python-bitcoinlib Documentation

Release 0.10.2dev

The python-bitcoinlib developers

Jul 31, 2018

Contents

CHAPTER 1

API Reference

This section contains auto-generated API documentation.

Contents:

1.1 bitcoin

1.1.1 bitcoin

class bitcoin.**MainParams**
Bases: *bitcoin.core.CoreMainParams*

BASE58_PREFIXES = {u'PUBKEY_ADDR': 0, u'SCRIPT_ADDR': 5, u'SECRET_KEY': 128}

DEFAULT_PORT = 8333

DNS_SEEDS = ((u'bitcoin.sipa.be', u'seed.bitcoin.sipa.be'), (u'bluematt.me', u'dnsseed

MESSAGE_START = '\xf9\xbe\xb4\xd9'

RPC_PORT = 8332

class bitcoin.**RegTestParams**
Bases: *bitcoin.core.CoreRegTestParams*

BASE58_PREFIXES = {u'PUBKEY_ADDR': 111, u'SCRIPT_ADDR': 196, u'SECRET_KEY': 239}

DEFAULT_PORT = 18444

DNS_SEEDS = ()

MESSAGE_START = '\xfa\xbf\xb5\xda'

RPC_PORT = 18443

bitcoin.**SelectParams**(*name*)
Select the chain parameters to use

name is one of 'mainnet', 'testnet', or 'regtest'

Default chain is 'mainnet'

class bitcoin.**TestNetParams**
Bases: *bitcoin.core.CoreTestNetParams*

BASE58_PREFIXES = {u'PUBKEY_ADDR': 111, u'SCRIPT_ADDR': 196, u'SECRET_KEY': 239}

DEFAULT_PORT = 18333

DNS_SEEDS = ((u'testnetbitcoin.jonasschnelli.ch', u'testnet-seed.bitcoin.jonasschnelli

MESSAGE_START = '\x0b\x11\t\x07'

RPC_PORT = 18332

1.1.2 base58

Base58 encoding and decoding

exception bitcoin.base58.**Base58Error**
Bases: exceptions.Exception

exception bitcoin.base58.**InvalidBase58Error**
Bases: *bitcoin.base58.Base58Error*

Raised on generic invalid base58 data, such as bad characters.

Checksum failures raise Base58ChecksumError specifically.

bitcoin.base58.**encode**(*b*)
Encode bytes to a base58-encoded string

bitcoin.base58.**decode**(*s*)
Decode a base58-encoding string, returning bytes

exception bitcoin.base58.**Base58ChecksumError**
Bases: *bitcoin.base58.Base58Error*

Raised on Base58 checksum errors

class bitcoin.base58.**CBase58Data**(*s*)
Bases: str

Base58-encoded data

Includes a version and checksum.

Initialize from base58-encoded string

Note: subclasses put your initialization routines here, but ignore the argument - that's handled by __new__(), and .from_bytes() will call __init__() with None in place of the string.

classmethod from_bytes(*data*, *nVersion*)
Instantiate from data and nVersion

to_bytes()
Convert to bytes instance

Note that it's the data represented that is converted; the checkum and nVersion is not included.

1.1.3 bloom

Bloom filter support

bitcoin.bloom.**MurmurHash3**(*x86_32*)
Used for bloom filters. See http://code.google.com/p/smhasher/source/browse/trunk/MurmurHash3.cpp

class bitcoin.bloom.**CBloomFilter**(*nElements*, *nFPRate*, *nTweak*, *nFlags*)
Bases: *bitcoin.core.serialize.Serializable*

Create a new bloom filter

The filter will have a given false-positive rate when filled with the given number of elements.

Note that if the given parameters will result in a filter outside the bounds of the protocol limits, the filter created will be as close to the given parameters as possible within the protocol limits. This will apply if nFPRate is very low or nElements is unreasonably high.

nTweak is a constant which is added to the seed value passed to the hash function It should generally always be a random value (and is largely only exposed for unit testing)

nFlags should be one of the UPDATE_* enums (but not _MASK)

IsRelevantAndUpdate(*tx_hash*)

IsWithinSizeConstraints()

MAX_BLOOM_FILTER_SIZE = 36000

MAX_HASH_FUNCS = 50

UPDATE_ALL = 1

UPDATE_MASK = 3

UPDATE_NONE = 0

UPDATE_P2PUBKEY_ONLY = 2

bloom_hash(*nHashNum*, *vDataToHash*)

contains(*elem*)
Test if the filter contains an element

elem may be a COutPoint or bytes

insert(*elem*)
Insert an element in the filter.

elem may be a COutPoint or bytes

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

1.1.4 messages

class bitcoin.messages.**MsgSerializable**(*protover=60002*)
Bases: *bitcoin.core.serialize.Serializable*

classmethod from_bytes(*b*, *protover=60002*)

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

classmethod stream_deserialize(*f*, *protover=60002*)

stream_serialize(*f*)

to_bytes()

class bitcoin.messages.**msg_version**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'version'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_verack**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'verack'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_addr**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'addr'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_alert**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'alert'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_inv**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'inv'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_getdata**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'getdata'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_getblocks**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'getblocks'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_getheaders**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'getheaders'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_headers**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'headers'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_tx**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'tx'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_block**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'block'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_getaddr**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'getaddr'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_ping**(*protover=60002*, *nonce=0*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'ping'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_pong**(*protover=60002*, *nonce=0*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'pong'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

class bitcoin.messages.**msg_mempool**(*protover=60002*)
Bases: *bitcoin.messages.MsgSerializable*

command = 'mempool'

classmethod msg_deser(*f*, *protover=60002*)

msg_ser(*f*)

