beem Documentation

Release 0.1

Holger Nahrstaedt

May 09, 2018

Contents

Steem 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, it comes with decentralized publishing of content.

The Steem 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 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

- it's 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")
steem.transfer("<to>", "<amount>", "<asset>", "<memo>", account="<from>")
```

```
from beem.blockchain import Blockchain
blockchain = Blockchain()
for op in Blockchain.ops():
    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.market import Market
# Not working at the moment
# market = Market("STEEM:SBD")
# print(market.ticker())
# market.steem.wallet.unlock("wallet-passphrase")
# print(market.sell(300, 100)  # sell 100 STEEM for 300 STEEM/SBD
```

```
from beem.dex import Dex
# not working at the moment
# dex = Dex()
# dex.steem.wallet.unlock("wallet-passphrase")
```

CHAPTER 3

General

3.1 Installation

3.1.1 Installation

Install with *pip3*:

```
$ sudo apt-get install libffi-dev libssl-dev python-dev
$ pip3 install beem
```

or the with *pip*:

```
$ pip install -U beem
```

Manual installation:

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

3.1.2 Upgrade

```
$ pip install --user --upgrade
```

3.2 Quickstart

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.

```
from pprint import pprint
from beem import Steem

testnet = Steem(
    "wss://testnet.steem.vc",
    nobroadcast=True,
    bundle=True,
)

testnet.wallet.unlock("supersecret")

testnet.transfer("test1", 1, "STEEM", account="test")
testnet.transfer("test1", 1, "STEEM", account="test")
testnet.transfer("test1", 1, "STEEM", account="test")
testnet.transfer("test1", 1, "STEEM", account="test")

pprint(testnet.broadcast())
```

3.3.2 Proposing a Transaction

In Steem, you can propose a transactions to any account. This is used to facilitate on-chain multisig transactions. With python-steem, you can do this simply by using the `proposer` attribute:

```
from pprint import pprint
from beem import Steem

testnet = Steem(
    "wss://testnet.steem.vc",
    proposer="test"
)
testnet.wallet.unlock("supersecret")
pprint(testnet.transfer("tst1", 1, "STEEM", account="test"))
```

3.3.3 Simple Sell Script

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

#
```

(continues on next page)

(continued from previous page)

```
# Instanciate Steem (pick network via API node)
#
steem = Steem(
    "wss://node.testnet.steem.eu",
    nobroadcast=True   # <<--- set this to False when you want to fire!
)

#
# Unlock the Wallet
#
steem.wallet.unlock("<supersecret>")

#
# 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(
    "GOLD:USD",
    steem_instance=steem
)

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

3.3.4 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


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

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


if __name__ == "__main__":
    #
    # Instanciate Steem (pick network via API node)
    #
    steem = Steem(
```

(continues on next page)

(continued from previous page)

```
        "wss://node.testnet.steem.eu",
        nobroadcast=True    # <<--- set this to False when you want to fire!
    )

    #
    # Unlock the Wallet
    #
    steem.wallet.unlock("<supersecret>")

    #
    # 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(
        "GOLD:USD",
        steem_instance=steem
    )

    sell()
```

3.4 Configuration

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

- API node URL
- default account name
- the encrypted master password

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 `steem.Steem` class, you can load the configuration directly by using:

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

3.4.1 API

class beem.storage.**Configuration**

This is the configuration storage that stores key/value pairs in the *config* table of the SQLite3 database.

checkBackup()

Backup the SQL database every 7 days

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

nodes = ['wss://steemd.pevo.science', 'wss://gtg.steem.house:8090', 'wss://rpc.steemli
Default configuration

3.5 Contributing to python-steem

We welcome your contributions to our project.

3.5.1 Repository

The *main* repository of python-steem is currently located at:

https://github.com/holgern/beem

3.5.2 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

3.5.3 How to Contribute

0. Familiarize yourself with *contributing on github <https://guides.github.com/activities/contributing-to-open-source/>*
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

3.5.4 Issues

Feel free to submit issues and enhancement requests.

3.5.5 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!

