Hi! This article is a simple guide to build a standard Blockchain in python. We create Clients and transactions as separate classes and use them to create blocks in a more clear and understandable manner. The whole article will come with 2 parts. This is the first part of it and here we define the necessary classes and build the first block of the blockchain. It is super easy and anyone can build your own blockchain by following this article. So let's get started!

We use Jupyter notebook for implementation and first of all, we import the necessary libraries to the notebook as follows.

```
# import libraries
import hashlib
import random
import string
import json
import binascii
import datetime
import collections
```

```
import Crypto
import Crypto.Random
from Crypto.Hash import SHA
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
```

Client Class

First, we create the Client Class. The client is someone who is able to send money from his wallet to another known person. Also, a client is a person who is able to accept money from a third party.

In the Client class, we generate 2 types of keys such as Private Key and Public Key. You should never expose your private key for others. The generated public key is used as the client's identity in transactions. We generate those key values using the RSA algorithm. During the object initialization, we create these keys and store them in the instance variables. We define a property called 'Identity' to return the Hex representation of the client's public key. (PKCS1 is the standard defines the mathematical definitions and properties that RSA public and private keys must have)

```
# Client Class generate the public and private keys
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)

# The generated public key will used as the client's identity
# it returns HEX representation of the public key

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

Now let us test the instance of a Client class and its generated public key

```
# Creates an instance of Client and assigns it to the variable Kavindu.
# We print the public key of Kavindu by calling its identity property
Kavindu = Client()
print (Kavindu.identity)
```

```
30819f300d06092a864886f70d010101050003818d0030818902818100f914786c4c35737304a06b8e1612230f708fd926fa57789ec18510886044690891093
d42d989ae33b4e7bdea3fd1b14a6ec8b709639c425d93e767b0e9850cbd11917896a1122bbd822b24c17a33b2fa66bdbb230b4b221dab03edacbb16587cf20f
b83fb6904f356800364f70c2a10f4695149141abfcf68135b2f19c72ee250203010001
```

Transaction Class

Now let us create the transaction class. Here we define the sender, receiver, time of transaction and the value of the transaction. To complete the transaction we need the receiver's public key. For example, when you want to receive money, some other sender will create a transaction and specify your public address in it. In the method, we take 3 parameters such as the sender's public key, receiver's public key and the value of the transaction. (Use of transactions list and last_block_hash variable will be explained later)

```
#in Transaction class, client will be able to send money to somebody.
transactions = []
last_block_hash=""

class Transaction:

    def __init__(self, sender,recipient, value):
        # senders public key
        self.sender=sender

        #Recivers public key
        self.recipient=recipient

        # Amount to be sent
        self.value=value

        # Time of Transaction
        self.time=datetime.datetime.now()
```

Next, we create a method called to_dict to combine all the 4 values in a transaction to a dictionary object. In the blockchain, the first block of the chain is called the "Genesis block". The Genesis block contains the first transaction originated by the creator of the blockchain. Thus, while creating the dictionary, we check if the sender is Genesis and if so we simply assign some string value ("Genesis") to the identity variable. Otherwise, we assign the sender's identity to the identity variable.

```
# Combine all 4 variables to a dictionary object
   def to_dict(self):
       if self.sender=="Genesis":
           identity="Genesis"

       else:
           identity=self.sender.identity

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

Next, we sign the dictionary object using the private key of the sender. We create a method sign_transaction to create a signature and convert it to ASCII representation to store in our blockchain.

```
      # We sign the dictionary object using the private key of the sender
    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')
```

Finally we create a method to display the transactions in a clear manner.

```
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 ('-----')
```

Let's create a few instances of Transaction Class and append it to the transactions list that we create earlier. 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 signing 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.

```
Kavinda = Client()
lakshan = Client()
Indunil = Client()
Dushan = Client()

t1 = Transaction(Kavinda,Indunil.identity,15.0)
t1.sign_transaction()
transactions.append(t1)

t2 = Transaction(Indunil,Dushan.identity,6.0)
t2.sign_transaction()
transactions.append(t2)

t3 = Transaction(lakshan,Indunil.identity,2.0)
t3.sign_transaction()
transactions.append(t3)

t4 = Transaction(lakshan,Dushan.identity,4.0)
t4.sign_transaction()
transactions.append(t4)

t5 = Transaction(Kavinda,Dushan.identity,7.0)
t5.sign_transaction()
transactions.append(t5)
```

Now let us display all the transactions. As you see below all the information including the generated keys of the clients will be displayed.

