**Practical 1**

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

**Practical 1 a)**

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

**Code:**

#pip install pycryptodome

#1A.- A simple client class that generates the private and public keys by using the built-in Python RSA algorithm and test it.

import Crypto

import binascii

from Crypto.PublicKey import RSA

from Crypto.Signature import PKCS1\_v1\_5

class Client:

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

# Creating random number for key

random = Crypto.Random.new().read

# Creating new public key and private key

self.\_private\_key = RSA.generate(1024, random)

self.\_public\_key = self.\_private\_key.publickey()

self.\_signer = PKCS1\_v1\_5.new(self.\_private\_key)

@property

def identity(self):

return binascii.hexlify(self.\_public\_key.exportKey(format='DER')).decode('ascii')

Demo = Client()

print(Demo.identity)

**Output:**

**A computer screen shot

Description automatically generated**

**Practical 1 b)**

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

**Code:**

#1B.- A transaction class to send and receive money and test it.

import Crypto

import binascii

import datetime

import collections

from Crypto.PublicKey import RSA

from Crypto.Signature import PKCS1\_v1\_5

from Crypto.Hash import SHA

class Client:

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

# Creating random number for key

random = Crypto.Random.new().read

# Creating new public key and private key

self.\_private\_key = RSA.generate(1024, random)

self.\_public\_key = self.\_private\_key.publickey()

self.\_signer = PKCS1\_v1\_5.new(self.\_private\_key)

@property

def identity(self):

return binascii.hexlify(self.\_public\_key.exportKey(format='DER')).decode('ascii')

class Transaction:

def \_\_init\_\_(self, sender, receiver, value):

self.sender = sender

self.receiver = receiver

self.value = value

self.time = datetime.datetime.now()

def to\_dict(self):

if self.sender == "Genesis":

identity = "Genesis"

else:

identity = self.sender.identity

return collections.OrderedDict({

'sender': identity,

'receiver': self.receiver,

'value': self.value,

'time': self.time

})

def sign\_transaction(self):

private\_key = self.sender.\_private\_key

signer = PKCS1\_v1\_5.new(private\_key)

h = SHA.new(str(self.to\_dict()).encode('utf8'))

return binascii.hexlify(signer.sign(h)).decode('ascii')

Ninad = Client()

print("-"\*50)

print("Ninad Key")

print(Ninad.identity)

KS = Client()

print("-"\*50)

print("KS Key")

print(KS.identity)

t = Transaction(Ninad, KS.identity, 10.0)

print("-"\*50)

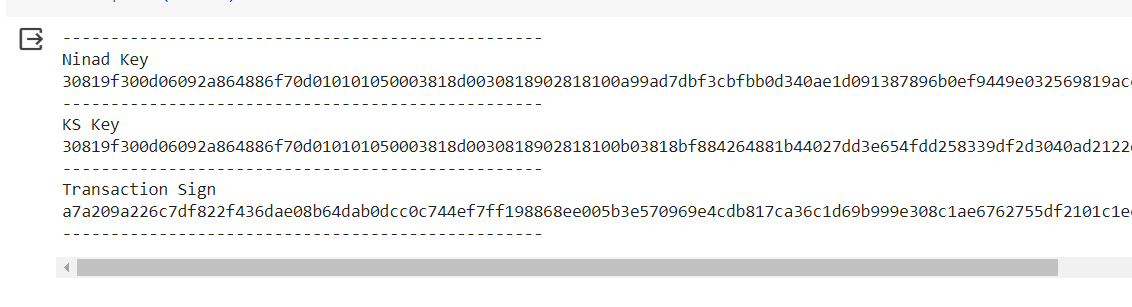
print("Transaction Sign")

signature = t.sign\_transaction()

print(signature)

print("-"\*50)

**Output:**

****

**Practical 1 c)**

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

**Code:**

#!pip install pycryptodome

import Crypto

import binascii

from Crypto.PublicKey import RSA

from Crypto.Hash import SHA

from Crypto.Signature import PKCS1\_v1\_5

import datetime

import collections

import hashlib

from hashlib import sha256

class Client:

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

# Creating random number for key

random = Crypto.Random.new().read

# Creating new public key and private key

self.\_private\_key = RSA.generate(1024, random)

self.\_public\_key = self.\_private\_key.publickey()

self.\_signer = PKCS1\_v1\_5.new(self.\_private\_key)

@property

def identity(self):

return binascii.hexlify(self.\_public\_key.exportKey(format="DER")).decode(

"ascii"

