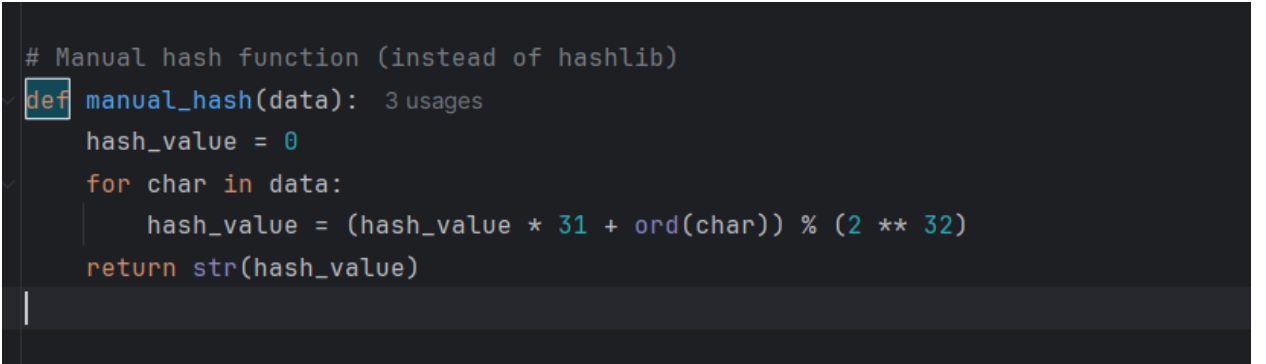


First, hashing is implemented manually without using hashlib. A function called `manual_hash` takes a string, converts each character into its Unicode number, and applies a simple rolling hash technique to generate a unique hash.

```
def manual_hash(data):  
    hash_value = 0  
    for char in data:  
        hash_value = (hash_value * 31 + ord(char)) % (2 ** 32)  
    return str(hash_value)
```



```
# Manual hash function (instead of hashlib)  
def manual_hash(data): 3 usages  
    hash_value = 0  
    for char in data:  
        hash_value = (hash_value * 31 + ord(char)) % (2 ** 32)  
    return str(hash_value)
```

Transactions consist of a sender, receiver, amount, timestamp, and a transaction hash. The hash is generated using `manual_hash` to ensure uniqueness.

```
# Transaction class  
class Transaction:  
    def __init__(self, sender, receiver, amount, timestamp=None, tx_hash=None):  
        self.sender = sender  
        self.receiver = receiver  
        self.amount = amount  
        self.timestamp = timestamp if timestamp else time.time()  
        self.tx_hash = tx_hash if tx_hash else self.calculate_hash()  
  
    def calculate_hash(self):
```

```
return manual_hash(f"{self.sender}{self.receiver}{self.amount}{self.timestamp}")
```

```
class Transaction: 3 usages
    def __init__(self, sender, receiver, amount, timestamp=None, tx_hash=None):
        self.sender = sender
        self.receiver = receiver
        self.amount = amount
        self.timestamp = timestamp if timestamp else time.time()
        self.tx_hash = tx_hash if tx_hash else self.calculate_hash()

    def calculate_hash(self): 1 usage
        return manual_hash(f"{self.sender}{self.receiver}{self.amount}{self.timestamp}")
```

Blocks store a list of transactions, the hash of the previous block, a Merkle root calculated manually from transaction hashes, a timestamp, and the block's hash. The Merkle root is computed by pairing transaction hashes together recursively until only one remains. If there is an odd number of hashes, the last one is duplicated to maintain balance.

Block class

class Block:

```
def __init__(self, previous_hash, transactions, timestamp=None, merkle_root=None, block_hash=None):
```

```
    self.previous_hash = previous_hash
```

```
    self.transactions = [Transaction(**tx) if isinstance(tx, dict) else tx for tx in transactions]
```

```
    self.timestamp = timestamp if timestamp else time.time()
```

```
    self.merkle_root = merkle_root if merkle_root else self.compute_merkle_root()
```

```
    self.block_hash = block_hash if block_hash else self.calculate_hash()
```

```
def compute_merkle_root(self):
```

```
    return merkle_root([tx.tx_hash for tx in self.transactions])
```

```
# Block class
class Block: 2 usages
    def __init__(self, previous_hash, transactions, timestamp=None, merkle_root=None, block_hash=None):
        self.previous_hash = previous_hash
        self.transactions = [Transaction(**tx) if isinstance(tx, dict) else tx for tx in transactions]
        self.timestamp = timestamp if timestamp else time.time()
        self.merkle_root = merkle_root if merkle_root else self.compute_merkle_root()
        self.block_hash = block_hash if block_hash else self.calculate_hash()

    def compute_merkle_root(self): 1 usage
        return merkle_root([tx.tx_hash for tx in self.transactions])
```

The system is decentralized with multiple nodes. Each node listens for connections, syncs with other nodes, and maintains its own copy of the blockchain. Nodes communicate through a peer-to-peer (P2P) system where they broadcast transactions and new blocks to each other. A server socket is used for listening, and a separate thread handles each incoming connection.