3.5.6 Copyright and Licensing

This library is open sources under the MIT license. We require your to release your code under that license as well.

3.6 Support and Questions

We have currently not setup a distinct channel for development around pysteemi. However, many of the contributors are frequently reading through these channels:

CHAPTER 4

Packages

4.1 beem

4.1.1 beem package

Submodules

beem.account module

class beem.account.**Account**(*account*, *id_item='name'*, *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.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 it's corresponding functions.

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

available_balances
List balances of an account. This call returns instances of `steem.amount.Amount`.

balance(*balances*, *symbol*)
Obtain the balance of a specific Asset. This call returns instances of `steem.amount.Amount`.

balances

ensure_full()

getSimilarAccountNames(*limit=5*)
Returns limit similar accounts with name as array

history(*limit=100*, *only_ops=[]*, *exclude_ops=[]*)
Returns a generator for individual account transactions. The latest operation will be first. This call can be used in a `for` loop.

Parameters

- **limit** (*int/datetime*) – limit number of transactions to return (*optional*)
- **only_ops** (*array*) – Limit generator by these operations (*optional*)
- **exclude_ops** (*array*) – Exclude thse operations from generator (*optional*)

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

name

profile
Returns the account profile

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

rep

reputation(*precision=2*)

reward_balances

saving_balances

total_balances

type_id = 2

beem.aes module

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 module

class beem.amount.**Amount**(**args*, *amount=None*, *asset=None*, *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)
- **steem_instance** (*steem.steem.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
```

(continues on next page)

(continued from previous page)

```
a * 2
a += b
a /= 2.0
```

amount
Returns the amount as float

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.asset module

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** (`beem.steem.Steem`) – Steem instance

Returns All data of an asset

Return type dict

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()`.

---

precision

refresh()
Refresh the data from the API server

symbol

type_id = 3

beem.steem module

class beem.steem.**Steem**(*node=''*, *rpcuser=''*, *rpcpassword=''*, *debug=False*, ***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)*
- **debug** (*bool*) – Enable Debugging *(optional)*
- **keys** (*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)*
- **proposer** (*str*) – Propose a transaction using this proposer *(optional)*
- **proposal_expiration** (*int*) – Expiration time (in seconds) for the proposal *(optional)*
- **proposal_review** (*int*) – Review period (in seconds) for the proposal *(optional)*
- **expiration** (*int*) – Delay in seconds until transactions are supposed to expire *(optional)*
- **blocking** (*str*) – Wait for broadcasted transactions to be included in a block and return full transaction (can be "head" or "irrversible")
- **bundle** (*bool*) – Do not broadcast transactions right away, but allow to bundle operations *(optional)*

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`, or `memo` keys for any account. This mode is only used for *foreign* signatures!

If no node is provided, it will connect to the node of http://uptick.rocks. It is **highly** recommended that you pick your own node instead. Default settings can be changed with:

```
uptick set node <host>
```

where `<host>` starts with `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.info())
```

All that is requires is for the user to have added a key with `uptick`

```
uptick addkey
```

and setting a default author:

```
uptick set default_account xeroc
```

This class also deals with edits, votes and reading content.

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 `active`)
- **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

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

Parameters

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

broadcast (*tx=None*)
Broadcast a transaction to the Steem network

Parameters **tx** (*tx*) – Signed transaction to broadcast

chain_params

clear ()

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*, ***kwargs*)
Create new account on Steem

The brainkey/password can be used to recover all generated keys (see *beembase.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. However, it **does not import the owner key** for security reasons. Do NOT expect to be able to recover it from the wallet if you lose your password!

Parameters

- **account_name** (*str*) – (**required**) new account name
- **registrar** (*str*) – which account should pay the registration fee (defaults to `default_account`)
- **owner_key** (*str*) – Main owner key
- **active_key** (*str*) – Main active 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_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`)

Raises ***AccountExistsException*** – if the account already exists on the blockchain

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 `active`)
- **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

- **witnesses** (*list*) – list of Witness name or id
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_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** (*operation*) – The operation (or list of operaions) to broadcast
- **account** (*operation*) – The account that authorizes the operation
- **permission** (*string*) – The required permission for signing (active, owner, posting)
- **append_to** (*object*) – This allows to provide an instance of ProposalsBuilder (see `steem.new_proposal()`) or 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 beem.txbuffer as instance of *beem.transactionbuilder.TransactionBuilder*. You may want to use your own txbuffer

info()
Returns the global properties

newWallet(*pwd*)
Create a new wallet. This method is basically only calls `beem.wallet.create()`.

