Part 1:

**Creating accounts on Avalanche and BSC**

This is the first part of the five-part Bridge project. In this project, you will be building a token bridge between two EVM-compatible blockchains (Ethereum Virtual Machine), specifically the Avalanche and BSC testnets.  
In later stages of this project you will be required to send transactions on these chains, which means that you’ll need tokens to pay the gas fees.

**Overview of Assignment**

In this assignment, you will use the [Python web3 library](https://web3py.readthedocs.io/en/stable/) to create a private key for an account(s) on both Avalanche and BSC.

Since we are working with two EVM chains, it’s enough to generate one account. The private key will resolve to the same address on the two networks.

Once you have created the account, you must get funds from **both** faucets sent to your new account

* [BSC Faucet](https://testnet.binance.org/faucet-smart)
* [Avalanche Faucet](https://faucet.avax.network/)

A faucet is a service where you can go to receive testnet funds. Since blockchains typically require the native token to update the blockchain, such as sending tokens or writing to the blockchain, the chain’s creator will often run a public faucet to give developers the tokens they need to use the testnet.  
Unfortunately, people are greedy, and faucets are often abused, so most faucets have a rate limit, as well as some kind of captcha to prevent users from writing bots to hit the faucet.  
Nevertheless, these are still somewhat unstable, and the fact that it can be difficult to get testnet funds has led to [markets for testnet funds](https://unchainedcrypto.com/goerli-testnet-eth-is-now-being-monetized/), of course, once there’s a market where you can buy testnet funds, there is also a market where you can  
sell testnet funds, and this makes it even attractive to attack the faucets.

You will need these funds in later assignments, so it is **important** that you save the private key(s) (or mnemonic(s)) associated with these accounts for later use. Feel free to periodically add more funds to your accounts from faucets.

**Creating accounts**

An account in Ethereum is an ECDSA private key (64 random bytes), and an account address, which is the 20 bytes derived from the ECDSA public key.

Both the [eth\_accounts](https://eth-account.readthedocs.io/en/stable/eth_account.html?highlight=recover_message" \l "module-eth_account.account) and [web3](https://web3py.readthedocs.io/en/stable/index.html) libraries provide tools for creating Ethereum accounts. They are documented in the eth\_accounts library, but work in both:

w3.eth.account.create()  
eth\_account.Account.create()

You’ll need to store the private keys (or mnemonics) for the accounts you generate, because you’ll need to be able to access the funds in these accounts after you get funds from the faucets.

In general, it is bad practice to store private keys in plaintext, but for the purposes of this assignment (where your testnet keys have no monetary value) it is acceptable (and recommended) that you simply write your private key(s) (or mnemonics) to a plaintext file which you can read later.

**Signing messages**

To sign a message using a private key, you can use the [sign\_message() function from web3](https://web3py.readthedocs.io/en/stable/web3.eth.account.html?highlight=encode" \l "sign-a-message).

Alternatively, you can use [sign\_message() from eth\_accounts](https://eth-account.readthedocs.io/en/stable/eth_account.html?highlight=sign" \l "eth_account.account.Account.sign_message)

You cannot use [web3.eth.sign()](https://web3py.readthedocs.io/en/stable/web3.eth.html?highlight=sign_message#web3.eth.Eth.sign) to sign a message, because that function is an interface to the key management built into your node (these are called [“hosted keys”](https://web3py.readthedocs.io/en/stable/web3.eth.account.html)). Since you are not running your own node, you must use the sign\_message function,  
as outlined [here](https://web3py.readthedocs.io/en/stable/web3.eth.account.html#sign-a-message)

**Verifying signatures**

To verify a signature, you can use  
[eth\_account.Account.recover\_message()](https://eth-account.readthedocs.io/en/stable/eth_account.html?highlight=recover_message#eth_account.account.Account.recover_message)

or w3.eth.account.recover\_message(), which has the same syntax

**Assignment**

Modify the file [gen\_keys.py](https://harvestprovide-switchdriver.codio.io/gen_keys.py) to complete the function "get\_keys()"

The autograder will call get\_keys() with a random challenge. Your function must sign the challenge, and return the signature as well as the address associated with the signature.

The autograder will check two things:

1. Does the signature verify using the address provided?
2. Does the address provided have a nonzero token balance on both BSC and Avalanche?

Part 2:

Current layout: 1 Panel with tree

**1. Bridge Contract (Destination Side)**

*1. Bridge Contract (Destination Side)*

**Overview**

This is the 2nd of 5 assignments in the token bridge project.

In this project, you will write the contract that controls the “wrapped” tokens on the destination side of the bridge.

**Wrapped assets**

When a user deposits “real” tokens on the source side of the bridge, the bridge operator will mint “bridge-wrapped” tokens on the destination side.