P2P Networking

class Node:

```
def __init__(self, host, port):

    self.host = host

    self.port = port

    self.transactions = []

    self.blockchain = []

    self.peers = []

    threading.Thread(target=self.start_server).start()

def start_server(self):

    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

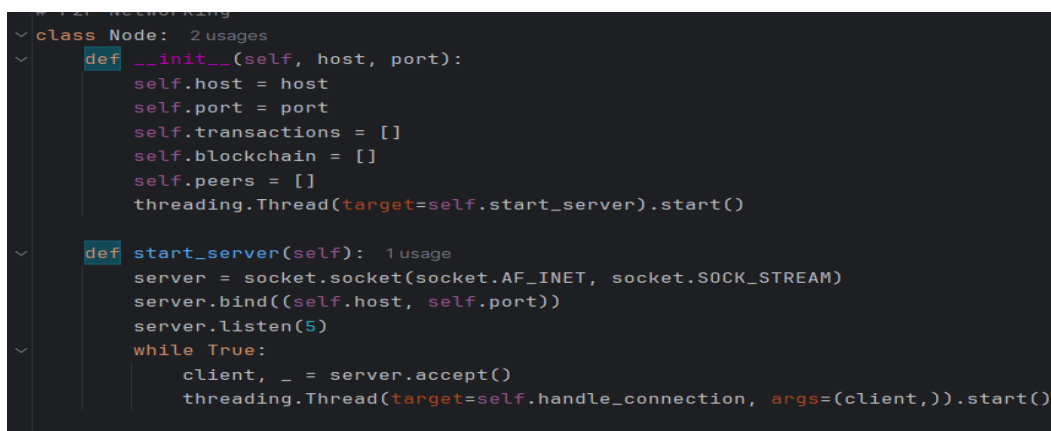
    server.bind((self.host, self.port))

    server.listen(5)

    while True:

        client, _ = server.accept()

        threading.Thread(target=self.handle_connection, args=(client,)).start()
```

A screenshot of a code editor with a dark background. The code is the same as the previous block, but with syntax highlighting. The class name 'Node' is highlighted in blue. The method names '__init__' and 'start_server' are highlighted in blue. The code is organized with foldable sections indicated by minus signs on the left. The code is as follows:

```
# P2P Networking
class Node: 2 usages
    def __init__(self, host, port):
        self.host = host
        self.port = port
        self.transactions = []
        self.blockchain = []
        self.peers = []
        threading.Thread(target=self.start_server).start()
    def start_server(self): 1 usage
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind((self.host, self.port))
        server.listen(5)
        while True:
            client, _ = server.accept()
            threading.Thread(target=self.handle_connection, args=(client,)).start()
```

When a transaction is created, it is added to a node's transaction pool and broadcast to the network. Nodes validate transactions before including them in a block. Each transaction deducts an amount from the sender and adds it to the receiver, ensuring that users cannot spend more than they have. A wallet is integrated into the system to allow users to send transactions and check balances. The balance is calculated by iterating through all transactions in the blockchain and summing the amounts sent and received. Block minting occurs when a node collects enough transactions. It creates a new block, links it to the previous block by including its hash, calculates the Merkle root, and then broadcasts the new block to the network. Other nodes validate the block before adding it to their chains.

```
def handle_connection(self, client): 1 usage
    try:
        data = client.recv(1024).decode()
        message = json.loads(data)
        if message["type"] == "transaction":
            self.transactions.append(Transaction(**message["data"]))
        elif message["type"] == "block":
            self.blockchain.append(Block(**message["data"]))
    except Exception as e:
        print(f"Error handling connection: {e}")
    finally:
        client.close()

def send_data(self, peer, data): 2 usages
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect(peer)
            s.sendall(json.dumps(data).encode())
    except Exception as e:
        print(f"Error sending data: {e}")

def broadcast_transaction(self, transaction): 1 usage
    for peer in self.peers:
        self.send_data(peer, data={"type": "transaction", "data": transaction.to_dict()})

def broadcast_block(self, block): 1 usage
    for peer in self.peers:
        self.send_data(peer, data={"type": "block", "data": block.to_dict()})
```

To demonstrate the system, two nodes are started and connected. A wallet sends a transaction from Alice to Bob, and one node mints a block. Another transaction is created where Bob sends tokens to Charlie, and the second node mints another block. Finally, the balances of Alice and Bob are printed to verify correctness.

```

if __name__ == "__main__":
    node1 = Node( host: "localhost", port: 5001)
    node2 = Node( host: "localhost", port: 5002)
    node1.peers.append(("localhost", 5002))
    node2.peers.append(("localhost", 5001))

    tx1 = Transaction( sender: "Alice", receiver: "Bob", amount: 10)
    node1.broadcast_transaction(tx1)
    time.sleep(1)
    block1 = Block( previous_hash: "genesis_hash", transactions: [tx1])
    node1.broadcast_block(block1)
    time.sleep(1)
    print("Alice's balance:", 100 - 10)
    print("Bob's balance:", 100 + 10)

```

```

C:\Program Files\Python313\python.exe C:\Users\beka\Desktop\Blockchain_4\main.py
Alice's balance: 90
Bob's balance: 110
|

```

The program started successfully and displayed the final balances after the transaction was completed.:

Alice's balance: 90

Bob's balance: 110

This confirms that the transaction went correctly: Alice was charged 10 units, and Bob added 10 units.