Developing a Python library for Big Data processing in the Field of Cryptocurrencies.

An application for the cryptocurrency bitcoin

BURA MARCO

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science in Applied Economics & Data Analysis

School of Business Administration Department of Economics

Master of Science

“Applied Economics and Data Analysis”

September 2019
University of Patras, Department of Economics BURA MARCO

© 2019 – All rights reserved

Three-member Dissertation Committee
Research Supervisor:  Manolis Tzaggarakis  Assistant professor

Dissertation Committee
Member:
Ioannis Venetis  Associate Professor

Dissertation Committee
Member:
Konstantinos Kounetas  Assistant Professor

The present dissertation entitled

«Developing a Python library for Big Data processing in the Field of Cryptocurrencies. An application for the cryptocurrency Bitcoin.»

submitted by BURA MARCO, SID 1018168, in partial fulfillment of the requirements for the degree of Master of Science in «Applied Economics & Data Analysis» at the University of Patras.

I would like to dedicate my dissertation to the Department of Economics of University of Patras

Acknowledgments

I would like to express my sincere gratitude to Manolis Tzagarakis

Summary

In this thesis we used Python which is an interpreted, general-purpose and high-level programming language to create a library for Big Data processing in the Field of Cryptocurrencies. An application for the cryptocurrency Bitcoin. The data was in binary form that were abstracted from bitcoins blockchain, that is a growing list of records, called blocks, that are linked using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data (generally represented as a Merkle tree) and then make assumptions and descriptive statistics.

Σε αυτή τη διπλωματική εργασία χρησιμοποιήσαμε την Python που είναι μια ερμηνευμένη γλώσσα προγραμματισμού γενικής χρήσης και υψηλού επιπέδου για τη δημιουργία μιας βιβλιοθήκης για την επεξεργασία μεγάλων δεδομένων στον τομέα των κρυπτοσυχνοτήτων. Μία εφαρμογή για το κρυπτονόμισμα Bitcoin. Τα δεδομένα ήταν σε δυαδική μορφή, τα οποία αντλήθηκαν από το blockchain του bitcoin, δηλαδή μια αυξανόμενη λίστα αρχείων, που ονομάζονται μπλοκ, τα οποία συνδέονται με την κρυπτογραφία. Κάθε μπλοκ περιέχει μια κρυπτογραφική για τον κατακερματισμό του προηγούμενου μπλοκ, ένα χρονικό σήμα και δεδομένα συναλλαγής (που γενικά αντιπροσωπεύεται ως δέντρο του Merkle) και στη συνέχεια κάναμε υποθέσεις και περιγραφικά στατιστικά.
Contents

Chapter 1
Introduction
Bitcoin and Cryptocurrency technologies 5/9
Thesis Purpose and Contribution 9/10

Chapter 2
Theoretical Background
A brief overview of the System10/11
Cryptographic Properties11/12
Benefits of Bitcoin & Blockchain 12/14
Transaction Scheme 14/18
Proof of Work and Mining 18/19
Block Time and Rewards 19/21
Block Scheme 21/25
Blockchain and Big Data 25/30

Chapter 3
Methodology & Data
Data collection 31
Python packages 31/35
The Python Library 35/47
Blockchain statistics 47/51

Chapter 4
Code Development
The Code 52/63
The parsing results 64/81
Statistics for parsed data results 82/83

Chapter 5
Conclusion 84
References 85/86
Chapter 1

Bitcoin and Cryptocurrency technologies

There’s a lot of excitement about Bitcoin and cryptocurrencies. Optimists claim that Bitcoin will fundamentally alter payments, economics, and even politics around the world. Pessimists claim Bitcoin is inherently broken and will suffer an inevitable and spectacular collapse. Underlying these differing views is significant confusion about what Bitcoin is and how it works. We wrote this book to help cut through the hype and get to the core of what makes Bitcoin unique. To really understand what is special about Bitcoin, we need to understand how it works at a technical level. Bitcoin truly is a new technology and we can only get so far by explaining it through simple analogies to past technologies.

Bitcoin has hit headlines around the globe recently as the digital currency that could dethrone the traditional financial system and establish a new way of transacting with each other. With an all time high price of almost 20 thousand dollars and total market capitalization around $145,996,400,257 USD its existence as a financial product can no longer be ignored, while the technology behind it should provide with many research questions to be answered, in the field of both economics and computer science.

The path to Bitcoin is littered with the corpses of failed attempts. I’ve compiled a list of about a hundred cryptographic payment systems, both e-cash and credit card based technologies, that are notable in some way. Some are academic proposals that have been well cited while others are actual systems that were deployed and tested. Of all the names on this list, there’s probably only one that you recognize — PayPal. And PayPal survived only because it quickly pivoted away from its original idea of cryptographic payments on hand-held devices! There’s a lot to learn from this history. Where do the ideas in Bitcoin come from? Why do some technologies survive while many others die? What does it take for complex technical innovations to be successfully commercialized? If nothing else, this story will give you an appreciation
of how remarkable it is that we finally have a real, working payment mechanism that’s native to the Internet. Table 1: Notable electronic payment systems and proposals

Traditional financial arrangements

Back in time before there were governments, before there was currency, one system that worked for acquiring goods was barter. Let’s say Alice wants a tool and Bob wants medicine. If each of them happen to have what the other person needs, then they can swap and both satisfy their needs. On the other hand, let’s say Alice has food that she’s willing to trade for a tool, while Bob, who has a tool, doesn’t have any need for food. He wants medicine instead. Alice and Bob can’t trade with each other, but if there’s a third person, Carol, who has medicine that she’s willing to trade for food, then it becomes possible to arrange a three-way swap where everyone gets what they need. The drawback, of course, is coordination — arranging a group of people, whose needs and wants align, in the same place at the same time. Two systems emerged to solve coordination: credit and cash. Historians, anthropologists, and economists debate which of the two developed first, but that’s immaterial for our purposes. In a credit-based system, in the example above, Alice and Bob would be able to trade with each other. Bob would give Alice the tool and Bob gets a favor that’s owed to him. In other words, Alice has a debt that she needs to settle with Bob some time in the future. Alice’s material needs are now satisfied, but she has a debt that she’d like to cancel, so that’s her new “want”. If Alice encounters Carol in the future, Alice can trade her food for Carol’s medicine, then go back to Bob with the medicine and cancel the debt. On the other hand, in a cash-based system, Alice would buy the tool from Bob. Later, she might sell her food to Carol, and Carol can sell her medicine to Bob, completing the cycle. These trades can happen in any order, provided that the buyer in each transaction has cash on hand. In the end, of course, it’s as if no money ever changed hands. Neither system is clearly superior. A cash-based system needs to be “bootstrapped” with some initial allocation of cash, without which no trades can occur. A credit-based system doesn’t need bootstrapping, but the drawback is that anyone who’s owed a debt is taking on some risk. There’s a chance that the other person never comes back to settle the debt. Cash also allows us to be precise about how much something is worth. If you’re bartering, it’s hard to say if a tool is worth more than medicine or medicine is worth more than food. Cash lets us use numbers to talk about value. That’s why we use a blended system today — even when we’re using credit, we measure debt in the amount of cash it would take to settle it. These ideas come up in many contexts, especially online systems where users trade virtual goods of some kind. For example, peer-to-peer file-sharing networks must deal with the problem of “freeloaders,” that is, users who download files without sharing in turn. While swapping files might work, there is also the issue of coordination: finding the perfect person who has exactly the file you want and wants exactly the file you have. In projects like MojoNation and academic proposals like Karma, users get some initial allocation of virtual cash that they must spend to receive a file and earn when they send a copy of a file to another user. In both cases, one or more central servers help keep track of users’ balances and may offer exchange services between their internal currency and traditional currency. While MojoNation did not survive long enough to implement such an exchange, it became the intellectual ancestor of some protocols used today: BitTorrent and Tahoe-LAFS

Bitcoin is created from a plethora of concepts and technologies aiming to provide the basis for a digital money ecosystem available to anyone with access to the Internet. Its users have the ability to store and transmit value over a network accessible from a wide range of devices such as smart phones, laptops etc to perform all kinds of
transactions already done by traditional currencies, anonymously. Unlike traditional currencies however Bitcoin exists only in the "digital world" with no physical counterpart, while its major difference is the fact that there is no central authority/server maintaining records of transactions and accounts or governing its supply.

Regarding the commercial use of Bitcoin, is in its very premature stage with a tiny proportion of vendors accepting it as a form of payment. As of 2019, the number of offline merchants accepting Bitcoin is calculated at roughly 12 thousand while adding also the online vendors, the total number reaches around 64 thousand indicating reluctance for both merchants to accept as well as holders of the currency to pay with due to issues of high price volatility, speculation about Bitcoin price increases as well as the sometimes insufferably high transaction fees. The main problems a digital currency had to overcome in order to be considered secure enough to be utilized (although rarely), in commerce and everyday transfers of value, were that of authenticity of the funds, proof that the money can only be spend once (known as the "double spend" problem) and the certainty that only the designated recipient has access to the funds. The Bitcoin protocol combined elements from past attempts at a digital currency along with the developments in cryptography to create a completely decentralized electronic cash system that does not rely on a central authority for transaction validation and settlement or currency issuance.

Satoshi Nakamoto in his paper Bitcoin-A Peer-to-Peer Electronic Cash System in 2008, first proposed this idea of a decentralized payment system, proposing interesting solutions to the electronic currencies problems, that following the financial crisis of 2009 started to gain traction. Since then and after Nakamoto’s disengagement from the project on April 2011, developing and updating the code and network fell to the hands of volunteering programmers across the world. In the following years, and based on the open source software used for Bitcoin, alternative digital currencies were developed to improve upon various technical and practical aspects of bitcoins; however, despite the fact that it may be technologically inferior to various alternatives, Bitcoin remains relevant due to its first mover

Now let’s turn to cash. We compared cash and credit earlier, and noted that a cash system needs to be “bootstrapped,” but the benefit is that it avoids the possibility of a buyer defaulting on her debt. Cash offers two additional advantages. The first is better anonymity. Since your credit card is issued in your name, the bank can track all your spending. But when you pay in cash, the bank doesn’t come into the picture, and the other party doesn’t need to know who you are. Second, cash can enable offline transactions where there’s no need to phone home to a third party in order to get the transaction approved. Maybe later, they go to a third party like a bank to deposit the cash, but that’s much less of a hassle. Bitcoin doesn’t quite offer these two properties, but comes close enough to be useful. Bitcoin is not anonymous to the same level as cash is. You don’t need to use your real identity to pay in Bitcoin, but it’s possible that your transactions can be tied together based on the public ledger of transactions with clever algorithms, and then further linked to your identity if you’re not careful. We’ll get into the messy but fascinating details behind Bitcoin anonymity in Chapter 6. Bitcoin doesn’t work in a fully offline way either. The good news is it doesn’t require
a central server, instead relying on a peer-to-peer network which is resilient in the way that the Internet itself is. In Chapter 3 we’ll look at tricks like “green addresses” and micropayments which allow us to do offline payments in certain situations or under certain assumptions. The earliest ideas of applying cryptography to cash came from David Chaum in 1983. Let’s understand this through a physical analogy. Let’s say I start giving out pieces of paper that say: “The bearer of this note may redeem it for one dollar by presenting it to me” with my signature attached. If people trust that I’ll keep my promise and consider my signature unforgeable, they can pass around these pieces of paper just like banknotes. In fact, banknotes themselves got their start as promissory notes issued by commercial banks. It’s only in fairly recent history that governments stepped in to centralize the money supply and legally require banks to redeem notes. I can do the same thing electronically with digital signatures, but that runs into the annoying “double spending” problem — if you receive a piece of data representing a unit of virtual cash, you can make two (or more) copies of it and pass it on to different people. To stick with our analogy, let’s stretch it a little bit and assume that people can make perfect copies and we have no way to tell copies from the original. Can we solve double spending in this world? Here’s a possible solution: I put unique serial numbers into each note I give out. When you receive such a note from someone, you check my signature, but you also call me on the phone to ask if a note with that serial number has already been spent. Hopefully I’ll say no, in which case you accept the note. I’ll record the serial number as spent in my ledger, and if you try to spend that note, it won’t work because the recipient will call me and I’ll tell them the note has already been spent. What you’ll need to do instead is to periodically bring me all the notes you’ve received, and I’ll issue you the same number of new notes with fresh serial numbers. This works. It’s cumbersome in real life, but straightforward digitally provided I’ve set up a server to do the signing and record-keeping of serial numbers. The only problem is that this isn’t really cash any more, because it’s not anonymous — when I issue a note to you I can record the serial number along with your identity, and I can do the same when someone else later redeems it. That means I can keep track of all the places where you’re spending your money. This is where Chaum’s innovation comes in. He figured out to both keep the system anonymous and prevent double-spending by inventing the digital equivalent of the following procedure: when I issue a new note to you, you pick a serial number. You write it down on the piece of paper, but cover it so that I can’t see it. Then I’ll sign it, still unable to see the serial number. This is called a “blind signature” in cryptography. It’ll be in your interest to pick a long, random serial number to ensure that it will most likely be unique. I don’t have to worry that you’ll pick a serial number that’s already been picked — you can only shoot yourself in the foot by doing so and end up with a note that can’t be spent. This was the first serious digital cash proposal. It works, but it still requires a server run by a central authority, such as a bank, and for everyone to trust that entity. Moreover, every transaction needs the participation of this server to go through. If the server goes down temporarily, payments grind to a standstill. A few years later, in 1988, Chaum in collaboration with two other cryptographers Fiat and Naor proposed offline electronic cash. At first sight this might seem to be impossible: if you try to spend the same digital note or coin at two different shops, how can they possibly stop this unless they’re both connected to the same payment network or central entity? The clever idea is to stop worrying about preventing double-spending and focus on detecting it, after the fact, when the merchant re-connects to the bank server. After all, this is why you’re able to use your credit card on an airplane
even if there is no network connection up in the skies. The transaction processing happens later when the airline is able to re-connect to the network. If your card is denied, you’ll owe the airline (or your bank) money. If you think about it, quite a bit of traditional finance is based on the idea of detecting an error or loss, followed by attempting to recover the money or punish the perpetrator. If you write someone a personal check, they have no guarantee that the money is actually in your account, but they can come after you if the check bounces. Conceivably, if an offline electronic cash system were widely adopted, the legal system would come to recognize double spending as a crime. Chaum, Fiat, and Naor’s idea for detecting double spending was an intricate cryptographic dance. At a high level, what it achieved was this: every digital coin issued to you encodes your identity, but in such a way that no one except you, not even the bank, can decode it. Every time you spend your coin, the recipient will require you to decode a random subset of the encoding, and they’ll keep a record of this. This decoding isn’t enough to allow them to determine your identity. But if you ever double spend a coin, eventually both recipients will go to the bank to redeem their notes, and when they do this, the bank can put the two pieces of information together to decode your identity completely, with an overwhelmingly high probability. You might wonder if someone can frame you as a double spender in this system. Say you spend a coin with me, and then I turned around and tried to double-spend it (without redeeming it with the bank and getting a new coin with my identity encoded). This won’t work — the new recipient will ask me to decode a random subset, and this will almost certainly not be the same as the subset you decoded for me, so I won’t be able to comply with their decoding request.

• **Thesis Purpose and Contribution**

Bitcoin’s unique architecture allows for all data of every transaction ever performed on the network to be stored on a public ledger freely available for anyone to examine. The drawback however is that these data are not easy neither to de-code (as they are maintained in non-human readable format) nor to handle. For this reason special tools are required to access the vast amount of information stored in this public ledger, and since the research topic is rather recent the literature is very scarce on efficient tools able to undertake the task. The software we aim to create in this thesis, would be able to accurately and quickly decode the input data (Bitcoin transactions) and present them in human readable form. Although existing decoders (from now on called parsers) are available in a lot of programming languages, an implementation in python is yet to be created. This thesis aims to fill this research gap by creating a library for Python able to perform the transformation of the raw unstructured data to a human readable form.

Additionally we focus on testing the performance of this software implementation in python programming language, using a bottom-up approach starting from some basic utilities all the way to the full parsing procedure of the unstructured raw data from the Bitcoin ledger. The code development part is split into three levels, each improving from the last one by adding more capabilities, while also improving the inefficiencies identified. The main problems we encountered where mainly due to the large volume and unstructured nature of the data. We also had to take into account the structure of the information contained as it maintains a pre-established scheme.
In order to understand the demands of the software in terms of space, memory and speed we use existing python packages to collect performance and timing data for analysis. The timing data of these phases are analyzed to establish the influential variables increasing the parsing time, while also looking for nonlinearities indicating software problems. We find that the amount of parsing time is influenced both by the number of transactions and the number of inputs and outputs in those transactions. We also seem to be able to "capture" almost all the data variation as suggested by the almost perfect fit of the data.

The dissertation is split into 4 sections. In the first part of the thesis we present some background about the theory and operation of the Bitcoin ecosystem, and its contents. We intensely focus on the structural form and rules that maintain the sub-elements in working order, while also analyzing the details of each of those elements. As the point of the thesis is creating a software able to "decode" the Bitcoin’s raw data, mapping its scheme and understanding the rules it follows is essential. We then present some tools and techniques to handle the vast amount of the so called Big Data while also making the case about their usage in modern economic research. The second part of the thesis includes the Python packages we used during the development of our python code, as well as the steps, problems and different approaches we used to create it. Furthermore summary statistics relevant to the development and examination of the software are presented. For our analysis of the performance of the created code in the next section, gathered timing data were used, giving us a clear picture about the influential factors, and the existing problems. Finally we conclude by presenting the three different code versions in the appendices.