Parameters **pwd** (*str*) – Password to use for the new wallet

Raises ***beem.exceptions.WalletExists*** – if there is already a wallet created

new_tx(**args*, ***kwargs*)
Let's obtain a new txbuffer

Returns int txid id of the new txbuffer

prefix

set_default_account(*account*)
Set the default account to be used

sign(*tx=None*, *wifs=[]*)
Sign a provided transaction witht he 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.

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`

tx()
Returns the default transaction buffer

txbuffer
Returns the currently active tx buffer

unlock(**args*, ***kwargs*)
Unlock the internal wallet

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`)

beem.block module

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

Read a single block from the chain

Parameters

- **block** (*int*) – block number
- **steem_instance** (`beem.steem.Steem`) – Steem instance
- **lazy** (*bool*) – Use lazy loading

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

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

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()`.

ops()

ops_statistics(*add_to_ops_stat=None*)

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

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

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

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

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

beem.blockchain module

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

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

Parameters

- **steem_instance** (beem.steem.Steem) – Steem instance
- **mode** (*str*) – (default) Irreversible block (irreversible) or actual head block (head)

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

from beem.blockchain import Blockchain chain = Blockchain()

Read current block and blockchain info .. code-block:: python

print(chain.get_current_block()) print(chain.info())

Monitor for new blocks .. .. code-block:: python

for block in chain.blocks(): print(block)

or each operation individually: .. code-block:: python

for operations in chain.ops(): print(operations)

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

---

Note: If you want instant confirmation, you need to instantiate class:*steem.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.

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

blocks(*start=None*, *stop=None*)
Yields blocks starting from `start`.

Parameters

- **start** (*int*) – Starting block
- **stop** (*int*) – Stop at this block
- **mode** (*str*) – We here have the choice between "head" (the last block) and "irreversible" (the block that is confirmed by 2/3 of all block producers and is thus irreversible)

get_all_accounts(*start=''*, *stop=''*, *steps=1000.0*, ***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_chain_properties()
Return witness elected chain properties

::

{'account_creation_fee': '30.000 STEEM', 'maximum_block_size': 65536, 'sbd_interest_rate': 250}

get_config()
Returns internal chain configuration.

get_current_block()
This call returns the current block

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

get_current_block_num()
This call returns the current block

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

get_current_median_history_price()
Returns the current median price

get_dynamic_global_properties()
This call returns the *dynamic global properties*

get_feed_history()
Returns the feed_history

get_hardfork_version()
Current Hardfork Version as String

get_network()
Identify the network

Returns Network parameters

Return type dict

get_next_scheduled_hardfork()
Returns Hardfork and live_time of the hardfork

get_state(*path='value'*)

ops(*start=None*, *stop=None*, ***kwargs*)
Yields all operations (including virtual operations) starting from `start`.

Parameters

- **start** (*int*) – Starting block
- **stop** (*int*) – Stop at this block
- **mode** (*str*) – We here have the choice between "head" (the last block) and "irreversible" (the block that is confirmed by 2/3 of all block producers and is thus irreversible)
- **only_virtual_ops** (*bool*) – Only yield virtual operations

This call returns a list that only carries one operation and its type!

ops_statistics(*start*, *stop=None*, *add_to_ops_stat=None*, *verbose=True*)
Generates a 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

:param dict add_to_ops_stat, if set, the result is added to add_to_ops_stat :param bool verbose, if True, the current block number and timestamp is printed This call returns a dict with all possible operations and their occurence.

