|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Experiment name** | **Page No.** |
| 1(A) | A simple client class that generates the private public keys by using the built-in python RSA algorithm and test it. | 2 |
| 1(B) | A transaction class to send and receive money and test it. | 4 |
| 2(A) | Create multiple transaction and display them. | 6 |
| 2(B) | Create blockchain, a genesis block and execute it. | 9 |
| 3(A) | Create a mining function and test it. | 12 |
| 3(B) | Add blocks to the minor and dump the blockchain. | 13 |
| 4 | Write a solidity program for variable, operators, loops, decision making and string. | 15 |
| 5 | Write a solidity program for array, Enums, structure and mapping. | 49 |
| 6 | Write a solidity program for function, function modifies, view function, pure function and fallback function. | 58 |
| 7 | Write a solidity program for mathematical and cryptographic function. | 63 |
| 8 | Write a solidity program for contracts, inheritance, constructors, abstract constructs, libraries, assembly, events and error handling. | 66 |

**AIM: WRITE A SOLIDITY PROGRAM FOR VARIABLES, OPERATORS, LOOPS, DECISION MAKING AND STRING.**

**A)Variables:**

supports three types of variables.

**State Variables** − Variables whose values are permanently stored in a contract storage.

**Local Variables** − Variables whose values are present till function is executing.

**Global Variables** − Special variables exists in the global namespace used to get information about the blockchain.i.e. blockhash(uint blockNumber) returns (bytes32), block.coinbase (address payable), block.difficulty (uint)…..and many more

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData; // State variable

constructor() public {

storedData = 10;

}

function getResult() public view returns(uint){

uint a = 1; // local variable

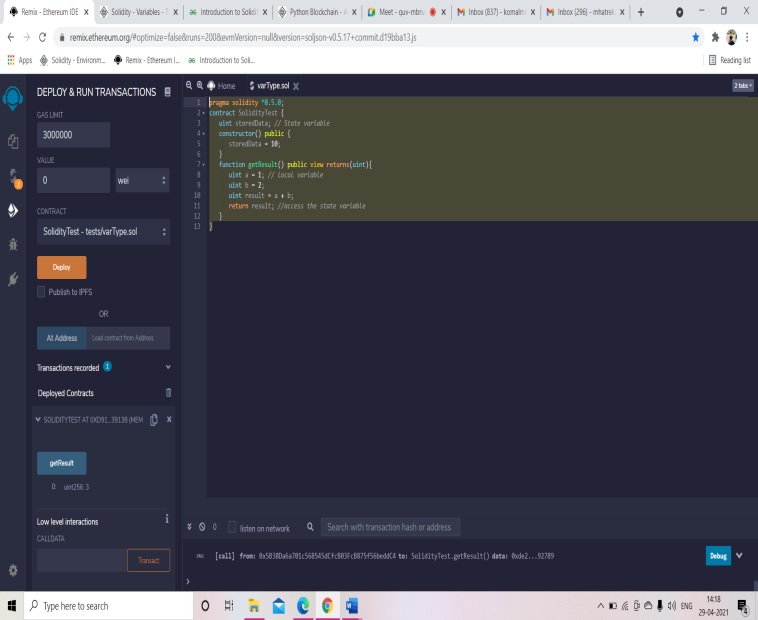
uint b = 2;

uint result = a + b;

return result; //access the state variable

}

}



**1.State Variable:**

// Solidity program to

// demonstrate state

// variables

pragma solidity ^0.5.0;

// Creating a contract

contract Solidity\_var\_Test {

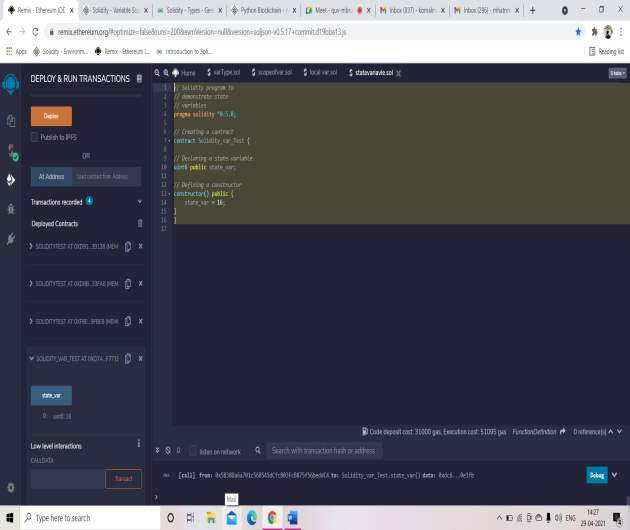
// Declaring a state variable

uint8 public state\_var;

// Defining a constructor

constructor() public {

state\_var = 16;}}



**2.Local Variable:**

// Solidity program to demonstrate

// local variables

pragma solidity ^0.5.0;

// Creating a contract

contract Solidity\_var\_Test {

// Defining function to show the declaration and

// scope of local variables

function getResult() public view returns(uint){

// Initializing local variables

uint local\_var1 = 1;

uint local\_var2 = 2;

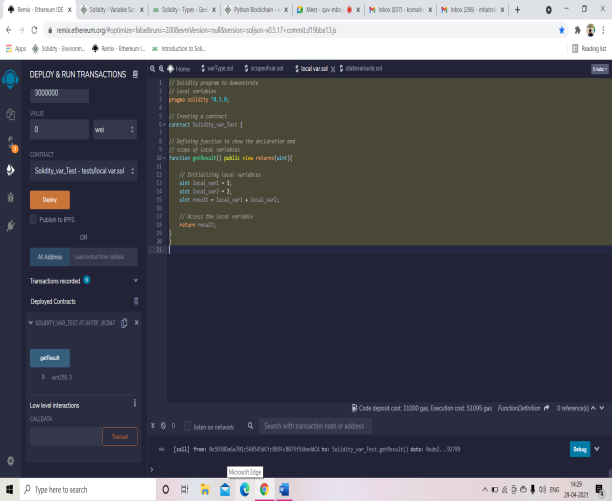
uint result = local\_var1 + local\_var2;

// Access the local variable

return result;

}

}



**3.Global variable:**

// Solidity program to

// show Global variables

pragma solidity ^0.5.0;

// Creating a contract

contract Test {

// Defining a variable

address public admin;

// Creating a constructor to

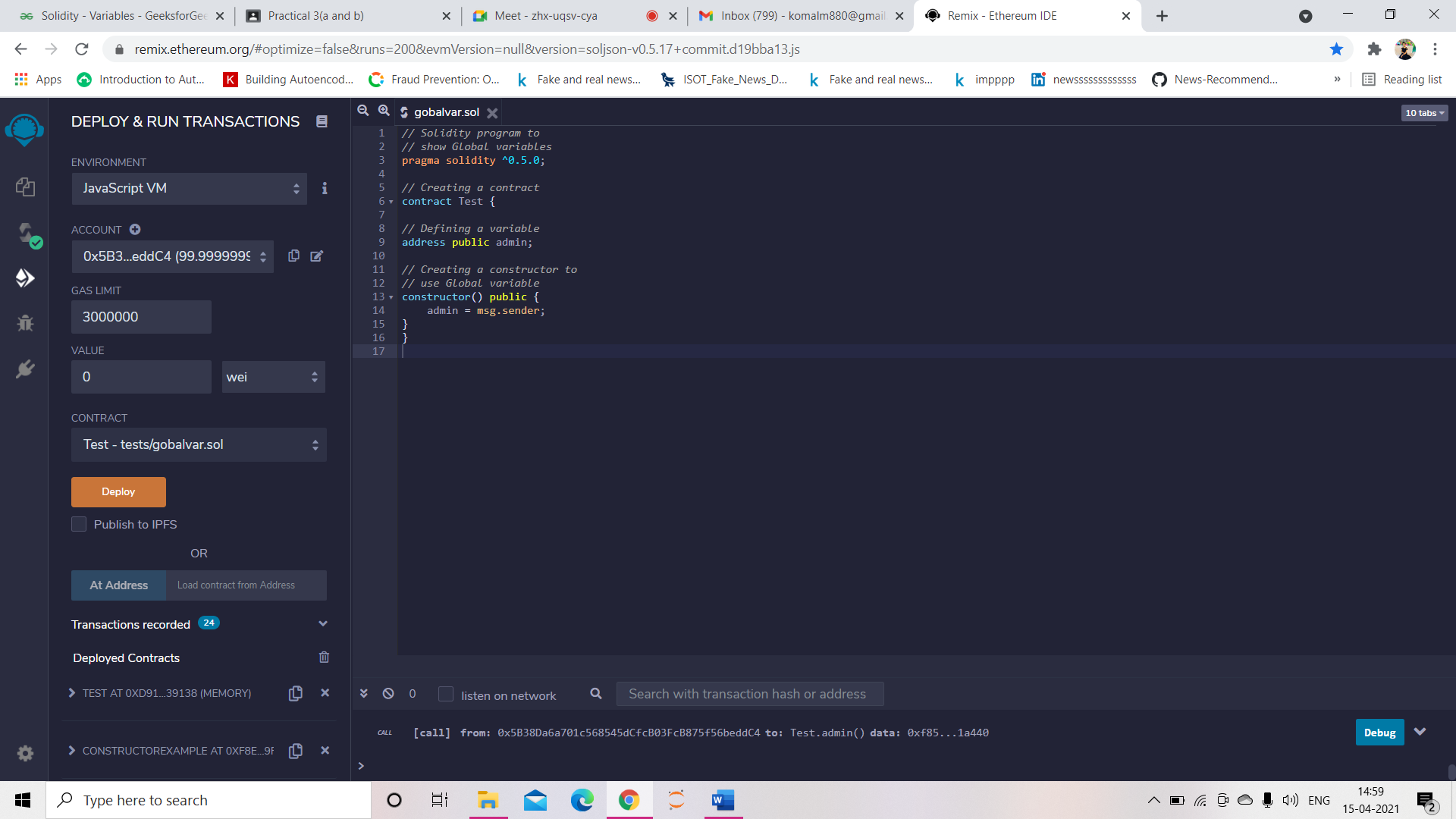
// use Global variable

constructor() public {

admin = msg.sender;

}

}





Scope of local variables is limited to function in which they are defined but State variables can have three types of scopes.

**Public** − Public state variables can be accessed internally as well as via messages. For a public state variable, an automatic getter function is generated.

**Internal** − Internal state variables can be accessed only internally from the current contract or contract deriving from it without using this.

**Private** − Private state variables can be accessed only internally from the current contract they are defined not in the derived contract from it.

**B)Operators**

Solidity supports the following types of operators.