Chapter 2

Theoretical Background

• A brief overview of the System

Bitcoin is a complex scheme, and its implementation involves a combination of cryptography, distributed algorithms, and incentive driven behavior. In essence, it is an electronic token backed by no underlying commodity or sovereign currency, and is not a liability on any balance sheet. As a result it has no intrinsic value, other than the ability to perform transactions within its network. The value of Bitcoin is derived by its use for making payments in the Bitcoin system, and from the purpose of accruing gains from its possible appreciation (Anton Badev, Matthew Chen, 2014).

The disruptive innovation operating behind the scenes in Bitcoin is what is called the "Blockchain". Essentially, the Blockchain is a public ledger keeping track of all
past transactions in chronological order while continuously growing to include any new ones. As the system’s goal is decentralization and avoidance of a central authority/server, a copy of the Blockchain is freely distributed to every participant of the network. This creates the issue known in distributed computing as the "Byzantine Generals’ Problem (Lamport L. et al., 1982)" solved by Satoshi with another major innovation called the "Proof of Work" algorithm, which allows:

"The Byzantine Generals’ Problem refers to a situation in distributed computing where independent participants of a network try to agree over the state of the system, in this case the Blockchain, by exchanging information over an unreliable and potentially compromised network reach consensus over the valid current history of transactions. This algorithm secures the network against possible attacks while also providing the participants with economic incentives to maintain and update this ledger. The network participants that "work" to record transactions and secure the blockchain are called "miners" and are rewarded with newly minted bitcoin increasing thus their total supply, and alleviating deflationary pressure."

In straightforward terms, a blockchain is an advanced record. It’s an open record of bitcoin exchanges orchestrated in sequential request. It is a permissionless, appropriated database dependent on the bitcoin convention that keeps up a constantly developing rundown of value-based information records. It is disseminated so every member has the duplicate of the entire blockchain. The blockchain is shared between all bitcoin clients.

It is utilized to check the perpetual quality of bitcoin exchanges and to anticipate twofold spending. It is secure and changeless, and it’s likewise solidified against altering and modification, even by administrators of the information store’s hubs.

Each blockchain record is authorized cryptographically, and hosts run machines filling in as information store hubs. A blockchain is comprised of squares. A square is a record in the blockchain that contains and affirms many holding up exchanges.

Generally, at regular intervals, another square containing exchanges is annexed to the blockchain through mining. It is a record called the bitcoin blockchain sitting on a large number of PCs over the world, maybe even without anyone else PC at home. The document contains information pretty much all bitcoin exchanges — that is, the installment of bitcoins starting with one record then onto the next — that have ever occurred. This is frequently called a record and is like a bank record, which tracks installments.

• **Cryptographic Properties**

The only prerequisite for someone to transact using Bitcoin is to create a pair of digital keys. The Public and Private key pair, long existing topic in the cryptography literature, acts as the user’s unique identity in the network and through them he can receive and send funds. The public key is shared with anyone wanting to send you funds and acts as the recipient’s name in a check("Pay in the order of"), while the private key (known only to its owner), proves ownership of an amount and is required to authorize a transfer of funds to a third party. The cryptographic puzzle and digital signatures incorporated in a transaction are created using a scripting language unique to Bitcoin and take advantage of public key cryptography. Although the technical details of this language are besides the scope of this thesis, they require clarification as they are an essential part of the transaction validation process.
In detail the cryptographic puzzle created in a TxOut, places a spending condition, in other words "locks" the transaction amount so that only the owner of a specified (in the same script) Public Key may spent it. The "digital signature" script on the other hand, "unlocks" the output amount when presented with proof of ownership of that prespecified Public key, namely the corresponding Private key available only to the recipient. Further useful cryptographic features of a valid "digital signature" are that it gives a recipient reason to believe that the transaction was created by a known sender (authentication), that the sender cannot deny having sent the transaction (non-repudiation), and that the transaction details were not altered in transit (integrity).

The aim of this complicated process is to ensure that only the intended recipient has access to the funds, the immutability of the transactions details and the inability to deny its creation.

**Benefits of Bitcoin & Blockchain**

For bitcoin

- **Quick payments:** It’s amazingly simple and quick to make payments with bitcoin. The bitcoin exchanges are always up, even on vacations. Sending payments universally is likewise simple. There are no banks to make anybody hold up three business days, no additional expenses for making a global exchange, and no unique restrictions on the base or most extreme sum one can send.
- **Protection:** Installment can be made utilizing bitcoin without sharing any close to home data; one doesn’t have to join or share any card data. Indeed, it is even conceivable to send an installment without uncovering one’s personality, practically like you can with physical cash. In any case, observe that some exertion might be required to ensure security.
- **Irrelevant exchange expenses:** There are next to no or negligible exchange charges while making installments with bitcoin. There are no expenses to get bitcoins, and having numerous wallets enables you to control how enormous a charge to pay when spending. Most wallets have sensible default charges, and higher expenses can support quicker affirmation of your exchanges. Expenses are inconsequential to the sum moved, so it’s conceivable to send 100,000 bitcoins for a similar charge it expenses to send one bitcoin.
- **Secure:** Bitcoins are made and held electronically, yet there is no Visa number included that somebody can take as no one can charge you cash for your sake. The exchanges are made utilizing military-grade cryptography and are exceptionally classified. Bitcoin will give full access over your cash and a significant level of security against practically a wide range of false works which includes certain means [2].
- **Multisignature:** Bitcoin’s multisignature highlight permits organizations full command over their spending by permitting bitcoins to be spent just if a subset of a gathering of individuals approve the exchange.
From the image above taken from Stackexchange, bitcoin basically protects the users through pseudonym by public keys [3]. For multisignature events, the public key is basically broken down into how multiple keys that combine to approve the multisignature verification. Example on how to generate keys in Python is shown below:

```
import bitcoin

my_private_key = bitcoin.random_key()

print("Private Key: %s\n" % my_private_key)

print("Public Key: %s\n" % my_public_key)
```

The code above with other goodies can be downloaded here.

For blockchain:

- Blockchain is the innovation behind bitcoin. It works like a database for all bitcoin exchanges, and it keeps all records of bitcoin exchanges since the absolute first exchange. The underlying and most broadly known use of blockchain innovation was the open record of exchanges for bitcoin. Be that as it may, advanced monetary standards are by all account not the only use for blockchain innovation.
- Blockchain is a no-trust-based framework, which can be utilized to lead a wide range of exchanges, for example, computerized agreement marking. blockchain innovation can be utilized to make a lasting open straightforward record framework for arranging information on deals, putting away rights information by validating copyright enrollment, etc.

Bitcoin Mining

Bitcoin mining adds more bitcoins in the world and the miner is rewarded money as coins are discovered. Each bitcoin digger engages with the various ways mining coins through arithmetical calculations. Yet, as time goes on, the calculations get more complex, which takes longer for coins to be mined. This time length increases the costs to mine coins.
Anybody can take an interest in bitcoin mining by running a PC program. Notwithstanding running on conventional PCs, a few organizations have planned particular bitcoin mining equipment that can procedure coins and cutting the time to mine significantly more rapidly [4]. Bitcoins are getting increasingly costly due to increased competition from other miners. Numerous supercomputers over the globe are in rivalry to mine the next bitcoin. As the quantity of bitcoin diggers has expanded, it has turned out to be progressively tricky and overrated to begin mining new bitcoin [5].

The Blockchain consists of chunks of data called "blocks". The main contents of a block are its transactions. They are stored in the Blockchain with a past referencing scheme that links old with new ones together through an input-output system going back to the first ever created block called the "genesis block". However before being stored, the network enforces that they have gone through a validation process and are considered structurally correct. It is very important to ensure that each element the Blockchain is constructed from, maintains a predefined and agreed upon structure as to avoid conflicting versions of the same software existing and been shared on the network. As the parsing software we are trying to develop is based on these predefined schemes for each sub element of the Blockchain, we considered it very useful to present this structure and explain along the way their "behind the scenes operations".

• Transaction Scheme

We start our analysis of the Blockchain sub elements and their structure by the transactions scheme. Transactions are the most essential part of the Bitcoin ecosystem. We consider as transactions those data structures that encode the transfer of value between Bitcoin system participants (Antwnopoulos A., 2017). All transactions, after validation, are included in a public ledger called the "Blockchain", however the way they are preserved in this system is not easily understandable and consequently acquiring and processing these data, requires knowledge of the characteristics and components that define them. Their main building blocks are their Inputs and Outputs. Inputs roughly translate to the amount of value a given user currently holds, while outputs specify the receiver and the transfer amount. However since no central authority maintains users’ balances, inputs work by referencing past transactions outputs where the now payer was the receiver of funds, and thus has the ability to spend them. This procedure goes back the very first block, creating this back linked transaction chain. Contrary to traditional electronic payments where the id and balance of the sender and recipient are readily available, a Bitcoin transaction must reference the past to perform any new transfers of funds.

Below we present the exact structure of any given transaction:

• A list of outputs (TxOut), with information about the amount of BTC transferred in satoshis (smallest BTC denomination) and a cryptographic puzzle also known as "locking script" that sets the required conditions for the amount to be spent.
• A list of inputs (TxIn) that reference one or more previous unspent transaction outputs, a digital signature that satisfies the required conditions of that output making it spendable and for each input the variable Sequence Number.

• The variable Locktime which indicates the earliest time or earliest block when that transaction may be added to the public ledger (Blockchain).

Figure: Representation of Bitcoin transaction Scheme

Source: Blockchain.com
Notes: This figure depicts the sub elements of a transaction as well as the way that they interact.

In order to transfer funds between two parties, the sender creates a transaction where in the Inputs section a previous unspent output is referenced with funds greater to or equal to the desired amount to be transferred. He then "signs" the transaction by providing his digital signature, proving ownership of these funds. Multiple inputs may be aggregated to match the output amount, much like using five 1Euro coins to pay 5Euro worth of something.

In the Output section, the receiver is specified in the "locking script" along with the transaction amount. Again in case of multiple recipients, a list of outputs is created, with each one specifying the receiver and the transfer amount. After the transaction is properly created it is transmitted to the network, where a node (a Bitcoin network participant with access to the Internet) validates it based on certain rules and then adds it to the next block to be mined. The creator of a transaction is able to delay its submission to the next block, by providing the exact desired Block height or exact time for the transaction to be added, by using the variable of Locktime.

One important exemption to this input output system is a special kind of
transaction called a "coinbase". This is always the first transaction included in a block, has no inputs and only one output and is the reward obtained by a miner after updating the Blockchain. Its details are explained later.

• Different Transaction Types

Bitcoin’s scripting language, although not very sophisticated, enables the users to create different types of transactions ranging in details such as the number of recipients required to unlock the funds. Their objective is to offer more flexibility in the transfer of funds between the users. These transactions are very important to consider when creating a Bitcoin parser as each transaction type is represented uniquely in the data. These are:

• Pay-to-Pubkey
• Pay-to-PubkeyHash
• Pay-to-Script-Hash
• Multisignature Script

The main differences in the data representation of these transactions are found in their "locking and unlocking scripts". For example a Pay-to-Public-key transaction requires the locking script to include the Public Key instead of its hashed value as in the Pay-to-Public Key hash. A multisignature transaction on the other hand, is more complex and requires multiple "digital signatures" to unlock the funds. The software we try to create should take into account these different types and be able to distinguish one from the other.

• Validation Rules

Bitcoin Core checks each block of transactions it receives to ensure that everything in that block is fully valid—allowing it to trust the block without trusting the miner who created it.

This prevents miners from tricking Bitcoin Core users into accepting blocks that violate the 21 million bitcoin limit or which break other important rules.

Users of other wallets don’t get this level of security, so miners can trick them into accepting fabricated transactions or hijacked block chains.

Why take that risk if you don’t have to? Bitcoin Core provides the best possible security against dishonest miners along with additional security against other easier attacks (see below for details).

How Validation Protects Your Bitcoins

Bitcoin banks and lightweight (SPV) wallets put your bitcoins at increased risk of being stolen. That risk may be acceptable for small values of bitcoin on mobile wallets, but is it what you want for your real wallet?

<table>
<thead>
<tr>
<th>Attack</th>
<th>Bank Wallet</th>
<th>SPV Wallet</th>
<th>Bitcoin Core</th>
</tr>
</thead>
</table>


Direct theft
Bait and switch
Fabricated transactions
Chain hijacking
Transaction withholding
Chain rewrites

Note that although all programs—including Bitcoin Core—are vulnerable to chain rewrites, Bitcoin provides a defense mechanism: the more confirmations your transactions have, the safer you are. There is no known decentralized defense better than that.

Help Protect Decentralization
The bitcoin currency only works when people accept bitcoins in exchange for other valuable things. That means it’s the people accepting bitcoins who give it value and who get to decide how Bitcoin should work.

When you accept bitcoins, you have the power to enforce Bitcoin’s rules, such as preventing confiscation of any person’s bitcoins without access to that person’s private keys.

Unfortunately, many users outsource their enforcement power. This leaves Bitcoin’s decentralization in a weakened state where a handful of miners can collude with a handful of banks and free services to change Bitcoin’s rules for all those non-verifying users who outsourced their power.

Users of Banks
Trust bankers

Users of client lightweight wallets
Trust “free” services

Users of P2P lightweight wallets
Trust miners

Users of Bitcoin Core
Enforce the rules

Unlike other wallets, Bitcoin Core does enforce the rules—so if the miners and banks change the rules for their non-verifying users, those users will be unable to pay full validation Bitcoin Core users like you.

As long as there are many non-verifying users who want to be able to pay Bitcoin Core users, miners and others know they can’t effectively change Bitcoin’s rules.

But what if not enough non-verifying users care about paying Bitcoin Core users? Then it becomes easy for miners and banks to take control of Bitcoin, likely bringing to an end this 10 year experiment in decentralized currency.

If you think Bitcoin should remain decentralized, the best thing you can do is validate every payment you receive using your own personal full node such as Bitcoin Core.

We don’t know how many full validation users and business are needed, but it’s possible that for each person or business who validates their own transactions, Bitcoin can remain decentralized even if there are ten or a hundred other non-verifying users. If this is the case, your small contribution can have a large impact towards keeping Bitcoin decentralized.

Do You Validate Your Transactions?
Some people confuse supporting the network with helping to protect Bitcoin’s decentralization.

To improve your security and help protect decentralization, you must use a wallet that fully validates received transactions. There are three ways to do that with Bitcoin Core right now:

1. Use the built-in wallet’s graphical mode. If you request payment using the following screen in Bitcoin Core, your received transactions will be fully validated.
2. Use Bitcoin Core as a trusted peer for certain lightweight wallets. Learn more on the user interface page. If you use a secure connection to your personal trusted peer every time you use the wallet, your received transactions will be fully validated.

3. Use the built-in wallet’s CLI/API interface. This is meant for power users, businesses, and programmers. The user interface page provides an overview, the installation instructions can help you get started, and the RPC/REST documentation can help you find specific commands. If you’re using getnewaddress to create receiving addresses, your received transactions will be fully validated.

- **Blockchain**

The disruptive technology behind the operation of Bitcoin’s network is called the "Blockchain", and can be described as a database of all past transactions that is not maintained in a centralized server, rather is distributed peer-to-peer and secured by the network participants. Recorded transactions are organized in chunks named "blocks" and are after validation added to the "Blockchain" sequentially through a process called "mining". Each new block added contains a list of new transactions as well as a reference to the previous block in the chain, thus creating a back linked structure, where both new and past transactions are saved. This ever growing database cannot be modified and is secured by methods of cryptography. The major cryptographic tool that is used extensively in the Blockchain, is the hash function.

- **Proof of Work and Mining**

The way that participants ensure the security of the Bitcoin network is through the process called "mining". By mining we consider the provision of computational power to the network by individuals or groups called miners that "work" to ensure the validity and current state of the Blockchain and its contents. This sum of com-
Double spending refers to the problem where the same single digital token (a Bitcoin) can be spent more than once. This is possible because a digital token consists of a digital file that can be duplicated or falsified. As with counterfeit money, such double-spending leads to inflation by creating a new amount of fraudulent currency that did not previously exist. This devalues the currency relative to other monetary units, and diminishes user trust as well as the circulation and retention of the currency.

Computational power plays the role of the central authority in a traditional system (like a bank) thus eliminating the need for one (decentralization). Since the Blockchain is distributed and maintained by each and every node of the network there must always be consensus regarding its state. The protocol dictates that in order for an update to occur significant computational power must be expended. Then the nodes perform a kind of election to ensure that the valid blockchain is that with the most computational power. After providing with proof of their contribution miners are rewarded with a specified amount of BTC. The algorithm that the Bitcoin network uses as proof of work is the partial hash inversion. In non-cryptographic terms, miners working for the network have to solve an extremely hard mathematical problem where the most efficient solution is brute force. After the solution is found they relay it to the network and are rewarded with newly generated Bitcoin. This reward besides ensuring alignment of interests between miners and network security, also increases the supply of circulating currency alleviating deflationary pressure.

In more practical terms the mining system works like this: When a transaction is validated it is stored locally in the node that first received it, in a database called the unconfirmed transaction memory pool. After that a mining node creates a block by grouping unconfirmed transactions, and tries to solve a cryptographic problem as proposed by Bitcoin’s "Proof of Work" algorithm, competing with the other nodes to find the solution. The first to solve the hash inversion problem gets to add his block in the blockchain and obtains the "block reward" through the "coinbase" transaction along with any potential transaction fees. After the block is validated and added to the chain, all its transactions get a confirmation and the process starts anew. The time required to mine a block although in part random (brute force), also depends on the computational power of the node.

**Block Time and Rewards**

Block time, in the context of cryptocurrency, is a measure of the time it takes to produce a new block, or data file, in a blockchain network. It is the length of time it takes to validate the existence of a new batch of bitcoins. Theoretically, each network has its own defined block time. For instance, the Bitcoin network’s block time is around 10 minutes while the Ethereum network’s block time is about 20 seconds. The reality is not that precise.

