Python Blockchain

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Blockchain is the current buzz that is dominating the software development trends. The development and designing of Blockchain involves three major components: client, miner and blockchain.

This tutorial is aimed to give you a crisp understanding of the process of building your own blockchain.

Audience

Any programming enthusiast who wants to keep in pace with the recent trend of Blockchain development can gain from this tutorial.

If you are a learner interested to learn the basics of Blockchain Development, this tutorial aptly suits your needs.

Prerequisites

This tutorial is written assuming that the learner has an idea on programming in Python and a basic idea on Blockchain. If you are new to any of these concepts, we suggest you to pick tutorials based on these concepts first before you plunge into this tutorial.

Copyright & Disclaimer

Table of Contents

1. Python Block chain – Introduction

In the tutorial on Blockchain, we have learnt in detail about the theory behind blockchain. The blockchain is the fundamental building block behind the world's most popular digital currency Bitcoin. The tutorial deeply dealt with the intricacies of Bitcoin explaining fully the blockchain architecture. The next step is to build our own blockchain.

Satoshi Nakamoto created the first virtual currency in the world called Bitcoin. Looking at the success of Bitcoin, many others created their own virtual currencies. To name a few - Litecoin, Zcash, and so on.

Now, you may also like to launch your own currency. Let us call this as TPCoin (TutorialsPoint Coin). You will write a blockchain to record all transactions that deal with TPCoin. The TPCoin can be used for buying Pizzas, Burgers, Salads, etc. There may be other service providers who would join your network and start accepting TPCoin as the currency for giving out their services. The possibilities are endless.

In this tutorial, let us understand how to construct such a system and launch your own digital currency in the market.

Components Involved in Blockchain Project Development

The entire blockchain project development consists of three major components:

- Client
- Miners
- Blockchain

Client

The Client is the one who will buy goods from other vendors. The client himself may become a vendor and will accept money from others against the goods he supplies. We assume here that the client can both be a supplier and a recipient of TPCoins. Thus, we will create a client class in our code that has the ability to send and receive money.

Miner

The Miner is the one who picks up the transactions from a transaction pool and assembles them in a block. The miner has to provide a valid proof-of-work to get the mining reward. All the money that miner collects as a fee will be for him to keep. He may spend that money on buying goods or services from other registered vendors on the network, just the way a Client described above does.

Blockchain

Finally, a Blockchain is a data structure that chains all the mined blocks in a chronological order. This chain is immutable and thus temper-proof.

You may follow this tutorial by typing out the code presented in each step in a new Jupyter notebook. Alternatively, you may download the entire Jupyter notebook from https://www.anaconda.com/.

In the next chapter, we will develop a client that uses our blockchain system.

2. Python Block chain – Developing Client

A client is somebody who holds TPCoins and transacts those for goods/services from other vendors on the network including his own. We should define a **Client** class for this purpose. To create a globally unique identification for the client, we use PKI (Public Key Infrastructure). In this chapter, let us talk about this in detail.

The client should be able to send money from his wallet to another known person. Similarly, the client should be able to accept money from a third party. For spending money, the client would create a transaction specifying the sender's name and the amount to be paid. For receiving money, the client will provide his identity to the third party - essentially a sender of the money. We do not store the balance amount of money the client holds in his wallet. During a transaction, we will compute the actual balance to ensure that the client has sufficient balance to make the payment.

To develop the **Client** class and for the rest of the code in the project, we will need to import many Python libraries. These are listed below:

```
# import libraries
import hashlib
import random
import string
import json
import binascii
import numpy as np
import pandas as pd
import pylab as pl
import logging
import datetime
import collections
```

In addition to the above standard libraries, we are going to sign our transactions, create hash of the objects, etc. For this, you will need to import the following libraries:

```
# following imports are required by PKI
import Crypto
import Crypto.Random
from Crypto.Hash import SHA
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
```

In the next chapter, let us talk about client class.

3. Python Block Chain – Client Class

The **Client** class generates the **private** and **public** keys by using the built-in Python **RSA** algorithm. The interested reader may refer to this tutorial for the implementation of RSA. During the object initialization, we create *private* and *public* keys and store their values in the instance variable.

```
self._private_key = RSA.generate(1024, random)
self._public_key = self._private_key.publickey()
```

Note that you should never lose your private key. For record keeping, the generated private key may be copied on a secured external storage or you may simply write down the ASCII representation of it on a piece of paper.

The generated **public** key will be used as the client's identity. For this, we define a property called **identity** that returns the HEX representation of the *public* key.

```
@property
    def identity(self):
        return
binascii.hexlify(self._public_key.exportKey(format='DER'))
.decode('ascii')
```

The **identity** is unique to each client and can be made publicly available. Anybody would be able to send virtual currency to you using this **identity** and it will get added to your wallet.

