The code is basically a small blockchain system where we handle transactions, validate them, and display everything in a GUI.

First, we use a Merkle Tree to organize transactions inside each block. A Merkle Tree works by hashing transactions and combining those hashes step by step until we get a single final hash, called the Merkle Root. This root acts like a fingerprint for all the transactions in that block. If even a tiny part of any transaction changes, the Merkle Root will be completely different, helping us detect tampering.

Each transaction has some important details: who sent the money, who received it, how much was sent, and a unique hash that identifies the transaction. Instead of just trusting that users have enough money to spend, we use something called the UTXO model (Unspent Transaction Outputs). This makes sure that users can only spend coins they actually own. Everyone starts with 100 coins, and if a transaction would put someone's balance below zero, it gets rejected.

When transactions are confirmed, they are grouped into a block, and this block stores the Merkle Root for verification. Then, we show everything in a simple GUI using Tkinter. The GUI works as a Blockchain Explorer, where you can see the blocks, transactions, sender and receiver addresses, amounts, and whether each block is valid.

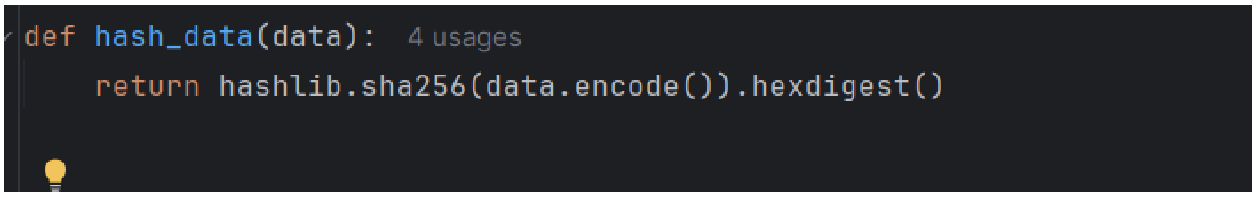
For validation, the system ensures that only valid transactions make it into a block. It checks that the Merkle Root matches and that no one's balance goes negative. If something is wrong, the block won't be added to the blockchain.

So, the whole system is like a mini blockchain that tracks transactions, verifies them properly, and lets you explore the data visually. It's a simple but effective way to simulate how real blockchains handle transactions securely. 🚀

Hashing Function

1)

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

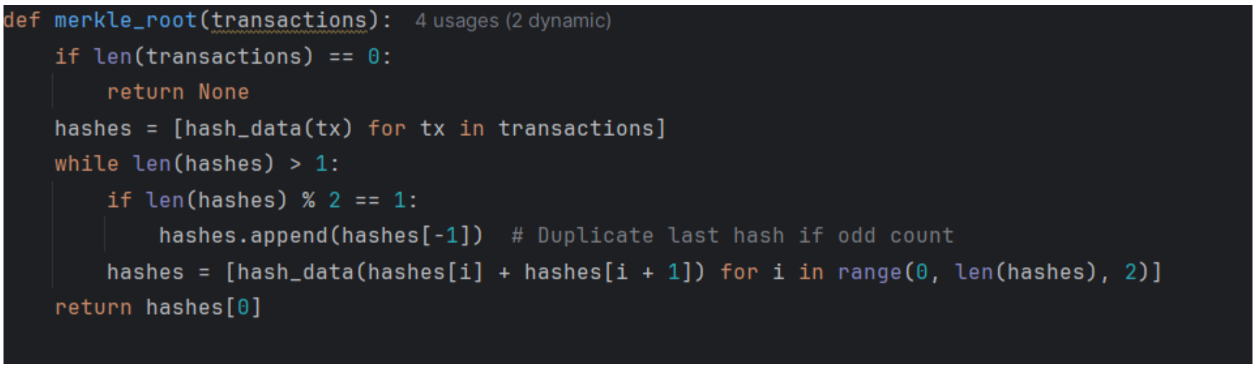


```
def hash_data(data):  4 usages
    return hashlib.sha256(data.encode()).hexdigest()
```

This function takes a string data, encodes it, and returns its SHA-256 hash

Merkle Tree Root Calculation

```
def merkle_root(transactions):

  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]
```



```
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]
```

