bitcoinlib.wallets Module Source Code

```
# -*- coding: utf-8 -*-
#    BitcoinLib - Python Cryptocurrency Library
#    WALLETS - HD wallet Class for Key and Transaction management
#    © 2016 - 2023 May - 1200 Web Development <http://1200wd.com/>
#
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU Affero General Public License as
#    published by the Free Software Foundation, either version 3 of the
#    License, or (at your option) any later version.
#
#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#    GNU Affero General Public License for more details.
#
#    You should have received a copy of the GNU Affero General Public License
#    along with this program.  If not, see <http://www.gnu.org/licenses/>.
#

import json
import random
from itertools import groupby
from operator import itemgetter
import numpy as np
import pickle
from bitcoinlib.db import *
from bitcoinlib.encoding import *
from bitcoinlib.keys import Address, BKeyError, HDKey, check_network_and_key, path_expand
from bitcoinlib.mnemonic import Mnemonic
from bitcoinlib.networks import Network
from bitcoinlib.values import Value, value_to_satoshi
from bitcoinlib.services.services import Service
from bitcoinlib.transactions import Input, Output, Transaction, get_unlocking_script_type
from bitcoinlib.scripts import Script
from sqlalchemy import func, or_

_logger = logging.getLogger(__name__)


class WalletError(Exception):
    """
    Handle Wallet class Exceptions

    """
    def __init__(self, msg=''):
        self.msg = msg
        _logger.error(msg)

    def __str__(self):
        return self.msg
```

```
def wallets_list(db_uri=None, include_cosigners=False, db_password=None):
    """
    List Wallets from database

    :param db_uri: URI of the database
    :type db_uri: str
    :param include_cosigners: Child wallets for multisig wallets are for internal use only and are skippec
    :type include_cosigners: bool
    :param db_password: Password to use for encrypted database. Requires the installation of sqlcipher (se
    documentation).
    :type db_password: str

    :return dict: Dictionary of wallets defined in database
    """

    session = Db(db_uri=db_uri, password=db_password).session
    wallets = session.query(DbWallet).order_by(DbWallet.id).all()
    wlst = []
    for w in wallets:
        if w.parent_id and not include_cosigners:
            continue
        wlst.append({
            'id': w.id,
            'name': w.name,
            'owner': w.owner,
            'network': w.network_name,
            'purpose': w.purpose,
            'scheme': w.scheme,
            'main_key_id': w.main_key_id,
            'parent_id': w.parent_id,
        })
    session.close()
    return wlst


def wallet_exists(wallet, db_uri=None, db_password=None):
    """
    Check if Wallets is defined in database

    :param wallet: Wallet ID as integer or Wallet Name as string
    :type wallet: int, str
    :param db_uri: URI of the database
    :type db_uri: str
       :param db_password: Password to use for encrypted database. Requires the installation of sc
documentation).
    :type db_password: str

    :return bool: True if wallet exists otherwise False
    """

    if wallet in [x['name'] for x in wallets_list(db_uri, db_password=db_password)]:
        return True
    if isinstance(wallet, int) and wallet in [x['id'] for x in wallets_list(db_uri, db_password=db_passwor
        return True
    return False


def wallet_create_or_open(
        name, keys='', owner='', network=None, account_id=0, purpose=None, scheme='bip32', sort_keys=True,
        password='', witness_type=None, encoding=None, multisig=None, sigs_required=None, cosigner_id=None
        key_path=None, db_uri=None, db_cache_uri=None, db_password=None):
```

```
    """
    Create a wallet with specified options if it doesn't exist, otherwise just open

    Returns Wallet object

    See Wallets class create method for option documentation
    """
    if wallet_exists(name, db_uri=db_uri, db_password=db_password):
        if keys or owner or password or witness_type or key_path:
            _logger.warning("Opening existing wallet, extra options are ignored")
        return Wallet(name, db_uri=db_uri, db_cache_uri=db_cache_uri, db_password=db_password)
    else:
        return Wallet.create(name, keys, owner, network, account_id, purpose, scheme, sort_keys,
                             password, witness_type, encoding, multisig, sigs_required, cosigner_id,
                             key_path, db_uri=db_uri, db_cache_uri=db_cache_uri, db_password=db_password)


def wallet_delete(wallet, db_uri=None, force=False, db_password=None):
    """
    Delete wallet and associated keys and transactions from the database. If wallet has unspent outputs it
    WalletError exception unless 'force=True' is specified

    :param wallet: Wallet ID as integer or Wallet Name as string
    :type wallet: int, str
    :param db_uri: URI of the database
    :type db_uri: str
    :param force: If set to True wallet will be deleted even if unspent outputs are found. Default is Fals
    :type force: bool
       :param db_password: Password to use for encrypted database. Requires the installation of sc
documentation).
    :type db_password: str

    :return int: Number of rows deleted, so 1 if successful
    """

    session = Db(db_uri=db_uri, password=db_password).session
    if isinstance(wallet, int) or wallet.isdigit():
        w = session.query(DbWallet).filter_by(id=wallet)
    else:
        w = session.query(DbWallet).filter_by(name=wallet)
    if not w or not w.first():
        session.close()
        raise WalletError("Wallet '%s' not found" % wallet)
    wallet_id = w.first().id

    # Delete co-signer wallets if this is a multisig wallet
    for cw in session.query(DbWallet).filter_by(parent_id=wallet_id).all():
        wallet_delete(cw.id, db_uri=db_uri, force=force)

    # Delete keys from this wallet and update transactions (remove key_id)
    ks = session.query(DbKey).filter_by(wallet_id=wallet_id)
    if bool([k for k in ks if k.balance and k.is_private]) and not force:
        session.close()
        raise WalletError("Wallet still has unspent outputs. Use 'force=True' to delete this wallet")
    k_ids = [k.id for k in ks]
    session.query(DbTransactionOutput).filter(DbTransactionOutput.key_id.in_(k_ids)).update(
        {DbTransactionOutput.key_id: None})
    session.query(DbTransactionInput).filter(DbTransactionInput.key_id.in_(k_ids)).update(
        {DbTransactionInput.key_id: None})
    session.query(DbKeyMultisigChildren).filter(DbKeyMultisigChildren.parent_id.in_(k_ids)).delete()
    session.query(DbKeyMultisigChildren).filter(DbKeyMultisigChildren.child_id.in_(k_ids)).delete()
```

```
    ks.delete()

    # Delete incomplete transactions from wallet
    txs = session.query(DbTransaction).filter_by(wallet_id=wallet_id, is_complete=False)
    for tx in txs:
        session.query(DbTransactionOutput).filter_by(transaction_id=tx.id).delete()
        session.query(DbTransactionInput).filter_by(transaction_id=tx.id).delete()
    txs.delete()

    # Unlink transactions from this wallet (remove wallet_id)
    session.query(DbTransaction).filter_by(wallet_id=wallet_id).update({DbTransaction.wallet_id: None})

    res = w.delete()
    session.commit()
    session.close()

    _logger.info("Wallet '%s' deleted" % wallet)

    return res


def wallet_empty(wallet, db_uri=None, db_password=None):
    """
    Remove all generated keys and transactions from wallet. Does not delete the wallet itself or the maste
    so everything can be recreated.

    :param wallet: Wallet ID as integer or Wallet Name as string
    :type wallet: int, str
    :param db_uri: URI of the database
    :type db_uri: str
       :param db_password: Password to use for encrypted database. Requires the installation of sq
documentation).
    :type db_password: str

    :return bool: True if successful
    """

    session = Db(db_uri=db_uri, password=db_password).session
    if isinstance(wallet, int) or wallet.isdigit():
        w = session.query(DbWallet).filter_by(id=wallet)
    else:
        w = session.query(DbWallet).filter_by(name=wallet)
    if not w or not w.first():
        raise WalletError("Wallet '%s' not found" % wallet)
    wallet_id = w.first().id

    # Delete keys from this wallet and update transactions (remove key_id)
    ks = session.query(DbKey).filter(DbKey.wallet_id == wallet_id, DbKey.parent_id != 0)
    for k in ks:
        session.query(DbTransactionOutput).filter_by(key_id=k.id).update({DbTransactionOutput.key_id: None
        session.query(DbTransactionInput).filter_by(key_id=k.id).update({DbTransactionInput.key_id: None})
        session.query(DbKeyMultisigChildren).filter_by(parent_id=k.id).delete()
        session.query(DbKeyMultisigChildren).filter_by(child_id=k.id).delete()
    ks.delete()

    # Delete incomplete transactions from wallet
    txs = session.query(DbTransaction).filter_by(wallet_id=wallet_id, is_complete=False)
    for tx in txs:
        session.query(DbTransactionOutput).filter_by(transaction_id=tx.id).delete()
        session.query(DbTransactionInput).filter_by(transaction_id=tx.id).delete()
    txs.delete()
```

```
    # Unlink transactions from this wallet (remove wallet_id)
    session.query(DbTransaction).filter_by(wallet_id=wallet_id).update({DbTransaction.wallet_id: None})

    session.commit()
    session.close()

    _logger.info("All keys and transactions from wallet '%s' deleted" % wallet)

    return True


def wallet_delete_if_exists(wallet, db_uri=None, force=False, db_password=None):
    """
      Delete wallet and associated keys from the database. If wallet has unspent outputs it raises
exception
    unless 'force=True' is specified. If the wallet does not exist return False

    :param wallet: Wallet ID as integer or Wallet Name as string
    :type wallet: int, str
    :param db_uri: URI of the database
    :type db_uri: str
    :param force: If set to True wallet will be deleted even if unspent outputs are found. Default is Fals
    :type force: bool
       :param db_password: Password to use for encrypted database. Requires the installation of sc
documentation).
    :type db_password: str

    :return int: Number of rows deleted, so 1 if successful
    """

    if wallet_exists(wallet, db_uri, db_password=db_password):
        return wallet_delete(wallet, db_uri, force, db_password=db_password)
    return False


def normalize_path(path):
    """
    Normalize BIP0044 key path for HD keys. Using single quotes for hardened keys

    >>> normalize_path("m/44h/2p/1'/0/100")
    "m/44'/2'/1'/0/100"

    :param path: BIP0044 key path
    :type path: str

    :return str: Normalized BIP0044 key path with single quotes
    """

    levels = path.split("/")
    npath = ""
    for level in levels:
        if not level:
            raise WalletError("Could not parse path. Index is empty.")
        nlevel = level
        if level[-1] in "'HhPp":
            nlevel = level[:-1] + "'"
        npath += nlevel + "/"
    if npath[-1] == "/":
        npath = npath[:-1]
    return npath
```

来=：Wallletkey

```
class WalletKey(object):
    """
    Used as attribute of Wallet class. Contains HDKey class, and adds extra wallet related information suc
    key ID, name, path and balance.

    All WalletKeys are stored in a database
    """

    @staticmethod
    def from_key(name, wallet_id, session, key, account_id=0, network=None, change=0, purpose=44, parent_i
                 path='m', key_type=None, encoding=None, witness_type=DEFAULT_WITNESS_TYPE, multisig=False
                 cosigner_id=None):
        """
        Create WalletKey from a HDKey object or key.

        Normally you don't need to call this method directly. Key creation is handled by the Wallet class.

        >>> w = wallet_create_or_open('hdwalletkey_test')
                                                                                                >>>           wi
'xprv9s21ZrQH143K2mcs9jcK4EjALbu2z1N9qsMTUG1frmnXM3NNCSGR57yLhwTccfNCwdSQEDftgjCGm96P29wGGcbBsPqZH85iqpoHA
        >>> wk = WalletKey.from_key('import_key', w.wallet_id, w._session, wif)
        >>> wk.address
        '1MwVEhGq6gg1eeSrEdZom5bHyPqXtJSnPg'
        >>> wk # doctest:+ELLIPSIS
                                                                        <WalletKey(key_id=...,          nam
wif=xprv9s21ZrQH143K2mcs9jcK4EjALbu2z1N9qsMTUG1frmnXM3NNCSGR57yLhwTccfNCwdSQEDftgjCGm96P29wGGcbBsPqZH85iqp
path=m)>

        :param name: New key name
        :type name: str
        :param wallet_id: ID of wallet where to store key
        :type wallet_id: int
        :param session: Required Sqlalchemy Session object
        :type session: sqlalchemy.orm.session.Session
        :param key: Optional key in any format accepted by the HDKey class
        :type key: str, int, byte, HDKey
        :param account_id: Account ID for specified key, default is 0
        :type account_id: int
        :param network: Network of specified key
        :type network: str
        :param change: Use 0 for normal key, and 1 for change key (for returned payments)
        :type change: int
        :param purpose: BIP0044 purpose field, default is 44
        :type purpose: int
        :param parent_id: Key ID of parent, default is 0 (no parent)
        :type parent_id: int
        :param path: BIP0044 path of given key, default is 'm' (masterkey)
        :type path: str
        :param key_type: Type of key, single or BIP44 type
        :type key_type: str
        :param encoding: Encoding used for address, i.e.: base58 or bech32. Default is base58
        :type encoding: str
        :param witness_type: Witness type used when creating transaction script: legacy, p2sh-segwit or se
        :type witness_type: str
         :param multisig: Specify if key is part of multisig wallet, used for create keys and key represe
as WIF and addreses
        :type multisig: bool
        :param cosigner_id: Set this if you would like to create keys for other cosigners.
        :type cosigner_id: int
```

key 包括物. HDkey

extra information

HDkey → Wallet key

wallet 的 ID

(str. int. byte. HDkey class)

是否用于多重签名

```
        :return WalletKey: WalletKey object
        """

        key_is_address = False
        if isinstance(key, HDKey):
            k = key
            if network is None:
                network = k.network.name
            elif network != k.network.name:
                raise WalletError("Specified network and key network should be the same")
        elif isinstance(key, Address):
            k = key
            key_is_address = True
            if network is None:
                network = k.network.name
            elif network != k.network.name:
                raise WalletError("Specified network and key network should be the same")
        else:
            if network is None:
                network = DEFAULT_NETWORK
            k = HDKey(import_key=key, network=network)
        if not encoding and witness_type:
            encoding = get_encoding_from_witness(witness_type)
        script_type = script_type_default(witness_type, multisig)

        if not key_is_address:
            keyexists = session.query(DbKey).\
                filter(DbKey.wallet_id == wallet_id,
                       DbKey.wif == k.wif(witness_type=witness_type, multisig=multisig, is_private=True)).
            if keyexists:
                _logger.warning("Key already exists in this wallet. Key ID: %d" % keyexists.id)
                return WalletKey(keyexists.id, session, k)

            if key_type != 'single' and k.depth != len(path.split('/'))-1:
                if path == 'm' and k.depth > 1:
                    path = "M"

            address = k.address(encoding=encoding, script_type=script_type)
            wk = session.query(DbKey).filter(
                DbKey.wallet_id == wallet_id,
                or_(DbKey.public == k.public_byte,
                    DbKey.wif == k.wif(witness_type=witness_type, multisig=multisig, is_private=False),
                    DbKey.address == address)).first()
            if wk:
                wk.wif = k.wif(witness_type=witness_type, multisig=multisig, is_private=True)
                wk.is_private = True
                wk.private = k.private_byte
                wk.public = k.public_byte
                wk.path = path
                session.commit()
                return WalletKey(wk.id, session, k)

                    nk = DbKey(name=name[:80], wallet_id=wallet_id, public=k.public_byte, private=k.
purpose=purpose,
                     account_id=account_id, depth=k.depth, change=change, address_index=k.child_index,
                     wif=k.wif(witness_type=witness_type, multisig=multisig, is_private=True), address=a
                     parent_id=parent_id, compressed=k.compressed, is_private=k.is_private, path=path,
                     key_type=key_type, network_name=network, encoding=encoding, cosigner_id=cosigner_ic
        else:
            keyexists = session.query(DbKey).\
```

```
            filter(DbKey.wallet_id == wallet_id,
                   DbKey.address == k.address).first()
        if keyexists:
            _logger.warning("Key with ID %s already exists" % keyexists.id)
            return WalletKey(keyexists.id, session, k)
        nk = DbKey(name=name[:80], wallet_id=wallet_id, purpose=purpose,
                   account_id=account_id, depth=k.depth, change=change, address=k.address,
                   parent_id=parent_id, compressed=k.compressed, is_private=False, path=path,
                   key_type=key_type, network_name=network, encoding=encoding, cosigner_id=cosigner_id

    session.merge(DbNetwork(name=network))
    session.add(nk)
    session.commit()
    return WalletKey(nk.id, session, k)

def _commit(self):
    try:
        self._session.commit()
    except Exception:
        self._session.rollback()
        raise

def __init__(self, key_id, session, hdkey_object=None):
    """
    Initialize WalletKey with specified ID, get information from database.

    :param key_id: ID of key as mentioned in database
    :type key_id: int
    :param session: Required Sqlalchemy Session object
    :type session: sqlalchemy.orm.session.Session
    :param hdkey_object: Optional HDKey object. Specify HDKey object if available for performance
    :type hdkey_object: HDKey

    """

    self._session = session
    wk = session.query(DbKey).filter_by(id=key_id).first()
    if wk:
        self._dbkey = wk
        self._hdkey_object = hdkey_object
        if hdkey_object and isinstance(hdkey_object, HDKey):
            assert(not wk.public or wk.public == hdkey_object.public_byte)
            assert(not wk.private or wk.private == hdkey_object.private_byte)
            self._hdkey_object = hdkey_object
        self.key_id = key_id
        self._name = wk.name
        self.wallet_id = wk.wallet_id
        self.key_public = None if not wk.public else wk.public
        self.key_private = None if not wk.private else wk.private
        self.account_id = wk.account_id
        self.change = wk.change
        self.address_index = wk.address_index
        self.wif = wk.wif
        self.address = wk.address
        self._balance = wk.balance
        self.purpose = wk.purpose
        self.parent_id = wk.parent_id
        self.is_private = wk.is_private
        self.path = wk.path
        self.wallet = wk.wallet
        self.network_name = wk.network_name
```

```
            if not self.network_name:
                self.network_name = wk.wallet.network_name
            self.network = Network(self.network_name)
            self.depth = wk.depth
            self.key_type = wk.key_type
            self.compressed = wk.compressed
            self.encoding = wk.encoding
            self.cosigner_id = wk.cosigner_id
            self.used = wk.used
        else:
            raise WalletError("Key with id %s not found" % key_id)

    def __repr__(self):
        return "<WalletKey(key_id=%d, name=%s, wif=%s, path=%s)>" % (self.key_id, self.name, self.wif, sel

    @property
    def name(self):
        """
        Return name of wallet key

        :return str:
        """
        return self._name

    @name.setter
    def name(self, value):
        """
        Set key name, update in database

        :param value: Name for this key
        :type value: str

        :return str:
        """

        self._name = value
        self._dbkey.name = value
        self._commit()

    def key(self):
        """
        Get HDKey object for current WalletKey

        :return HDKey:
        """

        self._hdkey_object = None
        if self.key_type == 'multisig':
            self._hdkey_object = []
            for kc in self._dbkey.multisig_children:
                        self._hdkey_object.append(HDKey.from_wif(kc.child_key.wif,  network=kc.child_key.
compressed=self.compressed))
        if self._hdkey_object is None and self.wif:
            self._hdkey_object = HDKey.from_wif(self.wif, network=self.network_name, compressed=self.compr
        return self._hdkey_object

    def balance(self, as_string=False):
        """
        Get total value of unspent outputs

        :param as_string: Specify 'string' to return a string in currency format
```

```
        :type as_string: bool

        :return float, str: Key balance
        """

        if as_string:
            return Value.from_satoshi(self._balance, network=self.network).str_unit()
        else:
            return self._balance

    def public(self):
        """
        Return current key as public WalletKey object with all private information removed

        :return WalletKey:
        """
        pub_key = self
        pub_key.is_private = False
        pub_key.key_private = None
        if self.key():
            pub_key.wif = self.key().wif()
        if self._hdkey_object:
            self._hdkey_object = pub_key._hdkey_object.public()
        self._dbkey = None
        return pub_key

    def as_dict(self, include_private=False):
        """
        Return current key information as dictionary

        :param include_private: Include private key information in dictionary
        :type include_private: bool

        """

        kdict = {
            'id': self.key_id,
            'key_type': self.key_type,
            'network': self.network.name,
            'is_private': self.is_private,
            'name': self.name,
            'key_public': '' if not self.key_public else self.key_public.hex(),
            'account_id':  self.account_id,
            'parent_id': self.parent_id,
            'depth': self.depth,
            'change': self.change,
            'address_index': self.address_index,
            'address': self.address,
            'encoding': self.encoding,
            'path': self.path,
            'balance': self.balance(),
            'balance_str': self.balance(as_string=True)
        }
        if include_private:
            kdict.update({
                'key_private': self.key_private.hex(),
                'wif': self.wif,
            })
        return kdict
```