The full code for the **Client** class is shown here:

```
class Client:

    def __init__(self):
        random = Crypto.Random.new().read
        self._private_key = RSA.generate(1024, random)
        self._public_key = self._private_key.publickey()
        self._signer = PKCS1_v1_5.new(self._private_key)

    @property
    def identity(self):
        return
binascii.hexlify(self._public_key.exportKey(format='DER')).decode('ascii
')
```

Testing Client

Now, we will write code that will illustrate how to use the **Client** class:

```
Dinesh = Client()

print (Dinesh.identity)
```

The above code creates an instance of **Client** and assigns it to the variable **Dinesh**. We print the public key of **Dinesh** by calling its **identity** method. The output is shown here:

```
30819f300d06092a864886f70d010101050003818d0030818902818100b547fafceeb131e07
0166a6b23fec473cce22c3f55c35ce535b31d4c74754fecd820aa94c1166643a49ea5f49f72
3181ff943eb3fdc5b2cb2db12d21c06c880ccf493e14dd3e93f3a9e175325790004954c34d3
c7bc2ccc9f0eb5332014937f9e49bca9b7856d351a553d9812367dc8f2ac734992a4e6a6ff6
6f347bd411d07f0203010001
```

Now, we let us move on to create a transaction in the next chapter.

4. Python Block Chain – Transaction Class

In this chapter, let us create a **Transaction** class so that a client will be able to send money to somebody. Note that a client can be both a sender or a recipient of the money. When you want to receive money, some other sender will create a transaction and specify your **public** address in it. We define the initialization of a transaction class as follows:

```
    def __init__(self, sender, recipient, value):

        self.sender = sender

        self.recipient = recipient

        self.value = value

        self.time = datetime.datetime.now()
```

The **init** method takes three parameters - the sender's **public** key, the recipient's **public** key, and the amount to be sent. These are stored in the instance variables for use by other methods. Additionally, we create one more variable for storing the time of transaction.

Next, we write a utility method called **to_dict** that combines all the four above-mentioned instance variables in a dictionary object. This is just to put the entire transaction information accessible through a single variable.

As you know from the earlier tutorial that the first block in the blockchain is a **Genesis** block. The Genesis block contains the first transaction initiated by the creator of the blockchain. The identity of this person may be kept a secret like in the case of Bitcoins. So when this first transaction is created, the creator may just send his identity as **Genesis**. Thus, while creating the dictionary, we check if the sender is **Genesis** and if so we simply assign some string value to the identity variable; else, we assign the sender's identity to the **identity** variable.

```
    if self.sender == "Genesis":

        identity = "Genesis"

    else:

        identity = self.sender.identity
```

We construct the dictionary using following line of code

```
    return collections.OrderedDict({'sender': identity,

                            'recipient': self.recipient,

                            'value': self.value,

                            'time' : self.time})
```

The entire code for the **to_dict** method is shown below:

```
def to_dict(self):
    if self.sender == "Genesis":
        identity = "Genesis"
    else:
        identity = self.sender.identity

    return collections.OrderedDict({'sender': identity,
                                'recipient': self.recipient,
                                'value': self.value,
                                'time' : self.time})
```

Finally, we will sign this dictionary object using the private key of the sender. As before, we use the built-in PKI with SHA algorithm. The generated signature is decoded to get the ASCII representation for printing and storing it in our blockchain. The **sign_transaction** method code is shown here:

```
def sign_transaction(self):
    private_key = self.sender._private_key
    signer = PKCS1_v1_5.new(private_key)
    h = SHA.new(str(self.to_dict()).encode('utf8'))
    return binascii.hexlify(signer.sign(h)).decode('ascii')
```

We will now test this **Transaction** class.

Testing Transaction Class

For this purpose, we will create two users, called **Dinesh** and **Ramesh**. Dinesh will send 5 TPCoins to Ramesh. For this first we create the clients called Dinesh and Ramesh.

```
Dinesh = Client()
Ramesh = Client()
```

Remember that when you instantiate a **Client** class, the **public and** private keys unique to the client would be created. As Dinesh is sending payment to Ramesh, he will need the public key of Ramesh which is obtained by using the identity property of the client.

Thus, we will create the transaction instance using following code:

```
t = Transaction(
    Dinesh,
    Ramesh.identity,
    5.0
)
```

Note that the first parameter is the sender, the second parameter is the public key of the recipient and the third parameter is the amount to be transferred. The **sign_transaction** method retrieves the sender's *private* key from the first parameter for singing the transaction.

After the transaction object is created, you will sign it by calling its **sign_transaction** method. This method returns the generated signature in the printable format. We generate and print the signature using following two lines of code:

```
signature = t.sign_transaction()

print (signature)
```

When you run the above code, you will see the output similar to this:

```
7c7e3c97629b218e9ec6e86b01f9abd8e361fd69e7d373c38420790b655b9abe3b575e343c7
13703ca1aee781acd7157a0624db3d57d7c2f1172730ee3f45af943338157f899965856f6b0
0e34db240b62673ad5a08c8e490f880b568efbc36035cae2e748f1d802d5e8e66298be826f5
c6363dc511222fb2416036ac04eb972
```

Now as our basic infrastructure of creating a client and a transaction is ready, we will now have multiple clients doing multiple transactions as in a real life situation.

5. Python Block Chain – Creating Multiple Transactions

The transactions made by various clients are queued in the system; the miners pick up the transactions from this queue and add it to the block. They will then mine the block and the winning miner would have the privilege of adding the block to the blockchain and thereby earn some money for himself.