Arithmetic Operators

Comparison Operators

Logical (or Relational) Operators

Assignment Operators

Conditional (or ternary) Operators

**1. Arithematic Operator**

// Solidity contract to demonstrate

// Arithematic Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Initializing variables

uint16 public a = 20;

uint16 public b = 10;

// Initializing a variable

// with sum

uint public sum = a + b;

// Initializing a variable

// with the difference

uint public diff = a - b;

// Initializing a variable

// with product

uint public mul = a \* b;

// Initializing a variable

// with quotient

uint public div = a / b;

// Initializing a variable

// with modulus

uint public mod = a % b;

// Initializing a variable

// decrement value

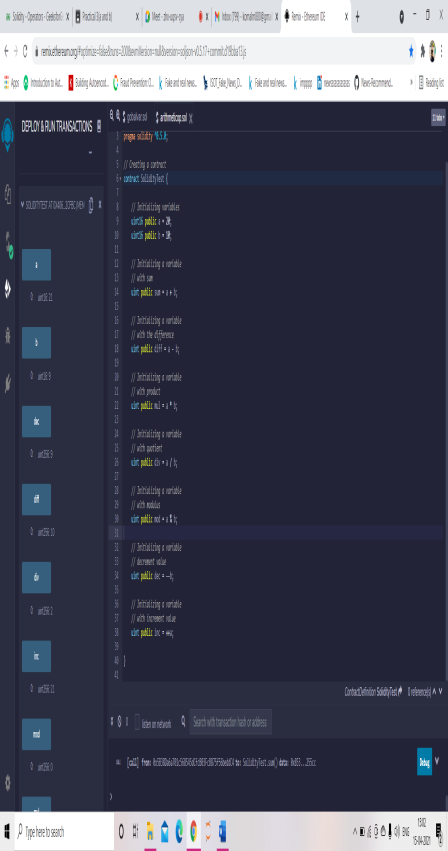
uint public dec = --b;

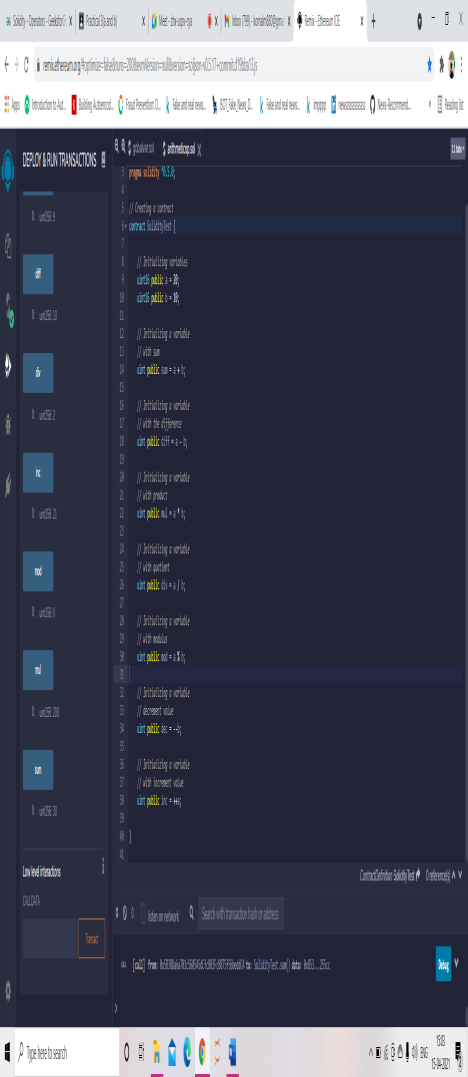
// Initializing a variable

// with increment value

uint public inc = ++a;

}





**2.Reltional Operator**

// Solidity program to demonstrate

// Relational Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Declaring variables

uint16 public a = 20;

uint16 public b = 10;

// Initializing a variable

// with bool equal result

bool public eq = a == b;

// Initializing a variable

// with bool not equal result

bool public noteq = a != b;

// Initializing a variable

// with bool greater than result

bool public gtr = a > b;

// Initializing a variable

// with bool less than result

bool public les = a < b;

// Initializing a variable

// with bool greater than equal to result

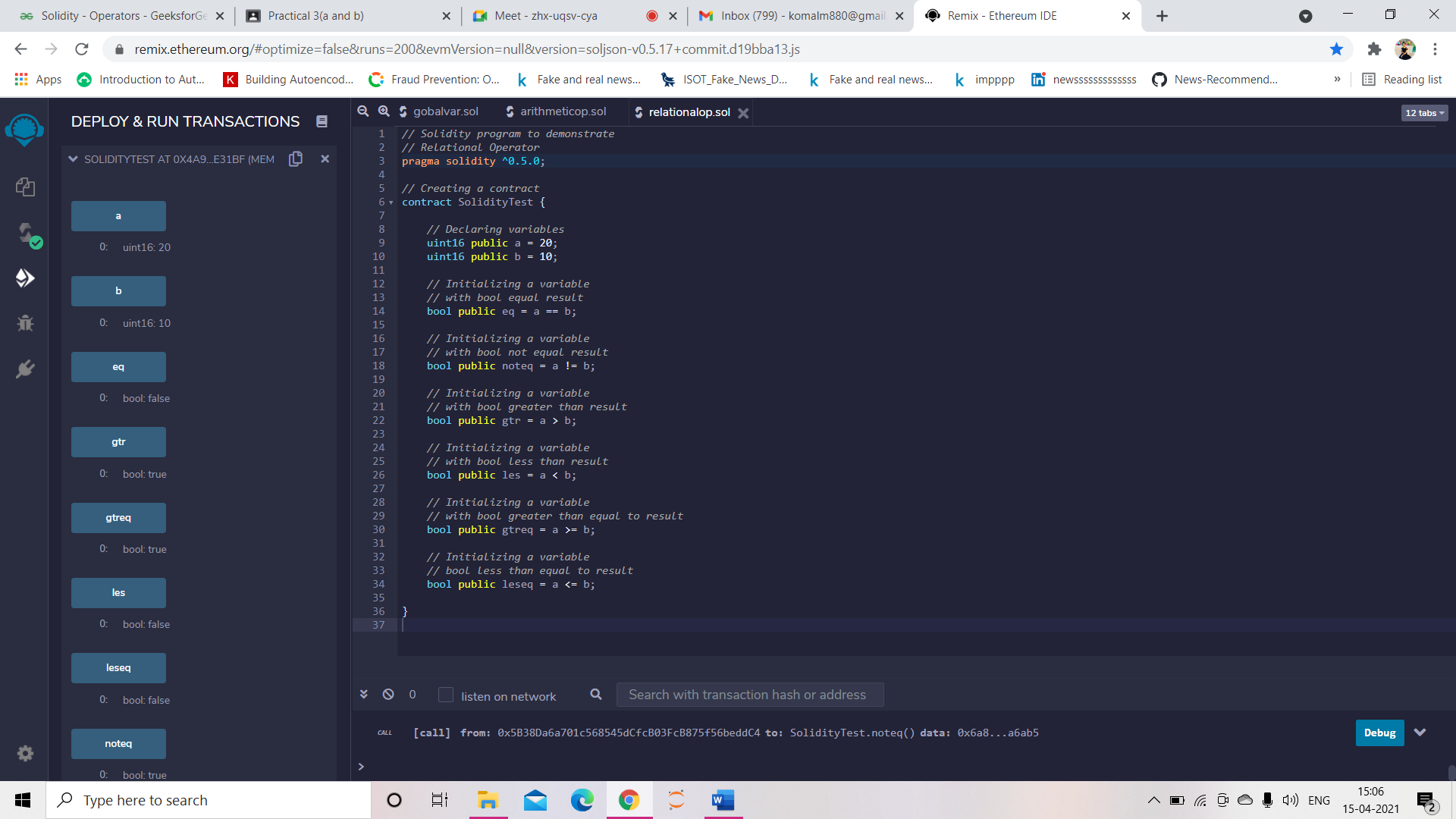
bool public gtreq = a >= b;

// Initializing a variable

// bool less than equal to result

bool public leseq = a <= b;

}



**3.Logical Operators**

// Solidity program to demonstrate

// Logical Operators

pragma solidity ^0.5.0;

// Creating a contract

contract logicalOperator{

// Defining function to demonstrate

// Logical operator

function Logic(

bool a, bool b) public view returns(

bool, bool, bool){

// Logical AND operator

bool and = a&&b;

// Logical OR operator

bool or = a||b;

// Logical NOT operator

bool not = !a;

return (and, or, not);

}

}



**4.Bitwise Operators**

// Solidity program to demonstrate

// Bitwise Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Declaring variables

uint16 public a = 20;

uint16 public b = 10;

// Initializing a variable

// to '&' value

uint16 public and = a & b;

// Initializing a variable

// to '|' value

uint16 public or = a | b;

// Initializing a variable

// to '^' value

uint16 public xor = a ^ b;

// Initializing a variable

// to '<<' value

uint16 public leftshift = a << b;

// Initializing a variable

// to '>>' value

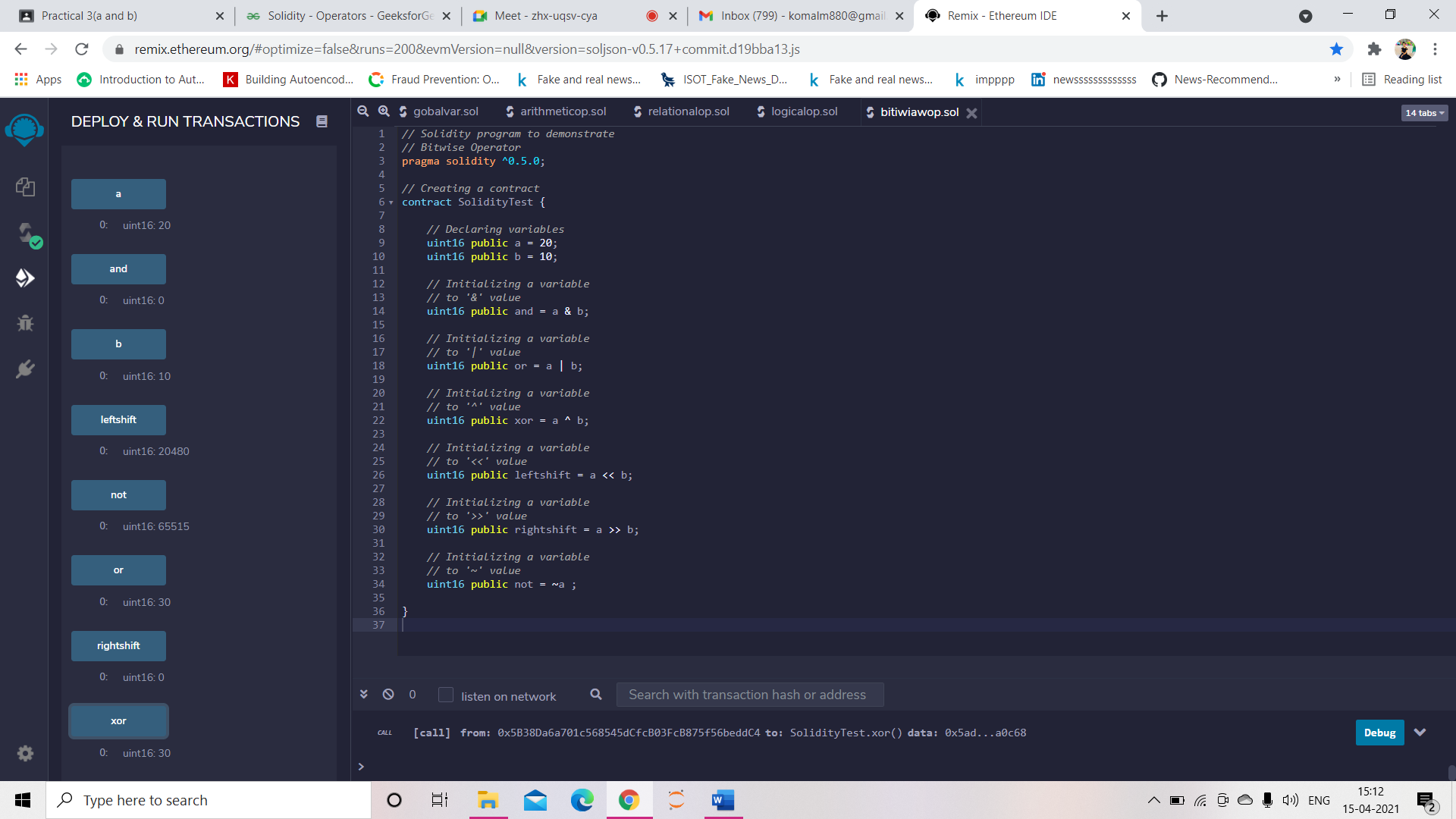
uint16 public rightshift = a >> b;

// Initializing a variable

// to '~' value

uint16 public not = ~a ;

}



**5.Assignment Operator**

// Solidity program to demonstrate

// Assignment Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Declaring variables

uint16 public assignment = 20;

uint public assignment\_add = 50;

uint public assign\_sub = 50;

uint public assign\_mul = 10;

uint public assign\_div = 50;

uint public assign\_mod = 32;

// Defining function to

// demonstrate Assignment Operator

function getResult() public{

assignment\_add += 10;

assign\_sub -= 20;

assign\_mul \*= 10;