张三：WalletTransaction

```
class WalletTransaction(Transaction):
    """
    Used as attribute of Wallet class. Child of Transaction object with extra reference to
    wallet and database object.

    All WalletTransaction items are stored in a database
    """

    def __init__(self, hdwallet, account_id=None, *args, **kwargs):
        """
        Initialize WalletTransaction object with reference to a Wallet object

        :param hdwallet: Wallet object, wallet name or ID
        :type hdWallet: HDwallet, str, int
        :param account_id: Account ID
        :type account_id: int
        :param args: Arguments for HDWallet parent class
        :type args: args
        :param kwargs: Keyword arguments for Wallet parent class
        :type kwargs: kwargs
        """

        assert isinstance(hdwallet, Wallet)
        self.hdwallet = hdwallet
        self.pushed = False
        self.error = None
        self.response_dict = None
        self.account_id = account_id
        if not account_id:
            self.account_id = self.hdwallet.default_account_id
        witness_type = 'legacy'
        if hdwallet.witness_type in ['segwit', 'p2sh-segwit']:
            witness_type = 'segwit'
        Transaction.__init__(self, witness_type=witness_type, *args, **kwargs)
        addresslist = hdwallet.addresslist()
        self.outgoing_tx = bool([i.address for i in self.inputs if i.address in addresslist])
        self.incoming_tx = bool([o.address for o in self.outputs if o.address in addresslist])

    def __repr__(self):
        return "<WalletTransaction(input_count=%d, output_count=%d, status=%s, network=%s)>" % \
               (len(self.inputs), len(self.outputs), self.status, self.network.name)

    def __deepcopy__(self, memo):
        cls = self.__class__
        result = cls.__new__(cls)
        memo[id(self)] = result
        self_dict = self.__dict__
        for k, v in self_dict.items():
            if k != 'hdwallet':
                setattr(result, k, deepcopy(v, memo))
        result.hdwallet = self.hdwallet
        return result

    @classmethod
    def from_transaction(cls, hdwallet, t):
        """
        Create WalletTransaction object from Transaction object

        :param hdwallet: Wallet object, wallet name or ID
        :type hdwallet: HDwallet, str, int
        :param t: Specify Transaction object
```

```
        :type t: Transaction

        :return WalletClass:
        """
        return cls(hdwallet=hdwallet, inputs=t.inputs, outputs=t.outputs, locktime=t.locktime, version=t.v
                   network=t.network.name, fee=t.fee, fee_per_kb=t.fee_per_kb, size=t.size, txid=t.txid,
                   txhash=t.txhash, date=t.date, confirmations=t.confirmations, block_height=t.block_heigh
                   block_hash=t.block_hash, input_total=t.input_total, output_total=t.output_total,
                   rawtx=t.rawtx, status=t.status, coinbase=t.coinbase, verified=t.verified, flag=t.flag)

    @classmethod
    def from_txid(cls, hdwallet, txid):
        """
        Read single transaction from database with given transaction ID / transaction hash

        :param hdwallet: Wallet object
        :type hdwallet: Wallet
        :param txid: Transaction hash as hexadecimal string
        :type txid: str, bytes

        :return WalletClass:

        """
        sess = hdwallet._session
        # If txid is unknown add it to database, else update
        db_tx_query = sess.query(DbTransaction). \
            filter(DbTransaction.wallet_id == hdwallet.wallet_id, DbTransaction.txid == to_bytes(txid))
        db_tx = db_tx_query.scalar()
        if not db_tx:
            return

        fee_per_kb = None
        if db_tx.fee and db_tx.size:
            fee_per_kb = int((db_tx.fee / db_tx.size) * 1000)
        network = Network(db_tx.network_name)

        inputs = []
        for inp in db_tx.inputs:
            sequence = 0xffffffff
            if inp.sequence:
                sequence = inp.sequence
            inp_keys = []
            if inp.key_id:
                key = hdwallet.key(inp.key_id)
                if key.key_type == 'multisig':
                    db_key = sess.query(DbKey).filter_by(id=key.key_id).scalar()
                    for ck in db_key.multisig_children:
                        inp_keys.append(ck.child_key.public.hex())
                else:
                    inp_keys = key.key()

            inputs.append(Input(
                prev_txid=inp.prev_txid, output_n=inp.output_n, keys=inp_keys, unlocking_script=inp.script
                script_type=inp.script_type, sequence=sequence, index_n=inp.index_n, value=inp.value,
                double_spend=inp.double_spend, witness_type=inp.witness_type, network=network, address=inp
                witnesses=inp.witnesses))

        outputs = []
        for out in db_tx.outputs:
            address = ''
            public_key = b''
```

```
            if out.key_id:
                key = hdwallet.key(out.key_id)
                address = key.address
                if key.key_type != 'multisig':
                    if key.key() and not isinstance(key.key(), Address):
                        public_key = key.key().public_hex
            outputs.append(Output(value=out.value, address=address, public_key=public_key,
                                  lock_script=out.script, spent=out.spent, output_n=out.output_n,
                                  script_type=out.script_type, network=network))

        return cls(hdwallet=hdwallet, inputs=inputs, outputs=outputs, locktime=db_tx.locktime,
                   version=db_tx.version, network=network, fee=db_tx.fee, fee_per_kb=fee_per_kb,
                   size=db_tx.size, txid=to_hexstring(txid), date=db_tx.date, confirmations=db_tx.confirma
                   block_height=db_tx.block_height, input_total=db_tx.input_total, output_total=db_tx.outp
                   rawtx=db_tx.raw, status=db_tx.status, coinbase=db_tx.coinbase,
                   verified=db_tx.verified)

    def to_transaction(self):
        return Transaction(self.inputs, self.outputs, self.locktime, self.version,
                           self.network.name, self.fee, self.fee_per_kb, self.size,
                           self.txid, self.txhash, self.date, self.confirmations,
                           self.block_height, self.block_hash, self.input_total,
                           self.output_total, self.rawtx, self.status, self.coinbase,
                           self.verified, self.witness_type, self.flag)

    def sign(self, keys=None, index_n=0, multisig_key_n=None, hash_type=SIGHASH_ALL, fail_on_unknown_key=F
             replace_signatures=False):
        """
        Sign this transaction. Use existing keys from wallet or use keys argument for extra keys.

        :param keys: Extra private keys to sign the transaction
        :type keys: HDKey, str
        :param index_n: Transaction index_n to sign
        :type index_n: int
          :param multisig_key_n: Index number of key for multisig input for segwit transactions. Leave
known. If not specified all possibilities will be checked
        :type multisig_key_n: int
        :param hash_type: Hashtype to use, default is SIGHASH_ALL
        :type hash_type: int
        :param fail_on_unknown_key: Method fails if public key from signature is not found in public key l
        :type fail_on_unknown_key: bool
        :param replace_signatures: Replace signature with new one if already signed.
        :type replace_signatures: bool

        :return None:
        """
        priv_key_list_arg = []
        if keys:
            key_paths = list(dict.fromkeys([ti.key_path for ti in self.inputs if ti.key_path[0] == 'm']))
            if not isinstance(keys, list):
                keys = [keys]
            for priv_key in keys:
                if not isinstance(priv_key, HDKey):
                    if isinstance(priv_key, str) and len(str(priv_key).split(' ')) > 4:
                        priv_key = HDKey.from_passphrase(priv_key, network=self.network)
                    else:
                        priv_key = HDKey(priv_key, network=self.network.name)
                priv_key_list_arg.append((None, priv_key))
                if key_paths and priv_key.depth == 0 and priv_key.key_type != "single":
                    for key_path in key_paths:
                        priv_key_list_arg.append((key_path, priv_key.subkey_for_path(key_path)))
```

```
        for ti in self.inputs:
            priv_key_list = []
            for (key_path, priv_key) in priv_key_list_arg:
                if (not key_path or key_path == ti.key_path) and priv_key not in priv_key_list:
                    priv_key_list.append(priv_key)
            priv_key_list += [k for k in ti.keys if k.is_private]
            Transaction.sign(self, priv_key_list, ti.index_n, multisig_key_n, hash_type, fail_on_unknown_k
                             replace_signatures)
        self.verify()
        self.error = ""

    def send(self, offline=False):
        """
        Verify and push transaction to network. Update UTXO's in database after successful send

        :param offline: Just return the transaction object and do not send it when offline = True. Default
        :type offline: bool

        :return None:

        """

        self.error = None
        if not self.verified and not self.verify():
            self.error = "Cannot verify transaction"
            return None

        if offline:
            return None

        srv = Service(network=self.network.name, providers=self.hdwallet.providers,
                      cache_uri=self.hdwallet.db_cache_uri)
        res = srv.sendrawtransaction(self.raw_hex())
        if not res:
            self.error = "Cannot send transaction. %s" % srv.errors
            return None
        if 'txid' in res:
            _logger.info("Successfully pushed transaction, result: %s" % res)
            self.txid = res['txid']
            self.status = 'unconfirmed'
            self.confirmations = 0
            self.pushed = True
            self.response_dict = srv.results
            self.store()

            # Update db: Update spent UTXO's, add transaction to database
            for inp in self.inputs:
                txid = inp.prev_txid
                utxos = self.hdwallet._session.query(DbTransactionOutput).join(DbTransaction).\
                    filter(DbTransaction.txid == txid,
                           DbTransactionOutput.output_n == inp.output_n_int,
                           DbTransactionOutput.spent.is_(False)).all()
                for u in utxos:
                    u.spent = True

            self.hdwallet._commit()
            self.hdwallet._balance_update(network=self.network.name)
            return None
        self.error = "Transaction not send, unknown response from service providers"

    def store(self):
```

```
        """
        Store this transaction to database

        :return int: Transaction index number
        """

        sess = self.hdwallet._session
        # If txid is unknown add it to database, else update
        db_tx_query = sess.query(DbTransaction). \
            filter(DbTransaction.wallet_id == self.hdwallet.wallet_id, DbTransaction.txid == bytes.fromhex
        db_tx = db_tx_query.scalar()
        if not db_tx:
            db_tx_query = sess.query(DbTransaction). \
                filter(DbTransaction.wallet_id.is_(None), DbTransaction.txid == bytes.fromhex(self.txid))
            db_tx = db_tx_query.first()
            if db_tx:
                db_tx.wallet_id = self.hdwallet.wallet_id

        if not db_tx:
            new_tx = DbTransaction(
                wallet_id=self.hdwallet.wallet_id, txid=bytes.fromhex(self.txid), block_height=self.block_
                size=self.size, confirmations=self.confirmations, date=self.date, fee=self.fee, status=sel
                input_total=self.input_total, output_total=self.output_total, network_name=self.network.na
                raw=self.rawtx, verified=self.verified, account_id=self.account_id)
            sess.add(new_tx)
            self.hdwallet._commit()
            txidn = new_tx.id
        else:
            txidn = db_tx.id
            db_tx.block_height = self.block_height if self.block_height else db_tx.block_height
            db_tx.confirmations = self.confirmations if self.confirmations else db_tx.confirmations
            db_tx.date = self.date if self.date else db_tx.date
            db_tx.fee = self.fee if self.fee else db_tx.fee
            db_tx.status = self.status if self.status else db_tx.status
            db_tx.input_total = self.input_total if self.input_total else db_tx.input_total
            db_tx.output_total = self.output_total if self.output_total else db_tx.output_total
            db_tx.network_name = self.network.name if self.network.name else db_tx.name
            db_tx.raw = self.rawtx if self.rawtx else db_tx.raw
            db_tx.verified = self.verified
            self.hdwallet._commit()

        assert txidn
        for ti in self.inputs:
            tx_key = sess.query(DbKey).filter_by(wallet_id=self.hdwallet.wallet_id, address=ti.address).sc
            key_id = None
            if tx_key:
                key_id = tx_key.id
                tx_key.used = True
            tx_input = sess.query(DbTransactionInput). \
                filter_by(transaction_id=txidn, index_n=ti.index_n).scalar()
            if not tx_input:
                witnesses = int_to_varbyteint(len(ti.witnesses)) + b''.join([bytes(varstr(w)) for w in ti.
                new_tx_item = DbTransactionInput(
                    transaction_id=txidn, output_n=ti.output_n_int, key_id=key_id, value=ti.value,
                    prev_txid=ti.prev_txid, index_n=ti.index_n, double_spend=ti.double_spend,
                    script=ti.unlocking_script, script_type=ti.script_type, witness_type=ti.witness_type,
                    sequence=ti.sequence, address=ti.address, witnesses=witnesses)
                sess.add(new_tx_item)
            elif key_id:
                tx_input.key_id = key_id
                if ti.value:
```

```
                    tx_input.value = ti.value
                if ti.prev_txid:
                    tx_input.prev_txid = ti.prev_txid
                if ti.unlocking_script:
                    tx_input.script = ti.unlocking_script

            self.hdwallet._commit()
        for to in self.outputs:
            tx_key = sess.query(DbKey).\
                filter_by(wallet_id=self.hdwallet.wallet_id, address=to.address).scalar()
            key_id = None
            if tx_key:
                key_id = tx_key.id
                tx_key.used = True
            spent = to.spent
            tx_output = sess.query(DbTransactionOutput). \
                filter_by(transaction_id=txidn, output_n=to.output_n).scalar()
            if not tx_output:
                new_tx_item = DbTransactionOutput(
                    transaction_id=txidn, output_n=to.output_n, key_id=key_id, address=to.address, value=t
                    spent=spent, script=to.lock_script, script_type=to.script_type)
                sess.add(new_tx_item)
            elif key_id:
                tx_output.key_id = key_id
                tx_output.spent = spent if spent is not None else tx_output.spent
            self.hdwallet._commit()
        return txidn

    def info(self):
        """
        Print Wallet transaction information to standard output. Include send information.
        """

        Transaction.info(self)
        print("Pushed to network: %s" % self.pushed)
        print("Wallet: %s" % self.hdwallet.name)
        if self.error:
            print("Errors: %s" % self.error)
        print("\n")

    def export(self, skip_change=True):
        """
        Export this transaction as list of tuples in the following format:
            (transaction_date, transaction_hash, in/out, addresses_in, addresses_out, value, fee)

        A transaction with multiple inputs or outputs results in multiple tuples.

         :param skip_change: Do not include outputs to own wallet (default). Please note: So if this is
then an internal transfer is not exported.
        :type skip_change: boolean

        :return list of tuple:
        """
        mut_list = []
        wlt_addresslist = self.hdwallet.addresslist()
        input_addresslist = [i.address for i in self.inputs]
        if self.outgoing_tx:
            fee_per_output = self.fee / len(self.outputs)
            for o in self.outputs:
                o_value = -o.value
                if o.address in wlt_addresslist:
```

```
                    if skip_change:
                        continue
                    elif self.incoming_tx:
                        o_value = 0
                mut_list.append((self.date, self.txid, 'out', input_addresslist, o.address, o_value, fee_p
        else:
            for o in self.outputs:
                if o.address not in wlt_addresslist:
                    continue
                mut_list.append((self.date, self.txid, 'in', input_addresslist, o.address, o.value, 0))
        return mut_list

    def save(self, filename=None):
        """
        Store transaction object as file, so it can be imported in bitcoinlib later with the :func:`load`

          :param filename: Location and name of file, leave empty to store transaction in bitcoinlib da
.bitcoinlib/<transaction_id.tx)
        :type filename: str

        :return:
        """
        if not filename:
            p = Path(BCL_DATA_DIR, '%s.tx' % self.txid)
        else:
            p = Path(filename)
            if not p.parent or str(p.parent) == '.':
                p = Path(BCL_DATA_DIR, filename)
        f = p.open('wb')
        t = self.to_transaction()
        pickle.dump(t, f)
        f.close()

    def delete(self):
        """
        Delete this transaction from database.

         WARNING: Results in incomplete wallets, transactions will NOT be automatically downloaded again
or updating wallet. In normal situations only used to remove old unconfirmed transactions

        :return int: Number of deleted transactions
        """

        session = self.hdwallet._session
        txid = bytes.fromhex(self.txid)
        tx_query = session.query(DbTransaction).filter_by(txid=txid)
        tx = tx_query.scalar()
        session.query(DbTransactionOutput).filter_by(transaction_id=tx.id).delete()
        session.query(DbTransactionInput).filter_by(transaction_id=tx.id).delete()
        session.query(DbKey).filter_by(latest_txid=txid).update({DbKey.latest_txid: None})
        res = tx_query.delete()
        self.hdwallet._commit()
        return res
```

class Wallet(object): 未回：Wallet

```
    """
    Class to create and manage keys Using the BIP0044 Hierarchical Deterministic wallet definitions, so yc
    use one Masterkey to generate as much child keys as you want in a structured manner.

    You can import keys in many format such as WIF or extended WIF, bytes, hexstring, seeds or private key
```

```
    For the Bitcoin network, Litecoin or any other network you define in the settings.

    Easily send and receive transactions. Compose transactions automatically or select unspent outputs.

    Each wallet name must be unique and can contain only one cointype and purpose, but practically unlimit
    accounts and addresses.
    """

    @classmethod
    def _create(cls, name, key, owner, network, account_id, purpose, scheme, parent_id, sort_keys,
                witness_type, encoding, multisig, sigs_required, cosigner_id, key_path, db_uri, db_cache_u
                db_password):
        db = Db(db_uri, db_password)
        session = db.session
        if (db_uri is None or db_uri.startswith("sqlite")) and db_cache_uri is None:
            db_cache_uri = DEFAULT_DATABASE_CACHE
        elif not db_cache_uri:
            db_cache_uri = db.db_uri
        db_uri = db.db_uri
        if session.query(DbWallet).filter_by(name=name).count():
            raise WalletError("Wallet with name '%s' already exists" % name)
        else:
            _logger.info("Create new wallet '%s'" % name)
        if not name:
            raise WalletError("Please enter wallet name")

        if not isinstance(key_path, list):
            key_path = key_path.split('/')
        key_depth = 1 if not key_path else len(key_path) - 1
        base_path = 'm'
        if hasattr(key, 'depth'):
            if key.depth is None:
                key.depth = key_depth
            if key.depth > 0:
                hardened_keys = [x for x in key_path if x[-1:] == "'"]
                if hardened_keys:
                    depth_public_master = key_path.index(hardened_keys[-1])
                    if depth_public_master != key.depth:
                        raise WalletError("Depth of provided public master key %d does not correspond with
                                          "%s. Did you provide correct witness_type and multisig attribute
                                          (key.depth, key_path))
                key_path = ['M'] + key_path[key.depth+1:]
                base_path = 'M'

        if isinstance(key_path, list):
            key_path = '/'.join(key_path)
        session.merge(DbNetwork(name=network))
        new_wallet = DbWallet(name=name, owner=owner, network_name=network, purpose=purpose, scheme=scheme
                              sort_keys=sort_keys, witness_type=witness_type, parent_id=parent_id, encodir
                              multisig=multisig, multisig_n_required=sigs_required, cosigner_id=cosigner_i
                              key_path=key_path)
        session.add(new_wallet)
        session.commit()
        new_wallet_id = new_wallet.id

        if scheme == 'bip32' and multisig and parent_id is None:
            w = cls(new_wallet_id, db_uri=db_uri, db_cache_uri=db_cache_uri)
        elif scheme == 'bip32':
            mk = WalletKey.from_key(key=key, name=name, session=session, wallet_id=new_wallet_id, network=
                                    account_id=account_id, purpose=purpose, key_type='bip32', encoding=enc
```

```
                                witness_type=witness_type, multisig=multisig, path=base_path)
            new_wallet.main_key_id = mk.key_id
            session.commit()

            w = cls(new_wallet_id, db_uri=db_uri, db_cache_uri=db_cache_uri, main_key_object=mk.key())
            w.key_for_path([0, 0], account_id=account_id, cosigner_id=cosigner_id)
        else:  # scheme == 'single':
            if not key:
                key = HDKey(network=network, depth=key_depth)
            mk = WalletKey.from_key(key=key, name=name, session=session, wallet_id=new_wallet_id, network=
                                account_id=account_id, purpose=purpose, key_type='single', encoding=en
                                witness_type=witness_type, multisig=multisig)
            new_wallet.main_key_id = mk.key_id
            session.commit()
            w = cls(new_wallet_id, db_uri=db_uri, db_cache_uri=db_cache_uri, main_key_object=mk.key())

        session.close()
        return w

    def _commit(self):
        try:
            self._session.commit()
        except Exception:
            self._session.rollback()
            raise
```

是自己创建的结构. Bitcoin 库没有"钱包"的概念.

函数1. create

```
    @classmethod
    def create(cls, name, keys=None, owner='', network=None, account_id=0, purpose=0, scheme='bip32',
               sort_keys=True, password='', witness_type=None, encoding=None, multisig=None, sigs_required
               cosigner_id=None, key_path=None, db_uri=None, db_cache_uri=None, db_password=None):
        """
        Create Wallet and insert in database. Generate masterkey or import key when specified.
```

介绍

① When only a name is specified a legacy Wallet with a single masterkey is created with standard p2w
scripts.

仅用 name 新建 Wallet 实例

```
        >>> if wallet_delete_if_exists('create_legacy_wallet_test'): pass
        >>> w = Wallet.create('create_legacy_wallet_test')
        >>> w
        <Wallet(name="create_legacy_wallet_test")>
```

② To create a multi signature wallet specify multiple keys (private or public) and provide the sigs_
argument if it different then len(keys)

需要多个 key 来创建实例

```
        >>> if wallet_delete_if_exists('create_legacy_multisig_wallet_test'): pass
        >>> w = Wallet.create('create_legacy_multisig_wallet_test', keys=[HDKey(), HDKey().public()])
```

③ To create a native segwit wallet use the option witness_type = 'segwit' and for old style addresse
embedded segwit script us 'ps2h-segwit' as witness_type.

