pycoin Documentation

Release unknown

Richard Kiss

Sep 22, 2023

Contents

Release vunknown. (*Installation*)

This documentation is a work-in-progress, and your contributions are welcome at <https://github.com/richardkiss/pycoin>.

The pycoin library implements many of utilities useful when dealing with bitcoin and some bitcoin-like alt-coins. It has been tested with Python 2.7, 3.6 and 3.7.

CHAPTER 1

A Note about Naming

Many of the names of data structures in bitcoin, like “script pubkey”, are derived from names that came from the original C++ source code (known here as the “satoshi client”). Often times, it appears these names were chosen out of expediency, and frequently they are overly generic which makes it difficult to understand or remember what they are for.

With the benefit of time and a lack of legacy users, pycoin has had the luxury to come up with alternative names for many of these structures that more clearly suggest their actual use. We will use the pycoin names for these structures in this documentation, but will also make mention of the “official” names used by the satoshi client.

CHAPTER 2

Networks

Although pycoin is primarily engineered for bitcoin, it supports various altcoins to various degrees (and has the capability to support altcoins fully… contributions welcome!).

Contents:

2.1 Installation

To install pycoin, run this command the terminal:

```
$ pip install pycoin
```

If you don't have pip installed, check out this tutorial.

To see if pycoin is correctly installed, try a command-line tool:

```
$ ku 1
```

You should see several lines of output, describing information about the bitcoin key corresponding to private key 1.

2.2 pycoin API

Note

pycoin started out as a loose collection of utilities, and is slowly evolving to be a more cohesive API.

2.2.1 Networks

A "network" is a particular coin, such as Bitcoin Mainnet or Bitcoin Testnet. There are two main ways to fetch a network:

```
from pycoin.symbols.btc import network
```

or

```
from pycoin.networks.registry import network_for_netcode
network = network_for_netcode("BTC")
```

These are the starting points. Nearly all API for a network can be accessed by drilling down below the network object.

Some useful network attributes include:

network.Tx the class for a transaction

network.Block the class for a block

network.message message api, to pack and parse messages used by bitcoin's peer-to-peer protocol

network.keychain an object to aggregate private key information, useful for signing transactions

network.parse for parsing human-readable items like keys, WIFs, addresses

network.contract api for creating standard scripts in bitcoin

network.address includes for_script API for turning a TxOut puzzle script into an address

network.keys api for creating private keys, public keys, and hierarchical keys, both BIP32 and Electrum

network.generator api for signing messages and verifying signed messages

network.validator api for validating whether or not transactions are correctly signed

network.tx_utils shortcuts for building and signing transactions

network.who_signed utilities to determine which public keys have signed partially signed multisig transactions

2.2.2 network.Tx

class pycoin.coins.Tx.**Tx**(**args*, ***kwargs*)

class TxIn(**args*, ***kwargs*)
The input part of a Tx that specifies where funds come from.

class TxOut(**args*, ***kwargs*)
The output part of a Tx that specifies where funds go.

as_bin(**args*, ***kwargs*)
Returns a binary blob containing the streamed transaction.

For information about the parameters, see *Tx.stream*

Returns binary blob that would parse to the given transaction

as_hex(**args*, ***kwargs*)
Returns a text string containing the streamed transaction encoded as hex.

For information about the parameters, see *Tx.stream*

Returns hex string that would parse to the given transaction

bad_solution_count(**args*, ***kwargs*)
Return a count of how many *TxIn* objects are not correctly solved.

check()
Basic checks that don't depend on network or block context.

classmethod from_bin(*blob*)
Return the Tx for the given binary blob.

Parameters **blob** – a binary blob containing a transaction streamed in standard form. The blob may also include the unspents (a nonstandard extension, optionally written by *Tx.stream*), and they will also be parsed.

Returns *Tx*

If parsing fails, an exception is raised.

classmethod from_hex(*hex_string*)
Return the Tx for the given hex string.

Parameters **hex_string** – a hex string containing a transaction streamed in standard form. The blob may also include the unspents (a nonstandard extension, optionally written by *Tx.stream*), and they will also be parsed.

Returns *Tx*

If parsing fails, an exception is raised.

hash(*hash_type=None*)
Return the hash for this Tx object.

id()
Return the human-readable hash for this Tx object.

classmethod parse(*f*)
Parse a transaction Tx from the file-like object f.

set_unspents(*unspents*)
Set the unspent inputs for a transaction.

Parameters **unspents** – a list of *TxOut* (or the subclass `Spendable`) objects corresponding to the *TxIn* objects for this transaction (same number of items in each list)

sign(**args*, ***kwargs*)
Sign all transaction inputs. The parameters vary depending upon the way the coins being spent are encumbered.

stream(*f*, **args*, ***kwargs*)
Stream a transaction Tx to the file-like object f.

class pycoin.coins.Tx.**TxIn**(**args*, ***kwargs*)
The input part of a Tx that specifies where funds come from.

class pycoin.coins.Tx.**TxOut**(**args*, ***kwargs*)
The output part of a Tx that specifies where funds go.

