

Functions

In Solidity, a Function is executable block that performs an operation.

```
28 ▾ function set(string memory _value, string memory _MyVal1) public {  
29     value = _value;  
30     myVal = _MyVal1;  
31 }  
32 }
```

Functions can take parameters, and/or return parameters.

```
23 ▾ function get() public view returns(string memory, string memory) {  
24     return (value, myVal);  
25 }  
26 }  
27 }  
28 ▾ function set(string memory _value, string memory _MyVal1) public {  
29     value = _value;  
30     myVal = _MyVal1;  
31 }  
32 }  
  
22 ▾ function testVal(uint _iTst) internal pure returns (string memory){  
23     uint inum;  
24     string memory myString;  
25     inum = _iTst;  
26     if(inum == 0){  
27         myString = "The number is 0";  
28     }  
29     if(inum > 0 && inum < 10){  
30         myString = "The number is small";  
31     }  
32     else if(inum >= 10 && inum < 100) {  
33         myString = "The number is big";  
34     }  
35     else if(inum >= 100 && inum < 1000) {  
36         myString = "The number is way big";  
37     }  
38     else if(inum >= 1000 && inum < 999999) {  
39         myString = "The number is huge";  
40     }  
41     else{  
42         myString = "The number is way huge and out of sight";  
43     }  
44     return myString;  
45 }  
46 }  
47 }  
48 }  
49 }  
50 }  
51 }  
52 }  
53 }  
54 }  
55 }  
56 }  
57 }
```

Functions need to be specified as public, private, internal, or external

Public functions are visible and can be called by other function or by the contract interface.

```
28 function set(string memory _value, string memory _MyVal1) public {  
29     value = _value;  
30     myVal = _MyVal1;  
31 }
```

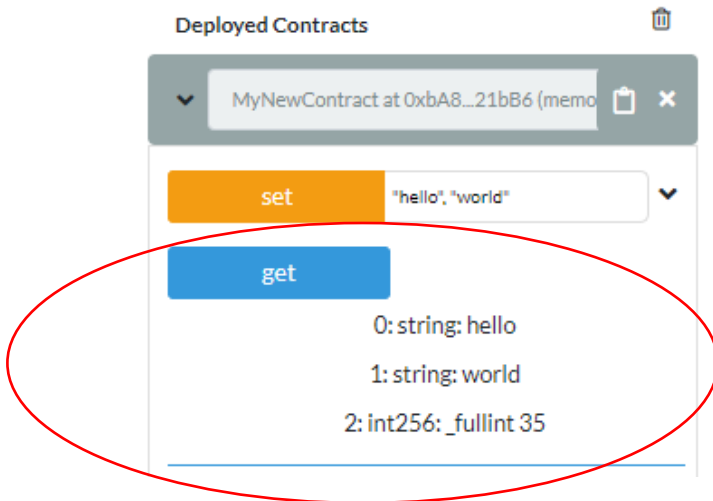
set string_value, string_MyVal1

Private functions are only visible within their own contract.

Private functions can only be called by other functions within their own contract and cannot be called by other contracts.

```
16 function setint (int256 i) private {  
17  
18     myInt=i;  
19     fullInt = myInt + 10;  
20  
21 }  
22  
23 function set(string memory _value, string memory _MyVal1) public {  
24     value = _value;  
25     myVal = _MyVal1;  
26     setint(j);  
27 }  
28  
29 function get() public view returns(string memory, string memory, int256 _fullint) {  
30     return (value, myVal, fullInt);  
31  
32 }  
33 }
```

The private function is part of the interface, but it was called by the set function, and its results were returned by the get function.



Functions declared as external can be called from other contracts

External functions cannot be called internally.

If there are large amounts of data, external functions may be more efficient.

The example below shows the set function, cannot call the external setint function.

```
16  function setint (int256 i) external {
17
18      myInt=i;
19      fullInt = myInt + 10;
20
21  }
22
23  function set(string memory _value, string memory _MyVal1) public {
24      value = _value;
25      myVal = _MyVal1;
26      setint(j);
27  }
28
```

Calling functions from other Contracts

Functions and data can be called from other contracts.

