protocoin Documentation

Release 0.2

Christian S. Perone

January 23, 2016

Contents

Welcome to the Protocoin documentation. Protocoin is pure Python implementation of the Bitcoin protocol parsing and networking. Protocoin doesn't implement the protocol rules, only the serialization/deserialization procedures and also basic clients to interact with the Bitcoin P2P network.

Note: Protocoin is intended to be used to build clients that will collect statistics of the network, but you can also use it to implement a full Bitcoin client.

CHAPTER 1

Useful links

Some useful links:

- Github Project
- Bitcoin Protocol Specification

CHAPTER 2

Contents

2.1 Why another Python client ?

There are many other projects implementing the Bitcoin protocol, but none of them has a good documentation and the majority of the projects are very confusing to understand and to reuse/extend in third-party applications.

The aim of Protocoin is to implement a pythonic and well documented framework that can be used/extended with little or no effort.

2.2 Changelog

In this section you'll find information about what's new in the newer releases of the project.

2.2.1 Release v.0.2

- Fixed a bug when the nodes were disconnecting;
- Implemented the GetAddr message command;
- Added the key/address management module;
- Added the utility module;
- Added a new example: creation of a brain wallet;
- Rewrite of the socket recv loop;
- Fixed a bug in the socket closing;
- Improved the send_message() of the BitcoinBasicClient;
- Fixed protocol support for TxIn sequence (thanks to @bmuller);
- Refactoring to add SerializableMessage class (thanks to @bmuller);
- Fixed a bug in BitcoinPublicKey class (thanks to @bmuller);

2.2.2 Release v.0.1

This is the first release of the project. Some messages of the protocol are still missing and will be implemented in the next versions, the list of features implemented in this release are:

- Documentation
- **Field Types**
  - Base classes
  - Int32LEField
  - UInt32LEField
  - Int64LEField
  - UInt64LEField
  - Int16LEField
  - UInt16LEField
  - UInt16BEField
  - FixedStringField
  - NestedField
  - ListField
  - IPv4AddressField
  - VariableIntegerField
  - VariableStringField
  - Hash
- **Serializers**
  - Base classes, metaclasses
  - MessageHeaderSerializer
  - IPv4AddressSerializer
  - IPv4AddressTimestampSerializer
  - VersionSerializer
  - VerAckSerializer
  - PingSerializer
  - PongSerializer
  - InventorySerializer
  - InventoryVectorSerializer
  - AddressVectorSerializer
  - GetDataSerializer
  - NotFoundSerializer
  - OutPointSerializer
  - TxInSerializer
  - TxOutSerializer
  - TxSerializer
  - BlockHeaderSerializer

  - BlockSerializer
  - HeaderVectorSerializer
  - MemPoolSerializer
- **Clients**
  - BitcoinBasicClient
  - BitcoinClient

2.3 Getting Started

In this section you'll find a tutorial to learn more about Protocoin.

2.3.1 Installation

To install Protocoin, use *pip* (recommended method) or *easy_install*:

```
pip install protocoin
```

2.3.2 Architecture

Protocoin uses a simple architecture of classes representing the data to be serialized and also classes representing the types of the fields to be serialized.

Protocoin is organized in four main submodules:

- *protocoin.fields*
- *protocoin.serializers*
- *protocoin.clients*
- *protocoin.keys*

Each module structure is described in the next sections.

Protocoin Fields

The *protocoin.fields* module contains all field types suported by the serializers. All field classes inherit from the base *protocoin.fields.Field* class, so if you want to create a new field type, you should inherit from this class too. There are some composite field types to help in common uses like the *protocoin.fields.VariableStringField* for instance, representing a string with variable length.

There are a lot of different fields you can use to extend the protocol, examples are: *protocoin.fields.Int32LEField* (a 32-bit integer little-endian), *protocoin.fields.UInt32LEField* (a 32-bit unsigned int little-endian), *protocoin.fields.Int64LEField* (a 64-bit integer little-endian), *protocoin.fields.UInt64LEField* (a 64-bit unsigned integer little-endiang), etc. For more information about the fields avaiable please see the module documentation.

