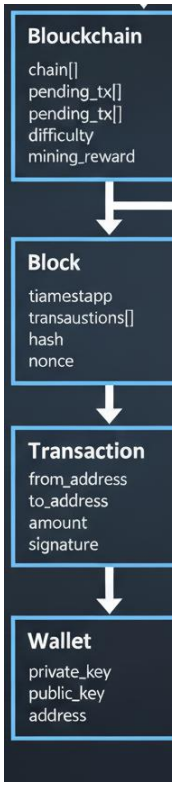
Title:

Block Chain Implementation in Python

By:

Samar Arafa – 220240333

Contents



Overall Code Structure

The code is designed in a modular structure where each class is a fundamental component of the blockchain system. The code structure divided into the following sections:

1. Import Libraries

The code starts with the import of the required libraries necessary to implement the core functionalities of the program.

- Cryptographic libraries (SHA-256 and ECDSA)
- Managing Time : Timings
- Digital signature verification
- Type hinting support

```
pip install cryptography
```

```
Requirement already satisfied: cryptography in /usr/local/lib/python3
Requirement already satisfied: cffi>=1.12 in /usr/local/lib/python3.1
Requirement already satisfied: pycparser in /usr/local/lib/python3.12
```

```
import hashlib
import time
import json
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.exceptions import InvalidSignature
from typing import List, Optional
```

Figure 1Import Libraries

2. Wallet Class

This class represents a user's identity in this system and includes :

- Asymmetric cryptography - generate a pair of public and private keys
- Deriving wallet address from public key
- Signature of data and transactions by private key

```python
# 1. Wallet Class (to create public/private keys)
#     The Wallet class represents a user's digital wallet.
#     It is responsible for generating and managing cryptographic
#     key pairs (private key and public key) using Elliptic Curve Cryptography.
class Wallet:
    def __init__(self):
        # Generate a private key using the SECP256K1 elliptic curve
        self.private_key = ec.generate_private_key(ec.SECP256K1())
        # Derive the corresponding public key from the private key
        self.public_key = self.private_key.public_key()

    def get_address(self):
        # Convert the public key to a string (wallet address).
        pub_bytes = self.public_key.public_bytes(
            encoding=serialization.Encoding.X962,
            format=serialization.PublicFormat.UncompressedPoint
        )
        return pub_bytes.hex()

    def sign(self, data):
        # Sign data using the private key.
        return self.private_key.sign(data.encode(), ec.ECDSA(hashes.SHA256()))

    def get_private_key(self):
        return self.private_key
```

Figure 2 Wallet Class

This class therefore forms the basis of transaction ownership proof and securing.

3. Transaction Class

This class manages financial transactions in the network and consists of:

- Defining Sender, Receiver, and Transfer Amount
- Calculation of transaction hash
- Digitally signing the transaction
- Verifying the transaction signature
- Converting transaction data into a format suitable for storage within a block

```python
class Transaction:
    # The Transaction class represents a transfer of value between two addresses.
    # Each transaction must be cryptographically signed by the sender
    # to ensure integrity and authorization.
    def __init__(self, from_address, to_address, amount):
        self.from_address = from_address
        self.to_address = to_address
        self.amount = amount
        self.signature = None

    def calculate_hash(self):
        # Generate a SHA-256 hash of the transaction data
        data = f"{self.from_address}{self.to_address}{self.amount}"
        return hashlib.sha256(data.encode()).hexdigest()

    def sign_transaction(self, wallet):
        # Sign the transaction using the wallet's private key
        if self.from_address != wallet.get_address():
            raise Exception("You cannot sign transactions for other wallets!")
        tx_hash = self.calculate_hash()
        self.signature = wallet.sign(tx_hash)

    def is_valid(self):
        # Check if the transaction is valid (signed correctly
        if self.from_address is None:  # Mining reward transaction
            return True
```

Figure 3Transaction Class

This ensures transactions cannot be forged or executed without authorization.

