python-bitshares Documentation

Release 0.1

Fabian Schuh

Oct 07, 2020

Contents

BitShares is a **blockchain-based autonomous company** (i.e. a DAC) that offers decentralized exchanging as well as sophisticated financial instruments as *products*.

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 BitShares, it comes with decentralized trading of assets as well as customized on-chain smart contracts.

CHAPTER 1

About this Library

The purpose of *pybitshares* is to simplify development of products and services that use the BitShares 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

General

2.1 Installation

2.1.1 Installation

Install with *pip*:

```
$ sudo apt-get install libffi-dev libssl-dev python3-dev
$ pip3 install bitshares
```

Manual installation:

```
$ git clone https://github.com/xeroc/python-bitshares/
$ cd python-bitshares
$ python3 setup.py install --user
```

2.1.2 Upgrade

```
$ pip3 install bitshares --user --upgrade
```

2.2 Quickstart

2.3 Tutorials

2.3.1 Bundle Many Operations

With BitShares, 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 bitshares import BitShares

testnet = BitShares(
    "wss://node.testnet.bitshares.eu",
    nobroadcast=True,
    bundle=True,
)

testnet.wallet.unlock("supersecret")

testnet.transfer("init0", 1, "TEST", account="xeroc")
testnet.transfer("init1", 1, "TEST", account="xeroc")
testnet.transfer("init2", 1, "TEST", account="xeroc")
testnet.transfer("init3", 1, "TEST", account="xeroc")

pprint(testnet.broadcast())
```

2.3.2 Proposing a Transaction

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

```
from pprint import pprint
from bitshares import BitShares

testnet = BitShares(
    "wss://node.testnet.bitshares.eu",
    proposer="xeroc"
)
testnet.wallet.unlock("supersecret")
pprint(testnet.transfer("init0", 1, "TEST", account="xeroc"))
```

2.3.3 Simple Sell Script

```
from bitshares import BitShares
from bitshares.market import Market
from bitshares.price import Price
from bitshares.amount import Amount

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

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

(continues on next page)

(continued from previous page)

```
#
# 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",
    bitshares_instance=bitshares
)

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

2.3.4 Sell at a timely rate

```
import threading
from bitshares import BitShares
from bitshares.market import Market
from bitshares.price import Price
from bitshares.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 BitShares (pick network via API node)
    #
    bitshares = BitShares(
        "wss://node.testnet.bitshares.eu",
        nobroadcast=True    # <<--- set this to False when you want to fire!
    )

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

    #
    # 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(
        "GOLD:USD",
        bitshares_instance=bitshares
    )

    sell()
```

2.4 Configuration

The pybitshares 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, **persistently**.

You can access those variables like a regular dictionary by using

```
from bitshares import BitShares
bitshares = BitShares()
print(bitshares.config.items())
```

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

```
bitshares.config["my-new-variable"] = "important-content"
print(bitshares.config["my-new-variable"])
```

2.5 FAQ

2.5.1 How to get order info on filled order

On CEX exchanges full order info usually available for canceled / filled orders. On BitShares such info is not available, because such info is stored in memory of bitshares-core node, and keeping non-actual orders info would take astonishing amounts of RAM.

Thus, such order info could be obtained in two ways:

- By querying account history from the node:

```
from bitshares.account import Account

a = Account('dexbot')
ops = a.history(only_ops=['fill_order'], limit=1)
for op in ops:
    print(op)
```

Note: this way has limitation: public nodes doesn't store full account history, only limited number of entries

- By querying elasticsearch plugin. In short, elasticsearch plugin export account history data into elasticsearch instance, from which it's can be obtained directly or via elasticsearch wrapper. See https://eswrapper.bitshares.eu/apidocs/ to get info on how to query the wrapper. A real-world example of elasticsearch wrapper usage for obtaining filled orders history is bitshares-tradehistory-analyzer

2.5.2 How to detect partially filled order

An Order have the following fields:

- `order['base']['amount']`: stores initial amount to sell
- `order['for_sale']['amount']`: stores remaining amount to sell

So, if your order initially sells 100 BTS, and 50 BTS was sold, the `order['for_sale']['amount']` will contain remaining 50 BTS.

2.6 Asyncio support

The library has full support of asyncio, though you need to be aware it has some limitations.

2.6.1 Example

A very basic example:

```
import asyncio

from bitshares.aio import BitShares
from bitshares.aio.instance import set_shared_bitshares_instance


async def info(loop, bitshares):
    await bitshares.connect()
    set_shared_bitshares_instance(bitshares)
    print(await bitshares.info())


def main():
    loop = asyncio.get_event_loop()
    bitshares = BitShares(loop=loop)
    loop.run_until_complete(info(loop, bitshares))


if __name__ == '__main__':
    main()
```

2.6.2 Instantiation of BitShares

To be able to perform calls, you need to explicitly run *await BitShares.connect()*. That is, creating of instance object and actual network connection are separate operations in async version:

```
bitshares = BitShares(loop=loop)
await bitshares.connect()
```

2.6.3 Limitations

- Most of the classes requires async init because during instantiation some API calls has to be performed:

```
await Amount('10 FOO')
```

- Several math operations are not available for `bitshares.aio.Amount`, `bitshares.aio.Price` objects. This includes multiplication, division etc. This limitation is due to unability to define python magic methods (`__mul__`, `__div__`, etc) as async coroutines
- Most of properties are awaitables too:

```
asset = await Asset('CNY')
await asset.max_market_fee
```

2.6.4 Subscriptions

In asyncio version subscription notifications are not handled in callback-based manner. Instead, they are available in *self.notifications* queue which is `asyncio.Queue`. You can use a single bitshares instance both for setting subscriptions and performing other API calls.

Here is the example of how to subscribe and handle events:

```
market = await Market("TEST/USD")
await bitshares.subscribe_to_market(market, event_id=4)

while True:
    event = await bitshares.notifications.get()
    print(event)
```

2.6.5 Debugging

To enable debugging on RPC level, you can raise loglevel on following loggers (don't forget to set formatter as well):

```
log = logging.getLogger("websockets")
log.setLevel(logging.DEBUG)

log = logging.getLogger("grapheneapi")
log.setLevel(logging.DEBUG)
```

2.6.6 Tests

Asyncio version has a dedicated testsuite which uses real API integration tests which are performed against local bitshares-core testnet. Bitshares node is spawned automatically inside docker container. You don't need to setup anything.

Before running tests you need to install dependencies via *pip intstall -r requirements-test.txt*

Run tests via *pytest -v tests/testnet/aio/*

2.7 BitShares classes inheritance

This document briefly describes how bitshares.xxx classes are inherited

2.7.1 AbstractBlockchainInstanceProvider (graphenelib) role

Typical class inheritance is *Foo(GrapheneFoo) -> GrapheneFoo(AbstractBlockchainInstanceProvider) -> AbstractBlockchainInstanceProvider*

This class provides access to RPC via *self.blockchain* property, which is set to *blockchain_instance* kwarg or *shared_blockchain_instance* as a fallback. *shared_blockchain_instance* in turn gets proper blockchain instance class calling *self.get_instance_class()*. *get_instance_class()* is overwritten in *bitshares.instance.BlockchainInstance*

inject method (used as *@BlockchainInstance.inject* decorator) is needed to provide blockchain instance class in common manner.

In short, *Foo* + *@BlockchainInstance.inject* init calls *AbstractBlockchainInstanceProvider.__init__* and *Foo.__init__*.

AbstractBlockchainInstanceProvider.__init__ sets *self._blockchain* from kwarg or via calling *self.shared_blockchain_instance()*, which leads to initizlizing *bitshares.Bitshares* (*bitshares.Bitshares* is inherited from *AbstractGrapheneChain*.

2.7.2 Asyncio versions

Typical async class inherited from corresponding async class from graphenelib, and from synchronous class, like *class Asset(GrapheneAsset, SyncAsset)*. So, async version needs to redefine only needed methods.

Most of async classes needs async *__init__* because they're loading some objects from the blockchain, which requires an API call performed via async RPC. To achieve this, async *AbstractBlockchainInstanceProvider* has different *inject()* method.

2.8 Contributing

Contributing

Introduction

You are here to help BitShares and the python-bitshares library? Awesome, feel welcome and read the following sections in order to know how to ask questions and how to work on something.

Code of Conduct

All members of our community are expected to follow our [Code of Conduct](CODE_OF_CONDUCT.md). Please make sure you are welcoming and friendly in all of our spaces.

Get in touch

Feel free to get in contact with the developer community on [Telegram](https://t.me/pybitshares).

Contributing to development

When contributing to this repository, please first discuss the change you wish to make via issue, email, or any other method with the owners of this repository before making a change.

Please note we have a code of conduct, please follow it in all your interactions with the project.

Your First Contribution

To familiarize yourself with the code and procedures, we recommend to get started with:

- review a Pull Request
- fix an Issue
- update the documentation

- make a website
- write a tutorial

Git Flow

This project makes heavy use of [git flow](http://nvie.com/posts/a-successful-git-branching-model/). If you are not familiar with it, then the most important thing for you to understand is that:

pull requests need to be made against the 'develop' branch!

Contributor License Agreement

Upon submission of a pull request, you will be asked to sign the [Contributor License Agreement](CLA.md) digitally through a github-connected service.

1. Where do I go from here?

If you've noticed a bug or have a question, [search the issue tracker][https://github.com/bitshares/python-bitshares/issues] to see if someone else in the community has already created a ticket. If not, go ahead and [make one][https://github.com/bitshares/python-bitshares/issues/new]!

2. Fork & create a branch

If this is something you think you can fix, then fork the repository and create a branch with a descriptive name.

A good branch name would be (where issue #325 is the ticket you're working on):

    git checkout -b 325-new-fancy-feature

3. Get the test suite running

Make sure to add a unit tests for your your code contribution in *tests/*. You may use other unit tests as template for your own tests.

Individual unit tests can be run by:

    python3 -m unittests tests/test_NEWFEATURE.py

The entire test suite can be run in a sandbox through

    tox

4. Did you find a bug?

- **Ensure the bug was not already reported** by [searching all issues][https://github.com/bitshares/python-bitshares/issues].
- If you're unable to find an open issue addressing the problem, [open a new one][https://github.com/bitshares/python-bitshares/issues/new]. Be sure to include a **title and clear description**, as much relevant information as possible, and a **code sample** or an **executable test case** demonstrating the expected behavior that is not occurring.
- If possible, use the relevant bug report templates to create the issue. Simply copy the content of the appropriate template into a .py file, make the necessary changes to demonstrate the issue, and **paste the content into the issue description**.

5. Implement your fix or feature

At this point, you're ready to make your changes! Feel free to ask for help; everyone is a beginner at first

6. Get the style right

Your patch should follow the same conventions & pass the same code quality checks as the rest of the project. [Codeclimate](https://codeclimate.com/github/bitshares/python-bitshares) will give you feedback in this regard. You can check & fix codeclimate's feedback by running it locally using [Codeclimate's CLI](https://codeclimate.com/github/bitshares/python-bitshares), via *codeclimate analyze*.

7. Make a Pull Request

Pull requests are supposed to go against the 'develop' branch, only!

At this point, you should switch back to your *develop* branch and make sure it's up to date with python-bitshares's *develop* branch:

> git remote add upstream git@github.com:bitshares/python-bitshares.git git checkout develop git pull upstream develop

Then update your feature branch from your local copy of develop, and push it!

> git checkout 325-new-fancy-feature git rebase develop git push –set-upstream origin 325-new-fancy-feature

Finally, go to GitHub and make a Pull Request :D

Travis CI will run our test suite against all supported Rails versions. We care about quality, so your PR won't be merged until all tests pass. It's unlikely, but it's possible that your changes pass tests in one Rails version but fail in another. In that case, you'll have to setup your development environment (as explained in step 3) to use the problematic Rails version, and investigate what's going on!

7. Keeping your Pull Request updated

If a maintainer asks you to "rebase" your PR, they're saying that a lot of code has changed, and that you need to update your branch so it's easier to merge.

To learn more about rebasing in Git, there are a lot of good git rebasing resources but here's the suggested workflow:

> git checkout 325-new-fancy-feature git pull –rebase upstream develop git push –force-with-lease 325-new-fancy-feature

8. Merging a PR (maintainers only)

A PR can only be merged into *develop* by a maintainer if:

- Pull request goes against *develop* branch.
- It is passing CI.
- It has been approved by at least two maintainers. If it was a maintainer who opened the PR, only one extra approval is needed.
- It has no requested changes.
- It is up to date with current *develop*.
- Did the contributor sign the [CLA](CLA.md)

Any maintainer is allowed to merge a PR if all of these conditions are met.

9. Shipping a release (maintainers only)

Maintainers need to do the following to push out a release:

> git checkout develop git fetch origin git rebase origin/develop # get latest tag git tag -l # git flow git flow release start x.y.z # bump version in setup.py git add setup.py; git commit -m "version bump" git flow release finish

Bundling and pushing the release:

> make release

2.9 Support and Questions

- https://t.me/pybitshares - dedicated channel
- https://bitsharestalk.org - BitShares forum
- https://t.me/BitSharesDEX - general telegram channel

CHAPTER 3

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.

3.1 Create a wallet

```
from bitshares import BitShares
bitshares = BitShares()
bitshares.wallet.create("secret-passphrase")
bitshares.wallet.addPrivateKey("<wif-key>")
```

3.2 Unlock the wallet for a transfer

```
from bitshares import BitShares
bitshares = BitShares()
bitshares.wallet.unlock("wallet-passphrase")
bitshares.transfer("<to>", "<amount>", "<asset>", "[<memo>]", account="<from>")
```

3.3 Monitor the BitShares Blockchain operation-wise

```
from bitshares.blockchain import Blockchain
blockchain = Blockchain()
for op in Blockchain.ops():
    print(op)
```

3.4 Obtain the content of one block

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

3.5 Obtain Account balance, open orders and history

```
from bitshares.account import Account
account = Account("init0")
print(account.balances)
print(account.openorders)
for h in account.history():
    print(h)
```

3.6 Print Market ticker and sell

```
from bitshares.market import Market
market = Market("USD:BTS")
print(market.ticker())
market.bitshares.wallet.unlock("wallet-passphrase")
print(market.sell(300, 100))  # sell 100 USD for 300 BTS/USD
```

3.7 Adjust collateral

```
from bitshares.dex import Dex
dex = Dex()
dex.bitshares.wallet.unlock("wallet-passphrase")
dex.adjust_collateral_ratio("SILVER", 3.5)
```

CHAPTER 4

Developers and Community

Discussions around development and use of this library can be found in a [dedicated Telegram Channel](https://t.me/pybitshares)

CHAPTER 5

Packages

5.1 bitshares

5.1.1 bitshares package

Subpackages

bitshares.aio package

Submodules

bitshares.aio.account module

class bitshares.aio.account.**Account**(**args*, ***kwargs*)

Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.account.Account*

This class allows to easily access Account data.

Parameters

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

Returns Account data

Return type dictionary

Raises **bitshares.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 bitshares.aio.account import Account
account = await Account("init0")
print(account)
```

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 `await Account.refresh()`.

