To implement Proof of Work (PoW) manually (without using the hashlib library), I modified the following:

Mining Logic:A mining system was added where each block must find a valid nonce that meets a specific difficulty level. Instead of hashlib, a simple sha256-like function was manually implemented by summing ASCII values of characters in a string and converting them to hex.

Nonce Calculation:The block's hash is computed in a loop, incrementing the nonce until it produces a hash with the required number of leading zeros (difficulty).

Rewards and Fees:The mining reward system was added by crediting the miner with a reward when they successfully mine a block. Transaction fees are also included in the mining process.

Conflict Handling in Mining:A scenario where two miners solve a block simultaneously was handled. The blockchain selects the longest valid chain (longest chain rule), ensuring that forks are resolved properly.

GUI IntegrationThe Tkinter-based Blockchain Explorer was modified to properly reflect mined blocks. Since PoW is a time-consuming process, mining is executed in a separate thread to prevent the GUI from freezing.

The final result is a functional Proof of Work blockchain with rewards, transaction fees, and GUI integration.

Logic mining

The code implements mining, where the block searches for the correct nonce to match the complexity.:

```
class ProofOfWork:
    def __init__(self, difficulty):
        self.difficulty = difficulty

    def mine_block(self, block):
        block.nonce = 0
        while not self.is_valid_proof(block):
            block.nonce += 1
```

```
return block
```

```
class ProofOfWork:  1 usage
    def __init__(self, difficulty):
        self.difficulty = difficulty

    def mine_block(self, block):  1 usage (1 dynamic)
        block.nonce = 0
        while not self.is_valid_proof(block):
            block.nonce += 1
        return block
```

Here, the nonce is incremented until a suitable value is found.

Calculating nonce

To calculate the nonce, the sha256 function is used, which summarizes ASCII character codes and converts them to hex.:

```
def sha256(data):
  return hex(sum(ord(c) for c in data) % (2 ** 256))[2:]
```

```
def sha256(data):  5 usages
    constants = [
        0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
        0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19
    ]
    return hex(sum(ord(c) for c in data) % (2 ** 256))[2:]
```

The block is hashed repeatedly with increasing nonce until the desired difficulty level is reached.:

```
def is_valid_proof(self, block):
  hash_value = sha256(f"{block.previous_hash}{block.merkle_root}{block.timestamp}{block.nonce}")
  return hash_value[:self.difficulty] == "0" * self.difficulty
```

```python
        return block

    def is_valid_proof(self, block):
        hash_value = sha256(f"{block.previous_hash}{block.merkle_root}{block.timestamp}{block.nonce}")
        return hash_value[:self.difficulty] == "0" * self.difficulty
```

3. **Awards and Commissions**

The code implements a reward system for miners and accounting for transaction fees.:

```python
class UTXO:
    def update_balances(self, transactions, miner):
        for tx in transactions:
            self.balances.setdefault(tx.sender, 100)
            self.balances.setdefault(tx.receiver, 100)
            self.balances.setdefault(miner, 0)

            if self.balances[tx.sender] < (tx.amount + tx.fee):
                return False

            self.balances[tx.sender] -= (tx.amount + tx.fee)
            self.balances[tx.receiver] += tx.amount
            self.balances[miner] += tx.fee
        return True
```

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

    def update_balances(self, transactions, miner):
        for tx in transactions:
            self.balances.setdefault(tx.sender, 100)
            self.balances.setdefault(tx.receiver, 100)
            self.balances.setdefault(miner, 0)

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

            self.balances[tx.sender] -= (tx.amount + tx.fee)
            self.balances[tx.receiver] += tx.amount
            self.balances[miner] += tx.fee
        return True
```

The miner receives a commission (tx.fee) for processing the transaction.

Conflict resolution

If two miners create blocks at the same time, a longer chain is selected.:

```
def resolve_conflicts(blockchain1, blockchain2):
   return blockchain1 if len(blockchain1) >= len(blockchain2) else blockchain2
```

```python
def resolve_conflicts(blockchain1, blockchain2):
    return blockchain1 if len(blockchain1) >= len(blockchain2) else blockchain2
```

This implements the longest chain rule.

GUI Integration

The code uses Tkinter to display the mining process and block structure.:

```
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()
```

```
class BlockchainExplorer:  1 usage
    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()
```

An interface for viewing ad blocks is created here.