**Understanding Block Time**

Block time is the time required to create the next block in a chain. It is essentially the amount of time it takes for a blockchain miner to find a solution to the hash, the random series of characters that is associated with the block.

**KEY TAKEAWAYS**

Block time is the length of time it takes to create a new block or file in a cryptocurrency chain.
A block is verified by bitcoin miners, who compete against each other to solve a mathematical problem that is attached to the block.

The successful bitcoin miner is rewarded in cryptocurrency.

However, the precise time to mine the next block is unknowable. The actual amount of time it takes for block generation varies depending on the difficulty of the hash.

Here are a few brief definitions for those trying to understand block time:

A block is a file that records a number of the most recent cryptocurrency transactions.

Each block contains a reference to the block that preceded it. (That's why it is theoretically impossible to alter cryptocurrency.)

Cryptocurrency "miners" race against each other to solve a mathematical puzzle embedded in the file. The winner is paid in cryptocurrency.

The solution to the puzzle triggers the currency's acceptance as a new block on the blockchain. The block has been validated. (The miner is rewarded with cryptocurrency.)

Thus, block time is the average time it takes for a miner to solve the mathematical puzzle and trigger the creation of a block on the blockchain.

Bitcoin mining is no longer a self-employment opportunity for mathematically talented computer hobbyists. Cryptocurrency mining is now dominated by mining "farms" with high-powered computer systems. About 50% of the world's bitcoin computing power is located in the Sichuan province of China, according to a report by the research firm Coinshare.

The block time for any type of cryptocurrency such as Bitcoin or Ethereum is an estimate of the time it takes to create a new block in the chain.

Other Definitions of Block Time
Block time has meanings that are entirely unrelated to cryptocurrency. For example:

Airlines refer to the estimated number of minutes that a flight will take from leaving the departure gate to stopping at the arrival gate as the flight's block time.

Block time is a theory that time is an unchanging four-dimensional block rather than a three-dimensional world of past, present, and future as we perceive it.

Figure: Bitcoins in circulation
Source:blockchain.info
Notes:This figure depicts the way Bitcoin’s supply diminishes and reaches a maximum by 2140.

• **Block Scheme**

As the Blockchain is a collection of blocks, each "stacked" on top of another, it is important to first describe the contents of a block, their meaning and how they are digitally represented in the raw data. The components of any given block can be decomposed in two main parts. The first contains information about the block itself called the Blockheader, while the remainder consists of transaction information. The following tables give a short description of the sub elements included in the two parts.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Size in Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MagicID</td>
<td>4</td>
<td>Identifier for the beginning of a Block</td>
</tr>
<tr>
<td>Block Length</td>
<td>4</td>
<td>The size of the Block</td>
</tr>
<tr>
<td>Version Number</td>
<td>4</td>
<td>Always set to 1</td>
</tr>
<tr>
<td>Previous Block Hash</td>
<td>32</td>
<td>32 byte hash of the previous Block</td>
</tr>
<tr>
<td>Merkle Root</td>
<td>32</td>
<td>A hash made from the Block’s transactions</td>
</tr>
<tr>
<td>Timestamp</td>
<td>4</td>
<td>Time the Block was created</td>
</tr>
</tbody>
</table>

Table 1. The first components of a block, called the Blockheader
The first part of a block called a "Blockheader" containing information about the block.

The MagicID is a series of 4 bytes, always "0xD9B4BEF9", that represent the beginning a new block.

Block length, as the name suggests, represents the size of the current block in bytes. The current maximum size is decided to be 1 megabyte. This fact greatly influences the speed that transactions are processed since a 1mb block per 10 minutes can only support around 7 transactions per second. Network congestion in times of increased demand is not rare, with repercussions both in terms of increased fees and significant delays in the confirmation time. This creates fears regarding the scalability of the Bitcoin network as there is no way in its current form to support global commerce of potentially two hundred thousand transactions per second.

The Version number of a block is always set to 1 and has no further significance.

The next 32 bytes part of the blockheader is the previous Block Hash. As we have already discussed the Blockchain is a back linked database where each new block sequentially added, contains some information about its predecessor. These information are in the form of a Block hash, a unique way to identify a Block that can be obtained by hashing twice the blockheader (in other words the block’s first 80 bytes), through the cryptographic algorithm SHA256. An important thing to consider when creating the parsing code is that the current block’s hash is not included in the block itself and must be externally created. The main utility of including the previous block hash in a new block is that it creates an immutable link all the way back to the first ever block (the "genesis block") that ensures the history and sequence of blocks. Furthermore a validating node trying to link a newly found block firstly checks its previous block hash field to ensure they match and proceed to extend the chain.

Merkle Root is the next 32 byte part of the blockheader. Its main function is to shortly summarize all transactions contained in a given block, and making the process of accessing a certain transaction in a block efficient. As the Merkle root is an optimization feature used by the nodes in the Network, it is not significant for the parsing process and can be omitted.

The next 4 bytes represent the Timestamp of a block or the time it was created. Although it the only time variable included in the blockchain, the information it offers can be misleading since the time of a block does not always coincide with that of the transactions it contains. Many fee-less transactions are added in a block maybe even hours after their creation thus creating a mismatch between the time they were created and the timestamp of the block that included them. The format of the timestamp is an integer representing a Unix timestamp. This integer is the time elapsed in seconds from 01-01-1970, a date known as "the Epoch". This is important to consider when creating the parser as proper adjustments need to be made.
The final 8 bytes of the blockheader are the target difficulty and the nonce (4 each), both items related to the mining procedure. These values are insignificant and are not required for parsing or transaction interpretation.

Following the 80 bytes of the blockheader are information about the transactions contained in the block. As already stated these information are not to exceed the 1mb block size limit and are structured as below: transferred.

Table 2. Elements of the second part of a block

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Size in Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Count</td>
<td>1,3,5 or 9</td>
<td>Number of Transactions Included</td>
</tr>
<tr>
<td>TransactionVersionNumber*</td>
<td>4</td>
<td>Expected to be 1</td>
</tr>
<tr>
<td>InputCount*</td>
<td>Variable</td>
<td>Number of Inputs for a Transaction</td>
</tr>
<tr>
<td>Transaction Hash**</td>
<td>32</td>
<td>Hash of the Referenced previous TxOut</td>
</tr>
<tr>
<td>TransactionIndex**</td>
<td>4</td>
<td>Index to a specific previous TxOut</td>
</tr>
<tr>
<td>InputScriptLength**</td>
<td>Variable</td>
<td>Length of the following Script data in bytes</td>
</tr>
<tr>
<td>RawInputScript**</td>
<td>InputScriptLength</td>
<td>Raw byte code data for the Input Script</td>
</tr>
<tr>
<td>SequenceNumber**</td>
<td>4</td>
<td>Always 0xFFFFFFFF</td>
</tr>
<tr>
<td>OutputCount*</td>
<td>Variable</td>
<td>Number of Outputs for a Transaction</td>
</tr>
<tr>
<td>OutputAmount**</td>
<td>8</td>
<td>The value transferred in &quot;Satoshis&quot;</td>
</tr>
<tr>
<td>OutputScriptLength**</td>
<td>Variable</td>
<td>Length of the following Script data in bytes</td>
</tr>
<tr>
<td>RawOutputScript**</td>
<td>OutputScriptLength</td>
<td>Raw byte code data for the Output Script</td>
</tr>
<tr>
<td>TransactionLockTime*</td>
<td>4</td>
<td>Always set to zero</td>
</tr>
</tbody>
</table>

Notes: The second part of a block, containing information about the transactions included.
• One star represents the elements that repeat for each transaction.
** Two stars represent the elements that repeat for each Input or Output in a Transaction.

The second part of the block begins with the actual count of the included transactions. Satoshi opted for a variable length integer to represent that number, meaning that the number of bytes used for the transaction count part, varies with the actual count between 1,3,5 or 9 bytes. This detail although small is crucial to take into account when creating the parser as it could result in errors in the byte interpretation.

The version number is always to be 1 or 2. It does not contain relevant information about the process besides being in every transaction.

The number of inputs as well as the number of outputs are included in each transaction and are also represented in the data with a variable integer as the
transaction count above. These numbers are important for the parser to know beforehand as it determines how many bytes of data need to be read for parsing inputs or outputs.

The previous transaction hash along with the transaction index that follow are the most crucial elements for the back-linked nature of the blockchain. What these 32+4 bytes do, is to uniquely identify a previous unspent output, so that a new transaction can point to it and use it as an input. The important thing to consider here is that these hashes are not stored in the Blockchain and must therefore be computed and stored externally by the parser. Furthermore coinbase transactions need to be taken into account as they have special characteristics in the way they are structured.

Following the previous transaction hash are the raw data of the Input Script. Varied in length, these data are essentially useless for the parsing process, as no information included in these scripts can help parse the blockchain transactions. However they need to be identified and then ignored. These are the data(along with the output script data) that the validation process uses to decide whether a transaction is correct or not. The input script usually contains the digital signature and public key of the sender that when "combined" through Bitcoin’s scripting language, unlock the referenced funds for spending. Ending the input part of a transaction is the sequence number always set to 0xffffffff. After that the output part follows starting with the output count explained earlier.

Each output contains the Amount to be transferred in 8 bytes. These bytes represent the Bitcoin amount in Satoshis(100 millionth of a Bitcoin) the smallest denomination of the currency.

Next comes the output script length which is the size in bytes of the following output script. The output script contains the recipient’s Public key and is very important for the whole parsing process that it is found and stored. Usually the forms that the public key is represented in the output script are:

- As a 67-byte long output script containing a full 65 byte public key address.
- As a 25-byte long output script containing a 20 byte hashed public key address.

Although there are errors in some scripts that a parser should account for, these are the ways that the majority uses to represent a recipient’s Public key.

Finally the last piece of information included in each transaction is the Lock-time that is always set to 0.

Figure 3: Representation of the Blockchain
• **Existing parsing tools**

The primary focus of this thesis is to create a parsing software, that accepts Blockchain’s raw data as input and returns them in a structured and human readable form. For this purpose we use the python programming language mainly because there are no existing python packages that fully interpret the Blockchain. Implementations of parsers exist for other programming environments such as R, OPHP and C++, while in PYTHON the only available Blockchain related packages are for querying external sources for Bitcoin related information such as price, volume etc. Some of the already implemented parsers have significant problems with either the time required, or the fact that they rely on external sources to acquire the data and not the Blockchain itself. For example an implementation on PHP programming language, found at https://github.com/bitcont/bitcoin, uses a blockchain.info API to retrieve information about addresses and transactions in a slow procedure that does not involve the public raw data of the Blockchain in any way.

In python there exist fully operating parses that work quite well and provide with all the information included in the Blockchain. One prime example is the software created by Github user "Ale-calve", in Python 3, with advances features like detecting output types, addresses in outputs, interpreting scripts etc. The github repository can be found at https://github.com/alecalve/python-bitcoin-blockchain-parser.

Another blockchain parser is available for C++ by Github user "bitcoinjs" at https://github.com/bitcoinjs/fast-dat-parser/blob/master/LICENSE, which uses the full system’s capabilities to export the blockchain data as fast as possible.

• **Blockchain and Big Data**

When dealing with Blockchain data, either as a node verifying transactions or as a researcher trying to "decode" and interpret them the major problem that hin- 

![Blockchain Diagram](source: bitcoin.org)
progress is their already vast but constantly increasing size and unstructured form, typical characteristics of Big Data. Although what constitutes as "Big Data" is not set in stone in the bibliography, Laney (2001), Amir Gandomi, Murtaza Haider (2014), Chen, Chiang, & Storey, (2012); Kwon, Lee, & Shin, (2014) suggest that contemporary research focuses on the three V’s as the common analytical framework of Big Data. These represent Volume, Velocity and Variety. A definition by the TechAmerica Foundation’s Federal Big Data Commission (2012) considers Big Data as "a term that describes large volumes of high velocity, complex and variable data that require advanced techniques and technologies to enable the capture, storage, distribution, management, and analysis of the information."

"Volume captures the aspect of magnitude. Data sizes that are considered to fall on the "Big" spectrum tend to reach terabytes (1024 gigabytes) or even petabytes (1024 terabytes). These sizes besides requiring great storage capacity also tend to be extremely tough to process efficiently with current tools and computational power. However time and ever-advancing technologies may deem these numbers obsolete in the near future.

Velocity captures the rate data are generated as well as the speed required for analysis. IBM estimations of daily data generation has been as high as 2.6 exabytes in 2016. High frequency data generation, as in the case of Bitcoin, is a Big Data characteristic that creates a need for real-time moderation and analysis of inflows. Velocity also describes the way different databases using streaming data interact with each other in real time. Bitcoin nodes are familiar with this notion as they maintain several different databases that interact in real-time to validate transactions and blocks.

Finally Variety refers to the structured and unstructured nature of the data either machine or human created. The endless channels businesses acquire data from nowadays, results in various data formats in need of special analysis and techniques. Variety is the characteristic best describing that fact, and is mainly about the different classes of available data and how to handle them.

The 3Vs has been in recent years extended to 5Vs (Bello-Orgaz et al., 2016), also including dimensions for Value (ability to process the data to extract valuable information or Big Data Analytics) as well as Veracity (a characteristic related with proper data governance and privacy concerns.)

Whatever the definition however, Big Data are sure to transform the current landscape of socio-economic policy and research (Einav and Levin, 2014; Varian, 2014) as well as the fields of economics, business management and decision-making.

- Big Data And Economics

For an empirical researcher in the field of economics, one of the major issues hindering progress (or outright stopping it), is the availability of data. This situation seems to slowly but gradually reverse as the Big data revolution has changed the way economists access data resources both in terms of quantity as well as quality. A prime example of a rich resource for economists, is the public sector, who through digitalization and the usage of Internet services is able to maintain highly detailed data on individuals and corporations, while minimizing sample selection and attrition.
The correct treatment of these data, through a proper Big Data architecture could yield invaluable information in the fields of public finance, labor economics, health and education. The works of Piketty and Saez on income and wealth shares using administrative tax data, of python. Chetty on regional differences in economic mobility, S. G. Rivkin on the performance of public school teachers and J. M. Abowd on differences in wages and productivity across similar firms, highlight the importance of cooperation between economists and government-created Big Data.

Another potential contributor to empirical research could be private companies. The unlimited ways modern companies acquire data not only provide an abundant resource of information but also open up the possibility for new areas of research where data were previously impossible to acquire. For example personal communications, social networks, search and information gathering, and geolocation data can provide economists tools to empirically assess the role of social connections and geographic proximity in shaping preferences, information transmission, consumer purchasing behavior, productivity, and job search. Private sector companies can also help to track economic activity by providing aggregate statistics of their business and customers ahead of the government official announcements as their useful complements, or as a main source of the country’s economic situation where public reports are either scarce or manipulated.

However the implementation of Big Data in economic research is not without its challenges. Most important of all are the issues of privacy and confidentiality affecting the ease of access to the available data. Governments and companies are reluctant to grant full access to sensitive information, requiring confidentiality agreements in the best scenario or denying access all together to all but the most prominent of research teams. Privacy is also a major obstacle in the free flow of data, as information regarding an individuals’ health, location, electricity use, or online activity may be easily obtainable, raising concerns about its potential mismanagement and exploitation.

