Bitcoin Utilities Documentation

Release 0.5.8

Konstantinos Karasavvas

Oct 31, 2022

CONTENTS

Contents:

KEYS AND ADDRESSES MODULE

class keys.**Address**(*address=None*, *hash160=None*, *script=None*)

    Represents a Bitcoin address

    hash160

        the hash160 string representation of the address; hash160 represents two consequtive hashes of the public key or the redeam script, first a SHA-256 and then an RIPEMD-160

        Type

            str

    from_address(*address*)

        instantiates an object from address string encoding

    from_hash160(*hash160_str*)

        instantiates an object from a hash160 hex string

    from_script(*redeem_script*)

        instantiates an object from a redeem_script

    to_string()

        returns the address's string encoding

    to_hash160()

        returns the address's hash160 hex string representation

        Raises

            • **TypeError** – No parameters passed

            • **ValueError** – If an invalid address or hash160 is provided.

    classmethod from_address(*address*)

        Creates an address object from an address string

    classmethod from_hash160(*hash160*)

        Creates an address object from a hash160 string

    classmethod from_script(*script*)

        Creates an address object from a Script object

    to_hash160()

        Returns as hash160 hex string

to_string()

> Returns as address string

> Pseudocode:
>> network_prefix = (1 byte version number)
>> data = network_prefix + hash160_bytes
>> data_hash = SHA-256(SHA-256(hash160_bytes))
>> checksum = (first 4 bytes of data_hash)
>> address_bytes = Base58CheckEncode(data + checksum)

class keys.**P2pkhAddress**(*address=None*, *hash160=None*)

> Encapsulates a P2PKH address.

> Check Address class for details

> **to_script_pub_key()**

>> returns the scriptPubKey (P2PKH) that corresponds to this address

> **get_type()**

>> returns the type of address

> **get_type()**

>> Returns the type of address

> **to_script_pub_key()**

>> Returns the scriptPubKey (P2PKH) that corresponds to this address

class keys.**P2shAddress**(*address=None*, *hash160=None*, *script=None*)

> Encapsulates a P2SH address.

> Check Address class for details

> **get_type()**

>> returns the type of address

> **get_type()**

>> Returns the type of address

> **to_script_pub_key()**

>> Returns the scriptPubKey (P2SH) that corresponds to this address

class keys.**P2wpkhAddress**(*address=None*, *witness_hash=None*, *version='p2wpkhv0'*)

> Encapsulates a P2WPKH address.

> Check Address class for details

> **to_script_pub_key()**

>> returns the scriptPubKey of a P2WPKH witness script

> **get_type()**

>> returns the type of address

> **get_type()**

>> Returns the type of address

`to_script_pub_key`()

Returns the scriptPubKey of a P2WPKH witness script

class keys.**P2wshAddress**(*address=None*, *witness_hash=None*, *script=None*, *version='p2wshv0'*)

Encapsulates a P2WSH address.

Check Address class for details

`from_script`(*witness_script*)

instantiates an object from a witness_script

`get_type`()

returns the type of address

`get_type`()

Returns the type of address

`to_script_pub_key`()

Returns the scriptPubKey of a P2WPKH witness script

class keys.**PrivateKey**(*wif=None*, *secret_exponent=None*)

Represents an ECDSA private key.

`key`

the raw key of 32 bytes

> **Type**
> bytes

`from_wif`(*wif*)

creates an object from a WIF of WIFC format (string)

`to_wif`(*compressed=True*)

returns as WIFC (compressed) or WIF format (string)

`to_bytes`()

returns the key's raw bytes

`sign_message`(*message*, *compressed=True*)

signs the message's digest and returns the signature

`sign_input`(*tx*, *txin_index*, *script*, *sighash=SIGHASH_ALL*)

signs the transaction's digest for a particular index and returns the signature.

`sign_segwit_input`(*tx*, *txin_index*, *script*, *amount*, *sighash=SIGHASH_ALL*)

signs the transaction's digest for a particular index and amount and returns the signature.

`get_public_key`()

returns the corresponding PublicKey object

classmethod `from_wif`(*wif*)

Creates key from WIFC or WIF format key

`get_public_key`()

Returns the corresponding PublicKey

sign_message(*message*, *compressed=True*)

    Signs the message with the private key (deterministically)

    Bitcoin uses a compact format for message signatures (for tx sigs it uses normal DER format). The format has the normal r and s parameters that ECDSA signatures have but also includes a prefix which encodes extra information. Using the prefix the public key can be reconstructed when verifying the signature.