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

Parameters

- **opNames** (*array*) – List of operations to filter for
- **start** (*int*) – Start at this block
- **stop** (*int*) – Stop at this block
- **mode** (*str*) – We here have the choice between "head" (the last block) and "irreversible" (the block that is confirmed by 2/3 of all block producers and is thus irreversible)

The dict output is formated such that `type` caries the operation type, timestamp and block_num are taken from the block the operation was stored in and the other key depend on the actualy operation.

beem.comment module

class beem.comment.**Comment**(*authorperm*, *full=False*, *lazy=False*, *steem_instance=None*)
Bases: beem.blockchainobject.BlockchainObject

Read data about a Comment/Post in the chain

Parameters

- **authorperm** (*str*) – perm link to post/comment
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

author

authorperm

body

category

id

is_comment()
Retuns True if post is a comment

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

json()

json_metadata

parent_author

parent_permlink

permlink

refresh()

title

type_id = 8

class beem.comment.**RecentByPath**(*path='promoted'*, *category=None*, *steem_instance=None*)
Bases: list

Obtain a list of votes for an account

Parameters

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

class beem.comment.**RecentReplies**(*author*, *skip_own=True*, *steem_instance=None*)
Bases: list

Obtain a list of recent replies

Parameters

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

beem.exceptions module

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.**BlockDoesNotExistsException**
Bases: Exception

The block does not exist

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.**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.**KeyNotFound**
Bases: Exception

Key not found

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 `peerplays.wallet.create()` to create a new wallet

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

Object was not found in proposal

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.**WalletExists**
Bases: Exception

A wallet has already been created and requires a password to be unlocked by means of steem.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

beem.market module

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

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

Parameters

- **steem_instance** (*steem.steem.Steem*) – Steem instance
- **base** (*steem.asset.Asset*) – Base asset
- **quote** (*steem.asset.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 it's 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 steem.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. `USD:BTS` with `USD` being the quote), while the `base` only refers to a secondary asset for a trade. This means, if you call `steem.market.Market.sell()` or `steem.market.Market.buy()`, you will sell/buy **only quote** and obtain/pay **only base**.

accountopenorders(*account=None*)
Returns open Orders

Parameters **account** (*steem.account.Account*) – Account name or instance of Account to show orders for in this market

accounttrades(*account=None*, *limit=25*)
Returns your trade history for a given market, specified by the "currencyPair" parameter. You may also specify "all" to get the orderbooks of all markets.

Parameters

- **currencyPair** (*str*) – Return results for a particular market only (default: "all")
- **limit** (*int*) – Limit the amount of orders (default: 25)

Output Parameters:

- *type*: sell or buy
- *rate*: price for *quote* denoted in *base* per *quote*
- *amount*: amount of quote
- *total*: amount of base at asked price (amount/price)

Note: This call goes through the trade history and searches for your account, if there are no orders within `limit` trades, this call will return an empty array.

buy(*price*, *amount*, *expiration=None*, *killfill=False*, *account=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 USD_BTS market is priced in BTS per USD.

Example: in the USD_BTS market, a price of 300 means a USD is worth 300 BTS

Note: All prices returned are in the **reversed** orientation as the market. I.e. in the BTC/BTS market, prices are BTS per BTC. 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 USD for 100 BTS/USD
- This means that you actually place a sell order for 1000 BTS in order to obtain **at least** 10 USD
- If an order on the market exists that sells USD for cheaper, you will end up with more than 10 USD

cancel (*orderNumber*, *account=None*)
Cancels an order you have placed in a given market. Requires only the "orderNumber". An order number takes the form `1.7.xxx`.

Parameters **orderNumber** (*str*) – The Order Object ide of the form `1.7.xxxx`

core_base_market ()
This returns an instance of the market that has the core market of the base asset. It means that base needs to be a market pegged asset and returns a market to it's collateral asset.

core_quote_market ()
This returns an instance of the market that has the core market of the quote asset. It means that quote needs to be a market pegged asset and returns a market to it's collateral asset.

