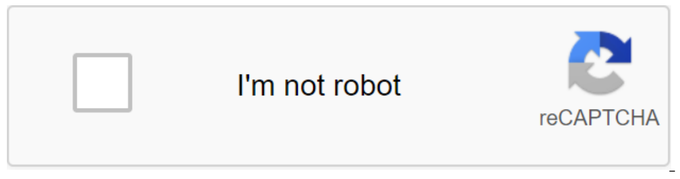

Continue

Blockchain programming in python pdf

This tutorial introduces Python developers, any level of programming skills, to blockchain. You will learn what blockchain is by introducing a public blockchain from scratch and building a simple application to use it. You'll be able to create endpoints for different blockchain functions using a flask micro-frame, and then run scripts on multiple machines to create a decentralized network. You'll also see how to create a simple user interface that interacts with the blockchain and stores information for any use case, such as peer-to-peer payments, chat rooms, or e-commerce. Python is a simple programming language to understand, so I chose it for this tutorial. As this tutorial progresses, you'll implement a public blockchain and see it in action. The code for the full sample of the app, written with Pure Python, is available on GitHub. Take the code. Now, to understand the blockchain from scratch, let's prepare it together. Premise Basic Knowledge Programming Python Knowledge REST-APIs Familiarity with the flask micro-frame (not necessarily but nice to have) Background Information In 2008, a whitepaper called Bitcoin: Peer-to-Peer Electronic Monetary System was released by a person (or maybe a group) named Satoshi Nakamoto. The document combined several cryptographic methods and a peer-to-peer network to transfer payments without the participation of any central authority (such as a bank). A cryptocurrency called Bitcoin was born. In addition to Bitcoin, the same paper introduced a distributed storage system (now popularly known as the blockchain), which had a much broader applicability than just payments or cryptocurrencies. Since then, blockchain has attracted interest in almost all industries. Blockchain is now the main technology behind fully digital cryptocurrencies such as Bitcoin, distributed computing technologies such as Ethereum, and open source platforms such as Hyperledger Fabric, on which the IBM Blockchain Platform is built. What is blockchain? Blockchain is a way of storing digital data. Data can be literally anything. For Bitcoin, these are transactions (Bitcoin transfer logs from one account to another), but they can even be files; Never mind. The data is stored in the form of blocks that are connected (or chained) to each other using cryptographic hashes - hence the name of the blockchain. All the magic is how this data is stored and added to the blockchain. Blockchain is essentially a related list that contains orderly data, with several limitations, such as: Blocks cannot be changed after adding; in other words, this app only. There are specific rules for the data to be applied to it. Its architecture is distributed. The application of these restrictions gives Benefits: The immutable and durability of data data Control or failure is a verifiable track audit order in which the data was added So how can these limitations reach these characteristics? We'll get more into that as we implement this blockchain. Start. About the app Let's briefly determine the scope of our mini-app. Our goal is to create an app that allows users to share information through placement. Since the content will be stored on the blockchain, it will be unchanged and permanent. Users will interact with the app through a simple web interface. The steps we will take using the bottom-up approach. Let's start by determining the structure of the data that we will store in the blockchain. A message is a message posted by any user in our app. Each post will consist of three main elements: Keep transactions in blocks We will store data in our blockchain in a format that is widely used: JSON. Here's what a post stored in the blockchain would look like: author:some_author_name,media:Some thoughts that the author wants to share, timestamp: The time when content was created, The generic term data is often replaced on the Internet by the term transactions. So, to avoid confusion and keep consistency, we'll use the term transaction to refer to data in our application example. Transactions are packed into blocks. The block may contain one or more transactions. Blocks containing transactions are often generated and added to the blockchain. Because there may be several blocks, each block must have a unique identifier: class block: def __init__ (self, index, transaction, time stamp): Designer for 'Block' class. :param index: Unique block ID. :params of transactions: A list of transactions. :param time stamp: Time to generate the block. self.index - index self.transactions - transactions self.timestamp - timestamp Add digital fingerprints to blocks We would like to prevent any tampering in the data stored inside the block, and detection is the first step to this. You can use