We will describe this mining process later when we discuss the creation of the blockchain. Before we write the code for multiple transactions, let us add a small utility function to print the contents of a given transaction.

Displaying Transaction

The **display_transaction** function accepts a single parameter of transaction type. The dictionary object within the received transaction is copied to a temporary variable called **dict** and using the dictionary keys, the various values are printed on the console.

```
def display_transaction(transaction):
    #for transaction in transactions:
    dict = transaction.to_dict()
    print ("sender: " + dict['sender'])
    print ('-----')
    print ("recipient: " + dict['recipient'])
    print ('-----')
    print ("value: " + str(dict['value']))
    print ('-----')
    print ("time: " + str(dict['time']))
    print ('-----')
```

Next, we define a transaction queue for storing our transaction objects.

Transaction Queue

To create a queue, we declare a global **list** variable called **transactions** as follows:

```
transactions = []
```

We will simply append each newly created transaction to this queue. Please note that for brevity, we will not implement the queue management logic in this tutorial.

Creating Multiple Clients

Now, we will start creating transactions. First, we will create four clients who will send money to each other for obtaining various services or goods from others.

```
Dinesh = Client()

Ramesh = Client()

Seema = Client()

Vijay = Client()
```

At this point, we have four clients called Dinesh, Ramesh, Seema, and Vijay. We currently assume that each of these clients hold some TPCoins in their wallets for transacting. The identity of each of these clients would be specified by using the identity property of these objects.

Creating First Transaction

Now, we initiate our first transaction as follows:

```
t1 = Transaction(

    Dinesh,

    Ramesh.identity,

    15.0

)
```

In this transaction Dinesh sends 5 TPCoins to Ramesh. For transaction to be successful, we will have to ensure that Dinesh has sufficient money in his wallet for this payment. Note that, we will need a genesis transaction to start TPCoin circulation in the system. You will write the transaction code for this genesis transaction very shortly as you read along.

We will sign this transaction using Dinesh's *private* key and add it to the transaction queue as follows:

```
t1.sign_transaction()

transactions.append(t1)
```

After the first transaction made by Dinesh, we will create several more transactions between different clients that we created above.

Adding More Transactions

We will now create several more transactions, each transaction given out a few TPCoins to another party. When somebody spends money, it is not necessary that he has to check for sufficient balances in this wallet. The miner in anyway would be validating each transaction for the balance that the sender has while initiating the transaction.

In case of insufficient balance, the miner will mark this transaction as invalid and would not add it to this block.

The following code creates and adds nine more transactions to our queue.

```
t2 = Transaction(
    Dinesh,
    Seema.identity,
    6.0
)

t2.sign_transaction()
transactions.append(t2)

t3 = Transaction(
    Ramesh,
    Vijay.identity,
    2.0
)

t3.sign_transaction()
transactions.append(t3)

t4 = Transaction(
    Seema,
    Ramesh.identity,
    4.0
)

t4.sign_transaction()
transactions.append(t4)

t5 = Transaction(
    Vijay,
    Seema.identity,
    7.0
)

t5.sign_transaction()
transactions.append(t5)
```

```
t6 = Transaction(
    Ramesh,
    Seema.identity,
    3.0
)

t6.sign_transaction()
transactions.append(t6)

t7 = Transaction(
    Seema,
    Dinesh.identity,
    8.0
)

t7.sign_transaction()
transactions.append(t7)

t8 = Transaction(
    Seema,
    Ramesh.identity,
    1.0
)

t8.sign_transaction()
transactions.append(t8)

t9 = Transaction(
    Vijay,
    Dinesh.identity,
    5.0
)

t9.sign_transaction()
transactions.append(t9)

t10 = Transaction(
    Vijay,
    Ramesh.identity,
    3.0
)

t10.sign_transaction()
transactions.append(t10)
```

When you run the above code, you will have ten transactions in the queue for the miners to create their blocks.

Dumping Transactions

As a blockchain manager, you may periodically like to review the contents of transaction queue. For this purpose, you can use the **display_transaction** function that we developed earlier. To dump all transactions in the queue, just iterate the transactions list and for each referenced transaction, call the **display_transaction** function as shown here:

```
for transaction in transactions:
    display_transaction (transaction)
    print ('--------------')
```

