python-peerplays Documentation

Release 0.1

Fabian Schuh

Mar 20, 2023

Contents

PeerPlays is a **blockchain-based autonomous company** (i.e. a DAC) that offers gaming and tournaments on a blockchain.

It is based on *Graphene* (tm), a blockchain technology stack (i.e. software) that allows for fast transactions and a scalable blockchain solution. In case of PeerPlays, it comes with decentralized gaming engine and allows setting up and running tournaments of any kind.

CHAPTER 1

About this Library

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

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

CHAPTER 2

Quickstart

Note:

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

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

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

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

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

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

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

CHAPTER 3

General

3.1 Installation

3.1.1 Installation

Install with *pip*:

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

Manual installation:

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

3.1.2 Upgrade

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

3.2 Quickstart

under construction

3.3 Tutorials

3.3.1 Bundle Many Operations

With PeerPlays, 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 peerplays import PeerPlays

testnet = PeerPlays(
    "wss://node.testnet.peerplays.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())
```

3.3.2 Proposing a Transaction

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

```
from pprint import pprint
from peerplays import PeerPlays

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

3.3.3 Simple Sell Script

```
from peerplays import PeerPlays
from peerplays.market import Market
from peerplays.price import Price
from peerplays.amount import Amount

#
# Instanciate PeerPlays (pick network via API node)
#
peerplays = PeerPlays(
    "wss://node.testnet.peerplays.eu",
```

(continues on next page)

(continued from previous page)

```
    nobroadcast=True    # <<--- set this to False when you want to fire!
)

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

#
# This defines the market we are looking at.
# The first asset in the first argument is the *quote*
# Sell and buy calls always refer to the *quote*
#
market = Market(
    "GOLD:USD",
    peerplays_instance=peerplays
)

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

3.3.4 Sell at a timely rate

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

(continues on next page)

(continued from previous page)

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

    #
    # This defines the market we are looking at.
    # The first asset in the first argument is the *quote*
    # Sell and buy calls always refer to the *quote*
    #
    market = Market(
        "GOLD:USD",
        peerplays_instance=peerplays
    )

    sell()
```

3.4 Configuration

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

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

and potentially more.

You can access those variables like a regular dictionary by using

```
from peerplays import PeerPlays
peerplays = PeerPlays()
print(peerplays.config.items())
```

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

3.5 Contributing to python-peerplays

We welcome your contributions to our project.

3.5.1 Repository

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

https://github.com/peerplays-network/python-peerplays

3.5.2 Flow

This project makes heavy use of git flow. If you are not familiar with it, then the most important thing for your to understand is that:

pull requests need to be made against the develop branch

3.5.3 How to Contribute

0. Familiarize yourself with *contributing on github <https://guides.github.com/activities/contributing-to-open-source/>*
1. Fork or branch from the master.
2. Create commits following the commit style
3. Start a pull request to the master branch
4. Wait for a @xeroc or another member to review

3.5.4 Issues

Feel free to submit issues and enhancement requests.

3.5.5 Contributing

Please refer to each project's style guidelines and guidelines for submitting patches and additions. In general, we follow the "fork-and-pull" Git workflow.

1. **Fork** the repo on GitHub
2. **Clone** the project to your own machine
3. **Commit** changes to your own branch
4. **Push** your work back up to your fork
5. Submit a **Pull request** so that we can review your changes

NOTE: Be sure to merge the latest from "upstream" before making a pull request!

3.5.6 Copyright and Licensing

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

3.6 Support and Questions

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

- https://peerplaystalk.org
- https://t.me/PeerPlaysDEX

3.7 Stati

List of statis and types used within PeerPlays:

```
class BetType(Enum):
    options = [
        "back",
        "lay",
    ]


class BettingMarketResolution(Enum):
    options = [
        "win",
        "not_win",
        "cancel",
        "BETTING_MARKET_RESOLUTION_COUNT",
    ]


class BettingMarketStatus(Enum):
    options = [
        "unresolved",  # no grading  has been published for this betting market
        "frozen",      # bets are suspended, no bets allowed
        "graded",      # grading of win or not_win has been published
        "canceled",    # the betting market is canceled, no further bets are allowed
        "settled",     # the betting market has been paid out
        "BETTING_MARKET_STATUS_COUNT"
    ]


class BettingMarketGroupStatus(Enum):
    options = [
        "upcoming",    # betting markets are accepting bets, will never go "in_play"
        "in_play",     # betting markets are delaying bets
        "closed",      # betting markets are no longer accepting bets
        "graded",      # witnesses have published win/not win for the betting markets
        "re_grading",  # initial win/not win grading has been challenged
        "settled",     # paid out
        "frozen",      # betting markets are not accepting bets
        "canceled",    # canceled
        "BETTING_MARKET_GROUP_STATUS_COUNT"
    ]


class EventStatus(Enum):
    options = [
        "upcoming",     # Event has not started yet, betting is allowed
        "in_progress",  # Event is in progress, if "in-play" betting is enabled, bets␣
↪will be delayed
        "frozen",       # Betting is temporarily disabled
        "finished",     # Event has finished, no more betting allowed
        "canceled",     # Event has been canceled, all betting markets have been␣
↪canceled
        "settled",      # All betting markets have been paid out
        "STATUS_COUNT"
    ]
```

CHAPTER 4

Command Line Tool

4.1 "peerplays" command line tool

The `peerplays` command line tool comes with the following features:

```
$ peerplays --help
Usage: peerplays [OPTIONS] COMMAND [ARGS]...

Options:
  --debug / --no-debug            Enable/Disable Debugging (no-broadcasting
                                  mode)
  --node TEXT                     Websocket URL for public Peerplays API
                                  (default: "wss://t.b.d./")
  --rpcuser TEXT                  Websocket user if authentication is required
  --rpcpassword TEXT              Websocket password if authentication is
                                  required
  -d, --nobroadcast / --broadcast
                                  Do not broadcast anything
  -x, --unsigned / --signed       Do not try to sign the transaction
  -e, --expires INTEGER           Expiration time in seconds (defaults to 30)
  -v, --verbose INTEGER           Verbosity (0-15)
  --version                       Show version
  --help                          Show this message and exit.

Commands:
  addkey                  Add a private key to the wallet
  allow                   Add a key/account to an account's permission
  approvecommittee        Approve committee member(s)
  approveproposal         Approve a proposal
  approvewitness          Approve witness(es)
  balance                 Show Account balances
  broadcast               Broadcast a json-formatted transaction
  changewalletpassphrase  Change the wallet passphrase
  configuration           Show configuration variables
```

(continues on next page)

(continued from previous page)

```
  delkey                  Delete a private key from the wallet
  disallow                Remove a key/account from an account's...
  disapprovecommittee     Disapprove committee member(s)
  disapproveproposal      Disapprove a proposal
  disapprovewitness       Disapprove witness(es)
  getkey                  Obtain private key in WIF format
  history                 Show history of an account
  info                    Obtain all kinds of information
  listaccounts            List accounts (for the connected network)
  listkeys                List all keys (for all networks)
  newaccount              Create a new account
  permissions             Show permissions of an account
  randomwif               Obtain a random private/public key pair
  set                     Set configuration key/value pair
  sign                    Sign a json-formatted transaction
  transfer                Transfer assets
  upgrade                 Upgrade Account
```

Further help can be obtained via:

```
$ peerplays <command> --help
```

CHAPTER 5

Packages

5.1 peerplays

5.1.1 peerplays package

Subpackages

peerplays.cli package

Submodules

peerplays.cli.account module

peerplays.cli.asset module

peerplays.cli.bookie module

peerplays.cli.bos module

peerplays.cli.cli module

peerplays.cli.committee module

peerplays.cli.decorators module

`peerplays.cli.decorators.`**`chain`**(*f*)

This decorator allows you to access `ctx.peerplays` which is an instance of PeerPlays.

`peerplays.cli.decorators.configfile(f)`
This decorator will parse a configuration file in YAML format and store the dictionary in `ctx.blockchain.config`

`peerplays.cli.decorators.customchain(**kwargsChain)`
This decorator allows you to access `ctx.peerplays` which is an instance of Peerplays. But in contrast to @chain, this is a decorator that expects parameters that are directed right to `PeerPlays()`.

