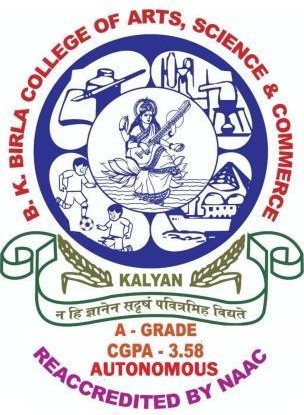
B. K. BIRLA COLLEGE OF ARTS, SCIENCE & COMMERCE (AUTONOMOUS), KALYAN

DEPARTMENT OF INFORMATION TECHNOLOGY



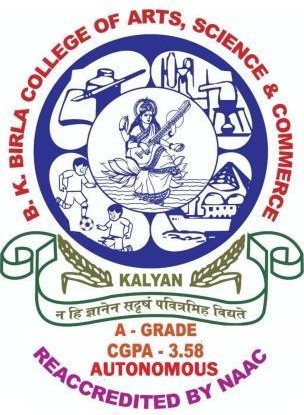
|  |  |
| --- | --- |
| **Student Name:** |  |
| **Student ID:** |  |
| **Class:** |  |
| **Subject:** |  |

# B. K. BIRLA COLLEGE OF ARTS, SCIENCE & COMMERCE (AUTONOMOUS), KALYAN

*(Affiliated to University of Mumbai)*

**KALYAN-MAHARASHTRA-421301**

# DEPARTMENT OF INFORMATION TECHNOLOGY



**CERTIFICATE**

This is to certify that Mr/Ms **NAME OF THE STUDENT** bearing Seat. No: **(SEAT NUMBER )**, in class **COURSE NAME** has successfully completed practical of the subject

Teacher’s Signature:

Place:

Date: College Seal

**INDEX**

|  |  |
| --- | --- |
| **SR. NO** | **PRACTICAL NAME** |
| 1. | A simple client class that generates the private and public keys by using the built- in Python RSA algorithm and test it. |
| 2. | A transaction class to send and receive money and test it. |
| 3. | Create multiple transactions and display them. |
| 4. | Create a blockchain, a genesis block and execute it. |
| 5. | Create a mining function and test it. |
| 6. | Add blocks to the miner and dump the blockchain. |
| 7. | Implement and demonstrate the use of the following in Solidity:  Variable, Operators, Loops, Decision Making, Strings, Arrays, Enums, Structs, Mappings, Conversions, Ether Units, Special Variables. |
| 8. | Demonstrate the use of Bitcoin Core API. |

**Practical No: 1**

**Aim:** A simple client class that generates the private and public keys by using the built- in

Python RSA algorithm and test it.

**Program:**

# following imports are required by PKI

!pip3 install pycryptodome

import hashlib

import random

import string

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')

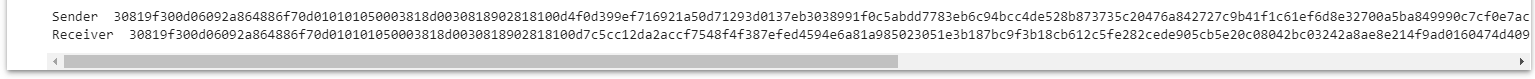
Account\_1 = Client()

Account\_2 = Client()

print ("Sender ",Account\_1.identity)

print ("Receiver ",Account\_2.identity)

**Output:**



**Practical No: 2**

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

**Program:**

# following imports are required by PKI

!pip3 install pycryptodome

!pip3 install crypto

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 ('-----')

sa = Client()

rb = Client()

t1 = Transaction(

sa,

rb.identity,

15.0

)

t1.sign\_transaction()

display\_transaction (t1)

sa2 = Client()

rb2 = Client()

t2 = Transaction(

sa,

rb.identity,

15.0

)

t2.sign\_transaction()

display\_transaction (t2)

