# Bitcart SDK

***Release 0.1***

**Nov 09, 2019**

# Contents:

**1 Installing Bitcart SDK** **1**

**2 API Reference** **3**
2.1 Coin class . . . . . . . . . . 3
2.2 Implemented coins . . . . . . . . . . 4

**Index** **15**

# CHAPTER 1

# Installing Bitcart SDK

Simply run

```
pip install bitcart
```

to install the library.

Or, run

```
pip install bitcart-async
```

To install async version of the library.

You can't install both.

Async version of the library has the same API, but is intended to be used in asyncio application. All is the same, just use async/await. For more details, check async branch README

But to initialize bitcoin instance you will need `rpc_url`, `rpc_login` and `rpc_password` (not required, defaults work with default ports and authentification). For that you'll need bitcart daemon, so:

```
git clone https://github.com/MrNaif2018/bitcart
cd bitcart
pip install -r requirements/base.txt
pip install -r requirements/daemons/btc.txt
```

Everywhere here `coin_name` refers to coin you're going to run or use, `COIN_NAME` is the same name but in caps. For example if you run bitcoin, `coin_name=btc, COIN_NAME=BTC`, for litecoin `coin_name=ltc, COIN_NAME=LTC`.

Run `pip install -r requirements/daemons/coin_name.txt` to install requirements for daemon of `coin_name`.

This will clone main bitcart repo and install dependencies, we recommend using virtualenv for consistency.(some daemons conflict one with another, so using one virtualenv per daemon is fine).

To run daemon, just start it:

```
python daemons/btc.py
```

Or, to run it in background(linux only)

```
python daemons/btc.py &
```

Note, to run a few daemons, use `python daemons/coin_name.py` for each `coin_name`.

Default user is electrum and password is electrumz, it runs on http://localhost:5000. To run daemon in testnet, set `COIN_NAME_TESTNET` variable to true. By default, if coin supports it, lightning network is enabled. To disable it, set `COIN_NAME_LIGHTNING` to false. For each daemon port is different. General scheme to get your daemon url is http://localhost:port Where port is the port your daemon uses. You can change port and host by using `COIN_NAME_HOST` and `COIN_NAME_PORT` env variables. Default ports are starting from 5000 and increase for each daemon by 1 (in order how they were added to bitcart). Refer to main docs for ports information. Bitcoin port is 5000, litecoin is 5001, etc. So, to initialize your bitcart instance right now, import it and use those settings:

```
from bitcart.coins.btc import BTC
btc = BTC("http://localhost:5000", xpub="your x/y/zpub or x/y/zprv",
        rpc_user="electrum", rpc_pass="electrumz")
```

All the variables are actually optional, so you can just do `btc = BTC()` and use it, but without a wallet. To use a wallet, pass xpub like so: `btc = BTC(xpub="your x/y/zpub or x/y/zprv or electrum seed")` Xpub, xprv or electrum seed is the thing that represents your wallet. You can get it from your wallet provider, or, for testing or not, from here.

You can configure default user and password in `conf/.env` file of cloned bitcart repo, like so:

```
COIN_NAME_USER=myuser
COIN_NAME_PASS=mypassword
```

After that you can freely use bitcart methods, refer to *API docs* for more information.

# CHAPTER 2

# API Reference

## 2.1 Coin class

**class** bitcart.coin.**Coin**

Bases: object

Coins should reimplement some methods, and initialize coin-specific info. Required information is: coin_name str friendly_name str providers list For more info see the docs.

