transactions Documentation

Release 0.2.0

transactions contributors

June 08, 2016

Contents

transactions is a small python library to easily create and push transactions to the bitcoin network.

CHAPTER 1

Installation

```
$ pip install transactions
```

CHAPTER 2

Contents

2.1 Examples

Let's assume the following cast of characters:

- **Alice** with the bitcoin address `'mhyCaF2HFk7CVwKmyQ8TahgVdjnHSr1pTv'`
- **Bob** with the bitcoin address `'mqXz83H4LCxjf2ie8hYNsTRByvtfV43Pa7'`
- **Carol** with the bitcoin address `'mtWg6ccLiZWw2Et7E5UqmHsYgrAi5wqiov'`

Also consider that one bitcoin is made up of satoshi, such that hundred million satoshi is one bitcoin.

Note: With `transactions` all amounts are in satoshi and we currently only support BIP32 wallets (hierarchical deterministic wallets, aka "HD Wallets").

2.1.1 Alice sends 10000 satoshi to Bob

```
from transactions import Transactions

transactions = Transactions(testnet=True)
tx = transactions.simple_transaction(
    'mhyCaF2HFk7CVwKmyQ8TahgVdjnHSr1pTv',
    ('mqXz83H4LCxjf2ie8hYNsTRByvtfV43Pa7', 10000),
)
tx_signed = transactions.sign_transaction(tx, 'master secret')
txid = transactions.push(tx_signed)
print txid
```

2.1.2 Bob sends 600 satoshi to Carol with a custom *op_return*

```
from transactions import Transactions

transactions = Transactions(testnet=True)
tx = transactions.simple_transaction(
    'mqXz83H4LCxjf2ie8hYNsTRByvtfV43Pa7',
    ('mtWg6ccLiZWw2Et7E5UqmHsYgrAi5wqiov', 600),
    op_return='HELLOFROMASCRIBE',
)
```

```
tx_signed = transactions.sign_transaction(tx, 'master secret')
txid = transactions.push(tx_signed)
print txid
```

Check it out fbbd6407b8fc73169918b2fce7f07aff6a486a241c253f0f8eeb942937fbb970

2.1.3 Get transactions of Alice

```
from transactions import Transactions

transactions = Transactions(testnet=True)

transactions.get('mhyCaF2HFk7CVwKmyQ8TahgVdjnHSr1pTv')

{'transactions': [{'amount': -20000,
   'confirmations': 5,
   'time': 1431333905,
   'txid': u'7f4902599ac9e5c9db347228b489c25fe5095f812c979dd84cc4e88f6812db9e'},
  {'amount': -40000,
   'confirmations': 11,
   'time': 1431329129,
   'txid': u'382639448115e859b0dc4092892bc0921edc8851a2b7adbd7b5ab39ccefb73ee'},
 ...
 'unspents': [{'amount': 809760000,
   'confirmations': 5,
   'txid': u'7f4902599ac9e5c9db347228b489c25fe5095f812c979dd84cc4e88f6812db9e',
   'vout': 1}]}
```

2.1.4 Get details of a transaction between Alice and Bob

```
from transactions import Transactions

transactions = Transactions(testnet=True)

transactions.get('382639448115e859b0dc4092892bc0921edc8851a2b7adbd7b5ab39ccefb73ee')

{u'block': 395966,
 u'confirmations': 11,
 u'days_destroyed': u'0.00',
 u'extras': None,
 u'fee': u'0.00010000',
 u'is_coinbased': 0,
 u'is_unconfirmed': False,
 u'time_utc': u'2015-05-11T09:25:29Z',
 u'trade': {u'vins': [{u'address': u'mhyCaF2HFk7CVwKmyQ8TahgVdjnHSr1pTv',
    u'amount': -0.0004,
    u'is_nonstandard': False,
    u'n': 3,
    u'type': 0,
    u'vout_tx': u'dece4f3d0de255bb53c20e89271d1236929d72e426e6e7860d97564c6b9e26ab'}],
  u'vouts': [{u'address': u'mqXz83H4LCxjf2ie8hYNsTRByvtfV43Pa7',
    u'amount': 0.0001,
    u'is_nonstandard': False,
    u'is_spent': 0,
    u'n': 0,
```

```
    u'type': 1},
...
u'type': 1}]}
```