---

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

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

bitshares
Alias for the specific blockchain.

blacklist(*account*)
Add an other account to the blacklist of this account

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

call_positions
Alias for :func:bitshares.account.Account.callpositions.

callpositions
List call positions (collateralized positions *Market Pegged Assets*)

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

ensure_full()

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

history(*first=0*, *last=0*, *limit=-1*, *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

- **first** (*int*) – sequence number of the first transaction to return (*optional*)
- **last** (*int*) – sequence number of the last transaction to return (*optional*)
- **limit** (*int*) – limit number of transactions to return (*optional*)
- **only_ops** (*array*) – Limit generator by these operations (*optional*)
- **exclude_ops** (*array*) – Exclude these operations from generator (*optional*).

... **note::** only_ops and exclude_ops takes an array of strings: The full list of operation ID's can be found in operationids.py. Example: ['transfer', 'fill_order']

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

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

is_ltm
Is the account a lifetime member (LTM)?

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

name

nolist(*account*)
Remove an other account from any list of this account

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

openorders
Returns open Orders.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

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

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

upgrade()
Upgrade account to life time member

values() → an object providing a view on D's values

whitelist(*account*)
Add an other account to the whitelist of this account

class bitshares.aio.account.**AccountUpdate**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.account.AccountUpdate*

This purpose of this class is to keep track of account updates as they are pushed through by *bitshares.notify.Notify*.

Instances of this class are dictionaries and take the following form:

account
In oder to obtain the actual `account.Account` from this class, you can use the `account` attribute.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

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

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

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.aio.amount module

class bitshares.aio.amount.**Amount**(**args*, ***kwargs*)

Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.amount.Amount*

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)
- **blockchain_instance** (bitshares.aio.bitshares.BitShares) – BitShares instance

Returns All data required to represent an Amount/Asset

Return type dict

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

```
from bitshares.aio.amount import Amount
from bitshares.aio.asset import Asset
a = await Amount("1 USD")
b = await Amount(1, "USD")
c = await Amount("20", await Asset("USD"))
```

Way to obtain a proper instance:

- `args` can be a string, e.g.: "1 USD"
- `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 *bitshares.aio.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 *bitshares.aio.asset.Asset*)

amount
Returns the amount as float

asset
Returns the asset as instance of *asset.Asset*

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

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

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

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

json()

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

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

symbol
Returns the symbol of the asset

tuple()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.aio.asset module

class bitshares.aio.asset.**Asset**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.asset.Asset*

BitShares asset.

Async version of `bitshares.bitshares.Asset`

add_authorities(*type*, *authorities=None*)
Add authorities to an assets white/black list.

Parameters

- **type** (*str*) – `blacklist` or `whitelist`
- **authorities** (*list*) – List of authorities (Accounts)

add_markets(*type*, *authorities=None*, *force_enable=True*)
Add markets to an assets white/black list.

Parameters

- **type** (*str*) – `blacklist` or `whitelist`
- **markets** (*list*) – List of markets (assets)
- **force_enable** (*bool*) – Force enable `white_list` flag

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

calls

chain
Short form for blockchain (for the lazy)

change_issuer(*new_issuer*, ***kwargs*)
Change asset issuer (needs signing with owner key!)

Parameters **new_issuer** (*str*) – account name

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

disableflag(*flag*)
Enable a certain flag.

Parameters **flag** (*str*) – Flag name

enableflag(*flag*)
Enable a certain flag.

Parameters **flag** (*str*) – Flag name

ensure_full()

feed

feeds

flags
List the permissions that are currently used (flags)

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_call_orders(*limit=100*)

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

get_settle_orders(*limit=100*)

getfromcache(*id*)
Get an element from the cache explicitly

halt()
Halt this asset from being moved or traded.

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

is_bitasset
Is the asset a market pegged asset?

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

issue(*amount*, *to*, *memo=None*, ***kwargs*)
Issue new shares of an asset.

Parameters

- **amount** (*float*) – Amount to issue
- **to** (*str*) – Recipient
- **memo** (*str*) – (optional) Memo message

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

market_fee_percent

max_market_fee

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

permissions
List the permissions for this asset that the issuer can obtain

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

precision

refresh()
Refresh the data from the API server

release(*whitelist_authorities=None*, *blacklist_authorities=None*, *whitelist_markets=None*, *blacklist_markets=None*)
Release this asset and allow unrestricted transfer, trading, etc.

Parameters

- **whitelist_authorities** (*list*) – List of accounts that serve as whitelist authorities
- **blacklist_authorities** (*list*) – List of accounts that serve as blacklist authorities
- **whitelist_markets** (*list*) – List of assets to allow trading with
- **blacklist_markets** (*list*) – List of assets to prevent trading with

remove_authorities(*type*, *authorities=None*)
Remove authorities from an assets white/black list.

Parameters

- **type** (*str*) – `blacklist` or `whitelist`
- **authorities** (*list*) – List of authorities (Accounts)

remove_markets(*type*, *authorities=None*)
Remove markets from an assets white/black list.

Parameters

- **type** (*str*) – `blacklist` or `whitelist`
- **markets** (*list*) – List of markets (assets)

seize(**args*)
Seize amount from an account and send to another.

... note:: This requires the override_authority to be set for this asset!

Parameters

- **from_account** (bitshares.account.Account) – From this account
- **to_account** (bitshares.account.Account) – To this account
- **amount** (bitshares.amount.Amount) – Amount to seize

set_market_fee(*percentage_fee*, *max_market_fee*)
Set trading percentage fee.

Parameters

- **percentage_fee** (*float*) – Percentage of fee
- **max_market_fee** (bitshares.amount.Amount) – Max Fee

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of SharedInstance.instance.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

setoptions(*flags*)
Enable a certain flag.

Flags:

- charge_market_fee
- white_list
- override_authority
- transfer_restricted
- disable_force_settle
- global_settle
- disable_confidential
- witness_fed_asset
- committee_fed_asset

Parameters **flag** (*dict*) – dictionary of flags and boolean

settlements

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

symbol

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

update_cer(*cer*, *account=None*, ***kwargs*)
Update the Core Exchange Rate (CER) of an asset

update_feed_producers(*producers*)
Update bitasset feed producers.

Parameters **producers** (*list*) – List of accounts that are allowed to produce a feed

values() → an object providing a view on D's values

bitshares.aio.bitshares module

class bitshares.aio.bitshares.**BitShares**(**args*, ***kwargs*)
Bases: graphenecommon.aio.chain.AbstractGrapheneChain, *bitshares.bitshares.BitShares*

BitShares async client.

This is an asyncio version of bitshares.BitShares

Parameters **loop** (*object*) – asyncio event loop

Example usage:

```
bitshares = BitShares(loop=loop)
await bitshares.connect()
```

account_whitelist(*account_to_whitelist*, *lists=None*, *account=None*, ***kwargs*)
Account whitelisting.

Parameters

- **account_to_whitelist** (*str*) – The account we want to add to either the white- or the blacklist
- **lists** (*set*) – (defaults to ('white')). Lists the user should be added to. Either empty set, 'black', 'white' or both.
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

allow (*foreign*, *weight=None*, *permission='active'*, *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

approvecommittee (*committees*, *account=None*, ***kwargs*)
Approve a committee.

Parameters

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

approveproposal (*proposal_ids*, *account=None*, *approver=None*, ***kwargs*)
Approve Proposal.

Parameters

- **proposal_id** (*list*) – Ids of the proposals
- **appprover** (*str*) – The account or key to use for approval (defaults to `account`)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

approvewitness (*witnesses*, *account=None*, ***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`)

approveworker (*workers*, *account=None*, ***kwargs*)
Approve a worker.

Parameters

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

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

bid_collateral (*additional_collateral*, *debt_covered*, *account=None*, ***kwargs*)

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

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

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

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

cancel_subscriptions()
Cancel all active subscriptions.

clear()

clear_cache()
Clear Caches

connect()
Connect to blockchain network

Async version does wallet initialization after connect because wallet depends on prefix which is available after connection only, and we want to keep __init__() synchronous.

create_account(*account_name*, *registrar=None*, *referrer='1.2.35641'*, *referrer_percent=50*, *owner_key=None*, *active_key=None*, *memo_key=None*, *owner_account=None*, *active_account=None*, *password=None*, *additional_owner_keys=None*, *additional_active_keys=None*, *additional_owner_accounts=None*, *additional_active_accounts=None*, *proxy_account='proxy-to-self'*, *storekeys=True*, ***kwargs*)
Create new account on BitShares.

The brainkey/password can be used to recover all generated keys (see *bitsharesbase.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

create_asset(*symbol*, *precision*, *max_supply*, *description=''*, *is_bitasset=False*, *is_prediction_market=False*, *market_fee_percent=0*, *max_market_fee=None*, *permissions=None*, *flags=None*, *whitelist_authorities=None*, *blacklist_authorities=None*, *whitelist_markets=None*, *blacklist_markets=None*, *bitasset_options=None*, *account=None*, ***kwargs*)

Create a new asset.

Parameters

- **symbol** (*str*) – Asset symbol
- **precision** (*int*) – Asset precision
- **max_supply** (*int*) – Asset max supply
- **description** (*str*) – (optional) Asset description
- **is_bitasset** (*bool*) – (optional) True = bitasset, False = UIA (default: False)
- **is_prediction_market** (*bool*) – (optional) True: PD, False = plain smartcoin (default: False)
- **market_fee_percent** (*float*) – (optional) Charge market fee (0-100) (default: 0)
- **max_market_fee** (*float*) – (optional) Absolute amount of max market fee, value of this option should be a whole number (default: same as max_supply)
- **permissions** (*dict*) – (optional) Asset permissions
- **flags** (*dict*) – (optional) Enabled asset flags
- **whitelist_authorities** (*list*) – (optional) List of accounts that serve as whitelist authorities
- **blacklist_authorities** (*list*) – (optional) List of accounts that serve as blacklist authorities
- **whitelist_markets** (*list*) – (optional) List of assets to allow trading with
- **blacklist_markets** (*list*) – (optional) List of assets to prevent trading with
- **bitasset_options** (*dict*) – (optional) Bitasset settings
- **account** (*str*) – (optional) the issuer account to (defaults to `default_account`)

create_committee_member(*url=''*, *account=None*, ***kwargs*)

Create a committee member.

Parameters

- **url** (*str*) – URL to read more about the worker
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

create_liquidity_pool(*asset_a*, *asset_b*, *share_asset*, *taker_fee_percent*, *withdrawal_fee_percent*, *account=None*, ***kwargs*)

Create a liquidity pool

Parameters

- **asset_a** (*str*) – First asset in the pool pair.
- **asset_b** (*str*) – Second asset in the pool pair.
- **share_asset** (*str*) – The asset which represents shares in the pool.

For asset parameters, these can be either symbols or asset_id strings. Note that network expects asset_a to have a lower-numbered asset_id than asset_b.

Parameters

- **taker_fee_percent** (*float*) – The pool's taker fee percentage.
- **withdrawal_fee_percent** (*float*) – The pool's withdrawal fee percent.

For percentages, meaningful range is [0.00, 100.00], where 1% is represented as 1.0. Smallest non-zero value recognized by BitShares chain is 0.01 for 0.01%.

create_voting_ticket(*target_type*, *amount_to_lock*, *account=None*, ***kwargs*)

Create a voting ticket

Parameters

- **target_type** (*int*, *str*) – Lock period target. Should be a string from operations.ticket_type_strings or the index of the intended string.
- **amount_to_lock** (Amount) – Amount to lock up for the duration selected in target_type.

create_worker(*name*, *daily_pay*, *end*, *url=''*, *begin=None*, *payment_type='vesting'*, *pay_vesting_period_days=0*, *account=None*, ***kwargs*)

Create a worker.

This removes the shares from the supply

Required

Parameters

- **name** (*str*) – Name of the worker
- **daily_pay** (bitshares.amount.Amount) – The amount to be paid daily
- **end** (*datetime*) – Date/time of end of the worker

Optional

Parameters

- **url** (*str*) – URL to read more about the worker
- **begin** (*datetime*) – Date/time of begin of the worker
- **payment_type** (*string*) – ["burn", "refund", "vesting"] (default: "vesting")
- **pay_vesting_period_days** (*int*) – Days of vesting (default: 0)
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

define_classes()

delete_liquidity_pool (*pool*, *account=None*, ***kwargs*)
Delete a liquidity pool

Parameters **pool** (*str*, Asset) – The liquidity pool to delete. Can be the pool id as a string, or can be an Asset, asset_id, or symbol of the share asset for the pool.

deposit_into_liquidity_pool (*pool*, *amount_a*, *amount_b*, *account=None*, ***kwargs*)
Deposit assets into a liquidity pool

Parameters

- **pool** (*str*, Asset) – The liquidity pool to use. Can be the pool id as a string, or can be an Asset, asset_id, or symbol of the share asset for the pool.
- **amount_a** (Amount) –
- **amount_b** (Amount) –

disallow (*foreign*, *permission='active'*, *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

disapprovecommittee (*committees*, *account=None*, ***kwargs*)
Disapprove a committee.

Parameters

- **committees** (*list*) – list of committee name or id
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

disapproveproposal (*proposal_ids*, *account=None*, *approver=None*, ***kwargs*)
Disapprove Proposal.

Parameters

- **proposal_ids** (*list*) – Ids of the proposals
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

disapprovewitness (*witnesses*, *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)

disapproveworker (*workers*, *account=None*, ***kwargs*)
Disapprove a worker.

Parameters

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

exchange_with_liquidity_pool(*pool*, *amount_to_sell*, *min_to_receive*, *account=None*, ***kwargs*)

Exchange assets against a liquidity pool

Parameters

- **pool** (*str,* `Asset`) – The liquidity pool to use. Can be the pool id as a string, or can be an Asset, asset_id, or symbol of the share asset for the pool.
- **amount_to_sell** (`Amount`) –
- **min_to_receive** (`Amount`) –

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 *new_proposal()*) or TransactionBuilder (see *new_tx()*) to specify where to put a specific operation.

