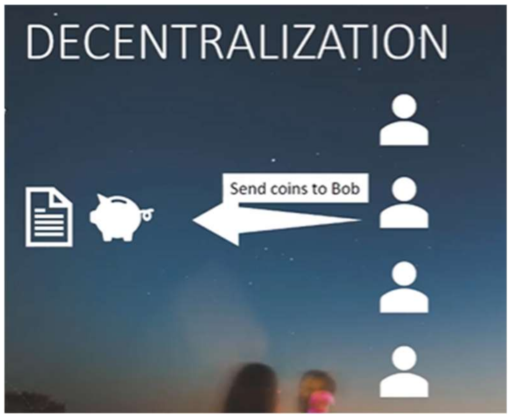
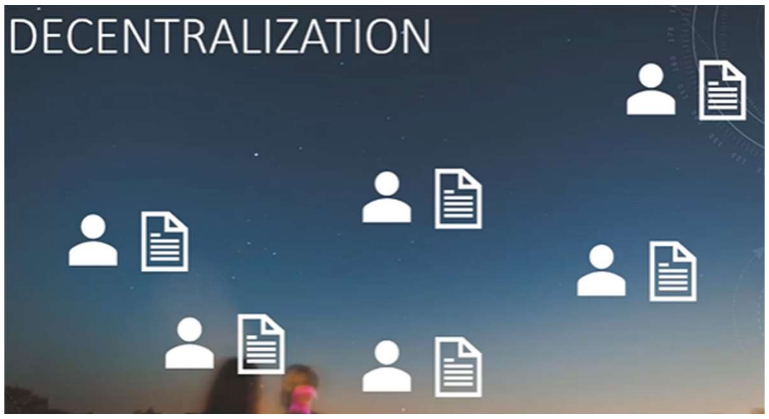
Vid 35 – Decentralization

At this point in our code → transactions originate, and blocks are added at a single point.

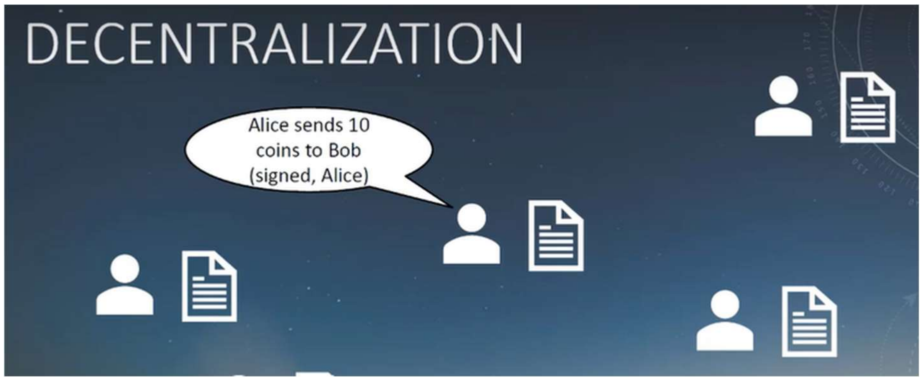
A bank can certainly use a blockchain to store its transactions. The bank would keep a blockchain ledger and users will be free to ask for their balances and to add new transactions.



A good decentralized crypto is one for which every user can have their own copy of the blockchain. They can also publish transactions signed with their public key and assemble transactions into a block and add them to the block.

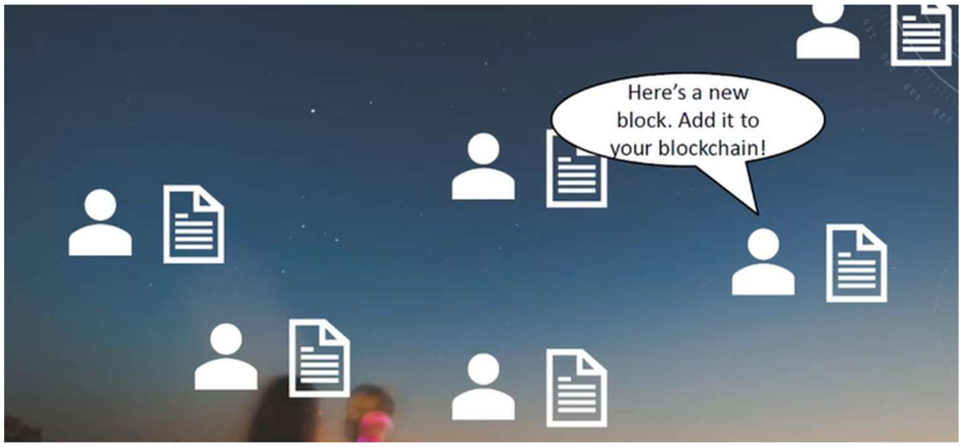


Our ecosystem will look like this. To begin were going to assume there are many users all w/ their own copies of the blockchain. Meaning that any user can tell how many coins are controlled by any public key at any time.



When a user wishes to spend coins they sign a transaction and publishes it (broadcasts it to the rest of the network)

Also want to decentralize the adding of the new blocks to the block chain. We want anyone to be able to take the transactions she's heard about, assemble them to a block and add them to the block chain.



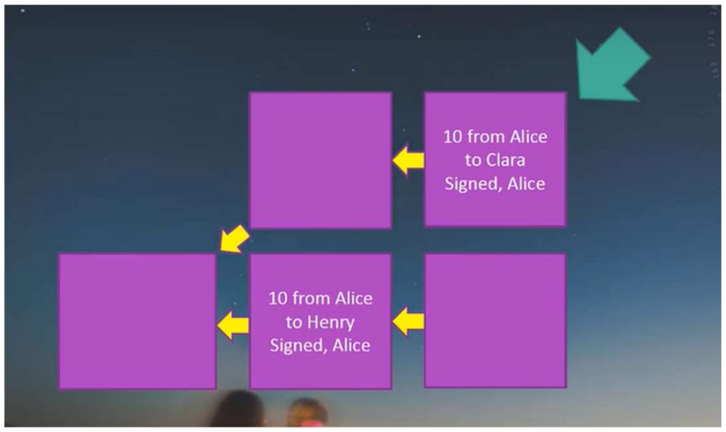
Vid 36 – the dark side of decentralization

Due to the decentralization of the blockchain potential problems arise, such as the realities of moving info through a network, and nefarious actor trying to steal coins or other ways to cheat.

With many miners adding blocks to the network, we need a way to resolve the case in which two blocks have the same parent. If two blocks are added at or near the same time, there may be some confusion as to which block comes next in the block chain.

