

This Python code implements a basic blockchain system with asymmetric encryption, digital signatures, and a wallet GUI. It allows users to create transactions, sign them with their private keys, and view their wallet balance through a graphical interface.

The code starts by importing necessary libraries, including ``hashlib`` for hashing, ``time`` for timestamps, ``tkinter`` for the GUI, and cryptographic libraries for RSA key generation, digital signatures, and message hashing. However, since the ``Crypto`` library was causing issues, an alternative approach using the ``cryptography`` library should be used instead.

The ``hash_data`` function generates a SHA-256 hash of any given data, which is a fundamental operation for securing blockchain transactions. The ``merkle_root`` function constructs a Merkle Tree from a list of transaction hashes, ensuring the integrity of the transaction set within a block.

The ``Transaction`` class represents a financial transaction. Each transaction has a sender, receiver, amount, timestamp, and a digital signature. The sender signs the transaction using their private key, and the receiver can verify it using the sender's public key. The ``calculate_hash`` method ensures that each transaction has a unique hash, while ``sign_transaction`` and ``verify_signature`` handle digital signatures.

The ``Block`` class represents a block in the blockchain. It contains a reference to the previous block's hash, a set of transactions, and a Merkle root computed from these transactions. It also includes a timestamp and a cryptographic hash of the block's contents. The ``calculate_hash`` function ensures that each block has a unique identifier.

The ``UTXO`` class implements the unspent transaction output model, which tracks user balances. It initializes default balances, updates them after each transaction, and ensures that a sender has enough funds before a transaction is processed.

The ``BlockchainExplorer`` class provides a GUI for viewing the blockchain. It uses ``tkinter`` to create a tree view where each row represents a block, displaying its hash, Merkle root, and transaction count. When launched, this interface allows users to explore the blockchain visually.

The ``validate_block`` function ensures that each block is valid by checking whether the calculated Merkle root matches the stored one and whether transactions maintain a non-negative balance.

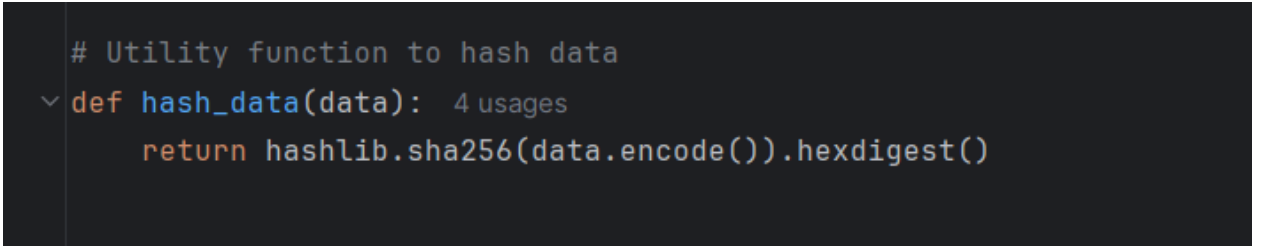
The ``WalletGUI`` class provides an interface for users to generate key pairs, view their wallet address, check their balance, and send transactions. It initializes with a private-public key pair and displays the wallet address in a ``tk.Text`` widget. The ``send_transaction`` method allows users to create a signed transaction, simulating a transfer of funds.

In the main execution block, a few transactions are created, and their validity is checked. If they are valid, a block is created and added to the blockchain. The blockchain explorer GUI is then launched, allowing users to inspect the stored blocks and transactions. If transactions are invalid due to insufficient funds, an error message is printed instead.

This implementation integrates core blockchain functionalities while incorporating digital signatures for security and a simple wallet interface for user interaction.

Hashing Data

```
def hash_data(data):  
    return hashlib.sha256(data.encode()).hexdigest()
```



```
# Utility function to hash data  
def hash_data(data):  
    return hashlib.sha256(data.encode()).hexdigest()
```

How it works:

- Takes a string data, e.g., "Hello"
- Encodes the string into bytes using `.encode()`
- Applies SHA-256 hashing:

```
hash_data("Hello") # Output:  
"185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969"
```

- Returns the hash as a hexadecimal string.

