broadcasted vote (True = not_broadcasted vote).

Only impactful for very big votes. Slight modification to the value calculation, as the not_broadcasted vote rshares decreases the reward pool.

vests_to_sp(*vests*, *timestamp=None*, *use_stored_data=True*)
Converts vests to SP

Parameters

- **vests/float vests** (amount.Amount) – Vests to convert

- **timestamp** (*int*) – (Optional) Can be used to calculate the conversion rate from the past

vote(*weight*, *identifier*, *account=None*, ***kwargs*)
Vote for a post

Parameters

- **weight** (*float*) – Voting weight. Range: -100.0 - +100.0.
- **identifier** (*str*) – Identifier for the post to vote. Takes the form `@author/permlink`.
- **account** (*str*) – (optional) Account to use for voting. If `account` is not defined, the `default_account` will be used or a ValueError will be raised

witness_set_properties(*wif*, *owner*, *props*, *use_condenser_api=True*)
Set witness properties

Parameters

- **wif** (*str*) – Private signing key
- **props** (*dict*) – Properties
- **owner** (*str*) – witness account name

Properties::

```
{
    "account_creation_fee": x,
    "account_subsidy_budget": x,
    "account_subsidy_decay": x,
    "maximum_block_size": x,
    "url": x,
    "sbd_exchange_rate": x,
    "sbd_interest_rate": x,
    "new_signing_key": x
}
```

witness_update(*signing_key*, *url*, *props*, *account=None*, ***kwargs*)
Creates/updates a witness

Parameters

- **signing_key** (*str*) – Public signing key
- **url** (*str*) – URL
- **props** (*dict*) – Properties
- **account** (*str*) – (optional) witness account name

Properties::

```
{
    "account_creation_fee": "3.000 STEEM",
    "maximum_block_size": 65536,
    "sbd_interest_rate": 0,
}
```

beem.steemconnect

class beem.steemconnect.**SteemConnect**(*steem_instance=None*, **args*, ***kwargs*)

Bases: object

Parameters **scope** (*str*) – comma separated string with scopes login,offline,vote,comment,delete_comment,comment_options,custom_json,claim_reward_balance

```
# Run the login_app in examples and login with a account
from beem import Steem
from beem.steemconnect import SteemConnect
from beem.comment import Comment
sc2 = SteemConnect(client_id="beem.app")
steem = Steem(steemconnect=sc2)
steem.wallet.unlock("supersecret-passphrase")
post = Comment("author/permlink", steem_instance=steem)
post.upvote(voter="test")  # replace "test" with your account
```

Examples for creating steemconnect v2 urls for broadcasting in browser:

```
from beem import Steem
from beem.account import Account
from beem.steemconnect import SteemConnect
from pprint import pprint
steem = Steem(nobroadcast=True, unsigned=True)
sc2 = SteemConnect(steem_instance=steem)
acc = Account("test", steem_instance=steem)
pprint(sc2.url_from_tx(acc.transfer("test1", 1, "STEEM", "test")))
```

```
'https://steemconnect.com/sign/transfer?from=test&to=test1&amount=1.000+STEEM&
↪memo=test'
```

```
from beem import Steem
from beem.transactionbuilder import TransactionBuilder
from beembase import operations
from beem.steemconnect import SteemConnect
from pprint import pprint
stm = Steem(nobroadcast=True, unsigned=True)
sc2 = SteemConnect(steem_instance=stm)
tx = TransactionBuilder(steem_instance=stm)
op = operations.Transfer(**{"from": 'test',
                            "to": 'test1',
                            "amount": '1.000 STEEM',
                            "memo": 'test'})
tx.appendOps(op)
pprint(sc2.url_from_tx(tx.json()))
```

```
'https://steemconnect.com/sign/transfer?from=test&to=test1&amount=1.000+STEEM&
↪memo=test'
```

broadcast(*operations*, *username=None*)

Broadcast an operation

Sample operations:

```
[
    [
```

(continues on next page)

(continued from previous page)

```
        'vote', {
                    'voter': 'gandalf',
                    'author': 'gtg',
                    'permlink': 'steem-pressure-4-need-for-speed',
                    'weight': 10000
                }
    ]
]
```

create_hot_sign_url(*operation*, *params*, *redirect_uri=None*)
Creates a link for broadcasting an operation

Parameters

- **operation** (*str*) – operation name (e.g.: vote)
- **params** (*dict*) – operation dict params
- **redirect_uri** (*str*) – Redirects to this uri, when set

get_access_token(*code*)

get_login_url(*redirect_uri*, ***kwargs*)
Returns a login url for receiving token from steemconnect

headers

me(*username=None*)
Calls the me function from steemconnect

```
from beem.steemconnect import SteemConnect
sc2 = SteemConnect()
sc2.steem.wallet.unlock("supersecret-passphrase")
sc2.me(username="test")
```

refresh_access_token(*code*, *scope*)

revoke_token(*access_token*)

set_access_token(*access_token*)
Is needed for *broadcast()* and *me()*

set_username(*username*, *permission='posting'*)
Set a username for the next *broadcast()* or *me()* operation. The necessary token is fetched from the wallet

update_user_metadata(*metadata*)

url_from_tx(*tx*, *redirect_uri=None*)
Creates a link for broadcasting an operation

Parameters

- **tx** (*dict*) – includes the operation, which should be broadcast
- **redirect_uri** (*str*) – Redirects to this uri, when set

beem.storage

class beem.storage.**Configuration**
Bases: *beem.storage.DataDir*

This is the configuration storage that stores key/value pairs in the *config* table of the SQLite3 database.

blockchain = 'steem'

checkBackup()
Backup the SQL database every 7 days

config_defaults = {'client_id': '', 'default_chain': 'steem', 'hot_sign_redirect_ur

create_table()
Create the new table in the SQLite database

delete(*key*)
Delete a key from the configuration store

exists_table()
Check if the database table exists

get(*key*, *default=None*)
Return the key if exists or a default value

items()

nodelist = [{'url': 'https://api.steemit.com', 'version': '0.20.2', 'type': 'appbas
Default configuration

nodes = ['https://steemd.minnowsupportproject.org', 'https://api.steemit.com', 'https:

class beem.storage.**DataDir**
Bases: object

This class ensures that the user's data is stored in its OS preotected user directory:

OSX:

- *~/Library/Application Support/<AppName>*

Windows:

- *C:Documents and Settings<User>Application DataLocal Settings<AppAuthor><AppName>*
- *C:Documents and Settings<User>Application Data<AppAuthor><AppName>*

Linux:

- *~/.local/share/<AppName>*

Furthermore, it offers an interface to generated backups in the *backups/* directory every now and then.

appauthor = 'beem'

appname = 'beem'

clean_data(*backupdir='backups'*)
Delete files older than 70 days

data_dir = '/home/docs/.local/share/beem'

mkdir_p()
Ensure that the directory in which the data is stored exists

recover_with_latest_backup(*backupdir='backups'*)
Replace database with latest backup

refreshBackup()
Make a new backup

sqlDataBaseFile = '/home/docs/.local/share/beem/beem.sqlite'

sqlite3_backup(*backupdir*)
Create timestamped database copy

sqlite3_copy(*src*, *dst*)
Copy sql file from src to dst

storageDatabase = 'beem.sqlite'

class beem.storage.**Key**
Bases: *beem.storage.DataDir*

This is the key storage that stores the public key and the (possibly encrypted) private key in the *keys* table in the SQLite3 database.

add(*wif*, *pub*)
Add a new public/private key pair (correspondence has to be checked elsewhere!)