1.1.5 net

class bitcoin.net.**CAddress**(*protover=60002*)
Bases: *bitcoin.core.serialize.Serializable*

classmethod stream_deserialize(*f*, *without_time=False*)

stream_serialize(*f*, *without_time=False*)

class bitcoin.net.**CInv**
Bases: *bitcoin.core.serialize.Serializable*

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

typemap = {0: u'Error', 1: u'TX', 2: u'Block', 3: u'FilteredBlock', 4: u'CompactB

class bitcoin.net.**CBlockLocator**(*protover=60002*)
Bases: *bitcoin.core.serialize.Serializable*

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

class bitcoin.net.**CUnsignedAlert**
Bases: *bitcoin.core.serialize.Serializable*

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

class bitcoin.net.**CAlert**
Bases: *bitcoin.core.serialize.Serializable*

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

1.1.6 rpc

Bitcoin Core RPC support

By default this uses the standard library `json` module. By monkey patching, a different implementation can be used instead, at your own risk:

```
>>> import simplejson
>>> import bitcoin.rpc
>>> bitcoin.rpc.json = simplejson
```

(`simplejson` is the externally maintained version of the same module and thus better optimized but perhaps less stable.)

exception bitcoin.rpc.**JSONRPCError**
Bases: `exceptions.Exception`

JSON-RPC protocol error base class

Subclasses of this class also exist for specific types of errors; the set of all subclasses is by no means complete.

SUBCLS_BY_CODE = {-28: <class 'bitcoin.rpc.InWarmupError'>, -27: <class 'bitcoin.rpc

exception bitcoin.rpc.**ForbiddenBySafeModeError**
Bases: *bitcoin.rpc.JSONRPCError*

RPC_ERROR_CODE = -2

exception bitcoin.rpc.**InvalidAddressOrKeyError**
Bases: *bitcoin.rpc.JSONRPCError*

RPC_ERROR_CODE = -5

exception bitcoin.rpc.**InvalidParameterError**
Bases: *bitcoin.rpc.JSONRPCError*

RPC_ERROR_CODE = -8

exception bitcoin.rpc.**VerifyError**
Bases: *bitcoin.rpc.JSONRPCError*

RPC_ERROR_CODE = -25

exception bitcoin.rpc.**VerifyRejectedError**
Bases: *bitcoin.rpc.JSONRPCError*

RPC_ERROR_CODE = -26

exception bitcoin.rpc.**VerifyAlreadyInChainError**
Bases: *bitcoin.rpc.JSONRPCError*

RPC_ERROR_CODE = -27

exception bitcoin.rpc.**InWarmupError**
Bases: *bitcoin.rpc.JSONRPCError*

RPC_ERROR_CODE = -28

class bitcoin.rpc.**RawProxy**(*service_url=None*, *service_port=None*, *btc_conf_file=None*, *timeout=30*, ***kwargs*)
Bases: bitcoin.rpc.BaseProxy

Low-level proxy to a bitcoin JSON-RPC service

Unlike `Proxy`, no conversion is done besides parsing JSON. As far as Python is concerned, you can call any method; `JSONRPCError` will be raised if the server does not recognize it.

class bitcoin.rpc.**Proxy**(*service_url=None*, *service_port=None*, *btc_conf_file=None*, *timeout=30*, ***kwargs*)
Bases: bitcoin.rpc.BaseProxy

Proxy to a bitcoin RPC service

Unlike `RawProxy`, data is passed as `bitcoin.core` objects or packed bytes, rather than JSON or hex strings. Not all methods are implemented yet; you can use `call` to access missing ones in a forward-compatible way. Assumes Bitcoin Core version >= v0.16.0; older versions mostly work, but there are a few incompatibilities.

Create a proxy object

If `service_url` is not specified, the username and password are read out of the file `btc_conf_file`. If `btc_conf_file` is not specified, `~/.bitcoin/bitcoin.conf` or equivalent is used by default. The default port is set according to the chain parameters in use: mainnet, testnet, or regtest.

Usually no arguments to `Proxy()` are needed; the local bitcoind will be used.

`timeout` - timeout in seconds before the HTTP interface times out

addnode(*node*)

addnodeonetry(*node*)

call(*service_name*, **args*)
Call an RPC method by name and raw (JSON encodable) arguments

dumpprivkey(*addr*)
Return the private key matching an address

fundrawtransaction(*tx*, *include_watching=False*)
Add inputs to a transaction until it has enough in value to meet its out value.

include_watching - Also select inputs which are watch only

Returns dict:

{'tx': Resulting tx, 'fee': Fee the resulting transaction pays, 'changepos': Position of added change output, or -1,

}

generate(*numblocks*)
Mine blocks immediately (before the RPC call returns)

numblocks - How many blocks are generated immediately.

Returns iterable of block hashes generated.

getaccountaddress(*account=None*)
Return the current Bitcoin address for receiving payments to this account.

getbalance(*account=u'*'*, *minconf=1*, *include_watchonly=False*)
Get the balance

account - The selected account. Defaults to "*" for entire wallet. It may be the default account using "".

minconf - Only include transactions confirmed at least this many times. (default=1)

include_watchonly - Also include balance in watch-only addresses (see 'importaddress') (default=False)

getbestblockhash()
Return hash of best (tip) block in longest block chain.

getblock(*block_hash*)
Get block <block_hash>

Raises IndexError if block_hash is not valid.

getblockcount()
Return the number of blocks in the longest block chain

getblockhash(*height*)
Return hash of block in best-block-chain at height.

