How to Program a

Block Chain Explorer

with Python and Bitcoin

A step-by-step guide
to programming a block explorer for
reading block and transaction data
from block chain structures

BY ALEX GORALE

How to Program a **Block Chain Explorer** with Python and Bitcoin

A Step-by-Step Guide to
Programming a Block Explorer for Reading
Block and Transaction Data from
Block Chain Structures

Alex Gorale
www.AlexGorale.com

Agora Publishing

Table of Contents

Introduction

Bitcoin is the world's first decentralized payment network. It is a global network of computers that control immense processing power - miners. The protocol that directs the network transforms these computers into digital accountants. Together they maintain a data structure named the block chain.

The block chain used in Bitcoin is like an accounting ledger. Users broadcast their transactions to a peer to peer network and miners check to make sure the transaction is valid. If a transaction is accepted it is stored in the miners' memory pools. When a new block is mined the transactions are included inside the block data structure.

Once a transaction has been mined inside of a block it is considered confirmed by the network. Each additional block mined afterwards is considered an additional confirmation.

Because Bitcoin is an open source, public system it is available for auditing, data mining, research, and more, any time. Although a common misconception, Bitcoin does not use encryption to store any information.

This book assumes you are already familiar with Python programming and Bitcoin concepts and will teach you how to use the Python programming language to decipher the information contained within the Bitcoin Block Chain.

Setup

The examples in this book are run on an Ubuntu 14.04 system using Python 2.7.6. The code contained within is OS agnostic and will work on Windows, Mac or Linux systems that run Python 2.

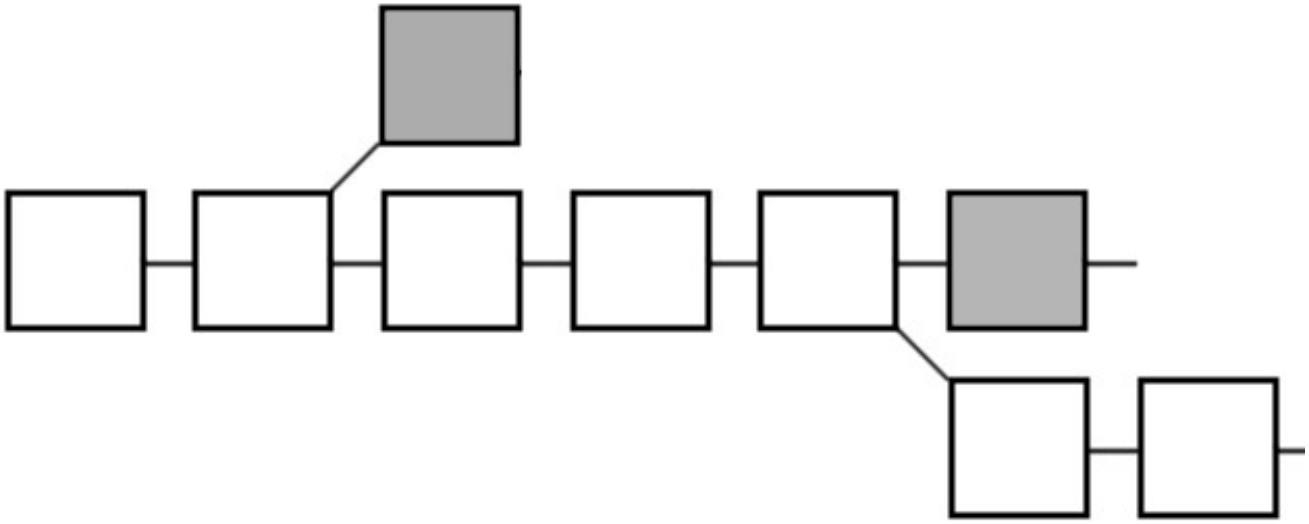
If you wish to explore an entire block chain you will have to download that yourself using a full-node client like Bitcoin Core. This book will use Bitcoin and Litecoin's Genesis Block as example data and provide data from both block chains in a Github repository, as well as full copies of the source code. The explorer is not limited to just Bitcoin or Litecoin and can be modified for currencies that deviate from the Bitcoin protocol.