The transactions are separated by a dashed line for distinction. If you run the above code, you would see the transaction list as shown below:

```
sender:
30819f300d06092a864886f70d010101050003818d0030818902818100bb064c99c49214
4a9f463480273aba93ac1db1f0da3cb9f3c1f9d058cf499fd8e54d244da0a8dd6ddd329e
c86794b04d773eb4841c9f935ea4d9ccc2821c7a1082d23b6c928d59863407f52fa05d8b
47e5157f8fe56c2ce3279c657f9c6a80500073b0be8093f748aef667c03e64f04f84d311
c4d866c12d79d3fc3034563dfb0203010001
-----
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100be93b516b28c6e
674abe7abdb11ce0fdf5bb728b75216b73f37a6432e4b402b3ad8139b8c0ba541a72c8ad
d126b6e1a1308fb98b727beb63c6060356bb177bb7d54b54dbe87aee7353d0a6baa93977
04de625d1836d3f42c7ee5683f6703259592cc24b09699376807f28fe0e00ff882974484
d805f874260dfc2d1627473b910203010001
-----
value: 15.0
-----
time: 2019-01-14 16:18:01.859915
-----
--------------

sender:
30819f300d06092a864886f70d010101050003818d0030818902818100bb064c99c49214
4a9f463480273aba93ac1db1f0da3cb9f3c1f9d058cf499fd8e54d244da0a8dd6ddd329e
c86794b04d773eb4841c9f935ea4d9ccc2821c7a1082d23b6c928d59863407f52fa05d8b
47e5157f8fe56c2ce3279c657f9c6a80500073b0be8093f748aef667c03e64f04f84d311
c4d866c12d79d3fc3034563dfb0203010001
-----
```

```
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100a070c82b34ae14
3cbe59b3a2afde7186e9d5bc274955d8112d87a00256a35369acc4d0edfe65e8f9dc93fb
d9ee74b9e7ea12334da38c8c9900e6ced1c4ce93f86e06611e656521a1eab561892b7db0
961b4f212d1fd5b5e49ae09cf8c603a068f9b723aa8a651032ff6f24e5de00387e4d0623
75799742a359b8f22c5362e5650203010001
-----
value: 6.0
-----
time: 2019-01-14 16:18:01.860966
-----
--------------
sender:
30819f300d06092a864886f70d010101050003818d0030818902818100be93b516b28c6e
674abe7abdb11ce0fdf5bb728b75216b73f37a6432e4b402b3ad8139b8c0ba541a72c8ad
d126b6e1a1308fb98b727beb63c6060356bb177bb7d54b54dbe87aee7353d0a6baa93977
04de625d1836d3f42c7ee5683f6703259592cc24b09699376807f28fe0e00ff882974484
d805f874260dfc2d1627473b910203010001
-----
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100cba097c0854876
f41338c62598c658f545182cfa4acebce147aedf328181f9c4930f14498fd03c0af6b0cc
e25be99452a81df4fa30a53eddbb7bb7b203adf8764a0ccd9db6913a576d68d642d8fd47
452590137869c25d9ff83d68ebe6d616056a8425b85b52e69715b8b85ae807b84638d8f0
0e321b65e4c33acaf6469e18e30203010001
-----
value: 2.0
-----
time: 2019-01-14 16:18:01.861958
-----
--------------
```

For brevity, I have printed only first few transactions in the list. In the above code, we print all transactions beginning with the very first transaction except for the genesis transaction which was never added to this list. As the transactions are added to the blocks periodically, you will generally be interested in viewing only the list of transactions which are yet to be mined. In that case, you will need to create an appropriate **for** loop to iterate through the transactions which are not yet mined.

So far, you have learned how to create clients, allow them to among themselves and maintain a queue of the pending transactions which are to be mined. Now, comes the most important part of this tutorial and that is creating a blockchain itself. You will learn this in the next lesson.

6. Python Block Chain – Block Class

A block consists of a varying number of transactions. For simplicity, in our case we will assume that the block consists of a fixed number of transactions, which is three in this case. As the block needs to store the list of these three transactions, we will declare an instance variable called **verified_transactions** as follows:

```
        self.verified_transactions = []
```

We have named this variable as **verified_transactions** to indicate that only the verified valid transactions will be added to the block. Each block also holds the hash value of the previous block, so that the chain of blocks becomes immutable.

To store the previous hash, we declare an instance variable as follows:

```
        self.previous_block_hash = ""
```

Finally, we declare one more variable called **Nonce** for storing the nonce created by the miner during the mining process.

```
        self.Nonce = ""
```

The full definition of the **Block** class is given below:

```
class Block:
    def __init__(self):
        self.verified_transactions = []
        self.previous_block_hash = ""
        self.Nonce = ""
```

As each block needs the value of the previous block's hash we declare a global variable called **last_block_hash** as follows:

```
last_block_hash = ""
```

Now let us create our first block in the blockchain.

7. Python Block Chain – Creating Genesis Block

We assume that the originator of TPCoins initially gives out 500 TPCoins to a known client **Dinesh**. For this, he first creates a Dinesh instance:

```
Dinesh = Client()
```

We then create a genesis transaction and send 500 TPCoins to Dinesh's public address.

```
t0 = Transaction (
    "Genesis",
    Dinesh.identity,
    500.0
)
```

Now, we create an instance of **Block** class and call it **block0**.

```
block0 = Block()
```

We initialize the **previous_block_hash** and **Nonce** instance variables to **None**, as this is the very first transaction to be stored in our blockchain.

```
block0.previous_block_hash = None
Nonce = None
```

Next, we will add the above t0 transaction to the **verified_transactions** list maintained within the block:

```
block0.verified_transactions.append (t0)
```

At this point, the block is completely initialized and is ready to be added to our blockchain. We will be creating the blockchain for this purpose. Before we add the block to the blockchain, we will hash the block and store its value in the global variable called **last_block_hash** that we declared previously. This value will be used by the next miner in his block.

We use the following two lines of coding for hashing the block and storing the digest value.

```
digest = hash (block0)
last_block_hash = digest
```

Finally, we create a blockchain as we see in the next chapter.

