Practical 1

**AIM: WRITE A SOLIDITY PROGRAM FOR VARIABLES AND DECISION MAKING.**

**Source Code:**

**Compile Contract:**

**Deploy Contract:**

Practical 2

**AIM: WRITE A SOLIDITY PROGRAM TO DEMONSTRATE LOOP STATEMENT.**

**Source Code:**

**// SPDX-License-Identifier: GPL-3.0**

**pragma solidity <0.9.0;**

**contract Loop {**

**function loop() pure public returns(uint result) {**

**for (uint i = 0; i < 10; i++){**

**if (i == 3){**

**continue;**

**}**

**if ( i == 5) {**

**break;**

**}**

**}**

**uint j;**

**while( j < 10 ) {**

**j++;**

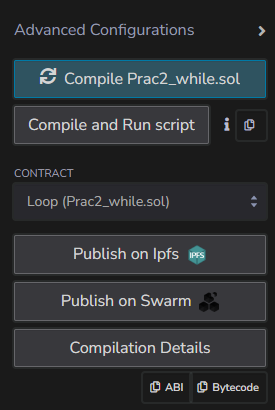
**}**

**return j;**

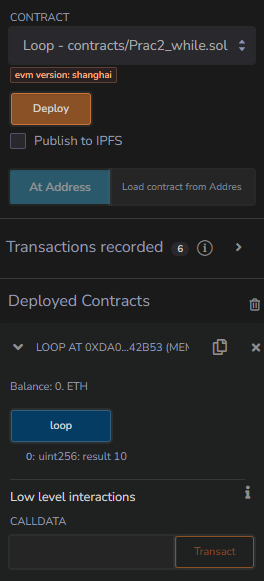
**}**

**}**

**Compile Contract:**

****

**Deploy Contract:**

****

**Source Code:**

**// SPDX-License-Identifier: GPL-3.0**

**pragma solidity <0.9.0;**

**contract IfElse {**

**function foo(uint \_x) public pure returns (uint) {**

**if (\_x < 10 ){**

**return 0;**

**} else if ( \_x < 20 ) {**

**return 1;**

**} else {**

**return 2;**

**}**

**}**

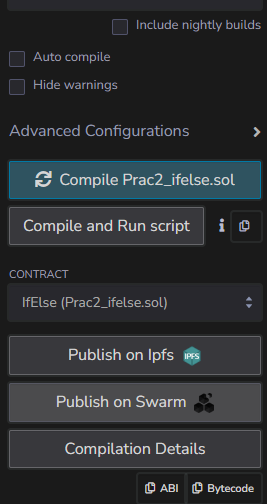
**function ternary(uint \_x ) public pure returns (uint) {**

**return \_x > 10 ? 1 : 2;**

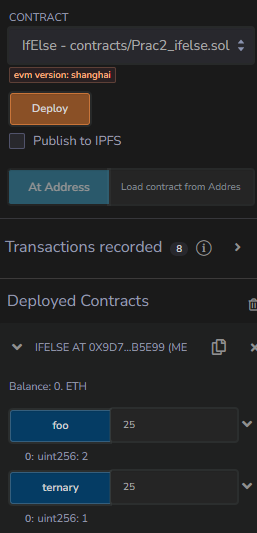
**}**

**}**

**Compile Contract:**

****

**Deploy Contract:**

****

**Source Code:**

**// SPDX-License-Identifier: GPL-3.0**

**pragma solidity <0.9.0;**

**contract ForContract {**

**uint storedData;**

**function set(uint x) public {**

**storedData = x;**

**for (uint i = 0; i != 5; i++) {**

**storedData = storedData + i;**

**}**

**}**

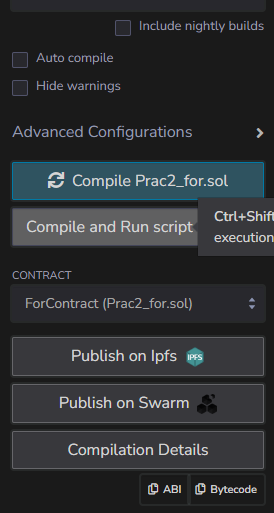
**function get() public view returns (uint) {**

