Unit 19: Multi-Blockchain Wallet in Python

Background

Your new startup is focusing on building a portfolio management system that supports not only traditional assets like gold, silver, stocks, etc, but crypto-assets as well! The problem is, there are so many coins out there! It's a good thing you understand how HD wallets work, since you'll need to build out a system that can create them.

You're in a race to get to the market. There aren't as many tools available in Python for this sort of thing, yet. Thankfully, you've found a command line tool, `hd-wallet-derive` that supports not only BIP32, BIP39, and BIP44, but also supports non-standard derivation paths for the most popular wallets out there today! However, you need to integrate the script into your backend with your dear old friend, Python.

Once you've integrated this "universal" wallet, you can begin to manage billions of addresses across 300+ coins, giving you a serious edge against the competition.

In this assignment, however, you will only need to get 2 coins working: Ethereum and Bitcoin Testnet.
Ethereum keys are the same format on any network, so the Ethereum keys should work with your custom networks or testnets.

Dependencies

The following dependencies are required for this assignment and were likely already installed as part of your preparation for in-class activities.

> **Important:** If you have *not* already installed the dependencies listed below, you may do so by following the instructions found in the following guides:
>
> - HD Wallet Derive Installation Guide
> - Blockchain TX Installation Guide.

Dependencies List:

- PHP must be installed on your operating system.
- You will need to clone the [`hd-wallet-derive`](https://github.com/dan-da/hd-wallet-derive) tool.
- [`bit`](https://ofek.github.io/bit/) Python Bitcoin library.
- [`web3.py`](https://github.com/ethereum/web3.py) Python Ethereum library.

Instructions

1. Project setup

- Create a project directory called `wallet` and `cd` into it.
- Clone the `hd-wallet-derive` tool into this folder and install it using the HD Wallet Derive Installation Guide
- Create a symlink called `derive` for the `hd-wallet-derive/hd-wallet-derive.php` script. This will clean up the command needed to run the script in our code, as we can call `./derive` instead of `./hd-wallet-derive/hd-wallet-derive.php`:
    - Make sure you are in the top level project directory - in this case the directory named `wallet`.
    - **Mac Users:** Run the following command: `ln -s hd-wallet-derive/hd-wallet-derive.php derive`.
    - **Windows Users:** Creating symlinks is not supported by default on Windows, only reading them, so Windows users must perform the following steps:
        - Open up Git-Bash as an administrator (right-click on Git-Bash in the start menu).
        - Within `bash`, run the command `export MSYS=winsymlinks:nativestrict`.
        - Run the following command: `ln -s hd-wallet-derive/hd-wallet-derive.php derive`.
    - Test that you can run the `./derive` script properly, by running the following command.

        ```
        ./derive --
        key=xprv9zbB6Xchu2zRkf6jSEnH9vuy7tpBuq2njDRr9efSGBXSYr1QtN8QHRur28QLQvKRqFThCxopdS1UD61a5q
        6jGyuJPGLDV9XfYHQto72DAE8 --cols=path,address --coin=ZEC --numderive=3 -g
        ```

    - The output should match what you see below:

        ```
        +------+-------------------------------------+
        | path | address                             |
        +------+-------------------------------------+
        | m/0  | t1V1Qp41kbHn159hvVXZL5M1MmVDRe6EdpA |
        | m/1  | t1Tw6iqFY1g9dKeAqPDAncaUjha8cn9SZqX |
        | m/2  | t1VGTPzBSSYd27GF8p9rGKGdFuWekKRhug4 |
        +------+-------------------------------------+
        ```

- Create a file called `wallet.py` -- this will be your universal wallet script. You can use this starter code as a starting point.
- Your directory tree should look something like this:

2. Setup constants

- In a separate file, `constants.py`, set the following constants:
  - `BTC = 'btc'`
  - `ETH = 'eth'`
  - `BTCTEST = 'btc-test'`
- In `wallet.py`, import all constants: `from constants import *`
- Use these anytime you reference these strings, both in function calls, and in setting object keys.

3. Generate a Mnemonic

- Generate a **new** 12 word mnemonic using `hd-wallet-derive` or by using this tool.
- Set this mnemonic as an environment variable by storing it a an `.env` file and importing it into your `wallet.py`.