Parameters

- **pub** (*str*) – Public key
- **wif** (*str*) – Private key

create_table()
Create the new table in the SQLite database

delete(*pub*)
Delete the key identified as *pub*

Parameters **pub** (*str*) – Public key

exists_table()
Check if the database table exists

getPrivateKeyForPublicKey(*pub*)
Returns the (possibly encrypted) private key that corresponds to a public key

Parameters **pub** (*str*) – Public key

The encryption scheme is BIP38

getPublicKeys(*prefix='STM'*)
Returns the public keys stored in the database

updateWif(*pub*, *wif*)
Change the wif to a pubkey

Parameters

- **pub** (*str*) – Public key
- **wif** (*str*) – Private key

wipe(*sure=False*)
Purge the entire wallet. No keys will survive this!

class beem.storage.**MasterPassword**(*password*)
Bases: object

The keys are encrypted with a Masterpassword that is stored in the configurationStore. It has a checksum to verify correctness of the password

changePassword(*newpassword*)
Change the password

config_key = 'encrypted_master_password'
This key identifies the encrypted master password stored in the confiration

decryptEncryptedMaster()
Decrypt the encrypted masterpassword

decrypted_master = ''

deriveChecksum(*s*)
Derive the checksum

getEncryptedMaster()
Obtain the encrypted masterkey

newMaster()
Generate a new random masterpassword

password = ''

saveEncrytpedMaster()
Store the encrypted master password in the configuration store

static wipe(*sure=False*)
Remove all keys from configStorage

class beem.storage.**Token**
Bases: *beem.storage.DataDir*

This is the token storage that stores the public username and the (possibly encrypted) token in the *token* table in the SQLite3 database.

add(*name*, *token*)
Add a new public/private token pair (correspondence has to be checked elsewhere!)

Parameters

- **name** (*str*) – Public name
- **token** (*str*) – Private token

create_table()
Create the new table in the SQLite database

delete(*name*)
Delete the key identified as *name*

Parameters **name** (*str*) – Public name

exists_table()
Check if the database table exists

getPublicNames()
Returns the public names stored in the database

getTokenForPublicName(*name*)
Returns the (possibly encrypted) private token that corresponds to a public name

Parameters **pub** (*str*) – Public name

The encryption scheme is BIP38

updateToken(*name*, *token*)
Change the token to a name

Parameters

- **name** (*str*) – Public name
- **token** (*str*) – Private token

wipe(*sure=False*)
Purge the entire wallet. No keys will survive this!

beem.storage.**get_default_config_storage**()

beem.storage.**get_default_key_storage**()

beem.storage.**get_default_token_storage**()

beem.transactionbuilder

class beem.transactionbuilder.**TransactionBuilder**(*tx={}*, *use_condenser_api=True*, *steem_instance=None*, ***kwargs*)

Bases: dict

This class simplifies the creation of transactions by adding operations and signers. To build your own transactions and sign them

Parameters

- **tx** (*dict*) – transaction (Optional). If not set, the new transaction is created.
- **expiration** (*int*) – Delay in seconds until transactions are supposed to expire *(optional)* (default is 30)
- **steem_instance** (Steem) – If not set, shared_steem_instance() is used

```
from beem.transactionbuilder import TransactionBuilder
from beembase.operations import Transfer
from beem import Steem
wif = "5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3"
stm = Steem(nobroadcast=True, keys={'active': wif})
tx = TransactionBuilder(steem_instance=stm)
transfer = {"from": "test", "to": "test1", "amount": "1 STEEM", "memo": ""}
tx.appendOps(Transfer(transfer))
tx.appendSigner("test", "active") # or tx.appendWif(wif)
signed_tx = tx.sign()
broadcast_tx = tx.broadcast()
```

addSigningInformation(*account*, *permission*, *reconstruct_tx=False*)
This is a private method that adds side information to a unsigned/partial transaction in order to simplify later signing (e.g. for multisig or coldstorage)

Not needed when "appendWif" was already or is going to be used

FIXME: Does not work with owner keys!

Parameters **reconstruct_tx** (*bool*) – when set to False and tx is already contructed, it will not reconstructed and already added signatures remain

appendMissingSignatures()
Store which accounts/keys are supposed to sign the transaction

This method is used for an offline-signer!

appendOps(*ops*, *append_to=None*)
Append op(s) to the transaction builder

Parameters **ops** (*list*) – One or a list of operations

appendSigner(*account*, *permission*)
Try to obtain the wif key from the wallet by telling which account and permission is supposed to sign the transaction It is possible to add more than one signer.

appendWif(*wif*)
Add a wif that should be used for signing of the transaction.

Parameters **wif** (*string*) – One wif key to use for signing a transaction.

broadcast(*max_block_age=-1*)
Broadcast a transaction to the steem network Returns the signed transaction and clears itself after broadast

Clears itself when broadcast was not successfully.

Parameters **max_block_age** (*int*) – parameter only used for appbase ready nodes

clear()
Clear the transaction builder and start from scratch

clearWifs()
Clear all stored wifs

constructTx(*ref_block_num=None*, *ref_block_prefix=None*)
Construct the actual transaction and store it in the class's dict store

get_parent()
TransactionBuilders don't have parents, they are their own parent

get_potential_signatures()
Returns public key from signature

get_required_signatures(*available_keys=[]*)
Returns public key from signature

get_transaction_hex()
Returns a hex value of the transaction

is_empty()
Check if ops is empty

json(*with_prefix=False*)
Show the transaction as plain json

list_operations()
List all ops

set_expiration(*p*)
Set expiration date

sign(*reconstruct_tx=True*)
Sign a provided transaction with the provided key(s) One or many wif keys to use for signing a transaction. The wif keys can be provided by "appendWif" or the signer can be defined "appendSigner". The wif keys from all signer that are defined by "appendSigner will be loaded from the wallet.

Parameters **reconstruct_tx** (*bool*) – when set to False and tx is already contructed, it will not reconstructed and already added signatures remain

verify_authority()
Verify the authority of the signed transaction

beem.utils

beem.utils.**addTzInfo**(*t*, *timezone='UTC'*)
Returns a datetime object with tzinfo added

beem.utils.**assets_from_string**(*text*)
Correctly split a string containing an asset pair.