... note:: append_to is exposed to every method used in the this 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 txbuffer as instance of `transactionbuilder.TransactionBuilder`. You may want to use your own txbuffer

fund_fee_pool(*symbol*, *amount*, *account=None*, ***kwargs*)

Fund the fee pool of an asset.

Parameters

- **symbol** (*str*) – The symbol to fund the fee pool of
- **amount** (*float*) – The amount to be burned.
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

htlc_create(*amount*, *to*, *preimage*, *hash_type='ripemd160'*, *account=None*, *expiration=3600*, ***kwargs*)

Create an HTLC contract.

Parameters

- **amount** (Amount) – Amount to lock
- **to** (*str*) – Recipient
- **expiration** (*int*) – Contract duration in seconds
- **hash_hex** (*str*) – Hash as string of hex digits
- **preimage** (*str*) – Preimage as ascii string. Note hex digits would be interpretted as ascii text, not as bytes. Not generally recommended to use this option. Options hash_hex and preimage are mutually exclusive.
- **preimage_length** (*int*) – If non-zero, htlc contract will require preimage of exact length. Generally OK to leave this as zero. Note if preimage param is provided, this value SHOULD be either zero or match exactly the length of the preimage, else an irredeemable htlc will be created. Optionally, a sentinal value of -1 can be used to compute length automatically from the preimage param.

htlc_redeem(*htlc_id*, *preimage*, *account=None*, ***kwargs*)
Redeem an htlc contract

Parameters

- **preimage** (*str*) – The preimage that unlocks the htlc
- **encoding** (*str*) – "utf-8", . . . , or "hex"

info()
Returns the global properties

is_connected()

newWallet(*pwd*)

new_proposal(*parent=None*, *proposer=None*, *proposal_expiration=None*, *proposal_review=None*, ***kwargs*)

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

Returns int txid id of the new txbuffer

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

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

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

prefix
Contains the prefix of the blockchain

propbuffer
Return the default proposal buffer

proposal(*proposer=None*, *proposal_expiration=None*, *proposal_review=None*)
Return the default proposal buffer

. . . note:: If any parameter is set, the default proposal parameters will be changed!

publish_price_feed(*symbol*, *settlement_price*, *cer=None*, *mssr=110*, *mcr=200*, *account=None*)
Publish a price feed for a market-pegged asset.

Parameters

- **symbol** (*str*) – Symbol of the asset to publish feed for
- **settlement_price** (bitshares.price.Price) – Price for settlement
- **cer** (bitshares.price.Price) – Core exchange Rate (default settlement_price + 5%)
- **mssr** (*float*) – Percentage for max short squeeze ratio (default: 110%)
- **mcr** (*float*) – Percentage for maintenance collateral ratio (default: 200%)
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

Note: The account needs to be allowed to produce a price feed for symbol. For witness produced feeds this means account is a witness account!

registrar = None
Generate new keys from password

reserve(*amount*, *account=None*, ***kwargs*)
Reserve/Burn an amount of this shares.

This removes the shares from the supply

Parameters

- **amount** (bitshares.amount.Amount) – The amount to be burned.
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

set_blocking(*block=True*)
This sets a flag that forces the broadcast to block until the transactions made it into a block

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

set_proxy(*proxy_account*, *account=None*, ***kwargs*)
Set a specific proxy for account.

Parameters

- **proxy_account** (bitshares.account.Account) – Account to be proxied
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

set_shared_instance()
This method allows to set the current instance as default

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.

subscribe_to_accounts(*accounts*, *event_id=1*)
Activate subscription to account-related events.

Parameters

- **accounts** (*list*) – account names or ids to subscribe
- **event_id** (*int*) – id of this subscription in upcoming notifications

subscribe_to_blocks(*event_id=2*)
Activate subscription to block.

Each time block is applied an event will occur in self.notifications.

Parameters **event_id** (*int*) – id of this subscription in upcoming notifications

subscribe_to_market(*market*, *event_id=4*)
Activate subscription on market events.

Parameters

- **market** (*str,bitshares.aio.Market*) – market to set subscription on
- **event_id** (*int*) – id of this subscription in upcoming notifications

subscribe_to_pending_transactions(*event_id=0*)
Activate subscription to pending transactions.

Each time transaction is pushed to database an event will occur in self.notifications.

Parameters **event_id** (*int*) – id of this subscription in upcoming notifications

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

unset_proxy(*account=None*, ***kwargs*)
Unset the proxy account to start voting yourself.

update_cer(*symbol*, *cer*, *account=None*)
Update the Core Exchange Rate (CER) of an asset.

Parameters

- **symbol** (*str*) – Symbol of the asset to publish feed for
- **cer** (bitshares.price.Price) – Core exchange Rate
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

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`)

update_voting_ticket(*ticket_id*, *new_target_type*, *amount_to_update*, *account=None*, ***kwargs*)
Update a voting ticket

Parameters

- **ticket_id** (*str*) – Id (e.g. "1.18.xxx") of the ticket to update.
- **target_type** (*int*, *str*) – New lock period target. Should be a string from operations.ticket_type_strings or the index of the intended string.
- **amount_to_update** (`Amount`, *None*) – Amount to move over to the new lock-up target. (Optional - absence implies update whole amount.)

update_witness(*witness_identifier*, *url=None*, *key=None*, ***kwargs*)
Upgrade a witness account.

Parameters

- **witness_identifier** (*str*) – Identifier for the witness
- **url** (*str*) – New URL for the witness
- **key** (*str*) – Public Key for the signing

upgrade_account(*account=None*, ***kwargs*)
Upgrade an account to Lifetime membership.

Parameters **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

vesting_balance_withdraw(*vesting_id*, *amount=None*, *account=None*, ***kwargs*)
Withdraw vesting balance.

Parameters

- **vesting_id** (*str*) – Id of the vesting object
- **Amount** (`bitshares.amount.Amount`) – to withdraw ("all" if not provided")
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

withdraw_from_liquidity_pool(*pool*, *share_amount*, *account=None*, ***kwargs*)
Withdraw stake from a liquidity pool

Parameters

- **pool** (*str*, `Asset`) – The liquidity pool to use. Can be the pool id as a string, or can be an Asset, asset_id, or symbol of the share asset for the pool.
- **share_amount** (`Amount`) – Amount of share asset to redeem. Must be a quantity of the pool's share_asset.

bitshares.aio.block module

class bitshares.aio.block.**Block**(**args*, ***kwargs*)

Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.block.Block*

Read a single block from the chain.

Parameters

- **block** (*int*) – block number
- **blockchain_instance** (bitshares.aio.bitshares.BitShares) – BitShares instance
- **lazy** (*bool*) – Use lazy loading
- **loop** – async event loop

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 bitshares.aio.block import Block
block = await Block(1)
print(block)
```

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

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

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

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

type_id = 'n/a'

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.aio.block.**BlockHeader**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.block.BlockHeader*

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

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

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

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

type_id = 'n/a'

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.aio.blockchain module

class bitshares.aio.blockchain.**Blockchain**(**args*, ***kwargs*)

Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.blockchain.Blockchain*

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

Parameters

- **blockchain_instance** (bitshares.aio.bitshares.BitShares) – BitShares instance
- **mode** (*str*) – (default) Irreversible block (`irreversible`) or actual head block (`head`)
- **max_block_wait_repetition** (*int*) – (default) 3 maximum wait time for next block ismax_block_wait_repetition * block_interval

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

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:.*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.

bitshares

Alias for the specific blockchain.

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

blockchain

blockchain_instance_class

alias of *bitshares.aio.instance.BlockchainInstance*

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)

chain
Short form for blockchain (for the lazy)

chainParameters()
The blockchain parameters, such as fees, and committee-controlled parameters are returned here

config()
Returns object 2.0.0

define_classes()
Needs to define instance variables that provide classes

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_block_interval()
This call returns the block interval

get_chain_properties()
Return chain properties

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_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

get_network()
Identify the network

Returns Network parameters

Return type dict

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

classmethod inject(*cls*)

is_irreversible_mode()

ops(*start=None*, *stop=None*, ***kwargs*)
Yields all operations (excluding 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!

participation_rate

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

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 * "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.

update_chain_parameters()

wait_for_and_get_block(*block_number*, *blocks_waiting_for=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*) – (default) difference between block_number and current head how many blocks we are willing to wait, positive int

bitshares.aio.blockchainobject module

class bitshares.aio.blockchainobject.**BlockchainObject**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.blockchainobject.BlockchainObject*

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.aio.blockchainobject.**Object**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.blockchainobject.Object*

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = False

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

refresh()
This is the refresh method that overloads the prototype in BlockchainObject.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.aio.committee module

class bitshares.aio.committee.**Committee**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.committee.Committee*

Read data about a Committee Member in the chain.

Parameters

- **member** (*str*) – Name of the Committee Member
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC
- **lazy** (*bool*) – Use lazy loading

account

account_id

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

refresh()

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.aio.dex module

class bitshares.aio.dex.**Dex**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*

This class simplifies interactions with the decentralized exchange.

Parameters **blockchain_instance** (bitshares.aio.bitshares.BitShares) – BitShares instance

---

Note: The methods of this class only deal with a single asset (at most). If you are looking to deal with orders for trading, please use *bitshares.aio.market.Market*.

---

adjust_collateral_ratio(*symbol*, *new_collateral_ratio*, *account=None*, *target_collateral_ratio=None*)
Adjust the collataral ratio of a debt position.

Parameters

- **symbol** (*str*) – Symbol to adjust collateral for
- **new_collateral_ratio** (*float*) – desired collateral ratio
- **target_collateral_ratio** (*float*) – Tag the call order so that in case of margin call, only enough debt is covered to get back to this ratio

Raises

- **ValueError** – if symbol is not a bitasset
- **ValueError** – if collateral ratio is smaller than maintenance collateral ratio
- **ValueError** – if required amounts of collateral are not available

adjust_debt(*delta*, *new_collateral_ratio=None*, *account=None*, *target_collateral_ratio=None*)
Adjust the amount of debt for an asset.

Parameters

- **delta** (Amount) – Delta amount of the debt (-10 means reduce debt by 10, +10 means borrow another 10)
- **new_collateral_ratio** (*float*) – collateral ratio to maintain (optional, by default tries to maintain old ratio)
- **target_collateral_ratio** (*float*) – Tag the call order so that in case of margin call, only enough debt is covered to get back to this ratio

Raises

- **ValueError** – if symbol is not a bitasset
- **ValueError** – if collateral ratio is smaller than maintenance collateral ratio
- **ValueError** – if required amounts of collateral are not available

bitshares
Alias for the specific blockchain.

blockchain

borrow(*amount*, *collateral_ratio=None*, *account=None*, *target_collateral_ratio=None*)
Borrow bitassets/smartcoins from the network by putting up collateral in a CFD at a given collateral ratio.

Parameters

- **amount** (Amount) – Amount to borrow (denoted in 'asset')
- **collateral_ratio** (*float*) – Collateral ratio to borrow at
- **target_collateral_ratio** (*float*) – Tag the call order so that in case of margin call, only enough debt is covered to get back to this ratio

Raises

- **ValueError** – if symbol is not a bitasset
- **ValueError** – if collateral ratio is smaller than maintenance collateral ratio
- **ValueError** – if required amounts of collateral are not available

chain
Short form for blockchain (for the lazy)

close_debt_position(*symbol*, *account=None*)
Close a debt position and reclaim the collateral.

Parameters **symbol** (*str*) – Symbol to close debt position for

Raises **ValueError** – if symbol has no open call position

define_classes()
Needs to define instance variables that provide classes

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

list_debt_positions(*account=None*)
List Call Positions (borrowed assets and amounts)

Returns Struct of assets with amounts and call price

Return type dict

Example:

returnFees()
Returns a dictionary of all fees that apply through the network.

Example output:

```
{'proposal_create': {'fee': 400000.0},
'asset_publish_feed': {'fee': 1000.0}, 'account_create':
{'basic_fee': 950000.0, 'price_per_kbyte': 20000.0,
'premium_fee': 40000000.0}, 'custom': {'fee': 20000.0},
'asset_fund_fee_pool': {'fee': 20000.0},
'override_transfer': {'fee': 400000.0}, 'fill_order':
{}, 'asset_update': {'price_per_kbyte': 20000.0, 'fee':
200000.0}, 'asset_update_feed_producers': {'fee':
10000000.0}, 'assert': {'fee': 20000.0},
'committee_member_create': {'fee': 100000000.0}}
```

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

bitshares.aio.genesisbalance module

class bitshares.aio.genesisbalance.**GenesisBalance**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.genesisbalance.GenesisBalance*

Read data about a Genesis Balances from the chain.

Parameters

- **identifier** (*str*) – identifier of the balance
- **blockchain_instance** (*bitshares*) – bitshares() instance to use when accesing a RPC

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

claim(*account=None*, ***kwargs*)
Claim a balance from the genesis block

Parameters

- **balance_id** (*str*) – The identifier that identifies the balance to claim (1.15.x)
- **account** (*str*) – (optional) the account that owns the bet (defaults to `default_account`)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

refresh()

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = 15

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.aio.genesisbalance.**GenesisBalances**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.genesisbalance.GenesisBalances*

List genesis balances that can be claimed from the keys in the wallet.

append()
Append object to the end of the list.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

define_classes()
Needs to define instance variables that provide classes

extend()
Extend list by appending elements from the iterable.

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

index()
Return first index of value.

Raises ValueError if the value is not present.

classmethod inject(*cls*)

insert()
Insert object before index.

pop()
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

sort()
Stable sort *IN PLACE*.

bitshares.aio.htlc module

class bitshares.aio.htlc.**Htlc**(**args*, ***kwargs*)
Bases: *bitshares.aio.blockchainobject.BlockchainObject*

Read data about an HTLC contract on the chain.

Parameters

- **id** (*str*) – id of the HTLC
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

refresh()

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = 16

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.aio.instance module

class bitshares.aio.instance.**BlockchainInstance**(**args*, ***kwargs*)
Bases: graphenecommon.aio.instance.AbstractBlockchainInstanceProvider

This is a class that allows compatibility with previous naming conventions.

bitshares
Alias for the specific blockchain.

blockchain

chain
Short form for blockchain (for the lazy)

define_classes()
Needs to define instance variables that provide classes

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of SharedInstance.instance.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

class bitshares.aio.instance.**SharedInstance**
Bases: object

This class merely offers a singelton for the Blockchain Instance.

config = {}

instance = None

bitshares.aio.instance.**set_shared_bitshares_instance**(*instance*)

bitshares.aio.instance.**set_shared_blockchain_instance**(*instance*)

bitshares.aio.instance.**set_shared_config**(*config*)

bitshares.aio.instance.**shared_bitshares_instance**()

bitshares.aio.instance.**shared_blockchain_instance**()

bitshares.aio.market module

class bitshares.aio.market.**Market**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.market.Market*

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

Parameters

- **blockchain_instance** (bitshares.aio.bitshares.BitShares) – BitShares instance
- **base** (bitshares.aio.asset.Asset) – Base asset
- **quote** (bitshares.aio.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 *bitshares.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 *bitshares.market.Market.sell()* or *bitshares.market.Market.buy()*, you will sell/buy **only quote** and obtain/pay **only base**.

---

accountopenorders(*account=None*)

Returns open Orders.

Parameters **account** (bitshares.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.

---

bitshares

Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

buy(*price*, *amount*, *expiration=None*, *killfill=False*, *account=None*, *returnOrderId=False*, ***kwargs*)
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*, ***kwargs*)
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

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

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.

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

get_limit_orders(*limit=25*)
Returns the list of limit orders for a given market.

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

Sample output:

```
[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)],
```

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

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 `:`)

classmethod inject(*cls*)

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

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

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:*bitshares.price.Order* and thus carries the keys `base`, `quote` and `price`. From those you can obtain the actual amounts for sale

Note: This method does order consolidation and hides some details of individual orders!

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

sell(*price*, *amount*, *expiration=None*, *killfill=False*, *account=None*, *returnOrderId=False*, ***kwargs*)
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%.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

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

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

volume24h()
Returns the 24-hour volume for all markets, plus totals for primary currencies.

Sample output:

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

bitshares.aio.memo module

class bitshares.aio.memo.**Memo**(**args*, ***kwargs*)

Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.memo.Memo*

Deals with Memos that are attached to a transfer.

Parameters

- **from_account** (bitshares.aio.account.Account) – Account that has sent the memo
- **to_account** (bitshares.aio.account.Account) – Account that has received the memo
- **blockchain_instance** (bitshares.aio.bitshares.BitShares) – BitShares 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 bitshares.aio.memo import Memo
m = await Memo("bitshareseu", "wallet.xeroc")
m.unlock_wallet("secret")
enc = (m.encrypt("foobar"))
print(enc)
>> {'nonce': '17329630356955254641', 'message': '8563e2bb2976e0217806d642901a2855
↪'}
print(m.decrypt(enc))
>> foobar
```

To decrypt a memo, simply use

```
from bitshares.aio.memo import Memo
m = await Memo()
m.blockchain.wallet.unlock("secret")
print(memo.decrypt(op_data["memo"]))
```

if `op_data` being the payload of a transfer operation.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

decrypt(*message*)
Decrypt a message

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

Returns decrypted message

Return type str

define_classes()
Needs to define instance variables that provide classes

encrypt(*message*)
Encrypt a memo

Parameters **message** (*str*) – clear text memo message

Returns encrypted message

Return type str

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

unlock_wallet(**args*, ***kwargs*)
Unlock the library internal wallet

bitshares.aio.message module

class bitshares.aio.message.**Message**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.message.Message*

MESSAGE_SPLIT = ('-----BEGIN BITSHARES SIGNED MESSAGE-----', '-----BEGIN META-----', '

SIGNED_MESSAGE_ENCAPSULATED = '\n{MESSAGE_SPLIT[0]}\n{message}\n{MESSAGE_SPLIT[1]}\nac

SIGNED_MESSAGE_META = '{message}\naccount={meta[account]}\nmemokey={meta[memokey]}\nbl

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

define_classes()
Needs to define instance variables that provide classes

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

sign(**args*, ***kwargs*)
Sign a message with an account's memo key

Parameters **account** (*str*) – (optional) the account that owns the bet (defaults to `default_account`)

Raises **ValueError** – If not account for signing is provided

Returns the signed message encapsulated in a known format

supported_formats = (<class 'graphenecommon.aio.message.MessageV1'>, <class 'graphenec

valid_exceptions = (<class 'graphenecommon.exceptions.AccountDoesNotExistsException'>,

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

bitshares.aio.price module

class bitshares.aio.price.**FilledOrder**(*order*, ***kwargs*)
Bases: *bitshares.aio.price.Price*

This class inherits *bitshares.aio.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 **blockchain_instance** (bitshares.aio.bitshares.BitShares) – BitShares instance

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

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!

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

invert()
Invert the price (e.g. go from `USD/BTS` into `BTS/USD`)

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

json()

return { "base": self["base"].json(), "quote": self["quote"].json()

}

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

market
Open the corresponding market.

Returns Instance of *bitshares.aio.market.Market* for the corresponding pair of assets.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

symbols()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.aio.price.**Order**(**args*, ***kwargs*)
Bases: *bitshares.aio.price.Price*

This class inherits *bitshares.aio.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 **blockchain_instance** (bitshares.aio.bitshares.BitShares) – BitShares 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`.

---

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!

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

invert()
Invert the price (e.g. go from `USD/BTS` into `BTS/USD`)

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

json()

return { "base": self["base"].json(), "quote": self["quote"].json()

}

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

market
Open the corresponding market.

Returns Instance of *bitshares.aio.market.Market* for the corresponding pair of assets.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

symbols()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.aio.price.**Price**(**args*, ***kwargs*)

Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.price.Price*

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 :class:`bitshares.amount.Amount`. The
amount themselves define the price.

.. note::

    The price (floating) is derived as ``base/quote``

:param list args: Allows to deal with different representations of a price
:param bitshares.aio.asset.Asset base: Base asset
:param bitshares.aio.asset.Asset quote: Quote asset
:param bitshares.aio.bitshares.BitShares blockchain_instance: BitShares instance
:returns: All data required to represent a price
:rtype: 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 :class:`bitshares.aio.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␣
↪:class:`bitshares.aio.amount.Amount`
    * ``args`` being a list of ``[quote, base]`` both being instances of ``str``␣
↪(``amount symbol``)
    * ``base`` and ``quote`` being instances of :class:`bitshares.aio.asset.
↪Amount`

This allows instanciations like:

* ``Price("0.315 USD/BTS")``
* ``Price(0.315, base="USD", quote="BTS")``
* ``Price(0.315, base=Asset("USD"), quote=Asset("BTS"))``
* ``Price({"base": {"amount": 1, "asset_id": "1.3.0"}, "quote": {"amount": 10,
↪"asset_id": "1.3.106"}})``
* ``Price({"receives": {"amount": 1, "asset_id": "1.3.0"}, "pays": {"amount": 10,
↪"asset_id": "1.3.106"}}, base_asset=Asset("1.3.0"))``
* ``Price(quote="10 GOLD", base="1 USD")``
* ``Price("10 GOLD", "1 USD")``
* ``Price(Amount("10 GOLD"), Amount("1 USD"))``
* ``Price(1.0, "USD/GOLD")``

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

.. code-block:: python
```

(continues on next page)

(continued from previous page)

```
>>> from bitshares.aio.price import Price
>>> await Price("0.3314 USD/BTS") * 2
0.662600000 USD/BTS
```

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!

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

invert()
Invert the price (e.g. go from `USD/BTS` into `BTS/USD`)

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

json()

return { "base": self["base"].json(), "quote": self["quote"].json()

}

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

market
Open the corresponding market.

Returns Instance of *bitshares.aio.market.Market* for the corresponding pair of assets.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

symbols()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.aio.price.**PriceFeed**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.price.PriceFeed*

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 **blockchain_instance** (bitshares.aio.bitshares.BitShares) – BitShares instance

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

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

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

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.aio.price.**UpdateCallOrder**(*call*, ***kwargs*)
Bases: *bitshares.aio.price.Price*

This class inherits *bitshares.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 **blockchain_instance** (bitshares.aio.bitshares.BitShares) – BitShares instance

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!

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

invert()
Invert the price (e.g. go from `USD/BTS` into `BTS/USD`)

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

json()

return { "base": self["base"].json(), "quote": self["quote"].json()

}

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

market
Open the corresponding market.

Returns Instance of *bitshares.aio.market.Market* for the corresponding pair of assets.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

symbols()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.aio.proposal module

class bitshares.aio.proposal.**Proposal**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.proposal.Proposal*

Read data about a Proposal Balance in the chain.

Parameters

- **id** (*str*) – Id of the proposal
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

expiration

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

is_in_review

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

proposed_operations

proposer
Return the proposer of the proposal if available in the backend, else returns None

refresh()

review_period

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.aio.proposal.**Proposals**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.proposal.Proposals*

Obtain a list of pending proposals for an account.

Parameters

- **account** (*str*) – Account name
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

append()
Append object to the end of the list.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

cache(*key*)
(legacy) store the current object with key `key`.

classmethod cache_objects(*data*, *key=None*)
This classmethod allows to feed multiple objects into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear()
Remove all items from list.

classmethod clear_cache()
Clear/Reset the entire Cache

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

define_classes()
Needs to define instance variables that provide classes

extend()
Extend list by appending elements from the iterable.

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

index()
Return first index of value.

Raises ValueError if the value is not present.

classmethod inject(*cls*)

insert()
Insert object before index.

items()
This overrides items() so that refresh() is called if the object is not already fetched

pop()
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

refresh(**args*, ***kwargs*)
Interface that needs to be implemented. This method is called when an object is requested that has not yet been fetched/stored

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

sort()
Stable sort *IN PLACE*.

store(*data*, *key=None*, **args*, ***kwargs*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

bitshares.aio.transactionbuilder module

class bitshares.aio.transactionbuilder.**ProposalBuilder**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.aio.transactionbuilder.ProposalBuilder*

Proposal Builder allows us to construct an independent Proposal that may later be added to an instance ot TransactionBuilder.

Parameters

- **proposer** (*str*) – Account name of the proposing user
- **proposal_expiration** (*int*) – Number seconds until the proposal is supposed to expire
- **proposal_review** (*int*) – Number of seconds for review of the proposal
- **transactionbuilder.TransactionBuilder** – Specify your own instance of transaction builder (optional)
- **blockchain_instance** (*instance*) – Blockchain instance

appendOps(*ops*, *append_to=None*)
Append op(s) to the transaction builder

Parameters **ops** (*list*) – One or a list of operations

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

broadcast()

chain
Short form for blockchain (for the lazy)

define_classes()
Needs to define instance variables that provide classes

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

get_parent()
This allows to referr to the actual parent of the Proposal

get_raw()
Returns an instance of base "Operations" for further processing

classmethod inject(*cls*)

is_empty()

json()
Return the json formated version of this proposal

list_operations()

set_expiration(*p*)

set_parent(*p*)

set_proposer(*p*)

set_review(*p*)

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

class bitshares.aio.transactionbuilder.**TransactionBuilder**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.aio.transactionbuilder.TransactionBuilder*

This class simplifies the creation of transactions by adding operations and signers.

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!

add_required_fees(*ops*, *asset_id='1.3.0'*)
Auxiliary method to obtain the required fees for a set of operations. Requires a websocket connection to a witness node!

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(*accounts*, *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.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

broadcast()
Broadcast a transaction to the blockchain network

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

chain
Short form for blockchain (for the lazy)

clear()
Clear the transaction builder and start from scratch

constructTx()
Construct the actual transaction and store it in the class's dict store

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_block_params(*use_head_block=False*)
Auxiliary method to obtain `ref_block_num` and `ref_block_prefix`. Requires a websocket connection to a witness node!

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

get_parent()
TransactionBuilders don't have parents, they are their own parent

classmethod inject(*cls*)

is_empty()

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

json()
Show the transaction as plain json

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

list_operations()

permission_types = ['active', 'owner']

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

set_expiration(*p*)

set_fee_asset(*fee_asset*)
Set asset to fee

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

sign()
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.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

verify_authority()
Verify the authority of the signed transaction

bitshares.aio.vesting module

class bitshares.aio.vesting.**Vesting**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.vesting.Vesting*

Read data about a Vesting Balance in the chain.

Parameters

- **id** (*str*) – Id of the vesting balance
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

account

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

claim(*amount=None*)

claimable

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

refresh()

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.aio.wallet module

class bitshares.aio.wallet.**Wallet**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.aio.wallet.Wallet*

addPrivateKey(*wif*)
Add a private key to the wallet database

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

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

create(*pwd*)
Alias for newWallet()

created()
Do we have a wallet database already?

define_classes()
Needs to define instance variables that provide classes

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(*current=False*)
Return all installed public keys

Parameters **current** (*bool*) – If true, returns only keys for currently connected blockchain

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

is_encrypted()
Is the key store encrypted?

lock()
Lock the wallet database

locked()
Is the wallet database locked?

newWallet(*pwd*)
Create a new wallet database

prefix

privatekey(*key*)

publickey_from_wif(*wif*)

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

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

rpc

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

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

unlock(*pwd*)
Unlock the wallet database

unlocked()
Is the wallet database unlocked?

wipe(*sure=False*)

bitshares.aio.witness module

class bitshares.aio.witness.**Witness**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.witness.Witness*

Read data about a witness in the chain.

Parameters

- **account_name** (*str*) – Name of the witness
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

account

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

is_active

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

refresh()

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

weight

class bitshares.aio.witness.**Witnesses**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.witness.Witnesses*

Obtain a list of **active** witnesses and the current schedule.

Parameters

- **only_active** (*bool*) – (False) Only return witnesses that are actively producing blocks

- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

append()
Append object to the end of the list.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

cache(*key*)
(legacy) store the current object with key `key`.

classmethod cache_objects(*data*, *key=None*)
This classmethod allows to feed multiple objects into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear()
Remove all items from list.

classmethod clear_cache()
Clear/Reset the entire Cache

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

define_classes()
Needs to define instance variables that provide classes

extend()
Extend list by appending elements from the iterable.

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

index()
Return first index of value.

Raises ValueError if the value is not present.

classmethod inject(*cls*)

insert()
Insert object before index.

items()
This overrides items() so that refresh() is called if the object is not already fetched

pop()
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

refresh(**args*, ***kwargs*)
Interface that needs to be implemented. This method is called when an object is requested that has not yet been fetched/stored

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

sort()
Stable sort *IN PLACE*.

store(*data*, *key=None*, **args*, ***kwargs*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

bitshares.aio.worker module

class bitshares.aio.worker.**Worker**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.worker.Worker*

Read data about a worker in the chain.

Parameters

- **id** (*str*) – id of the worker
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

account

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overrides items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

post_format()

refresh()

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.aio.worker.**Workers**(**args*, ***kwargs*)
Bases: *bitshares.aio.instance.BlockchainInstance*, *bitshares.aio.worker.Workers*

Obtain a list of workers for an account.

Parameters

- **account_name/id** (*str*) – Name/id of the account (optional)
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

append()
Append object to the end of the list.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.aio.instance.BlockchainInstance*

cache(*key*)
(legacy) store the current object with key `key`.

classmethod cache_objects(*data*, *key=None*)
This classmethod allows to feed multiple objects into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear()
Remove all items from list.

classmethod clear_cache()
Clear/Reset the entire Cache

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

define_classes()
Needs to define instance variables that provide classes

extend()
Extend list by appending elements from the iterable.

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

index()
Return first index of value.

Raises ValueError if the value is not present.

classmethod inject(*cls*)

insert()
Insert object before index.

items()
This overrides items() so that refresh() is called if the object is not already fetched

pop()
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

refresh(**args*, ***kwargs*)
Interface that needs to be implemented. This method is called when an object is requested that has not yet been fetched/stored

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

sort()
Stable sort *IN PLACE*.

store(*data*, *key=None*, **args*, ***kwargs*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

Module contents

Submodules

bitshares.account module

class bitshares.account.**Account**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.account.Account*

This class allows to easily access Account data.

Parameters

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

Returns Account data

Return type dictionary

Raises **bitshares.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 bitshares.account import Account
account = Account("init0")
print(account)
```

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()`.