2.2.3 network.Block

class pycoin.block.**Block**(*version*, *previous_block_hash*, *merkle_root*, *timestamp*, *difficulty*, *nonce*)
A Block is an element of the Bitcoin chain.

as_bin()
Return the block (or header) as binary.

as_hex()
Return the block (or header) as hex.

check_merkle_hash()
Raise a BadMerkleRootError if the Merkle hash of the transactions does not match the Merkle hash included in the block.

hash()
Calculate the hash for the block header. Note that this has the bytes in the opposite order from how the header is usually displayed (so the long string of 00 bytes is at the end, not the beginning).

id()
Returns the hash of the block displayed with the bytes in the order they are usually displayed in.

classmethod parse(*f*, *include_transactions=True*, *include_offsets=None*, *check_merkle_hash=True*)
Parse the Block from the file-like object

classmethod parse_as_header(*f*)
Parse the Block header from the file-like object

previous_block_id()
Returns the hash of the previous block, with the bytes in the order they are usually displayed in.

stream(*f*)
Stream the block header in the standard way to the file-like object f. The Block subclass also includes the transactions.

stream_header(*f*)
Stream the block header in the standard way to the file-like object f.

2.2.4 network.message

message.**pack**(***kwargs*)

message.**parse**(*data*)

2.2.5 InvItem Module

class pycoin.message.InvItem.**InvItem**(*item_type*, *data*, *dont_check=False*)
Bases: object

classmethod parse(*f*)

stream(*f*)

2.2.6 PeerAddress Module

class pycoin.message.PeerAddress.**PeerAddress**(*services*, *ip_bin*, *port*)
Bases: object

host()

classmethod parse(*f*)

stream(*f*)

pycoin.message.PeerAddress.**ip_bin_to_ip4_addr**(*ip_bin*)

pycoin.message.PeerAddress.**ip_bin_to_ip6_addr**(*ip_bin*)

2.2.7 network.keychain

class pycoin.key.Keychain.**Keychain**(*sqlite3_db=None*)

2.2.8 network.parse

class pycoin.networks.ParseAPI.**ParseAPI**(*network*, *bip32_prv_prefix=None*, *bip32_pub_prefix=None*, *bip49_prv_prefix=None*, *bip49_pub_prefix=None*, *bip84_prv_prefix=None*, *bip84_pub_prefix=None*, *address_prefix=None*, *pay_to_script_prefix=None*, *bech32_hrp=None*, *wif_prefix=None*, *sec_prefix=None*)

address(*s*)
Parse an address, any of p2pkh, p2sh, p2pkh_segwit, or p2sh_segwit. Return a *Contract*, or None.

bip32(*s*)
Parse a bip32 public key from a text string, either a seed, a prv or a pub. Return a BIP32 or None.

bip32_prv(*s*)
Parse a bip32 private key from a text string ("xprv" type). Return a BIP32 or None.

bip32_pub(*s*)
Parse a bip32 public key from a text string ("xpub" type). Return a BIP32 or None.

bip32_seed(*s*)
Parse a bip32 private key from a seed. Return a BIP32 or None.

bip49(*s*)
Parse a bip49 public key from a text string, either a seed, a prv or a pub. Return a BIP49 or None.

bip49_prv(*s*)
Parse a bip84 private key from a text string ("yprv" type). Return a BIP49 or None.

bip49_pub(*s*)
Parse a bip84 public key from a text string ("ypub" type). Return a BIP49 or None.

bip84(*s*)
Parse a bip84 public key from a text string, either a seed, a prv or a pub. Return a BIP84 or None.

bip84_prv(*s*)
Parse a bip84 private key from a text string ("zprv" type). Return a BIP84 or None.

bip84_pub(*s*)
Parse a bip84 public key from a text string ("zpub" type). Return a BIP84 or None.

electrum_prv(*s*)
Parse an electrum private key from a text string in seed form ("E:xxx" where xxx is a 64-character hex string). Return a ElectrumWallet or None.

electrum_pub(*s*)
Parse an electrum public key from a text string in seed form ("E:xxx" where xxx is a 128-character hex string). Return a ElectrumWallet or None.

electrum_seed(*s*)
Parse an electrum key from a text string in seed form ("E:xxx" where xxx is a 32-character hex string). Return a ElectrumWallet or None.

hd_seed(*s*)
Parse a bip32 private key from a seed. Return a BIP32 or None.

hierarchical_key(*s*)
Parse text as some kind of hierarchical key. Return a subclass of `Key`, or None.

input(*s*)
NOT YET SUPPORTED

p2pkh(*s*)
Parse a pay-to-public-key-hash address. Return a *Contract* or None.

p2pkh_segwit(*s*)
Parse a pay-to-pubkey-hash segwit address. Return a *Contract* or None.