So, we can have a branch in what would otherwise be a linear blockchain. This branching is also a thing that hackers can take advantage of in attempt to steal coins

Imagine Alice sends coins to Henry and that transaction gets added to the block chain. Someone looking at this last block in the block chain will know that the coins now belong to Henry. But if Alice can prevent ppl from seeing that block, she can try to spend those coins again. For example, if she were to mine another block with the same parent as the block that involved her spend but leaving out her spend, she can keep control of the coins for anyone who sees her new block.

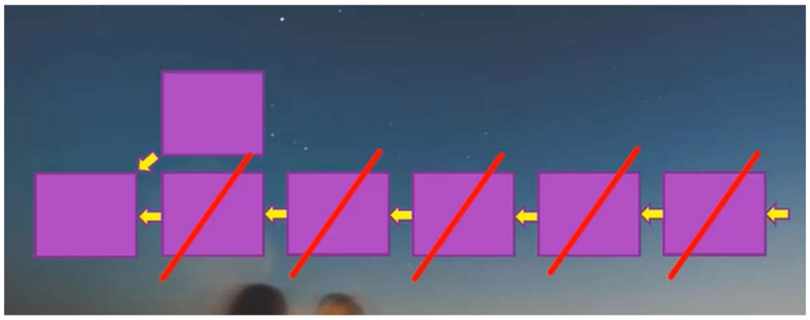

10 from Alice to Clara Signed, Alice
10 from Alice to Henry Signed, Alice

Here she's trying to replace the block that says 10 coins go from Alice to Henry with a Block that says "10 coins go from Alice to Henry" with a block that doesn't include that → later she is going to send those same coins to Clara.

<u>Double Spend Attack</u> → Clara receives goods from both Clara and Henry with the same coins.

Vid 37 – Proof of Work

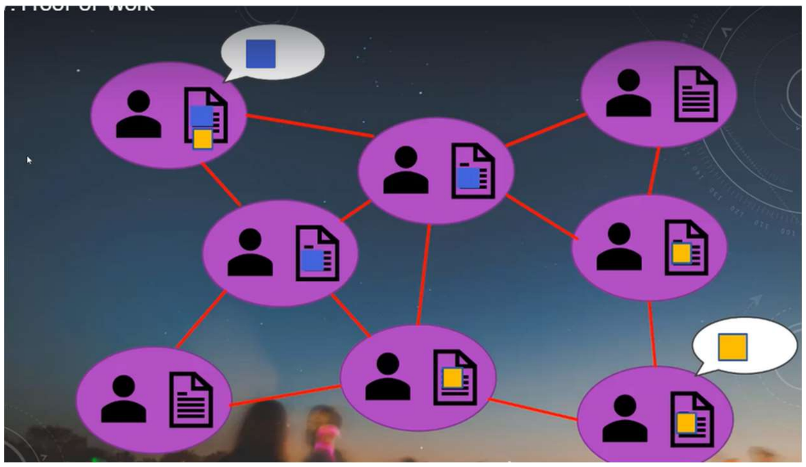
If Alice is very persuasive, she might be able to add a branch to a very old block and erase a significant amount of history.



For this reason, we'll write our code such that the minors only add blocks to the longest valid chain that they see. So, if Alice announces a new block that has a very old parent block, users might store that block, but miners will look for the longest chain and Alice's block will be ignored.

A Branch in the block chain is not always a sign of an attack on our network. Sometimes two blocks are added very close to the same time. As new blocks are passed thru the network they may arrive in a different order at different nodes.

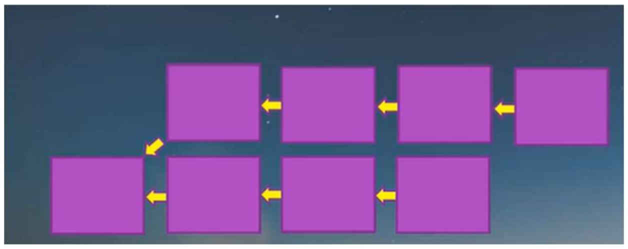
If most miners are added to the longest chain, branches are not impossible, but will likely be short and short lived.



To keep the blockchains code secure and make it trustworthy, it must be open sourced, so anyone can see the code and see if it abides by all regulations. Also means that anyone can modify it to disobey the requirement to add to the longest chain.

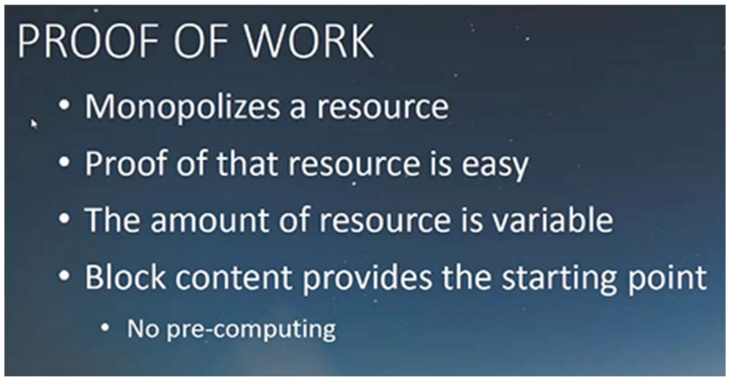
If bad actors were committed to eliminating a certain block, they could try to add blocks onto an alternative chain until that chain becomes the longest. This should be a problem unless a major portion of the blocks are added by attackers.

Cheap to set up many copies of modified nefarious software and overwhelm the network with billions of new blocks added to some branch chain.



One approach to avoid this is to have a network of nodes that you **trust and only add blocks from the nodes that you know are following the rules.** It is difficult to reliably establish trust in an anonymous and decentralized network

The Byzantine Generals Problem → the problem of arriving at a consensus with the possibility of some attacking nodes.



PROOF OF WORK

- Monopolizes a resource
- Proof of that resource is easy
- The amount of resource is variable
- Block content provides the starting point
  - No pre-computing

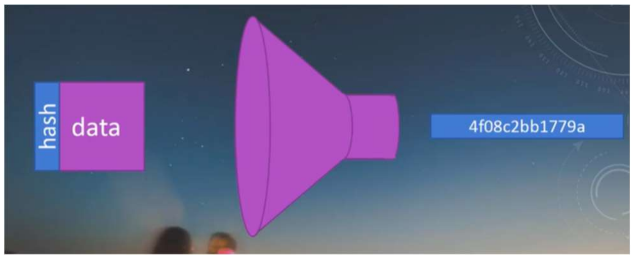
Proof-of-Work: Under this scheme, to add a block to the block chain, the user is required to have done some computational work. Possible to set up many nodes but to add many blocks, requires a lot of computation.