Splits the string into two assets with the separator being on of the following: `:`, `/`, or `-`.

beem.utils.**construct_authorperm**(**args*)
Create a post identifier from comment/post object or arguments. Examples:

```
>>> from beem.utils import construct_authorperm
>>> print(construct_authorperm('username', 'permlink'))
@username/permlink
>>> print(construct_authorperm({'author': 'username', 'permlink':
↪'permlink'}))
@username/permlink
```

beem.utils.**construct_authorpermvoter**(**args*)
Create a vote identifier from vote object or arguments. Examples:

```
>>> from beem.utils import construct_authorpermvoter
>>> print(construct_authorpermvoter('username', 'permlink', 'voter'))
@username/permlink|voter
>>> print(construct_authorpermvoter({'author': 'username', 'permlink':
↪'permlink', 'voter': 'voter'}))
@username/permlink|voter
```

beem.utils.**derive_beneficiaries**(*beneficiaries*)

beem.utils.**derive_permlink**(*title*, *parent_permlink=None*, *parent_author=None*, *max_permlink_length=256*)
Derive a permlink from a comment title (for root level comments) or the parent permlink and optionally the parent author (for replies).

beem.utils.**derive_tags**(*tags*)

beem.utils.**findall_patch_hunks**(*body=None*)

beem.utils.**formatTime**(*t*)
Properly Format Time for permlinks

beem.utils.**formatTimeFromNow**(*secs=0*)
Properly Format Time that is *x* seconds in the future

Parameters **secs** (*int*) – Seconds to go in the future (*x>0*) or the past (*x<0*)

Returns Properly formated time for Graphene (*%Y-%m-%dT%H:%M:%S*)

Return type str

beem.utils.**formatTimeString**(*t*)
Properly Format Time for permlinks

beem.utils.**formatTimedelta**(*td*)
Format timedelta to String

beem.utils.**formatToTimeStamp**(*t*)
Returns a timestamp integer

Parameters **t** (*datetime*) – datetime object

Returns Timestamp as integer

beem.utils.**load_dirty_json**(*dirty_json*)

beem.utils.**make_patch**(*a*, *b*, *n=3*)

beem.utils.**parse_time**(*block_time*)
Take a string representation of time from the blockchain, and parse it into datetime object.

beem.utils.**remove_from_dict**(*obj*, *keys=[]*, *keep_keys=True*)
Prune a class or dictionary of all but keys (keep_keys=True). Prune a class or dictionary of specified keys.(keep_keys=False).

beem.utils.**reputation_to_score**(*rep*)
Converts the account reputation value into the reputation score

beem.utils.**resolve_authorperm**(*identifier*)
Correctly split a string containing an authorperm.

Splits the string into author and permlink with the following separator: `/`.

Examples:

```
>>> from beem.utils import resolve_authorperm
>>> author, permlink = resolve_authorperm('https://d.tube/#!/v/pottlund/
↪m5cqkd1a')
>>> author, permlink = resolve_authorperm("https://steemit.com/witness-
↪category/@gtg/24lfrm-gtg-witness-log")
>>> author, permlink = resolve_authorperm("@gtg/24lfrm-gtg-witness-log")
>>> author, permlink = resolve_authorperm("https://busy.org/@gtg/24lfrm-
↪gtg-witness-log")
```

beem.utils.**resolve_authorpermvoter**(*identifier*)
Correctly split a string containing an authorpermvoter.

Splits the string into author and permlink with the following separator: `/` and `|`.

beem.utils.**resolve_root_identifier**(*url*)

beem.utils.**sanitize_permlink**(*permlink*)

beem.utils.**seperate_yaml_dict_from_body**(*content*)

beem.vote

class beem.vote.**AccountVotes**(*account*, *start=None*, *stop=None*, *raw_data=False*, *lazy=False*, *full=False*, *steem_instance=None*)
Bases: *beem.vote.VotesObject*

Obtain a list of votes for an account Lists the last 100+ votes on the given account.

Parameters

- **account** (*str*) – Account name
- **steem_instance** (Steem) – Steem() instance to use when accesing a RPC

class beem.vote.**ActiveVotes**(*authorperm*, *lazy=False*, *full=False*, *steem_instance=None*)
Bases: *beem.vote.VotesObject*

Obtain a list of votes for a post

Parameters

- **authorperm** (*str*) – authorperm link
- **steem_instance** (Steem) – Steem() instance to use when accesing a RPC

class beem.vote.**Vote**(*voter*, *authorperm=None*, *full=False*, *lazy=False*, *steem_instance=None*)
Bases: *beem.blockchainobject.BlockchainObject*

Read data about a Vote in the chain

Parameters

- **authorperm** (*str*) – perm link to post/comment
- **steem_instance** (Steem) – Steem() instance to use when accesing a RPC

```
>>> from beem.vote import Vote
>>> from beem import Steem
>>> stm = Steem()
>>> v = Vote("@gtg/steem-pressure-4-need-for-speed|gandalf", steem_instance=stm)
```

authorperm

json()

percent

refresh()

rep

reputation

rshares

sbd

time

type_id = 11

votee

voter

weight

class beem.vote.**VotesObject**
Bases: list

get_list(*var='voter'*, *voter=None*, *votee=None*, *start=None*, *stop=None*, *start_percent=None*, *stop_percent=None*, *sort_key='time'*, *reverse=True*)

get_sorted_list(*sort_key='time'*, *reverse=True*)

printAsTable(*voter=None*, *votee=None*, *start=None*, *stop=None*, *start_percent=None*, *stop_percent=None*, *sort_key='time'*, *reverse=True*, *allow_refresh=True*, *return_str=False*, ***kwargs*)

print_stats(*return_str=False*, ***kwargs*)

beem.wallet

class beem.wallet.**Wallet**(*steem_instance=None*, **args*, ***kwargs*)
Bases: object

The wallet is meant to maintain access to private keys for your accounts. It either uses manually provided private keys or uses a SQLite database managed by storage.py.

Parameters

- **rpc** (SteemNodeRPC) – RPC connection to a Steem node
- **keys** (*array, dict, str*) – Predefine the wif keys to shortcut the wallet database

Three wallet operation modes are possible:

- **Wallet Database**: Here, beem loads the keys from the locally stored wallet SQLite database (see `storage.py`). To use this mode, simply call *beem.steem.Steem* without the `keys` parameter
- **Providing Keys**: Here, you can provide the keys for your accounts manually. All you need to do is add the wif keys for the accounts you want to use as a simple array using the `keys` parameter to *beem.steem.Steem*.
- **Force keys**: This more is for advanced users and requires that you know what you are doing. Here, the `keys` parameter is a dictionary that overwrite the `active`, `owner`, `posting` or `memo` keys for any account. This mode is only used for *foreign* signatures!

A new wallet can be created by using:

```
from beem import Steem
steem = Steem()
steem.wallet.wipe(True)
steem.wallet.create("supersecret-passphrase")
```

This will raise *beem.exceptions.WalletExists* if you already have a wallet installed.

The wallet can be unlocked for signing using

```
from beem import Steem
steem = Steem()
steem.wallet.unlock("supersecret-passphrase")
```

A private key can be added by using the *addPrivateKey()* method that is available **after** unlocking the wallet with the correct passphrase:

```
from beem import Steem
steem = Steem()
steem.wallet.unlock("supersecret-passphrase")
steem.wallet.addPrivateKey("5xxxxxxxxxxxxxxxxxxx")
```

Note: The private key has to be either in hexadecimal or in wallet import format (wif) (starting with a `5`).

