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

Code:

```
!pip3 install Crypto
!pip install pycryptodomes
import hashlib
import random
import binascii
import datetime
import collections
from Crypto.PublicKey import RSA
from Crypto import Random
from Crypto.Cipher import PKCS1_v1_5

class Client:
  def __init__(self):
    random = Random.new().read
    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')

Dinesh = Client()
print("sender ", Dinesh.identity)
```

Aim: Create multiple transactions and display them

Code:

```
!pip install pycryptodomes
import hashlib
import binascii
import datetime
import collections
from Crypto.PublicKey import RSA
from Crypto import Random
from Crypto.Cipher import PKCS1_v1_5
from collections import OrderedDict
import Crypto
import Crypto.Random
from Crypto.Hash import SHA
from Crypto.Signature import PKCS1_v1_5
class Client:
   def __init__(self):
     random = Random.new().read
     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, recipient, value):
      self.sender = sender
      self.recipient = recipient
      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,
        'recipient': self.recipient,
        '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 display_transaction(transaction):
      #for transaction in transactions:
      dict = transaction.to_dict()
      print ("sender: " + dict['sender'])
      print ('-----')
      print ("recipient: " + dict['recipient'])
      print ('-----')
      print ("value: " + str(dict['value']))
      print ('-----')
      print ("time: " + str(dict['time']))
      print ('-----')
transactions = []
A = Client()
B = Client()
t1 = Transaction(
   A,
```

```
  B.identity,
  15.0
)
t1.sign_transaction()
display_transaction (t1)
```

Aim: **Create a transaction class to send and receive money and test it**

Code:

```
!pip install pycryptodomes
import hashlib
import binascii
import datetime
import collections

from Crypto.PublicKey import RSA
from Crypto import Random
from Crypto.Cipher import PKCS1_v1_5
from collections import OrderedDict
import Crypto
import Crypto.Random
from Crypto.Hash import SHA
from Crypto.Signature import PKCS1_v1_5
class Client:
   def __init__(self):
     random = Random.new().read
     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, recipient, value):
    self.sender = sender
    self.recipient = recipient
    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,
      'recipient': self.recipient,
      '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 display_transaction(transaction):
    #for transaction in transactions:
    dict = transaction.to_dict()
    print ("sender: " + dict['sender'])
```

```
        print ('-----')
        print ("recipient: " + dict['recipient'])
        print ('-----')
        print ("value: " + str(dict['value']))
        print ('-----')
        print ("time: " + str(dict['time']))
        print ('-----')

transactions = []

Dinesh = Client()
Ramesh = Client()
Suresh = Client()

t1 = Transaction(
   Dinesh,
   Ramesh.identity,
   15.0
)

t1.sign_transaction()
transactions.append(t1)

t2 = Transaction(
   Ramesh,
   Suresh.identity,
   25.0
)
t2.sign_transaction()
transactions.append(t2)
```

```
t3 = Transaction(
  Ramesh,
  Suresh.identity,
  200.0
)
t3.sign_transaction()
transactions.append(t3)

tn=1
for t in transactions:
  print("Transaction #",tn)
  display_transaction (t)
  tn=tn+1
  print ('--------------')
```

Aim: **<u>Create a blockchain, a genesis block and execute it.</u>**

<u>Code:</u>

```
!pip install pycryptodomes
import hashlib
import binascii
import datetime
import collections

from Crypto.PublicKey import RSA
from Crypto import Random
from Crypto.Cipher import PKCS1_v1_5
from collections import OrderedDict
import Crypto
import Crypto.Random
```

```
from Crypto.Hash import SHA
from Crypto.Signature import PKCS1_v1_5

class Client:
    def __init__(self):
      random = Random.new().read
      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, recipient, value):
       self.sender = sender
       self.recipient = recipient
       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,
          'recipient': self.recipient,
          '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 display_transaction(transaction):
    #for transaction in transactions:
    dict = transaction.to_dict()
    print ("sender: " + dict['sender'])
    print ('-----')
    print ("recipient: " + dict['recipient'])
    print ('-----')
    print ("value: " + str(dict['value']))
    print ('-----')
    print ("time: " + str(dict['time']))
    print ('-----')

def dump_blockchain (self):
  print ("Number of blocks in the 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 ('=====================================')

class Block:
  def __init__(self):
```

```
    self.verified_transactions = []
    self.previous_block_hash = ""
    self.Nonce = ""

Dinesh = Client()

t0 = Transaction (
  "Genesis",
  Dinesh.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 = []
TPCoins.append (block0)

dump_blockchain(TPCoins)
```