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

```
sender: 30819f300d06092a864886f70d010101050003818d0030818902818100b15cbb6869c70c4631243847b6c3d7fb776238775dcf87fa1c70015a955
b3d9b7eb59470f58ce12c0ad5e84c9da327e80ecd2d9452e50c2de9019ff6e6a4c08d61e74458f0b2717bd92b4d50af624234816cddd3a5fc9aacff291e8d
bfeda831c322ca86ef85a589d0cc94627849bdd21d385afd58e75769741343c8074bed6f0203010001
-----
recipient: 30819f300d06092a864886f70d010101050003818d0030818902818100d9bf49f340baa5c90013e39eedc77d6cf59e75361d1b181cd99d3a1
39bc61573905ae0c9a5cc1b2ba2233b5e606e2967b65f6d85eb87035ae6137d8b5fa1f854342e933ec4c236345fea03055904380e6c6bb00395c874135dfd
ef5dd5a7ba37d1f37287bd47df2673bcec42c89ad1f476b2d4cb14784502e4f5c5f444df3c30203010001
-----
value: 15.0
-----
time: 2020-03-31 20:32:14.291155
-----
--------------
sender: 30819f300d06092a864886f70d010101050003818d0030818902818100b15cbb6869c70c4631243847b6c3d7fb776238775dcf87fa1c70015a955
b3d9b7eb59470f58ce12c0ad5e84c9da327e80ecd2d9452e50c2de9019ff6e6a4c08d61e74458f0b2717bd92b4d50af624234816cddd3a5fc9aacff291e8d
bfeda831c322ca86ef85a589d0cc94627849bdd21d385afd58e75769741343c8074bed6f0203010001
-----
recipient: 30819f300d06092a864886f70d010101050003818d003081890281810098896c72f50dcc7a5caf75218360c75c3ca437879b7f97a2be9c3b57
a4556bc722c9b074276f0357cba76b0c0d98762ee1240971a8f9b1a7415698d2dccabb3bb50591912155a1bf4794921e51c6d8b467d56899d0ba29ec8c7c1
6f9bd08ac0c9f38e90f69fd86408691a1b5d5f146dee1611ad16344fa8628a183be739a0ccb0203010001
```

Block Class

So far we create two classes; Client and transaction. Now let us look at the Block Class. A block can consist of many transactions. For our explanation, we will assume it has a fixed number of transactions which is three.

First, we create an instance variable “verified_transactions” to include the valid transactions. Each block also keeps the hash value of the previous block, so that the chain of blocks becomes unchangeable. To store the previous hash, we declare an instance variable called “previous_block_hash”. Next, we declare another variable called “Nonce” to store the nonce value generated by the miner in the mining process. **In mining, every single miner starts trying to find the solution to that one Nonce that will satisfy the hash for the block**

```
class Block:

    def __init__(self):
        self.verified_transaction=[]
        self.previous_block_hash=""
        self.Nonce=""
```

.

Create the Genesis Block

Now we create the first block of the blockchain; The Genesis block. First, we create a Client object and transaction object and set the parameters accordingly.

As this is the first block, we do not have the sender’s public key. So we send the string “Genesis” as the parameter. Next, we create the block object and append the transaction details and hash the entire block and store its value in a variable called last_block_hash. This value will be used for the “previous_block_hash” in the second block that we create.

```
# Create a Client object
Mahee=Client()

# Create a transaction Object
t0 = Transaction ("Genesis",Mahee.identity,500.0)

# Create a block Object
# Here we create a Genises Block
block0=Block()
block0.verified_transaction.append(t0)
block0.previous_block_hash=None
Nonce=None

# Hash the entire block
digest=hash(block0)

# Assign the hashed value to variable last_block_hash
last_block_hash=digest
```

Create the BlockChain and display the Blocks

We create a list called TPCoins to store all the blocks. Then we define a method to display the blocks inside the TPCoins list.

```
# TPCoins store the Blocks
TPCoins=[]

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

    for x in range (len(TPCoins)): # loop through the TPCoins array

        # assign each block in TPCoins to a temp variable block_temp
        block_temp = TPCoins[x]
        print ("block # " + str(x))

        # display the transactions inside the block
        for transaction in block_temp.verified_transaction:
            display_transaction (transaction)
            print ('--------------')

        print ('====================================')
```

Let's append our Genesis block to the list and display its details.

```
# Append the Genesis block to TPCoins
TPCoins.append(block0)
dump_blockchain(TPCoins)
```

```
Number of blocks in the chain: 1
block # 0
sender: Genesis
-----
recipient: 30819f300d06092a864886f70d010101050003818d0030818902818100d26e
0d15882931198f8ca19fff62b2f51e96238974833b7918928e695fe433d15a41abb3c5b0a
4b669f2b220a16f86413357b9bba0fc81e991ffcb8e282b140f6173a5563f64074898eefb
a29b3093a2eca878b16d978f137dd1eb27a1ccdaf292521a31332cac09566786e425cf5c8
5009bb7564fb65ee26342bd1b8696619d0203010001
-----
value: 500.0
-----
time: 2020-03-31 20:32:14.779778
-----
--------------
====================================
```

Creating Miners

We are now done with creating Client, Transaction, Block classes and creating a Genesis block. In this article we would create miners and add more blocks to our blockchain.

So who are Miners?