p2sh(*s*)
Parse a pay-to-script-hash address. Return a *Contract* or None.

p2sh_segwit(*s*)
Parse a pay-to-script-hash segwit address. Return a *Contract* or None.

p2tr(*s*)
Parse a pay-to-taproot segwit address. Return a *Contract* or None.

parse_b58_hashed(*s*)
Override me to change how the b58 hashing check is done.

payable(*s*)
Parse text as either an address or a script to be compiled. Return a *Contract*, or None.

private_key(*s*)
Parse text as some kind of private key. Return a subclass of `Key`, or None.

public_key(*s*)
Parse text as either a public pair or an sec. Return a subclass of `Key`, or None.

public_pair(*s*)
Parse a public pair X/Y or X,Y where X is a coordinate and Y is a coordinate or the string “even” or “odd”. Return a `Key` or None.

script(*s*)
Parse a script by compiling it. Return a *Contract* or None.

script_preimage(*s*)
NOT YET SUPPORTED

sec(*s*)
Parse a public pair as a text SEC. Return a `Key` or None.

secret(*s*)
Parse text either a private key or a private hierarchical key. Return a subclass of `Key`, or None.

secret_exponent(*s*)
Parse an integer secret exponent. Return a `Key` or None.

spendable(*s*)
NOT YET SUPPORTED

tx(*s*)
NOT YET SUPPORTED

wif(*s*)
Parse a WIF. Return a `Key` or None.

2.2.9 network.contract

classmethod contract.**for_address**(*address*)

classmethod contract.**for_p2pk**(*sec*)

classmethod contract.**for_p2pkh**(*hash160*)

classmethod contract.**for_p2pkh_wit**(*hash160*)

classmethod contract.**for_p2sh**(*hash160*)

classmethod contract.**for_p2sh_wit**(*hash256*)

classmethod contract.**for_multisig**(*m*, *sec_keys*)

classmethod contract.**for_nulldata**(*data*)

classmethod contract.**for_p2s**(*underlying_script*)

classmethod contract.**for_p2s_wit**(*underlying_script*)

classmethod contract.**for_info**(*info*)

classmethod contract.**info_for_script**(*script*)

2.2.10 network.address

pycoin.symbols.btc.network.**address**

2.2.11 network.keys

keys.**public**(*is_compressed=None*)

keys.**private**(*is_compressed=True*)

classmethod keys.**bip32_seed**(*master_secret*)
Generate a Wallet from a master password.

classmethod keys.**bip32_deserialize**(*data*)

keys.**electrum_seed**()

keys.**electrum_private**()

class pycoin.symbols.btc.network.keys.**InvalidSecretExponentError**

class pycoin.symbols.btc.network.keys.**InvalidPublicPairError**

2.2.12 network.generator

Most bitcoin-like cryptocurrencies use an ECC group called secp256k1 for digital signatures. The ecdsa.secp256k1 generator for this group provides most of the functionality you will need.

```
from pycoin.symbols.btc import network
public_key = network.generator * 1
print(public_key)
```

For bitcoin, network.generator is pycoin.ecdsa.secp256k1.secp256k1_generator, which is an instance of a *Generator*.

2.2.13 network.msg

classmethod msg.**sign**(*key*, *message*, *verbose=False*)
Return a signature, encoded in Base64, which can be verified by anyone using the public key.

classmethod msg.**verify**(*key_or_address*, *signature*, *message=None*, *msg_hash=None*)
Take a signature, encoded in Base64, and verify it against a key object (which implies the public key), or a specific base58-encoded pubkey hash.

classmethod msg.**parse_signed**(*msg_in*)
Take an "armoured" message and split into the message body, signing address and the base64 signature. Should work on all altcoin networks, and should accept both Inputs.IO and Multibit formats but not Armory.

Looks like RFC2550 <https://www.ietf.org/rfc/rfc2440.txt> was an "inspiration" for this, so in case of confusion it's a reference, but I've never found a real spec for this. Should be a BIP really.

classmethod msg.**hash_for_signing**(*msg*)
Return a hash of msg, according to odd bitcoin method: double SHA256 over a bitcoin encoded stream of two strings: a fixed magic prefix and the actual message.

classmethod msg.**signature_for_message_hash**(*secret_exponent*, *msg_hash*, *is_compressed*)
Return a signature, encoded in Base64, of msg_hash.

classmethod msg.**pair_for_message_hash**(*signature*, *msg_hash*)
Take a signature, encoded in Base64, and return the pair it was signed by. May raise EncodingError (from _decode_signature)

2.2.14 network.validator

class pycoin.symbols.btc.network.validator.**ScriptError**

class pycoin.symbols.btc.network.validator.**ValidationFailureError**

pycoin.symbols.btc.network.validator.**errno**
alias of pycoin.satoshi.errno

pycoin.symbols.btc.network.validator.**flags**
alias of pycoin.satoshi.flags

2.2.15 network.tx_utils

tx_utils.**create_tx**(***kwargs*)

tx_utils.**sign_tx**(***kwargs*)

tx_utils.**create_signed_tx**(***kwargs*)

tx_utils.**split_with_remainder**(***kwargs*)

tx_utils.**distribute_from_split_pool**(*fee*)

Parameters

- **tx** – a transaction
- **fee** – integer, satoshi value to set aside for transaction fee

This function looks at TxOut items of a transaction tx and and puts TxOut items with a coin value of zero into a "split pool". Funds not explicitly claimed by the fee or other TxOut items are shared as equally as possible among the split pool. If the amount to be split does not divide evenly, some of the earlier TxOut items will get an extra satoshi. This function modifies *tx* in place.