MasterPassword = None

addPrivateKey(*wif*)
Add a private key to the wallet database

Parameters **wif** (*str*) – Private key

addToken(*name*, *token*)

changePassphrase(*new_pwd*)
Change the passphrase for the wallet database

clear_local_keys()
Clear all manually provided keys

clear_local_token()
Clear all manually provided token

configStorage = None

create(*pwd*)
Alias for *newWallet()*

Parameters **pwd** (*str*) – Passphrase for the created wallet

created()
Do we have a wallet database already?

decrypt_token(*enctoken*)
decrypt a wif key

decrypt_wif(*encwif*)
decrypt a wif key

deriveChecksum(*s*)
Derive the checksum

encrypt_token(*token*)
Encrypt a token key

encrypt_wif(*wif*)
Encrypt a wif key

getAccount(*pub*)
Get the account data for a public key (first account found for this public key)

Parameters **pub** (*str*) – Public key

getAccountFromPrivateKey(*wif*)
Obtain account name from private key

getAccountFromPublicKey(*pub*)
Obtain the first account name from public key

Parameters **pub** (*str*) – Public key

Note: this returns only the first account with the given key. To get all accounts associated with a given public key, use *getAccountsFromPublicKey()*.

getAccounts()
Return all accounts installed in the wallet database

getAccountsFromPublicKey(*pub*)
Obtain all account names associated with a public key

Parameters **pub** (*str*) – Public key

getActiveKeyForAccount(*name*)
Obtain owner Active Key for an account from the wallet database

getActiveKeysForAccount(*name*)
Obtain list of all owner Active Keys for an account from the wallet database

getAllAccounts(*pub*)
Get the account data for a public key (all accounts found for this public key)

Parameters **pub** (*str*) – Public key

getKeyForAccount(*name*, *key_type*)
Obtain *key_type* Private Key for an account from the wallet database

Parameters

- **name** (*str*) – Account name
- **key_type** (*str*) – key type, has to be one of “owner”, “active”, “posting” or “memo”

getKeyType(*account*, *pub*)
Get key type

Parameters

- **account** (*Account, dict*) – Account data
- **pub** (*str*) – Public key

getKeysForAccount(*name*, *key_type*)
Obtain a List of *key_type* Private Keys for an account from the wallet database

Parameters

- **name** (*str*) – Account name
- **key_type** (*str*) – key type, has to be one of “owner”, “active”, “posting” or “memo”

getMemoKeyForAccount(*name*)
Obtain owner Memo Key for an account from the wallet database

getOwnerKeyForAccount(*name*)
Obtain owner Private Key for an account from the wallet database

getOwnerKeysForAccount(*name*)
Obtain list of all owner Private Keys for an account from the wallet database

getPostingKeyForAccount(*name*)
Obtain owner Posting Key for an account from the wallet database

getPostingKeysForAccount(*name*)
Obtain list of all owner Posting Keys for an account from the wallet database

getPrivateKeyForPublicKey(*pub*)
Obtain the private key for a given public key

Parameters **pub** (*str*) – Public Key

getPublicKeys()
Return all installed public keys

getPublicNames()
Return all installed public token

getTokenForAccountName(*name*)
Obtain the private token for a given public name

Parameters **name** (*str*) – Public name

keyMap = {}

keyStorage = None

keys = {}

lock()
Lock the wallet database

locked()
Is the wallet database locked?

masterpassword = None

newWallet(*pwd*)
Create a new wallet database

Parameters **pwd** (*str*) – Passphrase for the created wallet

prefix

removeAccount(*account*)
Remove all keys associated with a given account

Parameters **account** (*str*) – name of account to be removed

removePrivateKeyFromPublicKey(*pub*)
Remove a key from the wallet database

Parameters **pub** (*str*) – Public key

removeTokenFromPublicName(*name*)
Remove a token from the wallet database

Parameters **name** (*str*) – token to be removed

rpc

setKeys(*loadkeys*)
This method is strictly only for in memory keys that are passed to Wallet/Steem with the `keys` argument

setToken(*loadtoken*)
This method is strictly only for in memory token that are passed to Wallet/Steem with the `token` argument

token = {}

tokenStorage = None

tryUnlockFromEnv()
Try to fetch the unlock password from UNLOCK environment variable and keyring when no password is given.

unlock(*pwd=None*)
Unlock the wallet database

unlocked()
Is the wallet database unlocked?

wipe(*sure=False*)
Purge all data in wallet database

beem.witness

class beem.witness.**GetWitnesses**(*name_list*, *batch_limit=100*, *lazy=False*, *full=True*, *steem_instance=None*)
Bases: *beem.witness.WitnessesObject*

Obtain a list of witnesses

Parameters

- **name_list** (*list*) – list of witneses to fetch
- **batch_limit** (*int*) – (optional) maximum number of witnesses to fetch per call, defaults to 100

- **steem_instance** (Steem) – Steem() instance to use when accessing a RPCcreator = Witness(creator, steem_instance=self)

```
from beem.witness import GetWitnesses
w = GetWitnesses(["gtg", "jesta"])
print(w[0].json())
print(w[1].json())
```

class beem.witness.**ListWitnesses**(*from_account=''*, *limit=100*, *lazy=False*, *full=False*, *steem_instance=None*)

Bases: *beem.witness.WitnessesObject*

List witnesses ranked by name

Parameters

- **from_account** (*str*) – Witness name from which the lists starts (default = "")
- **limit** (*int*) – Limits the number of shown witnesses (default = 100)
- **steem_instance** (Steem) – Steem instance to use when accesing a RPC

```
>>> from beem.witness import ListWitnesses
>>> ListWitnesses(from_account="gtg", limit=100)
<ListWitnesses gtg>
```

class beem.witness.**Witness**(*owner*, *full=False*, *lazy=False*, *steem_instance=None*)

Bases: *beem.blockchainobject.BlockchainObject*

Read data about a witness in the chain

Parameters

- **account_name** (*str*) – Name of the witness
- **steem_instance** (Steem) – Steem instance to use when accesing a RPC

```
>>> from beem.witness import Witness
>>> Witness("gtg")
<Witness gtg>
```

account

feed_publish(*base*, *quote=None*, *account=None*)

Publish a feed price as a witness.

Parameters

- **base** (*float*) – USD Price of STEEM in SBD (implied price)
- **quote** (*float*) – (optional) Quote Price. Should be 1.000 (default), unless we are adjusting the feed to support the peg.
- **account** (*str*) – (optional) the source account for the transfer if not self["owner"]

is_active

json()

refresh()

type_id = 3

update(*signing_key*, *url*, *props*, *account=None*)

Update witness

Parameters

- **signing_key** (*str*) – Signing key
- **url** (*str*) – URL
- **props** (*dict*) – Properties
- **account** (*str*) – (optional) witness account name

Properties::

```
{
    "account_creation_fee": x,
    "maximum_block_size": x,
    "sbd_interest_rate": x,
}
```

class beem.witness.**Witnesses**(*lazy=False*, *full=True*, *steem_instance=None*)