Contracts can be declared within other contracts using the **new** command.

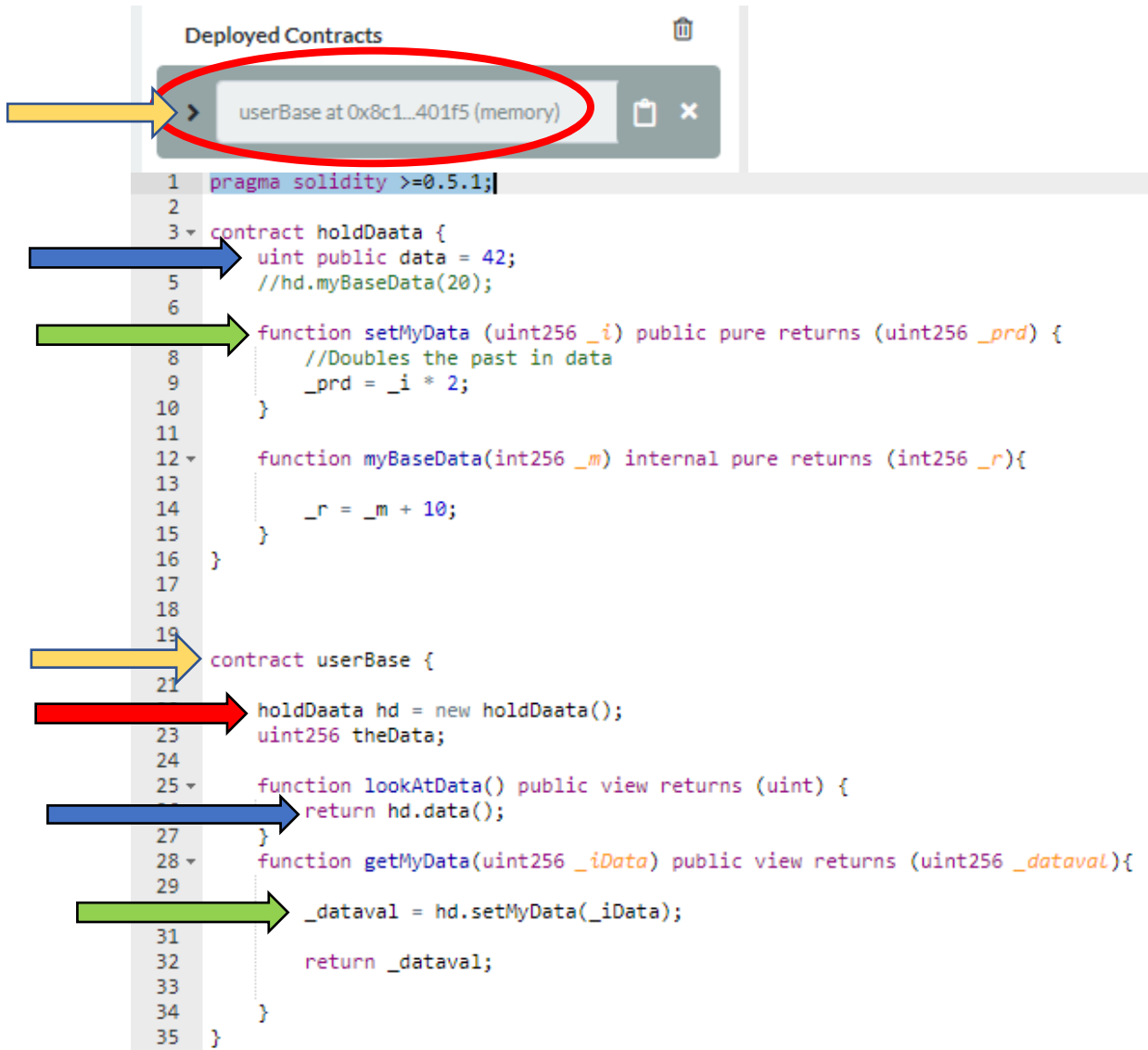
Once a contract is declared, within another contract, the newly declared contract's public functions and data can be called.