Goals

By the end of this guide you will learn the following:

1. How data is stored in a block chain using Bitcoin as the example.
2. How transactions are tracked through the block chain's ledger system.
3. How the block chain data structure defined by a cryptocurrency protocol maps to software classes
4. How to read data from a block chain using the Python programming language with Bitcoin and Litecoin block data as examples.

What is a Block Chain?



The Block Chain is just a metaphor for visualizing how the Bitcoin protocol stores information. Literally, the block chain is a digital file. Blocks are indexed sections of the block chain file.

Currently, Bitcoin software caps the size of a block at 1 megabyte. The first 88 bytes of each block contains identifying meta data about the block. The remainder of the space is used to store transaction information.

Block chains are used to establish decentralized consensus among computer systems. The most widely known and successful implementation of a block chain is currently Bitcoin. Even though the block chain files are stored locally on users' computers the content is identical for every machine following the Bitcoin protocol.

In Bitcoin mining powerful ASIC chips are looking for a portion of a cryptographic hash that falls within a tiny range of numbers. Bitcoin sets its range of numbers such that a correct number is found approximately every ten minutes. The protocol accounts for new machines joining the mining network by readjusting the mining difficulty every ten days.

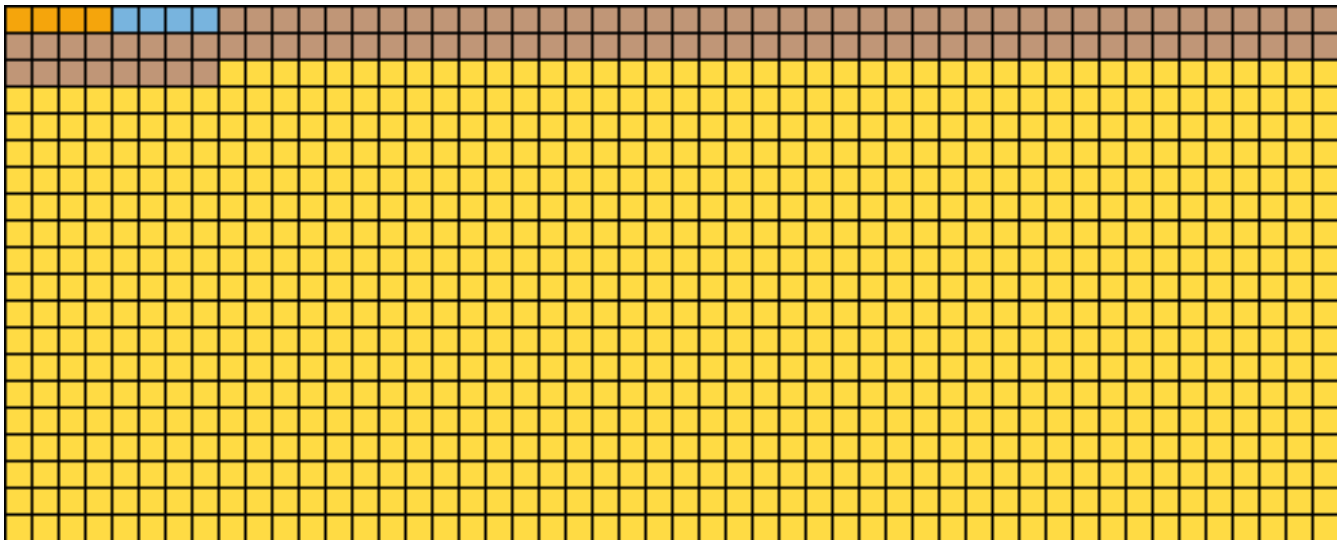
When a correct guess is made the information that makes up a Bitcoin block is broadcast to the rest of the Bitcoin network. If the rest of the network validates the block before anyone else the miner successfully mines a block. The data broadcast by that miner is appended to the block chain files across the entire global network.

Each block contains 88 bytes of identifying meta data and a list of transactions. Each transaction can be thought of as two array lists containing transaction inputs and outputs.

Each block contains a reference to the block that came before. This linkage is the chain portion of the term block chain. Blocks are aware of the block that came before it – similar to a singly-linked list. However, the data is written to disk sequentially, seeking between blocks using the magic number is more practical than following the previous block hash stored in the block header.

How is Data Structured Inside a Block?

