Blockchain Program Documentation

Cryptographic Concepts CSI2108

Calvin Tate | 10443858 | Due Date: 14/05/2020

Contents

[Introduction 3](#_Toc40348990)

[Transaction Recording Program 3](#_Toc40348991)

[Flow of Execution 3](#_Toc40348992)

[Transact Function 4](#_Toc40348993)

[Blocking Program 6](#_Toc40348994)

[Flow of Execution 6](#_Toc40348995)

[Block Function 8](#_Toc40348996)

[Verify Blockchain Function 10](#_Toc40348997)

[WatchDir Function 10](#_Toc40348998)

[Sha Hash Function 10](#_Toc40348999)

[Nonce Function 10](#_Toc40349000)

# Introduction

The purpose of this documentation is to provide an explanation of the logic behind the blockchain system I have created for Case Study: Part 3. This will be done by going through the flow of execution and explaining what each function performs at each function call. First, we will look at the transaction recording program and then move on to the blocking program. The blockchain system has two different programs – the transaction recording program and the blocking program. We will start first with the transaction recording program.

# Launching the Programs

The folders “blockchain” and “transactions” are created by the respective programs and the sequence for launching the programs does not matter.

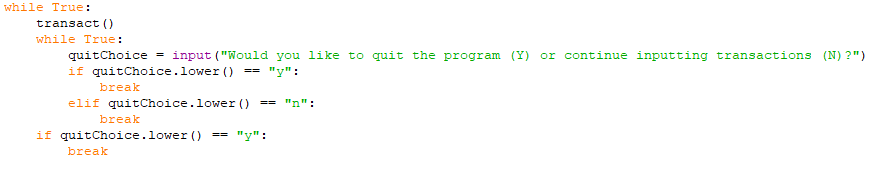
# Transaction Recording Program

## Flow of Execution

The transaction recording program begins with the following if statement that uses the os module’s .path.isdir() function to check if the “transactions/” file exists. If the file does not exist, then the os.mkdir() will create a file called “transactions”.



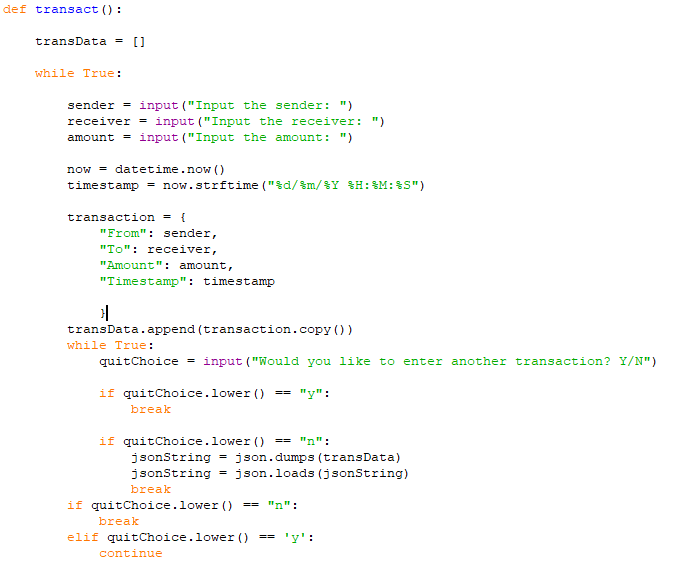
Following this is a while loop statement. The first statement within this loop is a call to the transact function. This function is explained in the following section [*Transact Function*](#_Transact_Function)*.* Following the execution of the transact function, another while loop is entered with the purpose of asking the user if they would like to continue or quit the program. The while loop is to ensure that the answer is either ‘y’ or ‘n’. If the user chooses to quit the program, program execution stops via the break.



