**PRACTICAL :- 1**

**Write the following programs for Blockchain in Python**

**Practical 1A:-**

**Aim:-** A simple client class that generates the private and public keys by using the builtin Python RSA algorithm and test it.

**Code:-**

# Import necessary modules from the cryptography library

from cryptography.hazmat.primitives.asymmetric import rsa

from cryptography.hazmat.primitives import serialization

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.primitives.asymmetric import padding

# Define a SimpleRSA\_Client class

class SimpleRSA\_Client:

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

# Initialize private\_key and public\_key attributes

self.private\_key = None

self.public\_key = None

def generate\_keys(self):

# Generate private key

self.private\_key = rsa.generate\_private\_key(

public\_exponent=65537, # Public exponent value for the RSA key

key\_size=2048, # Key size (in bits) for the RSA key

backend=default\_backend() # Use the default backend

)

# Generate corresponding public key from the private key

self.public\_key = self.private\_key.public\_key()

def encrypt\_message(self, message):

if not self.public\_key:

raise ValueError("Public key is missing. Generate keys first.")

# Encrypt the provided message using the public key

ciphertext = self.public\_key.encrypt(

message.encode(), # Convert the message to bytes

padding.OAEP( # Use OAEP padding scheme

mgf=padding.MGF1(algorithm=hashes.SHA256()), # Mask Generation Function

algorithm=hashes.SHA256(), # Hashing algorithm

label=None # No additional label

)

)

return ciphertext

def decrypt\_message(self, ciphertext):

if not self.private\_key:

raise ValueError("Private key is missing. Generate keys first.")

# Decrypt the provided ciphertext using the private key

plaintext = self.private\_key.decrypt(

ciphertext, # The ciphertext to decrypt

padding.OAEP( # Use OAEP padding scheme (should match the one used for encryption)

mgf=padding.MGF1(algorithm=hashes.SHA256()), # Mask Generation Function

algorithm=hashes.SHA256(), # Hashing algorithm

label=None # No additional label

)

)

return plaintext.decode() # Decode the decrypted bytes to a string

# Testing the Simple RSA Client

if \_\_name\_\_ == "\_\_main\_\_":

# Create an instance of SimpleRSA\_Client

client = SimpleRSA\_Client()

# Generate RSA keys

client.generate\_keys()

# Message to be encrypted

message = "Hello, Simple RSA by Muskan!"

# Encrypt the message

ciphertext = client.encrypt\_message(message)

print("Encrypted Message:")

print(ciphertext)

print()

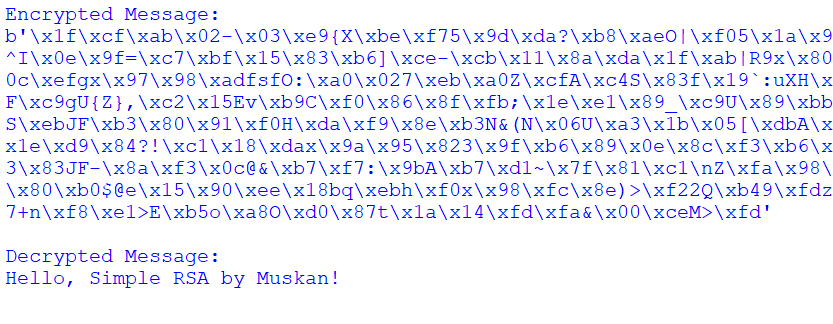
# Decrypt the ciphertext

decrypted\_message = client.decrypt\_message(ciphertext)

print("Decrypted Message:")

print(decrypted\_message).

**Output:-**

****

**Practical 1B:-**

**Aim:-** A transaction class to send and receive money and test it

**Code:-**

class Transaction:

def \_\_init\_\_(self, initial\_balance=0):

self.balance = initial\_balance

def send\_money(self, amount):

if amount <= 0:

raise ValueError("Amount must be greater than zero.")

if self.balance < amount:

raise ValueError("Insufficient funds.")

self.balance -= amount

return amount

def receive\_money(self, amount):

if amount <= 0:

raise ValueError("Amount must be greater than zero.")

self.balance += amount

return amount

# Testing the Transaction class

if \_\_name\_\_ == "\_\_main\_\_":

# Create a new transaction object with initial balance of 100

transaction = Transaction(initial\_balance=100)

print("Initial Balance:", transaction.balance)

# Send money

sent\_amount = transaction.send\_money(50)

print("Sent Amount:", sent\_amount)

print("Current Balance after sending:", transaction.balance)

# Receive money

received\_amount = transaction.receive\_money(30)

print("Received Amount:", received\_amount)

print("Current Balance after receiving:", transaction.balance)

# Trying to send more than balance

try:

transaction.send\_money(80)

except ValueError as e:

print("Error:", e)

# Trying to send negative amount

try:

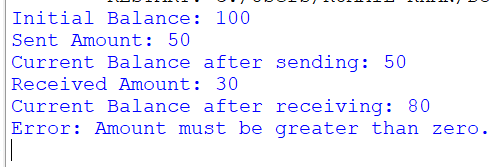
transaction.send\_money(-10)

except ValueError as e:

print("Error:", e)