**return storedData;**

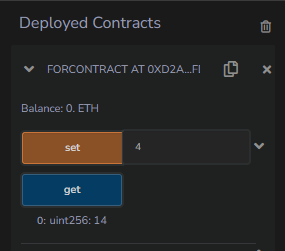
**}**

**}**

**Compile Contract:**

****

**Deploy Contract:**

****

**Source Code:**

**// SPDX-License-Identifier: GPL-3.0**

**pragma solidity <0.9.0;**

**contract Types {**

**uint i = 10;**

**bool even;**

**function decision\_making() public payable returns(bool) {**

**if (i % 2 == 0) {**

**even = true;**

**} else {**

**even = false;**

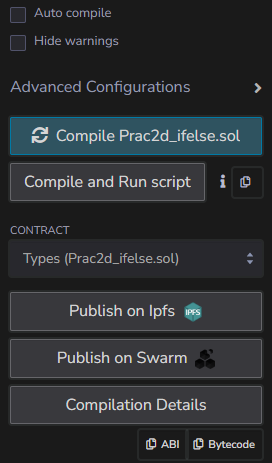
**}**

**return even;**

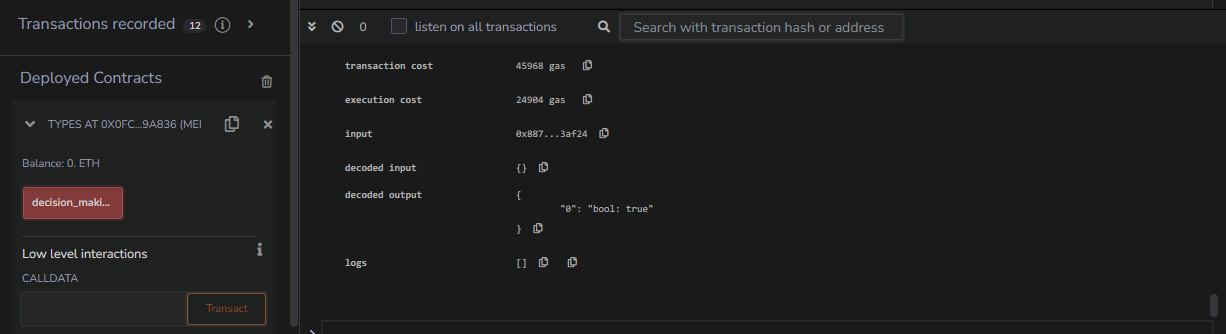
**}**

**}**

**Compile Contract:**

****

**Deploy Contract:**

****

**Source Code:**

**// SPDX-License-Identifier: GPL-3.0**

