Bitcart SDK

Release 1.15.0.0

MrNaif2018

Mar 15, 2024

CONTENTS:

CHAPTER

ONE

INSTALLING BITCART SDK

Simply run

```
pip install bitcart
```

to install the library.

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/bitcart/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 other network than mainnet, set `COIN_NAME_NETWORK` variable to network name (testnet, regtest). 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 import BTC
btc = BTC(xpub="your (x/y/z)pub or (x/y/z)prv or electrum seed")
```

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

TWO

BITCART SDK EVENTS SYSTEM

2.1 Introduction

To be able to listen for incoming payments, transactions, new blocks or more, you should use events system.

In SDK, each event may have one handler.

One handler may handle multiple events at once

Using event system requires wallet.

Event handler signature is the following:

```
def handler(event, arg1, arg2):
    pass # process event
```

Where `event` is the only required argument everywhere.

Depending on the event being handled, it may provide additional arguments. All arguments must be named exactly as event is specifying.

If using event system from APIManager, then additional required argument `instance` is passed:

```
def handler(instance, event, arg1, arg2):
    pass # process event
```

Also, async handlers are supported too, like so:

```
async def handler(event, arg1, arg2):
    pass # process event, you can await here
```

2.2 Registering event handlers

To register an event handler, use `on` decorator:

```
@coin.on("event")
def handler(event, arg1, arg2):
    pass
```

You can also register a handler for multiple events, then you should mark all arguments, except for `event` and `instance` (if provided) optional, like so:

```
@coin.on(["event1", "event2"])
def handler(event, arg1=None, arg2=None, arg3=None):
    pass # event argument can be used to get what event is currently being processed
```

Or you can use `add_event_handler` method instead:

```
def handler(event, arg1, arg2):
    pass

coin.add_event_handler("event", handler)
```

They work identically.

2.3 Events list

2.3.1 new_block

Called when a new block is mined

Additional data:

- height (int): height of this block

2.3.2 new_transaction

Called when a new transaction on this wallet has appeared

Additional data:

- tx (str): tx hash of this transaction

2.3.3 new_payment

Called when status of payment request has changed. (See `get_request/add_request`)

Additional data:

- address (str): address related to this payment request
- status (int): new status code
- status_str (str): string version of new status code

2.4 Listening for updates

To receive updates, you should use one of the available event delivery methods: polling or websocket

2.4.1 Polling

Polling is good for quick testing, but not very good for production.

In this method SDK calls `get_updates` daemon method constantly, processing any new updates received.

To use it, run:

```
coin.poll_updates()
```

It will start an infinite loop.

2.4.2 Websocket

Websocket is a bit harder to set up sometimes, but works better.

Instead of constantly calling daemon method to get updates, daemon will send updates when they are available via an estabilished connection.

That way, you don't need to know your URL, you should only know daemon URL, and a channel between SDK and daemon will be set up.

To use it, run:

```
coin.start_websocket()
```

It will connect to your daemon's `/ws` endpoint, with auto-reconnecting in case of unexpected websocket close.

There may be unlimited number of websockets per wallet or not.

Under the hood, if using APIManager, daemon will send all it's updates to SDK, and SDK will filter only the one you need.

If using coin object, daemon will only send updates about this wallet.

2.4.3 Manual updates processing

If you need complete control over updates delivery, you can pass updates to coin's method directly:

```
coin.process_updates(updates_list)
```

Where `updates_list` is a list of dictionaries.

Each dictionary must contain event key, and additional keys for data required for this event.

2.5 Processing updates for multiple wallets/currencies

If you need to process updates for multiple wallets/currencies, take a look at *APIManager documentation*

CHAPTER

THREE

APIMANAGER

APIManager provides an easy-to-use interface to manage multiple wallets/currencies.

3.1 Setup

Create APIManager like so:

```
manager = APIManager({"BTC": ["xpub1", "xpub2"], "currency2": ["xpub1", "xpub3"]})
```

This will load all specified wallets to the manager.

3.2 Accessing wallets

You can access wallets in a manager like so:

```
manager.BTC.xpub1 # access wallet <=> BTC(xpub="xpub1")
# or
manager["BTC"]["xpub1"] # same wallet, but via dict-like interface
manager["currency2"]["xpub3"] # <=> currency2(xpub="xpub3")
manager.BTC[xpub].balance() # <=> BTC(xpub=xpub).balance()
```

3.3 Adding new wallets to existing manager

```
manager.add_wallet("BTC", "xpub3") # adds wallet to currency BTC with xpub="xpub3"
manager.add_wallets("currency2", ["xpub1", "xpub2"]) # batch add
```

3.4 Coin objects creation utilities

```
manager.load_wallet("currency", "xpub") # returns currency(xpub=xpub)
manager.load_wallets("currency", ["xpub1", "xpub2"]) # returns a dict of xpub-
↪currency(xpub=xpub)
```

3.5 Listening for updates on all wallets in a manager

You can register event handlers like you did before, on individual coin instances, or globally on manager object.