This function calculates the Merkle root:

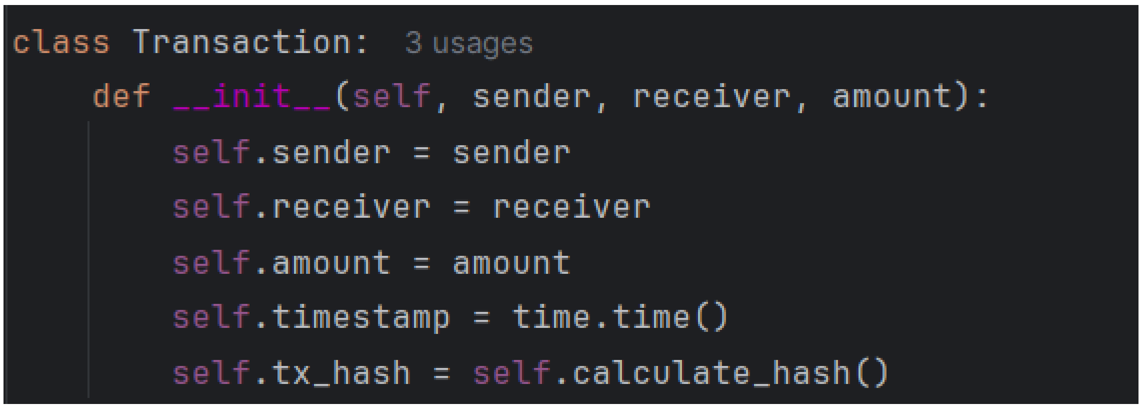
Converts each transaction into a hash.

If the number of hashes is odd, the last hash is duplicated.

Iteratively combines and hashes pairs until only one root hash remains.

Transaction Class

```
class Transaction:

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

    self.sender = sender

    self.receiver = receiver

    self.amount = amount

    self.timestamp = time.time()

    self.tx_hash = self.calculate_hash()
```



```python
class Transaction:  3 usages
    def __init__(self, sender, receiver, amount):
        self.sender = sender
        self.receiver = receiver
        self.amount = amount
        self.timestamp = time.time()
        self.tx_hash = self.calculate_hash()
```

This class creates a transaction object with:

sender – the sender's name.

receiver – the receiver's name.

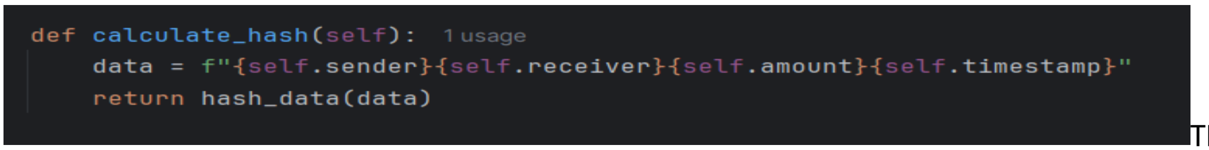
amount – the transaction amount.

timestamp – the transaction creation time.

tx_hash – the unique transaction hash.

Transaction Hash Calculation

```
def calculate_hash(self):

  data = f"{self.sender}{self.receiver}{self.amount}{self.timestamp}"

  return hash_data(data)
```