    Prefix values:

        27 - 0x1B = first key with even y
        28 - 0x1C = first key with odd y
        29 - 0x1D = second key with even y
        30 - 0x1E = second key with odd y

    If key is compressed add 4 (31 - 0x1F, 32 - 0x20, 33 - 0x21, 34 - 0x22 respectively)

    Returns a Bitcoin compact signature in Base64

to_bytes()

    Returns key's bytes

to_wif(*compressed=True*)

    Returns key in WIFC or WIF string

    Pseudocode:

        network_prefix = (1 byte version number)
        data = network_prefix + (32 bytes number/key) [+ 0x01 if compressed]
        data_hash = SHA-256(SHA-256(data))
        checksum = (first 4 bytes of data_hash)
        wif = Base58CheckEncode(data + checksum)

class keys.**PublicKey**(*hex_str*)

    Represents an ECDSA public key.

key

    the raw public key of 64 bytes (x, y coordinates of the ECDSA curve)

        Type
            bytes

from_hex(*hex_str*)

    creates an object from a hex string in SEC format

from_message_signature(*signature*)

    NO-OP!

verify_message(*address*, *signature*, *message*)

    Class method that constructs the public key, confirms the address and verifies the signature

verify(*signature*, *message*)

    returns true if the message was signed with this public key's corresponding private key.

to_hex(*compressed=True*)

> returns the key as hex string (in SEC format - compressed by default)

to_bytes()

> returns the key's raw bytes

to_hash160()

> returns the hash160 hex string of the public key

get_address(*compressed=True*))

> returns the corresponding P2pkhAddress object

get_segwit_address()

> returns the corresponding P2wpkhAddress object

classmethod from_hex(*hex_str*)

> Creates a public key from a hex string (SEC format)

get_address(*compressed=True*)

> Returns the corresponding P2PKH Address (default compressed)

get_segwit_address()

> Returns the corresponding P2WPKH address
>
> Only compressed is allowed. It is otherwise identical to normal P2PKH address.

to_bytes()

> Returns key's bytes

to_hash160(*compressed=True*)

> Returns the RIPEMD(SHA256( )) of the public key in hex

to_hex(*compressed=True*)

> Returns public key as a hex string (SEC format - compressed by default)

verify(*signature*, *message*)

> Verifies that the message was signed with this public key's corresponding private key.

classmethod verify_message(*address*, *signature*, *message*)

> Creates a public key from a message signature and verifies message
>
> Bitcoin uses a compact format for message signatures (for tx sigs it uses normal DER format). The format has the normal r and s parameters that ECDSA signatures have but also includes a prefix which encodes extra information. Using the prefix the public key can be reconstructed from the signature.
>
> Prefix values:
>> 27 - 0x1B = first key with even y
>> 28 - 0x1C = first key with odd y
>> 29 - 0x1D = second key with even y
>> 30 - 0x1E = second key with odd y
>
> If key is compressed add 4 (31 - 0x1F, 32 - 0x20, 33 - 0x21, 34 - 0x22 respectively)
>
>> **Raises**
>>> **ValueError** – If signature is invalid

class keys.**SegwitAddress**(*address=None*, *witness_hash=None*, *script=None*, *version='p2wpkhv0'*)

    Represents a Bitcoin segwit address

    Note that currently the python bech32 reference implementation is used (by Pieter Wuille).

    witness_hash

        the hash string representation of either the address; it can be either a public key hash (P2WPKH) or the hash of the script (P2WSH)

            Type

                str

    from_address(*address*)

        instantiates an object from address string encoding

    from_hash(*hash_str*)

        instantiates an object from a hash hex string

    from_script(*witness_script*)

        instantiates an object from a witness_script

    to_string()

        returns the address's string encoding (Bech32)

    to_hash()

        returns the address's hash hex string representation

        Raises

            • **TypeError** – No parameters passed

            • **ValueError** – If an invalid address or hash is provided.

    classmethod from_address(*address*)

        Creates an address object from an address string

    classmethod from_hash(*witness_hash*)

        Creates an address object from a hash string

    classmethod from_script(*script*)

        Creates an address object from a Script object

    to_hash()

        Returns as hash hex string

    to_string()

        Returns as address string

        Uses a segwit's python reference implementation for now. (TODO)

TRANSACTIONS MODULE

class transactions.**Locktime**(*value*)

Helps setting up appropriate locktime.

value

The value of the block height or the Unix epoch (seconds from 1 Jan 1970 UTC)

Type
int

for_transaction()

Serializes the locktime as required in a transaction

Raises
ValueError – if the value is not within range of 2 bytes.

for_transaction()

Creates a timelock as expected from Transaction

class transactions.**Sequence**(*seq_type*, *value=None*, *is_type_block=True*)