At the byte-level a block looks like this:



- **Magic Number** – 4 Bytes; Used to identify origin network and delimit blocks for seek operations. Hex Value: F9 BE B4 D9
- **Blocksize** – 4 Bytes; Indicates the number of following bytes until the end of the block
- **Block Header** – 80 Bytes; 4 bytes version, 32 bytes previous block hash, 32 bytes transaction merkle root, 4 byte timestamp, 4 byte network difficulty, 4 byte cryptographic nonce.
- **Transactions** – Dynamic. 1-9 byte Variable Integer transaction counter. The remaining block space is used determined by the number of transactions.

Just like blocks, a transaction is made up of smaller structures. Every transaction contains a version number, a list for inputs and outputs, a variable integer counter for inputs and outputs, and a lock time. The lock time prevents the transaction from being mined until the block chain reaches the indicated height, or chain length.

Each input contains reference data to a past transaction output. The input frees that previous output balance of bitcoin to be spent in the current transaction outputs. Multiple inputs may be included in a transaction. Bitcoin are spent in a transaction as indicated by the outputs.

Each output contains a value describing a sum of bitcoin, a Bitcoin payment Script, and a variable integer indicating the length (in bytes) of the payment Script. Bitcoin uses a scripting system to determine the instructions for spending bitcoin. The majority of scripts follow two rules for Bitcoin to be spent:

1. Provide the public key that hashes to the given address
2. Provide a signature from the matching private key to prove ownership of that public key

An output is one half of a transaction. We can say that every Bitcoin transaction effectively completes a previous transaction and begins another.

Every transaction requires an input except for the first transaction inside of every block. This special transaction is called a coinbase transaction. It is how new bitcoin are created and miner's receive their fees. Transactions whose outputs do not consume the total bitcoin unlocked by their inputs contribute those coins to miner fees and their sum is added to the block reward in the coinbase.

It is possible to trace every Bitcoin transaction by matching

inputs to outputs. Eventually, the path would lead directly to the original block and coinbase transaction where the coins were created. Similarly, matching outputs to inputs would show you how the coins were spent over time leading directly to their current wallet holders.

The code in the next section follows the rules as of the release of Bitcoin Core 0.11.0. To maintain compatibility with other block chains or hard forks to Bitcoin it's important to understand that the classes are built to read data directly from the hard drive as the block chain protocol describes. Since change is the only thing certain in Bitcoin reference the official Bitcoin wiki's description of the protocol.

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

The Bitcoin wiki will track changes and give you the ability to scale your source code with any changes to Bitcoin, or other block chain, you are reading.

Building the Software

Once you have python installed and configured create a new file and save the following code as a '.py' file. I named mine 'blocktools.py'. This file contains the building blocks for the classes we will build to read the block chain later.

```
import struct

def uint1(stream):
     return ord(stream.read(1))

def uint2(stream):
     return struct.unpack('H', stream.read(2))[0]

def uint4(stream):
     return struct.unpack('I', stream.read(4))[0]

def uint8(stream):
     return struct.unpack('Q', stream.read(8))[0]

def hash32(stream):
     return stream.read(32)[::-1]

def time(stream):
     time = uint4(stream)
     return time

def varint(stream):
     size = uint1(stream)

     if size < 0xfd:
          return size
     if size == 0xfd:
          return uint2(stream)
     if size == 0xfe:
          return uint4(stream)
     if size == 0xff:
          return uint8(stream)
     return -1

def hashStr(bytebuffer):
     return ''.join(('%x'%ord(a)) for a in bytebuffer)
```

Each method will read a number of bytes from a binary file and return an unsigned value. These methods will be used according to the way Bitcoin defines. For this I used the Bitcoin wiki.

Block structure

Field	Size
Magic no	4 bytes
Blocksize	4 bytes
Blockheader	80 bytes
Transaction counter	1 - 9 bytes
transactions	<Transaction counter>-many transactions

Field	Size (Bytes)
Version	4
hashPrevBlock	32
hashMerkleRoot	32
Time	4
Bits	4
Nonce	4

Eventually these methods will be used in classes to map out objects as specified by the protocol. First, we're going to make a unit test and read the Bitcoin Genesis Block in a unit test.