```
@manager.on("new_transaction")
def handler(instance, event, tx):
    pass # instance is coin instance currently processing the event
```

To start connection to websocket, run:

```
manager.start_websocket()
```

CHAPTER

FOUR

API REFERENCE

4.1 Implemented coins

4.1.1 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: str | None = None*, *rpc_user: str | None = None*, *rpc_pass: str | None = None*, *xpub: str | None = None*, *proxy: str | None = None*, *session: ClientSession | None = None*)

Bases: `Coin`, `EventDelivery`

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

BALANCE_ATTRS = ['confirmed', 'unconfirmed', 'unmatured', 'lightning']

EXPIRATION_KEY = 'expiry'

RPC_PASS = 'electrumz'

RPC_URL = 'http://localhost:5000'

RPC_USER = 'electrum'

add_invoice(*amount: int | str | Decimal*, *description: str = ''*, *expire: int | float = 15*) → dict

Create a lightning invoice

Create a lightning invoice and return invoice data with bolt invoice id All parameters are the same as in add_request

Example:

```
>>> a.add_invoice(0.5, "My invoice", 20)
{'time': 1562762334, 'amount': 50000000, 'exp': 1200, 'invoice': 'lnbc500m',...
```

Parameters

- **self** (BTC) – self
- **amount** (*AmountType*) – amount to open invoice
- **description** (*str, optional*) – Description of invoice. Defaults to “”.

- **expire** (*Union[int, float], optional*) – The time invoice will expire in. In minutes. Defaults to 15.

Returns
dict – Invoice data

add_request(*amount: int | str | Decimal | None = None*, *description: str = ''*, *expire: 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.add_request(0.5, "My invoice", 20)
{'time': 1562762334, 'amount': 50000000, 'exp': 1200, 'address': 'xxx',...
```

Parameters

- **self** (BTC) – self
- **amount** (*Optional[AmountType]*) – amount to open invoice. Defaults to None.
- **description** (*str, optional*) – Description of invoice. Defaults to "".
- **expire** (*Union[int, float], optional*) – The time invoice will expire in. In minutes. Defaults to 15.

Returns
dict – Invoice data

additional_xpub_fields: list[str] = []

balance() → dict

Get balance of wallet

Example:

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

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: str = 'BTC'

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: str = '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': ...
```

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

Returns
list – List of transactions

get_config(*key: str*) → Any

Get config key

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_invoice(*rhash: str*) → dict

Get lightning invoice info

Get lightning invoice information by rhash got from add_invoice

Example:

```
>>> c.get_invoice(
↪"e34d7fb4cda66e0760fc193496c302055d0fd960cfd982432355c8bfeecd5f33")
{'is_lightning': True, 'amount_BTC': Decimal('0.5'), 'timestamp': 1619273042,
↪'expiry': 900, ...
```

Parameters
rhash (*str*) – invoice rhash

Returns
dict – invoice data

get_request(*address: str*) → dict

Get invoice info

Get invoice information by address got from add_request

Example:

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

Parameters

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

Returns
dict – Invoice data

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

Returns
dict – transaction info

help() → list

Get help

Returns a list of all available RPC methods

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',...
```

Parameters
self (BTC) – self

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

is_eth_based = False

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_peers(*gossip: bool = False*) → list

Get a list of lightning peers

Parameters
gossip (*bool, optional*) – Whether to return peers of a gossip (one per node) or of a wallet. Defaults to False.

Returns
list – list of lightning peers

lnpay(*invoice: str*) → bool

Pay lightning invoice

Returns True on success, False otherwise

Parameters
invoice (*str*) – invoice to pay

Returns
bool – success or not

property node_id: Any

Return an attribute of instance, which is of type owner.

open_channel(*node_id: str*, *amount: int | str | Decimal*) → 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** (*AmountType*) – amount to open channel

Returns

str – string of format txid:output_index

pay_to(*address: str*, *amount: int | str | Decimal*, *fee: int | str | Decimal | Callable | None = None*, *feerate: int | str | Decimal | None = None*, *broadcast: bool = True*) → 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** (*AmountType*) – amount of bitcoins to send
- **fee** (*Optional[Union[AmountType, Callable]], optional*) – Either a fixed fee, or a callable getting size and default fee as argument and returning fee. Defaults to None.
- **feerate** (*Optional[AmountType], 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

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[dict | tuple]*, *fee: int | str | Decimal | Callable | None = None*, *feerate: int | str | Decimal | None = None*, *broadcast: bool = True*) → 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[AmountType, Callable]], optional*) – Either a fixed fee, or a callable getting size and default fee as argument and returning fee. Defaults to None.
- **feerate** (*Optional[AmountType], 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

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: int | float = 1*) → 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 1.