Raises IndexError if height is not valid.

getblockheader(*block_hash*, *verbose=False*)
Get block header <block_hash>

verbose - If true a dict is returned with the values returned by getblockheader that are not in the block header itself (height, nextblockhash, etc.)

Raises IndexError if block_hash is not valid.

getinfo()
Return a JSON object containing various state info

getmininginfo()
Return a JSON object containing mining-related information

getnewaddress(*account=None*)
Return a new Bitcoin address for receiving payments.

If account is not None, it is added to the address book so payments received with the address will be credited to account.

getrawchangeaddress()
Returns a new Bitcoin address, for receiving change.

This is for use with raw transactions, NOT normal use.

getrawmempool(*verbose=False*)
Return the mempool

getrawtransaction(*txid*, *verbose=False*)
Return transaction with hash txid

Raises IndexError if transaction not found.

verbose - If true a dict is returned instead with additional information on the transaction.

Note that if all txouts are spent and the transaction index is not enabled the transaction may not be available.

getreceivedbyaddress(*addr*, *minconf=1*)
Return total amount received by given a (wallet) address

Get the amount received by <address> in transactions with at least [minconf] confirmations.

Works only for addresses in the local wallet; other addresses will always show zero.

addr - The address. (CBitcoinAddress instance)

minconf - Only include transactions confirmed at least this many times. (default=1)

gettransaction(*txid*)
Get detailed information about in-wallet transaction txid

Raises IndexError if transaction not found in the wallet.

FIXME: Returned data types are not yet converted.

gettxout(*outpoint*, *includemempool=True*)
Return details about an unspent transaction output.

Raises IndexError if outpoint is not found or was spent.

includemempool - Include mempool txouts

importaddress(*addr*, *label=u''*, *rescan=True*)
Adds an address or pubkey to wallet without the associated privkey.

listunspent(*minconf=0*, *maxconf=9999999*, *addrs=None*)
Return unspent transaction outputs in wallet

Outputs will have between minconf and maxconf (inclusive) confirmations, optionally filtered to only include txouts paid to addresses in addrs.

lockunspent(*unlock*, *outpoints*)
Lock or unlock outpoints

removenode(*node*)

sendmany(*fromaccount*, *payments*, *minconf=1*, *comment=u''*, *subtractfeefromamount=[]*)
Send amount to given addresses.

payments - dict with {address: amount}

sendrawtransaction(*tx*, *allowhighfees=False*)
Submit transaction to local node and network.

allowhighfees - Allow even if fees are unreasonably high.

sendtoaddress(*addr*, *amount*, *comment=u''*, *commentto=u''*, *subtractfeefromamount=False*)
Send amount to a given address

signrawtransaction(*tx*, **args*)
Sign inputs for transaction

FIXME: implement options

submitblock(*block*, *params=None*)
Submit a new block to the network.

params is optional and is currently ignored by bitcoind. See https://en.bitcoin.it/wiki/BIP_0022 for full specification.

unlockwallet(*password*, *timeout=60*)
Stores the wallet decryption key in memory for 'timeout' seconds.

password - The wallet passphrase.

timeout - The time to keep the decryption key in seconds. (default=60)

validateaddress(*address*)
Return information about an address

1.1.7 signature

class bitcoin.signature.**DERSignature**(*r*, *s*, *length*)
Bases: *bitcoin.core.serialize.ImmutableSerializable*

length

r

s

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

1.1.8 signmessage

class bitcoin.signmessage.**BitcoinMessage**(*message=u''*, *magic=u'Bitcoin Signed Message:n'*)
Bases: *bitcoin.core.serialize.ImmutableSerializable*

magic

message

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

bitcoin.signmessage.**SignMessage**(*key*, *message*)

bitcoin.signmessage.**VerifyMessage**(*address*, *message*, *sig*)

1.1.9 wallet

Wallet-related functionality

Includes things like representing addresses and converting them to/from scriptPubKeys; currently there is no actual wallet support implemented.

exception bitcoin.wallet.**CBitcoinAddressError**
Bases: *bitcoin.base58.Base58Error*

Raised when an invalid Bitcoin address is encountered

class bitcoin.wallet.**CBitcoinAddress**(*s*)
Bases: *bitcoin.base58.CBase58Data*

A Bitcoin address

Initialize from base58-encoded string

Note: subclasses put your initialization routines here, but ignore the argument - that's handled by __new__(), and .from_bytes() will call __init__() with None in place of the string.

classmethod from_bytes(*data*, *nVersion*)

classmethod from_scriptPubKey(*scriptPubKey*)
Convert a scriptPubKey to a CBitcoinAddress

Returns a CBitcoinAddress subclass, either P2SHBitcoinAddress or P2PKHBitcoinAddress. If the scriptPubKey is not recognized CBitcoinAddressError will be raised.

to_scriptPubKey()
Convert an address to a scriptPubKey

class bitcoin.wallet.**P2SHBitcoinAddress**(*s*)
Bases: *bitcoin.wallet.CBitcoinAddress*