Example of code for the unsigned 32-bit integer field:

```
class UInt32LEField(Field):
    datatype = "<I"

    def parse(self, value):
        self.value = value

    def deserialize(self, stream):
        data_size = struct.calcsize(self.datatype)
        data = stream.read(data_size)
        return struct.unpack(self.datatype, data)[0]

    def serialize(self):
        data = struct.pack(self.datatype, self.value)
        return data
```

Protocoin Serializers

Serializers are classes that describe the field types (in the correct order) that will be used to serializer or deserialize the message or a part of a message, for instance, see this example of a *protocoin.serializers.IPv4Address* object class and then its serializer class implementation:

```
class IPv4Address(object):
    def __init__(self):
        self.services = fields.SERVICES["NODE_NETWORK"]
        self.ip_address = "0.0.0.0"
        self.port = 8333

class IPv4AddressSerializer(Serializer):
    model_class = IPv4Address
    services = fields.UInt64LEField()
    ip_address = fields.IPv4AddressField()
    port = fields.UInt16BEField()
```

To serialize a message, you simple do:

```
address = IPv4Address()
serializer = IPv4AddressSerializer()
binary_data = serializer.serialize(address)
```

and to deserialize:

```
address = serializer.deserialize(binary_data)
```

Warning: It is important to subclass the *protocoin.serializers.Serializer* class in order for the serializer to work, Serializers uses Python metaclasses magic to deserialize the fields using the correct types and also the correct order.

Note that we have a special attribute on the serializer that is defining the *model_class* for the serializer, this class is used to instantiate the correct object class in the deserialization of the data.

There are some useful fields you can use to nest another serializer or a list of serializers inside a serializer, see in this example of the implementation of the Version (*protocoin.serializers.Version*) command:

```
class VersionSerializer(Serializer):
    model_class = Version
    version = fields.Int32LEField()
    services = fields.UInt64LEField()
```

```
    timestamp = fields.Int64LEField()
    addr_recv = fields.NestedField(IPv4AddressSerializer)
    addr_from = fields.NestedField(IPv4AddressSerializer)
    nonce = fields.UInt64LEField()
    user_agent = fields.VariableStringField()
```

Note that the fields *addr_recv* and *addr_from* are using the special field called *protocoin.fields.NestedField*.

Note: There are other special fields like the *protocoin.fields.ListField*, that will create a vector of objects using the correct Bitcoin format to serialize vectors of data.

Network Clients

Protocoin also have useful classes to implement a network client for the Bitcoin P2P network.

A basic network client

The most basic class available to implement a client is the *protocoin.clients.BitcoinBasicClient*, which is a simple client of the Bitcoin network that accepts a socket in the constructor and then will handle and route the messages received to the correct methods of the class, see this example of a basic client:

```
import socket
from protocoin.clients import BitcoinBasicClient

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be", 8333))
client = BitcoinBasicClient(sock)
client.loop()
```

Note that this client is very basic, in the example above, the client will connect into the node **bitcoin.sipa.be** (a seed node) in the port 8333 and then will wait for messages. The *protocoin.clients.BitcoinBasicClient* class doesn't implement the handshake of the protocol and also doesn't answer the pings of the nodes, so you may be disconnected from the node and it is your reponsability to implement the handshake and the Pong response message to the Ping message. To implement answer according to the received messages from the network node, you can implement methods with the name **handle_[name of the command]**, to implement the handling method to show a message every time that a Version message arrives, you can do like in the example below:

```
class MyBitcoinClient(BitcoinBasicClient):
    def handle_version(self, message_header, message):
        print "A version was received !"
```

If you want to answer the version command message with a VerAck message, you just need to create the message, the serializer and then call the *protocoin.clients.BitcoinBasicClient.send_message()* method of the Bitcoin class, like in the example below:

```
class MyBitcoinClient(BitcoinBasicClient):
    def handle_version(self, message_header, message):
        verack = VerAck()
        verack_serial = VerAckSerializer()
        self.send_message(verack, verack_serial)
```

Since these problems are very common, there are another class which implements a node that will stay up in the Bitcoin network. To use this class, just subclass the *protocoin.clients.BitcoinClient* class, for more information read the next section.