Characteristics of a good Proof-of-Work Algorithm:

1. **Must monopolize a real-world resource for some amount of time**
   - Algorithms require many int or floating-point operations indicating that a user has monopolized the compute units of their computer
   - Other algorithms are memory heavy → setting up a problem that requires an amount of system memory to be occupied or a lot of data to be moved in and out of memory while doing a much lighter weight computation
2. **Proof must be easily verifiable**
   - Node submitting new block should do a lot of work, but it should require very little work from all the other nodes to determine that the work has been done
   - b/c we don't want to repeat all the work that the miner did to add the block
3. **The amount of resource required for each block needs to be variable over time**
   - For EX: at the beginning of bitcoin the work required for a new block could be done in a few hours on your home computer. Today, it takes several thousand nodes of the worlds fastest supercomputer to add a block in anything less than a day.
     - B/c of the large increase in the # of ppl mining
4. **The work needs to take the new block as its starting point**
   - If the work is not connected to a specific block, an attacker can just wait for a new block to be published, then copy the proof of work from that block and attach it to his own.
   - If the work isn't connected at any specific block, how do you know which block is the real one?
   - **Pre-Computing the work →** if the work were unconnected to the block, you can do a lot of pre-computation and have the proof of work done for thousands of blocks ahead of time, and then publish them all at once to overwhelm the network. NO POSSIBILITY OF PRE-COMPUTATION!!
     - Do this by making the block content the starting point for the computation

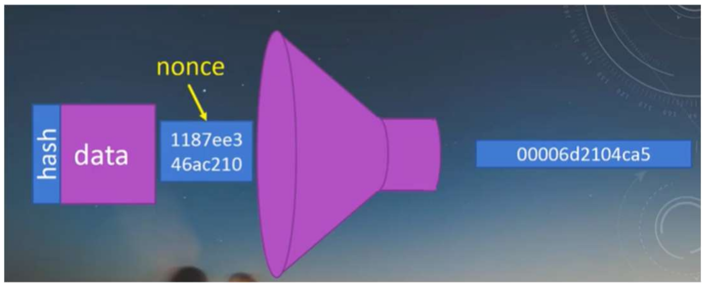
Why it helps to have a variable amount of work that you need to do to add each block?

Vid 38 – Nonces



Hash takes a string of any arbitrary length and produces a hash of one specified length, this is what a hash of some data might look like.

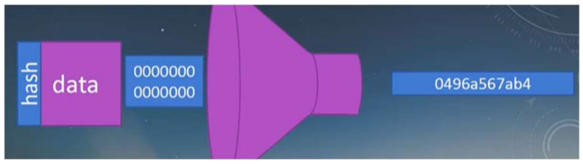
The Algorithm used by Bitcoin is called a cryptographic nonce

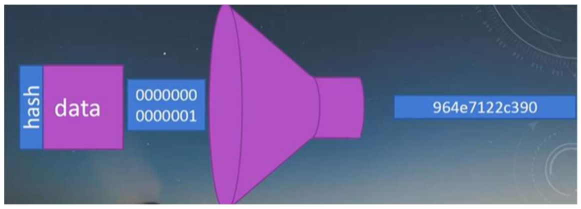


The proof-of-work algorithm used by bitcoin and others involves adding some bits to the end of a new block. They can be anything, but for the block to be valid, the hash of the block plus the bits must have a number of leading zeroes. These are called the "nonce" and this is the mining process.

As before the miner must verify that the transactions and the hash of the previous block are valid.

Miners spend most resources on trying different nonces to see which one will give leading zeroes in the hash. Once the miner is sure that the transactions in the block are valid, he then tries a potential nonce and hashes the block + nonce. (Minimal work is done for transactions)



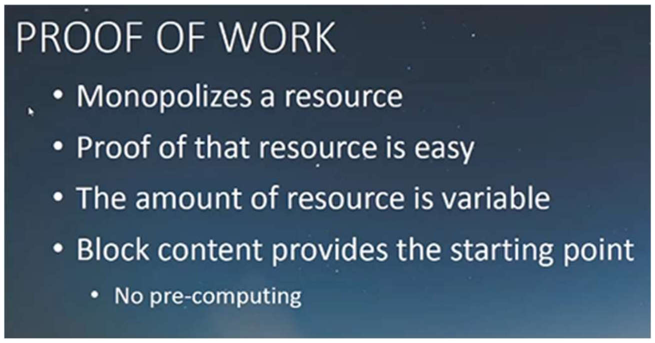


If the hash has the specified # of leading zeroes, he's done.

More likely this potential nonce doesn't produce enough leading zeroes. So, the miner then adds a different nonce, hashes it, and looks for leading zeroes again. Miner keeps trying nonces until it sees a hash w/ enough leading zeroes to be valid.

Once found → it'll publish that new block, including the nonce it's found, to the block chain

A miner usually tries trillions of nonces before finding one w/ enough leading zeroes.



PROOF OF WORK

- Monopolizes a resource
- Proof of that resource is easy
- The amount of resource is variable
- Block content provides the starting point
  - No pre-computing

Comparing algorithm w/ list of requirements:

- ✓ **Monopolizes a resource for a period of time (if hash is non-malleable)**
  - Non-malleable: 2 similar inputs hash to very dissimilar outputs.
    - If any changes made to input, we'll have no info about the output.
    - Only thing can do is guess a new nonce, hash and check it
  - w/ a malleable hash might be possible to find a shortcut where the hash w/ 1 nonce gives a clue as to which to try next.
- ✓ **Easy to Verify**
  - Miner has to do trillions of hash computations looking for the right nonce but when a user is reading the block chain, he can verify the nonce w/ just **1 hash computation**
- ✓ **W/ nonce finding, we can vary the amount of work required by the changing # of leading zeroes that make a block valid.**
  - Understand why necessary later
- ✓ **Block Content is a starting point for nonce finding**
  - Because were hashing the block + the nonce altogether not just the nonce itself.

When mining in practice → important to constantly check for new transactions and blocks. If two miners are working to add a new block, they might have many of the same transactions in their new block. But what if one successfully adds a new block while the other is still working?

The remaining **miners must make 2 changes →**

1. Update the previous block hash of their new block so that they are adding to the end of the chain.
2. When miners hear of a new block, need to remove any transactions that were included in the new block.
   - Including the same transactions in 2 different blocks results in an invalid block.

Why is the variable “hardness” needed in algorithm?

To convince these miners to do this work, buy the equipment, and spend all the energy they needed, there need to be an offer **REWARDED** each time they add a new block. Specifically, we allow miner to deposit a specified # of newly created coins to whatever address the successful miner chooses.

For our coins → allow our miners to add transactions w/ no input and output 25 coins.