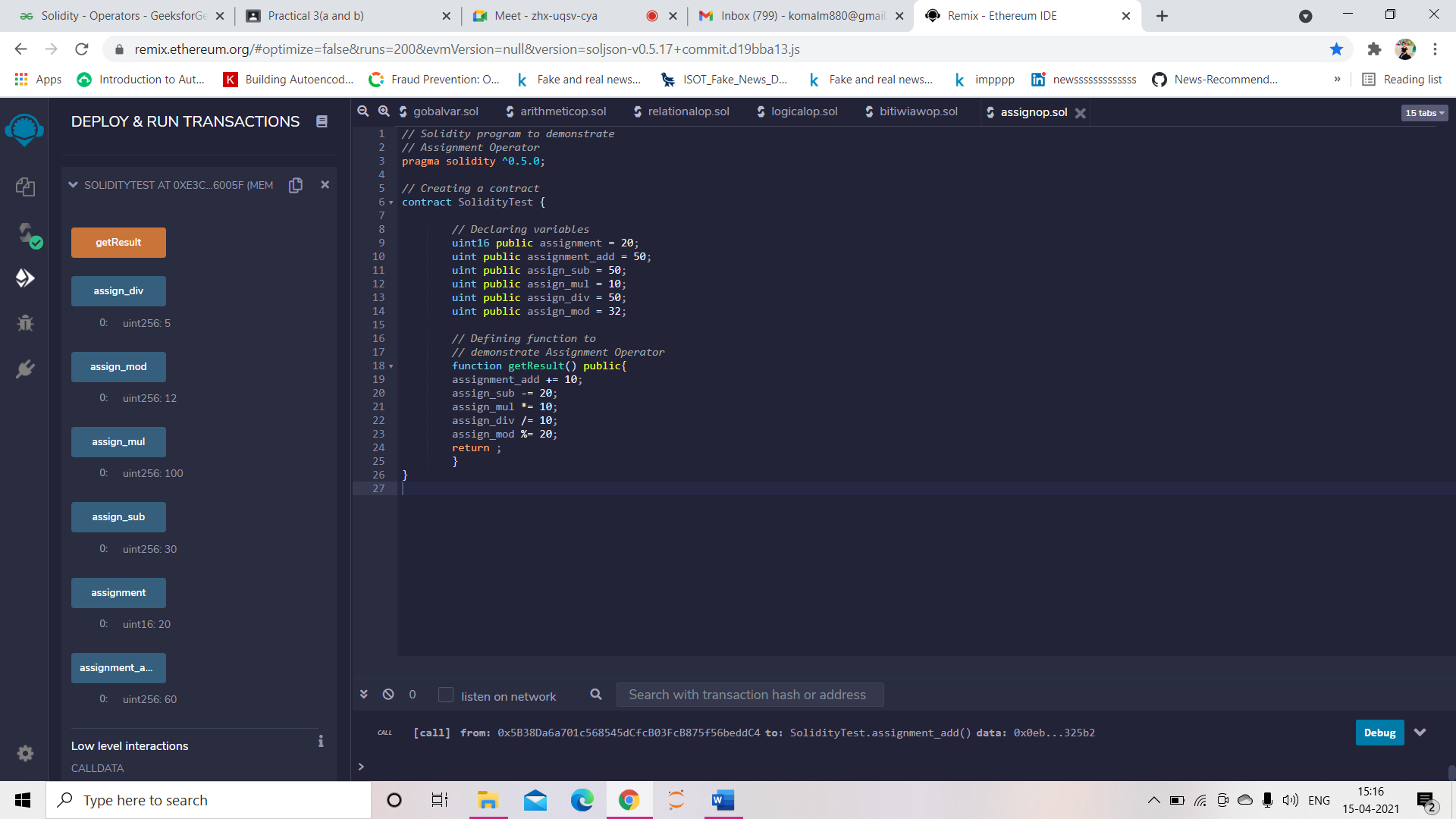
assign\_div /= 10;

assign\_mod %= 20;

return ;

}

}



6**.Conditional Operators**

// Solidity program to demonstrate

// Conditional Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest{

// Defining function to demonstrate

// conditional operator

function sub(

uint a, uint b) public view returns(

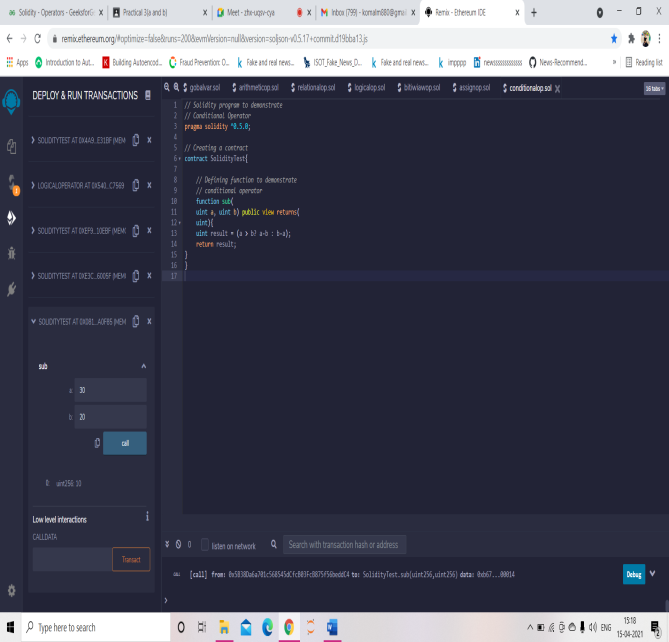
uint){

uint result = (a > b? a-b : b-a);

return result;

}

}



**C)Loops:**

1.While loop: The most basic loop in Solidity is the **while** loop which would be discussed in this chapter. The purpose of a **while** loop is to execute a statement or code block repeatedly as long as an **expression** is true. Once the expression becomes **false,** the loop terminates.

2.do-while loop: The **do...while** loop is similar to the **while** loop except that the condition check happens at the end of the loop. This means that the loop will always be executed at least once, even if the condition is **false**.

3.for loop: The **for** loop is the most compact form of looping. It includes the following three important parts −

The **loop initialization** where we initialize our counter to a starting value. The initialization statement is executed before the loop begins.

The **test statement** which will test if a given condition is true or not. If the condition is true, then the code given inside the loop will be executed, otherwise the control will come out of the loop.

The **iteration statement** where you can increase or decrease your counter.

4.loop control: Solidity provides full control to handle loops and switch statements. There may be a situation when you need to come out of a loop without reaching its bottom. There may also be a situation when you want to skip a part of your code block and start the next iteration of the loop.To handle all such situations, Solidity provides **break** and **continue** statements. These statements are used to immediately come out of any loop or to start the next iteration of any loop respectively.

**1.While Loop**

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public{

storedData = 10;

}

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

uint a = 10;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (j != 0) {

len++;

j /= 10;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) { // while loop

bstr[k--] = byte(uint8(48 + \_i % 10));

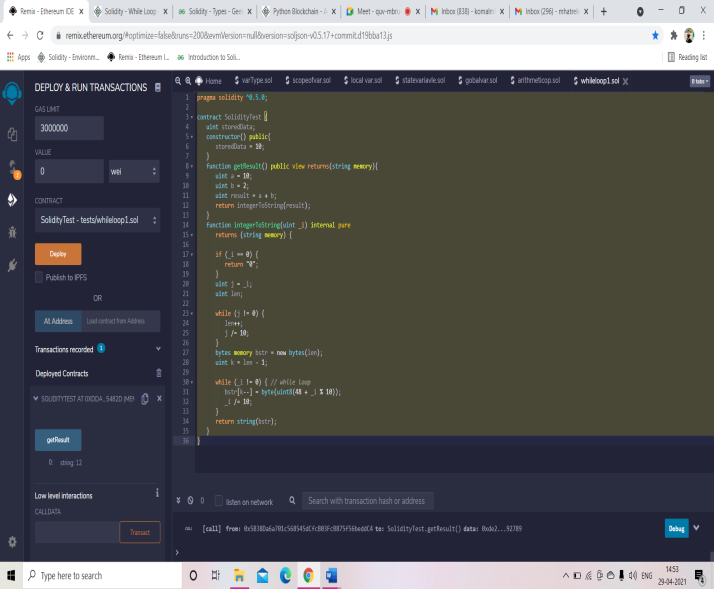
\_i /= 10;

}

return string(bstr);

}

}



**2.Do-while loop:**

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public{

storedData = 10;

}

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

uint a = 10;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (j != 0) {

len++;

j /= 10;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

do { // do while loop

bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

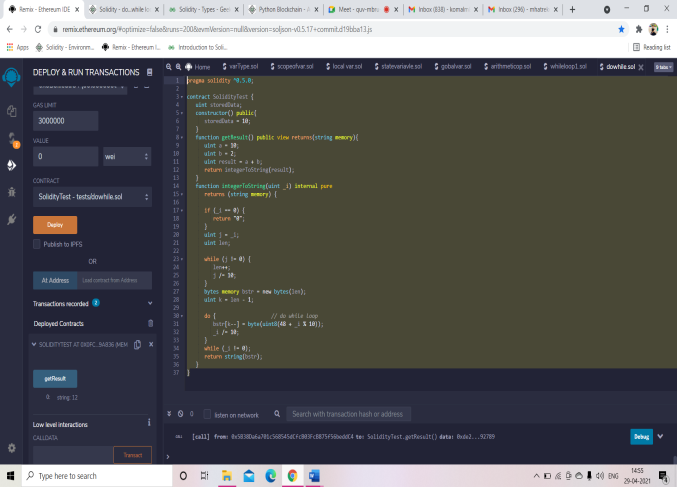
}

while (\_i != 0);

return string(bstr);

}

}



**3.For Loop:**

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public{

storedData = 10;

}

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

uint a = 10;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j=0;

uint len;

for (j = \_i; j != 0; j /= 10) { //for loop example

len++;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

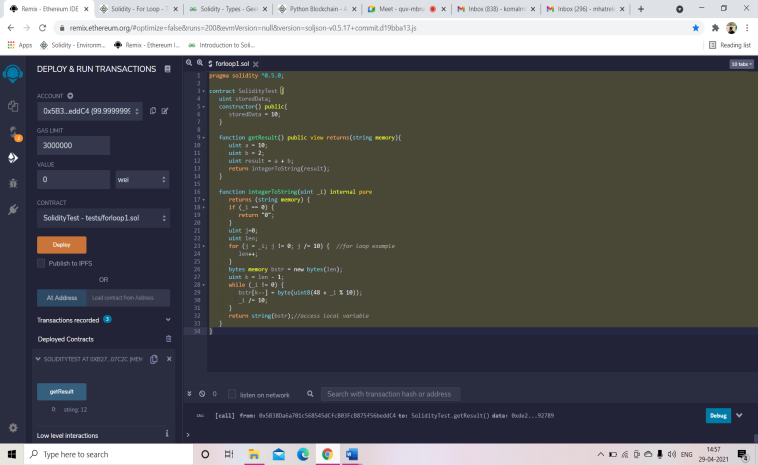
bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);//access local variable

}}



**4.loop Control: (Break statement)**

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public{

storedData = 10;

}

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

uint a = 1;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (true) {

len++;

j /= 10;

if(j==0){

break; //using break statement

}

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

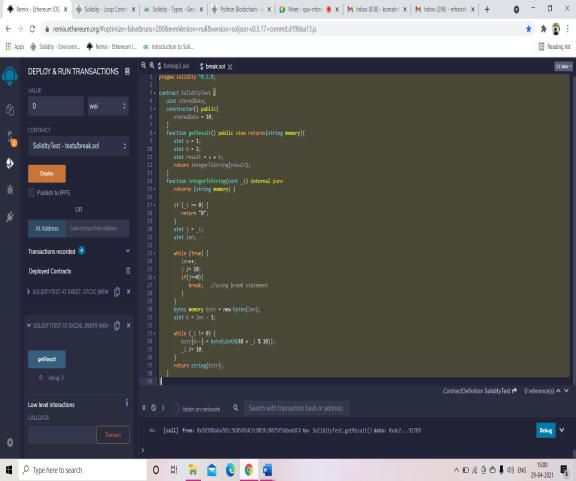
\_i /= 10;

}

return string(bstr);

}

}



**(continue statement)**

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public{

storedData = 10;

}

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

uint n = 1;

uint sum = 0;

while( n < 10){

n++;

if(n == 5){

continue; // skip n in sum when it is 5.