**pragma solidity <0.9.0;**

**contract ifelseif {**

**uint i = 12;**

**string result;**

**function decision\_making() public returns(string memory) {**

**if(i < 10){**

**result = "less than 10";**

**}**

**else if (i == 10){**

**result = "Equal to 10";**

**}**

**else {**

**result = "Greater than 10";**

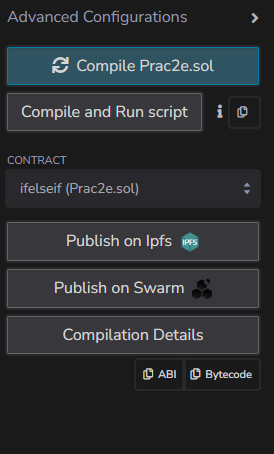
**}**

**return result;**

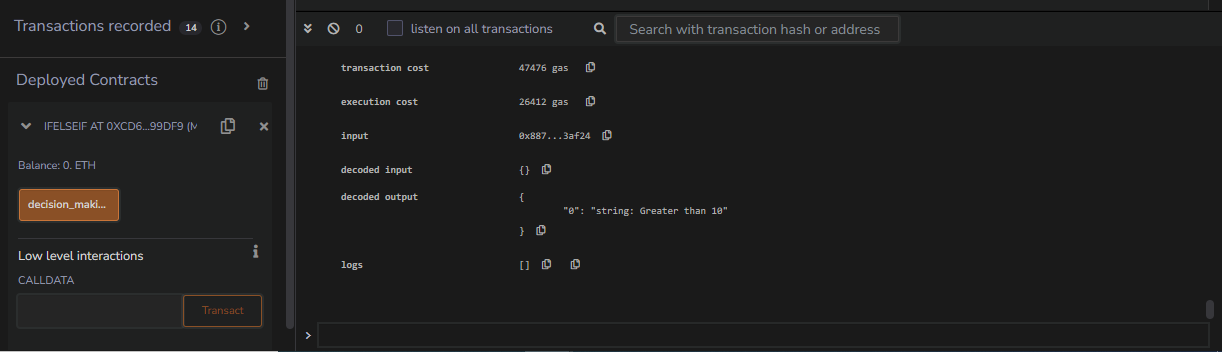
**}**

**}**

**Compile Contract:**

****

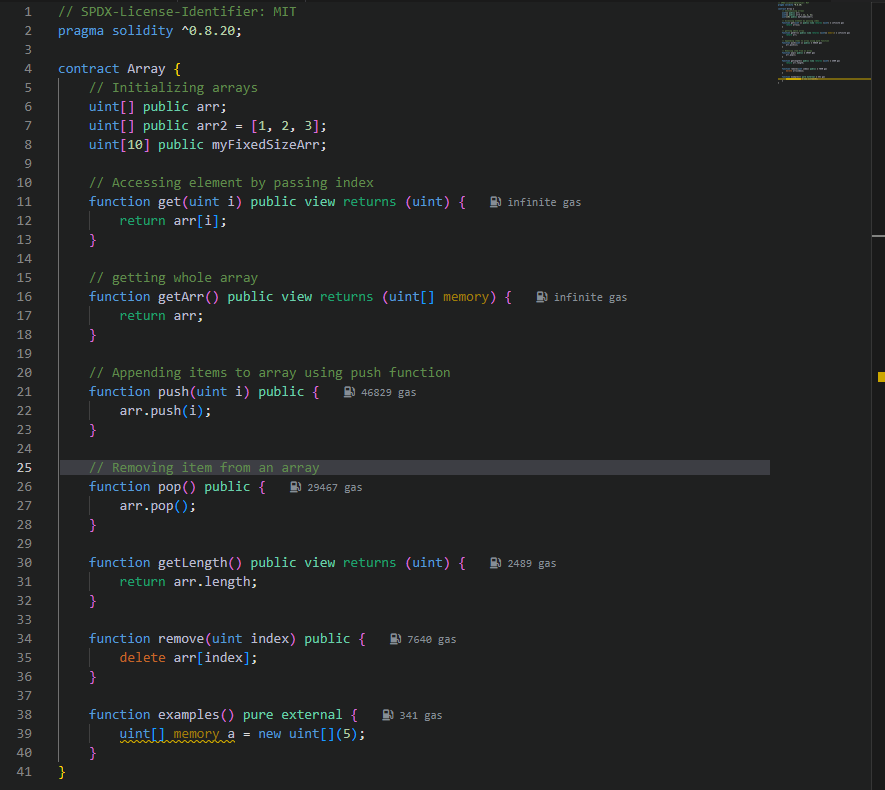
**Deploy Contract:**

****

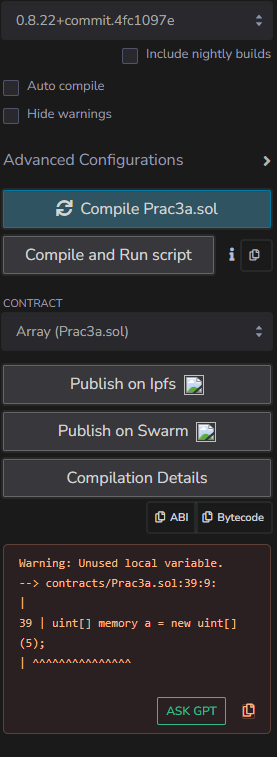
**Prac 3: Write a solidity program to demonstrate arrays and constructor**