A more complete client implementation

The *protocoin.clients.BitcoinClient* class implements the minimum required protocol rules to a client stay online on the Bitcoin network. This class will answer to Ping message commands with Pong messages and also have a handshake method that will send the Version command and answer the Version with the VerAck command message too. See an example of the use:

```
import socket
from protocoin.clients import BitcoinClient

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be", 8333))
client = BitcoinClient(sock)
client.handshake()
client.loop()
```

In the example above, the handshake is done before entering the message loop.

Bitcoin Keys – Creating, exporting/importing and conversions

The *protocoin.keys* module contains classes to represent and handle Bitcoin Private Keys as well Public Keys. The two main classes in this module are *protocoin.keys.BitcoinPublicKey* and *protocoin.keys.BitcoinPrivateKey*. These classes contain methods to generate new key pairs (Private and Public), to convert the keys into Bitcoin addresses or Bitcoin WIF (Wallet Import Format) and to import keys from different formats.

Creating Private Keys and Public Keys

In order to create a new Private Key, you only need to instantiate the *protocoin.keys.BitcoinPrivateKey* class without any parameter:

```
from protocoin import keys
priv_key = keys.BitcoinPrivateKey()
print priv_key
```

The example above, will create a new Private Key called *priv_key* and will output the string representation of the Private Key in hex:

```
<BitcoinPrivateKey hexkey=[E005459416BE7FDC13FA24BA2F2C0DE289F47495D6E94CF2DFBC9FB941CBB565]>
```

You can now use this generated Private Key to create your Public Key like in the example below:

```
from protocoin import keys
priv_key = keys.BitcoinPrivateKey()
pub_key = priv_key.generate_public_key()
print pub_key
```

This example will output:

```
<BitcoinPublicKey address=[19eQMjBSeeo8fhCRPEVCfnauhsCFVGgV6H]>
```

Which is the Bitcoin address for the Public Key. You can also convert the Public Key to hext format using the method *protocoin.keys.BitcoinPublicKey.to_hex()*.

Importing and Exporting Keys

You can also export a Private Key into the WIF (Wallet Import Format, used by wallets to import Private Keys):

```
from protocoin import keys
priv_key = keys.BitcoinPrivateKey()
print priv_key.to_wif()
```

In this case, the output will be:

```
5KWwtPkCodUs9WfbrSjzjLqnfbohABUAuLs3NpdxLqi4U6MjuKC
```

Which is the Private Key in the WIF format. You can also create a new Private Key or a new Public Key using the hex representation in the construction:

```
from protocoin import keys
hex_key = "E005459416BE7FDC13FA24BA2F2C0DE289F47495D6E94CF2DFBC9FB941CBB565"
priv_key = keys.BitcoinPrivateKey(hex_key)
```

If you have only the WIF format and you need to use it to create a new Private Key, you can use the *protocoin.keys.BitcoinPrivateKey.from_wif()* method to import it and then create a new Private Key object like in the example below:

```
priv_key_wif = "5KWwtPkCodUs9WfbrSjzjLqnfbohABUAuLs3NpdxLqi4U6MjuKC"
priv_key = BitcoinPrivateKey.from_wif(priv_key_wif)
```

2.4 Examples

In this section you can see various examples using Protocoin API.

2.4.1 Receiving blocks in real-time

In this example we will print the block information as well the block hash when blocks arrive in real-time from the nodes. It will also print the name of each message received:

```
import socket
from protocoin.clients import *

class MyBitcoinClient(BitcoinClient):
    def handle_block(self, message_header, message):
        print message
        print "Block hash:", message.calculate_hash()

    def handle_inv(self, message_header, message):
        getdata = GetData()
        getdata.inventory = message.inventory
        self.send_message(getdata)

    def handle_message_header(self, message_header, payload):
        print "Received message:", message_header.command
```

```
    def send_message(self, message):
        BitcoinClient.send_message(self, message)
        print "Message sent:", message.command

def run_main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(("bitcoin.sipa.be", 8333))
    client = MyBitcoinClient(sock)
    client.handshake()
    client.loop()

if __name__ == "__main__":
    run_main()
```