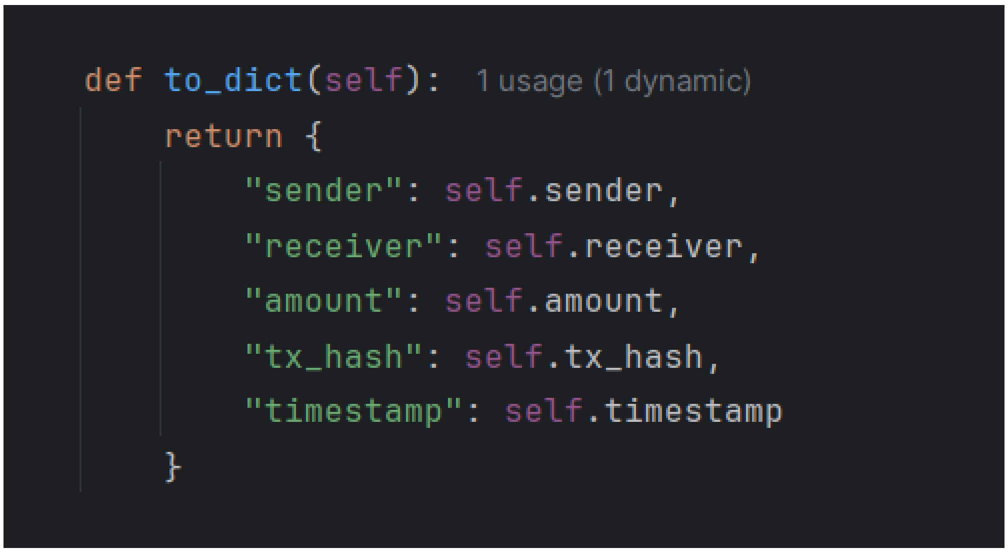


```python
def calculate_hash(self):  1 usage
    data = f"{self.sender}{self.receiver}{self.amount}{self.timestamp}"
    return hash_data(data)
```

This method forms a string with transaction details and computes its hash.

Convert Transaction to Dictionary

```
def to_dict(self):

  return {

    "sender": self.sender,

    "receiver": self.receiver,

    "amount": self.amount,

    "tx_hash": self.tx_hash,
        "timestamp": self.timestamp
    }
```

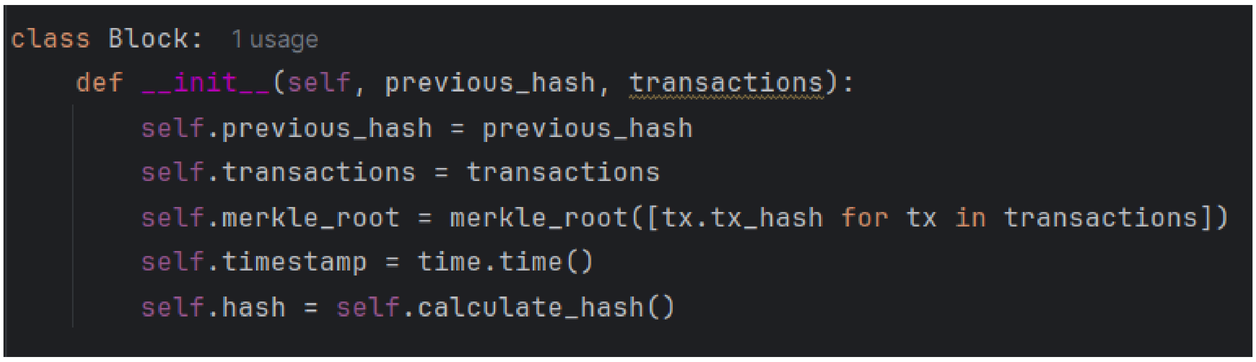


```
    def to_dict(self):  1 usage (1 dynamic)
        return {
            "sender": self.sender,
            "receiver": self.receiver,
            "amount": self.amount,
            "tx_hash": self.tx_hash,
            "timestamp": self.timestamp
        }
```

This method returns the transaction as a dictionary.