)

class Transaction:

def \_\_init\_\_(self, sender, receiver, value):

self.sender = sender

self.receiver = receiver

self.value = value

self.time = datetime.datetime.now()

def to\_dict(self):

if self.sender == "Genesis":

identity = "Genesis"

else:

identity = self.sender.identity

return collections.OrderedDict(

{

"sender": identity,

"receiver": self.receiver,

"value": self.value,

"time": self.time,

}

)

def sign\_transaction(self):

private\_key = self.sender.\_private\_key

signer = PKCS1\_v1\_5.new(private\_key)

h = SHA.new(str(self.to\_dict()).encode("utf8"))

return binascii.hexlify(signer.sign(h)).decode("ascii")

def sha256(message):

return hashlib.sha256(message.encode("ascii")).hexdigest

def mine(message, difficulty=1):

assert difficulty >= 1

prefix = "1" \* difficulty

for i in range(1000):

digest = sha256(str(hash(message)) + str(i))

if digest.startwith(prefix):

print("after" + str(i) + "iteration found nonce:" + digest)

return digest

class Block:

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

self.verified\_transactions = []

self.previous\_block\_hash = ""

self.Nonce = ""

last\_block\_hash = ""

def display\_transaction(transaction):

dict = transaction.to\_dict()

print("Sender: " + dict["sender"])

print("-----")

print("Receiver: " + dict["receiver"])

print("-----")

print("Value: " + str(dict["value"]))

print("-----")

print("Time: " + str(dict["time"]))

print("-----")

TPCoins = []

def dump\_blockchain(self):

print("Number of blocks in chain" + str(len(self)))

for x in range(len(Block.TPCoins)):

block\_temp = Block.TPCoins[x]

print("block #" + str(x))

for transaction in block\_temp.verified\_transactions:

Block.display\_transaction(transaction)

print("-------")

last\_transaction\_index = 0

transactions = []

Ninad = Client()

ks = Client()

vighnesh = Client()

sairaj = Client()

t1 = Transaction(Ninad, ks.identity, 15.0)

t1.sign\_transaction()

transactions.append(t1)

t2 = Transaction(Ninad, vighnesh.identity, 6.0)

t2.sign\_transaction()

transactions.append(t2)

t3 = Transaction(Ninad, sairaj.identity, 16.0)

t3.sign\_transaction()

transactions.append(t3)

t4 = Transaction(vighnesh, Ninad.identity, 8.0)

t4.sign\_transaction()

transactions.append(t4)

t5 = Transaction(vighnesh, ks.identity, 19.0)

t5.sign\_transaction()

transactions.append(t5)

t6 = Transaction(vighnesh, sairaj.identity, 35.0)

t6.sign\_transaction()

transactions.append(t6)

t7 = Transaction(sairaj, vighnesh.identity, 5.0)

t7.sign\_transaction()

transactions.append(t7)

t8 = Transaction(sairaj, Ninad.identity, 12.0)

t8.sign\_transaction()

transactions.append(t8)

t9 = Transaction(sairaj, ks.identity, 25.0)

t9.sign\_transaction()

transactions.append(t9)

t10 = Transaction(Ninad, ks.identity, 1.0)

t10.sign\_transaction()

transactions.append(t10)

for transaction in transactions:

display\_transaction(transaction)

print("\*" \* 50)

**Output:**



A screenshot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

**Practical 1 d)**

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

**Code:**

# Aim 1D - Create a blockchain, a genesis block and execute it.

#!pip install pycryptodome

import Crypto

import binascii

import datetime

import collections

from Crypto.PublicKey import RSA

from Crypto.Hash import SHA

from Crypto.Signature import PKCS1\_v1\_5

class Client:

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

# Creating random number for key

random = Crypto.Random.new().read

# Creating new public key and private key

self.\_private\_key = RSA.generate(1024, random)

self.\_public\_key = self.\_private\_key.publickey()

self.\_signer = PKCS1\_v1\_5.new(self.\_private\_key)

@property

def identity(self):

return binascii.hexlify(self.\_public\_key.exportKey(format="DER")).decode(

"ascii"

)

class Transaction:

def \_\_init\_\_(self, sender, receiver, value):

self.sender = sender

self.receiver = receiver

self.value = value

self.time = datetime.datetime.now()

def to\_dict(self):

if self.sender == "Genesis":

identity = "Genesis"

else:

identity = self.sender.identity

return collections.OrderedDict(

{

"sender": identity,

"receiver": self.receiver,

"value": self.value,

"time": self.time,

}

)

def sign\_transaction(self):

private\_key = self.sender.\_private\_key

signer = PKCS1\_v1\_5.new(private\_key)

h = SHA.new(str(self.to\_dict()).encode("utf8"))

return binascii.hexlify(signer.sign(h)).decode("ascii")

class Block:

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

self.verified\_transactions = []

self.previous\_block\_hash = ""

self.Nonce = ""

last\_block\_hash = ""

def display\_transaction(transaction):

dict = transaction.to\_dict()

print("Sender: " + dict["sender"])

print("-----")

print("Receiver: " + dict["receiver"]) # Corrected typo

print("-----")

print("Value: " + str(dict["value"]))

print("-----")

print("Time: " + str(dict["time"]))

print("-----")

Ninad = Client()

t0 = Transaction("Genesis", Ninad.identity, 500.0)

block0 = Block()

block0.previous\_block\_hash = None

Nonce = None

block0.verified\_transactions.append(t0)

digest = hash(block0)

last\_block\_hash = digest

TPCoins = []

def dump\_blockchain(self):

print("Number of blocks in chain: " + str(len(self)))

for x in range(len(TPCoins)):

block\_temp = TPCoins[x]

print("block #" + str(x))

for transaction in block\_temp.verified\_transactions:

Block.display\_transaction(transaction)

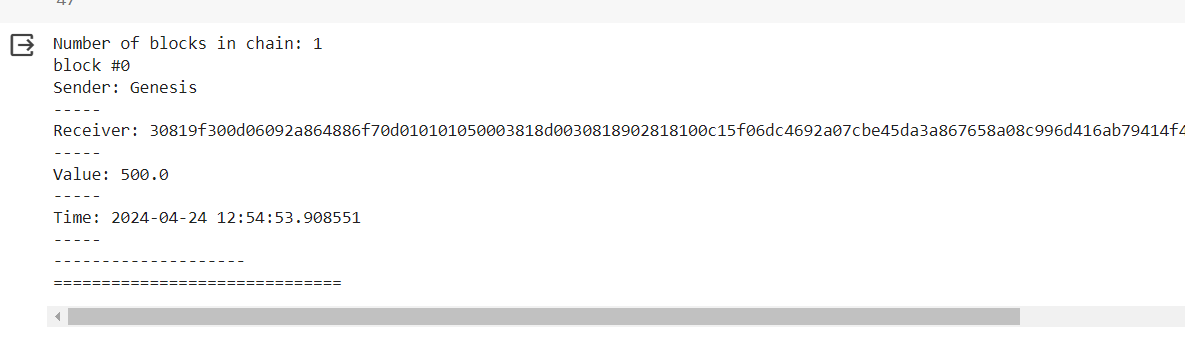
print("-" \* 20)

print("=" \* 30)

TPCoins.append(block0)

dump\_blockchain(TPCoins)

**Output:**



**Practical 1 e)**

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

**Code:**

# 1e. Create a mining function and test it.

import hashlib

def sha256(message):

return hashlib.sha256(message.encode("ascii")).hexdigest()

def mine(message, difficulty=1):

assert difficulty >= 1

prefix = "1" \* difficulty

for i in range(1000):

digest = sha256(str(hash(message)) + str(i))

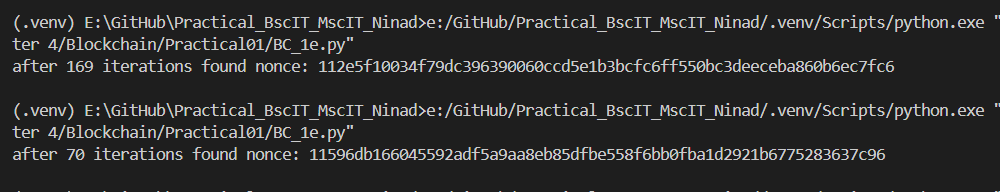
if digest.startswith(prefix):

print("after " + str(i) + " iterations found nonce: " + digest)

return digest

mine("Testmessage", 2)

**Output:**

****

**Practical 1 f)**

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

**Code:**

#!pip install pycryptodome

import Crypto

import binascii

import datetime

import collections

from Crypto.PublicKey import RSA

from Crypto.Hash import SHA

from Crypto.Signature import PKCS1\_v1\_5

import hashlib

class Client:

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

# Creating random number for key

random = Crypto.Random.new().read

# Creating new public key and private key

self.\_private\_key = RSA.generate(1024, random)

self.\_public\_key = self.\_private\_key.publickey()

self.\_signer = PKCS1\_v1\_5.new(self.\_private\_key)

@property

def identity(self):

return binascii.hexlify(self.\_public\_key.exportKey(format="DER")).decode(

"ascii"