SegWit (隔离见证) 钱包

```
        >>> if wallet_delete_if_exists('create_segwit_wallet_test'): pass
        >>> w = Wallet.create('create_segwit_wallet_test', witness_type='segwit')
```

④ Use a masterkey WIF when creating a wallet:

使用 masterkey WIF 创建

```
        >>>                                                                          wi
'xprv9s21ZrQH143K3cxbMVswDTYgAc9CeXABQjCD9zmXCpXw4MxN93LanEARbBmV3utHZS9Db4FX1C1RbC5KSNAjQ5WNJ1dDBJ34Pjfi
        >>> if wallet_delete_if_exists('bitcoinlib_legacy_wallet_test', force=True): pass
        >>> w = Wallet.create('bitcoinlib_legacy_wallet_test', wif)
        >>> w
        <Wallet(name="bitcoinlib_legacy_wallet_test")>
```

```
        >>> # Add some test utxo data:
                                            >>>     if     w.utxo_add('16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg',
'748799c9047321cb27a6320a827f1f69d767fe889c14bf11f27549638d566fe4', 0): pass

        Please mention account_id if you are using multiple accounts.
```

参数

```
        :param name: Unique name of this Wallet
        :type name: str
           :param keys: Masterkey to or list of keys to use for this wallet. Will be automatically cr
specified. One or more keys are obligatory for multisig wallets. Can contain all key formats accepted
object, a HDKey object or BIP39 passphrase
         :type keys: str, bytes, int, HDKey, HDWalletKey, list of str, list of bytes, list of int, list o
of HDWalletKey
        :param owner: Wallet owner for your own reference
        :type owner: str
        :param network: Network name, use default if not specified
        :type network: str
        :param account_id: Account ID, default is 0
        :type account_id: int
        :param purpose: BIP43 purpose field, will be derived from witness_type and multisig by default
        :type purpose: int
        :param scheme: Key structure type, i.e. BIP32 or single
        :type scheme: str
        :param sort_keys: Sort keys according to BIP45 standard (used for multisig keys)
        :type sort_keys: bool
        :param password: Password to protect passphrase, only used if a passphrase is supplied in the 'key
        :type password: str
          :param witness_type: Specify witness type, default is 'legacy'. Use 'segwit' for native segre
wallet, or 'p2sh-segwit' for legacy compatible wallets
        :type witness_type: str
         :param encoding: Encoding used for address generation: base58 or bech32. Default is derive from
witness type
        :type encoding: str
        :param multisig: Multisig wallet or child of a multisig wallet, default is None / derive from numb
        :type multisig: bool
           :param sigs_required: Number of signatures required for validation if using a multisignature
example 2 for 2-of-3 multisignature. Default is all keys must be signed
        :type sigs_required: int
         :param cosigner_id: Set this if wallet contains only public keys, more than one private key or
like to create keys for other cosigners. Note: provided keys of a multisig wallet are sorted if sort
(default) so if your provided key list is not sorted the cosigned_id may be different.
        :type cosigner_id: int
          :param key_path: Key path for multisig wallet, use to create your own non-standard key path.
follow the following rules:
            * Path start with masterkey (m) and end with change / address_index
            * If accounts are used, the account level must be 3. I.e.: m/purpose/coin_type/account/
            * All keys must be hardened, except for change, address_index or cosigner_id
            * Max length of path is 8 levels
        :type key_path: list, str
        :param db_uri: URI of the database for wallets, wallet transactions and keys
        :type db_uri: str
        :param db_cache_uri: URI of the cache database. If not specified  the default cache database is us
sqlite, for other databasetypes the cache database is merged with the wallet database (db_uri)
        :type db_cache_uri: str
        :param db_password: Password to encrypt database. Requires the installation of sqlcipher (see docu
        :type db_password: str

        :return Wallet:
        """
```