The role of a minor is to build the blockchain of records that forms the blocks. Let us understand it in simple terms. First, a transaction occurs and all the data about the transaction is added to the block. Also we add the hash value of the previous block to keep up the chain. This whole block is again convert in to a hash value with the use of hashing algorithms.

A miner is someone who trying to identify the numeric value which is called as 'Nonce' that satisfy the generated hash value. Every single miner starts trying to find the solution to that one Nonce that will satisfy the hash for the block. At some specific point, one of those miners in the world with higher speed and great hardware specifications will solve the challenge to identify the Nonce value. Now, the rest of other miners will start verifying that block which is mined by the winner. If the Nonce is correct, it will end up with the new block which will be added to the blockchain

Okay let us continue with the implementation. As the next step we will create a method called "sha256" which accept one parameter. The role of this method is to generate hash value, encode it in to an ASCII, generate a hexadecimal digest and returns the value to the caller.

Next we create a mine function which accept 2 parameters. Here we define our own mining 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.

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

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

```
# Develop a Mine function
# Here it tries to find the solution to that one NOnce that will satisfy the hash for the block
# We send the block and the difficulty level as the parameters to the mine function.

def mine(message,difficulty=1):
    #The difficulty level needs to be greater or equal to 1, we ensure this with the following assert statement:
    assert difficulty >= 1

    #We create a prefix variable using the set difficulty level.
    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
```

Let's test the mine function and see the result. Here we will send a string parameter as the message and difficulty level to 2.

```
In [432]: # Test the mine function
          mine("test message", 2)

          after 260 iterations found nonce: 116daa1738f66cf3a3c7b08df7bc47dc8e68ba11b
          e7124b9dc45d47ba312efff

Out[432]: '116daa1738f66cf3a3c7b08df7bc47dc8e68ba11be7124b9dc45d47ba312efff'
```

As you can see the output is the nonce value that satisfy the hash value of the message. That value is found after 260 iterations. If you increase the difficulty level no. of iterations will be increased.

Add new Blocks to the Chain

Now let us see how new blocks can be added to the chain with the mining function. Here for our simplicity we create 3 transactions and append it to the new block instance that we create from the Block class. Then we assign the previous_block_hash and now get the nonce value by calling mine function.

Now our block is completed. Let us add it to the TPCoins list which contains the all the blocks we create.

```
#Miner 2 adds a block
last_transaction_index=0
block = Block()

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

# Assign the value of last_block_hash to the previous_block_hash attribute in the block
block.previous_block_hash = last_block_hash

# Send the block which has (Transactions + previous_bloc_hash) for the mine function
block.Nonce = mine (block, 2)

TPCoins.append (block)

# Hased the currrent block and store its hashed value in the last_block_hash variable
digest = hash (block)
#print("Block hash of the current Block "+ str(digest))
last_block_hash = digest
```

```
after 162 iterations found nonce: 11b8535b312c5e80587eb2e2a8ff67ad66440b1aac34318fec9c49bdacbe67d4
```

Congratulations! You are done with creating your first blockchain!! Let us see the output of the full chain

```
dump_blockchain(TPCoins)
```

```
Number of blocks in the chain: 2
block # 0
sender: Genesis
-----
recipient: 30819f300d06092a864886f70d01010105000381800308189028181100d26e0d15882931198f8ca19fff62b2f51e96238974833b7918928e695f
e433d15a41abb3c5b0a4b669f2b220a16f86413357b9bba0fc81e991ffcb8e282b140f6173a5563f64074898eefba29b3093a2eca878b16d978f137dd1eb27a
1ccdaf292521a31332cac09566786e425cf5c85009bb7564fb65ee26342bd1b8696619d0203010001
-----
value: 500.0
-----
time: 2020-03-31 20:32:14.779778
-----
--------------
====================================
block # 1
sender: 30819f300d06092a864886f70d01010105000381800308189028181100b15cbb6869c70c4631243847b6c3d7fb776238775dcf87fa1c70015a955b3
d9b7eb59470f58ce12c0ad5e84c9da327e80ecd2d9452e50c2de9019ff6e6a4c08d61e74458f0b2717bd92b4d50af624234816cddd3a5fc9aacff291e8dbfed
a831c322ca86ef85a589d0cc94627849bdd21d385afd58e75769741343c8074bed6f0203010001
-----
recipient: 30819f300d06092a864886f70d01010105000381800308189028181008d9bf49f340baa5c90013e39eedc77d6cf59e75361d1b181cd99d3a139
bc61573905ae0c9a5cc1b2ba2233b5e606e2967b65f6d85eb87035ae6137d8b5fa1f854342e933ec4c236345fea03055904380e6c6bb00395c874135dfdef5d
d5a7ba37d1f37287bd47df2673bcec42c89ad1f476b2d4cb14784502e4f5c5f444df3c30203010001
-----
value: 15.0
-----
time: 2020-03-31 20:32:14.291155
-----
```