2.2 Code

2.2.1 `transactions` module

class `transactions.`**`Transactions`**(*service=u'blockr'*, *testnet=False*, *username=u''*, *password=u''*, *host=u''*, *port=u''*)

Transactions: Bitcoin for Humans

All amounts are in satoshi

`__init__`(*service=u'blockr'*, *testnet=False*, *username=u''*, *password=u''*, *host=u''*, *port=u''*)

Parameters

- **`service`** (*str*) – currently supports _blockr_ for blockr.io and and _daemon_ for bitcoin daemon. Defaults to _blockr_
- **`testnet`** (*bool*) – use True if you want to use tesnet. Defaults to False
- **`username`** (*str*) – username to connect to the bitcoin daemon
- **`password`** (*str*) – password to connect to the bitcoin daemon
- **`hosti`** (*str*) – host of the bitcoin daemon
- **`port`** (*str*) – port of the bitcoin daemon

`build_transaction`(*inputs*, *outputs*)

Thin wrapper around `bitcoin.mktx(inputs, outputs)`

Parameters

- **`inputs`** (*dict*) – inputs in the form of `{'output': 'txid:vout', 'value': amount in satoshi}`
- **`outputs`** (*dict*) – outputs in the form of `{'address': to_address, 'value': amount in satoshi}`

Returns transaction

`get`(*hash*, *account=u'*'*, *max_transactions=100*, *min_confirmations=6*, *raw=False*)

Parameters

- **`hash`** – can be a bitcoin address or a transaction id. If it's a bitcoin address it will return a list of transactions up to `max_transactions` a list of unspents with confirmed transactions greater or equal to `min_confirmantions`
- **`account`** (*Optional[str]*) – used when using the bitcoind. bitcoind does not provide an easy way to retrieve transactions for a single address. By using account we can retrieve transactions for addresses in a specific account

Returns transaction

`push`(*tx*)

Parameters **`tx`** – hex of signed transaction

Returns pushed transaction

sign_transaction(*tx*, *master_password*, *path=u''*)

Parameters

- **tx** – hex transaction to sign
- **master_password** – master password for BIP32 wallets. Can be either a master_secret or a wif
- **path** (*Optional[str]*) – optional path to the leaf address of the BIP32 wallet. This allows us to retrieve private key for the leaf address if one was used to construct the transaction.

Returns signed transaction

Note: Only BIP32 hierarchical deterministic wallets are currently supported.

simple_transaction(*from_address*, *to*, *op_return=None*, *min_confirmations=6*)

Parameters

- **from_address** (*str*) – bitcoin address originating the transaction
- **to** – tuple of `(to_address, amount)` or list of tuples `[(to_addr1, amount1), (to_addr2, amount2)]`. Amounts are in *satoshi*
- **op_return** (*str*) – ability to set custom `op_return`
- **min_confirmations** (*int*) – minimal number of required confirmations

Returns transaction

2.2.2 service module

Defines the main `BitcoinService` class which other services should subclass.

class `transactions.services.service.`**BitcoinService**(*testnet=False*)

_min_dust
int

Minimum tx accepted by blockr.io. Defaults to''3000''.

maxTransactionFee
int

Maximum transaction fee. Defaults to `50000`.

_min_transaction_fee
int

Minimum mining fee. Defaults to `30000`.

Todo

Give a bit more explanations about each attribute.

__init__(*testnet=False*)

Parameters **testnet** (*bool*) – Set to `True` to use the testnet. Defaults to False, meaning that the mainnet will be used.

2.3 Theory

The intent here is to provide some kind of fundamental knowledge with respect to bitcoin.

As a starting point the material here is currently heavily inspired by the draft version of the book Bitcoin and Cryptocurrency Technologies by

- Arvind Narayanan, Princeton University
- Joseph Bonneau, Princeton University
- Edward Felten, Princeton University
- Andrew Miller, University of Maryland
- Steven Goldfeder, Princeton University
- Jeremy Clark, Concordia University

The long term intention is to extend the material as much as it makes sense meanwhile weaving a connection to the engineering side of bitcoin.