**Output:**



**Practical No: 3**

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

**Program:**

# following imports are required by PKI

!pip3 install pycryptodome

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 ('-----')

transactions = []

a = Client()

b = Client()

c = Client()

t1 = Transaction(

a,

b.identity,

15.0

)

t1.sign\_transaction()

transactions.append(t1)

t2 = Transaction(

b,

c.identity,

25.0

)

t2.sign\_transaction()

transactions.append(t2)

t3 = Transaction(

a,

c.identity,

200.0

)

t3.sign\_transaction()

transactions.append(t3)

tn=1

for t in transactions:#t1 t2 t3

print("Transaction #",tn)

display\_transaction (t)

tn=tn+1

print ('--------------')

**Output:**



**Practical No: 4**

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

**Program:**

# following imports are required by PKI

!pip3 install pycryptodome

import hashlib

import random

import binascii

import numpy as np

import pandas as pd

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 = ""

a = Client()

t0 = Transaction (

"Genesis",

a.identity,

500.0

)

block0 = Block()

block0.previous\_block\_hash = None

Nonce = None

block0.verified\_transactions.append (t0)

digest = hash (block0)

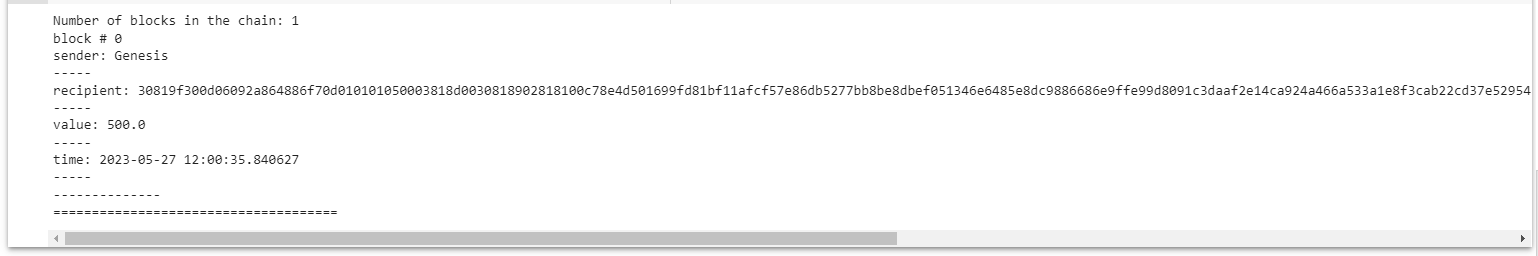
last\_block\_hash = digest

TPCoins = [] #coinbase

TPCoins.append (block0)

dump\_blockchain(TPCoins)

**Output:**



**Practical No: 5**

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

**Program:**

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'\*3=> '111'

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):

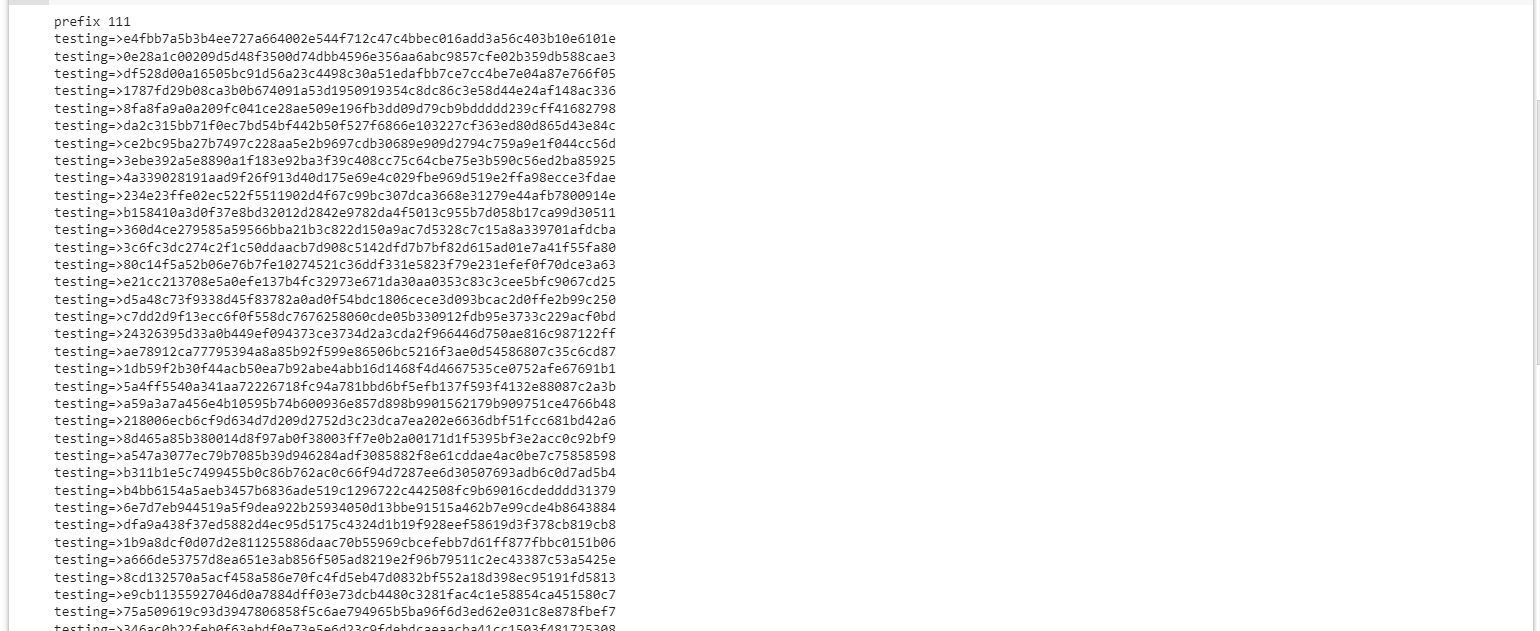
print ("after " + str(i) + " iterations found nonce: "+ digest)