实现

```
        if multisig is None:
```

```
    if keys and isinstance(keys, list) and len(keys) > 1:
        multisig = True
    else:
        multisig = False
if scheme not in ['bip32', 'single']:
    raise WalletError("Only bip32 or single key scheme's are supported at the moment")
if witness_type not in [None, 'legacy', 'p2sh-segwit', 'segwit']:
    raise WalletError("Witness type %s not supported at the moment" % witness_type)
if name.isdigit():
    raise WalletError("Wallet name '%s' invalid, please include letter characters" % name)

if multisig:
    if password:
        raise WalletError("Password protected multisig wallets not supported")
    if scheme != 'bip32':
        raise WalletError("Multisig wallets should use bip32 scheme not %s" % scheme)
    if sigs_required is None:
        sigs_required = len(keys)
    if sigs_required > len(keys):
        raise WalletError("Number of keys required to sign is greater then number of keys providec
elif not isinstance(keys, list):
    keys = [keys]
if len(keys) > 15:
    raise WalletError("Redeemscripts with more then 15 keys are non-standard and could result in "
                      "locked up funds")

hdkey_list = []
if keys and isinstance(keys, list) and sort_keys:
    keys.sort(key=lambda x: ('0' if isinstance(x, HDKey) else '1'))
for key in keys:
    if isinstance(key, HDKey):
        if network and network != key.network.name:
            raise WalletError("Network from key (%s) is different then specified network (%s)" %
                              (key.network.name, network))
        network = key.network.name
        if witness_type is None:
            witness_type = key.witness_type
    elif key:
        # If key consists of several words assume it is a passphrase and convert it to a HDKey obj
        if isinstance(key, str) and len(key.split(" ")) > 1:
            if not network:
                raise WalletError("Please specify network when using passphrase to create a key")
            key = HDKey.from_seed(Mnemonic().to_seed(key, password), network=network)
        else:
            try:
                if isinstance(key, WalletKey):
                    key = key._hdkey_object
                else:
                    key = HDKey(key, password=password, network=network)
            except BKeyError:
                try:
                    scheme = 'single'
                    key = Address.parse(key, encoding=encoding, network=network)
                except Exception:
                    raise WalletError("Invalid key or address: %s" % key)
            if network is None:
                network = key.network.name
            if witness_type is None:
                witness_type = key.witness_type
    hdkey_list.append(key)
```

```
    if network is None:
        network = DEFAULT_NETWORK
    if witness_type is None:
        witness_type = DEFAULT_WITNESS_TYPE
    if network in ['dash', 'dash_testnet', 'dogecoin', 'dogecoin_testnet'] and witness_type != 'legacy
        raise WalletError("Segwit is not supported for %s wallets" % network.capitalize())
    elif network in ('dogecoin', 'dogecoin_testnet') and witness_type not in ('legacy', 'p2sh-segwit')
        raise WalletError("Pure segwit addresses are not supported for Dogecoin wallets. "
                          "Please use p2sh-segwit instead")

    if not key_path:
        if scheme == 'single':
            key_path = ['m']
            purpose = 0
        else:
            ks = [k for k in WALLET_KEY_STRUCTURES if k['witness_type'] == witness_type and
                  k['multisig'] == multisig and k['purpose'] is not None]
            if len(ks) > 1:
                raise WalletError("Please check definitions in WALLET_KEY_STRUCTURES. Multiple options
                                  "witness_type - multisig combination")
            if ks and not purpose:
                purpose = ks[0]['purpose']
            if ks and not encoding:
                encoding = ks[0]['encoding']
            key_path = ks[0]['key_path']
    else:
        if purpose is None:
            purpose = 0
    if not encoding:
        encoding = get_encoding_from_witness(witness_type)

    if multisig:
        key = ''
    else:
        key = hdkey_list[0]

    main_key_path = key_path
    if multisig:
        if sort_keys:
            hdkey_list.sort(key=lambda x: x.public_byte)
        cos_prv_lst = [hdkey_list.index(cw) for cw in hdkey_list if cw.is_private]
        if cosigner_id is None:
            if not cos_prv_lst:
                raise WalletError("This wallet does not contain any private keys, please specify cosig
                                  "this wallet")
            elif len(cos_prv_lst) > 1:
                raise WalletError("This wallet contains more then 1 private key, please specify "
                                  "cosigner_id for this wallet")
            cosigner_id = 0 if not cos_prv_lst else cos_prv_lst[0]
        if hdkey_list[cosigner_id].key_type == 'single':
            main_key_path = 'm'

    hdpm = cls._create(name, key, owner=owner, network=network, account_id=account_id, purpose=purpose
                       scheme=scheme, parent_id=None, sort_keys=sort_keys, witness_type=witness_type,
                       encoding=encoding, multisig=multisig, sigs_required=sigs_required, cosigner_id=
                       key_path=main_key_path, db_uri=db_uri, db_cache_uri=db_cache_uri, db_password=d

    if multisig:
        wlt_cos_id = 0
        for cokey in hdkey_list:
            if hdpm.network.name != cokey.network.name:
```

```
                    raise WalletError("Network for key %s (%s) is different then network specified: %s/%s"
                                      (cokey.wif(is_private=False), cokey.network.name, network, hdpm.netw
                scheme = 'bip32'
                wn = name + '-cosigner-%d' % wlt_cos_id
                c_key_path = key_path
                if cokey.key_type == 'single':
                    scheme = 'single'
                    c_key_path = ['m']
                w = cls._create(name=wn, key=cokey, owner=owner, network=network, account_id=account_id,
                                purpose=hdpm.purpose, scheme=scheme, parent_id=hdpm.wallet_id, sort_keys=s
                                witness_type=hdpm.witness_type, encoding=encoding, multisig=True,
                                sigs_required=None, cosigner_id=wlt_cos_id, key_path=c_key_path,
                                db_uri=db_uri, db_cache_uri=db_cache_uri, db_password=db_password)
                hdpm.cosigner.append(w)
                wlt_cos_id += 1
            # hdpm._dbwallet = hdpm._session.query(DbWallet).filter(DbWallet.id == hdpm.wallet_id)
            # hdpm._dbwallet.update({DbWallet.cosigner_id: hdpm.cosigner_id})
            # hdpm._dbwallet.update({DbWallet.key_path: hdpm.key_path})
            # hdpm._session.commit()

        return hdpm

    def __enter__(self):
        return self

    def __init__(self, wallet, db_uri=None, db_cache_uri=None, session=None, main_key_object=None, db_pass
        """
        Open a wallet with given ID or name

        :param wallet: Wallet name or ID
        :type wallet: int, str
        :param db_uri: URI of the database
        :type db_uri: str
        :param db_cache_uri: URI of the cache database. If not specified  the default cache database is us
sqlite, for other databasetypes the cache database is merged with the wallet database (db_uri)
        :type db_cache_uri: str
        :param session: Sqlalchemy session
        :type session: sqlalchemy.orm.session.Session
        :param main_key_object: Pass main key object to save time
        :type main_key_object: HDKey
        """

        if session:
            self._session = session
        else:
            dbinit = Db(db_uri=db_uri, password=db_password)
            self._session = dbinit.session
            self._engine = dbinit.engine
        self.db_uri = db_uri
        self.db_cache_uri = db_cache_uri
        if isinstance(wallet, int) or wallet.isdigit():
            db_wlt = self._session.query(DbWallet).filter_by(id=wallet).scalar()
        else:
            db_wlt = self._session.query(DbWallet).filter_by(name=wallet).scalar()
        if db_wlt:
            self._dbwallet = db_wlt
            self.wallet_id = db_wlt.id
            self._name = db_wlt.name
            self._owner = db_wlt.owner
            self.network = Network(db_wlt.network_name)
            self.purpose = db_wlt.purpose
```

```
            self.scheme = db_wlt.scheme
            self._balance = None
            self._balances = []
            self.main_key_id = db_wlt.main_key_id
            self.main_key = None
            self._default_account_id = db_wlt.default_account_id
            self.multisig_n_required = db_wlt.multisig_n_required
            co_sign_wallets = self._session.query(DbWallet).\
                filter(DbWallet.parent_id == self.wallet_id).order_by(DbWallet.name).all()
            self.cosigner = [Wallet(w.id, db_uri=db_uri, db_cache_uri=db_cache_uri) for w in co_sign_walle
            self.sort_keys = db_wlt.sort_keys
            if db_wlt.main_key_id:
                self.main_key = WalletKey(self.main_key_id, session=self._session, hdkey_object=main_key_o
            if self._default_account_id is None:
                self._default_account_id = 0
                if self.main_key:
                    self._default_account_id = self.main_key.account_id
            _logger.info("Opening wallet '%s'" % self.name)
            self._key_objects = {
                self.main_key_id: self.main_key
            }
            self.providers = None
            self.witness_type = db_wlt.witness_type
            self.encoding = db_wlt.encoding
            self.multisig = db_wlt.multisig
            self.cosigner_id = db_wlt.cosigner_id
            self.script_type = script_type_default(self.witness_type, self.multisig, locking_script=True)
            self.key_path = [] if not db_wlt.key_path else db_wlt.key_path.split('/')
            self.depth_public_master = 0
            self.parent_id = db_wlt.parent_id
            if self.main_key and self.main_key.depth > 0:
                self.depth_public_master = self.main_key.depth
                self.key_depth = self.depth_public_master + len(self.key_path) - 1
            else:
                hardened_keys = [x for x in self.key_path if x[-1:] == "'"]
                if hardened_keys:
                    self.depth_public_master = self.key_path.index(hardened_keys[-1])
                self.key_depth = len(self.key_path) - 1
            self.last_updated = None
        else:
            raise WalletError("Wallet '%s' not found, please specify correct wallet ID or name." % wallet)

    def __exit__(self, exception_type, exception_value, traceback):
        try:
            self._session.close()
            self._engine.dispose()
        except Exception:
            pass

    def __del__(self):
        try:
            self._session.close()
            self._engine.dispose()
        except Exception:
            pass

    def __repr__(self):
        db_uri = self.db_uri.split('?')[0]
        if DEFAULT_DATABASE in db_uri:
            return "<Wallet(name=\"%s\")>" % self.name
        return "<Wallet(name=\"%s\", db_uri=\"%s\")>" % \
```

```
            (self.name, db_uri)

    def __str__(self):
        return self.name

    def _get_account_defaults(self, network=None, account_id=None, key_id=None):
        """
        Check parameter values for network and account ID, return defaults if no network or account ID is
        If a network is specified but no account ID this method returns the first account ID it finds.

        :param network: Network code, leave empty for default
        :type network: str
        :param account_id: Account ID, leave emtpy for default
        :type account_id: int
        :param key_id: Key ID to just update 1 key
        :type key_id: int

        :return: network code, account ID and DbKey instance of account ID key
        """

        if key_id:
            kobj = self.key(key_id)
            network = kobj.network_name
            account_id = kobj.account_id
        if network is None:
            network = self.network.name
        if account_id is None and network == self.network.name:
            account_id = self.default_account_id
        qr = self._session.query(DbKey).\
            filter_by(wallet_id=self.wallet_id, purpose=self.purpose, depth=self.depth_public_master,
                      network_name=network)
        if account_id is not None:
            qr = qr.filter_by(account_id=account_id)
        acckey = qr.first()
        if len(qr.all()) > 1 and "account'" in self.key_path:
            _logger.warning("No account_id specified and more than one account found for this network %s.
                            "Using a random account" % network)
        if account_id is None:
            if acckey:
                account_id = acckey.account_id
            else:
                account_id = 0
        return network, account_id, acckey

    @property
    def default_account_id(self):
        return self._default_account_id

    @default_account_id.setter
    def default_account_id(self, value):
        self._default_account_id = value
        self._dbwallet = self._session.query(DbWallet).filter(DbWallet.id == self.wallet_id). \
            update({DbWallet.default_account_id: value})
        self._commit()

    @property
    def owner(self):
        """
        Get wallet Owner

        :return str:
```

```
        """

        return self._owner

    @owner.setter
    def owner(self, value):
        """
        Set wallet Owner in database

        :param value: Owner
        :type value: str

        :return str:
        """

        self._owner = value
        self._dbwallet = self._session.query(DbWallet).filter(DbWallet.id == self.wallet_id).\
            update({DbWallet.owner: value})
        self._commit()

    @property
    def name(self):
        """
        Get wallet name

        :return str:
        """

        return self._name

    @name.setter
    def name(self, value):
        """
        Set wallet name, update in database

        :param value: Name for this wallet
        :type value: str

        :return str:
        """

        if wallet_exists(value, db_uri=self.db_uri):
            raise WalletError("Wallet with name '%s' already exists" % value)
        self._name = value
        self._session.query(DbWallet).filter(DbWallet.id == self.wallet_id).update({DbWallet.name: value})
        self._commit()

    def default_network_set(self, network):
        if not isinstance(network, Network):
            network = Network(network)
        self.network = network
        self._session.query(DbWallet).filter(DbWallet.id == self.wallet_id).\
            update({DbWallet.network_name: network.name})
        self._commit()

    def import_master_key(self, hdkey, name='Masterkey (imported)'):
        """
        Import (another) masterkey in this wallet

        :param hdkey: Private key
        :type hdkey: HDKey, str
```

```
        :param name: Key name of masterkey
        :type name: str

        :return HDKey: Main key as HDKey object
        """

        network, account_id, acckey = self._get_account_defaults()
        if not isinstance(hdkey, HDKey):
            hdkey = HDKey(hdkey)
        if not isinstance(self.main_key, WalletKey):
            raise WalletError("Main wallet key is not an WalletKey instance. Type %s" % type(self.main_key
        if not hdkey.is_private or hdkey.depth != 0:
            raise WalletError("Please supply a valid private BIP32 master key with key depth 0")
        if self.main_key.is_private:
            raise WalletError("Main key is already a private key, cannot import key")
        if (self.main_key.depth != 1 and self.main_key.depth != 3 and self.main_key.depth != 4) or \
                self.main_key.key_type != 'bip32':
            raise WalletError("Current main key is not a valid BIP32 public master key")
        # pm = self.public_master()
        if not (self.network.name == self.main_key.network.name == hdkey.network.name):
            raise WalletError("Network of Wallet class, main account key and the imported private key must
                              "the same network")
        if self.main_key.wif != hdkey.public_master().wif():
            raise WalletError("This key does not correspond to current public master key")

        hdkey.key_type = 'bip32'
        ks = [k for k in WALLET_KEY_STRUCTURES if
              k['witness_type'] == self.witness_type and k['multisig'] == self.multisig and k['purpose'] i
        if len(ks) > 1:
            raise WalletError("Please check definitions in WALLET_KEY_STRUCTURES. Multiple options found f
                              "witness_type - multisig combination")
        self.key_path = ks[0]['key_path']
        self.main_key = WalletKey.from_key(
            key=hdkey, name=name, session=self._session, wallet_id=self.wallet_id, network=network,
            account_id=account_id, purpose=self.purpose, key_type='bip32', witness_type=self.witness_type)
        self.main_key_id = self.main_key.key_id
        self._key_objects.update({self.main_key_id: self.main_key})
        self._session.query(DbWallet).filter(DbWallet.id == self.wallet_id).\
            update({DbWallet.main_key_id: self.main_key_id})

        for key in self.keys(is_private=False):
            kp = key.path.split("/")
            if kp and kp[0] == 'M':
                kp = self.key_path[:self.depth_public_master+1] + kp[1:]
            self.key_for_path(kp, recreate=True)

        self._commit()
        return self.main_key

    def import_key(self, key, account_id=0, name='', network=None, purpose=44, key_type=None):
        """
        Add new single key to wallet.

        :param key: Key to import
        :type key: str, bytes, int, HDKey, Address
        :param account_id: Account ID. Default is last used or created account ID.
        :type account_id: int
        :param name: Specify name for key, leave empty for default
        :type name: str
         :param network: Network name, method will try to extract from key if not specified. Raises warni
could not be detected
```

```
        :type network: str
        :param purpose: BIP definition used, default is BIP44
        :type purpose: int
         :param key_type: Key type of imported key, can be single. Unrelated to wallet, bip32, bip44 or m
or extra master key import. Default is 'single'
        :type key_type: str

        :return WalletKey:
        """

        if self.scheme not in ['bip32', 'single']:
            raise WalletError("Keys can only be imported to a BIP32 or single type wallet, create a new wa
                              "instead")
        if isinstance(key, (HDKey, Address)):
            network = key.network.name
            hdkey = key
            if network not in self.network_list():
                raise WalletError("Network %s not found in this wallet" % network)
        else:
            if isinstance(key, str) and len(key.split(" ")) > 1:
                if network is None:
                    network = self.network
                hdkey = HDKey.from_seed(Mnemonic().to_seed(key), network=network)
            else:
                if network is None:
                    network = check_network_and_key(key, default_network=self.network.name)
                if network not in self.network_list():
                    raise WalletError("Network %s not available in this wallet, please create an account f
                                      "network first." % network)
                hdkey = HDKey(key, network=network, key_type=key_type)

        if not self.multisig:
            if self.main_key and self.main_key.depth == self.depth_public_master and \
                        not isinstance(hdkey, Address) and hdkey.is_private and hdkey.depth == 0 and s
'bip32':
                return self.import_master_key(hdkey, name)

            if key_type is None:
                hdkey.key_type = 'single'
                key_type = 'single'

            ik_path = 'm'
            if key_type == 'single':
                # Create path for unrelated import keys
                hdkey.depth = self.key_depth
                last_import_key = self._session.query(DbKey).filter(DbKey.path.like("import_key_%")).\
                    order_by(DbKey.path.desc()).first()
                if last_import_key:
                    ik_path = "import_key_" + str(int(last_import_key.path[-5:]) + 1).zfill(5)
                else:
                    ik_path = "import_key_00001"
                if not name:
                    name = ik_path

            mk = WalletKey.from_key(
                key=hdkey, name=name, wallet_id=self.wallet_id, network=network, key_type=key_type,
                account_id=account_id, purpose=purpose, session=self._session, path=ik_path,
                witness_type=self.witness_type)
            self._key_objects.update({mk.key_id: mk})
            if mk.key_id == self.main_key.key_id:
                self.main_key = mk
```

```
            return mk
        else:
            account_key = hdkey.public_master(witness_type=self.witness_type, multisig=True).wif()
            for w in self.cosigner:
                if w.main_key.key().wif_public() == account_key:
                    _logger.debug("Import new private cosigner key in this multisig wallet: %s" % account_
                    return w.import_master_key(hdkey)
            raise WalletError("Unknown key: Can only import a private key for a known public key in multis

    def _new_key_multisig(self, public_keys, name, account_id, change, cosigner_id, network, address_index
        if self.sort_keys:
            public_keys.sort(key=lambda pubk: pubk.key_public)
        public_key_list = [pubk.key_public for pubk in public_keys]
        public_key_ids = [str(x.key_id) for x in public_keys]

        # Calculate redeemscript and address and add multisig key to database
        # redeemscript = serialize_multisig_redeemscript(public_key_list, n_required=self.multisig_n_requi

        # todo: pass key object, reuse key objects
        redeemscript = Script(script_types=['multisig'], keys=public_key_list,
                              sigs_required=self.multisig_n_required).serialize()
        script_type = 'p2sh'
        if self.witness_type == 'p2sh-segwit':
            script_type = 'p2sh_p2wsh'
        address = Address(redeemscript, encoding=self.encoding, script_type=script_type, network=network)
        already_found_key = self._session.query(DbKey).filter_by(wallet_id=self.wallet_id,
                                                                 address=address.address).first()
        if already_found_key:
            return self.key(already_found_key.id)
        path = [pubk.path for pubk in public_keys if pubk.wallet.cosigner_id == self.cosigner_id][0]
        depth = self.cosigner[self.cosigner_id].main_key.depth + len(path.split("/")) - 1
        if not name:
            name = "Multisig Key " + '/'.join(public_key_ids)

        multisig_key = DbKey(
            name=name[:80], wallet_id=self.wallet_id, purpose=self.purpose, account_id=account_id,
            depth=depth, change=change, address_index=address_index, parent_id=0, is_private=False, path=p
            public=address.hash_bytes, wif='multisig-%s' % address, address=address.address, cosigner_id=c
            key_type='multisig', network_name=network)
        self._session.add(multisig_key)
        self._commit()
        for child_id in public_key_ids:
                                      self._session.add(DbKeyMultisigChildren(key_order=public_key_ids.inc
parent_id=multisig_key.id,
                                                              child_id=int(child_id)))
        self._commit()
        return self.key(multisig_key.id)

    def new_key(self, name='', account_id=None, change=0, cosigner_id=None, network=None):
        """
        Create a new HD Key derived from this wallet's masterkey. An account will be created for this wall
        with index 0 if there is no account defined yet.

        >>> w = Wallet('create_legacy_wallet_test')
        >>> w.new_key('my key') # doctest:+ELLIPSIS
        <WalletKey(key_id=..., name=my key, wif=..., path=m/44'/0'/0'/0/...)>

         :param name: Key name. Does not have to be unique but if you use it at reference you might choo
this. If not specified 'Key #' with a unique sequence number will be used
        :type name: str
        :param account_id: Account ID. Default is last used or created account ID.
```

```
        :type account_id: int
        :param change: Change (1) or payments (0). Default is 0
        :type change: int
        :param cosigner_id: Cosigner ID for key path
        :type cosigner_id: int
        :param network: Network name. Leave empty for default network
        :type network: str

        :return WalletKey:
        """

        if self.scheme == 'single':
            return self.main_key

        network, account_id, _ = self._get_account_defaults(network, account_id)
        if network != self.network.name and "coin_type'" not in self.key_path:
            raise WalletError("Multiple networks not supported by wallet key structure")
        if self.multisig:
            if not self.multisig_n_required:
                raise WalletError("Multisig_n_required not set, cannot create new key")
            if cosigner_id is None:
                if self.cosigner_id is None:
                    raise WalletError("Missing Cosigner ID value, cannot create new key")
                cosigner_id = self.cosigner_id

        address_index = 0
                  if  self.multisig  and  cosigner_id  is  not  None  and  (len(self.cosigner)  >  cos
self.cosigner[cosigner_id].key_path == 'm' or self.cosigner[cosigner_id].key_path == ['m']):
            req_path = []
        else:
            prevkey = self._session.query(DbKey).\
                filter_by(wallet_id=self.wallet_id, purpose=self.purpose, network_name=network, account_id
                          change=change, cosigner_id=cosigner_id, depth=self.key_depth).\
                order_by(DbKey.address_index.desc()).first()
            if prevkey:
                address_index = prevkey.address_index + 1
            req_path = [change, address_index]

        return self.key_for_path(req_path, name=name, account_id=account_id, network=network,
                                 cosigner_id=cosigner_id, address_index=address_index)

    def new_key_change(self, name='', account_id=None, network=None):
        """
        Create new key to receive change for a transaction. Calls :func:`new_key` method with change=1.

        :param name: Key name. Default name is 'Change #' with an address index
        :type name: str
        :param account_id: Account ID. Default is last used or created account ID.
        :type account_id: int
        :param network: Network name. Leave empty for default network
        :type network: str

        :return WalletKey:
        """

        return self.new_key(name=name, account_id=account_id, network=network, change=1)

    def scan_key(self, key):
        """
        Scan for new transactions for specified wallet key and update wallet transactions
```

```
        :param key: The wallet key as object or index
        :type key: WalletKey, int

        :return bool: New transactions found?

        """
        if isinstance(key, int):
            key = self.key(key)
        txs_found = False
        should_be_finished_count = 0
        while True:
            n_new = self.transactions_update(key_id=key.key_id)
            if n_new and n_new < MAX_TRANSACTIONS:
                if should_be_finished_count:
                    _logger.info("Possible recursive loop detected in scan_key(%d): retry %d/5" %
                                 (key.key_id, should_be_finished_count))
                should_be_finished_count += 1
            logger.info("Scanned key %d, %s Found %d new transactions" % (key.key_id, key.address, n_new))
            if not n_new or should_be_finished_count > 5:
                break
            txs_found = True
        return txs_found

    def scan(self, scan_gap_limit=5, account_id=None, change=None, rescan_used=False, network=None, keys_i
        """
         Generate new addresses/keys and scan for new transactions using the Service providers. Updates a
balances.

          Keep scanning for new transactions until no new transactions are found for 'scan_gap_limit' ad
scan keys from default network and account unless another network or account is specified.

        Use the faster :func:`utxos_update` method if you are only interested in unspent outputs.
        Use the :func:`transactions_update` method if you would like to manage the key creation yourself o
to scan a single key.

        :param scan_gap_limit: Amount of new keys and change keys (addresses) created for this wallet. Def
scanning stops if after 5 addresses no transaction are found.
        :type scan_gap_limit: int
        :param account_id: Account ID. Default is last used or created account ID.
        :type account_id: int
