Learn Blockchains by Building One | by Daniel van Flymen | MediumDaniel van FlymenFollow--344ListenShareYou're here because, like me, you're psyched about the rise of Cryptocurrencies. And you want to know how Blockchains work—the fundamental technology behind them.But understanding Blockchains isn't easy—or at least wasn't for me. I trudged through dense videos, followed porous tutorials, and dealt with the amplified frustration of too few examples.I like learning by doing. It forces me to deal with the subject matter at a code level, which gets it sticking. If you do the same, at the end of this guide you'll have a functioning Blockchain with a solid grasp of how they work.Remember that a blockchain is an immutable, sequential chain of records called Blocks. They can contain transactions, files or any data you like, really. But the important thing is that they're chained together using hashes.If you aren't sure what a hash is, here's an explanation.Who is this guide aimed at? You should be comfy reading and writing some basic Python, as well as have some understanding of how HTTP requests work, since we'll be talking to our Blockchain over HTTP.What do I need? Make sure that Python 3.6+ (along with pip) is installed. You'll also need to install Flask and the wonderful Requests library:Oh, you'll also need an HTTP Client, like Postman or cURL. But anything will do.Where's the final code? The source code is available here.Open up your favourite text editor or IDE, personally I PyCharm. Create a new file, called blockchain.py. We'll only use a single file, but if you get lost, you can always refer to the source code.We'll create a Blockchain class whose constructor creates an initial empty list (to store our blockchain), and another to store transactions. Here's the blueprint for our class:Our Blockchain class is responsible for managing the chain. It will store transactions and have some helper methods for adding new blocks to the chain. Let's start fleshing out some methods.Each Block has an index, a timestamp (in Unix time), a list of transactions, a proof (more on that later), and the hash of the previous Block.Here's an example of what a single Block looks like:At this point, the idea of a chain should be apparent—each new block contains within itself, the hash of the previous Block. This is crucial because it's what gives blockchains immutability: If an attacker corrupted an earlier Block in the chain then all subsequent blocks will contain incorrect hashes.Does this make sense? If it doesn't, take some time to let it sink in—it's the core idea behind blockchains.We'll need a way of adding transactions to a Block. Our new_transaction() method is responsible for this, and it's pretty straight-forward:After new_transaction() adds a transaction to the list, it returns the index of the block which the transaction will be added to—the next one to be mined. This will be useful later on, to the user submitting the transaction.When our Blockchain is instantiated we'll need to seed it with a genesis block—a block with no predecessors. We'll also need to add a "proof" to our genesis block which is the result of mining (or proof of work). We'll talk more about mining later.In addition to creating the genesis block in our constructor, we'll also flesh out the methods for new_block(), new_transaction() and hash():The above should be straight-forward—I've added some comments and docstrings to help keep it clear. We're almost done with representing our blockchain. But at this point, you must be wondering how new blocks are created, forged or mined.A Proof of Work algorithm (PoW) is how new Blocks are created or mined on the blockchain. The goal of PoW is to discover a number which solves a problem. The number must be difficult to find but easy to verify—computationally speaking—by anyone on the network. This is the core idea behind Proof of Work.We'll look at a very simple example to help this sink in.Let's decide that the hash of some integer x multiplied by another y must end in 0. So, hash(x * y) = ac23dc...0. And for this simplified

example, let’s fix x = 5. Implementing this in Python:The solution here is y = 21. Since, the produced hash ends in 0:In Bitcoin, the Proof of Work algorithm is called Hashcash. And it’s not too different from our basic example above. It’s the algorithm that miners race to solve in order to create a new block. In general, the difficulty is determined by the number of characters searched for in a string. The miners are then rewarded for their solution by receiving a coin—in a transaction.The network is able to easily verify their solution.Let’s implement a similar algorithm for our blockchain. Our rule will be similar to the example above:Find a number p that when hashed with the previous block’s solution a hash with 4 leading 0s is produced.To adjust the difficulty of the algorithm, we could modify the number of leading zeroes. But 4 is sufficient. You’ll find out that the addition of a single leading zero makes a mammoth difference to the time required to find a solution.Our class is almost complete and we’re ready to begin interacting with it using HTTP requests.We’re going to use the Python Flask Framework. It’s a micro-framework and it makes it easy to map endpoints to Python functions. This allows us talk to our blockchain over the web using HTTP requests.We’ll create three methods:Our “server” will form a single node in our blockchain network. Let’s create some boilerplate code:A brief explanation of what we’ve added above:This is what the request for a transaction will look like. It’s what the user sends to the server:Since we already have our class method for adding transactions to a block, the rest is easy. Let’s write the function for adding transactions:Our mining endpoint is where the magic happens, and it’s easy. It has to do three things:Note that the recipient of the mined block is the address of our node. And most of what we’ve done here is just interact with the methods on our Blockchain class. At this point, we’re done, and can start interacting with our blockchain.You can use plain old cURL or Postman to interact with our API over a network.Fire up the server:Let’s try mining a block by making a GET request to http://localhost:5000/mine:Let’s create a new transaction by making a POST request tohttp://localhost:5000/transactions/new with a body containing our transaction structure:If you aren’t using Postman, then you can make the equivalent request using cURL:I restarted my server, and mined two blocks, to give 3 in total. Let’s inspect the full chain by requesting http://localhost:5000/chain:This is very cool. We’ve got a basic Blockchain that accepts transactions and allows us to mine new Blocks. But the whole point of Blockchains is that they should be decentralized. And if they’re decentralized, how on earth do we ensure that they all reflect the same chain? This is called the problem of Consensus, and we’ll have to implement a Consensus Algorithm if we want more than one node in our network.Before we can implement a Consensus Algorithm, we need a way to let a node know about neighbouring nodes on the network. Each node on our network should keep a registry of other nodes on the network. Thus, we’ll need some more endpoints:We’ll need to modify our Blockchain’s constructor and provide a method for registering nodes:Note that we’ve used a set() to hold the list of nodes. This is a cheap way of ensuring that the addition of new nodes is idempotent—meaning that no matter how many times we add a specific node, it appears exactly once.As mentioned, a conflict is when one node has a different chain to another node. To resolve this, we’ll make the rule that the longest valid chain is authoritative. In other words, the longest chain on the network is the de-facto one. Using this algorithm, we reach Consensus amongst the nodes in our network.The first method valid_chain() is responsible for checking if a chain is valid by looping through each block and verifying both the hash and the proof.resolve_conflicts() is a method which loops through all our neighbouring nodes, downloads their chains and verifies them using the above method. If a valid chain is found,

whose length is greater than ours, we replace ours.Let’s register the two endpoints to our API, one for adding neighbouring nodes and the another for resolving conflicts:At this point you can grab a different machine if you like, and spin up different nodes on your network. Or spin up processes using different ports on the same machine. I spun up another node on my machine, on a different port, and registered it with my current node. Thus, I have two nodes: http://localhost:5000 and http://localhost:5001.I then mined some new Blocks on node 2, to ensure the chain was longer. Afterward, I called GET /nodes/resolve on node 1, where the chain was replaced by the Consensus Algorithm:And that’s a wrap... Go get some friends together to help test out your Blockchain.I hope that this has inspired you to create something new. I’m ecstatic about Cryptocurrencies because I believe that Blockchains will rapidly change the way we think about economies, governments and record-keeping.Update: I’m planning on following up with a Part 2, where we’ll extend our Blockchain to have a Transaction Validation Mechanism as well as discuss some ways in which you can productionize your Blockchain.If you enjoyed this guide, or have any suggestions or questions, let me know in the comments. And if you’ve spotted any errors, feel free to contribute to the code here!