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

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

bitshares
Alias for the specific blockchain.

blacklist (*account*)
Add an other account to the blacklist of this account

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object (*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

call_positions
Alias for :func:bitshares.account.Account.callpositions.

callpositions
List call positions (collateralized positions *Market Pegged Assets*)

chain
Short form for blockchain (for the lazy)

clear () → None. Remove all items from D.

classmethod clear_cache ()
Clear/Reset the entire Cache

copy () → a shallow copy of D

define_classes ()
Needs to define instance variables that provide classes

ensure_full ()

fromkeys ()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class ()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache (*id*)
Get an element from the cache explicitly

history (*first=0*, *last=0*, *limit=-1*, *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

- **first** (*int*) – sequence number of the first transaction to return (*optional*)
- **last** (*int*) – sequence number of the last transaction to return (*optional*)
- **limit** (*int*) – limit number of transactions to return (*optional*)
- **only_ops** (*array*) – Limit generator by these operations (*optional*)
- **exclude_ops** (*array*) – Exclude these operations from generator (*optional*).

... note:: only_ops and exclude_ops takes an array of strings: The full list of operation ID's can be found in operationids.py. Example: ['transfer', 'fill_order']

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

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

is_ltm
Is the account a lifetime member (LTM)?

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

name

nolist(*account*)
Remove an other account from any list of this account

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

openorders
Returns open Orders.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

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

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

upgrade()
Upgrade account to life time member

values() → an object providing a view on D's values

whitelist(*account*)
Add an other account to the whitelist of this account

class bitshares.account.**AccountUpdate**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.account.AccountUpdate*

This purpose of this class is to keep track of account updates as they are pushed through by *bitshares.notify.Notify*.

Instances of this class are dictionaries and take the following form:

account
In oder to obtain the actual `account.Account` from this class, you can use the `account` attribute.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

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

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

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.amount module

class bitshares.amount.**Amount**(**args*, ***kwargs*)

Bases: *bitshares.instance.BlockchainInstance*, *bitshares.amount.Amount*

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)
- **blockchain_instance** (bitshares.bitshares.BitShares) – BitShares instance

Returns All data required to represent an Amount/Asset

Return type dict

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

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

Way to obtain a proper instance:

- `args` can be a string, e.g.: "1 USD"
- `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 *bitshares.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 *bitshares.asset.Asset*)

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

```
Amount("1 USD") * 2
Amount("15 GOLD") + Amount("0.5 GOLD")
```

amount
Returns the amount as float

asset
Returns the asset as instance of *asset.Asset*

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

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

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

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

json()

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

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

symbol
Returns the symbol of the asset

tuple()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.asset module

class bitshares.asset.**Asset**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.asset.Asset*

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 data
- **blockchain_instance** (bitshares.bitshares.BitShares) – BitShares 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()`.

---

add_authorities(*type*, *authorities=None*)
Add authorities to an assets white/black list.

Parameters

- **type** (*str*) – `blacklist` or `whitelist`
- **authorities** (*list*) – List of authorities (Accounts)

add_markets(*type*, *authorities=None*, *force_enable=True*)
Add markets to an assets white/black list.

Parameters

- **type** (*str*) – `blacklist` or `whitelist`
- **markets** (*list*) – List of markets (assets)
- **force_enable** (*bool*) – Force enable `white_list` flag

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

calls

chain
Short form for blockchain (for the lazy)

change_issuer(*new_issuer*, ***kwargs*)
Change asset issuer (needs signing with owner key!)

Parameters **new_issuer** (*str*) – account name

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

disableflag(*flag*)
Enable a certain flag.

Parameters **flag** (*str*) – Flag name

enableflag(*flag*)
Enable a certain flag.

Parameters **flag** (*str*) – Flag name

ensure_full()

feed

feeds

flags
List the permissions that are currently used (flags)

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_call_orders(*limit=100*)

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

get_settle_orders(*limit=100*)

getfromcache(*id*)
Get an element from the cache explicitly

halt()
Halt this asset from being moved or traded.

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

is_bitasset
Is the asset a market pegged asset?

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

issue(*amount*, *to*, *memo=None*, ***kwargs*)
Issue new shares of an asset.

Parameters

- **amount** (*float*) – Amount to issue
- **to** (*str*) – Recipient
- **memo** (*str*) – (optional) Memo message

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

market_fee_percent

max_market_fee

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

permissions
List the permissions for this asset that the issuer can obtain

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

precision

refresh()
Refresh the data from the API server

release(*whitelist_authorities=None*, *blacklist_authorities=None*, *whitelist_markets=None*, *blacklist_markets=None*)
Release this asset and allow unrestricted transfer, trading, etc.

Parameters

- **whitelist_authorities** (*list*) – List of accounts that serve as whitelist authorities
- **blacklist_authorities** (*list*) – List of accounts that serve as blacklist authorities

- **whitelist_markets** (*list*) – List of assets to allow trading with
- **blacklist_markets** (*list*) – List of assets to prevent trading with

remove_authorities (*type*, *authorities=None*)
Remove authorities from an assets white/black list.

Parameters

- **type** (*str*) – `blacklist` or `whitelist`
- **authorities** (*list*) – List of authorities (Accounts)

remove_markets (*type*, *authorities=None*)
Remove markets from an assets white/black list.

Parameters

- **type** (*str*) – `blacklist` or `whitelist`
- **markets** (*list*) – List of markets (assets)

seize (*from_account*, *to_account*, *amount*, ***kwargs*)
Seize amount from an account and send to another.

... note:: This requires the override_authority to be set for this asset!

Parameters

- **from_account** (`bitshares.account.Account`) – From this account
- **to_account** (`bitshares.account.Account`) – To this account
- **amount** (`bitshares.amount.Amount`) – Amount to seize

set_market_fee (*percentage_fee*, *max_market_fee*, ***kwargs*)
Set trading percentage fee.

Parameters

- **percentage_fee** (*float*) – Percentage of fee
- **max_market_fee** (`bitshares.amount.Amount`) – Max Fee

classmethod set_shared_blockchain_instance (*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance ()
This method allows to set the current instance as default

setdefault ()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

setoptions (*flags*, ***kwargs*)
Enable a certain flag.

Flags:

- charge_market_fee

- white_list
- override_authority
- transfer_restricted
- disable_force_settle
- global_settle
- disable_confidential
- witness_fed_asset
- committee_fed_asset

Parameters **flag** (*dict*) – dictionary of flags and boolean

settlements

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

symbol

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

update_cer(*cer*, *account=None*, ***kwargs*)
Update the Core Exchange Rate (CER) of an asset

update_feed_producers(*producers*)
Update bitasset feed producers.

Parameters **producers** (*list*) – List of accounts that are allowed to produce a feed

values() → an object providing a view on D's values

bitshares.bitshares module

class bitshares.bitshares.**BitShares**(*node=''*, *rpcuser=''*, *rpcpassword=''*, *debug=False*, *skip_wallet_init=False*, ***kwargs*)
Bases: `graphenecommon.chain.AbstractGrapheneChain`

Connect to the BitShares 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 bitshareslibs load the keys from the locally stored wallet SQLite database (see `storage.py`). To use this mode, simply call `BitShares()` 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 `BitShares()`.
- **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 BitShares.