The example above will output:

```
<Block Version=[2] Timestamp=[Fri Nov 22 13:58:59 2013] Nonce=[1719395575] Hash=[0000000000000004b798
Block hash: 000000000000004b798ea6eb896bb3d39f1f1b19d285a0d48167e8661387e58
```

Note that in the example above, the **handle_inv** was implemented in order to retrieve the inventory data using the GetData message command. Without the GetData command, we only receive the Inv message command.

2.4.2 Inspecting transactions output

In the example below we're showing the output value in BTCs for each transaction output:

```
import socket
from protocoin.clients import *

class MyBitcoinClient(BitcoinClient):
    def handle_tx(self, message_header, message):
        print message
        for tx_out in message.tx_out:
            print "BTC: %.8f" % tx_out.get_btc_value()

    def handle_inv(self, message_header, message):
        getdata = GetData()
        getdata.inventory = message.inventory
        self.send_message(getdata)

def run_main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(("bitcoin.sipa.be", 8333))
    print "Connected !"
    client = MyBitcoinClient(sock)
    client.handshake()
    client.loop()

if __name__ == "__main__":
    run_main()
```

The example above will show the following output for every transaction, in this example it is showing a transaction with 13 inputs and 2 outputs of 0.25 BTC and 0.00936411 BTC:

```
<Tx Version=[1] Lock Time=[Always Locked] TxIn Count=[13] TxOut Count=[2]>
BTC: 0.25000000
BTC: 0.00936411
```

2.4.3 Creating your own brain wallet algorithm

Note: A brainwallet refers to the concept of storing Bitcoins in one's own mind by memorization of a passphrase. As long as the passphrase is not recorded anywhere, the Bitcoins can be thought of as existing nowhere except in the mind of the holder. – *Bitcoin Wiki*

The process to create a brain wallet is to use a deterministic random seed based on the hash of a password. To implement this we will use the *entropy* parameter in the creation of the Private Key:

```
import hashlib
from protocoin import keys

def brainwallet(num_bytes):
    hashobj = hashlib.sha256("my super secret password seed")
    return hashobj.digest()

priv_key = keys.BitcoinPrivateKey(entropy=brainwallet)
pub_key = priv_key.generate_public_key()
```

In the example above, a hash (SHA256) is used to create entropy for the generation of the Private Key. The Private Key and the Public Key will be always the same if you always use the same password.

Warning: Remember that if you're going to use this method to generate a key pair and the brain wallet password is forgotten then the Bitcoins are lost forever. Remember to always create backups (encrypted) of your wallet data.

2.5 API Documentation

All modules listed below are under the "protocoin" module.

2.5.1 protocoin.fields – Fields

class protocoin.fields.**BlockLocator**
A block locator type used for getblocks and getheaders

class protocoin.fields.**Field**
Base class for the Fields. This class only implements the counter to keep the order of the fields on the serializer classes.

deserialize(*stream*)
This method must read the stream data and then deserialize and return the deserialized content.

Returns the deserialized content

Parameters **stream** – stream of data to read

parse(*value*)
This method should be implemented to parse the value parameter into the field internal representation.

Parameters **value** – value to be parsed

serialize()
Serialize the internal representation and return the serialized data.

Returns the serialized data

class protocoin.fields.**FixedStringField**(*length*)
A fixed length string field.

Example of use:

```
class MessageHeaderSerializer(Serializer):
    model_class = MessageHeader
    magic = fields.UInt32LEField()
    command = fields.FixedStringField(12)
    length = fields.UInt32LEField()
    checksum = fields.UInt32LEField()
```

class protocoin.fields.**Hash**
A hash type field.

class protocoin.fields.**IPv4AddressField**
An IPv4 address field without timestamp and reserved IPv6 space.

class protocoin.fields.**Int16LEField**
16-bit little-endian integer field.

class protocoin.fields.**Int32LEField**
32-bit little-endian integer field.

class protocoin.fields.**Int64LEField**
64-bit little-endian integer field.

class protocoin.fields.**ListField**(*serializer_class*)
A field used to serialize/deserialize a list of serializers.