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()
```



```
class Block:  1 usage
    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()
```

This class creates a block containing:

previous_hash – hash of the previous block.

transactions – a list of transactions.

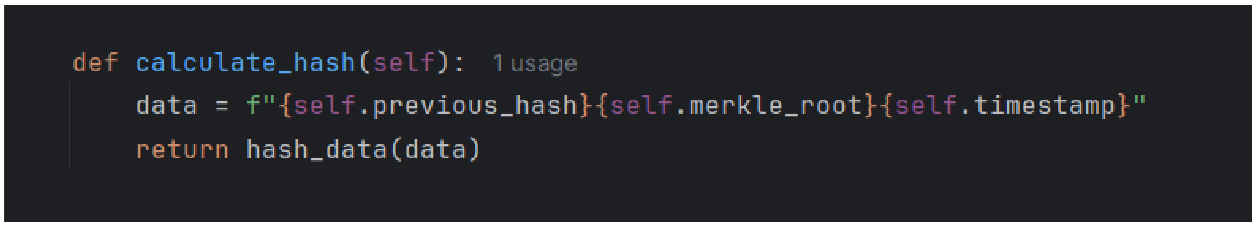
merkle_root – the Merkle root of all transactions.

timestamp – block creation time.

hash – the hash of the current block.

Block Hash Calculation

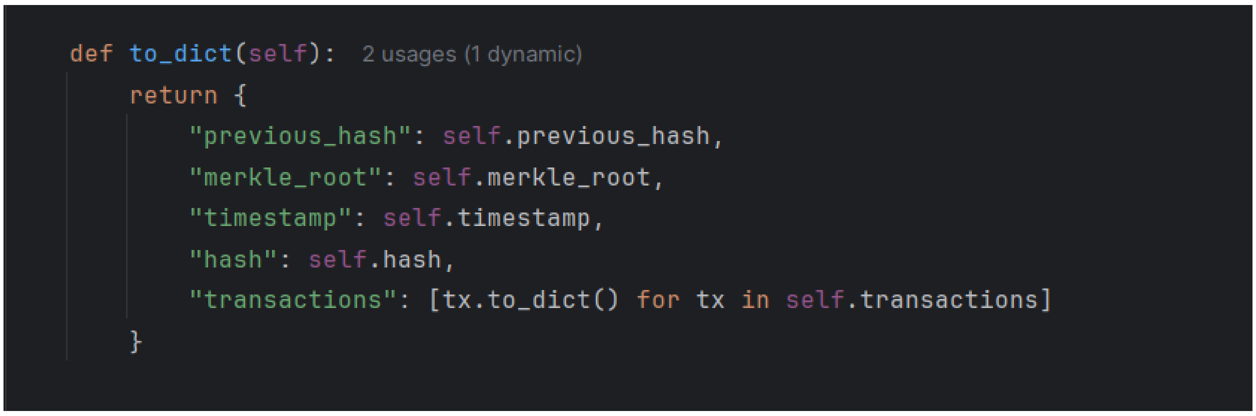
```
def calculate_hash(self):

  data = f"{self.previous_hash}{self.merkle_root}{self.timestamp}"

  return hash_data(data)
```



```
def calculate_hash(self):  1 usage
    data = f"{self.previous_hash}{self.merkle_root}{self.timestamp}"
    return hash_data(data)
```

Convert Block to Dictionary

```
def to_dict(self):

  return {

    "previous_hash": self.previous_hash,

    "merkle_root": self.merkle_root,

    "timestamp": self.timestamp,

    "hash": self.hash,

    "transactions": [tx.to_dict() for tx in self.transactions]