... code-block::python

@main.command() @click.option("–worker", default=None) @click.pass_context @customchain(foo="bar") @unlock def list(ctx, worker):

print(ctx.obj)

`peerplays.cli.decorators.offline(f)`
This decorator allows you to access `ctx.peerplays` which is an instance of PeerPlays with `offline=True`.

`peerplays.cli.decorators.offlineChain(f)`
This decorator allows you to access `ctx.peerplays` which is an instance of PeerPlays with `offline=True`.

`peerplays.cli.decorators.online(f)`
This decorator allows you to access `ctx.peerplays` which is an instance of PeerPlays.

`peerplays.cli.decorators.onlineChain(f)`
This decorator allows you to access `ctx.peerplays` which is an instance of PeerPlays.

`peerplays.cli.decorators.unlock(f)`
This decorator will unlock the wallet by either asking for a passphrase or taking the environmental variable `UNLOCK`

`peerplays.cli.decorators.unlockWallet(f)`
This decorator will unlock the wallet by either asking for a passphrase or taking the environmental variable `UNLOCK`

`peerplays.cli.decorators.verbose(f)`
Add verbose flags and add logging handlers

peerplays.cli.info module

peerplays.cli.main module

peerplays.cli.message module

peerplays.cli.proposal module

peerplays.cli.rpc module

peerplays.cli.ui module

`peerplays.cli.ui.get_terminal(text='Password', confirm=False, allowedempty=False)`

`peerplays.cli.ui.maplist2dict(dlist)`
Convert a list of tuples into a dictionary

`peerplays.cli.ui.pprintOperation(op)`

`peerplays.cli.ui.pretty_print`(*o*, **args*, ***kwargs*)

`peerplays.cli.ui.print_permissions`(*account*)

`peerplays.cli.ui.print_version`(*ctx*, *param*, *value*)

peerplays.cli.wallet module

peerplays.cli.witness module

Module contents

Submodules

peerplays.account module

class `peerplays.account.Account`(**args*, ***kwargs*)

Bases: `peerplays.instance.BlockchainInstance`, `peerplays.account.Account`

This class allows to easily access Account data

Parameters

- **account_name** (*str*) – Name of the account
- **blockchain_instance** (*peerplays.peerplays.peerplays*) – peerplays 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 **peerplays.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 peerplays.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`.

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

blockchain

blockchain_instance_class
alias of `peerplays.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

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. *peerplays.PeerPlays*

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.

peerplays
Alias for the specific blockchain

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

static set_cache_store(*klass*, **args*, ***kwargs*)

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 peerplays.account.**AccountUpdate**(**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.account.AccountUpdate*

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

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

... code-block: js

{'id': '2.6.29', 'lifetime_fees_paid': '44261516129', 'most_recent_op': '2.9.0', 'owner': '1.2.29', 'pending_fees': 0, 'pending_vested_fees': 16310, 'total_core_in_orders': '6788845277634', 'total_ops': 0}

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

account_class
alias of *Account*

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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

peerplays.amount module

class peerplays.amount.**Amount**(**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.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** (*peerplays.peerplays.peerplays*) – peerplays 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 *peerplays.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 *peerplays.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*

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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

peerplays.asset module

class peerplays.asset.**Asset**(**args*, ***kwargs*)

Bases: *peerplays.instance.BlockchainInstance*, *peerplays.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** (*instance*) – Instance of blockchain

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()`.

---

blockchain

blockchain_instance_class

alias of *peerplays.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

ensure_full()

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_instance_class()

Should return the Chain instance class, e.g. *peerplays.PeerPlays*

getfromcache(*id*)

Get an element from the cache explicitly

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?

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.

peerplays
Alias for the specific blockchain

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

static set_cache_store(*klass*, **args*, ***kwargs*)

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

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

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

peerplays.bet module

class peerplays.bet.**Bet**(**args*, ***kwargs*)
Bases: *peerplays.blockchainobject.BlockchainObject*

Read data about a Bet on the chain

Parameters

- **identifier** (*str*) – Identifier
- **blockchain_instance** (*peerplays*) – PeerPlays() instance to use when accesing a RPC

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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.

peerplays
Alias for the specific blockchain

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()

static set_cache_store(*klass*, **args*, ***kwargs*)

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 = 26

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

peerplays.bettingmarket module

class peerplays.bettingmarket.**BettingMarket**(**args*, ***kwargs*)
Bases: *peerplays.blockchainobject.BlockchainObject*

Read data about a Betting Market on the chain

Parameters

- **identifier** (*str*) – Identifier
- **blockchain_instance** (*peerplays*) – PeerPlays() instance to use when accesing a RPC

bettingmarketgroup

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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.

peerplays
Alias for the specific blockchain

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()

static set_cache_store(*klass*, **args*, ***kwargs*)

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 = 25

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 peerplays.bettingmarket.**BettingMarkets**(*betting_market_group_id*, **args*, ***kwargs*)

Bases: *peerplays.blockchainobject.BlockchainObjects*, *peerplays.instance.BlockchainInstance*

List of all available BettingMarkets

Parameters **betting_market_group_id** (*str*) – Market Group ID (`1.24.xxx`)

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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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*.

static set_cache_store(*klass*, **args*, ***kwargs*)

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

peerplays.bettingmarketgroup module

class peerplays.bettingmarketgroup.**BettingMarketGroup**(**args*, ***kwargs*)
Bases: *peerplays.blockchainobject.BlockchainObject*

Read data about a Betting Market Group on the chain

Parameters

- **identifier** (*str*) – Identifier
- **blockchain_instance** (*peerplays*) – PeerPlays() instance to use when accesing a RPC

bettingmarkets

blockchain

blockchain_instance_class
alias of *peerplays.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

event

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_dynamic_type()

get_instance_class()
Should return the Chain instance class, e.g. *peerplays.PeerPlays*

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

identifier = None

incached(*id*)
Is an element cached?

classmethod inject(*cls*)

is_dynamic()

is_dynamic_type(*other_type*)

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.

peerplays
Alias for the specific blockchain

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()

resolve(*results*, ***kwargs*)

static set_cache_store(*klass*, **args*, ***kwargs*)

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 = 24

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 peerplays.bettingmarketgroup.**BettingMarketGroups**(*event_id*, **args*, ***kwargs*)
Bases: *peerplays.blockchainobject.BlockchainObjects*, *peerplays.instance.BlockchainInstance*

List of all available BettingMarketGroups

Parameters **strevent_id** – Event ID (`1.22.xxx`)

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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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*.

static set_cache_store(*klass*, **args*, ***kwargs*)

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

peerplays.block module

class peerplays.block.**Block**(**args*, ***kwargs*)

Bases: *peerplays.instance.BlockchainInstance*, *peerplays.block.Block*

Read a single block from the chain

Parameters

- **block** (*int*) – block number
- **blockchain_instance** (*instance*) – blockchain 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 .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()`.

blockchain

blockchain_instance_class

alias of *peerplays.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. *peerplays.PeerPlays*

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.

peerplays
Alias for the specific blockchain

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

static set_cache_store(*klass*, **args*, ***kwargs*)

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 peerplays.block.**BlockHeader**(**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.block.BlockHeader*

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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.

peerplays
Alias for the specific blockchain

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

static set_cache_store(*klass*, **args*, ***kwargs*)

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

peerplays.blockchain module

class peerplays.blockchain.**Blockchain**(**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.blockchain.Blockchain*

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

Parameters

- **blockchain_instance** (*instance*) – 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.