2.3.1 Computer Science of Bitcoin

The goal of this section is to dwell on the fundamentals of bitcoin from the point of view of data structures and algorithms.

Some of the key concepts are:

- secure hash functions
- hash pointers and pointer-based acyclic data structures
- digital signatures
- cryptocurrencies

Cryptographic Hash Functions

Very briefly, a basic hash function has three main characteristics:

- input value is a string of any size
- output value is of fixed size (i.e.: 256 bits)
- for a string of n bits, the hash function has a running time of O(n)

This is more or less good enough to implement a hash table.

In order to make the basic hash function cryptographically secure, three additional characteristics are required:

- collisionresistance
- hiding
- puzzlefriendliness

A hash collition means that for two different input strings the hash function returns the same hash.

Hash functions have collisions since the number of possible inputs is infinite whereas the number of possible outputs is finite.

collisionresistance A hash function is collision-resistant if it is computationally hard to find its collisions.

hiding Reverse engineering a hash function is computationally hard. That is, given the output of a hash function, the input string cannot be found.

puzzlefriendliness Very roughly this means that one can pick a puzzle id, k, and bind it to a target result y, such that it is difficult to find a value x, which when fed to the hash function in combination with k, will yield y. By difficult, is meant that there are no better approaches than random trials, and that finding x requires substantial time, more than 2^n for if y has n bits.

Hash function in use in Bitcoin

Several cryptocurrencies like Bitcoin use a hash function named SHA-256 for verifying transactions and calculating proof-of-work or proof-of-stake. [1]

For a more in-depth study of the SHA-256 hash function one may consult Descriptions of SHA-256, SHA-384, and SHA-512 by NIST.

Hash Pointer -based Data Structures

A hash pointer points to a location where data is stored along with the hash of that data at a given point in time.

Using a hash pointer one can retrieve the data, and verify that the data hasn't changed.

Using hash pointers, one can build various pointer-based acyclic data structures such as linked lists, trees, and more generally directed acyclic graphs.

The bitcoin blockchain can be viewed as a linked list of binary trees, relying on hash pointers. The hash pointer -based linked list is more precisely called a hash chain, whereas the hash pointer -based binary tree is called a hash tree, or Merkle tree, named after its inventor Ralph Merkle.

Transactions are assembled into a hash tree to form a "block." Those blocks are then linked to form a hash chain (block chain).

Note: Binary hash trees make it relatively efficient to show the chain of transactions a transaction is linked to within a tree. For a tree with n transactions, only about log(n) transactions are necessary.

Digital Signatures

A digital signature requires three steps:

- private / public key pair generation
- signature
- verification

Expressed in code:

[1] https://en.wikipedia.org/wiki/SHA-2#Applications

```
private_key, public_key = generate_key_pair(key_size, passphrase=None)

signature = sign(private_key, message)

is_valid = verify(public_key, message, signature)
```

There are two important requirements, one somewhat obvious, and the other more complex.

- Valid signatures must verify. That is:

```
verify(public_key, message, sign(private_key, message)) is True
```

- Reverse engineering the digital signature scheme, aka forging signatures is computationally impossible. That is, for any given message for which the the signature, and public key are known, it is not possible to find the private key, or to figure out how to create new valid signatures for different messages.

ECDSA: digital signature used in Bitcoin

For its digital signatures Bitcoin uses the Elliptic Curve Digital Signature Algorithm (ECDSA) [3] with a specific curve that is fine-tuned via the domain parameters known as secp256k1.

Sizes of keys, message, and signature when using ECDSA [2]

- Private key: 256 bits
- Public key, uncompressed: 512 bits
- Public key, compressed: 257 bits
- Message to be signed: 256 bits
- Signature: 512 bits

Public Keys as Identities & Bitcoin Addresses

Using a digital signature scheme, public keys can be used as identities. In Bitcoin, public keys are used to identify the sender and receiver in a transaction. Bitcoin refers to these public keys as "addresses". The sender can sign the transaction with their private key, meanwhile the receiver can verify the signature of the transaction using the public key of the sender.