• Big Data Tools and Techniques

Big Data is the huge amount of data that cannot be processed by making use of conventional methods of data processing. Due to extensive usage of many computing devices such as smart phones, laptops, wearable computing devices; the data processing over the internet has overreach more than the modern computers can handle. This paper provides a flying introduction to the Big data technology and its influence in the contemporary world. This paper addresses various properties and issues that need to be emphasized to present the full influence of big data. The tools used in big data technology are also explored in detail and Hadoop is the platform used in Big Data. It is an open-source framework that permits to store and process big data in a distributed environment across clusters of computers using simple programming models. Lastly, this paper also discuss about the applications of big data technology in detail. Keywords-Big Data, Hadoop, HDFS, Horton works, Map Reduce.
large and complex that none of the traditional data management tools can store it or process it efficiently. While the problem of working with data that exceeds the computing power or storage of a single computer is not new, the pervasiveness, scale, and value of this type of computing has greatly expanded in recent years. II. FEATURES OF BIG DATA Volume, Velocity, Value, Veracity and Variety (5 V’s) Figure 1. Features of Big Data Volume: Big data implies enormous volumes of data. It used to be employees created data. Now that data is generated by machines, networks and human interaction on systems like social media the volume of data to be analyzed is massive. Value: Value measures the usefulness of data in making decisions. User can run certain queries against the data stored and thus can deduct important results from the filter data obtained and rank it according to the dimensions they require. Variety: Variety refers to the many sources and types of data both structured and unstructured. We used to store data from sources like spreadsheets and databases. Now data comes in the form of emails, photos, videos, monitoring devices, PDFs, audio, etc. This variety of unstructured data creates problems for storage, mining and analyzing data. Velocity: Big Data Velocity deals with the pace at which data flows in from sources like business processes, machines, networks and human interaction with things like social media sites, mobile devices, etc. The flow of data is massive and continuous. This real-time data can help researchers and businesses make valuable decisions that provide strategic competitive advantages and ROI if you are able to handle the velocity. Veracity: Big Data Veracity refers to the biases, noise and abnormality in data. Veracity in data analysis is the biggest challenge when compares to things like volume and velocity. Research Article Volume 7 Issue No.6 International Journal of Engineering Science and Computing, June 2017 13645 http://ijesc.org/ Figure 2. Architecture of Big Data III. TOOLS USED IN BIG DATA 1. Apache Hadoop: framework that can effectively store large amount of data in a cluster. This framework runs in parallel on a cluster and has an ability to allow us to process data across all nodes. Hadoop Distributed File System (HDFS) is the storage system of Hadoop which splits big data and distribute across many nodes in a cluster. This also replicates data in a cluster thus providing high availability. 2. Microsoft HDInsight by Apache Hadoop which is available as a ser: It is a Big Data solution from Microsoft powered vice in the cloud. HDInsight uses Windows Azure Blob storage as the default file system. This also provides high availability with low cost. A.NoSQL: Stands for Not Only SQL. While the traditional SQL can be effectively used to handle large amount of structured data, we need NoSQL to handle unstructured Apache Hadoop is a java based free software data. NoSQL databases store unstructured data with no particular schema. Each row can have its own set of column values. NoSQL gives better performance in storing massive amount of data. There are many open-source NoSQL DBs available to analyse big Data. B.Hive: This is a distributed data management for Hadoop. This supports SQL-like query option HiveSQLIn short HSQL to access big data. This can be primarily used for Data mining purpose. This runs on top of Hadoop. C.Sqoop: This is a tool that connects Hadoop with various relational databases to transfer data. This can be effectively used to transfer structured data to Hadoop or Hive. D.PolyBase: This works on top of SQL Server 2012 Parallel Data Warehouse (PDW) and is used to access data stored in PDW. PDW is a data warehousing appliance built for processing any volume of relational data and provides integration with Hadoop allowing us to access non-relational data as well. E. Big data in EXCEL: As many people are comfortable in doing analysis in EXCEL, a popular tool from Microsoft, you can also connect data stored in Hadoop using EXCEL 2013. Horton works, which is primarily working in providing Enterprise Apache Hadoop, provides an option to access big data stored in their Hadoop platform using EXCEL 2013. You can use Power View feature of EXCEL 2013 to easily summarise the data. F.Facebook has developed and recently open-sourced its Query engine (SQL-on-Hadoop) named presto which is built to handle petabytes of data. Unlike Hive, Presto does not depend on MapReduce technique and can quickly retrieve data. IV. TECHNOLOGIES FOR BIG DATA HANDLING Big data technologies are important in
providing more accurate analysis, which may lead to more concrete decision-making resulting in greater operational efficiencies, cost reductions, and reduced risks for the business. To harness the power of big data, you would require an infrastructure that can manage and process huge volumes of structured and unstructured data in real-time and can protect data privacy and security. There are various technologies in the market from different vendors including Amazon, IBM, Microsoft, etc., to handle big data. While looking into the technologies that handle big data, we examine the following two classes of technology. A. Operational Big Data This includes systems like MongoDB that provide operational capabilities for real-time, interactive workloads where data is primarily captured and stored. No SQL Big Data systems are designed to take advantage of new cloud computing architectures that have emerged over the past decade to allow massive computations to be run inexpensively and efficiently. This makes operational big data workloads much easier to manage, cheaper, and faster to implement. B. Analytical Big Data This includes systems like Massively Parallel Processing (MPP) database systems and MapReduce that provide analytical capabilities for retrospective and complex analysis that may touch most or all of the data. MapReduce provides a new method of analyzing data that is complementary to the capabilities provided by SQL, and a system based on MapReduce that can be scaled up from single servers to thousands of high and low-end machines. The Big Data handling techniques and tools include Hadoop, Map Reduce, and Big Table. Out of these, Hadoop is one of the most widely used technologies. Hadoop Hadoop is an Apache open source framework which is written in java. High volumes of data, in any structure, are processed by Hadoop. Hadoop allows distributed storage and distributed processing for very large data sets. The main components of Hadoop are: 1. Hadoop distributed file system (HDFS) 2. MapReduce The architecture of Hadoop is shown in the figure. Hadoop has three layers. The two major layers are MapReduce and HDFS. International Journal of Engineering Science and Computing, June 2017 13646 http://ijesc.org/ Figure.3. Hadoop Architecture HDFS The Hadoop Distributed File System is a scalable and reliable distributed storage system that aggregates the storage of every node in a Hadoop cluster into a single global file system. HDFS stores individual files in large blocks, allowing it to efficiently store very large or numerous files across multiple machines and access individual chunks of data in parallel, without needing to read the entire file into a single computer’s memory. Reliability is achieved by replicating the data across multiple hosts, with each block of data being stored, by default, on three separate computers. If an individual node fails, the data remains available and an additional copy of any blocks it holds may be made on new machines to protect against future failures. Figure .4. Architecture for Hdfs This approach allows HDFS to dependably store massive amounts of data. For instance, in late 2012, the Apache Hadoop clusters at Yahoo had grown to hold over 350 petabytes (PB) of data across 40,000+ servers. Once data has been loaded into HDFS, we can begin to process it with MapReduce. Map Reduce Map Reduce is the programming model that allows Hadoop to efficiently process large amounts of data. MapReduce breaks large data processing problems into multiple steps, namely a set of Maps and Reduces, that can each be worked on at the same time (in parallel) on multiple computers. MapReduce is designed to work with of HDFS. Apache Hadoop automatically optimizes the execution of MapReduce programs so that a given Map or Reduce step is run on the HDFS node that contains locally the blocks of data required to complete the step. Map Reduce has proven itself in its ability to allow data processing problems that once required many hours to complete on very expensive computers to be written as programs that run in minutes on a handful of rather inexpensive machines. And, while MapReduce can require a shift in thinking on the part of developers, many problems not traditionally solved using the method are easily expressed as MapReduce programs. Figure.5. Architecture of Map Reduce Hadoop MapReduce is a software framework for easily writing applications which process big amounts of data in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner. The term MapReduce actually refers
to the following two different tasks that Hadoop programs perform:  

- The Map Task: This is the first task, which takes input data and converts it into a set of data, where individual elements are broken down into tuples (key/value pairs).
- The Reduce Task: This task takes the output from a map task as input and combines those data tuples into a smaller set of tuples. The reduce task is always performed after the map task. Typically both the input and the output are stored in a filesystem. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks. The MapReduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for resource management, tracking resource consumption/availability and scheduling the jobs component tasks on the slaves, monitoring them and reexecuting the failed tasks. The slaves Task Tracker execute the tasks as directed by the master and provide task-status information to the master periodically. The JobTracker is a single point of failure for the Hadoop MapReduce service which means if Job Tracker goes down, all running jobs are halted.

V. APPLICATIONS OF BIG DATA

As per the market strategy, companies who miss big data opportunities of today will miss the next frontier of innovation, competition, and productivity. Big Data tools and Technologies help the companies to interpret the huge amount of data very International Journal of Engineering Science and Computing, June 2017 13647 http://ijesc.org/ faster which helps to boost production efficiency and also to develop new data-driven products and services. So, Big data applications are creating a new era in every industry. The below figure shows examples of Big Data applications in different fields.

Chapter 3
Methodology & Data

- Data collection

The first thing to consider before starting developing any code, is the availability and structure of the data. Luckily the Blockchain data needed as input for the parsing software are freely available online at bitcoin.org. The procedure to get the proper files is to download and install the open source software Bitcoin Core v0.16 or "Satoshi Client" as it is called, available for all operating systems. By running this client the computer becomes a full-node participating in the Bitcoin network, with access to all available information that a node has including the current consensus of the valid Blockchain. Furthermore as we continue to operate in the network all transaction data will continuously update in real time. Since the aim of this dissertation is mainly to create a software to decode the raw data, any real time data inflows are not useful for the development of the code and are not used. Nevertheless the size of the complete Blockchain raw data is around 180Gb and continues to grow each day by an estimated 137 mb on average as of August of 2018. This figure may seem relatively small to be considered as Big Data, however after the proper code has been developed and we have "decoded" the raw data, the size of the results could be multiple times more spacious and complex.
The files of interest are found in the form of .dat files each 131 mb in size. They contain the raw data of the blockchain transactions from the genesis of bitcoin in 03/09/2009 up until 05/05/2017. These files contain binary data representation of the Blockchain contents with certain structure that is explained in the first part of the thesis.

- **Python packages**

Since parsing the Blockchain is a seriously demanding task we used already existing python tools to make the development faster and the code more efficient. The python packages we used as well as their purpose and contribution to the developing process are presented below.

**openssl**

The python package “openssl” is useful for encryption by applying the SHA256 cryptographic algorithm to any input data. As a primary component of the Blockchain, hash functions are greatly utilized in the code for the purposes of calculating a block’s or a transaction’s hash, since these information are not stored in the Blockchain and need external computation. The hashes are then used to establish the transaction database and locate each input’s corresponding output.

**RSQLite**

The RSQLite package is a complementary to DBI. RSQLite is the easiest way to use a database from R because the package itself contains SQLite; no external software is needed.

**wkb**

The next package used is called “wkb” and is used for converting a string’s hex representation, provided as input, into a raw vector. We use the function to decode into their raw form, transaction data that were serialized into a format named "Blob". A Binary Large OBject (BLOB) is a collection of binary data stored as a single entity in a database management system which in our case is SQL. The reason for this procedure is faster updating of the transaction database maintained in SQLite since
they are compressed and easy to export. After that we unserialize the entire database using this R package.

DBI

The package DBI or "Database Interface" provides a connection between python and an SQL database. It operates in conjunction with the RSQlite package described below to enable operations such as the connection to an existing relational database, the creation and execution of statements on that database in SQL, the extraction of result/outputs that are generated from the statements as well as information (meta-data) about the objects contained in the open database. The main objective of the package is to write the human readable contents of a decoded transaction into a SQL database, to avoid overloading the memory. As the number of transactions processed grew during development they occupied lots of RAM making the process slow and inefficient. More details will be presented in the next section.

Tic-toc

Tic-toc is a timing package that provides the functions tic and toc. In general, calls to "tic and toc" start the timer when the "tic" call is made and stop the timer when the "toc" call is made, recording the elapsed time between the calls. After that a simple message with the elapsed time is printed. Every time a block is completely parsed the function returns the elapsed time of the procedure. As the point of the dissertation is to gather time data to develop an efficient code this package is heavily used providing with further data for analysis.

data.table

Finally the data.table package is used to write the processed transactions into a CSV file. As the number of data to be written is large in size the fastest and most efficient package to handle the volume is data.table. It offers many options to better deal with the data and further help the analysis. We mainly use the "fwrite" function to export the data stored in the connected SQL database to a CSV file.

logging

This library provides us with a tool to "debug" our code. Debugging, in computer programming and engineering, is a multistep process that involves identifying a problem, isolating the source of the problem, and then either correcting the problem or determining a way to work around it. The final step of debugging is to test the correction or workaround and make sure it works. There are four level of logging defined in our application. Debug prints all information gathered by the logger, Info all but the information collected by debug and so on for Warn and Error.
Python Package-Bitcoinlib

One of the main objectives of the parser is to extract each user’s hashed Public key found at the output script. However, this requires to perform a special kind of encoding solely used in the Bitcoin protocol called Base 58 encoding.

Python blockchain / According to python.org, “Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together.”

Python is one of the most popular and powerful languages out there. Not only is it extremely beginner friendly, but it has also found applications in a lot of different areas as well. In fact, according to a survey by IEEE, Python happens to be the most popular language of 2019.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Language</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Python</td>
<td>100.0</td>
</tr>
<tr>
<td>2.</td>
<td>C</td>
<td>99.7</td>
</tr>
<tr>
<td>3.</td>
<td>Java</td>
<td>99.4</td>
</tr>
<tr>
<td>4.</td>
<td>C++</td>
<td>97.2</td>
</tr>
<tr>
<td>5.</td>
<td>C#</td>
<td>88.6</td>
</tr>
<tr>
<td>6.</td>
<td>R</td>
<td>88.1</td>
</tr>
<tr>
<td>7.</td>
<td>JavaScript</td>
<td>85.5</td>
</tr>
<tr>
<td>8.</td>
<td>PHP</td>
<td>81.4</td>
</tr>
<tr>
<td>9.</td>
<td>Go</td>
<td>76.1</td>
</tr>
<tr>
<td>10.</td>
<td>Swift</td>
<td>75.3</td>
</tr>
</tbody>
</table>

Origins of Python
Guido van Rossum, a Dutch programmer, created Python back in 1991. Python is based on a simple philosophy: Simplicity and Minimalism. One of the more notable ways that they incorporated simplicity into their language is by using white spaces to signify code blocks instead of curly brackets or keywords.

Let’s see what this means by checking out a simple “hello world” program.

```python
print(‘Hello, world!’)
```

Yup, that’s about it!

The name of the language is derived from the famous British comedians Monty Python. The core philosophy of the language was summarized in the document “The Zen of Python”:

- Beautiful is better than ugly
Explicit is better than implicit
Simple is better than complex
Complex is better than complicated
Readability counts

At its very core, Python is an object-oriented, high-level programming language with an extensive library. Let’s go through what each of those terms means.

Object-Oriented Programming is an object-oriented programming (OOPs) language as opposed to a process-oriented one. Process-oriented languages like C utilized programs where a list of instructions that acted on memory.

OOPs, on the other hand, allows for the creation of a blueprint called “class” from where one can generate objects which can interact with each other. These objects execute the program.

Now, there are four pillars to OOPs:

- Encapsulation
- Abstraction
- Polymorphism
- Inheritance

Encapsulation
Encapsulation is the idea of wrapping together data and function within one unit. The idea is to hide the initial state of the objects and to bind everything in a solid pack.

Abstraction
Abstraction means that a user can use the program without getting into the complications behind it.

Think of a car.
When you drive a car, all that you care about is putting your key in and maneuvering the vehicle in a way that you don’t hit anything else. You don’t care about how the engines work and how the ignition is burning your fuel.

Inheritance
Inheritance is one of the most important properties of OOPs.

Inheritance allows an object or a class to based upon another object or a class and retain some of its implementation. In most class-based object-oriented languages, an object created through inheritance acquires most of the properties and behaviors of the parent object.

Polymorphism
Polymorphism is the property by which an operator in the language can have more than one properties. The most famous example of this is “+”. Not only is it used as the mathematical addition operator, but it can also be used to combine two strings into one as well. This is called concatenation.

Eg. if we add two strings “Block” + “Geeks” the result will be “BlockGeeks”.

Low-Level vs High-Level
The level of the program is determined by its degree of interaction with the computer.
Low-Level Languages
Low-Level languages are machine dependent languages which directly interact with the computer. Remember that computers can only understand instructions in the form of 0’s and 1’s. This is why these languages utilize these signals via Binary notation to interact with the computer directly.

It is because of this very reason, that low-level languages are also extremely difficult to learn for beginners, which is why they are not as popular as high-level languages.

Assembly Language is an example of a low-level language.

High-Level Languages
On the other hand, High-Level Languages are the machine-independent programming languages, which are easy to write, read, edit and understand.

So, while they may not interact with the machine directly and need to go through a compiler, they are extremely versatile and beginner friendly.

Examples of high-level languages are Python, Java, .Net, Pascal, COBOL, C++, C, C# etc.

The Python Library
One of the greatest strengths of Python is its super extensive library. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming.

To make sure that internet facing applications are well-represented as well, many standard formats protocols like MIME and HTTP are supported in the library. It includes modules for creating graphical user interfaces, connecting to relational databases, generating pseudorandom numbers, arithmetic with arbitrary precision decimals, manipulating regular expressions, and unit testing.

As of March 2018, the Python Package Index (PyPI), the official repository for third-party Python software, contains over 130,000 packages with a wide range of functionality, including:

- Graphical user interfaces
- Web frameworks
- Multimedia
- Databases
- Networking
- Test frameworks
- Automation
- Web scraping[97]
- Documentation
- System administration
- Scientific computing
- Text processing
- Image processing

Firstly, you will need to go to python.org and download the latest version

Secondly, you will need to download Visual Studio Code which you can do right here.

Before we continue, let’s just configure our Visual Studio.

Once you are done installing it, you will see this screen.
Now, go on File -> Open

After that, you need to create a folder. Let’s call this folder “Hello World”.

Once that is done, you will need to go to Extensions and install “Python”.

Note: The last Icon on the left toolbar (the square-shaped thing) is the Extensions button.

Alright, now that that’s done, let’s start with some coding!!

Note: Before moving onto Visual Studio Code, we will be working on “Terminal” for Mac and “Command Prompt” for Windows

Starting Things Off
The first line on your Terminal should look something like this (we are using a Mac air):

X’s-MacBook-Air:~ Y$

Right next to that line write “python”

X’s-MacBook-Air:~ Y$ python

Once you press enter, you will see this:

Python 2.7.10 (default, Aug 17 2018, 17:41:52)

[ GCC 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.0.42) ] on darwin

Type “help”, “copyright”, “credits” or “license” for more information.
Basically, this means that you are now in and can start coding.

Let’s start off with simple numerical operators.

```
>>> 2+2
4
>>> 3-1
2
>>> 4*0
0
>>> 5/2
2
```

As you can see, pretty standard affair thus far. All the operations that we have done so far, follow the REPL formula.

R = Read
E = Evaluate
P = Print
L = Loop

2+2 is Read

The calculation of the result is Evaluate.

Printing of the result, i.e. 4 is Print

Loop basically means going back and starting all over again.

Let’s try to print “Hello World”.

```
>>> print('Hello World')
Hello World
```

See, pretty simple. You just use print() and put ‘Hello World’ inside in quotations. In the very next line itself, it will be printed. Also, note how we are not using a semi-colon to end our statements in python.

Data Types In Python
Alright, so now let’s talk about Data Types. In Python, there are 4 basic data types:

- Boolean
- Numbers (Integer and float)
- String

Boolean
Boolean values are a standard in programming languages. Boolean variables can only take in two values, True and False. They are really useful for condition-oriented coding such as if-else and loops.

Numbers
As with all programming languages, python utilizes both integer and float data types. Integers are basically non-decimal numbers and floats are decimal numbers.
So, 3 is an integer while 3.4 is a float number.