4. Derive the wallet keys

- Create a function called `derive_wallets` that does the following:
  - Use the `subprocess` library to create a shell command that calls the `./derive` script from Python. Make sure to properly wait for the process. **Windows Users** may need to prepend the `php` command in front of `./derive` like so: `php ./derive`.
  - The following flags must be passed into the shell command as variables:
    - Mnemonic (`--mnemonic`) must be set from an environment variable, or default to a test mnemonic
    - Coin (`--coin`)
    - Numderive (`--numderive`) to set number of child keys generated
    - Format (`--format=json`) to parse the output into a JSON object using `json.loads(output)`
- Create a dictionary object called `coins` that uses the `derive_wallets` function to derive `ETH` and `BTCTEST` wallets.
- When done properly, the final object should look something like this (there are only 3 children each in this image):

```
{'btc-test': [{'address': 'n45VqiDaT4R9WCe5fGMoW74ZsSaWWCgWyA',
               'index': 0,
               'path': "m/44'/1'/0'/0/0",
               'privkey': 'cRTceGbr1yp3XAPCUhmo1UsNByZ2rL5YPQrFgvEpWsPoyyo19qHD',
               'pubkey': '0220a6428ab158f2cd454c02057266d09222f891e6e9c3004965f5a6f90a882767',
               'pubkeyhash': 'f77a6940a979411119cccced6b681362c3db6a04',
               'xprv': 'tprv8jDpzBJzB7fMEXavmLRURKFjyPUFjUSoJTdcQMuVLP1zYZdjrAY5Bi4wgPz8RwZYz9PWFVXSHbioHPqNun7Dt5WK7MMoRqafR38aHwrYWeE',
               'xpub': 'tpubDFus8bMEKVM27zciez64piurYQzBtodhsmEPgswnkepPP3tWUZMfNCgorWJSsYHp7VuNpYfAiCA2dd1WvX6Pj5ZzAx1Gxt4TiU4J8XG8936'},
              {'address': 'n25obABzVfPwnxNEB24JFFq1waNZnPQwA9',
               'index': 1,
               'path': "m/44'/1'/0'/0/1",
               'privkey': 'cV5gDqwA4oY8oHPWwHkmvhzp41ABegnaGYQDhrsz3w3hijBDke7Q',
               'pubkey': '03ff6fdba3d1cb4c0bdc1b89a1acd0096488ec5dea5307182eb993562ee663aa51',
               'pubkeyhash': 'e198f9cc95b2d4ea3c49c9a69b71f47cbe609de0',
               'xprv': 'tprv8jDpzBJzB7fMGooj7iusbG6HKSCN8i4HJwJwCV2XhC33WhoKGDaRHVM3mLD42AhCX2ttsoct8My2CXpUdxSyYuHRRj9nsn4riBtry6hN4hM',
               'xpub': 'tpubDFus8bMEKVM2AGqX1NaTzfkPtTiJJ3FBtEuiV14q7TqSMC45tcQ1TyxuwVLe6zSp5JE8vbSKcheHE5DTar4ke3FnDyxFzmKruftBWyjKbYX'},
              {'address': 'mnabvJwhctVcM3jnw8R8Pd5JtQsYLTGtkD',
               'index': 2,
               'path': "m/44'/1'/0'/0/2",