**balance**() → dict

Get balance of wallet

Example:

```
>>> self.balance()
{"confirmed": "0.00005", "unconfirmed": 0, "unmatured": 0}
```

**Raises** NotImplementedError – Implement in your subclass

**Returns** *dict* – It should return dict of balance statuses

**coin_name = 'Base'**

**friendly_name = 'Base'**

**get_address**(*address: str*) → list

Get address history

This method should return list of transaction informations for specified address

Example:

```
>>> c.get_address("31smpLFzLnza6k8tJbVpxXiatGjiEQDmzc")
[{'tx_hash': '7854bdf4c4e27276ecc1fb8d666d6799a248f5e81bdd58b16432d1ddd1d4c332
↪', 'height': 581878, 'tx': {'partial': False,...
```

**Parameters** **address** (*str*) – address to get transactions for

**Raises** NotImplementedError – Override this method in subclass

**Returns** *list* – List of transactions

**get_tx** (*tx: str*) → dict

Get transaction information

Given tx hash of transaction, return full information as dictionary

Example:

```
>>> c.get_tx("54604b116b28124e31d2d20bbd4561e6f8398dca4b892080bffc8c87c27762ba
↪")
{'partial': False, 'version': 2, 'segwit_ser': True, 'inputs': [{'prevout_hash
↪': 'xxxx',...
```

**Parameters** **tx** (*str*) – tx_hash

**Raises** NotImplementedError – Implement in your subclass

**Returns** *dict* – transaction info

**help** () → list

Get help

Returns a list of all available RPC methods

**Raises** NotImplementedError – Implement in your subclass

**Returns** *list* – RPC methods list

**providers = []**

## 2.2 Implemented coins

### 2.2.1 BTC

BTC class supports lightning out of the box. For lightning methods to work, it must be enabled from the daemon (enabled by default and edited by BTC_LIGHTNING environment variable). If lightning is disabled, LightningDisabledError is raised when calling lightning methods.

**class** bitcart.coins.btc.**BTC** (*rpc_url: Optional[str] = None*, *rpc_user: Optional[str] = None*, *rpc_pass: Optional[str] = None*, *xpub: Optional[str] = None*, *session: Optional[requests.Session] = None*)

Bases: *bitcart.coin.Coin*

**ALLOWED_EVENTS = ['new_block', 'new_transaction', 'new_payment']**

**RPC_PASS = 'electrumz'**

**RPC_URL = 'http://localhost:5000'**

**RPC_USER = 'electrum'**

**add_event_handler** (*events: Union[Iterable[str], str]*, *func: Callable*) → None

Add event handler to handle event(s) provided

**Parameters**

- **self** (BTC) – self
- **events** (*Union[Iterable[str], str]*) – event or events
- **func** (*Callable*) – function to handle those

**Returns** *None* – None

**addinvoice** (*amount: Union[int, float], message: Optional[str] = ''*) → str

Create lightning invoice

Create lightning invoice and return bolt invoice id

Example:

```
>>> a.addinvoice(0.5)
'lnbc500m1pwnt87fpp5d60sykcjd2swk72t3g0njwmdytfe4fu65fz5v...'
```

**Parameters**

- **self** (BTC) – self
- **amount** (*Union[int,float]*) – invoice amount
- **message** (*Optional[str], optional*) – Invoice message. Defaults to "".

**Returns** *str* – bolt invoice id

**addrequest** (*amount: Union[int, float], description: str = '', expire: Union[int, float] = 15*) → dict

Add invoice

Create an invoice and request amount in BTC, it will expire by parameter provided. If expire is None, it will last forever.

Example:

```
>>> c.addrequest(0.5, "My invoice", 20)
{'time': 1562762334, 'amount': 50000000, 'exp': 1200, 'address': 'xxx',...
```

**Parameters**

- **self** (BTC) – self
- **amount** (*Union[int, float]*) – amount to open invoice
- **description** (*str, optional*) – Description of invoice. Defaults to "".
- **expire** (*Union[int, float], optional*) – The time invoice will expire in. Defaults to 15.

**Returns** *dict* – Invoice data

**balance** () → dict

Get balance of wallet

Example:

```
>>> self.balance()
{"confirmed": "0.00005", "unconfirmed": 0, "unmatured": 0}
```

**Raises** NotImplementedError – Implement in your subclass

**Returns** *dict* – It should return dict of balance statuses

**close_channel** (*channel_id: str*, *force: bool = False*) → str

Close lightning channel

Close channel by channel_id got from open_channel, returns transaction id

**Parameters**

- **self** (BTC) – self
- **channel_id** (*str*) – channel_id from open_channel
- **force** (*bool*) – Create new address beyond gap limit, if no more addresses are available.

