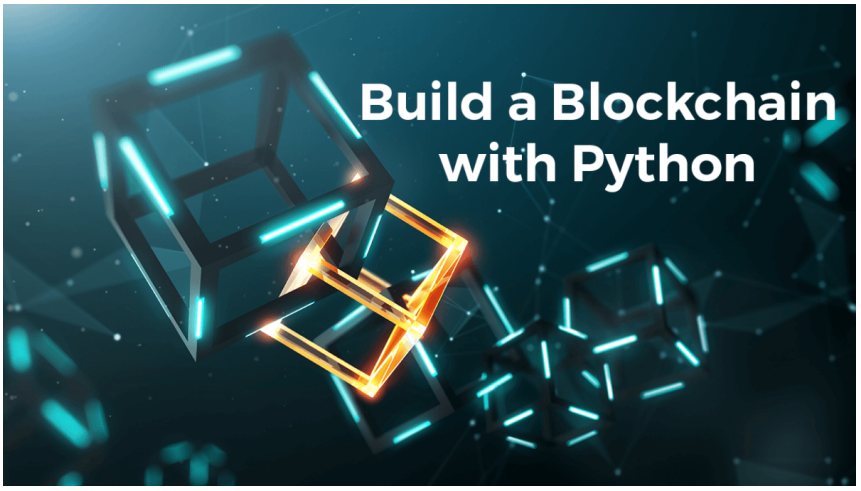
February 6, 2020 | bitcoin, Blockchain, how to build, proof of work, python

# How To Build A Blockchain In Python (Get Pre-Built Runtime)




Sign up for a free ActiveState Platform account so you can download the Blockchain runtime environment and build your very own blockchain

Before we explain how to build a blockchain in Python, let's go back to the very start. In 2008, an author (or authors) under the pseudonym Satoshi Nakamoto released a white paper describing a purely peer-to-peer version of electronic cash. Unique to this electronic cash system, transactions would not have to rely on third-party verifications to ensure the security of each transaction. Instead, each transaction would be timestamped, then hashed into an ongoing chain of hash-based proof-of-work.

So what are *hashing* and *proof-of-work*? I'll introduce these concepts in this article and reveal how they establish the groundwork for an encrypted electronic cash system, or cryptocurrency. The specific form of electronic currency that Satoshi Nakamoto described in his (or her) paper became known as Bitcoin, the first cryptocurrency. But how is this useful when you are trying to build a blockchain in Python?

## BLOG AUTHOR

Dante Sblendorio

**Guest blogger**: Dante is a physicist currently pursuing a PhD in Physics at École polytechnique fédérale de Lausanne. He has a Masters in Data Science, and continues to experiment with and find novel applications for machine learning algorithms. He lives in Lausanne, Switzerland.

## PRACTICAL INFO IN YOUR INBOX

Get our latest blogs, resources and insights to help you create more value with open source languages

Email*

Email Address

Sign Me Up »

Contact Sales Sign In Create Account

ActiveState®

GET A DEMO

WHY ACTIVESTATE PRODUCTS SOLUTIONS PRICING RESOURCES

The system that Bitcoin relies upon — a growing list of records (i.e. blocks) that are linked to one another — is known as a blockchain. Bitcoin was the first successful application of this system, and shortly after its rise in popularity, other cryptocurrencies were founded on the same principles. This system, however, is not restricted to storing financial information. Rather, the type of data being stored is inconsequential to and independent of the blockchain network.

Fundamentally, the data stored in a blockchain must have the following characteristics:

- Immutable
- Unhackable
- Persistent (no loss of data)
- Distributed

These qualities are necessary to maintain the integrity of the blockchain and the security of the network within which the transactions occur. To illustrate the simplicity and elegance of such a system, and to explain the subtleties, I will walk you through the process of creating your very own blockchain in Python. To keep it simple, I will assume that the data stored in the block is transactional data, as cryptocurrencies are currently the dominant use case for blockchain.

## Build A Blockchain In Python With This Pre-Built Runtime Environment

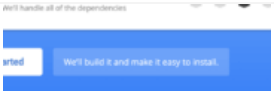
To follow along with the code in this tutorial, you'll need to have a recent version of Python installed. I'll be using **ActivePython**, for which you have two choices:

- Download and install the pre-built Blockchain runtime environment (including Python 3.6) for Windows 10 or CentOS 7, or
- Build your own custom Python runtime with just the packages you'll need for this project, by creating a free ActiveState Platform account, after which you will see something like the following image:

Contact Sales Sign In Create Account

ActiveState®

GET A DEMO



WHY ACTIVESTATE PRODUCTS SOLUTIONS PRICING RESOURCES

Click the Get Started button and choose Python 3.6 and the OS you're working in. In addition to the standard packages included in Python, we'll need to add Flask in order to create a REST API that can expose the blockchain and test it out.