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

Code:

```
!pip install pycryptodomes
import hashlib

def sha256(message):
    return hashlib.sha256(message.encode('ascii')).hexdigest()
```

```
def mine(message, difficulty=1):
  assert difficulty >= 1
  #if(difficulty <1):
  #     return
  #'1'*2=> '11'
  prefix = '1' * difficulty
  print("prefix",prefix)
  for i in range(1000):
    digest = sha256(str(hash(message)) + str(i))
    print("testing=>"+digest)
    if digest.startswith(prefix):
      print ("after " + str(i) + " iterations found nonce: "+ digest)
      return i #i= nonce value

mine ("test message",2)
```

Aim: **<u>Add blocks to the miner and dump the blockchain.</u>**

Code:

```
!pip install pycryptodomes
import hashlib
import random
import binascii
import datetime
import collections

from Crypto.PublicKey import RSA
from Crypto import Random
from Crypto.Cipher import PKCS1_v1_5
from collections import OrderedDict
import Crypto
```

```
import Crypto.Random
from Crypto.Hash import SHA
from Crypto.Signature import PKCS1_v1_5

class Client:
   def __init__(self):
     random = Random.new().read
     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, recipient, value):
      self.sender = sender
      self.recipient = recipient
      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,
         'recipient': self.recipient,
         '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 display_transaction(transaction):
      #for transaction in transactions:
      dict = transaction.to_dict()
      print ("sender: " + dict['sender'])
      print ('-----')
      print ("recipient: " + dict['recipient'])
      print ('-----')
      print ("value: " + str(dict['value']))
      print ('-----')
      print ("time: " + str(dict['time']))
      print ('-----')

def dump_blockchain (self):
   print ("Number of blocks in the 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 ('=====================================')

class Block:
```

```
  def __init__(self):
    self.verified_transactions = []
    self.previous_block_hash = ""
    self.Nonce = ""

def sha256(message):
    return hashlib.sha256(message.encode('ascii')).hexdigest()

def mine(message, difficulty=1):
  assert difficulty >= 1
  #if(difficulty <1):
  #      return
  #'1'*3=> '111'
  prefix = '1' * difficulty
  for i in range(1000):
    digest = sha256(str(hash(message)) + str(i))
    if digest.startswith(prefix):
      return i #i= nonce value

A = Client()
B =Client()
C =Client()
t0 = Transaction (
  "Genesis",
  A.identity,
  500.0
)

t1 = Transaction (
  A,
  B.identity,
```

```
  40.0
)
t2 = Transaction (
  A,
  C.identity,
  70.0
)
t3 = Transaction (
  B,
  C.identity,
  700.0
)
#blockchain
TPCoins = []

block0 = Block()
block0.previous_block_hash = None
Nonce = None
block0.verified_transactions.append (t0)
digest = hash (block0)
last_block_hash = digest #last_block_hash it is hash of block0
TPCoins.append (block0)

block1 = Block()
block1.previous_block_hash = last_block_hash
block1.verified_transactions.append (t1)
block1.verified_transactions.append (t2)
block1.Nonce=mine (block1, 2)
digest = hash (block1)
last_block_hash = digest
TPCoins.append (block1)
```

```
block2 = Block()
block2.previous_block_hash = last_block_hash
block2.verified_transactions.append (t3)
Nonce = mine (block2, 2)
block2.Nonce=mine (block2, 2)
digest = hash (block2)
last_block_hash = digest
TPCoins.append (block2)

dump_blockchain(TPCoins)
```

practical 2:

Aim: **write a solidity program for variables, operators, loops, decision making and string.**

a. <u>Variables:</u>

Code:

```
pragma solidity ^0.8.25;
contract SolidityTest {
uint storedData; // State variable
constructor() public {
storedData = 10;
}
function getResult() public view returns(uint){
uint a = 1; // local variable
uint b = 2;
uint result = a + b;
return result; //access the state variable
}
}
```

b. **State Variable:**

code:

```
pragma solidity ^0.8.25;

contract Solidity_var_Test {
uint8 public state_var;
constructor() public {
state_var = 16;
}
}
```

c. **Local Variable**

code:

```
pragma solidity ^0.8.25;
contract Solidity_var_Test {
function getResult() public view returns(uint){
uint local_var1 = 1;
uint local_var2 = 2;
uint result = local_var1 + local_var2;
return result;
}
}
```

d.global variable

```
pragma solidity ^0.8.25;
contract Test {
address public admin;
```

```
constructor() public {
admin = msg.sender;
}
}
```