In the example below, we have two contracts, holdDaata, and userBase.

Contract holdData is declared within userBase, as hd, with the **new** command.

Contract holdDaata's public functions and data are available to contract userBase.

When you run a contract where other contract are in the same file, make sure the correct contract is select for execution



Contact userBase cannot call contract's holdDaata's myBaseData function directly, as function myBaseData is internal.

A compiler error is returned.

Another function within contract holdDaata will have to call function myBaseData for it to be used by function userBase.

```
1  pragma solidity >=0.5.1;
2
3  contract holdDaata {
4      uint public data = 42;
5      //hd.myBaseData(20);
6
7      function setMyData (uint256 _i) public pure returns (uint256 _prd) {
8          //Doubles the past in data
9          _prd = _i * 2;
10     }
11
12     function myBaseData(int256 _m) internal pure returns (int256 _r){
13
14         _r = _m + 10;
15     }
16 }
17
18
19
20 contract userBase {
21
22     holdDaata hd = new holdDaata();
23     uint256 public theData;
24
25     function lookAtData() public view returns (uint256 _k) {
26         return hd.data();
27     }
28     function getMyData(uint256 _iData) public returns (uint256 _dataval){
29
30         _dataval = hd.setMyData(_iData);
31
32         hd.myBaseData(20);
33
34         theData = _dataval;
35         return theData;
36     }
37 }
38 }
```

Calling Functions

Solidity supports calling functions, from other functions.

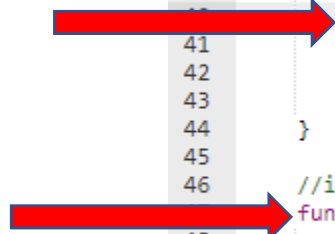
A function call may or may not return data

A function call may or may not take input parameters.

A function call is performed the same way it is performed in Java, C++, or JavaScript.

The example below shows function `addUser` calling function `userCnt`, to count the number of users.

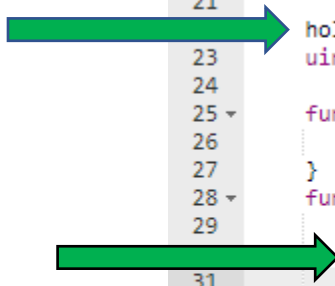
```
37 function addUser(string memory _fname, string memory _lname) public onlyOwner {  
38  
39     //adding internal function  
40     userCnt();  
41     users.push(User(_fname, _lname));  
42  
43  
44 }  
45  
46 //internal  
47 function userCnt() internal {  
48     userCounter += 1;  
49  
50 }  
51  
52
```



As discussed earlier, functions can be called from other contracts.

The other contract must be instantiated using `new`

```
20 contract userBase {  
21  
22     holdDaata hd = new holdDaata();  
23     uint256 public theData;  
24  
25     function lookAtData() public view returns (uint256 _k) {  
26         return hd.data();  
27     }  
28     function getMyData(uint256 _iData) public returns (uint256 _dataval){  
29  
30         _dataval = hd.setMyData(_iData);  
31  
32         theData = _dataval;  
33         return theData;  
34     }  
35 }  
36
```



Inheritance

Solidity supports inheritance and polymorphism similar to Java and C++

Inheritance between contract is implemented using `is` keyword

Inheritance is defined in the function header, and multiple inheritance is supported.

The example below shows contract `userBase` is or inherits the attributes and functions of function `basicData`

Introduction to Blockchain Development with Ethereum by Coding-Bootcamps.com
Session 7- Smart Contracts- Functions

```
21 contract basicData {  
22     uint256 public keyData = 100;  
23     uint256 public newVaue;  
24  
25  
26     function setVal(uint256 _iVal) public {  
27         newValue = _iVal;  
28     }  
29 }  
30  
31 contract userBase is basicData{  
32     holdDaata hd = new holdDaata();  
33     uint256 public theData;  
34  
35     function lookAtData() public view returns (uint256 _k) {  
36         return hd.data();  
37     }  
38  
39     function getMyData(uint256 _iData, uint256 _imyData) public returns (uint256 _dataval, uint256 _key){  
40  
41         _dataval = hd.setMyData(_iData);  
42         setVal(_imyData);  
43         theData = _dataval;  
44         return (theData, keyData);  
45     }  
46 }  
47  
48
```