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 *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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

peerplays.blockchainobject module

class peerplays.blockchainobject.**BlockchainObject**(**args*, ***kwargs*)

Bases: *peerplays.instance.BlockchainInstance*, *peerplays.blockchainobject.BlockchainObject*

blockchain

blockchain_instance_class

alias of *peerplays.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. *peerplays.PeerPlays*

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.

peerplays

Alias for the specific blockchain

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.

static set_cache_store(*klass*, **args*, ***kwargs*)

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 peerplays.blockchainobject.**BlockchainObjects**(**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.blockchainobject.BlockchainObjects*

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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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*.

static set_cache_store(*klass*, **args*, ***kwargs*)

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

peerplays.committee module

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

Read data about a Committee Member in the chain

Parameters

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

account

account_id

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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.

peerplays
Alias for the specific blockchain

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()

static set_cache_store(*klass*, **args*, ***kwargs*)

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

peerplays.event module

class peerplays.event.**Event**(**args*, ***kwargs*)
Bases: *peerplays.blockchainobject.BlockchainObject*

Read data about an event on the chain

Parameters

- **identifier** (*str*) – Identifier
- **blockchain_instance** (*peerplays*) – PeerPlays() instance to use when accesing a RPC

bettingmarketgroups

blockchain

blockchain_instance_class
alias of *peerplays.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

eventgroup

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. *peerplays.PeerPlays*

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.

peerplays
Alias for the specific blockchain

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()

static set_cache_store(*klass*, **args*, ***kwargs*)

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

set_status(*status*, *scores=[]*, ***kwargs*)

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 = 22

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 peerplays.event.**Events**(*eventgroup_id*, **args*, ***kwargs*)
Bases: *peerplays.blockchainobject.BlockchainObjects*, *peerplays.instance.BlockchainInstance*

List of all available events in an eventgroup

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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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*.

static set_cache_store(*klass*, **args*, ***kwargs*)

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

peerplays.eventgroup module

class peerplays.eventgroup.**EventGroup**(**args*, ***kwargs*)
Bases: *peerplays.blockchainobject.BlockchainObject*

Read data about an event group on the chain

Parameters

- **identifier** (*str*) – Identifier
- **blockchain_instance** (*peerplays*) – PeerPlays() instance to use when accesing a RPC

blockchain

blockchain_instance_class
alias of *peerplays.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

events

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. *peerplays.PeerPlays*

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.

peerplays
Alias for the specific blockchain

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()

static set_cache_store(*klass*, **args*, ***kwargs*)

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

sport

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 = 21

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 peerplays.eventgroup.**EventGroups**(*sport_id*, **args*, ***kwargs*)
Bases: *peerplays.blockchainobject.BlockchainObjects*, *peerplays.instance.BlockchainInstance*

List of all available EventGroups

Parameters **sport_id** (*str*) – Sport ID (`1.20.xxx`)

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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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*.

static set_cache_store(*klass*, **args*, ***kwargs*)

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

peerplays.exceptions module

exception peerplays.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 peerplays.exceptions.**BetDoesNotExistException**

Bases: Exception

This bet does not exist

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception peerplays.exceptions.**BettingMarketDoesNotExistException**

Bases: Exception

Betting market does not exist

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception peerplays.exceptions.**BettingMarketGroupDoesNotExistException**

Bases: Exception

Betting Market Group does not exist

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception peerplays.exceptions.**EventDoesNotExistException**

Bases: Exception

This event does not exist

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception peerplays.exceptions.**EventGroupDoesNotExistException**

Bases: Exception

This event group does not exist

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception peerplays.exceptions.**GenesisBalanceDoesNotExistsException**

Bases: Exception

The provided genesis balance id does not exist

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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

The transaction requires signature of a higher authority

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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

An RPC connection is required

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception peerplays.exceptions.**RuleDoesNotExistException**
Bases: Exception

Rule does not exist

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception peerplays.exceptions.**SportDoesNotExistException**
Bases: Exception

Sport does not exist

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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

The password provided could not properly unlock the wallet

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

peerplays.genesisbalance module

class peerplays.genesisbalance.**GenesisBalance**(**args*, ***kwargs*)

Bases: *peerplays.instance.BlockchainInstance*, *peerplays.genesisbalance.GenesisBalance*

Read data about a Committee Member in the chain

Parameters

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

blockchain

blockchain_instance_class

alias of *peerplays.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. *peerplays.PeerPlays*

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.

peerplays
Alias for the specific blockchain

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()

static set_cache_store(*klass*, **args*, ***kwargs*)

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 peerplays.genesisbalance.**GenesisBalances**(**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.genesisbalance.GenesisBalances*

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

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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

index()
Return first index of value.

Raises ValueError if the value is not present.

classmethod inject(*cls*)

insert()
Insert object before index.

peerplays
Alias for the specific blockchain

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*.

peerplays.instance module

class peerplays.instance.**BlockchainInstance**(**args*, ***kwargs*)
Bases: `graphenecommon.instance.AbstractBlockchainInstanceProvider`

This is a class that allows compatibility with previous naming conventions

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. *peerplays.PeerPlays*

classmethod inject(*cls*)

peerplays
Alias for the specific blockchain

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 peerplays.instance.**SharedInstance**

Bases: object

This class merely offers a singelton for the Blockchain Instance

config = {}

instance = None

peerplays.instance.**set_shared_blockchain_instance**(*instance*)

peerplays.instance.**set_shared_config**(*config*)

peerplays.instance.**set_shared_peerplays_instance**(*instance*)

peerplays.instance.**shared_blockchain_instance**()

peerplays.instance.**shared_peerplays_instance**()

peerplays.market module

class peerplays.market.**Market**(**args*, ***kwargs*)

Bases: *peerplays.instance.BlockchainInstance*, *peerplays.market.Market*

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

Parameters

- **blockchain_instance** (peerplays.peerplays.PeerPlays) – Peerplays instance
- **base** (peerplays.asset.Asset) – Base asset
- **quote** (peerplays.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 *peerplays.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. BTC:PPY with BTC being the quote), while the base only refers to a secondary asset for a trade. This means, if you call *peerplays.market.Market.sell()* or *peerplays.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.

blockchain

blockchain_instance_class

alias of `peerplays.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 BTC_PPY market is priced in PPY per BTC.

Example: in the BTC_PPY market, a price of 400 means a BTC is worth 400 PPY

Note: All prices returned are in the **reversed** orientation as the market. I.e. in the BTC/PPY market, prices are PPY 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 BTC for 100 PPY/BTC
- This means that you actually place a sell order for 1000 PPY in order to obtain **at least** 10 PPY

- If an order on the market exists that sells BTC for cheaper, you will end up with more than 10 BTC

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. *peerplays.PeerPlays*

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 BTC/PPY (1.9103 BTC|519.29602 PPY),
0.003676 BTC/PPY (299.9997 BTC|81606.16394 PPY),
0.003665 BTC/PPY (288.4618 BTC|78706.21881 PPY),
0.003665 BTC/PPY (3.5285 BTC|962.74409 PPY),
0.003665 BTC/PPY (72.5474 BTC|19794.41299 PPY),
[0.003738 BTC/PPY (36.4715 BTC|9756.17339 PPY),
0.003738 BTC/PPY (18.6915 BTC|5000.00000 PPY),
0.003742 BTC/PPY (182.6881 BTC|48820.22081 PPY),
0.003772 BTC/PPY (4.5200 BTC|1198.14798 PPY),
0.003799 BTC/PPY (148.4975 BTC|39086.59741 PPY)]
```

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. `BTC:PPY`

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

Note: Each bid is an instance of class:*peerplays.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!

peerplays
Alias for the specific blockchain

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 BTC_PPY market is priced in PPY per BTC.

Example: in the BTC_PPY market, a price of 300 means a BTC is worth 300 PPY

Note: All prices returned are in the **reversed** orientation as the market. I.e. in the BTC/PPY market, prices are PPY 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-PPY asset
- `price24h`: the price 24h ago

Sample Output:

```
{
    {
        "quoteVolume": 48328.73333,
        "quoteSettlement_price": 332.3344827586207,
        "lowestAsk": 340.0,
        "baseVolume": 144.1862,
        "percentChange": -1.9607843231354893,
```

(continues on next page)

(continued from previous page)

```
        "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:

```
{
    "PPY": 41.12345,
    "BTC": 1.0
}
```

peerplays.memo module

class peerplays.memo.**Memo** (**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.memo.Memo*

Deals with Memos that are attached to a transfer

Parameters

- **from_account** (peerplays.account.Account) – Account that has sent the memo
- **to_account** (peerplays.account.Account) – Account that has received the memo
- **blockchain_instance** (peerplays.peerplays.PeerPlays) – 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 peerplays.memo import Memo
m = Memo("from", "to")
m.unlock_wallet("secret")
enc = (m.encrypt("foobar"))
print(enc)
>> {'nonce': '17329630356955254641', 'message': '8563e2bb2976e0217806d642901a2855
↪'}
```

(continues on next page)

(continued from previous page)

```
print(m.decrypt(enc))
>> foobar
```

To decrypt a memo, simply use

```
from peerplays.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.