The idea is to have a class that allows to do this:

```
from bitshares import BitShares
bitshares = BitShares()
print(bitshares.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.

account_whitelist(*account_to_whitelist*, *lists=None*, *account=None*, ***kwargs*)
Account whitelisting.

Parameters

- **account_to_whitelist** (*str*) – The account we want to add to either the white- or the blacklist
- **lists** (*set*) – (defaults to `('white')`). Lists the user should be added to. Either empty set, 'black', 'white' or both.
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

allow(*foreign*, *weight=None*, *permission='active'*, *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

approvecommittee(*committees*, *account=None*, ***kwargs*)
Approve a committee.

Parameters

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

approveproposal(*proposal_ids*, *account=None*, *approver=None*, ***kwargs*)
Approve Proposal.

Parameters

- **proposal_id** (*list*) – Ids of the proposals
- **appprover** (*str*) – The account or key to use for approval (defaults to `account`)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

approvewitness(*witnesses*, *account=None*, ***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`)

approveworker(*workers*, *account=None*, ***kwargs*)
Approve a worker.

Parameters

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

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

bid_collateral(*additional_collateral*, *debt_covered*, *account=None*, ***kwargs*)

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

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

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

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

clear()

clear_cache()
Clear Caches

connect(*node=''*, *rpcuser=''*, *rpcpassword=''*, ***kwargs*)
Connect to blockchain network (internal use only)

create_account(*account_name*, *registrar=None*, *referrer='1.2.35641'*, *referrer_percent=50*, *owner_key=None*, *active_key=None*, *memo_key=None*, *owner_account=None*, *active_account=None*, *password=None*, *additional_owner_keys=None*, *additional_active_keys=None*, *additional_owner_accounts=None*, *additional_active_accounts=None*, *proxy_account='proxy-to-self'*, *storekeys=True*, ***kwargs*)
Create new account on BitShares.

The brainkey/password can be used to recover all generated keys (see *bitsharesbase.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

create_asset(*symbol*, *precision*, *max_supply*, *description=''*, *is_bitasset=False*, *is_prediction_market=False*, *market_fee_percent=0*, *max_market_fee=None*, *permissions=None*, *flags=None*, *whitelist_authorities=None*, *blacklist_authorities=None*, *whitelist_markets=None*, *blacklist_markets=None*, *bitasset_options=None*, *account=None*, ***kwargs*)

Create a new asset.

Parameters

- **symbol** (*str*) – Asset symbol
- **precision** (*int*) – Asset precision
- **max_supply** (*int*) – Asset max supply
- **description** (*str*) – (optional) Asset description
- **is_bitasset** (*bool*) – (optional) True = bitasset, False = UIA (default: False)
- **is_prediction_market** (*bool*) – (optional) True: PD, False = plain smartcoin (default: False)
- **market_fee_percent** (*float*) – (optional) Charge market fee (0-100) (default: 0)
- **max_market_fee** (*float*) – (optional) Absolute amount of max market fee, value of this option should be a whole number (default: same as max_supply)
- **permissions** (*dict*) – (optional) Asset permissions
- **flags** (*dict*) – (optional) Enabled asset flags
- **whitelist_authorities** (*list*) – (optional) List of accounts that serve as whitelist authorities
- **blacklist_authorities** (*list*) – (optional) List of accounts that serve as blacklist authorities

- **whitelist_markets** (*list*) – (optional) List of assets to allow trading with
- **blacklist_markets** (*list*) – (optional) List of assets to prevent trading with
- **bitasset_options** (*dict*) – (optional) Bitasset settings
- **account** (*str*) – (optional) the issuer account to (defaults to `default_account`)

create_committee_member(*url=''*, *account=None*, ***kwargs*)
Create a committee member.

Parameters

- **url** (*str*) – URL to read more about the worker
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

create_liquidity_pool(*asset_a*, *asset_b*, *share_asset*, *taker_fee_percent*, *withdrawal_fee_percent*, *account=None*, ***kwargs*)
Create a liquidity pool

Parameters

- **asset_a** (*str*) – First asset in the pool pair.
- **asset_b** (*str*) – Second asset in the pool pair.
- **share_asset** (*str*) – The asset which represents shares in the pool.

For asset parameters, these can be either symbols or asset_id strings. Note that network expects asset_a to have a lower-numbered asset_id than asset_b.

Parameters

- **taker_fee_percent** (*float*) – The pool's taker fee percentage.
- **withdrawal_fee_percent** (*float*) – The pool's withdrawal fee percent.

For percentages, meaningful range is [0.00, 100.00], where 1% is represented as 1.0. Smallest non-zero value recognized by BitShares chain is 0.01 for 0.01%.

create_voting_ticket(*target_type*, *amount_to_lock*, *account=None*, ***kwargs*)
Create a voting ticket

Parameters

- **target_type** (*int,str*) – Lock period target. Should be a string from operations.ticket_type_strings or the index of the intended string.
- **amount_to_lock** (`Amount`) – Amount to lock up for the duration selected in target_type.

create_worker(*name*, *daily_pay*, *end*, *url=''*, *begin=None*, *payment_type='vesting'*, *pay_vesting_period_days=0*, *account=None*, ***kwargs*)
Create a worker.

This removes the shares from the supply

Required

Parameters

- **name** (*str*) – Name of the worke
- **daily_pay** (`bitshares.amount.Amount`) – The amount to be paid daily
- **end** (*datetime*) – Date/time of end of the worker

Optional

Parameters

- **url** (*str*) – URL to read more about the worker
- **begin** (*datetime*) – Date/time of begin of the worker
- **payment_type** (*string*) – ["burn", "refund", "vesting"] (default: "vesting")
- **pay_vesting_period_days** (*int*) – Days of vesting (default: 0)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

define_classes()

delete_liquidity_pool(*pool*, *account=None*, ***kwargs*)
Delete a liquidity pool

Parameters **pool** (*str*, `Asset`) – The liquidity pool to delete. Can be the pool id as a string, or can be an Asset, asset_id, or symbol of the share asset for the pool.

deposit_into_liquidity_pool(*pool*, *amount_a*, *amount_b*, *account=None*, ***kwargs*)
Deposit assets into a liquidity pool

Parameters

- **pool** (*str*, `Asset`) – The liquidity pool to use. Can be the pool id as a string, or can be an Asset, asset_id, or symbol of the share asset for the pool.
- **amount_a** (`Amount`) –
- **amount_b** (`Amount`) –

disallow(*foreign*, *permission='active'*, *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

disapprovecommittee(*committees*, *account=None*, ***kwargs*)
Disapprove a committee.

Parameters

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

disapproveproposal(*proposal_ids*, *account=None*, *approver=None*, ***kwargs*)
Disapprove Proposal.

Parameters

- **proposal_ids** (*list*) – Ids of the proposals

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

disapprovewitness(*witnesses*, *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`)

disapproveworker(*workers*, *account=None*, ***kwargs*)
Disapprove a worker.

Parameters

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

exchange_with_liquidity_pool(*pool*, *amount_to_sell*, *min_to_receive*, *account=None*, ***kwargs*)
Exchange assets against a liquidity pool

Parameters

- **pool** (*str,*Asset) – The liquidity pool to use. Can be the pool id as a string, or can be an Asset, asset_id, or symbol of the share asset for the pool.
- **amount_to_sell** (Amount) –
- **min_to_receive** (Amount) –

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 *new_proposal()*) or TransactionBuilder (see *new_tx()*) to specify where to put a specific operation.

... note:: `append_to` is exposed to every method used in the this 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 `txbuffer` as instance of `transactionbuilder.TransactionBuilder`. You may want to use your own txbuffer

fund_fee_pool(*symbol*, *amount*, *account=None*, ***kwargs*)
Fund the fee pool of an asset.

Parameters

- **symbol** (*str*) – The symbol to fund the fee pool of
- **amount** (*float*) – The amount to be burned.
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

htlc_create(*amount*, *to*, **args*, *hash_type=None*, *hash_hex=None*, *expiration=3600*, *preimage=None*, *preimage_length=0*, *account=None*, ***kwargs*)
Create an HTLC contract.

Parameters

- **amount** (Amount) – Amount to lock
- **to** (*str*) – Recipient
- **expiration** (*int*) – Contract duration in seconds
- **hash_hex** (*str*) – Hash as string of hex digits
- **preimage** (*str*) – Preimage as ascii string. Note hex digits would be interpretted as ascii text, not as bytes. Not generally recommended to use this option. Options hash_hex and preimage are mutually exclusive.
- **preimage_length** (*int*) – If non-zero, htlc contract will require preimage of exact length. Generally OK to leave this as zero. Note if preimage param is provided, this value SHOULD be either zero or match exactly the length of the preimage, else an irredeemable htlc will be created. Optionally, a sentinal value of -1 can be used to compute length automatically from the preimage param.

htlc_redeem(*htlc_id*, *preimage*, *encoding='utf-8'*, *account=None*, ***kwargs*)
Redeem an htlc contract

Parameters

- **preimage** (*str*) – The preimage that unlocks the htlc
- **encoding** (*str*) – "utf-8", . . . , or "hex"

info()
Returns the global properties

is_connected()

newWallet(*pwd*)

new_proposal(*parent=None*, *proposer=None*, *proposal_expiration=None*, *proposal_review=None*, ***kwargs*)

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

Returns int txid id of the new txbuffer

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

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

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

prefix
Contains the prefix of the blockchain

propbuffer
Return the default proposal buffer

proposal(*proposer=None*, *proposal_expiration=None*, *proposal_review=None*)
Return the default proposal buffer

... note:: If any parameter is set, the default proposal parameters will be changed!

publish_price_feed(*symbol*, *settlement_price*, *cer=None*, *mssr=110*, *mcr=200*, *account=None*)
Publish a price feed for a market-pegged asset.

Parameters

- **symbol** (*str*) – Symbol of the asset to publish feed for
- **settlement_price** (bitshares.price.Price) – Price for settlement
- **cer** (bitshares.price.Price) – Core exchange Rate (default `settlement_price + 5%`)
- **mssr** (*float*) – Percentage for max short squeeze ratio (default: 110%)
- **mcr** (*float*) – Percentage for maintenance collateral ratio (default: 200%)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

Note: The `account` needs to be allowed to produce a price feed for `symbol`. For witness produced feeds this means `account` is a witness account!

---

reserve(*amount*, *account=None*, ***kwargs*)
Reserve/Burn an amount of this shares.

This removes the shares from the supply

Parameters

- **amount** (bitshares.amount.Amount) – The amount to be burned.
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

set_blocking(*block=True*)
This sets a flag that forces the broadcast to block until the transactions made it into a block

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

set_proxy(*proxy_account*, *account=None*, ***kwargs*)
Set a specific proxy for account.

Parameters

- **proxy_account** (bitshares.account.Account) – Account to be proxied
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

set_shared_instance()
This method allows to set the current instance as default

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

unset_proxy (*account=None*, ***kwargs*)
Unset the proxy account to start voting yourself.

update_cer (*symbol*, *cer*, *account=None*)
Update the Core Exchange Rate (CER) of an asset.

Parameters

- **symbol** (*str*) – Symbol of the asset to publish feed for
- **cer** (`bitshares.price.Price`) – Core exchange Rate
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

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`)

update_voting_ticket (*ticket_id*, *new_target_type*, *amount_to_update*, *account=None*, ***kwargs*)
Update a voting ticket

Parameters

- **ticket_id** (*str*) – Id (e.g. "1.18.xxx") of the ticket to update.
- **target_type** (*int, str*) – New lock period target. Should be a string from operations.ticket_type_strings or the index of the intended string.
- **amount_to_update** (Amount, *None*) – Amount to move over to the new lock-up target. (Optional - absence implies update whole amount.)

update_witness(*witness_identifier*, *url=None*, *key=None*, ***kwargs*)
Upgrade a witness account.

Parameters

- **witness_identifier** (*str*) – Identifier for the witness
- **url** (*str*) – New URL for the witness
- **key** (*str*) – Public Key for the signing

upgrade_account(*account=None*, ***kwargs*)
Upgrade an account to Lifetime membership.

Parameters **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

vesting_balance_withdraw(*vesting_id*, *amount=None*, *account=None*, ***kwargs*)
Withdraw vesting balance.

Parameters

- **vesting_id** (*str*) – Id of the vesting object
- **Amount** (bitshares.amount.Amount) – to withdraw ("all" if not provided")
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

withdraw_from_liquidity_pool(*pool*, *share_amount*, *account=None*, ***kwargs*)
Withdraw stake from a liquidity pool

Parameters

- **pool** (*str*, Asset) – The liquidity pool to use. Can be the pool id as a string, or can be an Asset, asset_id, or symbol of the share asset for the pool.
- **share_amount** (Amount) – Amount of share asset to redeem. Must be a quantity of the pool's share_asset.

bitshares.block module

class bitshares.block.**Block**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.block.Block*

Read a single block from the chain.

Parameters

- **block** (*int*) – block number
- **blockchain_instance** (bitshares.bitshares.BitShares) – BitShares 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 bitshares.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()`.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

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

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

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

type_id = 'n/a'

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.block.**BlockHeader**(**args*, ***kwargs*)

Bases: *bitshares.instance.BlockchainInstance*, *bitshares.block.BlockHeader*

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

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

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

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

type_id = 'n/a'

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.blockchain module

class bitshares.blockchain.**Blockchain**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.blockchain.Blockchain*

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

Parameters

- **blockchain_instance** (bitshares.bitshares.BitShares) – BitShares instance

- **mode** (*str*) – (default) Irreversible block (`irreversible`) or actual head block (`head`)
- **max_block_wait_repetition** (*int*) – (default) 3 maximum wait time for next block ismax_block_wait_repetition * block_interval

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

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:.*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.

---

bitshares
Alias for the specific blockchain.

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

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

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)

chain
Short form for blockchain (for the lazy)

chainParameters()
The blockchain parameters, such as fees, and committee-controlled parameters are returned here

config()
Returns object 2.0.0

define_classes()
Needs to define instance variables that provide classes

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_block_interval()
This call returns the block interval

get_chain_properties()
Return chain properties

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_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

get_network()
Identify the network

Returns Network parameters

Return type dict

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

classmethod inject(*cls*)

is_irreversible_mode()

ops(*start=None*, *stop=None*, ***kwargs*)
Yields all operations (excluding 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!

participation_rate

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

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 * "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.

update_chain_parameters()

wait_for_and_get_block(*block_number*, *blocks_waiting_for=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*) – (default) difference between block_number and current head how many blocks we are willing to wait, positive int

bitshares.blockchainobject module

class bitshares.blockchainobject.**BlockchainObject**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.blockchainobject.BlockchainObject*

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.blockchainobject.**Object**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.blockchainobject.Object*

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = False

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

refresh()
This is the refresh method that overloads the prototype in BlockchainObject.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.committee module

class bitshares.committee.**Committee**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.committee.Committee*

Read data about a Committee Member in the chain.

Parameters

- **member** (*str*) – Name of the Committee Member
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC
- **lazy** (*bool*) – Use lazy loading

account

account_id

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

refresh()

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.dex module

class bitshares.dex.**Dex**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*

This class simplifies interactions with the decentralized exchange.

Parameters **blockchain_instance** (bitshares.bitshares.BitShares) – BitShares instance

---

Note: The methods of this class only deal with a single asset (at most). If you are looking to deal with orders for trading, please use *bitshares.market.Market*.

---

adjust_collateral_ratio(*symbol*, *new_collateral_ratio*, *account=None*, *target_collateral_ratio=None*)
Adjust the collataral ratio of a debt position.

Parameters

- **symbol** (*str*) – Symbol to adjust collateral for
- **new_collateral_ratio** (*float*) – desired collateral ratio
- **target_collateral_ratio** (*float*) – Tag the call order so that in case of margin call, only enough debt is covered to get back to this ratio

Raises

- **ValueError** – if symbol is not a bitasset
- **ValueError** – if collateral ratio is smaller than maintenance collateral ratio
- **ValueError** – if required amounts of collateral are not available

adjust_debt(*delta*, *new_collateral_ratio=None*, *account=None*, *target_collateral_ratio=None*)
Adjust the amount of debt for an asset.

Parameters

- **delta** (Amount) – Delta amount of the debt (-10 means reduce debt by 10, +10 means borrow another 10)
- **new_collateral_ratio** (*float*) – collateral ratio to maintain (optional, by default tries to maintain old ratio)
- **target_collateral_ratio** (*float*) – Tag the call order so that in case of margin call, only enough debt is covered to get back to this ratio

Raises

- **ValueError** – if symbol is not a bitasset
- **ValueError** – if collateral ratio is smaller than maintenance collateral ratio
- **ValueError** – if required amounts of collateral are not available

bitshares
Alias for the specific blockchain.

blockchain

borrow(*amount*, *collateral_ratio=None*, *account=None*, *target_collateral_ratio=None*)
Borrow bitassets/smartcoins from the network by putting up collateral in a CFD at a given collateral ratio.

Parameters

- **amount** (Amount) – Amount to borrow (denoted in 'asset')
- **collateral_ratio** (*float*) – Collateral ratio to borrow at
- **target_collateral_ratio** (*float*) – Tag the call order so that in case of margin call, only enough debt is covered to get back to this ratio

Raises

- **ValueError** – if symbol is not a bitasset
- **ValueError** – if collateral ratio is smaller than maintenance collateral ratio
- **ValueError** – if required amounts of collateral are not available

chain
Short form for blockchain (for the lazy)

close_debt_position(*symbol*, *account=None*)
Close a debt position and reclaim the collateral.

Parameters **symbol** (*str*) – Symbol to close debt position for

Raises **ValueError** – if symbol has no open call position

define_classes()
Needs to define instance variables that provide classes

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

list_debt_positions(*account=None*)
List Call Positions (borrowed assets and amounts)

Returns Struct of assets with amounts and call price

Return type dict

Example:

returnFees()
Returns a dictionary of all fees that apply through the network.

Example output:

```
{'proposal_create': {'fee': 400000.0},
'asset_publish_feed': {'fee': 1000.0}, 'account_create':
{'basic_fee': 950000.0, 'price_per_kbyte': 20000.0,
'premium_fee': 40000000.0}, 'custom': {'fee': 20000.0},
'asset_fund_fee_pool': {'fee': 20000.0},
'override_transfer': {'fee': 400000.0}, 'fill_order':
{}, 'asset_update': {'price_per_kbyte': 20000.0, 'fee':
200000.0}, 'asset_update_feed_producers': {'fee':
10000000.0}, 'assert': {'fee': 20000.0},
'committee_member_create': {'fee': 100000000.0}}
```

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

bitshares.exceptions module

exception bitshares.exceptions.**AccountExistsException**
Bases: `Exception`

The requested account already exists.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception bitshares.exceptions.**HtlcDoesNotExistException**
Bases: `Exception`

HTLC object does not exist.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception bitshares.exceptions.**ObjectNotInProposalBuffer**
Bases: `Exception`

Object was not found in proposal.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception bitshares.exceptions.**RPCConnectionRequired**
Bases: `Exception`

An RPC connection is required.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

bitshares.genesisbalance module

class bitshares.genesisbalance.**GenesisBalance**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.genesisbalance.GenesisBalance*

Read data about a Genesis Balances from the chain.

Parameters

- **identifier** (*str*) – identifier of the balance
- **blockchain_instance** (*bitshares*) – bitshares() instance to use when accesing a RPC

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

claim(*account=None*, ***kwargs*)
Claim a balance from the genesis block

Parameters

- **balance_id** (*str*) – The identifier that identifies the balance to claim (1.15.x)
- **account** (*str*) – (optional) the account that owns the bet (defaults to `default_account`)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

refresh()

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = 15

type_ids = []

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.genesisbalance.**GenesisBalances**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.genesisbalance.GenesisBalances*

List genesis balances that can be claimed from the keys in the wallet.

append()
Append object to the end of the list.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

define_classes()
Needs to define instance variables that provide classes

extend()
Extend list by appending elements from the iterable.

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

index()
Return first index of value.

Raises ValueError if the value is not present.

classmethod inject(*cls*)

insert()
Insert object before index.

pop()
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

sort()
Stable sort *IN PLACE*.

bitshares.htlc module

class bitshares.htlc.**Htlc**(**args*, ***kwargs*)
Bases: *bitshares.blockchainobject.BlockchainObject*

Read data about an HTLC contract on the chain.

Parameters

- **id** (*str*) – id of the HTLC
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

refresh()

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = 16

type_ids = []

update(⟦*E*⟧, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.instance module

class bitshares.instance.**BlockchainInstance**(**args*, ***kwargs*)
Bases: graphenecommon.instance.AbstractBlockchainInstanceProvider

This is a class that allows compatibility with previous naming conventions.

bitshares
Alias for the specific blockchain.

blockchain

chain
Short form for blockchain (for the lazy)

define_classes()
Needs to define instance variables that provide classes

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of SharedInstance.instance.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

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

This class merely offers a singelton for the Blockchain Instance.

config = {}

instance = None

bitshares.instance.**set_shared_bitshares_instance**(*instance*)

bitshares.instance.**set_shared_blockchain_instance**(*instance*)

bitshares.instance.**set_shared_config**(*config*)

bitshares.instance.**shared_bitshares_instance**()

bitshares.instance.**shared_blockchain_instance**()

bitshares.market module

class bitshares.market.**Market**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.market.Market*

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

Parameters

- **blockchain_instance** (bitshares.bitshares.BitShares) – BitShares instance
- **base** (bitshares.asset.Asset) – Base asset
- **quote** (bitshares.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 *bitshares.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 *bitshares.market.Market.sell()* or *bitshares.market.Market.buy()*, you will sell/buy **only quote** and obtain/pay **only base**.

---

accountopenorders(*account=None*)
Returns open Orders.

Parameters **account** (bitshares.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.

---

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

buy (*price*, *amount*, *expiration=None*, *killfill=False*, *account=None*, *returnOrderId=False*, ***kwargs*)
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*, ***kwargs*)
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`

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

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.

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

get_limit_orders (*limit=25*)
Returns the list of limit orders for a given market.

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

Sample output:

```
[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),
[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:*bitshares.price.Order* and thus carries the keys `base`, `quote` and `price`. From those you can obtain the actual amounts for sale

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 `:`)

classmethod inject(*cls*)

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

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

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:*bitshares.price.Order* and thus carries the keys `base`, `quote` and `price`. From those you can obtain the actual amounts for sale

Note: This method does order consolidation and hides some details of individual orders!