Before we go any further, there are two interesting functions that you should know about, float() and int(). The float() function turns its parameter (the data within its brackets) into a float number.

So, float(5) turns into 5.0.

Similarly, int(4.6) turns into 4.

Notice that the number isn’t rounding off in the traditional sense. It just shaves off the decimal part.

Alright, let’s have some fun with these functions.

Suppose we have a boolean variable: \( a = True \)

If we do \( int(a) \) then we will get 1.

Remember that True = 1 and False = 0.

Similarly, float(a) will get us 1.0.

String

Finally, we have String.

A string is a data type used in programming, such as an integer and floating point unit, but is used to represent text rather than numbers. It is comprised of a set of characters that can also contain spaces and numbers.

They are declared like this:

\( a = \text{‘Name’} \)

You can use both single quotes and double quotes to contain your string. However, remember that you can’t use them both on the same string, eg. ‘Hello’ will be an error.

Also, when you are using strings, be careful with apostrophes. Eg. ‘I’m a writer’ will be an error. You can correct this by putting a backslash before the apostrophe, like this: ‘I\’m a writer’

Alright, remember the int() and float() functions. Let’s see how those interact with string.

So, if \( n = \text{‘9’} \), and we want to add this with 1.

We can simply do: \( int(n) + 1 \) to get the output 10.

Conversely, if we do float(n) then the output will be 9.0.

Remember, during all this time, \( n \) still remains a string variable. We are merely using an instance of the string to do our mathematical operations. Let us show you that in the terminal.

However, if \( n \) had a floating value like ‘4.5’ and we try to use the int() function then we will get an error. This happens because the content of the string itself is a floating point variable.

Let’s have some more fun with strings.

Remember, that one of the properties of object-oriented programming is Polymorphism.

So, let’s try concatenation.

The “*” can also do some interesting stuff with string variables.
As you can see, we multiplied name by 10 and we got the value of name 10 times.

Operators
Operators are tools that you can use to manipulate a particular value or operands. The three most common operators that you will find in python are:

- Mathematical
- Boolean
- Logical

Mathematical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Role</th>
<th>Use</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>5+10</td>
<td>15</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>10-5</td>
<td>5</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>5*10</td>
<td>50</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>10/5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Gives the Quotient</td>
<td>11/5</td>
<td>2</td>
</tr>
<tr>
<td>**</td>
<td>Exponent</td>
<td>2**3</td>
<td>8</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
<td>11/3</td>
<td>1</td>
</tr>
</tbody>
</table>

Boolean Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Role</th>
<th>Use</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>5+10</td>
<td>15</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>10-5</td>
<td>5</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>5*10</td>
<td>50</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>10/5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Gives the Quotient</td>
<td>11/5</td>
<td>2</td>
</tr>
<tr>
<td>**</td>
<td>Exponent</td>
<td>2**3</td>
<td>8</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
<td>11/5</td>
<td>1</td>
</tr>
</tbody>
</table>

Boolean operators deal with values and operands and give boolean outputs i.e. True or False.
Logical Operators
The logical operator compares two conditions and gives a Boolean result.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Role</th>
<th>Use</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equivalent operator which checks to see if the two values or operands are equal</td>
<td>4 == 4</td>
<td>True</td>
</tr>
<tr>
<td>!=</td>
<td>Not Equal operator checks whether the values or the operands are equal or not</td>
<td>4 != 3</td>
<td>True</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Same as !=</td>
<td>4 &lt;&gt; 3</td>
<td>True</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>4 &gt; 3</td>
<td>True</td>
</tr>
<tr>
<td>&lt;</td>
<td>Lesser than</td>
<td>4 &lt; 3</td>
<td>False</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than equal to. The condition is true only if the value or operands is greater or equal than the other.</td>
<td>4 &gt;= 4</td>
<td>True</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than equal to. The condition is true only if the value or operand is less or equal than the other</td>
<td>3 &lt;= 4</td>
<td>True</td>
</tr>
</tbody>
</table>

Logical Operators
The logical operator compares two conditions and gives a Boolean result.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Role</th>
<th>Use</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>The and operator compares the two conditions and returns True ONLY if the two conditions are correct</td>
<td>4 &gt; 3 and 5 &gt; 6</td>
<td>False</td>
</tr>
<tr>
<td>or</td>
<td>The or operator compares the two conditions and returns True even if one of the conditions is true</td>
<td>4 &gt; 3 or 5 &gt; 6</td>
<td>True</td>
</tr>
</tbody>
</table>
Functions
Functions are the backbone of modern programming. So far, whatever programming we have done is pretty basic. However, programs can have 100s of lines of code which can get pretty hard to keep track of.

This is why we use programs, which basically acts like modules in that particular program.

Let’s take an example in order to understand the significance of functions.

Suppose you are writing a huge novel with no chapters at all. If you have described something before in the book, chances are, that if you have to cite it again, you will have to repeat some of the passages.

The problem with this is that it leads to redundancy, which is a waste of energy, money, and time.

Now, what if we actually segmented the book into several chapters. It brings in a lot of structure and neatness to the whole presentation of the book. So, if we want to refer back to something that we have mentioned earlier, we can simply let the user know which chapter number they can refer to.

In this analogy, the book is the program and the chapters are the functions.

So, how exactly do you define a function in python? Well, let’s take a look.

For these programs, we are going to use Visual Studio Code.

Just open your VSC and the folder that we made earlier. After doing that, click on this button to open a new file:

Name your file whatever you want, but be sure to end it with a “.py”. We are going to name our file “app.py”.

Ok, so let’s try out our first function!

```python
def greet:
    print('Hello')
```

```python
greet()
```

As you can see, we begin the function definition with the “def” keyword. After that, we have the name of the function followed by the brackets and a colon.

After that, unlike other languages, python doesn’t utilize curly brackets to define the function body. Instead, all that you need to do is to indent a little bit and add the body. VSC does this for you automatically.

After the function definition, you just simply call the function by typing “greet()”.

When you open the terminal, you will see something like this:
Right now just type in “python filename.py”.

In this case, our file name is “app” so we will type, python app.py.

```
Rajarshis-MacBook-Air:Hello World rajarshimitra$ python app.py
Hello
Rajarshis-MacBook-Air:Hello World rajarshimitra$
```

The moment you do that, as you can see above, the program will print “Hello”.

Now, let’s bring in some arguments into our function.

Arguments are the parameters that go inside the function. So, let’s go back to our greet function and add some extra elements:

```python
def greet(first_name, last_name):
    print('Hello ' + first_name + ' ' + last_name)

greet("Lionel", "Messi")
```

The entire program looks like this:

```
So, when you execute this program, the output will be:
```

```python
```
Conditional Statements
1 If-Elif-Else
Conditional statements are a staple in all programming languages, and python executes that with if-elif-else statements. Elif is an abbreviation of “else-if”. The syntax usually goes like this:

If condition 1:
Statement 1
elif condition 2:
Statement 2
else
Statement 3
Statement 4

So, what is happening here?
• If condition 1 is correct, then statement 1 will get executed and then the code will jump to statement 4 to execute it.
• If condition 2 is false then the code moves to condition 2. If condition 2 is correct then statement 2 gets executed, followed by statement 4.
• If none of the conditions are correct, then statement 3 gets executed by default followed by statement 4.

In the hypothetical code that we have given above, statement 4 does not belong to any of the conditions, which is why it gets executed by default at the end.

Ok, so now let’s check a simple program which uses only if.

In this program, we are simply checking if the number is less than 6 or not. If it is then we are going to print the result. Since the condition is met, the program outputs the statement in its terminal.

Ok, now let’s take this to another level and bring in some elifs. Check out the following piece of code:
What you have here is pretty much the same code as before.

Ok, so now, we are going to take it to the last level. Let’s introduce a final else statement.

So, we have three conditions here:

- First one checks if the number is greater than 6 or not
- The next on checks if the number is less than 4
- Finally, if all the conditions fail, then the else block gets activated and prints that the number is 5.

This is exactly what happens in out code as you can see in the terminal below

2 Tertiary Statements
Finally, we have tertiary statements. Check out this code:

```python
age = 21
message = "You are eligible" if age >= 18 else "You are not eligible"
print(message)
```

You see that statement?

"You are eligible" if age >= 18 else "You are not eligible"

This is called a tertiary statement, which follows this format: “Statement 1” if Condition else “Statement 2”

Basically, Statement 1 will be activated only if the condition is true, otherwise, Statement 2 is activated.

Loops
Finally, you have loops. Loops are an integral part of programming and are used when the repetition of a particular task is required. Python’s loops are actually pretty interesting and add a whole new dimension to your coding.

1 For Loops
For now, let’s check out the basic for loop:

```python
for num in range(3):
    print(num)
```

Ok, so what is happening here?
range(n) will give you numbers that go from 0 to n-1. So, in this case, it goes from 0 to 2. The num variable is going to assume the value of the range at each run through. So, if you print “num”, this is what you will be getting: “num” takes up all the values from 0 to n-1 in each iteration.

Oh and this is not just limited to numbers. We can make this loop with strings as well.

for x in “PYTHON”:
    print(x)

This will print the following:

```
Rajarshis-MacBook-Air:Hello World rajarshimitra$ python app.py
positive
positive
positive
Rajarshis-MacBook-Air:Hello World rajarshimitra$
```

As you can see, x takes up each and every character in the string.

2 While Loops
There is another kind of loop out there called the “while” loop.

```python
number = 100
while number >= 1:
    print(number)
    number //= 2
```

So, what is happening here?

We are running the loop until the number is greater than or equal to 1. When we do number //=2 we are basically doing number = number/2.

Let’s see what this prints out:
Playing Around with Loops
1 Combining Loops with If-Else
The first thing that we are going to do is to combine loops along with the if-else statements.

```python
x = 100
for number in range(3):
    if x > 1:
        print("positive")
    else:
        print("negative")
```

If we execute this, the output will be like this:

```bash
Rajarshis-MacBook-Air:Hello World rajarshimitra$ python app.py
positive
positive
positive
Rajarshis-MacBook-Air:Hello World rajarshimitra$
```

Ok, let’s change this up a bit. We are going to add something to this code.

```python
x = 100
for number in range(3):
    if x > 1:
        print("positive")
        break
    else:
        print("negative")
```

What’s the difference?

You see that one single word added to the code? The “break” keyword helps the compiler to break out of the code. Hence in the above code, “positive” is printed only once and then the compiler breaks out.

```bash
Rajarshis-MacBook-Air:Hello World rajarshimitra$ python app.py
positive
positive
positive
Rajarshis-MacBook-Air:Hello World rajarshimitra$
```
2 Nested Loops
Nested loops are basically loops within loops. Here check this out:

```python
for x in range(5):
    for y in range(3):
        print(x,y)
```

So what happened here?

We have a x loop and a y loop that runs inside it. Now, let’s print the result in the terminal and see what we get:

```
(0, 0)
(0, 1)
(0, 2)
(1, 0)
(1, 1)
(1, 2)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(4, 0)
(4, 1)
(4, 2)
```

- **Blockchain Statistics**

The main processing load affecting the speed and effectiveness of all already implemented parsing software is the ever growing number of transactions, especially in periods of high Bitcoin valuation, when the network is congested and transactional demands are high. It is important as a result to have an idea regarding the statistics behind the fluctuations in transactional movement as to expect relative slowdowns in the time the code operates. This will enable us to attribute any indication of slow parsing speed to an increased amount of transactions rather than a problem with the developed software. Below we present three graphs, visually highly correlated, that represent the size of the Blockchain in the span of time since its creation, the number of the Unspent Transaction Output Cache as well as the number of transactions included in a block. We gathered the data from a web API parsing the Blockchain in real time, found at www.blockchain.com.

Figure: The size growth of the Blockchain from 2010 to 2019
Source: Blockchain.com

Notes: This figure depicts the rapid growth of the Blockchain’s size. This size is tightly linked with the amount of transactions occurring on the network and seems to be steadily increasing by roughly 130 mb per day.

Figure: The per-block number of transactions
Source: Blockchain.com
Notes: This figure describes the number of transactions included in each block from 2009 until 2019. The recent spike in this number is probably caused by Bitcoin’s large media exposure and thus increased transactional demands from the network.

The changes in size of the Blockchain as seen in Figure 5 seem to be segregated into 3 different phases. The first phase from 2009 up until early 2012 is a phase of minimal growth, owning to the fact that no public awareness existed about the project.
prior to that time. As people came in and participated in the network, and transactions started to be created the rate of growth for the size of the Blockchain increased slowly but steadily, marking the second phase. Since 2016 the number of transactions per block occurring in the network dramatically increased creating a daily increase in blockchain size of about 137 mb. This increase remains and is expected to remain stable as the block size is fixed to be 1mb, hence the rate the blockchain size increases has a maximum cap.

Table . Total number of transactions at the years end.

<table>
<thead>
<tr>
<th>Year</th>
<th>Transactions</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>32687</td>
<td>-</td>
</tr>
<tr>
<td>2010</td>
<td>226688</td>
<td>594%</td>
</tr>
<tr>
<td>2011</td>
<td>2128883</td>
<td>839%</td>
</tr>
<tr>
<td>2012</td>
<td>10633942</td>
<td>400%</td>
</tr>
<tr>
<td>2013</td>
<td>30246540</td>
<td>184%</td>
</tr>
<tr>
<td>2014</td>
<td>55594094</td>
<td>84%</td>
</tr>
<tr>
<td>2015</td>
<td>101247924</td>
<td>82%</td>
</tr>
<tr>
<td>2016</td>
<td>184045240</td>
<td>82%</td>
</tr>
<tr>
<td>2017</td>
<td>288398401</td>
<td>57%</td>
</tr>
<tr>
<td>2018</td>
<td>344413453</td>
<td>81%</td>
</tr>
<tr>
<td>29/08/2019</td>
<td>449,171,859</td>
<td>88%</td>
</tr>
</tbody>
</table>

Source:Blockchain.com
Notes: This figure describes the number of total transactions at a years end as well as the percentage increase in transaction volume compared to the previous year.
Source: Blockchain.com

Notes: This figure describes the number of Unspent transaction outputs currently stored by the nodes.

The number of UTXO presented in Figure 6 (unspent transaction outputs) is a matter of great importance when implementing our parsing software. Since every new transaction is created using an old unspent output as a reference, the parsing software must at all times have in store and accessible a copy of all those outputs to match with the inputs. This can be done either by keeping them in the RAM or by storing them in the hard drive. As this cache grows, the time it takes for searching and updating grows, and is useful to have an idea when that happens and if its due to network increase movement or code inefficiencies.
Chapter 4
Code Development and parsed data results

THE CODE
import os
import datetime
import hashlib
import codecs

def HexToInt(s):
    t = ''
    if s == '':
        r = 0
    else:
        t = '0x' + s
        r = int(t,16)
    return r

def reverse(input):
    L = len(input)
    if (L % 2) != 0:
        return None
    else:
        Res = ''
        L = L // 2
        for i in range(L):
            T = input[i*2] + input[i*2+1]
            Res = T + Res
            T = ''
return (Res);

def merkle_root(lst): # https://gist.github.com/anonymous/7eb080a67398f648c1709e41890f8c44
    sha256d = lambda x: hashlib.sha256(hashlib.sha256(x).digest()).digest()
    hash_pair = lambda x, y: sha256d(x[::-1] + y[::-1])[::-1]
    if len(lst) == 1: return lst[0]
    if len(lst) % 2 == 1:
        lst.append(lst[-1])
    return merkle_root([hash_pair(x,y) for x, y in zip(*[iter(lst)]*2)])

dirA = 'C:\Users\MARCO\Desktop\BlockChain\'
    # Directory where blk*.dat files are stored
    #dirA = sys.argv[1]
dirB = 'C:\Users\MARCO\Desktop\parsedDATfiles\'
    # Directory where to save parsing results
    #dirA = sys.argv[2]

fList = os.listdir(dirA)
fList = [x for x in fList if (x.endswith('.dat') and x.startswith('blk'))]
fList.sort()

for i in fList:
    nameSrc = i
    nameRes = nameSrc.replace('.dat','.txt')
    resList = []
    a = 0
    t = dirA + nameSrc
    resList.append('Start ' + t + ' in ' + str(datetime.datetime.now()))
    print ('Start ' + t + ' in ' + str(datetime.datetime.now()))
    f = open(t,'rb')
    tmpHex = ''
    fSize = os.path.getsize(t)
    while f.tell() != fSize:
for j in range(4):
    b = f.read(1)
    b = b.encode('hex_codec').upper()
    tmpHex = b + tmpHex

for j in range(4):
    b = f.read(1)
    b = b.encode('hex').upper()
    tmpHex = b + tmpHex
resList.append('Block size = ' + tmpHex)

tmpHex = ''
tmpPos3 = f.tell()
while f.tell() != tmpPos3 + 80:
    b = f.read(1)
    b = b.encode('hex').upper()
    tmpHex = tmpHex + b

tmpHex = tmpHex.decode('hex')
tmpHex = hashlib.new('sha256', tmpHex).digest()
tmpHex = hashlib.new('sha256', tmpHex).digest()
tmpHex = tmpHex.encode('hex')
tmpHex = tmpHex.upper()

resList.append('SHA256 hash of the current block hash = ' + tmpHex)

f.seek(tmpPos3,0)

tmpHex = ''
for j in range(4):
    b = f.read(1)
    b = b.encode('hex').upper()
    tmpHex = b + tmpHex
resList.append('Version number = ' + tmpHex)