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

classmethod inject(*cls*)

peerplays
Alias for the specific blockchain

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

peerplays.message module

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

MESSAGE_SPLIT = ('-----BEGIN PEERPLAYS 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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

classmethod inject(*cls*)

peerplays
Alias for the specific blockchain

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.message.MessageV1'>, <class 'graphenecommo

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

peerplays.notify module

class peerplays.notify.**Notify**(*accounts=[], objects=[], on_tx=None, on_object=None, on_block=None, on_account=None, peerplays_instance=None*)

Bases: `events.events.Events`

Notifications on Blockchain events.

Parameters

- **accounts** (*list*) – Account names/ids to be notified about when changing
- **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
- **peerplays_instance** (`peerplays.peerplays.PeerPlays`) – PeerPlays instance

Example

```
from pprint import pprint
from peerplays.notify import Notify

notify = Notify(
    accounts=["xeroc"],
    on_account=print,
    on_block=print,
    on_tx=print
)
notify.listen()
```

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:peerplays.account.AccountUpdate‘

peerplays.peerplays module

class peerplays.peerplays.**PeerPlays**(*node=”, rpcuser=”, rpcpassword=”, debug=False, skip_wallet_init=False, **kwargs*)

Bases: graphenecommon.chain.AbstractGrapheneChain

Connect to the PeerPlays 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 peerplayslibs load the keys from the locally stored wallet SQLite database (see `storage.py`). To use this mode, simply call `PeerPlays()` 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 `PeerPlays()`.
- **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://ppy-node.peerplays.eu. It is **highly** recommended that you pick your own node instead. Default settings can be changed with:

```
peerplays set node <host>
```

where `<host>` starts with `ws://` or `wss://`.

The purpose of this class it to simplify interaction with PeerPlays.

The idea is to have a class that allows to do this:

```
from peerplays import PeerPlays
peerplays = PeerPlays()
print(peerplays.info())
```

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

```
peerplays addkey
```

and setting a default author:

```
peerplays set default_account xeroc
```

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

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
- **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`)

bet_cancel (*bet_to_cancel*, *account=None*, ***kwargs*)

Cancel a bet

Parameters

- **bet_to_cancel** (*str*) – The identifier that identifies the bet to cancel
- **account** (*str*) – (optional) the account that owns the bet (defaults to `default_account`)

bet_place (*betting_market_id*, *amount_to_bet*, *backer_multiplier*, *back_or_lay*, *account=None*, ***kwargs*)

Place a bet

Parameters

- **betting_market_id** (*str*) – The identifier for the market to bet in
- **amount_to_bet** (`peerplays.amount.Amount`) – Amount to bet with
- **backer_multiplier** (*int*) – Multipler for backer
- **back_or_lay** (*str*) – "back" or "lay" the bet
- **account** (*str*) – (optional) the account to bet (defaults to `default_account`)

betting_market_create (*payout_condition*, *description*, *group_id='0.0.0'*, *account=None*, ***kwargs*)

Create an event group. This needs to be **proposed**.

Parameters

- **payout_condition** (*list*) – Internationalized names, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **description** (*list*) – Internationalized descriptions, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **group_id** (*str*) – Group ID to create the market for (defaults to *relative* id `0.0.0`)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

betting_market_group_create (*description*, *event_id='0.0.0'*, *rules_id='0.0.0'*, *asset=None*, *delay_before_settling=0*, *never_in_play=False*, *resolution_constraint='exactly_one_winner'*, *account=None*, ***kwargs*)

Create an betting market. This needs to be **proposed**.

Parameters

- **description** (*list*) – Internationalized list of descriptions
- **event_id** (*str*) – Event ID to create this for (defaults to *relative* id `0.0.0`)
- **rule_id** (*str*) – Rule ID to create this with (defaults to *relative* id `0.0.0`)
- **asset** (`peerplays.asset.Asset`) – Asset to be used for this market
- **delay_before_settling** (*int*) – Delay in seconds before settling (defaults to 0 seconds - immediatelly)
- **never_in_play** (*bool*) – Set this market group as *never in play* (defaults to *False*)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

betting_market_group_update(*betting_market_group_id*, *description=None*, *event_id=None*, *rules_id=None*, *status=None*, *account=None*, ***kwargs*)

Update an betting market. This needs to be **proposed**.

Parameters

- **betting_market_group_id** (*str*) – Id of the betting market group to update
- **description** (*list*) – Internationalized list of descriptions
- **event_id** (*str*) – Event ID to create this for
- **rule_id** (*str*) – Rule ID to create this with
- **status** (*str*) – New Status
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

betting_market_resolve(*betting_market_group_id*, *results*, *account=None*, ***kwargs*)

Create an betting market. This needs to be **proposed**.

Parameters

- **betting_market_group_id** (*str*) – Market Group ID to resolve
- **results** (*list*) – Array of Result of the market (`win`, `not_win`, or `cancel`)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

Results take the form::

```
[
    ["1.21.257", "win"],
    ["1.21.258", "not_win"],
    ["1.21.259", "cancel"],
]
```

betting_market_rules_create(*names*, *descriptions*, *account=None*, ***kwargs*)

Create betting market rules

Parameters

- **names** (*list*) – Internationalized names, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **descriptions** (*list*) – Internationalized descriptions, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

betting_market_rules_update(*rules_id*, *names*, *descriptions*, *account=None*, ***kwargs*)

Update betting market rules

Parameters

- **rules_id** (*str*) – Id of the betting market rules to update
- **names** (*list*) – Internationalized names, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **descriptions** (*list*) – Internationalized descriptions, e.g. `[['de', 'Foo'], ['en', 'bar']]`

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

betting_market_update(*betting_market_id*, *payout_condition*, *description*, *group_id='0.0.0'*, *account=None*, ***kwargs*)

Update an event group. This needs to be **proposed**.

Parameters

- **betting_market_id** (*str*) – Id of the betting market to update
- **payout_condition** (*list*) – Internationalized names, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **description** (*list*) – Internationalized descriptions, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **group_id** (*str*) – Group ID to create the market for (defaults to *relative* id `0.0.0`)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

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_offer(*issuer_account_id_or_name*, *offer_id*, ***kwargs*)

clear()

clear_cache()

Clear Caches

connect(*node=''*, *rpcuser=''*, *rpcpassword=''*, ***kwargs*)

Connect to blockchain network (internal use only)

create_account(*account_name*, *registrar=None*, *referrer='1.2.0'*, *referrer_percent=50*, *owner_key=None*, *active_key=None*, *memo_key=None*, *password=None*, *additional_owner_keys=[]*, *additional_active_keys=[]*, *additional_owner_accounts=[]*, *additional_active_accounts=[]*, *proxy_account='proxy-to-self'*, *storekeys=True*, ***kwargs*)

Create new account on PeerPlays

The brainkey/password can be used to recover all generated keys (see *peerplaysbase.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_bid(*bidder_account_id_or_name*, *bid_price*, *offer_id*, ***kwargs*)

create_offer(*item_ids*, *issuer_id_or_name*, *minimum_price*, *maximum_price*, *buying_item*, *offer_expiration_date*, *memo=None*, ***kwargs*)

custom_account_authority_create(*permission_id*, *operation_type*, *valid_from*, *valid_to*, *owner_account=None*, ***kwargs*)

custom_account_authority_delete(*auth_id*, *owner_account=None*, ***kwargs*)

custom_account_authority_update(*auth_id*, *new_valid_from*, *new_valid_to*, *owner_account=None*, ***kwargs*)

custom_permission_create(*permission_name*, *owner_account=None*, *weight_threshold=[]*, *account_auths=[]*, *key_auths=[]*, *address_auths=[]*, ***kwargs*)

custom_permission_delete(*permission_id*, *owner_account=None*, ***kwargs*)

custom_permission_update(*permission_id*, *owner_account=None*, *weight_threshold=[]*, *account_auths=[]*, *key_auths=[]*, *address_auths=[]*, ***kwargs*)

define_classes()

deleteproposal(*proposal_id*, *account=None*, ***kwargs*)

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`)

event_create(*name*, *season*, *start_time*, *event_group_id='0.0.0'*, *account=None*, ***kwargs*)
Create an event. This needs to be **proposed**.