- Miner will likely choose to deposit to own account

Letting miners create new coins, raises the risk of creating too many coins, and making # of coins in circulation unpredictable. (Like other currencies, the value of coins is reliant on there being a limited #)

To avoid this → crypto currencies aim to create coins at approximately a constant rate.

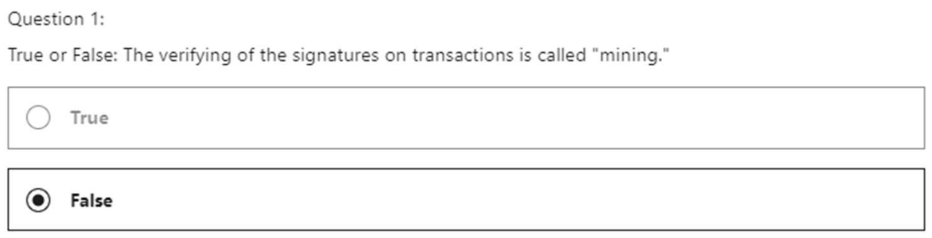
- Bitcoin produces 25 coins about every 10 mins. This is the emission rate
- it is kept relatively constant by adjusting the # of leading zeros required for a valid block.

If coins are created too quickly → it usually means more miners are devoting more resources to mining. So, all the nodes in the network know to increase the required # of leading zeros in the hash and slow the emission rate again.

On the other hand, **if blocks are coming too slowly →** the miners will decrease the # of leading zeros required to lift the emission rate back up.

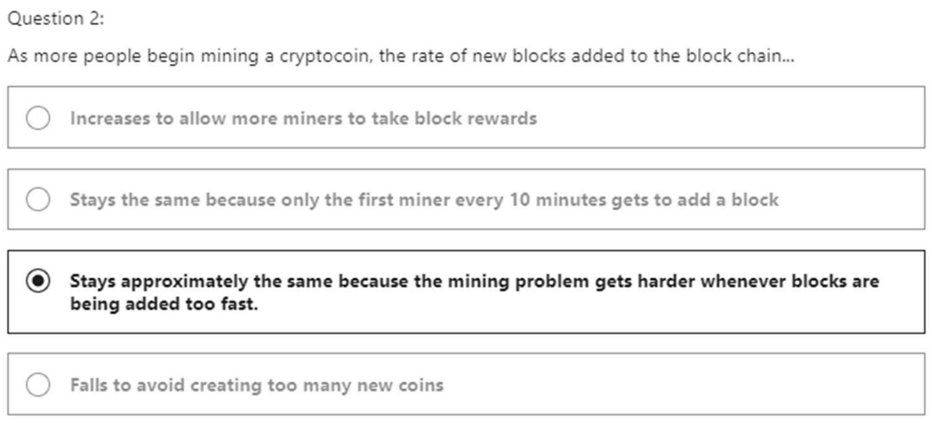
- EX: if mining resources are being taken out of coin, maybe into another coin, maybe they’re doing some other kind of work

Quiz #4:


Question 1:

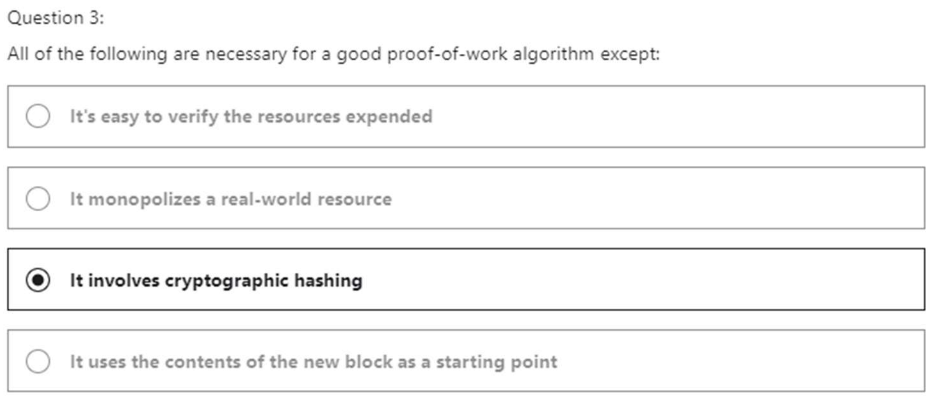
True or False: The verifying of the signatures on transactions is called "mining."

- True
- [✓ False]


Question 2:

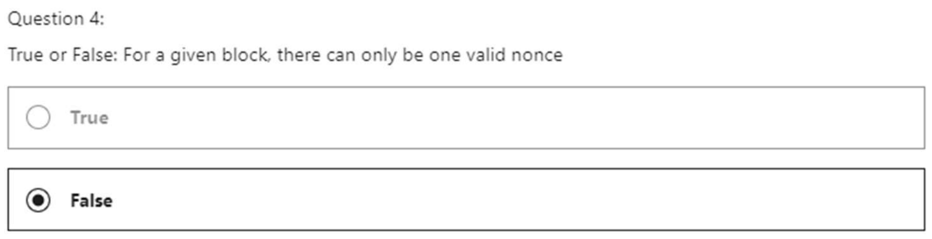
As more people begin mining a cryptocoin, the rate of new blocks added to the block chain...

- Increases to allow more miners to take block rewards
- Stays the same because only the first miner every 10 minutes gets to add a block
- [✓ Stays approximately the same because the mining problem gets harder whenever blocks are being added too fast.]
- Falls to avoid creating too many new coins


Question 3:

All of the following are necessary for a good proof-of-work algorithm except:

- It's easy to verify the resources expended
- It monopolizes a real-world resource
- [✓ It involves cryptographic hashing]
- It uses the contents of the new block as a starting point


Question 4:

True or False: For a given block, there can only be one valid nonce

- True
- [✓ False]

Vid 39 – Assignment 1: Mining Rewards

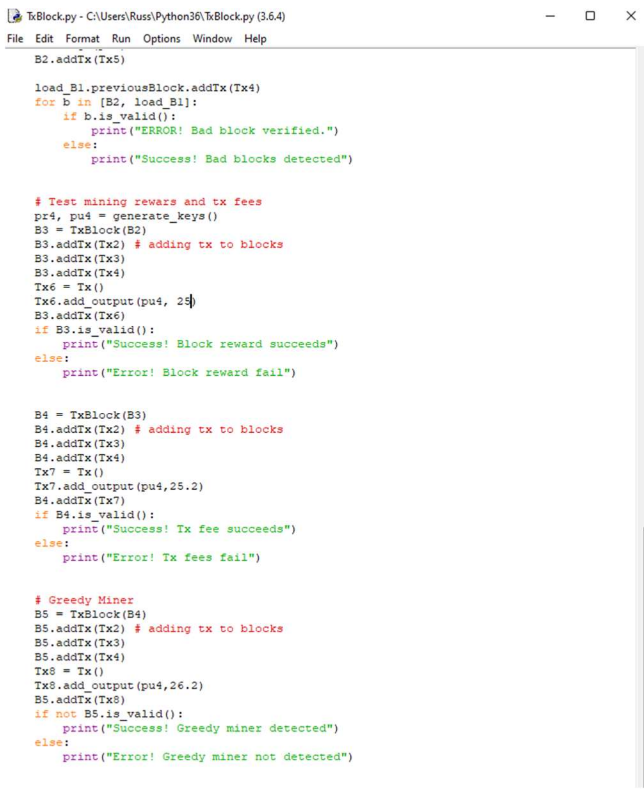
TxBlock → created a bunch or private and public key pairs, created some transactions, then checked those transactions for validity.