        :param change: Filter by change addresses. Set to True to include only change addresses, False to
regular addresses. None (default) to disable filter and include both
        :type change: bool
         :param rescan_used: Rescan already used addressed. Default is False, so funds send to old addr
ignored by default.
        :type rescan_used: bool
        :param network: Network name. Leave empty for default network
        :type network: str
        :param keys_ignore: Id's of keys to ignore
        :type keys_ignore: list of int

        :return:
        """

        network, account_id, _ = self._get_account_defaults(network, account_id)
        if self.scheme != 'bip32' and self.scheme != 'multisig' and scan_gap_limit < 2:
            raise WalletError("The wallet scan() method is only available for BIP32 wallets")
        if keys_ignore is None:
            keys_ignore = []

        # Rescan used addresses
```

```
        if rescan_used:
            for key in self.keys_addresses(account_id=account_id, change=change, network=network, used=Tru
                self.scan_key(key.id)

        # Update already known transactions with known block height
        self.transactions_update_confirmations()

        # Check unconfirmed transactions
        db_txs = self._session.query(DbTransaction). \
            filter(DbTransaction.wallet_id == self.wallet_id,
                   DbTransaction.network_name == network, DbTransaction.confirmations == 0).all()
        for db_tx in db_txs:
            self.transactions_update_by_txids([db_tx.txid])

        # Scan each key address, stop when no new transactions are found after set scan gap limit
        if change is None:
            change_range = [0, 1]
        else:
            change_range = [change]
        counter = 0
        for chg in change_range:
            while True:
                if self.scheme == 'single':
                    keys_to_scan = [self.key(k.id) for k in self.keys_addresses()[counter:counter+scan_gap
                    counter += scan_gap_limit
                else:
                    keys_to_scan = self.get_keys(account_id, network, number_of_keys=scan_gap_limit, chang
                n_highest_updated = 0
                for key in keys_to_scan:
                    if key.key_id in keys_ignore:
                        continue
                    keys_ignore.append(key.key_id)
                    n_high_new = 0
                    if self.scan_key(key):
                        if not key.address_index:
                            key.address_index = 0
                        n_high_new = key.address_index + 1
                    if n_high_new > n_highest_updated:
                        n_highest_updated = n_high_new
                if not n_highest_updated:
                    break

    def _get_key(self, account_id=None, network=None, cosigner_id=None, number_of_keys=1, change=0, as_lis
        network, account_id, _ = self._get_account_defaults(network, account_id)
        if cosigner_id is None:
            cosigner_id = self.cosigner_id
        elif cosigner_id > len(self.cosigner):
            raise WalletError("Cosigner ID (%d) can not be greater then number of cosigners for this walle
                              (cosigner_id, len(self.cosigner)))

        last_used_qr = self._session.query(DbKey.id).\
            filter_by(wallet_id=self.wallet_id, account_id=account_id, network_name=network, cosigner_id=c
                      used=True, change=change, depth=self.key_depth).\
            order_by(DbKey.id.desc()).first()
        last_used_key_id = 0
        if last_used_qr:
            last_used_key_id = last_used_qr.id
        dbkey = self._session.query(DbKey).\
            filter_by(wallet_id=self.wallet_id, account_id=account_id, network_name=network, cosigner_id=c
                      used=False, change=change, depth=self.key_depth).filter(DbKey.id > last_used_key_id)
            order_by(DbKey.id.desc()).all()
```

```
        key_list = []
        if self.scheme == 'single' and len(dbkey):
            number_of_keys = len(dbkey) if number_of_keys > len(dbkey) else number_of_keys
        for i in range(number_of_keys):
            if dbkey:
                dk = dbkey.pop()
                nk = self.key(dk.id)
            else:
                nk = self.new_key(account_id=account_id, change=change, cosigner_id=cosigner_id, network=n
            key_list.append(nk)
        if as_list:
            return key_list
        else:
            return key_list[0]

    def get_key(self, account_id=None, network=None, cosigner_id=None, change=0):
        """
        Get a unused key / address or create a new one with :func:`new_key` if there are no unused keys.
        Returns a key from this wallet which has no transactions linked to it.

        Use the get_keys() method to a list of unused keys. Calling the get_key() method repeatelly to rec
        list of key doesn't work: since the key is unused it would return the same result every time you c
        method.

        >>> w = Wallet('create_legacy_wallet_test')
        >>> w.get_key() # doctest:+ELLIPSIS
        <WalletKey(key_id=..., name=..., wif=..., path=m/44'/0'/0'/0/...)>

        :param account_id: Account ID. Default is last used or created account ID.
        :type account_id: int
        :param network: Network name. Leave empty for default network
        :type network: str
        :param cosigner_id: Cosigner ID for key path
        :type cosigner_id: int
        :param change: Payment (0) or change key (1). Default is 0
        :type change: int

        :return WalletKey:
        """
        return self._get_key(account_id, network, cosigner_id, change=change, as_list=False)

    def get_keys(self, account_id=None, network=None, cosigner_id=None, number_of_keys=1, change=0):
        """
        Get a list of unused keys / addresses or create a new ones with :func:`new_key` if there are no ur
        Returns a list of keys from this wallet which has no transactions linked to it.

        Use the get_key() method to get a single key.

        :param account_id: Account ID. Default is last used or created account ID.
        :type account_id: int
        :param network: Network name. Leave empty for default network
        :type network: str
        :param cosigner_id: Cosigner ID for key path
        :type cosigner_id: int
        :param number_of_keys: Number of keys to return. Default is 1
        :type number_of_keys: int
        :param change: Payment (0) or change key (1). Default is 0
        :type change: int

        :return list of WalletKey:
        """
```

```
    if self.scheme == 'single':
        raise WalletError("Single wallet has only one (master)key. Use get_key() or main_key() method"
    return self._get_key(account_id, network, cosigner_id, number_of_keys, change, as_list=True)

def get_key_change(self, account_id=None, network=None):
    """
    Get a unused change key or create a new one if there are no unused keys.
    Wrapper for the :func:`get_key` method

    :param account_id: Account ID. Default is last used or created account ID.
    :type account_id: int
    :param network: Network name. Leave empty for default network
    :type network: str

    :return WalletKey:
    """

    return self._get_key(account_id=account_id, network=network, change=1, as_list=False)

def get_keys_change(self, account_id=None, network=None, number_of_keys=1):
    """
    Get a unused change key or create a new one if there are no unused keys.
    Wrapper for the :func:`get_key` method

    :param account_id: Account ID. Default is last used or created account ID.
    :type account_id: int
    :param network: Network name. Leave empty for default network
    :type network: str
    :param number_of_keys: Number of keys to return. Default is 1
    :type number_of_keys: int

    :return list of WalletKey:
    """

    return self._get_key(account_id=account_id, network=network, change=1, number_of_keys=number_of_ke
                         as_list=True)

def new_account(self, name='', account_id=None, network=None):
    """
    Create a new account with a child key for payments and 1 for change.

    An account key can only be created if wallet contains a masterkey.

    :param name: Account Name. If not specified "Account #" with the account_id will be used as name
    :type name: str
    :param account_id: Account ID. Default is last accounts ID + 1
    :type account_id: int
    :param network: Network name. Leave empty for default network
    :type network: str

    :return WalletKey:
    """

    if self.scheme != 'bip32':
        raise WalletError("We can only create new accounts for a wallet with a BIP32 key scheme")
    if self.main_key and (self.main_key.depth != 0 or self.main_key.is_private is False):
        raise WalletError("A master private key of depth 0 is needed to create new accounts (depth: %d
                          self.main_key.depth)
    if "account'" not in self.key_path:
        raise WalletError("Accounts are not supported for this wallet. Account not found in key path %
                          self.key_path)
```

```
        if network is None:
            network = self.network.name
        elif network != self.network.name and "coin_type'" not in self.key_path:
            raise WalletError("Multiple networks not supported by wallet key structure")

        duplicate_cointypes = [Network(x).name for x in self.network_list() if Network(x).name != network
                               Network(x).bip44_cointype == Network(network).bip44_cointype]
        if duplicate_cointypes:
            raise WalletError("Can not create new account for network %s with same BIP44 cointype: %s" %
                              (network, duplicate_cointypes))

        # Determine account_id and name
        if account_id is None:
            account_id = 0
            qr = self._session.query(DbKey). \
                filter_by(wallet_id=self.wallet_id, purpose=self.purpose, network_name=network). \
                order_by(DbKey.account_id.desc()).first()
            if qr:
                account_id = qr.account_id + 1
        if self.keys(account_id=account_id, depth=self.depth_public_master, network=network):
            raise WalletError("Account with ID %d already exists for this wallet" % account_id)

        acckey = self.key_for_path([], level_offset=self.depth_public_master-self.key_depth, account_id=ac
                                   name=name, network=network)
        self.key_for_path([0, 0], network=network, account_id=account_id)
        self.key_for_path([1, 0], network=network, account_id=account_id)
        return acckey

    def path_expand(self, path, level_offset=None, account_id=None, cosigner_id=0, address_index=None, cha
                    network=DEFAULT_NETWORK):
        """
        Create key path. Specify part of key path to expand to key path used in this wallet.

        >>> w = Wallet('create_legacy_wallet_test')
        >>> w.path_expand([0,1200])
        ['m', "44'", "0'", "0'", '0', '1200']

        >>> w = Wallet('create_legacy_multisig_wallet_test')
        >>> w.path_expand([0,2], cosigner_id=1)
        ['m', "45'", '1', '0', '2']

        :param path: Part of path, for example [0, 2] for change=0 and address_index=2
        :type path: list, str
        :param level_offset: Just create part of path. For example -2 means create path with the last 2 i
address_index) or 1 will return the master key 'm'
        :type level_offset: int
        :param account_id: Account ID
        :type account_id: int
        :param cosigner_id: ID of cosigner
        :type cosigner_id: int
        :param address_index: Index of key, normally provided to 'path' argument
        :type address_index: int
        :param change: Change key = 1 or normal = 0, normally provided to 'path' argument
        :type change: int
        :param network: Network name. Leave empty for default network
        :type network: str

        :return list:
        """
        network, account_id, _ = self._get_account_defaults(network, account_id)
        return path_expand(path, self.key_path, level_offset, account_id=account_id, cosigner_id=cosigner_
```

```
                            address_index=address_index, change=change, purpose=self.purpose,
                            witness_type=self.witness_type, network=network)

    def key_for_path(self, path, level_offset=None, name=None, account_id=None, cosigner_id=None,
                     address_index=0, change=0, network=None, recreate=False):
        """
        Return key for specified path. Derive all wallet keys in path if they not already exists

        >>> w = wallet_create_or_open('key_for_path_example')
        >>> key = w.key_for_path([0, 0])
        >>> key.path
        "m/44'/0'/0'/0/0"

        >>> w.key_for_path([], level_offset=-2).path
        "m/44'/0'/0'"

        >>> w.key_for_path([], w.depth_public_master + 1).path
        "m/44'/0'/0'"

        Arguments provided in 'path' take precedence over other arguments. The address_index argument is i
        >>> key = w.key_for_path([0, 10], address_index=1000)
        >>> key.path
        "m/44'/0'/0'/0/10"
        >>> key.address_index
        10

        :param path: Part of key path, i.e. [0, 0] for [change=0, address_index=0]
        :type path: list, str
         :param level_offset: Just create part of path, when creating keys. For example -2 means create
last 2 items (change, address_index) or 1 will return the master key 'm'
        :type level_offset: int
        :param name: Specify key name for latest/highest key in structure
        :type name: str
        :param account_id: Account ID
        :type account_id: int
        :param cosigner_id: ID of cosigner
        :type cosigner_id: int
        :param address_index: Index of key, normally provided to 'path' argument
        :type address_index: int
        :param change: Change key = 1 or normal = 0, normally provided to 'path' argument
        :type change: int
        :param network: Network name. Leave empty for default network
        :type network: str
         :param recreate: Recreate key, even if already found in wallet. Can be used to update public key
key info
        :type recreate: bool

        :return WalletKey:
        """

        network, account_id, _ = self._get_account_defaults(network, account_id)
        cosigner_id = cosigner_id if cosigner_id is not None else self.cosigner_id
        level_offset_key = level_offset
        if level_offset and self.main_key and level_offset > 0:
            level_offset_key = level_offset - self.main_key.depth

        key_path = self.key_path
        if self.multisig and cosigner_id is not None and len(self.cosigner) > cosigner_id:
            key_path = self.cosigner[cosigner_id].key_path
        fullpath = path_expand(path, key_path, level_offset_key, account_id=account_id, cosigner_id=cosigr
                               purpose=self.purpose, address_index=address_index, change=change,
```

```
                        witness_type=self.witness_type, network=network)

    if self.multisig and self.cosigner:
        public_keys = []
        for wlt in self.cosigner:
            if wlt.scheme == 'single':
                wk = wlt.main_key
            else:
                wk = wlt.key_for_path(path, level_offset=level_offset, account_id=account_id, name=nam
                                      cosigner_id=cosigner_id, network=network, recreate=recreate)
            public_keys.append(wk)
        return self._new_key_multisig(public_keys, name, account_id, change, cosigner_id, network, add

    # Check for closest ancestor in wallet\
    wpath = fullpath
    if self.main_key.depth and fullpath and fullpath[0] != 'M':
        wpath = ["M"] + fullpath[self.main_key.depth + 1:]
    dbkey = None
    while wpath and not dbkey:
        qr = self._session.query(DbKey).filter_by(path=normalize_path('/'.join(wpath)), wallet_id=self
        if recreate:
            qr = qr.filter_by(is_private=True)
        dbkey = qr.first()
        wpath = wpath[:-1]
    if not dbkey:
        _logger.warning("No master or public master key found in this wallet")
        return None
    else:
        topkey = self.key(dbkey.id)

    # Key already found in db, return key
    if dbkey and dbkey.path == normalize_path('/'.join(fullpath)) and not recreate:
        return topkey
    else:
        # Create 1 or more keys add them to wallet
        nk = None
        parent_id = topkey.key_id
        ck = topkey.key()
        newpath = topkey.path
        n_items = len(str(dbkey.path).split('/'))
        for lvl in fullpath[n_items:]:
            ck = ck.subkey_for_path(lvl, network=network)
            newpath += '/' + lvl
            if not account_id:
                account_id = 0 if "account'" not in self.key_path or self.key_path.index("account'") >
                    fullpath) \
                    else int(fullpath[self.key_path.index("account'")][:-1])
            change = None if "change" not in self.key_path or self.key_path.index("change") >= len(ful
                else int(fullpath[self.key_path.index("change")])
            if name and len(fullpath) == len(newpath.split('/')):
                key_name = name
            else:
                key_name = "%s %s" % (self.key_path[len(newpath.split('/'))-1], lvl)
                key_name = key_name.replace("'", "").replace("_", " ")
            nk = WalletKey.from_key(key=ck, name=key_name, wallet_id=self.wallet_id, account_id=accoun
                                    change=change, purpose=self.purpose, path=newpath, parent_id=parer
                                    encoding=self.encoding, witness_type=self.witness_type,
                                    cosigner_id=cosigner_id, network=network, session=self._session)
            self._key_objects.update({nk.key_id: nk})
            parent_id = nk.key_id
        return nk
```

```
    def keys(self, account_id=None, name=None, key_id=None, change=None, depth=None, used=None, is_private
             has_balance=None, is_active=None, network=None, include_private=False, as_dict=False):
        """
        Search for keys in database. Include 0 or more of account_id, name, key_id, change and depth.

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> all_wallet_keys = w.keys()
        >>> w.keys(depth=0) # doctest:+ELLIPSIS
                                                                [<DbKey(id=...,       name='bitcoinlib_legacy_
wif='xprv9s21ZrQH143K3cxbMVswDTYgAc9CeXABQjCD9zmXCpXw4MxN93LanEARbBmV3utHZS9Db4FX1C1RbC5KSNAjQ5WNJ1dDBJ34E

        Returns a list of DbKey object or dictionary object if as_dict is True

        :param account_id: Search for account ID
        :type account_id: int
        :param name: Search for Name
        :type name: str
        :param key_id: Search for Key ID
        :type key_id: int
        :param change: Search for Change
        :type change: int
        :param depth: Only include keys with this depth
        :type depth: int
        :param used: Only return used or unused keys
        :type used: bool
        :param is_private: Only return private keys
        :type is_private: bool
        :param has_balance: Only include keys with a balance or without a balance, default is both
        :type has_balance: bool
          :param is_active: Hide inactive keys. Only include active keys with either a balance or whic
default is None (show all)
        :type is_active: bool
        :param network: Network name filter
        :type network: str
        :param include_private: Include private key information in dictionary
        :type include_private: bool
        :param as_dict: Return keys as dictionary objects. Default is False: DbKey objects
        :type as_dict: bool

        :return list of DbKey: List of Keys
        """

        qr = self._session.query(DbKey).filter_by(wallet_id=self.wallet_id).order_by(DbKey.id)
        if network is not None:
            qr = qr.filter(DbKey.network_name == network)
        if account_id is not None:
            qr = qr.filter(DbKey.account_id == account_id)
            if self.scheme == 'bip32' and depth is None:
                qr = qr.filter(DbKey.depth >= 3)
        if change is not None:
            qr = qr.filter(DbKey.change == change)
            if self.scheme == 'bip32' and depth is None:
                qr = qr.filter(DbKey.depth > self.key_depth - 1)
        if depth is not None:
            qr = qr.filter(DbKey.depth == depth)
        if name is not None:
            qr = qr.filter(DbKey.name == name)
        if key_id is not None:
            qr = qr.filter(DbKey.id == key_id)
            is_active = False
```

```
        elif used is not None:
            qr = qr.filter(DbKey.used == used)
        if is_private is not None:
            qr = qr.filter(DbKey.is_private == is_private)
        if has_balance is True and is_active is True:
            raise WalletError("Cannot use has_balance and is_active parameter together")
        if has_balance is not None:
            if has_balance:
                qr = qr.filter(DbKey.balance != 0)
            else:
                qr = qr.filter(DbKey.balance == 0)
        if is_active:  # Unused keys and keys with a balance
            qr = qr.filter(or_(DbKey.balance != 0, DbKey.used.is_(False)))
        keys = qr.order_by(DbKey.depth).all()
        if as_dict:
            keys = [x.__dict__ for x in keys]
            keys2 = []
            private_fields = []
            if not include_private:
                private_fields += ['private', 'wif']
            for key in keys:
                keys2.append({k: v for (k, v) in key.items()
                              if k[:1] != '_' and k != 'wallet' and k not in private_fields})
            return keys2
        qr.session.close()
        return keys

    def keys_networks(self, used=None, as_dict=False):
        """
        Get keys of defined networks for this wallet. Wrapper for the :func:`keys` method

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> network_key = w.keys_networks()
        >>> # Address index of hardened key 0' is 2147483648
        >>> network_key[0].address_index
        2147483648
        >>> network_key[0].path
        "m/44'/0'"

        :param used: Only return used or unused keys
        :type used: bool
        :param as_dict: Return as dictionary or DbKey object. Default is False: DbKey objects
        :type as_dict: bool

        :return list of (DbKey, dict):

        """

        if self.scheme != 'bip32':
            raise WalletError("The 'keys_network' method can only be used with BIP32 type wallets")
        try:
            depth = self.key_path.index("coin_type'")
        except ValueError:
            return []
        if self.multisig and self.cosigner:
                _logger.warning("No network keys available for multisig wallet, use networks() method
networks")
        return self.keys(depth=depth, used=used, as_dict=as_dict)

    def keys_accounts(self, account_id=None, network=DEFAULT_NETWORK, as_dict=False):
        """
```

```
        Get Database records of account key(s) with for current wallet. Wrapper for the :func:`keys` methc

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> account_key = w.keys_accounts()
        >>> account_key[0].path
        "m/44'/0'/0'"

         Returns nothing if no account keys are available for instance in multisig or single account wal
case use :func:`accounts` method instead.

        :param account_id: Search for Account ID
        :type account_id: int
        :param network: Network name filter
        :type network: str
        :param as_dict: Return as dictionary or DbKey object. Default is False: DbKey objects
        :type as_dict: bool

        :return list of (DbKey, dict):
        """

        return self.keys(account_id, depth=self.depth_public_master, network=network, as_dict=as_dict)

    def keys_addresses(self, account_id=None, used=None, is_active=None, change=None, network=None, depth=
                       as_dict=False):
        """
        Get address keys of specified account_id for current wallet. Wrapper for the :func:`keys` methods.

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> w.keys_addresses()[0].address
        '16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg'

        :param account_id: Account ID
        :type account_id: int
        :param used: Only return used or unused keys
        :type used: bool
          :param is_active: Hide inactive keys. Only include active keys with either a balance or whic
default is True
        :type is_active: bool
        :param change: Search for Change
        :type change: int
        :param network: Network name filter
        :type network: str
        :param depth: Filter by key depth. Default for BIP44 and multisig is 5
        :type depth: int
        :param as_dict: Return as dictionary or DbKey object. Default is False: DbKey objects
        :type as_dict: bool

        :return list of (DbKey, dict)
        """

        if depth is None:
            depth = self.key_depth
        return self.keys(account_id, depth=depth, used=used, change=change, is_active=is_active, network=r
                         as_dict=as_dict)

    def keys_address_payment(self, account_id=None, used=None, network=None, as_dict=False):
        """
          Get payment addresses (change=0) of specified account_id for current wallet. Wrapper for the
methods.

        :param account_id: Account ID
```

```
        :type account_id: int
        :param used: Only return used or unused keys
        :type used: bool
        :param network: Network name filter
        :type network: str
        :param as_dict: Return as dictionary or DbKey object. Default is False: DbKey objects
        :type as_dict: bool

        :return list of (DbKey, dict)
        """

        return self.keys(account_id, depth=self.key_depth, change=0, used=used, network=network, as_dict=a

    def keys_address_change(self, account_id=None, used=None, network=None, as_dict=False):
        """
          Get payment addresses (change=1) of specified account_id for current wallet. Wrapper for the
methods.

        :param account_id: Account ID
        :type account_id: int
        :param used: Only return used or unused keys
        :type used: bool
        :param network: Network name filter
        :type network: str
        :param as_dict: Return as dictionary or DbKey object. Default is False: DbKey objects
        :type as_dict: bool

        :return list of (DbKey, dict)
        """

        return self.keys(account_id, depth=self.key_depth, change=1, used=used, network=network, as_dict=a

    def addresslist(self, account_id=None, used=None, network=None, change=None, depth=None, key_id=None):
        """
        Get list of addresses defined in current wallet. Wrapper for the :func:`keys` methods.

        Use :func:`keys_addresses` method to receive full key objects

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> w.addresslist()[0]
        '16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg'

        :param account_id: Account ID
        :type account_id: int
        :param used: Only return used or unused keys
        :type used: bool, None
        :param network: Network name filter
        :type network: str
        :param change: Only include change addresses or not. Default is None which returns both
        :param depth: Filter by key depth. Default is None for standard key depth. Use -1 to show all keys
        :type depth: int
        :param key_id: Key ID to get address of just 1 key
        :type key_id: int

        :return list of str: List of address strings
        """

        addresslist = []
        if depth is None:
            depth = self.key_depth
        elif depth == -1:
```

```
            depth = None
        for key in self.keys(account_id=account_id, depth=depth, used=used, network=network, change=change
                             key_id=key_id, is_active=False):
            addresslist.append(key.address)
        return addresslist

    def key(self, term):
        """
        Return single key with given ID or name as WalletKey object

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> w.key('change 0').address
        '1HabJXe8mTwXiMzUWW5KdpYbFWu3hvtsbF'

        :param term: Search term can be key ID, key address, key WIF or key name
        :type term: int, str

        :return WalletKey: Single key as object
        """

        dbkey = None
        qr = self._session.query(DbKey).filter_by(wallet_id=self.wallet_id)
        if self.purpose:
            qr = qr.filter_by(purpose=self.purpose)
        if isinstance(term, numbers.Number):
            dbkey = qr.filter_by(id=term).scalar()
        if not dbkey:
            dbkey = qr.filter_by(address=term).first()
        if not dbkey:
            dbkey = qr.filter_by(wif=term).first()
        if not dbkey:
            dbkey = qr.filter_by(name=term).first()
        if dbkey:
            if dbkey.id in self._key_objects.keys():
                return self._key_objects[dbkey.id]
            else:
                hdwltkey = WalletKey(key_id=dbkey.id, session=self._session)
                self._key_objects.update({dbkey.id: hdwltkey})
                return hdwltkey
        else:
