# Import required libraries

import hashlib # For SHA-256 hashing

import time # To record timestamps and measure mining time

from dataclasses import dataclass # For cleaner block structure

import json # To serialize block data consistently

# Define a Block class using dataclass

@dataclass

class Block:

index: int # Block number in the chain

timestamp: float # Time when the block was created

data: str # Payload (can be transactions or any info)

previous\_hash: str # Hash of the previous block in the chain

nonce: int = 0 # Value adjusted by miner to solve PoW

hash: str = "" # Final computed hash of the block

def header(self) -> bytes:

"""

Returns a serialized (JSON) version of the block’s header

that will be hashed to compute PoW.

"""

header\_dict = {

"index": self.index,

"timestamp": self.timestamp,

"data": self.data,

"previous\_hash": self.previous\_hash,

"nonce": self.nonce

}

# Convert to JSON string with sorted keys for deterministic output

return json.dumps(header\_dict, sort\_keys=True).encode()

# Function to compute SHA-256 hash of given data

def sha256\_hex(data: bytes) -> str:

return hashlib.sha256(data).hexdigest()

# Define the Blockchain class

class SimpleBlockchain:

def \_\_init\_\_(self, difficulty: int = 4):

# Initialize an empty blockchain

self.chain = []

# Difficulty = required number of leading zeros in hash

self.difficulty = difficulty

# Create and add the Genesis block (first block)

genesis = self.create\_genesis\_block()

self.chain.append(genesis)

def create\_genesis\_block(self) -> Block:

# Genesis block has index 0, no previous hash

genesis = Block(index=0, timestamp=time.time(), data="Genesis Block", previous\_hash="0")

# Compute its hash (without PoW enforcement)

genesis.hash = sha256\_hex(genesis.header())

return genesis

def last\_block(self) -> Block:

# Returns the most recent block in the chain

return self.chain[-1]

def mine\_block(self, data: str, max\_nonce: int = 2\*\*32-1) -> Block:

# Prepare a new block with incremented index and given data

index = self.last\_block().index + 1

previous\_hash = self.last\_block().hash

block = Block(index=index, timestamp=time.time(), data=data, previous\_hash=previous\_hash)

# Mining target: hash must start with this prefix of zeros

target\_prefix = "0" \* self.difficulty

# Start mining timer

start = time.time()

nonce = 0

# Try all nonce values until max\_nonce

while nonce <= max\_nonce:

block.nonce = nonce # Set current nonce

h = sha256\_hex(block.header()) # Compute hash of block header

# Check if hash meets difficulty target

if h.startswith(target\_prefix):

block.hash = h # Assign valid hash

elapsed = time.time() - start

# Print mining success info

print(f"[mined] index={block.index} nonce={block.nonce} hash={block.hash} time={elapsed:.4f}s")

return block

nonce += 1 # Increment nonce and try again

# If no valid nonce found

raise RuntimeError("Failed to find valid nonce")

def add\_block(self, block: Block):

# Add block only if it is valid compared to the previous block

if self.is\_valid\_new\_block(block, self.last\_block()):

self.chain.append(block)

return True

return False

def is\_valid\_new\_block(self, block: Block, previous\_block: Block) -> bool:

# Check index continuity

if block.index != previous\_block.index + 1:

return False

# Check previous hash matches

if block.previous\_hash != previous\_block.hash:

return False

# Check recomputed hash matches stored hash

if sha256\_hex(block.header()) != block.hash:

return False

# Check PoW requirement (leading zeros)

if not block.hash.startswith("0" \* self.difficulty):

return False

return True

def is\_chain\_valid(self) -> bool:

# Verify all blocks in the chain except genesis

for i in range(1, len(self.chain)):

if not self.is\_valid\_new\_block(self.chain[i], self.chain[i - 1]):

return False

return True

# -------- Run Demonstration --------

difficulty = 3 # Required number of leading zeros in hash

num\_blocks = 3 # Number of blocks to mine

# Initialize blockchain

bc = SimpleBlockchain(difficulty=difficulty)

print(f"Genesis hash: {bc.chain[0].hash}")

# Mine blocks one by one

for i in range(num\_blocks):

payload = f"Block payload #{i+1}" # Sample data inside block

print(f"\nMining block {i+1} with difficulty={difficulty} ...")

block = bc.mine\_block(data=payload) # Mine the block

added = bc.add\_block(block) # Add it to blockchain if valid

print("Block added:", added)

# Display final blockchain

print("\n=== Final Blockchain ===")

print("Valid chain?", bc.is\_chain\_valid())

for blk in bc.chain:

print(f"Index={blk.index} | Nonce={blk.nonce} | Hash={blk.hash[:25]}... | Data={blk.data}")