}

sum = sum + n;

}

return integerToString(sum);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (true) {

len++;

j /= 10;

if(j==0){

break; //using break statement

}

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);

}

}



**D)Decision Making:**

While writing a program, there may be a situation when you need to adopt one out of a given set of paths. In such cases, you need to use conditional statements that allow your program to make correct decisions and perform right actions.Solidity supports conditional statements which are used to perform different actions based on different conditions. Here we will explain the **if..else** statement.

1.if statement: The **if** statement is the fundamental control statement that allows Solidity to make decisions and execute statements conditionally.

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public {

storedData = 10;

}

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

uint a = 1;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) { // if statement

return "0";

}

uint j = \_i;

uint len;

while (j != 0) {

len++;

j /= 10;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

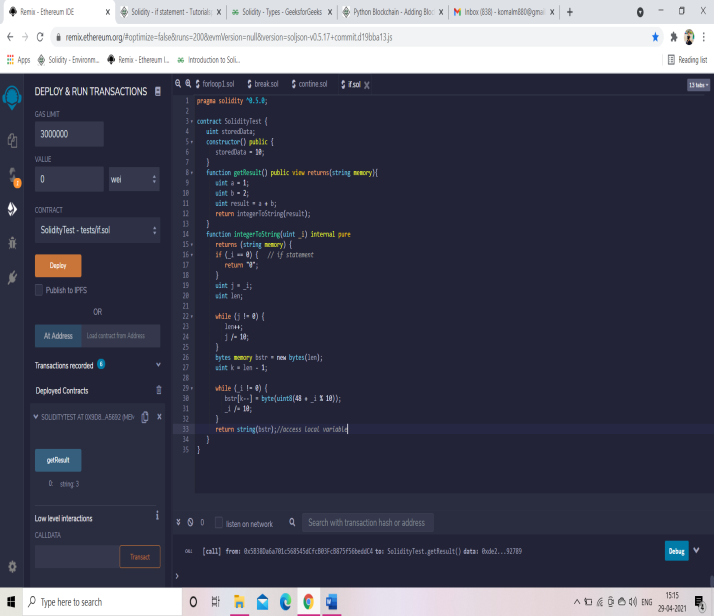
bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);//access local variable

}}



**2.if-else statement:** The **'if...else'** statement is the next form of control statement that allows Solidity to execute statements in a more controlled way.

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

// Declaring state variables

uint i = 10;

bool even;

// Defining function to

// demonstrate the use of

// 'if...else statement'

function decision\_making(

) public payable returns(bool){

if (i%2 == 0){

even = true;

}

else{

even = false;

}

return even;

}

}



**3.if-else..if statement**: The **if...else if...** statement is an advanced form of **if...else** that allows Solidity to make a correct decision out of several conditions.

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

// Declaring state variables

uint i = 12;

string result;

// Defining function to

// demonstrate the use

// of 'if...else if...else

// statement'

function decision\_making (

) public returns(string memory){

if(i<10){

result = "less than 10";

}

else if(i == 10){

result = "equal to 10";

}

else{

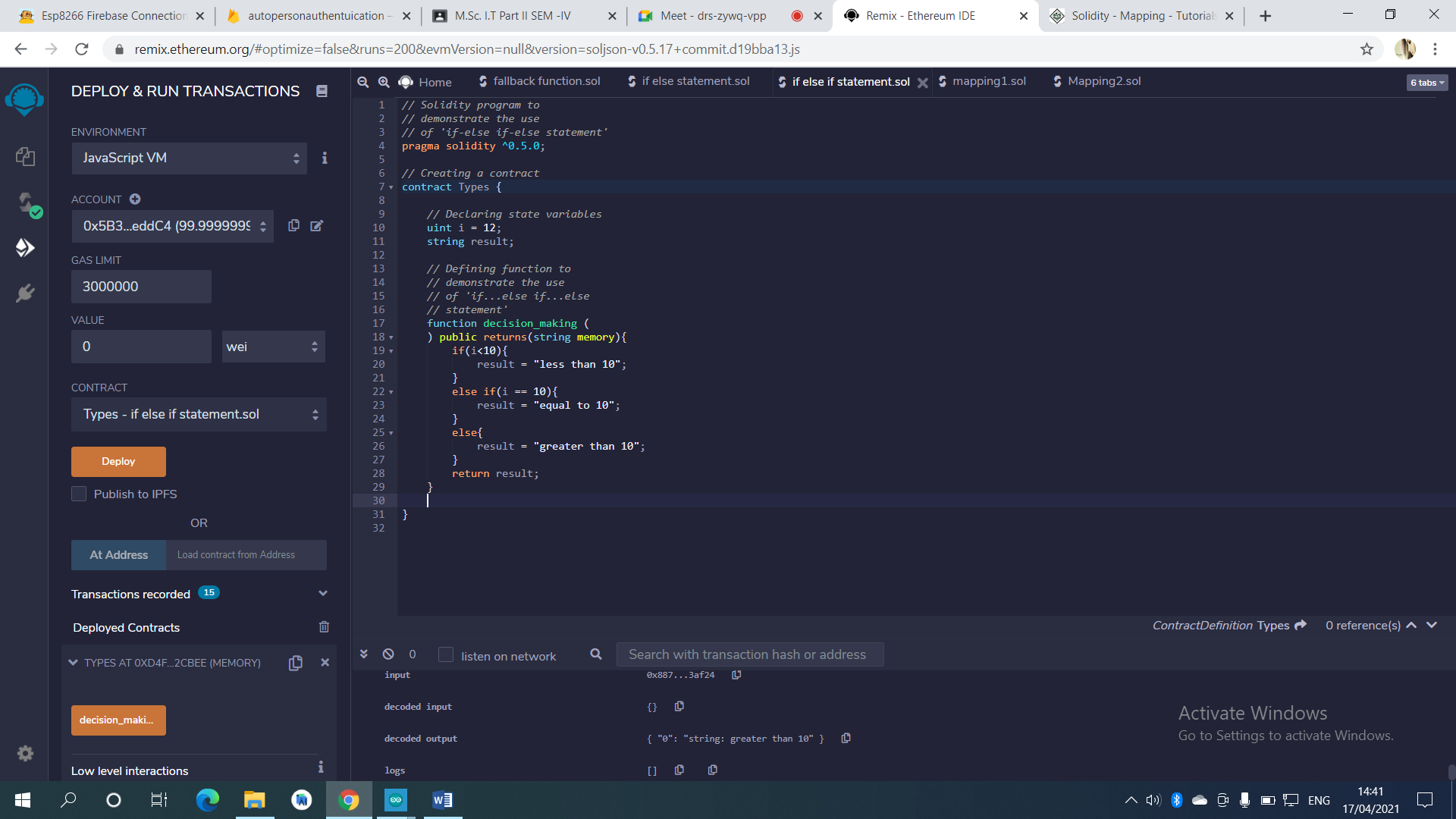
result = "greater than 10";

}

return result;

}

}



**String:**

// Solidity program to demonstrate

// how to create a contract

pragma solidity ^0.4.23;

// Creating a contract

contract Test {

// Declaring variable

string str;

// Defining a constructor

constructor(string str\_in){

str = str\_in;

}

// Defining a function to

// return value of variable 'str'

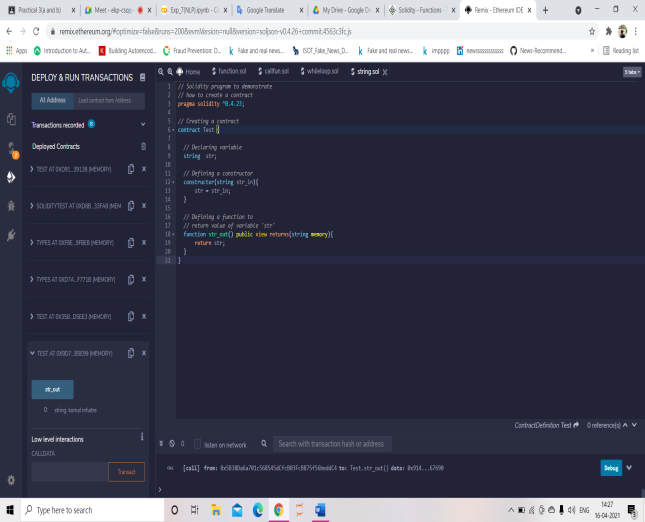
function str\_out() public view returns(string memory){

return str;

}

}

Note: after deploy it asked u to enter string then enter string over there and then see the output after clicking on str\_out button



**PRACTICAL NO.:5**

**AIM: WRITE A SOLIDITY PROGRAM FOR STRING, ARRAYS, ENUMS, STRUCTURE & MAPPINGS.**

**A) String:**

Solidity supports String literal using both double quote (") and single quote ('). It provides string as a data type to declare a variable of type String.(Int to str)

pragma solidity ^0.5.0;

contract SolidityTest {

constructor() public{

}

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

uint a = 1;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (j != 0) {

len++;

j /= 10;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

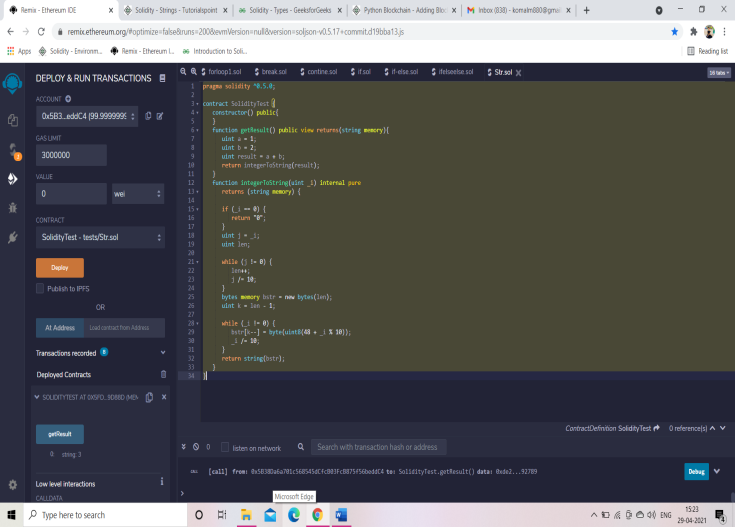
\_i /= 10;

}

return string(bstr);

}

}



**B)Array:**

Array is a data structure, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

// Solidity program to demonstrate

// accessing elements of an array

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

// Declaring an array

uint[6] data;

uint x;

// Defining function to

// assign values to array

function array\_example() public returns (uint[6] memory)

{

data = [uint(10), 20, 30, 40, 50, 60];

}

function result() public view returns(uint[6] memory){

return data;

}

// Defining function to access

// values from the array

// from a specific index

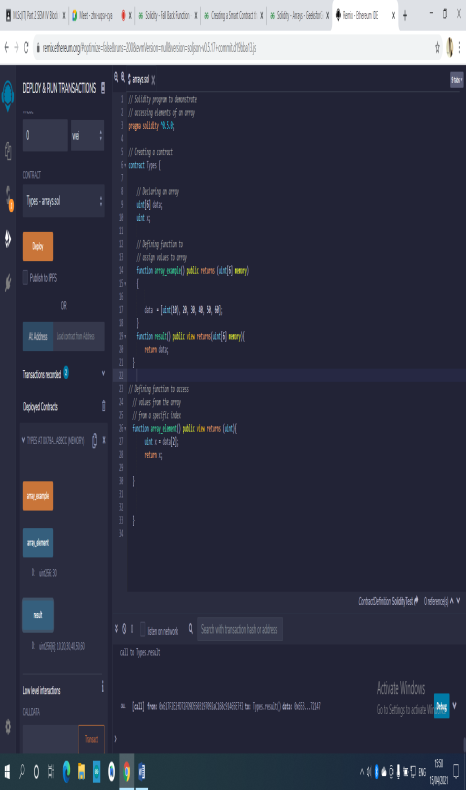
function array\_element() public view returns (uint){

uint x = data[2];

return x;

}

}



**C)Enums:**

Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums. With the use of enums it is possible to reduce the number of bugs in your code.

