Web3 Practical Prep

Practical 2: Demonstrate loop and conditional statements

// SPDX-License-Identifier: MIT

pragma solidity <0.9.0;

contract Loop {

function sumForLoop(uint limit) public pure returns (uint sum) {

sum = 0;

for(uint i = 0; i < limit; i++){

sum += i;

}

return sum;

}

function sumWhileLoop(uint limit) public pure returns (uint sum) {

sum = 0;

uint i = 0;

while(i < limit) {

sum += i;

i++;

}

return sum;

}

// Conditional Statements

function oddOrEven(uint number) public pure returns (string memory) {

if (number % 2 == 0) {

return "Even";

}else {

return "Non-even";

}

}

function oddOrEvenTernary(uint number) pure public returns (string memory) {

return number % 2 == 0 ? "Even" : "Not even";

}

function ageCheck(uint age) pure public returns (string memory) {

if (age < 18){

return "Minor";

} else if (age > 18 && age < 65){

return "Adult";

} else {

return "Senior";

}

}

}

Practical 3: Demonstrate arrays, constructor and inheritance

Array

// SPDX-License-Identifier: MIT

pragma solidity <0.9.0;

contract Prac3\_Array {

uint[5] fixedArray = [1, 2, 3, 4, 5];

uint[] dynamicArray;

function pushElement(uint number) public returns(uint[] memory) {

dynamicArray.push(number);

return dynamicArray;

}

function popElement() public returns (uint[] memory) {

dynamicArray.pop();

return dynamicArray;

}

function removeIndex(uint index) public returns(uint[] memory) {

require(dynamicArray.length > index, "Index out of bound");

delete dynamicArray[index];

return dynamicArray;

}

function getLength() public view returns (uint) {

return dynamicArray.length;

}

function getDynamicArray() public view returns (uint[] memory) {

return dynamicArray;

}

function getFixedArray() public view returns(uint[5] memory) {

return fixedArray;

}

}

Constructor and inheritance

// SPDX-License-Identifier: MIT

pragma solidity <0.9.0;

contract Person {

// Constructor to initialize name

constructor(string memory \_name) {

name = \_name;

}

string public name;

}

contract Employee is Person{

constructor(string memory \_name, uint \_salary) Person(\_name) {

salary = \_salary;

}

uint public salary;

}

contract Manager is Employee{

constructor(string memory \_name, uint \_salary, string memory \_department) Employee(\_name, \_salary) {

department = \_department;

}

string public department;

}

Practical 4: Strings

// SPDX-License-Identifier: MIT

pragma solidity <0.9.0;

contract Test {

string store;

event LogMessage(string message, string value);

function concatenateStrings(string memory s1, string memory s2) public pure returns (string memory) {

return string.concat(s1, s2);

}

function getStore() public view returns (string memory) {

return store;

}

function setStore(string memory inputStr) public returns (string memory) {

store = inputStr;

return store;

}

function logStore() public {

emit LogMessage("Value of store is: ", store);

}

}

Practical 5: Enum and structures

Enum

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Weekdays {

enum Weekday {

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

Weekday public choosenWeekDay;

function getChoosenWeekDay() public view returns(Weekday) {

return choosenWeekDay;

}

function setWeekDay(Weekday newWeekDay) public {

choosenWeekDay = newWeekDay;

}

function isWeekedn() public view returns(bool) {

return choosenWeekDay == Weekday.Sunday || choosenWeekDay == Weekday.Saturday ? true : false;

}

}

contract Library {

struct Book {

string name;

string writer;

uint id;

bool available;

}

Book book1 = Book("Introduction to python", "Sumita Arora", 1, true);

function getBookInfo() public view returns(string memory, string memory, uint, bool) {

return (book1.name, book1.writer, book1.id, book1.available);

}

function setAvailable(bool \_isAvailable) public {

book1.available = \_isAvailable;

}

function setNewId(uint \_newId) public {

book1.id = \_newId;

}

}

**Prac 6: Write the following programs for blockchain in python**

1. **A simple client class that generates the private and public key by using the built in python RSA algorithm and test it**
2. **A transaction class to send and receive money and test it**

import Crypto.Random as Random

from Crypto.PublicKey import RSA

from Crypto.Signature import PKCS1\_v1\_5

from Crypto.Hash import SHA

import binascii

import datetime

import collections

class Client:

def \_\_init\_\_(self) -> None:

random = Random.new().read()

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

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

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

@property

def identity(self):

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

class Transaction:

def \_\_init\_\_(self, sender, recipient, value) -> None:

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')

**Prac 7**

**import Crypto.Random as Random**

**from Crypto.PublicKey import RSA**

**from Crypto.Signature import PKCS1\_v1\_5**

**from Crypto.Hash import SHA**

**import binascii**

**import datetime**

**import collections**

**class Client:**

**def \_\_init\_\_(self) -> None:**

**random = Random.new().read**

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

**self.\_public\_key = self.\_private\_key.public\_key()**

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

**@property**

**def identity(self):**

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

**class Transaction:**

**def \_\_init\_\_(self, sender, recipient, value) -> None:**

**self.sender = sender**

**self.recipient = recipient**

**self.value = value**

**self.time = datetime.datetime.now()**

**def to\_dict(self):**

**if isinstance(self.sender, Client):**

**identity = self.sender.identity**

**else:**

**identity = self.sender**

**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')**

**class Block:**

**def \_\_init\_\_(self) -> None:**

**self.verified\_transactions = []**

**self.previous\_block\_hash = ""**

**self.Nonce = ""**

**last\_block\_hash = ""**

**Hrisabh = Client()**

**t0 = Transaction("Genisis", Hrisabh.identity, 3000)**

**block0 = Block()**

**block0.previous\_block\_hash = None**

**block0.Nonce = None**

**block0.verified\_transactions.append(t0)**

**digest = hash(block0)**

**last\_block\_hash = digest**

**TPCoins = []**

**def display\_transactions(transaction):**

**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 block 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\_transactions(transaction)**

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

**print("===========================")**

**TPCoins.append(block0)**

**dump\_blockchain(TPCoins)**

**Algo**

**Make a client**

**hrisabh = Client()**

**Make a transaction**

**t0 = Transaction('Genesis', hrisabh.identity, 3000)**

**Make a block**

**b0 = Block()**

**Set the necessary block attributes**

**b0.previous\_block\_hash = None**

**b0.Nonce = None**

**b0.verified\_transactions.append(t0)**

**Hash the block and store the hash into a variable**

**last\_block\_hash = hash(b0)**

**Make a array to store blocks**

**TPCoins = []**

**Append a block to the block list**

**TPCoins.append(b0)**

**Print the transaction details using dump\_blockchain function**