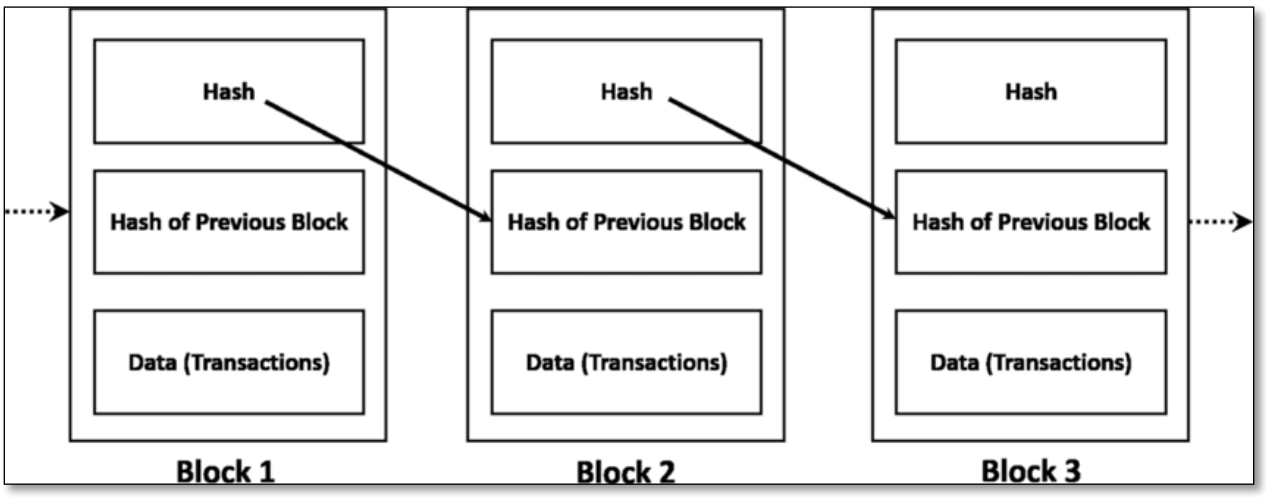
Python Blockchain

Contents

Time required: 60 minutes

What exactly is a blockchain? The best way to understand it is to see one in action , or better yet, to build one. Essentially a blockchain is a way to store data with its history — and a blockchain can be built in any programming language.

Blockchain



A blockchain consists of several key components:

- **Distributed Ledger**: This is a decentralized database that stores all the transactions that have occurred on the blockchain.
- **Blocks**: Each block contains a set of transactions, and each block is linked to the previous block through a cryptographic hash.
- **Cryptographic Hash**: This is a unique digital signature that is created for each block. It is used to verify the integrity of the block and to link it to the previous block.
- **Consensus Mechanism**: This is a process by which the network participants in the blockchain network reach agreement on the validity of a new block that is added to the chain.
- **Mining**: This is the process by which network participants compete to add new blocks to the chain. Miners are rewarded with cryptocurrency for successfully adding a new block.

Step 1: Put Some Data in a Block

The starting point for a blockchain is actually simple.

A blockchain has blocks like this:
[ ]
With data in them:
[data]

Our blocks are going to have transaction data in them. The transaction will be a sentence saying what happened. We'll model our simple blockchain on Bitcoin. Our first fictional transaction is a payment of 1 bitcoin from one wallet (Wallet 1) to another wallet (Wallet 2).

Our first block - known as the genesis block - looks like this.

```python
"""
    Name: blockchain_1.py
    Author:
    Created: 02/10/2024
    Purpose: Demonstrate blockchain in Python
"""

# Start with a basic block
# Define a transaction where Wallet 1 pays 1 bitcoin to Wallet 2
transaction_1 = "Wallet 1 paid 1 bitcoin to Wallet 2"

# Create a genesis block containing the transaction
genesis_block = (transaction_1)

# Print the contents of the genesis block
print(genesis_block)
```

Example run:

```
Wallet 1 paid 1 bitcoin to Wallet 2
```

Step 2: Prepare the Chains

We need something to chain blocks together.