cryptographic hash functions to find out if data has been tampered with in a block. A hash function is a feature that takes data of any size and produces fixed-size (hash) data from it, which is commonly used to identify input. Characteristics of the ideal hash function: it should be easily calculated. It should be deterministic, that is, the same data always lead to the same hash. It should be evenly random, meaning even one small change in data should significantly change the hash. The consequence of this is: It is almost impossible to guess the inputs taking into account the hash. (The only way is to try all possible input combinations.) If you both input and hash, you can simply pass the entrance through hash hash to check the hash provided. This asymmetry effort required to clear the hash out of input (easily) against figuring out hash input (almost impossible) is that blockchain leverages to get the desired characteristics. There are various popular hash features. Here's an example in Python that uses the hashing feature SHA-256: b919fcae38e2bdaebb6c0404098e5c70563d2dc51e085f784c058ff208516' sha256 (data).hexdigest (data).hexdigest (data).'i.e. How many times you run it, the result will be the same 256 character line 'b919fcae38e2bdaebb6c04ed4098e5c70563d2dc51e1085f784c058ff208516' data bSome variable length data2 - Added one character at the end. '9fcaab521baf83f07512a7a7a7a7a7a0f567f6eef2688e8b9490694ada0a3ddeec8' - Note that the hash has completely changed! We will store the hash block in the box inside our block of the object, and it will act as a digital fingerprint (or signature) of the data contained in it: from hashlib import sha256 import json def compute_hash (block): Returns the hash of the copy block, first turning it into a JSON line. block_string and json.dumps (self.__dict__, sort_keys-Truth) return sha256 (block_string.encode ()))hexdigest () Note: In most cryptocurrencies, even individual transactions in the block are hashed and then stored to form a hash tree (also known as a merkle tree). The root of the tree usually represents the hash block. This is not a necessary requirement for the functioning of the blockchain, so we omit it so that everything was simple. Okay, we've already installed the blocks. Blockchain should be a set of blocks. We can store all the blocks on the Python list (the equivalent of an array). But it is not enough, because what to do if someone intentionally replaces the old block with a new block in the collection? Creating a new block with modified transactions, calculating the hash and replacing it with any old block doesn't matter much in our current implementation. We need a way to make sure that any changes in previous blocks cancel the entire chain. Bitcoin's way of doing this is to create dependency among successive blocks by chaining them with a hash block just in front of them. By chaining here, we mean to include the hash of the previous block in the current block in a new box called previous_hash. Well, if each block is connected to the previous block through the previous_hash, what about the very first block? This block is called a genesis block, and it can be created either by hand or by some unique logic. Let's add a previous_hash field to the Block class and implement the initial structure of our Blockchain class. from hashliba import sha256 import json import time-class Block: def__init__ (self, transactions, time stamp, previous_hash): Designer for the Block class. :param index: Unique block ID. :params of transactions: A list of transactions. :param time stamp: Time to generate the block. :pprevious_hash: The hash of the previous block in the chain, of which this block is a part. self.transactions - self.timestamp - timestamp self.previous_hash and previous_hash - Adding the previous hash field def compute_hash (self): Returns the hash of the copy block, first converting it into a JSON line. block_string and json.dumps (self.__dict__, sort_keys) - the line equivalent also believes that the previous_hash field now returns sha256 (block_string.encode()self.create_genesis_block create_genesis_block __init__). : The function of generating the genesis of the block and claims it to the chain. The block has an index of 0, previous_hash as 0, and a valid hash. genesis_block - Block (0, No, time.time(), 0) genesis_block.hash - genesis_block.compute-hash () self.chain.append (genesis_block) @property def last_block (self): A quick python method to get the latest block in the chain. Note that the chain will always consist of at least one block (i.e. the genesis block) of the self.chain-1 return Now that the contents of any of the previous blocks change: the hash of the previous block will change. This will cause a mismatch with the previous_hash of the field in the next block. Because