Parameters

- **name** (*list*) – Internationalized names, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **season** (*list*) – Internationalized season, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **event_group_id** (*str*) – Event group ID to create the event for (defaults to *relative* id `0.0.0`)
- **start_time** (*datetime*) – Time of the start of the event
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

event_group_create(*names*, *sport_id='0.0.0'*, *account=None*, ***kwargs*)
Create an event group. This needs to be **proposed**.

Parameters

- **names** (*list*) – Internationalized names, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **sport_id** (*str*) – Sport ID to create the event group for (defaults to *relative* id `0.0.0`)

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

event_group_update(*event_group_id*, *names=[]*, *sport_id='0.0.0'*, *account=None*, ***kwargs*)

Update an event group. This needs to be **proposed**.

Parameters

- **event_id** (*str*) – Id of the event group to update
- **names** (*list*) – Internationalized names, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **sport_id** (*str*) – Sport ID to create the event group for (defaults to *relative* id `0.0.0`)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

event_update(*event_id*, *name=None*, *season=None*, *start_time=None*, *event_group_id=None*, *status=None*, *account=None*, ***kwargs*)

Update an event. This needs to be **proposed**.

Parameters

- **event_id** (*str*) – Id of the event to update
- **name** (*list*) – Internationalized names, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **season** (*list*) – Internationalized season, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **event_group_id** (*str*) – Event group ID to create the event for (defaults to *relative* id `0.0.0`)
- **start_time** (*datetime*) – Time of the start of the event
- **status** (*str*) – Event status
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

event_update_status(*event_id*, *status*, *scores=[]*, *account=None*, ***kwargs*)

Update the status of an event. This needs to be **proposed**.

Parameters

- **event_id** (*str*) – Id of the event to update
- **status** (*str*) – Event status
- **scores** (*list*) – List of strings that represent the scores of a match (defaults to [])
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

eventgroup_delete(*event_group_id='0.0.0'*, *account=None*, ***kwargs*)

Delete an eventgroup. This needs to be **propose**.

Parameters

- **event_group_id** (*str*) – ID of the event group to be deleted
- **account** (*str*) – (optional) Account used to verify the operation

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

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

nft_approve (*operator_*, *approved*, *token_id*, ***kwargs*)

nft_metadata_create (*owner_account_id_or_name*, *name*, *symbol*, *base_uri*, *revenue_partner=None*, *revenue_split=200*, *is_transferable=True*, *is_sellable=True*, *role_id=None*, *max_supply=None*, *lottery_options=None*, ***kwargs*)

nft_metadata_update (*owner_account_id_or_name*, *nft_metadata_id*, *name*, *symbol*, *base_uri*, *revenue_partner=None*, *revenue_split=200*, *is_transferable=True*, *is_sellable=True*, *role_id=None*, ***kwargs*)

nft_mint(*metadata_owner_account_id_or_name*, *metadata_id*, *owner_account_id_or_name*, *approved_account_id_or_name*, *approved_operators*, *token_uri*, ***kwargs*)

nft_safe_transfer_from(*operator_*, *from_*, *to_*, *token_id*, *data*, ***kwargs*)

nft_set_approval_for_all(*owner*, *operator_*, *approved*, ***kwargs*)

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!

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_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.

sport_create(*names*, *account=None*, ***kwargs*)
Create a sport. This needs to be **proposed**.

Parameters

- **names** (*list*) – Internationalized names, e.g. `[['de', 'Foo'], ['en', 'bar']]`
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

sport_delete(*sport_id='0.0.0'*, *account=None*, ***kwargs*)
Remove a sport. This needs to be **proposed**.

Parameters

- **sport_id** (*str*) – Sport ID to identify the Sport to be deleted
- **account** (*str*) – (optional) Account used to verify the operation

sport_update(*sport_id*, *names=[]*, *account=None*, ***kwargs*)
Update a sport. This needs to be **proposed**.

Parameters

- **sport_id** (*str*) – The id of the sport to update
- **names** (*list*) – Internationalized names, e.g. `[['de', 'Foo'], ['en', 'bar']]`

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

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

Parameters

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

tx()
Returns the default transaction buffer

txbuffer
Returns the currently active tx buffer

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

update_memo_key(*key*, *account=None*, ***kwargs*)
Update an account's memo public key

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

Parameters

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

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`)

peerplays.peerplays2 module

class peerplays.peerplays2.**PeerPlays**(*urlWalletServer*)
Bases: `object`

This class is http endpoint based implementation of peerplays operations : param str urlWalletServer: Remote wallet server

```
from peerplays.peerplays2 import PeerPlays as PeerPlays2
peerplays2 = PeerPlays2(urlWalletServer=urlWalletServer)
```

where `<urlWalletServer>` starts with `http://` or `https://`.

The purpose of this class it to simplify interaction with a few of the new PeerPlays features and changes.

The idea is to have a class that allows to do this

WalletCall(*method*, *params=[]*)
Genric method for making calls to peerplays node through remote wallet. :param str method: Name of the cli_wallet command to call :param str params: Parameters to the command

create_account(*account_name*, *registrar='None'*, *referrer='1.2.0'*, *referrer_percent=50*, *owner_key=None*, *active_key=None*, *memo_key=None*)
Create new account. This method is more for back compatibility :param str accountName: New account name :param str ownerKey: Owner key :param str activeKey: Active key :param str registrAccount: Registrar :param str referreAccount: Referrer :param str referrerPercent: Referrer percent

import_key(*accountName*, *wif*)
Import keys to the wallet :param str accountName: AccoutName :param strr wif: WIF of the account

info()
Info command

is_locked()
Check if wallet is locked

register_account(*accountName*, *ownerKey*, *activeKey*, *registrarAccount*, *referrerAccount*, *referrerPercent*)
Create new account :param str accountName: New account name :param str ownerKey: Owner key :param str activeKey: Active key :param str registrAccount: Registrar :param str referreAccount: Referrer :param str referrerPercent: Referrer percent

set_password(*password*)
Set remote wallet password param str password: New wallet password

suggest_brain_key()

unlock(*password*)
Method to unlock wallet :param str password: Remote wallet password

wallet_server()

wallet_server_start()

peerplays.price module

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

This class inherits *peerplays.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** (*peerplays.peerplays.PeerPlays*) – PeerPlays 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!

blockchain

blockchain_instance_class
alias of `peerplays.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. *peerplays.PeerPlays*

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 `peerplays.market.Market` for the corresponding pair of assets.

peerplays
Alias for the specific blockchain

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 peerplays.price.**Order**(**args*, ***kwargs*)
Bases: *peerplays.price.Price*

This class inherits *peerplays.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** (*peerplays.peerplays.PeerPlays*) – PeerPlays 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!