**Array**

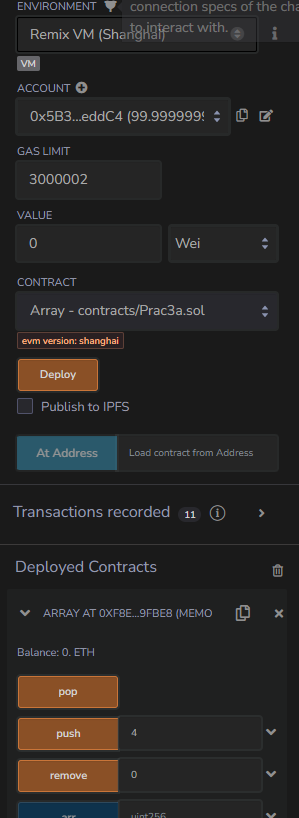
**Source Code:**

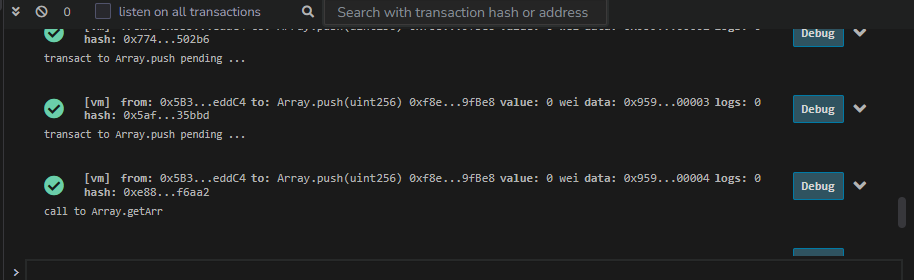
****

**Compile contract:**

****

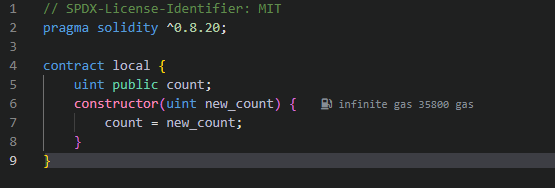
**Deploy contract:**

****

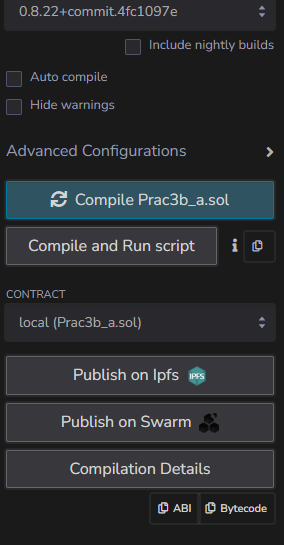
****

**Constructor Implmentation:**

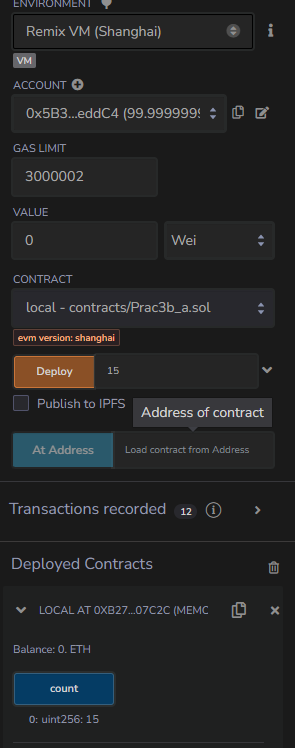
**Source Code:**

****

**Compile Contract:**

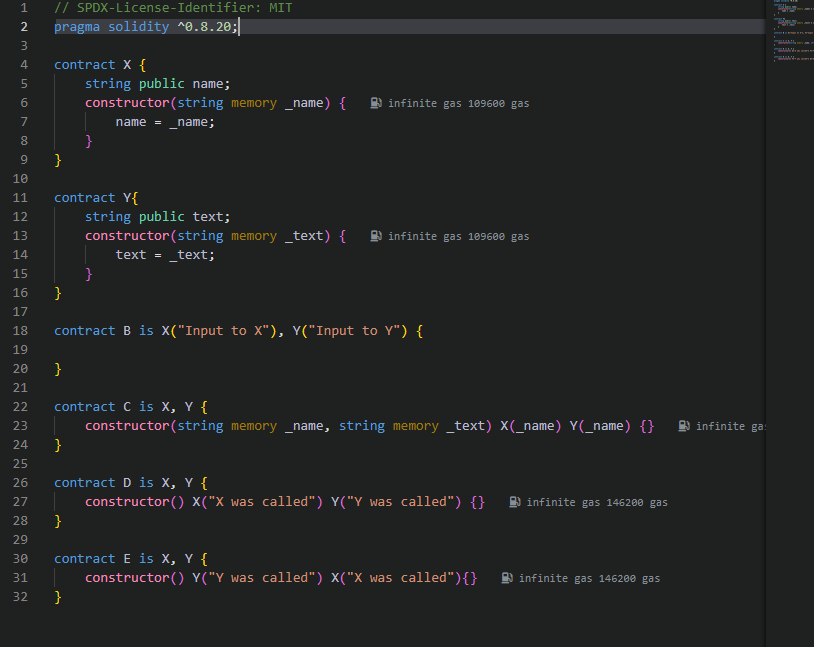
****

**Deploy Contract:**

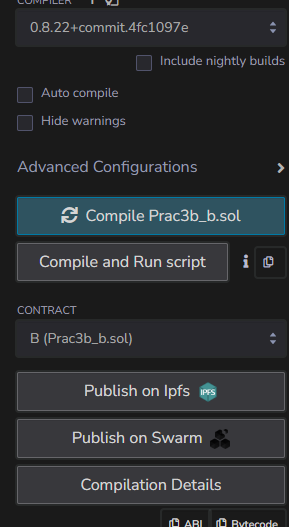
****

**Inheritance**

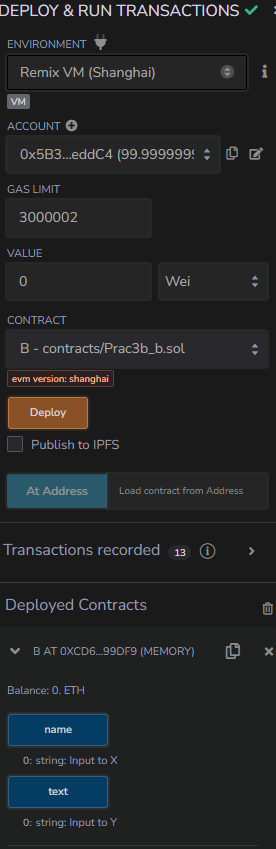
**Source Code:**

****

**Compile Contract:**

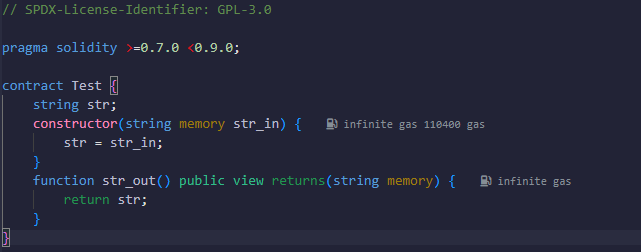
****

**Deploy Contract:**

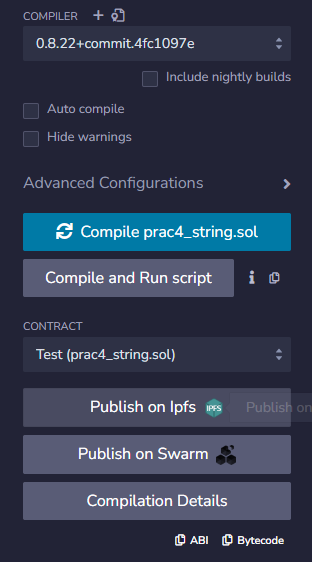
****

**Prac 4: Write a solidity program to demonstrate strings**

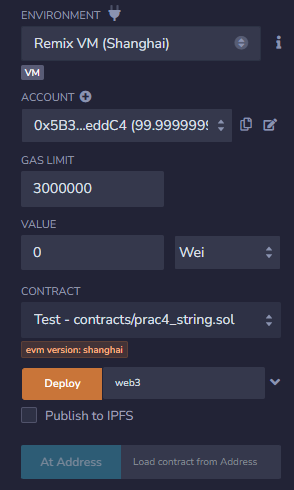
**Source Code:**

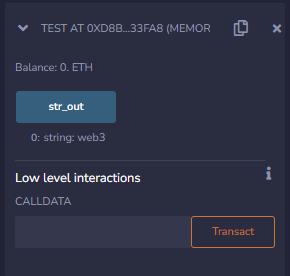
****

**Compile Contract:**

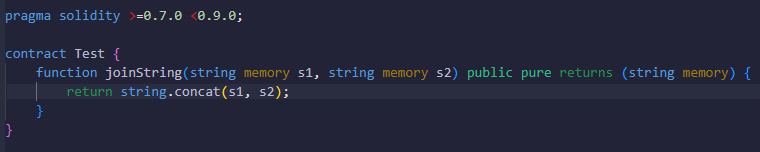