**Returns** *str* – tx_id of closed channel

**coin_name = 'BTC'**

**configure_webhook** (*autoconfigure: bool = True*) → None

**connect** (*connection_string: str*) → bool

Connect to lightning node

connection string must respect format pubkey@ipaddress

**Parameters** **connection_string** (*str*) – connection string

**Returns** *bool* – True on success, False otherwise

**friendly_name = 'Bitcoin'**

**get_address** (*address: str*) → list

Get address history

This method should return list of transaction informations for specified address

Example:

```
>>> c.get_address("31smpLFzLnza6k8tJbVpxXiatGjiEQDmzc")
[{'tx_hash': '7854bdf4c4e27276ecc1fb8d666d6799a248f5e81bdd58b16432d1ddd1d4c332
↪', 'height': 581878, 'tx': {'partial': False,...
```

**Parameters** **address** (*str*) – address to get transactions for

**Raises** NotImplementedError – Override this method in subclass

**Returns** *list* – List of transactions

**get_config** (*key: str*, *default: Any = None*) → Any

Get config key

If the key doesn't exist, default value is returned. Keys are stored in electrum's config file, check *bitcart.coins.btc.BTC.set_config()* doc for details.

Example:

```
>>> c.get_config("x")
5
```

**Parameters**

- **self** (BTC) – self
- **key** (*str*) – key to get

- **default** (*Any, optional*) – The value to default to when key doesn't exist. Defaults to None.

**Returns** *Any* – value of the key or default value provided

**get_tx** (*tx: str*) → dict

Get transaction information

Given tx hash of transaction, return full information as dictionary

Example:

```
>>> c.get_tx("54604b116b28124e31d2d20bbd4561e6f8398dca4b892080bffc8c87c27762ba
↪")
{'partial': False, 'version': 2, 'segwit_ser': True, 'inputs': [{'prevout_hash
↪': 'xxxx',...
```

**Parameters** **tx** (*str*) – tx_hash

**Raises** NotImplementedError – Implement in your subclass

**Returns** *dict* – transaction info

**getrequest** (*address: str*) → dict

Get invoice info

Get invoice information by address got from addrequest

Example:

```
>>> c.getrequest("1A6jnc6xQwmhsChNLcyKAQNWPcWsVYqCqJ")
{'time': 1562762334, 'amount': 50000000, 'exp': 1200, 'address':
↪'1A6jnc6xQwmhsChNLcyKAQNWPcWsVYqCqJ',...
```

**Parameters**

- **self** (BTC) – self
- **address** (*str*) – address of invoice

**Returns** *dict* – Invoice data

**handle_webhook_async** (*request: web.Request*) → web.Response

**handle_webhook_sync** () → dict

**help** () → list

Get help

Returns a list of all available RPC methods

**Raises** NotImplementedError – Implement in your subclass

**Returns** *list* – RPC methods list

**history** () → dict

Get transaction history of wallet

Example:

```
>>> c.history()
{'summary': {'end_balance': '0.', 'end_date': None, 'from_height': None,
↪'incoming': '0.00185511',...
```

(continues on next page)

(continued from previous page)

```
```

**Parameters** **self** (BTC) – self

**Returns** *dict* – dictionary with some data, where key transactions is list of transactions

**list_channels**() → list

List all channels ever opened

Possible channel statuses: OPENING, OPEN, CLOSED, DISCONNECTED

Example:

```
>>> a.server.list_channels()
[{'local_htlcs': {'adds': {}, 'locked_in': {}, 'settles': {}, 'fails': {}},
↪'remote_htlcs': ...
```

