**1.Representing Transactions**

In this lesson, you'll build a small blockchain of your own in Python! This lesson assumes a familiarity with Python syntax, functions, loops, importing libraries, and constructing classes, but there are some hints along the way to help out. If you've never used Python before, you might want to [learn some with Codecademy](https://www.codecademy.com/learn/learn-python).

The blockchain is a new way of storing and moving data securely. The data mostly consists of transactions which include messages exchanged between two parties. Before we start creating our blockchain, let's think of a way to store a transaction like the one shown below:

amount: 30

sender: Alice

receiver: Bob

In this example, Alice is trying to transfer 30 units of some currency to Bob. Can you think of a Python data type to best represent the above transaction?

This transaction is best represented in the form of a Python *dictionary*, with keys for the required fields and values specific to the transaction.

These transactions are all stored inside the *mempool*, a pool of transactions that miners reference when selecting the set of transactions they want to verify.

**Instructions**

**1.**

Let's create a transaction and add it to the *mempool*. Name the new transaction my\_transaction and add amount, sender, and receiver as key-values.

Hint

The syntax for creating a dictionary in Python is dictionary = { }

**2.**

Add my\_transaction to the mempool list.

Hint

Syntax for appending an item to a Python list is as follows:

list.append(element)

**3.**

Create a new list called block\_transactions and add three transactions from the mempool list. This will allow us to prepare the transactions to be added to our future Block structure

----------------------------------------------------------

# 2. Creating Blocks

Now let's think of a way to represent a block in Python. We could create a bigger dictionary and store our data inside this dictionary. But since blocks can be represented as objects, let's create a Block Class which we can easily use to create new blocks.

Recall that a Block contains the following properties:

* Timestamp
* Transaction
* Hash
* Previous Hash
* Nonce

In this exercise, we will be creating the default constructor for the Block class in our Mini-Blockchain.

**Instructions**

**1.**

Every Block in the blockchain has a timestamp associated with it. In order to dynamically generate a timestamp, we must import a Python module that returns the current date and time.

Import the datetime module from the datetime library.

Stuck? Get a hint

**2.**

Inside the [datetime module](https://docs.python.org/2/library/datetime.html) there is a .now() method that returns the current date and time.

Call the datetime module's .now()method to print out the current date and time.

Stuck? Get a hint

**3.**

Now let's work on creating our Block. We will be passing transactions and previous\_hash to the default constructor each time a Block is created.

Complete the \_\_init\_\_() method inside the Block class by initializing the following instance variables:

* transactions
* previous\_hash
* nonce (with a default value of 0).

Stuck? Get a hint

**4.**

Inside the \_\_init\_\_() method, create a timestamp instance variable that stores the current date and time.

Hint

Call the .now() method from the datetimemodule and store the result in timestamp.

# 3. Hashing and SHA-256

Before we create a more dynamic blockchain, let’s learn how to use a hash function in Python. Specifically, we will be using the [SHA-256 hash function](https://docs.python.org/2/library/hashlib.html) which can be easily imported in Python.

We will use the SHA-256 as a regular function that takes in a random string as its argument. To properly use this function in Python 3, our string must be encoded before being passed as an argument. To encode the string, we use the .encode() method.

**Instructions**

**1.**

Import sha256 from the hashlib Python library.

**2.**

Create a variable called text. Initialize the variable with this string I am excited to learn about blockchain.

**3.**

Create a sha256 hash object, using the constructor sha256() and pass the textvariable as its argument. Assign the value of this object to a variable called hash\_result.

Be sure to use the .encode() method on the text variable.

Stuck? Get a hint

**4.**

Call the .hexdigest() method on hash\_result and print the result.

Stuck? Get a hint

**5.**

Modify the text variable by adding an exclamation mark at the end and running your code.

Notice how this hash is completely different from the last one?

# 4. Generating Block Hashes

Block hashes are used to uniquely identify and maintain the integrity of each block. The SHA-256 algorithm is used to generate the hash of the block using the timestamp, data, and previous hash — the three properties of our Block class!

Let’s create the .generate\_hash() method for the Block class.