2.2.16 network.who_signed

classmethod who_signed.**solution_blobs**(*tx*, *tx_in_idx*)
This iterator yields data blobs that appear in the last solution_script or the witness.

classmethod who_signed.**extract_signatures**(*tx*, *tx_in_idx*)

classmethod who_signed.**extract_secs**(*tx*, *tx_in_idx*)

classmethod who_signed.**public_pairs_for_script**(*tx*, *tx_in_idx*, *generator*)
For a given script, iterate over and pull out public pairs encoded as sec values.

classmethod who_signed.**public_pairs_signed**(*tx*, *tx_in_idx*)

classmethod who_signed.**who_signed_tx**(*tx*, *tx_in_idx*)
Given a transaction (tx) an input index (tx_in_idx), attempt to figure out which addresses where used in signing (so far). This method depends on tx.unspents being properly configured. This should work on partially-signed MULTISIG transactions (it will return as many addresses as there are good signatures). Returns a list of (public_pairs, sig_type) pairs.

2.3 Command-line Tools

pycoin includes the command-line tools *ku* and *tx*.

For now, please refer to github.

2.4 Recipes

These recipes include some command-line utilities written with many comments and designed to be easy to follow. You can use them as a template for your own code.

For now, look at the source code and its comments on github.

2.5 A Bitcoin Primer

We will describe bitcoin specifically here. Many cryptocurrencies are variations of bitcoin that tweak a few details.

2.5.1 Overview

Bitcoin is a digital currency that will eventually generate a number of bitcoins slightly under 21 million (2.1e7). Each bitcoin can be further subdivided into ten million (1e8) quantum units each of which is known as a "satoshi".

The Bitcoin network is a distributed append-only database that is designed to append one block to the linked-list of blocks once every ten minutes. This database keeps track of *unspents* (more commonly known as "UTXOs" for *unspent transaction outputs*).

2.5.2 Unspents

An unspent can be thought of as a roll of coins of any number of satoshis protected by a *puzzle script* (more commonly known as a "script pubkey", because it almost always contains a reference to a public key). This puzzle is written in

a custom bitcoin stack-based low level scripting language, but is usually one of only a few common forms, including pay-to-public-key, and pay-to-script.

An unspent is a potential input to a transaction.

2.5.3 Unspent Database

The bitcoin database is a ledger of unspents. It doesn't explicitly define ownership of bitcoins; instead, rules are applied that allow bitcoin to be reassigned if the puzzles can be solved. So you can think of "owning" bitcoin as being equivalent to having the information required to solve the puzzle. In other words, the only benefit ownership confers is the ability to reassign ownership. This may seem odd at first, but it's the essence of how all money works.

2.5.4 Transactions

To spend coins, one creates a *transaction* (or *Tx*). Roughly speaking, a transaction unrolls rolled-up and locked coins from unspents, puts them in a big pile, then rerolls them and locks them in new unspents. The old unspents are now spent and discarded as historical relics.

A transaction consists of a bit of metadata, one or more inputs, each of which is known as a *TxIn* (commonly known as a vin), and one or more outputs, each of which is known as a *TxOut* (or vout).

Transactions are named by their transaction ID, which is a 256-bit binary string, generally expressed as a 64-character hex id. Example:

```
0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098
```

This binary string is generated by streaming the transaction to a binary blob in a canonical way, then taking the sha256 hash of the blob **twice**. Note that by convention, the hex id is written in reverse, little-endian order.

2.5.5 TxIn

A TxIn defines the input coins for a transaction. It points to unspent coins via transaction ID and the index of the TxOut. This TxOut must not be spent by any other confirmed transaction (this is called a "double-spend", and is akin to offering the same ten dollar bill to two different people).

Each TxIn refers to exactly one unspent, and includes a *solution script* that corresponds to the unspent's puzzle script, and "unlocks" it. If the unspent's puzzle script is a lock, the solution script is a key – but a key that (usually) unlocks the coins *only* for the transaction the TxIn is embedded in.

To "unlock" the coins, the puzzle script must be solved. This generally requires knowing the private key corresponding to the public key, and creating a signature proving that the private key is known and approves the resultant transfer of coins expressed by the transaction. This is done by hashing parts of the transaction, including the TxOut list, and having the solution include a digital signature on that hash.