Initialize from base58-encoded string

Note: subclasses put your initialization routines here, but ignore the argument - that's handled by __new__(), and .from_bytes() will call __init__() with None in place of the string.

classmethod from_bytes(*data*, *nVersion=None*)

classmethod from_redeemScript(*redeemScript*)
Convert a redeemScript to a P2SH address

Convenience function: equivalent to P2SHBitcoinAddress.from_scriptPubKey(redeemScript.to_p2sh_scriptPubKey())

classmethod from_scriptPubKey(*scriptPubKey*)
Convert a scriptPubKey to a P2SH address

Raises CBitcoinAddressError if the scriptPubKey isn't of the correct form.

to_scriptPubKey()
Convert an address to a scriptPubKey

class bitcoin.wallet.**P2PKHBitcoinAddress**(*s*)
Bases: *bitcoin.wallet.CBitcoinAddress*

Initialize from base58-encoded string

Note: subclasses put your initialization routines here, but ignore the argument - that's handled by __new__(), and .from_bytes() will call __init__() with None in place of the string.

classmethod from_bytes(*data*, *nVersion=None*)

classmethod from_pubkey(*pubkey*, *accept_invalid=False*)
Create a P2PKH bitcoin address from a pubkey

Raises CBitcoinAddressError if pubkey is invalid, unless accept_invalid is True.

The pubkey must be a bytes instance; CECKey instances are not accepted.

classmethod from_scriptPubKey(*scriptPubKey*, *accept_non_canonical_pushdata=True*, *accept_bare_checksig=True*)
Convert a scriptPubKey to a P2PKH address

Raises CBitcoinAddressError if the scriptPubKey isn't of the correct form.

accept_non_canonical_pushdata - Allow non-canonical pushes (default True)

accept_bare_checksig - Treat bare-checksig as P2PKH scriptPubKeys (default True)

to_scriptPubKey(*nested=False*)
Convert an address to a scriptPubKey

class bitcoin.wallet.**CKey**(*secret*, *compressed=True*)
Bases: object

An encapsulated private key

Attributes:

pub - The corresponding CPubKey for this private key

is_compressed - True if compressed

is_compressed

sign(*hash*)

sign_compact(*hash*)

exception bitcoin.wallet.**CBitcoinSecretError**
Bases: *bitcoin.base58.Base58Error*

class bitcoin.wallet.**CBitcoinSecret**(*s*)
Bases: *bitcoin.base58.CBase58Data*, *bitcoin.wallet.CKey*

A base58-encoded secret key

classmethod from_secret_bytes(*secret*, *compressed=True*)
Create a secret key from a 32-byte secret

1.2 bitcoin.core

Everything consensus critical is found in the core subpackage.

1.2.1 core

bitcoin.core.**Hash**(*msg*)
SHA256^2)(msg) -> bytes

bitcoin.core.**Hash160**(*msg*)
RIPEME160(SHA256(msg)) -> bytes

bitcoin.core.**MoneyRange**(*nValue*, *params=None*)

bitcoin.core.**x**(*h*)
Convert a hex string to bytes

bitcoin.core.**b2x**(*b*)
Convert bytes to a hex string

bitcoin.core.**lx**(*h*)
Convert a little-endian hex string to bytes

Lets you write uint256's and uint160's the way the Satoshi codebase shows them.

bitcoin.core.**b2lx**(*b*)
Convert bytes to a little-endian hex string

Lets you show uint256's and uint160's the way the Satoshi codebase shows them.

bitcoin.core.**str_money_value**(*value*)
Convert an integer money value to a fixed point string

exception bitcoin.core.**ValidationError**
Bases: exceptions.Exception

Base class for all blockchain validation errors

Everything that is related to validating the blockchain, blocks, transactions, scripts, etc. is derived from this class.

class bitcoin.core.**COutPoint**(*hash='x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x0*
n=4294967295)
Bases: *bitcoin.core.serialize.ImmutableSerializable*

The combination of a transaction hash and an index n into its vout

classmethod from_outpoint(*outpoint*)
Create an immutable copy of an existing OutPoint

If outpoint is already immutable (outpoint.__class__ is COutPoint) it is returned directly.

hash

is_null()

n

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

class bitcoin.core.**CMutableOutPoint**(*hash='x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x0*
n=4294967295)
Bases: *bitcoin.core.COutPoint*

A mutable COutPoint

GetHash()
Return the hash of the serialized object

classmethod from_outpoint(*outpoint*)
Create a mutable copy of an existing COutPoint

class bitcoin.core.**CTxIn**(*prevout=COutPoint()*, *scriptSig=CScript([])*, *nSequence=4294967295*)
Bases: *bitcoin.core.serialize.ImmutableSerializable*

An input of a transaction

Contains the location of the previous transaction's output that it claims, and a signature that matches the output's public key.

classmethod from_txin(*txin*)
Create an immutable copy of an existing TxIn

If txin is already immutable (txin.__class__ is CTxIn) it is returned directly.

is_final()

nSequence

prevout

scriptSig

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

class bitcoin.core.**CMutableTxIn**(*prevout=None*, *scriptSig=CScript([])*, *nSequence=4294967295*)
Bases: *bitcoin.core.CTxIn*

A mutable CTxIn

GetHash()
Return the hash of the serialized object

classmethod from_txin(*txin*)
Create a fully mutable copy of an existing TxIn

class bitcoin.core.**CTxOut**(*nValue=-1*, *scriptPubKey=CScript([])*)
Bases: *bitcoin.core.serialize.ImmutableSerializable*