Bases: *beem.witness.WitnessesObject*

Obtain a list of **active** witnesses and the current schedule

Parameters **steem_instance** (Steem) – Steem instance to use when accesing a RPC

```
>>> from beem.witness import Witnesses
>>> Witnesses()
<Witnesses >
```

refresh()

class beem.witness.**WitnessesObject**

Bases: list

get_votes_sum()

printAsTable(*sort_key='votes'*, *reverse=True*, *return_str=False*, ***kwargs*)

class beem.witness.**WitnessesRankedByVote**(*from_account=''*, *limit=100*, *lazy=False*, *full=False*, *steem_instance=None*)

Bases: *beem.witness.WitnessesObject*

Obtain a list of witnesses ranked by Vote

Parameters

- **from_account** (*str*) – Witness name from which the lists starts (default = "")
- **limit** (*int*) – Limits the number of shown witnesses (default = 100)
- **steem_instance** (Steem) – Steem instance to use when accesing a RPC

```
>>> from beem.witness import WitnessesRankedByVote
>>> WitnessesRankedByVote(limit=100)
<WitnessesRankedByVote >
```

class beem.witness.**WitnessesVotedByAccount**(*account*, *lazy=False*, *full=True*, *steem_instance=None*)

Bases: *beem.witness.WitnessesObject*

Obtain a list of witnesses which have been voted by an account

Parameters

- **account** (*str*) – Account name

- **steem_instance** (Steem) – Steem instance to use when accesing a RPC

```
>>> from beem.witness import WitnessesVotedByAccount
>>> WitnessesVotedByAccount("gtg")
<WitnessesVotedByAccount gtg>
```

3.7.2 beemapi Modules

beemapi.exceptions

exception beemapi.exceptions.**ApiNotSupported**
Bases: *beemapi.exceptions.RPCError*

exception beemapi.exceptions.**CallRetriesReached**
Bases: Exception

CallRetriesReached Exception. Only for internal use

exception beemapi.exceptions.**FollowApiNotEnabled**
Bases: *beemapi.exceptions.RPCError*

exception beemapi.exceptions.**InvalidEndpointUrl**
Bases: Exception

exception beemapi.exceptions.**MissingRequiredActiveAuthority**
Bases: *beemapi.exceptions.RPCError*

exception beemapi.exceptions.**NoAccessApi**
Bases: *beemapi.exceptions.RPCError*

exception beemapi.exceptions.**NoApiWithName**
Bases: *beemapi.exceptions.RPCError*

exception beemapi.exceptions.**NoMethodWithName**
Bases: *beemapi.exceptions.RPCError*

exception beemapi.exceptions.**NumRetriesReached**
Bases: Exception

NumRetriesReached Exception.

exception beemapi.exceptions.**RPCConnection**
Bases: Exception

RPCConnection Exception.

exception beemapi.exceptions.**RPCError**
Bases: Exception

RPCError Exception.

exception beemapi.exceptions.**RPCErrorDoRetry**
Bases: Exception

RPCErrorDoRetry Exception.

exception beemapi.exceptions.**TimeoutException**
Bases: Exception

exception beemapi.exceptions.**UnauthorizedError**
Bases: Exception

UnauthorizedError Exception.

exception beemapi.exceptions.**UnhandledRPCError**
Bases: *beemapi.exceptions.RPCError*

exception beemapi.exceptions.**UnkownKey**
Bases: *beemapi.exceptions.RPCError*

exception beemapi.exceptions.**UnnecessarySignatureDetected**
Bases: Exception

exception beemapi.exceptions.**VotedBeforeWaitTimeReached**
Bases: Exception

exception beemapi.exceptions.**WorkingNodeMissing**
Bases: Exception

beemapi.exceptions.**decodeRPCErrorMsg**(*e*)
Helper function to decode the raised Exception and give it a python Exception class

beemapi.graphenerpc

Note: This is a low level class that can be used in combination with GrapheneClient

This class allows to call API methods exposed by the witness node via websockets. It does **not** support notifications and is not run asynchronously.

graphennewsrpc.

class beemapi.graphenerpc.**GrapheneRPC**(*urls*, *user=None*, *password=None*, ***kwargs*)
Bases: object

This class allows to call API methods synchronously, without callbacks.

It logs warnings and errors.

Parameters

- **urls** (*str*) – Either a single Websocket/Http URL, or a list of URLs
- **user** (*str*) – Username for Authentication
- **password** (*str*) – Password for Authentication
- **num_retries** (*int*) – Try x times to num_retries to a node on disconnect, -1 for indefinitely (default is 100)
- **num_retries_call** (*int*) – Repeat num_retries_call times a rpc call on node error (default is 5)
- **timeout** (*int*) – Timeout setting for https nodes (default is 60)
- **autoconnect** (*bool*) – When set to false, connection is performed on the first rpc call (default is True)
- **use_condenser** (*bool*) – Use the old condenser_api rpc protocol on nodes with version 0.19.4 or higher. The settings has no effect on nodes with version of 0.19.3 or lower.
- **custom_chains** (*dict*) – custom chain which should be added to the known chains

Available APIs:

- database

- network_node
- network_broadcast

Usage:

```
from beemapi.graphenerpc import GrapheneRPC
ws = GrapheneRPC("wss://steemd.pevo.science","","")
print(ws.get_account_count())

ws = GrapheneRPC("https://api.steemit.com","","")
print(ws.get_account_count())
```

Note: This class allows to call methods available via websocket. If you want to use the notification subsystem, please use `GrapheneWebsocket` instead.

---

error_cnt

error_cnt_call

get_network(*props=None*)
Identify the connected network. This call returns a dictionary with keys chain_id, core_symbol and prefix

get_request_id()
Get request id.

get_use_appbase()
Returns True if appbase ready and appbase calls are set

is_appbase_ready()
Check if node is appbase ready

next()
Switches to the next node url

num_retries

num_retries_call

request_send(*payload*)

rpcclose()
Close Websocket

rpcconnect(*next_url=True*)
Connect to next url in a loop.

rpcexec(*payload*)
Execute a call by sending the payload.

Parameters **payload** (*json*) – Payload data

Raises

- **ValueError** – if the server does not respond in proper JSON format
- ***RPCError*** – if the server returns an error

rpclogin(*user*, *password*)
Login into Websocket

version_string_to_int(*network_version*)

ws_send(*payload*)

class beemapi.graphenerpc.**SessionInstance**
Bases: object

Singelton for the Session Instance

instance = None

beemapi.graphenerpc.**create_ws_instance**(*use_ssl=True*, *enable_multithread=True*)
Get websocket instance

beemapi.graphenerpc.**set_session_instance**(*instance*)
Set session instance

beemapi.graphenerpc.**shared_session_instance**()
Get session instance

beemapi.node

class beemapi.node.**Node**(*url*)
Bases: object

class beemapi.node.**Nodes**(*urls*, *num_retries*, *num_retries_call*)
Bases: list

Stores Node URLs and error counts

disable_node()
Disable current node

error_cnt

error_cnt_call

export_working_nodes()

increase_error_cnt()
Increase node error count for current node

increase_error_cnt_call()
Increase call error count for current node

next()

node

num_retries_call_reached

reset_error_cnt()
Set node error count for current node to zero

reset_error_cnt_call()
Set call error count for current node to zero

sleep_and_check_retries(*errorMsg=None*, *sleep=True*, *call_retry=False*, *showMsg=True*)
Sleep and check if num_retries is reached

url

working_nodes_count

beemapi.steemnoderpc

class beemapi.steemnoderpc.**SteemNodeRPC**(**args*, ***kwargs*)
Bases: *beemapi.graphenerpc.GrapheneRPC*

This class allows to call API methods exposed by the witness node via websockets / rpc-json.