2.5.6 TxOut

A TxOut defines the output coins of a transaction. It's a structure that includes an integer representing the number of satoshis dedicated to this output and the puzzle script that protects these coins.

Each TxOut of a confirmed transaction corresponds to an unspent, that can be referred to by a TxIn in a future transaction (but only one!).

2.5.7 Fees

The total coins reassigned in the list of TxOut objects must be no greater than the coins collected up in the list of TxIn objects (or else coins can be created out of thin air). However, the TxOut total can be less than the TxIn total, in which case, the excess coins are a "fee" that can be collected by the miner that confirms the transaction, encouraging its confirmation. In practice, all transactions have a fee, and it is generally proportional to the size of the transaction in bytes.

2.5.8 Puzzle Script

A puzzle script is a program written in a non-Turing-complete language known as Bitcoin script. Bitcoin script is a low-level interpreted stack-based script language with opcodes that operate on the stack, with a few special opcodes tuned to verifying ECDSA digital cryptographic signatures.

2.5.9 Block

Transactions are passed around the peer-to-peer Bitcoin network, and eventually reach validating nodes known as "miners". Miners attempt to bundle transactions into a *block* using a proof-of-work trick that makes finding blocks a time-consuming process. Each block includes a reference to the previous block, forming a linked list-style data structure, known as a *blockchain*.

Although blocks are hard to find, their correctness is easy to verify. Once a block is found, it's shared with peers on the network who can verify it's correctness, accept the block as the next "ledger page" in the ledger, and begin trying to extend the blockchain by creating a new block that refers to the previous.

2.5.10 Mining

Mining is conceptually akin to this process: create a lottery ticket (a block that includes a list of transactions), check to see if the ticket is a winner, and repeat until a winner is found.

It's quite simple to create a potential block candidate: assemble a list of validated transactions and increment a nonce until the block header starts with a sufficient number of zero bits.

Blocks contain a reward, consisting of a block reward that decreases over time (from 50 BTC, to 25 BTC, 12.5 BTC, etc., each step lasting about four years), plus all the fees from each transaction included in the block. All coins in existence can be traced back to the initial block in which they are generated as a reward for creating the block.

2.5.11 ECDSA Keys

A bitcoin ECDSA private key is an integer between 1 and 115792089237316195423570985008687907852837564279074904382605163 which is about 1.15e77 (inclusive). This number is called a "secret exponent".

Each private key has a corresponding public key of the form (x, y) where x and y are 256-bit integers. Note that once x is determined, the y value is also determined to be one of two values Y1 or Y2, one of which is odd, the other of which is even. So it's enough to specify the x value and the parity of y.

These public keys are streamed to a binary blob using the SEC standard, which defines a *compressed* (33-byte) and *uncompressed* (65-byte, legacy and generally no longer used) form. The 33-byte compressed form is 02 or 03 byte depending on y being even or odd, then the 32 bytes of x value. The 65-byte uncompressed form is a 04 byte, followed by the 32 bytes of x value, then the 32 bytes of y value.

2.5.12 Public Key Hash (PKH)

After public keys is formatted as a binary blob using the SEC standard, it is hashed twice: first to a 256 bit value using sha256, then that result is hashed to a 160 bit value using ripmd160. This value is called a hash160, or a public key hash (pkh).

2.5.13 Base58

Base 58 is frequently used by bitcoin to encode data that would otherwise be binary. It consists of the digits (10 characters), the upper case letter (26 characters), and the lower case letters (26 characters) EXCEPT 0 (zero), o (lower-case O), i (lower-case I) and l (lower case L). These characters were likely excluded due to their potential confusion with other similar-looking characters. Note that 10 + 26 + 26 - 4 = 58.

2.5.14 BIP32

BIP32 (where BIP stands for "Bitcoin improvement proposal") describes a way to create a hierarchical tree of private or public keys, where child keys can be derived by keys higher in the tree. For examples, please refer to the documentation for *ku*.

2.6 Contract

2.6.1 `__init__` Module

class pycoin.networks.Contract.**Contract**(*script_info*, *network*)

Bases: object

A script that encumbers coins.

address()

Return a string with the address for this script (or None if this script does have a corresponding address).

disassemble()

Return a text string of the disassembly of the script.

hash160()

Return a 20-byte hash corresponding to this script (or None if not applicable).

info()

ku_output()

Return a 20-byte hash corresponding to this script (or None if not applicable).

override_network(*override_network*)

script()

Return a bytes with a binary image of the script.

2.7 ECDSA

2.7.1 `__init__` Module

2.7.2 `Curve` Module

class pycoin.ecdsa.Curve.**Curve**(*p*, *a*, *b*, *order=None*)

This class implements an Elliptic curve intended for use in Elliptic curve cryptography

An elliptic curve `EC<p, a, b>` for a (usually large) prime p and integers a and b is a group. The members of the group are (x, y) points (where x and y are integers over the field of integers modulo p) that satisfy the relation `y**2 = x**3 + a*x + b (mod p)`. There is a group operation + and an extra point known as the "point at infinity" thrown in to act as the identity for the group.