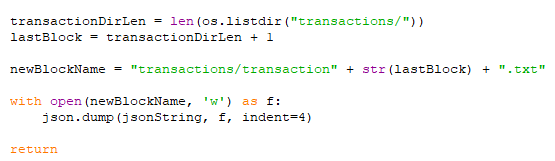
## Transact Function

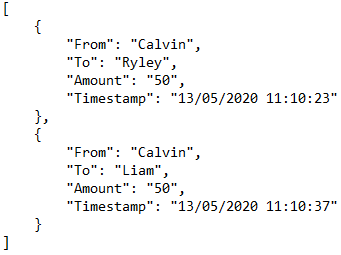
The first section of the transact function consists of a while loop that prompts the user for transaction data. First, an empty transData list is created to hold the transaction data. A while loop is then entered that repeatedly prompts for the sender, receiver and amount of the transaction and then stores this information into a transaction dictionary. The timestamp is also generated using the time module datetime.now() with the format day/month/year hour:minute:second.

After each transaction input, the transaction dictionary is appended to the transData list and the execution enters a while loop that asks if the user would like to enter another transaction. If the user enters ‘n’, the transData list is then converted to a JSON string using the dumps and loads functions from the JSON module. Using the functions together allowed for “pretty print” JSON seen in figure 1. The if and elif statements after the loop control whether another transaction is inputted based on the user choice. If the loop is broken, the next section of the function executes.

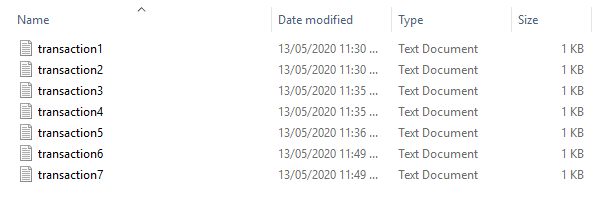


In this blockchain system, the transactions recorded previously will form a “transaction block” which are stored in their own files – starting at “transaction1”. This second section of the transact function first finds how many files are in the transactions folder using len() with os.listdir(). Adding 1 to this number gives the number of the new “transaction block”. This new number is concatenated into a directory string stored in newBlockName. This directory is then opened and the text file is created and the transaction data converted to a JSON string earlier is stored in the text file. At this point the function returns execution and the while loop explained before. Figure 2 is an example of what the transaction folder will look like.





Example of a transaction block in JSON file, Figure 1



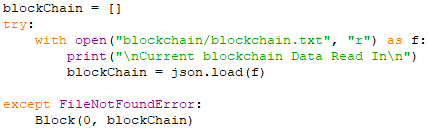
Example of the transaction folder, Figure 2

# Blocking Program

## Flow of Execution

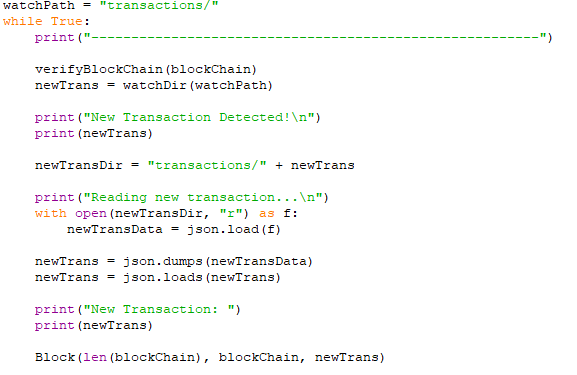
Execution of the blocking program begins with the initialisation of an empty list called blockChain. This is followed by a try block that attempts to open the blockchain file. If successful, json.load() is used to load in the JSON data from a file and returns a dictionary. This dictionary is then placed into the blockChain list.

If the try block throws a FileNotFoundError exception, then the Block function is called with an index of 0. This prompts the Block function to create the blockchain directory/file and place the first block inside. This is explained further in the [*Block Function*](#_Block_Function)section.