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

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

orderbook (*limit=25*)
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)

Sample output:

```
{'bids': [0.003679 USD/BTS (1.9103 USD|519.29602 BTS),
0.003676 USD/BTS (299.9997 USD|81606.16394 BTS),
0.003665 USD/BTS (288.4618 USD|78706.21881 BTS),
0.003665 USD/BTS (3.5285 USD|962.74409 BTS),
0.003665 USD/BTS (72.5474 USD|19794.41299 BTS)],
'asks': [0.003738 USD/BTS (36.4715 USD|9756.17339 BTS),
0.003738 USD/BTS (18.6915 USD|5000.00000 BTS),
0.003742 USD/BTS (182.6881 USD|48820.22081 BTS),
0.003772 USD/BTS (4.5200 USD|1198.14798 BTS),
0.003799 USD/BTS (148.4975 USD|39086.59741 BTS)]}
```

Note: Each bid is an instance of class:*steem.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*, *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 USD_BTS market is priced in BTS per USD.

Example: in the USD_BTS market, a price of 300 means a USD is worth 300 BTS

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

---

ticker()

Returns the ticker for all markets.

Output Parameters:

- last: Price of the order last filled
- lowestAsk: Price of the lowest ask
- highestBid: Price of the highest bid
- baseVolume: Volume of the base asset
- quoteVolume: Volume of the quote asset
- percentChange: 24h change percentage (in %)
- settlement_price: Settlement Price for borrow/settlement
- core_exchange_rate: Core exchange rate for payment of fee in non-BTS asset
- price24h: the price 24h ago

Sample Output:

```
{
    {
        "quoteVolume": 48328.73333,
        "quoteSettlement_price": 332.3344827586207,
        "lowestAsk": 340.0,
        "baseVolume": 144.1862,
        "percentChange": -1.9607843231354893,
        "highestBid": 334.20000000000005,
        "latest": 333.33333330133934,
    }
}
```

trades(*limit=25*, *start=None*, *stop=None*)

Returns your trade history for a given market.

Parameters

- **limit** (*int*) – Limit the amount of orders (default: 25)
- **start** (*datetime*) – start time
- **stop** (*datetime*) – stop time

volume24h()

Returns the 24-hour volume for all markets, plus totals for primary currencies.

Sample output:

```
{
    "BTS": 361666.63617,
    "USD": 1087.0
}
```

beem.memo module

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** (beem.account.Account) – Account that has sent the memo
- **to_account** (beem.account.Account) – Account that has received the memo
- **steem_instance** (beem.steem.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(memo.decrypt(op_data["memo"]))
```

if op_data being the payload of a transfer operation.

In BitShares, 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 publick 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 retreived with the *get_account* call:

```
get_account <accountname>
{
  [...]
  "options": {
    "memo_key": "GPH5TPTziKkLexhVKsQKtSpo4bAv5RnB8oXcG4sMHEwCcTf3r7dqE",
    [...]
  },
  [...]
}
```

while the memo private key can be dumped with *dump_private_keys*

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.

The high level memo class makes use of the pysteem 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)
```

```
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

# Instantiate Memo for decoding
memo = Memo()

# Unlock wallet
memo.unlock_wallet(getpass())

# Decode memo
```

(continues on next page)

(continued from previous page)

```
# Raises exception if required keys not available in the wallet
print(memo.decrypt(op_data["memo"]))
```

decrypt(*memo*)
Decrypt a memo

Parameters **memo** (*str*) – encrypted memo message

Returns encrypted memo

Return type str

encrypt(*memo*)
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 module

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.notify module

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** (beem.steem.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 module

class beem.price.**FilledOrder**(*order*, *steem_instance=None*, ***kwargs*)
Bases: *beem.price.Price*

This class inherits steem.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.Steem*) – Steem instance

Note: Instances of this class come with an additional time key that shows when the order has been filled!

class beem.price.**Order**(**args*, *steem_instance=None*, ***kwargs*)
Bases: *beem.price.Price*

This class inherits steem.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.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**(**args*, *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** (beem.asset.Asset) – Base asset
- **quote** (beem.asset.Asset) – Quote asset
- **steem_instance** (beem.steem.Steem) – Steem instance

