python-bitcoinlib Documentation

Release 0.8.1-dev

Isaac Cook

Sep 24, 2017

Contents

This Python2/3 library provides an easy interface to the bitcoin data structures and protocol. The approach is low-level and “ground up”, with a focus on providing tools to manipulate the internals of how Bitcoin works.

CHAPTER 1

Getting Started

Requirements

To install python-bitcoinlib:

```
sudo apt-get install libssl-dev
pip install python-bitcoinlib
# Or for the latest git version
pip install git+https://github.com/petertodd/python-bitcoinlib
```

The RPC interface, `bitcoin.rpc`, is designed to work with Bitcoin Core v0.9. Older versions mostly work but there do exist some incompatibilities.

Example Code

See examples/ directory. For instance this example creates a transaction spending a pay-to-script-hash transaction output:

```
$ PYTHONPATH=. examples/spend-pay-to-script-hash-txout.py
<hex-encoded transaction>
```

Also see dust-b-gone for a simple example of Bitcoin Core wallet interaction through the RPC interface: https://github.com/petertodd/dust-b-gone

Selecting the chain to use

Do the following:

```
import bitcoin
bitcoin.SelectParams(NAME)
```

Where NAME is one of ‘testnet’, ‘mainnet’, or ‘regtest’. The chain currently selected is a global variable that changes behavior everywhere, just like in the Satoshi codebase.

Unit tests

Under bitcoin/tests using test data from Bitcoin Core. To run them:

```
tox
```

CHAPTER 2

Advisories

Mutable vs. Immutable objects

Like the Bitcoin Core codebase CTransaction is immutable and CMutableTransaction is mutable; unlike the Bitcoin Core codebase this distinction also applies to COutPoint, CTxIn, CTxOut, and CBlock.

Endianness Gotchas

Rather confusingly Bitcoin Core shows transaction and block hashes as little-endian hex rather than the big-endian the rest of the world uses for SHA256. python-bitcoinlib provides the convenience functions x() and lx() in bitcoin.core to convert from big-endian and little-endian hex to raw bytes to accomodate this. In addition see b2x() and b2lx() for conversion from bytes to big/little-endian hex.

CHAPTER 3

Consensus-critical API

Core

Basic core definitions, datastructures, and (context-independent) validation

class `bitcoin.core.`**`CBlock`**(*nVersion=2*, *hashPrevBlock='x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x*
hashMerkleRoot='x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x0
nTime=0, *nBits=0*, *nNonce=0*, *vtx=()*)

A block including all transactions in it

`GetHash`()

Return the block hash

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

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

`calc_merkle_root`()

Calculate the 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.

classmethod **`stream_deserialize`**(*f*)

`stream_serialize`(*f*)

`vMerkleTree`

`vtx`

class `bitcoin.core.`**`CBlockHeader`**(*nVersion=2*, *hashPrevBlock='x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x0*
hashMerkleRoot='x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x0
nTime=0, *nBits=0*, *nNonce=0*)

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.`**`CMutableOutPoint`**(*hash='x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x0*
n=4294967295)

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.`**`CMutableTransaction`**(*vin=None*, *vout=None*, *nLockTime=0*, *nVersion=1*)

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.`**`CMutableTxIn`**(*prevout=None*, *scriptSig=CScript([])*, *nSequence=4294967295*)

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.`**`CMutableTxOut`**(*nValue=-1*, *scriptPubKey=CScript([])*)

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.**COutPoint**(*hash='x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x0*
n=4294967295)
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 output 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.**CTransaction**(*vin=()*, *vout=()*, *nLockTime=0*, *nVersion=1*)
A transaction

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.

is_coinbase()

nLockTime

nVersion

classmethod stream_deserialize(*f*)

stream_serialize(*f*)

vin

vout

class bitcoin.core.**CTxIn**(*prevout=COutPoint()*, *scriptSig=CScript([])*, *nSequence=4294967295*)
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.`**`CTxOut`**(*nValue=-1*, *scriptPubKey=CScript([])*)

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*)

`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 matches transactions. cur_time - Current time. Defaults to time.time()

exception `bitcoin.core.`**`CheckBlockError`**

`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.`**`CheckBlockHeaderError`**

`bitcoin.core.`**`CheckProofOfWork`**(*hash*, *nBits*)

Check a proof-of-work

Raises CheckProofOfWorkError

exception `bitcoin.core.`**`CheckProofOfWorkError`**

`bitcoin.core.`**`CheckTransaction`**(*tx*)

Basic transaction checks that don't depend on any context.