the input to calculate the hash of any block also previous_hash the field, the next block's hash will also change. Ultimately, the whole chain is invalid after the block is replaced, and the only way to fix it is to refil up the entire chain. Implementing proof algorithm work There is one problem, however. If we change the previous block, the hashes of all subsequent blocks can be fairly easily listed to create another valid blockchain. To prevent this, we can use asymmetry in the hash functions that we discussed earlier to make the hash calculation task difficult and random. Here's how we do it: Instead of taking a hash for the block, we add some limitations to it. Let's add a limitation that our hash should start with n leading zeros where n can be any positive integer. We know that if we don't change the block data, the hash won't change, and of course we don't want to change existing data. So what do we do? Simple! We'll add some bogus data that we can change. Let's introduce a new field in our block called nonce. Nonce is a number that we can continue to change until we get a hash that satisfies our limitations. Nonce satisfying restriction serves as proof that some calculations have been performed. the technique is a simplified version of the Hashcash algorithm used in Bitcoin. The number of zeros specified in the restriction determines the complexity of our proof of the algorithm of work (the greater the number of zeros, the harder it is to find out what is not). Also, due to asymmetry, proof of work is difficult to calculate, but very easy to verify once you figure out nonce (you just have to run the hash function again): Class Blockchain: - Complexity of the PoW difficulty algorithm No. 2 Previous contd code. def proof_of_work (self, block): a feature that tries different nonce values to get a hash that meets our difficulty criteria. Block.nonce No 0 computed_hash and block.compute_hash () rather than computed_hash.startswith ('0' Blockchain.difficulty): block.nonce No 1 computed_hash and block.compute_hash () return computed_hash Notice that there is no specific logic to figure out nonce quickly; it's just brute force. The only definite improvement you can make is to use hardware chips that are specifically designed to calculate hash function in fewer CPU instructions. Add blocks to the chain To add a block to the chain, we first need to make sure that: the data has not been tampered with (proof of work is provided correctly). The order of transactions is maintained (previous_hash the block field, to the hash of the last block in our chain). Let's see the code to add blocks to the chain: Blockchain class: Previous contd code. def add_block (self, block, proof): A function that adds a block to the chain after checking. The test includes: - Check whether the proof is valid. - previous_hash mentioned in the block and the last block hash in a chain match. previous_hash and self.last_block.hash if previous_hash ! block.previous_hash: Return False, if not Blockchain.is_valid_proof (block, proof): Return False block.hash - proof of self.chain.append (block) return True def is_valid_proof (self, block, block_hash): Check if block_hash is a valid hash block and meets the difficulty criteria. Return (block_hash.startswith ('0' Blockchain.difficulty) and block_hash - block.compute_hash ()) Mining Transactions will initially be stored as a pool of unconfirmed transactions. The process of putting unconfirmed transactions into a block and computational evidence of work is known as block mining. Once nonce satisfying our limitations are calculated, we can say that the block has been mined and it can be entered into the blockchain. In most cryptocurrencies (including Bitcoin), miners can get some cryptocurrency as a reward for spending their computing power on computing evidence. Here's what our mining function looks like: Blockchain: def __init__ (self): self.unconfirmed_transactions - data that have yet to be obtained self.chain - self.create_genesis_block () Previous code contd... def add_new_transaction (self, transaction): self.unconfirmed_transactions.append (transaction) def mine (self): This function serves as an interface for adding pending transactions to the blockchain, adding them to the block and figuring out evidence of work. if not self.unconfirmed_transactions: return False last_block - self.last_block new_block - Block (index -last_block.index No. 1, transaction-self.unconfirmed_transactions, time and time(), previous_hash-last_block.hash) proof - self.proof_of_work (new_block) self.add_block (new_block, proof) self.unconfirmed_transactions - new_block.index Alright, we're almost there. With this point, you can see the combined code on GitHub. Well, now it's time to create interfaces for our blockchain site to interact