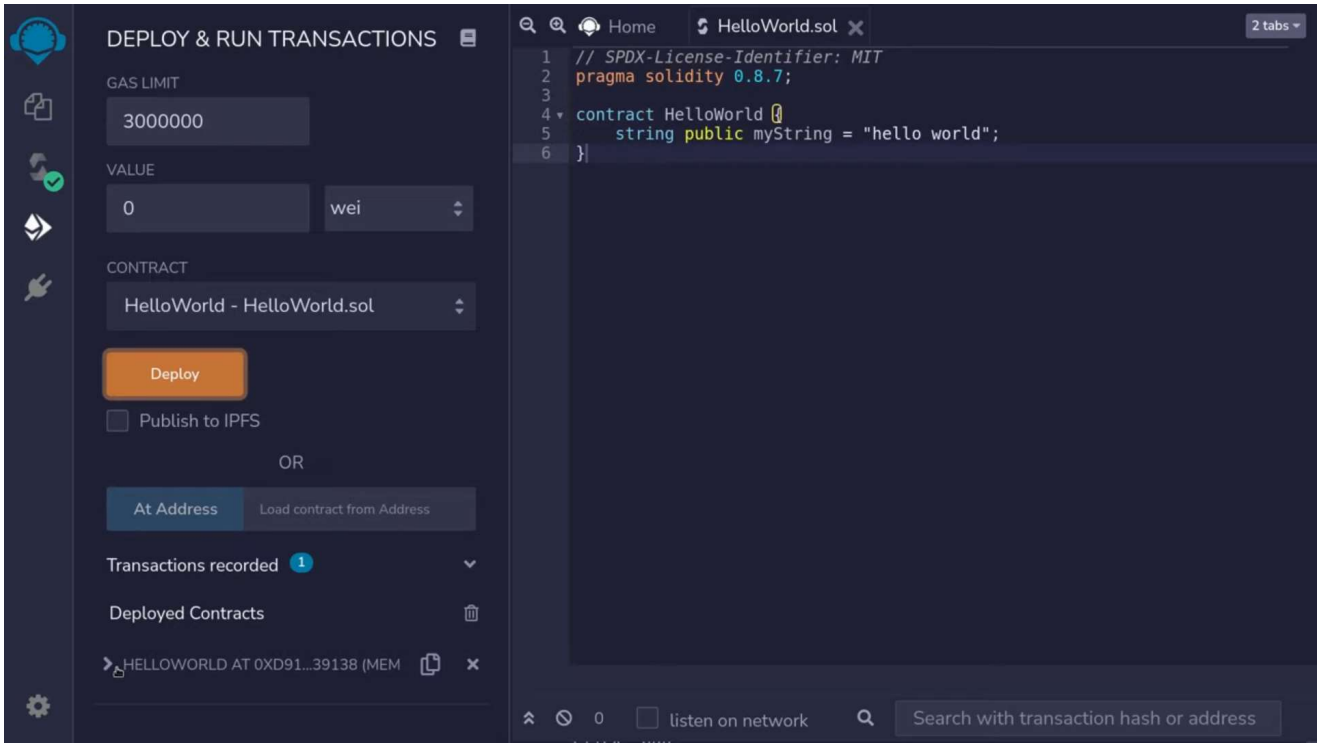
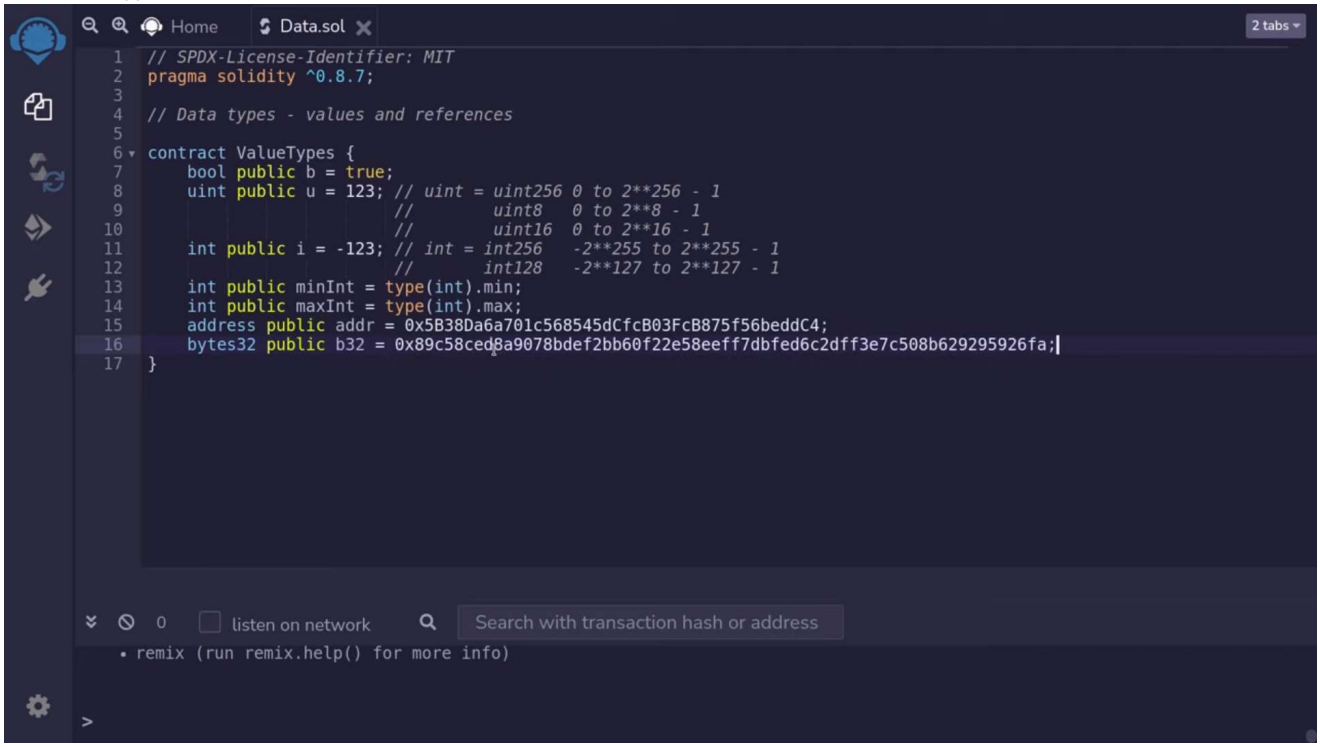
1. Hello World:



2. Data Types:



3. Pure and View Functions:

ViewAndPureFunc.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

contract ViewAndPureFunctions {
    uint public num;

    function viewFunc() external view returns (uint) {
        return num;
    }

    function pureFunc() external pure returns (uint) {
        return 1;
    }

    function addToNum(uint x) external view returns (uint) {
        return num + x;
    }

    function add(uint x, uint y) external pure returns (uint) {
        return x + y;
    }
}
```

4. If Else and Ternery:

IfElse.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

contract IfElse {
    function example(uint _x) external pure returns (uint) {
        if (_x < 10) {
            return 1;
        } else if (_x < 20) {
            return 2;
        } else {
            return 3;
        }
    }

    function ternary(uint _x) external pure returns (uint) {
        // if (_x < 10) {
        //     return 1;
        // }
        // return 2;
        return _x < 10 ? 1 : 2;
    }
}
```

5. While and For Loop:

```
pragma solidity ^0.8.7;

contract ForAndWhileLoops {
    function loops() external pure {
        for (uint i = 0; i < 10; i++) {
            // code
            if (i == 3) {
                continue;
            }
            // more code
            if (i == 5) {
                break;
            }
        }

        uint j = 0;
        while (j < 10) {
            // code
            j++;
        }
    }

    function sum(uint _n) external pure returns (uint) {
        uint s;
        for (uint i = 1; i <= _n; i++) {
            s += i;
        }
        return s;
    }
}
```

6. Error handling => Assert, Require and Revert:

```
pragma solidity ^0.8.7;

// require, revert, assert
// - gas refund, state updates are reverted
// custom error - save gas
contract Error {
    function testRequire(uint _i) public pure {
        require(_i <= 10, "i > 10");
        // code
    }

    function testRevert(uint _i) public pure {
        if (_i > 10) {
            revert("i > 10");
        }
    }

    uint public num = 123;

    function testAssert() public view {
        assert(num == 123);
    }
}
```