**Instructions**

**1.**

In the .generate\_hash() method, create the variable block\_contents, which combines the Block timestamp, transactions, nonce, and previous hash.

Wrap each item in the str() method in order to convert them to a string type to prepare for hashing.

Return the result.

Hint

To combine strings together the following syntax is used:

string = sub\_string1 + sub\_string2 + sub\_string3

**2.**

Create a variable block\_hash.

Create a new hash with sha256() and block\_contents and save the value to block\_hash.

Remember to use the .encode() method to encode the string.

Update the method to return block\_hash.

**3.**

Return the text hash value by calling the hexdigest() method on block\_hash.

**4.**

Uncomment the line in \_\_init\_\_() that calls the generate\_hash() function to complete the Block class

**5. Creating the Blockchain Class**

Each computer participant has their own copy of the blockchain. Ideally, each copy of the blockchain should have the same properties and functionality to add and validate blocks.

We can represent the blockchain as an object. We are using objects for our implementation, because they offer the flexibility to create specific attributes and methods. Representing it as an object also allows us to create blockchain instances for each computer participant.

To review, our blockchain contains the following:

* **Chain:** A list that that holds all the blocks inside the blockchain.
* **Unverified Transactions:** A list that represents all the unverified transactions before being passed into a block.
* **Genesis Block:** A block automatically generated when the blockchain is initialized.

**Instructions**

**1.**

Fill in the \_\_init\_\_() method inside the Blockchain class by initializing instance variables chain and all\_transactions as empty Python lists.

**2.**

Complete the method genesis\_block by instantiating a new Block object with an empty transactions list and a previous hash of 0.

Append the resulting block to the chain.

Hint

* The syntax to instantiate a new Block object is Block(transactions, previous\_hash).
* To access the Blockchain chain use self.chain.

**3.**

Since we want to automatically create a Genesis Block when a new blockchain object is created, call the method .genesis\_block() inside the \_\_init\_\_() method.

Hint

The right way to initialize a genesis block is by writing self.genesis\_block()

**6. Adding Blocks to the Blockchain**

Now that we have everything in place, let’s begin adding blocks to the blockchain.

**Instructions**

**1.**

Complete the function add\_block().

To do this, create a variable named new\_block that takes in a transaction and the previous\_block‘s hash. Append new\_blockto the end of the chain.

Hint

The proper way to access a specific block’s hash from the chain is through this line of code: self.chain[index].hash.

* The previous block can be accessed by self.chain[len(self.chain)-1].
* Your resulting code should be as follows: Block(transactions, previous\_block\_hash)

# 7. Checking for a Broken Chain

Hashing helps to maintain the integrity of the blockchain. In this exercise, we will see this in action. Let’s write a .validate\_chain() method that checks to see if blocks are linked to each other properly.

In order to validate the entire blockchain, we must loop through each of the blocks stored inside the blockchain itself. Then, we will check through each of the blocks to ensure that the previous hash value matches with the hash value inside our previous block.

**Instructions**

**1.**

In the .validate\_chain() method, create a for loop with the loop variable i and traverse through the entire length of self.chain. Be sure to start at index 1 instead of index 0.

Inside the for loop, create a variable current and assign it to the current block being indexed with i. Create another variable previous and assign it to the previous block using index i-1.

Hint

Loop through the correct part of the chain with the following snippet:

for i in range(1, len(self.chain)):

**2.**

Verify that the hash of the current block is NOT equal to the value the current block generates via the generate\_hash()method. If this condition is true, then the blockchain is not valid, therefore we should return False.

**3.**

Verify that the hash of the previous hash of the current block is NOT equal to the value generated over the previous block using the generate\_hash() method. If this condition is true, then the blockchain is not valid, therefore we should return False.

**4.**

If the above conditions are not satisfied, then the blockchain is valid! Return True outside the for loop.

# 8. Hacking the Chain

Now we can use the code in our previous exercise to spot a broken link. Let’s try tampering with the contents of the block and see how that creates a mismatch between hash values.

**Instructions**

**1.**

Instantiate a new Blockchain object called my\_blockchain.

**2.**

Add a new block to my\_blockchain and pass in new\_transactionsas the argument.