An output of a transaction

Contains the public key that the next input must be able to sign with to claim it.

classmethod from_txout(*txout*)
Create an immutable copy of an existing TxOut

If txout is already immutable (txout.__class__ is CTxOut) then it will be returned directly.

is_valid()

nValue

scriptPubKey

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

class bitcoin.core.**CMutableTxOut**(*nValue=-1*, *scriptPubKey=CScript([])*)
Bases: *bitcoin.core.CTxOut*

A mutable CTxOut

GetHash()
Return the hash of the serialized object

classmethod from_txout(*txout*)
Create a fullly mutable copy of an existing TxOut

class bitcoin.core.**CTransaction**(*vin=()*, *vout=()*, *nLockTime=0*, *nVersion=1*, *witness=CTxWitness()*)
Bases: *bitcoin.core.serialize.ImmutableSerializable*

A transaction

Create a new transaction

vin and vout are iterables of transaction inputs and outputs respectively. If their contents are not already immutable, immutable copies will be made.

GetTxid()
Get the transaction ID. This differs from the transactions hash as given by GetHash. GetTxid excludes witness data, while GetHash includes it.

classmethod from_tx(*tx*)
Create an immutable copy of a pre-existing transaction

If tx is already immutable (tx.__class__ is CTransaction) then it will be returned directly.

has_witness()
True if witness

is_coinbase()

nLockTime

nVersion

classmethod stream_deserialize(*f*)
Deserialize transaction

This implementation corresponds to Bitcoin's SerializeTransaction() and consensus behavior. Note that Bitcoin's DecodeHexTx() also has the option to attempt deserializing as a non-witness transaction first, falling back to the consensus behavior if it fails. The difference lies in transactions which have zero inputs: they are invalid but may be (de)serialized anyway for the purpose of signing them and adding inputs. If the behavior of DecodeHexTx() is needed it could be added, but not here.

stream_serialize(*f*, *include_witness=True*)

vin

vout

wit

class bitcoin.core.**CMutableTransaction**(*vin=None*, *vout=None*, *nLockTime=0*, *nVersion=1*, *witness=None*)
Bases: *bitcoin.core.CTransaction*

A mutable transaction

GetHash()
Return the hash of the serialized object

classmethod from_tx(*tx*)
Create a fully mutable copy of a pre-existing transaction

class bitcoin.core.**CTxWitness**(*vtxinwit=()*)
Bases: *bitcoin.core.serialize.ImmutableSerializable*

Witness data for all inputs to a transaction

classmethod from_txwitness(*txwitness*)
Create an immutable copy of an existing TxWitness

If txwitness is already immutable (txwitness.__class__ is CTxWitness) it is returned directly.

is_null()

stream_deserialize(*f*)

stream_serialize(*f*)

vtxinwit

class bitcoin.core.**CTxInWitness**(*scriptWitness=CScriptWitness()*)
Bases: *bitcoin.core.serialize.ImmutableSerializable*

Witness data for a single transaction input

classmethod from_txinwitness(*txinwitness*)
Create an immutable copy of an existing TxInWitness

If txin is already immutable (txin.__class__ is CTxIn) it is returned directly.

is_null()

scriptWitness

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

class bitcoin.core.**CBlockHeader**(*nVersion=2*, *hashPrevBlock='x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x0*
hashMerkleRoot='x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00
nTime=0, *nBits=0*, *nNonce=0*)
Bases: *bitcoin.core.serialize.ImmutableSerializable*

A block header

static calc_difficulty(*nBits*)
Calculate difficulty from nBits target

difficulty

hashMerkleRoot

hashPrevBlock

nBits

nNonce

nTime

nVersion

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

class bitcoin.core.**CBlock**(*nVersion=2*, *hashPrevBlock='x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x0*
hashMerkleRoot='x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x0
nTime=0, *nBits=0*, *nNonce=0*, *vtx=()*)
Bases: *bitcoin.core.CBlockHeader*

A block including all transactions in it

Create a new block

GetHash()
Return the block hash

Note that this is the hash of the header, not the entire serialized block.

GetWeight()
Return the block weight: (stripped_size * 3) + total_size

static build_merkle_tree_from_txids(*txids*)
Build a full CBlock merkle tree from txids

txids - iterable of txids

Returns a new merkle tree in deepest first order. The last element is the merkle root.

WARNING! If you're reading this because you're learning about crypto and/or designing a new system that will use merkle trees, keep in mind that the following merkle tree algorithm has a serious flaw related to duplicate txids, resulting in a vulnerability. (CVE-2012-2459) Bitcoin has since worked around the flaw, but for new applications you should use something different; don't just copy-and-paste this code without understanding the problem first.

static build_merkle_tree_from_txs(*txs*)
Build a full merkle tree from transactions

static build_witness_merkle_tree_from_txs(*txs*)
Calculate the witness merkle tree from transactions

calc_merkle_root()
Calculate the merkle root

The calculated merkle root is not cached; every invocation re-calculates it from scratch.

calc_witness_merkle_root()
Calculate the witness merkle root

The calculated merkle root is not cached; every invocation re-calculates it from scratch.

get_header()
Return the block header

Returned header is a new object.