4. Block Class

A block consists of a set of validated transactions, which are cryptographically tied to the preceding block. In blockchain technology, a block represents the basic unit, which consists of a set of transactions and is linked to the preceding block

This class describes the main storage unit in the chain, where:

- It contains a list of confirmed transactions
- It is connected to the previous block via its hash
- It uses the Proof of Work mechanism
- Validates all transactions inside the block

```python
class Block:
    # The Block class represents a block in the blockchain.
    # Each block contains a list of transactions and is linked
    # to the previous block through a cryptographic hash.
    def __init__(self, timestamp, transactions):
        self.timestamp = timestamp
        self.transactions = transactions
        self.previous_hash = ""
        self.hash = ""
        self.nonce = 0
        self.hash = self.calculate_hash()

    def calculate_hash(self):
        # Calculate SHA-256 hash of the block
        data = str(self.timestamp) + str([tx.to_dict() for tx in self.transactions]) + self.previous_hash + str(self.nonce)
        return hashlib.sha256(data.encode()).hexdigest()

    def mine_block(self, difficulty):
        # Proof of Work: mine the block until the hash has required leading zeros
        target = "0" * difficulty
        while self.hash[:difficulty] != target:
            self.nonce += 1
            self.hash = self.calculate_hash()
        print(f"Block mined: {self.hash}")

    def has_valid_transactions(self):
        # Check if all transactions in this block are valid
```

Figure 4Block Class

Within each block, the following components are included:

- **Timestamp:** Creation time of the block
- **Transactions:** List of transactions (sender, receiver, amount, and digital signature)
- **Previous block hash :** Connecting the block to the previous block to make the chain continuous
- **Block hash:** A unique digital fingerprint created using SHA-256 of the data within the block
- **Nonce:** Number which is changed during mining to meet its Proof of Work requirement

Any change in the block content even a single character makes the entire block's hash will change, and any attempt to tamper would be immediately traceable

5. Blockchain Class

This class manages the entire chain and includes:

- Creating the Genesis Block
- Storing all blocks in chronological order
- Handling pending transactions
- Performing mining operations
- Wallet Balance Calculations
- Verifying the integrity of the entire chain

```python
class Blockchain:
    # The Blockchain class manages the entire blockchain system,
    # including block creation, mining, transaction handling, and chain validation.
    def __init__(self, difficulty: int = 2):
        self.chain = [self.create_genesis_block()]
        self.difficulty = difficulty
        self.pending_transactions = []
        self.mining_reward = 100.0

    def create_genesis_block(self):
        # Create the first block in the chain (Genesis Block
        return Block(time.time(), [])

    def get_latest_block(self):
        return self.chain[-1]

    def add_transaction(self, transaction):
        # Add a new transaction to pending transactions
        if not transaction.from_address or not transaction.to_address:
            raise Exception("Transaction must include from and to address.")
        if not transaction.is_valid():
            raise Exception("Cannot add invalid transaction to chain.")
        self.pending_transactions.append(transaction)

    def mine_pending_transactions(self, mining_reward_address):
        # Mine a block containing pending transactions and reward the miner
        block = Block(time.time(), self.pending_transactions)
        block.previous_hash = self.get_latest_block().hash
```

Figure 5Blockchain Class

How the flow works:

1. The user will create a Wallet and receive an address and a private key.
2. The user creates a Transaction, then signs it using their wallet.
3. The transaction is added to the pending transaction pool in the Blockchain object.
4. During mining:
    - A new Block is created containing the pending transactions
    - Proof of Work is applied until a valid hash is found
    - The block is added to the chain
    - A mining reward(100 coins) is granted to the miner (as a pending transaction), it only appears in the miner's balance after the next block is mined.

Example:

During a test run, the following output was observed:

```
  Mining the first block...
Block mined: 00e906df9bc2418f21e39069ebfa830b751f6622d897b4efa858e57016269695
 Balance of Wallet 1: -10.0
```

Here, Wallet 1 sent 10 coins before any mining occurred. Since the mining reward hadn't been processed yet, the balance appeared as -10.0.

After mining a second block:

```
Mining the second block...
Block mined: 00b99d1c61bb3a17f5417e58cc35d609da45b40162f93cb0ca0e44f8dabbfe5c
Balance of Wallet 1: 90.0
```

The reward from the first block (100 coins) was now applied, resulting in a final balance of 90.0 (-10 + 100).