To do this we use a **hash.** A hash is a one-way cryptographic algorithm that creates a particular value from any input. The value is one-way meaning it cannot be decrypted. Use a strong enough hash method and two different inputs will practically never lead to the same hash value.

A hash is the key ingredient to a blockchain.

1. Each block gets a hash.
2. Each block also stores the hash of the prior block.
3. The hash for each block is derived from the other data that will go in the block.
4. The hash itself is added onto the block.

Our block data structure is now:
[prior block hash, data, block hash]

The thing to remember here is that a practically unique code (the hash) is being created for each block that everyone on the network can use to verify blocks as they receive them.

Python has a built-in method: hash(). But it won't work for a blockchain as it produces different results in each session. We need a stable hash so let's import a Python library that has the same hashing algorithm Bitcoin uses, it's called SHA-256. We'll create our own hashing function from it.

Modify the program with the following code.

```
"""
    Name: blockchain_2.py
    Author:
    Created: 02/10/2024
    Purpose: Demonstrate blockchain in Python
"""
# Python hashing library
import hashlib
```

Import the built-in Python hashlib library.

```
# --------------------------- BIT HASH ----------------------------------- #
CodiumAI: Options | Test this function
def bit_hash(data):

    # Convert the input data to a string
    string_data = str(data)

    # Encode the string data into bytes using UTF-8 text encoding
    formatted_data = string_data.encode('utf-8')

    # Compute the SHA-256 hash of the formatted data
    bit_hash = hashlib.sha256(formatted_data)

    # Convert it to a hexadecimal representation
    bit_hash = bit_hash.hexdigest()

    # Return the resulting hexadecimal hash
    return bit_hash
```

Step 3: Add Chain Links to Our Block

Let's get the hashes for our block and add them to our new data structure:

[prior block hash, data, block hash]

There is no prior block for the genesis block, we use zero for the prior block hash.

Replace all code after the bit_hash function.

```
# Start with a basic block
# Define a transaction where Wallet 1 pays 1 bitcoin to Wallet 2
transaction_1 = "Wallet 1 paid 1 bitcoin to Wallet 2"

# Create a genesis block containing the transaction hash
genesis_block_hash = bit_hash((0, transaction_1))

# Print the contents of the genesis block
print(genesis_block_hash)
```

Example run shows the hash created for our genesis block.

```
3297da8cca590c96a5e120b68daddf1774af7256c978f6e6503d8b6638b1223a
```

Let's create our genesis block, this time complete with hashes for our chain.

```
# Recreate the genesis block - this time with hashes for our chain
genesis_block = (0, transaction_1, genesis_block_hash)

# Print the complete genisis block
print(genesis_block)
```

The complete genesis block combines the transaction with the hash.

```
(0, 'Wallet 1 paid 1 bitcoin to Wallet 2', '3297da8cca590c96a5e120b68daddf1774af7256c978f6e6503d8b6638b1223a')
```

Step 4: Create the Next Block

Our blockchain is ready for chaining - let's create the second block.

Add this code snippet below the previous one:

```
# Create the transaction for block 2
transaction_2 = "Wallet#1 paid 2 bitcoin to Wallet#2"

# Create the block 2 hash which combines the previous hash
block_2_hash = bit_hash((genesis_block_hash, transaction_2))

# The block now holds all the transactions details and hash chains
block_2 = (genesis_block_hash, transaction_2, block_2_hash)

print(block_2)
```

Example run:

```
3297da8cca590c96a5e120b68daddf1774af7256c978f6e6503d8b6638b1223a
(0, 'Wallet 1 paid 1 bitcoin to Wallet 2', '3297da8cca590c96a5e120b68daddf1774af7256c978f6e6503d8b6638b1223a'
)
('3297da8cca590c96a5e120b68daddf1774af7256c978f6e6503d8b6638b1223a', 'Wallet#1 paid 2 bitcoin to Wallet#2', '
a1e0f228b1194d84bcbb56897a7245c7f95700c580d68e7acfa711be76ceae6f')
```

We can see the pattern for creating the next block — all we need to do is create its hash and then create the new block with new transaction data.