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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 `peerplays.market.Market` for the corresponding pair of assets.

peerplays
Alias for the specific blockchain

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 peerplays.price.**Price**(**args*, ***kwargs*)

Bases: *peerplays.instance.BlockchainInstance*, *peerplays.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:`peerplays.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 peerplays.asset.Asset base: Base asset
:param peerplays.asset.Asset quote: Quote asset
:param peerplays.peerplays.PeerPlays blockchain_instance: PeerPlays 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:`peerplays.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:`peerplays.amount.Amount`
    * ``args`` being a list of ``[quote, base]`` both being instances of ``str``␣
↪(``amount symbol``)
    * ``base`` and ``quote`` being instances of :class:`peerplays.asset.Amount`

This allows instanciations like:

* ``Price("0.315 BTC/PPY")``
* ``Price(0.315, base="BTC", quote="PPY")``
* ``Price(0.315, base=Asset("BTC"), quote=Asset("PPY"))``
* ``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 BTC")``
* ``Price("10 GOLD", "1 BTC")``
* ``Price(Amount("10 GOLD"), Amount("1 BTC"))``
* ``Price(1.0, "BTC/GOLD")``

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

.. code-block:: python

    >>> from peerplays.price import Price
```

(continues on next page)

(continued from previous page)

```
>>> Price("0.3314 BTC/PPY") * 2
0.662600000 BTC/PPY
```

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!

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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 *peerplays.market.Market* for the corresponding pair of assets.

peerplays
Alias for the specific blockchain

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 peerplays.price.**PriceFeed**(**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.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** (*peerplays.peerplays.PeerPlays*) – PeerPlays instance

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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 peerplays.price.**UpdateCallOrder**(*call*, ***kwargs*)
Bases: *peerplays.price.Price*

This class inherits *peerplays.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** (peerplays.peerplays.PeerPlays) – PeerPlays 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!

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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 *peerplays.market.Market* for the corresponding pair of assets.

peerplays
Alias for the specific blockchain

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

peerplays.proposal module

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

Read data about a Proposal Balance in the chain

Parameters

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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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.

peerplays
Alias for the specific blockchain

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

static set_cache_store(*klass*, **args*, ***kwargs*)

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 peerplays.proposal.**Proposals**(**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.proposal.Proposals*

Obtain a list of pending proposals for an account

Parameters

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

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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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*.

static set_cache_store(*klass*, **args*, ***kwargs*)

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

peerplays.rule module

class peerplays.rule.**Rule**(**args*, ***kwargs*)
Bases: *peerplays.blockchainobject.BlockchainObject*

Read data about a Rule object

Parameters

- **identifier** (*str*) – Identifier for the rule
- **blockchain_instance** (*peerplays*) – PeerPlays() instance to use when accesing a RPC

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

grading

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.

peerplays
Alias for the specific blockchain

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()

static set_cache_store(*klass*, **args*, ***kwargs*)

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 = 23

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 peerplays.rule.**Rules**(**args*, *limit=1000*, ***kwargs*)
Bases: *peerplays.blockchainobject.BlockchainObjects*, *peerplays.instance.BlockchainInstance*

List of all Rules

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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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*.

static set_cache_store(*klass*, **args*, ***kwargs*)

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

peerplays.son module

class peerplays.son.**Son**(*urlWitness*)
Bases: `object`

This class is http endpoint based implementation of Son operations

create_son(*account_name*, *url*, *sidechainPublicKeyListOfList*)

delete_sidechain_address(*account_name*, *sidechain*)

heartbeat()

is_locked()

report_down()

request_son_maintenance(*account_name*)

set_password(*password*)

sidechain_deposit_transaction(*sidechain*, *transaction_id*, *operation_index*, *sidechain_from*, *sidechain_to*, *sidechain_currency*, *siechain_amount*, *peerplays_from_name_or_id*, *peerplays_to_name_or_id*)

params: const sidechain_type& sidechain, const string &transaction_id, uint32_t operation_index, const string &sidechain_from, const string &sidechain_to, const string &sidechain_currency, int64_t sidechain_amount, const string &peerplays_from_name_or_id, const string &peerplays_to_name_or_id

sidechain_withdrawal_transaction(*son_name*, *block_num*, *sidechain*, *peerplays_uid*, *peerplays_transaction_id*, *peerplays_from*, *widthdraw_sidechain*, *widthdraw_address*, *widthdraw_currency*, *widthdraw_amount*)

unlock(*password*)

update_son(*account_name*, *url*, *sidechainPublicKeyListOfList*)

update_son_votes(*voting_account*, *sons_to_approve*, *sons_to_reject*, *sidechain*, *desired_number_of_sons*)

params: string voting_account, sons_to_approve, sons_to_reject, sidechain, desired_number_of_sons

update_witness_votes(*voting_account*, *witnesses_to_approve*, *witnesses_to_reject*, *desired_number_of_witnesses*)

params: voting_account, witnesses_to_approve, witnesses_to_reject, desired_number_of_witnesses,

vote_for_son(*voting_account*, *son*, *sidechain*, *approve*)

params: string voting_account, string son, string sidechain, bool approve, bool broadcast

vote_for_witness(*voting_account*, *witness*, *approve*)

params: string voting_account, string witness, bool approve, bool broadcast

peerplays.son.**WalletCall**(*method*, *params=[]*)

peerplays.sport module

class peerplays.sport.**Sport**(**args*, ***kwargs*)

Bases: *peerplays.blockchainobject.BlockchainObject*

Read data about a sport on the chain

Parameters

- **identifier** (*str*) – Identifier

- **blockchain_instance** (*peerplays*) – PeerPlays() instance to use when accesing a RPC

blockchain

blockchain_instance_class
alias of *peerplays.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

eventgroups

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. *peerplays.PeerPlays*

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.

peerplays
Alias for the specific blockchain

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()

static set_cache_store(*klass*, **args*, ***kwargs*)

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 = 20

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 peerplays.sport.**Sports**(**args*, ***kwargs*)
Bases: *peerplays.blockchainobject.BlockchainObjects*, *peerplays.instance.BlockchainInstance*

List of all available sports

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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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

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

refresh(**args*, ***kargs*)
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*.

static set_cache_store(*klass*, **args*, ***kwargs*)

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*.

sports
DEPRECATED

store(*data*, *key=None*, **args*, ***kwargs*)
Cache the list

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

peerplays.storage module

peerplays.storage.**get_default_config_store**(**args*, ***kwargs*)

peerplays.storage.**get_default_key_store**(*config*, **args*, ***kwargs*)

peerplays.transactionbuilder module

class peerplays.transactionbuilder.**ProposalBuilder**(**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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 ()

peerplays
Alias for the specific blockchain

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 peerplays.transactionbuilder.**TransactionBuilder**(**args*, ***kwargs*)

Bases: *peerplays.instance.BlockchainInstance*, *peerplays.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.

blockchain

blockchain_instance_class

alias of *peerplays.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. *peerplays.PeerPlays*

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()

peerplays
Alias for the specific blockchain

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

peerplays.utils module

peerplays.utils.**dList2Dict**(*l*)

peerplays.utils.**dict2dList**(*l*)

peerplays.utils.**map2dict**(*darray*)
Reformat a list of maps to a dictionary

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

peerplays.wallet module

class peerplays.wallet.**Wallet**(**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.wallet.Wallet*

addPrivateKey(*wif*)
Add a private key to the wallet database

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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*)

peerplays.witness module

class peerplays.witness.**Witness**(**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.witness.Witness*

Read data about a witness in the chain

Parameters

- **account_name** (*str*) – Name of the witness
- **blockchain_instance** (*peerplays*) – peerplays() instance to use when accesing a RPC

account

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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.

peerplays
Alias for the specific blockchain

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()

static set_cache_store(*klass*, **args*, ***kwargs*)

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 peerplays.witness.**Witnesses**(**args*, ***kwargs*)
Bases: *peerplays.instance.BlockchainInstance*, *peerplays.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** (*peerplays*) – peerplays() instance to use when accesing a RPC

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

blockchain

blockchain_instance_class
alias of *peerplays.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. *peerplays.PeerPlays*

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

peerplays
Alias for the specific blockchain

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*.

static set_cache_store(*klass*, **args*, ***kwargs*)

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 peerplaysbase

CHAPTER 6

Tutorials

6.1 Tutorials

6.1.1 Building PeerPlays Node

Downloading the sources

The sources can be downloaded from:

```
https://github.com/peerplays-network/peerplays
```

Dependencies

Development Toolkit

The following dependencies were necessary for a clean install of Ubuntu 16.10:

```
sudo apt-get update
sudo apt-get install gcc-5 g++-5 gcc g++ cmake make \
                     libbz2-dev libdb++-dev libdb-dev \
                     libssl-dev openssl libreadline-dev \
                     autotools-dev build-essential \
                     g++ libbz2-dev libicu-dev python-dev \
                     autoconf libtool git
```

Boost 1.60

You need to download the Boost tarball for Boost 1.60.0.