Make a second .py file. I call mine 'parser.py' and copy the following code:

```
#/usr/bin/python

from blocktools import *

with open('1M.dat', 'rb') as blockfile:
                print "Magic Number:\t %8x" % uint4(blockfile)
                print "Blocksize:\t %u" % uint4(blockfile)

                """Block Header"""
                print "Version:\t %d" % uint4(blockfile)
                print "Previous Hash\t %s" %
hashStr(hash32(blockfile))
                print "Merkle Root\t %s" % hashStr(hash32(blockfile))
                print "Time\t\t %s" % str(time(blockfile))
                print "Difficulty\t %8x" % uint4(blockfile)
                print "Nonce\t\t %s" % uint4(blockfile)

                print "Tx Count\t %d" % varint(blockfile)

                print "Version Number\t %s" % uint4(blockfile)
                print "Inputs\t\t %s" % varint(blockfile)
                print "Previous Tx\t %s" % hashStr(hash32(blockfile))
                print "Prev Index \t %d" % uint4(blockfile)
                script_len = varint(blockfile)
                print "Script Length\t %d" % script_len
                script_sig = blockfile.read(script_len)
                print "ScriptSig\t %s" % hashStr(script_sig)
                print "ScriptSig\t %s" %
hashStr(script_sig).decode('hex')
                print "Seq Num\t\t %8x" % uint4(blockfile)

                print "Outputs\t\t %s" % varint(blockfile)
                print "Value\t\t %s" %
str((uint8(blockfile)*1.0)/100000000.00)
                script_len = varint(blockfile)
                print "Script Length\t %d" % script_len
                script_pubkey = blockfile.read(script_len)
                print "Script Pub Key\t %s" % hashStr(script_pubkey)
                print "Lock Time %8x" % uint4(blockfile)
                print
```

Running the parser.py unit test script against the first block in the Bitcoin block chain – the Genesis Block - will give the following output:

```
Magic Number:    d9b4bef9
Blocksize:  285
Version:    1
Previous Hash    0000000000000000000000000000000000
Merkle Root
      4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda3
      3b
Time        1231006505
Difficulty  1d00ffff
Nonce       2083236893
Tx Count    1
Version Number   1
Inputs      1
Previous Tx 0000000000000000000000000000000000
Prev Index  4294967295
Script Length    77
ScriptSig
      4ffff01d1445546865205469 6d65732030332f4a616e2f3230303920436861
      6e63656c6c6f72206f6e206272696e6b206f66207365636f6e64206261696c
      6f757420666f722062616e6b73
ScriptSig   O??##EThe Times 03/Jan/2009 Chancellor on brink of
      second bailout for banks
Seq Num          ffffffff
Outputs          1
Value       50.0
Script Length    67
Script Pub Key
      414678afdb0fe5548271967f1a67130b7105cd6a828e0399a67962e0ea1f61
      deb649f6bc3f4cef38c4f3554e51ec112de5c384df7bab8d578a4c702b6bf1
      1d5fac
Lock Time        0
```

For Windows users you'll need to copy your blk*.dat files from your bitcoin data directory.

Users with access to the command line can use dd to pull out a sample data set.

```
    dd if=.bitcoin/blocks/blk00000.dat of=1M.dat bs=1M count=1
```

The above command will make a copy of the first megabyte of data from the first block file in the Bitcoin Block Chain. This will be the sample data.

If your code ran congratulations on successfully parsing the first block of the Bitcoin block chain. If not then try cloning the github repository:

https://github.com/tenthirtyone/blocktools

And run the parser.py script contained there. This repository contains sample block chain data for Bitcoin and Litecoin.

The parser.py file is good for testing out methods but it's not a streaming reader yet.