)

class Transaction:

def \_\_init\_\_(self, sender, receiver, value):

self.sender = sender

self.receiver = receiver

self.value = value

self.time = datetime.datetime.now()

def to\_dict(self):

if self.sender == "Genesis":

identity = "Genesis"

else:

identity = self.sender.identity

return collections.OrderedDict(

{

"sender": identity,

"receiver": self.receiver,

"value": self.value,

"time": self.time,

}

)

def sign\_transaction(self):

private\_key = self.sender.\_private\_key

signer = PKCS1\_v1\_5.new(private\_key)

h = SHA.new(str(self.to\_dict()).encode("utf8"))

return binascii.hexlify(signer.sign(h)).decode("ascii")

def sha256(message):

return hashlib.sha256(message.encode("ascii")).hexdigest

def mine(message, difficulty=1):

assert difficulty >= 1

prefix = "1" \* difficulty

for i in range(1000):

digest = sha256(str(hash(message)) + str(i))

if str(digest).startswith(prefix):

print("after " + str(i) + " iteration found nonce:" + digest)

return digest

class Block:

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

self.verified\_transactions = []

self.previous\_block\_hash = ""

self.Nonce = ""

last\_block\_hash = ""

def display\_transaction(transaction):

dict = transaction.to\_dict()

print("Sender: " + dict["sender"])

print("-----")

print("Receiver: " + dict["receiver"])

print("-----")

print("Value: " + str(dict["value"]))

print("-----")

print("Time: " + str(dict["time"]))

print("-----")

TPCoins = []

def dump\_blockchain(self):

print("Number of blocks in chain" + str(len(self)))

for x in range(len(TPCoins)):

block\_temp = TPCoins[x]

print("block #" + str(x))

for transaction in block\_temp.verified\_transactions:

display\_transaction(transaction)

print("-------")

print("=" \* 50)

last\_transaction\_index = 0

transactions = []

Ninad = Client()

ks = Client()

vighnesh = Client()

sairaj = Client()

t1 = Transaction(Ninad, ks.identity, 15.0)

t1.sign\_transaction()

transactions.append(t1)

t2 = Transaction(Ninad, vighnesh.identity, 6.0)

t2.sign\_transaction()

transactions.append(t2)

t3 = Transaction(Ninad, sairaj.identity, 16.0)

t3.sign\_transaction()

transactions.append(t3)

t4 = Transaction(vighnesh, Ninad.identity, 8.0)

t4.sign\_transaction()

transactions.append(t4)

t5 = Transaction(vighnesh, ks.identity, 19.0)

t5.sign\_transaction()

transactions.append(t5)

t6 = Transaction(vighnesh, sairaj.identity, 35.0)

t6.sign\_transaction()

transactions.append(t6)

t7 = Transaction(sairaj, vighnesh.identity, 5.0)

t7.sign\_transaction()

transactions.append(t7)

t8 = Transaction(sairaj, Ninad.identity, 12.0)

t8.sign\_transaction()

transactions.append(t8)

t9 = Transaction(sairaj, ks.identity, 25.0)

t9.sign\_transaction()

transactions.append(t9)

t10 = Transaction(Ninad, ks.identity, 1.0)

t10.sign\_transaction()

transactions.append(t10)

# miner 1 adds block

block = Block()

for i in range(3): # Limit loop iterations to list length

temp\_transaction = transactions[last\_transaction\_index]

# validatetransaction

# if valid

block.verified\_transactions.append(temp\_transaction)

last\_transaction\_index += 1

block.previous\_block\_hash = last\_block\_hash

block.Nonce = mine(block, 2)

digest = hash(block)

TPCoins.append(block)

last\_block\_hash = digest

###

# miner 2 adds block

block = Block()

for i in range(3):

temp\_transaction = transactions[last\_transaction\_index]

# validatetransaction

# if valid

block.verified\_transactions.append(temp\_transaction)

last\_transaction\_index += 1

block.previous\_block\_hash = last\_block\_hash

block.Nonce = mine(block, 2)

digest = hash(block)

TPCoins.append(block)

last\_block\_hash = digest

###

# miner 3 adds block

block = Block()

for i in range(3):

temp\_transaction = transactions[last\_transaction\_index]

# validatetransaction

# if valid

block.verified\_transactions.append(temp\_transaction)

last\_transaction\_index += 1

block.previous\_block\_hash = last\_block\_hash

block.Nonce = mine(block, 2)

digest = hash(block)

TPCoins.append(block)

last\_block\_hash = digest

dump\_blockchain(TPCoins)

**Output:**

****

A screenshot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Practical 2**

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

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

**Code:**

**Output:**

**Practical 3**

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

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

**Code:**

**Output:**

**Practical 4**

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

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

**Code:**

**Output:**

**Practical 5**

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

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

**Code:**

**Output:**

**Practical 6**

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

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

**Code:**

**Output:**

**Practical 7**

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

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

**Code:**

**Output:**

**Practical 8**

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

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

**Code:**

**Output:**