The group operation is a truly marvelous property of this construct, a description of which this margin is too narrow to contain, so please refer to the links above for more information.

Parameters

- **p** – a prime
- **a** – an integer coefficient
- **b** – an integer constant
- **order** – (optional) the order of the group made up by the points on the curve. Any point on the curve times the order is the identity for this group (the point at infinity). Although this is optional, it's required for some operations.

Point(*x*, *y*)

Returns a *Point* object with coordinates `(x, y)`

add(*p0*, *p1*)

Parameters

- **p0** – a point
- **p1** – a point

Returns the sum of the two points

contains_point(*x*, *y*)

Parameters

- **x** – x coordinate of a point
- **y** – y coordinate of a point

Returns True if the point (x, y) is on the curve, False otherwise

infinity()

Returns the "point at infinity" (also known as 0, or the identity).

inverse_mod(*a*, *m*)

Parameters

- **a** – an integer
- **m** – another integer

Returns the value `b` such that `a * b == 1 (mod m)`

multiply(*p*, *e*)
multiply a point by an integer.

Parameters

- **p** – a point
- **e** – an integer

Returns the result, equivalent to adding p to itself e times

order()

Returns the order of the curve.

p()

Returns the prime modulus of the curve.

2.7.3 `Generator` Module

class pycoin.ecdsa.Generator.**Generator**(*p*, *a*, *b*, *basis*, *order*, *entropy_f=<built-in function urandom>*)

Bases: *pycoin.ecdsa.Curve.Curve*, *pycoin.ecdsa.Point.Point*

A Generator is a specific point on an elliptic curve that defines a trapdoor function from integers to curve points.

Parameters

- **p** – the prime for the *Curve*
- **a** – the a value for the *Curve*
- **b** – the b value for the *Curve*
- **basis** – a *Point* on the given *Curve*
- **order** – the order for the *Curve*

The constructor raises `NoSuchPointError` if the point is invalid. The point at infinity is `(x, y) == (None, None)`.

__mul__(*e*)
Multiply the generator by an integer. Uses the blinding factor.

inverse(*a*)

Returns n where `a * n == 1 (mod p) for the curve's prime p`.

modular_sqrt(*a*)

Returns n where `n * n == a (mod p) for the curve's prime p`. If no such n exists, an arbitrary value will be returned.

points_for_x(*x*)

Param x: an integer x coordinate

Returns (p0, p1) where each p is a `Point` with given x coordinate, and p0's y value is even.

To get a point with particular parity, use:: points_for_x(x)[1 if is_y_supposed_to_be_odd else 0]

possible_public_pairs_for_signature(*value*, *signature*, *y_parity=None*)

Param value: an integer value

Param signature: an `(r, s)` pair of integers representing an ecdsa signature of `value`

Param y_parity: (optional) for a given value and signature, there are either two points that sign it, or none if the signature is invalid. One of the points has an even y value, the other an odd. If this parameter is set, only points whose y value matches this value will be returned in the list.

Returns a list of *Point* objects p where each p is a possible public key for which `signature` correctly signs the given `value`. If something goes wrong, this list will be empty.

raw_mul(*e*)

Param e: an integer value

Returns e * self

This method uses a precomputed table as an optimization.

sign(*secret_exponent*, *val*, *gen_k=None*)

Param secret_exponent: a *Point* on the curve

Param val: an integer value

Param gen_k: a function generating __k values__

Returns a pair of integers `(r, s)` represents an ecdsa signature of `val` with public key `self * secret_exponent`.

If gen_k is set, it will be called with (n, secret_exponent, val), and an unguessable K value should be returned. Otherwise, the default K value, generated according to rfc6979 will be used.

sign_with_recid(*secret_exponent*, *val*, *gen_k=None*)

Param secret_exponent: a *Point* on the curve

Param val: an integer value

Param gen_k: a function generating __k values__

Returns a tuple of integers `(r, s, recid)` where `(r, s)` represents an ecdsa signature of `val` with public key `self * secret_exponent`; and `recid` is the **recovery id**, a number from 0-3 used to eliminate ambiguity about which public key signed the value.

If gen_k is set, it will be called with (n, secret_exponent, val), and an unguessable K value should be returned. Otherwise, the default K value, generated according to rfc6979 will be used.

verify(*public_pair*, *val*, *sig*)

Param public_pair: a *Point* on the curve

Param val: an integer value

Param sig: a pair of integers `(r, s)` representing an ecdsa signature

Returns True if and only if the signature `sig` is a valid signature of `val` using `public_pair` public key.

2.7.4 `Point` Module

exception pycoin.ecdsa.Point.**NoSuchPointError**

Bases: `exceptions.ValueError`

class pycoin.ecdsa.Point.**Point**(*x*, *y*, *curve*)
Bases: tuple

A point on an elliptic curve. This is a subclass of tuple (forced to a 2-tuple), and also includes a reference to the underlying Curve.

