1. Install MetaMask wallet. Create an account and deposit some ethers from the faucets into your account. Create several accounts in MetaMask and exchange ethers within accounts. Explore Blockchain Explorer and EtherScan.
2. Explore remix IDE. Write a smart contract to print “hello world” in solidity on Remix IDE. Write a smart contract to add two numbers taken input from the user.

**PRINT HELLO WORLD**

//SPDX-License-Identifier: GPL-3.0  
pragma solidity >= 0.4.16 < 0.9.0;  
contract PrintHelloWorld  
{  
    function sayHello() public pure returns(**string** memory){  
        **return** "Hello World";  
    }  
      
}

**ADD 2 NUMBERS FROM INPUT**

// SPDX-License-Identifier: MIT  
pragma solidity >= 0.4.16 < 0.9.0;  
  
contract AddTwoNumebers{  
    **uint** a;  
    **uint** b;  
    **uint** sum;  
  
    function set(**uint** x, **uint** y) public{  
        a = x;  
        b = y;  
        sum = a+b;  
    }  
      
    function get() public view returns (**uint**){  
       **return** sum;  
    }  
}

1. Write a smart contract to create a simple calculator using the concept of constructor using global and local variables.

// SPDX-License-Identifier: MIT  
pragma solidity >= 0.4.16 < 0.9.0;  
  
contract Calculator{  
    **int** a;  
    **int** b;  
  
    function set(**int** x,**int** y) public{  
        a=x;  
        b=y;  
    }  
  
    function add() public view returns(**int**){  
        **return** a+b;  
    }  
  
    function subtract() public view returns(**int**){  
        **return** a-b;  
    }  
  
    function multiply() public view returns(int256){  
        **return** a\*b;  
    }  
  
    function divide() public view returns (int256) {  
        **return** a/b;  
    }  
  
    function modulus() public view returns(**int**){  
        **return** a%b;  
    }  
  
    function power() public view returns (int256) {  
        require(b >= 0, "Exponent must be non-negative");  
        int256 result = 1;  
        **for** (int256 i = 0; i < b; i++) {  
            result \*= a;  
        }  
        **return** result;  
    }  
}

1. Write a smart contract to calculate the sum of 1st n natural numbers using for loop, while loop and do while loop to demonstrate the working of loops.

**FOR LOOP**

// SPDX-License-Identifier: MIT  
pragma solidity >= 0.4.16 < 0.9.0;  
  
contract SumOfN{  
    uint256 n;  
    function set(uint256 \_n) public{  
        require(\_n>0);  
        n = \_n;  
    }  
    function get() public view returns (uint256){  
        **uint** sum=0;  
        **for** (**uint** i=1; i<=n; i++)   
        {  
           sum+=i;  
        }  
        **return** sum;  
    }  
}

**WHILE LOOP**

function get() public view returns (uint256){  
        **uint** sum=0;  
        **uint** i = 1;  
  
        while(i<=n){  
            sum+=i;  
            i++;  
        }  
        **return** sum;  
    }

**DO-WHILE LOOP**

 function get() public view returns (uint256){  
        **uint** sum=0;  
        **uint** i = 1;  
        do{  
            sum += i;  
            i = i + 1;  
        }while(i<=n);  
        **return** sum;  
    }  
}

1. Write a smart contract to create a smart contracts circle, rectangle and square which are inherited from smart contract shapes using the concept of inheritance.

// SPDX-License-Identifier: MIT  
pragma solidity >=0.4.16 <0.9.0;  
  
abstract contract Shape {  
    function area() public virtual view returns (**uint**);  
    function perimeter() public virtual view returns (**uint**);  
}  
  
contract Circle is Shape {  
    **uint** private radius;  
  
    constructor(**uint** \_radius) {  
        radius = \_radius;  
    }  
    function area() public view override returns (**uint**) {  
        **return** **uint**(314 \* radius \* radius) / 100;  
    }  
    function perimeter() public view override returns (**uint**) {  
        **return** **uint**(628 \* radius) / 100;  
    }  
}  
  
contract Rectangle is Shape{  
    **uint** private length;  
    **uint** private width;  
  
    constructor(**uint** \_l, **uint** \_w) {  
        length = \_l;  
        width = \_w;  
    }  
    function area() public view override returns (**uint**) {  
        **return** (length \* width);  
    }  
    function perimeter() public view override returns(**uint**) {  
        **return** 2\*(length + width);  
    }  
}  
  
contract Square is Rectangle{  
    constructor (**uint** \_l) Rectangle(\_l,\_l){}  
}

1. Write a smart contract to update the state variable value and emit an event (ValueChanged) that logs the previous value, the new value, and the address of the caller to showcase event handling in solidity.