get_witness_commitment_index()
Find txout # of witness commitment in coinbase

Return None or an index

classmethod stream_deserialize(*f*)

stream_serialize(*f*, *include_witness=True*)

vMerkleTree

vWitnessMerkleTree

vtx

class bitcoin.core.**CoreChainParams**
Bases: object

Define consensus-critical parameters of a given instance of the Bitcoin system

GENESIS_BLOCK = None

MAX_MONEY = None

NAME = None

PROOF_OF_WORK_LIMIT = None

SUBSIDY_HALVING_INTERVAL = None

class bitcoin.core.**CoreMainParams**
Bases: *bitcoin.core.CoreChainParams*

GENESIS_BLOCK = CBlock(1, lx(000000000000000000000000000000000000000000000000000000000000000000

MAX_MONEY = 2100000000000000

NAME = 'mainnet'

PROOF_OF_WORK_LIMIT = 26959946667150639794667015087019630673637144422540572481103610243

SUBSIDY_HALVING_INTERVAL = 210000

class bitcoin.core.**CoreTestNetParams**
Bases: *bitcoin.core.CoreMainParams*

GENESIS_BLOCK = CBlock(1, lx(0000000000000000000000000000000000000000000000000000000000000000

NAME = 'testnet'

class bitcoin.core.**CoreRegTestParams**
Bases: *bitcoin.core.CoreTestNetParams*

GENESIS_BLOCK = CBlock(1, lx(0000000000000000000000000000000000000000000000000000000000000000

NAME = 'regtest'

PROOF_OF_WORK_LIMIT = 578960446186580977117854925043439539266349923328202820197287920039

SUBSIDY_HALVING_INTERVAL = 150

exception bitcoin.core.**CheckTransactionError**
Bases: *bitcoin.core.ValidationError*

bitcoin.core.**CheckTransaction**(*tx*)
Basic transaction checks that don't depend on any context.

Raises CheckTransactionError

exception bitcoin.core.**CheckBlockHeaderError**
Bases: *bitcoin.core.ValidationError*

exception bitcoin.core.**CheckProofOfWorkError**
Bases: *bitcoin.core.CheckBlockHeaderError*

bitcoin.core.**CheckProofOfWork**(*hash*, *nBits*)
Check a proof-of-work

Raises CheckProofOfWorkError

bitcoin.core.**CheckBlockHeader**(*block_header*, *fCheckPoW=True*, *cur_time=None*)
Context independent CBlockHeader checks.

fCheckPoW - Check proof-of-work.

cur_time - Current time. Defaults to time.time()

Raises CBlockHeaderError if block header is invalid.

exception bitcoin.core.**CheckBlockError**
Bases: *bitcoin.core.CheckBlockHeaderError*

bitcoin.core.**GetLegacySigOpCount**(*tx*)

bitcoin.core.**CheckBlock**(*block*, *fCheckPoW=True*, *fCheckMerkleRoot=True*, *cur_time=None*)
Context independent CBlock checks.

CheckBlockHeader() is called first, which may raise a CheckBlockHeader exception, followed the block tests. CheckTransaction() is called for every transaction.

fCheckPoW - Check proof-of-work.

fCheckMerkleRoot - Check merkle root and witness merkle root matches transactions.

- Check witness commitment in coinbase

cur_time - Current time. Defaults to time.time()

1.2.2 key

ECC secp256k1 crypto routines

WARNING: This module does not mlock() secrets; your private keys may end up on disk in swap! Use with caution!

class bitcoin.core.key.**CECKey**

Wrapper around OpenSSL's EC_KEY

POINT_CONVERSION_COMPRESSED = 2

POINT_CONVERSION_UNCOMPRESSED = 4

get_ecdh_key(*other_pubkey*, *kdf=<function <lambda>>*)

get_privkey()

get_pubkey()

get_raw_ecdh_key(*other_pubkey*)

recover(*sigR*, *sigS*, *msg*, *msglen*, *recid*, *check*)

Perform ECDSA key recovery (see SEC1 4.1.6) for curves over (mod p)-fields

recid selects which key is recovered

if check is non-zero, additional checks are performed

set_compressed(*compressed*)

set_privkey(*key*)

set_pubkey(*key*)

set_secretbytes(*secret*)

sign(*hash*)

sign_compact(*hash*)

signature_to_low_s(*sig*)

verify(*hash*, *sig*)

Verify a DER signature

class bitcoin.core.key.**CPubKey**

Bases: str

An encapsulated public key

Attributes:

is_valid - Corresponds to CPubKey.IsValid()

is_fullyvalid - Corresponds to CPubKey.IsFullyValid()

is_compressed - Corresponds to CPubKey.IsCompressed()

is_compressed

is_valid

classmethod recover_compact(*hash*, *sig*)

Recover a public key from a compact signature.

verify(*hash*, *sig*)

1.2.3 script

Scripts

Functionality to build scripts, as well as SignatureHash(). Script evaluation is in bitcoin.core.scripteval

class bitcoin.core.script.**CScriptOp**
Bases: int

A single script opcode

decode_op_n()
Decode a small integer opcode, returning an integer

static encode_op_n(*n*)
Encode a small integer op, returning an opcode

static encode_op_pushdata(*d*)
Encode a PUSHDATA op, returning bytes

is_small_int()
Return true if the op pushes a small integer to the stack

exception bitcoin.core.script.**CScriptInvalidError**
Bases: exceptions.Exception

Base class for CScript exceptions

exception bitcoin.core.script.**CScriptTruncatedPushDataError**(*msg*, *data*)
Bases: *bitcoin.core.script.CScriptInvalidError*

Invalid pushdata due to truncation

class bitcoin.core.script.**CScript**
Bases: str

Serialized script

A bytes subclass, so you can use this directly whenever bytes are accepted. Note that this means that indexing does *not* work - you'll get an index by byte rather than opcode. This format was chosen for efficiency so that the general case would not require creating a lot of little CScriptOP objects.

iter(script) however does iterate by opcode.

