// Calculator.sol

// Simple calculator contract

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Calculator {

function add(uint256 a, uint256 b) public pure returns (uint256) {

return a + b;

}

function subtract(uint256 a, uint256 b) public pure returns (uint256) {

return a - b;

}

function multiply(uint256 a, uint256 b) public pure returns (uint256) {

return a \* b;

}

function divide(uint256 a, uint256 b) public pure returns (uint256) {

require(b > 0, "Cannot divide by zero");

return a / b;

}

}

// Property.sol

// Simple property ownership contract

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Property {

address public owner;

constructor() {

owner = msg.sender;

}

modifier onlyOwner() {

require(msg.sender == owner, "Only owner can call this function");

\_;

}

function transferOwnership(address newOwner) public onlyOwner {

owner = newOwner;

}

}

// Lottery.sol

// Simple lottery contract

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Lottery {

address public manager;

address[] public players;

constructor() {

manager = msg.sender;

}

modifier onlyManager() {

require(msg.sender == manager, "Only manager can call this function");

\_;

}

function enter() public payable {

require(msg.value > 0.01 ether, "Minimum entry fee is 0.01 ether");

players.push(msg.sender);

}

function pickWinner() public onlyManager {

require(players.length > 0, "No players to pick a winner");

address winner = players[random() % players.length];

payable(winner).transfer(address(this).balance);

players = new address[](0); // Reset players array for the next round

}

function getPlayers() public view returns (address[] memory) {

return players;

}

function random() private view returns (uint256) {

return uint256(keccak256(abi.encodePacked(block.difficulty, block.timestamp, players)));

}

}

#Create the Blockchain using Python, Flask micro web framework and Postman

import hashlib

import json

from time import time

from flask import Flask, jsonify, request

class Blockchain:

def \_\_init\_\_(self):

self.chain = []

self.current\_transactions = []

# Create the genesis block

self.new\_block(previous\_hash='1', proof=100)

def new\_block(self, proof, previous\_hash=None):

block = {

'index': len(self.chain) + 1,

'timestamp': time(),

'transactions': self.current\_transactions,

'proof': proof,

'previous\_hash': previous\_hash or self.hash(self.chain[-1]) if self.chain else '1',

}

# Reset the current list of transactions

self.current\_transactions = []

self.chain.append(block)

return block

def new\_transaction(self, sender, recipient, amount):

transaction = {

'sender': sender,

'recipient': recipient,

'amount': amount,

}

self.current\_transactions.append(transaction)

return self.last\_block['index'] + 1

@property

def last\_block(self):

return self.chain[-1]

@staticmethod

def hash(block):

# Hash a block using SHA-256

block\_string = json.dumps(block, sort\_keys=True).encode()

return hashlib.sha256(block\_string).hexdigest()

# Create a Flask web server

app = Flask(\_\_name\_\_)

# Create a blockchain instance

blockchain = Blockchain()

# Define API endpoints

@app.route('/mine', methods=['GET'])

def mine():

# Mine a new block

last\_block = blockchain.last\_block

last\_proof = last\_block['proof']

proof = blockchain.proof\_of\_work(last\_proof)

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

# Add a reward transaction for mining

blockchain.new\_transaction(sender="0", recipient=node\_identifier, amount=1)

# Create the new block

block = blockchain.new\_block(proof, previous\_hash)

response = {

'message': 'New Block Forged',

'index': block['index'],

'transactions': block['transactions'],

'proof': block['proof'],

'previous\_hash': block['previous\_hash'],

}

return jsonify(response), 200

@app.route('/transactions/new', methods=['POST'])

def new\_transaction():

# Create a new transaction

values = request.get\_json()

# Check that required fields are in the POST data

required = ['sender', 'recipient', 'amount']

if not all(k in values for k in required):

return 'Missing values', 400

# Create a new transaction

index = blockchain.new\_transaction(values['sender'], values['recipient'], values['amount'])

response = {'message': f'Transaction will be added to Block {index}'}

return jsonify(response), 201

@app.route('/chain', methods=['GET'])

def full\_chain():

# Return the full blockchain

response = {

'chain': blockchain.chain,

'length': len(blockchain.chain),

}

return jsonify(response), 200

# Run the server

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=5000)

Sure! Here are five basic programs written in Python to read the blocks of Bitcoin using real-time blocks (.dat files):

1. Reading the Magic Number and Block Size:

import binascii dat\_file = open('blk00000.dat', 'rb') magic\_number = binascii.hexlify(dat\_file.read(4)).decode() block\_size = int.from\_bytes(dat\_file.read(4), byteorder='little', signed=False) print(f"Magic Number: {magic\_number}") print(f"Block Size: {block\_size}")

1. Reading the Block Version:

dat\_file = open('blk00000.dat', 'rb') dat\_file.seek(8) # Skip magic number **and** block size block\_version = int.from\_bytes(dat\_file.read(4), byteorder='little', signed=False) print(f"Block Version: {block\_version}")

1. Reading the Previous Block Hash:

dat\_file = open('blk00000.dat', 'rb') dat\_file.seek(80) # Skip magic number, block size, **and** block version previous\_block\_hash = binascii.hexlify(dat\_file.read(32)).decode() print(f"Previous Block Hash: {previous\_block\_hash}")

1. Reading the Merkle Root:

dat\_file = open('blk00000.dat', 'rb') dat\_file.seek(112) # Skip magic number, block size, block version, **and** previous block hash merkle\_root = binascii.hexlify(dat\_file.read(32)).decode() print(f"Merkle Root: {merkle\_root}")

1. Reading the Timestamp:

import datetime dat\_file = open('blk00000.dat', 'rb') dat\_file.seek(144) # Skip magic number, block size, block version, previous block hash, **and** merkle root timestamp = int.from\_bytes(dat\_file.read(4), byteorder='little', signed=False) formatted\_timestamp = datetime.datetime.fromtimestamp(timestamp).strftime('%Y-%m-%d %H:%M:%S') print(f"Timestamp: {formatted\_timestamp}") These programs demonstrate the basic reading of various block components in Bitcoin using Python. Note that you may need to adjust the seek values based on the structure of the .dat file you are working with.