# Print initial balances of both users

**Output:-**

****

**Practical 1C:-**

**Aim:-** Create multiple transactions and display them.

**Code:-**

class Transaction:

def \_\_init\_\_(self, initial\_balance=0):

self.balance = initial\_balance

def send\_money(self, amount):

if amount <= 0:

raise ValueError("Amount must be greater than zero.")

if self.balance < amount:

raise ValueError("Insufficient funds.")

self.balance -= amount

return amount

def receive\_money(self, amount):

if amount <= 0:

raise ValueError("Amount must be greater than zero.")

self.balance += amount

return amount

# Create multiple transactions

transactions = [

Transaction(100), # Transaction 1 with initial balance of 100

Transaction(50), # Transaction 2 with initial balance of 50

Transaction(200) # Transaction 3 with initial balance of 200

]

# Perform transactions

transactions[0].send\_money(30) # Transaction 1 sends 30

transactions[1].receive\_money(20) # Transaction 2 receives 20

transactions[2].send\_money(50) # Transaction 3 sends 50

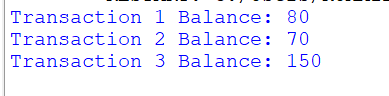
transactions[0].receive\_money(10) # Transaction 1 receives 10

# Display transactions and balances

for i, transaction in enumerate(transactions, 1):

print(f"Transaction {i} Balance:", transaction.balance)

**Output:-**

****

**Practical 1D:-**

**Aim:-** Create a blockchain, a genesis block and execute it.

**Code:-**

import hashlib

import time

class Block:

def \_\_init\_\_(self, index, timestamp, data, previous\_hash):

self.index = index

self.timestamp = timestamp

self.data = data

self.previous\_hash = previous\_hash

self.hash = self.calculate\_hash()

def calculate\_hash(self):

"""

Calculate the hash of the block using its properties.

"""

block\_info = str(self.index) + str(self.timestamp) + str(self.data) + str(self.previous\_hash)

return hashlib.sha256(block\_info.encode()).hexdigest()

class Blockchain:

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

self.chain = [self.create\_genesis\_block()]

def create\_genesis\_block(self):

"""

Create the first block (genesis block) of the blockchain.

"""

return Block(0, time.time(), "Genesis Block", "0")

def add\_block(self, data):

"""

Add a new block to the blockchain.

"""

previous\_block = self.chain[-1]

new\_block = Block(len(self.chain), time.time(), data, previous\_block.hash)

self.chain.append(new\_block)

def print\_chain(self):

"""

Print the contents of the blockchain.

"""

for block in self.chain:

print(f"Index: {block.index}")

print(f"Timestamp: {block.timestamp}")

print(f"Data: {block.data}")

print(f"Previous Hash: {block.previous\_hash}")

print(f"Hash: {block.hash}")

print()

# Create a blockchain

blockchain = Blockchain()

# Print the initial state (genesis block)

print("Initial Blockchain State (Genesis Block):")

blockchain.print\_chain()

# Add new blocks to the blockchain

blockchain.add\_block("Transaction 1")

blockchain.add\_block("Transaction 2")

# Print the updated blockchain

print("Updated Blockchain State:")

blockchain.print\_chain()

**Output:-**



**Practical 1E:-**

**Aim:-** Create a mining function and test it

**Code:-**

import hashlib # Library for hashing

import time # Library for timestamp

class Block:

def \_\_init\_\_(self, index, timestamp, data, previous\_hash):

"""

Constructor for Block class.

Args:

index: Index of the block in the blockchain.

timestamp: Timestamp of when the block was created.

data: Data or transaction details stored in the block.

previous\_hash: Hash of the previous block in the chain.

"""

self.index = index

self.timestamp = timestamp

self.data = data

self.previous\_hash = previous\_hash

self.hash = self.calculate\_hash() # Calculate hash of the block

def calculate\_hash(self):

"""

Calculate the hash of the block using its properties.

"""

# Concatenate block properties into a single string

block\_info = str(self.index) + str(self.timestamp) + str(self.data) + str(self.previous\_hash)

# Hash the concatenated string using SHA-256 and return the hexadecimal digest

return hashlib.sha256(block\_info.encode()).hexdigest()

class Blockchain:

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

"""

Constructor for Blockchain class.

"""

self.chain = [self.create\_genesis\_block()] # Initialize blockchain with genesis block

self.difficulty = 2 # Set difficulty level for mining

def create\_genesis\_block(self):

"""

Create the first block (genesis block) of the blockchain.

"""

# The genesis block has index 0, a timestamp, static data "Genesis Block",

# and a previous hash of "0" (since it's the first block)

return Block(0, time.time(), "Genesis Block", "0")

def get\_last\_block(self):

"""

Get the last block in the blockchain.