with the app we're going to create. We will use a popular Python micro-frame called Flask to create a REST API that interacts with and triggers different operations in our blockchain site. If you've worked with any web platform before, the code below shouldn't be hard to follow along. from flasks import flasks, import requests - Initial application for flask application - flask (__name__) - To initiate a blockchain object. Blockchain () We need an end point for our application to present a new transaction. This will be used by our app to add new data (messages) to the blockchain: 'Flaba is a way to announce endpoints @app.route ('/new_transaction', POST) def new_transaction (): tx_data and request.get_json () required_fields author, content for field in required_fields: if not tx_data.get (field): Return Invalid transaction data, 404 tx_data timestamp.) blockchain.add_new_transaction (tx_data) Throwback Success, 201 Here's the end point to bring back a copy of the node chain. Our app will use this endpoint to request all data to display: @app.route('/chain', def get_chain (): chain_data - for block in blockchain.chain: chain_data.append (block.__dict__) return json.dumps (length:len (chain_data), chain: chain_data) Here's the end point to request a node to mine unconfirmed transactions (if any). We will use it to initiate a mine command from our app ourselves: @app.route ('/mine', methods 'GET') def mine_unconfirmed_transactions (): result blockchain.mine () if not the result: return No transactions at the mine return Block No min. format (result) @app.route ('/pending_tx') def get_pending_tx (): reverse json.dumps (blockchain.unconfirmed_transactions) These rest endpoints can be used to play with our blockchain, creating some transactions. To build consensus and Up to this point, the blockchain that we have implemented is designed to start launching one computer. Despite the fact that we associate the block with hash and apply evidence of work restriction, we still can not trust any entity (in our case one machine). We need the data to spread, we need a few knots, behav blockchain. So, to move from one site to a peer-to-peer network, let's first create a mechanism that allows the new site to learn about other nodes on the network: Contains the addresses of other participating peer members and a set of endpoints to add new nodes to the network @app.route ('/register_node', 'POST') def register_new_peers (): Host address of the site node_address - request.get_json () node_address if not node_address: return Invalid data, 400 - Add a node to peers.add (node_address) blockchain to a newly registered node, so that it can synchronize the get_chain() @app.route ('/register_with', methods'post)register_with_existing_node): Internally calls the end point 'register_node' to register the current node with the deleted node listed in the request, and synchronize the blockchain as well as the remote node. node_address - request.get_json () node_address, if not node_address: return Invalid data, 400 data - node_address: request.host_url: Content type: app/register_node json node_address - Make a request for registration with a remote site and get requests for response to the information. landfills (data), blanks), if response.status_code No. 200: global blockchain of global peers - the upgrade chain and peers chain_dump - response.json () blockchain - create_chain_from_dump (chain_dump) peers.update (response.json)))) return Registration successfully, 200 others: response.status_code def create_chain_from_dump (chain_dump): blockchain and blockchain () for idx, block_data in the list (chain_dump): block and block (block_data index, block_data transaction, block_data time stamp, block_data previous_hash) block_data proof : added : blockchain.add_block (block, block, proof) if not added: raise the exception (Chain dump forged!!) still: - genesis block, no verification required blockchain.chain.append (block) the return of the blockchain New node participating in the network, can cause a method of register_with_existing_node (through the endpoint /register_with) to register with existing nodes in the network. This will help with the following: Ask a remote site to add a new site to the list of known nodes. Initiate the blockchain of a new node using a remote node. Reuse the blockchain on the network if the site leaves the network. However, there is a problem with multiple nodes. Due to deliberate manipulation or unintentional (e.g. network latency), a copy of multiple node chains may be different. In this case, the nodes need to agree on some version of the chain to maintain the integrity of the entire system. In other words, we must reach a consensus. A simple consensus algorithm can be agreed on the longest active chain when the chains of the