GetSigOpCount(*fAccurate*)
Get the SigOp count.

fAccurate - Accurately count CHECKMULTISIG, see BIP16 for details.

Note that this is consensus-critical.

has_canonical_pushes()
Test if script only uses canonical pushes

Not yet consensus critical; may be in the future.

is_p2sh()
Test if the script is a p2sh scriptPubKey

Note that this test is consensus-critical.

is_push_only()
Test if the script only contains pushdata ops

Note that this test is consensus-critical.

Scripts that contain invalid pushdata ops return False, matching the behavior in Bitcoin Core.

is_unspendable()
Test if the script is provably unspendable

is_valid()
Return True if the script is valid, False otherwise

The script is valid if all PUSHDATA's are valid; invalid opcodes do not make is_valid() return False.

is_witness_scriptpubkey()
Returns true if this is a scriptpubkey signaling segregated witness data.

is_witness_v0_keyhash()
Returns true if this is a scriptpubkey for V0 P2WPKH.

is_witness_v0_nested_keyhash()
Returns true if this is a scriptpubkey for V0 P2WPKH embedded in P2SH.

is_witness_v0_nested_scripthash()
Returns true if this is a scriptpubkey for V0 P2WSH embedded in P2SH.

is_witness_v0_scripthash()
Returns true if this is a scriptpubkey for V0 P2WSH.

join(*iterable*) → string
Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

raw_iter()
Raw iteration

Yields tuples of (opcode, data, sop_idx) so that the different possible PUSHDATA encodings can be accurately distinguished, as well as determining the exact opcode byte indexes. (sop_idx)

to_p2sh_scriptPubKey(*checksize=True*)
Create P2SH scriptPubKey from this redeemScript

That is, create the P2SH scriptPubKey that requires this script as a redeemScript to spend.

checksize - Check if the redeemScript is larger than the 520-byte max pushdata limit; raise ValueError if limit exceeded.

Since a >520-byte PUSHDATA makes EvalScript() fail, it's not actually possible to redeem P2SH outputs with redeem scripts >520 bytes.

witness_version()
Returns the witness version on [0,16].

class bitcoin.core.script.**CScriptWitness**(*stack=()*)
Bases: *bitcoin.core.serialize.ImmutableSerializable*

An encoding of the data elements on the initial stack for (segregated witness)

is_null()

stack

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

bitcoin.core.script.**FindAndDelete**(*script*, *sig*)
Consensus critical, see FindAndDelete() in Satoshi codebase

bitcoin.core.script.**RawSignatureHash**(*script*, *txTo*, *inIdx*, *hashtype*)
Consensus-correct SignatureHash

Returns (hash, err) to precisely match the consensus-critical behavior of the SIGHASH_SINGLE bug. (inIdx is *not* checked for validity)

If you're just writing wallet software you probably want SignatureHash() instead.

bitcoin.core.script.**SignatureHash**(*script*, *txTo*, *inIdx*, *hashtype*, *amount=None*, *sigversion=0*)
Calculate a signature hash

'Cooked' version that checks if inIdx is out of bounds - this is *not* consensus-correct behavior, but is what you probably want for general wallet use.

bitcoin.core.script.**IsLowDERSignature**(*sig*)
Loosely correlates with IsLowDERSignature() from script/interpreter.cpp Verifies that the S value in a DER signature is the lowest possible value. Used by BIP62 malleability fixes.

1.2.4 scripteval

Script evaluation

Be warned that there are highly likely to be consensus bugs in this code; it is unlikely to match Satoshi Bitcoin exactly. Think carefully before using this module.

exception bitcoin.core.scripteval.**EvalScriptError**(*msg*, *sop=None*, *sop_data=None*, *sop_pc=None*, *stack=None*, *scriptIn=None*, *txTo=None*, *inIdx=None*, *flags=None*, *altstack=None*, *vfExec=None*, *pbegincodehash=None*, *nOpCount=None*)
Bases: *bitcoin.core.ValidationError*

Base class for exceptions raised when a script fails during EvalScript()

The execution state just prior the opcode raising the is saved. (if available)

exception bitcoin.core.scripteval.**MaxOpCountError**(***kwargs*)
Bases: *bitcoin.core.scripteval.EvalScriptError*

exception bitcoin.core.scripteval.**MissingOpArgumentsError**(*opcode*, *s*, *n*, ***kwargs*)
Bases: *bitcoin.core.scripteval.EvalScriptError*

Missing arguments

exception bitcoin.core.scripteval.**ArgumentsInvalidError**(*opcode*, *msg*, ***kwargs*)
Bases: *bitcoin.core.scripteval.EvalScriptError*

Arguments are invalid

exception bitcoin.core.scripteval.**VerifyOpFailedError**(*opcode*, ***kwargs*)
Bases: *bitcoin.core.scripteval.EvalScriptError*