Example of use:

```
class TxSerializer(Serializer):
    model_class = Tx
    version = fields.UInt32LEField()
    tx_in = fields.ListField(TxInSerializer)
    tx_out = fields.ListField(TxOutSerializer)
    lock_time = fields.UInt32LEField()
```

class protocoin.fields.**NestedField**(*serializer_class*)
A field used to nest another serializer.

Example of use:

```
class TxInSerializer(Serializer):
    model_class = TxIn
    previous_output = fields.NestedField(OutPointSerializer)
    signature_script = fields.VariableStringField()
    sequence = fields.UInt32LEField()
```

class protocoin.fields.**PrimaryField**
This is a base class for all fields that has only one value and their value can be represented by a Python struct datatype.

Example of use:

```
class UInt32LEField(PrimaryField):
    datatype = "<I"
```

deserialize(*stream*)
Deserialize the stream using the struct data type specified.

Parameters **stream** – the data stream

parse(*value*)
This method will set the internal value to the specified value.

Parameters **value** – the value to be set

serialize()
Serialize the internal data and then return the serialized data.

class protocoin.fields.**UInt16BEField**
16-bit big-endian unsigned integer field.

class protocoin.fields.**UInt16LEField**
16-bit little-endian unsigned integer field.

class protocoin.fields.**UInt32LEField**
32-bit little-endian unsigned integer field.

class protocoin.fields.**UInt64LEField**
64-bit little-endian unsigned integer field.

class protocoin.fields.**VariableIntegerField**
A variable size integer field.

class protocoin.fields.**VariableStringField**
A variable length string field.

2.5.2 protocoin.serializers – Serializers

class protocoin.serializers.**AddressVector**
A vector of addresses.

class protocoin.serializers.**AddressVectorSerializer**
Serializer for the addresses vector.

model_class
alias of *AddressVector*

class protocoin.serializers.**Block**
The block message. This message contains all the transactions present in the block.

class protocoin.serializers.**BlockHeader**
The header of the block.

calculate_hash()
This method will calculate the hash of the block.

class protocoin.serializers.**BlockHeaderSerializer**
The serializer for the block header.

model_class
alias of *BlockHeader*

class protocoin.serializers.**BlockSerializer**
The deserializer for the blocks.

model_class
alias of *Block*

class protocoin.serializers.**GetAddr**
The getaddr command.

class protocoin.serializers.**GetAddrSerializer**
The serializer for the getaddr command.

model_class
alias of *GetAddr*

class protocoin.serializers.**GetBlocks**(*hashes*)
The getblocks command.

class protocoin.serializers.**GetData**
GetData message command.

class protocoin.serializers.**GetDataSerializer**
Serializer for the GetData command.

model_class
alias of *GetData*

class protocoin.serializers.**HeaderVector**
The header only vector.

class protocoin.serializers.**HeaderVectorSerializer**
Serializer for the block header vector.

model_class
alias of *HeaderVector*

class protocoin.serializers.**IPv4Address**
The IPv4 Address (without timestamp).

class protocoin.serializers.**IPv4AddressSerializer**
Serializer for the IPv4Address.

model_class
alias of *IPv4Address*

class protocoin.serializers.**IPv4AddressTimestamp**
The IPv4 Address with timestamp.

class protocoin.serializers.**IPv4AddressTimestampSerializer**
Serializer for the IPv4AddressTimestamp.

model_class
alias of *IPv4AddressTimestamp*

class protocoin.serializers.**Inventory**
The Inventory representation.

type_to_text()
Converts the inventory type to text representation.

class protocoin.serializers.**InventorySerializer**
The serializer for the Inventory.

model_class
alias of *Inventory*

class protocoin.serializers.**InventoryVector**
A vector of inventories.

class protocoin.serializers.**InventoryVectorSerializer**
The serializer for the vector of inventories.

model_class
alias of *InventoryVector*

class protocoin.serializers.**MemPool**
The mempool command.

class protocoin.serializers.**MemPoolSerializer**
The serializer for the mempool command.

model_class
alias of *MemPool*

class protocoin.serializers.**MessageHeader**(*coin='bitcoin'*)
The header of all bitcoin messages.

class protocoin.serializers.**MessageHeaderSerializer**
Serializer for the MessageHeader.

static calc_checksum(*payload*)
Calculate the checksum of the specified payload.