Print out the contents of my\_blockchain to see the new block!

Hint

Remember, the .add\_block() method is used to add a block. This method has new\_transactions as its parameter. Similarly, the print\_blocks() method is used to print the contents of a Blockchain object.

**3.**

Select the transactions found in my\_blockchain‘s chainattribute. Set the transactions variable of Block 1 to the string "fake\_transactions".

Hint

Use my\_blockchain.chain[1] to grab Block 1 on the blockchain. You can access this block’s transactions using the transactions attribute.

**4.**

Now let’s check if the blockchain is still valid by calling the correct method on my\_blockchain!

Hint

Remember, the validate\_chain() method is used to validate a Blockchain object.

# 9. Nonce and Proof-of-Work

Let’s review the concepts of nonce and proof of work. In this exercise, we will implement an example that demonstrates the difficulty of the math problem that helps protect the blockchain from potential attackers.

**Instructions**

**1.**

Import the sha256 hash function from the Python hashlib library

**2.**

Create a variable called difficulty and assign it a value of 2. This sets the number of leading zeros that the hash we find must have.

Create another variable called nonce and assign it to a value of 0. This will be our default starting value.

**3.**

Using the .str() method, cast nonce and new\_transactions into strings. Pass the two strings into the sha256 function.

Store the resulting hash value into a variable called proof and print it out!

**Note:** Use the .hexdigest() method over the resulting sha256 function to properly store the hash value.

Stuck? Get a hint

**4.**

Come up with some code that increments the nonce value until the generated hash has difficulty number of leading zeros. Once the desired proof has been found, store it in a variable called final\_proof and print it out to see the correct hash!

Hint

Think about using a while loop to keep on incrementing the nonce value until the following condition is True.

hash\_value[:2] != '0'\*difficulty

# 10. Implementing Proof-of-Work

Now that we’ve seen a simple example of Proof-of-Work, let’s integrate it into our blockchain! Complete the proof\_of\_work() method inside the Blockchain class.

**Instructions**

**1.**

Inside the .proof\_of\_work() method, create a local variable proof and assign it the block’s hash.

Hint

Call the generate\_hash() method over the Blockobject to retrieve its hash.

**2.**

Finish the rest of the method by creating a loop that increments the nonce value until the hash with the required difficulty has been generated.

After finding the correct hash, set the value of the block.nonce back to 0 and return the correct hash outside of the loop.

Hint

The variable difficulty refers to the number of leading zeros required in the hash.

Try using a loop to automatically increase the nonce value as it generates new block hashes.

# 11. Adding Blocks to the Chain Securely

Now that we have implemented our Proof-of-Work method, we can now work on adding new blocks securely.

**Instructions**

**1.**

In the .add\_block() method, calculate the proof\_of\_work for the new\_block. Assign the calculated proof\_of\_work to a variable named proof before appending the new\_block to the blockchain.

Return, in order, the calculated proof and the new\_block itself.

Hint

The proper way to call the Proof-of-Work method is the following: .proof\_of\_work(block\_name).

Remember that you can return two values from a function by separating them with a comma: return value\_one, value\_two.

# 12. Blockchain Summary

Congratulations! You have completed all the steps required to build a basic blockchain! In this exercise, we will bring the key parts together to review what we have built so far.

\*Note: \* The blockchain we have built only exists on a local machine. It is important to know that actual blockchain applications operate on multiple computers in a decentralized manner.

**Instructions**

**1.**

Create a Blockchain object named local\_blockchain. Verify that this automatically creates a Genesis Block by printing out the contents of local\_blockchain.

Hint

Use the .print\_blocks() method to print out the contents of the blockchain.

**2.**

Individually add block\_one\_transactions, block\_two\_transactions, and block\_three\_transactions respectively into local\_blockchain. Print out the contents of local\_blockchain to see what the block holds.

Hint

Use the method .add\_block(transactions) to add blocks that contain the required transactions.

**3.**

Modify the second block you added in local\_blockchain by changing the block’s transactions to fake\_transactions. Check to see if the blockchain is still valid using the correct method.

Hint

You can access the block by indexing into local\_blockchain.chain. You can validate the block using the .validate\_chain() method.