Contract userBase uses setValue to set the value of variable newValue, and it also has the variable keyData

Deployed Contracts

userBase at 0x610...44118 (memory)

getMyData 6,20

setVal 12

keyData

0: uint256: 100

lookAtData

newValue

0: uint256: 12

theData

Returns, and Return

When a function returns data, it is declared in the header, and also in the function's statement.

If the function statement returns the data declared in the header, return is not needed.

No return statement is needed

```
contract Simple {  
    function arithmetic(uint _a, uint _b)  
        public  
        pure  
        returns (uint o_sum, uint o_product)  
    {  
        o_sum = _a + _b;  
        o_product = _a * _b;  
    }  
}
```

```
contract Simple {  
    function arithmetic(uint _a, uint _b)  
        public  
        pure  
        returns (uint o_sum, uint o_product)  
    {  
        return (_a + _b, _a * _b);  
    }  
}
```


Pure Functions

Pure functions do not update or read the transaction or block state values.

Hashes are sensitive to changes.

Deterministic functions return the same value every time they are run.

Pure functions do access state values such as: address.balance

Pure functions do not access state variables: block, tx, msg

Pure functions only call other pure functions.

```
1 pragma solidity >=0.5.0 <0.7.0;
2
3 contract addEm {
4     function myAdd(uint256 numVal, uint256 baseVal) public pure returns (uint _a) {
5
6         _a = numVal * baseVal + 16;
7     }
8 }
9 }
```

Pure function myAdd attempts to call not pure function testP, and a compiler error is returned

```
1 pragma solidity >=0.5.0 <0.7.0;
2
3 contract addEm {
4     function myAdd(uint256 numVal, uint256 baseVal) public pure returns (uint _a) {
5
6         _a = numVal * baseVal + 16;
7         testP("test Pure");
8     }
9 }
10
11 function testP(string memory _mystr) public returns (string memory _str){
12
13     _str = _mystr;
14 }
15
16 }
```

Now it works

```
1 pragma solidity >=0.5.0 <0.7.0;
2
3 contract addEm {
4     function myAdd(uint256 numVal, uint256 baseVal) public pure returns (uint _a) {
5
6         _a = numVal * baseVal + 16;
7         testP("test Pure");
8     }
9 }
10
11 function testP(string memory _mystr) public pure returns (string memory _str){
12
13     _str = _mystr;
14 }
15
16 }
```

View Functions

Just like pure functions view functions do not modify the state.

View functions do not write to state variables or emit events.

View functions do not create other contracts or make Ether calls.

View functions only call other functions declare as view or pure.

```
1  pragma solidity >=0.5.0 <0.7.0;
2
3  contract addEm {
4      function myAdd(uint256 numVal, uint256 baseVal) public view returns (uint _a) {
5
6          _a = numVal * baseVal + 16 + now;
7
8      }
9
10 }
```

Fallback Functions

A fallback function is an unnamed function in a contract.

A fallback function does not take or return arguments

A fallback must have external visibility.

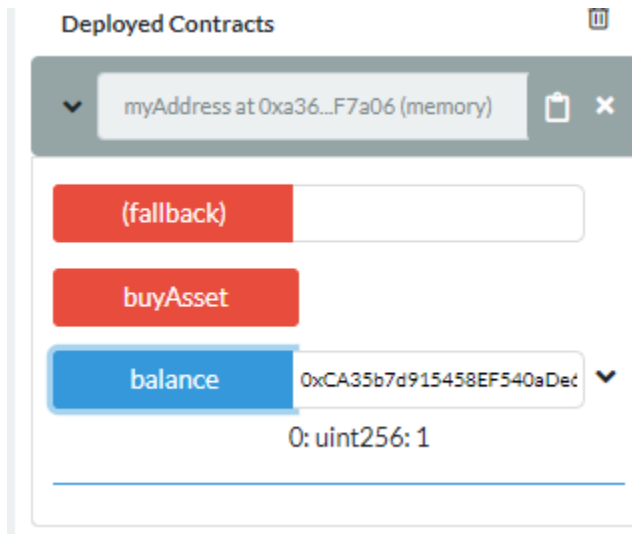
As the name implies, the fallback is called if no other functions match request function identifier.