****

**Deploy Contract:**

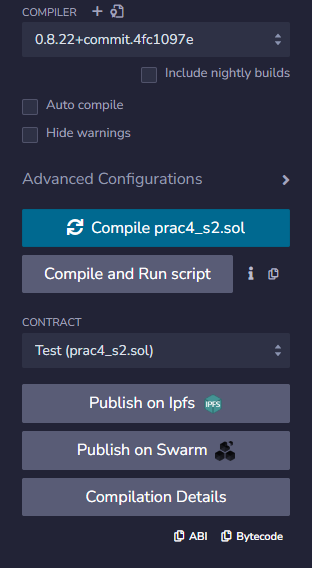
****

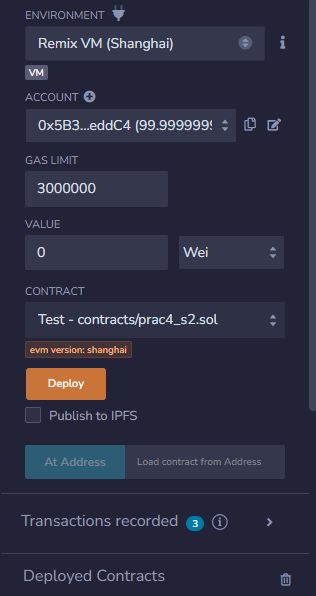
****

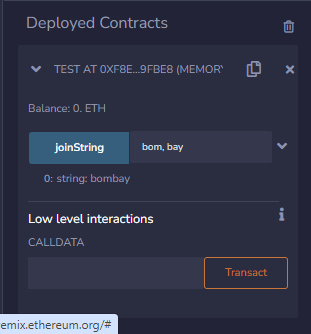
**Concatenate String:**

****

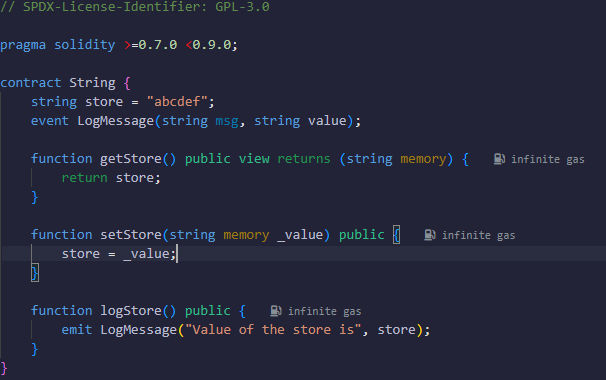
**Deploy Contract:**

****

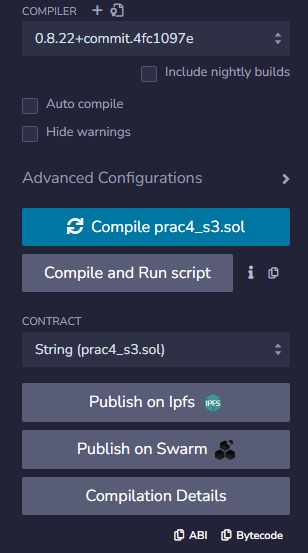
****

****

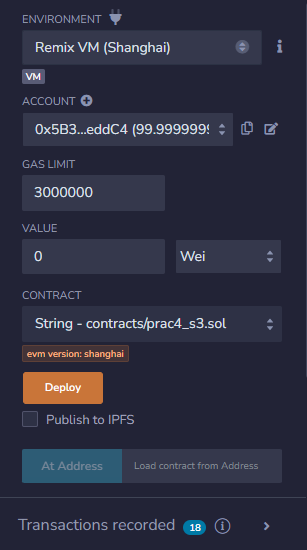
**Code:**

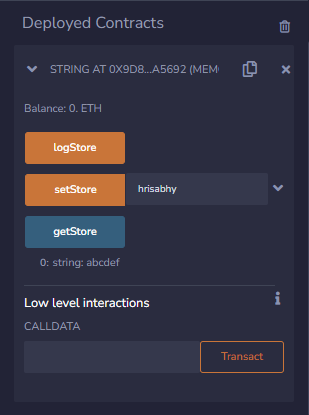
****

**Compile Contract:**

****

**Deploy Code:**

****

****

**Prac 5: Write a solidity program to demonstrate structure and enumerator**

**Code:**

**// SPDX-License-Identifier: GPL-3.0**

**pragma solidity <0.9.0;**

**contract Types{**

**enum week\_days{**

**Monday,**

**Tuesday,**

**Wednesday,**

**Thrusday,**

**Friday,**

**Saturday,**

**Sunday**

**}**

**week\_days week;**

**week\_days choice;**

**week\_days constant default\_value = week\_days.Sunday;**

**function set\_value() public {**

**choice = week\_days.Thrusday;**

**}**

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