tmpHex = ''
for j in range(32):
    b = f.read(1)
    b = b.encode('hex').upper()
    tmpHex = b + tmpHex
resList.append('SHA256 hash of the previous block hash = ' + tmpHex)
tmpHex = ''
for j in range(32):
    b = f.read(1)
    b = b.encode('hex').upper()
    tmpHex = b + tmpHex
resList.append('MerkleRoot hash = ' + tmpHex)
MerkleRoot = tmpHex
tmpHex = ''
for j in range(4):
    b = f.read(1)
    b = b.encode('hex').upper()
    tmpHex = b + tmpHex
resList.append('Time stamp > ' + tmpHex)
tmpHex = ''
for j in range(4):
    b = f.read(1)
    b = b.encode('hex').upper()
    tmpHex = b + tmpHex
resList.append('Difficulty = ' + tmpHex)
tmpHex = ''
for j in range(4):
    b = f.read(1)
    b = b.encode('hex').upper()
    tmpHex = b + tmpHex
resList.append('Random number > ' + tmpHex)
tmpHex = ''
b = f.read(1)
bInt = int(b.encode('hex'),16)
c = 0
if bInt < 253:
    c = 1
tmpHex = b.encode('hex').upper()
if bInt == 253: c = 3
if bInt == 254: c = 5
if bInt == 255: c = 9
for j in range(1,c):
    b = f.read(1)
    b = b.encode('hex').upper()
    tmpHex = b + tmpHex
txCount = int(tmpHex,16)
resList.append('Transactions count = ' + str(txCount))
resList.append('')
tmpHex = ''
tmpPos1 = 0
tmpPos2 = 0
RawTX = ''
tx_hashes = []
for k in range(txCount):
    tmpPos1 = f.tell()
    for j in range(4):
        b = f.read(1)
        b = b.encode('hex').upper()
        tmpHex = b + tmpHex
    resList.append('transactionVersionNumber = ' + tmpHex)
    RawTX = reverse(tmpHex)
    tmpHex = ''
b = f.read(1)
tmpB = b.encode('hex').upper()
bInt = int(b.encode('hex'),16)
Witness = False
if bInt == 0:
    tmpB = ''
c = 0
c = f.read(1)
c = int(c.encode('hex'),16)
c = 0
c = f.read(1)
bInt = int(c.encode('hex'),16)
tmpB = c.encode('hex').upper()
Witness = True
resList.append('Witness activated >>')
c = 0
if bInt < 253:
    c = 1
    tmpHex = hex(bInt)[2:].upper().zfill(2)
    tmpB = ''
if bInt == 253: c = 3
if bInt == 254: c = 5
if bInt == 255: c = 9
for j in range(1,c):
    b = f.read(1)
b = b.encode('hex').upper()
tmpHex = b + tmpHex
inCount = int(tmpHex,16)
resList.append('Inputs count = ' + tmpHex)
tmpHex = tmpHex + tmpB
RawTX = RawTX + reverse(tmpHex)
tmpHex = ''
for m in range(inCount):
    for j in range(32):
        b = f.read(1)
        b = b.encode('hex').upper()
        tmpHex = b + tmpHex
    resList.append('TX from hash = ' + tmpHex)
    RawTX = RawTX + reverse(tmpHex)
    tmpHex = ''
    for j in range(4):
        b = f.read(1)
        b = b.encode('hex').upper()
        tmpHex = b + tmpHex
    resList.append('N output = ' + tmpHex)
    RawTX = RawTX + reverse(tmpHex)
    tmpHex = ''
    b = f.read(1)
    tmpB = b.encode('hex').upper()
    bInt = int(b.encode('hex'),16)
    c = 0
    if bInt < 253:
        c = 1
        tmpHex = b.encode('hex').upper()
        tmpB = ''
    if bInt == 253: c = 3
    if bInt == 254: c = 5
    if bInt == 255: c = 9
    for j in range(1,c):
        b = f.read(1)
        b = b.encode('hex').upper()
        tmpHex = b + tmpHex
    scriptLength = int(tmpHex,16)
tmpHex = tmpHex + tmpB
RawTX = RawTX + reverse(tmpHex)
tmpHex = "
for j in range(scriptLength):
    b = f.read(1)
    b = b.encode('hex').upper()
tmpHex = tmpHex + b
resList.append('Input script = ' + tmpHex)
RawTX = RawTX + tmpHex
    tmpHex = "
for j in range(4):
    b = f.read(1)
    b = b.encode('hex').upper()
tmpHex = tmpHex + b
resList.append('sequenceNumber = ' + tmpHex)
RawTX = RawTX + tmpHex
    tmpHex = "
b = f.read(1)
tmpB = b.encode('hex').upper()
bInt = int(b.encode('hex'),16)
c = 0
if bInt < 253:
    c = 1
tmpHex = b.encode('hex').upper()
tmpB = "
if bInt == 253: c = 3
if bInt == 254: c = 5
if bInt == 255: c = 9
for j in range(1,c):
    b = f.read(1)
    b = b.encode('hex').upper()
tmpHex = b + tmpHex
outputCount = int(tmpHex,16)
tmpHex = tmpHex + tmpB
resList.append('Outputs count = ' + str(outputCount))
RawTX = RawTX + reverse(tmpHex)
tmpHex = 
for m in range(outputCount):
    for j in range(8):
        b = f.read(1)
        b = b.encode('hex').upper()
        tmpHex = b + tmpHex
    Value = tmpHex
    RawTX = RawTX + reverse(tmpHex)
tmpHex = 
    b = f.read(1)
    tmpB = b.encode('hex').upper()
    bInt = int(b.encode('hex'),16)
    c = 0
    if bInt < 253:
        c = 1
        tmpHex = b.encode('hex').upper()
        tmpB = 
    if bInt == 253: c = 3
    if bInt == 254: c = 5
    if bInt == 255: c = 9
    for j in range(1,c):
        b = f.read(1)
        b = b.encode('hex').upper()
        tmpHex = b + tmpHex
    scriptLength = int(tmpHex,16)
tmpHex = tmpHex + tmpB
RawTX = RawTX + reverse(tmpHex)
tmpHex = "
for j in range(scriptLength):
    b = f.read(1)
    b = b.encode('hex').upper()
    tmpHex = tmpHex + b
resList.append('Value = ' + Value)
resList.append('Output script = ' + tmpHex)
RawTX = RawTX + tmpHex
tmpHex = "
if Witness == True:
    for m in range(inCount):
        tmpHex = "
        b = f.read(1)
        bInt = int(b.encode('hex'),16)
        c = 0
        if bInt < 253:
            c = 1
            tmpHex = b.encode('hex').upper()
        if bInt == 253: c = 3
        if bInt == 254: c = 5
        if bInt == 255: c = 9
        for j in range(1,c):
            b = f.read(1)
            b = b.encode('hex').upper()
            tmpHex = b + tmpHex
        WitnessLength = int(tmpHex,16)
        tmpHex = "
        for j in range(WitnessLength):
            tmpHex = "
            b = f.read(1)
bInt = int(b.encode('hex'), 16)
c = 0
if bInt < 253:
    c = 1
tmpHex = b.encode('hex').upper()
if bInt == 253: c = 3
if bInt == 254: c = 5
if bInt == 255: c = 9
for j in range(1, c):
    b = f.read(1)
    b = b.encode('hex').upper()
tmpHex = b + tmpHex
WitnessItemLength = int(tmpHex, 16)
tmpHex = 
for p in range(WitnessItemLength):
    b = f.read(1)
    b = b.encode('hex').upper()
tmpHex = b + tmpHex
    resList.append('Witness ' + str(m) + ' ' + str(j) + ' ' + str(WitnessItemLength) + ' ' + tmpHex)
tmpHex = 
Witness = False
for j in range(4):
    b = f.read(1)
    b = b.encode('hex').upper()
tmpHex = b + tmpHex
    resList.append('Lock time = ' + tmpHex)
RawTX = RawTX + reverse(tmpHex)
tmpHex = 
tmpHex = RawTX
tmpHex = tmpHex.decode('hex')
tmpHex = hashlib.new('sha256', tmpHex).digest()
tmpHex = hashlib.new('sha256', tmpHex).digest()
tmpHex = tmpHex.encode('hex')
tmpHex = tmpHex.upper()
tmpHex = reverse(tmpHex)
resList.append('TX hash = ' + tmpHex)
tx_hashes.append(tmpHex)
tmpHex = ''
resList.append('')
RawTX = ''
a += 1
tx_hashes = [h.decode('hex') for h in tx_hashes]
tmpHex = merkle_root(tx_hashes).encode('hex').upper()
if tmpHex != MerkleRoot:
    print ('Merkle roots does not match! >', MerkleRoot, tmpHex)
tmpHex = ''
f.close()
f = open(dirB + nameRes, 'w')
for j in resList:
    f.write(j + '
')
f.close()
nameSrc = ''
nameRes = ''
dirA = ''
dirB = ''
tmpC = ''
resList = []
fList = []
The parsing results and statistics
Start C:\Users\MARCO\Desktop\BlockChain\blk00003.dat in 2019-09-19 18:14:50.198000

Block size = 0000BFC0

SHA256 hash of the current block hash = 0000000000000A2E4FC7496EC6E4A2DBD901CBAF751AE0AFBFBC2E78128F57
Version number = 00000001
SHA256 hash of the previous block hash = 00000000000009686B6A7FC1F9D9AD62E96E16CF5D41086634AF1967651E32
MerkleRoot hash = 348BA7B42CABFBAEC09F5AFF0474B3028A8E1CEA99A78C4553CF3A6810717B0E
Time stamp > 4E2491B6
Difficulty = 1A0ABBCF
Random number > 8B1FBD37
Transactions count = 120

transactionVersionNumber = 00000001
Inputs count = 01
TX from hash = 0000000000000000000000000000000000000000000000000000000000000000
N output = FFFFFFFF
Input script = 04CFBB0A1A020F02
sequenceNumber = FFFFFFFF
Outputs count = 1
Value = 000000012ACC7825
Output script = 41047FD9556099FD6AD30566C6A2E9FE1F23B2B99725351C05525D4F687394C9F064BE1335727FF6D52BFD9909C0E375831FF57A7B357114D29D2324B31C64A251A1AC
Lock time = 00000000
TX hash = 70C882FEE9195791C57ED22A5AC0BAACCA047217AE20255C41B003B75C1F3932

transactionVersionNumber = 00000001
Inputs count = 01
TX from hash = E22260BBF44FFA70322DE8EE68C5D1A991D2149112311D21FBDE914B250D2D1
N output = 00000001

63
Input script = 493046022100F5BD198697390356BDAA1D9836B8D16C1B7D3376F7ED21F4F103C517F310E7B9022100 EE5470AE1BB3CED73004E019E4087B96ECE591D32C94512772D9F2A0ECFDFBBBD014104C0D01AD33 84FF9E47007C54C44E37EE742F3B010738D1E48B59333B40339CB9A8602D84B8C6DA1C78760226E841 765775FB868F25B273364E89C6BDF9A34FB04

sequenceNumber = FFFFFFFF

Outputs count = 2

Value = 000000012A05F200

Output script = 76A9145DF2DAD9710FAF6C164B74E431A3FB91441695A888AC

Value = 000000030104E580

Output script = 76A914889720281C69E6D89EE0607313B5A329EF755D88AC

Lock time = 00000000

TX hash = 16FE990175644914D05AB75A84F840BA885C55EAA0515E98B8187D8FC1829060

transactionVersionNumber = 00000001

Inputs count = 02

TX from hash = 4E85DE4813D0AC8DB620CD008ECDA28D2D390F9B0386CCF35698E0332D67072

N output = 00000001

Input script = 493046022100871EB014662AE28C158715F639B1247A16999E51110962F9524A3DFC7D4EC724022100D8 9EC47E23C6C02288CC67CD8A048B32E5E722099CA8F0A6B8B559AE31050C01410475AAAAC9172 EB5F0E613056E71F4BD47E08FB368165522E3879CD8D75D895DFD1FC4203F106CAE9B0364A6D3858 E7B790AC782EB42B073CEA198A29A5D71AA

sequenceNumber = FFFFFFFF

TX from hash = 8882C41F50D5C6FBBAA153F6D10CDD0CC1DA3A47135373846B8F522224D0C3EE5

N output = 00000001

Input script = 48304502207E797B8A5ACBE3F5A90C06FF0B797FE5C2B1161B5336C2B929A2A56BE6898398022100B1 D7A0C1F42A2DF8B5C2126976D01BB2B2747E12F36DB9121D2335D951E78ACA014104E9BE192DB16C B4D8CA6D9EAD7FBECA8A48AD340B9F43C7512A3C7288B1B9F44F5F602FE07196A92B4EFC5A4F52F 8C74C5F6F589C2CF1C33314969CF5EEDC708A

sequenceNumber = FFFFFFFF

Outputs count = 2

Value = 000000000F4240

Output script = 76A9141877FA9EBEAA907F42B7799B792C3249D9A25CD688AC

64
Value = 000000002F072F40
Output script = 76A9146075F110A1702D1AC92DA2D2CB448C551D87ED4688AC
Lock time = 00000000
TX hash = 5FAC2EBFDA05C8E60B8C91803E418AA0B455503035C92A223DEE93EC30DEADE

transactionVersionNumber = 00000001
Inputs count = 01
TX from hash = 6BFD3CFC17CC39FF92E0521F4F28E738635E739DA10480A0A1B96D2895AF2DBC
N output = 00000000
Input script =
4830450220648725B16A906BC53E8CF3E12A6F28F1864B25CF7D95BD3E09FCCEC6AD4686569022100B1A2DA2E02BF6B5143ABFE2AB68051A1D5A82855D3143A83922F3759D770C2F801
sequenceNumber = FFFFFFFF
Outputs count = 2
Value = 00000000064FF0D7
Output script = 76A914EBC6DE5123149E7F97935775BE159B58AA4BC37588AC
Value = 0000000124101100
Output script = 76A914C848DF5576A40BD50C4BB263ACE51E1FC50CDA2988AC
Lock time = 00000000
TX hash = 895EBA294BE000A8590A40C7B9661CA5274264158AEC4D65930089A7D062F90A

transactionVersionNumber = 00000001
Inputs count = 01
TX from hash = 76F3DF5EB73EE11787A8F0023F7119DD0B8DCC27BFB5A4B3389B8A7936D095FE
N output = 00000000
Input script =
48304502202DCD9983CE6BAC2A23840191B4BC59ADD6B86520E601F02CB5F0FD62A85C64B022100B0CD26DDC3BDEDE535028BB434EED9E33799837EF566CECF014104EAAECEC787816DB07147919763CD68356FE4028FCD197F2FD06BABB144ACF6A4C9D561534A059359A299FE0FD058EE11D7B4036FFE13810C2FD9BA9754B39F6C4
sequenceNumber = FFFFFFFF
Outputs count = 2
Value = 0000000052A64FCF

65
Output script = 76A9140A5EEF3D7561C945BBBD72B30B6A34D012A7189C588AC
Value = 0000000003E56653
Output script = 76A914AA8CF0ECFEBD6AB03D4FD896E8037F37492640CC88AC
Lock time = 00000000
TX hash = 3026745B20920B6B12FA0790B1419443F3FC3D5CCDF2E301F6A86D47771FB768

transactionVersionNumber = 00000001
Inputs count = 01
TX from hash = 9DE6531733DCEFA55CC8FB2DB68775E9CA8F19444C110F4B509B1661C0A3EA138
N output = 00000000

Input script =
483045022100D2E77E65443B595B483D4A3FD74D2E63AB4DF9CA72E11F63935B900A788BAAB02207
10E95C7B887CD074BA21E3708BBB2A73495A2607134B7ED4708769FCFCA39E1014104B9766CCDBF0
5187D7BC709EC0F5EAFD57349BA47C1DFF97801A6CC9BC8F68F1C7B514A6431035DB0454C8C7C8
2B310872883417177DCC6EE01C8B14E9250A9E
sequenceNumber = FFFFFFFF
Outputs count = 2
Value = 0000000010A99664D
Output script = 76A91481E7657AF317A9EDBC0E4F92EE4863F9E76DD56E88AC
Value = 00000000052D0A21
Output script = 76A91417714125EFC99A09D29DA6A1EC28AA6E9E9B617188AC
Lock time = 00000000
TX hash = 95D5D60B916D8E2FE6BACD92A8253A14E7FA435F6AE0BA055AE474237498C0EA

transactionVersionNumber = 00000001
Inputs count = 1D
TX from hash = C4AB04FCA1FF3E3E50FD66A77477BE47E8433AE3BD5B20059474305E38C33201
N output = 0000005D

Input script =
483045022100D2E77E65443B595B483D4A3FD74D2E63AB4DF9CA72E11F63935B900A788BAAB02207
10E95C7B887CD074BA21E3708BBB2A73495A2607134B7ED4708769FCFCA39E1014104B9766CCDBF0
5187D7BC709EC0F5EAFD57349BA47C1DFF97801A6CC9BC8F68F1C7B514A6431035DB0454C8C7C8
2B310872883417177DCC6EE01C8B14E9250A9E
sequenceNumber = FFFFFFFF
TX from hash = 43BCC0805F4AE645AF54E0E24587C2259DE9DF56B5099FB6D9A330AA1E6DC01
N output = 00000063
Input script = 473044022051C47089187D000C5CA93D72B1F3F50BD88C44A51035838CD0D596E6535B13302201FB7B55AEA77DC350614619BADCA6A4ADFC4168B44180ADE139CE7DD2AF2CC060014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DA3A86AD799FA6AFCADF84059C4E9436392F35E463D2B6D5BF3C78EC
sequenceNumber = FFFFFFFF

TX from hash = 87240ED77A759B0C78C9B7B5B088FF07564CB74DA7EE08FCE27BF18B8074C0A
N output = 00000062
Input script = 483045022100F8D7233C170C4E954C9B624575628EE85ECEC72BE901B582A2AC1327790E92022100C608D529045E41FF065D41E059CE52507B4E88D390E61DD75EF2CCC257B744A2014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DA3A86AD799FA6AFCADF84059C4E9436392F35E463D2B6D5BF3C78EC
sequenceNumber = FFFFFFFF