8. Python Blockchain – Creating Blockchain

A blockchain contains a list of blocks chained to each other. To store the entire list, we will create a list variable called TPCoins:

```
TPCoins = []
```

We will also write a utility method called **dump_blockchain** for dumping the contents of the entire blockchain. We first print the length of the blockchain so that we know how many blocks are currently present in the blockchain.

```
def dump_blockchain (self):
    print ("Number of blocks in the chain: " + str(len (self)))
```

Note that as the time passes, the number of blocks in the blockchain would be extraordinarily high for printing. Thus, when you print the contents of the blockchain you may have to decide on the range that you would like to examine. In the code below, we have printed the entire blockchain as we would not be adding too many blocks in the current demo.

To iterate through the chain, we set up a **for** loop as follows:

```
    for x in range (len(TPCoins)):
        block_temp = TPCoins[x]
```

Each referenced block is copied to a temporary variable called **block_temp**.

We print the block number as a heading for each block. Note that the numbers would start with zero, the first block is a genesis block that is numbered zero.

```
            print ("block # " + str(x))
```

Within each block, we have stored a list of three transactions (except for the genesis block) in a variable called **verified_transactions**. We iterate this list in a **for** loop and for each retrieved item, we call **display_transaction** function to display the transaction details.

```
        for transaction in block_temp.verified_transactions:
            display_transaction (transaction)
```

The entire function definition is shown below:

```
def dump_blockchain (self):
    print ("Number of blocks in the chain: " + str(len (self)))
    for x in range (len(TPCoins)):
        block_temp = TPCoins[x]
        print ("block # " + str(x))
        for transaction in block_temp.verified_transactions:
            display_transaction (transaction)
            print ('--------------')
        print ('=====================================')
```

Note that here we have inserted the separators at appropriate points in the code to demarcate the blocks and transactions within it.

As we have now created a blockchain for storing blocks, our next task is to create blocks and start adding it to the blockchain. For this purpose, we will add a genesis block that you have already created in the earlier step.

9. Python Blockchain – Adding Genesis Block

Adding a block to the blockchain involves appending the created block to our **TPCoins** list.

```
TPCoins.append (block0)
```

Note that unlike the rest of the blocks in the system, the genesis block contains only one transaction which is initiated by the originator of the TPCoins system. Now, you will dump the contents of the blockchain by calling our global function **dump_blockchain**:

```
dump_blockchain(TPCoins)
```

When you execute this function, you will see the following output:

```
Number of blocks in the chain: 1
block # 0
sender: Genesis
-----
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100ed272b52ccb539
e2cd779c6cc10ed1dfadf5d97c6ab6de90ed0372b2655626fb79f62d0e01081c163b0864
cc68d426bbe9438e8566303bb77414d4bfcaa3468ab7febac099294de10273a816f7047d
4087b4bafa11f141544d48e2f10b842cab91faf33153900c7bf6c08c9e47a7df8aa7e60d
c9e0798fb2ba3484bbdad2e4430203010001
-----
value: 500.0
-----
time: 2019-01-14 16:18:02.042739
-----
--------------
=====================================
```

At this point the blockchain system is ready to use. We will now enable the interested clients to become miners by providing them a mining functionality.

10. Python Blockchain – Creating Miners

For enabling mining, we need to develop a mining function. The mining functionality needs to generate a digest on a given message string and provide a proof-of-work. Let us discuss this in this chapter.

Message Digest Function

We will write a utility function called **sha256** for creating a digest on a given message:

```
def sha256(message):
    return hashlib.sha256(message.encode('ascii')).hexdigest()
```

The **sha256** function takes a **message** as a parameter, encodes it to ASCII, generates a hexadecimal digest and returns the value to the caller.

Mining Function

We now develop the **mine** function that implements our own mining strategy. Our strategy in this case would be to generate a hash on the given message that is prefixed with a given number of 1's. The given number of 1's is specified as a parameter to **mine** function specified as the difficulty level.

For example, if you specify a difficulty level of 2, the generated hash on a given message should start with two 1's - like 11xxxxxxxx. If the difficulty level is 3, the generated hash should start with three 1's - like 111xxxxxxxx. Given these requirements, we will now develop the mining function as shown in the steps given below.

Step 1

The mining function takes two parameters - the message and the difficulty level.

```
def mine(message, difficulty=1):
```

Step 2

The difficulty level needs to be greater or equal to 1, we ensure this with the following assert statement:

```
    assert difficulty >= 1
```

Step 3

We create a **prefix** variable using the set difficulty level.

```
    prefix = '1' * difficulty
```

Note if the difficulty level is 2 the prefix would be "11" and if the difficulty level is 3, the prefix would be "111", and so on. We will check if this prefix exists in the generated digest of the message. To digest the message itself, we use the following two lines of code:

```
    for i in range(1000):
        digest = sha256(str(hash(message)) + str(i))
```

We keep on adding a new number **i** to the message hash in each iteration and generate a new digest on the combined message. As the input to the **sha256** function changes in every iteration, the **digest** value would also change. We check if this **digest** value has above-set **prefix**.

```
        if digest.startswith(prefix):
```

If the condition is satisfied, we will terminate the **for** loop and return the **digest** value to the caller.

