**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?