Raises CheckTransactionError

exception `bitcoin.core.`**`CheckTransactionError`**

class `bitcoin.core.`**`CoreChainParams`**

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

`GENESIS_BLOCK` **= None**

`NAME` **= None**

`PROOF_OF_WORK_LIMIT` **= None**

`SUBSIDY_HALVING_INTERVAL` **= None**

class `bitcoin.core.`**`CoreMainParams`**

GENESIS_BLOCK = CBlock(1, lx(0000000000000000000000000000000000000000000000000000000000000000), lx(4a5e1e4

NAME = u'mainnet'

PROOF_OF_WORK_LIMIT = 26959946667150639794667015087019630673637144422540572481103610249215L

SUBSIDY_HALVING_INTERVAL = 210000

class `bitcoin.core.`**`CoreRegTestParams`**

GENESIS_BLOCK = CBlock(1, lx(0000000000000000000000000000000000000000000000000000000000000000), lx(4a5e1e4

NAME = u'regtest'

PROOF_OF_WORK_LIMIT = 57896044618658097711785492504343953926634992332820282019728792003956564819967L

SUBSIDY_HALVING_INTERVAL = 150

class `bitcoin.core.`**`CoreTestNetParams`**

GENESIS_BLOCK = CBlock(1, lx(0000000000000000000000000000000000000000000000000000000000000000), lx(4a5e1e4

NAME = u'testnet'

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

`bitcoin.core.`**`MoneyRange`**(*nValue*)

exception `bitcoin.core.`**`ValidationError`**

Base class for all blockchain validation errors

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

`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.`**`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.`**`str_money_value`**(*value*)

Convert an integer money value to a fixed point string

`bitcoin.core.`**`x`**(*h*)

Convert a hex string to bytes

Bignum handling

Bignum routines

`bitcoin.core.bignum.`**`bin2bn`**(*s*)

`bitcoin.core.bignum.`**`bn2bin`**(*v*)

`bitcoin.core.bignum.`**`bn2mpi`**(*v*)

`bitcoin.core.bignum.`**`bn2vch`**`(`*v*`)`

`bitcoin.core.bignum.`**`bn_bytes`**`(`*v*, *have_ext=False*`)`

`bitcoin.core.bignum.`**`mpi2bn`**`(`*s*`)`

`bitcoin.core.bignum.`**`mpi2vch`**`(`*s*`)`

`bitcoin.core.bignum.`**`vch2bn`**`(`*s*`)`

`bitcoin.core.bignum.`**`vch2mpi`**`(`*s*`)`

ECC Public Keys

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*`)`

`set_compressed``(`*compressed*`)`

`set_privkey``(`*key*`)`

`set_pubkey``(`*key*`)`

`set_secretbytes``(`*secret*`)`

`sign``(`*hash*`)`

`verify``(`*hash*, *sig*`)`

Verify a DER signature

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

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`

`verify``(`*hash*, *sig*`)`

Scripts and Opcodes

Scripts

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

class `bitcoin.core.script.`**`CScript`**

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.

`join`(*iterable*)

`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.

exception `bitcoin.core.script.`**`CScriptInvalidError`**
Base class for CScript exceptions

class `bitcoin.core.script.`**`CScriptOp`**
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.`**`CScriptTruncatedPushDataError`**`(`*msg*, *data*`)`
Invalid pushdata due to truncation

`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*`)`
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.

Script evaluation/verification

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.`**`ArgumentsInvalidError`**`(`*opcode*, *msg*, ***kwargs*`)`
Arguments are invalid

`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.`**`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*`)`
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*)

exception `bitcoin.core.scripteval.`**`MissingOpArgumentsError`**(*opcode*, *s*, *n*, ***kwargs*)
Missing arguments

exception `bitcoin.core.scripteval.`**`VerifyOpFailedError`**(*opcode*, ***kwargs*)
A VERIFY opcode failed

`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.`**`VerifyScriptError`**

`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.

exception `bitcoin.core.scripteval.`**`VerifySignatureError`**

Serialization

Serialization routines

You probably don't need to use these directly.

class `bitcoin.core.serialize.`**`BytesSerializer`**
Serialization of bytes instances

classmethod **`stream_deserialize`**(*f*)

classmethod **`stream_serialize`**(*b*, *f*)

exception `bitcoin.core.serialize.`**`DeserializationExtraDataError`**(*msg*, *obj*, *padding*)
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.`**`Hash`**(*msg*)
SHA256^2)(msg) -> bytes

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

class `bitcoin.core.serialize.`**`ImmutableSerializable`**
Immutable serializable object

`GetHash`()
Return the hash of the serialized object

class `bitcoin.core.serialize.`**`Serializable`**
Base class for serializable objects

GetHash()
Return the hash of the serialized object

classmethod deserialize(*buf*, *allow_padding=False*)
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 DeserializationExtra-DataError will be raised.