various participating nodes in the network appear to diverge. The rationale for this approach is that the longest chain is a good estimate of the greatest amount of work done (remember, proof of the work is difficult to calculate): the Blockchain class the previous code continued ... def check_chain_validity (cls, chain): An assistant method to check if the entire blockchain is valid. result - True previous_hash - 0 - Iterate through each block block in the chain: block_hash - block.hash - remove the hash field to recomponate the hash again - using the method 'compute_hash'. delattr (block, hash) if not cls.is_valid_proof (block, block.hash) or No previous_hash ! block.previous_hash: result - False break block.hash, previous_hash and block_hash, block_hash return result def consensus(): Our simple consensus algorithm. If a longer active chain is found, our chain is replaced by it. Global Blockchain longest_chain - No current_len - len (blockchain.chain) for nodes near peers: answer - requests.get ("/chain'.format)) length - response.json ())'length' chain - response.json ()'chain', If the length - current_len and blockchain.check_chain_validity (chain): current_len - length of longest_chain - chain, if longest_chain: blockchain and longest_chain return True return False Next, we must develop a way for any site to announce the network that it is mined the block so that everyone can update their blockchain and move on to demining other transactions. Other nodes can simply check the proof of work and add a mined block to their chains (remember that the check is simple once nonce is known): - the end point to add a block mined by someone else to the node chain. The node first checks the block and then adds it to the chain. @app.route ('/add_block', def methods verify_and_add_block(): block_data and request.get_json (block_data index, transaction block_data, block_datatimestamp, block_data

previous_hash) proof - block_data'hash' added - blockchain.add_block (block, proof), if not added: the return of the block was dropped the knot, 400 return Block added to the chain, 201 def announce_new_block (block): The function to declare into the network as soon as the block was mined. Other blocks can simply check proof of work and add it to their chains. peer-to-peer: URL - No add_block.format (peering) requests.post (url, data'json.dumps (block.__dict__, sort_keys)) Method announce_new_block must be called each block block knot so that peers can add it to their chains. @app.route ('/mine', methods 'GET') def mine_unconfirmed_transactions (): the result of blockchain.mine () if not the result: return No transactions to the mine yet: - Making sure that we have the longest chain before announcing the network chain_length (blockchain.chain) consensus (if chain_length th len (blockchain chain.): - announce the recently mined blocking of the network announce_new_block (blockchain.last_block) return Block. format (blockchain.last_block.index Alright, blockchain server set up. You can see the code up to this point on GitHub. We used Jinja2 templates to visualize web pages and some CSS to make things look good. Our app needs to connect to the site on the blockchain network to get data as well as provide new data. There may also be several nodes as well. import date import json import requests from flask import render_template, redirect, request from the import app Node in the blockchain network that our app will communicate with. CONNECTED_NODE_ADDRESS messages - the fetch_posts feature receives data from the end point of the node/chain, analyzes the data and stores it locally. def fetch_posts (): Function to get the chain out of the blockchain site, disassemble the data, and store it locally. get_chain_address - W/chain.format (CONNECTED_NODE_ADDRESS) answer - requests.get (get_chain_address) if response.status_code - 200: content and chain json.loads (answer.content) for a block in the chain: for tx in the transaction block: txindex - blocks indextxhash - block previous_hash content.append (tx) global posts - content content, key'lambda k: k'timestamp', reverse -Truth) The app has an HTML form for user input and then makes a POST request to a connected site to add a transaction to the pool of unconfirmed transactions. The transaction is then mined online and then finally brought in as soon as we update our web page: @app.route ('/imagine', methods 'POST')) def submit_textarea (): The end point to create a new transaction through our app post_content post_object 'content': post_content, - Send a transaction new_tx_address - W/new_transaction.format (CONNECTED_NODE_ADDRESS) requests.post (new_tx_address, json'post_object, headers'Content-type: 'app/json') - Return to the home redirect return page ('/') It's done! The final code can be found on GitHub. Clone