**Returns** *list* – list of channels

**list_fiat**() → Iterable[str]

List of all available fiat currencies to get price for

This list is list of only valid currencies that could be passed to rate() function

Example:

```
>>> c.list_fiat()
['AED', 'ARS', 'AUD', 'BCH', 'BDT', 'BHD', 'BMD', 'BNB', 'BRL', 'BTC', ...]
```

**Parameters** **self** (BTC) – self

**Returns** *Iterable[str]* – list of available fiat currencies

**lnpay**(*invoice: str*) → bool

Pay lightning invoice

Returns True on success, False otherwise

**Parameters** **invoice** (str) – invoice to pay

**Returns** *bool* – success or not

**node_id**

Get node id

Electrum's lightning implementation itself is a lightning node, that way you can get a super light node, this method returns it's id

Example:

```
>>> a.node_id
'030ff29580149a366bdddf148713fa808f0f4b934dccd5f7820f3d613e03c86e55'
```

**Returns** *str* – id of your node

**on**(*events: Union[Iterable[str], str]*) → Callable

Register on event

Register callback function to be run when event is emmited

All available events are accessable as:

```
>>> btc.ALLOWED_EVENTS
['new_block', 'new_transaction']
```

Function signature must be

```
def handler(event, **kwargs):
```

kwargs sent differ from event to event, as for now new_block event sends height kwarg as new block height new_transaction event sends tx kwarg as tx_hash of new transaction

**Parameters**

- **self** (BTC) – self
- **events** (*Union[Iterable[str], str]*) – event name or list of events for function to be run on

**Returns** *Callable* – It is a decorator

**open_channel** (*node_id: str, amount: Union[int, float]*) → str

Open lightning channel

Open channel with node, returns string of format txid:output_index

**Parameters**

- **self** (BTC) – self
- **node_id** (*str*) – id of node to open channel with
- **amount** (*Union[int, float]*) – amount to open channel

**Returns** *str* – string of format txid:output_index

**pay_to** (*address: str, amount: float, fee: Union[float, Callable, None] = None, feerate: Optional[float] = None, broadcast: bool = True*) → Union[dict, str]

Pay to address

This function creates a transaction, your wallet must have sufficent balance and address must exist.

Examples:

```
>>> btc.pay_to("mkHS9ne12qx9pS9VojpwU5xtRd4T7X7ZUt", 0.001)
'608d9af34032868fd2849723a4de9ccd874a51544a7fba879a18c847e37e577b'
```

```
>>> btc.pay_to("mkHS9ne12qx9pS9VojpwU5xtRd4T7X7ZUt",0.001, feerate=1)
'23d0aec06f6ea6100ba9c6ce8a1fa5d333a6c1d39a780b5fadc4b2836d71b66f'
```

```
>>> btc.pay_to("mkHS9ne12qx9pS9VojpwU5xtRd4T7X7ZUt", 0.001, broadcast=False)
{'hex': '02000000026.....', 'complete': True, 'final': False, 'name': None,
↪'csv_delay': 0, 'cltv_expiry': 0}
```

**Parameters**

- **self** (BTC) – self
- **address** (*str*) – address where to send BTC
- **amount** (*float*) – amount of bitcoins to send
- **fee** (*Optional[Union[float, Callable]], optional*) – Either a fixed fee, or a callable getting size and default fee as argument and returning fee. Defaults to None.

- **feerate** (*Optional[float], optional*) – A sat/byte feerate, can't be passed together with fee argument. Defaults to None.
- **broadcast** (*bool, optional*) – Whether to broadcast transaction to network. Defaults to True.