               'privkey': 'cVaowifWnHBduhovoE4foXVxyNdG3G1vgM1779JRxWVGsfCTGuPo',
               'pubkey': '03056c49ec5d3fb0f00de33c5c49e45d1024cb3fc2a5741c5a78e11d356a25cc50',
               'pubkeyhash': '4d7989464a08d311401f398fbb1c81311853f8bd',
               'xprv': 'tprv8jDpzBJzB7fMK9LTxyPkQTqPeFZa6aNNqfErX64iajDLQ2Y2fnk46wGNFAVTQPfXZzb3WVUydbHswZTdh1kPKs1wLVLtNwTXMeFywKNr5va',
               'xpub': 'tpubDFus8bMEKVM2CcNFrd4LosVWDH5WFuZHQxqdoc72111jEWnoJBZeHRtERHcLPWxKyyHD4PhyKuCT3khfstRTWsuubZzzWjF9fY32mQHZu1r'}],
 'eth': [{'address': '0x14B367F42c68F9503C73853E75d0338036Defd59',
          'index': 0,
          'path': "m/44'/60'/0'/0/0",
          'privkey': '0x8fb83ee71a09a7e81d9e5196cbfe7ba927303e2a849ea51eed2132b697d3965d',
          'pubkey': '03be721ee6fcba56364ce80462aac058d622be93db58f540959ca87297b18a87bb',
          'pubkeyhash': '4b8bcca00e3fc78edf72f8a8065cfbde1216cb52',
          'xprv': 'xprvA4BLDCwe4amS6iCTF9cgoh6qYNNsMfQLx2VbYJL1oUiRf5RxFGHkB7YRkuwSVhsPkPF39U42iSky4pSXikPQQdn8MKUo7npqw9vv8Ua6tLU',
          'xpub': 'xpub6HAgciUXtxKjKCGvMB9hAq3a6QDMm88CKFRCLgjdMpFQXsm6nobziurucDGXfktWHvPPvFf3RMKfxuuyVr3GFugUxe5qvGFw6xYYTR25ry3'},
         {'address': '0xEc3aFB60C77E31f7e6905A6fEB6033550EF54Ed7',
          'index': 1,
          'path': "m/44'/60'/0'/0/1",
          'privkey': '0x3c3e2a5bbacdea9967fcf2ad13cdaf753d69c942fd09e57ec33a87a9ec70a080',
          'pubkey': '02d693f2b9224ebfab310c500b6d4f266733abd3cd4063d69806d13f4b62da42ab',
          'pubkeyhash': '8b7a500d8005e1a71a65175e4876a17faf2064d3',
          'xprv': 'xprvA4BLDCwe4amS8bVSywsFzbRcCfeGeCfdAoGfVANsQ3ywba7LvQVrkTKATy7BQ1vko4hQbCdNc2gEzJZWKdRupVmmpWz2WPRdxzpShsh8NXg',
          'xpub': 'xpub6HAgciUXtxKjM5Zv5yQGMjNLkhUm3fPUY2CGHYnUxPWvUNSVTwp7JFdeKFJv83yCKVA4dbA3LKK8AZm5qqndHvtGCgubsBVug8eRz3LpeYC'},
         {'address': '0x644070416c1e7A2b44FfF53219Ec7318628F18D0',
          'index': 2,
          'path': "m/44'/60'/0'/0/2",
          'privkey': '0x26d9a668dc28d4754c8a6de98d73a7c0c282afd6192744096603e469d9d8b21c',
          'pubkey': '02029530a9f83114b7ae9aca41f29ce96276a881ec2fe09c75cbe8d02509fc343f',
          'pubkeyhash': 'c158f33e1fa253d114ba8db5cdd2c383d6fe1318',
          'xprv': 'xprvA4BLDCwe4amSAFjVyzP2RxU3huouFoRewQpKTfZzmfs4GKwbhLyJ1iBoYVo1DzzJnShn9rSGVXupxGAZW4dxxDsSSH3faEtEB5sBgdbQRjX',
          'xpub': 'xpub6HAgciUXtxKjNjoy61v2o6QnFwePfG9WJdjvG3ycL1Q398GkEtHYZWWHPkYoGHetPSJjWwKSHixnRmF1tao6yGYTXZ84kfzAxFtLkoCFFjr'}]}
```