Parameters **payload** – The binary data payload.

model_class
alias of *MessageHeader*

class protocoin.serializers.**NotFound**
NotFound command message.

class protocoin.serializers.**NotFoundSerializer**
Serializer for the NotFound message.

model_class
alias of *NotFound*

class protocoin.serializers.**OutPoint**
The OutPoint representation.

class protocoin.serializers.**OutPointSerializer**
The OutPoint representation serializer.

model_class
alias of *OutPoint*

class protocoin.serializers.**Ping**
The ping command, which should always be answered with a Pong.

class protocoin.serializers.**PingSerializer**
The ping command serializer.

model_class
alias of *Ping*

class protocoin.serializers.**Pong**
The pong command, usually returned when a ping command arrives.

class protocoin.serializers.**PongSerializer**
The pong command serializer.

model_class
alias of *Pong*

class protocoin.serializers.**Serializer**
The main serializer class, inherit from this class to create custom serializers.

Example of use:

```
class VerAckSerializer(Serializer):
    model_class = VerAck
```

deserialize(*stream*)
This method will read the stream and then will deserialize the binary data information present on it.

Parameters **stream** – A file-like object (StringIO, file, socket, etc.)

serialize(*obj*, *fields=None*)
This method will receive an object and then will serialize it according to the fields declared on the serializer.

Parameters **obj** – The object to serializer.

class protocoin.serializers.**SerializerABC**
The serializer abstract base class.

class protocoin.serializers.**SerializerMeta**
The serializer meta class. This class will create an attribute called '_fields' in each serializer with the ordered dict of fields present on the subclasses.

classmethod **get_fields**(*meta*, *bases*, *attrs*, *field_class*)
This method will construct an ordered dict with all the fields present on the serializer classes.

class protocoin.serializers.**Tx**
The main transaction representation, this object will contain all the inputs and outputs of the transaction.

calculate_hash()
This method will calculate the hash of the transaction.

class protocoin.serializers.**TxIn**
The transaction input representation.

class protocoin.serializers.**TxInSerializer**
The transaction input serializer.

model_class
alias of *TxIn*

class protocoin.serializers.**TxOut**
The transaction output.

class protocoin.serializers.**TxOutSerializer**
The transaction output serializer.

model_class
alias of *TxOut*

class protocoin.serializers.**TxSerializer**
The transaction serializer.

model_class
alias of *Tx*

class protocoin.serializers.**VerAck**
The version acknowledge (verack) command.

class protocoin.serializers.**VerAckSerializer**
The serializer for the verack command.

model_class
alias of *VerAck*

class protocoin.serializers.**Version**
The version command.

class protocoin.serializers.**VersionSerializer**
The version command serializer.

model_class
alias of *Version*

2.5.3 protocoin.clients – Clients

class protocoin.clients.**BitcoinBasicClient**(*socket*)
The base class for a Bitcoin network client, this class implements utility functions to create your own class.

Parameters **socket** – a socket that supports the makefile() method.

close_stream()
This method will close the socket stream.

handle_message_header(*message_header*, *payload*)
This method will be called for every message before the message payload deserialization.

Parameters

- **message_header** – The message header
- **payload** – The payload of the message

loop()
This is the main method of the client, it will enter in a receive/send loop.

send_message(*message*)
This method will serialize the message using the appropriate serializer based on the message command and then it will send it to the socket stream.