**Raises**

- `ValueError` – If address or amount is invalid or in other cases
- `TypeError` – if you have provided both fee and feerate

**Returns** *Union[dict, str]* – tx hash of ready transaction or raw transaction, depending on broadcast argument.

**pay_to_many**(*outputs: Iterable[Union[dict, tuple]], fee: Union[float, Callable, None] = None, feerate: Optional[float] = None, broadcast: bool = True*) → Union[dict, str]

Pay to multiple addresses(batch transaction)

This function creates a batch transaction, your wallet must have sufficent balance and addresses must exist. outputs parameter is either an iterable of `(address, amount)` tuples(or any iterables) or a dict with two keys: address and amount `{"address": "someaddress", "amount": 0.5}`

Examples:

```
>>> btc.pay_to_many([{"address":"mkHS9ne12qx9pS9VojpwU5xtRd4T7X7ZUt","amount
↪":0.001}, {"address":"mv4rnyY3Su5gjcDNzbMLKBQkBicCtHUtFB","amount":0.0001}])
'60fa120d9f868a7bd03d6bbd1e225923cab0ba7a3a6b961861053c90365ed40a'
```

```
>>> btc.pay_to_many([("mkHS9ne12qx9pS9VojpwU5xtRd4T7X7ZUt",0.001), (
↪"mv4rnyY3Su5gjcDNzbMLKBQkBicCtHUtFB",0.0001)])
'd80f14e20af2ceaa43a8b7e15402d420246d39e235d87874f929977fb0b1cab8'
```

```
>>> btc.pay_to_many((("mkHS9ne12qx9pS9VojpwU5xtRd4T7X7ZUt",0.001),(
↪"mkHS9ne12qx9pS9VojpwU5xtRd4T7X7ZUt",0.001)), feerate=1)
'0a6611876e04a6f2742eac02d4fac4c242dda154d85f0d547bbac1a33dbbbe34'
```

```
>>> btc.pay_to_many([("mkHS9ne12qx9pS9VojpwU5xtRd4T7X7ZUt",0.001), (
↪"mv4rnyY3Su5gjcDNzbMLKBQkBicCtHUtFB",0.0001)], broadcast=False)
{'hex': '0200000...', 'complete': True, 'final': False}
```

**Parameters**

- **self** (BTC) – self
- **outputs** (*Iterable[Union[dict, tuple]]*) – An iterable with dictionary or iterable as the item
- **fee** (*Optional[Union[float, Callable]], optional*) – Either a fixed fee, or a callable getting size and default fee as argument and returning fee. Defaults to None.
- **feerate** (*Optional[float], optional*) – A sat/byte feerate, can't be passed together with fee argument. Defaults to None.
- **broadcast** (*bool, optional*) – Whether to broadcast transaction to network. Defaults to True.

**Raises**

- `ValueError` – If address or amount is invalid or in other cases

- `TypeError` – if you have provided both fee and feerate

**Returns** *Union[dict, str]* – tx hash of ready transaction or raw transaction, depending on broadcast argument.

**poll_updates** (*timeout: Union[int, float] = 2*) → None

**poll_updates_sync** (*timeout: Union[int, float] = 2*) → None

Poll updates

Poll daemon for new transactions in wallet, this will block forever in while True loop checking for new transactions

Example can be found on main page of docs

**Parameters**

- **self** (`BTC`) – self
- **timeout** (*Union[int, float], optional*) – seconds to wait before requesting transactions again. Defaults to 2.

**Raises** `InvalidEventError` – If server sent invalid event name not matching ALLOWED_EVENTS

**Returns** *None* – This function runs forever

**process_updates** (*updates: Iterable[dict]*) → None

**providers = ['jsonrpcrequests']**

**rate** (*currency: str = 'USD', accurate: bool = False*) → Union[float, decimal.Decimal]

Get bitcoin price in selected fiat currency

It uses the same method as electrum wallet gets exchange rate-via different payment providers

Examples:

```
>>> c.rate()
9878.527
```

```
>>> c.rate("RUB")
757108.226
```

**Parameters**

- **self** (`BTC`) – self
- **currency** (*str, optional*) – Currency to get rate into. Defaults to "USD".
- **accurate** (*bool, optional*) – Whether to return values harder to work with(decimals) or not very accurate floats. Defaults to False.