## Creating The First Block

I will use the standard JSON format to store data in each block. The data for each block looks something like:

Copy

```
{
    "author": "author_name",
    "timestamp": "transaction_time",
    "data": "transaction_data"
}
```

To implement this in Python, we first create a block class with the aforementioned attributes. We also want to make each block unique in order to ensure that duplications do not occur:

Copy

```
class Block:
    def __init__(self, index, transactions, timestamp, previous
        self.index = index
        self.transactions = transactions
        self.timestamp = timestamp
        self.previous_hash = previous_hash
        self.nonce = nonce
```

Don't worry about the `previous_hash` or `nonce` variables for now (we'll discuss them later on).

As I mentioned, one of the characteristics of the data in each block is immutability, which can be implemented using a cryptographic hash function. This is a one-way algorithm that takes arbitrarily-sized input data (known as a key) and maps it onto values of fixed sizes (hash value). To illustrate why a hash function is useful for us, consider the following example:

1. Alice and Bob are racing to solve a difficult math problem

Contact Sales Sign In Create Account

ActiveState®

GET A DEMO

WHY ACTIVESTATE PRODUCTS SOLUTIONS PRICING RESOURCES

4. Alice now shares her answer with Bob so he can put it through the hash function and check to see if the resulting hash value matches the one that Alice initially provided him.
5. The hash values match, meaning that Alice did indeed solve the problem correctly before Bob.

Python can use any standard cryptographic hash function, such as those in the US National Security Agency's (NSA) set of SHA-2 functions. For example, SHA-256 can be implemented by adding a `compute_hash` method within the class block:

Copy

```
from hashlib import sha256
import json

class Block:
    def __init__(self, index, transactions, timestamp, previous
        self.index = index
        self.transactions = transactions
        self.timestamp = timestamp
        self.previous_hash = previous_hash
        self.nonce = nonce

    def compute_hash(self):
        block_string = json.dumps(self.__dict__, sort_keys=True
        return sha256(block_string.encode()).hexdigest()
```

Hashing each block ensures the security of each one individually, making it extremely difficult to tamper with the data within the blocks. Now that we've established a single block, we need a way to chain them together.

## Coding Your Blockchain

Let's create a new class for the blockchain. In order to ensure the immutability of the entire blockchain, we will use the clever approach of including a hash of the previous block within the current block. The awareness of all data within each block establishes a mechanism for protecting the entire chain's integrity (partially, at least). This is why we included the `previous_hash` variable in the block class. We also need a way to initialize the blockchain, so we define the `create_genesis_blockmethod`. This creates an initial block with an index of 0 and a previous hash of 0. We then add this to the list `chain` that keeps track of each block.

Contact Sales Sign In Create Account

ActiveState®

GET A DEMO

WHY ACTIVESTATE PRODUCTS SOLUTIONS PRICING RESOURCES

```
        self.unconfirmed_transactions = []
        self.chain = []
        self.create_genesis_block()

    def create_genesis_block(self):
        genesis_block = Block(0, [], time.time(), "0")
        genesis_block.hash = genesis_block.compute_hash()
        self.chain.append(genesis_block)
    @property
    def last_block(self):
        return self.chain[-1]
```

## A Proof-Of-Work System For Blockchain

The hashing that we've described so far only gets us part of the way there. As it stands, it is feasible for someone to modify a previous block in the chain and then recompute each of the following blocks to create another valid chain. We would also like to implement a way for users to come to a consensus on a single chronological history of the chain in the correct order in which the transactions were made. To solve this, Satoshi Nakamoto established a proof-of-work system.

The proof-of-work system makes it progressively more difficult to perform the work required to create a new block. This means that someone who modifies a previous block would have to redo the work of the block and all of the blocks that follow it. The proof-of-work system requires scanning for a value that starts with a certain number of zero bits when hashed. This value is known as a *nonce* value. The number of leading zero bits is known as the *difficulty*. The average work required to create a block increases exponentially with the number of leading zero bits, and therefore, by increasing the *difficulty* with each new block, we can sufficiently prevent users from modifying previous blocks, since it is practically impossible to redo the following blocks and catch up to others.