**return choice;**

**}**

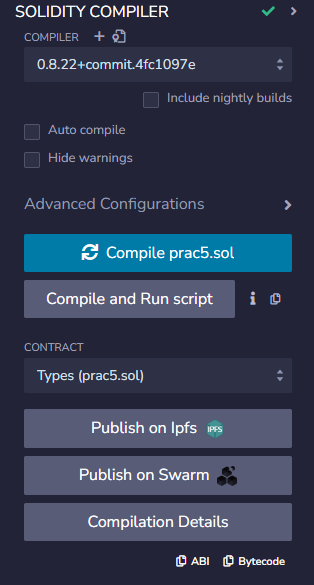
**function getdefaultvalu() public pure returns(week\_days) {**

**return default\_value;**

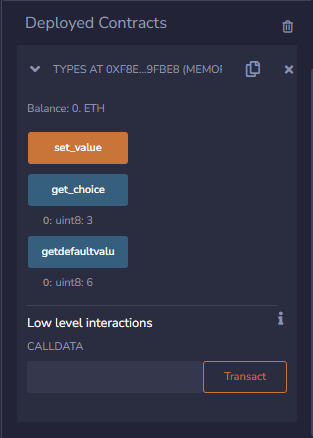
**}**

**}**

**Compile contract:**

****

**Deployed contract:**

****

**2nd Program**

**// SPDX-License-Identifier: GPL-3.0**

**pragma solidity <0.9.0;**

**contract test{**

**struct Book{**

**string name;**

**string writer;**

**uint id;**

**bool available;**

**}**

**Book book1;**

**Book book2 = Book("Building Etherium DApps", "Roberto Infante", 2, false);**

**function set\_book\_detail() public{**

**book1 = Book("Introducing Etherium & Solidity", "Chris Damen", 1, true);**

**}**

**function book\_info() public view returns(string memory, string memory, uint, bool){**

**return(book2.name, book2.writer, book2.id, book2.available);**

**}**

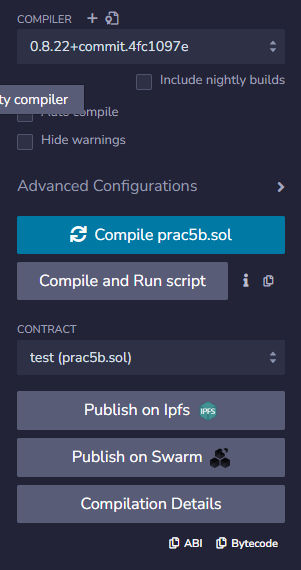
**function get\_details() public view returns(string memory, uint) {**