**Returns** *Union[float, Decimal]* – price of 1 bitcoin in selected fiat currency

**set_config** (*key: str, value: Any*) → bool

Set config key to specified value

It sets the config value in electrum's config store, usually $HOME/.electrum/config

You can set any keys and values using this function(as long as JSON serializable), and some are used to configure underlying electrum daemon.

Example:

```
>>> c.set_config("x", 5)
True
```

**Parameters**

- **self** (BTC) – self
- **key** (*str*) – key to set
- **value** (*Any*) – value to set

**Returns** *bool* – True on success, False otherwise

**start_webhook** (*port: int = 6000*, ***kwargs*) → None

**validate_key** (*key: str*) → bool

Validate whether provided key is valid to restore a wallet

If the key is x/y/z pub/prv or electrum seed at the network daemon is running at, then it would be valid(True), else False

Examples:

```
>>> c.validate_key("test")
False
```

```
>>> c.validate_key("your awesome electrum seed")
True
```

```
>>> c.validate_key("x/y/z pub/prv here")
True
```

**Parameters**

- **self** (BTC) – self
- **key** (*str*) – key to check

**Returns** *bool* – Whether the key is valid or not

### 2.2.2 LTC

LTC class supports lightning out of the box. For lightning methods to work, it must be enabled from the daemon (enabled by default and edited by `LTC_LIGHTNING` environment variable). If lightning is disabled, `LightningDisabledError` is raised when calling lightning methods.

**class** bitcart.coins.ltc.**LTC** (*rpc_url: Optional[str] = None*, *rpc_user: Optional[str] = None*, *rpc_pass: Optional[str] = None*, *xpub: Optional[str] = None*, *session: Optional[requests.Session] = None*)

Bases: *bitcart.coins.btc.BTC*

**RPC_URL = 'http://localhost:5001'**

**coin_name = 'LTC'**

**friendly_name = 'Litecoin'**

### 2.2.3 GZRO

GZRO class supports lightning out of the box. For lightning methods to work, it must be enabled from the daemon (enabled by default and edited by `GZRO_LIGHTNING` environment variable). If lightning is disabled, `LightningDisabledError` is raised when calling lightning methods.

**class** `bitcart.coins.gzro.`**GZRO**(*rpc_url: Optional[str] = None*, *rpc_user: Optional[str] = None*, *rpc_pass: Optional[str] = None*, *xpub: Optional[str] = None*, *session: Optional[requests.Session] = None*)

Bases: *bitcart.coins.btc.BTC*

**RPC_URL = 'http://localhost:5002'**

**coin_name = 'GZRO'**

**friendly_name = 'GravityZero'**

Bitcart is a platform to simplify cryptocurrencies adaptation. This SDK is part of bitcart. Using this SDK you can easily connect to bitcart daemon and code scripts around it easily.

Behold, the power of Bitcart:

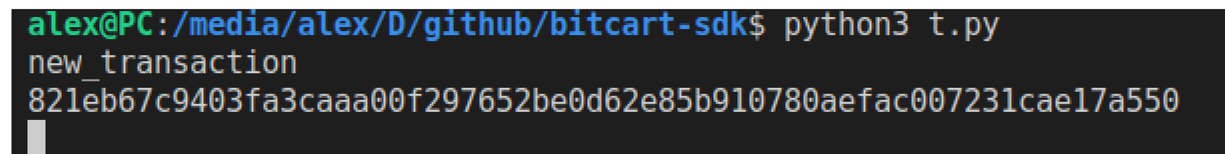
```
from bitcart.coins.btc import BTC

btc = BTC(xpub="your x/y/zpub or x/y/zprv")


@btc.on("new_transaction")
def callback_func(event, tx):
    print(event)
    print(tx)


btc.poll_updates()
```

This simple script will listen for any new transaction on your wallet's addresses and print information about them like so:




And if you add `print(btc.get_tx(tx))` it would print full information about every transaction, too!

To run this script, refer to *installation* section. For examples of usage, check examples directory in github repository.