This try block is followed by the main loop of the blocking program. The loop first calls the verifyBlockChain function which goes through the block chain list to verify the previous hash values. This function is explained further in the [*Verify Blockchain Function*](#_Verify_Blockchain_Function)section. The next statement is an assignment of the variable newTrans with the return of the call to the watchDir function. This function simply waits and watches the transaction folder for any new transaction blocks. When a new transaction is seen, the name of the new file is returned to the newTrans variable. This function is explained further in the [*Watch Dir*](#_WatchDir_Function)section.

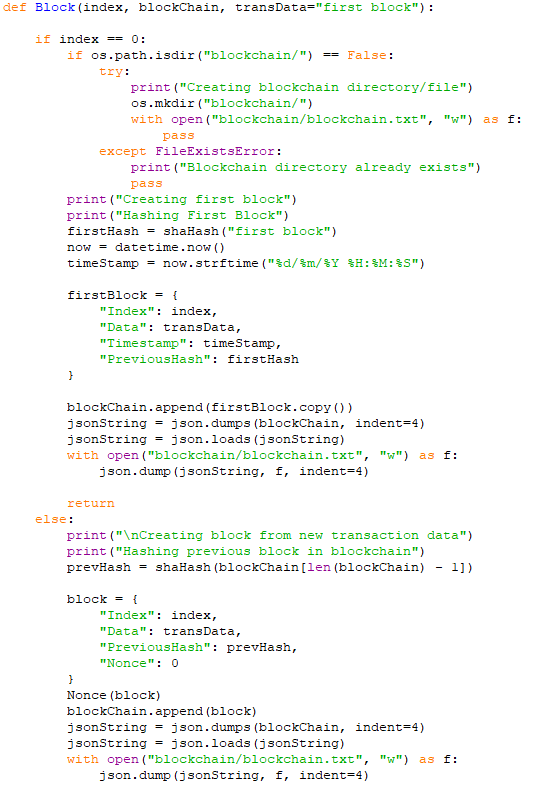
Once a new transaction is detected, the transaction file is then opened and the new transaction data is loaded into the newTransData variable. This variable is then parsed by the dumps and loads functions to create a dictionary. At this point the end of the loop is reached and the block function is called with the index being the length of the blockChain list, the blockChain list itself and the newTrans transaction data. Once the block function completes execution, the blockchain file will be updated with a new block and the program will begin waiting for a new transaction.



## Block Function

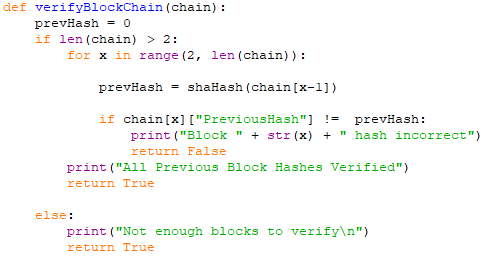
The block function is split into two distinct sections of code that are determined by an if/else evaluation of the index provided in the parameters. If the index is equal to 0, then this indicates that the first block must be generated and placed into a blockchain file. When creating the first block, the function checks to see if the “blockchain” directory exists. If it does not exist, it is created along with the blockchain file. The function then generates a SHA256 hash (using the shaHash function explained in the [*Sha Hash Function*](#_Sha_Hash_Function) section) of the string “first block” and generates the timestamp. The hash and timestamp, along with the index (0) and the default of the parameter transData (“first block”) are placed into a dictionary called firstBlock. This firstBlock dictionary is then appended to the blockChain list. The blockChain list is then parsed into a JSON string using the dumps and loads functions, with the block then being written to the blockchain file. At his point the function returns.

However, if the index parameter equals anything other than 0, then a block other than the first block is generated using the provided parameters. First, the hash of the last block in the blockChain list is generated using the shaHash function. The block dictionary is then assigned the index and transData parameters along with the prevHash variable. The nonce function is then called with the block dictionary as a parameter. The purpose of this function is to provide a proof of work known as the nonce number. The nonce number is a number that when added to the block, causes the blocks hash to have a total of 14 zeros within the string. To find this number, the nonce function utilises a brute force technique and is explained further in the [*Nonce Function*](#_Nonce_Function)section. After the nonce number is found, the new block is added to the blockChain list and is then also parsed to a JSON string and written to the blockchain file.