// Solidity program to demonstrate

// how to use 'enumerator'

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

// Creating an enumerator

enum week\_days

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

// Declaring variables of

// type enumerator

week\_days week;

week\_days choice;

// Setting a default value

week\_days constant default\_value

= week\_days.Sunday;

// Defining a function to

// set value of choice

function set\_value() public {

choice = week\_days.Thursday;

}

// Defining a function to

// return value of choice

function get\_choice(

) public view returns (week\_days) {

return choice;

}

// Defining function to

// return default value

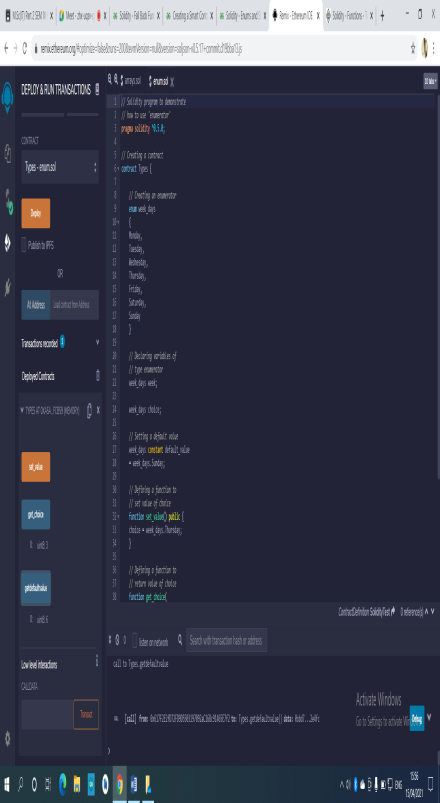
function getdefaultvalue(

) public pure returns(week\_days) {

return default\_value;

}

}



**D)Structure:**

Struct types are used to represent a record.

pragma solidity ^0.5.0;

contract test {

struct Book {

string title;

string author;

uint book\_id;

}

Book book;

function setBook() public {

book = Book('Learn Java', 'TP', 1);

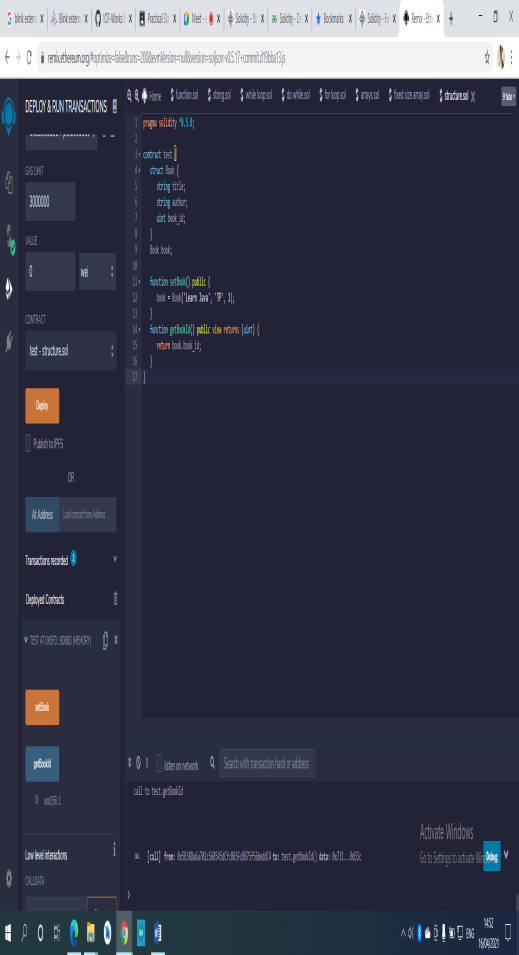
}

function getBookId() public view returns (uint) {

return book.book\_id;

}

}



**E)Mappings:**

Mapping is a reference type as arrays and structs. Following is the syntax to declare a mapping type.

mapping(\_KeyType => \_ValueType) where ,

**\_KeyType** − can be any built-in types plus bytes and string. No reference type or complex objects are allowed.

**\_ValueType** − can be any type.

pragma solidity ^0.5.0;

contract LedgerBalance {

mapping(address => uint) balance;

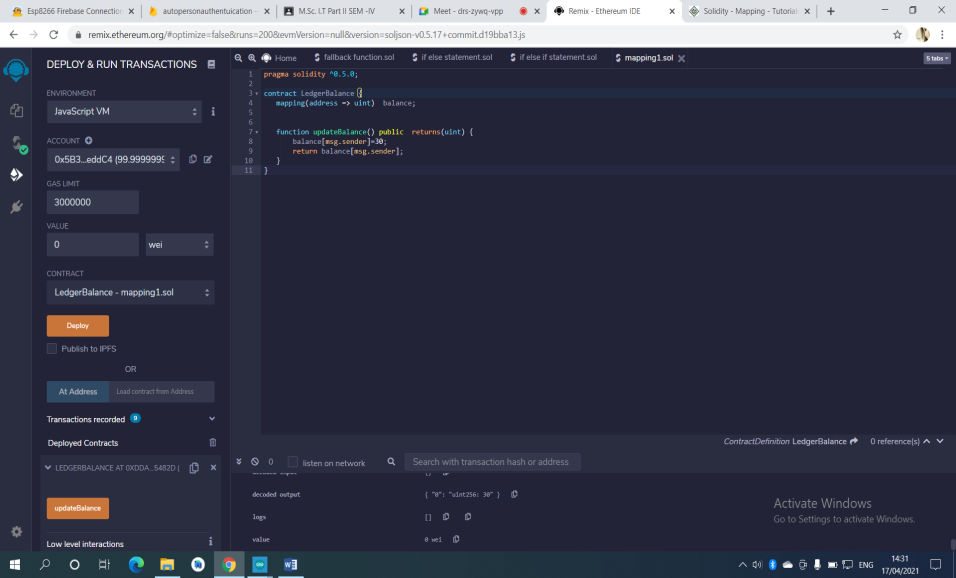
function updateBalance() public returns(uint) {

balance[msg.sender]=30;

return balance[msg.sender];

}

}



**Mapping program for String.**

pragma solidity ^0.5.0;

contract LedgerBalance {

mapping(address => string) name;

function updateBalance() public returns(string memory){

name[msg.sender] = "Mrunali";

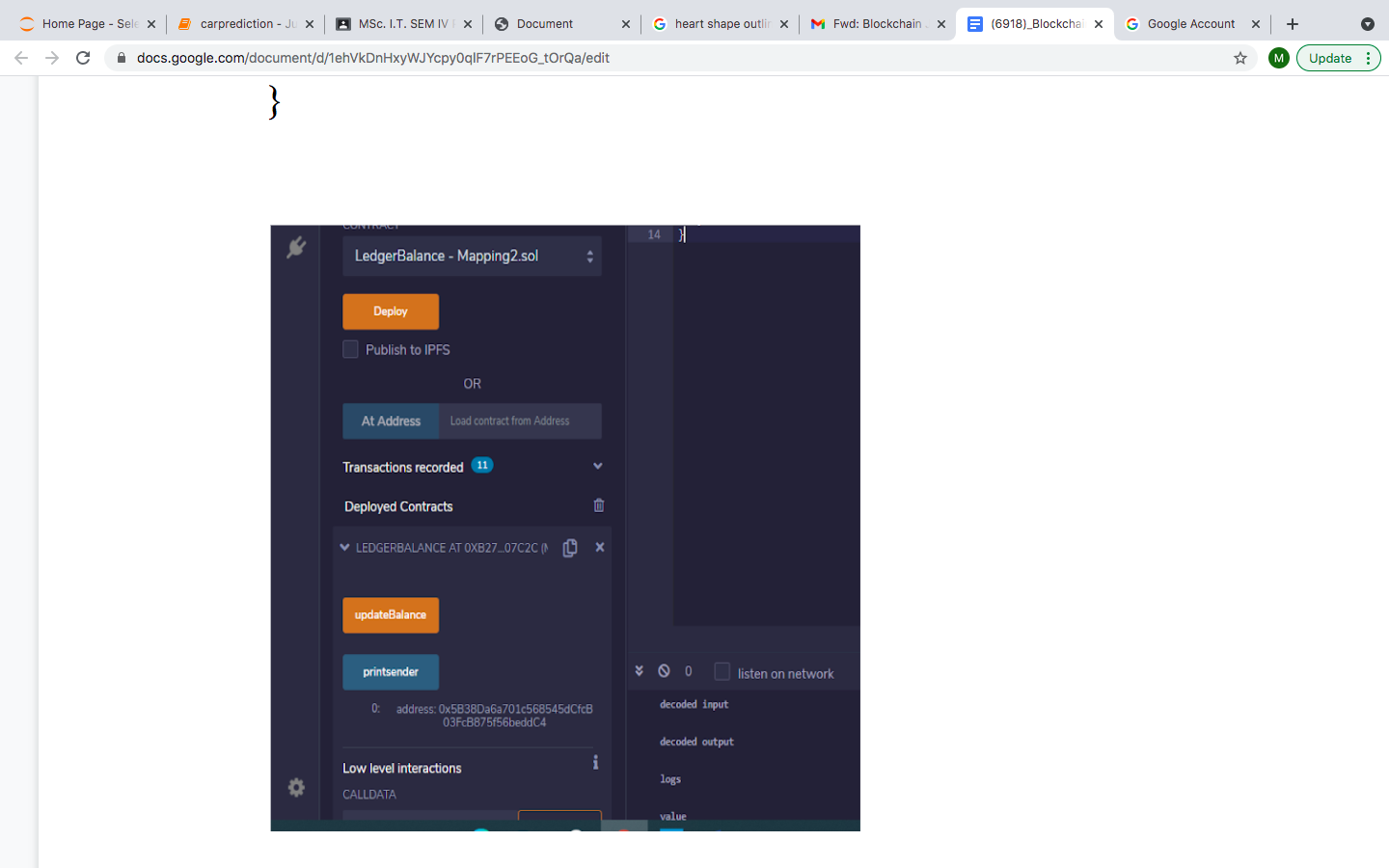
return name[msg.sender];

}

function printsender() public view returns(address) {

return msg.sender;

}

}

**PRACTICAL NO.:6**

**AIM: WRITE A SOLIDITY PROGRAM FOR FUNCTION, VIEW FUNCTION, PURE FUNCTION & FALLBACK FUNCTION.**

**A)Function:**

A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes. Functions allow a programmer to divide a big program into a number of small and manageable functions.