7. Constructor:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

contract Constructor {
    address public owner;
    uint public x;

    constructor(uint _x) {
        owner = msg.sender;
        x = _x;
    }
}
```

8. Function Outputs:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

// Return multiple outputs
// Named outputs
// Destructuring Assignment

contract FunctionOutputs {
    function returnMany() public pure returns (uint, bool) {
        return (1, true);
    }

    function named() public pure returns (uint x, bool b) {
        return (1, true);
    }

    function assigned() public pure returns (uint x, bool b) {
        x = 1;
        b = true;
    }

    function destructingAssigments() public pure {
        (uint x, bool b) = returnMany();
        (, bool _b) = returnMany();
    }
}
```

9. Arrays:

Home | README.txt | Array.sol | 3 tabs

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

// Array - dynamic or fixed size
// Initialization
// Insert (push), get, update, delete, pop, length
// Creating array in memory
// Returning array from function

contract Array {
    uint[] public nums = [1, 2, 3];
    uint[3] public numsFixed = [4, 5, 6];

    function examples() external {
        nums.push(4); // [1, 2, 3, 4]
        uint x = nums[1];
        nums[2] = 777; // [1, 2, 777, 4]
        delete nums[1]; // [1, 0, 777, 4]
        nums.pop(); // [1, 0, 777]
        uint len = nums.length;

        // create array in memory
        uint[] memory a = new uint[](5);
        a[1] = 123;
    }

    function returnArray() external view returns (uint[] memory) {
        return nums;
    }
}
```

uint256[] nums | 7 reference(s)

0 | listen on network | Search with transaction hash or address

The following libraries are accessible:

10. Mappings:

Home | 4_Ballot_test.sol | README.txt | Mapping.sol | 4 tabs

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

// Mapping
// How to declare a mapping (simple and nested)
// Set, get, delete

// ["alice", "bob", "charlie"]
// { "alice": true, "bob": true, "charlie": true }
contract Mapping {
    mapping(address => uint) public balances;
    mapping(address => mapping(address => bool)) public isFriend;

    function examples() external {
        balances[msg.sender] = 123;
        uint bal = balances[msg.sender];
        uint bal2 = balances[address(1)]; // 0

        balances[msg.sender] += 456; // 123 + 456 = 579

        delete balances[msg.sender];
    }
}
```

0 | listen on network | Search with transaction hash or address

The following libraries are accessible:

11. Struct:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.7;

contract test {
   struct Employee {
      uint id;
      string name;
      uint salary;
   }
   Employee employee;

   function addEmployee() public {
      employee = Employee(1,'john', 5000);
   }
   function getEmployyeeId() public view returns (uint) {
      return employee.id;
   }
}
```

12. Enum:

```
// Solidity program example for Enum Type declaration
pragma solidity ^0.5.0;


// Defining contract
contract EnumTest {
    enum Number { One,Two,Three}
    Number public number=Number.Three;
    constructor() public {
    }
    function getNumber() public view returns (uint) {
     return uint(number);
}
}
```

13. Strings:

String literals are enclosed in double quotes(") or single quotes(')

If it is declared in a global contract, called a global scope variable

```
pragma solidity ^0.5.0;

// Create contract for String type testing
contract StringTest {
    string name;
    string role="admin";

}
```

There is another way to declare string types with byte32. Byte32 works with string declaration as well as string literal syntax.

```
pragma solidity ^0.5.0;

// Create contract for String type testing
contract StringTest {
    byte32 name;
    byte32 role="admin";

}
```

Both `string` and `byte32` do store the string data.

14. State and Local Variables:

```
pragma solidity ^0.5.0;
contract VariablesContractTest {
   /*
   *State variables are declared inside a contract ie global scope
   */
   uint age; // State variable
   constructor() public {
      age = 10;
   }
   function add() public view returns(uint){
      // local variables are declared and accessed inside a function
      uint localVariable1 = 12; // local variable
      uint localVariable2 = 2; // local variable
      uint sum = localVariable1 + localVariable2; // local variable
      return sum; // access the local variable
   }
}
```