The fallback executes when the contract receive Ether.

A fallback must be payable to receive Ether and add it to the balance

```
1  pragma solidity ^0.5.1;
2
3  contract myAddress {
4
5      //mapping is a key / value pair
6      mapping(address => uint256 ) public balance;
7      address payable mywallet1;
8
9      event Purchase (
10         address _buyer,
11         uint256 _amount
12     );
13
14     //event are for subscribing and filtering for an event.
15
16     constructor (address payable _wallet) public {
17         mywallet1 = _wallet;
18     }
19
20
21     function() external payable {
22
23         buyAsset();
24     }
25
26
27     function buyAsset() public payable{
28         //buy
29
30         balance[msg.sender] += 1;
31         mywallet1.transfer(msg.value);
32         //send ether to a wallet
33         emit Purchase(msg.sender, 1);
34     }
35
36
37
38 }
```

Clicking (fallback) runs buyAsset. The fallback is payable, therefore the fallback button is red. Since it is payable, it can add to the address balance.



Function Modifiers

Function Modifiers, as the name implies, change the way a function executes.

Modifier is like custom function types

Modifiers are called in the function's header and can test conditions during run time.

Modifiers are inheritable.

The example below restricts running of the function addUser only to the address that deployed the contract.

If the account address running the contract changes, the addUser function will return an error.

```
46 modifier onlyadminUser() {  
47     require (msg.sender == admin);  
48     _;  
49 }  
50  
51  
52  
53  
54 function addUser(string memory _fname, string memory _lname) public onlyadminUser {  
55     //adding internal function  
56     addUserCnt();  
57     users[usrCounter] = User(usrCounter, _fname, _lname);  
58 }  
59  
60  
61
```

Deploying account address: 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c
It successfully runs.

Account 0xCA3...a733c (99.9)

Gas limit 3000000

Value 0 wei

MyMappingContract - browser/myT

Deploy

or

At Address Load contract from Address

Transactions recorded: 2

Deployed Contracts

MyMappingContract at 0x5E7...26e9f (me)

addUser "Tom", "Sims"

users 1

0: uint256: _id1 1
1: string: _fname Tom
2: string: _lname Sims


usrCounter




0: uint256: 1

Success

[vm] from:0xca3...a733c to:MyMappingContract.addUser(string,string) 0x5e7...26e9f value:0 wei data:0x079...00000 logs:0 hash:0x8b4...0205c

Now the account address running function addUser is changed to:
0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB

Account 


0x4B0...4D2dB (99.5   



Gas limit

3000000

Value

0

wei 


MyMappingContract - browser/myT  


Deploy




or

At Address


Load contract from Address

Transactions recorded: 3 


Deployed Contracts 

 MyMappingContract at 0x5E7...26e9f (me  

addUser

"John", "Dahl" 

users

1 

0: uint256: _id1 1


1: string: _fname Tom


2: string: _lname Sims

usrCounter

0: uint256: 1

Error is returned

 [vm] from:0x4b0...4d2db to:MyMappingContract.addUser(string,string) 0x5e7...26e9f value:0 wei data:0x079...00000 logs:0 hash:0x585...42169

Debug 

transaction to MyMappingContract.addUser errored: VM error: revert.
revert. The transaction has been reverted to the initial state.
Note: The called function should be payable if you send value and the value you send should be less than your current balance. Debug the transaction to get more information.

Constructors

Constructors are optional functions used for contract initialization.

Constructors are the first to execute when the contract runs.

Once the constructor runs, the whole contract is deployed to the blockchain.