pragma solidity ^0.5.0;

contract SolidityTest {

constructor() public{

}

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

uint a = 1;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (j != 0) {

len++;

j /= 10;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

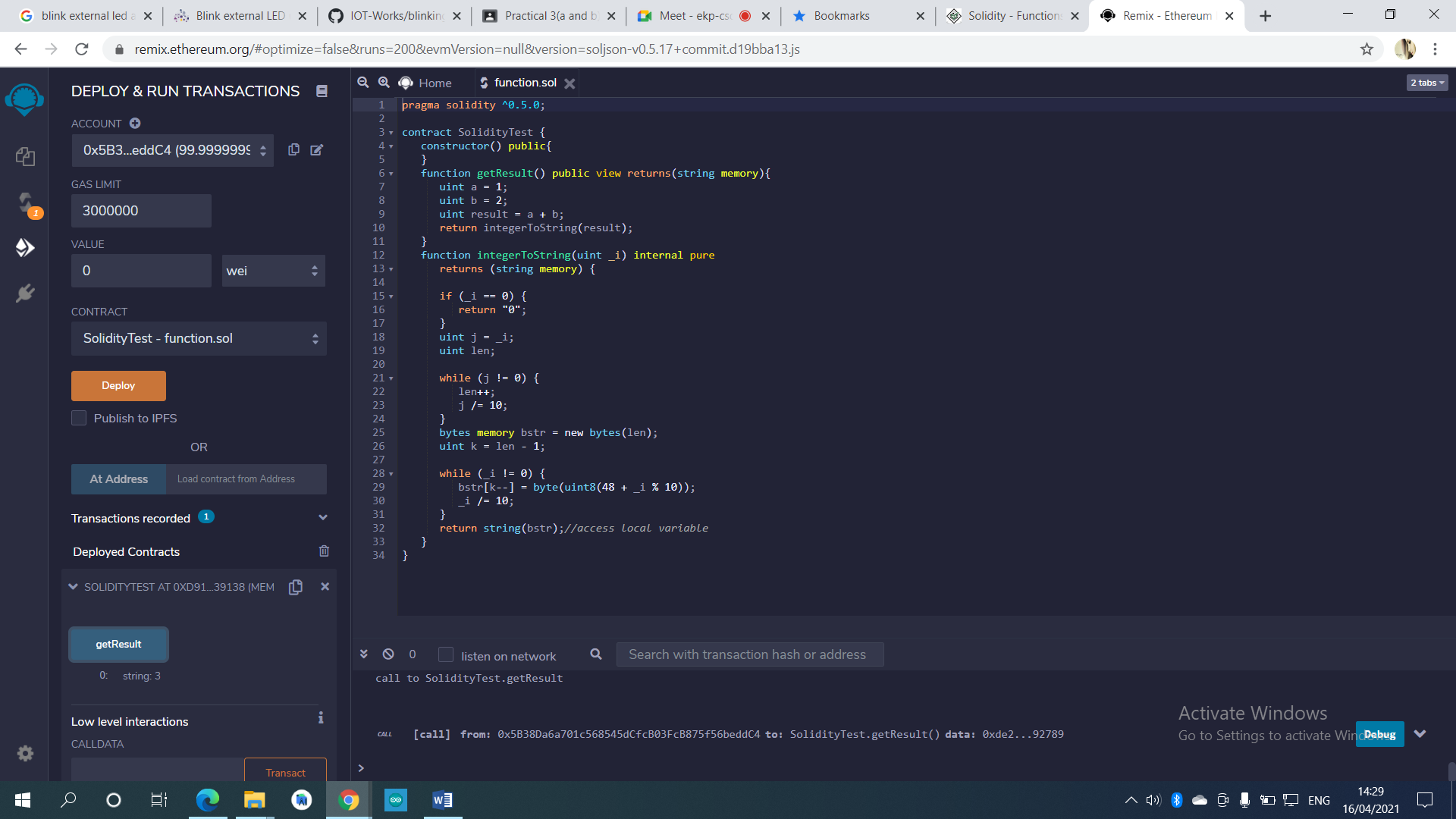
\_i /= 10;

}

return string(bstr);//access local variable

}

}



**B)View Function:**

View functions ensure that they will not modify the state. A function can be declared as **view**. Getter method are by default view functions.

pragma solidity ^0.5.0;

contract Test {

function getResult() public view returns(uint product, uint sum){

uint a = 1; // local variable

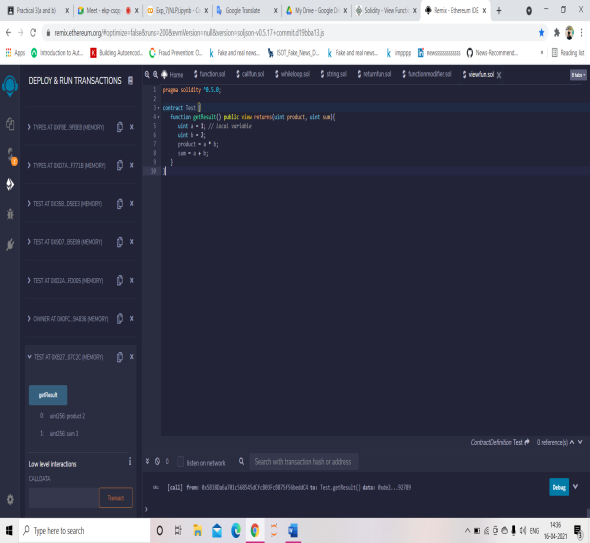
uint b = 2;

product = a \* b;

sum = a + b;

}

}

`

**C)Pure Function:**

Pure functions ensure that they not read or modify the state. A function can be declared as **pure**. Pure functions can use the revert() and require() functions to revert potential state changes if an error occurs.

pragma solidity ^0.5.0;

contract Test {

function getResult() public pure returns(uint product, uint sum){

uint a = 1;

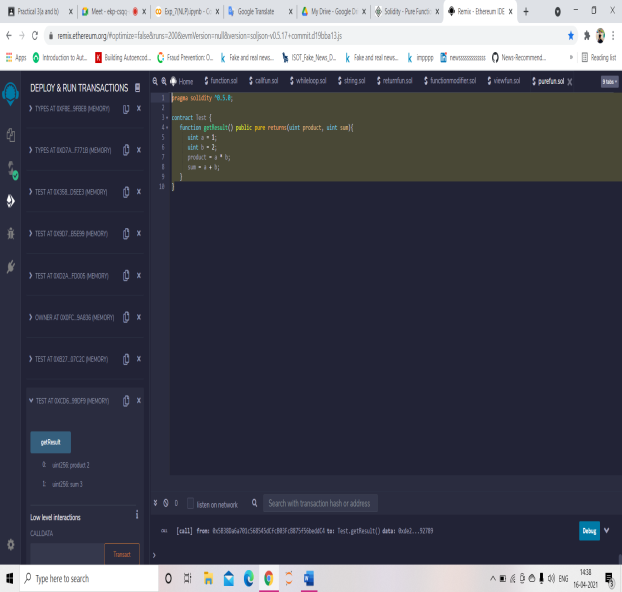
uint b = 2;

product = a \* b;

sum = a + b;

}

}



**D)Fallback Function:**

Fallback function is a special function available to a contract.

pragma solidity ^0.5.0;

contract Test {

uint public x ;

function() external { x = 1; }

}

contract Sink {

function() external payable { }

}

contract Caller {

function callTest(Test test) public returns (bool) {

(bool success,) = address(test).call(abi.encodeWithSignature("nonExistingFunction()"));

require(success);

// test.x is now 1

address payable testPayable = address(uint160(address(test)));

// Sending ether to Test contract,

// the transfer will fail, i.e. this returns false here.

return (testPayable.send(2 ether));

}

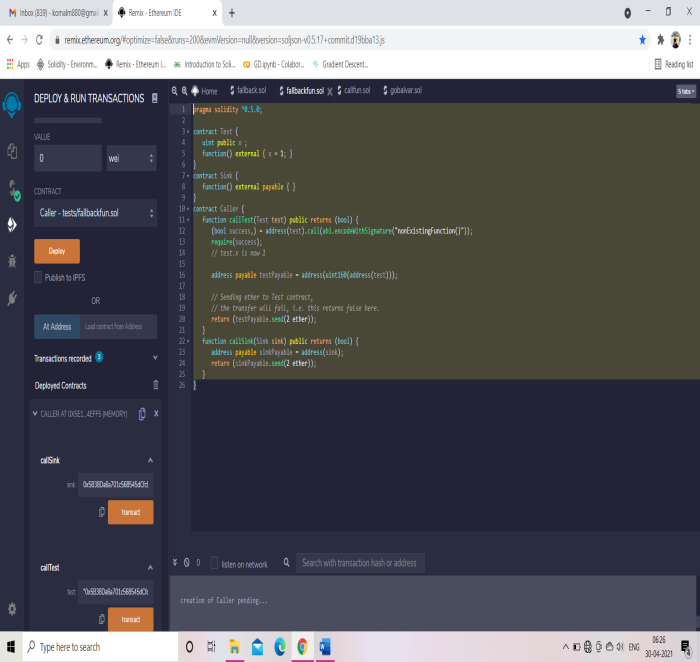
function callSink(Sink sink) public returns (bool) {

address payable sinkPayable = address(sink);

return (sinkPayable.send(2 ether));

}

}



**PRACTICAL NO.:7**

**AIM: WRITE A SOLIDITY PROGRAM FOR FUNCTION OVERLOADING, MATHEMATICAL FUNCTION & CRYPTOGRAPHIC FUNCTIONS.**

**Function Overloading:**

The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

pragma solidity ^0.5.0;

contract Test {

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

return a + b;

}

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

return a + b + c;

}

function callSumWithTwoArguments() public pure returns(uint){

return getSum(2,2);

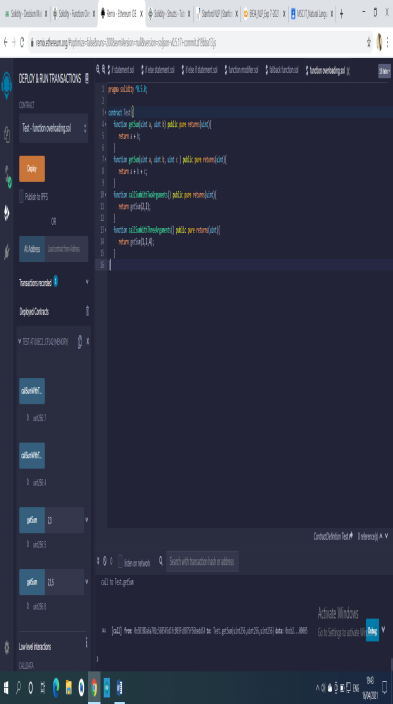
}

function callSumWithThreeArguments() public pure returns(uint){

return getSum(1,2,4);

}

}



**Mathematical Function:**

Solidity provides inbuilt mathematical functions as well.

pragma solidity ^0.5.0;

contract Test {

function callAddMod() public pure returns(uint){

return addmod(4, 5, 3);

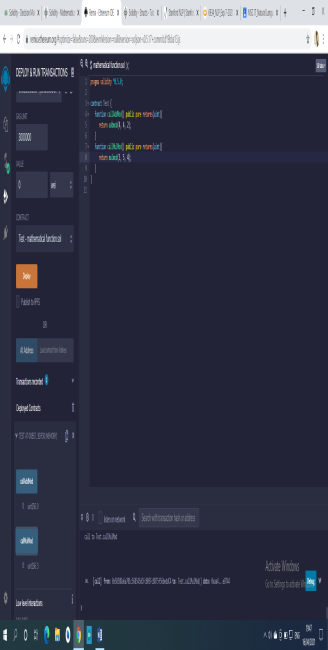
}

function callMulMod() public pure returns(uint){

return mulmod(4, 5, 3);

}

}



**Cryptographic Function:**

Solidity provides inbuilt cryptographic functions as well.

pragma solidity ^0.5.0;

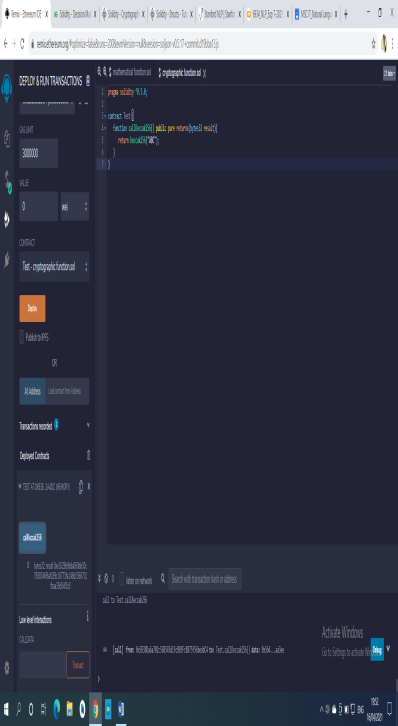
contract Test {

function callKeccak256() public pure returns(bytes32 result){

return keccak256("ABC");

}

}



**PRACTICAL NO.:8**

**AIM: WRITE A SOLIDITY PROGRAM FOR CONTRACT, INHERITANCE, CONSTRUCTORS, ABSTRACT CONTRACTS, INTERFACES, LIBRARIES, ASSEMBLY, EVENTS, ERROR HANDLING.**

**A)Contract:**

Contract in Solidity is similar to a Class in C++. A Contract have following properties.