Two Simple Cryptocurrency Models

To make it easier to understand how bitcoin works, *Narayanan et al.* [2] present two simplified cryptocurrency models, which they call "GoofyCoin", and "ScroogeCoin". The first model (GoofyCoin) is somewhat naive, especially with respect to double-spending attacks, and is therefore insecure. The second model (ScroogeCoin) resolves the double-spending attack problem, but depends on the honesty of Scrooge, and is therefore centralized. This section briefly reviews these two models, which are somewhat useful to build upon to understand how bitcoin works.

Note: Double-spending attacks

[3] https://en.bitcoin.it/wiki/Elliptic_Curve_Digital_Signature_Algorithm
[2] https://d28rh4a8wq0iu5.cloudfront.net/bitcointech/readings/princeton_bitcoin_book.pdf

"Double-spending is the result of successfully spending some money more than once. Bitcoin protects against double spending by verifying each transaction added to the block chain to ensure that the inputs for the transaction had not previously already been spent." [4]

GoofyCoin: a ledger-less cryptocurrency

The GoofyCoin cryptocurrency model is based on two main principles:

1. One authority (Goofy) can create coins at will, and assign these newly created coins to themself.
2. The owner of a coin can transfer their coin to whomever they wish.

Coin Creation

The creation of a `goofycoin` works like so:

```
coin_id = generate_unique_coin_id()
coin_creation_msg = 'create_coin [coin_id]'
coin_creation_signature = sign(goofy_private_key, coin_creation_msg)
```

The `coin_creation_msg` and `coin_creation_signature`, taken together, form a coin. For this example, let's say that a coin is a tuple:

```
goofycoin = (coin_creation_msg, coin_creation_signature)
```

In a more explicit manner:

```
goofycoin = (
    'create_coin [coin_id]',
    sign(goofy_private_key,  'create_coin [coin_id]'),
)
```

Using the public key of Goofy, anyone can verify whether a `goofycoin` is valid:

```
is_valid = verify(goofy_public_key, goofy_coin[0], goofy_coin[1])
```

or more explicitly:

```
is_valid = verify(
    goofy_public_key,
    'create_coin [coin_id]',
    coin_creation_signature,
)
```

Lastly, in order to be able to refrence the coin, in future transactions, we can hash the information of the coin, such that referencing the coin will be done via the hash. So let's assume the following dictionary, for the coin creation transaction:

```
transaction = {
    coin_hash: 'a9f268dbfda',
    coin : (
        'create_coin [coin_id]',
        sign(goofy_private_key,  'create_coin [coin_id]'),
    )
}
```

[4] https://en.bitcoin.it/wiki/Double-spending

Coin Transfer

To transfer the above coin (`a9f268dbfda`) to Alice, Goofy would create the following transaction:

```
transaction = {
    coin_hash: 'b3a364d1a1z',
    coin : (
        'pay_to alice_pubkey: a9f268dbfda',
        sign(goofy_private_key, 'pay_to alice_pubkey: a9f268dbfda'),
    )
}
```

If Alice wanted to transfer her new coin (`b3a364d1a1z`) to Bob, she would then create the following transaction:

```
transaction = {
    coin_hash: '86b9dd63864',
    coin : (
        'pay_to bob_pubkey: b3a364d1a1z',
        sign(goofy_private_key, 'pay_to bob_pubkey: b3a364d1a1z'),
    )
}
```

Double-spending The GoofyCoin model does not prevent Alice from transferring the same coin to multiple recipients. Hence, in addition to the previous transfer she made to Bob, Alice could transfer the same coin to Carol:

```
transaction = {
    coin_hash: 'a1z2g5pw34',
    coin : (
        'pay_to carol_pubkey: b3a364d1a1z',
        sign(goofy_private_key, 'pay_to carol_pubkey: b3a364d1a1z'),
    )
}
```

The two transactions (`86b9dd63864`, `a1z2g5pw34`) are conflicting, because two people can't own the same coin at the same time.