  }
```



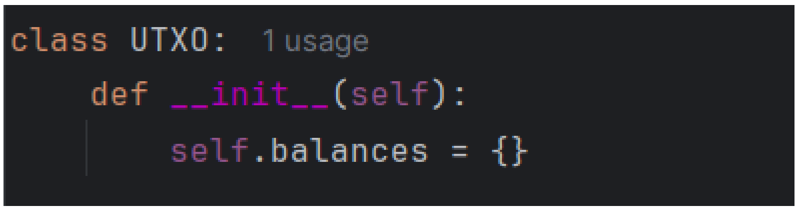
```
def to_dict(self):  2 usages (1 dynamic)
    return {
        "previous_hash": self.previous_hash,
        "merkle_root": self.merkle_root,
        "timestamp": self.timestamp,
        "hash": self.hash,
        "transactions": [tx.to_dict() for tx in self.transactions]
    }
```

This method returns the block as a dictionary.

UTXO (Unspent Transaction Output) Class

class UTXO:

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

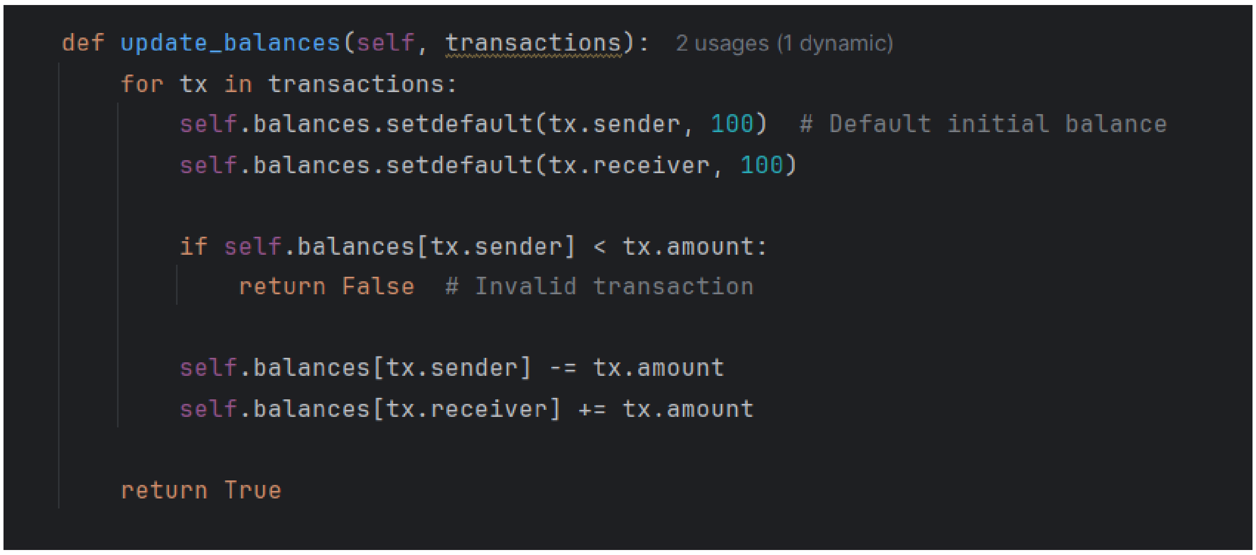


```
class UTXO:  1 usage
    def __init__(self):
        self.balances = {}
```

This class keeps track of user balances.

10) Update Balances

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

    if self.balances[tx.sender] < tx.amount:
      return False  # Invalid transaction

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

  return True
```


```
def update_balances(self, transactions):  2 usages (1 dynamic)
    for tx in transactions:
        self.balances.setdefault(tx.sender, 100)  # Default initial balance
        self.balances.setdefault(tx.receiver, 100)

        if self.balances[tx.sender] < tx.amount:
            return False  # Invalid transaction

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

    return True
```

Ensures the sender has enough balance.

Deducts the amount from the sender and adds it to the receiver.

Blockchain Explorer (GUI)

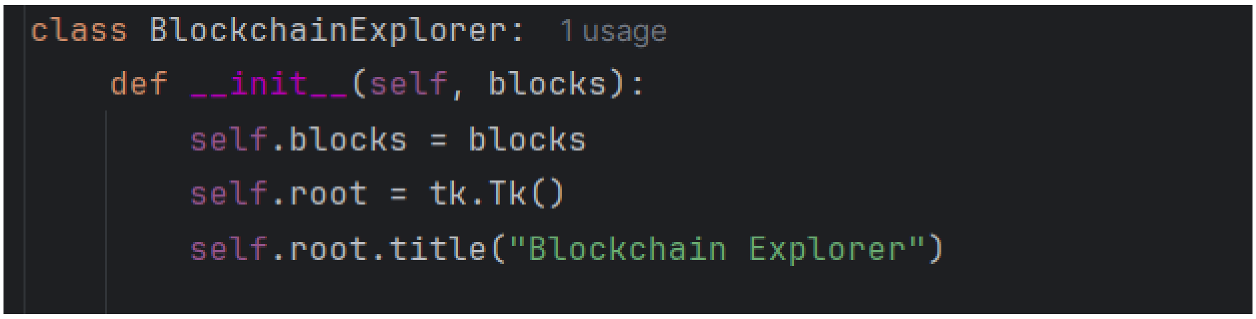
class BlockchainExplorer:

def __init__(self, blocks):

self.blocks = blocks

self.root = tk.Tk()

self.root.title("Blockchain Explorer")


```
class BlockchainExplorer:  1 usage
    def __init__(self, blocks):
        self.blocks = blocks
        self.root = tk.Tk()
        self.root.title("Blockchain Explorer")
```
This class creates a simple GUI to explore blockchain data.