The entire **mine** code is shown here:

```
def mine(message, difficulty=1):
    assert difficulty >= 1
    prefix = '1' * difficulty
    for i in range(1000):
        digest = sha256(str(hash(message)) + str(i))
        if digest.startswith(prefix):
            print ("after " + str(i) + " iterations found nonce: "
+ digest)
            return digest
```

For your understanding, we have added the **print** statement that prints the digest value and the number of iterations it took to meet the condition before returning from the function.

Testing Mining Function

To test our mining function, simply execute the following statement:

```
mine ("test message", 2)
```

When you run the above code, you will see the output similar to the one below:

```
after 138 iterations found nonce:
11008a740eb2fa6bf8d55baecda42a41993ca65ce66b2d3889477e6bfad1484c
```

Note that the generated digest starts with "11". If you change the difficulty level to 3, the generated digest will start with "111", and of course, it will probably require more number of iterations. As you can see, a miner with more processing power will be able to mine a given message earlier. That's how the miners compete with each other for earning their revenues.

Now, we are ready to add more blocks to our blockchain. Let us learn this in our next chapter.

11. Python Blockchain – Adding Blocks

Each miner will pick up the transactions from a previously created transaction pool. To track the number of messages already mined, we have to create a global variable:

```
last_transaction_index = 0
```

We will now have our first miner adding a block to the blockchain.

Adding First Block

To add a new block, we first create an instance of the **Block** class.

```
block = Block()
```

We pick up the top 3 transactions from the queue:

```
for i in range(3):
    temp_transaction = transactions[last_transaction_index]
    # validate transaction
```

Before adding the transaction to the block the miner will verify the validity of the transaction. The transaction validity is verified by testing for equality the hash provided by the sender against the hash generated by the miner using sender's public key. Also, the miner will verify that the sender has sufficient balance to pay for the current transaction.

For brevity, we have not included this functionality in the tutorial. After the transaction is validated, we add it to the **verified_transactions** list in the **block** instance.

```
    block.verified_transactions.append (temp_transaction)
```

We increment the last transaction index so that the next miner will pick up subsequent transactions in the queue.

```
    last_transaction_index += 1
```

We add exactly three transactions to the block. Once this is done, we will initialize the rest of the instance variables of the **Block** class. We first add the hash of the last block.

```
block.previous_block_hash = last_block_hash
```

Next, we mine the block with a difficulty level of 2.

```
block.Nonce = mine (block, 2)
```

Note that the first parameter to the **mine** function is a binary object. We now hash the entire block and create a digest on it.

```
digest = hash (block)
```

Finally, we add the created block to the blockchain and re-initialize the global variable **last_block_hash** for the use in next block.

The entire code for adding the block is shown below:

```
block = Block()

for i in range(3):
    temp_transaction = transactions[last_transaction_index]
    # validate transaction
    # if valid
    block.verified_transactions.append (temp_transaction)
    last_transaction_index += 1

block.previous_block_hash = last_block_hash
block.Nonce = mine (block, 2)
digest = hash (block)
TPCoins.append (block)
last_block_hash = digest
```

Adding More Blocks

We will now add two more blocks to our blockchain. The code for adding the next two blocks is given below:

```
# Miner 2 adds a block
block = Block()

for i in range(3):
    temp_transaction = transactions[last_transaction_index]
    # validate transaction
    # if valid
    block.verified_transactions.append (temp_transaction)
    last_transaction_index += 1
block.previous_block_hash = last_block_hash
block.Nonce = mine (block, 2)digest = hash (block)
TPCoins.append (block)last_block_hash = digest
```

```
# Miner 3 adds a block
block = Block()

for i in range(3):
    temp_transaction = transactions[last_transaction_index]
    #display_transaction (temp_transaction)
    # validate transaction
    # if valid
    block.verified_transactions.append (temp_transaction)
    last_transaction_index += 1

block.previous_block_hash = last_block_hash
block.Nonce = mine (block, 2)
digest = hash (block)

TPCoins.append (block)
last_block_hash = digest
```

When you add these two blocks, you will also see the number of iterations it took to find the Nonce. At this point, our blockchain consists of totally 4 blocks including the genesis block.

Dumping Entire Blockchain

You can verify the contents of the entire blockchain using the following statement:

```
dump_blockchain(TPCoins)
```