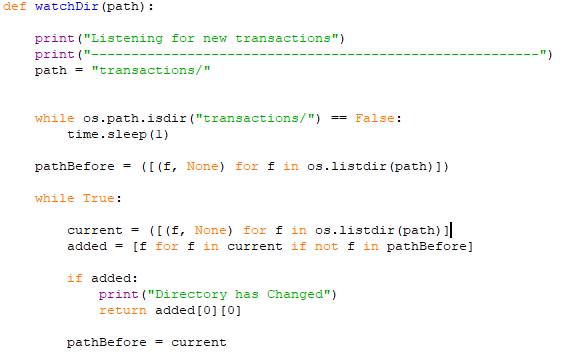
## Verify Blockchain Function

This function is not part of that assignment requirements but was created to make verifying the blockchain system easier and quicker. The function first checks the number of blocks in the chain provided. If two or less, then the function does not execute. If greater than two then a for loop executes over the range of the blockchain, starting at 2. The prevHash variable is given the hash of the previous block (chain[x-1]) using the shaHash function. An if statement then checks whether the current block’s previous hash (chain[x][“PreviousHash”]) is equal to the hash of the previous block generated just before. If the two hashes are not equal to each other, then the function returns False and the index of the block containing the incorrect hash is printed. Else if no mismatches are found, then True is returned.



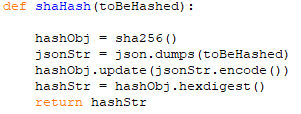
## WatchDir Function

The purpose of this function is to monitor the “transactions” folder for new transactions. The first while loop executes only when os.path.isdir(“transactions”) evaluates to False. This means that the transaction recording programs has not been launched and hasn’t had a chance to create the directory. The loop uses time.sleep to pause execution for one second, repeating until the evaluation becomes True (“transactions” folder created). At this point the loop exits and a pathBefore list is populated with the names of all the files within the directory. This is achieved using a list comprehension that grabs the name of each file that is returned by os.listdir. The purpose is to keep a record of what the director looked like before entering the while loop. Within this loop, there are two more list comprehensions. The current list is populated again with what the directory currently contains. The added list comprehension lists the files that are contained in the current list but not within the pathBefore list. If any file name is appended to the added list, then the if statement will return the name of that file.



## Sha Hash Function

The shaHash function is the smallest and simplest function within the programs yet is really helpful. It simply takes an object to be hashed and produces its sha256 hash. First a sha256 hash object is instantiated to hashObj and the object toBeHashed is parsed to a JSON string. The hashObj is then updated with the encoded JSON string. Finally, the hashStr variable is given the hash string of hashObj using the hexdigest function which is then returned.



## Nonce Function

The Nonce function is the function that performs the proof of work calculation – producing a nonce number. The function begins by initialising nonce to 0 and enters a while loop that executes while nonce is less than or equal to 50,000. The loop initialises nonceCount to 0 and sets the ‘Nonce’ key of the blockDict to nonce. A hash of the blockDict is then generated and placed in hashOfBlockStr. The hash string is then iterated over, with each character being checked whether it is equal to 0. If this evaluates to true, then nonceCount is incremented.

Once the string is fully iterated, another if statement checks if nonceCount is equal to 14. If this is false, then nonceCount is reset and the nonce number is incremented. Again, this nonce number is placed into the ‘Nonce’ value of blockDict, the hash is regenerated and the string iteration begins again. If this is true, then the nonce number has been found and the function returns as the blockDict object has already been given the nonce number. If no nonce number is found, then the nonce is settled to 50,000.