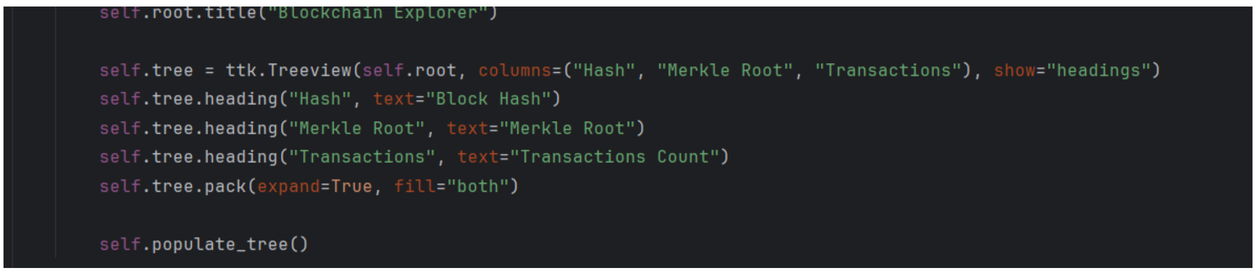
Creating Table View

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.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()
```

A table is created with the following columns:

Block Hash – the block hash.

Merkle Root – the Merkle root.

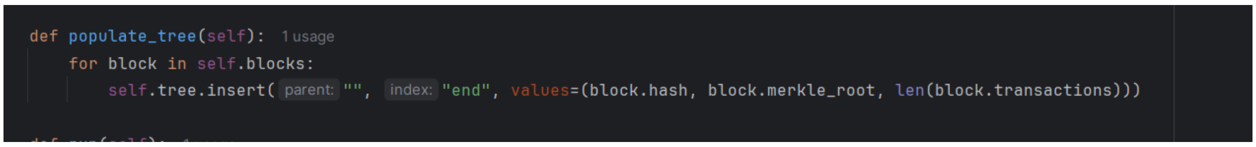
Transactions Count – the number of transactions.

Populate Table with Block Data

def populate_tree(self):

  for block in self.blocks:

    self.tree.insert("", "end", values=(block.hash, block.merkle_root, len(block.transactions)))



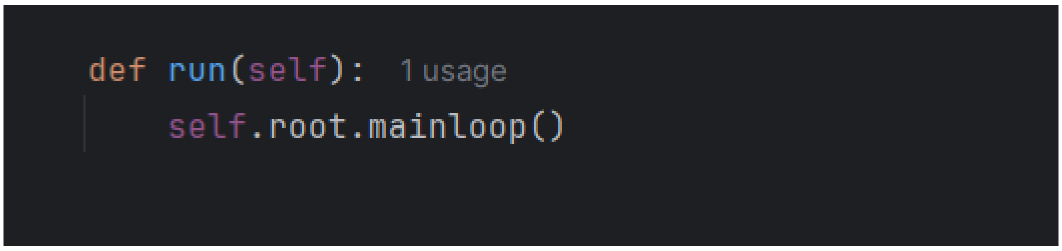
```
def populate_tree(self):  1 usage
    for block in self.blocks:
        self.tree.insert( parent: "",  index: "end", values=(block.hash, block.merkle_root, len(block.transactions)))
```

This method adds blocks to the table.

Run the GUI

def run(self):

  self.root.mainloop()



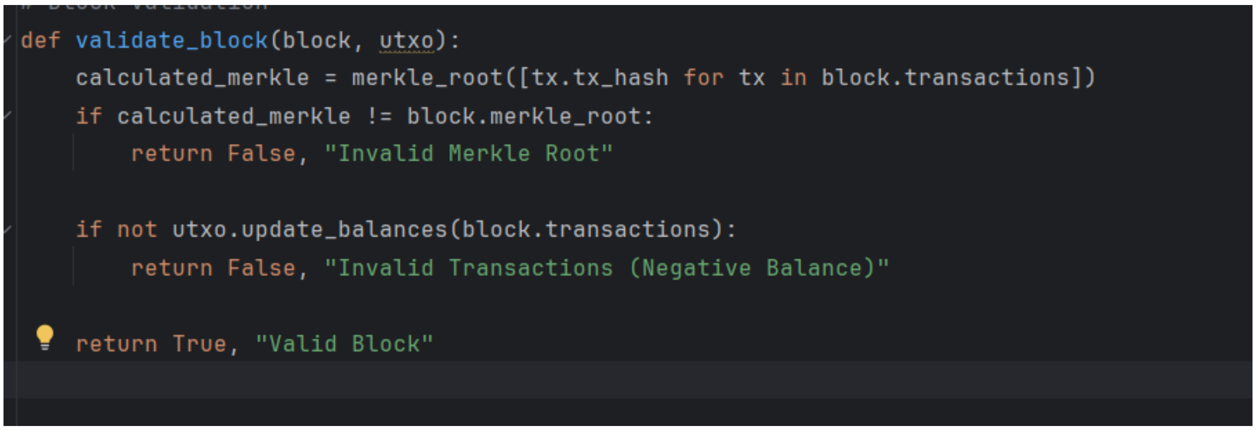
```
def run(self):  1 usage
    self.root.mainloop()
```

Launches the graphical interface.

Block Validation

def validate_block(block, utxo):