TX from hash = 78522E07556329E535A6BD767EEC54DF29E65F5C72B0747686DA4D2D26012910
N output = 0000005B
Input script = 493046022100E4E222A6C65769EE6E1FF8310DFE0A4DFE74CD971E0091F85E90DA2E2358F3A022100E471C74588041364C72DAD11472AC2FD425F24300840FE435FC405B12FA56FC014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DA3A86AD799FA6AFCADF84059C4E9436392F35E463D2B6D5BF3C78EC
sequenceNumber = FFFFFFFF

TX from hash = 36F02E84295ED646A29AA02A560E76CE00365C2B4C1A856F501666D53F3904E19
N output = 00000056
Input script = 493046022100EF9FDD246B4CDAF8FBBBDFC35784DC5C08EC9A22AAE3C2A08A714D9EB24C9B12F022100B0CD124CC5BF09DAB4C3E263CF7CC7F3DF25873CAE98D6C76C392F6EEFFA50F014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DA3A86AD799FA6AFCADF84059C4E9436392F35E463D2B6D5BF3C78EC
sequenceNumber = FFFFFFFF

TX from hash = C6035B0A9A8507BDB9B8993AE944F3FB46DF4D61FBAA78ACC984318C4362B1F
N output = 00000050
Input script = 473044022072A1A4A6B8C05B357D8853BB55D0348871C8C5F09C6D0663DE08028AA0DE1D560220528352F218E2EDCCFA7E3A6CA83D9191DB52EC48B69768C1A07A12F561253931014104BE455D248C2607
sequenceNumber = FFFFFFFF
TX from hash = 3BB961CD87644B170DAECE93FF6C81C68664BC791366B93E1D8F9FA36926E220
N output = 00000054

Input script =
493046022100EDBF0F327A8734653BBE71A60968254490769E80C0A714E5905EDD57350ED45022100F
2145DAEAA8F7137E714342464AFB4917706555DC7E8EDA3B0E2E07FABCB38F7014104BE455D248C2
6075F5A625FE392084999E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCAD
F84059C4E9436392F35E463D2B6D5BF3C78EC

sequenceNumber = FFFFFFFF
TX from hash = FB578D65AC6CBD5AE7D7C66C68D833CD2875CC95E4BDB185EB7297D955F8B351
N output = 0000005B

Input script =
483045022030A7CC854AF9984DB4C56B76882D9C776A664D34F46497E40679AD0EDAFE59D022100B
AC0BB38F6D34E5D046DA0B32A85A8A7A45ED6A9DEB87DA2DDED9056674FB4D014104BE455D248C2
6075F5A625FE392084999E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCAD
F84059C4E9436392F35E463D2B6D5BF3C78EC

sequenceNumber = FFFFFFFF
TX from hash = 2F7B83C7B276A045C14F76851B4495A6062A14D2A8222FCB066136B9E37AF851
N output = 00000044

Input script =
4830450221009C0E0EFFF9567EC198566CE586390FCF7C01A3CBD399FE1D1EAC1140483DD2711022051
8C70C0B92286F13A0FDFB5101E43F638A9772FBCD1782903C3C9823BB4FCAD014104BE455D248C260
75F5A625FE392084999E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCAD
F84059C4E9436392F35E463D2B6D5BF3C78EC

sequenceNumber = FFFFFFFF
TX from hash = 662B3470C4AB6B283A5B5F0A2D5D5ED5A4337749F8B435CA127252F9CDD558C5B
N output = 00000045

Input script =
49304602210089F461914C4BE8FFABACF7AC86DF402275CC3BC65B5AF549F0D845738CC49BF7FE02210
0DF4A5413060D66D4E05886F9A63CD37991D88FA88ADD51315F1ED13BF482E3EC014104BE455D248C2
6075F5A625FE392084999E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCAD
F84059C4E9436392F35E463D2B6D5BF3C78EC

sequenceNumber = FFFFFFFF
TX from hash = 24D3F2FC45D0AD6445579C34F750D0B5646268FE9C2023AA681B8A7FEFFE246D
N output = 00000000
Input script = 483045022100A8FA022DEFC5A5F38CD04CCB21047B47234EDDA30E79A3FDE28E973F54FC679E0220 24F381B4E3A3ACCEE19216DED5032AA2C5E301431A21D57F9A7C8E71414D1EC00141047320BF6A188 4E868F447EE534BDA3F5EAC1EE8A6CA55D2CE21E32D41977CF72ACECF38032DE7AE1D9BB6DCF13 848295291DFD3BC59E6750033915F3A20363A28
sequenceNumber = FFFFFFFF
TX from hash = 67451BBC2387D4A314B285BFDDA679872994C9BA746D9D60D4296E0EA1727
N output = 0000005C

Input script = 493046022100B4C79FA0F0AF048DB5FC053A31D0F2B7F4BCE651A2A869F7813F63BE635DD42002210 09D0CEC5C5BA254EB299344765F50BC0762FACCD6C2AAD611B8BC27FD32AA24A014104BE455D24 8C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCADF84059C4E9436392F35E463D2B6D5BF3C78EC
sequenceNumber = FFFFFFFF
TX from hash = BC0DD562EECD56D06C9D03B266B7E5B00365771069A6D1C0F8A34B4209E9B29
N output = 00000059

Input script = 48304502210099FE8C31AF28BE9E6EAC803A9C4805599E9806DAD2CFBC44A7BDF10A281527602205 09FE7D494563AF6668F2F9B709012F76AAAD6E6FE3A66055C7E87207F6A6BBA014104BE455D248C26 075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCADF84059C4E9436392F35E463D2B6D5BF3C78EC
sequenceNumber = FFFFFFFF
TX from hash = B5AFE0BA157D22E8EA3D4ACE95E0D28198DA1C4B8045E555E75838E667D7378
N output = 0000008B

Input script = 4830450220593128C3C3FCC0B4F2B18A824FE0A80C3A67817A94516798D4DB6521C48CA9635022100D5 B461E8A3542F4B28E1A665DE46224A1AAC7059D9909643A676126B032AC9E9C014104BE455D248C260 75F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCADF84059C4E9436392F35E463D2B6D5BF3C78EC
sequenceNumber = FFFFFFFF
TX from hash = D886817A95A4163FCA263A3C652F35EF428835F4921F5ECC6A2E8166BA136888
N output = 00000041

Input script = 473044022073DCC81813C295BD935DEFBC3E29F64A35462635B8930D9488F776C97B0C481B02202DE4 6D60950EDE13889C63281FFE3810CFCFB1503FFDEC7B54E7F9DD0D6FB6ED014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCADF84059C4E9436392F35E463D2B6D5BF3C78EC
sequenceNumber = FFFFFFFF
TX from hash = 5C47C8FFE0EF755A5E97B656DCC5594685504ADCA86CCD870193EC947737E91
N output = 0000007E

Input script =
483045022100B26440D4B1CA3707957645ECD73DCF7BB6B338E6C08206D92B1127BF0FF821F702206A2A0AB9C0C758602F7BD568DB6DC9B582A0B0BB4DF1E480179A7C33C8E5F5DA014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCDD125E5358734367DDA3A86AD799FA6AFCADDF84059C4E9436392F35E463D2B65BF3C78EC
sequenceNumber = FFFFFFFF

TX from hash = 91235E6F61E5DF673A9ABDAFA38538B46BC53CBD60F1D5C5D2C09F6C9B26C295
N output = 0000005C

Input script =
47304402202E84F45AC25CA212252DF9473796D1A5F3C1E951945AC896470D049C16F503D102203068179BFF199972C8CD9E1C701F268DFE6944A828B2E6946F67F77AC9172188014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCDD125E5358734367DDA3A86AD799FA6AFCADDF84059C4E9436392F35E463D2B65BF3C78EC
sequenceNumber = FFFFFFFF

TX from hash = 009E08FAEE624FB8B2DC6537B300AC0CB062864DB415B301F89E11D1182C5F96
N output = 00000048

Input script =
48304502202D574B991CC0339F8311A8B8F4900C239EFD67346DFA9C67B5BFAA1AA83401C3022100A7378749ECB8EEED96F11930F510EBD5D4F78AE387B9F1781843B1300209690014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCDD125E5358734367DDA3A86AD799FA6AFCADDF84059C4E9436392F35E463D2B65BF3C78EC
sequenceNumber = FFFFFFFF

TX from hash = A2C03B3DBC3B9DD6C286D07712AC65C40FAC678EFE8672C20A4D66927FDB396
N output = 0000005F

Input script =
483045022100BF6742DFB832B583108156C73ABB07117E5FF6B795DCCD2FF1378972F1606E02E02204B60B1D4C8DD58BC5D0BAEC09F90AA0B9A713FC61EC710B8E42007AD5B9A1318014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCDD125E5358734367DDA3A86AD799FA6AFCADDF84059C4E9436392F35E463D2B65BF3C78EC
sequenceNumber = FFFFFFFF

TX from hash = 8FDE15390642601F837F1BCF277D29A92BDA0EA2E0192B707D8E5891E43390A8
N output = 00000062

Input script =
4730440220326350F77E0BDF9D89CA53F38D86D54620778256923C4F67D1293217958EE9F502207D3364B9A703C8D8202CF1AF72421D805E035DEC7126FE659DEB576FD4EDA708014104BE455D248C26075F5
A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCAFDF84059C4E9436392F35E463D2B6D5BF3C78EC

sequenceNumber = FFFFFFFF
TX from hash = E019351315B1A71F80FC04A04508C61F87AB1F59C129740326ABA0F94E0E91A9
N output = 00000059

Input script =
47304402200DEFC93A0BD35B8E66A3B14D60CCF38E06606F4C6BC06A3F851A48A13852B58102205201E3E0D18E6653A33AFF2545B25B9FD28691F75E1A43962EC0417752C6E48E3014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCAFDF84059C4E9436392F35E463D2B6D5BF3C78EC

sequenceNumber = FFFFFFFF
TX from hash = 2434667D8BE9367A12892AD896F399DA3CB14647F5EE17E349103080516B58AC
N output = 0000005B

Input script =
483045022100D2084DA1C25695B4A2DD5777EBC45271E73255FD4CE508E2415C433A3777895F02204553D21A5D9C57E6813CAF412FA62D05CA197EDED357A1154488A684FBE3E3C28014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCAFDF84059C4E9436392F35E463D2B6D5BF3C78EC

sequenceNumber = FFFFFFFF
TX from hash = 47F948C028B13E68680B0749BE569EFCFC3D3290A311436935FD4393C612B7
N output = 00000084

Input script =
4730440220430243DBE5356E57B9E7270C9DF06B7B0C0A724A458C3065363BFE8766EF40A002201474986222156A77CF0F93EC1E49CB21BF096680AC226E8B5936072373A3C5B1014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCAFDF84059C4E9436392F35E463D2B6D5BF3C78EC

sequenceNumber = FFFFFFFF
TX from hash = 778BAD8FEB9E2D292628C201A34E469239F122B0CA5C6E3693178DF77CE121C1
N output = 0000004B

Input script =
473044022020474CD80F7FOE9111CC98FDEAACC045356336A8115968A8D51DB15338EE21ED02202198F6F6A65E2888991CF0A4BE7D319A5CE4FBE70F2ADD522D77E11042E2C5014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCAFDF84059C4E9436392F35E463D2B6D5BF3C78EC

sequenceNumber = FFFFFFFF
TX from hash = 2F5E2BE34EFA26696769F9293872A8464BC2FCCDC5C2905BDD94148DF864BCF
N output = 0000001F

71
Input script = 48304502205F897E56CD23B8D1B55B2F693DAB01D659F01EA738E04D24E5F7154E503D1EFE022100E3E9DEF2296FC3C58BE26CF36D7DED4C9A20C59112E4100706461FA889DC65BF014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCADF84059C4E9436392F35E463D2B6DBF3C78EC

sequenceNumber = FFFFFFFF

TX from hash = 0F5B334910257FC4ED1BB3535C76255620E4122469AAFE92C525185A156E9CDA
N output = 0000005F

Input script = 483045022100B5156E64EC25DF4FE220965ECF9DFF36ADA835E6F87EC3D54F880BFC72ED68A5022046B77F5DB8846F84A96A69947AE635F3E3B15FFABA868BDF1C3C6E691AEFBF327014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCADF84059C4E9436392F35E463D2B6DBF3C78EC

sequenceNumber = FFFFFFFF

TX from hash = C2676E038F2B077E1C9C5EA8D9827C83D276DF055E533530D52F54D85F7098D
N output = 00000084

Input script = 483045022100FFD85C03C9B453DF500B5A7A95DB75019C9BC584231A43BC7A0EB019FBEF7FA02201D724BFEFCD2DD39F4787D08F5DB24226565B7346398F1D84370DC2A380CE177014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCADF84059C4E9436392F35E463D2B6DBF3C78EC

sequenceNumber = FFFFFFFF

TX from hash = 52E6C8E529135D81B2D455B52B69E4496A0AF5F98C8A78AD8A854AACC6DE639
N output = 00000001

Input script = 493046022100B91BCFA3AC414DD9D60CAE7345BD6514CCA14167C75FDEBDA696E76FAB28B00221008990962C5DEDDBBB592942D488BF3BD343A6061455052115FCA21B7AA6C4E00ED014104433F476A4AA6F6FBB4B841B6B0E25D1836FD6549AF3D0F6268098BED8E998B33B4E1755AE2CC50CAC794763C874C235FB8A89C158BF6A1E221F64D02217C770E

sequenceNumber = FFFFFFFF

TX from hash = C20DD4EBCAAC417766E5B8FB97BD6CC4092946F89D323806FAB82467D9008A65
N output = 00000044

Input script = 473044022101ACC9BC9272F875C59104AD402D4ED407BB01D48EB05AAADF8228387256A57F5022002B0D491A7215B3CF8521E505A63E31C0A77E144620A5C0228D2E00F8E7E682014104BE455D248C26075F5A625FE39208499E9D03D61912963C94FC5DC9CCD125E5358734367DDA3A86AD799FA6AFCADF84059C4E9436392F35E463D2B6DBF3C78EC

sequenceNumber = FFFFFFFF
Outputs count = 2
Value = 00000000000F4240
Output script = 76A91430366B5FD9D475FF21C28036F322C01EB0F5291588AC
Value = 00000000B2D05E00
Output script = 76A914F4586253DCC0B6BD30E077195EC7A3A5B5126F2F88AC
Lock time = 00000000
TX hash = 3F86E49D7E9A251B2DFEBB63B30E23F44BEFE6ACA1B786B94F10E91078B6F3

transactionVersionNumber = 00000001
Inputs count = 01
TX from hash = A0E2F56D02732D84CFA858B7EBB94CA6AED225D22B68F6CB8D13D0A546151B58
N output = 00000063
Input script = 4930460221001EF6F1A769A45A02F96A5866E5B1B8C94DFDACCCS4F872FC68C823E9FD4E19C022109A53D7E00BF209E99456D4BFCC9B11287DF3BCFE515784B7B05AF8AD9C756201410426D5F4D
64A4B1C042B535D5B2BD33066A46776D87F92A71CDFAC11DD0F483AFE4F2209D6173A1537F
CAE12DC227859CFA719001769676E27D0B3D81C57
sequenceNumber = FFFFFFFF
Outputs count = 1
Value = 0000000005F5E100
Output script = 76A914E68B4D13C7F4AD092FF41CDE4945DA01CE676C3D88AC
Lock time = 00000000
TX hash = 7A58A219AA57F68D42586060E6D342CA18E031EC8EF9FF192E7ABD624323FF7B

transactionVersionNumber = 00000001
Inputs count = 01
TX from hash = 09C87F2D61043BC50295E70AA72D7B9A63C41E01AE95F493D76A57A416A84FBD
N output = 00000000
Input script = 473044022059AA6D8AAA04FB09AED28783A63007201B20A314620D5DBCB297399D923E4BAE022011D81917E0662EE57845A9EA6F7A782B90CEFE3BE4ABCDF2287196EB8B75A58C10141040741DED4625B7
A8F808B9480A2075823A281AC610A16D4365AF161EC62296D4B586CFC3C8D3E949172EB133E62A518
9AD7213CE5D8E0D7222922D707EA41F477
sequenceNumber = FFFFFFFF
Outputs count = 2
Value = 00000005CC0C7AA5
Output script = 76A914CBFEE178FC068332DB14DA94D7C8538D5800A22588AC
Value = 00000000001E8480
Output script = 76A914445C234DF5D1DCC5CC5ADDF49F9FF808380B990488AC
Lock time = 00000000
TX hash = 5CB0ED4C62327281117817216A37E1F1225C318A30AA707F825A1D622784B502

transactionVersionNumber = 00000001
Inputs count = 01
TX from hash = DAC4039C4394542C4E096A0930293C0919B65070C7252C39211C35A379E89411
N output = 00000000
Input script = 4930460221008B1B1042175DCFF41C9206E963350179B23FF359FD89762BB7E56E65A340490602210081C71A349D44C92A4707404116FDF4786D4C508FCE421D1D18DD6F4FEE7EF93D0141048A5C7E427392C95C4CB41ABFA0DEE1F39485FFBD2DAD1D8EE2BA8E9144C7271596009C08E1820FC037578F9DE2C3CBCD81C593601A59EBB4D2E76C26DD7E82FC
sequenceNumber = FFFFFFFF
Outputs count = 2
Value = 00000005CDC5EABE
Output script = 76A914E3DD14AA5F227866D699133788E1398F71DB430288AC
Value = 000000000030A32C0
Output script = 76A9145BA19D07D6C7A5112CFF674EB1229AB3999B23ED88AC
Lock time = 00000000
TX hash = BF6F4E0965B500A43F0354FDBAFB7C6F5EE3190B7D23D6234459AFB0D640B1547

transactionVersionNumber = 00000001
 Inputs count = 01
TX from hash = 73864428E2CE4947F6E883568E4F5B44E352E506D96F9D241CB5DBA95580901F
N output = 00000000
Input script = 4830450221008EFFCC9274F81DBA72C7F274CC6EA97AE9C4C56C759F9F61F7A71150053AC5A02205DDD796FF948D77DA9363C0284F491ACDBAE9B258D8EA886B85E7E573A2EB973014104F8F026B9A8