Returns

None – This function runs forever

process_updates(*updates: Iterable[dict]*, **args: Any*, *pass_instance: bool = False*, ***kwargs: Any*) → None

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

property spec: Any

Return an attribute of instance, which is of type owner.

start_websocket(*reconnect_callback: Callable | None = None*, *force_connect: bool = False*, *auto_reconnect: bool = True*) → None

Start a websocket connection to daemon

Parameters

- **reconnect_callback** (*Optional[Callable], optional*) – Callback to be called right after each succesful connection. Defaults to None.
- **force_connect** (*bool, optional*) – Whether to try reconnecting even on first failure (handshake) to daemon. Defaults to False.
- **auto_reconnect** (*bool, optional*) – Whether to enable auto-reconnecting on web-socket closing. Defaults to True.

validate_key(*key: str*, **args: Any*, ***kwargs: Any*) → 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

xpub_name: str = 'Xpub'

4.1.2 BCH

BCH supports Schnorr signatures, they are enabled out of the box

class bitcart.coins.bch.**BCH**(*rpc_url: str | None = None*, *rpc_user: str | None = None*, *rpc_pass: str | None = None*, *xpub: str | None = None*, *proxy: str | None = None*, *session: ClientSession | None = None*)

Bases: *BTC*

AMOUNT_FIELD = 'amount (BCH)'

EXPIRATION_KEY = 'expiration'

RPC_URL = 'http://localhost:5004'

coin_name: str = 'BCH'

event_handlers: dict[str, Callable]

friendly_name: str = 'Bitcoin Cash'

history() → dict

Get transaction history of wallet

Example:

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

Parameters

self (BTC) – self

Returns

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

property node_id: Any

Return an attribute of instance, which is of type owner.

server: RPCProxy

property spec: Any

Return an attribute of instance, which is of type owner.

xpub: str | None

4.1.3 XMR

XMR support is based on our custom daemon implementation which tries to follow electrum APIs as closely as possible

class bitcart.coins.xmr.**XMR**(*rpc_url: str | None = None*, *rpc_user: str | None = None*, *rpc_pass: str | None = None*, *xpub: str | None = None*, *proxy: str | None = None*, *session: ClientSession | None = None*)

Bases: *ETH*

RPC_URL = 'http://localhost:5011'

additional_xpub_fields: list[str] = ['address']

coin_name: str = 'XMR'

event_handlers: dict[str, Callable]

friendly_name: str = 'Monero'

get_address(**args: Any*, ***kwargs: Any*) → NoReturn

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': ...
```

Parameters

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

Returns

list – List of transactions

history() → dict

Get transaction history of wallet

Example:

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

Parameters

self (BTC) – self

Returns

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

property node_id: Any

Return an attribute of instance, which is of type owner.

pay_to_many(**args: Any*, ***kwargs: Any*) → NoReturn

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[AmountType, Callable]], optional*) – Either a fixed fee, or a callable getting size and default fee as argument and returning fee. Defaults to None.
- **feerate** (*Optional[AmountType], 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

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.

server: RPCProxy

property spec: Any

Return an attribute of instance, which is of type owner.

xpub: str | None

xpub_name: str = 'Secret viewkey'

4.1.4 ETH

ETH support is based on our custom daemon implementation which tries to follow electrum APIs as closely as possible

class bitcart.coins.eth.**ETH**(*rpc_url: str | None = None*, *rpc_user: str | None = None*, *rpc_pass: str | None = None*, *xpub: str | None = None*, *proxy: str | None = None*, *session: ClientSession | None = None*)

Bases: *BTC*

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

EXPIRATION_KEY = 'expiration'

RPC_URL = 'http://localhost:5002'

coin_name: str = 'ETH'

event_handlers: dict[str, Callable]

friendly_name: str = 'Ethereum'

get_address(**args: Any*, ***kwargs: Any*) → NoReturn

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': ...
```