Parameters

- **urls** (*str*) – Either a single Websocket/Http URL, or a list of URLs
- **user** (*str*) – Username for Authentication
- **password** (*str*) – Password for Authentication
- **num_retries** (*int*) – Try x times to num_retries to a node on disconnect, -1 for indefinitely
- **num_retries_call** (*int*) – Repeat num_retries_call times a rpc call on node error (default is 5)
- **timeout** (*int*) – Timeout setting for https nodes (default is 60)
- **use_condenser** (*bool*) – Use the old condenser_api rpc protocol on nodes with version 0.19.4 or higher. The settings has no effect on nodes with version of 0.19.3 or lower.

get_account(*name*, ***kwargs*)
Get full account details from account name

Parameters **name** (*str*) – Account name

rpcexec(*payload*)
Execute a call by sending the payload. It makes use of the GrapheneRPC library. In here, we mostly deal with Steem specific error handling

Parameters **payload** (*json*) – Payload data

Raises

- **ValueError** – if the server does not respond in proper JSON format
- ***RPCError*** – if the server returns an error

set_next_node_on_empty_reply(*next_node_on_empty_reply=True*)
Switch to next node on empty reply for the next rpc call

beemapi.websocket

This class allows subscribe to push notifications from the Steem node.

```
from pprint import pprint
from beemapi.websocket import SteemWebsocket

ws = SteemWebsocket(
    "wss://gtg.steem.house:8090",
    accounts=["test"],
    on_block=print,
)

ws.run_forever()
```

class beemapi.websocket.**SteemWebsocket**(*urls*, *user=''*, *password=''*, *only_block_id=False*, *on_block=None*, *keep_alive=25*, *num_retries=-1*, *timeout=60*, **args*, ***kwargs*)

Create a websocket connection and request push notifications

Parameters

- **urls** (*str*) – Either a single Websocket URL, or a list of URLs
- **user** (*str*) – Username for Authentication
- **password** (*str*) – Password for Authentication
- **keep_alive** (*int*) – seconds between a ping to the backend (defaults to 25seconds)

After instanciating this class, you can add event slots for:

- on_block

which will be called accordingly with the notification message received from the Steem node:

```
ws = SteemWebsocket(
    "wss://gtg.steem.house:8090",
)
ws.on_block += print
ws.run_forever()
```

_SteemWebsocket__set_subscriptions()
set subscriptions ot on_block function

__events__ = ['on_block']

__getattr__(*name*)
Map all methods to RPC calls and pass through the arguments

__init__(*urls*, *user=''*, *password=''*, *only_block_id=False*, *on_block=None*, *keep_alive=25*, *num_retries=-1*, *timeout=60*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'beemapi.websocket'

_ping()
Send keep_alive request

cancel_subscriptions()
cancel_all_subscriptions removed from api

close()
Closes the websocket connection and waits for the ping thread to close

get_request_id()
Generates next request id

on_close(*ws*)
Called when websocket connection is closed

on_error(*ws*, *error*)
Called on websocket errors

on_message(*ws*, *reply*, **args*)
This method is called by the websocket connection on every message that is received. If we receive a notice, we hand over post-processing and signalling of events to process_notice.

on_open(*ws*)
This method will be called once the websocket connection is established. It will

- login,
- register to the database api, and
- subscribe to the objects defined if there is a callback/slot available for callbacks

process_block(*data*)
This method is called on notices that need processing. Here, we call the `on_block` slot.

reset_subscriptions(*accounts=[]*)
Reset subscriptions

rpcexec(*payload*)
Execute a call by sending the payload.

Parameters **payload** (*json*) – Payload data

Raises

- **ValueError** – if the server does not respond in proper JSON format
- ***RPCError*** – if the server returns an error

run_forever()
This method is used to run the websocket app continuously. It will execute callbacks as defined and try to stay connected with the provided APIs

stop()
Stop running Websocket

3.7.3 beembase Modules

beembase.memo

beembase.memo.**decode_memo**(*priv*, *message*)
Decode a message with a shared secret between Alice and Bob

Parameters

- **priv** (PrivateKey) – Private Key (of Bob)
- **message** (*base58encoded*) – Encrypted Memo message

Returns Decrypted message

Return type str

Raises **ValueError** – if message cannot be decoded as valid UTF-8 string

beembase.memo.**decode_memo_bts**(*priv*, *pub*, *nonce*, *message*)
Decode a message with a shared secret between Alice and Bob

Parameters

- **priv** (PrivateKey) – Private Key (of Bob)
- **pub** (PublicKey) – Public Key (of Alice)
- **nonce** (*int*) – Nonce used for Encryption
- **message** (*bytes*) – Encrypted Memo message

Returns Decrypted message

Return type str

Raises **ValueError** – if message cannot be decoded as valid UTF-8 string

beembase.memo.**encode_memo**(*priv*, *pub*, *nonce*, *message*, ***kwargs*)
Encode a message with a shared secret between Alice and Bob

Parameters

- **priv** (PrivateKey) – Private Key (of Alice)
- **pub** (PublicKey) – Public Key (of Bob)
- **nonce** (*int*) – Random nonce
- **message** (*str*) – Memo message

Returns Encrypted message

Return type hex

beembase.memo.**encode_memo_bts**(*priv*, *pub*, *nonce*, *message*)
Encode a message with a shared secret between Alice and Bob

Parameters

- **priv** (PrivateKey) – Private Key (of Alice)
- **pub** (PublicKey) – Public Key (of Bob)
- **nonce** (*int*) – Random nonce
- **message** (*str*) – Memo message

Returns Encrypted message

Return type hex

beembase.memo.**get_shared_secret**(*priv*, *pub*)
Derive the share secret between `priv` and `pub`

Parameters

- **priv** (Base58) – Private Key
- **pub** (Base58) – Public Key

Returns Shared secret

Return type hex

The shared secret is generated such that:

```
Pub(Alice) * Priv(Bob) = Pub(Bob) * Priv(Alice)
```

beembase.memo.**init_aes**(*shared_secret*, *nonce*)
Initialize AES instance

Parameters

- **shared_secret** (*hex*) – Shared Secret to use as encryption key
- **nonce** (*int*) – Random nonce

Returns AES instance and checksum of the encryption key

Return type length 2 tuple

beembase.memo.**init_aes_bts**(*shared_secret*, *nonce*)
Initialize AES instance

Parameters

- **shared_secret** (*hex*) – Shared Secret to use as encryption key
- **nonce** (*int*) – Random nonce

Returns AES instance

Return type AES

beembase.objects

class beembase.objects.**Amount**(*d*, *prefix='STM'*)
Bases: object

class beembase.objects.**Beneficiaries**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

class beembase.objects.**Beneficiary**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

class beembase.objects.**CommentOptionExtensions**(*o*)
Bases: beemgraphenebase.types.Static_variant

Serialize Comment Payout Beneficiaries.