A VERIFY opcode failed

bitcoin.core.scripteval.**EvalScript**(*stack*, *scriptIn*, *txTo*, *inIdx*, *flags=()*)
Evaluate a script

stack - Initial stack

scriptIn - Script

txTo - Transaction the script is a part of

inIdx - txin index of the scriptSig

flags - SCRIPT_VERIFY_* flags to apply

exception bitcoin.core.scripteval.**VerifyScriptError**
Bases: *bitcoin.core.ValidationError*

bitcoin.core.scripteval.**VerifyScript**(*scriptSig*, *scriptPubKey*, *txTo*, *inIdx*, *flags=()*)
Verify a scriptSig satisfies a scriptPubKey

scriptSig - Signature

scriptPubKey - PubKey

txTo - Spending transaction

inIdx - Index of the transaction input containing scriptSig

Raises a ValidationError subclass if the validation fails.

exception bitcoin.core.scripteval.**VerifySignatureError**
Bases: *bitcoin.core.ValidationError*

bitcoin.core.scripteval.**VerifySignature**(*txFrom*, *txTo*, *inIdx*)
Verify a scriptSig signature can spend a txout

Verifies that the scriptSig in txTo.vin[inIdx] is a valid scriptSig for the corresponding COutPoint in transaction txFrom.

1.2.5 serialize

Serialization routines

You probabably don't need to use these directly.

bitcoin.core.serialize.**Hash**(*msg*)
SHA256^2)(msg) -> bytes

bitcoin.core.serialize.**Hash160**(*msg*)
RIPEME160(SHA256(msg)) -> bytes

exception bitcoin.core.serialize.**SerializationError**
Bases: exceptions.Exception

Base class for serialization errors

exception bitcoin.core.serialize.**SerializationTruncationError**
Bases: *bitcoin.core.serialize.SerializationError*

Serialized data was truncated

Thrown by deserialize() and stream_deserialize()

exception bitcoin.core.serialize.**DeserializationExtraDataError**(*msg*, *obj*, *padding*)
Bases: *bitcoin.core.serialize.SerializationError*

Deserialized data had extra data at the end

Thrown by deserialize() when not all data is consumed during deserialization. The deserialized object and extra padding not consumed are saved.

bitcoin.core.serialize.**ser_read**(*f*, *n*)
Read from a stream safely

Raises SerializationError and SerializationTruncationError appropriately. Use this instead of f.read() in your classes stream_(de)serialization() functions.

class bitcoin.core.serialize.**Serializable**
Bases: object

Base class for serializable objects

GetHash()
Return the hash of the serialized object

classmethod deserialize(*buf*, *allow_padding=False*, *params={}*)
Deserialize bytes, returning an instance

allow_padding - Allow buf to include extra padding. (default False)

If allow_padding is False and not all bytes are consumed during deserialization DeserializationExtraDataError will be raised.

serialize(*params={}*)
Serialize, returning bytes

classmethod stream_deserialize(*f*, ***kwargs*)
Deserialize from a stream

stream_serialize(*f*, ***kwargs*)
Serialize to a stream

class bitcoin.core.serialize.**ImmutableSerializable**
Bases: *bitcoin.core.serialize.Serializable*

Immutable serializable object

GetHash()
Return the hash of the serialized object

class bitcoin.core.serialize.**Serializer**
Bases: object

Base class for object serializers

classmethod deserialize(*buf*)

classmethod serialize(*obj*)

classmethod stream_deserialize(*f*)

classmethod stream_serialize(*obj*, *f*)

class bitcoin.core.serialize.**VarIntSerializer**
Bases: *bitcoin.core.serialize.Serializer*

Serialization of variable length ints

classmethod stream_deserialize(*f*)

classmethod stream_serialize(*i*, *f*)

class bitcoin.core.serialize.**BytesSerializer**
Bases: *bitcoin.core.serialize.Serializer*

Serialization of bytes instances

classmethod stream_deserialize(*f*)

classmethod stream_serialize(*b*, *f*)

class bitcoin.core.serialize.**VectorSerializer**
Bases: *bitcoin.core.serialize.Serializer*

Base class for serializers of object vectors

classmethod stream_deserialize(*inner_cls*, *f*, *inner_params={}*)

classmethod stream_serialize(*inner_cls*, *objs*, *f*, *inner_params={}*)

class bitcoin.core.serialize.**uint256VectorSerializer**
Bases: *bitcoin.core.serialize.Serializer*

Serialize vectors of uint256

classmethod stream_deserialize(*f*)

classmethod stream_serialize(*uints*, *f*)

class bitcoin.core.serialize.**intVectorSerializer**
Bases: *bitcoin.core.serialize.Serializer*

classmethod stream_deserialize(*f*)

classmethod stream_serialize(*ints*, *f*)

class bitcoin.core.serialize.**VarStringSerializer**
Bases: *bitcoin.core.serialize.Serializer*

Serialize variable length byte strings

classmethod stream_deserialize(*f*)

classmethod stream_serialize(*s*, *f*)

bitcoin.core.serialize.**uint256_from_str**(*s*)
Convert bytes to uint256

bitcoin.core.serialize.**uint256_from_compact**(*c*)
Convert compact encoding to uint256

Used for the nBits compact encoding of the target in the block header.

bitcoin.core.serialize.**compact_from_uint256**(*v*)
Convert uint256 to compact encoding

bitcoin.core.serialize.**uint256_to_str**(*u*)

bitcoin.core.serialize.**uint256_to_shortstr**(*u*)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

D

E

F

G

H

I

J

L

M

N

P

R

S

T

U

V

W

X