            raise BKeyError("Key '%s' not found" % term)

    def account(self, account_id):
        """
        Returns wallet key of specific BIP44 account.

        Account keys have a BIP44 path depth of 3 and have the format m/purpose'/network'/account'

        I.e: Use account(0).key().wif_public() to get wallet's public master key

        :param account_id: ID of account. Default is 0
        :type account_id: int

        :return WalletKey:
        """

        if "account'" not in self.key_path:
            raise WalletError("Accounts are not supported for this wallet. Account not found in key path %
                              self.key_path)
        qr = self._session.query(DbKey).\
            filter_by(wallet_id=self.wallet_id, purpose=self.purpose, network_name=self.network.name,
```

```
                account_id=account_id, depth=3).scalar()
        if not qr:
            raise WalletError("Account with ID %d not found in this wallet" % account_id)
        key_id = qr.id
        return self.key(key_id)

    def accounts(self, network=None):
        """
        Get list of accounts for this wallet

        :param network: Network name filter. Default filter is network of first main key
        :type network: str

        :return list of integers: List of accounts IDs
        """

        network, _, _ = self._get_account_defaults(network)
        if self.multisig and self.cosigner:
            if self.cosigner_id is None:
                raise WalletError("Missing Cosigner ID value for this wallet, cannot fetch account ID")
            accounts = [wk.account_id for wk in self.cosigner[self.cosigner_id].keys_accounts(network=netw
        else:
            accounts = [wk.account_id for wk in self.keys_accounts(network=network)]
        if not accounts:
            accounts = [self.default_account_id]
        return list(dict.fromkeys(accounts))

    def networks(self, as_dict=False):
        """
        Get list of networks used by this wallet

        :param as_dict: Return as dictionary or as Network objects, default is Network objects
        :type as_dict: bool

        :return list of (Network, dict):
        """

        nw_list = [self.network]
        if self.multisig and self.cosigner:
            keys_qr = self._session.query(DbKey.network_name).\
                filter_by(wallet_id=self.wallet_id, depth=self.key_depth).\
                group_by(DbKey.network_name).all()
            nw_list += [Network(nw[0]) for nw in keys_qr]
        elif self.main_key.key_type != 'single':
            wks = self.keys_networks()
            for wk in wks:
                nw_list.append(Network(wk.network_name))

        networks = []
        nw_list = list(dict.fromkeys(nw_list))
        for nw in nw_list:
            if as_dict:
                nw = nw.__dict__
                if '_sa_instance_state' in nw:
                    del nw['_sa_instance_state']
            networks.append(nw)

        return networks

    def network_list(self, field='name'):
        """
```

```
        Wrapper for :func:`networks` method, returns a flat list with currently used
        networks for this wallet.

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> w.network_list()
        ['bitcoin']

        :return list of str:
        """

        return [getattr(x, field) for x in self.networks()]

    def balance_update_from_serviceprovider(self, account_id=None, network=None):
        """
         Update balance of currents account addresses using default Service objects :func:`getbalance` m
total
        wallet balance in database.

        Please Note: Does not update UTXO's or the balance per key! For this use the :func:`updatebalance`
        instead

        :param account_id: Account ID. Leave empty for default account
        :type account_id: int
        :param network: Network name. Leave empty for default network
        :type network: str

        :return int: Total balance
        """

        network, account_id, acckey = self._get_account_defaults(network, account_id)
        srv = Service(network=network, providers=self.providers, cache_uri=self.db_cache_uri)
        balance = srv.getbalance(self.addresslist(account_id=account_id, network=network))
        if srv.results:
            new_balance = {
                'account_id': account_id,
                'network': network,
                'balance': balance
            }
            old_balance_item = [bi for bi in self._balances if bi['network'] == network and
                                bi['account_id'] == account_id]
            if old_balance_item:
                item_n = self._balances.index(old_balance_item[0])
                self._balances[item_n] = new_balance
            else:
                self._balances.append(new_balance)
        return balance

    def balance(self, account_id=None, network=None, as_string=False):
        """
        Get total of unspent outputs

        :param account_id: Account ID filter
        :type account_id: int
        :param network: Network name. Leave empty for default network
        :type network: str
        :param as_string: Set True to return a string in currency format. Default returns float.
        :type as_string: boolean

        :return float, str: Key balance
        """
```

```
        self._balance_update(account_id, network)
        network, account_id, _ = self._get_account_defaults(network, account_id)

        balance = 0
        b_res = [b['balance'] for b in self._balances if b['account_id'] == account_id and b['network'] ==
        if len(b_res):
            balance = b_res[0]
        if as_string:
            return Value.from_satoshi(balance, network=network).str_unit()
        else:
            return float(balance)

    def _balance_update(self, account_id=None, network=None, key_id=None, min_confirms=0):
        """
        Update balance from UTXO's in database. To get most recent balance use :func:`utxos_update` first.

        Also updates balance of wallet and keys in this wallet for the specified account or all accounts i
        no account is specified.

        :param account_id: Account ID filter
        :type account_id: int
        :param network: Network name. Leave empty for default network
        :type network: str
        :param key_id: Key ID Filter
        :type key_id: int
        :param min_confirms: Minimal confirmations needed to include in balance (default = 0)
        :type min_confirms: int

        :return: Updated balance
        """

        qr = self._session.query(DbTransactionOutput, func.sum(DbTransactionOutput.value), DbTransaction.n
                                 DbTransaction.account_id).\
            join(DbTransaction). \
            filter(DbTransactionOutput.spent.is_(False),
                   DbTransaction.wallet_id == self.wallet_id,
                   DbTransaction.confirmations >= min_confirms)
        if account_id is not None:
            qr = qr.filter(DbTransaction.account_id == account_id)
        if network is not None:
            qr = qr.filter(DbTransaction.network_name == network)
        if key_id is not None:
            qr = qr.filter(DbTransactionOutput.key_id == key_id)
        else:
            qr = qr.filter(DbTransactionOutput.key_id.isnot(None))
        utxos = qr.group_by(
            DbTransactionOutput.key_id,
            DbTransactionOutput.transaction_id,
            DbTransactionOutput.output_n,
            DbTransaction.network_name,
            DbTransaction.account_id
        ).all()

        key_values = [
            {
                'id': utxo[0].key_id,
                'network': utxo[2],
                'account_id': utxo[3],
                'balance': utxo[1]
            }
            for utxo in utxos
```

```
        ]

        grouper = itemgetter("id", "network", "account_id")
        key_balance_list = []
        for key, grp in groupby(sorted(key_values, key=grouper), grouper):
            nw_acc_dict = dict(zip(["id", "network", "account_id"], key))
            nw_acc_dict["balance"] = sum(item["balance"] for item in grp)
            key_balance_list.append(nw_acc_dict)

        grouper = itemgetter("network", "account_id")
        balance_list = []
        for key, grp in groupby(sorted(key_balance_list, key=grouper), grouper):
            nw_acc_dict = dict(zip(["network", "account_id"], key))
            nw_acc_dict["balance"] = sum(item["balance"] for item in grp)
            balance_list.append(nw_acc_dict)

        # Add keys with no UTXO's with 0 balance
        for key in self.keys(account_id=account_id, network=network, key_id=key_id):
            if key.id not in [utxo[0].key_id for utxo in utxos]:
                key_balance_list.append({
                    'id': key.id,
                    'network': network,
                    'account_id': key.account_id,
                    'balance': 0
                })

        if not key_id:
            for bl in balance_list:
                bl_item = [b for b in self._balances if
                           b['network'] == bl['network'] and b['account_id'] == bl['account_id']]
                if not bl_item:
                    self._balances.append(bl)
                    continue
                lx = self._balances.index(bl_item[0])
                self._balances[lx].update(bl)

        self._balance = sum([b['balance'] for b in balance_list if b['network'] == self.network.name])

        # Bulk update database
        for kb in key_balance_list:
            if kb['id'] in self._key_objects:
                self._key_objects[kb['id']]._balance = kb['balance']
        self._session.bulk_update_mappings(DbKey, key_balance_list)
        self._commit()
        _logger.info("Got balance for %d key(s)" % len(key_balance_list))
        return self._balances

    def utxos_update(self, account_id=None, used=None, networks=None, key_id=None, depth=None, change=None
                     utxos=None, update_balance=True, max_utxos=MAX_TRANSACTIONS, rescan_all=True):
        """
        Update UTXO's (Unspent Outputs) for addresses/keys in this wallet using various Service providers.

        This method does not import transactions: use :func:`transactions_update` function or to look for
use :func:`scan`.

        :param account_id: Account ID
        :type account_id: int
        :param used: Only check for UTXO for used or unused keys. Default is both
        :type used: bool
         :param networks: Network name filter as string or list of strings. Leave empty to update all use
wallet
```

```
        :type networks: str, list
        :param key_id: Key ID to just update 1 key
        :type key_id: int
        :param depth: Only update keys with this depth, default is depth 5 according to BIP0048 standard.
None to update all keys of this wallet.
        :type depth: int
        :param change: Only update change or normal keys, default is both (None)
        :type change: int
        :param utxos: List of unspent outputs in dictionary format specified below. For usage on an offlir
import utxos with the utxos parameter as a list of dictionaries
        :type utxos: list of dict.

        .. code-block:: json

            {
              "address": "n2S9Czehjvdmpwd2YqekxuUC1Tz5ZdK3YN",
              "script": "",
              "confirmations": 10,
              "output_n": 1,
              "txid": "9df91f89a3eb4259ce04af66ad4caf3c9a297feea5e0b3bc506898b6728c5003",
              "value": 8970937
            }

        :param update_balance: Option to disable balance update after fetching UTXO's. Can be used when
method is called several times in a row. Default is True
        :type update_balance: bool
        :param max_utxos: Maximum number of UTXO's to update
        :type max_utxos: int
        :param rescan_all: Remove old utxo's and rescan wallet. Default is True. Set to False if you wo
utxo's sets. Value will be ignored if key_id is specified in your call
        :type rescan_all: bool

        :return int: Number of new UTXO's added
        """

        _, account_id, acckey = self._get_account_defaults('', account_id, key_id)

        single_key = None
        if key_id:
            single_key = self._session.query(DbKey).filter_by(id=key_id).scalar()
            networks = [single_key.network_name]
            account_id = single_key.account_id
            rescan_all = False
        if networks is None:
            networks = self.network_list()
        elif not isinstance(networks, list):
            networks = [networks]
        elif len(networks) != 1 and utxos is not None:
            raise WalletError("Please specify maximum 1 network when passing utxo's")

        count_utxos = 0
        for network in networks:
            # Remove current UTXO's
            if rescan_all:
                cur_utxos = self._session.query(DbTransactionOutput). \
                    join(DbTransaction). \
                    filter(DbTransactionOutput.spent.is_(False),
                           DbTransaction.account_id == account_id,
                           DbTransaction.wallet_id == self.wallet_id,
                           DbTransaction.network_name == network).all()
                for u in cur_utxos:
```

```
            self._session.query(DbTransactionOutput).filter_by(
                transaction_id=u.transaction_id, output_n=u.output_n).update({DbTransactionOutput.
        self._commit()

    if account_id is None and not self.multisig:
        accounts = self.accounts(network=network)
    else:
        accounts = [account_id]
    for account_id in accounts:
        if depth is None:
            depth = self.key_depth
        if utxos is None:
            # Get all UTXO's for this wallet from default Service object
            addresslist = self.addresslist(account_id=account_id, used=used, network=network, key_
                                           change=change, depth=depth)
            random.shuffle(addresslist)
            srv = Service(network=network, providers=self.providers, cache_uri=self.db_cache_uri)
            srv = Service(network=network, providers=self.providers, cache_uri=self.db_cache_uri)
            utxos = []
            for address in addresslist:
                if rescan_all:
                    last_txid = ''
                else:
                    last_txid = self.utxo_last(address)
                new_utxos = srv.getutxos(address, after_txid=last_txid, limit=max_utxos)
                if new_utxos:
                    utxos += new_utxos
                elif new_utxos is False:
                    raise WalletError("No response from any service provider, could not update UTX
                                      "Errors: %s" % srv.errors)
            if srv.complete:
                self.last_updated = datetime.now()
            elif utxos and 'date' in utxos[-1:][0]:
                self.last_updated = utxos[-1:][0]['date']

        # If UTXO is new, add to database otherwise update depth (confirmation count)
        for utxo in utxos:
            key = single_key
            if not single_key:
                key = self._session.query(DbKey).\
                    filter_by(wallet_id=self.wallet_id, address=utxo['address']).scalar()
            if not key:
                raise WalletError("Key with address %s not found in this wallet" % utxo['address']
            key.used = True
            status = 'unconfirmed'
            if utxo['confirmations']:
                status = 'confirmed'

            # Update confirmations in db if utxo was already imported
            transaction_in_db = self._session.query(DbTransaction).\
                filter_by(wallet_id=self.wallet_id, txid=bytes.fromhex(utxo['txid']),
                          network_name=network)
            utxo_in_db = self._session.query(DbTransactionOutput).join(DbTransaction).\
                filter(DbTransaction.wallet_id == self.wallet_id,
                       DbTransaction.txid == bytes.fromhex(utxo['txid']),
                       DbTransactionOutput.output_n == utxo['output_n'])
            spent_in_db = self._session.query(DbTransactionInput).join(DbTransaction).\
                filter(DbTransaction.wallet_id == self.wallet_id,
                       DbTransactionInput.prev_txid == bytes.fromhex(utxo['txid']),
                       DbTransactionInput.output_n == utxo['output_n'])
            if utxo_in_db.count():
```

```
                        utxo_record = utxo_in_db.scalar()
                        if not utxo_record.key_id:
                            count_utxos += 1
                        utxo_record.key_id = key.id
                        utxo_record.spent = bool(spent_in_db.count())
                        if transaction_in_db.count():
                            transaction_record = transaction_in_db.scalar()
                            transaction_record.confirmations = utxo['confirmations']
                            transaction_record.status = status
                    else:
                        # Add transaction if not exist and then add output
                        if not transaction_in_db.count():
                            block_height = None
                            if block_height in utxo and utxo['block_height']:
                                block_height = utxo['block_height']
                            new_tx = DbTransaction(
                                wallet_id=self.wallet_id, txid=bytes.fromhex(utxo['txid']), status=status,
                                is_complete=False, block_height=block_height, account_id=account_id,
                                confirmations=utxo['confirmations'], network_name=network)
                            self._session.add(new_tx)
                            # TODO: Get unique id before inserting to increase performance for large utxo-
                            self._commit()
                            tid = new_tx.id
                        else:
                            tid = transaction_in_db.scalar().id

                        script_type = script_type_default(self.witness_type, multisig=self.multisig,
                                                          locking_script=True)
                        new_utxo = DbTransactionOutput(transaction_id=tid,  output_n=utxo['output_n'],
                                                       value=utxo['value'], key_id=key.id, address=utxo['a
                                                       script=bytes.fromhex(utxo['script']),
                                                       script_type=script_type,
                                                       spent=bool(spent_in_db.count()))
                        self._session.add(new_utxo)
                        count_utxos += 1

                    self._commit()

                _logger.info("Got %d new UTXOs for account %s" % (count_utxos, account_id))
                self._commit()
                if update_balance:
                    self._balance_update(account_id=account_id, network=network, key_id=key_id, min_confir
                utxos = None
        return count_utxos

    def utxos(self, account_id=None, network=None, min_confirms=0, key_id=None):
        """
        Get UTXO's (Unspent Outputs) from database. Use :func:`utxos_update` method first for updated valu

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> w.utxos()  # doctest:+SKIP
           [{'value': 100000000, 'script': '', 'output_n': 0, 'transaction_id': ..., 'spent': False, '
'p2pkh',    'key_id':    ...,    'address':    '16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg',    'confirmations':
'748799c9047321cb27a6320a827f1f69d767fe889c14bf11f27549638d566fe4', 'network_name': 'bitcoin'}]

        :param account_id: Account ID
        :type account_id: int
        :param network: Network name. Leave empty for default network
        :type network: str
        :param min_confirms: Minimal confirmation needed to include in output list
        :type min_confirms: int
```

```
        :param key_id: Key ID or list of key IDs to filter utxo's for specific keys
        :type key_id: int, list

        :return list: List of transactions
        """

        first_key_id = key_id
        if isinstance(key_id, list):
            first_key_id = key_id[0]
        network, account_id, acckey = self._get_account_defaults(network, account_id, first_key_id)

        qr = self._session.query(DbTransactionOutput, DbKey.address, DbTransaction.confirmations, DbTransa
                                 DbKey.network_name).\
            join(DbTransaction).join(DbKey). \
            filter(DbTransactionOutput.spent.is_(False),
                   DbTransaction.account_id == account_id,
                   DbTransaction.wallet_id == self.wallet_id,
                   DbTransaction.network_name == network,
                   DbTransaction.confirmations >= min_confirms)
        if isinstance(key_id, int):
            qr = qr.filter(DbKey.id == key_id)
        elif isinstance(key_id, list):
            qr = qr.filter(DbKey.id.in_(key_id))
        utxos = qr.order_by(DbTransaction.confirmations.desc()).all()
        res = []
        for utxo in utxos:
            u = utxo[0].__dict__
            if '_sa_instance_state' in u:
                del u['_sa_instance_state']
            u['address'] = utxo[1]
            u['confirmations'] = int(utxo[2])
            u['txid'] = utxo[3].hex()
            u['network_name'] = utxo[4]
            res.append(u)
        return res

    def utxo_add(self, address, value, txid, output_n, confirmations=1, script=''):
        """
        Add a single UTXO to the wallet database. To update all utxo's use :func:`utxos_update` method.

        Use this method for testing, offline wallets or if you wish to override standard method of retreiv

        This method does not check if UTXO exists or is still spendable.

        :param address: Address of Unspent Output. Address should be available in wallet
        :type address: str
        :param value: Value of output in sathosis or smallest denominator for type of currency
        :type value: int
        :param txid: Transaction hash or previous output as hex-string
        :type txid: str
        :param output_n: Output number of previous transaction output
        :type output_n: int
        :param confirmations: Number of confirmations. Default is 0, unconfirmed
        :type confirmations: int
        :param script: Locking script of previous output as hex-string
        :type script: str

        :return int: Number of new UTXO's added, so 1 if successful
        """

        utxo = {
```

```
            'address': address,
            'script': script,
            'confirmations': confirmations,
            'output_n': output_n,
            'txid': txid,
            'value': value
        }
        return self.utxos_update(utxos=[utxo])

    def utxo_last(self, address):
        """
        Get transaction ID for latest utxo in database for given address

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> w.utxo_last('16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg')
        '748799c9047321cb27a6320a827f1f69d767fe889c14bf11f27549638d566fe4'

        :param address: The address
        :type address: str

        :return str:
        """
        to = self._session.query(
            DbTransaction.txid, DbTransaction.confirmations). \
            join(DbTransactionOutput).join(DbKey). \
            filter(DbKey.address == address, DbTransaction.wallet_id == self.wallet_id,
                   DbTransactionOutput.spent.is_(False)). \
            order_by(DbTransaction.confirmations).first()
        return '' if not to else to[0].hex()

    def transactions_update_confirmations(self):
        """
        Update number of confirmations and status for transactions in database

        :return:
        """
        network = self.network.name
        srv = Service(network=network, providers=self.providers, cache_uri=self.db_cache_uri)
        blockcount = srv.blockcount()
        db_txs = self._session.query(DbTransaction). \
            filter(DbTransaction.wallet_id == self.wallet_id,
                   DbTransaction.network_name == network, DbTransaction.block_height > 0).all()
        for db_tx in db_txs:
            self._session.query(DbTransaction).filter_by(id=db_tx.id). \
                update({DbTransaction.status: 'confirmed',
                        DbTransaction.confirmations: (blockcount - DbTransaction.block_height) + 1})
        self._commit()

    def transactions_update_by_txids(self, txids):
        """
        Update transaction or list or transaction for this wallet with provided transaction ID

        :param txids: Transaction ID, or list of transaction IDs
        :type txids: str, list of str, bytes, list of bytes

        :return:
        """
        if not isinstance(txids, list):
            txids = [txids]
        txids = list(dict.fromkeys(txids))
```

```
        txs = []
        srv = Service(network=self.network.name, providers=self.providers, cache_uri=self.db_cache_uri)
        for txid in txids:
            tx = srv.gettransaction(to_hexstring(txid))
            if tx:
                txs.append(tx)

        # TODO: Avoid duplicate code in this method and transaction_update()
        utxo_set = set()
        for t in txs:
            wt = WalletTransaction.from_transaction(self, t)
            wt.store()
            utxos = [(ti.prev_txid.hex(), ti.output_n_int) for ti in wt.inputs]
            utxo_set.update(utxos)

        for utxo in list(utxo_set):
            tos = self._session.query(DbTransactionOutput).join(DbTransaction). \
                filter(DbTransaction.txid == bytes.fromhex(utxo[0]), DbTransactionOutput.output_n == utxo[
                       DbTransactionOutput.spent.is_(False)).all()
            for u in tos:
                u.spent = True
        self._commit()
        # self._balance_update(account_id=account_id, network=network, key_id=key_id)

    def transactions_update(self, account_id=None, used=None, network=None, key_id=None, depth=None, chang
                            limit=MAX_TRANSACTIONS):
        """
          Update wallets transaction from service providers. Get all transactions for known keys in thi
balances and unspent outputs (UTXO's) are updated as well. Only scan keys from default network and a
another network or account is specified.

           Use the :func:`scan` method for automatic address generation/management, and use the :func:`
method to only look for unspent outputs and balances.

        :param account_id: Account ID
        :type account_id: int
        :param used: Only update used or unused keys, specify None to update both. Default is None
        :type used: bool, None
        :param network: Network name. Leave empty for default network
        :type network: str
        :param key_id: Key ID to just update 1 key
        :type key_id: int
        :param depth: Only update keys with this depth, default is depth 5 according to BIP0048 standard.
None to update all keys of this wallet.
        :type depth: int
        :param change: Only update change or normal keys, default is both (None)
        :type change: int
           :param limit: Stop update after limit transactions to avoid timeouts with service providers
MAX_TRANSACTIONS defined in config.py
        :type limit: int

        :return bool: True if all transactions are updated
        """

        network, account_id, acckey = self._get_account_defaults(network, account_id, key_id)
        if depth is None:
            depth = self.key_depth

        # Update number of confirmations and status for already known transactions
        if not key_id:
            self.transactions_update_confirmations()
```

```
        srv = Service(network=network, providers=self.providers, cache_uri=self.db_cache_uri)
        blockcount = srv.blockcount()
        db_txs = self._session.query(DbTransaction).\
            filter(DbTransaction.wallet_id == self.wallet_id,
                   DbTransaction.network_name == network, DbTransaction.block_height > 0).all()
        for db_tx in db_txs:
            self._session.query(DbTransaction).filter_by(id=db_tx.id).\
                update({DbTransaction.status: 'confirmed',
                        DbTransaction.confirmations: (blockcount - DbTransaction.block_height) + 1})
        self._commit()

        # Get transactions for wallet's addresses
        txs = []
        addresslist = self.addresslist(
            account_id=account_id, used=used, network=network, key_id=key_id, change=change, depth=depth)
        last_updated = datetime.now()
        for address in addresslist:
            txs += srv.gettransactions(address, limit=limit, after_txid=self.transaction_last(address))
            if not srv.complete:
                if txs and txs[-1].date and txs[-1].date < last_updated:
                    last_updated = txs[-1].date
            if txs and txs[-1].confirmations:
                dbkey = self._session.query(DbKey).filter(DbKey.address == address, DbKey.wallet_id == sel
                if not dbkey.update({DbKey.latest_txid: bytes.fromhex(txs[-1].txid)}):
                    raise WalletError("Failed to update latest transaction id for key with address %s" % a
                self._commit()
        if txs is False:
            raise WalletError("No response from any service provider, could not update transactions")

        # Update Transaction outputs to get list of unspent outputs (UTXO's)
        utxo_set = set()
        for t in txs:
            wt = WalletTransaction.from_transaction(self, t)
            wt.store()
            utxos = [(ti.prev_txid.hex(), ti.output_n_int) for ti in wt.inputs]
            utxo_set.update(utxos)
        for utxo in list(utxo_set):
            tos = self._session.query(DbTransactionOutput).join(DbTransaction).\
                filter(DbTransaction.txid == bytes.fromhex(utxo[0]), DbTransactionOutput.output_n == utxo[
                       DbTransactionOutput.spent.is_(False), DbTransaction.wallet_id == self.wallet_id).al
            for u in tos:
                u.spent = True

        self.last_updated = last_updated
        self._commit()
        self._balance_update(account_id=account_id, network=network, key_id=key_id)

        return len(txs)

    def transaction_last(self, address):
        """
        Get transaction ID for latest transaction in database for given address

        :param address: The address
        :type address: str

        :return str:
        """
        txid = self._session.query(DbKey.latest_txid).\
            filter(DbKey.address == address, DbKey.wallet_id == self.wallet_id).scalar()
```

```
    return '' if not txid else txid.hex()

def transactions(self, account_id=None, network=None, include_new=False, key_id=None, as_dict=False):
    """
    Get all known transactions input and outputs for this wallet.

    The transaction only includes the inputs and outputs related to this wallet. To get full transacti
    use the :func:`transactions_full` method.

    >>> w = Wallet('bitcoinlib_legacy_wallet_test')
    >>> w.transactions()
    [<WalletTransaction(input_count=0, output_count=1, status=confirmed, network=bitcoin)>]

    :param account_id: Filter by Account ID. Leave empty for default account_id
    :type account_id: int, None
    :param network: Filter by network name. Leave empty for default network
    :type network: str, None
    :param include_new: Also include new and incomplete transactions in list. Default is False
    :type include_new: bool
    :param key_id: Filter by key ID
    :type key_id: int, None
    :param as_dict: Output as dictionary or WalletTransaction object
    :type as_dict: bool

    :return list of WalletTransaction: List of WalletTransaction or transactions as dictionary
    """

    network, account_id, acckey = self._get_account_defaults(network, account_id, key_id)
    # Transaction inputs
    qr = self._session.query(DbTransactionInput, DbTransactionInput.address, DbTransaction.confirmatic
                             DbTransaction.txid, DbTransaction.network_name, DbTransaction.status). \
        join(DbTransaction).join(DbKey). \
        filter(DbTransaction.account_id == account_id,
               DbTransaction.wallet_id == self.wallet_id,
               DbKey.wallet_id == self.wallet_id,
               DbTransaction.network_name == network)
    if key_id is not None:
        qr = qr.filter(DbTransactionInput.key_id == key_id)
    if not include_new:
        qr = qr.filter(or_(DbTransaction.status == 'confirmed', DbTransaction.status == 'unconfirmed')
    txs = qr.all()
    # Transaction outputs
    qr = self._session.query(DbTransactionOutput, DbTransactionOutput.address, DbTransaction.confirmat
                             DbTransaction.txid, DbTransaction.network_name, DbTransaction.status). \
        join(DbTransaction).join(DbKey). \
        filter(DbTransaction.account_id == account_id,
               DbTransaction.wallet_id == self.wallet_id,
               DbKey.wallet_id == self.wallet_id,
               DbTransaction.network_name == network)
    if key_id is not None:
        qr = qr.filter(DbTransactionOutput.key_id == key_id)
    if not include_new:
        qr = qr.filter(or_(DbTransaction.status == 'confirmed', DbTransaction.status == 'unconfirmed')
    txs += qr.all()

    txs = sorted(txs, key=lambda k: (k[2], pow(10, 20)-k[0].transaction_id, k[3]), reverse=True)
    res = []
    txids = []
    for tx in txs:
        txid = tx[3].hex()
        if as_dict:
```

```
            u = tx[0].__dict__
            u['block_height'] = tx[0].transaction.block_height
            u['date'] = tx[0].transaction.date
            if '_sa_instance_state' in u:
                del u['_sa_instance_state']
            u['address'] = tx[1]
            u['confirmations'] = None if tx[2] is None else int(tx[2])
            u['txid'] = txid
            u['network_name'] = tx[4]
            u['status'] = tx[5]
            if 'index_n' in u:
                u['is_output'] = True
                u['value'] = -u['value']
            else:
                u['is_output'] = False
        else:
            if txid in txids:
                continue
            txids.append(txid)
            u = self.transaction(txid)
        res.append(u)
    return res

def transactions_full(self, network=None, include_new=False, limit=0, offset=0):
    """
    Get all transactions of this wallet as WalletTransaction objects

    Use the :func:`transactions` method to only get the inputs and outputs transaction parts related t

    :param network: Filter by network name. Leave empty for default network
    :type network: str
    :param include_new: Also include new and incomplete transactions in list. Default is False
    :type include_new: bool
    :param limit: Maximum number of transactions to return. Combine with offset parameter to use as pa
    :type limit: int
    :param offset: Number of transactions to skip
    :type offset: int

    :return list of WalletTransaction:
    """
    network, _, _ = self._get_account_defaults(network)
    qr = self._session.query(DbTransaction.txid, DbTransaction.network_name, DbTransaction.status). \
        filter(DbTransaction.wallet_id == self.wallet_id,
               DbTransaction.network_name == network)
    if not include_new:
        qr = qr.filter(or_(DbTransaction.status == 'confirmed', DbTransaction.status == 'unconfirmed')
    txs = []
    if limit:
        qr = qr.limit(limit)
    if offset:
        qr = qr.offset(offset)
    for tx in qr.all():
        txs.append(self.transaction(tx[0].hex()))
    return txs

def transactions_export(self, account_id=None, network=None, include_new=False, key_id=None, skip_char
    """
    Export wallets transactions as list of tuples with the following fields:
        (transaction_date, transaction_hash, in/out, addresses_in, addresses_out, value, value_cumulat

    :param account_id: Filter by Account ID. Leave empty for default account_id
```

```
        :type account_id: int, None
        :param network: Filter by network name. Leave empty for default network
        :type network: str, None
        :param include_new: Also include new and incomplete transactions in list. Default is False
        :type include_new: bool
        :param key_id: Filter by key ID
        :type key_id: int, None
        :param skip_change: Do not include change outputs. Default is True
        :type skip_change: bool

        :return list of tuple:
        """

        txs_tuples = []
        cumulative_value = 0
        for t in self.transactions(account_id, network, include_new, key_id):
            te = t.export(skip_change=skip_change)

            # When transaction is outgoing deduct fee from cumulative value
            if t.outgoing_tx:
                cumulative_value -= t.fee

            # Loop through all transaction inputs and outputs
            for tei in te:
                # Create string with  list of inputs addresses for incoming transactions, and outputs addr
                # for outgoing txs
                addr_list_in = tei[3] if isinstance(tei[3], list) else [tei[3]]
                addr_list_out = tei[4] if isinstance(tei[4], list) else [tei[4]]
                cumulative_value += tei[5]
                txs_tuples.append((tei[0], tei[1], tei[2], addr_list_in, addr_list_out, tei[5], cumulative
                                   tei[6]))
        return txs_tuples

    def transaction(self, txid):
        """
        Get WalletTransaction object for given transaction ID (transaction hash)

        :param txid: Hexadecimal transaction hash
        :type txid: str

        :return WalletTransaction:
        """
        return WalletTransaction.from_txid(self, txid)

    def transaction_spent(self, txid, output_n):
        """
        Check if transaction with given transaction ID and output_n is spent and return txid of spent tran

         Retrieves information from database, does not update transaction and does not check if transac
with service providers.

        :param txid: Hexadecimal transaction hash
        :type txid: str, bytes
        :param output_n: Output n
        :type output_n: int, bytes

        :return str: Transaction ID
        """
        txid = to_bytes(txid)
        if isinstance(output_n, bytes):
            output_n = int.from_bytes(output_n, 'big')
```

```
        qr = self._session.query(DbTransactionInput, DbTransaction.confirmations,
                                 DbTransaction.txid, DbTransaction.status). \
            join(DbTransaction). \
            filter(DbTransaction.wallet_id == self.wallet_id,
                   DbTransactionInput.prev_txid == txid, DbTransactionInput.output_n == output_n).scalar()
        if qr:
            return qr.transaction.txid.hex()

    def _objects_by_key_id(self, key_id):
        key = self._session.query(DbKey).filter_by(id=key_id).scalar()
        if not key:
            raise WalletError("Key '%s' not found in this wallet" % key_id)
        if key.key_type == 'multisig':
            inp_keys = [HDKey.from_wif(ck.child_key.wif, network=ck.child_key.network_name) for ck in
                        key.multisig_children]
        elif key.key_type in ['bip32', 'single']:
            if not key.wif:
                raise WalletError("WIF of key is empty cannot create HDKey")
            inp_keys = [HDKey.from_wif(key.wif, network=key.network_name)]
        else:
            raise WalletError("Input key type %s not supported" % key.key_type)
        return inp_keys, key

    def select_inputs(self, amount, variance=None, input_key_id=None, account_id=None, network=None, min_c
                      max_utxos=None, return_input_obj=True, skip_dust_amounts=True):
        """
        Select available unspent transaction outputs (UTXO's) which can be used as inputs for a transactio
        the specified amount.

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> w.select_inputs(50000000)
                    [<Input(prev_txid='748799c9047321cb27a6320a827f1f69d767fe889c14bf11f27549638d566fe4',
address='16QaHuFkfuebXGcYHmehRXBBX7RG9NbtLg', index_n=0, type='sig_pubkey')>]

        :param amount: Total value of inputs in the smallest denominator (sathosi) to select
        :type amount: int
           :param variance: Allowed difference in total input value. Default is dust amount of sele
Difference will be added to transaction fee.
        :type variance: int
         :param input_key_id: Limit UTXO's search for inputs to this key ID or list of key IDs. Only vali
array is specified
        :type input_key_id: int, list
        :param account_id: Account ID
        :type account_id: int
        :param network: Network name. Leave empty for default network
        :type network: str
        :param min_confirms: Minimal confirmation needed for an UTXO before it will be included in inputs.
confirmation. Option is ignored if input_arr is provided.
        :type min_confirms: int
        :param max_utxos: Maximum number of UTXO's to use. Set to 1 for optimal privacy. Default is None:
        :type max_utxos: int
        :param return_input_obj: Return inputs as Input class object. Default is True
        :type return_input_obj: bool
        :param skip_dust_amounts: Do not include small amount to avoid dust inputs
        :type skip_dust_amounts: bool

        :return: List of previous outputs
        :rtype: list of DbTransactionOutput, list of Input
        """

        network, account_id, _ = self._get_account_defaults(network, account_id)
```

```
        dust_amount = Network(network).dust_amount
        if variance is None:
            variance = dust_amount

        utxo_query = self._session.query(DbTransactionOutput).join(DbTransaction).join(DbKey). \
            filter(DbTransaction.wallet_id == self.wallet_id, DbTransaction.account_id == account_id,
                   DbTransaction.network_name == network, DbKey.public != b'',
                   DbTransactionOutput.spent.is_(False), DbTransaction.confirmations >= min_confirms)
        if input_key_id:
            if isinstance(input_key_id, int):
                utxo_query = utxo_query.filter(DbKey.id == input_key_id)
            else:
                utxo_query = utxo_query.filter(DbKey.id.in_(input_key_id))
        if skip_dust_amounts:
            utxo_query = utxo_query.filter(DbTransactionOutput.value > dust_amount)
        utxos = utxo_query.order_by(DbTransaction.confirmations.desc()).all()
        if not utxos:
                raise WalletError("Create transaction: No unspent transaction outputs found or no key
UTXO's")

        # TODO: Find 1 or 2 UTXO's with exact amount +/- self.network.dust_amount

        # Try to find one utxo with exact amount
        one_utxo = utxo_query.filter(DbTransactionOutput.spent.is_(False),
                                     DbTransactionOutput.value >= amount,
                                     DbTransactionOutput.value <= amount + variance).first()
        selected_utxos = []
        if one_utxo:
            selected_utxos = [one_utxo]
        else:
            # Try to find one utxo with higher amount
            one_utxo = utxo_query. \
                filter(DbTransactionOutput.spent.is_(False), DbTransactionOutput.value >= amount).\
                order_by(DbTransactionOutput.value).first()
            if one_utxo:
                selected_utxos = [one_utxo]
            elif max_utxos and max_utxos <= 1:
                _logger.info("No single UTXO found with requested amount, use higher 'max_utxo' setting to
                             "multiple UTXO's")
                return []

        # Otherwise compose of 2 or more lesser outputs
        if not selected_utxos:
            lessers = utxo_query. \
                filter(DbTransactionOutput.spent.is_(False), DbTransactionOutput.value < amount).\
                order_by(DbTransactionOutput.value.desc()).all()
            total_amount = 0
            selected_utxos = []
            for utxo in lessers[:max_utxos]:
                if total_amount < amount:
                    selected_utxos.append(utxo)
                    total_amount += utxo.value
            if total_amount < amount:
                return []
        if not return_input_obj:
            return selected_utxos
        else:
            inputs = []
            for utxo in selected_utxos:
                # amount_total_input += utxo.value
                inp_keys, key = self._objects_by_key_id(utxo.key_id)
```

```
                multisig = False if len(inp_keys) < 2 else True
                script_type = get_unlocking_script_type(utxo.script_type, multisig=multisig)
                inputs.append(Input(utxo.transaction.txid, utxo.output_n, keys=inp_keys, script_type=scrip
                              sigs_required=self.multisig_n_required, sort=self.sort_keys, address=key.adc
                              compressed=key.compressed, value=utxo.value, network=key.network_name))
            return inputs

        def transaction_create(self, output_arr, input_arr=None, input_key_id=None, account_id=None,
fee=None,
                               min_confirms=1, max_utxos=None, locktime=0, number_of_change_outputs=1,
                               random_output_order=True):
        """
        Create new transaction with specified outputs.

               Inputs can be specified but if not provided they will be selected from wallets
:func:`select_inputs` method.

        Output array is a list of 1 or more addresses and amounts.

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> t = w.transaction_create([('1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb', 200000)])
        >>> t
        <WalletTransaction(input_count=1, output_count=2, status=new, network=bitcoin)>
        >>> t.outputs # doctest:+ELLIPSIS
        [<Output(value=..., address=..., type=p2pkh)>, <Output(value=..., address=..., type=p2pkh)>]

         :param output_arr: List of output as Output objects or tuples with address and amount. Must con
one item. Example: [('mxdLD8SAGS9fe2EeCXALDHcdTTbppMHp8N', 5000000)]
        :type output_arr: list of Output, tuple
         :param input_arr: List of inputs as Input objects or tuples with reference to a UTXO, a wallet k
The format is [(txid, output_n, key_ids, value, signatures, unlocking_script, address)]
        :type input_arr: list of Input, tuple
        :param input_key_id: Limit UTXO's search for inputs to this key_id. Only valid if no input array i
        :type input_key_id: int
        :param account_id: Account ID
        :type account_id: int
        :param network: Network name. Leave empty for default network
        :type network: str
         :param fee: Set fee manually, leave empty to calculate fees automatically. Set fees in the smal
denominator, for example satoshi's if you are using bitcoins. You can also supply a string: 'low', 'norm
to determine fees automatically.
        :type fee: int, str
        :param min_confirms: Minimal confirmation needed for an UTXO before it will be included in inputs.
confirmation. Option is ignored if input_arr is provided.
        :type min_confirms: int
        :param max_utxos: Maximum number of UTXO's to use. Set to 1 for optimal privacy. Default is None:
        :type max_utxos: int
         :param locktime: Transaction level locktime. Locks the transaction until a specified block (valu
million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transac
locktime
        :type locktime: int
        :param number_of_change_outputs: Number of change outputs to create when there is a change value.
Use 0 for random number of outputs: between 1 and 5 depending on send and change amount
number_of_change_outputs: int
        :type number_of_change_outputs: int
        :param random_output_order: Shuffle order of transaction outputs to increase privacy. Default is T
        :type random_output_order: bool

        :return WalletTransaction: object
        """
```

```
if not isinstance(output_arr, list):
    raise WalletError("Output array must be a list of tuples with address and amount. "
                      "Use 'send_to' method to send to one address")
if not network and output_arr:
    if isinstance(output_arr[0], Output):
        network = output_arr[0].network.name
    elif isinstance(output_arr[0][1], str):
        network = Value(output_arr[0][1]).network.name
network, account_id, acckey = self._get_account_defaults(network, account_id)

if input_arr and max_utxos and len(input_arr) > max_utxos:
    raise WalletError("Input array contains %d UTXO's but max_utxos=%d parameter specified" %
                      (len(input_arr), max_utxos))

# Create transaction and add outputs
amount_total_output = 0
transaction = WalletTransaction(hdwallet=self, account_id=account_id, network=network, locktime=lo
transaction.outgoing_tx = True
for o in output_arr:
    if isinstance(o, Output):
        transaction.outputs.append(o)
        amount_total_output += o.value
    else:
        value = value_to_satoshi(o[1], network=transaction.network)
        amount_total_output += value
        addr = o[0]
        if isinstance(addr, WalletKey):
            addr = addr.key()
        transaction.add_output(value, addr)

srv = Service(network=network, providers=self.providers, cache_uri=self.db_cache_uri)
transaction.fee_per_kb = None
if isinstance(fee, int):
    fee_estimate = fee
else:
    n_blocks = 3
    priority = ''
    if isinstance(fee, str):
        priority = fee
    transaction.fee_per_kb = srv.estimatefee(blocks=n_blocks, priority=priority)
    if not input_arr:
        fee_estimate = int(transaction.estimate_size(number_of_change_outputs=number_of_change_out
                           1000.0 * transaction.fee_per_kb)
    else:
        fee_estimate = 0
    if isinstance(fee, str):
        fee = fee_estimate

# Add inputs
sequence = 0xffffffff
if 0 < transaction.locktime < 0xffffffff:
    sequence = 0xfffffffe
amount_total_input = 0
if input_arr is None:
    selected_utxos = self.select_inputs(amount_total_output + fee_estimate, transaction.network.du
                                        input_key_id, account_id, network, min_confirms, max_utxos
    if not selected_utxos:
        raise WalletError("Not enough unspent transaction outputs found")
    for utxo in selected_utxos:
        amount_total_input += utxo.value
        inp_keys, key = self._objects_by_key_id(utxo.key_id)
```

```
            multisig = False if isinstance(inp_keys, list) and len(inp_keys) < 2 else True
            unlock_script_type = get_unlocking_script_type(utxo.script_type, self.witness_type, multis
            transaction.add_input(utxo.transaction.txid, utxo.output_n, keys=inp_keys,
                                  script_type=unlock_script_type, sigs_required=self.multisig_n_requir
                                  sort=self.sort_keys, compressed=key.compressed, value=utxo.value,
                                  address=utxo.key.address, sequence=sequence,
                                  key_path=utxo.key.path, witness_type=self.witness_type)
            # FIXME: Missing locktime_cltv=locktime_cltv, locktime_csv=locktime_csv (?)
    else:
        for inp in input_arr:
            locktime_cltv = None
            locktime_csv = None
            unlocking_script_unsigned = None
            unlocking_script_type = ''
            if isinstance(inp, Input):
                prev_txid = inp.prev_txid
                output_n = inp.output_n
                key_id = None
                value = inp.value
                signatures = inp.signatures
                unlocking_script = inp.unlocking_script
                unlocking_script_unsigned = inp.unlocking_script_unsigned
                unlocking_script_type = inp.script_type
                address = inp.address
                sequence = inp.sequence
                locktime_cltv = inp.locktime_cltv
                locktime_csv = inp.locktime_csv
            # elif isinstance(inp, DbTransactionOutput):
            #     prev_txid = inp.transaction.txid
            #     output_n = inp.output_n
            #     key_id = inp.key_id
            #     value = inp.value
            #     signatures = None
            #     # FIXME: This is probably not an unlocking_script
            #     unlocking_script = inp.script
            #     unlocking_script_type = get_unlocking_script_type(inp.script_type)
            #     address = inp.key.address
            else:
                prev_txid = inp[0]
                output_n = inp[1]
                key_id = None if len(inp) <= 2 else inp[2]
                value = 0 if len(inp) <= 3 else inp[3]
                signatures = None if len(inp) <= 4 else inp[4]
                unlocking_script = b'' if len(inp) <= 5 else inp[5]
                address = '' if len(inp) <= 6 else inp[6]
            # Get key_ids, value from Db if not specified
            if not (key_id and value and unlocking_script_type):
                if not isinstance(output_n, TYPE_INT):
                    output_n = int.from_bytes(output_n, 'big')
                inp_utxo = self._session.query(DbTransactionOutput).join(DbTransaction). \
                    filter(DbTransaction.wallet_id == self.wallet_id,
                           DbTransaction.txid == to_bytes(prev_txid),
                           DbTransactionOutput.output_n == output_n).first()
                if inp_utxo:
                    key_id = inp_utxo.key_id
                    value = inp_utxo.value
                    address = inp_utxo.key.address
                    unlocking_script_type = get_unlocking_script_type(inp_utxo.script_type, multisig=s
                    # witness_type = inp_utxo.witness_type
                else:
                    _logger.info("UTXO %s not found in this wallet. Please update UTXO's if this is nc
```

```
                                "offline wallet" % to_hexstring(prev_txid))
                    key_id = self._session.query(DbKey.id).\
                        filter(DbKey.wallet_id == self.wallet_id, DbKey.address == address).scalar()
                    if not key_id:
                        raise WalletError("UTXO %s and key with address %s not found in this wallet" %
                            to_hexstring(prev_txid), address))
                    if not value:
                        raise WalletError("Input value is zero for address %s. Import or update UTXO's
                                          "or import transaction as dictionary" % address)

                amount_total_input += value
                inp_keys, key = self._objects_by_key_id(key_id)
                transaction.add_input(prev_txid, output_n, keys=inp_keys, script_type=unlocking_script_typ
                                      sigs_required=self.multisig_n_required, sort=self.sort_keys,
                                      compressed=key.compressed, value=value, signatures=signatures,
                                      unlocking_script=unlocking_script, address=address,
                                      unlocking_script_unsigned=unlocking_script_unsigned,
                                      sequence=sequence, locktime_cltv=locktime_cltv, locktime_csv=locktim
                                      witness_type=self.witness_type, key_path=key.path)
        # Calculate fees
        transaction.fee = fee
        fee_per_output = None
        transaction.size = transaction.estimate_size(number_of_change_outputs=number_of_change_outputs)
        if fee is None:
            if not input_arr:
                if not transaction.fee_per_kb:
                    transaction.fee_per_kb = srv.estimatefee()
                if transaction.fee_per_kb < transaction.network.fee_min:
                    transaction.fee_per_kb = transaction.network.fee_min
                transaction.fee = int((transaction.size / 1000.0) * transaction.fee_per_kb)
                fee_per_output = int((50 / 1000.0) * transaction.fee_per_kb)
            else:
                if amount_total_output and amount_total_input:
                    fee = False
                else:
                    transaction.fee = 0

        if fee is False:
            transaction.change = 0
            transaction.fee = int(amount_total_input - amount_total_output)
        else:
            transaction.change = int(amount_total_input - (amount_total_output + transaction.fee))

        # Skip change if amount is smaller than the dust limit or estimated fee
                    if   (fee_per_output   and   transaction.change   <   fee_per_output)   or   transactio
transaction.network.dust_amount:
            transaction.fee += transaction.change
            transaction.change = 0
        if transaction.change < 0:
            raise WalletError("Total amount of outputs is greater then total amount of inputs")
        if transaction.change:
            min_output_value = transaction.network.dust_amount * 2 + transaction.network.fee_min * 4
            if transaction.fee and transaction.size:
                if not transaction.fee_per_kb:
                    transaction.fee_per_kb = int((transaction.fee * 1000.0) / transaction.vsize)
                min_output_value = transaction.fee_per_kb + transaction.network.fee_min * 4 + \
                                   transaction.network.dust_amount

            if number_of_change_outputs == 0:
                if transaction.change < amount_total_output / 10 or transaction.change < min_output_value
                    number_of_change_outputs = 1
```

```
            elif transaction.change / 10 > amount_total_output:
                number_of_change_outputs = random.randint(2, 5)
            else:
                number_of_change_outputs = random.randint(1, 3)
                # Prefer 1 and 2 as number of change outputs
                if number_of_change_outputs == 3:
                    number_of_change_outputs = random.randint(3, 4)
            transaction.size = transaction.estimate_size(number_of_change_outputs=number_of_change_out

        average_change = transaction.change // number_of_change_outputs
        if number_of_change_outputs > 1 and average_change < min_output_value:
            raise WalletError("Not enough funds to create multiple change outputs. Try less change out
                              "or lower fees")

        if self.scheme == 'single':
            change_keys = [self.get_key(account_id=account_id, network=network, change=1)]
        else:
            change_keys = self.get_keys(account_id=account_id, network=network, change=1,
                                        number_of_keys=number_of_change_outputs)