**Constructor** − A special function declared with constructor keyword which will be executed once per contract and is invoked when a contract is created.

**State Variables** − Variables per Contract to store the state of the contract.

**Functions** − Functions per Contract which can modify the state variables to alter the state of a contract.

// Calling function from external contract

pragma solidity ^0.5.0;

contract C {

//private state variable

uint private data;

//public state variable

uint public info;

//constructor

constructor() public {

info = 10;

}

//private function

function increment(uint a) private pure returns(uint) { return a + 1; }

//public function

function updateData(uint a) public { data = a; }

function getData() public view returns(uint) { return data; }

function compute(uint a, uint b) internal pure returns (uint) { return a + b; }

}

//Derived Contract

contract E is C {

uint private result;

C private c;

constructor() public {

c = new C();

}

function getComputedResult() public {

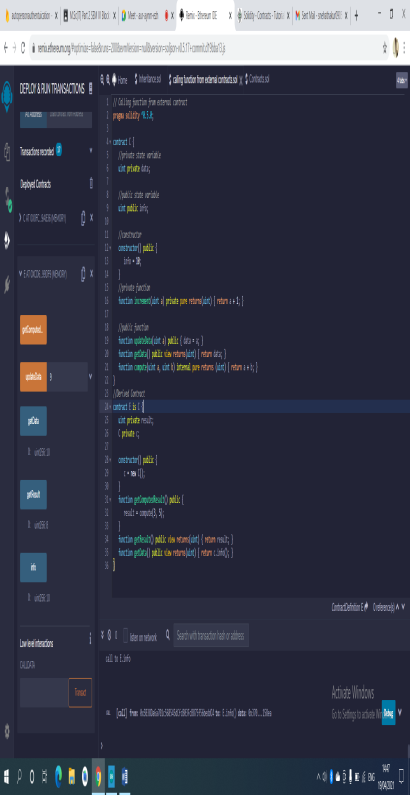
result = compute(3, 5);

}

function getResult() public view returns(uint) { return result; }

function getData() public view returns(uint) { return c.info(); }

}



**B)Inheritance:**

Inheritance is a way to extend functionality of a contract. Solidity supports both single as well as multiple inheritance.

// Solidity program to

// demonstrate

// Single Inheritance

pragma solidity >=0.4.22 <0.6.0;

// Defining contract

contract parent{

// Declaring internal

// state variable

uint internal sum;

// Defining external function

// to set value of internal

// state variable sum

function setValue() external {

uint a = 20;

uint b = 20;

sum = a + b;

}

}

// Defining child contract

contract child is parent{

// Defining external function

// to return value of

// internal state variable sum

function getValue() external view returns(uint) {

return sum;

}

}

// Defining calling contract

contract caller {

// Creating child contract object

child cc = new child();

// Defining function to call

// setValue and getValue functions

function testInheritance() public {

cc.setValue();

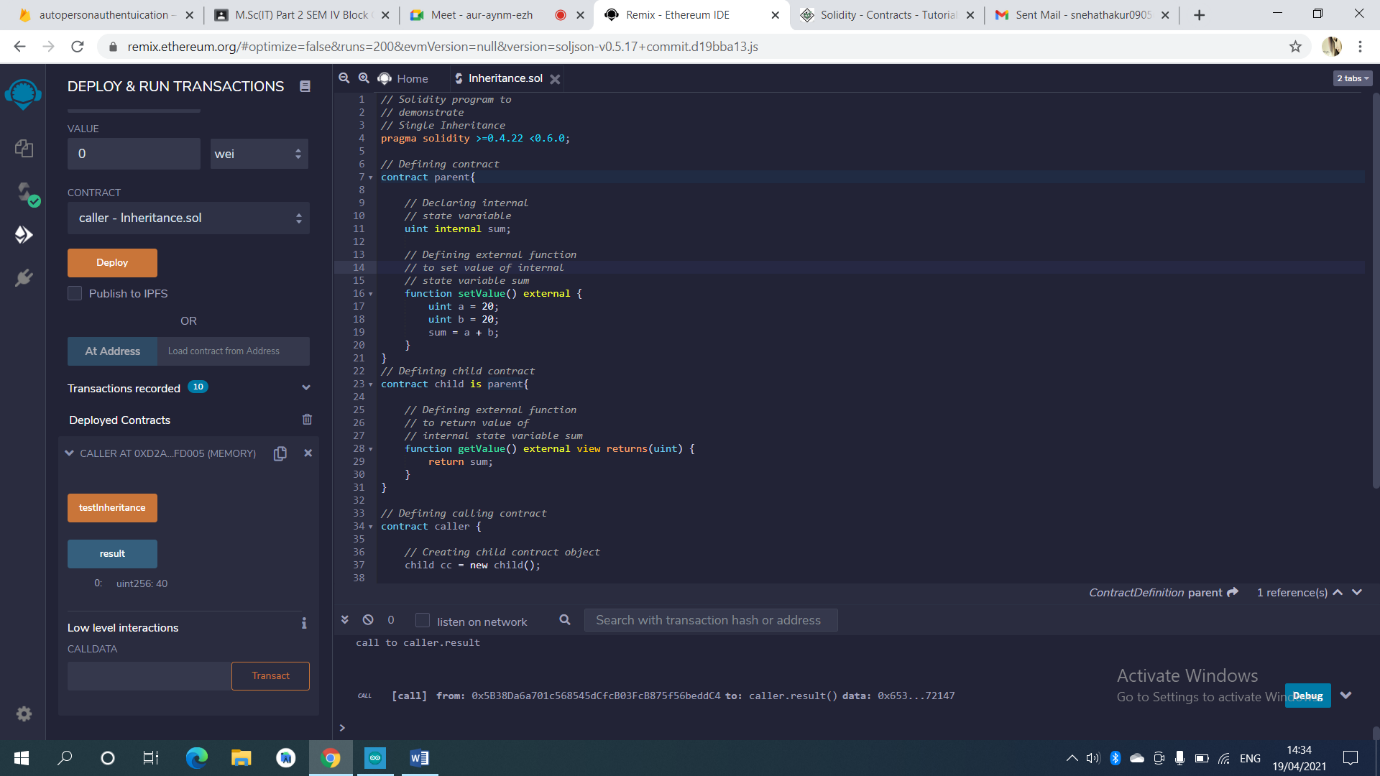
}

function result() public view returns(uint ){

return cc.getValue();

}

}



**C)Constructors:**

Constructor is a special function declared using constructor keyword. It is an optional function and is used to initialize state variables of a contract. Following are the key characteristics of a constructor.

A contract can have only one constructor.

A constructor code is executed once when a contract is created and it is used to initialize contract state.

A constructor can be either public or internal.

An internal constructor marks the contract as abstract.

In case, no constructor is defined, a default constructor is present in the contract.

pragma solidity ^0.5.0;

contract Base {

uint data;

constructor(uint \_data) public {

data = \_data;

}

function getresult()public view returns(uint){

return data;

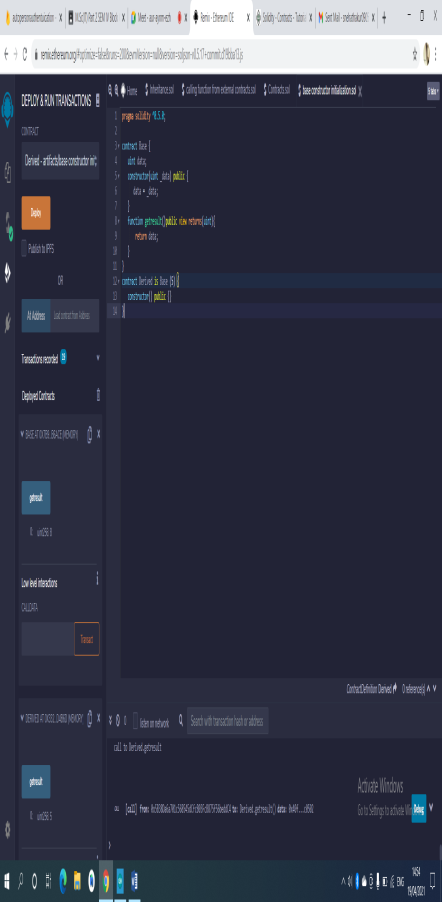
}

}

contract Derived is Base (5) {

constructor() public {}

}



// Indirect Initialization of Base Constructor

pragma solidity ^0.5.0;

contract Base {

uint data;

constructor(uint \_data) public {

data = \_data;

}

function getresult()public view returns(uint){

return data;

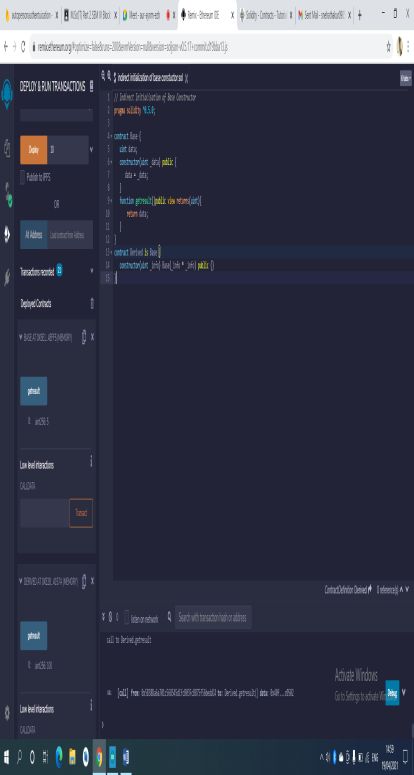
}

}

contract Derived is Base {

constructor(uint \_info) Base(\_info \* \_info) public {}

}



**D)Abstract Contracts:**

Abstract Contract is one which contains at least one function without any implementation. Such a contract is used as a base contract. Generally an abstract contract contains both implemented as well as abstract functions. Derived contract will implement the abstract function and use the existing functions as and when required.

pragma solidity ^0.5.0;

contract Calculator {

function getResult() public view returns(uint);