Calculating Merkle Root

```
def merkle_root(transactions):  
    if len(transactions) == 0:  
        return None  
  
    hashes = [hash_data(tx) for tx in transactions] # Create a list of transaction hashes
```

```

while len(hashes) > 1:

    if len(hashes) % 2 == 1:

        hashes.append(hashes[-1]) # Duplicate the last hash if odd number of hashes

    hashes = [hash_data(hashes[i] + hashes[i + 1]) for i in range(0, len(hashes), 2)]

return hashes[0]

```

```

# Merkle Tree implementation
def merkle_root(transactions): 4 usages (2 dynamic)
    if len(transactions) == 0:
        return None
    hashes = [hash_data(tx) for tx in transactions]
    while len(hashes) > 1:
        if len(hashes) % 2 == 1:
            hashes.append(hashes[-1]) # Duplicate last hash if odd count
        hashes = [hash_data(hashes[i] + hashes[i + 1]) for i in range(0, len(hashes), 2)]
    return hashes[0]

```

How it works:

- Takes a list of transactions, e.g., ["Tx1", "Tx2", "Tx3"]
- Hashes each transaction

```
hashes = ["h1", "h2", "h3"]
```

If the number of hashes is odd, duplicates the last one:

```
hashes = ["h1", "h2", "h3", "h3"]
```

Concatenates hashes in pairs and rehashes:

```
["hash(h1 + h2)", "hash(h3 + h3)"]
```

Generating RSA Keys

```

def generate_key_pair():

    private_key = rsa.generate_private_key(

        public_exponent=65537,

        key_size=2048

    )

    public_key = private_key.public_key()

```

```
return private_key, public_key
```

```
# Generate RSA key pair
def generate_key_pair(): 1 usage
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048
    )
    public_key = private_key.public_key()
    return private_key, public_key

# Digital signature function
```

How it works:

- Creates a private key `private_key` (2048 bits).
- Derives the public key `public_key`.
- Example:

```
private_key, public_key = generate_key_pair()
```

Signing Data

```
def sign_data(private_key, data):

    data_bytes = json.dumps(data, sort_keys=True).encode()

    signature = private_key.sign(
        data_bytes,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

    return signature.hex()
```

```

# Digital signature function
def sign_data(private_key, data): 1 usage
    data_bytes = json.dumps(data, sort_keys=True).encode()
    signature = private_key.sign(
        data_bytes,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return signature.hex()

```

How it works:

- Converts data to JSON and encodes it into bytes.
- Signs the data using `private_key.sign()`.
- Converts the signature to a hexadecimal string.

```
sign = sign_data(private_key, {"msg": "Hello"})
```

Verifying Signatures

```

def verify_signature(public_key, data, signature):
    data_bytes = json.dumps(data, sort_keys=True).encode()
    try:
        public_key.verify(
            bytes.fromhex(signature),
            data_bytes,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            )
        )
    except:
        return False
    return True

```

```
    ),  
    hashes.SHA256()  
)  
return True  
  
except:  
    return False
```

```
# Verify signature  
def verify_signature(public_key, data, signature):  
    data_bytes = json.dumps(data, sort_keys=True).encode()  
    try:  
        public_key.verify(  
            bytes.fromhex(signature),  
            data_bytes,  
            padding.PSS(  
                mgf=padding.MGF1(hashes.SHA256()),  
                salt_length=padding.PSS.MAX_LENGTH  
            ),  
            hashes.SHA256()  
        )  
        return True  
    except:  
        return False
```

How it works:

- Converts signature back from hex.
- Verifies the signature using `public_key.verify()`.
- Returns True if valid, False otherwise.

Transaction Class

class Transaction:

def __init__(self, sender, receiver, amount, private_key):

self.sender = sender

self.receiver = receiver

self.amount = amount

self.timestamp = time.time()

self.tx_hash = self.calculate_hash()

self.signature = sign_data(private_key, self.to_dict())

```
# Transaction structure
class Transaction:
    def __init__(self, sender, receiver, amount, private_key):
        self.sender = sender
        self.receiver = receiver
        self.amount = amount
        self.timestamp = time.time()
        self.tx_hash = self.calculate_hash()
        self.signature = sign_data(private_key, self.to_dict())
```

How it works:

- Stores sender, receiver, and amount.
- Records the current timestamp self.timestamp.
- Computes the transaction hash self.tx_hash.
- Signs the transaction.