        if number_of_change_outputs > 1:
            rand_prop = transaction.change - number_of_change_outputs * min_output_value
            change_amounts = list(((np.random.dirichlet(np.ones(number_of_change_outputs), size=1)[0]
                                    rand_prop) + min_output_value).astype(int))
            # Fix rounding problems / small amount differences
            diffs = transaction.change - sum(change_amounts)
            for idx, co in enumerate(change_amounts):
                if co - diffs > min_output_value:
                    change_amounts[idx] += change_amounts.index(co) + diffs
                    break
        else:
            change_amounts = [transaction.change]

        for idx, ck in enumerate(change_keys):
            on = transaction.add_output(change_amounts[idx], ck.address, encoding=self.encoding)
            transaction.outputs[on].key_id = ck.key_id

    # Shuffle output order to increase privacy
    if random_output_order:
        transaction.shuffle()

    # Check tx values
    transaction.input_total = sum([i.value for i in transaction.inputs])
    transaction.output_total = sum([o.value for o in transaction.outputs])
    if transaction.input_total != transaction.fee + transaction.output_total:
        raise WalletError("Sum of inputs values is not equal to sum of outputs values plus fees")

    transaction.txid = transaction.signature_hash()[::-1].hex()
    if not transaction.fee_per_kb:
        transaction.fee_per_kb = int((transaction.fee * 1000.0) / transaction.vsize)
    if transaction.fee_per_kb < transaction.network.fee_min:
        raise WalletError("Fee per kB of %d is lower then minimal network fee of %d" %
                          (transaction.fee_per_kb, transaction.network.fee_min))
    elif transaction.fee_per_kb > transaction.network.fee_max:
        raise WalletError("Fee per kB of %d is higher then maximum network fee of %d" %
                          (transaction.fee_per_kb, transaction.network.fee_max))

    return transaction

def transaction_import(self, t):
    """
```

```
    Import a Transaction into this wallet. Link inputs to wallet keys if possible and return WalletTra
    object. Only imports Transaction objects or dictionaries, use
    :func:`transaction_import_raw` method to import a raw transaction.

    :param t: A Transaction object or dictionary
    :type t: Transaction, dict

    :return WalletTransaction:

    """
    if isinstance(t, Transaction):
        rt = self.transaction_create(t.outputs, t.inputs, fee=t.fee, network=t.network.name,
                                     random_output_order=False)
        rt.block_height = t.block_height
        rt.confirmations = t.confirmations
        rt.witness_type = t.witness_type
        rt.date = t.date
        rt.txid = t.txid
        rt.txhash = t.txhash
        rt.locktime = t.locktime
        rt.version = t.version
        rt.version_int = t.version_int
        rt.block_hash = t.block_hash
        rt.rawtx = t.rawtx
        rt.coinbase = t.coinbase
        rt.flag = t.flag
        rt.size = t.size
        if not t.size:
            rt.size = len(t.raw())
        rt.vsize = t.vsize
        if not t.vsize:
            rt.vsize = rt.size
        rt.fee_per_kb = int((rt.fee / float(rt.vsize)) * 1000)
    elif isinstance(t, dict):
        input_arr = []
        for i in t['inputs']:
            signatures = [bytes.fromhex(sig) for sig in i['signatures']]
            script = b'' if 'script' not in i else i['script']
            address = '' if 'address' not in i else i['address']
            input_arr.append((i['prev_txid'], i['output_n'], None, int(i['value']), signatures, script
                              address))
        output_arr = []
        for o in t['outputs']:
            output_arr.append((o['address'], int(o['value'])))
        rt = self.transaction_create(output_arr, input_arr, fee=t['fee'], network=t['network'],
                                     random_output_order=False)
        rt.block_height = t['block_height']
        rt.confirmations = t['confirmations']
        rt.witness_type = t['witness_type']
        rt.date = t['date']
        rt.txid = t['txid']
        rt.txhash = t['txhash']
        rt.locktime = t['locktime']
        rt.version = t['version'].to_bytes(4, 'big')
        rt.version_int = t['version']
        rt.block_hash = t['block_hash']
        rt.rawtx = t['raw']
        rt.coinbase = t['coinbase']
        rt.flag = t['flag']
        rt.size = t['size']
        if not t['size']:
```

```
                rt.size = len(rt.raw())
            rt.vsize = t['vsize']
            if not rt.vsize:
                rt.vsize = rt.size
            rt.fee_per_kb = int((rt.fee / float(rt.vsize)) * 1000)
        else:
            raise WalletError("Import transaction must be of type Transaction or dict")
        rt.verify()
        return rt

    def transaction_import_raw(self, rawtx, network=None):
        """
        Import a raw transaction. Link inputs to wallet keys if possible and return WalletTransaction obje

        :param rawtx: Raw transaction
        :type rawtx: str, bytes
        :param network: Network name. Leave empty for default network
        :type network: str

        :return WalletTransaction:
        """

        if network is None:
            network = self.network.name
        if isinstance(rawtx, str):
            rawtx = bytes.fromhex(rawtx)
        t_import = Transaction.parse_bytes(rawtx, network=network)
        rt = self.transaction_create(t_import.outputs, t_import.inputs, network=network, locktime=t_import
                                     random_output_order=False)
        rt.version_int = t_import.version_int
        rt.version = t_import.version
        rt.verify()
        rt.size = len(rawtx)
        rt.calc_weight_units()
        rt.fee_per_kb = int((rt.fee / float(rt.vsize)) * 1000)
        return rt

    def send(self, output_arr, input_arr=None, input_key_id=None, account_id=None, network=None, fee=None,
             min_confirms=1, priv_keys=None, max_utxos=None, locktime=0, offline=True, number_of_change_ou
        """
        Create a new transaction with specified outputs and push it to the network.
        Inputs can be specified but if not provided they will be selected from wallets utxo's
        Output array is a list of 1 or more addresses and amounts.

         Uses the :func:`transaction_create` method to create a new transaction, and uses a random serv
send the transaction.

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> t = w.send([('1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb', 200000)], offline=True)
        >>> t
        <WalletTransaction(input_count=1, output_count=2, status=new, network=bitcoin)>
        >>> t.outputs # doctest:+ELLIPSIS
        [<Output(value=..., address=..., type=p2pkh)>, <Output(value=..., address=..., type=p2pkh)>]

          :param output_arr: List of output tuples with address and amount. Must contain at least one i
[('mxdLD8SAGS9fe2EeCXALDHcdTTbppMHp8N', 5000000)]. Address can be an address string, Address object, HD
WalletKey object
        :type output_arr: list
        :param input_arr: List of inputs tuples with reference to a UTXO, a wallet key and value. The form
output_n, key_id, value)]
        :type input_arr: list
```

```
        :param input_key_id: Limit UTXO's search for inputs to this key ID or list of key IDs. Only vali
array is specified
        :type input_key_id: int, list
        :param account_id: Account ID
        :type account_id: int
        :param network: Network name. Leave empty for default network
        :type network: str
        :param fee: Set fee manually, leave empty to calculate fees automatically. Set fees in the smal
denominator, for example satoshi's if you are using bitcoins. You can also supply a string: 'low', 'norm
to determine fees automatically.
        :type fee: int, str
        :param min_confirms: Minimal confirmation needed for an UTXO before it will be included in input
1. Option is ignored if input_arr is provided.
        :type min_confirms: int
        :param priv_keys: Specify extra private key if not available in this wallet
        :type priv_keys: HDKey, list
        :param max_utxos: Maximum number of UTXO's to use. Set to 1 for optimal privacy. Default is None:
        :type max_utxos: int
        :param locktime: Transaction level locktime. Locks the transaction until a specified block (valu
million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transac
locktime
        :type locktime: int
        :param offline: Just return the transaction object and do not send it when offline = True. Default
        :type offline: bool
        :param number_of_change_outputs: Number of change outputs to create when there is a change value.
Use 0 for random number of outputs: between 1 and 5 depending on send and change amount
        :type number_of_change_outputs: int

        :return WalletTransaction:
        """

        if input_arr and max_utxos and len(input_arr) > max_utxos:
            raise WalletError("Input array contains %d UTXO's but max_utxos=%d parameter specified" %
                              (len(input_arr), max_utxos))

        transaction = self.transaction_create(output_arr, input_arr, input_key_id, account_id, network, fe
                                              min_confirms, max_utxos, locktime, number_of_change_outputs)
        transaction.sign(priv_keys)
        # Calculate exact fees and update change output if necessary
        if fee is None and transaction.fee_per_kb and transaction.change:
            fee_exact = transaction.calculate_fee()
            # Recreate transaction if fee estimation more than 10% off
            if fee_exact != self.network.fee_min and fee_exact != self.network.fee_max and \
                    fee_exact and abs((float(transaction.fee) - float(fee_exact)) / float(fee_exact)) > 0.
                _logger.info("Transaction fee not correctly estimated (est.: %d, real: %d). "
                             "Recreate transaction with correct fee" % (transaction.fee, fee_exact))
                transaction = self.transaction_create(output_arr, input_arr, input_key_id, account_id, net
                                                      fee_exact, min_confirms, max_utxos, locktime,
                                                      number_of_change_outputs)
                transaction.sign(priv_keys)

        transaction.rawtx = transaction.raw()
        transaction.size = len(transaction.rawtx)
        transaction.calc_weight_units()
        transaction.fee_per_kb = int(float(transaction.fee) / float(transaction.vsize) * 1000)
        transaction.txid = transaction.signature_hash()[::-1].hex()
        transaction.send(offline)
        return transaction

    def send_to(self, to_address, amount, input_key_id=None, account_id=None, network=None, fee=None, min_
                priv_keys=None, locktime=0, offline=True, number_of_change_outputs=1):
```

```
        """
        Create transaction and send it with default Service objects :func:`services.sendrawtransaction` me

        Wrapper for wallet :func:`send` method.

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> t = w.send_to('1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb', 200000, offline=True)
        >>> t
        <WalletTransaction(input_count=1, output_count=2, status=new, network=bitcoin)>
        >>> t.outputs # doctest:+ELLIPSIS
        [<Output(value=..., address=..., type=p2pkh)>, <Output(value=..., address=..., type=p2pkh)>]

        :param to_address: Single output address as string Address object, HDKey object or WalletKey objec
        :type to_address: str, Address, HDKey, WalletKey
        :param amount: Output is the smallest denominator for this network (ie: Satoshi's for Bitcoin), as
or value string as accepted by Value class
        :type amount: int, str, Value
         :param input_key_id: Limit UTXO's search for inputs to this key ID or list of key IDs. Only vali
array is specified
        :type input_key_id: int, list
        :param account_id: Account ID, default is last used
        :type account_id: int
        :param network: Network name. Leave empty for default network
        :type network: str
         :param fee: Set fee manually, leave empty to calculate fees automatically. Set fees in the smal
denominator, for example satoshi's if you are using bitcoins. You can also supply a string: 'low', 'norm
to determine fees automatically.
        :type fee: int, str
         :param min_confirms: Minimal confirmation needed for an UTXO before it will be included in input
1. Option is ignored if input_arr is provided.
        :type min_confirms: int
        :param priv_keys: Specify extra private key if not available in this wallet
        :type priv_keys: HDKey, list
         :param locktime: Transaction level locktime. Locks the transaction until a specified block (valu
million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transac
locktime
        :type locktime: int
        :param offline: Just return the transaction object and do not send it when offline = True. Default
        :type offline: bool
        :param number_of_change_outputs: Number of change outputs to create when there is a change value.
Use 0 for random number of outputs: between 1 and 5 depending on send and change amount
        :type number_of_change_outputs: int

        :return WalletTransaction:
        """

        outputs = [(to_address, amount)]
        return self.send(outputs, input_key_id=input_key_id, account_id=account_id, network=network, fee=f
                         min_confirms=min_confirms, priv_keys=priv_keys, locktime=locktime, offline=offlir
                         number_of_change_outputs=number_of_change_outputs)

    def sweep(self, to_address, account_id=None, input_key_id=None, network=None, max_utxos=999, min_confi
              fee_per_kb=None, fee=None, locktime=0, offline=True):
        """
        Sweep all unspent transaction outputs (UTXO's) and send them to one or more output addresses.

        Wrapper for the :func:`send` method.

        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> t = w.sweep('1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb')
        >>> t
```

```
        <WalletTransaction(input_count=1, output_count=1, status=new, network=bitcoin)>
        >>> t.outputs # doctest:+ELLIPSIS
        [<Output(value=..., address=1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb, type=p2pkh)>]

        Output to multiple addresses

        >>> to_list = [('1J9GDZMKEr3ZTj8q6pwtMy4Arvt92FDBTb', 100000), (w.get_key(), 0)]
        >>> w.sweep(to_list)
        <WalletTransaction(input_count=1, output_count=2, status=new, network=bitcoin)>

        :param to_address: Single output address or list of outputs in format [(<adddress>, <amount>)]. I
a list of outputs, use amount value = 0 to indicate a change output
        :type to_address: str, list
        :param account_id: Wallet's account ID
        :type account_id: int
        :param input_key_id: Limit sweep to UTXO's with this key ID or list of key IDs
        :type input_key_id: int, list
        :param network: Network name. Leave empty for default network
        :type network: str
        :param max_utxos: Limit maximum number of outputs to use. Default is 999
        :type max_utxos: int
        :param min_confirms: Minimal confirmations needed to include utxo
        :type min_confirms: int
          :param fee_per_kb: Fee per kilobyte transaction size, leave empty to get estimated fee costs
provider. This option is ignored when the 'fee' option is specified
        :type fee_per_kb: int
        :param fee: Total transaction fee in the smallest denominator (i.e. satoshis). Leave empty to get
from service providers. You can also supply a string: 'low', 'normal' or 'high' to determine fees automati
        :type fee: int, str
        :param locktime: Transaction level locktime. Locks the transaction until a specified block (valu
million) or until a certain time (Timestamp in seconds after 1-jan-1970). Default value is 0 for transac
locktime
        :type locktime: int
        :param offline: Just return the transaction object and do not send it when offline = True. Default
        :type offline: bool

        :return WalletTransaction:
        """

        network, account_id, acckey = self._get_account_defaults(network, account_id)

        utxos = self.utxos(account_id=account_id, network=network, min_confirms=min_confirms, key_id=input
        utxos = utxos[0:max_utxos]
        input_arr = []
        total_amount = 0
        if not utxos:
            raise WalletError("Cannot sweep wallet, no UTXO's found")
        for utxo in utxos:
            # Skip dust transactions to avoid forced address reuse
            if utxo['value'] <= self.network.dust_amount:
                continue
            input_arr.append((utxo['txid'], utxo['output_n'], utxo['key_id'], utxo['value']))
            total_amount += utxo['value']
        srv = Service(network=network, providers=self.providers, cache_uri=self.db_cache_uri)

        if isinstance(fee, str):
            n_outputs = 1 if not isinstance(to_address, list) else len(to_address)
            fee_per_kb = srv.estimatefee(priority=fee)
            tr_size = 125 + (len(input_arr) * (77 + self.multisig_n_required * 72)) + n_outputs * 30
            fee = 100 + int((tr_size / 1000.0) * fee_per_kb)
```

```
        if not fee:
            if fee_per_kb is None:
                fee_per_kb = srv.estimatefee()
            tr_size = 125 + (len(input_arr) * 125)
            fee = int((tr_size / 1000.0) * fee_per_kb)
        if total_amount - fee <= self.network.dust_amount:
            raise WalletError("Amount to send is smaller then dust amount: %s" % (total_amount - fee))

        if isinstance(to_address, str):
            to_list = [(to_address, total_amount - fee)]
        else:
            to_list = []
            for o in to_address:
                if o[1] == 0:
                    o_amount = total_amount - sum([x[1] for x in to_list]) - fee
                    if o_amount > 0:
                        to_list.append((o[0], o_amount))
                else:
                    to_list.append(o)

        if sum(x[1] for x in to_list) + fee != total_amount:
            raise WalletError("Total amount of outputs does not match total input amount. If you specify a
                              "outputs, use amount value = 0 to indicate a change/rest output")

        return self.send(to_list, input_arr, network=network, fee=fee, min_confirms=min_confirms, locktime
                         offline=offline)

    def wif(self, is_private=False, account_id=0):
        """
        Return Wallet Import Format string for master private or public key which can be used to import ke
        recreate wallet in other software.

        A list of keys will be exported for a multisig wallet.

        :param is_private: Export public or private key, default is False
        :type is_private: bool
        :param account_id: Account ID of key to export
        :type account_id: bool

        :return list, str:
        """
        if not self.multisig or not self.cosigner:
            if is_private and self.main_key:
                return self.main_key.wif
            else:
                return self.public_master(account_id=account_id).key().\
                    wif(is_private=is_private, witness_type=self.witness_type, multisig=self.multisig)
        else:
            wiflist = []
            for cs in self.cosigner:
                wiflist.append(cs.wif(is_private=is_private))
            return wiflist

    def public_master(self, account_id=None, name=None, as_private=False, network=None):
        """
          Return public master key(s) for this wallet. Use to import in other wallets to sign transacti
keys.

        For a multisig wallet all public master keys are return as list.

        Returns private key information if available and as_private is True is specified
```

```
        >>> w = Wallet('bitcoinlib_legacy_wallet_test')
        >>> w.public_master().wif

'xpub6D2qEr8Z8WYKKns2xZYyyvvRviPh1NKt1kfHwwfiTxJwj7peReEJt3iXoWWsr8tXWTsejDjMfAezM53KVFVkSZzA5i2pNy3otprtY

        :param account_id: Account ID of key to export
        :type account_id: int
        :param name: Optional name for account key
        :type name: str
        :param as_private: Export public or private key, default is False
        :type as_private: bool
        :param network: Network name. Leave empty for default network
        :type network: str

        :return list of WalletKey, WalletKey:
        """
        if self.main_key and self.main_key.key_type == 'single':
            key = self.main_key
            return key if as_private else key.public()
        elif not self.cosigner:
            depth = -self.key_depth + self.depth_public_master
            key = self.key_for_path([], depth, name=name, account_id=account_id, network=network,
                                    cosigner_id=self.cosigner_id)
            return key if as_private else key.public()
        else:
            pm_list = []
            for cs in self.cosigner:
                pm_list.append(cs.public_master(account_id, name, as_private, network))
            return pm_list

    def transaction_load(self, txid=None, filename=None):
        """
        Load transaction object from file which has been stored with the :func:`Transaction.save` method.

        Specify transaction ID or filename.

        :param txid: Transaction ID. Transaction object will be read from .bitcoinlib datadir
        :type txid: str
        :param filename: Name of transaction object file
        :type filename: str

        :return Transaction:
        """
        if not filename and not txid:
            raise WalletError("Please supply filename or txid")
        elif not filename and txid:
            p = Path(BCL_DATA_DIR, '%s.tx' % txid)
        else:
            p = Path(filename)
            if not p.parent or str(p.parent) == '.':
                p = Path(BCL_DATA_DIR, filename)
        f = p.open('rb')
        t = pickle.load(f)
        f.close()
        return self.transaction_import(t)

    def info(self, detail=3):
        """
        Prints wallet information to standard output
```

```
          :param detail: Level of detail to show. Specify a number between 0 and 5, with 0 low detail
detail
        :type detail: int
        """

        print("=== WALLET ===")
        print(" ID                            %s" % self.wallet_id)
        print(" Name                          %s" % self.name)
        print(" Owner                         %s" % self.owner)
        print(" Scheme                        %s" % self.scheme)
        print(" Multisig                      %s" % self.multisig)
        if self.multisig:
            print(" Multisig Wallet IDs           %s" % str([w.wallet_id for w in self.cosigner]).strip('
            print(" Cosigner ID                   %s" % self.cosigner_id)
        print(" Witness type                  %s" % self.witness_type)
        print(" Main network                  %s" % self.network.name)
        print(" Latest update                 %s" % self.last_updated)

        if self.multisig:
            print("\n= Multisig Public Master Keys =")
            for cs in self.cosigner:
                print("%5s %3s %-70s %-6s %-8s %s" %
                      (cs.cosigner_id, cs.main_key.key_id, cs.wif(is_private=False), cs.scheme,
                       "main" if cs.main_key.is_private else "cosigner",
                       '*' if cs.cosigner_id == self.cosigner_id else ''))

            print("For main keys a private master key is available in this wallet to sign transactions. "
                  "* cosigner key for this wallet")

        if detail and self.main_key:
            print("\n= Wallet Master Key =")
            print(" ID                            %s" % self.main_key_id)
            print(" Private                       %s" % self.main_key.is_private)
            print(" Depth                         %s" % self.main_key.depth)

        balances = self._balance_update()
        if detail > 1:
            for nw in self.networks():
                print("\n- NETWORK: %s -" % nw.name)
                print("- - Keys")
                if detail < 4:
                    ds = [self.key_depth]
                elif detail < 5:
                    if self.purpose == 45:
                        ds = [0, self.key_depth]
                    else:
                        ds = [0, self.depth_public_master, self.key_depth]
                else:
                    ds = range(8)
                for d in ds:
                    is_active = True
                    if detail > 3:
                        is_active = False
                    for key in self.keys(depth=d, network=nw.name, is_active=is_active):
                        print("%5s %-28s %-45s %-25s %25s" %
                              (key.id, key.path, key.address, key.name,
                               Value.from_satoshi(key.balance, network=nw).str_unit(currency_repr='symbol'

                if detail > 2:
                    include_new = False
                    if detail > 3:
```

```
                    include_new = True
                accounts = self.accounts(network=nw.name)
                if not accounts:
                    accounts = [0]
                for account_id in accounts:
                    txs = self.transactions(include_new=include_new, account_id=account_id, network=nw
                                            as_dict=True)
                    print("\n- - Transactions Account %d (%d)" % (account_id, len(txs)))
                    for tx in txs:
                        spent = " "
                        address = tx['address']
                        if not tx['address']:
                            address = 'nulldata'
                        elif 'spent' in tx and tx['spent'] is False:
                            spent = "U"
                        status = ""
                        if tx['status'] not in ['confirmed', 'unconfirmed']:
                            status = tx['status']
                        print("%64s %43s %8d %21s %s %s" % (tx['txid'], address, tx['confirmations'],
                                                            Value.from_satoshi(tx['value'], network=nw
                                                                currency_repr='symbol'),
                                                            spent, status))

        print("\n= Balance Totals (includes unconfirmed) =")
        for na_balance in balances:
            print("%-20s %-20s %20s" % (na_balance['network'], "(Account %s)" % na_balance['account_id'],
                                        Value.from_satoshi(na_balance['balance'], network=na_balance['netw
                                        str_unit(currency_repr='symbol')))
        print("\n")

    def as_dict(self, include_private=False):
        """
        Return wallet information in dictionary format

        :param include_private: Include private key information in dictionary
        :type include_private: bool

        :return dict:
        """

        keys = []
        transactions = []
        for netw in self.networks():
            for key in self.keys(network=netw.name, include_private=include_private, as_dict=True):
                keys.append(key)

            if self.multisig:
                for t in self.transactions(include_new=True, account_id=0, network=netw.name):
                    transactions.append(t.as_dict())
            else:
                accounts = self.accounts(network=netw.name)
                if not accounts:
                    accounts = [0]
                for account_id in accounts:
                    for t in self.transactions(include_new=True, account_id=account_id, network=netw.name)
                        transactions.append(t.as_dict())

        return {
            'wallet_id': self.wallet_id,
            'name': self.name,
            'owner': self._owner,
```

```
            'scheme': self.scheme,
            'witness_type': self.witness_type,
            'main_network': self.network.name,
            'main_balance': self.balance(),
            'main_balance_str': self.balance(as_string=True),
            'balances': self._balances,
            'default_account_id': self.default_account_id,
            'multisig_n_required': self.multisig_n_required,
            'cosigner_wallet_ids': [w.wallet_id for w in self.cosigner],
            'cosigner_public_masters': [w.public_master().key().wif() for w in self.cosigner],
            'sort_keys': self.sort_keys,
            'main_key_id': self.main_key_id,
            'encoding': self.encoding,
            'keys': keys,
            'transactions': transactions,
        }

    def as_json(self, include_private=False):
        """
        Get current key as json formatted string

        :param include_private: Include private key information in JSON
        :type include_private: bool

        :return str:
        """
        adict = self.as_dict(include_private=include_private)
        return json.dumps(adict, indent=4, default=str)
```