Parameters **beneficiaries** (*list*) – A static_variant containing beneficiaries.

Example:

```
[0,
    {'beneficiaries': [
        {'account': 'furion', 'weight': 10000}
    ]}
]
```

class beembase.objects.**ExchangeRate**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

class beembase.objects.**Extension**(*d*)
Bases: beemgraphenebase.types.Array

class beembase.objects.**Memo**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

class beembase.objects.**Operation**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.Operation*

getOperationNameForId(*i*)
Convert an operation id into the corresponding string

json()

operations()

class beembase.objects.**Permission**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

class beembase.objects.**Price**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

class beembase.objects.**SocialActionCommentCreate**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

class beembase.objects.**SocialActionCommentDelete**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

class beembase.objects.**SocialActionCommentUpdate**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

class beembase.objects.**SocialActionVariant**(*o*)
Bases: beemgraphenebase.types.Static_variant

class beembase.objects.**WitnessProps**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

beembase.objecttypes

beembase.objecttypes.**object_type = {'account': 2, 'account_history': 18, 'block_summary'**
Object types for object ids

beembase.operationids

beembase.operationids.**getOperationNameForId**(*i*)
Convert an operation id into the corresponding string

beembase.operationids.**ops = ['vote', 'comment', 'transfer', 'transfer_to_vesting', 'withdra**
Operation ids

beembase.operations

beembase.operationids.**getOperationNameForId**(*i*)
Convert an operation id into the corresponding string

beembase.operationids.**ops = ['vote', 'comment', 'transfer', 'transfer_to_vesting', 'withdra**
Operation ids

beembase.signedtransactions

class beembase.signedtransactions.**Signed_Transaction**(**args*, ***kwargs*)
Bases: *beemgraphenebase.signedtransactions.Signed_Transaction*

Create a signed transaction and offer method to create the signature

Parameters

- **refNum** (*num*) – parameter ref_block_num (see *beembase.transactions.getBlockParams()*)
- **refPrefix** (*num*) – parameter ref_block_prefix (see *beembase.transactions.getBlockParams()*)
- **expiration** (*str*) – expiration date
- **operations** (*array*) – array of operations
- **custom_chains** (*dict*) – custom chain which should be added to the known chains

add_custom_chains(*custom_chain*)

getKnownChains()

getOperationKlass()

sign (*wifkeys*, *chain='STEEM'*)
Sign the transaction with the provided private keys.

Parameters

- **wifkeys** (*array*) – Array of wif keys
- **chain** (*str*) – identifier for the chain

verify (*pubkeys=[]*, *chain='STEEM'*, *recover_parameter=False*)
Returned pubkeys have to be checked if they are existing

beembase.transactions

beembase.transactions.**getBlockParams** (*ws*)
Auxiliary method to obtain `ref_block_num` and `ref_block_prefix`. Requires a websocket connection to a witness node!

3.7.4 beemgraphenebase Modules

beemgraphenebase.account

class beemgraphenebase.account.**Address** (*address=None*, *pubkey=None*, *prefix='STM'*)
Bases: `object`

Address class

This class serves as an address representation for Public Keys.

Parameters

- **address** (*str*) – Base58 encoded address (defaults to `None`)
- **pubkey** (*str*) – Base58 encoded pubkey (defaults to `None`)
- **prefix** (*str*) – Network prefix (defaults to `STM`)

Example:

```
Address("STMFN9r6VYzBK8EKtMewfNbfiGCr56pHDBFi")
```

derive256address_with_version (*version=56*)
Derive address using `RIPEMD160(SHA256(x))` and adding version + checksum

derivesha256address ()
Derive address using `RIPEMD160(SHA256(x))`

derivesha512address ()
Derive address using `RIPEMD160(SHA512(x))`

get_public_key ()
Returns the pubkey

class beemgraphenebase.account.**BrainKey** (*brainkey=None*, *sequence=0*)
Bases: `object`

Brainkey implementation similar to the graphene-ui web-wallet.

Parameters

- **brainkey** (*str*) – Brain Key

- **sequence** (*int*) – Sequence number for consecutive keys

Keys in Graphene are derived from a seed brain key which is a string of 16 words out of a predefined dictionary with 49744 words. It is a simple single-chain key derivation scheme that is not compatible with BIP44 but easy to use.

Given the brain key, a private key is derived as:

```
privkey = SHA256(SHA512(brainkey + " " + sequence))
```

Incrementing the sequence number yields a new key that can be regenerated given the brain key.

get_blind_private()
Derive private key from the brain key (and no sequence number)

get_brainkey()
Return brain key of this instance

get_private()
Derive private key from the brain key and the current sequence number

get_private_key()

get_public()

get_public_key()

next_sequence()
Increment the sequence number by 1

normalize(*brainkey*)
Correct formating with single whitespace syntax and no trailing space

suggest()
Suggest a new random brain key. Randomness is provided by the operating system using `os.urandom()`.

class beemgraphenebase.account.**PasswordKey**(*account*, *password*, *role='active'*, *prefix='STM'*)
Bases: `object`

This class derives a private key given the account name, the role and a password. It leverages the technology of Brainkeys and allows people to have a secure private key by providing a passphrase only.

get_private()
Derive private key from the brain key and the current sequence number

get_private_key()

get_public()

get_public_key()

class beemgraphenebase.account.**PrivateKey**(*wif=None*, *prefix='STM'*)
Bases: *beemgraphenebase.account.PublicKey*

Derives the compressed and uncompressed public keys and constructs two instances of *PublicKey*:

Parameters

- **wif** (*str*) – Base58check-encoded wif key
- **prefix** (*str*) – Network prefix (defaults to `STM`)

Example:

```
PrivateKey("5HqUkGuo62BfcJU5vNhTXKJRXuUi9QSE6jp8C3uBJ2BVHtB8WSd")
```

Compressed vs. Uncompressed:

- **PrivateKey("w-i-f").pubkey:** Instance of *PublicKey* using compressed key.
- **PrivateKey("w-i-f").pubkey.address:** Instance of *Address* using compressed key.
- **PrivateKey("w-i-f").uncompressed:** Instance of *PublicKey* using uncompressed key.
- **PrivateKey("w-i-f").uncompressed.address:** Instance of *Address* using uncompressed key.

child(*offset256*)
Derive new private key from this key and a sha256 "offset"

compressedpubkey()
Derive uncompressed public key

derive_from_seed(*offset*)
Derive private key using "generate_from_seed" method. Here, the key itself serves as a *seed*, and *offset* is expected to be a sha256 digest.

derive_private_key(*sequence*)
Derive new private key from this private key and an arbitrary sequence number

get_public_key()
Returns the pubkey

get_secret()
Get sha256 digest of the wif key.

class beemgraphenebase.account.**PublicKey**(*pk*, *prefix='STM'*)
Bases: *beemgraphenebase.account.Address*

This class deals with Public Keys and inherits `Address`.