Returns All data required to represent a price

Return type dict

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
>>> Price("0.3314 SBD/STEEM") * 2
0.662600000 SBD/STEEM
```

as_base(*base*)
Returns the price instance so that the base asset is `base`.

Note: This makes a copy of the object!

as_quote(*quote*)
Returns the price instance so that the quote asset is `quote`.

Note: This makes a copy of the object!

copy() → a shallow copy of D

invert()
Invert the price (e.g. go from `SBD/STEEM` into `STEEM/SBD`)

json()

market
Open the corresponding market

Returns Instance of `steem.market.Market` for the corresponding pair of assets.

symbols()

class beem.price.**PriceFeed**(*feed*, *steem_instance=None*)
Bases: `dict`

This class is used to represent a price feed consisting of

- a witness,
- a symbol,
- a core exchange rate,
- the maintenance collateral ratio,
- the max short squeeze ratio,
- a settlement price, and
- a date

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

class beem.price.**UpdateCallOrder**(*call*, *steem_instance=None*, ***kwargs*)
Bases: *beem.price.Price*

This class inherits `steem.price.Price` but has the `base` and `quote` Amounts not only be used to represent the **call price** (as a ratio of base and quote).

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

beem.storage module

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.

checkBackup()
Backup the SQL database every 7 days

config_defaults = {'node': ['wss://steemd.pevo.science', 'wss://gtg.steem.house:8090'

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()

nodes = ['wss://steemd.pevo.science', 'wss://gtg.steem.house:8090', 'wss://rpc.steemli
Default configuration

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()
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

refreshBackup()
Make a new backup

sqlDataBaseFile = '/home/docs/.local/share/beem/beem.sqlite'

sqlite3_backup(*dbfile*, *backupdir*)
Create timestamped database copy

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()
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

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 = ''

purge()
Remove the masterpassword from the configuration store

saveEncrytpedMaster()
Store the encrypted master password in the configuration store

beem.transactionbuilder module

class beem.transactionbuilder.**TransactionBuilder**(*tx={}*, *proposer=None*, *expiration=None*, *steem_instance=None*)

Bases: dict

This class simplifies the creation of transactions by adding operations and signers. To build your own transactions and sign them

```
from beem.transactionbuilder import TransactionBuilder
from beembase.operations import Transfer
tx = TransactionBuilder()
tx.appendOps(Transfer(**{
         "from": "test",
         "to": "test1",
         "amount": "1 STEEM",
         "memo": ""
     }))
tx.appendSigner("test", "active")
tx.sign()
tx.broadcast()
```

addSigningInformation(*account*, *permission*)
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)

FIXME: Does not work with owner keys!

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

appendWif(*wif*)
Add a wif that should be used for signing of the transaction.

broadcast()
Broadcast a transaction to the steem network

Parameters **tx** (*tx*) – Signed transaction to broadcast

clear()
Clear the transaction builder and start from scratch

constructTx()
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

is_empty()

json()
Show the transaction as plain json

list_operations()

set_expiration(*p*)

sign()
Sign a provided transaction witht he 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.

verify_authority()
Verify the authority of the signed transaction

beem.utils module

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*, *username_prefix='@'*)
Create a post identifier from comment/post object or arguments. Examples:

:: construct_authorperm('username', 'permlink') construct_authorperm({'author': 'username',

'permlink': 'permlink'})

beem.utils.**construct_authorpermvoter**(**args*, *username_prefix='@'*)
Create a vote identifier from vote object or arguments. Examples:

:: construct_authorpermvoter('username', 'permlink', 'voter') construct_authorpermvoter({'author': 'username',

'permlink': 'permlink', 'voter': 'voter'})

beem.utils.**derive_permlink**(*title*, *parent_permlink=None*, *parent_author=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.**parse_time**(*block_time*)
Take a string representation of time from the blockchain, and parse it into datetime object.

beem.utils.**resolve_authorperm**(*identifier*)
Correctly split a string containing an authorperm.

Splits the string into author and permlink with the following separator: `/`.

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.**sanitize_permlink**(*permlink*)

beem.utils.**test_proposal_in_buffer**(*buf*, *operation_name*, *id*)

beem.vote module

class beem.vote.**AccountVotes**(*account*, *steem_instance=None*)
Bases: `list`

Obtain a list of votes for an account

Parameters

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

class beem.vote.**ActiveVotes**(*authorperm*, *steem_instance=None*)
Bases: `list`

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
v = Vote("theaussiegame/cryptokittie-giveaway-number-2|")
```

authorpermvoter

json()

percent

refresh()

reputation

rshares

time

type_id = 11

voter

weight

beem.wallet module

class beem.wallet.**Wallet**(*rpc=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*, *string*) – 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 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!