Parameters

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

Returns

list – List of transactions

history() → dict

Get transaction history of wallet

Example:

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

Parameters

self (BTC) – self

Returns

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

is_eth_based = True

property node_id: Any

Return an attribute of instance, which is of type owner.

pay_to_many(**args: Any*, ***kwargs: Any*) → NoReturn

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[AmountType, Callable]], optional*) – Either a fixed fee, or a callable getting size and default fee as argument and returning fee. Defaults to None.
- **feerate** (*Optional[AmountType], 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

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.

server: RPCProxy

property spec: Any

Return an attribute of instance, which is of type owner.

xpub: str | None

xpub_name: str = 'Address'

4.1.5 BNB

BNB support is based on our custom daemon implementation which tries to follow electrum APIs as closely as possible

class bitcart.coins.bnb.**BNB**(*rpc_url: str | None = None*, *rpc_user: str | None = None*, *rpc_pass: str | None = None*, *xpub: str | None = None*, *proxy: str | None = None*, *session: ClientSession | None = None*)

Bases: *ETH*

RPC_URL = 'http://localhost:5006'

coin_name: str = 'BNB'

event_handlers: dict[str, Callable]

friendly_name: str = 'Binance Smart Chain'

property node_id: Any

Return an attribute of instance, which is of type owner.

server: RPCProxy

property spec: Any

Return an attribute of instance, which is of type owner.

xpub: str | None

4.1.6 SmartBCH

SmartBCH support is based on our custom daemon implementation which tries to follow electrum APIs as closely as possible

class bitcart.coins.sbch.**SBCH**(*rpc_url: str | None = None*, *rpc_user: str | None = None*, *rpc_pass: str | None = None*, *xpub: str | None = None*, *proxy: str | None = None*, *session: ClientSession | None = None*)

Bases: *ETH*

RPC_URL = 'http://localhost:5007'

coin_name: str = 'SBCH'

event_handlers: dict[str, Callable]

friendly_name: str = 'Smart Bitcoin Cash'

property node_id: Any

Return an attribute of instance, which is of type owner.

server: RPCProxy

property spec: Any

Return an attribute of instance, which is of type owner.

xpub: str | None

4.1.7 Polygon (MATIC)

Polygon (MATIC) support is based on our custom daemon implementation which tries to follow electrum APIs as closely as possible

class bitcart.coins.matic.**MATIC**(*rpc_url: str | None = None*, *rpc_user: str | None = None*, *rpc_pass: str | None = None*, *xpub: str | None = None*, *proxy: str | None = None*, *session: ClientSession | None = None*)

Bases: *ETH*

RPC_URL = 'http://localhost:5008'

coin_name: str = 'MATIC'

event_handlers: dict[str, Callable]

friendly_name: str = 'Polygon'

property node_id: Any

Return an attribute of instance, which is of type owner.

server: RPCProxy

property spec: Any

Return an attribute of instance, which is of type owner.

xpub: str | None

4.1.8 TRON (TRX)

TRON (TRX) support is based on our custom daemon implementation which tries to follow electrum APIs as closely as possible

class bitcart.coins.trx.**TRX**(*rpc_url: str | None = None*, *rpc_user: str | None = None*, *rpc_pass: str | None = None*, *xpub: str | None = None*, *proxy: str | None = None*, *session: ClientSession | None = None*)

Bases: *ETH*

RPC_URL = 'http://localhost:5009'

coin_name: str = 'TRX'

event_handlers: dict[str, Callable]

friendly_name: str = 'Tron'

property node_id: Any

Return an attribute of instance, which is of type owner.

server: RPCProxy

property spec: Any

Return an attribute of instance, which is of type owner.

xpub: str | None

4.1.9 XRG

XRG supports Schnorr signatures, they are enabled out of the box

class bitcart.coins.xrg.**XRG**(*rpc_url: str | None = None*, *rpc_user: str | None = None*, *rpc_pass: str | None = None*, *xpub: str | None = None*, *proxy: str | None = None*, *session: ClientSession | None = None*)

Bases: *BCH*

AMOUNT_FIELD = 'amount (XRG)'

RPC_URL = 'http://localhost:5005'

coin_name: str = 'XRG'

event_handlers: dict[str, Callable]

friendly_name: str = 'Ergon'

property node_id: Any

Return an attribute of instance, which is of type owner.

server: RPCProxy

property spec: Any

Return an attribute of instance, which is of type owner.

xpub: str | None

4.1.10 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: str | None = None*, *rpc_user: str | None = None*, *rpc_pass: str | None = None*, *xpub: str | None = None*, *proxy: str | None = None*, *session: ClientSession | None = None*)