return i #i= nonce value

n=mine ("test message",3)

print(n)

**Output:**



**Practical No: 6**

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

**Program:**

# following imports are required by PKI

!pip3 install pycryptodome

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

Dinesh = Client()

Ramesh =Client()

Vikas =Client()

t0 = Transaction (

"Genesis",

Dinesh.identity,

500.0

)

t1 = Transaction (

Ramesh,

Dinesh.identity,

40.0

)

t2 = Transaction (

Ramesh,

Dinesh.identity,

70.0

)

t3 = Transaction (

Vikas,

Ramesh.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)

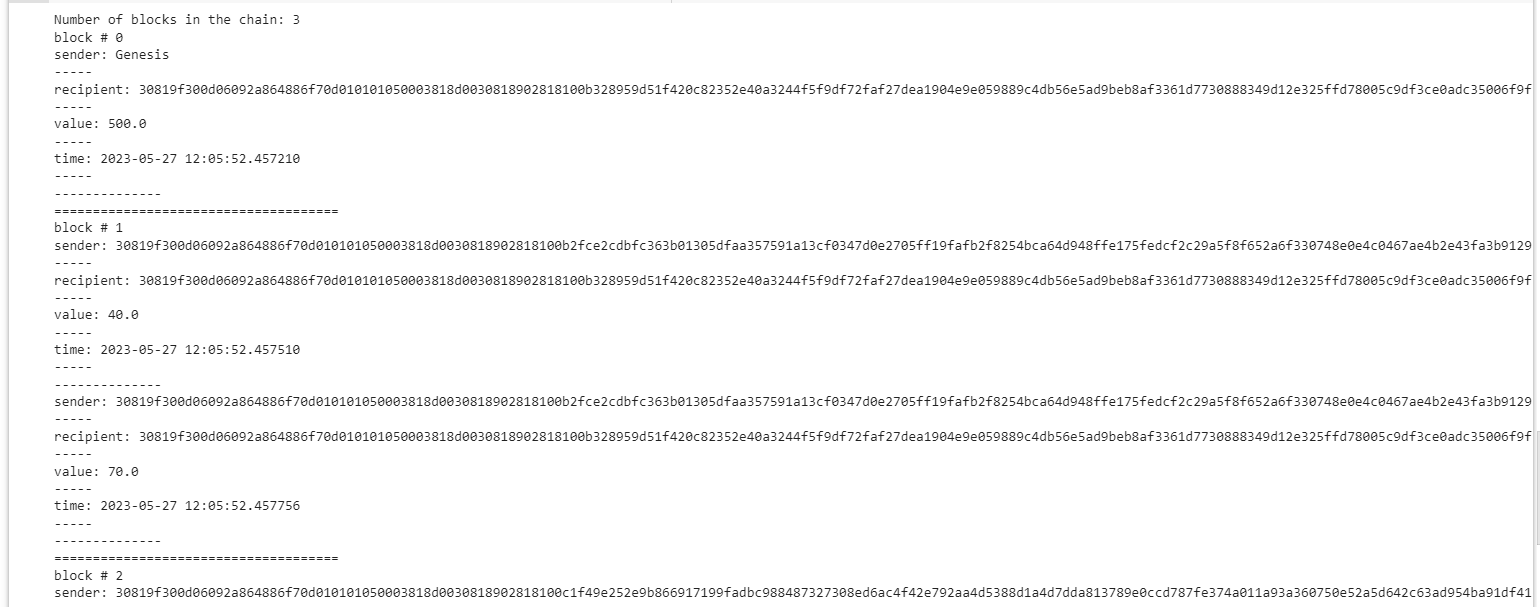
digest = hash (block2)