**return (book1.name, book1.id);**

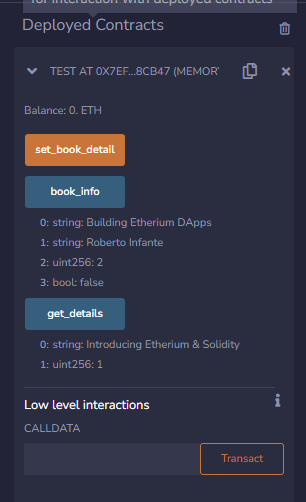
**}**

**}**

**Compile Code:**

****

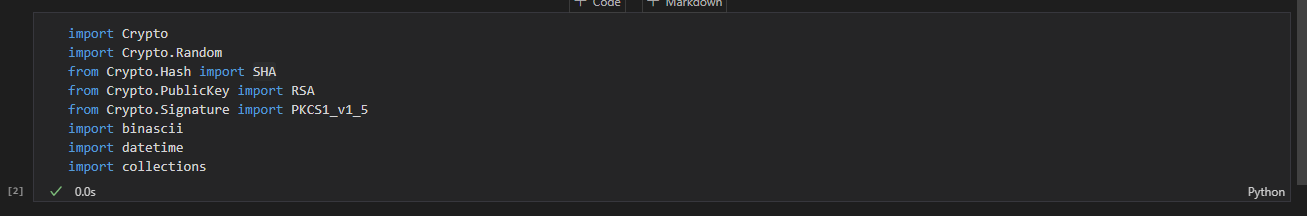
**Deploy Code:**

****

**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.**

**Code:**

****

**class Client:**

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

**random = Crypto.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, recipent, value) -> None:**

**self.sender = sender**

**self.recipent = recipent**

**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,**

**"recipent": self.recipent,**

**"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**

**Code:**

**import Crypto**

**import Crypto.Random**

**from Crypto.Hash import SHA**

**from Crypto.PublicKey import RSA**

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

**import binascii**

**import datetime**

**import collections**

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

**dict = transaction.to\_dict()**

**print("sender" + dict['sender'])**

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

**print("recipent" + dict['recipent'])**

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

**print("value:" + str(dict['value']))**

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

**print("time:" + str(dict['time']))**

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

**class Client:**

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

**random = Crypto.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, recipent, value) -> None:**

**self.sender = sender**

**self.recipent = recipent**

**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,**

**"recipent": self.recipent,**

**"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')**

**transactions = []**

**Dinesh = Client()**

**Ramesh = Client()**

**Seema = Client()**

**Vijay = Client()**

**t1 = Transaction(**

**Dinesh,**