To allow miners to take rewards, gonna have to relax some of the requirements we have for valid transactions and for a valid block.

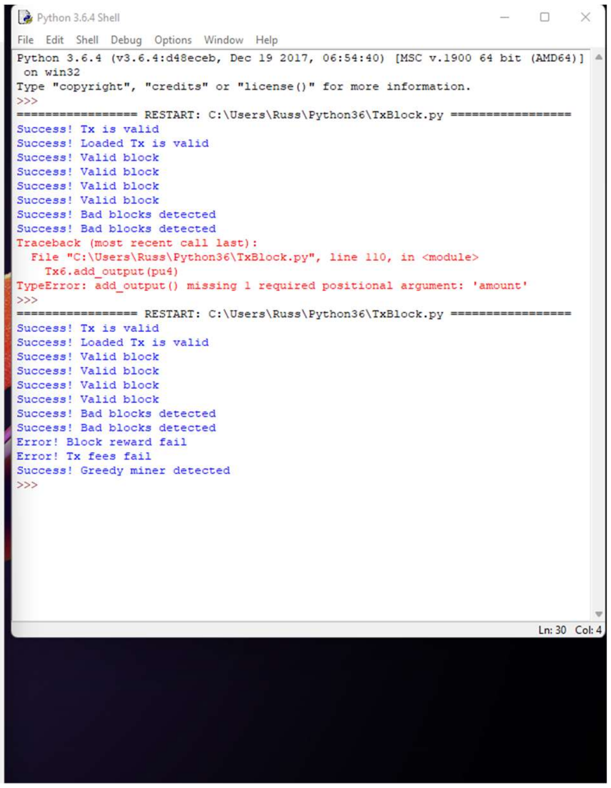
- EX: we specify that a tx always must have more or the same # of inputs that it has outputs. We have to relax that b/c we need to allow the miner to say, "this is my block, I get to add the reward to my own wallet (public key)."

Transactions coming in signed and gonna assume we're a miner and we see these transactions coming over the network. We shouldn't see miner using the private key at any point. He sees the tx, its already signed, he adds them to the block and then says I'm gonna add my own tx to collect my reward. Tx is gonna be flagged as invalid, according to our requirements, b/c it has more output than input

Block reward fails, tx fee failed bc already invalid tx's and the greedy miner was detected bc they were always invalid any block reward he would have taken would have been invalid, so the fact that its too high, we haven't done it right yet


TxBlock.py - C:\Users\Russ\Python36\TxBlock.py (3.6.4)
File Edit Format Run Options Window Help

```
    B2.addTx(Tx5)

    load_B1.previousBlock.addTx(Tx4)
    for b in [B2, load_B1]:
        if b.is_valid():
            print("ERROR! Bad block verified.")
        else:
            print("Success! Bad blocks detected")


    # Test mining rewars and tx fees
    pr4, pu4 = generate_keys()
    B3 = TxBlock(B2)
    B3.addTx(Tx2) # adding tx to blocks
    B3.addTx(Tx3)
    B3.addTx(Tx4)
    Tx6 = Tx()
    Tx6.add_output(pu4, 25)
    B3.addTx(Tx6)
    if B3.is_valid():
        print("Success! Block reward succeeds")
    else:
        print("Error! Block reward fail")


    B4 = TxBlock(B3)
    B4.addTx(Tx2) # adding tx to blocks
    B4.addTx(Tx3)
    B4.addTx(Tx4)
    Tx7 = Tx()
    Tx7.add_output(pu4,25.2)
    B4.addTx(Tx7)
    if B4.is_valid():
        print("Success! Tx fee succeeds")
    else:
        print("Error! Tx fees fail")


    # Greedy Miner
    B5 = TxBlock(B4)
    B5.addTx(Tx2) # adding tx to blocks
    B5.addTx(Tx3)
    B5.addTx(Tx4)
    Tx8 = Tx()
    Tx8.add_output(pu4,26.2)
    B5.addTx(Tx8)
    if not B5.is_valid():
        print("Success! Greedy miner detected")
    else:
        print("Error! Greedy miner not detected")
```