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

sell(*price*, *amount*, *expiration=None*, *killfill=False*, *account=None*, *returnOrderId=False*, ***kwargs*)
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%.

---

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

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,
```

(continues on next page)

(continued from previous page)

```
        "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

update ([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values () → an object providing a view on D's values

volume24h ()
Returns the 24-hour volume for all markets, plus totals for primary currencies.

Sample output:

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

bitshares.memo module

class bitshares.memo.**Memo** (**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.memo.Memo*

Deals with Memos that are attached to a transfer.

Parameters

- **from_account** (bitshares.account.Account) – Account that has sent the memo
- **to_account** (bitshares.account.Account) – Account that has received the memo
- **blockchain_instance** (bitshares.bitshares.BitShares) – BitShares 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 bitshares.memo import Memo
m = Memo("bitshareseu", "wallet.xeroc")
m.unlock_wallet("secret")
```

(continues on next page)

(continued from previous page)

```
enc = (m.encrypt("foobar"))
print(enc)
>> {'nonce': '17329630356955254641', 'message': '8563e2bb2976e0217806d642901a2855
↪'}
print(m.decrypt(enc))
>> foobar
```

To decrypt a memo, simply use

```
from bitshares.memo import Memo
m = Memo()
m.blockchain.wallet.unlock("secret")
print(memo.decrypt(op_data["memo"]))
```

if `op_data` being the payload of a transfer operation.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

decrypt(*message*)
Decrypt a message

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

Returns decrypted message

Return type str

define_classes()
Needs to define instance variables that provide classes

encrypt(*message*)
Encrypt a memo

Parameters **message** (*str*) – clear text memo message

Returns encrypted message

Return type str

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

unlock_wallet(**args*, ***kwargs*)
Unlock the library internal wallet

bitshares.message module

class bitshares.message.**Message**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.message.Message*

MESSAGE_SPLIT = ('-----BEGIN BITSHARES SIGNED MESSAGE-----', '-----BEGIN META-----', '

SIGNED_MESSAGE_ENCAPSULATED = '\n{MESSAGE_SPLIT[0]}\n{message}\n{MESSAGE_SPLIT[1]}\nac

SIGNED_MESSAGE_META = '{message}\naccount={meta[account]}\nmemokey={meta[memokey]}\nbl

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

define_classes()
Needs to define instance variables that provide classes

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

sign(**args*, ***kwargs*)
Sign a message with an account's memo key

Parameters **account** (*str*) – (optional) the account that owns the bet (defaults to `default_account`)

Raises **ValueError** – If not account for signing is provided

Returns the signed message encapsulated in a known format

supported_formats = (<class 'grapheneсommon.message.MessageV1'>, <class 'grapheneсommo

valid_exceptions = (<class 'grapheneсommon.exceptions.AccountDoesNotExistsException'>,

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

bitshares.notify module

class bitshares.notify.**Notify**(*accounts=None*, *markets=None*, *objects=None*, *on_tx=None*, *on_object=None*, *on_block=None*, *on_account=None*, *on_market=None*, *keep_alive=25*, ***kwargs*)

Bases: `events.events.Events`, *bitshares.instance.BlockchainInstance*

Notifications on Blockchain events.

Parameters

- **accounts** (*list*) – Account names/ids to be notified about when changing
- **markets** (*list*) – Instances of *bitshares.market.Market* that identify markets to be monitored
- **objects** (*list*) – Object ids to be notified about when changed
- **on_tx** (*fnt*) – Callback that will be called for each transaction received
- **on_block** (*fnt*) – Callback that will be called for each block received
- **on_account** (*fnt*) – Callback that will be called for changes of the listed accounts
- **on_market** (*fnt*) – Callback that will be called for changes of the listed markets
- **blockchain_instance** (bitshares.bitshares.BitShares) – BitShares instance

Example

```
from pprint import pprint
from bitshares.notify import Notify
from bitshares.market import Market

notify = Notify(
    markets=["TEST:GOLD"],
    accounts=["xeroc"],
    on_market=print,
    on_account=print,
    on_block=print,
    on_tx=print
)
notify.listen()
```

bitshares

Alias for the specific blockchain.

blockchain

chain
Short form for blockchain (for the lazy)

close()
Cleanly close the Notify instance.

define_classes()
Needs to define instance variables that provide classes

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

get_market_ids(*markets*)

classmethod inject(*cls*)

listen()
This call initiates the listening/notification process.

It behaves similar to `run_forever()`.

process_account(*message*)
This is used for processing of account Updates.

It will return instances of :class:bitshares.account.AccountUpdate‘

process_market(*data*)
This method is used for post processing of market notifications. It will return instances of either.

- *bitshares.price.Order* or
- *bitshares.price.FilledOrder* or
- *bitshares.price.UpdateCallOrder*

Also possible are limit order updates (margin calls)

reset_subscriptions(*accounts=None*, *markets=None*, *objects=None*)
Change the subscriptions of a running Notify instance.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

bitshares.price module

class bitshares.price.**FilledOrder**(*order*, ***kwargs*)
Bases: *bitshares.price.Price*

This class inherits *bitshares.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 **blockchain_instance** (bitshares.bitshares.BitShares) – BitShares instance

---

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

---

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!

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

invert()
Invert the price (e.g. go from `USD/BTS` into `BTS/USD`)

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

json()

return { "base": self["base"].json(), "quote": self["quote"].json()

}

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

market
Open the corresponding market.

Returns Instance of *bitshares.market.Market* for the corresponding pair of assets.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

symbols()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.price.**Order**(**args*, ***kwargs*)
Bases: *bitshares.price.Price*

This class inherits *bitshares.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 **blockchain_instance** (bitshares.bitshares.BitShares) – BitShares 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`.

---

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!

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

for_sale

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

invert()
Invert the price (e.g. go from `USD/BTS` into `BTS/USD`)

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

json()

return { "base": self["base"].json(), "quote": self["quote"].json()

}

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

market
Open the corresponding market.

Returns Instance of *bitshares.market.Market* for the corresponding pair of assets.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

price

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

symbols()

to_buy

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.price.**Price**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.price.Price*

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 :class:`bitshares.amount.Amount`. The
amount themselves define the price.

.. note::

    The price (floating) is derived as ``base/quote``

:param list args: Allows to deal with different representations of a price
:param bitshares.asset.Asset base: Base asset
:param bitshares.asset.Asset quote: Quote asset
:param bitshares.bitshares.BitShares blockchain_instance: BitShares instance
:returns: All data required to represent a price
:rtype: 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 :class:`bitshares.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␣
↪:class:`bitshares.amount.Amount`
    * ``args`` being a list of ``[quote, base]`` both being instances of ``str``␣
↪(``amount symbol``)
    * ``base`` and ``quote`` being instances of :class:`bitshares.asset.Amount`

This allows instanciations like:

* ``Price("0.315 USD/BTS")``
```

(continues on next page)

(continued from previous page)

```
* ``Price(0.315, base="USD", quote="BTS")``
* ``Price(0.315, base=Asset("USD"), quote=Asset("BTS"))``
* ``Price({"base": {"amount": 1, "asset_id": "1.3.0"}, "quote": {"amount": 10,
↪"asset_id": "1.3.106"}})``
* ``Price({"receives": {"amount": 1, "asset_id": "1.3.0"}, "pays": {"amount": 10,
↪"asset_id": "1.3.106"}}, base_asset=Asset("1.3.0"))``
* ``Price(quote="10 GOLD", base="1 USD")``
* ``Price("10 GOLD", "1 USD")``
* ``Price(Amount("10 GOLD"), Amount("1 USD"))``
* ``Price(1.0, "USD/GOLD")``

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

.. code-block:: python

    >>> from bitshares.price import Price
    >>> Price("0.3314 USD/BTS") * 2
    0.662600000 USD/BTS
```

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!

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

invert()
Invert the price (e.g. go from `USD/BTS` into `BTS/USD`)

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

json()

return { "base": self["base"].json(), "quote": self["quote"].json()

}

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

market
Open the corresponding market.

Returns Instance of *bitshares.market.Market* for the corresponding pair of assets.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

symbols()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.price.**PriceFeed**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.price.PriceFeed*

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 **blockchain_instance** (bitshares.bitshares.BitShares) – BitShares instance

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

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

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

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.price.**UpdateCallOrder**(*call*, ***kwargs*)
Bases: *bitshares.price.Price*

This class inherits *bitshares.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 **blockchain_instance** (bitshares.bitshares.BitShares) – BitShares instance

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!

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

invert()
Invert the price (e.g. go from `USD/BTS` into `BTS/USD`)

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

json()

return { "base": self["base"].json(), "quote": self["quote"].json()

}

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

market
Open the corresponding market.

Returns Instance of *bitshares.market.Market* for the corresponding pair of assets.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

symbols()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.proposal module

class bitshares.proposal.**Proposal**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.proposal.Proposal*

Read data about a Proposal Balance in the chain.

Parameters

- **id** (*str*) – Id of the proposal
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

expiration

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

is_in_review

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

proposed_operations

proposer
Return the proposer of the proposal if available in the backend, else returns None

refresh()

review_period

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.proposal.**Proposals**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.proposal.Proposals*

Obtain a list of pending proposals for an account.

Parameters

- **account** (*str*) – Account name
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

append()
Append object to the end of the list.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

cache(*key*)
(legacy) store the current object with key `key`.

classmethod cache_objects(*data*, *key=None*)
This classmethod allows to feed multiple objects into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear()
Remove all items from list.

classmethod clear_cache()
Clear/Reset the entire Cache

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

define_classes()
Needs to define instance variables that provide classes

extend()
Extend list by appending elements from the iterable.

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

index()
Return first index of value.

Raises ValueError if the value is not present.

classmethod inject(*cls*)

insert()
Insert object before index.

items()
This overwrites items() so that refresh() is called if the object is not already fetched

pop()
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

refresh(**args*, ***kwargs*)
Interface that needs to be implemented. This method is called when an object is requested that has not yet been fetched/stored

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

sort()
Stable sort *IN PLACE*.

store(*data*, *key=None*, **args*, ***kwargs*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

bitshares.storage module

bitshares.storage.**get_default_config_store**(**args*, ***kwargs*)

bitshares.storage.**get_default_key_store**(*config*, **args*, ***kwargs*)

bitshares.transactionbuilder module

class bitshares.transactionbuilder.**ProposalBuilder**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.transactionbuilder.ProposalBuilder*

Proposal Builder allows us to construct an independent Proposal that may later be added to an instance ot TransactionBuilder.

Parameters

- **proposer** (*str*) – Account name of the proposing user
- **proposal_expiration** (*int*) – Number seconds until the proposal is supposed to expire
- **proposal_review** (*int*) – Number of seconds for review of the proposal
- **transactionbuilder.TransactionBuilder** – Specify your own instance of transaction builder (optional)
- **blockchain_instance** (*instance*) – Blockchain instance

appendOps(*ops*, *append_to=None*)
Append op(s) to the transaction builder

Parameters **ops** (*list*) – One or a list of operations

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

broadcast()

chain
Short form for blockchain (for the lazy)

define_classes()
Needs to define instance variables that provide classes

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

get_parent()
This allows to referr to the actual parent of the Proposal

get_raw()
Returns an instance of base "Operations" for further processing

classmethod inject(*cls*)

is_empty()

json()
Return the json formated version of this proposal

list_operations()

set_expiration(*p*)

set_parent(*p*)

set_proposer(*p*)

set_review(*p*)

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

class bitshares.transactionbuilder.**TransactionBuilder**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.transactionbuilder.TransactionBuilder*

This class simplifies the creation of transactions by adding operations and signers.

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!

add_required_fees(*ops*, *asset_id='1.3.0'*)
Auxiliary method to obtain the required fees for a set of operations. Requires a websocket connection to a witness node!

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(*accounts*, *permission*)
Try to obtain the wif key from the wallet by telling which account and permission is supposed to sign the transaction

Parameters

- **accounts** (*str*, *list*, *tuple*, *set*) – accounts to sign transaction with
- **permission** (*str*) – type of permission, e.g. "active", "owner" etc

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

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

broadcast()
Broadcast a transaction to the blockchain network

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

chain
Short form for blockchain (for the lazy)

clear()
Clear the transaction builder and start from scratch

constructTx()
Construct the actual transaction and store it in the class's dict store

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_block_params(*use_head_block=False*)
Auxiliary method to obtain `ref_block_num` and `ref_block_prefix`. Requires a websocket connection to a witness node!

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

get_parent()
TransactionBuilders don't have parents, they are their own parent

classmethod inject(*cls*)

is_empty()

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

json()
Show the transaction as plain json

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

list_operations()

permission_types = ['active', 'owner']

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

set_expiration(*p*)

set_fee_asset(*fee_asset*)
Set asset to fee

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

sign()
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.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

verify_authority()
Verify the authority of the signed transaction

bitshares.utils module

bitshares.vesting module

class bitshares.vesting.**Vesting**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.vesting.Vesting*

Read data about a Vesting Balance in the chain.