"""

# The last block in the chain is simply the last element of the chain list

return self.chain[-1]

def mine\_block(self, data):

"""

Mine a new block with the provided data.

Args:

data: Data or transaction details for the new block.

"""

# Get the last block in the chain

last\_block = self.get\_last\_block()

# Create a new block with incremented index, current timestamp, provided data,

# and the hash of the previous block as the previous hash

new\_block = Block(last\_block.index + 1, time.time(), data, last\_block.hash)

# Proof of Work: Find a hash that starts with the required number of zeros (difficulty)

while not new\_block.hash.startswith('0' \* self.difficulty):

new\_block.timestamp = time.time() # Update timestamp for each attempt

new\_block.hash = new\_block.calculate\_hash() # Recalculate the hash

# Once a valid hash is found (meets difficulty requirement), add the block to the chain

self.chain.append(new\_block)

return new\_block

def print\_chain(self):

"""

Print the contents of the blockchain.

"""

for block in self.chain:

# Print details of each block: index, timestamp, data, previous hash, and hash

print(f"Index: {block.index}")

print(f"Timestamp: {block.timestamp}")

print(f"Data: {block.data}")

print(f"Previous Hash: {block.previous\_hash}")

print(f"Hash: {block.hash}")

print()

# Create a blockchain instance

blockchain = Blockchain()

# Print the initial state of the blockchain (genesis block)

print("Initial Blockchain State (Genesis Block):")

blockchain.print\_chain()

# Mine new blocks with some example data

blockchain.mine\_block("Transaction 1")

blockchain.mine\_block("Transaction 2")

# Print the updated state of the blockchain after mining

print("Updated Blockchain State:")

blockchain.print\_chain()

**Output:-**

****

**Practical 1F:-**

**Aim:-** Add blocks to the miner and dump the blockchain.

**Code:-**

import hashlib

import time

class Block:

def \_\_init\_\_(self, index, timestamp, data, previous\_hash):

"""

Constructor for Block class.

Args:

index: Index of the block in the blockchain.

timestamp: Timestamp of when the block was created.

data: Data or transaction details stored in the block.

previous\_hash: Hash of the previous block in the chain.

"""

self.index = index

self.timestamp = timestamp

self.data = data

self.previous\_hash = previous\_hash

self.hash = self.calculate\_hash() # Calculate hash of the block

def calculate\_hash(self):

"""

Calculate the hash of the block using its properties.

"""

# Concatenate block properties into a single string

block\_info = str(self.index) + str(self.timestamp) + str(self.data) + str(self.previous\_hash)

# Hash the concatenated string using SHA-256 and return the hexadecimal digest

return hashlib.sha256(block\_info.encode()).hexdigest()

class Blockchain:

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

"""

Constructor for Blockchain class.

"""

self.chain = [self.create\_genesis\_block()] # Initialize blockchain with genesis block

self.difficulty = 2 # Set difficulty level for mining

def create\_genesis\_block(self):

"""

Create the first block (genesis block) of the blockchain.

"""

# The genesis block has index 0, a timestamp, static data "Genesis Block",

# and a previous hash of "0" (since it's the first block)

return Block(0, time.time(), "Genesis Block", "0")

def get\_last\_block(self):

"""

Get the last block in the blockchain.

"""

# The last block in the chain is simply the last element of the chain list

return self.chain[-1]

def mine\_block(self, data):

"""

Mine a new block with the provided data.

Args:

data: Data or transaction details for the new block.

"""

# Get the last block in the chain

last\_block = self.get\_last\_block()

# Create a new block with incremented index, current timestamp, provided data,

# and the hash of the previous block as the previous hash

new\_block = Block(last\_block.index + 1, time.time(), data, last\_block.hash)

# Proof of Work: Find a hash that starts with the required number of zeros (difficulty)

while not new\_block.hash.startswith('0' \* self.difficulty):

new\_block.timestamp = time.time() # Update timestamp for each attempt

new\_block.hash = new\_block.calculate\_hash() # Recalculate the hash

# Once a valid hash is found (meets difficulty requirement), add the block to the chain

self.chain.append(new\_block)

return new\_block

def dump\_blockchain(self):

"""

Dump the entire blockchain by printing the contents of each block.

"""

print("Dumping Blockchain:")

for block in self.chain:

# Print details of each block: index, timestamp, data, previous hash, and hash

print(f"Index: {block.index}")

print(f"Timestamp: {block.timestamp}")

print(f"Data: {block.data}")

print(f"Previous Hash: {block.previous\_hash}")

print(f"Hash: {block.hash}")

print()

# Create a blockchain instance

blockchain = Blockchain()

# Print the initial state of the blockchain (genesis block)

print("Initial Blockchain State (Genesis Block):")

blockchain.dump\_blockchain()

# Mine new blocks with some example data

blockchain.mine\_block("Transaction 1")

blockchain.mine\_block("Transaction 2")

blockchain.mine\_block("Transaction 3")

# Print the updated state of the blockchain after mining

print("Updated Blockchain State:")

blockchain.dump\_blockchain()

**Output:-**

****