This class supports the operators +, − (unary and binary) and ⋆.

Parameters

- **x** – x coordinate
- **y** – y coordinate
- **curve** – the *Curve* this point must be on

The constructor raises *NoSuchPointError* if the point is invalid. The point at infinity is `(x, y) == (None, None)`.

__add__(*other*)
Add one point to another point.

__mul__(*e*)
Multiply a point by an integer.

__neg__()
Unary negation

__sub__(*other*)
Subtract one point from another point.

check_on_curve()
raise *NoSuchPointError* if the point is not actually on the curve.

curve()

Returns the *Curve* this point is on

2.7.5 encrypt Module

pycoin.ecdsa.encrypt.**generate_shared_public_key**(*my_private_key*, *their_public_pair*, *generator*)
Two parties each generate a private key and share their public key with the other party over an insecure channel. The shared public key can be generated by either side, but not by eavesdroppers. You can then use the entropy from the shared public key to created a common symmetric key for encryption. (This is beyond of the scope of pycoin.)

See also <https://en.wikipedia.org/wiki/Key_exchange>

Parameters

- **my_private_key** – an integer private key
- **their_public_pair** – a pair `(x, y)` representing a public key for the `generator`
- **generator** – a *Generator*

Returns a *Point*, which can be used as a shared public key.

2.7.6 intstream Module

pycoin.ecdsa.intstream.**from_bytes**(*bytes*, *byteorder='big'*, *signed=False*)
This is the same functionality as `int.from_bytes` in python 3

pycoin.ecdsa.intstream.**to_bytes**(*v*, *length*, *byteorder='big'*)
This is the same functionality as `int.to_bytes` in python 3

2.7.7 rfc6979 Module

pycoin.ecdsa.rfc6979.**bit_length**(*v*)
the `int.bit_length` in python 3

pycoin.ecdsa.rfc6979.**deterministic_generate_k**(*generator_order*, *secret_exponent*, *val*, *hash_f=<built-in function openssl_sha256>*)

Parameters

- **generator_order** – result from *pycoin.ecdsa.Generator.Generator.order*, necessary to ensure the k value is within bound
- **secret_exponent** – an integer secret_exponent to generate the k value for
- **val** – the value to be signed, also used as an entropy source for the k value

Returns an integer k such that `1 <= k < generator_order`, complying with <https://tools.ietf.org/html/rfc6979>

2.7.8 secp256k1 Module

class pycoin.ecdsa.secp256k1.**GeneratorWithOptimizations**(*p*, *a*, *b*, *basis*, *order*, *entropy_f=<built-in function urandom>*)

Bases: `pycoin.ecdsa.native.secp256k1.noop`, `pycoin.ecdsa.native.openssl.Optimizations`, *pycoin.ecdsa.Generator.Generator*

2.7.9 secp256r1 Module

class pycoin.ecdsa.secp256r1.**GeneratorWithOptimizations**(*p*, *a*, *b*, *basis*, *order*, *entropy_f=<built-in function urandom>*)

Bases: `pycoin.ecdsa.native.openssl.Optimizations`, *pycoin.ecdsa.Generator.Generator*

2.7.10 native.bignum Module

Arrange to access a shared-object version of the bignum library using Python ctypes.

pycoin.ecdsa.native.bignum.**bignum_type_for_library**(*library*)

2.7.11 native.openssl Module

class pycoin.ecdsa.native.openssl.**BignumContext**
Bases: _ctypes.Structure

pycoin.ecdsa.native.openssl.**create_OpenSSLOptimizations**(*curve_id*)

pycoin.ecdsa.native.openssl.**load_library**()

pycoin.ecdsa.native.openssl.**set_api**(*library*, *api_info*)

2.7.12 native.secp256k1 Module

pycoin.ecdsa.native.secp256k1.**LibSECP256K1Optimizations**
alias of pycoin.ecdsa.native.secp256k1.noop

class pycoin.ecdsa.native.secp256k1.**Optimizations**

multiply(*p*, *e*)
Multiply a point by an integer.

sign(*secret_exponent*, *val*, *gen_k=None*)

verify(*public_pair*, *val*, *signature_pair*)

pycoin.ecdsa.native.secp256k1.**create_LibSECP256K1Optimizations**()

pycoin.ecdsa.native.secp256k1.**load_library**()

2.8 Services

2.8.1 __init__ Module

pycoin.services.__init__.**spendables_for_address**(*address*, *netcode*, *format=None*)
Return a list of Spendable objects for the given bitcoin address.

Set format to "text" or "dict" to transform return value from an object to a string or dict.

This is intended to be a convenience function. There is no way to know that the list returned is a complete list of spendables for the address in question.