```
calculated_merkle = merkle_root([tx.tx_hash for tx in block.transactions])

if calculated_merkle != block.merkle_root:

    return False, "Invalid Merkle Root"


if not utxo.update_balances(block.transactions):

    return False, "Invalid Transactions (Negative Balance)"


return True, "Valid Block"
```



```
def validate_block(block, utxo):
    calculated_merkle = merkle_root([tx.tx_hash for tx in block.transactions])
    if calculated_merkle != block.merkle_root:
        return False, "Invalid Merkle Root"

    if not utxo.update_balances(block.transactions):
        return False, "Invalid Transactions (Negative Balance)"

    return True, "Valid Block"
```

This function verifies:

The Merkle root is correctly calculated.

Transactions do not result in negative balances.

Main Execution

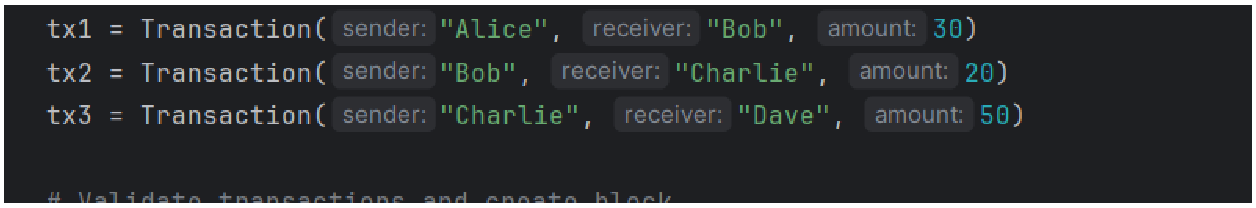
```
if __name__ == "__main__":
```

```
if __name__ == "__main__":
```

Defines the entry point of the script.

17) Create Transactions

```
tx1 = Transaction("Alice", "Bob", 30)

tx2 = Transaction("Bob", "Charlie", 20)