Constructors are either public or private.

If the constructor is omitted a default Solidity runs a default constructor.

The Constructor below ensures that the contract is deployed by an owner Ethereum wallet address. The contract will not deploy without an Ethereum wallet address.

```
1  pragma solidity ^0.5.1;
2
3  contract myAddress {
4
5      //mapping is a key / value pair
6      mapping(address => uint256 ) public balance;
7      address payable mywallet1;
8
9      event Purchase (
10         address _buyer,
11         uint256 _amount
12     );
13
14     //event are for subscribing and filtering for an event.
15
16     constructor (address payable _wallet) public {
17         mywallet1 = _wallet;
18     }
19
20
21     function() external payable {
22
23         buyAsset();
24     }
25
26
27     function buyAsset() public payable{
28         //buy
29
30         balance[msg.sender] += 1;
31         mywallet1.transfer(msg.value);
32         //send ether to a wallet
33         emit Purchase(msg.sender, 1);
34     }
35
36
37
38 }
```

Introduction to Blockchain Development with Ethereum by Coding-Bootcamps.com
Session 7- Smart Contracts- Functions

Deploy

0xCA35b7d915458EF540aDe6068c

▼

or

At Address

Load contract from Address

Transactions recorded: 1

▼

Deployed Contracts

🗑

▼

myAddress at 0xB87...69cfA (memory)

📄

✕

(fallback)

buyAsset

balance

address

▼

The constructor below initialized the Ethereum address for admin. The admin address id the contract owner.

```
32  constructor() public {  
33  
34      admin = msg.sender;  
35  
36  }
```


Overloading Functions

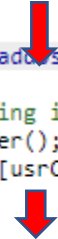
Solidity supports overloading functions

A contract can have more than one function with the same name, as long as parameters are different.

At runtime the contract runs the called function whose parameters match.

The example below shows 2 functions in the same contract named addUser

This is permitted because the functions parameters are different.



```
60 function addUser(string memory _fname, string memory _lname) public {
61
62     //adding internal function
63     addUser();
64     users[usrCounter] = User(usrCounter, _fname, _lname);
65 }
66
67 function addUser() internal {
68     usrCounter += 1;
69 }
70 }
```

When addUser is called with the parameters, addUser(), the addUser at line 67 is called and executed, function addUser().

addUser	"Mike", "Sims" ✓
---------	------------------

If addUser is called with 2 string input parameters, the addUser, function addUser(string memory _fname, string memory _lname) public {, is called, and executed.

Ethereum payments

The smart contract takes action on Ethereum transactions.

Smart contracts will involve updating Ethereum wallets by transferring Ether.

A smart contract transfers ether between users, or Ethereum Addresses using special global variables.

Solidity provides a rich set of tools to facilitate the transfer of assets and Ether

Both Remix and Truffle provide simulated Ether to develop and test Ethereum payments.

Block and Transaction Properties

- `blockhash(uint blockNumber)` returns (bytes32): hash of the given block - only works for 256 most recent, excluding current, blocks
- `block.coinbase` (address payable): current block miner's address
- `block.difficulty` (uint): current block difficulty
- `block.gaslimit` (uint): current block gaslimit
- `block.number` (uint): current block number
- `block.timestamp` (uint): current block timestamp as seconds since Unix epoch
- `gasleft()` returns (uint256): remaining gas
- `msg.data` (bytes calldata): complete calldata
- `msg.sender` (address payable): sender of the message (current call)
- `msg.sig` (bytes4): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (uint): number of wei sent with the message
- `now` (uint): current block timestamp (alias for `block.timestamp`)
- `tx.gasprice` (uint): gas price of the transaction
- `tx.origin` (address payable): sender of the transaction (full call chain)

ABI Specification

Blockchain is built on hashing.

Solidity Contracts use the Application Binary Interface (ABI) specification.