Next section will show how double-spending attacks can be prevented via a centralized ledger, which keeps track of past transactions.

ScroogeCoin – a ledger-based cryptocurrency

The ScroogeCoin model relies on an append-only public ledger in which transactions are permanently recorded.

The ledger is maintained by a trusted authority, Scrooge, who can also issue new coins.

A rough sketch of the data structure of the ledger is as follwos;

```
{'prev': 0,
 'tx_id': 0,
 'tx': {...}},

{'prev': hash_function(tx_0),
 'tx_id': 1,
 'tx': {...}},

{'prev': hash_function(tx_1),
 'tx_id': 2,
```

```
 'tx': {...}},

...
```

The chain of transactions cannot be tempered with because of the use of hash pointers. For example, if the content of transaction *1* was changed, the pointer in transaction *2* would no longer point to transaction *1*, and the chain would be broken.

The final hash pointer of the chain is signed by the trusted authority, Scrooge, who then publishes the chain. In this model, a transaction that is not in the signed chain is ignored. Ii is the responsibility of the trusted authority to verify that a transaction is not a double spend.

The ScroogeCoin model supports two types of transactions;

`create_coins` Creates new coins, and is valid if signed by the trusted authority, Scrooge.

`pay_coins` Consumes coins, and produces new coins of the same value, that may belong to new people. The transaction must be signed by each owner of the consumed coins. Moreover, each input coin must not have been already spent.

A new transaction must be validated by the trusted authority, and once validated will be signed and added to the chain of transactions, at which point, and only then, the new transaction will be considered to have occurred.

This model works reasonably well, except for the dependence on a trusted authority. In brief:

- The very existence of the chain relies on one central power.
- The central power can create unlimited amount of coins for itself.
- The central power can deny service to whomever it wishes by simply ignoring transactions.
- The central power can require users of the system to pay fees in order for their transactions to be considered.

The above problems seem sufficient to motivate efforts to decentralize the ScroogeCoin model. This brings the next topic of study: how can such a system be efficiently decentralized?

2.3.2 References

2.4 Practice

2.4.1 Running a bitcoin node in regtest mode

```
bitcoind -regtest
```

Bitcoin clients: bitcoin-cli & json rpc

json rpc ref: https://en.bitcoin.it/wiki/API_reference_%28JSON-RPC%29

- via curl
- via python with python-bitcoinrpc
- via python with requests
- via transactions

curl

```
$ curl --user user --data-binary  \
    '{"jsonrpc": "1.0", "id":"dummy", "method": "getinfo", "params": [] }'  \
    -H 'content-type: text/plain;' http://127.0.0.1:18332/
```

bitcoin json rpc revisited with docker

We can repeat the same as in the previous section, but this time with some parts, and everything dockerized.

We add the twist that:

1. the bitcoin server is running in a container, meanwhile client calls are made from the docker host
2. the bitcoin server and client are running in separate containers

For 1. and 2. we connect to the bitcoin node:

- via curl
- via python with python-bitcoinrpc
- via python with requests
- via transactions

host - container

Running bitcoind in container and making rpc calls to it from the host machine, (sender_ip)

given the following `bitcoin.conf`:

```
dnsseed=0
rpcuser=a
rpcallowip=<sender_ip>
```

```
docker run --rm --name btc -v ~/.bitcoin-docker:/root/.bitcoin -p <sender_ip>:58332:18332 btc5 bitcoi
```

```
curl --user a:b --data-binary '{"jsonrpc": "1.0", "id":"", "method": "getinfo", "params": [] }' -H 'c
```

container-container

We can use docker-compose.