You can verify that they really do come from the existing transaction by calling tx_utils.validate_unspents.

pycoin.services.__init__.**get_tx_db**(*netcode=None*)

pycoin.services.__init__.**get_default_providers_for_netcode**(*netcode=None*)

pycoin.services.__init__.**set_default_providers_for_netcode**(*netcode*, *provider_list*)

2.8.2 agent Module

pycoin.services.agent.**urlopen**(*url*, *data=None*)

2.8.3 bitcoind Module

class pycoin.services.bitcoind.**BitcoindProvider**(*bitcoind_url*)
Bases: object

bitcoind_agrees_on_transaction_validity(*tx*)

tx_for_tx_hash(*tx_hash*)

pycoin.services.bitcoind.**bitcoind_agrees_on_transaction_validity**(*bitcoind_url*, *tx*)

pycoin.services.bitcoind.**unspent_to_bitcoind_dict**(*tx_in*, *tx_out*)

2.8.4 blockchain_info Module

class pycoin.services.blockchain_info.**BlockchainInfoProvider**(*netcode*)
Bases: object

broadcast_tx(*tx*)

payments_for_address(*address*)
return an array of (TX ids, net_payment)

spendables_for_address(*address*)
Return a list of Spendable objects for the given bitcoin address.

tx_for_tx_hash(*tx_hash*)
Get a Tx by its hash.

pycoin.services.blockchain_info.**send_tx**(*self*, *tx*)

2.8.5 blockcypher Module

class pycoin.services.blockcypher.**BlockcypherProvider**(*api_key=''*, *netcode=None*)
Bases: object

base_url(*args*)

broadcast_tx(*tx*)
broadcast a transaction to the network

get_balance(*address*)
returns the balance object from blockcypher for address

spendables_for_address(*address*)
Return a list of Spendable objects for the given bitcoin address.

tx_for_tx_hash(*tx_hash*)
returns the pycoin.tx object for tx_hash

2.8.6 chain_so Module

class pycoin.services.chain_so.**ChainSoProvider**(*netcode=None*)
Bases: object

base_url(*method*, *args*)

spendables_for_address(*address*)
Return a list of Spendable objects for the given bitcoin address.

tx_for_tx_hash(*tx_hash*)
Get a Tx by its hash.

2.8.7 env Module

pycoin.services.env.**config_string_for_netcode_from_env**(*netcode*)

pycoin.services.env.**main_cache_dir**()

pycoin.services.env.**tx_read_cache_dirs**()

pycoin.services.env.**tx_writable_cache_dir**()

2.8.8 insight Module

class pycoin.services.insight.**InsightProvider**(*base_url='https://insight.bitpay.com'*, *netcode=None*)

Bases: object

get_block_height(*block_hash*)

get_blockchain_tip()

get_blockheader(*block_hash*)

get_blockheader_with_transaction_hashes(*block_hash*)

get_tx_confirmation_block(*tx_hash*)

send_tx(*tx*)

spendables_for_address(*address*)
Return a list of Spendable objects for the given bitcoin address.

spendables_for_addresses(*addresses*)

tx_for_tx_hash(*tx_hash*)

pycoin.services.insight.**tx_from_json_dict**(*r*)

2.8.9 providers Module

pycoin.services.providers.**all_providers_message**(*method*, *netcode*)

pycoin.services.providers.**bitcoin_rpc_init**(*match*, *netcode*)

pycoin.services.providers.**get_default_providers_for_netcode**(*netcode=None*)

pycoin.services.providers.**get_tx_db**(*netcode=None*)

pycoin.services.providers.**insight_init**(*match*, *netcode*)

pycoin.services.providers.**message_about_spendables_for_address_env**(*netcode*)

pycoin.services.providers.**message_about_tx_cache_env**()

pycoin.services.providers.**message_about_tx_for_tx_hash_env**(*netcode*)

pycoin.services.providers.**provider_for_descriptor_and_netcode**(*descriptor*, *netcode=None*)

pycoin.services.providers.**providers_for_config_string**(*config_string*, *netcode*)

pycoin.services.providers.**providers_for_netcode_from_env**(*netcode*)

pycoin.services.providers.**service_provider_methods**(*method_name*, *service_providers*)

pycoin.services.providers.**set_default_providers_for_netcode**(*netcode*, *provider_list*)

pycoin.services.providers.**spendables_for_address**(*address*, *netcode*, *format=None*)

Return a list of Spendable objects for the given bitcoin address.

Set format to "text" or "dict" to transform return value from an object to a string or dict.

This is intended to be a convenience function. There is no way to know that the list returned is a complete list of spendables for the address in question.

You can verify that they really do come from the existing transaction by calling tx_utils.validate_unspents.

2.8.10 tx_db Module

class pycoin.services.tx_db.**TxDb**(*lookup_methods=[]*, *read_only_paths=[]*, *writable_cache_path=None*)

Bases: object

This object can be used in many places that expect a dict.

get(*key*)

paths_for_hash(*hash*)

put(*tx*)

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

H

I

K

L

M

N

O

P

R

S

T

U

V

W