last\_block\_hash = digest

TPCoins.append (block2)

dump\_blockchain(TPCoins)

**Output:**



**Practical No: 7**

**Aim:** Implement and demonstrate the use of the following in Solidity:Variable, Operators,

Loops, Decision Making, Strings, Arrays, Enums, Structs, Mappings, Conversions,

Ether Units, Special Variables.

**Aim: Types of Variable**

**Program:**

pragma solidity ^0.5.0;

contract Pract1 {

  // State variable

  int x = 15;

  // Global variable

  int public y = 10;

  // Function to get the value

  function getValue(int z) public {

    y = y + z;

  }

  // Function to show the value

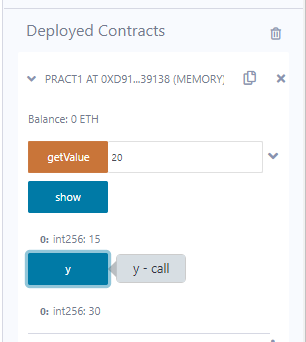
  function show() public view returns (int) {

    return x;

  }

}

**Output:**



**Aim: Relational Operators**

**Program:**

pragma solidity ^0.5.0;

contract Pract2 {

  // State variables

  bool public a = true;

  bool public b = false;

  // Boolean operators

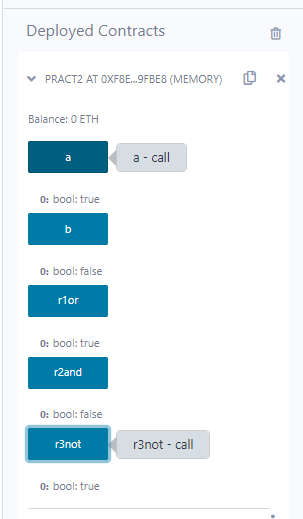
  bool public r1or = a || b;

  bool public r2and = a && b;

  bool public r3not = !b;

}

**Output:**



**Aim: For Loop**

**Program:**

pragma solidity ^0.5.0;

contract Pract3 {

  function test(int s, int e) public view returns(int) {

    int i;

    int sum = 0;

    // For loop

    for (i = s; i <= e; i++) {

      sum += i;

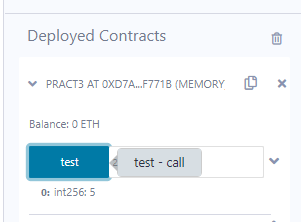
    }

    return sum;

  }

}

**Output:**



**Aim: While Loop**

**Program:**

pragma solidity ^0.5.0;

contract Pract3 {

  function test(int s, int e) public view returns(int) {

    int i;

    int sum = 0;

    // While loop

    i = s;

    while (i <= e) {

      sum += i;

      i++;

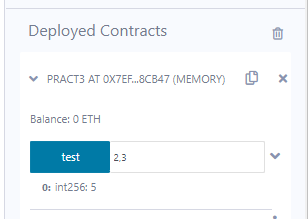
    }

    return sum;

  }

}

**Output:**



**Aim: Do-While Loop**

**Program:**

pragma solidity ^0.5.0;

contract Pract3 {

  function test(int s, int e) public view returns(int) {

    int i;

    int sum = 0;