// SPDX-License-Identifier: MIT  
pragma solidity >= 0.4.16 < 0.9.0;  
  
contract StateUpdater {  
    **uint** public value;  
  
    event ValueChanged(**uint** indexed oldValue, **uint** indexed newValue, address indexed caller);  
  
    function updateValue(**uint** \_newValue) public {  
        **uint** oldValue = value;  
        value = \_newValue;  
        emit ValueChanged(oldValue, \_newValue, msg.sender);  
    }  
}

1. Write a program to send ethers and receive ethers using a payable function. Also explore how to evaluate the gas value of the transaction?

// SPDX-License-Identifier: MIT  
pragma solidity >= 0.4.16 < 0.9.0;  
  
contract EtherTransaction {  
    address public owner;  
  
    event Received(address indexed sender, **uint** amount);  
  
    event Sent(address indexed recipient, **uint** amount);  
  
    constructor() {  
        owner = msg.sender; // Set the contract deployer as the owner  
    }  
  
    receive() external payable {  
        emit Received(msg.sender, msg.value);  
    }  
  
    function sendEther(address payable \_recipient, **uint** \_amount) public payable {  
        require(msg.sender == owner, "Only the owner can send Ether");  
        require(address(this).balance >= \_amount, "Insufficient balance in the contract");  
        \_recipient.transfer(\_amount);  
        emit Sent(\_recipient, \_amount);  
    }  
  
    function getBalance() public view returns (**uint**) {  
        **return** address(this).balance;  
    }  
}

1. Write a code that manages a balance with deposit, withdrawal, and reset functions, using error handling to ensure valid operations and prevent invalid states.

// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;  
  
contract BalanceManager {  
    // State variable to store the balance  
    **uint** public balance;  
  
    // Event to log deposits  
    event Deposited(address indexed user, **uint** amount);  
  
    // Event to log withdrawals  
    event Withdrawn(address indexed user, **uint** amount);  
  
    // Event to log balance reset  
    event Reset(address indexed user);  
  
    // Function to deposit Ether  
    function deposit() public payable {  
        require(msg.value > 0, "Deposit amount must be greater than zero");  
        balance += msg.value;  
        emit Deposited(msg.sender, msg.value);  
    }  
  
    // Function to withdraw Ether  
    function withdraw(**uint** \_amount) public {  
        require(\_amount > 0, "Withdrawal amount must be greater than zero");  
        require(balance >= \_amount, "Insufficient balance for withdrawal");  
          
        balance -= \_amount;  
        payable(msg.sender).transfer(\_amount);  
        emit Withdrawn(msg.sender, \_amount);  
    }  
  
    // Function to reset the balance  
    function reset() public {  
        require(balance == 0, "Balance must be zero to reset");  
        emit Reset(msg.sender);  
    }  
  
    // Get the contract's balance (for debugging or testing)  
    function getContractBalance() public view returns (**uint**) {  
        **return** address(this).balance;  
    }  
}

1. Write a smart contract to define a simple ERC20 token contract with minting and burning functions, allowing the owner to create new tokens and users to destroy their own tokens.ERC20 tokens.

// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;  
  
**import** "@openzeppelin/contracts/token/ERC20/ERC20.sol";  
**import** "@openzeppelin/contracts/access/Ownable.sol";  
  
contract SimpleToken is ERC20, Ownable {  
    // Constructor to initialize the token with a name, symbol, and owner  
    constructor(**string** memory name, **string** memory symbol)   
        ERC20(name, symbol)   
        Ownable(msg.sender) // Pass the deployer as the initial owner  
    {}  
      
    // Function for the owner to mint new tokens  
    function mint(address to, uint256 amount) public onlyOwner {  
        \_mint(to, amount);  
    }  
  
    // Function for users to burn their own tokens  
    function burn(uint256 amount) public {  
        \_burn(msg.sender, amount);  
    }  
}

1. Interact with the smart contracts created using Web3 using python or javascript