In one shell:

```
$ docker-compose run --rm bitcoin
```

In another shell:

```
$ docker ps

CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
94787e1325a3        sbellem/bitcoin     "bitcoind -regtest -p"   5 seconds ago       Up 5 seconds
```

Using the `CONTAINER ID` or `NAME`:

```
$ docker exec -it transactions_bitcoin_run_1 bash
# bitcoin-cli -regtest getinfo
```

```
root@94787e1325a3:/# curl --user a:b --data-binary \
    '{"jsonrpc": "1.0", "id":"", "method": "getinfo", "params": [] }' \
    -H 'content-type: text/plain;' http://localhost:18332 \
    | python -m json.tool

% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
Dload  Upload   Total   Spent    Left  Speed
100   386  100   323  100    63  57483  11211 --:--:-- --:--:-- --:--:-- 64600
{
    "error": null,
    "id": "",
    "result": {
        "balance": 0.0,
        "blocks": 0,
        "connections": 0,
        "difficulty": 4.656542373906925e-10,
        "errors": "",
        "keypoololdest": 1459269071,
        "keypoolsize": 101,
        "paytxfee": 0.0,
        "protocolversion": 70012,
        "proxy": "",
        "relayfee": 1e-05,
        "testnet": false,
        "timeoffset": 0,
        "version": 120000,
        "walletversion": 60000
    }
}
```

2.5 Under the Hood

The intent of this section is to document what goes on under the hood of `transactions`.

We'll use three main "pillars" to organize and present the information:

- *Bitcoin Network*
- *Bitcoin Addresses*
- *Bitcoin Transactions*

One additional section will be used to present some key aspects of the libraries that `transactions` rely on, especially the two bitcoin libraries: `pycoin` and `pybitcointools`.

- *Libraries used by transactions*

2.5.1 Bitcoin Network

There are multiple ways that one may connect to the bitcoin network. For the sake of simplicity, let's say that they are two main ways:

via a daemon that is communicating directly with a node located at a specific host and port

via a blockchain explorer that is communicating with the bitcoin network via the public API of the blockchain explorer service such as blockr.io

Different Modes of the Bitcoin Network

The bitcoin daemon and other bitcoin core programs can be run in three different "network modes":

mainnet *The original and main network for Bitcoin transactions, where satoshis have real economic value.* [1]

testnet *A global testing environment in which developers can obtain and spend satoshis that have no real-world value on a network that is very similar to the Bitcoin.* [2]

regtest *A local testing environment in which developers can almost instantly generate blocks on demand for testing events, and can create private satoshis with no real-world value.* [3]

Running a bitcoin node in regtest mode

bitcoin json rpc

ref: https://en.bitcoin.it/wiki/API_reference_%28JSON-RPC%29

- via curl
- via python with python-bitcoinrpc
- via python with requests
- via transactions

curl

```
$ curl --user user --data-binary  \
    '{"jsonrpc": "1.0", "id":"dummy", "method": "getinfo", "params": [] }'  \
    -H 'content-type: text/plain;' http://127.0.0.1:18332/
```

docker

host - container

Runnign bitcoind in container and making rpc calls to it from the host machine, (sender_ip)

given the following `bitcoin.conf`:

```
dnsseed=0
rpcuser=a
rpcallowip=<sender_ip>
```

```
docker run --rm --name btc -v ~/.bitcoin-docker:/root/.bitcoin -p <sender_ip>:58332:18332 btc5 bitcoi
```

```
curl --user a:b --data-binary '{"jsonrpc": "1.0", "id":"", "method": "getinfo", "params": [] }' -H 'c
```

[1] https://bitcoin.org/en/glossary/mainnet
[2] https://bitcoin.org/en/glossary/testnet
[3] https://bitcoin.org/en/glossary/regression-test-mode

container-container

Making rpc calls from a container to the bitcoind running in another container.

Connecting to the Bitcoin Network with `transactions`

When using `transactions`, one can interact with the bitcoin network via a daemon or via a blockchain explorer. When connecting via a daemon it is possible to connect to the three networks: mainnet, testnet, or regtest, whereas when connecting via a blockchain explorer one may connect to the mainnet or testnet.

The supported blockchain explorer is blockr.io

Todo

show code examples

2.5.2 Bitcoin Addresses

Todo

Show how a bitcoin address is created.

2.5.3 Bitcoin Transactions

Todo

Show the different steps required to publish a transaction in the bitcoin network.