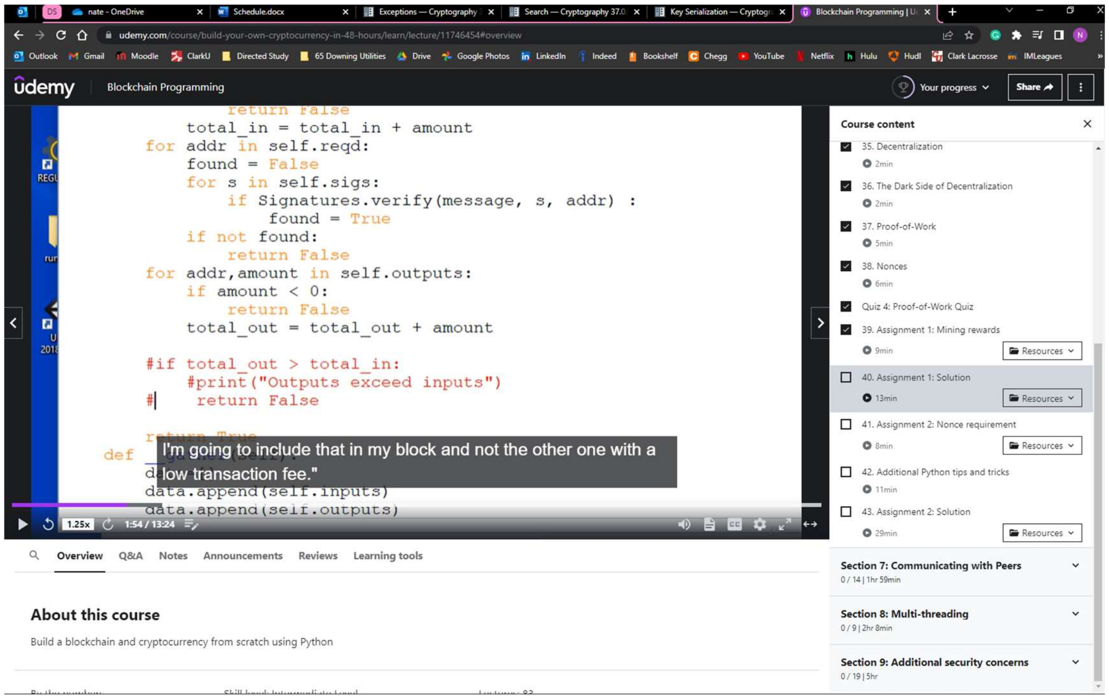


```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit (AMD64)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>>
================= RESTART: C:\Users\Russ\Python36\TxBlock.py =================
Success! Tx is valid
Success! Loaded Tx is valid
Success! Valid block
Success! Valid block
Success! Valid block
Success! Valid block
Success! Bad blocks detected
Success! Bad blocks detected
Traceback (most recent call last):
  File "C:\Users\Russ\Python36\TxBlock.py", line 110, in <module>
    Tx6.add_output(pu4)
TypeError: add_output() missing 1 required positional argument: 'amount'
>>>
================= RESTART: C:\Users\Russ\Python36\TxBlock.py =================
Success! Tx is valid
Success! Loaded Tx is valid
Success! Valid block
Success! Valid block
Success! Valid block
Success! Valid block
Success! Bad blocks detected
Success! Bad blocks detected
Error! Block reward fail
Error! Tx fees fail
Success! Greedy miner detected
>>>
```

This is a check that as we try to fix and allow the miner to take these block rewards, we don't want the greedy miner to slip through.

Vid 40 – Assignment 1 solution

For many coins when trying to make a tx it helps to have a higher tx fee that you're offering to the miners b/c miners are looking for the most return on their investment.



```
            return False
        total_in = total_in + amount
    for addr in self.reqd:
        found = False
        for s in self.sigs:
            if Signatures.verify(message, s, addr) :
                found = True
        if not found:
            return False
    for addr,amount in self.outputs:
        if amount < 0:
            return False
        total_out = total_out + amount

    #if total_out > total_in:
        #print("Outputs exceed inputs")
    #|      return False

    return True
def
    d
    data.append(self.inputs)
    data.append(self.outputs)
```

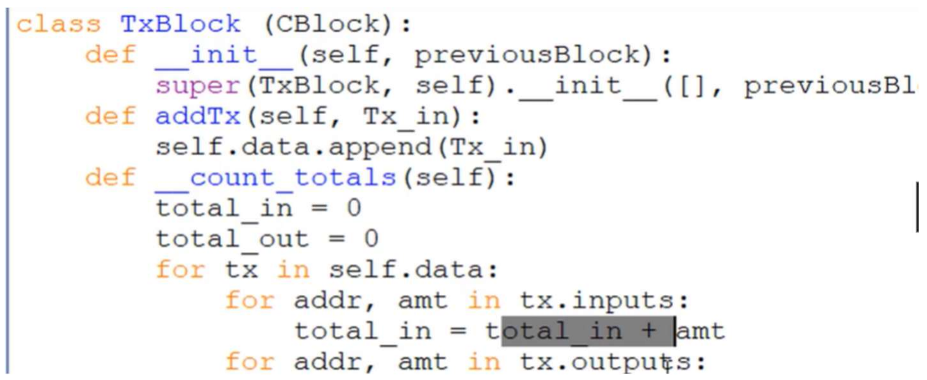
"I'm going to include that in my block and not the other one with a low transaction fee."

Course content
- ☑ 35. Decentralization — 2min
- ☑ 36. The Dark Side of Decentralization — 2min
- ☑ 37. Proof-of-Work — 5min
- ☑ 38. Nonces — 6min
- ☑ Quiz 4: Proof-of-Work Quiz
- ☑ 39. Assignment 1: Mining rewards — 9min
- ☐ 40. Assignment 1: Solution — 13min
- ☐ 41. Assignment 2: Nonce requirement — 8min
- ☐ 42. Additional Python tips and tricks — 11min
- ☐ 43. Assignment 2: Solution — 29min

Section 7: Communicating with Peers — 0 / 14 | 1hr 59min
Section 8: Multi-threading — 0 / 9 | 2hr 8min
Section 9: Additional security concerns — 0 / 19 | 5hr

About this course
Build a blockchain and cryptocurrency from scratch using Python

for every tx we pull out the address and amount out of the inputs and add that to the total in

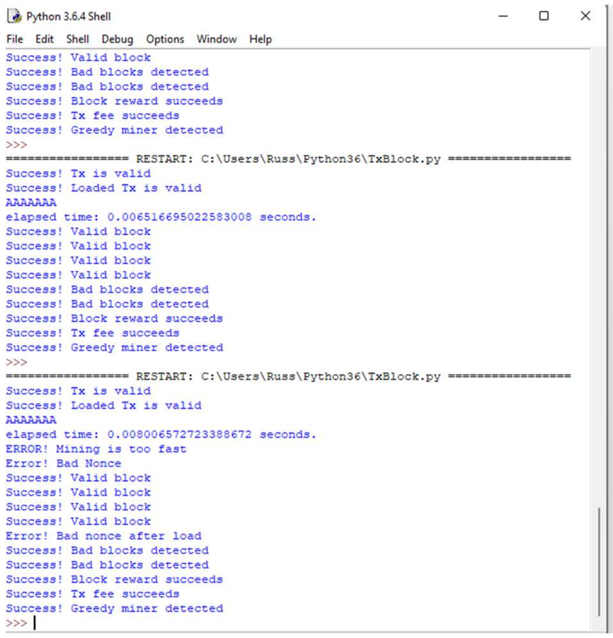


```
class TxBlock (CBlock):
    def __init__(self, previousBlock):
        super(TxBlock, self).__init__([], previousBl
    def addTx(self, Tx_in):
        self.data.append(Tx_in)
    def __count_totals(self):
        total_in = 0
        total_out = 0
        for tx in self.data:
            for addr, amt in tx.inputs:
                total_in = total_in + amt
            for addr, amt in tx.outputs:
```

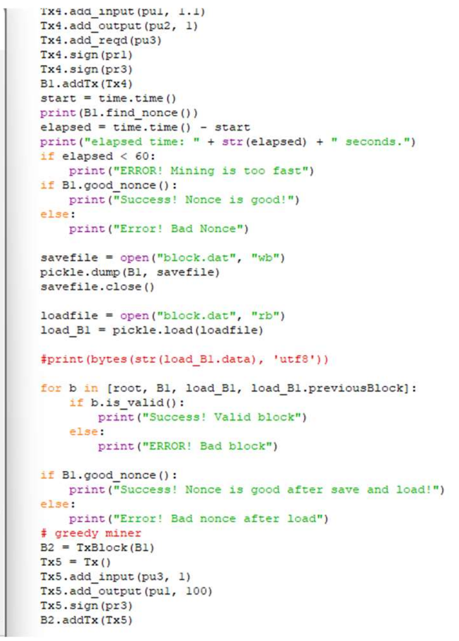
Vid 41 – Assignment 2

Add the requirement to the blocks that the nonce produce leading zeros.