```
export BOOST_ROOT=$HOME/opt/boost_1.60.0
wget -c 'http://sourceforge.net/projects/boost/files/boost/1.60.0/boost_1.60.0.tar.
↪bz2/download'\
     -O boost_1.60.0.tar.bz2
tar xjf boost_1.60.0.tar.bz2
cd boost_1.60.0/
./bootstrap.sh "--prefix=$BOOST_ROOT"
./b2 install
```

Building PeerPlays

After downloading the PeerPlays sources we can run `cmake` for configuration and compile with `make`:

```
cd peerplays
export CC=gcc-5 CXX=g++-5
cmake -DBOOST_ROOT="$BOOST_ROOT" -DCMAKE_BUILD_TYPE=Debug .
make
```

Note that the environmental variable `$BOOST_ROOT` should point to your install directory of boost if you have installed it manually (see first line in the previous example)

Binaries

After compilation, the binaries are located in:

```
./programs/witness_node
./programs/cli_wallet
./programs/delayed_node
```

6.1.2 Howto Interface your Exchange with PeerPlays

This Howto serves as an introduction for exchanges that want to interface with PeerPlays to allow trading of assets from the PeerPlays network.

We here start by introducing the overall concept of trusted node setup, having different APIs that reply in JSON and describe the structure of the received information (blocks etc).

Afterwards, we will go into more detail w.r.t. to the python-peerplays library that helps you deal with the blockchain and can be seen as a full-featured wallet (to replace the cli-wallet).

Trusted Network and Client Configuration

Introduction

Similar to other crypto currencies, it is recommended to wait for several confirmations of a transcation. Even though the consensus scheme of Graphene is alot more secure than regular proof-of-work or other proof-of-stake schemes, we still support exchanges that require more confirmations for deposits.

We provide a so called *delayed* full node which accepts two additional parameters for the configuration besides those already available with the standard daemon.

- *trusted-node* RPC endpoint of a trusted validating node (required)

The trusted-node is a regular full node directly connected to the P2P network that works as a proxy. The delay between the trusted node and the delayed node is chosen automatically in a way that ensures that blocks that are available in the delayed node are guarenteed to be **irreversible**. Thus, the delayed full node will be behind the real blockchain by a few seconds up to only a few minutes.

Note: **Irrversibility**: On DPOS chains, blocks are irreversible if it has been approved/confirmed by at least 2/3 of all block validators (i.e. witnesses)

Overview of the Setup

In the following, we will setup and use the following network::

```
P2P network <-> Trusted Full Node <-> Delayed Full Node <-> API
```

- P2P network: The PeerPlays client uses a peer-to-peer network to connect and broadcasts transactions there. A block producing full node will eventually catch your transcaction and validate it by adding it into a new block.
- Trusted Full Node: We will use a Full node to connect to the network directly. We call it *trusted* since it is supposed to be under our control.
- Delayed Full Node: The delayed full node node will provide us with a delayed and several times confirmed and verified blockchain. Even though DPOS is more resistant against forks than most other blockchain consensus schemes, we delay the blockchain here to reduces the risk of forks even more. In the end, the delayed full node is supposed to never enter an invalid fork.
- API: Since we have a delayed full node that we can fully trust, we will interface with this node to query the blockchain and receive notifications from it once balance changes.

The delayed full node should be in the same *local* network as the trusted full node, however only the trusted full node requires public internet access. Hence we will work with the following IPs:

- **Trusted Full Node:**
  - extern: *internet access*
  - intern: *192.168.0.100*
- **Delayed Full Node:**
  - extern: *no* internet access required
  - intern: *192.168.0.101*

Let's go into more detail on how to set these up.

Trusted Full Node

For the trusted full node, the default settings can be used. Later, we will need to open the RPC port and listen to an IP address to connect the delayed full node to:

```
./programs/witness_node/witness_node --rpc-endpoint="192.168.0.100:8090"
```

Note: A *witness* node is identical to a full node if no authorized block-signing private key is provided.

Delayed Full Node

The delayed full node will need the IP address and port of the p2p-endpoint from the trusted full node and the number of blocks that should be delayed. We also need to open the RPC/Websocket port (to the local network!) so that we can interface using RPC-JSON calls.

For our example and for 10 blocks delayed (i.e. 30 seconds for 3 second block intervals), we need::

```
./programs/delayed_node/delayed_node --trusted-node="192.168.0.100:8090" --rpc-
↪endpoint="192.168.0.101:8090"
```

We can now connect via RPC:

- *192.168.0.100:8090* : The trusted full node exposed to the internet
- *192.168.0.101:8090* : The delayed full node not exposed to the internet

Note: For security reasons, an exchange should only interface with the delayed full node.

For obvious reasons, the trusted full node is should be running before attempting to start the delayed full node.

Remote Procedure Calls

Prerequisits

This page assumes that you either have a full node or a wallet running and listening to port `8090`, locally.

Note: The set of available commands depends on application you connect to.

Call Format

In Graphene, RPC calls are state-less and accessible via regular JSON formated RPC-HTTP-calls. The correct structure of the JSON call is

```
{
 "jsonrpc": "2.0",
 "id": 1
 "method": "get_accounts",
 "params": [["1.2.0", "1.2.1"]],
}
```

The `get_accounts` call is available in the Full Node's `database` API and takes only one argument which is an array of account ids (here: `["1.2.0", "1.2.1"]`).

Example Call with *curl*

Such as call can be submitted via `curl`:

```
curl --data '{"jsonrpc":"2.0","method":"call", "params":[0, "get_accounts", [["1.2.0",
↪ "1.2.1"]]],"id":0}' https://ppy-node.bitshares.eu
```

Successful Calls

The API will return a properly JSON formated response carrying the same `id` as the request to distinguish subsequent calls.

```
{
 "id":1,
 "result":  ..data..
}
```

Errors

In case of an error, the resulting answer will carry an `error` attribute and a detailed description:

```
{
  "id": 0
  "error": {
    "data": {
      "code": error-code,
      "name": " .. name of exception .."
      "message": " .. message of exception ..",
      "stack": [ .. stack trace .. ],
    },
    "code": 1,
  },
}
```

Remarks

Wallet specific commands, such as `transfer` and market orders, are only available if connecting to `cli_wallet` because only the wallet has the private keys and signing capabilities and some calls will only execute of the wallet is unlocked.

The full node offers a set of API(s), of which only the `database` calls are avaiable via RPC. Calls that are restricted by default (i.e. `network_node_api`) or have been restricted by configuration are not accessible via RPC because a statefull protocol (websocket) is required for login.

Interfacing via RPC and Websockets

Overview

APIs are separated into two categories, namely

- the **Blockchain API** which is used to query blockchain data (account, assets, trading history, etc.) and
- the **CLI Wallet API** which has your private keys loaded and is required when interacting with the blockchain with new transactions.

Blockchain API

The blockchain API (as provided by the `witness_node` application), allows to read the blockchain.

```
from peerplaysapi.node import PeerPlaysNodeRPC
ppy = PeerPlaysNodeRPC("wss://hostname")
print(ppy.get_account_by_name("init0"))
print(ppy.get_block(1))
```

Note: It is important to understand that the blockchain API does not know about private keys, and cannot sign transactions for you. All it does is validate and broadcast transactions to the P2P network.

CLI Wallet API

The cli-wallet api, as provided by the `cli_wallet` binary, allows to **create and sign transactions** and broadcast them.

