Building A Simple Blockchain Data Structure With Python

arjuna sky kok Follow
Apr 17, 2018 · 5 min read

Photo by Mateusz Dach from Pexels

Here, I am going to build a simple blockchain data structure which is the foundation of Bitcoin. This data structure only is not enough to build even a simple cryptocurrency. But we have to start somewhere.

Before building a blockchain data structure, I have to explain about hashing. Bitcoin uses SHA-256. Here how you can do it in Python:

```
>>> import hashlib
>>> hashlib.sha256(b”hello world”).hexdigest()
‘b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9’
```

That is how you do hashing (SHA-256) in Python. But what is actually hash? What does 256 in SHA-256 means actually?

Hashing is a process which you turn anything (as long as you can represent it as a string) into a fixed 256 bit string. In the previous example, the string “hello world” has a length 11. Actually the length of “hello world” is depended on how you count it. But for simplicity, we just count how many characters. That “hello world” string is turned into a fixed-size string, which is ‘b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9’. Say I hash another string with different length.

```
>>> import hashlib
>>> hashlib.sha256(b”I am the best president. Ever.”).hexdigest()
‘fba0e4cc233f9df81278130ed748a31a842bce8f911766b661f8a4f7ddff5341’
```

The string “I am the best president. Ever.” has a different length than “hello world”. But the output has the same length, which is about 64 characters. So any input will be turned into 64 random characters string. Even a string which has a 23 kilometers length will be turned into a 64 random characters string.

This is a hexadecimal string. That’s why it has 64 characters. If you turn it into a bit string, it will have a 256 characters length.

```
>>>
bin(0xb94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde
9)
‘0b1011100101001101001001111011100110010011010011010011111000001000101
1001010010111001010010110101111101101001111011010101111111010110001 0
01000010011101111111000110111101001010011100000001110111010010000100 0
10001111011110101100111000101110111111001101111101001’
```

That is why it is called SHA-256.

Now, there are some properties from hashing which is very important for Bitcoin. The first is called collision free. I mean if you are going to turn anything into a fixed 256 bit string, surely there will be more than 1 input that has the same output. The size of possible inputs is bigger than the size of possible output. Yes, that’s correct. But finding x and y where x is different than y and hash(x) is equal hash(y) is extremely hard (in SHA-256 case; some smart people have found collision in SHA-1).

So there is no apparent relation between the input and the output. Even you change the tiny bit of the input, the output will be totally different. This is the second property.

```
>>> hashlib.sha256(b”1").hexdigest()
‘6b86b273ff34fce19d6b804eff5a3f5747ada4eaa22f1d49c01e52ddb7875b4b’
>>> hashlib.sha256(b”2").hexdigest()
‘d4735e3a265e16eee03f59718b9b5d03019c07d8b6c51f90da3a666eec13ab35’
```

```
>>> hashlib.sha256(b”3").hexdigest()
‘4e07408562bedb8b60ce05c1decfe3ad16b72230967de01f640b7e4729b49fce’
>>> hashlib.sha256(b”11").hexdigest()
‘4fc82b26aecb47d2868c4efbe3581732a3e7cbcc6c2efb32062c08170a05eeb8’
```

So the only way to to find different inputs which have the same output, you need to test all combination of characters with different length. “abc”, “maybe a loooooong string”, “17”, etc. It’s totally impractical.