Supported coins list( means lightning is supported):

- Bitcoin ()
- Litecoin ()
- Gravity Zero ()

# Index


## A

add_event_handler() (*bitcart.coins.btc.BTC method*), 4
addinvoice() (*bitcart.coins.btc.BTC method*), 5
addrequest() (*bitcart.coins.btc.BTC method*), 5
ALLOWED_EVENTS (*bitcart.coins.btc.BTC attribute*), 4

## B

balance() (*bitcart.coin.Coin method*), 3
balance() (*bitcart.coins.btc.BTC method*), 5
BTC (*class in bitcart.coins.btc*), 4

## C

close_channel() (*bitcart.coins.btc.BTC method*), 6
Coin (*class in bitcart.coin*), 3
coin_name (*bitcart.coin.Coin attribute*), 3
coin_name (*bitcart.coins.btc.BTC attribute*), 6
coin_name (*bitcart.coins.gzro.GZRO attribute*), 13
coin_name (*bitcart.coins.ltc.LTC attribute*), 12
configure_webhook() (*bitcart.coins.btc.BTC method*), 6
connect() (*bitcart.coins.btc.BTC method*), 6

## F

friendly_name (*bitcart.coin.Coin attribute*), 3
friendly_name (*bitcart.coins.btc.BTC attribute*), 6
friendly_name (*bitcart.coins.gzro.GZRO attribute*), 13
friendly_name (*bitcart.coins.ltc.LTC attribute*), 12

## G

get_address() (*bitcart.coin.Coin method*), 3
get_address() (*bitcart.coins.btc.BTC method*), 6
get_config() (*bitcart.coins.btc.BTC method*), 6
get_tx() (*bitcart.coin.Coin method*), 4
get_tx() (*bitcart.coins.btc.BTC method*), 7
getrequest() (*bitcart.coins.btc.BTC method*), 7
GZRO (*class in bitcart.coins.gzro*), 13

## H

handle_webhook_async() (*bitcart.coins.btc.BTC method*), 7
handle_webhook_sync() (*bitcart.coins.btc.BTC method*), 7
help() (*bitcart.coin.Coin method*), 4
help() (*bitcart.coins.btc.BTC method*), 7
history() (*bitcart.coins.btc.BTC method*), 7

## L

list_channels() (*bitcart.coins.btc.BTC method*), 8
list_fiat() (*bitcart.coins.btc.BTC method*), 8
lnpay() (*bitcart.coins.btc.BTC method*), 8
LTC (*class in bitcart.coins.ltc*), 12

## N

node_id (*bitcart.coins.btc.BTC attribute*), 8

## O

on() (*bitcart.coins.btc.BTC method*), 8
open_channel() (*bitcart.coins.btc.BTC method*), 9

## P

pay_to() (*bitcart.coins.btc.BTC method*), 9
pay_to_many() (*bitcart.coins.btc.BTC method*), 10
poll_updates() (*bitcart.coins.btc.BTC method*), 11
poll_updates_sync() (*bitcart.coins.btc.BTC method*), 11
process_updates() (*bitcart.coins.btc.BTC method*), 11
providers (*bitcart.coin.Coin attribute*), 4
providers (*bitcart.coins.btc.BTC attribute*), 11

## R

rate() (*bitcart.coins.btc.BTC method*), 11
RPC_PASS (*bitcart.coins.btc.BTC attribute*), 4
RPC_URL (*bitcart.coins.btc.BTC attribute*), 4
RPC_URL (*bitcart.coins.gzro.GZRO attribute*), 13
RPC_URL (*bitcart.coins.ltc.LTC attribute*), 12

RPC_USER (*bitcart.coins.btc.BTC attribute*), 4

## S

set_config() (*bitcart.coins.btc.BTC method*), 11
start_webhook() (*bitcart.coins.btc.BTC method*), 12

## V

validate_key() (*bitcart.coins.btc.BTC method*), 12