**Ramesh.identity,**

**15.0**

**)**

**t1.sign\_transaction()**

**transactions.append(t1)**

**t2 = Transaction(Dinesh, Seema.identity, 6.0)**

**t2.sign\_transaction()**

**t3 = Transaction(Ramesh, Vijay.identity, 2.0)**

**t3.sign\_transaction()**

**transactions.append(t3)**

**t4 = Transaction(**

**Seema,**

**Ramesh.identity,**

**4.0**

**)**

**t4.sign\_transaction()**

**transactions.append(t4)**

**t5 = Transaction(**

**Vijay,**

**Seema.identity,**

**7.0**

**)**

**t5.sign\_transaction()**

**transactions.append(t5)**

**t6 = Transaction(**

**Ramesh,**

**Seema.identity,**

**3.0**

**)**

**t6.sign\_transaction()**

**transactions.append(t6)**

**t7 = Transaction(**

**Seema,**

**Dinesh.identity,**

**8.0**

**)**

**t8 = Transaction(**

**Seema,**

**Ramesh.identity,**

**1.0**

**)**

**t8.sign\_transaction()**

**transactions.append(t8)**

**t9 = Transaction(**

**Vijay,**

**Dinesh.identity,**

**5.0**

**)**

**t9.sign\_transaction()**

**transactions.append(t9)**

**t10 = Transaction(**

**Vijay,**

**Ramesh.identity,**

**3.0**

**)**

**t10.sign\_transaction()**

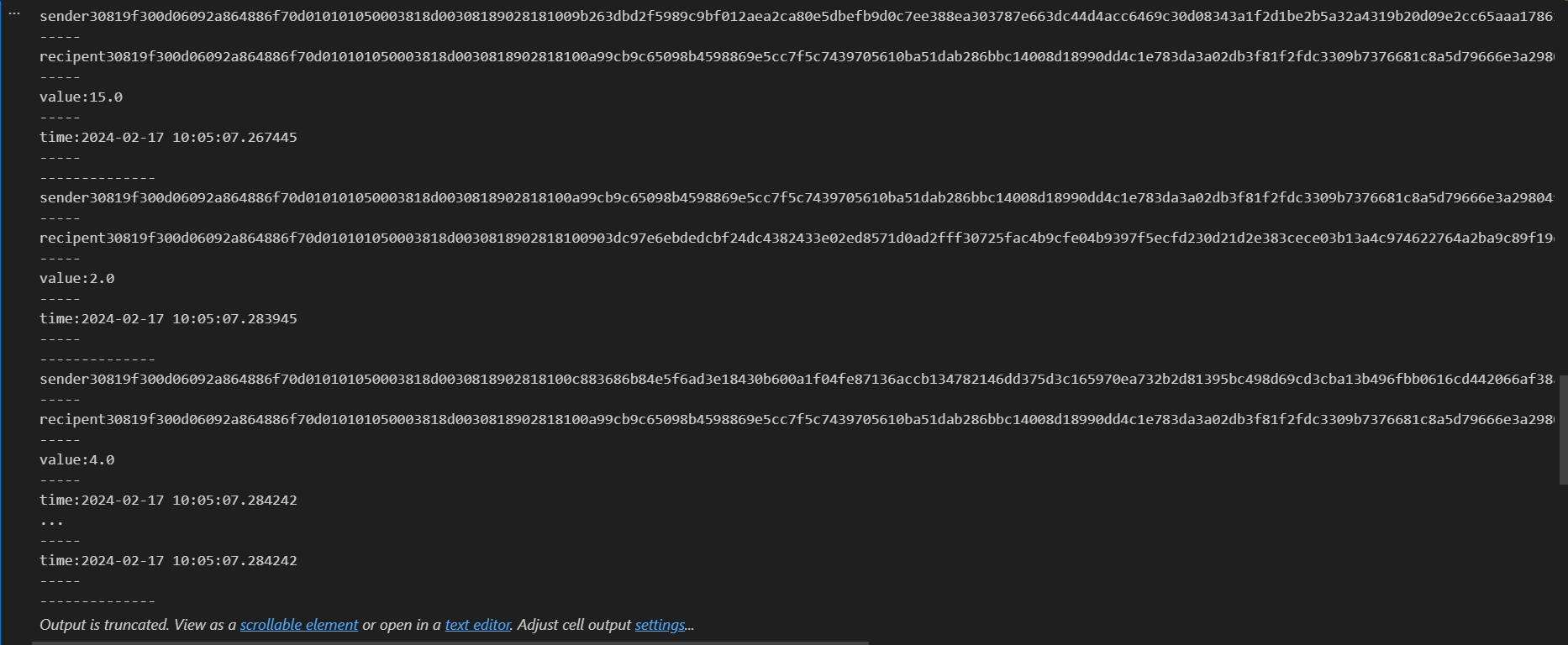
**transactions.append(t10)**

**for transaction in transactions:**

**display\_transaction (transaction)**

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

**Output:**

****

**class Block:**

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

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

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

**self.Nonce = ""**

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

**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 = []**

**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)**

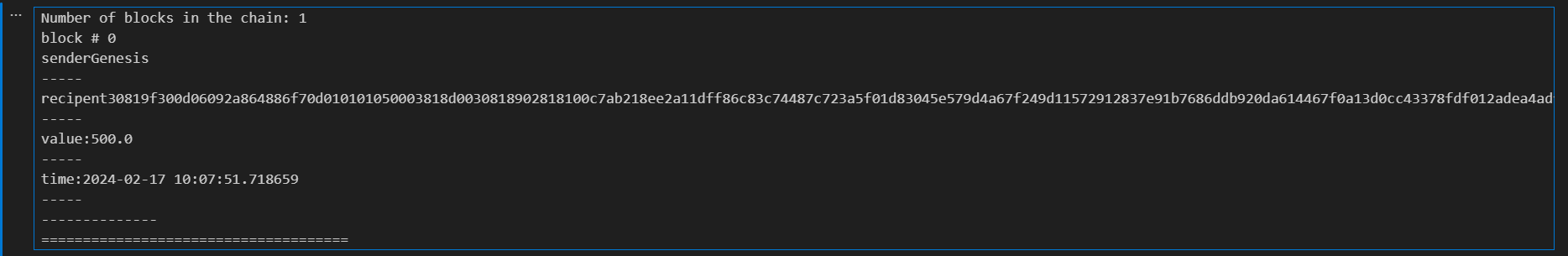
**print ('--------------')**

**print ('=====================================')**

**TPCoins.append (block0)**

**dump\_blockchain(TPCoins)**

**Output:**

****