serialize()
Serialize, returning bytes

classmethod stream_deserialize(*f*)
Deserialize from a stream

stream_serialize(*f*)
Serialize to a stream

exception bitcoin.core.serialize.**SerializationError**
Base class for serialization errors

exception bitcoin.core.serialize.**SerializationTruncationError**
Serialized data was truncated

Thrown by deserialize() and stream_deserialize()

class bitcoin.core.serialize.**Serializer**
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**
Serialization of variable length ints

classmethod stream_deserialize(*f*)

classmethod stream_serialize(*i*, *f*)

class bitcoin.core.serialize.**VarStringSerializer**
Serialize variable length strings

classmethod stream_deserialize(*f*)

classmethod stream_serialize(*s*, *f*)

class bitcoin.core.serialize.**VectorSerializer**
Base class for serializers of object vectors

classmethod stream_deserialize(*inner_cls*, *f*)

classmethod stream_serialize(*inner_cls*, *objs*, *f*)

class bitcoin.core.serialize.**intVectorSerialzer**

classmethod stream_deserialize(*f*)

classmethod stream_serialize(*ints*, *f*) `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.`**`uint256VectorSerializer`**

Serialize vectors of uint256

classmethod **`stream_deserialize`**(*f*)

classmethod **`stream_serialize`**(*uints*, *f*)

`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.`**`uint256_from_str`**(*s*)

Convert bytes to uint256

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

CHAPTER 4

Non-consensus-critical API

Chain selection

class `bitcoin.`**`MainParams`**

> **`BASE58_PREFIXES`** = **{u'SECRET_KEY': 128, u'SCRIPT_ADDR': 5, u'PUBKEY_ADDR': 0}**
>
> **`DEFAULT_PORT`** = **8333**
>
> **`DNS_SEEDS`** = **((u'bitcoin.sipa.be', u'seed.bitcoin.sipa.be'), (u'bluematt.me', u'dnsseed.bluematt.me'), (u'dashjr.org', u'd**
>
> **`MESSAGE_START`** = **'\xf9\xbe\xb4\xd9'**
>
> **`RPC_PORT`** = **8332**

class `bitcoin.`**`RegTestParams`**

> **`BASE58_PREFIXES`** = **{u'SECRET_KEY': 239, u'SCRIPT_ADDR': 196, u'PUBKEY_ADDR': 111}**
>
> **`DEFAULT_PORT`** = **18444**
>
> **`DNS_SEEDS`** = **()**
>
> **`MESSAGE_START`** = **'\xfa\xbf\xb5\xda'**
>
> **`RPC_PORT`** = **18332**

`bitcoin.`**`SelectParams`**(*name*)

> Select the chain parameters to use
>
> name is one of 'mainnet', 'testnet', or 'regtest'
>
> Default chain is 'mainnet'

class `bitcoin.`**`TestNetParams`**

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

DEFAULT_PORT = 18333

DNS_SEEDS = ((u'bitcoin.petertodd.org', u'testnet-seed.bitcoin.petertodd.org'), (u'bluematt.me', u'testnet-seed.bluematt

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

RPC_PORT = 18332

Base58 encoding

Base58 encoding and decoding

exception `bitcoin.base58.`**`Base58ChecksumError`**
Raised on Base58 checksum errors

exception `bitcoin.base58.`**`Base58Error`**

class `bitcoin.base58.`**`CBase58Data`**(*s*)
Base58-encoded data

Includes a version and checksum.

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.

exception `bitcoin.base58.`**`InvalidBase58Error`**
Raised on generic invalid base58 data, such as bad characters.

Checksum failures raise Base58ChecksumError specifically.

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

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

Bloom filters

Bloom filter support

class `bitcoin.bloom.`**`CBloomFilter`**(*nElements*, *nFPRate*, *nTweak*, *nFlags*)

`IsRelevantAndUpdate`(*tx*, *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*)

`bitcoin.bloom.`**MurmurHash3** (*x86_32*)

Used for bloom filters. See http://code.google.com/p/smhasher/source/browse/trunk/MurmurHash3.cpp

`bitcoin.bloom.`**ROTL32** (*x*, *r*)

Network communication

class `bitcoin.net.`**CAddress** (*protover=60002*)

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

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

class `bitcoin.net.`**CAlert**

classmethod stream_deserialize (*f*)

stream_serialize (*f*)

class `bitcoin.net.`**CBlockLocator** (*protover=60002*)

classmethod stream_deserialize (*f*)

stream_serialize (*f*)

class `bitcoin.net.`**CInv**

classmethod stream_deserialize (*f*)

stream_serialize (*f*)

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

class `bitcoin.net.`**CUnsignedAlert**

classmethod stream_deserialize (*f*)

stream_serialize (*f*)

Network messages

members

undoc-members

Bitcoin Core RPC interface

Bitcoin Core RPC support

members

undoc-members

Wallet-related code

Wallet-related functionality

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

members

undoc-members

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index