Step 5: Create New Blocks

Add this code snippet below the **bit_hash** function.

```python
# ---------------------------- CREATE BLOCK ---------------------------------#
CodiumAI: Options | Test this function
def create_block(prior_block_hash, prior_block_number, transaction_data):
    """ function named 'create_block' with parameters for prior block hash,
    prior block number, and transaction data
    """
    # Calculate the new block number by incrementing the prior block number
    block_number = prior_block_number + 1

    # Generate a new block hash using the 'bit_hash' function with the
    # concatenation of prior block hash, block number, and transaction data
    block_hash = bit_hash((prior_block_hash, block_number, transaction_data))

    # Create a new block tuple containing prior block hash, block number,
    # transaction data, and the newly calculated block hash
    new_block = (prior_block_hash, block_number, transaction_data, block_hash)

    # Return the newly created block
    return new_block
```

Step 6: Print Blocks

We'll now use our function to mine, I mean print blocks to the terminal.

We follow the programming convention of starting our block numbers at zero rather than one (computers do work in binary 0/1 after all.)

Add this after the two functions replacing all the code.

```python
# Create genesis block using the 'create_block' function with initial values
# Mumbering starts at 0
genesis_block = create_block(0, -1, "Wallet 1 paid 1 bitcoin to Wallet 2")

# Print the details of the genesis block
print("Genisys Block:")
print(genesis_block)
print()

# Initialize the prior_block variable with the genesis block for the loop
prior_block = genesis_block

# Iterate through the loop to generate and print 3 additional blocks
for i in range(3):
    print(f"Block {i + 1}:")
    # Extract prior block hash and block number from the prior_block tuple
    prior_block_hash = prior_block[3]
    prior_block_number = prior_block[1]

    # Extracted prior block hash and block number are used as inputs
    # Update the transaction data to represent a payment from
    # Wallet 1 to Wallet 2, with an incremented bitcoin amount
    next_block = create_block(
        prior_block_hash, prior_block_number, "Wallet 1 paid " +
        str(i + 2) + " bitcoin to Wallet 2"
    )

    # Print the details of the newly generated block
    print(next_block)
    print()
    # Update the prior_block variable for the next iteration
    prior_block = next_block
```

Example run:

```
Genisys Block:
(0, 0, 'Wallet 1 paid 1 bitcoin to Wallet 2', 'd3ea58cdb234a0024d5a176dbc
1f3c559aabae5ff192b95eb2b9b85853ee4a1d')

Block 1:
('d3ea58cdb234a0024d5a176dbc1f3c559aabae5ff192b95eb2b9b85853ee4a1d', 1, '
Wallet 1 paid 2 bitcoin to Wallet 2', '006b32145f5cae02f7fff49e4a114b5127
de172f4e77e3fec65efc0d3233bec7')

Block 2:
('006b32145f5cae02f7fff49e4a114b5127de172f4e77e3fec65efc0d3233bec7', 2, '
Wallet 1 paid 3 bitcoin to Wallet 2', '2c0287df987838e744c8513af1907a61ef
089fc1948098b53a36178d8610fc1a')

Block 3:
('2c0287df987838e744c8513af1907a61ef089fc1948098b53a36178d8610fc1a', 3, '
Wallet 1 paid 4 bitcoin to Wallet 2', '74ea42440b65f1161d019bf3956e01120c
046b3f5c6017b17f07c95e3bdbaa76')
```

Great — the blocks are chaining together. The hash of each block becomes the prior hash of the next block.

Anyone can verify a block as valid and that it chains back to the prior block, also meaning that it is consistent with the entire history of all prior blocks.

Assignment Submission

1. Attach the code.
2. Attach a screenshot showing a successful run of the program.
3. Submit the assignment in Blackboard.