    // Do-while loop

    i = s;

    do {

      sum += i;

      i++;

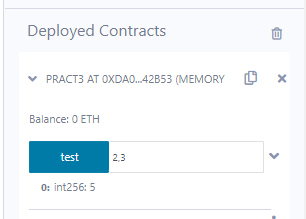
    } while (i <= e);

    return sum;

  }

}

**Output:**



**Aim: If Else**

**Program:**

pragma solidity ^0.5.0;

contract Pract4 {

  function test(int x) public view returns(string memory) {

    // Check if the number is even

    if (x % 2 == 0) {

      return "Number is even";

    } else {

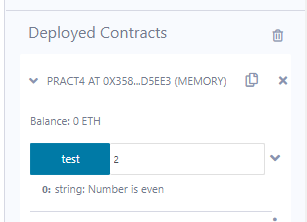
      return "Number is odd";

    }

  }

}

**Output:**



**Aim: string**

**Program:**

pragma solidity ^0.5.0;

contract IfElse {

    uint public age;

    string public status;

    function checkAge(uint \_age) public returns (string memory) {

        age = \_age;

        if (age >= 18) {

            status = "You are mature.";

        } else {

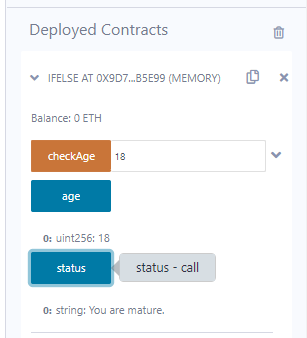
            status = "You are a minor.";

        }

    }

}

**Output:**



**Aim: Array**

**Program:**

pragma solidity ^0.5.0;

contract Types {

  uint[5] data;

  constructor() public {

    data = [uint(10), 20, 30, 40, 50];

  }

  function array\_example() public view returns (uint, uint) {

    // Return the element at index 0 of the `data` array and the element at index 4.

    return (data[0], data[4]);

  }

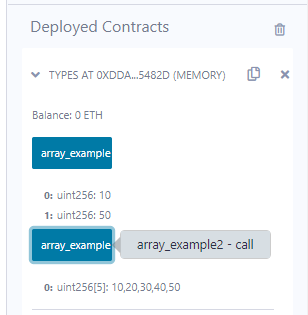
  function array\_example2() public view returns (uint[5] memory) {

    return data;

  }

}

**Output:**



**Aim: Enum**

**Program:**

pragma solidity ^0.5.0;

contract Types {

  // Declare an enum called `week\_days` with 7 values.

  enum week\_days {

    Monday,

    Tuesday,

    Wednesday,

    Thursday,

    Friday,

    Saturday,

    Sunday

  }

  // Declare a variable called `choice` of type `week\_days`.

  week\_days choice;

  function set\_value() public {

    // Set the value of the `choice` variable to `week\_days.Thursday`.

    choice = week\_days.Thursday;

  }

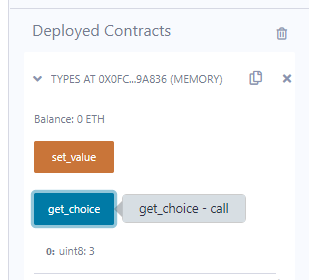
  function get\_choice() public view returns (week\_days) {

    return choice;

  }

}

**Output:**



**Aim: Arithmetic Operations**

**Program:**

pragma solidity ^0.5.0;

contract SolidityTest {

  uint16 public a = 20;

  uint16 public b = 10;

  // sum of `a` and `b`.

  uint public sum = a + b;

  // difference of `a` and `b`.

  uint public diff = a - b;

  // product of `a` and `b`.

  uint public mul = a \* b;

  // quotient of `a` and `b`.

  uint public div = a / b;

  // modulus of `a` and `b`.

  uint public mod = a % b;

  // decrement value of `b`.

  uint public dec = --b;

  // increment value of `a`.

  uint public inc = ++a;

}

**Output:**



**Aim: Structure**

**Program:**

pragma solidity ^0.5.0;

contract test {

  // This struct defines a book.

  struct Book {

    string title;

    string author;

    uint book\_id;

  }

  // This variable stores a book.