Helps setting up appropriate sequence. Used to provide the sequence to transaction inputs and to scripts.

value

The value of the block height or the 512 seconds increments

Type
int

seq_type

Specifies the type of sequence (TYPE_RELATIVE_TIMELOCK | TYPE_ABSOLUTE_TIMELOCK | TYPE_REPLACE_BY_FEE

Type
int

is_type_block

If type is TYPE_RELATIVE_TIMELOCK then this specifies its type (block height or 512 secs increments)

Type
bool

for_input_sequence()

Serializes the relative sequence as required in a transaction

for_script()

Returns the appropriate integer for a script; e.g. for relative timelocks

> **Raises**
> **ValueError** – if the value is not within range of 2 bytes.

for_input_sequence()

Creates a relative timelock sequence value as expected from TxInput sequence attribute

for_script()

Creates a relative/absolute timelock sequence value as expected in scripts

class transactions.**Transaction**(*inputs=None*, *outputs=None*, *locktime=b'\x00\x00\x00\x00'*,
  version=b'\x02\x00\x00\x00', *has_segwit=False*, *witnesses=None*)

Represents a Bitcoin transaction

inputs

A list of all the transaction inputs

> **Type**
> list (*TxInput*)

outputs

A list of all the transaction outputs

> **Type**
> list (*TxOutput*)

locktime

The transaction's locktime parameter

> **Type**
> bytes

version

The transaction version

> **Type**
> bytes

has_segwit

Specifies a tx that includes segwit inputs

> **Type**
> bool

witnesses

The witness scripts that correspond to the inputs

> **Type**
> list (*Script*)

stream()

Converts Transaction to bytes

serialize()

Converts Transaction to hex string

get_txid()

> Calculates txid and returns it

get_hash()

> Calculates tx hash (wtxid) and returns it

get_wtxid()

> Calculates tx hash (wtxid) and returns it

get_size()

> Calculates the tx size

get_vsize()

> Calculates the tx segwit size

copy()

> creates a copy of the object (classmethod)

get_transaction_digest(*txin_index*, *script*, *sighash*)

> returns the transaction input's digest that is to be signed according

get_transaction_segwit_digest(*txin_index*, *script*, *amount*, *sighash*)

> returns the transaction input's segwit digest that is to be signed according to sighash

classmethod copy(*tx*)

> Deep copy of Transaction

static from_raw(*txraw*)

> Imports a Transaction from hexadecimal data

> > **txinputraw**
> >
> > > The hexadecimal raw string of the Transaction
> > > > **Type**
> > > > > string (hex)

> > **cursor**
> >
> > > The cursor of which the algorithm will start to read the data
> > > > **Type**
> > > > > int

> > **has_segwit**
> >
> > > Is the Tx Input segwit or not
> > > > **Type**
> > > > > boolean

get_hash()

> Hashes the serialized (bytes) tx including segwit marker and witnesses

get_size()

> Gets the size of the transaction

get_transaction_digest(*txin_index*, *script*, *sighash=1*)

> Returns the transaction's digest for signing.

> SIGHASH types (see constants.py):
> > SIGHASH_ALL - signs all inputs and outputs (default)

SIGHASH_NONE - signs all of the inputs

SIGHASH_SINGLE - signs all inputs but only txin_index output

SIGHASH_ANYONECANPAY (only combined with one of the above)

- with ALL - signs all outputs but only txin_index input

- with NONE - signs only the txin_index input

- with SINGLE - signs txin_index input and output

txin_index

The index of the input that we wish to sign

Type

int

script

The scriptPubKey of the UTXO that we want to spend

Type

list (string)

sighash

The type of the signature hash to be created

Type

int

get_transaction_segwit_digest(*txin_index*, *script*, *amount*, *sighash=1*)

Returns the segwit transaction's digest for signing.

SIGHASH types (see constants.py):

SIGHASH_ALL - signs all inputs and outputs (default)

SIGHASH_NONE - signs all of the inputs

SIGHASH_SINGLE - signs all inputs but only txin_index output

SIGHASH_ANYONECANPAY (only combined with one of the above)

- with ALL - signs all outputs but only txin_index input

- with NONE - signs only the txin_index input

- with SINGLE - signs txin_index input and output

txin_index

The index of the input that we wish to sign

Type

int

script

The scriptPubKey of the UTXO that we want to spend

Type

list (string)

amount

The amount of the UTXO to spend is included in the signature for segwit (in satoshis)

Type

int/float/Decimal

sighash

The type of the signature hash to be created

Type

int

get_txid()

Hashes the serialized (bytes) tx to get a unique id

get_vsize()

Gets the virtual size of the transaction.