Parameters **message** – The message object to send

class protocoin.clients.**BitcoinClient**(*socket*)
This class implements all the protocol rules needed for a client to stay up in the network. It will handle the handshake rules as well answer the ping messages.

handle_ping(*message_header*, *message*)
This method will handle the Ping message and then will answer every Ping message with a Pong message using the nonce received.

Parameters

- **message_header** – The header of the Ping message
- **message** – The Ping message

handle_version(*message_header*, *message*)
This method will handle the Version message and will send a VerAck message when it receives the Version message.

Parameters

- **message_header** – The Version message header
- **message** – The Version message

handshake()
This method will implement the handshake of the Bitcoin protocol. It will send the Version message.

2.5.4 protocoin.util – Utility

protocoin.util.**base58_decode**(*address*)
This function converts an base58 string to a numeric format.

Parameters **address** – The base58 string

Returns The numeric value decoded

protocoin.util.**base58_encode**(*address_bignum*)
This function converts an address in bignum formatting to a string in base58, it doesn't prepend the '1' prefix for the Bitcoin address.

Parameters **address_bignum** – The address in numeric format

Returns The string in base58

2.5.5 protocoin.keys – Keys and Address Management

class protocoin.keys.**BitcoinPrivateKey**(*hexkey=None*, *entropy=None*)
This is a representation for Bitcoin private keys. In this class you'll find methods to import/export keys from multiple formats. Use a hex string representation to construct a new Public Key or use the clas methods to import from another format. If no parameter is specified on the construction of this class, a new Private Key will be created.

Parameters

- **hexkey** – The key in hex string format
- **entropy** – A function that accepts a parameter with the number of bytes and returns the same amount of bytes of random data, use a good source of entropy. When this parameter is ommited, the OS entropy source is used.

classmethod **from_string**(*klass*, *stringkey*)
This method will create a new Private Key using the specified string data.

Parameters **stringkey** – The key in string format

Returns A new Private Key

classmethod **from_wif**(*klass*, *wifkey*)
This method will create a new Private Key from a WIF format string.

Parameters **wifkey** – The private key in WIF format

Returns A new Private Key

generate_public_key()
This method will create a new Public Key based on this Private Key.

Returns A new Public Key

to_hex()
This method will convert the Private Key to a hex string representation.

Returns Hex string representation of the Private Key

to_string()
This method will convert the Private Key to a string representation.

Returns String representation of the Private Key

to_wif()
This method will export the Private Key to WIF (Wallet Import Format).

:returns:: The Private Key in WIF format.

class protocoin.keys.**BitcoinPublicKey**(*hexkey*)
This is a representation for Bitcoin public keys. In this class you'll find methods to import/export keys from multiple formats. Use a hex string representation to construct a new public key or use the clas methods to import from another format.

Parameters **hexkey** – The key in hex string format

classmethod **from_private_key**(*klass*, *private_key*)
This class method will create a new Public Key based on a private key.

Parameters **private_key** – The private key

Returns a new public key

to_address()
This method will convert the public key to a bitcoin address.

Returns bitcoin address for the public key

to_hex()
This method will convert the public key to a hex string representation.

Returns Hex string representation of the public key

to_string()
This method will convert the public key to a string representation.

Returns String representation of the public key

2.6 Contribute or Report a bug

Protocoin is an open-source project created and maintained by Christian S. Perone. You can help it by donating or helping with a pull-request or a bug report. You can get the source-code of the project in the Github project page.

2.7 License

BSD License:

```
Copyright (c) 2013, Christian S. Perone
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

        1. Redistributions of source code must retain the above copyright
           notice, this list of conditions and the following disclaimer.
        2. Redistributions in binary form must reproduce the above copyright
           notice, this list of conditions and the following disclaimer in the
           documentation and/or other materials provided with the distribution.
        3. All advertising materials mentioning features or use of this software
       must display the following acknowledgement:
       This product includes software developed by Christian S. Perone.
        4. Neither the name of the Christian S. Perone nor the
```

```
       names of its contributors may be used to endorse or promote products
       derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY CODEFISH ''AS IS'' AND ANY
EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL CODEFISH BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

p

U

V