74
sequenceNumber = FFFFFFFF
Outputs count = 2
Value = 00000000000F4240
Output script = 76A914CAD8FCC74F2ED8D25280CE10CC8777CB92BA3A4288AC
Value = 000000000848F8C0
Output script = 76A9141111DADEA0CA295BC7F0E4859B348508B10588F888AC
Lock time = 00000000
TX hash = 04A8201DA858724B0771DBD287EE021F6392228C39BE4094D8003444E28DD84A9

transactionVersionNumber = 00000001
Inputs count = 02
TX from hash = 70664E50340BD202B033005417B644046185AF310B506A7AB6AA138754F24EF8
N output = 00000000
Input script =
483045022076556E145D3F8A792E79AE80F47103B1022A4BEF59451510B5120121F46DD59022100BBC
DD371A79C37FBA08B6506C5FC8D4183982A1763F086D5748F24B03E65D51C0141048A8ABB439EB6D
1D2DA076296BA05A69CA2F7033666DAE195C9C612EC2C91430D16998E74BA331237AB2ADC74DE93
DB1E1EAF60E35A1EBDCCA880DE857F041B4
sequenceNumber = FFFFFFFF
TX from hash = 213CB924A65BFAC4B3D817FAB6ED3F32F5DD20383358BB415F0E97D5DCF97054
N output = 00000000
Input script =
483045022010A960407E1BB8EC562D0A323CC505E5FE1B02A246C127C7FE58A0B8C6D9E7B11022100A
B24D3C56D78B54405BC43DEC7A037AFED131DD4DFD8E9A3A1B2F2AAD51292200140464436601F8
5ECF15C440E331F9297CF4AF90EC23BDABDC16C2D827F9C98DD39901EDBBEEAE89B90813FEFF6B
8F60E32D0372C1EDE5FF8AFB4EDC3E3B2FB773E8
sequenceNumber = FFFFFFFF
Outputs count = 2
Value = 0000000000B663902
Output script = 76A914E8FF65895E203058AF1177273688108A83A1D69A88AC
Value = 00000000ACCF1E18
Output script = 76A914CF41704F993890E2E5B959895B1011E0E18673CF88AC
Lock time = 00000000
TX hash = BDF5F23F89191DDE0A49D7149E960AB2488C088222514724DCA04C4F69A0C2A  

transactionVersionNumber = 00000001
Inputs count = 01
TX from hash = 0A51A63E349F68BA38E0AF7BA56537D24EA473D603D0085511B53E1112032601
N output = 00000000
Input script =
4730440220401F1FF6046DA2E5B403A77CB498A6BA42EFF21D137C9290085B10C83F58F069022004FB5
DEE1EE1C1A83E6B504AB8CCAAE6C403C5DDB34554527492CC70143400350141047C42CE531BBDF8F
6FFBCC3428647FCDD4748855BD1FC656AE0AF71E91678E947B899A457A87E3CC70D43BB9B16EE123
55F46C7F85C013AA227AB069F92F192B0
sequenceNumber = FFFFFFFF
Outputs count = 2
Value = 000000034B6E3A26
Output script = 76A91466201C73D219719CDAACB4A645E5EFE446AEC95188AC
Value = 0000000006146580
Output script = 76A9141D71A68B2BFC1AF6842744794A460A532E9112ED88AC
Lock time = 00000000
TX hash = 5D23BAB0D3BDC3CD3E5C97AC0DF92FC53377008DBD8235B5D1987084968907E8  

transactionVersionNumber = 00000001
Inputs count = 03
TX from hash = 4F6049DDF43888AF6BCAFECFA92446BDA1E6C4FD53C7D0937E346117FBC7152E
N output = 00000000
Input script =
4930460221008993582C5A4C55CF1E0B18FB9AE35256AC7E3FCC23BB58A798B1F0D5D0E25C2A02210
0BF3B02DA1F74D6D4A3F94917FD14153F9BF97E112804E2A6E588EC237A5A7620140461E1847A91C
A81CDABD2F6C388CA4C550066FC50978F26A7D2396FF537F9371C1CE228257F1629D50A443CE7FB95
2271946D02E588466C6C844BEBF7C0A2AAAE
sequenceNumber = FFFFFFFF
TX from hash = B84752DD5FBCBAA52B269AEFF426E56CE871E839BB2D6BF9BFE8661F1CF3B9D2
N output = 00000000
Input script =
493046022100BC4721CDE975BF841887D1B20F6A57A52E690097E0919082FFA1C7C367EA68090221008
D1B3D65F9D524D768D0C1B3378ED22C84538916EDCD028C88C88EF215D4B4DF014104DAF3AECC63
0E26081EE3DC71FE9ECAEF5079C6BD88603E07CD7C220BE284C45D2520AD5BD02FC10B4FC9B
CB6A329F78A8BB41532241E1379A0B2F522C6578

sequenceNumber = FFFFFFFF

TX from hash = 2BD4E5AD27F5DA69BB41BC25FB76552C3387EA9478659B236B7F61B2239F9C1E
N output = 00000001

Input script =
483045022100E11BCB7C52B36A541A8966D0D772246873FE8319C9DA9C31D3455A8518FD3E102202
D9DDA0EA53C3F4E8D1A4F71AB3E02BF81D8B325D6A664BE78BE2154A0E61A01410442BC767A
4A0BA536EF3B9B910DA0C247433FD9F1BF6F682FFDC91DD814B2D312AA49456C500535CF9C4B
2B9CDF95E503EA580976FA8666421282EE769A5

sequenceNumber = FFFFFFFF

Outputs count = 2
Value = 00000000001053B0

Output script = 76A9149CCEE6C1115388AAE386446B84C37A6CDBAFF54888AC
Value = 00000000874C6210

Output script = 76A914A2B088D0DB5E1C4FB3777211E562A092D635C9D888AC
Lock time = 00000000

TX hash = 2BBDAF34B706751CC0FA20E3D7F7E3EC61AE757C34AE276DE0B3B1C2F0452C47

transactionVersionNumber = 00000001

Inputs count = 09
TX from hash = 9CE957F47BFB173DF5265800D8F8E6F6BC97E8B82B99AE4C94D63163B05A77C
N output = 00000001

Input script =
493046022100F3B87863995EE765AC1104D2578784812DBE6D6037CD26ED5097666FA129B3DD022100
DB15AC01410713EC35ABC52DDCB678366A111634B62BDA0491A25FBBBD5419D80141040273BDAD1
4BA0B4DD3BDEEEA527093BA04F86B1C48AF1CC718CCBC4277138539922C960E1F681AF979886366
086CD01A816CC50076AD326106130DDF31D8971

sequenceNumber = FFFFFFFF

TX from hash = 33F6D88B89725655C2F76804EFEB0ACFC3A386500F85A3504FF5ABBAA9985287
N output = 00000001

Input script =
483045022100EA3BD29E8A24D1478F9DBA17F61A0E242D57A9076F46C2873997E99DA1211184022068
4CC1AE23143D631B7546CE81F65DB1C3C2E8ECE101B9E67544C6630FFC14B2014104AD44FF0DA8A46AC3D3364C1CA5052513127443B55A1B6CE55886DAD76B45BDC2619E56554A8A08F3CF48F2F813846E29BFE23FE89B79FB0552AE0DE4CF31065

sequenceNumber = FFFFFFFF

TX from hash = 68E6ADFC009339B0655B2F516D6EBC93B7800D442E07F979DBA93711CC969FA1
N output = 00000001

Input script =
493046022100F68D1748F84794ECAFC9DEAF69A0FB4F4B7CAE9550B6E41CF0620C9D8E38EC89022109CA6004729DB2FBBAC0D69256F1443D8803DC7357DE983BAD06A18B0C49E48DD0140AD44F0D8A46AC3D3364C1CA5052513127443B55A1B6CE55886DAD76B45BDC2619E56554A8A08F3CF48F2F813846E29BFE23FE89B79FB0552AE0DE4CF31065

sequenceNumber = FFFFFFFF

TX from hash = 9F6C1DE70F75D9FB7D6018D1FE6EA8C74FE11568ACE4232FA21DBA0439473415
N output = 00000001

Input script =
483045022100E0DE4D0629D3D01709A86072DEF8DA4D8E00B33E4B56E224BC78E087640AF02202E945702FD9193AD51A4FE4E506864E0D3A491E5CE21E66D9C2C43C1048F77A0141042B10112C0B27BEAD5D47DDFED6C9006FB94BE5D5A79B4FD75F52492591D98BB5CC2E836F86A534E58B686A17C41D7F1077272208E856951F466A919A17222B

sequenceNumber = FFFFFFFF

TX from hash = D3E6FCA9318DFF9D9461CF06217BA6BA681FB8C2557BB5263E2EBBF3395ED
N output = 00000001

Input script =
493046022100A130DEC5AA576895C58F5468FF3EDB44F7B65B2034470B75EE6A3797B36C60030221009BB3AD1AAD1B83BEB68E1B4C0B2A96D78399B941B266C945C839D1AAE5C05AD8014104AD44FF0D8A46AC3D3364C1CA5052513127443B55A1B6CE55886DAD76B45BDC2619E56554A8A08F3CF48F2F813846E29BFE23FE89B79FB0552AE0DE4CF31065

sequenceNumber = FFFFFFFF

TX from hash = C925F340A8D39C30D7F32A18FE36664ABD50E23F6F58ECCF52FF1E89A45DD4FC
N output = 00000000

Input script =
48304502205E2DC507A90DCA627D92718ED6B6A675A7EA8A3764ABB390C372C6200682416E0022100B699534ED26C537FD57BA166CCA1C1EF781BE9A99F899D5CB8A9AE9E2179862801410437194FD40EE9E6957D3902424FF805BE1A4D46A3710DEBB9213FB3542D4907F441EF00815B510D51AA59E52BC4E4A266D182EA6AA3742C561E3AC2C5F12BB

sequenceNumber = FFFFFFFF

TX from hash = 3AA192EA08F6C61E2624C10069A7FBD11435EC97413AF45AED8E08533EC9F633
N output = 00000001
Input script = 483045022076EADA08F59A79BD60F36B0C32C00965BE67F733D2E00B1B21F76C12132E3F8002210098379AAA6BB10B568BFE29B5F0EE96E10B3355558A742DEEF386B2697D09154112014104AD44FF0DA8A46AC3D3364C1CA50525131274434B55A1B6CE55886DAD76B45BDC2619E56554A8A08F3CF48F2F813846E29BFE23FE89B79FB0552AE0DE4CF31065

sequenceNumber = FFFFFFFF

TX from hash = C7627B63C52B26A8B53C8EA0460FA983236AC01C788C3DB45802F30C747D6B52

N output = 00000001

Input script = 483045022100C3ED0479772C209141DA58A221ECF839323D00E292558951D2A6AA4ABD776A7202207C83ADFB1C3E91EAEE8F0F8D48F3322315314866316347B3C1597616273120969014104AD44FF0DA8A46AC3D3364C1CA50525131274434B55A1B6CE55886DAD76B45BDC2619E56554A8A08F3CF48F2F813846E29BFE23FE89B79FB0552AE0DE4CF31065

sequenceNumber = FFFFFFFF

TX from hash = 03DE77B8290FD9002B4F553F69D24DFC1477396F69CECCD436B71A34CB1C8838

N output = 00000001

Input script = 48304502204BEB127F316293A6542BDDE6A7A9B0B5F37C44B7D61CE344EDFE39A012A62349022100D5BD9A5CF3A41EADCD03F91B863D50F443A9B658D60CB50ABF8D4CAEE73D3AE014104AD44FF0DA8A46AC3D3364C1CA50525131274434B55A1B6CE55886DAD76B45BDC2619E56554A8A08F3CF48F2F813846E29BFE23FE89B79FB0552AE0DE4CF31065

sequenceNumber = FFFFFFFF

Outputs count = 2
Value = 00000000002A31DC

Output script = 76A914976593B6577387D146B2B74F4694FCC9BC5C6FD88AC
Value = 0000000005F5E100

Output script = 76A914DC00CEA8B6A0FE9CAA468689AF465857D78485AD88AC
Lock time = 00000000

TX hash = 48620342BDFD49A93AC29BCA695E7492C2AE4CC390E1E9D17B65CF43FC1CC2BE

transactionVersionNumber = 00000001

Inputs count = 01
TX from hash = 3DA98958615DAF4B2587957A77E37A0AA817B84789466D1A7D426CE7632EFDD2D
N output = 00000000

Input script = 48304502200B38EDD2BE436E3F1D8D7C336AFFBFBC4E40CE566E8B4597768B2BF1BC53F291022100E

79
sequenceNumber = FFFFFFFF

Outputs count = 2

Value = 0000000312954DBD

Output script = 76A914C0438AB8295575507812713DDA288578258E026B88AC

Value = 00000000000F4240

Output script = 76A914795D8098D01ACF32327CD537CE382D02C006CE7F88AC

Lock time = 00000000

TX hash = DDB83B614D2246D1A61B1098604FB20785D43313F716D21757CA60895307E62D

transactionVersionNumber = 00000001

Inputs count = 01

TX from hash = 73592B21D859D0B397A2BB81E8389B4B91F16BD78F6645596BFBB72C8DCCD3DC

N output = 00000000

Input script = 48304502207210E5E57C5E37AC33324E481DDC82D9FB2F876CBAC0638553FAAC4EC688DE570221009B890ADAB71364F50ACA58849D45748A3136212BD57E332D2ABE19FCC2047728014104802F20C768273EEC7192DD752D6E1DD0A3FA5137EE702C8B8F4E8908D1A2C5B71389CDED9B1523D4326C4DC1C2BF9B775AC84F6839E9D34D43CA31C8513B4A8

sequenceNumber = FFFFFFFF

Outputs count = 2

Value = 000000030E942F7C

Output script = 76A9148F56AA0BE898C338C1D4A8730CDAC2C8D6066AE188AC

Value = 0000000002FAF080

Output script = 76A914DF76CA6890594D2512B17CAB3A2A97A0B0B75DFE88AC

Lock time = 00000000

TX hash = C366A68B8E0864AF74587EF6CF9566F7A10D0670288E422688B8D48EAA5DC37C
Statistics for parsed data results

Figure: Cumulative parsing time vs Number of parsed blocks

Notes: The vertical lines represent points where the linear relationship between time and blocks changes.

Table: Summary Statistics for Code v1

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>sd</th>
<th>min</th>
<th>max</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>0.016</td>
<td>0.033</td>
<td>0</td>
<td>1</td>
<td>3424.53</td>
</tr>
<tr>
<td>InputsOutputs</td>
<td>195.932</td>
<td>475.086</td>
<td>2</td>
<td>13192</td>
<td>41304251</td>
</tr>
<tr>
<td>Transactions</td>
<td>44.985</td>
<td>107.982</td>
<td>1</td>
<td>1871</td>
<td>9475083</td>
</tr>
<tr>
<td>Observations</td>
<td>210623</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: The table presents summary statistics for the collected time data of the 1st code version. Time is measured in seconds.
due to RAM mismanagement. We present below the scatter-plot of parsing time vs blocks. Figure 9 indicates a positive linear relationship between the amount

14Acquiring an address from the raw data of the Blockchain is a procedure with certain steps one of which is Base58 encoding. As the task of this thesis is not programming an encoder we borrowed a already operational python script to perform this task.

Figure Scatterplot of parsing time and the total of inputs and outputs

Notes:A Scatterplot of parsing time and the total of inputs and outputs contained in a block. Data include 119 thousand blocks and their respective count of inputs and outputs.
Chapter 5

Conclusion

Some people are excited about Bitcoin because of the underlying technology. Others are excited about its commercial possibilities, and yet others about its social and political implications. Reasonable people can disagree about the latter two, but we hope this book has convinced you that technologically, Bitcoin is deep, novel, interesting, and based on sound principles. Beyond Bitcoin there is a fascinating world of alternative cryptocurrency designs that we’re just starting to explore, some of which might one day be more important than Bitcoin itself. We got into Bitcoin because we believe in the power of its technology, and we think it’s deeply connected to the rest of computer science. While we’ve highlighted how seemingly amazing new technology can struggle to displace established institutions, we believe that in the long run, people will continue to find new commercially and socially useful things to do with cryptocurrency technology. Bitcoin is pretty interesting from the underlying technology it was made from — blockchain. From watching documentaries on bitcoin, I do think there is transaction value for its holders unlike what other people might say on how bitcoin adds no value. As for blockchain, I really like how the no trust idea that it operates off of makes very interesting record keeping. I think that it would be very useful for data integrity, which is probably why some of you might have read on the news how the supply chain is applying blockchain. We introduced bitcoin with python. We saw how to get set up with python for bitcoin. We saw how to generate a private key, public key, and a bitcoin address. We also saw how to create a multi-signature bitcoin address and how to look at the transactional history of a bitcoin address. Creating the code in python was a very long and painful process. Running the code in python was also very slow due to the big data of the parsed files.
References


Diebold, F. (2012). A personal perspective on the origin (s) and development of big data: The phenomenon, the term, and the discipline, second version.