Block Class

class Block:

def __init__(self, previous_hash, transactions):

self.previous_hash = previous_hash

self.transactions = transactions

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

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

self.hash = self.calculate_hash()
```

```
# Block structure
class Block:
    def __init__(self, previous_hash, transactions):
        self.previous_hash = previous_hash
        self.transactions = transactions
        self.merkle_root = merkle_root([tx.tx_hash for tx in transactions])
        self.timestamp = time.time()
        self.hash = self.calculate_hash()
```

How it works:

- Stores the previous block's hash previous_hash.
- Computes the Merkle root merkle_root().
- Calculates the hash for the current block.

UTXO Class (Unspent Transaction Outputs)

```
class UTXO:
```

```
    def __init__(self):
        self.balances = {}
```

```
    def update_balances(self, transactions):
        for tx in transactions:
            self.balances.setdefault(tx.sender, 100)
            self.balances.setdefault(tx.receiver, 100)

            if self.balances[tx.sender] < tx.amount:
                return False # Insufficient funds

            self.balances[tx.sender] -= tx.amount
            self.balances[tx.receiver] += tx.amount
```


return True

```
5 # UTXO model to track balances
6
7 class UTXO:
8     def __init__(self):
9         self.balances = {}
10
11     def update_balances(self, transactions): 1 usage (1 dynamic)
12         for tx in transactions:
13             self.balances.setdefault(tx.sender, 100) # Default initial balance
14             self.balances.setdefault(tx.receiver, 100)
15
16             if self.balances[tx.sender] < tx.amount:
17                 return False # Invalid transaction
18
19             self.balances[tx.sender] -= tx.amount
20             self.balances[tx.receiver] += tx.amount
21
22         return True
```

How it works:

- Initializes balances (user balances).
- Checks if the sender has enough funds.
- Updates balances after a transfer.

BlockchainExplorer Class (GUI for Blockchain)

```
class BlockchainExplorer:
```

```
    def __init__(self, blocks):
```

```
        self.blocks = blocks
```

```
        self.root = tk.Tk()
```

```
        self.root.title("Blockchain Explorer")
```

```
        self.tree = ttk.Treeview(self.root, columns=("Hash", "Merkle Root", "Transactions"),
show="headings")
```

```
        self.tree.heading("Hash", text="Block Hash")
```

```
        self.tree.heading("Merkle Root", text="Merkle Root")
```

```
        self.tree.heading("Transactions", text="Transactions Count")
```

```
        self.tree.pack(expand=True, fill="both")
```

```
self.populate_tree()
```

```
# Blockchain Explorer GUI
class BlockchainExplorer:
    def __init__(self, blocks):
        self.blocks = blocks
        self.root = tk.Tk()
        self.root.title("Blockchain Explorer")

        self.tree = ttk.Treeview(self.root, columns=("Hash", "Merkle Root", "Transactions"), show="headings")
        self.tree.heading("Hash", text="Block Hash")
        self.tree.heading("Merkle Root", text="Merkle Root")
        self.tree.heading("Transactions", text="Transactions Count")
        self.tree.pack(expand=True, fill="both")

        self.populate_tree()
```

How it works:

- Displays a list of blocks using tkinter.
- Populates a table with block hashes.

Wallet Initialization (__init__)

```
self.private_key, self.public_key = generate_key_pair()
```

```
self.address = self.public_key.public_bytes(
```

```
    encoding=serialization.Encoding.PEM,
```

```
    format=serialization.PublicFormat.SubjectPublicKeyInfo
```

```
).decode()
```

```
class WalletGUI:
    def __init__(self):
        self.root = tk.Tk()
        self.root.title("Blockchain Wallet")

        self.private_key, self.public_key = generate_key_pair()
        self.address = self.public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        ).decode()

        tk.Label(self.root, text="Your Wallet Address:").pack()
```

Launches WalletGUI(), which creates a wallet with RSA keys.

Allows sending transactions.

Sending a Transaction (send_transaction)