Note that this is how many bridges work in practice. For example, “real” USDC on Ethereum is controlled by [this contract](https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48) (created by Circle), and if you send USDC over the Avalanche Bridge, the Bridge will mint you “bridge-wrapped” USDC (called USDC.e) which is controlled by [this contract](https://snowtrace.io/token/0xa7d7079b0fead91f3e65f86e8915cb59c1a4c664) (created by the Avalanche Bridge) on the Avalanche C-chain. Meanwhile, the “real” USDC (on Ethereum) is held in custody by the [Avalanche bridge contract](https://etherscan.io/address/0x8eb8a3b98659cce290402893d0123abb75e3ab28) on Ethereum.

The Avalanche bridge has a different contract for each different bridged token, e.g., the USDC.e contract controls bridge-wrapped USDC, and the USDT.e contract controls bridge-wrapped USDT. Your bridge will do the same thing, deploying a new bridge-wrapped token contract for each new type of asset that is bridged. Although the *contract* for each of this bridge-wrapped assets is different, the underlying *code* is the same, i.e., they are all controlled by copies of the same contract. In this assignment, we have provided you with the contract that controls these bridge-wrapped tokens (

BridgeToken.sol

). You will **not** need to modify this file.

The BridgeToken is an [ERC20 token](https://ethereum.org/en/developers/docs/standards/tokens/erc-20/), but it also includes a few extra features that are not part of the standard ERC20 interface. The

BridgeToken.sol

 contract is based on the [standard ERC20 contract provided by OpenZeppelin](https://docs.openzeppelin.com/contracts/4.x/erc20).

The main difference is that the BridgeToken has an extra variable called “underlying” which gives the address (on the source chain) of the token that is being wrapped. The BridgeToken also allows the Destination contract to burn BridgeTokens at will.

**Assignment**

In this assignment you will complete the destination contract (

Destination.sol

)), which handles the deploying and minting of new BridgeTokens.

You will need to complete the three functions on the destination contract:

1. createToken()
2. wrap()
3. unwrap()

**createToken()**

The first time a user wishes to transfer an ERC20 (e.g. USDC) over the bridge, the owner of the destination contract will need create new BridgeToken instance on the destination chain.

Only “creator,” i.e., someone assigned the CREATOR\_ROLE on the destination contract should be allowed to call this function. This allows the bridge owner to control what type of assets are bridged, e.g. you could allow USDC to pass over your bridge, but not SHIB.

The function takes three arguments:

1. The address of the underlying asset (on the source chain)
2. The name of the underlying asset
3. The symbol of the underlying asset

When the createToken function is called, it will deploy a new BridgeToken contract, and return the address of the newly created contract.

The createToken function should emit a Creation event.

**wrap()**

When a user deposits tokens on the source chain, the bridge operator will call the wrap() function to mint them the correct BridgeToken on the destination chain.

Only a "warden", i.e., an address assigned the “WARDEN” role on the destination contract should be allowed to call the wrap() function

This function takes three arguments

1. The address of the underlying asset (on the source chain)
2. The address that will receive the newly wrapped tokens
3. The amount of tokens to mint

This function should lookup the BridgeToken that corresponds to the underlying asset, and mint the correct amount of BridgeTokens to the recipient.

This function must check that underlying asset has been “registered,” i.e., that the owner of the destination contract has called createToken on the underlying asset.

The wrap function should emit a Wrap event.

**unwrap()**

When a user wishes to return back across the bridge, they will burn their BridgeToken by calling unwrap()

This function takes three arguments

1. The address of BridgeToken that is being unwrapped (on the destination chain)
2. The address of the recipient of the underlying tokens (on the source chain)
3. The amount of tokens to burn

The unwrap() function should emit an Unwrap event.

Anyone should be able to unwrap BridgeTokens, but only tokens they own.

**Access control**

To prevent abuse, the bridge should only bridge “registered” tokens. But who is allowed to “register” a token?  
Similarly, the destination contract should only issue a “wrapped” token when there was a deposit on the source side. Since  
the deposit contract on the source chain cannot call the wrap() function on the destination chain directly, some intermediary must monitor the source chain and call the wrap() function on the destination chain. This intermediary is often called a “guardian” or a “warden.”

In principle, the authority who registers new tokens could also be the same as the warden who announces deposit events, but a basic security principle is that it’s good to separate roles (sometimes this is called the [Principle of Least Privilege](https://www.paloaltonetworks.com/cyberpedia/what-is-the-principle-of-least-privilege)).

OpenZeppelin provides the [AccessControl](https://docs.openzeppelin.com/contracts/2.x/access-control" \l "role-based-access-control), which makes it easy to define “roles” and restrict certain functions to certain roles.