```
from peerplaysapi.wallet import PeerPlaysWalletRPC
rpc = PeerPlaysWalletRPC("localhost", 8090)
print(rpc.info())
```

Howto Monitor the blockchain for certain operations

Block Structure

A block takes the following form:

```
{'extensions': [],
 'previous': '000583428a021b14c02f0faaff12a4c686e475e3',
 'timestamp': '2017-04-21T08:38:35',
 'transaction_merkle_root': '328be3287f89aa4d21c69cb617c4fcc372465493',
 'transactions': [{'expiration': '2017-04-21T08:39:03',
                   'extensions': [],
                   'operation_results': [[0, {}]],
                   'operations': [
                       [0,
                           {'amount': {'amount': 100000,
                                       'asset_id': '1.3.0'},
                            'extensions': [],
                            'fee': {'amount': 2089843,
                                    'asset_id': '1.3.0'},
                            'from': '1.2.18',
                            'memo': {'from':
↪'PPY1894jUspGi6fZwnUmaeCPDZpke6m4T9bHtKrd966M7qYz665xjr',
                                     'message': '5d09c06c4794f9bcdef9d269774209be',
                                     'nonce': '7364013452905740719',
                                     'to':
↪'PPY16MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV'},
                            'to': '1.2.6'}]
                   ],
                   'ref_block_num': 33602,
                   'ref_block_prefix': 337314442,
                   'signatures': ['1f3755deaa7f9........']}],
 'witness': '1.6.4',
 'witness_signature': '2052571f091c4542...........'}
```

Please note that a block can **carry multiple transactions** while each transaction **carries multiple operations**. Each operation could be a **transfer**, or any other type of operation from a list of available operations. Technically, an operation could be seen as a smart contract that comes with operation-specific side-information and results in some changes in the blockchain database.

In the example above, the operation type is identified by the `0`, which makes it a `transfer` and the structure afterwards carries the transfer-specific side information, e.g. `from`, `to` accounts, `fee` aswell as the `memo`.

Polling Approach

Blocks can be polled with as little code as this:

```
from peerplays.blockchain import Blockchain
chain = Blockchain()
for block in chain.blocks(start=START_BLOCK):
    print(block)
```

Note: `chain.blocks()` is a blocking call that will wait for new blocks and yield them to the for loop when they arrive.

Alternatively, one can construct a loop that only yields the operations on the blockchain and does not show the block structure:

```
from peerplays.blockchain import Blockchain
chain = Blockchain()
for op in chain.ops(start=START_BLOCK):  # Note the `ops`
    print(op)
```

If you are only interested in transfers, you may want to use this instead:

```
from peerplays.blockchain import Blockchain
chain = Blockchain()
for transfer in chain.stream(opNames=["transfer"], start=START_BLOCK):  # Note the␣
↪`ops`
   print(transfer)
```

Warning: By default, the `Blockchain()` instance will only look at **irrversible** blocks, this means that blocks are only considered if they are approved/signed by a majority of the witnesses and this lacks behind the head block by a short period of time (in the seconds to low minutes).

Notification Approach

under construction

Decoding the Memo

In Peerplays, memos are usually encrypted using a distinct memo key. That way, exposing the memo private key will only expose transaction memos (for that key) and not compromise any funds. It is thus safe to store the memo private key in 3rd party services and scripts.

Obtaining memo wif key from cli_wallet

The memo public key can be obtained from the cli_wallet account settings or via command line::

```
get_account myaccount
```

in the cli wallet. The corresponding private key can be obtain from::

```
get_private_key <pubkey>
```

Note that the latter command exposes all private keys in clear-text wif.

That private key can be added to the pypeerplays wallet with:

```
from peerplays import PeerPlays
ppy = PeerPlays()
# Create a new wallet if not yet exist
ppy.wallet.create("wallet-decrypt-password")
ppy.wallet.unlock("wallet-decrypt-password")
ppy.wallet.addPrivateKey("5xxxxxxxxxxx")
```

Decoding the memo

The memo is encoded with a DH-shared secret key. We don't want to go into too much detail here, but a simple python module can help you here:

The encrypted memo can be decoded with:

```
from peerplays.memo import Memo
transfer_operation = {
    'amount': {'amount': 100000, 'asset_id': '1.3.0'},
    'extensions': [],
    'fee': {'amount': 2089843, 'asset_id': '1.3.0'},
    'from': '1.2.18',
    'memo': {'from': 'PPY1894jUspGi6fZwnUmaeCPDZpke6m4T9bHtKrd966M7qYz665xjr',
             'message': '5d09c06c4794f9bcdef9d269774209be',
             'nonce': 7364013452905740719,
             'to': 'PPY16MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV'},
    'to': '1.2.6'}
memo = Memo(
    transfer_operation["from"],
    transfer_operation["to"],
)
memo.peerplays.wallet.unlock("wallet-decrypt-password")
print(memo.decrypt(transfer_operation["memo"]))
```

Alternatively, the 'history' command on the *cli-wallet* API, exposes the decrypted memo aswell.

6.1.3 Setup a witness and block producing node

After *having setup a node*, we can setup a witness and block producing node. We will need:

- A compiled `witness_node`
- A compiled `cli_wallet`
- A registered account

- The active private key to that account
- Some little funds to pay for witness registration in your account

Lunching the cli_wallet

We first need to launch the cli_wallet and setup a local wallet with it::

```
./programs/cli_wallet/cli_wallet --server-rpc-endpoint wss://node-to-some-public-api-
↪node
```

First thing to do is setting up a password for the newly created wallet prior to importing any private keys::

```
>>> set_password <password>
null
>>> unlock <password>
null
>>>
```

Basic Account Management

We can import your account with:

```
>>> import_key <accountname> <active wif key>
true
>>> list_my_accounts
[{
"id": "1.2.15",
...
"name": <accountname>,
...
]
>>> list_account_balances <accountname>
XXXXXXX PPY
```

Registering a Witness

To become a witness and be able to produce blocks, you first need to create a witness object that can be voted in.

We create a new witness by issuing::

```
>>> create_witness <accountname> "http://<url-to-proposal>" true
{
  "ref_block_num": 139,
  "ref_block_prefix": 3692461913,
  "relative_expiration": 3,
  "operations": [[
  21,{
    "fee": {
      "amount": 0,
      "asset_id": "1.3.0"
    },
    "witness_account": "1.2.16",
    "url": "url-to-proposal",
```

(continues on next page)

(continued from previous page)

```
    "block_signing_key": "<PUBLIC KEY>",
    "initial_secret": "0000000000000000000000000000000000000000000000000000000000000000"
  }
]
  ],
  "signatures": [

↪"1f2ad5597af2ac4bf7a50f1eef2db49c9c0f7616718776624c2c09a2dd72a0c53a26e8c2bc928f783624c4632924330fc0
↪"
  ]
}
```

The cli_wallet will create a new public key for signing `<PUBLIC KEY>`. We now need to obtain the private key for that::

```
get_private_key <PUBLIC KEY>
```

Configuration of the Witness Node

Get the witness object using:

```
get_witness <witness-account>
```

and take note of two things. The `id` is displayed in `get_global_properties` when the witness is voted in, and we will need it on the `witness_node` command line to produce blocks. We'll also need the public `signing_key` so we can look up the correspoinding private key.

```
>>> get_witness <accountname>
{
  [...]
  "id": "1.6.10",
  "signing_key": "GPH7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8",
  [...]
}
```

The `id` and the `signing_key` are the two important parameters, here. Let's get the private key for that signing key with::

```
get_private_key <PUBLIC KEY>
```

Now we need to start the witness, so shut down the wallet (ctrl-d), and shut down the witness (ctrl-c). Re-launch the witness, now mentioning the new witness 1.6.10 and its keypair::

```
./witness_node --rpc-endpoint=127.0.0.1:8090 \
               --witness-id '"1.6.10"' \
               --private-key '["GPH7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8
↪", "5JGi7DM7J8fSTizZ4D9roNgd8dUc5pirUe9taxYCUUsnvQ4zCaQ"]'
```

Alternatively, you can also add this line into yout config.ini::

```
witness-id = "1.6.10"
private-key = ["GPH7vQ7GmRSJfDHxKdBmWMeDMFENpmHWKn99J457BNApiX1T5TNM8",
↪"5JGi7DM7J8fSTizZ4D9roNgd8dUc5pirUe9taxYCUUsnvQ4zCaQ"]
```

Note: Make sure to use YOUR public/private keys instead of the once given above!

Verifying Block Production

If you monitor the output of the *witness_node*, you should see it generate blocks signed by your witness::

```
Witness 1.6.10 production slot has arrived; generating a block now...
Generated block #367 with timestamp 2015-07-05T20:46:30 at time 2015-07-05T20:46:30
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

A

B

C

D

E

F

G

H

I

P

R

S

T

U

V

W