Parameters

- **pk** (*str*) – Base58 encoded public key
- **prefix** (*str*) – Network prefix (defaults to `STM`)

Example:

```
PublicKey("STM6UtYWWs3rkZGV8JA86qrgkG6tyFksgECefKE1MiH4HkLD8PFGL")
```

Note: By default, graphene-based networks deal with **compressed** public keys. If an **uncompressed** key is required, the method *unCompressed()* can be used:

```
PublicKey("xxxxx").unCompressed()
```

---

compressed()
Derive compressed public key

get_public_key()
Returns the pubkey

point()
Return the point for the public key

unCompressed()
Derive uncompressed key

beemgraphenebase.base58

class beemgraphenebase.base58.**Base58**(*data*, *prefix='GPH'*)
Bases: object

Base58 base class

This class serves as an abstraction layer to deal with base58 encoded strings and their corresponding hex and binary representation throughout the library.

Parameters

- **data** (*hex, wif, bip38 encrypted wif, base58 string*) – Data to initialize object, e.g. pubkey data, address data, ...
- **prefix** (*str*) – Prefix to use for Address/PubKey strings (defaults to GPH)

Returns Base58 object initialized with data

Return type *Base58*

Raises **ValueError** – if data cannot be decoded

- bytes(Base58): Returns the raw data
- str(Base58): Returns the readable Base58CheckEncoded data.
- repr(Base58): Gives the hex representation of the data.
- format(Base58,_format): Formats the instance according to _format
  - "btc": prefixed with 0x80. Yields a valid btc address
  - "wif": prefixed with 0x00. Yields a valid wif key
  - "bts": prefixed with BTS
  - etc.

beemgraphenebase.base58.**b58decode**(*v*)

beemgraphenebase.base58.**b58encode**(*v*)

beemgraphenebase.base58.**base58CheckDecode**(*s*)

beemgraphenebase.base58.**base58CheckEncode**(*version*, *payload*)

beemgraphenebase.base58.**base58decode**(*base58_str*)

beemgraphenebase.base58.**base58encode**(*hexstring*)

beemgraphenebase.base58.**doublesha256**(*s*)

beemgraphenebase.base58.**gphBase58CheckDecode**(*s*)

beemgraphenebase.base58.**gphBase58CheckEncode**(*s*)

beemgraphenebase.base58.**log = <Logger beemgraphenebase.base58 (WARNING)>**
Default Prefix

beemgraphenebase.base58.**ripemd160**(*s*)

beemgraphenebase.bip38

exception beemgraphenebase.bip38.**SaltException**
Bases: Exception

beemgraphenebase.bip38.**decrypt**(*encrypted_privkey*, *passphrase*)
BIP0038 non-ec-multiply decryption. Returns WIF privkey.

Parameters

- **encrypted_privkey** (Base58) – Private key
- **passphrase** (*str*) – UTF-8 encoded passphrase for decryption

Returns BIP0038 non-ec-multiply decrypted key

Return type *Base58*

Raises ***SaltException*** – if checksum verification failed (e.g. wrong password)

beemgraphenebase.bip38.**encrypt**(*privkey*, *passphrase*)
BIP0038 non-ec-multiply encryption. Returns BIP0038 encrypted privkey.

Parameters

- **privkey** (Base58) – Private key
- **passphrase** (*str*) – UTF-8 encoded passphrase for encryption

Returns BIP0038 non-ec-multiply encrypted wif key

Return type *Base58*

beemgraphenebase.ecdsasig

beemgraphenebase.ecdsasig.**compressedPubkey**(*pk*)

beemgraphenebase.ecdsasig.**recoverPubkeyParameter**(*message*, *digest*, *signature*, *pubkey*)
Use to derive a number that allows to easily recover the public key from the signature

beemgraphenebase.ecdsasig.**recover_public_key**(*digest*, *signature*, *i*, *message=None*)
Recover the public key from the the signature

beemgraphenebase.ecdsasig.**sign_message**(*message*, *wif*, *hashfn=<built-in function openssl_sha256>*)
Sign a digest with a wif key

Parameters **wif** (*str*) – Private key in

beemgraphenebase.ecdsasig.**verify_message**(*message*, *signature*, *hashfn=<built-in function openssl_sha256>*, *recover_parameter=None*)

beemgraphenebase.objects

class beemgraphenebase.objects.**GrapheneObject**(*data=None*)
Bases: object

Core abstraction class

This class is used for any JSON reflected object in Graphene.

- instance.__json__(): encodes data into json format
- bytes(instance): encodes data into wire format

- `str(instances)`: dumps json object as string

json()

toJson()

class beemgraphenebase.objects.**Operation**(*op*)
Bases: object

getOperationNameForId(*i*)
Convert an operation id into the corresponding string

operations()

beemgraphenebase.objects.**isArgsThisClass**(*self*, *args*)

beemgraphenebase.objecttypes

beemgraphenebase.objecttypes.**object_type = {'OBJECT_TYPE_COUNT': 3, 'account': 2, 'base':**
Object types for object ids

beemgraphenebase.operations

beemgraphenebase.operationids.**operations = {'demooepration': 0}**
Operation ids

beemgraphenebase.signedtransactions

class beemgraphenebase.signedtransactions.**Signed_Transaction**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

Create a signed transaction and offer method to create the signature

Parameters

- **refNum** (*num*) – parameter ref_block_num (see *beembase.transactions.getBlockParams()*)
- **refPrefix** (*num*) – parameter ref_block_prefix (see *beembase.transactions.getBlockParams()*)
- **expiration** (*str*) – expiration date
- **operations** (*array*) – array of operations

derSigToHexSig(*s*)
Format DER to HEX signature

deriveDigest(*chain*)

getChainParams(*chain*)

getKnownChains()

getOperationKlass()

id
The transaction id of this transaction

sign(*wifkeys*, *chain=None*)
Sign the transaction with the provided private keys.

Parameters

- **wifkeys** (*array*) – Array of wif keys
- **chain** (*str*) – identifier for the chain

verify (*pubkeys=[]*, *chain=None*, *recover_parameter=False*)
Returned pubkeys have to be checked if they are existing

3.8 Contributing to beem

We welcome your contributions to our project.

3.8.1 Repository

The repository of beem is currently located at:

https://github.com/holgern/beem

3.8.2 Flow

This project makes heavy use of git flow. If you are not familiar with it, then the most important thing for your to understand is that:

pull requests need to be made against the develop branch

3.8.3 How to Contribute

0. Familiarize yourself with contributing on github
1. Fork or branch from the master.
2. Create commits following the commit style
3. Start a pull request to the master branch
4. Wait for a @holger80 or another member to review

3.8.4 Issues

Feel free to submit issues and enhancement requests.

3.8.5 Contributing

Please refer to each project's style guidelines and guidelines for submitting patches and additions. In general, we follow the "fork-and-pull" Git workflow.

1. **Fork** the repo on GitHub
2. **Clone** the project to your own machine
3. **Commit** changes to your own branch
4. **Push** your work back up to your fork

5. Submit a **Pull request** so that we can review your changes

Note: Be sure to merge the latest from "upstream" before making a pull request!

3.8.6 Copyright and Licensing

This library is open sources under the MIT license. We require your to release your code under that license as well.

3.9 Support and Questions

Help and discussion channel for beem can be found here:

- https://discord.gg/4HM592V

3.10 Indices and Tables

- genindex
- modindex

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W