Bases: *BTC*

RPC_URL = 'http://localhost:5001'

coin_name: str = 'LTC'

event_handlers: dict[str, Callable]

friendly_name: str = 'Litecoin'

property node_id: Any

Return an attribute of instance, which is of type owner.

server: RPCProxy

property spec: Any

Return an attribute of instance, which is of type owner.

xpub: str | None

4.1.11 BSTY

BSTY 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 `BSTY_LIGHTNING` environment variable). If lightning is disabled, `LightningDisabledError` is raised when calling lightning methods.

class bitcart.coins.bsty.**BSTY**(*rpc_url: str | None = None*, *rpc_user: str | None = None*, *rpc_pass: str | None = None*, *xpub: str | None = None*, *proxy: str | None = None*, *session: ClientSession | None = None*)

Bases: *BTC*

RPC_URL = 'http://localhost:5003'

coin_name: str = 'BSTY'

event_handlers: dict[str, Callable]

friendly_name: str = 'GlobalBoost'

property node_id: Any

Return an attribute of instance, which is of type owner.

server: RPCProxy

property spec: Any

Return an attribute of instance, which is of type owner.

xpub: str | None

4.1.12 GRS

GRS 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 `GRS_LIGHTNING` environment variable). If lightning is disabled, `LightningDisabledError` is raised when calling lightning methods.

class bitcart.coins.grs.**GRS**(*rpc_url: str | None = None*, *rpc_user: str | None = None*, *rpc_pass: str | None = None*, *xpub: str | None = None*, *proxy: str | None = None*, *session: ClientSession | None = None*)

Bases: *BTC*

RPC_URL = 'http://localhost:5010'

coin_name: str = 'GRS'

event_handlers: dict[str, Callable]

friendly_name: str = 'Groestlcoin'

property node_id: Any

Return an attribute of instance, which is of type owner.

server: RPCProxy

property spec: Any

Return an attribute of instance, which is of type owner.

xpub: str | None

4.2 Utilities

bitcart.utils.**bitcoins**(*amount: int*) → Decimal

Convert amount from satoshis to bitcoins

Parameters
amount (*int*) – amount in satoshis

Returns
Decimal – amount in bitcoins

async bitcart.utils.**call_universal**(*func: Callable*, **args: Any*, ***kwargs: Any*) → Any

Call a function: async or sync one. All passed arguments are passed to the function too

Parameters
func (*Callable*) – a function to call: either sync or async one

Returns
Any – function execution result

bitcart.utils.**convert_amount_type**(*amount: str | Decimal*) → Decimal

Convert amount from str to Decimal

Parameters
amount (*Union[str, Decimal]*) – amount

Returns
Decimal

bitcart.utils.**json_encode**(*obj: Any*) → Any

json.dumps supporting Decimals

Parameters
obj (*Any*) – any object

Returns
Any – return value of json.dumps

bitcart.utils.**satoshis**(*amount: str | Decimal*) → int

Convert amount from bitcoins to satoshis

Parameters
amount (*Union[str, Decimal]*) – bitcoin amount

Returns
int – same amount in satoshis

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:

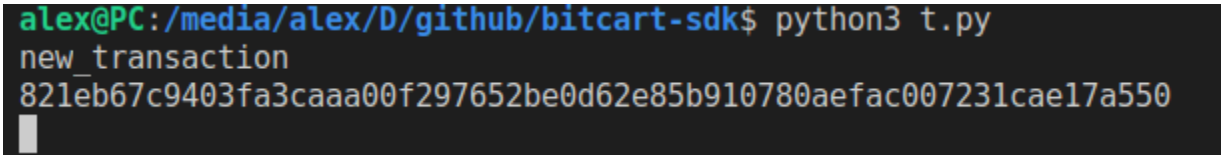
```
from bitcart import BTC

btc = BTC(xpub="your (x/y/z)pub or (x/y/z)prv or electrum seed")


@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 ()
- Bitcoin Cash
- Monero
- Ethereum
- Binance coin (BNB)
- SmartBCH
- Polygon (MATIC)
- Tron (TRX)
- Ergon
- Litecoin ()
- Globalboost ()
- Groestlcoin ()

To use proxy, install optional dependencies:

`pip install bitcart[proxy]` HTTP, SOCKS4 and SOCKS5 proxies supported.

To use, pass proxy url to coin constructor:

```
btc = BTC(proxy="socks5://localhost:9050")
```

PYTHON MODULE INDEX

b

INDEX