Create another file. Name this one 'block.py'. This file contains the classes for the block chain objects. Copy the following code inside:

```
from blocktools import *

class BlockHeader:
     def __init__(self, blockchain):
          self.version = uint4(blockchain)
          self.previousHash = hash32(blockchain)
          self.merkleHash = hash32(blockchain)
          self.time = uint4(blockchain)
          self.bits = uint4(blockchain)
          self.nonce = uint4(blockchain)
     def toString(self):
          print "Version:\t %d" % self.version
          print "Previous Hash\t %s" % hashStr(self.previousHash)
          print "Merkle Root\t %s" % hashStr(self.merkleHash)
          print "Time\t\t %s" % str(self.time)
          print "Difficulty\t %8x" % self.bits
          print "Nonce\t\t %s" % self.nonce

class Block:
     def __init__(self, blockchain):
          self.magicNum = uint4(blockchain)
          self.blocksize = uint4(blockchain)
          self.setHeader(blockchain)
          self.txCount = varint(blockchain)
          self.Txs = []

          for i in range(0, self.txCount):
               tx = Tx(blockchain)
               self.Txs.append(tx)

     def setHeader(self, blockchain):
          self.blockHeader = BlockHeader(blockchain)

     def toString(self):
          print ""
          print "Magic No: \t%8x" % self.magicNum
          print "Blocksize: \t", self.blocksize
          print ""
          print "#"*10 + " Block Header " + "#"*10
          self.blockHeader.toString()
          print
```

```
          print "##### Tx Count: %d" % self.txCount
          for t in self.Txs:
               t.toString()

class Tx:
     def __init__(self, blockchain):
          self.version = uint4(blockchain)
          self.inCount = varint(blockchain)
          self.inputs = []
          for i in range(0, self.inCount):
               input = txInput(blockchain)
               self.inputs.append(input)
          self.outCount = varint(blockchain)
          self.outputs = []
          if self.outCount > 0:
               for i in range(0, self.outCount):
                    output = txOutput(blockchain)
                    self.outputs.append(output)
          self.lockTime = uint4(blockchain)

     def toString(self):
          print ""
          print "="*10 + " New Transaction " + "="*10
          print "Tx Version:\t %d" % self.version
          print "Inputs:\t\t %d" % self.inCount
          for i in self.inputs:
               i.toString()

          print "Outputs:\t %d" % self.outCount
          for o in self.outputs:
               o.toString()
          print "Lock Time:\t %d" % self.lockTime


class txInput:
     def __init__(self, blockchain):
          self.prevhash = hash32(blockchain)
          self.txOutId = uint4(blockchain)
          self.scriptLen = varint(blockchain)
          self.scriptSig = blockchain.read(self.scriptLen)
          self.seqNo = uint4(blockchain)

     def toString(self):
          print "Previous Hash:\t %s" % hashStr(self.prevhash)
          print "Tx Out Index:\t %8x" % self.txOutId
          print "Script Length:\t %d" % self.scriptLen
          print "Script Sig:\t %s" % hashStr(self.scriptSig)
```

```
          print "Sequence:\t %8x" % self.seqNo

class txOutput:
     def __init__(self, blockchain):
          self.value = uint8(blockchain)
          self.scriptLen = varint(blockchain)
          self.pubkey = blockchain.read(self.scriptLen)

     def toString(self):
          print "Value:\t\t %d" % self.value
          print "Script Len:\t %d" % self.scriptLen
          print "Pubkey:\t\t %s" % hashStr(self.pubkey)
```

The structure of the classes and files mirrors the datatypes specified in the Bitcoin protocol. To modify this code for another block chain simply add or modify the class files to match the chosen block chains protocol.

blocktools.py creates the building blocks for the classes mapped in block.py. Using these two simple files all that's needed is providing some block data. Now all that's needed is a file a main method that can kick off the read process.

Once the block tools and class files have proven themselves in the parser.py unit test make a final '.py' file. Mine is 'sight.py'. Copy the following code:

```
#!/usr/bin/python
import sys
from blocktools import *
from block import Block, BlockHeader

def parse(blockchain):
        print 'print Parsing Block Chain'
        counter = 0
        while True:
                print counter
                block = Block(blockchain)
                block.toString()
                counter+=1

def main():
        if len(sys.argv) < 2:
                print 'Usage: blockparser.py filename'
        else:
                with open(sys.argv[1], 'rb') as blockchain:
                        parse(blockchain)



if __name__ == '__main__':
        main()
```

Running 'python sight.py [filename]' where filename is a valid block chain file will fill your console window with a wall of text as block information goes flying by.

The script will continue running until it reaches the end of the file. The wonderful thing about this code is it can be adapted to any Bitcoin-based block chain. For example, running sight.py against the included 1Mltc.dat file will parse the Litecoin block chain.

From the command line this data can be redirected to a file or you could write a quick database adapter and persist the information.

Summary

The previous chapters explained how block chains store information using Bitcoin as the example. After an explanation of the block and transaction data structures and giving instructions on how to trace bitcoin through the block chain ledger system was a guided code example for parsing block chains.

A repository complete with the code and example data used in this book is located at the following github address:

https://github.com/tenthirtyone/blocktools