For non-segwit txs this is identical to get_size(). For segwit txs the marker and witnesses length needs to be reduced to 1/4 of its original length. Thus it is substructed from size and then it is divided by 4 before added back to size to produce vsize (always rounded up).

https://en.bitcoin.it/wiki/Weight_units

get_wtxid()

Hashes the serialized (bytes) tx including segwit marker and witnesses

serialize()

Converts to hex string

stream(*has_segwit*)

Converts to bytes

class transactions.**TxInput**(*txid*, *txout_index*, *script_sig=[]*, *sequence=b'\xff\xff\xff\xff'*)

Represents a transaction input.

A transaction input requires a transaction id of a UTXO and the index of that UTXO.

txid

the transaction id as a hex string (little-endian as displayed by tools)

Type

str

txout_index

the index of the UTXO that we want to spend

Type

int

script_sig

the op code and data of the script as string

Type

list (strings)

sequence

the input sequence (for timelocks, RBF, etc.)

Type

bytes

stream()

converts TxInput to bytes

copy()

creates a copy of the object (classmethod)

classmethod copy(*txin*)

    Deep copy of TxInput

static from_raw(*txinputraw*, *cursor=0*, *has_segwit=False*)

    Imports a TxInput from a Transaction's hexadecimal data

    txinputraw

        The hexadecimal raw string of the Transaction

            Type

                string (hex)

    cursor

        The cursor of which the algorithm will start to read the data

            Type

                int

    has_segwit

        Is the Tx Input segwit or not

            Type

                boolean

stream()

    Converts to bytes

class transactions.TxOutput(*amount*, *script_pubkey*)

    Represents a transaction output

    amount

        the value we want to send to this output in satoshis

            Type

                int/float/Decimal

    script_pubkey

        the script that will lock this amount

            Type

                list (string)

    stream()

        converts TxInput to bytes

    copy()

        creates a copy of the object (classmethod)

    classmethod copy(*txout*)

        Deep copy of TxOutput

    static from_raw(*txoutputraw*, *cursor=0*, *has_segwit=False*)

        Imports a TxOutput from a Transaction's hexadecimal data

    txinputraw

        The hexadecimal raw string of the Transaction

            Type

                string (hex)

cursor

The cursor of which the algorithm will start to read the data

> **Type**
>
> int

has_segwit

Is the Tx Output segwit or not

> **Type**
>
> boolean

stream()

Converts to bytes

SCRIPT MODULE

class script.**Script**(*script*)

> Represents any script in Bitcoin
>
> A Script contains just a list of OP_CODES and also knows how to serialize into bytes
>
> **script**
>
> > the list with all the script OP_CODES and data
> >
> > > **Type**
> > > > list
>
> **to_bytes**()
>
> > returns a serialized byte version of the script
>
> **get_script**()
>
> > returns the list of strings that makes up this script
> >
> > > **Raises**
> > > > **ValueError** – If string data is too large or integer is negative
>
> **classmethod copy**(*script*)
>
> > Deep copy of Script
>
> **static from_raw**(*scriptraw*, *has_segwit=False*)
>
> > **Imports a Script commands list from raw hexadecimal data**
> >
> > > **txinputraw**
> > > > [string (hex)] The hexadecimal raw string representing the Script commands
> > >
> > > **has_segwit**
> > > > [boolean] Is the Tx Input segwit or not
>
> **get_script**()
>
> > Returns script as array of strings
>
> **to_bytes**(*segwit=False*)
>
> > Converts the script to bytes
> >
> > If an OP code the appropriate byte is included according to: https://en.bitcoin.it/wiki/Script If not consider it data (signature, public key, public key hash, etc.) and and include with appropriate OP_PUSHDATA OP code plus length
>
> **to_hex**()
>
> > Converts the script to hexadecimal

`to_p2sh_script_pub_key()`

> Converts script to p2sh scriptPubKey (locking script)
>
> Calculates the hash160 (via the address) of the script and uses it to construct a P2SH script.

`to_p2wsh_script_pub_key()`

> Converts script to p2wsh scriptPubKey (locking script)
>
> Calculates the sha256 of the script and uses it to construct a P2WSH script.

PROXY MODULE

class proxy.**NodeProxy**(*rpcuser=None*, *rpcpassword=None*, *host=None*, *port=None*)

    Simple Bitcoin node proxy that can call all of Bitcoin's JSON-RPC functionality.

    proxy

        a bitcoinrpc AuthServiceProxy object

            Type

                object

    get_proxy()

        Returns bitcoinrpc AuthServiceProxy object

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k
keys, 3

p
proxy, 19

s
script, 17

t
transactions, 9

INDEX