tx3 = Transaction("Charlie", "Dave", 50)
```



```
    tx1 = Transaction(sender: "Alice", receiver: "Bob", amount: 30)
    tx2 = Transaction(sender: "Bob", receiver: "Charlie", amount: 20)
    tx3 = Transaction(sender: "Charlie", receiver: "Dave", amount: 50)
```

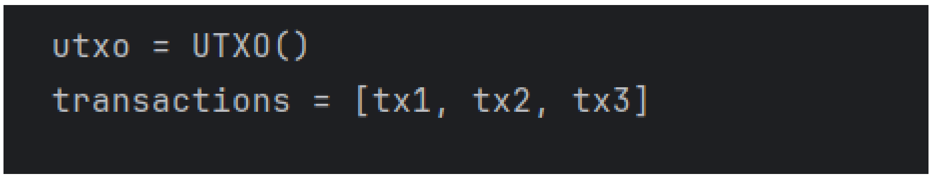
Three transactions are created.

Validate Transactions and Update Balances

utxo = UTXO()

transactions = [tx1, tx2, tx3]

if utxo.update_balances(transactions):



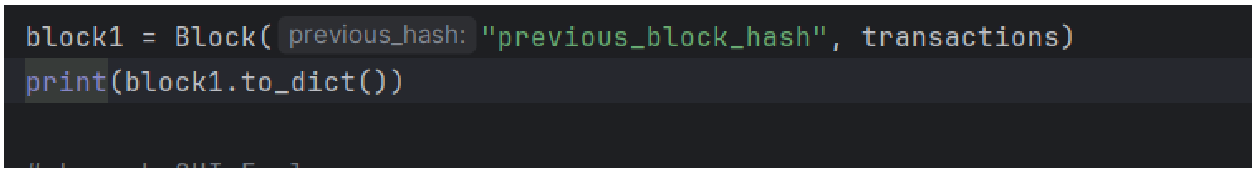
```
utxo = UTXO()
transactions = [tx1, tx2, tx3]
```

Checks if balances are sufficient.

Create Block

block1 = Block("previous_block_hash", transactions)

print(block1.to_dict())



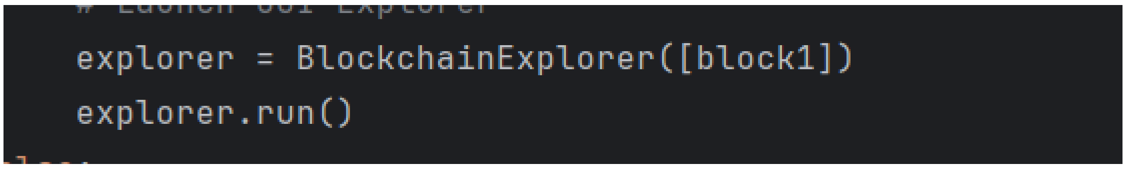
```
block1 = Block( previous_hash: "previous_block_hash", transactions)
print(block1.to_dict())
```

If transactions are valid, a block is created.

Launch Blockchain Explorer

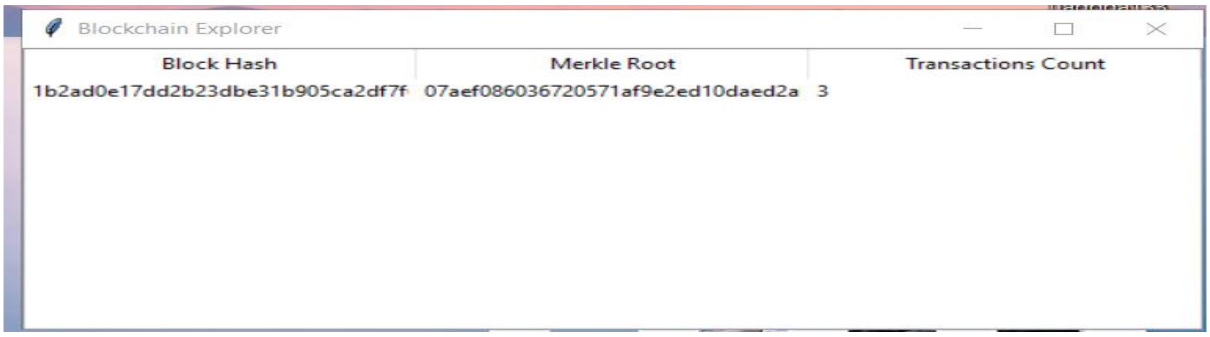
explorer = BlockchainExplorer([block1])

explorer.run()



```
explorer = BlockchainExplorer([block1])
explorer.run()
```

Displays the block data in a GUI."



```
Block Hash                         Merkle Root                        Transactions Count
1b2ad0e17dd2b23dbe31b905ca2df7f   07aef086036720571af9e2ed10daed2a   3
```

Final Summary

This code implements a basic blockchain model:

1. **Transactions are created** with sender, receiver, and amount.
2. **Merkle Tree Root is computed** for data integrity.
3. **Blocks are generated** linking to the previous block.
4. **UTXO model ensures valid balances** before processing transactions.
5. **Validation checks** Merkle root and balance integrity.
6. **A GUI displays blockchain data** using a simple table.

This is a minimal blockchain simulation that validates transactions but lacks consensus mechanisms and mining.