ABI uses a hash function called Keccak256

A common use for ABI and Keccak256 is string comparison

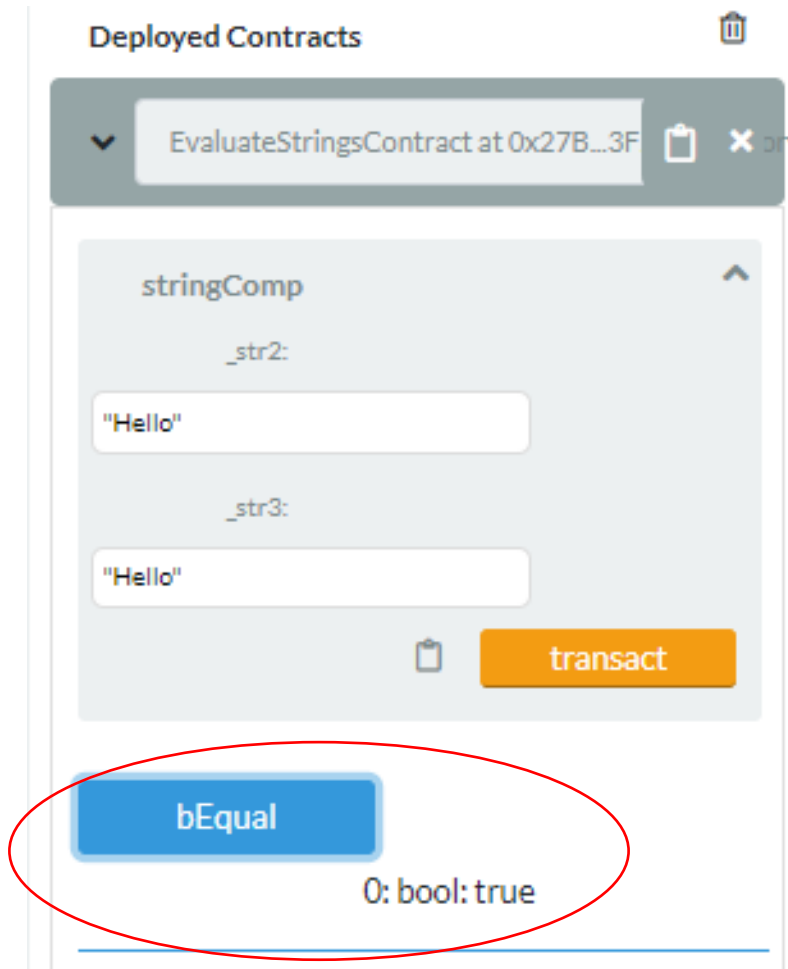
Solidity does not evaluate string equality.