You would see the output similar to the one shown below:

```
Number of blocks in the chain: 4
block # 0
sender: Genesis
-----
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100ed272b52ccb539e2cd779
c6cc10ed1dfadf5d97c6ab6de90ed0372b2655626fb79f62d0e01081c163b0864cc68d426bbe943
8e8566303bb77414d4bfcaa3468ab7febac099294de10273a816f7047d4087b4bafa11f141544d4
8e2f10b842cab91faf33153900c7bf6c08c9e47a7df8aa7e60dc9e0798fb2ba3484bbdad2e44302
03010001
-----
value: 500.0
-----
time: 2019-01-14 16:18:02.042739
-----
```

```
--------------
=====================================

block # 1
sender:
30819f300d06092a864886f70d010101050003818d0030818902818100bb064c99c492144a9f463
480273aba93ac1db1f0da3cb9f3c1f9d058cf499fd8e54d244da0a8dd6ddd329ec86794b04d773e
b4841c9f935ea4d9ccc2821c7a1082d23b6c928d59863407f52fa05d8b47e5157f8fe56c2ce3279
c657f9c6a80500073b0be8093f748aef667c03e64f04f84d311c4d866c12d79d3fc3034563dfb02
03010001
-----
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100be93b516b28c6e674abe7
abdb11ce0fdf5bb728b75216b73f37a6432e4b402b3ad8139b8c0ba541a72c8add126b6e1a1308f
b98b727beb63c6060356bb177bb7d54b54dbe87aee7353d0a6baa9397704de625d1836d3f42c7ee
5683f6703259592cc24b09699376807f28fe0e00ff882974484d805f874260dfc2d1627473b9102
03010001
-----
value: 15.0
-----
time: 2019-01-14 16:18:01.859915
-----
--------------
sender:
30819f300d06092a864886f70d010101050003818d0030818902818100bb064c99c492144a9f463
480273aba93ac1db1f0da3cb9f3c1f9d058cf499fd8e54d244da0a8dd6ddd329ec86794b04d773e
b4841c9f935ea4d9ccc2821c7a1082d23b6c928d59863407f52fa05d8b47e5157f8fe56c2ce3279
c657f9c6a80500073b0be8093f748aef667c03e64f04f84d311c4d866c12d79d3fc3034563dfb02
03010001
-----
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100a070c82b34ae143cbe59b
3a2afde7186e9d5bc274955d8112d87a00256a35369acc4d0edfe65e8f9dc93fbd9ee74b9e7ea12
334da38c8c9900e6ced1c4ce93f86e06611e656521a1eab561892b7db0961b4f212d1fd5b5e49ae
09cf8c603a068f9b723aa8a651032ff6f24e5de00387e4d062375799742a359b8f22c5362e56502
03010001
-----
value: 6.0
-----
time: 2019-01-14 16:18:01.860966
-----
--------------
sender:
30819f300d06092a864886f70d010101050003818d0030818902818100be93b516b28c6e674abe7
abdb11ce0fdf5bb728b75216b73f37a6432e4b402b3ad8139b8c0ba541a72c8add126b6e1a1308f
b98b727beb63c6060356bb177bb7d54b54dbe87aee7353d0a6baa9397704de625d1836d3f42c7ee
5683f6703259592cc24b09699376807f28fe0e00ff882974484d805f874260dfc2d1627473b9102
03010001
-----
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100cba097c0854876f41338c
62598c658f545182cfa4acebce147aedf328181f9c4930f14498fd03c0af6b0cce25be99452a81d
f4fa30a53eddbb7bb7b203adf8764a0ccd9db6913a576d68d642d8fd47452590137869c25d9ff83
d68ebe6d616056a8425b85b52e69715b8b85ae807b84638d8f00e321b65e4c33acaf6469e18e302
03010001
-----
```

```
value: 2.0
-----
time: 2019-01-14 16:18:01.861958
-----
--------------
=====================================
block # 2
sender:
30819f300d06092a864886f70d010101050003818d0030818902818100a070c82b34ae143cbe59b
3a2afde7186e9d5bc274955d8112d87a00256a35369acc4d0edfe65e8f9dc93fbd9ee74b9e7ea12
334da38c8c9900e6ced1c4ce93f86e06611e656521a1eab561892b7db0961b4f212d1fd5b5e49ae
09cf8c603a068f9b723aa8a651032ff6f24e5de00387e4d062375799742a359b8f22c5362e56502
03010001
-----
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100be93b516b28c6e674abe7
abdb11ce0fdf5bb728b75216b73f37a6432e4b402b3ad8139b8c0ba541a72c8add126b6e1a1308f
b98b727beb63c6060356bb177bb7d54b54dbe87aee7353d0a6baa9397704de625d1836d3f42c7ee
5683f6703259592cc24b09699376807f28fe0e00ff882974484d805f874260dfc2d1627473b9102
03010001
-----
value: 4.0
-----
time: 2019-01-14 16:18:01.862946
-----
--------------
sender:
30819f300d06092a864886f70d010101050003818d0030818902818100cba097c0854876f41338c
62598c658f545182cfa4acebce147aedf328181f9c4930f14498fd03c0af6b0cce25be99452a81d
f4fa30a53eddbb7bb7b203adf8764a0ccd9db6913a576d68d642d8fd47452590137869c25d9ff83
d68ebe6d616056a8425b85b52e69715b8b85ae807b84638d8f00e321b65e4c33acaf6469e18e302
03010001
-----
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100a070c82b34ae143cbe59b
3a2afde7186e9d5bc274955d8112d87a00256a35369acc4d0edfe65e8f9dc93fbd9ee74b9e7ea12
334da38c8c9900e6ced1c4ce93f86e06611e656521a1eab561892b7db0961b4f212d1fd5b5e49ae
09cf8c603a068f9b723aa8a651032ff6f24e5de00387e4d062375799742a359b8f22c5362e56502
03010001
-----
value: 7.0
-----
time: 2019-01-14 16:18:01.863932
-----
--------------
sender:
30819f300d06092a864886f70d010101050003818d0030818902818100be93b516b28c6e674abe7
abdb11ce0fdf5bb728b75216b73f37a6432e4b402b3ad8139b8c0ba541a72c8add126b6e1a1308f
b98b727beb63c6060356bb177bb7d54b54dbe87aee7353d0a6baa9397704de625d1836d3f42c7ee
5683f6703259592cc24b09699376807f28fe0e00ff882974484d805f874260dfc2d1627473b9102
03010001
-----
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100a070c82b34ae143cbe59b
3a2afde7186e9d5bc274955d8112d87a00256a35369acc4d0edfe65e8f9dc93fbd9ee74b9e7ea12
334da38c8c9900e6ced1c4ce93f86e06611e656521a1eab561892b7db0961b4f212d1fd5b5e49ae
```

```
09cf8c603a068f9b723aa8a651032ff6f24e5de00387e4d062375799742a359b8f22c5362e56502
03010001
-----
value: 3.0
-----
time: 2019-01-14 16:18:01.865099
-----
--------------
=====================================
block # 3
sender:
30819f300d06092a864886f70d010101050003818d0030818902818100a070c82b34ae143cbe59b
3a2afde7186e9d5bc274955d8112d87a00256a35369acc4d0edfe65e8f9dc93fbd9ee74b9e7ea12
334da38c8c9900e6ced1c4ce93f86e06611e656521a1eab561892b7db0961b4f212d1fd5b5e49ae
09cf8c603a068f9b723aa8a651032ff6f24e5de00387e4d062375799742a359b8f22c5362e56502
03010001
-----
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100bb064c99c492144a9f463
480273aba93ac1db1f0da3cb9f3c1f9d058cf499fd8e54d244da0a8dd6ddd329ec86794b04d773e
b4841c9f935ea4d9ccc2821c7a1082d23b6c928d59863407f52fa05d8b47e5157f8fe56c2ce3279
c657f9c6a80500073b0be8093f748aef667c03e64f04f84d311c4d866c12d79d3fc3034563dfb02
03010001
-----
value: 8.0
-----
time: 2019-01-14 16:18:01.866219
-----
--------------
sender:
30819f300d06092a864886f70d010101050003818d0030818902818100a070c82b34ae143cbe59b
3a2afde7186e9d5bc274955d8112d87a00256a35369acc4d0edfe65e8f9dc93fbd9ee74b9e7ea12
334da38c8c9900e6ced1c4ce93f86e06611e656521a1eab561892b7db0961b4f212d1fd5b5e49ae
09cf8c603a068f9b723aa8a651032ff6f24e5de00387e4d062375799742a359b8f22c5362e56502
03010001
-----
recipient:
30819f300d06092a864886f70d010101050003818d0030818902818100be93b516b28c6e674abe7
abdb11ce0fdf5bb728b75216b73f37a6432e4b402b3ad8139b8c0ba541a72c8add126b6e1a1308f
b98b727beb63c6060356bb177bb7d54b54dbe87aee7353d0a6baa9397704de625d1836d3f42c7ee
5683f6703259592cc24b09699376807f28fe0e00ff882974484d805f874260dfc2d1627473b9102
03010001
-----
value: 1.0
-----
time: 2019-01-14 16:18:01.867223
-----
--------------
sender:
30819f300d06092a864886f70d010101050003818d0030818902818100cba097c0854876f41338c
62598c658f545182cfa4acebce147aedf328181f9c4930f14498fd03c0af6b0cce25be99452a81d
f4fa30a53eddbb7bb7b203adf8764a0ccd9db6913a576d68d642d8fd47452590137869c25d9ff83
d68ebe6d616056a8425b85b52e69715b8b85ae807b84638d8f00e321b65e4c33acaf6469e18e302
03010001
-----
recipient:
```

```
30819f300d06092a864886f70d010101050003818d0030818902818100bb064c99c492144a9f463
480273aba93ac1db1f0da3cb9f3c1f9d058cf499fd8e54d244da0a8dd6ddd329ec86794b04d773e
b4841c9f935ea4d9ccc2821c7a1082d23b6c928d59863407f52fa05d8b47e5157f8fe56c2ce3279
c657f9c6a80500073b0be8093f748aef667c03e64f04f84d311c4d866c12d79d3fc3034563dfb02
03010001
-----
value: 5.0
-----
time: 2019-01-14 16:18:01.868241
-----
--------------
=====================================
```

12. Python Blockchain – Scope and Conclusion

In this tutorial, we have learnt how to construct a blockchain project in Python. There are many areas where you need to add further functionality to this project.

For instance, you will need to write functions for managing the transactions queue. After the transactions are mined and the mined block is accepted by the system, they need not be stored any more.

Also, the miners would certainly prefer to pick up the transactions with the highest fee. At the same time, you will have to ensure that the transactions with low or no fee would not starve forever.

You will need to develop algorithms for managing the queue. Also, the current tutorial does not include the client interface code. You will need to develop this for both normal clients and the miners. The full-fledged blockchain project would run into several more lines of code and is beyond the scope of this tutorial. The interested reader may download the bitcoin source for further study.

Conclusions

This crisp tutorial should get you started on creating your own blockchain project.

For full-fledged blockchain project development, you can learn more from the bitcoin source.

For larger commercial or non-commercial projects, you may consider using Ethereum - a ready to use blockchain app platform.