Project: $git clone Set dependencies: $CD python_blockchain_app $Peep set-r requirements.txt Start server node blockchain: $ export $flask run - Port 8000 One copy of our blockchain hub is being launched and operating in port 8000. Start the app at another terminal session: the app must run in . Numbers 1 - 3 illustrate how to place content, request a node for a mine, and resync with a chain. Figure 1. Post some content Figure 2. Request a node for mine figure 3. Resyncing with a chain for updated data Running with multiple nodes to play around, rotating from multiple custom nodes, use register_with/endpoint to register a new site with an existing peer network. Here's an example of a scenario you can try: already running a $ flask run - port 8000 - spinning new knots $ flask run --port 8001-port--port 80018002 - you can use the following cURL queries to register nodes in port 8001 and 8002 with an already running 8000: $ curl -- X POST Content Type: app/json'd'node_address: - $8,000 - curl -X POST - - H 'Content-Type: app/json' -d'node_address: - This will make the node in port 8000 aware of nodes in port 8001 and 8002, and vice versa. The new nodes also synchronize the chain with the existing node so they can actively participate in the mining process. To update the site that syncs the app interface (the default is the local port of 8000), change the CONNECTED_NODE_ADDRESS field in the views.py file. Once you've done all of this, you can run the app (python run_app.py) and create transactions (post messages via the web interface), and once you get the transactions, all the nodes on the network will update the chain. The node chain can also be checked by referring to the end point/chain using cURL or Postman. $-X GET $10,000 -X GET Authentication Deals You May Have Noticed a Flaw in the App: Anyone can change any name and post any content. In addition, the post is subject to tampering when sending a transaction to the blockchain network. One way to solve this problem is to create user accounts using the cryptography of public and private key keys. Each new user needs a public key (similar to the username) and a private key that can be placed in our app. The keys are used to create and verify a digital signature. Here's how it works: Each new transaction submitted (sent) is signed by the user's personal key. This signature is added to the transaction data along with user information. During the verification phase, when mining transactions, we can check whether the claimed owner is the same as stated in the transaction data, and that the message has not been changed. This can be done with the help of and the public key of the sought-after post owner. Conclusion This tutorial covers the basics of public blockchain. If you've followed together, you should now be able to implement the blockchain from scratch and create a simple application that allows users to share information about the blockchain. This implementation is not as complex as other public blockchains such as Bitcoin or Ethereum (and still has some loopholes) - but if you keep asking the right questions according to your requirements, you will eventually get there. The main thing to note is that most of the work on blockchain development for your needs is about combining existing concepts of computer science, no more! Next steps, you can unscrew a few nodes in the cloud and play around with the app you've created. You can deploy any Flask app in the IBM cloud. You can also use a tunnel service, such as ngrok, to create a public URL for your local server, and then you can interact with multiple machines. There's a lot to explore in this space! Here are some ways to continue the development of blockchain skills: Continue to learn blockchain technology by getting their hands on the new IBM Blockchain Platform platform. You can quickly promote a pre-production blockchain network, deploy sample applications, and develop and deploy client applications. Started! Stay in the know-know with the Blockchain newsletter from the IBM developer. Check out the latest questions and subscribe. Borrow a Blockchain Hub by an IBM developer. It is your source for tools and tutorials, along with code and community support, to develop and deploy blockchain solutions for your business. Continue to build your blockchain skills through the IBM Developer Blockchain learning path, which gives you the basics, then shows you how to start building apps, and offers useful uses for perspective. And don't forget to check out the many blockchain code templates on IBM Developer that provide road maps to solve complex problems, and include reviews, architecture diagrams, process streams, repo pointers, and additional reading. Reading. blockchain programming in python pdf. blockchain programming in python tutorial