- You should now be able to select child accounts (and thus, private keys) by accessing items in the `coins` dictionary like so: `coins[COINTYPE][INDEX]['privkey']`.

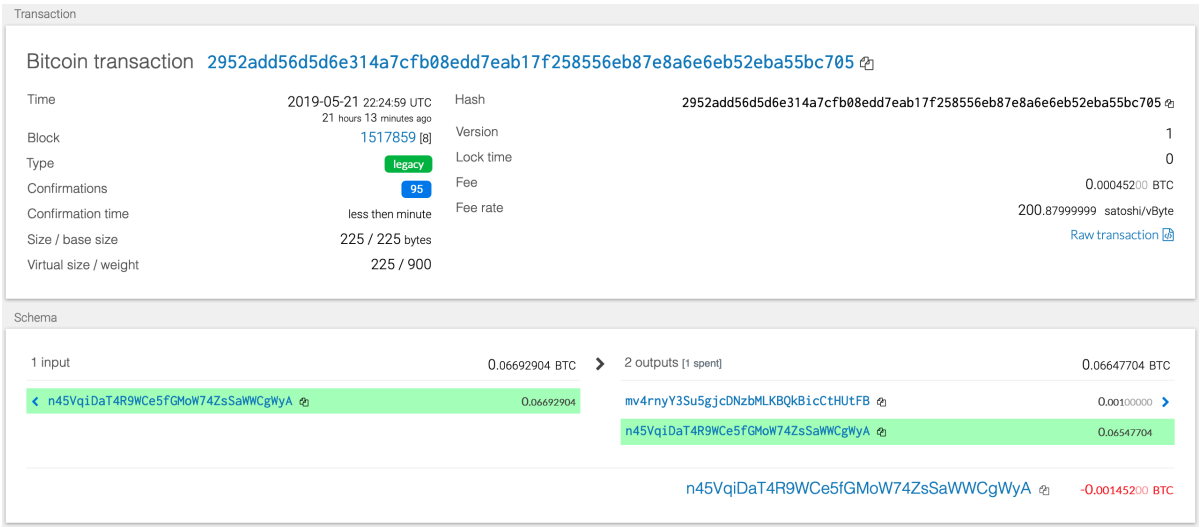
5. Linking the transaction signing libraries

- Use `bit` and `web3.py` to leverage the keys stored in the `coins` object by creating three more functions:
  - `priv_key_to_account`:
    - This function will convert the `privkey` string in a child key to an account object that `bit` or `web3.py` can use to transact.
    - This function needs the following parameters:
      - `coin` -- the coin type (defined in `constants.py`).
      - `priv_key` -- the `privkey` string will be passed through here.
    - You will need to check the coin type, then return one of the following functions based on the library:
      - For `ETH`, return `Account.privateKeyToAccount(priv_key)`
        - This function returns an account object from the private key string. You can read more about this object here.
      - For `BTCTEST`, return `PrivateKeyTestnet(priv_key)`
        - This is a function from the `bit` libarary that converts the private key string into a WIF (Wallet Import Format) object. WIF is a special format bitcoin uses to designate the types of keys it generates.
        - You can read more about this function here.
  - `create_tx`:
    - This function will create the raw, unsigned transaction that contains all metadata needed to transact.
    - This function needs the following parameters:

        - `coin` -- the coin type (defined in `constants.py`).
        - `account` -- the account object from `priv_key_to_account`.
        - `to` -- the recipient address.
        - `amount` -- the amount of the coin to send.
    - You will need to check the coin type, then return one of the following functions based on the library:
        - For `ETH`, return an object containing `to`, `from`, `value`, `gas`, `gasPrice`, `nonce`, and `chainID`. Make sure to calculate all of these values properly using `web3.py`!
        - For `BTCTEST`, return `PrivateKeyTestnet.prepare_transaction(account.address, [(to, amount, BTC)])`
- `send_tx`:
    - This function will call `create_tx`, sign the transaction, then send it to the designated network.
    - This function needs the following parameters:
        - `coin` -- the coin type (defined in `constants.py`).
        - `account` -- the account object from `priv_key_to_account`.
        - `to` -- the recipient address.
        - `amount` -- the amount of the coin to send.
    - You may notice these are the exact same parameters as `create_tx`. `send_tx` will call `create_tx`, so it needs all of this information available.
    - You will need to check the coin, then create a `raw_tx` object by calling `create_tx`. Then, you will need to sign the `raw_tx` using `bit` or `web3.py` (hint: the account objects have a sign transaction function within).
    - Once you've signed the transaction, you will need to send it to the designated blockchain network.
        - For `ETH`, return `w3.eth.sendRawTransaction(signed.rawTransaction)`
        - For `BTCTEST`, return `NetworkAPI.broadcast_tx_testnet(signed)`