}

contract Test is Calculator {

function getResult() public view returns(uint) {

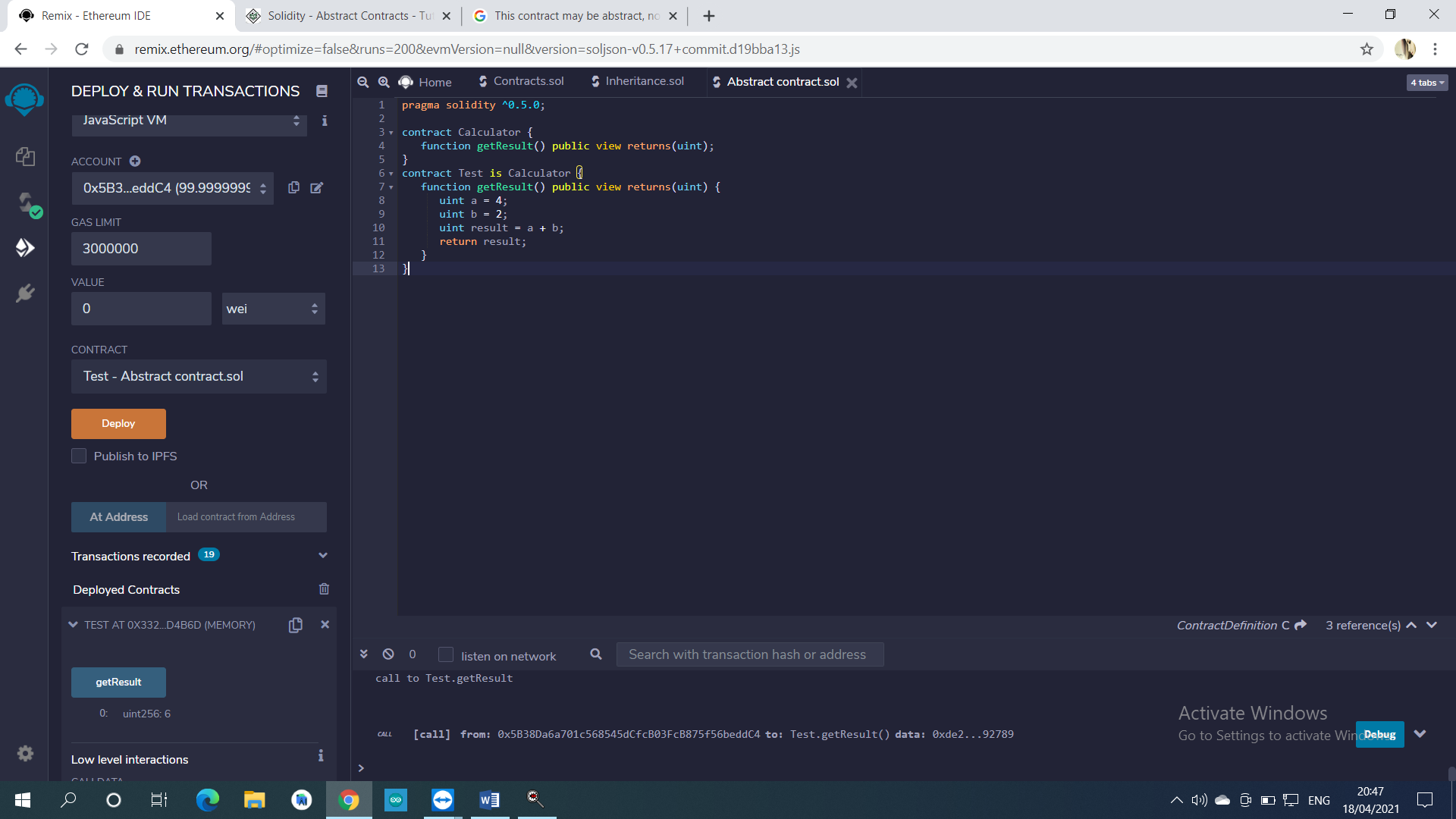
uint a = 4;

uint b = 2;

uint result = a + b;

return result;

}

}

**E)Interfaces:**

Interfaces are similar to abstract contracts and are created using interface keyword. Following are the key characteristics of an interface.

Interface can not have any function with implementation.

Functions of an interface can be only of type external.

Interface can not have constructor.

Interface can not have state variables.

pragma solidity ^0.5.0;

interface Calculator {

function getResult() external view returns(uint);

}

contract Test is Calculator {

constructor() public {}

function getResult() external view returns(uint){

uint a = 5;

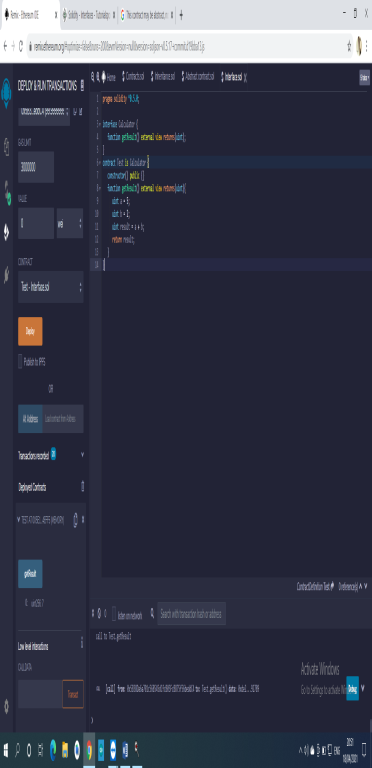
uint b = 2;

uint result = a + b;

return result;

}

}



**F)Libraries:**

Libraries are similar to Contracts but are mainly intended for reuse. A Library contains functions which other contracts can call. Solidity have certain restrictions on use of a Library.

pragma solidity ^0.5.0;

library Search {

function indexOf(uint[] storage self, uint value) public view returns (uint) {

for (uint i = 0; i < self.length; i++)

if (self[i] == value) return i;

return uint(-1);}

}

contract Test {

uint[] data;

uint value;

uint index;

constructor() public {

data.push(6);

data.push(7);

data.push(8);

data.push(9);

data.push(10);

}

function isValuePresent() external {

value = 9;

//search if value is present in the array using Library function

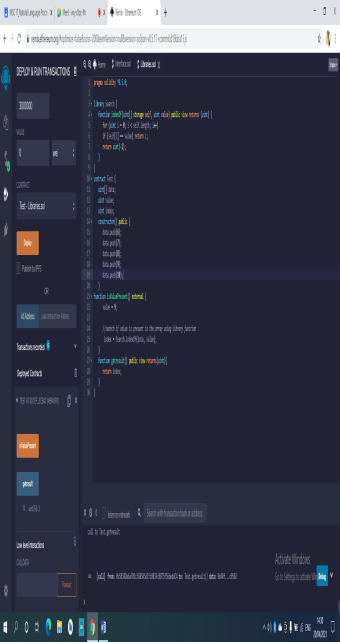
index = Search.indexOf(data, value);

}

function getresult() public view returns(uint){

return index;

}}



**G)Assembly:**

Solidity provides an option to use assembly language to write inline assembly within Solidity source code. We can also write a standalone assembly code which then be converted to bytecode. Standalone Assembly is an intermediate language for a Solidity compiler and it converts the Solidity code into a Standalone Assembly and then to byte code. We can used the same language used in Inline Assembly to write code in a Standalone assembly.

pragma solidity ^0.5.0;

library Sum {

function sumUsingInlineAssembly(uint[] memory \_data) public pure returns (uint o\_sum) {

for (uint i = 0; i < \_data.length; ++i) {

assembly {

o\_sum := add(o\_sum, mload(add(add(\_data, 0x20), mul(i, 0x20))))

}}

}

}

contract Test {

uint[] data;

constructor() public {

data.push(1);

data.push(2);

data.push(3);

data.push(4);

data.push(5);

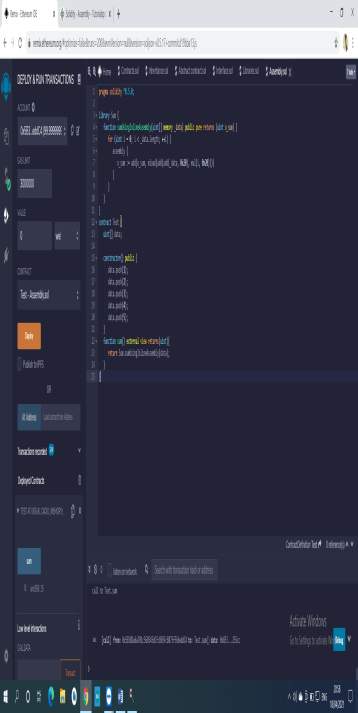
}

function sum() external view returns(uint){

return Sum.sumUsingInlineAssembly(data);

}

}



**H)Events:**

Event is an inheritable member of a contract. An event is emitted, it stores the arguments passed in transaction logs. These logs are stored on blockchain and are accessible using address of the contract till the contract is present on the blockchain. An event generated is not accessible from within contracts, not even the one which have created and emitted them.

// Solidity program to demonstrate

// creating an event

pragma solidity ^0.4.21;

// Creating a contract

contract eventExample {

// Declaring state variables

uint256 public value = 0;

// Declaring an event

event Increment(address owner);

// Defining a function for logging event

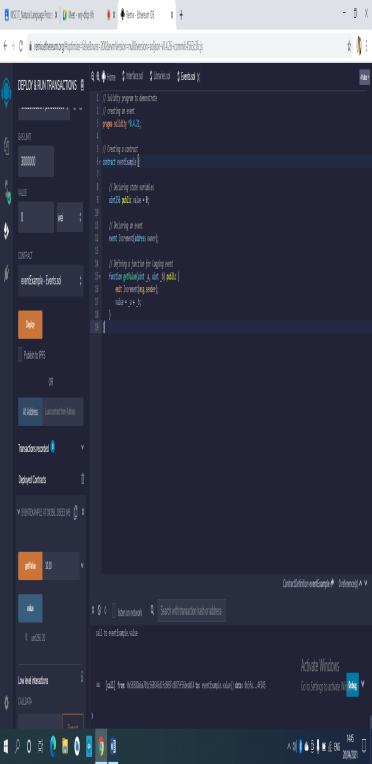
function getValue(uint \_a, uint \_b) public {

emit Increment(msg.sender);

value = \_a + \_b;

}

}



**I)Error Handling:**

Solidity provides various functions for error handling. Generally when an error occurs, the state is reverted back to its original state. Other checks are to prevent unauthorized code access.

**Solidity program to demonstrate require statement.**

// Solidity program to

// demonstrate require

// statement

pragma solidity ^0.5.0;

// Creating a contract

contract requireStatement {

// Defining function to

// check input

function checkInput(uint8 \_input) public view returns(string memory){

require(\_input >= 0, "invalid uint");

require(\_input <= 255, "invalid uint8");

return "Input is Uint8";

}

// Defining function to

// use require statement

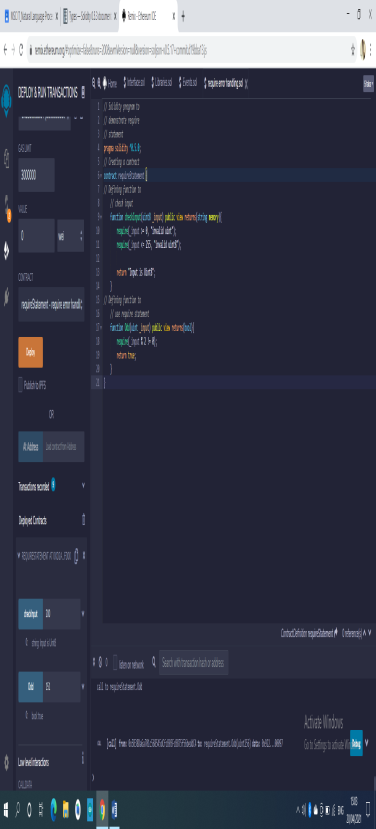
function Odd(uint \_input) public view returns(bool){

require(\_input % 2 != 0);

return true;

}

}



**Solidity program to demonstrate assert statement.**

// Solidity program to

// demonstrate assert

// statement

pragma solidity ^0.5.0;

// Creating a contract

contract assertStatement {

// Defining a state variable

bool result;

// Defining a function

// to check condition

function checkOverflow(uint8 \_num1, uint8 \_num2) public {

uint8 sum = \_num1 + \_num2;

assert(sum<=255);

result = true;

}

// Defining a function to

// print result of assert

// statement

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

if(result == true){

return "No Overflow";

}

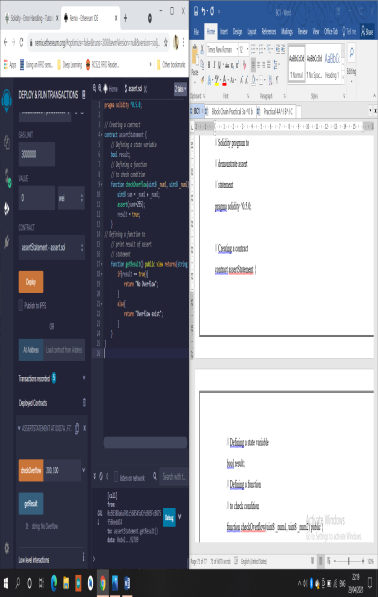
else{

return "Overflow exist";

}

}

}



**Solidity program to demonstrate revert statement.**

// Solidity program to

// demonstrate revert

// statement

pragma solidity ^0.5.0;

// Creating a contract

contract revertStatement {

// Defining a function

// to check condition

function checkOverflow(uint \_num1, uint \_num2) public view returns(

string memory, uint) {

uint sum = \_num1 + \_num2;

if(sum < 0 || sum > 255){

revert(" Overflow Exist");

}

else{

return ("No Overflow", sum);

}

}

}