Solidity will evaluate the string hashes and return equal, or not equal

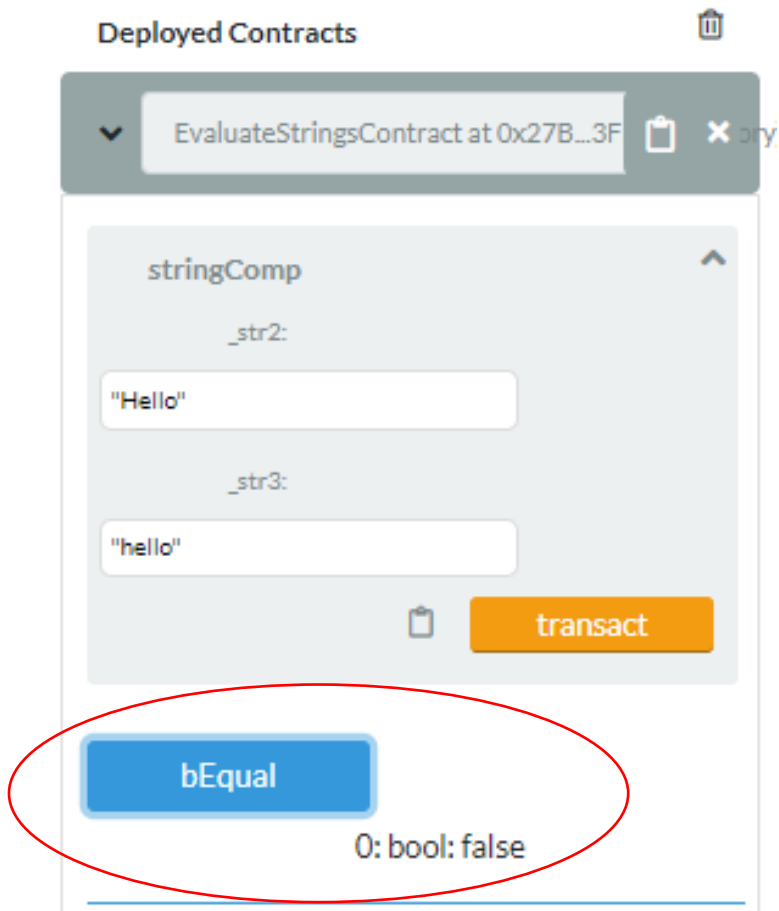
Introduction to Blockchain Development with Ethereum by Coding-Bootcamps.com
Session 7- Smart Contracts- Functions

```
1  pragma solidity >=0.5.0 <0.7.0;
2
3  contract EvaluateStringsContract {
4
5      bool public bEqual;
6      string public mystring1;
7      string public mystring2;
8
9
10
11  function stringComp(string memory _str2, string memory _str3) public returns(string memory _str0, string memory _str1) {
12
13      bEqual = compareStringsbyBytes(_str2, _str3);
14
15
16      if (bEqual == true){
17          _str0 = _str2;
18          _str1 = _str3;
19
20          mystring1 = _str0;
21          mystring2 = _str1;
22          return (mystring1, mystring2);
23      }
24
25  }
26
27
28
29  function compareStringsbyBytes(string memory s1, string memory s2) internal pure returns(bool){
30      return keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2));
31  }
32
33 }
```

String "Hello" is equal to string "Hello"



String "Hello" is not equal to string "hello"



Ethereum Time

The Ethereum Blockchain uses the epoch time, or Posix time.

Sometimes called Unix time, it is a common, maintained time from a specific point in time called the epoch.

Epoch time is the number of seconds since the epoch.

A good source of epoch time is the website: <https://www.epochconverter.com>.

The epoch can be converted to a date etc.

Function modifiers can be used to read epoch time and only execute within a time windows.

The current time can be return by using block.timestamp.

The current time can be compared to an epoch time in the future.

The modifier below will not allow the function to run until the block time is 1567612315 or later.

```
uint256 openingTime = 1567612315 ;  
modifier onlyWhileOpen() {  
    require (block.timestamp >= openingTime);  
    _;  
}  
  
function addUser(string memory _fname, string memory _lname) public onlyWhileOpen {  
    //adding internal function  
    addUser();  
    users[usrCounter] = User(usrCounter, _fname, _lname);  
}
```



Epoch & Unix Timestamp Conversion Tools

The current Unix epoch time is **1567968904**

Address Payable

Address Payable is a Solidity data type that can receive Ether.

Address Payable is used in when transaction results in payments made to other Ethereum addresses

```
1 pragma solidity ^0.5.1;  
2  
3 contract mySendReceiveEther {  
4  
5     //mapping is a key / value pair  
6     mapping(address => uint256 ) public balance;  
7     address payable mywallet1;  
8  
9     event Purchase (  
10         address _buyer,  
11         uint256 _amount  
12     );  
13 };
```

More Resources

To read more on blockchain and understand it in depth, the reading the following articles are highly recommended:

- History and Evolution of Blockchain Technology from Bitcoin
- Overview of Blockchain evolution and phases from Ethereum to Hyperledger
- Comprehensive overview and analysis of blockchain use cases in many industries
- Overview of blockchain technology and blockchain development

Also, the following are more tutorials and resources on Ethereum blockchain development.

- How to Write Ethereum Smart Contracts with Solidity in 1 hour
- How to Build Auction DApp with Ethereum and Solidity Programming Language
- How to Work with Ethereum Blockchain Applications through Remix IDE
- Certified Solidity Professional Certification exam
- Learn Ethereum: Build your own DApps with Ethereum and smart contracts book by Brian Wu