Parameters

- **id** (*str*) – Id of the vesting balance
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

account

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

claim(*amount=None*)

claimable

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

refresh()

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitshares.wallet module

class bitshares.wallet.**Wallet**(**args*, ***kwargs*)
Bases: `bitshares.instance.BlockchainInstance`, `bitshares.wallet.Wallet`

addPrivateKey(*wif*)
Add a private key to the wallet database

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of `bitshares.instance.BlockchainInstance`

chain
Short form for blockchain (for the lazy)

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

create(*pwd*)
Alias for newWallet()

created()
Do we have a wallet database already?

define_classes()
Needs to define instance variables that provide classes

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(*current=False*)
Return all installed public keys

Parameters **current** (*bool*) – If true, returns only keys for currently connected blockchain

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

classmethod inject(*cls*)

is_encrypted()
Is the key store encrypted?

lock()
Lock the wallet database

locked()
Is the wallet database locked?

newWallet(*pwd*)
Create a new wallet database

prefix

privatekey(*key*)

publickey_from_wif(*wif*)

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

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

rpc

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

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

unlock(*pwd*)
Unlock the wallet database

unlocked()
Is the wallet database unlocked?

wipe(*sure=False*)

bitshares.witness module

class bitshares.witness.**Witness**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.witness.Witness*

Read data about a witness in the chain.

Parameters

- **account_name** (*str*) – Name of the witness
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

account

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

is_active

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

refresh()

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

weight

class bitshares.witness.**Witnesses**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.witness.Witnesses*

Obtain a list of **active** witnesses and the current schedule.

Parameters

- **only_active** (*bool*) – (False) Only return witnesses that are actively producing blocks
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

append()
Append object to the end of the list.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

cache(*key*)
(legacy) store the current object with key `key`.

classmethod cache_objects(*data*, *key=None*)
This classmethod allows to feed multiple objects into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear()
Remove all items from list.

classmethod clear_cache()
Clear/Reset the entire Cache

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

define_classes()
Needs to define instance variables that provide classes

extend()
Extend list by appending elements from the iterable.

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

index()
Return first index of value.

Raises ValueError if the value is not present.

classmethod inject(*cls*)

insert()
Insert object before index.

items()
This overwrites items() so that refresh() is called if the object is not already fetched

pop()
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

refresh(**args*, ***kwargs*)
Interface that needs to be implemented. This method is called when an object is requested that has not yet been fetched/stored

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

sort()
Stable sort *IN PLACE*.

store(*data*, *key=None*, **args*, ***kwargs*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

bitshares.worker module

class bitshares.worker.**Worker**(**args*, ***kwargs*)

Bases: *bitshares.instance.BlockchainInstance*, *bitshares.worker.Worker*

Read data about a worker in the chain.

Parameters

- **id** (*str*) – id of the worker
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

account

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

classmethod cache_object(*data*, *key=None*)
This classmethod allows to feed an object into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear() → None. Remove all items from D.

classmethod clear_cache()
Clear/Reset the entire Cache

copy() → a shallow copy of D

define_classes()
Needs to define instance variables that provide classes

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

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

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

items()
This overwrites items() so that refresh() is called if the object is not already fetched

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

static objectid_valid(*i*)
Test if a string looks like a regular object id of the form::

```
xxxx.yyyyy.zzzz
```

with those being numbers.

perform_id_tests = True

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

post_format()

refresh()

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

space_id = 1

store(*data*, *key='id'*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

test_valid_objectid(*i*)
Alias for objectid_valid

testid(*id*)
In contrast to validity, this method tests if the objectid matches the type_id provided in self.type_id or self.type_ids

type_id = None

type_ids = []

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitshares.worker.**Workers**(**args*, ***kwargs*)
Bases: *bitshares.instance.BlockchainInstance*, *bitshares.worker.Workers*

Obtain a list of workers for an account.

Parameters

- **account_name/id** (*str*) – Name/id of the account (optional)
- **blockchain_instance** (*bitshares*) – BitShares() instance to use when accesing a RPC

append()
Append object to the end of the list.

bitshares
Alias for the specific blockchain.

blockchain

blockchain_instance_class
alias of *bitshares.instance.BlockchainInstance*

cache(*key*)
(legacy) store the current object with key `key`.

classmethod cache_objects(*data*, *key=None*)
This classmethod allows to feed multiple objects into the cache is is mostly used for testing

chain
Short form for blockchain (for the lazy)

clear()
Remove all items from list.

classmethod clear_cache()
Clear/Reset the entire Cache

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

define_classes()
Needs to define instance variables that provide classes

extend()
Extend list by appending elements from the iterable.

get_instance_class()
Should return the Chain instance class, e.g. *bitshares.BitShares*

getfromcache(*id*)
Get an element from the cache explicitly

identifier = None

incached(*id*)
Is an element cached?

index()
Return first index of value.

Raises ValueError if the value is not present.

classmethod inject(*cls*)

insert()
Insert object before index.

items()
This overwrites items() so that refresh() is called if the object is not already fetched

pop()
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

refresh(**args*, ***kwargs*)
Interface that needs to be implemented. This method is called when an object is requested that has not yet been fetched/stored

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

classmethod set_shared_blockchain_instance(*instance*)
This method allows us to override default instance for all users of `SharedInstance.instance`.

Parameters **instance** (*chaininstance*) – Chain instance

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

set_shared_instance()
This method allows to set the current instance as default

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

sort()
Stable sort *IN PLACE*.

store(*data*, *key=None*, **args*, ***kwargs*)
Cache the list

Parameters **data** (*list*) – List of objects to cache

Module contents

5.2 bitsharesbase

5.2.1 bitsharesbase package

Submodules

bitsharesbase.account module

class bitsharesbase.account.**Address**(*address*, *prefix=None*)
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 BTS)

Example:

```
Address("BTSFN9r6VYzBK8EKtMewfNbfiGCr56pHDBFi")
```

classmethod from_pubkey(*pubkey*, *compressed=True*, *version=56*, *prefix=None*)
Load an address provided the public key.

Version: 56 => PTS

prefix = 'BTS'

set_prefix(*prefix*)

class bitsharesbase.account.**BrainKey**(*brainkey=None*, *sequence=0*, *prefix=None*)
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.

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

prefix = 'BTS'

set_prefix(*prefix*)

static suggest()
Suggest a new random brain key. Randomness is provided by the operating system using os.urandom().

class bitsharesbase.account.**PasswordKey**(*account*, *password*, *role='active'*, *prefix=None*)
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.

get_private()
Derive private key from the brain key and the current sequence number

get_private_key()

get_public()

get_public_key()

prefix = 'BTS'

set_prefix(*prefix*)

class bitsharesbase.account.**PrivateKey**(*wif=None*, *prefix=None*)
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 BTS)

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.

address

bitcoin

child(*offset256*)
Derive new private key from this key and a sha256 "offset"

compressed

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_secret()
Get sha256 digest of the wif key.

prefix = 'BTS'

pubkey

set_prefix(*prefix*)

uncompressed

class bitsharesbase.account.**PublicKey**(*pk*, *prefix=None*)
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 BTS)

Example::

```
PublicKey("BTS6UtYWWs3rkZGV8JA86qrgkG6tyFksgECefKE1MiH4HkLD8PFGL")
```

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()
```

---

add(*digest256*)
Derive new public key from this key and a sha256 "digest"

address
Obtain a GrapheneAddress from a public key

child(*offset256*)
Derive new public key from this key and a sha256 "offset"

compressed()
returns the compressed key

compressed_key

classmethod from_privkey(*privkey*, *prefix=None*)
Derive uncompressed public key

point()
Return the point for the public key

prefix = 'BTS'

pubkey

set_prefix(*prefix*)

unCompressed()
Alias for self.uncompressed() - LEGACY

uncompressed()
Derive uncompressed key

bitsharesbase.asset_permissions module

bitsharesbase.asset_permissions.**force_flag**(*perms*, *flags*)

bitsharesbase.asset_permissions.**test_permissions**(*perms*, *flags*)

bitsharesbase.asset_permissions.**todict**(*number*)

bitsharesbase.asset_permissions.**toint**(*permissions*)

bitsharesbase.bip38 module

bitsharesbase.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)

bitsharesbase.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

bitsharesbase.chains module

bitsharesbase.memo module

bitsharesbase.objects module

class bitsharesbase.objects.**AccountCreateExtensions**(**args*, ***kwargs*)
Bases: *bitsharesbase.objects.Extension*

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

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class Null_ext(**args*, ***kwargs*)
Bases: `graphenebase.objects.GrapheneObject`

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

sorted_options = [('null_ext', <class 'bitsharesbase.objects.AccountCreateExtensions.N

bitsharesbase.objects.**AccountId**(*asset*)

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

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.objects.**AssertPredicate**(*o*)
Bases: graphenebase.types.Static_variant

bitsharesbase.objects.**AssetId**(*asset*)

class bitsharesbase.objects.**AssetOptions**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.objects.**BitAssetOptions**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.objects.**CallOrderExtension**(**args*, ***kwargs*)
Bases: *bitsharesbase.objects.Extension*

sorted_options = [('target_collateral_ratio', <function CallOrderExtension.targetColla

targetCollateralRatio()

class bitsharesbase.objects.**Extension**(**args*, ***kwargs*)
Bases: graphenebase.types.Array

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

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

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

Need to overwrite a few attributes to load proper object_types from bitshares.

object_types = {'OBJECT_TYPE_COUNT': 19, 'account': 2, 'asset': 3, 'balance': 15, '

class bitsharesbase.objects.**Operation**(*op*, ***kwargs*)
Bases: graphenebase.objects.Operation

Need to overwrite a few attributes to load proper operations from bitshares.

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

fromlist = ['operations']

getOperationIdForName(*name*)

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

id

index()
Return first index of value.

Raises ValueError if the value is not present.

insert()
Insert object before index.

json()

klass()

klass_name

module = 'bitsharesbase.operations'

op

opId

operation

operations = {'account_create': 5, 'account_transfer': 9, 'account_update': 6, 'acc

ops

pop()
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

set(***data*)

sort()
Stable sort *IN PLACE*.

toJson()

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

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

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

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.objects.**PriceFeed**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k[, d]*) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.objects.**SpecialAuthority**(*o*)
Bases: graphenebase.types.Static_variant

class bitsharesbase.objects.**Worker_initializer**(*o*)
Bases: graphenebase.types.Static_variant

bitsharesbase.objecttypes module

bitsharesbase.objecttypes.**object_type = {'OBJECT_TYPE_COUNT': 19, 'account': 2, 'asset':**
Object types for object ids

bitsharesbase.operationids module

bitsharesbase.operationids.**getOperationName**(*id: str*)
This method returns the name representation of an operation given its value as used in the API.

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

bitsharesbase.operationids.**ops = ['transfer', 'limit_order_create', 'limit_order_cancel',**
Operation ids

bitsharesbase.operations module

class bitsharesbase.operations.**Account_create**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Account_update**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Account_upgrade**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Account_whitelist**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

black_listed = 2

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

no_listing = 0

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

white_and_black_listed = 3

white_listed = 1

class bitsharesbase.operations.**Assert**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_claim_fees**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_claim_pool**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_create**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_fund_fee_pool**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k[, d]*) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_global_settle**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_issue**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_publish_feed**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_reserve**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_settle**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

detail(**args*, ***kwargs*)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_update**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_update_bitasset**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_update_feed_producers**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Asset_update_issuer**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Balance_claim**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

detail(**args*, ***kwargs*)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Bid_collateral**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

detail(**args*, ***kwargs*)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Call_order_update**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Committee_member_create**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Custom**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**HtlcHash**(*o*)
Bases: graphenebase.types.Static_variant

elements = [<class 'graphenebase.types.Ripemd160'>, <class 'graphenebase.types.Sha1'>,

class bitsharesbase.operations.**Htlc_create**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

detail(**args*, ***kwargs*)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Htlc_extend**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

detail(**args*, ***kwargs*)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Htlc_redeem**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

detail(**args*, ***kwargs*)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Limit_order_cancel**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Limit_order_create**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Liquidity_pool_create**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Liquidity_pool_delete**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Liquidity_pool_deposit**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Liquidity_pool_exchange**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Liquidity_pool_withdraw**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Op_wrapper**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Override_transfer**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Proposal_create**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Proposal_update**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Ticket_create_operation**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Ticket_update_operation**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Transfer**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Vesting_balance_withdraw**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Withdraw_permission_create**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Witness_update**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class bitsharesbase.operations.**Worker_create**(**args*, ***kwargs*)
Bases: graphenebase.objects.GrapheneObject

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

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

json()

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

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

bitsharesbase.operations.**fill_classmaps**()

bitsharesbase.operations.**getOperationClassForId**(*op_id*)
Convert an operation id into the corresponding class.

bitsharesbase.operations.**getOperationIdForClass**(*name*)
Convert an operation classname into the corresponding id.

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

bitsharesbase.signedtransactions module

class bitsharesbase.signedtransactions.**Signed_Transaction**(**args*, ***kwargs*)
Bases: graphenebase.signedtransactions.Signed_Transaction

Create a signed transaction and offer method to create the signature.

Parameters

- **refNum** (*num*) – parameter ref_block_num (see getBlockParams)
- **refPrefix** (*num*) – parameter ref_block_prefix (see getBlockParams)
- **expiration** (*str*) – expiration date
- **operations** (*Array*) – array of operations

clear() → None. Remove all items from od.

copy() → a shallow copy of od

data
Read data explicitly (backwards compatibility)

default_prefix = 'BTS'

deriveDigest(*chain*)

detail(**args*, ***kwargs*)

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

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

getChainParams(*chain*)

getKnownChains()

getOperationKlass()

get_default_prefix()

id
The transaction id of this transaction

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

json()

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

known_chains = {'BTS': {'chain_id': '4018d7844c78f6a6c41c6a552b898022310fc5dec06da467

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

operation_klass
alias of *bitsharesbase.objects.Operation*

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

toJson()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

verify(*pubkeys=[]*, *chain=None*)

bitsharesbase.transactions module

Module contents

CHAPTER 6

Glossary

6.1 Market Pegged Assets

Market Pegged Assets are a class of financial instruments available on the BitShares network. They implement a loan where one party gets stability with respect to an underlying asset and the other gets leverage against another digital asset that servers as backing collateral for the first.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W