But is there any possibility that when you hash something, the output is already same as hashing of “hello world”? Or just any two different strings but have the same hash? Of course, there is. The question is how minuscule the probability is. There are around 2^{256} possibilities of the output of SHA-256. How big is 2^{256}? 115792089237316195423570985008687907853269984665640564039457584007913129639936. Or 1.15 e+77. Or roughly 10^{77}. If you think that number is big but not very big, I have a bad news for you. The total atom in observable universe (that is the universe that you can see up to 46 billions light years in any direction from your chair) is 10^{78} to 10^{82}. https://www.universetoday.com/36302/atoms-in-the-universe/

My computer has Nvidia Geforce 1080 Ti. It has 11.3 teraflops (tera = 10^{12}). Flop is floating operation. Hashing is integer operation. So it's apple to orange. But for simplicity, say hashing is an floating operation as well and requires 3000 operations per hash. So my graphic card can compute 3766666666 hash per second. To find a collision, described in birthday attack, we need only to compute 2^{128} hashes. Say every human on this planet has my graphic card and together we compute the collision attack. It takes:

```
>>> 2**128 / (7000000000 * 3766666666.6666665)
1.2905778770705631e+19
```

That number is longer than the age of universe (around 10^{17} seconds). https://www.space.com/24054-how-old-is-the-universe.html

To describe the hashing algorithm, it is quite a work. But someday I'll explain the code behind SHA-256. Now that you can comprehend the vastness of hashing, let's move on. Blockchain is like a linked list, a data structure known by many computer science students. Let's create a block. The first block in Bitcoin is called genesis block.

```
import hashlib, json

block_genesis = {
 'prev_hash': None,
 'transactions': [1, 3, 4, 2]
}
```

The transactions represents the… *well*, transactions. In Bitcoin, it will be like “Jason pays 2 btc to Mary Sue”, “Kylo Rein pays 10 btc to Yoda”. For simplicity, we just put normal integers.

We serialized the block so it can be hashed.

```
block_genesis_serialized = json.dumps(block_genesis,
sort_keys=True).encode('utf-8')
block_genesis_hash =
hashlib.sha256(block_genesis_serialized).hexdigest()
```

Now we have another block.

```
block_2 = {
 'prev_hash': block_genesis_hash,
 'transactions': [3, 3, 3, 8, 7, 12]
}
```

We hash the block 2.

```
block_2_serialized = json.dumps(block_2, sort_keys=True).encode('utf-
8')
block_2_hash = hashlib.sha256(block_2_serialized).hexdigest()
```

We build another block.

```
block_3 = {
 'prev_hash': block_2_hash,
 'transactions': [3, 4, 4, 8, 34]
}
```

We hash the block 3. This will be the last block, I promise.

```
block_3_serialized = json.dumps(block_3, sort_keys=True).encode('utf-
8')
block_3_hash = hashlib.sha256(block_3_serialized).hexdigest()
```

To make sure that data has not been tampered, I only need to check the last block's hash, instead of checking all the data from genesis block to the last block. If it is different, than someone tried to tamper the data.

```
import hashlib, json

block_genesis = {
 'prev_hash': None,
 'transactions': [1, 3, 4, 2]
}

block_2 = {
 'prev_hash': None,
 'transactions': [3, 3, 3, 8, 7, 12]
}

block_3 = {
 'prev_hash': None,
 'transactions': [3, 4, 4, 8, 34]
}

def hash_blocks(blocks):
 prev_hash = None
 for block in blocks:
  block['prev_hash'] = prev_hash
  block_serialized = json.dumps(block, sort_keys=True).encode('utf-
8')
  block_hash = hashlib.sha256(block_serialized).hexdigest()
  prev_hash = block_hash

 return prev_hash
```

```
print(“Original hash”)
print(hash_blocks([block_genesis, block_2, block_3]))

print(“Tampering the data”)
block_genesis[‘transactions’][0] = 3

print(“After being tampered”)
print(hash_blocks([block_genesis, block_2, block_3]))
```

The result:

```
Original hash
45eda4f7a76bf0f92a0acda2ce4752dfbe167473376f766f22d7ec68501cac40
Tampering the data
After being tampered
27d68dae05428be6aa244869196a481f431fca6645dd33c3df7a740afa03b7d9
```

This is the basic of the blockchain. But this is not enough. How do you decide the next block to be added? You need consensus and proof of work. Then the block structure in Blockchain is much more complicated. We’ll cover that in next articles.