A new wallet can be created by using:

```
from beem import Steem
steem = Steem()
steem.wallet.create("supersecret-passphrase")
```

This will raise an exception 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 steem.wallet.Wallet.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

changePassphrase(*new_pwd*)
Change the passphrase for the wallet database

configStorage = None

create(*pwd*)
Alias for newWallet()

created()
Do we have a wallet database already?

decrypt_wif(*encwif*)
decrypt a wif 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)

getAccountFromPrivateKey(*wif*)
Obtain account name from private key

getAccountFromPublicKey(*pub*)
Obtain the first account name from public key

getAccounts()
Return all accounts installed in the wallet database

getAccountsFromPublicKey(*pub*)
Obtain all accounts associated with a public key

getActiveKeyForAccount(*name*)
Obtain owner Active Key for an account from the wallet database

getAllAccounts(*pub*)
Get the account data for a public key (all accounts found for this public key)

getKeyType(*account*, *pub*)
Get key type

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

getPrivateKeyForPublicKey(*pub*)
Obtain the private key for a given public key

Parameters **pub** (*str*) – Public Key

getPublicKeys()
Return all installed public keys

keyMap = {}

keyStorage = None

keys = {}

lock()
Lock the wallet database

locked()
Is the wallet database locked?

masterpassword = None

newWallet(*pwd*)
Create a new wallet database

purgeWallet()
Purge all data in wallet database

removeAccount(*account*)
Remove all keys associated with a given account

removePrivateKeyFromPublicKey(*pub*)
Remove a key from the wallet database

rpc = None

setKeys(*loadkeys*)
This method is strictly only for in memory keys that are passed to Wallet/Steem with the `keys` argument

tryUnlockFromEnv()

unlock(*pwd=None*)
Unlock the wallet database

unlocked()
Is the wallet database unlocked?

beem.witness module

class beem.witness.**LookupWitnesses**(*from_account*, *limit*, *steem_instance=None*)
Bases: *beem.witness.WitnessesObject*

Obtain a list of witnesses which have been voted by an account

Parameters

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

class beem.witness.**Witness**(*owner*, *id_item='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")
```

account

refresh()

type_id = 3

class beem.witness.**Witnesses**(*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

class beem.witness.**WitnessesByIds**(*witness_ids*, *steem_instance=None*)
Bases: *beem.witness.WitnessesObject*

Obtain a list of witnesses which have been voted by an account

Parameters

- **witness_ids** (*list*) – list of witness_ids
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

class beem.witness.**WitnessesObject**
Bases: list

printAsTable(*sort_key='votes'*, *reverse=True*)

class beem.witness.**WitnessesRankedByVote**(*from_account=''*, *limit=100*, *steem_instance=None*)
Bases: *beem.witness.WitnessesObject*

Obtain a list of witnesses ranked by Vote

Parameters

- **from_account** (*str*) – Witness name
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

class beem.witness.**WitnessesVotedByAccount**(*account*, *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

Module contents

4.2 beembase

4.2.1 beembase package

Submodules

beembase.account module

class beembase.account.**Address**(**args*, ***kwargs*)
Bases: graphenebase.account.Address

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("GPHFN9r6VYzBK8EKtMewfNbfiGCr56pHDBFi")
```

class beembase.account.**BrainKey**(**args*, ***kwargs*)
Bases: graphenebase.account.BrainKey

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.

class beembase.account.**PasswordKey**(**args*, ***kwargs*)
Bases: graphenebase.account.PasswordKey

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.