Lifecycle of a transaction: creation, signing, publishing, confirmation

- Using `create` to fetch a transaction
- Using `sign` to fetch a transaction
- Using `push` to publish a transaction
- Using `get` to fetch a transaction

Elements of the payload of a transaction

Blocks

Transactions are assembled into blocks.

credits:

- https://gist.github.com/shirriff/c9fb5d98e6da79d9a772#file-merkle-py
- https://github.com/richardkiss/pycoin

Example:

```
$ curl https://blockexplorer.com/api/block/0000000000000000e067a478024addfecdc93628978aa52d91fabd4292
```

Or in python:

```
import json

import requests


BLOCKEXPLORER_API_URL = 'https://blockexplorer.com/api'
BLOCKHASH = '0000000000000000e067a478024addfecdc93628978aa52d91fabd4292982a50q'

url = '{}/block/{}'.format(BLOCKEXPLORER_API_URL, blockhash)
response = requests.get(url)
block = json.loads(response.content)

block
{u'bits': u'19015f53',
 u'chainwork': u'0000000000000000000000000000000000000000001a6eca45b2459ce9eed8',
 u'confirmations': 120187,
 u'difficulty': 3129573174.5222874,
 u'hash': u'0000000000000000e067a478024addfecdc93628978aa52d91fabd4292982a50',
 u'height': 286819,
 u'isMainChain': True,
 u'merkleroot': u'871714dcbae6c8193a2bb9b2a69fe1c0440399f38d94b3a0f1b447275a29978a',
 u'nextblockhash': u'0000000000000000b0f08ec6a3d1e84994498ecf993a9981f57982cfdb66c443',
 u'nonce': 856192328,
 u'poolInfo': {u'poolName': u'ghash.io', u'url': u'https://ghash.io/'},
 u'previousblockhash': u'00000000000000117c80378b8da0e33559b5997f2ad55e2f7d18ec1975b9717',
 u'reward': 25,
 u'size': 152509,
 u'time': 1392872245,
 u'tx': [u'00baf6626abc2df808da36a518c69f09b0d2ed0a79421ccfde4f559d2e42128b',
         u'91c5e9f288437262f218c60f986e8bc10fb35ab3b9f6de477ff0eb554da89dea',
         u'46685c94b82b84fa05b6a0f36de6ff46475520113d5cb8c6fb060e043a0dbc5c',
         u'ba7ed2544c78ad793ef5bb0ebe0b1c62e8eb9404691165ffcb08662d1733d7a8',
         u'b8dc1b7b7ed847c3595e7b02dbd7372aa221756b718c5f2943c75654faf48589',
         ...]
 u'version': 2}


'merkleroot': u'871714dcbae6c8193a2bb9b2a69fe1c0440399f38d94b3a0f1b447275a29978a',
```

The merkle root corresponds to the cummulative hashing of the transactions hashes.

That is, each transaction is hashed. Each hash is a leaf of a binary tree.

A binary tree is built by pairing leaves, concatenating the pair, and computing the hash of the concatenated pair. The same process is repeated for the parent, recursively all the way to the root, resulting in the merkle root.

At each level of the tree, if the number of hashes is odd, then the last hash is included twice.

Progammatically, this means:

```
def merkleroot(hashes);
    if len(hashes) == 1:
        return hashes[0]
    if len(hashes) % 2 == 1;
        hashes.append(hashes[-1])
    parent_hashes = []
```

```
    for i in range(0, len(hashes), 2);
        h = sec_hash_algo(hashes[i] + hashes[i+1])
        parent_hashes.append(h)
    return merkle_root(parent_hashes)
```

Todo

bitcoin data dir

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

2.5.4 Libraries used by `transactions`

Todo

Present libraries used; `requests`, `pycoin`, `pybitcointools`

Dive into the details of how pycoin and pybitcointools are used and work under the hood.

2.5.5 References

2.6 Contributing

Pull requests, feedback, and suggestions are welcome. The github repository is at https://github.com/ascribe/transactions

You can also send inquiries directly to devel@ascribe.io

2.7 Background

This was developed by ascribe GmbH as part of the overall ascribe technology stack. http://www.ascribe.io

2.8 License

Licensed under the Apache License, Version 2.0.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

t

Symbols

B

G

M

P

S

T