Going to look at how varying the # of leading zeros affects the time that is required to mine a new block.



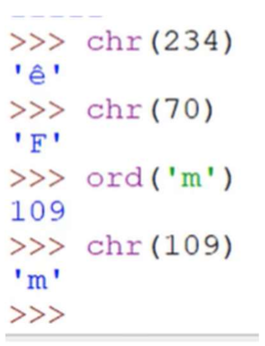
```
Success! Valid block
Success! Bad blocks detected
Success! Bad blocks detected
Success! Block reward succeeds
Success! Tx fee succeeds
Success! Greedy miner detected
>>>
================= RESTART: C:\Users\Russ\Python36\TxBlock.py =================
Success! Tx is valid
Success! Loaded Tx is valid
AAAAAAA
elapsed time: 0.006516695022583008 seconds.
Success! Valid block
Success! Valid block
Success! Valid block
Success! Valid block
Success! Bad blocks detected
Success! Bad blocks detected
Success! Block reward succeeds
Success! Tx fee succeeds
Success! Greedy miner detected
>>>
================= RESTART: C:\Users\Russ\Python36\TxBlock.py =================
Success! Tx is valid
Success! Loaded Tx is valid
AAAAAAA
elapsed time: 0.008006572723388672 seconds.
ERROR! Mining is too fast
Error! Bad Nonce
Success! Valid block
Success! Valid block
Success! Valid block
Success! Valid block
Error! Bad nonce after load
Success! Bad blocks detected
Success! Bad blocks detected
Success! Block reward succeeds
Success! Tx fee succeeds
Success! Greedy miner detected
>>>
```



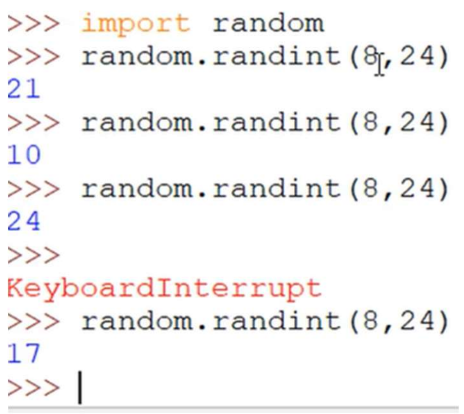
```python
Tx4.add_input(pu1, 1.1)
Tx4.add_output(pu2, 1)
Tx4.add_reqd(pu3)
Tx4.sign(pr1)
Tx4.sign(pr3)
B1.addTx(Tx4)
start = time.time()
print(B1.find_nonce())
elapsed = time.time() - start
print("elapsed time: " + str(elapsed) + " seconds.")
if elapsed < 60:
    print("ERROR! Mining is too fast")
if B1.good_nonce():
    print("Success! Nonce is good!")
else:
    print("Error! Bad Nonce")

savefile = open("block.dat", "wb")
pickle.dump(B1, savefile)
savefile.close()

loadfile = open("block.dat", "rb")
load_B1 = pickle.load(loadfile)

#print(bytes(str(load_B1.data), 'utf8'))

for b in [root, B1, load_B1, load_B1.previousBlock]:
    if b.is_valid():
        print("Success! Valid block")
    else:
        print("ERROR! Bad block")

if B1.good_nonce():
    print("Success! Nonce is good after save and load!")
else:
    print("Error! Bad nonce after load")
# greedy miner
B2 = TxBlock(B1)
Tx5 = Tx()
Tx5.add_input(pu3, 1)
Tx5.add_output(pu1, 100)
Tx5.sign(pr3)
B2.addTx(Tx5)
```

Vid 42 – Python tips and tricks

Discusses in-line lists and randint

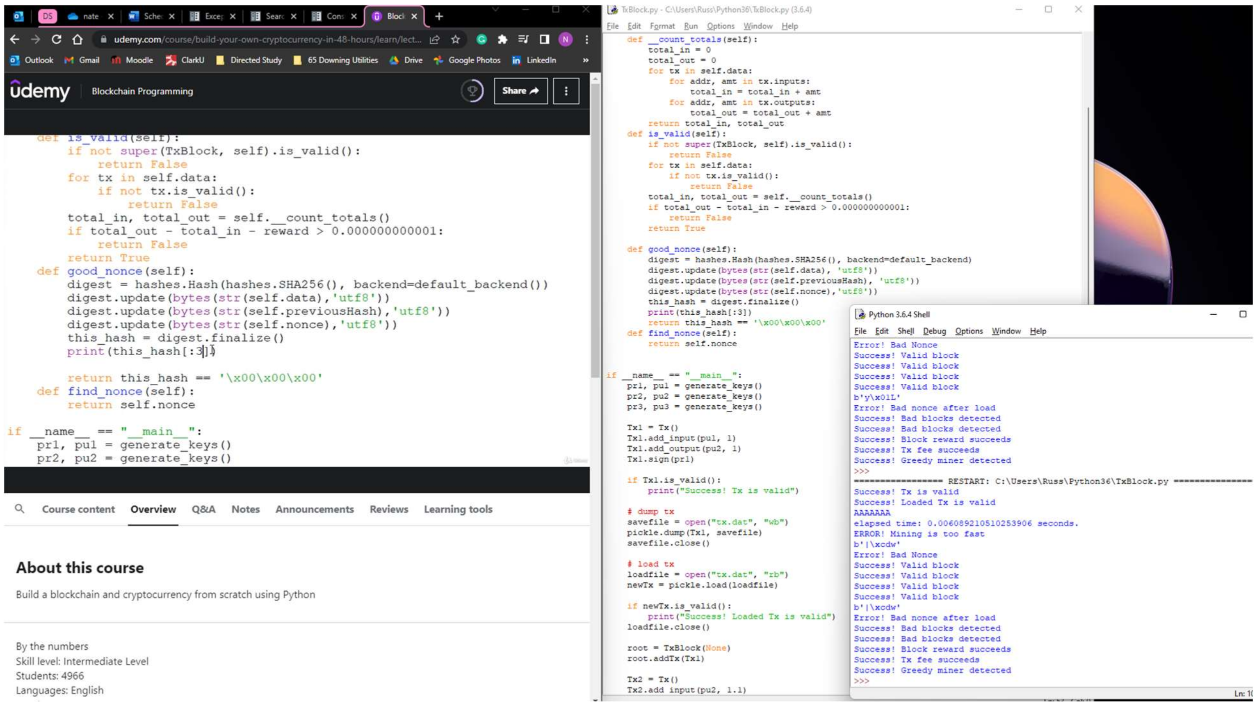


```
>>> chr(234)
'ê'
>>> chr(70)
'F'
>>> ord('m')
109
>>> chr(109)
'm'
>>>
```



```
>>> import random
>>> random.randint(8,24)
21
>>> random.randint(8,24)
10
>>> random.randint(8,24)
24
>>>
KeyboardInterrupt
>>> random.randint(8,24)
17
>>>
```