class beembase.account.**PrivateKey**(**args*, ***kwargs*)
Bases: graphenebase.account.PrivateKey

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.

class beembase.account.**PublicKey**(**args*, ***kwargs*)

Bases: graphenebase.account.PublicKey

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("GPH6UtYWWs3rkZGV8JA86qrgkG6tyFksgECefKE1MiH4HkLD8PFGL")
```

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()
```

---

beembase.bip38 module

beembase.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)

beembase.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

beembase.chains module

beembase.memo module

beembase.memo.**decode_memo**(*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*)
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

Return type AES

beembase.objects module

beembase.objects.**AccountId**(*asset*)

class beembase.objects.**AccountOptions**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

class beembase.objects.**Amount**(*d*)
Bases: object

beembase.objects.**AssetId**(*asset*)

class beembase.objects.**Beneficiaries**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

class beembase.objects.**Beneficiary**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

class beembase.objects.**CommentOptionExtensions**(*o*)
Bases: graphenebase.types.Static_variant

Serialize Comment Payout Beneficiaries. Args:

beneficiaries (list): A static_variant containing beneficiaries.

Example:

::

[0,

{'beneficiaries': [{'account': 'furion', 'weight': 10000}

]}

]

class beembase.objects.**ExchangeRate**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

class beembase.objects.**Extension**(*d*)
Bases: graphenebase.types.Array

class beembase.objects.**Memo**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

class beembase.objects.**ObjectId**(*object_str*, *type_verify=None*)
Bases: graphenebase.types.ObjectId

Encodes object/protocol ids

class beembase.objects.**Operation**(**args*, ***kwargs*)
Bases: graphenebase.objects.Operation

getOperationNameForId(*i*)
Convert an operation id into the corresponding string

json()

operations()

class beembase.objects.**Permission**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

class beembase.objects.**Price**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

class beembase.objects.**WitnessProps**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

beembase.objecttypes module

beembase.objecttypes.**object_type = {'account': 2, 'account_history': 18, 'block_summary'**
Object types for object ids

beembase.operationids module

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 module

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.transactions module

Module contents

4.3 beemapi

4.3.1 beemapi package

Submodules

SteemNodeRPC

This class allows to call API methods exposed by the witness node via websockets.

Defintion

class beemapi.steemnoderpc.**SteemNodeRPC**(**args*, ***kwargs*)
This class allows to call API methods exposed by the witness node via websockets.

__getattr__(*name*)
Map all methods to RPC calls and pass through the arguments

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

beemapi.exceptions module

exception beemapi.exceptions.**MissingRequiredActiveAuthority**
Bases: grapheneapi.exceptions.RPCError

exception beemapi.exceptions.**NoAccessApi**
Bases: grapheneapi.exceptions.RPCError

exception beemapi.exceptions.**NoMethodWithName**
Bases: grapheneapi.exceptions.RPCError

exception beemapi.exceptions.**NumRetriesReached**
Bases: Exception

exception beemapi.exceptions.**UnhandledRPCError**
Bases: grapheneapi.exceptions.RPCError

beemapi.exceptions.**decodeRPCErrorMsg**(*e*)
Helper function to decode the raised Exception and give it a python Exception class

SteemWebsocket

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()
```

Defintion

class beemapi.websocket.**SteemWebsocket**(*urls*, *user=''*, *password=''*, **args*, *only_block_id=False*, *on_block=None*, *keep_alive=25*, *num_retries=-1*, ***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()
```

Notices:

- on_block:

```
'0062f19df70ecf3a478a84b4607d9ad8b3e3b607'
```

_SteemWebsocket__set_subscriptions()

__events__ = ['on_block']

__getattr__(*name*)

Map all methods to RPC calls and pass through the arguments

__init__(*urls*, *user=''*, *password=''*, **args*, *only_block_id=False*, *on_block=None*, *keep_alive=25*, *num_retries=-1*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'beemapi.websocket'

_ping()

cancel_subscriptions()

close()

Closes the websocket connection and waits for the ping thread to close

get_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=[]*)

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

Module contents

CHAPTER 5

Glossary

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

S

T

U

V

W