```
def send_transaction(self):  
  
    transaction = Transaction(self.address, "Bob", 30, self.private_key)  
  
    messagebox.showinfo("Transaction Sent", f"Sent 30 to Bob\nTx Hash:  
{transaction.tx_hash}")
```

```
def send_transaction(self): 1 usage  
    receiver = "Bob"  
    amount = 30  
    transaction = Transaction(self.address, receiver, amount, self.private_key)  
    messagebox.showinfo(title: "Transaction Sent", message: f"Sent {amount} to {receiver}\nTx Hash: {transaction.tx_hash}")
```

This method creates and signs a transaction:

1. **Recipient:** "Bob", amount — 30 coins.
2. **A Transaction object is created**, signed with the private key.
3. A confirmation message is displayed.

Running the Interface

```
def run(self):  
  
    self.root.mainloop()
```

```
def run(self): 1 usage  
    self.root.mainloop()
```

The application launches a graphical window and keeps running until closed.

Running the Wallet GUI

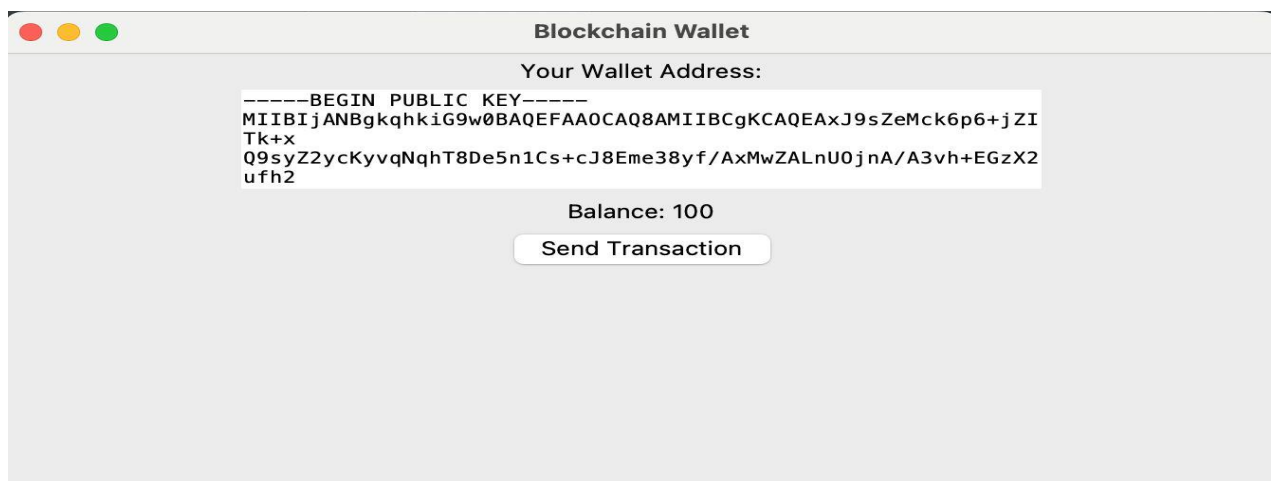
```
if __name__ == "__main__":  
  
    wallet = WalletGUI()  
  
    wallet.run()
```

```

# Main execution
if __name__ == "__main__":
    wallet = WalletGUI()
    wallet.run()

```

If this file is run directly (__main__), a wallet object is created and the GUI is started (wallet.run()).



Code Summary

This code implements a **basic blockchain system** with cryptographic security, digital signatures, and a wallet GUI.

Key Features:

Asymmetric key generation (public and private keys) for signing transactions.

Digital signatures to protect transaction data.

Hashing and Merkle Tree construction to ensure transaction integrity.

Block creation and linking into a blockchain.

User balance tracking and transaction validation.

Graphical interfaces for the wallet and blockchain explorer using tkinter.

Conclusion: This code demonstrates the fundamental principles of blockchain, providing data security through cryptography and a user-friendly interface.