6. Send some transactions!

- Now, you should be able to fund these wallets using testnet faucets.
- Open up a new terminal window inside of `wallet`.
- Then run the command `python` to open the Python shell.
- Within the Python shell, run the command `from wallet import *`. This will allow you to access the functions in `wallet.py` interactively.
- You'll need to set the account with `priv_key_to_account` and use `send_tx` to send transactions.
    - **Bitcoin Testnet transaction**
        - Fund a `BTCTEST` address using this testnet faucet.
        - Use a block explorer to watch transactions on the address.
        - Send a transaction to another testnet address (either one of your own, or the faucet's).
        - Screenshot the confirmation of the transaction like so:



- **Local PoA Ethereum transaction**
  - Add one of the `ETH` addresses to the pre-allocated accounts in your `networkname.json`.
  - Delete the `geth` folder in each node, then re-initialize using `geth --datadir nodeX init networkname.json`.
    This will create a new chain, and will pre-fund the new account.
  - Add the following middleware
    to `web3.py` to support the PoA algorithm:

```
from web3.middleware import geth_poa_middleware

w3.middleware_onion.inject(geth_poa_middleware, layer=0)
```

  - Due to a bug in `web3.py`, you will need to send a transaction or two with MyCrypto first, since the `w3.eth.generateGasPrice()` function does not work with an empty chain. You can use one of the `ETH` address `privkey`,
    or one of the `node` keystore files.
  - Send a transaction from the pre-funded address within the wallet to another, then copy the `txid` into MyCrypto's TX Status, and screenshot the successful transaction like so:

Status	SUCCESSFUL
TX Hash	`0x28120c78e5709475301f01525b9ac2e851e75419a9f302a22b2304572df6e923`
Block Number	`11`
From Address	`0x14B367F42c68F9503C73853E75d0338036Defd59`
To Address	`0xEc3aFB60C77E31f7e6905A6fEB6033550EF54Ed7`
Amount	`10000 ETH`
Gas Price	`20 Gwei`
Gas Limit	`21000`
Gas Used	`21000`
Transaction Fee	`0.00042 ETH`

7. Challenge Mode - OPTIONAL

- Add support for `BTC`.
- Add support for `LTC` using the sister library, [`lit`](https://github.com/blockterms/lit).
- Add a function to track transaction status by `txid`.

Submission

- Create a `README.md` that contains the test transaction screenshots, as well as the code used to send them. Pair the screenshot with the line(s) of code.
- Write a short description about what the wallet does, what it's built with, and how to use it.
- Include installing pip dependencies using `requirements.txt`, as well as cloning and installing `hd-wallet-derive`. You may include the `hd-wallet-derive` folder in your repo, but still include the install instructions. You do not need to include Python or PHP installation instructions.
- Upload the project to a new GitHub repository.
- Celebrate the fact that you now have an incredibly powerful wallet that you can expand to hundreds of coins!

Requirements

Project Setup (30 points)

To receive all points, your code must:

- Create a project directory. (5 points)
- Create a separate file to contain the constants. (8 points)
- Generate a mnemonic phrase. (8 points)
- Derive the wallet keys. (9 points)

Transactions (40 points)

To receive all points, your code must:

- Link the transactions with the signing libraries. (10 points)
- Fund the wallet, send transactions, and add screenshots of the transactions to your ReadME.md. (10 points)
- Fund a BTCTEST address, send transactions, and add screenshots of the transactions to your ReadME.md. (10 points)
- Send transactions using the local PoA Ethereum network, and add screenshots of the transactions to your ReadME.md. (10 points)

Coding Conventions and Formatting (10 points)

To receive all points, your code must:

- Place imports at the beginning of the file, just after any module comments and docstrings and before module globals and constants. (3 points)
- Name functions and variables with lowercase characters and with words separated by underscores. (2 points)
- Follow Don't Repeat Yourself (DRY) principles by creating maintainable and reusable code. (3 points)
- Use concise logic and creative engineering where possible. (2 points)

Deployment and Submission (10 points)

To receive all points, you must:

- Submit a link to a GitHub repository that's cloned to your local machine and contains your files. (5 points)
- Include appropriate commit messages in your files. (5 points)

Code Comments (10 points)

To receive all points, your code must:

- Be well commented with concise, relevant notes that other developers can understand. (10 points)