Vid 43 – Solution 2


udemy.com/course/build-your-own-cryptocurrency-in-48-hours/learn/lect...
Outlook
Gmail
Moodle
ClarkU
Directed Study
65 Downing Utilities
Drive
Google Photos
LinkedIn
Blockchain Programming
Share
Course content
Overview
Q&A
Notes
Announcements
Reviews
Learning tools
TxBlock.py - C:\Users\Russ\Python36\TxBlock.py (3.6.4)
File Edit Format Run Options Window Help
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help

```
    def is_valid(self):
        if not super(TxBlock, self).is_valid():
            return False
        for tx in self.data:
            if not tx.is_valid():
                return False
        total_in, total_out = self.__count_totals()
        if total_out - total_in - reward > 0.000000000001:
            return False
        return True
    def good_nonce(self):
        digest = hashes.Hash(hashes.SHA256(), backend=default_backend())
        digest.update(bytes(str(self.data),'utf8'))
        digest.update(bytes(str(self.previousHash),'utf8'))
        digest.update(bytes(str(self.nonce),'utf8'))
        this_hash = digest.finalize()
        print(this_hash[:3])

        return this_hash == '\x00\x00\x00'
    def find_nonce(self):
        return self.nonce

if __name__ == "__main__":
    pr1, pu1 = generate_keys()
    pr2, pu2 = generate_keys()
```

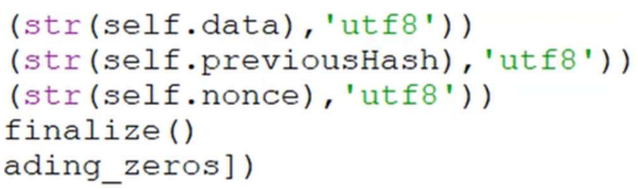
About this course

Build a blockchain and cryptocurrency from scratch using Python

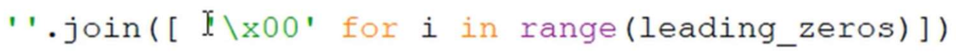
By the numbers
Skill level: Intermediate Level
Students: 4966
Languages: English

```
    def __count_totals(self):
        total_in = 0
        total_out = 0
        for tx in self.data:
            for addr, amt in tx.inputs:
                total_in = total_in + amt
            for addr, amt in tx.outputs:
                total_out = total_out + amt
        return total_in, total_out
    def is_valid(self):
        if not super(TxBlock, self).is_valid():
            return False
        for tx in self.data:
            if not tx.is_valid():
                return False
        total_in, total_out = self.__count_totals()
        if total_out - total_in - reward > 0.000000000001:
            return False
        return True

    def good_nonce(self):
        digest = hashes.Hash(hashes.SHA256(), backend=default_backend)
        digest.update(bytes(str(self.data), 'utf8'))
        digest.update(bytes(str(self.previousHash), 'utf8'))
        digest.update(bytes(str(self.nonce),'utf8'))
        this_hash = digest.finalize()
        print(this_hash[:3])
        return this_hash == '\x00\x00\x00'
    def find_nonce(self):
        return self.nonce


if __name__ == "__main__":
    pr1, pu1 = generate_keys()
    pr2, pu2 = generate_keys()
    pr3, pu3 = generate_keys()

    Tx1 = Tx()
    Tx1.add_input(pu1, 1)
    Tx1.add_output(pu2, 1)
    Tx1.sign(pr1)

    if Tx1.is_valid():
        print("Success! Tx is valid")

    # dump tx
    savefile = open("tx.dat", "wb")
    pickle.dump(Tx1, savefile)
    savefile.close()

    # load tx
    loadfile = open("tx.dat", "rb")
    newTx = pickle.load(loadfile)

    if newTx.is_valid():
        print("Success! Loaded Tx is valid")
    loadfile.close()

    root = TxBlock(None)
    root.addTx(Tx1)

    Tx2 = Tx()
    Tx2.add_input(pu2, 1.1)
```

```
Error! Bad Nonce
Success! Valid block
Success! Valid block
Success! Valid block
Success! Valid block
b'y\x01L'
Error! Bad nonce after load
Success! Bad blocks detected
Success! Bad blocks detected
Success! Block reward succeeds
Success! Tx fee succeeds
Success! Greedy miner detected
>>>
================= RESTART: C:\Users\Russ\Python36\TxBlock.py ===============
Success! Tx is valid
Success! Loaded Tx is valid
AAAAAAA
elapsed time: 0.006089210510253906 seconds.
ERROR! Mining is too fast
b'|\xcdw'
Error! Bad Nonce
Success! Valid block
Success! Valid block
Success! Valid block
Success! Valid block
b'|\xcdw'
Error! Bad nonce after load
Success! Bad blocks detected
Success! Bad blocks detected
Success! Block reward succeeds
Success! Tx fee succeeds
Success! Greedy miner detected
>>>
```



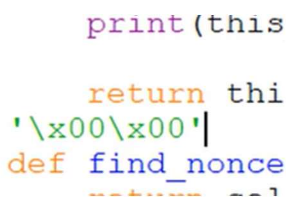
```
(str(self.data),'utf8'))
(str(self.previousHash),'utf8'))
(str(self.nonce),'utf8'))
finalize()
ading_zeros])
```



```
''.join([ '\x00' for i in range(leading_zeros)])
```

First were gonna make a list, in this list there is gonna be 1 zero hexadecimal character for of our leading zeros. So if leading_zeros is 2 were gonna get 2 of these → \x00 and then were gonna join them w/out a separator

should give us →



```
    print(this

    return thi
'\x00\x00'
def find_nonce
```