5. Validation at any stage can be done via:
    - tx.is_valid() : checks the signature
    - block.has_valid_transactions() : validates all transactions in the block
    - blockchain.is_chain_valid() : verifies the entire blockchain

Proof of Work

Proof of work is done by ensuring that the hash value begins with a certain number of zeros (such as “00” or “0000”). The nonce value is incremented again and again until a hash satisfying the condition is obtained

It is a time-consuming and computationally intensive process that prevents the formation of random/fraudulent blocks and assures that the addition of a new block is not easy. Level of difficulty (amount of leading zeros) can be varied to regulate rate of blocks production

Blockchain Validation

The is_chain_valid() function checks the integrity of the blockchain by performing a series of sequential checks that makes sure each block is linked correctly and has not been tampered with.

- **Transaction validation:** This checks that each block’s set of transactions are valid using has_valid_transactions(), which verifies the digital signatures (using the ECDSA algorithm on the secp256k1 elliptic curve).
- **Hash consistency:** The stored hash in each block is compared against a recalculated hash based on the block’s current contents to detect tampering.
- **Chain linking:** This ensures that in every block, the previous_hash accurately refers to the hash of the very previous block.
- **Proof of Work verification:** Each block's hash is checked for the required number of leading zeros based on the difficulty level.

Changing any previous transaction, amount for example, will lead to a hash mismatch or a verification failure of a signature, making the blockchain invalid. As confirmed by the final test:

Change amount:

```
# Attempt tampering
coin.chain[1].transactions[0].amount = 999
print(f" After tampering, is blockchain valid? {coin.is_chain_valid()}")
```

Output:

```
 After tampering, is blockchain valid? False
```

Challenges Faced

- **The use of cryptocurrency:** Python does not have a native implementation for the secp256k1 curve used in the Bitcoin cryptocurrency; therefore, an additional library (cryptography) had to be applied. It was necessary to learn how to convert the keys to a format that was easy to store and compare.
- **Hash consistency**: While hashing the transactions into a string for hashing, it was necessary that a consistent format be maintained (for example, use json instead of a string), since a slight variation in the format would result in a totally different hash.

Example of System Execution:

The following code demonstrates the creation of wallets, adding a transaction, mining blocks, and verifying blockchain integrity:

```
wallet1 = Wallet()
wallet2 = Wallet()
addr1 = wallet1.get_address()
addr2 = wallet2.get_address()

print(" Wallet 1:", addr1[:30], "...")
print(" Wallet 2:", addr2[:30], "...")

# Create blockchain
coin = Blockchain(difficulty=2)

# Create and sign a transaction
tx = Transaction(addr1, addr2, 10)
tx.sign_transaction(wallet1)
coin.add_transaction(tx)

print("  Mining the first block...")
coin.mine_pending_transactions(addr1)

print(f" Balance of Wallet 1: {coin.get_balance_of_address(addr1)}")
print(f" Is blockchain valid? {coin.is_chain_valid()}")
print("  Mining the second block...")
coin.mine_pending_transactions(addr1)
print(f" Balance of Wallet 1: {coin.get_balance_of_address(addr1)}")

# Attempt tampering
coin.chain[1].transactions[0].amount = 999
print(f" After tampering, is blockchain valid? {coin.is_chain_valid()}")
```

Output:

```
 Wallet 1: 0408ad8c511221e498ea79fa1cb17f ...
 Wallet 2: 0480e5cf4239659d29d1e025b21393 ...
  Mining the first block...
Block mined: 00e906df9bc2418f21e39069ebfa830b751f6622d897b4efa858e57016269695
 Balance of Wallet 1: -10.0
 Is blockchain valid? True
  Mining the second block...
Block mined: 00b99d1c61bb3a17f5417e58cc35d609da45b40162f93cb0ca0e44f8dabbfe5c
 Balance of Wallet 1: 90.0
 After tampering, is blockchain valid? False
```

Analysis:

- Two wallets were created; each has a distinct address.
- 10 units were transmitted from Wallet 1 to Wallet 2. The transaction had been correctly signed.
- The first block was mined with the Proof of Work method resulting in a balance of -10 for Wallet 1 before the reward for mining the first block.
- After the second block of data was mined, the reward amount was added to the balance of Wallet 1, and the balance became 90.

- Any modification made within a previous transaction will render the chain verification process invalid, thereby ensuring the effectiveness of the blockchain technology against any data tampering.