  Book book;

  // This function sets the values of the book's members.

  function setBook() public {

    book.title = "Learn Java";

    book.author = "TP";

    book.book\_id = 1;

  }

  // This function returns the ID of the book.

  function getBookId() public view returns (uint) {

    return book.book\_id;

  }

  // This function returns the title, author, and ID of the book.

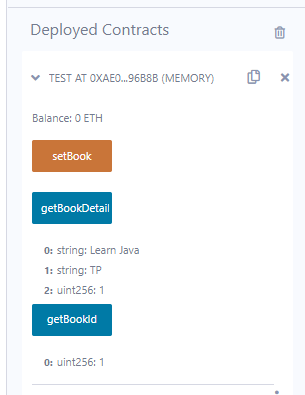
  function getBookDetail() public view returns (string memory, string memory, uint) {

    return (book.title, book.author, book.book\_id);

  }

}

**Output:**



**Aim: Type of Function (View, Pure)**

**Program:**

pragma solidity ^0.5.0;

contract Test {

  // global integer.

  int public x = 10;

  // state integer.

  int y = 90;

  function f1() public returns (int) {

    // We can read and update the global integer.

    x = 100;

    return x;

  }

  function f2() public view returns (int) {

    // We can only read the global integer.

    return x;

  }

  function f3() public pure returns (int) {

    // We cannot access or update the global integer in a pure function.

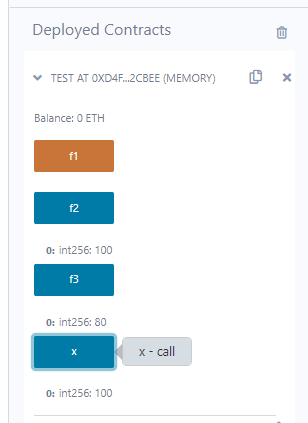
    int z = 80;

    return z;

  }

}

**Output:**



**Aim: Function Overloading**

**Program:**

pragma solidity ^0.5.0;

contract Test {

  function getSum(uint a, uint b) public pure returns (uint) {

    // sum of `a` and `b`.

    return a + b;

  }

  function getSum(uint a, uint b, uint c) public pure returns (uint) {

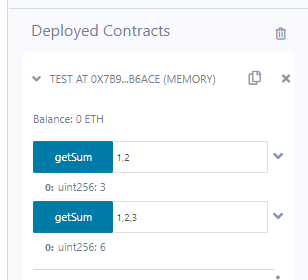
    // sum of `a`, `b`, and `c`.

    return a + b + c;

  }

}

**Output:**



**Aim: Mathematical Function**

**Program:**

pragma solidity ^0.5.0;

contract Test {

  function callAddMod() public pure returns (uint) {

    // Return the result of the `addmod` operation.

    return addmod(4, 5, 3);

  }

  function callMulMod() public pure returns (uint) {

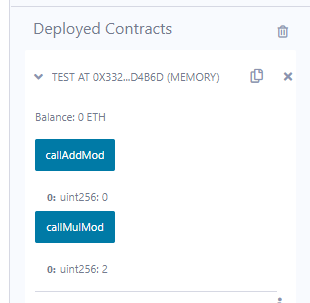
    // Return the result of the `mulmod` operation.

    return mulmod(4, 5, 3);

  }

}

**Output:**



**Aim: Cryptographic Function**

**Program:**

pragma solidity ^0.5.0;

contract Test {

  function callKeccak256() public pure returns (bytes32) {

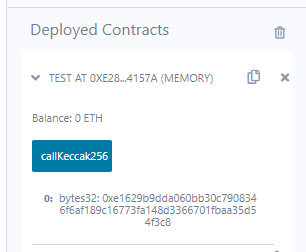
    // Return the result of the `keccak256` operation.

    return keccak256("ABC");

  }

}

**Output:**



**Practical No: 8**

**Aim:** Demonstrate the use of Bitcoin Core API.

**Program:**

from bitcoinlib.wallets import Wallet

w = Wallet.create('Wallet8')

key1 = w.get\_key()

print(key1.address)

w.scan()

print(w.info())

**Output:**