To implement this system, we can add a `proof_of_work` method in the blockchain class:

Copy

```
difficulty = 2
def proof_of_work(self, block):
        block.nonce =
        computed_hash = block.compute_hash()
        while not computed_hash.startswith('0' * Blockchain.dif
            block.nonce += 1
            computed_hash = block.compute_hash()
        return computed_hash
```

Contact Sales Sign In Create Account

ActiveState®

GET A DEMO

proof that satisfies the difficulty criteria we can add it to the chain. The process of performing the computational work within this system is commonly known as mining.

WHY ACTIVESTATE PRODUCTS SOLUTIONS PRICING RESOURCES

Copy

```
def add_block(self, block, proof):
        previous_hash = self.last_block.hash
        if previous_hash != block.previous_hash:
            return False
        if not self.is_valid_proof(block, proof):
            return False
        block.hash = proof
        self.chain.append(block)
        return True

def is_valid_proof(self, block, block_hash):
        return (block_hash.startswith('0' * Blockchain.difficul
                block_hash == block.compute_hash())

def add_new_transaction(self, transaction):
            self.unconfirmed_transactions.append(transaction)

def mine(self):
        if not self.unconfirmed_transactions:
            return False

        last_block = self.last_block

        new_block = Block(index=last_block.index + 1,
                          transactions=self.unconfirmed_transac
                          timestamp=time.time(),
                          previous_hash=last_block.hash)

        proof = self.proof_of_work(new_block)
        self.add_block(new_block, proof)
        self.unconfirmed_transactions = []
        return new_block.index
```

## REST API For Blockchain

Up to this point, I have explained the fundamentals of creating a blockchain:

1. Define a single block
2. Define a blockchain

Contact Sales Sign In Create Account

ActiveState®

GET A DEMO

API. Flask is a lightweight web application framework written for Python.

WHY ACTIVESTATE PRODUCTS SOLUTIONS PRICING RESOURCES

Copy

```
from flask import Flask, request
import requests

app =  Flask(__name__)

blockchain = Blockchain()
```

First, we define our web application and create a local blockchain. Then, we create an endpoint that allows us to send a query to display the relevant information of the blockchain.

Copy

```
@app.route('/chain', methods=['GET'])
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})
app.run(debug=True, port=5000)
```

Before we can query it, we first need to activate the blockchain application in the command prompt by running:

Copy

```
python3 Blockchain.py
```

You should see something like this:

Copy

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to qu
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 105-118-129
```

Then, in another shell, we can send a query with cURL by running:

Copy

```
curl  http://127.0.0.1:5000/chain
```

The output should contain all the information about the blockchain:

Pretty cool, right?! Note that the chain only has one block at this point (the genesis block). Feel free to take advantage of the mining function that we built to add more blocks to the chain.

## Build A Blockchain In Python: Summary

In this tutorial, we used Python to create an ongoing chain of hash-based proof-of-work.

- First, we established the concept of a block and a blockchain, including protocols for hashing each block and creating the first block.
- Then, we built a proof-of-work system and a way to add new blocks through mining.
- Finally, we created an application with Flask and queried it.

Since Satoshi Nakamoto first introduced these concepts in his (or her) white paper, many other cryptocurrencies have popped up, as have many more ideas for potential blockchain applications.

- All of the code used in this article can be found on my GitLab repository.
- Sign up for a free ActiveState Platform account so you can download the Blockchain runtime environment and build your very own blockchain

## Related blogs:

> How To Build a Recommendation Engine in Python

> How to Build a Digital Virtual Assistant in Python

+ How do you create a blockchain in Python?

+ Why is a Python blockchain the best choice?

+ Can I create my own blockchain with Python?

+ What is a blockchain in Python?

Share



© 2021 ActiveState Software Inc. All rights reserved. ActiveState®, ActivePerl®, ActiveTcl®, ActivePython®, Komodo®, ActiveGo™, ActiveRuby™, ActiveNode™, ActiveLua™, and The Open Source Languages Company™ are all trademarks of ActiveState.



Stay Up-To-Date On ActiveState News

Email Address

SIGN ME UP »

You can unsubscribe at any time. For more information, consult our Privacy Policy.

ActiveState